NAME

setuid − checklist for security of setuid programs

DESCRIPTION

Writing a secure setuid (or setgid) program is tricky. There are a number of possible ways of subverting such a program. The most conspicuous security holes occur when a setuid program is not sufficiently careful to avoid giving away access to resources it legitimately has the use of. Most of the other attacks are basically a matter of altering the program's environment in unexpected ways and hoping it will fail in some security-breaching manner. There are generally three categories of environment manipulation: supplying a legal but unexpected environment that may cause the program to directly do something insecure, arranging for error conditions that the program may not handle correctly, and the specialized subcategory of giving the program inadequate resources in hopes that it won't respond properly.

The following are general considerations of security when writing a setuid program.

☐ The program should run with the weakest userid possible, preferably one used only by itself. A security hole in a setuid program running with a highly-privileged userid can compromise an entire system. Security-critical programs like *passwd*(1) should always have private userids, to minimize possible damage from penetrations elsewhere.

☐ The result of *getlogin* or *ttyname* may be wrong if the descriptors have been meddled with. There is *no* foolproof way to determine the controlling terminal or the login name (as opposed to uid) on V7.

☐ On some systems (not ours), the setuid bit may not be honored if the program is run by *root*, so the program may find itself running as *root*.

☐ Programs that attempt to use *creat* for locking can foul up when run by *root*; use of *link* is preferred when implementing locking. Using *chmod* for locking is an obvious disaster.

☐ Breaking an existing lock is very dangerous; the breakdown of a locking protocol may be symptomatic of far worse problems. Doing so on the basis of the lock being 'old' is sometimes necessary, but programs can run for surprising lengths of time on heavily-loaded systems.

☐ Care must be taken that user requests for i/o are checked for permissions using the user's permissions, not the program's. Use of *access* is recommended.

☐ Programs executed at user request (e.g. shell escapes) must not receive the setuid program's permissions; use of daughter processes and *setuid(getuid())* plus *setgid(getgid())* after *fork* but before *exec* is vital.

☐ Similarly, programs executed at user request must not receive other sensitive resources, notably file descriptors. Use of *closeall*(3) or close-on-exec arrangements, on systems which have them, is recommended.

☐ Programs activated by one user but handling traffic on behalf of others (e.g. daemons) should avoid doing *setuid(getuid())* or *setgid(getgid())*, since the original invoker's identity is almost certainly inappropriate. On systems which permit it, use of *setuid(geteuid())* and *setgid(getegid())* is recommended when performing work on behalf of the system as opposed to a specific user.

☐ There are inherent permission problems when a setuid program executes another setuid program, since the permissions are not additive. Care should be taken that created files are not owned by the wrong person. Use of *setuid(geteuid())* and its gid counterpart can help, if the system allows them.

☐ Care should be taken that newly-created files do not have the wrong permission or ownership even momentarily. Permissions should be arranged by using *umask* in advance, rather than by creating the file wide-open and then using *chmod*. Ownership can get sticky due to the limitations of the setuid concept, although using a daughter process connected by a pipe can help.

☐ Setuid programs should be especially careful about error checking, and the normal response to a strange situation should be termination, rather than an attempt to carry on.

The following are ways in which the program may be induced to carelessly give away its special privileges.

☐ The directory the program is started in, or directories it may plausibly *chdir* to, may contain programs with the same names as system programs, placed there in hopes that the program will activate a shell with a permissive **PATH** setting. **PATH** should *always* be standardized before invoking a shell (either directly or via *popen* or *execvp/execlp*).

☐ Similarly, a bizarre **IFS** setting may alter the interpretation of a shell command in really strange ways, possibly causing a user-supplied program to be invoked. **IFS** too should always be standardized before invoking a shell. (Our shell does this automatically.)

☐ Environment variables in general cannot be trusted. Their contents should never be taken for granted.

☐ Setuid shell files (on systems which implement such) simply cannot cope adequately with some of these problems. They also have some nasty problems like trying to run a *.profile* when run under a suitable name. They are terminally insecure, and must be avoided.

☐ Relying on the contents of files placed in publically-writeable directories, such as */tmp*, is a nearly-incurable security problem. Setuid programs should avoid using */tmp* entirely, if humanly possible. The sticky-directories modification (sticky bit on for a directory means only owner of a file can remove it) (we have this feature) helps, but is not a complete solution.

☐ A related problem is that spool directories, holding information that the program will trust later, must never be publically writeable even if the files in the directory are protected. Among other sinister manipulations that can be performed, note that on many Unixes (not ours), a core dump of a setuid program is owned by the program's owner and not by the user running it.

The following are unusual but possible error conditions that the program should cope with properly (resource-exhaustion questions are considered separately, see below).

☐ The value of *argc* might be 0.

☐ The setting of the *umask* might not be sensible. In any case, it should be standardized when creating files not intended to be owned by the user.

☐ One or more of the standard descriptors might be closed, so that an opened file might get (say) descriptor 1, causing chaos if the program tries to do a *printf*.

☐ The current directory (or any of its parents) may be unreadable and unsearchable. On many systems *pwd*(1) does not run setuid-root, so it can fail under such conditions.

☐ Descriptors shared by other processes (i.e., any that are open on startup) may be manipulated in strange ways by said processes.

☐ The standard descriptors may refer to a terminal which has a bizarre mode setting, or which cannot be opened again, or which gives end-of-file on any read attempt, or which cannot be read or written successfully.

☐ The process may be hit by interrupt, quit, hangup, or broken-pipe signals, singly or in fast succession. The user may deliberately exploit the race conditions inherent in catching signals; ignoring signals is safe, but catching them is not.

☐ Although non-keyboard signals cannot be sent by ordinary users in V7, they may perhaps be sent by the system authorities (e.g. to indicate that the system is about to shut down), so the possibility cannot be ignored.

☐ On some systems (not ours) there may be an *alarm* signal pending on startup.

☐ The program may have children it did not create. This is normal when the process is part of a pipeline.

☐ In some non-V7 systems, users can change the ownerships of their files. Setuid programs should avoid trusting the owner identification of a file.

☐ User-supplied arguments and input data *must* be checked meticulously. Overly-long input stored in an array without proper bound checking can easily breach security. When software depends on a file being in a specific format, user-supplied data should never be inserted into the file without being checked first. Meticulous checking includes allowing for the possibility of non-ASCII characters.

- Temporary files left in public directories like */tmp* might vanish at inconvenient times.

The following are resource-exhaustion possibilities that the program should respond properly to.

- The user might have used up all of his allowed processes, so any attempt to create a new one (via *fork* or *popen*) will fail.
- There might be many files open, exhausting the supply of descriptors. Running *closeall*(3), on systems which have it, is recommended.
- There might be many arguments.
- The arguments and the environment together might occupy a great deal of space.

Systems which impose other resource limitations can open setuid programs to similar resource-exhaustion attacks.

Setuid programs which execute ordinary programs without reducing authority pass all the above problems on to such unprepared children. Standardizing the execution environment is only a partial solution.

SEE ALSO

closeall(3), standard(3)

HISTORY

Locally written, although based on outside contributions.

BUGS

The list really is rather long... and probably incomplete.

Neither the author nor the University of Toronto accepts any responsibility whatever for the use or non-use of this information.