10 Security

MUCH OF THE POWER OF A GNU/LINUX SYSTEM COMES FROM its support for multiple users and for networking. Many people can use the system at once, and they can connect to the system from remote locations. Unfortunately, with this power comes risk, especially for systems connected to the Internet. Under some circumstances, a remote "hacker" can connect to the system and read, modify, or remove files that are stored on the machine. Or, two users on the same machine can read, modify, or remove each other's files when they should not be allowed to do so. When this happens, the system's security is said to have been *compromised*.

The Linux kernel provides a variety of facilities to ensure that these events do not take place. But to avoid security breaches, ordinary applications must be careful as well. For example, imagine that you are developing accounting software. Although you might want all users to be able to file expense reports with the system, you wouldn't want all users to be able to *approve* those reports. You might want users to be able to view their own payroll information, but you certainly wouldn't want them to be able to view everyone else's payroll information. You might want managers to be able to view the salaries of employees in their departments, but you wouldn't want them to view the salaries of employees in other departments.

To enforce these kinds of controls, you have to be very careful. It's amazingly easy to make a mistake that allows users to do something you didn't intend them to be able to do. The best approach is to enlist the help of security experts. Still, every application developer ought to understand the basics.

10.1 Users and Groups

Each Linux user is assigned a unique number, called a *user ID*, or *UID*. Of course, when you log in, you use a username rather than a user ID. The system converts your username to a particular user ID, and from then on it's only the user ID that counts.

You can actually have more than one username for the same user ID. As far as the system is concerned, the user IDs, not the usernames, matter. There's no way to give one username more power than another if they both correspond to the same user ID.

You can control access to a file or other resource by associating it with a particular user ID. Then only the user corresponding to that user ID can access the resource. For example, you can create a file that only you can read, or a directory in which only you can create new files. That's good enough for many simple cases.

Sometimes, however, you want to share a resource among multiple users. For example, if you're a manager, you might want to create a file that any manager can read but that ordinary employees cannot. Linux doesn't allow you to associate multiple user IDs with a file, so you can't just create a list of all the people to whom you want to give access and attach them all to the file.

You can, however, create a *group*. Each group is assigned a unique number, called a *group ID*, or *GID*. Every group contains one or more user IDs. A single user ID can be a member of lots of groups, but groups can't contain other groups; they can contain only users. Like users, groups have names. Also like usernames, however, the group names don't really matter; the system always uses the group ID internally.

Continuing our example, you could create a `managers` group and put the user IDs for all the managers in this group. You could then create a file that can be read by anyone in the `managers` group but not by people who aren't in the group. In general, you can associate only one group with a resource. There's no way to specify that users can access a file only if they're in either group 7 or group 42, for example.

If you're curious to see what your user ID is and what groups you are in, you can use the `id` command. For example, the output might look like this:

```
% id
uid=501(mitchell) gid=501(mitchell) groups=501(mitchell),503(csl)
```

The first part shows you that the user ID for the user who ran the command was 501. The command also figures out what the corresponding username is and displays that in parentheses. The command shows that user ID 501 is actually in two groups: group 501 (called `mitchell`) and group 503 (called `csl`). You're probably wondering why group 501 appears twice: once in the `gid` field and once in the `groups` field. We'll explain this later.

10.1.1 The Superuser

One user account is very special.[1] This user has user ID 0 and usually has the username `root`. It is also sometimes referred to as the *superuser* account. The `root` user can do just about anything: read any file, remove any file, add new users, turn off network access, and so forth. Lots of special operations can be performed only by processes running with root privilege—that is, running as user `root`.

The trouble with this design is that a lot of programs need to be run by `root` because a lot of programs need to perform one of these special operations. If any of these programs misbehaves, chaos can result. There's no effective way to contain a program when it's run by `root`; it can do *anything*. Programs run by root must be written very carefully.

10.2 Process User IDs and Process Group IDs

Until now, we've talked about commands being executed by a particular user. That's not quite accurate because the computer never really knows which user is using it. If Eve learns Alice's username and password, then Eve can log in as Alice, and the computer will let Eve do everything that Alice can do. The system knows only which user ID is in use, not which user is typing the commands. If Alice can't be trusted to keep her password to herself, for example, then nothing you do as an application developer will prevent Eve from accessing Alice's files. The responsibility for system security is shared among the application developer, the users of the system, and the administrators of the system.

Every process has an associated user ID and group ID. When you invoke a command, it typically runs in a process whose user and group IDs are the same as your user and group IDs. When we say that a user performs an operation, we really mean that a process with the corresponding user ID performs that operation. When the process makes a system call, the kernel decides whether to allow the operation to proceed. It makes that determination by examining the permissions associated with the resources that the process is trying to access and by checking the user ID and group ID associated with the process trying to perform the action.

Now you know what that middle field printed by the `id` command is all about. It's showing the group ID of the current process. Even though user 501 is in multiple groups, the current process can have only one group ID. In the example shown previously, the current group ID is 501.

If you have to manipulate user IDs and group IDs in your program (and you will, if you're writing programs that deal with security), then you should use the `uid_t` and `gid_t` types defined in `<sys/types.h>`. Even though user IDs and group IDs are essentially just integers, avoid making any assumptions about how many bits are used in these types or perform arithmetic operations on them. Just treat them as opaque handles for user and group identity.

1. The fact that there is only one special user gave AT&T the name for its UNIX operating system. In contrast, an earlier operating system that had multiple special users was called MULTICS. GNU/Linux, of course, is mostly compatible with UNIX.

To get the user ID and group ID for the current process, you can use the `geteuid` and `getegid` functions, declared in `<unistd.h>`. These functions don't take any parameters, and they always work; you don't have to check for errors. Listing 10.1 shows a simple program that provides a subset of the functionality provide by the `id` command:

Listing 10.1 **(*simpleid.c*) Print User and Group IDs**

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
  uid_t uid = geteuid ();
  gid_t gid = getegid ();
  printf ("uid=%d gid=%d\n", (int) uid, (int) gid);
  return 0;
}
```

When this program is run (by the same user who ran the real `id` program) the output is as follows:

```
% ./simpleid
uid=501 gid=501
```

10.3 File System Permissions

A good way to see users and groups in action is to look at file system permissions. By examining how the system associates permissions with each file and then seeing how the kernel checks to see who is allowed to access which files, the concepts of user ID and group ID should become clearer.

Each file has exactly one *owning user* and exactly one *owning group*. When you create a new file, the file is owned by the user and group of the creating process.[2]

The basic things that you can do with files, as far as Linux is concerned, are *read* from them, *write* to them, and *execute* them. (Note that creating a file and removing a file are not considered things you can do with the file; they're considered things you can do with the directory containing the file. We'll get to this a little later.) If you can't read a file, Linux won't let you examine the file's contents. If you can't write a file, you can't change its contents. If there's a program file for which you do not have execute permission, you cannot run the program.

2. Actually, there are some rare exceptions, involving *sticky bits*, discussed later in Section 10.3.2, "Sticky Bits."

Linux enables you to designate which of these three actions—reading, writing, and executing—can be performed by the owning user, owning group, and everybody else. For example, you could say that the owning user can do anything she wants with the file, that anyone in the owning group can read and execute the file (but not write to it), and that nobody else can access the file at all.

You can view these *permission bits* interactively with the `ls` command by using the `-l` or `-o` options and programmatically with the `stat` system call. You can set the *permission bits* interactively with the `chmod` program[3] or programmatically with the system call of the same name. To look at the permissions on a file named `hello`, use `ls -l hello`. Here's how the output might look:

```
% ls -l hello
-rwxr-x---    1 samuel   csl         11734 Jan 22 16:29 hello
```

The `samuel` and `csl` fields indicate that the owning user is `samuel` and that the owning group is `csl`.

The string of characters at the beginning of the line indicates the permissions associated with the file. The first dash indicates that this is a normal file. It would be `d` for a directory, or it can be other letters for special kinds of files such as devices (see Chapter 6, "Devices") or named pipes (see Chapter 5, "Interprocess Communication," Section 5.4, "Pipes"). The next three characters show permissions for the owning user; they indicate that `samuel` can read, write, and execute the file. The next three characters show permissions for members of the `csl` group; these members are allowed only to read and execute the file. The last three characters show permissions for everyone else; these users are not allowed to do anything with `hello`.

Let's see how this works. First, let's try to access the file as the user `nobody`, who is not in the `csl` group:

```
% id
uid=99(nobody) gid=99(nobody) groups=99(nobody)
% cat hello
cat: hello: Permission denied
% echo hi > hello
sh: ./hello: Permission denied
% ./hello
sh: ./hello: Permission denied
```

We can't read the file, which is why `cat` fails; we can't write to the file, which is why `echo` fails; and we can't run the file, which is why `./hello` fails.

3. You'll sometimes see the permission bits for a file referred to as the file's *mode*. The name of the `chmod` command is short for "change mode."

Things are better if we are accessing the file as `mitchell`, who is a member of the `csl` group:

```
% id
uid=501(mitchell) gid=501(mitchell) groups=501(mitchell),503(csl)
% cat hello
#!/bin/bash
echo "Hello, world."
% ./hello
Hello, world.
% echo hi > hello
bash: ./hello: Permission denied
```

We can list the contents of the file, and we can run it (it's a simple shell script), but we still can't write to it.

If we run as the owner (`samuel`), we can even overwrite the file:

```
% id
uid=502(samuel) gid=502(samuel) groups=502(samuel),503(csl)
% echo hi > hello
% cat hello
hi
```

You can change the permissions associated with a file only if you are the file's owner (or the superuser). For example, if you now want to allow everyone to execute the file, you can do this:

```
% chmod o+x hello
% ls -l hello
-rwxr-x--x    1 samuel   csl             3 Jan 22 16:38 hello
```

Note that there's now an `x` at the end of the first string of characters. The `o+x` bit means that you want to add the execute permission for other people (not the file's owner or members of its owning group). You could use `g-w` instead, to remove the write permission from the group. See the man page in section 1 for `chmod` for details about this syntax:

```
% man 1 chmod
```

Programmatically, you can use the `stat` system call to find the permissions associated with a file. This function takes two parameters: the name of the file you want to find out about, and the address of a data structure that is filled in with information about the file. See Appendix B, "Low-Level I/O," Section B.2, "`stat`," for a discussion of other information that you can obtain with `stat`. Listing 10.2 shows an example of using `stat` to obtain file permissions.

Listing 10.2 **(*stat-perm.c*) Determine File Owner's Write Permission**

```
#include <stdio.h>
#include <sys/stat.h>

int main (int argc, char* argv[])
{
  const char* const filename = argv[1];
  struct stat buf;
```

```
    /* Get file information.  */
    stat (filename, &buf);
    /* If the permissions are set such that the file's owner can write
       to it, print a message.  */
    if (buf.st_mode & S_IWUSR)
      printf ("Owning user can write `%s'.\n", filename);
    return 0;
  }
```

If you run this program on our `hello` program, it says:

```
% ./stat-perm hello
Owning user can write 'hello'.
```

The `S_IWUSR` constant corresponds to write permission for the owning user. There are other constants for all the other bits. For example, `S_IRGRP` is read permission for the owning group, and `S_IXOTH` is execute permission for users who are neither the owning user nor a member of the owning group. If you store permissions in a variable, use the typedef `mode_t` for that variable. Like most system calls, `stat` will return `-1` and set `errno` if it can't obtain information about the file.

You can use the `chmod` function to change the permission bits on an existing file. You call `chmod` with the name of the file you want to change and the permission bits you want set, presented as the bitwise or of the various permission constants mentioned previously. For example, this next line would make `hello` readable and executable by its owning user but would disable all other permissions associated with `hello`:

```
chmod ("hello", S_IRUSR | S_IXUSR);
```

The same permission bits apply to directories, but they have different meanings. If a user is allowed to read from a directory, the user is allowed to see the list of files that are present in that directory. If a user is allowed to write to a directory, the user is allowed to add or remove files from the directory. Note that a user may remove files from a directory if she is allowed to write to the directory, *even if she does not have permission to modify the file she is removing*. If a user is allowed to execute a directory, the user is allowed to enter that directory and access the files therein. Without execute access to a directory, a user is not allowed to access the files in that directory independent of the permissions on the files themselves.

To summarize, let's review how the kernel decides whether to allow a process to access a particular file. It checks to see whether the accessing user is the owning user, a member of the owning group, or someone else. The category into which the accessing user falls is used to determine which set of read/write/execute bits are checked. Then the kernel checks the operation that is being performed against the permission bits that apply to this user.[4]

4. The kernel may also deny access to a file if a component directory in its file path is inaccessible. For instance, if a process may not access the directory `/tmp/private/`, it may not read `/tmp/private/data`, even if the permissions on the latter are set to allow the access.

There is one important exception: Processes running as `root` (those with user ID 0) are always allowed to access any file, regardless of the permissions associated with it.

10.3.1 Security Hole: Programs Without Execute Permissions

Here's a first example of where security gets very tricky. You might think that if you disallow execution of a program, then nobody can run it. After all, that's what it means to disallow execution. But a malicious user can make a copy of the program, change the permissions to make it executable, and then run the copy! If you rely on users not being able to run programs that aren't executable but then don't prevent them from copying the programs, you have a *security hole*—a means by which users can perform some action that you didn't intend.

10.3.2 Sticky Bits

In addition to read, write, and execute permissions, there is a magic bit called the *sticky bit*.[5] This bit applies only to directories.

A directory that has the sticky bit set allows you to delete a file only if you are the owner of the file. As mentioned previously, you can ordinarily delete a file if you have write access to the directory that contains it, even if you are not the file's owner. When the sticky bit is set, you *still* must have write access to the directory, but you must also be the owner of the file that you want to delete.

A few directories on the typical GNU/Linux system have the sticky bit set. For example, the `/tmp` directory, in which any user can place temporary files, has the sticky bit set. This directory is specifically designed to be used by all users, so the directory must be writable by everyone. But it would be bad if one user could delete another user's files, so the sticky bit is set on the directory. Then only the owning user (or `root`, of course) can remove a file.

You can see the sticky bit is set because of the `t` at the end of the permission bits when you run `ls` on `/tmp`:

```
% ls -ld /tmp
drwxrwxrwt   12 root     root         2048 Jan 24 17:51 /tmp
```

The corresponding constant to use with `stat` and `chmod` is `S_ISVTX`.

If your program creates directories that behave like `/tmp`, in that lots of people put things there but shouldn't be able to remove each other's files, then you should set the sticky bit on the directory. You can set the sticky bit on a directory with the `chmod` command by invoking the following:

```
% chmod o+t directory
```

5. This name is anachronistic; it goes back to a time when setting the sticky bit caused a program to be retained in main memory even when it was done executing. The pages allocated to the program were "stuck" in memory.

To set the sticky bit programmatically, call `chmod` with the `S_ISVTX` mode flag. For example, to set the sticky bit of the directory specified by `dir_path` to those of the `/tmp` and give full read, write, and execute permissions to all users, use this call:

```
chmod (dir_path, S_IRWXU | S_IRWXG | S_IRWXO | S_ISVTX);
```

10.4 Real and Effective IDs

Until now, we've talked about the user ID and group ID associated with a process as if there were only one such user ID and one such group ID. But, actually, it's not quite that simple.

Every process really has two user IDs: the *effective user ID* and the *real user ID*. (Of course, there's also an *effective group ID* and *real group ID*. Just about everything that's true about user IDs is also true about group IDs.) Most of the time, the kernel checks only the effective user ID. For example, if a process tries to open a file, the kernel checks the effective user ID when deciding whether to let the process access the file.

The `geteuid` and `getegid` functions described previously return the effective user ID and the effective group ID. Corresponding `getuid` and `getgid` functions return the real user ID and real group ID.

If the kernel cares about only the effective user ID, it doesn't seem like there's much point in having a distinction between a real user ID and an effective user ID. However, there is one very important case in which the real user ID matters. If you want to change the effective user ID of an already running process, the kernel looks at the real user ID as well as the effective user ID.

Before looking at *how* you can change the effective user ID of a process, let's examine *why* you would want to do such a thing by looking back at our accounting package. Suppose that there's a server process that might need to look at any file on the system, regardless of the user who created it. Such a process must run as `root` because only `root` can be guaranteed to be capable of looking at any file. But now suppose that a request comes in from a particular user (say, `mitchell`) to access some file. The server process could carefully examine the permissions associated with the files in question and try to decide whether `mitchell` should be allowed to access those files. But that would mean duplicating all the processing that the kernel would normally do to check file access permissions. Reimplementing that logic would be complex, error-prone, and tedious.

A better approach is simply to temporarily change the effective user ID of the process from `root` to `mitchell` and then try to perform the operations required. If `mitchell` is not allowed to access the data, the kernel will prevent the process from doing so and will return appropriate indications of error. After all the operations taken on behalf of `mitchell` are complete, the process can restore its original effective user ID to `root`.

Programs that authenticate users when they log in take advantage of the capability to change user IDs as well. These login programs run as `root`. When the user enters a username and password, the login program verifies the username and password in the system password database. Then the login program changes both the effective user ID and the real ID to be that of the user. Finally, the login program calls `exec` to start the user's shell, leaving the user running a shell whose effective user ID and real user ID are that of the user.

The function used to change the user IDs for a process is `setreuid`. (There is, of course, a corresponding `setregid` function as well.) This function takes two arguments. The first argument is the desired real user ID; the second is the desired effective user ID. For example, here's how you would exchange the effective and real user IDs:

```
setreuid (geteuid(), getuid ());
```

Obviously, the kernel won't let just any process change its user IDs. If a process were allowed to change its effective user ID at will, then any user could easily impersonate any other user, simply by changing the effective user ID of one of his processes. The kernel will let a process running with an effective user ID of 0 change its user IDs as it sees fit. (Again, notice how much power a process running as `root` has! A process whose effective user ID is 0 can do absolutely anything it pleases.) Any other process, however, can do only one of the following things:

- Set its effective user ID to be the same as its real user ID
- Set its real user ID to be the same as its effective user ID
- Swap the two user IDs

The first alternative would be used by our accounting process when it has finished accessing files as `mitchell` and wants to return to being `root`. The second alternative could be used by a login program after it has set the effective user ID to that of the user who just logged in. Setting the real user ID ensures that the user will never be able go back to being `root`. Swapping the two user IDs is almost a historical artifact; modern programs rarely use this functionality.

You can pass -1 to either argument to `setreuid` if you want to leave that user ID alone. There's also a convenience function called `seteuid`. This function sets the effective user ID, but it doesn't modify the real user ID. The following two statements both do exactly the same thing:

```
seteuid (id);
setreuid (-1, id);
```

10.4.1 Setuid Programs

Using the previous techniques, you know how to make a `root` process impersonate another process temporarily and then return to being `root`. You also know how to make a `root` process drop all its special privileges by setting both its real user ID and its effective user ID.

Here's a puzzle: Can you, running as a non-`root` user, ever become `root`? That doesn't seem possible, using the previous techniques, but here's proof that it can be done:

```
% whoami
mitchell
% su
Password: ...
% whoami
root
```

The `whoami` command is just like `id`, except that it shows only the effective user ID, not all the other information. The `su` command enables you to become the superuser if you know the `root` password.

How does `su` work? Because we know that the shell was originally running with both its real user ID and its effective user ID set to `mitchell`, `setreuid` won't allow us to change either user ID.

The trick is that the `su` program is a *setuid* program. That means that when it is run, the effective user ID of the process will be that of the file's owner rather than the effective user ID of the process that performed the `exec` call. (The real user ID will still be that of the executing user.) To create a setuid program, you use `chmod +s` at the command line, or use the `S_ISUID` flag if calling `chmod` programmatically.[6]

For example, consider the program in Listing 10.3.

Listing 10.3 **(*setuid-test.c*) Setuid Demonstration Program**

```c
#include <stdio.h>
#include <unistd.h>

int main ()
{
  printf ("uid=%d euid=%d\n", (int) getuid (), (int) geteuid ());
  return 0;
}
```

Now suppose that this program is setuid and owned by `root`. In that case, the `ls` output will look like this:

```
-rwsrws--x    1 root     root        11931 Jan 24 18:25 setuid-test
```

The `s` bits indicate that the file is not only executable (as an `x` bit would indicate) but also setuid and setgid. When we use this program, we get output like this:

```
% whoami
mitchell
% ./setuid-test
uid=501 euid=0
```

6. Of course, there is a similar notion of a setgid program. When run, its effective group ID is the same as that of the group owner of the file. Most setuid programs are also setgid programs.

Note that the effective user ID is set to 0 when the program is run.

You can use the `chmod` command with the `u+s` or `g+s` arguments to set the setuid and setgid bits on an executable file, respectively—for example:

```
% ls -l program
-rwxr-xr-x    1 samuel   csl             0 Jan 30 23:38 program
% chmod g+s program
% ls -l program
-rwxr-sr-x    1 samuel   csl             0 Jan 30 23:38 program
% chmod u+s program
% ls -l program
-rwsr-sr-x    1 samuel   csl             0 Jan 30 23:38 program
```

You can also use the `chmod` call with the `S_ISUID` or `S_ISGID` mode flags.

`su` is capable of changing the effective user ID through this mechanism. It runs initially with an effective user ID of 0. Then it prompts you for a password. If the password matches the `root` password, it sets its real user ID to be `root` as well and then starts a new shell. Otherwise, it exits, unceremoniously leaving you as a non-privileged user.

Take a look at the permissions on the `su` program:

```
% ls -l /bin/su
-rwsr-xr-x    1 root     root        14188 Mar  7  2000 /bin/su
```

Notice that it's owned by `root` and that the setuid bit is set.

Note that `su` doesn't actually change the user ID of the shell from which it was run. Instead, it starts a new shell process with the new user ID. The original shell is blocked until the new shell completes and `su` exits.

10.5 Authenticating Users

Often, if you have a setuid program, you don't want to offer its services to everyone. For example, the `su` program lets you become `root` only if you know the `root` password. The program makes you prove that you are entitled to become `root` before going ahead with its actions. This process is called *authentication*—the `su` program is checking to see that you are authentic.

If you're administering a very secure system, you probably don't want to let people log in just by typing an ordinary password. Users tend to write down passwords, and black hats tend to find them. Users tend to pick passwords that involve their birthdays, the names of their pets, and so forth.[7] Passwords just aren't all that secure.

7. It has been found that system administrators tend to pick the word *god* as their password more often than any other password. (Make of that what you will.) So, if you ever need `root` access on a machine and the sysadmin isn't around, a little divine inspiration might be just what you need.

For example, many organizations now require the use of special "one-time" passwords that are generated by special electronic ID cards that users keep with them. The same password can't be used twice, and you can't get a valid password out of the ID card without entering a PIN. So, an attacker must obtain both the physical card and the PIN to break in. In a really secure facility, retinal scans or other kinds of biometric testing are used.

If you're writing a program that must perform authentication, you should allow the system administrator to use whatever means of authentication is appropriate for that installation. GNU/Linux comes with a very useful library that makes this very easy. This facility, called *Pluggable Authentication Modules*, or *PAM*, makes it easy to write applications that authenticate their users as the system administrator sees fit.

It's easiest to see how PAM works by looking at a simple PAM application. Listing 10.4 illustrates the use of PAM.

Listing 10.4 **(*pam.c*) PAM Example**

```
#include <security/pam_appl.h>
#include <security/pam_misc.h>
#include <stdio.h>

int main ()
{
  pam_handle_t* pamh;
  struct pam_conv pamc;

  /* Set up the PAM conversation.  */
  pamc.conv = &misc_conv;
  pamc.appdata_ptr = NULL;
  /* Start a new authentication session.  */
  pam_start ("su", getenv ("USER"), &pamc, &pamh);
  /* Authenticate the user.  */
  if (pam_authenticate (pamh, 0) != PAM_SUCCESS)
    fprintf (stderr, "Authentication failed!\n");
  else
    fprintf (stderr, "Authentication OK.\n");
  /* All done.  */
  pam_end (pamh, 0);
  return 0;
}
```

To compile this program, you have to link it with two libraries: the `libpam` library and a helper library called `libpam_misc`:

```
% gcc -o pam pam.c -lpam -lpam_misc
```

This program starts off by building up a PAM *conversation object*. This object is used by the PAM library whenever it needs to prompt the user for information. The `misc_conv` function used in this example is a standard conversation function that uses the terminal for input and output. You could write your own function that pops up a dialog box, or that uses speech for input and output, or that provides even more exotic input and output methods.

The program then calls `pam_start`. This function initializes the PAM library. The first argument is a service name. You should use a name that uniquely identifies your application. For example, if your application is named whizbang, you should probably use that for the service name, too. However, the program probably won't work until the system administrator explicitly configures the system to work with your service. So, in this example, we use the su service, which says that our program should authenticate users in the same way that the `su` command does. You should *not* use this technique in a real program. Pick a real service name, and have your installation scripts help the system administrator to set up a correct PAM configuration for your application.

The second argument is the name of the user whom you want to authenticate. In this example, we use the value of the `USER` environment variable. (Normally, this is the username that corresponds to the effective user ID of the current process, but that's not always the case.) In most real programs, you would prompt for a username at this point. The third argument indicates the PAM conversation, discussed previously. The call to `pam_start` fills in the handle provided as the fourth argument. Pass this handle to subsequent calls to PAM library routines.

Next, the program calls `pam_authenticate`. The second argument enables you to pass various flags; the value 0 means to use the default options. The return value from this function indicates whether authentication succeeded.

Finally, the programs calls `pam_end` to clean up any allocated data structures.

Let's assume that the valid password for the current user is "password" (an exceptionally poor password). Then, running this program with the correct password produces the expected:

```
% ./pam
Password: password

Authentication OK.
```

If you run this program in a terminal, the password probably won't actually appear when you type it in; it's hidden to prevent others from peeking at your password over your shoulder as you type.

However, if a hacker tries to use the wrong password, the PAM library will correctly indicate failure:

```
% ./pam
Password: badguess

Authentication failed!
```

The basics covered here are enough for most simple programs. Full documentation about how PAM works is available in `/usr/doc/pam` on most GNU/Linux systems.

10.6 More Security Holes

Although this chapter will point out a few common security holes, you should by no means rely on this book to cover all possible security holes. A great many have already been discovered, and many more are out there waiting to be found. If you are trying to write secure code, there is really no substitute for having a security expert audit your code.

10.6.1 Buffer Overruns

Almost every major Internet application daemon, including the `sendmail` daemon, the `finger` daemon, the `talk` daemon, and others, has at one point been compromised through a *buffer overrun*.

If you are writing any code that will ever be run as `root`, you absolutely must be aware of this particular kind of security hole. If you are writing a program that performs any kind of interprocess communication, you should definitely be aware of this kind of security hole. If you are writing a program that reads files (or *might* read files) that are not owned by the user executing the program, you should be aware of this kind of security hole. That last criterion applies to almost every program. Fundamentally, if you're going to write GNU/Linux software, you ought to know about buffer overruns.

The idea behind a buffer overrun attack is to trick a program into executing code that it did not intend to execute. The usual mechanism for achieving this feat is to overwrite some portion of the program's process stack. The program's stack contains, among other things, the memory location to which the program will transfer control when the current function returns. Therefore, if you can put the code that you want to have executed into memory somewhere and then change the return address to point to that piece of memory, you can cause the program to execute anything. When the program returns from the function it is executing, it will jump to the new code and execute whatever is there, running with the privileges of the current process. Clearly, if the current process is running as `root`, this would be a disaster. If the process is running as another user, it's a disaster "only" for that user—and anybody else who depends on the contents of files owned by that user, and so forth.

If the program is running as a daemon and listening for incoming network connections, the situation is even worse. A daemon typically runs as `root`. If it contains buffer overrun bugs, anyone who can connect via the network to a computer running the daemon can seize control of the computer by sending a malignant sequence of data to the daemon over the network. A program that does not engage in network communications is much safer because only users who are already able to log in to the computer running the program are able to attack it.

The buggy versions of `finger`, `talk`, and `sendmail` all shared a common flaw. Each used a fixed-length string buffer, which implied a constant upper limit on the size of the string but then allowed network clients to provide strings that overflowed the buffer. For example, they contained code similar to this:

```
#include <stdio.h>

int main ()
{
  /* Nobody in their right mind would have more than 32 characters in
     their username.  Plus, I think UNIX allows only 8-character
     usernames.  So, this should be plenty of space.  */
  char username[32];
  /* Prompt the user for the username.  */
  printf ("Enter your username: ");
  /* Read a line of input.  */
  gets (username);
  /* Do other things here...  */

  return 0;
}
```

The combination of the 32-character buffer with the `gets` function permits a buffer overrun. The `gets` function reads user input up until the next newline character and stores the entire result in the `username` buffer. The comments in the code are correct in that people generally have short usernames, so no well-meaning user is likely to type in more than 32 characters. But when you're writing secure software, you must consider what a malicious attacker might do. In this case, the attacker might deliberately type in a very long username. Local variables such as `username` are stored on the stack, so by exceeding the array bounds, it's possible to put arbitrary bytes onto the stack beyond the area reserved for the `username` variable. The username will overrun the buffer and overwrite parts of the surrounding stack, allowing the kind of attack described previously.

Fortunately, it's relatively easy to prevent buffer overruns. When reading strings, you should always use a function, such as `getline`, that either dynamically allocates a sufficiently large buffer or stops reading input if the buffer is full. For example, you could use this:

```
char* username = getline (NULL, 0, stdin);
```

This call automatically uses `malloc` to allocate a buffer big enough to hold the line and returns it to you. You have to remember to call `free` to deallocate the buffer, of course, to avoid leaking memory.

Your life will be even easier if you use C++ or another language that provides simple primitives for reading input. In C++, for example, you can simply use this:

```
string username;
getline (cin, username);
```

The `username` string will automatically be deallocated as well; you don't have to remember to `free` it.[8]

Of course, buffer overruns can occur with any statically sized array, not just with strings. If you want to write secure code, you should never write into a data structure, on the stack or elsewhere, without verifying that you're not going to write beyond its region of memory.

10.6.2 Race Conditions in */tmp*

Another very common problem involves the creation of files with predictable names, typically in the `/tmp` directory. Suppose that your program `prog`, running as `root`, always creates a temporary file called `/tmp/prog` and writes some vital information there. A malicious user can create a symbolic link from `/tmp/prog` to any other file on the system. When your program goes to create the file, the `open` system call will succeed. However, the data that you write will not go to `/tmp/prog`; instead, it will be written to some arbitrary file of the attacker's choosing.

This kind of attack is said to exploit a *race condition*. There is implicitly a race between you and the attacker. Whoever manages to create the file first wins.

This attack is often used to destroy important parts of the file system. By creating the appropriate links, the attacker can trick a program running as `root` that is supposed to write a temporary file into overwriting an important system file instead. For example, by making a symbolic link to `/etc/passwd`, the attacker can wipe out the system's password database. There are also ways in which a malicious user can obtain `root` access using this technique.

One attempt at avoiding this attack is to use a randomized name for the file. For example, you could read from `/dev/random` to get some bits to use in the name of the file. This certainly makes it harder for a malicious user to guess the filename, but it doesn't make it impossible. The attacker might just create a large number of symbolic links, using many potential names. Even if she has to try 10,000 times before wining the race condition, that one time could be disastrous.

Another approach is to use the `O_EXCL` flag when calling `open`. This flag causes `open` to fail if the file already exists. Unfortunately, if you're using the Network File System (NFS), or if anyone who's using your program might ever be using NFS, that's not a sufficiently robust approach because `O_EXCL` is not reliable when NFS is in use. You can't ever really know for sure whether your code will be used on a system that uses NFS, so if you're highly paranoid, don't rely on using `O_EXCL`.

In Chapter 2, "Writing Good GNU/Linux Software," Section 2.1.7, "Using Temporary Files," we showed how to use `mkstemp` to create temporary files. Unfortunately, what `mkstemp` does on Linux is open the file with `O_EXCL` after trying to pick a name that is hard to guess. In other words, using `mkstemp` is still insecure if `/tmp` is mounted over NFS.[9] So, using `mkstemp` is better than nothing, but it's not fully secure.

8. Some programmers believe that C++ is a horrible and overly complex language. Their arguments about multiple inheritance and other such complications have some merit, but it is easier to write code that avoids buffer overruns and other similar problems in C++ than in C.

9. Obviously, if you're also a system administrator, you shouldn't mount `/tmp` over NFS.

One approach that works is to call `lstat` on the newly created file (lstat is discussed in Section B.2, "`stat`"). The `lstat` function is like `stat`, except that if the file referred to is a symbolic link, `lstat` tells you about the link, not the file to which it refers. If `lstat` tells you that your new file is an ordinary file, not a symbolic link, and that it is owned by you, then you should be okay.

Listing 10.5 presents a function that tries to securely open a file in `/tmp`. The authors of this book have not had it audited professionally, nor are we professional security experts, so there's a good chance that it has a weakness, too. We do not recommend that you use this code without getting an audit, but it should at least convince you that writing secure code is tricky. To help dissuade you, we've deliberately made the interface difficult to use in real programs. Error checking is an important part of writing secure software, so we've included error-checking logic in this example.

Listing 10.5 **(*temp-file.c*) Create a Temporary File**

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

/* Returns the file descriptor for a newly created temporary file.
   The temporary file will be readable and writable by the effective
   user ID of the current process but will not be readable or
   writable by anybody else.

   Returns -1 if the temporary file could not be created.  */

int secure_temp_file ()
{
  /* This file descriptor points to /dev/random and allows us to get
     a good source of random bits.  */
  static int random_fd = -1;
  /* A random integer.  */
  unsigned int random;
  /* A buffer, used to convert from a numeric to a string
     representation of random.  This buffer has fixed size, meaning
     that we potentially have a buffer overrun bug if the integers on
     this machine have a *lot* of bits.  */
  char filename[128];
  /* The file descriptor for the new temporary file.  */
  int fd;
  /* Information about the newly created file.  */
  struct stat stat_buf;

  /* If we haven't already opened /dev/random, do so now.  (This is
     not threadsafe.)  */
  if (random_fd == -1) {
      /* Open /dev/random.  Note that we're assuming that /dev/random
         really is a source of random bits, not a file full of zeros
         placed there by an attacker.  */
      random_fd = open ("/dev/random", O_RDONLY);
      /* If we couldn't open /dev/random, give up.  */
      if (random_fd == -1)
        return -1;
    }

    /* Read an integer from /dev/random.  */
    if (read (random_fd, &random, sizeof (random)) !=
        sizeof (random))
      return -1;
    /* Create a filename out of the random number.  */
    sprintf (filename, "/tmp/%u", random);
    /* Try to open the file.  */
    fd = open (filename,
               /* Use O_EXECL, even though it doesn't work under NFS.  */
               O_RDWR | O_CREAT | O_EXCL,
               /* Make sure nobody else can read or write the file.  */
               S_IRUSR | S_IWUSR);
    if (fd == -1)
      return -1;

    /* Call lstat on the file, to make sure that it is not a symbolic
       link.  */
    if (lstat (filename, &stat_buf) == -1)
      return -1;
    /* If the file is not a regular file, someone has tried to trick
       us.  */
    if (!S_ISREG (stat_buf.st_mode))
      return -1;
    /* If we don't own the file, someone else might remove it, read it,
       or change it while we're looking at it.  */
    if (stat_buf.st_uid != geteuid () || stat_buf.st_gid != getegid ())
      return -1;
    /* If there are any more permission bits set on the file,
       something's fishy.  */
    if ((stat_buf.st_mode & ~(S_IRUSR | S_IWUSR)) != 0)
      return -1;

    return fd;
  }
```

This function calls `open` to create the file and then calls `lstat` a few lines later to make sure that the file is not a symbolic link. If you're thinking carefully, you'll realize that there seems to be a race condition at this point. In particular, an attacker could remove the file and replace it with a symbolic link between the time we call `open` and the time we call `lstat`. That won't harm us directly because we already have an open file descriptor to the newly created file, but it will cause us to indicate an error to our caller. This attack doesn't create any direct harm, but it does make it impossible for our program to get its work done. Such an attack is called a *denial-of-service* (*DoS*) attack.

Fortunately, the sticky bit comes to the rescue. Because the sticky bit is set on `/tmp`, nobody else can remove files from that directory. Of course, `root` can still remove files from `/tmp`, but if the attacker has `root` privilege, there's nothing you can do to protect your program.

If you choose to assume competent system administration, then `/tmp` will not be mounted via NFS. And if the system administrator was foolish enough to mount `/tmp` over NFS, then there's a good chance that the sticky bit isn't set, either. So, for most practical purposes, we think it's safe to use `mkstemp`. But you should be aware of these issues, and you should definitely not rely on `O_EXCL` to work correctly if the directory in use is not `/tmp`—nor you should rely on the sticky bit being set anywhere else.

10.6.3 Using *system* or *popen*

The third common security hole that every programmer should bear in mind involves using the shell to execute other programs. As a toy example, let's consider a dictionary server. This program is designed to accept connections via the Internet. Each client sends a word, and the server tells it whether that is a valid English word. Because every GNU/Linux system comes with a list of about 45,000 English words in `/usr/dict/words`, an easy way to build this server is to invoke the `grep` program, like this:

```
% grep -x word /usr/dict/words
```

Here, *`word`* is the word that the user is curious about. The exit code from `grep` will tell you whether that word appears in `/usr/dict/words`.[10]

Listing 10.6 shows how you might try to code the part of the server that invokes `grep`:

Listing 10.6 **(*grep-dictionary.c*) Search for a Word in the Dictionary**

```
#include <stdio.h>
#include <stdlib.h>

/* Returns a nonzero value if and only if the WORD appears in
   /usr/dict/words.  */

int grep_for_word (const char* word)
{
  size_t length;
  char* buffer;
  int exit_code;
```

10. If you don't know about `grep`, you should look at the manual pages. It's an incredibly useful program.

```
/* Build up the string 'grep -x WORD /usr/dict/words'.  Allocate the
     string dynamically to avoid buffer overruns.  */
  length =
    strlen ("grep -x ") + strlen (word) + strlen (" /usr/dict/words") + 1;
  buffer = (char*) malloc (length);
  sprintf (buffer, "grep -x %s /usr/dict/words", word);

  /* Run the command.  */
  exit_code = system (buffer);
  /* Free the buffer.  */
  free (buffer);
  /* If grep returned 0, then the word was present in the
     dictionary.  */
  return exit_code == 0;
}
```

Note that by calculating the number of characters we need and then allocating the buffer dynamically, we're sure to be safe from buffer overruns.

Unfortunately, the use of the `system` function (described in Chapter 3, "Processes," Section 3.2.1, "Using `system`") is unsafe. This function invokes the standard system shell to run the command and then returns the exit value. But what happens if a malicious hacker sends a "word" that is actually the following line or a similar string?

```
foo /dev/null; rm -rf /
```

In that case, the server will execute this command:

```
grep -x foo /dev/null; rm -rf / /usr/dict/words
```

Now the problem is obvious. The user has turned one command, ostensibly the invocation of `grep`, into two commands because the shell treats a semicolon as a command separator. The first command is still a harmless invocation of `grep`, but the second removes all files on the entire system! Even if the server is not running as `root`, all the files that can be removed by the user running the server will be removed. The same problem can arise with `popen` (described in Section 5.4.4, "`popen` and `pclose`"), which creates a pipe between the parent and child process but still uses the shell to run the command.

There are two ways to avoid these problems. One is to use the `exec` family of functions instead of `system` or `popen`. That solution avoids the problem because characters that the shell treats specially (such as the semicolon in the previous command) are not treated specially when they appear in the argument list to an `exec` call. Of course, you give up the convenience of `system` and `popen`.

The other alternative is to validate the string to make sure that it is benign. In the dictionary server example, you would make sure that the word provided contains only alphabetic characters, using the `isalpha` function. If it doesn't contain any other characters, there's no way to trick the shell into executing a second command. Don't implement the check by looking for dangerous and unexpected characters; it's always safer to explicitly check for the characters that you know are safe rather than try to anticipate all the characters that might cause trouble.