CS 656 LAB 2
Linux Access Control and Set-UID Program Vulnerability

Notes:

- This is an individual lab.
- The code and other answers you submit MUST be entirely your own work, and you are bound by the WSU Academic Integrity Policy (https://www.wichita.edu/about/student_conduct/ai.php). You MAY consult with other students about the conceptualization of the tasks and the meaning of the questions, but you MUST NOT look at any part of someone else's solution or collaborate with anyone. You may consult published references, provided that you appropriately cite them in your reports and programs, as you would do in an academic paper.
- Read the entire document carefully before you start working on the lab.

GOOD LUCK!

1 Overview

The learning objective of this lab is to understand Linux access control mechanisms and how environment variables affect the behavior of privileged `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 (i.e., not by users), users can indeed affect the behaviors via environment variables. This lab covers the following topics:

- Linux access control lists
- Environment variables and `Set-UID` programs
- Securely invoke external programs
- Capability leaking

Lab Environment. This lab has been tested on Ubuntu 20.04. You will be given access to the remote Ubuntu environment with necessary packages installed. You may also run your own virtual machine by downloading it from the SEED labs website (see the details here: https://github.com/seed-labs/seed-labs/blob/master/manuals/vm/seedvm-manual.md).

2 Submission

Submit a PDF document with your answers to the questions in this lab. Your report should have a subheading for each question, and your answers should be inside the corresponding subheading. If applicable, list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.

Note: *Your report must contain your Name (Lastname, Firstname) and WSU ID. Use the following format for your report's PDF filename:* `lab2report_YOUR_WSU_ID.pdf`*. There is a 10% reduction of points if your report does not follow the correct filename format and/or missing name/ID inside the document.*

3 Lab Tasks

3.1 Task 1: Linux Access Control Warm-up

The purpose of this task is to familiarize students with access control lists (ACLs) in the Linux OS.

Deliverables. Answer the following questions in your report. Clearly mark task and question numbers.

Task 1.1: Examining File Permissions with the `chmod` command **[8 Points]**

1. Create a test user called wushock. Type `sudo adduser wushock` and enter a password and WuShock's info (you can provide fake info for WuShock's profile). Make sure you can remember the password.
2. Your original user is referred to as `user1` in this document.
3. As user1, create a new group called `scratch_group` by typing `sudo groupadd scratch_group`.
4. Add both user1 and wushock to the new group, i.e., `sudo usermod -a -G scratch_group user1` and `sudo usermod -a -G scratch_group wushock`.
5. As user1, switch to the new group by typing `newgrp scratch_group`
6. Create a new directory inside your home directory called `scratch` by typing `mkdir /home/cs656/scratch`. Move into the new directory: `cd scratch`
7. Using your original user `user1`, create a short text file in the `scratch` directory and call it your name using the echo command and a stream output redirect (>) by typing `echo "Play Angry!" > user1_file`.
8. As user1, set the permissions `644 (-rw-r--r--)` on the file using octal syntax by typing `chmod 0644 user1_file`.

   Q 1.1
   Search for Linux file permissions online.
   What permission setting are you specifying with 0644?

9. Type `ls -l user1_file` and verify that the permission string in the first column of your file matches the value `(-rw-r--r--)`.
10. Now, open a shell as wushock. Type `sudo -u wushock bash` and enter your root password. You are now logged in as wushock.
11. As wushock, try to access (read) the file by typing `cat user1_file`,

12. As wushock, try to append `(>>)` a line to the file by typing `echo "Go Shockers!" >> user1_file`.

13. As wushock, try to change the name of the file by typing `mv user1_file wushock_file`.

Q 1.2
Did the name of the file changed?
If so, why where you successful in doing so this time?

14. Note: if you couldn't change the file name, again switch the user `wushock` to the `scratch_group` by typing `newgrp scratch_group`.

15. As wushock, try to change the mode (permissions) of the file using octal syntax by typing `chmod 600 wushock_file`.

Q 1.3
Why do you see an operation not permitted error?

16. As wushock, try to delete the file by typing `rm wushock_file`. (The system may or may not ask for verification, but it should permit the deletion.) This is true despite the fact that wushock doesn't own the file because wushock can write to the directory in which the file resides.

17. As user1 (type `exit` to jump out of user wushock's shell), repeat step 7 to re-create the test file (remember you can scroll through your recent command history with the up and down arrow keys). Make sure you are in the `scratch_group` by running step 5.

18. As user1, give the file more restrictive permissions using symbolic syntax by typing `chmod u-w,g-w,o=w user1_file`.

19. Typing `ls -l user1_file` should reveal permissions of `(-r--r---w-)`.

Q 1.4
Whose permissions did you change using the above chmod command?

20. Remove wushock from the `scratch_group` by typing `sudo deluser wushock scratch_group`

21. As wushock, repeat steps 10-15 (but do not delete the file at the next step). Note you may need to exit wushock's terminal and then log in as wushock again.

Q 1.5
Why does running cat on the file produce a permission denied error, but the echo/append command does not?

Q 1.6
Are you still able to change the name but not the mode (permissions) of the file?

22. As `user_1`, change the directory permissions to allow wushock to rename files: `chmod o+w .` (notice the period at the end)

23. As `wushock`, rename the file as `wushock_file`
24. As user1, try to access (read) the file by typing `cat wushock_file`.

Q 1.7
What do you see displayed?

25. As user1, try to overwrite the file with no text ("nulling" the file) by typing `> wushock_file`.

Q 1.8
Why do you see a permission denied error?

26. As either user1 or wushock delete the file by typing `rm wushock_file`.

3.1.1 Task 1.2: Examining Default File Permissions with the umask command [2 Points]

1. `umask` sets the default permissions for newly created files and directories, but works by removing the permissions set. For example a umask value of 022 subtracts 022 from the base value of 666 for files or 777 for directories, thus removing the write permissions of group and other (world) for newly created files and directories.
2. Only use one of the logged in users for this exercise.
3. Change into your own home directory by typing `cd ~`.
4. Check your current default file permissions by typing `umask`.

Q 1.9
What umask setting is returned and how do you calculate the default permissions of a new file?

5. Create a new file by typing `touch test_file1`.
6. Type `ls -l` and verify that the permission string in the first column of your file matches the value `(-rw-r--r--)`
7. Create new default file permissions `(-rw-------)` by typing `umask 0077`
8. Create a second new file by typing `touch test_file2`.
9. Type `ls -l` and verify that the permission string in the first column of your new file matches the value `(-rw-------)`.
10. Create new default file permissions `(-rw-rw-rw-)` by typing `umask 0000`
11. Create a third new file by typing `touch test_file3`.
12. Type `ls -l` and verify that the permission string in the first column of your new file matches the value `(-rw-rw-rw-)`.

Q 1.10
What do each of these `umask` settings mean for users attempting to access any new files or directories you create?

13. Reset your `umask` setting to the more secure default file permissions of `(-rw-r-----)` by typing `umask 0027`.
14. Check your current default file permissions to confirm the change by typing `umask`
15. Delete the test files by typing `rm test_file*`.

3.2 Task 2: Environment Variable and Set-UID Programs [18 Points]

`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, 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 since it escalates the user's privilege, it is 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. Write the following program that can print out all the environment variables in the current process. Call the resulting binary `task2`.

Listing 1: `task2.c`

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

extern char **environ;
int main()
{
   int i = 0;
   while (environ[i] != NULL) {
      printf("%s\n", environ[i]);
      i++;
   }
}
```

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

```
$ sudo chown root task2
$ sudo chmod 4755 task2
```

Step 3. In your 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).

These environment variables are set in the user's shell process. For example:

```
$ export PATH=/some/path
```

Note: If you want to "append" to the environment variable, use the following command instead: `export PATH=/some/path:$PATH`

Step 4. 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.

Step 5. Check whether all the environment variables you set in the shell process (parent) get into the `Set-UID` child process. You can check this by typing `task2| grep NameOfYourVariable`. The `grep` command will filter out any lines that do not match the `NameOfYourVariable` string.

Deliverables. Include screenshots as instructed and answer the following questions in your report. Clearly mark task and question numbers.

2.1. A screenshot of the output of the command in Step 5. This screenshot should support your next answer.

2.2. Did all three of the variables that you set in Step 3 passed to the `Set-UID` child process? If yes, explain why in terms of how environment variables are passed between the shell its child process. If not, explain why not.

3.3 Task 3: The `PATH` Environment Variable and `Set-UID` Programs [18 Points]

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.

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:

Listing 2: `task3.c`

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

Step 1. Compile the above program, change its owner to `root`, and make it a `Set-UID` program. Call your program `task3`.

Step 2. Note that, the `system(cmd)` function executes the `/bin/sh` program first, and then asks this shell program to run the `cmd` command. In Ubuntu 20.04 (and several versions before), `/bin/sh` is actually a symbolic link pointing to `/bin/dash`. This shell program has a countermeasure that prevents itself from being executed in a `Set-UID` process. Basically, if `dash` 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 victim program is a `Set-UID` program, the countermeasure in `/bin/dash` 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 have installed a shell program called `zsh` in our Ubuntu 20.04 VM. We use the following commands to link `/bin/sh` to `/bin/zsh`:

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

Step 3. Create a test file in the directory `/root`. You will need to use `sudo` because `/root` is owned by the root user. That is, `sudo touch testfile`.

Step 4. Run `./task3 /root`.

Deliverables. Include screenshots as instructed and answer the following questions in your report. Clearly mark task and question numbers.

3.1. A screen shot of the output of the command in Step 3 showing the `task3` successfully displays the test file created in step 4. You should not use the `sudo` command to run `task3`. The screenshot should support the answer to the next questions.

3.2. Is `task3` running with the root privilege? If yes, explain why in terms of the properties of `Set-UID` programs. If not, explain why not.

3.4 Task 4: The `LD_PRELOAD` Environment Variable and `Set-UID` Programs [18 Points]

In this task, we 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 run time.

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 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, we will only study `LD_PRELOAD`. We will see how these environment variables influence the behavior of dynamic loader/linker when running a normal program. Follow these steps:

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

   Listing 3: `mylib.c`

   ```
   #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. We can compile the above program using the following commands (in the `-lc` argument, the second character is ℓ):

   ```
       $ gcc -fPIC -g -c mylib.c
       $ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
   ```

   The `-fPIC` option stands for "position independent code", which allows the OS to move it to different locations in RAM. This is often the case with libraries where there may already by a library of the same name loaded to memory.

3. Now, set the LD_PRELOAD environment variable:

```
      $ export LD_PRELOAD=./libmylib.so.1.0.1
```

4. Finally, compile the following program task4, and in the same directory as the above dynamic link library libmylib.so.1.0.1:

Listing 4: task4.c

```
      #include <unistd.h>
      int main()
      {
         sleep(1);
         return 0;
      }
```

Deliverables. Include screenshots as instructed and answer the following questions in your report. Clearly mark task and question numbers.

4.1. Make task4 a regular program, and run it as a normal user, e.g., the cs656 user.

   (a) Include a screenshot of the output of task4. The screenshot should support your answer to the next question.

   (b) Did myprog use your malicious library or the standard libc library? Explain why.

4.2. Make task4 a Set-UID owned by root and run it as a normal user, e.g., the cs656 user.

   (a) Include a screenshot of the output of task4. The screenshot should support your answer to the next question.

   (b) Did task4 use your malicious library or the standard libc library? Explain why.

4.3. Make task4 a Set-UID program owned by root. Export the LD_PRELOAD environment variable. But this time export in the root account. Run task4 as the cs656 user.

   (a) Include a screenshot of the output of task4. The screenshot should support your answer to the next question.

   (b) Did task4 use your malicious library or the standard libc library? Explain why.

4.4. Make task4 a Set-UID program owned by a non-root user different from the cs656 user, say user1. Export the LD_PRELOAD environment variable again. But this time in the user1 account. Run task4 as the cs656 user.

   (a) Include a screenshot of the output of task4. The screenshot should support your answer to the next question.

   (b) Did task4 use your malicious library or the standard libc library? Explain why.

3.5 Task 5: Invoking External Programs Using `system()` versus `execve()` [18 Points]

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 affect 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 shell has another dangerous consequence, and this time, it has nothing to do with environment variables. Let us 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.

Listing 5: `task5.c`

```
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 followings.
   system(command);
   // execve(v[0], v, NULL);

   return 0 ;
}
```

Step 1. Compile the above program. The program will use `system()` to invoke the command.

Step 2. Make the program a root-owned `Set-UID` program.

Deliverables. Include screenshots as instructed and answer the following questions in your report. Clearly mark task and question numbers.

5.1. Show how you could compromise the integrity of the system as a normal user (e.g., the `cs656` user) by writing or deleting a file owned by root using the program written by Vince. <u>**Note:**</u> Do not delete or modify a file in a way that will cause your VM to malfunction. For example, do not delete the `/etc/shadow` file.

(a) Provide a screenshot of your command that exploits Vince's program to launch the attack. The screenshot should also include the output.

5.2. 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 a root-owned `Set-UID`.

(a) Does your attack still work? Explain why or why not.

3.6 Task 6: Capability Leaking [18 Points]

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 privilege 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.

Step 1. Compile the following program, change its owner to root, and make it a `Set-UID` program. Call the resulting binary `task6`. Run the program as a normal user.

Listing 6: `task6.c`

```
void main()
{
   int fd;
   char *v[2];

   /* 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);
   }

   // Print out the file descriptor value
   printf("fd is %d\n", fd);

   // Permanently disable the privilege by making the
   // effective uid the same as the real uid
```

```
      setuid(getuid());

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

Step 2. Create the file `/etc/zzz`.

Deliverables. Include screenshots as instructed and answer the following questions in your report. Clearly mark task and question numbers.

6.1. Run the program as a normal user.

   (a) A screenshot of the output of `task6` and a screenshot of `cat /etc/zzz`. Your outputs should support your nswer to the next question.

   (b) Will the file `/etc/zzz` be modified? Explain why or why not.