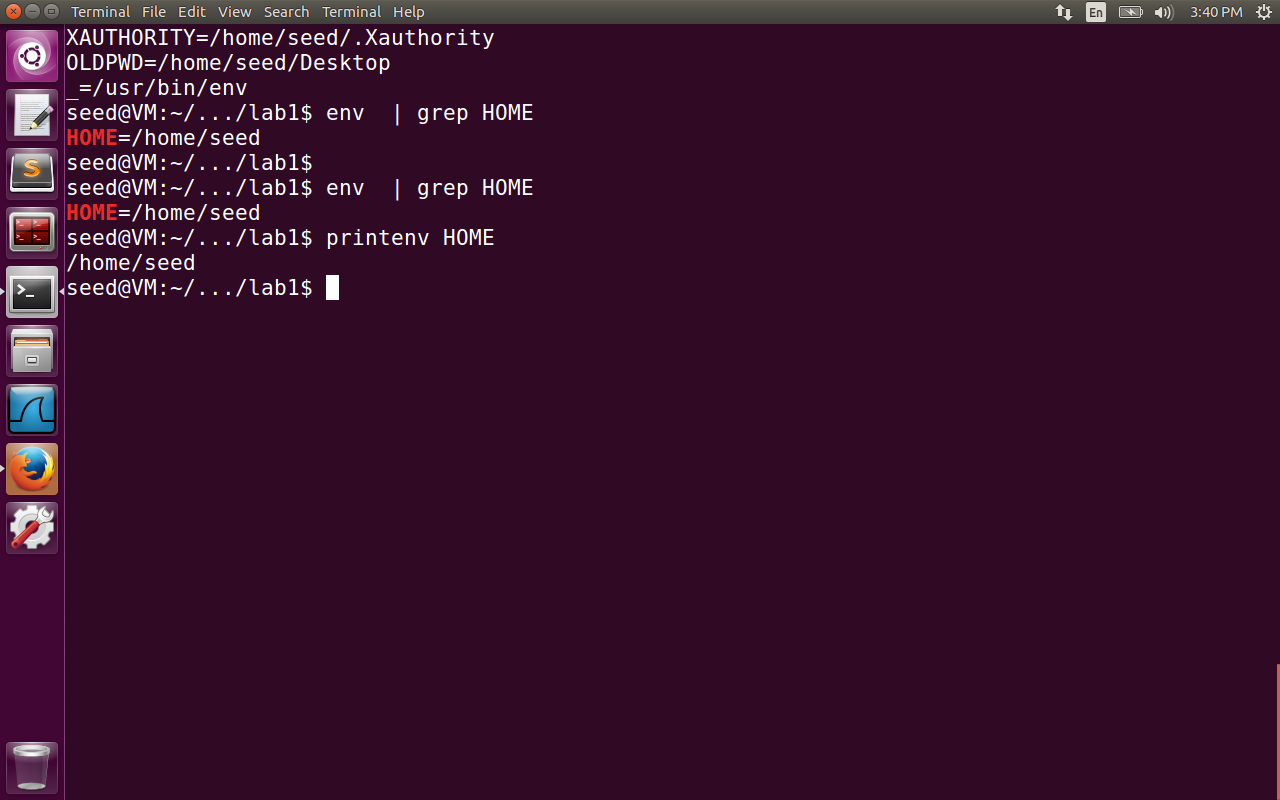
STUDENT NAME: JASHWANTH REDDY GANGULA

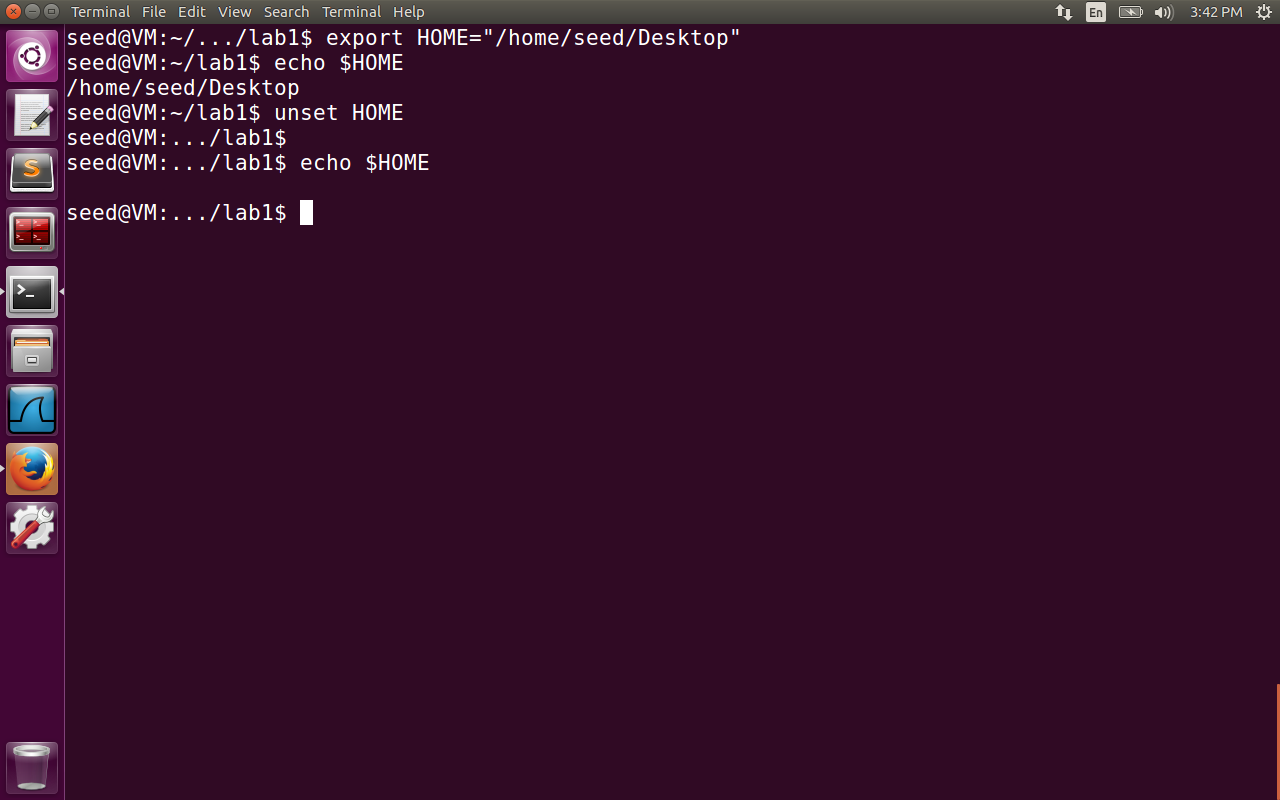
SUID: 646254141

**Task 1: Manipulating environment variables**

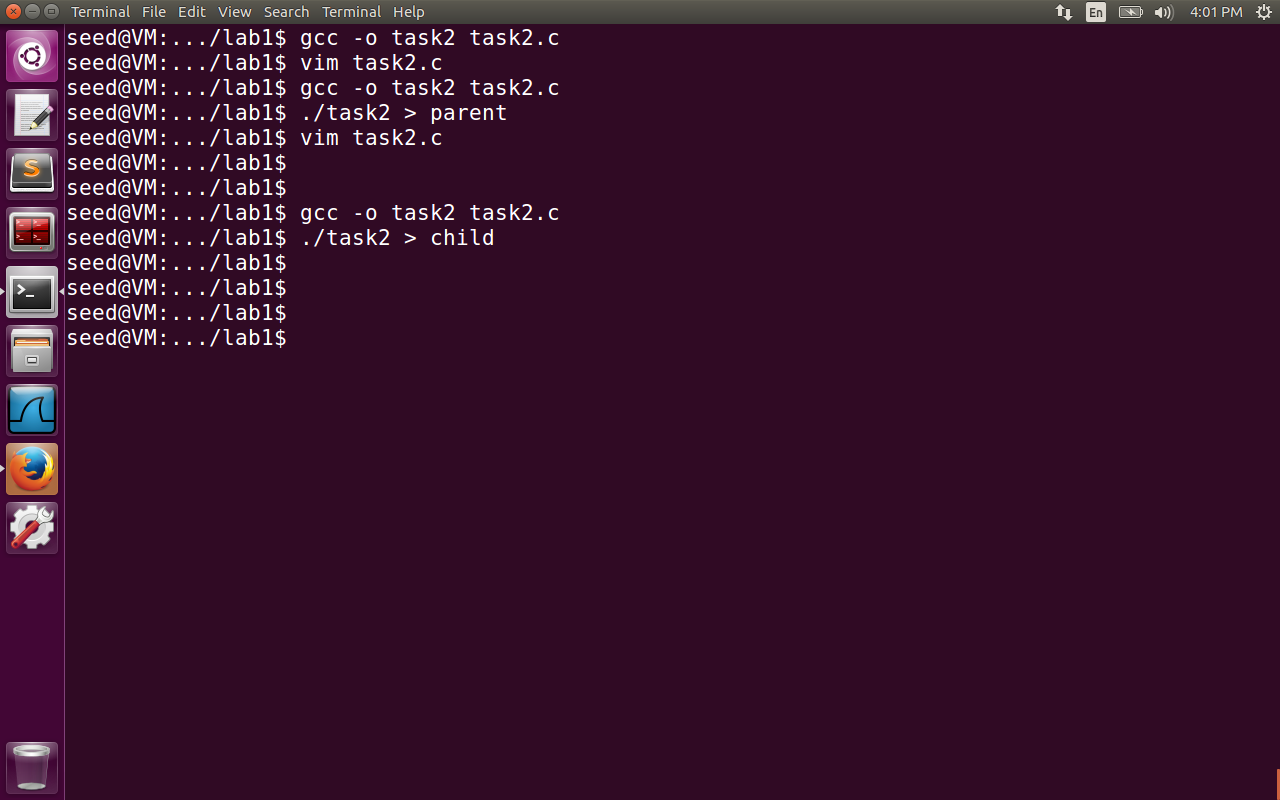
env is used below and grepped for find HOME environment variable.



export is used to change the value of HOME environment variable. Unset is used to unset the current value and echo is used to show the value after unset command is used.

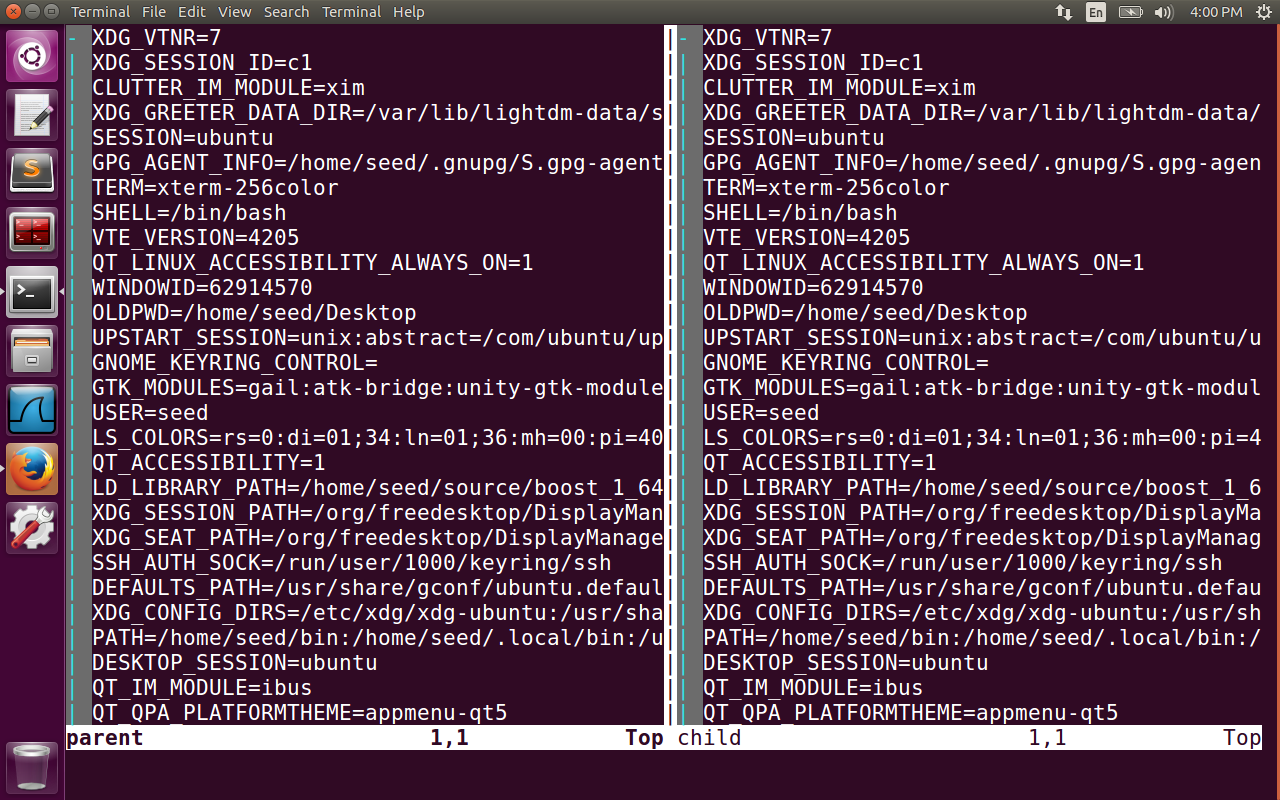


**Task 2: Inheriting environment variables from parents**

****

First the program is compiled for parent process by commenting out the printenv function for child process. Next the program is run for child process and commented for parent process. The outputs are saved in parent and child files.

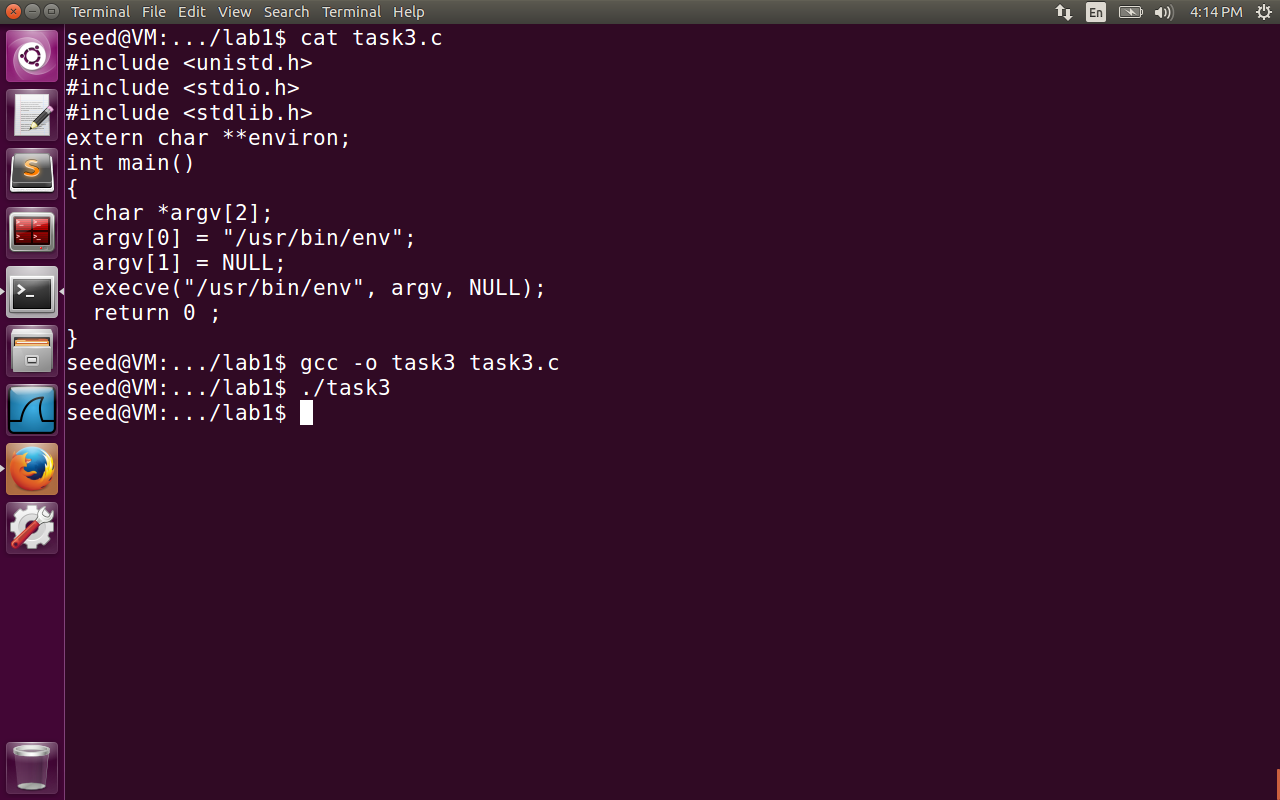
Vimdiff is performed between parent and child files for the environment variables.



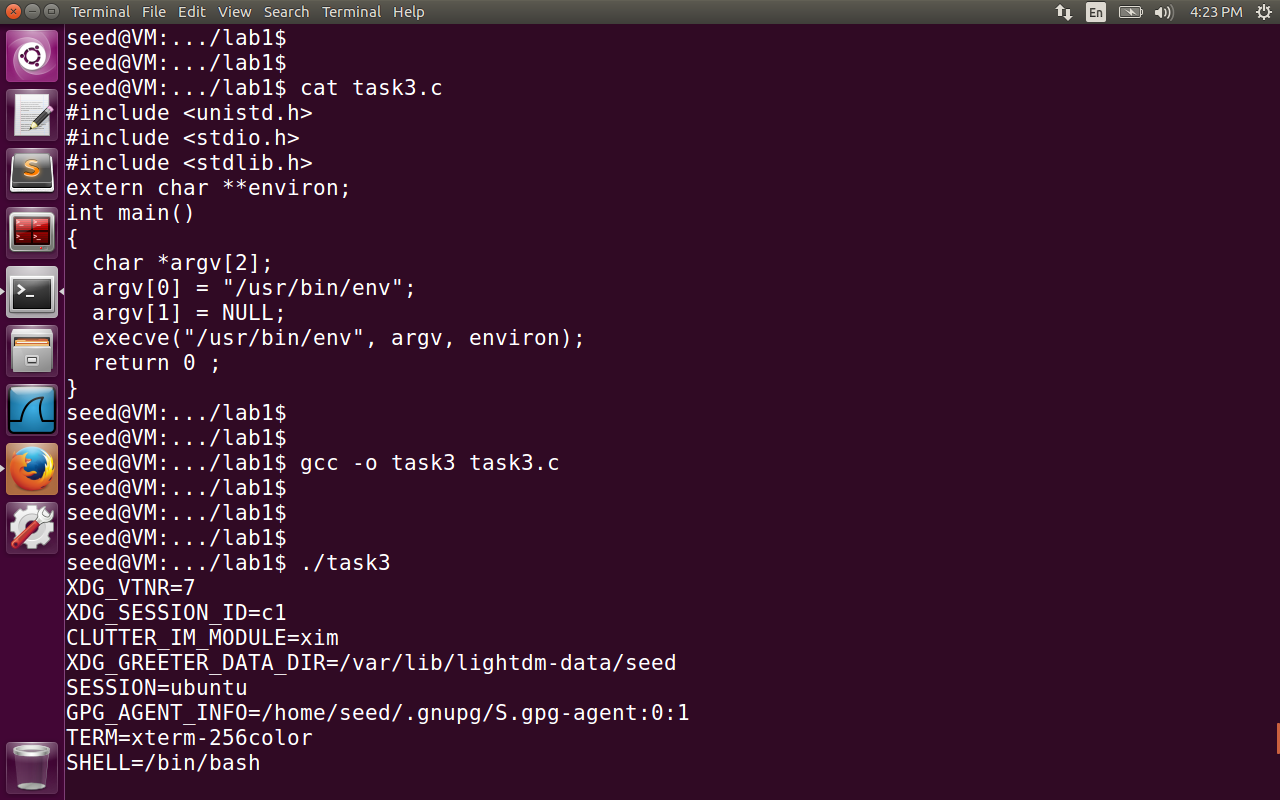
Observation: When fork is used to create child process from the parent process, it inherits all the environment variables. The diff command displays no difference between the two files.

**Task 3: Environment variables and execve()**

**Step1:**

****

**Step2:**

****

**Step3:**

**int execve(const char \****filename***, char \*const** *argv***[],**

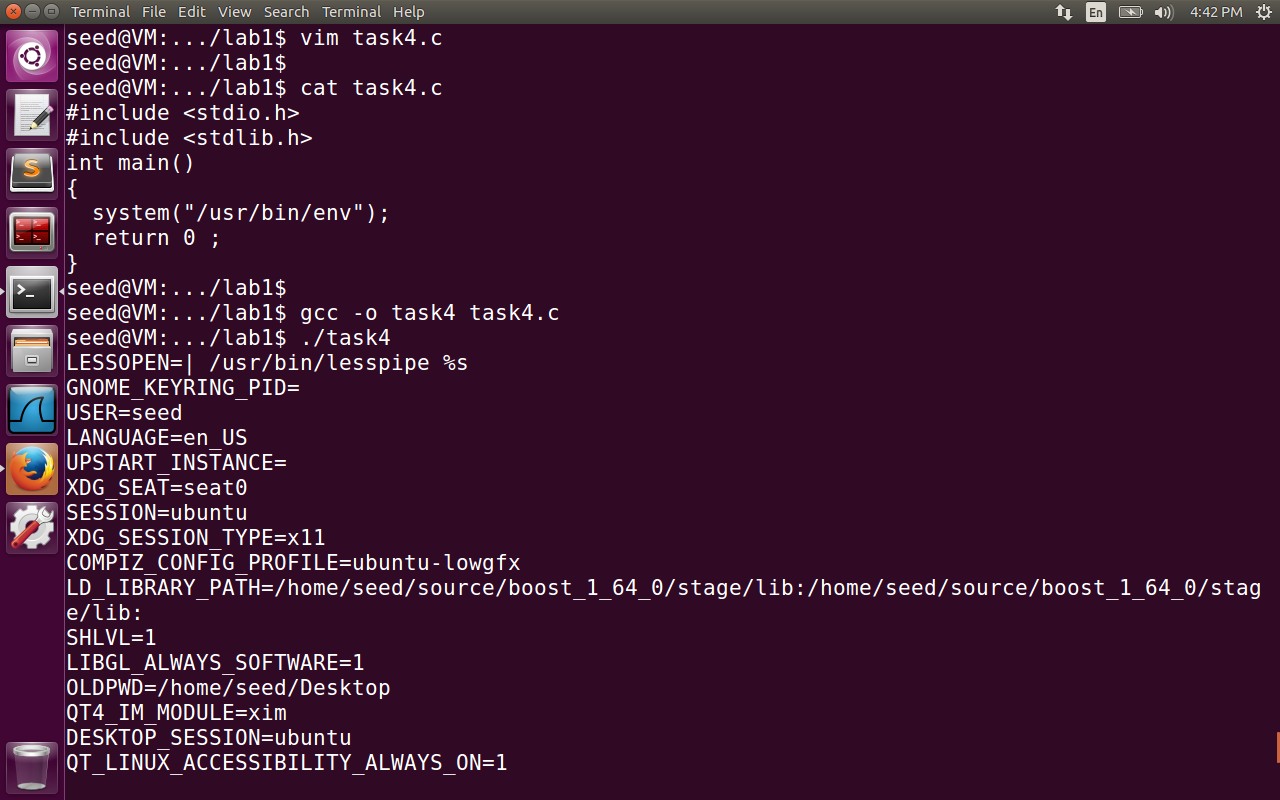
**char \*const** *envp***[]);**

The execve command takes argv[0] as the filename pointed to be executed. The last argument char \*const envp[]is an array of strings, conventionally of the form **key=value**, which are

passed as environment to the new program.

In step1 NULL is passed as argument for envp[] , in step2 environ variable stores all the array of strings pointed by env variable. Hence the strings are printed for step2.

**Task 4: Environment variables and system ()**

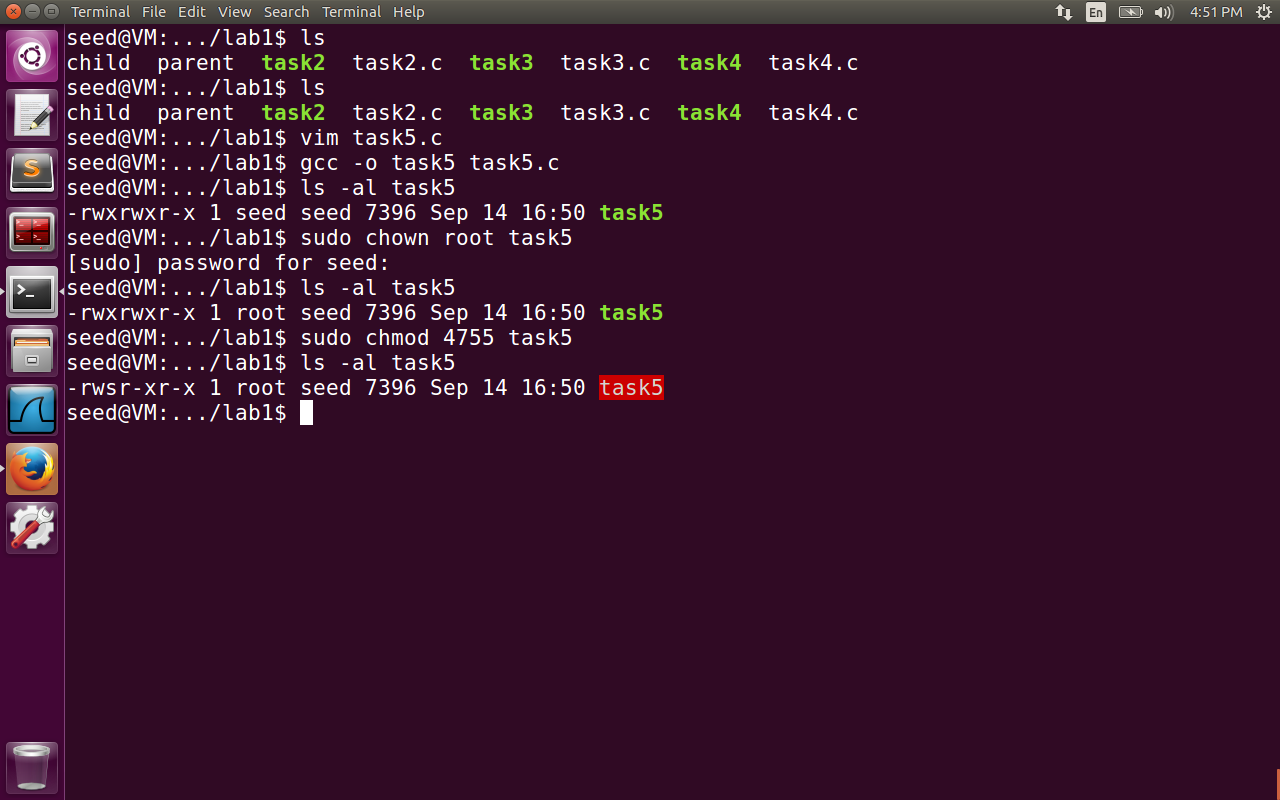
****

**Observation:** In case of system() command the environment variables are passed from the program to /bin/sh. The next command is executed by execl() and passes the environment variables to the execv function. Hence all the environment variables are printed when the system() command is used. Observe the output in the image above.

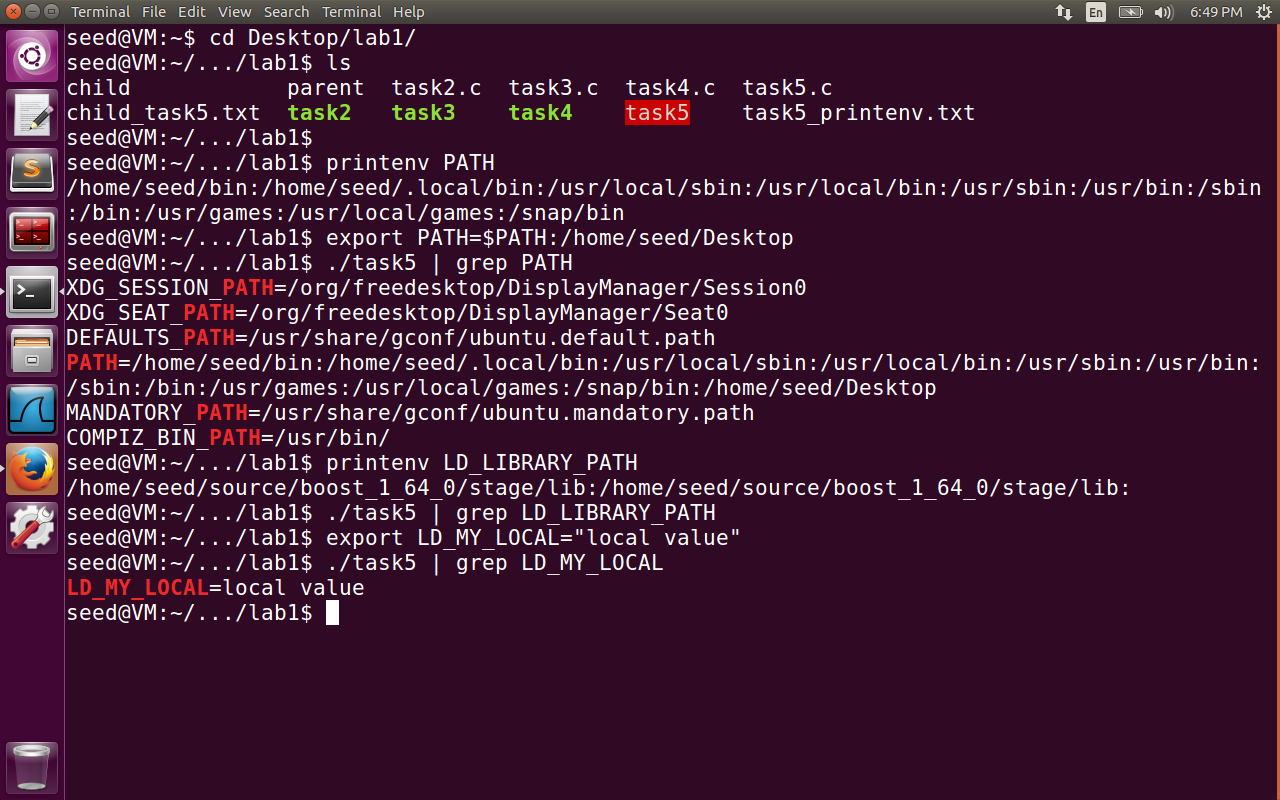
**Task 5: Environment variable and Set-UID Programs**

**Step1,2:**

Created task5.c file, compiled it, changed the ownership to root and made it a SETUID program.

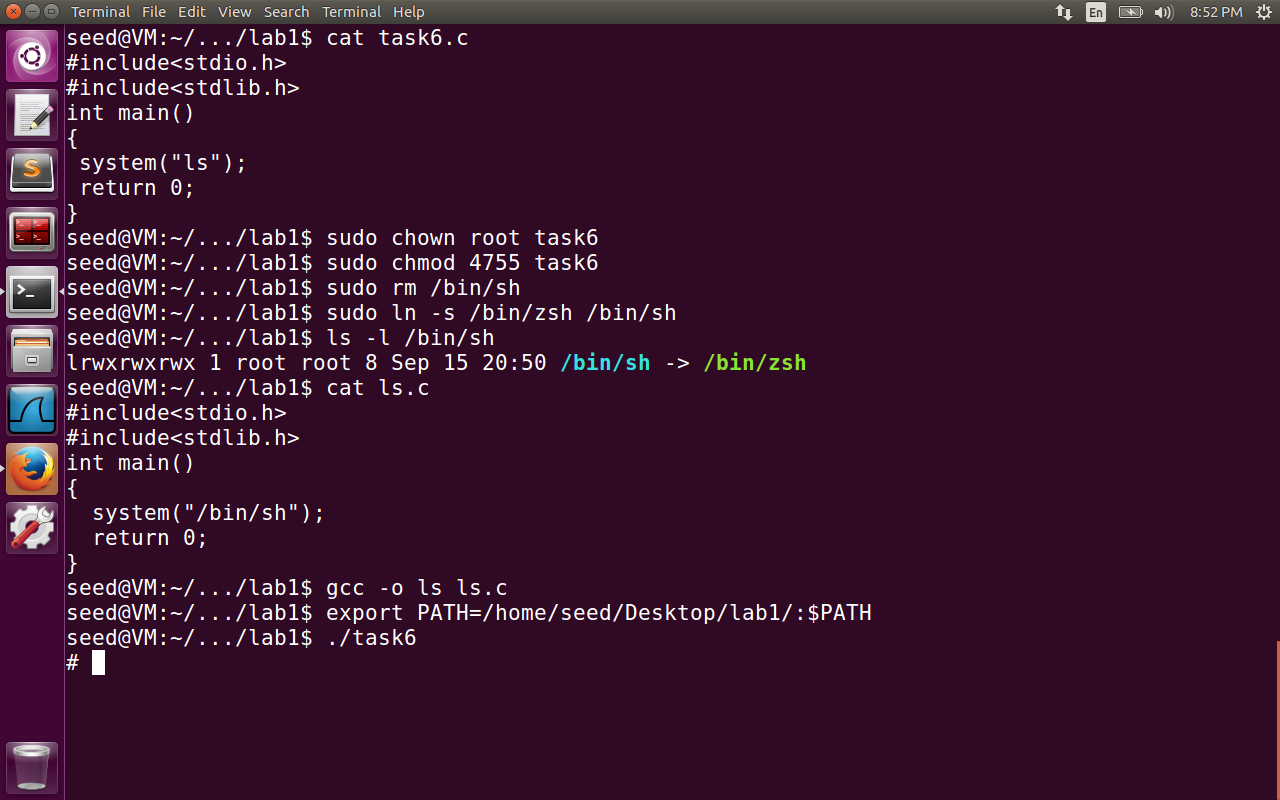


Step3:



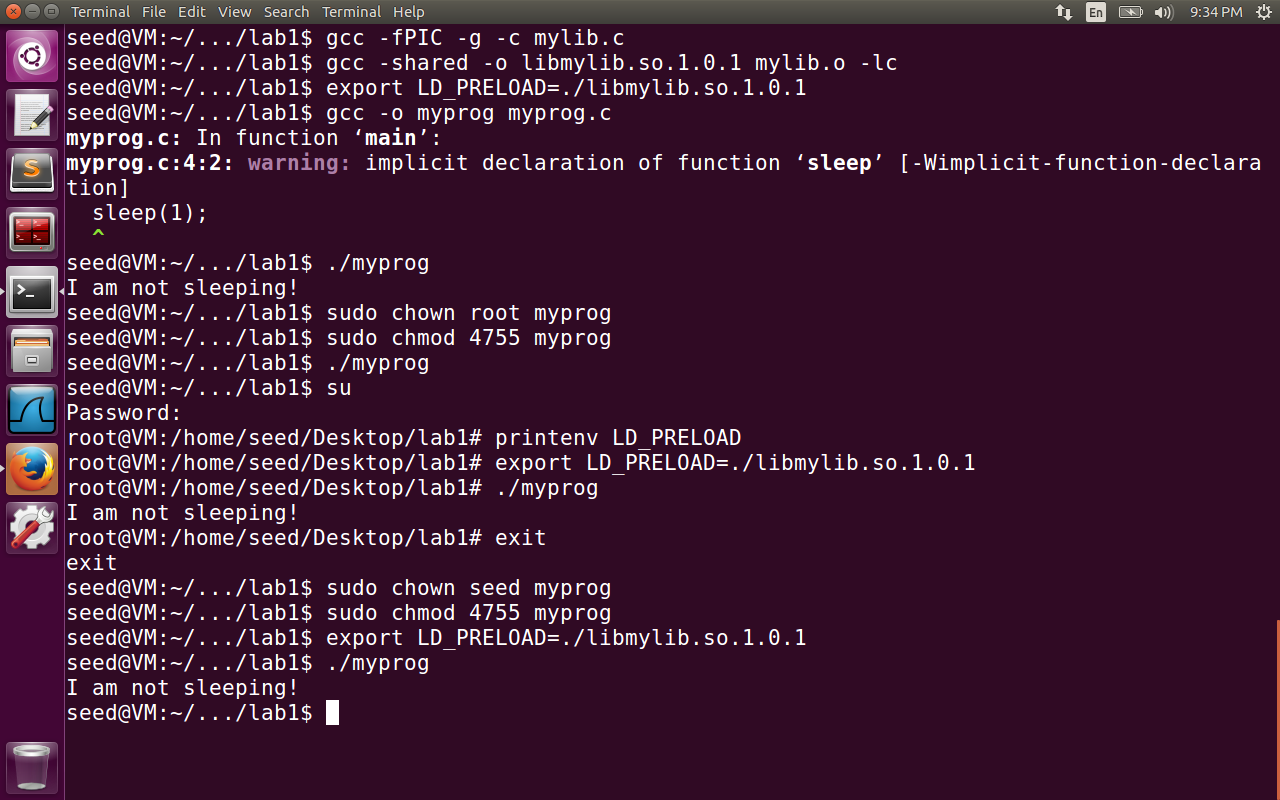
The PATH variable and LD\_MY\_LOCAL environment variables are inherited from the user changes to the shell. However, to my surprise found that LD\_LIBRARY\_PATH is set in the shell, but it is not available in the list when the taks5 executable is run. LD\_LIBRARY\_PATH variable is only used during the time of linking the dynamic libraries. It is not consulted during link time, hence it is not listed when the task5 executable is run.

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



In the following task6.c, the local ls.c file is created to access the root privilege. All the following command after executing ./task6 will be executed as root. This is a serious vulnerability because we have modified PATH variable to execute the local ls command present in the directory instead of /usr/bin/ls command. The setuid program is effected because /bin/sh is pointing to /bin/zsh instead of /bin/bash. The bash vulnerability is fixed in Ubuntu 16.04 version hence used the zsh shell to expose the vulnerability. The system(command) is equivalent to   
execl(“/bin/sh”, “sh”, “-c”, command, (char \*)0); The conclusion is to never use system() command in setuid programs because it exposes the shell as ‘middle person’. The shell surface exposes larger surface for vulnerable attacks. Hence it is advised to use execv() which is directly executed by the OS.

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

****

LD\_PRELOAD as the name implies includes a list of additional, user-specified, ELF shared objects to be loaded before all others.

The following task7 involved four activites.

1)myprog is normal program run under normal user – The sleep program is loaded from local shared folder because of exporting LD\_PRELOAD.

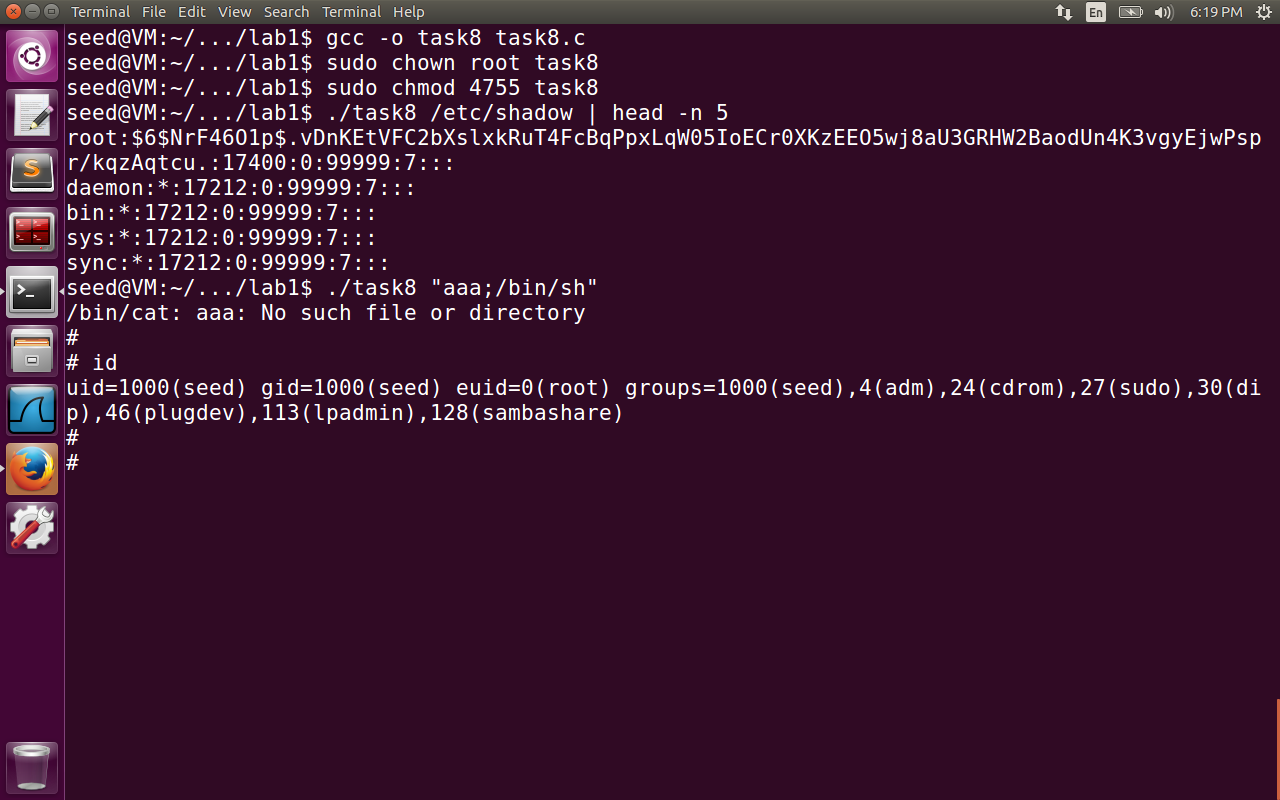
2)made myprog a setuid program and ran it as normal user. In this case the myprog is forked by the shell program. But since it is a setuid program the cpu will make sure the child process (i.e., task7 is child process for shell process) doesnot inherit the environment variables of the form LD\_\*. Hence the program is not vulnerable.

3)In third case we logged into root account and changed the actual value of LD\_PRELOAD variable. Hence the local library is executed as expected.

4)Even for normal user, the setuid program is set with normal user, there is no mismatch between effective user id and running user id. Hence LD\_\* will not be ignored, and will run the local shared library code.

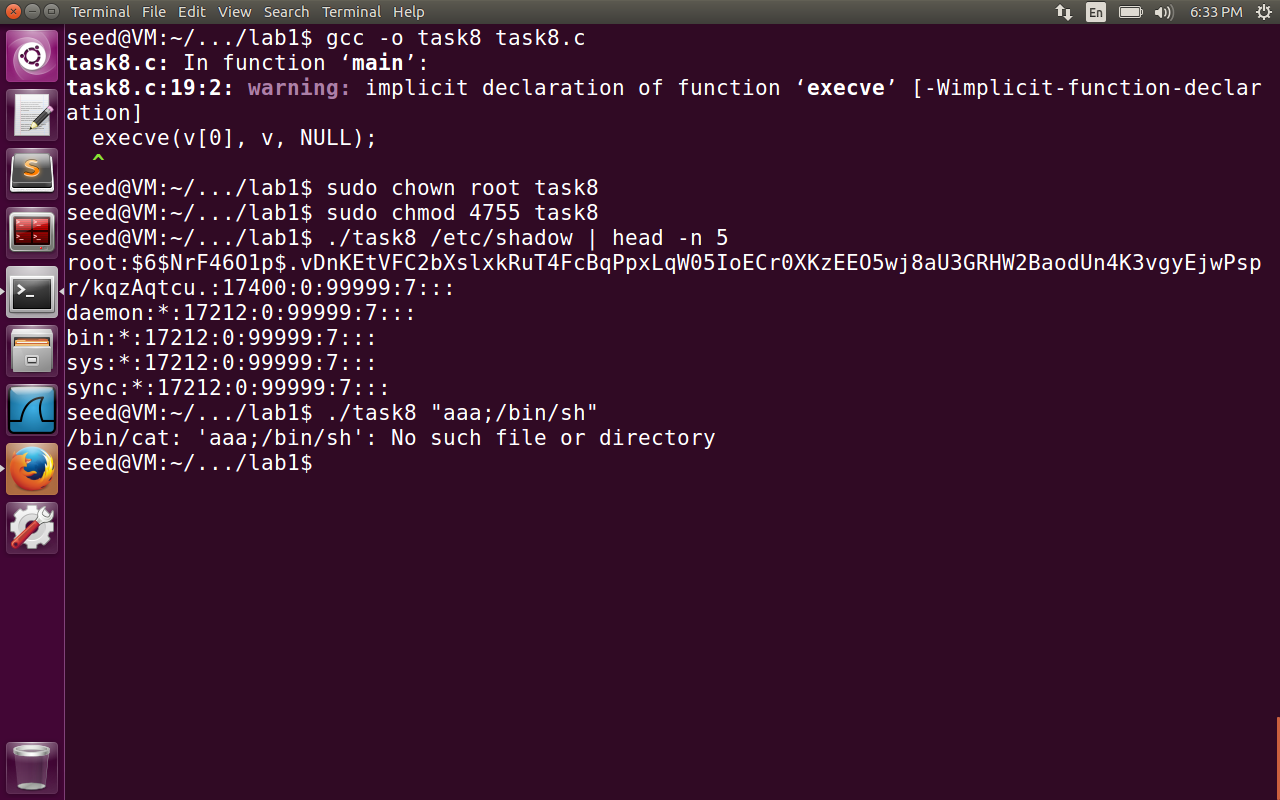
**Task 8: Invoking external programs using system() versus execve()**

**Step1:**

****

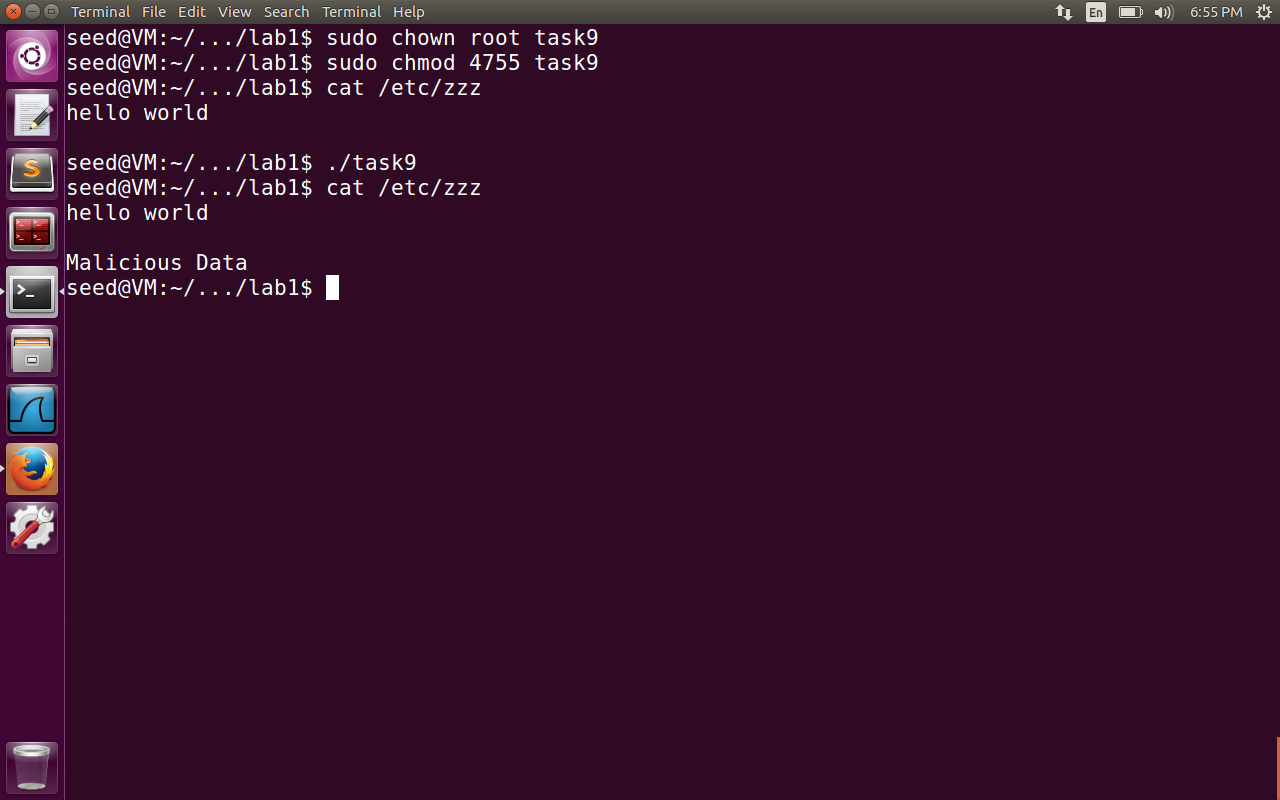
Here the step1 is executed using the system command. We observed that the user will get root privileges and hence can edit any file even though this is just a cat program to display the information in the file. This type of attack is explicitly through user input, where the input is not properly checked. For example, in the diagram the user input given is “aaa;/bin/sh”. There are two shell commands. First one is /bin/cat aa (executed by the shell) for which the output is file does not exist. The second command executed by the shell gives the user root privilege and hence modify any file he wants or delete the files he wants to.

Step2:



In the second step, we are using execve command instead of system(). So, it doesnot invoke the shell program as the mediator to execute the cat command. Hence the entire input is taken as the filename and hence it is not vulnerable. It is advised to use execve command for setuid programs, because if we invoke system() command , it in turn invokes shell program which increases the vulnerable surface.

**Task 9: Capability Leaking**



In the task9 , we have opened the file “/etc/zzz” with the root privileges but did not close the file descriptor. By default, fd will be 3, because (0,1,2 are reserved for std input, output and error files). So even the user can guess the file descriptor in case it is not closed. After revoking the privilege, the capability leaking here is not closing the file descriptor. To demonstrate this effect, the fork () command is used to show that the child process will inherit the opened file descriptor and adds the malicious content to the file, even after the parent process completed its execution by closing the file descriptor.