A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks

Istvan Simon
California State University, Hayward
Hayward, CA 94542
simon@csuhayward.edu
http://www.mcs.csuhayward.edu/~simon/security/boflo.html
January, 31 2001

Abstract

For the past several years Buffer Overflow attacks have been the main method of compromising a computing system's security. Many of these attacks have been devastatingly effective, allowing the attacker to attain administrator privileges on the attacked system. We review the anatomy of these attacks and the reasons why conventional methods of defense have been ineffective, and likely to remain so in the foreseeable future. Recently, however, several promising methods of defense have been proposed. We compare the strengths and weaknesses of these defense methods.

1. Introduction

Buffer Overflows have been successfully used as a method of penetrating systems' security for over 12 years. One of the first buffer overflow attacks which attracted widespread attention due to its spectacular success was Robert Morris's Internet Worm. In 1988 Morris released a program which succeeded in infecting thousands of Unix hosts on the Internet. One of the methods Morris used to gain access to a vulnerable system was a buffer overflow bug in the *fingerd* daemon [9, 29]. Once it gained access to a vulnerable system, Morris's program installed itself on the machine, and used several methods to attempt to spread itself to other machines. The original intent of Morris was to spread to other systems relatively slowly and undetected, without causing a significant disruption on any of the affected machines. However, his attack failed completely in this. Morris made a programming error which caused his worm to spread at a much higher rate than originally intended. Because of this error, machines were infected and reinfected so rapidly that the worm ended up overwhelming the attacked systems. Of course this caused his program to be detected immediately, and transformed it into the most devastating denial of service attack until that time. Morris's program usually did not gain administrative root access, and did not destroy any information on the penetrated system, nor leave time bombs or other malicious code behind [9].

From 1988 to 1996 the number of buffer overflow attacks remained relatively low [2, 30]. The known vulnerabilities were fixed, and because the attack method was little known and thought to be difficult to execute few new vulnerabilities were discovered. This changed dramatically in 1996 when Levy published a very well written paper [17] which simultaneously showed that it was very likely that many programs harbored buffer overflow vulnerabilities, and also demonstrated techniques of constructing buffer overflow attacks which were likely to succeed against a target program suspected of being vulnerable, even if the attacker had no access to the actual source code of the target program. The combination of these two factors stimulated attackers to a flurry of research activity which lead to many discoveries of new vulnerabilities. In addition, many of the attacks were automated, which permitted the attack to be carried out even by people with little or no knowledge. People who are relatively unsophisticated but interested in such attacks are often called *Script Kiddies*. Unfortunately, there are far too many script kiddies, who seem to have plenty of time on their hands, and also the energy, patience and persistence to keep hacking systems this way. The unhappy result is that these automated attacks have become a serious nuisance to the overworked system administrators responsible for maintaining the integrity of their systems under continuous attack.

In this paper we review the anatomy of buffer overflow attacks and discuss the reasons why they are so prevalent and deadly. We then examine the traditional methods of defense and show why they are unlikely to succeed in stopping these attacks effectively in the near future. We also discuss more recent approaches of defense, and discuss their relative weaknesses and strengths. While no method of defense is 100% effective, we show why these newer methods are more likely succeed than their predecessors.

2. What is a Buffer Overflow Attack?

A buffer overflow occurs in a program when the program stores more information in an array, the *buffer*, than the space reserved for it. This causes the areas adjacent to the buffer to be overwritten, corrupting the values previously stored there. Buffer overflows are always programming errors which are typically introduced into a program because the programmer failed to anticipate that the information copied into the buffer by the program may exceed its size. Unfortunately, as we shall soon see, buffer overflow programming errors are quite common because of certain widely used and dangerous C programming practices. Once a buffer overflow vulnerability is present in a program inadequate testing may not uncover it, so that the vulnerability may lurk in the program hidden , undiscovered and silent for years. This potentially opens up the program to be the target of a sudden attack which exploits the vulnerability to gain unauthorized access to a system.

A buffer overflow may happen accidentally during the execution of a program. When this happens, however, it is very unlikely that it will lead to a security compromise of the system. Most often the clobbering of information in areas adjacent to the buffer will cause the program to crash or produce obviously incorrect results. In a buffer overflow attack, on the other hand, the objective of the attacker is to use the vulnerability to corrupt information in a carefully designed way in order to execute *attack*

code previously planted by the attacker. If this succeeds, the attacker effectively hijacked the control of the program. Once control is transferred to the attack code, it grants unauthorized access to the attacker. Typically the attack code just spawns a shell, which allows the attacker to execute arbitrary commands on the system.

When a new shell is spawned in a Unix system, it inherits the access privileges of the process that spawned it. Consequently, if the attacked process containing the buffer overflow vulnerability runs with root privileges, the attacker will also get a root shell. We remark that though we limited our discussion to UNIX systems, buffer overflows are applicable to most Operating Systems . In particular, many attacks have been successful against Windows NT and Windows 2000 systems [8, 12, 13, 21, 22, 26]. Axelsson [1] compared the security of Windows NT and UNIX systems against known types of attack, and found them to be roughly equally vulnerable.

A buffer overflow attack may be local or remote. In a local attack the attacker already has access to the system and may be interested in escalating his/her access privilege. A remote attack is delivered through a network port, and may achieve simultaneously both gaining unauthorized access and maximum access privilege.

Summarizing , we see that a buffer overflow attack usually consists of three parts:

1. The planting of the attack code into the target program;

2. The actual copying into the buffer which overflows it and corrupts adjacent data structures;

3. The hijacking of control to execute the attack code;

We now examine in more detail the main type of buffer overflow attacks.

3. Smashing the Stack

One classification of buffer overflow attacks depends on where the buffer is allocated. If the buffer is a local variable of a function, the buffer resides on the run-time stack. This is the type of attack examined in Levy's article [17], and it is by far the most prevalent form of buffer overflow attack.

When a function is called in a C program, before the execution jumps to the actual code of the called function , the *activation record* of the function must be pushed on the run-time stack. In a C program the activation record consists of the following fields:

1. space allocated for each parameter of the function;
2. the *return address;*

3. the *dynamic link;*
4. space allocated to each local variable of the function.

For convenience we will consider the address of the dynamic link field to be the base address of the activation record. The function must be able to access it's parameters and local variables. This requires that during the execution of the function a register hold the base address of the activation record of the function., i.e. the address of the dynamic link field. Parameters are below this address on the stack, and local variables above. When the function returns, this register must be restored to its previous value, to point to the activation record of the calling function. To be able to do this, when the function is called the value of this register is saved in the dynamic link field. Thus the dynamic link field of each activation record points to the dynamic link field of the previous activation record on the stack, which in turn points to the dynamic link field of the previous activation record, and so on, all the way to the bottom of the stack. The first activation record on the stack is that of *main*(). This chain of pointers is called the *dynamic chain.*

In many C compilers the buffer grows towards the bottom of the stack. Thus if the buffer overflows and the overflow is long enough the return address will be corrupted, (as well as everything else in between, including the dynamic link.) If the return address is overwritten by the buffer overflow so as to point to the attack code, this will be executed when the function returns. Thus, in this type of attack, the return address on the stack is used to hijack the control of the program.

Overwriting the return address, as explained above, gives the attacker the means of hijacking the control of the program, but where should the attack code be stored? Most commonly it is stored in the buffer itself. Thus the *payload* string which is copied into the buffer will contain both the binary machine language attack code as well as the address of this code which will overwrite the return address.

There are a few difficulties that the attacker must overcome to carry out this plan. If the attacker has the source code of the attacked program it may be possible to determine exactly how big the buffer is and how far it is from the return address, determining how big the payload string must be. Also, the payload string cannot contain the null character since this would abort the copying of the payload into the buffer. Some copying routines of the C library use carriage returns and new lines as a delimiter instead, so these characters should also be similarly avoided in the payload string.

Access to the source code is nowadays quite common for many Operating Systems, e.g. Linux, OpenBSD, Free BSD, and even Solaris. Levy [17] shows, however, that there is no need to have access to the source, or even knowledge of the exact details of how the attacked program works. The address of the attack code can be guessed, and through various techniques an approximate guess will do. For example, the attack code could start with a long list of *no operation* instructions, so that control could be passed to any of these in order to correctly execute the crucial part of the attack code which spawns the shell and comes after the no ops. This technique was already used in the Morris worm. Similarly, the tail of the payload string could consist of a repeated list of the guessed address of the attack code that we want to overwrite the return address with. These techniques increase considerably

the chances of guessing the address of the attack code close enough for the attack to work. For more details check Levy's article [17].

We now examine why buffer overflows are so common. Suppose that the buffer is a character array used to store strings. Most programs have string inputs or environment variables which can be used by the attacker to deliver the attack. The program must read this input and parse it in order to make the appropriate response to the input. Often, to parse the input,the program will first copy it into a local variable of a function and then parse it. To do this the programmer reserves a large enough buffer for any *reasonable* input. To copy the input into the buffer the program will typically use a string copying function of the standard C library such as *strcpy*(). If done carelessly, this introduces a buffer overflow vulnerability. This pattern is so well established in the C programmer's repertoire that it makes very likely that many programs will contain buffer overflow vulnerabilities.

The problem arises partly because C represents strings in a dangerous way. The length of a string is determined by terminating the sequence of characters by a null character. This representation is convenient, because strings can have arbitrary length and yet it allows for efficient processing of strings. But at the same time it is also dangerous, because the scheme breaks down if a string is not null terminated, and because there is no way of knowing the length of the string prior to processing all its characters. The typical C culture emphasizes efficiency over correctness, prudence or safety, which compounds the problem. It would require a massive amount of education to change this well entrenched programming practice. A consequence of this is that it is unlikely that buffer overflow vulnerabilities can be eradicated at the source by not introducing them into a program in the first place. Not only it will be difficult to eliminate the vulnerability from the enormous quantity of software already deployed, but it seems likely that programmers will continue to write new vulnerable software.

Miller [20] studied the behavior of UNIX utilities when given random input in many distributions, both commercial and open source. His study is important and relevant to our discussion, because while unexpected input is not necessarily directly related to buffer overflows, the inability of programs to handle unexpected input comes from the same tendency of programmers to concentrate only on *reasonable* input that leads also to buffer overflow flaws. Attackers are not reasonable. On the contrary, they wish to exploit this blind spot of programmers for unreasonableness, to find a hole in the program's logic that they can use for their own purposes. So Miller's study provides some evidence on how common buffer overflow problems are likely to be. Unfortunately, in almost all distributions more than half of the utilities crashed under Miller's experiment.

Miller also gives us some insight into the speed with which vendors are making progress in improving the quality of their software, if at all, because he repeated the study five years later. Indeed, his results show that progress is being made. But progress has been very modest.

Another interesting result of Miller is the confirmation of the widely held anecdotal belief that Open Source provides significantly higher quality software than commercial offerings. This seems to suggest the power of somewhat chaotic large-scale parallelism over better organization of small-scale

parallelism. The former is prevalent in the Open Source model, in which many pairs of eyes scrutinize the software but relatively uncoordinated. The latter is characteristic of commercial organizations, with fewer pairs of eyes scrutinizing the software but in a much more systematic and organized fashion.

4. Traditional defenses

We now examine some traditional defenses against buffer overflow vulnerabilities such as the ones discussed in the last section. We already mentioned the first and most obvious of these which is eliminating the error from the target program. We have seen briefly at the end of the last section that unfortunately this approach is unlikely to succeed. Here we elaborate on further obstacles to this defense.

First, there is the magnitude of the problem. To eliminate the bug a very large number of programs must be examined. The number of potential targets already deployed is very large. There are some tools that one can use to automate the search for the vulnerability [11, 12, 30]. For example, a very simple scheme would be to search for the use of the unsafe functions in the C library, which like *strcpy*() have been identified, and replace them with safe functions which takes the size of the buffer into account, like *strncpy*(). Still, manual auditing of the code must be used for each program which makes this a massive and very expensive approach. This is not to say that this work should not be undertaken, and indeed there are efforts under way to systematically audit the code of at least two free versions of Unix , OpenBSD and Linux [19, 23]. In the case of the former this effort seems to have already achieved considerable success, accounting for the reputation of OpenBSD among the security community as being the most secure Unix distribution currently available. One wonders why similar efforts are not under way by the commercial vendors of Operating Systems, which one would suppose could better afford the cost. While the value of such systematic auditing of code has been successfully demonstrated, the approach is not guaranteed to produce buffer-overflow-free code. Some buffer overflows have been found even in already audited code [10].

Not surprisingly, most installations must rely on the vendor to provide them with reliable code. Even if the source code is available, they must deploy code which they can't hope to fully understand themselves. Unfortunately this reliance on the vendor seems misplaced in many cases, as vulnerabilities seem to be all too common. with most vendors. This rather discouraging state of affairs is very frustrating, yet seems to be the main approach traditionally recommended. Security specialists [22, 26] recommend that the administrator of a system follow closely the release of security patches by the vendor, so that as soon as they are released they can install them. This presumably makes their systems more secure. However, this approach has serious shortcomings. The first problem is that it is costly in terms of the administrator's time and effort. Many systems are administered not by professional system administrators but by people whose primary job is something else. For these systems this approach is simply too impractical and untenable. The cure is worse than the disease. Thus the high cost of this method of defense guarantees that many systems will fail to install the patches in a timely manner, which in turn provides attackers with plenty of vulnerable systems, even for

vulnerabilities which have already been fixed. Furthermore, as we remarked in the last section, programmers keep introducing new vulnerabilities with every new release of the operating system.

5. Recent defenses

Recently new defenses have been discovered that are more promising than the traditional approaches discussed above. We examine three methods and discuss their strengths and weaknesses. One of the attractive features of all these three methods is that they are all relatively low cost measures that can be easily implemented by any system administrator independently of the vendor and they are all effective to some degree against buffer overflow vulnerabilities not yet discovered. So one of the common characteristics of these three methods is that they offer valuable protection with current code which is vulnerable. The other most significant advantage of these methods is that they are proactive methods of defense rather than the reactive methods discussed in the previous section. They allow a significant measure of protection without forcing the administrator to have to wait for the vendor to do something to secure his system.

5.1 Disabling Stack Execution

Several vendors now offer this method of defense. [7, 18] Most systems do not need code to be ever executed on the stack. Since the most common buffer overflows, as seen in Section 3, rely on code to be injected into the buffer and then executed, a simple solution is the option to install the operating system with stack execution disabled. The idea is simple, inexpensive to install, and relatively effective against the current crop of attacks.

There are some serious weaknesses to this approach. First, though rare, some programs do rely on the stack to be executable. More importantly, the defense is weak. Though the code in the current crop of stack based buffer overflows is often stored into the buffer, a little reflection will immediately reveal that this is not really essential. The attacker does not care where the attack code is. All the attacker needs is that this code be *somewhere* in memory and that it's address or approximate address be known to the attacker so he can overflow the return address with it to hijack control. We think that it is only a matter of time before a new crop of buffer overflow attacks will appear that do not store the code on the stack and which will become immune to this defense. Wojtczuk explores methods to bypass the non executable stack defense in [31]

5.2 Safer C library support

A much more robust alternative would be if we could provide a safe version to the C library functions on which the attack relies to overwrite the return address. This idea seems to have occurred independently

to several people. Alexander Snarskii seems to have been the first one to think of it [28]. He implemented it for the FreeBSD version of Unix and offered it to the development group of FreeBSD. His explanation of the method was unfortunately a little obscure, and either he may not have fully realized the true power of his method, or if he did, he certainly did not elaborate on it in his note. Thus Snarskii's idea had less impact than it should have had. Baratloo, Tsai , and Singh from Bell Labs independently rediscovered the idea , and wrote a much more substantial white paper about it [2]. This author also rediscovered this defense independently. The Bell Labs group implemented the vulnerable functions in a library called *LibSafe*, which can be freely downloaded from their site.

Can we replace a vulnerable function in the C library by a safer version? We will discuss the idea in terms of *strcpy()*, but it will become readily apparent that the method generalizes to any of the other vulnerable string manipulation functions. At first sight a safer version of *strcpy*() appears impossible because *strcpy*() does not know the size of the buffer that it is copying into. So complete avoidance of overflowing the buffer is not possible. Nonetheless, *strcpy*() has access to the dynamic chain on the stack, and successive dynamic links are like bright markers delimiting the activation records of all the currently active functions. The idea is to use this information to prevent *strcpy*() from corrupting the return address or the dynamic link fields.

Using these markers and the address of the buffer itself *strcpy*() can first determine which activation record contains the buffer, or else that the buffer is not on the stack at all. To do this *strcpy*() finds the interval [*a*,*b*] of consecutive dynamic links which contains the buffer. The cases in which the buffer is either below the first activation record on the stack, or above the last activation record can be handled as special cases with appropriate values of either *a* or *b*. Once the values of *a* and *b* are determined, we can compute an upper bound on the size of buffer. For example, if the buffer grows towards the bottom of the stack then |*buffer -a* | is an upper bound on the size of the buffer. This can be used by *strcpy* () to limit the length of the copied string so that neither the dynamic link nor the return address are overwritten. Furthermore, *strcpy*() can detect an attempt to do so, report the problem to syslog, and safely terminate the application.

LibSafe does not replace the standard C library. The method relies instead on the loader searching *LibSafe* before the standard C library, so that the safe functions are used instead of the standard library functions. This scheme is more flexible than replacing the functions in the C library itself. For example, it is possible to have one program use the C library functions and another use the *LibSafe* versions. By setting appropriate environment variables *LibSafe* can be installed as the default library. But from a security perspective, there seems to be little reason to keep the vulnerable functions installed on the system, so the usefulness of this extra flexibility is somewhat questionable.

This defense has several advantages. It is effective against all buffer overflow attacks that attempt to smash the stack in which the target program uses one of the vulnerable C library functions to copy into the buffer. The method does not totally prevent buffer overflows. It can't, because it does not know the true size of the buffer. It is still possible to overflow areas between the buffer and the dynamic link. But the critical return address and the dynamic link fields are protected from being overwritten.

The method fails to provide any protection against heap based buffer overflow attacks (see below), or attacks which do not need to hijack control by overwriting the return address. Both of these kinds of attack, however, are much harder to pull off, and consequently much rarer. The method would also fail to protect a program that does not use the standard C library functions to copy into the buffer. For example, if the target program contains custom code to copy the string into the buffer it will not be protected. However, it seems clear that few programs will have such custom code. Generally speaking it is considered to be bad programming practice to "reinvent the wheel", so programmers are encouraged to use the standard libraries.

Though programs that rely on custom code may contain buffer overflow vulnerabilities just as much as those that use the standard C library, they will be less likely to be detected. Because of this they will enjoy some immunity from attack. This is security through obscurity, which in general is not a good way to secure a system. Nonetheless it is of some security value.

The overhead of the safe functions is negligible, and the cost of installing the library and configure the system to use it is very low. Another advantage is that it works with the binaries of the target program, and does not require access to their source code. Finally, it can be deployed without having to wait for the vendor to react to security threats, which is a very desirable feature. It is a much more robust defense than disabling stack execution. Though we have discussed variants of attacks against which it will offer no protection, it is very effective against the class of attacks that it is designed for, and it cannot be easily circumvented. The attacker has no way of interfering with the detection of the buffer overflow attack, because this occurs before the attacker has a chance to hijack control. We conclude that overall, this defense offers a very significant improvement of the security of a system at very low cost. In our opinion it is a sure winner.

We also mention Andrey Kolishak's BOWall protection [16]. This is available for Windows NT systems, with full source. This solution has some similarities to both the safer Library approach, and to the methods to be presented in the next Section.

Kolishak's approach is similar to the others in this Section, because it works by replacing the DLL's that contain the vulnerable library functions with a safer library version. However, unlike *LibSafe* or Snarskii's method, it seems to be a buffer overflow detection system, which is more similar to the methods of the next Section. It works by saving the return address when the function enters, and checking it before actually returning. If corruption of the return address is detected it does not return, so hijacking of control is prevented. Kolishak also has a second component of BOWall which relies on some specific Windows NT security features.

5.3 Compiler Techniques

Range checking of indices is a defense that is 100% effective against buffer overflow attacks. For

example, buffer overflow attacks are impossible in a Java program, because Java automatically checks that an array index is within the proper bounds. Unfortunately, full-blown range checking in C is impossible, because of the dichotomy between arrays and pointers. Some compilers will offer protection if the array is accessed with an indexing operation, like in the expression *buffer*[*i*] but not in an expression like *buffer + i* . When the compiler compiles a function like *strcpy*(char* *dest* , char* *src*) the two arguments are just pointers, and it is impossible for the compiler to know the lengths of the corresponding arrays. So the compiler cannot generate code to do range checking inside of the function. Jones and Kelly implemented some range checking techniques in C [14, 15].

C programmers do not always appreciate range checking because of the associated overhead, but this excessive preoccupation with performance is often only justified in the most demanding applications. Snappy performance is always a desirable feature, but for most applications it is much less of a critical issue than programmers tend to assume. For example, the calendar manager daemon in Solaris has been shown to be vulnerable to a buffer overflow attack which compromises root, yet there seems to be no reason why such an application could not have been written in Java, which would have rendered the attack impossible. We believe that performance would not have been a critical issue in this case. Some security flaws have been uncovered in Java and quickly fixed. These flaws did not invalidate Java's security model, which appears to be sound, but usually were implementation problems of the Java Virtual Machine, which of course is just another C program, so subject to the same programmer errors as any other program [6].

Cowan, Wagle, Pu, Beattie and Walpole [4] devised a fresh approach to the problem. Their method does not prevent the corruption of the return address and dynamic link, but instead prevents the hijacking of control by detecting that an attack took place before the control is hijacked. Just as was the case with the safe library approach, the key idea is simple and elegant. It is based on the assumption that if a buffer overflow attack took place then everything between the buffer and the return address is likely to be corrupted. They propose to modify the compiler so that it protects the critical return address and dynamic link part of the activation record by allocating an extra field aptly called the *canary* after the dynamic link and before the local variables in the activation record. When the activation record is pushed on the stack a value is stored in the canary field. Before the function returns the integrity of the canary is checked. If it was corrupted the canary sings and the attack is detected. (Equivalently, another possible metaphor, is to say that the attack makes the canary die. This is analogous to the use of canaries in mines.) In this case, the program is gracefully terminated with an *syslog* error message alerting about the buffer overflow thus thwarting the attack. These ideas were implemented in the *StackGuard* project and used in some experiments to protect an entire Linux distribution recompiled with this technique.

To avoid the possibility of the attacker forging the value stored in the canary they propose either storing terminator symbols, like the null character, carriage return , line feed, and eof, whose inclusion in the payload string would stop the copying operation by the various vulnerable functions of the C library, or to choose a random canary value, chosen independently each time the program is started, which would make its forging very difficult for the attacker.

"Bulba" and "Kil3r" explored ways that *StackGuard* could be circumvented [3]. For example, they reasoned that if a pointer variable were between the buffer and the return address, then a first buffer overflow could corrupt this pointer variable, without corrupting anything else, so as to make it point to the return address field. A second copying operation then could overwrite the return address without corrupting the canary. This would defeat the *StackGuard* protection by avoiding the basic assumption that *everything* between the buffer and the return address must have been corrupted by a buffer overflow. To deal with such an attack Cowan proposed an even better choice to be stored in the canary, namely he proposed to store a value that depended both on a random value and the correct return address. Specifically, he proposed to compute the XOR of these two values. This countermeasure was easy to implement, and it would detect the attack even if the canary was not changed.

"Bulba" and "Kil3r" demonstrate that their techniques work with examples of vulnerable code. But in our opinion these attacks are somewhat optimistic about the conditions that must be present in the target program for the attack to work, so that their techniques currently seem more a proof of concept than a serious and immediate threat to defeat the *StackGuard* defense. It is quite possible that such target programs exist and are deployed widely enough for their techniques to become a serious breach of the defense, but that has not been demonstrated yet. Of course, one must be always vigilant when dealing with security, and the prudent approach is always to assume that no defense is foolproof.

The performance overhead of *StackGuard* is worse than that of the *LibSafe* defense, in part because *StackGuard* imposes an overhead on every function called, but better than the overhead of range checking which incurs a small extra cost on every array access. In any event, this overhead is still small. *StackGuard* is effective even against custom code, since *StackGuard* is a buffer overflow detection method, so it does not care how the buffer overflow happened. However we noted that custom code attacks seem to be less likely than those that rely on the standard library functions. On the other hand, assuming that an administrator has access to the modified compiler, the cost of protection is much larger than that of the *LibSafe* approach, because it requires recompilation of every target program to be protected. This also means that one has to have access to the source code of the target program, or put another way, *StackGuard* cannot protect a program for which we have no source code, whereas *LibSafe* can.

In some ways the three methods discussed in this section are complementary, so they can be applied independently and simultaneously. By doing so the robustness against future attacks circumventing the defenses is also enhanced. Given the very low cost of deployment and overhead of the first two methods, and moderate cost of deployment and low overhead for the last one, deploying these methods should be recommended.

6. Buffer Overflow Variants

We mentioned earlier that the Morris worm used several methods to try to infect a machine, one of which was essentially a stack smashing attack on *fingerd*. This part of the worm sent a TCP/IP packet to port 79, the finger port . The packet consisted of 400 VAX *nop* instructions, followed by code that *exec*'d a shell, followed by a part that would overflow the return address to point to this code. The worm also attacked Sun machines with a similarly constructed packet, but there was an error in the part that was supposed to overwrite the return address in the Sun version, which caused it to fail, even though the Suns were also vulnerable. *fingerd* parsed its input by reading it into a local buffer of 512 bytes with *gets*(). The above packet had 536 bytes, so this caused it to overflow and corrupt the return address. Once the control was hijacked, the shell would read its input from the network connection and write it's output to the network connection. The client side that had sent this packet would then send over a C program of the worm, which was compiled and then run on the newly infected machine [9].

Though the *fingerd* attack was a straightforward stack smashing attack, exactly as described in Section 3, initially it found few imitators. Perhaps it was not realized for a while how widespread the vulnerability was in other targets as well. Another factor may have been that the worm was fairly complex, and the *fingerd* attack was just one of several methods used by it. Because of this it may have been understood by a relatively small number of people. It was not until the publication of Levy's paper in Phrack that the hacker community seem to have realized that many programs were likely to harbor a stack smashing vulnerability.

Once the stack smashing attack was well understood variants started to appear. We discuss some of these variants in this section. Looking at the general steps of a buffer overflow attack that we discussed in Section 2, we may see the possible variants. In some programs there is no need for Step 1 (planting the attack code) because the code might be already present in the target program. In such cases Step 2 (overflowing the buffer) might seek to corrupt not the return address but another resource. For example, the string that the code would *exec*. In this case the hijacking of control works differently.

In the stack smashing examples, Step 1 (planting the attack code) might be accomplished in several ways: through an input to the target program, or an environment variable, or a network port on which the target program listens (as in *fingerd*, basically just another form of input). The buffer overflow of Step 2 copies the planted code into the buffer and overwrites the return address on the stack. The hijacking of control occurs when the function returns to the wrong return address, and executes the attack code instead.

While the stack is convenient for Steps 2 and 3 of the attack, we may look at other ways that a program may hijack control not involving the stack. These will then lead to a different class of potential attacks. Any structure in which function pointers are stored, or addresses to which the program will jump are potential targets for Step 3. If these structures can be corrupted by a buffer overflow, we have a

potential new technique for an attack. If the pattern necessary to carry this out is also sufficiently widespread we have another dangerous variant.

For example, heap based attacks are possible in C++ targets[24]. Each object has a virtual function table where each entry points to the corresponding member function of the object. By corrupting the virtual function table of an object , control can be hijacked when any of the object's operations is invoked. Instead of executing the object's intended operation, the attack code will be executed. Every object-oriented program will have this pattern, so in theory this looks quite promising. But the difficulty here is finding an application in which an adjacent object has a buffer that can be overflown. The order in which objects are allocated on the heap depends on the particular run-time conditions present, so might be frequently difficult or impossible to predict. Consequently, the chances of a successful attack through heap based buffer overflows that corrupt the virtual function table are much more difficult to carry out than the stack smashing attack.

7. Conclusions

We have analyzed the characteristics of several buffer overflow attacks, the reasons for their popularity, and the effectiveness and costs of various defenses against them. Until recently the attackers seemed to have the upper hand, and the traditional defenses seemed largely impotent to stop these attacks. We analyzed the reasons for this. Among the reasons we cited are the cost of deployment of the traditional defenses, the reactive nature of the methods of defense, and the dependence of the average installation on the operating system vendor to provide the solutions to the attacks. The recent appearance of effective defenses that break some of these obstacles give reason for hope that finally the defenders might have a chance to gain the upper hand against this type of attack.

8. References

1. Stefan Axelsson, A Comparison of the Security of Windows NT and UNIX, 1998
   http://www.securityfocus.com/data/library/nt-vs-unix.pdf

2. Arash Baratloo, Timothy Tsai, and Navjot Singh, Libsafe: Protecting Critical Elements of Stacks
   http://www.securityfocus.com/library/2267
   http://www.bell-labs.com/org/11356/libsafe.html

3. Bulba and Kil3r, Bypassing StackGuard and Stackshield, *Phrack Magazine* 56 No 5, 1999.
   http://phrack.infonexus.com/search.phtml?view&article=p56-5

4. Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole, Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, in *DARPA Information Survivability*

Conference and Expo 2000.
http://www.cse.ogi.edu/DISC/projects/immunix/publications.html
http://www.securityfocus.com/library/1674

5. David Curry, Improving the Security of your Unix System, 1990
http://www.securityfocus.com/library/1913

6. Drew Dean, Edward W. Felten, and Dan S. Wallach, Java Security: From HotJava to Netscape and Beyond, in *Proc. of the IEEE Symp. on Security and Privacy, 1996*
http://www.cs.princeton.edu/sip/pub/secure96.html

7. Casper Dik, Non-Executable Stack for Solaris, posted to *comp.security.unix* January 2, 1997.
http://x10.dejanews.com/

8. DilDog, The TAO of Windows Buffer Overflow, 1998
http://www.cultdeadcow.com/cDc_files/cDc-351/

9. Mark W. Eichin and Jon A. Rochlis, With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988, 1988.
http://www.mit.edu:8001/people/eichin/www/virus/main.html

10. Chris Evans, Nasty Security Hole in *lprm*, posted in *BugTraq*, April 18, 1998
http://www.securityfocus.com/archive/1/9023

11. HalVar Flake, Auditing Binaries for Security Vulnerabilities, in *Black Hat Europe Conference*, 2000.
http://www.blackhat.com/html/bh-europe-00/bh-europe-00-speakers.html

12. Nishad Herath, (Joey_) Advanced Windows NT Security, in *Black Hat Asia Conference*, 2000.
http://www.blackhat.com/html/bh-asia-00/bh-europe-00-speakers.html#Joey

13. Barnaby Jack, (dark spyrit), Win32 Buffer Overflows (Location, Exploitation, and Prevention), *Phrack Magazine* 55,(15), 1999.
http://phrack.infonexus.com/search.phtml?view&article=p55-15

14. Richard Jones, Bounds Checking Patches for gcc.
http://web.inter.NL.net/hcc/Haj.Ten.Brugge

15. Richard Jones and Paul Kelly, Bounds Checking for C, 1995
http://www-ala.doc.ic.ac.uk/phjk/BoundsChecking.html

16. Andrey Kolishak, BOWall, (Buffer Overflow Protection for Windows NT), 2000
http://developer.nizhny.ru/bo/eng/BOWall

17. Elias Levy (alpha1), Smashing the Stack for Fun and Profit, *Phrack Magazine* 49 (14), 1996. http://phrack.infonexus.com/search.phtml?view&article=p49-14

18. The Linux OpenWall Project, Nonexecutable Stack Patch for Linux http://www.openwall.com/linux/

19. The Linux Security Audit Project, http://lsap.org/

20. Barton P. Miller, Fuzz Revisited: A Re-examination of the Reliability of Unix Utilities and Services, 1998 http://www.securityfocus.com/library/2087

21. Mudge, How to Write Buffer Overflows, 1997 http://l0pht.com/advisories/buffero.html

22. Ryan Russel, Rain Forest Puppy, Elias Levy, Blue Boar, Dan Kaminsky, Oliver Friedrichs, Riley Eller, Greg Hoglund, Jeremy Rauch, and Georgi Guninski, *Hack Proofing Your Network Internet Tradecraft*, Syngress, 2000.

23. Theo Raadt, OpenBSD Security http://openbsd.org/security.html

24. rix, Smashing C++ VPTRS, *Phrack Magazine* 56 (8), 2000. http://phrack.infonexus.com/search.phtml?view&article=p56-8

25. W. Olin Sibert, Malicious Data and Computer Security, 1996. http://www.securityfocus.com/library/2168

26. Joel Scambray, Stuart McClure, George Kurtz, *Hacking Exposed, Network Security Secrets & Solutions* , Second Edition, Osborne/McGrawHill , 2001

27. Nathan P. Smith, Stack Smashing Vulnerabilities in the UNIX Operating System, 1997. http://millcomm.com/nate/machines/security/stack-smashing/nate-buffer.ps

28. Alexander Snarskii, FreeBSD Stack Integrity Patch, 1997 ftp.lucky.net/pub/unix/local/libc-letter

29. Eugene Spafford, The Internet Worm Program: Analysis, *Computer Communications Review*, 1989

30. David Wagner, Jeffrey S Foster, Eric A. Brewer, and Alexander Aiken, A First Step towards Automated Detection of Buffer Overrun Vulnerabilities, in *Proc. 7th Network and Distributed*

System Security Symposium 2000.
http://www.cs.berkeley.edu/~daw/papers/overruns-ndss00.ps

31. Rafel Wojtczuk, Defeating Solar Designer's Non-Executable Stack Patch, in *Bugtraq*, January 30 1998
http://www.securityfocus.com/templates/archive.pike?list=1&mid=8470&fromthread=0&date=1998-01-30