Task 1: Manipulating Environment Variables

Before starting, in order to check for the default shell configured for my account, I executed:

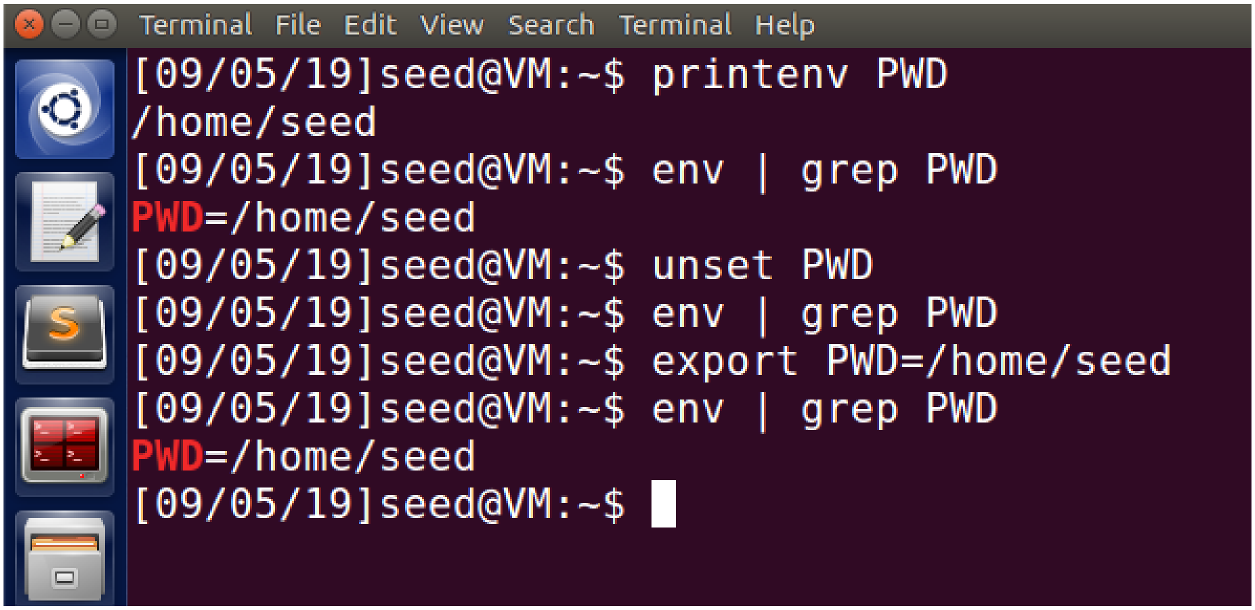
seed@VM:~$ cat /etc/passwd

Looking for my account, the default shell is bash, as seen in the output's last field:

seed:x:1000:1000:seed,,,:/home/seed:/bin/bash

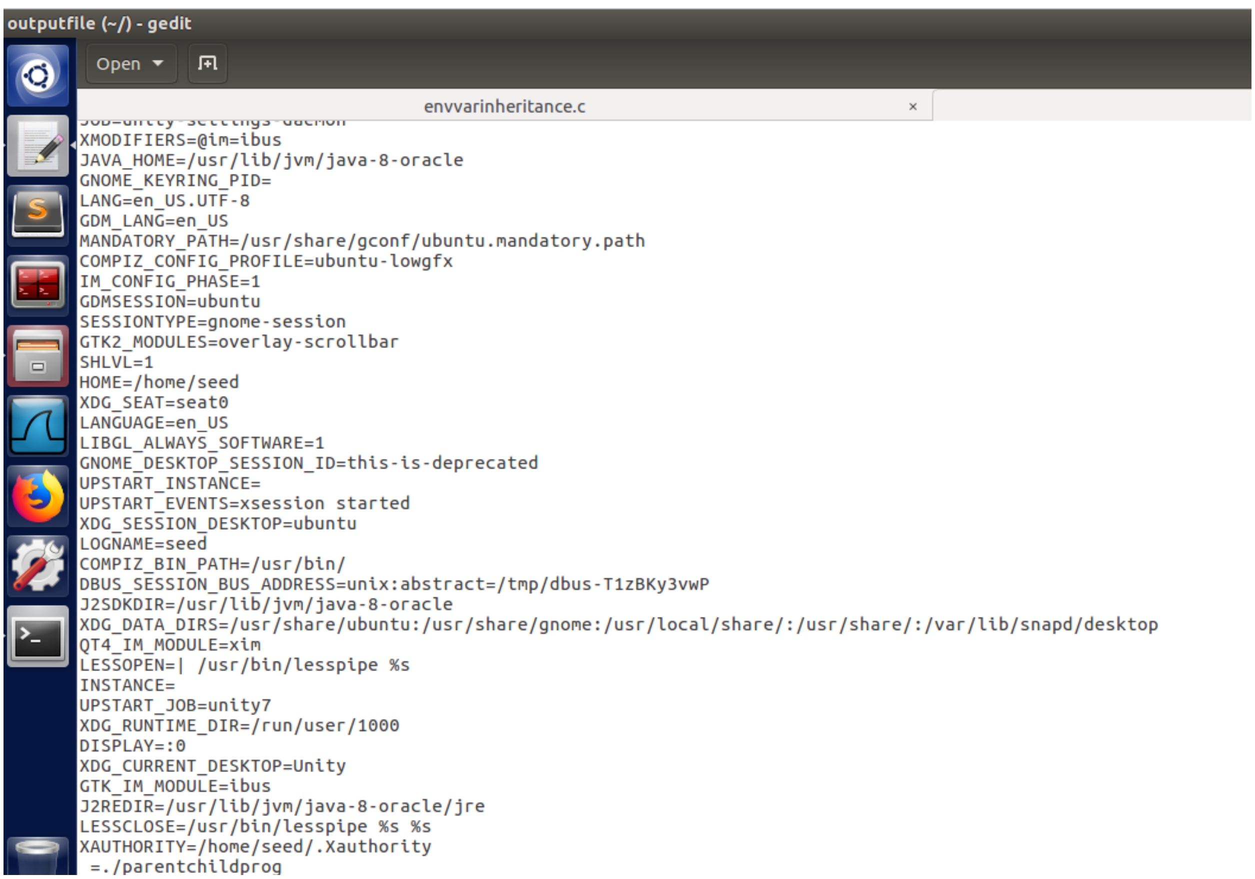
In order to print all environment variables, I type printenv. The output shows that environment variables are just variable = value pairs. Entering env would give a similar output:

Next, I type in printenv PWD, which returns only the value of the variable PWD. In order to find out all the variables that consist of a substring PWD, I use env | grep PWD, which gets all the variables and values that contains PWD as a substring within them. The output is in the format of variable = value. The unset command helps to delete a particular environment variable, as is seen in the output. Once we unset PWD and then try to find it using env command, it returns nothing because there is no variable PWD. Using the export command, we can set the environment variable and value, as seen in the output. This command can be used to create or edit a particular environment variable. The output shows a demo of these commands:



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

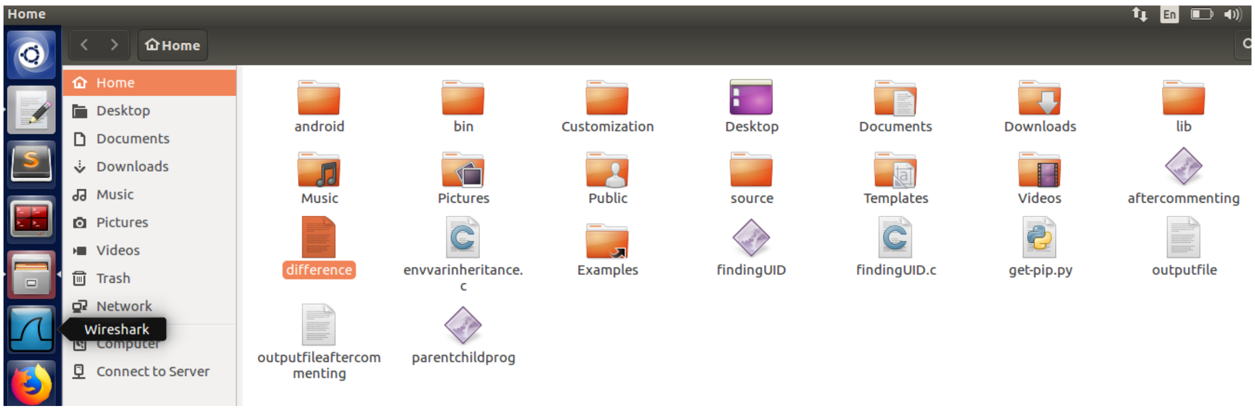
The content of the output of parentchildprog containing child process with printenv is stored in file named outputfile. It displays all the environment variables of the child process.



The parent process with printenv command displays a similar output.

The following files are created as a result of this task:

envvarinheritance.c	→	The program of the task
parentchildprog	→	The executable file with child process printenv()
aftercommenting	→	The executable file with parent process printenv()
outputfile	→	Output from the parentchildprog file
outputfileaftercommenting	→	Output from the aftercommenting file
difference	→	Output of the diff command consisting of file differences



The following commands were used to compile and run the programs, storing the output in the respective files. The diff command helps in finding the difference between the two output files:

```
[09/05/19]seed@VM:~$ man fork
[09/05/19]seed@VM:~$ gcc envvarinheritance.c -o parentchildprog
[09/05/19]seed@VM:~$ ./parentchildprog > outputfile
[09/05/19]seed@VM:~$ gcc envvarinheritance.c -o aftercommenting
[09/05/19]seed@VM:~$ ./aftercommenting > outputfileaftercommenting
[09/05/19]seed@VM:~$ diff outputfile outputfileaftercommenting | tee difference
70c70
< _=./parentchildprog
---
> _=./aftercommenting
[09/05/19]seed@VM:~$ 
```

The diff command's syntax is as following:

diff file1 (left file) file2 (right file) | option (tee – in order to write the output to both the standard output and one or more files) filename

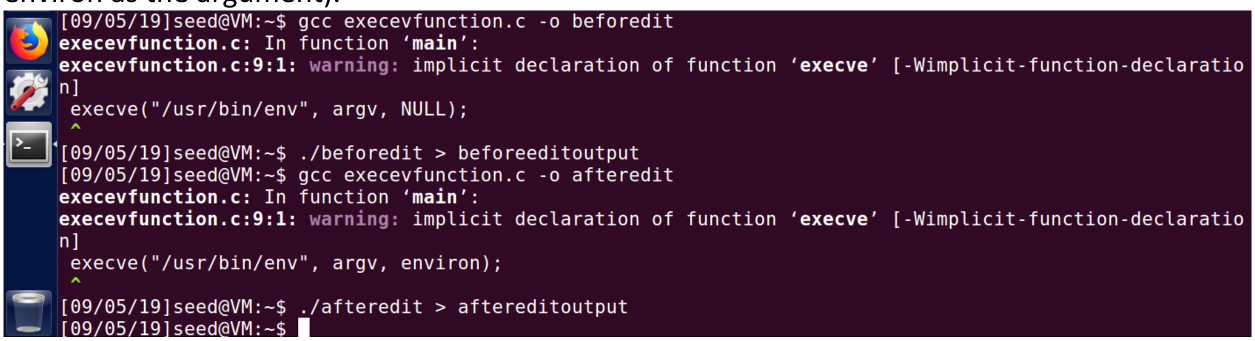
The output is interpreted as follows:
70c70 means that in the 70th line (left) in left file is changed to the 70th line (right) in the right file, where c stands for changing and the left and right numbers indicate the line number.
The < denotes lines in the left file and > indicates in the right file showing the changed content.

This shows that the _ environment variable takes on the value of the last command executed, here the command of program execution. It is considered a special shell variable and contains different values depending on the scenario.

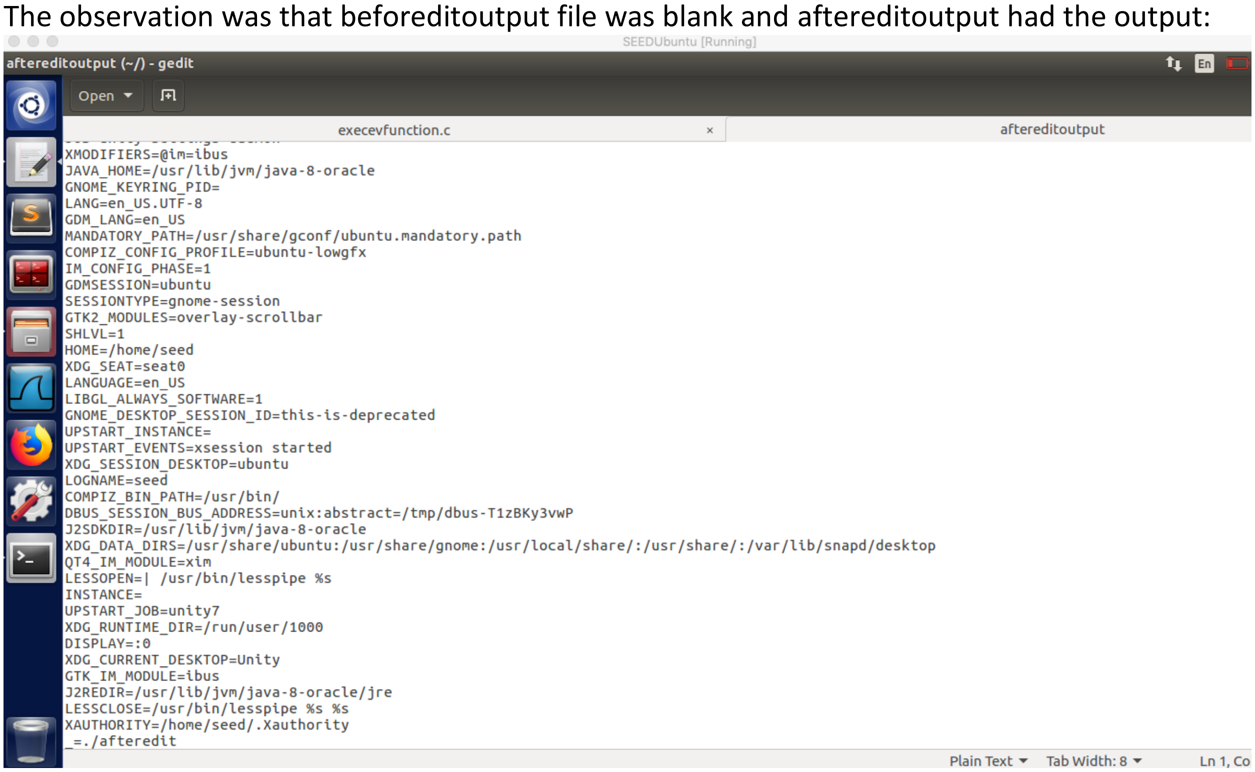
This shows that the _ environment variable changed depending on the compiled program being run but other than that there is no change in the environment variables. If both the programs were compiled into a file with the same name, there would not be any difference between the output of the parent and child process.

Task 3: Environment Variables and execve ()

Here, as seen, the Task 3 program is compiled and executed into respective output files and the output is stored in beforeeditoutput (with NULL as the argument) and aftereditoutput (with environ as the argument).



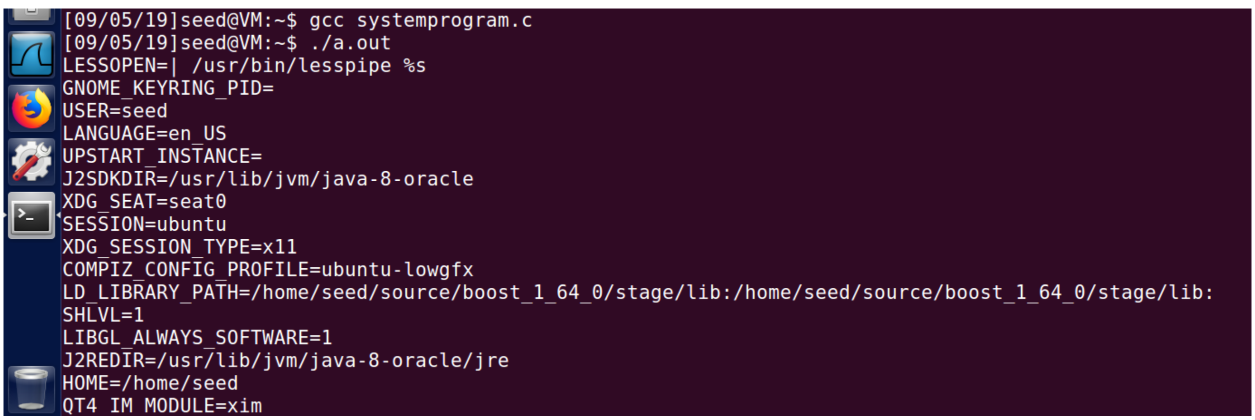
The observation was that beforeditoutput file was blank and aftereditoutput had the output:



The explanation for this is that even though the global environ variable was specified in the program, the beforeedit program contained NULL as the third argument of the execve and the afteredit program contained environ variable as the third argument of the execve. This change affected the output of the program because the third argument to execve() function specifies the environment variable of the current process. Since the environ variable was not passed in the initial program and hence no environment variables were associated with this new process, the output was null. But after editing the program, we passed the environ variable as the third argument to execve, which contained all the environment variables of the current process, the output of the program had all the environment variables, as expected. In conclusion, the third argument of the execve() command gets the program its environment variables.

Task 4: Environment Variables and system ()

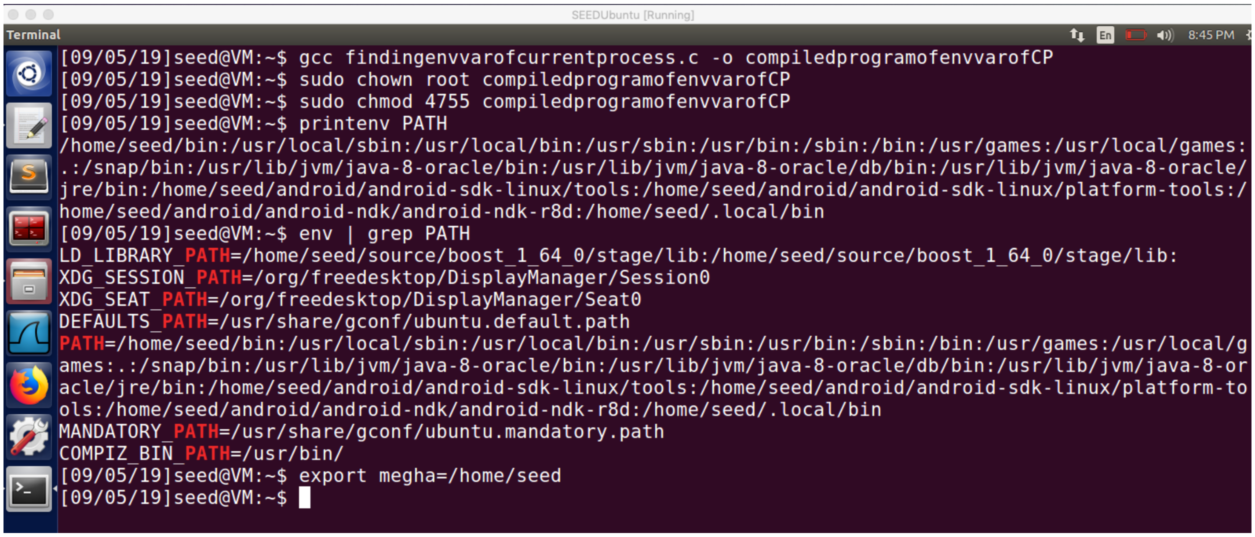
The program is compiled and executed and as seen, even though we don't explicitly send any environment variables in the program, the output shows the environment variable of the current process. This happens because the system function implicitly passes the environment variables to the called function /bin/sh.



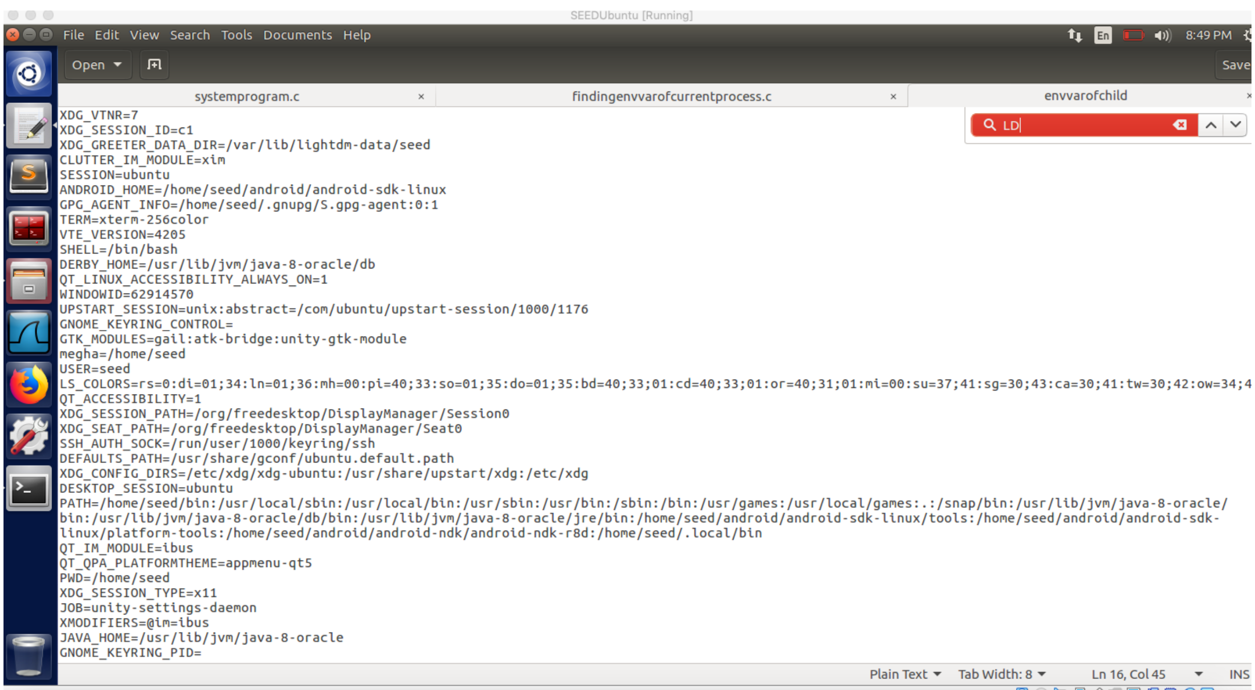
Task 5: Environment Variable and Set-UID Programs

After compiling the given program, we change the ownership and permission of the file using the following commands:
sudo chown root filename (making the root as the owner of filename)
sudo chmod 4755 filename (making the program a SET-UID program by setting set-uid bit)

This makes the program a SET-UID root program. Then on looking for the environment variables, since PATH and LD_LIBRARY_PATH are already present, I only initialize a new variable with name megha and value /home/seed using export command and allow the other environment values to be the same. The following screenshot shows the performed steps:



On running the above compiled program and storing the output in a file named envvarofchild, it's seen that the child process inherits the PATH and megha environment variable but there is no LD environment variable, as can be seen in the screenshot (on searching for LD in the file, it does not return any values):



This shows that the SET-UID program's child process may not inherit all the environment variables of the parent process, LD_LIBRARY_PATH being one of them over here. 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

The given program is written in the file named task6.c and compiled into task6compiled file. Then the compiled program's owner is changed into root and its converted into a SET-UID program. Then, we check the current value of the environment variable PATH and also the working directory of the program. The following screenshot shows these tasks:

```
[09/05/19]seed@VM:~$ gcc task6.c -o task6compiled
task6.c: In function ‘main’:
task6.c:3:1: warning: implicit declaration of function ‘system’ [-Wimplicit-function-declaration]
 system("ls");
 ^
[09/05/19]seed@VM:~$ sudo chown root task6compiled
[09/05/19]seed@VM:~$ sudo chmod 4755 task6compiled
[09/05/19]seed@VM:~$ env | grep PATH
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost_1_64_0/stage/lib:
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
PATH=/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/g
ames:.:/snap/bin:/usr/lib/jvm/java-8-oracle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-or
acle/jre/bin:/home/seed/android/android-sdk-linux/tools:/home/seed/android/android-sdk-linux/platform-to
ols:/home/seed/android/android-ndk/android-ndk-r8d:/home/seed/.local/bin
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
COMPIZ_BIN_PATH=/usr/bin/
[09/05/19]seed@VM:~$ pwd
/home/seed
```

Confirming that task6compiled is a SET-UID program with root as the owner:

```
[09/06/19]seed@VM:~$ ll task6compiled
-rwsr-xr-x 1 root seed 7348 Sep  6 10:00 task6compiled
[09/06/19]seed@VM:~$
```

The following shows the contents of my ls program created in the Task6 folder:

```
ls.c (~/Task6) - gedit
Open

#include <stdio.h>

void main()
{
printf("This is Megha Jakhotia\n");
}
```

In order to run my program instead of the standard ls program, I changed the value of environment variable PATH and provided the path to my file as the first value of the variable. This makes the program to search for the file in my directory first before any other directory and since I have the file with the same name as ls, the current program will execute my program.
The screenshot shows that I have changed the value of the PATH variable:

```
[09/05/19]seed@VM:~$ export PATH=/home/seed/Task6:$PATH
[09/05/19]seed@VM:~$ env | grep PATH
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost
_1_64_0/stage/lib:
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
PATH=/home/seed/Task6:/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/u
sr/bin:/sbin:/bin:/usr/games:/usr/local/games:.:/snap/bin:/usr/lib/jvm/java-8-or
acle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-oracle/jre/bin:/h
 Firefox Web Browser oid/android-sdk-linux/tools:/home/seed/android/android-sdk-linux/pl
atform-tools:/home/seed/android/android-ndk/android-ndk-r8d:/home/seed/.local/bi
n
```

The following shows that I have compiled my program with the filename ls. On running the task6compiled, it runs my ls program instead of the system's ls:

```
[09/05/19]seed@VM:~$ cd Task6
[09/05/19]seed@VM:~/Task6$ gcc ls.c -o ls
[09/05/19]seed@VM:~/Task6$ cd ..
[09/05/19]seed@VM:~$ ./task6compiled
This is Megha Jakhotia
[09/05/19]seed@VM:~$
```

The following show that the Task6 folder consists of the ls program with normal user privileges:

```
[09/06/19]seed@VM:~$ cd Task6
[09/06/19]seed@VM:~/Task6$ ls -l
total 12
-rwxrwxr-x 1 seed seed 7340 Sep  6 09:48 ls
-rw-rw-r-- 1 seed seed   72 Sep  5 22:13 ls.c
```

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 priveleges 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.

In order to overcome the dash shell security mechanism of dropping SET-UID program's privileges on being called from one, we link the /bin/sh to another shell:

```
[09/06/19]seed@VM:~$ sudo rm /bin/sh
[09/06/19]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
```

Task 7: The LD_PRELOAD Environment Variable and Set-UID Programs

First, I create a program named mylib.c that has the sleep function overriding the system's sleep function as given in the assignment. This function is just printing a statement on the standard output. After this, we compile the program using the following command:

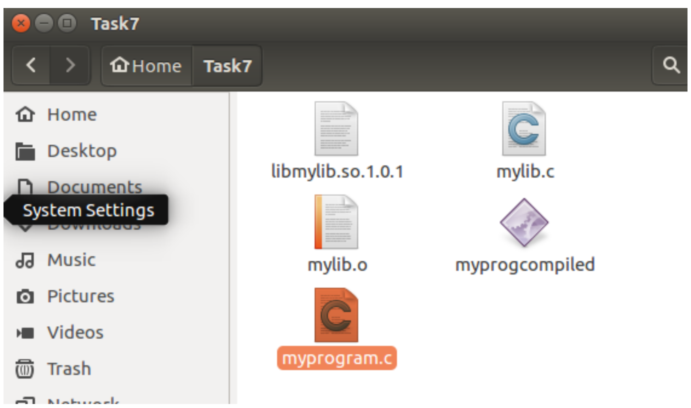
gcc -fPIC -g -c mylib.c (where -fPIC means that emit position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table, -g means producing debugging information and -c means compiling the file but not linking it.)

gcc -shared -o filename mylib.o -lc (where -shared produces a shared object that can be linked to other objects to form a executable, -o file stores the output in file.)

Next, we mention this executable output file as the value of LD_PRELOAD variable. This makes any program to load this library before executing the program.

After this, we write a program calling the sleep function in the same directory and compile it.

The following shows the created files:



1. On running this program as a normal user, we see that the program calls the sleep function defined by us, and prints out the statement defined by us in that function:

```
[09/06/19]seed@VM:~$ cd Task7
[09/06/19]seed@VM:~/Task7$ gcc -fPIC -g -c mylib.c
[09/06/19]seed@VM:~/Task7$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
[09/06/19]seed@VM:~/Task7$ export LD_PRELOAD=./libmylib.so.1.0.1
[09/06/19]seed@VM:~/Task7$ gcc myprogram.c -o myprogcompiled
myprogram.c: In function ‘main’:
myprogram.c:3:1: warning: implicit declaration of function ‘sleep’ [-Wimplicit-f
unction-declaration]
 sleep(1);
 ^
[09/06/19]seed@VM:~/Task7$ ./myprogcompiled
I am not sleeping!
```

On running the same program in different scenarios as specified in the lab document, I noticed that in certain situations, the library containing my sleep function was not called and instead the system defined sleep function was executed. In order to understand this behavior, I edited my program and added a system call to execute the env | grep LD command to see the process's environment variables. I mentioned grep LD because the only way any program would load my defined library and execute the sleep function was by the environment variable LD_PRELOAD. The following shows the content of my program:

```
myprogram.c (~/Task7) - gedit
Open
int main()
{
sleep(1);
system("env | grep LD");
return 0;
}
```

```
mylib.c (~/Task7) - gedit
Open
#include <stdio.h>
void sleep (int s)
{
/* If this is invoked by a privileged program,
you can do damages here! */
printf("I am not sleeping!\n");
}
```

After making this change in the program, I recompiled it and ran it as a normal user, just like before. The output was the string in the printf statement in my sleep function and the environment variables of the program containing LD substring. As seen, this process contained LD_PRELOAD as one of its environment variables:

```
[09/06/19]seed@VM:~/Task7$ gcc myprogram.c -o myprogcompiled
myprogram.c: In function 'main':
myprogram.c:3:1: warning: implicit declaration of function 'sleep' [-Wimplicit-f
unction-declaration]
 sleep(1);
 ^
myprogram.c:4:1: warning: implicit declaration of function 'system' [-Wimplicit-
function-declaration]
 system("env | grep LD");
 ^
[09/06/19]seed@VM:~/Task7$ ./myprogcompiled
I am not sleeping!
LD_PRELOAD=./libmylib.so.1.0.1
OLDPWD=/home/seed/Task7
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost
_1_64_0/stage/lib:
[09/06/19]seed@VM:~/Task7$
```

Next, I made my program a SET-UID root program and ran the program again. The output shows that my library containing sleep function was not called and also shows that 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 my library and function but the system-defined sleep function causing the program to sleep:

```
[09/06/19]seed@VM:~/Task7$ sudo chown root myprogcompiled
[09/06/19]seed@VM:~/Task7$ sudo chmod 4755 myprogcompiled
[09/06/19]seed@VM:~/Task7$ ll myprogcompiled
-rwsr-xr-x 1 root seed 7388 Sep  6 14:28 myprogcompiled
[09/06/19]seed@VM:~/Task7$ ./myprogcompiled
OLDPWD=/home/seed/Task7
```

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.

```
[09/06/19]seed@VM:~/Task7$ su
Password:
root@VM:/home/seed/Task7# export LD_PRELOAD=./libmylib.so.1.0.1
root@VM:/home/seed/Task7# ll myprogcompiled
-rwsr-xr-x 1 root seed 7388 Sep  6 14:28 myprogcompiled*
root@VM:/home/seed/Task7# ./myprogcompiled
I am not sleeping!
LD_PRELOAD=./libmylib.so.1.0.1
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost
_1_64_0/stage/lib:
OLDPWD=/home/seed/Task7
```

Next, we make this file's owner as megha (another user account other than root) and make it a SET-UID program. After this, we log into the megha's account and set the LD_PRELOAD variable again. On running the program again, we see that user-defined sleep function is called and also the LD_PRELOAD variable is present in the current process.

```
[09/06/19]seed@VM:~/Task7$ sudo chown megha myprogcompiled
[09/06/19]seed@VM:~/Task7$ sudo chmod 4755 myprogcompiled
[09/06/19]seed@VM:~/Task7$ ll myprogcompiled
-rwsr-xr-x 1 megha seed 7388 Sep  6 14:28 myprogcompiled
[09/06/19]seed@VM:~/Task7$ su megha
Password:
megha@VM:/home/seed/Task7$ export LD_PRELOAD=./libmylib.so.1.0.1
megha@VM:/home/seed/Task7$ env | grep LD
LD_PRELOAD=./libmylib.so.1.0.1
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost
_1_64_0/stage/lib:
megha@VM:/home/seed/Task7$ ll myprogcompiled
ll: command not found
megha@VM:/home/seed/Task7$ ls -l myprogcompiled
-rwsr-xr-x 1 megha seed 7388 Sep  6 14:28 myprogcompiled
megha@VM:/home/seed/Task7$ ./myprogcompiled
I am not sleeping!
LD_PRELOAD=./libmylib.so.1.0.1
LD_LIBRARY_PATH=/home/seed/source/boost_1_64_0/stage/lib:/home/seed/source/boost
_1_64_0/stage/lib:
OLDPWD=/home/seed/Task7
megha@VM:/home/seed/Task7$
```

This behavior indicates that the LD_PRELOAD variable is present if the effective and real ID are the same and is dropped if they are different. This is due to the SET-UID program's security mechanism. In the first, third and fourth case, since the owner and the account executing the file were the same, the LD_PRELOAD variable was present everytime and user-defined library was preloaded. Whereas, in the second case, the effective ID was of root and real ID was of seed, the LD_PRELOAD variable was dropped, and system-defined sleep function was called instead.

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

Here, first I compile the program provided into a file named Task8compiled. Next, this file is converted into a root-owned SET-UID program with executable permission to other users:

```
[09/06/19]seed@VM:~/Task8$ gcc Task8.c -o Task8compiled
[09/06/19]seed@VM:~/Task8$ ll
total 16
-rw-rw-r-- 1 seed seed   33 Sep  6 15:01 doctrial
-rw-rw-r-- 1 seed seed  415 Sep  6 15:22 Task8.c
-rwxrwxr-x 1 seed seed 7544 Sep  6 15:23 Task8compiled
[09/06/19]seed@VM:~/Task8$ sudo chown root Task8compiled
[09/06/19]seed@VM:~/Task8$ sudo chmod 4755 Task8compiled
[09/06/19]seed@VM:~/Task8$ ll
total 16
-rw-rw-r-- 1 seed seed   33 Sep  6 15:01 doctrial
-rw-rw-r-- 1 seed seed  415 Sep  6 15:22 Task8.c
-rwsr-xr-x 1 root seed 7544 Sep  6 15:23 Task8compiled
```

On running this program, the normal functionality will output the contents of the file specified:

```
[09/06/19]seed@VM:~/Task8$ ./Task8compiled document
This is a trial document to be read and deleted - Megha
```

Next, consider that Bob is using the megha user account (Treating Bob as others (normal user)). Here, as we can see the program runs normally when we just provide the file to be read. But, if we provide a malicious input such as "document;/bin/sh", here the program will first read the contents of the document and then run /bin/sh as a command (according to the program.) The /bin/sh allows Bob to run the shell program which has root privileges and bob then runs the rm command to remove a file on which it did not have the write permission. The root terminal is indicated by the #. This shows that even though Bob did not have any permission to write, it could remove a file easily by assuming the privileges of the root user.

The problem here is the system call inside the program which does not separate the command and user input. The user input is eventually treated as a command instead of data/document name.

```
[09/06/19]seed@VM:~/Task8$ su megha
Password:
megha@VM:/home/seed/Task8$ ls -l
total 16
-rw-rw-r-- 1 seed seed   56 Sep  6 15:35 document
-rw-rw-r-- 1 seed seed  415 Sep  6 15:22 Task8.c
-rwsr-xr-x 1 root seed 7544 Sep  6 15:23 Task8compiled
megha@VM:/home/seed/Task8$ ./Task8compiled document
This is a trial document to be read and deleted - Megha
megha@VM:/home/seed/Task8$ ./Task8compiled "document;/bin/sh"
This is a trial document to be read and deleted - Megha
# rm document
# exit
megha@VM:/home/seed/Task8$ ./Task8compiled document
/bin/cat: document: No such file or directory
megha@VM:/home/seed/Task8$
```

This can be avoided by segregating the user input and command in the program. Since the system call requires constructing the command using the input, we should avoid using system function in the program and instead use execve function which treats anything inputted from the user as input string and does not allow it to be run as a command. For this, we edit our program and compile it again, making it a root-owned SET-UID program:

```
[09/06/19]seed@VM:~/Task8$ gcc Task8.c -o usingexecve
Task8.c: In function 'main':
Task8.c:17:1: warning: implicit declaration of function 'execve' [-Wimplicit-fun
ction-declaration]
 execve(v[0], v, NULL);
 ^
[09/06/19]seed@VM:~/Task8$ sudo chown root usingexecve
[09/06/19]seed@VM:~/Task8$ sudo chmod 4755 usingexecve
[09/06/19]seed@VM:~/Task8$ ll
total 24
-rw-rw-r-- 1 seed seed   35 Sep  6 15:40 DocumentTrial
-rw-rw-r-- 1 seed seed  415 Sep  6 15:40 Task8.c
-rwsr-xr-x 1 root seed 7544 Sep  6 15:23 Task8compiled
-rwsr-xr-x 1 root seed 7544 Sep  6 15:41 usingexecve
[09/06/19]seed@VM:~/Task8$
```

We again try to perform the same attack and see that it fails because the entire user inputted string is considered as a file name rather than separating the string on ';' as document name and command as before. Also, if a user forgets the quotes and just types in the string, the terminal of the same user is opened and not of the root user, hence Bob will not have the permission to write:

```
[09/06/19]seed@VM:~/Task8$ su megha
Password:
megha@VM:/home/seed/Task8$ ./usingexecve DocumentTrial
We will try to delete this - Megha
megha@VM:/home/seed/Task8$ ./usingexecve "DocumentTrial;/bin/sh"
/bin/cat: 'DocumentTrial;/bin/sh': No such file or directory
megha@VM:/home/seed/Task8$ ./usingexecve DocumentTrial;/bin/sh
We will try to delete this - Megha
$ rm DocumentTrial
rm: remove write-protected regular file 'DocumentTrial'? y
rm: cannot remove 'DocumentTrial': Permission denied
```

This happens because, as seen in the program, the command in system is constructed using strings inputted while executing. In terminal, we can enter multiple commands using ';' and hence the second part after ';' in the input is directly considered as a command rather than a part of the file name. There is no input validation while using system (), but there is some when we use execve. When we use execve, the input is directly entered as the second parameter to the function which in fact is considered as the entire file name and is not appended into a string to construct the command, as before. This avoids this kind of attack.

Task 9: Capability Leaking

Here, we compile the given program and make it root-owned SET-UID program:

```
[09/06/19]seed@VM:~/Task9$ gcc task9.c -o task9
task9.c: In function 'main':
task9.c:16:1: warning: implicit declaration of function 'sleep' [-Wimplicit-function-declaration]
 sleep(1);
 ^
task9.c:19:1: warning: implicit declaration of function 'setuid' [-Wimplicit-function-declaration]
 Terminator getuid()); /* getuid() returns the real uid */
 ^
task9.c:19:8: warning: implicit declaration of function 'getuid' [-Wimplicit-function-declaration]
 setuid(getuid()); /* getuid() returns the real uid */
        ^
task9.c:20:5: warning: implicit declaration of function 'fork' [-Wimplicit-function-declaration]
 if (fork()) { /* In the parent process */
     ^
task9.c:21:1: warning: implicit declaration of function 'close' [-Wimplicit-function-declaration]
 close (fd);
 ^
task9.c:27:1: warning: implicit declaration of function 'write' [-Wimplicit-function-declaration]
 write (fd, "Malicious Data\n", 15);
 ^
[09/06/19]seed@VM:~/Task9$ sudo chown root task9
[09/06/19]seed@VM:~/Task9$ sudo chmod 4755 task9
[09/06/19]seed@VM:~/Task9$ ll
total 12
-rwsr-xr-x 1 root seed 7640 Sep  6 15:54 task9
-rw-rw-r-- 1 seed seed  884 Sep  6 15:52 task9.c
```

Here, we create a file named zzz in the /etc folder containing a print statement in the main:

```
Terminal
[09/06/19]seed@VM:/$ cd /etc
[09/06/19]seed@VM:/etc$ ll zzz
-rw-r--r-- 1 root root 34 Sep  6 16:41 zzz
[09/06/19]seed@VM:/etc$ cat zzz
This is an important file - Megha
[09/06/19]seed@VM:/etc$ 
```

Next, we run the program and again see the content of the zzz file, 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.

```
[09/06/19]seed@VM:~/Task9$ ./task9
[09/06/19]seed@VM:~/Task9$ cat /etc/zzz
This is an important file - Megha
Malicious Data
[09/06/19]seed@VM:~/Task9$ 
```