PRO.M.IS
security built in

Lab 3: Environment variables

Purpose

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.

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 dv2595 account. 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 seperate programs; they are two of the Bash's internal commands (you will not be able to find them outside of Bash).

Task 2: Passing Environment Variables from Parent Process to 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, and describe your observation. Because the output contains many strings, you should save the output into a file, such as using a.out > child (assuming that a.out is your executable file name).

```
/* child_env.c */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
extern char **environ;

void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}

void main()
{
    pid_t childPid;
    switch(childPid = fork()) {
    case 0: /* child process */
        printenv();
        exit(0);

    default:  /* parent process */
        //printenv();
        exit(0);
    }
}
```

Step 2. Now 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, and describe your observation. Save the output in another file.

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

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?

Step 1. Please compile and run the following 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.

```
/* list_env.c */
#include <stdio.h>
#include <stdlib.h>
extern char **environ;

int main()
{
    char *argv[2];
    argv[0] = "/usr/bin/env";
    argv[1] = NULL;
    execve("/usr/bin/env", argv, NULL);
    return 0 ;
}
```

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

```
execve("/usr/bin/env", argv, environ);
```

Step 3. Please draw your conclusion regarding how the new program gets its environment variables.

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 is passed to the new program */bin/sh*. Please compile and run the following program to verify this.

```
/* system_env.c */
```

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    system("/usr/bin/env");
    return 0 ;
}
```

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, then 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 it escalates the user's privilege when executed, making it 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.

```c
/* setuid_env.c */
#include <stdio.h>
#include <stdlib.h>
extern char **environ;

void 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.

```
$ sudo chown root setuid_env
$ sudo chmod 4755 setuid_env
```

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).

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 observation. If there are surprises to you, describe them.

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. In Bash, you can change the PATH environment variable in the following way (this example adds the directory */home/dv2595* to the beginning of the PATH environment variable):

```
$ export PATH=/home/seed:$PATH
```

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:

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

Please compile the above program, and change its owner to root, and make it a Set-UID program. Can you let this Set-UID program run your code instead of */bin/ls*? If you can, is your code running with the root privilege? Describe and explain your observations.

Note (Ubuntu 20.04 VM): 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 */bin/sh* is actually a symbolic link pointing to the */bin/dash* shell. The dash shell in Ubuntu 20.04 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 *zsh*:

```
$ sudo rm /bin/sh
$ sudo ln -s /bin/zsh /bin/sh
```

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:

```c
/* puts.c: fake */
#include <stdio.h>
int puts(const char *s)
{
    FILE* fout = stdout;
    fprintf(fout, "Cruel world!\n");
    return 0;
}
```

1. Build a dynamic link library. Create the program shown above, and name it *puts.c*. It basically overrides the *puts()* function in *libc*:

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

   ```
   % gcc -fPIC -g -c puts.c
   % gcc -shared -o libputs.so.1.0.1 puts.o -lc
   ```

3. Now, set the LD_PRELOAD environment variable:

```
% export LD_PRELOAD=./libputs.so.1.0.1
```

4. Now, set the LD PRELOAD environment variable:

```
% export LD_PRELOAD=./libputs.so.1.0.1
```

5. Finally,compile the following program *myprog*, and in the same directory as the above dynamic link library *libputs.so.1.0.1*:

```
/* myprog.c */
#include <stdio.h>
int main(int argc, char *argv[])
{
      puts("Hello world");
}
```

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

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

Step 3. 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. Please design an experiment to figure out the main causes, and explain why the behaviors in Step 2 are different. (Hint: the child process may not inherit the LD * environment variables).

Deliverables

Short report (3-6 A4 pages) summarizing the outcome of each task, how you solve it (if solution not already provided in the task description), and answers to task questions (if any). Feel free to use screenshots in the report, if it makes the explanations easier/more comprehensible. For tasks that require you to write a program, please provide the source code as separated text files (so it can easily be compiled for verification).