CONTROLLING FILE AND DIRECTORY PERMISSIONS

As we know, in Linux the root user is all-powerful. Other users on the system have more limited capabilities and permissions and almost never have the access that root user has. The root user is a part of the root group by default. Each new user on the system must be added to a group in order to inherit the permissions of that group.

r – permission to read

w – permission to write

x – permission to execute

GRANTING OWNERSHIP TO AN INDIVIDUAL USER

Syntax> chown <user> <path to the file>

Kali> chown matt /tmp/mattfile

Here, we give the command, the name of the user we are giving ownership to and then the location and name of the relevant file.

GRANTING OWNERSHIP TO A GROUP

Syntax> chgrp <group> <path to the file>

CHECKING PERMISSION

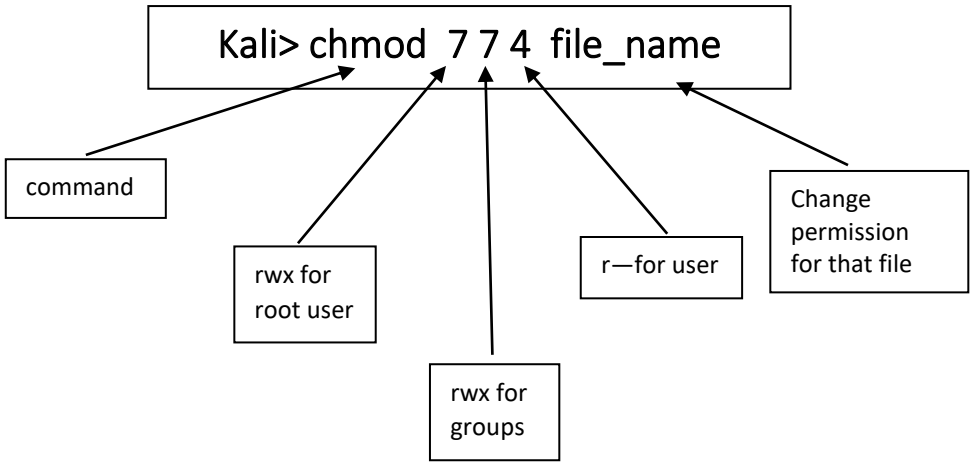
Kali> ls -l

1. The file types. d – for directory, (-) – file, c – character file
2. The permissions on the file for owner-groups-users respectively
3. The number of links
4. The owner of the file
5. The size of the file in bytes
6. When the file was created or last modification
7. The name of the file

CHANGING PERMISSION

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

The octal representation for -wx is 3,
Because w=2 and x=3 i.e., w+x=3
Viz: rwx=4+2+1=7



CHANGING PERMISSION WITH UGO

- Removes a permission

+ Adds a permission

= Sets a permission

u for user

g for group

o for others

viz,

```
kali> chmod u+x, o+x file
kali> chmod a+x file        → a+x execute permission for all user
kali> chmod a-x file
```

SETTING MORE DEFAULT PERMISSION WITH MASK

As you have been seen, Linux automatically assigns base permission – usually *666* for files and *777* for directories. So, we can change the default permissions allocated to files and directories created by each user with *umask* (unmask) method. This method represents the permissions we want to remove from the base permissions on a file or directory to make them more secure.

The *umask* is three-digit decimal number corresponding to the three permissions digits but the umask number is subtracted from the permissions number to give the new permission status. This means that when a new file or a directory is created, its permissions are set to the default value minus the value in *umask*.

For example:

If the umask is set to 022, a new file with the original default permissions of 666 will now have the permissions 644.

In Kali, as with the most Debian systems, the *umask* is preconfigured to *022*, meaning the Kali default is *644* for files and *755* for directories. The *umask* value is not universal to all users on the

system. Each user can set a personal default *umask* value for the files and directories in their personal .profile file. To change the *umask* value for a user, edit the file /home/username/.profile and for example, add umask 007 to set it so only the user and members of the user's group have permissions.

SPECIAL PERMISSIONS

In addition to the three general-purpose permissions, *rwx*, Linux has the three special permissions that are slightly more complicated. These special permissions are set user ID (or SUID), set group ID (or SGID), and sticky bit.

GRANTING TEMPORARY ROOT PERMISSION WITH SUID:

If the user only has read or write permissions, they cannot execute. This may seem straightforward, but there are exceptions to this rule.

You may have encountered a case in which a file requires the permissions of the root user during execution for all users, even those who are not root. For example, a file that allows users to change their password would need access to the /etc/shadow file—the order to execute. In case we can temporarily grant the owner's privileges to execute the file by setting the *SUID* bit on the program.

Basically, the *SUID* bit says that any user can execute the file with the permissions of the owner but those permissions don't extend beyond the user of that file.

To set the *SUID* bit, enter a 4 before the regular permissions, so a file with a new resulting permission of 644 is represented as 4644 when the SUID bit is set. Setting the *SUID* on a file is not something a typical user would do, but if we want to do so, we'll use the:

```
Kali> chmod 4644 filename
```

GRANTING THE ROOT USER'S GROUP PERMISSIONS SGID:

SGID also grants temporary elevated permissions, but it grants the permissions of the file owner's group, rather than of the file's owner. This means that, with an *SGID* bit set, someone without execute permissions can execute a file if the owner belongs to the group that has permissions to execute that file.

The *SGID* bit works slightly different when applied to a directory: when the bit is set on a directory, ownership of new files created in that directory goes to the directory creator's group, rather than the file creator's group. This is very useful when a directory is shared by multiple users. All users in that group can execute the file(s), not just a single user.

```
Kali> chmod 2644 filename
```

Regular permission with 2 at the beginning.

THE OUTMODED STICKY BIT

The sticky bit is a permission bit that we can set on a directory to allow a user to delete or rename files within that directory. However, the sticky bit is legacy of older Unix systems, and modern systems (like Linux) ignore it.

SPECIAL PERMISSIONS, PRIVILEGE ESCALATION, AND THE HACKER

As a hacker, these special permissions can be used to exploit Linux systems through privilege escalation, whereby a regular user gains root or sysadmin privileges and the associated permissions.

One way to do is to exploit the *SUID* bit. A system administrator or software developer might set the *SUID* bit on a program to allow that program access to files with root privileges. For instance, scripts that need to change passwords often have the *SUID* bit set. We, the hacker, can use that permission to gain temporary root privileges and do something malicious, such as get access to the passwords at /etc/shadow.

```
Kali> find / -user root -prem -4000
```

With this command, we ask kali to start looking at the top of the file system with the / syntax. It then looks everywhere below / for files that are owned by root, specified with user root, and that have the SUID permission bit set (-perm -4000).

The output reveals numerous files that have the *SUID* bit set. Let's navigate to the /usr/bin directory, where many of these files reside, and then run a long listing directory and scroll down to the sudo file:

```
Kali> cd /usr/bin
```

```
Kali> ls -l     => -rwsr-xr-x     1          root      root      ..          ..          ..
```

The first set of permissions for the owner has an "s" in place of the x. This is how Linux represents that the SUID bit is set. This means that anyone who runs the *sudo* file has the privileges of the root user, which can be a security concern for the sysadmin and a potential attack vector for the hacker. For instance, some applications need to access the /etc/shadow file to successfully complete their tasks. If the attacker can gain control of that application, they can use that application's access to the passwords on a Linux system.

Linux has a well-developed system of security that protects files and directories from unauthorized access. The aspiring hacker needs to have a basic understanding of this system not only to protect their files but also to execute new tools and files. In some cases, hackers can exploit the SUID and SGID permissions to escalate privileges from a regular user to a root user.