38

WRITING SECURE PRIVILEGED PROGRAMS

Privileged programs have access to features and resources (files, devices, and so on) that are not available to ordinary users. A program can run with privileges by two general means:

- The program was started under a privileged user ID. Many daemons and network servers, which are typically run as *root*, fall into this category.
- The program has its set-user-ID or set-group-ID permission bit set. When a set-user-ID (set-group-ID) program is execed, it changes the effective user (group) ID of the process to be the same as the owner (group) of the program file. (We first described set-user-ID and set-group-ID programs in Section 9.3.) In this chapter, we'll sometimes use the term set-user-ID-*root* to distinguish a set-user-ID program that gives superuser privileges to a process from one that gives a process another effective identity.

If a privileged program contains bugs, or can be subverted by a malicious user, then the security of the system or an application can be compromised. From a security viewpoint, we should write programs so as to minimize both the chance of a compromise and the damage that can be done if a compromise does occur. These topics form the subject of this chapter, which provides a set of recommended practices for secure programming, and describes various pitfalls that should be avoided when writing privileged programs.

38.1 Is a Set-User-ID or Set-Group-ID Program Required?

One of the best pieces of advice concerning set-user-ID and set-group-ID programs is to avoid writing them whenever possible. If there is an alternative way of performing a task that doesn't involve giving a program privilege, we should generally employ that alternative, since it eliminates the possibility of a security compromise.

Sometimes, we can isolate the functionality that needs privilege into a separate program that performs a single task, and exec that program in a child process as required. This technique can be especially useful for libraries. One example of such a use is provided by the *pt_chown* program described in Section 64.2.2.

Even in cases where a set-user-ID or set-group-ID is needed, it isn't always necessary for a set-user-ID program to give a process *root* credentials. If giving a process some other credentials suffices, then this option should be preferred, since running with *root* privileges opens the gates to possible security compromises.

Consider a set-user-ID program that needs to allow users to update a file on which they do not have write permission. A safer way to do this is to create a dedicated group account (group ID) for this program, change the group ownership of the file to that group (and make the file writable by that group), and write a set-group-ID program that sets the process's effective group ID to the dedicated group ID. Since the dedicated group ID is not otherwise privileged, this greatly limits the damage that can be done if the program contains bugs or can otherwise be subverted.

38.2 Operate with Least Privilege

A set-user-ID (or set-group-ID) program typically requires privileges only to perform certain operations. While the program (especially one assuming superuser privileges) is performing other work, it should disable these privileges. When privileges will never again be required, they should be dropped permanently. In other words, the program should always operate with the *least privilege* required to accomplish the tasks that it is currently performing. The saved set-user-ID facility was designed for this purpose (Section 9.4).

Hold privileges only while they are required

In a set-user-ID program, we can use the following sequence of *seteuid()* calls to temporarily drop and then reacquire privileges:

```
uid_t orig_euid;

orig_euid = geteuid();
if (seteuid(getuid()) == -1)            /* Drop privileges */
    errExit("seteuid");

/* Do unprivileged work */

if (seteuid(orig_euid) == -1)           /* Reacquire privileges */
    errExit("seteuid");

/* Do privileged work */
```

The first call makes the effective user ID of the calling process the same as its real ID. The second call restores the effective user ID to the value held in the saved set-user-ID.

For set-group-ID programs, the saved set-group-ID saves the program's initial effective group ID, and *setegid()* is used to drop and reacquire privilege. We describe *seteuid()*, *setegid()*, and other similar system calls mentioned in the following recommendations in Chapter 9 and summarize them in Table 9-1 (on page 181).

The safest practice is to drop privileges immediately on program startup, and then temporarily reacquire them as needed at later points in the program. If, at a certain point, privileges will never be required again, then the program should drop them irreversibly, by ensuring that the saved set-user-ID is also changed. This eliminates the possibility of the program being tricked into reacquiring privilege, perhaps via the stack-crashing technique described in Section 38.9.

Drop privileges permanently when they will never again be required

If a set-user-ID or set-group-ID program finishes all tasks that require privileges, then it should drop its privileges permanently in order to eliminate any security risk that could occur because the program is compromised by a bug or other unexpected behavior. Dropping privileges permanently is accomplished by resetting all process user (group) IDs to the same value as the real (group) ID.

From a set-user-ID-*root* program whose effective user ID is currently 0, we can reset all user IDs using the following code:

```
if (setuid(getuid()) == -1)
    errExit("setuid");
```

However, the above code does *not* reset the saved set-user-ID if the effective user ID of the calling process is currently nonzero: when called from a program whose effective user ID is nonzero, *setuid()* changes only the effective user ID (Section 9.7.1). In other words, in a set-user-ID-*root* program, the following sequence doesn't permanently drop the user ID 0:

```
/* Initial UIDs:    real=1000 effective=0 saved=0 */

/* 1. Usual call to temporarily drop privilege */

orig_euid = geteuid();
if (seteuid(getuid() == -1)
    errExit("seteuid");

/* UIDs changed to: real=1000 effective=1000 saved=0 */

/* 2. Looks like the right way to permanently drop privilege (WRONG!) */

if (setuid(getuid() == -1)
    errExit("setuid");

/* UIDs unchanged:  real=1000 effective=1000 saved=0 */
```

Instead, we must regain privilege prior to dropping it permanently, by inserting the following call between steps 1 and 2 above:

```
if (seteuid(orig_euid) == -1)
    errExit("seteuid");
```

On the other hand, if we have a set-user-ID program owned by a user other than *root*, then, because *setuid()* is insufficient to change the set-user-ID identifier, we must use either *setreuid()* or *setresuid()* to permanently drop the privileged identifier. For example, we could achieve the desired result using *setreuid()*, as follows:

```
if (setreuid(getuid(), getuid()) == -1)
    errExit("setreuid");
```

This code relies on a feature of the Linux implementation of *setreuid()*: if the first (*ruid*) argument is not −1, then the saved set-user-ID is also set to the same value as the (new) effective user ID. SUSv3 doesn't specify this feature, but many other implementations behave the same way as Linux.

The *setregid()* or *setresgid()* system call must likewise be used to permanently drop a privileged group ID in a set-group-ID program, since, when the effective user ID of a program is nonzero, *setgid()* changes only the effective group ID of the calling process.

General points on changing process credentials

In the preceding pages, we described techniques for temporarily and permanently dropping privileges. We now add a few general points regarding the use of these techniques:

- The semantics of some of the system calls that change process credentials vary across systems. Furthermore, the semantics of some of these system calls vary depending on whether or not the caller is privileged (effective user ID of 0). For details, see Chapter 9, and especially Section 9.7.4. Because of these variations, [Tsafrir et al., 2008] recommends that applications should use system-specific *nonstandard* system calls for changing process credentials, since, in many cases, these nonstandard system calls provide simpler and more consistent semantics than their standard counterparts. On Linux, this would translate to using *setresuid()* and *setresgid()* to change user and group credentials. Although these system calls are not present on all systems, their use is likely to be less prone to error. ([Tsafrir et al., 2008] proposes a library of functions that make credential changes using what they consider to be the best interfaces available on each platform.)
- On Linux, even if the caller has an effective user ID of 0, system calls for changing credentials may not behave as expected if the program has explicitly manipulated its capabilities. For example, if the `CAP_SETUID` capability has been disabled, then attempts to change process user IDs will fail or, even worse, silently change only some of the requested user IDs.

- Because of the possibilities listed in the two preceding points, it is highly recommended practice (see, for example, [Tsafrir et al., 2008]) to not only check that a credential-changing system call has succeeded, but also to verify that the change occurred as expected. For example, if we are temporarily dropping or reacquiring a privileged user ID using *seteuid()*, then we should follow that call with a *geteuid()* call that verifies that the effective user ID is what we expect. Similarly, if we are dropping a privileged user ID permanently, then we should verify that the real user ID, effective user ID, and saved set-user-ID have all been successfully changed to the unprivileged user ID. Unfortunately, while there are standard system calls for retrieving the real and effective IDs, there are no standard system calls for retrieving the saved set IDs. Linux and a few other systems provide *getresuid()* and *getresgid()* for this purpose; on some other systems, we may need to employ techniques such as parsing information in `/proc` files.
- Some credential changes can be made only by processes with an effective user ID of 0. Therefore, when changing multiple IDs—supplementary group IDs, group IDs, and user IDs—we should drop the privileged effective user ID last when dropping privileged IDs. Conversely, we should raise the privileged effective user ID first when raising privileged IDs.

38.3 Be Careful When Executing a Program

Caution is required when a privileged program executes another program, either directly, via an *exec()*, or indirectly, via *system()*, *popen()*, or a similar library function.

Drop privileges permanently before execing another program

If a set-user-ID (or set-group-ID) program executes another program, then it should ensure that all process user (group) IDs are reset to the same value as the real user (group) ID, so that the new program doesn't start with privileges and also can't reacquire them. One way to do this is to reset all of the IDs before performing the *exec()*, using the techniques described in Section 38.2.

The same result can be achieved by preceding the *exec()* with the call *setuid(getuid())*. Even though this *setuid()* call changes only the effective user ID in a process whose effective user ID is nonzero, privileges are nevertheless dropped because (as described in Section 9.4) a successful *exec()* goes on to copy the effective user ID to the saved set-user-ID. (If the *exec()* fails, then the saved set-user-ID is left unchanged. This may be useful if the program then needs to perform other privileged work because the *exec()* failed.)

A similar approach (i.e., *setgid(getgid())*) can be used with set-group-ID programs, since a successful *exec()* also copies the effective group ID to the saved set-group-ID.

As an example, suppose that we have a set-user-ID program owned by user ID 200. When this program is executed by a user whose ID is 1000, the user IDs of the resulting process will be as follows:

```
real=1000 effective=200 saved=200
```

If this program subsequently executes the call *setuid(getuid())*, then the process user IDs are changed to the following:

```
real=1000 effective=1000 saved=200
```

When the process executes an unprivileged program, the effective user ID of the process is copied to the saved set-user-ID, resulting in the following set of process user IDs:

```
real=1000 effective=1000 saved=1000
```

Avoid executing a shell (or other interpreter) with privileges

Privileged programs running under user control should never exec a shell, either directly or indirectly (via *system()*, *popen()*, *execlp()*, *execvp()*, or other similar library functions). The complexity and power of shells (and other unconstrained interpreters such as *awk*) mean that it is virtually impossible to eliminate all security loopholes, even if the execed shell doesn't allow interactive access. The consequent risk is that the user may be able to execute arbitrary shell commands under the effective user ID of the process. If a shell must be execed, ensure that privileges are permanently dropped beforehand.

> An example of the kind of security loophole that can occur when execing a shell is noted in the discussion of *system()* in Section 27.6.

A few UNIX implementations honor the set-user-ID and set-group-ID permission bits when they are applied to interpreter scripts (Section 27.3), so that, when the script is run, the process executing the script assumes the identity of some other (privileged) user. Because of the security risks just described, Linux, like some other UNIX implementations, silently ignores the set-user-ID and set-group-ID permission bits when execing a script. Even on implementations where set-user-ID and set-group-ID scripts are permitted, their use should be avoided.

Close all unnecessary file descriptors before an *exec()*

In Section 27.4, we noted that, by default, file descriptors remain open across an *exec()*. A privileged program may open a file that normal processes can't access. The resulting open file descriptor represents a privileged resource. The file descriptor should be closed before an *exec()*, so that the execed program can't access the associated file. We can do this either by explicitly closing the file descriptor or by setting its close-on-exec flag (Section 27.4).

38.4 Avoid Exposing Sensitive Information

When a program reads passwords or other sensitive information, it should perform whatever processing is required, and then immediately erase the information from memory. (We show an example of this in Section 8.5.) Leaving such information in memory is a security risk, for the following reasons:

- The virtual memory page containing the data may be swapped out (unless it is locked in memory using *mlock()* or similar), and could then be read from the swap area by a privileged program.

- If the process receives a signal that causes it to produce a core dump file, then that file may be read to obtain the information.

Following on from the last point, as a general principle, a secure program should prevent core dumps, so that a core dump file can't be inspected for sensitive information. A program can ensure that a core dump file is not created by using *setrlimit()* to set the `RLIMIT_CORE` resource limit to 0 (see Section 36.3).

> By default, Linux doesn't permit a set-user-ID program to perform a core dump in response to a signal (Section 22.1), even if the program has dropped all privileges. However, other UNIX implementations may not provide this security feature.

38.5 Confine the Process

In this section, we consider ways in which we can confine a program to limit the damage that is done if the program is compromised.

Consider using capabilities

The Linux capabilities scheme divides the traditional all-or-nothing UNIX privilege scheme into distinct units called *capabilities*. A process can independently enable or disable individual capabilities. By enabling just those capabilities that it requires, a program operates with less privilege than it would have if run with full *root* privileges. This reduces the potential for damage if the program is compromised.

Furthermore, using capabilities and the *securebits* flags, we can create a process that has a limited set of capabilities but is not owned by *root* (i.e., all of its user IDs are nonzero). Such a process can no longer use *exec()* to regain a full set of capabilities. We describe capabilities and the *securebits* flags in Chapter 39.

Consider using a *chroot* jail

A useful security technique in certain cases is to establish a *chroot* jail to limit the set of directories and files that a program may access. (Make sure to also call *chdir()* to change the process's current working directory to a location within the jail.) Note, however, that a *chroot* jail is insufficient to confine a set-user-ID-*root* program (see Section 18.12).

> An alternative to using a *chroot* jail is a *virtual server*, which is a server implemented on top of a virtual kernel. Because each virtual kernel is isolated from other virtual kernels that may be running on the same hardware, a virtual server is more secure and flexible than a *chroot* jail. (Several other modern operating systems also provide their own implementations of virtual servers.) The oldest virtualization implementation on Linux is User-Mode Linux (UML), which is a standard part of the Linux 2.6 kernel. Further information about UML can be found at *http://user-mode-linux.sourceforge.net/*. More recent virtual kernel projects include Xen (*http://www.cl.cam.ac.uk/Research/SRG/netos/xen/*) and KVM (*http://kvm.qumranet.com/*).

38.6 Beware of Signals and Race Conditions

A user may send arbitrary signals to a set-user-ID program that they have started. Such signals may arrive at any time and with any frequency. We need to consider the race conditions that can occur if a signal is delivered at *any* point in the execution of the program. Where appropriate, signals should be caught, blocked, or ignored to prevent possible security problems. Furthermore, the design of signal handlers should be as simple as possible, in order to reduce the risk of inadvertently creating a race condition.

This issue is particularly relevant with the signals that stop a process (e.g., `SIGTSTP` and `SIGSTOP`). The problematic scenario is the following:

1. A set-user-ID program determines some information about its run-time environment.
2. The user manages to stop the process running the program and change details of the run-time environment. Such changes may include modifying the permissions on a file, changing the target of a symbolic link, or removing a file that the program depends on.
3. The user resumes the process with a `SIGCONT` signal. At this point, the program will continue execution based on now false assumptions about its run-time environment, and these assumptions may lead to a security breach.

The situation described here is really just a special case of a *time-of-check*, *time-of-use* race condition. A privileged process should avoid performing operations based on previous verifications that may no longer hold (refer to the discussion of the *access()* system call in Section 15.4.4 for a specific example). This guideline applies even when the user can't send signals to the process. The ability to stop a process simply allows a user to widen the interval between the time of the check and the time of use.

> Although it may be difficult on a single attempt to stop a process between the time of check and time of use, a malicious user could execute a set-user-ID program repeatedly, and use another program or a shell script to repeatedly send stop signals to the set-user-ID program and change the run-time environment. This greatly improves the chances of subverting the set-user-ID program.

38.7 Pitfalls When Performing File Operations and File I/O

If a privileged process needs to create a file, then we must take care of that file's ownership and permissions to ensure that there is never a point, no matter how brief, when the file is vulnerable to malicious manipulation. The following guidelines apply:

- The process umask (Section 15.4.6) should be set to a value that ensures that the process never creates publicly writable files, since these could be modified by a malicious user.
- Since the ownership of a file is taken from the effective user ID of the creating process, judicious use of *seteuid()* or *setreuid()* to temporarily change process credentials may be required in order ensure that a newly created file doesn't belong to the wrong user. Since the group ownership of the file *may* be taken

from process's effective group ID (see Section 15.3.1), a similar statement applies with respect to set-group-ID programs, and the corresponding group ID calls can be used to avoid such problems. (To be strictly accurate, on Linux, the owner of a new file is determined by the process's file-system user ID, which normally has the same value as the process's effective user ID; refer to Section 9.5.)

- If a set-user-ID-*root* program must create a file that initially it must own, but which will eventually be owned by another user, the file should be created so that it is initially not writable by other users, either by using a suitable *mode* argument to *open()* or by setting the process umask before calling *open()*. Afterward, the program can change its ownership with *fchown()*, and then change its permissions, if necessary, with *fchmod()*. The key point is that a set-user-ID program should ensure that it never creates a file that is owned by the program owner and that is even momentarily writable by other users.
- Checks on file attributes should be performed on open file descriptors (e.g., *open()* followed by *fstat()*), rather than by checking the attributes associated with a pathname and then opening the file (e.g., *stat()* followed by *open()*). The latter method creates a time-of-use, time-of-check problem.
- If a program must ensure that it is the creator of a file, then the `O_EXCL` flag should be used when calling *open()*.
- A privileged program should avoid creating or relying on files in publicly writable directories such as `/tmp`, since this leaves the program vulnerable to malicious attempts to create unauthorized files with names expected by the privileged program. A program that absolutely must create a file in a publicly writable directory should at least ensure that the file has an unpredictable name, by using a function such as *mkstemp()* (Section 5.12).

38.8 Don't Trust Inputs or the Environment

Privileged programs should avoid making assumptions about the input they are given, or the environment in which they are running.

Don't trust the environment list

Set-user-ID and set-group-ID programs should not assume that the values of environment variables are reliable. Two variables that are particularly relevant are `PATH` and `IFS`.

`PATH` determines where the shell (and thus *system()* and *popen()*), as well as *execlp()* and *execvp()*, search for programs. A malicious user can set `PATH` to a value that may trick a set-user-ID program employing one of these functions into executing an arbitrary program with privilege. If these functions are to be used, `PATH` should be set to a trustworthy list of directories (but better still, absolute pathnames should be specified when execing programs). However, as already noted, it is best to drop privileges before execing a shell or employing one of the aforementioned functions.

`IFS` specifies the delimiting characters that the shell interprets as separating the words of a command line. This variable should be set to an empty string, which

means that only white-space characters are interpreted by the shell as word separators. Some shells always set `IFS` in this way on startup. (Section 27.6 describes one vulnerability relating to `IFS` that appeared in older Bourne shells.)

In some cases, it may be safest to erase the entire environment list (Section 6.7), and then restore selected environment variables with known-safe values, especially when executing other programs or calling libraries that may be affected by environment variable settings.

Handle untrusted user inputs defensively

A privileged program should carefully validate all inputs from untrusted sources before taking action based on those inputs. Such validation may include verifying that numbers fall within acceptable limits, and that strings are of an acceptable length and consist of acceptable characters. Among inputs that may need to be validated in this way are those coming from user-created files, command-line arguments, interactive inputs, CGI inputs, email messages, environment variables, interprocess communication channels (FIFOs, shared memory, and so on) accessible by untrusted users, and network packets.

Avoid unreliable assumptions about the process's run-time environment

A set-user-ID program should avoid making unreliable assumptions about its initial run-time environment. For example, standard input, output, or error may have been closed. (These descriptors might have been closed in the program that execs the set-user-ID program.) In this case, opening a file could inadvertently reuse descriptor 1 (for example), so that, while the program thinks it is writing to standard output, it is actually writing to the file it opened.

There are many other possibilities to consider. For example, a process may exhaust various resource limits, such as the limit on the number of processes that may be created, the CPU time resource limit, or the file size resource limit, with the result that various system calls may fail or various signals may be generated. Malicious users may attempt to deliberately engineer resource exhaustion in an attempt to subvert a program.

38.9 Beware of Buffer Overruns

Beware of buffer overruns (overflows), where an input value or copied string exceeds the allocated buffer space. Never use *gets()*, and employ functions such as *scanf()*, *sprintf()*, *strcpy()*, and *strcat()* with caution (e.g., guarding their use with `if` statements that prevent buffer overruns).

Buffer overruns allow techniques such as *stack crashing* (also known as *stack smashing*), whereby a malicious user employs a buffer overrun to place carefully coded bytes into a stack frame in order to force the privileged program to execute arbitrary code. (Several online sources explain the details of stack crashing; see also [Erickson, 2008] and [Anley, 2007].) Buffer overruns are probably the single most common source of security breaches on computer systems, as evidenced by the frequency of advisories posted by CERT (*http://www.cert.org/*) and to Bugtraq (*http://www.securityfocus.com/*). Buffer overruns are particularly dangerous in network servers, since they leave a system open to remote attack from anywhere on a network.

In order to make stack crashing more difficult—in particular, to make such attacks much more time-consuming when conducted remotely against network servers—from kernel 2.6.12 onward, Linux implements *address-space randomization*. This technique randomly varies the location of the stack over an 8 MB range at the top of virtual memory. In addition, the locations of memory mappings may also be randomized, if the soft `RLIMIT_STACK` limit is not infinite and the Linux-specific `/proc/sys/vm/legacy_va_layout` file contains the value 0.

More recent x86-32 architectures provide hardware support for marking page tables as *NX* ("no execute"). This feature is used to prevent execution of program code on the stack, thus making stack crashing more difficult.

There are safe alternatives to many of the functions mentioned above—for example, *snprintf()*, *strncpy()*, and *strncat()*—that allow the caller to specify the maximum number of characters that should be copied. These functions take the specified maximum into account in order to avoid overrunning the target buffer. In general, these alternatives are preferable, but must still be handled with care. In particular, note the following points:

- With most of these functions, if the specified maximum is reached, then a truncated version of the source string is placed in the target buffer. Since such a truncated string may be meaningless in terms of the semantics of the program, the caller must check if truncation occurred (e.g., using the return value from *snprintf()*), and take appropriate action if it has.
- Using *strncpy()* can carry a performance impact. If, in the call *strncpy(s1, s2, n)*, the string pointed to by *s2* is less than *n* bytes in length, then padding null bytes are written to *s1* to ensure that *n* bytes in total are written.
- If the maximum size value given to *strncpy()* is not long enough to permit the inclusion of the terminating null character, then the target string is *not* null-terminated.

Some UNIX implementations provide the *strlcpy()* function, which, given a length argument *n*, copies at most *n − 1* bytes to the destination buffer and always appends a null character at the end of the buffer. However, this function is not specified in SUSv3 and is not implemented in *glibc*. Furthermore, in cases where the caller is not carefully checking string lengths, this function only substitutes one problem (buffer overflows) for another (silently discarding data).

38.10 Beware of Denial-of-Service Attacks

With the increase in Internet-based services has come a corresponding increase in the opportunities for remote denial-of-service attacks. These attacks attempt to make a service unavailable to legitimate clients, either by sending the server malformed data that causes it to crash or by overloading it with bogus requests.

Local denial-of-service attacks are also possible. The most well-known example is when a user runs a simple fork bomb (a program that repeatedly forks, thus consuming all of the process slots on the system). However, the origin of local denial-of-service attacks is much easier to determine, and they can generally be prevented by suitable physical and password security measures.

Dealing with malformed requests is straightforward—a server should be programmed to rigorously check its inputs and avoid buffer overruns, as described above.

Overload attacks are more difficult to deal with. Since the server can't control the behavior of remote clients or the rate at which they submit requests, such attacks are impossible to prevent. (The server may not even be able to determine the true origin of the attack, since the source IP address of a network packet can be spoofed. Alternatively, distributed attacks may enlist unwitting intermediary hosts to direct an attack at a target system.) Nevertheless, various measures can be taken to minimize the risk and consequences of an overload attack:

- The server should perform load throttling, dropping requests when the load exceeds some predetermined limit. This will have the consequence of dropping legitimate requests, but prevents the server and host machine from becoming overloaded. The use of resource limits and disk quotas may also be helpful in limiting excessive loads. (Refer to *http://sourceforge.net/projects/linuxquota/* for more information on disk quotas.)
- A server should employ timeouts for communication with a client, so that if the client (perhaps deliberately) doesn't respond, the server is not tied up indefinitely waiting on the client.
- In the event of an overload, the server should log suitable messages so that the system administrator is notified of the problem. (However, logging should be throttled, so that logging itself does not overload the system.)
- The server should be programmed so that it doesn't crash in the face of an unexpected load. For example, bounds checking should be rigorously performed to ensure that excessive requests don't overflow a data structure.
- Data structures should be designed to avoid *algorithmic-complexity attacks*. For example, a binary tree may be balanced and deliver acceptable performance under typical loads. However, an attacker could construct a sequence of inputs that result in an unbalanced tree (the equivalent of a linked list in the worst case), which could cripple performance. [Crosby & Wallach, 2003] details the nature of such attacks and discusses data-structuring techniques that can be used to avoid them.

38.11 Check Return Statuses and Fail Safely

A privileged program should always check to see whether system calls and library functions succeed, and whether they return expected values. (This is true for all programs, of course, but is especially important for privileged programs.) Various system calls can fail, even for a program running as *root*. For example, *fork()* may fail if the system-wide limit on the number of processes is encountered, an *open()* for writing may fail on a read-only file system, or *chdir()* may fail if the target directory doesn't exist.

Even where a system call succeeds, it may be necessary to check its result. For example, where it matters, a privileged program should check that a successful *open()* has not returned one of the three standard file descriptors: 0, 1, or 2.

Finally, if a privileged program encounters an unexpected situation, then the appropriate behavior is usually either to terminate or, in the case of a server, to drop the client request. Attempting to fix unexpected problems typically requires making assumptions that may not be justified in all circumstances and may lead to the creation of security loopholes. In such situations, it is safer to have the program terminate, or to have the server log a message and discard the client's request.

38.12 Summary

Privileged programs have access to system resources that are not available to ordinary users. If such programs can be subverted, then the security of the system can be compromised. In this chapter, we presented a set of guidelines for writing privileged programs. The aim of these guidelines is twofold: to minimize the chances of a privileged program being subverted, and to minimize the damage that can be done in the event that a privileged program is subverted.

Further information

[Viega & McGraw, 2002] covers a broad range of topics relating to the design and implementation of secure software. General information about security on UNIX systems, as well as a chapter on secure-programming techniques can be found in [Garfinkel et al., 2003]. Computer security is covered at some length in [Bishop, 2005], and at even greater length by the same author in [Bishop, 2003]. [Peikari & Chuvakin, 2004] describes computer security with a focus on the various means by which system may be attacked. [Erickson, 2008] and [Anley, 2007] both provide a thorough discussion of various security exploits, providing enough detail for wise programmers to avoid these exploits. [Chen et al., 2002] is a paper describing and analyzing the UNIX set-user-ID model. [Tsafrir et al., 2008] revises and enhances the discussion of various points in [Chen et al., 2002]. [Drepper, 2009] provides a wealth of tips on secure and defensive programming on Linux.

Several sources of information about writing secure programs are available online, including the following:

- Matt Bishop has written a range of security-related papers, which are available online at *http://nob.cs.ucdavis.edu/~bishop/secprog*. The most interesting of these is "How to Write a Setuid Program," (originally published in *;login: 12(1) Jan/Feb 1986*). Although somewhat dated, this paper contains a wealth of useful tips.
- The *Secure Programming for Linux and Unix HOWTO*, written by David Wheeler, is available at *http://www.dwheeler.com/secure-programs/*.
- A useful checklist for writing set-user-ID programs is available online at *http://www.homeport.org/~adam/setuid.7.html*.

38.13 Exercises

38-1. Log in as a normal, unprivileged user, create an executable file (or copy an existing file such as `/bin/sleep`), and enable the set-user-ID permission bit on that file (*chmod u+s*). Try modifying the file (e.g., *cat >> file*). What happens to the file permissions as a result (*ls −l*)? Why does this happen?

38-2. Write a set-user-ID-*root* program similar to the *sudo(8)* program. This program should take command-line options and arguments as follows:

```
$ ./douser [-u user ] program-file arg1 arg2 ...
```

The *douser* program executes *program-file*, with the given arguments, as though it was run by *user*. (If the *−u* option is omitted, then *user* should default to *root*.) Before executing *program-file*, *douser* should request the password for *user*, authenticate it against the standard password file (see Listing 8-2, on page 164), and then set all of the process user and group IDs to the correct values for that user.