

Mini Project 3: Set-UID Program Vulnerability

(due April 13, end of the day)

Set-UID is an important security mechanism in Unix operating systems. When a Set-UID program is run, it assumes the owner's privileges. For example, if the program's owner is root, 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 unfortunately, it is also the culprit of many bad things.

In this project, the main goal is to: (1) understand why Set-UID is needed and how it is implemented and (2) be aware of its bad side: understand its potential security problems.

Settings

Use the SEED Virtual Machine to complete all tasks in this mini project. A few sample programs are provided. Please download MP3.tar.gz from Canvas.

Task 1: Explore SetUID Programs

Explore a few Set-UID programs: `passwd`, `chsh`, and `sudo`. Run these programs in their default location (`/bin` or `/usr/bin` directories) and then in the directory of your choice (e.g., Desktop or Downloads).

Question 1. Did the programs work appropriately in both cases? Please briefly justify your observations.

Task 2: Exploring Environment Variables

2.1 Manipulating Environment Variables

We can view the environment variables using commands `printenv` or `env`. For example, to view the path of the working directory (`PWD`), we can use commands such as `printenv PWD` or `env | grep PWD`. We can also set or unset environment variables using commands `export` and `unset`, respectively.

Question 2. Please set an environment variable called "foo" with a value of your choice, show its value, and unset it. Show your results with screenshots.

2.2 Passing Environment Variables from Parent Process to Child Process

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. You can view the manual of `fork()` using the 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.

Please download the sample programs and view the program `myprintenv.c`. You will find the parent process (currently commented out) and the child process in the `main()` function.

1. Please compile and run the program `myprintenv.c`. It will generate a binary called `a.out`. Compile and save the output into a file, for example, `a.out > testfile`.

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 program again. Save the output to another file.

Question 3. Compare the difference of the two output files using the `diff` command. Please describe your observations.

2.3 Environment Variables and `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 those of the program loaded. Essentially, `execve()` runs the new program inside the calling process.

Please find the `myenv.c` program. It executes the program `"/usr/bin/env"`, which prints out the environment variables of the current process.

1. Compile and run the program `myenv.c`. What do you observe?
2. Change the invocation of `execve()` in `myenv.c` to `execve("/usr/bin/env", argv, environ);`. Compile and run the program. What do you observe?

Question 4. How does the new program get its environment variables? Please explain based on your observations.

2.4 Environment Variables and `system()`

Environment variables can be affected when a new program is executed via the `system()` function. Unlike `execve()`, `system()` actually executes `"/bin/sh -c command"`. In other words, it executes `/bin/sh` and asks the shell to execute the command. Check its manual using `man system`. In particular, the `system()` function uses `execl()` to execute `/bin/sh`. To do so, `execl()` calls `execve()` and pass the environment variables array to it.

Please compile and run the program `mssystem.c`. Take a screenshot of the result.

Question 5. How does the new program `/bin/sh` get its environment variables? Please explain based on your observations.

Task 3: Environment Variables and Set-UID Programs

When a Set-UID program runs, it assumes the owner's privileges. Therefore, Set-UID programs could result in privilege escalation. It is quite risky despite being useful in many tasks.

3.1 Use Environment Variables to Affect Set-UID Programs

The behaviors of Set-UID programs are decided by their program logic. However, users can affect their behaviors via environment variables. The sample program `printall.c` prints out all the environment variables in the current process.

1. Compile `printall.c`. Then, change its ownership to root and make it a Set-UID program. (Hint: use `chown root`, `chmod 4755` commands).
2. In your shell (you need to be in a normal user account but not the root account), use the `export` command to set the following environment variables (they may already exist):

- PATH
- LD_LIBRARY_PATH
- ANY NAME (hint: this is an environment variable defined by you)

3. Now, run the Set-UID program from Step 1 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.

Question 6. 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.

3.2 The PATH Environment Variable

Calling `system("cmd")` within a Set-UID program is dangerous, 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. The `system("cmd")` function first executes the `/bin/sh` program and then asks this shell program to run the `cmd` command. In Ubuntu 20.04, `/bin/sh` is a symbolic link pointing to `/bin/bash`.

In this task, we provide a sample program `mys.c`, which executes the `/bin/ls` command. However, it uses the relative path for the `ls` command, rather than the absolute path.

1. Copy `mys.c` to the `/home/seed/` directory. Then, add the directory `/home/seed` to the beginning of the `PATH` environment variable. You can do this in `bash` using the command: `export PATH=/home/seed:$PATH`.
2. Compile `mys.c` to get an executable file (name the output `mys`). Change its owner to `root`, make it a Set-UID program, and then run `mys`. What do you see? Please take a screenshot of your result.
3. Now, let's modify the `system()` command in `mys.c` to execute some *malicious* codes. Then, repeat step 2 to compile the program, change its owner to `root`, and make it a Set-UID file.

For example, you can use `system("cat /etc/shadow")` to view the "shadow" file, which requires the root privilege. You can also add another command `system("whoami")` in `mys.c`, which prints the name of the user executing the code (i.e., effective user ID).

Question 7. In step 3, can you get the Set-UID program to run a malicious command (such as `system("cat /etc/shadow")` or other commands of your choice)? Please report your observations (with screenshots). Are the programs running with the root privilege?

Note: When we call `system()`, `/bin/bash` detects if it is executed in a Set-UID process. If so, it immediately changes the effective user ID to the process's real user ID, essentially dropping the user's privilege. So, in step 3, the permission to display the shadow file will be denied, because the effective user ID is changed to `seed` when running the program, even though it is a Set-UID program. This countermeasure of `/bin/bash` can address the Set-UID vulnerability and prevent the attack (as the one in step 3).

4. Next, let's explore the Set-UID vulnerability in other shells that do not have such a countermeasure. To do so, we will link `/bin/sh` to another shell, the `zsh` shell, using the command `sudo ln -sf /bin/zsh /bin/sh`. Run the program that you created in step 3. What do you see? Please take a screenshot of your result.

Note: You may need to open a new terminal window to make `zsh` active.

Submission

Submit a project report to describe what you have done and what you have observed in each task. Please explain the observations that are interesting or surprising. Please include screenshots and code snippets of important findings. Simply attaching code or screenshots without any explanation will not receive credits.