Information Security: Environment Variable and Set-UID Program

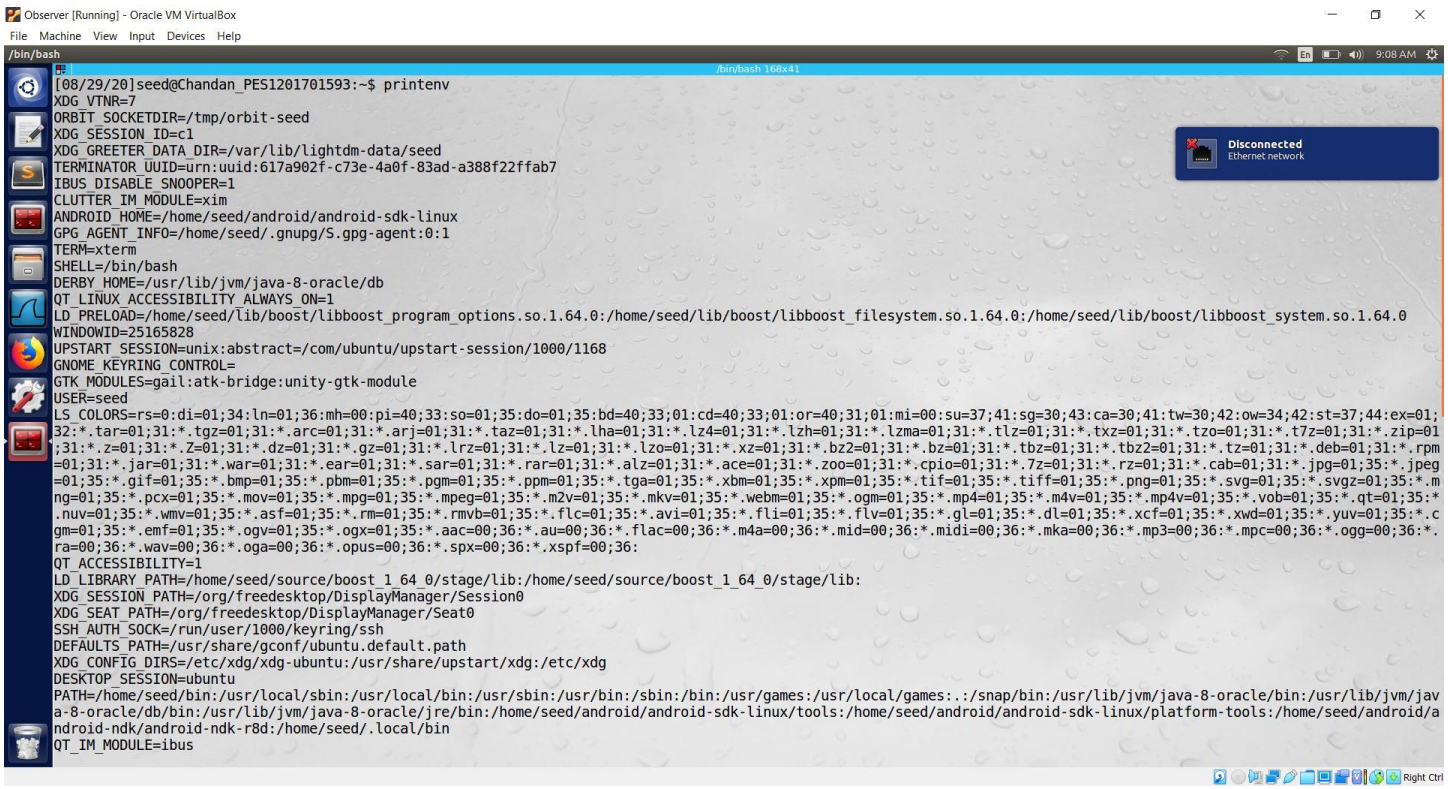
Name : Chandan N Bhat

PES1201701593

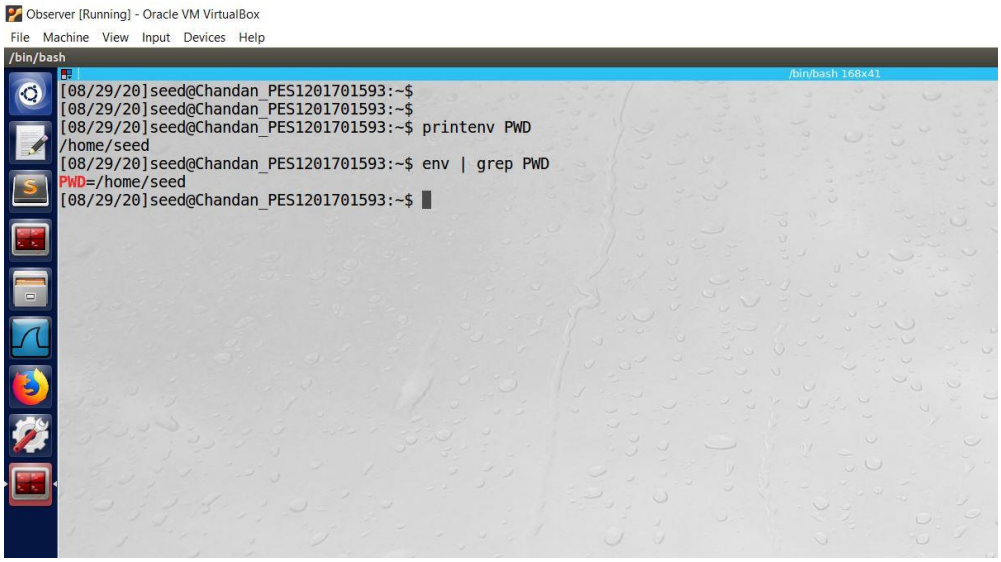
Section H

Task 1: Manipulating environment variables

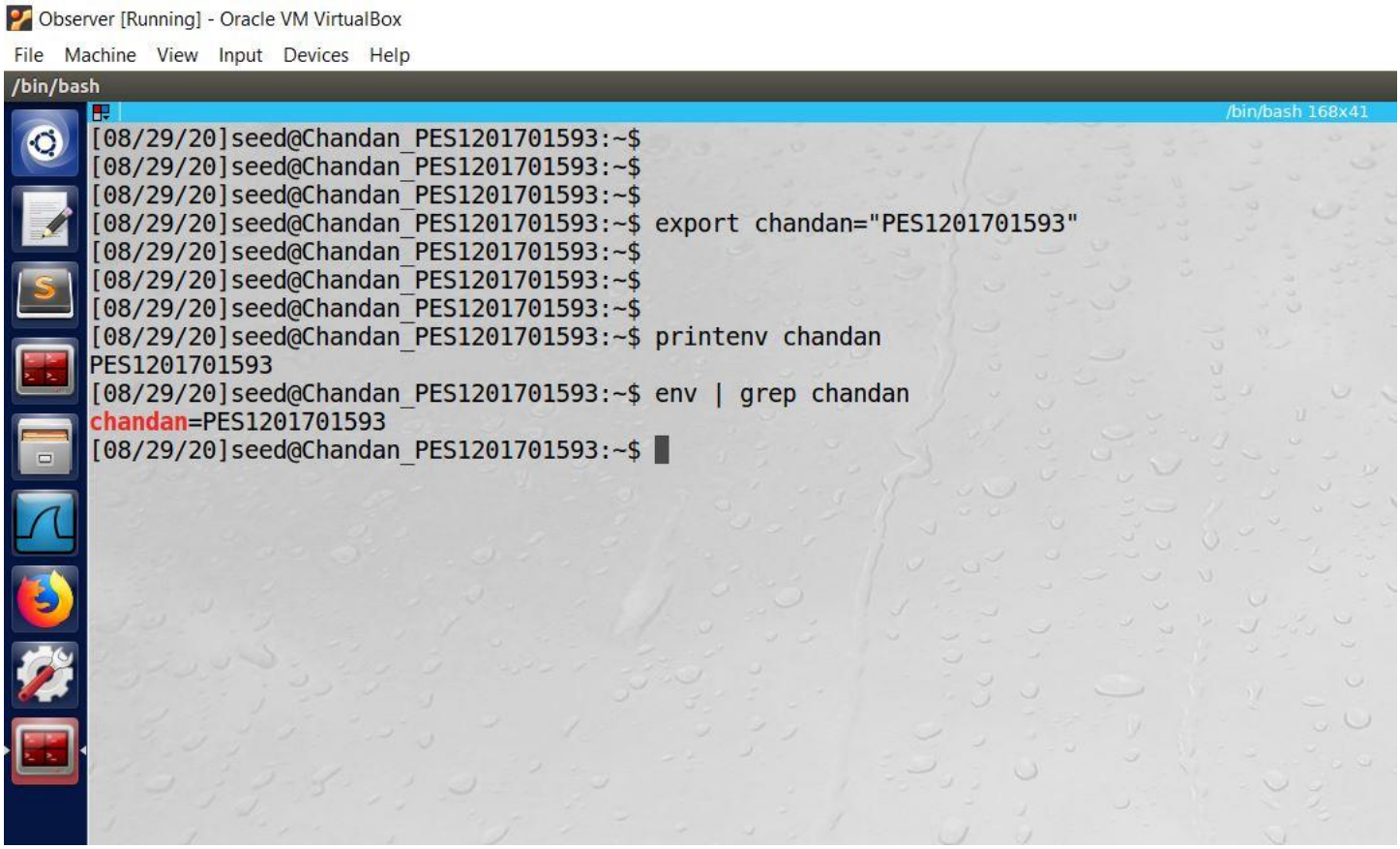
Here we use few Linux commands to set and unset environment variables. We use the default shell, Bash. The “printenv” or “env” command can be used to print out the environment variables



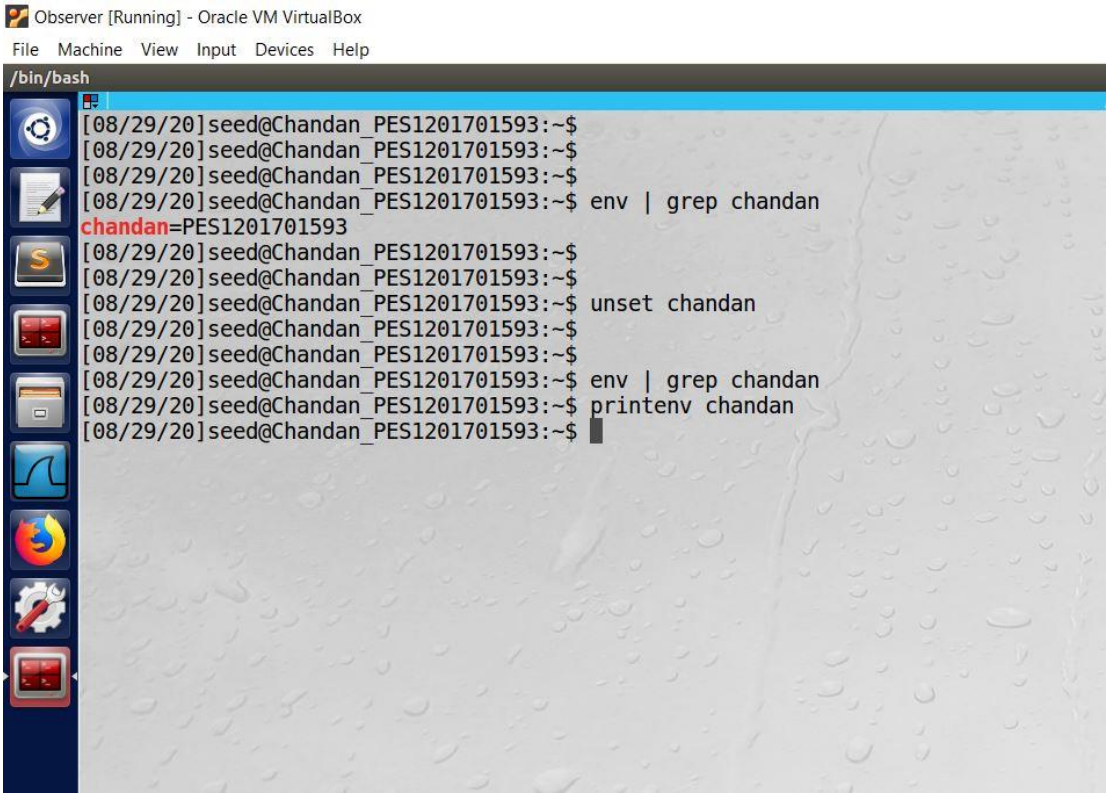
Including the key i.e., “pwd” for instance, will print the particular environment variable.



Next using the “export” command we set an environment variable as shown in the below screenshot, where we set an environment variable “chandan=PES1201701593”.



We can unset an existing environment variable using the unset command as shown below



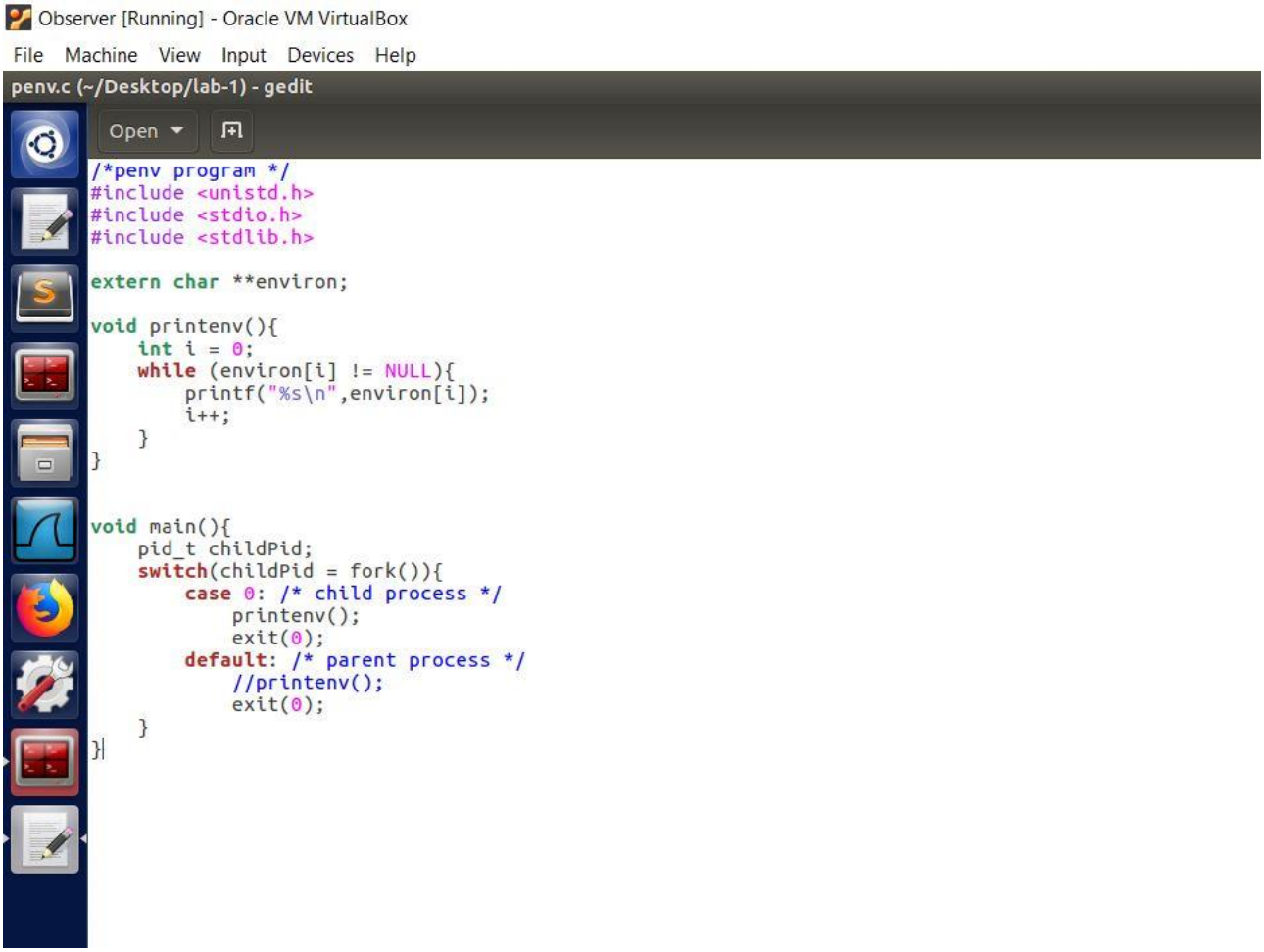
We see that once we unset the environment variable, and try to print it, nothing is displayed in the output as we have unset the environment variable.

Thus, we successfully set and unset environment variables using Bash's internal commands "export" and "unset". We also print the environment variables using "printenv" command.

Task 2: Inheriting environment variables from parents

Now we will understand how environment variables are inherited by child processes from the parent processes. The fork() system call duplicates the calling process which is called the child process. During this process the child process inherits various attributes from its parent such as file descriptors. We will check if environment variables are inherited by the child process in this task.

1. We first run a simple C program that prints the environment variables of the process.



2. We first print all the environment variables of the child process created by fork() in a file ,say child.txt.

Observer [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

/bin/bash

/bin/bash 168x41

```
[08/29/20]seed@Chandan_PES1201701593:~/.../lab-1$
[08/29/20]seed@Chandan_PES1201701593:~/.../lab-1$
[08/29/20]seed@Chandan_PES1201701593:~/.../lab-1$ gcc penv.c
[08/29/20]seed@Chandan_PES1201701593:~/.../lab-1$ ./a.out > child.txt
[08/29/20]seed@Chandan_PES1201701593:~/.../lab-1$
[08/29/20]seed@Chandan_PES1201701593:~/.../lab-1$ ls -l child.txt
-rw-rw-r-- 1 seed seed 4126 Aug 29 09:26 child.txt
[08/29/20]seed@Chandan_PES1201701593:~/.../lab-1$
[08/29/20]seed@Chandan_PES1201701593:~/.../lab-1$
[08/29/20]seed@Chandan_PES1201701593:~/.../lab-1$
[08/29/20]seed@Chandan_PES1201701593:~/.../lab-1$
```

3. The we print all the environment variables of the parent process in another file, say parent.txt. We make a minor change in the same code, as shown below.

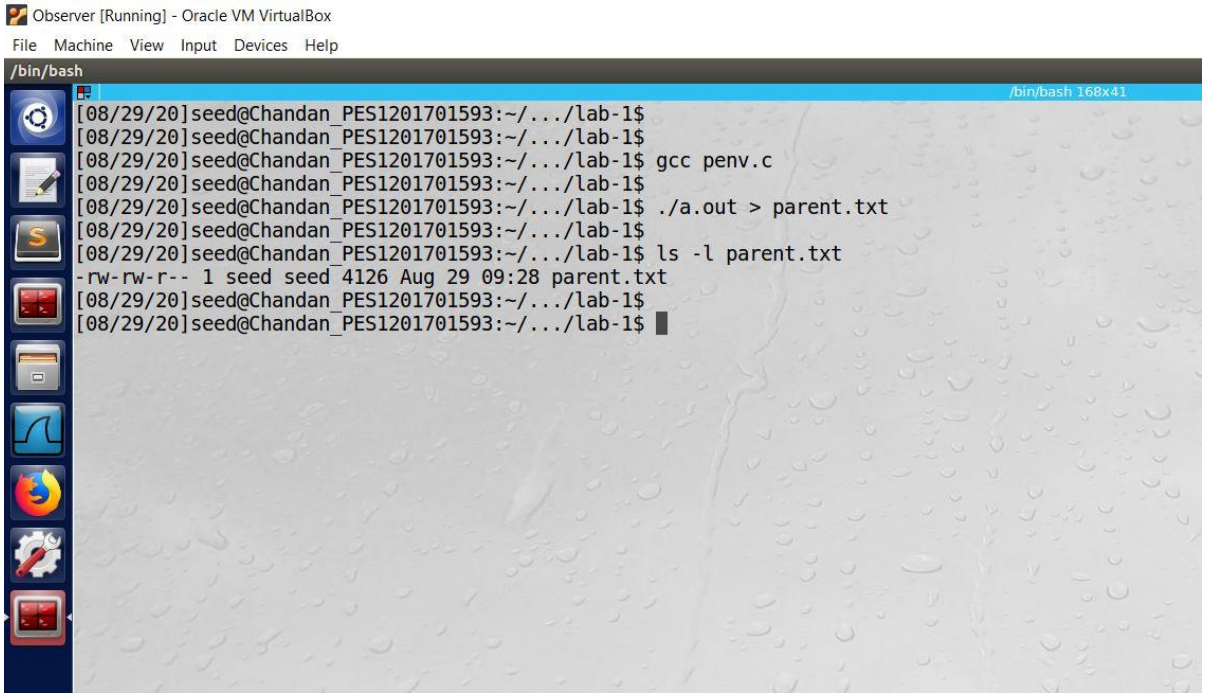
Observer [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

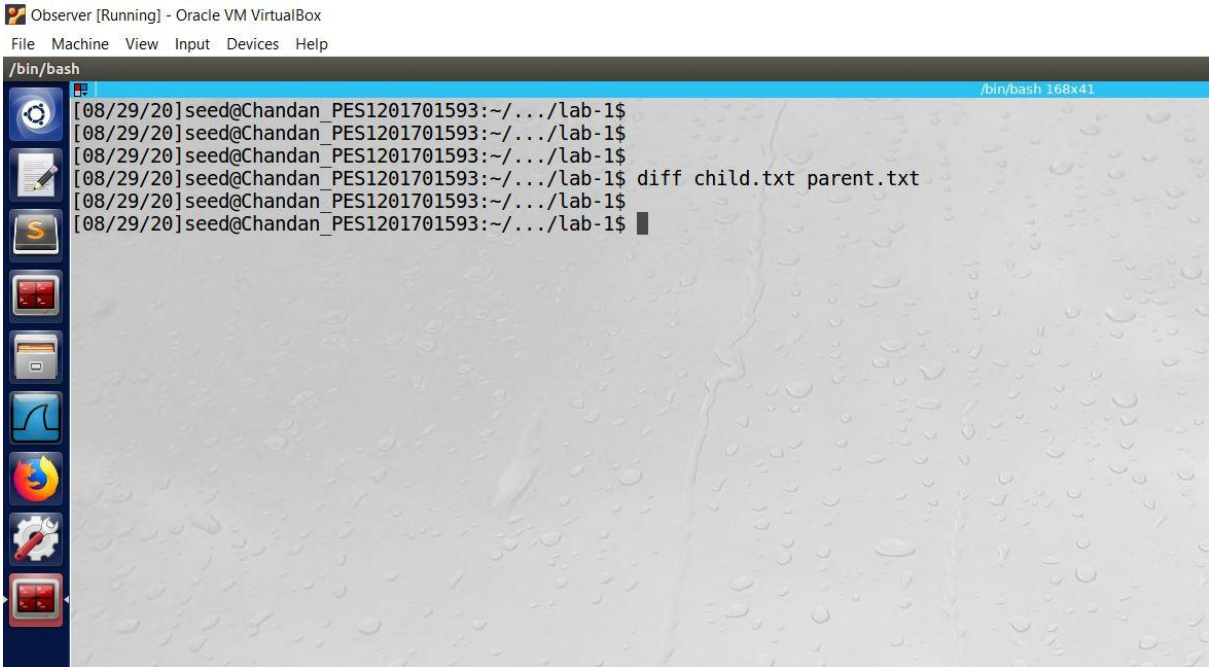
penv.c (~/Desktop/lab-1) - gedit

Open

```c
/*penv program */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

extern char **environ;

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


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



4. Then using the “diff” command we compare the differences between the two files.

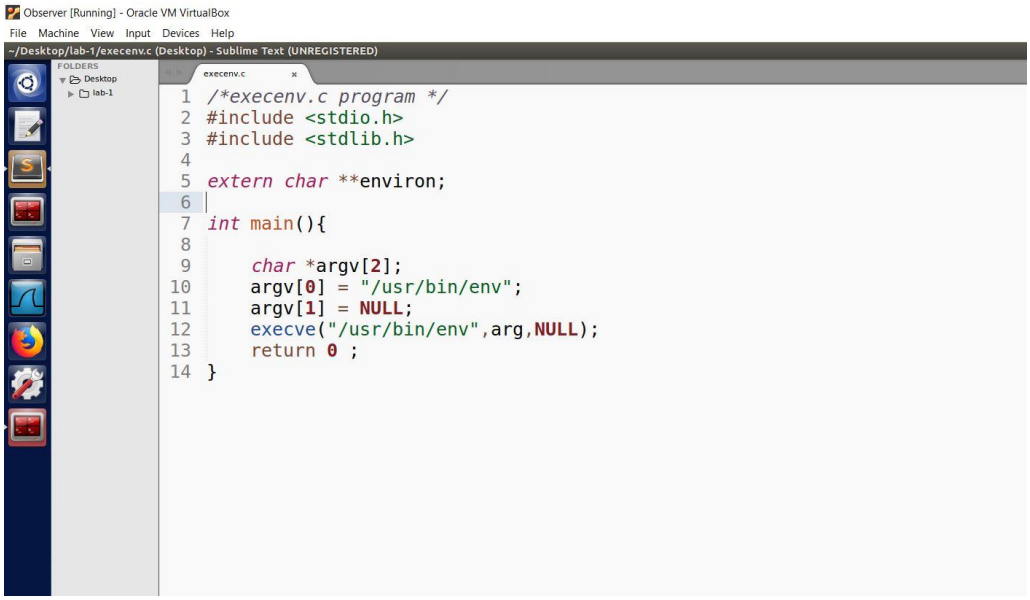


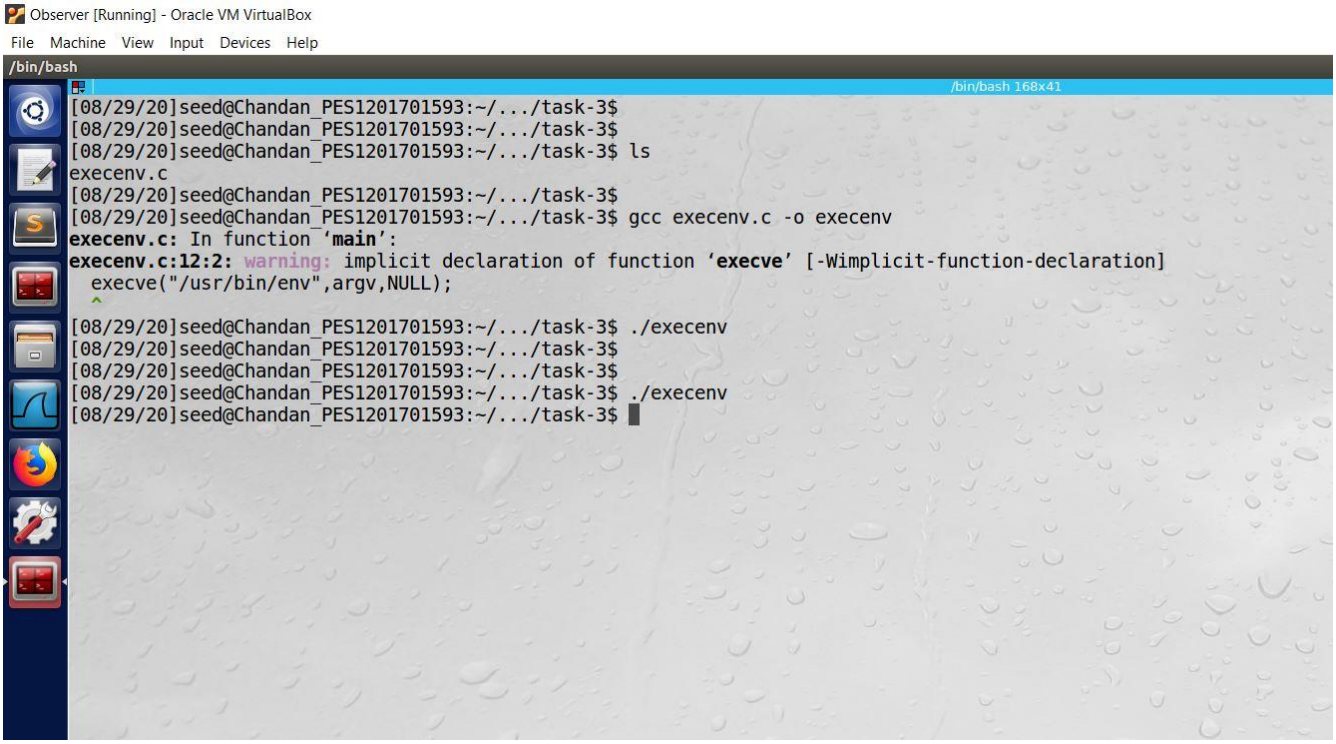
Thus we observer that the diff command output is empty indicating that there is no difference between the files child.txt and parent.txt which contain the environment variables of the child and parent process. Thus, we can conclude that the child process inherits the environment variables from its parent process. But additionally, a parent process may use CreateProcess() to create a child process and pass a new set of environment variables to it.

Task 3: Environment variables and execve()

Now we will understand how environment variables are affected when a new program is executed via execev(). The function execve() calls a system call to load a new command and execute it. No new process is created but the calling process's attributes are overwritten by that of the program overloaded. We will observe if the environment variables are inherited by the new program.

1. We will run a C program which executes the program ' /usr/bin/env ' using execve().

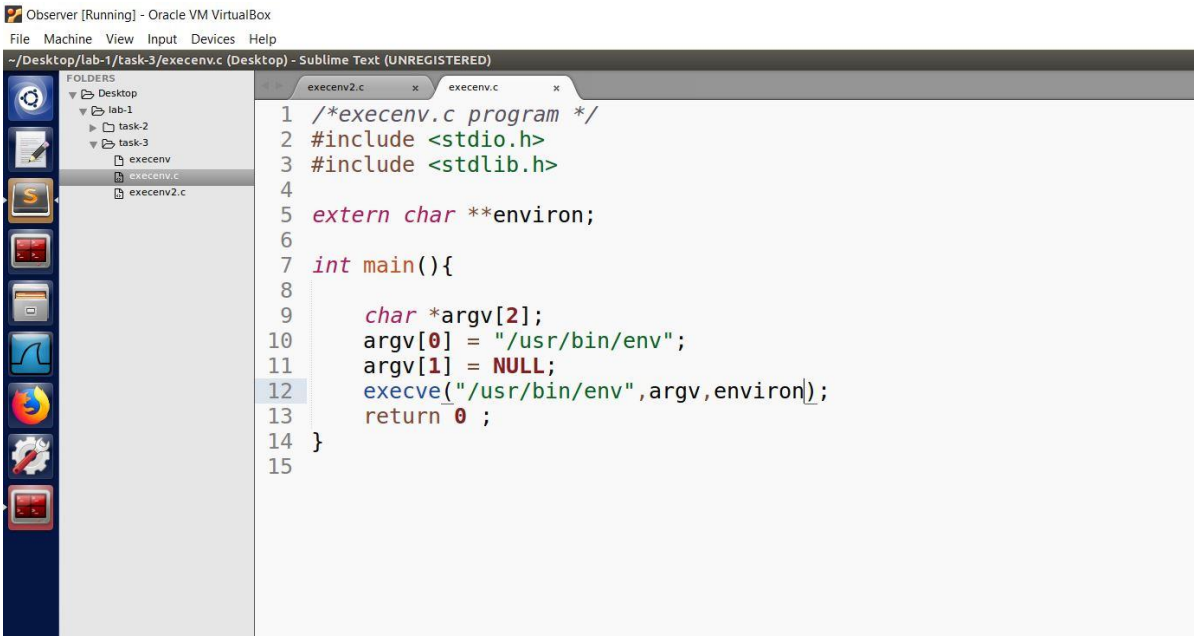


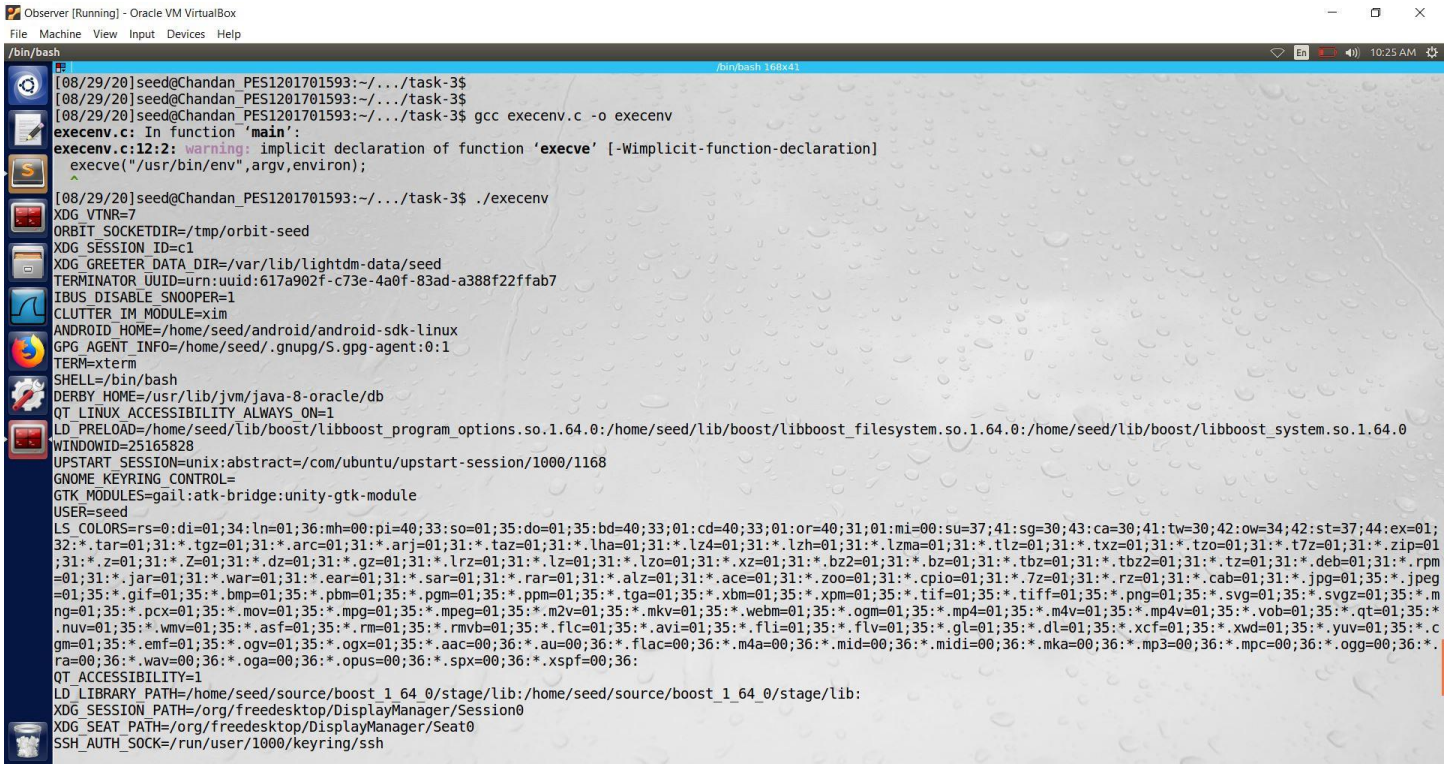


We will observe that there is no output printed in the standard output, thus we can infer that no environment variables are inherited the the new program when executed via execve() sys call.

Next, we tweak the program a bit and observe the output again.

2. Next we change the invocation of execve() to – execve("/usr/bin/env", argv , environ).





The above screenshot shows that on running the program with the above-mentioned changes, it prints a list of environment variables. Thus, we can observe that environment variables are inherited by the new program from the calling process when the execve() system call is invoked with the 3rd argument as "environ" instead of "NULL".

Reading from the documentation of execve() it takes the new environment of the process as an argument. Thus passing "NULL" as the argument creates the new process with no environment variables while passing "environ" as the argument creates a new process with user environment.

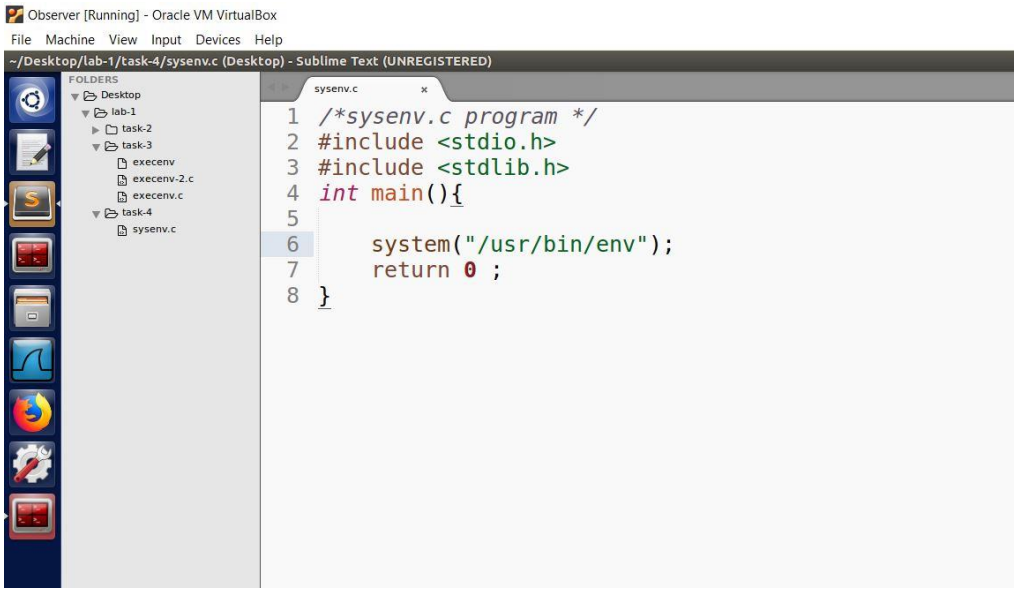
Task 4: Environment variables and system

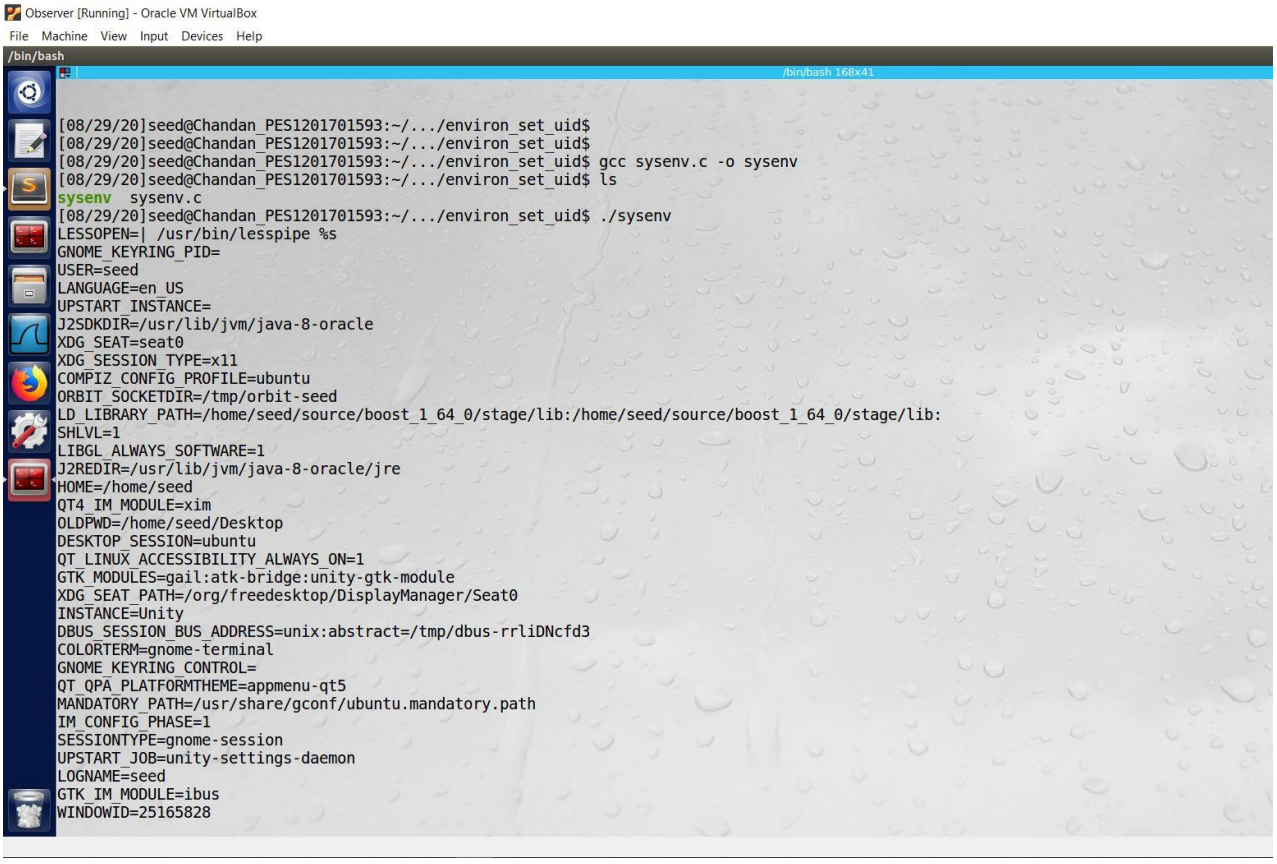
Now we will understand how environment variables are affected when a new program is executed via system(). This is similar to execve() except the fact that system() actually executes "/bin/sh -c command" and asks the shell to execute the command.

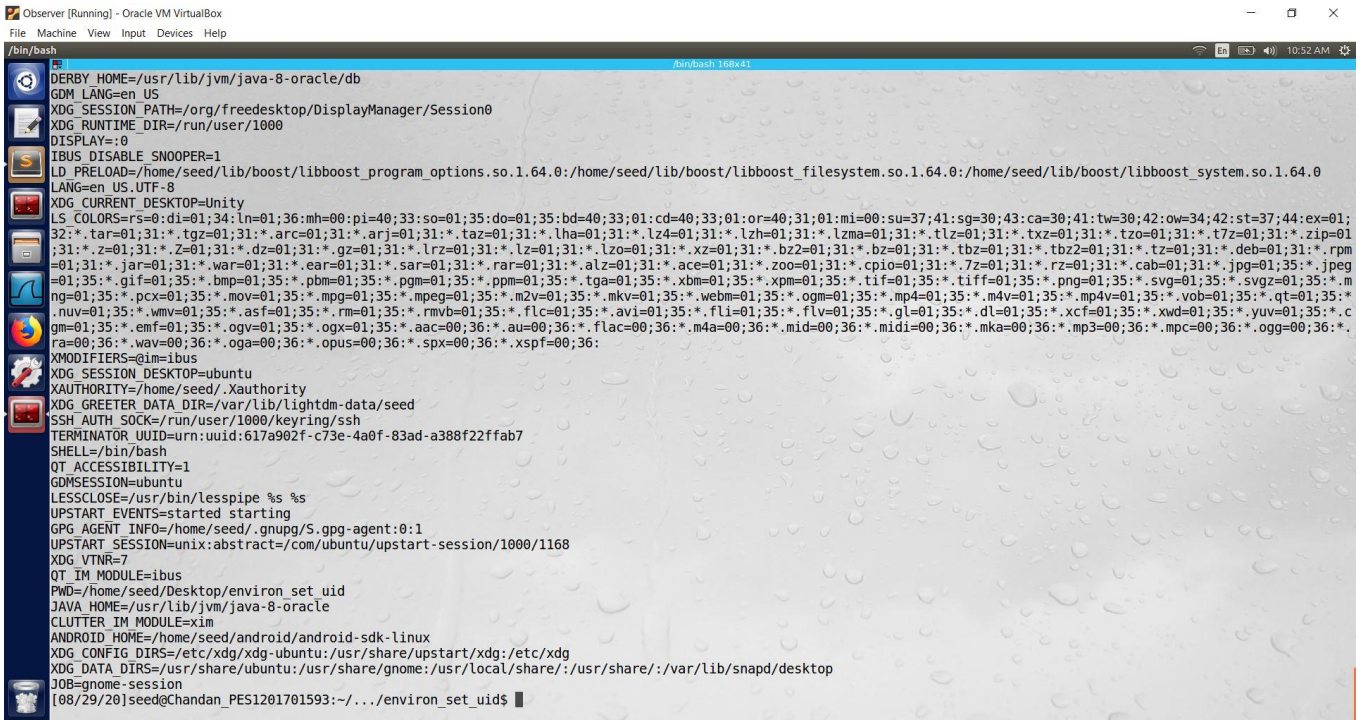
system() = execel() + execve()

Therefore using system(), the environment variables of the calling process is passed to the new program /bin/sh

1. We first run a C program that uses the system() call that runs "/usr/bin/env".



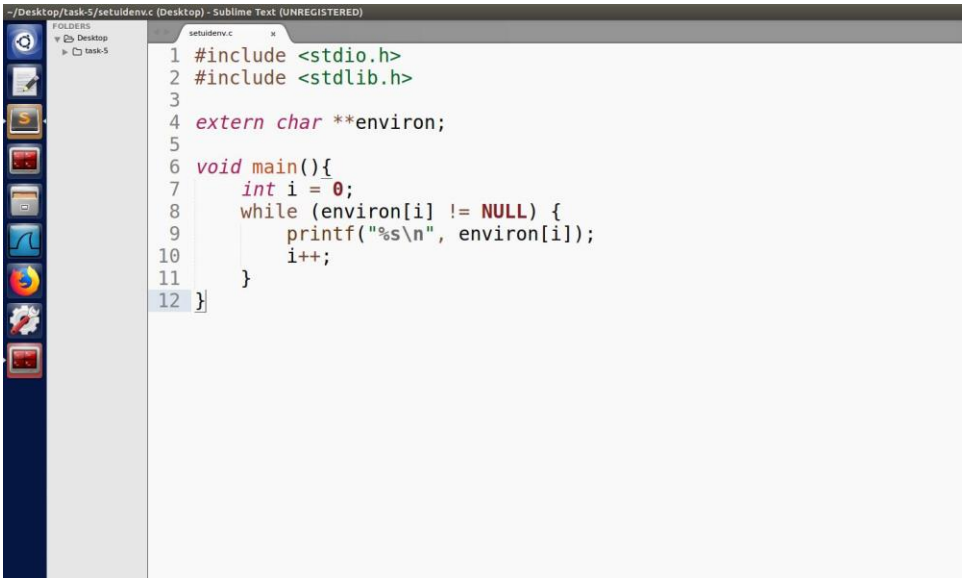




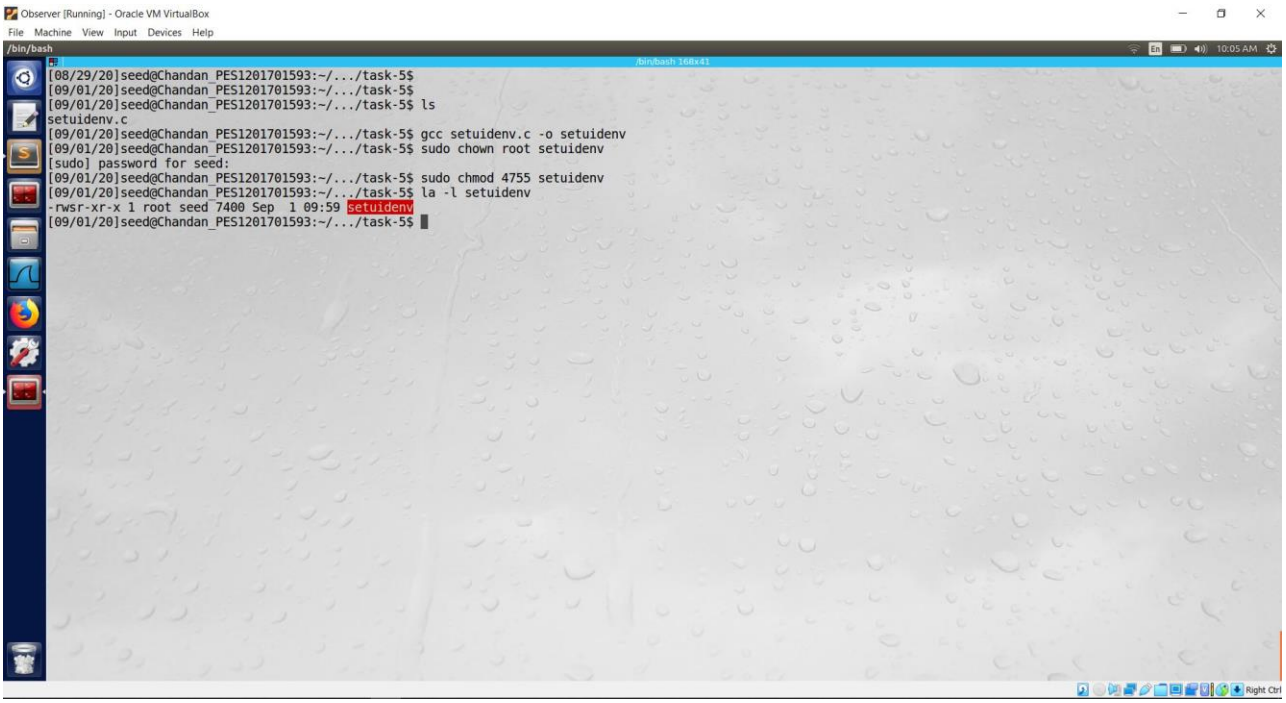
Task 5: Environment variables and Set UID programs

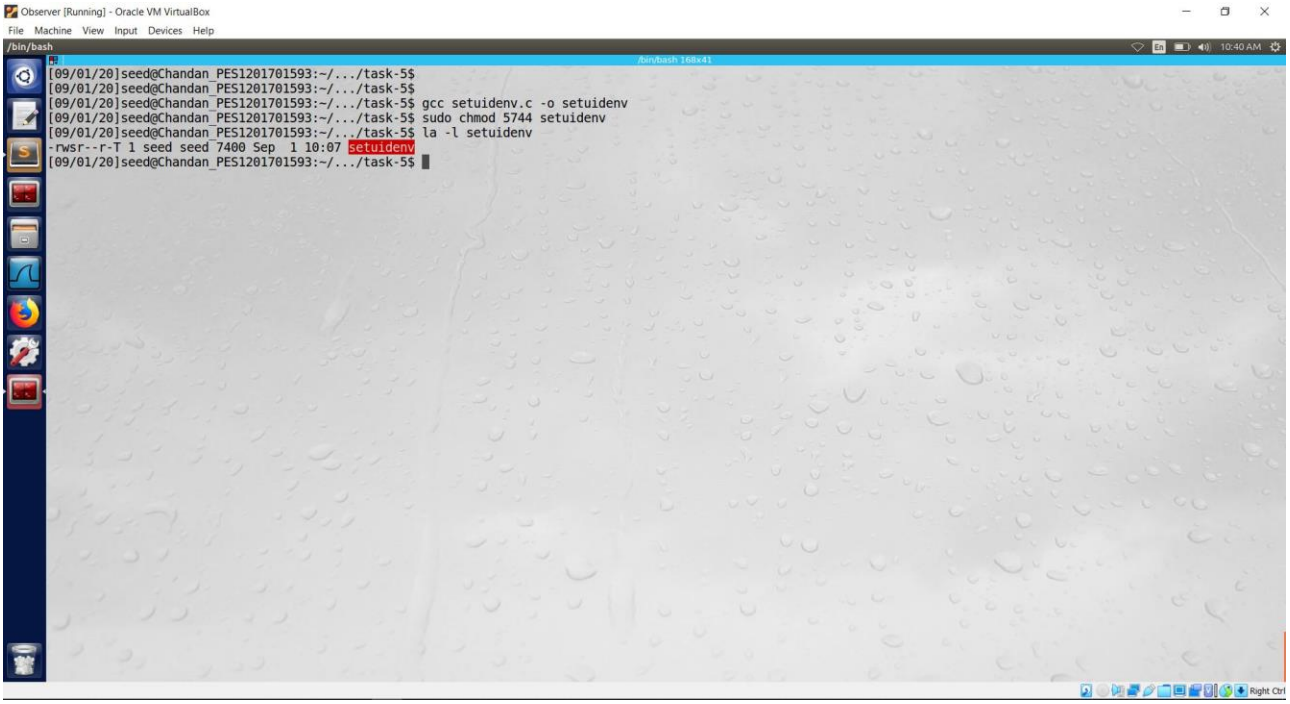
Here we will explore Set-UID programs which are security mechanism in linux. A Set-UID program assumes root privileges when run, the owner of the program is root. This escalation of the privileges makes it risky, and its behaviour can be affected by environment variables. We will observe if environment variables are inherited by Set-UID program's processes from user's process.

1. We write a program that prints all the environment variables of the current process

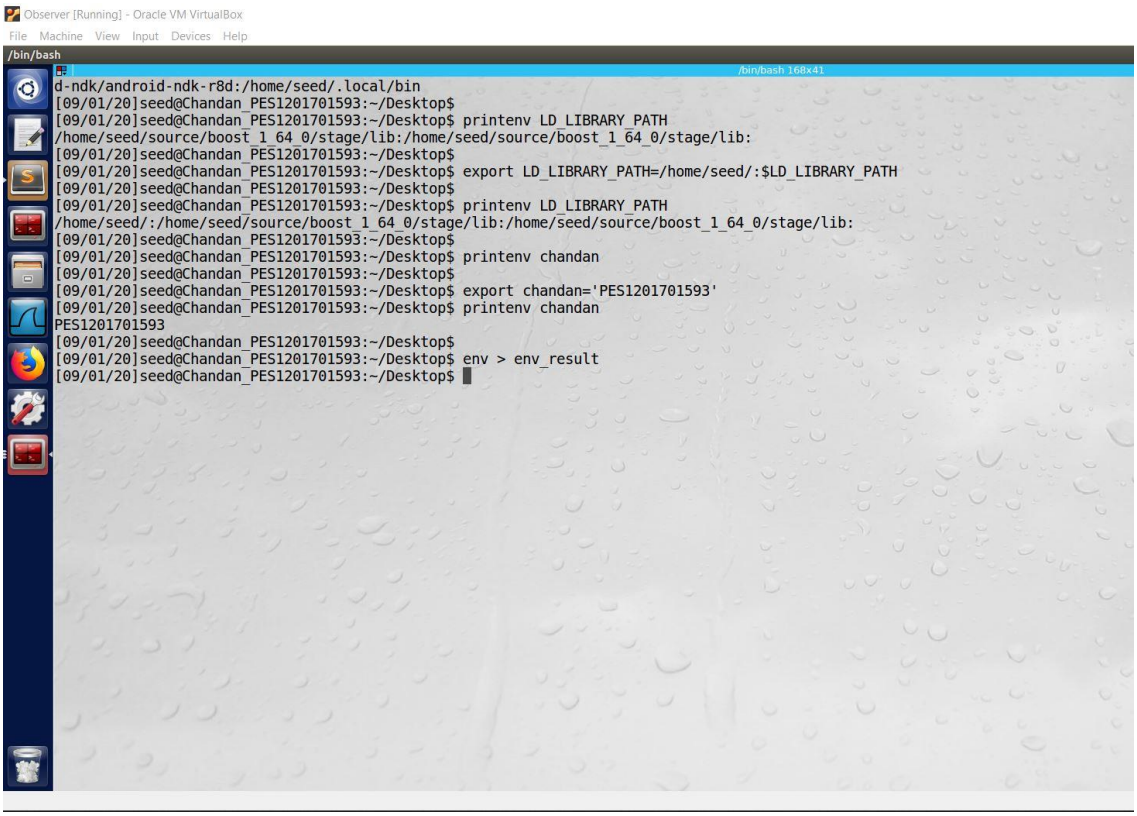


2. We compile the above program, change its ownership to root and make it a Set-UID program

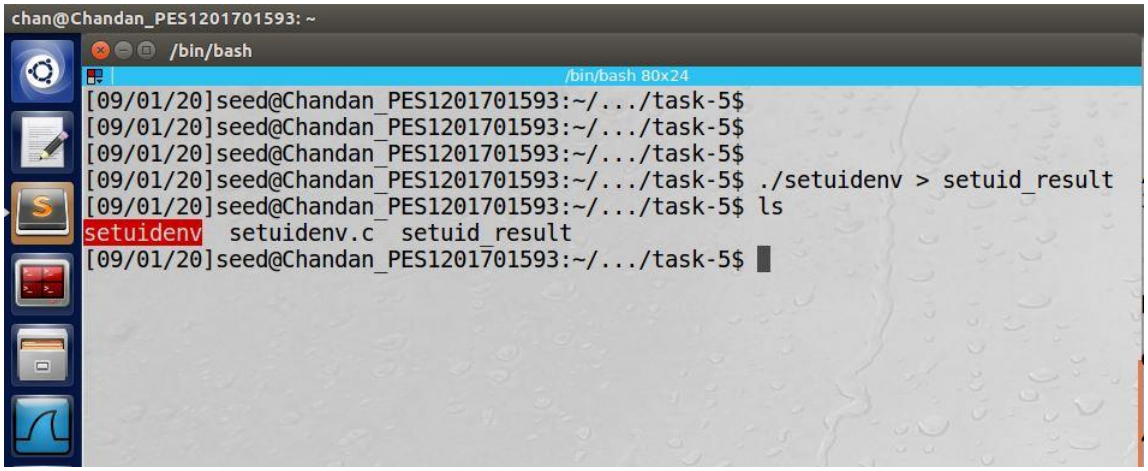




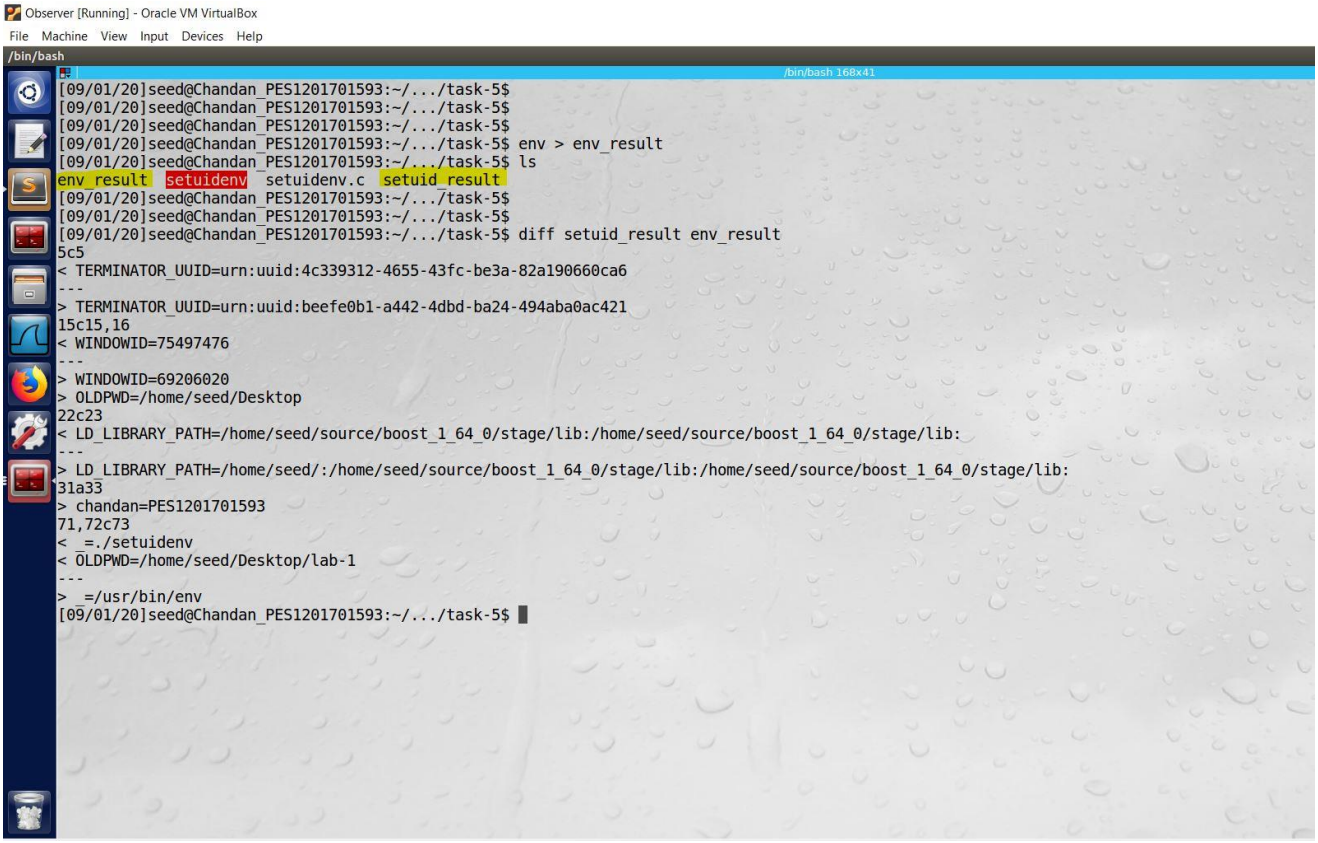
3. In the bash shell, from a non-root account we set the below environment variables using export command and direct its environment variables to env_result file. These environment variables are set in the user's shell process.



4. We run the program setuidenv, which is a Set-UID program and direct its output(env vars of current process) to another file setuid_result file. The shell forks a child process, which further runs the program.



5. Using the diff command we can compare both the env_result and setuid_result to check whether all the environment variables set in parent(shell process) are inherited by the Set-UID child process. We can conclude that the SET-UID program's child process may not inherit all the environment variables of the parent process, LD_LIBRARY_PATH being one of them. We have seen why this happens in task 7 of this lab.



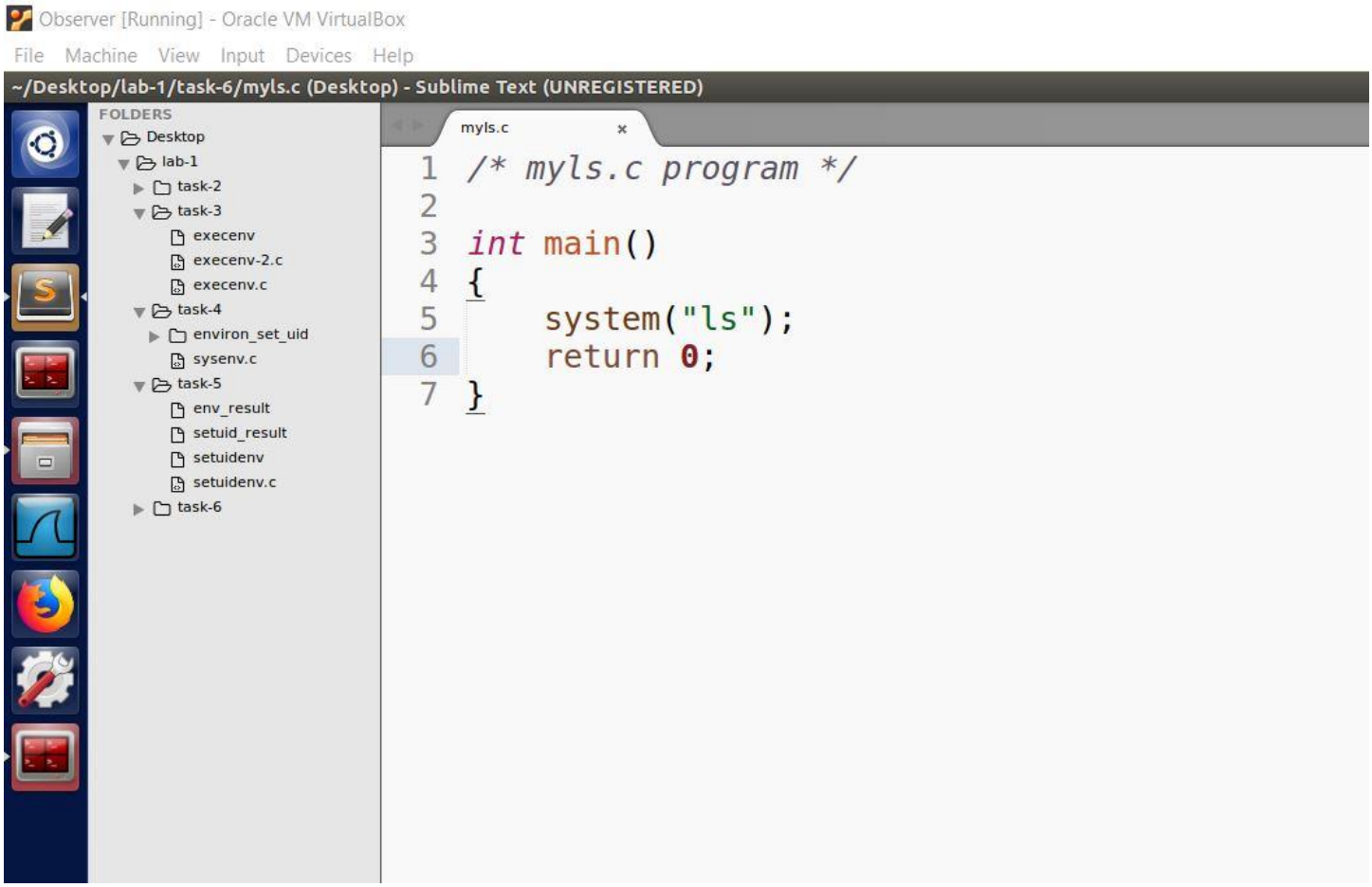
Task 6: The PATH Environment variable and Set-UID Programs

Calling system() within a Set-UID program is quite dangerous because of the shell program that it invokes. The actual behaviour of the shell can be affected by environment variables such as PATH. By changing these variables, malicious users can control the behaviour of the Set-UID program.

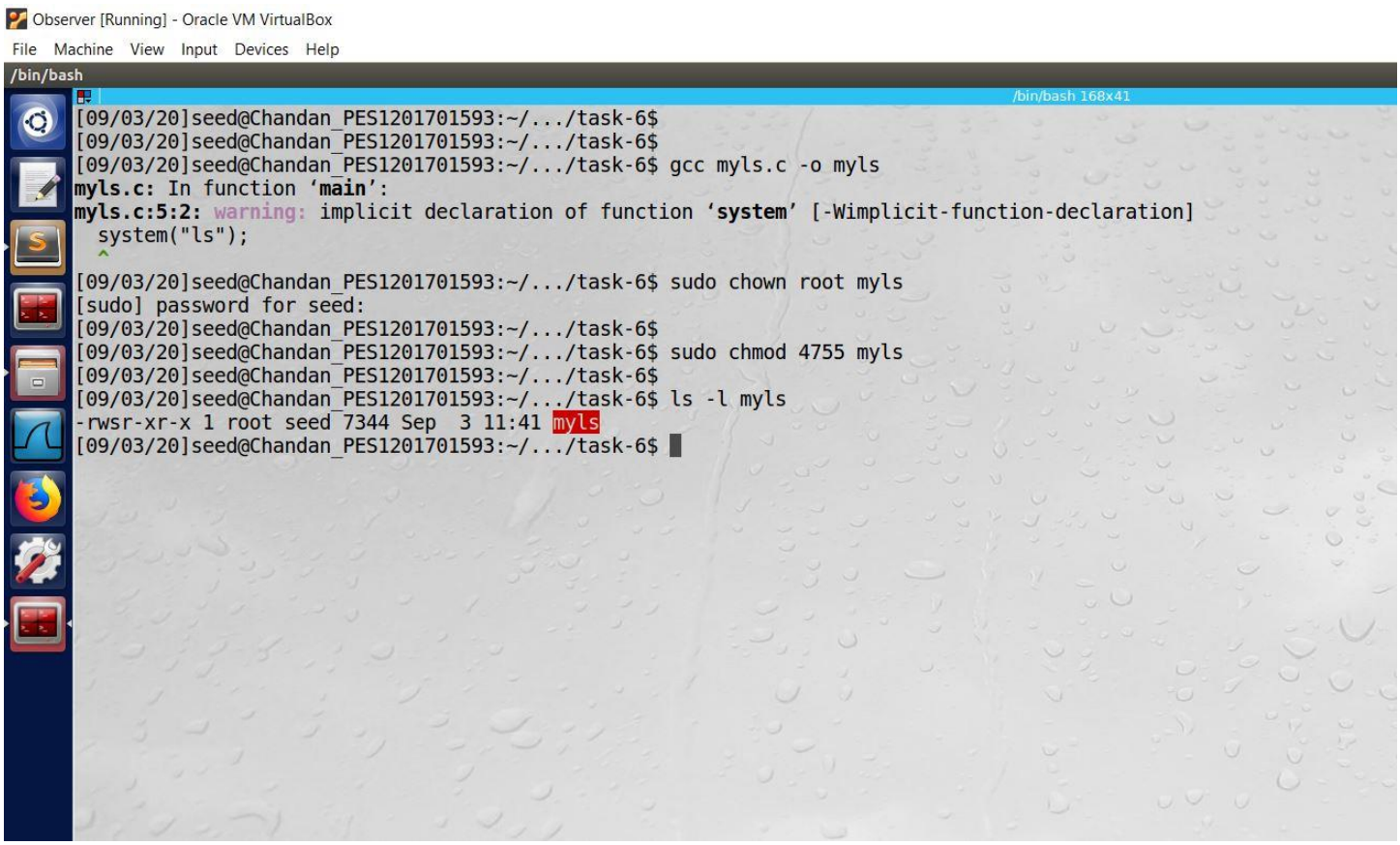
The PATH environment variable in bash can be changed using the export variable.

" export PATH=/home/seed:$PATH "

The below Set-UID program is supposed to execute the /bin/ls command, however, the programmer specifies just the relative path for "ls", than its absolute path.

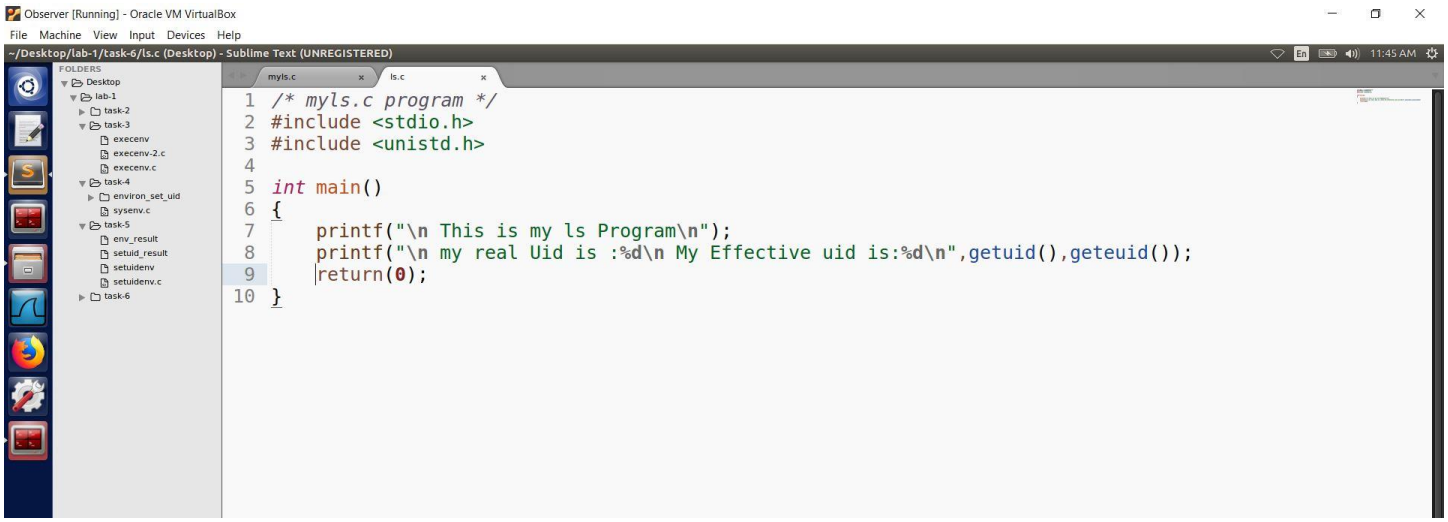


We compile the above program, change its ownership to root and make it a Set-UID program.

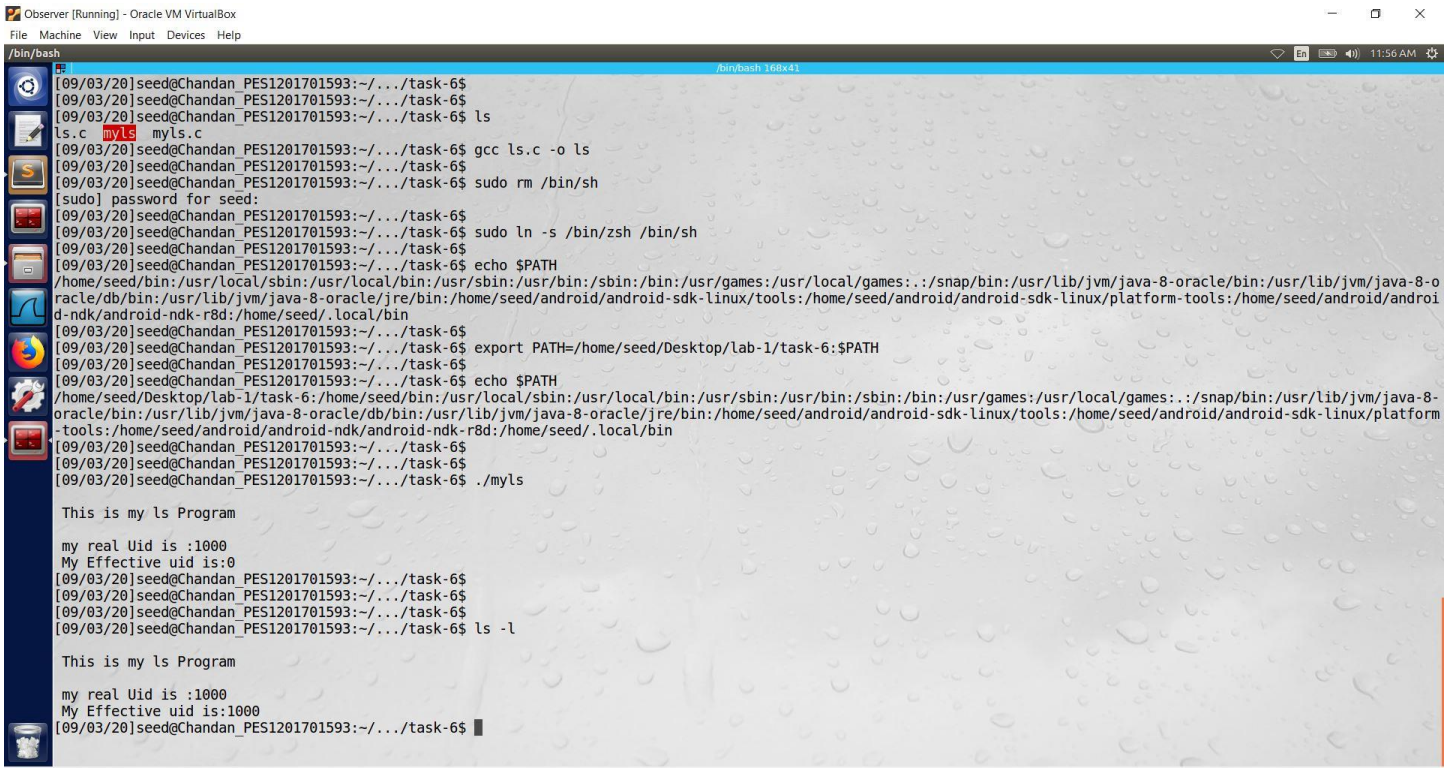


We can confirm that the above program is a Set-UID program by observing “-rwsr-xr-x” and its owner is set to root.

Now we will try to run our code instead of “/bin/ls” using the Set-UID program. In order to achieve this we first compile a program called ls which will be executed instead of /bin/ls. This program that we compile can have any logic that we/malicious user wants to achieve, for now we will just print few things.



The above code is compiled to an executable called “ls”. Next, we change the value of the environment variable PATH, using the export keyword, and provide the path to our file, which could be malicious. This makes the program to search for the file in our directory, and as we have compiled our program as “ls”, the “myls” Set-UID program will in-turn execute our program (our “ls” program) instead of the /bin/ls program.



In order to overcome the dash shell security mechanism of dropping SET-UID program's privileges, we link the "/bin/sh" to another shell as shown in the above screenshot.

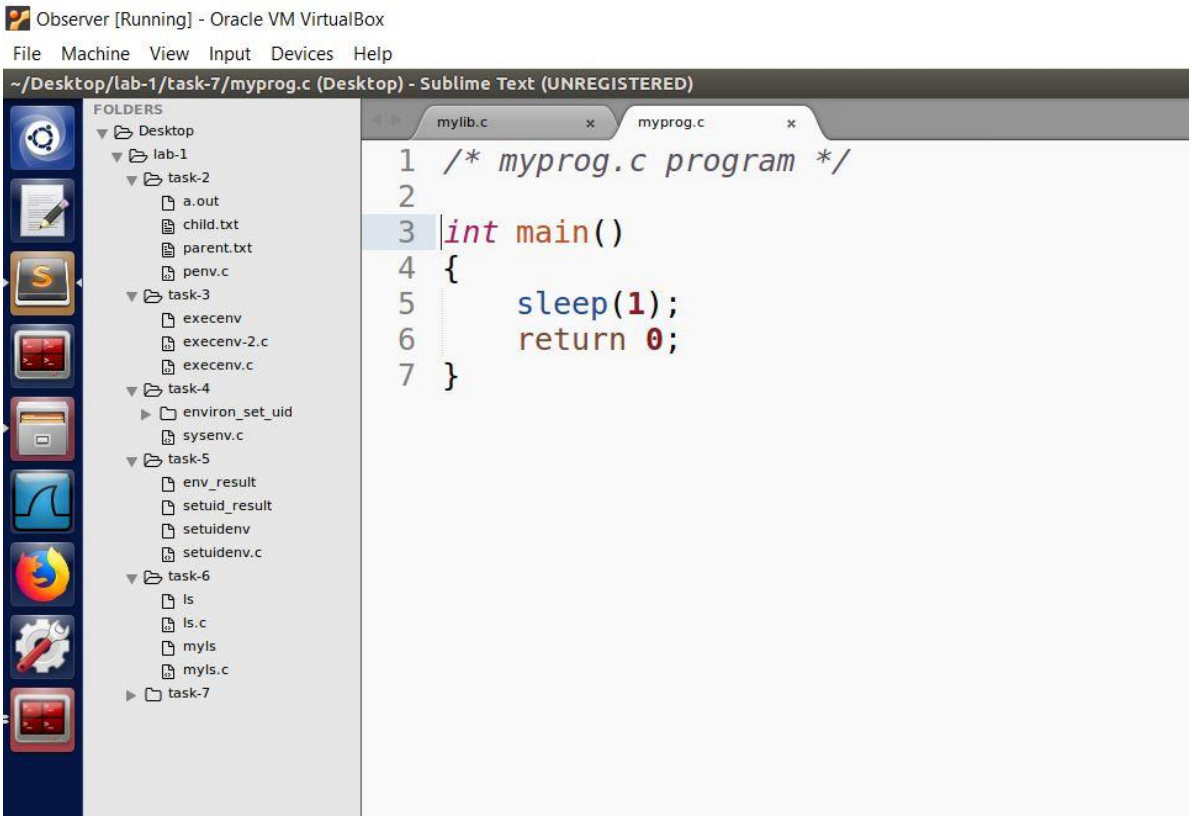
Thus, we observe that the environment variables can be altered to control the behaviour of the Set-UID program specially when the shell is invoked. Moreover, we used a relative path instead of absolute path. As a result the system() invoked a shell which looks for a ls program in the "PATH" which was modified. Therefore changing the PATH environment variable to a folder containing malicious file with the same name as the program called by system(), the attacker can easily run the malicious code with root privileges.

A Set-UID program with system() with relative path of the program can be a major vulnerability.

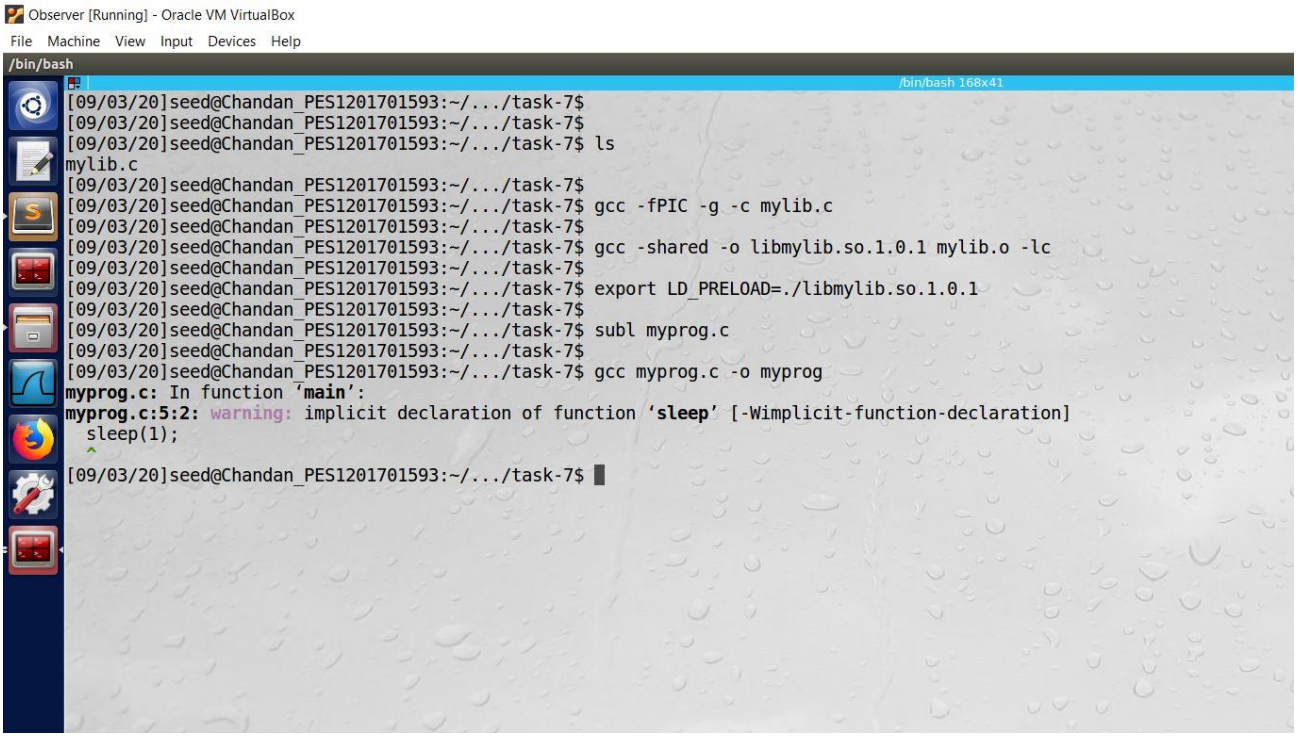
Task 7: The LD_PRELOAD Environment Variable and Set-UID Programs

Here we will observe how Set-UID program deals with environment variables such as LD_PRELOAD, LD_LIBRARY_PATH etc. which influence the behaviour of the dynamic linker/loader in linux.

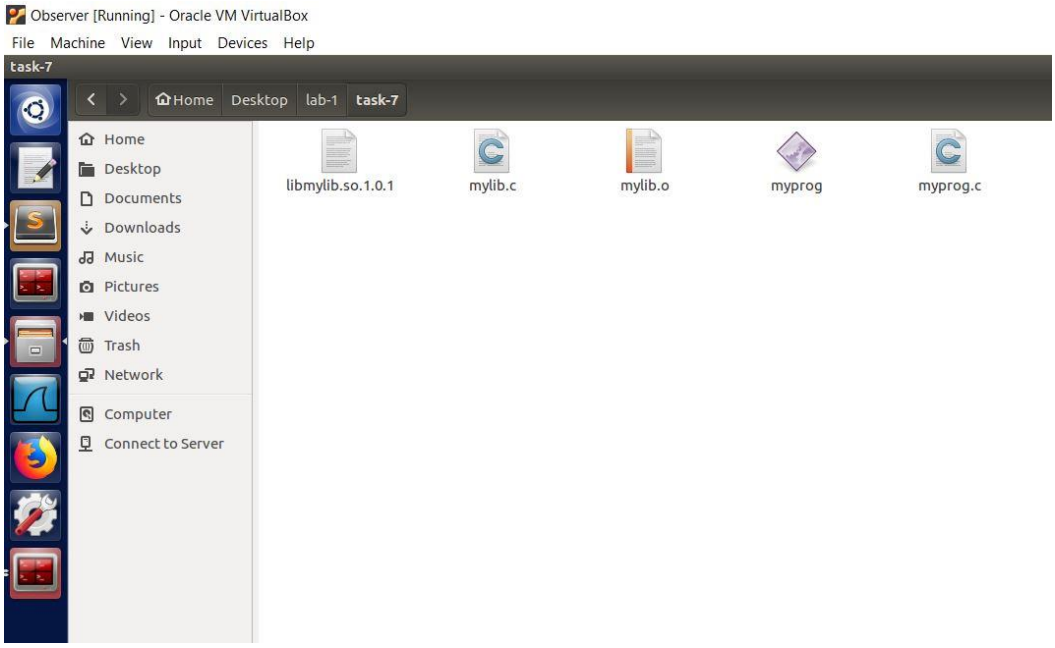
1. First, see how these environment variables influence the behavior of dynamic loader/linker when running a normal program.
   We first build a dynamic link library called mylib.c, it overrides the sleep() in libc.



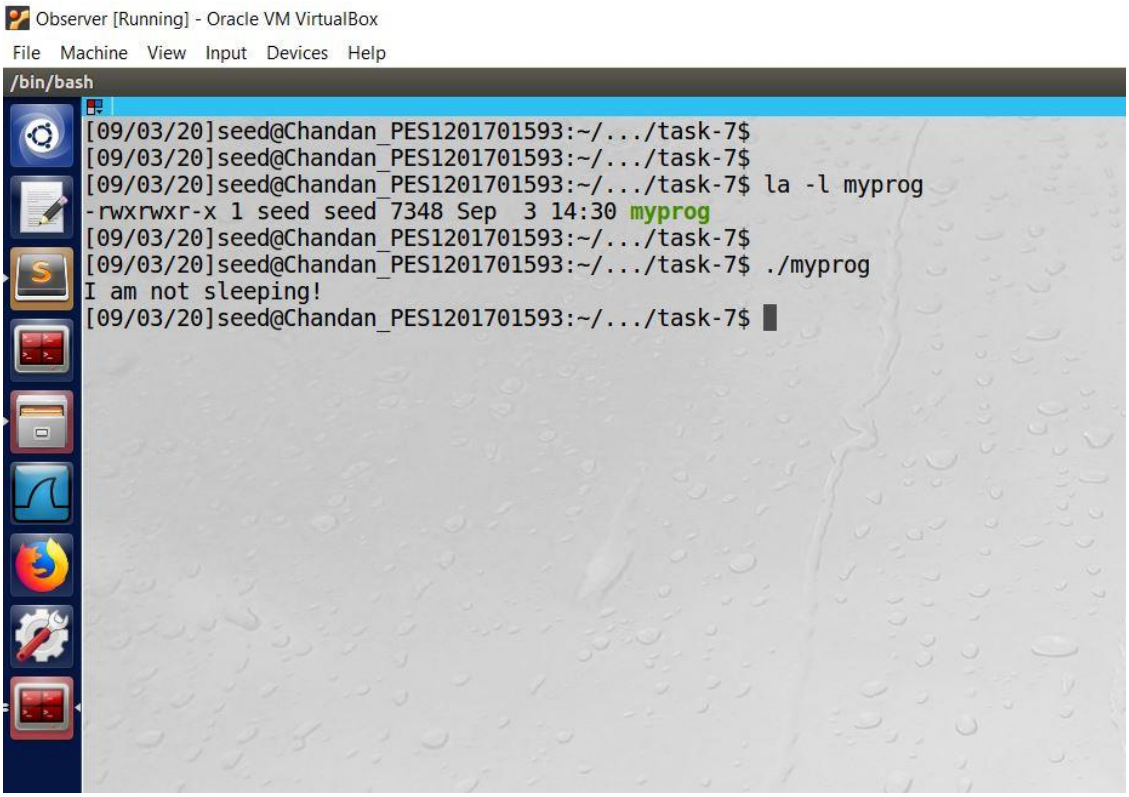
2. We compile the above program. Then set the LD_PRELOAD environment variable. Finally we compile the myprog program in the same directory as that of the dynamic link library as shown below.



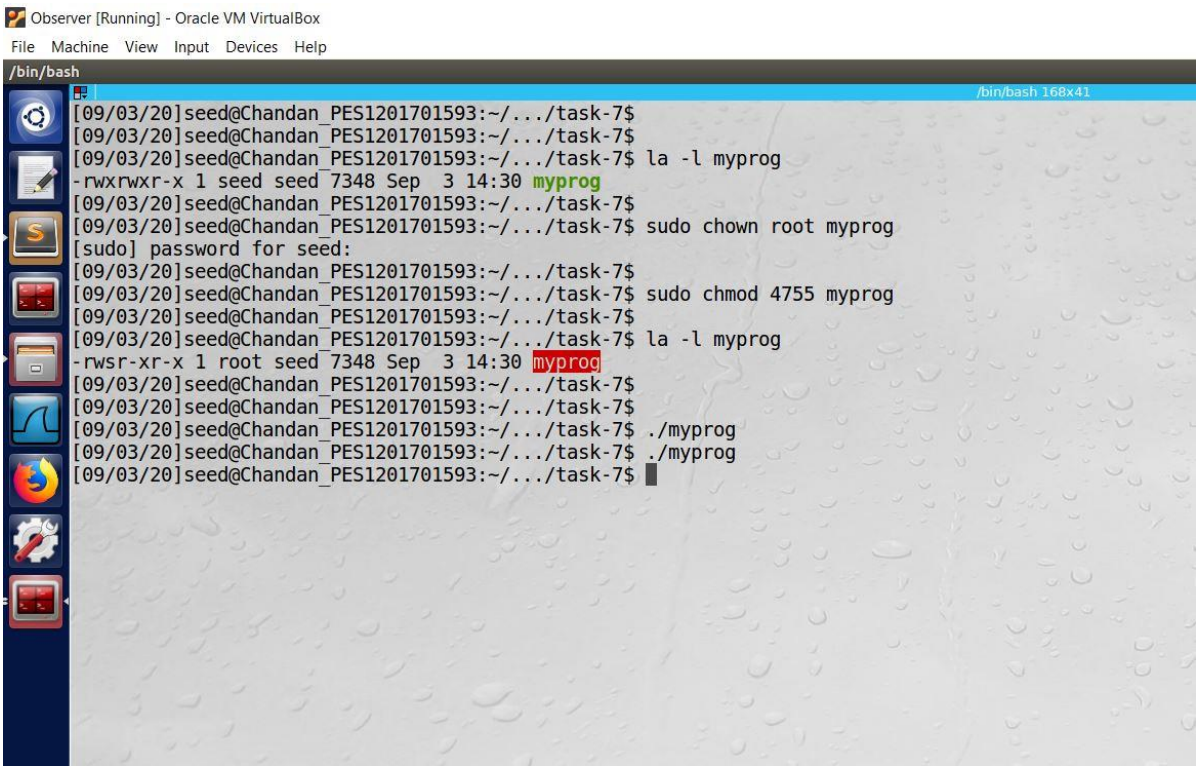
3. We can observe that all files are present in the same directory



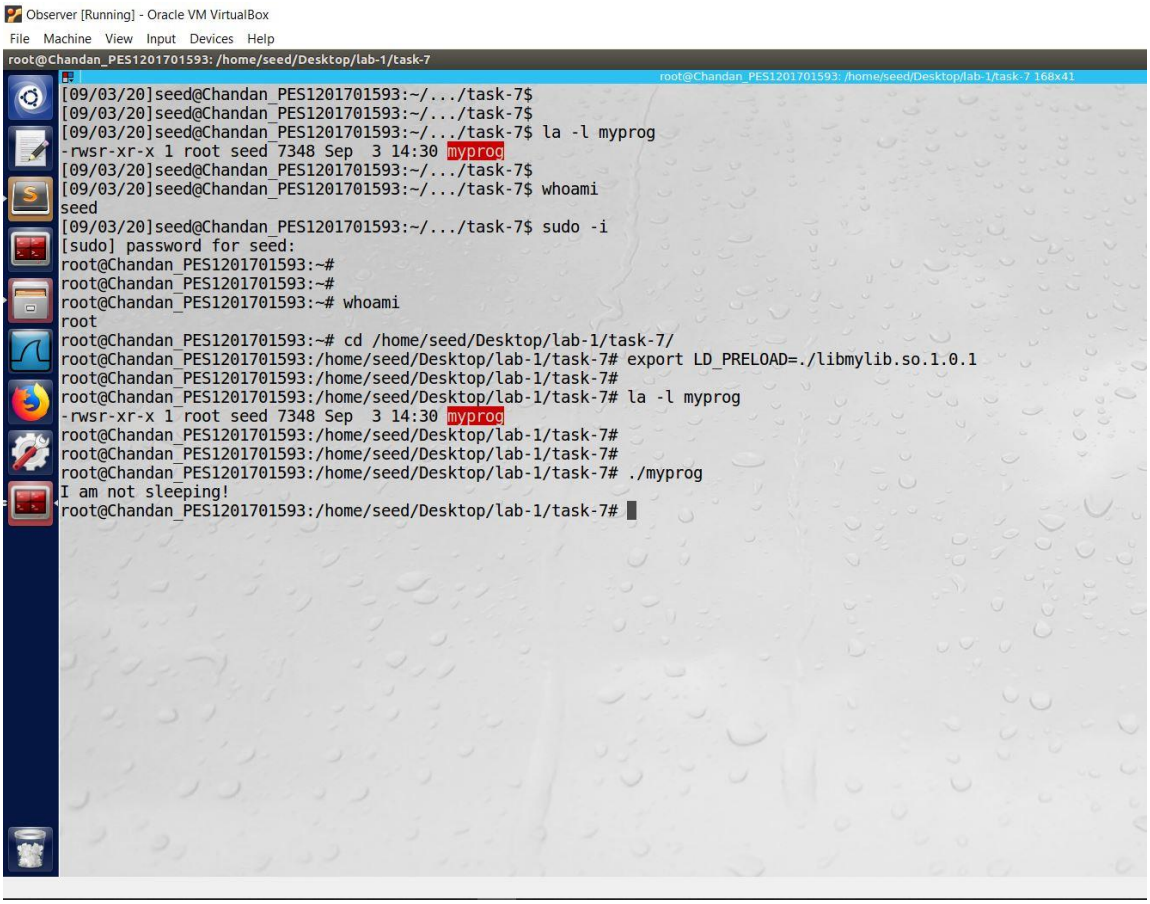
4. Next we run the myprog program under different conditions and observe the result
    a. <u>Make myprog a regular program, and run it as a normal user:</u> In this case we observe that the program calls the sleep() defined by us and also prints what we specified in the sleep() function.



    b. <u>Make myprog a Set-UID root program, and run it as a normal user:</u>
    In this case we observe that the sleep() function in mylib was not called and the system-defined sleep() was called.

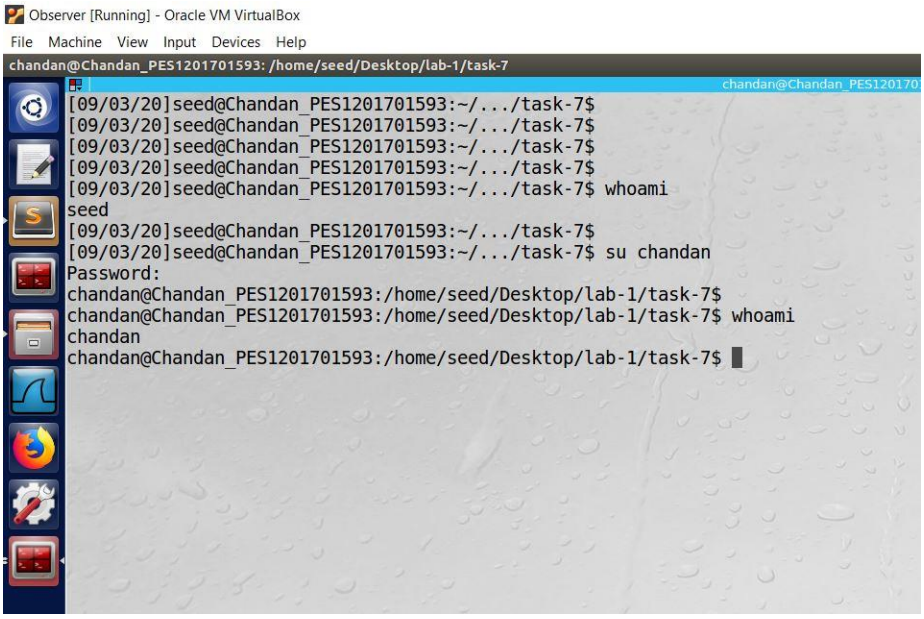


c. Make myprog a Set-UID root program, export the LD PRELOAD environment variable again in the root account and run it:
In this case we observe that the user defined sleep() function from mylib is called instead of the system defined sleep() is called.

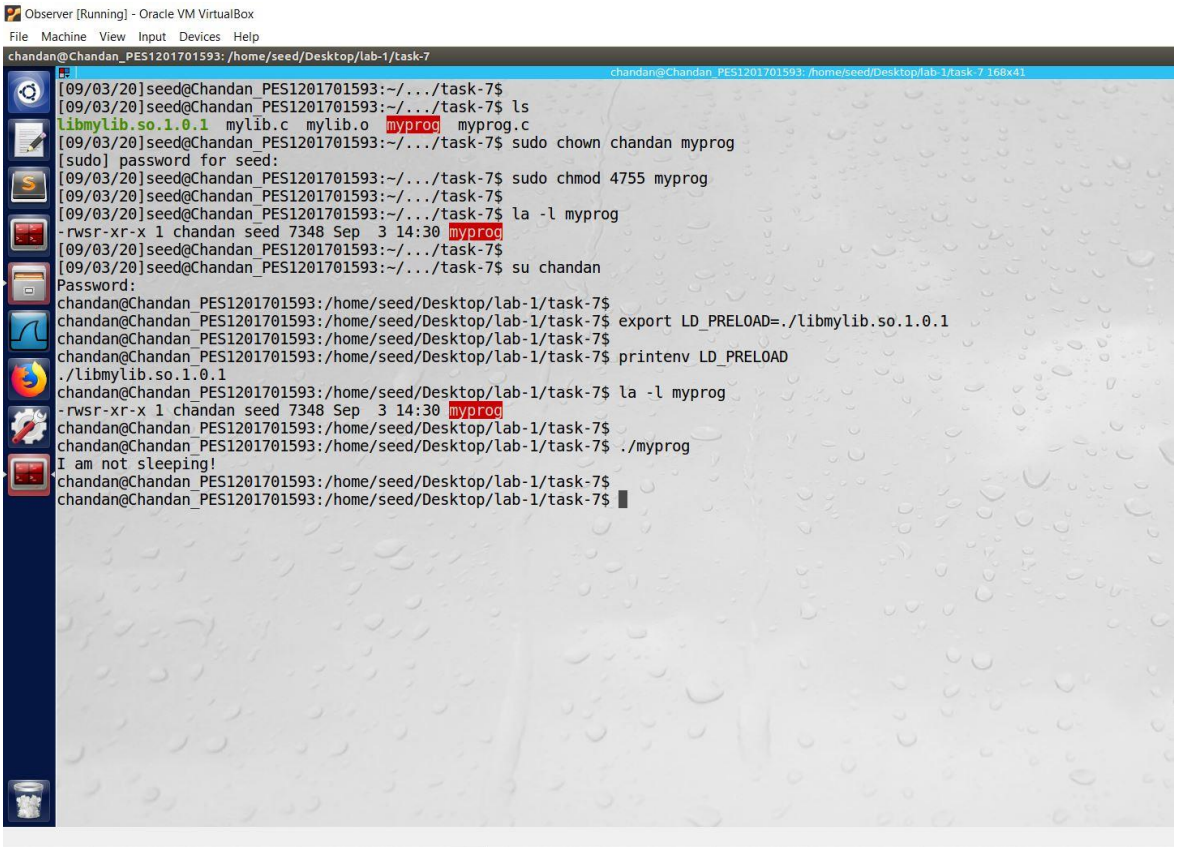


d. Make myprog a Set-UID “chandan” program (i.e., the owner is chandan, which is another user account), export the LD PRELOAD environment variable again in a different user’s account (not-root user) and run it:

We first create a new user account “chandan” as shown below.



Next, we make this file’s owner as chandan and make it a SET-UID program. After this, we log into the chandan’s account and set the LD_PRELOAD variable again. On running the program again, we see that user-defined sleep function is called.



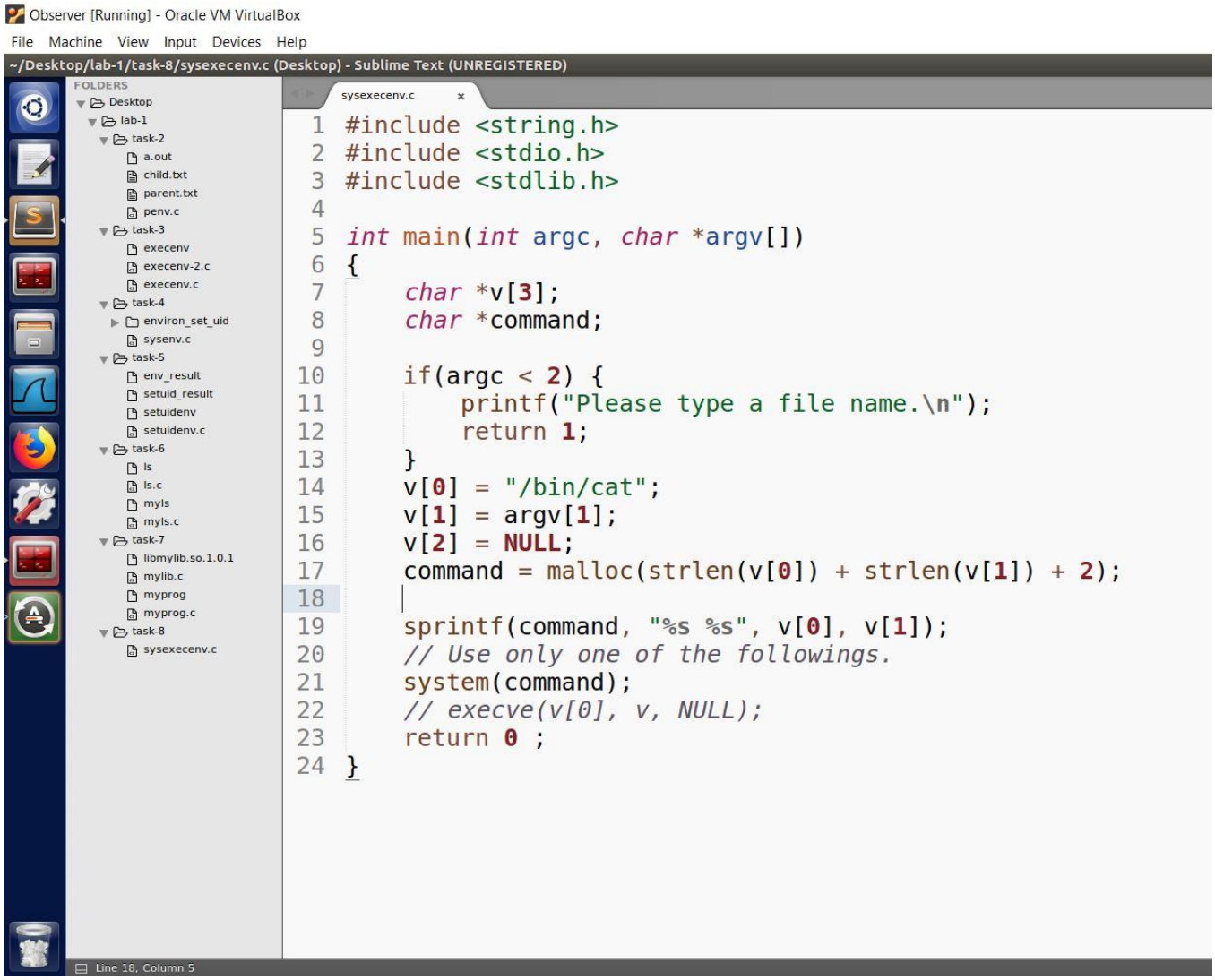
Observing the above 4 cases, we can conclude that in the cases where the user-defined sleep() function was called, the LD_PRELOAD environment variable was present/inherited thus the mylib library function was called. In the other case where the system defined sleep() was called the LD_PRELOAD environment variable was not present as it was dropped by the Set-UID child process.

The LD_PRELOAD environment variable is dropped if the real and effective id's are different. This is because of Set-UID security mechanism as mentioned in this article http://web.archive.org/web/20130711061934/http://www.jayconrod.com/cgi/view_post.py?23 and https://stackoverflow.com/questions/9232892/ld-preload-with-setuid-binary.

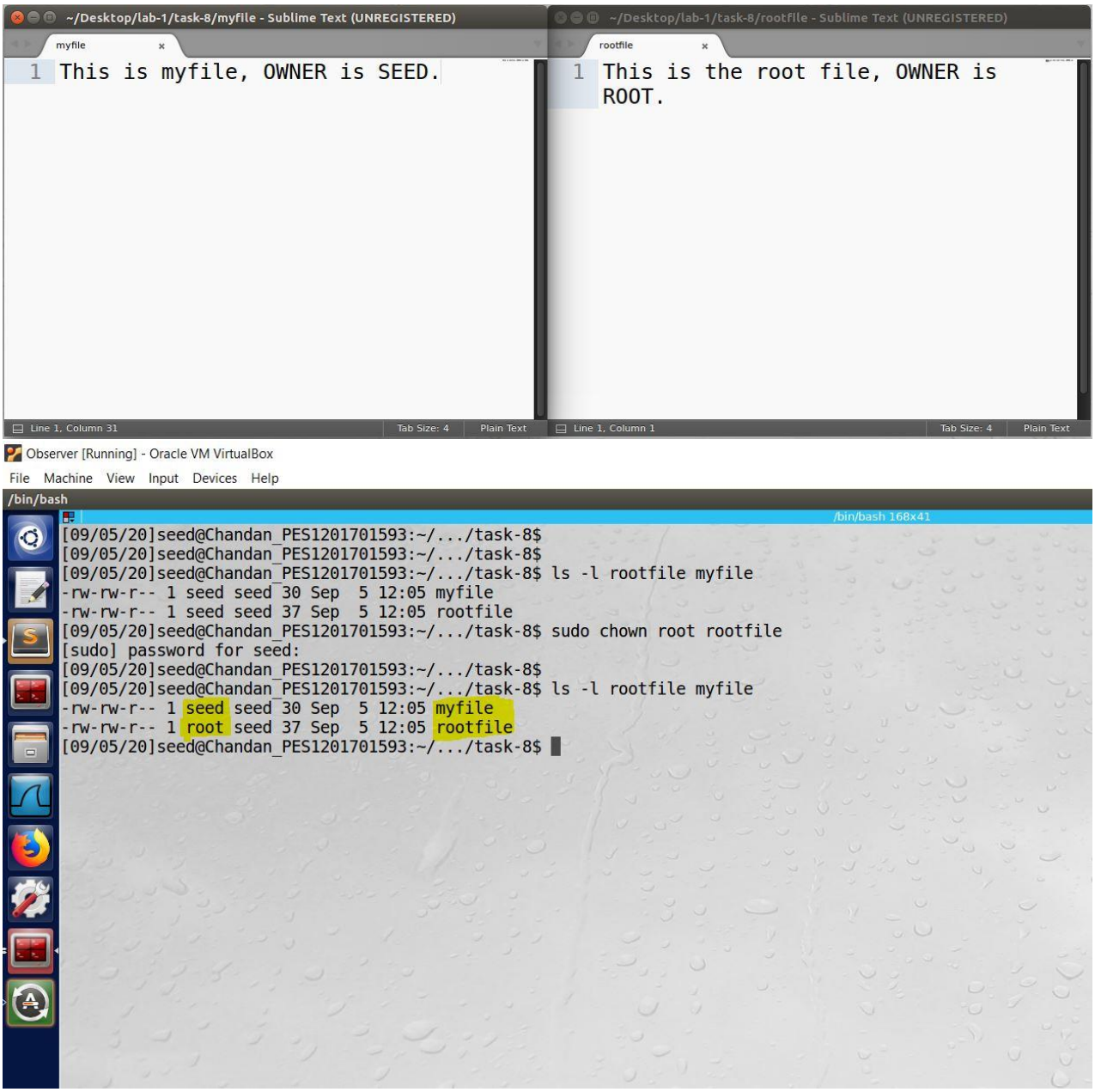
Task 8: Invoking external programs using system() versus execve()

We have observed that system() is quite dangerous when compared execve() when used with Set-UID programs. Apart from affecting environment variables, invoking a shell has various other consequences which we will try to observe in this task.

1. We compile the below program, make root its owner and make it a Set-UID program. The program we run uses system() to invoke the command Bob specifies.

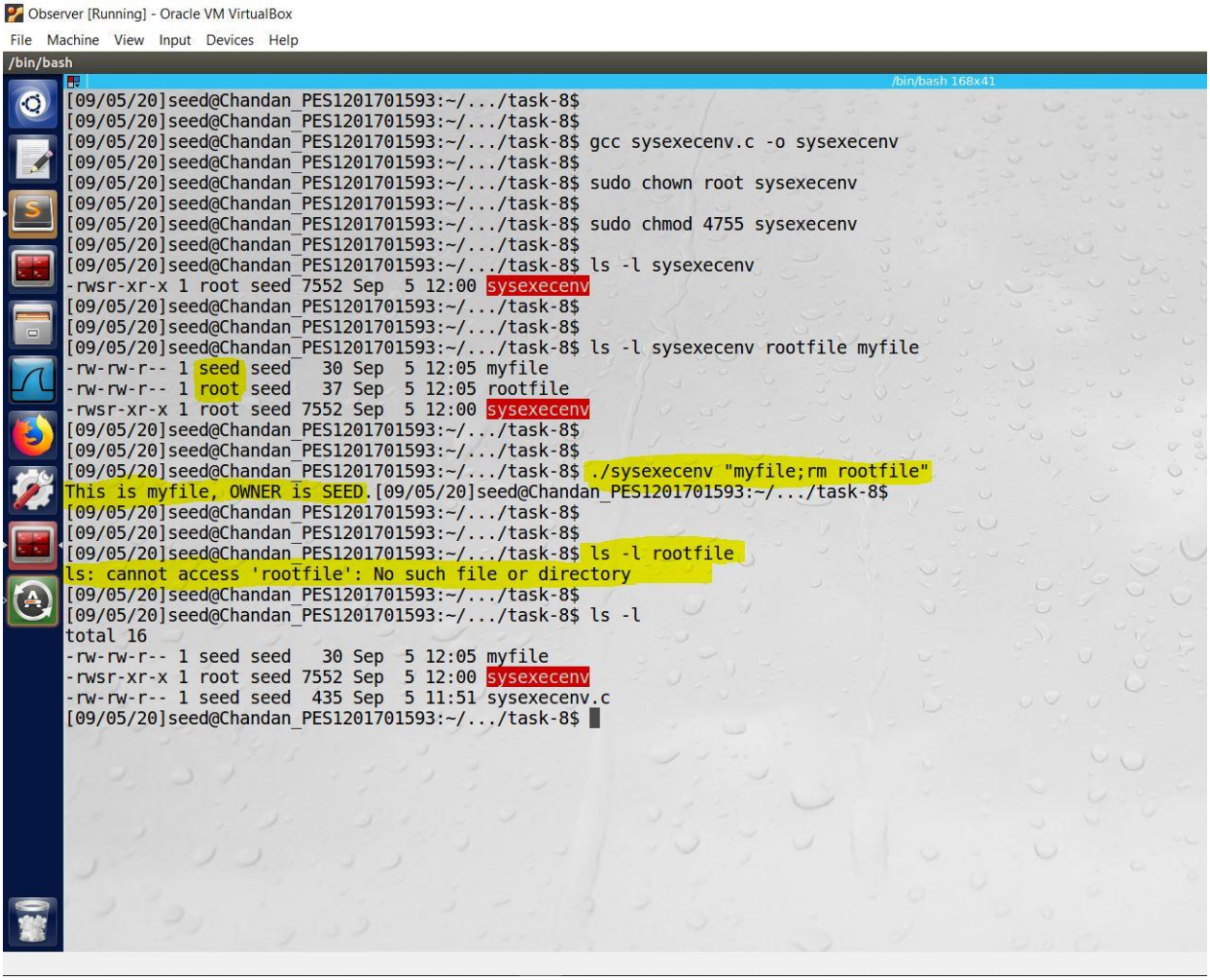


2. We create two files, “rootfile” whose owner is root and “myfile” whose owner is a non-root account (seed).

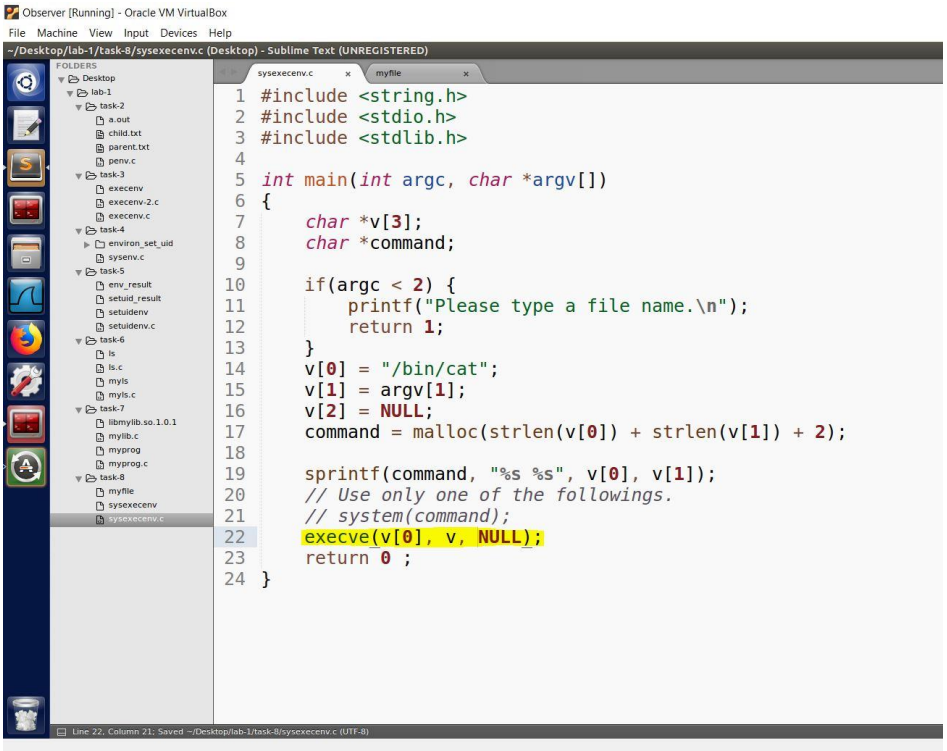


3. We run the Set-UID program sysexecenv which uses system() and passing “myfile;rm rootfile” as the argument to the executable as shown below.

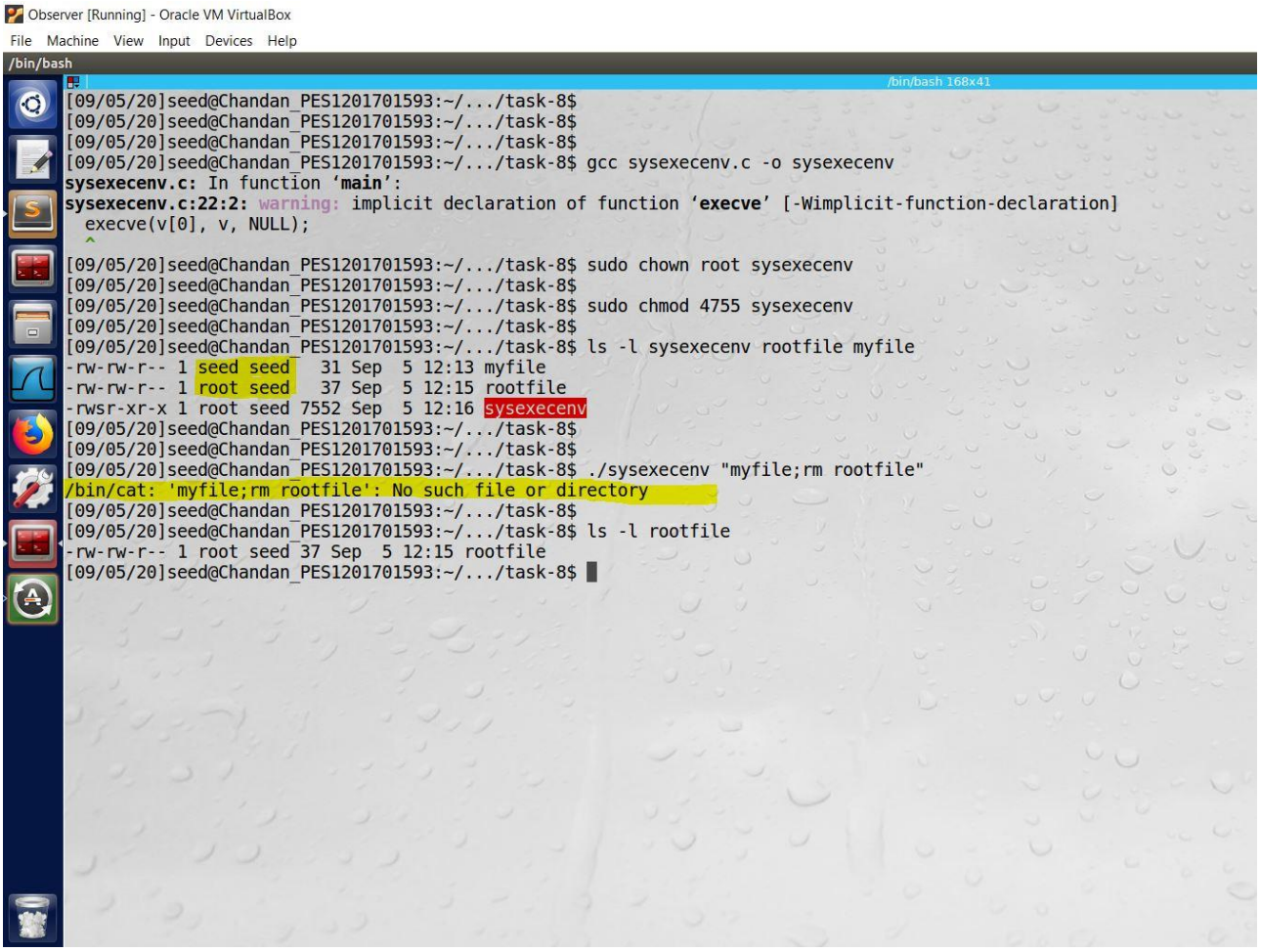
   After executing the sysexecenv program we observe that the “rootfile” is no longer available and is been deleted. If we observe this it is evident that the program reads the entire input i.e. reads the file name “myfile” and later also executes the command “rm rootfile”. Since this being a Set-UID program it has the root privileges even though the program is run by a normal user (Bob). So even if Bob had no permission to write, he can easily remove the file with the root privileges. Thus system() is quite dangerous in case of Set-UID programs.



4. Now we replace the system() call with execve() unlike system() considers the input from user as a string rather than considering it a command. We compile it, make its owner root and make it a Set-UID program.



5. We run the above compiled program with the same string as argument to the executable. But this time instead of considering it as commands, the entire string is treated as user input string thus printing an error 'myfile;rm rootfile': No such file or directory unlike system(). Thus execve is safer that system() especially with Set-UID programs.

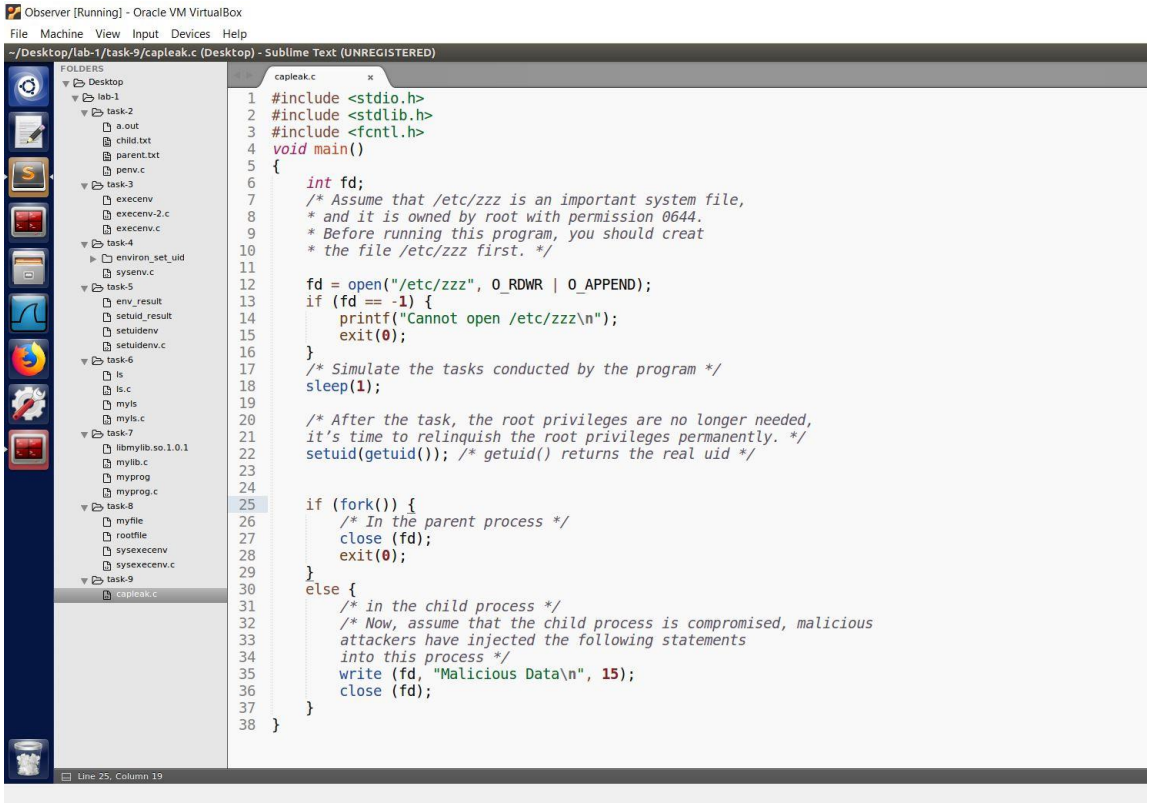


Task 9: Capability Leaking

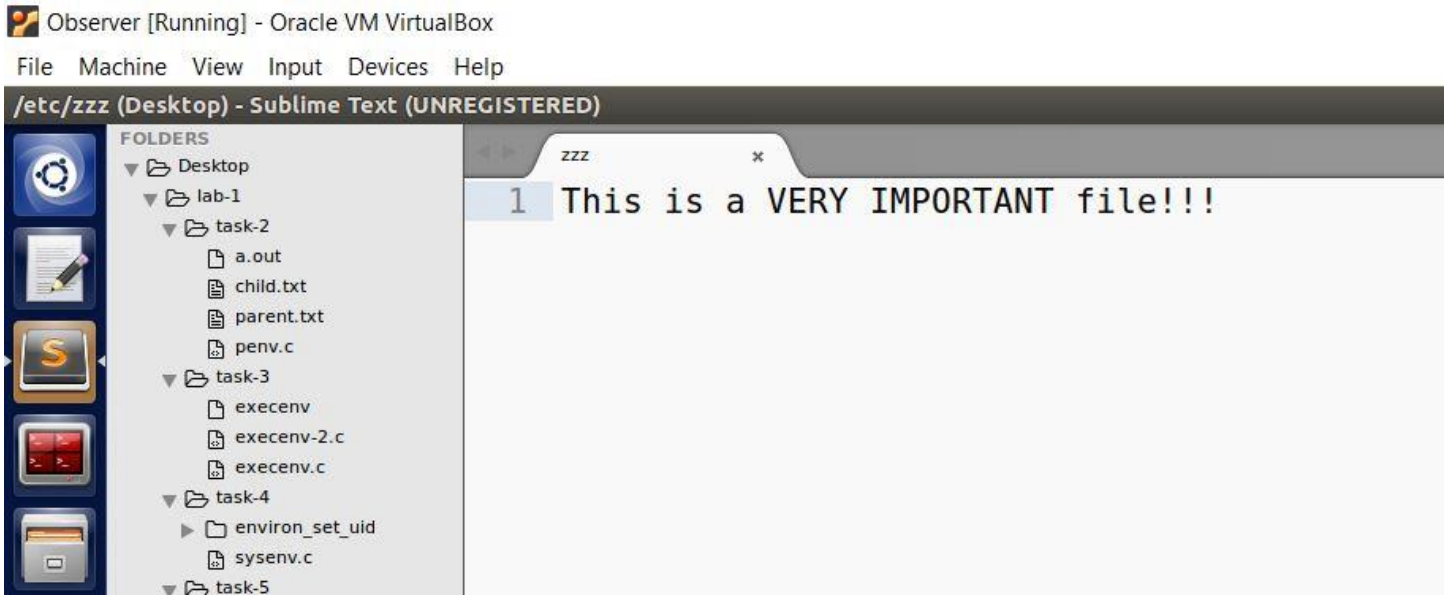
Set-UID programs permanently relinquish their root privileges if such privileges are not needed. But sometimes the root privileges must be revoked and setuid() system call can be used. setuid() sets the effective user ID of calling process and if the effective UID of caller is root, the real UID and saved set-user-ID are also set.

But the above is associated with capability leaking. In this case although the effective user ID of the process becomes non-privileged, the process is still privileged because it possesses privileged capabilities.

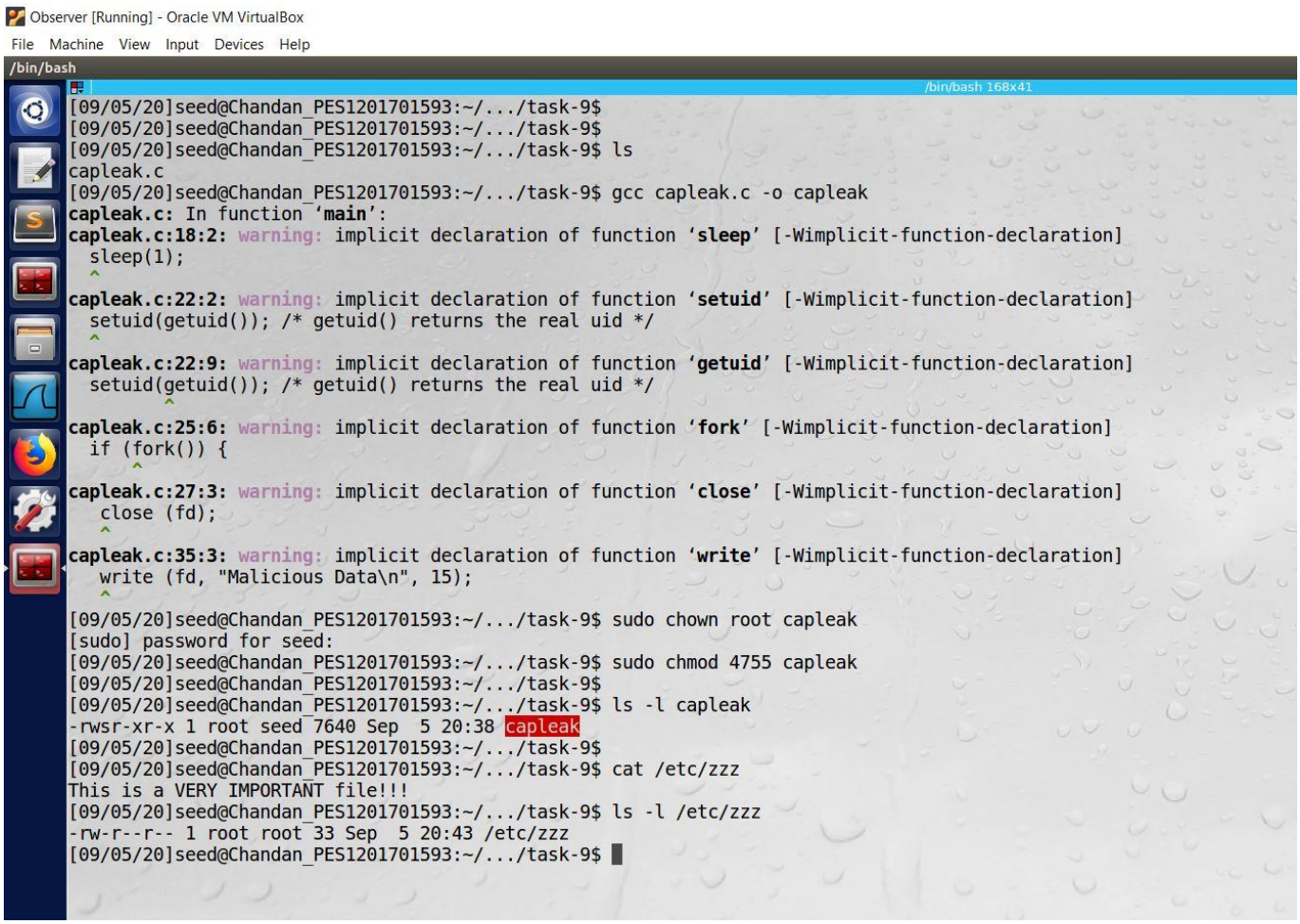
1. We compile the below program, make its owner root and make it a Set-UID program.

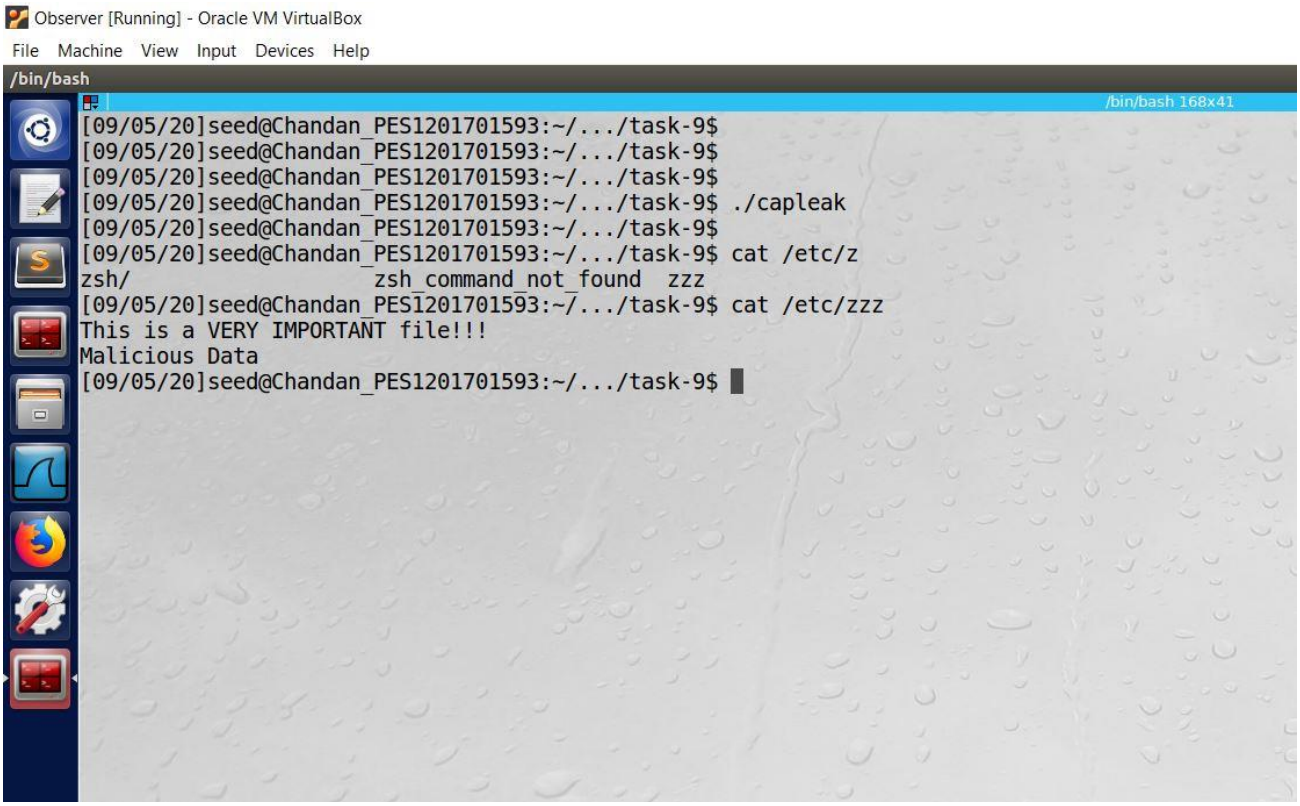


2. We create a file named zzz at /etc/ with its owner being root assuming it to be a very important file.



3. We compile and run the capleak program as shown below.





After running the capleak program we observe that the content of /etc/zzz file is modified as seen in the above screenshot. This happened because even if the privileges were dropped the file

was still open with all the privileges and after forking the attacker can easily modify the contents of the file. Thus closing the file descriptor was essential to prevent the capability leak.

THANK YOU