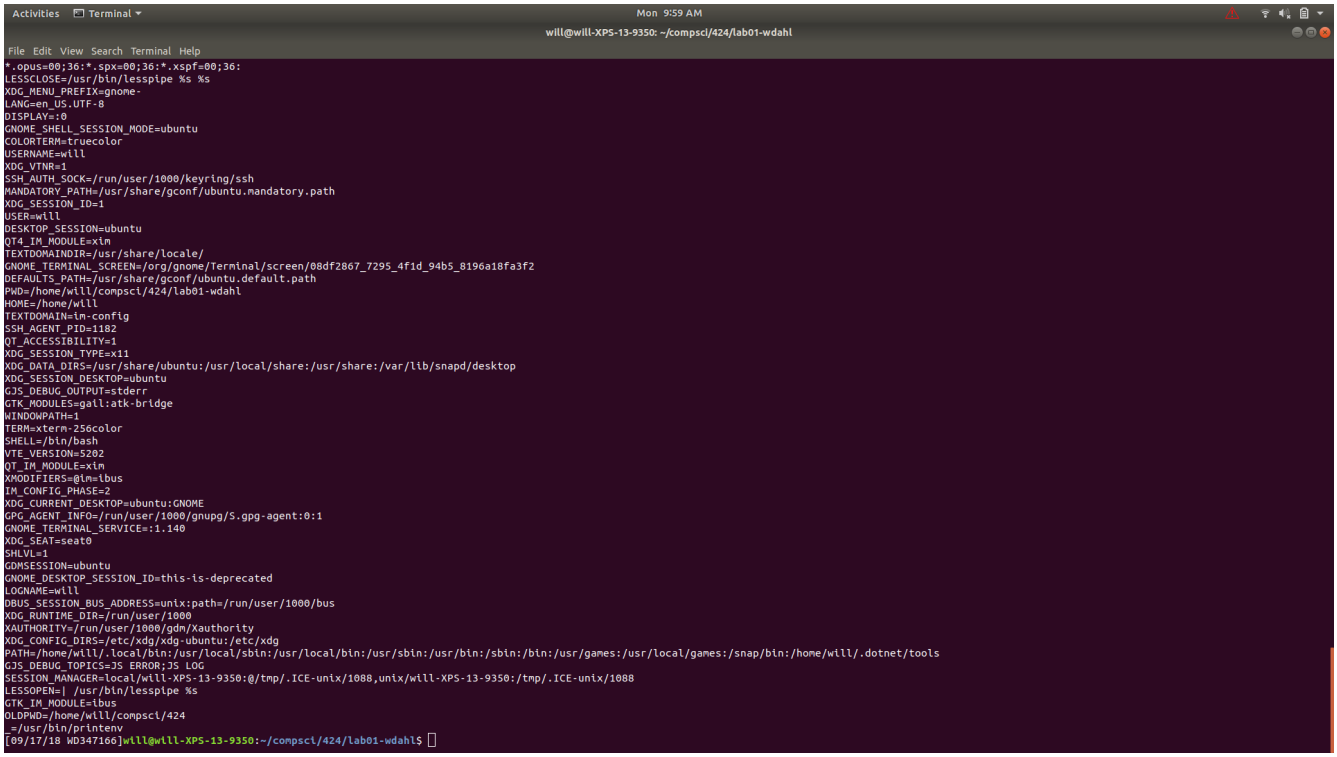
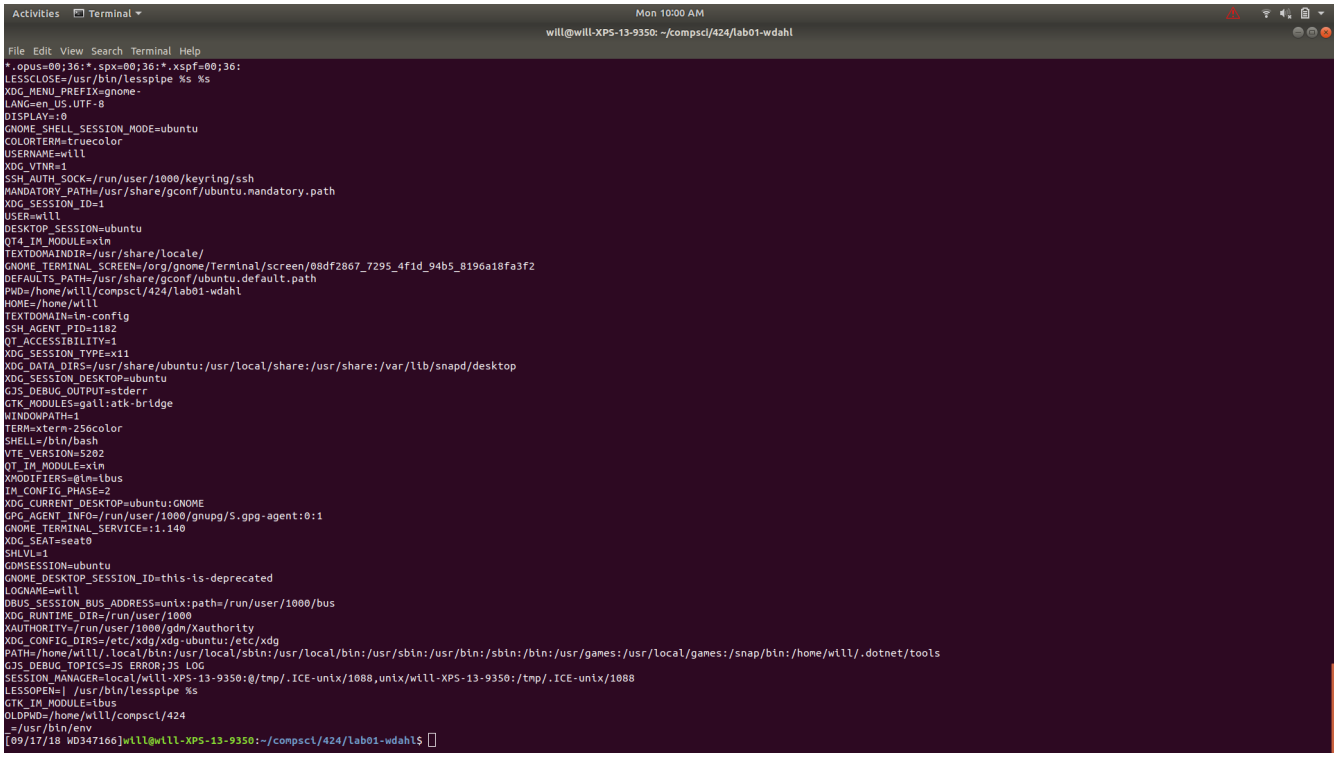
William Dahl
Professor Amir Masoumzadehtork
ICSI 424 Information Security – Lab 01
9/17/2018

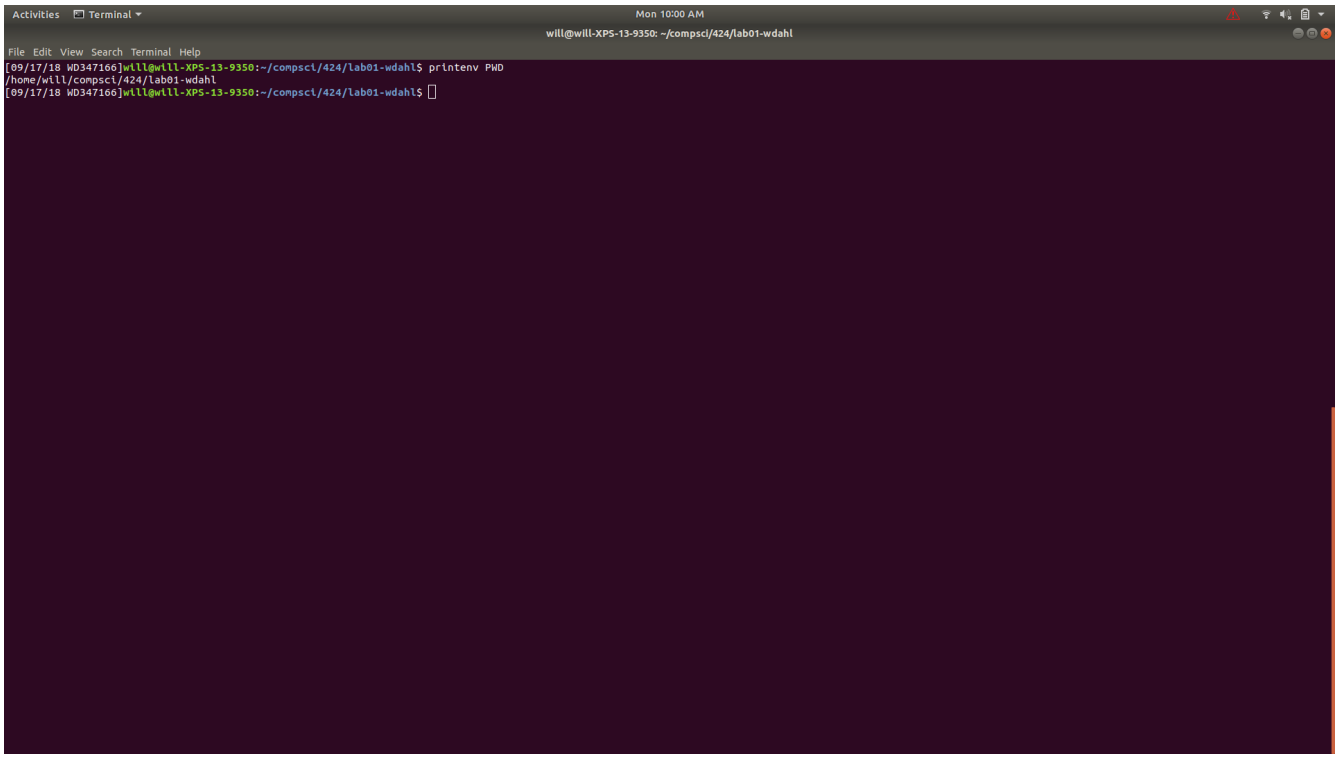
Task 1: In this screenshot I ran printenv in my bash prompt which printed out all of my environmental variables



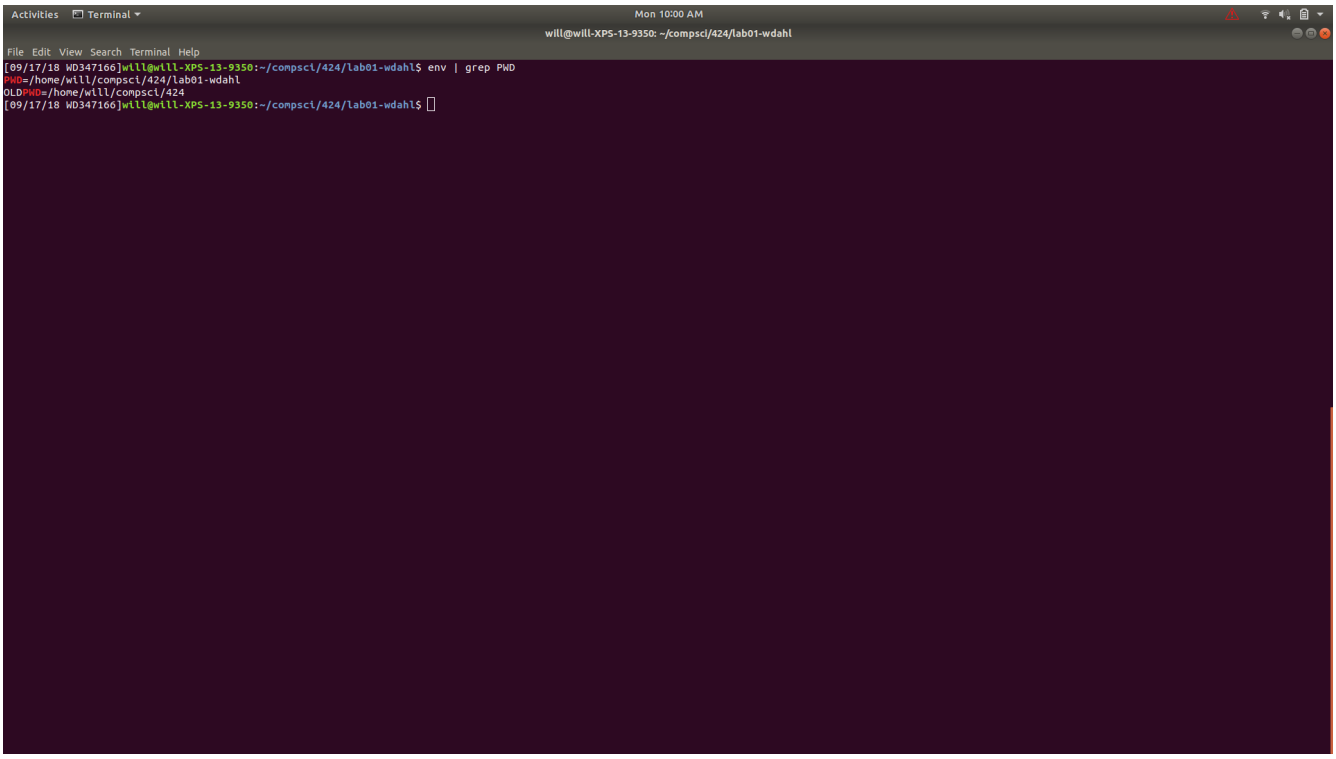
In this screenshot I ran env in my bash prompt which printed put all my environmental variables



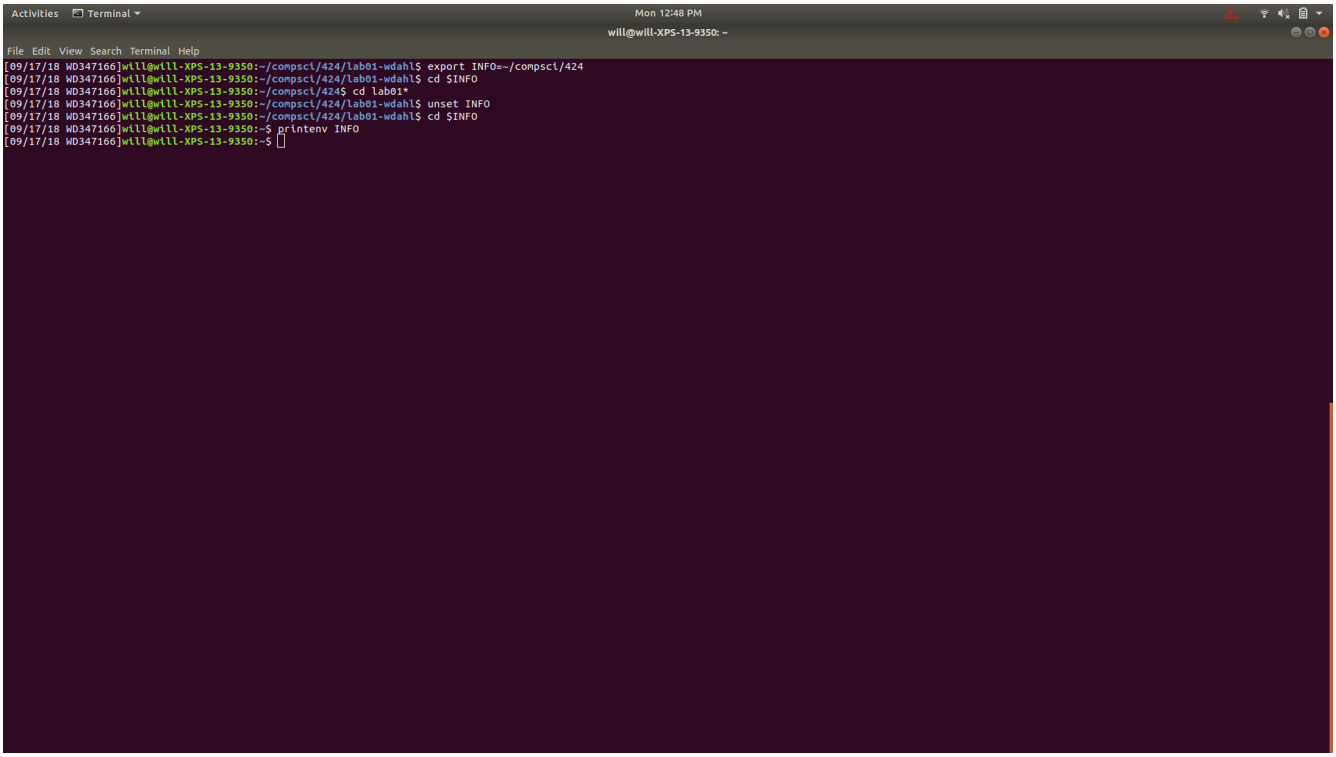
In this screenshot I ran printenv PWD in my prompt which returned my current working directory



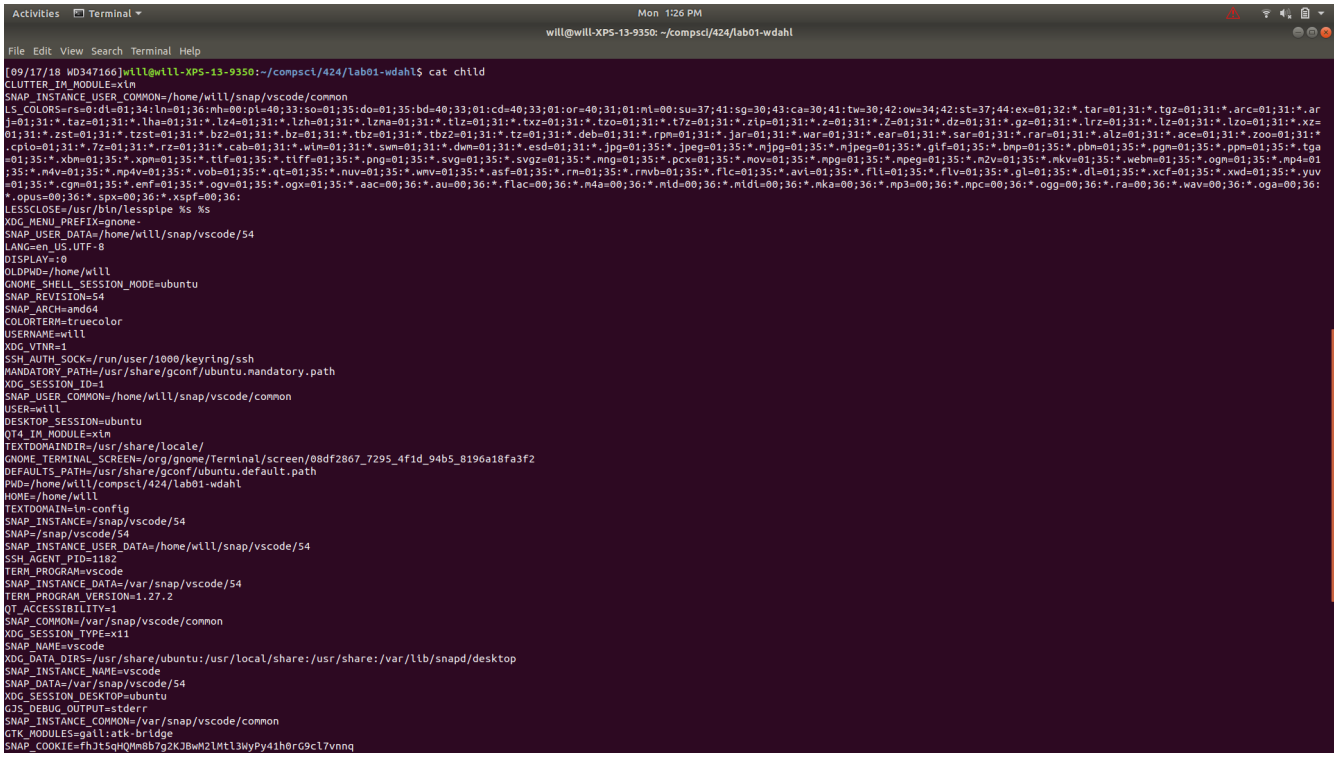
In this screenshot I ran env | grep PWD which returned anything in my environment variables containing the string PWD, so it returned my current working directory as well as my old working directory which is the parent of my current working directory



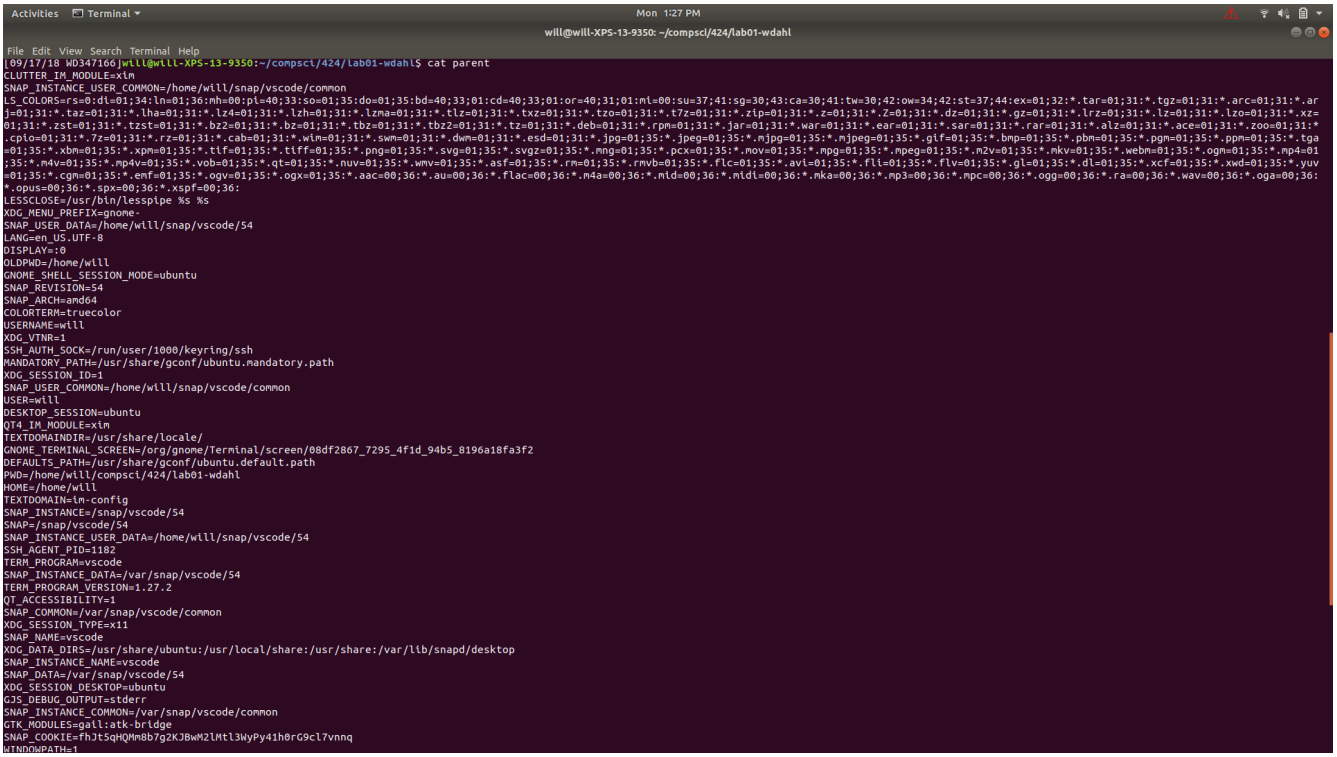
In this screenshot I set an Environmental Variable INFO to be a path, when I run cd $INFO it takes me to that path. When I unset INFO and I ran cd INFO again it took me back to my home directory because INFO is empty.



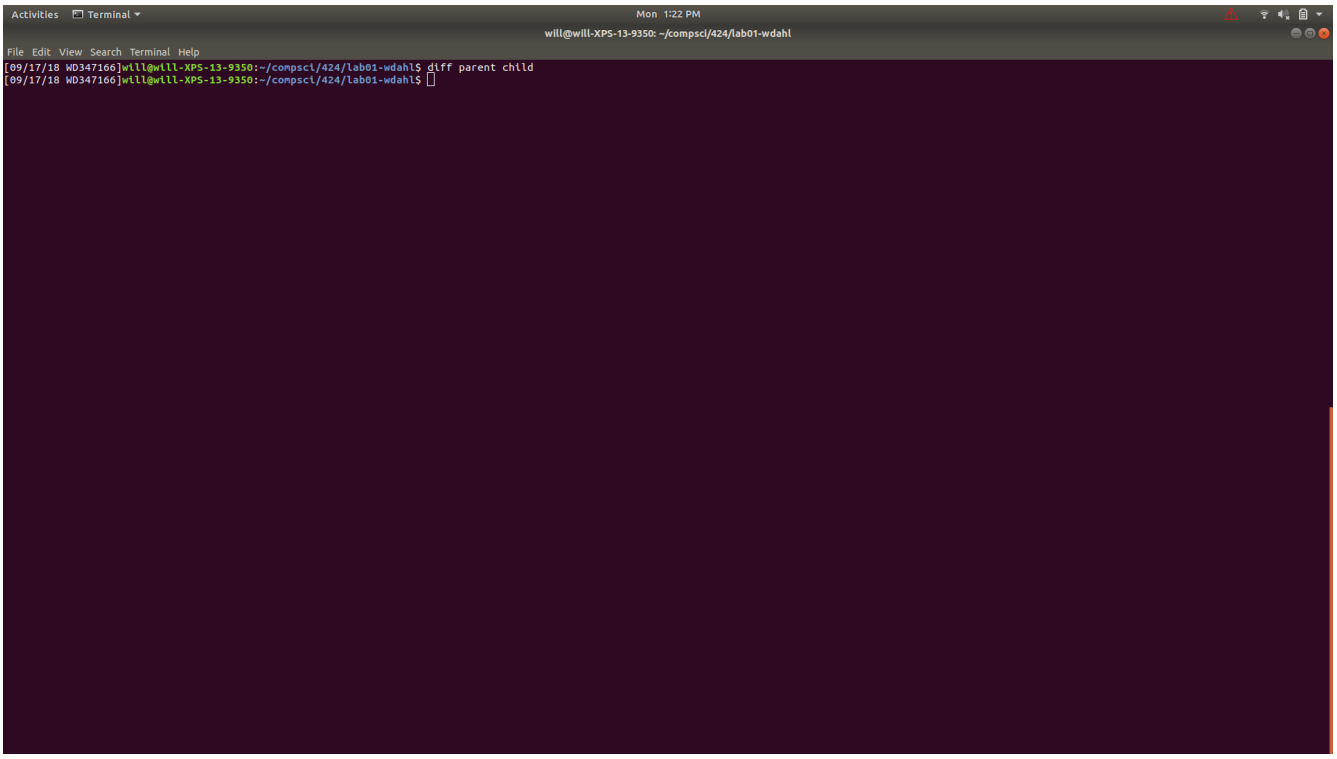
Task 2: This screen shot shows the output when the environmental variables of the child process are printed.



This screenshot shows the output when the environmental variables of the parent process are printed

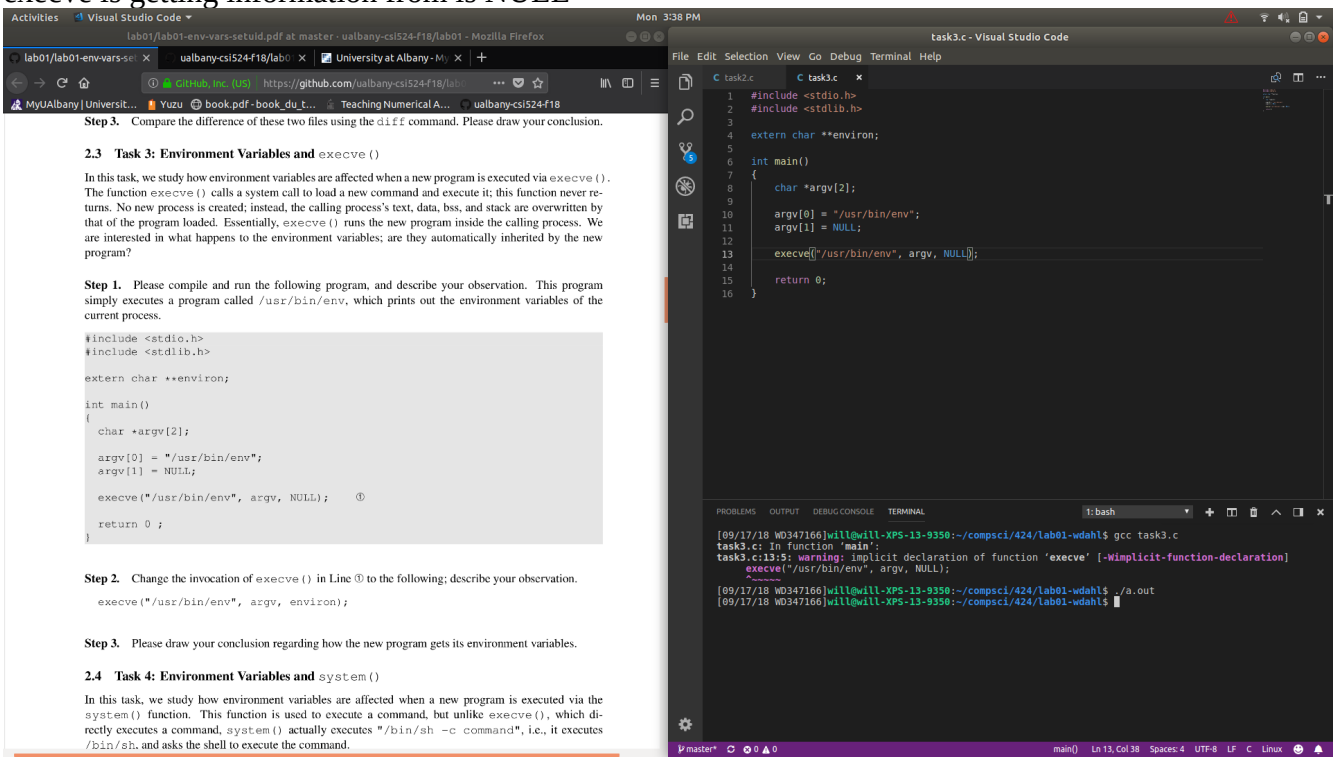


This screenshot shows the output when diff is run against the outputs from the child process and the parent process.

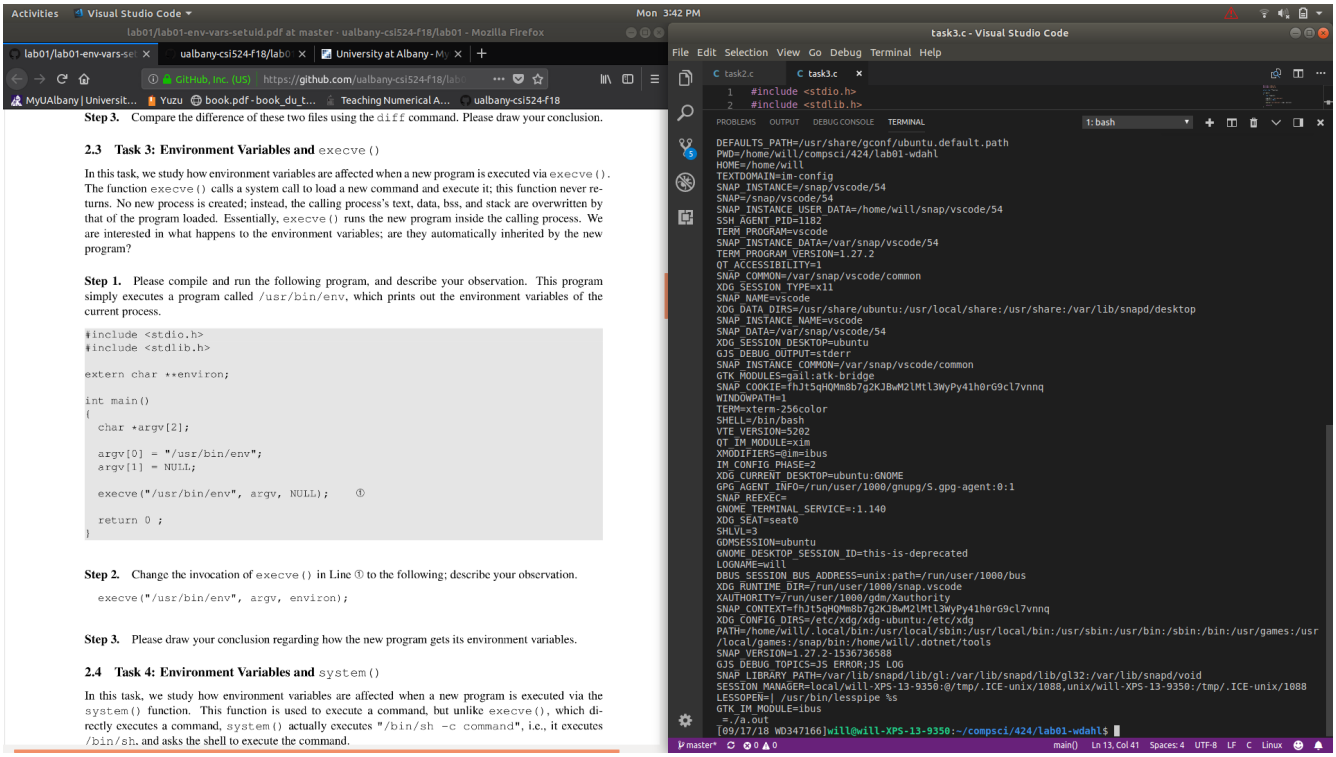


After running the diff command against the output form the child process there is no difference between the environmental variables in the parent and child processes. This is because no new Environmental variable were created within the program.

Task 3: In this screen shot we see that nothing is printed out because the source that the command execve is getting information from is NULL

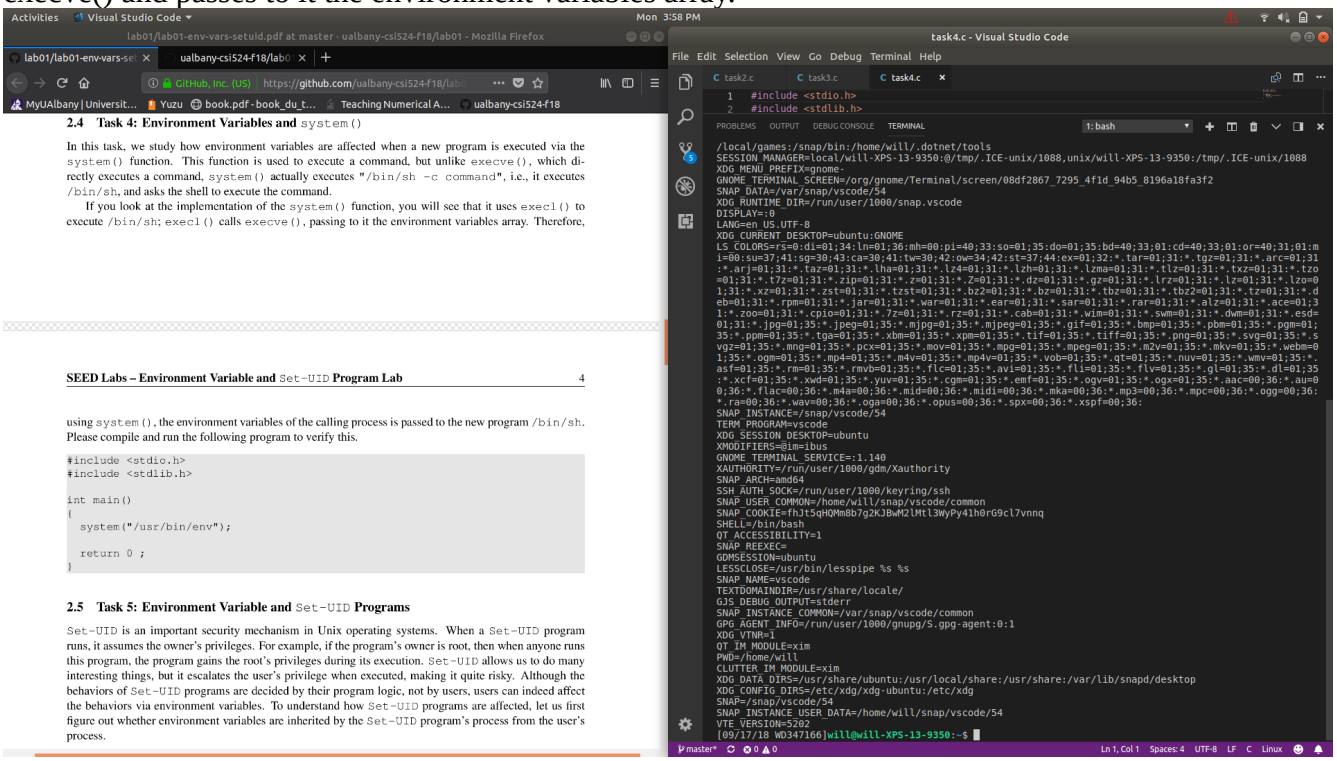


In this screenshot we see that the processes environmental variables are printed out. This is because the source that execve was given was changed from NULL to environ which contained all the information regarding the environmental variables in the process.



Without environ being passed to evecve the process doesn't know of any environmental variables. That is why in order to see the environmental variables printed out the external environment needs to be given to the command so that It can print out the environmental variables.

Task 4: In this screenshot we see that the environment variables were printed out by running task4.c and no environ was needed. This is because the System call to “/bin/sh/env” uses execl() which calls execve() and passes to it the environment variables array.



2.4 Task 4: Environment Variables and `system()`

In this task, we study how environment variables are affected when a new program is executed via the `system()` function. This function is used to execute a command, but unlike `execve()`, which directly executes a command, `system()` actually executes `"/bin/sh -c command"`, i.e., it executes `/bin/sh`, and asks the shell to execute the command.

If you look at the implementation of the `system()` function, you will see that it uses `execl()` to execute `/bin/sh`; `execl()` calls `execve()`, passing to it the environment variables array. Therefore,

SEED Labs – Environment Variable and `Set-UID` Program Lab 4

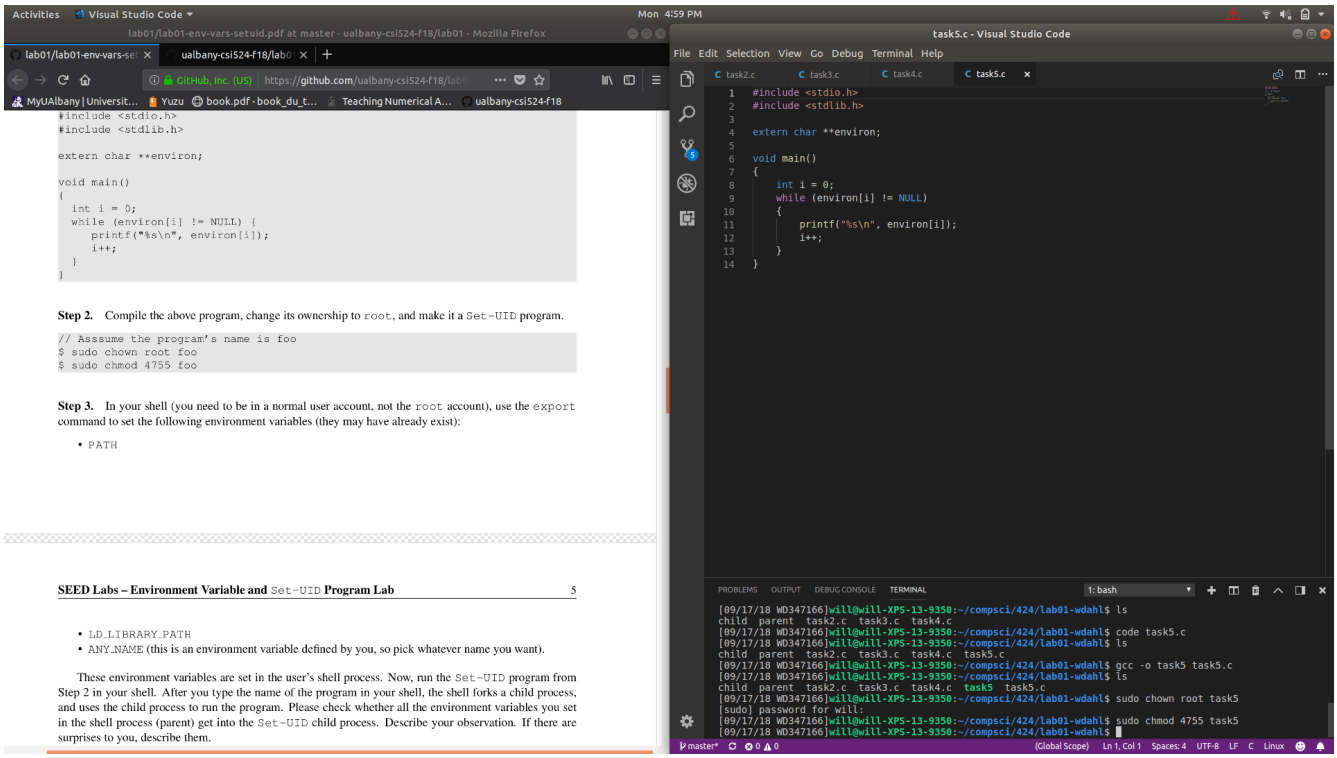
using `system()`, the environment variables of the calling process is passed to the new program `/bin/sh`. Please compile and run the following program to verify this.

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

int main()
{
  system("/usr/bin/env");

  return 0 ;
}
```

2.5 Task 5: Environment Variable and `Set-UID` Programs

`Set-UID` is an important security mechanism in Unix operating systems. When a `Set-UID` program runs, it assumes the owner's privileges. For example, if the program's owner is root, then when anyone runs this program, the program gains the root's privileges during its execution. `Set-UID` allows us to do many interesting things, but it escalates the user's privilege when executed, making it quite risky. Although the behaviors of `Set-UID` programs are decided by their program logic, not by users, users can indeed affect the behaviors via environment variables. To understand how `Set-UID` programs are affected, let us first figure out whether environment variables are inherited by the `Set-UID` program's process from the user's process.

Task 5: In this screen shot I changed the executable file for the code in task 5 to be owned by root and have a mode of 4755 to make it a Set-UID program.



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

extern char **environ;

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

Step 2. Compile the above program, change its ownership to `root`, and make it a `Set-UID` program.

```
// Assume the program's name is foo
$ sudo chown root foo
$ sudo chmod 4755 foo
```

Step 3. In your shell (you need to be in a normal user account, not the `root` account), use the `export` command to set the following environment variables (they may have already exist):

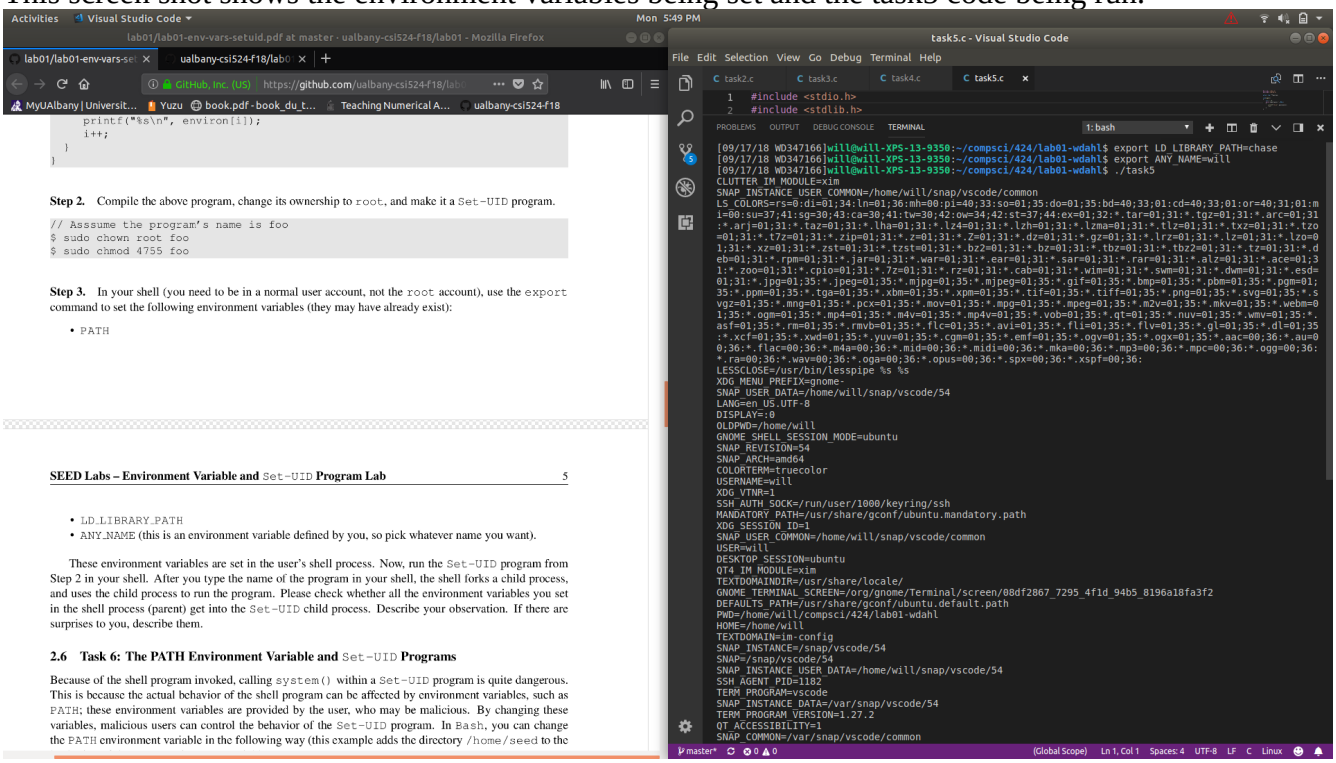
- `PATH`

SEED Labs – Environment Variable and `Set-UID` Program Lab 5

- `LD_LIBRARY_PATH`
- `ANY_NAME` (this is an environment variable defined by you, so pick whatever name you want).

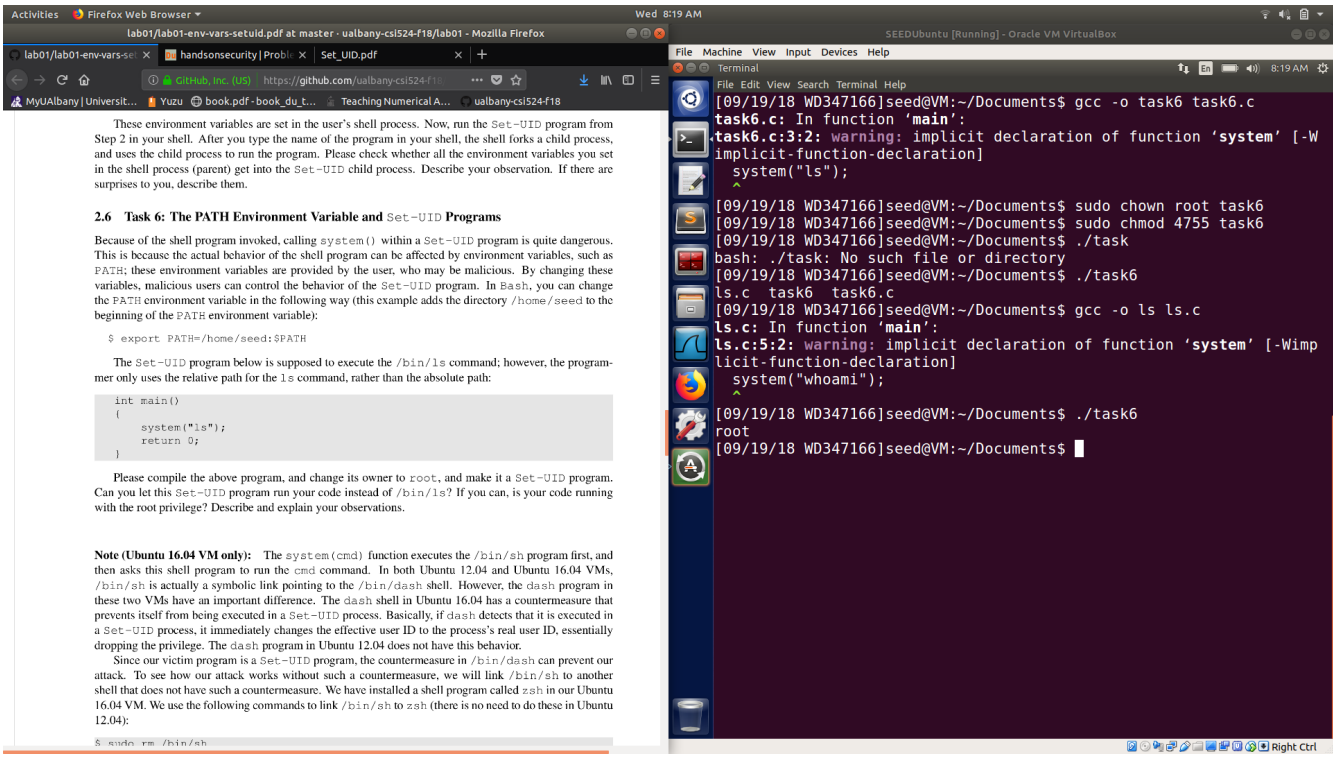
These environment variables are set in the user's shell process. Now, run the `Set-UID` program from Step 2 in your shell. After you type the name of the program in your shell, the shell forks a child process, and uses the child process to run the program. Please check whether all the environment variables you set in the shell process (parent) get into the `Set-UID` child process. Describe your observation. If there are surprises to you, describe them.

This screen shot shows the environment variables being set and the task5 code being run.



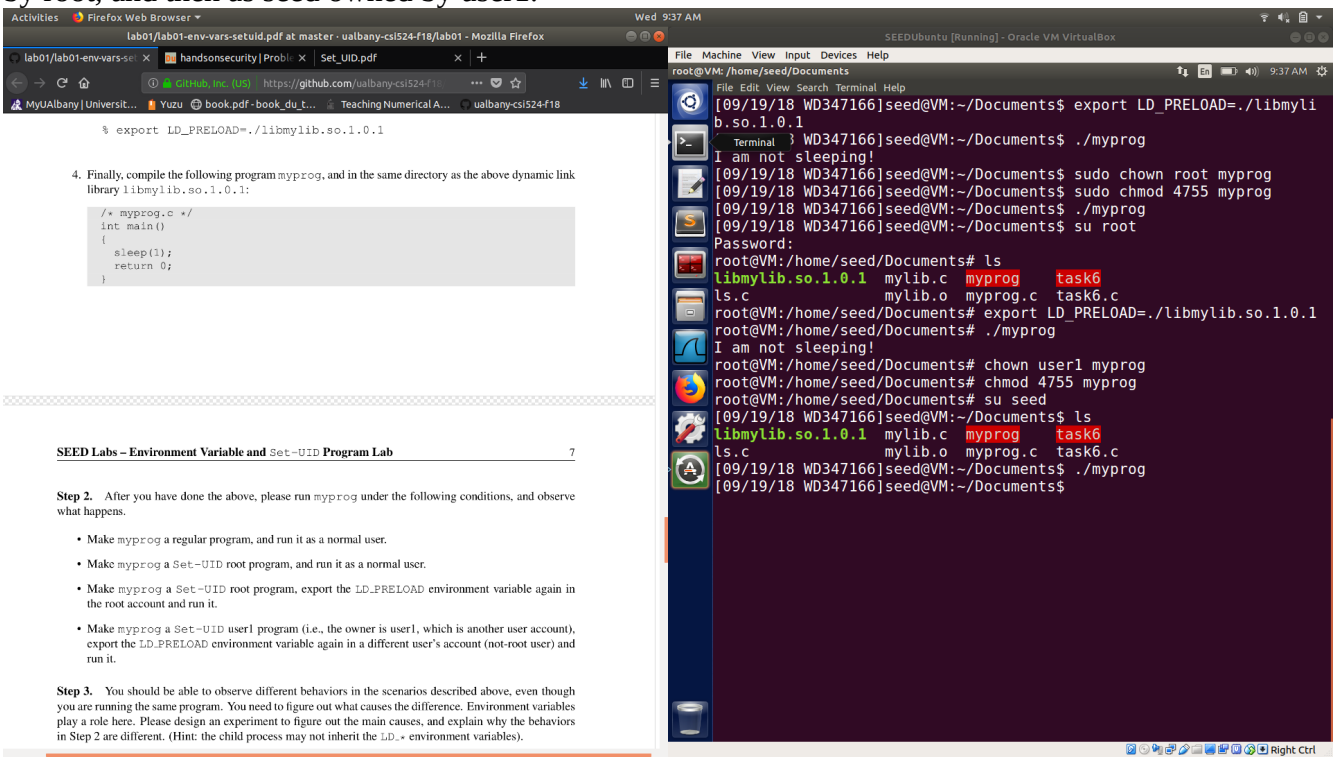
In the environment variables outputted by the code, the only exported variable that did not appear in the child processes environment variables was LD_LIBRARY_PATH. Both Path and my made up variable appeared in the child processes environment variables. I was surprised to see that LD_LIBRARY_PATH did not appear but ANY_NAME did.

Task 6: In this screenshot I compile the code for task6.c and change the executable task6 to be owned by root and to be a SET_UID program. Then I run the code and it preforms as expected. However after I compile the code in ls.c and name the executable ls, task6 will instead execute my code which is a system call using the command “whoami” which this displays that the program thinks I’m root thus showing that my code is running with root privileges.



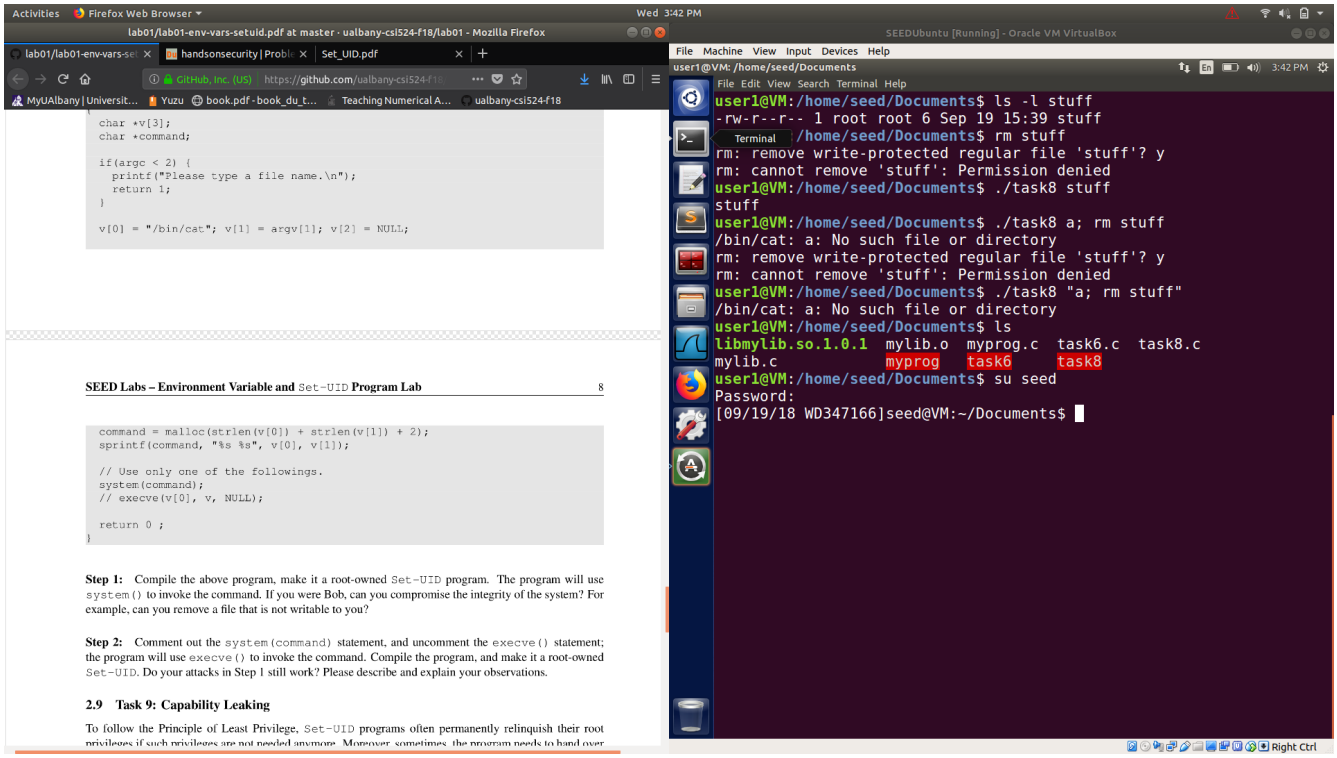
Altering the PATH variable by putting a “.” at the beginning (referencing the current directory) caused the ls file in the current directory to execute instead of the shell command ls. My program also shows that it is being run with root privilege as the shell thinks that It is a root privilege program. This shows that I could write my own malicious code and have it execute with root privileges.

Task 7: In this screenshot I ran myprog as seed owned by seed, as seed owned by root, as root owned by root, and then as seed owned by user1.

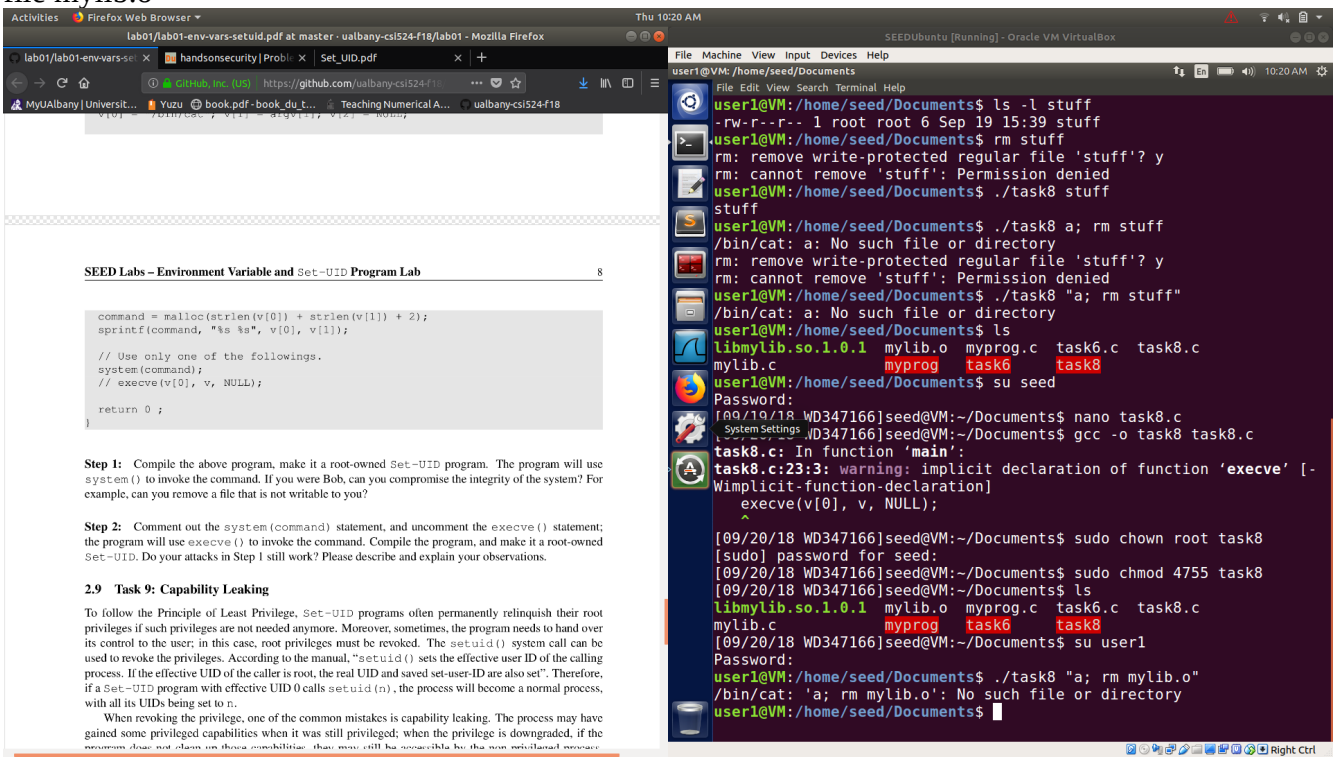


The output from myprog is different because in the cases where “I am not sleeping” printed out, the EUID and the RUID matched so It used the library specified in LD_PRELAOD. However in the other times it didnt work was because the EUID and the RUID did not match so as a countermeasure the system ignores the LD_PRELOAD variable.

Task 8: In this screen shot I use the user1 account I created for task 7 to try and remove a file called stuff which is owned by root and is write protected.



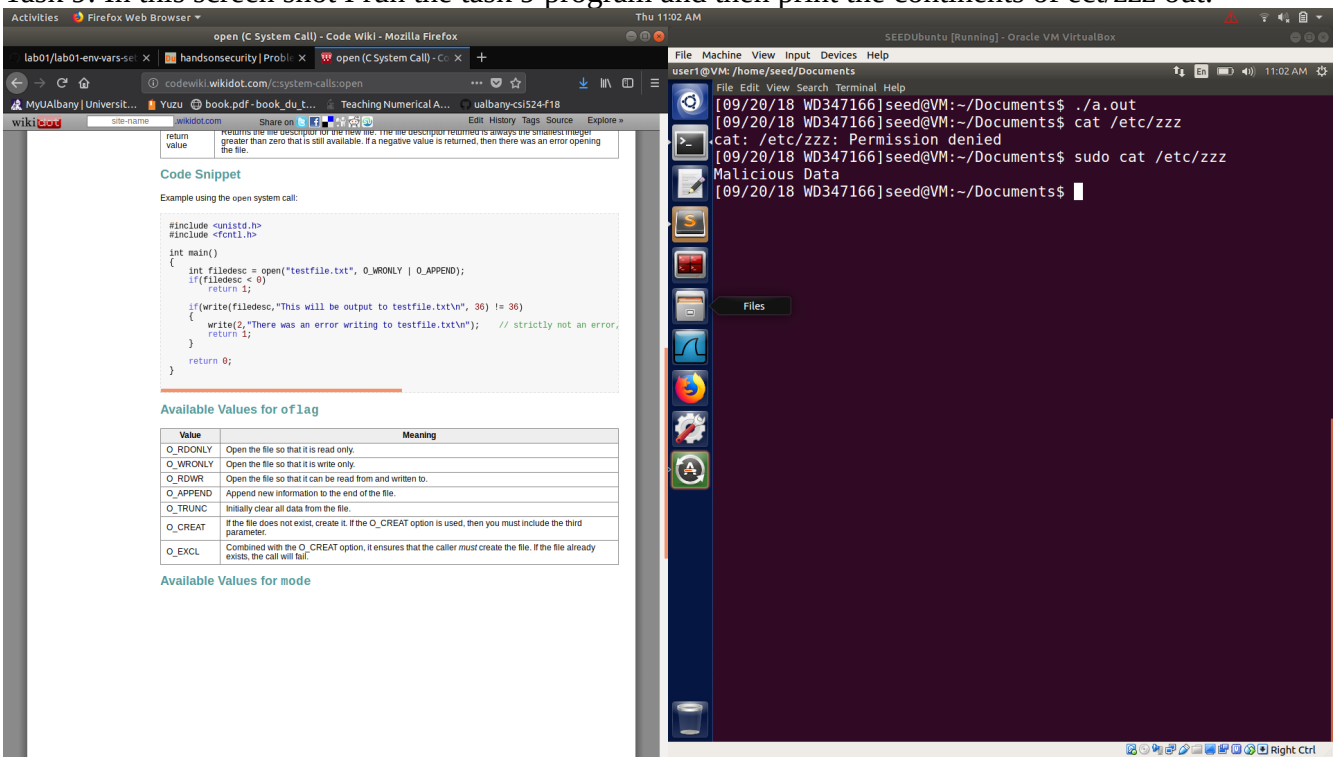
In this screen shot I run task8 code again but this time using the execve function and try to remove the file mylib.o



When using the system call I was able to remove a write protected file because the system() function makes a call to bin/sh which then executes the command passed to it, in this case “a; rm stuff”. Thus the command a; rm stuff was executed with root privileges. A is not a file that command doesnt work

but the ; allows for us to put in more than one command thus rm stuff was executed with root privileges and I was able to remove the write protected file. However, when using execve() the entire string “a; rm mylib.o” was passed to bin/cat and sense there is no directory called that the command did not work.

Task 9: In this screen shot I run the task 9 program and then print the continents of ect/zzz out.



The continents of ect/zzz were changed because we are able to open the file thanks to the root privileges and even though the uid was changed later in the program we were still able to write to the file because it was already opened.