Users and Access Control

Ashkan Jangodaz
British Columbia Institute of Technology

Access Control

- Computers are shared resources.
- Even a personal laptop runs multiple programs simultaneously, and servers handle requests from thousands of users.
- This creates a fundamental problem: **how does the system decide who gets to do what?**

Analogy

- Imagine a filing cabinet in an office. Without any rules:
  - Anyone could read any document
  - Anyone could modify or delete files
  - Anyone could add misleading information
  - No one would know who did what
- This is chaos.
- Organizations solve this with locks, keys, and sign-out sheets. Computers solve it with **access control**.

Access Control

- Access control answers three questions:
  1. **Who are you?** (Authentication: proving identity)
  2. **What are you allowed to do?** (Authorization: checking permissions)
  3. **What did you do?** (Accounting: logging actions)

Least Privilege

- **Principle of Least Privilege**
  - Users and processes should have the **minimum permissions necessary** to do their job, and no more.
- **Why?**
  - Limits damage from mistakes (can't accidentally delete what you can't access).
  - Limits damage from compromise (attacker inherits your limited permissions).
  - Makes security auditing tractable.

Identity

- An identity is **how a system recognizes and tracks an actor**.
  - In the physical world, your identity might be proven by your face, your fingerprint, your driver's license, or your employee badge (card).
- In a computer system identity is typically represented by a **username.**
- **Identity management** matters so much for security
  - If identities can be impersonated, the entire authorization system is undermined.

Individual vs. Shared Identity

- **Individual identity:** One human corresponds to one system identity.
  - Alice logs in as alice.
  - Bob logs in as bob.
  - Actions can be traced back to specific people.
- **Shared identity:** Multiple humans use the same system identity.
  - Five system administrators all log in as admin.
  - Actions can be traced to the shared account but not to specific individuals.

Groups

- Managing permissions for individual users doesn't scale.
- Imagine a company with 500 employees and 1,000 files.
  - If you had to specify permissions for each user on each file, you'd have 500,000 permission entries to manage.
  - When someone joins the company, you'd have to update potentially thousands of entries.

Groups

- **Groups** solve this by creating named **collections** of users.
  - Instead of saying "**Alice**, **Bob**, **Carol**, **Dave**, and **Eve**" can read the financial reports, you say members of the **finance group** can read the financial reports.
  - When **Frank** joins the **finance team**, you add him to the **finance group** once, and he automatically inherits access to everything the group can access.

Groups

- A **group** represents a trust decision:
  - "Everyone in this group is trusted with the same level of access to certain resources."
- This is powerful but requires careful thought:
  - Adding someone to a group grants them access to *everything* that group can access
  - Groups often have access to more resources
  - "Temporary" group membership tends to become permanent
  - Group membership is often not audited regularly

Permissions

- A **permission** is a recorded rule that says:
  - "This identity can perform this action on this object."
- The three components:
  - **Subject:** Who is acting (user or group)
  - **Object:** What is being acted upon (file, directory, device, etc.)
  - **Action:** What operation is being performed (read, write, execute, delete, etc.)
- Permissions are not suggestions. They are enforcement mechanisms. The operating system kernel checks permissions on every access attempt and blocks unauthorized actions.

Permissions

- When you set permissions on a file, you're making a policy decision:
- **I have decided that:**
  - The owner should be able to read and modify this file
  - Team members should be able to read it but not modify it
  - Everyone else should not be able to access it at all
- The *`chmod` command doesn't make policy decisions, you do.
  - The command just records your decision so the system can enforce it.

*chmod = change mode

Access Control Models

- Different systems implement access control differently.
- **Discretionary Access Control (DAC)**
  - In **DAC**, the **owner** of a resource **decides** who can access it.
  - If you create a file, you control its permissions. You can make it readable by everyone or lock it down so only you can access it.
- Most operating systems (Linux, Windows, macOS) use DAC as their primary model.

Access Control Models

- **Mandatory Access Control (MAC)**
  - In **MAC**, a **central authority** defines security policy, and users cannot override it.
  - Even if you own a file, you might not be able to share it if policy forbids it.
- **Role-Based Access Control (RBAC)**
  - In **RBAC**, permissions are assigned to **roles**, and users are assigned to roles. (**assigned by system administrators**)
  - A "**Database Administrator**" role might have permissions to manage databases.
  - Anyone assigned that role gets those permissions.

How Linux Represents Users

- Every user on a Linux system has an entry in `/etc/passwd`.
  - Is a **user database.**
  - Despite the name, this file doesn't contain passwords (those are in `/etc/shadow`).
- It contains user metadata:
  - `username:x:UID:GID:comment:home_directory:shell`
- Example:
  - `alice:x:1001:1001:AliceSmith:/home/alice:/bin/bash`

How Linux Represents Users

- **Breaking this down:**
  - **alice:** username (the human-readable identity)
  - **x:** password is in /etc/shadow
  - **1001:** UID (user ID, the numeric identity the kernel uses)
  - **1001:** GID (primary group ID)
  - **Alice Smith:** user Human-readable info like comment/description/#Tel
  - **/home/alice:** home directory
  - **/bin/bash:** default shell

How Linux Represents Users

- **UIDs: The Real Identity**
  - Internally, Linux doesn't use usernames, it uses numeric User IDs (UIDs).
  - The username is just a convenience for humans. When you run `ls -l` and see "alice", the system looked up UID 1001 and translated it to "alice" for display.
- **Special UIDs:**
  - **UID 0** — root, the superuser, has unrestricted access
  - **UIDs 1-999** — typically reserved for system accounts (services, daemons)
  - **UIDs 1000+** — typically regular users

Managing Users

- # Create a new user
  - `sudo useradd -m -s /bin/bash newuser`
- # Set password
  - `sudo passwd newuser`
- # Delete a user
  - `sudo userdel newuser`
- # View current user
  - `whoami`
- # View current user's UID and groups
  - `id`

Managing Groups

- # Create a new group
  - `sudo groupadd mygroup`
- # Delete a group sudo
  - `groupdel mygroup`
- # Add user to a supplementary group (IMPORTANT: -a means "append")
  - `sudo usermod -aG groupname username`

Primary and Supplementary Groups

- **Primary group (login group)**
  - Every user has **exactly one primary group**.
  - Stored in /etc/passwd.
  - Assigned at login.
- **Supplementary (secondary) groups**
  - Users can belong to **multiple extra groups**.
  - Stored in /etc/group.
  - Used for shared access.
  - Checked after owner, before others.

Primary and Supplementary Groups

- If you forget `-a` you will remove the user from all their other groups.
- # Alice is in group: developers
  - `sudo usermod -G newgroup alice`
    - # Alice is now ONLY in group: newgroup (developers removed!)
  - `sudo usermod -aG newgroup alice`
    - # Alice is now in groups: developers, newgroup

File Permissions

- Every file and directory in Linux has:
  - Owning user (one user)
  - Owning group (one group)
  - Permission bits for three scopes: **owner, group, and others**
- When a process tries to access a file, the kernel checks:
  1. Is the process's effective UID the file's owner?
     - Use owner permissions.
  2. Is the process's effective GID (or any supplementary GID) the file's
     - Use group permissions
  3. Otherwise
     - Use "other" permissions

File Permissions

- This is checked **in order**.
  - If you're the owner, group permissions don't matter (even if they're more permissive).
- Each scope (owner, group, other) has three permission bits
- This creates 9 bits total, displayed as:
  - rwxrwxrwx (e.g., r-x = 101)
- Example: rwxr-x---
  - **Owner:** read, write, execute
  - **Group:** read, execute (no write)
  - **Other:** nothing

File vs. Directory

- **For files:**
  - **r:** Can read the file's data (cat, less, cp)
  - **w:** Can modify the file's data (echo >>, vim, truncate)
  - **x:** Can execute the file (run it as a program)
- **For directories:**
  - **r:** Can list the directory's contents (ls)
  - **w:** Can modify the directory's contents (create, delete, rename files)
  - **x:** Can traverse the directory (pass through it to access contents)

Directory Permission Combinations

Permissions	What You Can Do
`---`	Nothing
`--x`	Enter directory, access files IF you know their names (can't list)
`-w-`	Nothing useful (need x to actually modify entries)
`-wx`	Enter and create/delete files, but can't see what's there
`r--`	List filenames only, can't access file contents or enter
`r-x`	Enter and list, but can't create/delete files
`rw-`	List filenames, still can't enter (rarely useful)
`rwx`	Full access

Octal Notation

- **The Numeric Shorthand**
  - Instead of writing rwxr-xr--, you can use octal (base-8) numbers.
- Each permission has a value:
  - Read (r) = 4 (100)
  - Write (w) = 2 (010)
  - Execute (x) = 1 (001)

Desired	Calculation	Octal
`rwx`	4 + 2 + 1	7
`rw-`	4 + 2 + 0	6
`r-x`	4 + 0 + 1	5
`r--`	4 + 0 + 0	4
`-wx`	0 + 2 + 1	3
`-w-`	0 + 2 + 0	2
`--x`	0 + 0 + 1	1
`---`	0 + 0 + 0	0

Octal Notation

- Then combine three digits for owner, group, other:
  - **755 = rwxr-xr-x**
    - owner: full, group: read/execute, other: read/execute
  - **644 = rw-r--r–**
    - owner: read/write, group: read, other: read
  - **700 = rwx------**
    - owner: full, everyone else: nothing

Common Permission Patterns

Octal	String	Typical Use
755	rwxr-xr-x	Executable files, directories others need to access
644	rw-r--r--	Regular files (documents, configs)
750	rwxr-x---	Directories for a specific group
640	rw-r-----	Sensitive files group can read
700	rwx------	Private directories
600	rw-------	Private sensitive files
777	rwxrwxrwx	Everything for everyone (usually a mistake)

Using `chmod`

- # Octal notation
  - `chmod 644 myfile`
- # Symbolic notation - who (+/- permission)
  - `chmod u+x myfile` # Add execute for owner
  - `chmod g-w myfile` # Remove write for group
  - `chmod o=r myfile` # Set other to read only
  - `chmod a+r myfile` # Add read for all (a = all)
  - `chmod ug+rw myfile` # Add read/write for owner and group

Ownership

- Viewing Ownership
  - `ls -l myfile`
  - rw-r--r-- 1 **alice developers** 4096 Jan 15 10:30 myfile
- Changing Ownership
  - `sudo chown bob myfile` # Change owner
  - `sudo chgrp developers myfile` # Change group
  - `sudo chown bob:developers myfile` # Change both at once
- Only root can change file ownership.
  - Regular users can only change group ownership, and only to groups they belong to.

The Testing Mindset

- You cannot reliably determine effective permissions just by reading configuration.
- You must test.
  - The user might not be in the group you think
  - ACLs might override basic permissions
  - The path might be blocked at a parent directory
  - Your mental model of Linux permissions might be wrong
- The only way to know what a user can actually do is to **try it as that user**.

Testing as Another User

- Use `sudo -u username` to run commands as another user:
  - # Can bob read this file?
    - `sudo -u bob cat /path/to/file`
  - # Can bob list this directory?
    - `sudo -u bob ls /path/to/directory/`
  - # Can bob create files here?
    - `sudo -u bob touch /path/to/directory/testfile`

Lab

- Lab introduces how **Linux permissions actually control access.**
- Create a test user and group, build a small directory and file hierarchy with carefully chosen permission modes, and then **verify access from another user's point of view.**
- Experimenting with reading, writing, and traversing directories, you will translate numeric permissions into clear, plain-English access statements (for example, who can read a file but cannot modify it).

Lab

- Focus is the **“mystery directory”**, which demonstrates that having execute permission on a directory allows access to files by name without allowing directory listing.
- By the end of the lab, you will understand how directory permissions differ from file permissions in practice.