

---

## Seed Labs 2.0

### Environment Variables and Set-UID Lab

---

SeedLab Link:

[https://seedsecuritylabs.org/Labs\\_20.04/Software/Environment\\_Variable\\_and\\_SetUID/](https://seedsecuritylabs.org/Labs_20.04/Software/Environment_Variable_and_SetUID/)

#### 1. Manipulating Environment Variables

In this task, we study the commands that can be used to set and unset environment variables. We are using Bash in the seed account. The default shell that a user uses is set in the `/etc/passwd` file (the last field of each entry). You can change this to another shell program using the command `chsh` (please do not do it for this lab). Please do the following tasks:

- Use **printenv** or **env** command to print out the environment variables. If you are interested in some particular environment variables, such as `PWD`, you can use `"printenv PWD"` or `"env | grep PWD"`.
- Use **export** and **unset** to set or unset environment variables. It should be noted that these two commands are not separate programs; they are two of the Bash's internal commands (you will not be able to find them outside of Bash).

**env**

```
cys24004-bharath@VM:~/Desktop/ex_8$ env
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/2443,unix/VM:/tmp/.ICE-unix/2443
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1001/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
SSH_AGENT_PID=2379
GTK_MODULES=gail:atk-bridge
DBUS_STARTER_BUS_TYPE=session
PWD=/home/cys24004/Desktop/ex_8
LOGNAME=cys24004-bharath
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1001/gnupg/S.gpg-agent:0:1
XAUTHORITY=/run/user/1001/gdm/Xauthority
WINDOWPATH=2
HOME=/home/cys24004
USERNAME=cys24004-bharath
IM_CONFIG_PHASE=1
LANG=en_US.UTF-8
```

```
cys24004-bharath@VM:~/Desktop/ex_8$ env | grep PWD
PWD=/home/cys24004/Desktop/ex_8
OLDPWD=/home/cys24004/Desktop
```

## export and unset

```
cys24004-bharath@VM:~/Desktop/ex_8$ EXPORT my_var="hello"
EXPORT: command not found
cys24004-bharath@VM:~/Desktop/ex_8$ export my_var="hello"
cys24004-bharath@VM:~/Desktop/ex_8$ echo $my_var
hello
cys24004-bharath@VM:~/Desktop/ex_8$ unset my_var
cys24004-bharath@VM:~/Desktop/ex_8$ echo $my_var

cys24004-bharath@VM:~/Desktop/ex_8$
```

## 2. Passing Environment Variables from Parent Process to Child Process

In this task, we study how a child process gets its environment variables from its parent. 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 (please see the manual of `fork()` by typing the following 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.

Step 1. Please compile and run the following program, and describe your observation. The program can be found in the Labsetup folder; it can be compiled using "`gcc myprintenv.c`", which will generate a binary called `a.out`. Let's run it and save the output into a file using "`a.out > file`".

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
extern char **environ;
void printenv()
{
    int i = 0;
    while (environ[i] != NULL) {
        printf("%s\n", environ[i]);
        i++;
    }
}
void main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
```

```
//printenv();  
exit(0);  
}  
}
```

```
cys24004-bharath@VM:~/Desktop/ex_8$ gcc myprintenv.c -o myprintenv  
cys24004-bharath@VM:~/Desktop/ex_8$ ./myprintenv > file1.txt  
cys24004-bharath@VM:~/Desktop/ex_8$ cat file1.txt  
SHELL=/bin/bash  
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/2443,unix/VM:/tmp/.ICE-unix/2443  
QT_ACCESSIBILITY=1  
COLORTERM=truecolor  
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg  
XDG_MENU_PREFIX=gnome-  
GNOME_DESKTOP_SESSION_ID=this-is-deprecated  
GNOME_SHELL_SESSION_MODE=ubuntu  
SSH_AUTH_SOCK=/run/user/1001/keyring/ssh  
XMODIFIERS=@im=ibus
```

Step 2. Now comment out the `printenv()` statement in the child process case (Line A), and uncomment the `printenv()` statement in the parent process case (Line B). Compile and run the code again, and describe your observation. Save the output in another file.

```
cys24004-bharath@VM:~/Desktop/ex_8$ gcc myprintenv.c -o myprintenv  
cys24004-bharath@VM:~/Desktop/ex_8$ ./myprintenv > file2.txt  
cys24004-bharath@VM:~/Desktop/ex_8$ cat file2.txt  
SHELL=/bin/bash  
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/2443,unix/VM:/tmp/.ICE-unix/2443  
QT_ACCESSIBILITY=1  
COLORTERM=truecolor  
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg  
XDG_MENU_PREFIX=gnome-  
GNOME_DESKTOP_SESSION_ID=this-is-deprecated  
GNOME_SHELL_SESSION_MODE=ubuntu  
SSH_AUTH_SOCK=/run/user/1001/keyring/ssh  
XMODIFIERS=@im=ibus  
DESKTOP_SESSION=ubuntu  
SSH_AGENT_PID=2379  
GTK_MODULES=gail:atk-bridge
```

Step 3. Compare the difference of these two files using the `diff` command. Please draw your conclusion.

```
cys24004-bharath@VM:~/Desktop/ex_8$ diff file1.txt file2.txt  
cys24004-bharath@VM:~/Desktop/ex_8$
```

**Observation :** No difference between file1 and file2 . The child process inherits the environment variables from its parent, so there should be no differences between the two files.

### 3. Environment Variables and `execve()`

In this task, we study how environment variables are affected when a new program is executed via `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 that of the program loaded. Essentially, `execve()` runs the new program inside the calling process. We are interested in what happens to the environment variables; are they automatically inherited by the new program?

Step 1. Please compile and run the following program, and describe your observation. This program simply executes a program called `/usr/bin/env`, which prints out the environment variables of the current process.

```
//myenv.c

#include <unistd.h>
extern char **environ;
int main()
{
    char *argv[2];
    argv[0] = "/usr/bin/env";
    argv[1] = NULL;
    execve("/usr/bin/env", argv, NULL);
    return 0;
}
```

```
cys24004-bharath@VM:~/Desktop/ex_8$ gcc myenv.c -o myenv
cys24004-bharath@VM:~/Desktop/ex_8$ ./myenv
```

Step 2. Change the invocation of `execve()` in Line 8 to the following; describe your observation.

```
execve("/usr/bin/env", argv, environ);
```

```
cys24004-bharath@VM:~/Desktop/ex_8$ gcc myenv.c -o myenv
cys24004-bharath@VM:~/Desktop/ex_8$ ./myenv
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/2443,unix/VM:/tmp/.ICE-unix/2443
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1001/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
SSH_AGENT_PID=2379
GTK_MODULES=gail:atk-bridge
DBUS_STARTER_BUS_TYPE=session
PWD=/home/cys24004/Desktop/ex_8
LOGNAME=cys24004-bharath
XDG_SESSION_DESKTOP=ubuntu
```

Step 3. Please draw your conclusion regarding how the new program gets its environment variables.

When using the `execve()` function, environment variables behave differently depending on how they are handled. If the environment variables are the same in both runs, it confirms that `execve()` automatically inherits the calling process's environment by default. Conversely, if there are differences, it shows that `execve()` does not pass environment variables unless they are explicitly provided. Thus, the expected conclusion is that `execve()` automatically inherits environment variables only when they are explicitly included in its third argument.

## 4.Environment Variables and system()

In this task, we study how environment variables are affected when a new program is 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. If you look at the implementation of the `system()` function, you will see that it uses `execl()` to execute `/bin/sh`; `execl()` calls `execve()`, passing to it the environment variables array. Therefore, using `system()`, the environment variables of the calling process is passed to the new program `/bin/sh`. Please compile and run the following program to verify this.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
system("/usr/bin/env");
return 0 ;
}
```

```
cys24004-bharath@VM:~/Desktop/ex_8$ gcc mysystem.c -o mysystem
cys24004-bharath@VM:~/Desktop/ex_8$ ./mysystem
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/2443,unix/VM:/tmp/.ICE-unix/2443
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1001/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
SSH_AGENT_PID=2379
```

This will run the `/usr/bin/env` through the `system()` function, which displays the current environment variables. The `system()` function utilizes `/bin/sh` to execute the specified command, ensuring that the environment variables from the calling process (our C program) are transferred to `/bin/sh`. Consequently, `/bin/sh` forwards these variables to `/usr/bin/env`. By examining the output, we can confirm that the environment variables from the original process are successfully passed to the program launched with `system()`.

## 5. Environment Variables and 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, 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.

```
#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.

```
cys24004-bharath@VM:~/Desktop/ex_8$ gcc printenv.c -o foo
cys24004-bharath@VM:~/Desktop/ex_8$ sudo chown root foo
cys24004-bharath@VM:~/Desktop/ex_8$ sudo chmod 4755 foo
```

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. Now, 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. 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.

```
cys24004-bharath@VM:~/Desktop/ex_8$ export PATH=$PATH:/home/cys2400-bharath/Desktop
cys24004-bharath@VM:~/Desktop/ex_8$ export LD_LIBRARY_PATH=/home/cys2400-bharath/Desktop/ex_8
cys24004-bharath@VM:~/Desktop/ex_8$ export MY_VAR="This is a custom environment variable"
cys24004-bharath@VM:~/Desktop/ex_8$ ./foo | grep MY_VAR
MY_VAR=This is a custom environment variable
```

### Observation:

The Set-UID program should inherit all environment variables from the calling process (the shell), including PATH, LD\_LIBRARY\_PATH, and any custom variables. However, some variables that could create security risks, like LD\_LIBRARY\_PATH, might be cleared or changed by the system for Set-UID programs. By checking the output, we can see how environment variables are inherited by Set-UID programs and whether any were adjusted for security reasons.



## 6. The PATH Environment Variable and Set-UID Programs

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. In Bash, you can change the `PATH` environment variable in the following way (this example adds the directory `/home/seed` to the beginning of the `PATH` environment variable):

```
$ export PATH=/home/seed:$PATH
```

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:

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

Please compile the above program, change its owner to `root`, and make it a Set-UID program. Can you get this Set-UID program to run your own malicious code, instead of `/bin/ls`? If you can, is your malicious code running with the `root` privilege? 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 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
```

```
cys24004-bharath@VM:~/Desktop/ex_8$ gcc setuid_ls.c -o setuid_ls
cys24004-bharath@VM:~/Desktop/ex_8$ sudo chown root setuid_ls
cys24004-bharath@VM:~/Desktop/ex_8$ sudo chmod 4755 setuid_ls
cys24004-bharath@VM:~/Desktop/ex_8$ sudo ln -sf /bin/zsh /bin/sh
cys24004-bharath@VM:~/Desktop/ex_8$ echo 'echo "This is a malicious script!"' > ~/malicious_ls.sh
cys24004-bharath@VM:~/Desktop/ex_8$ chmod +x ~/malicious_ls.sh
cys24004-bharath@VM:~/Desktop/ex_8$ export PATH=~/:$PATH
cys24004-bharath@VM:~/Desktop/ex_8$ ./setuid_ls
file1.txt  foo      myenv.c    myprintenv.c  mysystem.c  setuid_ls
file2.txt  myenv    myprintenv mysystem      printenv.c  setuid_ls.c
cys24004-bharath@VM:~/Desktop/ex_8$
```

### Observation:

Instead of running the actual `/bin/ls` command, the Set-UID program will run the malicious `ls` script. If the malicious code runs with root privileges, it indicates that the Set-UID program is vulnerable to a PATH manipulation attack.

## 7. The LD PRELOAD Environment Variable and Set-UID Programs

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.

Step 1. First, we will see how these environment variables influence the behavior of dynamic loader/linker when running a normal program. Please 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`:

```
#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 ```):

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

3. Now, set the LD PRELOAD environment variable:

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

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

```
/* myprog.c */
#include <unistd.h>
int main()
{
```



```
sleep(1);  
return 0;  
}
```

Step 2. After you have done the above, please run myprog under the following conditions, and observe what happens.

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

```
cys24004-bharath@VM:~/Desktop/ex_8$ gcc -fPIC -g -c mylib.c  
cys24004-bharath@VM:~/Desktop/ex_8$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc  
cys24004-bharath@VM:~/Desktop/ex_8$ export LD_PRELOAD=./libmylib.so.1.0.1  
cys24004-bharath@VM:~/Desktop/ex_8$ gcc myprog.c -o myprog  
cys24004-bharath@VM:~/Desktop/ex_8$ ./myprog  
I am not sleeping!
```

- Make myprog a Set-UID root program, and run it as a normal user.

```
cys24004-bharath@VM:~/Desktop/ex_8$ sudo chown root myprog  
cys24004-bharath@VM:~/Desktop/ex_8$ sudo chmod 4755 myprog  
cys24004-bharath@VM:~/Desktop/ex_8$ ./myprog
```

- Make myprog a Set-UID root program, export the LD PRELOAD environment variable again in the root account and run it.

```
cys24004-bharath@VM:~/Desktop/ex_8$ sudo su  
root@VM:/home/cys24004/Desktop/ex_8# export LD_PRELOAD=./libmylib.so.1.0.1  
root@VM:/home/cys24004/Desktop/ex_8# ./myprog  
I am not sleeping!
```

- Make myprog a Set-UID user1 program (i.e., the owner is user1, which is another user account), export the LD PRELOAD environment variable again in a different user's account (not-root user) and run it.

```
root@VM:/home/cys24004/Desktop/ex_8# sudo adduser user1
```

```
root@VM:/home/cys24004/Desktop/ex_8# sudo chown user1 myprog  
root@VM:/home/cys24004/Desktop/ex_8# su user1  
user1@VM:/home/cys24004/Desktop/ex_8$ export LD_PRELOAD=./libmylib.so.1.0.1  
user1@VM:/home/cys24004/Desktop/ex_8$ ./myprog  
I am not sleeping!  
user1@VM:/home/cys24004/Desktop/ex_8$
```

Step 3. You should be able to observe different behaviors in the scenarios described above, even though you are running the same program. You need to figure out what causes the difference. Environment variables play a role here. Please design an experiment to figure out the main causes, and explain why the behaviors in Step 2 are different. (Hint: the child process may not inherit the LD \* environment variables).

### Observation:

If the Set-UID program (./myprog) prints "I'm not sleeping!", it means the LD\_PRELOAD environment variable is still active, and the program is using the custom library's sleep function override. However, if nothing is printed, it suggests the system is ignoring the LD\_PRELOAD

variable for Set-UID programs as a security measure to prevent malicious manipulation of libraries.

## 8. Invoking External Programs Using `system()` versus `execve()`

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 affects 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.

`/catall.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, make it a root-owned Set-UID program. The program will use `system()` to invoke the command. If you were Bob, can you compromise the integrity of the system? For example, can you remove a file that is not writable to you?

```
cys24004-bharath@VM:~/Desktop/ex_8$ gcc catall.c -o catall
cys24004-bharath@VM:~/Desktop/ex_8$ sudo chown root catall
cys24004-bharath@VM:~/Desktop/ex_8$ sudo chmod 4755 catall
cys24004-bharath@VM:~/Desktop/ex_8$ vim file.txt
cys24004-bharath@VM:~/Desktop/ex_8$ ./catall file.txt
hello! sample text file...
```

Step 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. Do your attacks in Step 1 still work? Please describe and explain your observations.

```
cys24004-bharath@VM:~/Desktop/ex_8$ gcc catall.c -o catall
cys24004-bharath@VM:~/Desktop/ex_8$ sudo chown root catall
cys24004-bharath@VM:~/Desktop/ex_8$ sudo chmod 4755 catall
cys24004-bharath@VM:~/Desktop/ex_8$ ./catall file.txt
hello! sample text file...
```

```
cys24004-bharath@VM:~/Desktop/ex_8$ ./catall "; rm file.txt"
/bin/cat: '; rm file.txt': No such file or directory
```

## 9. Capability Leaking

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.

Compile the following program, change its owner to root, and make it a Set-UID program. Run the program as a normal user. Can you exploit the capability leaking vulnerability in this program? The goal is to write to the `/etc/zxx` file as a normal user.

```
//cap_leak.c
void main()
{
    int fd;
    char *v[2];
    /* Assume that /etc/zxx is an important system file,
     * and it is owned by root with permission 0644.
     * Before running this program, you should create
     * the file /etc/zxx first. */
    fd = open("/etc/zxx", O_RDWR | O_APPEND);
    if (fd == -1) {
        printf("Cannot open /etc/zxx\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);
}
```

```
cys24004-bharath@VM:~/Desktop/ex_8$ gcc cap_leak.c -o cap_leak
cys24004-bharath@VM:~/Desktop/ex_8$ sudo touch /etc/zzz
cys24004-bharath@VM:~/Desktop/ex_8$ sudo chmod 0644 /etc/zzz
cys24004-bharath@VM:~/Desktop/ex_8$ sudo ./cap_leak
fd is 3
#
```

**Observation:**

The program opens a file with root privileges but fails to close the file descriptor after dropping privileges. This allows a normal user to write to a file they shouldn't have access to.