How To Write a Setuid Program

Matt Bishop

Research Institute for Advanced Computer Science

NASA Ames Research Center

Moffett Field, CA 94035

ABSTRACT

UNIX† systems allow certain programs to grant privileges to users temporarily; these are called setuid programs. Because they explicitly violate the protection scheme designed into UNIX, they are among the most difficult programs to write. This paper discusses how to write these programs to make using them to compromise a UNIX system as difficult as possible.

Introduction

A typical problem in systems programming is often posed as a problem of keeping records [1]. Suppose someone has written a program and wishes to keep a record of its use. This file, which we shall call the *history file*, must be writable by the program (so it can be kept up to date), but not by anyone else (so that the entries in it are accurate.) UNIX solves this problem by providing two sets of identifications for processes. The first set, called the *real* user identification and group identification (or UID and GID, respectively), indicate the real user of the process. The second set, called the *effective* UID and GID, indicate what rights the process has, which may be, and often are, different from the

† UNIX is a trademark of Bell Laboratories.

real UID and GID. The protection mask of the file which, when executed, produces the process contains a bit which is called the *setuid* bit. (There is another such bit for the effective GID.) If that bit is not set, the effective UID of the process will be that of the person executing the file; but if the setuid bit is set (so the program runs in *setuid mode*), the effective UID will be that of the owner of the file, not that of the person executing the file. In either case, the real UID and GID are those of the person executing the file. So if only the owner of the history file (who is the user with the same UID as the file) can write on it, the setuid bit of the file containing the program is turned on, and the UIDs of this file and the history file are the same, then when someone runs the program, that process can write into the history file.

These programs are called *setuid programs*, and exist to allow ordinary users to perform functions which they could not perform otherwise. Without them, many UNIX systems would be quite unusable. An example of a setuid program performing an essential function is a program which lists the active processes on a system with protected memory. Since memory is protected, normally only the privileged user *root* could scan memory to list these processes. However, this would prevent other users from keeping track of their jobs. As with the history file, the solution is to use a setuid program, with *root* privileges, to read memory and list the active processes.

This paper discusses the security problems introduced by setuid programs, and offers suggestions on methods of programming to reduce, or eliminate, these threats. The reader should bear in mind that on some systems, the mere existence of a setuid program introduces security holes; however, it is possible to eliminate the obvious ones.

Attacks

Before we discuss the ways to deal with the security problems, let us look at the two main types of attacks setuid programs can cause. The first involves executing a sequence of commands defined by the attacker (either interactively or via a script), and the second substituting data of the attacker's choosing for data created by a program.

In the first, an attacker takes advantage of the setuid program's running with special privileges to force it to execute whatever commands he wants. As an example, suppose an attacker found a copy of the Bourne shell *sh* (1)† that was setuid to *root*. The attacker could then execute the shell, and − since the shell would be interactive − type whatever commands he desired. As the shell is setuid to *root*, these commands would be executed as though *root* had typed them. Thus, the attacker could do anything he wanted, since *root* is the most highly privileged user on the system. Even if the shell were changed to read from a command file (called a *script*) rather than accept commands interactively, the attacker could simply create his own script and run the shell using it. This is an example of something that should be avoided, and sounds like it is easy to avoid − but it occurs surprisingly often.

† The usual notation for referencing UNIX commands is to put the name of the command in italics, and the first time the name appears in a document, to follow it by the section number of the *UNIX Programmers' Manual* in which it appears; this number is enclosed in parentheses. There are two versions of the manual referred to in this paper, one for 4.2 BSD UNIX [2], and one for System V UNIX [3]. Most commands are in the same section in both manuals; when this is not true, the section for each manual will be given.

One way such an attack was performed provides a classic example of why one needs to be careful when designing system programs. A UNIX utility called *at*(1) gives one the capability to have a command file executed at a specified time; the *at* program spools the command file and a daemon executes it at the appropriate time. The daemon determined when to execute the command file by the name under which it was spooled. However, the daemon assumed the owner of the command file was the person who requested that script to be executed; hence, if one could find a world-writable file owned by another in the appropriate directory, one could run many commands with the other's privileges. Cases like this are the reason much of the emphasis on writing good setuid programs involves being very sure those programs do not create world-writable files by accident.

There are other, more subtle, problems with world-writable files. Occasionally programs will use temporary files for various purposes, the function of the program depending on what is in the file. (These programs need not be setuid to anyone.) If the program closes the temporary file at any point and then reopens it later, an attacker can replace the temporary file with a file with other data that will cause the program to act as the attacker desires. If the replacement file has the same owner and group as the temporary file, it can be very difficult for the program to determine if it is being spoofed.

Setuid programs create the conditions under which the tools needed for these two attacks can be made. That does not mean those tools will be made; with attention to detail, programmers and system administrators can prevent an attacker from using setuid programs to compromise the system in these ways. In order to provide some context for discussion, we should look at the ways in which setuid programs interact with their environment.

Threats from the Environment

The term *environment* refers to the milieu in which a process executes. Attributes of the environment relevant to this discussion are the UID and GID of the process, the files that the process opens, and the list of environment variables provided by the command interpreter under which the process executes. When a process creates a subprocess, all these attributes are inherited unless specifically reset. This can lead to problems.

Be as Restrictive as Possible in Choosing the UID and GID

The basic rule of computer security is to minimize damage resulting from a break-in. For this reason, when creating a setuid program, it should be given the least dangerous UID and GID possible. If, for example, game programs were setuid to *root*, and there were a way to get a shell with *root* privileges from within a game, the game player could compromise the entire computer system. It would be far safer to have a user called *games* and run the game programs setuid to that user. Then, if there were a way to get a shell from within a game, at worst it would be setuid to *games* and only game programs could be compromised.

Related to this is the next rule.

Reset Effective UIDs Before Calling exec†

Resetting the effective UID and GID before calling *exec* seems obvious, but it is often overlooked. When it is, the user may find himself running a program with unexpected privileges. This happened once at a site which had its game programs setuid to *root*; unfortunately, some of the games allowed the user to run subshells from within the games. Needless to say, this problem was fixed the day it was discovered!

One difficulty for many programmers is that *exec* is often called within a library subroutine such as *popen*(3) or *system*(3) and that the programmer is either not aware of this, or forgets that these functions do not reset the effective UIDs and GIDs before calling *exec*. Whenever calling a routine that is designed to execute a command as though that command were typed at the keyboard, the effective UID and GID should be reset unless there is a specific reason not to.

Close All But Necessary File Descriptors Before Calling exec

This is another requirement that most setuid programs overlook. The problem of failing to do this becomes especially acute when the program being *exec*'ed may be a user program rather than a system one. If, for example, the setuid program were reading a sensitive file, and that file had descriptor number 9, then any *exec*ed program could also read the sensitive file (because, as the manual page warns, ‘‘[d]escriptors open in the

† *Exec* is a generic term for a number of system and library calls; these are described by the manual pages *exec*(2) in the System V manual and *execve*(2) and *execl*(3) in the 4.2 BSD manual.

calling process remain open in the new process ...'')

The easiest way to prevent this is to set a flag indicating that a sensitive file is to be closed whenever an *exec* occurs. The flag should be set immediately after opening the file. Let the sensitive file's descriptor be *sfd*. In both System V and 4.2 BSD, the system call

fcntl(*sfd*, F_SETFD, 1)

will cause the file to close across *exec*s; in both Version 7 and 4.2 BSD, the call

ioctl(*sfd*, FIOCLEX, NULL)

will have the same effect. (See *fcntl*(2) and *ioctl*(2) for more information.)

Be Sure a Restricted Root Really Restricts

The *chroot*(2) system call, which may be used only by *root*, will force the process to treat the argument directory as the root of the file system. For example, the call

chroot(``/usr/riacs'')

makes the root directory ``/usr/riacs'' so far as the process which executed the system call is concerned. Further, the entry ``..'' in the new root directory is interpreted as naming the root directory. Where symbolic links are available, they too are handled correctly.

However, it is possible for *root* to link directories just as an ordinary user links files. This is not done often, because it creates loops in the UNIX file system (and that creates problems for many programs), but it does occasionally occur. These directory links can be followed regardless of whether they remain in the subtree with the restricted root. To continue the example above, if ‘‘/usr/demo’’ were linked to ‘‘/usr/riacs/demo’’, the sequence of commands

```
cd /demo
cd ..
```

would make the current working directory be ‘‘/usr’’. Using relative path names at this point (since an initial ‘‘/’’ is interpreted as ‘‘/usr/riacs’’), the user could access any file on the system. Therefore, when using this call, one must be certain that no directories are linked to any of the descendants of the new root.

Check the Environment In Which the Process Will Run

The environment to a large degree depends upon certain variables which are inherited from the parent process. Among these are the variables **PATH** (which controls the order and names of directories searched by the shell for programs to be executed), **IFS** (a list of characters which are treated as word separators), and the parent’s *umask*, which controls the protection mode of files that the subprocess creates.

One of the more insidious threats comes from routines which rely on the shell to execute a program. (The routines to be wary of here are *popen*, *system*, *execlp*(3), and *execvp*(3)†.) The danger is that the shell will not execute the program intended. As an example, suppose a program that is setuid to *root* uses *popen* to execute the program *printfile*. As *popen* uses the shell to execute the command, all a user needs to do is to

† *execlp* and *execvp* are in section 2 of the System V manual.

alter his **PATH** environment variable so that a private directory is checked before the system directories. Then, he writes his own program called *printfile* and puts it in that private directory. This private copy can do anything he likes. When the *popen* routine is executed, his private copy of *printfile* will be run, with *root* privileges.

On first blush, limiting the path to a known, safe path would seem to fix the problem. Alas, it does not. When the Bourne shell *sh* is used, there is an environment variable **IFS** which contains a list of characters that are to be treated as word separators. For example, if **IFS** is set to ‘‘o’’, then the shell command *show* (which prints mail messages on the screen) will be treated as the command *sh* with one argument *w* (since the ‘‘o’’ is treated as a blank.) Hence, one could force the setuid process to execute a program other than the one intended.

Within a setuid program, all subprograms should be invoked by their full path name, or some path known to be safe should be prefixed to the command; and the **IFS** variable should be explicitly set to the empty string (which makes white space the only command separators.)

The danger from a badly-set *umask* is that a world-writable file owned by the effective UID of a setuid process may be produced. When a setuid process must write to a file owned by the person who is running the setuid program, and that file must not be writable by anyone else, a subtle but nonetheless dangerous situation arises. The usual implementation is for the process to create the file, *chown*(2) it to the real UID and real GID of the process, and then write to it. However, if the *umask* is set to 0, and the process is interrupted after the file is created but before it is *chown*ed the process will leave a world-writable file owned by the user who has the effective UID of the setuid process.

There are two ways to prevent this; the first is fairly simple, but requires the effective UID to be that of *root*. (The other method does not suffer from this restriction; it is described in a later section.) The *umask*(2) system call can be used to reset the *umask* within the setuid process so that the file is at no time world-writable; this setting overrides any other, previous settings. Hence, simply reset *umask* to the desired value (such as 022, which prevents the file from being opened for writing by anyone other than the owner) and then open the file. (The *umask* can be reset afterwards without affecting the mode of the opened file.) Upon return, the process can safely *chown* the file to the real UID and GID of the process. (Incidentally, only *root* can *chown* a file, which is why this method will not work for programs the effective UID of which is not *root*.) Note that if the process is interrupted between the *open*(2) and the *chown* the resulting file will have the same UID and GID as the process' effective UID and GID, but the person who ran the process will not be able to write to that file (unless, of course, his UID and GID are the same as the process' effective UID and GID.)

As a related problem, *umask* is often set to a dangerous value by the parent process; for example, if a daemon is started at boot time (from the file ''/etc/rc'' or ''/etc/rc.local''), its default *umask* will be 0. Hence, any files it creates will be created world-writable unless the protection mask used in the system call creating the file is set otherwise.

Programming Style

Although threats from the environment create a number of security holes, inappropriate programming style creates many more. While many of the problems in programming style are fairly typical (see [4] for a discussion of programming style in general), some are unique to UNIX and some to setuid programs.

Do Not Write Interpreted Scripts That Are Setuid

Some versions of UNIX allow command scripts, such as shell scripts, to be made setuid. This is done by applying the setuid bit to the command interpreter used, and then interpreting the commands in the script. Unfortunately, given the power and complexity of many command interpreters, it is often possible to force them to perform actions which were not intended, and which allow the user to violate system security. This leaves the owner of the setuid script open to a devastating attack. In general, such scripts should be avoided.

As an example, suppose a site has a setuid script of *sh* commands. An attacker simply executes the script in such a way that the shell which interprets the commands appears to have been invoked by a person logging in. UNIX applies the setuid bit on the script to the shell, and since it appears to be a login shell, it becomes interactive. At that point, the attacker can type his own commands, regardless of what is in the script.

One way to avoid having a setuid script is to turn off the setuid bit on the script, and rather than calling the script directly, use a setuid program to invoke it. This program should take care to call the command interpreter by its full path name, and reset environment information such as file descriptors and environment variables to a known state. However, this method should be used only as a last resort and as a temporary measure, since with many command interpreters it is possible even under these conditions to force them to take some undesirable action.

Do Not Use creat *for Locking*

According to its manual page, ‘‘The mode given [*creat*(2)] is arbitrary; it need not allow writing. This feature has been used ... by programs to construct a simple exclusive locking mechanism.’’ In other words, one way to make a lock file is to *creat* a file with an unwritable mode (mode 000 is the most popular for this). Then, if another user tried to *creat* the same file, *creat* would fail, returning .

The only problem is that such a scheme does not work when at least one of the processes has *root*’s UID, because protection modes are ignored when the effective UID is that of *root*. Hence, *root* can overwrite the existing file regardless of its protection mode. To do locking in a setuid program, it is best to use *link*(2). If a link to an already-existing file is attempted, *link* fails, even if the process doing the linking is a *root* process and the file is not owned by *root*.

With 4.2 Berkeley UNIX, an alternative is to use the *flock*(3) system call, but this has disadvantages (specifically, it creates advisory locks only, and it is not portable to other versions of UNIX).

The issue of covert channels [5] also arises here; that is, information can be sent illicitly by controlling resources. However, this problem is much broader than the scope of this paper, so we shall pass over it.

Catch All Signals

When a process created by running a setuid file dumps core, the core file has the same UID as the real UID of the process†. By setting *umask*s properly, it is possible to obtain a world-writable file owned by someone else. To prevent this, setuid programs should catch all signals possible.

Some signals, such as **SIGKILL** (in System V and 4.2BSD) and **SIGSTOP** (in 4.2BSD), cannot be caught. Moreover, on some versions of UNIX, such as Version 7, there is an inherent race condition in signal handlers, When a signal is caught, the signal trap is reset to its default value and *then* the handler is called. As a result, receiving the same signal immediately after a previous one will cause the signal to take effect regardless of whether it is being trapped. On such a version of UNIX, signals cannot be safely caught. However, if a signal is being *ignored*, sending the process a signal will *not* cause

† On some versions of UNIX, such as 4.2BSD, no core file is produced if the real and effective UIDs of the process differ.

the default action to be reinstated; so, signals can be safely ignored.

The signals **SIGCHLD**, **SIGCONT**, **SIGTSTP**, **SIGTTIN**, and **SIGTTOU**† (all of which relate to the stopping and starting of jobs and the termination of child processes) should be caught unless there is a specific reason not to do this, because if data is kept in a world-writable file, or data or lock files in a world-writable directory such as ‘‘/tmp’’, one can easily change information the process (presumably) relies upon. Note, however, that if a system call which creates a child (such as *system*, *popen*, or *fork*(2)) is used, the **SIGCHLD** signal will be sent to the process when the command given *system* is finished; in this case, it would be wise to ignore **SIGCHLD**.

This brings us to our next point.

Be Sure Verification Really Verifies

When writing a setuid program, it is often tempting to assume data upon which decisions are based is reliable. For example, consider a spooler. One setuid process spools jobs, and another (called the *daemon*) runs them. Assuming that the spooled files were placed there by the spooler, and hence are ‘‘safe’’, is again a recipe for disaster; the *at* spooler discussed earlier is an example of this. Rather, the daemon should attempt to verify that the spooler placed the file there; for example, the spooler should log that a file was spooled, who spooled it, when it was spooled, and any other useful information, in a

† The latter four are used by various versions of Berkeley UNIX and their derivatives to suspend and continue jobs. They do not exist on many UNIXes, including System V.

protected file, and the daemon should check the information in the log against the spooled file's attributes. With the problem involving *at*, since the log file is protected, the daemon would never execute a file not placed in the spool area by the spooler.

Make Only Safe Assumptions About Recovery Of Errors

If the setuid program encounters an unexpected situation that the program cannot handle (such as running out of file descriptors), the program should not attempt to correct for the situation. It should stop. This is the opposite of the standard programming maxim about robustness of programs, but there is a very good reason for this rule. When a program tries to handle an unknown or unexpected situation, very often the programmer has made certain assumptions which do not hold up; for example, early versions of the command *su*(1) made the assumption that if the password file could not be opened, something was disastrously wrong with the system and the person should be given *root* privileges so that he could fix the problem. Such assumptions can pose extreme danger to the system and its users.

When writing a setuid program, keep track of things that can go wrong − a command too long, an input line too long, data in the wrong format, a failed system call, and so forth − and at each step ask, ‘‘if this occurred, what should be done?’’ If none of the assumptions can be verified, or the assumptions do not cover all cases, at that point the setuid program should *stop*. Do not attempt to recover unless recovery is guaranteed; it is too easy to produce undesirable side-effects in the process.

Once again, when writing a setuid program, if you are not sure how to handle a condition, exit. That way, the user cannot do any damage as a result of encountering (or creating) the condition.

For an excellent discussion of error detection and recovery under UNIX, see [6].

Be Careful With I/O Operations

When a setuid process must create and write to a file owned by the person who is running the setuid program, either of two problems may arise. If the setuid process does not have permission to create a file in the current working directory, the file cannot be created. Worse, it is possible that the file may be created and left writable by anyone. The usual implementation is for the process to create the file, *chown* it to the real UID and real GID of the process, and then write to it. However, if the *umask* is set to 0, and the process is interrupted after the file is created but before it is *chown*ed, the process will leave a world-writable file owned by the user who has the effective UID of the setuid process.

The section on checking the environment described a method of dealing with this situation when the program is setuid to *root*. That method does not work when the program is setuid to some other user. In that case, the way to prevent a setuid program from creating a world-writable file owned by the effective UID of the process is far more complex, but eliminates the need for any *chown* system calls. In this method, the process *fork*(2)s, and the child resets its effective UID and GID to the real UID and GID. The parent then writes the data to the child via *pipe* (2) rather than to the file; meanwhile, the child creates the file and copies the data from the pipe to the file. That way, the file is never owned by the user whose UID is the effective UID of the setuid process.

Some UNIX systems, notably 4.2 BSD, allow a third method. The basic problem here is that the system call *setuid* (3)† can only set the effective UID to the real UID (unless the process runs with *root* privileges, in which case both the effective and real UIDs are reset to the argument.) Once the effective UID is reset with this call, the old effective UID can never be restored (again, unless the process runs with *root* privileges.) So it is necessary to avoid resetting any UIDs when creating the file; this leads to the creation of another process or the use of *chown* . However, 4.2BSD provides the capability to reset the effective UID independently of the real UID using the system call *setreuid* (2). A similar call, *setregid* (2), exists for the real and effective GIDs So, all the program need do is use these calls to exchange the effective and real UIDs, and the effective and real GIDs. That way, the old effective UID can be easily restored, and there will not be a problem creating a file owned by the person executing the setuid program.

Conclusion

To summarize, the rules to remember when writing a setuid program are:

- Be as restrictive as possible in choosing the UID and GID.

† This system call is in section 2 of the System V manual.

- Reset effective UIDs and GIDs before calling *exec*.
- Close all but necessary file descriptors before calling *exec*.
- Be sure a restricted root really restricts.
- Check the environment in which the process will run.
- Do not write interpreted scripts that are setuid.
- Do not use *creat* for locking.
- Catch all signals.
- Be sure verification really verifies.
- Make only safe assumptions about recovery of errors.
- Be careful with I/O operations.

Setuid programs are a device to allow users to acquire new privileges for a limited amount of time. As such, they provide a means for overriding the protection scheme designed into UNIX. Unfortunately, given the way protection is handled in UNIX, it is the best solution possible; anything else would require users to share passwords widely, or the UNIX kernel to be rewritten to allow access lists for files and processes. For these reasons, setuid programs need to be written to keep the protection system as potent as possible even when they evade certain aspects of it. Thus, the designers and implementors of setuid programs should take great care when writing them.

Acknowledgements Thanks to Bob Brown, Peter Denning, George Gobel, Chris Kent, Rich Kulawiec, Dawn Maneval, and Kirk Smith, who reviewed an earlier draft of this paper, and made many constructive suggestions.

References

[1] Aleph-Null, ‘‘Computer Recreations,’’ *Software − Practise and Experience* **1**(2) pp. 201 − 204 (April − June 1971)

[2] *UNIX System V Release 2.0 Programmer Reference Manual*, DEC Processor Version, AT&T Technologies (April 1984)

[3] *UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA (August 1983)

[4] Kernighan, Brian and Plauger, P., *The Elements of Programming Style, Second Edition*, McGraw-Hill Book Company, New York, NY (1978)

[5] Lampson, Butler, ''A Note on the Confinement Problem,'' *CACM* **16**(10) pp. 613 − 615 (October 1973)

[6] Darwin, Ian and Collyer, Geoff, ''Can't Happen or /* NOTREACHED */ or Real Programs Dump Core,'' 1985 Winter USENIX Proceedings (January 1985)