George, Levi

CS44500 – Computer Security

01/21/23

Lab 2

Task 1. Print Environment Variables [Looks good to me]

Text, letter, email

Description automatically generated

Text

Description automatically generated

It is possible to set using export VARNAME=VARVALUE, I set MY\_NAME and showed it in this printenv

Task 2. Passing Env Variables from Parent to Child

2.2 Pass Env Var from Parent to Child

Text

Description automatically generated

According to this experiment, child threads will get their env variables from their parent. However, they don’t share other specific things, they do share env variables. Some examples of things they do not share from the man page would be process ID, memory locks, semaphore adjustments nor timers.

Task 3. Env Var and Execve()

Text

Description automatically generated

It appears that we need to explicitly pass our environment variables to the new overriding process that execve generates, if we include NULL then it will get no env variables, but if we pass our current shell’s env variables it will inherit those and use them.

Task 4. Env Vars and System()

Running this code

Table

Description automatically generated

I get this output, which shows that our new shell will get our env variables, we can see this based on the personalized env variables that I made such as MY\_NAME or ANY\_NAME.

Graphical user interface, text, application, email

Description automatically generated

Task 5. Env Var and Set-UID

Running this code

Graphical user interface, text, application

Description automatically generated

And running these commands will set the env variables, create the set-uid program and let us run the printenv code.

Graphical user interface, text

Description automatically generated

(I switched to putty on this part)

Text

Description automatically generated

We see that the vars we set before, such as PATH and ANY\_NAME are present. Which shows that even if we think a malicious user can’t do much with their own env variables, by escalating privileges and editing certain programs in their own directories (like making LS do something different) they can be very powerful.

TASK 6: PATH env var and SET\_UID

Using this malicious code

Graphical user interface

Description automatically generated with low confidence

I was able to get this output (note that I had an earlier version without the /etc/shadow read)

Text

Description automatically generated with medium confidence

I was able to get my malicious code to run, it just prints out the message shown in the snippet.

I added the etc shadow read after this first test.

I edited my code and tried to get it to print the shadow file using another system() call, but was unable to do so.

Text

Description automatically generated

However, I did return to this task after my first attempt, I reset my shell to use ZSH instead of DASH (as per the task’s notes). After this I was able to print out the contents of our shadow file.

[Changing the shell]



[Compiling and moving the malicious LS program]

Graphical user interface, application

Description automatically generated

[Compiling system(ls) code, running code and activating malicious code]

Graphical user interface, text, application, email

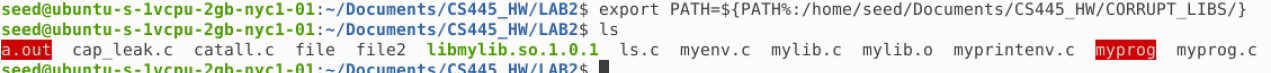
Description automatically generated

[Last portion of shadow file]

Text

Description automatically generated

One thing I encountered, was that my ls program would continue to run, making finding out what my folders held within slightly difficult. I removed the malicious compiled code from my corrupt library, then ran the following command to remove the bad path from the env variable.



TASK 7: LD\_PRELOAD ENV VAR and SET\_UID (RETURN TO THIS ONE)

Text

Description automatically generated

This is the code that we will inject via a library

I ran these commands below to build and set the false library as a preloaded library



This code will be run to use the malicious sleep library

Text

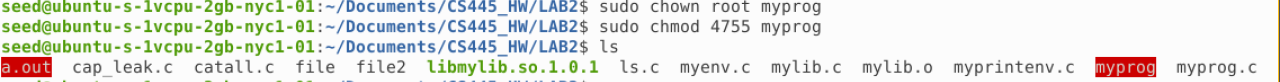
Description automatically generated with low confidence

When I run the code as a regular user

A picture containing text

Description automatically generated

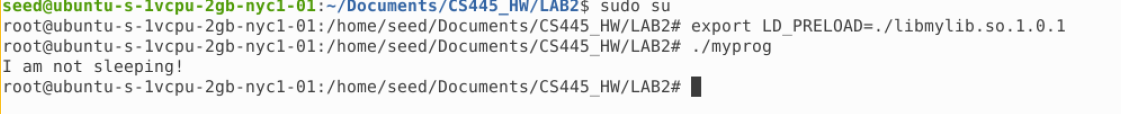
Set the code as a Set-UID



This is what happens when I run the code as a normal user and it is Set-UID



I switched to root and used LD\_PRELOAD again, then ran the code in root. Here is what happened.



When I went through the process of changing users and giving them control of the program I got this output

Text

Description automatically generated

I would need to export the preload library before the effect occurs. I think when we switch users they refresh the environment variables or start a separate shell. This would set the environment variables for them to not include my corrupted library. I see this when I switch user to “steve” export the corrupt library as an env var, then proceed to run the prog, it will show the corrupt code’s message. However, when I exit that users acct or shell, then re-enter and run the code, it doesn’t show the corrupt code’s message. So, we must be starting a new shell based on that user’s default shell. This would wipe out the corrupt LD\_PRELOAD everytime we switch users.

Graphical user interface, text, application, email

Description automatically generated

However, this does not explain why changing the permissions and making it Set-UID would not let it run the corrupt library.

I added a new line of code to my program, one that prints the env variables.

Text, letter

Description automatically generated

When I ran the code as a regular user, it printed the corrupt message.

When I ran the code as a regular user and set the program as a set-uid, it printed no message and no LD\_PRELOAD environment variable.

When we run printenv in root, we get generic env variables, when we run printenv as seed, we get the env variables that I set in previous tasks (such as ANY\_NAME). However, when we run myprog as set-uid and run it from seed, we still get the previous task variables. When we run certain programs as a different user, we may not inherit LD\_\* variables since we are attempting to run the program as them, but I set both root and seed’s LD+PRELOAD as the same library. So, maybe when we run the program under set-uid and our EUID and RUID are different, the system recognizes this and stops us from using those load environment variables as a security measure. Everytime I ran the code as a different user and the code was Set-UID the malicious statements never worked. I ran the code as root, seed, and even another user. Each time the malicious code ran. However, as soon as I set the program as a Set-UID and changed my user the code failed to run.

TASK 8: Invoking external programs using system() versus execve() [Looks good]

Graphical user interface, text, application, email

Description automatically generated

I was able to invoke a new shell and use that to remove the write protected “steve.txt”

Graphical user interface, text, application

Description automatically generated

When using the execve() version of the code, I was unable to use the same vulnerability to access a new shell. It seems like the reason why this works is because system runs commands directly, for example the %s character will let us inject our string into the code, so we can perform that injection, while with execve, it will run the command separately and disallows us from performing that same injection.

TASK 9: Capability Leaking [Looks good to me]

Graphical user interface, text

Description automatically generated

I was able to get into the file and write to it, by directly referencing that file descriptor. Which is just an integer that the process uses to reference that file. So, when we exit the cap\_leak.c program which accesses the file, it opens that /etc/zzz, so the process assigns a file descriptor to it. Since we don’t clear the file descriptors we can still access them via that descriptor hence using “echo Ihavehackedthesystem >& 3” will add info into the file at descriptor 3. To avoid this issue, we would have to clear the file descriptor via the close() command.