CSCI 191T – Computer Security

Assignment 1 Key: SetUID and Env.Variables

Instructions:

There are two sections in this assignment. For Section 1, answer all questions completely. For Section 2, execute the code (using files from Assignment1_Files.zip) and answer all questions.

The final submission is a single report (PDF file). You need to submit a detailed report, with:

Section 1 - Detailed answers and observations.

Section 2 - 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. Also list the important code snippets followed by explanation. **Simply attaching code or screenshot without any explanation will not receive credits.**

Section 1

10 points each.

1. Alice runs a Set-UID program that is owned by Bob. The program tries to read from /tmp/x, which is readable to Alice, but not to anybody else. Can this program successfully read from the file?
   Key points:
   a. No, the program will not read from the file when Alice runs it.
   b. EUID is Bob, therefore, even if Alice is the only one that can read the file, Alice will still be denied permission.

2. Both system() and execve() can be used to execute external programs. Why is system() unsafe while execve() is safe?
   Key points:
   a. system() function:
      1. Program name and arguments all passed as one input string.
      2. The shell interprets this string and executes the program included in the string.
      3. Data can be interpreted as code.
      4. Untrusted code may be executed.
   b. execve() function:
      1. The function itself takes separate arguments, first – the program, second – data/args needed by the program.
      2. Code and data are separate, therefore data is not interpreted as code.

3. The superuser wants to give Alice a permission to view all the files in the system using the **more** command. He does the following:
   $ cp /bin/more /tmp/mymore
   $ sudo chown root /tmp/mymore
   $ sudo chmod 4700 /tmp/mymore

   Basically, the above commands turns /tmp/mymore into a Set-UID program. Right now, because the permission is set to 4700, other users cannot execute the program. The superuser uses another command (now shown) to grant the execution permission only to Alice. We are not assuming that Alice is completely trusted. It is OK if Alice can only read other people's files, but it is not OK if Alice can gain any privilege beyond that, such as writing to other people's files. Read the manual of the **"more"** program and find out what Alice can do to gain more privilege.

   Key points:

   a. *more* does allow any commands to be run using "!<cmd>" within the more program.
   b. *more* will invoke a shell program to run the command. Therefore, the *SHELL* environment variable holds the key.
   c. No problem if /bin/bash is the shell being invoked, it automatically drops privilege.
   d. Alice, can change SHELL variable to point to a zsh, instead of bash, which does not have countermeasures and helps Alice gain higher privileges

4. Assume that you have a file that you would allow other users to read, only if a user's ID is smaller than 1000. Describe how you can actually achieve this.

   Key points:

   a. First, I can make a setuid program that reads the file.
   b. Within the setuid program I can call getuid() to get the real user's ID. Eg, If Alice is running my setuid program, getuid() will return Alice's user ID and geteuid() will return my user ID (effective user).
   c. Using the return value from getuid(), the program can decide to continue if value > 1000, else quit.

5. What is the difference between environment variables and shell variables?

   Key points

   a. Shell variables:
      1. Are internal variables maintained by a shell program.
      2. Can be used within a shell script or command.
      3. Can be created, assigned and deleted using shell commands.
      4. The shell program contains a local copy of environment variables which are the initial shell variables (changes don't affect each other).
      5. When a child process is created through a shell program, the environment variables and exported shell variables become the child process's environment variables

b. Environment variables:
   1. Set of dynamic name-value pairs stored within a process (does not have to be a shell process) .
   2. The initial set of environment variables are set during system startup or by init and startup scripts.
   3. In case of child process creation with fork(), environment variables are copied into child's memory (as, child's memory is duplicate of parent's memory).
   4. In case of child process creation with execve() system call, environment variables have to passed on through the third argument to the execve() call.

6. The followings are two different ways to print out environment variables. Describe their differences:
   $ /usr/bin/env
   $ /usr/bin/strings /proc/$$/environ
   Key points:
   a. The first command: env command is a shell process. A child process is created to run env, and we are actually printing the environment variables of that child process.
   b. The second command: Is not a child process, the strings command is used to print out the contents of a file that has environment variables of the current shell process.
   c. Although in most cases these would be similar (due to shell passing its variables to the child process), they are not the same.

7. In Linux, many environment variables are ignored by the dynamic linker if the program to be executed is a Set-UID program. Two such examples are LD PRELOAD and LD LIBRARY PATH. Read the manual of ld-linux (https://linux.die.net/man/8/ld-linux) and explain why the following environment variables are also ignored:

   • LD AUDIT

   • LD DEBUG OUTPUT

   Key points:
   a. LD_AUDIT :
      1. List of Executable and Linkable Format (ELF) files, user-specified, loaded before all others in a separate linker namespace. (similar to LD_PRELOAD).
      2. Untrusted code in this list can be linked first and can create problems. If used in combination with SET-UID program, problems can be severe.
   b. LD_DEBUG_OUTPUT:
      1. Location to print debug output. Default is standard output – 1.
      2. Dynamic linker's debugging info can be saved to a file. If used in combination with SET-UID program, a protected file can be the location to which debug info is written to. Can also lead to privilege escalation.

Section 2

15 Points each

1. 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 (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. Compile and run the following program, and describe your observation.

The program "myprintenv.c", can be found in the "Assignment2_Files.zip"; 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".

```
#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();          ①
      exit(0);
    default:  /* parent process */
      //printenv();        ②
      exit(0);
  }
}
```

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.

Step 3. Compare the difference of these two files using the *diff* command. Describe your conclusion

Key points:

a. Create two versions of the program based on the descriptions.

b. Store the outputs in different files (eg. op1 and op2)
c. Use “diff op1 op2”, to print out the differences. There are none.
d. Screenshots and Description of how there are no differences due to the child getting a copy of the variables from parent.

2. 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. Compile and run the following program, and describe your observation.

The program “myenv.c” can be found in the “Assignment2_Files.zip”; it can be compiled using "gcc myenv.c", This program simply executes a program called /usr/bin/env, which prints out the environment variables of the current process.

```c
#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);

  return 0 ;
}
```

Step 2. Change the invocation of execve() in Line ① to the following and describe your observation.
execve("/usr/bin/env", argv, environ);

Step 3. Describe your conclusion regarding how the new program gets its environment variable.

Key points:

a. If using NULL, as the third argument in execve call, no environment variables are passed to the env process.

b. If using environ, as the third argument in execve call, all environment variables from current process are passed to the env process.
c. Snapshots and conclusions.