1. **Overview**

The learning objective of this lab is for students to understand how environment variables affect program and system behaviors. Environment variables are a set of dynamically named values that can affect the way running processes behave. They are used by most operating systems since they were introduced to Unix in 1979. Although environment variables affect program behaviors, how they achieve this is not well understood by many programmers. As a result, if a program uses environment variables, but the programmer does not know that they are used, the program may have vulnerabilities.

After completing this laboratory, students will understand how environment variables work, how they are propagated from parent to child processes, and how they affect system/program behaviors. We are particularly interested in how environment variables affect the behavior of Set-UID programs, which are usually privileged programs.

This lab covers the following topics:

* Environment Variables
* Set-UID Programs
* Securely Invoking External Programs
* Capability Leaking
* Dynamic Loader/Linking

1. **Lab Tasks**

Files needed for this lab are included in the Lab\_01\_Env\_Variables\_SetUID folder

* 1. **Task A: Manipulating Environment Variables**

In this task, we study the commands that can be used to set and unset environment variables. We will be using the Bash shell for these tasks. 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

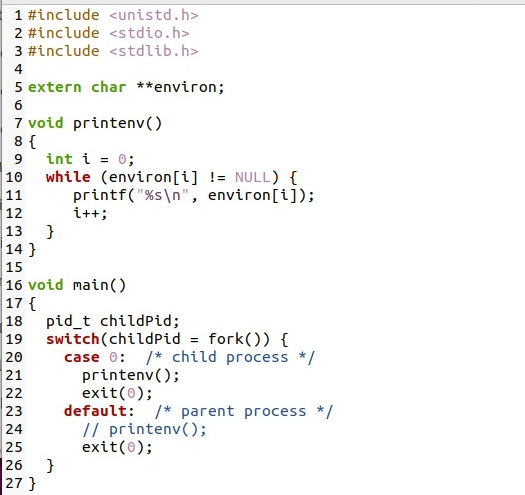
for this lab). Please do the following tasks:

* 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".
* Use export and unset to set or unset environment variables. It should be noted that these two commands are not separate programs; they are two of the Bash’s internal commands (you will not be able to find them outside of Bash).
  1. **Task B: Passing Environment Variables from a Parent to a Child Process**

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.

**Step 1.** Please compile and run the following program. The program can be found in the setup folder; it can be compiled using "gcc myprintenv.c", which will generate a binary called a.out. Let’s run it and save the output into a file using "./a.out > file".



Listing 1: myprintenv.c

**Step 2.** Comment out the printenv() statement in the child process case and uncomment the printenv() statement in the parent process case. Compile and run the code again saving the output in another file.

**Step 3.** Compare the differences between the two outputs using the diff command. Please describe your observations and conclusions on your submission sheet.

* 1. **Task C: Environment Variables and execve()**

In this task, we will examine 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, saving its output into a file. This program simply executes a program called /usr/bin/env, which prints out the environment variables of the current process.



Listing 2: myenv.c

**Step 2.** Change the invocation of execve() to the following. Compile and run your program again saving the output to a file.

execve(“/usr/bin/env”, argv, environ);

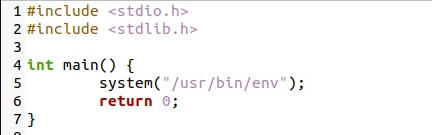
**Step 3.** Compare the differences between the two outputs using the diff command. Please describe your observations and conclusions on your submission sheet.

* 1. **Task D: Environment Variables and 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, using system(), the environment variables of the calling process are passed to the new program /bin/sh.

**Step 1.** Please compile and run the following program to verify this.



* 1. **Task E: Environment Variables and Set-UID Programs**

Set-UID is an important security mechanism in the Unix operating system. When a Set-UID program runs, it assumes the owner’s privileges. For example, if the program’s owner is root, 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 since it escalates the user’s privilege, it is quite risky. Although the behaviors of

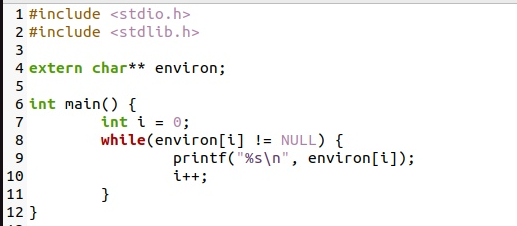
Set-UID programs are decided by their program logic, not by users, users can indeed affect the behaviors via environment variables. To understand how Set-UID programs are affected, let us first figure out whether environment variables are inherited by the Set-UID program’s process from the user’s process.

**NOTE:** The system(cmd) function executes the /bin/sh program first, and then asks this shell program to run the cmd command. In Ubuntu 22.04 (and several versions before), /bin/sh is actually a symbolic link pointing to /bin/dash. This shell program 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. 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 shell that does not have such a countermeasure. We have installed a shell program called zsh in our Ubuntu

20.04 VM. We use the following commands to link /bin/sh to /bin/zsh:



**Step 1.** Write the following program that can print out all environment variables in the current process.



**Step 2.** 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:

* PATH
* LD\_LIBRARY\_PATH
* ANY\_NAME (this is an environment variable defined by you, so pick whatever name you want)

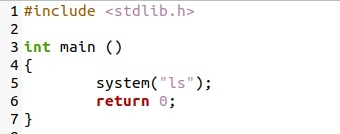
These environment variables are set in the user’s shell process. Now, run the Set-UID program from Step 2 in your shell. After you type the name of the program in your shell, the shell forks a child process, and uses the child process to run the program. Please check whether all the environment variables you set in the shell process (parent) get into the Set-UID child process. Describe your observations in your submission sheet. If there are surprises to you, describe them.

* 1. **Task F: The PATH Environment Variable and 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; and 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.



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, change its owner to root, and make it a Set-UID program. Can you get this Set-UID program to run your own malicious code, instead of /bin/ls? If you can, is your malicious code running with root privilege? Describe and explain your observations in your submission sheet.

* 1. **Task G: The LD\_PRELOAD Environment Variable and 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.

**Step 1.** First, we will see how these environment variables influence the behavior of the 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.



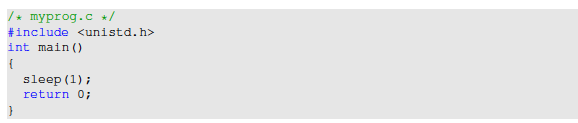
* + 1. We can compile the above program using the following commands



* + 1. Now, set the LD\_PRELOAD environment variable



* + 1. 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, run myprog under the following conditions and log your observations in your submission sheet.

* 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.

You should be able to observe different behaviors in the scenarios described above, even though you are running the same program. Can you figure out what causes the difference. Environment variables play a role here. Please design and perform an experiment to figure out the main causes, and explain why the behaviors are different.

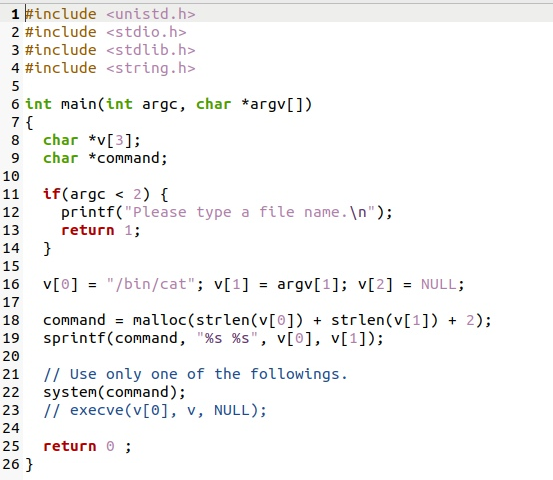
* 1. **Task H: Invoking External Programs using system() vs. execve()**

Although system() and execve() can both be used to run new programs, system() is quite dangerous 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 superuser 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.



Listing 3 catall.c

**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 in your submission sheet.

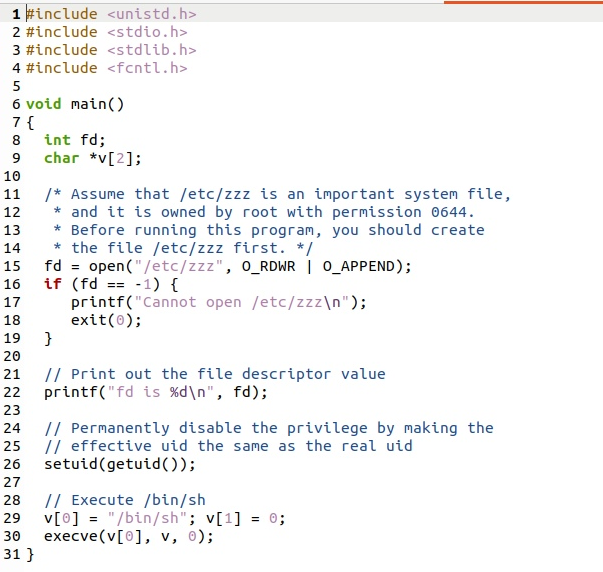
* 1. **Task I: Capability Leaking**

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 hand over 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. Can you exploit the capability leaking vulnerability in this program? Describe your approach in your submission sheet. The goal is to write to the /etc/zzz file as a normal user.



Listing 4 cap\_leak.c

1. **Submission**

Once you have finished all of the tasks, email your submission to [brian.swenson@gtri.gatech.edu](mailto:brian.swenson@gtri.gatech.edu). Include in your submission your name and the name of your partner if you worked on the lab in pairs.

1. **References**

This lab is based on the seed 2.0 labs created by Wenliang Du and is protected by the following license:

