LINUX SECURITY

part_1.sh

Author: aimery de La Roncière @ aimerydlr
University: ECE Paris – Ing5 CYB
YEAR: 2025-2026

Planning Ing5 CYB Gr01

11/09/2025 (2h):

- Introduction to cybersecurity and Linux basic reminders
- Linux Privilege and Access Management

12/09/2025 (2h):

- Linux Privilege and Access Management
- Cryptography and Communication Security

09/10/2025 (4h):

- Cryptography and Communication Security
- Securing Services and Applications

06/11/2025 (2h):

- Audit and Monitoring

07/11/2025 (2h):

- Hands-on labs

21/11/2025 (4h):

- Vulnerability assessment and Patch Management
- Hands-on labs presentation

Table of contents

I. Introduction to cybersecurity and Linux basic reminders

I.1. Very… very brief introduction to cybersecurity

Computer security refers to the set of technical, organizational, and human means aimed at protecting information systems against threats that could compromise their proper functioning, the integrity of data, or the confidentiality of information.

CIA triad

Figure 1 - CIA triad picture

Source : https://www.nist.gov/image/cia-triad

Documentation: https://www.nist.gov/blogs/manufacturing-innovation-blog/cybersecurity-critical-component-industry-40-implementation

Strong security can harm availability if it frustrates users or blocks access. A good security program balances protection with usability, involves the whole organization, and considers technical debt.

Defense in Depth

Defense in Depth is a cybersecurity strategy that uses multiple layers of security controls across an information system. The goal is to provide redundancy and slow down or prevent attackers from breaching the system. If one layer is bypassed, others still provide protection. These layers typically include physical security, network security, endpoint protection, application security, identity and access management, and user awareness. By combining technical, administrative, and physical controls, Defense in Depth reduces the risk of a single point of failure.
Documentation: https://cyber.gouv.fr/publications/la-defense-en-profondeur-appliquee-aux-systemes-dinformation

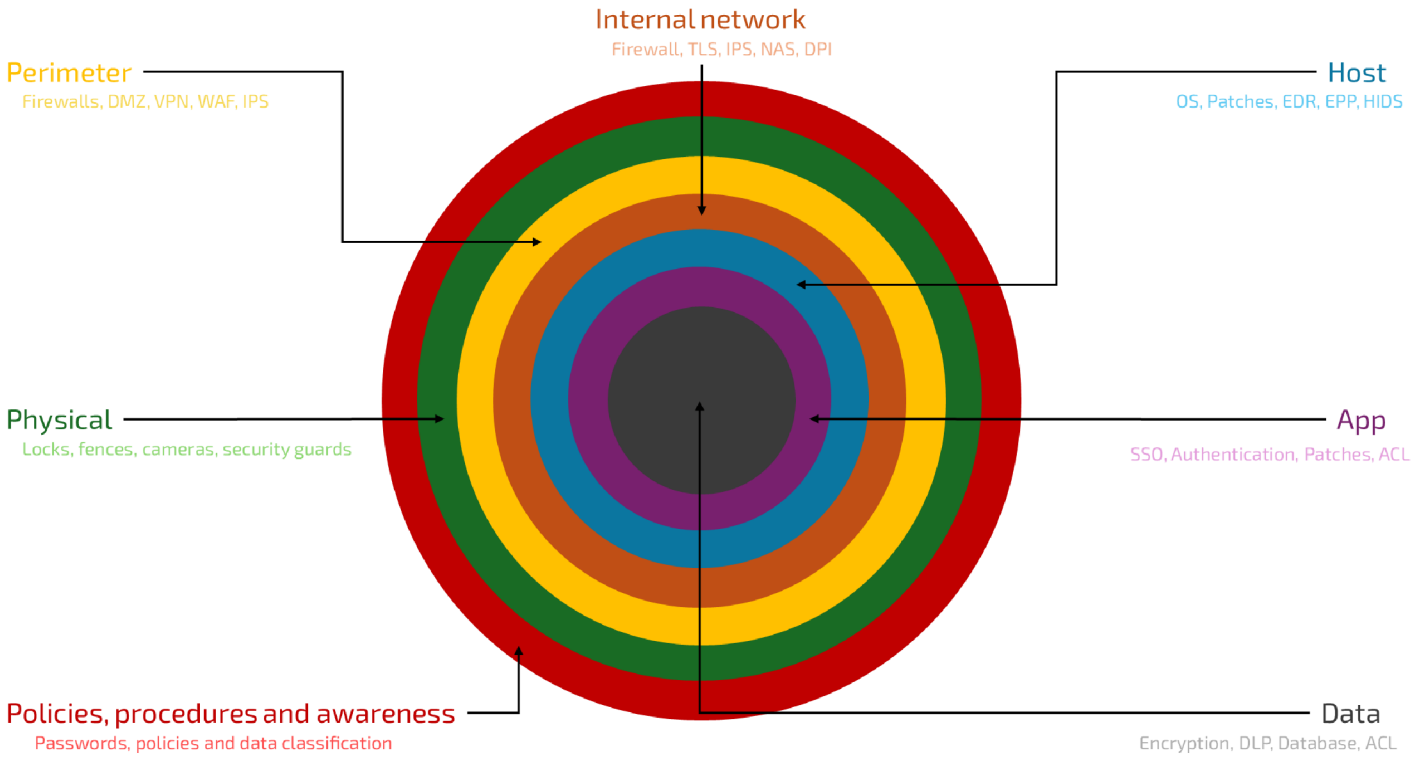

Figure 2 - Defense in depth

I.2. Basic Linux Concepts Refresher

Who Are You? Understanding UID, GID and root

In Linux, each user and group are identified by a numeric ID rather than just a name. These identifiers are used internally by the kernel to manage access rights and ownership.

- The UID (User ID) uniquely identifies each user.
- The GID (Group ID) uniquely identifies each group.
- The root user, the superuser, always has UID 0, which grants full administrative privileges over the system.

By default, regular user accounts (created by system administrators or during installation) are assigned UIDs starting from 1000. This helps distinguish them from system or service accounts, which typically have UIDs below 1000 (e.g., daemon, www-data, nobody).

This separation is important for security: it allows the system to assign limited privileges to background services and ensures that only human users are granted interactive access.

```
id                                  # Display current user UID, GID, and group memberships

whoami                              # Display the current username

getent passwd root                  # View UID/GID for another user (e.g., 'root')

cat /etc/passwd | cut -d: -f1,3,4   # View all users and their UIDs/GIDs
```


Getting Around: The Linux Filesystem Hierarchy (FHS)

Linux systems follow a standardized directory layout known as the Filesystem Hierarchy Standard (FHS). This structure organizes files and directories based on their roles, making system navigation predictable and consistent across distributions. For example, configuration files are found in /etc, user data in /home, and system binaries in /bin or /usr/bin. Familiarity with the FHS is crucial for system administration and security management.
Documentation: https://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/Linux-Filesystem-Hierarchy.html

Key directories

- /bin : Essential user commands (e.g., cp, mv, rm). Required for system startup. Symlink to /usr/bin.
- /boot : Bootloader files (e.g., GRUB) and Linux kernels. A separate partition is recommended.
- /dev : Device files for physical hardware (e.g., /dev/sda1) and virtual devices (/dev/null, /dev/random).
- /etc : System configuration files and settings for installed software.
- /home : User home directories, e.g., /home/username. A separate partition is recommended.
- /lib : Essential shared libraries needed by binaries in /bin and /sbin.
- /mnt : Temporary mount point for internal filesystems. Typically used for manual mounts.
- /media : Mount points for removable media such as USB drives or CDs.
- /opt : Optional or third-party application packages.
- /proc : Virtual filesystem providing information about the kernel and processes (e.g., /proc/cpuinfo).
- /root : Home directory of the root user (empty by default on Ubuntu).
- /sbin : Essential system binaries for administrators (e.g., fsck, shutdown, reboot). Symlink to /usr/sbin.
- /srv : Data for services provided by the system (e.g., FTP, HTTP, databases).
- /tmp : Temporary files. Often cleared at reboot; a separate partition is recommended.
- /usr : Unix System Resources used for user utilities, applications, libraries and documentation.
- /var : Variable data such as logs, spools, mail, caches, and databases.

```
ls -la / # List the top-level directories in the root filesystem
tree / -pug # Display a tree view of the root filesystem including permissions, owner, and group
```


The Art of Self-Help: Understanding man, --help, and RTFM

man – The Manual Pages

The man command provides detailed documentation on commands, system calls, configuration files, and more.
Documentation: https://tldp.org/manpages/man.html

Manuals are divided into sections:

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in /dev)
5. File formats and conventions, e.g. /etc/passwd
6. Games
7. Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7), man-pages(7)
8. System administration commands (usually only for root)
9. Kernel routines [Non standard]

```
man man          # Open the manual page for the 'man' command
man 5 passwd     # Show the manual for the passwd file format (section 5)
man -k ssh       # Search for man pages related to 'ssh'
```

--help – Quick Command Syntax

Most CLI tools offer a quick overview of options using the --help or -h flag. This is useful for discovering command-line options without diving into full man pages.

```
grep --help
```


RTFM

In the Linux world, RTFM stands for "Read The F*ing Manual". Far from being just slang, it reflects a real culture of self-sufficiency and documentation-first troubleshooting that is key to working effectively in Unix-like systems.

Before asking someone for help, the expected first step is to consult:

- The manual pages (man)
- The program's built-in help (--help)
- Relevant configuration files and examples
- Internet is your friend

Understanding and using these resources is a foundational skill for system administration, development, and cybersecurity on Linux.

Directing the Flow: Redirection and Pipes

In Linux and other Unix-like systems, every running process interacts with three standard data streams. The *>&* symbol is used to redirect one file descriptor to another.
Documentation: https://tldp.org/LDP/abs/html/io-redirection.html

- Standard Input (stdin) – file descriptor *&0*: This is where a process reads its input, typically from the keyboard or a file.
- Standard Output (stdout) – file descriptor *&1*: This is where a process sends its regular output, usually to the terminal.
- Standard Error (stderr) – file descriptor *&2*: This is where a process writes error messages, keeping them separate from normal output.

These streams can be redirected or manipulated using shell operators to control the flow of data, making shell scripting and automation highly flexible.

Redirection Operators

- *0< out.txt*: standard input comes from out.txt.
- *>, 1>:* sends standard output to a file, overwriting it
- *>>:* appends standard output to the file instead of overwriting
- Input redirection < sends the contents of a file as input to a command
- Error redirection (*2>*, *2>>*) redirects error messages (stderr) to a file

Pipe |

Pipes connect the stdout of one command to the stdin of another, forming command chains.

Tee

The *tee* command is useful for duplicating output:

- It reads from stdin,
- Writes to both stdout and a file.

Examples:

```
echo "Hello" > file.txt          # Overwrite file.txt with "Hello"
echo "World" >> file.txt         # Append "World" to file.txt
cat < file.txt                   # Read and display contents of file.txt
cat file.txt | grep Hello        # Filter lines containing "Hello"
ls | tee listing.txt             # Show output and save it to listing.txt
command > out.txt 2>&1           # Redirects standard error (2) to standard output (1)
cut -d , -f 1 file.csv > students.txt 2> errors.log #Extracts first column of file.csv to students.txt, logs errors to errors.log
```


Where Is That Command? Using which

When multiple versions of a program or command exist on a system, it's important to know exactly which executable will run. The which command shows the full path to the binary that the shell executes when you enter a command. This helps verify the command location and troubleshoot environment path issues or version conflicts.

```
which python3
which ls
```


Managing Services: systemd

Most modern Linux distributions use systemd as their init system — that is, the system and service manager responsible for initializing the system during boot and managing background processes known as services or daemons.

Unlike older init systems (like SysVinit), systemd offers parallelized service startup, socket and D-Bus activation, service dependency management, and unified logging, making it both powerful and complex.

The primary tool for interacting with systemd is the systemctl command. It allows system administrators to:

- Check the status of services,
- Start, stop, restart, or reload services on demand,
- Enable or disable services so they start or don't start automatically at boot time,
- Investigate service logs using the journalctl command.

```
systemctl status ssh          # Check the status of SSH service
systemctl start nginx         # Start the Nginx web server
systemctl stop nginx          # Stop the Nginx web server
systemctl enable fail2ban     # Enable fail2ban to start on boot
```


II. Linux Privilege and Access Management

In Linux, controlling who can do what is a critical component of system security. Understanding how user identity, permissions, and privilege elevation work is essential to secure any system. This section covers the core mechanisms available to manage access and privilege on a Linux system.

II.1. User Identification and Authentication

In any secure system, the first step is knowing who is trying to access it (identification) and verifying that identity (authentication). Linux handles this through user and group management, password verification, and a powerful, modular framework called PAM (Pluggable Authentication Modules).

Each user and group on a Linux system is associated with a unique numerical ID:

- UID (User ID) identifies a user.
- GID (Group ID) identifies a group.

The most privileged user, root, always has UID 0, and has full control over the system. Regular users typically have UIDs starting from 1000, while system and service accounts (e.g., daemon, www-data) are assigned UIDs below 1000.

User and group information

- /etc/passwd — contains account details

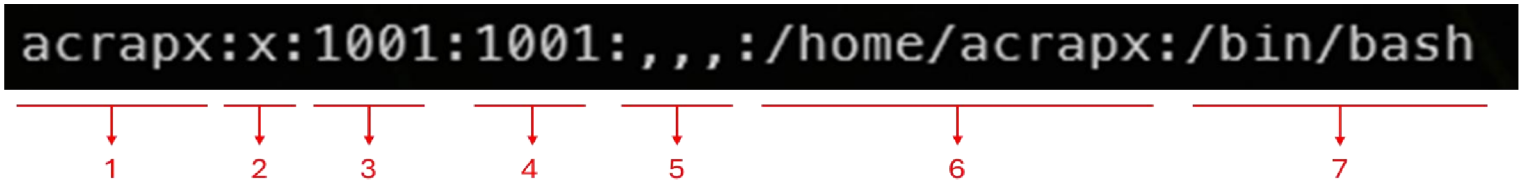

Figure 3 - /etc/passwd description

- 1 : username : The login name used to identify the user.
- 2 : x : Placeholder indicating the encrypted password is stored in /etc/shadow. (Historically, the hashed password was here.)
- 3 : UID : User ID: A unique numeric identifier for the user. UID 0 is reserved for root; UIDs ≥1000 are for regular users.
- 4 : GID : Group ID: Primary group's numeric ID, defined in /etc/group.
- 5 : comment (GECOS) : Optional field typically used to store the user's full name or other information (e.g., phone number).
- 6 : home_directory : The user's personal directory, usually /home/username.
- 7 : shell : The login shell program (e.g., /bin/bash, /usr/bin/zsh, or /usr/sbin/nologin).

```
ntpsec:x:122:126::/nonexistent:/usr/sbin/nologin
```

Figure 4 - user without home directory or shell

- /etc/group — defines group names and associated GIDs

```
kaboxer:x:137:kali,acrapx
xrdp:x:138:
acrapx:x:1001:
docker:x:139:
beef-xss:x:140:
debian-tor:x:141:
_bloodhound:x:142:
```

Figure 5 - /etc/group description

- /etc/shadow — stores hashed passwords and password policies (only readable by root)

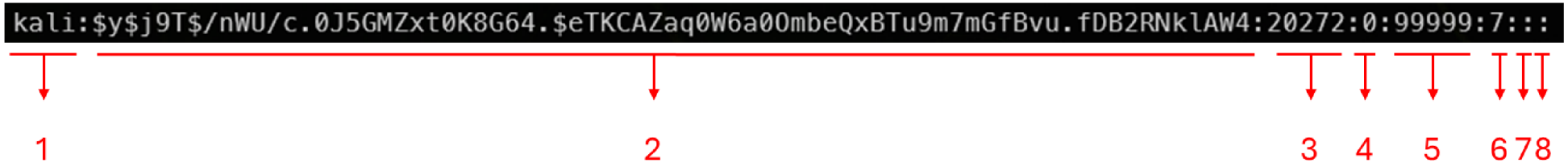

Figure 6 - /etc/shadow description

- 1 : username : The login name of the user (must match an entry in /etc/passwd).
- 2 : password : The hashed password. May also contain special values like:
  - *!* or *: lock password to prevent access
  - *Empty* for no password (insecure)

```
xrdp:!:20057:::::
```

Figure 7 - locked password

- 3 : last_change : Number of days since Jan 1, 1970 (Epoch) when the password was last changed.

- 4 : min_age : Minimum number of days required between password changes. Prevents users from changing passwords too frequently.
- 5 : max_age : Maximum number of days the password is valid. After this period, the password must be changed.
- 6 : warn : Number of days before expiration the user is warned to change their password.
- 7 : inactive : Number of days after password expiration during which the account remains usable. After this period, the account is disabled.
- 8 : expire : The absolute expiration date of the account, in days since Jan 1, 1970. After this date, the account is disabled, regardless of the password. Empty means no expiration.

Shadow-utils

The shadow-utils package (or simply shadow, depending on the distribution) is a core component of Linux and Unix systems. It provides the tools for managing user accounts and passwords, while enhancing security through the use of the /etc/shadow file.

Its main purpose is to enable the secure management of user credentials by separating sensitive information—such as password hashes—into a protected file (/etc/shadow) that is inaccessible to unprivileged users.

Low-level tools like useradd, userdel, or groupadd are non-interactive and designed to be script-friendly. Unlike high-level commands such as adduser or deluser that prompt the user and guide them interactively, low-level utilities are ideal for automation and batch operations in shell scripts or system provisioning tasks.

/etc/login.defs – Login Configuration File

The /etc/login.defs file is a system-wide configuration file that defines default settings used by essential user account management utilities such as useradd, usermod, passwd, and login.

It plays a key role in account creation, password policy enforcement, and login behavior by specifying default parameters that affect how new users are created and how authentication behaves.

Key Parameters in /etc/login.defs

Here are some important directives commonly found in the file:

- PASS_MAX_DAYS : Maximum number of days a password is valid before it must be changed.
- PASS_MIN_DAYS : Minimum number of days required between password changes.
- PASS_WARN_AGE : Number of days before password expiration to warn the user.
- UID_MIN / UID_MAX : Range of UIDs assigned to regular (non-system) users.
- GID_MIN / GID_MAX : Same as above, but for group IDs.
- CREATE_HOME : Whether to create a home directory when a new user is added (yes or no).
- ENCRYPT_METHOD : Hashing algorithm used for password encryption (e.g., SHA512).
- UMASK : Default file permission mask applied when new files are created.
- LOGIN_RETRIES : Number of allowed failed login attempts before action is taken.
- LOGIN_TIMEOUT : Time (in seconds) before a login attempt times out.

adduser

Creates a new user interactively. It is a high-level frontend used mainly on Debian-based systems. It wraps around the lower-level *useradd* and automates:

- Home directory creation
- Password setup
  User information prompts

```
sudo adduser Alice                              # Create user 'Alice' with default options
sudo adduser --shell /bin/zsh Bob               # Create user 'Bob' with zsh as default shell
sudo adduser --uid 1050 --disabled-password eve    # Create 'eve' with UID 1050 and no password
```

usermod

Modifies an existing user account. It can:

- Change the user's shell, home directory, UID
  Add the user to supplementary groups

```
sudo usermod -aG sudo Alice                # Add 'Alice' to the 'sudo' group (append, not overwrite)
sudo usermod -s /bin/bash eve              # Change login shell of 'eve' to bash
sudo usermod -d /srv/users/Alice Alice     # Move 'Alice's home directory to /srv/users/Alice
```

deluser

Deletes a user from the system and optionally removes their home directory and mail spool. It's a Debian-specific wrapper around *userdel*.

```
sudo deluser Alice                         # Delete user 'Alice' but keep home dir
sudo deluser --remove-home Bob             # Delete 'Bob' and remove home dir and mail spool
sudo deluser --backup --remove-home carol  # Backup 'carol's home dir then remove it
```


addgroup

Creates a new group on the system. This is a Debian-specific wrapper around *groupadd*.

```
sudo addgroup devs                          # Create a group named 'devs'
sudo addgroup --gid 2022 projectX           # Create group 'projectX' with GID 2022
sudo addgroup --system sysmon               # Create a system group 'sysmon'
```

groupmod

Modifies existing group properties (e.g., rename, change GID).

```
sudo groupmod -n engineers devs             # Rename group 'devs' to 'engineers'
sudo groupmod -g 3030 devs                  # Change GID of 'devs' to 3030
sudo groupmod -p $(openssl passwd ...) devs # Set password for 'devs' group (legacy use)
```

delgroup

Removes a group from the system. Debian-specific wrapper around *groupdel*.

```
sudo delgroup devs                          # Delete group 'devs'
sudo delgroup --only-if-empty devs          # Delete only if no users belong to the group
sudo delgroup --quiet devs                  # Suppress output when deleting group
```

passwd

Changes a user's password or account status (e.g., lock, unlock). It modifies both */etc/passwd* and */etc/shadow* entries.

```
passwd                                      # Change password for current user
sudo passwd Alice                           # Set password for 'Alice'
sudo passwd -l Alice                        # Lock 'Alice's account (disable password login)
```


chage

Manages password aging policies (max/min age, expiration warning, etc.). It modifies the *shadow* file's aging fields.

```
sudo chage -l Alice                    # List password aging info for 'Alice'
sudo chage -M 90 Alice                 # Force password change every 90 days
sudo chage -E 2025-12-31 Alice         # Expire 'Alice's account on December 31, 2025
```

gpasswd

Administers group membership and sets group passwords (rarely used today).

Often used to:

- Add/remove users from groups
- Designate group administrators

```
sudo gpasswd -a Alice devs             # Add 'Alice' to 'devs' group
sudo gpasswd -d Alice devs             # Remove 'Alice' from 'devs' group
sudo gpasswd -A Bob devs               # Make 'Bob' the administrator of group 'devs'
```


PAM (Pluggable Authentication Modules)

PAM is a flexible and modular framework used by Linux and Unix systems to handle authentication tasks. It provides a centralized and consistent way to *authenticate users, manage sessions, enforce password policies, and perform account managemen*t across various applications and services.

PAM works by stacking multiple independent modules, each responsible for a specific authentication function, such as verifying passwords, checking account validity, or logging authentication attempts. This modularity allows system administrators to configure authentication policies dynamically without modifying the application code.

PAM is a *modular authentication* framework. Each PAM-aware application delegates authentication tasks (like verifying passwords or locking inactive accounts) to PAM, which then processes a stack of modules defined in configuration files.

PAM Configuration Files

PAM's behavior is defined in configuration files located in */etc/pam.d/* or directly in */etc/pam.conf*. Each file in /etc/pam.d/ defines how PAM handles that specific service.

```
/etc/pam.d
chfn        common-account    common-session                lightdm             login      passwd    runuser-l  su      sudo-i
chpasswd    common-auth       common-session-noninteractive lightdm-autologin   newusers   ppp       samba      su-l    xrdp-sesman
chsh        common-password   cron                          lightdm-greeter     other      runuser   sshd       sudo
```

Figure 8 - List of common PAM configuration files at /etc/pam.d/

On Debian-based systems, */etc/pam.d/common-** files are included in other service definitions via the *@include* directive. This allows centralized control over authentication behavior.

```
/etc/pam.d
common-account
common-auth
common-password
common-session
common-session-noninteractive
```

Figure 9 - common-* PAM files

PAM Rule Syntax

Each line in a PAM config file has this structure:

```
<module_type>  <control_flag>  <module_path>  [arguments]
```

Module Types

Each PAM rule corresponds to one of four module types, each responsible for a different aspect of access control:

- *auth* : Verifies user identity (e.g., password check, biometric scan)
- *account* : Verifies account validity (e.g., expiration date, group membership)
- *password* : Handles password changes and complexity rules
- *session* : Sets up and tears down user sessions (e.g., mounts, logging, limits)

Control Flags

- *required* : Module must succeed. If it fails, the result is failed — but PAM continues processing.
- *requisite* : Like required, but failure immediately aborts the process.
- *sufficient* : If this module succeeds and no required module has failed, authentication succeeds immediately.
- *optional* : Result is ignored unless it's the only module of that type.

Module path

The module path defines which PAM module (shared library) will be used to process the authentication step. These modules are typically stored in:

- */lib/x86_64-linux-gnu/security/* for 64 bits
- */lib/i386-linux-gnu/security/* for 32 bits

Examples of common modules:

- pam_unix.so : Traditional UNIX authentication (uses /etc/shadow, /etc/passwd)
- pam_sss.so : Integration with SSSD (used for LDAP, Kerberos, etc.)
- pam_rootok.so : Allows authentication if UID is 0 (used in sudo, su)
- pam_env.so : Sets environment variables for sessions
- pam_limits.so : Enforces limits from /etc/security/limits.conf
- pam_faillock.so : Tracks failed login attempts and enforces account lockouts
- pam_listfile.so : Allows or denies access based on file content (users/groups/hosts)
- pam_tally2.so : Older module for tracking login failures
- pam_motd.so : Displays system messages at login
- pam_exec.so : Runs arbitrary scripts or binaries as part of the PAM stack

Each module is a .so (shared object) file loaded at runtime.

Module Arguments ([arguments])

Modules can accept optional arguments to modify their behavior. These key-value pairs or flags allow fine-tuning of authentication behavior.

Examples of common arguments:

- nullok : Accept empty passwords (use with caution)
- try_first_pass : Use previously entered password (from another module), don't prompt again
- use_first_pass : Same as above, but fail if no previous password exists
- debug : Enables verbose logging to help with troubleshooting
- deny=N : (For pam_faillock.so) Number of failed attempts before lockout
- unlock_time=N : (For pam_faillock.so) Time (in seconds) until account is automatically unlocked
- file=/path : (For pam_listfile.so) Path to list of allowed/denied users/groups

Important: Arguments are module-specific — always refer to the module's man page for valid options.

Example: su PAM module

su PAM module

```
cat /etc/pam.d/su

auth       sufficient pam_rootok.so

session    required   pam_env.so readenv=1
session    required   pam_env.so readenv=1 envfile=/etc/default/locale
session    optional   pam_mail.so nopen
session    required   pam_limits.so

@include common-auth
@include common-account
```

Explanation

- *auth sufficient pam_rootok.so* : If the current user is root (UID 0), authentication is immediately accepted without prompting for a password.
- *session required pam_env.so readenv=1* : Reads /etc/environment and sets up environment variables for the session.
- *session required pam_env.so readenv=1 envfile=/etc/default/locale* : Adds locale-specific variables from /etc/default/locale.
- *session optional pam_mail.so nopen* : Sets the MAIL environment variable without notifying about new mail when switching users.
- *session required pam_limits.so* : Applies resource limits (e.g., max number of processes, open files) based on /etc/security/limits.conf.
- *@include common-auth* : Includes system-wide authentication rules (e.g., password checks via pam_unix.so, optional 2FA, etc.).
- *@include common-account* : Includes checks for account validity, expiration, etc.
- *@include common-session* : Includes standard session setup (e.g., pam_systemd.so, pam_umask.so, etc.).

common-auth PAM module

```
cat /etc/pam.d/common-auth

auth       sufficient pam_rootok.so

auth    [success=2 default=ignore]      pam_unix.so nullok
auth    [success=1 default=ignore]      pam_winbind.so krb5_auth krb5_ccache_type=FILE cached_login try_first_pass
auth    requisite                       pam_deny.so
auth    required                        pam_permit.so@include common-session
```

Explanation

- auth [success=2 default=ignore] pam_unix.so nullok : Traditional Unix authentication using /etc/shadow. The nullok option allows authentication even if the password is empty. If successful, skip the next two lines.
- auth [success=1 default=ignore] pam_winbind.so krb5_auth krb5_ccache_type=FILE cached_login try_first_pass : Winbind authentication (e.g., Kerberos/Active Directory). If successful, skip the next line. Attempts to use the previously entered password with try_first_pass.
- auth requisite pam_deny.so : Immediately denies authentication if reached (means no previous module succeeded).
- auth required pam_permit.so : Always returns success to ensure a positive return code if no previous module has done so.

Modify su PAM module to enhance security

- Enable group-based control as follows (to be added):

```
vim /etc/pam.d/su

auth       required pam_wheel.so group=wheel
```

This will prevent any user who is not a member of the “wheel” group from using su.

II.2. Managing Privileges: sudo, su, pkexec and Special Permission Bits

In any robust and secure computing environment, controlling who can execute what commands is paramount. Linux, at its core, is designed with a strong emphasis on privilege separation, ensuring that users operate with the minimum necessary permissions for their tasks. This fundamental principale safeguards system integrity by preventing unauthorized access and unintended modifications.

This chapter delves into the critical mechanisms that allow for controlled privilege escalation—the process of temporarily gaining higher access rights to perform administrative functions. Understanding these mechanisms is critical for system hardening and audit purposes.

su: Substitute User

The su (substitute user) command allows a user to assume the identity of another user account, most often root. Using su requires knowing the target user's password. It does not enforce fine-grained permission control or logging by default.

```
su                          # If no username is specified, it defaults to root
su - Alice                  # Switch to user Alice and load Alice's environment as if Alice had logged in directly
su -c "mkdir ~/test" Alice  # Execute the command 'mkdir ~/test' as user Alice
```

sudo: Superuser Do

sudo is the preferred method for privilege escalation. It allows a user to execute commands with another user's privileges (typically root), while maintaining granular access control and auditability.

```
sudo apt update
sudo systemctl restart nginx  # Restarts the Nginx service as a superuser
sudo -u www-data bash         # Opens a bash shell with the www-data user's privileges
```


Managing sudo Access: visudo, sudoers & sudoers.d

To control who can execute privileged commands via sudo, Linux systems rely on a powerful access control configuration: the sudoers policy. This policy determines which users or groups can execute which commands, on which machines, and with or without a password.

/etc/sudoers: The Main Configuration File

The /etc/sudoers file is the central authority that governs all sudo permissions on a system. It contains rules defining:

- Who can run commands as which user (typically root)
- Which commands are allowed
- Whether a password is required
- On which hosts (in multi-host environments)

In the sudoers file, the general syntax is:

```
user  host = (target_user : target_group) command
```

- *host*: The name of the machine on which the rule applies. By default, it is often set to ALL, meaning all hosts.
- *(target_user : target_group)*: Specifies as which user and group the command can be executed. target_group is optional and defines the effective group.
- *command*: The command(s) the user is allowed to run. This can include an absolute path, specific arguments, or the keyword ALL.

/etc/sudoers configuration example:

```
# root can execute any command on any host as any user or group, with password.
root    ALL=(ALL:ALL) ALL

# Alice can restart Apache without a password on any host, as any user.
Alice   ALL=(ALL:ALL) NOPASSWD: /usr/bin/systemctl restart apache2

# Bob can run apt update and upgrade as root, but must authenticate (password required).
Bob     ALL=(root) /usr/bin/apt update, /usr/bin/apt upgrade
```

visudo: Safely Editing sudoers

Directly editing /etc/sudoers with a regular text editor is dangerous, syntax errors can lock you out of root access. Instead, you must use visudo.

This command:

- Locks the file to avoid concurrent edits
- Validates syntax before saving
- Prevents misconfiguration

visudo uses your system's default editor (usually vi or nano). You can change it by setting the EDITOR environment variable.

```
sudo visudo                                 # Edit the main sudoers file safely with syntax checking
sudo visudo -f /etc/sudoers.d/custom_rules  # Edit a specific sudoers include file safely
sudo EDITOR=nano visudo                     # Edit the sudoers file using nano as the editor instead of the default vi
```


/etc/sudoers.d/: Modular Configuration

To avoid cluttering the main sudoers file, it's best practice to use the /etc/sudoers.d/ directory for modular and maintainable rules.

Each file in this directory can contain sudo rules for a specific user, group, or role.

```
# Create a sudoers file allowing user 'devops' to run all commands without a password
echo "devops ALL=(ALL) NOPASSWD:ALL" | sudo tee /etc/sudoers.d/devops
# Set strict permissions to secure the sudoers file
sudo chmod 440 /etc/sudoers.d/devops
```

- Advantages:
  - Keeps configurations organized
  - Supports automated provisioning
  - Changes can be tracked per user/team
- Important:
  - Files must have 0440 permissions
  - Must be owned by root
  - Always validate rules using visudo -cf /etc/sudoers.d/filename

pkexec: PolicyKit Execution

pkexec is part of Polkit (PolicyKit) and provides an alternative to sudo, often used in graphical or desktop environments. Permissions are managed via .policy files in */usr/share/polkit-1/actions/* and runtime policies under */etc/polkit-1/*.

While useful in GUI contexts, pkexec has been historically vulnerable (e.g., CVE-2021-4034 "PwnKit"), so it must be tightly controlled or disabled.

```
pkexec systemctl restart apache2      # Restart Apache service with root privileges via PolicyKit
pkexec bash                           # Open a root shell with administrative privileges
pkexec --user Alice gedit /opt/file   # Run the text editor gedit as user 'Alice' to edit /opt/file
```


Special Permission Bits

Linux also supports special permission bits that extend the basic DAC model by modifying how files and directories behave. This topic will be covered in II.3 Discretionary Access Control (DAC).

Managing privileges best practices summary

- Use visudo to safely edit any sudo-related rule.
- Never edit /etc/sudoers directly without syntax checks.
- Split rules into files under /etc/sudoers.d/ for better structure.
- Restrict commands allowed by sudo whenever possible.
- Always use full pathnames in allowed commands for security.
- Use pkexec to execute commands as another user with PolicyKit authorization.
- Configure PolicyKit policies carefully to avoid privilege escalation.
- Avoid using pkexec in scripts without proper checks and logging.
- Special Permission Bits
- Understand and use setuid, setgid, and the sticky bit for controlled privilege escalation on executables and directories.
- Limit the use of setuid/setgid binaries to trusted programs only.
- Regularly audit binaries with special permission bits to detect potential security risks.

root privilege escalation comparison

Command	Main Purpose	Loaded Environment	Password Required	Logged via sudoers	Comment
sudo -s	Start a root shell (non-login shell)	root minimal environment filtred by sudo. (Limited PATH, usual variables kept)	User's password	Yes	A quick root shell that doesn't load root's full login profile. It uses sudo's default security-conscious environment filtering.
sudo -Es	Like -s, but preserves user's environment (-E + -s)	Preserved invoking user's environment (bypasses sudo's default cleaning)	User's password	Yes	Combines the immediate shell access of -s with the environment preservation of -E. Use with extreme caution as it carries the same security risks as -E but within an interactive shell.
sudo su	Invoke su as root via sudo (non-login shell)	User's environment	User's password	Yes	Less clean than sudo -i, but commonly used.
sudo -i	Simulate a full root login shell	Root's login environment	User's password	Yes	Recommended: loads root's profile and ensures clean login context.
su	Switch to another user (default: root) (non-login shell)	User's environment	Root's password	No	Not traceable via sudo, discouraged in secured environments.
su - / su -l	Switch to root and simulate a full login session	Full root login environment	Root's password	No	Functionally equivalent to sudo -i but lacks sudo policy control.
su root	Same as su, explicitly switching to root (non-login shell)	User's environment	Root's password	No	Identical to su; adds clarity but no functional difference.

II.3. Discretionary Access Control (DAC)

Discretionary Access Control (DAC) is a fundamental security model used in Linux and many other operating systems to regulate access to files and resources. Under DAC, the owner of a resource (usually a file or directory) has the authority to decide who else can access that resource and what level of access they have.

Key Concepts of DAC

- *Ownership*: Every file and directory on a Linux system is owned by a user and a group. The owner controls permissions for that resource.
- *Permissions*: Access is governed by permission bits that specify what actions the owner, group members, and others can perform.
- *Flexibility*: Owners can grant or restrict access at their discretion, hence the name.

Linux File Permissions

Linux implements DAC primarily through a permission system based on three sets of permissions:

- *Read* (r) : Allows reading the file or listing the directory contents.
- *Write* (w) : Allows modifying the file or adding/removing files in a directory.
- *Execute* (x) : Allows running a file as a program or accessing a directory.

Permissions are assigned to three categories of users:

- *Owner (User)*: The user who owns the file.
- *Group*: Users who belong to the file's group.
- *Others*: All other users.

Permission Representation

Permissions are displayed using symbolic or octal notation:

Symbolic notation: e.g. -rwxr-xr--

- The first character indicates the file type (- for regular files, d for directories).
- The next three characters represent owner permissions.
- The following three represent group permissions.
- The last three represent permissions for others.

Octal notation:

- Use of octal. e.g. 754 corresponds to rwxr-xr--

Special Permission Bits

Linux also supports special permission bits that extend the basic DAC model by modifying how files and directories behave:

Sticky Bit (t)

Applied primarily to directories, the sticky bit restricts deletion or renaming of files inside the directory. Only the file's owner, the directory's owner, or the root user can delete or rename files. This is commonly used on shared directories like /tmp to prevent users from deleting each other's files.

Set User ID (SUID) (s on user permission bits)

When set on an executable file, SUID causes the program to run with the privileges of the file's owner rather than the user who launched it. This is useful for programs requiring temporary elevated privileges, such as passwd, which allows users to change their passwords.

Set Group ID (SGID) (s on group permission bits)

When set on an executable, SGID causes the program to run with the group privileges of the file rather than the user's group.

Example symbolic permissions with special bits

```
-rwsr-xr-x #SUID set (s replaces x in user permissions)
drwxrwsr-x #SGID set on a directory (s replaces x in group permissions)
drwxrwxrwt #Sticky bit set on a directory (t replaces x for others)
```

How DAC Works in Practice

When a user attempts to access a file, the kernel checks the user's identity and group membership, then compares this to the file's permission bits to decide whether to allow the requested operation.

Example:

A file with permissions -rw-r----- owned by user Alice and group staff:

- Alice can read and write the file.
- Members of staff group can read the file.
- Others have no access.

Changing Permissions and Ownership

chmod

Used to change the access permissions of a file or directory.

```
chmod 755 /path/to/file     # Gives rwx to the owner, and r-x to group and others
chmod u+s /usr/bin/passwd   # Sets the SUID bit on passwd to run with the owner's privileges
chmod g+w /shared/folder    # Adds write permission for the group on a shared folder
```


chown

Changes the owner and/or group associated with a file or directory.

```
chown Alice /home/Alice/file.txt          # Changes the file owner to Alice
chown Alice:developers /var/www/project   # Changes owner to Alice and group to developers
chown -R root:root /etc/ssl               # Recursively changes owner and group in /etc/ssl
```

chgrp

Changes only the group owner of a file or directory.

```
chgrp developers /shared/project           # Changes the group of the file or folder to developers
chgrp -R staff /mnt/data                   # Recursively changes the group to staff in /mnt/data
chgrp wheel /usr/local/bin/some_executable # Assigns the group wheel to an executable
```

Default File Permissions and the Role of *umask*

In Linux, when a new file or directory is created, it receives default permissions. These are then altered by a user or system defined mask, called *umask*, which restricts specific permission bits. This mechanism is essential for defining secure defaults and preventing excessive permissions on new files.

The umask controls the default permission set of newly created files and directories. It acts as a filter that removes permission bits from the system defaults. *Each user session has its own umask value*.

By default, permissions for files and directories without umask are:

- Files: Created by default with permission bits 666 (read and write for everyone)
- Directories: Created with 777 (read, write, and execute for everyone)

The umask value determines which permissions are *removed* from these defaults.

Defining umask: Octal and Symbolic Notation

You can define umask using either octal or symbolic notation:

- Octal Notation

```
umask 0027
```

In practice, only the last three digits of the umask are commonly used (user, group, others). The first digit exists but is rarely applied, as it controls special permission bits (setuid, setgid, sticky).

- Symbolic Notation

```
umask u=rwx,g=rx,o=
```

This is equivalent to umask 027. It means:

- User has read, write, execute
- Group has read and execute
- Others have no permissions

Bitwise Calculation and Logic

The umask is a bitwise mask that subtracts permissions from the defaults using a bitwise AND NOT operation.

	User	Group	Other
Default (666)	110	110	110
umask (027)	000	010	111
Result (AND-NOT)	110	100	000

Permissions become 640 (rw- r-- ---).

Precedence Order for umask Configuration

The system determines the effective umask using the following priority:

- Shell scripts in /etc/profile.d/ : highest priority, overrides all others
- /etc/profile : global settings for login shells
- PAM module pam_umask.so : defined in /etc/pam.d/postlogin
- /etc/login.defs : system-wide configuration file for user defaults
- /etc/default/login : lowest priority, rarely used

Check /etc/login.defs for a UMASK directive. If missing or commented out, the PAM module often falls back to its built-in default (commonly 002), hardcoded in the binary.

Temporary Definition for Current Session

You can set a temporary umask that only applies to the current shell session:

```
umask 027
umask u=rwx,g=rx,o=
```

Use this to restrict permissions during sensitive operations or scripting.

II.4. Mandatory Access Control (MAC)

Mandatory Access Control (MAC) is a security model in which access decisions are enforced by a central authority based on predefined policies and not left to the discretion of individual users (as is the case with Discretionary Access Control, or DAC). In MAC, users cannot change access permissions on files they own. Instead, access is mandated by system-wide policies defined by administrators.

AppArmor (Application Armor)

AppArmor is a Linux kernel security module that implements Mandatory Access Control (MAC). Unlike SELinux, which is label-based and policy-heavy, AppArmor uses *path-based* profiles to confine programs, restricting their capabilities based on the filesystem paths they access.

It is designed to be simpler and more intuitive than SELinux, making it easier to understand and configure, especially for administrators new to MAC.

Documentation:
https://blog.stephane-robert.info/docs/securiser/durcissement/apparmor/
https://gitlab.com/apparmor/apparmor/-/wikis/QuickProfileLanguage

- Typical files and folders:
  - */etc/apparmor/*: Main directory containing all AppArmor profiles
  - */etc/apparmor.d/*: Profiles stored as individual files for each program
  - */etc/apparmor.d/disable/*: Directory where profiles are symlinked when disabled
  - */etc/apparmor.d/abstractions/*: Holds reusable rule sets (abstractions) that can be included in multiple profiles for common permissions (e.g., access to basic system libraries or networking).
  - */etc/apparmor.d/tunables/*: Contains variable definitions used in profiles, such as paths that vary between distributions or systems (e.g., user home directories). These files define macros like @{HOME} or @{PROC} used throughout profiles.

Install and activate AppArmor

```
sudo apt update
sudo apt -y install apparmor apparmor-profiles apparmor-utils

sudo systemctl start apparmor       # Starts the AppArmor service.
sudo systemctl enable apparmor      # Enables AppArmor to start automatically on boot.
```


AppArmor Profile

An AppArmor profile is a set of security rules that define what resources (files, directories, network, capabilities) a specific program can access and what actions it can perform. Each profile is associated with the path of an executable and specifies allowed permissions in a simple, readable syntax. Profiles can grant read, write, execute access to files, allow or deny network connections, and control Linux capabilities.

Profiles are typically located in */etc/apparmor.d/*.

- nginx AppArmor profile creation
  - You can either write an AppArmor profile from scratch, or use *aa-genprof* to generate one by analyzing application behavior. This requires log data, which can be provided by rsyslog, or by enabling persistent logging with journald (so that /var/log/syslog or equivalent is available):

```
sudo aa-genprof nginx   # Launches the interactive AppArmor profile generation tool for nginx to create or update its profile.
```

  - In another terminal, start nginx and generate traffic (e.g., open it in a browser) to trigger file accesses:

```
sudo systemctl start nginx   # Starts the nginx service.
```

  - In the aa-genprof terminal, you'll be prompted to review and allow or deny accesses. Follow the prompts.
  - Once finished, the profile will be saved in /etc/apparmor.d/usr.sbin.nginx.
  - After generation, the profile can be edited to refine permissions. For example:

```
abi <abi/3.0>,

include <tunables/global>

/usr/sbin/nginx {
  include <abstractions/base>
  include <abstractions/dovecot-common>
  include <abstractions/postfix-common>
  include <abstractions/totem>

  capability dac_override,

  /usr/sbin/nginx mr,
  /var/log/nginx/access.log w,
  /var/log/nginx/error.log w,
  /var/www/html/ rw,
  owner /etc/group r,
  owner /etc/nginx/mime.types r,
  owner /etc/nginx/nginx.conf r,
  owner /etc/nginx/sites-available/default r,
  owner /etc/nsswitch.conf r,
  owner /etc/passwd r,
  owner /run/nginx.pid r,
  owner /run/nginx.pid w,
}
```

- After editing a profile, you must reload it for the changes to take effect:

```
sudo apparmor_parser -r /etc/apparmor.d/usr.sbin.nginx   # Reloads the nginx AppArmor profile to apply any changes without
restarting the service.
```

AppArmor supports the #include directive to reuse predefined rule sets and improve readability. These include files are located in:

- /etc/apparmor.d/abstractions/
- /etc/apparmor.d/tunables/

AppArmor Modes

AppArmor profiles can operate in two main modes: enforce and complain. In enforce mode, the profile actively blocks any actions that are not explicitly allowed, preventing the program from performing unauthorized operations. This mode is used in production environments for strong security guarantees. In complain mode (also called learning mode), the profile does not block unauthorized actions but logs them instead, allowing administrators to monitor what would have been denied without disrupting the program's functionality. This mode is useful for debugging and creating new profiles safely before enforcing them.

```
sudo aa-enforce /etc/apparmor.d/usr.sbin.nginx  # Switch to enforce mode
sudo aa-complain /etc/apparmor.d/usr.sbin.nginx # Switch to complain mode
sudo aa-disable /etc/apparmor.d/usr.sbin.nginx  # Disable profile (unload it)
```

With *aa-disable*, AppArmor will *disable* the profile by *unloading* it from the kernel and *preventing it from being reloaded at boot*. This is not done by deleting or editing the profile itself, but rather by creating a symbolic link in a special directory: /etc/apparmor.d/disable/.

Verifying the Profile Status

Once an AppArmor profile is created or modified, it's important to verify that it is correctly loaded and operating in the intended mode. To list all currently loaded AppArmor profiles and their modes:

```
sudo apparmor_status    # Displays the current status of AppArmor, including loaded profiles and their modes.
sudo aa-status          # Shows detailed information about AppArmor profiles, including which are enforced, complain, or disabled.
```


SELinux

SELinux (Security-Enhanced Linux) is a powerful Mandatory Access Control (MAC) system originally developed by the NSA and integrated into the Linux kernel. Like AppArmor, SELinux enforces security policies that restrict programs' capabilities beyond traditional Unix permissions. However, SELinux uses a more complex and fine-grained policy language based on labels and rules, offering stronger and more flexible security controls.

Documentation:
https://blog.stephane-robert.info/docs/securiser/durcissement/selinux/
https://documentation.suse.com/fr-fr/sle-micro/6.0/html/Micro-selinux/index.html

Comparison SELinux with AppArmor

SELinux and AppArmor are both MAC systems designed to restrict programs' capabilities beyond standard Linux permissions. However, they differ in their approach:

- SELinux uses a *label-based system*, where all files, processes, and objects have security contexts (labels). Policies define how these labeled entities can interact, providing fine-grained control. SELinux is considered more complex and powerful but has a steeper learning curve.
- AppArmor relies on *path-based profiles*, which define what files or directories a program can access. It is generally easier to configure and understand but may offer slightly less granularity than SELinux.

Installing and activating SELinux on Debian-based systems

SELinux is not installed or enabled by default on Debian or derivatives such as Ubuntu. To install SELinux tools and policies, you can use the following commands:

```
sudo apt update
sudo apt install selinux-basics selinux-policy-default auditd
sudo selinux-activate
sudo reboot
```

SELinux Security Labels (Contexts)

SELinux operates by assigning security labels, also called security contexts, to all files, processes, and system objects. A security context typically has four parts:

user:role:type:level

- user: The SELinux user identity.
- role: Defines the role of the process or user (e.g., system_r).
- type: The most important part, determines the type enforcement rules (e.g., httpd_t for the Apache process).
- level: Used for Multi-Level Security (MLS), often left default in typical setups.

When a process tries to access a file or resource, SELinux checks the policy rules to see if the process's context is allowed to interact with the object's context in the requested way (read, write, execute, etc.).

This label-based system enables fine-grained security enforcement, limiting the damage a compromised process can do.

II.5. ACL (Access Control Lists) Management

ACLs (Access Control Lists) are an extension to the traditional Unix file permission model. While standard permissions allow defining access for the owner, group, and others, ACLs provide fine-grained control by allowing you to specify permissions for multiple users and groups beyond the defaults.

ACLs are especially useful in collaborative environments where multiple users need different levels of access to the same files or directories.

Documentation: https://doc.ubuntu-fr.org/acl

Each file or directory can have:

- A base set of traditional permissions (owner, group, others),
- An optional ACL defining extra rules for specific users or groups,
- A mask, which limits the effective permissions of named users and groups.

ACLs can be enabled on most Linux filesystems (e.g., ext4, xfs) by mounting them with the acl option (though it's often enabled by default).

ACL Entry Types

An ACL entry format is:

[entity]:[name]:[permissions]

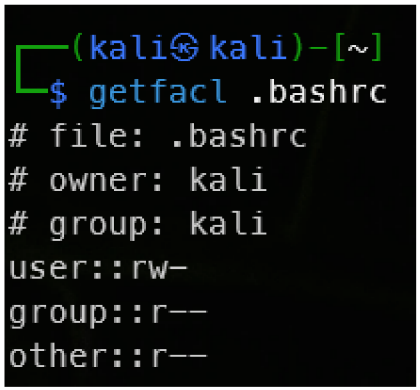

Figure 10 – bashrc default acl

Where:

- *user:username:rw-*: user-specific rule
- *group:groupname:r--*: group-specific rule
- *mask::r-x*: mask that caps all named users/groups
- *other::r-x*: "others" class

getfacl

Displays the ACLs associated with a file or directory

```
getfacl file.txt              # Shows extended ACL entries
getfacl -R /path/to/item      # Display the ACLs of a directory and its files with recursion
getfacl --skip-base -R .      # Display the ACLs of a a folder and skip items that have no ACLs with recursion
```


setacl

Modifies or sets ACLs on files and directories.

```
setfacl -m u:Alice:r file.txt          # Grants read permission on file.txt to user Alice
setfacl -R -m u:Bob:rwx /project       # Applies ACLs recursively to directory contents
setfacl -x u:Alice file.txt            # Removes ACL for user Alice
setfacl -d -m u:john:rwX /shared       # All new files under /shared will inherit ACLs for john
```

The default ACLs allow inheritance ACL permissions to be granted for any subdirectories or files created in a directory. However, these default ACLs do not apply to objects already present in the directory. When configuring a share with multiple accesses, it will therefore be necessary to proceed in two steps:

1. Modify the ACL of existing files
2. Apply a default ACL

Coexistence of ACLs and Standard Permissions

ACLs coexist with standard permissions. If both are set, ACLs take precedence for listed users.

The ls -l output will show a + sign after the mode if ACLs are present:

```
-rw-r--r--+ 1 user group 1234 Apr 10 10:00 file.txt
```


ACLs and Backup Tools

Not all backup or file management tools preserve ACLs by default. When ACLs are an essential part of your security model, it's crucial to ensure they are preserved during backup and restore operations.

Here are a few examples of tools and options that do support ACL preservation:

```
rsync -aA /data/ /backup/data/        # The -a option enables archive mode, and -A ensures ACLs are preserved during transfer.
tar --acls -cf backup.tar /project    # Creates a tar archive of /project including ACLs.
cp -a /source/file /dest/             # The -a (archive) flag in cp also preserves ACLs among other metadata.
```

You should always verify tool compatibility in production environments. Some older backup systems may silently ignore ACLs, potentially weakening your security controls after restore.

II.6. Linux Capabilities Management

Traditional Unix security is binary: a process either runs as root (UID 0) and can do everything or runs as an unprivileged user and can do almost nothing outside DAC/MAC rules. Linux capabilities split root's omnipotence into fine-grained privileges that can be independently granted to processes or attached to executables via extended attributes. This enables least privileged designs (e.g., allow binding to a privileged TCP port without granting full root).

Documentation: https://man7.org/linux/man-pages/man7/capabilities.7.html

Key goals:

- Remove setuid root whenever possible.
- Grant only the minimal capabilities needed to an executable or service.
- Limit damage in case of compromission

A Classic Example: ping

Traditionally, ping used raw sockets, which require root privileges. Older systems made /bin/ping setuid root. This means any user executing it runs the binary with full root access which could be an attack vector.

On modern systems with filesystem support and libcap installed, ping no longer needs to be setuid. It receives the minimal capability cap_net_raw, allowing it to open raw sockets, and nothing more:

```
getcap /bin/ping
# /bin/ping cap_net_raw=ep
```

cap_net_raw: allows sending/receiving raw network packets. ep: granted to the Effective and Permitted capability sets when executed. This change greatly reduces attack surface.

Managing File Capabilities

- To assign a capability to a file, you must be root:

```
sudo setcap cap_net_raw=+ep /bin/ping
```

- To remove capabilities from a file:

```
sudo setcap -r /bin/ping
```

sudo setcap -r /bin/ping

- To inspect capabilities on binaries recursively:

```
sudo getcap -r / 2>/dev/null
```

These capabilities are stored as extended file attributes under the security.capability namespace.

Security Implication

Using capabilities instead of setuid binaries aligns with the principle of least privilege:

- The binary gets only the permission it needs
- Exploiting the binary no longer grants full root access
- It reduces the risk of privilege escalation vulnerabilities

Administrators should:

- Regularly audit binaries using:

```
sudo getcap -r / 2>/dev/null
```

- Replace setuid root binaries with capabilities whenever possible
- Avoid assigning dangerous capabilities like cap_sys_admin or cap_sys_module unless absolutely necessary