13.1 Introduction

In order to access or modify a file or directory, the correct *permissions* must be set. There are three different permissions that can be placed on a file or directory: read, write, and execute.

The manner in which these permissions apply differs for files and directories, as shown in the chart below:

Permission	Effects on File	Effects on Directory
read (`r`)	Allows for file contents to be read or copied.	Without execute permission on the directory, allows for a non-detailed listing of files. With execute permission, `ls -l` can provide a detailed listing.
write (`w`)	Allows for contents to be modified or overwritten. Allows for files to be added or removed from a directory.	For this permission to work, the directory must also have execute permission.
execute (`x`)	Allows for a file to be run as a process, although script files require read permission, as well.	Allows a user to change to the directory if parent directories have write permission as well.

When listing a file with the `ls -l` command, the output includes permission information:

```
sysadmin@localhost:~$ ls -l /bin/ls
-rwxr-xr-x. 1 root root 118644 Apr  4  2013 /bin/ls
```

Below is a refresher on the fields relevant to permissions.

File Type Field

```
-rwxr-xr-x. 1 root root 118644 Apr  4  2013 /bin/ls
```

The first character of this output indicates the type of a file. Recall if the first character is a `-`, as in the previous example, this is a regular file. If the character was a `d` it would be a directory.

Consider This

If the first character of the output of the `ls -l` command is an `l`, then the file is a symbolic link. A symbolic link file "points to" another file in the file system.

To access a file that a symbolic link points to, you need to have the appropriate permissions on both the symbolic link file and the file that it points to.

Permissions Field

```
-rwxr-xr-x. 1 root root 118644 Apr  4  2013 /bin/ls
```

After the file type character, the permissions are displayed. The permissions are broken into three sets of three characters:

- `rwx`r-xr-x: The first set for the user who owns the file.
- rwx`r-xr-x`:The second set for the group that owns the file.
- rwxr-x`r-x`: The last set for everyone else, which means "anyone who is not the user that owns the file or a member of the group that owns the file".

User Owner Field

```
-rwxr-xr-x. 1 root root 118644 Apr  4  2013 /bin/ls
```

After the link count, the file's user owner is displayed.

Group Owner Field

```
-rwxr-xr-x. 1 root root 118644 Apr  4  2013 /bin/ls
```

After the user owner field, the file group owner is displayed.

13.2 Changing Ownership

Changing the User Owner

Initially, the owner of a file is the user who creates it. The user owner can only be changed by a user with root privileges using the `chown` command. By executing `chown <newuser> pathname`, the root user can change the owner of the file `<pathname>` to the `<newuser>` account.

Changing the Group Owner

When a user creates a file or directory, their primary group will normally be the group owner. The `id` command reveals the current user's identity, current primary group and all group memberships.

To create a file or directory that will be owned by a group different from your current primary group, one option is to change your current primary group to another group you belong to by using the `newgrp` command. For example, to change the current primary group from `student`, to a secondary group called `circles`, execute: `newgrp circles`. After executing that command, any new files or directories created would be group owned by the `circles` group.

The `newgrp` command opens a new shell and assigns the primary group for that shell to the specified group. To go back to the default primary group, use the `exit` command to close the new shell that was started by the `newgrp`command.

Note: You must be a member of the group that you want to change to, additionally, administrators can password protect groups.

Another option is for the user owner of the file to change the group owner by using the `chgrp` command. For example, if you forgot to change your primary group to `circles` before you created `myfile`, then you can execute the `chgrp circles myfile` command.

The `chgrp` command does not change your current primary group, so when creating many files, it is more efficient to just use the `newgrp` command instead of executing the `chgrp` command for each file.

Note: Only the owner of the file and the root user can change the group ownership of a file.

13.3 Understanding Permissions

User and group owner information are important considerations when determining what permissions will be effective for a particular user. To determine which set applies to a particular user, first examine the user's identity. By using the `whoami`command or the `id` command, you can display your user identity.

To understand which permissions will apply to you, consider the following:

- `rwx`r-xr-x: If your current account is the user owner of the file, then the first set of the three permissions will apply and the other permissions have no effect.
- rwx`r-x`r-x: If your current account *is not* the user owner of the file and you *are* a member of the group that owns the file, then the group permissions will apply and the other permissions have no effect.
- rwxr-x`r-x`: If you are not the user who owns the file or a member of the group that owns the file, the third set of permissions applies to you. This last set of permissions is known as the permissions for "others"; it is sometimes referred to as the "world permissions".

For example, use the `ls -l /etc` command in the virtual terminal and examine the permissions displayed.

Answer the following questions:

- Who owns the files in the /etc directory?
- Can members of the root group modify (write to) these files?
- Can users who are not root, nor members of the root group modify these files?

Consider This

Understanding which permissions apply is an important skill set in Linux. For example, consider the following output of the `ls -l` command:

```
-r--rw-rwx. 1 bob staff 999 Apr 10 2013 /home/bob/test
```

In this scenario, the user `bob` ends up having less access to this file than members of the `staff` group or everyone else. The user `bob` only has the permissions of `r--`. It doesn't matter if `bob` is a member of the `staff`group; once user ownership has been established, only the user owner's permissions apply.

13.4 Changing Basic File Permissions

The `chmod` command is used to change the permissions of a file or directory. Only the root user or the user who owns the file is able to change the permissions of a file.

Consider This

Why is the command called `chmod` instead of `chperm`? Permissions used to be referred to as "modes of access", so the command `chmod` really means "change the modes of access".

There are two techniques of changing permissions with the `chmod` command: symbolic and octal (also called the numeric method). The symbolic method is good for changing one set of permissions at a time. The octal or numeric method requires knowledge of the octal value of

each of the permissions and requires all three sets of permissions (user, group, other) to be specified every time. When changing more than one permission set, the octal method is probably the better method.

The Symbolic Method

To use the symbolic method of `chmod` you will use the following symbols to represent which set of permissions you are changing:

Symbol	Meaning
`u`	user: the user who owns the file
`g`	group: the group who owns the file
`o`	others: people other than the user owner or member of the group owner
`a`	all: to refer to the user, group and others

Use the above symbols for who you are specifying along with an action symbol:

Symbol	Meaning
`+`	Add the permission, if necessary
`=`	Specify the exact permission
`-`	Remove the permission, if necessary

After an action symbol, specify one or more permissions (`r`=read, `w`=write and `x`=execute), then a space and the pathnames for the files to assign those permissions. To specify permissions for more than one set, use commas (and no spaces) between each set.

Examples

Add execute permission for the user owner:

```
sysadmin@localhost:~$  chmod u+x myscript
```

Remove write permission from the group owner:

```
sysadmin@localhost:~$  chmod g-w file
```

Assign others to have only the read permission, removes write permission from the group owner, and adds execute permission for the user owner:

```
sysadmin@localhost:~$  chmod o=r,g-w,u+x myscript
```

Assign everyone no permission:

```
sysadmin@localhost:~$  chmod a=- file
```

In the following example, a file is created with the `touch` command and then its permissions are modified using the symbolic method:

```
sysadmin@localhost:~$  touch sample
sysadmin@localhost:~$  ls -l sample
-rw-rw-r-- 1 sysadmin sysadmin 0 Oct 17 18:05 sample
sysadmin@localhost:~$  chmod g-w,o-r sample
sysadmin@localhost:~$  ls -l sample
-rw-r----- 1 sysadmin sysadmin 0 Oct 17 18:05 sample
```

The Octal Method

Using the octal method requires that the permissions for all three sets be specified. To do so, add together the octal value for the read, write and execute permission for each set:

Permission	Octal Value
read (`r`)	`4`
write (`w`)	`2`
execute (`x`)	`1`

For example, set the permissions of a file to `rwxr-xr-x`. Using the values found in the previous table, this would calculate as `7` for the user owner's permission, `5` for the group owner's permissions and `5` for others:

These numbers are easy to derive by just adding together the octal value for the permissions shown. So, for read, write and execute, add `4` + `2` + `1` to get `7`. Or, for read, not write and execute, add `4` + `0` + `1` to get `5`.

Examples

Change permissions to `rwxrw-r--`:

```
sysadmin@localhost:~$  chmod 764 myscript
```

Change permissions to `rw-r--r--`:

```
sysadmin@localhost:~$  chmod 644 myfile
```

Change permissions to `rwxr--r--`:

```
sysadmin@localhost:~$  chmod 744 myscript
```

Change permissions to `---------`:

```
sysadmin@localhost:~$  chmod 000 myfile
```

Considering the last example, what can be done with a file if there are no permissions for anyone on the file? The owner of the file can always use the `chmod` command at some point in the future to grant permissions on the file. Also, with write and execute permission `-wx` on the directory that contains this file, a user can also remove it with the `rm` command.

In the following example, the permissions of the sample file that was created previously are modified using the octal method:

```
sysadmin@localhost:~$  ls -l sample
-rw-r----- 1 sysadmin sysadmin 0 Oct 17 18:05 sample
sysadmin@localhost:~$  chmod 754 sample
sysadmin@localhost:~$  ls -l sample
-rwxr-xr-- 1 sysadmin sysadmin 0 Oct 17 18:05 sample
```

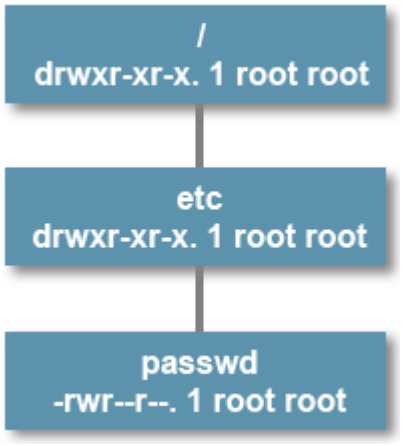
13.5 Permission Scenarios

Before discussing changing or setting permissions, a solid understanding of how permissions really work is required. Many beginning Linux users get hung up on the short descriptions of "read", "write" and "execute", but these words don't really define what a user can *do* with a file or directory.

To understand how permissions behave, several scenarios will be presented. While you review these scenarios, keep in mind the following table from a previous slide:

Permission	Effect of File	Effect on Directory
read (`r`)	Allows for file contents to be read or copied.	Without execute permission on the directory, allows for a non-detailed listing of files. With execute permission, `ls -l` can provide a detailed listing.
write (`w`)	Allows for contents to be modified or overwritten. Allows for files to be added or removed from a directory.	For this permission to work, the directory must also have execute permission.
execute (`x`)	Allows for a file to be run as a process, although script files require read permission, as well.	Allows a user to change to the directory if parent directories have execute permission as well

For each scenario below, a diagram is provided to describe the directory hierarchy. In this diagram, key permission information is provided for each file and directory. For example, consider the following:

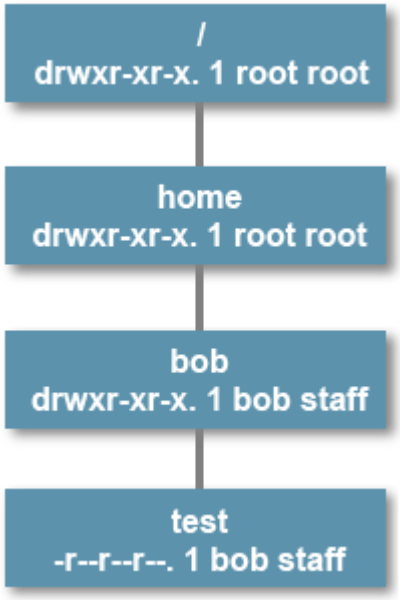


The first box describes the `/` directory. The permissions are `rwxr-xr-x`, the user owner is `root` and the group owner is `root`.

The second box describes the `etc` directory, which is a subdirectory under the `/` directory. The third box describes the `passwd` file, which is a file under the `etc` directory.

Scenario #1

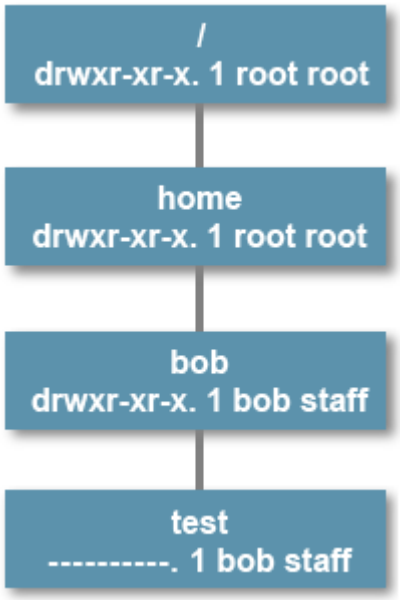
Question: Given the following diagram, what can the user `bob` do with the /home/bob/test file?



Answer: The user `bob` can view and copy the file because this user has the permissions of `r--` on the file.

Scenario #2

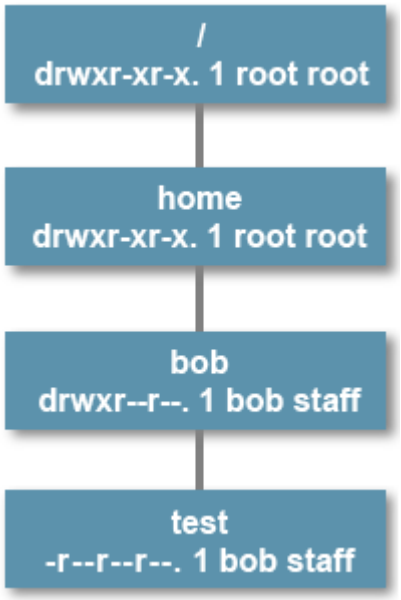
Question: Given the following diagram, can the user `bob` delete the `/home/bob/test` file?



Answer: Yes. While it may seem like the user `bob` shouldn't be able to delete this file because he has no permissions on the file itself, to delete a file a user needs to have write permission in the directory that the file is stored in. File permissions do not apply when deleting a file.

Scenario #3

Question: Given the following diagram, can the user `sue` view the contents of the `/home/bob/test` file?

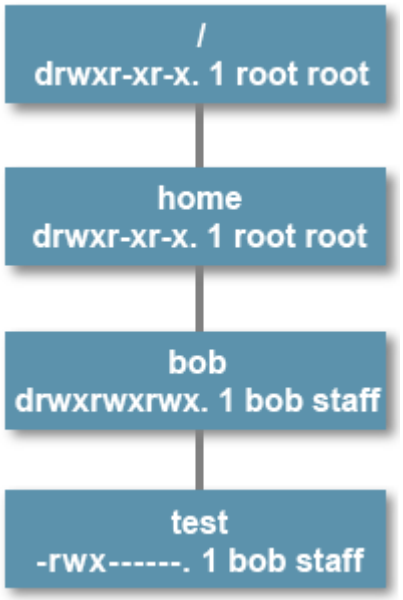


Answer: No. Initially it may seem that the user `sue` should be able to view the contents of the `/home/bob/test` file because all users have read permission on the file.

However, when checking permissions, it is important to check the permissions on the directories "above" the file in question. In this case, only the user `bob` has the ability to "get into" the `/home/bob` directory because only the user `bob` has execute permission on that directory. If you can't get into a directory, it doesn't matter what the permissions are on the files in that directory are set to.

Scenario #4

Question: Given the following diagram, who can delete the `/home/bob/test`file?



Answer: All users on the system can. Recall that to delete a file, a user needs write and execute permissions on the directory that the file is stored in. All users have write and execute permission on the `/home/bob` directory. This is a huge mistake because now all users can delete all files in bob's home directory.

13.6 Changing Advanced File Permissions

The following permissions are considered advanced because typically they are only set by the administrator (the root user) and they perform very specialized functions. They can be set using the `chmod` command, using either the symbolic or octal method.

Permission	Symbol	Octal Value	Purpose
setuid on a file	An `s` replaces the `x` for the user owner permissions Set with `u+s`	`4000`	Causes an executable file to execute under user owner identity, instead of the user running the command.
setgid on a file	An `s` replaces the `x` for the group owner permissions Set with `g+s`	`2000`	Causes an executable file to execute under group owner identity, instead of the user running the command.
setgid on a directory	An `s` replaces the `x` for the group owner permissions Set with `g+s`	`2000`	Causes new files and directories that are created inside to be owned by the group that owns the directory.
sticky on a directory	A `t` replaces the `x` for the others permissions Set with `o+t`	`1000`	Causes files inside directory to be able to be removed only by the user owner, or the root user.

Note: A leading `0`, such as `0755`, will remove all special permissions from a file or directory.

13.6.1 setuid Permission

The setuid permission is used on executable files to allow users who are not the root user to execute those files as if they were the root user.

For example, the `passwd` command has the setuid permission. The `passwd`command modifies the `/etc/shadow` file in order to update the value for the user's password. That file is not normally modifiable by an ordinary user; in fact, ordinary users normally have no permissions on the file.

Since the `/usr/bin/passwd` command is owned by the root user and has setuid permission, it executes with a "duel personality", which allows it to access files either as the person who is running the command or as the root user. When the `passwd` command attempts to update the `/etc/shadow` file, it uses the credentials of the root user to modify the file (the term *credentials* is akin to "authority").

The following demonstrates what this setuid executable file looks like when listed with `ls -l`:

```
sysadmin@localhost:~$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 47032 Feb 17  2014 /usr/bin/passwd
```

Notice from the listing, the fourth character is an `s`, where there would normally be an `x` if this file was just executable. When this character is a lowercase `s`, it indicates that both the setuid and execute permissions are set. An uppercase `S`in the fourth character position means that the file does *not* have the execute permission, only the setuid permission. Without the execute permission for the user owner, the setuid permission is ineffective.

13.6.2 setgid Permission

On a File

The setgid permission on a file works very similarly to the setuid permission, except that instead of executing as the user who owns the file, setgid permission will execute as the group that owns the file. The following displays the setgid executable file `/usr/bin/wall` when listed with the `ls -l` command:

```
sysadmin@localhost:~$ ls -l /usr/bin/wall
-rwxr-sr-x 1 root tty 19024 Jun  3 20:54 /usr/bin/wall
```

Notice from the listing, the seventh character is an `s` where there would normally be an `x` for the group execute permission. The lowercase `s` indicates that this file has both the setgid and execute permission set. An `S` instead of the `s`means the file lacks execute permission for the group. Without execute permission for the group, the setgid permission will be ineffective.

Consider This

The way that the Linux kernel restricts both setuid and setgid executables is stricter than some UNIX implementations. While Linux doesn't prevent users from setting these permissions on script files, the Linux kernel will not execute scripts with setuid or setgid permissions.

In order to effectively set these permissions, the Linux kernel will only honor them on executable files that are in a binary format. This increases the security of the system.

On a Directory

In most cases, once an installation of a Linux distribution is complete, the files that require setuid and setgid permission should have those permissions automatically set. While some of the setuid and setgid permissions on files may be removed for security reasons, very seldom are new setuid or setgid files created. On the other hand, it is fairly common for administrators to add setgid to *directories*.

Using setgid on a directory can be extremely helpful for users trying to share a directory of files with other users who are in different groups. To understand why, consider the following scenario.

Users from different groups in your organization have asked the administrator to set up a directory in which they can share files. For this example, there will be three users: `joe` who is a member of the `staff` group, `maya` who is a member of the `payroll` group and `steve` who is a member of the `acct` group. To accomplish this, the administrator takes the following steps:

- **Step #1:** Create a new group for the three users:

```
root@localhost:~# groupadd -r common
```

- **Step #2:** Add the users to the group:

```
root@localhost:~# usermod -aG common joe
root@localhost:~# usermod -aG common maya
```

- **Step #3:** Create a directory and make common the group owner:

```
root@localhost:~# mkdir /shared
root@localhost:~# chgrp common /shared
```

- **Step #4:** Make the directory writable for the group:

```
root@localhost:~# chmod 770 /shared
```

This solution is almost perfect: the users `joe`, `maya` and `steve` can now access the `/shared` directory and add new files. However, there is a potential problem: for example, when the user `joe` makes a new file in the `/shared`directory, it will likely end up looking like the following when displayed with the `ls -l` command:

```
-rw-rw----. 2 joe staff 8987 Jan 10 09:08 data.txt
```

The problem with this is that neither `maya` or `steve` are members of the `staff`group. As a result, their permissions are `---`, which means they can't access the contents of this file.

The `joe` user could have used the `newgrp` command to switch to the `common`group before creating the file. Or, after creating the file, the `joe` user could have used the `chgrp` command to change the group ownership to the `common` group. However, users won't always remember to run these commands; some may not even know that these commands exist.

Instead of having to train users how to use the `newgrp` and `chgrp` commands, the administrator can set up a directory with setgid permission. If a directory is setgid, then all new files created or copied into the directory will *automatically* be owned by the group that owns the directory. This means users don't have to use the `newgrp` or `chgrp` commands because the group ownership will be managed automatically.

So, a better solution to this scenario would be to use the following command for Step #4:

```
root@localhost:~# chmod 2775 /shared
```

Listing the details of the directory after running the previous command would result in the following output:

```
drwxrwsr-x. 2 root common 4096 Jan 10 09:08 /shared
```

After completing these steps, the `joe`, `maya` and `steve` users would now be able to easily create files in the `/shared` directory that would automatically be owned by the `common` group.

13.6.3 sticky bit Permission

The final advanced permission to discuss is the sticky permission, which is sometimes called the "sticky bit". Setting this permission can be important to prevent users from deleting other user's files.

Recall that to be able to delete a file, only the write and execute permission on the directory are necessary. The write and execute permission is also necessary to add files to a directory. So, if an administrator wants to make a directory where any user can add a file, the required permissions mean that user can also delete any other user's file in that directory.

The sticky permission on a directory modifies the write permission on a directory so that only specific users can delete a file within the directory:

- The user who owns the file
- The user who owns the directory that has the sticky bit set (typically this is the root user)
- The root user

On a typical Linux installation there will normally be two directories that have the sticky permission set by default: the `/tmp` and `/var/tmp` directories. Besides user home directories, these two directories are the only locations that regular users have write permission by default.

As their name suggests, these directories are intended to be used for temporary files. Any user can copy files to the `/tmp` or `/var/tmp` directory to share with others.

Since the /tmp or /var/tmp directories have the sticky permission, you don't have to worry about users deleting the shared files. However, there is a reason that these directories are considered temporary: these directories are automatically purged of their contents on a regular schedule. If you put a file in them, it will be removed automatically at some point in the future.

When a directory has the sticky permission, a `t` will replace the `x` in the set of permissions for others. For example, the following shows the output of the `ls -ld /tmp` command:

```
sysadmin@localhost:~$ ls -ld /tmp
drwxrwxrwt 1 root root 0 Oct 17 19:17 /tmp
```

The `/tmp` directory has an octal permission mode of `1777`, or full permissions for everyone plus the sticky permission on the directory. When the “others” permission set includes the execute permission, then the `t` will be in lowercase. If there is an uppercase `T`, it means that the execute permission is not present for others. This does not mean that the directory does not have an effective sticky permission, however it does mean that the users who are affected by the "others" permission can't change to this directory or create files in the directory.

There are cases in which users may not want to have execute for others, but still have the sticky bit permission on a directory. For example, when creating shared directories with the setgid permission. Consider the following command:

```
root@localhost:~# chmod 3770 /shared
```

Notice that the special permissions of setgid and stickybit can be added together, the `2000` permission plus the `1000` permission gives you `3000` for the special permissions. Add this to the basic permissions for the user, group and others; The owner and group have full access and others get none. Listing the directory after making the changes with the `ls -ld /shared` command results in the following output:

```
drwxrws--T. 2 root common 4096 Jan 10 09:08 /shared
```

The uppercase `T` in the execute permissions position for others indicates there is no execute permission for others. However, since multiple users of the group still have access, the sticky permission is effective for the `common` group.

Consider This

The `stat` command can displays permissions both in symbolic and octal notation, as well as user owner and group owner information. For example, look at the first line that starts with `Access`:

```
sysadmin@localhost:~$ stat /tmp
  File: '/tmp'
  Size: 0               Blocks: 0          IO Block: 4096   directory
Device: 3ah/58d Inode: 1493        Links: 1
Access: (1777/drwxrwxrwt)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2014-09-18 23:37:24.364779732 +0000
Modify: 2014-10-17 19:17:01.430467987 +0000
Change: 2014-10-17 19:17:01.430467987 +0000
 Birth: -
```

13.7 Default File Permissions

Unlike other operating systems where the permissions on a new directory or file may be inherited from the parent directory, Linux sets the default permissions on these new objects based upon the value of the creator's UMASK setting. The `umask` command is used to both set the UMASK value and display it.

The `umask` command is automatically executed when a shell is started. To have a persistent setting for UMASK, a custom `umask` command can be added to the `~/.bashrc` file.

To customize the UMASK value, it is important to understand what the UMASK value does. The value of the UMASK value only affects the permissions placed on new files and directories at the time they are created. It only affects the basic permissions for the user owner, the group owner and others. The UMASK value does not affect the special advanced permissions of setuid, setgid or sticky bit.

The UMASK is an octal value based upon the same values that you saw earlier in this section:

Permission	Octal Value
read	`4`
write	`2`
execute	`1`
none	`0`

When using the `chmod` command with octal values, add together the values for the user, the group and others. However, the UMASK octal value used to specify permissions to be *removed*. In other words, the octal value set for the UMASK is subtracted from the maximum possible permission to determine the permissions that are set when a file or directory is created.

Understanding UMASK for Files

By default, the maximum permissions that will be placed on a brand new file are `rw-rw-rw-`, which can be represented octal as `666`. The execute bit is turned off for security reasons. The UMASK value can be used to specify which of these default permissions to remove for new files. Three octal values will be provided: the value of the permissions to remove for the user owner, the group owner and others.

For example, to set a UMASK value for new files that would result in full permissions for the owner (result: `rw-`), remove or *mask* write permissions for the group owner (result: `r--`) and remove all permissions from others (result: `---`), calculate the UMASK value as follows:

- The first digit for the user owner would be a `0`, which would not mask any of the default permissions.
- The second digit would be a `2`, which would mask only the write permission.
- The third digit would be a `6`, which would mask the read and write permissions.

As a result, the UMASK value `026` would result in new files having the permissions of `rw-r-----`.

Another example: To set a UMASK value for new files that would remove write permissions for the owner (result: `r--`) and remove read and write permissions for the group and others (result: `---`), calculate the UMASK value as follows:

- The first digit for the user owner would be a `2`, which would mask only the write permission.
- The second digit would be a `6`, which would mask the read and write permissions.
- The third digit would be a `6`, which would mask the read and write permissions.

As a result, the UMASK value `266` would result in new files having the permissions of `r--------`.

Understanding UMASK for Directories

The way the UMASK is applied to regular files is different from directory files. For security reasons, regular files are not allowed to be executable at the time they are created. It would be dangerous to allow for a file to be able to run as a process without a user explicitly assigning the execute permission. So, regardless of whether you include the execute permission in the UMASK value, it will not apply to regular files.

For directories, the execute permission is critical to properly access the directory. Without the execute permission, you cannot navigate to a directory and the write permission is not functional. Essentially, directories are not very useful at all without execute permission, so new directories are allowed to be executable by default. The default permissions for new directories is `rwxrwxrwx` or `777`.

For example, if you wanted a UMASK value for new directories that would result in full permissions for the user owner (result: `rwx`), remove or mask write permissions for the group owner (result: `r-x`) and remove all permissions from others (result: `---`), then you could calculate the UMASK value as follows:

- The first digit for the user owner would be a `0`, which would not mask any of the default permissions.
- The second digit would be a `2`, which would mask only the write permission.
- The third digit would be a `7`, which would mask the read and write permissions.

As a result, the UMASK value `027` would result in new directories having the permissions of `r-xr-x---`.

Very Important: While the UMASK value effects the permissions for new files and directories differently (because of different maximum permissions), there is not a separate UMASK value for files and directories; the single UMASK value applies to both, as you can see from the following table of commonly used UMASK values:

umask	File Permissions	Directory Permissions	Description
`002`	`664` or `rw-rw-r--`	`775` or `rwxrwxr-x`	Default for ordinary users
`022`	`644` or `rw-r--r--`	`755` or `rwxr-xr-x`	Default for root user
`007`	`660` or `rw-rw----`	`770` or `rwxrwx---`	No access for others
`077`	`600` or `rw-------`	`700` or `rwx------`	Private to user

The following commands will display the current UMASK value, sets it to a different value and displays the new value:

```
sysadmin@localhost:~$ umask
0002
sysadmin@localhost:~$ umask 027
sysadmin@localhost:~$ umask
```

```
0027
```

Note: The initial `0` in the UMASK value is for special permissions (setuid, setgid and sticky bit). Since those are never set by default, the initial `0` is not necessary when setting the UMASK value.

Chapter 13: File Permissions and Ownership

This chapter will cover the following exam objectives:

104.5: Manage file permissions and ownership

Weight: 3

Description: Candidates should be able to control file access through the proper use of permissions and ownerships.

Key Knowledge Areas:

- Manage access permissions on regular and special files as well as directories
  Section 13.4 | Section 13.6
- Use access modes such as suid, sgid and the sticky bit to maintain security
  Section 13.6.1 | Section 13.6.2 | Section 13.6.3
- Know how to change the file creation mask
  Section 13.7
- Use the group field to grant file access to group members
  Section 13.4 | Section 13.6.2

Chapter 13: File Permissions and Ownership

chgrp

Command that is used to change the primary group of a file. Essentially it changes what group is the owner of the FILE.
Section 13.1

chmod

Command that is used to change the mode bits of a FILE. The chmod utility can be used to change the files permissions. For example setting the read, write, and execute bits.
Section 13.4

chown

Command that is used to change the ownership of a FILE. The chown utility can also be used to change the primary group of a FILE as well.
Section 13.1

umask

Command that sets the calling process's file mode creation mask. The umask utility will set the default permissions for FILEs when they are created.
Section 13.6