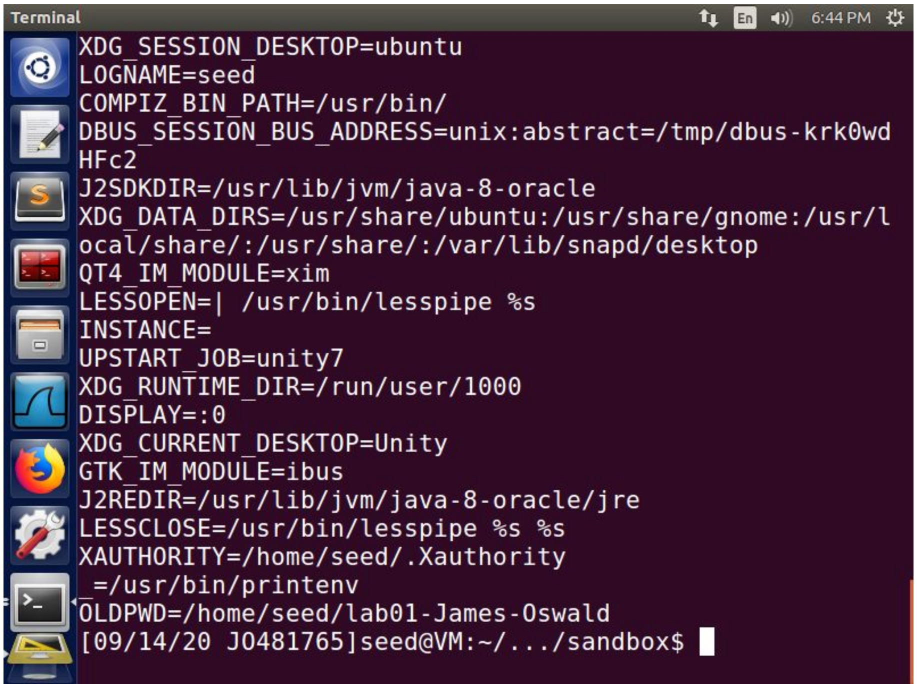
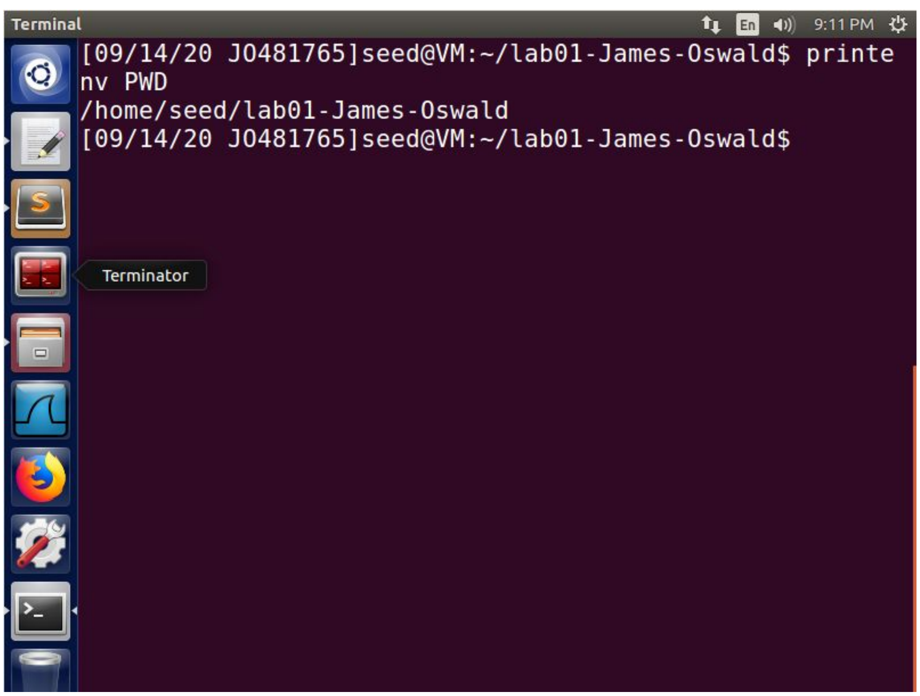
James Oswald
Lab 01

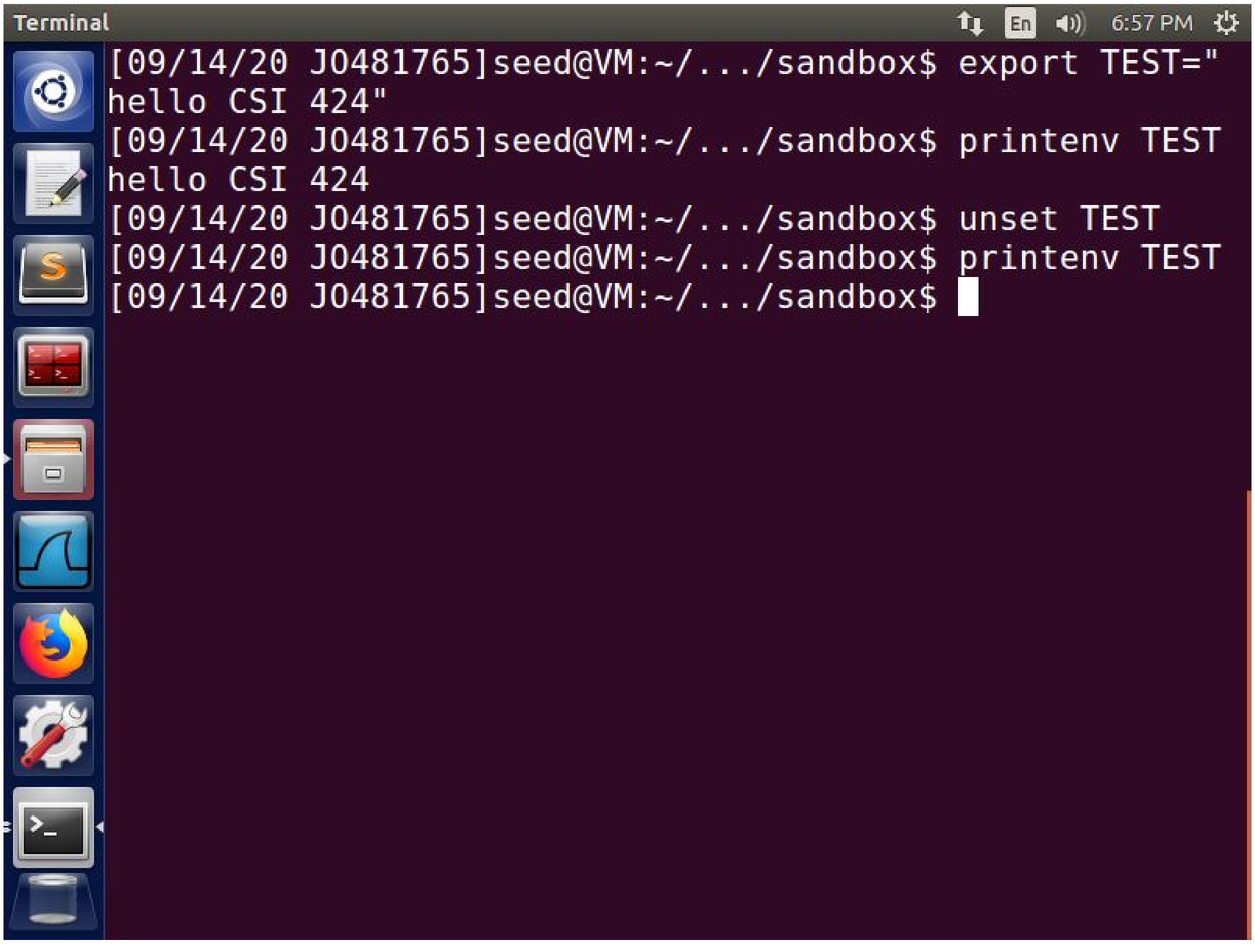
Task #1: Manipulating Environment Variables
Bullet #1:
Running printenv to get a list of all the environmental variables.



Running printenv PWD to see the value of a single environmental variable



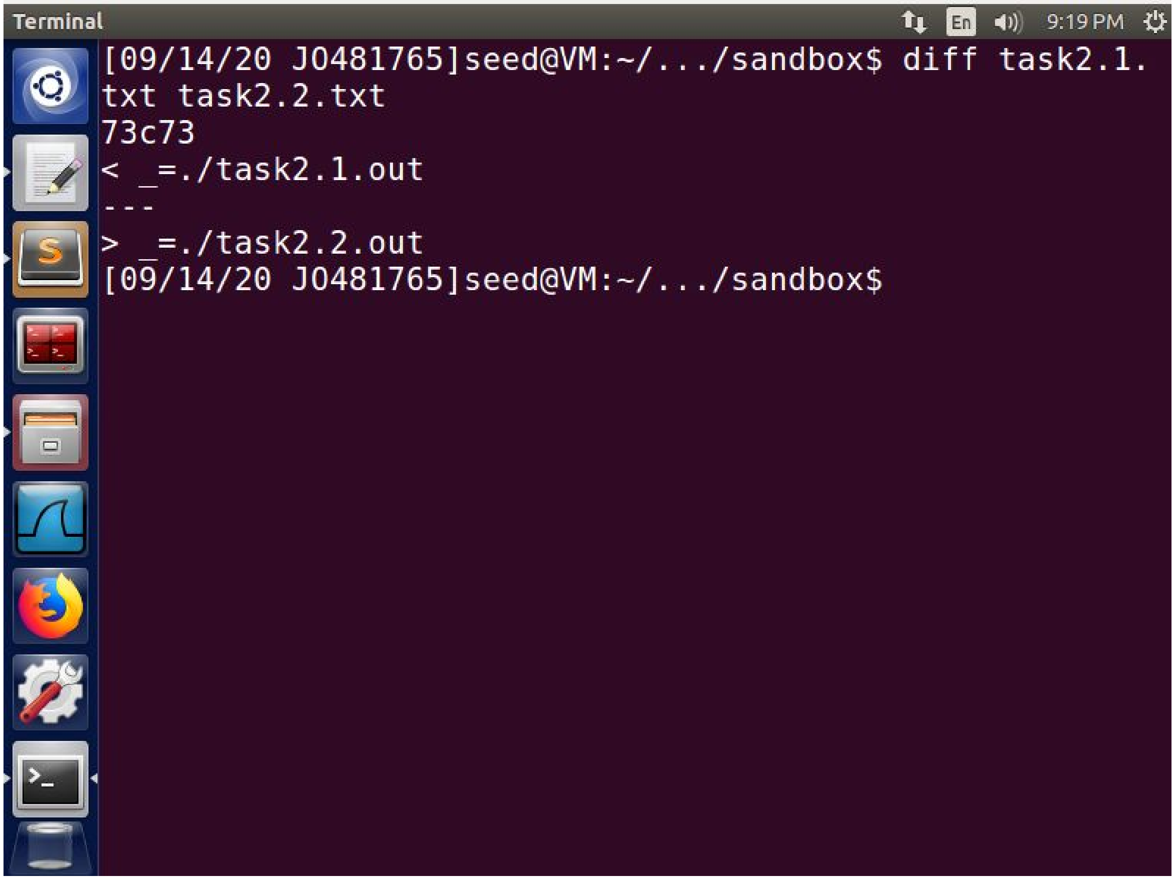
Bullet #2:
Testing out using export and unset to set and unset a single custom environmental variable



Task #2: Passing Environment Variables from Parent Process to Child Process
Step #1
Running the program and looking at the strings produced by the child process in task2.1.txt, I see that the file contains the same things printed by printenv with the exception of $_ being the name of the program.

Step #2
After changing the program and seeing the and looking at the strings produced by the child process in task2.2.txt, I see the same environmental variables with the exception of $_.

Step#3
Running diff on the two output files I see that the only difference is the _ environmental variable.



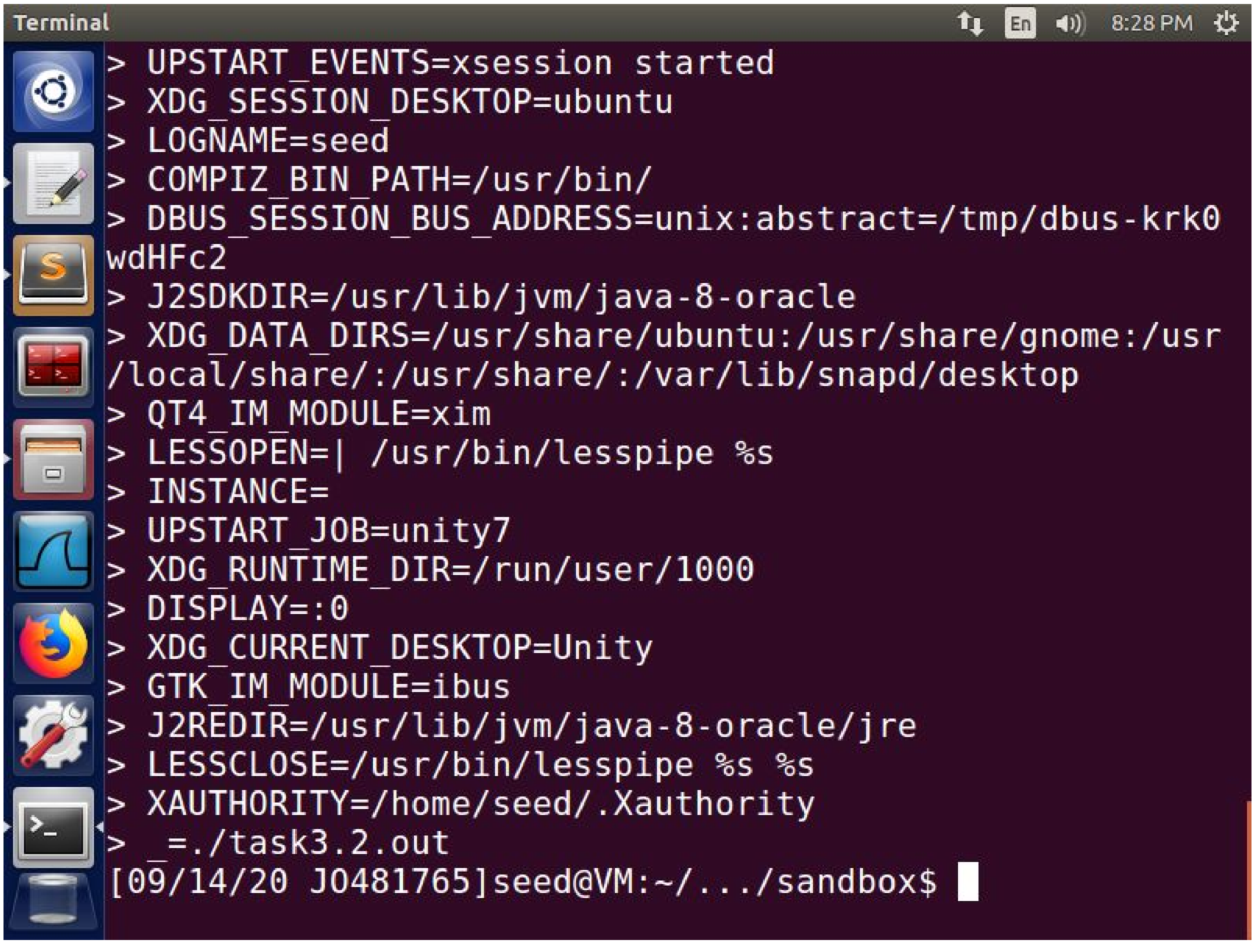
This allows me to draw the conclusion that a child process created with fork() will share the same environment variables as its parent.

Task #3: Environment Variables and execve()

Step #1:
I compiled and ran task3.1.c piping the output to task3.1.txt. The file was empty, no environmental variables were printed.

Step #2:
I made the change to task3.1.c passing in environ as a parameter to execve. I piped the result into task3.2.txt. The file was full and had all the environmental variables present.

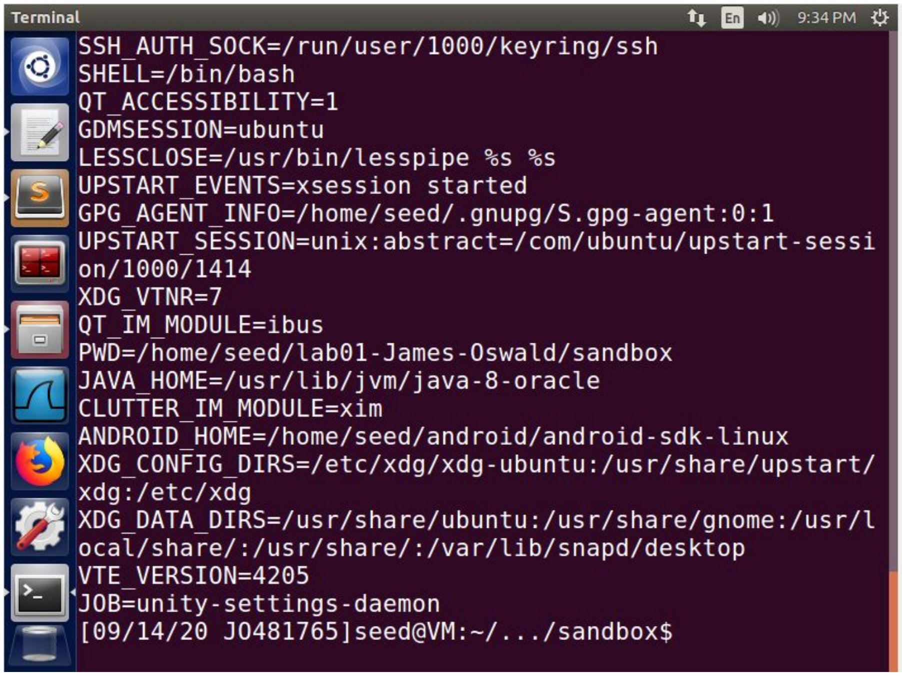
Step #3:
I ran diff on the output files to confirm and see that indeed only the program from task 2 produced the environmental variables as output.



From this result the conclusion can be drawn that the new program gets its environmental variables via the environ parameter being passed to it as the third parameter in execve().

Task #4: Environment Variables and system()

We easily verify that the system passes in environmental variables by compiling the given program and running it. I took the liberty of piping the result into task4.txt, which contained all the proper environmental variables. But have also ran the program to show this to be the case with a screenshot:



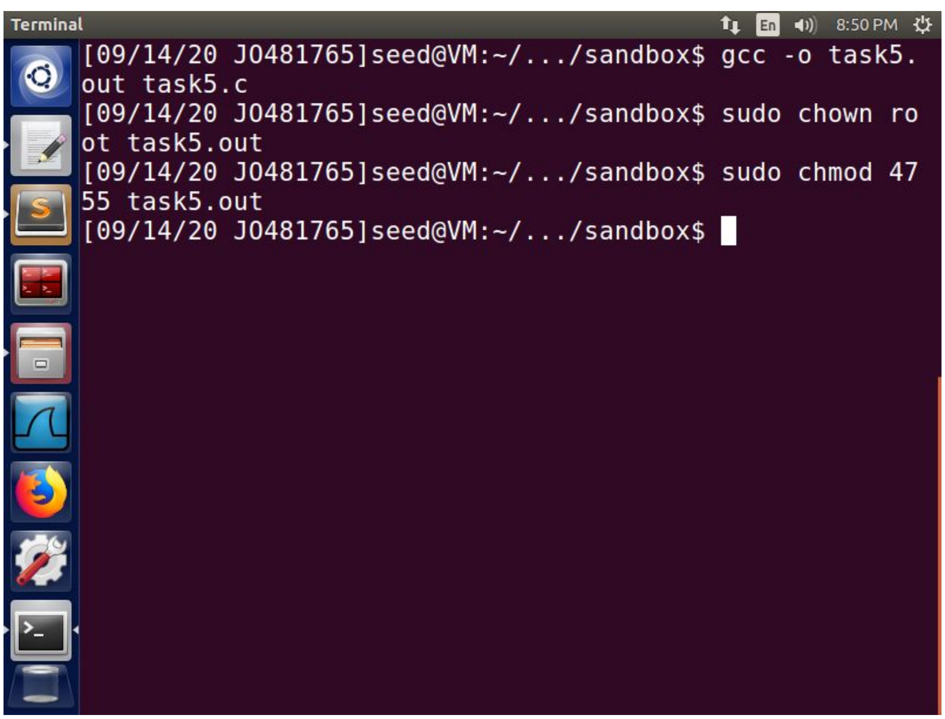
Task #5: Environment Variable and Set-UID Programs

Step #1:

I wrote the program and saved it as task5.c compiling it to task5.out

Step #2:

I performed the necessary modifications with chown and chmod to make it a Set-UID program



Step#3:
I set the new environmental variables using export and then ran task5.out

```
[09/14/20 J0481765]seed@VM:~/.../sandbox$ gcc -o task5.
out task5.c
[09/14/20 J0481765]seed@VM:~/.../sandbox$ sudo chown ro
ot .
[09/14/20 J0481765]seed@VM:~/.../sandbox$ sudo chmod 47
55 .
[09/14/20 J0481765]seed@VM:~/.../sandbox$ export PATH=$
PATH:/fake/path/addition
[09/14/20 J0481765]seed@VM:~/.../sandbox$ export LD_LIB
RARY_PATH=/fake/path
[09/14/20 J0481765]seed@VM:~/.../sandbox$ export MEMEVA
R="This is a joke var"
```

The result was rather surprising, despite being a Set-UID root program, its environmental variables were the ones I had just set using export.

```
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed
CLUTTER_IM_MODULE=xim
SESSION=ubuntu
ANDROID_HOME=/home/seed/android/android-sdk-linux
GPG_AGENT_INFO=/home/seed/.gnupg/S.gpg-agent:0:1
TERM=xterm-256color
VTE_VERSION=4205
XDG_MENU_PREFIX=gnome-
SHELL=/bin/bash
DERBY_HOME=/usr/lib/jvm/java-8-oracle/db
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
WINDOWID=60817497
OLDPWD=/home/seed/lab01-James-Oswald
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-sessi
on/1000/1414
GNOME_KEYRING_CONTROL=
GTK_MODULES=gail:atk-bridge:unity-gtk-module
MEMEVAR=This is a joke var
USER=seed
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;3
5:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su
=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=
```

2.6 Task 6: The PATH Environment Variable and Set-UID Programs

First setting up the code and making its executable a Set-UID program

```
[09/14/20 J0481765]seed@VM:~/.../sandbox$ gcc -o task6.
out task6.c
[09/14/20 J0481765]seed@VM:~/.../sandbox$ sudo chown ro
ot task6.out
[09/14/20 J0481765]seed@VM:~/.../sandbox$ sudo chmod 47
55 task6.out
[09/14/20 J0481765]seed@VM:~/.../sandbox$
```

Now adding my working directory to the first spot in the path using

```
[09/14/20 J0481765]seed@VM:~/.../sandbox$ export PATH=$
PWD:$PATH
[09/14/20 J0481765]seed@VM:~/.../sandbox$
```

To test if I can run my own program and weather it has root privileges, I write my own program to replace ls

task3.1.c × task3.2.c × task4.c × task5.c × task6.c × ls.c ×

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

int main()
{
    system("whoami");
    return 0;
}

```

And i compile it as ls

Now running task.6.out

```
[09/14/20 J0481765]seed@VM:~/.../sandbox$ task6.out
seed
[09/14/20 J0481765]seed@VM:~/.../sandbox$
```

We see that I have successfully run my own code and replaced ls, but It’s not running as root, rather its running as seed and is without root privileges.

Task 7: The LD_PRELOAD Environment Variable and Set-UID Programs

Step #1

Setting LD_PRELOAD for the seed, saving and compiling the program

```
[09/14/20 J0481765]seed@VM:~/.../sandbox$ gcc -fPIC -g
-c mylib.c
[09/14/20 J0481765]seed@VM:~/.../sandbox$ gcc -shared -
o libmylib.so.1.0.1 mylib.o -lc
[09/14/20 J0481765]seed@VM:~/.../sandbox$ export LD_PRE
LOAD=./libmylib.so.1.0.1
[09/14/20 J0481765]seed@VM:~/.../sandbox$
```

Step #2:

Testing with different users and setUID states. As a normal user with LD_PRELOAD set, our code runs, when changed to a setUID program our code doesn't run.

```
[09/14/20 J0481765]seed@VM:~/.../sandbox$ myprog.out
[09/14/20 J0481765]seed@VM:~/.../sandbox$ export LD_PRE
LOAD=./libmylib.so.1.0.1
[09/14/20 J0481765]seed@VM:~/.../sandbox$ myprog.out
I am not sleeping!
[09/14/20 J0481765]seed@VM:~/.../sandbox$ sudo chown ro
ot myprog.out
[09/14/20 J0481765]seed@VM:~/.../sandbox$ sudo chmod 47
55 myprog.out
[09/14/20 J0481765]seed@VM:~/.../sandbox$ myprog.out
[09/14/20 J0481765]seed@VM:~/.../sandbox$
```

Switching to root and setting LD_PRELOAD will make our code run.

```
[09/14/20 J0481765]seed@VM:~/.../sandbox$ sudo -i
root@VM:~# cd /home/seed/lab01-James-Oswald/sandbox/
root@VM:/home/seed/lab01-James-Oswald/sandbox# export e
xport LD_PRELOAD=./libmylib.so.1.0.1
root@VM:/home/seed/lab01-James-Oswald/sandbox# ./myprog
.out
I am not sleeping!
root@VM:/home/seed/lab01-James-Oswald/sandbox#
```

Switching the program to be owned by user1 and setting LD_PRELOAD will not make our code run.

```
[09/14/20 JO481765]seed@VM:~/.../sandbox$ sudo -i
root@VM:~# cd /home/seed/lab01-James-Oswald/sandbox/
root@VM:/home/seed/lab01-James-Oswald/sandbox# su user1
 -c "export LD_PRELOAD=./libmylib.so.1.0.1"
root@VM:/home/seed/lab01-James-Oswald/sandbox# su user1
 -c "myprog.out"
root@VM:/home/seed/lab01-James-Oswald/sandbox#
```

Step #3:

To figure out the cause, i can set up an experiment by printing out weather LD PRELOAD has been set. The conclusion of this experiment is that when trying to run code with elevated privileges not as a user whos LD_PRELOAD has been set, the code won’t run.

Task #8: Invoking External Programs Using system() versus execve():

Step #1

```
[09/14/20 JO481765]seed@VM:~/.../sandbox$ gcc -o task8.
out task8.c
[09/14/20 JO481765]seed@VM:~/.../sandbox$ sudo chmod 47
55 task8.out
[09/14/20 JO481765]seed@VM:~/.../sandbox$ sudo chown ro
ot task8.out
[09/14/20 JO481765]seed@VM:~/.../sandbox$
```

If i was bob I would not be able to compromise the system via PATH since the path to /bin/cat is absolute (I tried doing this, the code is in /sandbox/bin/cat.c), however I may be able to take advantage of LD_PRELOAD to redefine sprintf to sore my own custom input in command.

No matter what you do, the child process with inherit the environmental variables via being invoked with system

Step 2:

```
[09/14/20 J0481765]seed@VM:~/.../sandbox$ gcc -o task8.
2.out task8.2.c
[09/14/20 J0481765]seed@VM:~/.../sandbox$ sudo chown ro
ot task8.2.out
[09/14/20 J0481765]seed@VM:~/.../sandbox$ sudo chmod 47
55 task8.2.out
[09/14/20 J0481765]seed@VM:~/.../sandbox$ 
```

Changing the line to be execve() with NULL passed in as its environmental vars ensures that we won't be able to use them maliciously in the subprocess even if we can change the behavior of the task 8 using them.

Task 9:

Creating the file /etc/zzz

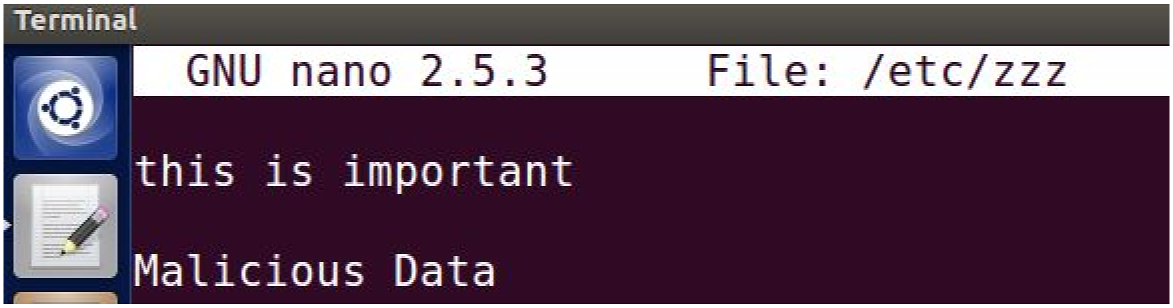
```
  GNU nano 2.5.3       File: /etc/zzz          Mod:

this is important
```

Making it owned by root and changing it to a Set UID program

```
[09/14/20 J0481765]seed@VM:~/.../sandbox$ sudo chown ro
ot task9.out
[09/14/20 J0481765]seed@VM:~/.../sandbox$ sudo chmod 47
55 task9.out
[09/14/20 J0481765]seed@VM:~/.../sandbox$ task9.out
[09/14/20 J0481765]seed@VM:~/.../sandbox$ 
```

Sure enough the program was able to insert malicious data into the file.



This result is very interesting and rather unexpected. Despite relinquishing root privileges, the process still retains the ability to write to a file it should no longer be able to write to. I suspect this is due to the file descriptor being created while the program still had root privileges.

All code can be found in the sandbox folder