ADMIN:DOCS

# chmod on Linux: from basics to best practices

Understand Linux file permissions from the ground up: inode bitmasks, numeric and symbolic notation, SUID/SGID, the sticky bit, umask, POSIX ACLs, capabilities, Ansible and container hardening.

**Category:** Linux beginners **Published:** 19.09.2024 **Last reviewed:** 08.09.2026

Anyone taking the first steps on the Linux command line eventually hits the same frustrating moment: you write your first shell script, type `./deploy.sh` with expectation — and the shell answers dryly with `bash: ./deploy.sh: Permission denied`. Or you try to open a configuration file and fail with `Permission denied`.

At first this hurdle feels like pure harassment. In fact it is the most important security foundation of the entire system. Linux was designed from the start as a real **multi-user system**. While historical single-user systems such as MS-DOS granted every program unrestricted access to the entire disk, Linux enforces a strict separation of powers. No file simply “belongs to the computer” — every document, every directory and every service process is subject to fixed ownership of owner and group.

This is where the `chmod` command (*Change Mode*) comes in. Together with the basic filesystem and navigation commands you use it to control precisely who may read, edit or execute files as a program — and you protect the system against mistakes as well as against unwanted access.

The permission model is covered step by step in full:

From the practical decoding of the line `-rwxr-xr-x` in `ls -l`, through the decisive difference between file and directory permissions, to the safe use of octal and symbolic notation. Then we look under the hood: we examine inode bitmasks, kernel checks and special bits (SUID, SGID, sticky bit) and apply this foundation directly in real scenarios for web servers, SSH hardening,

Source: https://admindocs.de/en/linux-beginners/chmod-on-linux-basics-to-best-practices

systemd sandboxing and containers.

# Historical origin and evolution of Unix permission models

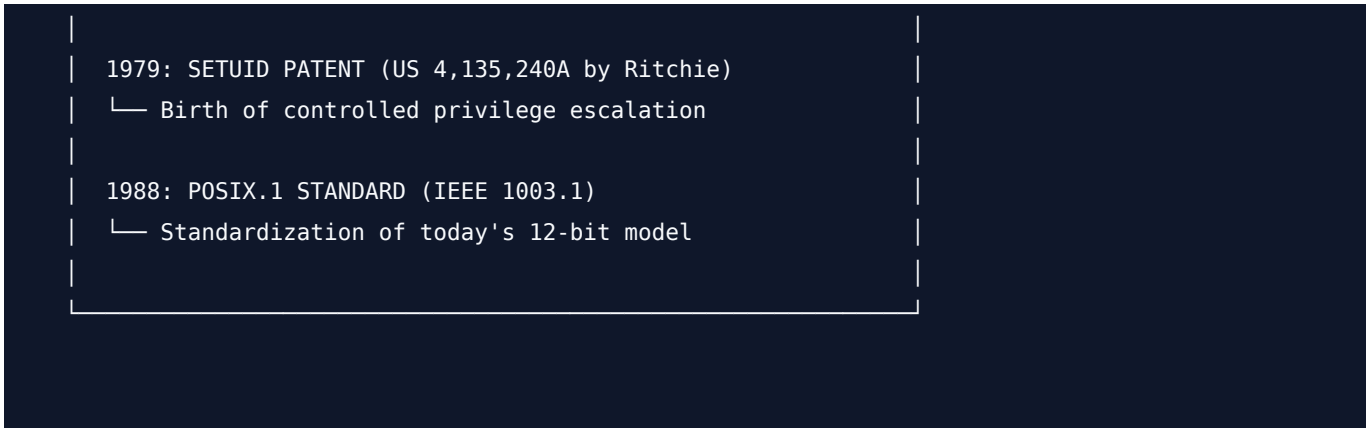
## Origin: Unix V1 at Bell Labs (1971)

The `chmod` command is one of the oldest and most persistent tools in computing history. On **3 November 1971**, **Ken Thompson** and **Dennis Ritchie** at Bell Laboratories published the first edition of Unix (*Unix V1*) on a DEC PDP-11. Already in that first version, `chmod` was a core part of the operating system.

The original permission model was still far more simplistic than today's POSIX standard:

- There were not yet user groups in the modern sense.
- Permissions in the filesystem were divided only into *user bits*, *non-user bits* and a global *execution bit*.
- The whole system was designed for a handful of researchers working together on text-processing and programming tools.






## From the Multics heritage to the Unix philosophy

Unix arose as a pragmatic counter-design to the highly complex **Multics** operating system ( *Multiplexed Information and Computing Service*), on which Bell Labs had previously collaborated with MIT and General Electric. While Multics already used comprehensive hierarchical access control lists (ACLs), dynamic rule sets and an eight-level ring security model in the late 1960s, Thompson and Ritchie deliberately chose a radical simplification:

- **Multics approach:** Maximum granularity and dynamic rule sets, which however meant enormous memory and CPU cost and were hard for administrators to oversee.
- **Unix approach:** The split into **Owner**, **Group** and **Others** (see also our guide to advanced user management) — a model that fits into the filesystem inode with minimal memory cost and reliably covers 99 % of everyday use cases.

## Evolution to the POSIX standard and Dennis Ritchie's SetUID patent

With the release of Unix Version 4 (1973), in which the kernel was rewritten in C, and later standardization by IEEE POSIX (POSIX.1 / IEEE 1003.1), the three-level octal scheme with read, write and execute bits became established.

A milestone was Dennis Ritchie's invention of the **SetUID mechanism**. To let unprivileged users change their own password (which required write access to the protected system file `/etc/passwd` and later `/etc/shadow`), Ritchie filed the SetUID concept for a patent in 1973 (granted in 1979 as US patent 4,135,240A). Bell Labs made this patent available to the entire computer industry without self-interest, and it remains the basis for tools such as `passwd`, `sudo` and `su`.

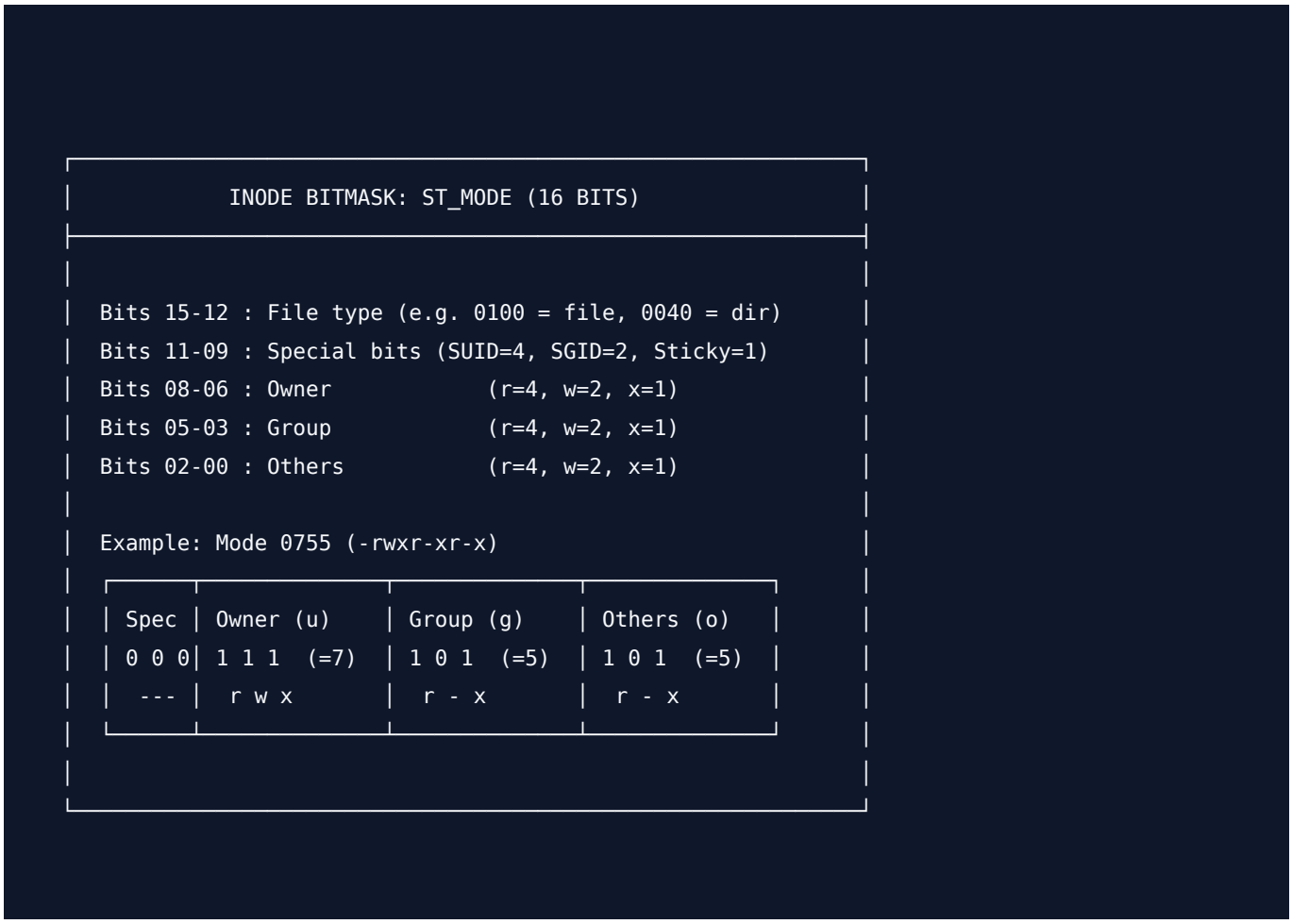
Source: https://admindocs.de/en/linux-beginners/chmod-on-linux-basics-to-best-practices

# The POSIX permission model under the hood

## How Linux stores permissions in the filesystem

File permissions in Linux are not stored in the directory entry itself, but in the filesystem metadata — the **inode** (*index node*). A directory entry is merely a lookup table of filename and inode number.

In the inode the Linux kernel reserves a 16-bit field named `st_mode` (defined in the C header `<sys/stat.h>`). This field encodes both the file type (for example regular file, directory, symlink, block device) and all access rights:



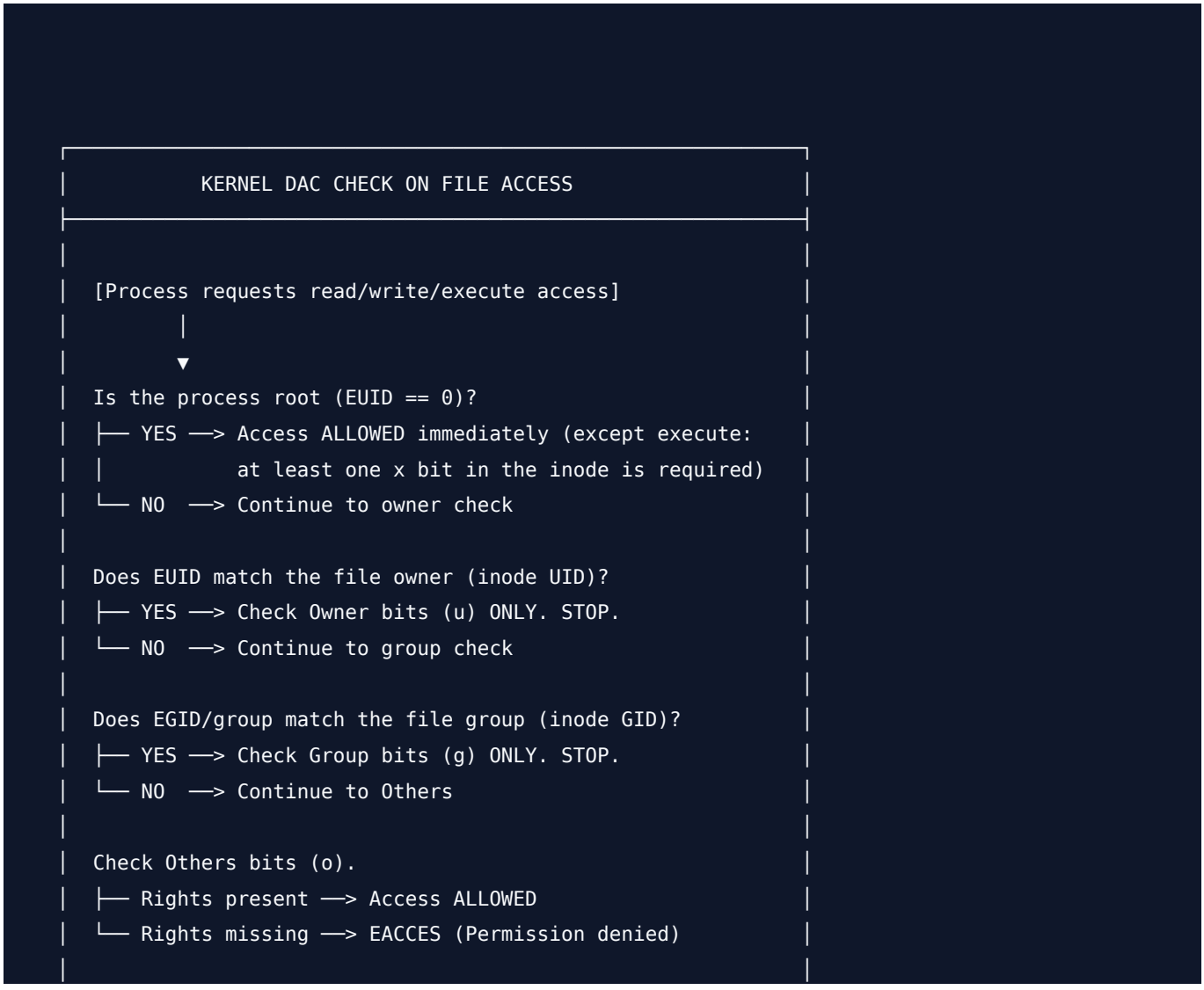

The upper 4 bits (bits 15–12) define the file type according to the POSIX standard:

- `0100` (octal `0100000` / `S_IFREG`): Regular file
- `0040` (octal `0040000` / `S_IFDIR`): Directory

- `0120` (octal `0120000` / `S_IFLNK`): Symbolic link
- `0060` (octal `0060000` / `S_IFBLK`): Block device (e.g. `/dev/sda`)
- `0020` (octal `0020000` / `S_IFCHR`): Character device (e.g. `/dev/tty`)
- `0010` (octal `0010000` / `S_IFIFO`): Named pipe (FIFO)
- `0140` (octal `0140000` / `S_IFSOCK`): UNIX domain socket

## The kernel check on file access (discretionary access control)

When a user process tries to open a file (via the system call `sys_open` or `sys_openat`), the Linux kernel performs a **discretionary access control (DAC)** check through the VFS function `generic_permission()`:



```
└──────────────────────────────────────────────────────────┘
```

> ⚠ **Fundamental principle:** The permission check stops at the first match! If the file owner has rights `000` (no rights) but the group has `777`, the owner still cannot read the file — even if they are a member of that group.

## System calls in the background: chmod(), fchmod(), fchmodat()

When you run `chmod` in the terminal, the program uses standardized C system calls from `<sys/stat.h>` under the hood:

```
#include <sys/stat.h>
#include <fcntl.h>

// 1. Classic call by file path
int chmod(const char *pathname, mode_t mode);

// 2. Descriptor-based call (avoids race conditions)
int fchmod(int fd, mode_t mode);

// 3. Relative call against a directory descriptor (POSIX.1-2008 / POSIX.1-2024)
int fchmodat(int dirfd, const char *pathname, mode_t mode, int flags);
```

Modern daemons and build systems prefer `fchmod()`, because opening a file descriptor first ensures that the permission change is applied atomically to the exact file, without an attacker placing a symlink at the path in between (*TOCTOU: Time-of-Check to Time-of-Use attack*).

## Inspecting inode metadata with the stat command

With the `stat` tool you can inspect inode data directly on the command line:

**□□ Practical example:**

Source: https://admindocs.de/en/linux-beginners/chmod-on-linux-basics-to-best-practices

We inspect the metadata of the protected shadow password file on the console and filter specifically for octal and symbolic values:

```
# Show detailed inode information
stat /etc/shadow

# Print only the octal value and filename
stat -c "%a %n" /etc/shadow

# Print symbolic notation, UID, GID and filename
stat -c "%A %U(%u) %G(%g) %n" /etc/shadow
```

**Sample output:**

```
  File: /etc/shadow
  Size: 1420       Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d Inode: 131075      Links: 1
Access: (0640/-rw-r-----)  Uid: (    0/    root)   Gid: (   42/  shadow)
Access: 2026-08-28 01:00:00.000000000 +0200
Modify: 2026-08-27 15:30:12.000000000 +0200
Change: 2026-08-27 15:30:12.000000000 +0200
 Birth: 2024-04-20 10:15:00.000000000 +0200
```

## Elementary permissions: files vs. directories in detail

A common misunderstanding among Linux beginners is transferring the rights logic of regular files onto directories. The bits `r`, `w` and `x` have a fundamentally different meaning on folders:

| Right | Meaning on regular files | Meaning on directories |
|---|---|---|

| | | |
|---|---|---|
| **Read ( `r` , Read, 4)** | Allows reading the file contents (e.g. with `cat` , `less` , an editor). | Allows listing the filenames contained in the folder (e.g. with `ls` ). |
| **Write ( `w` , Write, 2)** | Allows modifying or overwriting the contents of a file. | Allows **creating, renaming and deleting files** inside the folder! |
| **Execute ( `x` , eXecute, 1)** | Allows running the file as a binary program or shell script. | **Traverse/search bit:** Allows entering the directory ( `cd` ) and accessing inodes of the files inside it. |

## The trap of directory write rights

Anyone who has write rights ( `w` ) on a directory can **delete or rename** files in it, even if the affected file is write-protected ( `chmod 400` ) or belongs to a completely different user!

Deleting a file does not change the file inode, but the directory inode (which removes the entry from the file list). That is why global shared directories such as `/tmp` have the **sticky bit**.

**□□ Practical example:**

Here we demonstrate the difference between file and directory permissions in a test folder:

```
# Demonstration: write right on a directory allows deleting foreign files
mkdir /tmp/testdir
chmod 777 /tmp/testdir

# User Alice creates a write-protected file:
touch /tmp/testdir/alice_secret.txt
chmod 400 /tmp/testdir/alice_secret.txt

# User Bob cannot read or edit the file, but can DELETE it without trouble:
rm /tmp/testdir/alice_secret.txt  # Works, because Bob has write right on /tmp/testdir!
```

## The traverse bit ( `x` ) on directories in practice

To access a file such as `/var/log/nginx/access.log` , a process needs not only read rights on the file itself, but **execute rights ( `x` ) on every parent directory in the path**:

- `/` (root directory) must have `x`.
- `/var/` must have `x`.
- `/var/log/` must have `x`.
- `/var/log/nginx/` must have `x`.

If even a single folder in the path lacks the `x` bit for the calling user, the kernel denies access with `EACCES (Permission denied)` — even if the file itself is set to `777`!

With the `namei` tool you can list the permission chain of a path without gaps:

**□□ Practical example:**

We check the entire directory hierarchy of a web-server log with `namei`:

```
# Check the permission chain for a file path
namei -l /var/log/nginx/access.log
```

**Sample output:**

```
f: /var/log/nginx/access.log
drwxr-xr-x root root /
drwxr-xr-x root root var
drwxrwxr-x root syslog log
drwx------ www-data adm nginx
-rw-r----- www-data adm access.log
```

## Special case: symlinks and hard links

Symbolic links (*symlinks*) always have the permission mask `lrwxrwxrwx` (`0777`) in the filesystem. On operations on symlinks (such as `cat link.txt` or `nano link.txt`) the Linux kernel always

follows the target file and checks only that inode's permissions.

Hard links, by contrast, share the same physical inode with the original file. Changing permissions via `chmod` on one hard link automatically changes the rights of all other hard links to the same inode.

## Numeric (octal) vs. symbolic notation

`chmod` supports two notation forms: the mathematical-binary **octal notation** and the readable **symbolic notation**. As a beginner you will meet both variants every day.

### 1. Numeric (octal) notation

Each access right corresponds to a fixed binary place value. If you memorize the numbers 4, 2 and 1, you can add up any mode in your head in no time:

- `r` (Read) = $2^2$ = **4**
- `w` (Write) = $2^1$ = **2**
- `x` (Execute) = $2^0$ = **1**
- `-` (no right) = **0**

Adding these values yields a three-digit octal number (or four digits with special bits):

| Octal | Binary | Symbolic | Meaning and typical use |
|---|---|---|---|
| **7** | `111` | `rwx` | Full access: read, write and execute / enter |
| **6** | `110` | `rw-` | Standard for regular files (read and write) |
| **5** | `101` | `r-x` | Standard for directories and programs (read and execute) |
| **4** | `100` | `r--` | Write-protected: read-only (e.g. sensitive configurations) |
| **3** | `011` | `-wx` | Write and enter only (rare special case) |
| **2** | `010` | `-w-` | Write only (e.g. pure write-only dropboxes) |
| **1** | `001` | `--x` | Execute / enter only (without listing filenames) |

| Octal | Binary | Symbolic | Meaning and typical use |
|---|---|---|---|
| `0` | `000` | `---` | No access at all for this user class |

**□□ Practical example:**

Typical chmod calls with octal numbers for different purposes:

```
# File readable and writable for the owner, read-only for group and others
chmod 644 /var/www/html/index.html

# Script full for the owner, only readable and executable for everyone else
chmod 755 /usr/local/bin/backup.sh

# Sensitive file (e.g. private SSH key) readable and writable only for the owner
chmod 600 ~/.ssh/id_ed25519

# Database directory strictly accessible only to the owner
chmod 700 /var/lib/postgresql/16/main
```

## 2. Symbolic notation

Symbolic notation modifies existing permissions in a targeted way without having to overwrite the remaining bits. The syntax follows this structure:

$$\text{[target class(es)]} \quad [\text{operator}] \quad [\text{rights}]$$

- **Target classes:**
- `u` (*User / Owner*): File owner
- `g` (*Group*): Group of the file
- `o` (*Others*): All other users
- `a` (*All*): All three classes (`u`, `g` and `o` together)
- **Operators:**
- `+` : Adds the given right

- `-` : Removes the given right
- `=` : Sets the rights exactly to this value (overwrites old values)
- **Right flags:** `r`, `w`, `x`, `s` (SUID/SGID), `t` (sticky bit), `X` (conditional execute)

**Practical example:**

Targeted adjustment of individual bits with symbolic notation:

```
# Add execute right for the owner on a file
chmod u+x deploy.sh

# Strip all rights from other users on a file
chmod o-rwx secret.conf

# Set group and others exactly to read and execute
chmod go=rx script.sh

# Grant read right to all users
chmod a+r public.txt

# Separate several changes with a comma:
chmod u=rw,g=r,o= config.json
```

## The conditional execute bit (`X`)

On recursive permission changes (`chmod -R`) you often face a dilemma: directories necessarily need the `x` bit (so they can be entered), but regular text or HTML files should never be executable.

With the **capital** `X` (*conditional execute*) `chmod` solves this cleanly: it sets the execute right **only on directories** or on files that are already executable for at least one class:

**Practical example:**

We harden a web directory recursively without accidentally making text files executable:

```
# Safe default for a web root:
# Directories become 755 (rwxr-xr-x), files stay 644 (rw-r--r--)
chmod -R u=rwX,go=rX /var/www/html/
```

### Copying rights from a reference (--reference)

`chmod` allows cloning permissions from a template file onto one or more target files:

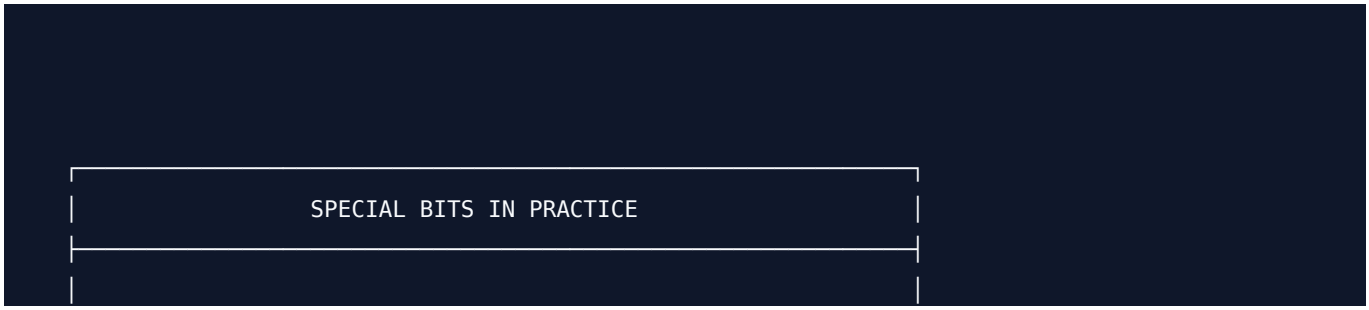
**□□ Practical example:**

Transfer rights from an existing configuration file 1:1 onto a new file:

```
# Copy rights from template.conf exactly onto new_service.conf
chmod --reference=/etc/app/template.conf /etc/app/new_service.conf
```

## The special bits: SUID, SGID and sticky bit

Besides the standard read, write and execute rights, POSIX provides three powerful special functions. They are controlled in four-digit octal notation by the first digit (4, 2, 1).



```
┌──────────────────────────────────────────────────────────┐
│  Bit 1: SUID (octal 4000 / u+s)                          │
│  ──> File runs with the OWNER's privileges               │
│      Example: /usr/bin/passwd (owner: root)              │
│                                                          │
│  Bit 2: SGID (octal 2000 / g+s)                          │
│  ──> Directory: new files inherit GROUP membership       │
│      Example: /srv/shared_team/ (group: devteam)         │
│                                                          │
│  Bit 3: Sticky bit (octal 1000 / +t)                     │
│  ──> Deletion protection: only the owner may delete      │
│      Example: /tmp and /var/tmp (drwxrwxrwt)             │
│                                                          │
└──────────────────────────────────────────────────────────┘
```

## 1. SUID (Set User ID – octal `4000` / `u+s`)

When an executable binary has the SUID bit, the process is not run with the rights of the calling user, but with the **rights of the file owner** (usually `root`).

- **Symbolic representation:** In `ls -l` output the owner's `x` is replaced by an `s` (e.g. `-rwsr-xr-x`).
- **Typical example:** The program `/usr/bin/passwd`. A normal user must be able to change their own password, which requires write access to the protected file `/etc/shadow`. Through the SUID bit, `passwd` runs temporarily with root rights.

**□□ Practical example:**

We demonstrate setting the SUID bit on a binary:

```
# Set the SUID bit on a binary
sudo chmod 4755 /usr/local/bin/custom-tool
# Alternatively symbolically:
sudo chmod u+s /usr/local/bin/custom-tool
```

> ⚠ **SUID security risk:** Every SUID-root binary is a potential target for privilege escalation. For that reason the modern Linux kernel completely ignores the SUID bit on shell and interpreter scripts ( `#!/bin/bash` , `#!/usr/bin/env python` ) for security reasons! On filesystems that users can write to (such as `/tmp` or `/home` ), the mount option `nosuid` should also be set in `/etc/fstab` .

## 2. SGID (Set Group ID – octal `2000` / `g+s` )

The SGID bit has two different application areas:

1. **On binaries:** The process runs with the permissions of the file group.
2. **On directories (important for teamwork):** Newly created files normally receive the creating user's primary group as their group. If the SGID bit is set on a directory, **all newly created files and subfolders automatically inherit the group membership of the parent directory**.

**□□ Practical example:**

We set up a directory for a developer team with automatic group inheritance:

```
# Set up a directory for shared teamwork
sudo mkdir /srv/project-files
sudo chown root:developers /srv/project-files

# Set the SGID bit (rights: drwxrws---)
sudo chmod 2770 /srv/project-files
```

## 3. Sticky bit (restricted deletion – octal `1000` / `+t` )

The sticky bit prevents unauthorized deletion or renaming of files in directories that several users can write to.

- **Symbolic representation:** The execute bit for others is shown as `t` (e.g. `drwxrwxrwt`).
- **Behavior:** In a folder with the sticky bit, a file may be deleted or renamed only by:

1. The owner of the file,
2. The owner of the directory,
3. The administrator (`root`).

**Practical example:**

Setting up a global temporary exchange directory with protection against foreign deletion:

```
# Set the sticky bit on a directory
sudo chmod 1777 /tmp/shared-scratch/
# Alternatively symbolically:
sudo chmod +t /tmp/shared-scratch/
```

## Capital `S` vs. lowercase `s` / capital `T` vs. lowercase `t`

If you see a capital `S` or capital `T` in `ls -l`, this means: **the special bit is active, but the underlying execute bit (`x`) is missing!**

- `-rwSr-xr-x`: SUID active, but the owner may not execute the file (misconfiguration).
- `drwxrwx--T`: Sticky bit active, but others may not enter the folder.
- `-rwsr-xr-x`: Correctly set SUID including execute (`s` lowercase).
- `drwxrwxrwt`: Correctly set sticky bit including execute (`t` lowercase).

## Modern alternative to SUID: Linux file capabilities

To grant programs individual kernel privileges without giving them full root rights via SUID, modern Linux uses **capabilities** (`setcap` / `getcap`):

**Practical example:**

Allow a custom service to open privileged ports without being root:

```
# Example: allow a web server to listen on privileged ports (<1024):
sudo setcap 'cap_net_bind_service=+ep' /usr/local/bin/custom-webserver

# Check active capabilities of a binary:
getcap /usr/local/bin/custom-webserver
```

## Default permissions and umask

When a process creates a new file or folder, the kernel does not immediately assign static rights, but applies a bitmask named **`umask`** (*user creation mask*).

### How the umask is calculated

The Linux kernel starts from maximum base rights:

- **New files:** Base `666` ( `rw-rw-rw-` — no execute for security reasons)
- **New directories:** Base `777` ( `rwxrwxrwx` — including execute so they can be entered)

The `umask` masks (subtracts) unwanted bits through a bitwise `AND NOT` operation:

$$\text{Effective rights} = \text{base rights} \land (\neg \text{umask})$$

```
┌──────────────────────────────────────────────────────────┐
│              UMASK CALCULATION (EXAMPLE 022)             │
├──────────────────────────────────────────────────────────┤
│                                                          │
│  Files:                                                  │
```

```
│    Base mode       : 6 6 6  (rw- rw- rw-)                │
│  - umask           : 0 2 2  (--- -w- -w-)                │
│  ──────────────────────────────────────────              │
│  = Effective file  : 6 4 4  (rw- r-- r--)                │
│                                                          │
│  Directories:                                            │
│    Base mode       : 7 7 7  (rwx rwx rwx)                │
│  - umask           : 0 2 2  (--- -w- -w-)                │
│  ──────────────────────────────────────────              │
│  = Effective dir   : 7 5 5  (rwx r-x r-x)                │
│                                                          │
└──────────────────────────────────────────────────────────┘
```

## Typical umask values compared:

| umask | Effective file rights | Effective directory rights | Purpose |
|---|---|---|---|
| **0022** | 644 ( rw-r--r-- ) | 755 ( rwxr-xr-x ) | Default for desktop and workstation systems |
| **0027** | 640 ( rw-r----- ) | 750 ( rwxr-x--- ) | Hardened server environment (others have no access at all) |
| **0077** | 600 ( rw------- ) | 700 ( rwx------ ) | High-security environments (only the owner has access) |

### □□ Practical example:

Query and adjust umask in the current shell:

```
# Show the current shell umask
umask

# Set umask temporarily to a restrictive value
```

```
umask 027
```

## Permanent umask configuration

To set the `umask` system-wide or per user:

1. **System-wide in `/etc/login.defs`:**

   ```
   UMASK 027
   ```

2. **Per user in `~/.bashrc` or `~/.profile`:**

   ```
   umask 027
   ```

3. **In systemd service units:**

   ```
   [Service]
   UMask=0027
   ```

# Extended attributes and alternatives: chattr and POSIX ACLs

Classic POSIX file rights reach their limits in two situations: when files should be protected even from the root user, or when more than one group needs different access rights.

## 1. Immutable files with `chattr`

The `chattr` tool (*change attribute*) manipulates extended filesystem flags (supported by ext4, XFS and Btrfs):

**□□ Practical example:**

We make a central system file immutable and test the protection:

```
# Make a file immutable (even root cannot delete or edit it)
sudo chattr +i /etc/resolv.conf

# Check attributes
lsattr /etc/resolv.conf

# Lift the lock again
sudo chattr -i /etc/resolv.conf

# Set a file to append-only (ideal for log files: only appending allowed)
sudo chattr +a /var/log/critical-audit.log
```

> **□□ Ransomware protection:** Setting the `+i` attribute on offline backups or configuration directories prevents automated malware or ransomware from overwriting or encrypting existing files on the system.

### Overview of the most important chattr attributes

| Flag | Attribute | Meaning and behavior |
|---|---|---|
| `+i` | Immutable | File cannot be modified, deleted, renamed or linked (even by root). |
| `+a` | Append-only | File can only be opened to append data (ideal for audit logs). |
| `+A` | No atime | Does not update the access timestamp on reads (saves I/O). |
| `+d` | No dump | File is ignored by the classic backup tool `dump`. |
| `+s` | Secure deletion | On deletion, data blocks on disk are overwritten with zeros. |
| `+c` | Compressed | Kernel compresses data blocks transparently on disk (filesystem-dependent). |

## 2. Fine-grained rights with POSIX ACLs (`setfacl` and `getfacl`)

When a third user (for example a backup service) needs read access to a file without becoming the owner or being added to the main group, **access control lists (ACLs)** come into play:

**□□ Practical example:**

Grant targeted extra permissions for individual system users:

```
# Grant user 'backupuser' read rights specifically
setfacl -m u:backupuser:r /etc/shadow

# Inspect current ACLs of a file
getfacl /etc/shadow

# Remove all extended ACLs again
setfacl -b /etc/shadow
```

### Inheritance with default ACLs on directories

With the `-d` switch (*default*) you define an ACL that is automatically inherited by all files and subfolders created in this directory in the future:

**□□ Practical example:**

We configure a default ACL for a group directory and back up the rules:

```
# Inherit a default ACL for the developer group on a directory
sudo setfacl -d -m g:devteam:rwx /srv/project-files/

# Back up all ACLs of a directory tree
getfacl -R /srv/project-files > /backup/permissions_acl.bak

# Apply saved ACLs to a restored directory
setfacl --restore=/backup/permissions_acl.bak
```

## Linux capabilities: granular rights without SUID root

Traditionally Unix divides processes into two classes: **unprivileged** (UID != 0, DAC checks apply fully) and **privileged** (UID == 0, root bypasses all DAC checks).

This all-or-nothing principle is risky: if a service such as a web server gets SUID root only so it can bind port 80, it has full control over the entire server in case of a vulnerability.

Linux solves this with **capabilities** (based on the POSIX.1e draft), which split root privileges into more than 40 discrete individual permissions.

### The most important Linux capabilities at a glance

| Capability | Function and permission | Typical use |
|---|---|---|

| | | |
|---|---|---|
| `CAP_NET_BIND_SERVICE` | Binding sockets to privileged ports (< 1024) | Web servers (Nginx, Caddy), DNS servers (named) |
| `CAP_NET_RAW` | Creating RAW and packet sockets | Network tools ( `ping` , `traceroute` , `tcpdump` ) |
| `CAP_SYS_TIME` | Changing the system clock and hardware RTC | Time synchronization ( `chronyd` , `systemd-timesyncd` ) |
| `CAP_DAC_OVERRIDE` | Bypassing all read, write and execute checks | Backup tools (Bacula, restic, BorgBackup) |
| `CAP_CHOWN` | Arbitrary changes of file owners and groups | File servers, container runtimes |
| `CAP_SYS_ADMIN` | Broad administration rights (mount, quota, BPF) | Container engines (Docker, Podman, LXC) |
| `CAP_SETUID` | Arbitrary switching of process UIDs | Login daemons (sshd, display managers) |

## Managing file capabilities (setcap and getcap)

Capabilities are stored in the extended attributes of the filesystem ( `security.capability` ):

**□□ Practical example:**

We check existing file capabilities and equip a program with network rights:

```
# 1. Check status: does ping have capabilities or SUID?
ls -l /bin/ping
getcap /bin/ping

# 2. Give ping RAW network capabilities (no SUID root needed!):
sudo setcap 'cap_net_raw=+ep' /bin/ping

# 3. Allow a web server to bind port 443 without root rights:
sudo setcap 'cap_net_bind_service=+ep' /usr/local/bin/my-web-app
```

```
# 4. Remove all capabilities from a file completely:
sudo setcap -r /usr/local/bin/my-web-app
```

**Explanation of the capability flags:**

- `e` (*Effective*): The capability is immediately active for the thread.
- `p` (*Permitted*): The process may use or activate this capability.
- `i` (*Inheritable*): The capability is inherited across system calls (`execve`) to child processes.

## Automation: permissions in Ansible and shell scripts

In modern DevOps and server infrastructures, file rights are no longer typed by hand, but managed declaratively through configuration management.

### 1. Defining permissions in Ansible playbooks

Basic concepts for playbooks and modules are covered in the guide Ansible fundamentals for Linux administrators.

The Ansible module `ansible.builtin.file` offers native support for symbolic and octal permissions as well as filesystem attributes:

**□□ Practical example:**

Declarative hardening of web roots and secrets via an Ansible playbook:

```
---
- name: Harden web server permissions and configuration
  hosts: webservers
  become: true
  tasks:
```

```
    - name: Create web-root directory with correct rights
      ansible.builtin.file:
        path: /var/www/production
        state: directory
        owner: webadmin
        group: www-data
        mode: '0755'

    - name: Secure and lock a sensitive configuration file
      ansible.builtin.file:
        path: /etc/app/production.secret.env
        state: file
        owner: root
        group: root
        mode: '0600'
        attributes: '+i'

    - name: Recursive rights with conditional execute bit
      ansible.builtin.file:
        path: /srv/shared_assets
        state: directory
        owner: deployer
        group: devteam
        mode: 'u=rwX,g=rX,o='
        recurse: true
```

## 2. Reusable Bash script for permission harmonization

Deeper shell techniques and automation patterns are covered in our module shell scripting and automation.

For local maintenance or deployment pipelines, a dedicated shell script that cleans directories and files atomically helps:

**□□ Practical example:**

A field-tested maintenance script for cleaning up mis-set permissions:

```
#!/usr/bin/env bash
# /usr/local/bin/fix-permissions.sh
set -euo pipefail

TARGET_DIR="${1:-/var/www/html}"

if [ ! -d "$TARGET_DIR" ]; then
    echo "Error: directory $TARGET_DIR does not exist!" >&2
    exit 1
fi

echo "Harmonizing permissions for: $TARGET_DIR"

# 1. Set all directories to 755 (drwxr-xr-x)
find "$TARGET_DIR" -type d -exec chmod 755 {} +

# 2. Set all regular files to 644 (-rw-r--r--)
find "$TARGET_DIR" -type f -exec chmod 644 {} +

# 3. Make shell scripts executable again in a targeted way
find "$TARGET_DIR" -type f -name "*.sh" -exec chmod 750 {} +

# 4. Isolate sensitive .env and .key files
find "$TARGET_DIR" -type f \( -name "*.env" -o -name "*.key" -o -name "*.pem" \) -exec
chmod 600 {} +

echo "Permissions updated successfully."
```

## 3. Git and file permissions

An important aspect in software projects: **Git does not store full Linux permissions!**

Git stores only a single permission flag in the index:

- `100644`: Regular file (not executable)
- `100755`: Executable file (scripts, binaries)

**□□ Practical example:**

Declare script files as executable directly in the version-control system:

```
# Mark a file in the Git index as executable (without chmod on the host):
git update-index --chmod=+x deploy.sh

# Check which mode Git stores for a file:
git ls-files --stage deploy.sh
```

## Containers, Docker and Kubernetes: UID/GID mapping

A classic problem when running Docker containers (see also installing and using Docker on Ubuntu) is permission conflicts on mounted volumes ( `-v /host/data:/app/data` ).

```
┌──────────────────────────────────────────────────────────┐
│            DOCKER VOLUME PERMISSION MAPPING              │
├──────────────────────────────────────────────────────────┤
│                                                          │
│  Host filesystem:                                        │
│  /srv/docker_data (owner: root:root / mode: 755)         │
│         |                                                │
│         ▼  Volume mount (-v)                             │
│  Container process:                                      │
│  Runs as node or www-data (UID 1000 or 1001)             │
│         |                                                │
│         ▼  Result on write attempt:                      │
│  EACCES: Permission denied!                              │
│                                                          │
│  Clean solutions:                                        │
│  1. Host chown: sudo chown -R 1000:1000 /srv/docker_data  │
│  2. POSIX ACL: setfacl -R -m u:1000:rwx /srv/docker_data  │
│  3. Rootless Docker or user-namespace remapping (userns)  │
```

```
│                                                    │
└────────────────────────────────────────────────────┘
```

> ⚠ **Anti-pattern:** Never set `chmod -R 777` on host volumes to fix Docker permission errors! The system then loses all isolation and every unprivileged user on the host can manipulate container files.

## Kubernetes securityContext and fsGroup

Details on pod security standards and cluster architectures are in our Kubernetes documentation.

In Kubernetes, rights management in the pod manifest is controlled through `securityContext`:

**□□ Practical example:**

A Kubernetes pod manifest with strict user and group isolation:

```
apiVersion: v1
kind: Pod
metadata:
  name: hardened-app
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 10001
    runAsGroup: 10001
    fsGroup: 10001
  containers:
    - name: web
      image: nginx:alpine
      volumeMounts:
        - mountPath: /data
```

```
            name: storage
    volumes:
      - name: storage
        emptyDir: {}
```

Through `fsGroup: 10001`, Kubernetes automatically changes the group ownership of all contained files to GID 10001 when mounting the volume and sets the group write right.

## 10 practice scenarios and server hardening (playbooks)

### 1. LAMP and LEMP stack (WordPress and Nginx)

A complete guide to server setup is in our LEMP stack practice guide on Ubuntu.

The golden rule is: **the web-server user (`www-data`) may read and execute files, but must never write in the web root by default!**

**□□ Practical example:**

We set the rights of a typical web root according to least privilege:

```
# 1. Set owner to the system user, group to www-data
sudo chown -R webadmin:www-data /var/www/html/

# 2. Set directories to 755 and files to 644
sudo find /var/www/html/ -type d -exec chmod 755 {} +
sudo find /var/www/html/ -type f -exec chmod 644 {} +

# 3. Make only the upload directory writable for the web server
sudo chown -R www-data:www-data /var/www/html/wp-content/uploads/
sudo chmod -R 775 /var/www/html/wp-content/uploads/

# 4. Forbid execution of scripts in the upload directory (Nginx configuration)
```

```
# location ~* /uploads/.*\.php$ { deny all; }
```

## 2. Hardening SSH permissions

For further protective measures including 2FA and intrusion prevention we refer to our guide on Linux server hardening with FIDO2 and CrowdSec.

The OpenSSH daemon refuses authentication if key files or the `.ssh` directory have permissions that are too open (*“UNPROTECTED PRIVATE KEY FILE”*):

**□□ Practical example:**

Secure SSH keys and configuration files to spec:

```
# Secure the SSH configuration directory
chmod 700 ~/.ssh

# Private key files (strictly owner only)
chmod 600 ~/.ssh/id_ed25519 ~/.ssh/id_rsa

# Public keys and authorized_keys
chmod 644 ~/.ssh/id_ed25519.pub
chmod 600 ~/.ssh/authorized_keys

# System-wide SSH host keys (/etc/ssh/)
sudo chmod 600 /etc/ssh/ssh_host_*_key
sudo chmod 644 /etc/ssh/ssh_host_*_key.pub
```

## 3. Hardening database servers

Database engines such as PostgreSQL or MariaDB refuse to run if their data stores are visible to other users:

**□□ Practical example:**

Protect database directories against unauthorized access:

```
# PostgreSQL data directory
sudo chown -R postgres:postgres /var/lib/postgresql/
sudo chmod 700 /var/lib/postgresql/data

# MySQL / MariaDB data directory
sudo chown -R mysql:mysql /var/lib/mysql/
sudo chmod 750 /var/lib/mysql/
```

## 4. Hardening Samba and NFS file shares

On network file servers (Samba/NFS), Linux file rights must be synchronized with the rights of the network protocols:

**□□ Practical example:**

Configuration example in `/etc/samba/smb.conf`:

```
[TeamShare]
path = /srv/samba/team
read only = no
create mask = 0660
directory mask = 0770
force group = devteam
inherit permissions = yes
```

## 5. Git bare repositories for team servers

When several developers push via SSH to a central Git repository on a server:

**□□ Practical example:**

Create a central bare repository with shared group permissions:

```
# Create a central bare repository with shared-group permissions
sudo mkdir -p /srv/git/project.git
cd /srv/git/project.git
sudo git init --bare --shared=group
sudo chown -R root:developers /srv/git/project.git
sudo chmod -R 2775 /srv/git/project.git
```

## 6. Hardening cron jobs and system scripts

Scripts under `/etc/cron.*` or `/usr/local/sbin` must be protected against manipulation by unprivileged users:

**□□ Practical example:**

Protect system-maintenance scripts against unauthorized modification:

```
# Cron scripts strictly writable and executable only for root
sudo chown root:root /usr/local/sbin/daily-backup.sh
sudo chmod 700 /usr/local/sbin/daily-backup.sh
```

## 7. Temporary execution directories (/tmp and /var/tmp)

**□□ Practical example:**

Ensure that the sticky bit is active on temporary system folders:

```
# Ensure the sticky bit is active on /tmp
sudo chmod 1777 /tmp /var/tmp
```

## 8. Hardening certificates and TLS private keys

**□□ Practical example:**

Strictly secure private key files for TLS certificates:

```
# Protect the TLS key directory
sudo chown -R root:ssl-cert /etc/ssl/private
sudo chmod 710 /etc/ssl/private
sudo chmod 640 /etc/ssl/private/*.key
```

## 9. Log directories for web and app services

**□□ Practical example:**

Configure log directories with group read rights for administrators and audit tools:

```
# Log directory for Nginx
sudo chown -R www-data:adm /var/log/nginx
sudo chmod 750 /var/log/nginx
sudo chmod 640 /var/log/nginx/*.log
```

### 10. Multi-user drop-in boxes (write without reading others)

**Practical example:**

A mailbox folder into which every user can drop files, but cannot inspect other people's files:

```
# A mailbox folder into which everyone can drop files, but cannot inspect others' files:
sudo mkdir /srv/dropin
sudo chmod 1733 /srv/dropin  # rwx-wx-wx with sticky bit
```

## 5 step-by-step lab exercises (hands-on)

To turn the theoretical concepts into practical administration skills, we walk through five typical practice scenarios.

### Lab 1: Collaborative developer directory with SGID and default ACLs

**Scenario:** Developers `alice` and `bob` belong to the group `devteam`. They need a shared project folder `/srv/devproject` in which every newly created file is immediately writable for both — without manual permission adjustment.

**Practical example:**

Step-by-step setup of the team folder with SGID and default ACLs:

```
# 1. Create group and users (if they do not exist)
sudo groupadd devteam
sudo usermod -aG devteam alice
sudo usermod -aG devteam bob
```

```
# 2. Create the directory and set permissions
sudo mkdir -p /srv/devproject
sudo chown root:devteam /srv/devproject
sudo chmod 2770 /srv/devproject

# 3. Set up default ACLs for maximum collaboration
sudo setfacl -d -m g:devteam:rwx /srv/devproject
sudo setfacl -m g:devteam:rwx /srv/devproject

# 4. Test as user Alice:
sudo -u alice touch /srv/devproject/api.py
ls -l /srv/devproject/api.py
```

**Result:** The file `api.py` automatically belongs to the group `devteam` and is immediately editable for Bob.

## Lab 2: Forensic analysis and removal of an SUID backdoor

**Scenario:** After a security incident the system should be inspected for unauthorized SUID binaries in temporary folders.

**□□ Practical example:**

Find and defuse suspicious SUID files:

```
# 1. Start a system-wide scan for SUID files
sudo find / -perm -4000 -type f 2>/dev/null > /tmp/suid_scan.txt

# 2. Check suspicious paths in /tmp, /dev/shm or /var/tmp
grep -E '^/tmp|^/dev/shm|^/var/tmp' /tmp/suid_scan.txt || echo "No SUID files in temp
directories!"

# 3. Defuse a found backdoor file:
# sudo chmod u-s /tmp/.hidden_root_shell
# sudo rm -f /tmp/.hidden_root_shell
```

## Lab 3: WordPress hardening with an immutable configuration

**Scenario:** A WordPress web root should be hardened so that attackers, even with an RCE vulnerability in a theme, can neither manipulate nor overwrite the central configuration file `wp-config.php`.

**□□ Practical example:**

Hardening and protection through the immutable attribute:

```
# 1. Set base rights (owner webadmin, group www-data)
sudo chown -R webadmin:www-data /var/www/wordpress
sudo find /var/www/wordpress -type d -exec chmod 755 {} +
sudo find /var/www/wordpress -type f -exec chmod 644 {} +

# 2. Set wp-config.php to 640 and lock it for webadmin/www-data
sudo chmod 640 /var/www/wordpress/wp-config.php
sudo chown webadmin:www-data /var/www/wordpress/wp-config.php

# 3. Enable the immutable bit (even root cannot delete the file now):
sudo chattr +i /var/www/wordpress/wp-config.php

# 4. Verify the protection
lsattr /var/www/wordpress/wp-config.php
```

## Lab 4: Secure SFTP-only chroot jail

**Scenario:** An external contractor (`sftpuser`) should be allowed to upload files to the server via SFTP, but must neither enter the system via SSH nor inspect other directories.

**□□ Practical example:**

Set up an isolated chroot directory for SFTP access:

```
# 1. Create SFTP group and user
sudo groupadd sftpusers
sudo useradd -g sftpusers -d /srv/sftp/sftpuser -s /usr/sbin/nologin sftpuser

# 2. Prepare the chroot directory (MUST belong to root:root 755!)
sudo mkdir -p /srv/sftp/sftpuser/uploads
sudo chown root:root /srv/sftp/sftpuser
sudo chmod 755 /srv/sftp/sftpuser

# 3. Make the upload folder writable for the SFTP user
sudo chown sftpuser:sftpusers /srv/sftp/sftpuser/uploads
sudo chmod 750 /srv/sftp/sftpuser/uploads

# 4. Configure OpenSSH (/etc/ssh/sshd_config):
# Match Group sftpusers
#     ChrootDirectory /srv/sftp/%u
#     ForceCommand internal-sftp
#     PasswordAuthentication yes
#     X11Forwarding no
#     AllowTcpForwarding no
```

## Lab 5: SELinux contexts vs. DAC permissions (RHEL/Fedora)

**Scenario:** Despite `chmod 777`, Nginx on a Fedora/RHEL system reports `403 Forbidden` when accessing an HTML document.

### □□ Practical example:

Analyze and repair SELinux contexts:

```
# 1. Check DAC rights:
ls -l /var/www/html/index.html
```

```
# 2. Show SELinux security contexts (-Z flag):
ls -Z /var/www/html/index.html

# If the type is not httpd_sys_content_t:
sudo restorecon -Rv /var/www/html/

# Alternatively assign the correct context manually:
sudo chcon -t httpd_sys_content_t /var/www/html/index.html
```

## systemd services and sandbox hardening

Modern Linux systems use `systemd` to isolate background services securely. Instead of relying solely on conventional filesystem permissions, service units offer built-in filesystem filters and dynamic user management:

**□□ Practical example:**

A hardened systemd service unit with strict isolation and a dynamic user:

```
[Unit]
Description=Hardened Web Application
After=network.target

[Service]
Type=simple
DynamicUser=yes
ExecStart=/usr/local/bin/app-server

# systemd filesystem hardening:
ProtectSystem=strict
ProtectHome=true
ReadWritePaths=/var/log/app /var/lib/app/data
ReadOnlyPaths=/etc/app
```

```
PrivateTmp=true
ProtectKernelTunables=true
ProtectControlGroups=true
CapabilityBoundingSet=
NoNewPrivileges=true
UMask=0027

[Install]
WantedBy=multi-user.target
```

**Explanation of the security directives:**

- `DynamicUser=yes`: Automatically creates a temporary, unprivileged UID/GID at runtime and discards it when the service stops — no manual system users needed!
- `ProtectSystem=strict`: Mounts the entire filesystem (`/usr`, `/boot`, `/etc`) as **read-only** for the service process.
- `ReadWritePaths=`: Unlocks only the directories that are strictly required for write access.
- `PrivateTmp=true`: Creates an isolated `/tmp` directory for the service, so other system users cannot inspect temporary files.
- `UMask=0027`: Forces all files created by the service to be protected from unauthorized users automatically.

# Troubleshooting and security auditing

## The 10 most common permission errors in practice

1. **“Permission denied” despite `chmod 777` on the file:**

- **Cause:** A parent directory in the path has no execute bit (`x`).
- **Fix:** Use `namei -l /path/to/file` to check the entire path tree for missing `x` bits.

1. **A script with `755` cannot be executed:**

- **Cause:** The filesystem is mounted with the `noexec` option in `/etc/fstab` (e.g. on `/tmp` or external USB sticks).

- **Fix:** Check `mount | grep "noexec"`.

1. **An SUID script is ignored:**

- **Cause:** The Linux kernel ignores SUID on shell and Python scripts.
- **Fix:** For script escalation use `sudoers` with `NOPASSWD` or a C wrapper binary instead.

1. **A file cannot be deleted even by `root` (*“Operation not permitted”*):**

- **Cause:** The filesystem attribute `+i` (immutable) is active.
- **Fix:** Check with `lsattr filename` and remove with `sudo chattr -i filename`.

1. **POSIX ACL mask blocks effective rights:**

- **Cause:** The ACL mask restricts group rights.
- **Fix:** Check `getfacl file` and open the mask with `setfacl -m m:rwx file`.

1. **SSH key is rejected (*“Permissions 0644 are too open”*):**

- **Cause:** Private SSH keys may only be readable by the owner.
- **Fix:** Run `chmod 600 ~/.ssh/id_ed25519`.

1. **The web server cannot overwrite uploaded files:**

- **Cause:** The upload folder belongs to `root` or has the wrong group permissions.
- **Fix:** `sudo chown -R www-data:www-data uploads/ && sudo chmod 775 uploads/`.

1. **A Docker container throws `EACCES` on a mounted host folder:**

- **Cause:** The process in the container runs under UID 1000, the host folder belongs to root.
- **Fix:** `sudo chown -R 1000:1000 /host/path` or set POSIX ACLs.

1. **A deleted file still occupies disk space:**

- **Cause:** A process still holds the file open in RAM (`open file descriptor`).
- **Fix:** Find it with `lsof | grep deleted` and restart the corresponding service.

1. **A new user cannot create files in a shared group folder:**

Source: https://admindocs.de/en/linux-beginners/chmod-on-linux-basics-to-best-practices

- **Cause:** The user has not logged in again after being added to the group (the group token in the kernel is stale).
- **Fix:** Log in again or run `newgrp groupname` in the shell.

## Complete security audit script for cron jobs

This shell script searches the entire filesystem for security risks and generates a clear audit report:

**□□ Practical example:**

An automated inspection script for regular security checks:

```
#!/usr/bin/env bash
# /usr/local/sbin/audit-permissions.sh
set -euo pipefail

REPORT_FILE="/var/log/permission-audit-$(date +%F).log"
echo "=== SECURITY PERMISSION AUDIT: $(date) ===" > "$REPORT_FILE"

echo -e "\n[1] SUID binaries (potential root escalation):" >> "$REPORT_FILE"
find / -perm -4000 -type f -exec ls -ld {} + 2>/dev/null >> "$REPORT_FILE"

echo -e "\n[2] SGID binaries and directories:" >> "$REPORT_FILE"
find / -perm -2000 -type f -exec ls -ld {} + 2>/dev/null >> "$REPORT_FILE"

echo -e "\n[3] World-writable files (chmod 777 / o+w):" >> "$REPORT_FILE"
find / -type f -perm -0002 ! -path "/proc/*" ! -path "/sys/*" -exec ls -ld {} +
2>/dev/null >> "$REPORT_FILE"

echo -e "\n[4] World-writable directories WITHOUT sticky bit:" >> "$REPORT_FILE"
find / -type d -perm -0002 ! -perm -1000 ! -path "/proc/*" ! -path "/sys/*" -exec ls -ld
{} + 2>/dev/null >> "$REPORT_FILE"

echo -e "\n[5] Files without valid user or group (orphaned inodes):" >> "$REPORT_FILE"
find / \( -nouser -o -nogroup \) ! -path "/proc/*" ! -path "/sys/*" -exec ls -ld {} +
2>/dev/null >> "$REPORT_FILE"
```

```
echo -e "\nAudit complete. Report is at: $REPORT_FILE"
```

### Real-time monitoring of chmod with the Linux Audit Framework (auditd)

To log suspicious permission changes on the system without gaps:

**□□ Practical example:**

Define and analyze an audit rule for chmod calls:

```
# 1. Define an audit rule for the 'chmod' system call
sudo auditctl -a always,exit -F arch=b64 -S chmod -S fchmod -S fchmodat -k chmod_monitor

# 2. Search audit logs for changes
sudo ausearch -k chmod_monitor --format text
```

## Filesystem differences: ext4, XFS, Btrfs and ZFS

Filesystems differ fundamentally in how they manage POSIX permissions, extended attributes and inodes:

| Filesystem | Inode structure | POSIX ACL support | Immutable (+i) support | Permission peculiarities |
|---|---|---|---|---|
| **ext4** | Static inode table (256 bytes) | Native via `acl` mount option | Fully supported | Default on Debian/Ubuntu, very robust |
| **XFS** | Dynamic inodes | Native, on by default | Fully supported | Default on RHEL/Fedora, excellent for large files |

| Filesystem | Inode structure | POSIX ACL support | Immutable (+i) support | Permission peculiarities |
|---|---|---|---|---|
| **Btrfs** | Subvolume inodes | Native support | Fully supported | Permissions apply inside subvolumes |
| **ZFS** | Dynamic object IDs | NFSv4 and POSIX ACLs | Partial (ZFS properties) | Supports granular NFSv4 ACLs |

## Filesystem mount options for maximum security

In `/etc/fstab`, filesystems can be hardened with security-relevant mount flags:

### □□ Practical example:

Record secure mount options in `/etc/fstab`:

```
# /etc/fstab example for hardened partitions:
# 1. Temporary directory without execute rights and SUID bits:
tmpfs   /tmp         tmpfs   defaults,noexec,nosuid,nodev   0  0

# 2. Secure shared memory:
tmpfs   /dev/shm     tmpfs   defaults,noexec,nosuid,nodev   0  0

# 3. User home partition without SUID escalation:
/dev/sdb1  /home     ext4    defaults,nosuid,nodev          0  2
```

- `noexec`: Prevents direct execution of binaries on this partition (effective protection against downloaded malware in `/tmp`).
- `nosuid`: Ignores SUID and SGID bits on all files of this partition (prevents unauthorized root escalation).
- `nodev`: Prevents interpretation of block and character devices.

## Command Reference (Cheatsheet)

For quick orientation in daily terminal work, this table summarizes all combinations, modes and use cases:

| Mode | Octal | Symbolic | Meaning and recommended use |
|---|---|---|---|
| **600** | `0600` | `-rw-------` | Private SSH keys ( `id_ed25519` ), certificate keys ( `.key` ), `.env` files |
| **640** | `0640` | `-rw-r-----` | System configuration files with group read (e.g. `/etc/shadow` ) |
| **644** | `0644` | `-rw-r--r--` | Default for public documents, HTML, CSS, JS and image files |
| **660** | `0660` | `-rw-rw----` | Shared working files for team groups or service sockets |
| **700** | `0700` | `drwx------` | Personal home directories, `~/.ssh/` , PostgreSQL data directory |
| **750** | `0750` | `drwxr-x---` | System directories with group read (e.g. web-server log directories) |
| **755** | `0755` | `drwxr-xr-x` | Default for directories, binaries ( `/usr/bin/` ) and shell scripts |
| **770** | `0770` | `drwxrwx---` | Team directories with full write and read for group members |
| **1777** | `1777` | `drwxrwxrwt` | Global temporary directories with sticky bit ( `/tmp` , `/var/tmp` ) |
| **2770** | `2770` | `drwxrws---` | Shared directories with SGID inheritance for group work |
| **4755** | `4755` | `-rwsr-xr-x` | SUID binaries with root escalation ( `/usr/bin/passwd` , `/usr/bin/sudo` ) |

## Further Resources

| Resource | Description | Type |
|---|---|---|
| POSIX.1-2024 specification for chmod | The binding Open Group and IEEE specification for the chmod utility (2024 edition) | Official specification |
| GNU Coreutils: File permissions | Detailed GNU manual on permission modes, octal numbers and umask | Official documentation |

| Resource | Description | Type |
|---|---|---|
| Arch Linux Wiki: File permissions and attributes | Practical reference for Linux file rights, SUID bits and extended ACLs | Community wiki |
| Debian Wiki: Permissions | Fundamentals and field-tested best practices for Unix and Linux rights management | Manual and reference |
| Linux Audit Framework (auditd) | Comprehensive manual on monitoring and logging system calls | Technical documentation |
| Docker Documentation: Manage data in Docker | Official guide to volume management and user isolation in containers | Official documentation |

## Conclusion

The `chmod` command and the POSIX permission model behind it are a timeless piece of Unix architecture: with only 12 bits in the inode you can realize a robust, resource-efficient and extremely fast security foundation for multi-user, server, container and cloud infrastructures.

Anyone who has internalized the functional differences of `r`, `w` and `x` between files and directories, works with the conditional execute bit `X`, uses special bits such as SGID and the sticky bit in a targeted way, and automates permissions through Ansible or shell scripts, masters every permission challenge — without ever falling back on unsafe `chmod 777` emergency fixes.

> **Practical tip:** In daily administration always use `chmod -R u=rwX,go=rX /target/path` instead of coarse octal numbers. That guarantees all subdirectories remain cleanly enterable ( `x` ), while regular text and image files are not accidentally marked as executable binaries.