## 9– Permissions

Operating systems in the Unix tradition differ from those in the MS-DOS tradition in that they are not only *multitasking* systems, but also *multi-user* systems, as well.

What exactly does this mean? It means that more than one person can be using the com- puter at the same time. While a typical computer will likely have only one keyboard and monitor, it can still be used by more than one user. For example, if a computer is attached to a network or the Internet, remote users can log in via ssh (secure shell) and operate the computer. In fact, remote users can execute graphical applications and have the graphical output appear on a remote display. The X Window System supports this as part of its basic design.

The multiuser capability of Linux is not a recent "innovation," but rather a feature that is deeply embedded into the design of the operating system. Considering the environment in which Unix was created, this makes perfect sense. Years ago, before computers were "personal," they were large, expensive, and centralized. A typical university computer system, for example, consisted of a large central computer located in one building and terminals which were located throughout the campus, each connected to the large central computer. The computer would support many users at the same time.

In order to make this practical, a method had to be devised to protect the users from each other. After all, the actions of one user could not be allowed to crash the computer, nor could one user interfere with the files belonging to another user.

In this chapter we are going to look at this essential part of system security and introduce the following commands:

* id – Display user identity
* chmod – Change a file's mode
* umask – Set the default file permissions
* su – Run a shell as another user
* sudo – Execute a command as another user
* chown – Change a file's owner

9 – Permissions

* + chgrp – Change a file's group ownership
  + passwd – Change a user's password

### Owners, Group Members, And Everybody Else

When we were exploring the system back in Chapter 3, we may have encountered a prob- lem when trying to examine a file such as /etc/shadow:

[me@linuxbox ~]$ **file /etc/shadow**

/etc/shadow: regular file, no read permission [me@linuxbox ~]$ **less /etc/shadow**

/etc/shadow: Permission denied

The reason for this error message is that, as regular users, we do not have permission to read this file.

In the Unix security model, a user may *own* files and directories. When a user owns a file or directory, the user has control over its access. Users can, in turn, belong to a *group* consisting of one or more users who are given access to files and directories by their owners. In addition to granting access to a group, an owner may also grant some set of access rights to everybody, which in Unix terms is referred to as the *world*. To find out in- formation about your identity, use the id command:

[me@linuxbox ~]$ **id**

uid=500(me) gid=500(me) groups=500(me)

Let's look at the output. When user accounts are created, users are assigned a number called a *user ID* or *uid* which is then, for the sake of the humans, mapped to a username. The user is assigned a *primary group ID* or *gid* and may belong to additional groups. The above example is from a Fedora system. On other systems, such as Ubuntu, the output may look a little different:

[me@linuxbox ~]$ **id**

uid=1000(me) gid=1000(me) groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(v ideo),46(plugdev),108(lpadmin),114(admin),1000(me)

As we can see, the uid and gid numbers are different. This is simply because Fedora starts its numbering of regular user accounts at 500, while Ubuntu starts at 1000. We can also

see that the Ubuntu user belongs to a lot more groups. This has to do with the way Ubuntu manages privileges for system devices and services.

So where does this information come from? Like so many things in Linux, from a couple of text files. User accounts are defined in the /etc/passwd file and groups are defined in the /etc/group file. When user accounts and groups are created, these files are modified along with /etc/shadow which holds information about the user's password. For each user account, the /etc/passwd file defines the user (login) name, uid, gid, the account's real name, home directory, and login shell. If you examine the contents of

/etc/passwd and /etc/group, you will notice that besides the regular user ac- counts, there are accounts for the superuser (uid 0) and various other system users.

In the next chapter, when we cover processes, you will see that some of these other “users” are, in fact, quite busy.

While many Unix-like systems assign regular users to a common group such as “users”, modern Linux practice is to create a unique, single-member group with the same name as the user. This makes certain types of permission assignment easier.

### Reading, Writing, And Executing

Access rights to files and directories are defined in terms of read access, write access, and execution access. If we look at the output of the ls command, we can get some clue as to how this is implemented:

[me@linuxbox ~]$ **> foo.txt**

[me@linuxbox ~]$ **ls -l foo.txt**

-rw-rw-r-- 1 me me 0 2008-03-06 14:52 foo.txt

The first ten characters of the listing are the *file attributes*. The first of these characters is the *file type*. Here are the file types you are most likely to see (there are other, less com - mon types too):

*Table 9-1: File Types*

**Attribute File Type**

- A regular file.

d A directory.

l A symbolic link. Notice that with symbolic links, the remaining file attributes are always “rwxrwxrwx” and are dummy values. The real file attributes are those of the file the symbolic link points to.

Reading, Writing, And Executing

c A *character special file*. This file type refers to a device that handles data as a stream of bytes, such as a terminal or modem.

b A *block special file*. This file type refers to a device that handles data in blocks, such as a hard drive or CD-ROM drive.

The remaining nine characters of the file attributes, called the *file mode,* represent the read, write, and execute permissions for the file's owner, the file's group owner, and everybody else:

|  |  |  |
| --- | --- | --- |
| **Owner** | **Group** | **World** |
| rwx | rwx | rwx |

When set, the r, w, and x mode attributes have the following effect on files and directo- ries:

*Table 9-2: Permission Attributes*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Files** | **Directories** |
| r | Allows a file to be opened and | Allows a directory's contents to |
|  | read. | be listed if the execute attribute |
|  |  | is also set. |
| w | Allows a file to be written to or | Allows files within a directory |
|  | truncated, however this attribute | to be created, deleted, and |
|  | does not allow files to be | renamed if the execute attribute |
|  | renamed or deleted. The ability | is also set. |
|  | to delete or rename files is |  |
|  | determined by directory |  |
|  | attributes. |  |
| x | Allows a file to be treated as a | Allows a directory to be |
|  | program and executed. Program | entered, e.g., cd *directory*. |
|  | files written in scripting |  |
|  | languages must also be set as |  |
|  | readable to be executed. |  |

Here are some examples of file attribute settings:

*Table 9-3: Permission Attribute Examples*

**File Attributes Meaning**

-rwx------ A regular file that is readable, writable, and executable by the file's owner. No one else has any access.

-rw------- A regular file that is readable and writable by the file's owner.

No one else has any access.

-rw-r--r-- A regular file that is readable and writable by the file's owner. Members of the file's owner group may read the file. The file is world-readable.

-rwxr-xr-x A regular file that is readable, writable, and executable by the file's owner. The file may be read and executed by everybody else.

-rw-rw---- A regular file that is readable and writable by the file's owner and members of the file's group owner only.

lrwxrwxrwx A symbolic link. All symbolic links have “dummy”

permissions. The real permissions are kept with the actual file pointed to by the symbolic link.

drwxrwx--- A directory. The owner and the members of the owner group may enter the directory and, create, rename and remove files within the directory.

drwxr-x--- A directory. The owner may enter the directory and create, rename and delete files within the directory. Members of the owner group may enter the directory but cannot create, delete or rename files.

#### chmod – Change File Mode

To change the mode (permissions) of a file or directory, the chmod command is used. Be aware that only the file’s owner or the superuser can change the mode of a file or direc- tory. chmod supports two distinct ways of specifying mode changes: octal number repre- sentation, or symbolic representation. We will cover octal number representation first.

Reading, Writing, And Executing

### What The Heck Is Octal?

*Octal* (base 8), and its cousin, *hexadecimal* (base 16) are number systems often used to express numbers on computers. We humans, owing to the fact that we (or at least most of us) were born with ten fingers, count using a base 10 number sys- tem. Computers, on the other the other hand, were born with only one finger and thus do all all their counting in *binary* (base 2). Their number system only has two numerals, 0 and 1. So in binary, counting looks like this:

0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011...

In octal, counting is done with the numerals zero through seven, like so: 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21...

Hexadecimal counting uses the numerals zero through nine plus the letters “A” through “F”:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13...

While we can see the sense in binary (since computers only have one finger), what are octal and hexadecimal good for? The answer has to do with human con- venience. Many times, small portions of data are represented on computers as *bit patterns*. Take for example an RGB color. On most computer displays, each pixel is composed of three color components: eight bits of red, eight bits of green, and eight bits of blue. A lovely medium blue would be a 24 digit number: 010000110110111111001101

How would you like to read and write those kinds of numbers all day? I didn't think so. Here's where another number system would help. Each digit in a hexa- decimal number represents four digits in binary. In octal, each digit represents three binary digits. So our 24 digit medium blue could be condensed down to a six digit hexadecimal number:

436FCD

Since the digits in the hexadecimal number “line up” with the bits in the binary number we can see that the red component of our color is 43, the green 6F, and the blue CD.

These days, hexadecimal notation (often spoken as “hex”) is more common than octal, but as we shall soon see, octal's ability to express three bits of binary will be very useful...

With octal notation we use octal numbers to set the pattern of desired permissions. Since each digit in an octal number represents three binary digits, this maps nicely to the scheme used to store the file mode. This table shows what we mean:

*Table 9-4: File Modes In Binary And Octal*

**Octal Binary File Mode**

|  |  |  |
| --- | --- | --- |
| 0 | 000 | --- |
| 1 | 001 | --x |
| 2 | 010 | -w- |
| 3 | 011 | -wx |
| 4 | 100 | r-- |
| 5 | 101 | r-x |
| 6 | 110 | rw- |
| 7 | 111 | rwx |

By using three octal digits, we can set the file mode for the owner, group owner, and world:

[me@linuxbox ~]$ **> foo.txt**

[me@linuxbox ~]$ **ls -l foo.txt**

-rw-rw-r-- 1 me me 0 2008-03-06 14:52 foo.txt [me@linuxbox ~]$ **chmod 600 foo.txt**

[me@linuxbox ~]$ **ls -l foo.txt**

-rw------- 1 me me 0 2008-03-06 14:52 foo.txt

By passing the argument “600”, we were able to set the permissions of the owner to read and write while removing all permissions from the group owner and world. Though re- membering the octal to binary mapping may seem inconvenient, you will usually only have to use a few common ones: 7 (rwx), 6 (rw-), 5 (r-x), 4 (r--), and 0 (---).

chmod also supports a symbolic notation for specifying file modes. Symbolic notation is divided into three parts: who the change will affect, which operation will be performed, and what permission will be set. To specify who is affected, a combination of the charac- ters “u”, “g”, “o”, and “a” is used as follows:

*Table 9-5: chmod Symbolic Notation*

**Symbol Meaning**

u Short for “user” but means the file or directory owner.

g Group owner.

o Short for “others,” but means world.

a Short for “all.” The combination of “u”, “g”, and “o”.

Reading, Writing, And Executing

If no character is specified, “all” will be assumed. The operation may be a “+” indicating that a permission is to be added, a “-” indicating that a permission is to be taken away, or a “=” indicating that only the specified permissions are to be applied and that all others are to be removed.

Permissions are specified with the “r”, “w”, and “x” characters. Here are some examples of symbolic notation:

*Table 9-6: chmod Symbolic Notation Examples*

**Notation Meaning**

u+x Add execute permission for the owner.

u-x Remove execute permission from the owner.

+x Add execute permission for the owner, group, and world.

Equivalent to a+x.

o-rw Remove the read and write permission from anyone besides the owner and group owner.

go=rw Set the group owner and anyone besides the owner to have read and write permission. If either the group owner or world previously had execute permissions, they are removed.

u+x,go=rx Add execute permission for the owner and set the permissions for the group and others to read and execute. Multiple specifications may be separated by commas.

Some people prefer to use octal notation, some folks really like the symbolic. Symbolic notation does offer the advantage of allowing you to set a single attribute without disturb- ing any of the others.

Take a look at the chmod man page for more details and a list of options. A word of cau- tion regarding the “--recursive” option: it acts on both files and directories, so it's not as useful as one would hope since, we rarely want files and directories to have the same per- missions.

#### Setting File Mode With The GUI

Now that we have seen how the permissions on files and directories are set, we can better understand the permission dialogs in the GUI. In both Nautilus (GNOME) and Kon- queror (KDE), right-clicking a file or directory icon will expose a properties dialog. Here is an example from KDE 3.5:

Graphical user interface, application

Description automatically generated

*Figure 2: KDE 3.5 File Properties Dialog*

Here we can see the settings for the owner, group, and world. In KDE, clicking the “Ad- vanced Permissions” button brings up another dialog that allows you to set each of the mode attributes individually. Another victory for understanding brought to us by the com- mand line!

#### umask – Set Default Permissions

The umask command controls the default permissions given to a file when it is created. It uses octal notation to express a *mask* of bits to be removed from a file's mode at- tributes. Let's take a look:

[me@linuxbox ~]$ **rm -f foo.txt**

[me@linuxbox ~]$ **umask**

0002

[me@linuxbox ~]$ **> foo.txt**

[me@linuxbox ~]$ **ls -l foo.txt**

-rw-rw-r-- 1 me me 0 2008-03-06 14:53 foo.txt

We first removed any old copy of foo.txt to make sure we were starting fresh. Next, we ran the umask command without an argument to see the current value. It responded

with the value 0002 (the value 0022 is another common default value), which is the oc- tal representation of our mask. We next create a new instance of the file foo.txt and observe its permissions.

We can see that both the owner and group get read and write permission, while everyone else only gets read permission. The reason that world does not have write permission is because of the value of the mask. Let's repeat our example, this time setting the mask our- selves:

[me@linuxbox ~]$ **rm foo.txt** [me@linuxbox ~]$ **umask 0000** [me@linuxbox ~]$ **> foo.txt** [me@linuxbox ~]$ **ls -l foo.txt**

-rw-rw-rw- 1 me me 0 2008-03-06 14:58 foo.txt

When we set the mask to 0000 (effectively turning it off), we see that the file is now world writable. To understand how this works, we have to look at octal numbers again. If we take the mask and expand it into binary, and then compare it to the attributes we can see what happens:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Original file mode** | --- | rw- | rw- | rw- |
| **Mask** | 000 | 000 | 000 | 010 |
| **Result** | --- rw- rw- r-- | | | |

Ignore for the moment the leading zeros (we'll get to those in a minute) and observe that where the 1 appears in our mask, an attribute was removed — in this case, the world write permission. That's what the mask does. Everywhere a 1 appears in the binary value of the mask, an attribute is unset. If we look at a mask value of 0022, we can see what it does:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Original file mode** | --- | rw- | rw- | rw- |
| **Mask** | 000 | 000 | 010 | 010 |
| **Result** | --- rw- r-- r-- | | | |

Again, where a 1 appears in the binary value, the corresponding attribute is unset. Play with some values (try some sevens) to get used to how this works. When you're done, re- member to clean up:

[me@linuxbox ~]$ **rm foo.txt; umask 0002**

Most of the time you won't have to change the mask; the default provided by your distri- bution will be fine. In some high-security situations, however, you will want to control it.

### Some Special Permissions

Though we usually see an octal permission mask expressed as a three digit num- ber, it is more technically correct to express it in four digits. Why? Because, in ad- dition to read, write, and execute permission, there are some other, less used, per- mission settings.

The first of these is the *setuid bit* (octal 4000). When applied to an executable file, it sets the *effective user ID* from that of the real user (the user actually running the program) to that of the program's owner. Most often this is given to a few pro- grams owned by the superuser. When an ordinary user runs a program that is “*se- tuid root*” , the program runs with the effective privileges of the superuser. This allows the program to access files and directories that an ordinary user would nor- mally be prohibited from accessing. Clearly, because this raises security concerns, the number of setuid programs must be held to an absolute minimum.

The second less-used setting is the *setgid bit* (octal 2000) which, like the setuid bit, changes the *effective group ID* from the *real group ID* of the real user to that of the file owner. If the setgid bit is set on a directory, newly created files in the directory will be given the group ownership of the directory rather the group own- ership of the file's creator. This is useful in a shared directory when members of a common group need access to all the files in the directory, regardless of the file owner's primary group.

The third is called the *sticky bit* (octal 1000). This is a holdover from ancient Unix, where it was possible to mark an executable file as “not swappable.” On files, Linux ignores the sticky bit, but if applied to a directory, it prevents users from deleting or renaming files unless the user is either the owner of the directory, the owner of the file, or the superuser. This is often used to control access to a shared directory, such as /tmp.

Here are some examples of using chmod with symbolic notation to set these spe- cial permissions. First assigning setuid to a program:

chmod u+s *program*

Next, assigning setgid to a directory:

chmod g+s *dir*

Finally, assigning the sticky bit to a directory:

chmod +t *dir*

When viewing the output from ls, you can determine the special permissions. Here are some examples. First, a program that is setuid:

-rw**s**r-xr-x

A directory that has the setgid attribute:

drwxrw**s**r-x

A directory with the sticky bit set:

drwxrwxrw**t**

### Changing Identities

At various times, we may find it necessary to take on the identity of another user. Often we want to gain superuser privileges to carry out some administrative task, but it is also possible to “become” another regular user for such things as testing an account. There are three ways to take on an alternate identity:

1. Log out and log back in as the alternate user.
2. Use the su command.
3. Use the sudo command.

We will skip the first technique since we know how to do it and it lacks the convenience of the other two. From within our own shell session, the su command allows you to as- sume the identity of another user, and either start a new shell session with that user's IDs, or to issue a single command as that user. The sudo command allows an administrator to set up a configuration file called /etc/sudoers, and define specific commands that particular users are permitted to execute under an assumed identity. The choice of which command to use is largely determined by which Linux distribution you use. Your distri- bution probably includes both commands, but its configuration will favor either one or the other. We'll start with su.

#### su – Run A Shell With Substitute User And Group IDs

The su command is used to start a shell as another user. The command syntax looks like this:

**su [-[l]] [*user*]**

If the “-l” option is included, the resulting shell session is a *login shell* for the specified user. This means that the user's environment is loaded and the working directory is

changed to the user's home directory. This is usually what we want. If the user is not specified, the superuser is assumed. Notice that (strangely) the “-l” may be abbreviated “-”, which is how it is most often used. To start a shell for the superuser, we would do this:

[me@linuxbox ~]$ **su -** Password: [root@linuxbox ~]#

After entering the command, we are prompted for the superuser's password. If it is suc- cessfully entered, a new shell prompt appears indicating that this shell has superuser priv- ileges (the trailing “#” rather than a “$”) and the current working directory is now the home directory for the superuser (normally /root.) Once in the new shell, we can carry out commands as the superuser. When finished, enter “exit” to return to the previous shell:

[root@linuxbox ~]# **exit**

[me@linuxbox ~]$

It is also possible to execute a single command rather than starting a new interactive com- mand by using su this way:

su -c '*command*'

Using this form, a single command line is passed to the new shell for execution. It is im- portant to enclose the command in quotes, as we do not want expansion to occur in our shell, but rather in the new shell:

[me@linuxbox ~]$ **su -c 'ls -l /root/\*'**

Password:

-rw------- 1 root root

754 2007-08-11 03:19 /root/anaconda-ks.cfg

/root/Mail: total 0 [me@linuxbox ~]$

#### sudo – Execute A Command As Another User

The sudo command is like su in many ways, but has some important additional capabil- ities. The administrator can configure sudo to allow an ordinary user to execute com- mands as a different user (usually the superuser) in a very controlled way. In particular, a user may be restricted to one or more specific commands and no others. Another impor- tant difference is that the use of sudo does not require access to the superuser's pass- word. To authenticate using sudo, the user uses his/her own password. Let's say, for ex- ample, that sudo has been configured to allow us to run a fictitious backup program called “backup\_script”, which requires superuser privileges. With sudo it would be done like this:

[me@linuxbox ~]$ **sudo backup\_script**

Password:

System Backup Starting...

After entering the command, we are prompted for our password (not the superuser's) and once the authentication is complete, the specified command is carried out. One important difference between su and sudo is that sudo does not start a new shell, nor does it load another user's environment. This means that commands do not need to be quoted any dif- ferently than they would be without using sudo. Note that this behavior can be overrid- den by specifying various options. See the sudo man page for details.

To see what privileges are granted by sudo, use the “-l” option to list them:

[me@linuxbox ~]$ **sudo -l**

User me may run the following commands on this host: (ALL) ALL

**Ubuntu And sudo**

One of the recurrent problems for regular users is how to perform certain tasks that require superuser privileges. These tasks include installing and updating soft- ware, editing system configuration files, and accessing devices. In the Windows world, this is often done by giving users administrative privileges. This allows users to perform these tasks. However, it also enables programs executed by the

user to have the same abilities. This is desirable in most cases, but it also permits *malware* (malicious software) such as viruses to have free reign of the computer. In the Unix world, there has always been a larger division between regular users and administrators, owing to the multiuser heritage of Unix. The approach taken in Unix is to grant superuser privileges only when needed. To do this, the su and sudo commands are commonly used.

Up until a few of years ago, most Linux distributions relied on su for this pur- pose. su didn't require the configuration that sudo required, and having a root account is traditional in Unix. This introduced a problem. Users were tempted to operate as root unnecessarily. In fact, some users operated their systems as the root user exclusively, since it does away with all those annoying “permission de- nied” messages. This is how you reduce the security of a Linux system to that of a Windows system. Not a good idea.

When Ubuntu was introduced, its creators took a different tack. By default, Ubuntu disables logins to the root account (by failing to set a password for the ac- count), and instead uses sudo to grant superuser privileges. The initial user ac- count is granted full access to superuser privileges via sudo and may grant simi- lar powers to subsequent user accounts.

#### chown – Change File Owner And Group

The chown command is used to change the owner and group owner of a file or directory. Superuser privileges are required to use this command. The syntax of chown looks like this:

**chown [owner][:[group]] file...**

chown can change the file owner and/or the file group owner depending on the first ar- gument of the command. Here are some examples:

*Table 9-7: chown Argument Examples*

**Argument Results**

bob Changes the ownership of the file from its current owner to user

bob.

bob:users Changes the ownership of the file from its current owner to user

bob and changes the file group owner to group users.

:admins Changes the group owner to the group admins. The file owner is unchanged.

bob: Change the file owner from the current owner to user bob and changes the group owner to the login group of user bob.

Let's say that we have two users; janet, who has access to superuser privileges and tony, who does not. User janet wants to copy a file from her home directory to the home directory of user tony. Since user janet wants tony to be able to edit the file, janet changes the ownership of the copied file from janet to tony:

[janet@linuxbox ~]$ **sudo cp myfile.txt ~tony**

Password:

[janet@linuxbox ~]$ **sudo ls -l ~tony/myfile.txt**

-rw-r--r-- 1 root root 8031 2008-03-20 14:30 /home/tony/myfile.txt [janet@linuxbox ~]$ **sudo chown tony: ~tony/myfile.txt** [janet@linuxbox ~]$ **sudo ls -l ~tony/myfile.txt**

-rw-r--r-- 1 tony tony 8031 2008-03-20 14:30 /home/tony/myfile.txt

Here we see user janet copy the file from her directory to the home directory of user tony. Next, janet changes the ownership of the file from root (a result of using sudo) to tony. Using the trailing colon in the first argument, janet also changed the group ownership of the file to the login group of tony, which happens to be group tony.

Notice that after the first use of sudo, janet was not prompted for her password? This is because sudo, in most configurations, “trusts” you for several minutes until its timer runs out.

#### chgrp – Change Group Ownership

In older versions of Unix, the chown command only changed file ownership, not group ownership. For that purpose, a separate command, chgrp was used. It works much the same way as chown, except for being more limited.

### Exercising Our Privileges

Now that we have learned how this permissions thing works, it's time to show it off. We are going to demonstrate the solution to a common problem — setting up a shared direc- tory. Let's imagine that we have two users named “bill” and “karen.” They both have mu- sic CD collections and wish to set up a shared directory, where they will each store their

music files as Ogg Vorbis or MP3. User bill has access to superuser privileges via

sudo.

The first thing that needs to happen is creating a group that will have both bill and karen as members. Using the graphical user management tool, bill creates a group called music and adds users bill and karen to it:

Graphical user interface, application

Description automatically generated

*Figure 3: Creating A New Group With GNOME*

Next, bill creates the directory for the music files:

[bill@linuxbox ~]$ **sudo mkdir /usr/local/share/Music**

Password:

Since bill is manipulating files outside his home directory, superuser privileges are re- quired. After the directory is created, it has the following ownerships and permissions:

[bill@linuxbox ~]$ **ls -ld /usr/local/share/Music**

drwxr-xr-x 2 root root 4096 2008-03-21 18:05 /usr/local/share/Music

As we can see, the directory is owned by root and has 755 permissions. To make this directory sharable, bill needs to change the group ownership and the group permissions to allow writing:

Exercising Our Privileges

[bill@linuxbox ~]$ **sudo chown :music /usr/local/share/Music** [bill@linuxbox ~]$ **sudo chmod 775 /usr/local/share/Music** [bill@linuxbox ~]$ **ls -ld /usr/local/share/Music**

drwxrwxr-x 2 root music 4096 2008-03-21 18:05 /usr/local/share/Music

So what does this all mean? It means that we now have a directory,

/usr/local/share/Music that is owned by root and allows read and write ac- cess to group music. Group music has members bill and karen, thus bill and karen can create files in directory /usr/local/share/Music. Other users can list the contents of the directory but cannot create files there.

But we still have a problem. With the current permissions, files and directories created within the Music directory will have the normal permissions of the users bill and karen:

[bill@linuxbox ~]$ **> /usr/local/share/Music/test\_file**

[bill@linuxbox ~]$ **ls -l /usr/local/share/Music**

-rw-r--r-- 1 bill bill 0 2008-03-24 20:03 test\_file

Actually there are two problems. First, the default umask on this system is 0022 which prevents group members from writing files belonging to other members of the group. This would not be a problem if the shared directory only contained files, but since this di- rectory will store music, and music is usually organized in a hierarchy of artists and al- bums, members of the group will need the ability to create files and directories inside di- rectories created by other members. We need to change the umask used by bill and karen to 0002 instead.

Second, each file and directory created by one member will be set to the primary group of the user rather than the group music. This can be fixed by setting the setgid bit on the directory:

[bill@linuxbox ~]$ **sudo chmod g+s /usr/local/share/Music**

[bill@linuxbox ~]$ **ls -ld /usr/local/share/Music**

drwxrwsr-x 2 root music 4096 2008-03-24 20:03 /usr/local/share/Music

Now we test to see if the new permissions fix the problem. bill sets his umask to 0002, removes the previous test file, and creates a new test file and directory:

[bill@linuxbox ~]$ **umask 0002**

[bill@linuxbox ~]$ **rm /usr/local/share/Music/test\_file** [bill@linuxbox ~]$ **> /usr/local/share/Music/test\_file** [bill@linuxbox ~]$ **mkdir /usr/local/share/Music/test\_dir** [bill@linuxbox ~]$ **ls -l /usr/local/share/Music**

drwxrwsr-x 2 bill music 4096 2008-03-24 20:24 test\_dir

-rw-rw-r-- 1 bill music 0 2008-03-24 20:22 test\_file [bill@linuxbox ~]$

Both files and directories are now created with the correct permissions to allow all mem- bers of the group music to create files and directories inside the Music directory.

The one remaining issue is umask. The necessary setting only lasts until the end of ses- sion and must be reset. In Chapter 11, we'll look at making the change to umask perma- nent.

### Changing Your Password

The last topic we'll cover in this chapter is setting passwords for yourself (and for other users if you have access to superuser privileges.) To set or change a password, the passwd command is used. The command syntax looks like this:

**passwd [*user*]**

To change your password, just enter the passwd command. You will be prompted for your old password and your new password:

[me@linuxbox ~]$ **passwd** (current) UNIX password: New UNIX password:

The passwd command will try to enforce use of “strong” passwords. This means it will refuse to accept passwords that are too short, too similar to previous passwords, are dic- tionary words, or are too easily guessed:

[me@linuxbox ~]$ **passwd**

(current) UNIX password:

New UNIX password:

BAD PASSWORD: is too similar to the old one New UNIX password:

BAD PASSWORD: it is WAY too short

Changing Your Password

New UNIX password:

BAD PASSWORD: it is based on a dictionary word

If you have superuser privileges, you can specify a username as an argument to the passwd command to set the password for another user. Other options are available to the superuser to allow account locking, password expiration, etc. See the passwd man page for details.

### Summing Up

In this chapter we have seen how Unix-like systems such as Linux manage user permis- sions to allow the read, write, and execution access to files and directories. The basic ideas of this system of permissions date back to the early days of Unix and have stood up pretty well to the test of time. But the native permissions mechanism in Unix-like sys- tems lacks the fine granularity of more modern systems.

### Further Reading

* Wikipedia has a good article on malware: <http://en.wikipedia.org/wiki/Malware>

There are number of command line programs used to create and maintain users and groups. For more information, see the man pages for the following commands:

* adduser
* useradd
* groupadd

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer screen

Description automatically generated with medium confidence



A screenshot of a computer program

Description automatically generated with medium confidence