Environment Variable and Set-UID Program Lab

Final Project Report

By Harish Marepalli

First Name: Harish

Last Name: Marepalli

SJSU ID: 016707314

Professor: Dr. Younghee Park

Table of Contents

Introduction:

An environment variable is a dynamic value that can be used by programs running in a specific environment. These variables provide information about the environment in which the program is running and can be accessed by the program through APIs. They are used by most operating systems since they were introduced to Unix in 1979. Although environment variables affect program behaviors, how they achieve that is not well understood by many programmers. As a result, if a program uses environment variables, but the programmer does not know that they are used, the program may have vulnerabilities.

A Set-UID (Set User ID) is a special permission in Unix/Linux operating systems that allows a program to run with the permissions of its owner instead of the user who executed it. This is often used to allow non-privileged users to execute certain privileged tasks. However, when these two concepts are combined, they can create a security vulnerability known as an Environment Variable and Set-UID Attack. This attack occurs when an attacker sets an environment variable that a privileged program uses, and then runs the program with the Set-UID permission. By doing so, the attacker can execute malicious code with the elevated privileges of the program.

The objective of this project is to understand what risks such privileged programs face and how they can be attacked if there are mistakes in the code. We also understand how environment variables work, how they are propagated from parent process to child, and how they affect system/program behaviors. We are particularly interested in how environment variables affect the behavior of Set-UID programs, which are usually privileged programs.

Lab Environment:

This lab has been tested on the SEED Ubuntu 20.04 VM. We can download a pre-built image from the SEED website and run the SEED VM on our own computer. However, most of the SEED labs can be conducted on the cloud, and we can follow our instruction to create a SEED VM on the cloud.

Below is the figure (Figure a) that shows the VM Virtual Box Manager setup.

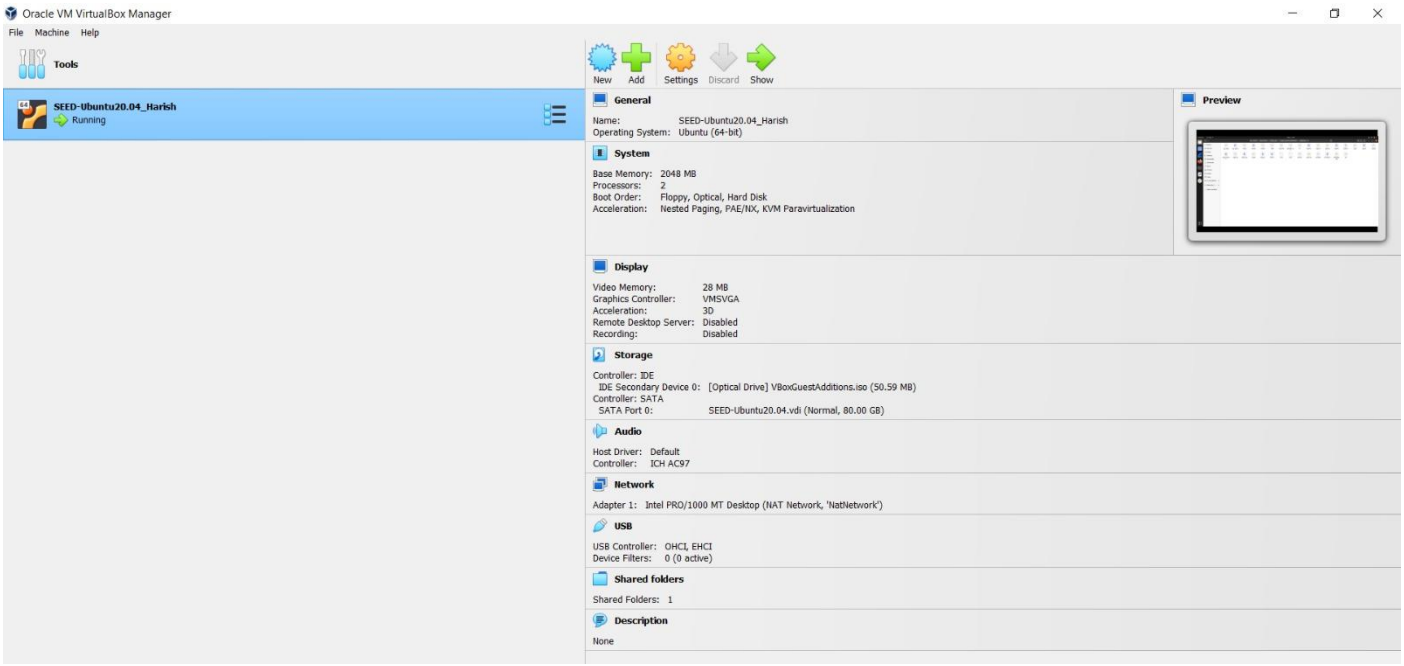

Figure a: VM Virtual Box Manager setup

Lab Tasks:

To conduct this project, we need to have Labsetup.zip, which can be downloaded from the lab's website.

Task 1: Manipulating Environment Variables:

In this task, we study the commands that can be used to set and unset environment variables. We are using Bash in the seed account. The default shell that a user use is set in the */etc/passwd* file (the last field of each entry). You can change this to another shell program using the command *chsh* (we will not do it for this lab). Below are the sub tasks that are being done in this task 1.

1. Use printenv or env command to print out the environment variables. If you are interested in some particular environment variables, such as PWD, you can use "printenv PWD" or "env | grep PWD".

This task is just to get to know basic environment variable visualization/manipulation commands, such as env and printenv.

Figure 1.1 shows the command *cat /etc/passwd*, that is being run.

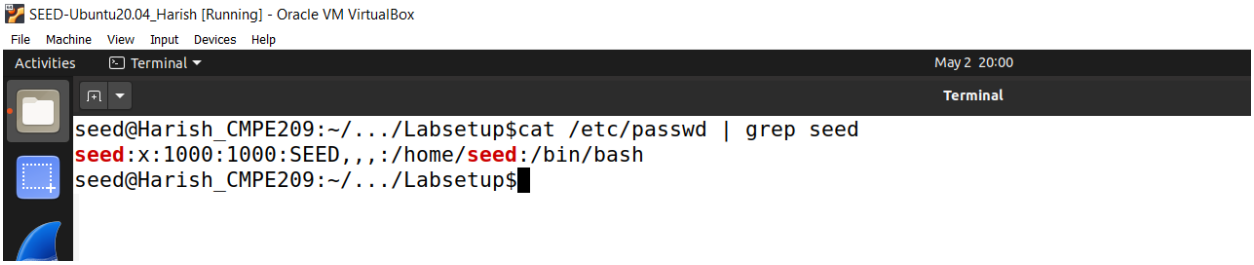

Figure 1.1

Figure 1.2 shows the *printenv* command which is one way to print out all the environment variables.

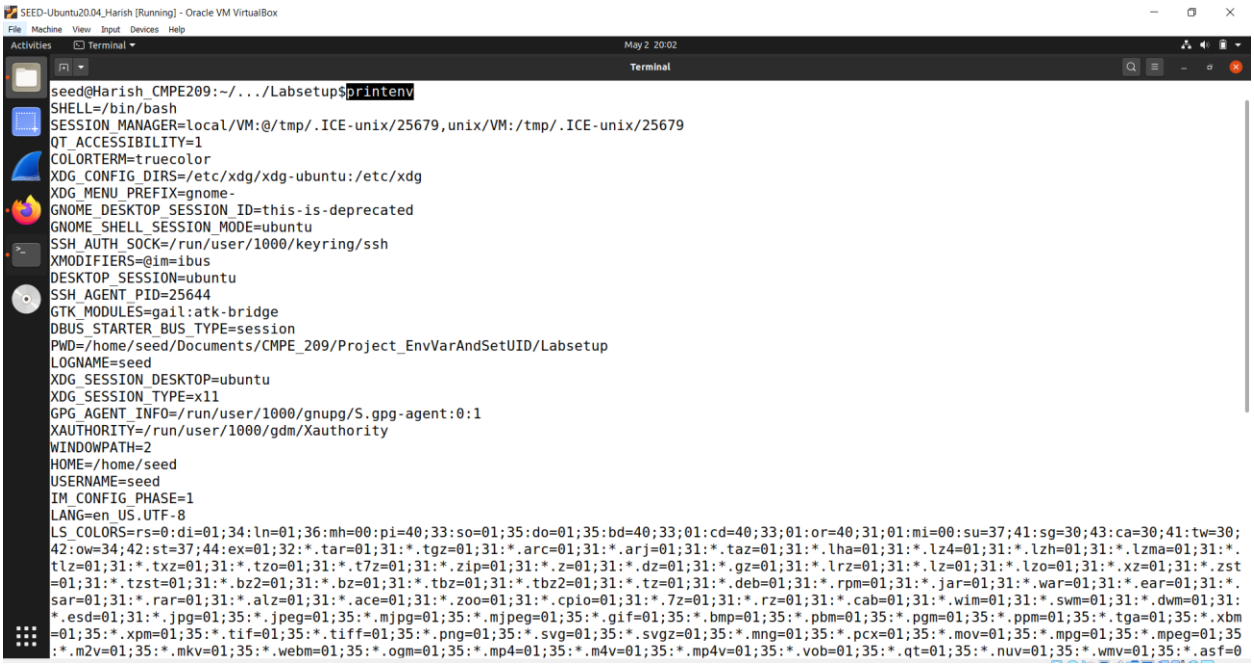

Figure 1.2

Figure 1.3 shows the continuation of the environment variables.

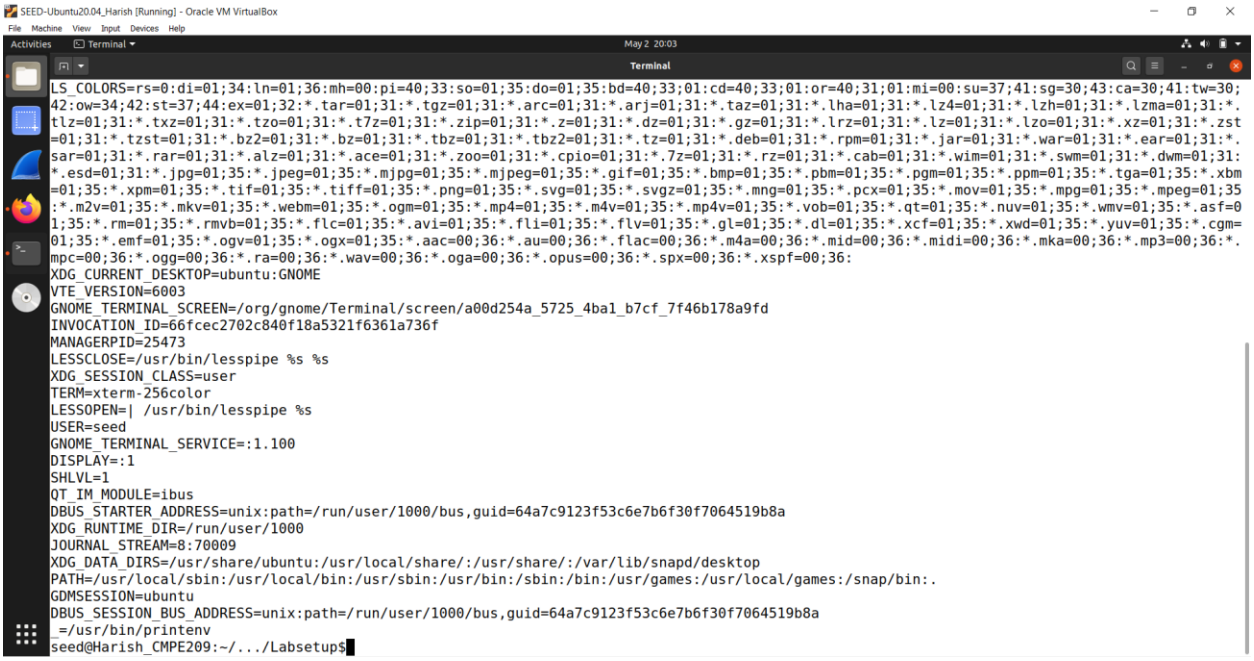

Figure 1.3

The second method to print the environment variables is by using *env* command. Figure 1.4 shows the same. we get the same result by using both the commands.

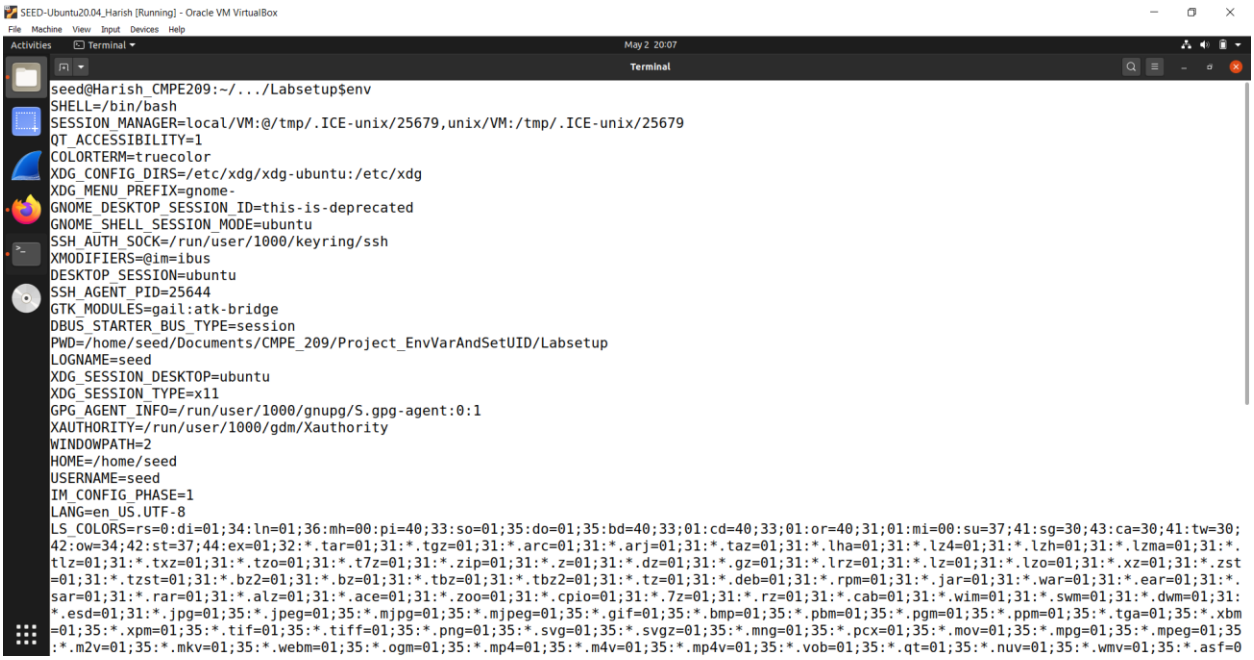

Figure 1.4

We can find the difference between these two ways of printing using the *diff* command. Figure 1.5 shows the running of the diff command.

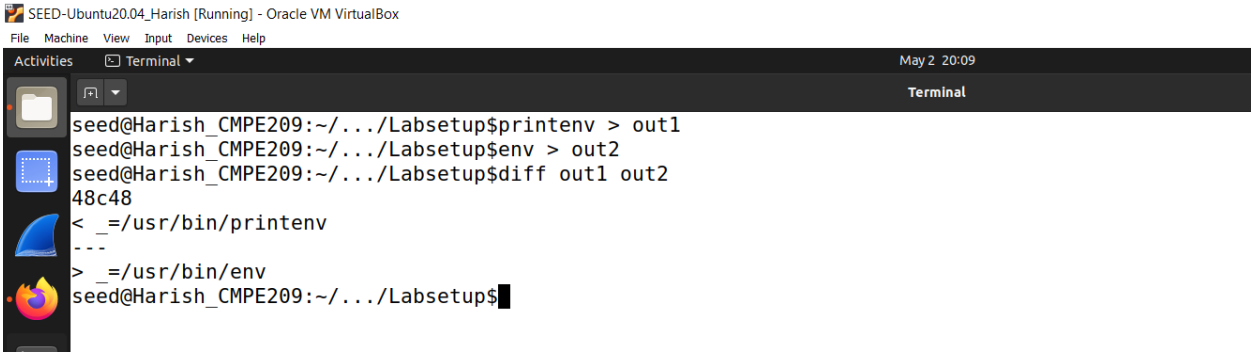

Figure 1.5

We can find some particular environment variable like *PWD* (Present Working Directory) using *printenv PWD* or *env | grep PWD*. Below figure, Figure 1.6 shows the usage of the same commands. The difference here is that the second command shows the environment variable name and value which is not the case with the first one.

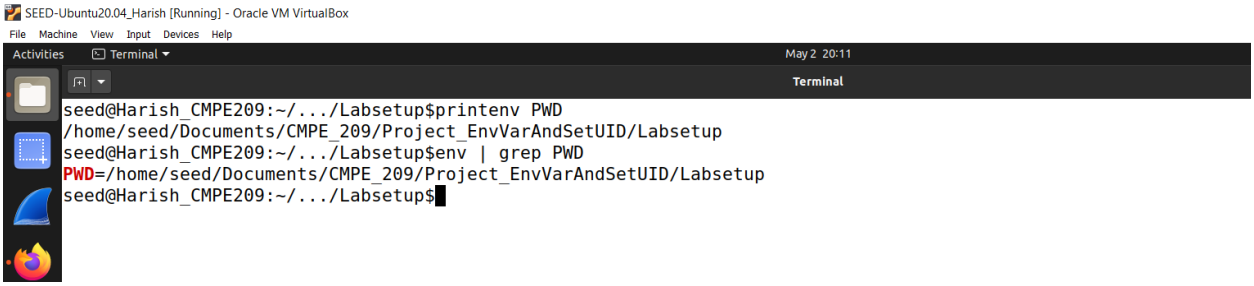

Figure 1.6

2. Use export and unset to set or unset environment variables. It should be noted that these two commands are not separate programs; they are two of the Bash's internal commands (we will not be able to find them outside of Bash).

This task is just to get to know basic environment variable visualization/manipulation commands, such as export and unset. These two are Bash internal commands.

Figure 1.7 shows the running of *export* command and the verification of the exported environment variable.

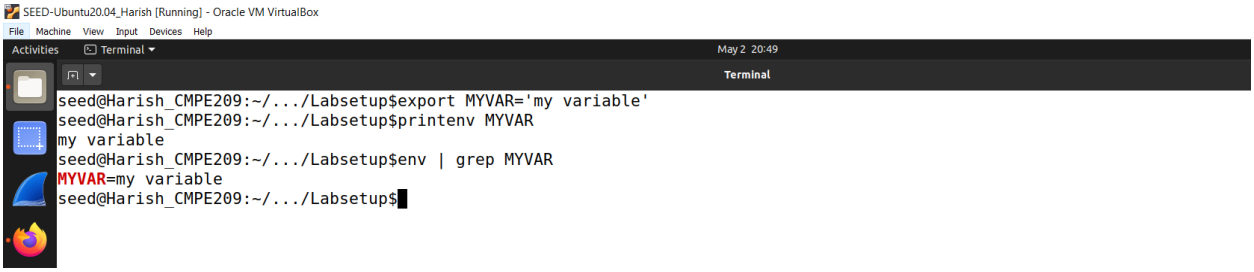

Figure 1.7

Figure 1.8 shows the running of *unset* command and the verification of the exported environment variable. The command unset is used to delete the environment value. So, after this is run, the value gets deleted from environment variable space.

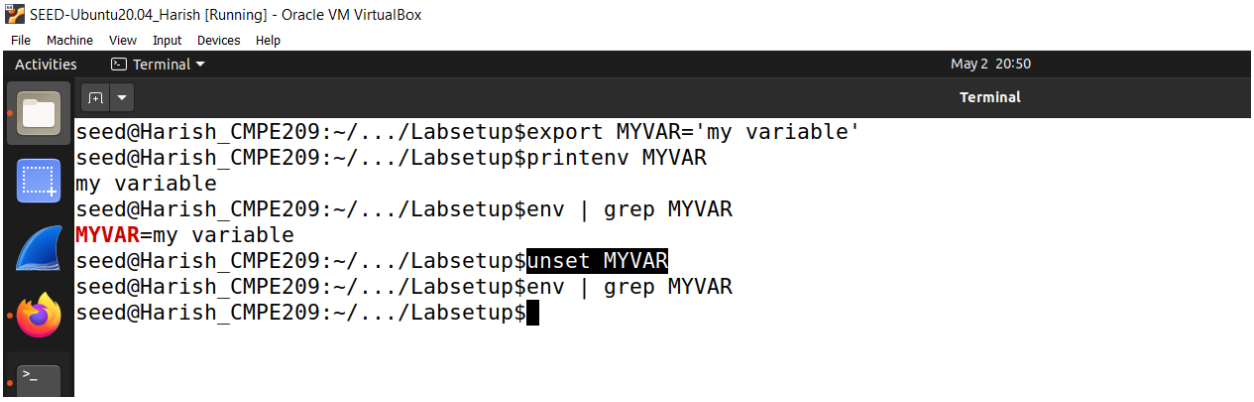

Figure 1.8

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

In this task, we study how a child process gets its environment variables from its parent. In Unix, *fork()* creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent; however, several things are not inherited by the child (please see the manual of *fork()* by typing the following command: *man fork*). In this task, we would like to know whether the parent's environment variables are inherited by the child process or not.

Step 1. Please compile and run the following program and describe your observation. The program can be found in the *Labsetup* folder; it can be compiled using "*gcc myprintenv.c*", which will generate a binary and we save this output to a file named c*hild*.

Figure 2.1 shows the *myprintenv.c* code.

Figure 2.1

Here, the *printenv* function is used to print out all the environment variables by using *environ*. In this program, it is used to create a child process using *fork* function. If the return value is zero, then it is a child process otherwise it will be the parent process. In both the processes, we print the environment variable.

The above program print the environment variable for the child process.

Figure 2.2 shows the compilation of *myprintenv.c* file and save it into a file named *child*.

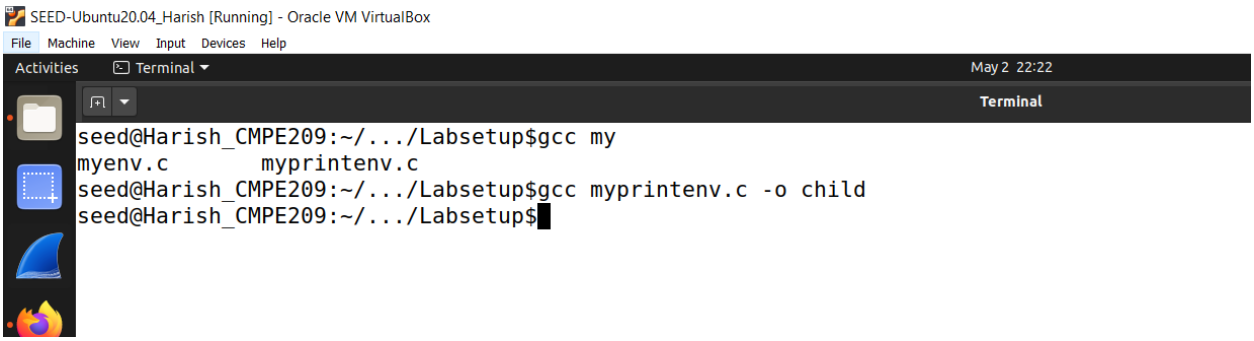

Figure 2.2

Step 2. Now comment out the printenv() statement in the child process case (Line ①), and uncomment the printenv() statement in the parent process case (Line ②). Compile and run the code again, and describe your observation. Save the output in another file.

Now, comment out the *printenv* method in child process and uncomment the same in the parent process.

Figure 2.3 shows the same.

Figure 2.3

Here, we commented the printenv method in child process and uncommented it in the parent process.

Figure 2.4 shows the compilation of *myprintenv.c* file and save it into a file named *parent*.

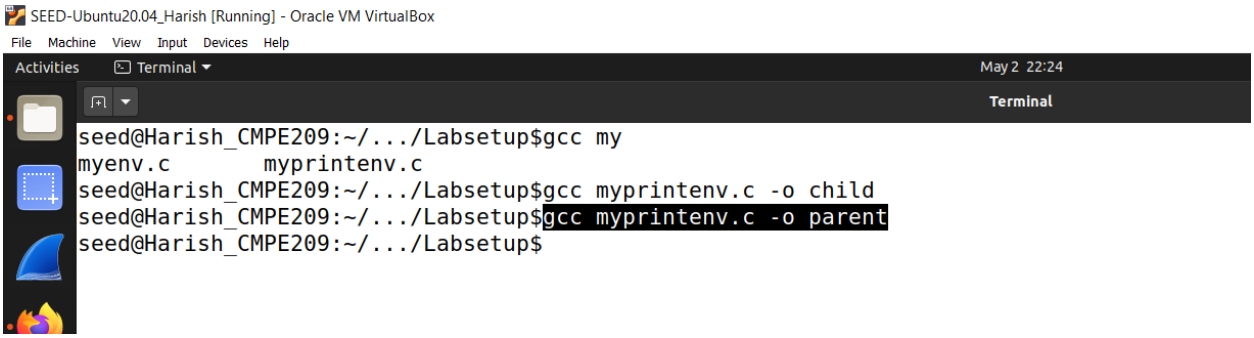

Figure 2.4

By running the above file, I observed the environment variables of the parent process.

We can view both the environment variables of child and parent processes using *./child* and *./parent* commands.

Step 3. Compare the difference of these two files using the diff command. Please draw your conclusion.

We save the output of child process in cout.txt and the parent process in pout.txt and compare both the files.

Figure 2.5 shows the same process.

Figure 2.5

Here, the programs are different in the environment variables passed from the parent process to the child process and they are mostly identical. We can conclude that the environment variables at the parent and the child processes are the same, which means that the child process inherits its environment variables from the parent process at the moment it is created with the call to fork().

Task 3: Environment Variables and execve ():

In this task, we study how environment variables are affected when a new program is executed via *execve()*. The function *execve()* calls a system call to load a new command and execute it; this function never returns. No new process is created; instead, the calling process's text, data, bss, and stack are overwritten by that of the program loaded. Essentially, *execve()* runs the new program inside the calling process. We are interested in what happens to the environment variables; are they automatically inherited by the new program?

Step 1. Please compile and run the following program and describe your observation. This program simply executes a program called */usr/bin/env*, which prints out the environment variables of the current process.

This is a safe way to execute a program. This brings us the environment variables of the current process.

Figure 3.1 shows the *myenv.c* file which has the path */usr/bin/env.*

Figure 3.1

Figure 3.2 shows the compilation of *myenv.c* code.

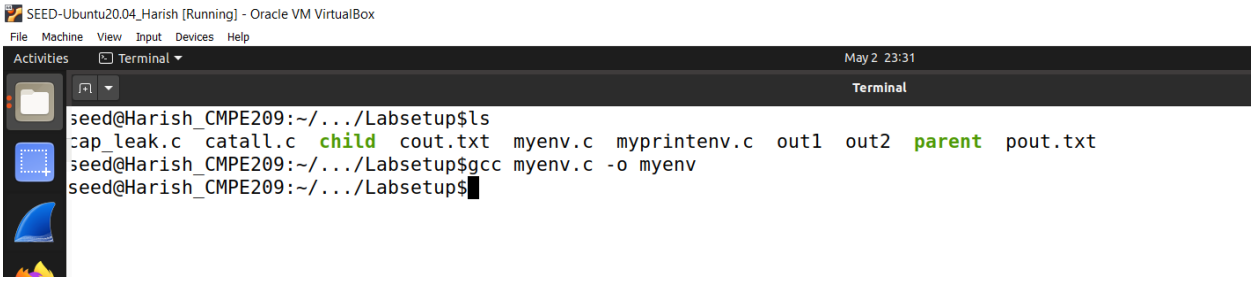

Figure 3.2

If we run *./myenv*, we see nothing gets printed out because the third parameters is used to pass environment variables to the process we created and in here since it is NULL, it prints nothing.

Step 2. Change the invocation of execve() in Line ① to the following; describe your observation.
execve("/usr/bin/env", argv, environ);

Here, instead of NULL, we put environ.

Figure 3.3 shows the code for the same. Here, the environ is passed to the the */usr/bin/env* process.

Figure 3.3

Figure 3.4 shows the compilation of *myenv.c* code. We compile it and run the command *./myenv2*.

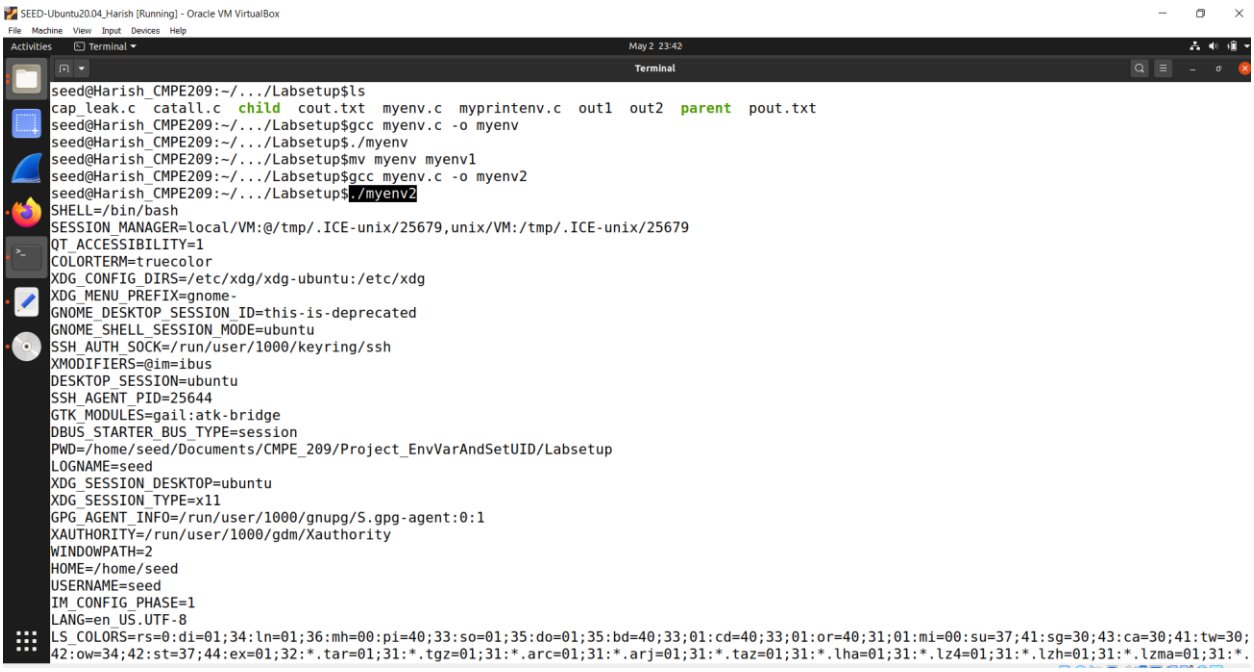

Figure 3.4

This time, we can see that the environment variables are printed out.

Step 3. Please draw your conclusion regarding how the new program gets its environment variables.

The new program must get its environment variables explicitly through the *execve* call. As we saw from the task, if no environment variables are passed through the call, the program will not have access to them.

Task 4: Environment Variables and system ():

In this task, we study how environment variables are affected when a new program is executed via the system() function. This function is used to execute a command, but unlike execve(), which directly executes a command, system() actually executes "/bin/sh -c command", i.e., it executes /bin/sh, and asks the shell to execute the command.

If you look at the implementation of the system() function, you will see that it uses execl() to execute /bin/sh; execl() calls execve(), passing to it the environment variables array. Therefore, using system(), the environment variables of the calling process is passed to the new program /bin/sh. Please compile and run the following program to verify this.

Figure 4.1 shows the code for *mysys.c*.

Figure 4.1

Figure 4.2 shows the compilation of *mysys.c*.

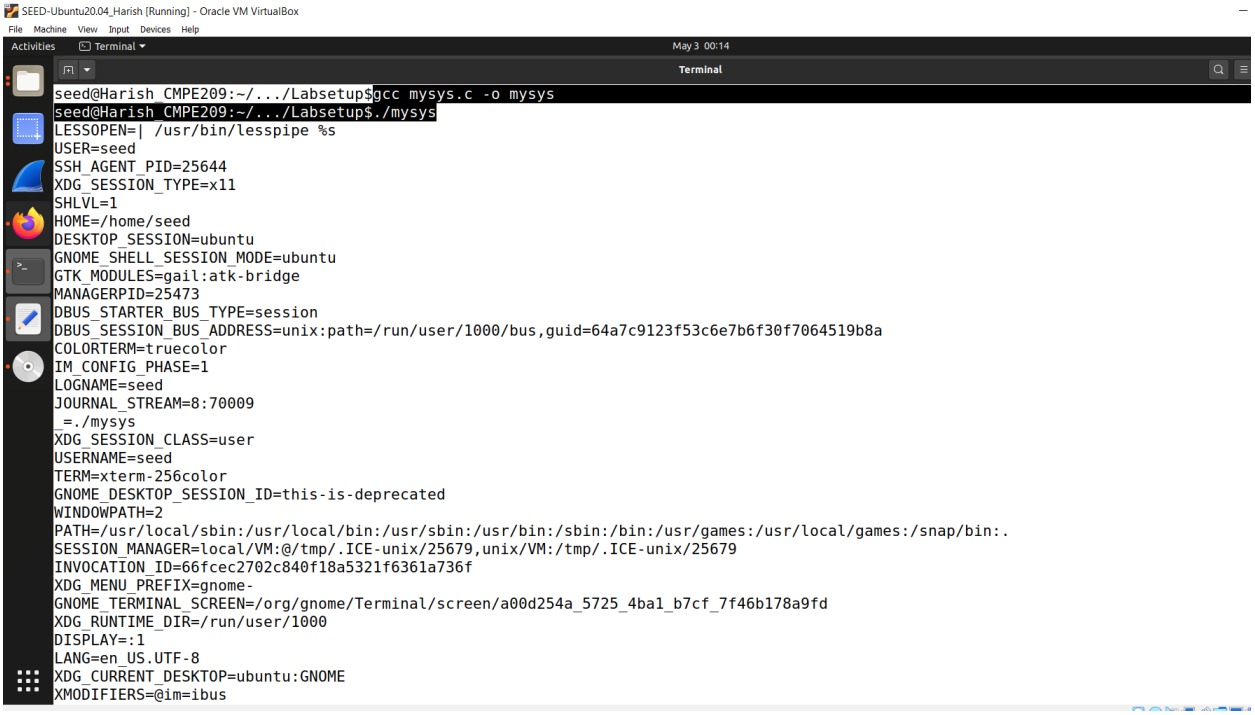

Figure 4.2

Figure 4.3 shows the continuation of environment variables.

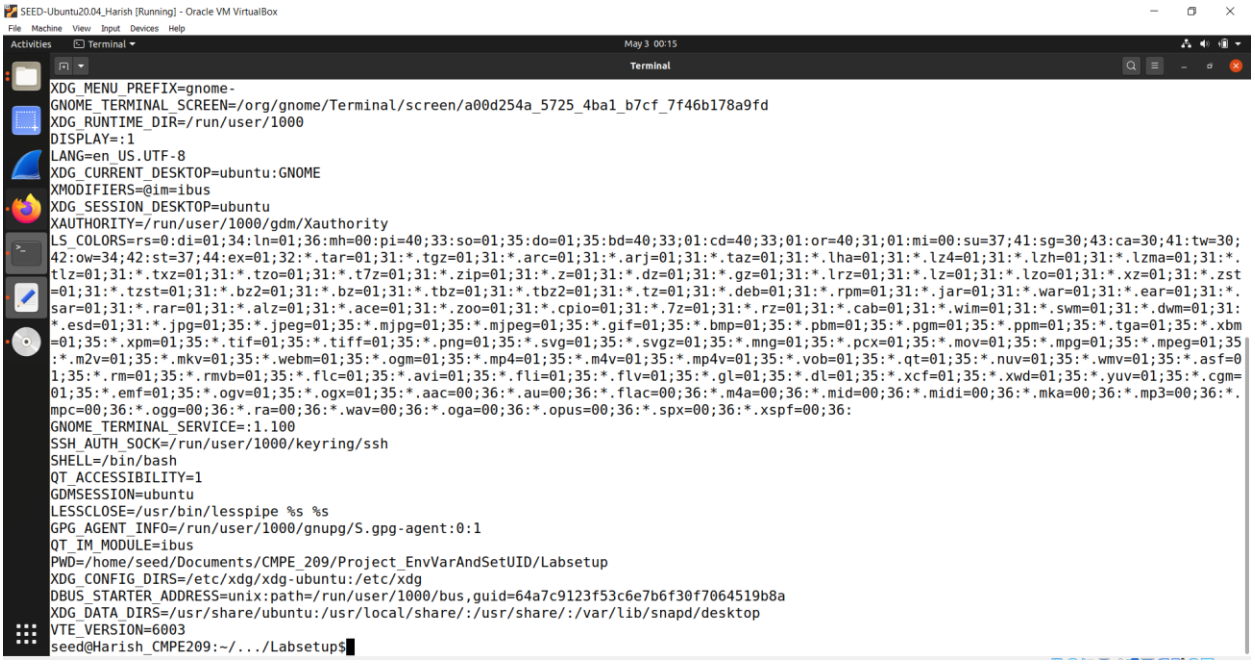

Figure 4.4

When *./mysys* is run, it can be seen that the environment variables are printed out.

But how do we know that or make sure that the program called */bin/sh*?

We verify it by passing command line parameter to *./mysys*. Since we did not handle the command line parameters in the code, we see nothing.

Figure 4.5 shows the same.

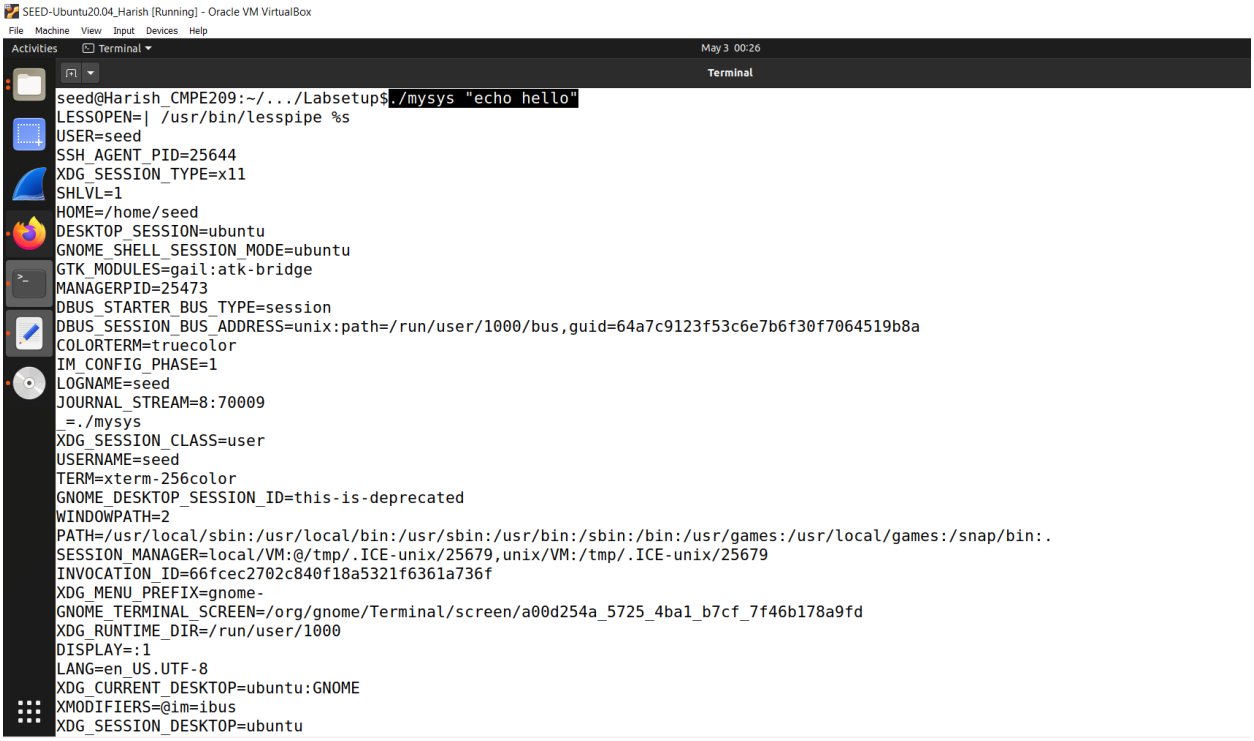

Figure 4.5

In the upcoming tasks, we see how to attack this program. It is vulnerable because if its core is implicitly linked to the vulnerable version, then we can attack it. So, we can conclude that by using the system() call, the environment variables are passed to the program because it uses *exec1* internally, which provides the environment variables to *execve* automatically.

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. For example, if the program's owner is root, when anyone runs this program, the program gains the root's privileges during its execution. Set-UID allows us to do many interesting things, but since it escalates the user's privilege, it is quite risky. Although the behaviors of Set-UID programs are decided by their program logic, not by users, users can indeed affect the behaviors via environment variables. To understand how Set-UID programs are affected, let us first figure out whether environment variables are inherited by the Set-UID program's process from the user's process.

Here, we want to check whether Set-UID programs are affected by normal user through the environment variables or not.

Step 1. Write the following program that can print out all the environment variables in the current process.

Here, we compile the code and change it to a Set-UID program owned by root.

Figure 5.1 shows the *printenv.c* code.

Figure 5.1

Step 2. Compile the above program, change its ownership to root, and make it a Set-UID program.

Figure 5.2 shows the compilation of the *printenv.c* file and running of it as well. We run it as *./printenv* and it prints out all the environment variables.

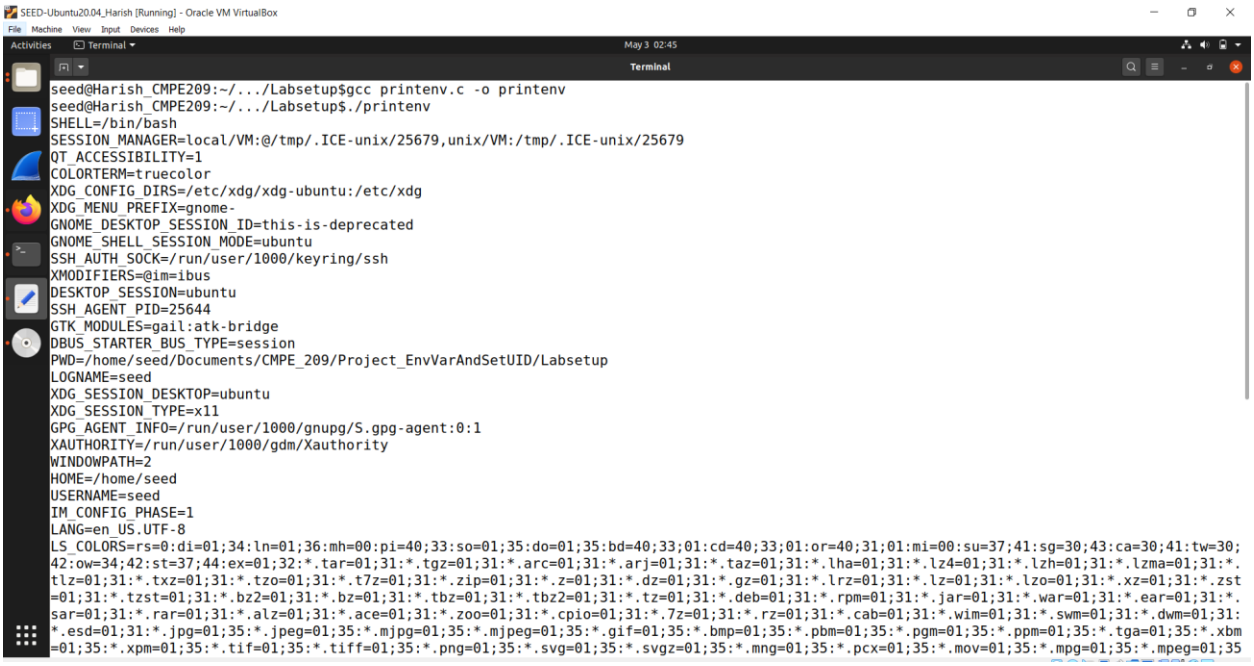

Figure 5.2

Figure 5.3 shows the continuation of the environment variables.

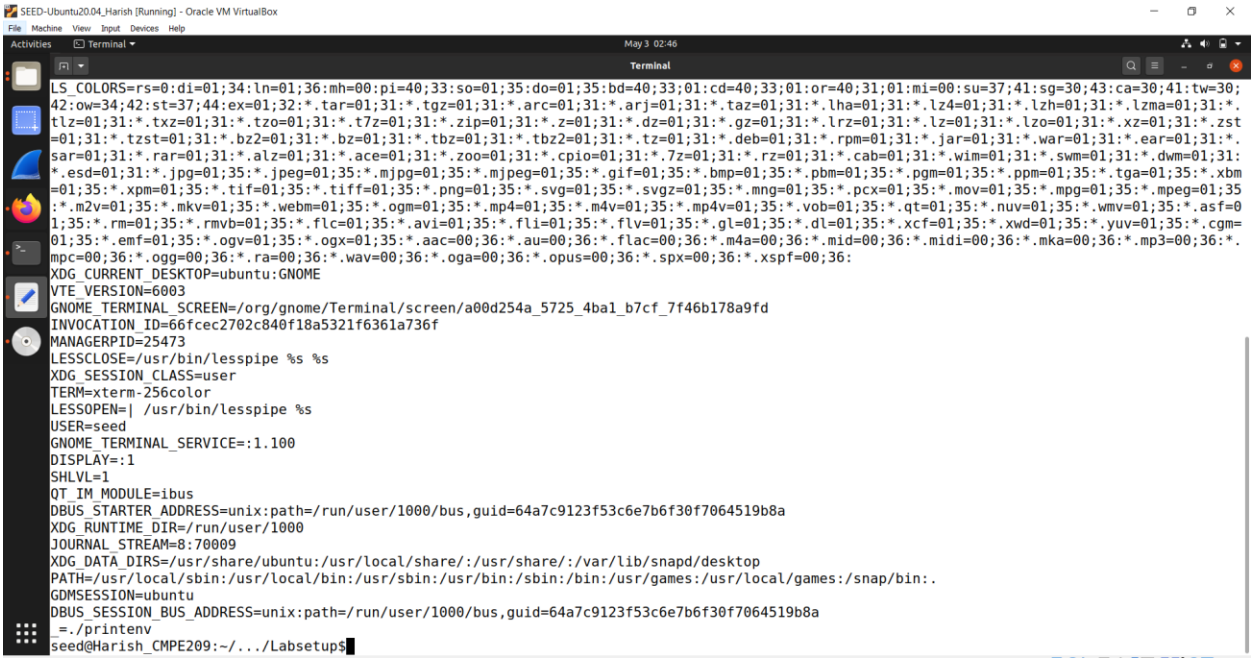

Figure 5.3

Now, we change the owner to root using the command *sudo chown root printenv* and change the mode using *sudo chmod 4755 printenv*.

After these modifications, we check the permissions using the command *ls -l printenv*.

Figure 5.4 shows the same commands that are being run.

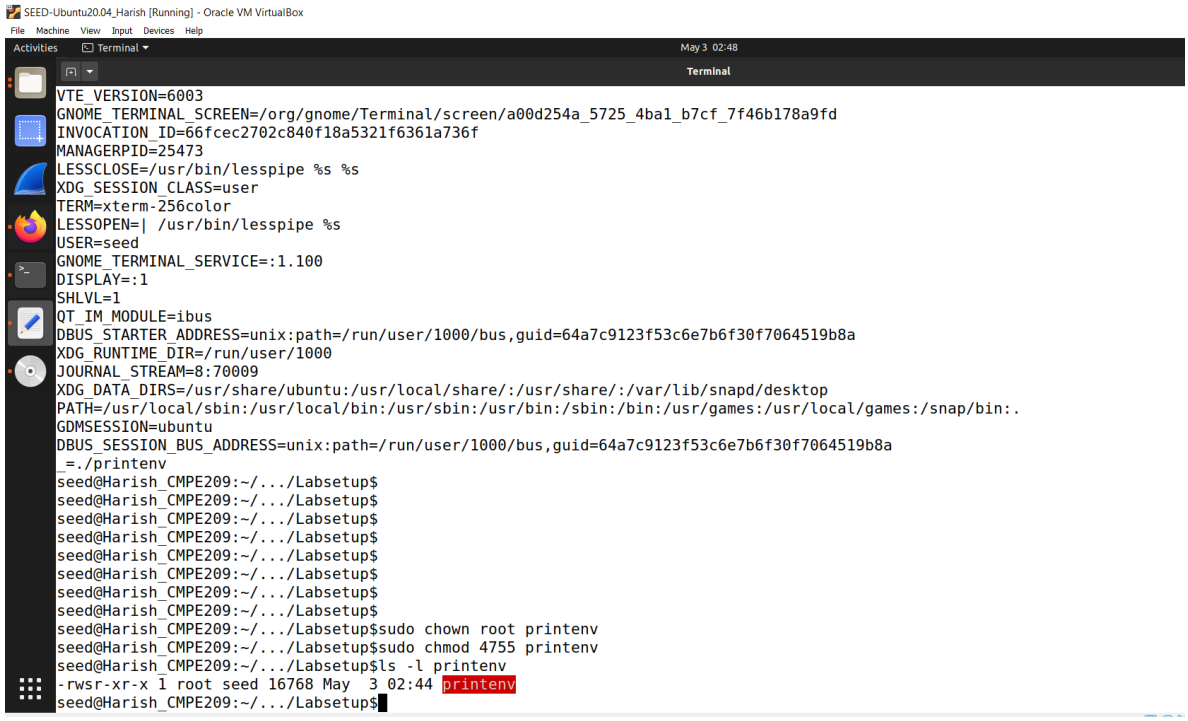

Figure 5.4

The "ls" command stands for "list" and is used to display a list of files and directories in the current working directory. The "-l" option tells the "ls" command to display the long format output of the file or directory, which includes detailed information such as the file permissions, ownership, creation date, size, and file name.

After we run the above commands, it becomes a Set-UID program owned by the root.

Step 3. In your shell (you need to be in a normal user account, not the root account), use the export command to set the following environment variables (they may have already exist):

- PATH
- LD_LIBRARY_PATH
- ANY NAME (this is an environment variable defined by you, so pick whatever name you want).

These environment variables are set in the user's shell process. Now, run the Set-UID program from Step 2 in your shell. After you type the name of the program in your shell, the shell forks a child process, and uses the child process to run the program. Please check whether all the environment variables you set in the shell process (parent) get into the Set-UID child process. Describe your observation. If there are surprises to you, describe them.

Now, we may check whether we have those environment variables which are mentioned above.

PATH would be there by default, because it is a system environment variable.

If we check for LD_LIBRARY_PATH, it gives empty result. So, we can create it using *export* command.

We also can create our own environment variable and we create one here with the name *MYVAR* and give the value as 'my variable'.

Here, we set the LD_LIBRARY_PATH to a value ".", which indicates the current folder.

Figure 5.5 shows all the commands that are mentioned above.

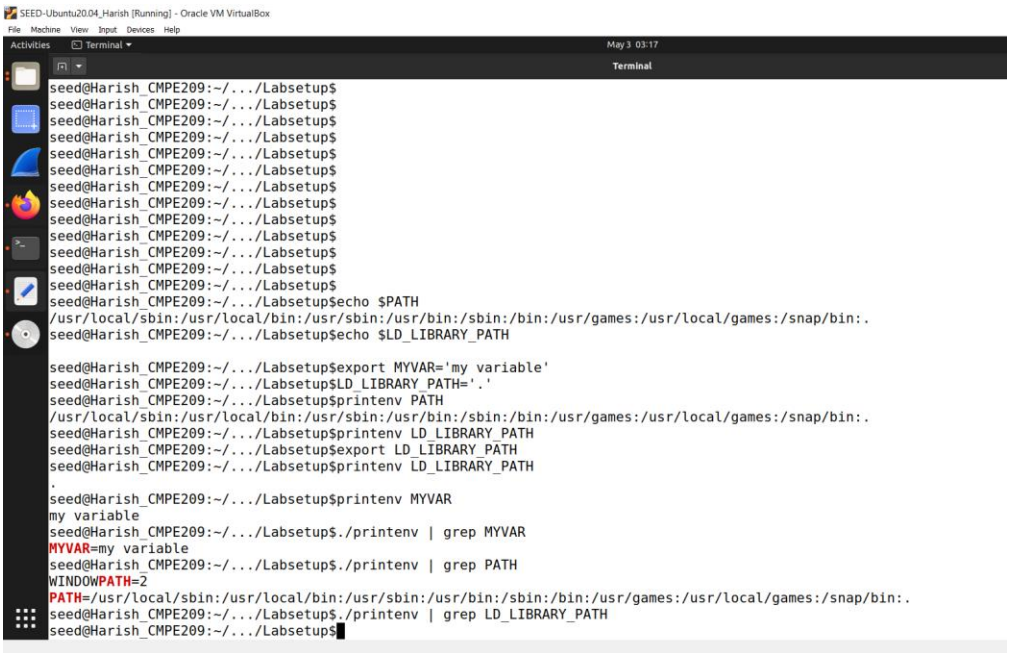

Figure 5.5

Now, we see all these environment variables and we run *printenv* to see whether we get these environment variables. Here, we can print the LD_LIBRARY_PATH only if we export it to the environment space. We see in the above screenshot that all the mentioned variables are set.

Now, we check whether we can print the environment variables in the Set-UID child process.

To achieve this, we run *./printenv | grep* as shown in the above screenshot.

From the above screenshot, we can see that we can be able to print the user created environment variable that is passed to Set-UID program. But coming to the LD_LIBRARY_PATH, it is not passed to the Set-UID program and return empty value because it is very dangerous if it can be manipulated by a normal user.

So, we can conclude that the environment variables are passed to the Set-UID child process. Surprisingly, however, the variable LD_LIBRARY_PATH doesn't seem to have been passed.

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 behavior 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 behavior of the Set-UID program. In Bash, you can change the PATH environment variable in the following way (this example adds the directory /home/seed to the beginning of the PATH environment variable):

$ export PATH=/home/seed:$PATH

The Set-UID program below is supposed to execute the /bin/ls command; however, the programmer only uses the relative path for the ls command, rather than the absolute path.

Figure 6.1 below shows the *myls.c* code which has the path as */bin/ls.*

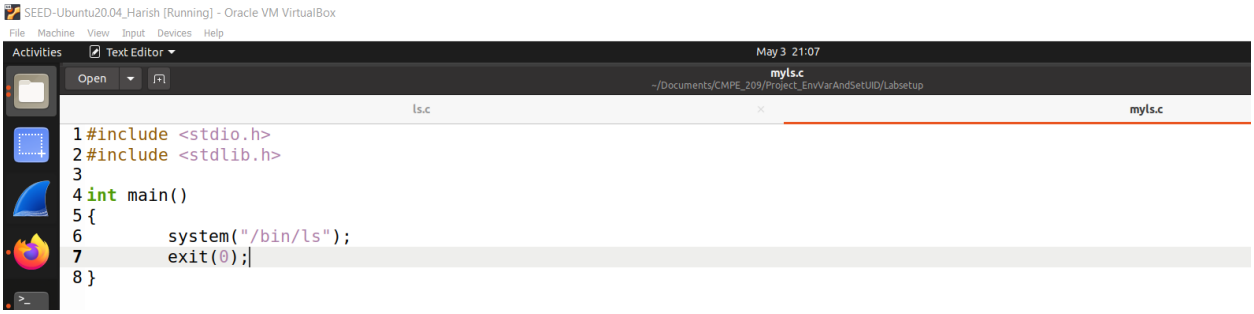

Figure 6.1

Here, we export our current folder environment variable, compile it, and change it to Set-UID program and run our malicious code. By this we can check that whether it will run the ls program in our home folder or /bin/ls.

Figure 6.2 shows the compilation of the *myls.c* code.

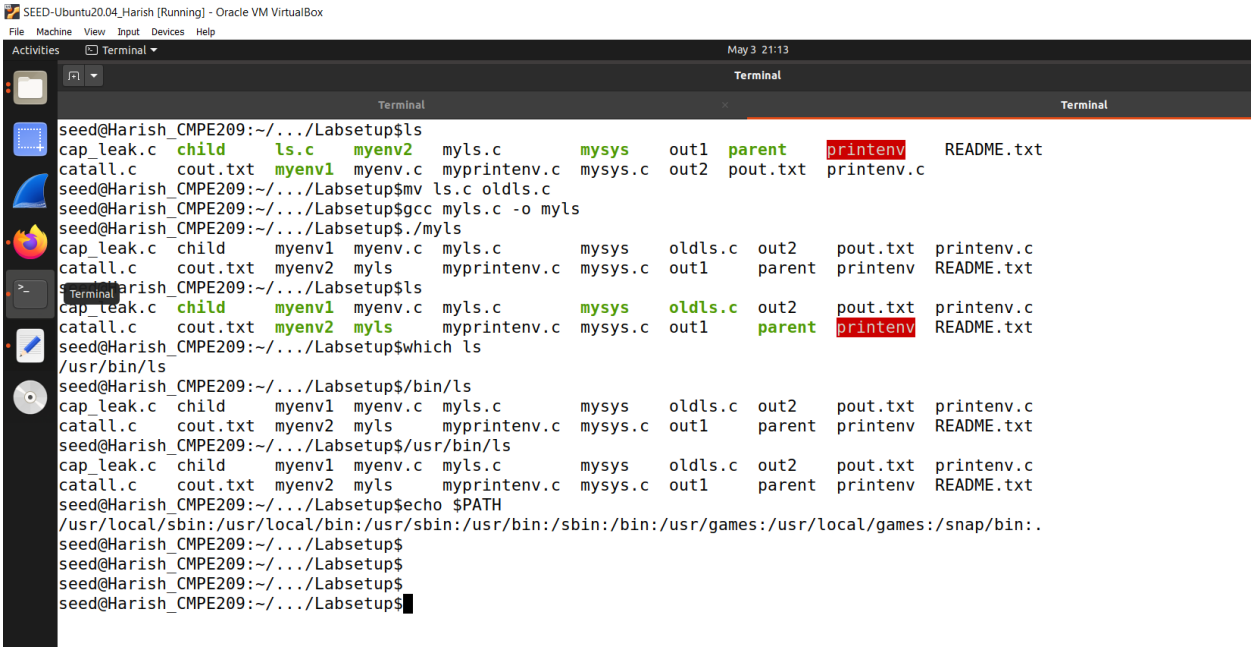

Figure 6.2

In the above screenshot, if we run *./myls*, it will call */bin/ls* and it lists out all the file names like *ls* does.

We see the same output for *./myls* and *ls*. Only change is that the printenv is highlighted with red color.

We can see which ls is being executed using *which ls* command. The /bin/ls and */usr/bin/ls* also return the same output.

After we run *which ls* command, we see that it returned */usr/bin/ls* and not */bin/ls/* and the reason is as below.

If we run the *echo $PATH* command, we see the PATH variables. Here, the system will search the folders in these paths in one of the variables following the order in the PATH variable. Here, */usr/bin* folder is in front of */bin* folder and that is the reason it has called */usr/bin/ls* instead of */bin/ls*.

Now, we try to manipulate the PATH environment variable as a normal user, and we see it in figure 6.3.

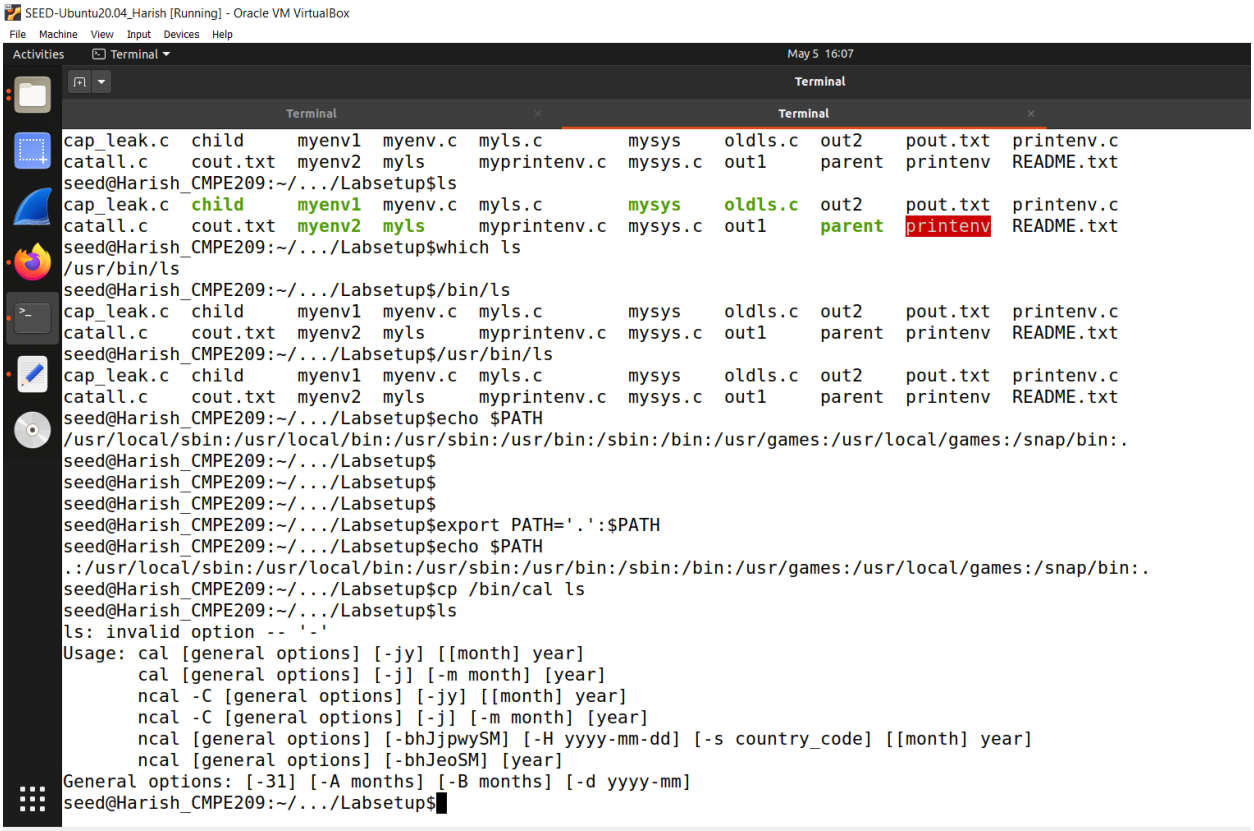

Figure 6.3

Here, we change the PATH in such a way that it first searches for local path instead of any other folders. If we check, we see that "." is at the front of the path.

Now, we try to make a fake *ls.* So, we copy the */bin/cal (calendar)* to *ls (current folder)*. If we run ls, we can see that it is trying to call *cal.*

Here, if we run *./myls*, it runs normal */bin/ls* instead of the manipulated *ls* (which runs calendar).

What if we want to print the normal *ls* instead of manipulated *ls*, which prints the calls the calendar.

This batch process is considered to be manipulated because we changed the PATH variable by prepending "." to the path.

We open a new terminal to check the PATH and it return the normal path (not the manipulated one). Figure 6.4 is used to explain the same.

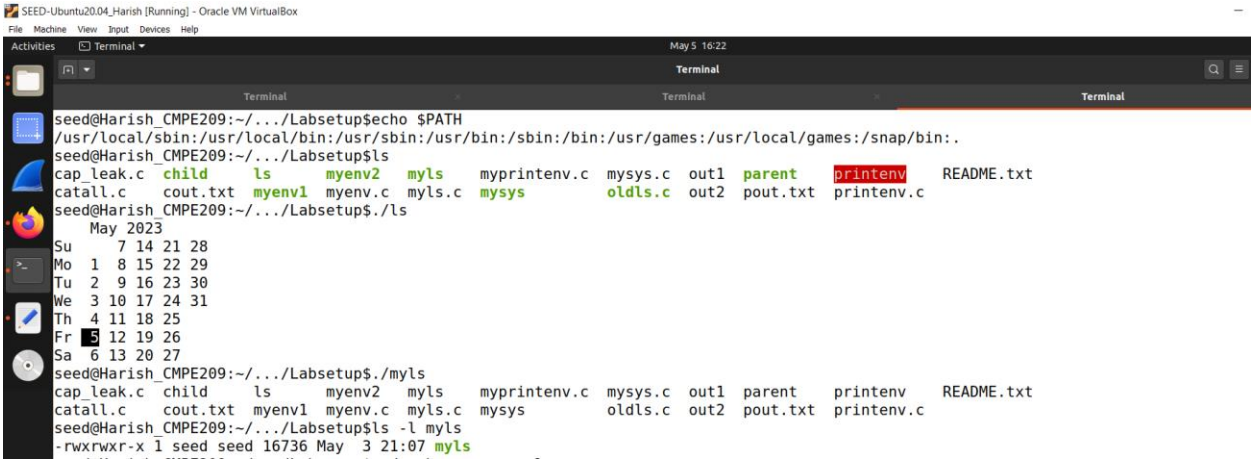

Figure 6.4

Here, we run the commands in a new terminal. If we type *ls*, then it gives the normal output. If we *./ls*, it runs the fake *ls (calendar)*. If we run ./myls here, it will give a normal output. If we check the permissions of it, we see that it is a normal program. If we type *./myls* in the manipulated terminal i.e. 2nd terminal as well, we get the normal output only.

Now, we change it to Set-UID program and change the mode as well. Figure 6.5 shows the same.

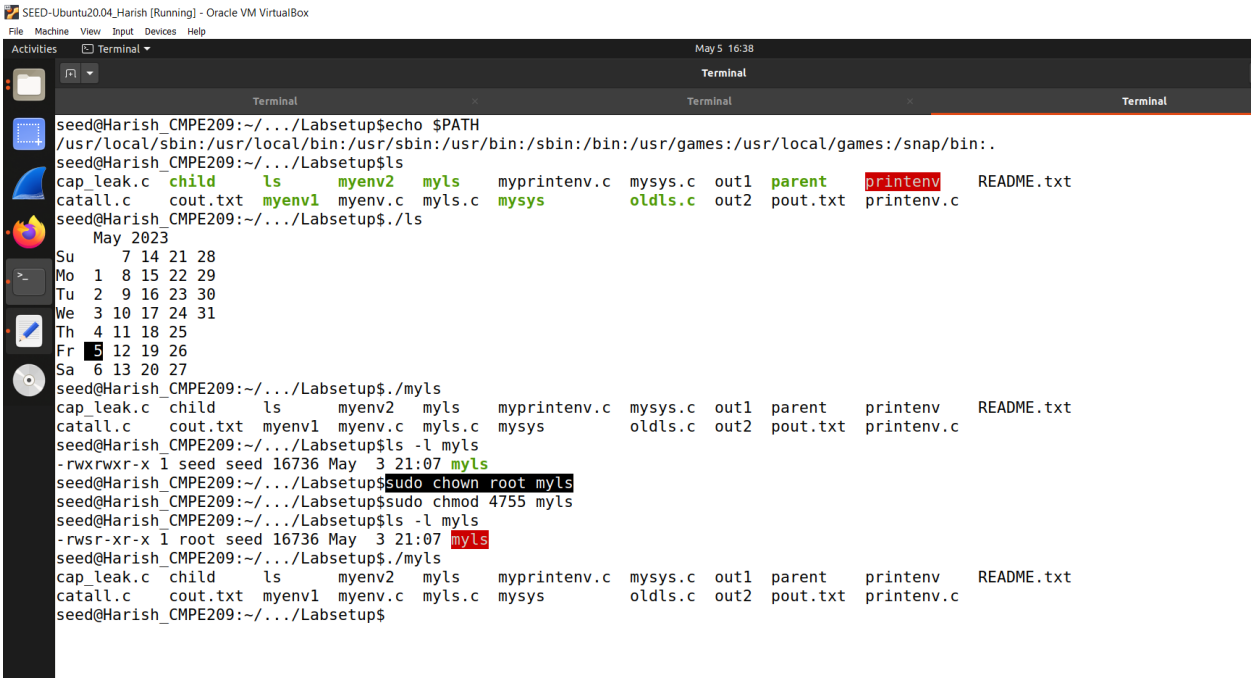

Figure 6.5

After running those commands, it becomes the Set-UID program. Now, run *./myls* in the manipulated terminal and is shown in the figure 6.6.

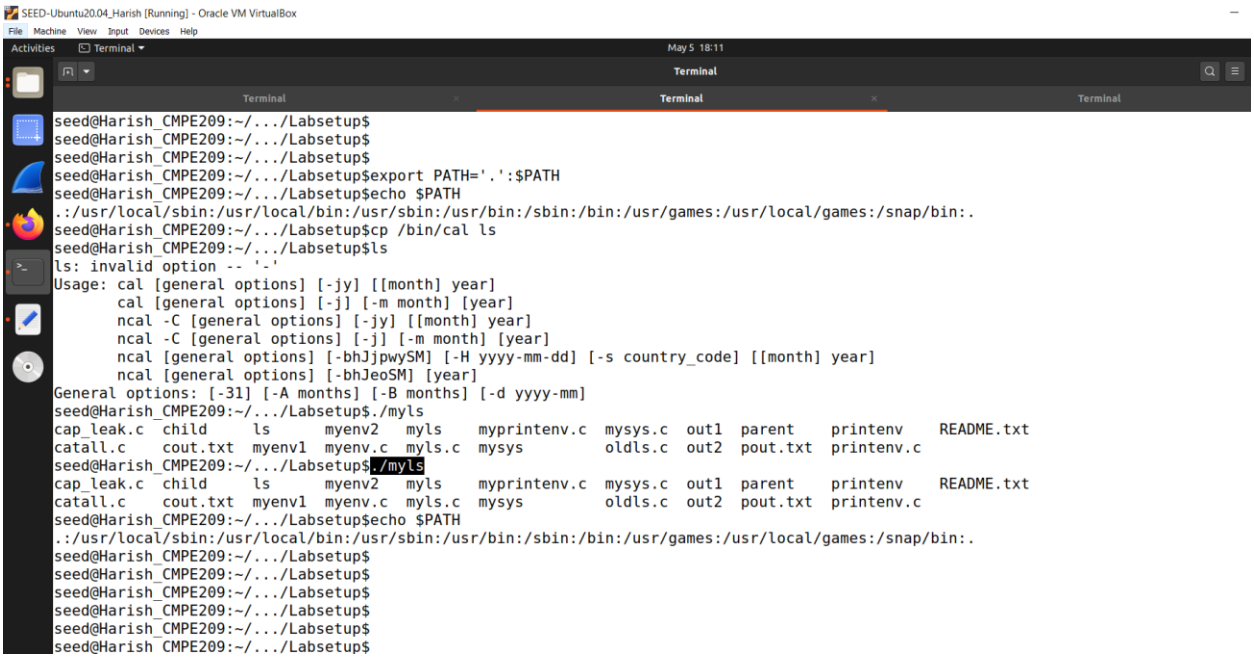

Figure 6.6

Even though, after changing it to the Set-UID program, it still displays the normal output. This is because we have given */bin/ls* in the code, and it goes to that location without going to any path in the PATH variable.

So, to make it go to the PATH variable, we make a change in the code. Figure 6.7 shows the changed *myls.c* code.

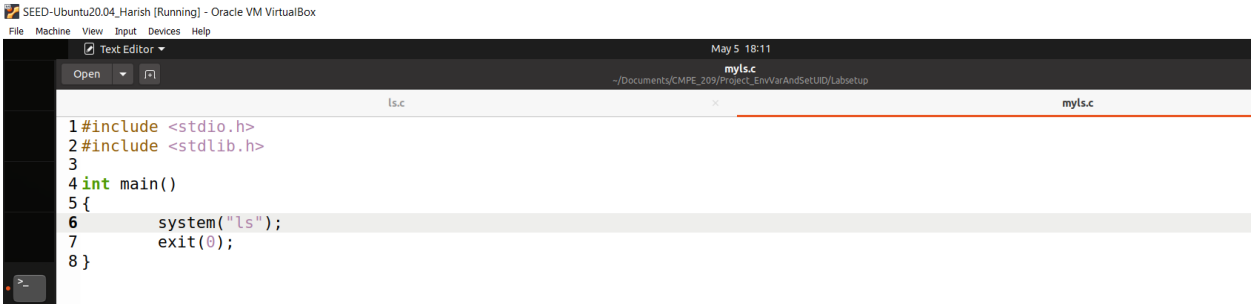

Figure 6.7

Here, in the code, we did not specify anything in the path before *ls* and we see what happens now.

Figure 6.8 shows the compilation of the above code in the normal terminal.

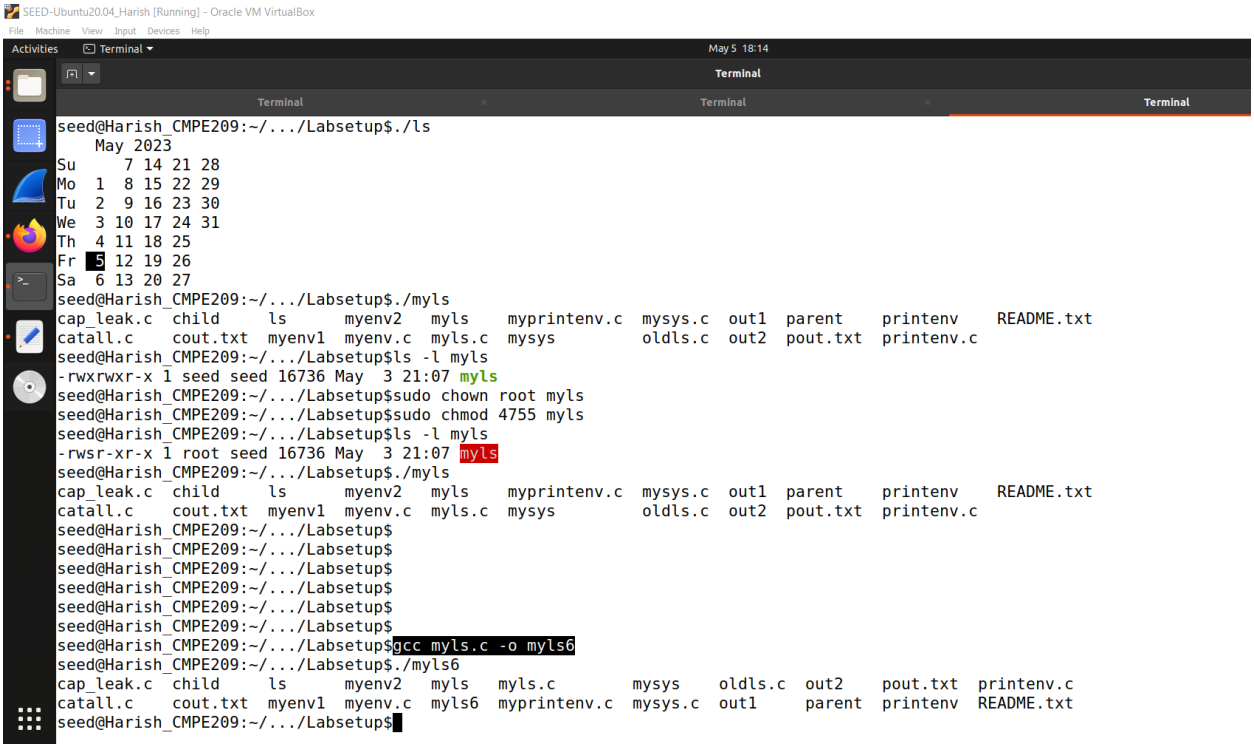

Figure 6.8

In the normal terminal after compiling and running *./myls6*, it will give the normal output.

So, for this reason, go to the manipulated terminal and run *./myls6*. The output is shown in the figure 6.9 below.

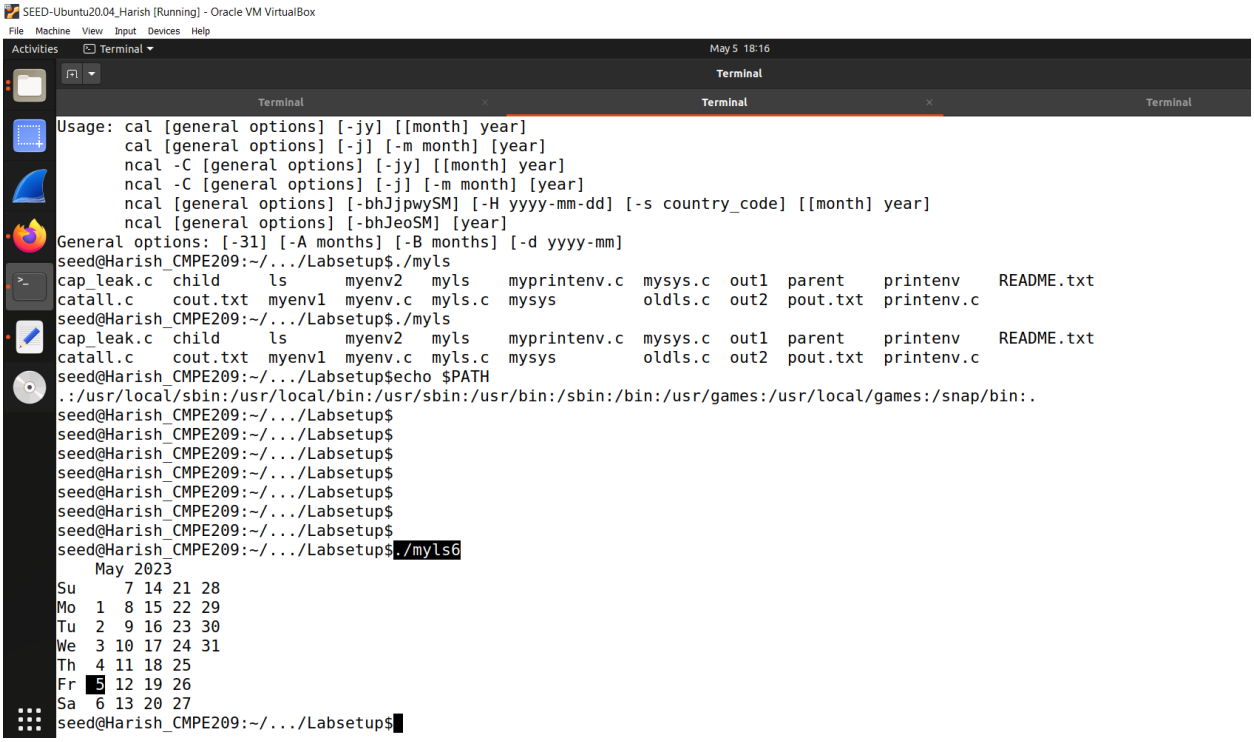

Figure 6.9

Now, if we see, the output is changed. It shows calendar output instead of the normal one. It calls fake *ls* as it searches the PATH variable now and there is a local folder in the PATH variable at the starting.

If it is a normal user program, it will be attacked.

How about if it is a Set-UID program?

For this, go to the normal terminal (3rd terminal) and change it to Set-UID program. Figure 6.10 shows the same.

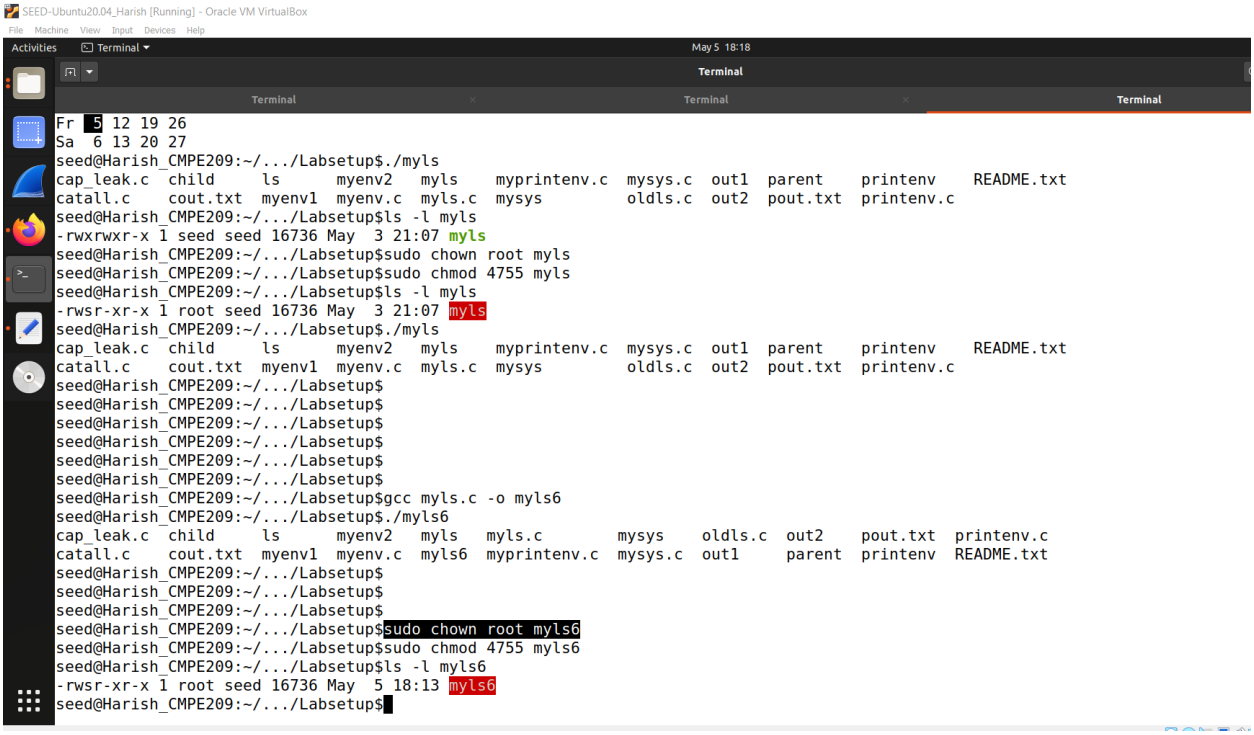

Figure 6.10

So, it is a Set-UID program owned by root.

Now, run *./myls6* in the manipulated terminal. Figure 6.11 shows the same.

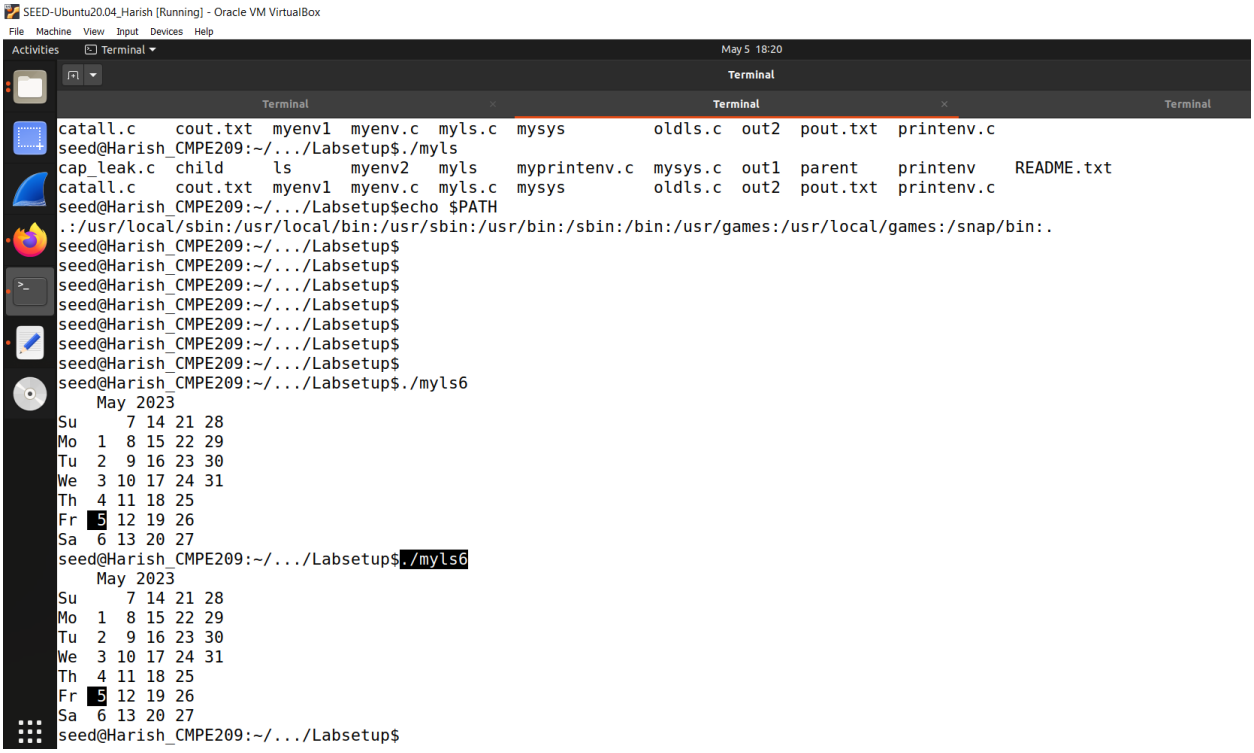

Figure 6.11

We can see that it is a Set-UID program and still accepts the manipulated PATH environment variable. So, this way the security problem or vulnerability is introduced into this program.

Can you get this Set-UID program to run your own malicious code, instead of /bin/ls?

Yes, we can get this Set-UID program to run our own malicious code, instead of /bin/ls because it has the fake ls i.e. calendar output.

If you can, is your malicious code running with the root privilege? Describe and explain your observations.

We can use the ID program in our normal environment. We can use the *id* command to show whether our process is privileged or not. We can use the id as a fake *ls.*

So, we remove our calendar fake *ls* and copy */bin/id* to *ls.* If we type ls now, it shows the file names.

If we run *./ls*, it shows the ids. Figure 6.12 shows the same.

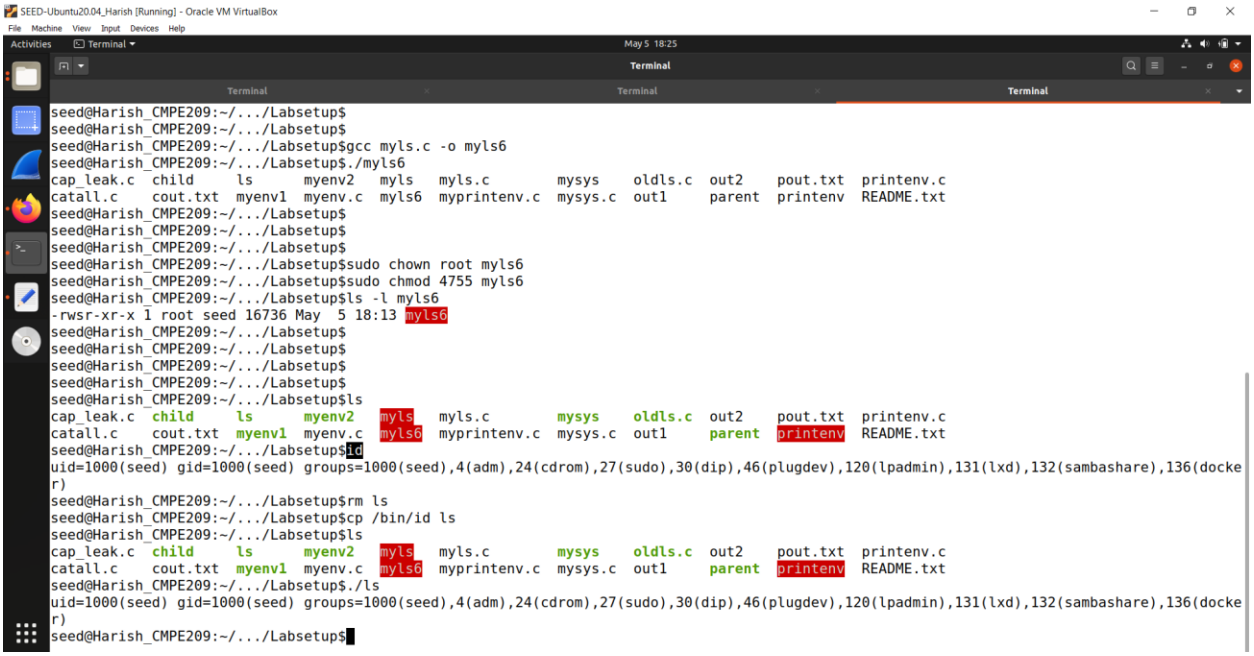

Figure 6.12

Go to malicious terminal and run the command *./myls6.* It prints out all the ids and it still a normal user. Figure 6.13 shows the same.

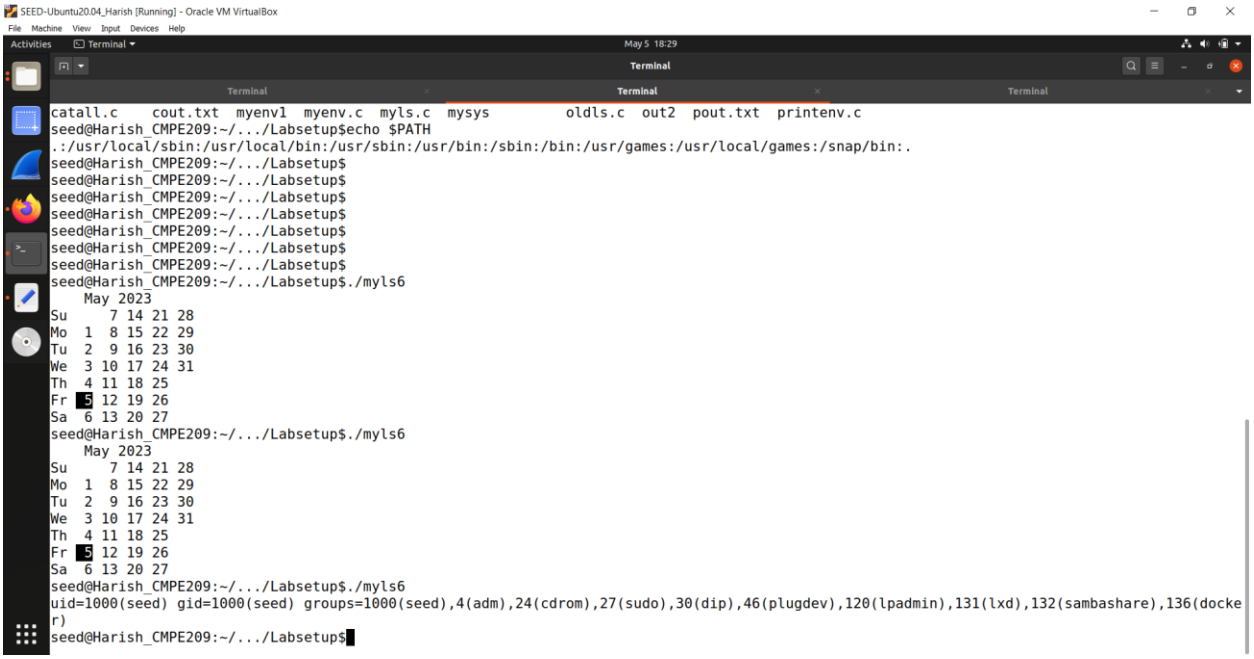

Figure 6.13

So, if it is a root privilege, we can run our malicious code. It is the protection provided by the operating system.

Note: The system(cmd) function executes the /bin/sh program first, and then asks this shell program to run the cmd command. In Ubuntu 20.04 (and several versions before), /bin/sh is actually a symbolic link pointing to /bin/dash. This shell program has a countermeasure that prevents itself from being executed in a Set-UID

process. Basically, if dash detects that it is executed in a Set-UID process, it immediately changes the effective user ID to the process's real user ID, essentially dropping the privilege. Since our victim program is a Set-UID program, the countermeasure in /bin/dash can prevent our attack. To see how our attack works without such a countermeasure, we will link /bin/sh to another shell that does not have such a countermeasure. We have installed a shell program called zsh in our Ubuntu 20.04 VM. We use the following commands to link /bin/sh to /bin/zsh: $ sudo ln -sf /bin/zsh /bin/sh.

In our normal terminal, let us link the /bin/sh to /bin/zsh. Figure 6.14 shows the same.

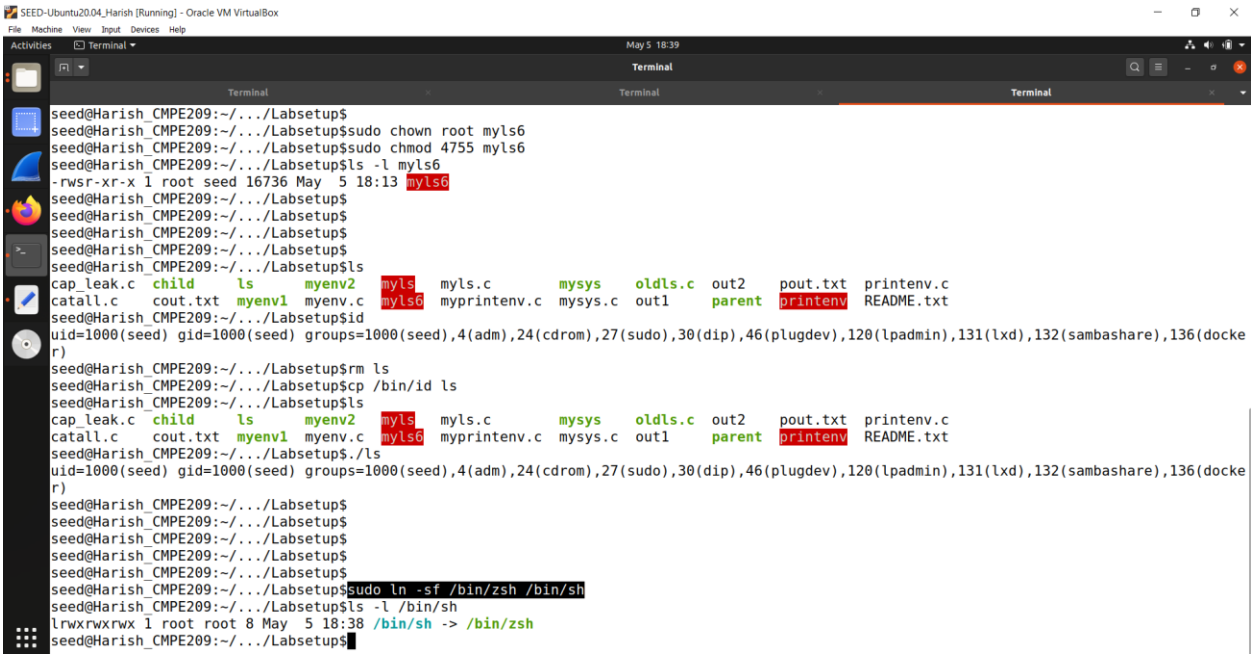

Figure 6.14

So, now it is linked to /bin/zsh. Go back to the manipulated terminal and run *./myls6* and we can see the *euid=0(root)*. It is shown in the figure 6.15.

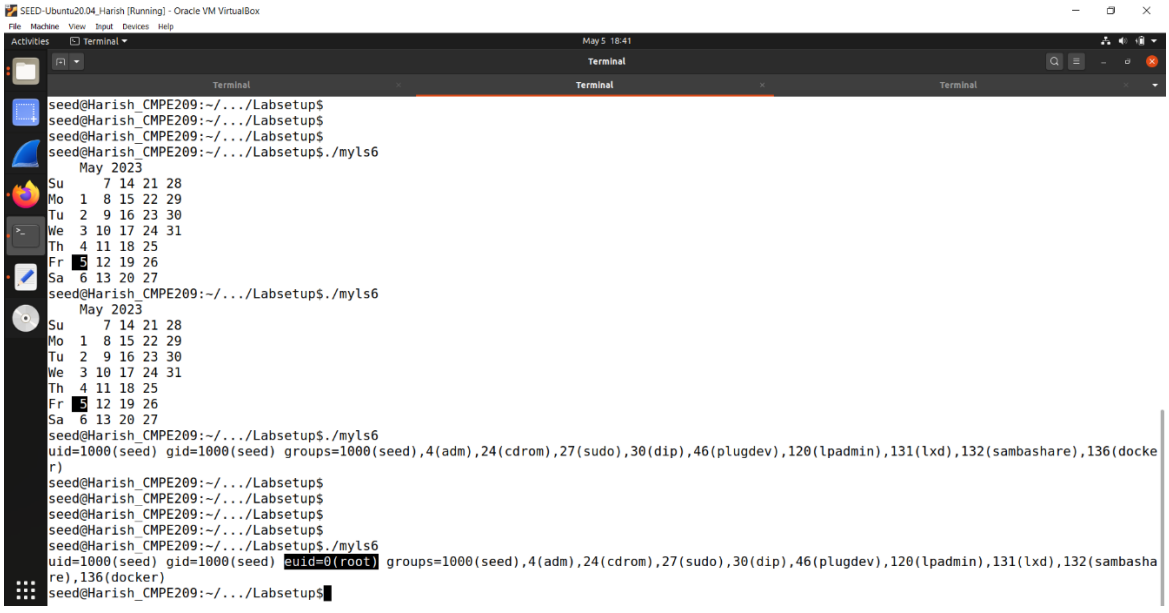

Figure 6.15

So ,it means we are able to run our malicious code with the root privilege.

As said, if the bin shell (/bin/sh)is linked to the /bin/dash, then we cannot run it with the root privilege and if it is linked to the zshell, we are able to run it with the root privilege.

We shall see whether we are able to run it if we link /bin/dash to /bin/sh. It means that we are turning the protection ON. It is run in the figure 6.16.

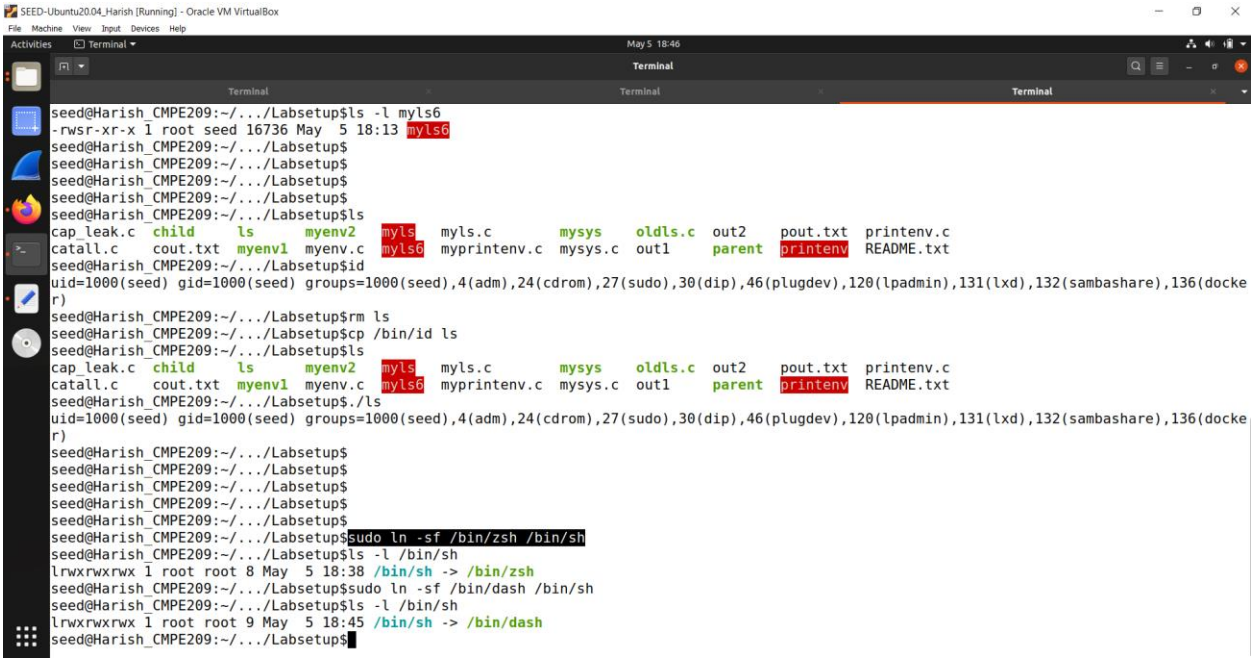

Figure 6.16

Go to the malicious terminal and run *./myls6.* We can see that the euid is gone in the figure 6.17 upon running it.

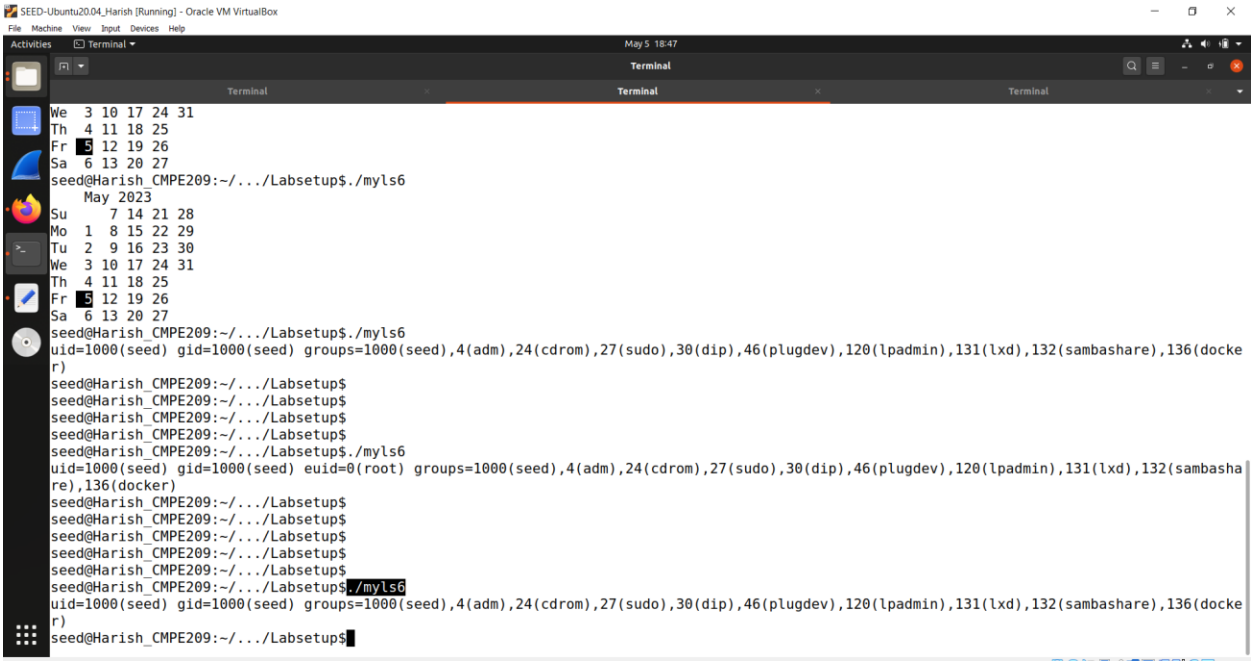

Figure 6.17

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

In this task, we study how Set-UID programs deal with some of the environment variables. Several environment variables, including LD PRELOAD, LD LIBRARY PATH, and other LD * influence the behavior of dynamic loader/linker. A dynamic loader/linker is the part of an operating system (OS) that loads (from persistent storage to RAM) and links the shared libraries needed by an executable at run time.

In Linux, ld.so or ld-linux.so, are the dynamic loader/linker (each for different types of binary). Among the environment variables that affect their behaviors, LD LIBRARY PATH and LD PRELOAD are the two that we are concerned in this lab. 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. In this task, we will only study LD PRELOAD.

Step 1. First, we will see how these environment variables influence the behavior of dynamic loader/linker when running a normal program. Please follow these steps:

1. Let us build a dynamic link library. Create the following program, and name it mylib.c. It basically overrides the sleep() function in libc:

Figure 7.1 shows the mylib.c code.

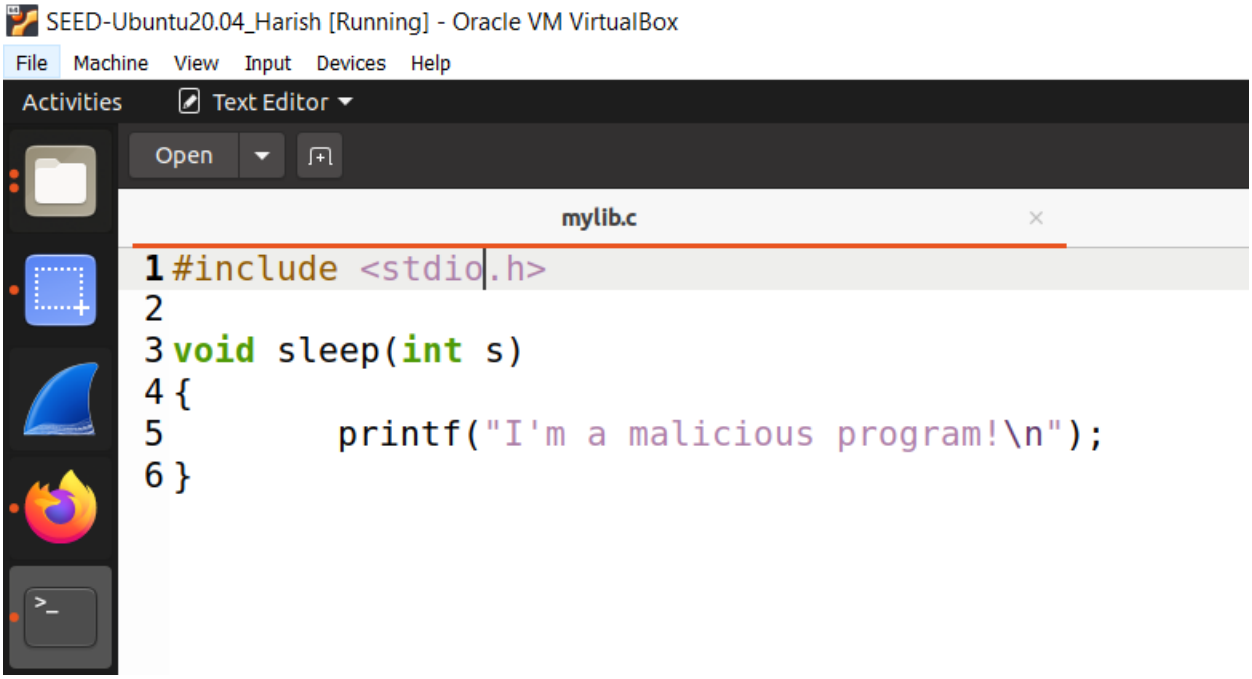

Figure 7.1

Here, it has a sleep function which overrides the sleep function in *libc*.

2. We can compile the above program using the following commands (in the -lc argument, the second character is `):

$ gcc -fPIC -g -c mylib.c

$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc

We compile the code in the normal terminal (we can close the manipulated terminal).

Figure 7.2 shows the compilation of mylib.c using the above-mentioned commands. We create the shared library using the second command.

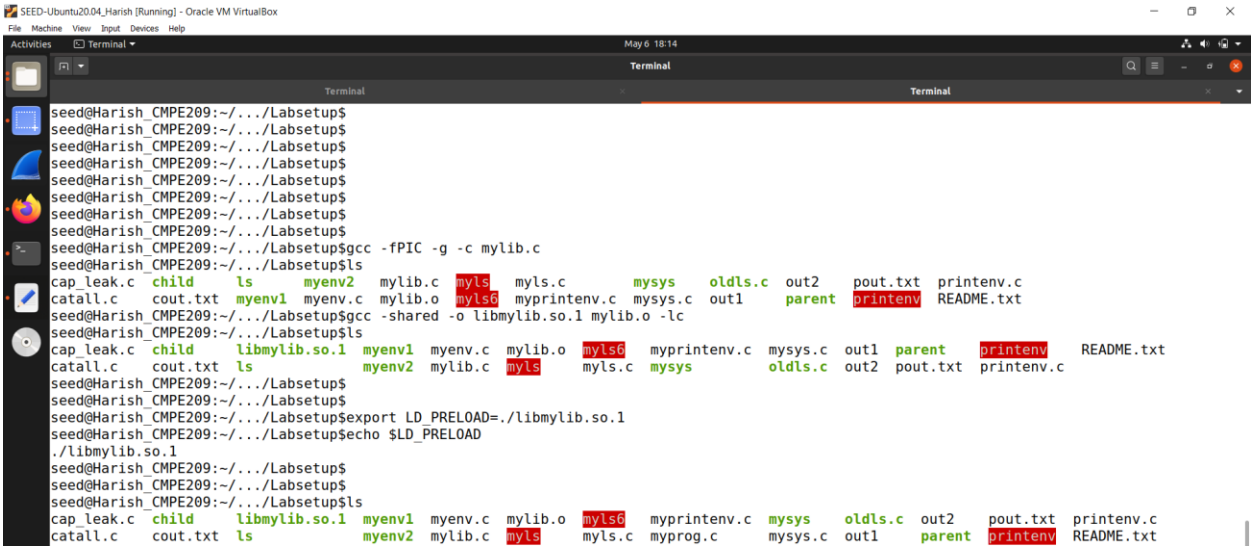

Figure 7.2

3. Now, set the LD PRELOAD environment variable:

$ export LD_PRELOAD=./libmylib.so.1

Here, we set the LD_PRELOAD environment variable.

Figure 7.2 shows the running of this command.

We can check the value using *echo* command.

4. Finally, compile the following program myprog, and in the same directory as the above dynamic link library libmylib.so.1:

```
/* myprog.c */

#include <unistd.h>

int main()

{

    sleep(1);

    return 0;

}
```

Now, since we manipulated the sleep function, it will call our malicious sleep function instead default sleep function that is present in the library.

Figure 7.3 shows the myprog.c code.

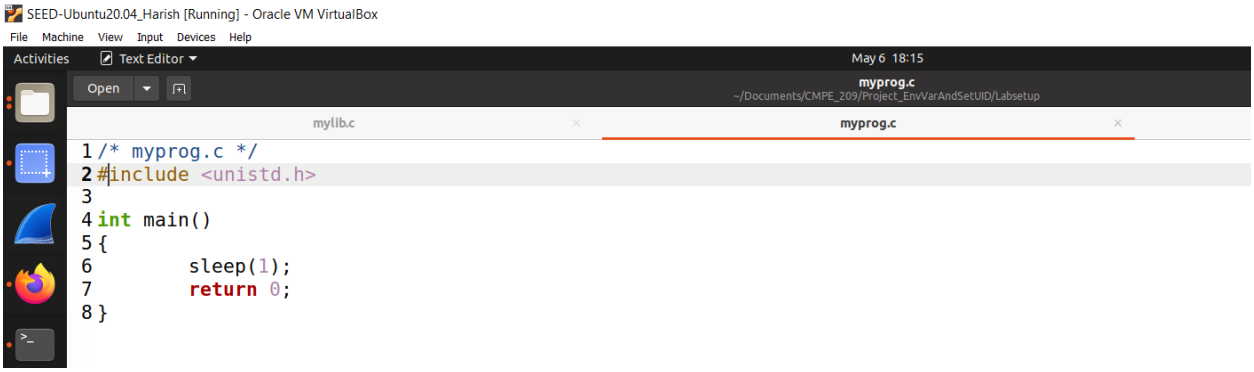

Figure 7.3

We compile the above program and run it. Figure 7.4 shows the compilation and running of the same commands.

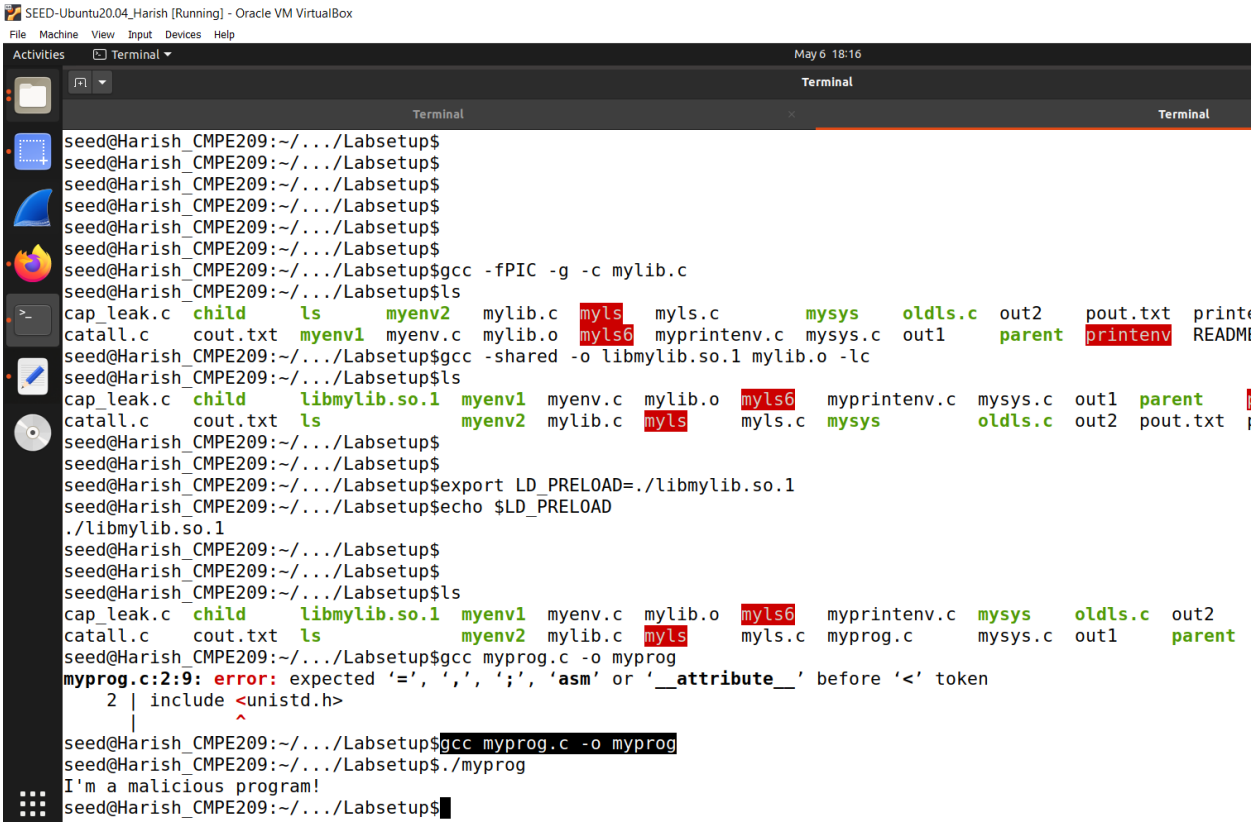

Figure 7.4

Step 2. After you have done the above, please run myprog under the following conditions, and observe what happens.

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

Here, we run the code using *./myprog* command.

After running we can see that it called the malicious program. It has print the message “I’m a malicious program!”.

Now, this is the way we do it using the normal user.

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

Now, we do it using the Set-UID program and run it as a normal user.

Here, change the owner to root and change the mode to 4755 and cross check the permissions using the respective command. So, it becomes a Set-UID program owned by the root.

Now, run the program as a normal user. using the command *./myprog.*

After running it, it just slept for one second and continued. From this, it can be inferred that the malicious program/library is not called. Figure 7.5 demonstrates the same.

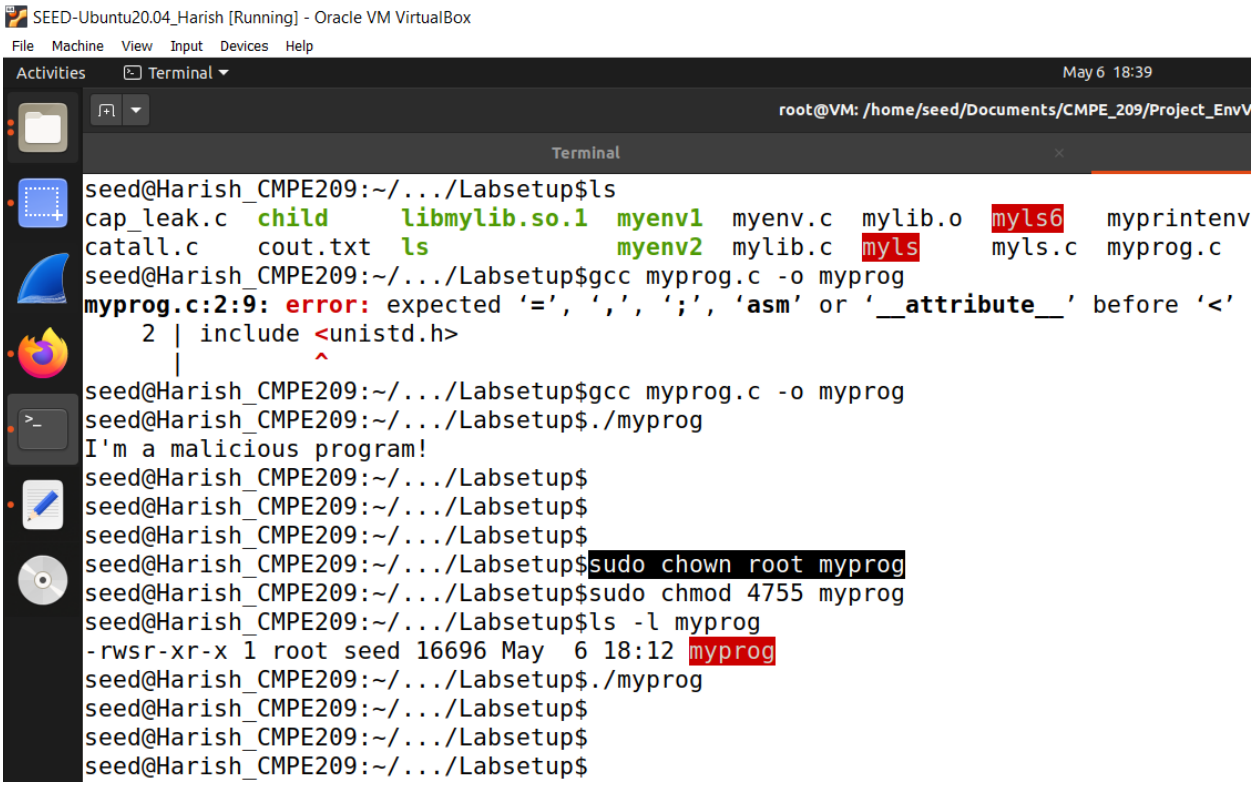

Figure 7.5

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

Now, we make it a Set-UID program and export the environment variable again in the root account and run it.

We switch to root using the command *sudo su.*

In this root account, LD_PRELOAD variable is empty. So, we again export it with the value *./libmylib.so.1.*

Now, run the program in this root account using the command *./myprog.* Figure 7.6 shows the same.

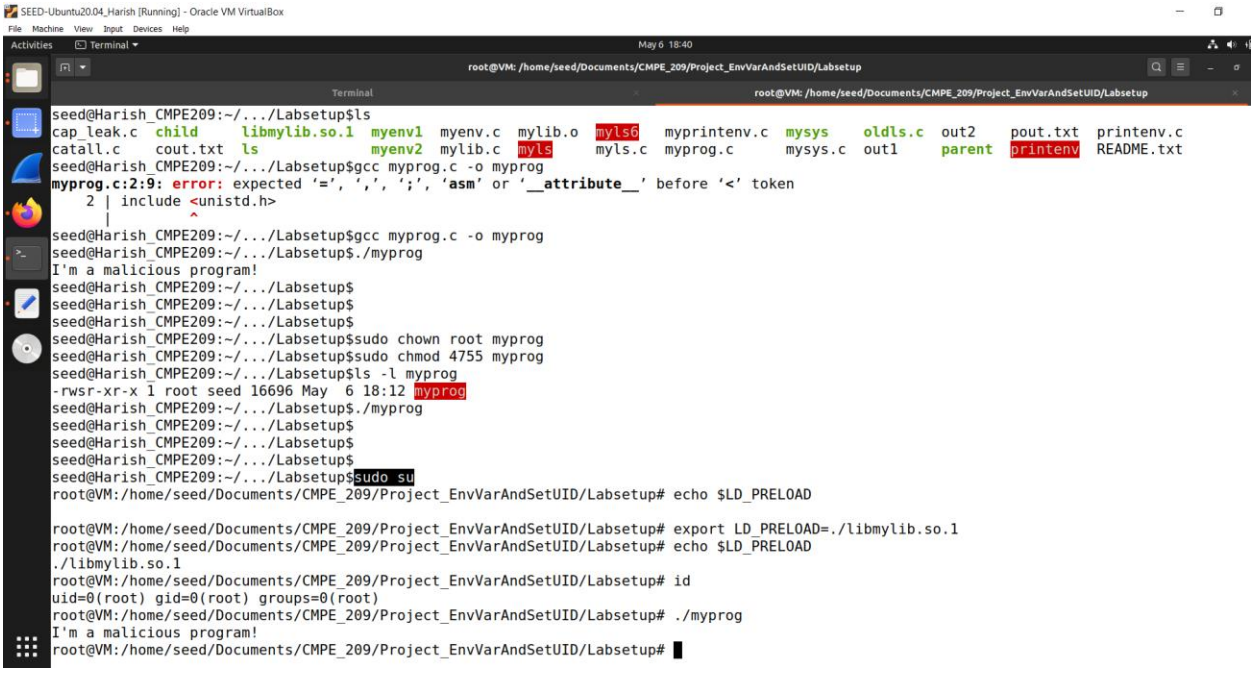

Figure 7.6

So, here we say that it has called the malicious program and it prints the message. So, we can conclude that if it is a root, then the malicious code is called.

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.

For this, we exit from the root account using the *exit* command and go back to the seed account.

Here, we need two user accounts.

We can add the user using the command *sudo adduser user1*.

So, we have created user1 and we change the owner to user1. Change the mode to 4755. Now it is a Set-UID program owned by the user1.

It is better to change the group to user1 as well and the command to do is *sudo chown user1:user1 myprog.*

Change the mode again to 4755.

Now, we run the program using the command *./myprog*. Figure 7.7 shows the running of the above commands.

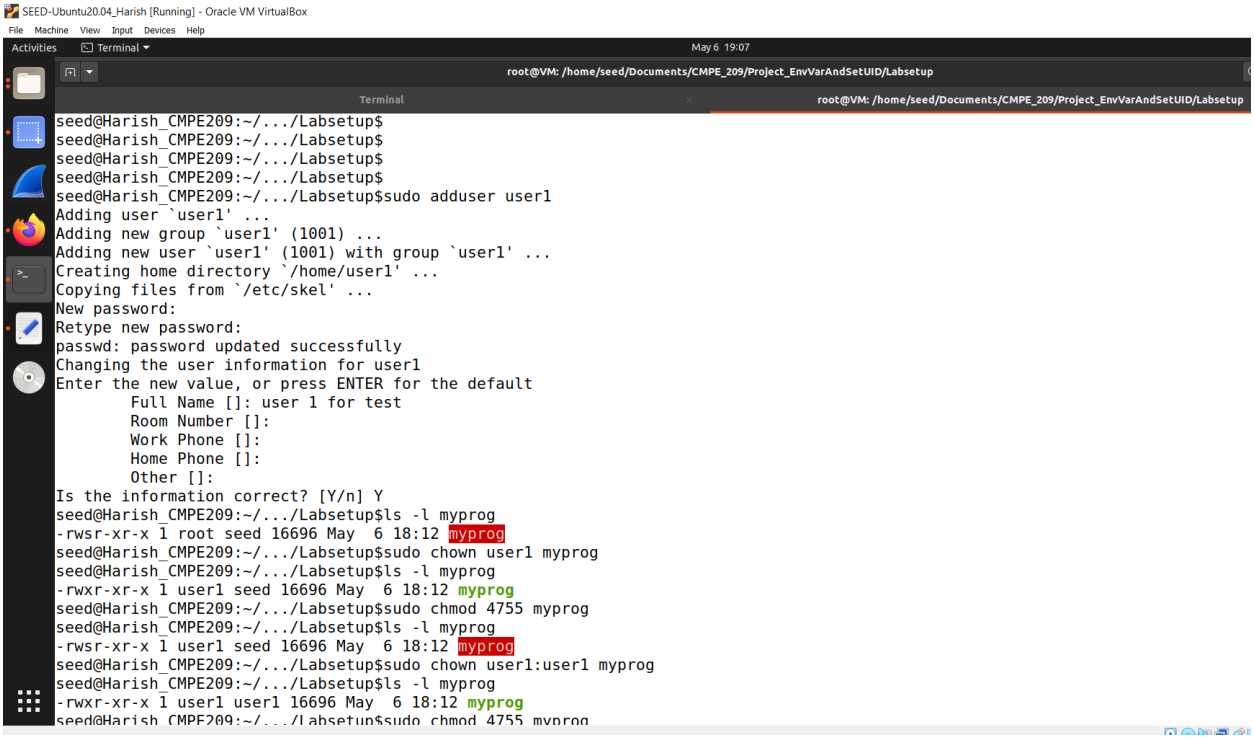

Figure 7.7

Figure 7.8 shows the continuation of the commands.

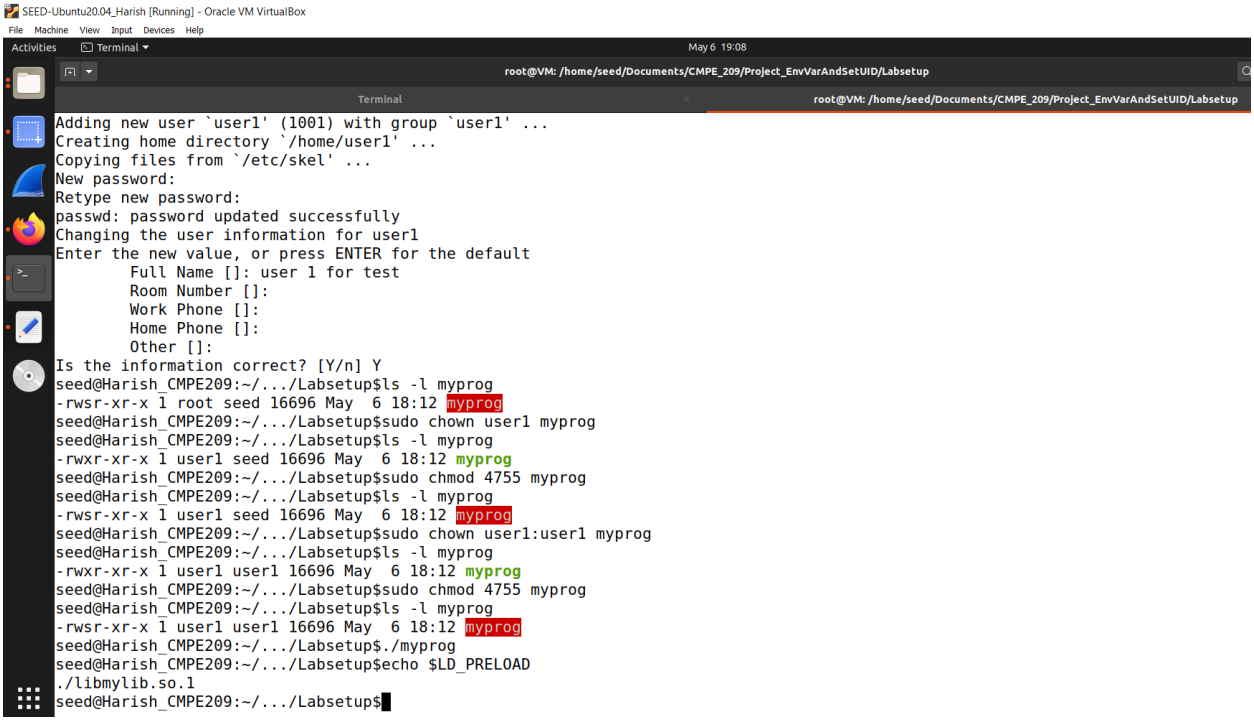

Figure 7.8

Here, we can see that it has called the normal sleep function and continued. It is not manipulated even though LD_PRELOAD is there. So, it is good that it is protected by the operating system.

Step 3. You should be able to observe different behaviors in the scenarios described above, even though you are running the same program. You need to figure out what causes the difference. Environment variables play a role here. Please design an experiment to figure out the main causes and explain why the behaviors in Step 2 are different. (Hint: the child process may not inherit the LD * environment variables).

When we run it the child process, LD_PRELOAD is not passed into the environment of this myprog of the Set-UID program

If it is a normal program, the LD_PRELOAD environment variable is passed into the environment and the malicious code will be called.

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

Although system() and execve() can both be used to run new programs, system() is quite dangerous if used in a privileged program, such as Set-UID programs. We have seen how the PATH environment variable affect the behavior of system(), because the variable affects how the shell works. execve() does not have the problem, because it does not invoke shell. Invoking shell has another dangerous consequence, and this time, it has nothing to do with environment variables. Let us look at the following scenario.

Bob works for an auditing agency, and he needs to investigate a company for a suspected fraud. For the investigation purpose, Bob needs to be able to read all the files in the company's Unix system; on the other hand, to protect the integrity of the system, Bob should not be able to modify any file. To achieve this goal, Vince, the superuser of the system, wrote a special set-root-uid program (see below), and then gave the executable permission to Bob. This program requires Bob to type a file name at the command line, and then it will run /bin/cat to display the specified file. Since the program is running as a root, it can display any file Bob specifies. However, since the program has no write operations, Vince is very sure that Bob cannot use this special program to modify any file.

We write a program which is named as *catall.c*. Figure 8.1 shows the code.

Figure 8.1

Arguments are passed into the program via the command line.

Step 1: Compile the above program, make it a root-owned Set-UID program. The program will use system() to invoke the command. If you were Bob, can you compromise the integrity of the system? For example, can you remove a file that is not writable to you?

Let us first compile it as a normal user and run. Run the command *./catall catall.c* to see the contents of the source code.

If we want to see the contents of these system protected files like for example *shadow*, we can run the command *./catall /etc/shadow.* It is shown in the figure 8.2

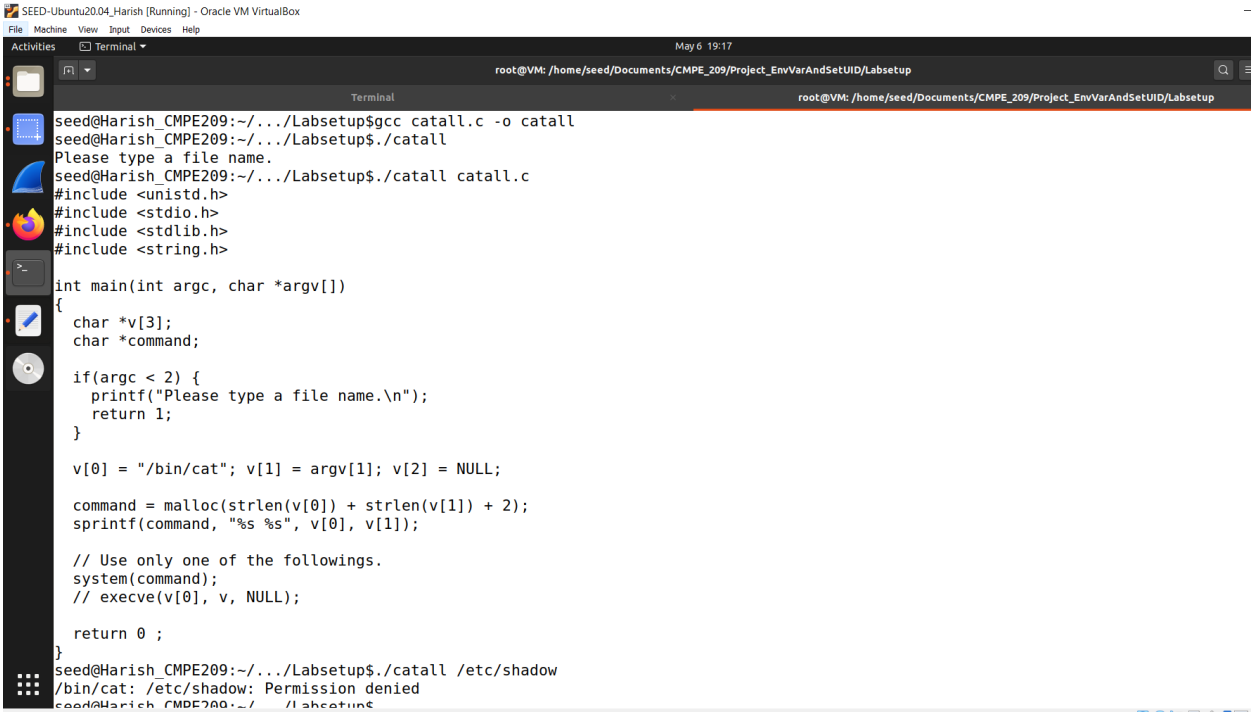

Figure 8.2

Here, we can see that after we had run the command ./catall /etc/shadow, we see the message as "Permission denied."

Here, we want Bob to use this catall to see any system files. So, we change it to a Set-UID program.

After it is a Set-UID program owned by the root, run it as a normal user (Seed is the normal user). We get the same permission denied message even though we run it a Set-UID program.

It is because /bin/sh is linked to /bin/dash and dash has protection.

So, let us change it to z shell using the command *sudo ln -sf /bin/zsh /bin/sh.*

After changing it to /bin/zsh, run the *./catall /etc/shadow* command.

All these commands are shown in the figure 8.3.

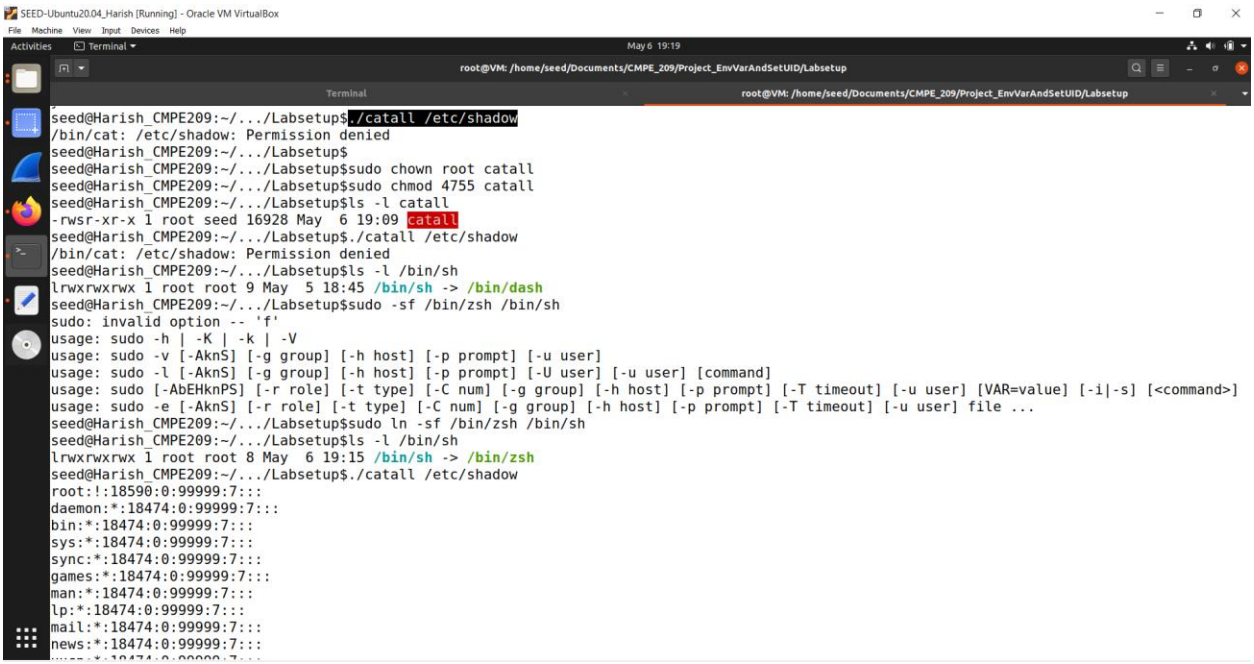

Figure 8.3

Figure 8.4 and 8.5 is the continuation of the output.

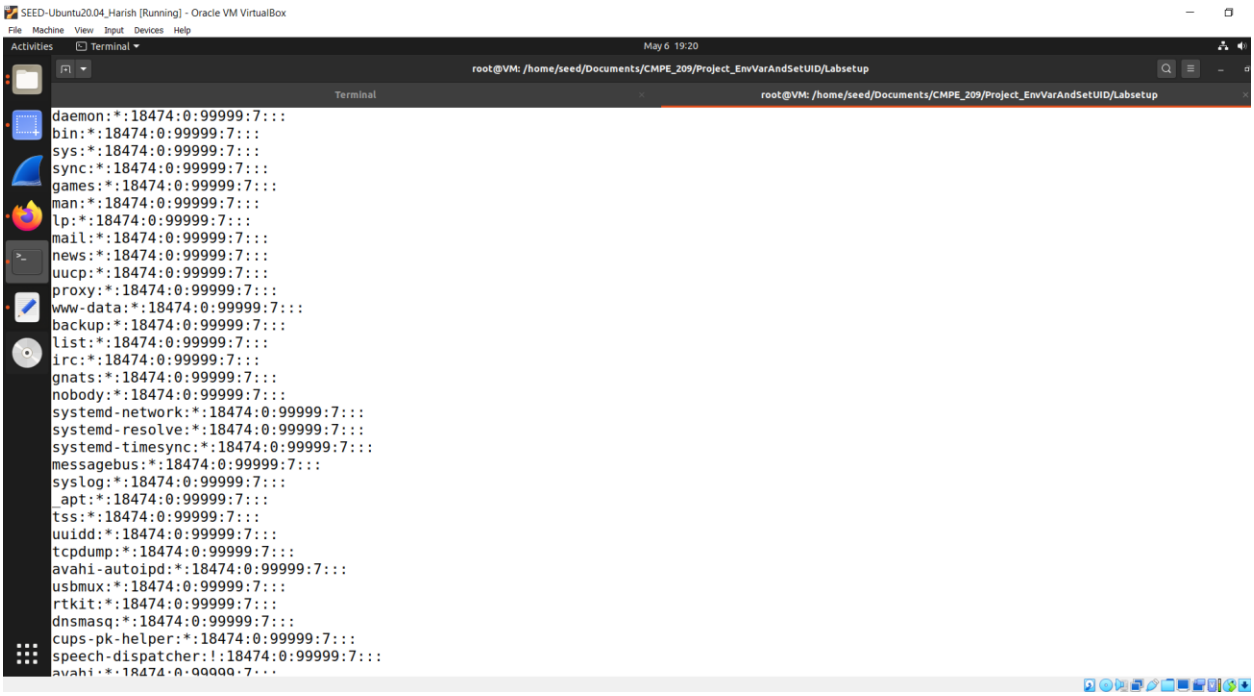

Figure 8.4

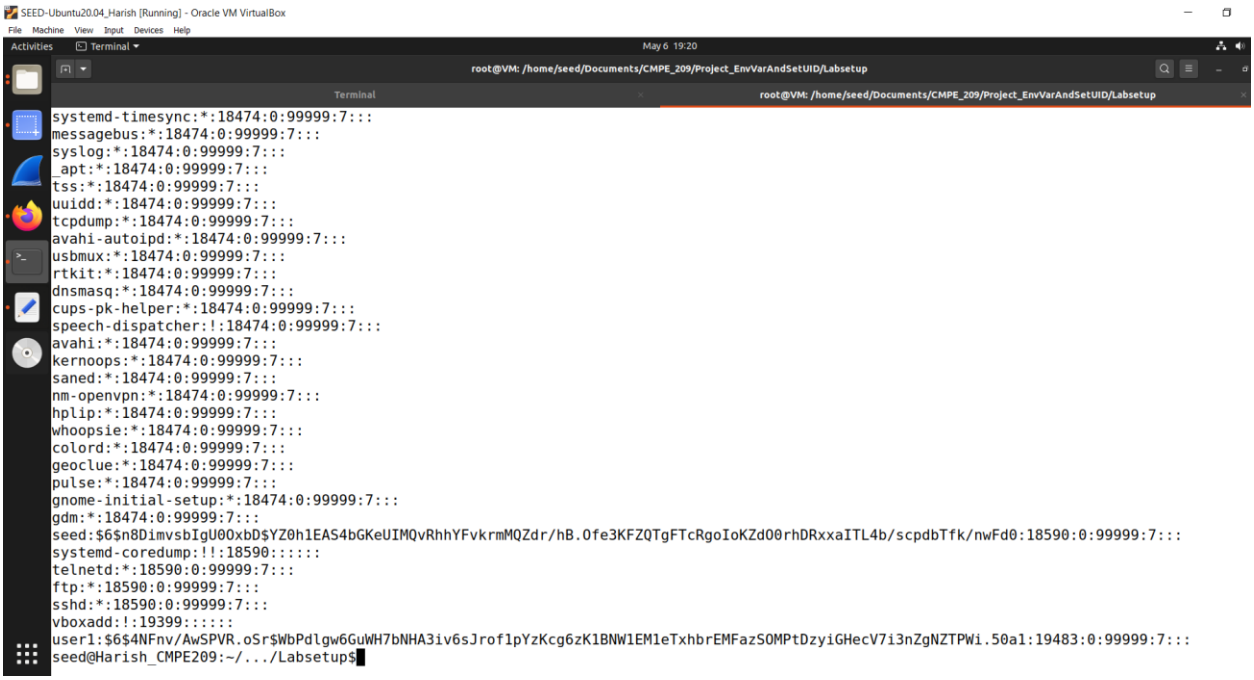

Figure 8.5

After we run the *./catall /etc/shadow* command, this time, as Bob, normal user we will be able to see any system protected files but it is designed only to read the contents.

If we see the source code, it uses *cat* which means we can only read them.

Now, go to new tab and let us create a file that is protected and cannot be deleted by a normal user. Let us take a file with the name *zzz*. It is owned by root and a normal user cannot delete it. So, we check if we can modify it.

We run the command *./catall "/etc/zzz;/bin/zsh"*. Figure 8.6 shows the same.

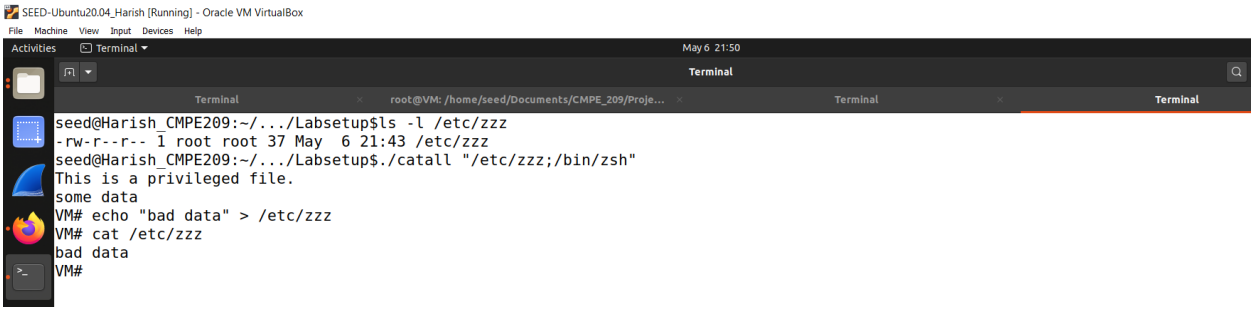

Figure 8.6

So, after running this command it had called /etc/zzz and print the contents that are inside the zzz file.

It had also invoked /bin/zsh and it took us to the root shell.

With root shell we can do anything, which means certainly we are able to compromise the entire system.

Let us try to modify the data as shown in the above figure. We run the command *echo "bad data" > /etc/zzz* and we see that the content in the file is modified.

So, we can say that Bob is able to compromise the integrity of the system. So, by just adding another parameter, we manipulated the command line parameter.

It exploits the system invoke because it will call the bin shell first and interpret the parameters passed as command line arguments and when we separate it with semicolon, it will assume it as two arguments.

So, when it sees first command it concatenates it with the /bin/cat and forms /bin/cat/etc/zzz to get a single command to show the contents of the file.

Then it takes as the second command which is after semicolon, and we get into root shell.

Step 2: Comment out the system(command) statement, and uncomment the execve() statement; the program will use execve() to invoke the command. Compile the program and make it a root-owned Set-UID. Do your attacks in Step 1 still work? Please describe and explain your observations.

Here, we comment out the system(command) line and uncomment the execve() line.

Exit from the shell. Make sure that we still have the vulnerable /bin/zsh linked to /bin/sh.

Modified code is shown in the figure 8.7.

Figure 8.7

Compile the above code using the command *gcc catall.c -o catall2* and change it to Set-UID program owned by the root.

After doing the above, run it to see whether we can attack it using the command *./catall2 "/etc/zzz;/bin/zsh"*.

It is shown in the figure 8.8.

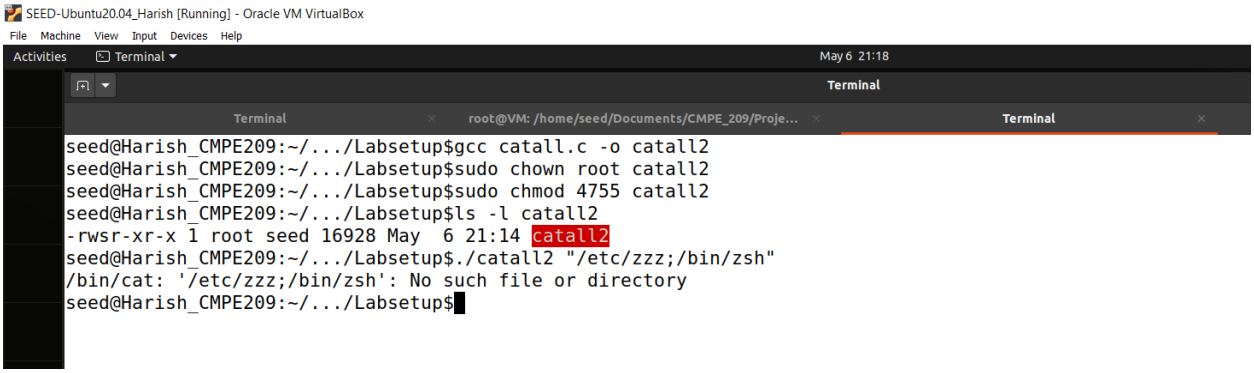

Figure 8.8

So, we see that it says "No such file or directory". Here, it interprets "/etc/zzz;/bin/zsh" as data and in the previous case the second part was interpreted as the command. So, we can conclude that we cannot attack using this technique (the attacks in Step 1 does not work).

Task 9: Capability Leaking:

To follow the Principle of Least Privilege, Set-UID programs often permanently relinquish their root privileges if such privileges are not needed anymore. Moreover, sometimes, the program needs to hand over its control to the user; in this case, root privileges must be revoked. The setuid() system call can be used to revoke the privileges. According to the manual, "setuid() sets the effective user ID of the calling process. If the effective UID of the caller is root, the real UID and saved set-user-ID are also set". Therefore, if a Set-UID program with effective UID 0 calls setuid(n), the process will become a normal process, with all its UIDs being set to n.

When revoking the privilege, one of the common mistakes is capability leaking. The process may have gained some privileged capabilities when it was still privileged; when the privilege is downgraded, if the program does not clean up those capabilities, they may still be accessible by the non-privileged process. In other words, although the effective user ID of the process becomes non-privileged, the process is still privileged because it possesses privileged capabilities.

Compile the following program, change its owner to root, and make it a Set-UID program. Run the program as a normal user. Can you exploit the capability leaking vulnerability in this program? The goal is to write to the /etc/zzz file as a normal user.

Create a file with the name *zzz* using the command *sudo cat > /etc/zzz*. When we run it, we get a message saying "Permission denied". This is the protection provided by the operating system.

So, instead of that command we run the following command:

sudo gedit /etc/zzz

This opens up a new empty file and write some content in it. Figure 9.1 shows the zzz file.

Figure 9.1

If we check the permissions, we can see that it is owned by root and can be read by anyone but not write it.

So, we can use *cat* command to show the contents in the file. It is shown in the figure 9.2.

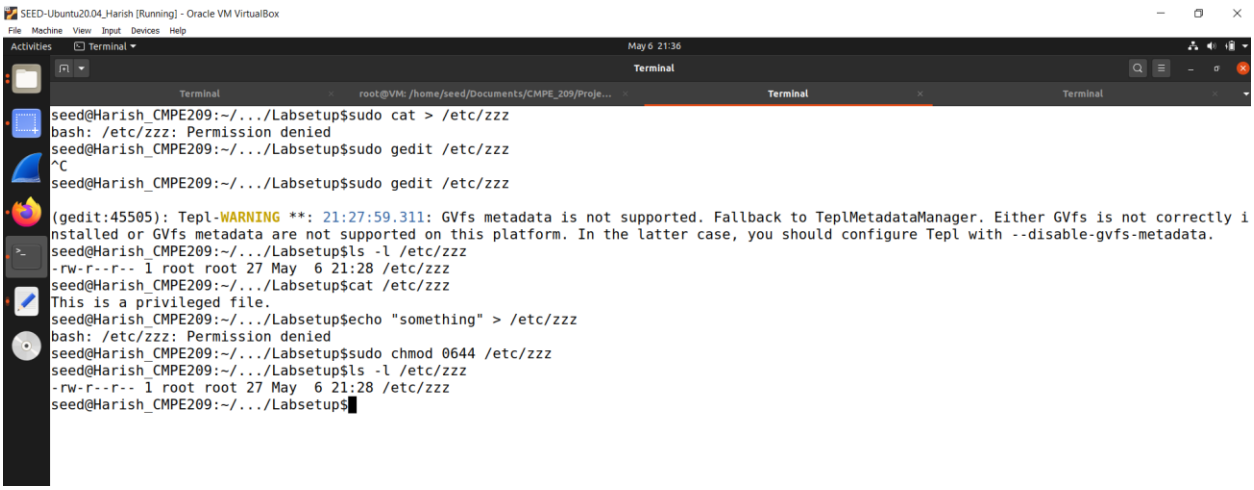

Figure 9.2

Here, we can see that if we try to write something into the file, we get an error saying “Permission denied”.

Now, we change the mode to 0644 and it is interpreted as -rw-r--r--.

Let us take the provided code that is named as cap_leak.c. It is shown in the figure 9.3.

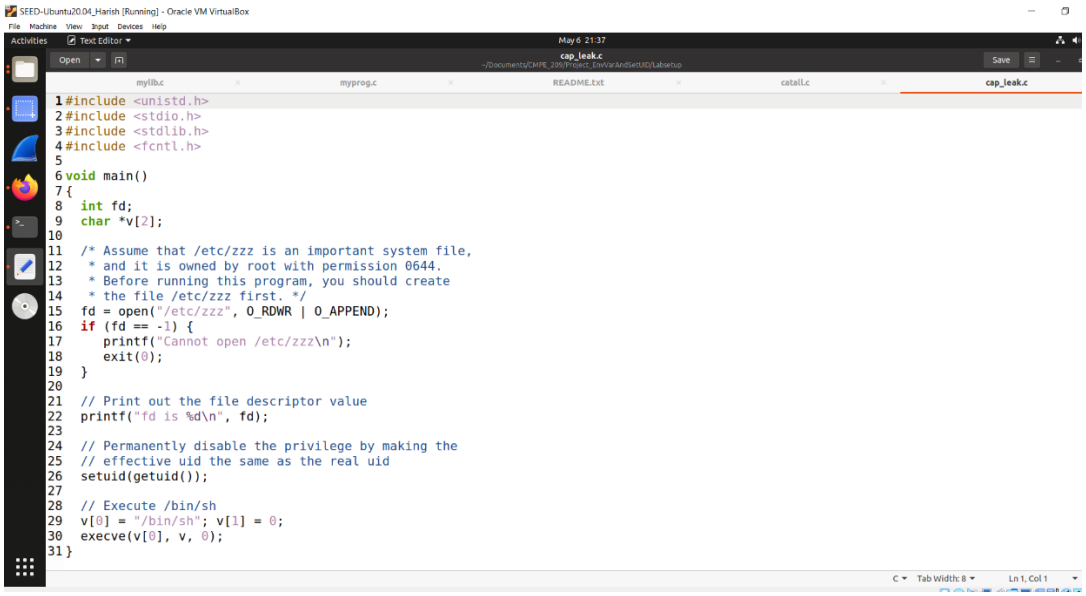

Figure 9.3

Compile this program and run it to see whether the normal user can write something into that file. Here, after we dropped the privilege, we did not close the file, which means that the privilege is still possessed by the process. So, this is called *Capability Leaking.*

Compile the code using the command *gcc cap_leak.c -o cap_leak.*

Make it as a Set-UID program owned by the root.

After verifying the permissions, run it using the command *./cap_leak.* It is shown in the figure 9.4 and 9.5.

Figure 9.4

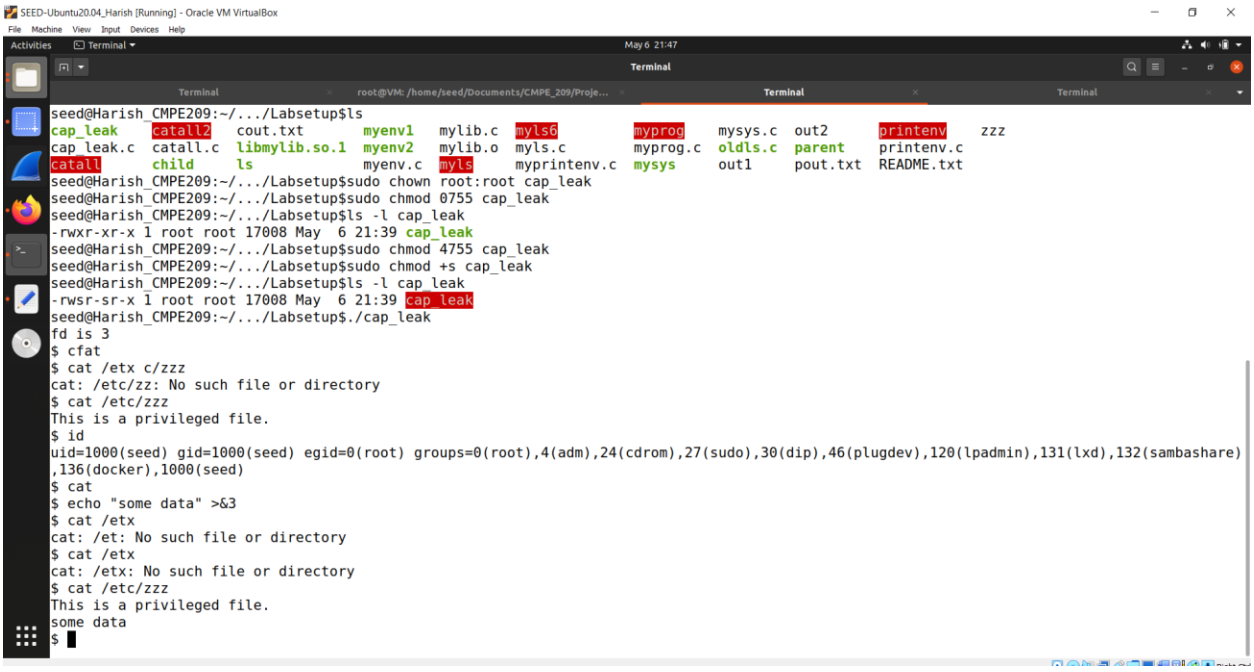

Figure 9.5

After we run it, we see that the file descriptor is 3 but it is not closed, and it also opened a shell for us.

Here, it shows $ symbol instead of # key. $ means that our privilege is dropped to normal user and # means it is a root user.

So, if we run *cat /etc/zzz* as a normal user, it says "This is a privileged file."

After that, we check whether we can write some data into the file.

We are able to write data into the file using the file descriptor and is shown in the figure 9.1 using the zzz file. The content that got added is "some data". So, this is the vulnerability due to the capability leaking. So, we can conclude that we can exploit the capability leaking vulnerability in this program.

Insight about this attack in terms of defense methods:

Environment variables are a way to store information that can be used by programs running on a system. They are typically used to store configuration information, such as paths to libraries or system settings. Set-UID programs, on the other hand, are programs that are executed with the privileges of their owner, rather than the privileges of the user who runs them.

An attacker can exploit the combination of environment variables and set-UID programs to gain elevated privileges on a system. The attacker can manipulate the environment variables to execute malicious code or to modify the behavior of the set-UID program.

To defend against such attacks, there are several measures that can be taken:

1. Avoid using environment variables in set-UID programs: This is the most effective defense measure. By avoiding the use of environment variables in set-UID programs, you eliminate the risk of an attacker manipulating them.
2. Validate input: It is important to validate all input to set-UID programs, including environment variables. This can help prevent attackers from exploiting vulnerabilities in the program.
3. Use secure coding practices: Set-UID programs should be designed and coded with security in mind. This includes practices such as input validation, error checking, and boundary checking.
4. Limit privileges: It is important to limit the privileges of set-UID programs as much as possible. This can help minimize the damage that an attacker can do if they are able to exploit the program.
5. Use access controls: Access controls, such as file permissions and user/group permissions, can help limit the damage that an attacker can do if they are able to exploit a set-UID program.
6. Regularly update software: Regularly updating software, including set-UID programs, can help mitigate vulnerabilities that could be exploited by attackers.

By implementing these defense measures, you can help reduce the risk of environment variable and set-UID program attacks.

Conclusion:

In this project, by performing this attack, I learned how can an attack be performed by changing the environment variables that are present in the system. I learned various of attacking using environment variables and defense methods as well.

Appendix:

Task 1: No code.

Task 2:

***myprintenv.c* –** Default Code:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

extern char **environ;

void printenv()
{
  int i = 0;
  while (environ[i] != NULL) {
     printf("%s\n", environ[i]);
     i++;
  }
}

void main()
{
  pid_t childPid;
  switch(childPid = fork()) {
    case 0:  /* child process */
       printenv();
       exit(0);
    default:  /* parent process */
        //printenv();
       exit(0);
```

```
  }
}
```

***myprintenv.c* –** Modified Code:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

extern char **environ;

void printenv()
{
  int i = 0;
  while (environ[i] != NULL) {
     printf("%s\n", environ[i]);
     i++;
  }
}

void main()
{
  pid_t childPid;
  switch(childPid = fork()) {
    case 0:  /* child process */
      //printenv();
      exit(0);
    default:  /* parent process */
       printenv();
      exit(0);
  }
```

```
}
```

Task 3:

myenv.c **–** Default Code:

```
#include <unistd.h>

extern char **environ;

int main()
{
  char *argv[2];

  argv[0] = "/usr/bin/env";
  argv[1] = NULL;

  execve("/usr/bin/env", argv, NULL);

  return 0 ;
}
```

myenv.c **–** Modified Code:

```
#include <unistd.h>

extern char **environ;

int main()
{
  char *argv[2];

  argv[0] = "/usr/bin/env";
```

```
 argv[1] = NULL;

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

 return 0 ;
}
```

Task 4:

mysys.c :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
	system("/usr/bin/env");
	return 0 ;
}
```

Task 5:

printenv.c :

```
#include <stdio.h>
#include <stdlib.h>

extern char **environ;
int main()
{
	int i = 0;
	while (environ[i] != NULL) {
		printf("%s\n", environ[i]);
		i++;
```

```
        }
}
```

Task 6:

myls.c – First code:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        system("/bin/ls");
        exit(0);
}
```

myls.c – Second code:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        system("ls");
        exit(0);
}
```

Task 7:

mylib.c :

```
#include <stdio.h>

void sleep(int s)
{
```

```
    printf("I'm a malicious program!\n");
}
```

myprog.c :

```
/* myprog.c */
#include <unistd.h>

int main()
{
    sleep(1);
    return 0;
}
```

Task 8:

catall.c – Default Code***:***

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.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 ;
}
```

catall.c – Modified Code***:***

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.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 ;
}
```

zzz file:

```
This is a privileged file.
some data
```

Task 9:

zzz file:

```
This is a privileged file.
some data
```

cap_leak.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

void main()
{
 int fd;
 char *v[2];
```

```
 /* Assume that /etc/zzz is an important system file,
  * and it is owned by root with permission 0644.
  * Before running this program, you should create
  * the file /etc/zzz first. */
 fd = open("/etc/zzz", O_RDWR | O_APPEND);
 if (fd == -1) {
    printf("Cannot open /etc/zzz\n");
    exit(0);
 }

 // Print out the file descriptor value
 printf("fd is %d\n", fd);

 // Permanently disable the privilege by making the
 // effective uid the same as the real uid
 setuid(getuid());

 // Execute /bin/sh
 v[0] = "/bin/sh"; v[1] = 0;
 execve(v[0], v, 0);
}
```