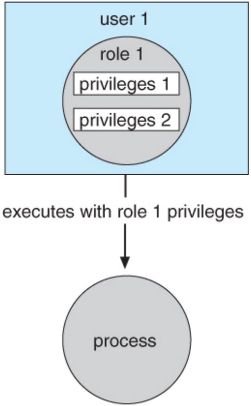


RBAC (Solaris10)

CSL 301

OPERATING SYSTEMS

Lecture 29

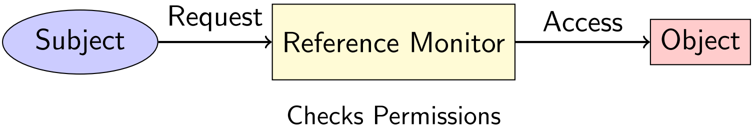
Access Control in Operating Systems

Instructor
Dr. Dhiman Saha

Image Source: Operating System Concepts (Book)

What is Access Control?

- **Goal**: Ensure that resources are used only by authorized parties in authorized ways.
- **Key Components**:
  - **Subjects**: The entities attempting to perform actions (e.g., users, processes).
  - **Objects**: The resources being accessed (e.g., files, devices, memory segments).
  - **Rights**: The allowed actions (e.g., read, write, execute).



The Access Control Matrix

- A conceptual model describing the protection state of a system.
- Rows represent **Subjects** (Domains).
- Columns represent **Objects**.
- Cells contain the **Access Rights**.

Objects

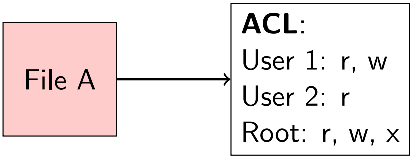
Subjects

	File A	**File B**	**Printer**
User 1	r, w	r	-
User 2	r	-	print
Root	r, w, x	r, w	print

- **Problem**: Sparse and large. Hard to store efficiently as a 2D array.

Implementation 1: Access Control Lists (ACLs)

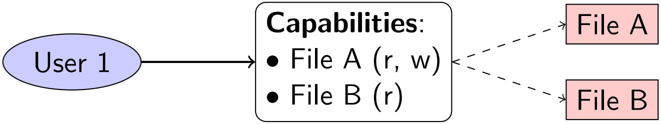
- Store the matrix by **column** (per object).
- Each object has a list of subjects and their rights.
- **Example**: UNIX file permissions (simplified).



- **Pros**: Easy to answer "Who can access this object?".
- **Cons**: Hard to answer "What can this user access?".

Implementation 2: Capabilities

- Store the matrix by **row** (per subject).
- Each subject holds a list of "tickets" or "keys" for objects.
- Possession of the capability grants access.



- **Pros**: Efficient for the subject; easy delegation.
- **Cons**: Revocation is difficult (how to take back the key?).

ACLs vs. Capabilities

Feature	ACLs	Capabilities
Analogy	Guest List	Ticket / Key
Storage	With Object	With Subject
Revocation	Easy (remove from list)	Hard (invalidate key)
Delegation	Hard (update list)	Easy (pass the key)
Least Privilege	Harder to enforce	Easier (give only needed ke

Access Control Models

Discretionary Access Control (DAC)

- The owner of the object decides who has access.
- Standard UNIX permissions.
- Flexible but prone to errors (e.g., Confused Deputy).

Mandatory Access Control (MAC)

- System enforces policy based on security labels (e.g., Top Secret).
- Users cannot override the policy.
- Used in high-security environments (e.g., SELinux).

Role-Based Access Control (RBAC)

- Permissions assigned to **Roles** (e.g., Manager, Admin).
- Users assigned to Roles.
- Simplifies management in large organizations.

The Name Space Problem

- **Single Computer**: Easy. If a name is in use, reject new assignment.
  - E.g., `/etc/passwd` is the same file for everyone on the system.
- **Distributed Systems**: Harder.
  - Multiple computers, potentially different domains.
  - How to ensure a name (e.g., user "remzi") means the same thing everywhere?
  - How to prevent name collisions (e.g., two users creating "project_x")?

Solutions to Name Space Problem

- **Don't Bother**: Accept that namespaces are different (e.g., PIDs).
- **Central Authority**: Require approval for name selection (e.g., AFS filenames).
- **Partitioning**: Assign portions of namespace to participants (e.g., DNS, IPv4).

The Android Challenge

- **Context**: Mobile devices differ from servers/desktops.
- **Single User, Many Apps**:
  - Apps from many authors (some potentially malicious).
  - No need to protect users from each other, but protect the *user* from the *apps*.
- **Goal**: Least Privilege.
  - Apps need some privileges to function (e.g., Internet, Contacts).
  - Shouldn't have full user privileges.

Android's Solution: UIDs and Permissions

- **Unique UIDs**:
  - Each app is assigned a unique Linux User ID (UID) at install time.
  - Uses standard Linux file permissions to sandbox apps.
  - App A cannot read App B's files.
- **Permissions**:
  - Developers declare required permissions (e.g., `READ_CONTACTS`).
  - Users grant permissions at install time or runtime.

Permission Labels (MAC)

- **Permission Labels**:
  - A form of Mandatory Access Control (MAC).
  - Labels define what an app can access and what it exposes.
  - Acts like a capability: possession of the label grants access.
- **Pros and Cons**:
  - **Pro**: Fine-grained control.
  - **Con**: User fatigue. Users often blindly grant permissions to get the app to work.

Privilege Escalation: A Double-Edged Sword

- **Utility**:
  - Allows users to temporarily gain higher privileges for specific tasks.
  - Mechanisms: `setuid`, `sudo`.
  - Essential for system administration and controlled access.
- **Danger**:
  - If a privileged program is compromised, the attacker gains those privileges.
  - "Privilege Escalation Considered Dangerous".

The Attacker's Playbook

1. **Gain a Foothold**:
   - Compromise a low-privilege application (e.g., a web server).
   - Access is limited (cannot change system files).
2. **Escalate Privileges**:
   - Look for bugs in `setuid` programs or misconfigurations.
   - Exploit these to gain **root** access.
3. **Total Control**:
   - Once root, the attacker owns the system.
   - Can install backdoors, steal data, or destroy the OS.

Real-World Examples

- **UNIX/Linux**:
  - Primarily ACL-based (Owner/Group/Other bits).
  - `rwx` for User, Group, Others.
  - `setuid`: Allows a program to run with the owner's ID (temporary privilege escalation).
  - `sudo`: Execute a command as another user (usually root).
- **Android**:
  - Uses UIDs for application sandboxing.
  - Permissions requested at install/runtime (Capability-like feel).

Summary

- Access Control is essential for OS security.
- **Reference Monitor** mediates all accesses.
- **ACLs** are object-centric (Guest List).
- **Capabilities** are subject-centric (Keys).
- Modern systems often use a hybrid or add layers like RBAC and MAC (SELinux, AppArmor) for better security.