## Lesson 1: The Foundation of Protection and the Principle of Least Privilege

In the architectural design of a modern operating system, **protection** refers to the specific internal mechanisms that control how processes or users access resources, whereas **security** is the broader defense of the system against external threats like viruses, worms, and identity theft.

The fundamental goal of any protection model is to treat the entire computer as a collection of diverse hardware and software objects, where every single object possesses a unique name and can only be interacted with through a strictly defined set of valid operations. By viewing the system this way, the operating system's primary protection problem becomes ensuring that every object is accessed **correctly and exclusively** by those processes that have been **explicitly** granted permission to do so.

To manage these permissions effectively, systems first distinguish between different users by assigning unique **User Identities (User IDs)** and **Group Identifiers (Group IDs)**, which are then associated with every file and process created by that user to determine access rights. In situations where a user requires more power for a specific task, a mechanism called **privilege escalation** allows them to temporarily change their effective ID to one with greater rights.

The most vital guiding rule for any protection system is the **Principle of Least Privilege**, which dictates that every program, user, and system should be granted only the minimum level of privilege necessary to complete its assigned task. This strategy is essential because it limits the potential damage that can occur if a system component contains a bug or is intentionally abused by a malicious actor, effectively containing the "blast radius" of any failure.

This management can be implemented in two ways:

- **Rough-grained:** It is characterized as being **easier and simpler** to implement, but it applies the principle of least privilege in **large chunks**. The specific example given is the traditional UNIX model, where a process either has the rights of a regular user or the total power of a "root" administrator.
- **Fine-grained:** This is noted as being **more complex** and having **more overhead**, but it provides much better **protection** by using detailed lists or roles to specify exact permissions for every individual resource. Examples include using detailed File **Access-Control Lists (ACLs)** or **Role-Based Access Control (RBAC)**.

**Visualization Aid: Protection vs. Security**

| Feature | Protection (Internal) | Security (External/Internal Defense) |
|---|---|---|
| **Primary Goal** | Controlling access to OS resources by processes and users. | Defending the system against active attacks and malicious software. |
| **Core Examples** | File permissions, memory protection, and access matrix rules. | Preventing denial-of-service attacks, viruses, and identity theft. |
| **Focus** | Ensuring only "allowed" operations happen on "specific" objects. | Protecting the overall integrity and availability of the system. |

## Lesson 2: Protection Domains and UNIX Implementation

A **protection domain** is a defined environment that specifies the resources a process may access at any given time, acting as a collection of permissions. To understand how these are built, we look at the specific rules and how systems like UNIX handle moving between them.

- **Access-Right and Domain Structure**: Every single permission within a domain is called an **access-right**, which is a pair consisting of an **object-name** and a **rights-set**. The rights-set is a subset of all valid operations (like read, write, or execute) that can be performed on that specific object, and the domain is simply the total set of these access-rights.
- **Domain Entities and Overlap**: Depending on the system, a domain can be associated with a **user**, a **process**, or even a **procedure**. It is important to note that domains can overlap, meaning two different domains might share the same access-right for a specific object, such as both having the right to use a particular printer.
- **Static vs. Dynamic Protection**: Protection can be **static**, where the access rights remain unchanged throughout the life of a process, or **dynamic**, which allows for **domain switching** or **privilege escalation** as a process's "need to know" changes.
- **UNIX Implementation of Domains**: In UNIX systems, a domain is typically associated with a **user-id**, and the system provides three distinct ways to switch between domains in which the primary difference is how the domain switch is triggered and how long it lasts:
    - **The setuid bit**: This is a property of the **file itself**. When a file with this bit turned "on" is executed, the process automatically adopts the identity and privileges of the file's owner rather than the user who ran it. Once the program finishes execution, the user's ID is immediately reset to its original value.
    - **The su command**: This is a method for switching domains via **passwords**. When a user executes `su`, they are attempting to enter another user's domain entirely. This requires providing the password of the **target domain** (the user

they want to become). Unlike `setuid`, this switch remains in effect for the entire session until the user explicitly exits that domain.

- **The sudo command**: This is a command prefix used to execute a **single, specific task** in another domain. Instead of switching the whole environment like `su`, it only applies to the command that follows it. To use it, the original domain must already have the necessary privilege, or the user must provide a password (typically their own, depending on system configuration).

In summary: `setuid` is triggered by a file, `su` switches your entire session via the target's password, and `sudo` executes just one command with elevated rights.

**Visualization: Domain Components and UNIX Switching**

| | Definition | UNIX Example |
|---|---|---|
| **Access-Right:** | A pair: <Object, Operations> | <File1, {read, write}> |
| **Domain:** | A set of multiple access-rights | The rights linked to a User-ID |
| **Switching:** | Moving from one domain to another | Using **sudo** to run a root command |

## Lesson 3: The Access Matrix Model

The **Access Matrix** is the fundamental theoretical model used to represent a system's protection state. It is designed to be a rigorous way of tracking which "subjects" (domains) have which permissions over which "objects" (resources).

- **Matrix Structure**:
  - **Rows** represent the different **protection domains** ($D_1$,$D_2$, . . ., $D_n$)
  - **Columns** represent every **object** in the system ($F_1$, $F_2$, printers, etc.), including the domains themselves.
  - **Entries**: Each cell, denoted as $Access(i, j)$, contains the specific set of operations (like read, write, execute) that a process in Domain $i$ is allowed to perform on Object $j$

To understand the Access Matrix fully, you must grasp a key architectural concept: while **rows** represent the domains (who has access) and **columns** represent the objects (what is being accessed), **a domain can also be an object**. This means a column in the matrix can actually be another domain, allowing one domain to have "rights" over another.

Here is how the special rights are categorized based on what they target:

**Rights Targeting Normal Objects (Files, Printers, etc.)**

- **Owner**: If a domain is the "owner" of an object, it has the authority to define or change the entire access column for that specific object.
- **Copy (denoted by `*`)**: If a right has an asterisk (e.g., `read*`), a process in that domain can copy that specific right to any other domain for that object.

**Rights Targeting Domain Objects (Making a Domain the "Object")**

When these rights are used, the target domain is treated as an object in its own column:

- **Control**: This right allows Domain $i$ to modify the permissions in the **row** of Domain $i$. This is only possible because Domain $j$ is acting as an **object** for Domain $i$ to manipulate.
- **Transfer (Switch)**: This allows a process to "switch" from its current Domain $i$ into Domain $j$. By doing this, the process leaves its original row and begins using the permissions of the target domain's row.

**Mechanism vs. Policy**

A major design strength of the access matrix is the strict separation of these two concepts:

- **Mechanism**: The **Operating System** provides the matrix and the rules. It ensures the matrix is only manipulated by authorised agents and that all access rules are strictly enforced.
- **Policy**: The **User** dictates the policy, deciding exactly who can access what object and in what mode (read, write, etc.).

**Visualization: Access Matrix with Special Rights**

| | File 1 | File 2 | Domain 2 |
|---|---|---|---|
| **Domain 1** | read* | owner | transfer (switch) |
| **Domain 2** | - | read, write | - |
| **Domain 3** | execute | - | control |

*In this matrix, Domain 1 exercises rights over normal objects like File 1 (copying its "read" right) and File 2 (ownership), while also treating Domain 2 as an object by switching into it. Meanwhile, Domain 3 targets Domain 2 as an object, using its "control" right to modify any permission in Domain 2's row.*

## Lesson 4: Implementation Strategies for the Access Matrix

In a real computer system, the access matrix is almost always **sparse**, meaning most of the cells are empty because most users only have access to a small fraction of the total objects. Storing this as a literal 2D grid in memory would be extremely wasteful, so operating systems use one of four primary strategies to implement it. First, you have to understand that:

**The User** (via their ID) is assigned a **Domain** (a list of rights), which the system checks whenever a process tries to access an **Object** (like a file).

### Option 1: The Global Table

This is a single master list containing triples in the format: `<domain, object, rights-set>`.

- **How it works:** Whenever a process tries to perform an action on an object, the operating system searches this entire giant table to see if that specific user-object-action combo exists.
- **Pros:** It is the simplest method to design and understand.
- **Cons:** The table becomes so massive that it **won't fit in the main memory**, which makes the system very slow. It also makes it hard to group things; for example, if you want to make a file "read-only" for everyone, you have to add a separate entry for every single user in the system.

### Option 2: Access-Control Lists (ACLs)

Instead of one big table, every object (like a file) has its own small list of `<domain, rights-set>` pairs.

- **How it works:** When you try to open a file, the system looks only at the list attached to that specific file to see if your name is on it.
- **Pros:** It matches how users think about security ("Who is allowed to see my file?"). It also allows for a **default set**, where the owner can say "anyone not on this list can only read".
- **Cons:** It is very slow because the system must check the list **every single time** you access the file. It is also hard to see all the rights for one specific user because their name is scattered across thousands of different files.

### Option 3: Capability Lists

This flips the ACL approach by being domain-based. Each domain carries a list of objects (like files) and their associated rights-sets (what can be done to them).

- **How it works:** Access is granted because the user holds a "capability"—a secure ticket or "secure pointer"—that proves to the system they have permission to access the resource. These capabilities are protected objects maintained by the OS and are never directly accessible by the domain itself
- **Pros:** It is very fast and excellent for keeping a specific user's permissions all in one place. "localizing information"
- **Cons: Revocation (cancelling access) is a nightmare**. Because the tickets are spread out in the pockets of many different users, the system has no easy way to find and "delete" a specific permission once it's been handed out.

### Option 4: Lock-Key

This uses unique bit patterns. Every object has a "lock" and every domain has a "key".

- **How it works:** A user can only use an object if their domain has a key that matches one of the object's locks.
- **Pros:** It is very flexible because keys can be passed between users easily. Cancelling access is simple: the owner just changes the lock on the object.
- **Cons:** It is complicated for the system to manage all the different bit patterns and decide who is allowed to change them.

### The Modern Hybrid Approach

Most systems today use a mix of **ACLs and Capabilities**.

- **How it works:** The very first time you try to open a file, the system does a slow check of the **Access-Control List**. If you are allowed in, the system gives you a temporary **Capability** (ticket) to hold onto. For the rest of your session, you just show that ticket to do your work quickly without the system having to check the slow list again. Once the process finishes (e.g., closes the file), the capability is destroyed.

**Visualization: Comparison of Implementation Trade-offs**

| Method | Storage Style | Best For... | Main Weakness |
|---|---|---|---|
| **Global Table** | Single Table | Simple systems | Too large for memory |
| **Access List** | By Object (Column) | User-centric security | Slow per-access checks |
| **Capability List** | By Domain (Row) | High performance | Hard to revoke rights |
| **Lock-Key** | Bit Patterns | Flexibility/Passing rights | Managing bit patterns |

## Lesson 5: Real-World Access Control and Revocation

This lesson explores how protection rules are applied in actual systems like Solaris and Java, and how an operating system takes away permissions once they have been granted.

**Real-World Examples: Solaris and Java**

- **Solaris 10 (Role-Based Access Control - RBAC):** Instead of giving power directly to users, Solaris uses "roles". A **privilege** is the specific right to run a system call, and these are assigned to **roles** rather than individuals. A user can assume a role by providing a password, which temporarily grants their process all the privileges linked to that role.
- **Java 2 Protection Model:** In Java, protection is managed entirely by the **Java Virtual Machine (JVM)**. Every class is assigned a **protection domain** as soon as it is loaded, which tells the JVM exactly what that class is allowed to do.
- **Stack Inspection:** If a Java program calls a method that needs special power, the JVM performs "stack inspection". It looks back through every previous step in the current process (the stack) to make sure every library or method involved has the right to perform that action.

**Revocation: Taking Back Access Rights**

Revocation is the process of removing a domain's access rights to an object, and it can be handled in several ways: immediate or delayed, selective or general, partial or total, and temporary or permanent. Here is how revocation works for each of the four access matrix strategies, along with the specific schemes used for capability lists.

- **Global Table:** Revocation is performed by searching the single master table for the specific triple `<domain, object, rights-set>` and deleting it. While conceptually simple, this process can be very slow because the table is often too large to fit in main memory, making the search for a specific entry inefficient.
- **Access-Control Lists (ACLs):** This is considered a simple and efficient method for revocation because the system only needs to find the specific list attached to the object and remove the entry for the domain. This method is immediate and allows for precise control, meaning it can be selective (removing one person) or general (removing everyone), and either partial or total.
- **Lock-Key:** This method is highly flexible and provides easy revocation. Access is granted only if a domain's bit pattern (key) matches one of the object's unique bit patterns (locks). To revoke access, the owner simply creates a new "lock" or bit pattern for the object, which instantly makes all existing keys in circulation invalid.
- **Capability Lists:** Revoking rights is more complex in this strategy than just deleting a name from an access list. Because the capabilities are spread across various domains, it requires a complex scheme just to locate them before they can be cancelled. Since these are "secure pointers" managed by the OS and hidden from the domain, the system must use specialized tracking methods to revoke them effectively. These methods are:
    1. **Reacquisition:** The operating system periodically deletes all capabilities from every domain in the system. If a process needs to continue using an object, it must "reacquire" the capability from the system; if the access has been revoked, the system simply denies the new request.
    2. **Back-pointers:** Each object maintains a list of pointers that track every capability currently referring to it (a method used in the Multics system). When revocation is necessary, the system follows these "back-pointers" to find and delete every instance of that specific capability wherever it exists in the system.
    3. **Indirection:** Capabilities point to a specific entry in a **global table**, which then points to the actual object. To revoke access, the system deletes the entry in the global table; this is an immediate "general" revocation, meaning it blocks access for every user who was using that global entry, regardless of whether they were specifically targeted.

4. **Keys:** Every capability is assigned a unique bit pattern (a key) when it is created, and this must match a "master key" associated with the object. To revoke access, the system generates a **new master key** for the object, which instantly invalidates all old capabilities because their bit patterns no longer match. A policy decision determines who is allowed to create and modify these keys, such as the object owner or other authorized users.

**Breakdown:**

- There are two global tables you must differentiate between:

- **Global Table (Access Matrix Implementation):** This is a single, massive master list for the entire system. It stores every permission as a triple: `<domain, object, rights-set>`. When a process tries to do something, the OS searches this one giant table to see if the action is allowed.
- **Global Table (Indirection in Capability Lists):** This is a specialized "middleman" table used specifically to make revoking capabilities easier. Instead of a capability pointing directly to a file, it points to an entry in this table, which then points to the file. To revoke access, the system deletes that specific entry in the table, instantly blocking everyone who was using it.

In short: the first one is the **entire database of rules**, while the second is a **redirection tool** used to cancel "tickets" (capabilities) more efficiently.

- The distinction between these two lies in **what they are used for** within the protection system:

- **Lock-Key (Implementation Strategy)**: This is one of the four basic ways to **store** the access matrix. In this model, an object can have a *list* of multiple bit patterns (locks), and a domain can have a *list* of multiple bit patterns (keys). Access is granted if any key from the domain's list matches any lock from the object's list. It is a general strategy for managing permissions across the whole system.
- **Keys (Capability Revocation Scheme)**: This is a specific **revocation tool** used *only* within the Capability List strategy to solve the "revocation nightmare". In this scheme, each individual capability (the "ticket") is assigned a unique bit pattern (a key) when it is created. This key must match a single **master key** associated with the object. To revoke access for *everyone* at once, the system simply generates a new master key for the object, making all circulating "tickets" instantly useless.

**In short:** Lock-Key is a way to **organize** the entire permission grid using lists of patterns, while the "Keys" scheme is a specialized **kill-switch** used to cancel distributed capabilities by changing one master value.