Sharing Files (Carefully)

By Jerry Peek

Linux systems have multiple users — and a permissions system that lets each user share or protect their files. This unique system has some odd twists. For example, did you know that, to rename or remove a file, you need write permission for the *directory* that the file is in? A thorough understanding of permissions lets you make your system more secure, share data easily with other users, and protect files from accidental changes. Let's look into the basics — and some dark corners — of Linux filesystem permissions.

The Basics: Users and Groups

In Linux, every user has a username (like `zoe`) and a corresponding user ID number (or UID) like `1324`. Every user also belongs to one or more groups, where each group also has its own name and ID number (the GID). You can run the *id* command to see your user and group information:

```
% id
uid=1324(zoe) gid=1324(zoe)
groups=1324(zoe),111(staff)
```

Every file and directory has a user owner and a group owner, as well as a set of permissions that dictates what access is allowed for the file's user, group, and others. For example, here's an `ls -ld` listing of *zoe*'s home directory (the `-d` option lists the directory itself instead of its contents):

```
% pwd
/home/zoe
% ls -dl
drwxr-x—x 5 zoe zoe 4096 Nov 2 10:33 .
```

Many Linux references cover `ls -l`, but here's a quick review.

- The first character on each `ls -l` output line shows the file's type. Here, `d` is a directory file. This file's user owner is `zoe`, and its group owner is the *group* named `zoe`.
- The next characters are the types of access allowed for the user owner (here, `rwx`), for members of the group (`r-x`), and for all others.

The meaning of the `rwx` permissions for files is simple: `r` permits reading from a file, `w` lets you modify it, and `x` lets you execute it (as a program). However, how `rwx` applies to directories isn't so obvious. Let's dig in!

Reading and Writing Directories

Read (`r`) access to a directory lets you list its contents using *ls* or with a GUI file browser. Why? As last month's column (available online at http://www.linux-mag.com/2002-10/power_01. html) explained, a directory is a special type of file that contains (among other things) the names of files that are stored "in" the directory. So, being able to read a directory file lets you see the names stored in it.

In the same way, write (`w`) access to a directory lets you modify the directory's contents — to change the names of the files stored in that directory. To add, remove, or rename a file, you need write permission on *the directory that contains the file's name*. If a directory is write-protected, the list of entries in that directory cannot be changed — accidentally or on purpose.

Directory "x" Permission: Surprise!

Execute (`x`) permission on a directory lets you access entries in that directory. Even if you have read permission on a directory (which lets you see what's in it), you can't *use* any of its entries unless you also have `x` permission.

For example, even if an individual file's access permissions allow you to read and write it, you still can't read or write the file unless you have execute permission on its directory. Here's a demonstration:

```
% mkdir safe
% chmod og-rwx safe
% echo "Hello" > safe/secret
% chmod a-x safe
% ls safe
secret
% cat safe/secret
cat: safe/secret: Permission denied
% chmod u+x safe
% cat safe/secret
Hello
```

In fact, you can't access a directory unless you also have execute permission on *all* of its *parent* directories, all the way up to the root directory! Similarly, without execute permission on a directory, you can't access its subdirectories.

But you don't need read (`r`) permission on a directory to access an entry in it *if you know the entry's exact name*! (Note that this may not be true on graphical systems, though. See the sidebar "Graphic Confusion".)

Sharing Files (Carefully)

Linux permissions let groups of users share files — even modify or remove other users' files — if they're given permission. Directory and file access permissions are often set to allow general file sharing, so let's see a specialized example.

Let's say that Zoe's home directory is on a networked filesystem shared between all computers in her office. She could tell her friends to point their photo viewers at her directory, *~zoe/xfotos* (the shell expands `~zoe` to the absolute pathname of *zoe*'s home directory; Zoe herself can use just `~`). Her permissions give everyone just enough access to read the photos, but not enough access to add, delete, or modify files:

```
% ls -dl ~ ~/xfotos
drwxr-x—x  zoe zoe ...  /home/zoe
drwxr-xr-x  zoe zoe ...  /home/zoe/xfotos
% ls -l ~/xfotos
-rw-r—r—  zoe zoe ...  alix.jpg
-rw-r—r—  zoe zoe ...  boss.jpg
...
```

This example shows one of the strengths of a Linux filesystem: it's easy to share when you want to. Zoe didn't need to fill friends' mailboxes with individual copies of her photos, or configure the office Web server so her friends (but no one else) could see those files. Instead, she simply "opened" her directory and her files just enough to let people read them.

An easy way to keep your private files private is to make specific directories in your home directory with owner-only access. For instance, make a directory named *personal* and type `chmod 700 personal` to give yourself all (`rwx`) permissions, but no access to members of your group or to others. All files and subdirectories under *personal* — no matter what their permissions — are protected from everyone except the system superuser. (You might prefer to give your private directories a less obvious name than *private*, which could be a red flag for crackers and others who get into your account somehow.)

Default Permissions: The umask

If you're concerned about security but also need to share files, check the permissions of every file — including "hidden" files like *.bashrc* — before you start sharing.

You should also set the correct *umask*. The umask defines the default permissions of *new* files and directories. The umask is a *bit mask* that specifies which permissions to *deny* when creating a file. You can see the current umask setting by typing `umask` at a shell prompt. To set it, use `umask` *nnn*, where *nnn* is the new umask.

With no umask (a umask of `0`), directories are created with mode 777 (which `ls -l` shows as `rwxrwxrwx`), and files get mode 666 (`rw-rw-rw-`). (If you haven't seen permissions written as octal numbers, think of each `r` as worth 4, each `w` as 2, and each `x` as 1. So `rwx` is worth 7 (4+2+1), `rw-` is 6 (4+2), and `r—` is 4.)

The three digits of a umask are written in the same order as you'd specify wth *chmod*: first, the user mask; then, the group; finally, all others. So, a umask of `002` denies no (`0`) permission for the user owner and group owner, and denies write (`2`) permission for others. A umask of `027` would deny no (`0`) permission for user, write (`2`) permission for group, and all (`7`, or 4+2+1) permissions for other users.

An easy way to see what a umask does is to set it, create an empty file and directory (with *touch* and *mkdir*), and then use `ls -l` to see the permissions (here we're using the shell's semicolon (`;`) operator to run two commands from a single prompt):

```
% umask 027
% touch file; mkdir dir
% ls -l file; ls -ld dir
-rw-r—-  jpeek  jpeek ... file
drwxr-x—-  jpeek  jpeek ... dir
% rm file; rmdir dir
```

The umask propagates as one process starts another. (See the sidebar "Spreading the News.") You can experiment with umask on the command line (from a shell prompt), but remember that changing the umask here won't affect other existing shells or processes.

Using Groups

Look back at the output of *id* near the start of this article. Zoe belongs to two groups: `zoe` and `staff`. Linux groups can have many members, and each user can belong to many groups.

For instance, all system administrators and staff might belong to the `staff` group. They have access to private, group-only directories and files that all non-members (non-staff users) can't access. In the same way, Zoe's manager and the office manager might be members of the `zoe` group — to have group-mode access to some or all of her files and directories.

If you belong to a group (and you always belong to at least one group: yours), you can change the group owner of any file or directory you own with the *chgrp* command.

For instance, Zoe might keep confidential notes on system users in a file named *users.xml*. She'd use `chgrp staff users.xml` to set the file's group owner to `staff`. Then she'd use `chmod 750 users.xml` (or `chmod u=rw,g=r,o= users. xml`) to give herself read and write access, give read access to members of the group `staff`, and no access to other users. Any `staff` member can open the file, but other users have no access.

If you belong to several groups, which group owns files you

create? By default, it's your primary group — the group listed first in the *id* output, after `gid=`. That's usually not what you want in a directory shared by all users from a particular group.

For example, the directory */prj/4staff* might have the permissions `drwxrwx—-` so all members of the `staff` group can create, remove, and rename files in that directory. But if zoe creates a file in that directory with her default group of `zoe`, other members of the `staff` group probably can't read it (unless they're also members of the `zoe` group).

The answer is to set the directory's *setgid* bit, as `pete` does below. (This bit is shown by `s` in the permission string `drwxrws—-`.) Files (and subdirectories) created in a directory with its setgid bit on will have the same group owner as their directory. So, with the following settings, any file created in */prj/4staff* will automatically have the group owner `staff`:

```
pete$ chmod g+s /prj/4staff
pete$ ls -ld /prj/4staff
drwxrws—- 5 pete   staff    /prj/4staff
```

SPREADING THE NEWS

How does a process, like a text editor or a shell, get the umask and the user and group IDs for creating new files? All of that information and other information (such as the current directory) is passed to a new process from its parent process. Once a new process is spawned, it's free to change values like its umask — but note that these changes *do not* affect the parent or other existing processes.)

You'll probably want to set your umask as early as possible, probably during the logon process or in the script that launches your window system. Setting the umask as soon as possible means that all subsequent child processes (such as terminal windows and file managers) inherit that umask from there.

GRAPHIC CONFUSION

Graphical applications may not handle Linux permissions very well. Because they usually show dialogs with icons for the files in a directory, they have to be able to read the directory. So, if users try to reach Zoe's *xfotos* subdirectory by clicking through */home* to */home/zoe*, they'll get some sort of error on her home directory because the GUI app can't read it.

Some applications — graphical and otherwise — also aren't sophisticated about write permission: they assume that, if they can read a directory, they also can write to it. Writes can fail in strange ways because, although a particular file in a directory may be writable, the directory itself isn't.

GUI file-open dialogs will generally let you type (or paste) an absolute pathname (like */home/zoe/xfotos*) into their "location" box. This lets you bypass unreadable intermediate directories. (It can also save time!)

Protecting Files from Yourself

If you're the only user of your Linux system, and your filesystems aren't networked to (or from) other computers, you may wonder why you need to understand permissions. One good reason is to prevent accidents — unintended deletes or changes to *your own* files and directories. Mistakes happen, so protecting yourself makes sense.

If *report.doc* is an important file, you can make it unwritable by everyone — including yourself — with `chmod a-w report.doc`. As we said earlier, though, setting permissions on the file's contents don't control whether the file can be removed or replaced! (For instance, if you run `editor report.doc`, the *editor* program might rename *report.doc* to *report.doc.bak*, then create a new *report.doc* file. That's perfectly legal.) To protect against all changes, the file's *directory* also needs to be unwritable.

Until next month...

The permissions scheme we've seen works on Unix systems, in general, as well as on Linux. Some Linux filesystems — the widely-used *ext2* and *ext3* types, for instance — also have other file *attributes*. The manual pages *lsattr* and *chattr* explain how to list and change file attributes.

Here's a quick example. The superuser can use *chattr* to set the `a` attribute, which makes a file append-only: the file can't be modified, only added to. Running *lsattr* shows this attribute, but `ls -l` doesn't.

```
# chattr +a log
# exit
% ls -l log
-rw-rw-r— 1 zoe staff 1168 Oct 30 12:29 log
% lsattr log
——-a— log
% cp /var/log/dmesg log
cp: overwrite `log'? y
cp: cannot create regular file `log':
Operation not permitted
% cat /var/log/dmesg >> log
% ls -l log
-rw-rw-r— 1 zoe staff 3109 Oct 30 12:33 log
```

Next month's column will dig into advanced features of Mozilla 1.1. Between now and then, why not think of ways to use your filesystem's permissions to protect — and share — your data?

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers at jpeek@jpeek.com.