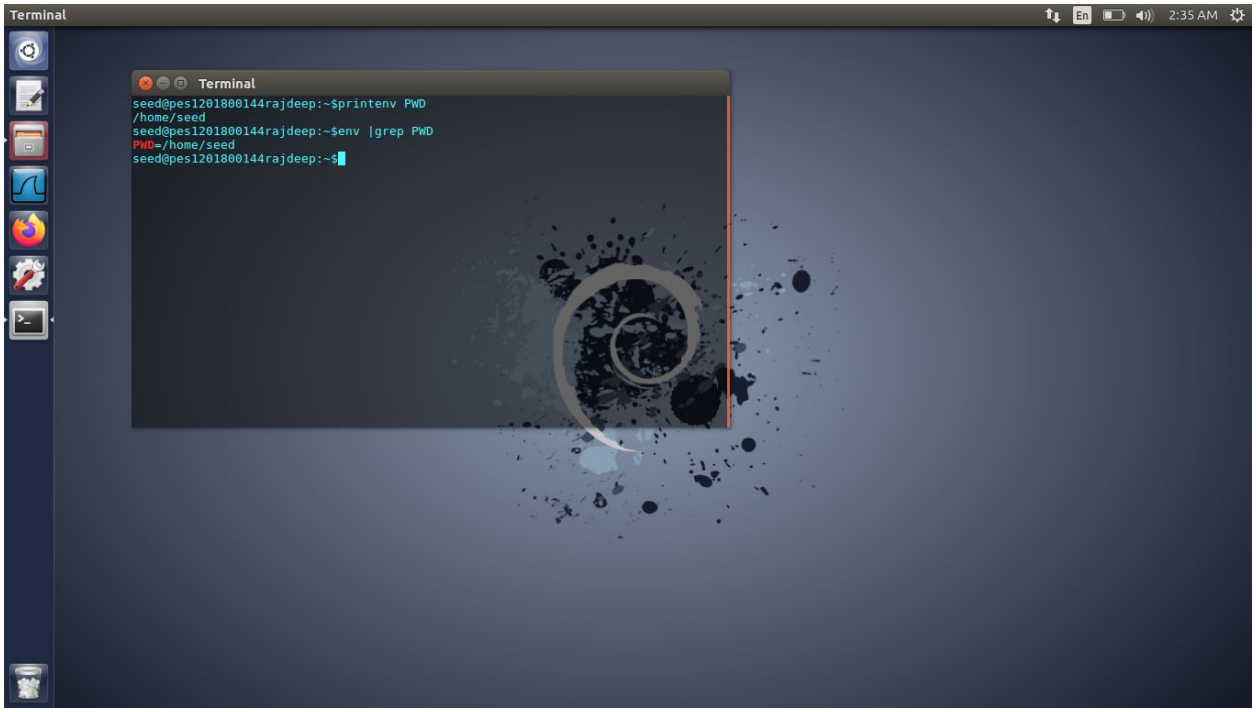
INFORMATION SECURITY LABORATORY
WEEK 1

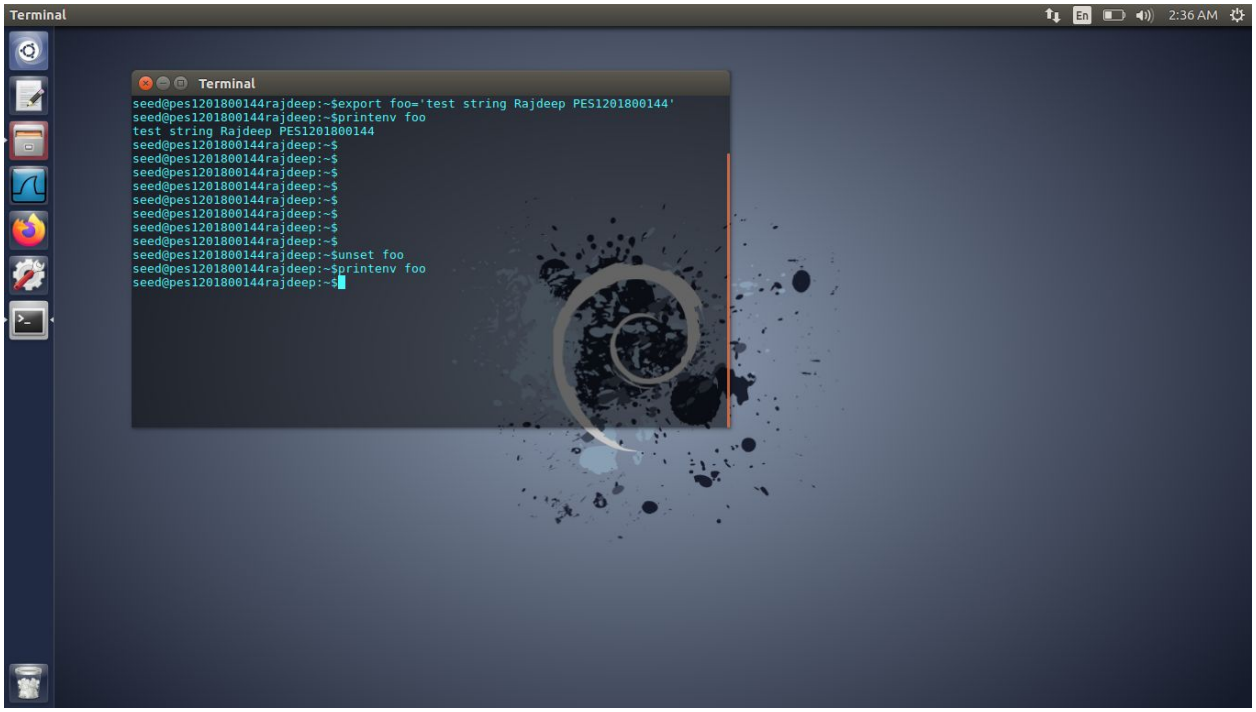
BY: RAJDEEP SENGUPTA
SRN: PES1201800144
SECTION: C

Note: Please find the terminal username as my SRN followed by my name ‘seed@pes1201800144rajdeep’. Also find the screenshots followed by observations for each task.

TASK 1: Manipulating environment variables

Screenshots Task 1:





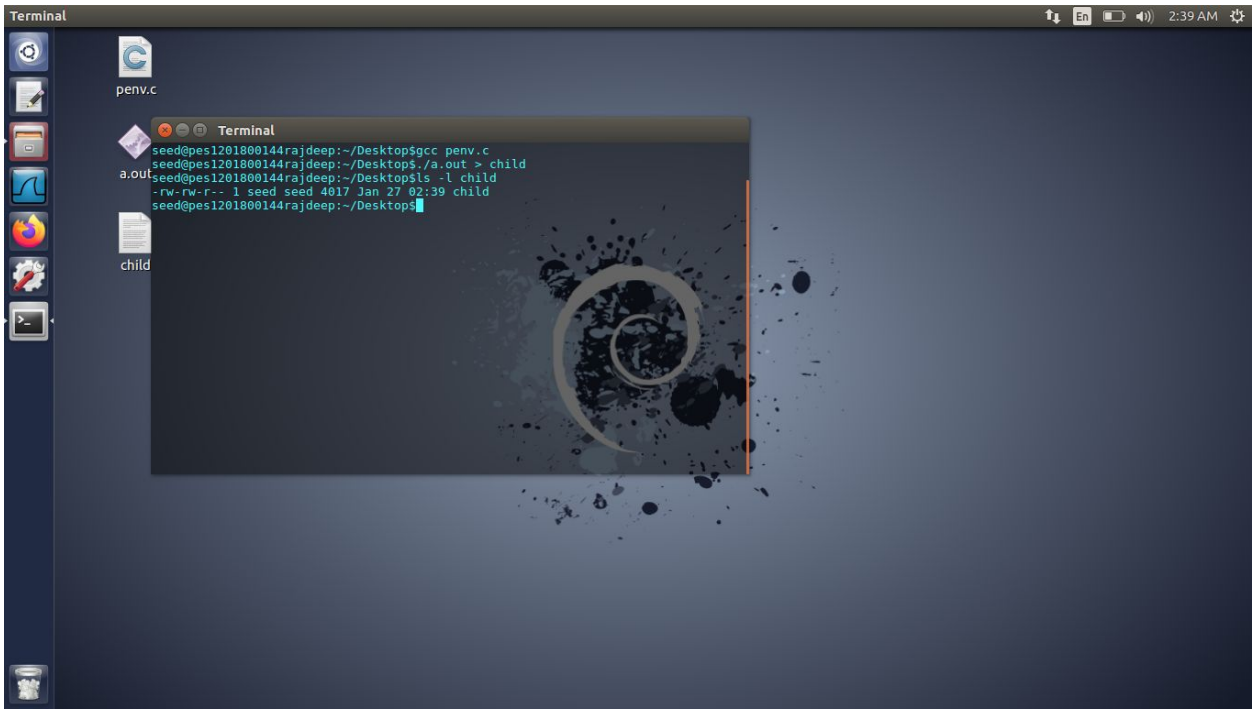
Observation Task 1:

printenv prints out the value of the environment variable
export command is used to set environment variables
unset command is used to unset environment variables
Using export and unset commands makes temporary environment variables which are unset when the system is rebooted. Whereas if the changes are directly made to the .bashrc file, then the changes stay permanently even after reboot.

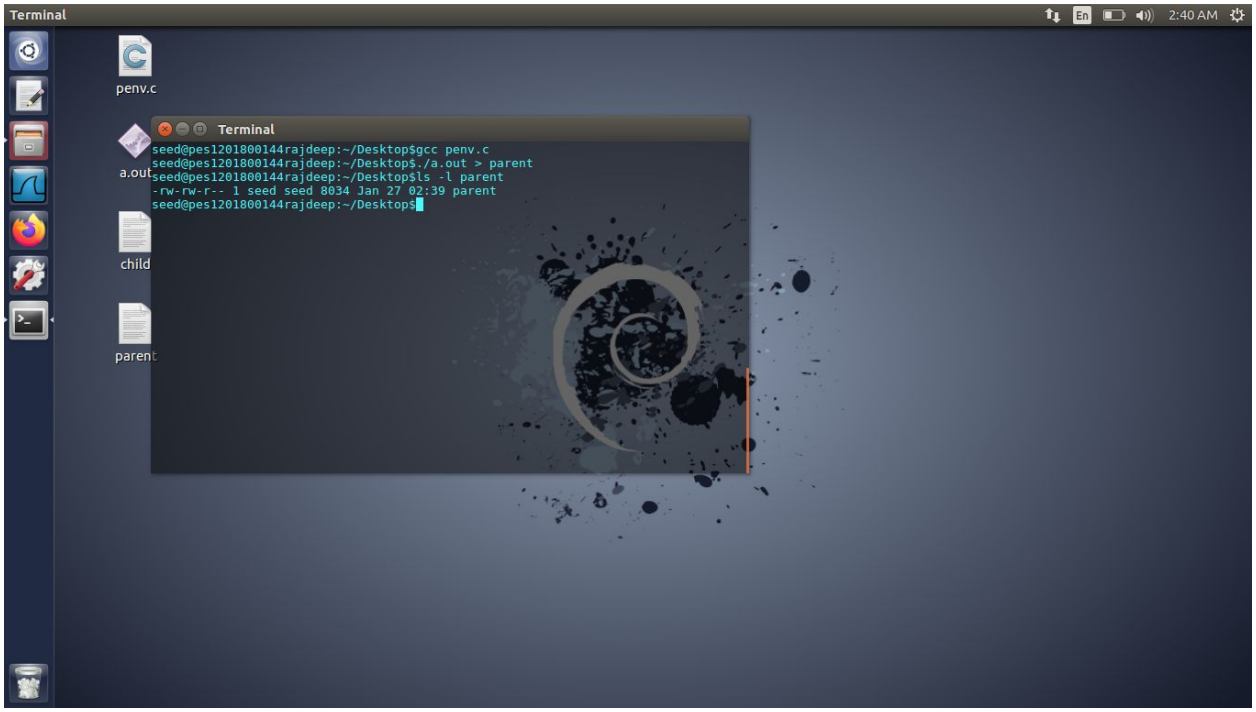
==========

TASK 2: Inheriting environment variables from parents

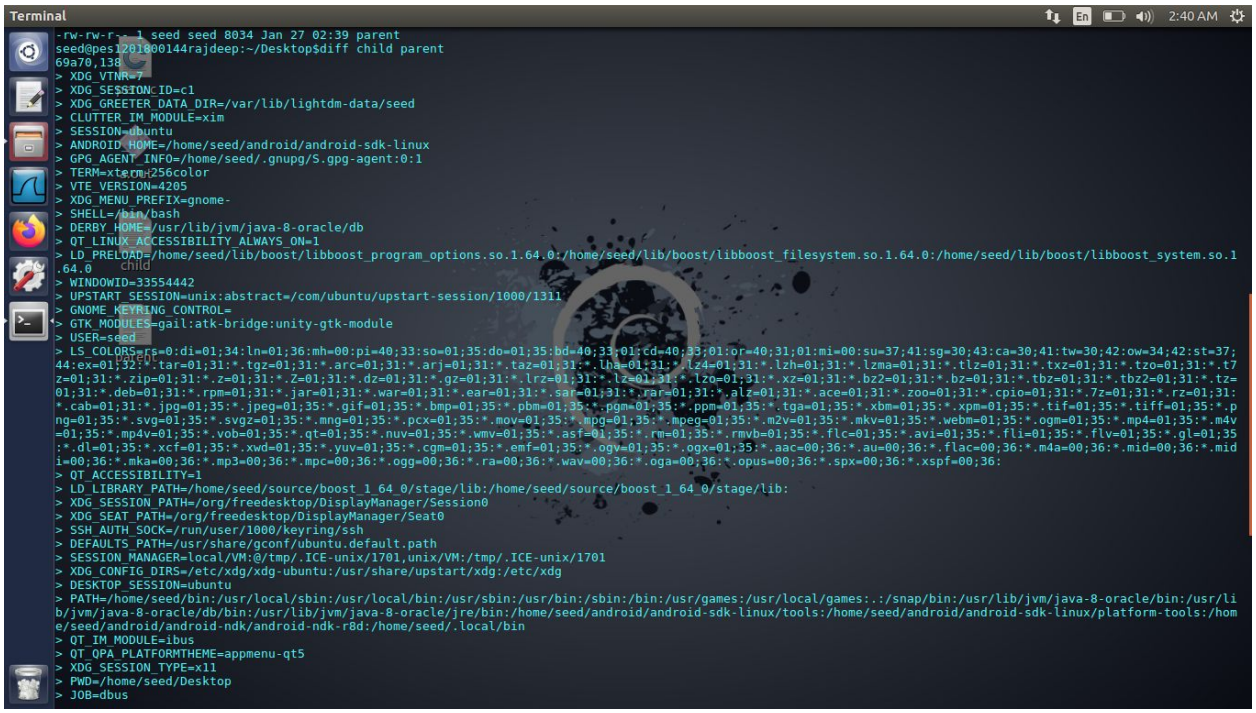
Screenshots Task 2:



Permissions of child output



Permissions of parent output



Output of ‘diff child parent’ command

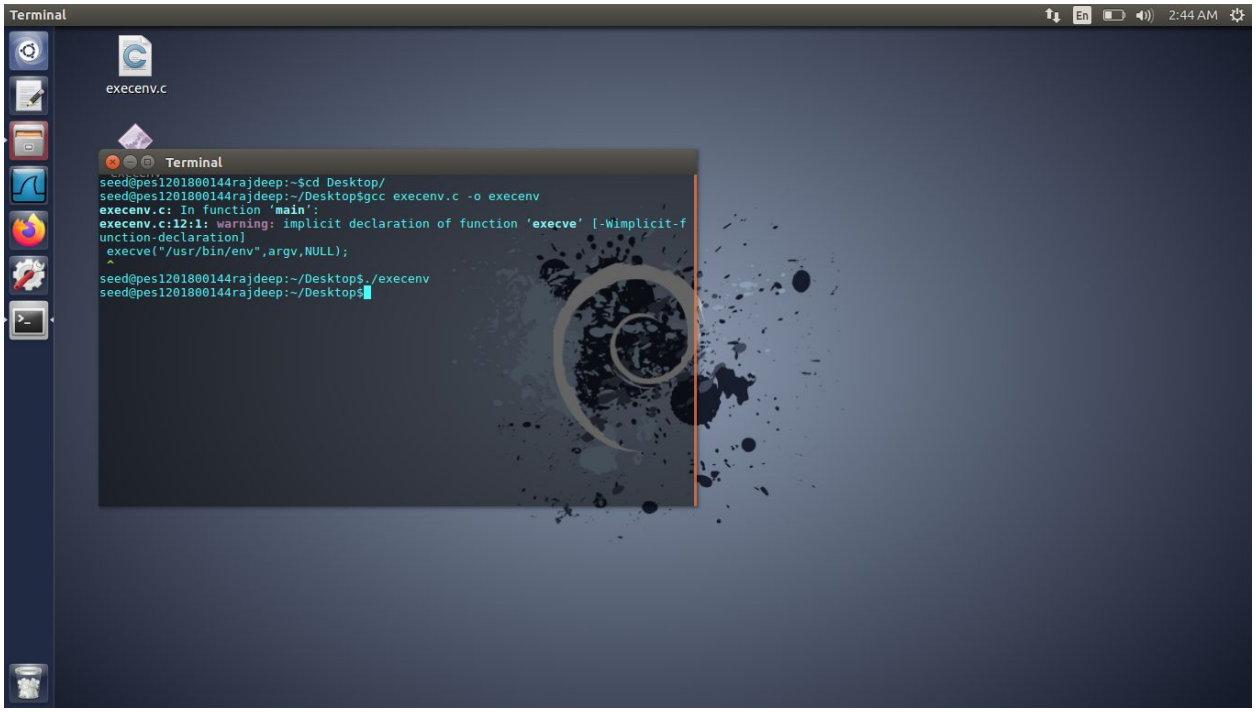
Observation Task 2:

There will not be any difference between parent and child programs

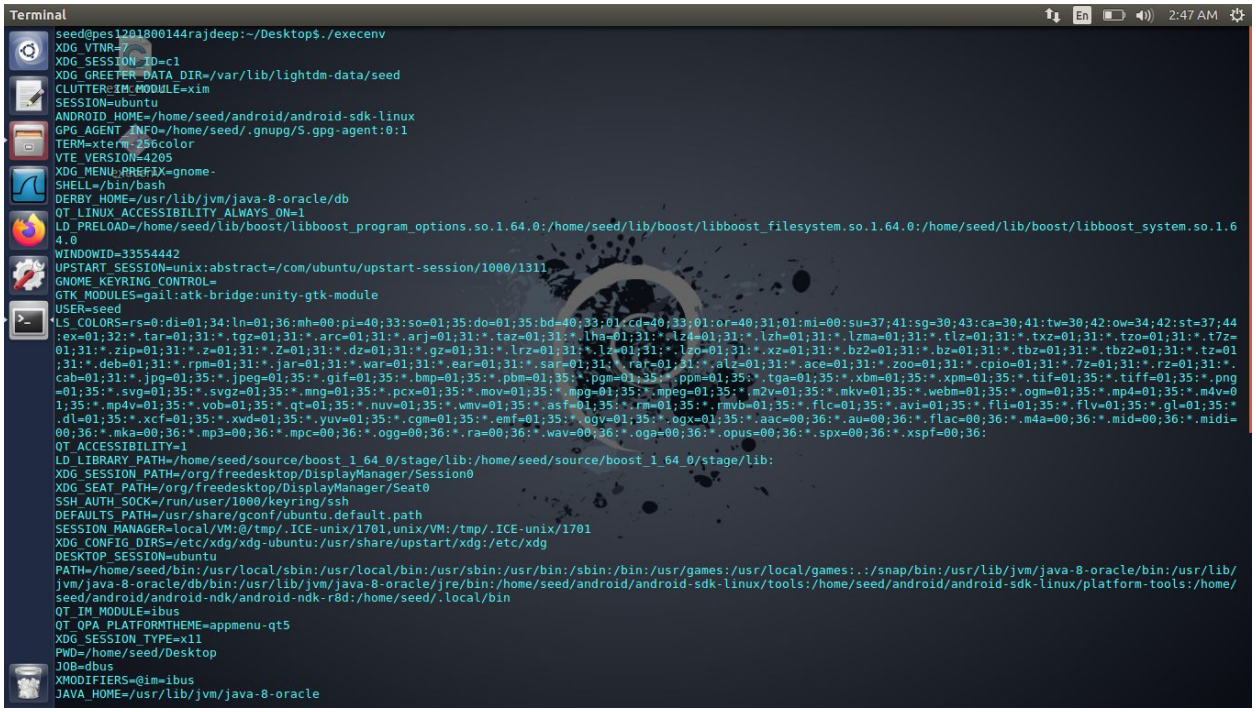
All the environment variables of the parent are inherited to the child for any process.

==========

TASK 3: Environment variables and execve()



Output on executing code with ‘execve(“/usr/bin/env”, arg, NULL)’



Output on executing code with ‘execve(“/usr/bin/env”, arg, environ)’

Observation Task 3:

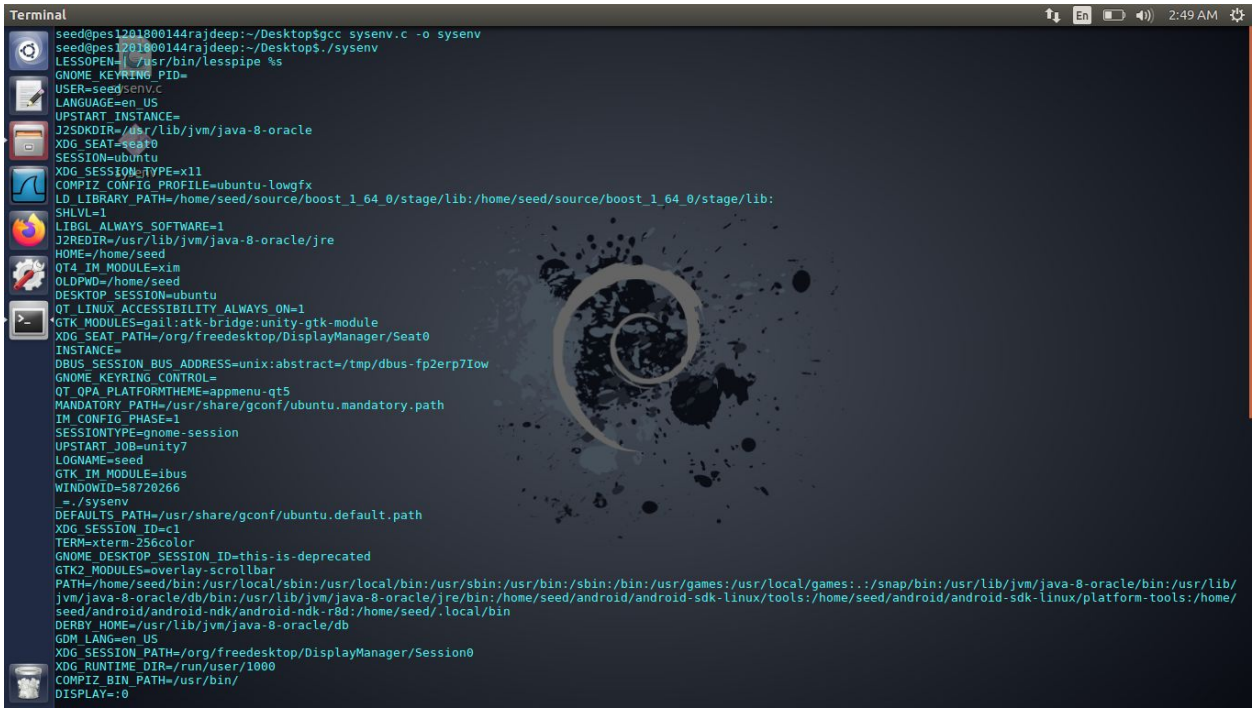
→ When execve(_, _, NULL) is called, environment variables are not inherited as seen in first screenshot

→ When execve(_, _, environ) is called, environment variables are passed to the new program and when the new program is executed, all the environment variables can be seen in the second screenshot

execve() replaces the code and data of the current process by code and data loaded from an executable file.

==========

TASK 4: Environment variables and system()
Screenshots Task 4:



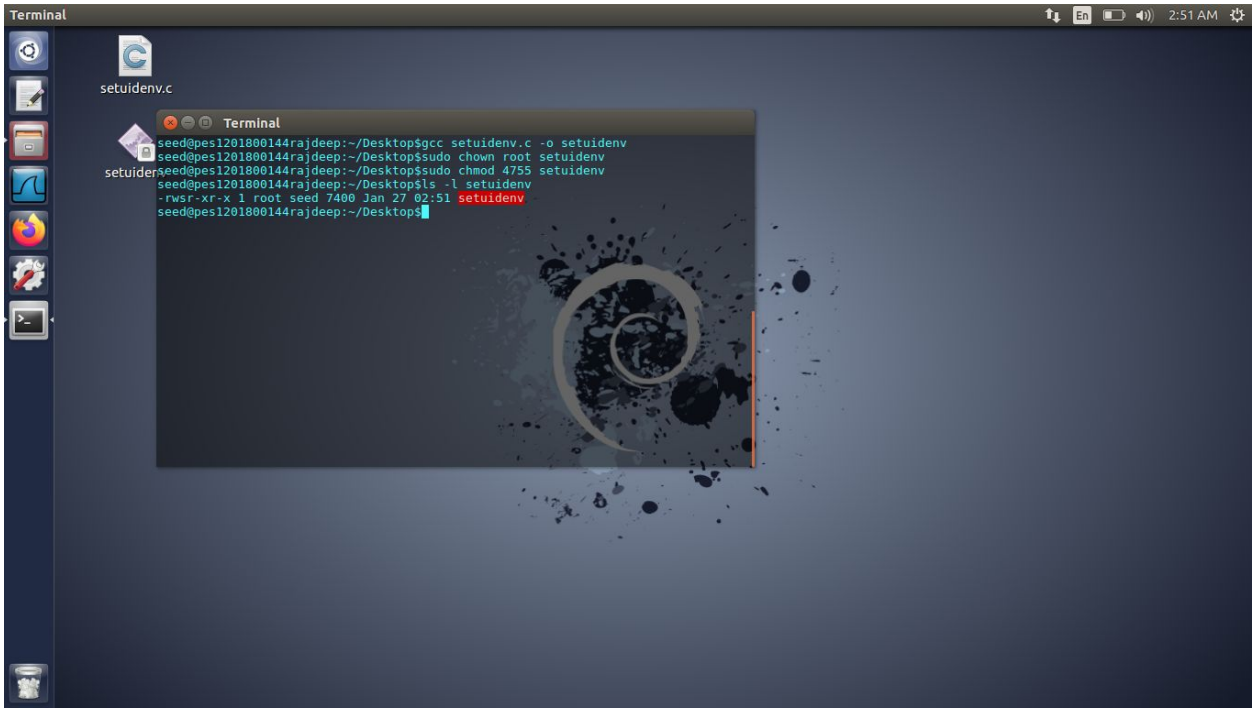
Execution using system() call. So the environment variables of calling process is passed to new program /bin/sh

Observation Task 4:

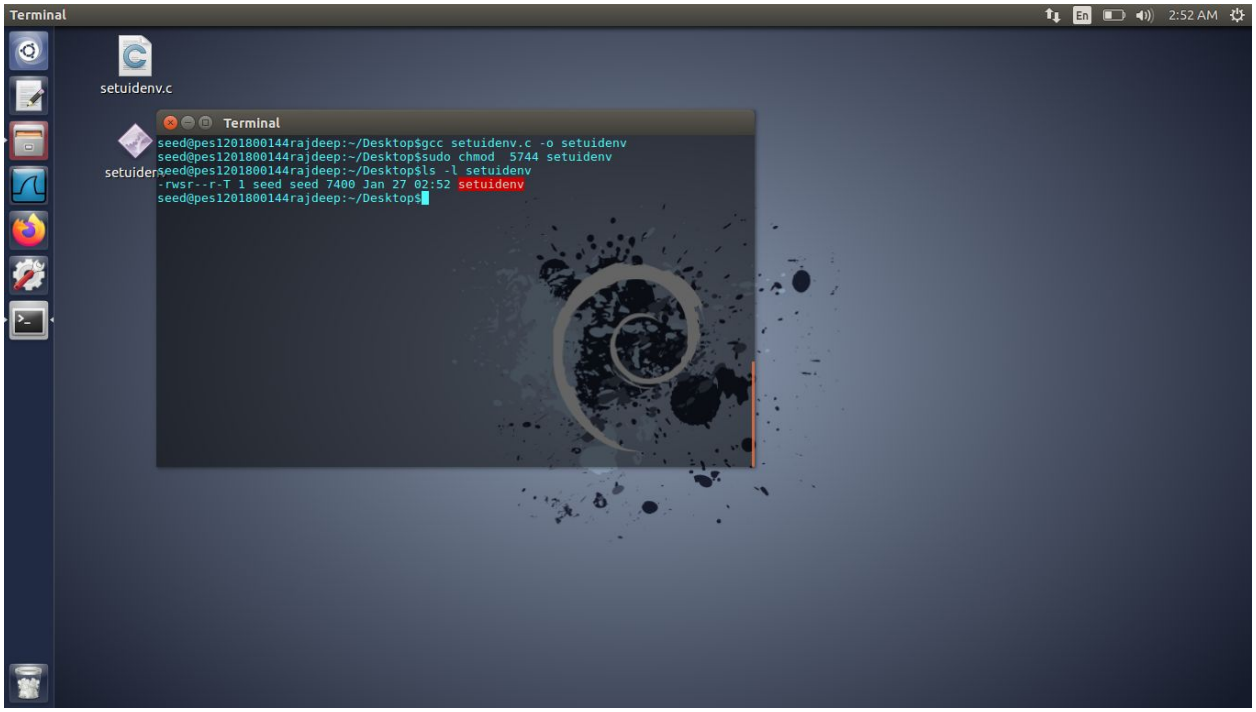
system() call inherits the environment variables as we can see in the screenshot. system() calls execl() which further calls execve(). Hence all the environment variables are passed to the new program like execve() call.

==========

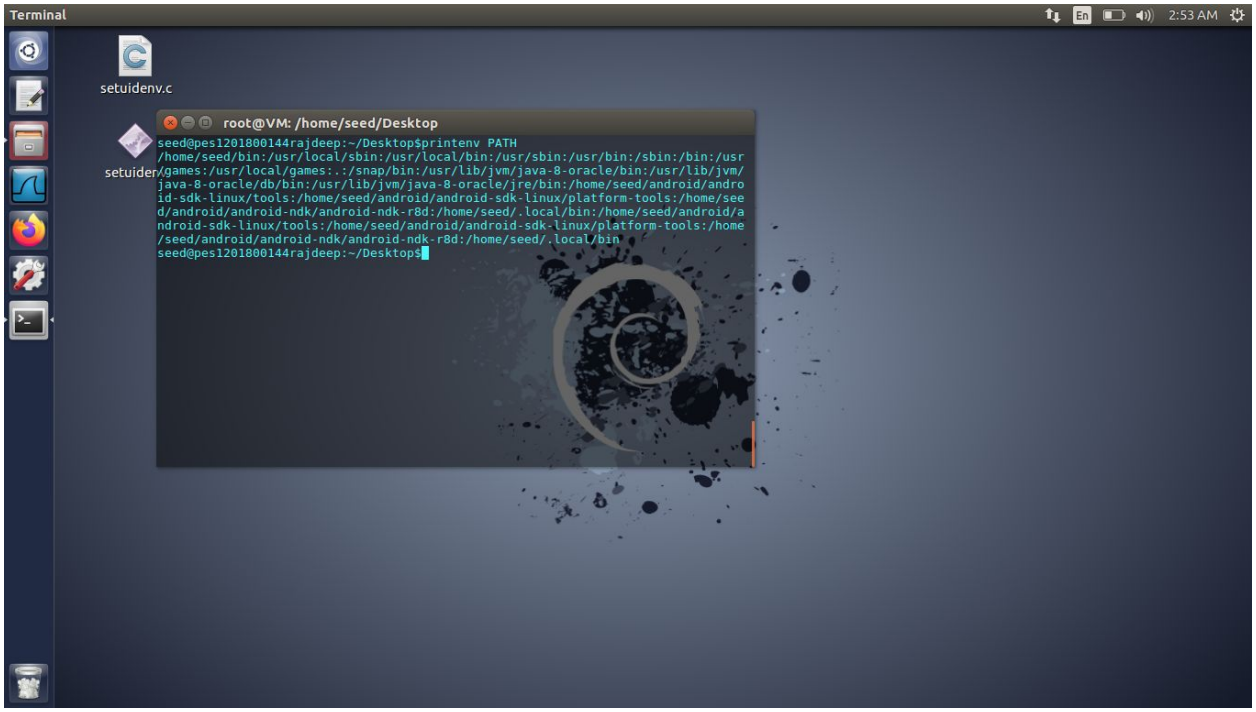
TASK 5: Environment variable and Set-UID programs



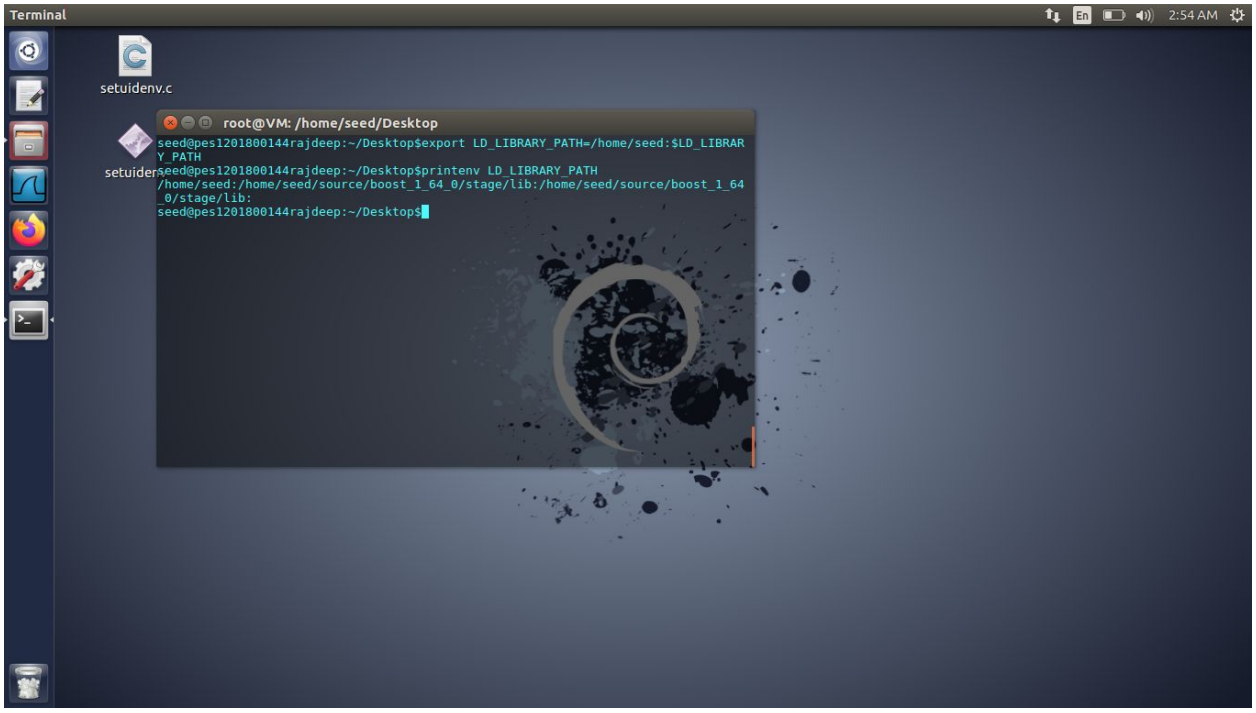
Making the code owner as root using chmod and chown



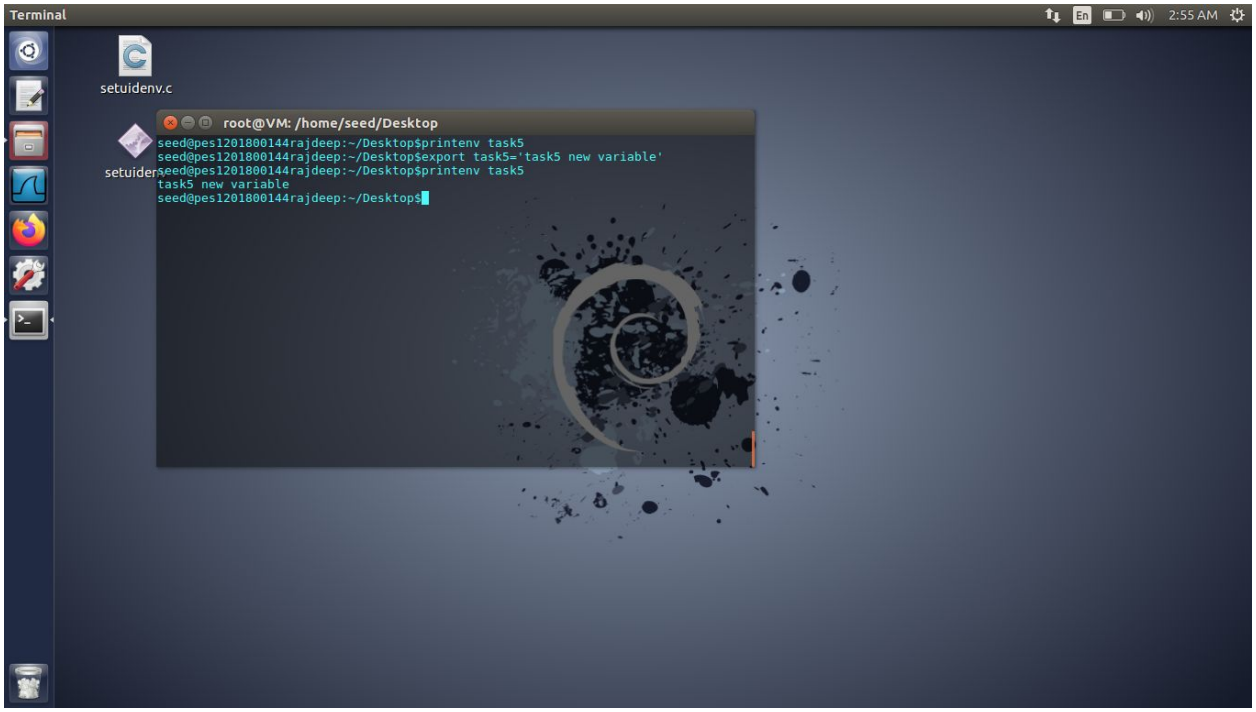
Changing owner to root and making it Set-UID program using chmod 5744



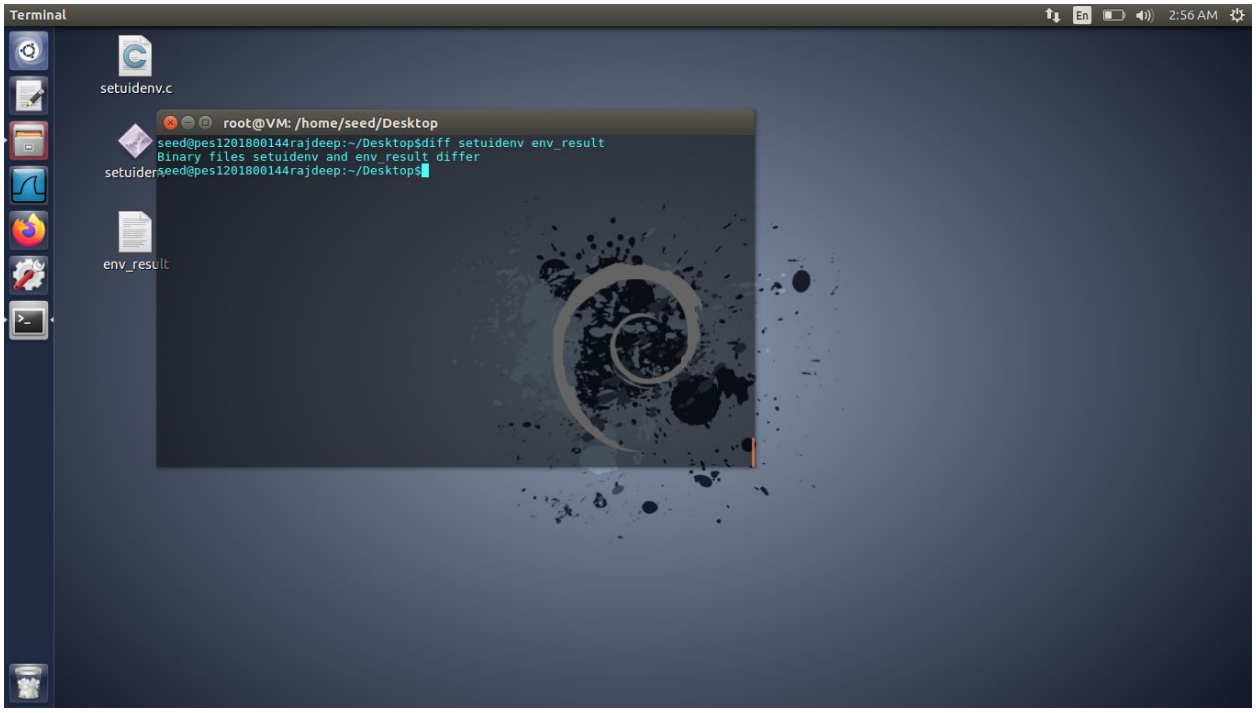
Output of ‘printenv PATH’



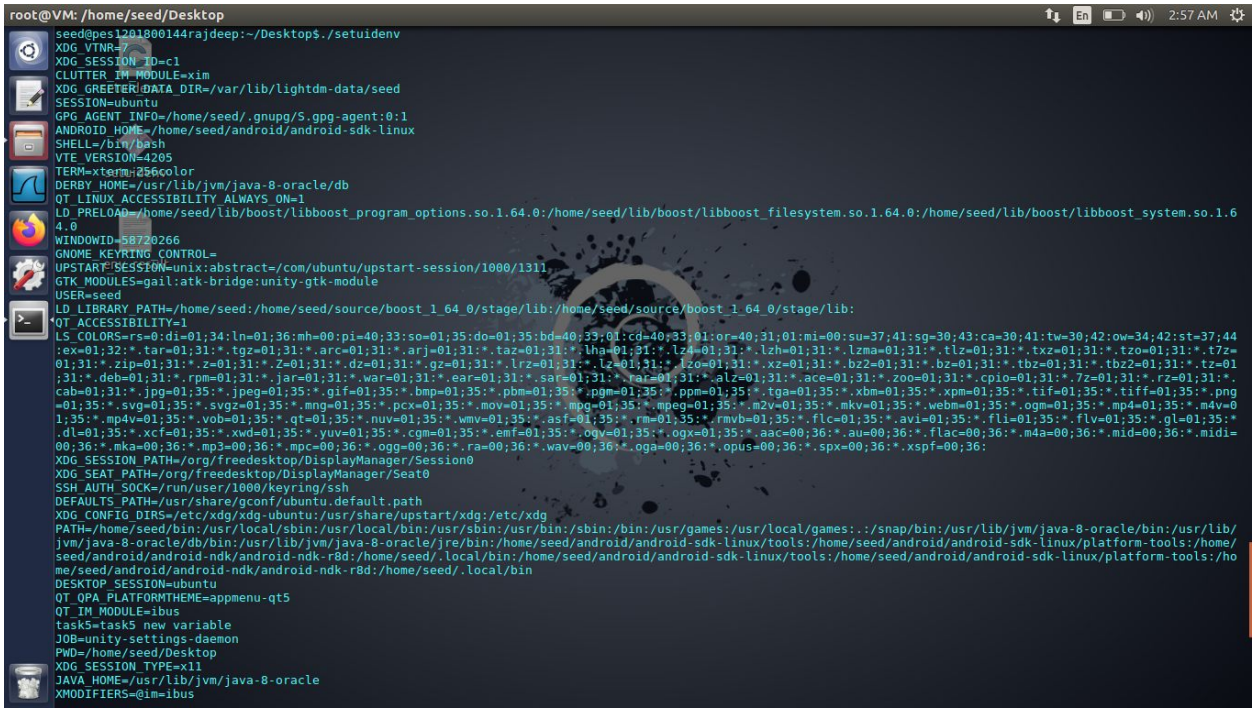
Setting LD_LIBRARY_PATH to /home/seed:$LD_LIBRARY_PATH



Setting task5 as new environment variable



Difference between ‘setuidenv’ and ‘env_result’



Executing the program after doing steps 2 and 3

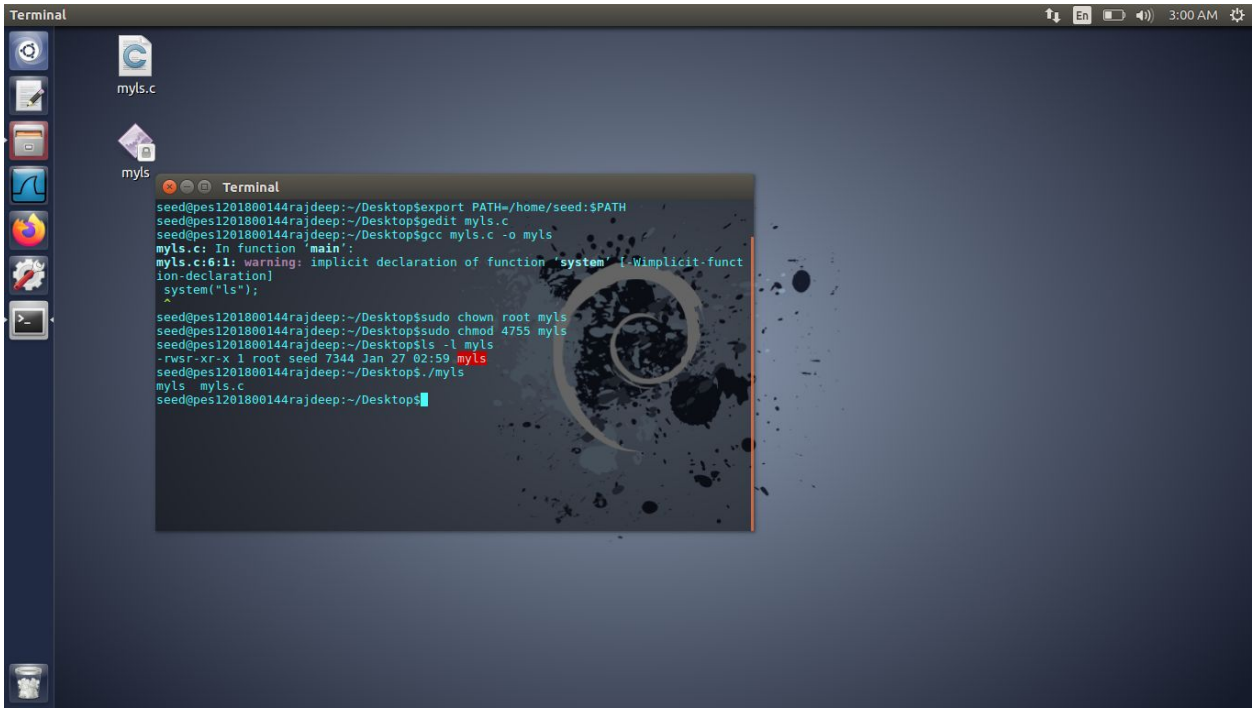
Observation Task 5:

The set-UID program does not inherit all environment variables like LD_LIBRARY_PATH. So we have to export LD_LIBRARY_PATH in Step 3 of this task. **This is because RUID(real user ID) and EUID(effective user ID) are different.**

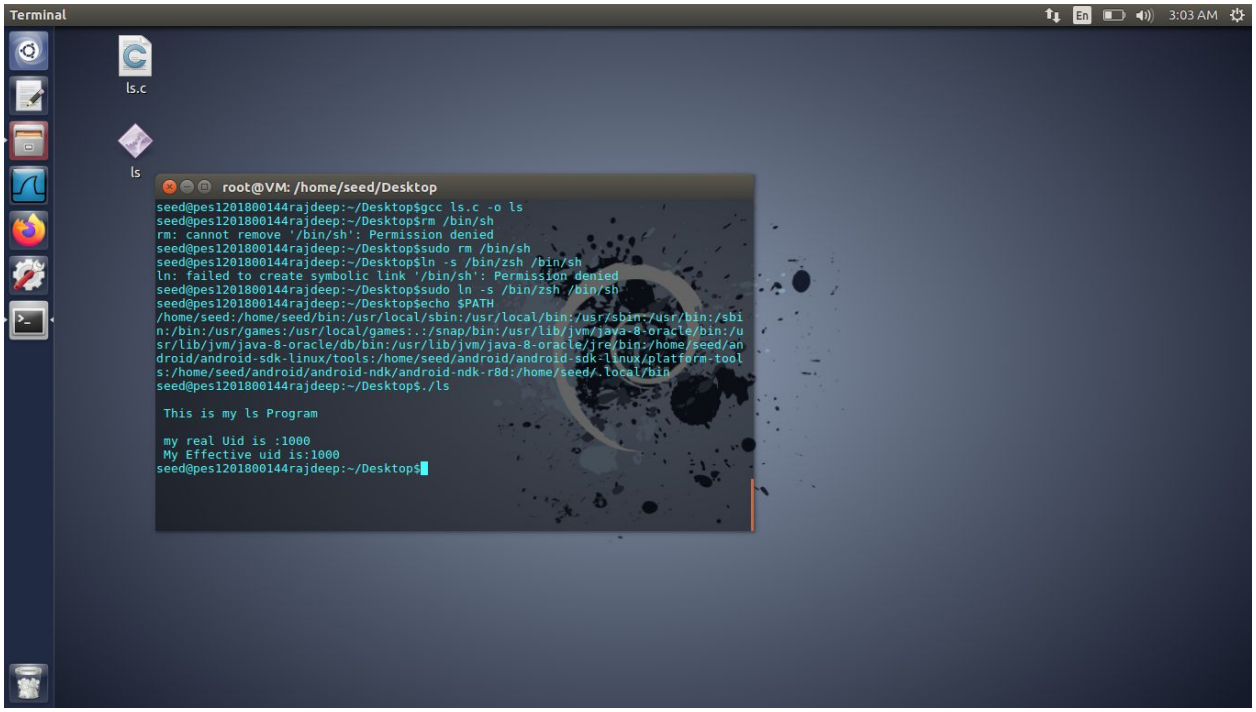
==========

TASK 6: The PATH environment variable and Set-UID programs

Screenshots Task 6:



Changing PATH environment variable, compiling the code and making the owner as root and changing permissions using chown and chmod



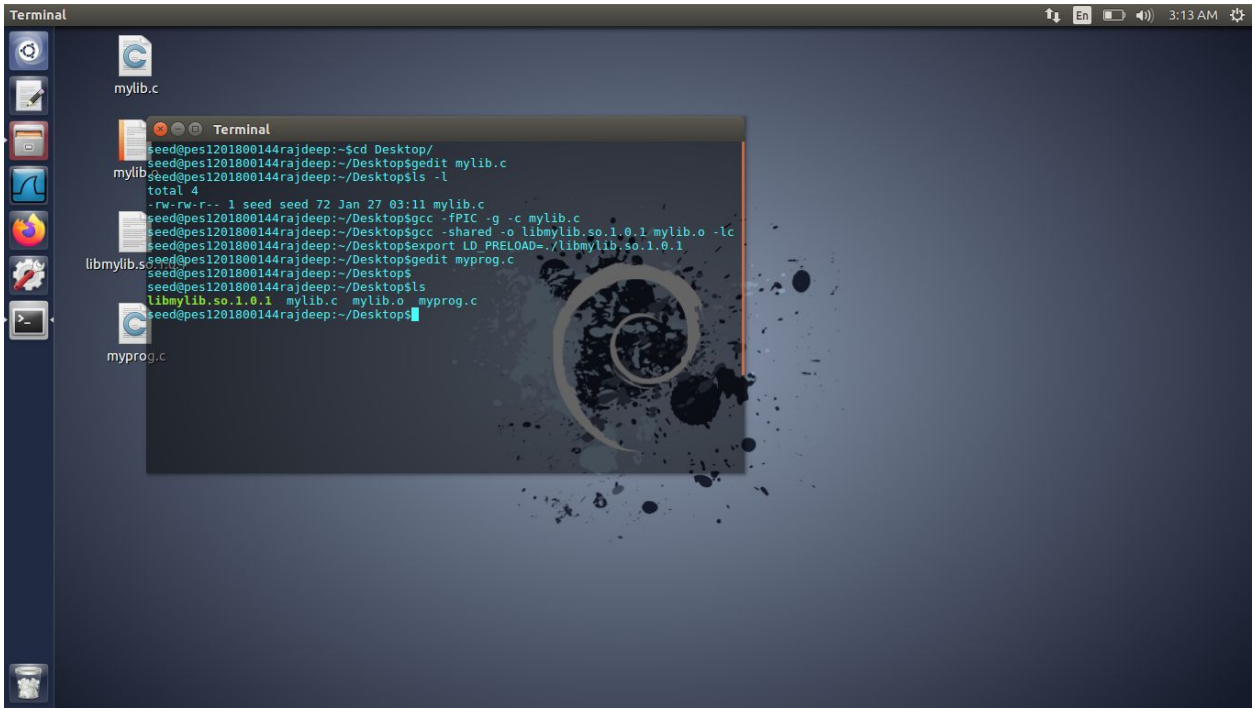
Removing /bin/sh and then using ‘ln -s’ for symbolic linking /bin/zsh to /bin/sh. Then exporting environment variable PATH to the current working directory and executing the program.

Observation Task 6:

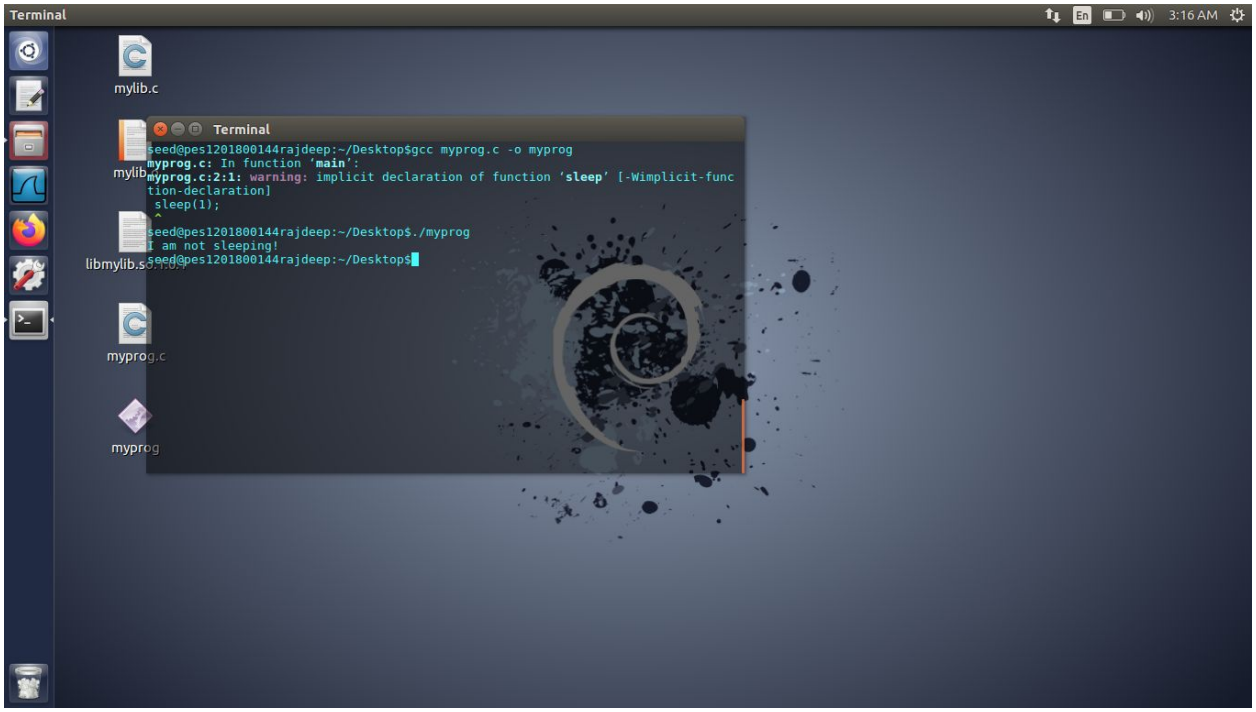
system() call with Set-UID is dangerous in terms of security as we can run our own program in place of the system's function call. For eg. in this example, our ‘ls’ program is executing instead of bash shell’s ‘ls’ command.

==========

TASK 7: The LD_PRELOAD variable and Set-UID programs
Screenshots Task 7:

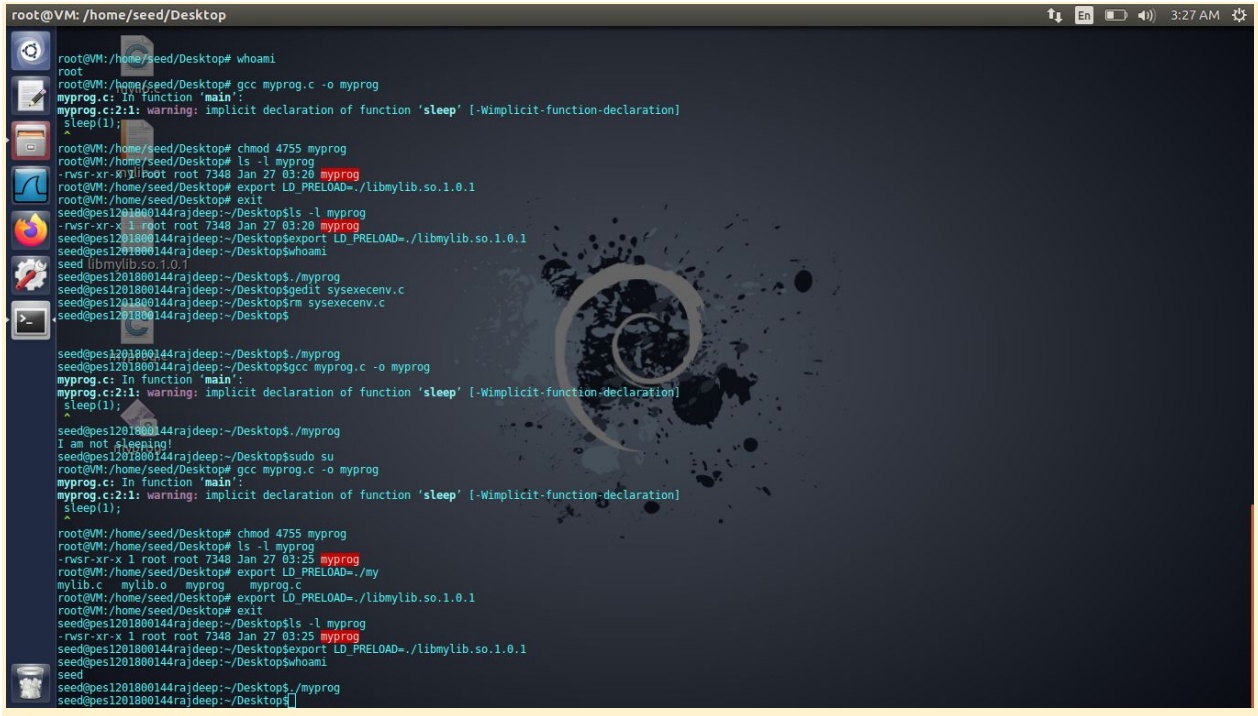


Setting LD_PRELOAD to libmylib.so.1.0.1 executable file.



Running the program in user mode.

The problem here is that the ‘sleep’ command should make the system sleep instead of executing the function with the same name.



Exporting LD_PRELOAD to libmylib.so.1.0.1 in root mode and then running program in user mode which results in the desired functioning of the program (calling sleep function makes machine sleep instead of executing user defined sleep function)

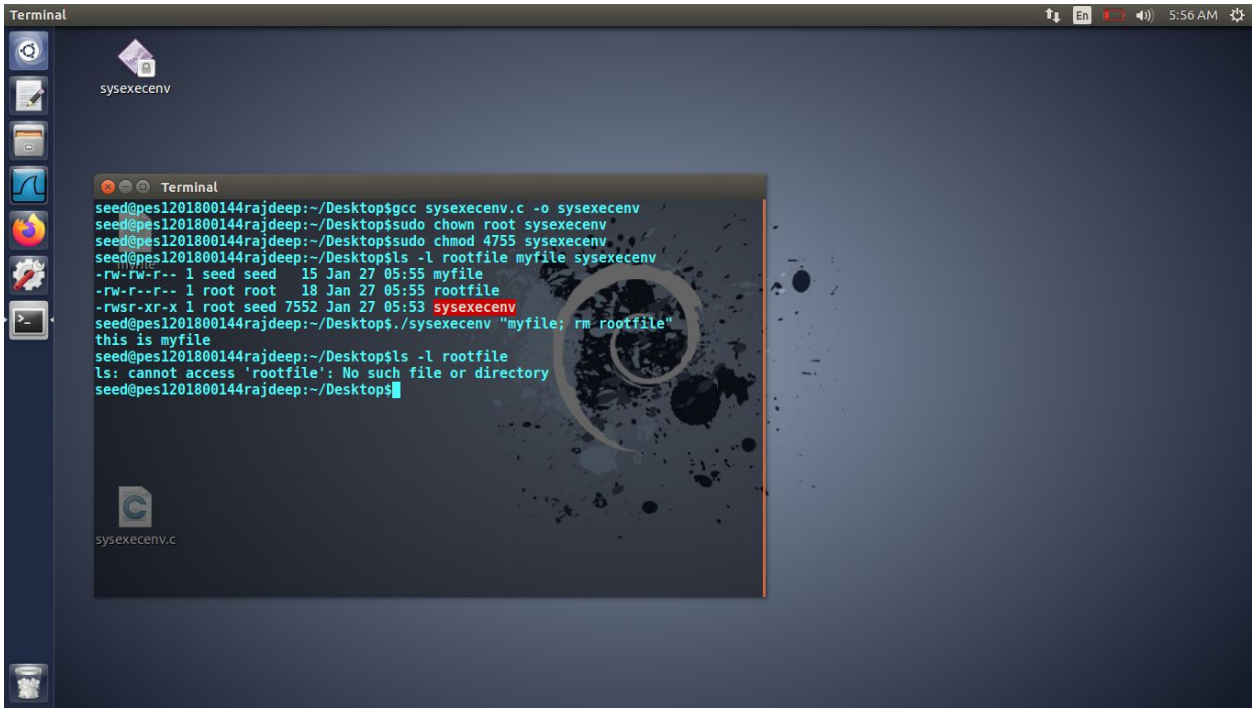
Observation Task 7:

LD_PRELOAD is inherited from the program when the RUID(real user ID) is the same as the EUID(effective user ID) unlike in Task 5. When RUID and EUID are different, our program(sleep function defined is program) is executed. Further when RUID and EUID become the same after compiling in root, the system's sleep function is called which makes the system sleep.

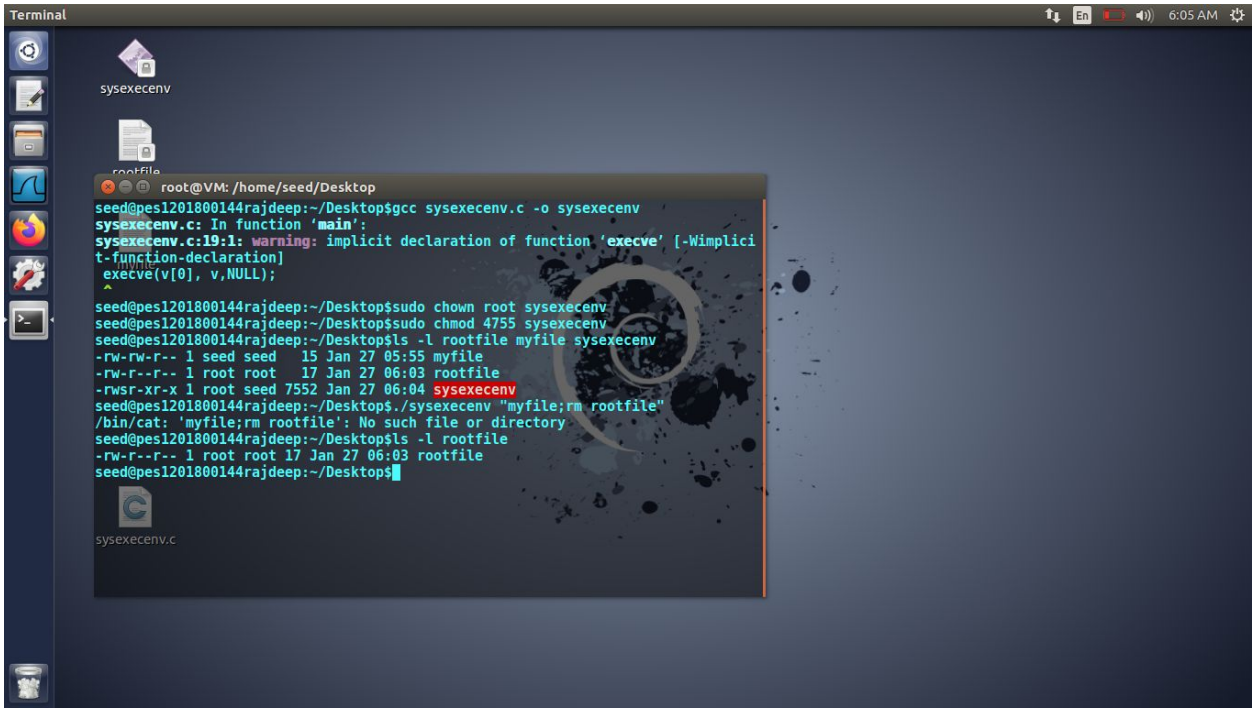
==========

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

Screenshots Task 8:



Compiling the code and making root as owner and changing permissions using chown and chmod. Using ‘rm’ command in user mode removes the root permitted file. This is because of using system() call.



Doing the same above but this time using execve() instead of system() solves the problem and doesn't allow non-root users to delete root permitted files.

Observation Task 8:

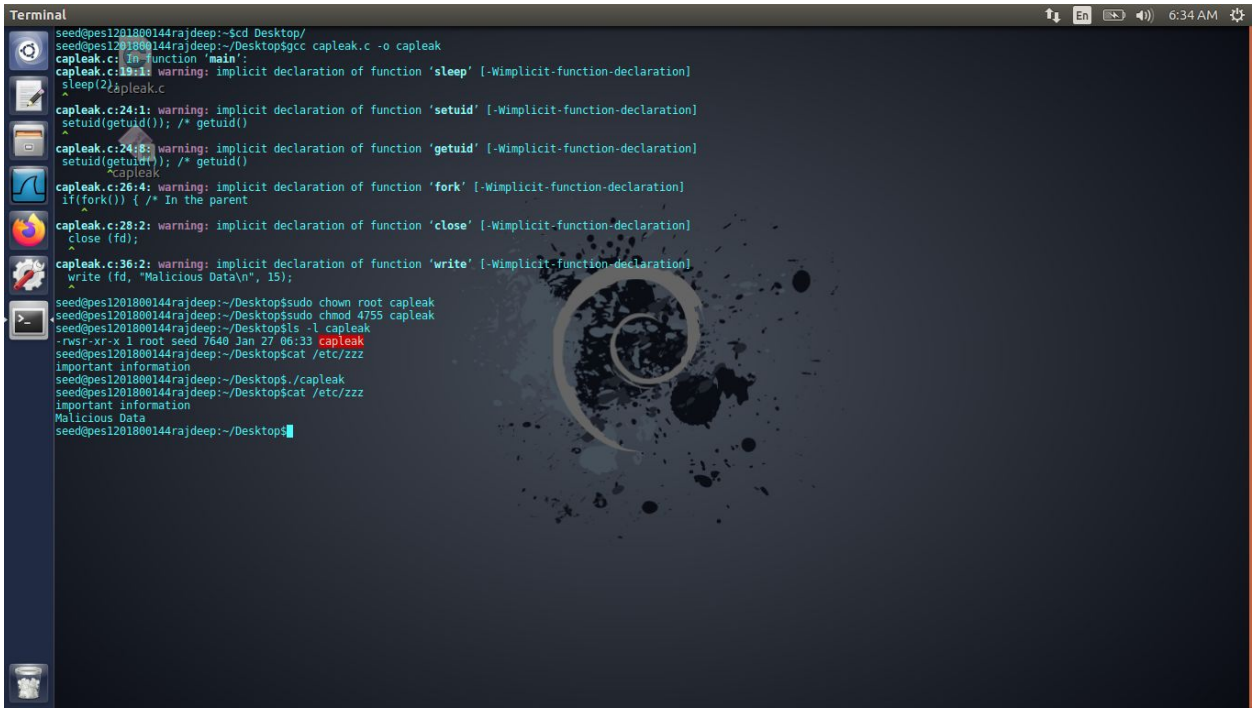
→ system() allows Bob to modify the files as superuser
→ execve() does not let Bob to modify the files as superuser

This is because system() function call invokes a shell which then executes the command whereas execve() directly executes function call.

In this case, system() call invokes a new shell with root permissions(root shell) which can read, modify and delete files which makes it dangerous. Whereas in case of execve() call, the root shell is never invoked so permission to modify or delete is not given to Bob.

==========

TASK 9: Capability Leaking



When a process is permitted to execute privileged commands, the child process can execute malicious code. This capability leaking is done when the program is made owned by root and root permissions are given using chown and chmod.

Observation Task 9:

The child process executes the malicious code whereas the privileges were only meant for the parent process. This is a massive vulnerability which can be exploited.

==========