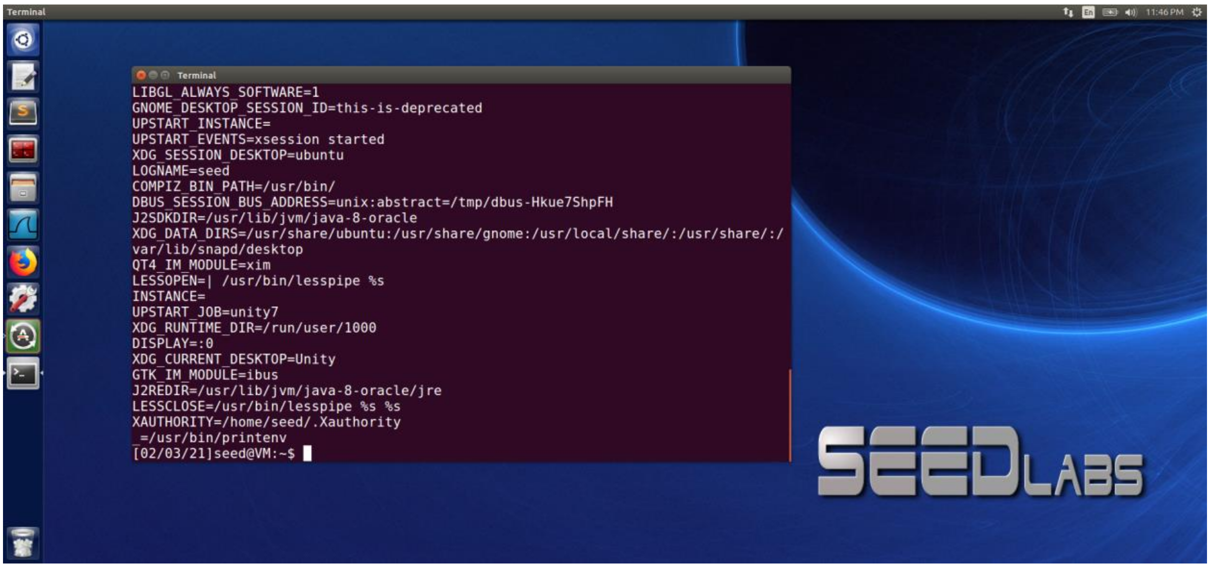
Name: Varun Pande

UTA-ID: 1001722538

2.1 Task 1: Manipulating Environment Variables:

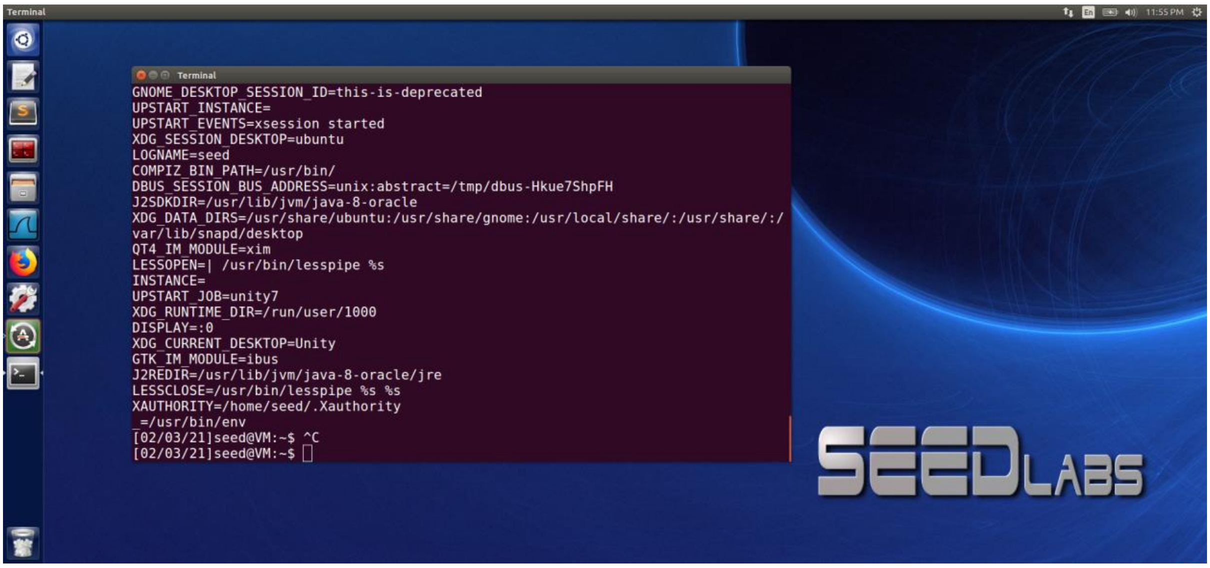
Cmd : printenv



Observation:

On executing the “printenv” command we can see that all the set environment variables get out putted to shell. Special ENV Variables like PATH are used to point the path of common executables, so that they can directly made to run simply by typing the name of program.

Cmd : env



Observation:

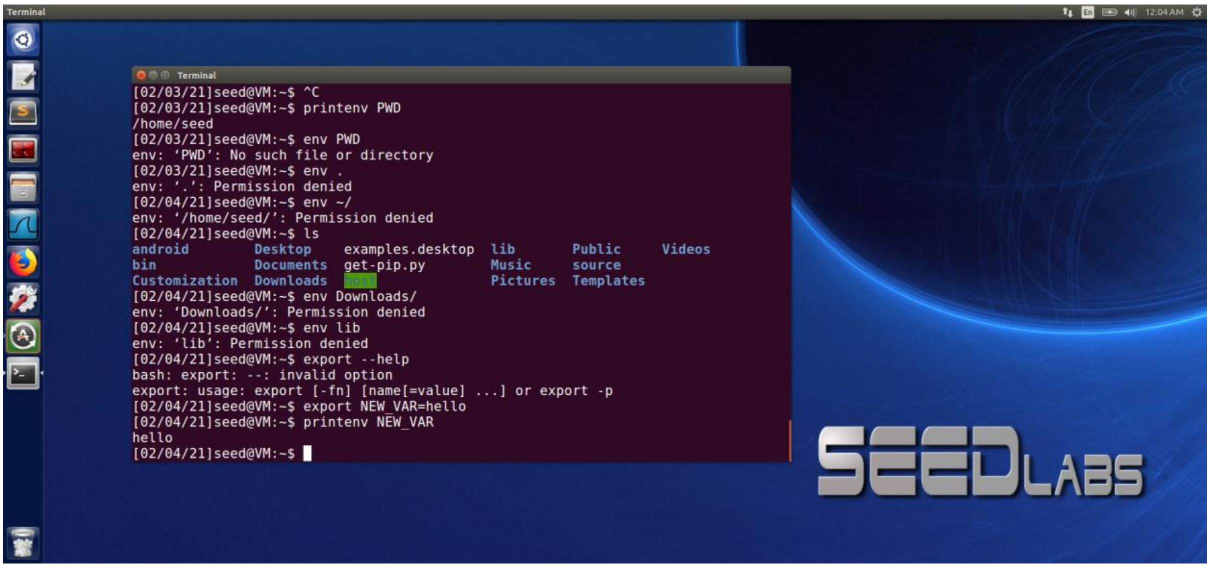
The same variable can be observed using the env command as well, but on careful observation we can see that at the end, we can figure out which command was use to display the variables.

Cmd : printenv PWD or env | grep PWD

Observation:

To filter the long list of ENV variables we can use printenv <name-of-variable> or env | grep <name-of-variable>. Note we cannot simply type “env PWD” it will result into an error.

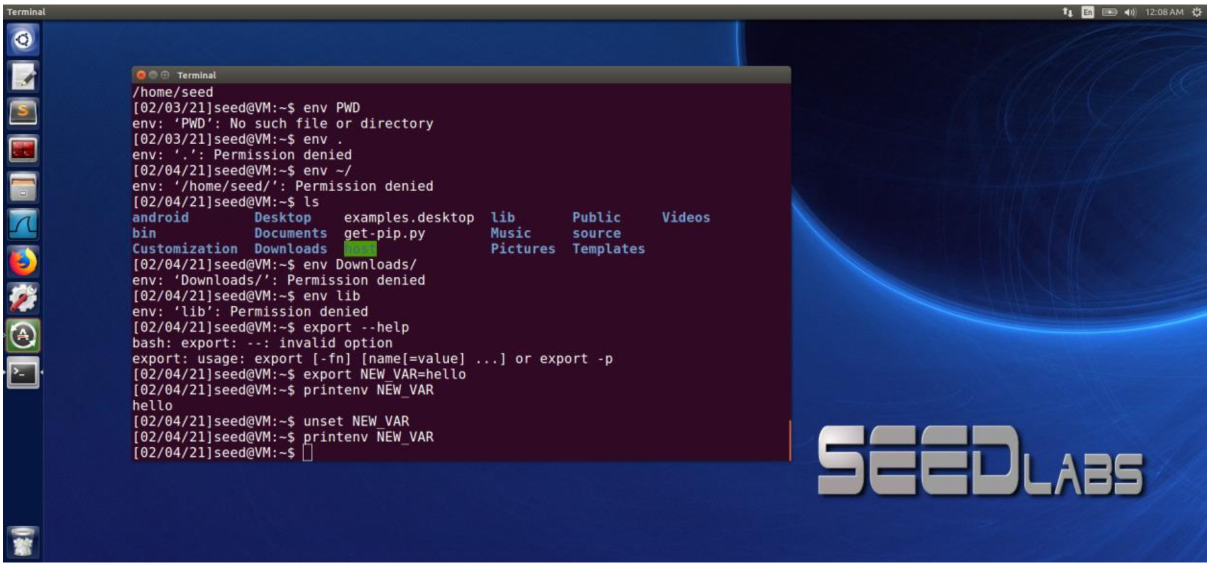
Cmd : export <VAR_Name>=<Value>



Observation:

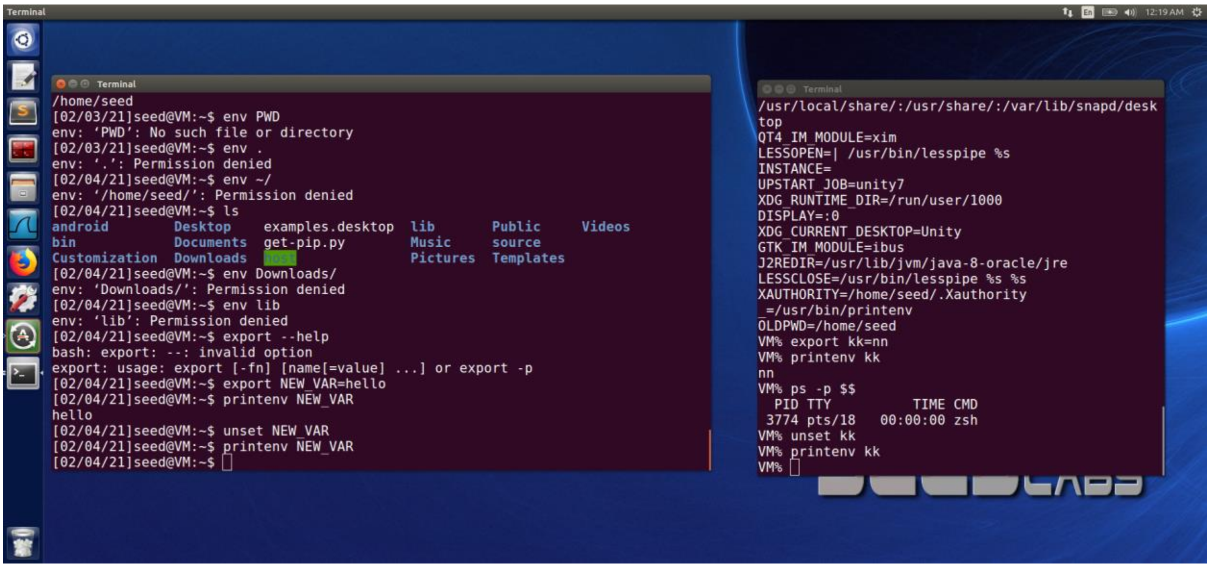
We can use export to create new ENV variables or set values of the one that already exist, and then run printenv to check if the variables are set. Sometimes we need to restart the bash terminal inorder for the change to take place.

Cmd : unset <ENV_VAR_name>



Observation:

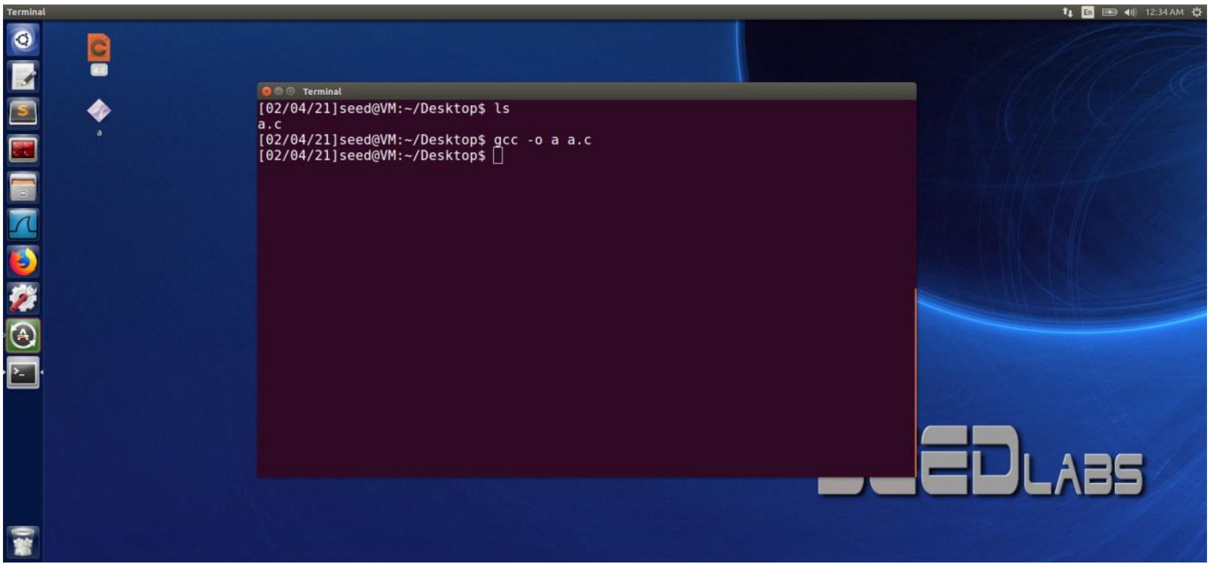
The unset command is used to delete the entry of ENV variable. On trying the above commands on "zsh" they seem to work just fine:



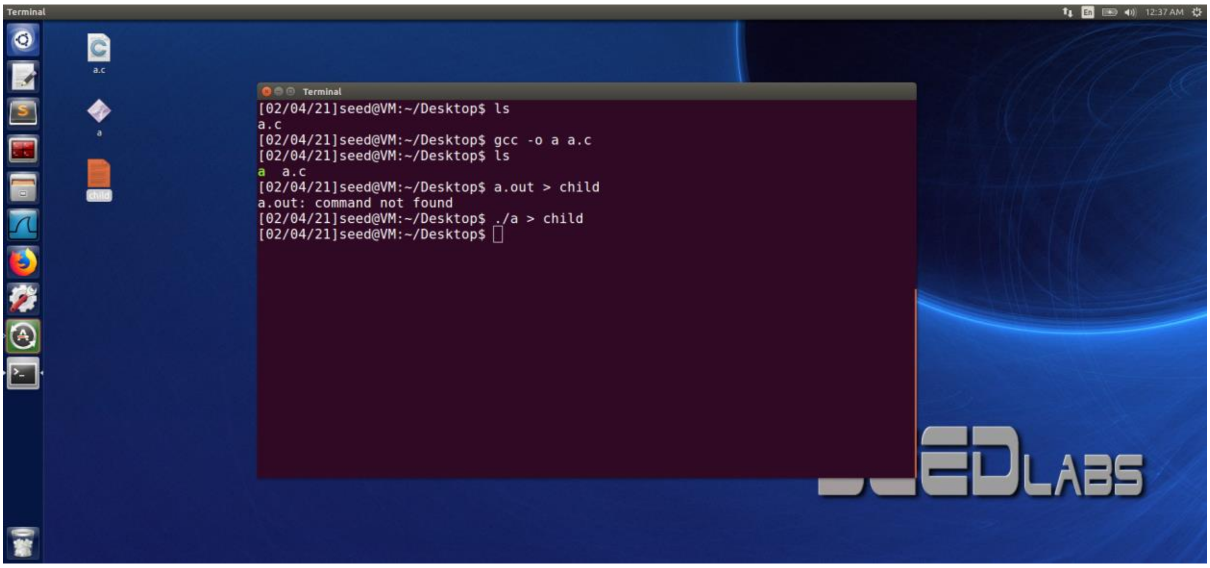
2.2 Task 2: Passing Environment Variables from Parent Process to Child Process

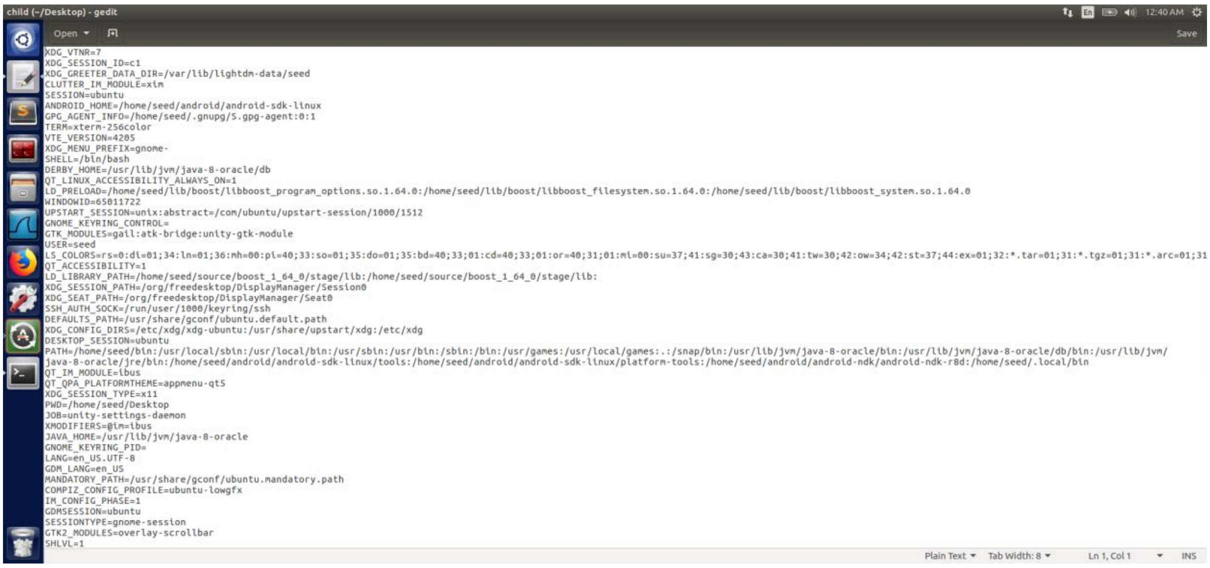
Step 1:

Created file "a.c" using vim on Desktop, compiled it using "gcc -o a a.c"



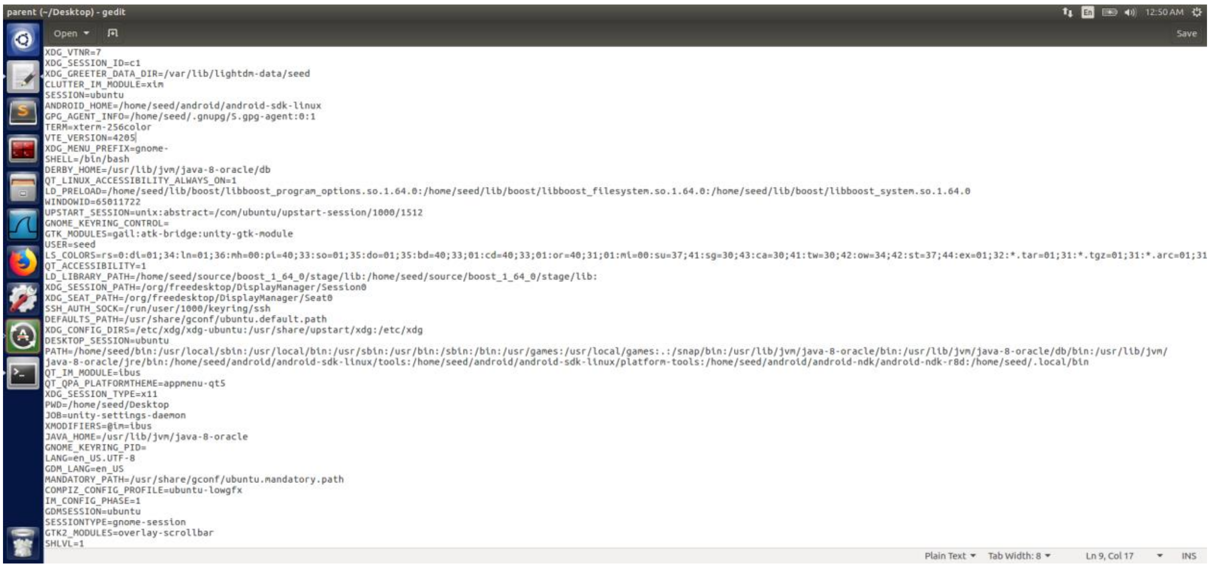
Observation:





On executing the above program all the environment variables of the forked child process got saved to child.txt file.

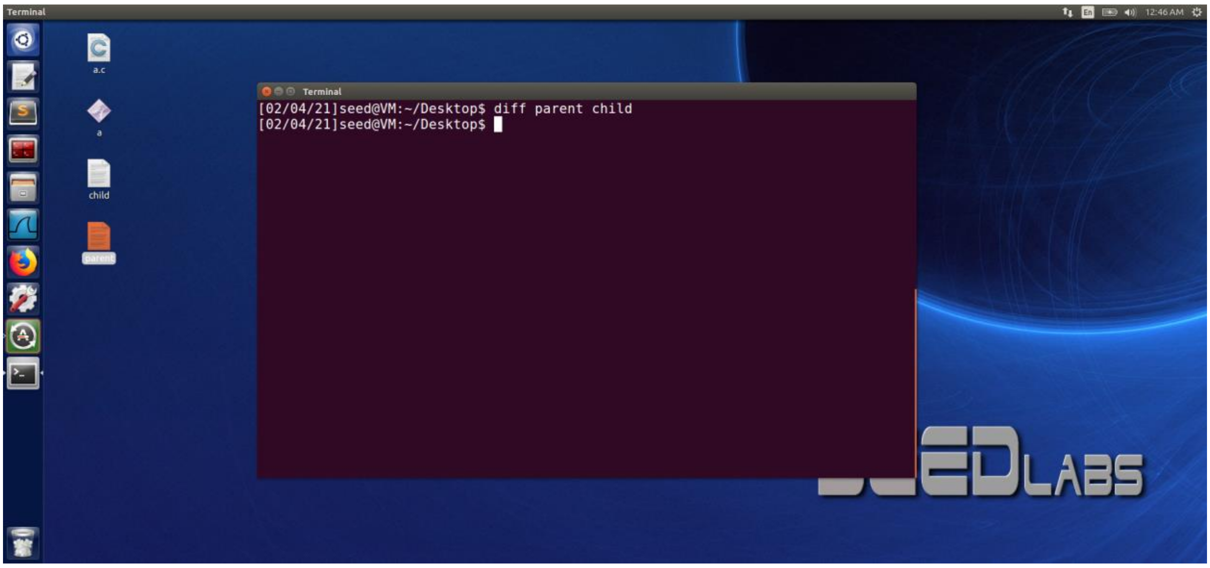
Step 2:



By commenting and uncommenting the required lines, now on executing the program the ENV variables of the parent process was saved in the parent.txt file.

Step 3:

On performing a diff on the files no difference was observed, because the fork command creates a duplicate of parent process.

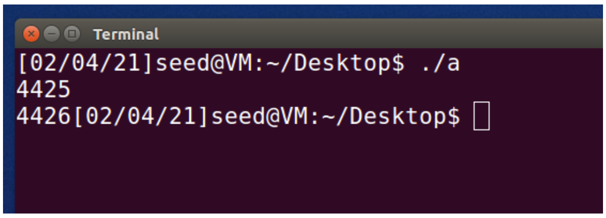


By Using the below command I have tried to print the process ID of the child and parent process:

printf("%d\n", getppid());

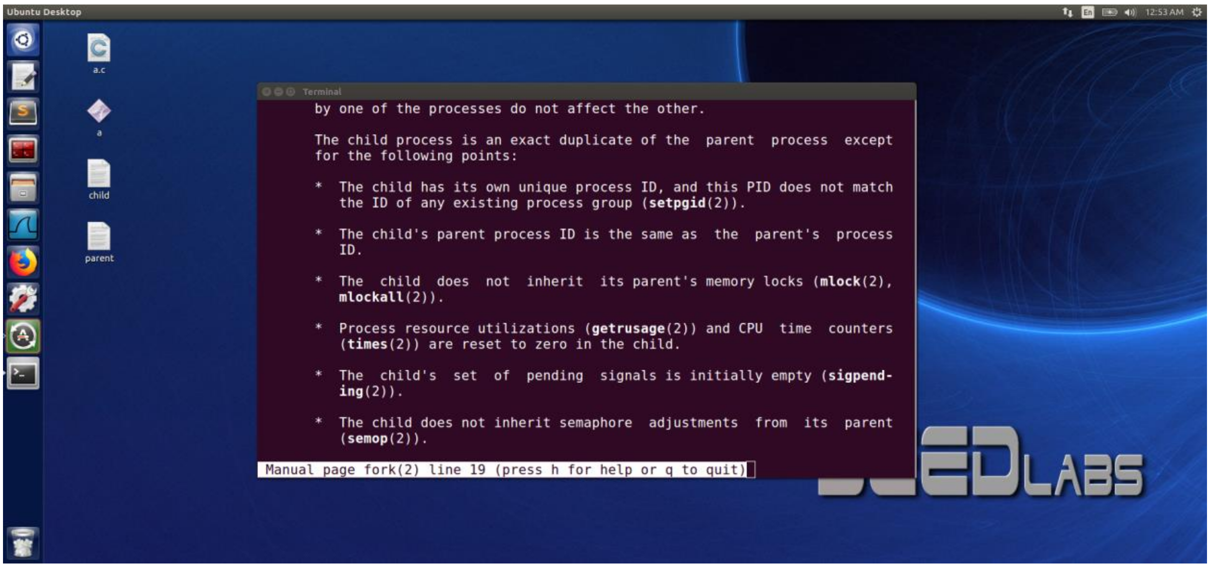
printf("%d",getpid());

following is the output:



It can be clearly seen from here that the parent and child have different PID's

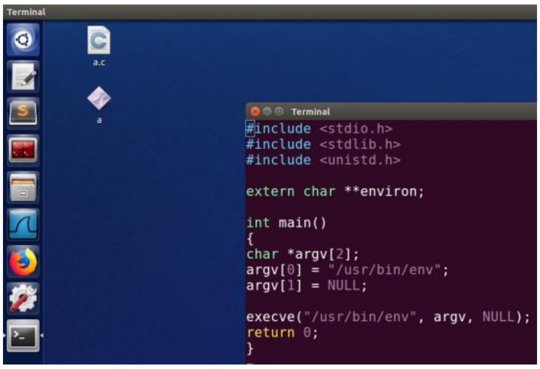
These are some of the differences that are highlighted on the man page of fork:



2.3 Task 3: Environment Variables and execve():

Step 1:

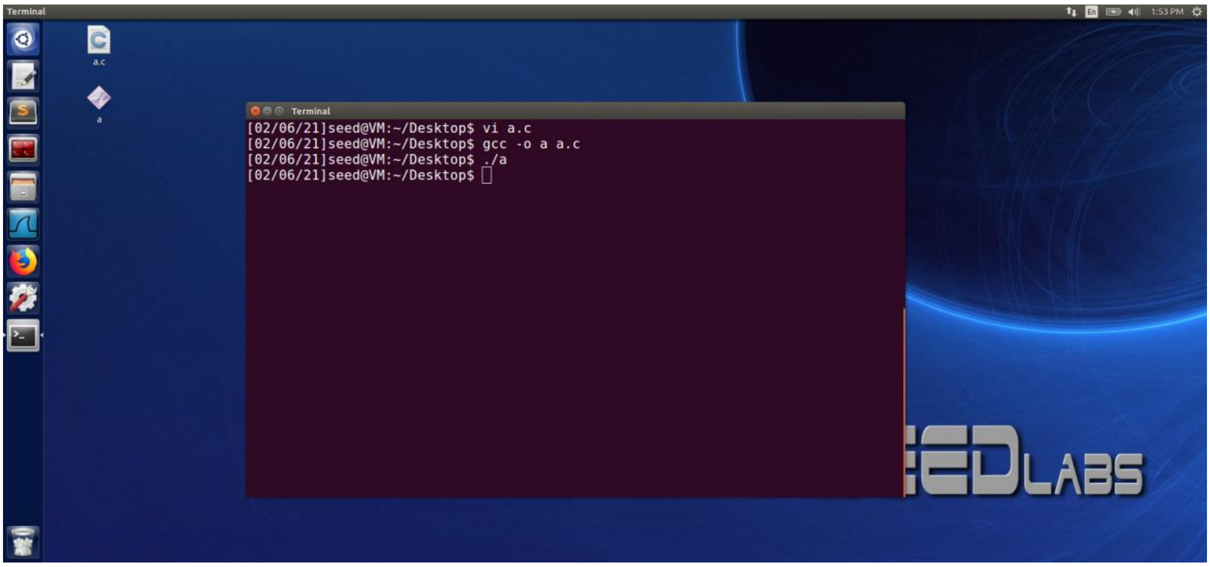
Compile and run the given code:



On executing the below code:

```
argv[0] = "/usr/bin/env";
argv[1] = NULL;
execve("/usr/bin/env", argv, NULL);
```

There was no output at all i.e. nothing was returned from the function.

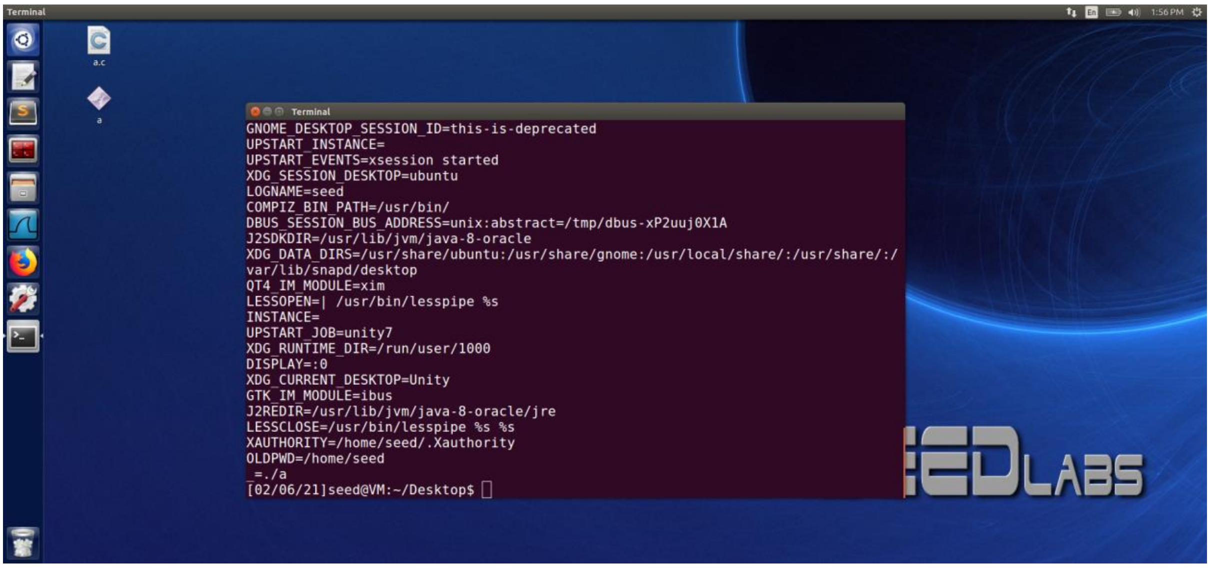


Step 2:

After changing to following code:

```
execve("/usr/bin/env", argv, environ);
```

Now we see the environment variables getting printed (also we can see from the last line that this function was called by the written code):

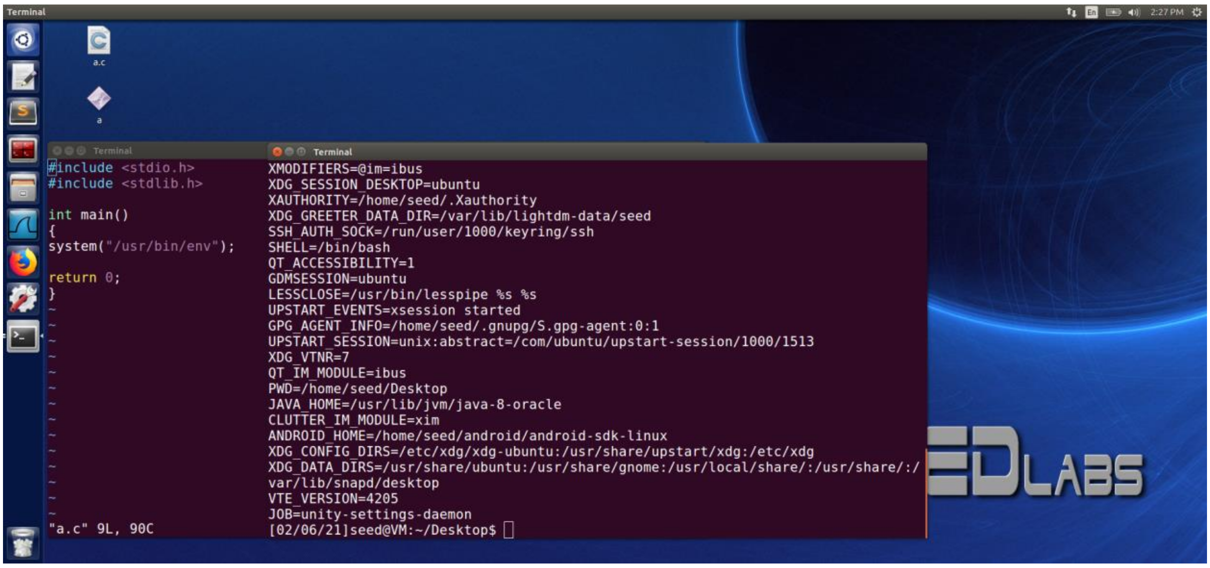


Step 3:

The third argument to execve() is the array of strings of environment variables. In the given code below lines were used to get the list of environment variables and pass it to the execve function.

```
extern char **environ;

execve("/usr/bin/env", argv, environ);
```

2.4 Task 4: Environment Variables and system()



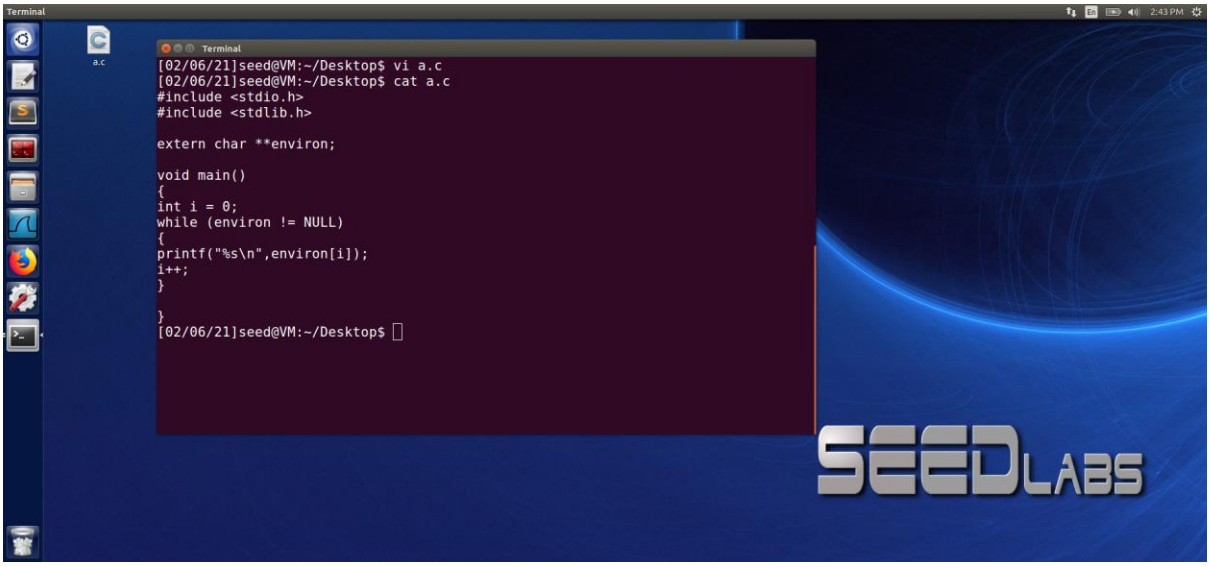
Observation:

From the above screenshot we don’t need to pass in the environment variable and also we can see that using system function internally calls a sperate function to run “execve” as the last line does not show any “_=./xx” being used to execute the “execve” cmd unlike that in task 3.

2.5 Task 5: Environment Variable and Set-UID Programs

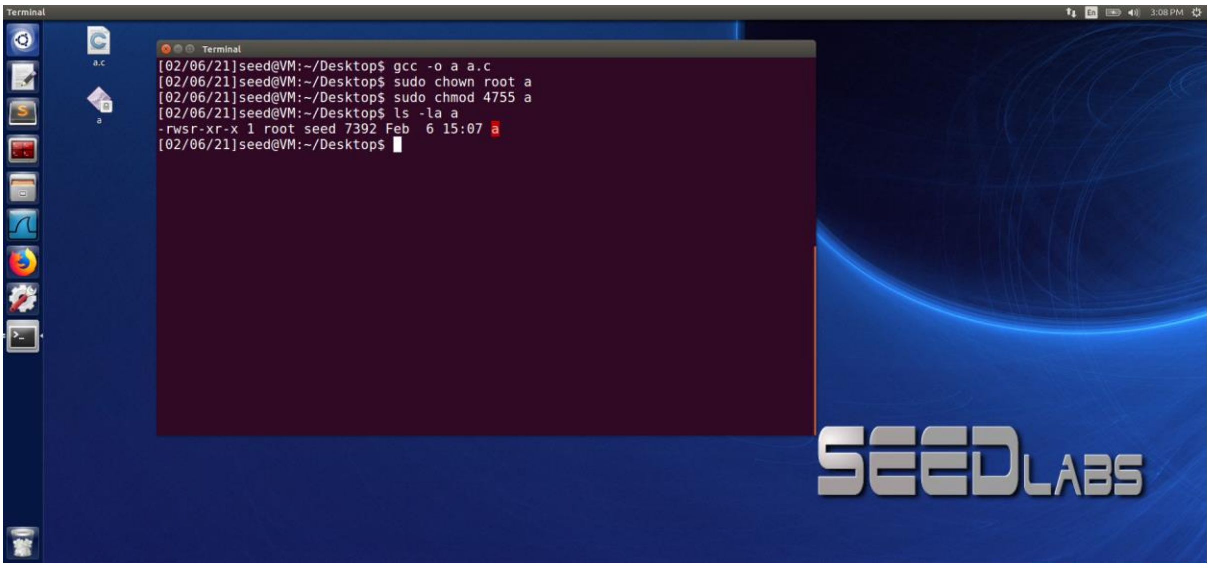
Step 1:

Writing program.



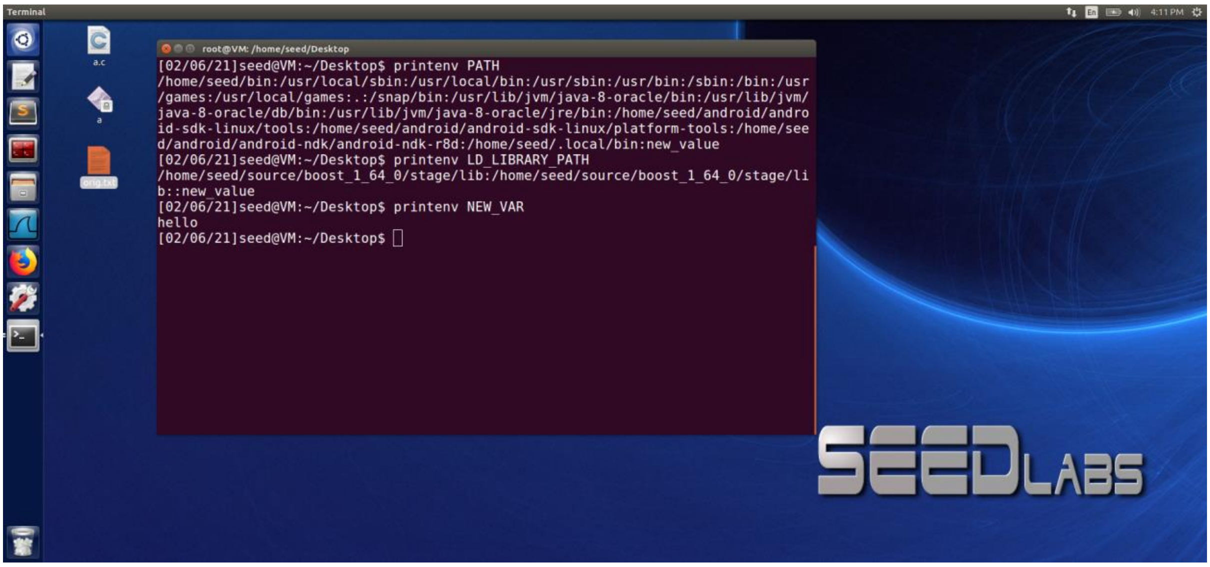
Step 2:

Compile, change its ownership to root, and make it a Set-UID program:



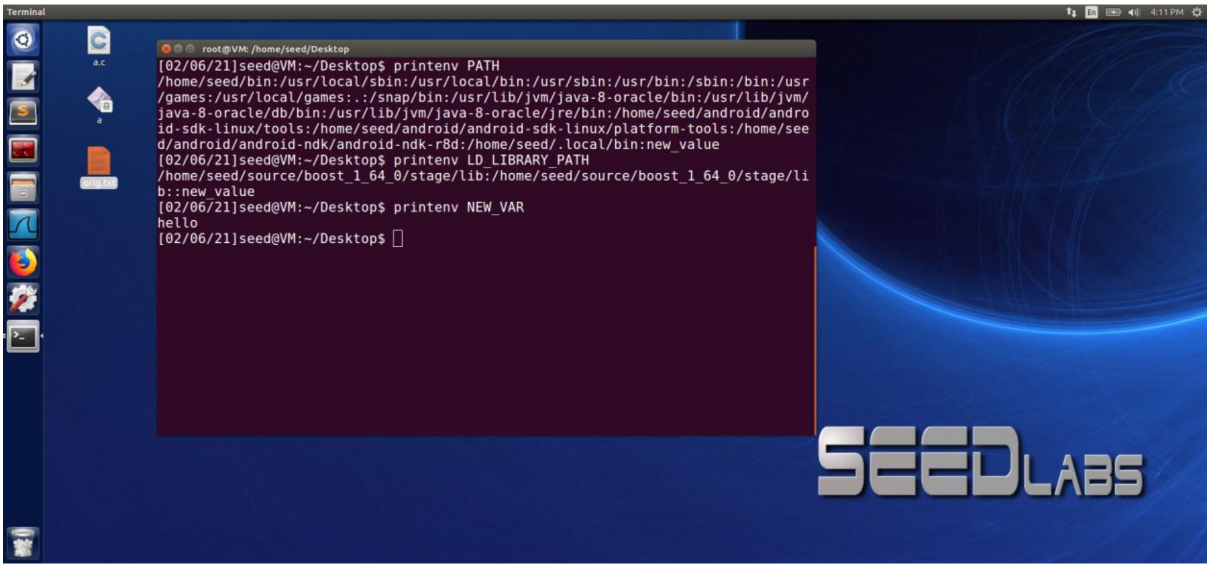
Step 3:

Checked the variable PATH, LD_LIBRARY_PATH and set the NEW_VAR variable using export. (new_value is appended to all the variables.)

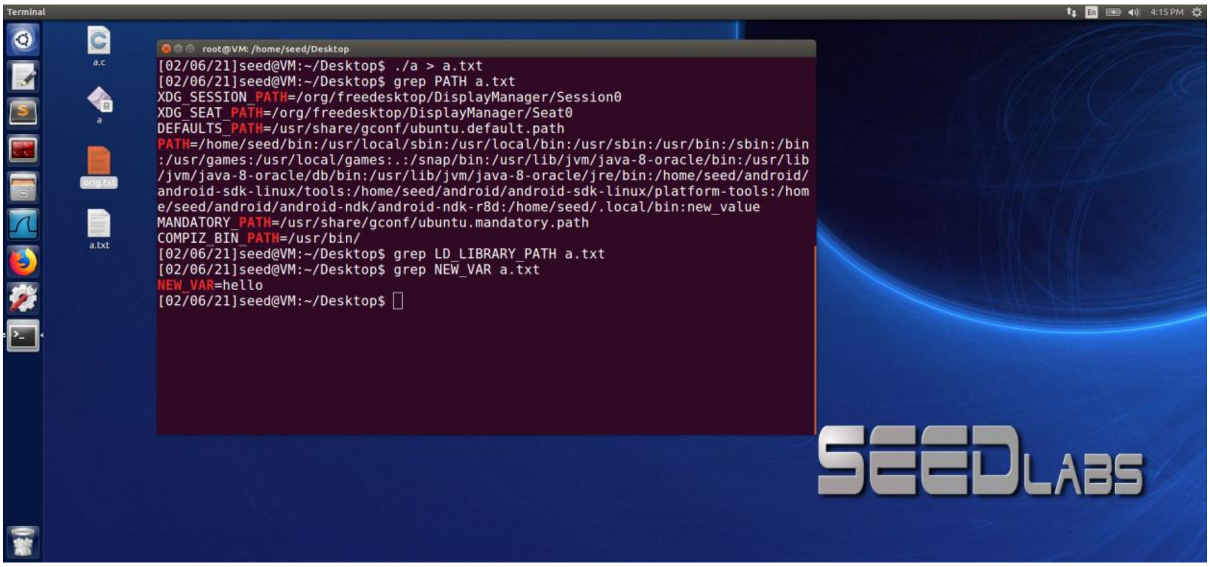


Observations:

PATH and LD_LIBRARY_PATH variables before the script.

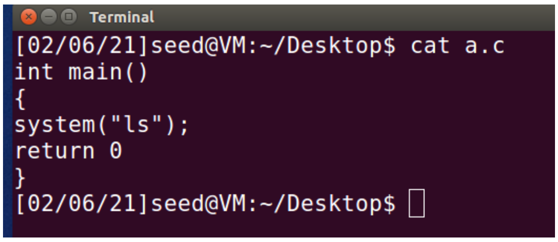


Only PATH and NEW_VAR variables were present, LD_LIBRARY_PATH variable after the script is run was not seen in the output.

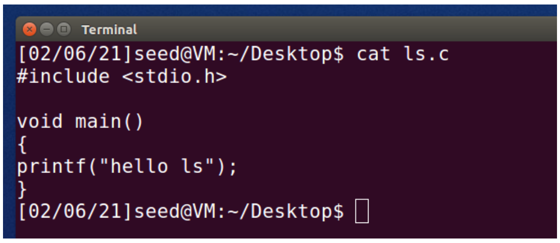


2.6 Task 6: The PATH Environment Variable and Set-UID Programs

Writing the program:

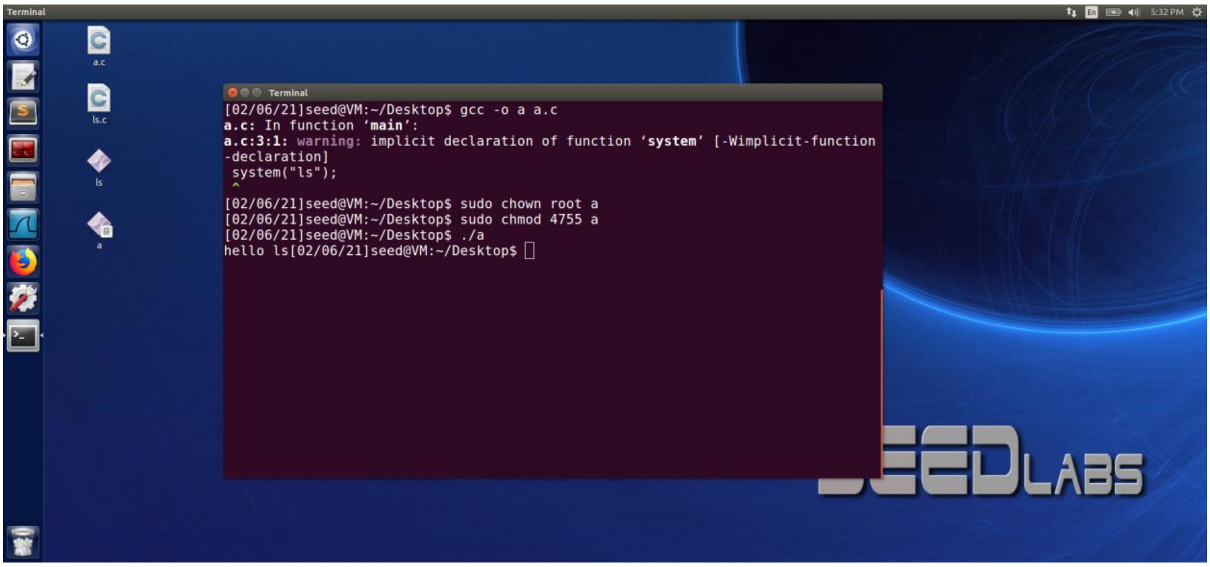


My custom program:

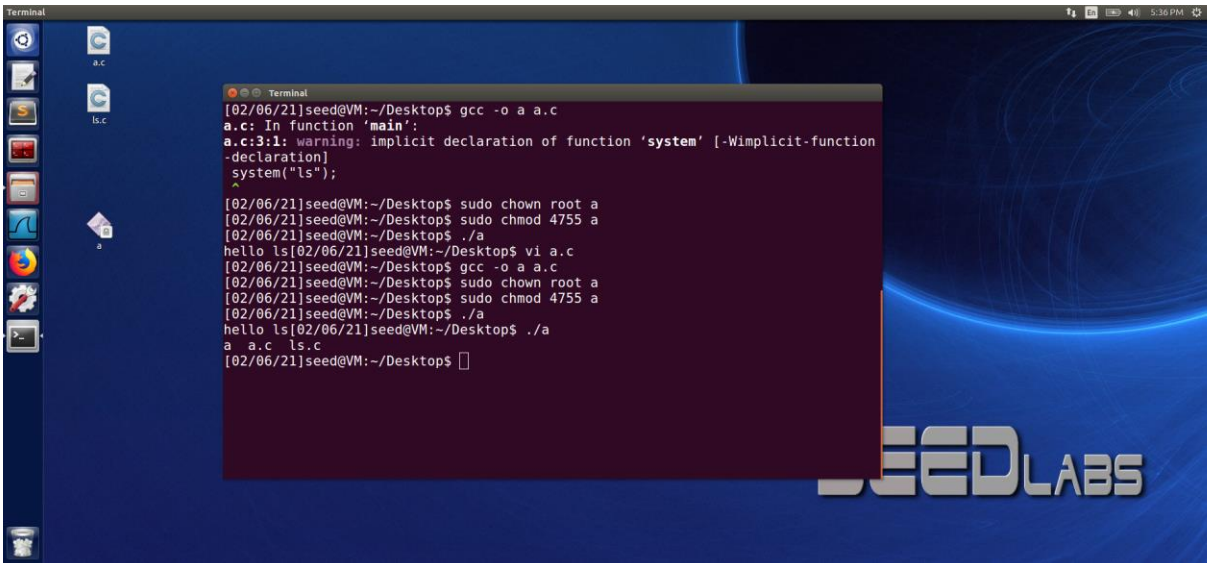


Pre-pending the path of Desktop so that when the program searches for ‘ls’ implementation while linking it reaches to my custom implementation. On compiling and executing we can see that it prints “hello ls”.

```
export PATH=~/Desktop:$PATH
```



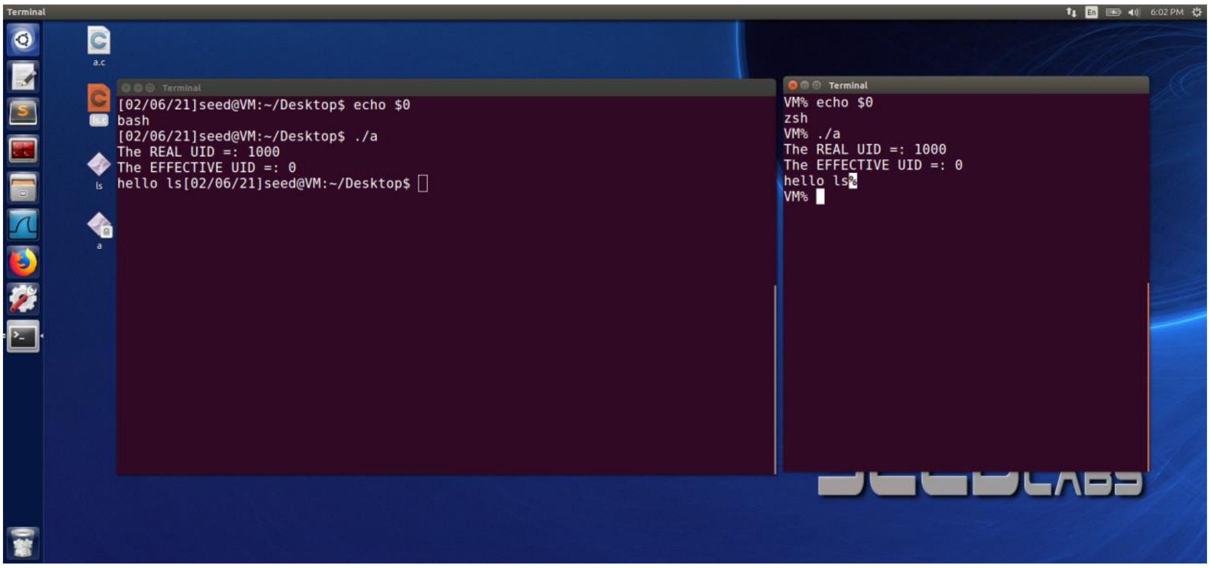
If I move my "ls" implementation to some place else it runs the /bin/ls implementation



On adding the following line to my implementation of ls to see the EUID and real UID:

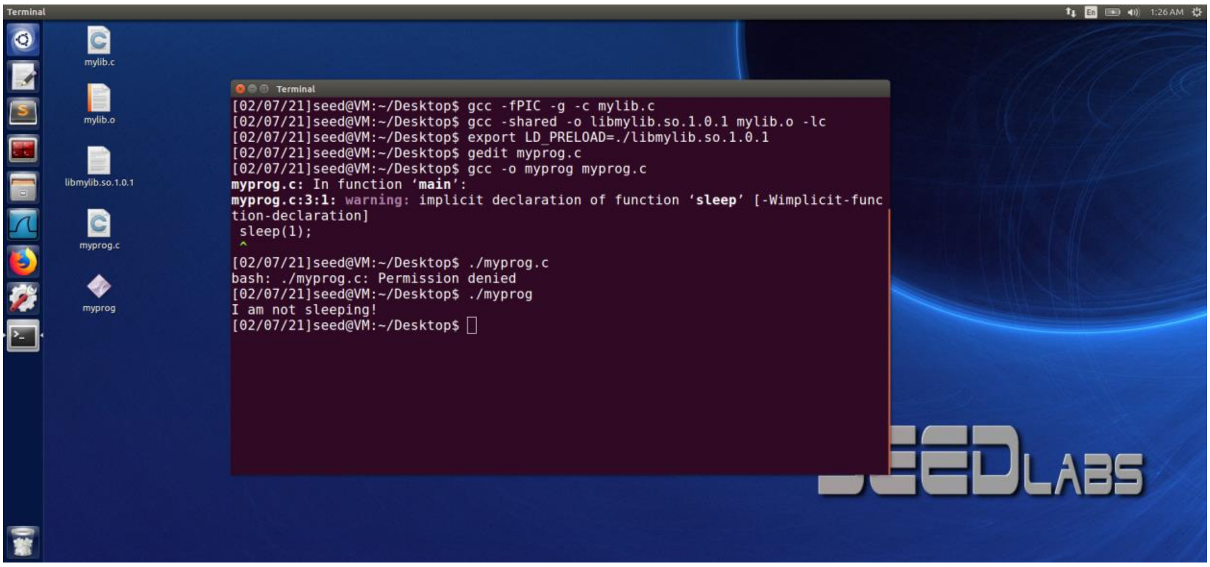
```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>


void main()

{

int real = getuid();

int euid = geteuid();

printf("The REAL UID =: %d\n", real);

printf("The EFFECTIVE UID =: %d\n",
euid);

printf("hello ls");

}
```

The output is same for both the shells (zsh and bash):

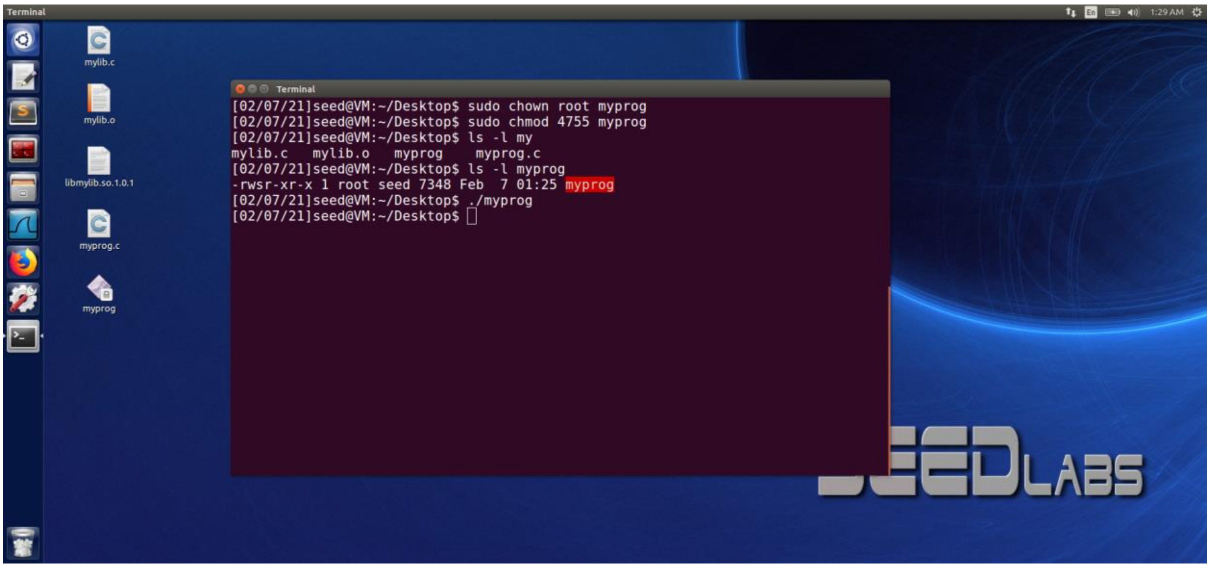


2.7 Task 7: The LD PRELOAD Environment Variable and Set-UID Programs:

1. Make myprog a regular program, and run it as a normal user.

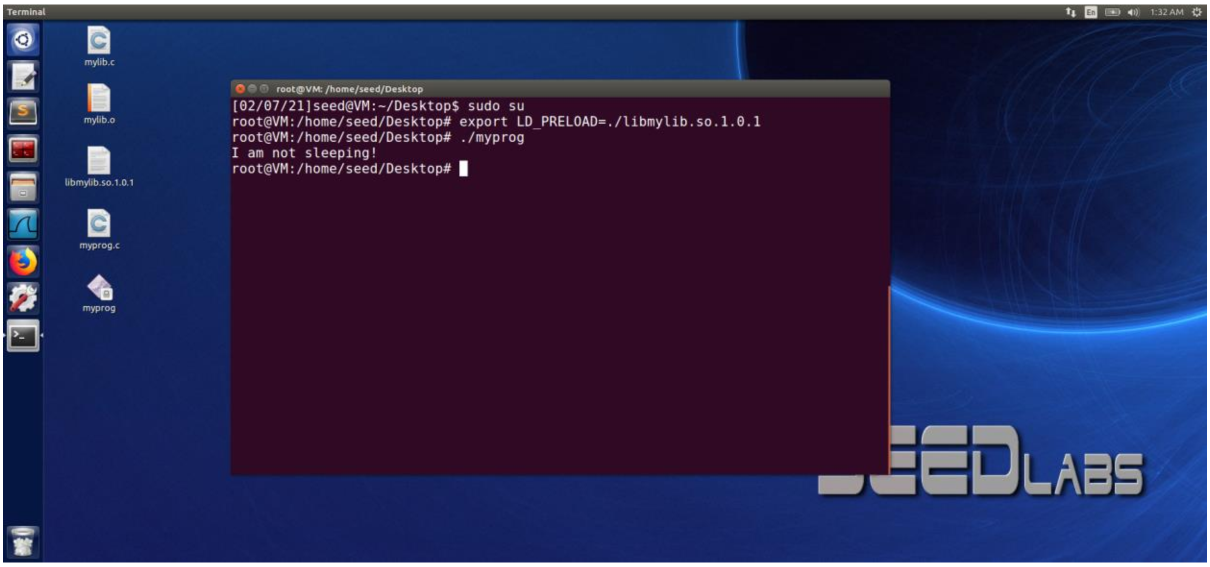


2. Make myprog a Set-UID root program, and run it as a normal user.



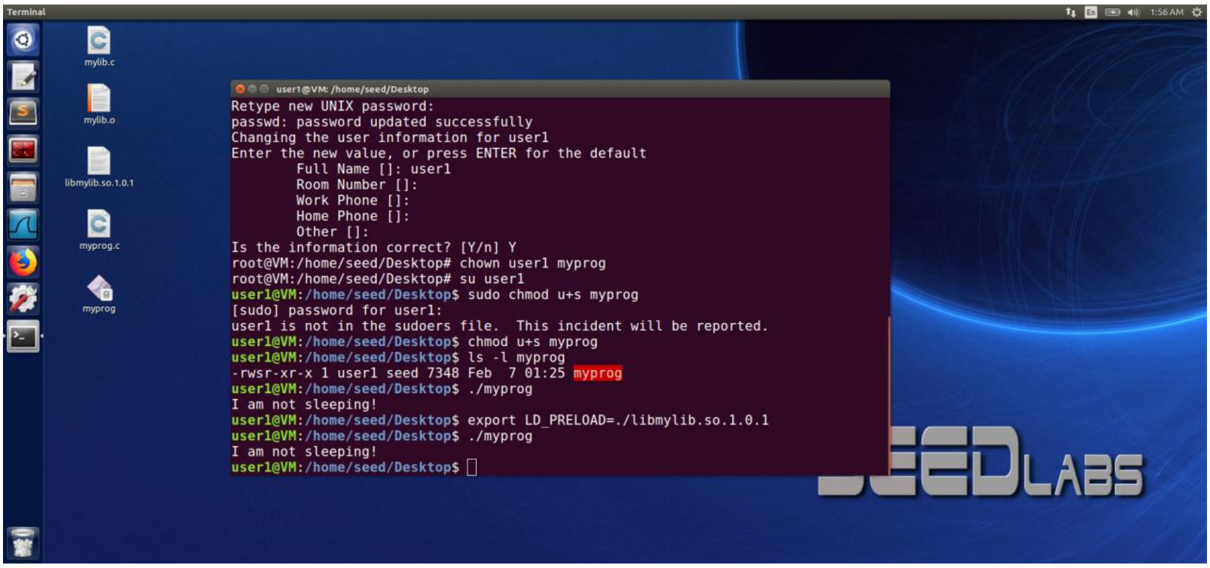
Observation: It paused for a second, i.e. it actually went to sleep for a second.

3. Make myprog a Set-UID root program, export the LD PRELOAD environment variable again in the root account and run it.



Observation: it ran the custom sleep function implemented by the malicious library.

4. Make myprog a Set-UID user1 program (i.e., the owner is user1, which is another user account), export the LD PRELOAD environment variable again in a different user's account (not-root user) and run it.

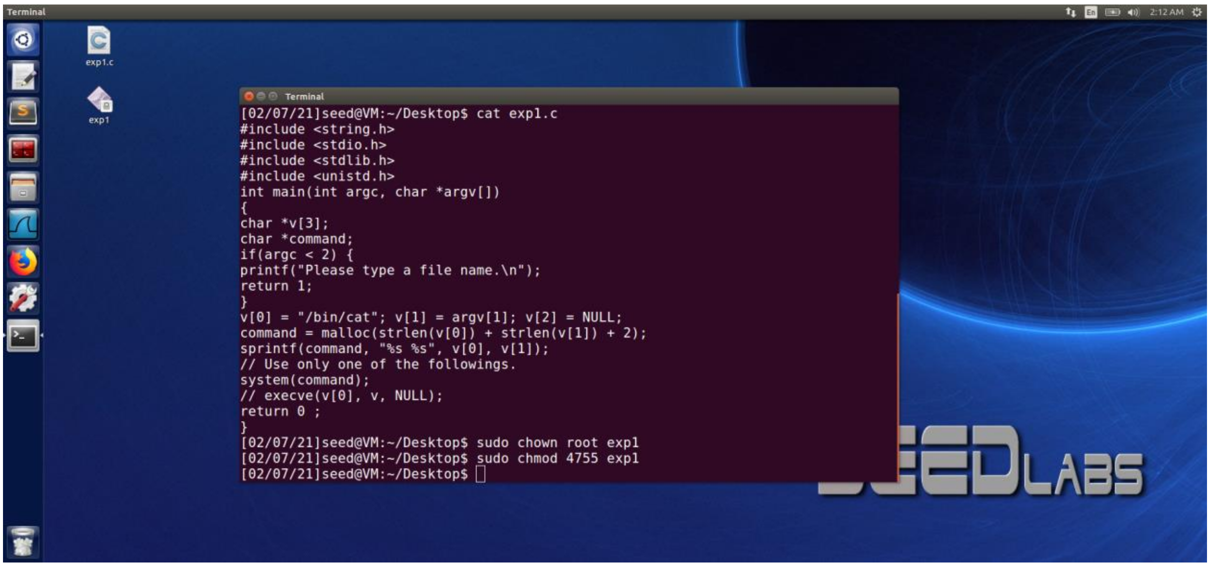


Observation: We can see that the malicious library created by us gets executed here.

2.8 (20 Points Total) Task 8: Invoking External Programs Using system() versus execve()

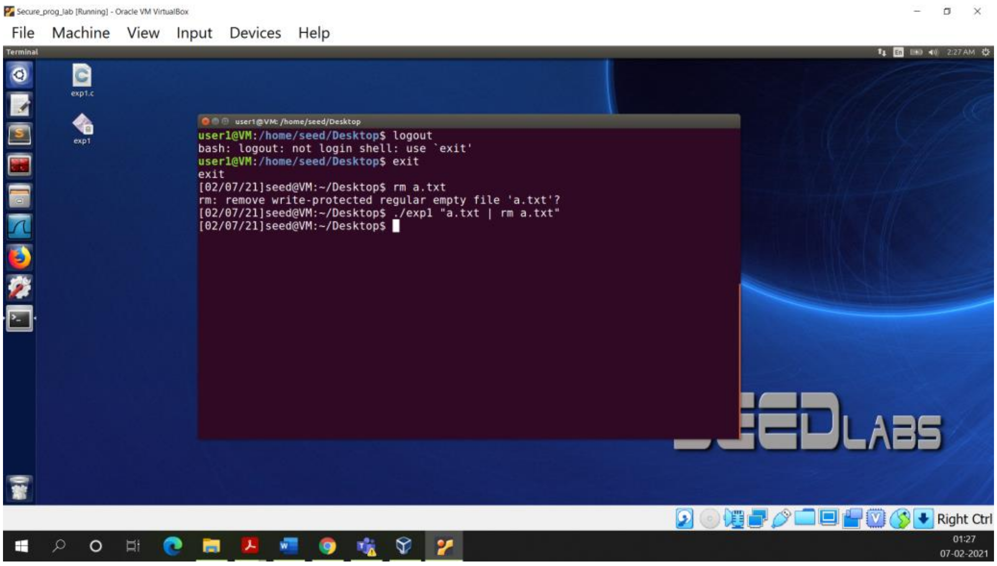
Step 1:

Executing the given code:



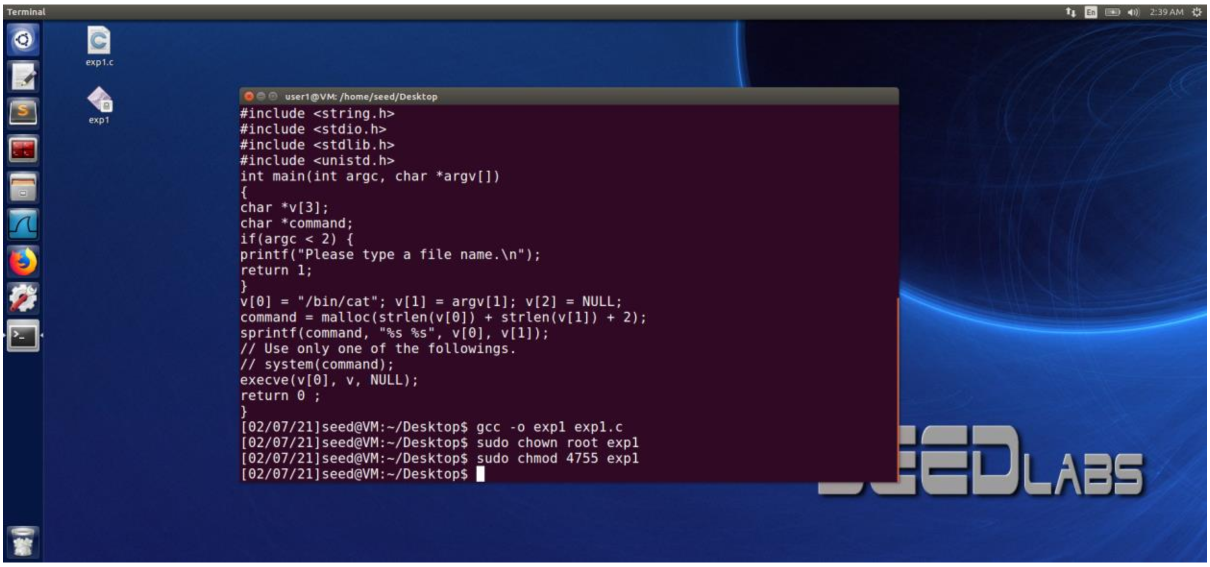
Observation:

I created a new file a.txt and changed its owner to root. Without running the program provided I was not allowed to remove the file. When the script to read the file is implemented using system, I was able to use the rm command to remove the file.



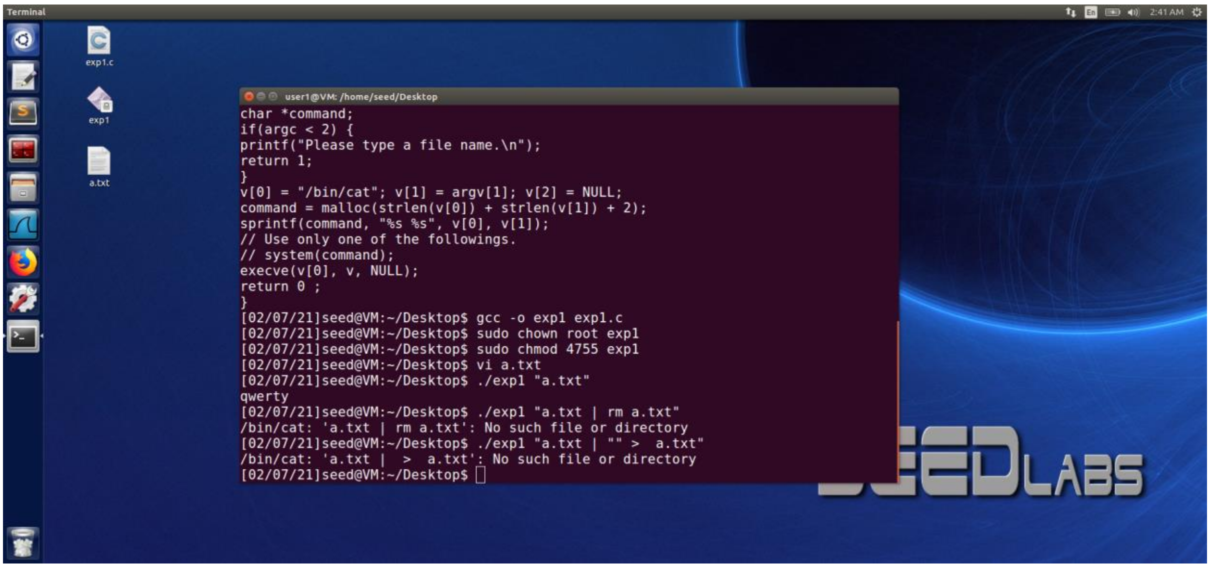
Step 2:

Compiling the file again after changes, along with set-UID to root.



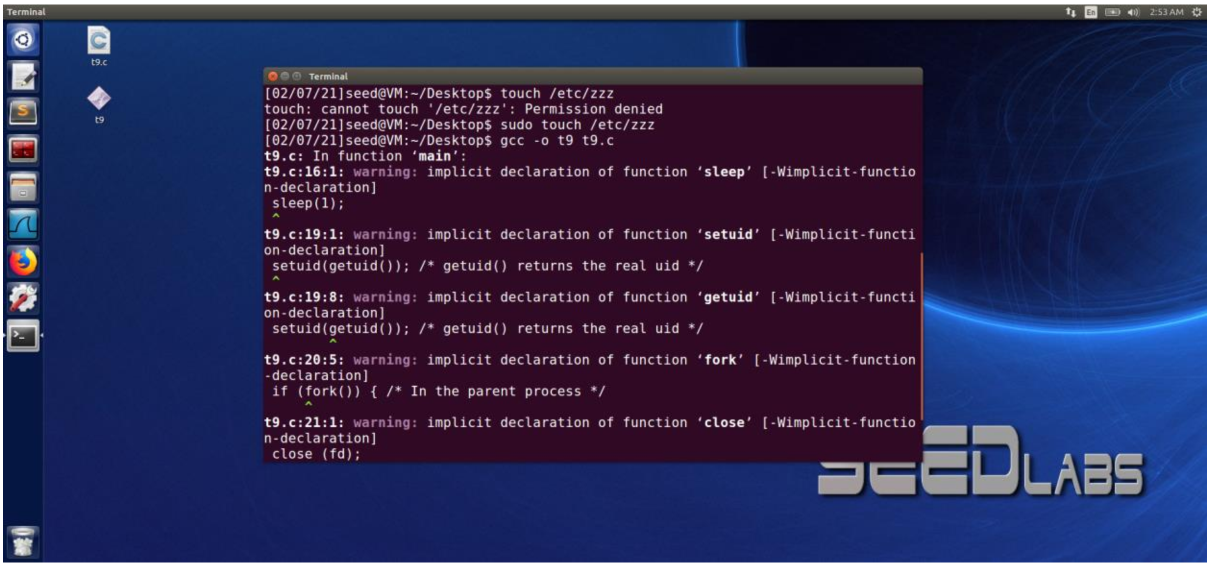
Observation:

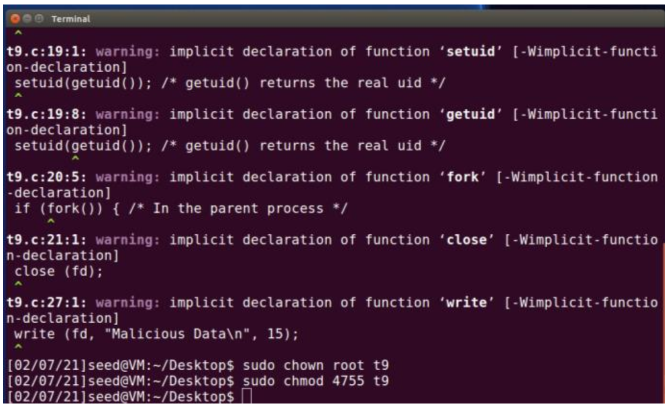
On executing the file reading script using the “execve” I was unable to remove the file nor perform any changes to it.



2.9 Task 9: Capability Leaking:

Creating the etc/zzz file and compiling the script.



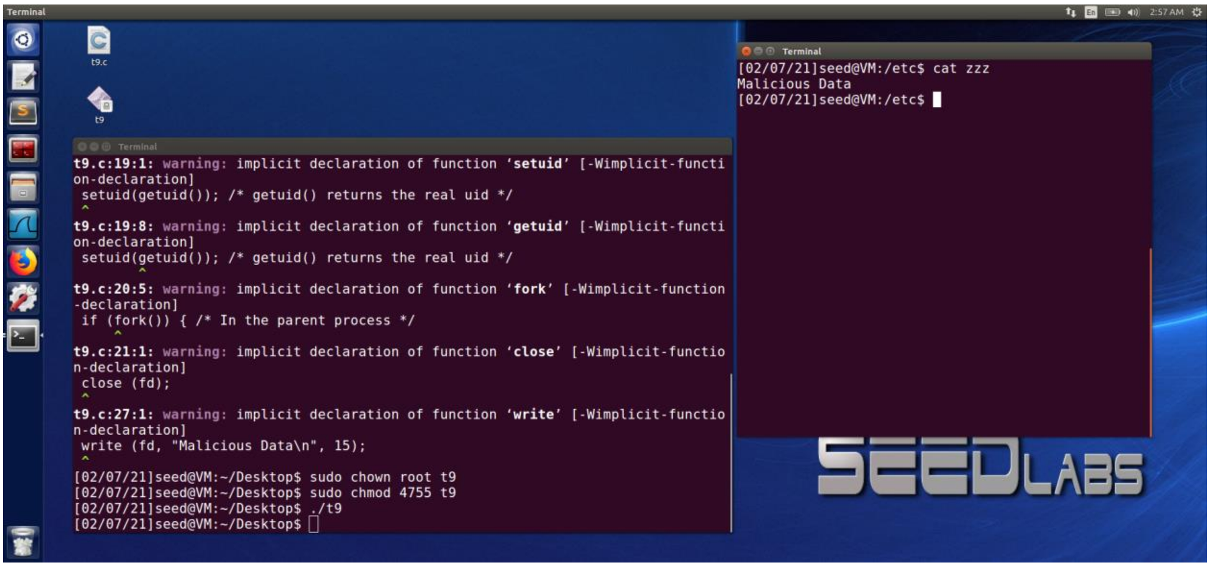


Observation:

From the below screenshot it is clear that the above script was able to modify the file “zzz”. Since the script is running with root privileges, the below highlighted line returns the root UID:

setuid(getuid());

Due to which the programs privileges are not reduced and post the process is forked the malicious code part executes with root privilege.


Terminal
2:57 AM
t9.c
t9
Terminal
t9.c:19:1: warning: implicit declaration of function ‘setuid’ [-Wimplicit-function-declaration]
setuid(getuid()); /* getuid() returns the real uid */
t9.c:19:8: warning: implicit declaration of function ‘getuid’ [-Wimplicit-function-declaration]
setuid(getuid()); /* getuid() returns the real uid */
t9.c:20:5: warning: implicit declaration of function ‘fork’ [-Wimplicit-function-declaration]
if (fork()) { /* In the parent process */
t9.c:21:1: warning: implicit declaration of function ‘close’ [-Wimplicit-function-declaration]
close (fd);
t9.c:27:1: warning: implicit declaration of function ‘write’ [-Wimplicit-function-declaration]
write (fd, "Malicious Data\n", 15);
[02/07/21]seed@VM:~/Desktop$ sudo chown root t9
[02/07/21]seed@VM:~/Desktop$ sudo chmod 4755 t9
[02/07/21]seed@VM:~/Desktop$ ./t9
[02/07/21]seed@VM:~/Desktop$
Terminal
[02/07/21]seed@VM:/etc$ cat zzz
Malicious Data
[02/07/21]seed@VM:/etc$
SEEDLABS