Environment Variable and Set-UID Program LAB

Contents

Lab 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 dynamic named values that can affect the way running processes will behave on a computer. They are used by most operating systems, since they were introduced to Unix in 1979. Although environment variables affect program behaviors, how they achieve that 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. In this lab, students will understand how environment variables work, how they are propagated from parent process to child, 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 invoke external programs
- Capability leaking
- Dynamic loader/linker

Task 1: Manipulating Environment Variables

In this task, we study the commands that can be used to set and unset environment variables. We are using Bash in the seed account.. 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".

To print out all the environment variables execute the following, in your terminal (normal VM Terminal)

Command:

```
$ printenv
```

To print out a particular environment variable for example - PWD, execute the following in your terminal

Command:

```
$ printenv 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 Bash's internal commands (you will not be able to find them outside of Bash).

We will create an environment variable 'foo' and assign our SRN as the value. We will then export it, print it out and later unset it. Execute the following commands on your terminal for the same -

Command:

```
$ export foo='PES1UG20CS000'
$ printenv foo
$ unset foo
$ printenv foo
```

Please take appropriate screenshots of all the steps, and explain your observations in detail.

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

Note - Please remember to change directory to your lab setup folder in your terminal before proceeding further

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 **'myprintenv.c'** program, and describe your observation. The program can be found in the Labsetup 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". Please execute the following commands in your terminal for Step 1 -

Command:

```
$ gcc myprintenv.c
$ a.out > child
```

We store the output of the Step in child.

Step 2. Now **comment out** the printenv() statement in the **child** process case (Line ①), and **uncomment** the printenv() statement in the **parent** process case (Line ②). Compile and run the code again, and describe your observation. Save the output in another file.

For further clarity, please check the below code for myprintenv.c -

```
void main()
{
        pid_t childPid;
        switch(childPid = fork()) {
        case 0: /* child process */
        printenv();  // 1
        exit(0);
        default: /* parent process */
        //printenv(); // 2
        exit(0); }
}
```

Please use nano (code editor) using the command - "nano myprintenv.c" to access the code. Note nano should be used further in this lab to create and edit programs directly from the Terminal.

We uncomment out 2 and comment 1. Then we execute the following commands in the terminal -

Command:

```
$ gcc myprintenv.c
$ a.out > parent
```

Step 3. Compare the difference of these two files using the diff command. Please draw your conclusion.

Command:

```
$ diff child parent
```

Please show appropriate screenshots for each step and explain your observations in detail.

Task 3: Environment Variables and execve()

In this task, we study 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?

The code **myenv.c** is given below, please refer to it. The same has been provided in the labsetup folder.

```
#include <unistd.h>
extern char **environ;
int main()
{
        char *argv[2];
        argv[0] = "/usr/bin/env";
        argv[1] = NULL;
        execve("/usr/bin/env", argv, NULL);          ①
        return 0 ;
}
```

Step 1. Please compile and run the above program, and describe your observation. This program simply executes a program called /usr/bin/env, which prints out the environment variables of the current process.

Command:

```
$ gcc myenv.c
$ ./a.out
```

Step 2. Change the invocation of execve() in Line ① to the following; describe your observation.

Change - execve("/usr/bin/env", argv, NULL); to execve("/usr/bin/env", argv, environ); in **Line ① (refer to the code given above, change using nano).**

Step 3. Execute the code and draw your conclusion regarding how the new program gets its environment variables. Execute the following commands:

Command:

```
$ gcc myenv.c
$ ./a.out
```

Please show appropriate screenshots for each step and explain your observations in detail.

Task 4: 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. Please compile and run the following program to verify this.

Create a program sysenv.c (Note this is not provided with the labsetup folder, so use the command

```
$ nano sysenv.c
```

and copy the code below)

sysenv.c

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        system("/usr/bin/env");
        return 0 ;
}
```

Now after you have successfully created the program file, execute the following commands:

Command:

```
$ gcc sysenv.c -o sysenv
$ ./sysenv
```

Please show appropriate screenshots and explain your observations in detail.

Task 5: Environment Variable and Set-UID Programs

Set-UID is an important security mechanism in Unix operating systems. 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

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

Create a program setuidenv.c (Note this is not provided with the labsetup folder, so use nano and copy the code below)

setuidenv.c

```
#include <stdio.h>
#include <stdlib.h>
extern char **environ;
int main()
        {
                int i = 0;
                while (environ[i] != NULL) {
                printf("%s\n", environ[i]);
                i++;
                }
        }
```

Step 2. Compile the above program, change its ownership to root, and make it a Set-UID program.

Command:

```
$ gcc setuidenv.c -o setuid
$ sudo chown root setuid
$ sudo chmod 4755 setuid
$ ls -l setuid
```

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 (they may have already exist):

- PATH
- LD LIBRARY PATH
- ANY NAME (this is an environment variable defined by you, so pick whatever name you want).

Command:

```
$ export PATH=/home/seed:$PATH
$ export LD_LIBRARY_PATH=/home/seed:$LD_LIBRARY_PATH
$ export task5=task5
$ ./setuid
```

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. From the environment variables printed, **please check whether**

all the environment variables you set in the shell process (parent) get into the Set-UID child process. Describe your observation. If there are surprises to you, describe them.

Please restart the lab (end terminal) or reset the PATH variable before proceeding to the next task.

Command to reset - PATH=$(getconf PATH)

Task 6: 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; 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.

We create a program **myls.c**, a Set-UID program to execute the /bin/ls command; however, the programmer only uses the relative path for the ls command, rather than the absolute path:

Create a program myls.c (Note this is not provided with the labsetup folder, so use nano and copy the code below)

myls.c

```
int main()
{
        system("ls");
        return 0;
}
```

Now we make it into a SET-UID program and execute it to test it's functionality.

Command:

```
$ gcc myls.c -o myls
$ sudo chown root myls
$ sudo chmod 4755 myls
$ ls -l myls
$ ./myls
```

Now our next step would be to get this Set-UID program to run our own malicious code, instead of /bin/ls.

We create our malicious code ls.c. **Note this is not provided with the labsetup folder, so use nano and copy the code below)**

ls.c

```
#include<stdio.h>
#include<unistd.h>
int main()
        {
                printf("This is the malicious program!\n");
                printf("real uid is %d\neffective uid is %d\n",getuid(),geteuid());
                return(0);
        }
```

The system(cmd) function executes the /bin/sh program first, and then asks this shell program to run the cmd command.

In Ubuntu 20.04 (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.

Command:

```
$ gcc ls.c -o ls
$ sudo rm /bin/sh
$ sudo ln -sf /bin/zsh /bin/sh
$ export PATH=/home/seed/Labsetup:$PATH
$ echo $PATH
$ ./myls
```

Describe and explain your observations with appropriate screenshots. If you can, is your malicious code running with the root privilege?

Please restart the lab (end terminal) or reset the PATH variable before proceeding to the next task.

To reset it -

Command:

```
$ sudo ln -sf /bin/bash /bin/sh
$ PATH=$(getconf PATH)
```

Task 7: 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 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:

   mylib.c

   ```
   #include <stdio.h>
   void sleep (int s)
   {
           /* If this is invoked by a privileged program,
           you can do damages here! */
           printf("I am not sleeping!\n");
   }
   ```

2. We can compile the above program using the following commands (in the -lc argument, the second character is `):

Command:

```
$ gcc -fPIC -g -c mylib.c
$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
```

3. Now, set the LD PRELOAD environment variable:

Command:

```
$ export LD_PRELOAD=./libmylib.so.1.0.1
```

4. Finally, compile the following program myprog.c, and in the **same directory (Labsetup)** as the above dynamic link library libmylib.so.1.0.1:

Create myprog.c using nano and copy the code given below

myprog.c

```
#include <unistd.h>
int main()
{
        sleep(1);
        return 0;
}
```

Step 2. After you have done the above, please run myprog under the following conditions, and observe what happens

1. Make myprog a regular program, and run it as a normal user

Command:

```
$ gcc myprog.c -o myprog
$./myprog
```

2. Make myprog a Set-UID root program, and run it as a normal user.

Command:

```
$ sudo chown root myprog
$ sudo chmod 4755 myprog
$ ./myprog
```

3. Make myprog a Set-UID root program, export the LD PRELOAD environment variable again in the root account and run it.

Command:

```
$ sudo su
# export LD_PRELOAD=./libmylib.so.1.0.1
# ./myprog
#exit
```

4. Make myprog a Set-UID user1 program (i.e., the owner is user1, which is another user account), export the LD PRELOAD environment variable again in a different user's account (not-root user) and run it.

Command:

```
$ sudo adduser user1
```

(enter the details)

```
$ gcc myprog.c -o myprog1
$ sudo chown user1 myprog1
$ sudo chmod 4755 myprog1
$  export LD_PRELOAD=./libmylib.so.1.0.1
$ ./myprog1
```

You should be able to observe different behaviors in the scenarios described above, even though you are running the same program. You need to figure out what causes the difference. Environment variables play a role here. Explain why the behaviors in Step 2 are different. (Hint: the child process may not inherit the LD * environment variables)

Please explain your observations in detail with the appropriate screenshots.

Task 8: Invoking External Programs Using system() versus 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 catall.c (provided with the Labsetup), 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.

Step 1: Compile the program catall.c (provided with the Labsetup), 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?

1. First we create two files - myfile and rootfile

Command:

```
$ cat > myfile
(Enter any sentence)
(Ctrl + D)
$ cat > rootfile
(Enter any sentence)
(Ctrl + D)
$ sudo chown root rootfile
```

2. Then we compile the catcall.c program and make it setuid

Command:

```
$ gcc catall.c -o catall
$ sudo chown root catall
$ sudo chmod 4755 catall
```

3. Now we execute the below command, and pass parameters to the code.

Command:

```
$ ./catall “myfile;rm rootfile”
$ cat rootfile
```

Please explain your observations in detail with the appropriate screenshots.

Step 2: **Comment out the system(command) statement, and uncomment the execve() statement (use nano catall.c);** the program will use execve() to invoke the command. Compile the program, and make it a root-owned Set-UID.

Command:

```
$ cat > rootfile
(Enter any sentence)
(Ctrl + D)
$ sudo chown root rootfile
$ gcc catall.c -o catall
$ sudo chown root catall
$ sudo chmod 4755 catall
$ ./catall "myfile;rm rootfile"
$ cat rootfile
```

Do your attacks in Step 1 still work? Please describe and explain your observations with appropriate screenshots.

Task 9: 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.

1. Create an important file /etc/zzz using the root user

Command:

```
$ su root
# cat > /etc/zzz
(Enter any sentence)
(Ctrl + D)
# exit
$ cat /etc/zzz
$ sudo chown root /etc/zzz
$ sudo chmod 0644 /etc/zzz
```

2. Compile the cap_leak.c program, change its owner to root, and make it a Set-UID program.

Command:

```
$ gcc cap_leak.c -o capleak
$ sudo chown root capleak
$ sudo chmod 4755 capleak
```

3. Run cap_leak.c and write some value

Command:

```
$ ./capleak
```

You will get a new shell and a file descriptor value (in most cases it will be 3)

Now in the new shell execute the following

Command:

```
$ echo “malicious data” > &3
$ cat /etc/zzz
```

Please show appropriate screenshots for each step and explain all your observations in detail.

Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.