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

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

- i.e., Use `su tutorial1`

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.

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?

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

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?

ii) How can you solve this issue?

In [ ]:
```
!echo ubuntu123 | sudo -S cat /etc/shadow
```

iii) What is the content of this file?

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

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.

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?

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`?

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?

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`.)*

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?

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.)*

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?

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?