

**入侵检测与数字取证**

**黄雨卓 57117205**

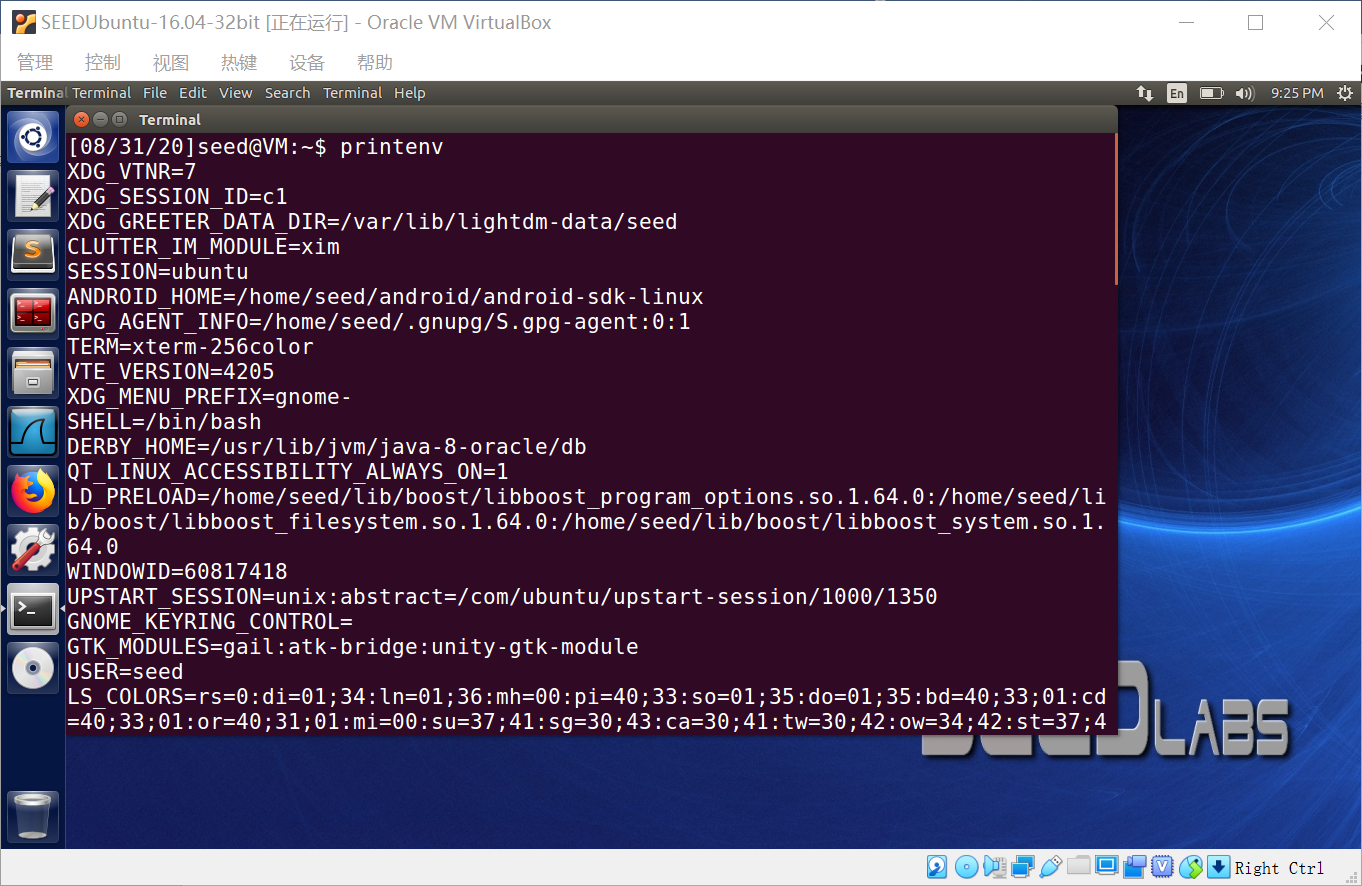
**LAB1**

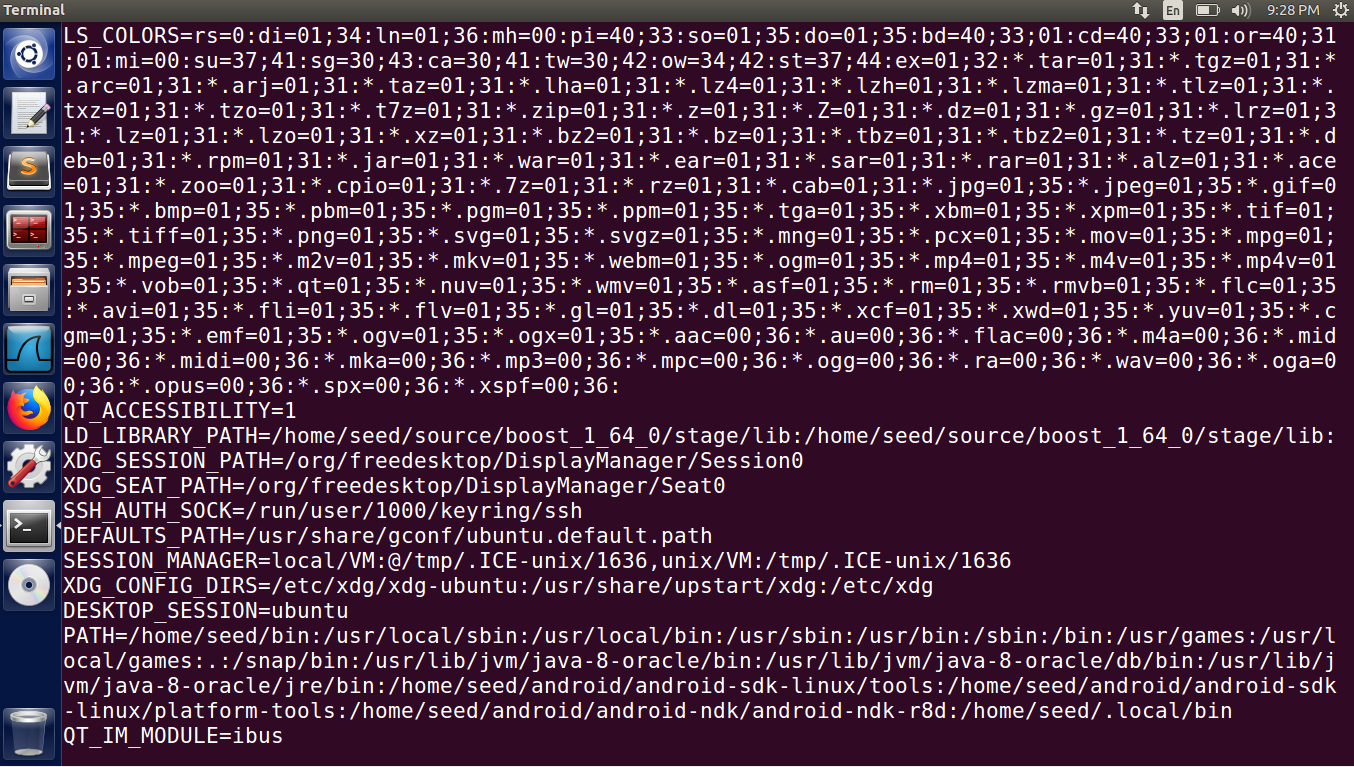
2020年9月1日

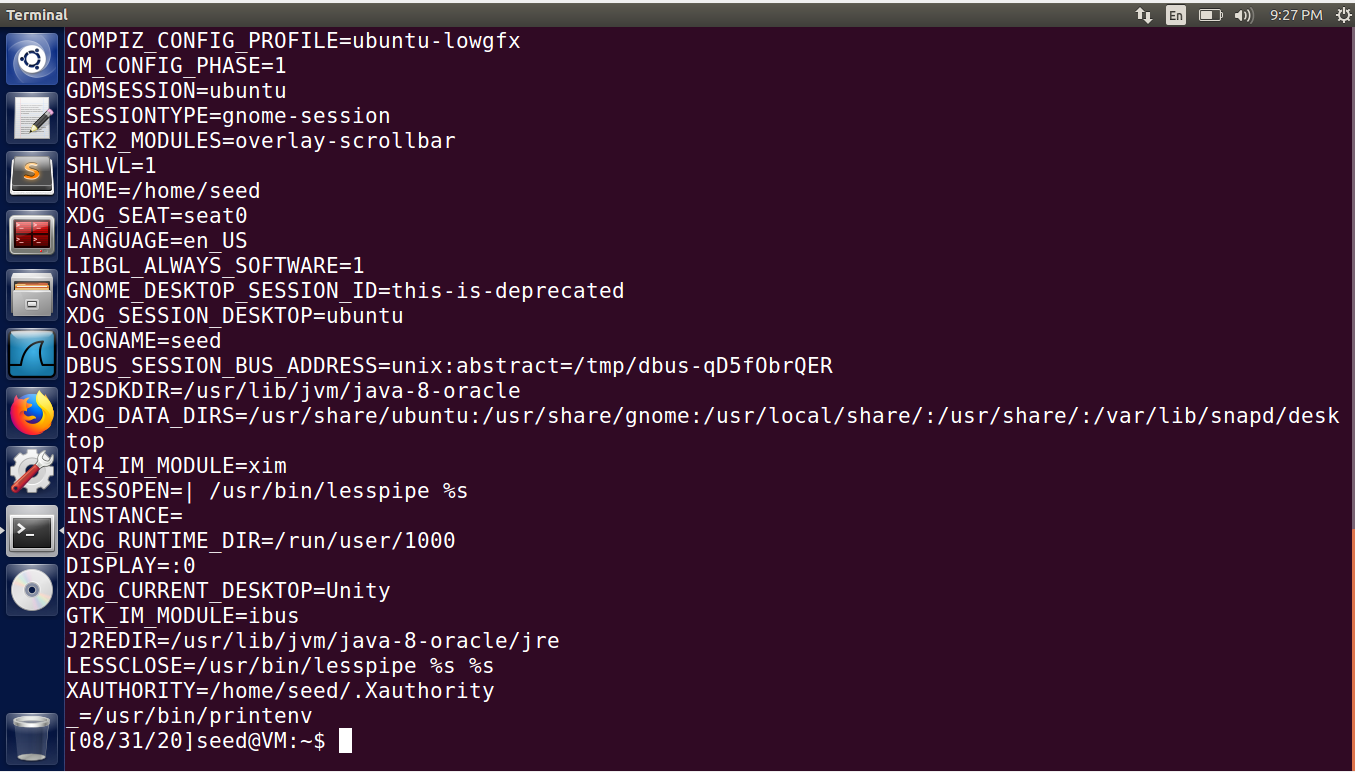
1.1Manipulating Environment Variables

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 uses 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 (please do not do it for this lab). Please do the following tasks:

• 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".



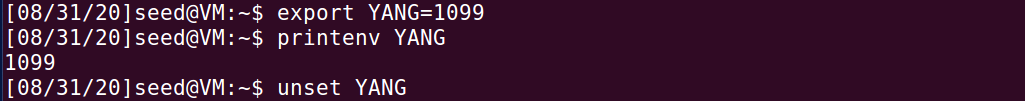




“printenv PWD”:



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

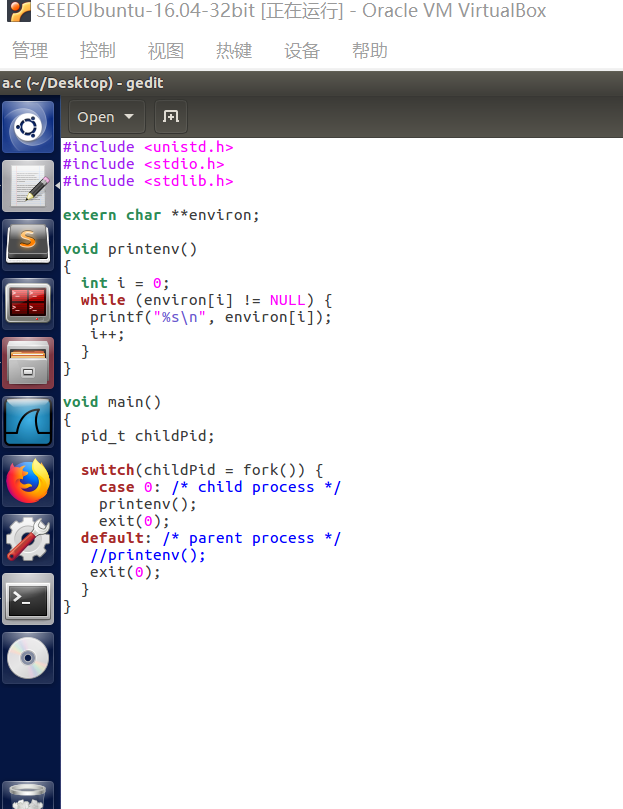




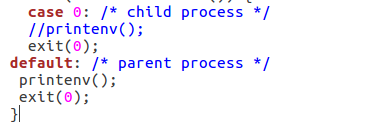
1.2PassingEnvironmentVariablesfromParentProcesstoChildProcess

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.

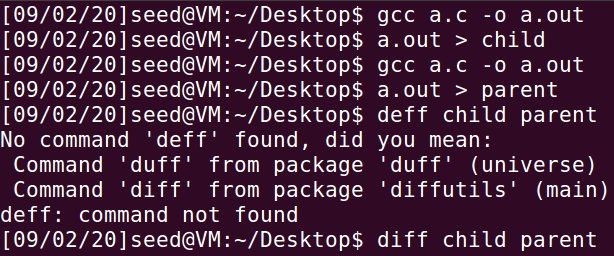
Step1. Please compile and run the following program, and describe your observation. Because the output contains many strings, you should save the output into a ﬁle, such as using a.out > child (assuming that a.out is your executable ﬁle name).



Step2. Nowcommentouttheprintenv()statementinthechildprocesscase(LineÀ),anduncomment the printenv() statement in the parent process case (Line Á). Compile and run the code again, and describe your observation. Save the output in another ﬁle.



Step3. Compare the difference of these two ﬁles using the diff command. Please you’re your conclusion.

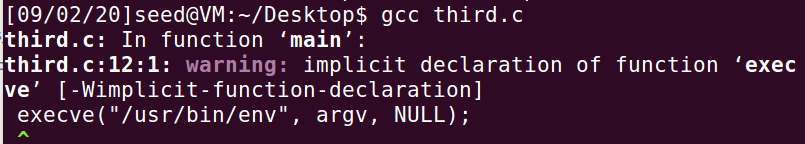
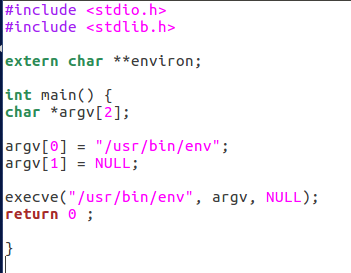


发现两个文件内容完全相同。说明在进程环境变量会继承父进程的环境变量。

1.3 EnvironmentVariablesand execve()

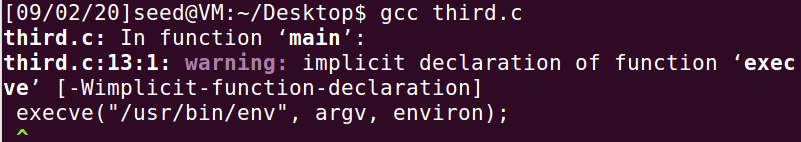
In this task, we study how environment variables are affected when a new program is executed via execve(). The function execve() calls a system call to load a new command and execute it; this function never returns. No new process is created; instead, the calling process’s text, data, bss, and stack 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.



Step2. Change the invocation of execve() in Line À to the following; describe your observation.

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



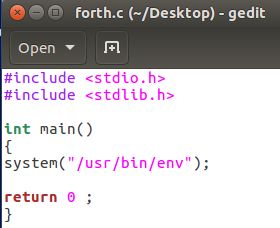
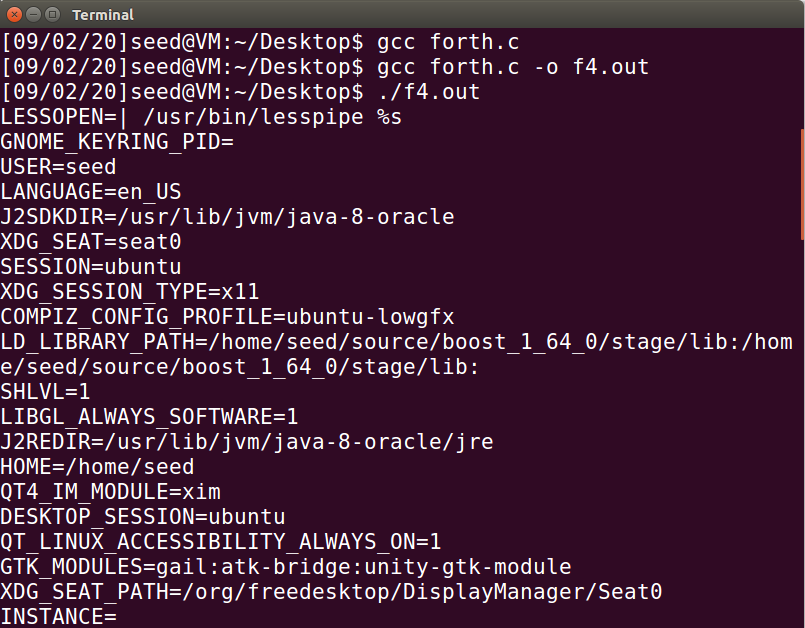
Step3. Please draw your conclusion regarding how the new program gets its environment variables.

与子进程继承父进程不同，execcve() 建新进程的环境变量会重新赋值。

1.4 EnvironmentVariablesand 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.

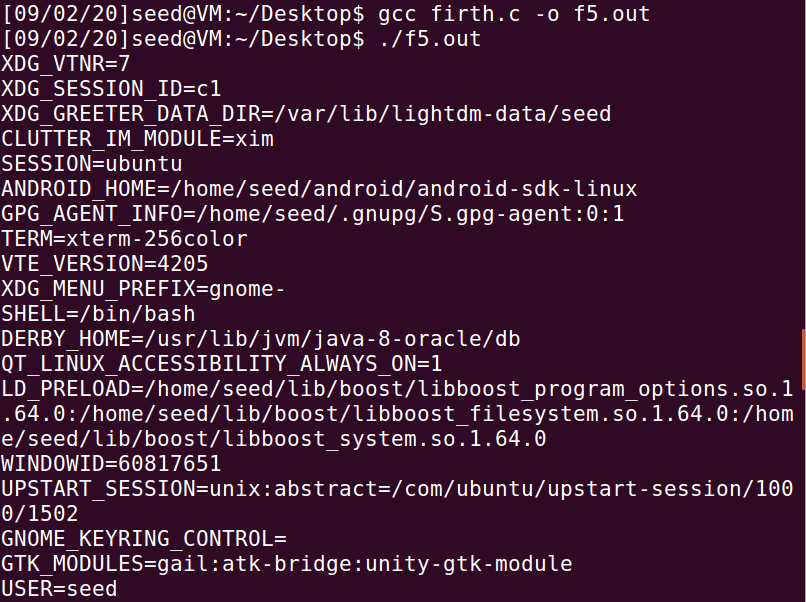
If you look at the implementation of the system() function, you will see that it uses execl() to execute /bin/sh; execl() calls execve(), passing to it the environment variables array. Therefore, usingsystem(),theenvironmentvariablesofthecallingprocessispassedtothenewprogram/bin/sh. Please compile and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

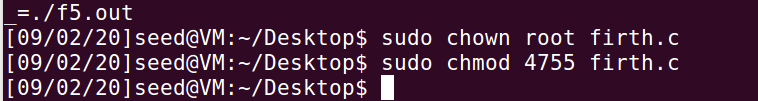
1.5 EnvironmentVariableand Set-UID Programs

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, then 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 it escalates the user’s privilege when executed, making it 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.

Step1. Write the following program that can print out all the environment variables in the current process.



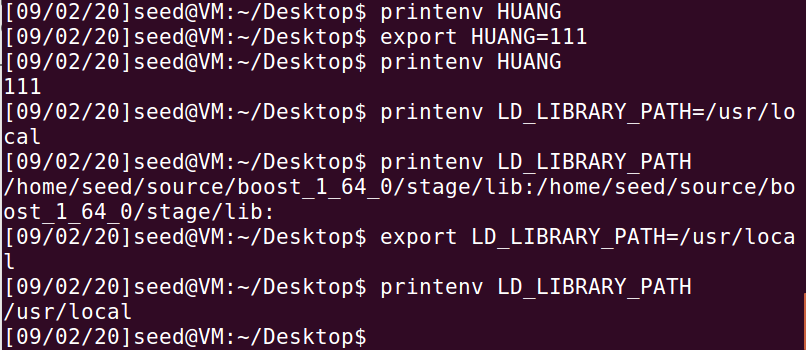
Step2. Compile the above program, change its ownership to root, and make it a Set-UID program.

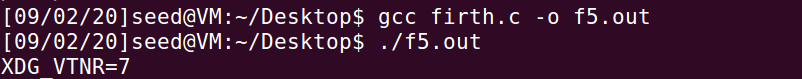


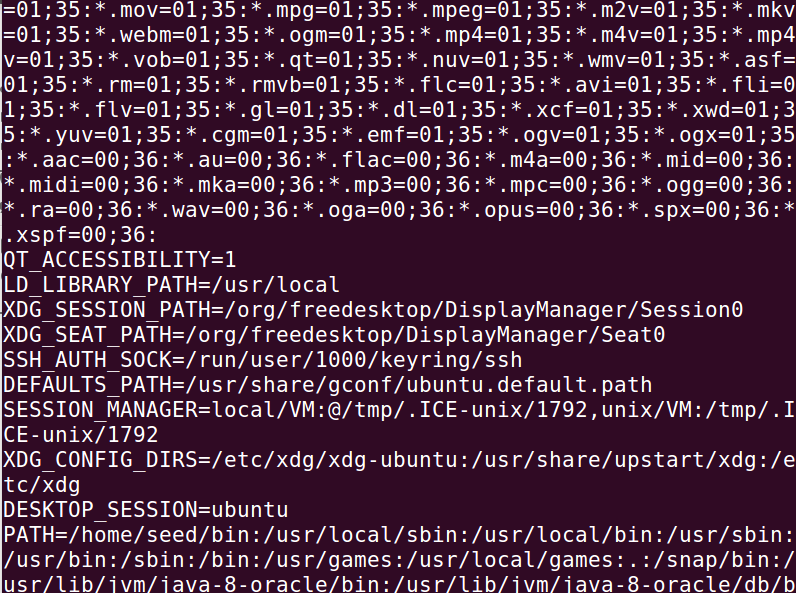
Step 3. In your shell (you need to be in a normal user account, not the root account), use the export command to set the following environment variables (they may have already exist):

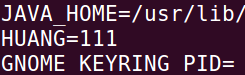
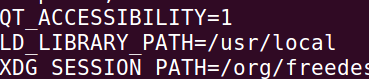
• LD\_ LIBRARY\_ PATH=” /home/seed/bin:/usr/local”

• HUANG=”111” (this is an environment variable deﬁned by you, so pick whatever name you want).

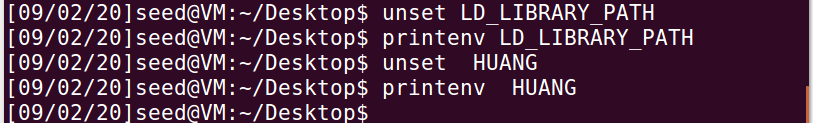








消除影响：



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.

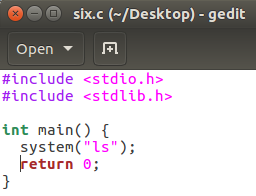
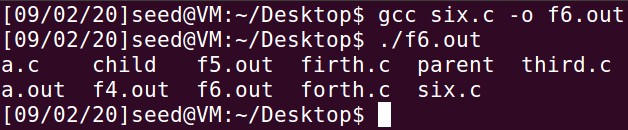
1.6 ThePATHEnvironmentVariableand Set-UID Programs

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):

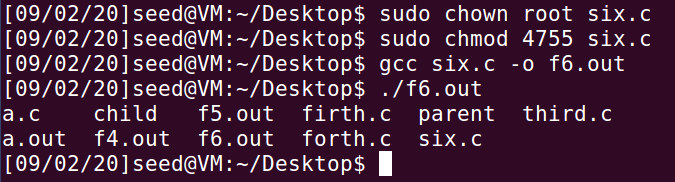
$ export PATH=/home/seed:$PATH

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:

Please compile the above program, and change its owner to root, and make it a Set-UID program. Can you let this Set-UID program run your code instead of /bin/ls? If you can, is your code running with the root privilege? Describe and explain your observations.

改成root权限：



Note(Ubuntu16.04VMonly): The system(cmd) function executes the /bin/sh program ﬁrst, and then asks this shell program to run the cmd command. In both Ubuntu 12.04 and Ubuntu 16.04 VMs, /bin/sh is actually a symbolic link pointing to the /bin/dash shell. However, the dash program in these two VMs have an important difference. The dash shell in Ubuntu 16.04 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. The dash program in Ubuntu 12.04 does not have this behavior. 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 shellthatdoesnothavesuchacountermeasure. Wehaveinstalledashellprogramcalled zsh inourUbuntu 16.04VM.Weusethefollowingcommandstolink/bin/shtozsh(thereisnoneedtodotheseinUbuntu 12.04):

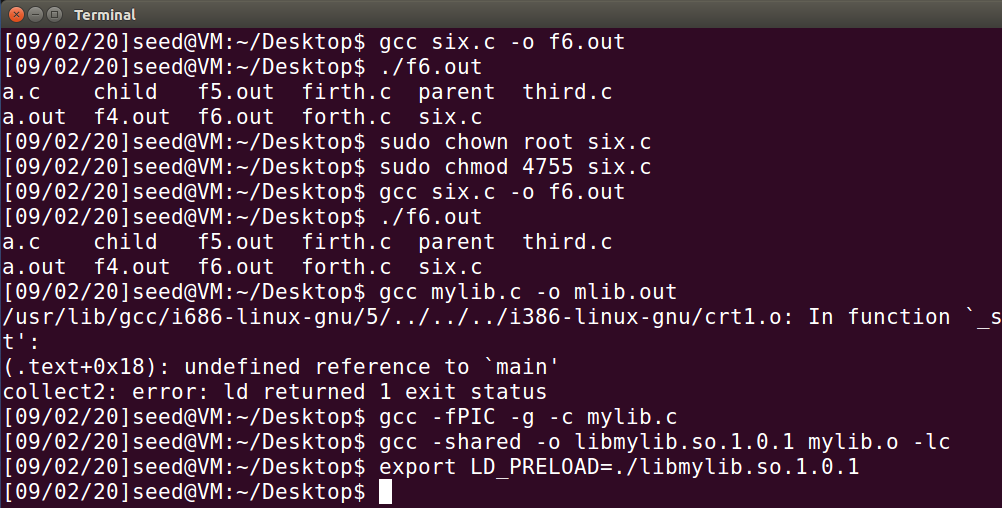
1.7 The LD PRELOAD EnvironmentVariableand Set-UID Programs

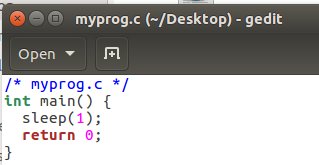
In this task, we study how Set-UID programs deal with some of the environment variables. Several environment variables, including LD PRELOAD, LD LIBRARY PATH, and other LD \* inﬂuence the behavior of dynamic loader/linker. A dynamic loader/linker is the part of an operating system (OS) that loads (from 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.

Step1. 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:





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

3. Now, set the LD PRELOAD environment variable:

% export LD\_PRELOAD=./libmylib.so.1.0.1

4. Finally, compile the following program myprog, and in the same directory as the above dynamic link library libmylib.so.1.0.1:

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

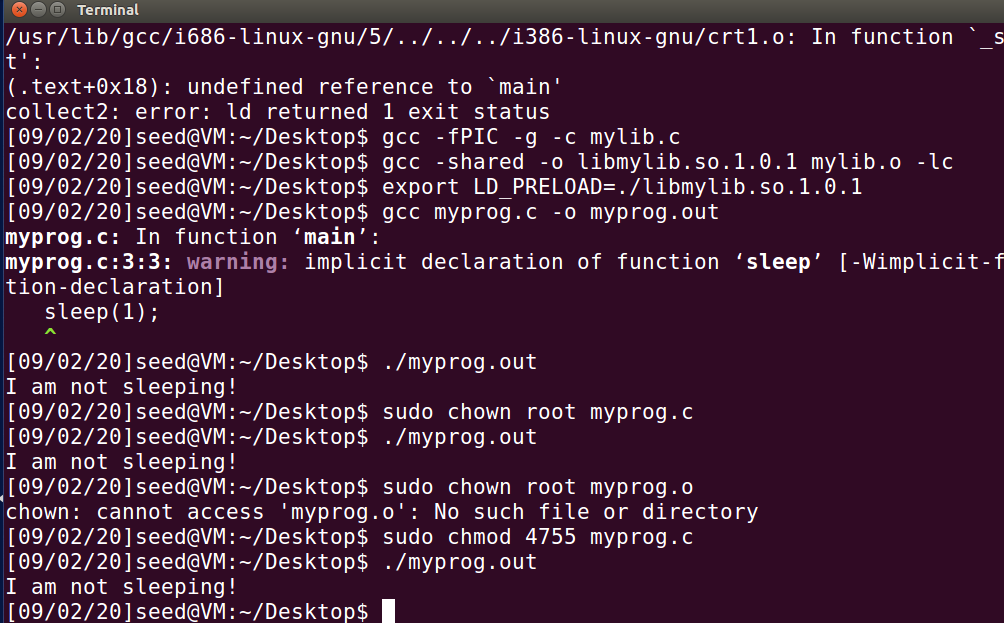
• Make myprog a regular program, and run it as a normal user.

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

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

• 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.

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).



1.8 InvokingExternalProgramsUsing system() versus execve()

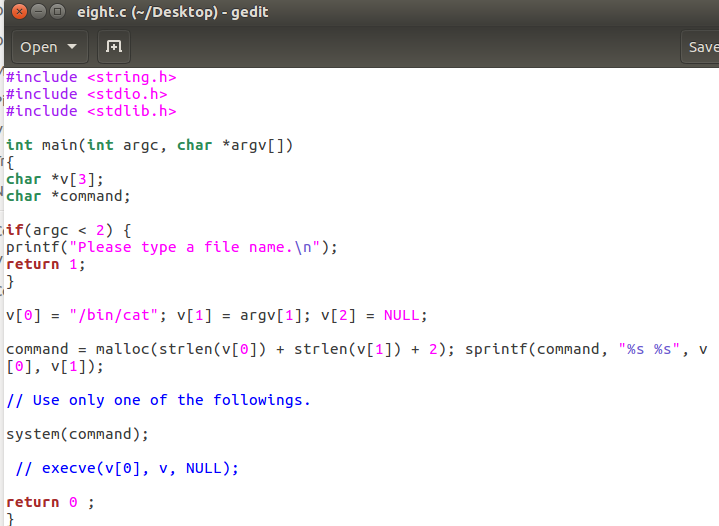
Although system() and execve() can both be used to run new programs, system() is quite danger-ous if used in a privileged program, such as Set-UID programs. We have seen how the PATH environment variable affect the behavior of system(), because the variable affects how the shell works. execve() does 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.

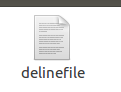
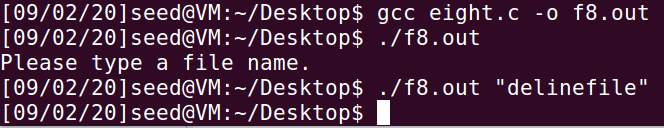
Bob works for an auditing agency, and he needs to investigate a company for a suspected fraud. For the investigation purpose, Bob needs to be able to read all the ﬁles in the company’s Unix system; on the other hand, to protect the integrity of the system, Bob should not be able to modify any ﬁle. To achieve this goal, Vince, the super user 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.

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?

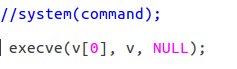
Step 2: Comment out the system(command) statement, and uncomment the execve() statement; the program will use execve() to invoke the command. Compile the program, and make it a root-owned Set-UID. Do your attacks in Step 1 still work? Please describe and explain your observations.

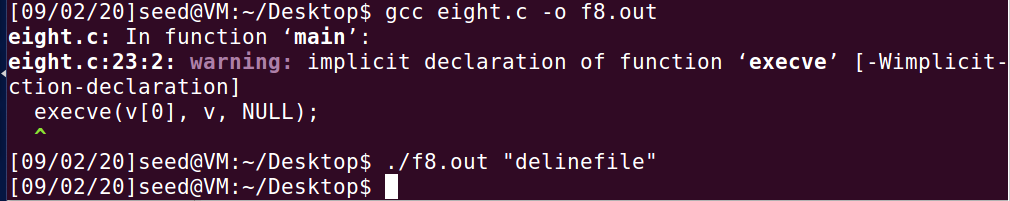
System时成功删除。



Execve时，出现问题，但是删除与上一条一样。





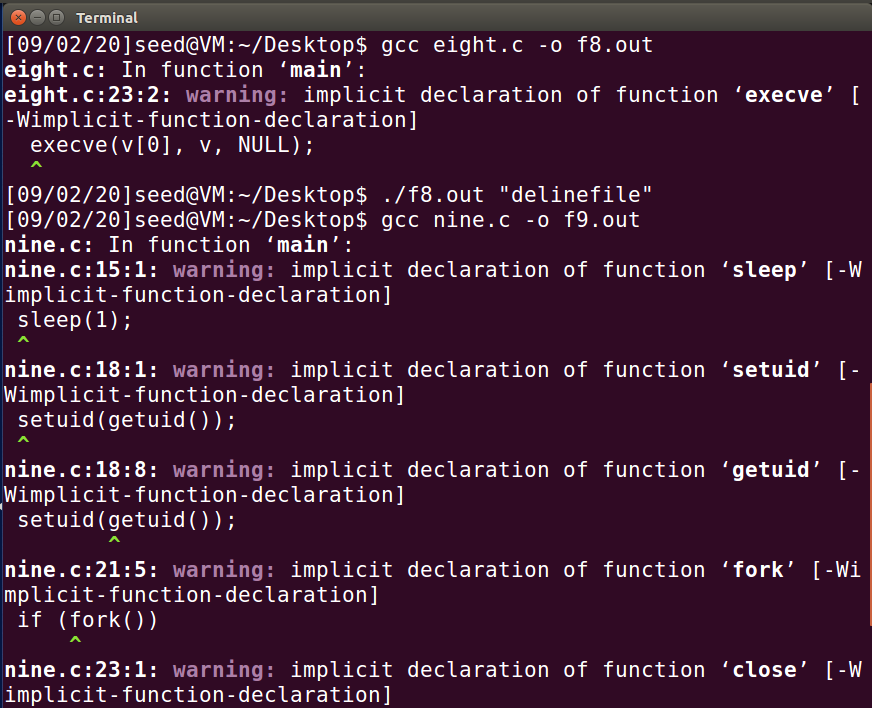
1.9 CapabilityLeakin)

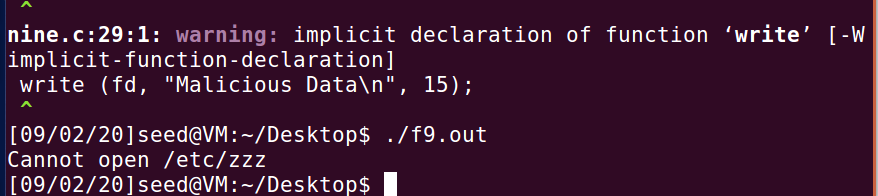
To follow the Principle of Least Privilege, Set-UID programs often permanently relinquish their root privileges if such privileges are not needed anymore. Moreover, sometimes, the program needs to handover its control to the user; in this case, root privileges must be revoked. The setuid() system call 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, and describe what you have observed. Will the ﬁle /etc/zzz be modiﬁed? Please explain your observation.

在普通权限时，不能打开：





Root权限下，仍然不可以打开：



