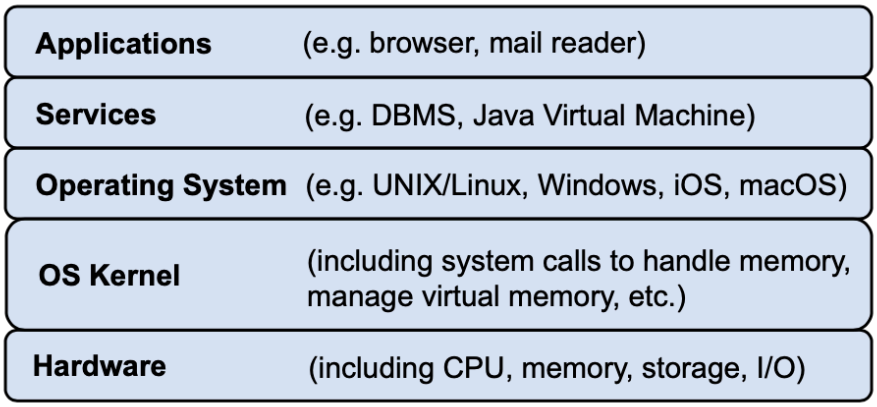
Access Control

Layering Model in Computer System Design
This is a model of the layers in a computer system:



We can actually view the OS Kernel as part of the OS. These layers are used as a guideline, and actual systems typically don't have distinct layers. For example, a windowing system may span multiple "layers".

How does this compare against Network Layers?

Network Layers	System Layers
The boundary is more well-defined	The boundary is less well-defined
Information and data flows from the topmost layer down to the lowest layer, and is transmitted from the lowest layer to the topmost layer	Every layer has its own "processes" and "data", although ultimately, the raw processes and data are handled by the hardware
A concern of data confidentiality and integrity	The main concern is about access to the processes and memory/storage (both volatile and non-volatile memory). Hence, besides data confidentiality and integrity (e.g. password file), there is also a concern of process "integrity" – which is whether it deviates from its original execution path

Using System Layers in Security

Let us assume there to be a system with Layers 2, 1 and 0, with 0 being the most important or more privileged layer. Suppose an attacker access Layer 1. He can then access data in Layer 1 i.e. Layer 1 is compromised.

What we need to thus ensure is that the attacker **must not** be able to directly manipulate objects and processes in Layer 0. This is very difficult to ensure, due to possible implementation errors, overlooked design errors, etc.

In the case of the System Layers, the “least important” layer is the application layer, while the “most important” layer is the hardware layer. But this is not really always true. The principle is to ensure that attackers should not be able to go anywhere that he should not be able to go.

It is thus insightful to figure out which layer a security mechanism or attack resides at. A (layer-based) security mechanism should have a **well-defined security perimeter or boundary**, where 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.

Basically, leave the vulnerable parts within the perimeter. However, quite often, it is difficult to determine this boundary or perimeter. An important design consideration of the security mechanism is how to prevent an attacker from getting access to a layer inside the boundary or perimeter.

For example, a Structured Query Language (SQL) Injection attack targets at the SQL Database Management System. The OS Password Management, which is in a layer below, should still remain intact even if the 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 encrypts its data before writing them to the file system, even if the access control of the file system is compromised e.g. a malicious user reads the files, the confidentiality of the data will still be preserved.

Access Control Model

Why Access Control?

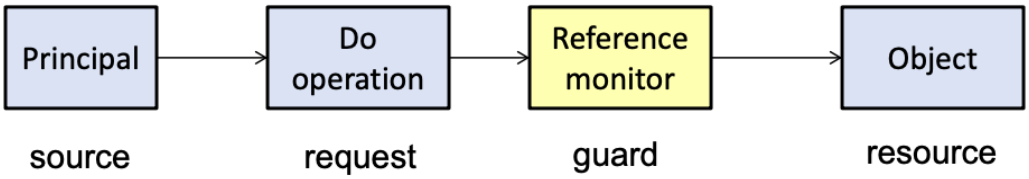
Access control is about the selective restriction of access to a place or other resource, and is required in computer systems, information systems, and physical systems, to prevent unauthorized people from accessing the system. The access control model gives a way to specify such restrictions on the subjects, objects and actions.

Different application domains have different interpretations of access control, and they have their own requirements as well. Some examples of application domains are:

- Operating system
- Social media (e.g. Facebook)
- Documents in an organization (which can be classified as “restricted”, “confidential”, “secret”, etc.)
- Physical access to different parts of a building

There is a concept called “Separation between policy and mechanism”. This design principle basically states that mechanisms should not dictate, or overly restrict, the policies. Decoupling the mechanism implementations from the policy specifications makes it possible for different applications to use the same mechanism implementations with different policies. Also, hardwiring policy and mechanism together makes policies rigid and harder to change, as trying to change the policy is likely to destabilize the mechanisms.

Principal/Subject, Operation, Object



As seen in the diagram above, a principal or subject wants to access an object with some operation. The reference monitor either grants or denies the access.

A principal is different from a subject in that the principal refers to the human user, while the subject refers to the entity in the system that operates on behalf of the principal, e.g. processes and requests.

Types of Accesses

Accesses can be classified into the following:

- Observe
  - To read a file or some data
  - For example, downloading a file from LumiNUS
- Alter
  - To write or delete a file, or change file properties
  - For example, uploading a file to the Workbin on LumiNUS
- Action
  - Execute a program

Who decides the access rights?

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 that decides the rights, which must be followed by everyone in the system. It is stricter as it’s a fixed policy.

Specifying Access Rights

Access Control Matrix
We can use a table called the access control matrix to specify the access rights of a particular principal to a particular object. This is similar to an adjacency matrix.

For example:

	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}	{}

where 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 possible pairs** of principals and objects, the table can be very large and is thus difficult to manage (i.e. low time complexity but high space complexity). Hence, it is often treated as an abstract concept only, and seldom explicitly deployed.

Much like an adjacency matrix can be expressed as an adjacency list or an edge list, the access control matrix can also be represented in two different ways: **Access Control List** or **Capabilities**.

Access Control List (ACL)
The access control list stores the access rights **to a particular object** as a list, i.e. it is object centered. This is an example:

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})	

Similar to how an adjacency list is implemented, the access control list may be implemented using a linked list concept.

Strength
The strength of an access control list is that it is easy to know all the subjects that can access a specific objects. This is because it is object centered.

Weakness
It is very difficult to find out all the objects that a specific subject can access. This will require O(os) searching through all o objects and their lists of subjects to find that individual subject.

Capabilities
Capabilities is the reverse of the access control list. It is subject centered, and every subject has a list of **capabilities,** where each capability is the access rights to an object.

A capability is defined as an unforgeable token that gives the possessor certain rights to an object.

Here is an example of an implementation of capabilities:

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})		

Strength
The strength of capabilities is that we can easily find out all the objects that a certain subject has access to, and the subject's access rights to that object.

Weakness
It is now very difficult to find out all the subjects that have access rights to a particular object.

Overall Drawback of both ACL and Capabilities
The sizes of the lists are still too large to manage. Hence, we need some ways to simplify the representation.

One such way is to group the subjects and objects and define the access rights on the defined groups. We thus need intermediate control.

Intermediate Control

In UNIX file permission, an ACL is used, but unlike the above implementation of the ACL, this ACL only specifies the rights for three parties:

- Owner
- Group
- World (others)

Subjects in the same group have the same access rights. Some systems thus demand that a subject is in a single group, but some systems do not put such a restriction.

Trivia: It is possible that an owner does not have read access, but others do.

There are many ways to perform this intermediate control.

Users and Groups

Groups are simply ways to classify users. In LumiNUS, project groups can be created, where objects created in that group can only be read by members of the group and the lecturers.

In UNIX, groups can only be created by `root`. The information on the groups are stored in the file `/etc/group`.

Privileges

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

For example, privilege 1 is the access right to execute process `mysh.sh`, while privilege 2 is the access right to execute `sudo`. One user may be assigned privilege 1, while another user may be assigned privileges 1 and 2.

This is however a bit troublesome to assign these privileges individually.

Role-based Access Control (RBAC)

The grouping can be determined by the role of the subjects, and the role itself is associated with a collection of procedures. In order to carry out these procedures, access rights to certain objects are then required.

If a subject is assigned a particular role, then we can use the **least privilege principle** to determine the role's access rights. The least privilege principle states that only access rights that are required to complete the role will be assigned, i.e. it's a need-to-know principle.

For example, a teaching assistant's role is to enter the grades for his students. He should thus have a "write" access on the grades. However, he should not have the write access to the students' names, since it is **not required** for the TA to complete his tasks.

Protection Rings

We can think of it as rings within rings, where the innermost ring has the lowest number, 0, and outer rings have increasingly larger numbers. This is somewhat similar to the example raised when discussing System Layers.

If a process is assigned a number i, we say that it runs in ring i. Objects with smaller numbers are thus more important. Often, we call processes with a lower ring number as having a "higher privilege".

Whether a subject can access an object is determined by its assigned number. A subject cannot access (i.e. both read and write) an object with a smaller ring number. It can only do so if its privilege gets "escalated" – something we will cover later.

UNIX only has two 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" itself.

In the Protection Rings model, subjects can thus access objects that are classified with the same or lower privilege. There are, however, reasonable alternatives.

There are two well-known models: Bell-LaPadula and Biba. Although they are rarely implemented as-it-is in a computer system, they serve as a good guideline.

In both models, objects and subjects are divided into linear models e.g. level 0, level 1, level 2, and a higher level corresponds to higher "security". For example, we may have a tiering system such as Unclassified, Confidential, Secret, Top Secret, etc. This is thus called multilevel security, which is when we have information at different security levels and thus have incompatible classifications.

Bell-LaPadula Model
The Bell-LaPadula Model focuses on confidentiality, and has the following restrictions:

1. **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.

2. **No write down**

A subject has **write access** to objects whose information level is **above** its current clearance level.

This prevents a subject from passing information to levels lower than its current level.

For example, to prevent information leakage, a clerk working in the highly-classified department should not be gossiping with staff from departments of lower security levels.

Biba Model
The Biba Model focuses on integrity, and has the following restrictions:

1. **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.

2. **No read down**

A subject only has **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.

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

Summary of the two models:

Bell-LaPadula Model (Confidentiality)
- A subject at a given security level cannot read an object at a higher security level
- A subject at a given security level cannot write to any object at a lower security level
- Both of these:
    - Prevents people from reading stuff that's too secure for them
    - Prevents leakage of information downwards
- Deals with secure "states", and classifies objects by security

Biba Model (Integrity)
- A subject at a given level of integrity must not read data at a lower integrity level
- A subject at a given level of integrity must not write to data at a higher level of integrity
- Both of these:
    - Prevents corruption of data of higher integrity levels by unauthorized parties
    - Prevents corruption of subjects by data from lower integrity levels
- Deals with integrity, and groups data by ordered levels of integrity

Example of an Access Control Policy (From Tutorial 6)
Let's consider the access control policy of LumiNUS forums.

The users (principals) are the lecturer, teaching assistants, students and guests.

Information associated to a post include author's name, title, content, rating of 1 to 5 stars, and number of users who have read the post.

Naturally, mandatory access control applies, as the rights of the principals on the available objects are set by the LumiNUS system. Let us consider Teaching Assistants as part of Lecturer.

Object > Principal v	Whole post (with no child post)	Author's name	Title	Content	Viewed	Rating
Lecturer (post-owner)	delete, create*	r	rw	rw	r	rw
Lecturer (non-post-owner)	delete	r	rw	rw	r	rw
Students (post-owner)	delete, create*	r	rw	rw	r	r
Students (non-post-owner)	-	r	r	r	r	r
Guests	-	r	r	r	r	r

*A whole post is created with default values set on its fields

Alternatively, instead of splitting the principals into post-owners and non-post-owners, another way is to split the objects into post-owners and non-post-owners. However, the above representation is more compact and hence more visually appealing.

Access Control in UNIX/Linux

The following section contains various technical information about users, groups and processes.

Terminology

Objects

In UNIX, objects of access control include:

- Files
- Directories
- Memory Devices
- I/O Devices

All of these resources are treated as files.

Users 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`
  - Any additional groups are stored in `/etc/group`

Each group:

- Has a unique group name
- Has a numeric **group identifier (GID)**

What is the purpose of UIDs and GIDs?

- To determine the **ownership** of various system resources
- To determine the **credentials** of running processes
- To control the **permissions granted** to processes that wish to access certain resources

Therefore, in UNIX, we can think of it as:
User (UID) + Group (GID) → Principal (UID + GID) → Subject (Process ID (PID)) → Object

Principals are made up of user identities (UID) and group identities (GID). The information on these principals or the user accounts are stored in the password file `/etc/passwd`

Example of how the user account may look like in that file:
`root:*:0:0:System Administrator:/var/root:/bin/sh`
In the above, `0:0` means that it has UID 0 and GID 0 respectively.

A special user is the **superuser**, with UID 0, and it usually has the username `root`. All security checks are turned off for `root`, as we mentioned earlier, UNIX's protection rings consists of only 2 rings: superuser and users.

Subjects are the processes. Each process has a process ID (PID). We can use the command `ps aux` or `ps -ef` to display a list of running processes.

Password File Protection

The passwd file is made world-readable because some information in `/etc/passwd` are needed by non-root processes. This file contains a list of user accounts in the format shown below.

```
jsmith:x:1001:1000:Joe Smith,Room 1007,(234)555-8910,(234)555-
0044,email:/home/jsmith:/bin/sh
```

The fields, in order from left to right, are:

1. **Username**
    a. This is the string that a user would type in when logging into the operating system, i.e. the logname.
    b. Must be unique across all users listed in the file.
2. **Information to validate a user's password**
    a. In most modern uses, this field is usually set to "x" or "*", or some other indicator.
    b. The actual password information is stored in a separate shadow password file.
    c. In older versions of UNIX, the location of "*" was the hashed password H(pw), allowing all users to have access to this hashed-password field.
        i. The availability of the hashed password allowed hackers to do offline password guessing.
        ii. Since many passwords are typically short, exhaustive search is able to obtain many passwords.
    d. Thus now the actual password is stored in `/etc/shadow`, which is not world-readable.
3. **User Identifier Number**
    a. Used by the OS for internal purposes
    b. Needs not be unique
4. **Group Identifier Number**
    a. Identifies the primary group of the user
    b. All files created by this user may initially be accessible to this group
5. **Gecos field**
    a. Commentary that describes this person or account
    b. Typically, it is a set of comma-separated values that includes the user's full name and contact details
6. **Path to the user's home directory**
7. **Program that is started every time the user logs into the system**
    a. For an interactive user, this is usually one of the system's command line interpreters (shells)

Shadow Password File

This is the file that actually contains the hashed passwords. The fields of the entry are:

- 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

The following is an example:

```
user1:$6$yonrs//S$bUdht9fglwJW0LduAxEJpcExtMfKokFMJoT8tGkKLx5xFGJk22/trPstOHXr4PdB
lD0AV1xko5LfFVDwW.aJS.:17275:0:99999:7:::
```

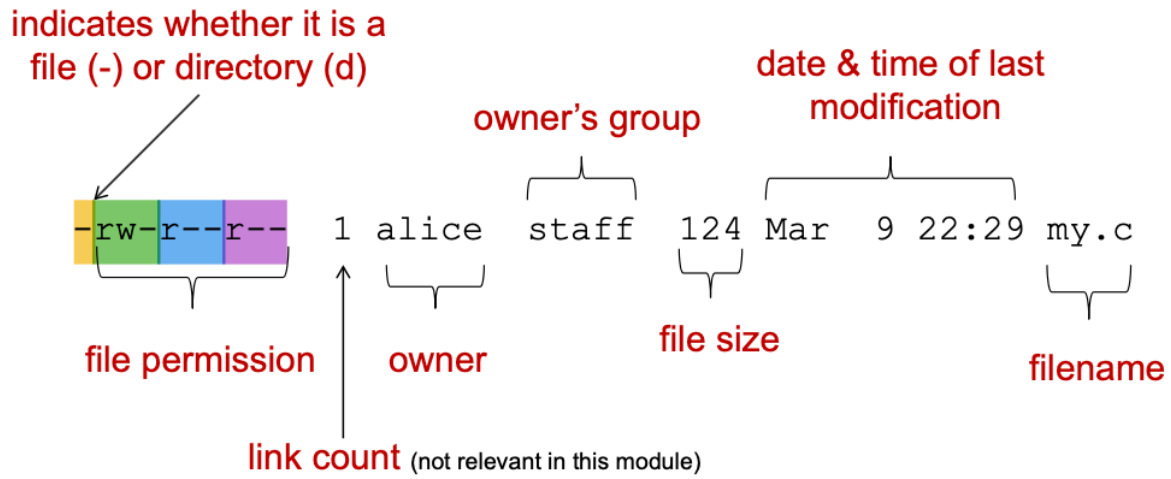
The second field, hashed password, is shown in red and has the following format: idsalt$hashed-key

- **id:** ID of the hash method used
  - 1 – MD5
  - 2a – Blowfish
  - 2y – Blowfish
  - 5 – SHA-256
  - 6 – SHA-512
- **salt:** up to 16 characters drawn from the set [a-zA-Z0-9./]
- **hashed-key:** hash of the password
  - 43 characters for SHA-256
  - 86 characters for SHA-512

Thus, in the example above, the id is 6 i.e. SHA512 is used, the salt is `yonrs//S`, and the hashed-key is the long 86 character string.

File System Permission

When we use `ls -la`, we see a list of all content within that certain directory, with the following information:



In the above image, we see that the **file permission** component is made up of 3 triples, that define the read, write and execute access. The breakdown is as follows:

- First triple: Owner ("user")
- Second triple: Group
- Third triple: Others (the "world")

Within each triple, the characters are:

- First character:
  - '-': access is not granted
  - 'r': read access is granted
- Second character:
  - '-': access is not granted
  - 'w': write access is granted (including delete)
- Third character:
  - '-': access is not granted
  - 'x': execute access is granted
  - 's': setuid/setgid and execute access is granted
  - 't': sticky and execute access is granted
  - 'S': setuid/setgid and execute access is **not** granted
  - 'T': sticky and execute access is **not** granted

The above is called the symbolic notation.

Changing File Permission Bits

You can use the chmod command to change the file permission bits. The command is:

```
chmod [options] mode[,mode] file1 [file2 ...]
```

The useful options available are:

- -R: Recursive, i.e. to include objects in subdirectories
- -f: force the processing to continue even if errors occur
- -v: verbose, i.e. show the objects changed

There are two notations for mode:

- Symbolic mode notation

    - Syntax: [references][operator][modes]
    - References:
        - u (user)
        - g (group)
        - o (others)
        - a (all)
    - Operators:
        - + (add)
        - - (remove)
        - = (set)
    - Mode:
        - r (read)
        - w (write)
        - x (execute)
        - s (setuid/gid)
        - t (sticky)
    - Examples:
        - `chmod g+w shared_dir`
            - Add write access to the shared directory for the group
        - `chmod ug=rw groupAgreements.txt`
            - Set read and write access for user and group to groupAgreements.txt
- Octal mode notation
    - This notation contains 3 or 4 octal digits.
    - The 3 rightmost digits refer to the permissions for the file user, the group, and others respectively.
    - There is an optional leading digit, allowing 4 digits to be given. This digit specifies the special file permissions:
        - setuid
        - setgid
        - sticky bit

#	Permission	rwx	Binary
7	read, write and execute	rwx	111
6	read and write	rw-	110
5	read and execute	r-x	101
4	read only	r--	100
3	write and execute	-wx	011
2	write only	-w-	010
1	execute only	--x	001
0	none	---	000

#	Permission	rwx	Binary
4	setuid	s/S	100
2	setgid	s/S	010
1	sticky	t/T	001
0	none	-	000

To some extent, these special file permissions “piggyback” on the third bit of the first three bits, i.e. using s/S/t/T instead of x or – to represent the leading digit.

- Examples
  - `chmod 664 sharedFile`
    - User can read and write
    - Group can read and write
    - Others can read only
  - `chmod 4755 setCtrls.sh`
    - setuid is enabled
    - setgid disabled
    - sticky bit disabled
    - User can read, write and execute
    - Group can read and execute
    - Others can read and execute

Directory Permissions
Directory permissions are slightly unique:
- **r:** Read access allows a user to view the directory’s contents
- **w:** Write access allows a user to create new files to delete files in the directory
- **x:** Execute access determines if a user can enter (cd) into the directory or run a program or script

If you want all group members to be able to write, edit or delete files within this directory, you can chmod g+w for that directory. However, this means that a user with write privileges in the directory can actually delete a file even if they do not have write permissions for the file.

Special File Permissions
Set-UID
- The process’ effective user ID is that of the **owner** of the executable file (which is usually root), rather than the user running the executable.
- For example: `-r-sr-sr-x 3 root sys 104580 Sep 16 12:02 /usr/bin/passwd` will run with root as the effective user ID.

Set-GID
- The process’ effective group ID is the **owner’s group**.
- For example, `-r-sr-sr-x 3 root sys 104580 Sep 16 12:02 /usr/bin/passwd` will run with sys as the effective group ID.

Sticky bit
- If a directory has the sticky bit set, the files within the directory can only be deleted by the owner of the file, the owner of the directory, or by root.
- This prevents a user from deleting files of other users from public directories such as `/tmp`.

Back to Access Control
In UNIX, objects are files, and each file is owned by a user and a group, and is associated with a 9-bit permission.

When a non-root user (subject) wishes to access a file (object), the following are checked in order:
1. If the user is the owner, the permission bits for the **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 right

3. If the user is not the owner nor a member of the group that owns the file, then the permission bits for **other** decide the access rights.

In general, the owner of a file and the superuser can change the permission bits.

Search Path Issues

When a user types in the command to execute a program, e.g. "`su`", without specifying the full path, what happens is that there will be a search for this program through the directories specified in the **search path.**

Use the command `echo $PATH` to see the search path. The search path is a list of directories, and the search will go through them one by one to find the program. Once a program with the name is found in a directory, the search stops and the program will be executed.

Suppose an attacker manages to store a malicious program at the directory that appears in the beginning of the search path, and the malicious program has a common name, e.g. "`su`". When a user exerts "`su`", the malicious program will be invoked instead.

To prevent such an attack, specify the full path always. Also avoid putting the current directory ("`.`") within the search path, e.g. `./a.out`. Say we have all these little scripts all over our filesystem; one day we will run the wrong one for sure. So, having our path as a predefined list of static paths is all about order and saving ourselves from a potential problem.

However, if you're going to add "`.`" to your `PATH`, it is recommended to append it to the end of the list (`export PATH=$PATH:.`). At least you won't override system-wide binaries this way.

However, if you're a root on the system and have your system exposed to other users' accounts, having "`.`" in PATH is a huge security risk – you can cd to some user's directory, and unintentionally run a malicious script there only because you mistyped a thing or there's a script that has the same name as a system-wide binary.

UNIX/Linux: Privilege Escalation (Controlled Invocation)

Certain resources in the UNIX/Linus system can only be accessed by superuser, for example, listening at the trusted port 0-1023, accessing `/etc/shadow` file etc. Sometimes, a user needs those resources for certain operations such as changing their password. But it is not advisable for user to elevate their user status to superuser.

A solution is **controlled invocation**: the OS provides a predefined set of operations (programs) in superuser mode, and the user can then invoke those operations with the superuser mode.

Processes and Set-UID

A process is a subject and has a Process ID as identification. A new process can be created by executing a file or due to a fork in an existing process.

The process is associated with process credentials:

- Real UID
- Effective UID

The real UID is inherited from the user who invokes the process. It identifies the real owner of the process. For example, if the user who invokes it the process is Alice, then the process' real UID is that of Alice.

Let's say we invoked the process by executing a file. Then:

- If set-UID is disabled i.e. the permission bit is "`x`", then the process' effective UID is the same as the real UID
  - real UID is `alice`
  - effective UID is `alice`
- If set-UID is enabled i.e. the permission bit is "`s`", then the process' effective UID is inherited from the UID of the file's owner.
  - real UID is `alice`
  - effective UID is `root`
    - If the owner is Bob, then the effective UID would be `bob`

Process (Subject) Accessing Files (Objects)

When the process wants to access a file, the effective UID of the process is treated as the "subject" and checked against the file permissions to decide whether it would be granted or denied access.

For example:

```
-rw------- 1 root staff 6 Mar 18 08:00 sensitive.txt
```

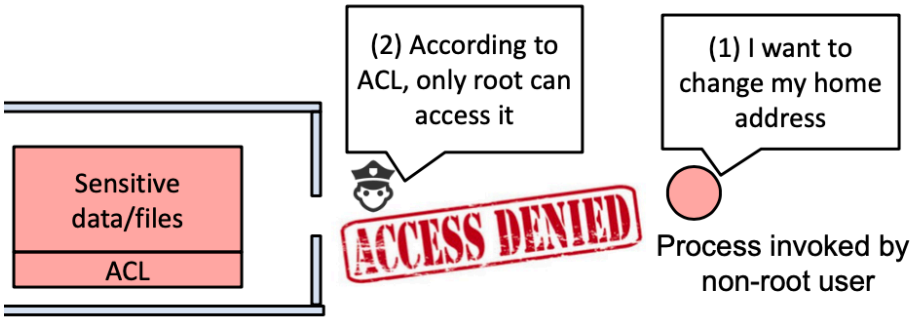
If the effective UID of a process is `alice`, then the process will be denied from reading the file. However, if the effective UID of a process is `root`, then the process will be allowed access.

Use Case of Set-UID

Consider a scenario where the file `employee.txt` contains personal information of the users. This is sensitive information, hence the system administrator sets it as 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 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 or write to it.



The solution would be to create an executable file `editprofile` owned by `root`:

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

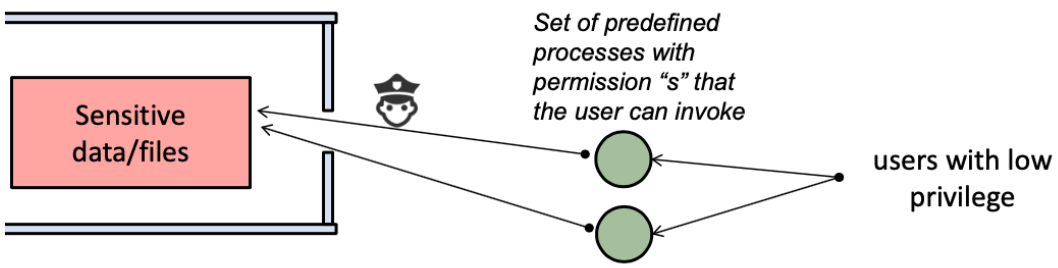
The program is made world-executable, such that any user can execute it. Furthermore, the set-UID bit is set, thus when it is executed by users other than `root`, the effective UID of the process will be `root`.

This is an example of a set-UID-root program or executable. Now, if `alice` executes the file, the process' real UID is `alice`, but its effective UID is `root`. This process can now read and write to the file `employee.txt`.

In the above example, the process `editprofile` is temporarily elevated to the superuser status i.e. `root`, such that it can access the sensitive data.

We can view the elevated process as the interfaces where a user can access the sensitive information. These processes are predefined "bridges" for the user to access the data. Note that the "bridge" can only be built by the `root`.

Due to the capability of the "bridges", it is important that they are correctly implemented and do not leak more information than required.



Privilege Escalation Attacks
Suppose that one of these processes is not implemented correctly, and contains an exploitable vulnerability. An attacker, by feeding this process with carefully-crafted input, can actually cause it to perform malicious operations, compromising confidentiality and integrity.

This can have serious implications, since the process is now running with an elevated privilege. An attack of such form is also known as a privilege escalation attack.

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 thus leads us to **secure programming** and **software security.**

Even More Complications – Real UID, Effective UID, Saved UID
The OS actually maintains three IDs for a process:
- Real UID
- Effective UID
- Saved UID

The saved UID is like a “temp” container and is useful for a process running with elevated privileges to drop its privilege temporarily – a good set-UID programming practice.

A process would remove its privileged user ID from its effective UID, but stores it in its saved UID. Later on, the process may restore the privilege by restoring the saved privileged UID back as its effective UID.

The details may easily confuse many programmers, not to mention that different UNIX versions may have different behaviours when it comes to this.

All this complexity is actually bad for security!

Backdoor Programs and Insider Threats (from Tutorial 7)

Some background info:

Shell program

Consider a UNIX-based OS and an executable program named `mysh`, which essentially is an existing command `sh` known as "shell". Running the program `mysh` with parameter `file1` will read and execute all commands contained in the file `file1`. Example:

```
$ mysh file1
```

Backdoor program

An attacker, who is assumed to already have a local access, can run the following commands using your account to plant a backdoor in your system in seconds:

```
$ cp /bin/sh /tmp/mysh
$ chmod u+s /tmp/mysh
```

Notice that you are the owner of the planted shell program. Therefore, when later the attacker run the planted set-UID program, the program's effective UID will be your UID.

Seeing set-UID-root in action

A set-UID-root program is an executable program owned by the root, and has its set-UID bit enabled. When it is invoked by a local non-root user, the process will run with its effective user ID (euid) set to the root, thus running with an elevated/escalated privilege.

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(){
  printf("Real user id = %d, Effective user id = %d\n", getuid(), geteuid());
  return 0;
}
```

When you compile the above code as root, enable its set-UID bit and run it as a local non-root user, the output differs from when its set-UID bit is disabled.

Backdoor Creation as an Insider/System Administrator

Suppose a system administrator had root access on a system and was about to leave a company, and wanted to plant a backdoor into the target system. The assumptions are:

- He would still have local access, through
  - Given local user access
  - Knew the password of an existing local user
  - Created a local user that would go unnoticed
- Root password would be changed after he left
- His home directory would be erased
- The new system admin would compare the system-level directories, such as `/usr/bin`, to make sure that no malicious changes had been made

The above-mentioned method of using set-UID probably can't work in your favourite modern UNIX-based OS. This is because, due to the above security concerns, many OSes ignore the elevated effective UID when running shell scripts. This can be viewed as an "ad-hoc" patch to the security concern.

Nonetheless, there are methods to get around it, and there is fundamentally no way of stopping the system administrator. The “ad-hoc” patch only makes it slightly more difficult for the admin to create a backdoor.

Failed Attempt to Plant a Copied Shell
Back to the given scenario, when the admin still has root access, as the root, he can attempt to plant a copied shell by doing:

```
# mkdir -p /games/pokemon/scores
# cp /bin/sh /games/pokemon/scores/removescores
# chmod u+s /games/pokemon/scores/removescores
```

Unfortunately, when the admin later logs in as a non-root local user and executes `removescores`, the invoked shell won’t run with root privilege. You can check this by using the `id` command

This is because in Ubuntu, `/bin/sh` actually points to `/bin/dash` (Debian Almquist shell), which is a lightweight `bash` variant. As mentioned above, as a security protection measure, `bash` automatically downgrades its privilege. When `bash` detects that it is executed in a set-UID process, it will immediately change its effective user ID to the process’ real user ID, thus essentially dropping the escalated privilege. (Note that bash in older versions of Ubuntu, e.g. version 12.04, does not have this protection measure.)

However, there are ways to bypass this protection measure. The real root can still invoke `bash` without its privilege gets dropped. Hence, if we can make `bash` run with its real user ID set to 0, then the root privilege will be retained.

Successful Attempt
Instead of copying `/bin/sh`, as the root, the admin can just compile `getroot.c` to create an executable named `/games/pokemon/scores/removescores`, and enable its set-UID bit. As you can see, `getroot.c` first turns the current set-UID process into a real root process by invoking, among others, `setuid(0)`.

What `setuid(0)` does is, if the effective user ID is 0, then it will set the real user ID to 0. Else, an unprivileged program can use `setuid()` to change its effective user ID to the real user ID (in the case that it was started as a set-UID non-root e.g. Bob running a program with Alice’s user ID as his effective user ID). Only a privileged programme i.e. effective user ID of 0 can call `setuid()` to set both user IDs to a value of its own choice.

If the calling process is privileged, setuid(0) sets both the real and effective user IDs of the process to 0. In this case, `getroot.c` is privileged. Once both real and effective user IDs are set to 0, a shell `/bin/sh` is subsequently invoked. As a result, when the admin executes the planted executable as a non-root local user, he will obtain a shell running with a root privilege as planned.

The technique above is just one way of creating a backdoor. There are other ways, such as:

- setting a non-root local user to have UID 0
- including a non-root local user into the `sudo` group