Karthikeyan G
CB.SC.P2CYS24008

Secure Coding Lab Experiment - 8

Environment Variable and Set-UID Program

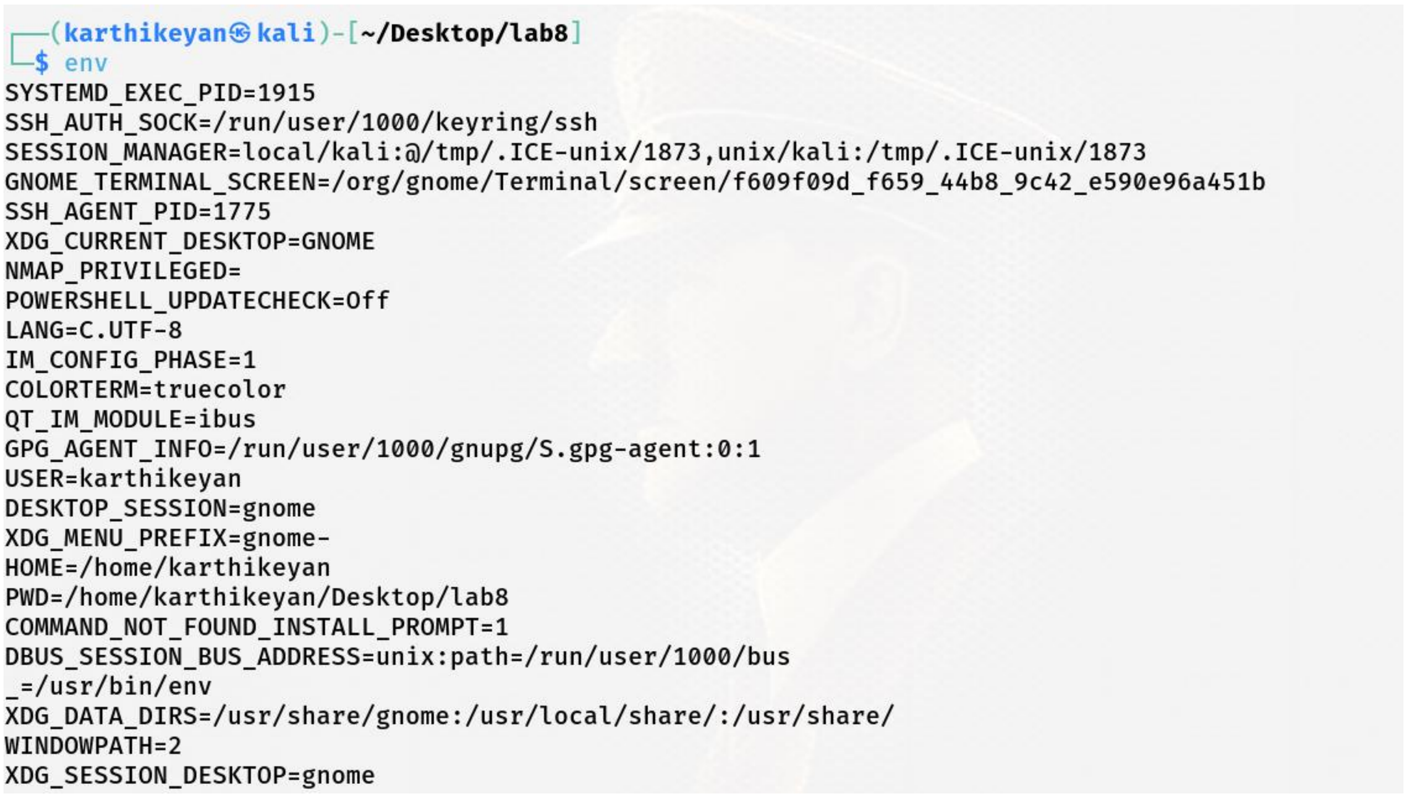
1.Manipulating Environment Variables

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 use 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:

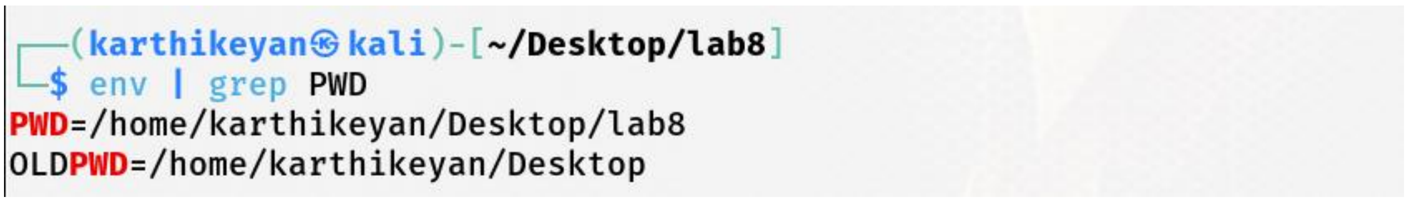
• Use **printenv** or **env** command to print o ut the environment variables. If you are interested in some particular environment variables, suc.ljgbv h as PWD, you can use "print env PWD" or "env | grep PWD".

• 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

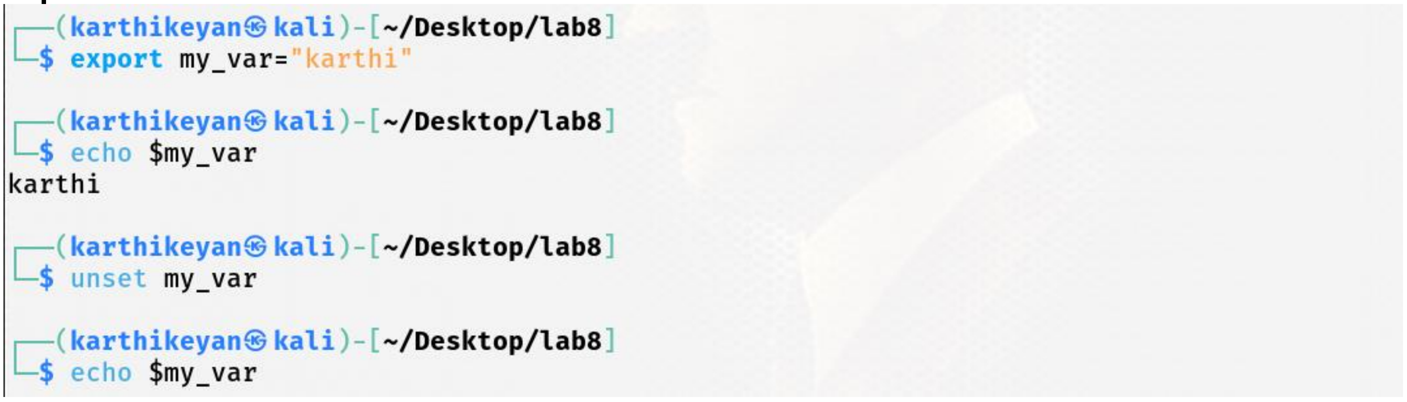


```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ env
SYSTEMD_EXEC_PID=1915
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
SESSION_MANAGER=local/kali:@/tmp/.ICE-unix/1873,unix/kali:/tmp/.ICE-unix/1873
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/f609f09d_f659_44b8_9c42_e590e96a451b
SSH_AGENT_PID=1775
XDG_CURRENT_DESKTOP=GNOME
NMAP_PRIVILEGED=
POWERSHELL_UPDATECHECK=Off
LANG=C.UTF-8
IM_CONFIG_PHASE=1
COLORTERM=truecolor
QT_IM_MODULE=ibus
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
USER=karthikeyan
DESKTOP_SESSION=gnome
XDG_MENU_PREFIX=gnome-
HOME=/home/karthikeyan
PWD=/home/karthikeyan/Desktop/lab8
COMMAND_NOT_FOUND_INSTALL_PROMPT=1
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
_=/usr/bin/env
XDG_DATA_DIRS=/usr/share/gnome:/usr/local/share/:/usr/share/
WINDOWPATH=2
XDG_SESSION_DESKTOP=gnome
```



```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ env | grep PWD
PWD=/home/karthikeyan/Desktop/lab8
OLDPWD=/home/karthikeyan/Desktop
```

export and unset



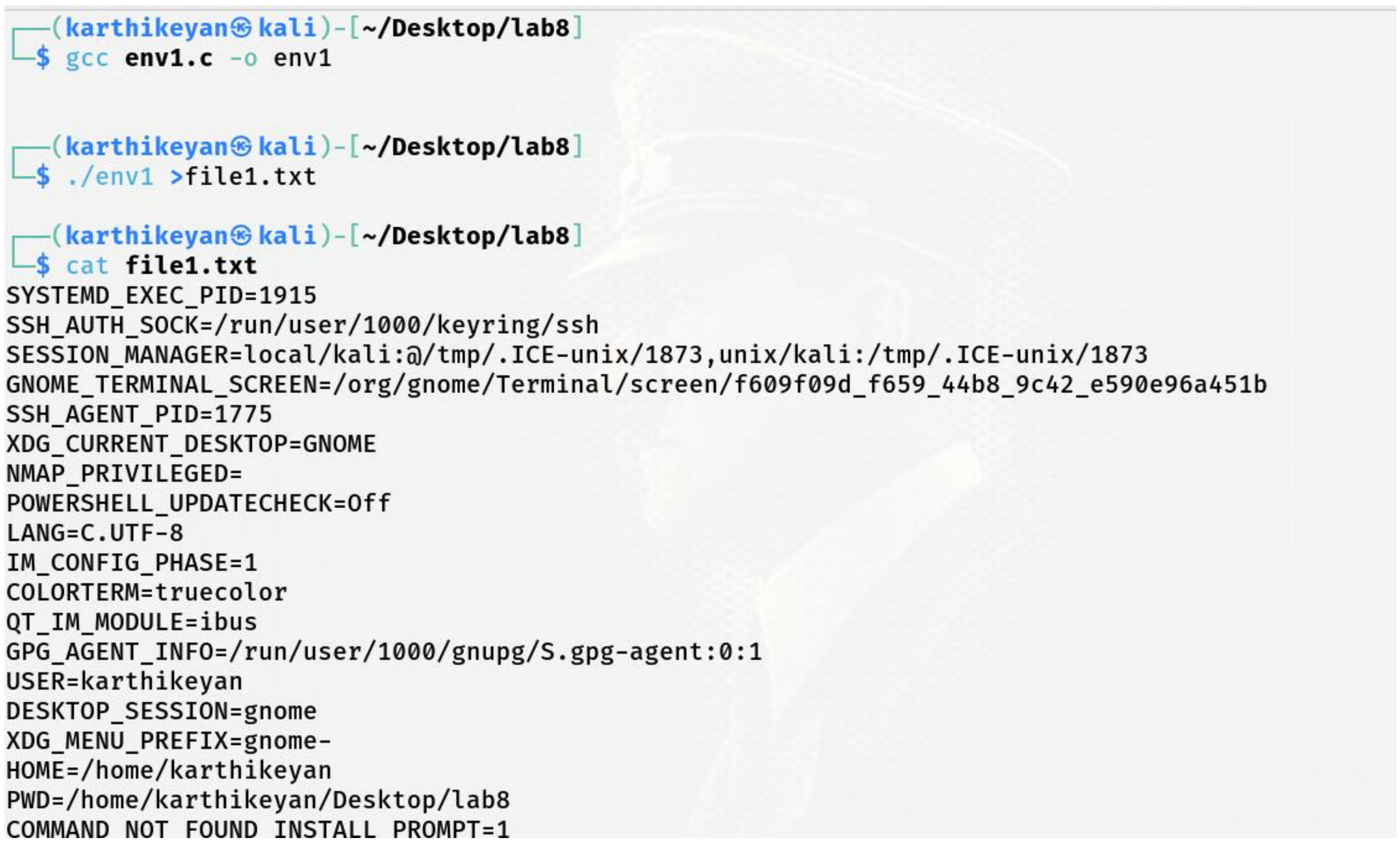
```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ export my_var="karthi"

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ echo $my_var
karthi

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ unset my_var

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ echo $my_var
```

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.

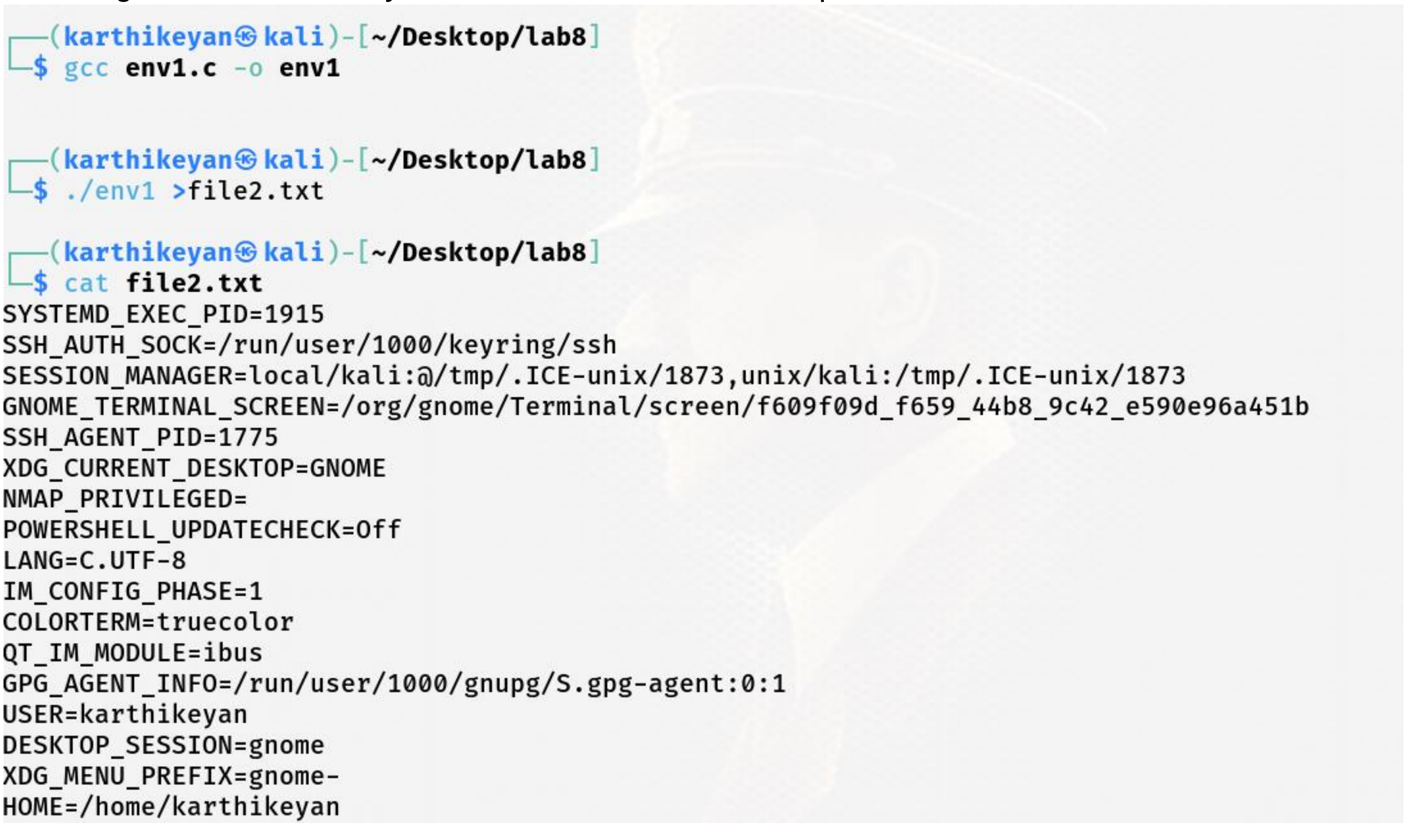
Step 1. Please compile and run the following program, and describe your observation. The program can be found in the Labsetup folder; it can be compiled using "gcc myprintenv.c", which will generate a binary called a.out. Let's run it and save the output into a file using "a.out > file".

```
#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);
}
}
```



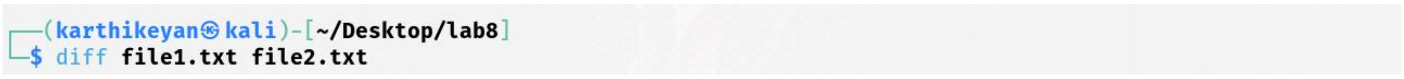
```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ gcc env1.c -o env1


┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ ./env1 >file1.txt

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ cat file1.txt
SYSTEMD_EXEC_PID=1915
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
SESSION_MANAGER=local/kali:@/tmp/.ICE-unix/1873,unix/kali:/tmp/.ICE-unix/1873
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/f609f09d_f659_44b8_9c42_e590e96a451b
SSH_AGENT_PID=1775
XDG_CURRENT_DESKTOP=GNOME
NMAP_PRIVILEGED=
POWERSHELL_UPDATECHECK=Off
LANG=C.UTF-8
IM_CONFIG_PHASE=1
COLORTERM=truecolor
QT_IM_MODULE=ibus
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
USER=karthikeyan
DESKTOP_SESSION=gnome
XDG_MENU_PREFIX=gnome-
HOME=/home/karthikeyan
PWD=/home/karthikeyan/Desktop/lab8
COMMAND_NOT_FOUND_INSTALL_PROMPT=1
```

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



```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ gcc env1.c -o env1


┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ ./env1 >file2.txt

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ cat file2.txt
SYSTEMD_EXEC_PID=1915
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
SESSION_MANAGER=local/kali:@/tmp/.ICE-unix/1873,unix/kali:/tmp/.ICE-unix/1873
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/f609f09d_f659_44b8_9c42_e590e96a451b
SSH_AGENT_PID=1775
XDG_CURRENT_DESKTOP=GNOME
NMAP_PRIVILEGED=
POWERSHELL_UPDATECHECK=Off
LANG=C.UTF-8
IM_CONFIG_PHASE=1
COLORTERM=truecolor
QT_IM_MODULE=ibus
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
USER=karthikeyan
DESKTOP_SESSION=gnome
XDG_MENU_PREFIX=gnome-
HOME=/home/karthikeyan
```

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



```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ diff file1.txt file2.txt
```

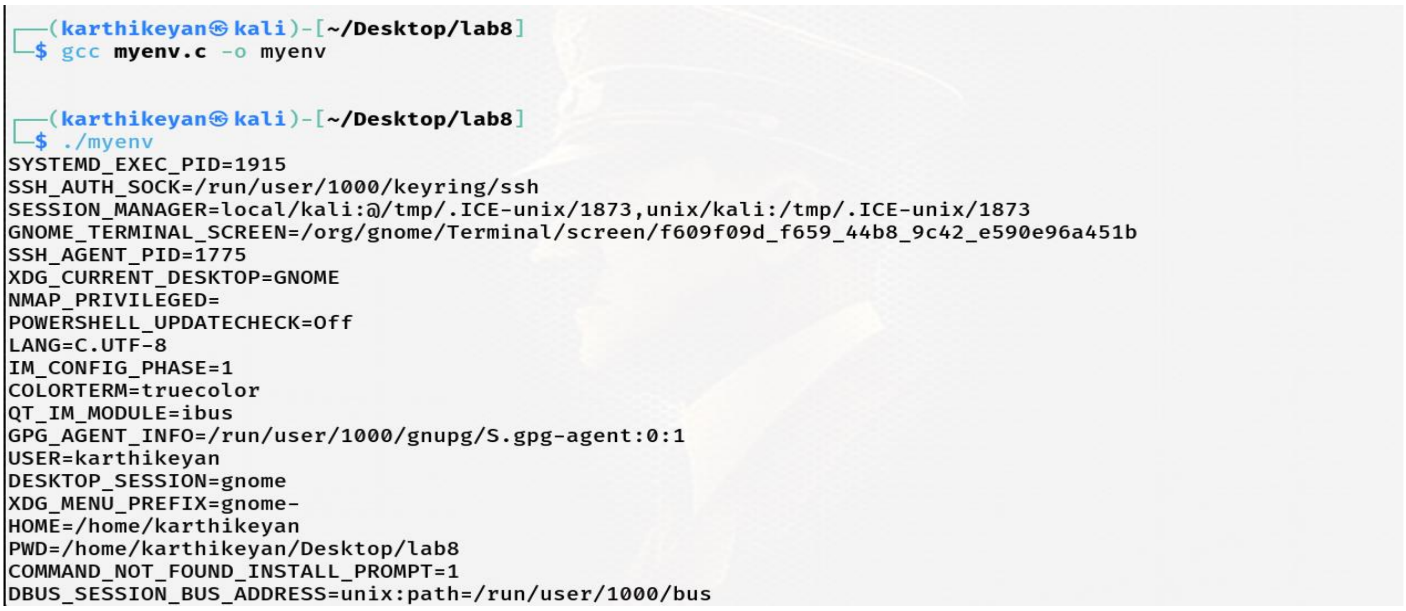
Observation: The child process seamlessly inherits the environment variables from its parent, ensuring no variations between the two files..

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.

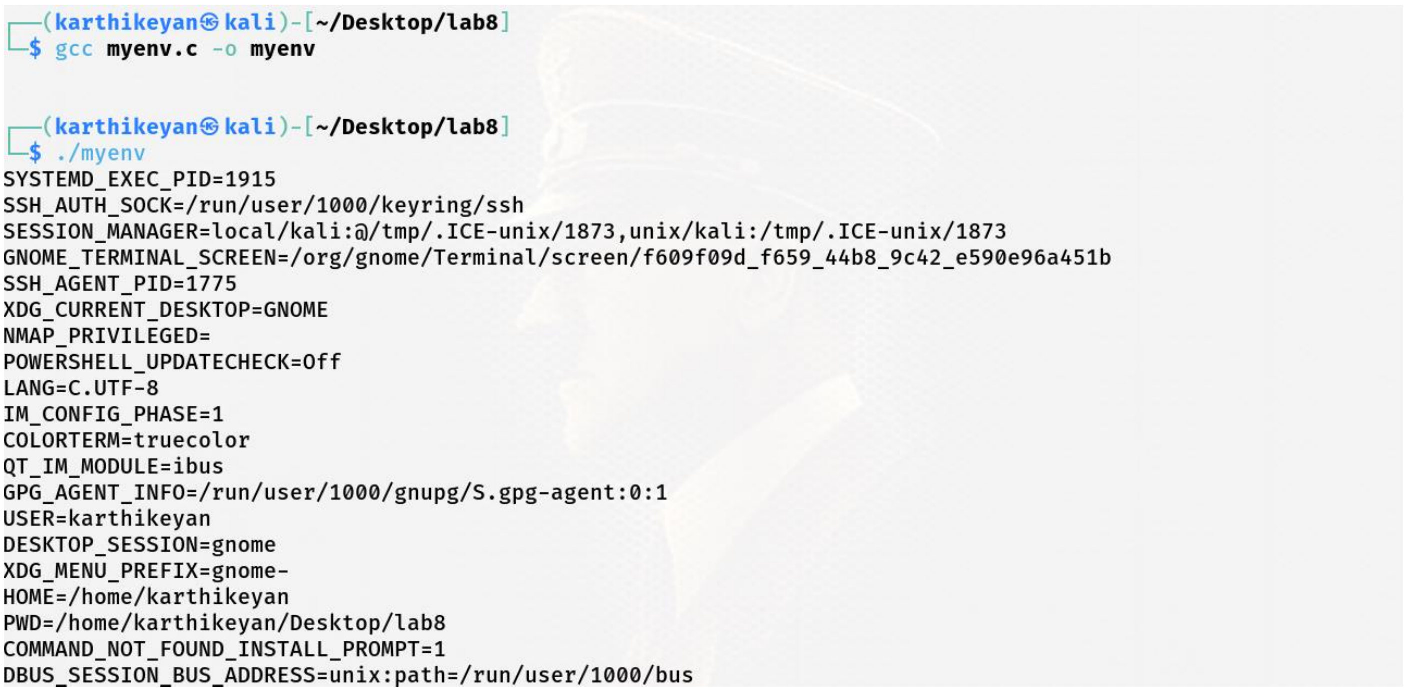
```
//myenv.c

#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); À
return 0 ;
}
```



```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ gcc myenv.c -o myenv


┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ ./myenv
SYSTEMD_EXEC_PID=1915
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
SESSION_MANAGER=local/kali:@/tmp/.ICE-unix/1873,unix/kali:/tmp/.ICE-unix/1873
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/f609f09d_f659_44b8_9c42_e590e96a451b
SSH_AGENT_PID=1775
XDG_CURRENT_DESKTOP=GNOME
NMAP_PRIVILEGED=
POWERSHELL_UPDATECHECK=Off
LANG=C.UTF-8
IM_CONFIG_PHASE=1
COLORTERM=truecolor
QT_IM_MODULE=ibus
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
USER=karthikeyan
DESKTOP_SESSION=gnome
XDG_MENU_PREFIX=gnome-
HOME=/home/karthikeyan
PWD=/home/karthikeyan/Desktop/lab8
COMMAND_NOT_FOUND_INSTALL_PROMPT=1
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
```

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

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



```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ gcc myenv.c -o myenv


┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ ./myenv
SYSTEMD_EXEC_PID=1915
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
SESSION_MANAGER=local/kali:@/tmp/.ICE-unix/1873,unix/kali:/tmp/.ICE-unix/1873
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/f609f09d_f659_44b8_9c42_e590e96a451b
SSH_AGENT_PID=1775
XDG_CURRENT_DESKTOP=GNOME
NMAP_PRIVILEGED=
POWERSHELL_UPDATECHECK=Off
LANG=C.UTF-8
IM_CONFIG_PHASE=1
COLORTERM=truecolor
QT_IM_MODULE=ibus
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
USER=karthikeyan
DESKTOP_SESSION=gnome
XDG_MENU_PREFIX=gnome-
HOME=/home/karthikeyan
PWD=/home/karthikeyan/Desktop/lab8
COMMAND_NOT_FOUND_INSTALL_PROMPT=1
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
```

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

Environment **Variables Management**: The behaviour of environment variables with execve() depends on how they are handled.

Same **Environment Variables in Both Executions**:

- Indicates that execve() automatically inherits the calling process's environment by default.

Differences **in Environment Variables**:

- Suggests that execve() does not pass environment variables unless they are explicitly specified.

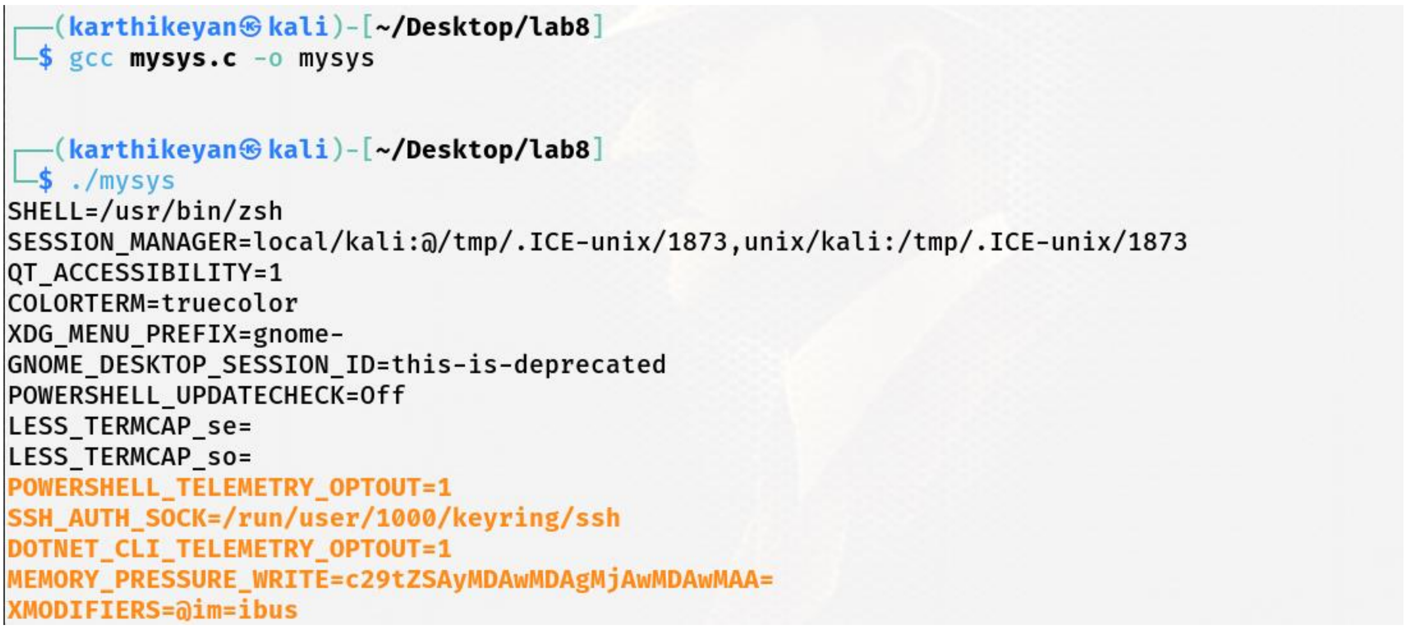
Conclusion:

- execve() 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 ;
}
```



```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ gcc mysys.c -o mysys


┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ ./mysys
SHELL=/usr/bin/zsh
SESSION_MANAGER=local/kali:@/tmp/.ICE-unix/1873,unix/kali:/tmp/.ICE-unix/1873
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
POWERSHELL_UPDATECHECK=Off
LESS_TERMCAP_se=
LESS_TERMCAP_so=
POWERSHELL_TELEMETRY_OPTOUT=1
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
DOTNET_CLI_TELEMETRY_OPTOUT=1
MEMORY_PRESSURE_WRITE=c29tZSAyMDAwMDAgMjAwMDAwMAA=
XMODIFIERS=@im=ibus
```

This code will execute /usr/bin/env through the system() function, which displays the current environment variables. The system() function uses /bin/sh to run the specified command, ensuring that the environment variables from the calling process (our C program) are passed to /bin/sh. In my case, since I have zsh configured as my default shell, it will execute through zsh instead. As a result, zsh forwards these environment variables to /usr/bin/env. By examining the output, we can verify that the environment variables from the original process are successfully passed to the program invoked with system()

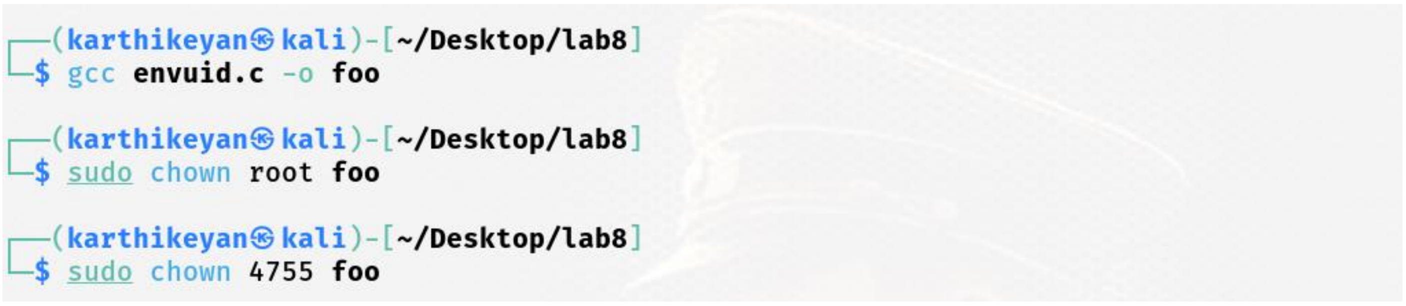
5. Environment Variables and Set-UID Programs

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.

```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.



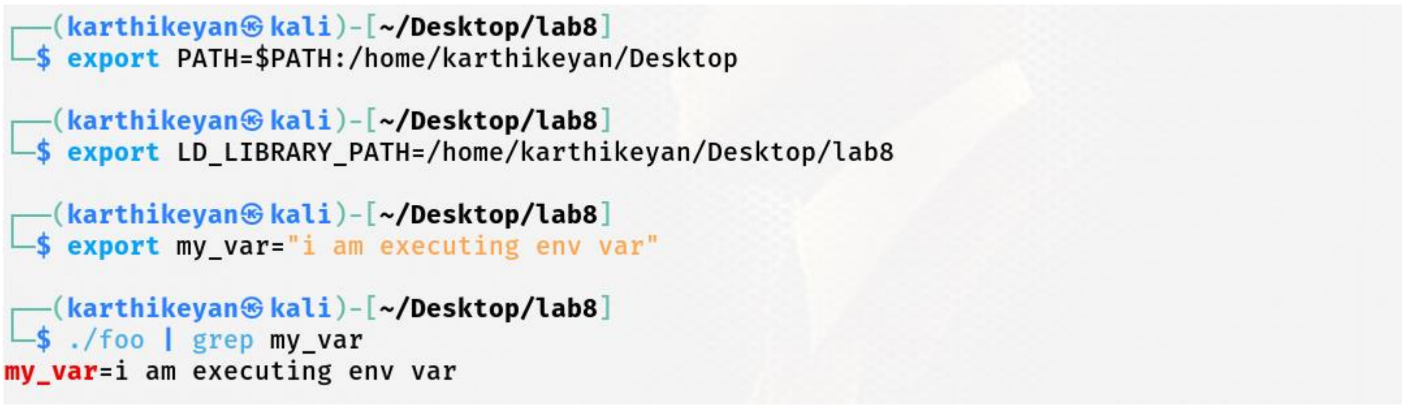
```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ gcc envuid.c -o foo

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ sudo chown root foo

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ sudo chown 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 existed):

• 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.

Observation:



```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ export PATH=$PATH:/home/karthikeyan/Desktop

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ export LD_LIBRARY_PATH=/home/karthikeyan/Desktop/lab8

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ export my_var="i am executing env var"

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ ./foo | grep my_var
my_var=i am executing env var
```

Set-UID Program and Environment Variable Handling

Set-UID programs inherit environment variables from the calling shell, including PATH, LD_LIBRARY_PATH, and custom variables. However, for security reasons, certain variables especially those that could introduce risks like LD_LIBRARY_PATH—may be cleared or modified by the system. By analyzing the output of Set-UID programs, we can understand how these environment variables are inherited and identify any adjustments made for enhanced security.

6. The path Environment Variables 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
```

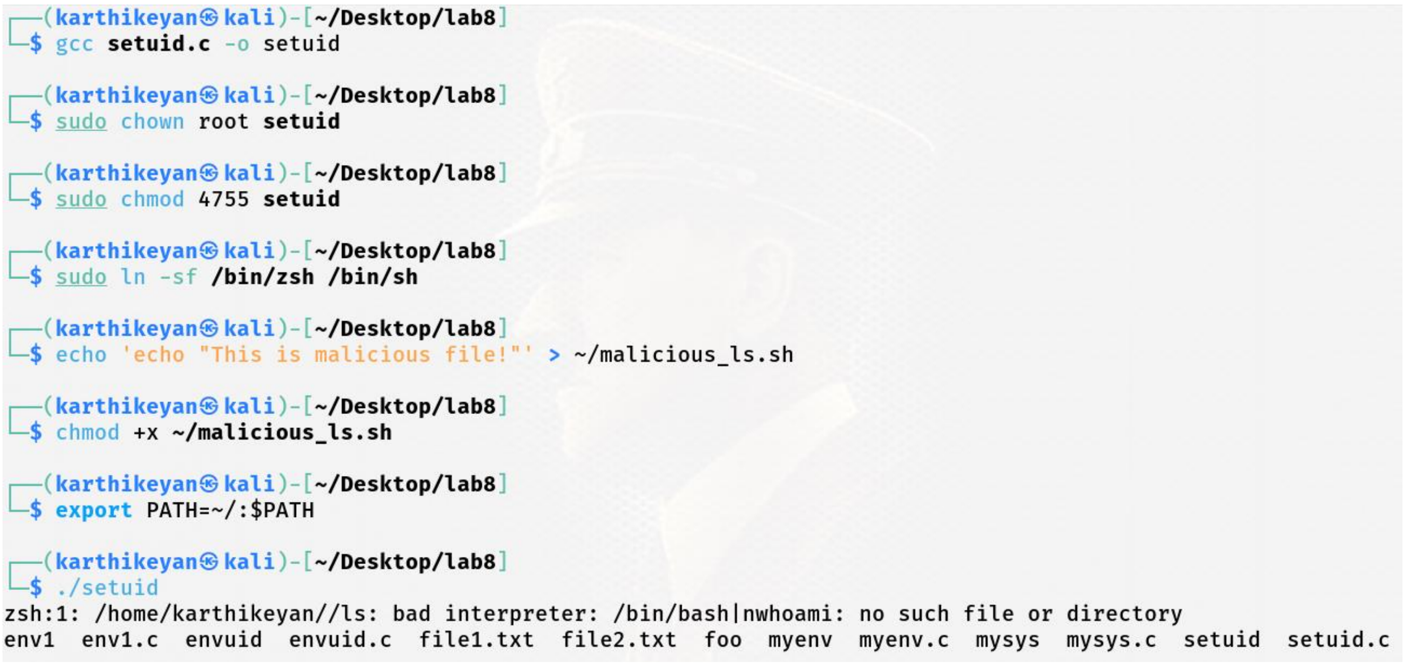
The Set-UID program below is supposed to execute the /bin/ls command; however, the programmer only uses the relative path for the ls command, rather than the absolute path:

```
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
```



```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ gcc setuid.c -o setuid

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ sudo chown root setuid

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ sudo chmod 4755 setuid

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ sudo ln -sf /bin/zsh /bin/sh

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ echo 'echo "This is malicious file!"' > ~/malicious_ls.sh

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ chmod +x ~/malicious_ls.sh

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ export PATH=~/:$PATH

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ ./setuid
zsh:1: /home/karthikeyan//ls: bad interpreter: /bin/bash|nwhoami: no such file or directory
env1   env1.c   envuid   envuid.c   file1.txt   file2.txt   foo   myenv   myenv.c   mysys   mysys.c   setuid   setuid.c
```

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
```

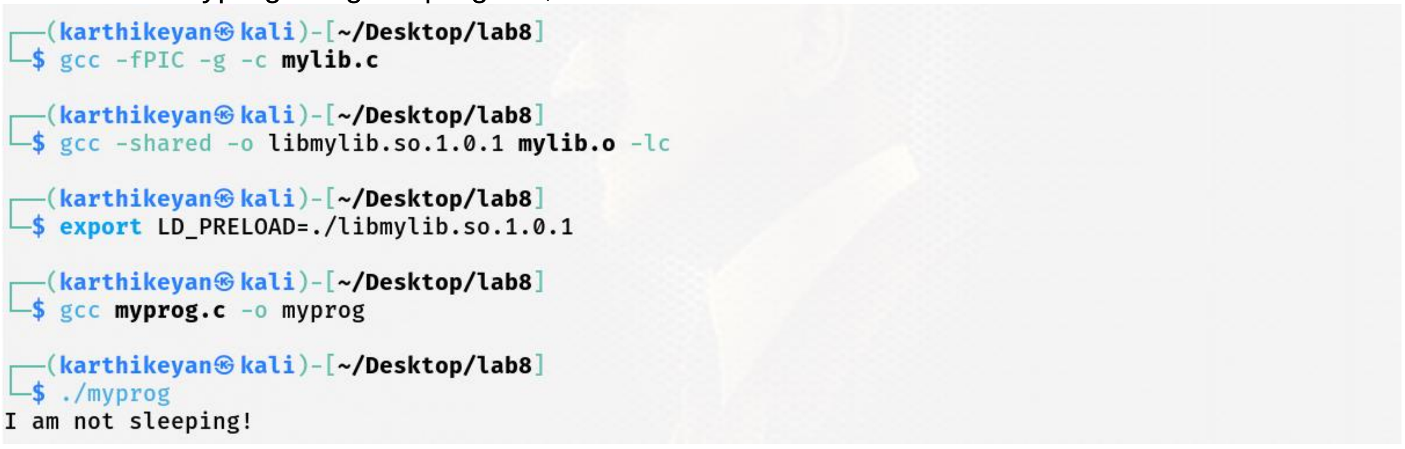
4. Finally, compile the following program myprog, and in the same directory as the above dynamic link library libmylib.so.1.0.1:

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

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

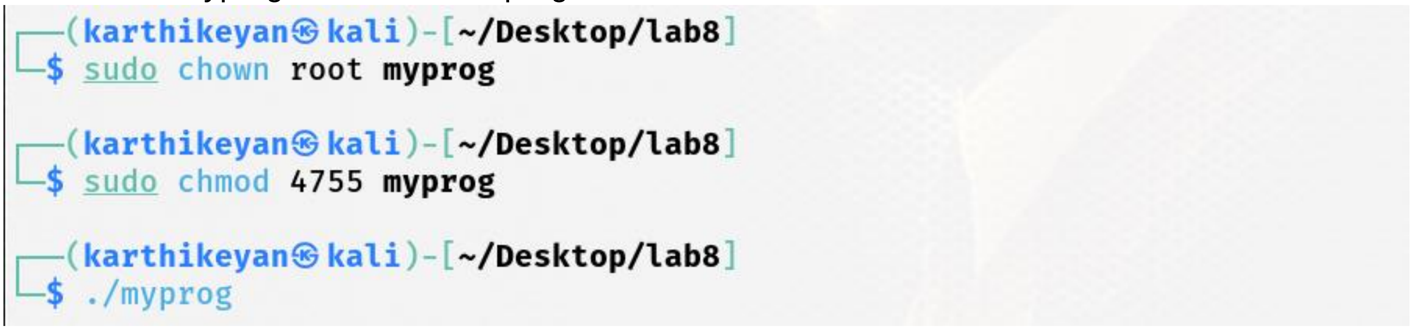
Step 2. After you have done the above, please run myprog under the following conditions, and observe what happens.

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



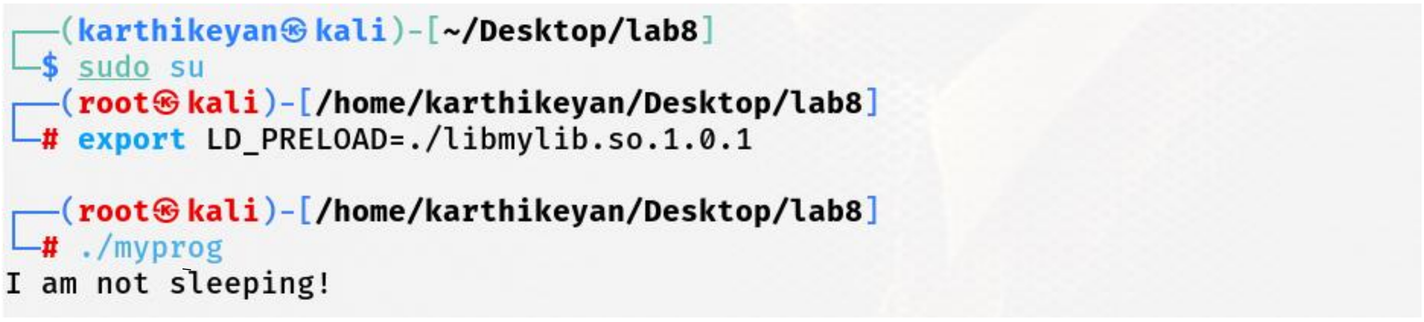
```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ gcc -fPIC -g -c mylib.c

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ export LD_PRELOAD=./libmylib.so.1.0.1

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ gcc myprog.c -o myprog

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ ./myprog
I am not sleeping!
```

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



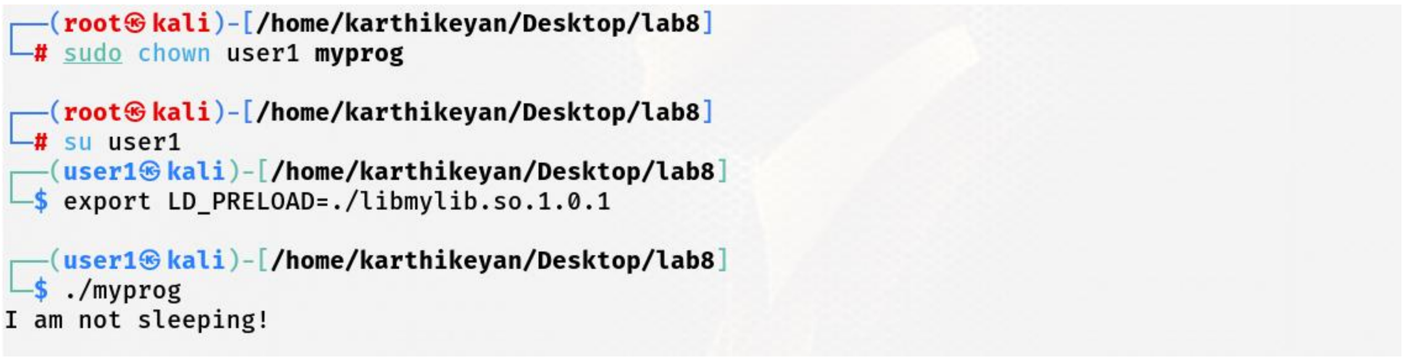
```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ sudo chown root myprog

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ sudo chmod 4755 myprog

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ ./myprog
```

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



```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ sudo su
┌──(root㉿kali)-[/home/karthikeyan/Desktop/lab8]
└─# export LD_PRELOAD=./libmylib.so.1.0.1

┌──(root㉿kali)-[/home/karthikeyan/Desktop/lab8]
└─# ./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㉿kali)-[/home/karthikeyan/Desktop/lab8]
└─# sudo chown user1 myprog

┌──(root㉿kali)-[/home/karthikeyan/Desktop/lab8]
└─# su user1
┌──(user1㉿kali)-[/home/karthikeyan/Desktop/lab8]
└─$ export LD_PRELOAD=./libmylib.so.1.0.1

┌──(user1㉿kali)-[/home/karthikeyan/Desktop/lab8]
└─$ ./myprog
I am not sleeping!
```

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) outputs "I'm not sleeping!", it indicates that the LD_PRELOAD environment variable is active, allowing a custom library's sleep function to override the default. Conversely, if there is no output, it suggests that the system is disregarding the LD_PRELOAD variable for Set-UID programs as a security measure against potential library manipulation.

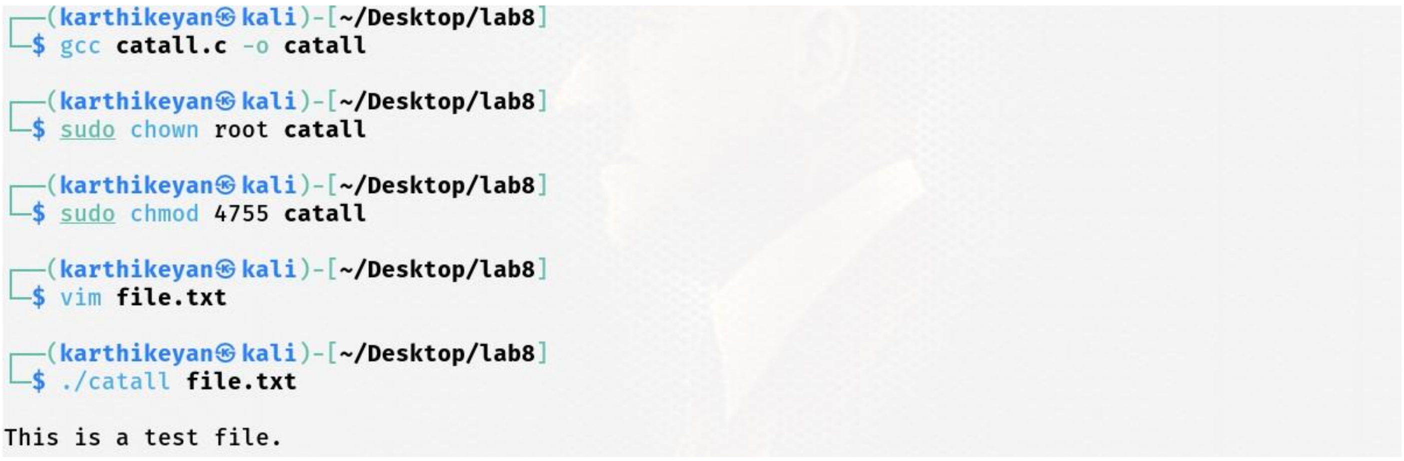
8. Invoking External Programs Using system() versus execve()

Although system() and execve() can both be used to run new programs, system() is quite danger-ous 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 conse-quence, 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.

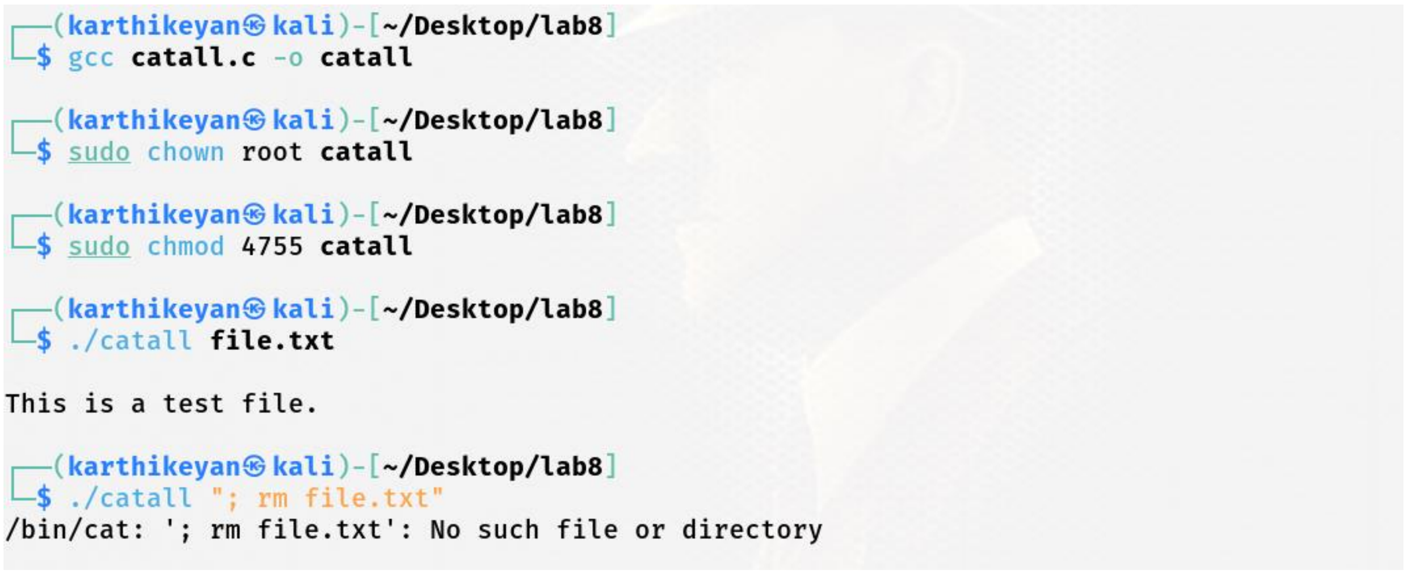
```
/catall.c
int main(int argc, char *argv[])
    {
    char *v[3];
    char *command;
    if(argc < 2) {
    printf("Please type a file name.\n");
    return 1;
    }

    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?



```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ gcc catall.c -o catall

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ sudo chown root catall

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ sudo chmod 4755 catall

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ vim file.txt

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ ./catall file.txt

This is a test 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.



```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ gcc catall.c -o catall

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ sudo chown root catall

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ sudo chmod 4755 catall

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ ./catall file.txt

This is a test file.

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ ./catall "; rm file.txt"
/bin/cat: '; rm file.txt': No such file or directory
```

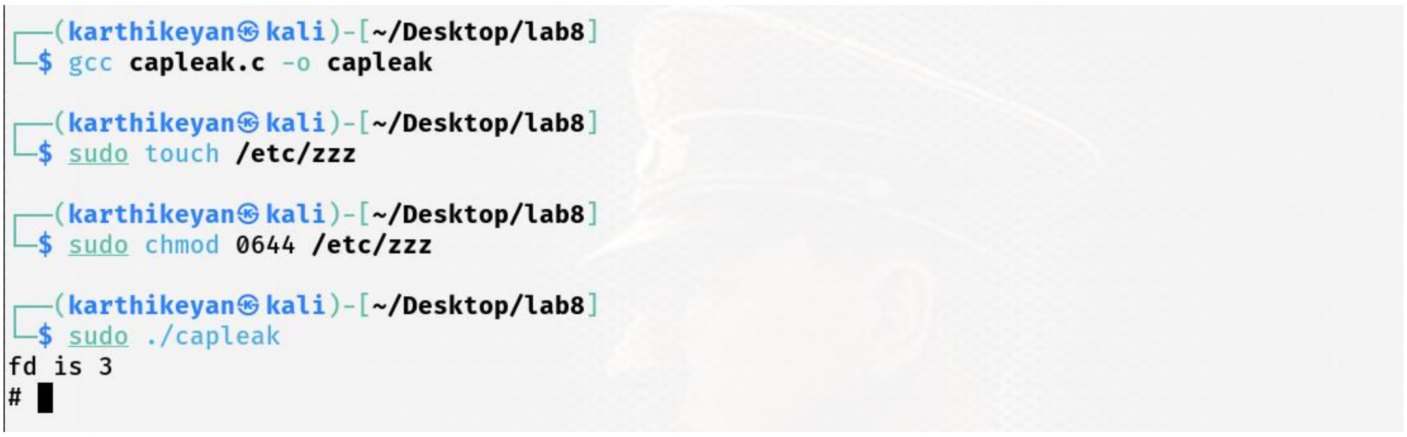
9. Capability Leaking

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/zzz file as a normal user.

```
// cap_leak.c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

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

    /*
     * Assume that /etc/zzz is an
important system file,
     * owned by root with permission
0644.
     * Before running this program,
create the file /etc/zzz.
     */

    // Attempt to open the file with
read/write and append permissions
    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());

    // Prepare to execute /bin/sh
    v[0] = "/bin/sh";
    v[1] = NULL;

    // Execute the shell
    execve(v[0], v, NULL);

    return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ gcc capleak.c -o capleak

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ sudo touch /etc/zzz

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ sudo chmod 0644 /etc/zzz

┌──(karthikeyan㉿kali)-[~/Desktop/lab8]
└─$ sudo ./capleak
fd is 3
# █
```

Observation:
The program opens a file with root privileges but neglects to close the file descriptor after dropping privileges, potentially allowing unauthorized users to write to the file, which they shouldn't be able to access.