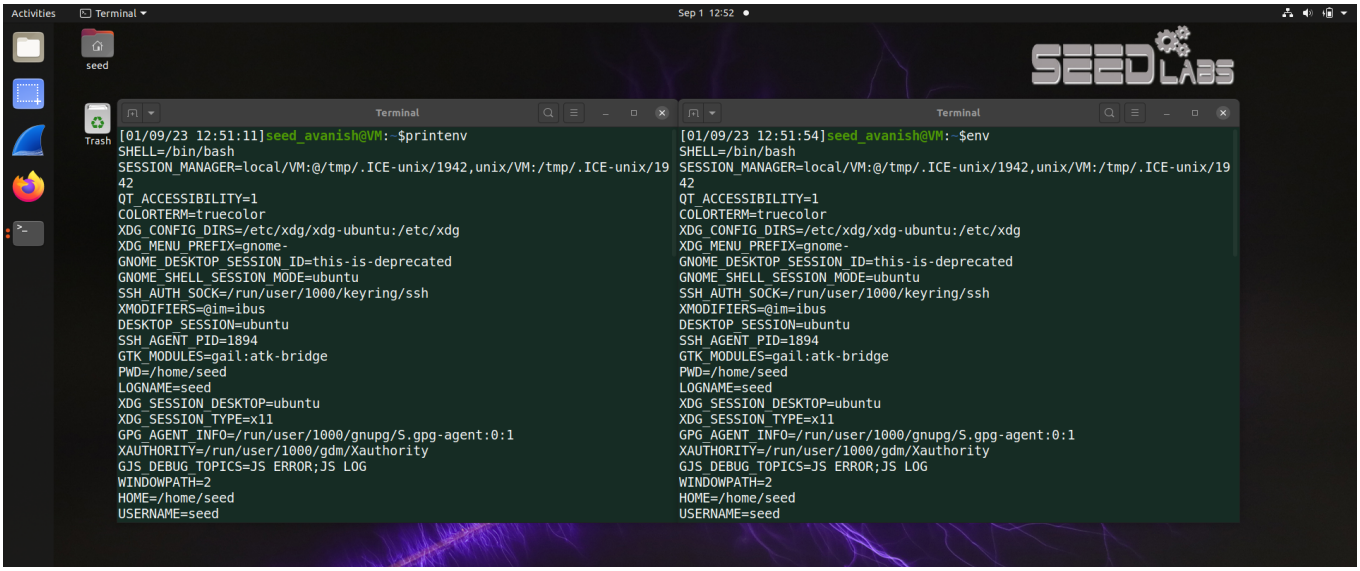
NETWORK SECURITY

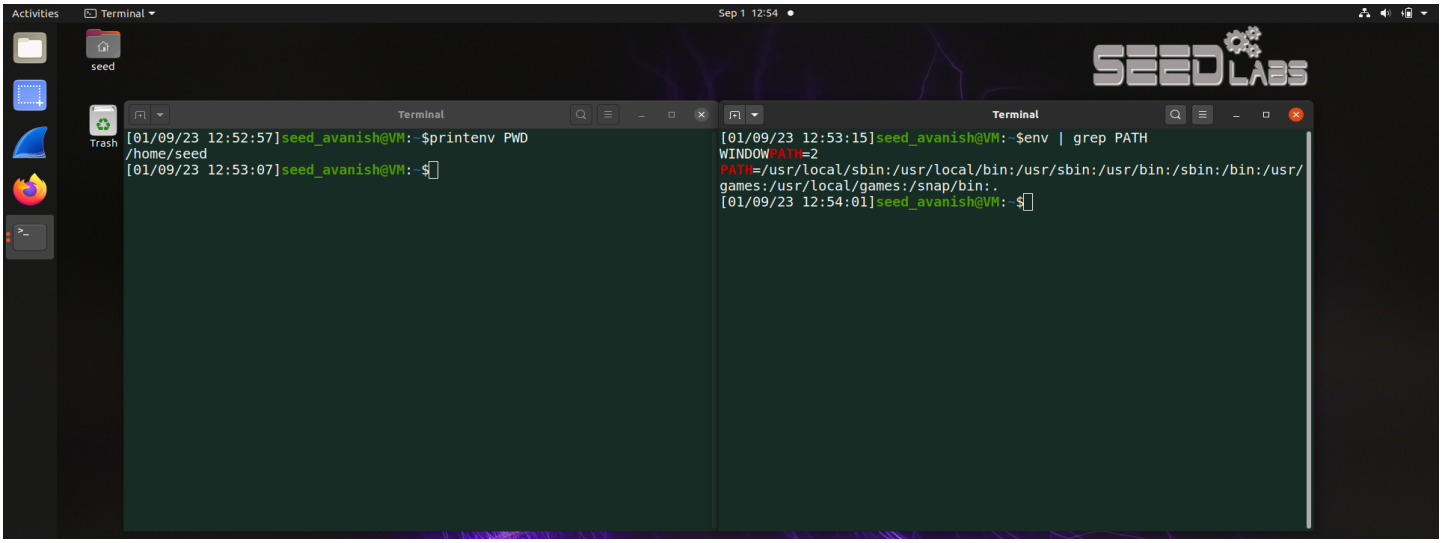
LAB 1: Environment Variable and Set-UID Program Lab

TASK 1: MANIPULATING ENVIRONMENT VARIABLES

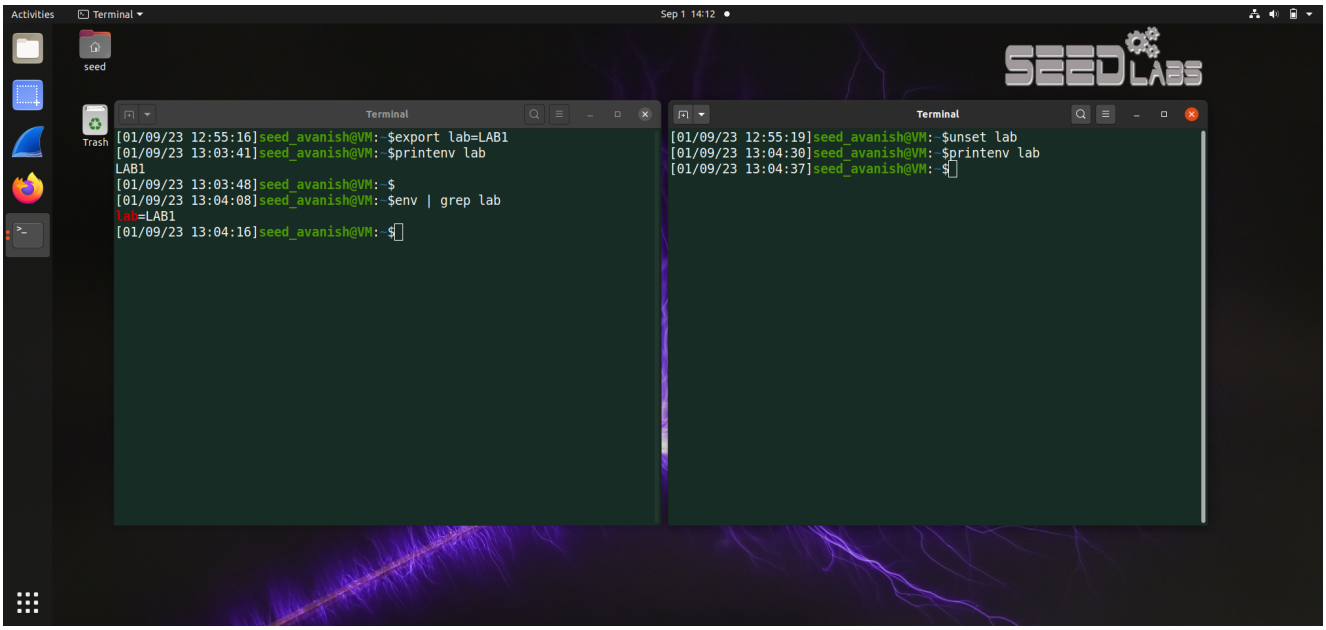


Observations And Explanations:

- To print out all environment variables, I used the **printenv** and **env** commands. Both commands display a list of all environment variables currently set in the Bash session.
- I observed that these commands provide a comprehensive list of environment variables, including system variables and user-defined variables.

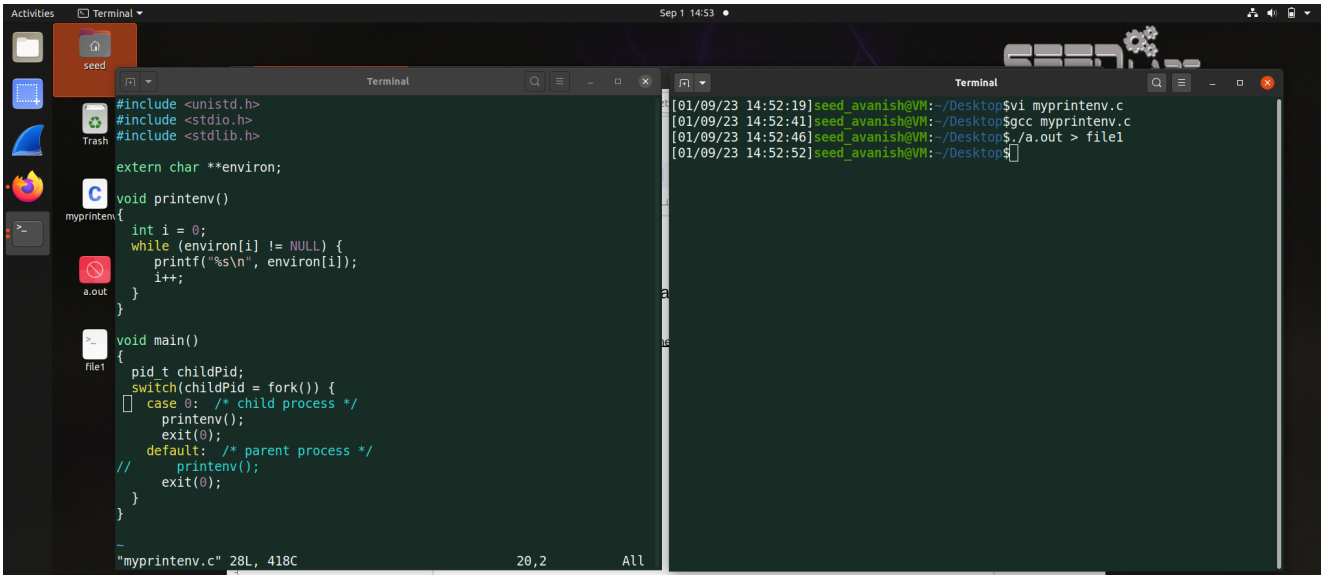


- To print a specific environment variable (e.g., **PWD** - the current working directory),
- These commands allowed me to view the value of a particular environment variable. In this case, the PWD variable represents the current working directory.



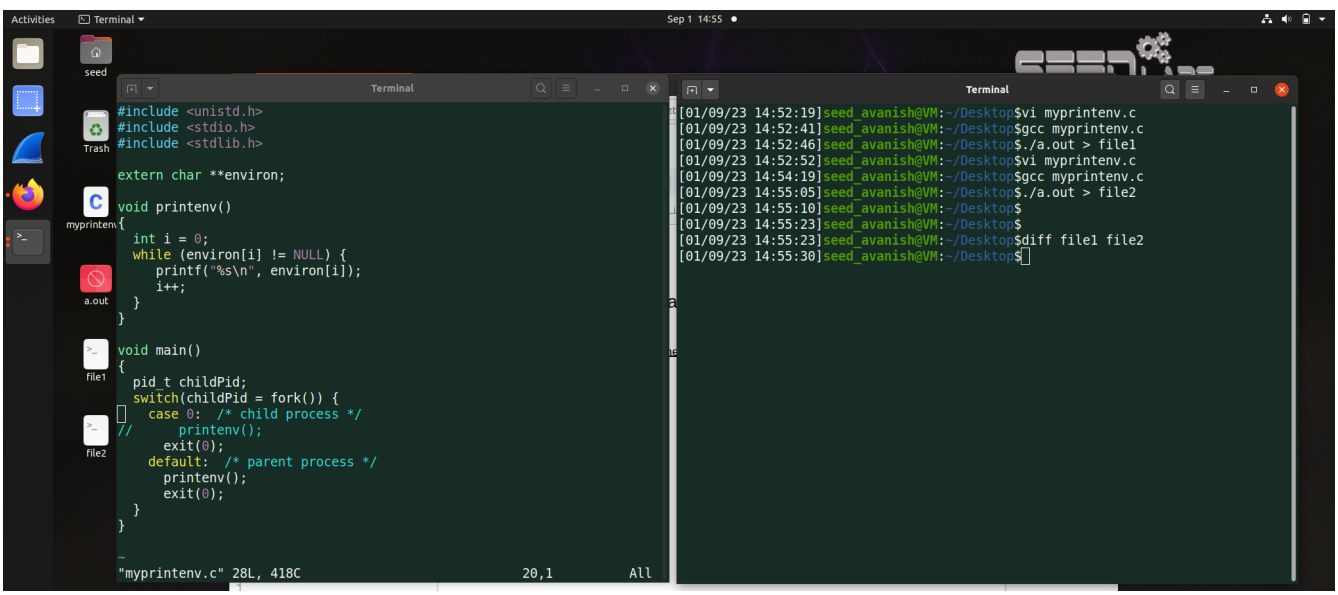
- To set an environment variable, I used the **export** command. For example, to set a variable named lab with the value LAB1 and can be accessed by printenv or env command within the current bash session.
- To **unset** (remove) an environment variable, I used the unset command.

TASK 2: PASSING ENVIRONMENT VARIABLES FROM PARENT PROCESS TO CHILD PROCESS



Observations:

- After running the above program the output saved in file1 contained the environment variables of the child process.

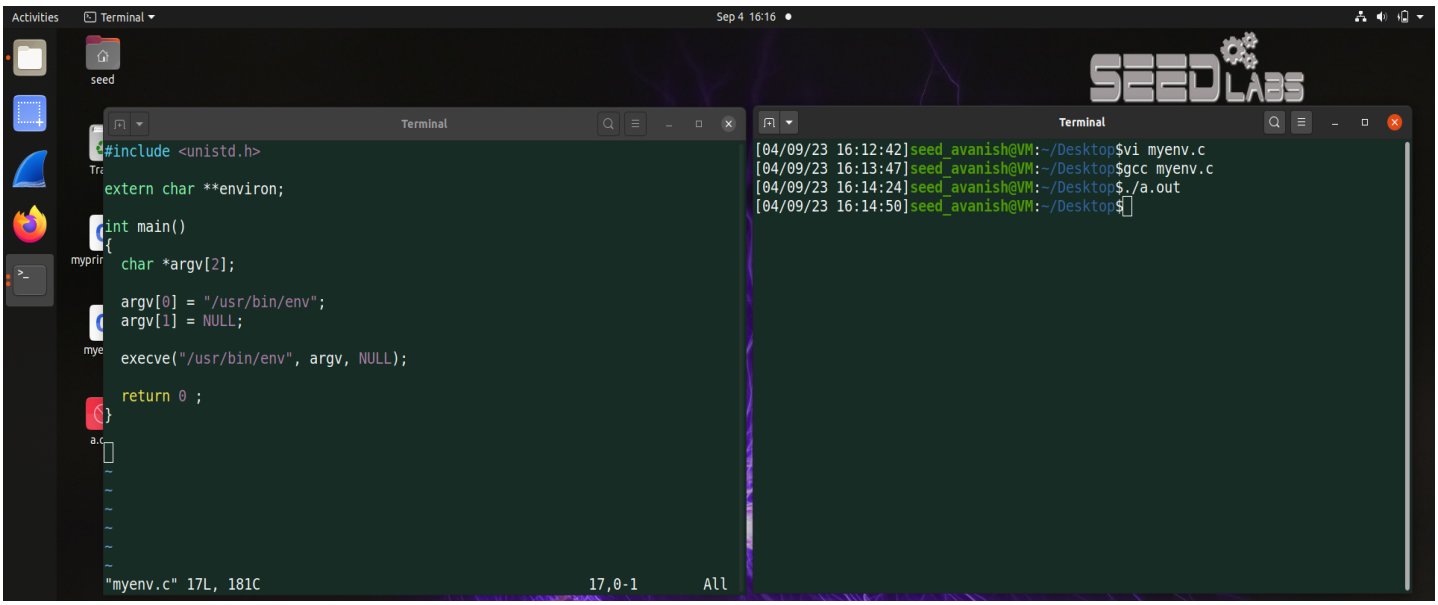


- After modifying**(commenting the printenv in child process and uncommenting the printenv in parent process)** the above program the output saved in file2 contained the environment variables of the parent process.
- The **diff** command showed that the two output files were different, specifically indicating differences in the lines related to environment variables.

Explanation:

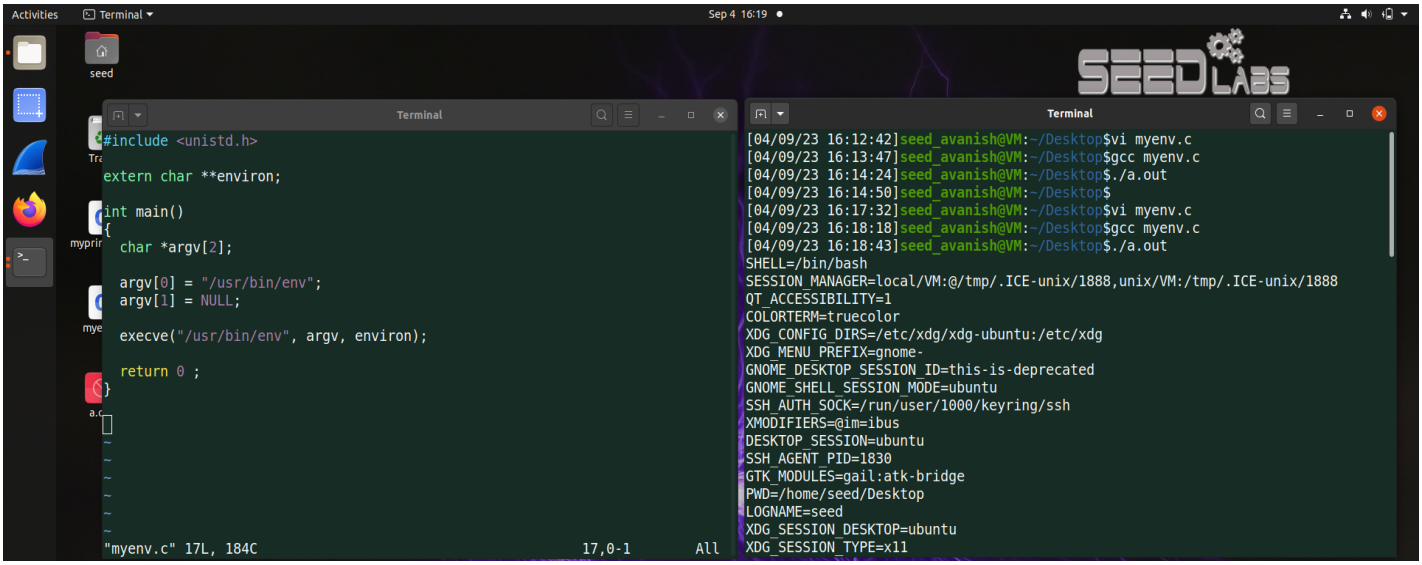
- The fork() system call in Unix creates a new process (the child) that is an exact duplicate of the calling process (the parent). This duplication includes the environment variables.
- we can conclude that by default, in Unix-based systems, environment variables are inherited by child processes from their parent processes when using the fork() system call.

Task 3: Environment Variables and execve()



Observations:

- Compiled and executed the myenv.c program, which contains a call to “execve("/usr/bin/env", argv,NULL);.” .
- As we saw from the task, if no environment variables are passed through the call, the program will not have access to them and there will be no output.

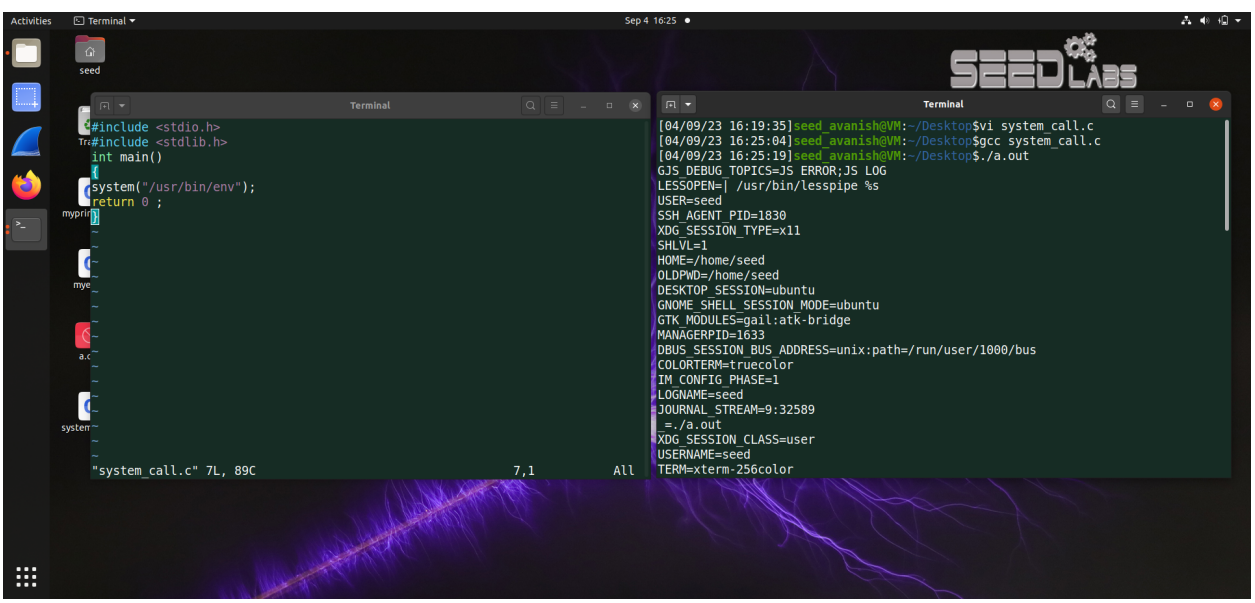


- Modified the execve() invocation to execve("/usr/bin/env", argv, environ);.
- The output displayed is the list of all environment variables from the parent process.

Explanations:

- when a new program is executed using the system() function, the environment variables of the calling process are indeed passed to the new program. This behavior aligns with the fact that system() internally uses execl() to execute /bin/sh, which, in turn, uses execve() and passes the environment variables array. Therefore, environment variables are effectively inherited by the new program when executed using system().

Task 4: Environment Variables and system()



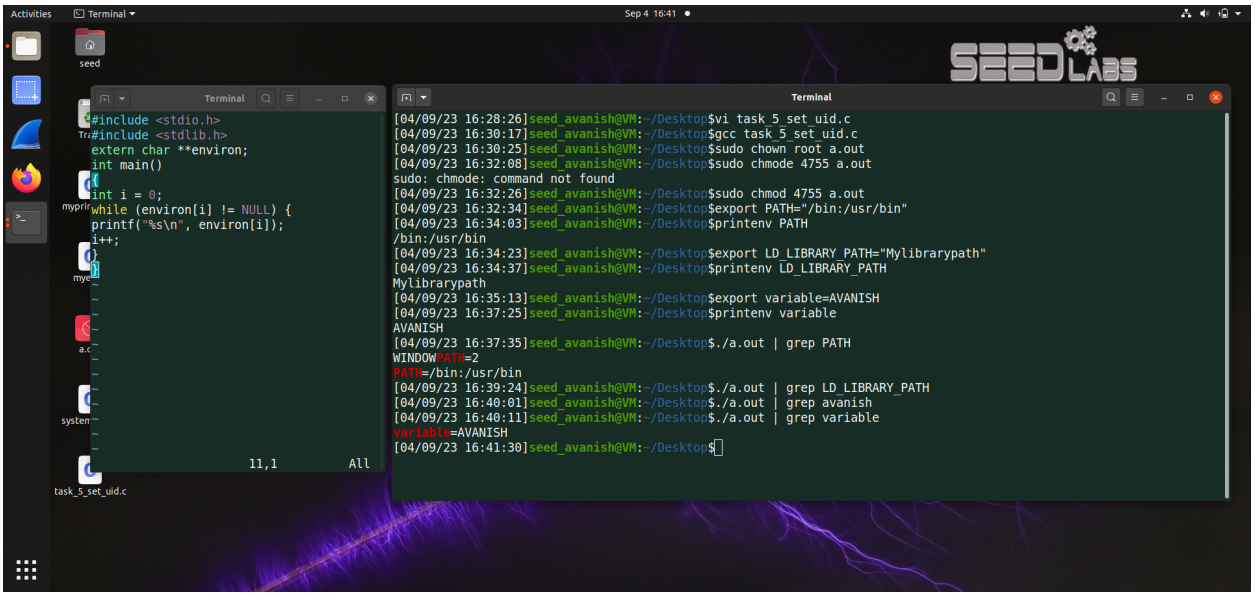
Observations:

- Compiled and executed the following program mysys.c:
- When running the program, the output displayed a list of environment variables. These environment variables were related to the calling process

Explanation:

- The system() function, when used to execute a command, invokes /bin/sh and asks it to execute the specified command.
- /bin/sh inherits the environment variables of the calling process, and this environment is passed on to the command it executes (/usr/bin/env in this case).
- As a result, the /usr/bin/env program inherited the environment variables of the calling process when executed via the system() function.

Task 5: Environment Variable and Set-UID Programs



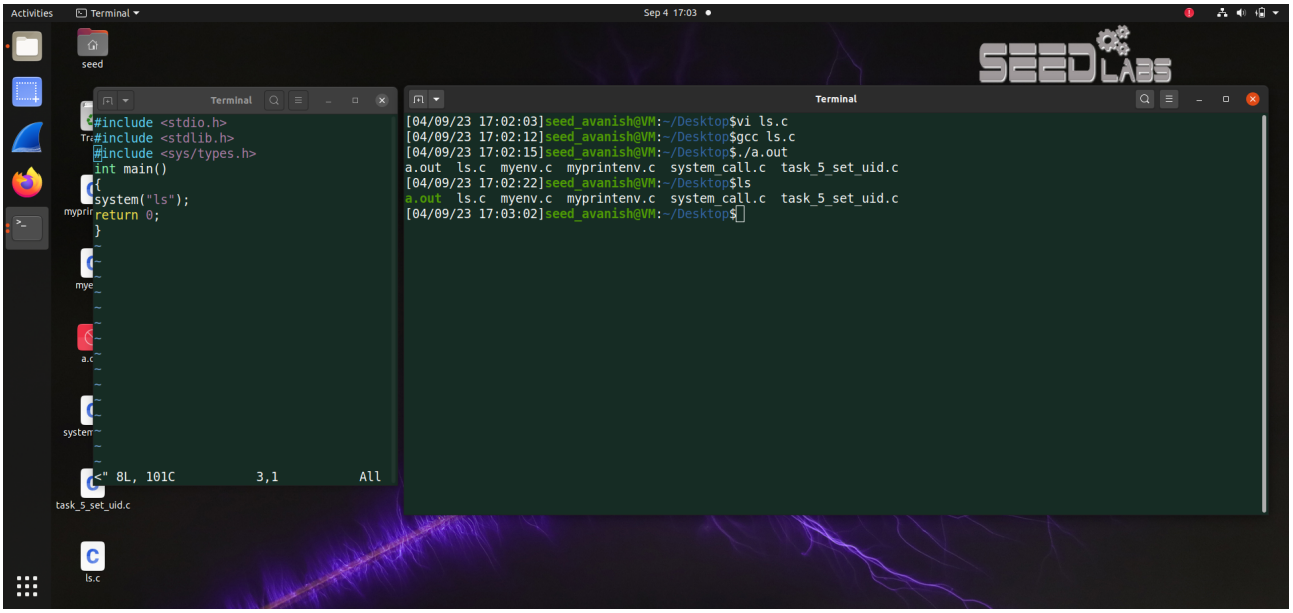
- Compiled a program that prints all the environment variables in the current process
- Changed its ownership to root, making it a Set-UID program.
- Set some environment variables (PATH, LD_LIBRARY_PATH, and ANY_NAME) in the user's shell process. Then, run the Set-UID program and observe whether these environment variables are inherited by the Set-UID child process.

Observations and Explanations:

- By default, Set-UID programs inherit the environment of the parent process (in this case, the user's shell).
- The export command in the user's shell process modified the environment variables in the shell's environment, and these changes were passed down to the Set-UID child process when the program was executed.

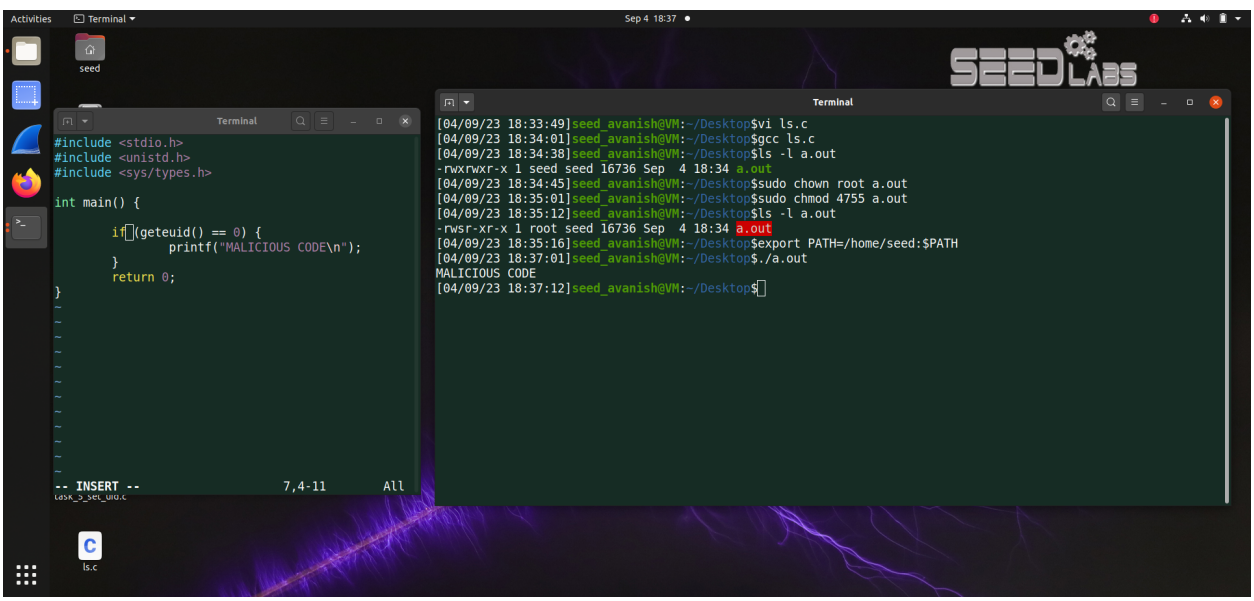
NOTE: Surprisingly, however, the variable LD_LIBRARY_PATH doesn't seem to have been passed.

Task 6: The PATH Environment Variable and Set-UID Programs



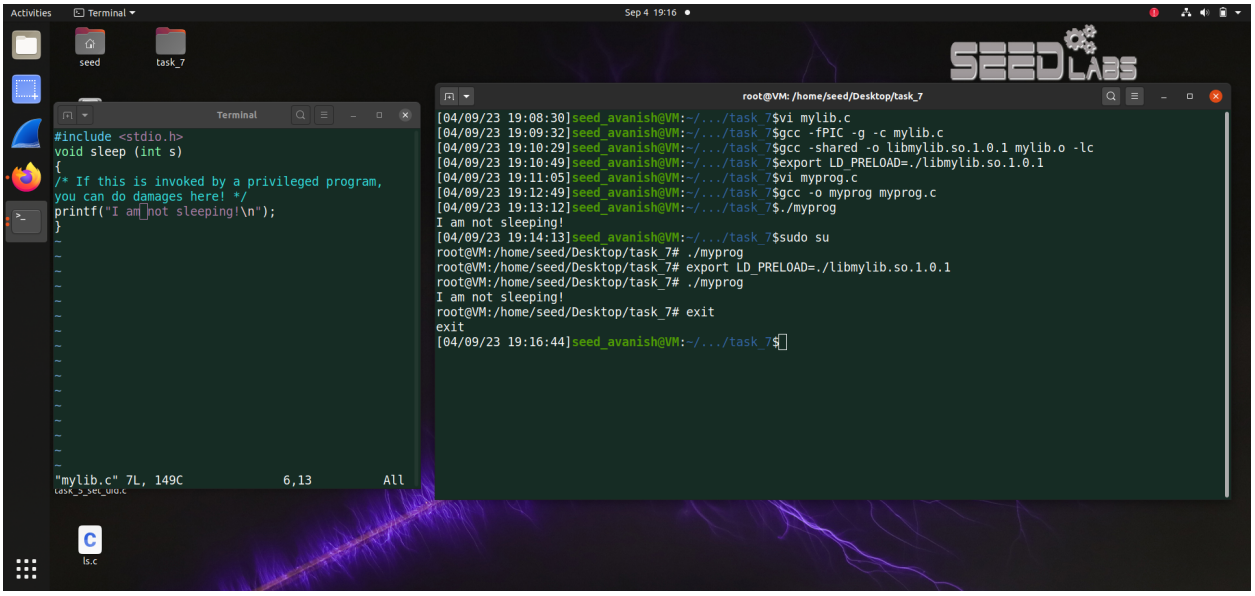
Observations and Explanations:

- This program is intended to execute the /bin/ls command but uses a relative path for ls.
- compile the program, change its owner to root, and make it a Set-UID program



- Now, we created a malicious program named ‘ls.c’ in the home/seed directory and added that directory to the PATH directory.
- Compiled the above program, changed its owner to root, and made it a Set-UID program.
- By checking the effective user id (by calling geteuid()), we found out that our “ls” was running with the root privilege.

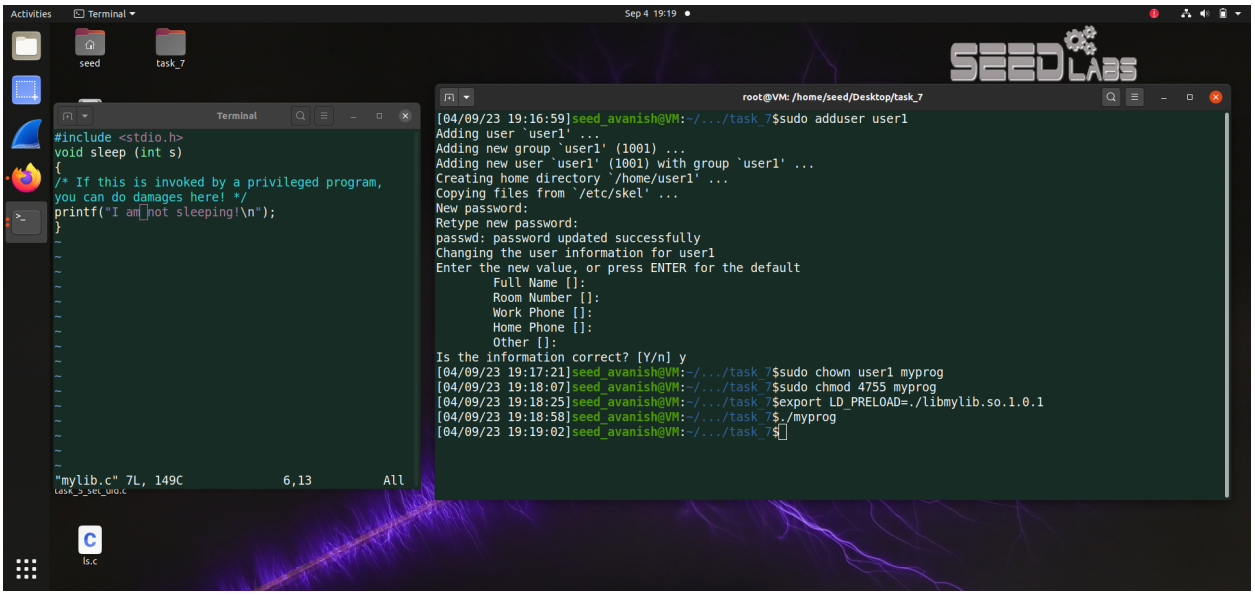
Task 7: The LD_PRELOAD Environment Variable and Set-UID Programs



- Created the mylib.c program, compiled and created the shared library
- LD_PRELOAD environment variable is set to point to the created library.

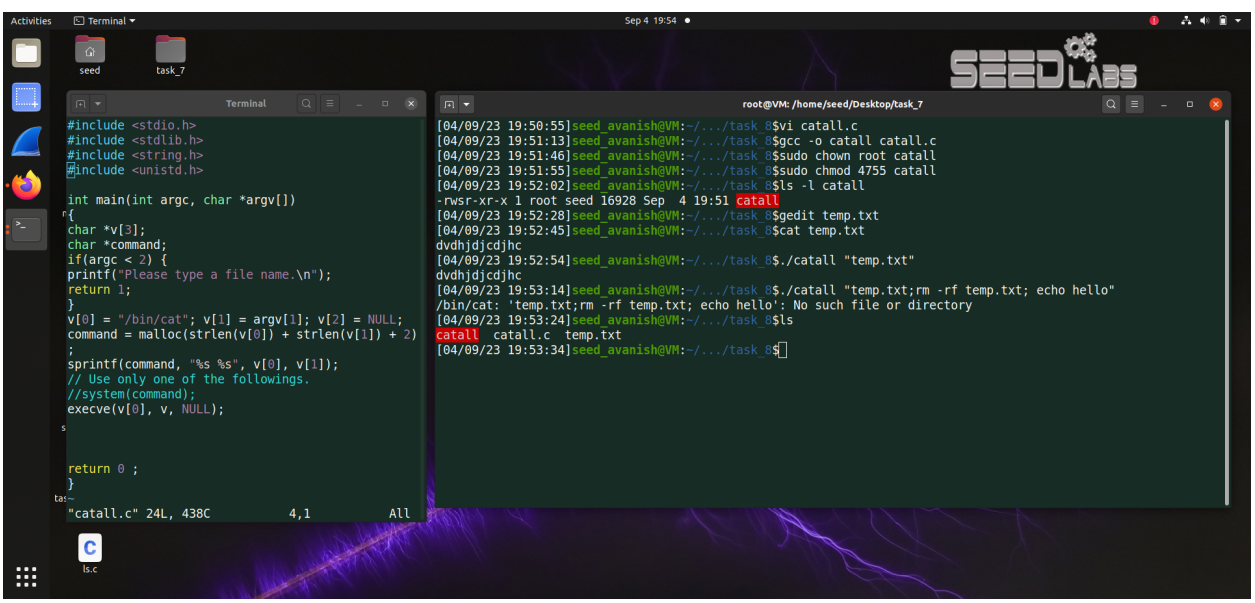
Observations And Explanations:

- **When running myprog as a regular program by a normal user**: The LD_PRELOAD library is used to override the sleep() function, so the program prints "I am not sleeping!" instead of sleeping for 1 second.
- **When making myprog a Set-UID root program and running it as a normal user**:
  - The program executes with the privileges of the root user.
  - However, the LD_PRELOAD library and environment variable are not inherited because the dynamic loader/linker (ld.so or ld-linux.so) does not allow the LD_PRELOAD variable to be inherited in Set-UID programs by default. This is a security measure to prevent privilege escalation attacks.
- **When making myprog a Set-UID root program, exporting the LD_PRELOAD environment variable again in the root account, and running it:**
  - After exporting LD_PRELOAD in the root account, it affects the behavior of myprog. The LD_PRELOAD variable is inherited, the program prints "I am not sleeping!" instead of sleeping for 1 second.



- **When making myprog a Set-UID user1 program and exporting the LD_PRELOAD environment variable in a different user's account and running it**:
  - The LD_PRELOAD variable is not inherited by the Set-UID program, even when exported by another user. The program runs with the user1's privileges but does not use the LD_PRELOAD library.

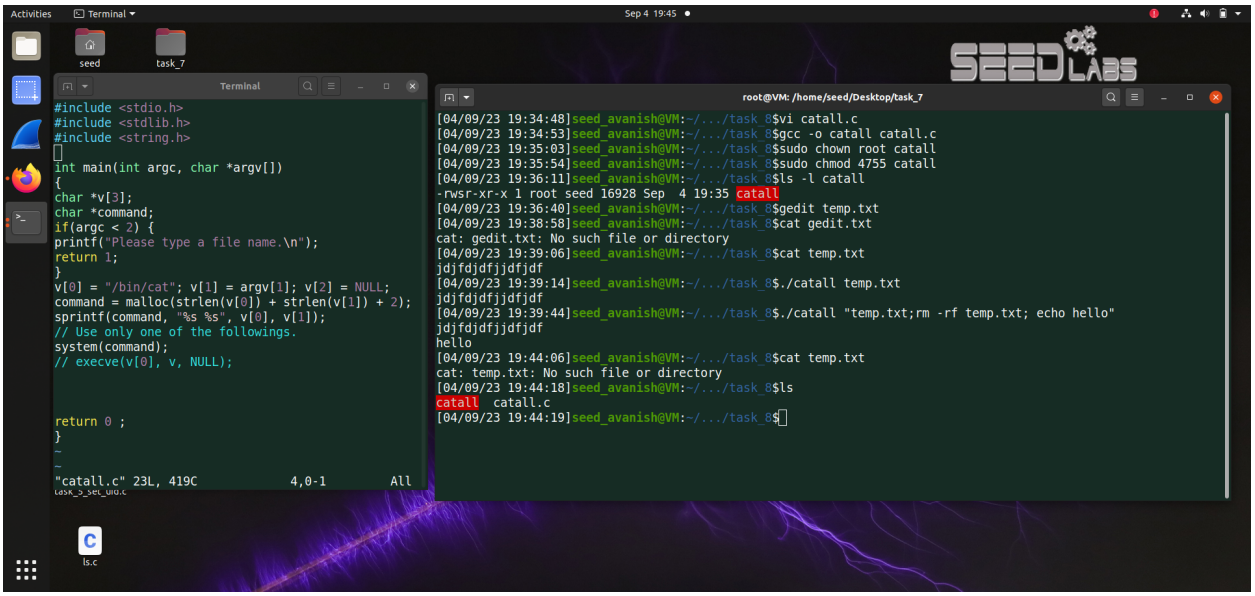
Task 8: Invoking External Programs Using system() versus execve()



Observations:

Using the execve() Function

- The program now uses execve(), it directly executes the specified command without invoking a shell. This significantly reduces the risk of arbitrary command execution.



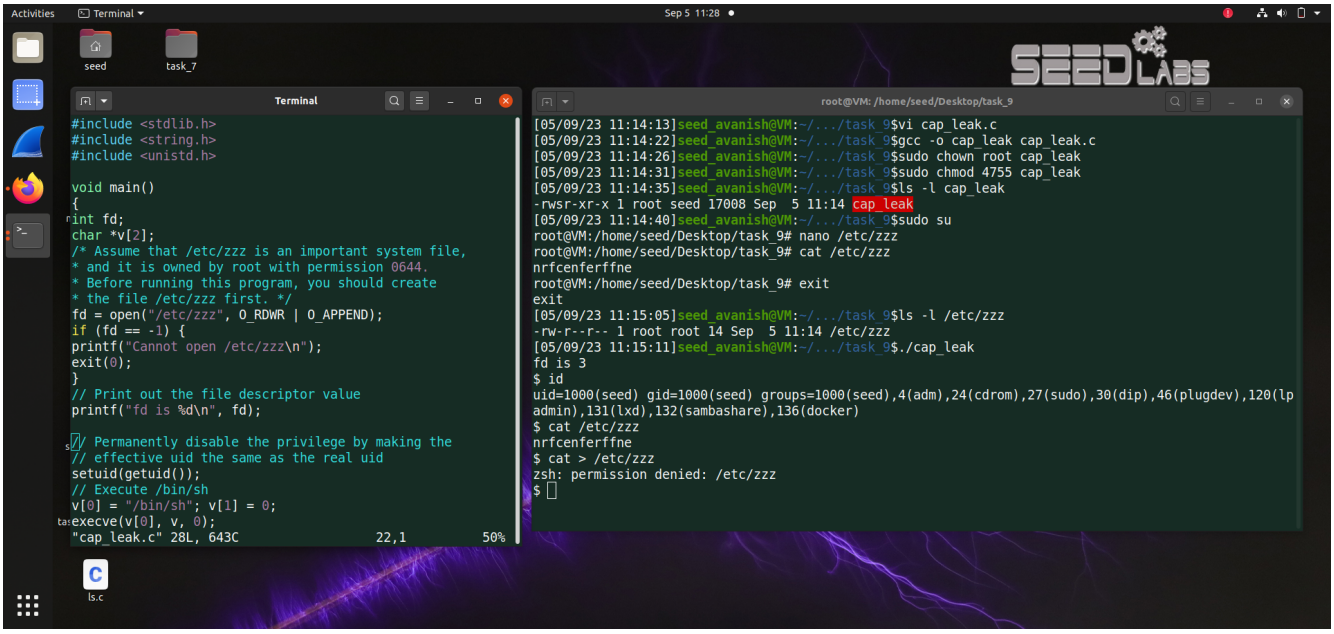
<u>Using the system() function</u>

- The program uses system(), it invokes a shell which can potentially be used to execute arbitrary commands, including those that could compromise system integrity. For example, if you enter a command like ; rm -rf /, it could lead to file deletion, even though the program was initially designed to prevent this.

<u>Explanations</u>:

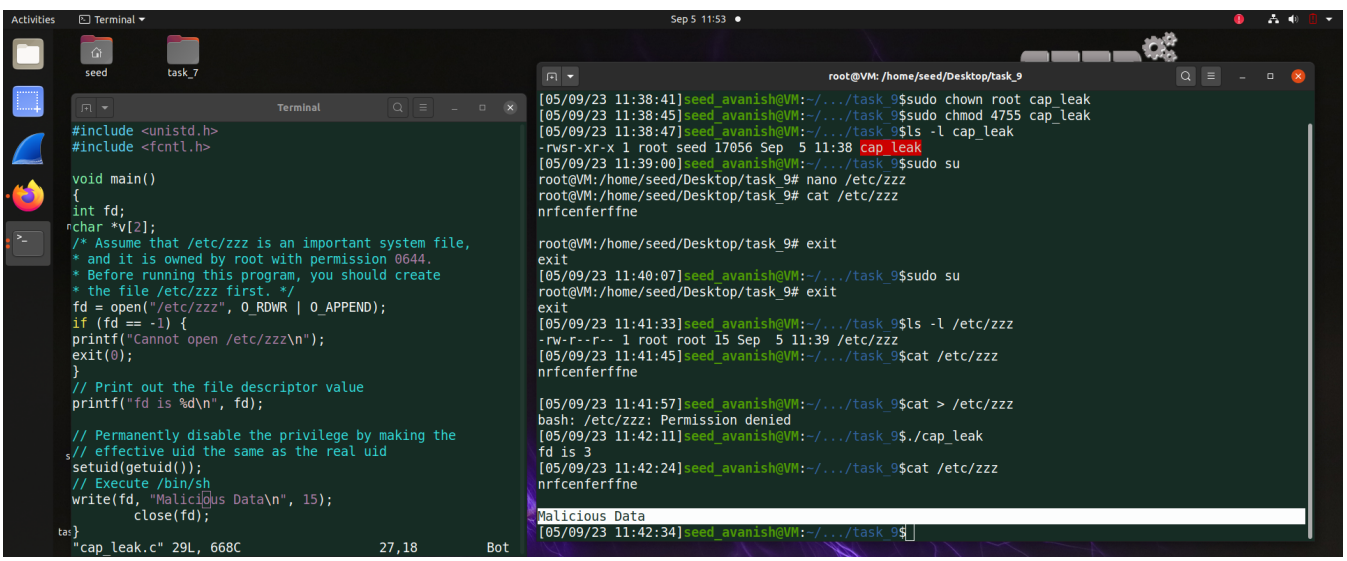
- When the program uses system(), it invokes a shell, allowing the user to potentially execute arbitrary commands, which could compromise system integrity.
- When the program uses execve(), it directly executes the specified command without involving a shell. This approach restricts the user to executing only the specified command and arguments, reducing the risk of arbitrary command execution.

Task 9: Capability Leaking



Observations:;

- The cap_leak program first opens the /etc/zzz file, which is an important system file owned by root with 0644 permissions. It then disables its privilege by using the setuid(getuid()) call, effectively making its effective UID the same as the real UID.
- When normally tried to modify the file it showed ‘permission denied’.



- After modifying the code and adding the write() function in the code.
  By running the command above, I was able to write to the /etc/zzz file, even as a normal user, exploiting the capability leaking vulnerability.

Explanations:

The program's failure to close the file descriptor before downgrading privileges resulted in the capability leaking vulnerability. As a result, even after downgrading privileges, the program still had access to the open file descriptor, allowing it to write to the file.