**2.1 – Task 1:** Manipulating Environment Variables

-2.1.1

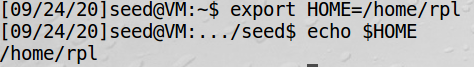


…





-2.1.2



**2.2 – Task 2:** Passing Environment Variables from Parent Process to Child Process



As you can see above, the diff command did not return any differences between the 2 output files. Therefore, we now know from this experiment (printenv rom the child process vs. printenv from the parent process) that child processes ***do*** inherit environment variables from their parent processes.

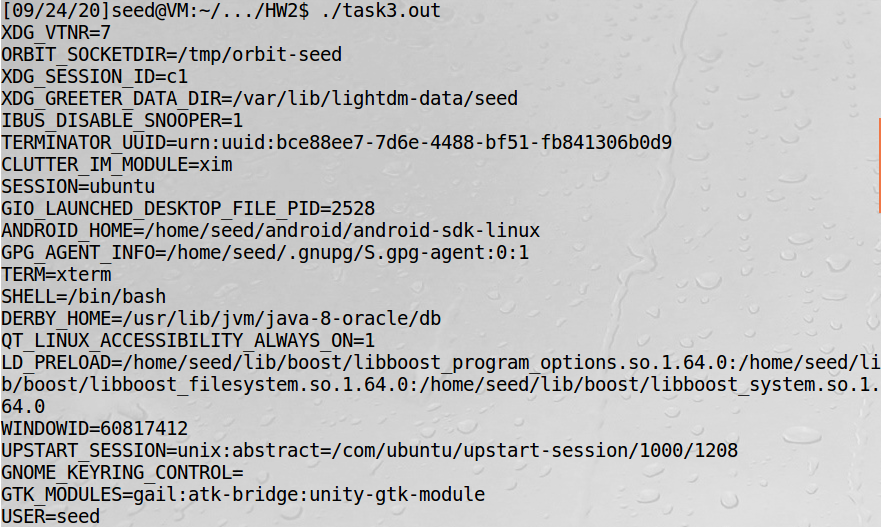
**2.3 – Task 3:** Environment Variables and execve()

-2.3.1



Compiling and running the initial program as is produced no output.

-2.3.2



Compiling and running the program after changing the execve() call prints out the environment variables.

-2.3.3

From this, we can conclude that the execve() function gets its “environment” from the *envp* parameter which is the third parameter passed to execve(), which in this case was environ. When we pass this environ variable to the execve call, it now knows where to run the /usr/bin/env program and therefore, where to look for the environment variables that it should print out.

**2.4 – Task 4:** Environment Variables and system()

-2.4.1



**2.5 – Task 5:** Environment Variables and Set-UID Programs

-2.5.2

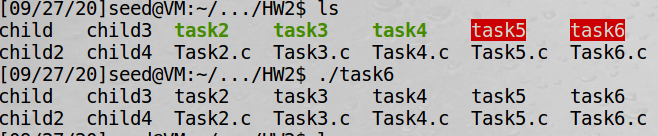


-2.5.3

I’m not sure what results from this step are supposed to be “surprising”, but I’m not sure I see anything. Basically, the results of the program are the results of if I typed the ‘printenv’ command. All variables that I exported in Step 2 made it into the Set-UID child process, I believe.



**2.6 – Task 6:** The PATH Environment Variable and Set-UID Programs



As you can see, running the ./task6 program does have the desired effect of printing out the results of the “ls” command, so it would seem that the code is running with root privilege (since it was able to do this and the program’s owner is root). I’m not sure if this is anything of note, but as you can see in the above screenshot, the result of the program run is actually slightly different than that of a normal “ls” command. The executable files are not highlighted/in a different text color. I’m not sure if this is normal behavior or not, but the program is still achieving the desired output.

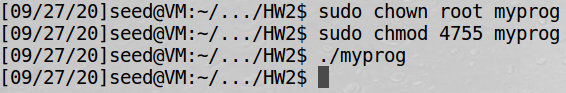
**2.7 – Task 7:** The LD PRELOAD Environment Variable and Set-UID Programs

-2.7.2

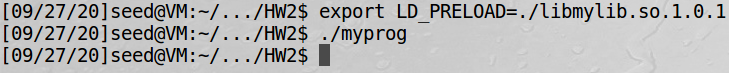
Regular program, normal user



Set-UID root program, normal user

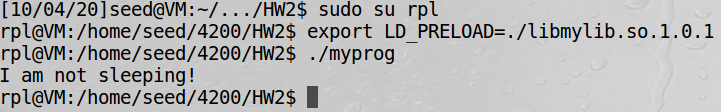


Set-UID root program, LD\_PRELOAD exported



Set-UID user1 program, LD\_PRELOAD exported in different user’s account

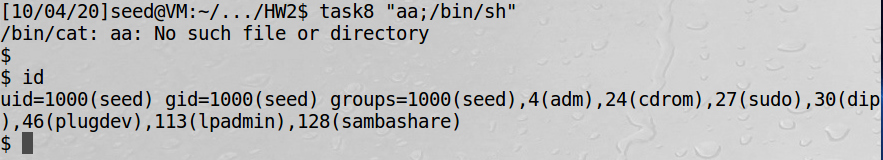




So the reason we see different behavior here is because of the way the environment variables are set here, namely the LD\_PRELOAD variable. When running as a regular program and as a normal user in the same directory as libmylib.so.1.0.1, the typical sleep() function gets overridden by the “I am not sleeping!” function. However, when we change it to be a root-owned Set-UID program, this does not happen and the normal sleep() function is called. Then, when we change the user to the rpl user (user1) and it is not root-owned anymore, the function is overridden.

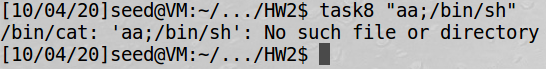
**2.8 – Task 8:** Invoking External Programs Using system() versus execve()

-2.8.1



We can trick the system into opening a root shell by running it with this “file name” that actually becomes part of the code due to the security vulnerabilities of system().

-2.8.2

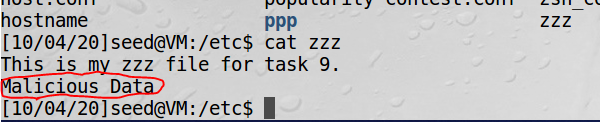


After switching out the system() call for an execve() call, we can see that the attack in the previous step no longer succeeds in opening a root shell. This is because unlike the system() call (where system just takes “command” as one parameter), the execve() call separates the command from the following command-line arguments. Execve() is called as follows: execve(v[0], v, NULL); Here we can see that the command name and the data (the following arguments, which are, in this case, “aa;/bin/sh”) are clearly separated so there is no way for the user data to become part of the code.

**2.9 – Task 9:** Capability Leaking







The file /etc/zzz was modified with the “malicious data” because the 0644 permission that the file was set to allowed the owner (root was the owner of both the ./task9 executable and the zzz file) to read and write to the file. In other words, since zzz had permission 0644, root (which was running ./task9) still had the permission to write the malicious data, despite the setuid() attempt in the code to relinquish root privileges.