Environment Variable and Set-UID Program Lab

In this lab, students will understand

- How environment variables work
- How they are propagated from parent process to child
- How they affect system/program behavior

This lab is particularly oriented in how environment variables affect the behavior of Set-UID programs, which are usually privileged programs.

Table of Contents

Overview

On September 24, 2014, a severe vulnerability in Bash was identified. Nicknamed Shellshock, this vulnerability can exploit many systems and be launched either remotely or from a local machine. In this lab, students need to work on this attack, so they can understand the Shellshock vulnerability. The learning objective of this lab is for students to get first-hand experience on this interesting attack, understand how it works, and think about the lessons that we can get out of this attack. This lab covers the following topics:

- Shellshock
- Environment variables
- Function definition in Bash
- Apache and CGI programs

Lab environment. This lab has been tested on our pre-built Ubuntu 16.04 VM, which can be downloaded from the SEED website. https://seedsecuritylabs.org/lab_env.html. Download the June 2019 version of ubuntu 16.04

Lab Tasks

Task 1: Manipulating environment variables

In this task, Study the commands that can be used to set and unset environment variables. Here using Bash in the seed 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 **Command:**

  $ printenv PWD

  or

  $ env | grep PWD

  Give your observation with a screen shot.

- Use export and unset to set 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). Use unset to unset the variable.

Command:

```
$ export foo='test string'
$ printenv foo

$ unset foo
$ printenv foo
```

Give your observation with a screen shot.

Task 2: Inheriting environment variables from parents

In this task, Study how environment variables are inherited by child processes from their parents. 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, Students should 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).

```
/*penv program */
#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);
}
}
```

Commands:

```
$ gcc penv.c
$ a.out>child
$ ls -l child
```

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

Commands:

```
$ gcc penv.c
$ a.out>parent
$ ls -l parent
```

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

Command:

```
$ diff child parent
```

Give your observation with a screen shot.

Task 3: Environment variables and execve()

In this task, 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. Here our interest is 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.

```
/*execenv.c program */
#include<stdio.h>
#include<unistd.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 ;
}
```

Commands:

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

Give your observation with a screen shot.

Step 2: Now, change the invocation of execve() to the following, and describe your observation. (make changes in the program given above)

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

Commands:

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

Give your observation with a screen shot.

Task 4: Environment variables and system()

In this task, 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; excel() 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.

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

Commands:

Cd Desktop/environ_set_uid/ use this location before executing prog

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

Give your observation with a screen shot.

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: We are going to write a program that can print out all the environment variables in the current process.

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

Commands:

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

Give your observation with a screen shot.

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

Commands:

$gcc setuidenv.c -o setuidenv

$sudo chmod 5744 setuidenv

$ la -l setuidenv

Give your observation with a screen shot.

Step 3: In your Bash 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).

Commands:

$printenv PATH

$printenv LD_LIBRARY_PATH

$export LD_LIBRARY_PATH=/home/seed:$LD_LIBRARY_PATH

$printenv LD_LIBRARY_PATH

$printenv task5

$export task5='task5 new variable'

$printenv task5

$env > env_result

$diff setuidenv env_result

$setuidenv > setuidenv_res

$diff setuidenv_res env_result

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.

Give your observation with a screen shot.

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/seed 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:

```
/* myls.c program */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
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.

Commands:

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

Give your observation with a screen shot.

```
/* ls.c program */

#include<stdio.h>
#include<unistd.h>
int  main()
{
printf("\n This is my ls Program\n");
printf("\n my real Uid is :%d\n My Effective uid is:%d\n",  getuid(),geteuid());
return(0);
}
```

Commands:

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

Give your observation with a screen shot.

Task 7: The LD PRELOAD environment variable and Set-UID Programs

In this task, 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 runtime.

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 with 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, students will only study LD_PRELOAD.

Step 1: First, see how these environment variables influence the behavior of dynamic loader/linker when running a normal program. Please follow these steps:

1. Build a dynamic link library. Create the following program, and name it mylib.c. It basically overrides the sleep() function in libc:

```
/*mylib.c  program*/
#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. 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, and it in the same directory as the above dynamic link library libmylib.so.1.0.1:

```
/* myprog.c program */
 #include<unistd.h>
 int  main()
{
sleep(1);
return 0;
}
```

Give your observation with a screen shot.

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 user1 program (i.eThe 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:

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

Give your observation with a screen shot.

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).

In root environment execute the myprog.c program

Command:

```
$ gcc myprog.c -o myprog
$ chmod 4755 mypro
$ ls -l myprog
$ export LD_PRELOAD=./libmylib.so.1.0.1
```

come out of root and check the behavior

Command:

```
$ ls -l myprog
$ export LD_PRELOAD=./libmylib.so.1.0.1
$ whoami
$ seed
$ ./myprog
```

Give your observation with a screen shot.

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 affects 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 a shell has another dangerous consequence, and this time, it has nothing to do with environment variables.

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.

```
 /* sysexecenv.c program */
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char *argv[])
{
char *v[3];
char *command;
if(argc < 2) {
printf("Please type a file name.\n");
return 1;
}
v[0] = "/bin/cat";
v[1] =  argv[1];
v[2] = NULL;
command =  malloc(strlen(v[0]) +  strlen(v[1]) + 2);
sprintf(command, "%s %s", v[0], v[1]);
// Use only one of the following.
system(command);
// execve(v[0], v, NULL);
return 0 ;
}
```

Step 1: Compile the above program, make root its owner, and change it to a 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? (create two files “myfile”(owner is seed) and “root file”(owner is root - using chown cmd))

Command:

```
$ gcc sysexecenv.c -o sys
$ sudo chown root sys
$ sudo chmod 4755 sys
$ ls -l rootfile myfile sys
$ ./sysexecenv “myfile;rm rootfile”
$ ls -l rootfile
```

Give your observation with a screen shot.

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 SetUID (owned by root). Do your attacks in Step 1 still work? Please describe and explain your observations.

Command:

```
$ gcc sysexecenv.c -o exec
$ sudo chown root exec
$ sudo chmod 4755 exec
$ ls -l rootfile myfile exec
$ ./sysexecenv “myfile;rm rootfile”
$ ls -l rootfile
```

Give your observation with a screen shot.

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 privileged 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, and describe what you have observed. Will the file /etc/zzz be modified? Please explain your observation.

```
/* capleak.c program */
    #include<stdio.h>
    #include<stdlib.h>
    #include<fcntl.h>
```

```
#include<unistd.h>
#include<sys/types.h>
void main()
{
int  fd ;
/* Assume that /etc/zzz is an important system file,  and it is owned by root with permission 0644. *  Before running this program, you should create the file  /etc/zzz first. */
fd = open("/etc/zzz", O_RDWR |  O_APPEND);
if (fd == -1) {
printf("Cannot open /etc/zzz\n");
exit(0);
}
/* Simulate the tasks conducted by the  program */
sleep(1);
/* After the task, the root privileges are no longer  needed, it's time to relinquish the root privileges  permanently. */
setuid(getuid());
/* getuid()  returns the real uid */
 if (fork())
{ /* In the parent  process */
close (fd);
exit(0);
}
else {
/* in the child process */
/* Now, assume that the child process is compromised, malicious attackers have injected the  following statements into this process */

write (fd, "Malicious Data\n", 15); close (fd);
}
}
```

Command:

```
$ gcc capleak.c -o capleak
$ sudo chown root capleak
$ sudo chmod 4755 capleak
$ ls -l
$ capleak $
$ cat /etc/zzz
$ ./capleak
$ cat /etc/zzz
```

Give your observation with a screen shot.

Submission

You need to submit a detailed lab report to describe what you have done and what you have observed, including screenshots and code snippets. You also need to provide explanation to the observations that are interesting or surprising. You are encouraged to pursue further investigation, beyond what is required by the lab description. Please submit in word or PDF format only.