SET-UID Program Exercise

1. Overview

The learning objective of this lab is for students to understand how environment variables affect program and system behaviors, and how attackers might take advantage of a `Set-UID` program. 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, and how they affect the behaviors of `Set-UID` programs.

Lab environment: Use the pre-built Ubuntu VM that has been provided for this class.

Submission: You need to submit a detailed lab report to describe what you have done and what you have observed. Follow the tasks and for each task answer the **Q#** specifically in your report. You may provide explanation to the observations that are interesting or surprising. You can always add code snippets or screenshots of what you have observed. You are encouraged to pursue further investigation, beyond what is required by the lab description. You can earn bonus points for extra efforts (at the discretion of your instructor). Only submit typed reports electronically. No handwritten reports accepted.

2. Tasks

Task 1: Using System() function

In this task we study how the system() function works. This function is used to execute a command from a C program, but unlike `execve()`, which directly executes a command, `system()` actually executes `"execl("/bin/sh", "sh", "-c", command, (char *) NULL);"`, meaning it executes a shell `/bin/sh` first and asks the shell to execute the command.

execl() function searches for an executable file if the specified filename does not contain a slash (/) character. The file is sought in the colon-separated list of directory pathnames specified in the `PATH` environment variable.
If the specified filename includes a slash character, then `PATH` is ignored, and the file at the specified pathname is executed.

Write a program called systemtest.c:

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

Compile it: gcc systemtest.c -o systemtest
Run the program to see how it works.

Now we are going to write our own “ls” program to see how the environmental variable `PATH` works.
Write a program called ls.c:

```
#include <stdio.h>
#include <unistd.h>

int main()
{
        printf("This is my ls program\n");
        return 0;
}
```

Compile it and run it.
Q1: If you run the systemtest program, what do you see as the result? How can we ensure that it runs our *ls* program

instead of `/bin/ls`? Describe and explain your observations.

You can change the `PATH` environment variable in the following way:
`PATH=.:$PATH`

The "." represents your current directory. So this new directory has been added to the beginning of the `PATH` variable, so the first location the system looks for is your current directory. You can view the `PATH` variable by typing: `echo $PATH`

Task 2: `Set-UID` programs

Now we want to make our `systemtest` program a `Set-UID` program. Change its ownership to *root*, and make it a Set-UID program:

```
$ sudo chown root systemtest
$ sudo chmod 4755 sysmtetest
```

Now any other user (e.g. you) who runs this program, will run it with root privileges. So the "ls" program that is running by the system() function is running with root privileges. How can you check this? Try changing our "ls" program so that it can print out the *real user id* and *effective user id*. You can use the functions `getuid()` and `geteuid()` respectively for this.

Q2: What do you expect to see as the result? What does actually happen? What is the real and effective user id of our "ls" program? Describe and explain your observations.

Note: The `system(cmd)` function executes the `/bin/sh` program first, and then asks this shell program to run the *cmd* command. In Ubuntu 16.04 VMs, `/bin/sh` is actually a symbolic link pointing to another shell. Depending on which shell it is linked to, the shell might have a countermeasure that prevents itself from being executed in a `Set-UID` process. If it 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 *systemtest* program is a `Set-UID` program, the countermeasure 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 use the following commands to link `/bin/sh` to `zsh`:

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

Task 3: Real attack

So far we wrote a "ls" program to run instead of the `/bin/ls` program that was the intent of the victim to run. Our "ls" program is not doing anything except printing a few lines. As an attacker, what should your "ls" program be? If your "ls" program is a shell itself and you (as an attacker) manage to execute it instead of `/bin/ls` with root privilege, you can run any other program or command from this shell. Try to copy a shell and name it "ls". Then try to run this shell from the systemtest program with root privilege. If you are successful you should get a new shell and if you have root privilege your shell will have a "#" sign in the beginning.

Q3: Describe the steps you took to run a shell from systemtest with root privileges. Explain your observations.

Task 4: Capability Leaking

To comply with the Principle of Least Privilege, `Set-UID` programs often permanently relinquish their root privileges if such privileges are not needed anymore. Also in cases when the program needs to hand over its control to the user, 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 calling process is privileged, 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.

Compile the following program, change its owner to root, and make it a `Set-UID` program. Run the program as a normal user. Before running the program you need to create the /etc/cap file as root (root owned file). Then change the permissions of the file to 644 (Read-only by others).

Q4: Describe what you have observed. Will the file /etc/cap be modified? Explain what was the expected behavior and what actually happened and why?

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

void main() {
     int fd;
     /* Assume that /etc/cap is an important system file,
        and it is owned by root with permission 0644.
        Before running this program, you should create
        the file /etc/cap first. */
    fd = open("/etc/cap", O_RDWR | O_APPEND);
    if (fd == -1) {
        printf("Cannot open /etc/cap\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);
    }
}
```