

Overview

- We examine some general guidelines for writing secure `setuid` programs
- Although presented in a security context many of the guidelines are simply best programming practices that should be followed in all of your code irrespective of its `setuid` status
- Recommendations are drawn from...
 - *How to Write Setuid Programs* by Bishop
 - *Secure Programming Cookbook* by Viega and Messier
 - *Secure Coding in C and C++* by Seacord

Choose a Restrictive UID and GID

- We want to minimise the possible damage resulting from a successful exploit so where appropriate create a new user and `setuid(newuser)`
- For example, if your program needs to update a shared high scores listing, make it `setuid games` rather than `setuid root` (cf. Apache, MySQL etc.)

Drop Privileges Before exec

- User IDs are inherited across `exec` so make sure to drop privileges before running new programs
- If you don't, the child process will inherit the parent's privileges and present a new avenue of attack

Close File Descriptors Before exec

- A `setuid` program may be reading a sensitive file, assigned file descriptor 5
- File descriptors are inherited across `exec`
- Crucially, access permissions are checked against an inode only once: on the call to `open`
- Thus an unprivileged program may gain unintended access to a privileged file available on an already open file descriptor
- To ensure this never happens, we can set a flag on the file descriptor to have it closed on `exec` with the `fcntl` system call

Restrict File System Access

- Normally the root directory of a process is the system root directory and thus the process can see all of the file system (and can manoeuvre freely about in it)
- Using the `chroot` system call a process can alter and limit its view of the file system by changing its root directory
- After a process has changed its root directory once it can only be restricted further
- Potential for damage after a break-in is thus constrained to some subdirectories of the file system
- Only the superuser can call `chroot`

Do Not Trust The Environment

- Inherited from the parent of a `setuid` process are environment variables such as `PATH`, `IFS` and `umask` settings: they are not to be trusted and must be sanitised by your program before use
- Certain threats come from library functions that rely on the shell to execute a program e.g. `popen`, `system`, `exec1p`
- Through manipulating the `PATH` variable the program run may not be the one intended
- The shell may not execute the intended program e.g. if a user has altered her `PATH` to search private before searching system directories

Do Not Trust The Environment

- The private copy may do anything and runs with the privileges of the `setuid` program
- The `IFS` variable contains a (potentially exploitable) list of characters interpreted as word separators
- For example, suppose we wanted to execute `/bin/show` but `IFS` had been reset to `○`
- This is an obviously contrived example but highlights the issue nonetheless

Do Not Trust The Environment

- Thus all external programs should be invoked through specifying their full path and `IFS` should be reset
- Best to avoid the latter shell-based commands altogether if at all possible
- A misconfigured `umask` may result in the creation of a world-writable file whose owner and group are that of the privileged process; its contents are not to be trusted
- When a `setuid` process must create or write to a file owned by the real user of the process and that file must not be writable by anyone else the threat of a race condition may arise

Do Not Trust The Environment

- A privileged process could create the file, change its owner to the user and then write to it
- However, if the process is interrupted after creation but before the change of ownership a world-writable file may be left on the system: its IDs are that of the privileged process; again an untrustworthy file is left in the system
- One solution is to set `umask` to `022`, create and `chown` the file before resetting `umask`
- However this approach only works for `root` as only `root` can `chown` a file; what can others do?
- Reset the effective user ID to the real user ID, set `umask` appropriately and create/write the file before raising privileges again if required

Disable Memory Dumps

- When a Unix program crashes it may produce a core dump: the process's memory is written out to disk, memory that may contain sensitive information
- The generated core may not be world-readable but still best to avoid generating it altogether as it may become readable at a later date
- We can use the `setrlimit` (set resource limits) system call to disable core file generation on a per process basis
- `setrlimit` can be used to further restrict process resource usage
- Resources that can be managed in this way include maximum time on CPU, maximum number of files that can be opened, maximum data segment size, maximum number of files that can be created

Make Safe Assumptions About Error Recovery

- If a `setuid` program encounters an unexpected situation, e.g. running out of file descriptors, do not try to handle it, simply stop and fail safely
- Check all return values for success, assume nothing

Be Careful with I/O Operations

- Our previous solution to file creation involved using `chown` which can only be called by `root`
- Better is to use `setreuid` or a variant to drop privileges temporarily, create and open the file

Beware TOCTOU Race Conditions

- A privileged program may use the `access` system call (uses the real user ID) to determine whether a user has access to a particular file and, where permitted, continue to `open` the file
- Between the `access` check and the `open` an attacker can replace the file with e.g. a symbolic link to sensitive information
- The privileged program will end up opening a file it shouldn't be allowed to: a better approach is to temporarily drop privileges before the `open`, again `setreuid` may be used
- These are time-of-check, time-of-use races

Use Temporary Files Safely

- Applications often create files for temporary use where that file may contain sensitive information
- A reasonable solution on Unix is to use `mkstemp` which will repeatedly generate a random filename and attempt to open it, if the open fails (e.g. the file exists already) the process is repeated
- On Linux, the created file will be readable and writable by the owner but no one else (0600) (before `glibc 2.0.7` the mode was 0666)
- We can make the file invisible to other by unlinking it after the call to `mkstemp`

Use Temporary Files Safely

- After the `unlink` the file can no longer be opened by name but still exists as we have a file descriptor pointing at it
- Once the last file descriptor pointing at the file is closed the file becomes inaccessible i.e. deleted
- The `mkstemp` system call returns a file descriptor for the now opened file, opened for exclusive access
- The POSIX specification says nothing about file creation mode so ensure `umask` is set appropriately before making a call to `mkstemp`