



Ownership & Permissions

Understanding File Access Control in Linux

By Sameer ur Rehman

```
# ls -l file
-rw-r--r-- 1 root root 0 Nov 19 23:49 file
```

Diagram illustrating the permissions for the file `file`:

- Owner (rw-)**: The first two characters (`rw`) represent the permissions for the owner.
- Group (r- -)**: The next two characters (`r- -`) represent the permissions for the group.
- Other (r- -)**: The last two characters (`r- -`) represent the permissions for others.

Legend:

- `r` = Readable
- `w` = Writeable
- `x` = Executable
- `-` = Denied

File type: The first character (`-`) indicates the file type (regular file).

What You Will Learn

Introduction — When Linux Decides Who Can Touch What

The **ls -l** Mystery — Reading File Identity Cards

- 2.1 The First Character — File Type
- 2.2 The Next Nine Characters — Permissions (rwx)
- 2.3 The 3rd Field — Hidden Secrets (./+)
- 2.4 Link Count — The File's Popularity
- 2.5 Owner and Group — Who Owns the File?
- 2.6 Size, Time, and Name — The File's Final Identity

How Permissions Are Evaluated (User → Group → Others)

What Are File Permission Modes?

- 4.1 Symbolic Mode (Letter Method)
- 4.2 Numeric/Octal Mode (Number Method)
- 4.3 The Difference in Practice (Symbolic vs Numeric)

What Ownership Really Means

- 5.1 Dual Ownership Structure (UID/GID, Others)
- 5.2 Who Can Change Ownership? (Rules & Exceptions)
- 5.3 The Ownership Logic Behind the Scene (UID/GID mapping)
- 5.4 Commands for Ownership Management (**chown**, **chgrp**)
- 5.5 Root Privilege Required (When **sudo** is needed)

Linux File vs Directory Permissions

- 6.1 File Permissions — Protect Content
- 6.2 Directory Permissions — Protect Names
- 6.3 How They Interact (Common Scenarios)
- 6.4 Quick Truth Table (file 000, dir 777)

Immutable & Append-Only Attributes (**chattr**)

- 7.1 What Are File Attributes? (**lsattr**, **chattr**)
- 7.2 Immutable (+i)
- 7.3 Append-Only (+a)
- 7.4 Viewing Attributes & Summary Table

Access Control Lists (ACLs)

- 8.1 What Are ACLs? (Why & When)
- 8.2 Add / Recursive / Default ACLs (**setfacl**, **getfacl**)
- 8.3 The Mask — Hidden Gatekeeper (**m:**)
- 8.4 Viewing ACLs (Example Output & Interpretation)
- 8.5 How ACLs Work with Regular Permissions
- 8.6 Managing ACLs — Common Flags (Cheat Sheet)

Summary

Next Lesson — User Management & Advanced Permissions

1 · Introduction — When Linux Decides Who Can Touch What

Imagine you and your friends share a computer at work. Each of you has your own folder to store project files. You create a report named `sales.txt` — by default, **you** become its owner, and your workgroup (say, `marketing`) becomes its group.

Now, Linux needs to decide who can **read**, **edit**, or **run** this file. That's where **permissions** come in:

- **Read (r)**: A user can open and view the file's content.
- **Write (w)**: A user can modify or delete the file.
- **Execute (x)**: A user can run the file as a program or script.

Each file has two key identities:

Identity	Meaning
Owner (User)	The user who created or owns the file.
Group	A collection of users who share access.

Together, they control how the **three permission roles** behave:

- **User (u)** → file's owner
- **Group (g)** → users belonging to the file's group
- **Others (o)** → everyone else on the system

When you create something new — say a report called `sales.txt` — Linux automatically assigns **ownership and rules** to protect it.

Let's explore how those rules work

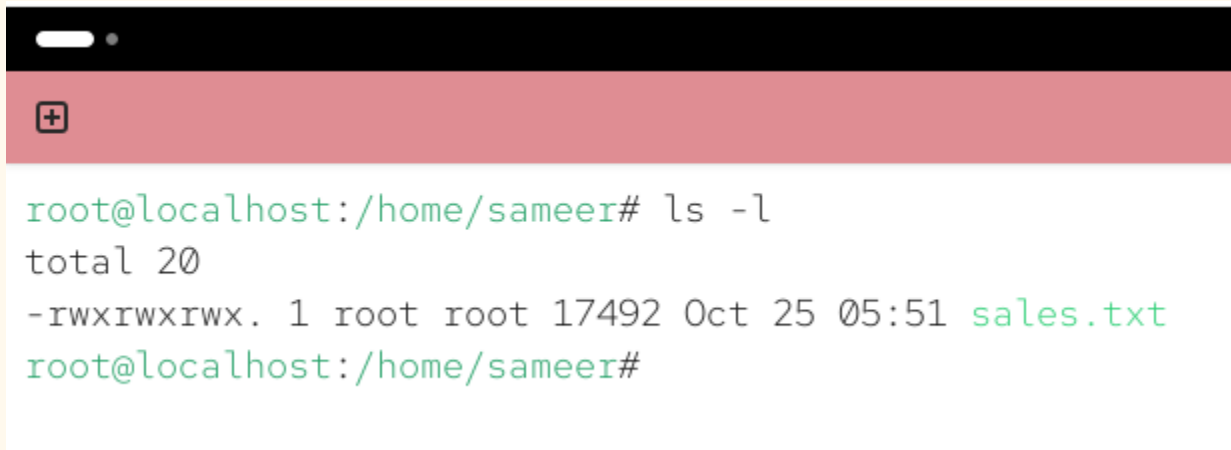
2. The `ls -l` Mystery — Reading File Identity Cards

Type this command:

```
ls -l
```

You might see something like this:

```
-rw-r-xr--. 1 owner group 1024 Sep 22 14:20 file.txt
```



```
root@localhost:/home/sameer# ls -l
total 20
-rwxrwxrwx. 1 root root 17492 Oct 25 05:51 sales.txt
root@localhost:/home/sameer#
```

That long string of letters and symbols in the `ls -l` output is like a **secret code** that describes everything about a file — who owns it, who can use it, and how.

Each part of the line tells a story, and there are **seven key fields** you need to understand to read it confidently.

2.1 The First Character — File Type

The first symbol tells what kind of file it is:

Symbol	Type	Example
-	Regular file	text, code, images

d	Directory	folders
l	Symbolic link	shortcuts
c	Character device	keyboard, terminal
b	Block device	hard drive, USB

Think of it like an ID badge — it tells Linux *what kind of worker* this file is.

2.2 The Next Nine Characters — Permissions (rwx)

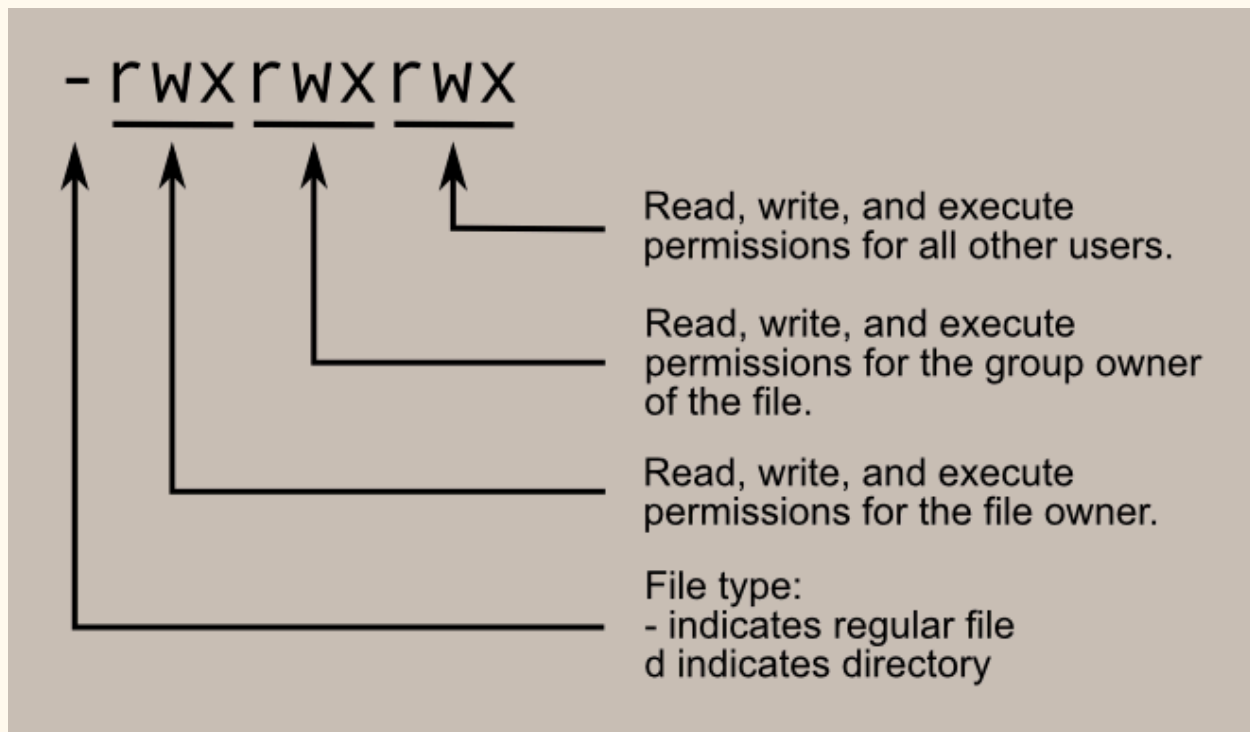
After that, you see something like **rw-r-xr--**.

These nine symbols are divided into three groups of three:

Role	Example	Meaning
User (Owner)	rw-	what the file creator can do
Group	r-x	what team members can do
Others	r--	what everyone else can do

Each role gets three keys:

- **r** → Read
- **w** → Write
- **x** → Execute



2.3 The 3rd field — Hidden Secrets

After the permission bits, sometimes you'll see a **dot (.)** or a **plus (+)**.

Symbol	Meaning
.	File has extended attributes (xattr)
+	File has Access Control Lists (ACLs)
(blank)	No extras — plain file

- Extended attributes are metadata fields attached to a file that store extra information the filesystem doesn't normally track.
- ACLs extend the traditional permission system so you can grant specific users or groups additional rights — beyond the file's main owner and group.

2.4 Link Count — The File's Popularity

When we talk about “**Link Count**” in Linux (`ls -l` 4th field), it **starts at 1** — because the very first link to any file is its **own name** in the directory.

Let's put it clearly:

Type	What It Means	Default Link Count
Regular File	The file's own name points to its inode	1
Directory	Includes itself (.) and its parent (..)	2 + number of subdirectories

Why 2 for directories?

- `.` → the directory itself
- `..` → its parent directory

So, if a folder contains **3 subfolders**,
`2 + 3 = 5` links total.

Remember:

A file's link count shows how many directory entries point to its inode.

If you create a hard link, that number increases because another name now points to the same file data.

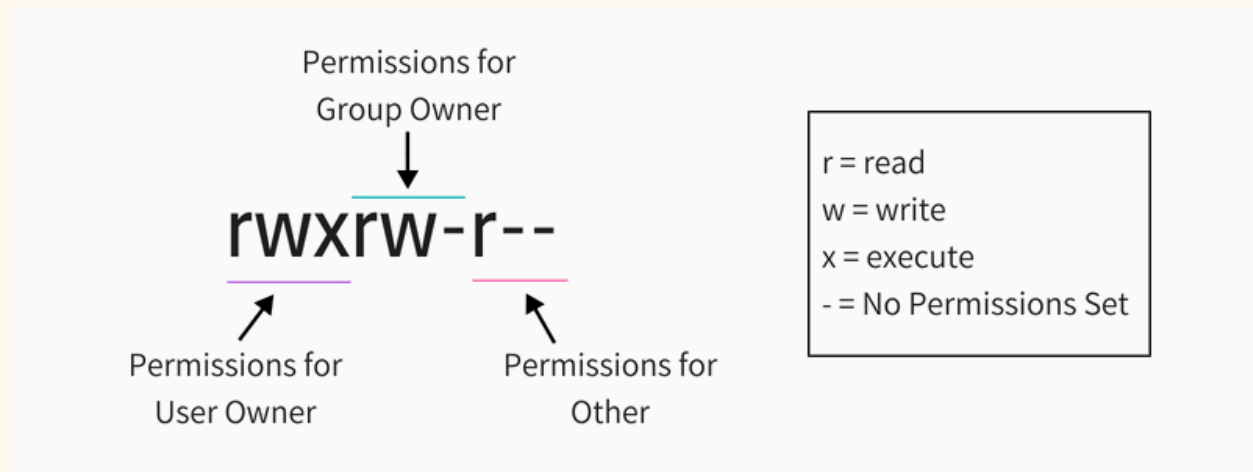
2.5 Owner and Group — Who Owns the File?

Every file in Linux has two guardians — the **Owner** and the **Group** — who decide who holds the keys to access it.

Field	Meaning
-------	---------

Owner (User)	The individual (username or UID) who created the file. Usually, this person has full control and can change its permissions.
Group	A team (identified by GID) that shares access. All members of this group follow the same permission set.

Together, **Owner** and **Group** decide who gets which keys:
r (read), **w (write)**, **x (execute)**.



2.6 · Size, Time, and Name — The File's Final Identity

The last three fields in the `ls -l` output describe the **file's physical and visible identity** — how big it is, when it was last touched, and what it's called.

Field	Meaning
File Size	The number of bytes the file occupies. Example: <code>1024</code> means the file is 1 KB in size.
Modification Time (mtime)	The last moment the file's content was changed — not just viewed, but actually edited.
File Name	The visible label that identifies the file, like <code>file.txt</code> . It's what you see in the directory listing, even though the real identity (inode) is hidden underneath.

Insight:

When you rename a file, only this **name field** changes — not its inode or metadata. That's why Linux can instantly rename even huge files without rewriting their contents.

3 · How Permissions Are Evaluated

Linux doesn't randomly pick which permissions to apply — it follows a **strict, step-by-step evaluation order** every time someone tries to access a file.

When you interact with a file, Linux checks permissions in this exact sequence:

1. **User (u)** → Are you the file's owner?
 - If yes, your access is decided by the **user permissions**, and the system **stops checking further**.
2. **Group (g)** → If you're not the owner, are you part of the file's group?
 - If yes, Linux uses the **group permissions** to decide what you can do.
3. **Others (o)** → If neither, you fall under **others** — the most restricted category.

Only **one category applies at a time** — whichever matches first.

This top-down checking order is what keeps Linux secure — even if a file is world-readable, the owner's or group's settings can override it for those who belong to higher roles.

4 · What Are File Permission Modes?

When you use the **chmod** command in Linux, you're telling the system **what kind of access each role (user, group, others)** should have for a file or directory.

There are **two ways** to describe those permissions:

1. **Symbolic mode** — using letters and symbols
2. **Numeric (Octal) mode** — using numbers (0–7)

Both do the same thing — they just express it differently.

4.1 Symbolic Mode (Letter Method)

This mode uses letters (**u**, **g**, **o**, **a**) and symbols (**+**, **-**, **=**) to describe **who** gets **what** access.

Meaning of letters

Symbol	Refers to
u	user (file owner)
g	group
o	others (everyone else)
a	all (u + g + o)

Meaning of signs

Symbol	Meaning
+	Add a permission
-	Remove a permission
=	Set exact permissions (overwrite all previous ones)

Symbolic Mode Examples

When using the **symbolic mode**, you describe permission changes to Linux using letters instead of numbers — it's like giving the system **plain-language instructions**.

Command	Meaning	Result
<code>chmod u+x file.txt</code>	Add execute permission for the file owner	Owner can now run the file
<code>chmod g-w file.txt</code>	Remove write permission for the group	Group members can no longer edit the file
<code>chmod o=r file.txt</code>	Allow others to read only	Others get <code>r--</code>
<code>chmod u=rwx,g=rx,o=</code>	Give full rights to owner, read+execute to group, none to others	Final mode: <code>rwxr-x---</code>

Think of it as talking to Linux in words:

“Give the user execute permission, take away write from the group, and let others read only.”

NOte:

Symbolic mode is great when you want to **tweak specific permissions** without changing everything — it’s intuitive and human-friendly.

4.2 Numeric (Octal) Mode (Number Method)

Numeric or octal mode expresses permissions using three digits — one for each role:

User / Group / Others

Each permission has a **number value**:

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

You **add them up** to get the total value for each role.

Combination	Calculation	Value	Meaning
---	0	0	No permissions
--x	1	1	Execute only
-w-	2	2	Write only
-wx	2+1	3	Write + Execute
r--	4	4	Read only
r-x	4+1	5	Read + Execute
rw-	4+2	6	Read + Write
rwX	4+2+1	7	Full access

Octal Mode Examples

Command: `chmod 744 file.txt`

Breakdown

Role	Value	Permission	Meaning
User	7	rwX	Full access
Group	4	r--	Read only
Others	4	r--	Read only

So the file becomes:

`rwXr--r--`

4.3 The Difference in Practice

Aspect	Symbolic Mode	Numeric Mode
Format	<code>chmod u+x file.txt</code>	<code>chmod 755 file.txt</code>
Style	Human-friendly, descriptive	Short and mathematical
Use	For adding/removing specific bits	For setting full modes quickly
Common In	Manual file adjustments	Scripts, configuration files

5 · What Ownership Really Means

In Linux, ownership is the foundation of the file security model.

Every file, directory, or device has two ownership fields that decide who the permission bits apply to.

- **User owner** → the person who created or owns the file.
- **Group owner** → a team (group) of users who can share access.

These ownerships decide which set of permissions (user, group, others) apply when someone tries to access a file.

5.1 · Dual Ownership Structure

Each file has:

1. User owner (UID) → typically the creator

- The system records this User ID (numeric value behind the username).

- Permissions under the u field (rwx for user) apply to this person.

2. Group owner (GID) → a team of users

- Members of this group get the file's group-level (g) permissions.
- Helps share files among collaborators.

3. Others → anyone else on the system not matching the above two.

Together, these form the Linux permission triad:

User (u) | Group (g) | Others (o)

5.2 · Who Can Change Ownership?

Changing who owns a file isn't something every user can do — it's a privileged operation. Linux treats ownership transfer as a security-sensitive action because it directly affects who controls access to the file.

Action	Who Can Do It	Notes
Change owner	Root only	Normal users can't "give away" their files. Only the root account (or users with sudo) can assign ownership to someone else.
Change group	Owner, if part of that group	You can switch the file's group only if you already belong to that group.
Modify permissions (chmod)	Owner or root	Controls who can read, write, or execute the file.
Delete file	Anyone with w + x on the directory	Deletion depends on directory permissions — not on who owns the file itself.

Important Note:

Even if you own a file, you can't stop someone from deleting it if the directory it lives in is world-writable (777).

That's because deletion affects the directory entry, not the file's contents.

“To protect a file from deletion, protect its directory, not just its permissions.”

5.3 · The Ownership Logic Behind the Scene

Behind every file's name, Linux quietly tracks two hidden numbers that define who really owns it.

ID Type	Meaning	Example
UID (User ID)	Identifies the file's user owner	1000 → ali
GID (Group ID)	Identifies the file's group owner	1001 → devteam

When you run `ls -l`, you see names like `ali` or `devteam`, but the filesystem doesn't actually store those names — it stores the **numeric IDs**.

These numbers are then mapped to names through `/etc/passwd` (for users) and `/etc/group` (for groups). We will discuss it later in user management.

Insight:

When a file is copied between systems, its UID and GID may point to different users — which is why understanding ownership logic is essential when managing shared or networked environments.

5.4 · Commands for Ownership Management

Now that you know what ownership means, let's look at how to change it in real life using two essential commands: `chown` and `chgrp`.

1. `chown` — Change File Owner

Used to assign a new user owner to a file or directory.

```
chown sameer file.txt
```

Makes sameer the new owner of file.txt.

You can also change both owner and group at the same time:

```
chown sameer:devteam file.txt
```

Sets User = sameer, Group = devteam

If you want to do it for an entire folder (and everything inside it)? Use the recursive option (-R):

```
chown -R sameer:devteam /project/
```

It Changes ownership for all files and subdirectories under [/project/](#).

2. `chgrp` — Change Group Ownership

If you only want to change the group owner, without affecting the file's user, use `chgrp`:

```
chgrp devteam file.txt
```

The file's group ownership changes to devteam.

For entire directories, add the -R flag:

```
chgrp -R devteam /shared/
```

Updates the group ownership for every file and subfolder inside [/shared/](#).

Permissions Reminder

Changing ownership doesn't change permissions — it only decides who the existing permissions apply to.

Example:

```
-rw-r----- sameer devteam file.txt
```

Interpretation:

- sameer → can read and write
- Members of devteam → can read
- Others → no access

Think of ownership as assigning responsibility, not granting power.

You're deciding whose keys the permission bits belong to — not what those keys can do.

5.5 · Root Privilege Required

Transferring ownership of files is a high-level administrative task, because it directly affects who controls access to system resources.

That's why only the root user (or anyone using sudo) can assign files to another user.

Example:

```
sudo chown sameer file.txt
```

The file `file.txt` now belongs to `sameer`.

Important Rule:

Normal users cannot transfer ownership of their files to others.

They can only change the group — and even that is allowed only if they are a member of the target group.

6 • Linux file vs directory permissions

In Linux, permissions behave **differently** for **files** and **directories** — because files hold *data*, and directories hold *names* (*links to inodes*).

Let's break it down simply.

6.1 File Permissions — Protect Content

File permissions decide **who can read, modify, or execute the file's contents**.

Bit	Symbol	Meaning (for files)	Example Action
4	r	Read file contents	<code>cat file.txt</code>
2	w	Modify or overwrite file	<code>echo > file.txt</code>
1	x	Execute file (if it's a script/binary)	<code>./script.sh</code>

Example:

```
-rw-r--r-- file.txt
```

- Owner: read + write
- Group: read only
- Others: read only

So only the **owner** can modify it, others can just view.

6.2 Directory Permissions — Protect *Names*

A directory doesn't store file contents — it stores **entries (filenames → inode numbers)**. So its bits mean something else:

Bit	Symbol	Meaning (for directories)	Example Action
4	r	List directory contents	<code>ls dir</code>
2	w	Modify directory entries (create/delete/rename)	<code>touch newfile,</code> <code>rm file.txt</code>
1	x	Access (traverse) directory	<code>cd dir</code> or open files by path

To delete or rename a file, you need **w + x** on the directory — not on the file itself!

6.3 How They Interact

Let's visualize a few scenarios

File Permission	Directory Permission
<code>000</code> file, <code>777</code> dir	Anyone can delete/rename file, but no one can open/read it.
<code>644</code> file, <code>755</code> dir	Everyone can read the file and list directory, only owner can modify.
<code>600</code> file, <code>700</code> dir	Only owner can see, open, modify. Others can't even list the directory.
<code>777</code> file, <code>000</code> dir	File becomes inaccessible — you can't even reach it without x on directory.

6.4 Quick truth table (If scenario: file 000, dir 777)

Actor	Read file contents	Write file contents	Delete/Rename file	Create in dir
root	Yes (ignores 000)	Yes (ignores 000)	Yes (w+x on dir)	Yes
owner	No (until <code>chmod</code>)	No (until <code>chmod</code>)	Yes (w+x on dir)	Yes
others	No	No	Yes (w+x on dir)	Yes

7 · Immutable & Append-Only Attributes (`chattr`)

Beyond the familiar `rwX` permissions managed by `chmod`, Linux gives you **deeper control** through **filesystem-level attributes**.

These attributes affect how the **kernel itself** treats a file — meaning they can override even root’s normal abilities.

7.1 · What Are File Attributes?

Think of attributes as “**special behavior flags**” stored in the filesystem’s metadata. They’re not part of the usual permission bits — they live **below** `chmod` and `chown`, directly influencing how files behave.

You can view or change them using:

```
lsattr file.txt    # View file attributes
```

```
chattr +i file.txt # Add (set) an attribute
```

These work on **ext2**, **ext3**, **ext4**, and similar Linux filesystems.

If `chmod` is like locking a door, `chattr` is like sealing it shut with concrete. Even root has to remove the seal before making changes.

7.2 The Immutable Attribute (+i)

Once marked immutable, the file is **completely locked** — even the root user can't change, rename, or delete it.

```
chattr +i important.txt
```

Now:

You can't edit or append data
You can't delete or rename it
You can't create hard links to it
But, you can still read or copy it

To unlock the file:

```
chattr -i important.txt
```

Think of +i as “total lock mode” — perfect for configuration files or security policies that should never be tampered with.

7.3 The Append-Only Attribute (+a)

The file can only **grow** — new data can be added at the end, but existing content can't be changed or removed.

```
chattr +a logfile.log
```

Now:

You can append new data using `>>`
But you can't edit or truncate the file
and you can't delete or rename it

To remove append protection:

```
chattr -a logfile.log
```

This is ideal for **security logs and audit trails**, ensuring that **past entries remain untouched** — they can only grow forward.

7.4 Viewing Attributes

To check which attributes are active, use:

```
lsattr
```

Example Output:

```
----i----- config.cfg
```

```
-----a----- audit.log
```

Here,

- **i** → immutable
- **a** → append-only

Attribute	Command	Description
Immutable (+i)	<code>chattr +i file.txt</code>	Prevents modification, deletion, or renaming — even by root.
Remove Immutable	<code>chattr -i file.txt</code>	Re-enables normal changes.
Append-only (+a)	<code>chattr +a log.txt</code>	Only allows appending new data (great for logs).
Remove Append	<code>chattr -a log.txt</code>	Restores normal write behavior.

While **chmod** controls **who** can access a file, **chattr** controls **what even root can do** with it. That's why attributes are one of the strongest safeguards in Linux for **critical system and log**

files

8 • Access Control Lists (ACLs)

Traditional Linux permissions (rwx for user, group, and others) are simple — but limited. They allow only one owner and one group. But what if you want to grant special access to another specific user without changing ownership?

That's where ACLs — Access Control Lists — come in.

8.1 • What Are ACLs?

ACLs extend the standard permission model, allowing **fine-grained access control**. You can assign **custom permissions** to any user or group, not just the file's owner or its main group.

8.2 Add or Modify an ACL

```
setfacl -m u:sameer:rw file.txt
```

-m → modify or add an entry

u:sameer:rw → give user **sameer** read and write access

Even if **sameer** isn't the file's owner or in its group, he can still read and write the file.

Apply ACLs Recursively

Apply the same ACL to a folder and everything inside it:

```
setfacl -R -m u:sameer:rw myfolder/
```

-R → recursive through all subfolders and files

sameer now has read/write access everywhere under `myfolder/`.

Default ACLs (for Future Files)

Default ACLs are automatically **inherited** by new files created inside a directory.

```
setfacl -m d:u:sameer:rw /shared/
```

`d:` = default

Every new file in `/shared` will automatically give sameer read/write access.

Check results with:

```
getfacl /shared
```

8.3 • The Mask — The Hidden Gatekeeper

The **mask** defines the **maximum effective permissions** that any ACL entry (except the owner) can have.

Example:

```
setfacl -m u:sameer:rw file.txt
```

```
setfacl -m m:r file.txt
```

sameer's ACL says `rw`, but the mask says only `r`.

Effective permission = `r` (limited by the mask).

Think of the mask like a **ceiling** — no ACL entry can go above it.

8.4 • Viewing ACLs

To check which ACLs exist on a file:

```
getfacl file.txt
```

Example Output:

```
# file: file.txt
# owner: root
# group: root
user::rw-
user:ali:rw-
group::r--
mask::rw-
other::r--
```

Interpretation:

- Normal owner permissions → `user::rw-`
- Extra ACL entry for Ali → `user:ali:rw-`
- Mask limits the maximum effective rights → `mask::rw-`
- Standard group and others still apply

8.5 · How ACLs Work with Regular Permissions

- Linux first checks standard ownership (user, group, others).
- If ACLs exist (you'll see a `+` after permission bits), it evaluates those next.
- Final access = the minimum of ACL rights and the mask.

Example:

```
ls -l
```

```
-rw-r--r--+ 1 root root file.txt
```

That **+** means the file has ACLs.

View full details with:

```
getfacl file.txt
```

Key to Note:

ACLs bridge the gap between “too open” and “too limited.”

They give system administrators precision control — perfect for multi-user environments, shared directories, and collaborative projects.

8.6 • Managing ACLs — Common Flags

When working with ACLs, you’ll often use specific flags with the **setfacl** command to modify, remove, or apply permissions efficiently.

Here’s a quick reference table every Linux learner should remember:

Flag	Meaning	Example Command
-m	Modify or add a new ACL entry	<pre>setfacl -m u:ali:rw file.txt</pre>
-x	Remove a specific ACL entry	<pre>setfacl -x u:ali file.txt</pre>
-b	Remove all ACL entries (reset file to normal permissions)	<pre>setfacl -b file.txt</pre>
-k	Remove default ACLs from a directory	<pre>setfacl -k /shared</pre>

-d	Define a default ACL for new files in a directory	<code>setfacl -m d:g:designers:r /projects</code>
-R	Apply ACLs recursively to all files and subdirectories	<code>setfacl -R -m g:qa:rwx /testdata</code>

Tip:

You can combine multiple flags in a single command — for example:

```
setfacl -Rm u:ali:rw,g:dev:rwx /data
```

Adds read/write for Ali and full access for group *dev*, recursively inside */data*.

Summary

In this lesson, we explored how Linux decides *who can access what* through ownership, permissions, and attributes. Every file in Linux carries its own identity — the **owner**, **group**, and **permission bits** — defining who can read, modify, or execute it. You also learned how commands like **chmod**, **chown**, **chgrp**, and **chattr** shape access control, while **ACLs** provide fine-grained flexibility beyond traditional *rxw* rules. Understanding these concepts is the foundation of Linux security and system administration.

Next Lesson: We will discuss **User Management and Advanced Permissions** — learning how Linux creates users, organizes groups, and enforces privilege control.