PRO.M.IS
security built in

Lab 2: Set-user-ID

Purpose

The learning objective of this lab is for students to understand how the Set-user-ID (SUID) permission bit affects the execution of programs and the security risks SUID programs may pose. Processes in Linux (UNIX) are associated with three type of identities: *real user ID*, *effective user ID* and *saved user ID*. The *real user ID* denotes the user that started the process (i.e. the process owner). The kernel uses the *effective user ID* to control access to privileged resources.

A program can manipulate the *real* and *effective user IDs*, while the *saved user ID* is managed by the kernel. If a user is the owner of a SUID executable, then the *effective user ID* for the running process will be the same as the real user ID. When the user is not the owner, the kernel will set the *effective user ID* to represent the file owner. Frequently, the owner is a privileged user, typically user *root*, and therefore the program will have privileged access to resources. The program can use system calls to drop its privileges, essentially changing the *effective user ID* to be the same as the *real user ID*. The program can keep toggling between privileged and restricted access as needed.

In addition, a program has also a *real group ID*, an *effective group ID* and a *saved group ID*. Their functionality is similar to that of the user IDs: they are used to control privileged access based on group membership. However, the focus of this is lab is on user IDs.

The tasks below follow the examples presented in the lecture.

Task 1:

Copy the file */bin/cat* to the home directory on the VM, change the file owner to *root* and try to read root-owned files. For example, you can try to read the file */etc/shadow*.

Task 2:

Make */bin/cat* root-SUID and try to read root-owned files

Task 3:

In this task, we study a vulnerability that exists when programs are 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.

Write a C program that uses the *system()* function call to execute */bin/id*, captures the output and prints it to the standard out (terminal). The program takes as input a user ID.

Task 4:

Make the executable from Task 3 SUID-root. Run it by passing a malicious string to it to make it start a shell. Use the id command to determine if this is a r*oot* shell.

Task 5:

Circumvent dash defense mechanism by soft-linking to */bin/zsh* as shown below. Did you get a *root* sheel now?

```
sudo ln -sf /bin/zsh /bin/sh
```

Task 6:

Write a C program that uses the *execve()* function call to execute */bin/id*, captures the output and prints it to the standard out (terminal). 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.

Task 7:

Make the program from Task 6 SUID-root. Verify that passing a malicious string as in Task 4 does not work anymore.

Task 8:

Compile the cap_leak.c program shown below, but don't make it SUID-root just yet. The program leaks a file descriptor into the shell process it starts. This is a vulnerability that could be exploited to obtain illegitimate access to data.

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

int main(int argc, char* argv[])
{
    int fd;
    char *v[2];

    fd = open("/etc/zzz", O_RDWR | O_APPEND);
    if (fd == -1) {
        printf("Cannot open /etc/zzz\n");
        exit(0);
    }

    printf("fd is %d\n", fd);

    setuid(getuid());
    v[0] = "/bin/sh"; v[1] = NULL;
    execve(v[0], v, 0);
}
```

Create the file */etc/zzz* using the command shown below.

```
sudo sh -c 'echo "Hello world" > /etc/zzz'
```

Verify if you can append text to the *zzz* file by redirecting data to the leaked file descriptor as shown below. You should replace 3 with the file descriptor shown by your program. What was the result?

```
$ echo "Cruel world" >& 3
```

Task 9:

Make cap_leak SUID-root. Verify if it possible to append text to the *zzz* file by redirecting to the leaked file descriptor. What was the result?

Task 10:

Fix the cap_leak program so that privileged capabilities are released before privileged access is dropped. Verify if it still possible to access data as before. What was the fix?

Task 11:

Restore the default shell in your VM as shown below.

```
sudo ln -sf /bin/dash /bin/sh
```

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).