Reading:
See [Gollmann] Chapter 5 & 7
See [PF] Chapter 2.2 (pages 72 – 85)
See [Andersen] Chapter 4 up to 4.2.4
Read Wiki http://en.wikipedia.org/wiki/File_system_permissions

Lecture 6: Access Control

6.1 Overview of layering model in computer system design
6.2 Access control model
6.3 Access control matrix
6.4 Intermediate control
6.5 Access control in UNIX/Linux
6.6* UNIX/Linux: Elevated privilege (controlled invocation)

* **Warning**: This part could be the most abstract and *complicated* notion in the module.
Complexity is bad for security. See https://www.schneier.com/news/archives/2012/12/complexity_the_worst.html.

6.1 Overview of Layering Model in Computer System Design

Layers in Computer System

Applications	(e.g. browser, mail reader)
Services	(e.g. DBMS, Java Virtual Machine)
Operating System	(e.g. UNIX/Linux, Windows, iOS, macOS)
OS Kernel	(including system calls to handle memory, manage virtual memory, etc.)
Hardware	(including CPU, memory, storage, I/O)

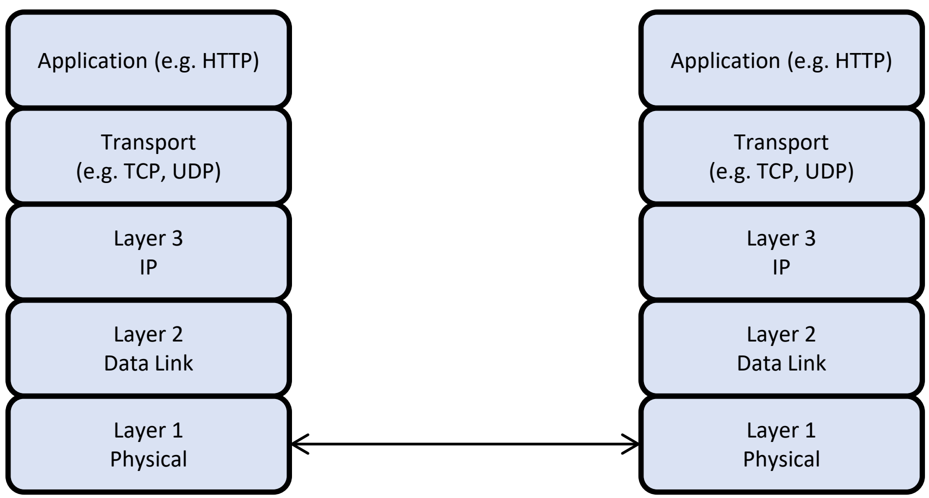
Remarks:

1. We can view OS kernel as part of the OS.
2. These layers are used as a *guideline*.
   Actual systems typically don't have distinct layers
   (for example, a windowing system may span across multiple "layers").

Compared to Layers in Communication Network

Network:

- The **boundary** is more well defined
- Information/data **flows** from the topmost layer down to the lowest layer, and is transmitted from the lowest layer to the topmost layer
- A **concern** of data confidentiality and integrity



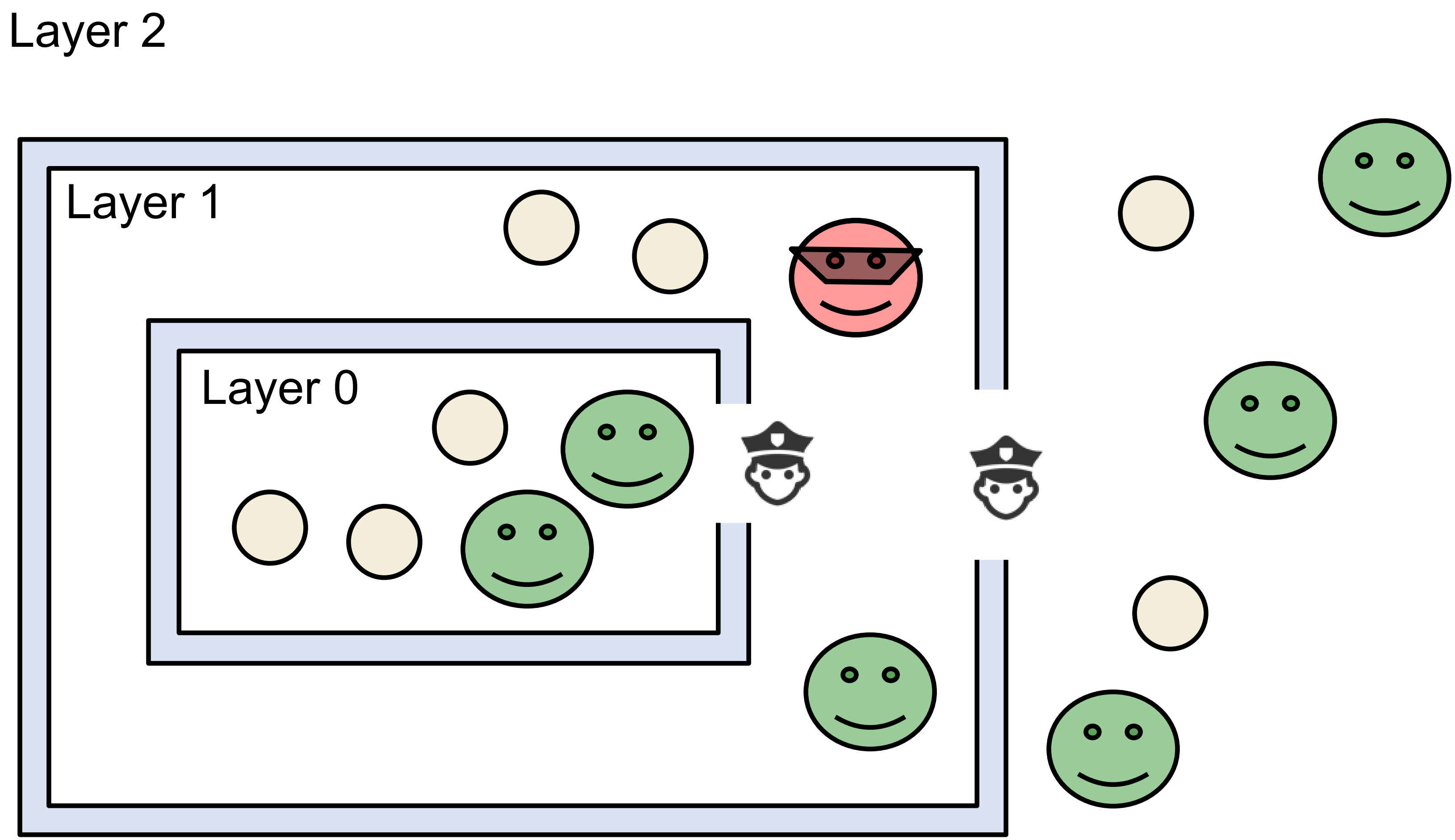
Compared to Layers in Communication Network

System:

- The **boundary** is *less* well-defined
- Every layer has its own "**processes**" and "**data**"
(although ultimately, the raw processes/data are handled by hardware)
- The main security **concern** is about access to the processes & memory/storage
- Hence, besides data confidentiality & integrity (e.g. password file), there is also a concern of **process "integrity"**:
whether it deviates from its original execution path

Applications	(e.g. browser, mail reader)
Services	(e.g. DBMS, Java Virtual Machine)
Operating System	(e.g. UNIX/Linux, Windows, iOS, macOS)
OS Kernel	(including system calls to handle memory, manage virtual memory, etc.)
Hardware	(including CPU, memory, storage, I/O)

Attackers and System's Layers



- Suppose an attacker sneaks into Layer 1
- The attacker must *not* be able to directly manipulate objects/processes in (more privileged) Layer 0
- Note that it is very difficult to ensure this requirement
(due to possible implementation errors, overlooks in the design, etc.)

Security Mechanism

- It is insightful to figure out at *what layer* a security mechanism/attack resides
- A (layered-based) *security mechanism* should have a well-defined ***security perimeter/boundary***:
  - See [Gollmann] Section 3.5: "The parts of the system that can malfunction without compromising the protection mechanism *lie outside* this perimeter. The parts of the system that can be used to disable the protection mechanism *lie within* this perimeter".
- Nevertheless, quite often, it is difficult to determine the boundary
- An important **design consideration** of the security mechanism is: on *how to prevent attacker from* getting access to a layer inside the boundary

Security Mechanism

  - For example: **SQL injection attacks** target at the SQL DBMS. The **OS password management** (which is in a layer below) *should remain intact* even if an SQL injection attack has been successfully carried out.

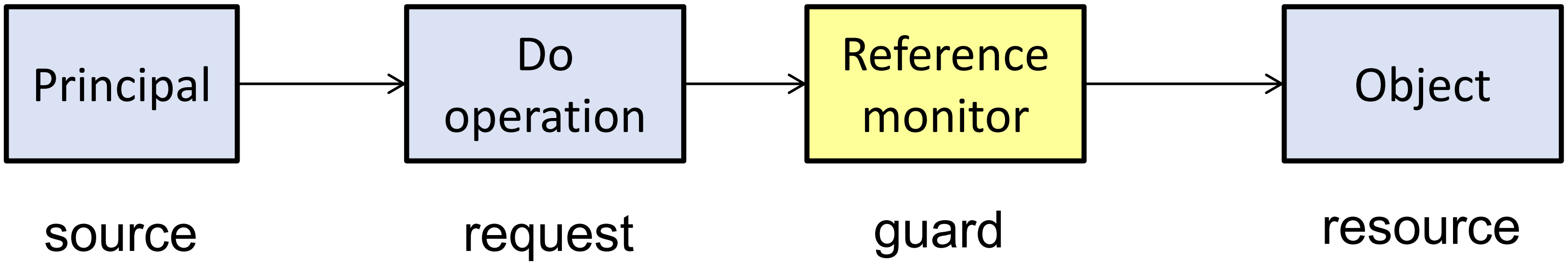
- It is also possible that an application takes care of *its own security* (i.e. self-secure itself):
  - For example: if an application **always encrypt** its data before writing them to the file system, even if the access control of the file system is compromised (e.g. malicious user can read someone else files), the *confidentiality* of the data will still be preserved.

6.2 Access Control Model

Why Access Control?

- Access control is required in computer system, information system, and physical system
- Different application domains have different requirements /interpretation
- Generally, **access control** is about: "*selective restriction of access to a place or other resource*" (Wiki)
- The access control system/model gives a way to **specify and enforce** such restriction on the subjects, objects and actions
- Examples of application domains:
    - OS
    - Social media (e.g. Facebook)
    - Documents in an organization (document can be classified as "restricted", "confidential", "secret", etc.)
    - Physical access to different part of the building

Principal/Subject, Operation, Object



- A *principal* (or *subject*) wants to access an *object* with some *operation*
- The *reference monitor* either grants or denies the access
- Examples:
  - LumiNUS: a ***student*** wants to ***submit*** a ***forum post***
  - LumiNUS: a ***student*** wants to ***read*** the ***grade of another student***
  - File system: a ***user*** wants to ***delete*** a ***file***
  - File system: a ***user Alice*** wants to ***change*** a ***file's mode*** so that it can be ***read*** by ***another user named Bob***

Principal/Subject, Operation, Object

Principals vs subjects:

- **Principals**: the human users
- **Subjects**: the entities in the system that operate on behalf of the principals, e.g. processes, requests

Accesses to objects can be classified to the following:

- **Observe:** e.g. reading a file
  (In LumiNUS: downloading a file from File/Workbin)
- **Alter**: e.g. writing a file, deleting a file, changing properties
  (In LumiNUS: uploading a file to the File/Workbin)
- **Action:** e.g. executing a program

Object Access Rights and Ownership

Who decides the access rights to an object?

There are two approaches:

1. ***Discretionary Access Control (DAC)***:
   the **owner** of the object decides the rights
2. ***Mandatory Access Control (MAC)***:
   a **system-wide policy** decides the rights, which must be followed by *everyone* in the system must follow

6.3 Access Control Matrix

Access Control Matrix (See [PF] pg. 79)

- How do we **specify the access rights** of a particular principal to a particular object? We can use a table called ***access control matrix***

	my.c	mysh.sh	sudo	a.txt
root	{r,w}	{r,x}	{r,s,o}	{r,w}
Alice	{r,w}	{r,x,o}	{r,s}	{r,w,o}
Bob	{r,w,o}	{}	{r,s}	{}

Note: r: read, w: write, x: execute, s: execute as owner, o: owner

- Although the above access control matrix can specify the access rights for **all pairs** of principals and objects, the table can be **very large**, and is thus **difficult to manage**
- Hence, it is often treated as an **abstract concept** only, and seldom explicitly deployed (see the next slide)

Access Control List (ACL) and Capabilities

- The access control matrix can be represented in two different ways: ***Access Control List (ACL)*** or ***capabilities***
- **Access Control List (ACL):**
  stores the *access rights* to a **particular object** as a list
- **Capabilities:**
  - **A subject** is given a list of *capabilities*,
    where each capability is the access rights to an object
  - "*A capability*: is an unforgeable token that gives the possessor certain rights to an object"
    (see [PG] pg. 82 on the description of "capability")

(**Question**: Does UNIX file system adopt ACL or capabilities?)

ACL and Capabilities: Examples

- **Access control matrix**:

	my.c	mysh.sh	sudo	a.txt
root	{r,w}	{r,x}	{r,s,o}	{r,w}
Alice	{}	{r,x,o}	{r,s}	{r,w,o}
Bob	{r,w,o}	{}	{r,s}	{}

- **ACL**: each object has a list of access rights to it

my.c	→ (root, {r,w}) → (Bob, {r,w,o})
mysh.sh	→ (root, {r,x}) → (Alice, {r,x,o})
sudo	→ (root, {r,s,o}) → (Alice, {r,s}) → (Bob, {r,s})
a.txt	→ (root, {r,w}) → (root, {r,w,o})

- **Capabilities**: each subject has a list of capabilities to objects

root	→ (my.c, {r,w}) → (mysh.sh, {r,x}) → (sudo, {r,s,o}) → (a.txt, {r,w})
Alice	→ (mysh.sh, {r,x,o})→ (sudo, {r,s}) → (a.txt, {r,w,o})
Bob	→ (my.c, {r,w,o}) → (sudo, {r,s})

Drawbacks of ACL and Capabilities

- Drawback of ACL:
  - It is difficult to get an overview of the objects that **a particular subject** has access rights to
  - I.e. it is hard to answer: “which objects can a *particular subject* access?”
  - As an illustration: in UNIX, suppose the system admin wants to generate the list of file that the user **`alice012`** has “r” access to. How to quickly generate this list?
- Drawbacks of capabilities:
  - It is difficult to get an overview of the subjects who have access rights to a **particular object**
  - I.e. it is hard to answer: “which subjects can access a *particular object*?”

Drawbacks of ACL and Capabilities

- Drawbacks of **both** methods:
  - The size of the lists can still to be **too large** to manage
  - Hence, we need some ways to **simplify** the representation
  - One simple method is to "*group*" the subjects/objects and define the access rights on **the defined groups**
  - We need an ***intermediate control***

6.4 Intermediate Control

Intermediate Control: Group

In Unix file permission, the ACL specifies the rights for the:

- **owner** (user)
- **group**
- **world (others)**

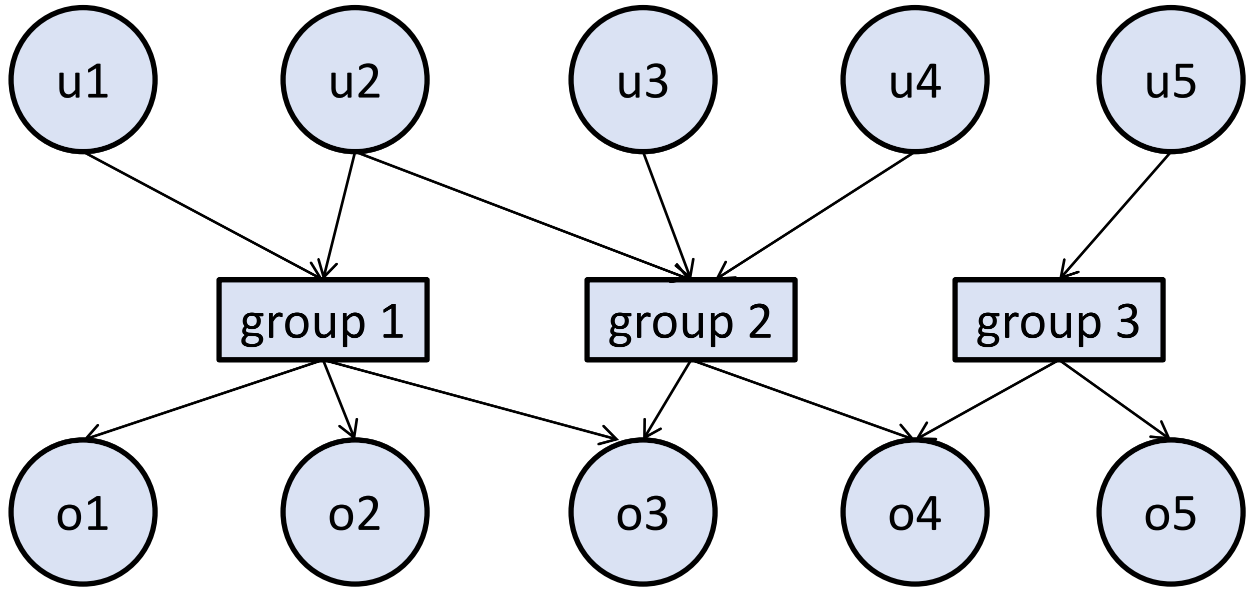
- Subjects in a same group: have the same access rights
- Some systems demand that a subject is in a single group, but some systems don't put such a restriction

Question: Is it possible in UNIX that an owner **does not** has a read access, but others have?

Answer: Strangely, yes. See Alice's file `temp` below:

```
--w-r--r--   1 alice   staff        3 Mar 13 00:27 temp
```

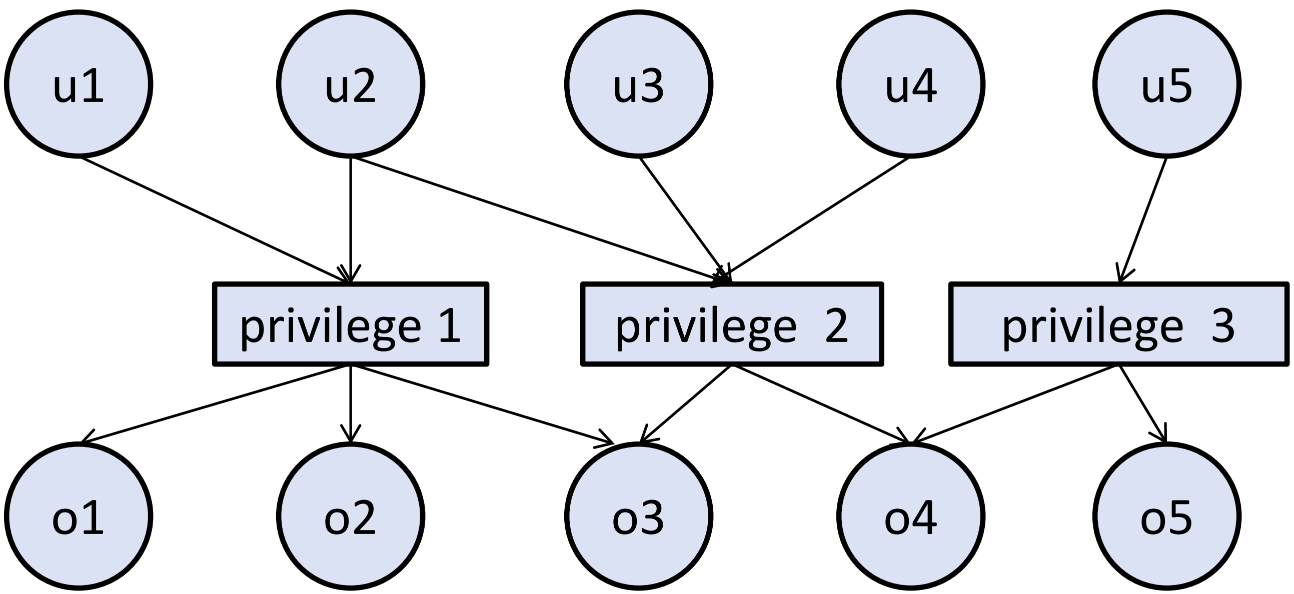
Users and Groups: Illustration



- In LumiNUS, *project groups* can be created.
  Objects created in a group can only be read by members of the group + *the lecturer(s)*.
- In UNIX, groups can only be created by root.
  The groups information is stored in the file `/etc/group`.

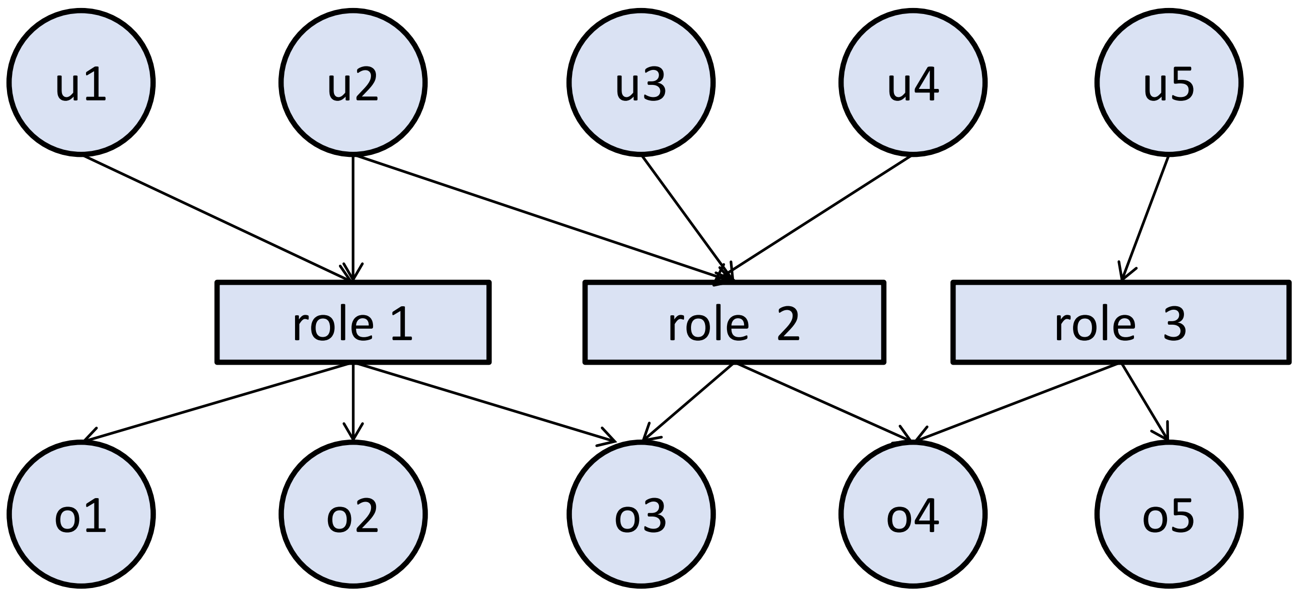
Intermediate Control: Privileges

- We often use the term ***privilege*** for:
  the access right to execute a process
- Privilege can also be viewed as an intermediate control



Intermediate Control: Role-based Access Control (RBAC)

- The grouping can be determined by the "***role***" of the subjects
- A **role** associates with a **collection of procedures**
- In order to carry out these procedures, **access rights** to certain objects are required

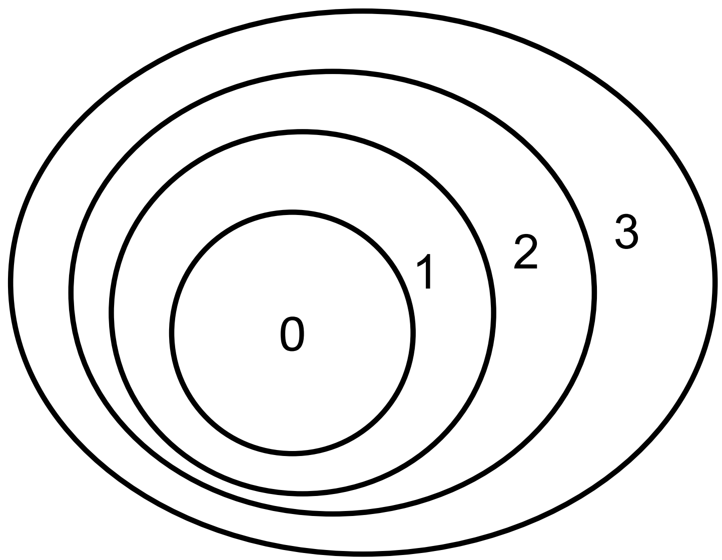


Intermediate Control: Role-based Access Control (RBAC)

- If a subject is assigned a particular role, the access rights to the objects can be determined based on the ***least privilege principle***
- The **least privilege principle**: only access rights that are **required** to complete the role will be assigned
- Examples on **LumiNUS's gradebook**:
  - The roles of a TA include entering the grade for all students. So the TA also **has** a "**write**" access on the grades.
  - Should we also give a TA an access right to ***change names***? This right is "irrelevant" as it is not related to the roles of the TA.
  - Since this change-names access right is **not** required for the TA to complete his/her tasks, by the least privilege principle**,** the TA is **not** given the access right.

Intermediate Control: Protection Rings

- Here, each object (data) and subject (process) are assigned a **number**:
  - If a process is assigned a number *i*,
    we say that the process runs **in ring *i***
  - Objects with **smaller number** are **more important**
  - Very often, we call processes with lower ring number as having "***higher privilege***"



Intermediate Control: Protection Rings

- Whether a subject can access an object is determined by their respective its **assigned number**:
  - A subject *cannot* access (i.e. both read/write) an object with smaller ring number
  - It can do so if its privilege gets "escalated"
- Some remarks:
  - UNIX has only **2** rings: **superuser** and **user**
  - We can also view this as a special case of RBAC in the sense that the ring number is the "role"

Examples of Intermediate Control: Bell-LaPadula vs Biba

- In protection rings, the subjects can access objects that are classified with **the same** or **lower** privilege.
  Are there reasonable *alternatives*?
- Here are two well-known models**:** ***Bell-LaPadula*** and ***Biba***: although they are rarely implemented as-it-is in computer system, they serve as a good guideline.
- In both models, objects and subjects are divided into **linear levels**, such as: level 0, level 1, level 2, ...
- **Higher level** corresponds to higher "security" (the opposite of protection rings).
  Example: security clearance: unclassified < confidential < secret < top secret.
- ***Multilevel security***: information with incompatible classifications (i.e. at different security levels).

Bell-LaPadula Model (for Confidentiality)

Read https://en.wikipedia.org/wiki/Bell%E2%80%93LaPadula_model.

Focus on **confidentiality**

Restrictions imposed by the Bell-LaPadula model:

- **No read up**:
  - A subject has only ***read access*** to objects whose security level is **below** the subject's current clearance level
  - This **prevents** a subject from getting access to information available in security levels **higher than** its current clearance level

level 3
level 2
level 1
level 0

level 3
level 2
level 1
level 0

Bell-LaPadula Model (for Confidentiality)

- **No write down**:
  - A subject has ***append access*** to objects whose security level is **higher** than its current clearance level
  - This **prevents** a subject from passing information to levels **lower than** its current level
  - Example: in order to prevent **information leakage**, a clerk working in the *highly-classified department* should not gossip with other staff from *lower security-level department*

Potential issues:

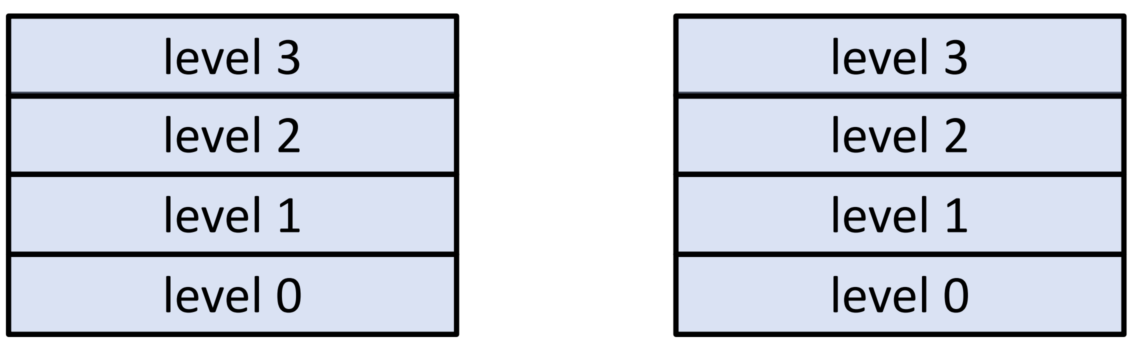
- Notice that a subject can append to objects at higher security level. Is there a concern of **integrity**?
  Is it possible that appending to an object would **distort** its original content?

Biba Model (for Integrity)

Focus on **Integrity**

Restrictions imposed by the Biba Model:

- **No write up**:
  - A subject has only ***write access*** to objects whose security level is **below** the subject's current clearance level.
  - This **prevents** a subject from compromising the integrity of objects with security levels **higher than** its current clearance level.



Biba Model (for Integrity)

- **No read down**:
  - A subject has ***only read*** access to objects whose security level is **higher** than its current clearance level
  - This **prevents** a subject from reading forged information from levels **lower than** its current level

Remark:

- In a model that imposed *both* Biba and Bell-LaPadula models, subjects then can only read/write to objects **in the same level**

6.4 Access Control in UNIX/Linux

Notes:

- An easy way to get UNIX-like environment on Windows is **Cygwin** (https://www.cygwin.com).
- Another way is by installing a **hypervisor** or **virtual machine monitor** (VMM): **VirtualBox** (https://www.virtualbox.org), or **VMWare** Player/Workstation (https://www.vmware.com). Then install Linux (e.g. **Ubuntu desktop**).
  Note: For VirtualBox, perform these additional installation steps as well:
  install VirtualBox Extension Pack and Guest Additions.
- Yet, another method: **Bash shell in Window 10**.
  (https://www.howtogeek.com/249966/how-to-install-and-use-the-linux-bash-shell-on-windows-10/).

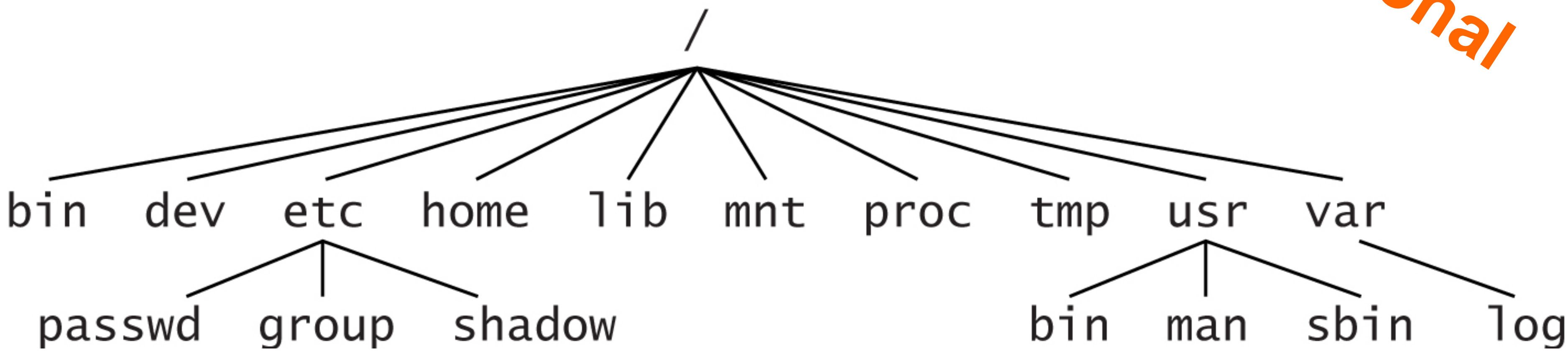
UNIX/Linux: Some Background

Optional

- History from 1970s
- Many versions:
  Solaris, AIX, Linux, Android, OS X + iOS
- Linux is open source (`http://www.kernel.org`)
- Many available tools (usually also open source)
- Many Linux distributions (distros):
  - Vary in setup, administration, kernel.
  - A popular choice: Ubuntu desktop

UNIX/Linux File System Structure

Optional



`/etc/passwd`: user database, password file

`/etc/shadow`: user database containing hashed user passwords, shadow password file

`/etc/group`: group database

`/bin/ls`

`man`

UNIX/Linux Manual Pages

Optional

- UNIX documentation using the **man** command
  - man is your friend!
  - Note: small variations in man with different UNIX

```
$ man ls
$ man man
```

- Organized in sections

```
$ man printf
$ man 1 printf
$ man 3 printf
```

- A free good resource to learning Linux commands: W. Shotts, "*The Linux Command Line*", `http://linuxcommand.org`

UNIX/Linux Access Control

- In Unix, **objects** of access control include:
  files, directories, memory devices, and I/O devices.
- All these resources are treated as **file**!
  (a notion of "***universality of I/O***", read also tis Wiki:
  http://en.wikipedia.org/wiki/File_system_permissions).

```
%ls -al
-r-s--x--x  1 root   wheel   164560 Sep 10  2014 sudo
-rwxr-xr-x  2 root   wheel    18608 Nov  7 06:32 sum
-rw-r--r--  1 alice  staff      124 Mar  9 22:29 myprog.c
lr-xr-xr-x  1 root   wheel        0 Mar 12 16:29 stdin
```

Question: What are the files in the directory `/dev`?

UNIX/Linux User and Groups

- Each **user**:
  - Has a unique **user/login name**
  - Has a numeric user identifier (**UID**): stored in `/etc/passwd`
  - Can belong to one or more groups:
    the **first group** is stored in `/etc/passwd`,
    additional group(s) are stored in `/etc/group`
- Each **group**:
  - Has a unique **group name**
  - Has a numeric group identifier (**GID**)
- Main purposes of UIDs and GIDs:
  - To determine **ownership** of various system resources
  - To determine the **credentials** of running processes
  - To control the **permissions** granted to processes
    that wish to access the resources *(more on this later)*

UNIX/Linux Principals & Subjects

- **Principals**: *user identities* (UIDs) and *group identities* (GIDs)
- Information of the user accounts are stored in the **password file** `/etc/passwd`

  E.g. `root:*:0:0:System Administrator:/var/root:/bin/sh`

  (Read Wiki page for details of these fields: https://en.wikipedia.org/wiki/Passwd)
- A special user is the *superuser*, with UID 0, and usually the username `root`:
  - All security checks are turned off for `root`.
    (Recall that UNIX's protection rings consists of only 2 rings: superuser, users).
- **Subjects**: *processes*.
  Each process has a process ID (PID): use the command `ps aux` or `ps -ef` to display a list of running processes

Remarks on the Password File Protection

- The file is made **world readable** because some information in `/etc/passwd` is needed by **non-root processes**
- Note that in the **older versions** of UNIX, the location of "*" was the hashed password $H(pw)$.
  As a result, all users can have access to this hashed-password field.
- The availability of the hashed password allows attackers to carry out an **offline password guessing**.
- Since passwords are typically short, exhaustive search are able to get many passwords.
- To prevent this, it is now replaced by "*", and the actual password is stored somewhere else, and it is **not** world-readable.
  The actual location depends on different versions of UNIX (e.g. the **shadow password file** `/etc/shadow`).

The Shadow Password File

The fields of an entry (separated by “:”):

- login name, hashed password, date of last password change, minimum password age, maximum password age, password warning period, password inactivity period, account expiration date, reserved field
- Example:

  user1:6yonrs//S$bUdht9fglwJW0LduAxEJpcExtMfKokFMJoT8tGkKLx5
  xFGJk22/trPstOHXr4PdBlD0AV1xko5LfFVDwW.aJS.:17275:0:99999:7:::

The **second** field (**hashed password**), which is shown in red, has the following format: idsalt$hashed-key

- **id**: ID of the hash-method used (5=SHA-256, 6= SHA-512, …)
- **salt**: up to 16 chars drawn from the set [a-zA-Z0-9./]
- **hashed-key**: hash of the password (e.g. 43 chars for SHA-256, 86 chars for SHA-512)

File System Permission

indicates whether it is a file (-) or directory (d)

owner's group

date & time of last modification

```
-rw-r--r--  1 alice  staff  124 Mar  9 22:29 my.c
```

file permission

owner

file size

filename

link count (not relevant in this module)

The **file permission** are grouped **into 3 triples**, that define the **read**, **write**, **execute** access for: **owner** (**"user"**), **group**, and **others** (**the "world"**):

'-' indicates access not granted

r: read

w: write (including delete)

x: execute (**s**: allow a user to execute with the permission of the *file owner*)

Changing File Permission Bits

- Use **`chmod`** command:
  `chmod` [options] mode[,mode] file1 [file2 ...]
- Useful options:
  - `-R`: Recursive, i.e. include objects in subdirectories
  - `-f`: force processing to continue if errors occur
  - `-v`: verbose, show objects changed (unchanged objects are not shown)
- Two notations for mode:
  - *Symbolic mode* notation
  - *Octal mode* notation
- See also : https://en.wikipedia.org/wiki/Chmod

Changing File Permission Bits

- **Symbolic mode** notation:
  - **Syntax**: [references][operator][modes]
  - *Reference*: u (user), g (group), o (others), a (all)
  - *Operator*: + (add), - (remove), = (set)
  - *Mode*:
    r (read), w (write), x (execute), s (setuid/gid), t (sticky)
  - Examples:
    `chmod g+w shared_dir`
    `chmod ug=rw groupAgreements.txt`
  - What are the file permission bits of `shared_dir` and `groupAgreements.txt`?

    `shared_dir: drwxr-xr-x` → `drwxrwxr-x`

Changing File Permission Bits

- **Octal mode** notation:
  - 3 or 4 octal digits
  - 3 rightmost digits refer to permissions for: the file **user**, the **group**, and **others**
  - Optional leading digit, when 4 digits are given, specifies: **the special file permissions** (setuid, setgid and sticky bit)
  - Examples:
    `chmod 664 sharedFile`
    `chmod 4755 setCtrls.sh`
  - `-rw-rw-r-` and `-rwsr-xr-x`

Octal	Binary	rwx
7	111	rwx
6	110	rw-
5	101	r-x
4	100	r--
3	011	-wx
2	010	-w-
1	001	--x
0	000	---

File System Permission: Additional Notes

- ***Directory*** permissions:
  - **r**: allows the contents of the directory to be **listed** if the x attribute is also set
  - **w**: allows files within the directory to be created, deleted, or renamed if the x attribute is also set
  - **x**: allows a directory to be **entered** (i.e. `cd dir`)
- ***Special file*** permissions:
  - **Set-UID**: the process' *effective user ID* of is the *owner* of the executable file (usually root), rather than the user running the executable

    ```
    -r-sr-sr-x   3 root     sys        104580 Sep
    16 12:02 /usr/bin/passwd
    ```

File System Permission: Other Notes

- **Set-GID**: the process' *effective group ID* is the group owner of the executable file
  `-r-sr-sr-x   3 root      sys          104580 Sep 16 12:02 /usr/bin/passwd`
- **Sticky bit**:
  If a **directory** has the sticky bit set,
  its file can be deleted *only by* the owner of the file,
  the owner of the directory, or by root.
  This prevents a user from deleting other users' files from public directories such as `/tmp`:
  `drwxrwxrwt 7  root   sys    400 Sep  3 13:37 tmp`
- See also: https://docs.oracle.com/cd/E19683-01/816-4883/secfile-69/index.html

Objects and Access control

- Recall that the objects are **files**.
- Each file is owned by a **user** and a **group**.
  Also recall that each file is associated with a **9-bit permission**.
- When a **non-root user** (subject) wants to access a file (object), the following are checked in order:
  1. If the user is the owner, the permission bits for **owner** decide the access rights
  2. If the user is not the owner, but the user's group (GID) owns the file, the permission bits for **group** decide the access rights
  3. If the user is not the owner, nor member of the group that own the file, then the permission bits for **other** decide
- The **owner** of a file or **superuser** can change the permission bits

Selected Issues: Search Path

- When a user types in the command to execute a program, say "`su`" without specifying the full path, which program would be executed:

  `/usr/bin/su` or `./su`?

- The program to be executed is searched through the directories specified in the ***search path.***
  Use a command `echo $PATH` to display the search path.

- When a program with the name is found in a directory, the search **stops** and the program will be **executed**.

- Suppose an attacker somehow stored a malicious program in the directory that appears in the **beginning** of the search path, and the malicious program has a **common name**, say "`su`".
  When a user executes "`su`", the **malicious program** will be invoked instead.

- To prevent such attack, specify the **full path**.

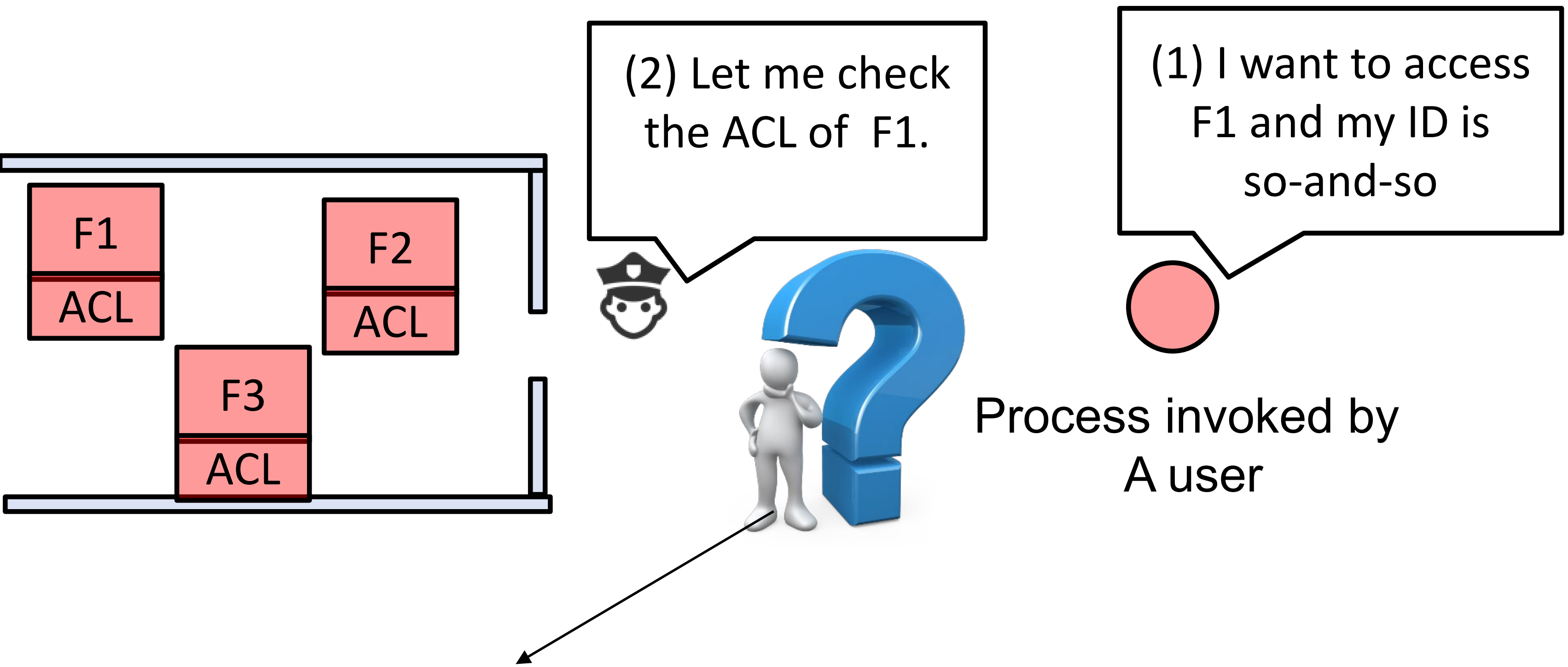
- Also **avoid** putting the current directory ("`.`") in the search path. *Why*?

Controlled Invocation (More on this in next section)

- Certain resources in UNIX/Linux can **only** be accessed by superuser: e.g. listening at the trusted port 0—1023, accessing `/etc/shadow` file
- Sometimes, a user **needs** those resources for certain operation: e.g. changing password
- It is **not** advisable to change the user status to superuser
- A **solution** is ***controlled invocation***: the OS provides *a predefined set of operations (programs) in a **superuser mode***, and the user can then invoke those operations with the superuser mode
- An example file:
  `-rws—x--x  3 root  bin  59808 Nov 17 07:21 /usr/bin/passwd`
  The "`s`" (set-UID) bit indicates that the privilege is escalated while a user is executing this process

6.6 UNIX/Linux: Privilege Escalation (Controlled Invocation)

Access Control and Reference Monitor



See Slide 48 again for the checking rules

Process and Set User ID (Set-UID)

- A process is a subject: has an identification (PID)
- A new process can be created by executing a file or due to a "fork" by an existing process
- A process is associated with ***process credentials***: a *real UID* and an *effective UID*
- The **real UID** is inherited from the user who invokes the process. It identifies the **real owner** of the process. E.g. if the user is `alice`, then the process' real UID is `alice`.*
- For processes created by executing a file, there are 2 cases:
  - If the set-User-ID (set-UID) is disabled (the permission bit is displayed as "**x**"), then the process' **effective UID** is same as **real UID**
  - If the set-User-ID (set-UID) is enabled (the permission bit is displayed as "**s**"), then the process' **effective UID** is inherited from the UID of the **file's owner**

Real UID and Effective UID: Examples

- If `alice` creates the process by executing the file:
  `-r-`**`x`**`r-xr-x   1 root   staff      6 Mar 18 08:00 check`

  Then the process' real UID is `alice`,

  whereas its effective UID is `alice`

- If the process is created by executing the following file:
  `-r-`**`s`**`r-xr-x   1 root   staff      6 Mar 18 08:00 check1`

  Then the process' real UID is `alice`,

  but its effective UID is `root`

This indicates that set-UID is enabled

When a Process (Subject) Wants to Read a File (Object)

- When a process wants to access a file, the **effective UID** of the process is treated as the "**subject**" and checked against the file permission to decide whether it will be granted or denied access

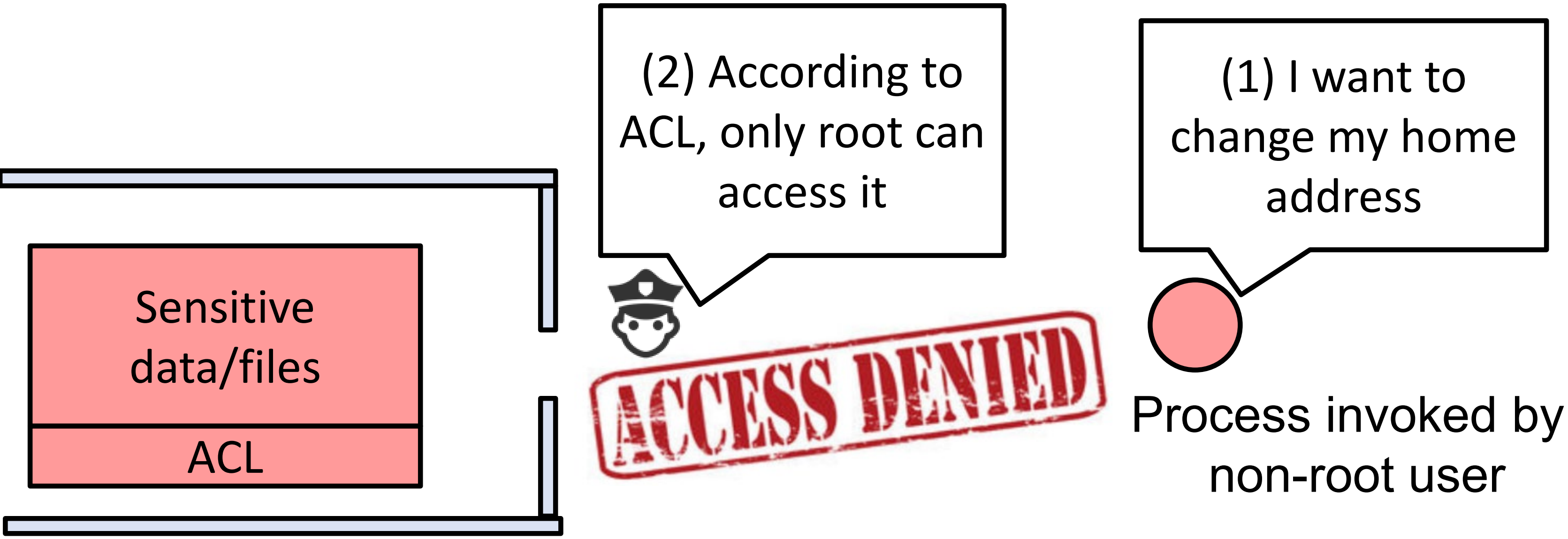
- Example:
  Consider a file owned by the root:
  `-rw-------   1 root   staff      6 Mar 18 08:00 sensitive.txt`

  - If the effective UID of a process is **`alice`**, then the process will be **denied** reading the file
  - If the effective UID of a process is **`root`**, then the process will be **allowed** to read the file

Use Case Scenario of “s” (Set-UID)

- Consider a scenario where the file `employee.txt` contains personal information of the users
- This is **sensitive** information, hence, the system administrator set it to **non-readable** except by root:
  `-rw-------   1 root  staff     6 Mar 18 08:00 employee.txt`
- However, users should be allowed to **self-view** and even **self-edit** some fields (for e.g. postal address) of their own profile
- Since the file permissible is set to “-” for all users (except root), a **process** created by any user (except root) **cannot** read/write it
- Now, we are stuck: there are data in the file that we want to **protect**, and data that we want the user to **access**
- *What can we do?*

Access Control and Reference Monitor



Solution

- Create an executable file `editprofile` owned by `root`:

  ```
  -r-sr-xr-x   1 root   staff       6 Mar 18 08:00 editprofile
  ```

- The program is made **world-executable**:
  any user can execute it

- Furthermore, the **set-UID bit** is set ("s"):
  when it is executed, its effective UID will be "root"

- This is an example of a ***set-UID-root*** program/executable

- Now, if `alice` executes the file, the process' real UID is `alice`, but its effective UID is `root`:
  this process now **can read/write** the file `employee.txt`

Sensitive data/files
ACL
(2) Ok
ACCESS GRANTED
(1) My effective UID is root
Process invoked by non-root user, but it is a **set-UID-root** process

Comparison: When Set-UID is *Disabled*

- If the user `alice` invokes the executable, the process will has its **effective ID** as **`alice`**
- When this process wants to read the file `employee.txt`, the OS (reference monitor) will **deny** the access

Process info: name (`editprofile`) real ID (`alice`) effective ID (`alice`)


```
-rw-------   1 root  staff     6 Mar 18 08:00 employee.txt
-r-xr-xr-x   1 root  staff     6 Mar 18 08:00 editprofile
```

Comparison: When Set-UID is *Enabled*

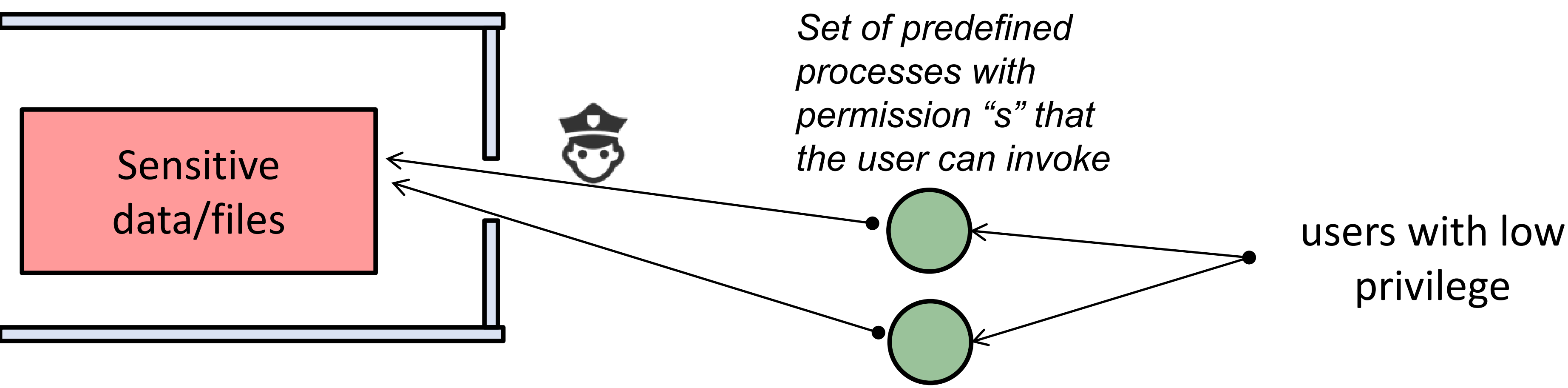
- But if the permission of the executable is "s" instead of "x", then the invoked process will has **`root`** as its effective ID
- Hence the OS **grants** the process to read the file
- Now, the process invoked by `alice` can access `employee.txt`

Process info: name (`editprofile`) real ID (`alice`) effective ID (`root`)


```
-rw-------   1 root  staff     6 Mar 18 08:00 employee.txt
-r-sr-xr-x   1 root  staff     6 Mar 18 08:00 editprofile
```

Elevated Privilege

- In this example, the process `editprofile` is temporarily **elevated** to superuser (i.e. root), so as to access the sensitive data
- We can view the elevated process as the **interfaces** where a user can access the "sensitive" information:
  - They are the predefined "**bridges**" for the user to access the data
  - Note that the "bridge" can only be built by **the root**
- So, it is important that these "bridges" are correctly implemented and do not leak more than required!



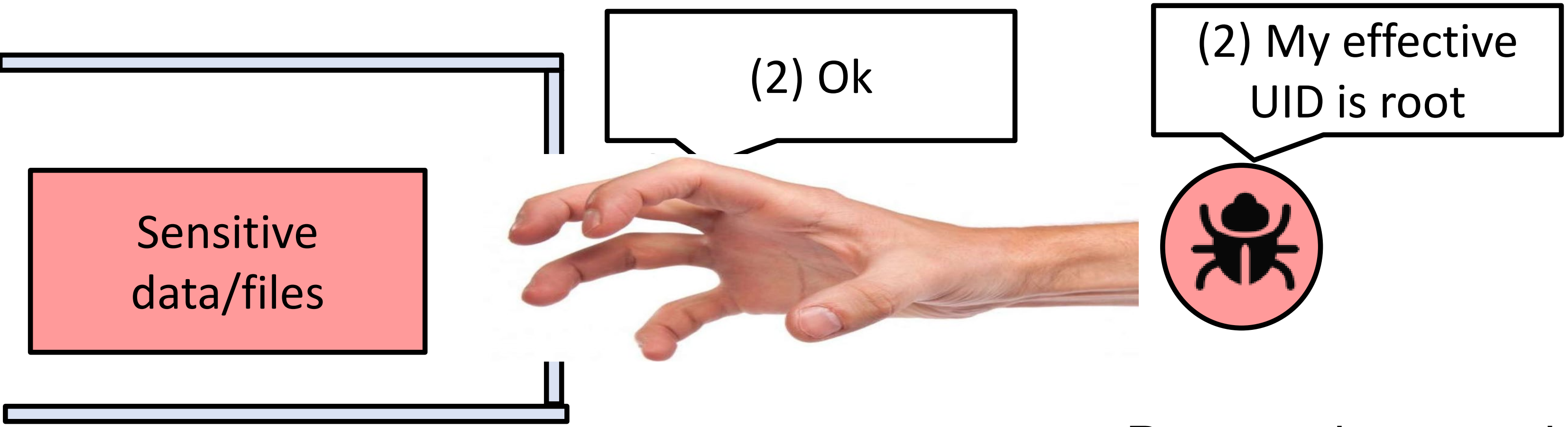
Privilege Escalation Go Wrong

- Suppose a "bridge" is *not* implemented correctly, and contains **exploitable vulnerability**
- An attacker, by feeding the bridge with a carefully-crafted input, can cause it to perform **malicious operations**
- This could have serious implication, since the process is now running with "***elevated privilege***"
- Attacks of such form also known as "***privilege escalation***" attacks

  (Read https://en.wikipedia.org/wiki/Privilege_escalation on privilege escalation:

  Privilege escalation *is the act of exploiting a bug, design flaw or configuration oversight in an operating system or software application to gain elevated access to resources that are normally protected from an application or user. The result is that an application with more privileges than intended by the application developer or system administrator can perform unauthorized actions.*)

- This leads us to another important security topic: **secure programming** and **software security**



Program has permission “s”, and the program has a ”bug”!

Footnote: More Complications (Real UID, Effective UID, Saved UID)

- The OS actually maintains three IDs for a process: real UID, effective UID, and **saved UID**
- **Saved UID** is like a "temp" container and is useful for a process running with elevated privileges to **drop privilege temporarily**: a good set-UID programming practice
- A process removes its privileged user ID from its effective UID, but stores it in its saved UID.
  Later, the process may restore privilege by restoring the saved privileged UID into its effective UID.
  (See https://en.wikipedia.org/wiki/User_identifier#Saved_user_ID)
- The details may easily **confuse** many programmers
  (Read http://stackoverflow.com/questions/8499296/realuid-saved-uid-effective-uid-whats-going-on)
- Different UNIX versions may have **different** behaviors: complexity is bad for security!
  (Optional: Chen et al., "Setuid Demystified", USENIX Security, 2002)