CHAPTER 14
Setting Ownership and Permissions

As a multiuser OS, Linux provides tools to help you secure your files against unwanted access—after all, you wouldn't want another user to read your personal files or even delete your work files, whether accidentally or intentionally. Linux handles these tasks through two features of files and directories: their *ownership* and their *permissions*. Every file has an associated owner (an account with which it's linked) and also an associated group. Three sets of permissions define what the file's owner, members of the file's group, and all other users can do with the file. Thus ownership and permissions are intertwined, although you use different text-mode commands to manipulate them. (GUI tools often combine the two, as described in this chapter.)

- **Setting ownership**
- **Setting permissions**

Setting Ownership

Linux's security model is based on that of Unix, which was designed as a multiuser OS. This security model therefore assumes the presence of multiple users on the computer, and it provides the means to associate individual files with the users who create them—that is, files have *owners*. You should thoroughly understand this concept, and with that understanding, you can change a file's ownership, using either a GUI file manager or a text-mode shell.

The *set user ID (SUID)* and *set group ID (SGID)* permission bits, described later in "Using Special Execute Permissions," can modify the account and group associated with a program.

Ownership also applies to running programs (or *processes*). Most programs that you run are tied to the account that you used to launch them. This identity, in conjunction with the file's ownership and permissions, determines whether a program may modify a file.

Understanding Ownership

Chapter 12, "Understanding Basic Security," and Chapter 13, "Creating Users and Groups," described Linux's system of accounts. These accounts are the basis of file ownership. Specifically, every file has an owner—an account with which it's associated. This association occurs by means of the account's *user ID (UID)* number. Every file is also associated with a *group* by means of a *group ID (GID)* number.

As described later, in the section "Setting Permissions," access to the file is controlled by

Bresnahan, Christine, and Blum, Richard. Linux Essentials (2). Hoboken, US: Sybex, 2015. ProQuest ebrary. Web. 17 January 2017.

means of permissions that can be set independently for the file's owner, the file's group, and all other users of the computer. As `root`, you can change the owner and group of any file. The file's owner can also change the file's group, but only to a group to which the user belongs.

The same principles of ownership apply to directories as apply to files: directories have owners and groups. These can be changed by `root` or, to a more limited extent, by the directory's owner.

Cross-Installation UIDs and GIDs

You may use multiple Linux installations, either dual-booting on one computer or installed on multiple computers. If you do, and if you transfer files from one installation to another, you may find that the ownership of files seems to change as you move them around. The same thing can happen with non-Linux Unix-like OSs, such as Mac OS X. The reason is that the filesystems for these OSs store ownership and group information by using UID and GID numbers, and a single user or group can have different UID or GID numbers on different computers, even if the name associated with the account or group is identical.

This problem is most likely to occur when using native Linux or Unix filesystems to transfer data, including both disk-based filesystems (such as Linux's ext2fs or Mac OS X's HFS+) or the Network File System (NFS) for remote file access. This problem is less likely to occur if you use a non-Linux/Unix filesystem, such as the File Allocation Table (FAT) or the New Technology File System (NTFS) for disks, or the Server Message Block/Common Internet File System (SMB/CIFS—handled by Samba in Linux) for network access.

If you run into this problem, several solutions exist, but many of them are beyond the scope of this book. One that you can use, though, is to change the UID or GID mappings on one or more installations so that they all match. Chapter 13 describes how to change a user's UID number with `usermod` and how to change a group's GID number with `groupmod`. When transferring data via removable disks, using FAT or NTFS can be a simple solution, provided that you don't need to preserve Unix-style permissions on the files.

Setting Ownership in a File Manager

As described in Chapter 4, "Using Common Linux Programs," a *file manager* enables you to manipulate files. You're probably familiar with file managers in Windows or Mac OS X. Linux's ownership and permissions are different from those of Windows, though, so you may want to know how to check on, and perhaps change, ownership features by using a Linux file manager. As noted in Chapter 4, you have a choice of several file managers in Linux. Most are similar in broad strokes but differ in some details. In this section, we use Nautilus, the default file manager used in the GNOME desktop, as an example.

If you want to change the file's owner, you must run Nautilus as `root`, but you can change the file's group to any group to which you belong as an ordinary user. The procedure to perform this task as `root` is as follows:

1. Launch a terminal window.
2. In the terminal window, type **`su`** to acquire `root` privileges.

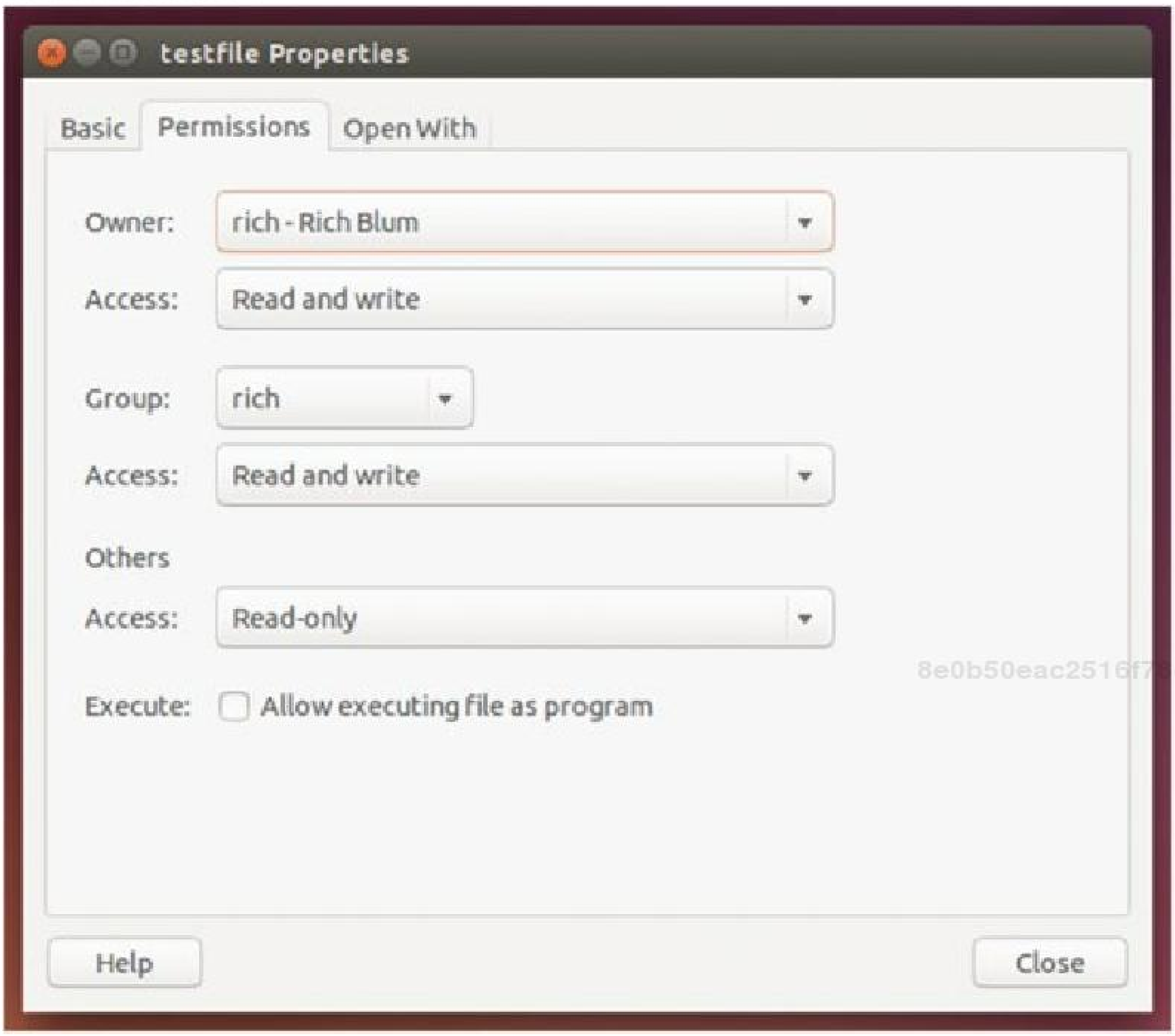
> **If you're using Ubuntu, you may instead need to use `sudo` to launch Nautilus.**

3. In the terminal window, type **`nautilus`** to launch Nautilus. You can optionally include the path to the directory in which you want Nautilus to start up. If you don't include a path, Nautilus will begin by displaying the contents of the `/root` directory.

> **The `/root` directory is the `root` account's home directory.**

4. Locate the file whose ownership you want to adjust and right-click it.
5. In the resulting menu, select Properties. The result is a Properties dialog box.
6. Click the Permissions tab in the Properties dialog box. The result resembles Figure 14.1.
7. To change the file's owner, select a new owner in the Owner field. This action is possible only if you run Nautilus as `root`.
8. To change the file's group, select a new group in the Group field. If you run Nautilus as an ordinary user, you will be able to select any group to which you belong, but if you run Nautilus as `root`, you will be able to select any group.
9. When you've adjusted the features that you want to change, click Close.

Bresnahan, Christine, and Blum, Richard. Linux Essentials (2). Hoboken, US: Sybex, 2015. ProQuest ebrary. Web. 17 January 2017.



Owner: rich - Rich Blum

Access: Read and write

Group: rich

Access: Read and write

Others

Access: Read-only

Execute: ☐ Allow executing file as program

Figure 14.1 Linux file managers give you access to the file's ownership and permission metadata.

If you want to change a file's group but not its owner, and if you're a member of the target group, you can launch Nautilus as an ordinary user. You can then pick up the preceding procedure at step 4.

You should be *extremely* cautious about running Nautilus as `root`. If you forget that you're running this program as `root`, you can easily create new files as `root`, which will require additional `root`-privilege actions to correct by changing file ownership. It's also easy to delete critical system files accidentally as `root`, which you could not delete as an ordinary user. For these reasons, we recommend that you use a text-mode shell to adjust file ownership. The change in the prompt makes it easier to notice that you're running as `root`, and if you're used to using a GUI, you're less likely to launch additional programs as `root` from a text-mode shell than from Nautilus.

Setting Ownership in a Shell

Certification Objective

The command to change the ownership of a file in the preferred text-mode manner is `chown`. In its most basic form, you pass it the name of a file followed by a username:

Bresnahan, Christine, and Blum, Richard. Linux Essentials (2). Hoboken, US: Sybex, 2015. ProQuest ebrary. Web. 17 January 2017.

The `chown` command's name stands for *change owner*.

```
# chown bob targetfile.odf
```

This example gives ownership of `targetfile.odf` to `bob`. You can change the file's principal owner and its group with a single command by separating the owner and group with a colon (`:`):

```
# chown bob:users targetfile.odf
```

This example gives ownership of `targetfile.odf` to `bob` and associates the file with the `users` group. To change the group without changing the owner, you can omit the owner, leaving the colon and group name:

```
$ chown:users targetfile.odf
```

Certification Objective

Alternatively, you can use the `chgrp` command, which works in the same way but changes *only* the group and does not require the colon before the group name:

```
$ chgrp users targetfile.odf
```

Note that the commands used to change the owner require `root` privileges, whereas you can change the group as an ordinary user—but only if you own the file and belong to the target group.

The `chown` and `chgrp` commands both support a number of options that modify what they do. The most useful of these is `-R` (or`--recursive`), which causes a change in ownership of all the files in an entire directory tree. For instance, suppose that the user `mary` has left a company, and an existing employee, `bob`, must access her files. If `mary`'s home directory was `/home/mary`, you might type this:

```
# chown -R bob /home/mary
```

This command gives `bob` ownership of the `/home/mary` directory, all the files in the `/home/mary` directory, including all its subdirectories, the files in the subdirectories, and so on. To make the transition a bit easier for `bob`, you might also want to move `mary`'s former home directory into `bob`'s home directory.

Setting Permissions

File ownership is meaningless without some way to specify what particular users can do with their own or other users' files. That's where permissions enter the picture. Linux's permission structure is modeled after that of Unix, and it requires a bit of explanation before you tackle the issue. Once you understand the basics, you can begin modifying permissions, using either a

GUI file manager or a text-mode shell. You can also set default permissions for new files that you create.

Understanding Permissions

Certification Objective

To understand Unix (and hence Linux) permissions, you may want to begin with the display created by the `ls` command, which lists the files in a directory, in conjunction with its `-l` option, which creates a long directory listing that includes files' permissions. For instance, to see a long listing of the file `test`, you might type the following:

> **Chapter 6, "Getting to Know the Command Line," introduced the `ls` command and describes additional `ls` options.**

```
$ ls -l test
-rwxr-xr-x  1 rich users      111 Apr 13 13:48  test
```

This line consists of several sections, which provide assorted pieces of information on the file:

Permissions The first column (`-rwxr-xr-x` in this example) is the file's permissions, which are of interest at the moment.

Number of Links The next column (`1` in this example) shows the number of hard links to the file—that is, the number of unique filenames that may be used to access the file.

> **Chapter 7, "Managing Files," describes links in more detail.**

Username The next column (`rich` in this example) identifies the file's owner by username.

Group Name The file's group (`users` in this example) appears next.

File Size This example file's size is quite small—111 bytes.

Time Stamp The time stamp (`Apr 13 13:48` in this example) identifies the time the file was last modified.

Filename Finally, `ls -l` shows the file's name—`test` in this example.

The string that begins this output (`-rwxr-xr-x` in this example) is a symbolic representation of the permissions string. Figure 14.2 shows how this string is broken into four parts.

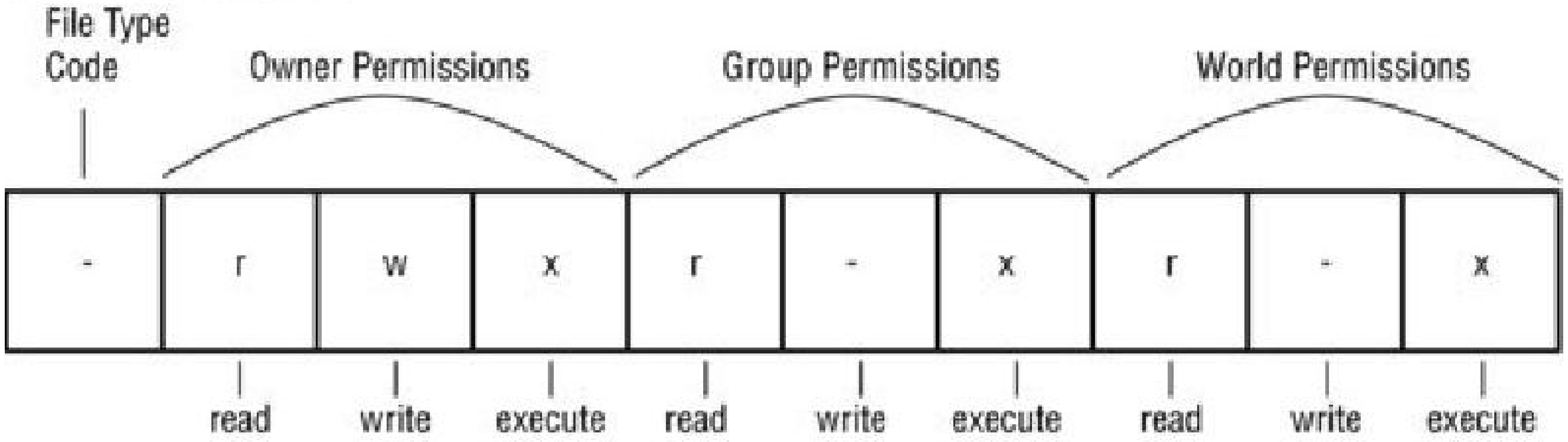


Figure 14.2 A symbolic representation of file permissions is broken into four parts.

File Type Code The first character is the file type code, which represents the file's type, as summarized in Table 14.1. This type character is sometimes omitted from descriptions when the file type is not relevant or when it's identified in some other way.

Table 14.1 Linux file type codes

Code	Name	Meaning
-	Normal data file	May be text, an executable program, graphics, compressed data, or just about any other type of data.
d	Directory	Disk directories are files, but they contain filenames and pointers to those named files' data structures.
`l`	Symbolic link	The file contains the name of another file or directory. When Linux accesses the symbolic link, it tries to read the linked-to file.
p	Named pipe	A pipe enables two running Linux programs to communicate with each other in a one-way fashion.
s	Socket	A socket is similar to a named pipe, but it permits network and bidirectional links.
b	Block device	A file that corresponds to a hardware device to and from which data is transferred in blocks of more than 1 byte. Disk devices (hard disks, USB flash drives, CD-ROMs, and so on) are common block devices.
c	Character device	A file that corresponds to a hardware device to and from which data is transferred in units of 1 byte. Examples include parallel and RS-232 serial port devices.

Most of the files that you'll manipulate are normal files, directories, and symbolic links.

Owner Permissions These permissions determine what the file's owner can do with the file.

Group Permissions These permissions determine what members of the file's group (who aren't its owner) can do with the file.

World (or "Other") Permissions These permissions determine what users who aren't the

Bresnahan, Christine, and Blum, Richard. Linux Essentials (2). Hoboken, US: Sybex, 2015. ProQuest ebrary. Web. 17 January 2017.

file's owner or members of its group can do with the file.

In each of the three sets of permissions, the string identifies the presence or absence of each of three types of access: read, write, and execute. Read and write permissions are fairly self-explanatory. If the execute permission is present, it means that the file may be run as a program. The absence of the permission is denoted by a dash (`-`) in the permission string. The presence of the permission is indicated by the letter `r` for read, `w` for write, or `x` for execute.

Setting the execute bit on a nonprogram file doesn't turn it into a program, of course; it just indicates that a user may run a file that is a program.

Thus the example permission string `-rwxr-xr-x` means that the file is a normal data file and that its owner, members of the file's group, and all other users can read and execute the file. Only the file's owner has write permission to the file.

Another representation of permissions is possible; it's compact but a bit confusing. It takes each of the three permissions groupings of the permission string (omitting the file type code) and converts it into a number from 0 to 7 (that is, a *base 8* or *octal* number). The result is a three-digit octal number. Each number is constructed by starting with a value of 0 and then:

- Adding 4 if read permissions are present
- Adding 2 if write permissions are present
- Adding 1 if execute permissions are present

These procedures involve binary numbers and logical, not arithmetic, operations. The arithmetic description is easier to understand, though.

The resulting three-digit code represents permissions for the owner, the group, and the world. Table 14.2 shows some examples of common permissions and their meanings.

Table 14.2 Example permissions and their interpretations

Permission string	Octal code	Meaning
`rwxrwxrwx`	777	Read, write, and execute permissions for all users.
`rwxr-xr-x`	755	Read and execute permission for all users. The file’s owner also has write permission.
`rwxr-x——`	750	Read and execute permission for the owner and group. The file’s owner also has write permission. Other users have no access to the file.
`rwx------`	700	Read, write, and execute permissions for the file’s owner only; all others have no access.
`rw-rw-rw-`	666	Read and write permissions for all users. No execute permissions for anybody.
`rw-rw-r--`	664	Read and write permissions for the owner and group. Read-only permission for all others.
`rw-rw----`	660	Read and write permissions for the owner and group. No world permissions.
`rw-r—r—`	644	Read and write permissions for the owner. Read-only permission for all others.
`rw-r-----`	640	Read and write permissions for the owner, and read-only permission for the group. No permission for others.
`rw-------`	600	Read and write permissions for the owner. No permission for anybody else.
`r--------`	400	Read permission for the owner. No permission for anybody else.

There are 512 possible combinations of permissions, so Table 14.2 is incomplete. It shows the most common and useful combinations, though.

Several special cases apply to permissions:

Directory Execute Bits Directories use the execute bit to grant permission to search the directory. This is a highly desirable characteristic for directories, so you’ll almost always find the execute bit set when the read bit is set.

Directory Write Permissions Directories are files that are interpreted in a special way. As such, if a user can write to a directory, that user can create, delete, or rename files in the directory, even if the user isn’t the owner of those files and does not have permission to write to those files.

Bresnahan, Christine, and Blum, Richard. Linux Essentials (2). Hoboken, US: Sybex, 2015. ProQuest ebrary. Web. 17 January 2017.

The usual rules for writing to directories can be modified with the *sticky bit*, which is described later in "Using Sticky Bits."

Symbolic Links Permissions on symbolic links are always 777 (`rwxrwxrwx`, or `lrwxrwxrwx` to include the file type code). This access applies only to the link file itself, not to the linked-to file. In other words, all users can read the contents of the link to discover the name of the file to which it points, but the permissions on the linked-to file determine its file access. Changing the permissions on a symbolic link affects the linked-to file.

`root` Many of the permission rules don't apply to `root`. The superuser can read or write any file on the computer—even files that grant access to nobody (that is, those that have 000 permissions). The superuser still needs an execute bit set to run a program file.

Setting Permissions in a File Manager

The procedure for setting permissions in a file manager is similar to that for setting the ownership of a file:

- You normally adjust these settings by using the same dialog box used to adjust ownership, such as the Nautilus dialog box shown earlier in Figure 14.1.

Details vary in other file managers, but the principles are the same as those described here.

- You don't need to be `root` to adjust the permissions of files that you own.
- You should use `root` access for this job only on files that you don't own.

As seen in Figure 14.1, there are three Access items associated with the Owner, the Group, and Others:

- The Owner item provides two options: Read-Only and Read and Write.
- The Group and Others items both provide Read-Only and Read and Write plus the None option. You can use these options to set the read and write permission bits on your file.

Nautilus requires setting the execute bit separately, by selecting the Allow Executing File As Program check box. This check box sets all three execute permission bits; you can't control execute permission more precisely with Nautilus. You also can't adjust the execute permissions on directories with Nautilus.

Setting Permissions in a Shell

Certification Objective

In a text-mode shell, you can use `chmod` to change permissions. This command is rather complex, mostly because of the complex ways that permissions may be changed. You can

Bresnahan, Christine, and Blum, Richard. Linux Essentials (2). Hoboken, US: Sybex, 2015. ProQuest ebrary. Web. 17 January 2017.

specify the permissions in two forms: as an octal number or in a symbolic form, which is a set of codes related to the string representation of the permissions.

> **The `chmod` command's name stands for *change mode*, mode being another name for permissions.**

The octal representation of the mode is the same as that described earlier and summarized in Table 14.2. For instance, to change permissions on `report.tex` to `rw-r––r––`, you can issue the following command:

```
$ chmod 644 report.tex
```

A symbolic mode, by contrast, consists of three components:

- A code indicating the permission set that you want to modify—`u` for the user (that is, the owner), `g` for the group, `o` for other users, and `a` for all permissions
- A symbol indicating whether you want to add (`+`), delete (`-`), or set the mode equal to (`=`) the stated value
- A code specifying what the permission should be, such as the common `r`, `w`, or `x` symbols, or various others for more-advanced operations

Using symbolic modes with `chmod` can be confusing, so we don't describe them fully here; however, you should be familiar with a few common types of use, as summarized in Table 14.3. Symbolic modes are more flexible than octal modes because you can specify symbolic modes that modify existing permissions, such as adding or removing execute permissions without affecting other permissions. You can also set only the user, group, or world permissions without affecting the others. With octal modes, you must set all three permission bits equal to a value that you specify.

> **As with the `chown` and `chgrp` commands, you can use the `-R` (or`––recursive`) option to `chmod` to have it operate on an entire directory tree.**

Table 14.3 Examples of symbolic permissions with **`chmod`**

Command	Initial permissions	End permissions
`chmod a+x bigprogram`	`rw–r––r––`	`rwxr-xr-x`
`chmod ug=rw report.tex`	`r––––––––`	`rw-rw––––`
`chmod o-rwx bigprogram`	`rwxrwxr–x`	`rwxrwx–––`
`chmod g-w,o-rw report.tex`	`rw-rw-rw-`	`rw-r–––––`

Setting the umask

The *user mask*, or *umask*, determines the default permissions for new files. The umask is the value that is *removed* from 666 (`rw-rw-rw-`) permissions when creating new files, or from 777 (`rwxrwxrwx`) when creating new directories. For instance, if the umask is 022, then files will be created with 644 permissions by default, and new directories will have 755 permissions. Note that the removal operation is not a simple subtraction but a bitwise removal. That is, a 7 value in a umask removes the corresponding `rwx` permissions; but for files, for which the starting point is `rw-`, the result is —— (0), not –1 (which is meaningless).

You can adjust the umask with the `umask` command, which takes the umask value, as in **`umask 022`**. Typically, this command appears in a system configuration file, such as `/etc/profile`, or in a user configuration file, such as `~/.bashrc`.

Using Special Permission Bits and File Features

When you investigate the Linux directory tree, you will encounter certain file types that require special attention. Sometimes, you may just want to be aware of how these files are handled, since they deviate from what you might expect based on the information presented in earlier chapters. In other cases, you may need to adjust how you use `ls` or other commands to deal with these files and directories—for example, when using the sticky bit, using special execute permissions, hiding files from view, or obtaining long listings of directories.

Using Sticky Bits

Certification Objective

Consider the following commands, typed on a system with a few files and subdirectories laid out in a particular way:

```
$ whoami
kirk
$ ls -l
total 0
drwxrwxrwx 2 root root 80 Dec 14 17:58 subdir
$ ls -l subdir/
total 2350
-rw-r — – 1 root root 2404268 Dec 14 17:59 f1701.tif
```

These commands establish the current configuration: The effective user ID is `kirk`, and the current directory has one subdirectory, called `subdir`, which `root` owns but to which `kirk`, like all of the system's users, has full read/write access. This subdirectory has one file, `f1701.tif`, which is owned by `root` and to which `kirk` has no access. You can verify that `kirk` can't write to the file by attempting to do so with the `touch` command:

```
$ touch subdir/f1701.tif
touch: cannot touch  'subdir/f1701.tif ': Permission denied
```

This error message verifies that `kirk` could not write to `subdir/f1701.tif`. The file, you might think, is safe from tampering. Not so fast! Try this:

```
$ rm subdir/f1701.tif
$ ls -l subdir/
total 0
```

The `rm` command returns no error message, and a subsequent check of `subdir` verifies that it's now empty. In other words, `kirk` could delete the file even without write permission to it! This may seem like a bug—after all, if you can't write to a file, you might think that you shouldn't be able to delete it. Recall, however, that directories are just a special type of file, one that holds other files' names and pointers to their lower-level data structures. Thus modifying a file requires write access to the file, but creating or deleting a file requires write access to the *directory in which it resides*. In this example, `kirk` has write access to the `subdir` directory, but not to the `f1701.tif` file within that directory. Thus `kirk` can delete the file but not modify it. This result is not a bug; it's just a counterintuitive feature.

Although Linux filesystems were designed to work this way, such behavior is not always desirable. The way to create a more intuitive result is to use a *sticky bit*, which is a special filesystem flag that alters this behavior. With the sticky bit set on a directory, Linux will permit you to delete a file only if you own either it or the containing directory; write permission to the containing directory is not enough. You can set the sticky bit with `chown` in either of two ways:

Using an Octal Code By prefixing the three-digit octal code described earlier in this chapter with another digit, you can set any of three special permission bits, one of which is the sticky bit. The code for the sticky bit is 1, so you would use an octal code that begins with 1, such as 1755, to set the sticky bit. Specifying a value of 0, as in 0755, removes the sticky bit.

Other odd numbers will set the sticky bit, too, but will also set additional special permission bits, which are described shortly, in "Using Special Execute Permissions."

Using a Symbolic Code Pass the symbolic code `t` for the world permissions, as in **`chmod o+t subdir`**, to set the sticky bit on `subdir`. You can remove the sticky bit in a similar way by using a minus sign, as in **`chmod o-t subdir`**.

Restoring the file and setting the sticky bit enables you to see the effect:

```
$ ls -l
total 0
drwxrwxrwt 2 root root 80 Dec 14 18:25 subdir
$ ls -l subdir/
total 304
-rw-r--r-- 1 root root 2404268 Dec 14 18:25 f1701.tif
$ rm subdir/f1701.tif
rm: cannot remove  `subdir/f1701.tif ': Operation not permitted
```

In this example, although `kirk` still has full read/write access to `subdir`, `kirk` cannot delete another user's files in that directory.

You can identify a directory with the sticky bit set by a small change in the symbolic mode shown by `ls -l`. The world execute bit is shown as a `t` rather than an `x`. In this example, the result is that `subdir`’s permission appears as `drwxrwxrwt` rather than `drwxrwxrwx`.

The sticky bit is particularly important for directories that are shared by many users. It’s a standard feature on `/tmp` and `/var/tmp`, for instance, since many users store temporary files in these directories, and you wouldn’t want one user to be able to delete another user’s temporary files. If you want users who collaborate on a project to be able to write files into each others’ home directories, you might want to consider setting the sticky bit on those home directories, or on the subdirectories in which users are sharing files.

> **If you delete `/tmp` or `/var/tmp` and need to re-create it, be sure to set the sticky bit on your new replacement directory!**

Using Special Execute Permissions

As described earlier in this chapter, the execute permission bit enables you to identify program files as such. Linux then allows you to run these programs. Such files run using your own credentials, which is generally a good thing—associating running processes with specific users is a key part of Linux’s security model. Occasionally, though, programs need to run with elevated privileges. For instance, the `passwd` program, which sets users’ passwords, must run as `root` to write, and in some cases to read, the configuration files it handles. Thus if users are to change their own passwords, `passwd` must have `root` privileges even when ordinary users run it.

Certification Objective

To accomplish this task, two special permission bits exist, similar to the sticky bit described earlier:

Set User ID (SUID) The *set user ID (SUID)* option tells Linux to run the program with the permissions of whoever owns the file rather than with the permissions of the user who runs the program. For instance, if a file is owned by `root` and has its SUID bit set, the program runs with `root` privileges and can therefore read any file on the computer. Some servers and other system programs run this way, which is often called SUID `root`. SUID programs are indicated by an `s` in the owner’s execute bit position in the permission string, as in `rwsr-xr-x`.

Set Group ID (SGID) The *set group ID (SGID)* option is similar to the SUID option, but it sets the group of the running program to the group of the file. It’s indicated by an `s` in the group execute bit position in the permission string, as in `rwxr-sr-x`.

You can set these bits by using `chmod`:

Using an Octal Code In the leading digit of a four-digit octal code, set the leading value to 4 to set the SUID bit, to 2 to set the SGID bit, or to 6 to set both bits. For instance, 4755 sets the SUID bit, but not the SGID bit, on an executable file.

Using a Symbolic Code Use the `s` symbolic code, in conjunction with `u` to specify the SGID bit, `g` to specify the SGID bit, or both to set both bits. For instance, typing **`chmod u+s myprog`** sets the SUID bit on `myprog`, whereas **`chmod ug-s myprog`** removes both the SUID bit and the SGID bit.

Ordinarily, you don't need to set or remove these bits; when necessary, the package management program sets these bits correctly when you install or upgrade a program. You might need to alter these bits if they've been mistakenly set or removed on files. In some cases, you might want or need to adjust these values on program files that you compile from source code or if you need to modify the way a program works. Be cautious when doing so, though. If you set the SUID or SGID bit on a garden-variety program, it will run with increased privileges. If the program contains bugs, those bugs will then be able to do more damage. If you accidentally remove these permissions, the results can be just as bad—programs like `passwd`, `sudo`, and `su` all rely on their SUID bits being set, so removing this feature can cause them to stop working.

Hiding Files from View

Certification Objective

If you're used to Windows, you may be familiar with the concept of a *hidden bit*, which hides files from view in file managers, by the Windows `DIR` command, and in most programs. If you're looking for something analogous in Linux, you won't find it—at least not in the form of a dedicated filesystem feature. Instead, Linux uses a file-naming convention to hide files from view: most tools, such as `ls`, hide files and directories from view if their names begin with a dot (`.`). Thus `ls` shows the file `afile.txt`, but not `.afile.txt`. Most file managers and dialog boxes that deal with files also hide such *dot files*, as they're commonly called; however, this practice is not universal.

Many user programs take advantage of this feature to keep their configuration files from cluttering your display. For instance, `~/.bashrc` is a Bash user configuration file, Evolution's configuration files go in the `~/.evolution` directory, and `~/.fonts.conf` holds user-specific font configuration information.

You can view dot files in various ways depending on the program in question. Some GUI tools have a check box that you can set in their configuration options to force the program to display such files. At the command line, you can add the `-a` option to the other options in `ls`:

```
$ ls -l
total 0
drwxrwxrwt 2 root root 80 Dec 14 18:25 subdir
$ ls -la
total 305
drwxr-xr-x 3 kirk users    104 Dec 14 18:44 .
drwxr-xr-x 3 kirk users    528 Dec 14 18:21 ..
-rw-r –r – 1 kirk users 309580 Dec 14 18:44 .f1701.tif
drwxrwxrwt 2 root root      80 Dec 14 18:25 subdir
```

Certification Objective

This example shows the hidden file `.f1701.tif` in the current directory. It also shows two hidden directory files. The first `.` refers to the current directory. The second `..` refers to the parent directory.

Recall from Chapter 7 that `..` is a relative directory reference. This hidden entry is why it works.

Note that renaming a file so that it begins with a dot will hide it, but this action will also make the file inaccessible to any program that uses the original filename. That is, if you rename `f1701.tif` to `.f1701.tif`, and if another program or file refers to the file as `f1701.tif`, that reference will no longer work. You *must* include the leading dot in any reference to the hidden file.

Viewing Directories

Certification Objective

Chapter 6 introduced the `ls` command, including many of its options. One of these deserves elaboration at this point: `-d`. If you're working in a directory that holds many subdirectories, and if you use a wildcard with `ls` that matches one or more subdirectories, you may get an unexpected result: the output will show the files in the matched subdirectories, rather than the information on the subdirectories themselves. For instance, say you start in a directory with two subdirectories, `subdir1` and `subdir2`:

```
$ ls -l subdir*
subdir1:
total 304
-rw-r––r–– 1 kirk users 309580 Dec 14 18:54 f1701.tif

subdir2:
total 84
-rw-r––r–– 1 kirk users 86016 Dec 14 18:54 106792c17.doc
```

The /proc and /sys directories contain real-time data populated automatically by the kernel so you can view process and device status. Those files and subdirectories may appear and change at any time, making it tricky to display them.

If instead you want information on the subdirectories rather than the contents of those subdirectories, you can include the `-d` option:

```
$ ls -ld subdir*
drwxr-xr-x 2 kirk users 80 Dec 14 18:54 subdir1
drwxr-xr-x 2 kirk users 80 Dec 14 18:54 subdir2
```

The Essentials and Beyond

File security is important on a multiuser OS such as Linux, and one of the pieces of the puzzle of security is ownership. In Linux, every file has one owner and one associated group. The superuser can set the owner with `chown`, and either the superuser or the file's owner can set the file's group with `chown` or `chgrp`. By itself, ownership is useless, so Linux supports the concept of file permissions to control which other users can access a file and in what ways. You can set permissions with the `chmod` utility. You can view ownership, permissions, and some additional file features by using the `-l` option to the `ls` command.

Suggested Exercises

- As `root`, copy a file that you created as an ordinary user, placing the copy in your ordinary user home directory. Using your normal account, try to edit the file with a text editor and save your changes. What happens? Try to delete that file with the `rm` command. What happens?
- Create a scratch file as an ordinary user. As `root`, use `chown` and `chmod` to experiment with different types of ownership and permissions to discover when you can read and write the file by using your normal login account.
- Use the `ls -l` command to view the ownership and permissions of files in your home directory, in `/usr/bin` (where many program files reside), and in `/etc` (where most system configuration files reside). What are the implications of the different ownership and permissions you see for who can read, write, and execute these files?

Review Questions

1. What command would you type (as `root`) to change the ownership of `somefile.txt` from `ralph` to `tony`?
   - A. `chown ralph:tony somefile.txt`
   - B. `chmod somefile.txt tony`
   - C. `chown somefile.txt tony`
   - D. `chown tony somefile.txt`
   - E. `chmod tony somefile.txt`
2. Typing **`ls -ld wonderjaye`** reveals a symbolic file mode of `drwxr-xr-x`. Which of the following are true? (Select all that apply.)
   - A. `wonderjaye` is a symbolic link.
   - B. `wonderjaye` is an executable program.
   - C. `wonderjaye` is a directory.

Bresnahan, Christine, and Blum, Richard. Linux Essentials (2). Hoboken, US: Sybex, 2015. ProQuest ebrary. Web. 17 January 2017.

   D. All users of the system may read `wonderjaye`.
   E. Any member of the file's group may write`wonderjaye`.

3. Which of the following commands can you use to change a file's group?
   A. `groupadd`
   B. `groupmod`
   C. `chmod`
   D. `ls`
   E. `chown`
4. True or false: A file with permissions of 755 can be read by any user on the computer, assuming that all users can read the directory in which it resides.

5. True or false: Only `root` may use the `chmod` command.
6. True or false: Only `root` may change a file's ownership with `chown`.
7. What option causes `chown` to change ownership on an entire directory tree?
8. What three-character symbolic string represents read and execute permission but no write permission?
9. What symbolic representation can you pass to `chmod` to give all users execute access to a file without affecting other permissions?
10. You want to set the sticky bit on an existing directory, `subdir`, without otherwise altering its permissions. To do so, you would type **`chmod`** ________ **`subdir`**.

Bresnahan, Christine, and Blum, Richard. Linux Essentials (2). Hoboken, US: Sybex, 2015. ProQuest ebrary. Web. 17 January 2017.