INFORMATION SECURITY LAB

LAB 1: Environment Variable and Set-UID Program

Name: Ankitha P

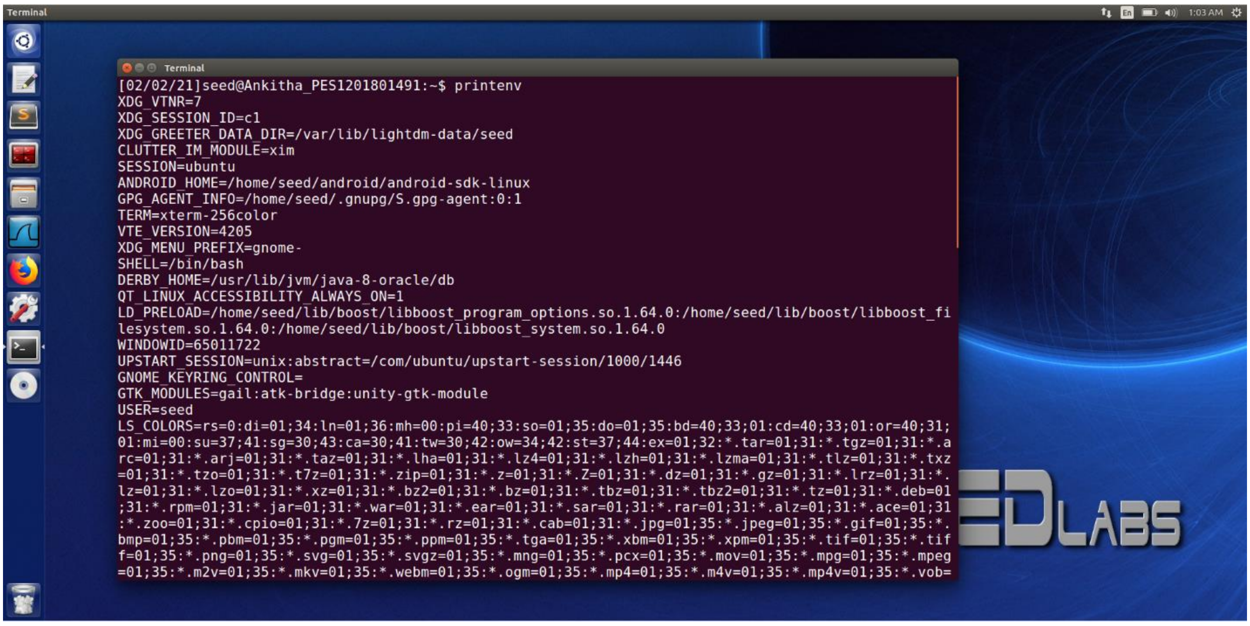
Class: 6 ‘D’
Date : 02/02/2021

OBJECTIVE

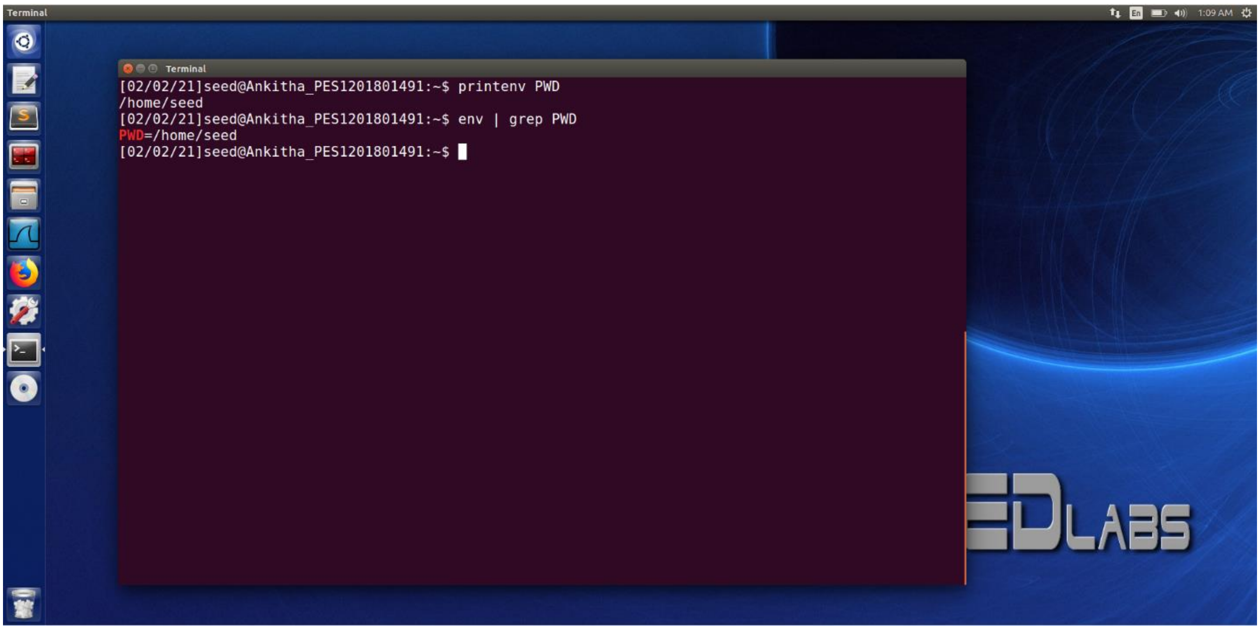
a) Understanding how environment variables work
b) Understanding how environment variables are propagated from parent process to child
c) Understanding how environment variables affect system/program behavior

EXECUTION

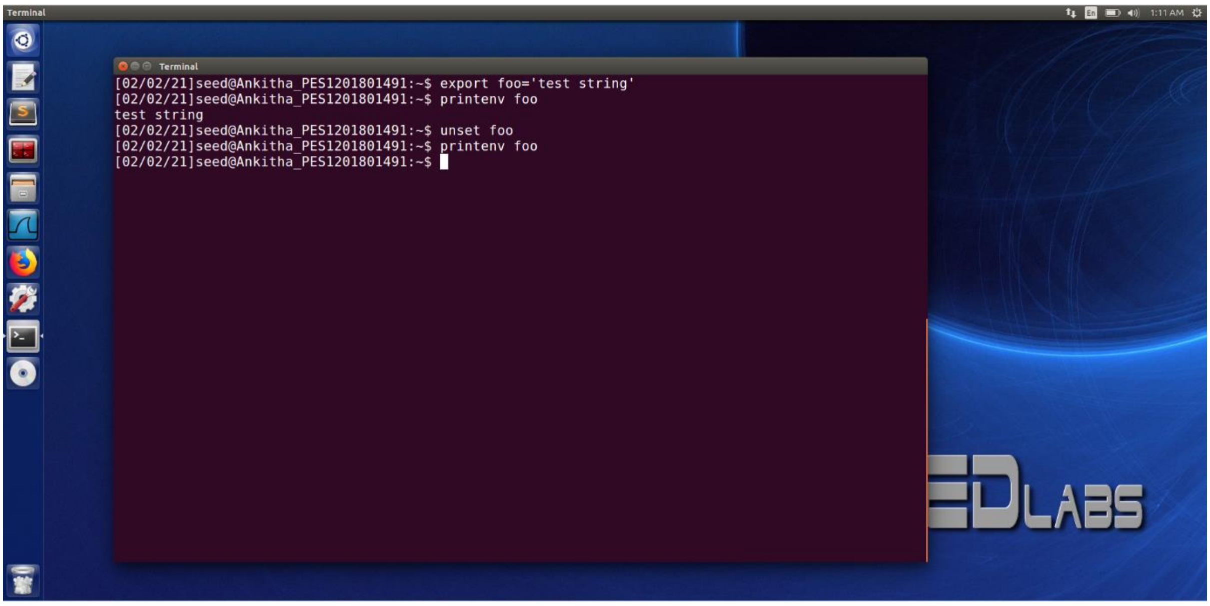
Task 1: Manipulating environment variables



The printenv or env command is used to print out the environment variables.

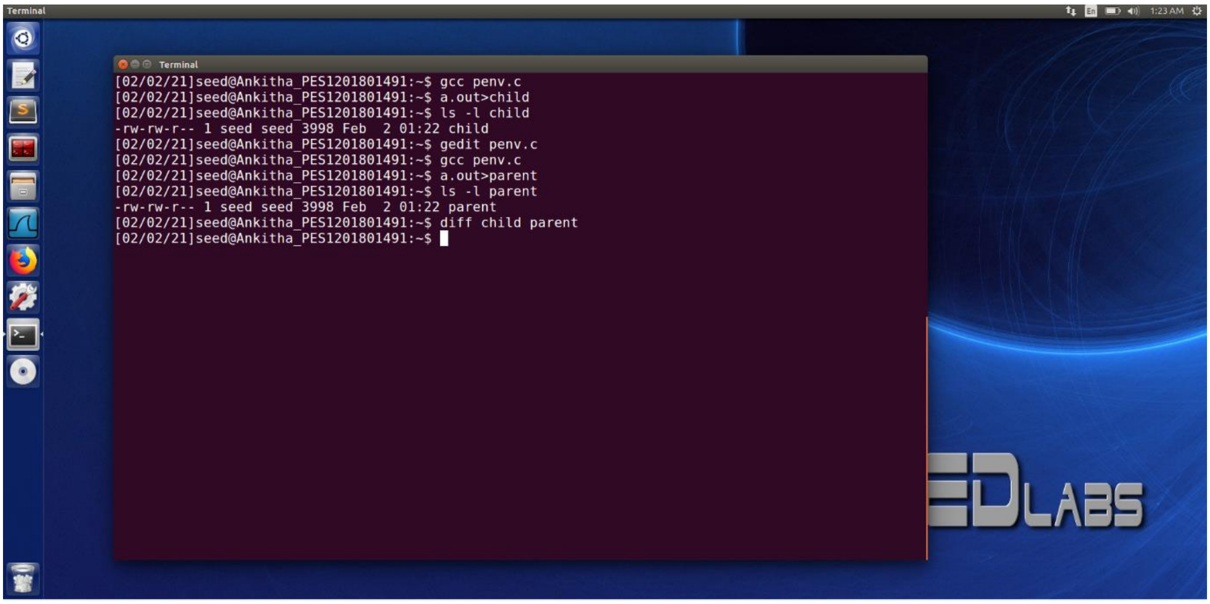


pwd stands for Print Working Directory and stores the present working directory or the directory which the user is currently using. Thus the command printenv pwd prints the current working directory on to the terminal.



export and unset command is used to set or unset the environment variables. Using export, the environment variable will be set for the current shell session. Here the variable ‘foo’ is set to value ‘test string’ with export command and is then unset by which the environment ‘foo’ no longer exists.

Task 2: Inheriting environment variables from parents



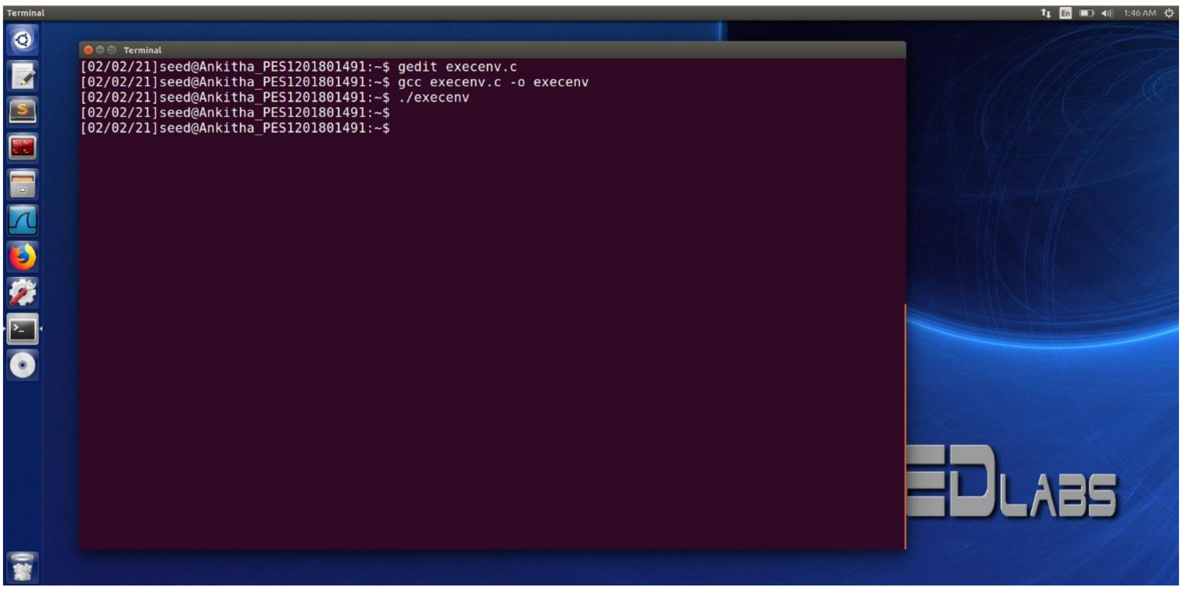
- Case 1: **child** –child process calls the printenv function (i.e pid==0)
- Case 2: **parent** – parent process calls the printenv function

The output of the `diff` command between the parent and child executable files shows that environment variables printed by the parent process and child process are the same and that the child process inherits all the environment variables of the parent of a fork call.

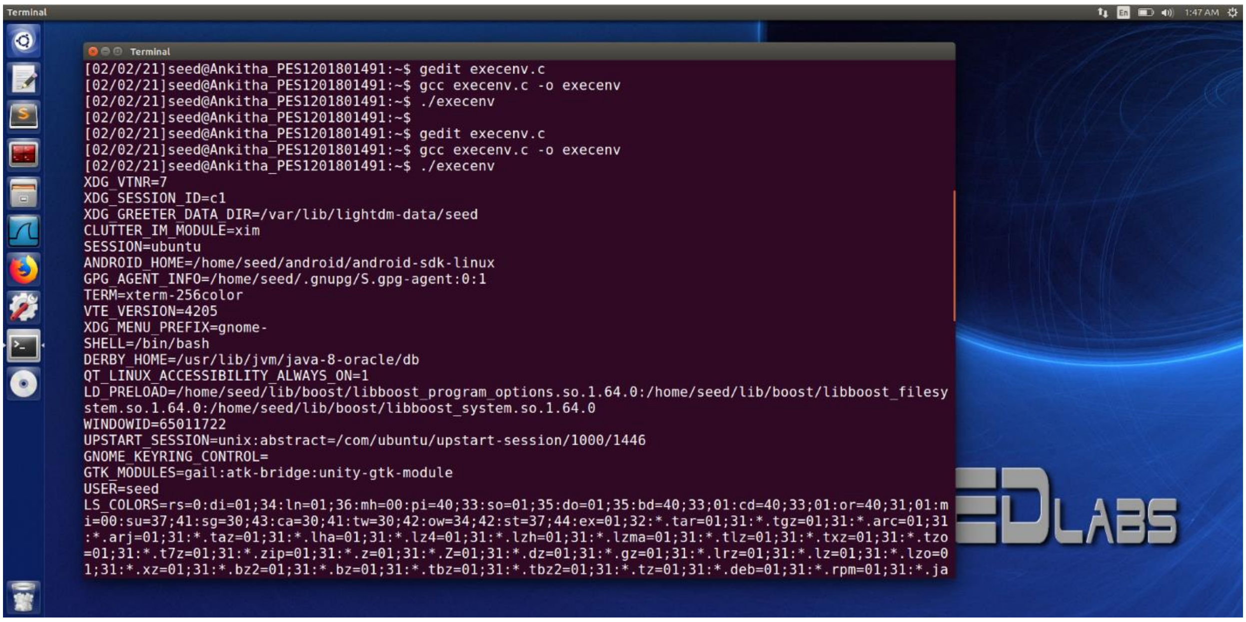
Task 3: Environment variables and execve()

execve() executes the program referred to by *pathname*. This causes the program that is currently being run by the calling process to be replaced with a new program. The program called /usr/bin/env prints the environment variables of the current process.

Case 1:



Case 2:

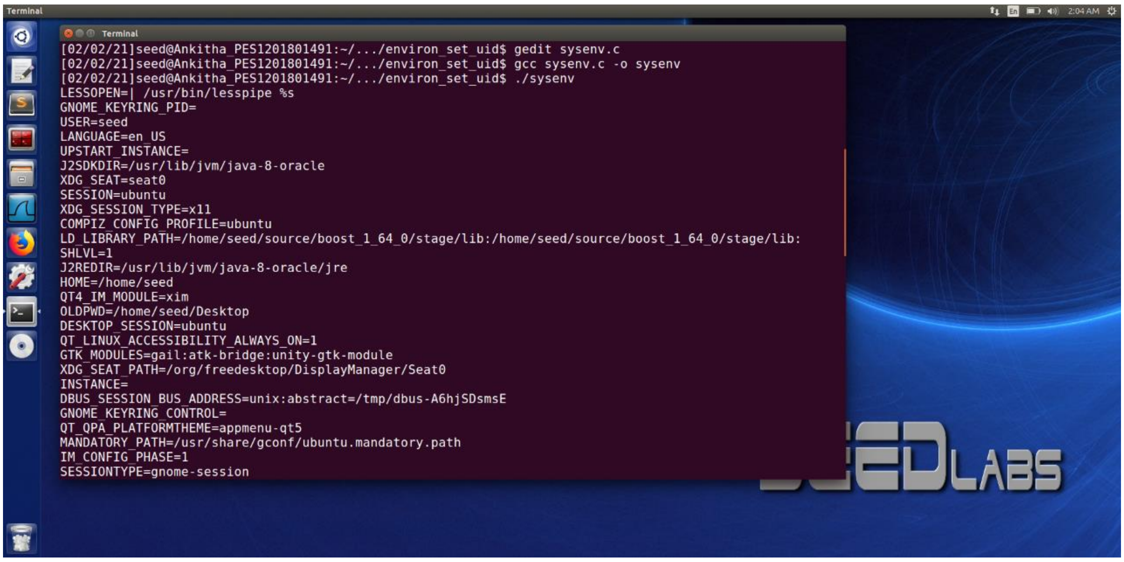


Case 1: The third argument passed to the execve function is NULL, thus no environment variables are associated with the process. Thus the usr/bin/env does not print any environment variables.

Case 2: We passed the environ variable as the third argument to execve, which contained all the environment variables of the current process, thus output of the program had all the environment variables.

In conclusion, the third argument of the execve() command gets the program its environment variables.

Task 4: Environment variables and system()



The code uses the system() function to execute the program /usr/bin/env which prints all the environment variables of the calling process. The system function uses execl() to execute /bin/sh.

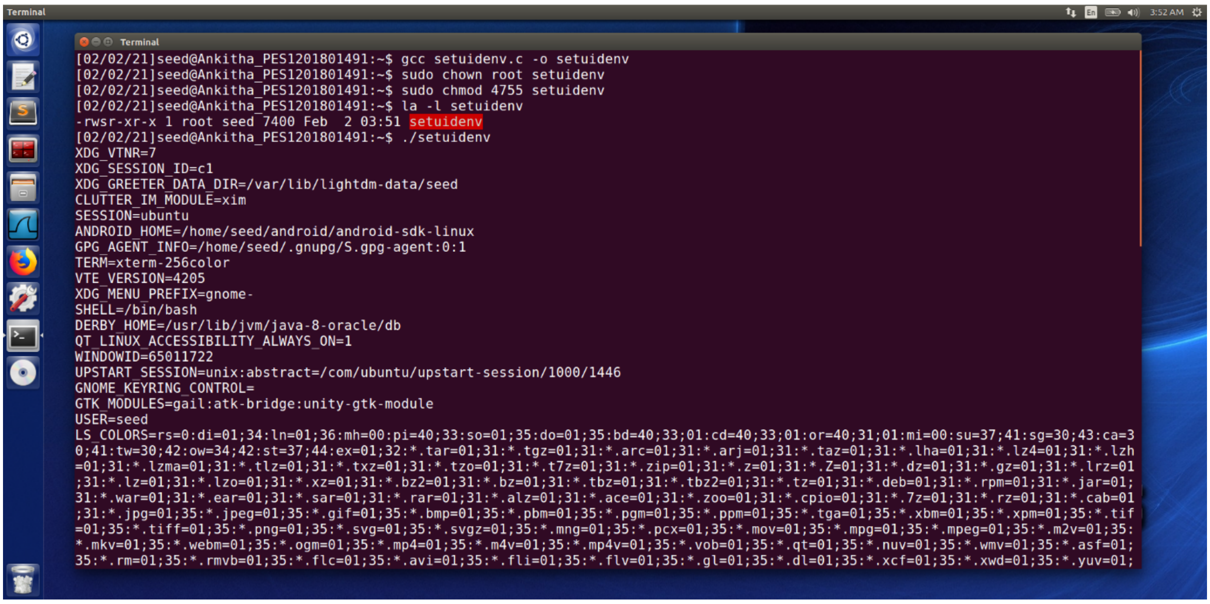
The program is compiled and executed and as seen, even though we don't explicitly send any environment variables in the program, the output shows the environment variable of the current process. This happens because the system function implicitly passes the environment variables to the called function /bin/sh. (i. system() function uses execl() to execute /bin/sh; excel() calls execve(), passing to it the environment variables array)

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. Set-UID allows us to do many interesting things, but it escalates the user's privilege when executed, making it quite risky.

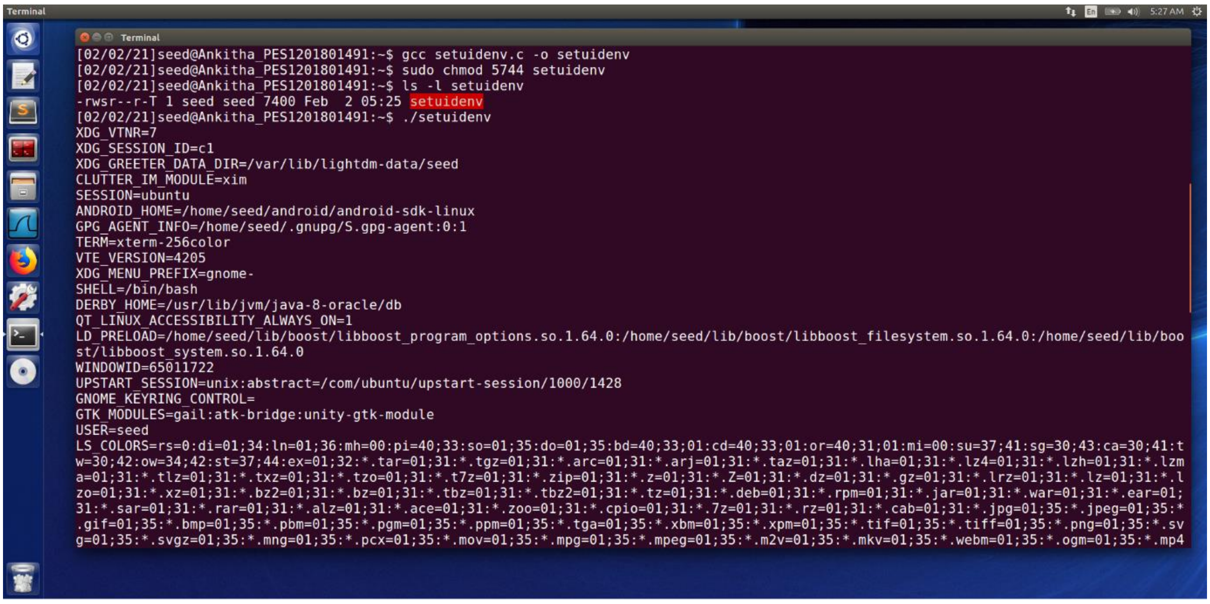
The program prints all the environment variables in the current process.

<u>CASE 1:</u>

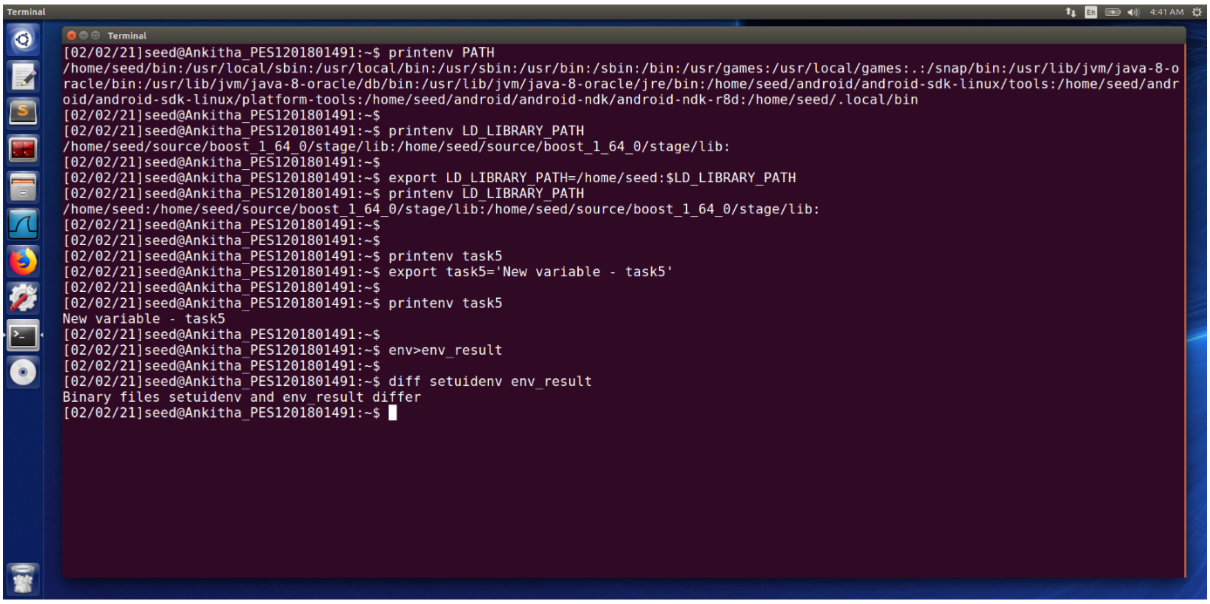


Above screenshot shows that the program is compiled, ownership changed to root, and made into Set-UID program. The mode is changed to 4755 that sets permissions so that User/owner can read,write and execute while Group and others can read, cant write and can execute.

<u>CASE 2:</u> The below screenshot shows that the program is compiled, ownership is the normal user and mode changed to 5744 that sets permissions so that User/owner can read,write and execute while Group and others can read, can't write and can't execute.



CASE 3:



In the bash shell (in the normal user account), export command is used to set some environment variables like PATH, LD_LIBRARY_PATH and a user defined task5 variable.

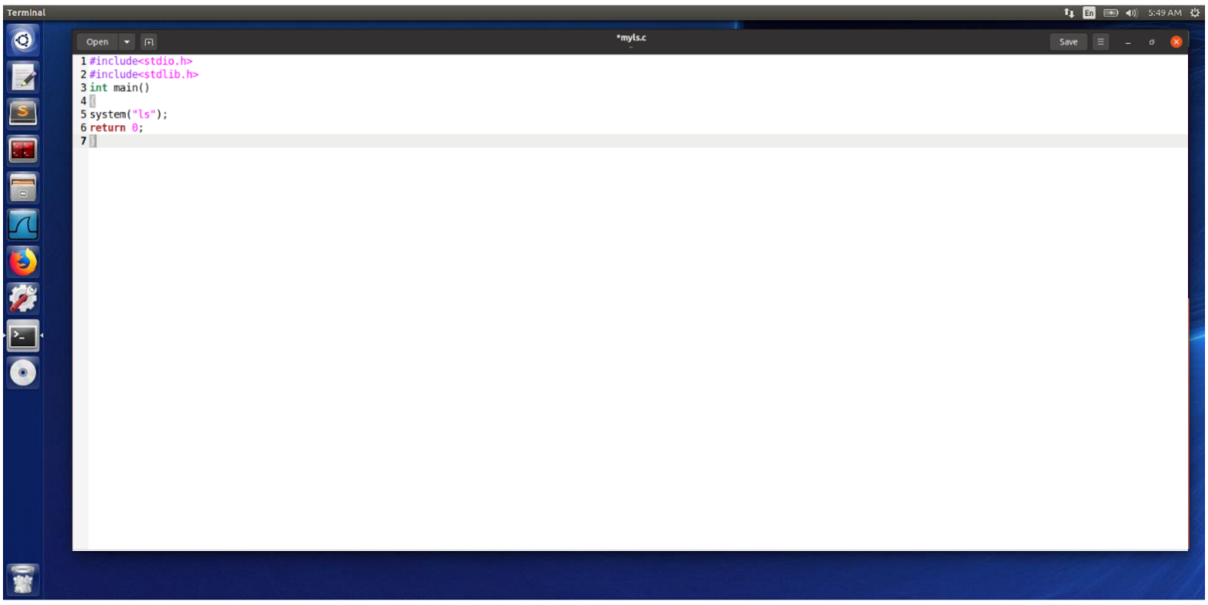
The result of setuidenv (the executable obtained by Set-UID program with permissions set to chmod 4755 and owner as root) and the env_result is not the same. Which implies that all the environment variables set in the shell process(parent) and Set-UID child process are not the same.

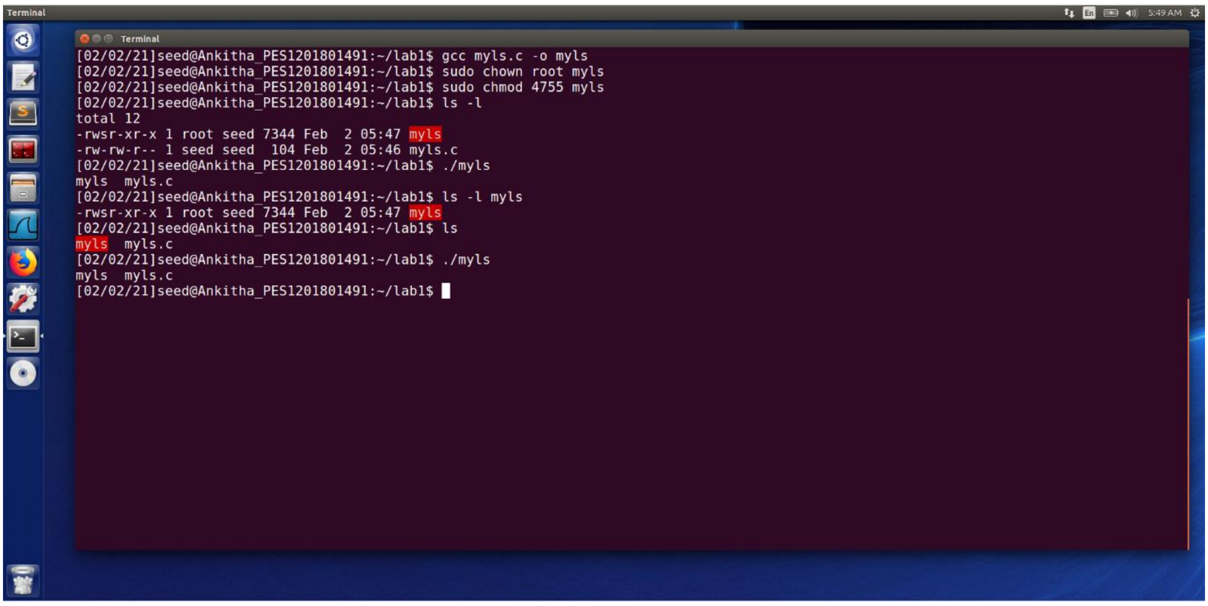
Also, it was observed that PATH variable and user defined task5 variable have been inherited by the set-UID pgm, whereas the LD_LIBRARY_PATH environment variable was not inherited by the process. This is a security mechanism implemented by the dynamic linker. The LD_LIBRARY_PATH is ignored here because the real user id and effective user id is different. That is why only the other two environment variables are seen in the output.

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

Because of the shell program invoked, calling system() within a Set-UID program is quite dangerous. This is because the actual behaviour of the shell program can be affected by environment variables, such as PATH; these environment variables are provided by the user, who may be malicious. By changing these variables, malicious users can control the behaviour of the Set-UID program.

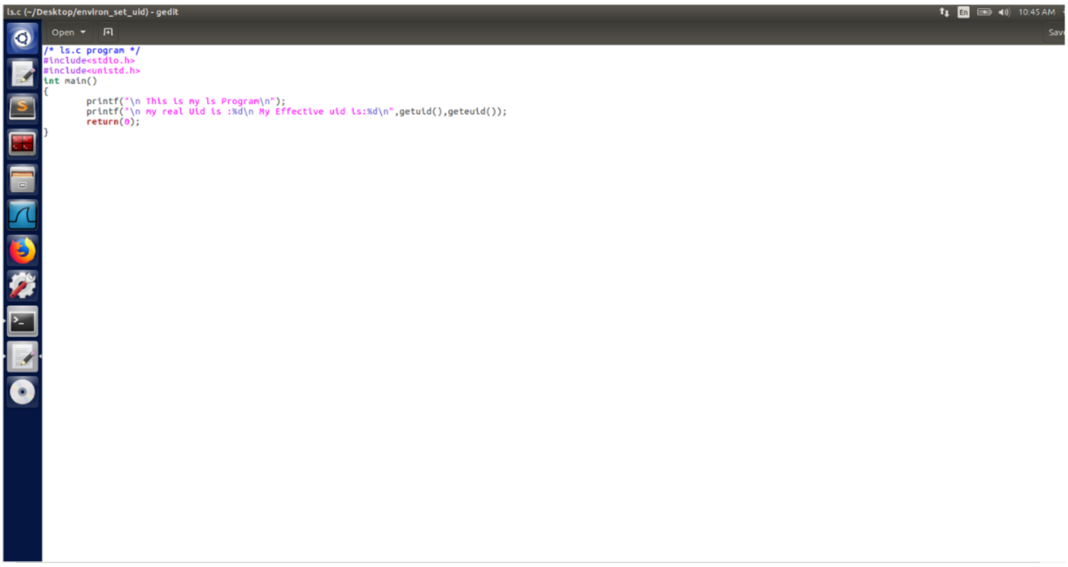
Case 1:

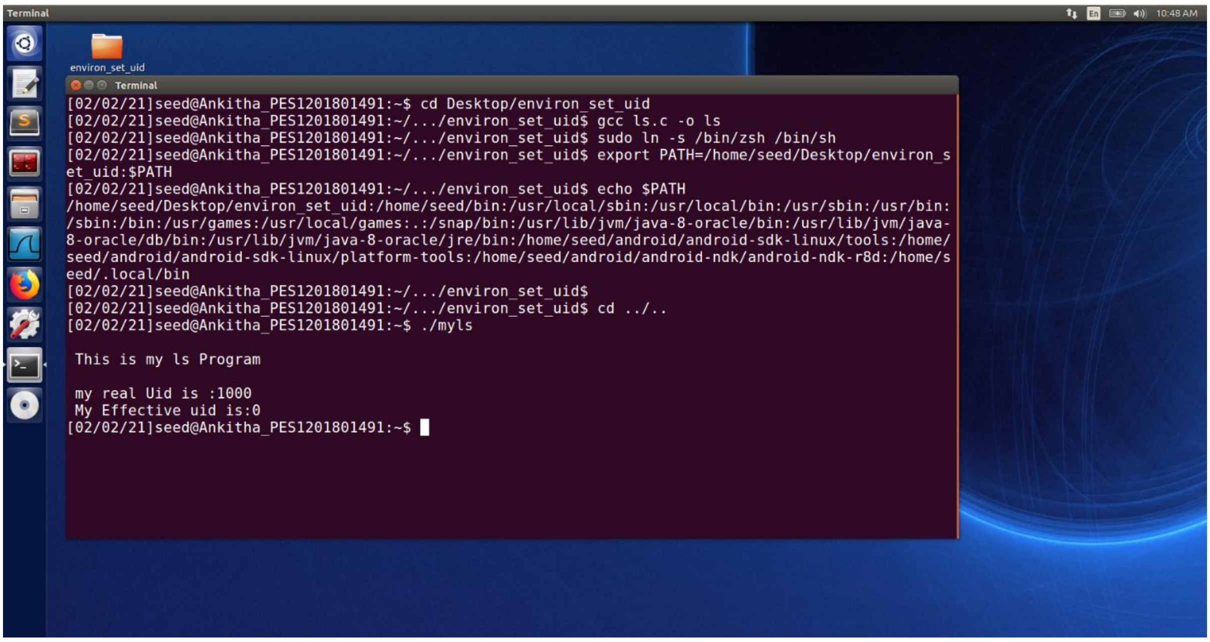




Here the myls is executable obtained by compiling myls.c, root as owner and permission as chmod 4755. On running this program, the ls command will be executed via system call and thus gives the same as that of /bin/ls.

Case 2:





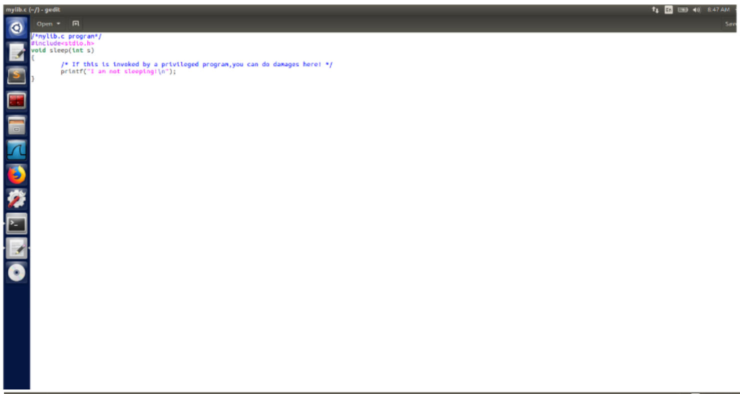
The ls.c program prints the userid and effective userid. The executable ls is added to the PATH environment variable. Since the PATH variable has higher priority, when the myls executable is called, it runs the ls executable instead of /bin/ls. The real UID is 1000 whereas the effective UID is 0.

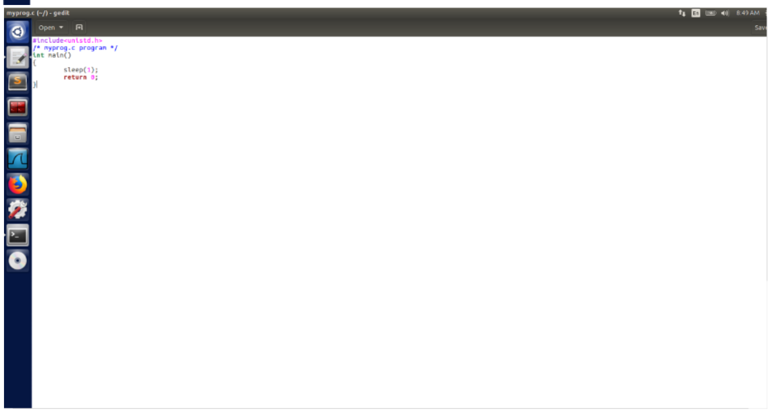
This shows the way in which PATH environment variable can be changed to point to a desired folder and execute the user-defined programs which could be malicious. Since we are using system(), it is potentially dangerous due to the inclusion of shell and the environment variables. Also, instead of specifying the absolute path, we have specified the relative path of the process. Due to this, the system() will spawn a shell which will look for a ls program in the location specified by the PATH environment variable. Hence by changing the PATH value to a folder containing a malicious file with the same name as specified in the program, the attacker can run a malicious code with root privileges because it is a root-owned SET-UID program. Hence using relative path and system function in a SET-UID program could lead to severe attacks.

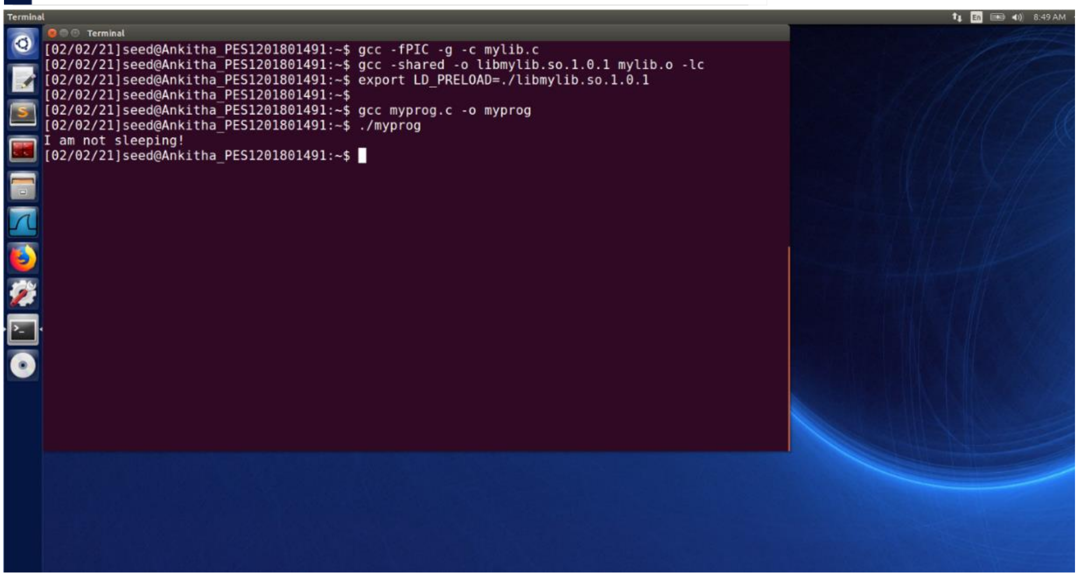
Task 7: The LD PRELOAD environment variable and Set-UID Programs

In Linux, LD LIBRARY PATH is a colon separated set of directories where libraries should be searched for first, before the standard set of directories. LD PRELOAD specifies a list of additional, user-specified, shared libraries to be loaded before all others.

mylib.c just prints to the terminal. mypro.c has a sleep function for 1ms





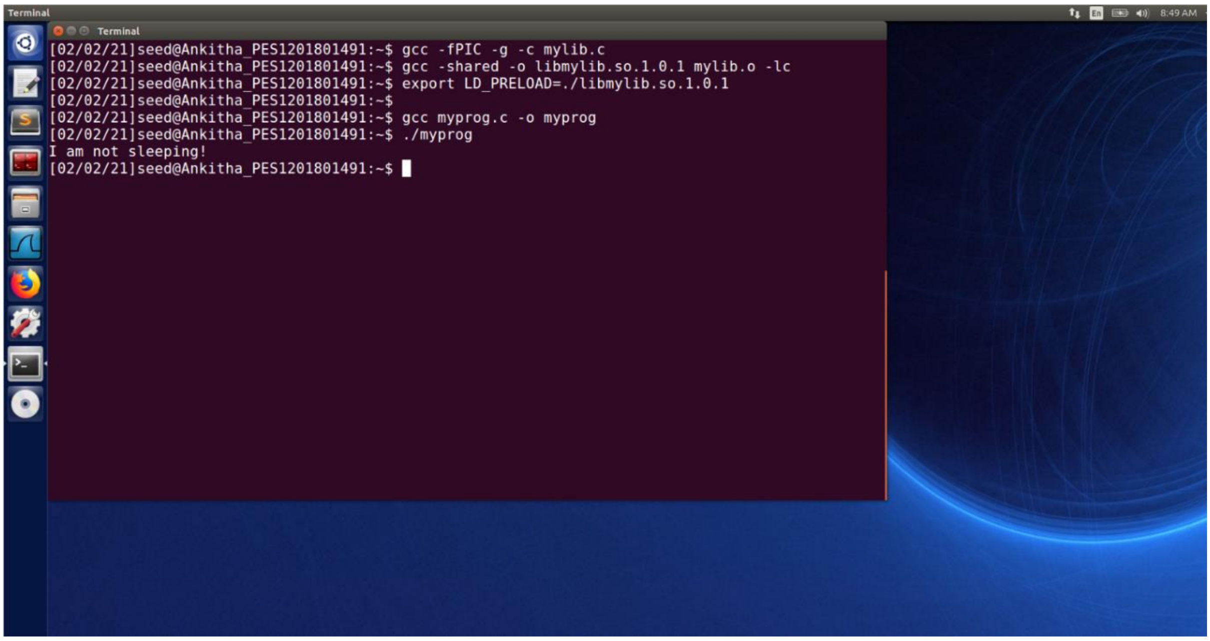


In the above screenshot, we compile the program using the following command: gcc -fPIC -g -c mylib.c (where -fPIC means that emit position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table, -g means producing debugging information and -c means compiling the file but not linking it.

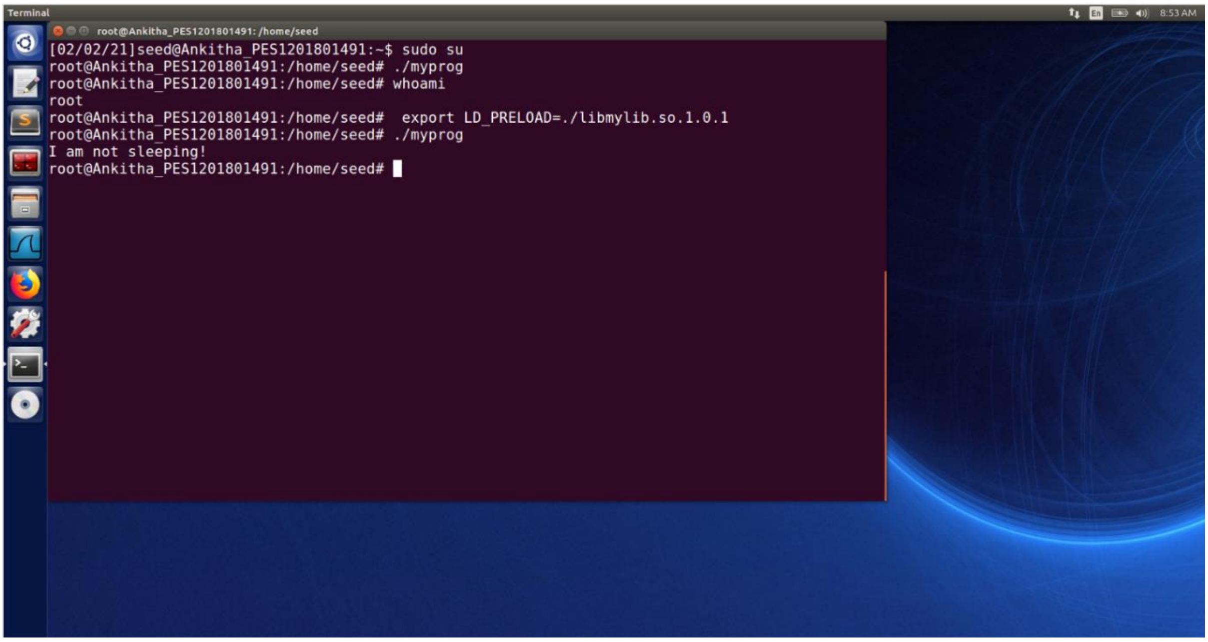
LD PRELOAD environment variable is set with Command: $ export LD_PRELOAD=./libmylib.so.1.0.1 myprog is compiled , and placed in the same directory as the above dynamic link library libmylib.so.1.0.1. When compiled program is run by a normal user, the sleep function in myprog gets overwritten and we get the "I am not sleeping message" from mylib.c.

STEP 2:

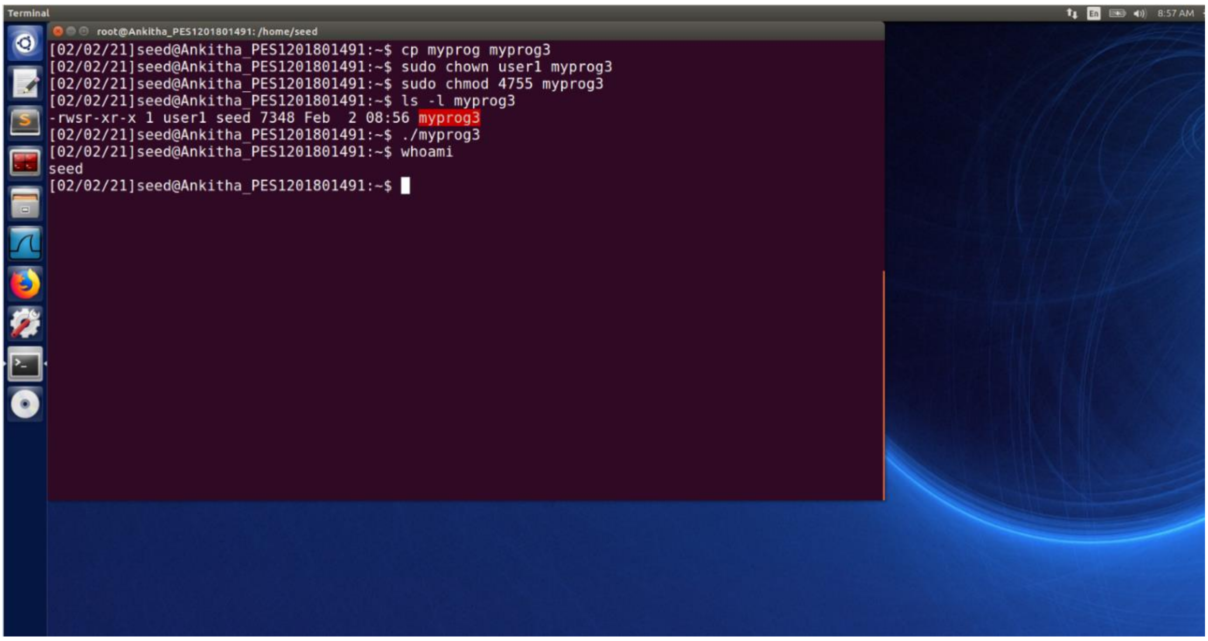
- When the myprog is made as Set-UID program and run as normal user, the system sleep function is called for 1 second. The library containing sleep function was not called and also the environment variable of that process did not contain the LD_PRELOAD variable. This showed that the SET-UID child process that was created did not inherit LD_PRELOAD variable and hence it did not load user defined library and function but the system-defined sleep function causing the program to sleep



- Since the program is already a SET-UID root program, I just logged into the root user account and defined the LD_PRELOAD variable. On running the program, we see that the user-defined sleep function is executed and LD_PRELOAD variable is present. This happens because we are in the root account and the function’s owner is root as well. This makes the process have the same real ID and effective ID, and hence the LD_PRELOAD variable is not dropped

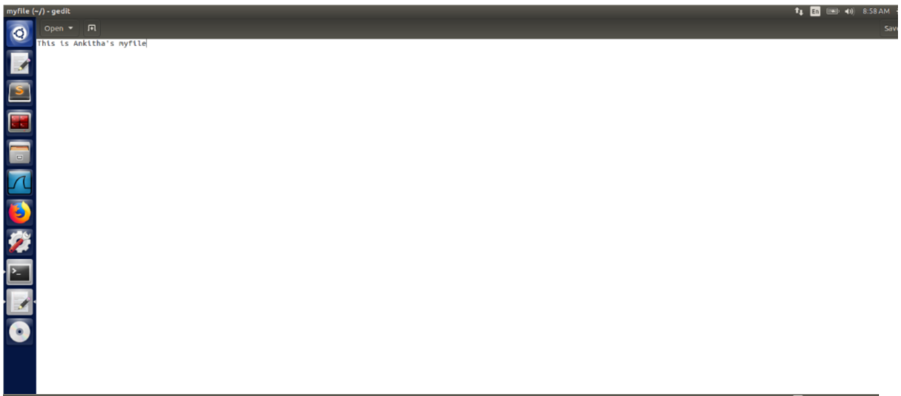


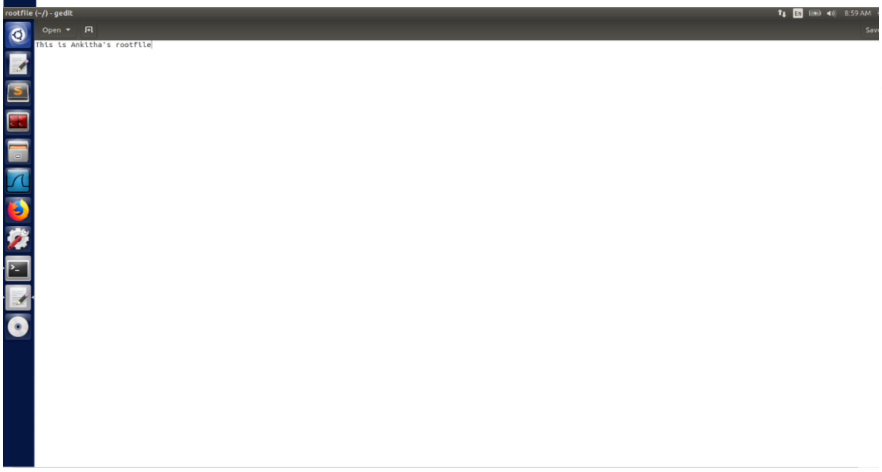
- The file's owner is made as user1 (another user account other than root) and make it a SET-UID program. On running the program again, we see that user-defined sleep function is called.



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

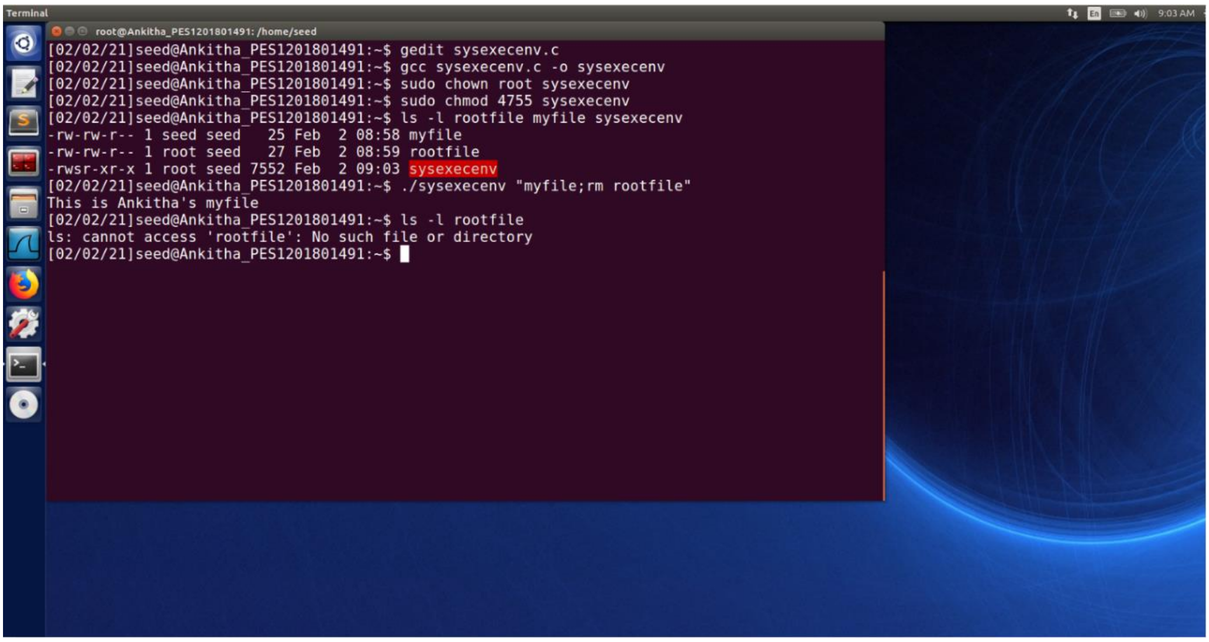
we have two files called myfile and rootfile where myfile is owned by the normal user and rootfile is owned by root.





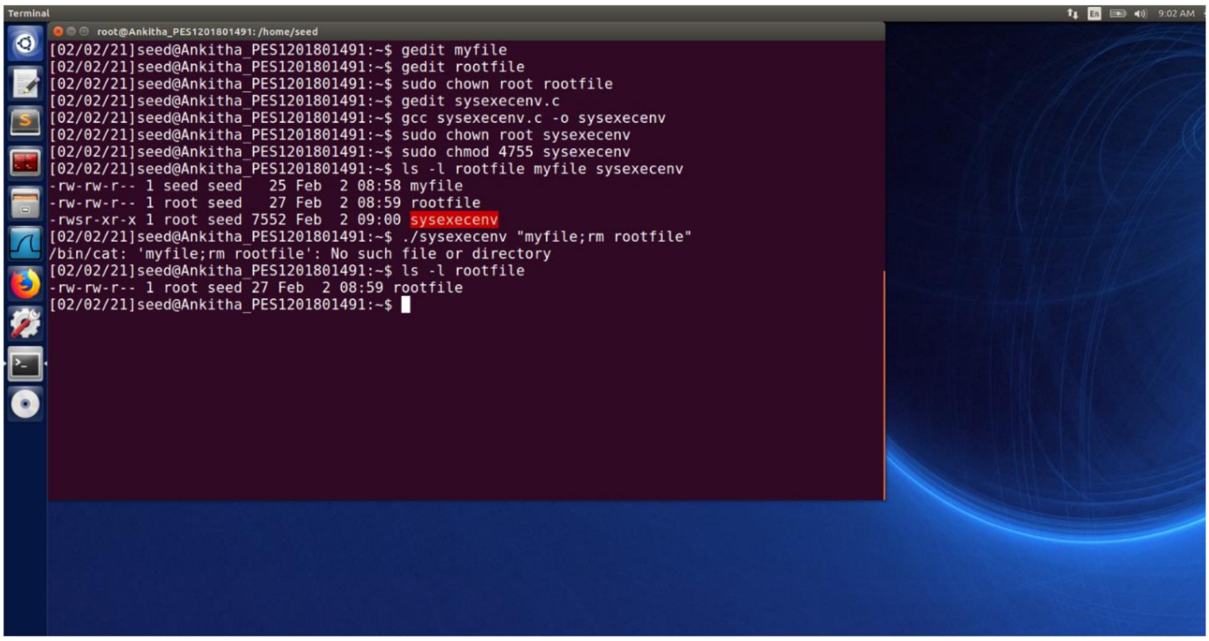
The below program requires a file name as command line input and then it will run /bin/cat to display the contents of the specified file. Both system and execve are used to run /bin/cat on the path provided as command line input. In the first case, the execve line is commented.

Case 1:



We make the program a root owned set-uid program using chown and chmod and we run the program with an input “myfile;rm rootfile”. Separately as commands the part after ‘;’ would be used to remove the rootfile. Since the system command passes all its arguments to shell, the inputs are treated as semicolon separated commands and are run leading to the removal of rootfile after the printing of myfile. Using `system`, we can modify unwanted files and easily compromise the integrity of the system.

Case 2: In the second case, we comment the system line and uncomment the execve line. Now the command line argument will be passed to the execve call.

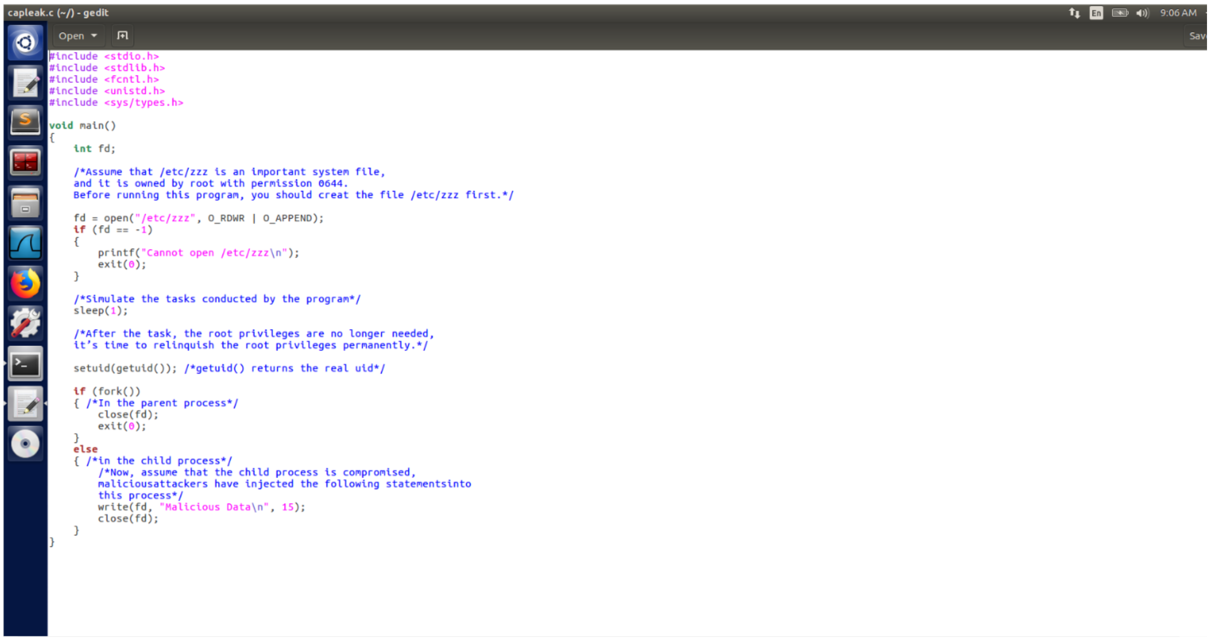


execve simply executes the program that is passed to it as the first argument. The program must be a binary executable or a script starting with a shebang directive and since our argument does not comply and is not a file, it throws an error. The attack does not work like in the first case where we use system function. The execve call is less dangerous than the system call.

Task 9: Capability Leaking

The below program is created to simply open a file /etc/zzz and then simulate a task by sleeping. The privileges are no longer needed after the simulated task so they are relinquished.

We compile the given program and make it root-owned SET-UID program



Here, we create a file named zzz in the /etc folder which can symbolise any important file which may fall under attack.

we run the program zzz file again, and we see that the file content is modified. This happens because even though in the program, we dropped the privileges, we did not close the file at the right time and hence the file was still running with privileged permissions that allowed the data in the file to be modified, even without the right permissions. Here, after calling fork, the control is passed to the child process and hence the malicious user is successful in modifying the content of a privileged file. This shows that it is important to close the file descriptor after dropping privileges, in order for it to have the appropriate permissions.