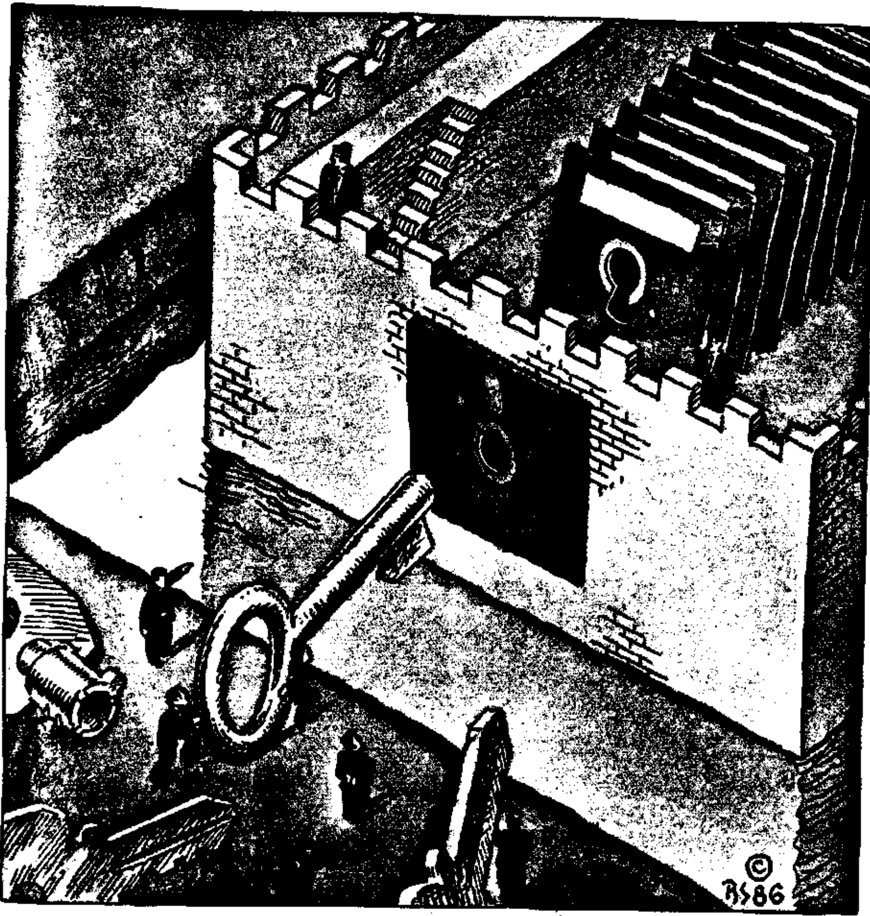
BY ALAN FILIPSKI AND JAMES HANKO

MAKING UNIX SECURE



Countermeasures to known methods of attack

UNIX evolved as a powerful operating system at Bell Laboratories, where it was used primarily by a few research programmers. Much experience in using UNIX also came from universities, particularly the University of California at Berkeley. These environments provided an army of competent and dedicated hackers to test UNIX system security. UNIX operating system security has always been an "open book."

R. Morris and K. Thompson wrote the following in "Password Security: A Case History" (from the programmer's manual for UNIX System III, volume II).

> We did not attempt to hide the security aspects of the operating system, thereby playing the customary make-believe game in which weaknesses of the system are not discussed no matter how apparent. Rather, we advertised the password algorithm and invited attack in the belief that this approach would minimize future trouble. The approach has been successful.

In this article, we examine some specific security-related features of the UNIX operating system and discuss known methods of attack along with countermeasures and their associated costs. We make no claim of completeness; it is axiomatic that someone will always build a better

(continued)

Alan Filipski has a Ph.D. in computer science from Michigan State University. He has taught at Central Michigan University and Arizona State University and is currently a principal staff engineer at Motorola Microsystems, working on UNIX System V.

James Hanko has M.S. and B.S. degrees in computer science from Pennsylvania State University. He works at Edge Computer Corp., Scottsdale, Arizona, and is currently a senior software design engineer working on porting UNIX System V.

Both authors can be reached at Mail Drop DW160, 2900 South Diablo Way, Tempe, AZ 85282.

ILLUSTRATED BY ROB SAUNDERS

mousetrap (or Trojan horse).

We provide this information in a way that, we hope, is interesting and useful yet stops short of being a "cookbook for crackers." We have often intentionally omitted details. Admittedly, we are treading a thin line. However, it is the consensus of most system administrators that wider dissemination of such information is ultimately beneficial to the security of UNIX installations.

Definition of Security

The level of "security" of computer systems can be measured by the amount of difficulty users would have accessing the system data or resources in a way unauthorized by the system administrator. The word "users" here includes legitimate users who modify or read data they are supposed to be excluded from, unauthorized persons who break into the system for the purpose of falsifying data or using system resources, or people who "decrypt" information they are supposed to read only in encrypted form.

We will not discuss protecting the user or administrator from his or her own carelessness or ignorance. If you are troubled by this kind of "insecurity" in a particular environment, you can write shell scripts or new shells to coddle the user to any degree required. Performing regular disk backups also effectively serves to limit the damage incurred in this way.

Physical versus Software Security

There are two approaches to securing a computer system. You can put the system in a windowless electromagnetically shielded room with no data lines leading out and hire a guard to stand at the door and check persons who enter and leave. This type of physical security is very effective. Unauthorized persons are prevented from gaining access to the system and even legitimate users can be stopped from carrying tapes out the door. Of course, this approach greatly reduces the usefulness of the system.

The second approach is to relax some of this physical control and compensate by using software security checks, such as passwords, permissions, log files, and encryption schemes. Although these schemes cannot absolutely prevent clever and determined attacks, especially by insiders, they can provide enough security to foil the casual snoop.

File Permissions and the Superuser

Perhaps the greatest security weakness of the UNIX operating system is the power of the superuser. There is no effective system of checks and balances against him or her.

Within the UNIX operating system, each file has an owner and group associated with it and a set of switch values that control the way it may be accessed by different types of users. By default, the owner of a file is its creator, and the group associated with a file is the group its owner belongs to.

Only the owner of a file, or root (the superuser), can change the file's owner, group, or the values of the file-permission bits. The permissions are useful not only to the system administrator, but also to individual users who want to enhance the privacy of some of their files.

Figure 1 shows the format of the file-permission bits. They are organized as a four-digit octal number. The bits constituting the most significant digit are setuid (set user identification), setgid (set group identification), and the "sticky" bit. The sticky bit causes the system to preserve a core image (of often-used files) after use, to enhance system speed.

The next three octal digits represent the permissions for owner, group, and other users, respectively. The bits within each octal digit represent read, write, and execute/search permission for that group.

For example, a permission of octal 750 = 111 101 000 means that the owner can read, write, or execute the file; members of the group associated with the file can read or execute it; and others can neither read, write, nor execute it. A default set of values for these bits is used when a new file is created. Often this default is 644, meaning that only the file owner can write to and read the file, and everyone else can only read it.

In a work environment where people frequently need to share information, this is reasonable. But if security is a concern, the default should be changed to something more restrictive (such as 640 or even 600) and then weakened for selective files as necessary. UNIX's Bourne shell pro-

(continued)

Figure 1: UNIX *file permissions. Under the* UNIX *operating system, a user's ability to access a particular file is governed by the setting of 12 bits (four octal digits) associated with that file. The high-order 3 bits are the* setuid, setgid, *and "sticky" bits, respectively. The remaining 9 bits represent* read, write, *and* execute/search *permissions for owner, group, and others, respectively.*

Inquiry 172 →

vides an umask function to change this default.

Since directories are a special kind of file within the UNIX operating system, the same 9 permission bits apply to them. However, the execute bit has a special meaning here, and it is called the search bit. The search permission bit in a directory is required to access any file with that directory in its path. This is distinct and independent from read permission in a directory.

The read permission bit allows the contents of a directory (the names of files) to be examined or displayed as, for example, by ls. This distinction provides some additional security. You can let someone execute, read, or write to one of your files without letting him or her list the contents of the directory that the file is in. Therefore, the user can access the file only if he or she knows its name.

Some claim that the three-level (owner, group, other) file-protection system of the UNIX operating system is inferior to the scheme in which an explicit access list of users must be given for each file. Actually, you can simulate this other file-protection scheme on UNIX in several ways.

For example, you can form a group that defines any given subset of users, and you can give the desired permission to the group. More flexibly, a setuid program can be written to check the log-on ID of the user who invokes it and allow access to files based on a list of IDs. The UNIX SCCS (Source Code Control System) utilities do something like this.

Two additional file-protection bits, setuid and setgid, are very important to the security of the UNIX operating system. The setuid bit on an executable program allows it to run with the effective file-accessing power of the program's owner rather than with that of the user (the usual case). Likewise, the setgid bit on an executable program allows it to run with the effective file-accessing power of the group of the program file.

These features, when used with care, can be powerful tools for implementing application software. For instance, you can implement an electronic mail facility wherein a fictitious user group, e.g., "mail," is associated with all files that represent mail in transit. You can prevent unauthorized reading of the undelivered mail by preventing those users not running under the group "mail" from accessing the files.

This presents a dilemma: How does a user not ordinarily in group "mail" send mail to another? The setgid bit resolves this by allowing the program that sends mail to temporarily run under the group "mail" instead of the user's real group.

If not used carefully, this feature can compromise the security of a system. For example, the mkdir or df programs, which are owned by the superuser and have the setuid bit on, might inadvertently be made writable by others. Then any user could copy another program over the original mkdir and do whatever he or she desired, running as the superuser. Afterward, the user could copy the original mkdir back in to (partially) cover his or her tracks.

TRAPDOORS

The goal of nearly every UNIX system break-in scheme is to allow the cracker to become superuser (root) even for a few moments. The cracker creates a setuid shell that he or she can execute. He or she then becomes root whenever he or she wants to.

Administrators take note: A setuid root program in a user directory is an advertisement of a security hole.

If source is available on the system, the cracker can be more sophisticated. To cover his or her tracks the user creates a doctored version of, say, mkdir or df, and installs it in place of the real one. This version behaves exactly like the real one except that when given a special option, such as mkdir –xyzzy, it transforms itself into a shell via an exec system call. Since the original utility was setuid to root, the shell will be setuid root also.

The cracker now becomes superuser whenever he or she wishes, even if the superuser password is changed and the cracker leaves no incriminating programs in his or her working directory. The only trace left is that the doctored utility is now perhaps different in size and its modification date updated. But even these tracks can be covered. A little clever surgery can make the utility the same size, and touch can be used to reset its modification times.

Such a subverted utility is sometimes called a trapdoor because it gives those who know how to use it secret access to the operating system.

UNIX System V provides some features to thwart unauthorized superusers. For example, the superuser can log on only at the system console, which is subject to physical security. Ordinary users at ordinary terminals can become superusers through the su command and the superuser password. The su command enters the real log-on ID of the user into a log file when executed, but this log should not be relied upon as a deterrent because, like any other file, it may be changed by the superuser.

Because superuser privileges are required for many administrative operations, superuser passwords should be guarded with great care, should be changed frequently, and should never be given to anyone unless absolutely necessary.

THE crypt UTILITY

Many versions of the UNIX operating system provide a utility called crypt. This utility implements a single-rotor encryption machine that is a simplified version of the German World War II "Enigma" machine. Using the utility is simple—to encrypt an ASCII file plain to produce an encrypted file cipher, you only have to type

crypt [password] <plain> cipher

where password is a character string that serves as an encryption key. The crypt utility provides its own inverse, so that to decrypt the file, you type

crypt [password] <cipher> plain

using the same string for password.

Recent studies indicate that crypt may not be as secure as previously thought and that the files produced

(continued)

Device-special disk, memory, and terminal files can compromise system security.

will yield to attack by a proficient cryptographer in a relatively short time. However, if you encrypt only short files with relatively long (but different) keys, crypt is still fairly secure. In most environments, people lack the skill and patience to break files encoded with crypt. You should use crypt for encoding information that is confidential but not likely to come under serious attack.

Unfortunately, some versions of the UNIX operating system being produced today, particularly those designated for export, do not have a crypt utility. If your UNIX system does not have one, obtain a similar utility to provide a certain amount of security against casual attacks.

SPECIAL /dev FILES

Within the UNIX operating system, devices such as disks, terminals, etc., are treated as special files. These files appear in the directory /dev. For example, /dev/tty00 might represent a terminal and /dev/dk10 might represent one of the disks. Like any other file, each special file has an owner, a group, and a set of permission bits associated with it. If permitted, a user may perform read and write operations on it.

Internally, a special file is represented by a pair of numbers—the major device number (an index to a particular device driver) and the minor device number (one of the devices controlled by that driver).

Special files simplify the user interface to devices by allowing one mechanism to handle access to both devices and regular files. However, the use of three types of device-special files can compromise system security. The devices in question are memory, disks, and terminals.

MEMORY

The UNIX operating system kernel was kept small and simple due to memory limitations of circa-1970 minicomputers. Therefore, UNIX uses memory device-special files instead of system calls to report status of running processes.

Reading or writing a location in these special files has the effect of reading or writing the associated system memory location. The ps utility uses the memory device-special files to read the information in the system's process table and report about running processes.

Such a structure may be appealing for system implementation, but it is a nightmare in terms of maintaining system security. The memory device-special files provide a window into the running system through which any user's proprietary programs or data can be observed. Even worse, this window allows access to critical variables in the kernel itself. Ordinary users should never have read or write permissions on the memory device-special files.

DISKS

Disk device-special files provide a convenient method for system administrators to specify particular disks to be mounted, backed up, etc. Reading the information from such a file would reveal the disk data in "raw" form; i.e., blocks concerned with maintaining directory structures and raw data blocks would be mixed in a seemingly random manner. However, a sophisticated user with knowledge of the file-system structure could follow the disk pointers and read or write any information without regard to the permissions recorded for files. Again, care must be taken to assure that ordinary users do not have access to the disk device-special files.

TERMINALS

Terminal device-special files pose a special security problem in UNIX systems. Ordinary users need read and write permission on these files while they are logged on to the associated terminal, to allow them to use the write utility to send real-time messages to each other.

The problem is that users should not be allowed read permissions for terminals that they are not logged on to. Such access allows them to intercept data that is entered at the keyboard, including passwords.

Most UNIX implementations check access permissions on open calls only. Therefore, user 1 can start a background process that opens user 1's terminal for reading while he or she is logged on to it. User 1 can then log off, allowing user 2 to log on. At any time, the background process can issue a read and thereby intercept input from the terminal.

Although such an attack is hard to defend, it is easy to detect. The telltale symptom is that the background process, not the system, receives user 2's input, and it appears that the input line of data is lost. If the system indicates that user 2's password is incorrect but user 2 is fairly certain it was entered correctly, that password may have been intercepted. If this occurs, it is a good idea for user 2 to log on to another terminal and quickly change his or her password.

DIAL-UP LINES

Dial-up lines pose additional problems. For example, the UNIX System V user's manual advises you to terminate a session simply by hanging up the phone. This is a bad idea; the next user who dials in may be able to resume your session.

Call-back modems offer improved security but should not be relied upon absolutely. Persons knowledgeable about the workings of the phone system may be able to foil these.

THE UNIX PASSWORD SYSTEM

Most time-sharing systems' passwords are kept in secret restricted files that ordinary users cannot read. The UNIX operating system takes a different approach. All passwords are stored in

(continued)

encrypted form in the file /etc/passwd, along with other information in plain form, such as log-on IDs, home directories, users' names, etc.

All users can read this file. This is the only place on the system where passwords are kept. Passwords are never stored anywhere in plain unencrypted form. However, the exception to this rule is the uucp file (L.sys, which we will discuss later), which contains passwords for restricted access to other systems.

No means are provided for decrypting the passwords in /etc/passwd, even by the superuser. If you forget your password, you cannot have the superuser find out what it is; you can only ask him or her to change it for you.

THE /etc/passwd FILE

Figure 2 shows a typical /etc/passwd file entry and defines the individual data fields within it. The system administrator can add new entries simply by editing the /etc/passwd file.

System administrators, please take note: Never set up log-on IDs with null passwords. This is sometimes done at universities where student accounts are set up without a password entry in /etc/passwd. The intention is that the student can supply it the first time he or she logs on. Anyone can peruse the /etc/passwd file looking for accounts with null passwords and commandeer those accounts.

ATTACKS VIA THE /etc/passwd FILE

One obvious way to break into the system is to try many passwords until you find one that works. You can do this by trying to log on many times with passwords generated combinatorially or read from a dictionary or list of proper names. A disadvantage to this method is that log-on programs are often intentionally slow and sometimes record a log of unsuccessful log-on attempts.

In a variation of this approach you encrypt trial passwords and compare the resulting strings with the encrypted strings in the publicly readable /etc/password file.

A number of UNIX security features make this type of search impractical. First, UNIX uses an iterated version of the DES (Data Encryption Standard) algorithm that is unavoidably slow when implemented in software. The password encryption library routine crypt( ) (no relation to the utility crypt) requires about 1.29 seconds of VAX-11/780 processor time to encrypt a single password. This is fast enough so that a legitimate log-on sequence does not take an inordinate amount of time, but an exploratory key search would.

To prevent you from using commercially available hardware (e.g., DES chips) to perform a key search on a password file, the UNIX software uses a DES version with some minor modifications. Figure 3 and figure 4 illustrate the use of the modified DES algorithm for password encryption and verification, as adapted to the UNIX operating system.

The fields within the entry are separated by colons.

al :	davRP5LxxmSP,90/A :	112 :	20 :	AD446-3570-AlFilipski(2000) :	/a/al :	
1	2	3	4	5	6	7

Field 1 is the log-on name of the user.

Field 2 is the password field:
- The first 11 characters, davRP5LxxmS are the encrypted password.
- The next two characters, P., are the salt.
- The comma is a subfield separator.
- The 9 and 0 following the comma mean that the password must be changed no less often than every 9 weeks and no more often than every 0 weeks.
- The /A is an encoded count of when the password was last changed, in weeks, since the beginning of 1970.

Field 3 is the numerical user ID.

Field 4 is the numerical group ID.
It is a key into the field /etc/group.

Field 5 is a comment field and can contain administrative information.

Field 6 is the user's home directory.

Field 7 is the null last field. It indicates that the user's default log-on shell is /bin/sh.

Figure 2: *A typical* /etc/passwd *file entry. The fields within the entry are separated by colons.*

salt

Another security feature that makes a key search impractical is the use of salt during password encryption. The new password utility obtains a random two-character string (the salt string) from the environment. Actually, this string is a function of the current time and process ID number (PID).

Twelve bits of this salt string are then used to modify, in one of 4096 different ways, the DES algorithm that encrypts the password string given by the user. The salt is stored in the /etc/passwd file along with the encrypted password. When a user enters a password at log-on time, the salt string from his or her entry in the /etc/passwd file encrypts his or her password.

Use of the salt string enhances password security in several ways. First, even if two users happen to choose the same password, their /etc/passwd entries will almost always look completely different. This prevents a user

(continued)

from making efficient sequential searches of the /etc/passwd file to see whether anyone is using a particular password. Determining if any of 100 people have the password 23skidoo, for example, requires 100 separate applications of the encryption algorithm.

Finally, the UNIX operating system imposes many restrictions on the length and composition of the passwords. For example (under UNIX System V, release 2), passwords cannot be either entirely numeric or entirely alphabetic. This discourages the use of strings that can be easily guessed, such as people's names. A password aging system is also implemented so that users can be automatically forced to change their passwords periodically. This feature should be used!

System V also provides the option of specifying a *minimum* length of time a password must exist before it can be changed. This feature is of questionable value. We recommend that it be set to 0 but that users be educated about the foolishness of using this to maintain the same password perpetually.

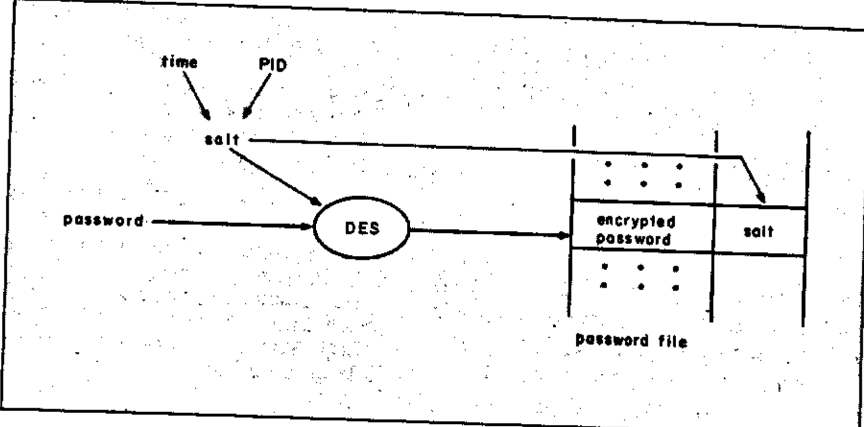


Figure 3: UNIX *password encryption. When the user enters a new password, it is encoded by a variant of the* DES *algorithm. The algorithm is modified in one of* 4096 *different ways by a random quantity called the* salt, *which is then stored in the file* /etc/passwd *along with the encrypted password.*

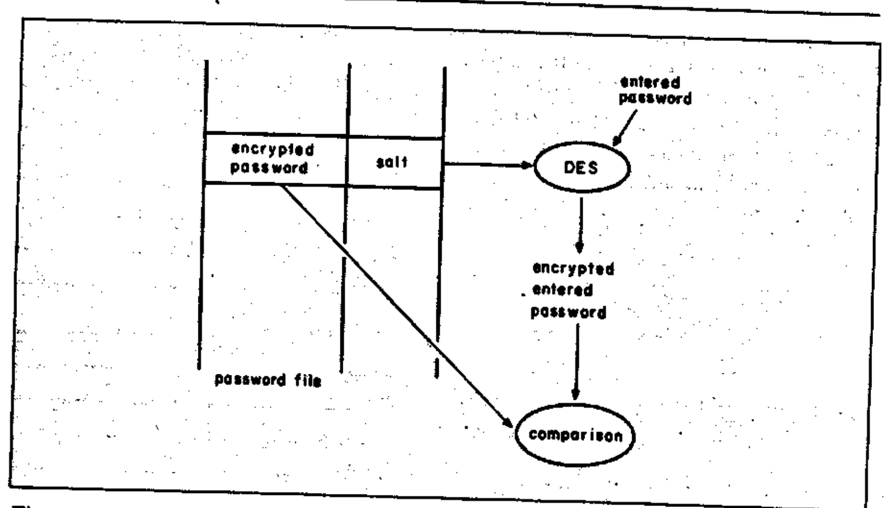


Figure 4: UNIX *password verification. The entered password is encrypted by a variant of the* DES *algorithm using the* salt *string readout of the* /etc/passwd *file entry for that user. The resulting encrypted password is compared to the encrypted password in the file. If they are the same, the password is accepted. Note that the password in the file is never "decrypted."*

THE TROJAN-HORSE PRINCIPLE

The cracker's Trojan-horse principle consists of getting a legitimate user to unwittingly execute or utilize program code set up by the intruder. Sometimes the planted code looks like an ordinary system utility. If the duped user happens to be a superuser, the security game is effectively over because it takes an intruder only a few instructions (as superuser) to set things up so that he or she can effectively become the superuser whenever he or she wishes.

For example, the intruder can quickly install a modified version of the su utility that bypasses the password check and log-file entry when a certain argument is given. Here are some of the more common ploys based on the Trojan-horse principle and some possible countermeasures for each.

FISHING FOR PASSWORDS

One technique uses a program that simulates the log-on sequence of the system. The intruder leaves this program running on a terminal that appears to be unused. When another user attempts to log on, the program easily dupes him or her into revealing his or her password. The password is then written into a file owned by the intruder, the password program kills itself, and a real log-on sequence is initiated.

A fishing program can be made sophisticated enough that users never know their passwords have been compromised. However, one feature of these programs arouses suspicion. After the fishing program obtains a password and writes it to a file, it must make a graceful transition to the real sequence for logging on.

The giveaway is that the user has already given his or her password correctly once yet must enter it again for the benefit of the "real" log-on program, which requires that the password be entered from a terminal. The fishing program usually tries to disguise this requirement by claiming

(continued)

that the password given is "incorrect." Therefore, whenever you are fairly sure that you have entered your password correctly but the system says that it is incorrect, it is a good idea to log on to another terminal and change your password immediately.

FAKE UTILITIES

Many installations have a directory where locally produced utilities can be placed. Often, users will include this directory in their path variable, which indicates where to search for commands. A user may submit a utility program that performs a service many users want but which also contains code designed to bypass the system protections. For example, the program can check if the user running it is root and, if so, perform some hidden operation, such as changing a file to setuid root, without the user's knowledge.

In a simpler method, the user places a program in his or her directory with the same name as a commonly used system command, such as ls. If another user, including root, executes this program instead of the real ls utility, the user will be at the mercy of the fake utility program.

The system administrator can limit his or her vulnerability to these attacks by keeping the current directory and any local utility directories out of the path for root.

The utility su should take care to set root's path to

```
PATH=/bin:/etc:/usr/bin
```

rather than

```
PATH=:/bin:/etc:/usr/bin
```

The colon at the beginning of the second path tells the shell to search the current directory first when looking for a command typed in by the user. Leaving this colon off makes sure that root will not inadvertently execute nonstandard utility programs.

This problem can also arise when a program uses the exec system call. For example, the code segment

```
execvp("sh",argv)
```

is dangerous and should be replaced by

```
execv("/bin/sh",argv)
```

if the intent is to execute the standard shell.

FAKE DISTRIBUTIONS

A remarkably simple and bold way of installing a Trojan horse in a system is to mail a doctored distribution tape to the system administrator with the return address of the source of the distribution tapes on it. The intruder wants the administrator to believe that the tape is an ordinary update and install it on the system. Therefore, the perpetrator of this ruse must know a considerable amount of inside information. The security-conscious administrator should be aware of this possibility.

MOUNTING A DOCTORED FILE SYSTEM

The UNIX operating system accepts the contents of a mounted file system as being completely accurate. However, it is possible for a user to create a file system on removable media. The user then, at another location, "doctors" the file system. By writing to the proper area on the media, the user sets the setuid bit on a program and changes its ownership to superuser. When the file system is subsequently remounted, the user runs his or her program as the superuser. The most effective control of this is to restrict physical access to the machine and media.

OTHER TECHNIQUES

Another technique involving user terminals is terminal page-mode buffering (see figure 5). Most moderately intelligent terminals have what is known as a page mode of operation. When the terminal is put into this mode (by certain escape sequences), it does not send information to the host but merely buffers it in screen memory. When the terminal receives certain other escape sequences, it sends the contents of its buffer to the host computer.

All of these operations can be performed by sending escape sequences (from the host) instead of typing them in at the keyboard. The cracker makes the system believe that commands he or she writes to the victim's terminal were really entered by the victim. (Under the UNIX operating system, a terminal is simply a special file and you can usually write to it for the purpose of real-time communication via the write command.) Any processes the commands invoke therefore run under the victim's user ID. The efficacy of this method depends on the ter-

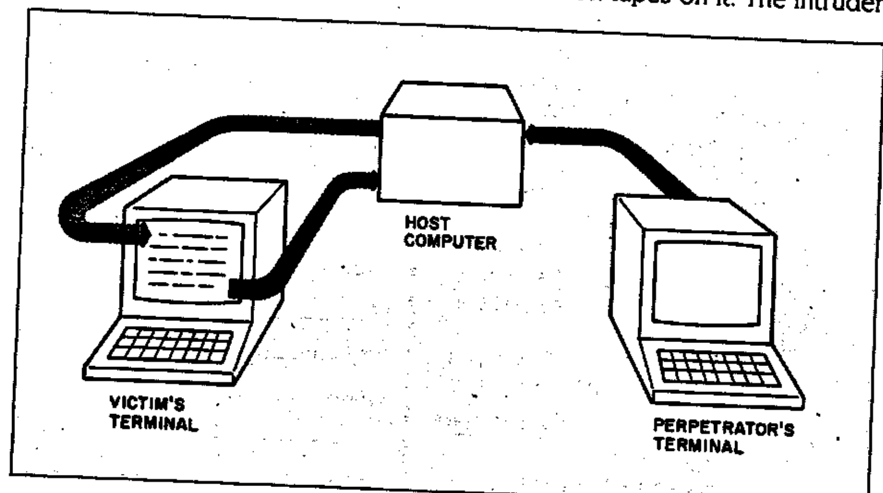


Figure 5: *Terminal page-mode buffering. The colored path indicates how the perpetrator can send his or her own commands to the victim's terminal and then to the host, where they are interpreted as commands entered by the victim. Terminals with this feature should not have* write *permission for "other" enabled.*

minal used; there may be some visible effect, such as a flash at the victim's terminal.

How can this type of attack be guarded against? Terminals should be provided without the page-mode feature or at least with the option of turning it off. Making your terminal nonwritable to others is easy—just type mesg -n. However, this does not provide certain protection against this trick, since mail or other means may be used to manipulate your terminal.

Attacks by Hogging Resources

The UNIX operating system evolved in a nonhostile environment and is relatively liberal in its granting of resources to a user. Typically, a user is allowed to run 25 concurrent processes, and each process can have 20 files open simultaneously.

These limits, in isolation, will not cause the system trouble and are generally more than enough for any legitimate use. However, in order to limit the amount of memory the kernel occupies, installations often limit the number of files open (systemwide) to around 250. This is also generally enough for even a heavily used system.

A problem arises, however, when a malicious user spawns 25 concurrent processes that each attempt to open 20 different files. The user quickly takes all available slots in the file table and essentially makes the system unusable by others. UNIX installations that might encounter such antisocial users should be adjusted to prevent any single user from causing such problems.

Another area of concern is disk-storage space. Although the size of a user's files is usually limited to 1 megabyte, typically there is no limit to the number of files a user can produce. (Or if there is, it is often only enforced at the time of logging on or off.) Therefore, it is possible for a user to allocate all the free blocks of a file system to himself or herself, resulting in irate users and possible damage to the file system. However, the damage

(continued)

Inquiry 292

DO YOU KNOW WHERE YOUR PROGRAM HAS BEEN?

If you know where your program is spending its time, you can improve its performance.

The Watcher makes it easy.

The Watcher collects data from one or more runs of your program. You can then instruct it to display as a histogram the percentage of time spent in different parts of your program and in DOS functions.

The Watcher uses symbolic information from the link map, including line numbers, or information you provide to relate the data to your source program.

THE WATCHER KNOWS!

$39.95

Stony Brook Software
Forest Road, P.O. Box 107
Wilton, New Hampshire 03086
MC or Visa orders: (603) 654-2525

For any .COM or .EXE on PC, XT, AT or compatibles DOS 2.X or 3.X. Not for use with Basic or other interpreters.

Inquiry 325

is usually minimal and easily repaired.

Attacks via uucp

The uucp family of utilities facilitates file transfer between different UNIX sites and also provides the capability for remote command execution. The uucp utilities provide a good set of security features to restrict the set of files and commands the remote user has access to, but they make up a complex system with many nooks and crannies and must be administered properly. The following security points must be observed carefully when setting up uucp utilities.

The uucp family allows the administrator to give a different log-on ID and password to each authorized remote-user system. Do this. Under no circumstances make this log-on ID the same as that of the uucp system administrator. Put all remote users in a single group that is used for nothing else. This information should be included in the /etc/passwd file.

Make sure that uucp log-ons do not get the standard shell. They should get the program uucico, which implements all the security restrictions of the uucp system. Also, their home directory should be /usr/spool/uucppublic. This log-on shell and directory are also included in the /etc/passwd entries for the remote users.

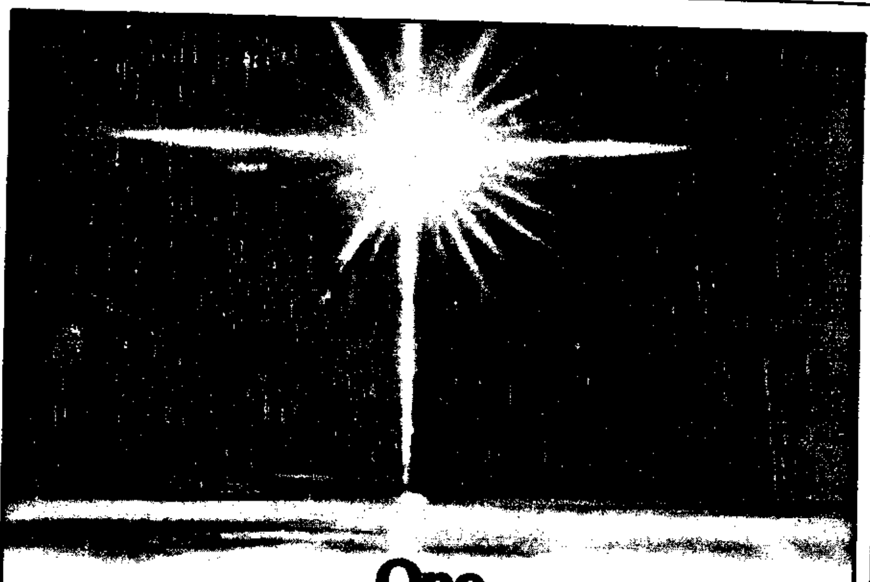
There is a file called /usr/lib/uucp/USERFILE that allows the administrator to specify, for each remote user, which UNIX subdirectories he or she will be allowed to copy files to and from. This check is in addition to the normal file-permission scheme, so that even if a file can be read by others, if it is not in one of the proper subdirectories, the remote user will not be allowed to uucp it. It is a good idea to prevent all uucp access to directories other than /usr/spool/uucppublic.

The uucp utilities provide a file that the administrator uses to specify which commands can be executed from a remote system. Do not put commands in here without a reason. For example, rmail (not mail) probably should be in here, while chmod probably should not.

Use the sequence-check feature, which keeps a sequence count of conversations with particular systems. If the sequence number given by the calling system is not what the called system expects, the conversation is disallowed. This prevents someone from masquerading as a particular remote system even if he or she knows the log-on ID and password assigned to it.

Use the call-back feature where appropriate. When this facility is set up in uucp, incoming calls are told: "OK, now I know you want to talk; I'll call you right back." This requires the

(continued)



One Star Always Shines Brightest.

Display Telecommunications Corporation announces a complete family of IBM® compatibles.

MEGA-BOARD-AT™
80286 CPU •
Our own Mega-BIOS-AT™

"...masterpiece of IBM imitation...minor masterpiece of the circuit designer's art." Winn L. Rosch, Cloning Your Own PC, PC Magazine, July 10, 1984.

AT-BIOS licensing.

MEGA-BOARD-XT™
The industry standard with our Mega-BIOS

OEM QUANTITY PRICING AVAILABLE ON REQUEST

XT Bare Board	59.95
XT Assembled 256K	299.95
Mega-Case	69.95
Mega-Kit	750.00
Mega BIOS ROM	29.95
Power Supply	89.95

MEGA-NET™
Token-passing ring LAN •
IBM NETBIOS Compatible

"The most compatible IBM clone I've ever worked with." Lee Konowe, American Software Club, Ridgefield, Ct.

IBM is a registered trademark of International Business Machines Corporation.

XT-BIOS licensing.

8445 Freeport Parkway • Suite 445 • Irving, TX 75063
1–800–227-8383 • For Technical Calls Only: 1–214–607-1382
TELEX 5106000176 DTC UD

We've Earned Our Reputation. Let Us Help You Earn Yours.

Inquiry 115

caller to be at the phone number that the called system knows as his or hers. The call-back feature may be specified for a remote system by setting a flag in that system's entry in the file /usr/lib/uucp/USERFILE.

Finally, make sure that permissions are set up properly on all the uucp administrative files. For example, the file /usr/lib/uucp/L.sys contains numbers and passwords for other systems and obviously must be unreadable by anyone but the uucp administrator. In general, it is necessary to set the permissions of the uucp files with extreme care.

If the above precautions are taken, uucp should not present any substantial security hole in a UNIX installation. However, any outside line to a computer slightly decreases the security of the system. An intruder who breaks into the remote system can now have some access to yours as well.

The security features of uucp are well designed to minimize this threat. However, uucp is a complicated system that no doubt contains security holes that have not yet been discovered or plugged. For example, ways have been suggested to induce uucp to create files whose owner is the uucp administrator but whose content is determined by some other user. Such a file may then be executed to gain access to sensitive information in /usr/lib/uucp. The uucp facilities provide a favorite hunting ground for crackers, and the security-conscious administrator would be well advised to keep an eye on it.

ATTACKS VIA MAIL

To facilitate communication among users, the UNIX mail system is set up as belonging to a mail administration group, with /bin/mail a setgid program. The text of all mail sent is kept in a central location (the directory /usr/mail in System V) and can be read and written to by anyone in the group mail.

As with any setuid or setgid program, mail must be carefully administered. Clearly, it would be disastrous to allow a user to create his or her own setgid mail program, for example, by writing over /bin/mail. Such a program could be used to read or forge mail.

Some common versions of the mail utility allow a user to easily forge a false signature on mail sent by him. This is a very serious defect. The mail program should determine the identity of the sender via the getuid system call and not rely on other (possibly faked) means of identification.

PREVENTING THE DAMAGE

The greatest security weakness of the UNIX operating system is the power inherent in root. Therefore, an important overall principle is to *minimize the use of* root.

This may be done in several ways. First, use specialized log-on IDs instead of root whenever possible. If user ID bin or nuucp suffices to do a job, don't use root.

Second, make judicious use of setuid programs in lieu of giving out root. For example, if a user occasionally needs to mount a particular file system, having a program that is setuid to root is preferable to giving out the root password and is more convenient than having the user request that the system administrator provide this service.

Finally, and perhaps obviously, change the root password frequently.

UNDOING THE DAMAGE

What can a system administrator do if he or she suspects that someone has broken root? There are several kinds of traces that an inept or casual cracker may have left behind. The utility su maintains a log of uses or attempted uses. Programs that are setuid to root can easily be discovered using the find utility.

For example, the following command prints out the path names of all files on the system that are owned by root and have the setuid bit set:

```
find / -perm -0004000 -user root -print
```

(On some systems, the ncheck utility may be used for the same purpose.) Finally, if the perpetrator has avoided these means of detection, it is possible to modify the kernel to print out a secret log of superuser activity to a file or to the console.

Ridding the system of all effects of a hostile superuser is a big job. Essentially, the system needs to be generated again from known secure sources. Special attention needs to be paid to all setuid programs.

COSTS AND REQUIREMENTS

There are many steps an administrator can take to prevent attacks. Many of these precautions are free. However, some security measures cost money, efficiency, or ease of use of the system. You must make an intelligent evaluation of the real security requirements of a particular installation before you establish a security program.

CONCLUSION

Some people claim that the UNIX operating system provides no security. While it is true that UNIX is inadequate for some types of classified government projects, so are most other operating systems.

We have outlined some security threats to the UNIX system and their related countermeasures. An administrator who is aware of these methods can maintain a UNIX system installation that provides a sufficient degree of privacy and protection for most applications. ■

BIBLIOGRAPHY

Grampp, F. T., and R. H. Morris. "UNIX Operating System Security," *AT&T Bell Laboratories Technical Journal*, volume 63, number 8, part 2, October 1984, pages 1649–1672.

Morris, R., and K. Thompson. "Password Security: A Case History," *Programmers Manual for UNIX System* III *Volume* II. *Supplementary Documents*, Western Electric Corporation, October 1981.

Reeds, J. A., and P. J. Weinberger. "File Security and the UNIX System CRYPT Command," *AT&T Bell Laboratories Technical Journal*, volume 63, number 8, part 2, October 1984, pages 1673–1684.

Ritchie, D. M. "On the Security of UNIX," *Programmers Manual for* UNIX *System* III *Volume* II. *Supplementary Documents*, Western Electric Corporation, October 1981.