# CSCI 476: Computer Security

Lecture 5: Set-UID and Environment Variables

Reese Pearsall

Fall 2022



*all images are stolen from the internet

# Announcements

Lab 1 Due **FRIDAY** 9/16 @ 11:59 PM

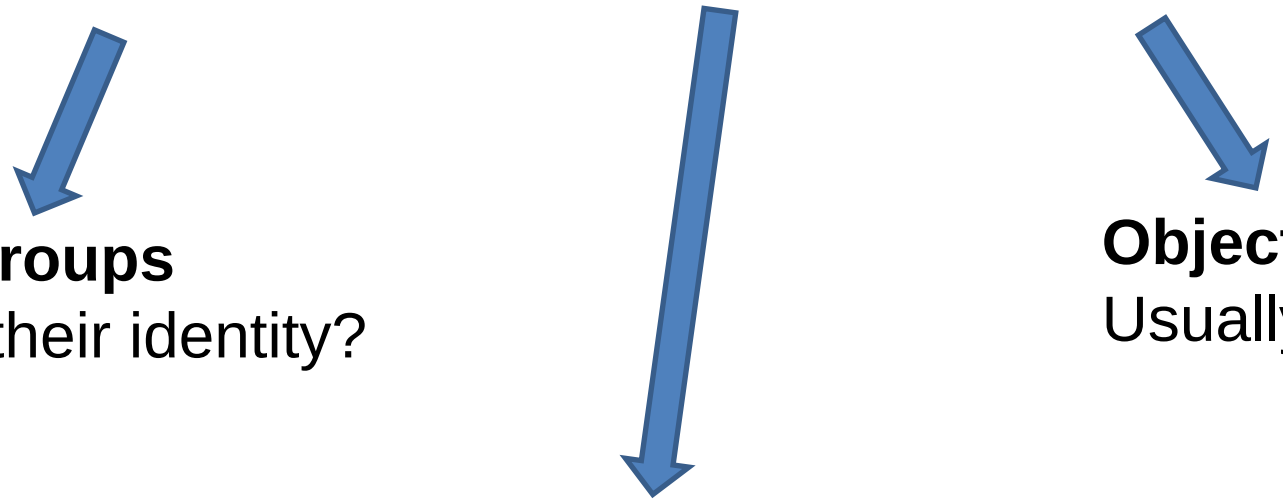
- Shouldn’t be too bad

Note taker still needed

How would you protect your computer and its resources?

# Access Control

**who** can do **what** to **whom?**



**users/groups**
what is their identity?

**Objects**
Usually things on a filesystem

**permissions (read/write/execute)**
Ok, I know the who– what are ***you*** permitted to do?

# **who** can do **what** to **whom?**

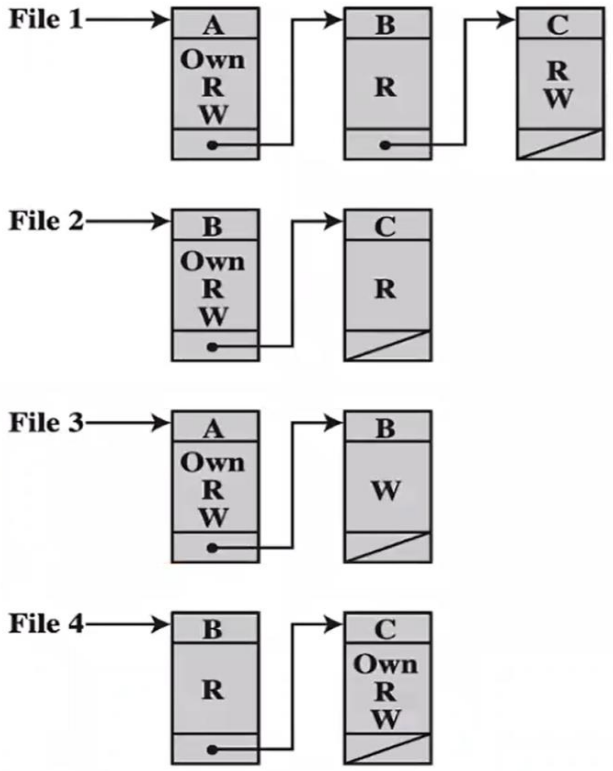
| | | OBJECTS | | | |
|---|---|---|---|---|---|
| | | File 1 | File 2 | File 3 | File 4 |
| SUBJECTS | User A | Own<br>Read<br>Write | | Own<br>Read<br>Write | |
| | User B | Read | Own<br>Read<br>Write | Write | Read |
| | User C | Read<br>Write | Read | | Own<br>Read<br>Write |

*Access Control Matrix*

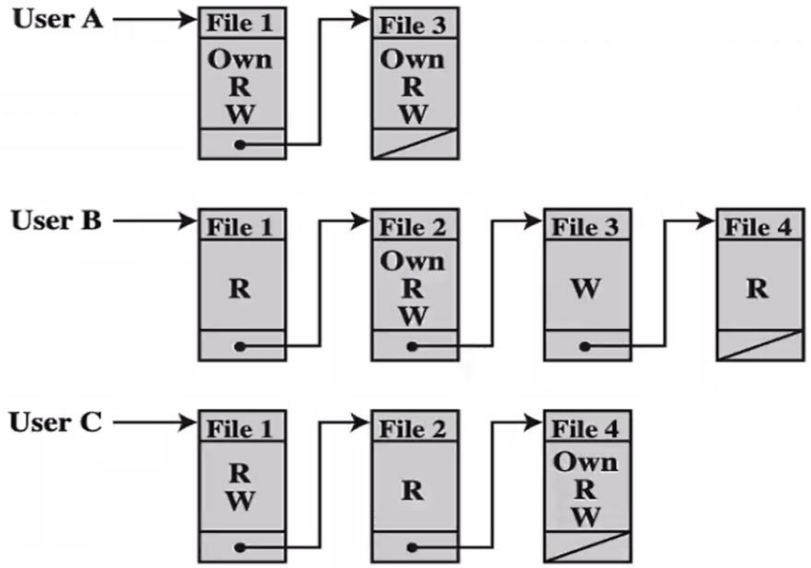
***What are some issues with this?***

# Access Control

**who** can do **what** to **whom?**



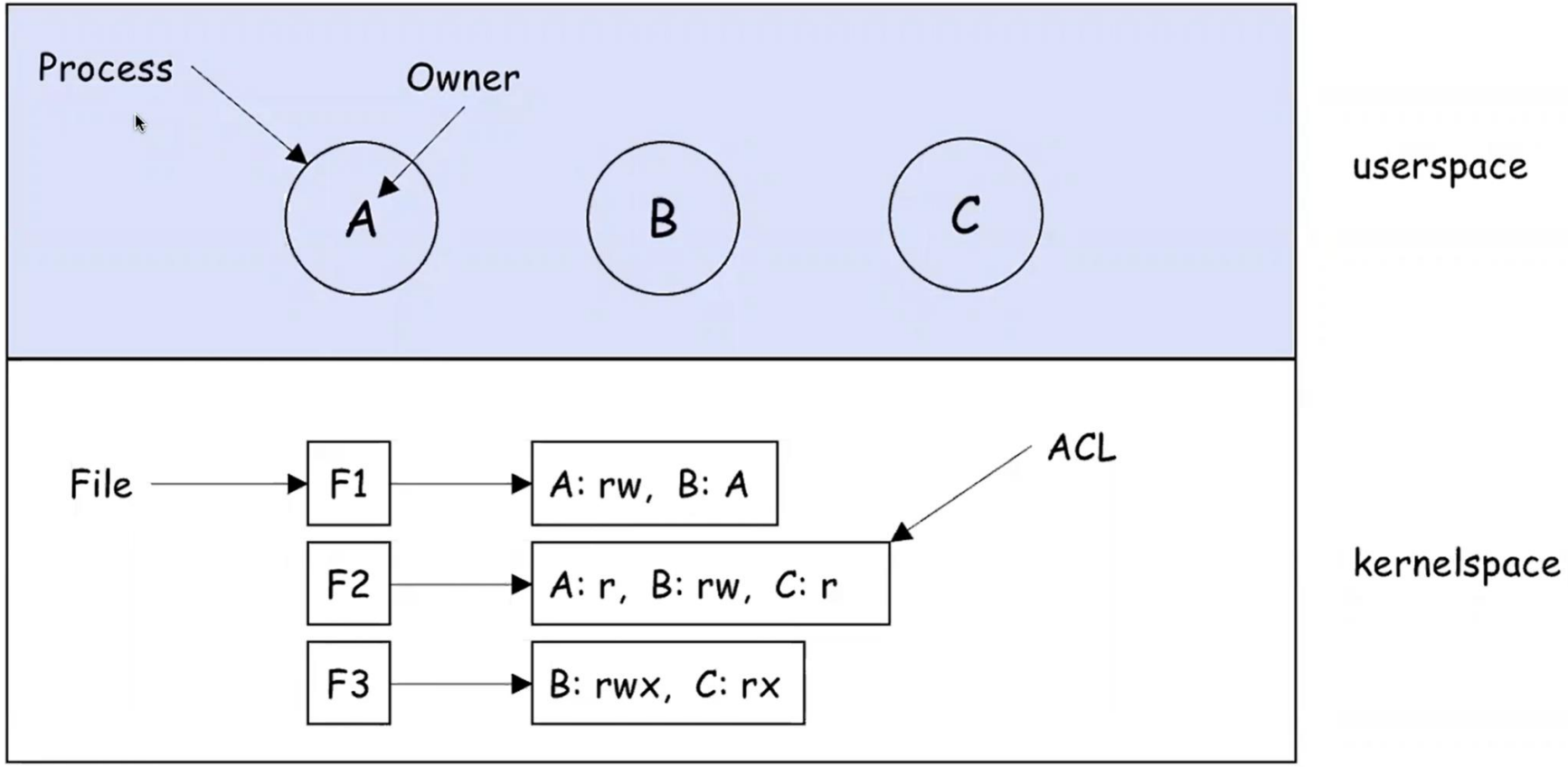

*Access Control list (ACL)*




Wont take up as much memory!

# Access Control

## **who** can do **what** to **whom?**



# Unix File Modes and Permissions

Every Unix file has a set of permissions that determine whether someone can read, write, or run the file

```
ls -l ~
ls -l /dev
```

```
[09/13/22]seed@VM:~$ ls -l ~
total 44
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Desktop
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Documents
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Downloads
drwxrwxr-x 2 seed seed 4096 Sep  1 14:37 lab0
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Music
drwxrwxr-x 2 seed seed 4096 Sep  6 15:23 os-review
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Pictures
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Public
drwxrwxr-x 2 seed seed 4096 Aug 25 13:41 shared
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Templates
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Videos
```

# Unix File Modes and Permissions

Every Unix file has a set of permissions that determine whether someone can read, write, or run the file

```
ls -l ~
ls -l /dev
```

```
[09/13/22]seed@VM:~$ ls -l ~
total 44
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Desktop
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Documents
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Downloads
drwxrwxr-x 2 seed seed 4096 Sep  1 14:37 lab0
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Music
drwxrwxr-x 2 seed seed 4096 Sep  6 15:23 os-review
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Pictures
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Public
drwxrwxr-x 2 seed seed 4096 Aug 25 13:41 shared
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Templates
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Videos
```

Permissions for the file

# Unix File Modes and Permissions

Every Unix file has a set of permissions that determine whether someone can read, write, or run the file

```
ls -l ~
ls -l /dev
```

Owner/group information

Permissions for the file

```
[09/13/22]seed@VM:~$ ls -l ~
total 44
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Desktop
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Documents
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Downloads
drwxrwxr-x 2 seed seed 4096 Sep  1 14:37 lab0
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Music
drwxrwxr-x 2 seed seed 4096 Sep  6 15:23 os-review
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Pictures
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Public
drwxrwxr-x 2 seed seed 4096 Aug 25 13:41 shared
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Templates
drwxr-xr-x 2 seed seed 4096 Nov 24  2020 Videos
```

# Unix File Modes and Permissions

Every Unix file has a set of permissions that determine whether someone can read, write, or run the file

```
$ ls -l file
-rw-r--r-- owner group date/time file
```

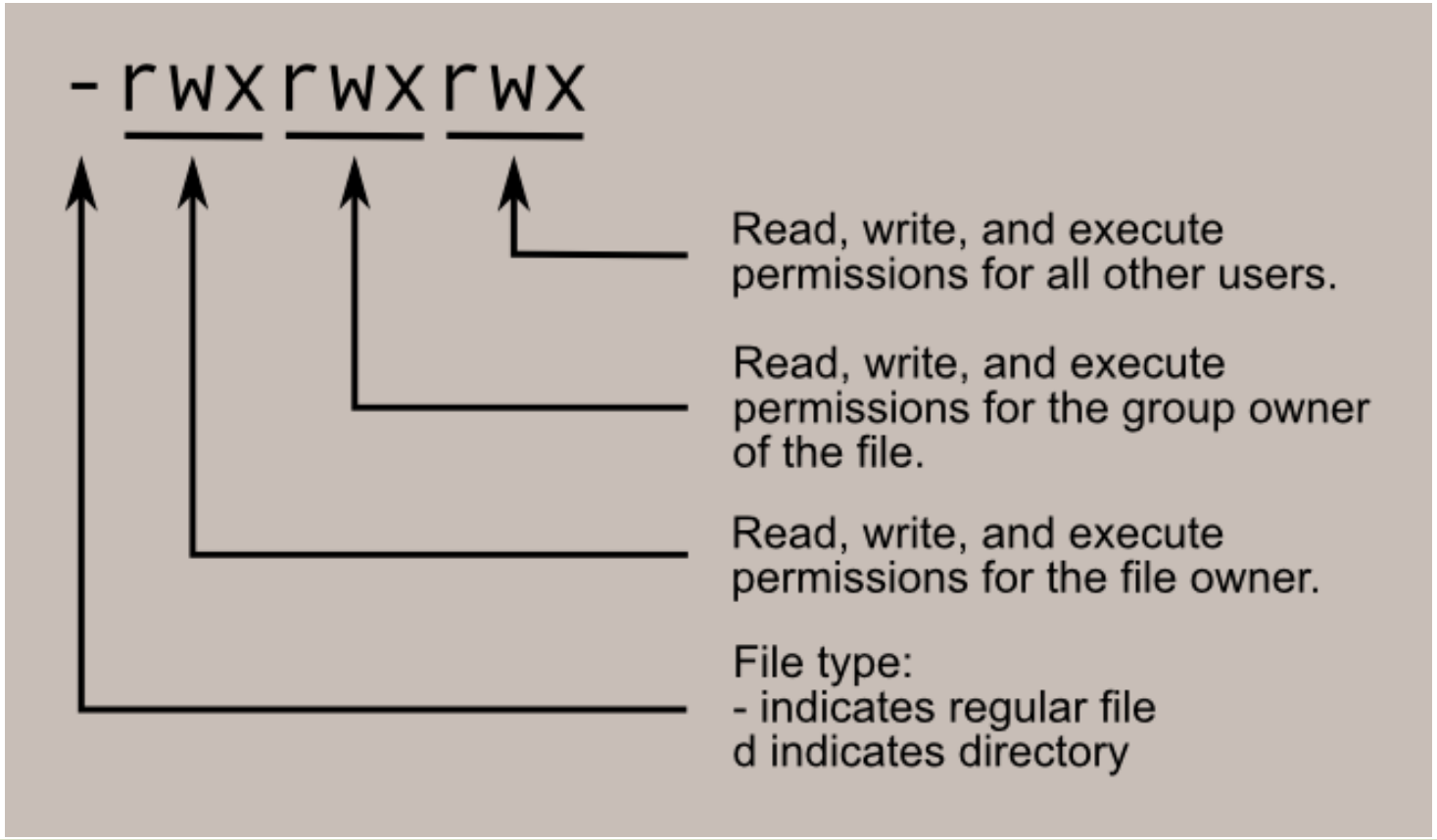
File permissions (4 parts)

- [file type][user][group][other]

# Unix File Modes and Permissions

File permissions (4 parts)

- [file type][user][group][other]



# Unix File Modes and Permissions

- [file type][user][group][other]

Suppose you have the following file:

```
$ ls -l F
-rw-rw-r--  B  G  ...  F
```

If user **A** asks to perform some operation **O** on a file object **F**, the OS checks:

- Is **A** the owner of **F**?

# Unix File Modes and Permissions

- [file type][user][group][other]

Suppose you have the following file:


If user **A** asks to perform some operation **O** on a file object **F**, the OS checks:

- Is **A** the owner of **F**?

  No, B is the owner

# Unix File Modes and Permissions

- [file type][user][group][other]

Suppose you have the following file:

```
$ ls -l F
-rw-rw-r--  B  G  ...  F
```

If user **A** asks to perform some operation **O** on a file object **F**, the OS checks:

- Is **A** the owner of **F**?
- Is **A** a member of **F**'s group?

# Unix File Modes and Permissions

- [file type][user][group][other]

Suppose you have the following file:


If user **A** asks to perform some operation **O** on a file object **F**, the OS checks:

- Is **A** the owner of **F**?
- Is **A** a member of **F**'s group? Suppose G = {B,C,F}

  A is not in F's group

# Unix File Modes and Permissions

- [file type][user][group][other]

Suppose you have the following file:

```
$ ls -l F
-rw-rw-r--  B  G  ...  F
```

If user **A** asks to perform some operation **O** on a file object **F**, the OS checks:

- Is **A** the owner of **F**?
- Is **A** a member of **F**'s group?
- Otherwise, what can they do?

# Unix File Modes and Permissions

- [file type][user][group][other]

Suppose you have the following file:


If user **A** asks to perform some operation **O** on a file object **F**, the OS checks:

- Is **A** the owner of **F**?
- Is **A** a member of **F**'s group?
- Otherwise, what can they do?

Everyone can **read** file F

# Unix File Modes and Permissions

- [file type][user][group][other]

Suppose **user C** asks to **execute** a **file object F2**. Will they be able to do so?

```
$ ls -l F
-rwxrwxrwx   B H   ...   F1
-rwxr-xr--   D G   ...   F2
-rw-r-----   D H   ...   F3
-rw-rw-rw-   B G   ...   F4
```

Note:

- Group = G = {A, C, K, M, Q, Z}
- Group = H = {A, B, C, Q}

# Unix File Modes and Permissions

- [file type][user][group][other]

Suppose **user C** asks to **execute** a **file object F2**. Will they be able to do so?

```
$ ls -l F
-rwxrwxrwx   B H   ...   F1
-rwxr-xr--   D G   ...   F2
-rw-r-----   D H   ...   F3
-rw-rw-rw-   B G   ...   F4
```

Note:

- Group = G = {A, C, K, M, Q, Z}
- Group = H = {A, B, C, Q}

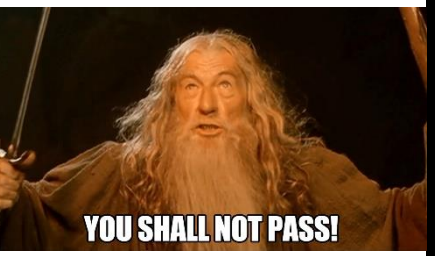
# Limitations of File-Based Access Control

When would a non-privilege user require more power/permissions?

# Limitations of File-Based Access Control

When would a non-privilege user require more power/permissions?

Changing password!

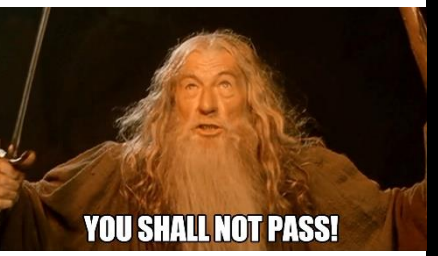



```
[seed@VM][~]$ ls -al /etc/passwd
-rw-r--r-- 1 root root 2886 Nov 24 09:12 /etc/passwd

[seed@VM][~]$ ls -al /etc/shadow
-rw-r----- 1 root shadow 1514 Nov 24 09:12 /etc/shadow
```

# Limitations of File-Based Access Control

When would a non-privilege user require more power/permissions?

Changing password!



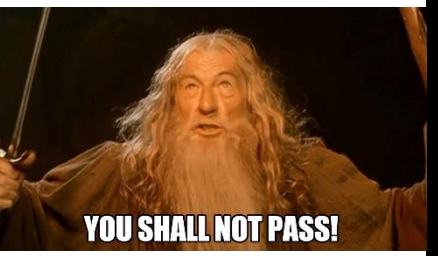

```
[seed@VM][~]$ ls -al /etc/passwd
-rw-r--r-- 1 root root 2886 Nov 24 09:12 /etc/passwd

[seed@VM][~]$ ls -al /etc/shadow
-rw-r----- 1 root shadow 1514 Nov 24 09:12 /etc/shadow
```

**`/etc/passwd`** and **`/etc/shadow`** hold encrypted passwords for the user, in order to change our password, we will need to have access to those directories

# Limitations of File-Based Access Control

When would a non-privilege user require more power/permissions?

Changing password!



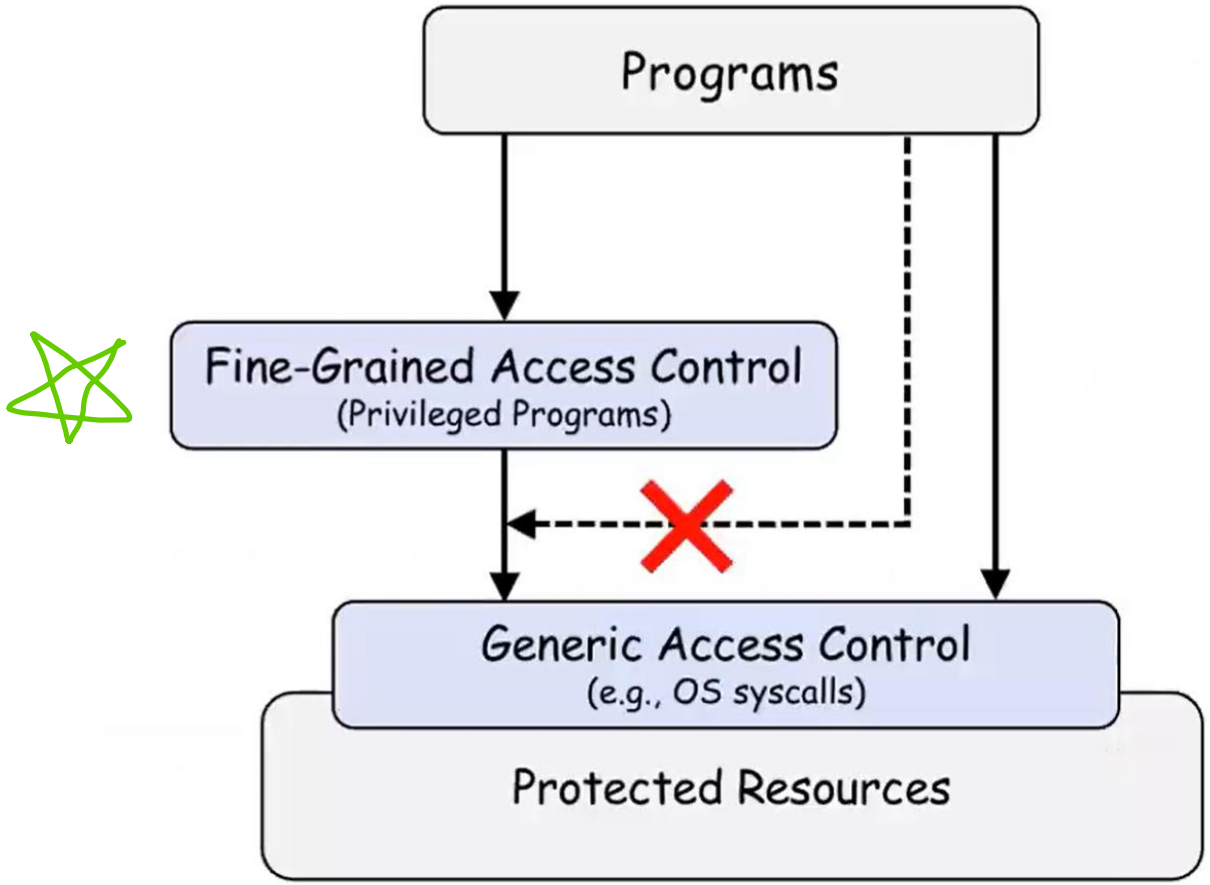

```
[seed@VM][~]$ ls -al /etc/passwd
-rw-r--r-- 1 root root 2886 Nov 24 09:12 /etc/passwd

[seed@VM][~]$ ls -al /etc/shadow
-rw-r----- 1 root shadow 1514 Nov 24 09:12 /etc/shadow
```

**`/etc/passwd`** and **`/etc/shadow`** hold encrypted passwords for the user, in order to change our password, we will need to have access to those directories

*root (aka admin) is the only person that has write permissions!*

# Limitations of File-Based Access Control

Instead of having a user deal with sensitive actions, lets have a privileged program do it for us!



# Types of Privileged Programs

- **Daemons**
  - Computer program that runs in the background
  - Needs to run as root or other privileged users

- **Set-UID Programs**
  - Widely used in UNIX systems
  - A normal program… but marked with a special bit

# The superman story

Superman got tired of saving the city every day

So, he decided to create a “super suit” that would give normal people his powers

**Problem:** Not all super people are good……

# The superman story

Superman got tired of saving the city every day

So, he decided to create a “super suit” that would give normal people his powers

**Problem:** Not all super people are good……

## Super suit 2.0

Super suit with a dope computer
Programmed to perform a specific task
No way to deviate from the pre-programmed task

# The superman story


Task: Stop Bowser

1. Fly North
2. Turn left and move forward
3. Punch

Super suit 2.0

People can hop in, and do the specific task to stop bowser

# The superman story


Task: Stop Bowser

1. Fly North
2. Turn left and move forward
3. Punch

# The superman story


Task: Stop Bowser

1. Fly North
2. Turn left and move forward
3. Punch

# The superman story


Task: Stop Bowser

1. Fly North
2. Turn left and move forward
3. Punch

# The superman story


Task: Stop Bowser

1. Fly North
2. Turn left and move forward
3. Punch

# The superman story


Task: Stop Bowser

1. Fly North
2. Turn left and move forward
3. Punch

This works great! People can only do the predetermined task and don’t have control!

# The superman story


Task: Stop Bowser

1. Fly North
2. Turn left and move forward
3. Punch

This works great! People can only do the predetermined task and don’t have control!

**Exploitable?**

# The superman story


Task: Stop Bowser

1. Fly North
2. Turn left and move forward
3. Punch

Suppose I come along,
and I see the power suit

And I decide to flip the suit around

Now what happens???

# The superman story


Task: Stop Bowser

1. Fly North
2. Turn left and move forward
3. Punch

Suppose I come along,
and I see the power suit

And I decide to flip the suit around

Now what happens???

# The superman story


Task: Stop Bowser

1. Fly North
2. Turn left and move forward
3. Punch

Suppose I come along,
and I see the power suit

And I decide to flip the suit around

Now what happens???

# The superman story


Task: Stop Bowser

1. Fly North
2. Turn left and move forward
3. Punch

Suppose I come along,
and I see the power suit

And I decide to flip the suit around

Now what happens???

# The superman story


Task: Stop Bowser

1. Fly North
2. Turn left and move forward
3. Punch

Suppose I come along,
and I see the power suit

And I decide to flip the suit around

Now what happens???

# The superman story

Task: Stop Bowser

1. Fly North
2. Turn left and move forward
3. Punch

Suppose I come along,
and I see the power suit

And I decide to flip the suit around

**I still followed the steps, but now we have a totally different outcome**

My plan was to rob the bank, and I had friends waiting this whole time!

# Set-UID In a Nutshell

**Set-UID** allows a user to run a program with the program owner's privilege

- User runs a program w/ temporarily elevated privileges

Created to deal with inflexibilities of UNIX access control

Example: The **`passwd`** program

```
[seed@VM][~]$ ls -al /usr/bin/passwd
-rwsr-xr-x 1 root root 68208 May 28  2020 /usr/bin/passwd
```

# Set-UID In a Nutshell

**Set-UID** allows a user to run a program with the program owner's privilege
- User runs a program w/ temporarily elevated privileges

Every process has two User IDs
- Real UID (RUID)– Identifies the **owner** of the process
- Effective UID (EUID)– Identifies **current privilege** of the process

When a normal program is executed
- RUID == EUID

When a Set-UID program is executed
- RUID != EUID
- EUID == ID of the program's owner

**If a program owner == root,**
**The program runs with root privileges**

# Set-UID In a Nutshell

**Set-UID** allows a user to run a program with the program owner's privilege
- User runs a program w/ temporarily elevated privileges

Every process has two User IDs
- Real UID (RUID)– Identifies the **owner** of the process
- Effective UID (EUID)– Identifies **current privilege** of the process

When a normal program is executed
- RUID == EUID

When a Set-UID program is executed
- RUID != EUID
- EUID == ID of the program's owner

**If a program owner == root,**
**The program runs with root privileges**

# Set-UID Program Demo

```
[seed@VM][~]$ cp /bin/cat ./mycat
[seed@VM][~]$ sudo chown root mycat
[seed@VM][~]$ ls -al mycat
-rwxr-xr-x 1 root seed 43416 Jan 25 21:15 mycat
```

**Change the owner** of a file to root

# Set-UID Program Demo

```
[seed@VM][~]$ cp /bin/cat ./mycat
[seed@VM][~]$ sudo chown root mycat
[seed@VM][~]$ ls -al mycat
-rwxr-xr-x 1 root seed 43416 Jan 25 21:15 mycat
```

**Change the owner** of a file to root

```
[seed@VM][~]$ mycat /etc/shadow
mycat: /etc/shadow: Permission denied
```

Running to program (normally)

# Set-UID Program Demo

```
[seed@VM][~]$ cp /bin/cat ./mycat
[seed@VM][~]$ sudo chown root mycat
[seed@VM][~]$ ls -al mycat
-rwxr-xr-x 1 root seed 43416 Jan 25 21:15 mycat
```

**Change the owner** of a file to root

```
[seed@VM][~]$ mycat /etc/shadow
mycat: /etc/shadow: Permission denied
```

Running to program (normally)

```
[seed@VM][~]$ sudo chmod 4755 mycat
[seed@VM][~]$ ls -al mycat
-rwsr-xr-x 1 root seed 43416 Jan 25 21:15 mycat
[seed@VM][~]$ mycat /etc/shadow
root:!:18590:0:99999:7:::
daemon:*:18474:0:99999:7:::
```

**Enable the Set-UID bit**

We have successfully made a Set-UID program!

# Announcements

Lab 1 Due **TOMORROW** 9/16 @ 11:59 PM

- Shouldn’t be too bad

Lab 2 (SET-UID) is posted. due ???

- You will be able to complete it after today

# Set-UID

Demo

A Set-UID program is just like any other program, except that is has a special bit sit

```
[09/15/22]seed@VM:~/lab2$ cp /usr/bin/id ./myid
[09/15/22]seed@VM:~/lab2$ chown root myid
chown: changing ownership of 'myid': Operation not permitted
[09/15/22]seed@VM:~/lab2$ sudo chown root myid
[09/15/22]seed@VM:~/lab2$ /myid
bash: /myid: No such file or directory
[09/15/22]seed@VM:~/lab2$ ./myid
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
```

*If the set-uidbit is enabled, the EUID is set according to the file owner*

```
[09/15/22]seed@VM:~/lab2$ chmod 4755 myid
chmod: changing permissions of 'myid': Operation not permitted
[09/15/22]seed@VM:~/lab2$ sudo chmod 4755 myid
[09/15/22]seed@VM:~/lab2$ ./myid
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
```

*Access control decisions made based on EUID, not RUID !*

**Steps for creating a set-uid program**

1. Change file ownership to root (`chown`)

2. Enable to Set-uid bit (`chmod`)

4755

4 = setuid bit

755 = owner r/w/x, group/others can r/w

# So…. Is Set-UID secure?

- Allows normal users to escalate privileges
  - This is different from directly giving escalated privileges (such as **`sudo`**)
  - Restricted behavior (think **power suit 2.0** )

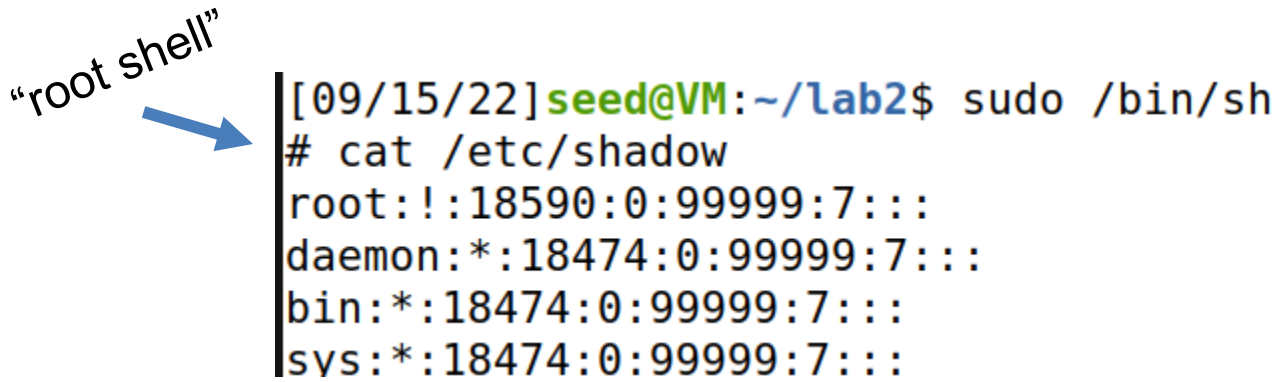
Are there any programs that **should not** be Set-UID programs?

# So…. Is Set-UID secure?

- Allows normal users to escalate privileges
- ➢ This is different from directly giving escalated privileges (such as **`sudo`**)
- ➢ Restricted behavior (think **power suit 2.0** )

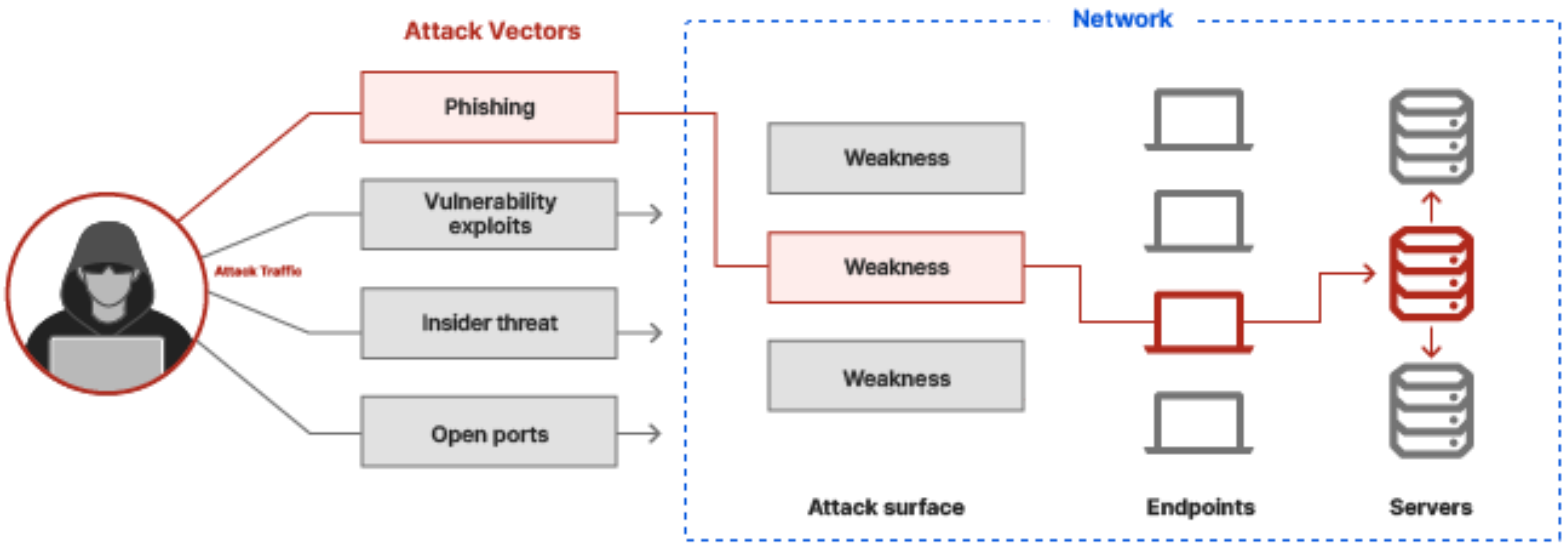
Are there any programs that **should not** be Set-UID programs?

`/bin/sh`

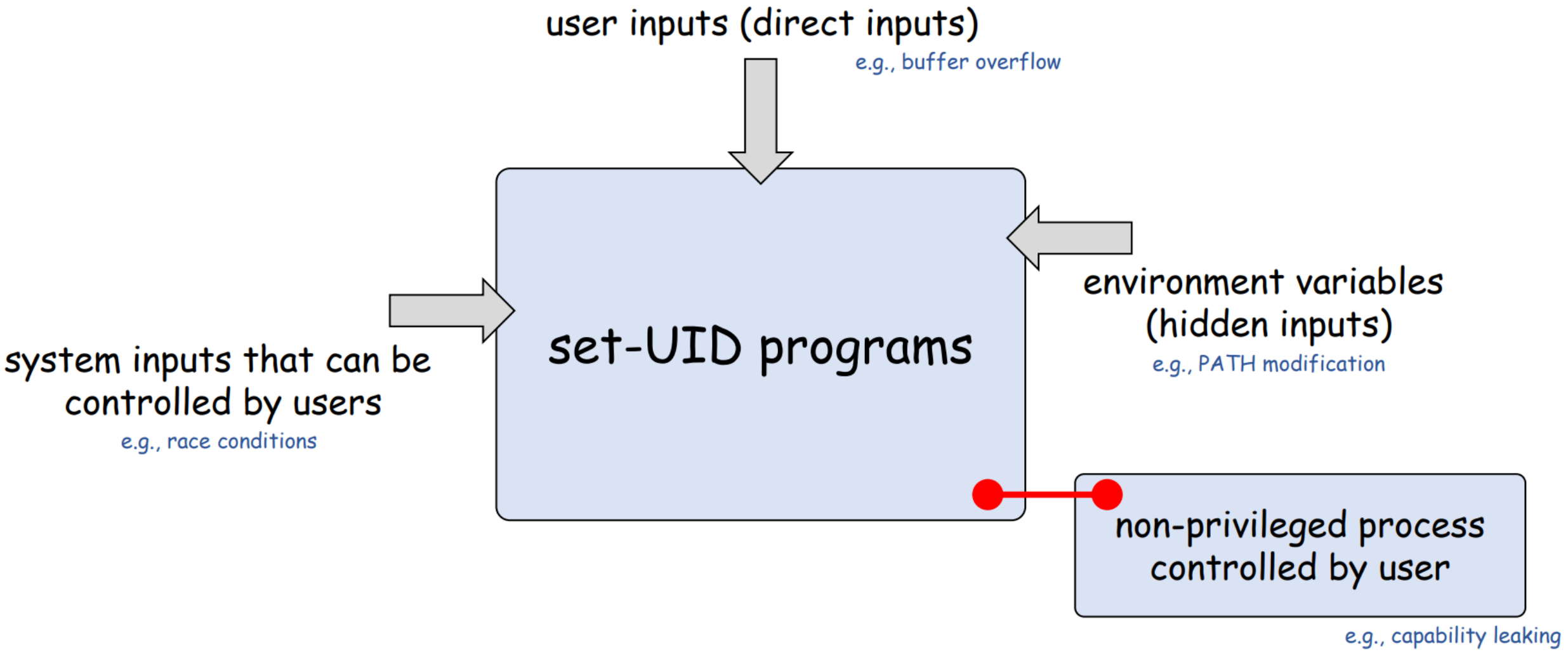


# Attack Surface of (Set-UID) Programs

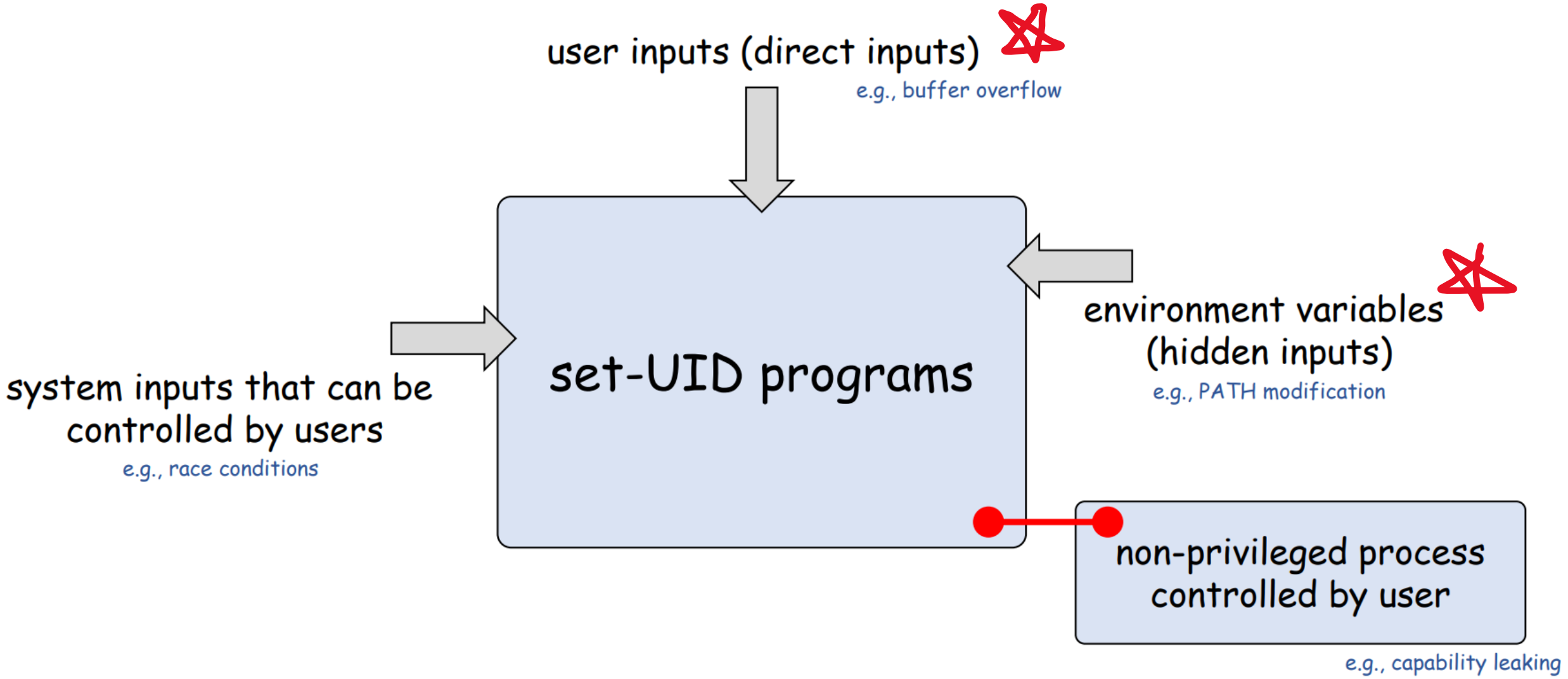
An **attack surface** is the aggregation of all exposed entry points/weaknesses into the system to gain unauthorized access



# Attack Surface of (Set-UID) Programs



# Attack Surface of (Set-UID) Programs



# Invoking Programs from within programs


# Preliminary setup for attack

`/bin/sh` is an alias for `/bin/dash`. `/bin/dash` has countermeasures for some of our attacks

We will need to run a command to set the unsafe version of shell

```
[seed@VM][~]$ sudo ln -sf /bin/zsh /bin/sh   # set shell to zsh (no countermeasure)
[seed@VM][~]$ sudo ln -sf /bin/dash /bin/sh  # set shell to dash (has countermeasure)
```

# Invoking Program with a program

We can invoke external commands/programs from INSIDE another program

- `system()`
- `exec()-family`

# System()

usage: **system**(*command*)

- Spawns a new process that executes the shell command that is specified in *command*

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    printf("I am going to start the calculator program! \n");
    system("/bin/bc");
}
```

# audit.c

usage: **system**(*command*)

- Suppose you are preparing for an audit. An auditor may need the access to view certain files.
- We will create a privileged program that will let the auditor view the content some file

```
./audit company_data.csv
```

```
./audit ../lab0/solution.docx
```

# audit.c

usage: **system**(*command*)

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char *v[3];

    if (argc < 2) {
        printf("Audit! Please type a file name.\n");
        return 1;
    }

    v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = 0;
    char *command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
    sprintf(command, "%s %s", v[0], v[1]);

    /*
     * Use only one of the following (comment out the other):
     */

    system(command);
    //execve(v[0], v, 0);

    return 0;
}
```

*The command line argument (file path) is appended to the string "/bin/cat"*

Spawns a new process that executes

**/bin/cat [FILE_PATH]**

ex. `/bin/cat my_file.txt`

# audit.c

usage: **system**(*command*)

- Suppose you are preparing for an audit. An auditor may need the access to view certain files.
- We will create a privileged program that will let the auditor view the content some file

```
./audit company_data.csv
```



Command that is executed:

```
/bin/cat company_data.csv
```

```
./audit ../lab0/solution.docx
```



```
/bin/cat ../lab0/solution.docx
```

# audit.c

usage: **system**(*command*)

- Suppose you are preparing for an audit. An auditor may need the access to view certain files.
- We will create a privileged program that will let the auditor view the content some file

```
./audit company_data.csv
```



Command that is executed:

```
/bin/cat company_data.csv
```

```
./audit ../lab0/solution.docx
```



```
/bin/cat ../lab0/solution.docx
```

**We have some control over the behavior of the program**

# audit.c

usage: **system**(*command*)

- Suppose you are preparing for an audit. An auditor may need the access to view certain files.
- We will create a privileged program that will let the auditor view the content some file

```
./audit company_data.csv
```



Command that is executed:

```
/bin/cat company_data.csv
```

```
./audit ../lab0/solution.docx
```



```
/bin/cat ../lab0/solution.docx
```

**We have some control over the behavior of the program**

If this is a Set-UID program…. things could get interesting

# audit.c

usage: **system**(*command*)

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char *v[3];

    if (argc < 2) {
        printf("Audit! Please type a file name.\n");
        return 1;
    }

    v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = 0;
    char *command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
    sprintf(command, "%s %s", v[0], v[1]);

    /*
     * Use only one of the following (comment out the other):
     */

    system(command);
    //execve(v[0], v, 0);

    return 0;
}
```

System is a very unsafe function

What type of input could we provide to exploit this?

# audit.c

usage: **system**(*command*)

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char *v[3];

    if (argc < 2) {
        printf("Audit! Please type a file name.\n");
        return 1;
    }

    v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = 0;
    char *command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
    sprintf(command, "%s %s", v[0], v[1]);

    /*
     * Use only one of the following (comment out the other):
     */

    system(command);
    //execve(v[0], v, 0);

    return 0;
}
```

System is a very unsafe function

What type of input could we provide to exploit this?

*hint*: the string passed into system can include multiple commands

# audit.c

usage: **system**(*command*)

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char *v[3];

    if (argc < 2) {
        printf("Audit! Please type a file name.\n");
        return 1;
    }

    v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = 0;
    char *command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
    sprintf(command, "%s %s", v[0], v[1]);

    /*
     * Use only one of the following (comment out the other):
     */

    system(command);
    //execve(v[0], v, 0);

    return 0;
}
```

System is a very unsafe function

What type of input could we provide to exploit this?

*hint*: the string passed into system can include multiple commands

**./audit "my_info.txt; /bin/sh"**

audit.c

usage: **system**(*command*)

**./audit "my_info.txt; /bin/sh"**

**system**(/bin/cat "my_info.txt; /bin/sh")

audit.c

usage: **system**(*command*)

**./audit "my_info.txt; /bin/sh"**

**system**(/bin/cat "my_info.txt; /bin/sh")

```
[09/15/22]seed@VM:~/lab2$ ./audit "my_info.txt; /bin/sh"
I have some information
#
```

*System() interprets this as two different commands*

usage: **system**(*command*)

**./audit "my_info.txt; /bin/sh"**

**system**(/bin/cat my_info.txt; /bin/sh)

```
[09/15/22]seed@VM:~/lab2$ ./audit "my_info.txt; /bin/sh"
I have some information
#
```

*System() interprets this as two different commands*

usage: **system**(*command*)

**./audit "my_info.txt; /bin/sh"**

**system**(/bin/cat my_info.txt; /bin/sh)

```
[09/15/22]seed@VM:~/lab2$ ./audit "my_info.txt; /bin/sh"
I have some information
# whoami
root
# cat /etc/shadow
root:!:18590:0:99999:7:::
daemon:*:18474:0:99999:7:::
bin:*:18474:0:99999:7:::
```

Because this is a Set-UID program.
When owner = root, the shell will be run with root permissions

audit.c

usage: **system**(*command*)

**./audit "my_info.txt; /bin/sh"**

**system**(/bin/cat my_info.txt; /bin/sh)

```
[09/15/22]seed@VM:~/lab2$ ./audit "my_info.txt; /bin/sh"
I have some information
# whoami
root
# cat /etc/shadow
root:!:18590:0:99999:7:::
daemon:*:18474:0:99999:7:::
bin:*:18474:0:99999:7:::
```

Because this is a Set-UID program.
When owner = root, the shell will be run with root permissions

**We have gained access into the system!**

# A safer way to invoke programs

```
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

`execve()` executes the program referred to by *`pathname`*. `argv[]` is the command line arguments for the command

# A safer way to invoke programs

```
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

`execve()` executes the program referred to by *`pathname`*. `argv[]` is the command line arguments for the command

# A safer way to invoke programs

```
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

`execve()` executes the program referred to by *`pathname`*. `argv[]` is the command line arguments for the command

Using `execve()` instead of `system()`

```
[09/15/22]seed@VM:~/lab2$ ./audit "aa;/bin/sh"
/bin/cat: 'aa;/bin/sh': No_such file or directory
```

**Fail!**

# A safer way to invoke programs

```
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

`execve()` executes the program referred to by *`pathname`*. `argv[]` is the command line arguments for the command

`execve("/bin/cat",["aa;/bin/sh"])`

`/bin/cat "aa;/bin/sh"`

*Treated as an entire argument to the command*

**Fail!**

Using `execve()` instead of `system()`

```
[09/15/22]seed@VM:~/lab2$ ./audit "aa;/bin/sh"
/bin/cat: 'aa;/bin/sh': No_such file or directory
```

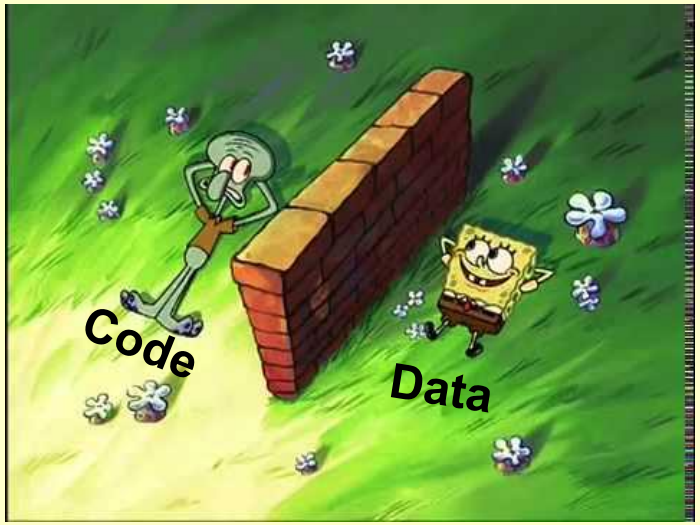
**Fail!**

What makes this unsafe? Why was this program exploitable?

# Principle of Isolation

There needs to be a clear separation of **data** and **code**

If user input needs to be treated as data, NONE of the contents should be treated as code

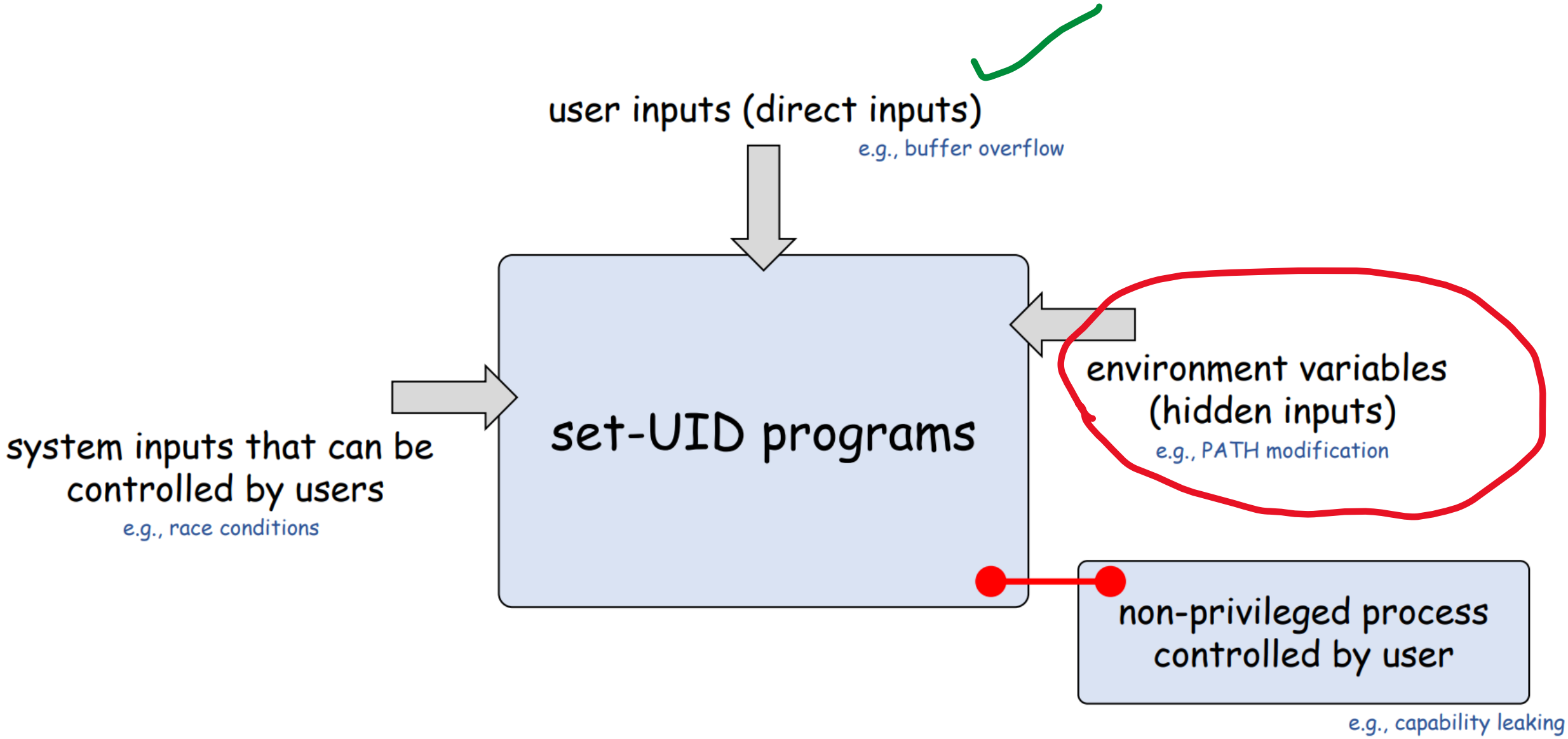


*The ability (and risks) of invoking external commands is not limited to C*

Python has a `system` call…

Perl has `open()`

PHP has `system`


user inputs (direct inputs)
e.g., buffer overflow
system inputs that can be controlled by users
e.g., race conditions
set-UID programs
environment variables (hidden inputs)
e.g., PATH modification
non-privileged process controlled by user
e.g., capability leaking

**Environment variable** are a set of dynamic named values that affect the way a running process will behave *(key-value pairs)*

**Environment variable** are a set of dynamic named values that affect the way a running process will behave *(key-value pairs)*

Example: The PATH variable

- We use command such as `ls` and `passwd`

We could be in any directory; how does it know to run /bin/ls ?

**Environment variable** are a set of dynamic named values that affect the way a running process will behave *(key-value pairs)*

Example: The PATH variable

- We use command such as `ls` and `passwd`

We could be in any directory; how does it know to run /bin/ls ?

If the full path is not provided, the shell process will use the PATH env. Variable to search for it!

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:.
```

**Environment variable** are a set of dynamic named values that affect the way a running process will behave *(key-value pairs)*

Example: The PATH variable

- We use command such as `ls` and `passwd`

We could be in any directory; how does it know to run /bin/ls ?

If the full path is not provided, the shell process will use the PATH env. Variable to search for it!

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:.
```

You can run the **`env`** command to print out all the environment variables

# Where do Env Variables come from?

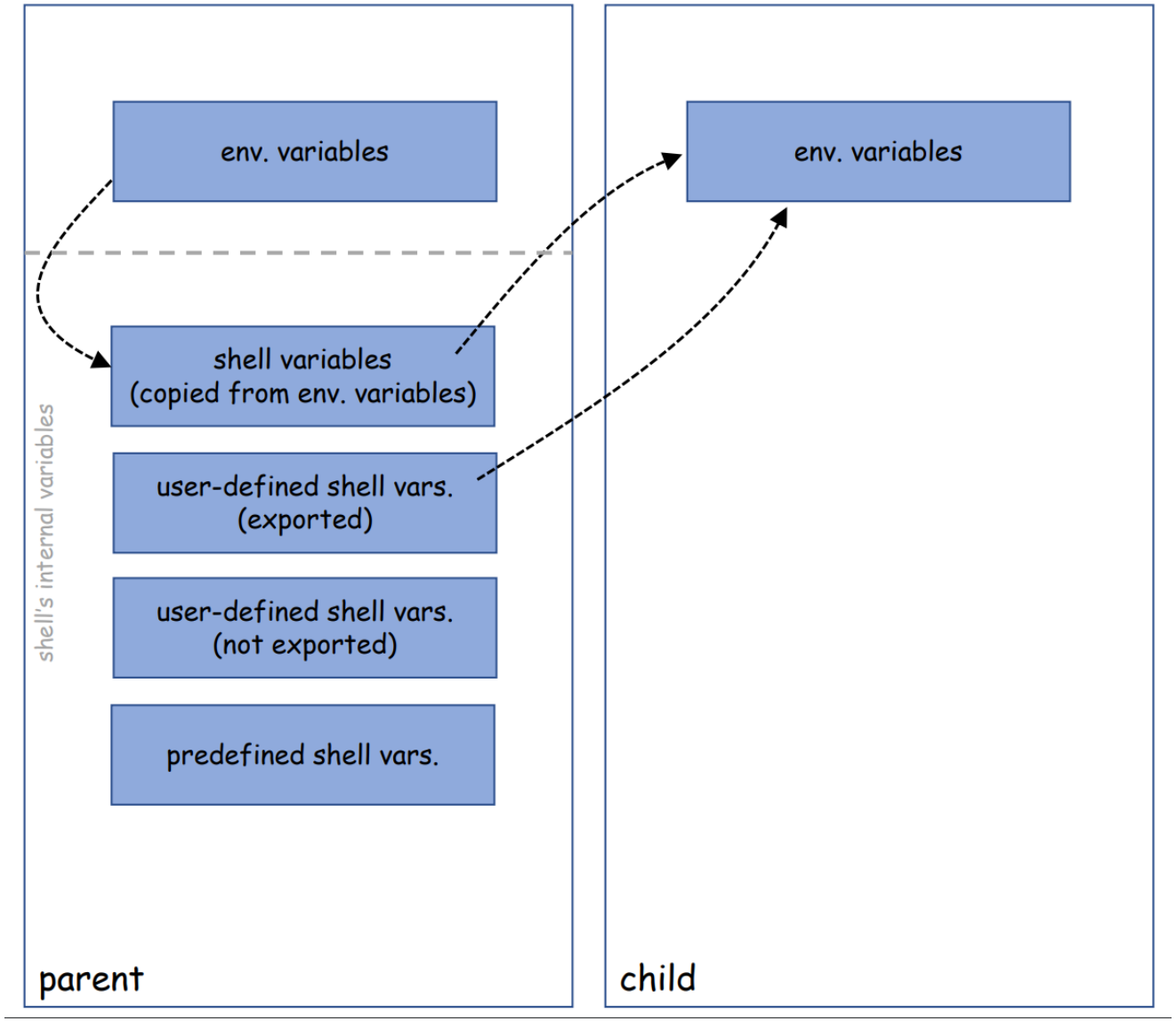
Processes can get environment variables in one of two ways:

**fork()** → the child process inherits its parent process's environment variables.
**exec()** → the memory space is overwritten, and all old environment variables are lost. However, execve() can explicitly pass environment variables from one process to another

```
./passenv 1
./passenv 2
./passenv 3
```

```c
#include <stdio.h>
#include <unistd.h>

extern char ** environ;
void main(int argc, char* argv[], char* envp[])
{
    int i = 0; char* v[2]; char* newenv[3];
    if (argc < 2) return;

    // Construct the argument array
    v[0] = "/usr/bin/env";   v[1] = NULL;

    // Construct the environment variable array
    newenv[0] = "AAA=aaa"; newenv[1] = "BBB=bbb"; newenv[2] = NULL;

    switch(argv[1][0]) {
    case '1': // Passing no environment variable.
        execve(v[0], v, NULL);
    case '2': // Passing a new set of environment variables.
        execve(v[0], v, newenv);
    case '3': // Passing all the environment variables.
        execve(v[0], v, environ);
    default:
        execve(v[0], v, NULL);
    }
}
```

# Where do Env Variables come from?

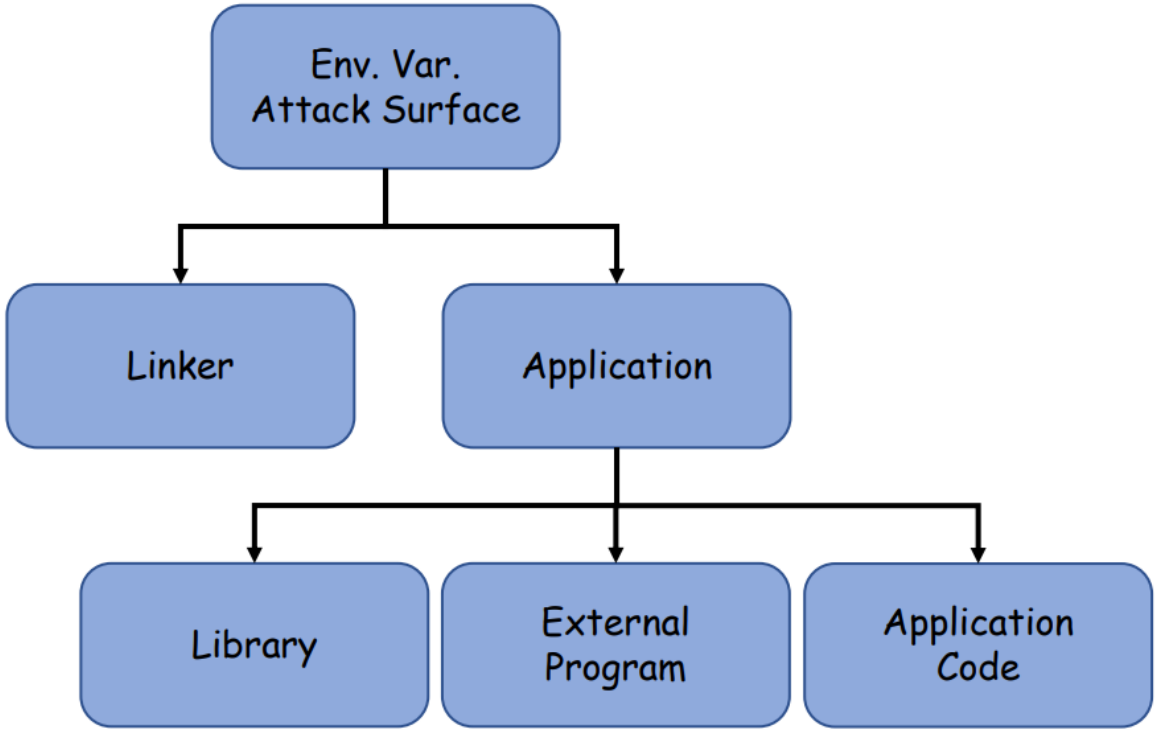


# Attacks with Env Variables

- Hidden usage of environment variable is part of what makes them so dangerous

- Users can also modify environment variables…

  - If Set-UID programs make use of environment variables, they become part of the attack surface



# Attacks with Env Variables

- A program may invoke an external program (e.g., via system()) to do some work
- **PATH** contains a list of directories to search for executable programs
- If a program is invoked without using the absolute path (e.g., system("ls"), the **PATH** env. variable is used to search for the program
- **PATH** can be set by users….

Any ideas???

# Attacks with Env Variables

- A program may invoke an external program (e.g., via system()) to do some work
- **PATH** contains a list of directories to search for executable programs
- If a program is invoked without using the absolute path (e.g., system("ls"), the **PATH** env. variable is used to search for the program
- **PATH** can be set by users….

Any ideas???

*Task 6 in Lab 2* ☺

# Attacks with Env Variables

```
[seed@VM][~]$ sudo ln -sf /bin/zsh /bin/sh   # set shell to zsh (no countermeasure)
[seed@VM][~]$ sudo ln -sf /bin/dash /bin/sh  # set shell to dash (has countermeasure)
```

Compile and set as Set-UID program

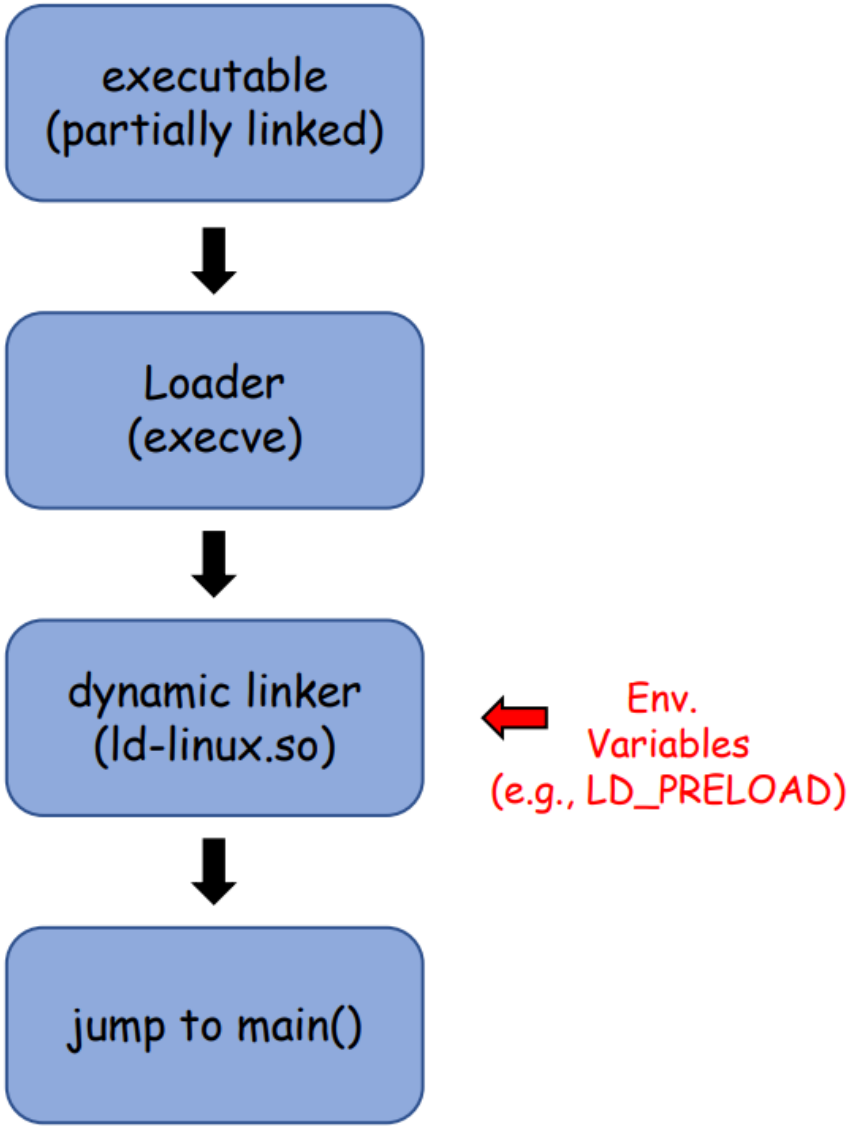
```
[seed@VM][~]$ gcc -o ls_vul ls_vul.c
[seed@VM][~]$ sudo chown root ls_vul
[seed@VM][~]$ sudo chmod 4755 ls_vul
[seed@VM][~]$ ls -al ls_vul
...

Q: How to get ls_vul to run attacker code for ls instead of /bin/ls program?!?!

[seed@VM][~]$ export PATH=/home/seed:$PATH # set PATH to look in seed's home dir first...
[seed@VM][~]$ echo $PATH
...

...and now...
```

**export** is used to define new variables

- **Linking** finds the external library code referenced in a program

- **Static linking** – linker combines program code/external code into final executable

- **Dynamic Linking**- linker uses env. Variables to locate external dependencies (increase the attack surface)




How does the linker know where to look?

- **LD_PRELOAD** contains a list of shared libraries to search first
- Provides precedent over standard function calls (malloc, free, etc)
- If functions are not found, it will consult the location specified in **LD_LIBRARY_PATH**
- We can set both values, which gives us an opportunity for users to influence linking

Task 7

- **LD_PRELOAD** contains a list of shared libraries to search first
- Provides precedent over standard function calls (malloc, free, etc)
- If functions are not found, it will consult the location specified in **LD_LIBRARY_PATH**
- We can set both values, which gives us an opportunity for users to influence linking

```c
/* myprog.c */
#include <unistd.h>
int main()
{
  sleep(1);
  return 0;
}
```

This will run the standard libc sleep

mylib.c

```c
#include <stdio.h>
void sleep (int s)
{
  /* If this is invoked by a privile
  printf("I am not sleeping!\n");
}
```

Write our own sleep function within out mylib.c program

- **LD_PRELOAD** contains a list of shared libraries to search first
- Provides precedent over standard function calls (malloc, free, etc)
- If functions are not found, it will consult the location specified in **LD_LIBRARY_PATH**
- We can set both values, which gives us an opportunity for users to influence linking

```c
/* myprog.c */
#include <unistd.h>
int main()
{
  sleep(1);
  return 0;
}
```

This will run the standard libc sleep

mylib.c

```c
#include <stdio.h>
void sleep (int s)
{
  /* If this is invoked by a privile
  printf("I am not sleeping!\n");
}
```

Write our own sleep function within out mylib.c program

Add function to shared library

```
$ gcc -fPIC -g -c mylib.c
$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
```

- **LD_PRELOAD** contains a list of shared libraries to search first
- Provides precedent over standard function calls (malloc, free, etc)
- If functions are not found, it will consult the location specified in **LD_LIBRARY_PATH**
- We can set both values, which gives us an opportunity for users to influence linking

```c
/* myprog.c */
#include <unistd.h>
int main()
{
  sleep(1);
  return 0;
}
```

This will run the standard libc sleep

mylib.c

```c
#include <stdio.h>
void sleep (int s)
{
  /* If this is invoked by a privile
  printf("I am not sleeping!\n");
}
```

Write our own sleep function within out mylib.c program

Add function to shared library

```
$ gcc -fPIC -g -c mylib.c
$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
```

```
$ export LD_PRELOAD=./libmylib.so.1.0.1
```

Set variable to look at our library, not default one

- **LD_PRELOAD** contains a list of shared libraries to search first
- Provides precedent over standard function calls (malloc, free, etc)
- If functions are not found, it will consult the location specified in **LD_LIBRARY_PATH**
- We can set both values, which gives us an opportunity for users to influence linking

```c
/* myprog.c */
#include <unistd.h>
int main()
{
  sleep(1);
  return 0;
}
```

This will run the standard libc sleep

mylib.c

```c
#include <stdio.h>
void sleep (int s)
{
  /* If this is invoked by a privile
  printf("I am not sleeping!\n");
}
```

Write our own sleep function within out mylib.c program

Add function to shared library

```
$ gcc -fPIC -g -c mylib.c
$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
```

```
$ export LD_PRELOAD=./libmylib.so.1.0.1
```

Set variable to look at our library, not default one

./myprog.c **???**

- **LD_PRELOAD** contains a list of shared libraries to search first
- Provides precedent over standard function calls (malloc, free, etc)
- If functions are not found, it will consult the location specified in **LD_LIBRARY_PATH**
- We can set both values, which gives us an opportunity for users to influence linking

```c
/* myprog.c */
#include <unistd.h>
int main()
{
  sleep(1);
  return 0;
}
```

This will run the standard libc sleep

mylib.c

```c
#include <stdio.h>
void sleep (int s)
{
  /* If this is invoked by a privile
  printf("I am not sleeping!\n");
}
```

Write our own sleep function within out mylib.c program

Add function to shared library

```
$ gcc -fPIC -g -c mylib.c
$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
```

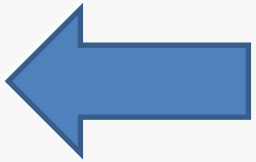
```
$ export LD_PRELOAD=./libmylib.so.1.0.1
```

Set variable to look at our library, not default one

./myprog.c **???**

```c
#include <fcntl.h>

int main()
{
    int fd;

    /*
     * Assume that /etc/zzz is an important system file,
     * and it is owned by root with permission 0644.
     * Before running this program, you should create
     * the file /etc/zzz first.
     */
    fd = open("/etc/zzz", O_RDWR | O_APPEND);
    if (fd == -1) {
        printf("Cannot open /etc/zzz\n");
        exit(0);
    }

    // Simulate the tasks conducted by the program
    sleep(1);
```

```c
     * After the task, elevated privileges are no longer needed;
     * it is time to relinquish these privileges!
     * NOTE: getuid() returns the real UID (RUID)
     */
    setuid(getuid());

    if (fork()) {  /* parent process */
        close (fd);
        exit(0);
    } else {  /* child process */

        /*
         * Now, assume that the child process is compromised, and that
         * malicious attackers have injected the following statements into this process
         */
        write (fd, "Malicious Data\n", 15);
        close (fd);
    }
```



We eventually fork and create a new process

**Capability leaking** occurs when some process gains escalated privileges, but does not clean up the privileges when downgraded

```c
#include <fcntl.h>

int main()
{
    int fd;

    /*
     * Assume that /etc/zzz is an important system file,
     * and it is owned by root with permission 0644.
     * Before running this program, you should create
     * the file /etc/zzz first.
     */
    fd = open("/etc/zzz", O_RDWR | O_APPEND);
    if (fd == -1) {
        printf("Cannot open /etc/zzz\n");
        exit(0);
    }

    // Simulate the tasks conducted by the program
    sleep(1);

    /*
     * After the task, elevated privileges are no longer needed;
     * it is time to relinquish these privileges!
     * NOTE: getuid() returns the real UID (RUID)
     */
    setuid(getuid());

    if (fork()) {  /* parent process */
        close (fd);
        exit(0);
    } else {  /* child process */

        /*
         * Now, assume that the child process is compromised, and that
         * malicious attackers have injected the following statements into this process
         */
        write (fd, "Malicious Data\n", 15);
        close (fd);
    }
```

fd is defined before the fork()

/etc/zzz is only writeable by root

We eventually fork and create a new process

**Capability leaking** occurs when some process gains escalated privileges, but does not clean up the privileges when downgraded

```c
#include <fcntl.h>

int main()
{
    int fd;

    /*
     * Assume that /etc/zzz is an important system file,
     * and it is owned by root with permission 0644.
     * Before running this program, you should create
     * the file /etc/zzz first.
     */
    fd = open("/etc/zzz", O_RDWR | O_APPEND);
    if (fd == -1) {
        printf("Cannot open /etc/zzz\n");
        exit(0);
    }

    // Simulate the tasks conducted by the program
    sleep(1);

    /* After the task, elevated privileges are no longer needed;
     * it is time to relinquish these privileges!
     * NOTE: getuid() returns the real UID (RUID)
     */
    setuid(getuid());

    if (fork()) {  /* parent process */
        close (fd);
        exit(0);
    } else {  /* child process */

        /*
         * Now, assume that the child process is compromised, and that
         * malicious attackers have injected the following statements into this process
         */
        write (fd, "Malicious Data\n", 15);
        close (fd);
    }
```

fd is defined before the fork()

/etc/zzz is only writeable by root

Drop privileges

We eventually fork and create a new process

**Capability leaking** occurs when some process gains escalated privileges, but does not clean up the privileges when downgraded

```c
#include <fcntl.h>

int main()
{
    int fd;

    /*
     * Assume that /etc/zzz is an important system file,
     * and it is owned by root with permission 0644.
     * Before running this program, you should create
     * the file /etc/zzz first.
     */
    fd = open("/etc/zzz", O_RDWR | O_APPEND);
    if (fd == -1) {
        printf("Cannot open /etc/zzz\n");
        exit(0);
    }

    // Simulate the tasks conducted by the program
    sleep(1);

    /*
     * After the task, elevated privileges are no longer needed;
     * it is time to relinquish these privileges!
     * NOTE: getuid() returns the real UID (RUID)
     */
    setuid(getuid());

    if (fork()) {  /* parent process */
        close (fd);
        exit(0);
    } else {  /* child process */

        /*
         * Now, assume that the child process is compromised, and that
         * malicious attackers have injected the following statements into this process
         */
        write (fd, "Malicious Data\n", 15);
        close (fd);
    }
```

fd is defined before the fork()

/etc/zzz is only writeable by root

Drop privileges

We eventually fork and create a new process

Close file in *parent process*

**Capability leaking** occurs when some process gains escalated privileges, but does not clean up the privileges when downgraded

```c
#include <fcntl.h>

int main()
{
    int fd;

    /*
     * Assume that /etc/zzz is an important system file,
     * and it is owned by root with permission 0644.
     * Before running this program, you should create
     * the file /etc/zzz first.
     */
    fd = open("/etc/zzz", O_RDWR | O_APPEND);
    if (fd == -1) {
        printf("Cannot open /etc/zzz\n");
        exit(0);
    }

    // Simulate the tasks conducted by the program
    sleep(1);

    /* After the task, elevated privileges are no longer needed;
     * it is time to relinquish these privileges!
     * NOTE: getuid() returns the real UID (RUID)
     */
    setuid(getuid());

    if (fork()) {  /* parent process */
        close (fd);
        exit(0);
    } else {  /* child process */

        /*
         * Now, assume that the child process is compromised, and that
         * malicious attackers have injected the following statements into this process
         */
        write (fd, "Malicious Data\n", 15);
        close (fd);
    }
```

fd is defined before the fork()

/etc/zzz is only writeable by root

Drop privileges

We eventually fork and create a new process

Close file in *parent process*

The file descriptor is still open in the child process!

**Capability leaking** occurs when some process gains escalated privileges, but does not clean up the privileges when downgraded

```
[09/15/22]seed@VM:~/lab2$ sudo touch /etc/zzz
[09/15/22]seed@VM:~/lab2$ ./cap_leak
[09/15/22]seed@VM:~/lab2$ sudo cat /etc/zzz
Malicious Data
```

# Principle of Least Privilege

**Subjects and Programs should be given only the privileges needed to complete their task**

Disable privileges when they aren't needed

# Lab 2