CSCI 476: Lab 01

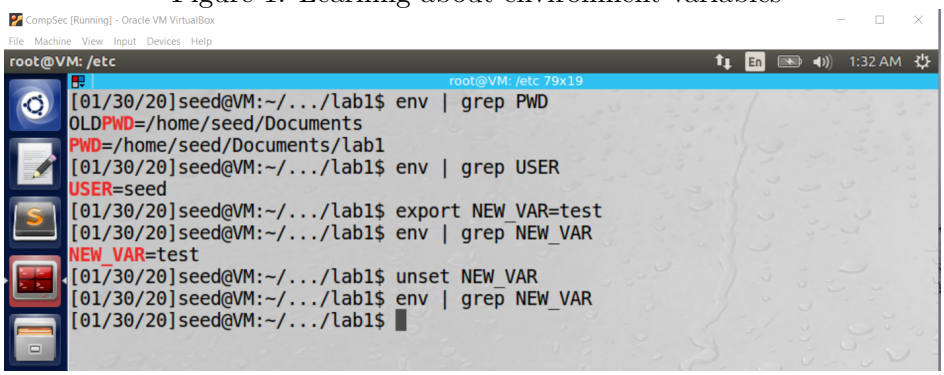
Nathan Stouffer

January 30, 2020

Task 1

In task 1, I familiarized myself with environment variables on linux command line. I did not provide a screenshot for all of my commands, but Figure 1 shows some experimentation.

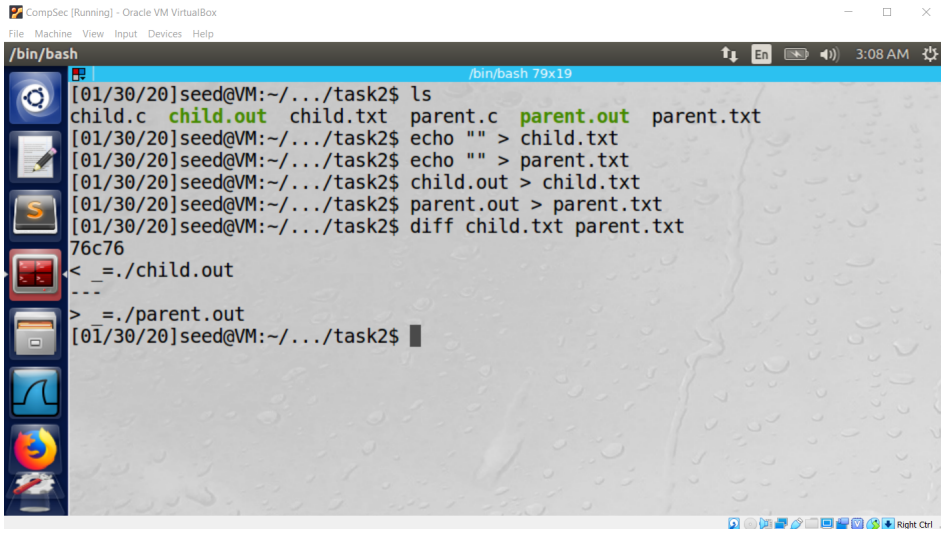
Figure 1: Learning about environment variables



Task 2

Task 2 focused on how a child process receives environment variables from the parent process. Below in Figure 2, there are two executable files with almost identical source code. The file parent.out prints out the environment variables of the parent process while child.out prints out the environment variables of a child process. The child process was created using the fork() method. In Figure 2, I first write an empty string to the two files child.txt and parent.txt, then I run each of the executable files (redirecting output to the appropriate file). Then, I used the diff command to see the differences in the file. The only difference is the name of the executable, so the environment variables must be the same. This means that, when using fork(), the child process inherits the environment variables of the parent process.

Figure 2: Using fork() method



Task 3

In task 3, I viewed how environment variables are affected when starting child processes via the execve() method. The executable null.out sends NULL as the final argument in execve() while env-output.out sends in the global variable environ. Each executable prints the environment variables of the child executable. As seen in Figure 3, null.out prints out nothing, so there are no environment variables. On the other hand, Figure 4 shows env-output.out to print the environment variables of the parent process. So, execve() must be told explicitly what it's environment variables should be.

Figure 3: Sending NULL as argument in execve()

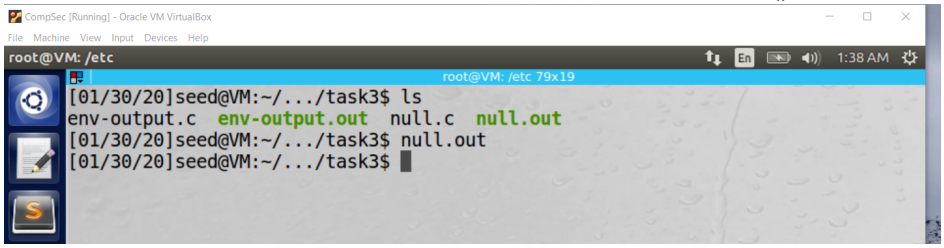
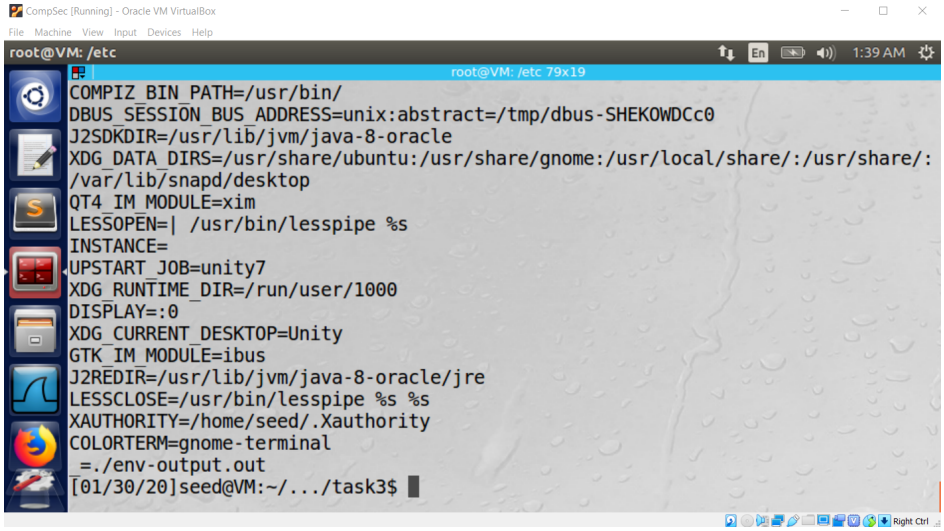


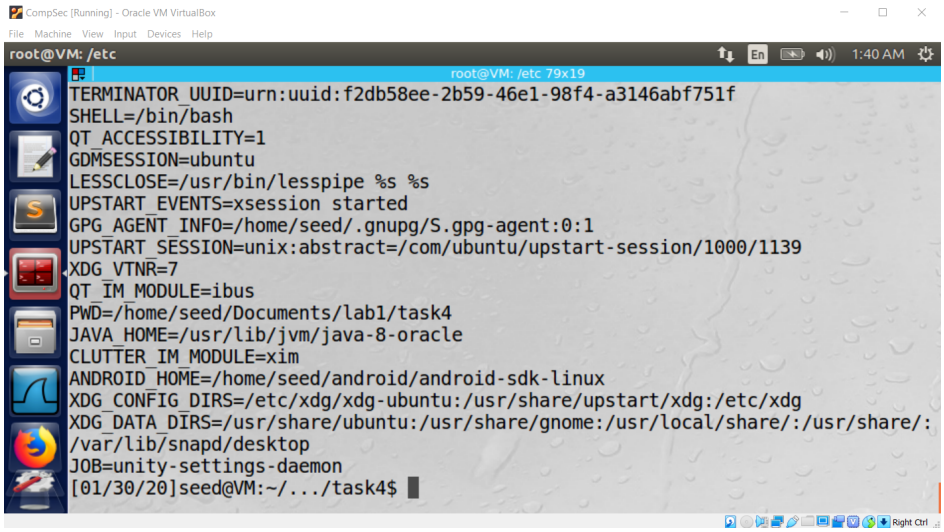
Figure 4: Sending environ as argument in execve()



Task 4

Task 4 is a simple test of using system() to execute other programs. The output of the call system("/usr/bin/env") is show in Figure 5. This verifies that system() starts a child process since a new process (a shell) is created and the environment variables of the parent process are printed. So the calling process must be a child.

Figure 5: Output of system("/usr/bin/env");



Task 5

Task 5 investigates how Set-UID programs are affected by environment variables. In Figure 6, I messed with some environment variables in the shell. I then ran the Set-UID program uid.out. The executable uid.out runs with root privilege, but is vulnerable. This is because Figures 7 and 8 show that the environment variables in the parent process get passed into the child process. For some reason, I could not find LD_LIBRARY_PATH in the output, but PATH and ANY_NAME most definitely appeared. The LD* issue is discussed further in Task 7.

Figure 6: Messing with environment variables

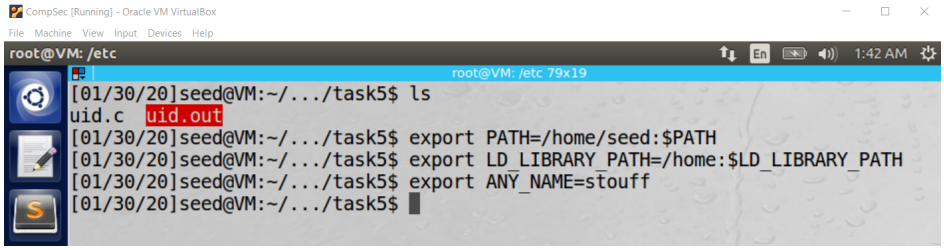


Figure 7: Showing PATH in child process

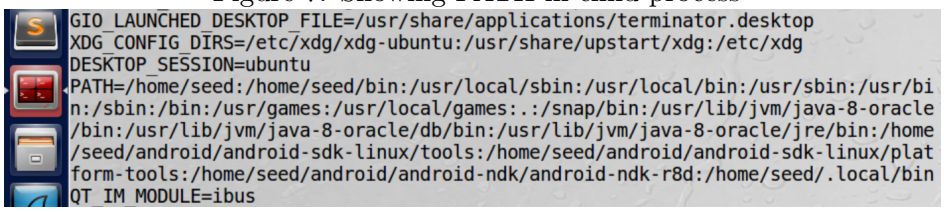
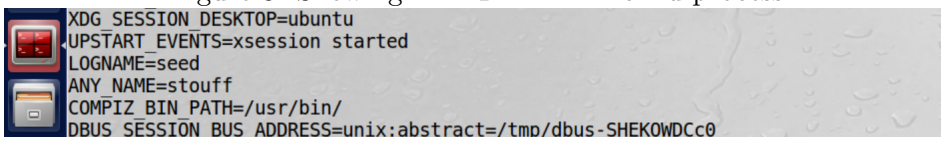


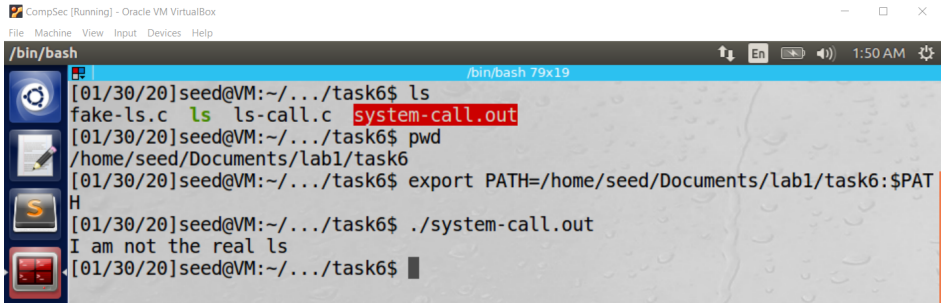
Figure 8: Showing ANY_NAME in child process



Task 6

Task 6 is an example of how changing the environment variables can affect the behavior of a program (even a Set-UID program). The executable system-call.out uses the command system("ls") to run the ls command. However, the absolute path of the command is not specified. So, I created a new executable called ls with different functionality. Then, I changed the PATH environment variable so that programs searched the location of my ls executable before looking elsewhere. Figure 9 shows that system-call.out used the functionality of my false ls command since it did not specify the absolute file path for ls.

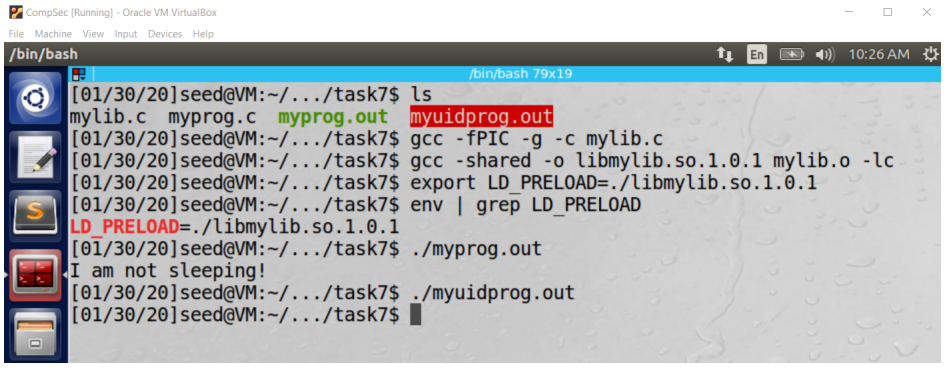
Figure 9: test caption



Task 7

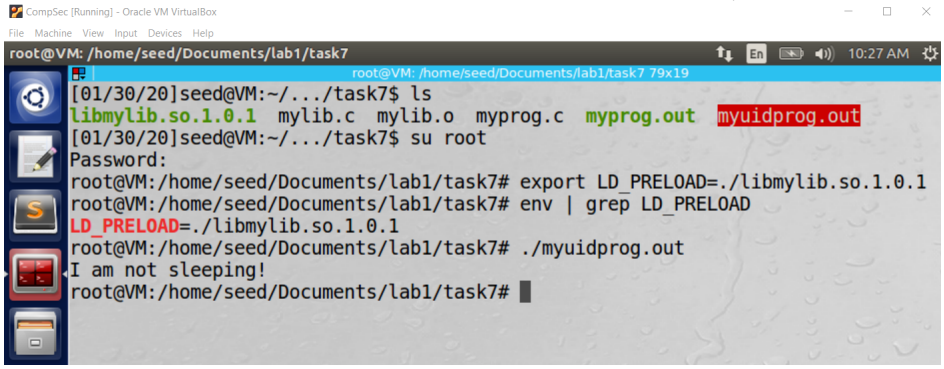
Task 7 looks at another way changing environment variables can affect the way programs run. In Figure 10, I dynamically link a library that overwrites the sleep(int s) function. I then add that library to the LD_PRELOAD environment variable so that any executables will look at the ./libmylib.so.1.0.1 before searching other libraries. This means if sleep(int s) is called, my code will run. Figure 10 shows this to be true for a regular program. However, it also shows that this is not the case for a root-owned Set-UID program. The executable myuidprog.out runs sleep(int s) as the expected sleep function.

Figure 10: Compiling library and running executables



In Figure 11, I change to the root user where I again export the LD_PRELOAD variable. Now running the root-owned file myuidprog.out runs my false code instead of the standard sleep(int s).

Figure 11: Running executable as root owner/user

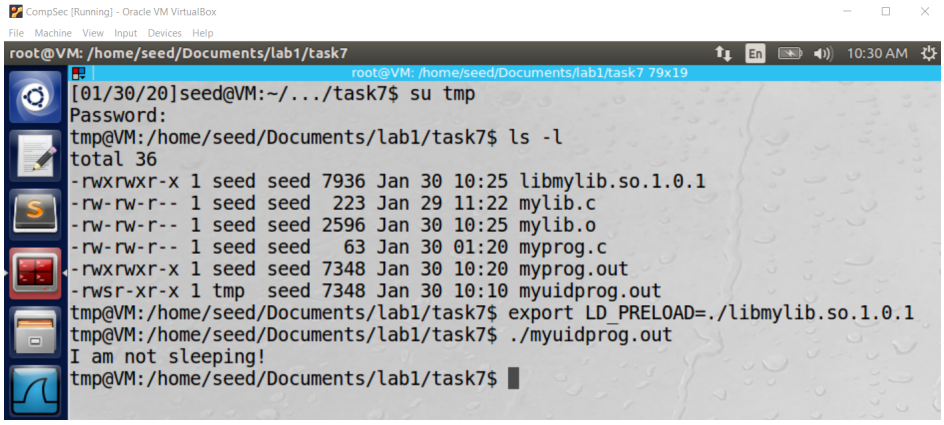


In Figure 12, I change to a temporary user called tmp. I again export the LD_PRELOAD variable. Now running the tmp-owned file myuidprog.out (note that this is a Set-UID program) runs my code in the dynamically linked library instead of the standard sleep(int s).

Based on the examples in this test. It appears that the function sleep(int s) in my dynamically linked library is called when the user executing is the same as the owner. As evidence for this, the only situation where "I am not sleeping!" was not printed to the console was when seed ran root-owned program myuidprog.out. Based on this, it seems that, for Set-UID programs, the child process does not receive LD* environment variables from the parent, but instead from the owner's environment. To test this, we will use the code from Task 4 and reference Task 5.

In Figure 13, I change the ownership of the executable system-test.out to root and export a LD_PRELOAD variable. As seen in Figure 14, the variable LD_PRELOAD is printed out.

Figure 12: Running executable with tmp as owner



However, recall that system-call.out is not a Set-UID program, it is only owned by root. In Task 5, I noted that I could not find the LD_LIBRARY_PATH variable in the environment variables output. Additionally, the executable ran in Task 5 was a Set-UID program. So this fits with my hypothesis that Set-UID programs ignore LD* variables of their parent and inherit from their owner. If this were not the case, then an attacker could easily manipulate a root-owned program with a false dynamically linked library and abuse root power. So, if I am correct, then it is a good security measure.

Figure 13: Showing commands

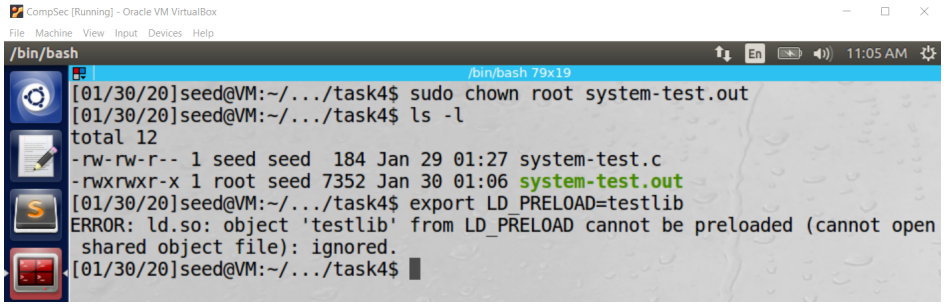
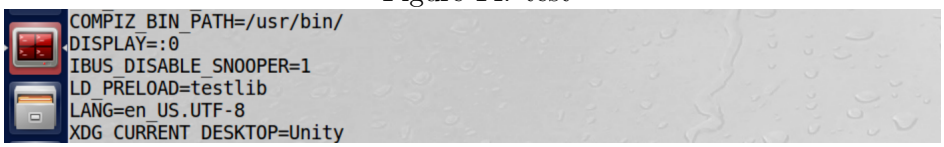


Figure 14: test

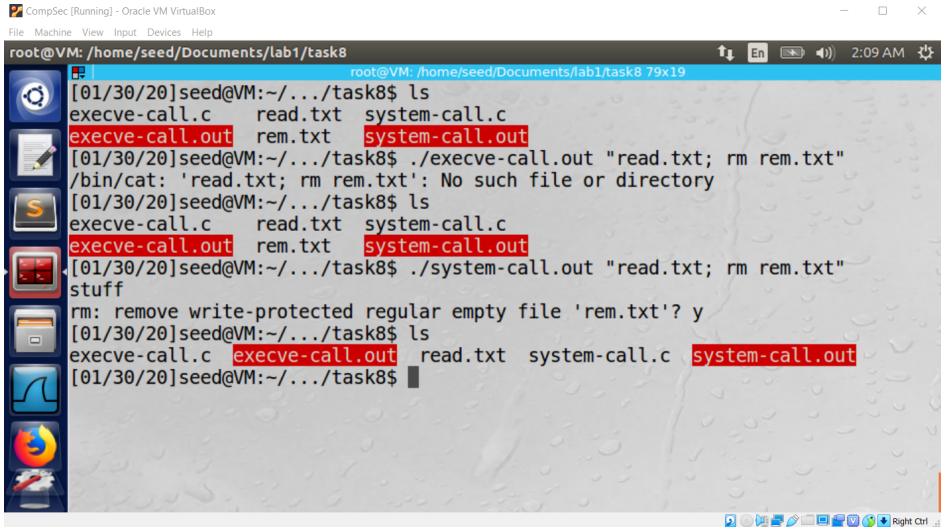


This is an especially dangerous attack since the attacker does not even have to really know what source code entails. As an attacker, you could dynamically link a common function like printf() and then overwrite that function to do what ever you wanted (so long as you are able to add the library to the LD_PRELOAD environment variable).

Task 8

Task 8 shows a vulnerability in calling system() that does not exist when calling execve(). In Figure 15, each executable has the same intention. A user runs the program with a file name as an argument, the executable then prints the contents of that file to the screen. However, on executable uses system() while the other uses execve(). For execve(), if a user were to try to tack on extra commands to the end of their file name, they would get an error (see Figure 15). While in system(), a user is free to tack on extra commands. This is because system() runs a full on command line, so the extra commands are executed as a terminal would execute them. This is made even more dangerous by the fact the system-call.out is a Set-UID program and is able to have full access any file (since they would have root privileges). I was even able to remove the write protected file rem.txt. In the context of the auditing example, the auditor would be able to do whatever they wanted to the system. On the other hand, execve() completely seperates its commands and arguments.

Figure 15: Showing differences between system() and execve()



Task 9

Task 9 focuses on capability leaking. A program can change its privileges at run-time by using the setuid() function. In edit-zzz.out, setuid() is used to change the effective user id to the real user id (that of the owner). While it does this successfully, setuid() does not tie up all the loose ends. For example, in edit-zzz.out opens etc/zzz with root permissions and does not close it before downgrading privileges. This means that, even with root privileges revoked, edit-zzz.out can continue to change the root file zzz. This is shown below in Figure 16.

Figure 16: Showing capability leaking