Tutorial 3 (Access Control)

In this tutorial, we continue with passwords and learn the basics of access control in Linux.

🛡️ 1. Access Control on Linux

In the lecture, we learned that **Linux implements access control via file permissions**—this is the
primary mechanism that users interact with. This concept is powerful because Linux (and UNIX)
treats **every object as a type of file**.

📂 How File Permissions Work

File permissions are grouped and applied in a structured way:

- A file **always has an owner**.
- A file **always has an associated group**; this group may have the same name as the owner
  (for every user, there is also a corresponding group), but it can also be any group defined on the system.

Permissions are then set for three categories:

1. **Owner** – the user who owns the file
2. **Group** – the group associated with the file
3. **Other** – everyone else on the system

🔑 Types of Permissions

There are three fundamental file permissions in Linux:

1. **Read (r)** – allows viewing or reading the contents of a file
2. **Write (w)** – allows modifying or deleting the contents of a file
3. **Execute (x)** – allows running a file (if it is a script or program)

These permissions form the core of Linux access control and are critical for securing files and resources.

🔍 a) Inspecting the List of Users

i) Take a look at the file `/etc/passwd`. Try to understand its contents.

- You can use commands like `cat` or `more` (e.g., `cat /etc/passwd`) to view its contents, or open it in an editor such as `vi` or `nano`.

- You can do this in a **separate terminal**.
- Alternatively, you can also use Jupyter Notebook. *(In Jupyter, prefix command-line commands with `!`)*.
- **Example:**
  `!cat /etc/passwd`

In [4]:
```
!cat /etc/passwd
```

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
_apt:x:42:65534::/nonexistent:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-network:x:998:998:systemd Network Management:/:/usr/sbin/nologin
dhcpcd:x:996:996:DHCP Client Daemon:/usr/lib/dhcpcd:/bin/false
messagebus:x:995:995:System Message Bus:/nonexistent:/usr/sbin/nologin
syslog:x:100:101::/nonexistent:/usr/sbin/nologin
systemd-resolve:x:989:989:systemd Resolver:/:/usr/sbin/nologin
_chrony:x:988:988:Chrony Daemon:/var/lib/chrony:/usr/sbin/nologin
tss:x:987:987:tss user for tpm2:/:/usr/sbin/nologin
uuidd:x:101:105::/run/uuidd:/usr/sbin/nologin
systemd-oom:x:986:986:systemd Userspace OOM Killer:/:/usr/sbin/nologin
whoopsie:x:102:108::/nonexistent:/bin/false
dnsmasq:x:999:65534:dnsmasq:/var/lib/misc:/usr/sbin/nologin
avahi:x:103:109:Avahi mDNS daemon:/run/avahi-daemon:/usr/sbin/nologin
nm-openvpn:x:984:984:NetworkManager OpenVPN:/var/lib/openvpn/chroot:/usr/s
bin/nologin
tcpdump:x:983:983:tcpdump:/nonexistent:/usr/sbin/nologin
sssd:x:104:110:SSSD system user:/var/lib/sss:/usr/sbin/nologin
speech-dispatcher:x:105:29:Speech Dispatcher:/run/speech-dispatcher:/bin/f
alse
usbmux:x:106:46:usbmux daemon:/var/lib/usbmux:/usr/sbin/nologin
cups-pk-helper:x:107:111:user for cups-pk-helper service:/nonexistent:/us
r/sbin/nologin
fwupd-refresh:x:982:982:Firmware update daemon:/var/lib/fwupd:/usr/sbin/no
login
saned:x:108:112::/var/lib/saned:/usr/sbin/nologin
geoclue:x:109:113::/var/lib/geoclue:/usr/sbin/nologin
cups-browsed:x:110:111::/nonexistent:/usr/sbin/nologin
pipewire:x:981:981:system user for pipewire:/nonexistent:/usr/sbin/nologin
hplip:x:111:7:HPLIP system user:/run/hplip:/bin/false
gnome-remote-desktop:x:980:980:GNOME Remote Desktop:/var/lib/gnome-remote-
desktop:/usr/sbin/nologin
polkitd:x:979:979:User for polkitd:/:/usr/sbin/nologin
rtkit:x:978:978:RealtimeKit:/proc:/usr/sbin/nologin
colord:x:977:977:colord colour management daemon:/var/lib/colord:/usr/sbi
n/nologin
gdm:x:975:975:Gnome Display Manager:/var/lib/gdm3:/bin/false
ubuntu:x:1000:1000:ubuntu:/home/ubuntu:/bin/bash
vboxadd:x:997:1::/var/run/vboxadd:/bin/false
tutorialuser1:x:1001:1001::/home/tutorialuser1:/bin/bash
tutorialuser2:x:1002:1002::/home/tutorialuser2:/bin/bash
```

> 💡 **What do you find in there?**

- Each user has a single line entry.
- Each entry contains:
  - **Username**
  - An `x` indicating the password is stored in an encrypted form (in a separate file)
  - **User ID (UID)** and **Group ID (GID)**
  - **Home directory**
  - **Default shell**

ii) Try to log in as the `tutorialuser1` user. What happens?

- i.e., Use `su tutorial1`

Solution:
When you run `su tutorialuser1`, it will prompt you for the password of the `tutorialuser1` account.
Since you do not know the password, you will **not be able to log in**.

iii) Use `sudo su` to become root and then use the `su` command to switch to the `tutorialuser1` user.

- i.e., run `sudo su` followed by `su tutorialuser1`.
- Use the command `whoami` to verify the current username.
- Use the command `cd` to navigate to the home directory of `tutorialuser1`.
- Use the command `pwd` to print the home directory path.

Solution:

1. Run `sudo su` to become the root user.
2. Then run `su tutorialuser1` to switch to the `tutorialuser1` account.
3. Use `whoami` and confirm that the logged-in user is `tutorialuser1`.
4. Run `cd` to navigate to the home directory of `tutorialuser1`.
5. Finally, use `pwd` to verify that you are in `/home/tutorialuser1`.

iv) Try to log in (using the `su` command) as some of the entries that don't appear to have human names (e.g., `www-data`, `mail`). What happens?

- **Example:** `su mail` (this will ask for a password, and you again don't have the password).
- Become root and try again:
  - **Example:** `sudo su`
  - Then run: `su mail` or `su www-data`
- You will see that you still can't log in.

- This is because these users are **not meant to log in from a normal shell**.
- If you inspect the `/etc/passwd` file, you'll notice that these accounts are configured with
  shells like `/usr/sbin/nologin` or `/bin/false`, which prevents interactive logins.

v) Find out what the `/usr/sbin/nologin` entry in the `/etc/passwd` file does.

Solution:
It indicates that the user is **not allowed to log in**.

- Read more at https://linuxconfig.org/disabling-user-logins-to-linux-system

b) More on Users

i) While logged in as the `ubuntu` user (or any other normal user), try to view the `/etc/shadow` file. What happens?

Solution:

- Command `cat /etc/shadow`
- You will get a **permission denied** error when trying to view `/etc/shadow` as a normal user:

ii) How can you solve this issue?

Solution:
To view the contents of `/etc/shadow`, you need elevated privileges. Use the following command:

`sudo cat /etc/shadow`

In [10…

```
!echo ubuntu123 | sudo -S cat /etc/shadow
```

```
[sudo: authenticate] Password:
root:*:20566:0:99999:7:::
daemon:*:20566:0:99999:7:::
bin:*:20566:0:99999:7:::
sys:*:20566:0:99999:7:::
sync:*:20566:0:99999:7:::
games:*:20566:0:99999:7:::
man:*:20566:0:99999:7:::
lp:*:20566:0:99999:7:::
mail:*:20566:0:99999:7:::
news:*:20566:0:99999:7:::
uucp:*:20566:0:99999:7:::
proxy:*:20566:0:99999:7:::
www-data:*:20566:0:99999:7:::
backup:*:20566:0:99999:7:::
list:*:20566:0:99999:7:::
irc:*:20566:0:99999:7:::
_apt:*:20566:0:99999:7:::
nobody:*:20566:0:99999:7:::
systemd-network:!*:20566:::::1:
dhcpcd:!*:20566:::::1:
messagebus:!*:20566::::::
syslog:!:20566::::::
systemd-resolve:!*:20566:::::1:
_chrony:!*:20566::::::
tss:!*:20566::::::
uuidd:!:20566::::::
systemd-oom:!*:20566:::::1:
whoopsie:!:20566::::::
dnsmasq:!:20566::::::
avahi:!:20566::::::
nm-openvpn:!*:20566:::::1:
tcpdump:!*:20566:::::1:
sssd:!:20566::::::
speech-dispatcher:!:20566::::::
usbmux:!:20566::::::
cups-pk-helper:!:20566::::::
fwupd-refresh:!*:20566::::::
saned:!:20566::::::
geoclue:!:20566::::::
cups-browsed:!:20566::::::
pipewire:!*:20566::::::
hplip:!:20566::::::
gnome-remote-desktop:!*:20566::::::
polkitd:!*:20566::::::
rtkit:!*:20566:::::1:
colord:!*:20566::::::
gdm:!*:20566::::::
ubuntu:$6$r.p1Q8y/65i1TYWn$FEzSj562DBt7bfGwDuTzlN9he2sHWQKV9DmspGo8puKf
Q6lt7fJRSIApEBVbAe..CGbwLfmyvKB84g/.XiCaF/:20641:0:99999:7:::
vboxadd:!:20641::::::
tutorialuser1:$y$j9T$5ZRzcJEszDCoapFCbC2EE0$ZIL1oivipjjgbNzDdKR.1kiH0fT
3aKPwEtOalRyqU55:20642:0:99999:7:::
tutorialuser2:$y$j9T$3rHfCMI0MO.3jwLASc3zB0$Hp2jtMVrPvR9nxl8RO0KyWGNpfa
iNFtww7lOWUlBx28:20642:0:99999:7:::
```

iii) What is the content of this file?

- Read about it here https://www.cyberciti.biz/faq/understanding-etcshadow-file/

> **Solution:**
> The `/etc/shadow` file stores encrypted password information and related security settings for user accounts.
> It is accessible only by the root user for security reasons.

iv) What are the file permissions for `/etc/shadow`, compared to `/etc/passwd`?

- If you don't know how to find them, type `man ls` to learn about the command and the necessary options.

> **Solution:**
>
> - Commands to use
>   - `ls -ltr /etc/passwd`
>   - `ls -ltr /etc/shadow`
> - The outputs will look like the below
>   - `-rw-r--r-- 1 root root 2026 Jul 29 05:51 /etc/passwd`
>   - `-rw-r----- 1 root shadow 1369 Jul 29 05:51 /etc/shadow`
> - The `/etc/passwd` file is owned by `root`. Root can read and write to it, while all others have read-only access.
> - The `/etc/shadow` file is owned by `root` and belongs to the `shadow` group. Root can read and write to it, members of the `shadow` group can read it, and all others have no access.
>
> **Explanation:**
>
> The `/etc/passwd` file contains basic user account information that needs to be readable by all users and system processes, which is why it has **read permissions for everyone**. However, only the root user can modify it to protect against unauthorized changes.
>
> The `/etc/shadow` file stores sensitive password hashes and security-related information. Because of its sensitive nature, access is restricted to the root user and the `shadow` group only. This restriction protects user credentials from being exposed to normal users or processes, enhancing system security.

c) Executing Files

- List the permissions of the **yes** program. **Hint:** You can find the location of this file by typing `which yes` into your terminal 🙂
- Why are you able to execute it?

Solution:

- tutorialuser1@lab22YQ9P:$ `which yes`
  - /usr/bin/yes
- tutorialuser1@lab22YQ9P:$ ls -ltr /usr/bin/yes -rwxr-xr-x 1 root root 39256 Sep 5 2019 /usr/bin/yes

Since there is `x` permission for the owner, group, and others, all users can execute it.

d) Setting permissions for files

An owner can set permissions for a file using the `chmod` command. Interestingly, there are several ways to specify permissions.
For example, one can use the syntax `g+r` / `g-r` to add or remove read permission for the group (similarly, `u+r` and `o+r` apply to the user and "others").

Another approach is to write the permissions in **octal notation**, where the first digit refers to the user, the second to the group, and the third to "others". Each digit encodes three bits:

- A value of **7** in octal is `111` in binary, meaning **read**, **write**, and **execute** permissions are all granted.
- For example, `chmod 777` would give full permissions to the user, group, and others.

Values of each permission setting:

Value (Octal)	Value (Binary)	Permission Granted
1	001	Execute
2	010	Write
4	100	Read

Question:
What would be the command to set the permissions of a file to match those of `/etc/passwd`?

Solution:

- ubuntu@lab22YQ9P:$ `ls -ltr /etc/passwd`
  - `-rw-r--r-- 1 root root 2026 Jul 29 05:51 /etc/passwd`

The permission string `-rw-r--r--` means:

- **Owner (user):** read and write (`rw-`)
- **Group:** read-only (`r--`)
- **Others:** read-only (`r--`)

This translates to octal **644** (User = 6, Group = 4, Others = 4).

- `touch test_file.txt` Create a new file named test_file.txt
- `chmod 644 test_file.txt` Change the permissions to 644
- `ls -ltr test_file.txt` Check the set permissions

The output will show `test_file.txt` now has the same permissions `rw-r--r--` as `/etc/passwd`.

e) Owners and groups

Now let's explore file ownership in more detail.

- How can you change the **owner** or **group** of a file?
- Try changing the owner of a file that belongs to `root`. Were you successful?
- Now try the same operation as the `root` user. Does it work this time?

Solution:

- ubuntu@lab22YQ9P:$ `ls -ltr test_file.txt`
  - `-rw-r--r-- 1 ubuntu ubuntu 0 Jul 29 06:20 test_file.txt`
- ubuntu@lab22YQ9P:$ `sudo chown root:root test_file.txt`
- ubuntu@lab22YQ9P:$ `ls -ltr test_file.txt`
  - `-rw-r--r-- 1 root root 0 Jul 29 06:20 test_file.txt`

As shown above, running `chown` with `sudo` (as root) successfully changes the **owner** and **group** of the file to `root`.

Once you do the change, You will not be able to change the ownership of a file owned by `root` as a normal user.

2. Potential problems with setuid/setgid

Back to file permissions again: setting permissions for a file doesn't just involve 3 octal digits (e.g., 444, 761, ...); it actually involves **4 digits**!
An extra digit can be placed at the start of the command, which is used to set the following special bits:

- **setuid bit (4):** When this bit is set on an executable file, the program runs with the privileges of the file's owner rather than the user who runs it. This is often used for programs that need elevated privileges (e.g., `passwd`).
- **setgid bit (2):** Similar to setuid, but it applies the file's group privileges instead of the user's. For directories, it ensures that files created inside inherit the directory's group.
- **sticky bit (1):** On directories, this bit ensures that only the file's owner (or root) can delete or rename the files inside, even if others have write permissions.

> ⚠️ **Note:** Misuse of the setuid or setgid bits can lead to security vulnerabilities because they can grant elevated privileges.

a) Setting the bits

- How can you set a file to have both the setuid and setgid bits enabled?
- Provide the command(s) to set a file's owner to be `bertie` and to enable its setuid bit. *(Hint: you may need to create the user `bertie` first using `man adduser`.)*

Solution:

1. **Create a file named my_program**

- ubuntu@lab22YQ9P:$ `touch my_program`

3. **Set both setuid and setgid bits on a file:**

- ubuntu@lab22YQ9P:$ `chmod 6755 my_program`
  - `6` at the start sets both **setuid (4)** and **setgid (2)** bits.
  - `755` sets `rwxr-xr-x` (owner: read/write/execute; group & others: read/execute).
- ubuntu@lab22YQ9P:$ `ls -ltr my_program`
  - `-rwsr-sr-x 1 ubuntu ubuntu 12345 Jul 29 06:30 my_program`
  - Notice the `s` in the owner and group execute positions indicating setuid and setgid bits are set.

---

3. **Set a file's owner to `bertie` and enable the setuid bit:**

- `sudo adduser bertie` *(if `bertie` does not already exist)*
- `sudo chown bertie my_program`
- `sudo chmod 4755 my_program`
- `ls -ltr my_program`
  - `-rwsr-xr-x 1 bertie ubuntu 12345 Jul 29 06:35 my_program`

> Here,
>
> - `4` at the start sets only the **setuid** bit.
> - `755` applies normal file permissions.
> - The lowercase `s` (or uppercase `S` if execute is not set) in the owner's execute position shows the setuid bit is enabled.

b) Special bits on a directory

When applied to directories, these special bits have a slightly different meaning.

- Create a directory named `bertiesworld`. Change its group ownership to `bertie`. Then set the appropriate directory bits so that any subdirectory or file created inside `bertiesworld` automatically inherits the group `bertie`.
- In what scenarios could this behavior be useful?
- Could this introduce any potential security risks?

> **Solution:**
>
> 1. **Create the directory and set the group to `bertie`:**
>
> - `ubuntu@lab22YQ9P:$ mkdir bertiesworld`
> - `ubuntu@lab22YQ9P:$ ls -ltr`
>   - `drwxrwxr-x 2 ubuntu ubuntu 4096 Jul 30 12:02 bertiesworld`
> - `ubuntu@lab22YQ9P:$ sudo chown ubuntu:bertie bertiesworld/`
> - `ubuntu@lab22YQ9P:$ ls -ltr`
>   - `drwxrwxr-x 2 ubuntu bertie 4096 Jul 30 12:02 bertiesworld`
>
> ---
>
> 2. **Create a test file inside the directory:**
>
> - `ubuntu@lab22YQ9P:$ cd bertiesworld/`
> - `ubuntu@lab22YQ9P:$ touch test_file.txt`
> - `ubuntu@lab22YQ9P:$ ls -ltr`
>   - `-rw-rw-r-- 1 ubuntu ubuntu 0 Jul 30 12:04 test_file.txt`
>
> > **Note:** The group is still `ubuntu`, not `bertie`.
>
> ---
>
> 3. **Set the setgid bit on the directory:**

- `ubuntu@lab22YQ9P:$ cd ../`
- `ubuntu@lab22YQ9P:$ sudo chmod g+s bertiesworld/`
- `ubuntu@lab22YQ9P:$ ls -ltr`
  - `drwxrwsr-x 2 ubuntu bertie 4096 Jul 30 12:06 bertiesworld`

> **Note:** The `s` in the group execute position shows that the **setgid** bit is now set.

---

4. **Create another test file and check the group:**

- `ubuntu@lab22YQ9P:$ cd bertiesworld/`
- `ubuntu@lab22YQ9P:$ touch test_file_bertie_2.txt`
- `ubuntu@lab22YQ9P:$ ls -ltr`
  - `-rw-rw-r-- 1 ubuntu bertie 0 Jul 30 12:13 test_file_bertie_2.txt`

Now, all new files and subdirectories created inside `bertiesworld` automatically inherit the group `bertie`.

When is this useful?

- This approach is useful when multiple users are collaborating and need shared group access to files and directories.
- **Potential risk:** It can become harder to track who has access if group memberships are not updated regularly. For example, a user who was removed from the project might still belong to the group and retain access.

c) Executables with setuid

- Why do some programs require the **setuid** bit?
- Are there any programs in your `/usr/bin` directory that have the setuid bit set and are owned by `root`? *(Hint: you can use the `find` command with the `-perm` flag.)*

Solution:

Some programs need to run with elevated (superuser) privileges even when the user running them does not have superuser access. The **setuid** bit allows these executables to run with the permissions of the file's owner (often `root`), which is required for tasks such as changing passwords or mounting filesystems.

To find all executables in `/usr/bin` that have the **setuid** bit set and are owned by `root`, you can run:

- `ubuntu@lab22YQ9P:$ find /usr/bin -user root -perm -4000 -exec ls -ldb {} \; > /tmp/filename`

Example output

`ubuntu@lab22YQ9P:~$ cat /tmp/filename`

- `-rwsr-xr-x 1 root root 39144 Mar 7 2020 /usr/bin/fusermount`
- `-rwsr-xr-x 1 root root 85064 Feb 6 2024 /usr/bin/chfn`
- `-rwsr-xr-x 1 root root 31032 Feb 21 2022 /usr/bin/pkexec`
- `-rwsr-xr-x 1 root root 68208 Feb 6 2024 /usr/bin/passwd`
- `-rwsr-xr-x 1 root root 88464 Feb 6 2024 /usr/bin/gpasswd`
- `-rwsr-xr-x 1 root root 44784 Feb 6 2024 /usr/bin/newgrp`
- `-rwsr-xr-x 1 root root 39144 Apr 9 2024 /usr/bin/umount`
- `-rwsr-xr-x 1 root root 67816 Apr 9 2024 /usr/bin/su`
- `-rwsr-xr-x 1 root root 53040 Feb 6 2024 /usr/bin/chsh`
- `-rwsr-xr-x 1 root root 55528 Apr 9 2024 /usr/bin/mount`
- `-rwsr-xr-x 1 root root 166056 Apr 4 2023 /usr/bin/sudo`

3. Real, Effective, and Saved UIDs in Practice

In the `Tutorial_3` folder, you will find a C program named `whoami3.c`.

- Inspect the code. Open `whoami3.c` and read through it. What does it do?
- As user `ubuntu`, compile the program:

  ```
  gcc whoami3.c -o whoami3
  ```

- As the user `ubuntu`, make it executable:

  ```
  chmod +x whoami3
  ```

- Run the program:

  ```
  ./whoami3
  ```

- What do you see? Explain your observation (look closely at the Real, Effective, and Saved UID values).
- Now change the program's ownership to root and set the SUID bit:

```
sudo chown root:root whoami3
sudo chmod u+s whoami3
```

- Run `ls -l whoami3` and note what has changed in the permission bits.
- Now run the program again, as user `ubuntu`:

```
./whoami3
```

- What do you see? Explain your observation.*
  - How do the Real, Effective, and Saved UIDs compare to step 4?
  - Why did some values change and others didn't?
  - What does this tell you about how SUID affects process privileges?

Solution:

What does the code do? The program calls `getresuid()` to retrieve the calling process's Real, Effective, and Saved UIDs, then prints all three. It doesn't modify anything — it's purely observational, letting us see how the OS tracks these three separate identities for every process.

First run (before SUID), expected output:

`Real: 1000  Effective: 1000  Saved: 1000` (1000 is the UID of the user ubuntu)

All three UIDs are identical and equal to the `ubuntu` user's own UID. This is the normal case — when a process is launched directly by a user, the kernel sets Real = Effective = Saved = that user's UID. There is no privilege distinction; the process can only do what `ubuntu` is allowed to do.

After `chown root:root` + `chmod u+s`:

`Real: 1000  Effective: 0  Saved: 0` (Recall 1000 is the UID of the user ubuntu, and 0 is the UID of root)

The owner execute bit `x` has become `s` (lowercase, since owner execute was already set). This is the SUID bit — it tells the kernel: "whenever this file is executed, set the process's effective UID to the file owner's UID (root), regardless of who runs it."

- **Real UID stays 1001 (`ubuntu`)** — the kernel always preserves who *actually launched* the process. This never changes just because SUID

is set; it's used for accounting, permission checks like "can this process signal that process," and audit logging.

- **Effective UID becomes 0 (`root`)** — this is the direct effect of the SUID bit. The kernel sets the effective UID to match the *file owner* (root) at `exec()` time. All permission checks the kernel does from this point on (file access, etc.) use the *effective* UID, not the real one — so the process now has root-level access even though `ubuntu` launched it.
- **Saved UID also becomes 0 (`root`)** — the kernel copies the new effective UID into the saved UID at `exec()` time. This exists so that if the program later temporarily drops privilege (e.g., `seteuid(ruid)` to act as `ubuntu` for safety), it can restore root privilege afterward via `seteuid(suid)` without needing to re-authenticate. This is exactly how programs like `passwd` briefly de-escalate and re-escalate privilege while writing to `/etc/shadow`.

Key takeaway

SUID doesn't change *who ran* the program (Real UID) — it changes *what the program is allowed to do* (Effective UID), with the Saved UID acting as a "memory" of the elevated privilege so it can be restored after a temporary drop. This is the mechanism that lets an unprivileged user run a small, trusted, root-owned program (like `passwd`) to perform a specific privileged action, without giving that user root access to everything else.

4. Privilege Escalation and Fallback with Saved UID

In the `Tutorial_3` folder, you will find a C program named `whoami4.c`. This program builds on `whoami3.c` — instead of just reading the UIDs once, it prints them, temporarily drops root privilege, prints them again, then restores root privilege using the saved UID and prints them a third time.

- Inspect the code. Open `whoami4.c` and read through it. What does it do? What functions does it call, and what do you expect each one to change?
- As user `ubuntu`, compile the program:

```
gcc whoami4.c -o whoami4
```

- Change the program's ownership to root and set the SUID bit:

```
sudo chown root:root whoami4
sudo chmod u+s whoami4
```

- As user `ubuntu`, run the program:

```
./whoami4
```

- You should see three lines of output, labelled `Start`, `Dropped`, and `Restored`. For each line, record the Real, Effective, and Saved UID.
- Explain your observations:
  - At `Start`, why is the Effective UID `0` even though `ubuntu` launched the program?
  - At `Dropped`, which UID changed, and which stayed the same? Why did we lose root privilege here?
  - At `Restored`, how did the program get root privilege back *without* asking for a password or re-running as root?
  - If the Saved UID did not exist, would this "drop then restore" pattern still be possible? Why or why not?
- Real-world connection: the `passwd` command uses exactly this pattern — briefly dropping to your real privilege level, then restoring root privilege only when it needs to write to `/etc/shadow`. Why might a program want to *minimise* the time it spends running with full root privilege, rather than just keeping it for the whole execution?

Solution:

What does the code do? The program calls `getresuid()` three times, at different points in execution, to show how the Real, Effective, and Saved UIDs change as the process deliberately drops and then restores its privilege. First it prints the UIDs as soon as the program starts (`Start`). Then it calls `seteuid(ruid)` to temporarily drop its effective privilege down to the real (unprivileged) user and prints the UIDs again (`Dropped`). Finally it calls `seteuid(suid)` to restore its effective privilege back to root using the saved UID, and prints the UIDs a third time (`Restored`).

Expected output:

```
Start     -> Real: 1000  Effective: 0  Saved: 0

Dropped   -> Real: 1000  Effective: 1000  Saved: 0

Restored  -> Real: 1000  Effective: 0  Saved: 0
```

- **At `Start`**, the Effective UID is already `0` (root), even though `ubuntu` (UID 1000) launched the program. This is the same SUID mechanism from the previous task — the kernel sets the effective UID to the file owner's UID (root) at `exec()` time, regardless of who ran the file. The Saved UID is also `0`, copied from the effective UID at the same moment.

- **At `Dropped`**, only the Effective UID changes, from `0` back to `1000`. The Real UID was never root to begin with, so it's

unaffected. Crucially, the **Saved UID stays `0`** — `seteuid()` only changes the *effective* UID, it does not touch the saved UID. At this point the process is voluntarily running with the same restricted privilege as `ubuntu`, even though it's still technically the same SUID-root process. This is a deliberate safety measure: while performing routine, non-privileged work, the program avoids holding root power it doesn't currently need.

- **At `Restored`**, the Effective UID jumps back to `0`. This works because `seteuid(suid)` is allowed to set the effective UID to *any* value that matches the real, effective, or **saved** UID — and the saved UID was still `0` from the very first `exec()`. No password, no re-authentication, and no call back into `sudo` was needed. The kernel already "remembered" that this process was entitled to root privilege, via the saved UID.
- **If the Saved UID did not exist**, this pattern would be impossible. Once the process dropped its effective UID to `1000`, it would have no way to prove it was ever allowed to be root again — it would need to be re-executed from a SUID-root binary from scratch. The saved UID is what makes a *reversible* drop possible, as opposed to a *permanent* one.

Real-world connection: `passwd` follows exactly this drop → restore pattern. It runs as SUID root, drops privilege while doing routine work like reading input or checking your current password, and only restores root privilege for the brief window where it actually needs to write to `/etc/shadow`. Minimising the time spent at full privilege reduces the "attack window" — if the program crashes, is exploited via a bug, or handles malicious input, it does so with the least privilege necessary for as much of its execution as possible. This is the Principle of Least Privilege applied at the level of a single process's lifetime, not just at the level of user accounts.

Key takeaway

`seteuid()` moves privilege between the Real, Effective, and Saved UIDs without discarding any of them — unlike `setuid()`, which collapses all three to the same value and permanently gives up the ability to restore a higher privilege. This distinction is exactly why security-sensitive SUID programs use `seteuid()` for temporary privilege changes, and reserve `setuid()` only for the point where they want to drop privilege for good.