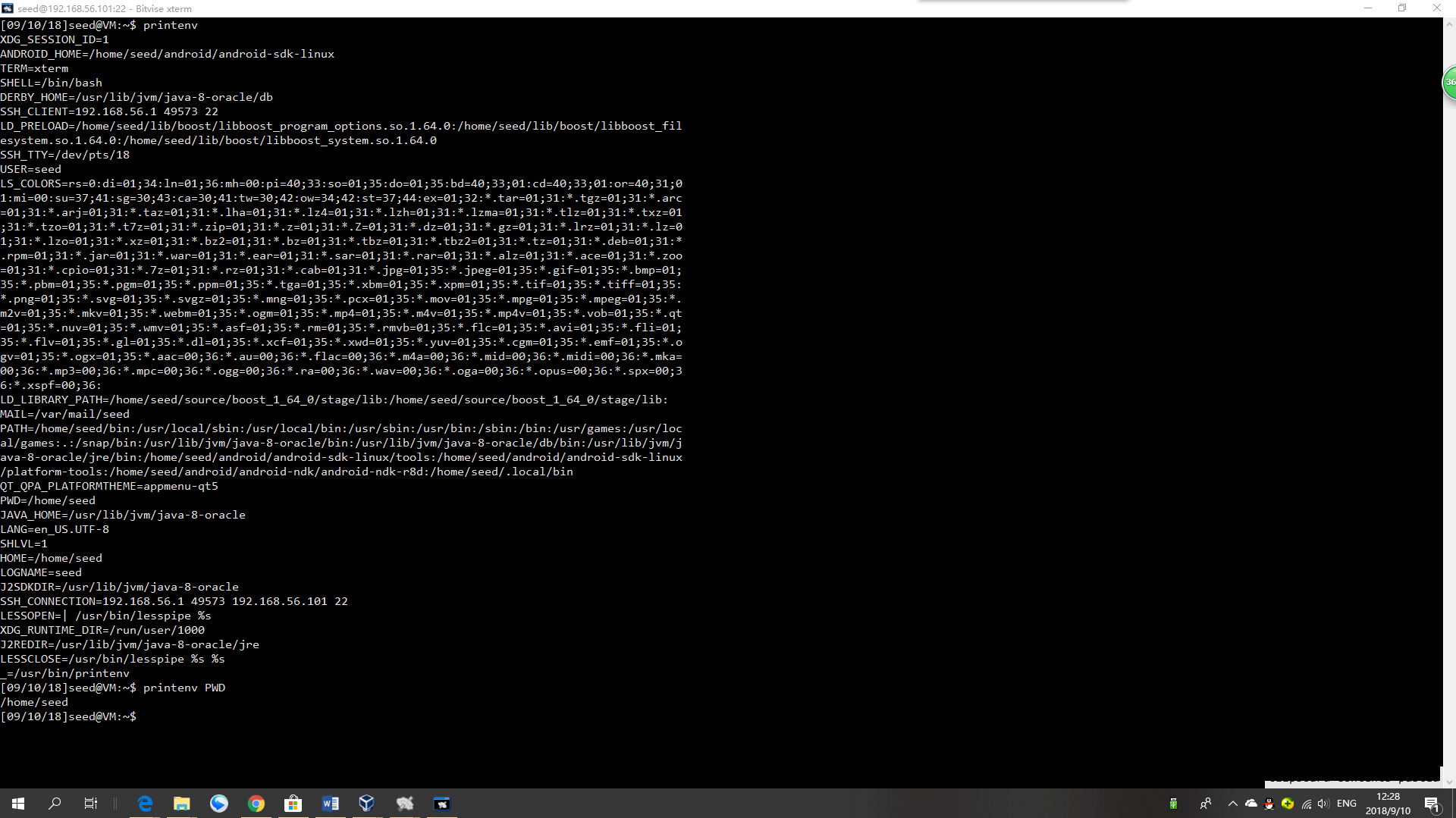
## Environment Variable and Set-UID Program Lab

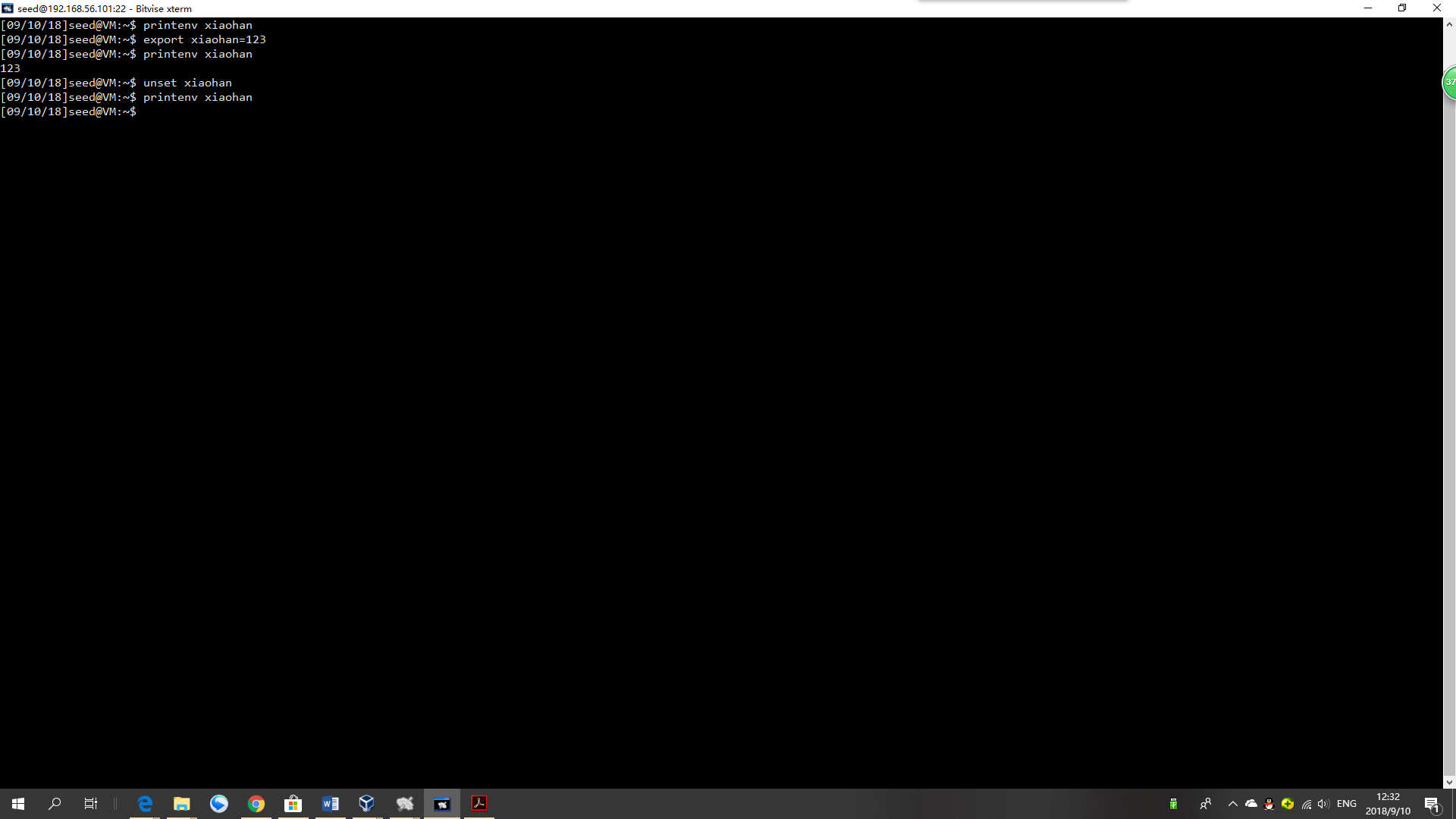
### Task 1: Manipulating Environment Variables

Ptintenv:



Observation:

As we can see, after running the printenv command, all system variables. And the printenv PWD command can directly find the system variable of PWD.



[09/10/18]seed@VM:~$ printenv xiaohan

[09/10/18]seed@VM:~$ export xiaohan=123

[09/10/18]seed@VM:~$ printenv xiaohan

123

[09/10/18]seed@VM:~$ unset xiaohan

[09/10/18]seed@VM:~$ printenv xiaohan

[09/10/18]seed@VM:~$

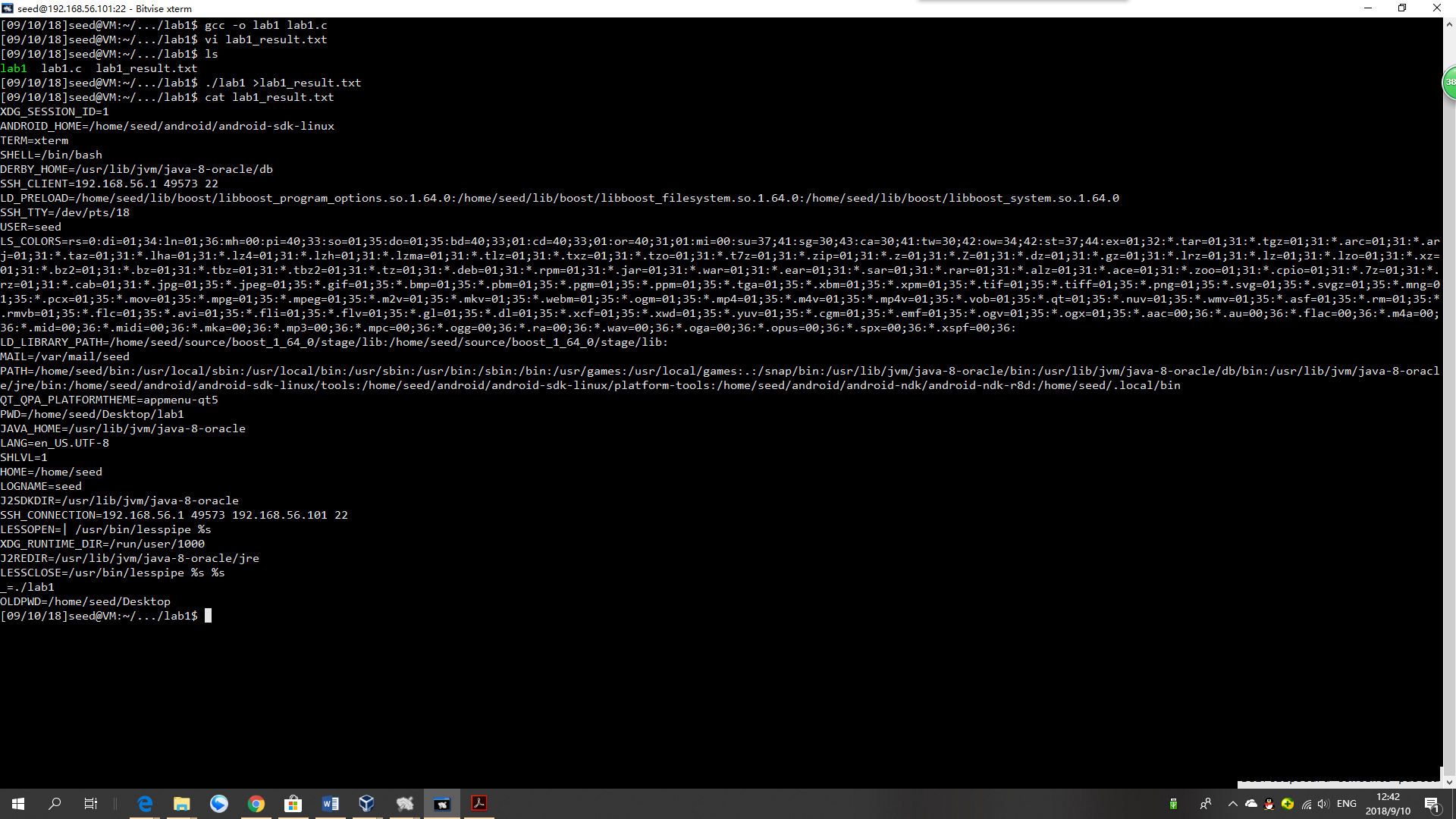
Observation:

In the export Xiaohan 123 command, we set a variable Xiaohan. After that, we find the value from printenv, which means the value has been added to the system variables successfully. After I run the command unset, I find that the variable has been cleaned.

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

Step 1:

Please compile and run the following program, and describe your observation. Because the outputcontains many strings, you should save the output into a file.



Observation:

The program executes the printenv command successfully and have save the output into a file called lab1\_result.

Step 2:

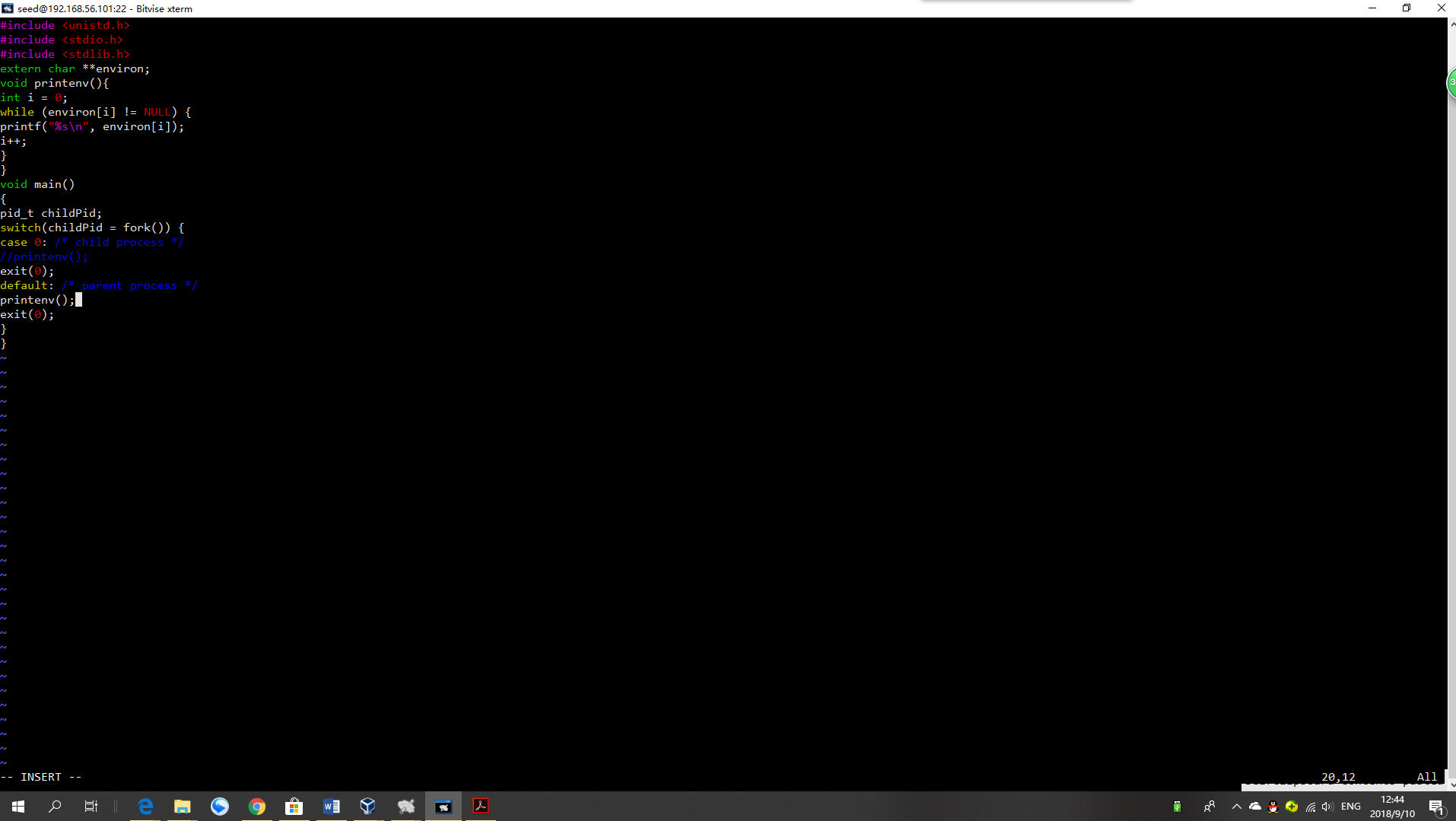
Now comment out the printenv() statement in the child process case (Line 1), and uncomment the printenv() statement in the parent process case (Line 2). Compile and run the code again, and describe your observation. Save the output in another file.

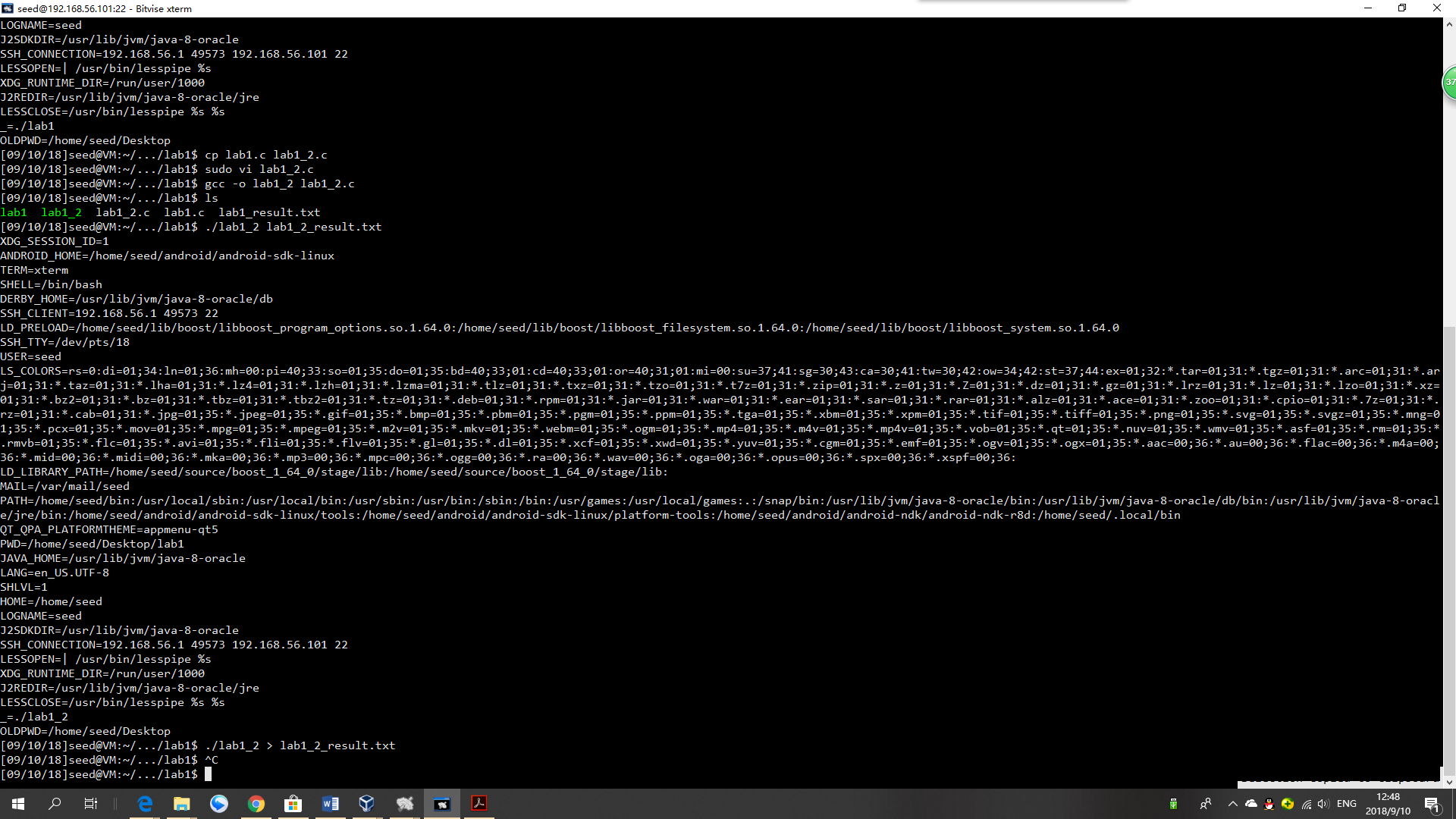
cp lab1.c lab1\_2.c

sudo vi lab1\_2.c

gcc -o lab1\_2 lab1\_2.c

./lab1\_2 > lab1\_2\_result.txt



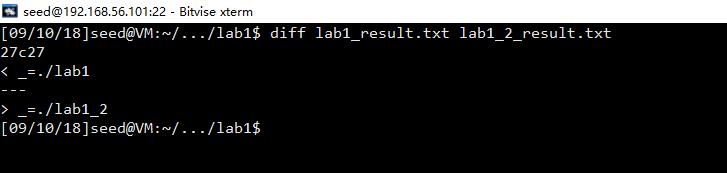


Observation:

The new program has been successfully compiled and executed.

Step 3:

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



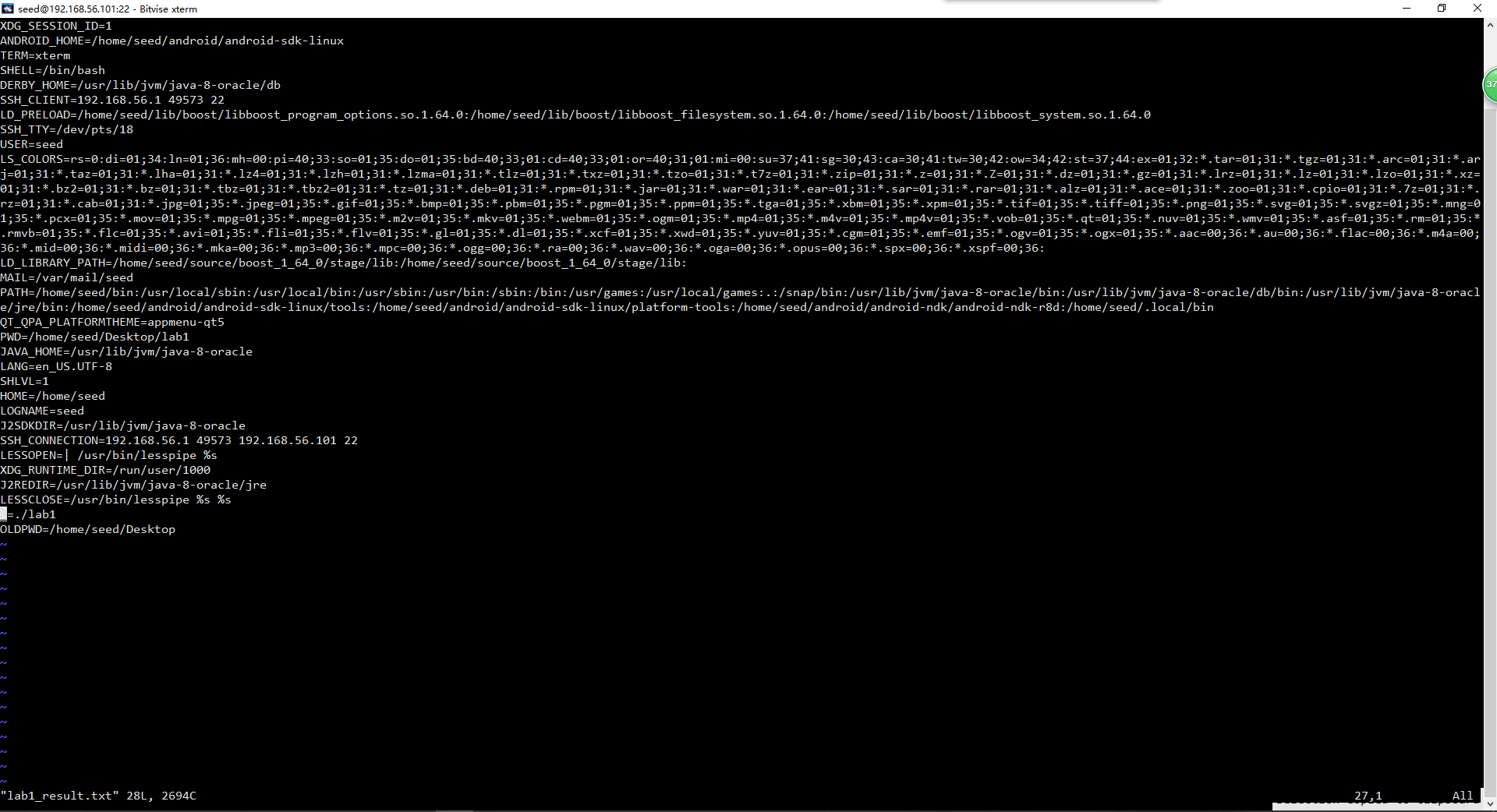
diff lab1\_result.txt lab1\_2\_result.txt

27c27

< \_=./lab1

---

> \_=./lab1\_2

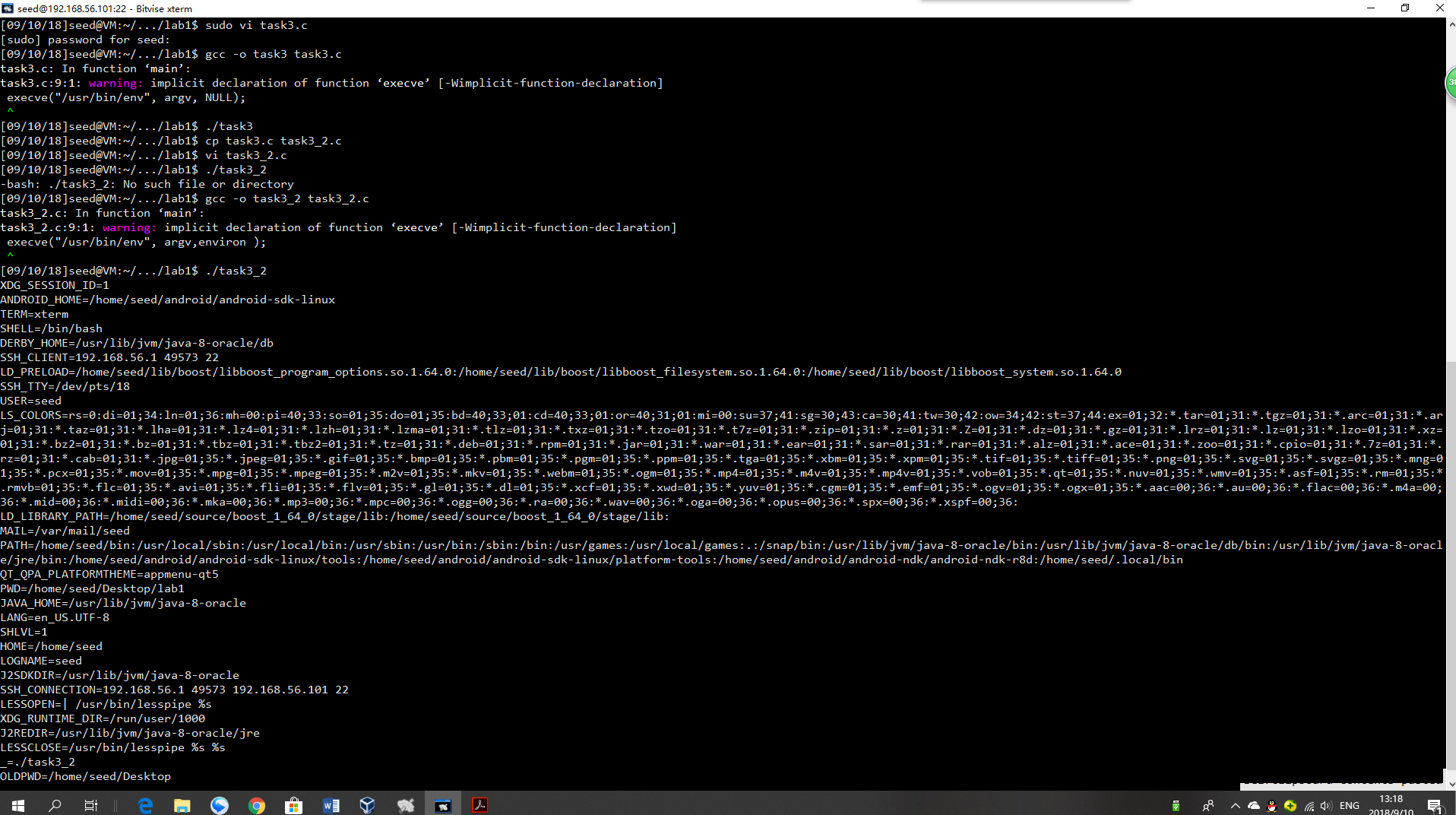


We can see that both results have the execution file name as their variable in the second last line. Both files have same output except the name.

Conclusion:

The variables are not decided by the thread which execute printenv.

### Task 3: Environment Variables and execve()



Observation:

Without changing the program, the original program won’t print anything. But while changing the value NULL to “environ”, the output has been shown the same as printenv’s output.

Explanation:

The execve() function taks three arguments: (1) the command to run, (2) the arguments used by command, (3) the environment variables passed to the new program. In the original program, the environment variables are set to NULL, while the env command need those variables to read, so it returns nothing. But After changing the program, set as “environ”, the system variables are read successfully.

### Task 4: Environment Variables and system()

#include <stdio.h>

#include <stdlib.h>

int main()

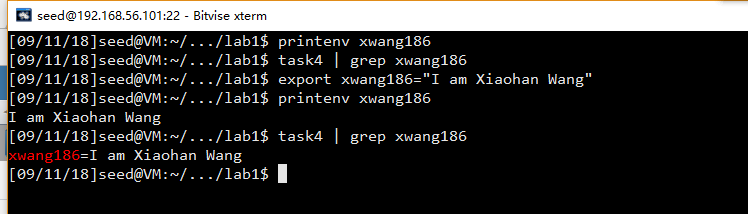
{

system("/usr/bin/env");

return 0 ;

}

Compile and run the program given:



*printenv xwang186*

*task4 | grep xwang186*

*export xwang186="I am Xiaohan Wang"*

*printenv xwang186*

*task4 | grep xwang186*

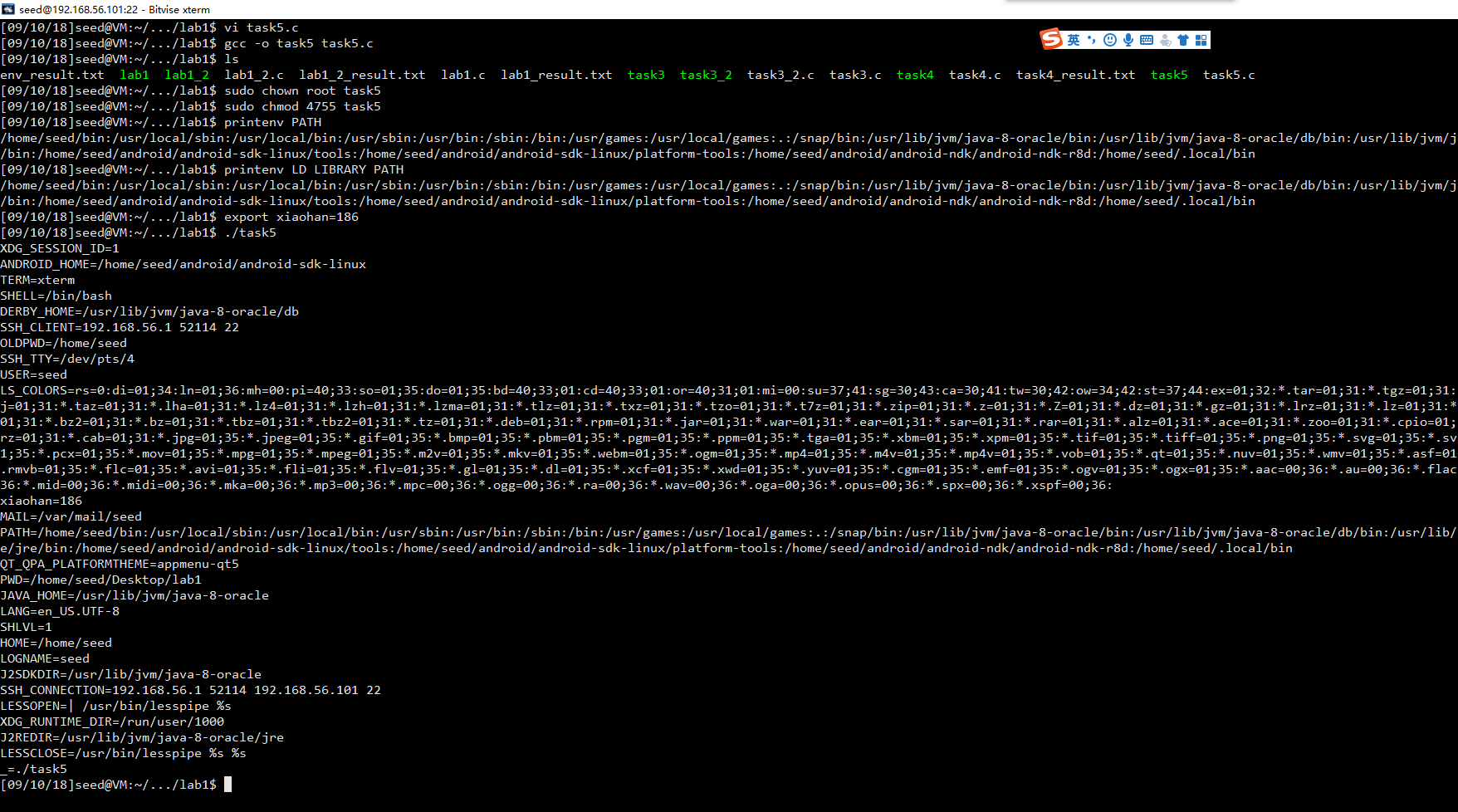
Observation:

The env command has been successfully executed. And we can see that the new variable “xwang186” shows inside and outside the program.

Explanation:

This is because the environment variables of the calling process is passed to a new program /bin/sh.

### Task 5: Environment Variable and Set-UID Programs



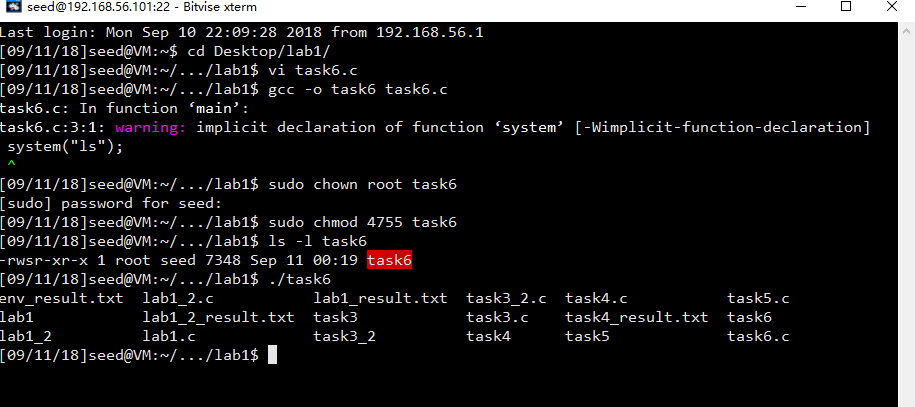
Observation:

As we can see, printenv command shows that there have been variables of PATH and LD LIBRARY PATH. However, after running the program, we can see the PATH value does not change, and I can find the “xiaohan” value as 186 the same as I set before, while the LD LIBRARY PATH variable disappears.

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

Firstly, run the program before doing the attack:

Compile and Run:

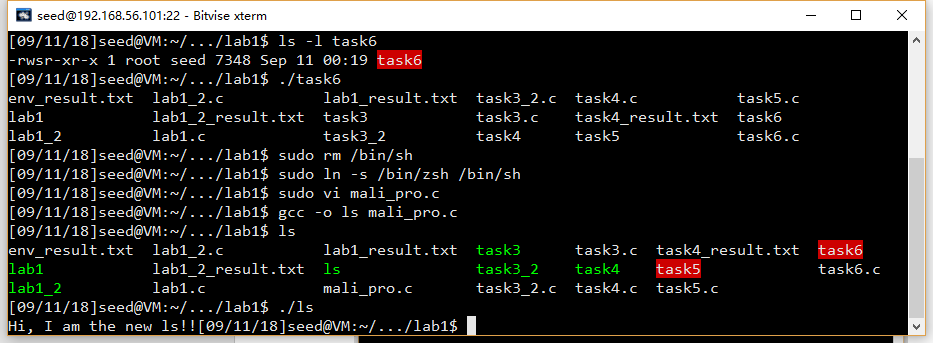


Observation:

As we can see, the program does “ls” command using a shell program. It works as the “ls” function provided by the system libraries.

Write my own ls program:

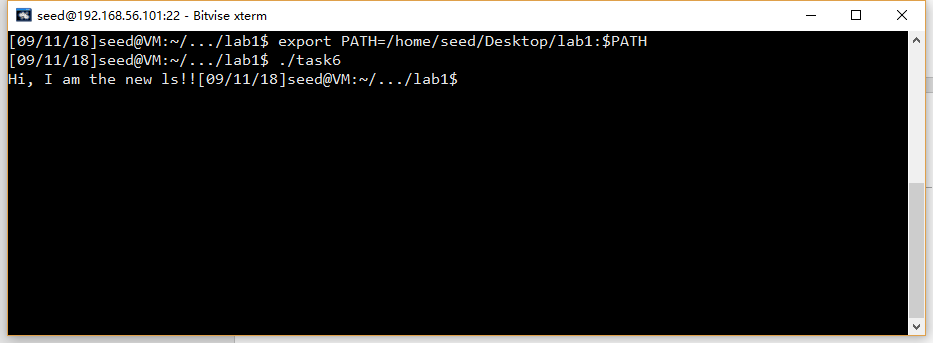




As we can see, my own ls can be run only using “./ls”

Change the path and add the current lib into PATH.

export PATH=/home/seed/Desktop/lab1:$PATH



Observation:

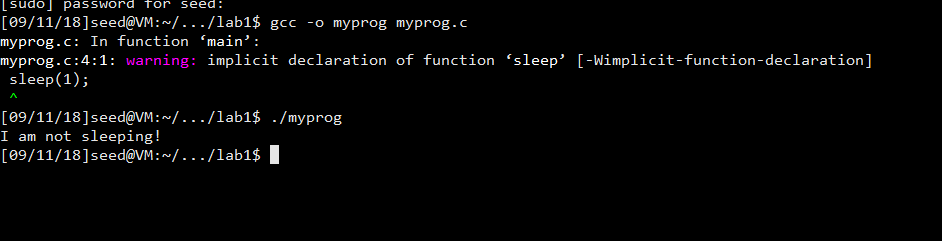
In the task6 program, the system function “bin/ls” should be used. However, After I changed the PATH to my own dictionary, the task6 has run my own codes of “ls”.

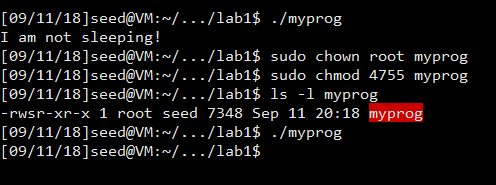
Explanation:

After changing the PATH variable, the ls function is found in the new path by default. Which means, system(“ls”) has been redirected to my own dictionary and run my own ls codes.

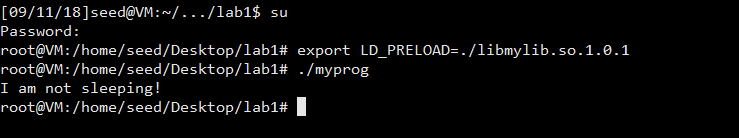
### Task 7: The LD PRELOAD Environment Variable and Set-UID Programs

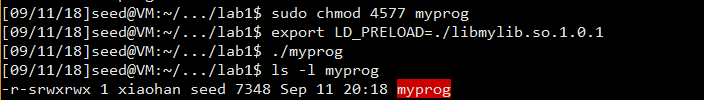
1 Make myprog a regular program, and run it as a normal user

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



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



4 Make myprog a Set-UID user1 program (i.e., the owner is user1, which is another user account), export the LD PRELOAD environment 

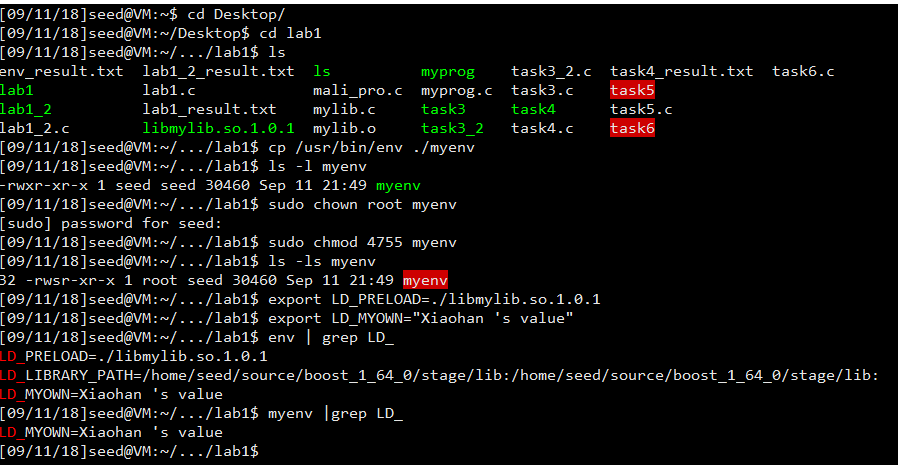
Step 3 Design and Explanation:

In the first test of step 2, because myprog is not a set-UID program, the myprog program is executed by default and using default library. After linking new libraries, the sleep function is found in the new library that we just added, and that is why the fake sleep function make effects.

In the second tests of step 2 and step 3, myprog function has been changed to “set-UID” program, with the ownership of root. As a result， when the process’s real and effective IDs differ, a countermeasure implemented by the dynamic linker make effects, ignoring the LD\_PRELOAD environment variables. In the second one, the real id is “seed”, but the effective id is root; in the third one, the effective id and real id are both “root”. That is why the countermeasure make effects in the second test but not in the third one.

In the last one, for the same reason, since the real id and the effective id differ, the countermeasure makes effect.

To verify all statement above, we use the fake env function to show how the countermeasure works.

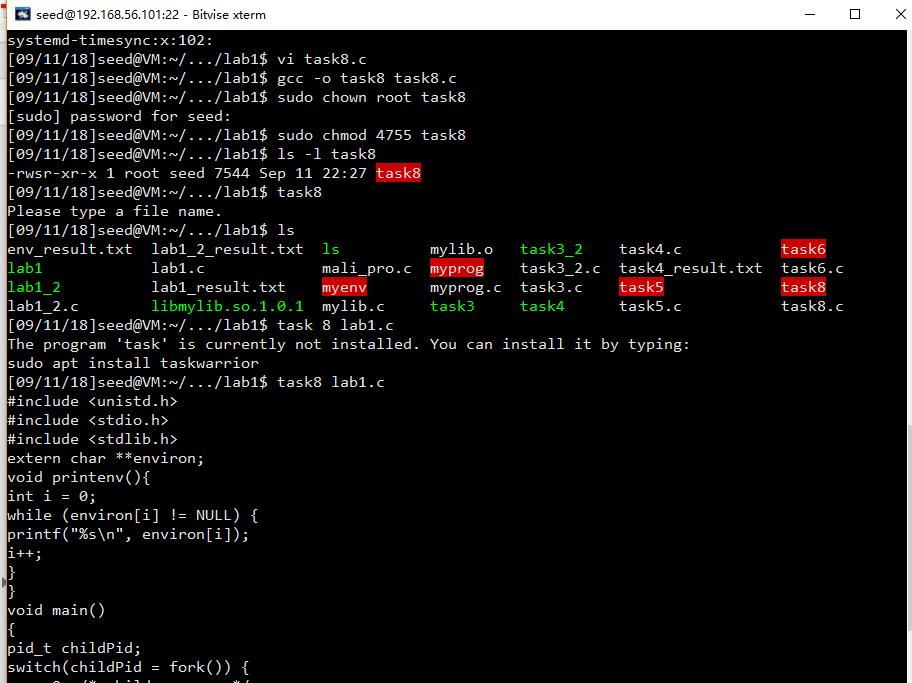


From the experiment above, we can find that the process running myenv does not have the LD\_PRELOAD variable However, the LD\_MYOWN is shown under myenv. This is because this parameter is defined by us and not used by the dynamic linker.

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

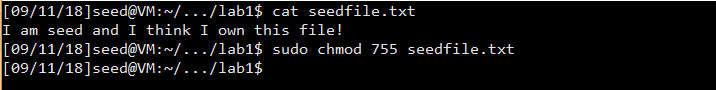
Step1

At first, let’s make out that the program can work correctly.

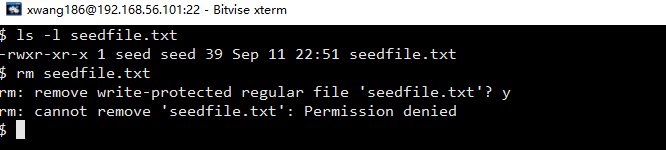


As we can see, since task8 is successfully compiled and set as a set-UID program, the program task8 can easily read files.

Let’s make a file that can only be removed by user seed.



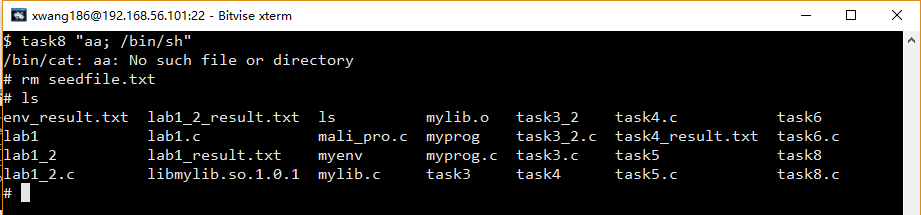
Lets’s log in with another user “xwang186”. I try to remove the seedfile.txt file, but I don’t have enough permission.



We can run the following command using task8:

task8 "aa; /bin/sh"

After that, I have the root privilege so that I can modify the file and remove the file.



Observation:

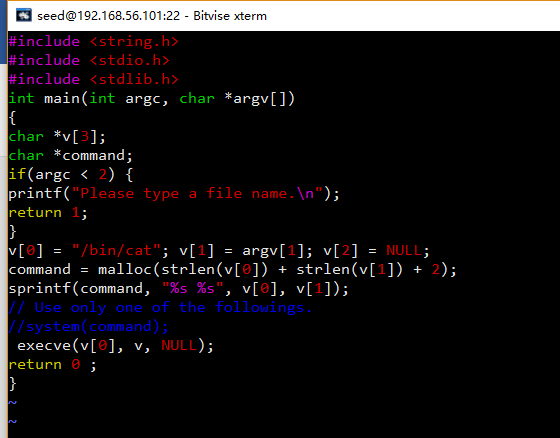
The attack is successful! Xwang186 got the root privilege and removed the file that can be only removed by seed and root.

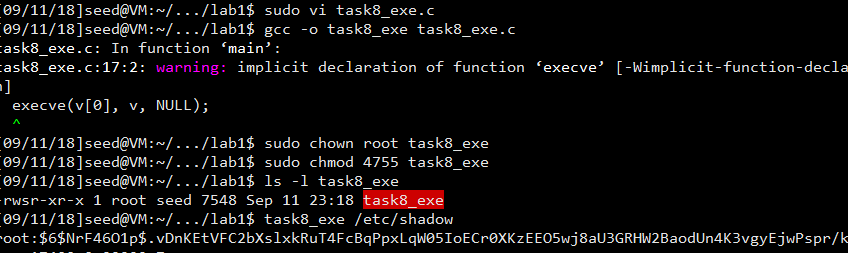
Explanation:

The system statement is only input whatever is the parameter into the bin/sh. Such that, although we can only execute commands start with “cat”, we can use “;” to run more than one command at once.

Step2

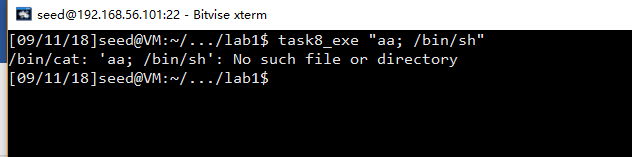
Modify the file:





Try the following command again:

task8\_exe "aa; /bin/sh"



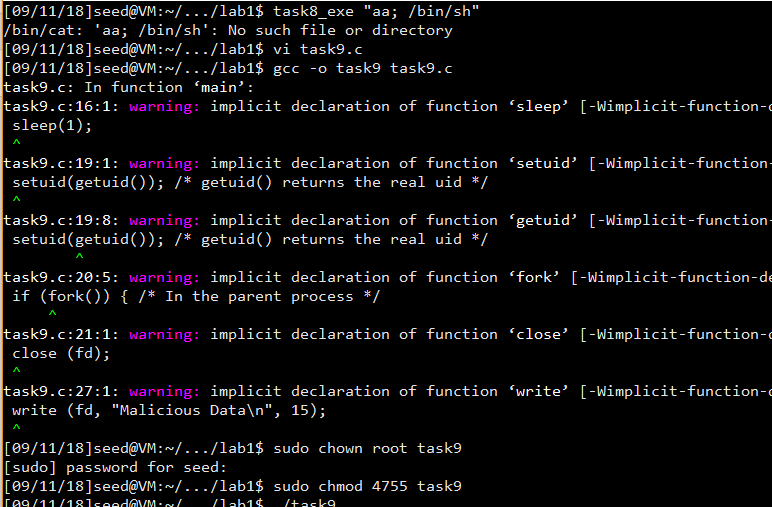
Observation:

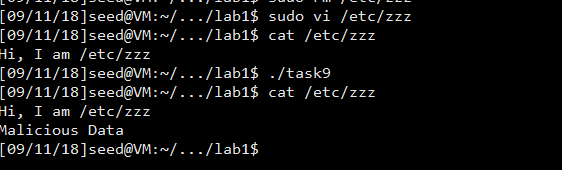
Using the execute version, the attack is denied.

Explanation:

Using the execute function, there is only one command “cat” is executed, which is cat. As a result, “;” can not mislead the program again and "aa; /bin/sh" can only be recognized as one filename.

### Task 9: Capability Leaking





Observation:

It shows clearly that the Malicious Data is successfully added to the file.

Explanation:

The reason why this attack is making effect is that the program applied for “fd”, which should be downgrade right after the program closed. But before running the command “close(fd)”, the fd is still active for the high privilege. As a result, the child process can write to the file using the high privilege.