**Task 1: Manipulating Environmental Variables**

printenv

A screenshot of a computer program

Description automatically generated

printenv PWD

A black and white text

Description automatically generated

env | grep PATH

A screenshot of a computer program

Description automatically generated

export MY\_VAR=”test”

unset MY\_VAR

A screenshot of a computer

Description automatically generated

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

gcc env\_variables.c -o a.out

a.out > child

child:

A screenshot of a computer

Description automatically generated

When the program was executed, the environment variables of the child process were printed. This output was saved to a file named child.

comment out printenv() and uncomment the other printenv():

gcc env\_variables.c -o a.out

a.out > parent

parent:

A screenshot of a computer

Description automatically generated

When the modified program was executed, the environment variables of the parent process were printed. This output was saved to a file named parent.

Comparing child and parent:

diff child parent

A black and white text

Description automatically generated with medium confidence

No differences were observed between the two files, indicating that the environment variables for both the child and parent processes were the same.

The child process inherits the environment variables of the parent process. This is evident from the absence of any differences between the two sets of environment variables printed by the child and parent processes. Thus, when a new process is created using fork(), the environment variables are shared and passed on from the parent to the child. This mechanism allows for consistency and continuity in the environment settings across process hierarchies.

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

Code was written into a file called task3.c and then compiled into task3prog

gcc task3.c -o task3prog

When the program was executed, no environment variables were printed. This is because execve() was called without passing any environment variables (the third argument was set to `NULL`).

A screenshot of a computer

Description automatically generated

The program was modified to pass the environment variables to the new program (`/usr/bin/env`). This time when the program was executed, the environment variables of the calling process were printed. This is because the `environ` variable, which contains all of the environment variables, was passed to execve().

A screenshot of a computer program

Description automatically generated

Based on the observations from these two programs it is observed that:

When execve() is invoked without passing environment variables, the new program does not inherit any environment variables from the calling process. If you want the new program to inherit the environment variables, you must explicitly pass the environ variables to the execve() function. While the new program replaces the calling process’s memory space, the inheritance of the environment variables is not automatic and depends on whether they are explicitly passed or not.

**Task 4: Environment Variable and system()**

task4.c compiled to task4prog and then executed

A screenshot of a computer

Description automatically generated

When the program is executed, it prints the environment variables of the current process. This is because the system() function internally invokes `/bin/sh` to execute the given command and this shell inherits the environment variables of the calling process.

The system() function by design ensures that the called command inherits the environment variables of the calling process. This behavior is expected since system() executes the command via a shell, which in turn inherits the environment of its parent process. Thus, any program or command invoked through system() will have access to the parent’s environment variables, making the propagation of environments settings consistent when using this function.

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

gcc task5.c -o task5prog

sudo chown root task5prog

sudo chmod 4755 task5prog

export PATH=$PATH:/some/random/path

export LD\_LIBRARY\_PATH=/some/other/random/path

export ANY\_NAME=my\_custom\_test\_variable

A screenshot of a computer program

Description automatically generated

When executed:

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

When the Set-UID program task5prog is executed, it will print the environment variables of the current process. For most environment variables, including the ANY\_NAME one that I created, they will be inherited by the Set-UID program as seen above in the screenshots. The LD\_LIBRARY\_PATH is not seen above because it is a special environment variable that can change the program behavior drastically. I think that it is not seen here because the system automatically cleared that variables for Set-UID binaries to prevent potential misuse.

Set-UID programs do inherit environment variables, but there are mechanisms in place to prevent potential misuse, especially when it comes to variables that can alter the behavior of a program in a way that could be abused for privilege escalation.

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

A screenshot of a computer

Description automatically generated

Creating a malicious ls program:

#include <stdio.h>

void main()

{

printf("Malicious code executed!\n");

}

Manipulate path variable to run malicious ls before real ls

export PATH=.:$PATH

When executing task6prog

A screenshot of a computer

Description automatically generated

When task6prog is executed, instead of the actual ls command running, my malicious ls command is run which displays the “Malicious code executed” message. When the Set-UID program uses system(ls), it looks for the ls command in directories specified in the PATH variable. I placed my current directory containing the malicious ls at the beginning of PATH so that the system executes my ls instead of the real one. Since task6prog is Set-UID root, the malicious ls runs with root privileges.

Using relative paths in Set-UID programs, especially with the system() function, is hazardous due to the potential manipulation of the PATH variable. Malicious users can exploit this to execute arbitrary code with elevated privileges. Developers should always use absolute paths in such programs and be cautious about potential environment variable manipulations.

**Task 7: The LD\_PRELOAD Environment Variable and Set-UID Programs**

Programs and code were created and compiled as outlined in the lab.

**myprog as regular program and normal user:**

A close up of a computer screen

Description automatically generated

“I am not sleeping!” is printed, indicating that the custom library’s sleep() function is invoked.

**myprog as Set-UID root program and run as normal user:**

A screenshot of a computer screen

Description automatically generated

The program appears to sleep for 1 second without printing anything, indicating that the custom sleep() function is not called.

**myprog as Set-UID root program, export LD\_PRELOAD environment variable in the root account and run it:**

A screenshot of a computer

Description automatically generated

For the Set-UID root program executed as root with the LD\_PRELOAD environment variable set under the root account, “I am not sleeping!” is printed. Which indicates that the custom library’s sleep() function is called.

**myprog as a Set-UID user1 program, export LD\_PRELOAD environment variable to a different user’s account and run it:**

**A screenshot of a computer program

Description automatically generated**

For the Set-UID user1 program executed by another user with LD\_PRELOAD set, the program sleeps for 1 second and there is nothing printed out. This indicates that the malicious sleep() is not called.

Regular programs inherit the LD\_PRELOAD environment variable and can be influenced by it. This can be exploited for malicious activities. For Set-UID programs, the LD\_PRELOAD environment variable from the user’s environment is not inherited when the program is executed. This is a security feature to prevent malicious library injection attacks on Set-UID programs. When executing the Set-UID program as the actual owner (root in this test), the LD\_PRELOAD set in the owner’s environment does not affect execution. For the last case, the behavior is consistent with the security measure against LD\_PRELOAD in Set-UID programs. The main takeaway is that the dynamic loader/linker treats LD\_\* environment variables specifically for Set-UID programs to prevent privilege escalation attacks. Normal users cannot influence the behavior of Set-UID programs with LD\_PRELOAD, but the actual owner of the program can.

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

gcc task8.c – o task8prog

touch testfile.txt

The first step can be tested by the creation of a file called testfile.txt and seeing if it can be removed using the program. First checking if the program can read the file:

A screenshot of a computer

Description automatically generated

Now attempting to remove the file using:

./task8prog; rm testfile.txt

A screenshot of a computer

Description automatically generated

The file was successfully removed. Bob would be able to remove a file that is not writable to me (Bob). Bob is able to do this because he can exploit the fact that system() invokes a shell and could interpret shell metacharacters. When system() is invoked, it calls /bin/sh -c command, where command is the string passed to system(). This context, ; allows us to execute multiple commands sequentially. So, after the cat command, the rm command is executed, deleting the target file.

**execve():**

The same command did not have any effect on the target file name when run. This is because execve() directly executes a given program and does not interpret any shell metacharacters. Hence, there is no risk of injection attacks as seen with system(). When Bob tries to pass the filename; rm as an argument, the program would try to display a file name named as “filename; rm” and would no interpret the ; as a command separator like the shell would. In this case, execve() will treat everything as part of the filename.

In conclusion, using system() in Set-UID programs is dangerous because it can interpret shell metacharacters, making it vulnerable to command injection attacks. An attacker can chain commands to carry out malicious activities, potentially leading to privilege escalation. Using execve() is safer because it does not interpret shell metacharacters. When invoking external programs, it is always better to use a more direct function like execve() rather than a function like system() that invokes a shell. Vince’s intention to restrict Bob to only read files and not modify them failed when using the system() function.

**Task 9: Capability Leaking**

sudo touch /etc/zzz/

sudo chmod 0644 /etc/zzz

Compile Set-UID program called task9.c and task9prog

gcc task9.c -o task9prog

sudo chown root task9prog

sudo chmod 4755 task9prog

Run program and observe:

A computer screen shot of a program

Description automatically generated

The program was able to open /etc/zzz successfully and got a file descriptor “fd”. The program opened the file and then relinquished the root privilege by calling “setuid(getuid()). This action should typically make the process lose all root capabilities. However, the child process, even after relinquishing the root privileges, was still able to write “Malicious Data” to the file /etc/zzz using the file descriptor “fd”. This indicates that although the effective user ID was changed to a non-root user, the file descriptor, which was obtained while the process had root privileges, still retained its capability. Using sudo cat /etc/zzz we can confirm that “Malicious Data” has been appended to it.

Capability leaking is a major security concern in Set-UID programs. In this case, the file descriptor is the “capability: that leaked from a privileged context to a non-privileged one. Even though the effective user ID was set to a non-root user, the child process was still able to operate with the file descriptor that was opened while the program had root privileges. The key takeaway is that it’s essential not only to drop privileges appropriately but also to ensure that any capabilities (like file descriptors) that were obtained during privileged operations are handled appropriately, cleaned up, or closed before transferring control to a potentially untrusted part of the code or relinquishing privileges.