

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 7: Access Control

7.1 Overview of layering model in computer system design

7.2 Access control

7.3 Access control representation

7.4 Intermediate control

7.5 Access control in UNIX/Linux

7.6 Controlled invocation & privilege elevation

7.7* Controlled Invocation in UNIX/Linux

* **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.

7.1 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)

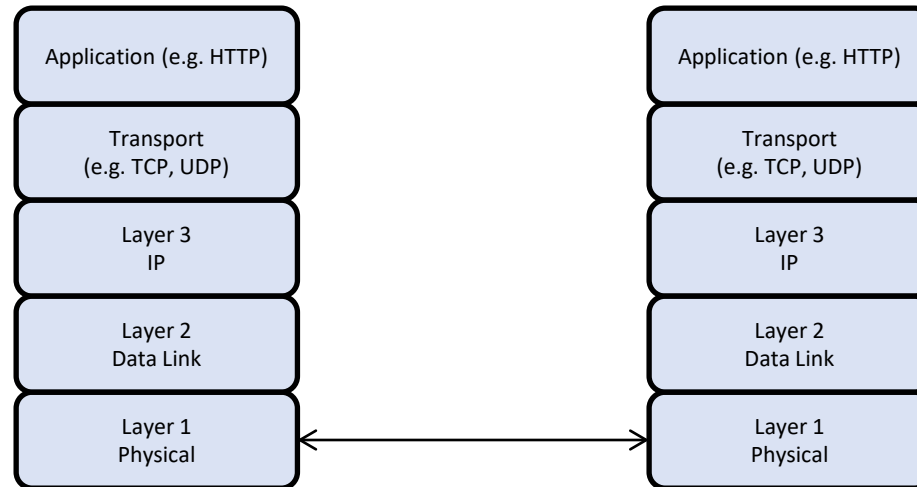
Remarks:

1. We can also view **OS kernel** as part of the **OS**
2. These layers are used as a *guideline*:
actual systems typically don't have **distinct layers**
(e.g., 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

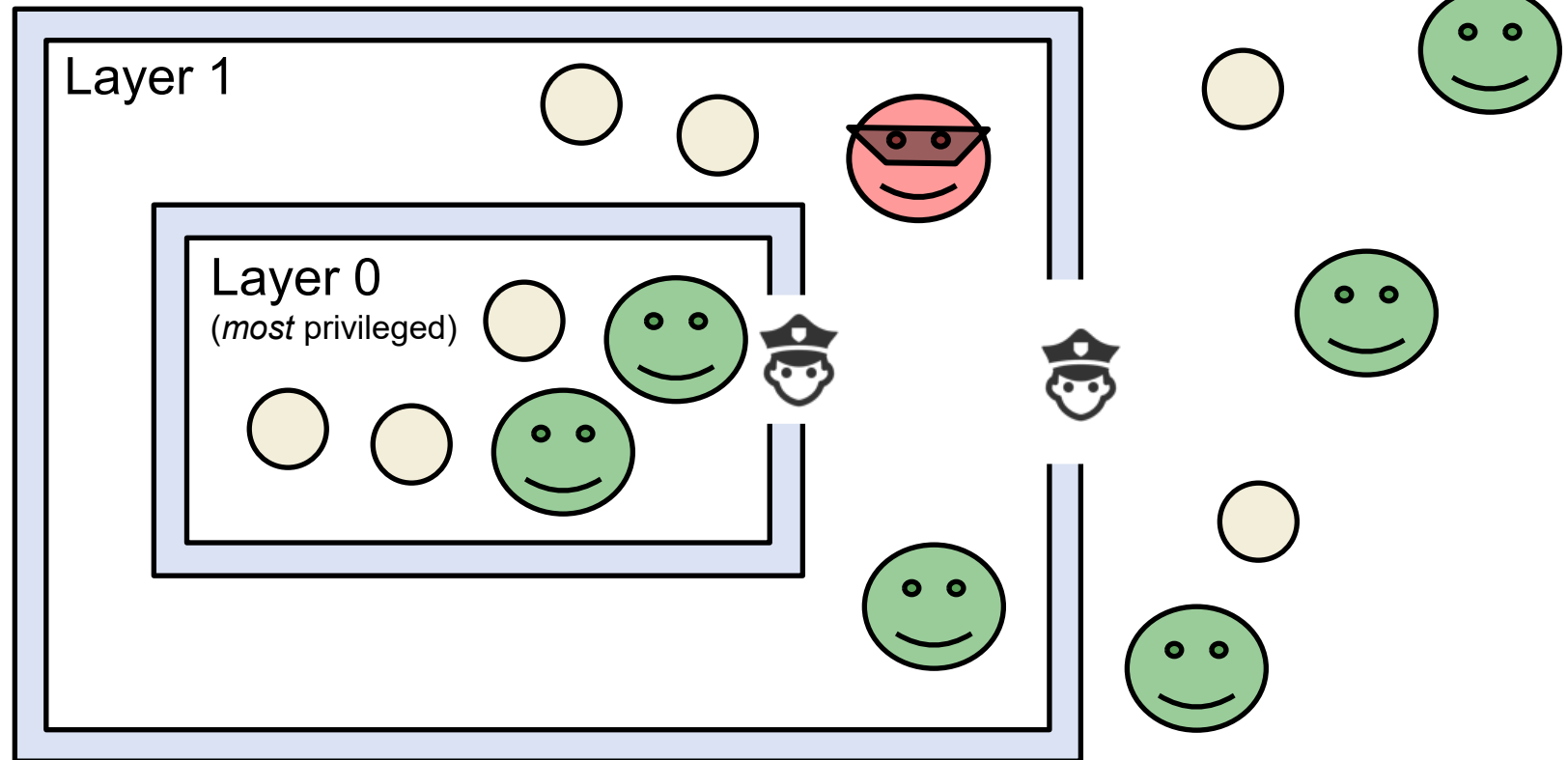
Computer system:

- 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

Layer 2
(least privileged)



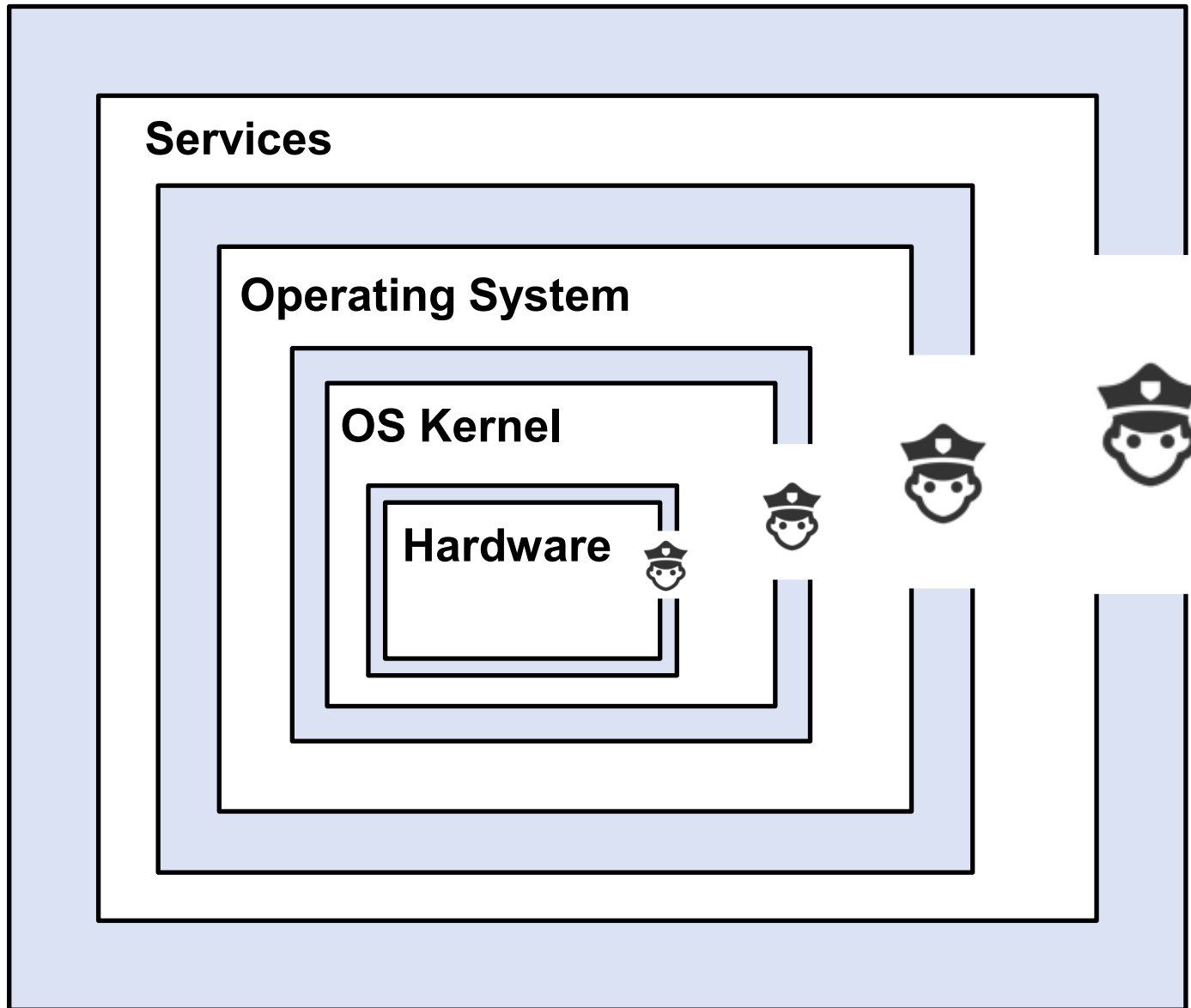
- Suppose an attacker sneaks into **Layer 1**
- The attacker must **not** be able to directly manipulate objects/data & processes in (more privileged) Layer 0
- Note that it is **very difficult** to ensure this requirement (due to possible implementation errors, overlooks in the design, etc.)

System's Layers: Attackers & Security Perimeter

- It is insightful to figure out at *what layer* a security mechanism or attack resides
- If there is bug or malware at a **lower layer**, it is very difficult for the upper layers to detect and sidestep it:
malware and vulnerabilities in OS are much more critical compared to those in the application layer (e.g. calculator app)
- 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.”
- Nonetheless, quite often, it is difficult to determine the boundary

System's Layers: With *Idealized* Security Perimeter/Boundary

Applications



System's Layers: Attackers & Security Perimeter

- An important **design consideration** of the security mechanism: **how to *prevent* attacker** from getting access to a *layer inside* the boundary
 - 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; so that 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**

7.2 Access Control

Access control is relevant to ***many systems***, such as:

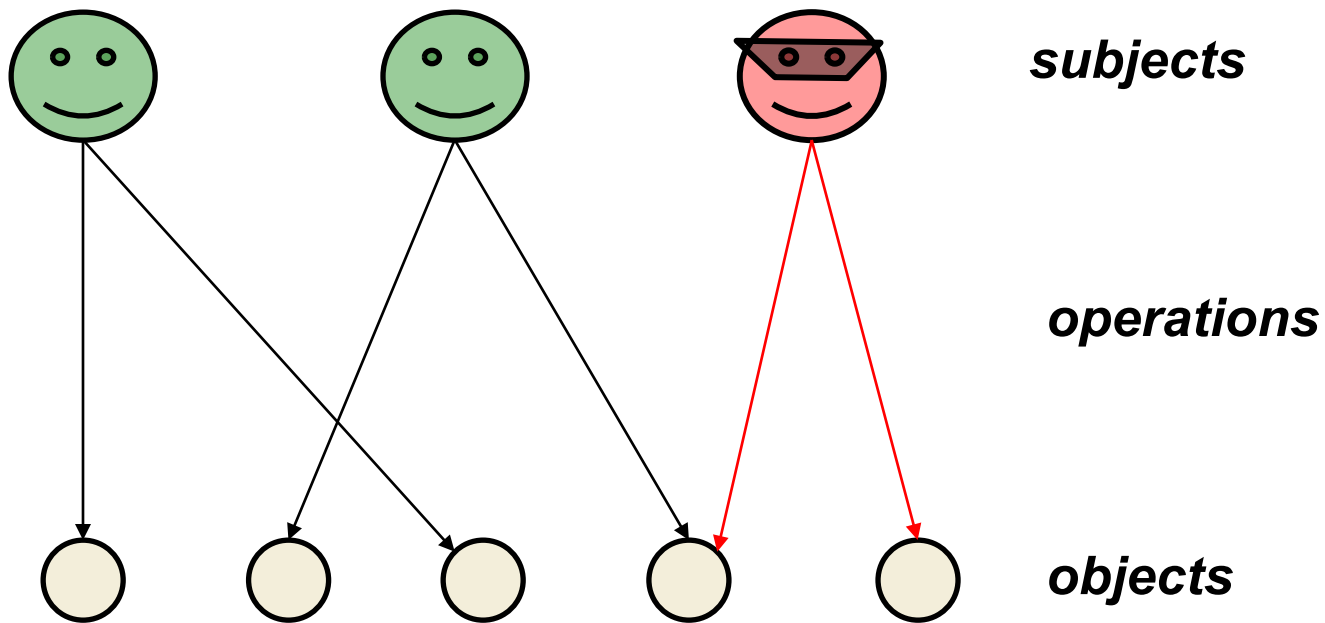
- File system (which files can a user read);
- LumiNUS (who can upload files to a Files' folder);
- Facebook (which can read my posts).

Why Access Control?

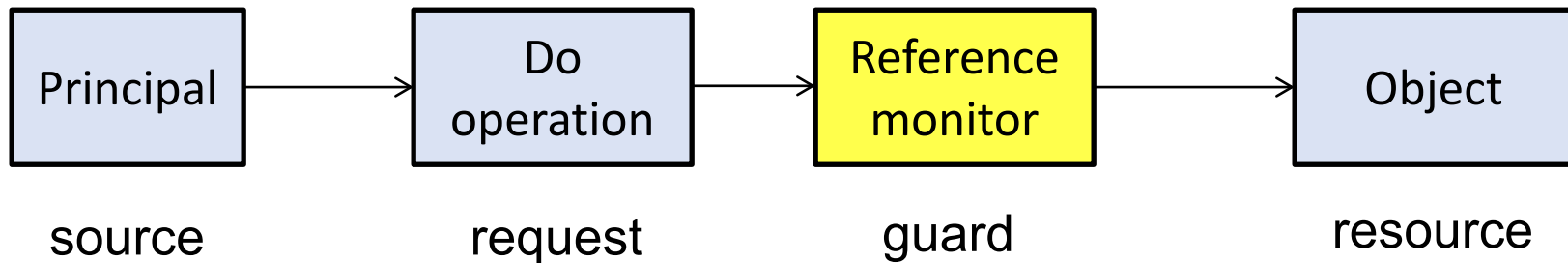
- Access control is **required** in computer system, information system, and even physical system
- Different application domains have different **requirements /interpretation**
- Examples of **application domains**:
 - OS
 - Social media (e.g. Facebook)
 - Documents in an organization (document can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

Access Control: Definition and Illustration

The *access control model/system* gives a way to **specify & enforce** restriction of *operations* on *objects* by *principals/subjects*



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
- Some 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 Files)
- **Alter**: e.g. writing a file, deleting a file, changing properties
(In LumiNUS: uploading a file to the Files)
- **Action**: e.g. executing a program

Object Access Rights and Ownership

Question: 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

(Question: Does UNIX file system adopt DAC or MAC?)

7.3 Access Control Representation

Access Control Representation

Question: How does a system represent access control?

Access Control Matrix (ACM): a matrix of principals vs objects

- ACM uses a **matrix/table** to specify the access rights of a particular principal to a particular object

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

- Sample **operations** in the table above: r=read, w=write, x=execute, o=owner, s=execute as the file owner
- Drawback:** the table would be very large, and thus difficult to manage
- Hence, ACM is seldom used

Access Control Representation

The access control matrix can be represented in 2 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?)

Access Control List (ACL)

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

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

- **Drawback** of ACL:
 - It is difficult to get an overview of which objects that **a particular subject** has access rights to
 - That is, it is hard to answer: “given a *particular subject*, which objects can this 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?

Capabilities: Examples

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

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

- **Drawback of capabilities:**
 - It is difficult to get an overview of the subjects who have access rights to a **particular object**
 - That is, it is hard to answer: “given a *particular object*, which subjects can access this 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
- The challenge faced:
 - How to specify a model that is ***fine grained*** (e.g. in Facebook, it can allow a user to specify which friends can view his/her particular photo), meets the requirements of ***security boundary***, and yet is ***easy to manage***
- Solution:
 - One simple method is to “***group***” the subjects/objects and define the access rights on **the defined groups**
 - We need an ***intermediate control***

7.4 Intermediate Control

Intermediate Control: Group in UNIX

In UNIX **file permission**, the ACL specifies the rights for the following *user classes*:

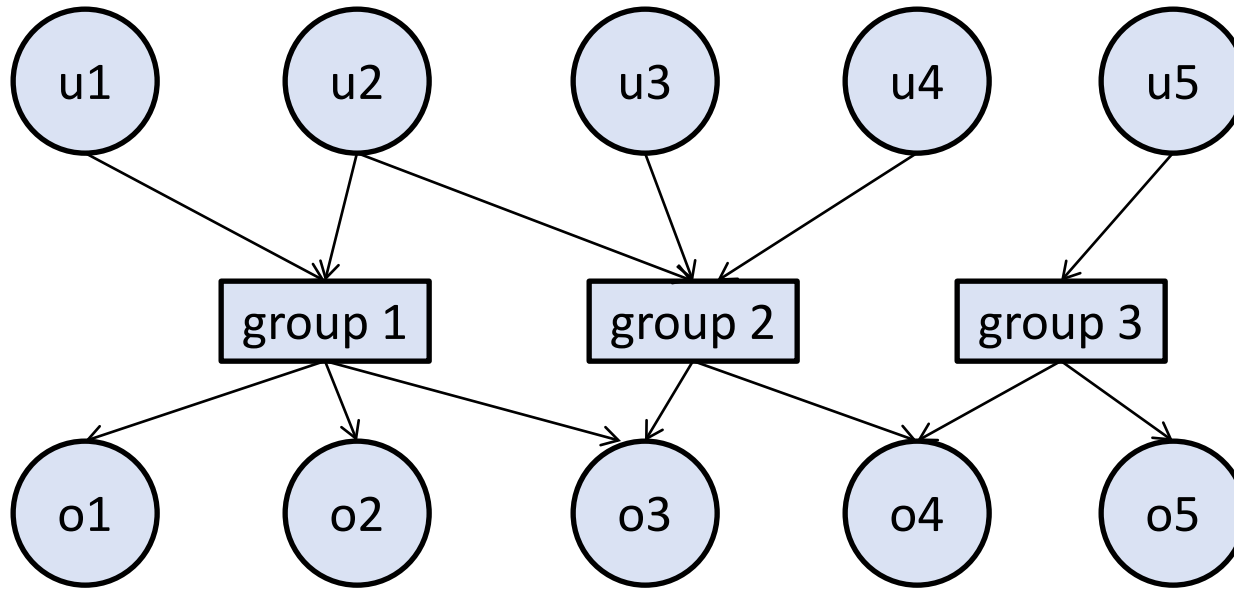
- **user**: the owner
- **group**: the owner's group(s)
- **others**: world
- Non-owner subjects **in the same group**:
have the same **group's** access rights
- Some systems demand that a subject is in a **single** group,
but some systems don't put such a restriction

Question: Is it possible in UNIX that an owner **does not** have 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
```

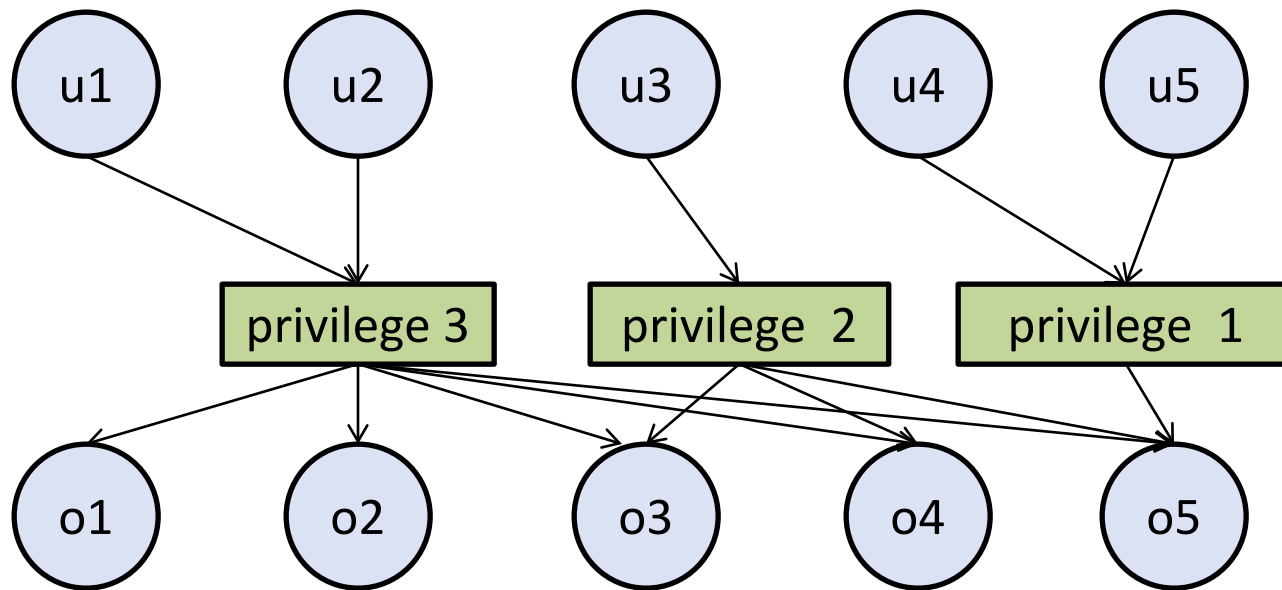
Users and Groups: Illustration



- 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 extra groups info is stored in the file `/etc/group`

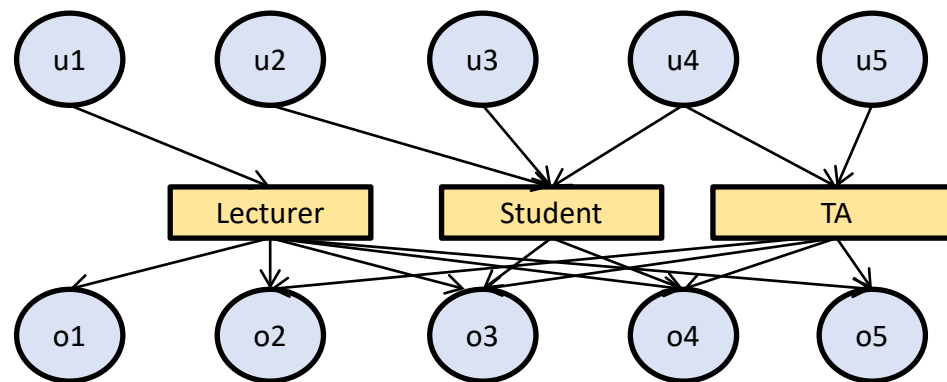
Intermediate Control: Privileges

- We sometime use the term ***privilege*** to describe the access rights
- 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 subject
- A role associates with a **set of procedures**: in order to carry out these procedures, access rights to certain objects are granted



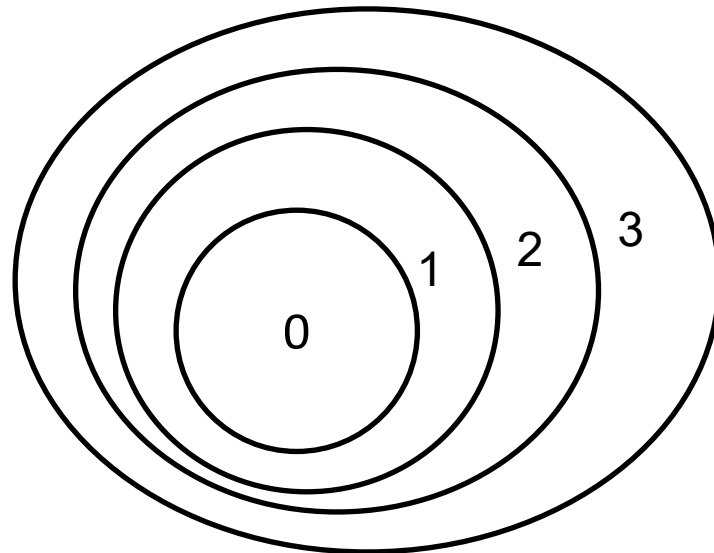
- For example, LumiNUS:
 - There are predefined rights for different roles: “Lecturer”, “TA”, “Student”
 - When Alice gets enrolled to CS2107 as a student, her rights in CS2107 are inherited from the “Student” role
 - When Alice gets appointed as a TA in CS1010, her rights in CS1010 are inherited from the “TA” role

Roles and the Least Privilege Principle: Remarks

- In design the access rights of a role, we should follow the ***least privilege principle***: access rights that are *not* required to complete the role will *not* be assigned
- For example, consider LumiNUS' gradebook:
 - The task of a student TA include entering the grade for each students
 - So, we should give the TA “write” access to the gradebook so that the TAs can complete their task
 - Should we give the TAs the right to delete a gradebook?
 - Since this access right is ***not*** required for the TA to complete their task, by the ***least privilege principle***, then the TA should *not* be given this access right

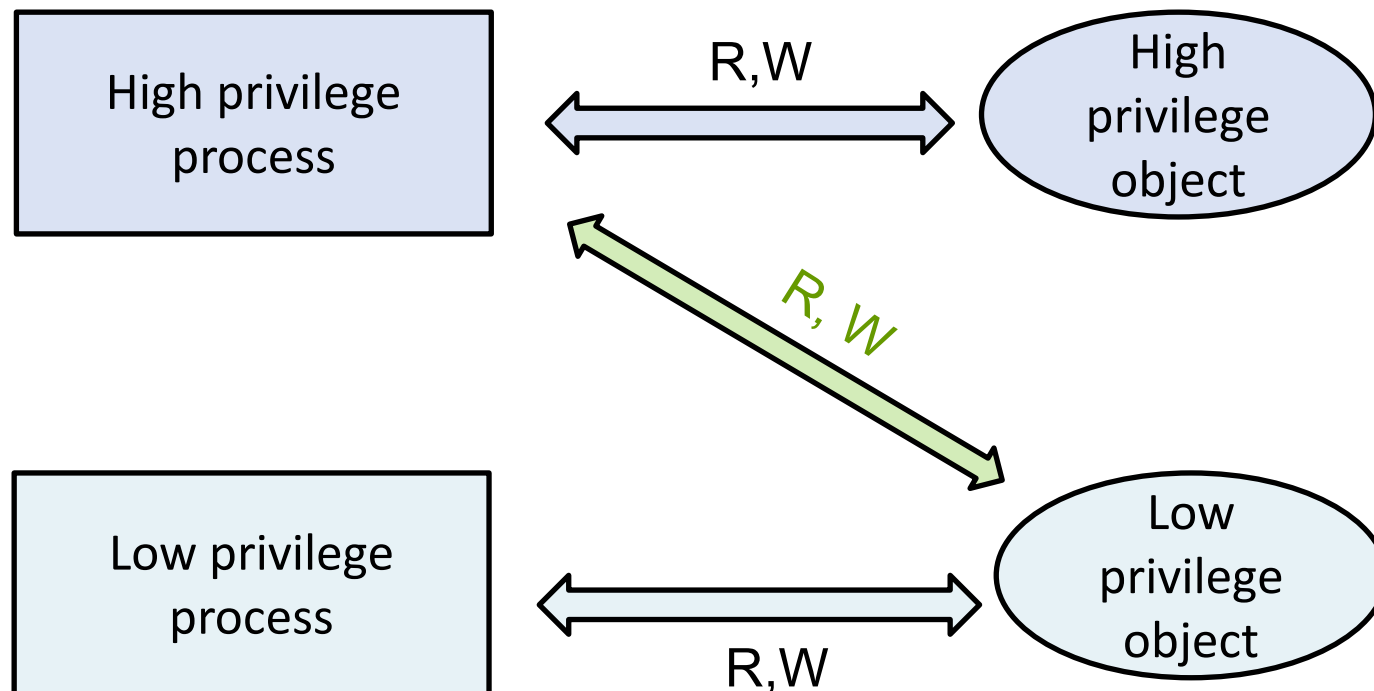
Intermediate Control: Protection Rings

- Each subject (e.g. process) and object (e.g. data) are assigned a **number**:
 - The **smaller** the number is, the **more important**:
we can call **processes** with lower ring number as having “*higher privilege*”;
likewise an **object** with smaller number are more **important**
- Whether a subject can access an object is determined by the assigned numbers:
 - If a **process** is assigned a number i : it runs **in ring i**
 - A subject **cannot** access (read/write) an object with smaller ring number



Protection Rings and UNIX

- UNIX has only 2 rings: *superuser* and (*normal/non-root*) *user*



(Note that the access control still regulates access of processes on objects of *different low-privilege users*)

7.5 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/>).

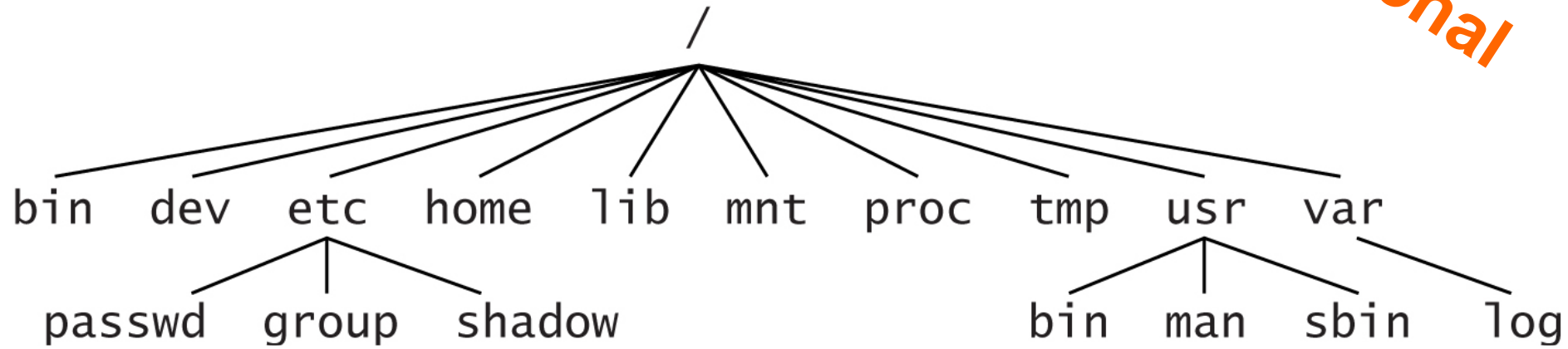
UNIX/Linux: Some Background

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 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 Objects

- 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 (*more on this later*):
 - To determine **ownership** of various system resources, e.g. files
 - To determine the **credentials** of running processes
 - To control the **permissions** granted to processes that wish to access the resources

UNIX/Linux Principals

- **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>)
- Additional group information of a user is stored in `/etc/group`
- A special user is the ***superuser***, with **UID 0**, and usually the username **root**:
 - All security checks are **turned off** for `root`
(UNIX users' protection rings with 2 rings: `superuser`, `users`)

UNIX Password File: Remarks on Its 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**, e.g. using John the Ripper.
- 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:\$6\$yonrs//S\$bUdht9fglwJW0LduAxEJpcExtMfKokFMJoT8tGkKLx5xFGJk22/trPstOHXr4PdBlD0AV1xko5LfFVDwW.aJS.:17275:0:99999:7:::

The **second** field (**hashed password**), which is shown in red, has the following format: **\$id\$salt\$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)

UNIX/Linux Subjects

- Subjects: *processes*
- A **new process**: created by invoking an executable file, or due to a “fork” by an existing process
- Each process has a **process ID (PID)**:
use the command `ps aux` or `ps -ef` to display a list of running processes

```
dave@howtogeek:~$ ps -ef | less
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	Nov15	?	00:00:03	/sbin/init splash
root	2	0	0	Nov15	?	00:00:00	[kthreadd]
root	3	2	0	Nov15	?	00:00:00	[rcu_gp]
root	4	2	0	Nov15	?	00:00:00	[rcu_par_gp]
root	6	2	0	Nov15	?	00:00:00	[kworker/0:0H-kb]
root	8	2	0	Nov15	?	00:00:00	[mm_percpu_wq]
root	9	2	0	Nov15	?	00:00:00	[ksoftirqd/0]
root	10	2	0	Nov15	?	00:00:02	[rcu_sched]
root	11	2	0	Nov15	?	00:00:00	[migration/0]
root	12	2	0	Nov15	?	00:00:00	[idle_inject/0]
root	14	2	0	Nov15	?	00:00:00	[cpuhp/0]
root	15	2	0	Nov15	?	00:00:00	[cpuhp/1]
root	16	2	0	Nov15	?	00:00:00	[idle_inject/1]
root	17	2	0	Nov15	?	00:00:00	[migration/1]
root	18	2	0	Nov15	?	00:00:00	[ksoftirqd/1]
root	20	2	0	Nov15	?	00:00:00	[kworker/1:0H-kb]
root	21	2	0	Nov15	?	00:00:00	[kdevtmpfs]
root	22	2	0	Nov15	?	00:00:00	[netns]
root	23	2	0	Nov15	?	00:00:00	[rcu_tasks_kthre

Source:

<https://www.howtogeek.com/448271/how-to-use-the-ps-command-to-monitor-linux-processes/>

Files and File System Permission

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

The diagram shows a file listing entry: `-rw-r--r-- 1 alice staff 124 Mar 9 22:29 my.c`. Brackets and arrows are used to identify the components:
- The first character `-` is identified by an arrow from the text "indicates whether it is a file (-) or directory (d)".
- The next nine characters `rw-r--r--` are grouped by a bracket and labeled "file permission".
- The number `1` is pointed to by an arrow from the text "link count (not relevant in this module)".
- The text `alice` is bracketed and labeled "owner".
- The text `staff` is bracketed and labeled "owner's group".
- The text `124` is bracketed and labeled "file size".
- The date and time `Mar 9 22:29` are bracketed and labeled "date & time of last modification".
- The text `my.c` is bracketed and labeled "filename".

The **file permission** are grouped into **3 triples**, that define the **read**, **write**, **execute** access for classes: **owner** (“user”), **group** & **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 → drwxrxr-x`

Changing File Permission Bits

- **Octal mode** notation:

- 3-4 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)

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	---

- Examples:

```
chmod 664 sharedFile
```

```
chmod 4755 setCtrls.sh
```

- **-rw-rw-r-** and **-rwsr-xr-x**

File System Permission: Additional Notes

- **Directory** permissions: *(optional)*
 - **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:** (*optional*)

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:

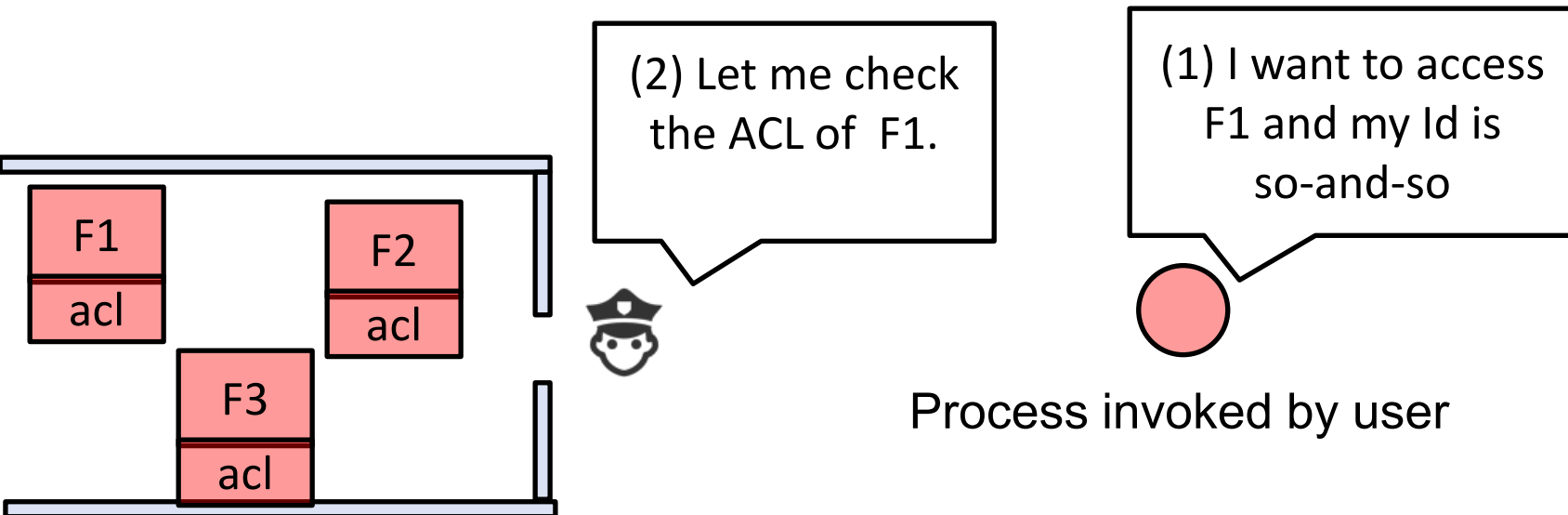
```
drwxrwxrwt 7 root sys 400 Sep 3 13:37 tmp
```

- See also: <https://docs.oracle.com/cd/E19683-01/816-4883/secfile-69/index.html>

# Objects and Their Access Control: Permission-Checking Rules!

- 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**;  
and that the **owner** of a file or **superuser** can change the permission bits
- 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

# UNIX's Access Control and Reference Monitor



## Permission-checking rules:

- When a user (subject) wants to access a file (object), the following are checked, in the following 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.

## **7.6 Controlled Invocation & Privilege Elevation**

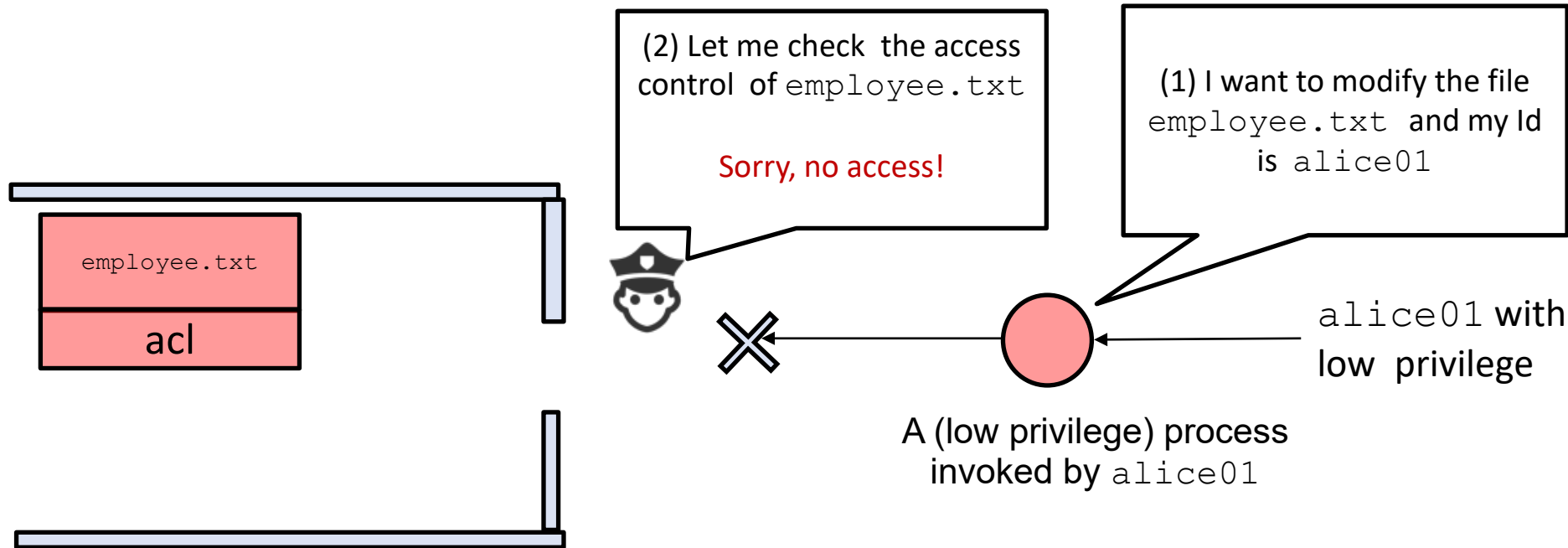


# Controlled Invocation

- The following is an **important** access control issue in UNIX/Linux
- Certain resources in UNIX/Linux can be accessed **only by superuser**, e.g.:
  - Listening at the trusted port 0—1023
  - Accessing `/etc/shadow` file
- Sometimes, a non-root user **needs** those resources for certain operations: e.g. changing password
- Yet, for security reasons, it is **not** advisable to promote the user status/privilege to superuser:
  - The non-root user may **abuse** the granted superuser privilege
- ***So, how?***

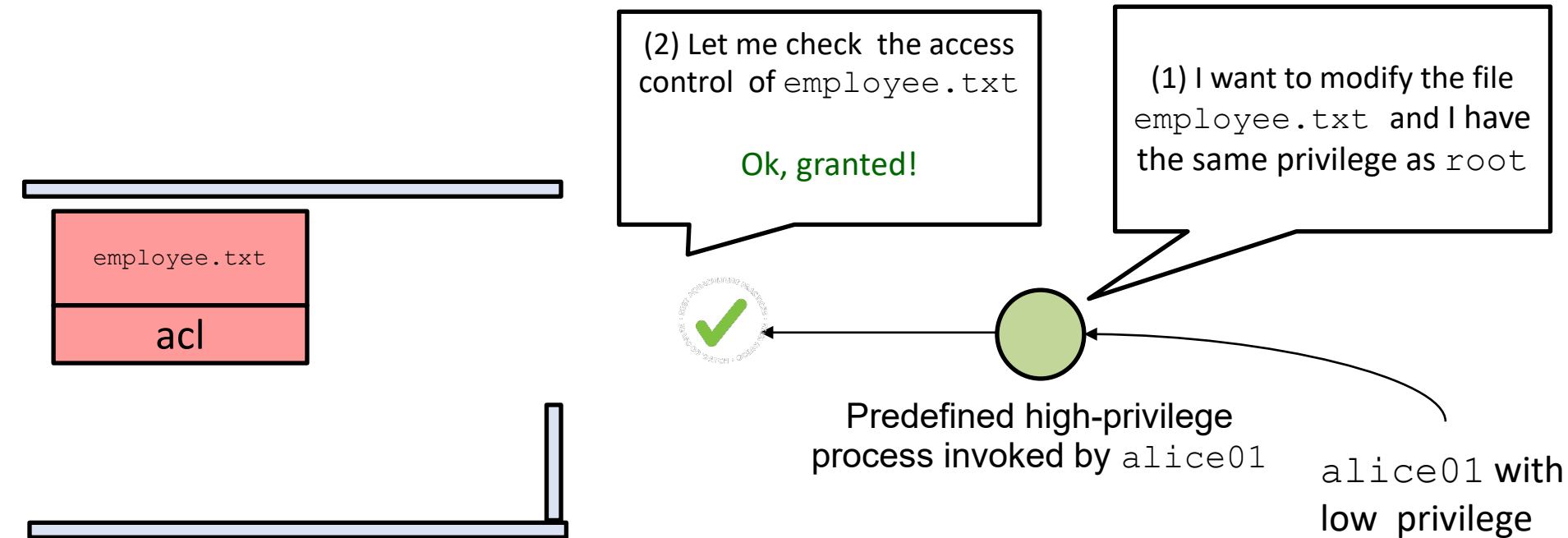
# Without Controlled Invocation

- `alice01` doesn't have access right to `employee.txt`, which contains the particulars (including addresses) of all staff members
- Any processes invoked by `alice01` inherit her rights



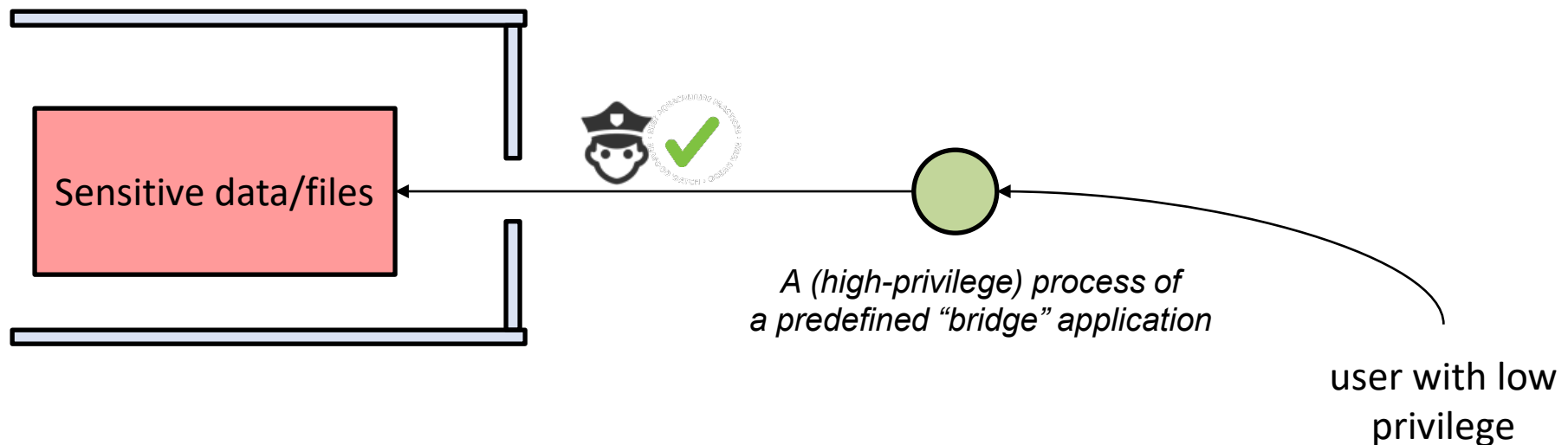
# With Controlled Invocation

- There is a set of **predefined programs** with “**elevated**” privilege
- A normal user `alice01` can't create applications with high privilege: they must be **created/provided** by **high-privilege users**, e.g. `root`
- However, any users can **invoke** these predefined applications



# Privilege Elevation: A Bridge Analogy

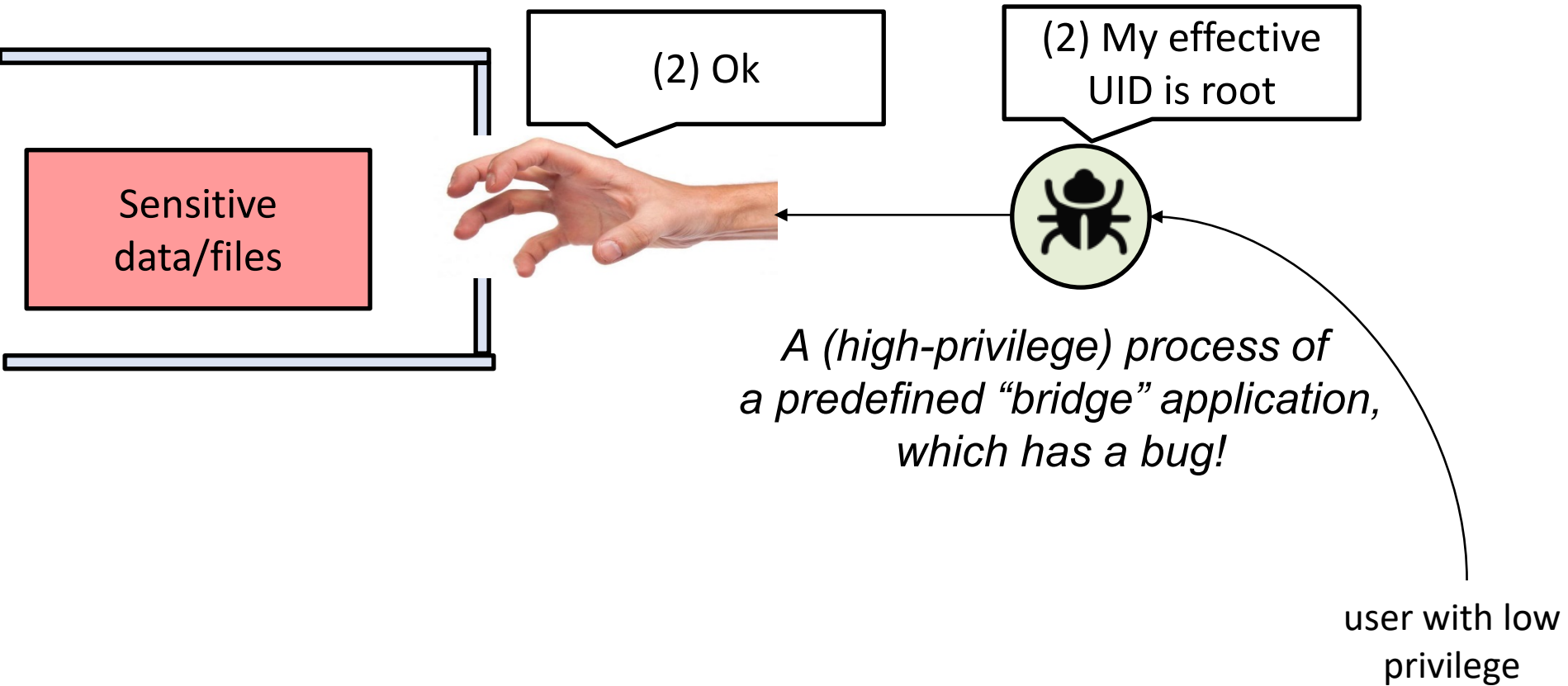
- We can view the predefined applications as predefined “**bridges**” for normal users to access sensitive data
- The applications can carry out **limited operations** only
- Note that such “bridges” can only be built by the system or provisioned by high-privilege users
- So, it is important that these “bridges” are correctly implemented, and do not grant access more than required!



# Privilege Elevation Goes Wrong

- Suppose a “bridge” is ***not*** implemented correctly, and contains **exploitable vulnerability**
- In some vulnerabilities, an attacker can trick the bridge to perform **“illegal” operations** not expected by the programmer/designer
- 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](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** (to be discussed later)

# Privilege Elevation Goes Wrong



## 7.7 Controlled Invocation in UNIX/Linux

Real UID, effective UID, privilege escalation

Note: The definitions, steps are rather complicated with many exceptions. As complexity is bad for security, users/programmers would then get confused, which leads to buggy implementations.

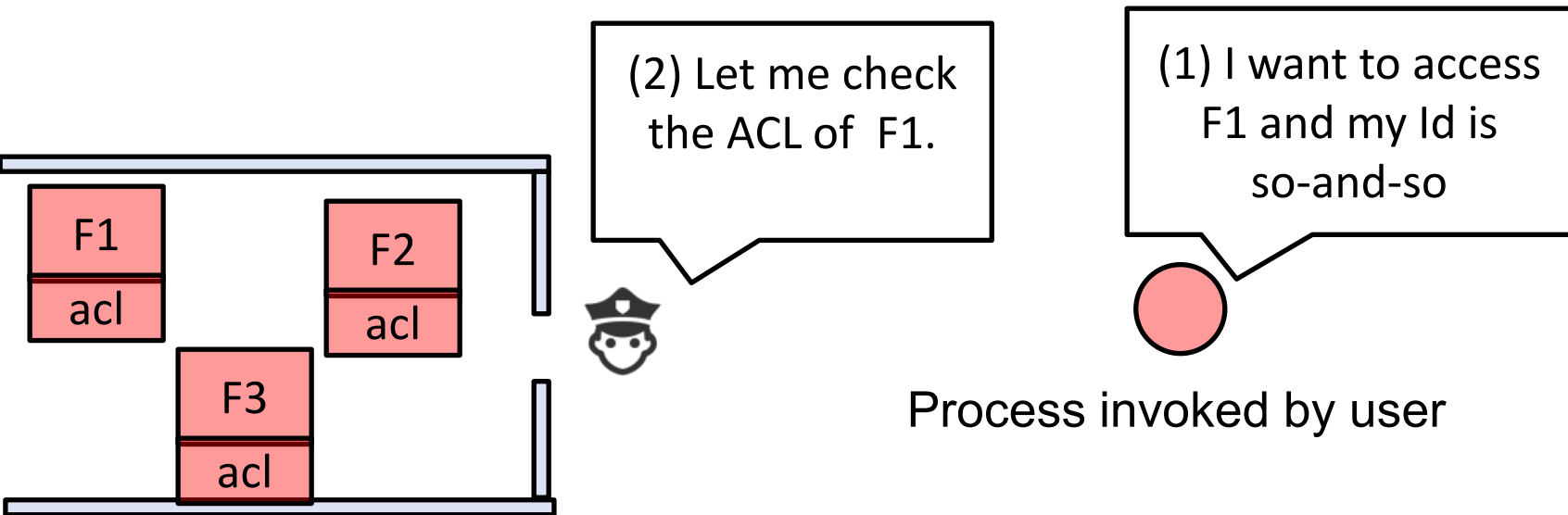
# Controlled Invocation in UNIX

- **Controlled-invocation provisioning** in UNIX:  
How to provide a predefined **set of operations (programs)** in a ***superuser mode***, and the non-root user can then run those operations with the superuser mode?
- The OS supports this solution using the **set-UID bit** of an executable
- 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 **elevated** to the ***file owner*** (in this case `root`) while a user is running this process:
  - If the file owner is *not* `root` but `bob`, then the privilege is correspondingly elevated to `bob`’s instead



# Let's Recap UNIX's Access Control and Reference Monitor



## Permission-checking rules:

- When a user (subject) wants to access a file (object), the following are checked, in the following 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.

# Process Credentials and Set User ID (Set-UID)

- A **process** is a subject: recall again that it has an identification (PID)
- A process is also 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`, 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**

\* Note: A process' real & effective UIDs are **integers**, but we simply refer to the **usernames**

# Real UID and Effective UID: Examples

- If `alice` creates the process by executing the file:


```
-r-xr-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-sr-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
```

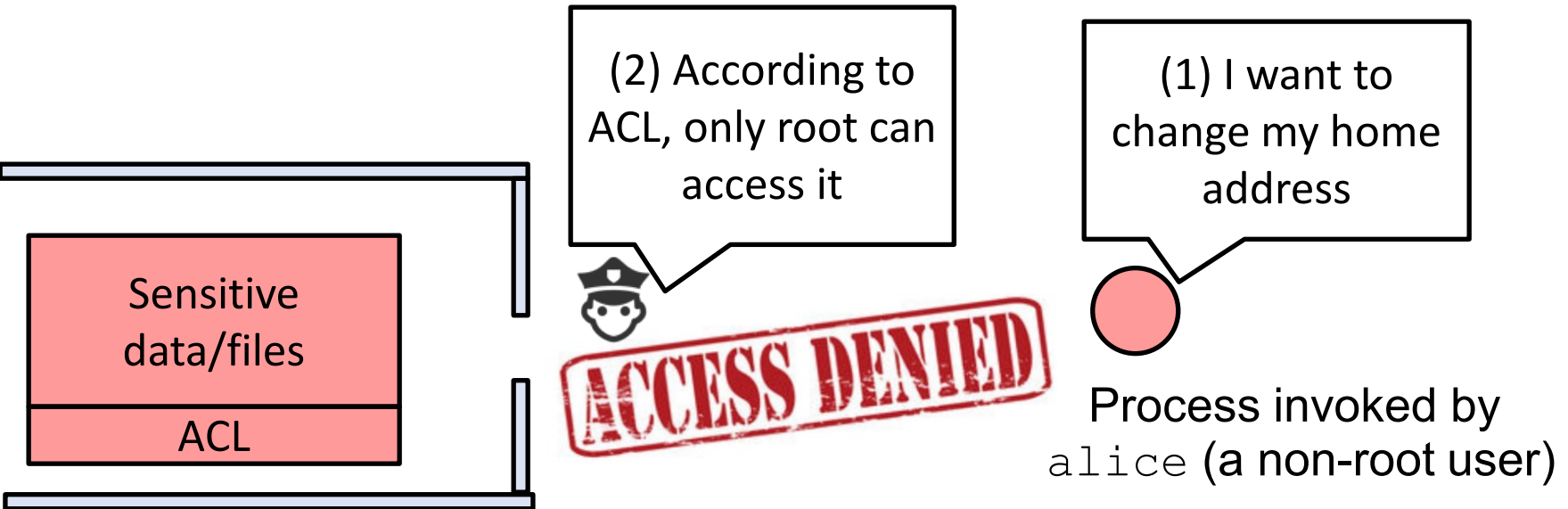
- If the effective UID of a process is **alice**, then the process will be **denied** reading the file
- If the effective UID of a process is **root**, then the process will be **allowed** to read the file

# 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`

# Solution





# Summary: When Set-UID is Disabled

- If the user `alice` invokes the executable, the process will have 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`)



|                         |                |                   |                    |                             |                           |
|-------------------------|----------------|-------------------|--------------------|-----------------------------|---------------------------|
| <code>-rw-----</code>   | <code>1</code> | <code>root</code> | <code>staff</code> | <code>6 Mar 18 08:00</code> | <code>employee.txt</code> |
| <code>-r-xr-xr-x</code> | <code>1</code> | <code>root</code> | <code>staff</code> | <code>6 Mar 18 08:00</code> | <code>editprofile</code>  |

## Summary: When Set-UID is *Enabled*

- But if the permission of the executable is “s” instead of “x”, then the invoked process will have **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`)

**ACCESS GRANTED**



|                                |                |                   |                    |                             |                           |
|--------------------------------|----------------|-------------------|--------------------|-----------------------------|---------------------------|
| <code>-rw-----</code>          | <code>1</code> | <code>root</code> | <code>staff</code> | <code>6 Mar 18 08:00</code> | <code>employee.txt</code> |
| <code>-r-<b>s</b>r-xr-x</code> | <code>1</code> | <code>root</code> | <code>staff</code> | <code>6 Mar 18 08:00</code> | <code>editprofile</code>  |

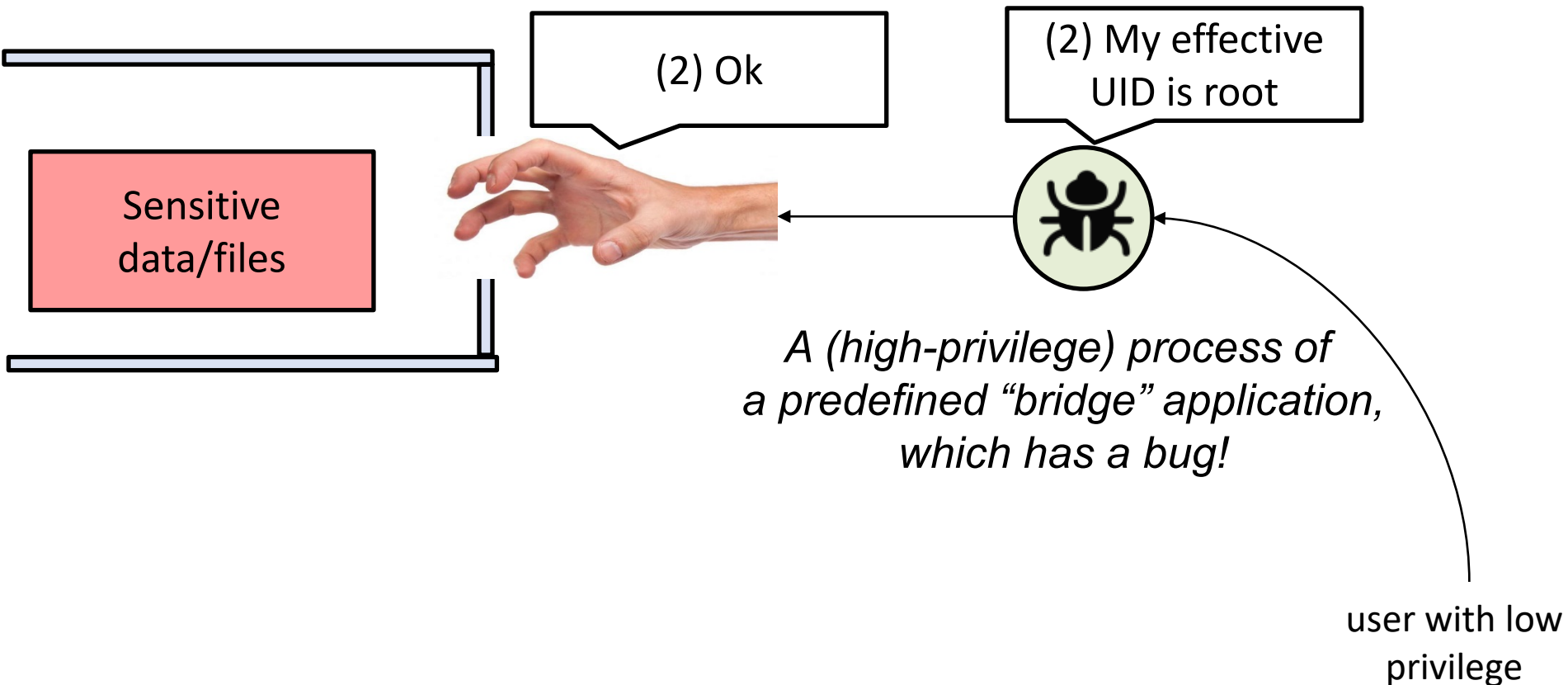
## Footnote: More Complications (Real UID, Effective UID, Saved UID)

Optional

- 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](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)

# Remember Again the Danger of *Vulnerable* Privilege Elevation

- If a “bridge” is ***not*** implemented correctly and contains **exploitable vulnerability**, this could have serious implication, since the process is now running with ***elevated privilege***



## Side Remark on UNIX's 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?*

# Summary & Takeaways

- The need to allow/block access: access control
- How to describe access control: ACM, ACL, capabilities, intermediate control, RBAC, ...
- Design guidelines: layered defense, security perimeter, the least privilege principle, segregation/compartment/segmentation, ...
- Access control in UNIX/Linux
- The needs for “access bridge” and privilege elevation
- Control invocation in UNIX/Linux