

PART SIX

Operating Systems Security

LINUX SECURITY

by Mick Bauer

Security Editor, Linux Journal

23.1 Introduction

23.2 Linux's Security Model

23.3 The Linux DAC in Depth: File-System Security

- Users, Groups, and Permissions
- Simple File Permissions
- Directory Permissions
- The Sticky Bit
- Setuid and Setgid
- Setgid and Directories
- Numeric Modes
- Kernel Space versus User Space

23.4 Linux Vulnerabilities

- Abuse of Programs Run “setuid root”
- Web Application Vulnerabilities
- Rootkit Attacks

23.5 Linux System Hardening

- OS Installation: Software Selection and Initial Setup
- Patch Management
- Network-Level Access Controls
- Antivirus Software
- User Management
- Logging
- Other System Security Tools

23.6 Application Security

- Running as an Unprivileged User/Group
- Running in a Chroot Jail
- Modularity
- Encryption
- Logging

23.7 Mandatory Access Controls

- SELinux
- Novell AppArmor

Like other general-purpose operating systems, Linux's wide range of features presents a broad attack surface. Even so, by leveraging native Linux security controls, carefully configuring Linux applications, and deploying certain add-on security packages, you can create highly secure Linux systems.

23.1 INTRODUCTION

Since Linus Torvalds created Linux in 1991, more or less on a whim, Linux has evolved into one of the world's most popular and versatile operating systems. Linux is free, open-sourced, and available in a wide variety of "distributions" targeted at almost every usage scenario imaginable. These distributions range from conservative, commercially supported versions such as Red Hat Enterprise Linux; to cutting-edge, completely free versions such as Ubuntu; to stripped-down but hyperstable "embedded" versions (designed for use in appliances and consumer products) such as uClinux.

The study and practice of Linux security therefore has wide-ranging uses and ramifications. New exploits against popular Linux applications affect many thousands of users around the world. New Linux security tools and techniques have just as profound of an impact, albeit a much more constructive one.

In this chapter we'll examine the Discretionary Access Control-based security model and architecture common to all Linux distributions and to most other UNIX-derived and UNIX-like operating systems (and also, to a surprising degree, to Microsoft Windows). We'll discuss the strengths and weaknesses of this ubiquitous model; typical vulnerabilities and exploits in Linux; best practices for mitigating those threats; and improvements to the Linux security model that are only slowly gaining popularity but that hold the promise to correct decades-old shortcomings in this platform.

23.2 LINUX'S SECURITY MODEL

Linux's traditional security model can be summed up quite succinctly: people or processes with "root" privileges can do anything; other accounts can do much less.

From the attacker's perspective, the challenge in cracking a Linux system therefore boils down to gaining root privileges. Once that happens, attackers can erase or edit logs; hide their processes, files, and directories; and basically redefine the reality of the system as experienced by its administrators and users. Thus, as it's most commonly practiced, Linux security (and UNIX security in general) is a game of "root takes all."

How can such a powerful operating system get by with such a limited security model? In fairness, many Linux system administrators fail to take full advantage of the security features available to them (features we're about to explore in depth). People can and do run robust, secure Linux systems by making careful use of native Linux security controls, plus selected add-on tools such as `sudo` or `Tripwire`. However, the crux of the problem of Linux security in general is that like the

UNIX operating systems on which it was based, Linux’s security model relies on **Discretionary Access Controls (DAC)**.

In the Linux DAC system, there are users, each of which belongs to one or more groups; and there are also **objects**: files and directories. Users read, write, and execute these objects, based on the objects’ **permissions**, of which each object has three sets: one each defining the permissions for the object’s user-owner, group-owner, and “other” (everyone else). These permissions are enforced by the Linux kernel, the “brain” of the operating system.

Because a process/program is actually just a file that gets copied into executable memory when run, permissions come into play twice with processes. Prior to being executed, a program’s file permissions restrict who can execute, access, or change it. When running, a process normally “runs as” (with the identity of) the user and group of the person or process that executed it.

Because processes “act as” users, if a running process attempts to read, write, or execute some other object, the kernel will first evaluate that object’s permissions against the process’s user and group identity, just as though the process was an actual human user. This basic transaction, wherein a **subject** (user or process) attempts some **action** (read, write, execute) against some **object** (file, directory, special file), is illustrated in Figure 23.1.

Whoever owns an object can set or change its permissions. Herein lies the Linux DAC model’s real weakness: The system **superuser** account, called “root,” has the ability to both take ownership and change the permissions of all objects in the system. And as it happens, it’s not uncommon for both processes and administrator-users to routinely run with root privileges, in ways that provide attackers with opportunities to hijack those privileges.

Those are the basic concepts behind the Linux DAC model. The same concepts in a different arrangement will come into play later when we examine Mandatory Access Controls such as SELinux. Now let’s take a closer look at how the Linux DAC implementation actually works.

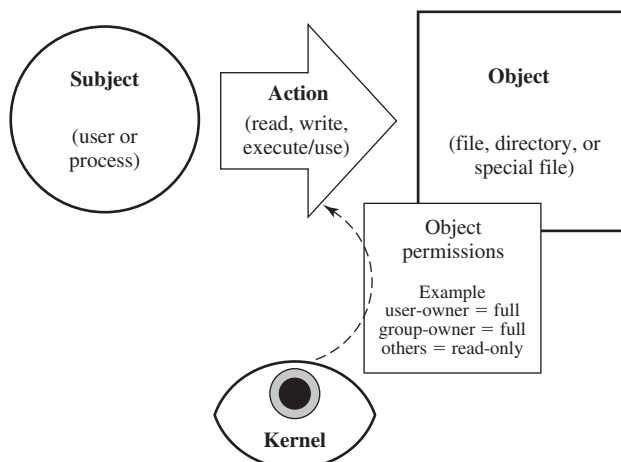


Figure 23.1 Linux Security Transactions

23.3 THE LINUX DAC IN DEPTH: FILE-SYSTEM SECURITY¹

So far, we haven't said anything about memory, device drivers, named pipes, and other system resources. Isn't there more to system security than users, files, and directories? Yes and no: In a sense, Linux treats *everything* as a file.

Documents, pictures, and even executable programs are very easy to conceptualize as files on your hard disk. But although we *think* of a directory as a container of files, in UNIX a directory is actually itself a file containing a list of other files.

Similarly, the CD-ROM drive attached to your system seems tangible enough, but to the Linux kernel, it too is a file: the “special” device-file `/dev/cdrom`. To send data from or write data to the CD-ROM drive, the Linux kernel actually reads to and writes from this special file. (Actually, on most systems, “`/dev/cdrom`” is a symbolic link to `/dev/hdb` or some other special file. And a symbolic link is in turn nothing more than a file that contains a pointer to another file.)

Other special files, such as named pipes, act as input/output (I/O) “conduits,” allowing one process or program to pass data to another. One common example of a named pipe on Linux systems is `/dev/urandom`: When a program reads this file, `/dev/urandom` returns random characters from the kernel's random number generator.

These examples illustrate how in Linux/UNIX, *nearly everything* is represented by a file. Once you understand this, it's much easier to understand why file-system security is such a big deal (and how it works).

Users, Groups, and Permissions

There are two things on a UNIX system that aren't represented by files: user accounts and group accounts, which for short we can call **users** and **groups**. (Various files contain information about a system's users and groups, but none of those files actually represents them.)

A user account represents someone or something capable of using files. As we saw in the previous section, a user account can be associated both with actual human beings and with processes. The standard Linux user account “lp,” for example, is used by the Line Printer Daemon (lpd): The lpd program runs as the user lp.

A group account is simply a list of user accounts. Each user account is defined with a **main group** membership, but may in fact belong to as many groups as you want or need it to. For example, the user “maestro” may have a main group membership in “conductors” and also belong to the group “pianists.”

A user's main group membership is specified in the user account's entry in `/etc/passwd`; you can add that user to additional groups by editing `/etc/group` and adding the username to the end of the entry for each group the user needs to belong to, or via the **usermod** command [see the `usermod(8)` manpage for more information].

¹This section is adapted from [BAUE04], with permission of the *Linux Journal*.

Listing 23-1 shows “maestro”’s entry in the file `/etc/passwd`, and Listing 23-2 shows part of the corresponding `/etc/group` file:

```
maestro:x:200:100:Maestro Edward Hizzersands:/home/
maestro:/bin/bash
```

Listing 23-1: An `/etc/passwd` Entry for the User “maestro”

```
conductors:x:100:
pianists:x:102:maestro,volodya
```

Listing 23-2: Two `/etc/group` Entries

In Listing 23-1, we see that the first field contains the name of the user account, “maestro;” the second field (“x”) is a placeholder for maestro’s password (which is actually stored in `/etc/shadow`); the third field shows maestro’s numeric userid (or “uid,” in this case “200”); and the fourth field shows the numeric groupid (or “gid,” in this case “100”) of maestro’s main group membership. The remaining fields specify a comment, maestro’s home directory, and maestro’s default login shell.

In Listing 23-2, from `/etc/group`, each line simply contains a groupname, a group-password (usually unused — “x” is a placeholder), numeric group-id (gid), and a comma-delimited list of users with “secondary” memberships in the group. Thus we see that the group “conductors” has a gid of “100”, which corresponds to the gid specified as maestro’s main group in Listing 23-1; and also that the group “pianists” includes the user “maestro” (plus another named “volodya”) as a secondary member.

The simplest way to modify `/etc/passwd` and `/etc/group` in order to create, modify, and delete user accounts is via the commands **useradd**, **usermod**, and **userdel**, respectively. All three of these commands can be used to set and modify groupmemberships, and all three commands are well documented in their respective manpages. (To see a quick usage summary, you can also type the command followed by “—help,” for example, “useradd —help”.)

So we’ve got user accounts, which are associated with different group accounts. Just what is all this good for?

Simple File Permissions

Each file on a UNIX system (which, as we’ve seen, means “practically every single thing on a UNIX system”) has two owners: a user and a group, each with its own set of permissions that specify what the user or group may do with the file (read it, write to it or delete it, and execute it). A third set of permissions pertains to **other**, that is, user accounts that don’t own the file or belong to the group that owns it.

Listing 23-3 shows a “long file-listing” for the file `/home/maestro/baton_dealers.txt`:

```
-rw-rw-r-- 1 maestro conductors 35414 Mar 25 01:38
baton_dealers.txt
```

Listing 23-3: File-Listing Showing Permissions

Permissions are listed in the order “user permissions, group permissions, other permissions.” Thus we see that for the file shown in Listing 23-3, its user-owner (“maestro”) may read and write/delete the file (“rw-”); its group-owner (“conductors”) may also read and write/delete the file (“rw-”); but that other users (who are neither “maestro” nor members of “conductors”) may only read the file.

There’s a third permission besides “read” and “write”: “execute,” denoted by “x” (when set). If maestro writes a shell script named “punish_bassoonists.sh”, and if he sets its permissions to “-rwxrw-r--”, then maestro will be able to execute his script by entering the name of the script at the command line. If, however, he forgets to do so, he won’t be able to run the script, even though he owns it. Permissions are usually set via the “chmod” command (short for “change mode”).

Directory Permissions

Directory permissions work slightly differently from permissions on regular files. “Read” and “write” are similar; for directories these permissions translate to “list the directory’s contents” and “create or delete files within the directory”, respectively. “Execute” is less intuitive; for directories, “execute” translates to “use anything within or change working directory to this directory”.

That is, if a user or group has execute permissions on a given directory, the user or group can list that directory’s contents, read that directory’s files (assuming those individual files’ own permissions include this), and change its own working directory to that directory, as with the command “cd”. If a user or group does not have execute permissions on a given directory, it will be unable to list or read anything in it, regardless of the permissions set on the things inside.

(Note that if you lack execute permissions on a directory but do have read permissions on an the directory, and you try to list its contents with ls, you will receive an error message that, in fact, lists the directory’s contents. But this doesn’t work if you have neither read nor execute permissions on the directory.)

Suppose our example system has a user named “biff” who belongs to the group “drummers”. And suppose further that his home directory contains a directory called “extreme_casseroles” that he wishes to share with his fellow percussionists. Listing 23-4 shows how biff might set that directory’s permissions:

```
bash-$ chmod g+rx extreme_casseroles
bash-$ ls -l extreme_casseroles
drwxr-x--- 8 biff drummers 288 Mar 25 01:38
extreme_casseroles
```

Listing 23-4: A Group-Readable Directory

Per Listing 23-4, only biff has the ability to create, change, or delete files inside extreme_casseroles. Other members of the group “drummers” may list its contents and cd to it. Everyone else on the system, however (except root, who is always all powerful), is blocked from listing, reading, cd-ing, or doing anything else with the directory.

The Sticky Bit

Suppose that our drummer friend Biff wants to allow his fellow drummers not only to read his recipes, but also to add their own. As we saw last time, all he needs to do is set the “group-write” bit for this directory, like this:

```
chmod g+w ./extreme_casseroles
```

There’s only one problem with this: “write” permissions include both the ability to create new files in this directory, but also to delete them. What’s to stop one of his drummer pals from deleting other people’s recipes? The “sticky bit.”

In older UNIX operating systems, the sticky bit was used to write a file (program) to memory so it would load more quickly when invoked. On Linux, however, it serves a different function: When you set the sticky bit on a directory, it limits users’ ability to delete things in that directory. That is, to delete a given file in the directory you must either own that file or own the directory, even if you belong to the group that owns the directory and group-write permissions are set on it.

To set the sticky bit, issue the command

```
chmod +t directory_name
```

In our example, this would be “chmod +t extreme_casseroles”. If we set the sticky bit on extreme_casseroles and then do a long listing of the directory itself, using “ls -ld extreme_casseroles”, we’ll see

```
drwxrwx--T 8 biff drummers 288 Mar 25 01:38
extreme_casseroles
```

Note the “T” at the end of the permissions string. We’d normally expect to see either “x” or “-” there, depending on whether the directory is “other-writable”. “T” denotes that the directory is not “other-executable” but has the sticky bit set. A lowercase “t” would denote that the directory is other-executable and has the sticky bit set.

To illustrate what effect this has, suppose a listing of the contents of extreme_casseroles/ looks like this (Listing 23-5):

```
drwxrwxr-T 3 biff drummers 192 2004-08-10 23:39.
drwxr-xr-x 3 biff drummers 4008 2004-08-10 23:39..
-rw-rw-r-- 1 biff drummers 18 2004-07-08 07:40
  chocolate_turkey_casserole.txt
-rw-rw-r-- 1 biff drummers 12 2004-08-08 15:10
  pineapple_mushroom_surprise.txt
drwxr-xr-x 2 biff drummers 80 2004-08-10 23:28 src
```

Listing 23-5: Contents of extreme_casseroles/

Suppose further that the user “crash” tries to delete the recipe file “pineapple_mushroom_surprise.txt”, which crash finds offensive. crash expects this to

work, because he belongs to the group “drummers” and the group-write bit is set on this file.

However, remember, biff just set the parent directory’s sticky bit. crash’s attempted deletion will fail, as we see in Listing 23-6 (user input in boldface):

```
crash@localhost:/extreme_casseroles> rm pineapple_
mushroom_surprise.txt
rm: cannot remove 'pineapple_mushroom_surprise.txt':
Operation not permitted
```

Listing 23-6: Attempting Deletion with Sticky Bit Set

The sticky bit only applies to the directory’s first level downward. In Listing 23-5 you may have noticed that besides the two nasty recipes, `extreme_casseroles/` also contains another directory, “src”. The contents of `src` will not be affected by `extreme_casseroles`’s sticky bit (though the directory `src` itself will be). If biff wants to protect `src`’s contents from group deletion, he’ll need to set `src`’s own sticky bit.

Setuid and Setgid

Now we come to two of the most dangerous permissions bits in the UNIX world: `setuid` and `setgid`. If set on an executable binary file, the `setuid` bit causes that program to “run as” its owner, no matter who executes it. Similarly, the `setgid` bit, when set on an executable, causes that program to run as a member of the group that owns it, again regardless of who executes it.

By *run as* we mean “to run with the same privileges as.” For example, suppose biff writes and compiles a C program, “killpineapple”, that behaves the same as the command “`rm /extreme_casseroles/pineapple_mushroom_surprise.txt`”. Suppose further that biff sets the `setuid` bit on `killpineapple`, with the command “`chmod +s ./killpineapple`”, and also makes it group executable. A long-listing of `killpineapple` might look like this:

```
-rwsr-xr-- 1 biff drummers 22 2004-08-11 23:01 killpineapple
```

If crash runs this program he will finally succeed in his quest to delete the Pineapple Mushroom Surprise recipe: `killpineapple` will run as though biff had executed it. When `killpineapple` attempts to delete `pineapple_mushroom_surprise.txt`, it will succeed because the file has user-write permissions and `killpineapple` is acting as its user-owner, biff.

Note that `setuid` and `setgid` are *very dangerous* if set on any file owned by root or any other privileged account or group. We illustrate `setuid` and `setgid` in this discussion so you understand what they do, not because you should actually *use* them for anything important. The command “`sudo`” is a much better tool for delegating root’s authority.

If you want a program to run `setuid`, that program must be group executable or other executable, for obvious reasons. Note also that the Linux kernel ignores the `setuid` and `setgid` bits on shell scripts; these bits only work on binary (compiled) executables.

setgid works the same way, but with group permissions: If you set the setgid bit on an executable file via the command “`chmod g+s filename`”, and if the file is also “other-executable” (`-r-xr-sr-x`), then when that program is executed it will run with the group-ID of the file rather than of the user who executed it.

In the preceding example, if we change killpineapple’s “other” permissions to “r-x” (`chmod o+x killpineapple`) and make it setgid (`chmod g+s killpineapple`), then no matter who executes killpineapple, killpineapple will exercise the permissions of the “drummers” group, because drummers is the group-owner of killpineapple.

Setgid and Directories

Setuid has no effect on directories, but setgid does, and it’s a little nonintuitive. Normally, when you create a file, it’s automatically owned by your user ID and your (primary) group ID. For example, if biff creates a file, the file will have a user-owner of “biff” and a group-owner of “drummers” (assuming that “drummers” is biff’s primary group, as listed in `/etc/passwd`).

Setting a directory’s setgid bit, however, causes any file created in that directory to inherit the directory’s group-owner. This is useful if users on your system tend to belong to secondary groups and routinely create files that need to be shared with other members of those groups.

For example, if the user “animal” is listed in `/etc/group` as being a secondary member of “drummers” but is listed in `/etc/passwd` as having a primary group of “muppets”, then animal will have no trouble creating files in the `extreme_casseroles/` directory, whose permissions are set to `drwxrwx-T`. However, by default animal’s files will belong to the group muppets, not to drummers, so unless animal manually reassigns his files’ group-ownership (`chgrp drummers newfile`) or resets their other-permissions (`chmod o+rw newfile`), then other members of drummers won’t be able to read or write animal’s recipes.

If, on the other hand, biff (or root) sets the setgid bit on `extreme_casseroles/` (`chmod g+s extreme_casseroles`), then when animal creates a new file therein, the file will have a group-owner of “drummers”, just like `extreme_casseroles/` itself. Note that all other permissions still apply: If the directory in question isn’t group-writable, then the setgid bit will have no effect (because group members won’t be able to create files inside it).

Numeric Modes

So far we’ve been using mnemonics to represent permissions: “r” for read, “w” for write, and so on. But internally, Linux uses numbers to represent permissions; only user-space programs display permissions as letters. The `chmod` command recognizes both mnemonic permission modifiers (“`u+rw,go-w`”) and **numeric modes**.

A numeric mode consists of four digits: As you read left to right, these represent special permissions, user permissions, group permissions, and other permissions (where, you’ll recall, “other” is short for “other users not covered by user permissions or group permissions”). For example, 0700 translates to “no special permissions set, all user permissions set, no group permissions set, no other permissions set.”

Each permission has a numeric value, and the permissions in each digit-place are additive: The digit represents the sum of all permission-bits you wish to set. If,

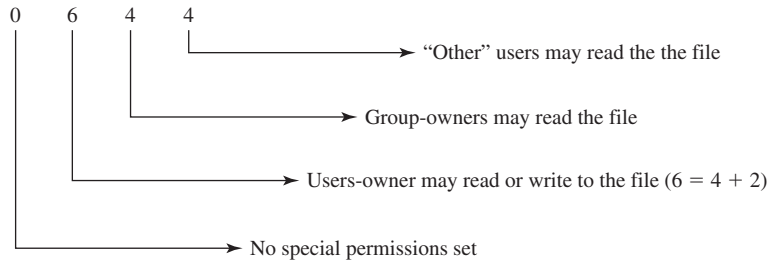


Figure 23.2 Permissions on mycooolfile

for example, user permissions are set to “7”, this represents 4 (the value for “read”) plus 2 (the value for “write”) plus 1 (the value for “execute”).

As just mentioned, the basic numeric values are 4 for read, 2 for write, and 1 for execute. (You can remember these by mentally repeating the phrase, “read-write-execute, 4-2-1.”) Why no “3,” you might wonder? Because (a) these values represent bits in a binary stream and are therefore all powers of 2; and (b) this way, no two combination of permissions have the same sum.

Special permissions are as follows: 4 stands for `setuid`, 2 stands for `setgid`, and 1 stands for sticky bit. For example, the numeric mode 3000 translates to “`setgid` set, stickybit set, no other permissions set” (which is, actually, a useless set of permissions).

Here’s one more example of a numeric mode. If I issue the command “`chmod 0644 mycooolfile`,” I’ll be setting the permissions of “mycooolfile” as shown in Figure 23.2.

For a more complete discussion of numeric modes, see the Linux “info” page for “coreutils,” node “Numeric Modes” (that is, enter the command “`info coreutils numeric`”).

Kernel Space versus User Space

It is a simplification to say that users, groups, files, and directories are all that matter in the Linux DAC: Memory is important, too. Therefore, we should at least briefly discuss kernel space and user space.

Kernel space refers to memory used by the Linux kernel and its loadable modules (e.g., device drivers). **User space** refers to memory used by all other processes. Because the kernel enforces the Linux DAC and, in real terms, dictates system reality, it’s extremely important to isolate kernel space from user space. For this reason, kernel space is never swapped to hard disk.

It’s also the reason that only root may load and unload kernel modules. As we’re about to see, one of the worst things that can happen on a compromised Linux system is for an attacker to gain the ability to load kernel modules.

23.4 LINUX VULNERABILITIES

In this section we’ll discuss the most common weaknesses in Linux systems.

First, a bit of terminology. A **vulnerability** is a specific weakness or security-related bug in an application or operating system. A **threat** is the combination of a vulnerability, an attacker, and a means for the attacker to exploit the vulnerability (called an **attack vector**).

Historically, some of the most common and far-reaching vulnerabilities in default Linux installations (unpatched and unsecured) have been

- Buffer overflows
- Race conditions
- Abuse of programs run “setuid root”
- Denial of service (DoS)
- Web application vulnerabilities
- Rootkit attacks

While you’ve already had exposure to most of these concepts earlier in this book, let’s take a closer look at how several of them apply to Linux.

Abuse of Programs Run “setuid root”

As we discussed in the previous section, any program whose “setuid” permission bit is set will run with the privileges of the user that owns it, rather than those of the process or user executing it. A **setuid root** program is a root-owned program with its setuid bit set; that is, a program that runs as root *no matter who executes it*.

If a setuid root program can be exploited or abused in some way (for example, via a buffer overflow vulnerability or race condition), then otherwise unprivileged users may be able to use that program to wield unauthorized root privileges, possibly including opening a **root shell** (a command-line session running with root privileges).

Running setuid root is necessary for programs that need to be run by unprivileged users yet must provide such users with access to privileged functions (for example, changing their password, which requires changes to protected system files). But such a program must be programmed very carefully, with impeccable user-input validation, strict memory management, and so on. That is, the program must be *designed* to be run setuid (or setgid) root. Even then, a root-owned program should only have its setuid bit set if absolutely necessary.

Due to a history of abuse against setuid root programs, major Linux distributions no longer ship with unnecessary setuid-root programs. But system attackers still scan for them.

Web Application Vulnerabilities

This is a very broad category of vulnerabilities, many of which also fall into other categories in this list. It warrants its own category because of the ubiquity of the World Wide Web: There are few attack surfaces as big and visible as an Internet-facing Web site.

While Web applications written in scripting languages such as PHP, Perl, and Java may not be as prone to classic buffer overflows (thanks to the additional layers of abstraction presented by those languages’ interpreters), they’re nonetheless prone to similar abuses of poor input handling, including cross-site scripting, SQL code injection, and a plethora of other vulnerabilities described in depth by the Open Web Application Security Project on the Project’s Web site (<http://www.owasp.org>).

Nowadays, few Linux distributions ship with “enabled-by-default” Web applications (such as the default cgi scripts included with older versions of the Apache Web server). However, many users install Web applications with known vulnerabilities, or write custom Web applications having easily identified and easily exploited flaws.

Rootkit Attacks

This attack, which allows an attacker to cover her tracks, typically occurs *after* root compromise: If a successful attacker is able to install a rootkit before being detected, all is very nearly lost.

Rootkits began as collections of “hacked replacements” for common UNIX commands (ls, ps, etc.) that behaved like the legitimate commands they replaced, except for hiding an attacker’s files, directories and processes. For example, if an attacker was able to replace a compromised Linux system’s ls command with a rootkit version of ls, then anyone executing the ls command to view files and directories would see everything except the attacker’s files and directories.

In the Linux world, since the advent of **loadable kernel modules** (LKMs), rootkits have more frequently taken the form of LKMs. This is particularly devious: An **LKM rootkit** does its business (covering the tracks of attackers) *in kernel space*, intercepting system calls pertaining to any user’s attempts to view the intruder’s resources.

In this way, files, directories, and processes owned by an attacker are hidden even to a compromised system’s standard, un-tampered-with commands, including customized software. Besides operating at a lower, more global level, another advantage of the LKM rootkit over traditional rootkits is that system integrity-checking tools such as Tripwire won’t generate alerts from system commands being replaced.

Luckily, even LKM rootkits do not always ensure complete invisibility for attackers. Many traditional and LKM rootkits can be detected with the script **chkrootkit**, available at www.chkrootkit.org. In general, however, if an attacker gets far enough to install an LKM rootkit, your system can be considered to be completely compromised; when and if you detect the breach (e.g., via a defaced Website, missing data, suspicious network traffic, etc.), the only way to restore your system with any confidence of completely shutting out the intruder will be to erase its hard disk (or replace it, if you have the means and inclination to analyze the old one), reinstall Linux, and apply all the latest software patches.

23.5 LINUX SYSTEM HARDENING

We’ve seen how Linux security is supposed to work, and how it most typically fails. The remainder of this chapter will focus on how to mitigate Linux security risks at the system and application levels. This section, obviously, deals with the first of these: OS-level security tools and techniques that protect the entire system. The final section in this chapter, on mandatory access controls, also describes system-level controls, but because this is both an advanced topic and an emerging technology (in the Linux world), we’ll consider it separately from the more fundamental controls in this section.

OS Installation: Software Selection and Initial Setup

Linux system security begins at operating system installation time: one of the most critical, system-impacting decisions a system administrator makes is what software will run on the system. Because it's hard enough for the typical, commonly over-worked system administrator to find the time to secure a system's critical applications, an unused application is liable to be left in a default, unhardened and unpatched state. Therefore, it's very important that from the start, careful consideration be given to which applications should be installed, and which should not.

What software should you not install? Common sense should be your guide: for example, an SMTP (e-mail) relay shouldn't need the Apache Web server; a database server shouldn't need an office productivity suite such as OpenOffice; and so on.

Given the plethora of roles Linux systems play (desktops, servers, laptops, firewalls, embedded systems, to name just a few), it's impossible to do much more than generalize in enumerating what software one shouldn't install. Nonetheless, here is a list of software packages that should seldom, if ever, be installed on hardened servers, especially Internet-facing servers:

- **X Window System:** Servers are usually remotely controlled via the Secure Shell, not locally via standard desktop sessions. Even if they are, X's history of security vulnerabilities makes plaintext-console sessions a safer choice for local access.
- **RPC Services:** Remote Procedure Call is a great convenience for developers, but both difficult to track through firewalls and too reliant on the easily spoofed UDP protocol.
- **R-Services:** rsh, rlogin, and rcp use only cleartext authentication (which can be eavesdropped) or source-IP-address-based authentication (which can sometimes be spoofed). The Secure Shell (SSH), which uses strong encryption, was created specifically to replace these commands and should be used instead.
- **inetd:** The Internet Daemon (inetd) is a poorly scaling means of starting critical network daemons, which should instead be started autonomously. inetd also tends, by default, to leave various unnecessary and potentially insecure services enabled, including RPC applications.
- **SMTP Daemons:** Traditionally, the Simple Mail Transport Protocol (SMTP) daemon Sendmail is enabled by default on many Linux distributions, despite Sendmail's history of security problems. As it happens, this is unnecessary on any system that doesn't need to receive relayed e-mail (i.e., that isn't an actual SMTP server). If a system only needs to send e-mail, Sendmail can be invoked as needed as a command and should not be left running as a daemon.
- **Telnet and other cleartext-logon services:** Because it passes logon credentials (usernames and passwords) over the network unencrypted, exposing them to eavesdroppers, telnet is no longer a viable tool for remote system access (and certainly not remote administration) via untrusted networks. The Secure Shell (SSH) is almost always a better choice than telnet. FTP, POP3, and IMAP also expose user credentials in this way, though many modern FTP, POP3, and IMAP server applications now support SSL or TLS encryption.

In addition to initial software selection and installation, Linux installation utilities also perform varying amounts of initial system and software configuration, including some or all of the following:

- Setting the root password
- Creating a non-root user account
- Setting an overall system security level (usually influencing initial file-permission settings)
- Enabling a simple host-based firewall policy
- Enabling SELinux or Novell AppArmor (see Section 23.7)

Patch Management

Carefully selecting what gets installed (and what doesn't get installed) on a Linux system is an important first step in securing it. All the server applications you do install, however, must be configured securely (the subject of Section 23.6), and they must also be kept up to date with security patches.

The bad news with patching is that you can never win the “patch rat-race”: There will always be software vulnerabilities that attackers are able to exploit for some period of time before vendors issue patches for them. (As yet unpatchable vulnerabilities are known as **zero-day**, or 0-day, vulnerabilities.)

The good news is, modern Linux distributions usually include tools for automatically downloading and installing security updates, which can minimize the time your system is vulnerable to things against which patches *are* available. For example, Red Hat, Fedora, and CentOS include **up2date** (**YUM** can be used instead); SuSE includes **YaST Online Update**; and Debian uses **apt-get**, though you must run it as a cron job for automatic updates.

Note that on change-controlled systems, you should not run automatic updates, because security patches can, on rare but significant occasions, introduce instability. For systems on which availability and uptime are of paramount importance, therefore, you should stage all patches on test systems before deploying them in production.

Network-Level Access Controls

One of the most important attack vectors in Linux threats is the network. A layered approach to security addresses not only actual vulnerabilities (e.g., patching and application hardening) but also the means by which attackers might exploit them (e.g., the network). Network-level access controls (that is, controls that restrict access to local resources based on the IP addresses of the networked systems attempting to access them) are therefore an important tool in Linux security.

Libwrappers and TCP Wrappers One of the most mature network access control mechanisms in Linux is libwrappers. In its original form, the software package TCP Wrappers, the daemon `tcpd` is used as a “wrapper” process for each service initiated by `inetd`.

Before allowing a connection to any given service, `tcpd` first evaluates access controls defined in the files `/etc/hosts.allow` and `/etc/hosts.deny`: If the transaction matches any rule in `hosts.allow` (which `tcpd` parses first), it's allowed. If no rule in

hosts.allow matches, tcpd then evaluates the transaction against the rules in hosts.deny; if any rule in hosts.deny matches, the transaction is logged and denied, but is otherwise permitted.

These access controls are based on the name of the local service being connected to, on the source IP address or hostname of the client attempting the connection, and on the username of the client attempting the connection (that is, the owner of the client process). Note that client usernames are validated via the **ident** service, which unfortunately is trivially easy to forge on the client side and makes this criterion's value questionable.

The best way to configure TCP Wrappers access controls is therefore to set a “deny all” policy in hosts.deny, such that the only transactions permitted are those explicitly specified in hosts.allow.

Because, as mentioned earlier, inetd is essentially obsolete, TCP Wrappers is no longer used as commonly as libwrappers, a system library that allows applications to defend themselves by leveraging /etc/hosts.allow and /etc/hosts.deny without requiring tcpd to act as an intermediary. In other words, libwrapper-aware applications can use the access controls in hosts.allow and hosts.deny via system calls provided by libwrappers.

Using iptables for “Local Firewall” Rules While libwrappers and TCP Wrappers are ubiquitous and easy to use, neither is nearly so powerful as the Linux kernel's native firewall mechanism, **netfilter**. Because netfilter is commonly referred to by the name of its user-space front end, **iptables**, we'll use the latter term here.

iptables is as useful run on multi-interface firewall systems that protect large networks, as it is when run on ordinary servers and desktop systems for local protection. Unsurprisingly, the iptables command has a steep learning curve, particularly for users who aren't network engineers. (Entire books, such as [SUEH05], are dedicated to this one command.)

Nearly all Linux distributions, however, now include utilities for automatically generating “personal” (local) firewall rules, especially at installation time. Typically, they prompt the administrator/user for local services that external hosts should be allowed to reach, if any (e.g., HTTP on TCP port 80, HTTPS on TCP port 443, and SSH on TCP port 22), and then generate rules that

- Allow incoming requests to those services;
- Block all other inbound (externally-originating) transactions; and
- Allow all outbound (locally-originating) services;

Note the last item: The assumption here is that all outbound network transactions are legitimate. However, this assumption does not hold if the system is compromised by a human attacker or by malware (e.g., a worm). On the one hand, if an attacker achieves root compromise, he or she can reconfigure iptables anyhow; on the other hand, if an attacker doesn't quite make it to root, then granular “egress rules” (allowing only selected outbound transactions) can at least limit the attacker's ability to connect back to his or her home system, scan and attack other systems, and engage in other potentially harmful network activity.

In cases in which this level of caution is justified, it may be necessary to create more complex iptables policies than your Linux installer's firewall wizard can

provide. Many people manually create their own startup script for this purpose (an iptables “policy” is actually just a list of iptables commands), but a tool such as Shorewall or Firewall Builder may instead be used.

Antivirus Software

Historically, Linux hasn’t been nearly so vulnerable to viruses as other operating systems (e.g., Windows). This may be due less to Linux’s being inherently more secure than to its lesser popularity as a desktop platform: Virus writers wanting to maximize the return on their efforts prefer to target Windows because of its ubiquity.

To some extent, then, Linux users have tended not to worry about viruses. To the degree that they have, most Linux system administrators have tended to rely on keeping up to date with security patches for protection against malware, which is arguably a more proactive technique than relying on signature-based antivirus tools.

And indeed, prompt patching of security holes is an effective protection against worms, which have historically been a much bigger threat against Linux systems than viruses. A worm is simply an automated network attack that exploits one or more specific application vulnerabilities. If those vulnerabilities are patched, the worm won’t infect the system.

Viruses, however, typically abuse the privileges of whatever user unwittingly executes them. Rather than actually exploiting a software vulnerability, the virus simply “runs as” the user. This may not have system-wide ramifications so long as that user isn’t root, but even relatively unprivileged users can execute network client applications, create large files that could fill a disk volume, and perform any number of other problematic actions.

Unfortunately, there’s no security patch to prevent users from double-clicking on e-mail attachments or loading hostile Web pages. Furthermore, as Linux’s popularity continues to grow, especially as a general-purpose desktop platform (versus its currently-prevalent role as a back-end server platform), we can expect Linux viruses to become much more common. Sooner or later, therefore, antivirus software will become much more important on Linux systems than it is presently.

(Nowadays, it’s far more common for antivirus software on Linux systems to be used to scan FTP archives, mail queues, etc., for viruses that target *other* systems than to be used to protect the system the antivirus software actually runs on.)

There are a variety of commercial and free antivirus software packages that run on (and protect) Linux, including products from McAfee, Symantec, and Sophos; and the free, open-source tool ClamAV.

User Management

As you’ll recall from Sections 23.2 and 23.3, the guiding principles in Linux user account security are as follows:

- Be very careful when setting file and directory permissions;
- Use group memberships to differentiate between different roles on your system; and
- Be extremely careful in granting and using root privileges.

Let's discuss some of the nuts and bolts of user- and group-account management, and delegation of root privileges. First, some commands.

You'll recall that in Section 23.3 we used the **chmod** command to set and change permissions for objects belonging to existing users and groups. To create, modify, and delete user accounts, use the **useradd**, **usermod**, and **userdel** commands, respectively. To create, modify, and delete group accounts, use the **groupadd**, **groupmod**, and **groupdel** commands, respectively. Alternatively, you can simply edit the file **/etc/passwd** directly to create, modify, or delete users, or edit **/etc/group** to create, modify, or delete groups.

Note that initial (primary) group memberships are set in each user's entry in **/etc/passwd**; supplementary (secondary) group memberships are set in **/etc/group**. (You can use the **usermod** command to change either primary or supplementary group memberships for any user.) To change your password, use the **passwd** command. If you're logged on as root, you can also use this command to change other users' passwords.

Password Aging Password aging (that is, maximum and minimum lifetime for user passwords) is set globally in the files **/etc/login.defs** and **/etc/default/useradd**, but these settings are only applied when new user accounts are created. To modify the password lifetime for an existing account, use the **chage** command.

As for the actual minimum and maximum password ages, passwords should have some minimum age to prevent users from rapidly "cycling through" password changes in attempts to reuse old passwords; seven days is a reasonable minimum password lifetime. Maximum lifetime is trickier: If this is too long, the odds of passwords being exposed before being changed will increase, but if it's too short, users frustrated with having to change their passwords frequently may feel justified in selecting easily guessed but also easily remembered passwords, writing passwords down, and otherwise mistreating their passwords in the name of convenience. Sixty days is a reasonable balance for many organizations.

In any event, it's much better to disable or delete defunct user accounts promptly and to educate users on protecting their passwords than it is to rely too much on password aging.

"Root Delegation:" su and sudo As we've seen, the fundamental problem with Linux and UNIX security is that far too often, permissions and authority on a given system boil down to "root can do anything, users can't do much of anything." Provided you know the root password, you can use the **su** command to promote yourself to root from whatever user you logged in as. Thus, the **su** command is as much a part of this problem as it is part of the solution.

Sadly, it's much easier to do a quick **su** to become root for a while than it is to create a granular system of group memberships and permissions that allows administrators and sub-administrators to have exactly the permissions they need. You can use the **su** command with the **"-c"** flag, which allows you to specify a single command to run as root rather than an entire shell session (for example, "**su -c rm somefile.txt**"), but because this requires you to enter the root password, everyone who needs to run a particular root command via this method will need to be given the root password. But it's never good for more than a small number of people to know root's password.

Another approach to solving the “root takes all” problem is to use SELinux’s Role-Based Access Controls (RBAC) (see Section 23.7), which enforce access controls that reduce root’s effective authority. However, this is much more complicated than setting up effective groups and group permissions. (However, adding that degree of complexity may be perfectly appropriate, depending on what’s at stake.)

A reasonable middle ground is to use the **sudo** command, which is a standard package on most Linux distributions. “sudo” is short for “superuser do”, and it allows users to execute specified commands as root without actually needing to know the root password (unlike **su**). **sudo** is configured via the file **/etc/sudoers**, but you shouldn’t edit this file directly; rather, you should use the command **visudo**, which opens a special vi (text editor) session.

As handy as it is, **sudo** is a very powerful tool, so use it wisely: Root privileges are never to be trifled with. It really is better to use user- and group permissions judiciously than to hand out root privileges even via **sudo**, and it’s better still to use an RBAC-based system like SELinux if feasible.

Logging

Logging isn’t a proactive control; even if you use an automated “log watcher” to parse logs in real time for security events, logs can only tell you about bad things that have already happened. But effective logging helps ensure that in the event of a system breach or failure, system administrators can more quickly and accurately identify what happened and thus most effectively focus their remediation and recovery efforts.

On Linux systems, system logs are handled either by the ubiquitous **Berkeley Syslog daemon** (**syslogd**) in conjunction with the **kernel log daemon** (**klogd**), or by the much-more-feature-rich **Syslog-NG**. System log daemons receive log data from a variety of sources (the kernel via **/proc/kmsg**, named pipes such as **/dev/log**, or the network), sort data by **facility** (category) and **severity**, and then write the log messages to log files (or to named pipes, the network, etc.). Figure 23.3 lists the facilities and severities, both in their mnemonic and numeric forms, of Linux logging facilities, plus **syslogd**’s actions (log targets).

Syslog-NG, the creation of Hungarian developer Balazs Scheidler, is preferable to **syslogd** for two reasons. First, it can use a much wider variety of log-data sources and destinations. Second, its “rules engine” (usually configured in **/etc/syslog-ng/syslog-ng.conf**) is much more flexible than **syslogd**’s simple configuration file (**/etc/syslogd.conf**), allowing you to create a much more sophisticated set of rules for evaluating and processing log data.

Naturally, both **syslogd** and **Syslog-NG** install with default settings for what gets logged, and where. While these default settings are adequate in many cases, you should never take for granted that they are. At the very least, you should decide what combination of local and remote logging to perform. If logs remain local to the system that generates them, they may be tampered with by an attacker. If some or all log data are transmitted over the network to some central log server, audit trails can be more effectively preserved, but log data may also be exposed to network eavesdroppers.

(The risk of eavesdropping is still another reason to use **Syslog-NG**; whereas **syslogd** only supports remote logging via the connectionless UDP protocol, **Syslog-NG** also supports logging via TCP, which can be encrypted via a TLS “wrapper” such as **Stunnel** or **Secure Shell**.)

Facilities	Facility Codes [†]	Priorities (in increasing order)	Priority Codes [†]	Actions
auth	4	none	n/a	/some/file (log to specified file)
auth-priv	10	debug	7	-some/file (log to spec'd file)
cron	9	info	6	but don't sync afterwards)
daemon	3	notice	5	/some/pipe (log to specified
kern	0	warning	4	pipe)
lpr	6	err	3	dec/some/tty_or_console
mail	2	crit	2	(log to specified console)
mark	n/a	alert	1	@remote.hostname.or.IP
news	7	emerg	0	(log to specified remote host)
syslog	5	* {"any	n/a	username1, username2, etc
user	1	facility"}		(log to these users' screens)
uucp	8			* (log to all users' screens)
local {0-7}	16-23	Usage of ! and = as prefixes with priorities		
* {"any facility"}	n/a	*.notice (no prefix)	=	"any event with priority of notice or higher"
		*.!notice	=	"no event with priority of notice or higher"
		*.=notice	=	"only events with priority of notice"
		*.! =notice	=	"only events with priority of notice"

[†]Numeric facility codes should not be used under Linux; they're here for reference only, as some other syslogd implementations (e.g., Cisco IOS) do use them.

Figure 23.3 Syslogd Reference

Local log files must be carefully managed. Logging messages from too many different log facilities to a single file may result in a logfile from which it is difficult to cull useful information; having too many different log files may make it difficult for administrators to remember where to look for a given audit trail. And in all cases, log files must not be allowed to fill disk volumes.

Most Linux distributions address this last problem via the **logrotate** command (typically run as a cron job), which decides how to rotate (archive or delete) system and application log files based both on global settings in the file `/etc/logrotate.conf` and on application-specific settings in the scripts contained in the directory `/etc/logrotate.d/`.

The Linux logging facility provides a local "system infrastructure" for both the kernel and applications, but it's usually also necessary to configure applications themselves to log appropriate levels of information. We revisit the subject of application-level logging in Section 23.6.

Other System Security Tools

Other tools worth mentioning that can greatly enhance Linux system security include the following:

- **Bastille:** A comprehensive system-hardening utility that educates as it secures
- **Tripwire:** A utility that maintains a database of characteristics of crucial system files and reports all changes made to them

- **Snort:** A powerful free Intrusion Detection System (IDS) that detects common network-based attacks
- **Nessus:** A modular security scanner that probes for common system and application vulnerabilities

23.6 APPLICATION SECURITY

Application security is a large topic; entire chapters in [BAUE05] are devoted to securing particular applications. However, many security features are implemented in similar ways across different applications. In this brief but important section, we'll examine some of these common features.

Running as an Unprivileged User/Group

Remember that in Linux and other UNIX-like operating systems, every process “runs as” some user. For network daemons in particular, it's extremely important that this user not be root; any process running as root is never more than a single buffer overflow or race condition away from being a means for attackers to achieve remote root compromise. Therefore, one of the most important security features a daemon can have is the ability to run as a nonprivileged user or group.

Running network processes as root isn't entirely avoidable; for example, only root can bind processes to “privileged ports” (TCP and UDP ports lower than 1024). However, it's still possible for a service's *parent* process to run as root in order to bind to a privileged port, but to then spawn a new child process that runs as an unprivileged user, each time an incoming connection is made.

Ideally, the unprivileged users and groups used by a given network daemon should be dedicated for that purpose, if for no other reason than for auditability (i.e., if entries start appearing in `/var/log/messages` indicating failed attempts by the user *ftpuser* to run the command `/sbin/halt`, it will be much easier to determine precisely what's going on if the *ftpuser* account isn't shared by five different network applications).

Running in a chroot Jail

If an FTP daemon serves files from a particular directory, say, `/srv/ftp/public`, there shouldn't be any reason for that daemon to have access to the rest of the file system. The **chroot** system call confines a process to some subset of `/`, that is, it maps a virtual “/” to some other directory (e.g., `/srv/ftp/public`). We call this directory to which we restrict the daemon a **chroot jail**. To the “chrooted” daemon, everything in the chroot jail appears to actually be in `/` (e.g., the “real” directory `/srv/ftp/public/etc/myconfigfile` appears as `/etc/myconfigfile` in the chroot jail). Things in directories outside the chroot jail (e.g., `/srv/www` or `/etc`.) aren't visible or reachable at all.

Chrooting therefore helps contain the effects of a given daemon's being compromised or hijacked. The main disadvantage of this method is added complexity: Certain files, directories, and special files typically must be copied into the chroot jail, and determining just what needs to go into the jail for the daemon to work properly can be tricky, though detailed procedures for chrooting many different Linux applications are easy to find on the World Wide Web.

Troubleshooting a chrooted application can also be difficult: Even if an application explicitly supports this feature, it may behave in unexpected ways when run chrooted. Note also that if the chrooted process runs as root, it can “break out” of the chroot jail with little difficulty. Still, the advantages usually far outweigh the disadvantages of chrooting network services.

Modularity

If an application runs in the form of a single, large, multipurpose process, it may be more difficult to run it as an unprivileged user; it may be harder to locate and fix security bugs in its source code (depending on how well documented and structured the code is); and it may be harder to disable unnecessary areas of functionality. In modern network service applications, therefore, **modularity** is a highly prized feature.

Postfix, for example, consists of a suite of daemons and commands, each dedicated to a different mail-transfer-related task. Only a couple of these processes ever run as root, and they practically never run all at the same time. Postfix therefore has a much smaller **attack surface** than the monolithic Sendmail. The popular Web server Apache used to be monolithic, but it now supports code modules that can be loaded at startup time as needed; this both reduces Apache’s memory footprint and reduces the threat posed by vulnerabilities in unused functionality areas.

Encryption

Sending logon credentials or application data over networks in clear text (i.e., unencrypted) exposes them to network eavesdropping attacks. Most Linux network applications therefore support encryption nowadays, most commonly via the OpenSSL library. Using application-level encryption is, in fact, the most effective way to ensure end-to-end encryption of network transactions.

The SSL and TLS protocols provided by OpenSSL require the use of **X.509 digital certificates**. These can be generated and signed by the user space **openssl** command. For optimal security, either a local or commercial (third-party) **Certificate Authority (CA)** should be used to sign all server certificates, but **self-signed** (that is, nonverifiable) certificates may also be used. [BAUE05] provides detailed instructions on how to create and use your own Certificate Authority with OpenSSL.

Logging

Most applications can be configured to log to whatever level of detail you want, ranging from “debugging” (maximum detail) to “none.” Some middle setting is usually the best choice, but you should not assume that the default setting is adequate.

In addition, many applications allow you to specify either a dedicated file to write application event data to, or a syslog **facility** to use when writing log data to **/dev/log** (see Section 23.5). If you wish to handle system logs in a consistent, centralized manner, it’s usually preferable for applications to send their log data to **/dev/log**. Note, however, that logrotate (also discussed in Section 23.5) can be configured to rotate *any* logs on the system, whether written by syslogd, Syslog-NG, or individual applications.

23.7 MANDATORY ACCESS CONTROLS

Linux (like most other general-purpose operating systems) uses a DAC security model, in which the owner of a given system object can set whatever access permissions on that resource he or she likes. Stringent security controls, in general, are optional.

In contrast, a computer with Mandatory Access Controls (MAC) has a global security policy that all users of the system are subject to. A user who creates a file on a MAC system generally may not set access controls on that file that are weaker than the controls dictated by the system security policy.

Compromising a system using a DAC-based security model is generally a simple matter of hijacking some process on that system that runs with root/Administrator privileges. On a MAC-based system, however, the only thing the superuser account is used for is maintaining the global security policy. Day-to-day system administration is performed using accounts that lack the authority to change the global security policy. As a result, it's impossible to compromise the entire system by attacking any one process. (Attacks on the policy-setting account are still possible, however; for example, by booting the system into single-user mode from its physical console.)

Unfortunately, while MAC schemes have been available on various platforms over the years, they have traditionally been much more complicated to configure and maintain than DAC-based operating systems. To create an effective global security policy requires detailed knowledge of the precise (intended) behavior of every application on the system. Furthermore, the more restrictive the security controls are on a given system, the less convenient that system becomes for its users to use.

Linux packagers Novell and Red Hat have addressed MAC complexity in similar ways. Novell's SuSE Linux includes AppArmor, a partial MAC implementation that restricts specific processes but leaves everything else subject to the conventional Linux DAC. In Fedora and Red Hat Enterprise Linux, SELinux has been implemented with a policy that, like AppArmor, restricts key network daemons, but relies on the Linux DAC to secure everything else.

What about high-sensitivity, high-security, multiuser scenarios? In those cases a "pure" SELinux implementation may be deployed, in which *all* processes, system resources, and data are regulated by comprehensive, granular access controls.

Let's take a closer look at SELinux and Novell AppArmor.

SELinux²

SELinux is the NSA's powerful implementation of mandatory access controls for Linux. This power, however, comes at a cost: It is a complicated technology and can be time-consuming to configure and troubleshoot. In this section, we'll discuss SELinux concepts and security models, ending with some pointers to more detailed information on managing SELinux.

²Much of this section is adapted from [BAUE07], with permission of the *Linux Journal*.

The Problem As noted earlier, Linux security often seems to boil down to a cycle of researchers and attackers discovering new security vulnerabilities in Linux applications and kernels; vendors and developers scrambling to release patches, with attackers wreaking havoc against unpatched systems in the mean time; and hapless system administrators finally applying that week's or month's patches, only to repeat the entire trail of tears soon afterward. Unfortunately, there will always be zero-day (as-yet-unpatched) vulnerabilities. SELinux is a mandatory access control implementation that doesn't prevent zero-day attacks, but it's specifically designed to contain their effects.

For example, suppose I have a daemon called `blinkend` that is running as the user `someguy`, and this daemon is hijacked by an attacker. `blinkend`'s sole function is to make a keyboard LED blink out jokes in Morse code, so you might think, well, the worst the attacker can do is blink some sort of insult, right? Wrong. The attacker can do anything the `someguy` account can do, which might include everything from executing the BASH shell to mounting CD-ROMs.

Under SELinux, however, the `blinkend` process would run in a narrowly defined domain of activity that would allow it to do its job (blinking the LED, possibly reading jokes from a particular text file, etc.). In other words, `blinkend`'s privileges would not be determined based on its user/owner; rather, they would be determined by much more narrow criteria. Provided that `blinkend`'s domain was sufficiently strictly defined, even a successful attack against the `blinkend` process would, at worst, result in naughty Morse code blinking.

That, in a nutshell, is the problem SELinux was designed to solve.

What SELinux Does By now you should understand how Linux's Discretionary Access Controls work. Even under SELinux, the Linux DACs still apply: If the ordinary Linux permissions on a given file block a particular action (for example, user A attempting to write file B), that action will still be blocked, and SELinux won't bother evaluating that action. But if the ordinary Linux permissions allow the action, SELinux will evaluate the action against its own security policies before allowing it to occur.

So how does SELinux do this? The starting point for SELinux seems similar to the DAC paradigm: It evaluates actions attempted by **subjects** against **objects**.

In SELinux, "subjects" are always processes. This may seem counterintuitive: Aren't subjects sometimes end users? Not exactly; users execute commands (processes). SELinux naturally pays close attention to who or what executes a given process, but the process itself, not the human being who executed it, is considered to be the subject.

In SELinux, we call actions "permissions," just like we do in the Linux DAC. The objects that are acted on, however, are different. Whereas in the Linux DAC model objects are always files or directories, in SELinux objects include not only files and directories but also other processes and various system resources in both kernel space and userland.

SELinux differentiates among a wide variety of object "classes" (categories)—dozens, in fact. You can read the complete list on the Web site "An Overview of Object Classes and Permissions," whose URL is in the Resources section. Not

surprisingly, “file” is the most commonly used object class. Other important object classes include

- dir
- socket
- tcp_socket
- unix_stream_socket
- file system
- node
- xserver
- cursor

Each object class has a particular set of possible permissions (actions). This makes sense; there are things you can do to directories, for example, that simply don’t apply to, say, X Servers. Each object class may have both “inherited” permissions that are common to other classes (for example, “read”), plus “unique” permissions that apply only to it. Just a few of the unique permissions associated with the “dir” class are

- search
- rmdir
- getattr
- remove_name
- reparent

These class names or actions are not explained here because you don’t need to understand them for their own sake; it is sufficient to know that SELinux goes much, much further than Linux DAC’s simple model of users, groups, files, directories, and read/write/execute permissions.

As you might guess, SELinux would be impossible to use if you had to create an individual rule for every possible action by every possible subject against every possible object. SELinux gets around this in two ways: (1) by taking the stance “that which is not expressly permitted is denied,” and (2) by grouping subjects, permissions, and objects in various ways. Both of these points have positive and negative ramifications.

The “default deny” stance allows you to only have to create rules/policies that describe the behaviors you expect and want, instead of all possible behaviors. It’s also, by far, the most secure design principle any access control technology can have. However, it also requires you to anticipate all possible allowable behavior by (and interaction between) every daemon and command on your system.

(This is why the “targeted” SELinux policy in Red Hat Enterprise Linux 4 and Fedora Core 3 actually implements what amounts to a “restrict only these particular services” policy, giving free rein to all processes not explicitly covered in the policy. No, this is not the most secure way to use SELinux, or even the way SELinux was originally designed to be used. But as we’ll see, it’s a justifiable compromise on general-purpose systems.)

The upside of SELinux’s various groupings (roles, types/domains, contexts, etc.) is, obviously, improved efficiency over having to always specify individual subjects, permissions, and objects. The downside is still more terminology and layers of abstraction.

Security Contexts: Users, Roles, and Domains Every individual subject and object controlled by SELinux is governed by a **security context**, each consisting of a **user**, a **role**, and a **domain** (also called a **type**).

A user is what you’d expect: an individual user, whether human or daemon. However, SELinux maintains its own list of users, separately from the Linux DAC system. In security contexts for subjects, the user label indicates which SELinux user account’s privileges the subject (which, again, must be a process) is running. In security contexts for objects, the user label indicates which SELinux user account owns the object.

A role is sort of like a group in the Linux DAC system, in that a role may be assumed by any of a number of preauthorized users, each of whom may be authorized to assume different roles at different times. The difference is that in SELinux, a user may only assume one role at a time and may only switch roles if and when authorized to do so. The role specified in a security context indicates which role the specified user is operating within for that particular context.

Finally, a domain is sort of like a sandbox: a combination of subjects and objects that may interact with each other. Domains are also called types, and although domains and types are two different things in the **Flask** security model on which the NSA based SELinux, in SELinux “domain” and “type” are synonymous.

This model, in which each process (subject) is assigned to a domain, wherein only certain operations are permitted, is called **Type Enforcement (TE)**, and it’s the heart of SELinux. Type Enforcement also constitutes the bulk of the SELinux implementation in Fedora and Red Hat Enterprise Linux.

There’s a bit more to it than that, but before we go into further depth, we present an example scenario to illustrate security contexts.

Suppose we’re securing my LED-blinking daemon, `blinkend`, with SELinux. As you’ll recall, it’s run with the privileges of the account “`someguy`,” and it reads the messages it blinks from a text file, which we’ll call `/home/someguy/messages.txt`.

Under SELinux, we’ll need an SELinux user called “`someguy`” (remember, this is in addition to the underlying Linux DAC’s “`someguy`” account, that is, the one in `/etc/passwd`). We’ll also need a role for `someguy` to assume in this context; we could call it “`blink_r`” (by convention, SELinux role names end with “`_r`”).

The heart of `blinkend`’s security context will be its domain, which we’ll call “`blinkend_t`” (by convention, SELinux domain names end with “`_t`” — “`t`” is short for “type”). `blinkend_t` will specify rules that allow the `blinkend` process to read the file `/home/someguy/messages.txt` and then write data to, say, `/dev/numlockled`.

The file `/home/someguy/messages.txt` and the special file `/dev/numlockled` will need security contexts of their own. Both of these contexts can probably use the `blinkend_t` domain, but because they describe objects, not subjects, they’ll specify the catch-all role “`object_r`.” Objects, which by definition are passive in nature (stuff gets done to them, not the other way around), generally don’t assume meaningful roles, but every security context must include a role.

Decision Making in SELinux There are two types of decisions SELinux must make concerning subjects, domains, and objects: **access** decisions and **transition** decisions. Access decisions involve subjects doing things to objects that already exist, or creating new things that remain in the expected domain. Access decisions are easy to understand; in our example, “may blinkend read /home/someguy/messages.txt?” is just such a decision.

Transition decisions, however, are a bit more subtle. They involve the invocation of processes in different domains than the one in which the subject process is running; or the creation of objects in different types than their parent directories. (Note: Even though “domain” and “type” are synonymous in SELinux, by convention we usually use “domain” when talking about processes, and “type” with files.)

That is, normally, if one process executes another, the second process will by default run within the same SELinux domain. If, for example, blinkend spawns a child process, the child process will run in the blinkend_t domain, the same as its parent. If, however, blinkend tries to spawn a process into some other domain, SELinux will need to make a domain transition decision to determine whether to allow this. Like everything else, transitions must be explicitly authorized in the SELinux policy. This is an important check against privilege-escalation attacks.

File transitions work in a similar way: If a subject creates a file in some directory (and if this file creation is allowed in the subject’s domain), the new file will normally inherit the security context (user, role, and domain) of the parent directory. For example, if blinkend’s security context allows it to write a new file in /home/someguy/, say, /home/someguy/error.log, then error.log will inherit the security context (user, role, and type) of /home/someguy/. If, for some reason, blinkend tries to label error.log with a different security context, SELinux will need to make a type transition decision.

Transition decisions are necessary because the same file or resource may be used in multiple domains/types; process and file transitions are a normal part of system operation. But if domains can be changed arbitrarily, attackers will have a much easier time doing mischief.

Role-Based Access Control Besides Type Enforcement, SELinux includes a second model, called **Role-Based Access Control** (RBAC). RBAC builds on the concepts we’ve already discussed, providing controls especially useful where real human users, as opposed to daemons and other automated processes, are concerned.

RBAC is relatively straightforward. To paraphrase [MCCA05], SELinux rules specify what **roles** each user may assume; other rules specify under what circumstances each user may **transition** from one authorized role to another (unlike groups in the Linux DAC, in RBAC one user may not assume more than one role at a time); and still other rules specify the **domains** each authorized role may operate in.

Multilevel Security The third security model implemented in SELinux is **Multilevel Security** (MLS), which is based on the **Bell-LaPadula** (BLP) model. Chapter 10 describes the BLP model in detail. In SELinux, MLS is enforced via file system labeling.

Managing SELinux Policies Unfortunately, creating and maintaining SELinux policies is complicated and time-consuming; a single SELinux policy may consist of hundreds of lines of text. In Red Hat and Fedora, this complexity is mitigated by the

inclusion of a default “targeted” policy that defines types for selected network applications but that allows everything else to run with only Linux DAC controls. You can use RHEL and Fedora’s **system-config-securitylevel** GUI to configure the targeted policy.

SELinux policies take the form of various, lengthy text files in **/etc/security/selinux**. SELinux commands common to all SELinux implementations (besides RHEL and Fedora) are **chcon**, **checkpolicy**, **getenforce**, **newrole**, **run_init**, **setenforce**, and **setfiles**. Tresys (<http://www.tresys.com>), however, maintains a suite of free, mainly GUI-based, SELinux tools that are a bit easier to use, including **SePCuT**, **SeUser**, **Apol**, and **SeAudit**.

For more information on using RHEL’s SELinux implementation, see [COKE05]. See [MCCA05] for more information on creating and maintaining custom SELinux policies.

Novell AppArmor³

AppArmor, Novell’s MAC implementation for SuSE, represents a major step forward in making MAC technology a feasible option for system administrators who want strong security controls but don’t have the time or patience to configure and maintain SELinux. As of this writing, AppArmor is only available for SuSE Linux and SuSE Linux Enterprise. AppArmor, like SELinux, is built on top of the Linux Security Modules.

As we’ve seen, SELinux implements three different types of MAC: Type Enforcement, Role-Based Access Controls, and Multi Level Security. In contrast, Novell AppArmor has a more modest objective: to restrict the behavior of selected applications in a very granular but targeted way. In focusing on applications (at the expense of roles and data classification), AppArmor is built on the assumption that the single biggest attack vector on most systems is application vulnerabilities. If the application’s behavior is restricted, then the behavior of any attacker who succeeds in exploiting some vulnerability in that application will also be restricted.

For example, suppose you’re running a Web application that runs as user “nobody” and uses user input to update a local text file. On a typical system, if an attacker compromised that Web application (for example, by sending unexpected input), the attacker might succeed in gaining a remote shell with the privileges of “nobody.” If that Web application were protected by AppArmor, however, all the attacker would be able to do would be to alter that single text file; it would neither be possible for the attacker to spawn a remote shell (an unexpected action) nor to read or write any other files.

Comprehensive? By no means: for non-AppArmor-protected applications, the usual (limited) user/group permissions still apply; normally, only a subset of applications on the system even have AppArmor profiles; and AppArmor provides no controls addressing data classification. To use SELinux terminology, AppArmor provides only nonglobal Type Enforcement, no Role-Based Access Controls, and no Multi Level Security.

³Much of this section is adapted from [BAUE06], with permission of the *Linux Journal*.

For the most part, root is still root, and if you use root access in a sloppy or risky fashion, AppArmor generally won't protect you from yourself. But if an AppArmor-protected application runs as root and becomes compromised somehow, that application's access will be contained, root privileges notwithstanding, because those privileges are trumped by the AppArmor policy (which is enforced at the kernel level, courtesy of Linux Security Modules).

AppArmor is therefore only a partial implementation of Mandatory Access Controls. But on networked systems, application security is arguably the single most important area of concern, and that's what AppArmor zeroes in on. What's more, AppArmor provides application security via an easy to use graphical user interface that is fully integrated with SuSE's system administration tool, YaST.

We are stopping well short of suggesting that AppArmor is interchangeable with SELinux. If, for example, you run Linux in a true multiuser environment (in which users have shell accounts) or use a Linux system to process highly sensitive data, there really is no substitute for the comprehensive layers of access controls in SELinux.

23.8 RECOMMENDED READING AND WEB SITE

[NEME06], while not being focused exclusively on security, contains one of the best general treatments of Linux and UNIX security. [TOXE02], while not as up to date, is focused exclusively on security. For step-by-step procedures on securing Linux-based Internet servers, see [BAUE05]. [SUEH05] is a good treatment of Linux firewalls. For coverage of SELinux, see [MCCA05] and [MAYE07].

BAUE05 Bauer, M. *Linux Server Security, Second Edition*. Sebastopol, CA: O'Reilly Media, 2005.

MAYE07 Mayer, F., MacMillan, K., and Caplan, D. *SELinux by Example: Using Security Enhanced Linux*. Upper Saddle River, NJ: Prentice Hall, 2007.

MCCA05 McCarty, B. *SELinux: NSA's Open Source Security Enhanced Linux*. Sebastopol, CA: O'Reilly Media, 2005.

NEME06 Nemeth, E., Snyder, G., and Hein, T. *Linux Administration Handbook*, Upper Saddle River, NJ: Prentice Hall, 2006.

SUEH05 Suehring, S., and Ziegler, R. *Linux Firewalls*, Upper Saddle River, NJ: Novell Press, 2005.

TOXE02 Toxen, B. *Real World Linux Security*, Upper Saddle River, NJ: Prentice Hall, 2002.



Recommended Web site:

- **NSA SELinux Web site:** Contains useful documentation on SELinux

23.9 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

attack vector Bell-LaPadula model buffer overflow chroot chroot jail denial of service (DoS) discretionary access control (DAC) kernel space loadable kernel modules (LKM)	mandatory access control (MAC) multilevel Security (MLS) object permissions race condition role-based access control (RBAC) root run as setuid bit setgid bit	root shell rootkit shellcode sticky bit subject superuser threat type enforcement user space vulnerability zero-day (0-day)
--	--	---

Review Questions

- 23.1 What type of access control model is Linux (and UNIX) based on?
- 23.2 What are the two different types of subject in this model?
- 23.3 What are the three different types of object in this model?
- 23.4 What are the two different types of owner in Linux file-system security?
- 23.5 What do the permissions “read,” “write,” and “execute” mean when applied to directories?
- 23.6 On Linux systems, what does the “sticky bit” do?
- 23.7 What effect does “setuid” have on files? On directories?
- 23.8 What effect does “setgid” have on directories? On files?
- 23.9 What integer represents the numeric code for the permission “read-write-execute?”
- 23.10 What is kernel space?
- 23.11 What are the components of a threat?
- 23.12 What is a rootkit? Why are they hard to detect?
- 23.13 What is the difference between r-services and RPC services? Why is their security problematic?
- 23.14 A software vulnerability for which there is no patch is called what?
- 23.15 What are the pros and cons of automated patching?
- 23.16 Which is a more flexible network access control mechanism, libwrappers or iptables? Why?
- 23.17 What threat does egress filtering mitigate?
- 23.18 On Linux systems, what is antivirus software most commonly used for?
- 23.19 What is the difference between su and sudo? Which is better for allowing more than one or two users to execute root-privileged commands?
- 23.20 What are logs useful for? What don’t they do?
- 23.21 What is a chroot jail? With what other common application security feature must it be combined to be effective?
- 23.22 Which open-source encryption library is the most common means of encrypting network application traffic?

- 23.23 What are the three access control models implemented in SELinux? Which is the most important in Red Hat and Fedora's SELinux implementation?
- 23.24 Under RBAC, how many roles may a user assume at one time?
- 23.25 Explain the principle "no read up, no write down." Which SELinux model does this describe?
- 23.26 Which Linux distribution includes Novell AppArmor?

Problems

- 23.1 What is dangerous about a process running as root? Your answer should describe more than one threat scenario.
- 23.2 Why are file system permissions so important in the Linux DAC model? How do they relate or map to the concept of "subject-action-object" transactions?
- 23.3 User "ahmed" owns a directory containing a file called "ourstuff" that he shares with users belonging to the group "staff." Those users may read and change this file but not delete it. They may not add other files to the directory. What would appropriate ownerships and permissions for both the directory "ourstuff" and the file "staff" look like? (Write your answers in the form of "long listing" output.)
- 23.4 Write the numeric mode for a file whose permissions-string is **-rwxr-x---**.
- 23.5 Suppose you operate an Apache-based Linux Web server that hosts your company's e-commerce site. Suppose further that there is a worm called "WorminatorX," which exploits a (fictional) buffer overflow bug in the Apache Web server package that can result in a remote root compromise. Construct a simple threat model that describes the risk this represents: attacker(s), attack vector, vulnerability, assets, likelihood of occurrence, likely impact, and plausible mitigations.
- 23.6 Why is logging important? What are its limitations as a security control? What are pros and cons of remote logging?
- 23.7 The command **ping**, which sends a simple "are you there?" query to other networked systems, usually runs setuid root. What is risky about this, and how could SELinux or Novell AppArmor mitigate those risks?
- 23.8 How does the "subject-action-object" concept map to SELinux's Type Enforcement model? How is this different from the default Linux DAC?
- 23.9 How are Novell AppArmor and the Red Hat "targeted" SELinux policy similar? Is either a true Mandatory Access Control implementation. If not, explain why.