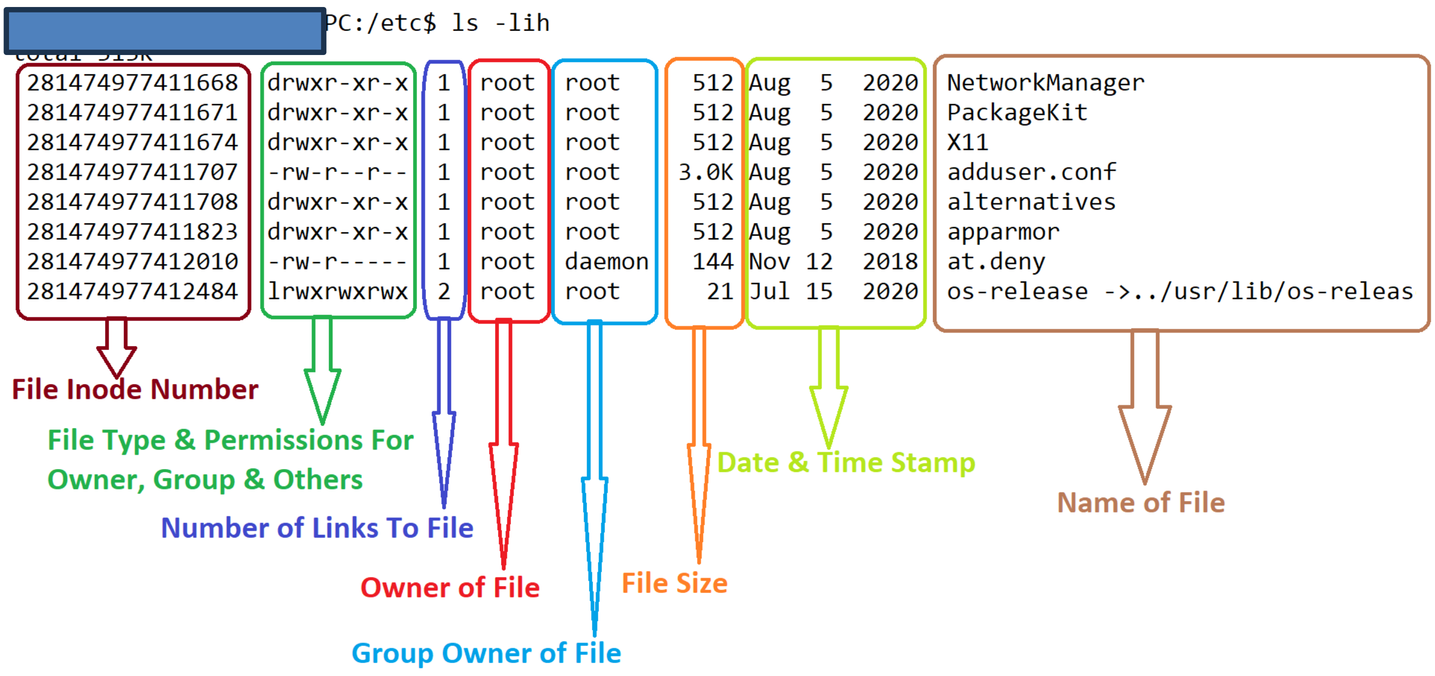
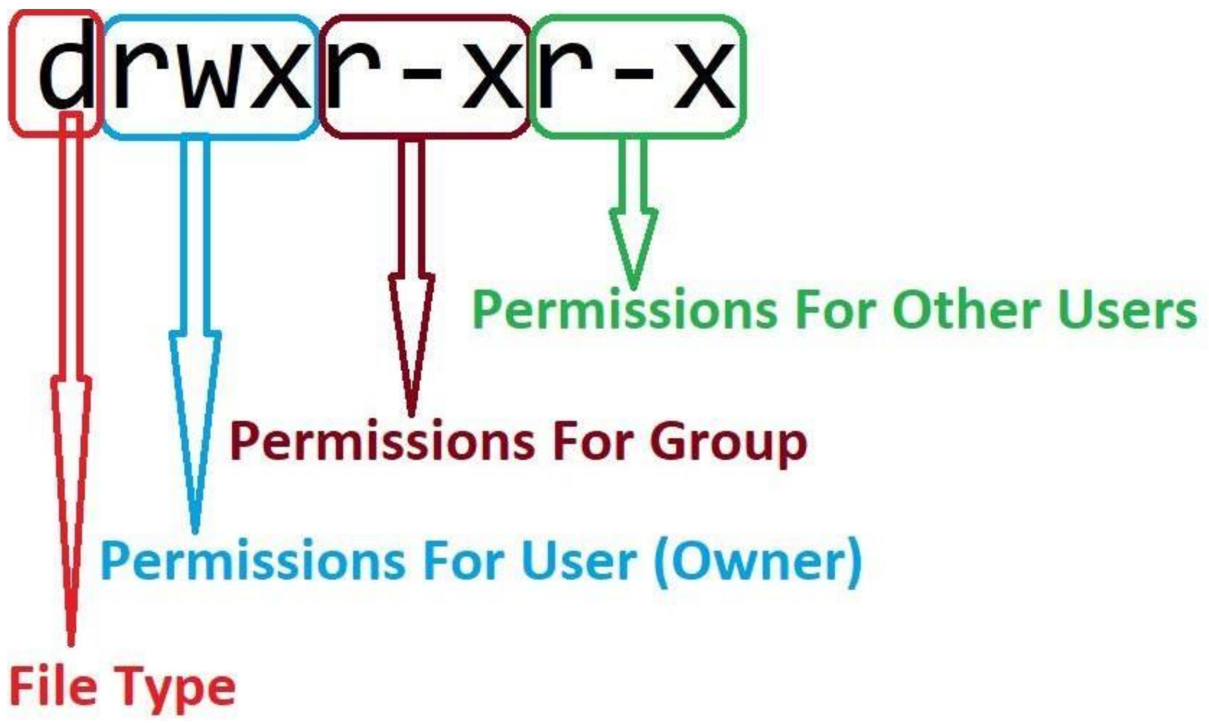
File Security: Standard File Permissions in Linux



```
[redacted]PC:/etc$ ls -lih
281474977411668 drwxr-xr-x 1 root root    512 Aug  5  2020 NetworkManager
281474977411671 drwxr-xr-x 1 root root    512 Aug  5  2020 PackageKit
281474977411674 drwxr-xr-x 1 root root    512 Aug  5  2020 X11
281474977411707 -rw-r--r-- 1 root root   3.0K Aug  5  2020 adduser.conf
281474977411708 drwxr-xr-x 1 root root    512 Aug  5  2020 alternatives
281474977411823 drwxr-xr-x 1 root root    512 Aug  5  2020 apparmor
281474977412010 -rw-r----- 1 root daemon  144 Nov 12  2018 at.deny
281474977412484 lrwxrwxrwx 2 root root     21 Jul 15  2020 os-release ->../usr/lib/os-releas
```



first character	**file type**
-	normal file
d	directory
l	symbolic link
p	named pipe
b	block device
c	character device
s	socket

permission	**on a file**	**on a directory**
r (read)	read file contents (cat)	read directory contents (ls)
w (write)	change file contents (vi)	create files in (touch)
x (execute)	execute the file	enter the directory (cd)

Permissions Bit

Weightage: Read

Permission ➔ r = 4 Write Permission ➔ w = 2Execute

Permission ➔ x = 1

Maximum Permissions (Full Permission) ➔ rwx =

4+2+1 = 7Minimum Permissions (No Permission)

➔ --- =

0+0+0 = 0

Permissions can be represented by 8 numbers starting from number 0 to number 7, Which means by Octal number system having a base of 8.

Octal Number System ➔ $(0\text{-}7)_8$

We need three binary bits to represent Octal numbers. These three bits can be used to represent permission, here first bit represent read (r), second bit represent write (w) and third bit represents execute (x) permissions.

Octal Number	Binary	Permissions
0	000	---
1	001	--x
2	010	-w-
3	011	-wx

4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

Here are the commands to list all users in Linux:

1. `cat /etc/passwd`
2. `cut -d: -f1 /etc/passwd`
3. `getent passwd`
4. `getent passwd | cut -d: -f1`
5. `compgen -u`

Chapter: File Permissions in Linux

1. Introduction to File Permissions

File permissions in Linux are used to control who can read, write, or execute a file. Permissions are essential for file security and are represented by three key characters:

- **r**: Read permission
- **w**: Write permission
- **x**: Execute permission

Permissions are grouped into three categories:

1. **User (u)**: Permissions for the owner of the file.
2. **Group (g)**: Permissions for the group assigned to the file.
3. **Others (o)**: Permissions for all other users on the system.

A typical permission string, such as `-rw-r--r--`, is broken down as follows:

- The first character (`-` or `d`) represents whether the object is a file or directory. `d` indicates a directory.
- The next three characters (`rw-`) show the permissions for the owner: read (`r`), write (`w`), and no execute permission (`-`).
- The next three characters (`r--`) represent group permissions, in this case allowing only read access.
- The final three characters (`r--`) are the permissions for others, also set to read-only.

2. File Security: Standard File Permissions in Linux

File permissions in Linux are defined using numeric values, with each type of permission having a corresponding weight:

- **r (read)**: 4
- **w (write)**: 2
- **x (execute)**: 1

These values allow for combining permissions based on their sum:

- Full Permission (rwx): 4 + 2 + 1 = 7
- No Permission (---): 0 + 0 + 0 = 0

Permissions are represented using an **octal number system** (base 8), with numbers ranging from 0 to 7. This table shows the octal values:

Octal Number	Binary	Permissions
0	000	—
1	001	–x
2	010	-w-
3	011	-wx
4	100	r–
5	101	r-x
6	110	rw-
7	111	rwx

For example, to set full permissions for the owner (rwx), read and execute for the group (r-x), and execute-only for others (--x), the octal value would be 751.

```
chmod 751 myfile.txt
```

3. Changing File Permissions with chmod

3.1 Symbolic Mode (Using ugo)

You can modify permissions using symbolic notation to specify user (u), group (g), others (o), or all (a) permissions. Operators like + (add), - (remove), and = (set exact) are used to manipulate permissions.

Example 1: Add read permission for everyone

```
chmod a+r myfile.txt
```

This command grants read permission to all users (user, group, others). If you check the file afterward:

```
ls -l myfile.txt
-rw-r--r--  1 user1 group1   62 Jan 15 16:10 myfile.txt
```

3.2 Absolute Mode (Using Octal Notation)

In octal mode, permissions are assigned using a three-digit octal number, where each digit controls the permissions for the owner, group, and others. The values are:

- **r (read)**: 4
- **w (write)**: 2
- **x (execute)**: 1

Example 2: Set full permissions for the owner, read-execute for the group, and execute-only for others

```
chmod 751 myfile.txt
```

After running this command, the file will have the following permissions:

```
ls -l myfile.txt
-rwxr-x--x  1 user1 group1   62 Jan 15 16:10 myfile.txt
```

4. Changing Ownership with chown and chgrp

You can also change file ownership using the chown command:

```
chown newuser myfile.txt
```

This changes the owner of myfile.txt to newuser. To change the group, use chgrp:

```
chgrp newgroup myfile.txt
```

To change both the owner and group simultaneously, use:

```
chown newuser:newgroup myfile.txt
```

5. Default Permissions and umask

When you create a new file or directory, the system sets default permissions based on the umask value. The umask defines the permissions that will be *subtracted* from the default full permission set (usually 666 for files and 777 for directories).

View current umask value:

```
umask
```

Change umask value:

```
umask 022
```

In this case, 022 removes write permissions for the group and others, so files will be created with permissions 644 and directories with 755.

6. Adding Permissions at the Time of Folder Creation

You can specify permissions during folder creation using the mkdir command with the -m option.

Example: Create a directory with specific permissions

```
mkdir -m 755 newdir
```

This creates newdir with permissions allowing the owner to read, write, and execute, while the group and others can only read and execute.

7. Advanced Example: Remove All Permissions

If you want to remove read, write, and execute permissions for all users from a file, use the following command:

```
chmod a-rwx f1
```

This command removes all permissions from the file f1 for the owner, group, and others.

Output:

```
ls -l f1
----------  1 user1 group1  62 Jan 15 16:10 f1
```

Chapter: Special File Permissions in Linux

Introduction

This chapter explores special file permissions in Linux: **SetUID**, **SetGID**, and the **Sticky Bit**. These permissions are essential for controlling access and enhancing security for executable files and directories.

Table of Contents

1. Understanding Special Permissions

Overview of Special Permissions

Special permissions in Linux control how files and directories are accessed and executed. They include **SetUID**, **SetGID**, and the **Sticky Bit**.

Importance and Use Cases

- **SetUID**: Grants temporary elevated privileges to execute files.
- **SetGID**: Ensures files and directories inherit specific group ownership.
- **Sticky Bit**: Restricts file deletion in shared directories.

2. SetUID (Set User ID)

What is SetUID?

SetUID allows executable files to run with the privileges of the file's owner. It is often used for programs that require elevated permissions.

Benefits of SetUID

- Provides elevated permissions temporarily.
- Allows specific programs to execute with root privileges.

Syntax for Setting SetUID

- **Symbolic**: `chmod u+s <file_name>`
- **Octal/Binary**: `chmod 4755 <file_name>`

Examples and Use Cases

- **Example**: Setting SetUID on a script.

```
• sudo chown root:root /path/to/script
• chmod u+s /path/to/script
```

Identifying SetUID

- `s`: When execute permission is set (`rws`).
- `S`: When execute permission is not set (`rwS`).

How to Remove SetUID

- **Symbolic**: `chmod u-s <file_name>`
- **Octal**: `chmod 0755 <file_name>`

3. SetGID (Set Group ID)

What is SetGID?

SetGID on directories ensures that files created within inherit the group ownership of the directory. On executable files, it allows execution with the group's privileges.

Benefits of SetGID

- Ensures consistent group ownership of files.

- Allows specific files to execute with group privileges.

Syntax for Setting SetGID

- **Symbolic**: `chmod g+s <file/directory_name>`
- **Octal/Binary**: `chmod 2755 <file/directory_name>`

Examples and Use Cases

- **Example**: Setting SetGID on a directory.

```
chown root:developers /shared
chmod g+s /shared
```

Identifying SetGID

- `s`: When execute permission is set (`rws`).
- `S`: When execute permission is not set (`rwS`).

How to Remove SetGID

- **Symbolic**: `chmod g-s <file/directory_name>`
- **Octal**: `chmod 0755 <file/directory_name>`

4. Sticky Bit

What is the Sticky Bit?

The Sticky Bit, when set on a directory, ensures that only the owner of a file can delete or rename their own files within the directory.

Benefits of the Sticky Bit

- Prevents accidental file deletion in shared directories.
- Enhances security in public directories like `/tmp`.

Syntax for Setting the Sticky Bit

- **Symbolic**: `chmod +t <directory_name>`
- **Octal/Binary**: `chmod 1777 <directory_name>`

Examples and Use Cases

- **Example**: Setting Sticky Bit on a directory.

```
chmod +t /shared
```

Identifying Sticky Bit

- `t`: When execute permission for others is set (`rwxrwxrwt`).
- `T`: When execute permission is not set (`rwxrwxr-T`).

How to Remove the Sticky Bit

- **Symbolic**: `chmod -t <directory_name>`
- **Octal**: `chmod 0777 <directory_name>`

5. Quick Reference Points

Summary of Special Permissions

- **SetUID**: `u+s` or `4755`
- **SetGID**: `g+s` or `2755`
- **Sticky Bit**: `+t` or `1777`

Common Commands and Syntax

- **SetUID**: `chmod u+s file`, `chmod 4755 file`
- **SetGID**: `chmod g+s file`, `chmod 2755 file`
- **Sticky Bit**: `chmod +t directory`, `chmod 1777 directory`

Tips for Managing Permissions

- Use **SetUID** for programs needing root privileges.
- Apply **SetGID** to directories for consistent group ownership.
- Set the **Sticky Bit** on shared directories to prevent unwanted file deletions.

6. Practical Applications

Use Cases for Special Permissions

- **SetUID**: Allows safe execution of administrative tasks.
- **SetGID**: Maintains group consistency in collaborative environments.
- **Sticky Bit**: Protects files in public or shared directories.

Security Considerations

- Regularly audit SetUID and SetGID permissions.
- Ensure only trusted applications have elevated privileges.
- Apply the Sticky Bit in shared environments to enhance security.

Best Practices

- Use SetUID sparingly to minimize security risks.
- Ensure SetGID is used in directories where consistent group ownership is needed.
- Apply the Sticky Bit to directories like `/tmp` to prevent unauthorized deletions.

7. Exercises and Challenges

Hands-On Exercises

1. **Create a directory with SetGID** and test file creation.
2. **Set SetUID** on a script and verify its behavior.
3. **Apply Sticky Bit** to a directory and test file deletion.

Real-World Scenarios

1. **Securing Temporary File Storage**: Configure `/tmp` with Sticky Bit.
2. **Setting Up a Collaborative Project Directory**: Use SetGID to ensure consistent group ownership.

Access Control Lists (ACL) in Linux (For RHCSA)

Definition:
ACLs (Access Control Lists) provide a way to grant specific permissions to multiple users or groups beyond the traditional owner-group-others model in Linux.

Why Use ACLs:

- The standard permission system is limiting when multiple users or groups require different access levels.
- ACLs allow more granular control, enabling you to grant custom read, write, or execute permissions to specific users or groups.

Key Syntax and Commands for ACLs

1. **Viewing ACLs:**

```
getfacl filename
```

   Displays the ACL for a file or directory.

3. **Setting ACLs Using Octal Values:**
   Use the octal system to set permissions:

```
setfacl -m u:user1:7 filename   # rwx for user1
setfacl -m g:group1:5 filename  # r-x for group1
```

6. **Removing ACLs:**
   - **Remove specific ACLs:**

```
setfacl -x u:user1 filename
setfacl -x g:groupname filename
```

   - **Remove all ACL entries (reset):**

```
setfacl -b filename
```

7. **Setting ACLs on Groups:**
   Assign ACLs for a group:

```
setfacl -m g:groupname:rwx filename
```

9. **Using ACLs without Modifying the Mask (`-n` or `--no-mask`):**
   Modify ACLs without changing the mask, which sets the maximum effective permissions for users and groups:
   - Short form:

```
setfacl -n -m u:user1:rwx filename
```

   - Long form:

```
setfacl --no-mask -m u:user1:rwx filename
```

RHCSA Topics Related to ACLs

1. **Default ACLs:**
   To set default ACLs for directories so that all new files inherit specific permissions:

```
2. setfacl -d -m u:user1:rwx directory_name
```

3. **ACLs on Directories:**
   Ensure proper directory-level permissions to allow users access to files within:

```
4. setfacl -m u:user1:rwx directory_name
```

5. **Masking in ACLs:**
   The mask controls the maximum permissions for all users/groups except the owner:

```
6. setfacl -m m::rwx filename
```

7. **Verifying if a File Supports ACLs:**
   Ensure the filesystem supports ACLs:

```
8. tune2fs -l /dev/sda1 | grep "Default mount options"
```

   or check with `mount` command options (`acl`).

9. **Persistent ACLs Across Reboots:**
   To ensure ACLs persist after reboots, make sure the filesystem is mounted with ACL support. Check or modify `/etc/fstab` with the `acl` option:

```
10.  UUID=xxxxxx / ext4 defaults,acl 1 1
```

Example:

Grant `rwx` to a specific user (`user1`) without modifying the mask:

```
setfacl -n -m u:user1:rwx myfile.txt
```

Remove all ACLs from the file `myfile.txt`:

```
setfacl -b myfile.txt
```