ADMIN:DOCS

Basics bis Best Practices: Alles über chmod in Linux

Verstehe Linux-Dateiberechtigungen von Grund auf: Inode-Bitmasken, numerische und symbolische Notation, SUID/SGID, Sticky Bit, umask, POSIX-ACLs, Capabilities, Ansible und Container-Hardening.

Kategorie: Linux-Neulinge **Veröffentlicht:** 19.09.2024 **Zuletzt geprüft:** 08.09.2026

Jeder, der die ersten Schritte auf der Linux-Kommandozeile wagt, erlebt früher oder später denselben frustrierenden Moment: Du schreibst dein erstes Shell-Skript, tippst erwartungsvoll `./deploy.sh` ein – und die Shell quittiert den Versuch trocken mit `bash: ./deploy.sh: Permission denied`. Oder du möchtest eine Konfigurationsdatei öffnen und scheiterst an der Meldung `Keine Berechtigung`.

Im ersten Augenblick wirkt diese Hürde wie reine Schikane. Tatsächlich ist sie das wichtigste Sicherheitsfundament des gesamten Systems. Linux wurde von Beginn an als echtes **Mehrbenutzersystem** (*Multi-User-System*) konzipiert. Während historische Einzelplatzsysteme wie MS-DOS jedem Programm schrankenlosen Zugriff auf den gesamten Datenträger gestatteten, herrscht unter Linux strikte Gewaltenteilung. Keine Datei gehört „einfach so dem Computer“ – jedes Dokument, jedes Verzeichnis und jeder Dienstprozess unterliegt festen Besitzverhältnissen aus Eigentümer und Gruppe.

Hier kommt der Befehl `chmod` (*Change Mode*) ins Spiel. Zusammen mit den grundlegenden Dateisystem- und Navigationsbefehlen steuerst du damit präzise, wer Dateien lesen, bearbeiten oder als Programm ausführen darf – und schützt dein System vor Fehlern ebenso wie vor ungewollten Zugriffen.

Dieser Leitfaden führt dich schrittweise durch das gesamte Berechtigungsmodell:

Quelle: https://admindocs.de/de/linux-neulinge/basics-bis-best-practices-alles-ueber-chmod-in-linux

Von der praktischen Entzifferung der Zeile `-rwxr-xr-x` in `ls -l` über den entscheidenden Unterschied zwischen Datei- und Verzeichnisrechten bis zur sicheren Anwendung der Oktal- und Symbolik-Notation. Anschließend blicken wir unter die Haube: Wir untersuchen Inode-Bitmasken, Kernel-Prüfungen und Spezial-Bits (SUID, SGID, Sticky Bit) und wenden dieses Fundament direkt in realen Szenarien für Webserver, SSH-Härtung, systemd-Sandboxing und Container an.

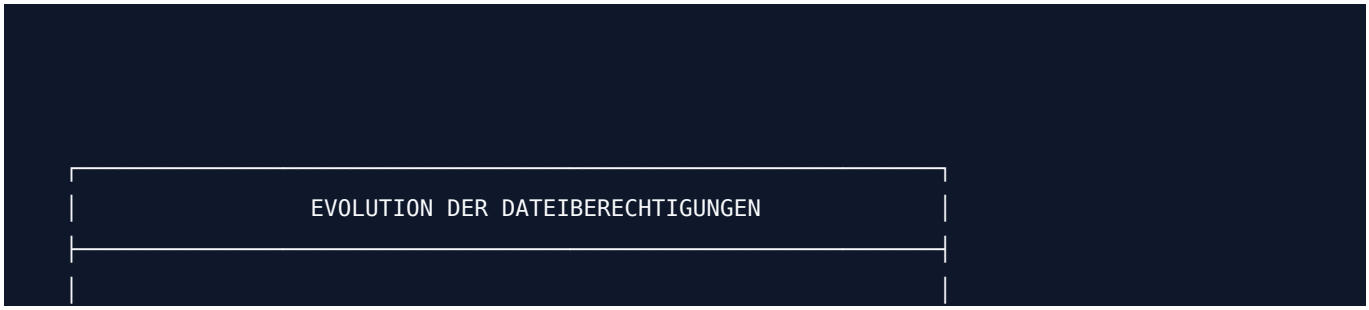
Historischer Ursprung & Evolution der Unix-Berechtigungsmodelle

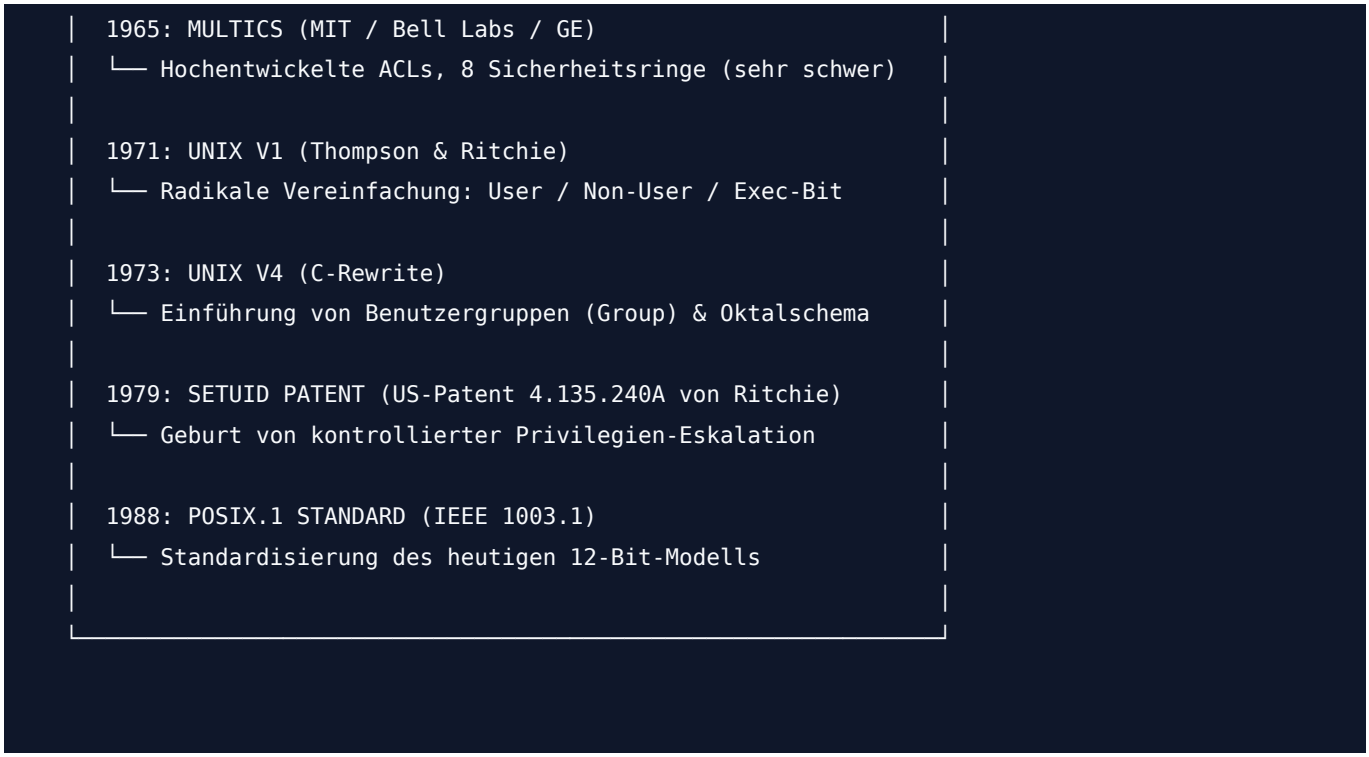
Ursprung: Unix V1 an den Bell Labs (1971)

Der Befehl `chmod` gehört zu den ältesten und beständigsten Werkzeugen der Informatikgeschichte. Am **3. November 1971** veröffentlichten **Ken Thompson** und **Dennis Ritchie** an den Bell Laboratories die erste Ausgabe von Unix (*Unix V1*) auf einer DEC PDP-11. Bereits in dieser Erstversion war `chmod` als Kernbestandteil des Betriebssystems enthalten.

Das ursprüngliche Berechtigungsmodell war jedoch noch deutlich simplizistischer aufgebaut als der heutige POSIX-Standard:

- Es existierten noch keine Benutzergruppen im heutigen Sinne.
- Berechtigungen wurden im Dateisystem lediglich in *User-Bits*, *Non-User-Bits* und ein globales *Execution-Bit* unterteilt.
- Das gesamte System war für eine Handvoll Forscher ausgelegt, die gemeinsam an Textverarbeitungs- und Programmierwerkzeugen arbeiteten.





Vom Multics-Erbe zur Unix-Philosophie

Unix entstand als pragmatischer Gegenentwurf zum hochkomplexen **Multics**-Betriebssystem (*Multiplexed Information and Computing Service*), an dem Bell Labs zuvor gemeinsam mit dem MIT und General Electric beteiligt war. Während Multics bereits Ende der 1960er Jahre umfassende hierarchische Access Control Lists (ACLs), dynamische Regelwerke und ein achtstufiges Ring-Sicherheitsmodell nutzte, entschieden sich Thompson und Ritchie bewusst für eine radikale Vereinfachung:

- **Multics-Ansatz:** Maximale Granularität und dynamische Regelwerke, die jedoch enormen Speicher- und Rechenaufwand bedeuteten und für Administratoren schwer überschaubar waren.
- **Unix-Ansatz:** Die Aufteilung in **Owner (Besitzer)**, **Group (Gruppe)** und **Others (Sonstige)** (siehe dazu auch unseren Leitfaden zur fortgeschrittenen Benutzerverwaltung){.badge-link-text} – ein Modell, das mit minimalem Speicheraufwand im Dateisystem-Inode auskommt und 99 % aller Anwendungsfälle im Alltag zuverlässig abdeckt.

Evolution zum POSIX-Standard & Dennis Ritchies SetUID-Patent

Mit der Veröffentlichung von Unix Version 4 (1973), in der der Kernel in C neu geschrieben wurde, und der späteren Standardisierung durch IEEE POSIX (POSIX.1 / IEEE 1003.1) etablierte sich das dreistufige Oktal-Schema mit Lese-, Schreib- und Ausführungs-Bits.

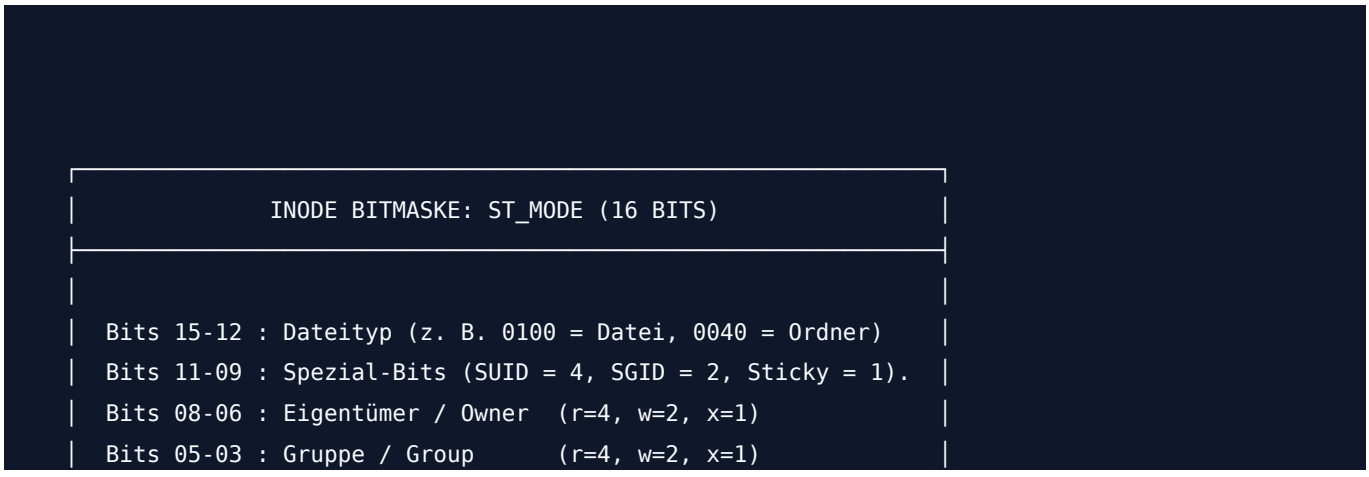
Ein Meilenstein war die Erfindung des **SetUID-Mechanismus** durch Dennis Ritchie. Um unprivilegierten Benutzern zu ermöglichen, ihr eigenes Passwort zu ändern (was Schreibzugriff auf die geschützte Systemdatei `/etc/passwd` bzw. später `/etc/shadow` erforderte), meldete Ritchie 1973 das SetUID-Konzept zum Patent an (erteilt 1979 als US-Patent 4.135.240A). Dieses Patent wurde von Bell Labs uneigennützig der gesamten Computerindustrie zur Verfügung gestellt und bildet bis heute die Basis für Werkzeuge wie `passwd`, `sudo` und `su`.

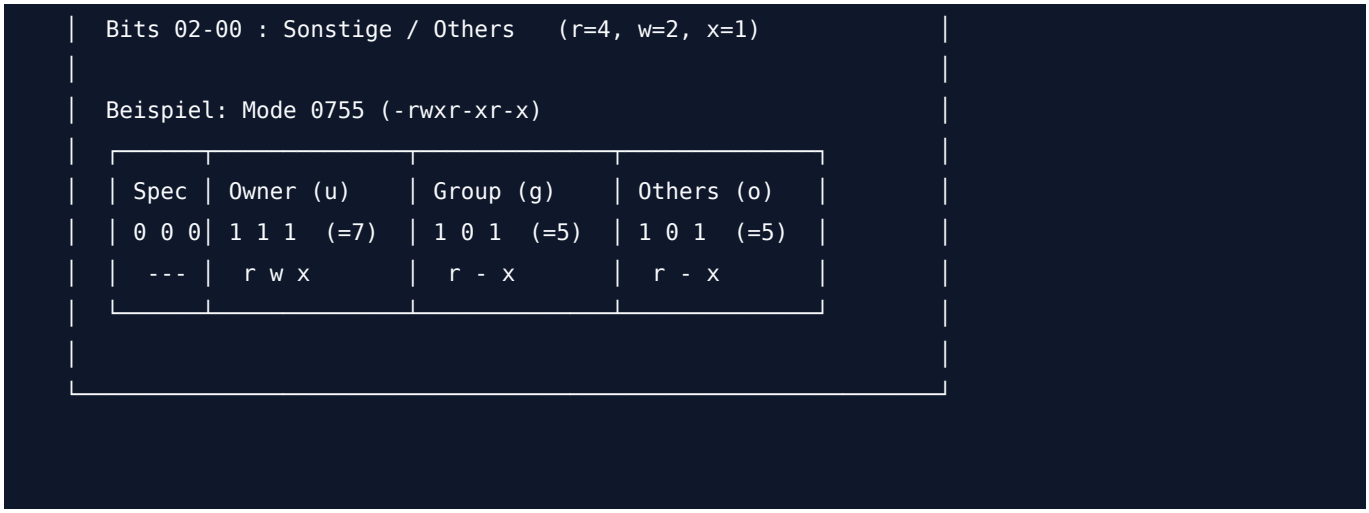
Das POSIX-Berechtigungsmodell unter der Haube

Wie Linux Berechtigungen im Dateisystem speichert

Dateiberechtigungen werden in Linux nicht im Verzeichniseintrag selbst gespeichert, sondern in den Metadaten des Dateisystems – dem sogenannten **Inode** (*Index Node*). Ein Verzeichniseintrag ist lediglich eine Zuordnungstabelle aus Dateiname und Inode-Nummer.

Im Inode reserviert der Linux-Kernel ein 16-Bit großes Datenfeld namens `st_mode` (definiert im C-Header `<sys/stat.h>`). Dieses Feld kodiert sowohl den Dateityp (z. B. reguläre Datei, Verzeichnis, Symlink, Block Device) als auch sämtliche Zugriffsrechte:





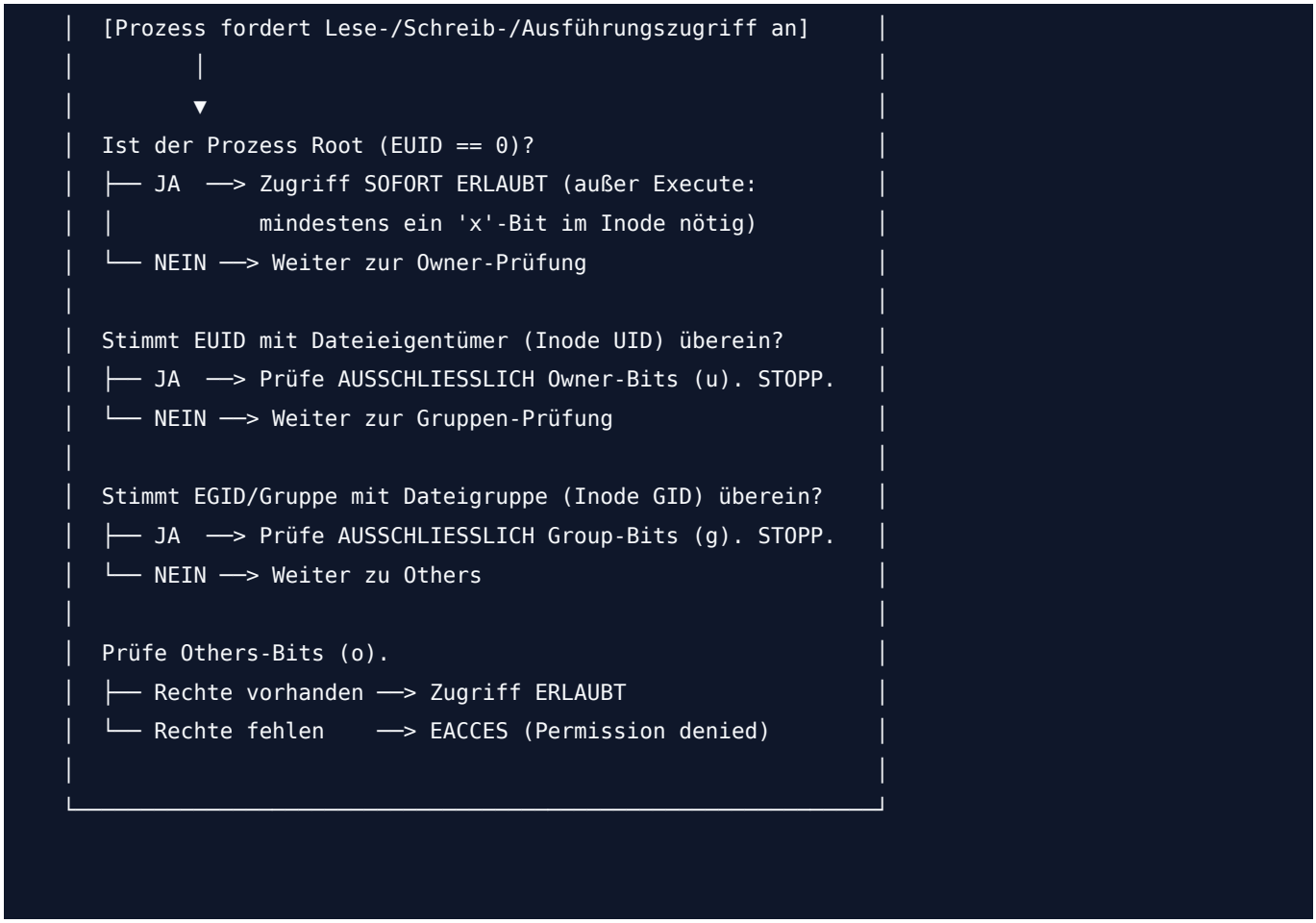
Die oberen 4 Bits (Bits 15–12) definieren den Dateityp nach POSIX-Standard:

- `0100` (Oktal `0100000` / `S_IFREG`): Reguläre Datei
- `0040` (Oktal `0040000` / `S_IFDIR`): Verzeichnis
- `0120` (Oktal `0120000` / `S_IFLNK`): Symbolischer Link
- `0060` (Oktal `0060000` / `S_IFBLK`): Blockorientiertes Gerät (z. B. `/dev/sda`)
- `0020` (Oktal `0020000` / `S_IFCHR`): Zeichenorientiertes Gerät (z. B. `/dev/tty`)
- `0010` (Oktal `0010000` / `S_IFIFO`): Named Pipe (FIFO)
- `0140` (Oktal `0140000` / `S_IFSOCK`): UNIX Domain Socket

Der Kernel-Prüfungsablauf bei Dateizugriff (Discretionary Access Control)

Wenn ein Benutzerprozess versucht, eine Datei zu öffnen (über den System-Call `sys_open` oder `sys_openat`), führt der Linux-Kernel eine Prüfung der **Discretionary Access Control (DAC)** über die VFS-Funktion `generic_permission()` durch:





> ⚠ **Fundamentales Prinzip:** Die Rechteprüfung bricht sofort beim ersten Treffer ab! Wenn der Dateieigentümer beispielsweise die Rechte `000` (keine Rechte) besitzt, die Gruppe aber `777`, kann der Eigentümer die Datei dennoch nicht lesen – selbst wenn er Mitglied der entsprechenden Gruppe ist.

System-Calls im Hintergrund: chmod(), fchmod(), fchmodat()

Wenn du im Terminal `chmod` ausführst, nutzt das Programm unter der Haube standardisierte C-System-Calls aus `<sys/stat.h>`:

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

```
#include <fcntl.h>

// 1. Klassischer Aufruf anhand des Dateipfads
int chmod(const char *pathname, mode_t mode);

// 2. Deskriptor-basierter Aufruf (verhindert Race Conditions)
int fchmod(int fd, mode_t mode);

// 3. Relativer Aufruf zu einem Verzeichnis-Deskriptor (POSIX.1-2008 / POSIX.1-2024)
int fchmodat(int dirfd, const char *pathname, mode_t mode, int flags);
```

Moderne Daemons und Build-Systeme bevorzugen `fchmod()`, da durch das vorherige Öffnen eines File-Descriptors sichergestellt wird, dass die Berechtigungsänderung atomar auf die exakte Datei angewendet wird, ohne dass ein Angreifer zwischenzeitlich einen Symlink an der Pfadstelle platzieren kann (*TOCTOU: Time-of-Check to Time-of-Use Angriff*).

Inode-Metadaten mit dem stat-Befehl inspizieren

Mit dem Werkzeug `stat` lassen sich Inode-Daten direkt auf der Kommandozeile analysieren:

□□ Praktisches Beispiel:

Wir untersuchen die Metadaten der geschützten Shadow-Passwortdatei auf der Konsole und filtern gezielt nach Oktal- und Symbolik-Werten:

```
# Detaillierte Inode-Informationen anzeigen
stat /etc/shadow

# Nur Oktalwert und Dateiname ausgeben
stat -c "%a %n" /etc/shadow

# Symbolische Notation, UID, GID und Dateiname ausgeben
stat -c "%A %U(%u) %G(%g) %n" /etc/shadow
```

Beispielausgabe:

```
  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
```

Elementare Berechtigungen: Dateien vs. Verzeichnisse im Detail

Ein häufiges Missverständnis bei Linux-Einsteigern ist die Übertragung der Rechte-Logik von regulären Dateien auf Verzeichnisse. Die Bits `r`, `w` und `x` haben bei Ordnern eine grundlegend andere Bedeutung:

Recht	Bedeutung bei regulären Dateien	Bedeutung bei Verzeichnissen
Lesen (`r`, Read, 4)	Erlaubt das Auslesen des Datei-Inhalts (z. B. mit `cat`, `less`, Editor).	Erlaubt das Auflisten der im Ordner enthaltenen Dateinamen (z. B. mit `ls`).
Schreiben (`w`, Write, 2)	Erlaubt das Modifizieren oder Überschreiben des Inhalts einer Datei.	Erlaubt das **Erstellen, Umbenennen und Löschen von Dateien** innerhalb des Ordners!
Ausführen (`x`, eXecute, 1)	Erlaubt das Ausführen der Datei als Binärprogramm oder Shellskript.	**Traverse/Search-Bit:** Erlaubt das Betreten des Verzeichnisses (`cd`) und den Zugriff auf Inodes der darin liegenden Dateien.

Die Tücke der Verzeichnis-Schreibrechte

Wer Schreibrechte (`w`) auf ein Verzeichnis besitzt, kann darin liegende Dateien **löschen oder umbenennen**, selbst wenn die betroffene Datei schreibgeschützt (`chmod 400`) ist oder einem völlig anderen Benutzer gehört!

Das Löschen einer Datei verändert nämlich nicht den Datei-Inode, sondern den Verzeichnis-Inode (der den Eintrag aus der Dateiliste austrägt). Aus diesem Grund existiert für globale Sammelverzeichnisse wie `/tmp` das **Sticky Bit**.

□□ Praktisches Beispiel:

Hier demonstrieren wir den Unterschied zwischen Datei- und Verzeichnisrechten in einem Testordner:

```
# Demonstration: Schreibrecht auf Verzeichnis erlaubt Löschen fremder Dateien
mkdir /tmp/testdir
chmod 777 /tmp/testdir

# Benutzer Alice erstellt eine schreibgeschützte Datei:
touch /tmp/testdir/alice_secret.txt
chmod 400 /tmp/testdir/alice_secret.txt

# Benutzer Bob kann die Datei zwar nicht lesen oder editieren, aber problemlos LÖSCHEN:
rm /tmp/testdir/alice_secret.txt  # Funktioniert, da Bob Schreibrecht auf /tmp/testdir
hat!
```

Das Traverse-Bit (`x`) auf Verzeichnissen in der Praxis

Um auf eine Datei wie `/var/log/nginx/access.log` zugreifen zu können, benötigt ein Prozess nicht nur Leserechte auf die Datei selbst, sondern **Ausführungsrechte (`x`) auf jedem einzelnen übergeordneten Verzeichnis im Pfad**:

- `/` (Root-Verzeichnis) muss `x` haben.
- `/var/` muss `x` haben.

Quelle: https://admindocs.de/de/linux-neulinge/basics-bis-best-practices-alles-ueber-chmod-in-linux

- `/var/log/` muss `x` haben.
- `/var/log/nginx/` muss `x` haben.

Fehlt auch nur einem einzigen Ordner im Pfad das `x`-Bit für den aufrufenden Benutzer, verweigert der Kernel den Zugriff mit `EACCES (Permission denied)` – selbst wenn die Datei selbst auf `777` steht!

Mit dem Werkzeug `namei` lässt sich die Berechtigungskette eines Pfades lückenlos auflisten:

□□ Praktisches Beispiel:

Wir prüfen die gesamte Verzeichnishierarchie eines Webserver-Logs mit `namei`:

```
# Berechtigungskette für einen Dateipfad prüfen
namei -l /var/log/nginx/access.log
```

Beispielausgabe:

```
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
```

Sonderfall: Symlinks und Hardlinks

Symbolische Links (*Symlinks*) besitzen im Dateisystem immer die Berechtigungsmaske `lrwxrwxrwx` (`0777`). Bei Operationen auf Symlinks (wie `cat link.txt` oder `nano link.txt`) folgt der Linux-Kernel stets der Zieldatei und prüft ausschließlich deren Inode-Berechtigungen.

Hardlinks hingegen teilen sich denselben physikalischen Inode mit der Originaldatei. Eine Änderung der Berechtigungen via `chmod` an einem Hardlink verändert automatisch die Rechte aller anderen Hardlinks auf denselben Inode.

Numerische (Oktale) vs. Symbolische Notation

`chmod` unterstützt zwei unterschiedliche Notationsformen: die mathematisch-binäre **Oktal-Notation** und die lesbare **symbolische Notation**. Als Einsteiger wirst du beiden Varianten täglich begegnen.

1. Numerische (Oktale) Notation

Jedes Zugriffsrecht entspricht einem festen binären Stellenwert. Wenn du dir die Zahlen 4, 2 und 1 einprägst, kannst du jeden Modus im Kopf im Handumdrehen zusammenrechnen:

- `r` (Read) = 2^2 = **4**
- `w` (Write) = 2^1 = **2**
- `x` (Execute) = 2^0 = **1**
- `-` (Kein Recht) = **0**

Durch Addition dieser Werte ergibt sich eine dreistellige Oktalzahl (bzw. vierstellig bei Spezial-Bits):

Oktalwert	Binärwert	Symbolisch	Bedeutung & typischer Einsatzzweck
`7`	`111`	`rwx`	Vollzugriff: Lesen, Schreiben und Ausführen / Betreten
`6`	`110`	`rw-`	Standard für reguläre Dateien (Lesen und Schreiben)
`5`	`101`	`r-x`	Standard für Verzeichnisse & Programme (Lesen und Ausführen)
`4`	`100`	`r--`	Schreibgeschützt: Nur lesbar (z. B. sensible Konfigurationen)
`3`	`011`	`-wx`	Nur Schreiben und Betreten (seltene Spezialanwendung)
`2`	`010`	`-w-`	Nur Schreiben (z. B. reine Write-Only Dropboxes)

Oktalwert	Binärwert	Symbolisch	Bedeutung & typischer Einsatzzweck
`1`	`001`	`--x`	Nur Ausführen / Betreten (ohne Dateinamen auflisten zu dürfen)
`0`	`000`	`---`	Keinerlei Zugriff für diese Benutzerklasse

□□ Praktisches Beispiel:

Typische chmod-Aufrufe mit Oktalzahlen für unterschiedliche Einsatzzwecke:

```
# Datei für Besitzer les- und schreibbar, für Gruppe und Andere nur lesbar
chmod 644 /var/www/html/index.html

# Skript für Besitzer voll, für alle anderen nur les- und ausführbar
chmod 755 /usr/local/bin/backup.sh

# Sensible Datei (z. B. privater SSH-Schlüssel) nur für Besitzer les- und schreibbar
chmod 600 ~/.ssh/id_ed25519

# Datenbank-Verzeichnis strikt nur für Besitzer zugänglich
chmod 700 /var/lib/postgresql/16/main
```

2. Symbolische Notation

Die symbolische Schreibweise modifiziert bestehende Berechtigungen gezielt, ohne die übrigen Bits überschreiben zu müssen. Die Syntax folgt der Struktur:

$$\text{[Zielgruppe(n)]} \quad [\text{Operator}] \quad [\text{Rechte}]$$

- **Zielgruppen:**
- `u` (*User / Owner*): Dateieigentümer
- `g` (*Group*): Gruppe der Datei
- `o` (*Others*): Alle sonstigen Benutzer
- `a` (*All*): Alle drei Klassen (`u`, `g` und `o` zusammen)

- **Operatoren:**
- `+` : Fügt das angegebene Recht hinzu
- `-` : Entfernt das angegebene Recht
- `=` : Setzt die Rechte exakt auf diesen Wert (überschreibt alte Werte)
- **Rechte-Flags:** `r`, `w`, `x`, `s` (SUID/SGID), `t` (Sticky Bit), `X` (Conditional Execute)

□□ Praktisches Beispiel:

Gezieltes Anpassen einzelner Bits mit symbolischer Notation:

```
# Einer Datei das Ausführungsrecht für den Besitzer hinzufügen
chmod u+x deploy.sh

# Anderen Benutzern alle Rechte an einer Datei entziehen
chmod o-rwx secret.conf

# Gruppe und Andere exakt auf Lese- und Ausführungsrechte setzen
chmod go=rx script.sh

# Allen Benutzern das Leserecht gewähren
chmod a+r public.txt

# Mehrere Änderungen mit Komma trennen:
chmod u=rw,g=r,o= config.json
```

Das bedingte Ausführungs-Bit (`X`)

Bei rekursiven Rechteänderungen (`chmod -R`) steht man oft vor einem Dilemma: Verzeichnisse benötigen zwingend das `x`-Bit (um betreten werden zu können), reguläre Text- oder HTML-Dateien sollten jedoch niemals ausführbar sein.

Mit dem **großen** `X` (*Conditional Execute*) löst `chmod` dieses Problem elegant: Es setzt das Ausführungsrecht **nur auf Verzeichnisse** oder Dateien, die bereits für mindestens eine Klasse ausführbar sind:

Quelle: https://admindocs.de/de/linux-neulinge/basics-bis-best-practices-alles-ueber-chmod-in-linux

□□ Praktisches Beispiel:

Wir härten ein Webverzeichnis rekursiv ab, ohne Textdateien versehentlich ausführbar zu machen:

```
# Sicherer Standard für einen Webroot:
# Ordner werden 755 (rwxr-xr-x), Dateien bleiben 644 (rw-r--r--)
chmod -R u=rwX,go=rX /var/www/html/
```

Referenzkopieren von Rechten (--reference)

`chmod` erlaubt das direkte Klonen von Berechtigungen einer Vorlagedatei auf eine oder mehrere Zieldateien:

□□ Praktisches Beispiel:

Rechte einer bestehenden Konfigurationsdatei 1:1 auf eine neue Datei übertragen:

```
# Rechte von template.conf exakt auf new_service.conf übertragen
chmod --reference=/etc/app/template.conf /etc/app/new_service.conf
```

Die Spezial-Bits: SUID, SGID und Sticky Bit

Neben den standardmäßigen Lese-, Schreib- und Ausführungsrechten stellt POSIX drei mächtige Sonderfunktionen bereit. Sie werden in der vierstelligen Oktalnotation über die erste Ziffer (4, 2, 1) gesteuert.



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

Wenn eine ausführbare Binärdatei das SUID-Bit besitzt, wird der Prozess nicht mit den Rechten des aufrufenden Benutzers ausgeführt, sondern mit den **Rechten des Dateieigentümers** (meist `root`).

- **Symbolische Darstellung:** In der `ls -l`-Ausgabe wird das `x` des Eigentümers durch ein `s` ersetzt (z. B. `-rwsr-xr-x`).
- **Typisches Beispiel:** Das Programm `/usr/bin/passwd`. Ein normaler Benutzer muss sein eigenes Passwort ändern können, was Schreibzugriff auf die geschützte Datei `/etc/shadow` erfordert. Durch das SUID-Bit läuft `passwd` temporär mit Root-Rechten.

□□ Praktisches Beispiel:

Wir demonstrieren das Setzen des SUID-Bits auf eine Binärdatei:

```
# SUID-Bit auf eine Binärdatei setzen
sudo chmod 4755 /usr/local/bin/custom-tool
# Alternativ symbolisch:
sudo chmod u+s /usr/local/bin/custom-tool
```

> ⚠ **Sicherheitsrisiko SUID:** Jede SUID-Root-Binärdatei stellt ein potenzielles Ziel für Privilegien-Eskalation dar. Aus diesem Grund ignoriert der moderne Linux-Kernel das SUID-Bit auf Shell- und Interpreterskripten (`#!/bin/bash` , `#!/usr/bin/env python`) aus Sicherheitsgründen vollständig! Auf Dateisystemen, die von Benutzern beschrieben werden können (wie `/tmp` oder `/home`), sollte zudem die Mount-Option `nosuid` in `/etc/fstab` gesetzt sein.

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

Das SGID-Bit hat zwei unterschiedliche Anwendungsbereiche:

1. **Auf Binärdateien:** Der Prozess läuft mit den Berechtigungen der Dateigruppe.
2. **Auf Verzeichnissen (wichtig für Teamarbeit):** Normalerweise erhalten neu erstellte Dateien als primäre Gruppe die Standardgruppe des erstellenden Benutzers. Ist auf einem Verzeichnis das SGID-Bit gesetzt, **erben alle neu erstellten Dateien und Unterordner automatisch die Gruppenmitgliedschaft des übergeordneten Verzeichnisses**.

□□ Praktisches Beispiel:

Wir richten ein Verzeichnis für ein Entwicklerteam mit automatischer Gruppenvererbung ein:

```
# Verzeichnis für gemeinsame Teamarbeit einrichten
```

```
sudo mkdir /srv/project-files
sudo chown root:developers /srv/project-files

# SGID-Bit setzen (Rechte: drwxrws---)
sudo chmod 2770 /srv/project-files
```

3. Sticky Bit (Restricted Deletion – Oktal `1000` / `+t`)

Das Sticky Bit verhindert das unbefugte Löschen oder Umbenennen von Dateien in Verzeichnissen, auf die mehrere Benutzer Schreibzugriff haben.

- **Symbolische Darstellung:** Das Ausführungsbit für Andere wird als `t` dargestellt (z. B. `drwxrwxrwt`).
- **Verhalten:** In einem Ordner mit Sticky Bit darf eine Datei ausschließlich gelöscht oder umbenannt werden von:

1. Dem Eigentümer der Datei,
2. Dem Eigentümer des Verzeichnisses,
3. Dem Administrator (`root`).

□□ Praktisches Beispiel:

Einrichten eines globalen temporären Austauschverzeichnisses mit Schutz vor fremdem Löschen:

```
# Sticky Bit auf ein Verzeichnis setzen
sudo chmod 1777 /tmp/shared-scratch/
# Alternativ symbolisch:
sudo chmod +t /tmp/shared-scratch/
```

Großes `S` vs. kleines `s` / Großes `T` vs. kleines `t`

Wenn du in `ls -l` ein großes `S` oder großes `T` siehst, bedeutet dies: **Das Spezial-Bit ist aktiv, aber das zugrundeliegende Ausführungs-Bit (`x`) fehlt!**

- `-rwSr-xr-x`: SUID aktiv, aber Besitzer darf Datei nicht ausführen (Fehlkonfiguration).
- `drwxrwx--T`: Sticky Bit aktiv, aber Andere dürfen Ordner nicht betreten.
- `-rwsr-xr-x`: Korrekt gesetztes SUID inklusive Execute (`s` klein).
- `drwxrwxrwt`: Korrekt gesetztes Sticky Bit inklusive Execute (`t` klein).

Moderne Alternative zu SUID: Linux File Capabilities

Um Programmen gezielt einzelne Kernel-Privilegien zu gewähren, ohne ihnen komplette Root-Rechte via SUID zu verleihen, nutzt modernes Linux **Capabilities** (`setcap` / `getcap`):

□□ Praktisches Beispiel:

Einem benutzerdefinierten Dienst erlauben, privilegierte Ports zu öffnen, ohne Root zu sein:

```
# Beispiel: Einem Webserver erlauben, auf privilegierten Ports (<1024) zu lauschen:
sudo setcap 'cap_net_bind_service=+ep' /usr/local/bin/custom-webserver

# Aktive Capabilities einer Binärdatei prüfen:
getcap /usr/local/bin/custom-webserver
```

Standardberechtigungen & umask

Wenn ein Prozess eine neue Datei oder einen neuen Ordner anlegt, vergibt der Kernel nicht sofort statische Rechte, sondern wendet eine Bitmaske namens **`umask`** (*User Creation Mask*) an.

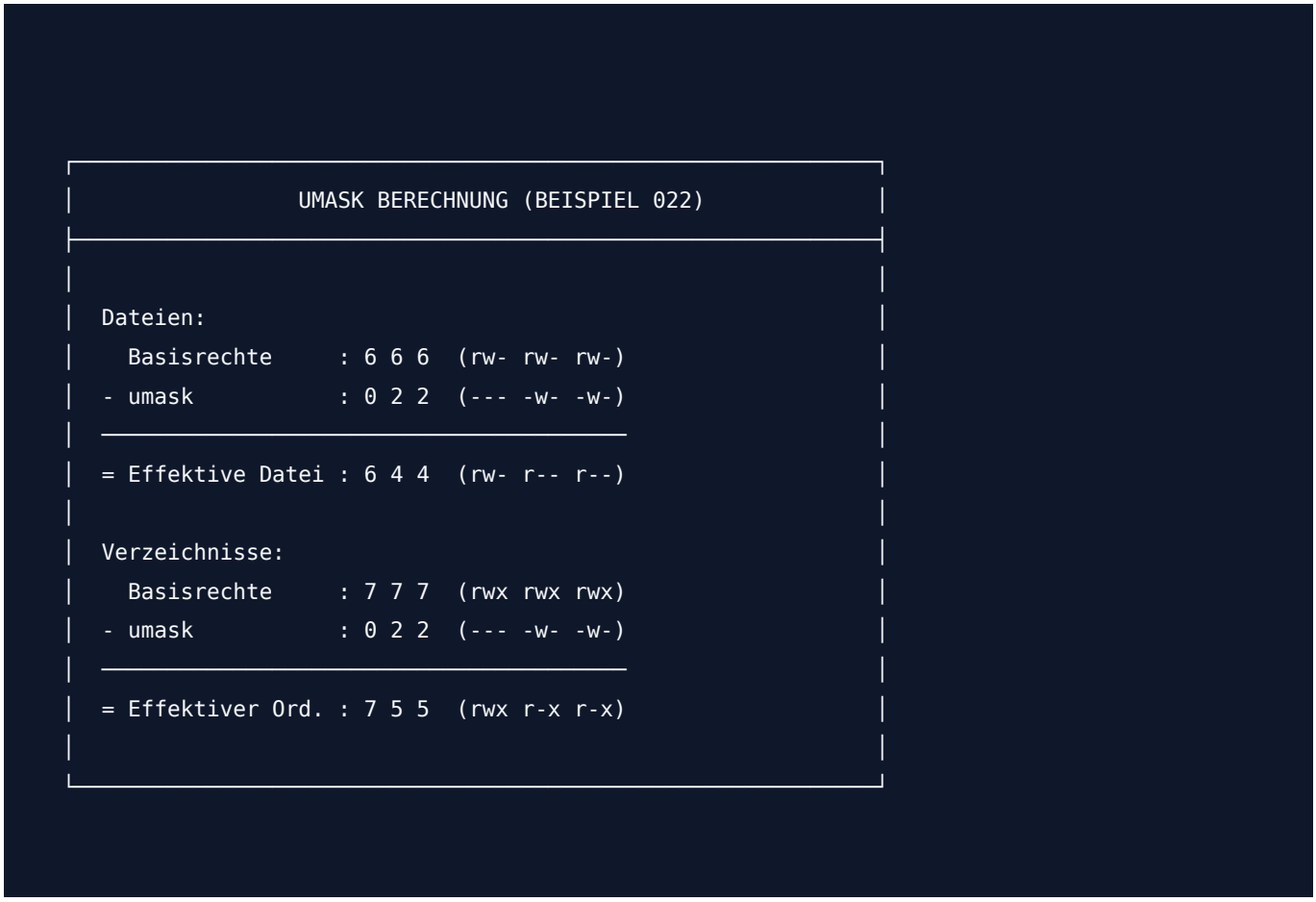
Wie die umask berechnet wird

Der Linux-Kernel geht von maximalen Basisrechten aus:

- **Neue Dateien:** Basis `666` (`rw-rw-rw-` – kein Execute aus Sicherheitsgründen)
- **Neue Verzeichnisse:** Basis `777` (`rwxrwxrwx` – inkl. Execute zum Betreten)

Die `umask` maskiert (subtrahiert) unerwünschte Bits über eine bitweise `AND NOT`-Operation:

$$\text{Effektive Rechte} = \text{Basisrechte} \land (\neg \text{umask})$$



Typische umask-Werte im Vergleich:

umask-Wert	Effektive Dateirechte	Effektive Ordnerrechte	Einsatzzweck
`0022`	`644` (`rw-r--r--`)	`755` (`rwxr-xr-x`)	Standard für Desktop- & Workstation-Systeme
`0027`	`640` (`rw-r-----`)	`750` (`rwxr-x---`)	Gehärtete Serverumgebung (Andere haben keinerlei Zugriff)
`0077`	`600` (`rw-------`)	`700` (`rwx------`)	Hochsicherheitsumgebungen (Nur Besitzer hat Zugriff)

Quelle: https://admindocs.de/de/linux-neulinge/basics-bis-best-practices-alles-ueber-chmod-in-linux

Praktisches Beispiel:

Umask in der aktuellen Shell abfragen und anpassen:

```
# Aktuelle umask der Shell anzeigen
umask

# Umask temporär auf restriktiv setzen
umask 027
```

Permanente Konfiguration der umask

Um die `umask` systemweit oder benutzerspezifisch festzulegen:

1. **Systemweit in `/etc/login.defs`:**

   ```
   UMASK 027
   ```

2. **Benutzerspezifisch in `~/.bashrc` oder `~/.profile`:**

   ```
   umask 027
   ```

3. **In systemd-Service-Units:**

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

Erweiterte Attribute & Alternativen: chattr & POSIX ACLs

Klassische POSIX-Dateirechte stoßen in zwei Situationen an ihre Grenzen: wenn Dateien selbst vor dem Root-Benutzer geschützt werden sollen oder wenn mehr als eine Gruppe unterschiedliche Zugriffsrechte benötigt.

1. Unveränderliche Dateien mit `chattr`

Das Werkzeug `chattr` (*change attribute*) manipuliert erweiterte Dateisystem-Flags (unterstützt von ext4, XFS und BTRFS):

□□ Praktisches Beispiel:

Wir machen eine zentrale Systemdatei unveränderlich und testen den Schutz:

```
# Datei unveränderlich machen (selbst root kann sie nicht löschen oder editieren)
sudo chattr +i /etc/resolv.conf

# Attribute überprüfen
lsattr /etc/resolv.conf

# Sperre wieder aufheben
sudo chattr -i /etc/resolv.conf

# Datei auf Append-Only setzen (ideal für Logfiles: nur anhängen erlaubt)
```

```
sudo chattr +a /var/log/critical-audit.log
```

> **Ransomware-Schutz:** Das Setzen des `+i`-Attributs auf Offline-Backups oder Konfigurationsverzeichnisse verhindert, dass automatisierte Schadskripte oder Ransomware bestehende Dateien im System überschreiben oder verschlüsseln.

Übersicht der wichtigsten chattr-Attribute

Flag	Attribut	Bedeutung & Verhalten
`+i`	Immutable	Datei kann nicht modifiziert, gelöscht, umbenannt oder verlinkt werden (selbst von Root).
`+a`	Append-only	Datei kann ausschließlich geöffnet werden, um Daten anzuhängen (ideal für Audit-Logs).
`+A`	No atime	Aktualisiert bei Lesezugriffen nicht den Access-Timestamp (spart I/O-Performance).
`+d`	No dump	Datei wird vom klassischen Backup-Tool `dump` ignoriert.
`+s`	Secure deletion	Beim Löschen werden die Datenblöcke auf der Festplatte mit Nullen überschrieben.
`+c`	Compressed	Kernel komprimiert Datenblöcke transparent auf dem Datenträger (Dateisystem-abhängig).

2. Feingranulare Rechte mit POSIX ACLs (`setfacl` & `getfacl`)

Wenn ein dritter Benutzer (z. B. ein Backup-Dienst) Lesezugriff auf eine Datei benötigt, ohne ihn zum Owner zu machen oder der Hauptgruppe hinzuzufügen, kommen **Access Control Lists (ACLs)** zum Einsatz:

Praktisches Beispiel:

Gezielte Sonderberechtigungen für einzelne Systembenutzer vergeben:

```
# Benutzer 'backupuser' gezielt Leserechte gewähren
setfacl -m u:backupuser:r /etc/shadow

# Aktuelle ACLs einer Datei einsehen
getfacl /etc/shadow

# Alle erweiterten ACLs wieder entfernen
setfacl -b /etc/shadow
```

Vererbung mit Default-ACLs auf Verzeichnissen

Mit dem Schalter `-d` (*default*) wird eine ACL definiert, die automatisch auf alle künftig in diesem Verzeichnis erstellten Dateien und Unterordner vererbt wird:

□□ Praktisches Beispiel:

Wir konfigurieren eine Standard-ACL für ein Gruppenverzeichnis und sichern die Regeln:

```
# Standard-ACL für Entwicklergruppe auf Verzeichnis vererben
sudo setfacl -d -m g:devteam:rwx /srv/project-files/

# Backup aller ACLs eines Verzeichnisbaums erstellen
getfacl -R /srv/project-files > /backup/permissions_acl.bak

# Gesicherte ACLs auf ein wiederhergestelltes Verzeichnis anwenden
setfacl --restore=/backup/permissions_acl.bak
```

Linux Capabilities: Granulare Rechte ohne SUID-Root

Traditionell unterteilt Unix Prozesse in zwei Klassen: **unprivilegiert** (UID != 0, DAC-Prüfungen greifen vollständig) und **privilegiert** (UID == 0, Root umgeht alle DAC-Prüfungen).

Dieses Alles-oder-Nichts-Prinzip ist riskant: Erhält ein Dienst wie ein Webserver SUID-Root, nur um Port 80 binden zu können, besitzt er im Falle einer Sicherheitslücke die volle Kontrolle über den gesamten Server.

Linux löst dieses Problem durch **Capabilities** (basierend auf dem POSIX.1e-Entwurf), die Root-Privilegien in über 40 diskrete Einzelberechtigungen zerlegen.

Die wichtigsten Linux Capabilities im Überblick

Capability	Funktion & Erlaubnis	Typischer Einsatzzweck
`CAP_NET_BIND_SERVICE`	Binden von Sockets an privilegierte Ports (< 1024)	Webserver (Nginx, Caddy), DNS-Server (Named)
`CAP_NET_RAW`	Erstellen von RAW- und Packet-Sockets	Netzwerk-Tools (`ping` , `traceroute` , `tcpdump`)
`CAP_SYS_TIME`	Ändern der Systemuhrzeit und Hardware-RTC	Zeitsynchronisation (`chronyd` , `systemd-timesyncd`)
`CAP_DAC_OVERRIDE`	Umgehen aller Lese-, Schreib- und Execute-Prüfungen	Backup-Tools (Bacula, Restic, BorgBackup)
`CAP_CHOWN`	Beliebiges Ändern von Datei-Eigentümern und Gruppen	Dateiserver, Container-Runtimes
`CAP_SYS_ADMIN`	Umfangreiche Administrationsrechte (Mount, Quota, BPF)	Container-Engines (Docker, Podman, LXC)
`CAP_SETUID`	Beliebiges Wechseln von Prozess-UIDs	Login-Daemons (SSHD, Display Manager)

Verwaltung von Datei-Capabilities (setcap & getcap)

Capabilities werden in den erweiterten Attributen des Dateisystems (`security.capability`) gespeichert:

□□ Praktisches Beispiel:

Wir prüfen bestehende Datei-Capabilities und statten ein Programm mit Netzwerk-Rechten aus:

```
# 1. Status prüfen: Besitzt ping Capabilities oder SUID?
ls -l /bin/ping
getcap /bin/ping

# 2. Ping mit RAW-Netzwerk-Capabilities ausstatten (kein SUID-Root nötig!):
sudo setcap 'cap_net_raw=+ep' /bin/ping

# 3. Webserver erlauben, Port 443 ohne Root-Rechte zu binden:
sudo setcap 'cap_net_bind_service=+ep' /usr/local/bin/my-web-app

# 4. Alle Capabilities von einer Datei restlos entfernen:
sudo setcap -r /usr/local/bin/my-web-app
```

Erklärung der Capability-Flags:

- `e` (*Effective*): Die Capability ist für den Thread sofort aktiv.
- `p` (*Permitted*): Der Prozess darf diese Capability nutzen oder aktivieren.
- `i` (*Inheritable*): Die Capability wird über System-Calls (`execve`) an Kindprozesse vererbt.

Automatisierung: Berechtigungen in Ansible & Shell-Skripten

In modernen DevOps- und Server-Infrastrukturen werden Dateirechte nicht mehr manuell getippt, sondern deklarativ über Konfigurationsmanagement verwaltet.

1. Berechtigungen in Ansible Playbooks definieren

Grundlegende Konzepte zu Playbooks und Modulen vermittelt der Leitfaden Ansible Grundlagen für Linux-Administratoren.

Das Ansible-Modul `ansible.builtin.file` bietet native Unterstützung für symbolische und oktale Berechtigungen sowie Dateisystem-Attribute:

□□ Praktisches Beispiel:

Deklarative Absicherung von Webroots und Secrets per Ansible Playbook:

```
---
- name: Webserver-Berechtigungen und Konfiguration härten
  hosts: webservers
  become: true
  tasks:
    - name: Webroot-Verzeichnis mit korrekten Rechten anlegen
      ansible.builtin.file:
        path: /var/www/production
        state: directory
        owner: webadmin
        group: www-data
        mode: '0755'

    - name: Sensible Konfigurationsdatei absichern und sperren
      ansible.builtin.file:
        path: /etc/app/production.secret.env
        state: file
        owner: root
        group: root
        mode: '0600'
        attributes: '+i'

    - name: Rekursive Rechtevergabe mit bedingtem Execute-Bit
      ansible.builtin.file:
        path: /srv/shared_assets
        state: directory
        owner: deployer
        group: devteam
        mode: 'u=rwX,g=rX,o='
        recurse: true
```

2. Wiederverwendbares Bash-Skript zur Rechte-Harmonisierung

Vertiefende Shell-Techniken und Automatisierungs-Muster behandelt unser Modul Shell-Scripting und Automatisierung.

Für lokale Wartungsarbeiten oder Deployment-Pipelines hilft ein dediziertes Shell-Skript, das Verzeichnisse und Dateien atomar bereinigt:

□□ Praktisches Beispiel:

Ein praxiserprobtes Wartungsskript zur Bereinigung verstellter Berechtigungen:

```
#!/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 "Fehler: Verzeichnis $TARGET_DIR existiert nicht!" >&2
    exit 1
fi

echo "Harmonisiere Berechtigungen für: $TARGET_DIR"

# 1. Alle Verzeichnisse auf 755 (drwxr-xr-x) setzen
find "$TARGET_DIR" -type d -exec chmod 755 {} +

# 2. Alle regulären Dateien auf 644 (-rw-r--r--) setzen
find "$TARGET_DIR" -type f -exec chmod 644 {} +

# 3. Shell-Skripte gezielt wieder ausführbar machen
find "$TARGET_DIR" -type f -name "*.sh" -exec chmod 750 {} +

# 4. Sensible .env und .key Dateien isolieren
find "$TARGET_DIR" -type f \( -name "*.env" -o -name "*.key" -o -name "*.pem" \) -exec
chmod 600 {} +

echo "Berechtigungen erfolgreich aktualisiert."
```

3. Git und Dateiberechtigungen

Ein wichtiger Aspekt in Softwareprojekten: **Git speichert keine vollständigen Linux-Berechtigungen!**

Git speichert im Index ausschließlich ein einziges Berechtigungs-Flag:

- `100644`: Reguläre Datei (nicht ausführbar)
- `100755`: Ausführbare Datei (Skripte, Binaries)

□□ Praktisches Beispiel:

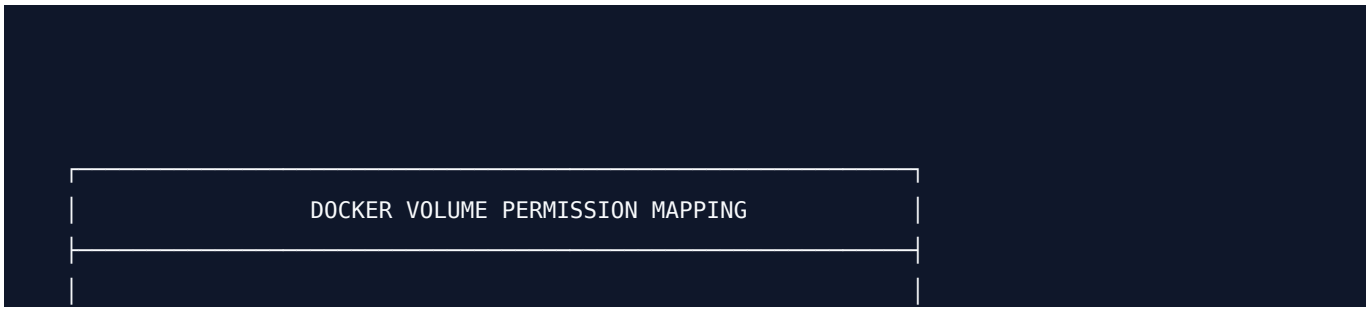
Skriptdateien direkt im Versionskontrollsystem als ausführbar deklarieren:

```
# Eine Datei im Git-Index explizit als ausführbar markieren (ohne chmod auf Host):
git update-index --chmod=+x deploy.sh

# Prüfen, welchen Modus Git für eine Datei speichert:
git ls-files --stage deploy.sh
```

Container, Docker & Kubernetes: UID/GID-Mapping

Ein klassisches Problem beim Betrieb von Docker-Containern (siehe auch Docker unter Ubuntu installieren und einrichten) sind Berechtigungskonflikte auf gemounteten Volumes (`-v /host/data:/app/data`).





> ⚠ **Anti-Pattern:** Setze niemals `chmod -R 777` auf Host-Volumes, um Docker-Berechtigungsfehler zu lösen! Dadurch verliert das System jegliche Isolation und jeder unprivilegierte Benutzer auf dem Host kann Container-Dateien manipulieren.

Kubernetes SecurityContext & fsGroup

Details zu Pod-Sicherheitsstandards und Cluster-Architekturen findest du in unserer Kubernetes-Dokumentation.

In Kubernetes wird die Rechteverwaltung im Pod-Manifest über den `securityContext` gesteuert:

□□ Praktisches Beispiel:

Ein Kubernetes-Pod-Manifest mit strikter Benutzer- und Gruppen-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: {}
```

Durch `fsGroup: 10001` ändert Kubernetes beim Mounten des Volumes automatisch die Gruppenberechtigung aller enthaltenen Dateien auf GID 10001 und setzt das Gruppen-Schreibrecht.

10 Praxisszenarien & Server-Härtung (Playbooks)

1. LAMP & LEMP Stack (WordPress & Nginx)

Eine vollständige Anleitung zur Server-Einrichtung findest du in unserem LEMP-Stack Praxisleitfaden auf Ubuntu.

Die goldene Regel lautet: **Der Webserver-Benutzer (`www-data`) darf Dateien lesen und ausführen, aber niemals standardmäßig im Webroot schreiben!**

□□ Praktisches Beispiel:

Wir setzen die Rechte eines typischen Webroots nach dem Least-Privilege-Prinzip:

```
# 1. Eigentümer auf Systembenutzer, Gruppe auf www-data setzen
sudo chown -R webadmin:www-data /var/www/html/

# 2. Verzeichnisse auf 755 und Dateien auf 644 setzen
sudo find /var/www/html/ -type d -exec chmod 755 {} +
sudo find /var/www/html/ -type f -exec chmod 644 {} +

# 3. Ausschließlich das Upload-Verzeichnis für den Webserver beschreibbar machen
sudo chown -R www-data:www-data /var/www/html/wp-content/uploads/
sudo chmod -R 775 /var/www/html/wp-content/uploads/

# 4. Ausführung von Skripten im Upload-Verzeichnis verbieten (Nginx-Konfiguration)
# location ~* /uploads/.*\.php$ { deny all; }
```

2. SSH-Berechtigungen abhärten

Für weitergehende Schutzmaßnahmen inklusive 2FA und Intrusion Prevention verweisen wir auf unseren Guide zur Linux-Server-Härtung mit FIDO2 und CrowdSec.

Der OpenSSH-Daemon verweigert die Authentifizierung, wenn Schlüsseldateien oder das `.ssh`-Verzeichnis zu offene Berechtigungen aufweisen (*„UNPROTECTED PRIVATE KEY FILE“*):

□□ Praktisches Beispiel:

SSH-Schlüssel und Konfigurationsdateien normgerecht absichern:

```
# SSH-Konfigurationsordner absichern
chmod 700 ~/.ssh

# Private Schlüsseldateien (strikt nur für Besitzer)
chmod 600 ~/.ssh/id_ed25519 ~/.ssh/id_rsa
```

```
# Öffentliche Schlüssel und authorized_keys
chmod 644 ~/.ssh/id_ed25519.pub
chmod 600 ~/.ssh/authorized_keys

# Systemweite SSH-Host-Keys (/etc/ssh/)
sudo chmod 600 /etc/ssh/ssh_host_*_key
sudo chmod 644 /etc/ssh/ssh_host_*_key.pub
```

3. Datenbank-Server absichern

Datenbank-Engines wie PostgreSQL oder MariaDB verweigern den Dienst, wenn ihre Datenspeicher für andere Benutzer einsehbar sind:

□□ Praktisches Beispiel:

Datenbank-Verzeichnisse vor unbefugtem Zugriff schützen:

```
# 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. Samba & NFS Dateifreigaben absichern

Bei Netzwerk-Dateiservern (Samba/NFS) müssen Linux-Dateirechte mit den Rechten der Netzwerkprotokolle synchronisiert werden:

□□ Praktisches Beispiel:

Konfigurationsbeispiel 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 für Team-Server

Wenn mehrere Entwickler per SSH auf ein zentrales Git-Repository auf einem Server pushen:

□□ Praktisches Beispiel:

Zentrales Bare-Repository mit gemeinsamer Gruppenberechtigung anlegen:

```
# Zentrales Bare-Repository mit Shared-Group Berechtigungen erstellen
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. Cronjobs und System-Skripte absichern

Skripte unter `/etc/cron.*` oder `/usr/local/sbin` müssen vor Manipulation durch unprivilegierte Benutzer geschützt werden:

□□ Praktisches Beispiel:

Systemwartungs-Skripte gegen unbefugte Modifikation sichern:

```
# Cron-Skripte strikt nur für root schreib- und ausführbar
sudo chown root:root /usr/local/sbin/daily-backup.sh
sudo chmod 700 /usr/local/sbin/daily-backup.sh
```

7. Temporäre Ausführungsverzeichnisse (/tmp & /var/tmp)

□□ Praktisches Beispiel:

Sicherstellen, dass das Sticky Bit auf temporären Systemordnern aktiv ist:

```
# Sicherstellen, dass das Sticky Bit auf /tmp aktiv ist
sudo chmod 1777 /tmp /var/tmp
```

8. Zertifikate und TLS-Private-Keys absichern

□□ Praktisches Beispiel:

Private Schlüsseldateien für TLS-Zertifikate strikt absichern:

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

9. Logfile-Verzeichnisse für Web- & App-Dienste

□□ Praktisches Beispiel:

Logverzeichnisse mit Gruppenleserecht für Administratoren und Audit-Tools konfigurieren:

```
# Logverzeichnis für 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 Boxen (Schreiben ohne fremdes Lesen)

□□ Praktisches Beispiel:

Ein Postfach-Ordner, in den jeder Benutzer Dateien ablegen, aber nicht die Dateien anderer einsehen darf:

```
# Ein Postfach-Ordner, in den jeder Dateien ablegen, aber nicht die Dateien anderer
einsehen darf:
sudo mkdir /srv/dropin
sudo chmod 1733 /srv/dropin  # rwx-wx-wx mit Sticky Bit
```

5 Schritt-für-Schritt Labor-Übungen (Hands-On)

Um die theoretischen Konzepte in praktische Administrations-Fertigkeiten zu überführen, spielen wir fünf typische Praxis-Szenarien durch.

Labor 1: Kollaboratives Entwickler-Verzeichnis mit SGID & Default-ACLs

Szenario: Die Entwickler `alice` und `bob` gehören zur Gruppe `devteam`. Sie benötigen einen gemeinsamen Projektordner `/srv/devproject`, in dem jede neu erstellte Datei sofort für beide beschreibbar ist – ohne manuelle Rechteanpassung.

□□ Praktisches Beispiel:

Schrittweise Einrichtung des Team-Ordners mit SGID und Default-ACLs:

```
# 1. Gruppe und Benutzer anlegen (falls nicht vorhanden)
sudo groupadd devteam
sudo usermod -aG devteam alice
sudo usermod -aG devteam bob

# 2. Verzeichnis erstellen und Berechtigungen setzen
sudo mkdir -p /srv/devproject
sudo chown root:devteam /srv/devproject
sudo chmod 2770 /srv/devproject

# 3. Default-ACLs für maximale Zusammenarbeit einrichten
sudo setfacl -d -m g:devteam:rwx /srv/devproject
sudo setfacl -m g:devteam:rwx /srv/devproject

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

Ergebnis: Die Datei `api.py` gehört automatisch der Gruppe `devteam` und ist für Bob sofort editierbar.

Labor 2: Forensische Analyse & Beseitigung eines SUID-Backdoors

Szenario: Nach einem Sicherheitsvorfall soll das System auf unbefugt gesetzte SUID-Binaries in temporären Ordnern untersucht werden.

□□ Praktisches Beispiel:

Aufspüren und Entschärfen verdächtiger SUID-Dateien:

```
# 1. Systemweiten Scan nach SUID-Dateien starten
sudo find / -perm -4000 -type f 2>/dev/null > /tmp/suid_scan.txt

# 2. Verdächtige Pfade in /tmp, /dev/shm oder /var/tmp prüfen
grep -E '^/tmp|^/dev/shm|^/var/tmp' /tmp/suid_scan.txt || echo "Keine SUID-Dateien in
Temp-Verzeichnissen!"

# 3. Gefundene Backdoor-Datei entschärfen:
# sudo chmod u-s /tmp/.hidden_root_shell
# sudo rm -f /tmp/.hidden_root_shell
```

Labor 3: WordPress-Härtung mit unveränderlicher Konfiguration

Szenario: Ein WordPress-Webroot soll so gehärtet werden, dass Angreifer selbst bei einer RCE-Schwachstelle im Theme die zentrale Konfigurationsdatei `wp-config.php` weder manipulieren noch überschreiben können.

□□ Praktisches Beispiel:

Härtung und Schutz durch das Immutable-Attribut:

```
# 1. Basisrechte setzen (Eigentümer webadmin, Gruppe 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. wp-config.php auf 640 setzen und für webadmin/www-data sperren
sudo chmod 640 /var/www/wordpress/wp-config.php
sudo chown webadmin:www-data /var/www/wordpress/wp-config.php
```

```
# 3. Immutable-Bit aktivieren (selbst Root kann die Datei jetzt nicht löschen):
sudo chattr +i /var/www/wordpress/wp-config.php

# 4. Verifikation des Schutzes
lsattr /var/www/wordpress/wp-config.php
```

Labor 4: Sichere SFTP-Only Chroot-Jail-Umgebung

Szenario: Ein externer Dienstleister (`sftpuser`) soll per SFTP Dateien auf den Server hochladen dürfen, darf jedoch das System weder per SSH betreten noch andere Verzeichnisse einsehen.

□□ Praktisches Beispiel:

Isoliertes Chroot-Verzeichnis für SFTP-Zugriffe aufsetzen:

```
# 1. SFTP-Gruppe und Benutzer anlegen
sudo groupadd sftpusers
sudo useradd -g sftpusers -d /srv/sftp/sftpuser -s /usr/sbin/nologin sftpuser

# 2. Chroot-Verzeichnis vorbereiten (MUSS zwingend root:root 755 gehören!)
sudo mkdir -p /srv/sftp/sftpuser/uploads
sudo chown root:root /srv/sftp/sftpuser
sudo chmod 755 /srv/sftp/sftpuser

# 3. Upload-Ordner für den SFTP-Benutzer beschreibbar machen
sudo chown sftpuser:sftpusers /srv/sftp/sftpuser/uploads
sudo chmod 750 /srv/sftp/sftpuser/uploads

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

Labor 5: SELinux-Kontexte vs. DAC-Berechtigungen (RHEL/Fedora)

Szenario: Trotz `chmod 777` meldet Nginx auf einem Fedora/RHEL-System beim Zugriff auf ein HTML-Dokument `403 Forbidden`.

□□ Praktisches Beispiel:

SELinux-Kontexte analysieren und reparieren:

```
# 1. DAC-Rechte prüfen:
ls -l /var/www/html/index.html

# 2. SELinux-Sicherheitskontexte anzeigen (-Z Flag):
ls -Z /var/www/html/index.html

# Falls der Typ nicht httpd_sys_content_t lautet:
sudo restorecon -Rv /var/www/html/

# Alternativ manuell korrekten Kontext zuweisen:
sudo chcon -t httpd_sys_content_t /var/www/html/index.html
```

Systemd-Services & Sandbox-Härtung

Moderne Linux-Systeme nutzen `systemd`, um Hintergrunddienste sicher zu isolieren. Anstatt sich ausschließlich auf herkömmliche Dateisystem-Berechtigungen zu verlassen, bieten Service-Units integrierte Dateisystem-Filter und dynamische Benutzerverwaltung:

□□ Praktisches Beispiel:

Eine gehärtete systemd-Service-Unit mit strikter Isolation und dynamischem Benutzer:

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

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

# Systemd Dateisystem-Härtung:
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
```

Erklärung der Sicherheitsdirektiven:

- `DynamicUser=yes`: Erzeugt zur Laufzeit automatisch eine temporäre, unprivilegierte UID/GID und verwirft diese beim Stoppen des Dienstes wieder – keine manuellen System-User nötig!
- `ProtectSystem=strict`: Mountet das gesamte Dateisystem (`/usr`, `/boot`, `/etc`) als **Read-Only** für den Dienstprozess.
- `ReadWritePaths=`: Schaltet ausschließlich die zwingend benötigten Verzeichnisse für Schreibzugriffe frei.

- `PrivateTmp=true`: Erzeugt ein isoliertes `/tmp`-Verzeichnis für den Dienst, wodurch andere Systembenutzer temporäre Dateien nicht einsehen können.
- `UMask=0027`: Erzwingt, dass alle vom Dienst erstellten Dateien automatisch vor Unbefugten geschützt werden.

Troubleshooting & Sicherheits-Auditing

Die 10 häufigsten Berechtigungsfehler in der Praxis

1. **„Permission denied“ trotz `chmod 777` auf die Datei:**

- **Ursache:** Ein übergeordnetes Verzeichnis im Pfad besitzt kein Ausführungsbit (`x`).
- **Lösung:** Mit `namei -l /pfad/zur/datei` den gesamten Pfadbaum auf fehlende `x`-Bits prüfen.

1. **Skript mit `755` lässt sich nicht ausführen:**

- **Ursache:** Das Dateisystem ist mit der Option `noexec` in `/etc/fstab` gemountet (z. B. auf `/tmp` oder externen USB-Sticks).
- **Lösung:** `mount | grep "noexec"` prüfen.

1. **SUID-Skript wird ignoriert:**

- **Ursache:** Der Linux-Kernel ignoriert SUID auf Shell- und Python-Skripten.
- **Lösung:** Für Skript-Eskalation stattdessen `sudoers` mit `NOPASSWD` oder ein C-Wrapper-Binary verwenden.

1. **Datei lässt sich selbst von `root` nicht löschen (*„Operation not permitted“*):**

- **Ursache:** Das Dateisystem-Attribut `+i` (Immutable) ist aktiv.
- **Lösung:** Mit `lsattr dateiname` prüfen und mit `sudo chattr -i dateiname` entfernen.

1. **POSIX-ACL Mask blockiert effektive Rechte:**

- **Ursache:** Die ACL-Maske schränkt Gruppenrechte ein.
- **Lösung:** `getfacl datei` prüfen und mit `setfacl -m m:rwx datei` die Maske öffnen.

Quelle: https://admindocs.de/de/linux-neulinge/basics-bis-best-practices-alles-ueber-chmod-in-linux

1. **SSH-Schlüssel wird abgewiesen (*„Permissions 0644 are too open“*):**

- **Ursache:** Private SSH-Keys dürfen nur für den Besitzer lesbar sein.
- **Lösung:** `chmod 600 ~/.ssh/id_ed25519` ausführen.

1. **Webserver kann hochgeladene Dateien nicht überschreiben:**

- **Ursache:** Upload-Ordner gehört `root` oder hat falsche Gruppenberechtigungen.
- **Lösung:** `sudo chown -R www-data:www-data uploads/ && sudo chmod 775 uploads/`.

1. **Docker-Container wirft `EACCES` auf gemountetem Host-Ordner:**

- **Ursache:** Prozess im Container läuft unter UID 1000, Host-Ordner gehört Root.
- **Lösung:** `sudo chown -R 1000:1000 /host/pfad` oder POSIX-ACLs setzen.

1. **Gelöschte Datei belegt weiterhin Speicherplatz:**

- **Ursache:** Ein Prozess hält die Datei noch im RAM geöffnet (`open file descriptor`).
- **Lösung:** Mit `lsof | grep deleted` ermitteln und den entsprechenden Dienst neu starten.

1. **Neuer Benutzer kann keine Dateien in geteiltem Gruppenordner erstellen:**

- **Ursache:** Der Benutzer hat sich nach dem Hinzufügen zur Gruppe nicht neu eingeloggt (Gruppentoken im Kernel ist veraltet).
- **Lösung:** Neu einloggen oder `newgrp gruppenname` in der Shell ausführen.

Vollständiges Sicherheits-Audit-Skript für Cronjobs

Dieses Shell-Skript durchsucht das gesamte Dateisystem nach Sicherheitsrisiken und generiert einen übersichtlichen Audit-Report:

□□ Praktisches Beispiel:

Ein automatisiertes Inspektionsskript für regelmäßige Sicherheitsprüfungen:

```
#!/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 (Potenzielle Root-Eskalation):" >> "$REPORT_FILE"
find / -perm -4000 -type f -exec ls -ld {} + 2>/dev/null >> "$REPORT_FILE"

echo -e "\n[2] SGID Binaries & 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 abgeschlossen. Report liegt unter: $REPORT_FILE"
```

Echtzeit-Überwachung von chmod mit dem Linux Audit Framework (auditd)

Um verdächtige Berechtigungsänderungen im System lückenlos zu protokollieren:

□□ Praktisches Beispiel:

Audit-Regel für chmod-Aufrufe definieren und analysieren:

```
# 1. Audit-Regel für den System-Call 'chmod' definieren
sudo auditctl -a always,exit -F arch=b64 -S chmod -S fchmod -S fchmodat -k chmod_monitor

# 2. Audit-Logs nach Änderungen durchsuchen
sudo ausearch -k chmod_monitor --format text
```

Dateisystem-Unterschiede: Ext4, XFS, BTRFS & ZFS

Dateisysteme unterscheiden sich fundamental in der Art und Weise, wie sie POSIX-Berechtigungen, Extended Attributes und Inodes verwalten:

Dateisystem	Inode-Struktur	POSIX ACL Support	Immutable (+i) Support	Besonderheiten bei Berechtigungen
ext4	Statische Inode-Tabelle (256 Bytes)	Nativ via `acl` Mount-Option	Vollständig unterstützt	Standard in Debian/Ubuntu, sehr robust
XFS	Dynamische Inodes	Nativ standardmäßig aktiv	Vollständig unterstützt	Standard in RHEL/Fedora, exzellent für große Dateien
BTRFS	Subvolume-Inodes	Nativ unterstützt	Vollständig unterstützt	Berechtigungen gelten innerhalb von Subvolumes
ZFS	Dynamische Object-IDs	NFSv4 & POSIX ACLs	Teilweise (ZFS Properties)	Unterstützt granulare NFSv4-ACLs

Dateisystem-Mount-Optionen für maximale Sicherheit

In `/etc/fstab` können Dateisysteme mit sicherheitsrelevanten Mount-Flags gehärtet werden:

□□ Praktisches Beispiel:

Sichere Mount-Optionen in `/etc/fstab` hinterlegen:

```
# /etc/fstab Beispiel für gehärtete Partitionen:
# 1. Temporäres Verzeichnis ohne Ausführungsrechte und SUID-Bits:
tmpfs   /tmp         tmpfs   defaults,noexec,nosuid,nodev   0  0

# 2. Shared Memory absichern:
tmpfs   /dev/shm     tmpfs   defaults,noexec,nosuid,nodev   0  0

# 3. Benutzer-Home-Partition ohne SUID-Eskalation:
/dev/sdb1  /home     ext4    defaults,nosuid,nodev          0  2
```

- `noexec`: Verhindert die direkte Ausführung von Binärdateien auf dieser Partition (effektiver Schutz vor heruntergeladener Malware in `/tmp`).
- `nosuid`: Ignoriert SUID- und SGID-Bits auf allen Dateien dieser Partition (verhindert unbefugte Root-Eskalation).
- `nodev`: Verhindert die Interpretation von Block- und Character-Devices.

Befehlsreferenz (Cheatsheet)

Zur schnellen Orientierung im Terminal-Alltag fasst diese Tabelle alle Kombinationen, Modi und Anwendungsfälle übersichtlich zusammen:

Modus	Oktal	Symbolisch	Bedeutung & Empfohlener Einsatzbereich
600	`0600`	`-rw-------`	Private SSH-Keys (`id_ed25519`), Zertifikats-Schlüssel (`.key`), `.env` -Dateien
640	`0640`	`-rw-r-----`	System-Konfigurationsdateien mit Gruppenleserecht (z. B. `/etc/shadow`)
644	`0644`	`-rw-r--r--`	Standard für öffentliche Dokumente, HTML-, CSS-, JS- und Bilddateien
660	`0660`	`-rw-rw----`	Gemeinsame Arbeitsdateien für Team-Gruppen oder Service-Sockets
700	`0700`	`drwx------`	Persönliche Home-Verzeichnisse, `~/.ssh/` , PostgreSQL Data Directory

Modus	Oktal	Symbolisch	Bedeutung & Empfohlener Einsatzbereich
750	`0750`	`drwxr-x---`	System-Verzeichnisse mit Gruppenleserecht (z. B. Webserver-Logverzeichnisse)
755	`0755`	`drwxr-xr-x`	Standard für Verzeichnisse, Binärprogramme (`/usr/bin/`) und Shellskripte
770	`0770`	`drwxrwx---`	Team-Verzeichnisse mit vollen Schreib- und Leserechten für Gruppenmitglieder
1777	`1777`	`drwxrwxrwt`	Globale temporäre Verzeichnisse mit Sticky Bit (`/tmp` , `/var/tmp`)
2770	`2770`	`drwxrws---`	Geteilte Verzeichnisse mit SGID-Vererbung für Gruppenarbeiten
4755	`4755`	`-rwsr-xr-x`	SUID-Binaries mit Root-Eskalation (`/usr/bin/passwd` , `/usr/bin/sudo`)

Weiterführende Ressourcen

Ressource	Beschreibung	Typ
POSIX.1-2024 Spezifikation für chmod	Die verbindliche Spezifikation der Open Group und IEEE für das Dienstprogramm chmod (Ausgabe 2024)	Offizielle Spezifikation
GNU Coreutils: File permissions	Ausführliches GNU-Handbuch zu Berechtigungsmodi, Oktalzahlen und umask	Offizielle Dokumentation
Arch Linux Wiki: File permissions and attributes	Praktische Referenz für Linux-Dateirechte, SUID-Bits und erweiterte ACLs	Community-Wiki
Debian Wiki: Permissions	Grundlagen und praxiserprobte Best Practices zur Unix- und Linux-Rechteverwaltung	Handbuch & Referenz
Linux Audit Framework (auditd)	Umfassendes Handbuch zur Überwachung und Protokollierung von System-Calls	Technische Dokumentation
Docker Documentation: Manage data in Docker	Offizielle Anleitung zur Volume-Verwaltung und Benutzer-Isolierung in Containern	Offizielle Dokumentation

Fazit

Der Befehl `chmod` und das dahinterstehende POSIX-Berechtigungsmodell sind ein zeitloses Meisterwerk der Unix-Architektur: Mit nur 12 Bits im Inode lässt sich ein robustes, ressourcenschonendes und extrem schnelles Sicherheitsfundament für Multi-User-, Server-, Container- und Cloud-Infrastrukturen realisieren.

Wer die Funktionsunterschiede von `r`, `w` und `x` zwischen Dateien und Verzeichnissen verinnerlicht hat, mit dem bedingten Ausführungs-Bit `X` arbeitet, Spezial-Bits wie SGID und Sticky Bit gezielt einsetzt und Berechtigungen über Ansible oder Shell-Skripte automatisiert, meistert jede Berechtigungsherausforderung souverän – ohne jemals auf unsichere `chmod 777`-Notlösungen zurückgreifen zu müssen.

> **Praxis-Tipp:** Nutze im Administrations-Alltag stets `chmod -R u=rwX,go=rX /zielpfad` anstelle von groben Oktalzahlen. Das garantiert, dass alle Unterverzeichnisse sauber betretbar bleiben (`x`), während reguläre Text- und Bilddateien nicht versehentlich als ausführbare Binaries markiert werden.