Linux Permissions

Understanding Ownership, Access Control, and Security on Linux Systems

Preface

Every file on a Linux system has a gatekeeper. Every directory, every script, every configuration file, every socket — all of them are governed by a quiet but unyielding system of rules that determines who can read, who can write, and who can execute. This system is the Linux permissions model, and it is one of the most fundamental pillars of Linux security.

Yet for many Linux users and administrators, permissions remain a source of confusion, frustration, and — too often — costly mistakes. A web application fails silently because a directory isn't executable. A cron job breaks after a routine deployment. A well-meaning administrator runs `chmod 777` to "fix" a problem and unknowingly opens a door that should have stayed locked. These scenarios play out every day on Linux systems around the world.

This book exists to make sure they don't play out on yours.

What This Book Covers

Linux Permissions: Understanding Ownership, Access Control, and Security on Linux Systems is a comprehensive, practical guide to how Linux controls access to files, directories, and system resources. It is written for system administrators, developers, DevOps engineers, and anyone who works with Linux and wants to move beyond guesswork toward genuine understanding.

We begin with the foundations — **why permissions matter** on Linux, how users and groups form the basis of ownership, and what read, write, and execute actually mean in practice. From there, we explore both symbolic and numeric (octal) modes for setting permissions, and we examine the distinct behaviors that permissions exhibit on files versus directories.

The middle chapters take you deeper into the Linux permissions landscape. You'll learn about **special permission bits** – SUID, SGID, and the sticky bit – and understand when they're essential and when they're dangerous. You'll master `umask` to control default permissions, and you'll work with **Access Control Lists (ACLs)** for the fine-grained access control that standard Linux permissions alone cannot provide.

The later chapters are where theory meets the real world. We tackle **permission troubleshooting** with systematic approaches, examine best practices for **web servers and system services** running on Linux, and introduce **auditing techniques** to verify that your permissions reflect your security intentions. We also provide an accessible introduction to **SELinux**, showing how mandatory access controls extend beyond traditional Linux permissions. A dedicated chapter on **permission anti-patterns** catalogs the most common mistakes so you can recognize and avoid them. The final chapter steps back to frame Linux permissions within the broader context of **security architecture**, pointing you toward a more holistic understanding.

Five appendices provide quick-reference material – cheat sheets for `chmod` and `chown`, an octal permission reference table, ACL command summaries, a troubleshooting checklist, and a **Linux security learning roadmap** for continuing your journey.

How to Use This Book

If you're new to Linux permissions, read the chapters in order. Each one builds on the last. If you're an experienced administrator looking to fill specific gaps, the chapters are self-contained enough to serve as targeted references. Either way, the appendices are designed to stay open on your desk — or your second monitor — while you work.

A Note of Thanks

This book was shaped by years of conversations with Linux administrators who learned permissions the hard way: through broken deployments, security incidents, and late-night troubleshooting sessions. Their experiences — and their willingness to share what went wrong — made this a better, more honest book. I'm also grateful to the broader Linux and open-source community, whose documentation, forum posts, and tireless knowledge-sharing form the bedrock on which any book like this is built.

The Goal

Linux permissions are not merely a technical detail. They are the first line of defense on every Linux system you manage. My goal is that by the final page, you won't just know *how* to set permissions — you'll understand *why* you're setting them, *what* could go wrong if you don't, and *how* to verify that your systems are as secure as you believe them to be.

Let's begin.

Bas van den Berg

Table of Contents

Chapter 1: Why Permissions Matter

Linux is, at its very core, a multi-user operating system. From the moment it was conceived as a reimagining of Unix principles, Linux was designed with the assumption that more than one person would be using the same machine, accessing the same resources, and running programs simultaneously. This fundamental design philosophy gave rise to one of the most critical and elegant subsystems in all of computing: the Linux permissions model. Before we dive into the technical mechanics of how permissions work, how to read them, and how to change them, it is essential that we first understand *why* they exist, what problems they solve, and what would happen in a world without them. This chapter lays the philosophical and practical groundwork for everything that follows in this book.

The Multi-User Nature of Linux

To truly appreciate why permissions matter, you must first understand the environment in which Linux operates. Unlike some operating systems that were originally designed for a single user sitting at a single desk, Linux inherited its DNA from Unix, which was born in the shared computing environments of Bell Labs in the early 1970s. In those days, a single powerful computer served dozens or even hundreds of users through terminals. Every user had their own login, their own files, and their own running processes, but they all shared the same physical hardware, the same kernel, and the same filesystem.

This multi-user heritage is not merely a historical curiosity. It is alive and well in modern Linux deployments. Consider a Linux web server hosting websites for multiple clients. Each client may have their own user account, their own directory of web files, and their own set of scripts and configurations. On a university Linux server, hundreds of students might log in to complete programming assignments, each needing a private workspace where their code cannot be seen or copied by classmates. In a corporate environment, different departments might share a single Linux file server, with accounting data that must remain invisible to the marketing team and vice versa.

Even on a single-user Linux desktop, the multi-user model is at work in ways that are not immediately obvious. The system itself runs dozens of services under different user accounts. Your web server might run as the user `www-data`, your database as `mysql` or `postgres`, your print spooler as `lp`, and your mail server as `mail`. Each of these service accounts is a "user" in the eyes of the Linux kernel, and the permissions system ensures that a compromised web server cannot read your database files, and a misbehaving print spooler cannot modify your system configuration.

The following table illustrates some common system users found on a typical Linux installation and the purpose each one serves:

System User	**Typical Purpose**	**Home Directory**	**Why It Exists**
root	Superuser with full system access	/root	Performs administrative tasks that require unrestricted access
www-data	Runs the Apache or Nginx web server	/var/www	Isolates web server processes from the rest of the system
mysql	Runs the MySQL database server	/var/lib/mysql	Prevents database files from being accessed by other services

nobody	Used for unprivileged operations	/nonexistent	Provides a user with minimal permissions for sandboxing
mail	Handles mail delivery subsystem	/var/mail	Separates mail processing from other system functions
lp	Manages printing services	/var/spool/lpd	Restricts printer operations to a dedicated account
sshd	Manages SSH daemon privilege separation	/run/sshd	Isolates SSH connection handling for security

You can see these users on your own system by examining the password file:

```
cat /etc/passwd
```

Each line in this file represents a user account. You will likely find dozens of entries, most of which are system accounts rather than human users. Every single one of these accounts interacts with the permissions system every time it reads a file, writes data, or executes a program. This is the landscape in which Linux permissions operate, and it is far more complex and populated than most people realize.

What Happens Without Permissions

To understand why something matters, it is often helpful to imagine what the world would look like without it. Let us conduct a thought experiment. Imagine a Linux system with no permissions whatsoever. Every file on the system, from the kernel itself to your personal photographs, is equally accessible to every user and every process. Anyone can read anything, write to anything, and execute anything.

The consequences would be immediate and catastrophic.

First, consider **privacy**. On a shared system, any user could read any other user's files. Personal documents, private keys, email, browser history, saved passwords, and financial records would all be open books. A curious or malicious user could browse through anyone's home directory at will. In a business context, this would mean that trade secrets, employee records, salary information, and strategic plans would be accessible to every person with a login.

Second, consider **integrity**. Without permissions, any user could modify any file. A disgruntled employee could alter financial records. A careless student could accidentally overwrite the operating system's configuration files. A buggy script could corrupt another user's data. The system itself would be perpetually unstable because any process could modify any system file. The configuration that tells your system how to boot, how to resolve network addresses, or how to authenticate users could be changed by anyone at any time.

Third, consider **security**. The password file that stores hashed credentials, the SSH keys that allow remote access, the certificates that secure web traffic, and the firewall rules that protect the network would all be modifiable by any user. An attacker who gained access to any account, no matter how unprivileged, would immediately have full control of the entire system. There would be no containment, no damage limitation, and no defense in depth.

Fourth, consider **system stability**. Linux relies on specific files being owned by specific users and having specific permissions. The kernel expects certain files to be read-only. The package manager expects certain directories to have certain ownership. System services expect their configuration files to be inaccessible to unauthorized users. Without permissions, these assumptions would all be violated, and the system would behave unpredictably.

Let us make this concrete with a real example. On a properly configured Linux system, the file `/etc/shadow` contains the hashed passwords for all user accounts. Let us examine its permissions:

```
ls -l /etc/shadow
```

The output will look something like this:

```
-rw-r----- 1 root shadow 1234 Jan 15 10:30 /etc/shadow
```

This tells us that only the root user can read and write this file, and members of the shadow group can read it. No other user on the system can access it at all. This is a deliberate and critical security measure. If any user could read this file, they could take the password hashes offline and attempt to crack them using brute-force tools. If any user could write to this file, they could change any user's password, including root's, and take over the system entirely.

Now consider another critical file:

```
ls -l /etc/passwd
```

```
-rw-r--r-- 1 root root 2345 Jan 15 10:30 /etc/passwd
```

This file is world-readable because programs need to look up user information, but it is only writable by root. The distinction between `/etc/passwd` being readable by everyone and `/etc/shadow` being readable only by root and the shadow group is a perfect example of the permissions system making nuanced, file-by-file security decisions.

The Three Pillars of Linux Security Through Permissions

The Linux permissions model rests on three fundamental pillars that work together to create a comprehensive security framework. Understanding these pillars is essential before we explore the technical details in later chapters.

The First Pillar: Ownership

Every file and every directory on a Linux system has an owner and a group. The owner is typically the user who created the file, and the group is typically the primary group of that user. Ownership answers the question: "Who does this belong to?" This is not merely a label. Ownership determines which set of permission rules apply to which users. When you access a file, the kernel first checks whether you are the owner, then whether you belong to the file's group, and finally falls back to the "other" category. This three-tier classification is the foundation upon which all permission decisions are made.

You can see the ownership of files in your home directory with:

```
ls -la ~/
```

Every file listed will show two names: the owner and the group. These two pieces of information, combined with the permission bits, determine exactly who can do what with each file.

The Second Pillar: Access Types

Linux defines three fundamental types of access: read, write, and execute. These three access types have slightly different meanings depending on whether they are applied to a file or a directory, which is a nuance we will explore in great detail in later chapters. For now, understand that read means the ability to see the contents, write means the ability to modify or delete, and execute means the ability to run a file as a program or enter a directory. Every combination of ownership category and access type creates a specific permission that can be independently granted or denied.

The following table summarizes the basic access types:

Access Type	Symbol	Effect on Files	Effect on Directories
Read	r	View the contents of the file	List the contents of the directory
Write	w	Modify or delete the file contents	Create, delete, or rename files within the directory
Execute	x	Run the file as a program or script	Enter the directory and access its contents

The Third Pillar: Enforcement by the Kernel

Permissions in Linux are not suggestions or guidelines. They are enforced by the kernel itself, which is the lowest and most authoritative layer of the operating system. When a process attempts to open a file, the kernel checks the permissions before allowing the operation. This check cannot be bypassed by user-space programs. It does not matter how clever a program is or what tricks it attempts. If the kernel says no, the answer is no. The only exception is the root user, who traditionally bypasses most permission checks, which is precisely why gaining root access is the ultimate goal of any attacker and why protecting it is the ultimate responsibility of any administrator.

You can observe the kernel enforcing permissions with a simple experiment. Create a file and remove all permissions from it:

```
touch /tmp/testfile
echo "secret data" > /tmp/testfile
chmod 000 /tmp/testfile
cat /tmp/testfile
```

As a regular user, the last command will produce:

```
cat: /tmp/testfile: Permission denied
```

This "Permission denied" message comes directly from the kernel. The `cat` program asked the kernel to open the file, the kernel checked the permissions, found that the requesting user had no read access, and refused the operation. This is kernel-level enforcement in action.

To clean up after this experiment:

```
chmod 644 /tmp/testfile
rm /tmp/testfile
```

Real-World Scenarios Where Permissions Save the Day

Understanding permissions in the abstract is valuable, but seeing them in action makes their importance visceral and real. Here are several scenarios drawn from real-world Linux administration where the permissions system prevents disaster.

Scenario One: The Shared Web Server

A hosting company runs a Linux server with 50 client websites. Each client has their own user account and their own directory under `/var/www/`. Client A's files are owned by `clienta:clienta` with permissions `750`, meaning Client A can read, write, and execute, the group can read and execute, and everyone else has no access at all. When Client B tries to access Client A's files, the kernel denies the request. Even if Client B discovers a vulnerability in their own web application that allows arbitrary file reading, the permissions system prevents the attacker from pivoting to other clients' data.

```
# Example of setting up isolated client directories
mkdir /var/www/clienta
chown clienta:clienta /var/www/clienta
chmod 750 /var/www/clienta

mkdir /var/www/clientb
chown clientb:clientb /var/www/clientb
```

```
chmod 750 /var/www/clientb
```

Scenario Two: The Careless Script

A system administrator writes a cleanup script that is supposed to delete temporary files from `/tmp/`. Due to a typo, the script contains `rm -rf / tmp/` instead of `rm -rf /tmp/`. On a system where the script runs as an unprivileged user, the permissions system prevents the script from deleting system files, configuration files, and other users' data. The damage is contained to files that the unprivileged user owns. This is why experienced administrators never run scripts as root unless absolutely necessary, and why the principle of least privilege is a cornerstone of Linux security.

Scenario Three: The Compromised Service

An attacker exploits a vulnerability in a web application running on a Linux server. The web application runs as the `www-data` user. Because of the permissions system, the attacker can only access files that `www-data` is permitted to read. They cannot read `/etc/shadow` to obtain password hashes. They cannot modify `/etc/ssh/sshd_config` to weaken SSH security. They cannot install a rootkit in `/usr/bin/`. The permissions system has transformed a complete system compromise into a limited breach that affects only the web application's own files.

The Principle of Least Privilege

One concept that runs like a thread through every aspect of Linux permissions is the principle of least privilege. This principle states that every user, every process, and every service should have only the minimum permissions necessary to perform its intended function, and nothing more. This principle is not unique to Linux, but Linux's permissions system provides the tools to implement it with precision.

When you create a new file, the default permissions should not be more generous than necessary. When you set up a new service, it should run under a dedicated user account with access only to the files it needs. When you grant a user access to a system, they should receive only the permissions required for their role.

The following table shows how the principle of least privilege applies to common Linux scenarios:

Scenario	**Least Privilege Approach**	**Overly Permissive Approach**	**Risk of Overly Permissive**
Web server files	Owned by root, readable by www-data	Owned by www-data with write access	Attacker can modify website content
User home directories	Mode 700, accessible only to owner	Mode 755, readable by everyone	Other users can browse private files
System configuration	Mode 644 or 600, owned by root	Mode 666, writable by everyone	Any user can alter system behavior
SSH private keys	Mode 600, readable only by owner	Mode 644, readable by group and others	Keys can be stolen and used for unauthorized access
Database files	Owned by database user, mode 700	World-readable	Sensitive data exposed to all users

You can check the permissions on your own SSH keys to see if they follow the principle of least privilege:

```
ls -la ~/.ssh/
```

If your private key (`id_rsa` or `id_ed25519`) has permissions more open than `600`, the SSH client itself will refuse to use it and display a warning. This is a perfect example of a program enforcing the principle of least privilege on behalf of the user.

```
# Correct permissions for SSH private keys
chmod 600 ~/.ssh/id_rsa
chmod 600 ~/.ssh/id_ed25519

# Correct permissions for the .ssh directory
chmod 700 ~/.ssh

# Correct permissions for the authorized_keys file
chmod 600 ~/.ssh/authorized_keys
```

Setting the Stage for What Comes Next

This chapter has established the "why" of Linux permissions. We have seen that Linux is inherently a multi-user system, that the absence of permissions would lead to chaos, that the permissions model rests on three pillars of ownership, access types, and kernel enforcement, and that the principle of least privilege guides how permissions should be applied in practice.

In the chapters that follow, we will move from philosophy to practice. We will learn how to read permission strings, how to use `chmod` to change permissions, how to use `chown` and `chgrp` to change ownership, how octal notation works, what special permissions like setuid, setgid, and the sticky bit do, and how Access Control Lists extend the basic permissions model for more complex scenarios. Every technical detail we explore will be grounded in the understanding we have built here: that permissions are not bureaucratic obstacles or arbitrary restrictions, but essential safeguards that make Linux the secure, stable, and trustworthy operating system that powers the majority of the world's servers, supercomputers, and embedded devices.

Exercises

The following exercises will help reinforce the concepts covered in this chapter. Complete each one on a Linux system to build hands-on familiarity with the topics discussed.

Exercise 1: Exploring System Users

Run the following command and count how many user accounts exist on your system. Identify which ones are human users and which are system accounts. System accounts typically have user IDs below 1000 on most modern Linux distributions.

```
cat /etc/passwd | wc -l
awk -F: '$3 < 1000 {print $1, $3}' /etc/passwd
awk -F: '$3 >= 1000 {print $1, $3}' /etc/passwd
```

Write down the total number of accounts, the number of system accounts, and the number of human accounts. Consider why so many system accounts exist and what role each one plays.

Exercise 2: Examining Critical File Permissions

Examine the permissions on the following critical system files and record what you observe. For each file, note who the owner is, what group it belongs to, and what the permission string says.

```
ls -l /etc/passwd
ls -l /etc/shadow
ls -l /etc/group
ls -l /etc/hostname
ls -l /etc/sudoers
```

For each file, answer these questions: Who can read this file? Who can write to it? Why do you think these specific permissions were chosen?

Exercise 3: Witnessing Permission Denial

Create a test file, write some content to it, then remove all permissions and attempt to read it. Observe the error message. Then restore the permissions and clean up.

```
echo "This is a test of Linux permissions" > /tmp/permission_test
chmod 000 /tmp/permission_test
cat /tmp/permission_test
chmod 644 /tmp/permission_test
rm /tmp/permission_test
```

Record the exact error message you receive. This message originates from the kernel's permission enforcement mechanism.

Exercise 4: Investigating Your Own Permissions

Examine the permissions on your home directory and its contents. Consider whether the permissions follow the principle of least privilege.

```
ls -la ~/
ls -ld ~/
stat ~/
```

The `stat` command provides detailed information about a file or directory, including its permissions in both symbolic and octal notation. Familiarize yourself with its output, as we will use it extensively in later chapters.

Exercise 5: Thinking About Least Privilege

Without making any changes to your system, identify three files or directories where you believe the permissions could be made more restrictive without breaking functionality. Write down what the current permissions are, what you would change them to, and why. This exercise develops the security mindset that is essential for effective Linux administration.

Note: Throughout these exercises, be careful not to change permissions on system files unless you fully understand the consequences. Working in `/tmp/` or in your own home directory is safe for experimentation. Modifying permissions on system files can render your system unbootable or insecure. When in doubt, observe and record rather than modify.

The understanding you build in this chapter is the foundation upon which all subsequent chapters rest. Permissions are not an advanced topic to be learned later. They are a fundamental aspect of how Linux works, and every command you run, every file you create, and every service you configure interacts with the permissions system. By understanding why permissions matter, you are prepared to learn how they work.

Chapter 2: Users, Groups, and Ownership

Linux is, at its very core, a multi-user operating system. This fundamental characteristic shapes everything about how it manages security, allocates resources, and controls access to files and processes. Before you can truly understand permissions in Linux, you must first understand the actors involved: users, groups, and the concept of ownership. These three pillars form the foundation upon which the entire Linux permission model is built. Without a clear grasp of who users are, how groups organize them, and what ownership means in the context of files and directories, any attempt to manage permissions will feel like navigating a dark room without a flashlight. This chapter will illuminate each of these concepts with depth, practical examples, and exercises that will transform your understanding from theoretical to operational.

Every action taken on a Linux system is performed in the context of a user. When you type a command in the terminal, when a web server responds to a request, when a scheduled task runs at midnight, there is always a user identity associated with that action. Linux does not allow anonymous activity. Even processes that appear to run "in the background" with no human interaction are tied to a specific user account. This design philosophy is not accidental. It is the deliberate result of decades of Unix and Linux development, where accountability and isolation between users were considered essential properties of a secure and stable operating system.

The Root User and Regular Users

At the top of the user hierarchy sits the root user, also known as the superuser. The root user has a User ID (UID) of 0, and this single number grants it unrestricted access to every file, every process, and every configuration on the system. The root user can read any file regardless of its permissions, kill any process regardless of who owns it, and modify any system configuration regardless of how it is protected. This level of power is both necessary and dangerous. It is necessary because someone must be able to administer the system, install software, configure network interfaces, and manage hardware. It is dangerous because a single mistake made as root can render the entire system unusable, corrupt data, or open security vulnerabilities.

Regular users, by contrast, operate within carefully defined boundaries. A regular user typically has a UID starting from 1000 on most modern Linux distributions such as Ubuntu, Fedora, and Debian. Each regular user has a home directory, usually located at `/home/username`, where they can create, modify, and delete files freely. Outside of their home directory, their ability to interact with the system is governed by permissions, which we will explore in great detail throughout this book.

Between root and regular users, there exists a category often called system users or service users. These are accounts created specifically to run services and daemons. For example, the `www-data` user might run the Apache web server, the `mysql` user might run the MySQL database server, and the `nobody` user might be used for processes that require minimal privileges. System users typically have UIDs between 1 and 999, they usually do not have a login shell, and they do not have a traditional home directory. Their purpose is to provide isolation: if a web server is compromised, the attacker gains only the privileges of the `www-data` user, not the privileges of root.

Let us examine the key files that store user information on a Linux system.

The /etc/passwd File

The `/etc/passwd` file is the primary database of user accounts on a Linux system. Despite its name, it no longer stores passwords on modern systems. Each line in this file represents a single user account and contains seven fields separated by colons.

Consider this example entry:

```
john:x:1001:1001:John Smith,Room 101,555-1234:/home/john:/bin/
bash
```

The following table explains each field in detail:

Field Position	Field Name	Example Value	Explanation
1	Username	john	The login name used to identify the user on the system. It must be unique and is typically lowercase.
2	Password Placeholder	x	Historically contained the encrypted password. The "x" indicates the password is stored in /etc/shadow instead.
3	User ID (UID)	1001	A unique numerical identifier for the user. The system uses this number internally rather than the username.
4	Group ID (GID)	1001	The numerical ID of the user's primary group. This corresponds to an entry in /etc/group.

5	GECOS Field	John Smith,Room 101,555-1234	A comment field containing additional information about the user such as full name, office location, and phone number.
6	Home Directory	/home/john	The absolute path to the user's home directory. This is the directory the user is placed in upon login.
7	Login Shell	/bin/bash	The program that runs when the user logs in. For interactive users, this is typically a shell like bash or zsh.

You can view the contents of this file by running:

```
cat /etc/passwd
```

To search for a specific user, you can use the `grep` command:

```
grep "john" /etc/passwd
```

The /etc/shadow File

The `/etc/shadow` file stores the actual encrypted passwords and password aging information. This file is readable only by root, which is a critical security measure. An example entry looks like this:

```
john:
$6$rounds=5000$saltsalt$hashedpasswordhere:19500:0:99999:7:::
```

Field Position	Field Name	Explanation
1	Username	Must match the username in /etc/passwd

2	Encrypted Password	The hashed password. A value of "!" or "*" means the account is locked.
3	Last Password Change	Days since January 1, 1970 when the password was last changed
4	Minimum Password Age	Minimum number of days between password changes
5	Maximum Password Age	Maximum number of days the password is valid
6	Warning Period	Number of days before expiration that the user is warned
7	Inactivity Period	Number of days after expiration before the account is disabled
8	Account Expiration	Date when the account expires, expressed as days since January 1, 1970
9	Reserved	Reserved for future use

Creating and Managing Users

Linux provides several commands for managing user accounts. The most commonly used are `useradd`, `usermod`, and `userdel`.

To create a new user with a home directory and a default shell:

```
sudo useradd -m -s /bin/bash -c "Jane Doe" jane
```

The flags used here deserve explanation:

Flag	Meaning
-m	Create the user's home directory if it does not exist
-s /bin/bash	Set the user's login shell to /bin/bash
-c "Jane Doe"	Set the GECOS comment field to the user's full name

After creating the user, you should set a password:

```
sudo passwd jane
```

To modify an existing user, the `usermod` command is used. For example, to change a user's shell:

```
sudo usermod -s /bin/zsh jane
```

To lock a user account, preventing login:

```
sudo usermod -L jane
```

To unlock the account:

```
sudo usermod -U jane
```

To delete a user and their home directory:

```
sudo userdel -r jane
```

Note: The `-r` flag with `userdel` removes the user's home directory and mail spool. Without this flag, the home directory remains on the filesystem, which can create orphaned files that still reference the deleted user's UID.

Understanding Groups

Groups in Linux serve as a mechanism for organizing users and managing collective access to resources. Rather than assigning permissions to each user individually, which would be tedious and error-prone on a system with hundreds of users, Linux allows you to assign permissions to a group and then add users to that group. Every user on a Linux system belongs to at least one group, known as their primary group. They may also belong to additional groups, known as supplementary groups or secondary groups.

When a user creates a file, that file is owned by the user and by the user's primary group. This is an important detail that directly affects how permissions work in practice.