Filesystem Permissions

Permissions Overview

As we've discussed, Slackware Linux is a multi-user operating system. Because of this, its filesystems are mutli-user as well. This means that every file or directory has a set of permissions that can grant or deny privileges to different users. There are three basic permissions and three sets of permissions for each file. Let's take a look at an example file.

```
darkstar:~$ ls -l /bin/ls
-rwxr-xr-x 1 root root 81820 2007-06-08 21:12 /bin/ls
```

Recall from chapter 4 that ***ls*** *-l* lists the permissions for a file or directory along with the user and group that *"own"* the file. In this case, the permissions are rwxr-xr-x, the user is root and the group is also root. The permissions section, while grouped together, is really three separate pieces. The first set of three letters are the permissions granted to the user that owns the file. The second set of three are those granted to the group owner, and the final three are permissions for everyone else.

Table 10.1. Permissions of /bin/ls

Set	Listing	Meaning
Owner	rwx	The owner "root" may read, write, and execute
Group	r-x	The group "root" may read and execute
Others	r-x	Everyone else may read and execute

The permissions are pretty self explanatory of course, at least for files. Read, write, and execute allow you to read a file, write to it, or execute it. But what do these permissions mean for directories? Simply put, the read permissions grants the ability to list the directory's contents (say with ***ls***). The write permission grants the ability to create new files in the directory as well as delete the entire directory, even if you otherwise wouldn't be able to delete some of the other files inside it. The execute permission grants the ability to actually enter the directory (with the ***bash*** built-in command cd for example).

Let's look at the permissions on a directory now.

```
darkstar:~$ ls -ld /home/alan
drwxr-x--- 60 alan users 3040 2008-06-06 17:14 /home/alan/
```

Here we see the permissions on my home directory and its ownership. The directory is owned by the user alan and the group users. The user is granted all rights (rwx), the group is granted only read and execute permissions (r-x), and everyone else is prohibited from doing anything.

chmod, chown, and chgrp

So now that we know what permissions are, how do we change them? And for that matter, how do we assign user and group ownership? The answer is right here in this section.

The first tool we'll discuss is the useful ***chown*** (1) command. Using ***chown***, we can (you guessed it), change the ownership of a file or directory. ***chown*** is historically used only to change the user ownership, but can change the group ownership as well.

```
darkstar:~# ls -l /tmp/foo
total 0
-rw-r--r-- 1 alan users 0 2008-06-06 22:29 a
-rw-r--r-- 1 alan users 0 2008-06-06 22:29 b
darkstar:~# chown root /tmp/foo/a
darkstar:~# ls -l /tmp/foo
total 0
-rw-r--r-- 1 root users 0 2008-06-06 22:29 a
-rw-r--r-- 1 alan users 0 2008-06-06 22:29 b
```

By using a colon after the user account, you may also specify a new group account.

```
darkstar:~# chown root:root /tmp/foo/b
darkstar:~#  ls -l /tmp/foo
total 0
-rw-r--r-- 1 root users 0 2008-06-06 22:29 a
-rw-r--r-- 1 root root  0 2008-06-06 22:29 b
```

chown can also be used recursively to change the ownership of all files and directories below a target directory. The following command would change all the files under the directory `/tmp/foo` to have their ownership set to root:root.

```
darkstar:~# chown -R root:root /tmp/foo/b
```

Specifying a colon and a group name without a user name will simply change the group for a file and leave the user ownership intact.

```
darkstar:~# chown :wheel /tmp/foo/a
darkstar:~# ls -l /tmp/foo
ls -l /tmp/foo
total 0
-rw-r--r-- 1 root wheel 0 2008-06-06 22:29 a
-rw-r--r-- 1 root root  0 2008-06-06 22:29 b
```

The younger brother of ***chown*** is the slightly less useful ***chgrp***(1). This command works just like ***chown***, except it can only change the group ownership of a file. Since ***chown*** can already do this, why bother with ***chgrp***? The answer is simple. Many other operating systems use a different version of ***chown*** that cannot change the group ownership, so if you ever come across one of those, now you know how.

There's a reason we discussed changing ownership before changing permissions. The first is a much easier concept to grasp. The tool for changing permissions on a file or directory is ***chmod***(1). The syntax for it is nearly identical to that for ***chown***, but rather than specify a user or group, the administrator must specify either a set of octal permissions or a set of alphabetic permissions. Neither one is especially easy to grasp the first time. We'll begin with the less complicated octal permissions.

Octal permissions derive their name from being assigned by one of eight digits, namely the numbers

0 through 7. Each permissions is assigned a number that is a power of 2, and those numbers are added together to get the final permissions for one of the permission sets. If this sounds confusing, maybe this table will help.

Table 10.2. Octal Permissions

Permission	Meaning
Read	4
Write	2
Execute	1

By adding these values together, we can reach any number between 0 and 7 and specify all possible permission combinations. For example, to grant both read and write privileges while denying execute, we would use the number 6. The number 3 would grant write and execute permissions, but deny the ability to read the file. We must specify a number for each of the three sets when using octal permissions. It's not possible to specify only a set of user or group permissions this way for example.

```
darkstar:~# ls -l /tmp/foo/a
-rw-r--r-- 1 root root  0 2008-06-06 22:29 a
darkstar:~# chmod 750 /tmp/foo/a
darkstar:~# ls -l /tmp/foo/a
-rwxr-x--- 1 root root  0 2008-06-06 22:29 a
```

chmod can also use letter values along with + or − to grant or deny permissions. While this may be easier to remember, it's often easier to use the octal permissions.

Table 10.3. Alphabetic Permissions

Permission	Letter Value
Read	r
Write	w
Execute	x

Table 10.4. Alphabetic Users and Groups

Accounts Affected	Letter Value
User/Owner	u
Group	g
Others/World	o

To use the letter values with ***chmod***, you must specify which set to use them with, either *"u"* for user, *"g"* for group, and *"o"* for all others. You must also specify whether you are adding or removing permissions with the "+" and "-" signs. Multiple sets can be changed at once by separating each with a comma.

```
darkstar:/tmp/foo# ls -l
total 0
-rw-r--r-- 1 alan users 0 2008-06-06 23:37 a
-rw-r--r-- 1 alan users 0 2008-06-06 23:37 b
-rw-r--r-- 1 alan users 0 2008-06-06 23:37 c
-rw-r--r-- 1 alan users 0 2008-06-06 23:37 d
```

```
darkstar:/tmp/foo# chmod u+x a
darkstar:/tmp/foo# chmod g+w b
darkstar:/tmp/foo# chmod u+x,g+x,o-r c
darkstar:/tmp/foo# chmod u+rx-w,g+r,o-r d
darkstar:/tmp/foo# ls -l
-rwxr--r-- 1 alan users 0 2008-06-06 23:37 a*
-rw-rw-r-- 1 alan users 0 2008-06-06 23:37 b
-rwxr-x--- 1 alan users 0 2008-06-06 23:37 c*
-r-xr----- 1 alan users 0 2008-06-06 23:37 d*
```

Which you prefer to use is entirely up to you. There are places where one is better than the other, so a real Slacker will know both inside out.

SUID, SGID, and the "Sticky" Bit

We're not quite done with permissions just yet. There are three other *"special"* permissions in addition to those mentioned above. They are SUID, SGID, and the sticky bit. When a file has one or more of these permissions set, it behaves in special ways. The SUID and SGID permissions change the way an application is run, while the sticky bit restricts deletion of files. These permissions are applied with ***chmod*** like read, write, and execute, but with a twist.

SUID and SGID stand for *"Set User ID"* and *"Set Group ID"* respectively. When an application with one of these bits is set, the application runs with the user or group ownership permissions of that application regardless of what user actually executed it. Let's take a look at a common SUID application, the humble ***passwd*** and the files it modifies.

```
darkstar:~# ls -l /usr/bin/passwd \
  /etc/passwd \
  /etc/shadow
-rw-r--r-- 1 root root    1106 2008-06-03 22:23 /etc/passwd
-rw-r----- 1 root shadow   627 2008-06-03 22:22 /etc/shadow
-rws--x--x 1 root root   34844 2008-03-24 16:11 /usr/bin/passwd*
```

Notice the permissions on ***passwd***. Instead of an `x` in the user's execute slot, we have an `s`. This tells us that ***passwd*** is a SUID program, and when we run it, the process will run as the user *"root"* rather than as the user that actually executed it. The reason for this is readily apparent as soon as you look at the two files it modifies. Neither `/etc/passwd` nor `/etc/shadow` are writable by anyone other than root. Since users need to change their personal information, ***passwd*** must be run as root in order to modify those files.

So what about the sticky bit? The sticky bit restricts the ability to move or delete files and is only ever set on directories. Non-root users cannot move or delete any files under a directory with the sticky bit set unless they are the owner of that file. Normally anyone with write permission to the file can do this, but the sticky bit prevents it for anyone but the owner (and of course, root). Let's take a look at a common *"sticky"* directory.

```
darkstar:~# ls -ld /tmp
drwxrwxrwt 1 root root   34844 2008-03-24 16:11 /tmp
```

Naturally, being a directory for the storage of temporary files system wide, `/tmp` needs to be readable, writable, and executable by anyone and everyone. Since any user is likely to have a file or two stored here at any time, it only makes good sense to prevent other users from deleting those files, so the sticky bit has been set. You can see it by the presence of the `t` in place of the `x` in the world permissions section.

Table 10.5. SUID, SGID, and "Sticky" Permissions

Permission Type	Octal Value	Letter Value
SUID	4	s
SGID	2	s
Sticky	1	t

When using octal permissions, you must specify an additional leading octal value. For example, to recreate the permission on `/tmp`, we would use 1777. To recreate those permissions on `/usr/bin/passwd`, we would use 4711. Essentially, any time this leading fourth octet isn't specified, ***chmod*** assumes its value to be 0.

```
darkstar:~# chmod 1777 /tmp
darkstar:~# chmod 4711 /usr/bin/passwd
```

Using the alphabetic permission values is slightly different. Assuming the two files above have permissions of 0000 (no permissions at all), here is how we would set them.

```
darkstar:~# chmod ug+rwx,o+rwt /tmp
darkstar:~# chmod u+rws,go+x /usr/bin/passwd
```

Chapter Navigation

Previous Chapter: Users and Groups

Next Chapter: Working with Filesystems

Sources

- Original source: http://www.slackbook.org/beta
- Originally written by Alan Hicks, Chris Lumens, David Cantrell, Logan Johnson

slackbook, filesystem, permissions, suid, sgid, sticky bit, chmod, chown, chgrp