LINUX JOURNAL
(/)

The chmod Command

Software (/taxonomy/term/17)

by Eric Goebelbecker on January 1, 1996

Do you know how to rename a file you can't read? Better yet, do you know how other users can rename your files? Have you ever ftp'd a program from another host and been unable to run it?

The subject of file permissions, and how to manipulate them with the **chmod** command, is a good place to start learning about these situations.

First, let's create a file and examine its long listing. (In order to fit in the magazine, all the listings in this article are trimmed to fit.)

```
$ touch test_file
$ ls -l test_file
-rw-rw-r--   1 eric     users
```

Since I created this file, it makes sense that the third column shows my user name as the file's owner and that the fourth shows my group. (On some systems, the group name may be the same as the user name.) As you follow along in these examples, you will see your username in place of “eric”.

The leftmost column of the directory listing shows the file's **mode**. Mode is the term used to refer to a file's permissions. ls displays the file's type and mode together as a grouping of ten one-character fields:

Type Owner Group World - rwx rw- r--

The type field has several valid values. For the sake of this tutorial, we are only concerned with two: empty (**-**) for a regular file, and **d** for directories.

The other three columns cover the three **classes** of access that are stored for each file in a Unix-like file system. Linux (and Unix) evaluates access in terms of user ownership, group ownership and world (or other).

For each of these classes, rights are evaluated in terms of three **operations**: reading (**r**), writing (**w**) and executing (**x**). The permissions above specify “full” access for the owner, reading and writing for group, and only reading for world (an unusual combination used for demonstration). Those permissions specify that

- The owner of the file is allowed to read, write and execute the file.
- Any user who is a member of the group that owns the file is permitted to write to the file.
- Any other user can only read the file.

Changing permissions
If test_file were a very important document that we did not want anyone to be able to modify or delete, we would need to remove write access from group:

```
$ chmod g-w test_file
$ ls -l test_file
-rw-r--r--   1 eric     users
```

We see that the **w** for group is now replaced with a **-**, signifying that write permission is denied to members of the group **users**.

If test_file contained sensitive information that only members of the group **users** should be able to review:

```
$ chmod o-r test_file
$ ls -l test_file
-rw-r-----   1 eric     users
```

Now we see that the last triplet of the mode field, which specifies permissions for world, are all dashes. This means that other users who do not belong to the **users** group have no permissions to do anything with test_file whatsoever.

The command line usage for chmod mode looks like this:

```
chmod [options] new-mode filename
```

The new mode is specified in **octal mode** or **symbolic mode**. We'll cover symbolic mode first. In the first example we used **g-w** to remove write permission for group. As you might be able to guess, **g** stood for group, **-** for remove and **w** represented write permission.

```
$ chmod g+wx test_file
$ ls -l test_file
-rw-rwx---   1 eric     users
```

This operation added permission for group to write and execute.

Let's look at an example of these permissions in action.

```
$ chmod u-rwx test_file
$ ls -l test_file
----rwx---   1 eric     users
$ cat test_file
cat: test_file: Permission denied
$ cat .profile > test_file
bash: test_file: Permission denied
```

We are not able to display the file's contents because we do not have read access to our own file. When we specified **u-rwx** to chmod, we removed all access for the user (the file's owner). We were also denied permission when we attempted to add the contents of another file to it since we removed write access. (I should note that **rm** would still be able to delete this file, although it will normally request confirmation.)

```
$ chmod u+rwx test_file
$ ls -l test_file
-rwxrwx---   1 eric     users
```

When we specify **u+rwx**, all permissions are restored. Removing permissions from a file we own does not affect our ability to restore the permissions, because the mode is **not** stored in the file. It is stored in a structure called an inode entry. Only the owner of the file (and root) may modify this.
Understanding chmod
Let's look at a summary of chmod's options, and then cover each option in depth:

User

> u user (owner)
>
> g group
>
> o other (world)
>
> a all (user, group, and other)

Operation

> + add
>
> - remove
>
> = set exactly

Mode

r read

w write

x execute

X conditionally set execute

s Set UID or set GID

t set “sticky” bit

```
$ chmod a+rwx test_file
$ ls -l test_file
-rwxrwxrwx   1 eric     users
```

This demonstrates the fourth possible symbol for user when using symbolic mode. We used **a** to set full permissions for all user classes at once. Let's delete the file and start over in order to demonstrate the difference between the **=** operator and the **+** and **-** operators. (From here on, we'll assume that you know how to get the directory listing, and won't list the ls command.)

```
$ rm test_file
$ touch test_file
-rw-rw-r--   1 eric     users
$ chmod g+x test_file
-rw-rwxr--   1 eric     users
```

This added execute permission for group.

```
$ chmod g=x test_file
-rw---xr--   1 eric     users
```

The **=** operators set group's permissions to execute, and in doing so removed read and write permission. While **+** and **-** set or unset the permissions specified, **=** will set *exactly* the mode specified and remove any others.

Read, write and execute modes are very straightforward when referring to files. Read and write allow a user to examine and modify/delete data from a file, respectively. Execute allows a user to execute a shell script or binary program. If you ftp a program from one host to another and then try to run it without setting execute permission, it will fail, since ftp does not set execute permission.
Directories
For directories, the rules can be a bit more complicated.

Read permission allows a user to examine the contents of a directory.

```
$ mkdir test_dir
$ touch test_dir/foo
$ ls test_dir
foo
$ chmod u-r test_dir
$ ls test_dir
ls: test_dir: Permission denied
```

Write permission allows a user to modify the contents **of the directory**. That means that lack of write permission on a directory does not prevent a user from modifying a file within the directory, if the file's permissions allow it. It **does** prevent the user from renaming, moving, deleting or creating any file in the directory. This is because a directory is a really a file that contains a list of filenames, and so read and write permission control access to that list.

```
$ chmod u=rx test_dir
dr-xrwxr-x   2 eric     users
$ touch test_dir/bar
touch: test_dir/bar: Permission denied
$ mv test_dir/foo ./foo
mv: cannot move `test_dir/foo' to `./foo':
Permission denied
```

This property also works the other way. Since write permission allows the modification of directory entries, a user can move or rename a file **without permission to examine the contents**. This is a very good reason for paying attention to write access for important directories.

To demonstrate:

```
$ ls -l test_dir
-rw-rw-r--   2 eric  users  foo
$ chmod u=rwx test_dir
$ chmod u=rx test_dir/foo
$ cat .bashrc > test_dir/foo
bash: test_dir/foo: Permission denied
$ mv test_dir/foo ./foo
$ ls test_dir
(It's empty)
$ ls foo
foo (It's in our present directory.)
```

Execute permission for directories (also referred to as search permission) is also very important. Execute permission is necessary for **accessing** a directory.

```
$ chmod u=rwx test_dir
$ cp ~/.bashrc test_dir
(any text file will do)
$ chmod u=rw test_dir
$ cd test_dir
bash: test_dir: Permission denied
$ cat test_dir/.bashrc
cat: test_dir/.bashrc: Permission denied
```

This copy of .bashrc does not do us a lot of good. However, setting execute permission for directory and not setting read or write can come in handy.

```
$ chmod u=x test_dir
$ cat test_dir/.bashrc
(we see the contents of the file)
$ ls test_dir
ls: test_dir: Permission denied
```

A directory that has execute permission only can be used to “hide” files. Only users who know the exact file name and path can access them; this includes both data files and programs.
Conditional execute
Let's return to test_file to examine the **X** option.

```
$ chmod u=rw,g=r,o=r test_file
-rw-r--r--   1 eric     users
$ chmod o+X test_file
-rw-r--r--   1 eric     users
$ chmod u+x test_file
-rwxr--r--   1 eric     users
$ chmod o+X test_file
-rwxr--r-x   1 eric     users
```

In the first command, we see that we can set options for more than one class at a time by using a comma to separate the mode specifications. Here, we set the mode so that no user has execute permission. In the second command, we try to set execute permission for other with **X**. This fails, because **X** only works when one of the classes already has execute permissions. When we add execute permissions for owner, **X** sets executable permission for other.

The **s** option sets or removes set UID (SUID) and set GID (SGID) mode. These modes are very important in terms of UNIX/Linux security. When a file has SUID mode set, the process executing it has the effective rights of the file's owner for the duration of the program's execution.

For example, the program **dip** is used to create SLIP network connections. This requires root access, because creating a network interface device requires root access. Instead of forcing users to become root in order to use dip, which would require that the users know the root password, the dip program can belong to root and have the SUID mode set.

```
$ ls -l /usr/sbin/dip
-r-s--x---   1 root     dip
```

The **s** in the spot for user's execute field indicates the SUID mode is set. Another example of a use for the SUID mode is the passwd program, which allows users to modify the passwd (or shadow) file.

For security reasons, the SUID bit can affect only binary programs; it has no effect on shell scripts in Linux.

The SGID mode sets the group instead of the owner, and is set with (for example) **g+s**. It also has another purpose.

When a user creates a new file the group ownership defaults to the user's default group, which is the one listed in the passwd file. Sometimes users belong to more than one group and want to share files. The SGID mode can provide a convenient method for this. If the SGID mode bit is set for a directory, new files created in that directory will belong to that group, regardless of the creator's default group. If you belong to more than one group, try this. (You can check what groups you belong to with the **id** command. The default group is listed first, and you can use the chgrp command to change the group ownership of a file to another group you are a member of.)

```
$ mkdir test_dir
$ chgrp nondefault test_dir
$ chmod g+s test_dir
$ touch test_dir/foo
$ ls -l test_dir/foo
-rw-rw-r--   1 eric     nondefault
```

The SUID and SGID modes can be a security hole. However, when used carefully, they are very valuable tools and actually enhance system security by providing an alternative to distributing important passwords.
Make it simple
Specifying user classes can be used to simplify copying permissions.

```
$ chmod g=u test_file
-rwxrwxr-x   1 eric     users
```

This copied the permissions from user to group. All of the classes can be used on the right side of the **+**, **-** or **=** operators in this way.

```
$ chmod o-u test_file
-rwxrwx---   1 eric     users
```

This cleared all of the permissions that user has from other.

The last mode listed above is the **t** option, known as the “sticky bit”. This mode is actually supported on the command line for compatibility purposes with shell scripts from older operating systems. It is not needed for Linux. If an installation guide instructs you to use it, it actually does nothing.
Do your math

File access modes can also be set using octal notation. This syntax is built by adding the mode fields together. For each user class, the fields are calculated this way:

- 4 Read
- 2 Write
- 1 Execute

Full permissions for any class would be **7**, no permissions would be **0**.

```
$ chmod 754 test_file
-rwxr-xr-x   1 eric     users
```

The classes are passed to chmod in the same order ls displays them. The mode we set is broken down this way:

```
      Owner = 4 + 2 + 1 = 7
      Group = 4 + 1     = 5
      World = 4         = 4
```

Octal mode is convenient because other utilities, such as find, expect modes to be expressed this way.

In octal mode, SUID and SGID are set by specifying them in another column *before* the user mode. For SUID use 4, for SGID use 2, and use 6 for both:

```
$ chmod 4755 test_file
-rwsr-xr-x   1 eric     users
```

Power chmod
Chmod also provides a few command line options to simplify administrative tasks. For changing file permissions in directory trees use **-R**.

```
$ chmod -R g-w test_dir
```

This would remove write permission for group for all of the files in and below test_dir.

In order to control the output of messages from chmod use **-c**, **-v** and **-f**:

```
$ chmod -v 700 test_file
mode of test_file changed to 0700 (rwx------)
```

This option caused chmod to display how the permissions of test_file were set. The **-c** option causes chmod to display messages only when files are changed, and the **-f** option suppresses messages about files that can't be changed.

Chmod also provides a **--version** option to display the version and **--help** to see a short help message.
Summary

File permissions are an integral part of Linux. The same concepts also apply to other operating system objects such as semaphores, shared memory, and NIS+. This tutorial provides you with some of the basic knowledge necessary to protect your data and have more fun with your Linux system, and provides you with mental building blocks for learning more about Linux.

Eric Goebelbecker (eric@cnct.com (mailto:eric@cnct.com)) is a systems analyst for Reuters America, Inc. He supports clients (mostly financial institutions) who use market data retrieval and manipulation APIs in trading rooms and back office operations. In his spare time (about 15 minutes a week...), he reads about philosophy and hacks around with Linux.

You May Like

For Open-Source Software, the Developers Are All of Us (/content/open-source-software-developers-are-all-us)

Derek Zimmer (/users/derek-zimmer-0)

(/content/open-source-software-developers-are-all-us)

(/content/lotfi-ben-othmane-martin-gilje-jaatun-and-edgar-weippls-empirical-research-software-security)

Lotfi ben Othmane, Martin Gilje Jaatun and Edgar Weippl's Empirical Research for Software Security (CRC Press) (/content/lotfi-ben-othmane-martin-gilje-jaatun-and-edgar-weippls-empirical-research-software-security)

James Gray (/users/james-gray)

Heirloom Software: the Past as Adventure (/content/heirloom-software-past-adventure)

Eric S. Raymond (/users/eric-s-raymond)

(/content/heirloom-software-past-adventure)

SoftMaker FreeOffice (/content/softmaker-freeoffice)

James Gray (/users/james-gray)

(/content/softmaker-freeoffice)

Connect With Us

(https://youtube.com/linuxjournalonline) (https://www.facebook.com/linuxjournal/) (https://twitter.com/linuxjournal)