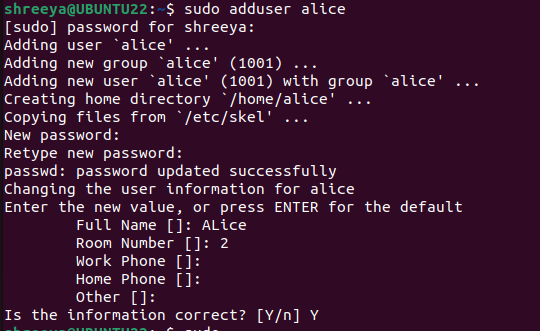
**LAB ASSIGNMENT 4**

**CH 1: SET-UID Privileged Programs and Attacks on Them**

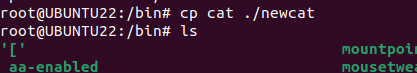
***Q1. Alice runs a Set-UID program that is owned by Bob. The program tries to read from /tmp/x, which is readable to Alice, but not to anybody else. Can this program successfully read from the file?***

Ans.

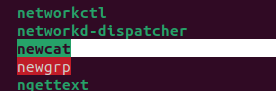
1. Creating new user – Alice



1. Copying the cat file to new cat command file ie ‘newcat’ in this case.



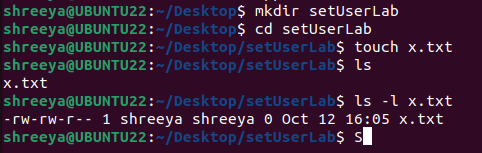
1. The new file shows up as an output of ls



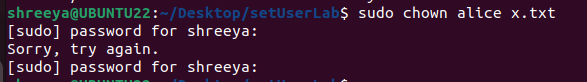
1. We see the owner status of the new file and we see that the owner is the root.



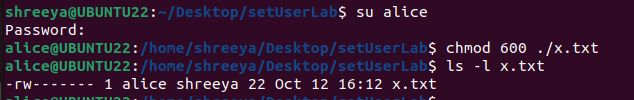
1. Creating a file in Bob(Shreeya) named x.txt. The current owner is Bob.



1. Changing the owner of the file to Alice.



1. Making the file read only by alice.



1. The owner of newcat is Bob



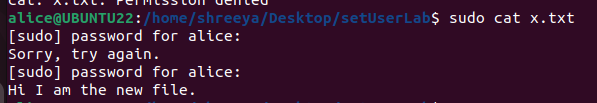
1. Alice cannot read x.txt read the file using mycat



1. Bob can still read it



1. Alice can still read it using cat command as this command has root permissions.



***Q2. A process tries to open a file for read. The process’s effective user ID is 1000, and real user ID is 2000. The file is readable to user ID 2000, but not to user ID 1000. Can this process successfully open the file?***

Ans.

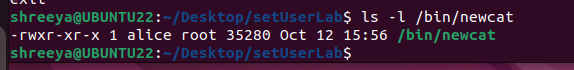
1. ID for Alice



1. ID for Shreeya



1. Current ownership status of newcat



1. Changing the owner of newcat.



1. Since owner of newcat is not Alice that is why it can not be used by Alice



1. Shreeya can still use it.

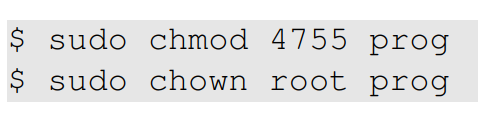


***Q3. A root-owned Set-UID program allows a normal user to gain the root privilege while executing the program. What prevents the user from doing bad things using the privilege?***

Ans. When a Set-UID program runs, it gains the privileges of the user who owns the executable (usually root) rather than the user who is executing the program. This can potentially allow normal users to perform actions with elevated privileges. However, there are several security measures in place to prevent abuse of this privilege:

* Minimal Privilege Principle: Set-UID programs should be designed to do the absolute minimum necessary to accomplish their task. Unnecessary privileges should not be elevated. This principle reduces the potential damage a user can do if they manage to exploit the program.

***Q4. We are trying to turn a program prog owned by the seed user into a Set-UID program that is owned by root. Can running the following commands achieve the goal?***



Ans. Yes, the commands you provided will set the Set-UID permission for the program prog and change its ownership to root, achieving the goal of making it a Set-UID program owned by root.

Here's what each command does:

1. **sudo chmod 4755 prog**: This command sets the permissions of the file prog to 4755. In octal notation, the 4 at the beginning represents the Set-UID permission, and 755 sets the read, write, and execute permissions for the owner, and read and execute permissions for others.
2. **sudo chown root prog**: This command changes the owner of the file prog to the root user.

***Q5. The chown command automatically disables the Set-UID bit, when it changes the owner of a Set-UID program. Please explain why it does that.***

Ans. When the ‘chown’ command is used to change the owner of a file, including the Set-UID program, the Set-UID program and Set-GID bits are automatically cleared to ensure security integrity. These are the reasons why this is important.

1. **Security Risk**: Allowing a non-privileged user to change the owner of a Set-UID program can be a significant security risk. If a regular user could change the owner of a Set-UID program to themselves and then execute it, they would effectively gain unauthorized access to the elevated privileges associated with the original owner (such as root). This could lead to exploitation and unauthorized access to sensitive system resources.
2. **Privilege Escalation:** Imagine a scenario where a vulnerable Set-UID program exists on a system, and a user discovers a way to exploit it. If the user could change the owner of this program to themselves without losing the Set-UID bit, they could execute arbitrary code with elevated privileges, which is a classic privilege escalation attack.
3. **Maintaining Security Boundaries:** The Set-UID bit is a powerful feature that allows users to execute programs with the permissions of the file owner. Clearing the Set-UID bit when the owner is changed helps maintain the security boundaries established by the system administrator. It ensures that only privileged users (typically, users with root access) can control which programs run with elevated privileges.

***Q6. When we debug a program, we can change the program’s internal variables during the execution. This can change a program’s behavior. Can we use this technique to debug a Set-UID program and change its behavior? For example, if the program is supposed to open the /tmp/xyz file, we can modify the filename string, so the Set-UID program ends up opening /etc/passwd***.

Ans.

***Q7. Both system() and execve() can be used to execute external programs. Why is system() unsafe while execve() is safe?***

Ans.

**1. Command Injection Vulnerabilities:**

* **system()**: **system()** passes the command string to the shell to be executed. For example, if you use **system("ls -l")**, it would internally execute **/bin/sh -c "ls -l"**. This makes it vulnerable to command injection attacks. If the command string is constructed from user input without proper validation, an attacker can inject malicious commands, leading to unintended and potentially harmful behavior.
* **execve()**: **execve()** does not use a shell to execute programs. It requires you to specify the program's path and arguments directly. For instance, **execve("/bin/ls", ["ls", "-l", NULL], NULL)** would execute **/bin/ls** directly, without involving a shell. Because there's no shell to interpret commands, there's no opportunity for command injection vulnerabilities if the arguments are properly sanitized.

**2. Environment Variables:**

* **system()**: **system()** uses the system's environment variables. If an attacker can manipulate the environment variables, they can potentially influence the behavior of the executed command.
* **execve()**: With **execve()**, you have complete control over the environment variables passed to the executed program. This means you can set up a clean and controlled environment, minimizing the risk of unintended behavior due to environment variable manipulation.

**3. Return Values and Error Handling:**

* **system()**: **system()** does not provide detailed error handling or the ability to capture the command's output or exit status. This can make it challenging to handle errors and respond appropriately to different outcomes.
* **execve()**: **execve()** provides detailed control over error handling. It allows you to handle errors directly, capture the program's output, and respond accordingly, enhancing the security of the overall system.

In summary, **system()** is less safe because it involves passing commands through a shell, making it susceptible to command injection attacks and environment variable manipulation. On the other hand, **execve()** is safer because it allows direct execution of programs without involving a shell, provides control over environment variables, and enables proper error handling, reducing the risk of security vulnerabilities. When executing external programs, it's generally recommended to use functions like **execve()** and sanitize inputs to prevent security threats.

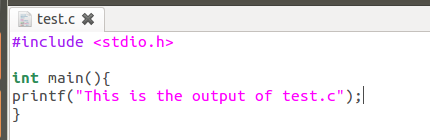
***Q8. When a program takes an input from users, we can redirect the input device, so the program can take the input from a file. For example, we can use prog < myfile to provide the data from myfile as input to the prog program. Now, if prog is a root-owned Set-UID program, can we use the following method to to get this privileged program to read from the /etc/shadow file?***

***prog < /etc/shadow***

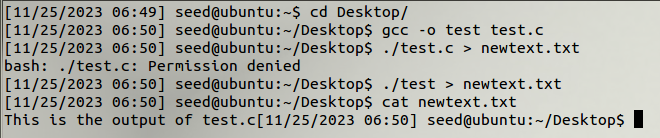
Ans:

* 1. **To demonstrate transferring the output of one file to a new file:**

TEST.C:

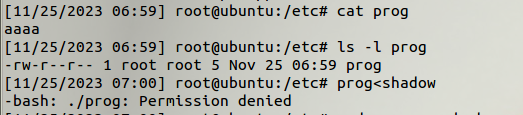


Output:



* 1. **To do so with SetUID programs:**

*The fact that prog is a Set-UID program with root privileges has nothing to do with the file that’s input after. Since a normal user can not access /etc/shadow, they can not use it to redirect the input device. Trying to do so will cause a permission denied error*

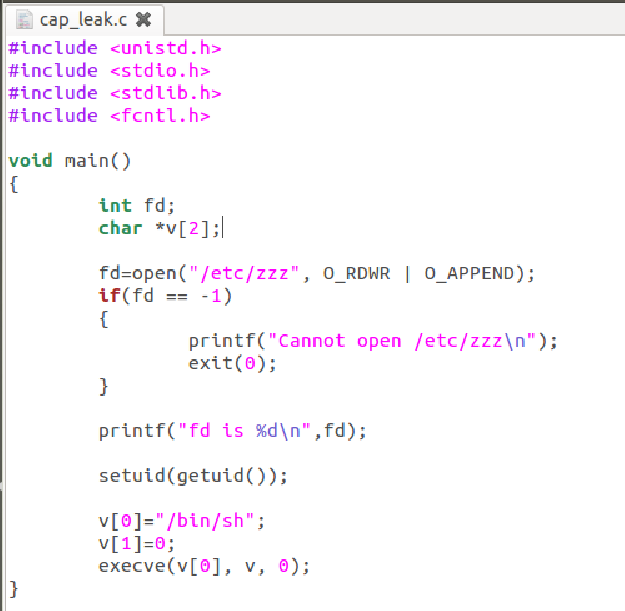
**

As we can see that prog is root owned set UID program and all the operations are happening is SUDO mode still we cannot achieve what has been asked for and we get a permission denied error.

***Q8.*** ***When a parent Set-UID process (effective user ID is root, and the real user ID is bob) creates a child process using fork(), the standard input, output, and error devices of the parent will be inherited by the child. If the child process drops its root privilege, it still retains the access right to these devices. This seems to be a capability leaking, similar to what we covered in Chapter 1.4.4. Can this pose any danger?***

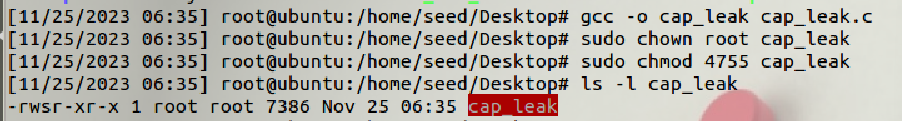
Ans: When a privileged process transitions to a non-privileged process, one of the common mistakes is capability leaking. The process may have gained some privileged capabilities when it was still privileged; when the privileges are downgraded, if the program does not clean up those capabilities, they 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. And the access to privileged resources is obviously a danger.

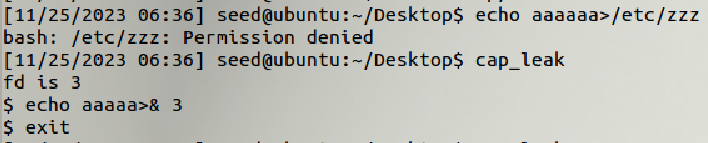
To demonstrate this we can run the following code:



Since the above program forgets to close the file, the file descriptor is still valid and the process which does not have privileges is still capable of writing to the /etc/zzz. From the execution result, we can see that the file descriptor number is 3. We can easily write to /etc/zzz using the command " echo . . . >&3 " , where " &3 " means file descriptor 3. Before running the Set- UID program, we were not able to write to the protected fil e /etc/zzz, but after gaining the file descriptor via the Set-UID program, we can successfully modify it.

Execution result:





We see from the output that after running **“cap\_leak”** the root access does not go away and we are able to run the command “**echo aaaa>& 3”**  by making use of this leaked capability.

To avoid this we must always close the file descriptor as **close(fd)**  to make sure that this does not happen.