

CHAPTER 1

**LINUX SECURITY
OVERVIEW**

This chapter introduces you to some of the security features of the Linux operating system. We will also cover aspects of Linux that differ from other UNIX-like operating systems. This chapter covers the basics of Linux security; if you are a seasoned Linux administrator, you will likely find some of this chapter familiar territory.

WHY THEY WANT TO ROOT YOUR BOX

The highest-level user on a Linux machine is named `root` (you'll learn more about users later). The `root` user has complete and total control over all aspects of the machine—you can't hide anything from `root`, and `root` can do whatever `root` wants to do. Therefore, for a cracker (also known as a "malicious hacker") to "root your box" means the cracker becomes the `root` user, thereby gaining complete control over your machine.

NOTE

Some kernel patches, such as LIDS and SELinux (discussed in Chapter 2), can contain the all-powerful nature of `root` and make your machine more secure, even in the event of a `root` compromise.

A common misconception of many Linux users is that their Linux machine is not interesting enough to be cracked. They think, "I don't have anything important on my machine; who would want to crack me?" This type of user's machine is exactly what crackers want to crack. Why? Because cracking a machine owned by an unsuspecting user is easy. And usually the cracker's ultimate goal is not the machine he or she has cracked, but other, more important machines.

They Want Your Bandwidth Crackers may want to crack your machine to use it as a stepping stone. In other words, they will crack your machine and do evil deeds from your machine so it appears as though *you* are the evil-doer, thereby hiding their trail.

They may want to use your machine to access another machine, and from that machine access another machine, and from that machine access another machine, and so on, on their way to obtaining `root` on a `.gov` machine.

They may use your machine as part of a group of computers they have compromised with the purpose of using the combined machines to perform distributed denial-of-service (DDoS) attacks, such as the attack against the root DNS servers in October 2002.

Perhaps they want access to your machine so that they can then have access to your employer's machine. Or your friend's machine. Or your kid's machine, especially if your child has a more sophisticated computer than you do (which, with today's kids, is quite common).

They Want Your CPU Crackers may want to crack your machine to use your CPU to execute their programs. Why waste their own resources cracking the numerous password files they procure (see Chapter 9) when they can have your machine do it for them?

They Want Your Disk Space Crackers may want to store data on your machine so they don't use up their own disk space. Perhaps they have pirated software (such as *warez*—software that has been stripped of its copy-protection and made available for downloading from the

Internet) they'd like to make available, or maybe they just want to store MP3s or their MPEGs of questionable moral content.

They Want Your Data Crackers may want your business's trade secrets for personal use or to sell. Or they may want your bank records or credit card numbers.

They Want to Destroy They may just want to wreak havoc. The sad fact is that some people in the world like to sabotage other people's computer systems for no other reason than that they can. Maybe they think it is cool, or maybe they have destructive personalities. Perhaps it brings them some sort of bizarre pleasure, or they want to impress their cracker friends. They might be bored with nothing better to do with their lives. Who knows why they want to crack your machine? Fact is, they do want to crack your machine. Our machines.

Therefore, it is up to us to educate ourselves on their tactics, strategies, and methods and protect ourselves from them.

THE OPEN SOURCE MOVEMENT

Open Source is all the buzz these days, but some readers (or should we say, some readers' bosses?) don't understand what the Open Source Movement is all about. We would like to spend some time discussing the positive security aspects of of Open Source software even though we are probably preaching to the choir. However, if you need to make a case for the merits of Open Source security to the suits (also known as "upper management"), feel free to refer them to this section of the book.



Cracking into Proprietary Operating Systems

<i>Popularity:</i>	9
<i>Simplicity:</i>	9
<i>Impact:</i>	9
<i>Risk Rating:</i>	9

Every program on every computer has, at one time or another, had a bug. If the program is the operating system itself, having a bug is a serious problem because the bug can be exploited by a person with questionable morals, allowing him to gain access to everything on the computer. This is a Bad Thing—gaining `root` access on your box allows the person to harm your data and your machine.

Operating system security exploits are discovered all the time, and they are exploited by crackers who have sophisticated channels of communication, who share their exploits with others of their ilk, and who pass along their method of gaining access to countless others with questionable moral intent. The result is that, when an operating system exploit is found, all machines using that operating system are also vulnerable to a similar attack.

If that operating system is proprietary—that is, owned and controlled by one company, person, or entity—this usually means that the software is *closed*, unavailable, unreadable, and unchangeable by others. If the software is closed, the exploit is vulnerable until that company, person, or entity decides to supply the resources necessary to fix the bug and make that fix available to those that use the proprietary operating system. The speed that the fix is available depends upon a lot of variables. The people who make decisions can decide the exploit is “not bad enough,” “too complicated to fix,” or “not important enough” to provide a solution. Or an executive decision may be made not to fix the problem at all since the work necessary to fix it is greater than the apparent risk.

Make no mistake about it, though: *all* programs have had bugs. Do have bugs. Will have bugs. That is the nature of software, that bugs do exist and will exist. Always have, always will. When we rely on an individual, company, or entity to fix the bugs, we are at their mercy, because they decide when, where, and how much they fix.

— Use an Open Source Operating System

Linux is part of what is known as the *Open Source movement*. The Linux operating system is free, but more important, Linux is open. That means that the source code for the operating system is available, and anyone can view the code and examine it, modify it, and suggest and make changes to it.

Thousands of programmers do just that. These programmers are constantly improving Linux, and when a bug or security exploit is found, these Open Source programmers fix it *immediately*. Solutions to serious security exploits—and Linux has serious security exploits because all operating systems have them—are usually available within hours of the exploit being discovered. We don’t have to wait for a monolithic, large, bottom-line-focused company to fix the problem for us.

Many programs are part of the Open Source movement, and some of these programs are the most popular programs used around the world:

- ▼ **Apache** A web server that is used on approximately two-thirds of all web sites on the Internet.
- **Perl** A popular programming language used to solve all sorts of problems.
- **Sendmail, PostFix, and Qmail** These mail servers handle the majority of the Internet’s email traffic.
- ▲ **Mozilla** A previously closed source web browser (Mozilla evolved from the Netscape browser) that became Open Source.

NOTE

Each of these programs is available on almost all distributions of Linux.

Open Source and Security

Proponents of Open Source claim that the nature of open source software makes it more secure. Critics of Open Source claim that open software is less secure.

Plusses of the Open Source Model

Open Source is more secure because anyone can view it, and anyone can improve it. In the case of the Linux kernel and applications, thousands of people are doing just that by becoming actively involved in improving the software.

NOTE

Many Open Source projects describe how one can contribute to the ongoing development of the product. An example of one such project is the Mozilla browser: read <http://www.mozilla.org/hacking/> to learn how you can contribute a software patch and <http://www.mozilla.org/quality/> to learn how you can help test the latest version.

In 1997, Eric Raymond wrote a watershed paper titled “The Cathedral and the Bazaar” (<http://www.tuxedo.org/~esr/writings/cathedral-bazaar/>). In this paper, he makes many good points about the benefits of Open Source software, but one of his most important points is this: If the software is Open Source, potentially thousands of programmers can view the software, and by viewing it, they can find, point out, and fix any errors.

Another excellent point that he makes is that Open Source software is thoroughly tested. When a beta (pre-release) version of the Linux kernel is released, thousands of programmers download it and begin using it in real-world applications. This pre-release, real-world use by thousands of programmers provides a test scenario that is almost impossible to match in closed source, proprietary software. Prior to a release of a new version of the Linux kernel or a Linux application, it has been viewed, tested, and improved upon by many diverse programmers who have no other goal than to produce a high-quality product. They don’t have to cut corners or ignore problems to satisfy “the suits.” They do this work with one purpose in mind: to create a reliable and secure product because it is what they want to use. If it is not secure, they won’t use it or they will fix it.

Drawbacks of the Open Source Model

As you can imagine, there are critics of Open Source software. And not surprisingly, many of them have the perspective of closed, proprietary software. One argument that the opponents of Open Source make is that the Open Source model requires a large group of programmers who are benevolent and have a real desire to create a reliable and secure product, and if they aren’t benevolent, the model fails. To a large degree, this is true. However, history has shown that the individuals who are committed to Open Source software, and to Linux in particular, are indeed benevolent. Their goal is high-quality software and the recognition that comes with being a part of a movement that is changing the world.

Examples of benevolent programmers can be seen in the world of Linux, headed by the likable, benevolent leader, Linus Torvalds. Through his direction, other programmers who believe in the concept of Open Source have created a world-class operating system. Another example is Larry Wall, the creator of Perl, a popular Open Source programming language. Through Larry’s guidance, many talented programmers have banded together to create a usable, powerful, robust programming language for no other reason than that it is the right thing to do.

Another criticism made about Open Source software is that it is anti-capitalistic (see <http://www.microsoft.com/presspass/exec/craig/05-03sharedsource.asp>). In other words, it is not good for the economy for companies to distribute code that is free and open. These

are strong words from a company that makes a profit on closed source, proprietary software and one that considers Open Source software a real threat—so real, in fact, that a Microsoft employee wrote a document that is now known as the “Halloween Document” (because it was leaked and published publicly on Halloween 1998; see <http://www.opensource.org/halloween/>). In that document, the Microsoft employee admitted that Open Source software is a threat to proprietary software and laid out the Microsoft strategy to fight the emergence of Open Source software.

LINUX USERS

Since Linux is a *multiuser* operating system, a Linux machine can have more than one user logged in at any time, and each of those users can log in more than once at any one time. Knowledge of the types of users and how to manage them is essential to system security.



Doing Everything as root

<i>Popularity:</i>	5
<i>Simplicity:</i>	9
<i>Impact:</i>	9
<i>Risk Rating:</i>	8

Always logging in as the `root` user is dangerous for several reasons. First, accidents happen, and it is easy to create disastrous typos when executing shell commands. Imagine accidentally typing the following, including that big-mistake-don't-do-this-on-purpose space before the asterisk:

```
root# rm -rf temp_files_ *
```

In addition, if a piece of vulnerable software is run as `root`, that program's flaw can affect the entire system. For instance, an attacker can exploit a bug in your mail reader that, if run as `root`, can wipe the hard drive.

Some operating systems, such as Windows 95/98/Me, run everything as the same user. Therefore, any person using the machine can delete important files or install monitoring software.

NOTE

There's a new product called LindowsOS that promises the ease of Windows with the stability and power of a Linux base. It runs many Windows applications (through WINE, a Win32 API emulator) and emulates the Windows security model: everything runs as `root`. Until the LindowsOS folks make it possible to run things as non-`root`, we suggest you avoid this product—as you would the operating system it is attempting to emulate.

— Create Non-root Users to Perform Non-root Tasks.

Even if you are the only user on your Linux machine, create a new user for yourself, and then do all non-`root` commands and operations as that user. To add a new user, execute the `adduser` command as `root`, and then give that user a good password (more on what makes a good password in Chapter 9):

```
root# adduser jdoe
root# passwd jdoe
Changing password for user jdoe
New password:
Retype new password:
```

Now that a non-`root` user exists, we can have that user perform tasks such as using a mail reader. If the mail reader is exploited by an attacker as `jdoe`, the effect on the machine is only that over which `jdoe` has control, which is much less than the damage `root` could do.



Shared Accounts

<i>Popularity:</i>	3
<i>Simplicity:</i>	9
<i>Impact:</i>	7
<i>Risk Rating:</i>	7

Imagine that you have a machine used by several people, and that everyone logs in using the same username. One day, that account is responsible for the deletion and corruption of some very important files and directories, or perhaps the account is being used to attack other sites, or the machine itself. You can only tell the account that was used, not the actual person who sat at the keyboard.

— Create a User for Each Person Accessing the Machine

Do not allow more than one person to share the same username. Give each user his or her own username; this allows for some simple auditing to be instantly available, such as what time a particular user logs in and out. You'll also see later how you can enforce permissions on files and users—to give them shared access where appropriate—and lock things down when necessary.

/etc/passwd

Information about all local Linux accounts is stored in the file `/etc/passwd`, including any user added with the `adduser` command. Here is an example of this file:

```
jdoe$ cat /etc/passwd
root:a1eGVpwjgvHGg:0:0:root:/root:/bin/bash
```

```
bin:*:1:1:bin:/bin:
daemon:*:2:2:daemon:/sbin:
adm:*:3:4:adm:/var/adm:
lp:*:4:7:lp:/var/spool/lpd:
sync:*:5:0:sync:/sbin:/bin/sync
mail:*:8:12:mail:/var/spool/mail:
news:*:9:13:news:/var/spool/news:
uucp:*:10:14:uucp:/var/spool/uucp:
gopher:*:13:30:gopher:/usr/lib/gopher-data:
ftp:*:14:50:FTP User:/home/ftp:
nobody:*:99:99:Nobody:/:
xfs:*:100:101:X Font Server:/etc/X11/fs:/bin/false
jdoe:2bT1cMw8zeSdw:500:100:John Doe:/home/jdoe:/bin/bash
bob:9d9WE322B/o.C:501:100:./home/bob:/bin/bash
```

Each line of `/etc/passwd` is a single record containing information about the user. For example, let's look at the entry for `jdoe`:

```
jdoe:2bT1cMw8zeSdw:500:100:John Doe:/home/jdoe:/bin/bash
```

The record has a number of fields that are colon separated:

<code>jdoe</code>	The username, unique for the Linux machine.
<code>2bT1cMw8zeSdw</code>	The encrypted password. If this field is an 'x' then it indicates the encrypted password is in the <code>/etc/shadow</code> file instead. See Chapter 9 for details.
<code>500</code>	The user ID number, unique for the Linux machine; this number is used by the operating system to keep track of files that <code>jdoe</code> owns or can access.
<code>100</code>	The group ID number (you'll learn more about groups in the section called "Linux Users," later in this chapter).
<code>John Doe</code>	The GECOS field, which can be any string but is usually the user's name.
<code>/home/jdoe</code>	The home directory, which is the directory the user is given to store personal files; the user will be put into this directory upon logging in.
<code>/bin/bash</code>	The shell; when the user logs in, this is the program that will accept and execute Linux commands.

Several shells are available for Linux, including the following:

/bin/sh	The Bourne shell, named for Steven Bourne, its creator.
/bin/ksh	The Korn shell, named for creator David Korn. It adds a number of features that were lacking in the Bourne shell. Ksh has been adopted as <i>the</i> POSIX (1003.2) shell.
/bin/bash	The Bourne Again shell, created by the Free Software Foundation, is an improved version of the Bourne shell. It incorporates the best elements from both <code>ksh</code> and <code>csh</code> . It also can be POSIX compliant and is the default shell for Linux systems.
/bin/csh	The C shell written by Bill Joy, founder of Sun Microsystems. It uses syntax closer to the C programming language. While it is a fair user shell, it is a bad shell scripting language.
/bin/tcsh	A variant of the C shell that supports command-line editing.



Executing (Unwisely) Aliased Commands

<i>Popularity:</i>	3
<i>Simplicity:</i>	8
<i>Impact:</i>	7
<i>Risk Rating:</i>	7

Some Linux distributions like to set up your shell environment with questionable aliases; for example, Red Hat sets `alias rm='rm -i'` for the `root` user. This alias executes the `rm` command in interactive mode. In interactive mode, if `rm` is about to remove a file that exists, it will ask if you really want to remove it:

```
root# rm -i foo
rm: remove 'foo'?
```

To remove the file, you must enter `y`.

The problem with this alias arises when `root` uses `rm` to remove an important file, expecting the command to issue a prompt if the file exists. But `root` may execute the command on a machine that does not alias `rm` to `rm -i`. The default behavior of `rm` is that the file is quietly removed with no interactive prompt, yet this user expects to be prompted if the file exists. The user is unhappy when the important file is deleted.

The first time you expect this behavior on a machine that does not alias `rm` by default, you will understand our objections.



Create Different Command Names for Aliases

We highly discourage the practice of making deadly commands safe. If you need a safe alias, try `alias del='rm -i'` instead, so that you never expect `rm` to behave interactively.

User Types

Though `/etc/passwd` seems to treat all users equally, you can actually think of Linux users being in one of three different groups:

- ▼ `root`
- Normal users
- ▲ System users

Let's take a quick look at the purposes of each.

root The superuser, normally named `root`, has complete control over the entire system. The `root` user can access all files on the system, and the `root` user is generally the only user who can execute certain programs. (For instance, `root` is the only user who can execute `httpd`—the Apache web server—since `httpd` binds to port 80, a port restricted to `root`.) A cracker wants complete control of the system; therefore, he wants to become `root`. Here is the `root` entry from our example `/etc/passwd` file:

```
root:aleGVpwjgvHGg:0:0:root:/root:/bin/bash
```

Notice that `root` has a user ID of 0. Any account with a user ID of 0 is a `root` user, even if the username is not `root`. Common other `root`-equivalent account names include `toor` and `super`.

Normal Users Normal users can log in and use the system typically to perform basic computing tasks such as surfing the web, reading email, creating documents, and so on. An example of a normal user is `jdoe`, who was added earlier in the chapter with the `adduser` command.

Normal users usually have a home directory (some users don't have a home directory and can't log in, such as those who have `/bin/ftponly` as a shell) and can create and manipulate files in their home directory and in other directories. These are the standard user accounts that human beings use to get their work done (assuming they are using Linux to get their work done). Normal users typically have restricted access to files and directories on the machine, and as a result, they cannot perform many system-level functions. (You'll learn various restrictions throughout the rest of this chapter.)

System Users System users don't log in. Their accounts are used for specific system purposes but they are not owned by a specific person. These users do not log in, and usually do not have a normal home directory assigned. (The home directory field in `/etc/passwd` does need a value—sometimes `'/'` will be used, or a nonexistent directory. Since these users cannot log in, the home directory is usually irrelevant.) Additionally, their shell in `/etc/passwd` should not be a valid login shell. A typical example would be `/bin/false`.

Examples of system users include `ftp`, `apache`, and `lp`. The `ftp` user is used for anonymous FTP access, the `apache` user typically handles HTTP requests (some Linux distributions have HTTP requests handled by the user `nobody` or `www-data`), and `lp` handles

printing functions (lp stands for line printer). The actual system users on your machine depend on your Linux distribution and the software you have installed.

Linux Groups

Linux implements the concept of *groups*. A group is a collection of one or more users. It is often convenient to collect a number of users together to define properties for the group, such as controls on what they can or cannot access.



Permissive Write Permissions

<i>Popularity:</i>	7
<i>Simplicity:</i>	8
<i>Impact:</i>	5
<i>Risk Rating:</i>	7

It is easy to have inappropriate permissions on files and directories. New Linux users will frequently grant full read and write access to files and directories to allow other users to work with them without any barriers. However, this means that both legitimate users and crackers have the same access. A cracker may find sensitive data in these files that may be helpful in cracking the `root` account. Or perhaps these files are used by security-related programs and the cracker can affect how they run by editing the files. Worse yet, any files that are in writable directories can be deleted even if the user is not the owner of the file. (Though you can prevent this by setting the sticky bit on the directory, as we discuss in the section, “Set the Sticky Bit for the Directory” later in this chapter.)



Use Group File Permissions

The groups on the Linux machine are defined in the file `/etc/group`. Here is a snippet of this file:

```
root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
sys:x:3:root,bin,adm
adm:x:4:root,adm,daemon
mail:x:12:mail
ftp:x:50:
nobody:x:99:
users:x:100:jdoe,bob
```

Each line of `/etc/group` contains a single record of information about the group. For example, let’s look at the entry for the `users` group:

```
users:x:100:jdoe,bob
```

The record includes a number of fields that are colon separated:

users	The unique name of the group.
x	The encrypted group password; if this field is empty, no password is needed, and if it is x, the encrypted password is read from the file <code>/etc/gshadow</code> . In general you do not want to have group passwords available at all.
100	The unique group ID number.
jdoe,bob	A comma-separated list of the group member usernames.

In this case, two people are in the users group, jdoe and bob. You can easily see what groups you are in by running the `id` command:

```
jdoe$ id -a
uid=100(jdoe) gid=100(users) groups=100(users)
```

NOTE

Every user has a default group that is specified in the fourth field of `/etc/passwd`. This means that you should check both `/etc/group` and `/etc/passwd` to see which users are in a given group.

To create a new group to protect our important files, as `root` execute the `groupadd` command to create the group. Then add the users to this group using the `usermod` command or editing `/etc/group` directly. Then use `chgrp` to change the group ownership of the file. Lastly, you'll run `chmod` (described in depth later in this chapter) to grant group read and write permissions to this file:

```
root# groupadd webadmin
root# usermod -G webadmin jdoe
root# chgrp webadmin index.html
root# chmod g+rw index.html
```

How to Place Controls on Users

Linux system security lets you place controls on users. Several different types of controls can be used, including file permissions, file attributes, filesystem quotas, and system resource limits.



Reading Personal Files

Popularity:	9
Simplicity:	9
Impact:	4
Risk Rating:	7

Without proper precautions, any user on the Linux machine can read another user's personal files—a resume sent to a prospective employer, a love letter, an email rant

against a boss (that was never sent, of course), or an idea for a new dot-com business (ripe for stealing).

Place Appropriate Permissions on Files and Directories

You should always set restrictive permissions bits on sensitive files, thus making them unreadable by others. For directories containing sensitive information, you can prevent others from even entering the directory at all.

The best idea is to set the tightest permissions possible on all your files and directories. Data in files is more secure if you lock down the files to the greatest extent possible and then grant less restrictive permissions on a case-by-case, as-needed basis.

File Permissions

Linux file permissions are mechanisms that allow a user to restrict access to a file or directory on the filesystem. For files, a user can specify who can read the file, who can write to the file, and who can execute the file (used for executable programs). For directories, a user can specify who can read the directory (list its contents), who can write to the directory (add or remove files from the directory), and who can execute programs located in the directory.

Let's look at a simple example of changing file permissions:

```
jdoh$ ls -l a.txt
-rw-rw-r-- 1 jdoh users 24043 Nov 5 07:40 a.txt
```

Here we execute `ls -l`. The `ls` command lists the contents of the directory—in this case, only the contents of the file `a.txt`. The `-l` option lists the file information in long mode, which displays quite a bit of information about the file. The output lists the following information:

```

-rw-rw-r-- 1 jdoh users 24043 Nov 5 07:40 a.txt
  ↑           ↑   ↑           ↑           ↑           ↑           ↑
Permissions Number of User   Group   Number of bytes Date last modified Name
             hard links

```

Notice that this file belongs to one user (`jdoh`) and to one group (`users`). The user and group are important information to know when we discuss file permissions.

The file permissions are as follows:

```
-rw-rw-r--
```

This information is divided into four parts:

```

-  rw-  rw-  r--
↑      ↑      ↑      ↑
File type User permissions Group permissions Other permissions

```

The first character of the output is the file type. The most common file types are shown here:

-	A normal file
d	A directory
l	A symbolic link
s	A socket
p	A FIFO pipe

Following the file type are three groups of three characters representing the permissions for the user, group, and world. The three characters indicate whether or not permission is granted to read the file (*r*), write to the file (*w*), or execute the file (*x*). If permission is granted, the letter is present. If permission is denied, the letter's position is held by a dash (-). Here is an example:

```
rwxr-x--x
```

Here, the first three characters are the permissions for the owner (the user). The permissions *rw**x* indicate that the user can read the file, write to the file, and execute the file. The next three characters are the permissions for the group associated with the file. The permissions *r-x* indicate that members of the group can read and execute the file but cannot write to the file. The last three characters are the permissions for everyone else, referred to as 'other'. The permissions *--x* indicate that others cannot read or write to the file but can execute the file.

Note that the three permissions are either granted or denied—they're either on or off. Since the permissions can be considered either on or off, the permissions can be thought of as a collection of 0's or 1's. For instance, *rw**x* has read permission on, write permission on, and execute permission on. Therefore, we can write these permissions as 111, and we can write them in octal format as value 7. Similarly, *r-x* has read permission on, write permission off, and execute permission on. Therefore, we can write these permissions as 101, and in octal format as the value 5.

If we put this idea into practice for user/group/other permission, these permissions

```
rwxr-x--x
```

in binary format are

```
111101001
```

If we treat this as a series of three groups of octal numbers, the value is 751.

Changing File Permissions

The `chmod` command changes file permissions. Here's its format:

```
chmod mode file [file ...]
```

To see how to use `chmod`, let's look at a file on our system:

```
jdoue$ ls -l a.txt
-rw-rw-r-- 1 jdoue users 10 Nov 15 12:19 a.txt
```

To change the permissions to an explicit mode, use the octal method:

```
jdoue$ chmod 751 a.txt
jdoue$ ls -l a.txt
-rwxr-x--x 1 jdoue users 10 Nov 15 12:19 a.txt
```

Notice how the permissions 751 translate to `rwxr-x--x`. And look at this:

```
jdoue$ chmod 640 a.txt
jdoue$ ls -l a.txt
-rw-r----- 1 jdoue users 10 Nov 15 12:19 a.txt
```

Here, 640 translates to `rw-r-----`.

You can also use the `chmod` command in symbolic mode, as follows:

```
jdoue$ ls -l a.txt
-rw-r----- 1 jdoue users 10 Nov 15 12:24 a.txt
jdoue$ chmod +x a.txt
jdoue$ ls -l a.txt
-rwxr-x--x 1 jdoue users 10 Nov 15 12:24 a.txt
```

Here, `chmod` is used with `+x`, which means “add executable permission.” When the `+` character is used, it means to add the permission, whereas the `-` character means to subtract or remove the permission. Here, `+x` means to add executable permissions for the user, group, and other.

The `chmod` command can also be used to change permissions for a specific group:

```
jdoue$ chmod g-r a.txt
jdoue$ ls -l a.txt
-rwx--x--x 1 jdoue users 10 Nov 15 12:24 a.txt
```

This example shows `chmod` being executed with `g-r`, which means “remove group read permissions.”

So a user can change permissions on all his love letters and rants about his boss with this:

```
jdoue$ chmod 600 love_letter_*
jdoue$ chmod 600 why_I_hate_my_boss_*
```

But more importantly, a user should *never* keep passwords in unencrypted files:

```
jdoue$ rm root_passwords_in_clear_text
```



Deleting Another User's Files in a Writable Directory

<i>Popularity:</i>	9
<i>Simplicity:</i>	9
<i>Impact:</i>	9
<i>Risk Rating:</i>	9

Sometimes it may be convenient to create a directory in which multiple users can create files—for example, in a web document `root` or a common work area. Anyone who has write permission to this directory, be it via group or other permissions, can not only create files in this directory, but can also delete any files, including those that are not owned by this user. (This may seem counterintuitive, but it's been the UNIX standard for a few decades now.)

For example, say we have a world-writable directory called `temp`:

```
jdoue$ ls -ld temp
drwxrwxrwx  2 jdoue  users      1024 Nov 29 15:03 temp
```

We see that the `temp` directory is owned by `jdoue`, yet it's writable by everyone. Now let's look at how a different user, `bob`, removes a file that `bob` cannot read and does not own:

```
bob$ ls -l
total 0
-rw-----  1 jdoue  users      0 Nov 29 15:00 a
-rw-----  1 root   root       0 Nov 29 14:59 b
-rw-----  1 bob    users     0 Nov 29 14:59 c
-rw-----  1 jdoue  users     0 Nov 29 14:59 d
bob$ cat b
cat: b: Permission denied
bob$ rm -f b
bob$ ls -l
total 0
-rw-----  1 jdoue  users      0 Nov 29 15:00 a
-rw-----  1 bob    users     0 Nov 29 14:59 c
-rw-----  1 jdoue  users     0 Nov 29 14:59 d
```

The `ls -ld temp` command shows that the user `bob` has read/write/execute permissions for the `temp` directory. Then we see that four files are included in the `temp` directory, three of which are not owned by `bob` and for which `bob` does not have read/write permissions. We see that user `bob` could successfully remove a file that he could not read. The user `bob` can do this because he can write to the directory—when a file is removed in Linux, it is the directory that is changed; therefore, it is the directory that must be writable.

— Set the Sticky Bit for the Directory

You can set permissions on a directory so that a user can remove only files within it that are owned by that user. In other words, by setting these permissions, a user cannot remove files that are owned by another user. To set this permission, use `chmod` with the `+t` option. This sets the *sticky bit*:

```

jdoe$ chmod +t temp
jdoe$ ls -ld temp
drwxrwxrwt  2 jdoe  users          1024 Nov 29 15:21 temp

```

Notice that the sticky bit is indicated by the `t` in the last position. Now that the sticky bit is set, other users cannot remove files or directories that they do not own:

```

bob$ ls -l
total 0
-rw-----  1 jdoe  users          0 Nov 29 15:00 a
-rw-----  1 bob   users          0 Nov 29 15:15 c
-rw-----  1 jdoe  users          0 Nov 29 14:59 d
bob1$ rm -f a
rm: cannot unlink 'a': Operation not permitted
bob$ rm c
bob$ ls -l
total 0
-rw-----  1 jdoe  users          0 Nov 29 15:00 a
-rw-----  1 jdoe  users          0 Nov 29 14:59 d

```

Now that the sticky bit is set, the user `bob` cannot remove a file owned by `jdoe`, yet `bob` can still remove the files he owns.

NOTE

In the past, the sticky bit had a purpose on executable files. The sticky bit has been used on executables. Frequently-used programs, such as editors and tools like `grep`, would have this bit set to tell the kernel that the program text (the executable code) should stay in memory even after it completes, in hopes that it would be available sooner the next time it was invoked. Now we have much better memory management and automatic caching, so this sort of performance boost happens automatically, and the sticky bit on files doesn't mean anything on nondirectories on standard Linux kernels.

A perfect example of a directory that has the sticky bit set is `/tmp`, a depository that all users can access for temporary files and directories. All users can create files and directories, but users can remove only files and directories that they own:

```

jdoe$ ls -ld /tmp
drwxrwxrwt 21 root  root          3072 Nov 29 13:41 /tmp

```

suid and sgid

There are two other permission bits you can apply to a file in addition to the read (r), write (w), execute (x) and sticky (t) bits. The first is the set-user-id bit—suid for short—which makes the program run as the file owner, regardless who actually executes the program. You set this bit as follows:

```
root# ls -l suidfile
rwxr-xr-x 21 jdoe users          28389 Nov 29 13:50 suidfile
root# chmod u+s suidfile
root# ls -l suidfile
rwsr-xr-x 21 jdoe users          28389 Nov 29 13:50 suidfile
```

The 's' in the user 'x' position is the suid bit. Similarly, we have the set-group-id bit—sgid for short—which makes the program run with the file's group membership regardless of who actually executes the program. You set this bit in the same fashion:

```
root# ls -l sgidfile
rwxr-xr-x 21 web  devel          22142 Nov 29 13:52 sgidfile
root# chmod g+s sgidfile
root# ls -l sgidfile
rwxr-sr-x 21 web  devel          22142 Nov 29 13:52 sgidfile
```

Often you will refer to files with one of these bits set by indicating which user or group is applicable. So `suidfile` could be referred to as a 'suid jdoe' executable, and `sgidfile` as a 'sgid devel' executable. We will discuss suid and sgid programs and the security implications they bring in Chapter 2.

NOTE

A program with a suid and/or sgid bit set is often called a `sXid` or `setXid` executable. This generic term allows you to talk about any executable that grants you additional privileges without specifically detailing which privileges those are.



Insecure Default Permissions

<i>Popularity:</i>	9
<i>Simplicity:</i>	9
<i>Impact:</i>	8
<i>Risk Rating:</i>	8

Suppose that a system administrator wants to `grep` through `/var/log/messages` looking for failed logins and save the result in the file `failed_logins.txt`. She might do this with this command (output wrapped for readability):

```
root# grep 'FAILED LOGIN' /var/log/messages > failed_logins.txt
root# cat failed_logins.txt
```

```
Jul 11 12:34:29 smarmy login[12109]: FAILED LOGIN SESSION FROM
foo.example.com FOR jdoe, Authentication failure
```

If a user tried to log in and accidentally typed his password in place of his username, `failed_logins.txt` would contain this:

```
root/# cat failed_logins.txt
Jul 11 12:34:29 smarmy login[12109]: FAILED LOGIN SESSION FROM
foo.example.com FOR Fido123, Authentication failure
```

NOTE

This user has chosen a bad password (Fido123). For a discussion of good passwords, see Chapter 9.

If the sys admin created this file, yet forgot to `chmod` the file to give it tighter permissions, the file would be readable by a cracker who would then have another password he could exploit.

— Use umask to Set Default Permissions

When a user creates a file or directory, that file or directory is given default permissions:

```
jdoe$ touch a.txt
jdoe$ mkdir directory_b
jdoe$ ls -l
total 1
-rw-rw-r--  1 jdoe  users          0 Nov 29 13:42 a.txt
drwxrwxr-x  2 jdoe  users       1024 Nov 29 13:43 directory_b
```

Notice that the default permissions for the user `jdoe` are

- ▼ 664 (`rw-rw-r--`) for files
- ▲ 775 (`rw-rwxr-x`) for directories

Default file and directory permissions are set according to the user's `umask` value. The `umask` value is used to mask off bits from the most permissive default values: 666 for files and 777 for directories. To display your `umask` value, execute the `umask` command:

```
jdoe$ umask
002
```

Here, the user `jdoe` has a `umask` value of 002. A simple way to determine the value of `jdoe`'s default permissions when `jdoe` creates files or directories is simply to subtract the value of `umask` from the system default permission values:

Files:	666	Directories:	777
	<u>002</u>		<u>002</u>
	664		775

NOTE

In actuality, a `umask` value is not subtracted from the default permissions, but it is usually easier to think of it that way. If you are comfortable with logical operators, the actual operation used is `(mode & ~umask)`. This effectively strips from the default permissions (666 or 777 usually) the bits that are set in the `umask`.

To change your effective default permission, change your `umask` value. To create the most restrictive permission, use a `umask` value of 777:

```
jdoe$ umask 777
jdoe$ touch c
jdoe$ ls -l
total 1
-rw-rw-r--  1 jdoe  users          0 Nov 29 13:42 a.txt
-----  1 jdoe  users          0 Nov 29 14:22 c
drwxrwxr-x  2 jdoe  users       1024 Nov 29 13:43 directory_b
```

Of course, this is too restrictive since `jdoe` does not have read and write permissions for the new file:

```
jdoe$ cat c
cat: c: Permission denied
```

To create files and directories with the most practical restrictive permissions, use a `umask` value of 077:

```
jdoe$ umask 077
jdoe$ touch d
jdoe$ mkdir directory_e
jdoe$ ls -l
total 2
-rw-rw-r--  1 jdoe  users          0 Nov 29 13:42 a.txt
-----  1 jdoe  users          0 Nov 29 14:22 c
-rw-----  1 jdoe  users          0 Nov 29 14:30 d
drwxrwxr-x  2 jdoe  users       1024 Nov 29 13:43 directory_b
drwx-----  2 jdoe  users       1024 Nov 29 14:30 directory_e
```

Notice how a `umask` value of 077 gave `jdoe` read/write permissions for the file `d` and read/write/execute permissions for `directory_e`, but no permissions to the group and other.

To set the `umask` value upon login, simply add the following command to your profile script (`~/.bash_profile` or similar):

```
umask 077
```

TIP

If you don't like all this octal number crunching, you can use symbolic notation for your `umask` setting if you are using bash. The `-S` flag tells `umask` to report the symbolic mask as opposed to the octal version:

```
$ /jdoe$/umask -S
u=rwx,g=rwx,o=r
$ jdoe$ umask u=rwx,g=r,o=
$ jdoe$ umask -S
u=rwx,g=r,o=
$ jdoe$ umask
037
```

As an administrator, you can also add `umask` changes in the global file `/etc/profile` to have it apply to all users.

General Rule for File Permissions The general rule for file permissions is to apply the most restrictive permission settings to files and then add permissions for specific users or groups as necessary. It is easy to add privileges, but it is difficult to take them away without getting into a tug of war.

File Attributes

In addition to modifying a file's permissions, a user can also modify a file's *attributes*. A file's attributes are changed with the `chattr` command, and they are listed with the `lsattr` command.

NOTE

These attributes can be used only on `ext2` or `ext3` filesystems (the standard Linux filesystems). Thus, you cannot use them if you use a different filesystem such as `reiserfs`. If an `ext2/ext3` filesystem is mounted remotely, such as over NFS, the attributes are still in effect; however, you cannot use the `lsattr` or `chattr` command to list or change the attributes from the client machine.



Read, Write, and Execute Permissions Are Not Enough

Popularity:	8
Simplicity:	9
Impact:	5
Risk Rating:	7

Sometimes the read, write, and execute permissions are not enough when it comes to particularly sensitive data. You may want to prevent files from being modified, even by `root`, to prevent tampering by an attacker who has compromised your system. You may want to store sensitive data in files temporarily, but know that they are securely wiped from the disk when deleted. Or you may want to be sure that some files are not backed up with `dump`, which usually stores its data on easily-readable tape media.

Use Advanced Filesystem Attributes

Attributes allow increased protection and security to be placed on a file or directory. For instance, the `i` attribute marks the file as *immutable*, which prevents the file from being modified, deleted, renamed, or linked—it's an excellent way to protect the file. The `s` attribute forces a file's contents to be wiped completely from the disk when the file is deleted. This ensures that the file's contents cannot be accessed after the file is deleted.

Following are the attributes that can be changed:

A	Don't update the file atime, which can be helpful for limiting disk I/O on a laptop or over NFS. This attribute is not supported by all kernels, specifically the older 2.0 series
a	Open the file only in append mode; this can be set only by root.
d	Marks the file as not a candidate for the dump program.
i	The file cannot be modified, deleted, or renamed; no link to it can be created; and no data can be written to the file.
s	When the file is deleted, its blocks are zeroed out and written back out to disk.
S	When the file is modified, the changes are written immediately to the disk, rather than being buffered.

As with `chmod`, an attribute is added with `+` and removed with `-`. Here is an example:

```
jdoe$ lsattr a.txt
----- a.txt
jdoe$ chattr +c a.txt
jdoe$ chattr +d a.txt
jdoe$ chattr +s a.txt
jdoe$ lsattr a.txt
s-c---d- a.txt
jdoe$ chattr -d a.txt
jdoe$ lsattr a.txt
s-c----- a.txt
```

TIP

If you want to use `chattr -A` to minimize disk writes, you'd probably be better off simply mounting the filesystem with the `noatime` option instead. This turns off last access time updating for the whole filesystem, and you don't need to worry about using `chattr` at all.

OTHER SECURITY CONTROLS

Every Linux system has a variety of security controls that do not need to be placed individually on users but that are automatically enforced by the Linux kernel itself. These re-

restrictions are present in other UNIX-like systems as well, but they may be foreign ideas to our underprivileged Windows brethren.

Signals

In Linux, users can send *signals* to processes. A signal is a message sent from one process to another. A common signal to send to a process is the `TERM`, or terminate, signal. This signal is sent to a process to force the process to terminate and is often used to kill a runaway process. This example shows a user killing a process:

```
jdoe$ kill -TERM 13958
```

This command sends the `TERM` signal to the process with process ID 13958.

Here is an example using `killall`:

```
root# killall -HUP httpd
```

This `killall` command sends a signal (in this case `HUP`, or hangup) to all processes named `httpd`. The `HUP` signal is often used to force the process to reread its configuration file and is usually used after the program's configuration has changed.

In Linux, users can send signals only to processes that they own. In other words, the user `jdoe` cannot kill a process owned by `jsmith`. The exception is the `root` user; `root` can send a signal to any process on the system. Of course, normal users will not be able to kill processes owned by `root`, such as `httpd` and `sendmail`.

Privileged Ports

The `root` user is the only user who can bind to a port with a value less than 1024. (*Binding* to a port means that a network service connects to and begins listening at a port on the machine.) There are two main reasons for this, both related to trust:

- ▼ You can trust that a connection coming from a port less than 1024 (such as 889) on the remote machine is from a program that is run by `root`. This is used in some protocols for authentication. For example, `rsh` and `ssh` can be configured to allow certain users to log in without a password from specified systems. One way to implement this is to have the `rsh` or `ssh` client be `suid root` executables, which are granted `root` access when they are run. This allows them to bind a privileged port, and inform the server of the actual user who issued the `rsh` or `ssh` command. Since the connection is coming from a privileged port, the server can trust that the client username supplied is accurate.
- ▲ If you attempt to connect to another machine at a low-numbered port (such as 22 for `ssh` or 143 for `imap`), you can trust that the program that you are contacting is the official daemon that is requesting your username and password, not some rogue server created by a clever user on that machine. This also applies to authentication services like the `ident/auth` port, which provides remote machines the `userid` associated with an existing connection.

Virtual Memory Management

Linux's virtual memory management system has built-in security. Each process has its own memory allocated immediately upon startup for the program and static variables. Any additional runtime memory allocation (using `malloc()` or similar) is processed by the kernel automatically. No process has access to the memory of other processes unless it was set up specifically ahead of time through standard interprocess communication (IPC) methods.

This results in security—one process cannot affect another's memory segment—and stability—a flaw in one process cannot harm another.

Another Linux memory management security feature is that any process that consumes too much memory is killed by the kernel, while other processes are unaffected. Since the kernel reclaims the memory from the killed process, no memory leak from the process occurs.

Other operating systems do not offer such compartmentalization. This means that all the system memory may be available to all of the processes on the machine.

System Logging

Linux has a standard logging facility that is easy to use and can be plugged into essentially any program that is written. This feature of Linux is powerful and easy to use. You can log almost any information, manipulate the format of the information, and direct the logged information to any file or process that you choose.

The logged information is usually written to a file, so it is easy to search and parse. This is good news to those of us who prefer not to view logged information with a GUI that is limited, difficult to use, and restrictive in its nature (the method of logging information to a restrictive GUI is used by several inferior operating systems). If the information is a file, the file can be edited and searched easily. Also, simple tools such as `grep` can locate specific text in the file, and other tools such as Perl can extract and transform the text quite easily.

Logging is covered extensively throughout Chapter 2, including software packages that can help you with log analysis.

`/etc/securetty`

The `/etc/securetty` file allows you to specify which TTY devices `root` is allowed to use to log in. All devices listed in this file that are not commented out are allowable. This is enforced by the `pam_securetty.so` PAM (Pluggable Authentication Module), which is usually configured by default on Linux distributions. (We discuss PAM in more detail in Chapter 9.)

NOTE

A TTY is a terminal that one can log into. Most Linux distributions provide six TTYs that are available at the keyboard. These TTYs can be accessed by using `ALT+F1`, `ALT+F2`... `ALT+F6`. (Though you must use `CTRL+ALT+F1` if you are in X Windows at the time.) These key combinations get you to the TTY devices `/dev/tty1` through `/dev/tty6`. Each terminal window you run in X Windows is attached to a pseudo-TTY, a device such as `/dev/pts/1`, though this is not functionally different than a hard-wired TTY like `/dev/tty1`.

Here is the default `/etc/securetty` for Red Hat version 7.3:

```
jdoue$ cat /etc/securetty
vc/1
vc/2
vc/3
vc/4
vc/5
vc/6
vc/7
vc/8
vc/9
vc/10
vc/11
tty1
tty2
tty3
tty4
tty5
tty6
tty7
tty8
tty9
tty10
tty11
```

This is way too many terminals, allowing too many options for a cracker to attempt to guess `root`'s password. We suggest you delete all lines except `tty1`, which means that `root` can log in to only the first terminal that Linux provides—the one that you see when a Linux box is powered up in non-X Windows. The file `/etc/securetty` should have this content only:

```
tty1
```

Now if an administrator needs to log in as `root` to perform administrative tasks, she must first log in as a normal user and then switch (using the `su` command) to `root` as in

```
jdoue$ su -
Password:
root#
```

Now, to log in as `root`, a cracker must have access to `tty1`, which means he must have physical access to the computer. Or a cracker must first crack a normal user's password (jdoue in this case) and then crack `root`'s password. (See Chapter 9 for more on password cracking.)

chrooting

An additional security control that Linux and other UNIX-like systems offer is the ability to change the `root` directory a process sees. This restricts the access the program has, which is helpful for untrusted services or those liable to be abused by an attacker.



Attacking a Network Service

<i>Popularity:</i>	8
<i>Simplicity:</i>	6
<i>Impact:</i>	9
<i>Risk Rating:</i>	8

If a cracker discovers a vulnerability in a network service (such as Sendmail or BIND), she will be able to access your machine and its entire filesystem. If the program is not running as `root`, the cracker will attempt to find local vulnerabilities on the system to elevate her privileges and gain `root`. Because more local insecurities exist than remote insecurities, chances are good that she'll find something on the system that she can leverage. Even if a local vulnerability is not found, the attacker can do other damage, such as deleting files out of spite.



chroot the Service

Creating a `chroot jail` is an effective way to run a program restricted to a portion of the directory tree. Suppose we create a jail for an imaginary program named `convict`. We will restrict this program to a portion of the directory tree under `/usr/local/convict`. Creating a jail for `convict` in this directory means that `convict` can see only this directory and files and subdirectories under this directory. If `convict` opens the file `/jury/verdict.txt`, the file accessed is actually stored in `/usr/local/convict/jury/verdict.txt`. So if the program accesses the file `/etc/localtime`, it really accesses the file in `convict's jail /usr/local/convict/etc/localtime`.

An example of a program that creates a `chroot jail` is this simple C program that calls the `chroot()` function. The argument to the `chroot()` function is a directory that will become the jail for this program:

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

#define CONVICT "bri"
#define CHROOT_DIR "/usr/local/convict"
```

```
#define bail(x) { perror(x); exit(1); }

int main() {
    char filename[4096], buf[BUFSIZ];
    int fd, count;
    struct passwd *pw = getpwnam(CONVICT);
    if ( ! pw ) bail("getpwnam");

    if ( chdir(CHROOT_DIR) == -1) bail("chdir");
    if ( chroot(CHROOT_DIR) == -1) bail("chroot");
    if ( setgid(pw->pw_gid) == -1) bail("setgid");
    if ( setuid(pw->pw_uid) == -1) bail("setuid");

    printf("Please enter a file name: ");
    scanf("%4095s", filename);
    printf("You entered %s\n", filename);
    if ( (fd=open(filename, O_RDONLY)) >0) {
        printf("Contents of %s:\n", filename);
        while ( (count=read(fd, buf, BUFSIZ)) > 0 ) {
            write(1, buf, count);
        }
    } else {
        printf("Failed to open %s\n", filename);
    }
}
```

Under this directory is a directory named `etc`, and in this directory is a file named `passwd` with these contents:

```
this is the chrooted /etc/passwd
```

These contents are somewhat different from the `/etc/passwd`, with which we are familiar. For this `chrooted` program, the file named `/etc/passwd` will actually refer to the file `/usr/local/convict/etc/passwd`, as shown here:

```
root# ./convict
Please enter a file name: /etc/passwd
You entered /etc/passwd
Contents of /etc/passwd:
this is the chrooted /etc/passwd
```

Running a Program in a chrooted Directory Manually

To run any program in a `chroot` directory, use the `chroot` command:

```
root# chroot /usr/local/convict /bin/convict
```

In this example, `chroot` will change `root` to the `/usr/local/convict` directory and then run your program, `/bin/convict`. Because the `chroot` is performed first, the program `/bin/convict` actually resides in `/usr/local/convict/bin/convict` on the real filesystem.

Setting Up a `chroot` Jail Directory

One problem with `chrooting` your software is that all the programs and libraries that are needed by your software must be copied into the `chroot` directory, which we usually call a `chroot jail` (since you can't ever get out). If we are in a jail and we execute `/bin/ls`, the fact that we are in the jail means we cannot see the actual `/bin` of the filesystem. Therefore, we must create a directory named `bin` and copy `ls` into it. And `ls` needs several libraries, as shown with the `ldd` command:

```
jdoe$ ldd /bin/ls
libtermcap.so.2 => /lib/libtermcap.so.2 (0x40033000)
libc.so.6 => /lib/i686/libc.so.6 (0x40037000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

This all means we will need to create a directory in our jail named `lib`, and we'll need to copy over these libraries into our directory. This is a lot of work! However, a few software packages may help make `chroot` jails easier to deal with.

Jail The Jail Chroot Project, located at <http://www.gsync.inf.uc3m.es/~assman/jail/>, is a group of C and Perl programs that can help automate the creation of a `chroot` jail. The `addjailsw` program will “automagically” set up the proper directories and files for us. From the `/bin` directory, `addjailsw` copies a number of commonly used debugging programs (such as `cp` and `more`) into the jail. It then runs the program you want to `chroot` under `strace` (to watch any system calls the program makes) and copies any of the support files it needs into the `chroot` jail. It also includes programs to use for creating new users in the `chroot` jail as well as for setting up often needed `/dev/` files such as `/dev/null`.

CAUTION

The Jail Chroot Project program is a bit overzealous about the files it populates in the `chroot` jail, so you should make sure to delete extraneous files when you're done debugging your jail. Particularly unneeded are the files in `/bin` and `/usr/bin`, and the real entries in `/etc/passwd`.

Cell Carole Fennelly wrote a helpful article (<http://www.theadamsfamily.net/~erek/snort/cell>) that includes source code to use in creating a jail cell. Using the shell scripts and helpful example configuration files that are provided, you can create secure jails with relative ease for any software package.

Zorp's Jailer Zorp is a modular proxy firewall created by BalaBit IT Ltd (<http://www.balabit.hu/en/news>). It includes a program named `jailer` (available at <http://www.balabit.hu/downloads/jailer>), which can be used to create `chroot` jails easily.

Jailer uses a `config` file that lists the packages (Debian packages, not RPM packages) that should be installed in the jail, in addition to any extra files or links that need to be created. It determines which package dependencies exist and installs them into the `chroot` jail automatically.

Using Linux Capabilities to Reduce the Risks of root

Linux has internal support for POSIX capabilities. This approach is a mechanism for providing discrete capabilities to processes that is different from the traditional all-powerful mechanism of `root`. This support will allow a process to run with the exact set of permissions it needs to perform its specific task.

TIP

After ten years of failing to complete the capability-based security model (POSIX 1003.1e) specification, the committee in charge dropped the draft. Though Linux and other systems are implementing capabilities, do not expect them to be handled in exactly the same way among different UNIX-like operating systems.

A process can be given full control of a set of capabilities such that it can pass them onto other programs the process runs, or such that you can restrict these capabilities to a program but not its children. This means you can offer permissions for a process that cannot be granted to other programs, which prevents attacks in which a cracker tricks a program into executing shell code (which traditionally runs `/bin/sh`) with higher privileges.

Take, for instance, a program that needs to bind a low-numbered port (less than 1024), which is traditionally restricted from all but the `root` user. If you set the program's `CAP_NET_BIND_SERVICE` capability, it is allowed to bind low ports, yet it does not have the other access held by `root`, such as the ability to read from and write to any file.

Using capabilities allows you to set extremely detailed permissions for users and programs, which can greatly enhance security. If you are writing a `suid` program, we strongly suggest you consider removing all but the necessary capabilities (using `syscall(SYS_capset . . .)`) at the beginning of the program to reduce the power it could have if compromised.

For further reading about Linux capabilities, see <http://www.kernel.org/pub/linux/libs/security/linux-privs/kernel-2.2/capfaq-0.2.txt>.

POORLY WRITTEN CODE

Poorly written code is a nuisance. No one likes programs that crash, lose data, or eat up your CPU cycles.

However annoying buggy code may be in normal usage, we're used to it. We expect it. We make frequent backups, and we upgrade often in order to replace old bugs with new and improved bugs.

But the real problem comes when bugs in programs can be abused by an attacker to gain access to our machine. Buggy code is the front door to many an attack, both remote and local. Since there are several kinds of programming errors that crop up in many areas of security, we'll cover them here first.

Failing to Drop Privileges

A program that has the `suid` or `sgid` bit set must be more carefully programmed than other software. Since the program is executing as a user or group other than the actual person who invoked it, errors could result in this person gaining access to things normally forbidden.



wmtv Command Execution Vulnerability

The `wmtv` program, a dockable Video4Linux TV player for Window Maker, had a feature that let a user launch an external command when the video window was double-clicked. All the user needed to do was start `wmtv` with an `-e programname` option.

Unfortunately, in order for `wmtv` to be able to open the raw video device, it needed to run as `root`. So in order for normal users to be able to run it, `wmtv` needed to be a `suid root` program.

Unfortunately, it failed to drop `root` privileges before the program launched. All a local user needed to do to become `root` was run

```
jdoe$ wmtv -e xterm moviefile
```

and then double-click. A new `xterm` running as `root` would instantly pop up.



Write the Program to Drop root Privileges When Necessary

When the exploit was discovered, the author immediately modified the program to drop `root` privileges before executing the command specified by the `-e` option. This fix was made available quickly, and all users were encouraged to upgrade immediately. In general, the code needed to do reset the `gid` and `uid` looks like this:

```
#define bail(s) { perror(s); exit(1); }

uid_t uid_cur = getuid();
gid_t gid_cur = getgid();

if ( setgid(gid_cur) < 0)
    bail("setgid");
if ( setuid(uid_cur) <0 )
    bail("setuid");
execl("/path/to/program" ...)
    bail("execl failed")
```

This resets the `uid` and `gid` to the values of the original caller. (Assuming you haven't swapped them along the way somewhere.)

TIP

As usual, staying informed and upgrading immediately are the most important countermeasures we can perform.

Buffer Overflows

Buffer overflows occur when you try to stick data into a space that is too small. Many languages, C and C++ being the most popular, do not have bounds checking built in—no warnings at compile time, no protection at run time. If you try to stick 100 bytes into an array that was only allocated 25 bytes, those extra 75 bytes will just overwrite other memory.

Think of it this way. In the real world, it's impossible to stick an elephant into the freezer. When you try in the programming world, you get an elephant that is part in the freezer, part in the refrigerator, and other parts of your kitchen too.

NOTE

Buffer overflows are often the cause of the "Memory Fault: (core dumped)" messages you see more often than you'd like to admit. If you see this, you're running buggy code, and should seriously think about a code audit.

Let's take a quick look at a simple overflowable program:

```
#include <stdio.h>

main () {
    char userinput[99999];

    gets(userinput);    /* bad idea - use fgets instead */

    overflow(userinput);
    exit(0);
}

int overflow( char *data) {
    char filename[1];

    strcpy(filename, data);
    /* do something */
    return;
}
```

The user is given the opportunity to input 99999 characters, and later this data is copied with `strcpy` into a one byte array. Since `strcpy` doesn't do any bounds checking, this ends up copying the user's data into the array, and then continues into other memory

locations. Eventually (pretty soon in our example, actually) it will overwrite the stack—a location of memory that tells the `overflow` function how to get back to `main` when `return` is called.

Now here's the real trick—the attacker can craft his data to contain valid machine code, and force the function to run it, instead of returning to `main`. This code is traditionally called shell code, because most buffer overflow exploits attempt to run a copy of `/bin/sh`, or make a `suid root` copy of `/bin/sh` in `/tmp`.

NOTE

The most famous description about creating buffer overflows is Aleph One's article "Smashing the Stack for Fun and Profit," from Phrack 49, available at <http://www.securityfocus.com/data/library/P49-14.txt>.

**Example Buffer Overflow Attacks**

<i>Popularity:</i>	8
<i>Simplicity:</i>	5
<i>Impact:</i>	9
<i>Risk Rating:</i>	7

Buffer overflows in a `suid` program can be disastrous. Since the program runs with different privileges than the invoking user's privileges, a cracker can exploit a buffer overflow to gain privileges. In the case of `suid root` programs, this means that the cracker can be given an instant `root` prompt, from which she can do any damage she cares to do.

Even in cases of `suid` or `sgid` programs under a non-`root` user, an overflow can be leveraged in a less direct way. Suppose, for example, that the `/usr/bin/cu` program is vulnerable to a buffer overflow. `cu` has the following permissions:

```
jdoue$ ls -l /usr/bin/cu
-r-sr-sr-x 1 uucp uucp 127924 Mar 7 2000 /usr/bin/cu*
```

If a buffer overflow occurs in `cu`, the cracker can gain `uucp` user and group permissions. `cu` is used to establish connections to other systems, and passwords are often hard-coded in the `/etc/uucp` area that are readable only by `uucp`. These passwords become available to the attacker.

Worse yet, since the program is owned by `uucp`, the attacker can overwrite the program with a trojaned version and remove the `suid` bit. When `root` next runs the `cu` command, the command will run as user `root`, and the attacker can compromise the `root` account.

Another example is the `man` program, which is generally a `sgid` `man` to allow it to save preformatted man pages. If any man pages are writable by the `man` group (which is common) and the `man` program is compromised (which has occurred on several occasions), an attacker can rewrite man pages.

This may seem uninteresting. However, the macro languages used by man pages are stronger than you might think. Many of them have the ability to call external programs. All an attacker needs to do is modify a man page to execute `chmod 666 /etc/shadow` and then wait until the `root` user reads that man page.

NOTE

Most Linux distributions do handle man pages safely, because the `troff` program, which is called by `man` to format the manual page, will disable unsafe macros first. However, this is not the case for all Linux distributions or other UNIX-like operating systems.

— Keep Your Programs Up to Date

The single most important step to avoid being cracked, including being compromised by your buffer overflows, is to keep your programs up to date. Subscribe to security mailing lists, especially those specific to your Linux distribution. Be prepared to upgrade packages when a vulnerability is found.

— Turn Off the `suid` and `sgid` Bit

A locally available buffer overflow gives an attacker an advantage only when the program is a `suid` or `sgid` program. Thus, you can also turn off the `suid` or `sgid` bit in programs that you do not use or simply uninstall them. For example, if you do not need to dial out on a modem with `cu`, uninstall it.

— Libsafe

`Libsafe` (<http://www.research.avayalabs.com/project/libsafe/>) is a dynamically loadable library that implements a layer of software that intercepts all function calls to library functions known to be vulnerable. A substitute version of a vulnerable function is invoked that is functionally equivalent to the original but contains any possible overflows within the current stack frame, preventing the overwriting of data that would allow a cracker to hijack the program.

`Libsafe` can be used system wide with little performance overhead. It has been shown to detect several known attacks and *potentially* stops all buffer overflow attacks that are not currently known.

— Openwall Project's Non-exec Stack Kernel Patch

The Openwall Project's Linux kernel patch (<http://www.openwall.com/linux/>) is a collection of security-related features for the Linux kernel, one of which makes the stack area non-executable. This defends against the easiest way an attacker could use to execute arbitrary instructions via a buffer overflow vulnerability. It is by no means a complete solution, but will stop the average script-kiddie, and give you notification of the attack that has occurred.

— StackGuard

`StackGuard` (<http://www.immunix.org/stackguard.html>) is a development of Immunix. It is a compiler that hardens programs against stack smashing attacks (buffer overflows of the function stack). Programs compiled with `StackGuard` do not require any source code changes. `StackGuard` protects the program by protecting return addresses on the stack from being altered by placing a "canary" word next to the return address. If the canary word has been altered, you know that an attack has been attempted, and `StackGuard` will raise an alert and terminate the program.

Format String Bugs

The problems with buffer overflows have been widely known since Aleph One released “Smashing the Stack.” True, this does not mean that the same bugs don’t bite us, but at least it’s a bug we’ve all been familiar with.

In 2000, a new class of vulnerabilities was found—the format string bug. Overnight, programmers the world over were performing code reviews of Open Source software to fix this innocuous-looking bug.

Format string attacks are another, newer way for crackers to gain root access.



Format String Attacks

<i>Popularity:</i>	6
<i>Simplicity:</i>	4
<i>Impact:</i>	9
<i>Risk Rating:</i>	6

A format string bug occurs when a programmer wants to print a simple string using one of the functions that supports formats, such as `printf()` or `syslog()`. The correct way to do this is to enter the following:

```
printf("%s", str);
```

However, in the interest of saving time and six characters, many programmers instead write the command without the first argument:

```
printf(str);
```

In this case, the programmer wants to print the string verbatim but instead has supplied a format string, which `printf()` scans for all the standard options such as `%s`, `%d`, and `%f`.

One of the options available in format strings is the `%n`, which writes the number of bytes printed. An attacker can carefully craft a format string that includes random data, format options, and possibly the exploit code to run as well. Using the `%n` option, arbitrary memory can be overwritten, such as the return pointer, causing the attacker’s code to run.

NOTE

Actually, `%n` will write the number of bytes that could have been printed. Thus if you used `%.100d`, the number of bytes would be incremented by 100. This actually makes format string bugs easier to write, since you don’t need to actually have 100 bytes in there.

Getting attacker-supplied input into this string is easier than you might think. If the user makes an error, the program might attempt to log in via `syslog()` or `sprintf()`, and if the error routine includes the violation itself in the output, the attacker can supply whatever she wishes.

Format String Countermeasures

Many format string attacks use the same principle used with buffer overflows—overwriting the function’s return call—and can thus be prevented by the buffer overflow countermeasures described previously in this chapter.

FormatGuard

FormatGuard (<http://www.immunix.org/formatguard.html>), proposed by Michael Frantzen and implemented by WireX, is designed to propose a general solution to the format bug vulnerability. It is part of the Immunix distribution (along with StackGuard). It provides a macro definition of `printf()` for calls with one argument, two arguments, and so on, up to 100 arguments. Each macro in turn calls a safe wrapper for `printf()` that rejects any arguments that are passed in above the number of format placeholders.

Libsafe

Libsafe (<http://www.research.avayalabs.com/project/libsafe/>) provides protection for format string vulnerabilities as well as buffer overflows.

CAUTION

Although buffer overflows and format string vulnerabilities are now well known, programmers continue to write sloppy code that can be exploited.

Race Conditions

The Linux kernel is a multitasking operating system, and many processes are running on the machine at same time. In actuality, the kernel doles out CPU access to each process as it sees fit. You are able to read your email while you download the newest copy of Mozilla because the kernel provides each process CPU time as it sees fit, rather than waiting for one process to finish (nonmultitasking) or offer to let go of the CPU (cooperative multitasking).

A race condition occurs when a programming decision involves a check of some resource—such as “does the file `foo` exist”—followed by an action that depends on the result. Since the kernel may give CPU time to another process between those two sections of code, attackers could make changes to the system that make the results of that check invalid.



Race Condition Attacks

<i>Popularity:</i>	6
<i>Simplicity:</i>	3
<i>Impact:</i>	8
<i>Risk Rating:</i>	6

Let’s create a program that’s supposed to be run by `root` to create `.forward` entries for users, called ‘`raceForward`.’ All it does is take a username and email address on the

command line and create a `/home/username/.forward` file with the email address in it. Here's an example that would seem to be very paranoid:

```
#!/usr/bin/perl
#
# raceForward
#
# An example program vulnerable to two race conditions.

($username, $email) = @ARGV;
$FILE = "/home/$username/.forward";

($uid,$gid) = (getpwnam($username))[2,3]
    or die "No such user $username";

# Check file
if ( ($fileuid) = (stat $FILE)[4] ) {
    unless ( $fileuid == $uid ) {
        die "Something is amiss with ${username}'s .forward.";
    }
} else {
    # Make sure it's not a dangling symlink too!
    if ( ($fileuid) = (lstat $FILE)[4] ) {
        die "Whoa - dangling symlink! Trickery suspected!"
    }
}

# Excellent - it's safe!
open FORWARD, ">$FILE" or die;
print FORWARD "$email\n";
close FORWARD;
chown $uid,$gid, $FILE or warn "Whoa, can't chown it."
chmod 0600, $FILE or warn "Can't chmod the file."
```

The code checks for an existing `.forward` file very carefully. It makes sure that if there is one, that it is owned by the user. It also checks with `lstat` to make sure that the file isn't a dangling symlink (a symlink that points to a nonexistent file.) Assuming everything is safe, it opens the file for writing.

The problem is that between these `stat` checks and the actual `open`, the user could create the file. Imagine if he created a dangling symlink to `/etc/nologin`. This script would end up creating `/etc/nologin`, and suddenly no one except `root` can log into the machine. Of course the timing has to be perfect—that's why it's called a race condition. Any time this many security checks are used, you should consider something simpler.

You might think of replacing all the `stat` calls with this instead:

```
# just delete - don't bother checking unlink $FILE
```

However this is still vulnerable to a symlink attack before the `open` occurs, allowing the attacker to overwrite a file as root, or between the `open` and the `chown`, which would allow the attacker to take ownership of any file on the system.

— Use Atomic System Calls

The best way to avoid race conditions is to use functions that are atomic (system calls that execute uninterrupted inside the kernel). The `open()` system call can take an argument that says “only create this file if it does not exist.” Unfortunately you can’t use this with Perl’s standard `open` function, but you can with `sysopen`. Our code becomes:

```
#!/usr/bin/perl
# runForward - no longer vulnerable to a race condition.  Uses
# sysopen to avoid symlink open attack, and fchown system call
# to avoid symlink race between create and chown.

use POSIX;  require "syscall.ph";

($username,$email) = @ARGV;
($uid,$gid,$home) = (getpwnam($username))[2,3,7]      || die

$FILE = "$home/.forward";
unlink $FILE;  # if it fails, sysopen will catch it.

sysopen( FORWARD, $FILE, O_RDWR|O_CREAT|O_EXCL, 0600) || die;
syscall(&SYS_fchown, fileno(FORWARD), $uid, $gid)==0  || die;

print FORWARD "$email\n";
close FORWARD;
```

There are many possible flags you’d use (see `man perllopentut` for a list). The ones we used here were `O_RDWR` to open it in read/write mode, `O_CREAT` to create it if necessary, and `O_EXCL` which will refuse to create the file if it already exists as an actual file or as a symlink. Additionally, the file permission (0600 in octal) is set in the `sysopen` call itself, which means we need not use `chmod` at all.

In this case, it is the use of `O_EXCL` that guarantees we generate this file without a chance of a race condition. The kernel executes the `sysopen` call atomically—no other processes can affect the filesystem while it is executing. Additionally, the use of the `fchown` system call assures that we are changing the file ownership of the file we opened with `sysopen`, and thus no symlink race condition is possible.

The corresponding C code for a safe open would be

```
...fd = open("filename", 0600, O_RDWR|O_CREAT|O_EXCL);
```

Drop Privileges

If you have a privileged process that must perform actions on the behalf of a different user, you may find the easiest way to avoid complicated permission checks and race conditions is to simply have the process perform those actions as the other user directly. This is achieved by *forking* into two processes, and having the child become the target user with the `setuid` call and performing the necessary actions. In general, the more simple and atomic you can make your software, the less chance there is for a race condition.

Auditing Tools

Many tools are available for examining Open Source programs and finding common security flaws. Most of these are written to find problems in C code (since most Open Source programs are written in C).

LCLint

LCLint (<http://lclint.cs.virginia.edu/>) is a tool that checks for problems with C programs. It performs many of the traditional lint checks with the addition of violations of information hiding, inconsistent modification of caller-visible state, inconsistent use of global variables, memory management errors, dangerous data sharing or unexpected aliasing, using undefined memory, dereferencing a null pointer, and other types of problems.

LCLint's user's guide can be found at <http://lclint.cs.virginia.edu/guide/guide-full.html>.

Cqual

Cqual (<http://www.cs.berkeley.edu/~jfoster/cqual/>) is a type checker that allows the programmer to define type qualifiers. The programmer can then add qualifier annotations in his code and Cqual will perform qualifier inference to verify the correctness of the annotations.

RATS

RATS (<http://www.securesoftware.com/rats.php>), Rough Auditing Tool for Security, audits C, C++, Perl, Python, and PHP code. It will alert you to many common coding problems. It is useful as a first step in performing a code review, but it is not a replacement for it, nor does it bill itself as such.

— Flawfinder

Flawfinder (<http://www.dwheeler.com/flawfinder>) examines source code for security flaws (hence its name). Flawfinder looks for functions with known problems such as buffer overflows (`strcpy()`, `scanf()`, and so on), format bugs (`printf()`, `syslog()`, and so on), race conditions (`access()`, `chmod()`, and so on), potential shell metacharacter problems (`exec()`, `system()`, and so on), and random number generators (`random()` and so on). A list of hits is generated and reported, informing the programmer of possible security problems.

— SPIKE and Sharefuzz

SPIKE (<http://sourceforge.net/projects/spike/>) is a helpful protocol analysis and reproduction tool. Sharefuzz (<http://sourceforge.net/projects/sharefuzz/>) is a program used to analyze suid programs for buffer overflows using `LD_PRELOAD`.

SUMMARY

Crackers want control of your machine. Denying them access is possible with Linux if you anticipate what crackers will try to do and employ steps to stop them.

To secure your Linux machine successfully from attack, you need to know the basic security features available in the Linux operating system. Some of these are common to other UNIX-like operating systems, such as users, groups, file permissions, and process resources. Other features may be present in other systems but differ in their implementations, such as extended file attributes. Some of these features have analogs outside the UNIX world, whereas others—even the most simplistic file permissions—are foreign to nonmultiuser systems.

In the following chapters, we reveal security attacks that crackers perform and the countermeasures you can proactively take to protect your system. To fully understand these attacks and to be able to protect yourself adequately, understanding the basic ideas discussed in this chapter is essential.

