chmod, chgrp, and chown

ACCESS GRANTED!

A sophisticated system of users and permissions precisely controls who has access to what on Linux. At the command line, you can define ownership with the chmod, chgrp, and chown tools. **BY HEIKE JURZIK**

Granular access privileges for files and directories are what make Linux a safe operating system. A precise definition of who is permitted to read, modify data, or execute specific programs provides excellent protection against any prying eyes and intentional misconfiguration.

The administrator, *root*, is subject to no restrictions, and this includes assigning read, write, and execute permissions to other users throughout the system. If you are the owner of a file or directory, you can grant access to these resources to other accounts. If you are also a member of a specific group, you can modify the group ownership of files and folders for more granular permission assignments to files.

Rights and Obligations

For every file (and thus for directories, device files, and so on), Linux precisely defines who is permitted to read, write, and execute that file. Additionally, every file belongs to a user and to a group. The three permissions are assigned separately for these three categories and for users who do not belong to any of the three categories:

- Read permission: Users can display the content of a file or folder on screen, copy the file, and do a few other things.
- Write permission: Users can change files and directories and store their changes. This also includes the ability to delete.
- Execute permission: For programs, execute permission means that the user is permitted to run the program. Execute for a directory means that the

Table 1: Permissions Overview

Octal number	Letters
0	---
1	--x
2	-w-
3 (= 2+1)	-wx
4	r--
5 (= 4+1)	r-x
6 (= 4+2)	rw-
7 (= 4+2+1)	rwx

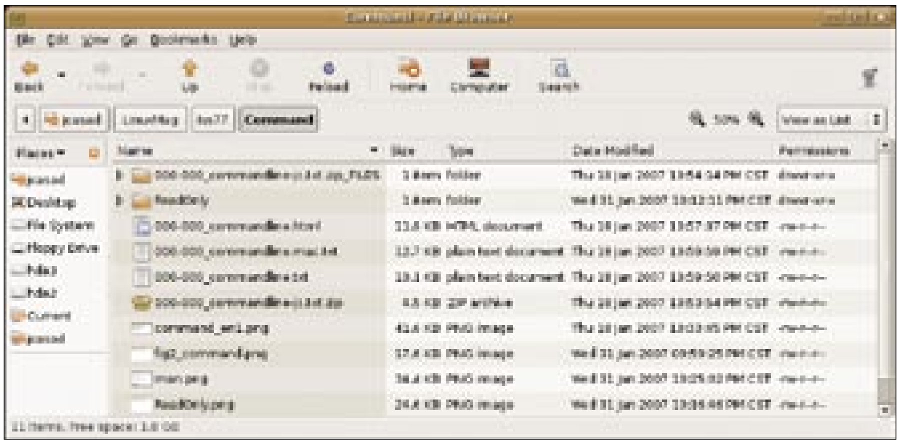

Figure 1: Most file managers provide an option for viewing file permissions.

user is permitted to change to the directory (the user will additionally need read permission to be able to view the folder content).

Discover Permissions

To discover the permissions for a file, you can either switch to a detailed folder view in a graphical file manager like Konqueror or Nautilus, or you can simply set the *-l* flag for the *ls* command.

In both cases, permissions are indicated by the letters *r* (for "read"), *w* (for "write"), and *x* (for "execute"). The first block of three shows the permissions for the owner, the second block refers to the group, and the third block refers to all users. Folders are indicated by a *d* (for "directory") at the start of the list (see Figure 1).

Special Permissions

Linux also has two special permissions: the *s* bit (also known as the setuid/setgid bit) and the *t* bit (also known as the sticky bit). Both replace the *x* in the *rwx* block of three.

The *s* is commonly seen with executable files, whereas the *t* bit is more common with directories.

As the name setuid/setgid bit (set user ID and set group ID, respectively) would suggest, this bit executes a program with the permissions of a user or group no matter who runs the program. In this way, nonprivileged users can access resources they would not normally be able to access.

Tip

Instead of *ugo*, you could simply say a for "*all*" with *chmod*.

Although this is a potential security risk, the *s* bit has its uses. Many programs, including *su*, *sudo*, *mount*, or *passwd* in the following example rely on the *s* bit:

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root ➘
27132 Jul 11 20:06 ➘
/usr/bin/passwd*
```

The *passwd* program modifies passwords, accessing the */etc/shadow* file in the process to enter the new password. By default, the file is protected against write access by nonprivileged users and reserved for use by the administrator to prevent just anybody manipulating the passwords. The *s* bit executes the *passwd* program as the root user and enters the new password in */etc/shadow* on behalf of root.

The other special permission, the *t* bit, commonly occurs in shared directories (read, write, and execute permissions for all) in place of the execute flag to ensure that users are only allowed to modify – and thus delete – their own data.

The sticky bit is also typically set for */tmp*, as seen here:

```
$ ls -ld /tmp
drwxrwxrwt 16 root ➘
root 4096 Jan
 28 19:51 /tmp/
```

The */tmp* folder stores temporary files for multiple users.

If everybody had the right to read, write, and execute these files, in theory, everybody would be able to clean up the system and delete arbitrary data.

However, the *t* bit prevents this from happening, ensuring that users can only delete their own files (or those files for which they have been given write permission). The exception to this rule is that the owner of the folder with the sticky bit is also allowed to delete within that folder.

Modifying Permissions

The *chmod* program lets you modify file and directory permissions, assuming you are the owner or the system administrator, and understands two different kinds of command.

In one mode, you can use letters to define permissions. In this case, *u* stands for "user" (owner), *g* for "group," and *o* for "others" (all other users); *r* stands for "read," *w* for "write," *x* for "execute," *s* for the setuid/setgid bit, and *t* for the sticky bit.

A combination of these letters with plus, minus, and equals signs tells *chmod* to add, remove, or assign, respectively, precisely these permissions. For example, to give a group read and write permissions for a file, you just type *chmod g+rw file*.

Removing permissions follows the same pattern: the *chmod o-rwx file* command removes all permissions for all users who are neither the owner nor members in the owner group.

You are also able to combine these two commands like this:

```
chmod g+rw,o-rwx datei
```

As mentioned previously, an equals sign lets you assign precisely the permissions specified at the command line. For example, the command:

```
chmod ugo=rxw directory
```

gives the owner, group members, and all other users read, write, and execute permissions for the specific directory that is in question.

GLOSSARY

Octal numbers: The octal system uses base 8; that is, it includes just eight numbers between 0 and 7. The next number after 7 is 10, 20 follows 17, and so on. Every number in an octal number is represented by three bits; in the case of permissions, the three bits specify what a user class is allowed to do [1].

The *chmod* program also understands letters. When you run the tool, you can pass in three- or four-digit octal numbers instead of letters.

You can calculate the numbers as follows: 4 stands for read permission, 2 for write permission, and 1 for execute permission. The first number refers to the owner, the second number to the group, and the third to all others.

On this basis, you can see that, for example, 644 would mean *u = rw,go = r* (resulting in *rw-r--r--*), or 777 would be *a = rwx* (resulting in *rwxrwxrwx*). The "Permissions Overview" table provides more details.

To set the *s* or *t* bit, you need to add this as a fourth number at the start of the block of three.

The number 4 represents the *s* bit for the owner (setuid), 2 sets the *s* bit for the group (setgid), and 1 sets the *t* bit. Listing 1 gives an example.

Changing Group Memberships

As a "normal" user, you are allowed to assign your own files to specific groups; however, this assumes that you are a member of the group in question. The following command tells you your own group memberships:

```
$ groups
huhn dialout cdrom ⮐
floppy audio
video
```

To assign a file to the *audio* group, you just type:

```
chgrp audio Datei
```

Changing Owners and Groups

On a Linux system, the system administrator is allowed to assign new owners and new groups to files and directories.

Listing 1: Example

```
01 $ ls -l script.sh
02 -rw-r--r-- 1 huhn huhn 3191789
   Oct 6 05:01 script.sh
03 $ chmod 4755 script.sh
04 $ ls -l script.sh
05 -rwsr-xr-x 1 huhn huhn 3191789
   Oct 6 05:01 script.sh
```

Let's imagine you just set up a new account called *mike*, and you've set up a home directory for Mike and copied critical configuration files from */etc/skel*.

Your last step would be to give Mike the permissions he needs to set up shop and use his home directory and the subdirectories below it.

The following command hands over the home directory and all the files in it (including the hidden configuration files) to the user *mike*:

```
chown -R mike /home/mike
```

The *-R* option used here tells *chown* to act recursively (this will be explained more later). It is also useful to be able to define a new group owner for the data at the same time:

```
chown -R mike:mike /home/mike
```

In other words, you just append the group name (some distributions have a default group called *users*, whereas other distributions use the account name as the default group), with a colon to separate it from the account name.

Across the Board

All three tools – *chmod*, *chgrp*, and *chown* – support an *-R* parameter for recursive actions. For example, if you want to permit the members of the *video* group to access a directory and the files it contains, just type:

```
chgrp -R video directory
```

The *-R* option can also save you much typing when used in combination with the *chmod* command.

To remove read, write, and execute permissions from this folder for all users who are *not* the owner or members of the *video* group, just type:

```
chmod -R o-rwx directory
```

Word of Warning

Be careful when you run recursive commands that remove the execute flag. If you mistakenly type *a-x*, instead of *o-x*, you will discover that you have locked yourself out: *chmod* removes execute permissions from the parent directory and your ability to change to the directory and modify the files.

The use of *find* can help you avoid this kind of dilemma:

```
find directory -type f -exec
chmod a-x "{}" ";"
```

The *find* command first discovers the files (*-type f*) and then runs *chmod* against them, ignoring the directory.

From the Beginning

The *umask* specifies the default permissions assigned to newly created files and directories. Typing the *umask* command without setting any parameters reveals the current setting:

```
$ umask
0022
```

What you see here is a four-digit octal number that specifies what to subtract from the the default values (0666 for files, 0777 for directories). In other words, new files are assigned 0644 (*rw-r--r--*), and new folders are assigned 0755 (*rwxr-xr-x*) when they are created. To change the umask, enter the file and the new value at the command line:

```
umask 0077
```

This entry means that new files and directories are only available to their owner. The umask is valid for the current shell, but you can add an entry to your bash configuration file *~/.bashrc* to make the change permanent. Working as root, you also could add a global entry to */etc/profile* to modify the umask for the system. ■

INFO

[1] Octal numbers: *http://en.wikipedia.org/wiki/Octal*

THE AUTHOR

Heike Jurzik studied German, Computer Science and English at the University of Cologne, Germany. She discovered Linux in 1996 and has been fascinated with the scope of the Linux command line ever since. In her leisure time you might find Heike hanging out at Irish folk sessions or visiting Ireland.