Experiment-13

Aim: Local Security Principles: Understanding Linux Security, Understand the Uses of root, difference b/w su and sudo command, assigning sudo privileges to new user. Working with passwords, perm issions modification using chmod. Chown, chgrp.

Understanding the root Account

Root is the most privileged account on a Linux/UNIX system. This account has the ability to carry out all facets of system administration, including adding accounts, changing user passwords, examining log files, installing software, etc. Utmost care must be taken when using this account. It has no security restrictions imposed upon it.

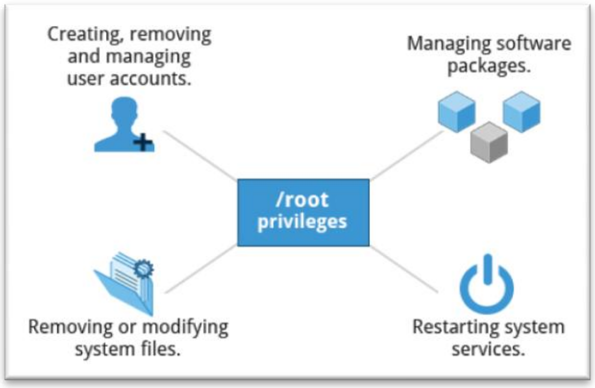
When you are signed in as, or acting as root**,** the shell prompt displays '#' This convention is intended to serve as a warning to you of the absolute power of this account.

Operations that Require root Privileges

Root privileges are required to perform operations such as:

- Creating, removing and managing user accounts.
- Managing software packages.
- Removing or modifying system files.
- Restarting system services.

Regular account users of Linux distributions may be allowed to install software packages, update some settings, and apply various kinds of changes to the system. However, root privilege is required for performing administration tasks such as restarting services, manually installing packages and managing parts of the filesystem that are outside the normal user's directories.


Creating, removing and managing user accounts.
Managing software packages.
/root privileges
Removing or modifying system files.
Restarting system services.

Comparing sudo and su

In Linux you can use either su or sudo to temporarily grant root access to a normal user; these methods are actually quite different. Listed below are the differences between the two commands.

Su	sudo
When elevating privilege, you need to enter the root password. Giving the root password to a normal user should never, ever be done.	When elevating privilege, you need to enter the user's password and not the root password.
Once a user elevates to the root account using su, the user can do anything that the root user can do for as long as the user wants, without being asked again for a password.	Offers more features and is considered more secure and more configurable. Exactly what the user is allowed to do can be precisely controlled and limited. By default the user will either always have to keep giving their password to do further operations with sudo, or can avoid doing so for a configurable time interval.
The command has limited logging features.	The command has detailed logging features.

The su Command

The su command allows you to **s**wtich **u**ser and run your commands as some other user under their user ID. When you run su without any arguments, it will try to open up a root shell by default and will therefore prompt you for the root password to proceed. After entering the root password, you are now the root user and anything you run during this session will be run as root.

```
[user1@centos7 ~]$ whoami
user1
[user1@centos7 ~]$ su
Password:
```

```
[root@centos7 user1]# whoami
root
```

Alternatively you can specify the user that you want to change to, which generally requires their password unless you are root.

```
[root@centos7 ~]# su - user1
Last login: Tue Aug 30 11:30:32 AEST 2016 on pts/0
[user1@centos7 ~]$ whoami
user1
[user1@centos7 ~]$ su - user2
Password:
Last login: Tue Aug 30 11:29:59 AEST 2016 on pts/0
```

As shown if user1 wants to switch to user2 they need the password, however root can switch to any other user without providing the password.

While it is not required that the **'-'** be specified, it is recommended for an interactive shell. As shown below if we switch user without specifying '-' the path or current working directory of /root are not changed, which may cause problems when user1 goes to run commands.

```
[root@centos7 ~]# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
[root@centos7 ~]# su user1
[user1@centos7 root]$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
[user1@centos7 root]$ exit
exit
[root@centos7 ~]# su - user1
Last login: Tue Aug 30 11:32:02 AEST 2016 on pts/0
```

```
[user1@centos7 ~]$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/user1/.local/bin:/home/user1/bin
```

Essentially you almost always want to make use of the '-'when using su.

We don't just have to enter a shell of the new user, we can optionally execute commands as that user with the -c flag.

```
[user1@centos7 ~]$ su -c whoami
Password:
root
```

Now that we understand the su command, let's see what sudo has to offer.

The sudo Command

The super **u**ser **do**, or sudo command on the other hand instead allows you to run a command as root from your current user. By default this will require you to provide your password again as a security measure.

The non root user account requires sudo privileges to do this, and this is normally setup by either adding the user or group to the /etc/sudoers file, or by adding the user to the wheel group.

```
[root@centos7 ~]# usermod -aG wheel user1
[root@centos7 ~]# id user1
uid=1000(user1) gid=1000(user1) groups=1000(user1),10(wheel)
[root@centos7 ~]# su - user1
Last login: Tue Aug 30 11:42:46 AEST 2016 on pts/0
[user1@centos7 ~]$ sudo whoami
```

```
We trust you have received the usual lecture from the local System Administrator. It usually boils down to these three things:

    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great power comes great responsibility.

[sudo] password for user1:
root
```

In this example the root user adds user1 to the wheel group, which is defined in the /etc/sudoers configuration file to provide root privileges via the sudo command. From there we run the whoami command as root with sudo, and after providing the password of user1 we are advised that we are root. Note that we are not dropped into a root shell after this.

After entering the user's password for the first time for sudo, subsequent sudo usage does not require the password, as shown below. Here we also see that user1 does not have permission to list contents of /root by itself, however it works correctly with sudo.

```
[user1@centos7 ~]$ ls -l /root
ls: cannot open directory /root: Permission denied
[user1@centos7 ~]$ sudo ls -l /root
total 3244
-rw-------. 1 root root     984 Aug 29 14:21 anaconda-ks.cfg
```

We can also use sudo in combination with the su command to enter an interactive root shell, rather than entering every command with the sudo prefix.

```
[user1@centos7 ~]$ sudo su -
Last login: Tue Aug 30 11:49:01 AEST 2016 on pts/0
```

```
[root@centos7 ~]# whoami
root
```

Similarly, we can get a shell with the -i flag.

```
[user1@centos7 ~]$ sudo -i
[sudo] password for user1:
[root@centos7 ~]# whoami
root
```

Differences Between su and sudo

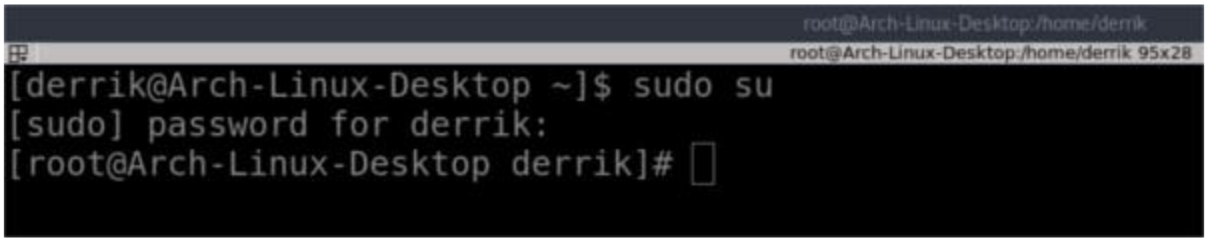
With the explanations out of the way for each command hopefully you can already see the key differences between the two.

They are indeed quite similar in some aspects, the ‘su’ command is basically equivalent to ‘sudo -i’, while the ‘sudo’ command is basically equivalent to ‘su -c’.

A major key difference is who gets the root password. If a user wishes to su to root then they require the password of the root account. If instead the user is executing a command with sudo, they only need their own password and sudo privileges. Therefore if you have multiple users that require root privileges on a system, providing sudo access is considered to be more secure as we can audit commands that have been executed by specific users without sharing the root user’s password with other people.

By default a non root user could use sudo privileges to change the root password, however the /etc/sudoers file can be used to only grant root access to specific commands that the user needs to run as root rather than being able to run anything as root. With sudo we can define security policy, allowing one group of users to perform only a specific subset of clearly defined commands as the root user.

sudo su

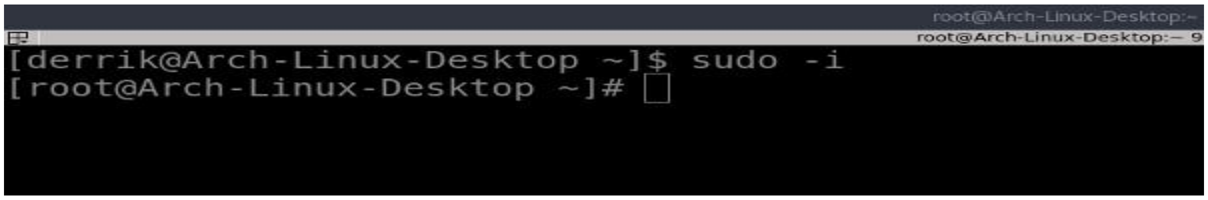


```
[derrik@Arch-Linux-Desktop ~]$ sudo su
[sudo] password for derrik:
[root@Arch-Linux-Desktop derrik]#
```

This command is essentially the same as just running su in the shell. Instead of telling the system to “switch users” directly, you’re telling it to run the “su” command as root. When sudo su is run, “.profile,” “.bashrc” and “/etc/profile” will be started, much like running su (or su root). This is because if any command is run with sudo in front of it, it’s a command that is given root privileges.

Though there isn’t very much difference from “su,” sudo su is still a very useful command for one important reason: When a user is running “su” to gain root access on a system, they must know the root password. The way root is given with sudo su is by requesting the current user’s password. This makes it possible to gain root without the root password which increases security.

sudo -i

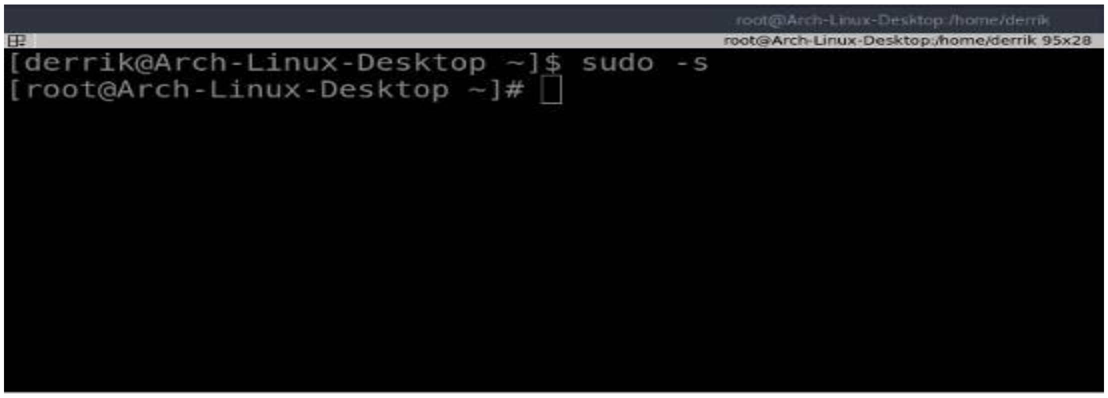


```
[derrik@Arch-Linux-Desktop ~]$ sudo -i
[root@Arch-Linux-Desktop ~]#
```

Using sudo –i is virtually the same as the sudo su command. Users can gain root by “sudo” and not by switching to the root user. Much like sudo su, the –i flag allows a user to get a root environment without having to know the root account password. Sudo -i is also very similar to using sudo su in that it’ll read all of the environmental files (.profile, etc.) and set the environment inside the shell with it.

Where it differs from “sudo su” is that sudo -i is a much cleaner way of gaining root and a root environment without directly interacting with the root user. How? With sudo su you’re using more than one root setuid commands. This fact makes it much more challenging to figure out what environmental variables will be kept and which ones will be changed (when swamping to the root environment). This is not true with sudo -i, and it is because of this most people view it as the preferred method to gain root without logging in directly.

sudo -s



```
root@Arch-Linux-Desktop:/home/derrik
root@Arch-Linux-Desktop:/home/derrik 95x28
[derrik@Arch-Linux-Desktop ~]$ sudo -s
[root@Arch-Linux-Desktop ~]# 
```

The -s switch for "sudo" command reads the $SHELL variable of the current user executing commands. This command works as if the user is running sudo /bin/bash. Sudo -s is a "non-login" style shell. This means that unlike a command like sudo -i or sudo su, the system will not read any environmental files. This means that when a user tells the shell to run sudo -s, it gains root but will not change the user or the user environment. Your home will not be the root home, etc.

This command is best used when the user doesn't want to touch root at all and just wants a root shell for easy command execution. Other commands talked about above gain root access, but touch root environmental files, and allow users fuller access to root (which can be a security issue).

Conclusion: What command should I use?

Each command has its use-case. The important thing here is to understand what each command does and when to use them. As it stands, `sudo -i` is the most practical, clean way to gain a root environment. On the other hand, those using `sudo -s` will find they can gain a root shell without the ability to touch the root environment, something that has added security benefits.

Permissions

The Linux operating system (and likewise, Linux) differs from other computing environments in that it is not only a *multitasking* system but it is also a *multi-user* system as well.

What exactly does this mean? It means that more than one user can be operating the computer at the same time. While your computer will only have one keyboard and monitor, it can still be used by more than one user. The multi-user capability of Linux is not a recent "innovation," but rather a feature that is deeply ingrained into the design of the operating system.

This will cover the following commands:

- chmod - modify file access rights
- su - temporarily become the superuser
- chown - change file ownership
- chgrp - change a file's group ownership

File permissions

Linux uses the same permissions scheme as Unix. Each file and directory on your system is assigned access rights for the owner of the file, the members of a group of related users, and everybody else. Rights can be assigned to read a file, to write a file, and to execute a file (i.e., run the file as a program).

To see the permission settings for a file, we can use the `ls` command as follows:

```
[me@linuxbox me]$ ls -l some_file
-rw-rw-r-- 1 me   me   1097374 Sep 26 18:48 some_file
```

We can determine a lot from examining the results of this command:

- The file "some_file" is owned by user "me"
- User "me" has the right to read and write this file
- The file is owned by the group "me"
- Members of the group "me" can also read and write this file
- Everybody else can read this file

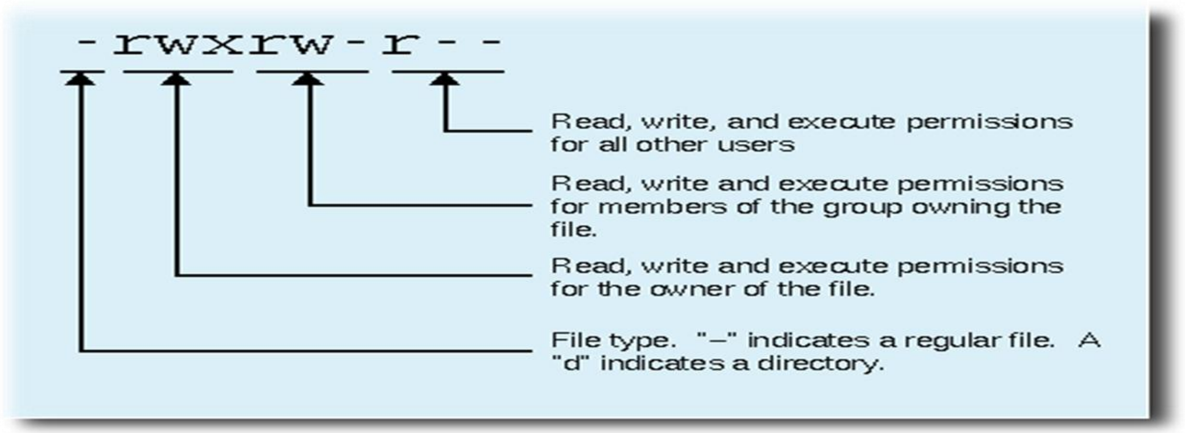
Let's try another example. We will look at the `bash` program which is located in the `/bin` directory:

```
[me@linuxbox me]$ ls -l /bin/bash
-rwxr-xr-x 1 root root  316848 Feb 27  2000 /bin/bash
```

Here we can see:

- The file "/bin/bash" is owned by user "root"
- The superuser has the right to read, write, and execute this file
- The file is owned by the group "root"
- Members of the group "root" can also read and execute this file
- Everybody else can read and execute this file

In the diagram below, we see how the first portion of the listing is interpreted. It consists of a character indicating the file type, followed by three sets of three characters that convey the reading, writing and execution permission for the owner, group, and everybody else.


-rwxrw-r--
Read, write, and execute permissions for all other users
Read, write and execute permissions for members of the group owning the file.
Read, write and execute permissions for the owner of the file.
File type. "–" indicates a regular file. A "d" indicates a directory.

chmod

The `chmod` command is used to change the permissions of a file or directory. To use it, you specify the desired permission settings and the file or files that you wish to modify. There are two ways to specify the permissions.

Binary	Octal	Permission
000	0	—
001	1	–x

Binary	Octal	Permission
010	2	-w-
011	3	-wx
100	4	r–
101	5	r-x
110	6	rw-
111	7	rwx

It is easy to think of the permission settings as a series of bits (which is how the computer thinks about them). Here's how it works:

rwx rwx rwx = 111 111 111

rw- rw- rw- = 110 110 110

rwx --- --- = 111 000 000

and so on...

rwx = 111 in binary = 7

rw- = 110 in binary = 6

r-x = 101 in binary = 5

r-- = 100 in binary = 4

chmod Command Syntax and Options

The format of a chmod command is:

```
chmod [who][+,-,=][permissions] filename
```

Consider the following chmod command:

```
chmod g+w ~/group-project.txt
```

This grants all members of the usergroup that owns the file ~/group-project.txt write permissions. Other possible options to change permissions of targeted users are:

Who (Letter)	Meaning
u	User
g	Group
o	Others
a	All

The + operator grants permissions whereas the - operator takes away permissions. Copying permissions is also possible:

```
chmod g=u ~/group-project.txt
```

The parameter g=u means grant group permissions to be same as the user's.
Multiple permissions can be specified by separating them with a comma, as in the following example:

```
chmod g+w,o-rw,a+x ~/group-project-files/
```

This adds write permissions to the usergroup members, and removes read and write permissions from the "other" users of the system. Finally the a+x adds the execute permissions to all

categories. This value may also be specified as +x. If no category is specified, the permission is added or subtracted to all permission categories.
In this notation the owner of the file is referred to as the user (e.g. u+x).

```
chmod -R +w,g=rw,o-rw, ~/group-project-files/
```

The -R option applies the modification to the permissions recursively to the directory specified and to all of its contents.

For example if you want a file that has -rw-rw-rwx permissions you will use the following:

Owner	**Group**	**Other**
read & write	read & write	read, write & execute
4+2=6	4+2=6	4+2+1=7

```
user@host:/home/user$ chmod 667 filename
```

Another example if you want a file that has --w-r-x--x permissions you will use the following:

Owner	**Group**	**Other**
write	read & execute	execute
2	4+1=5	1

```
user@host:/home/user$ chmod 251 filename
```

Here are a few examples of chmod usage with numbers (try these out on your system).

Now, if you represent each of the three sets of permissions (owner, group, and other) as a single digit, you have a pretty convenient way of expressing the possible permissions settings. For example, if we wanted to set `some_file` to have read and write permission for the owner, but wanted to keep the file private from others, we would:

```
[me@linuxbox me]$ chmod 600 some_file
```

Here is a table of numbers that covers all the common settings. The ones beginning with "7" are used with programs (since they enable execution) and the rest are for other kinds of files.

Value ***Meaning***

777	*(rwxrwxrwx)* No restrictions on permissions. Anybody may do anything. Generally not a desirable setting.
755	*(rwxr-xr-x)* The file's owner may read, write, and execute the file. All others may read and execute the file. This setting is common for programs that are used by all users.
700	*(rwx------)* The file's owner may read, write, and execute the file. Nobody else has any rights. This setting is useful for programs that only the owner may use and must be kept private from others.
666	*(rw-rw-rw-)* All users may read and write the file.
644	*(rw-r--r--)* The owner may read and write a file, while all others may only read the file. A common setting for data files that everybody may read, but only the owner may change.
600	*(rw-------)* The owner may read and write a file. All others have no rights. A common setting for data files that the owner wants to keep private.

Directory permissions

The `chmod` command can also be used to control the access permissions for directories. In most ways, the permissions scheme for directories works the same way as they do with files. However, the execution permission is used in a different way. It provides control for access to file listing and other things. Here are some useful settings for directories:

Value	***Meaning***
777	*(rwxrwxrwx)* No restrictions on permissions. Anybody may list files, create new files in the directory and delete files in the directory. Generally not a good setting.

755 *(rwxr-xr-x)* The directory owner has full access. All others may list the directory, but cannot create files nor delete them. This setting is common for directories that you wish to share with other users.

700 *(rwx------)* The directory owner has full access. Nobody else has any rights. This setting is useful for directories that only the owner may use and must be kept private from others.

Changing Owners and Groups

While creating an account on Unix, it assigns a **owner ID** and a **group ID** to each user. All the permissions mentioned above are also assigned based on the Owner and the Groups.

Two commands are available to change the owner and the group of files −

- **chown** − The **chown** command stands for **"change owner"** and is used to change the owner of a file.
- **chgrp** − The **chgrp** command stands for **"change group"** and is used to change the group of a file.

Changing File Ownership

You can change the owner of a file by using the **chown** command. Here's an example: Suppose I wanted to change the owner of `some_file` from "me" to "you". I could:

```
[me@linuxbox me]$ su
Password:
[root@linuxbox me]# chown you some_file
[root@linuxbox me]# exit
[me@linuxbox me]$
```

Notice that in order to change the owner of a file, you must be the superuser. To do this, our example employed the **su** command, then we executed **chown**, and finally we typed **exit** to return to our previous session.

chown works the same way on directories as it does on files.

Changing Group Ownership

The group ownership of a file or directory may be changed with **chgrp**. This command is used like this:

```
[me@linuxbox me]$ chgrp new_group some_file
```

In the example above, we changed the group ownership of `some_file` from its previous group to "new_group". You must be the owner of the file or directory to perform a **chgrp**.

Specifying the new owner

New ownership of *file* is specified by the argument *new-owner*, which takes this general form:

[*user*[:*group*]]

Specifically, there are five ways to format *new-owner*:

new-owner form	Description
user	The name of the user to own the file. In this form, the colon (":") and the *group* is omitted. The owning group is not altered.
user:*group*	The *user* and *group* to own the file, separated by a colon, with no spaces in between.
:*group*	The group to own the file. In this form, *user* is omitted, and the *group* must be preceded by a colon.
user:	If *group* is omitted, but a colon follows *user*, the owner is changed to *user*, and the owning group is changed to the login group of *user*.
:	Specifying a colon with no *user* or *group* is accepted, but ownership will not be changed. This form does not cause an error, but changes nothing.

Notes on usage

- *user* and *group* can be specified by name or by number.
- Only root can change the owner of a file. The owner cannot transfer ownership, unless the owner is root, or uses **sudo** to run the command.
- The owning group of a file can be changed by the file's owner, if the owner belongs to that group. The owning group of a file can be changed, by root, to any group. Members of the owning group other than the owner cannot change the file's owning group.

- The owning group can also be changed by using the **chgrp** command. **chgrp** and **chown** use the same system call, and are functionally identical.
- Certain miscellaneous file operations can be performed only by the owner or root. For instance, only owner or root can manually change a file's "atime" or "mtime" (access time or modification time) using the **touch** command.
- Because of these restrictions, you will almost always want to run **chown** as root, or with **sudo**.

Options

Option	Description
-c, **--changes**	Similar to **--verbose** mode, but only displays information about files that are actually changed. For example: changed ownership of 'dir/dir1/file1' from hope:neil to hope:hope
-v, **--verbose**	Display verbose information for every file processed. For example: changed ownership of 'dir/dir1/file1' from hope:neil to hope:hope ownership of 'dir/dir1' retained as hope:hope
-f, **--silent**, **--quiet**	Quiet mode. Do not display output.
--dereference	Dereference all symbolic links. If *file* is a symlink, change the owner of the referenced file, not the symlink itself. This is the default behavior.
-h, **--no-dereference**	Never dereference symbolic links. If *file* is a symlink, change the owner of the symlink rather than the referenced file.
--from=*currentowner***:***currentgroup*	Change the owner or group of each file only if its current owner or group match *currentowner* and/or *currentgroup*. Either may be omitted, in which case a match is not required for the other attribute.

--no-preserve-root	Do not treat / (the root directory) in any special way. This is the default behavior. If the **--preserve-root** option is previously specified in the command, this option will cancel it.
--reference=*ref-file*	Use the owner and group of file *ref-file*, rather than specifying ownership with *new-owner*.
-R, **--recursive**	Operate on files and directories recursively. Enter each matching directory, and operate on all its contents.

Why change a file's ownership?

You should use **chown** when you want a file's user or group permissions to apply to a different user or group.

Hypothetical scenarios

Here are some examples of when you might use **chown**:

- You create a file, **myfile.txt**, using **sudo** or while logged in as root, so the file is owned by **root**. However, you intend the file to be used by your regular user account, **myuser**.

  Use **chown** to change the owner:

```
sudo chown myuser myfile.txt
```

- You own **myfile.txt**, but you want to give it to another user on the system named **notme**. You also want to change the owning group to that user's group, **notmygroup**.

Use chown to change the owner and group:

```
sudo chown notme:notmygroup myfile.txt
```

- You just transferred an entire directory of files, **otherfiles**, from another computer. All the files and directories are owned by your username on the other system, and you want your current user and group to own them all.

  Change the ownership of the directory and all its contents recursively, with the **-R** option:

```
sudo chown -R myuser:mygroup otherfiles
```

The above command will change the ownership of every file, subdirectory, and subdirectory contents in **otherfiles**.

Examples

Viewing ownership

Before you use **chown**, you may want to check the current ownership of a file. You can view a file's ownership, permissions, and other important information with the **ls** command, using the **-l** option:

```
ls -l myscript.sh

-rwxrw-r-- 1 hope hopeusers 12 Nov  5 13:14 myscript.sh
```

In the output, you see several fields of information listed, including the permissions and ownership of the file. It might not make sense at first, so let's describe it in detail.

Here's what the information means:

Data	Field position	Description
-	Field **1**, character **1**	**File type**: **d** for a directory, **l** (lowercase L) for a symbolic link, or **-** (a dash) for a regular file.
rwx	Field **1**, characters **2-4**	**User permissions**. The owner can read ("**r**"), write to ("**w**"), and execute ("**x**") this file.
rw-	Field **1**, characters **5-7**	**Group permissions**. The owning group can read and write to this file, but cannot execute it as a command.
r--	Field **1**, characters **8-10**	**Other permissions**, also known as **world** permissions. Any other user on the system is allowed to read the file only.
1	Field **2**	**Number of symbolic links** to this file. If there are no symbolic links to the file, this number is **1**, because the original file name is included in this count. If there were one symbolic link to the file, this number would be **2**, or **3** for two symbolic links, etc.
hope	Field **3**	**Name of owner**. This is the name of the user who owns the file. When this user tries to access the file, access is restricted according to the **user permissions**.
hopeusers	Field **4**	**Name of owning group**. This is the user group who owns the file. When a user who is a member of this group tries to access the file, access is restricted according to the **group permissions**.
12	Field **5**	**Size**. This file contains **12** bytes of data.
Nov	Field **6**	**Mtime (month)**. Abbreviated name of the month when the file's contents were last modified. This file was last modified in the month of November.
5	Field **7**	**Mtime (day of month)**. This file was last modified on the fifth day of November.

13:14	Field **8**	**Mtime (time, or year)**. This file was last modified at **13:14** (1:34 P.M.) on November 5 of this year. If it was modified over a year ago, this field would list the year instead, for instance **2015**.
myscript.sh	Field **9**	**File name**. The name of the file.

So the important fields here are 1, 3 and 4. They tell us that user **hope** can read, write, or execute the file's contents, and members of the group **hopeusers** can read or write to it.

Changing ownership

Change the owner of **file.txt** to user **hope**

```
sudo chown hope file.txt
```

Change the owner of **file1**, **file2**, and **file3** to user **hope**

```
sudo chown hope file1 file2 file3
```

Here, the asterisk ("*") is a wildcard which the shell expands to a list of every file whose name begins with "**file**". If the current directory contains four files named **file1**, **file2**, **file3**, and **file4**, all these files' names are passed to the **chown** command, and their owners changed to user **hope**

```
sudo chown hope file*
```

Change the owner of file or directory **myfiles** to user **hope**.

```
sudo chown hope myfiles
```

Change the owner of **myfiles** to user **hope**. If **myfiles** is a directory, **chown** will recursively (**-R**) search that directory, and change the owner of all files, subdirectories, and subdirectory contents.

```
sudo chown -R hope myfiles
```

Change the owners of **file1** and **file2** to user **hope**, and the owning groups to **admins**.

```
sudo chown hope:admins file1 file2
```

Change the owner of **file1** to user **hope**, and the owning group to hope's login group.

```
sudo chown hope: file1
```

Change the owning group of **file2** to group **othergroup**. Notice that this is the only command in these examples which may be run without **sudo**.

```
chown :othergroup file2
```

If user **hope** runs the previous command but does not belong to group **othergroup**, the command will fail, unless it is run with **sudo**.

Change the ownership of **file1** to the user with numeric UID **1000**, and the group with numeric GID **1001**.

```
sudo chown 1000:1001 file1
```

Same as the previous command. If user **hope** has UID 1000, and another user is named "1000" but has UID 1002, this command form (with the "**+**" signs) unambiguously changes the owner to **hope**.

```
sudo chown +1000:+1001 file1
```

Recursively change the ownership of directory **Documents**, and all files and subdirectories therein, to user **hope**, group **hope**.

```
sudo chown -R hope:hope Documents
```