**21CY682 – Secure Coding lab – I**

**Assignment Topic: Environmental variables**

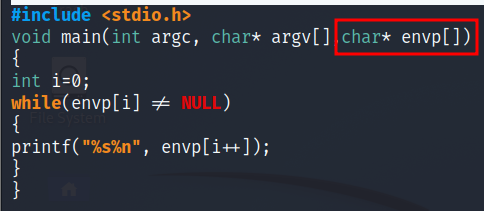
**Register Number: CYS22005**

**Date: 19/11/2022**

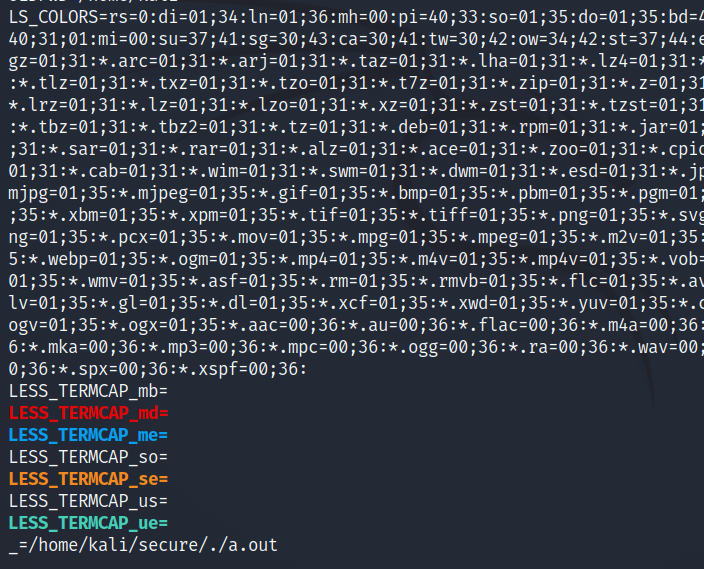
**Name: B.Shebu**

**How to access environmental variables –**

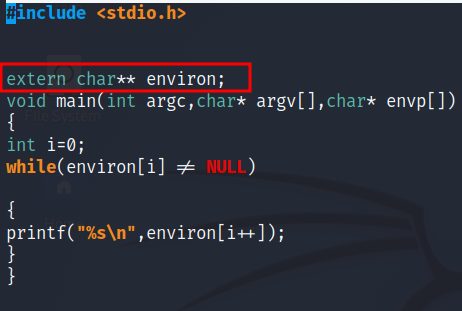
**- From main function**



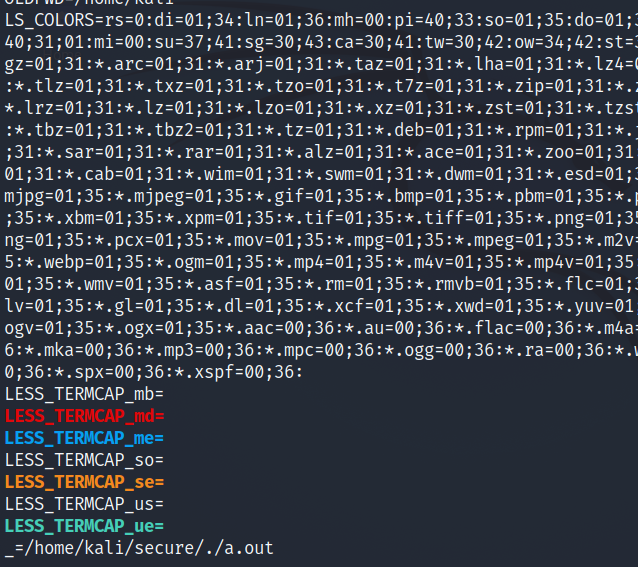
OUTPUT -



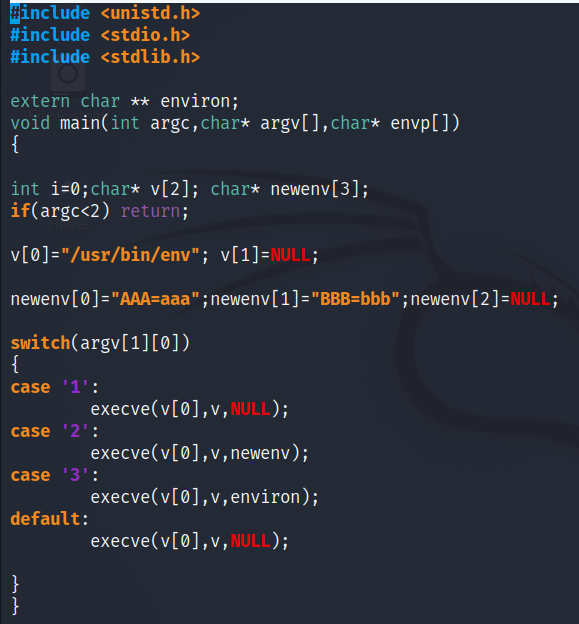
- **More reliable way using global variable**



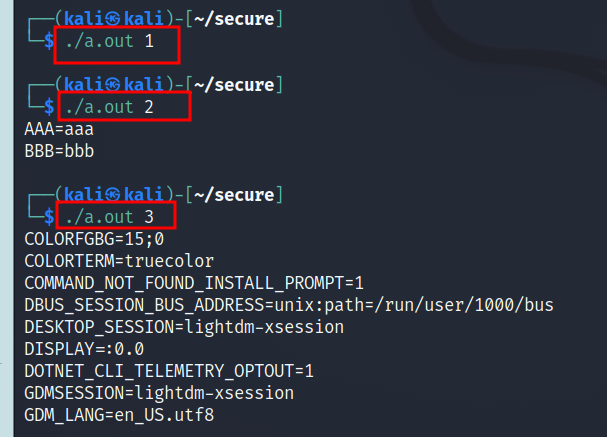
OUTPUT -

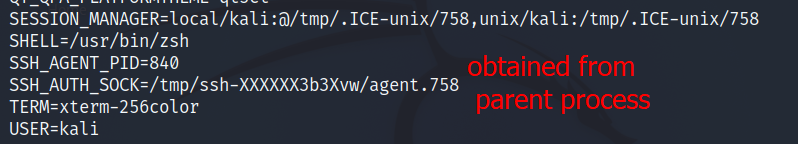


**Execve and environmental variables -**



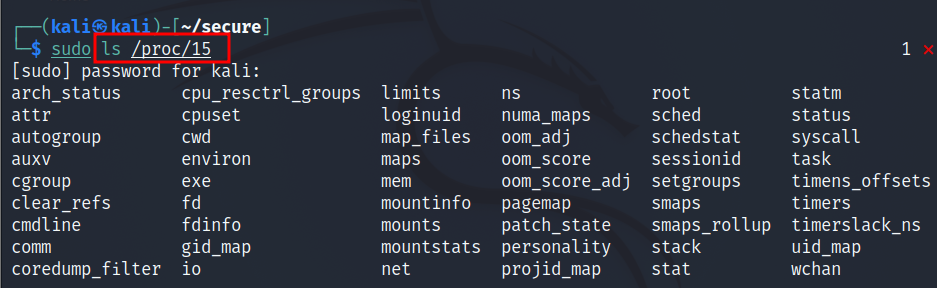
OUTPUT -

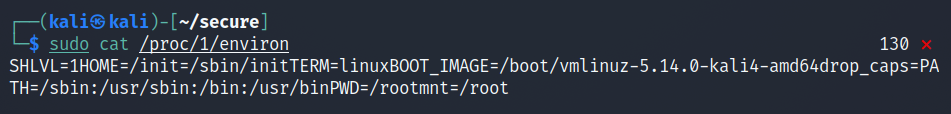




**/PROC file system -**

/proc is a virtual file system . It contains directory for each process, using process id as name of the directory.



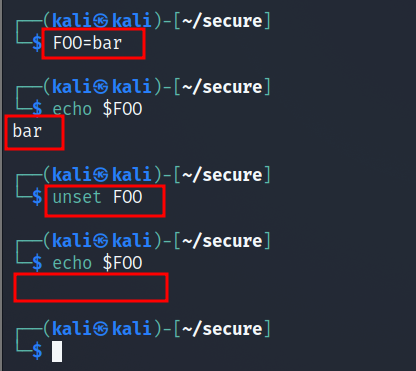


**SHELL & ENVIRONMENTAL VARIABLES –**

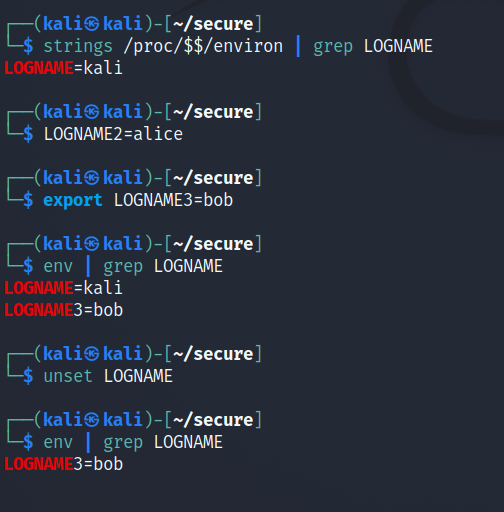
Shell variables are internal variables used by the shell.

Shell provides built-in commands to allow users create , assign and delete shell variables.

EXAMPLE – Here we create a shell variable called FOO.



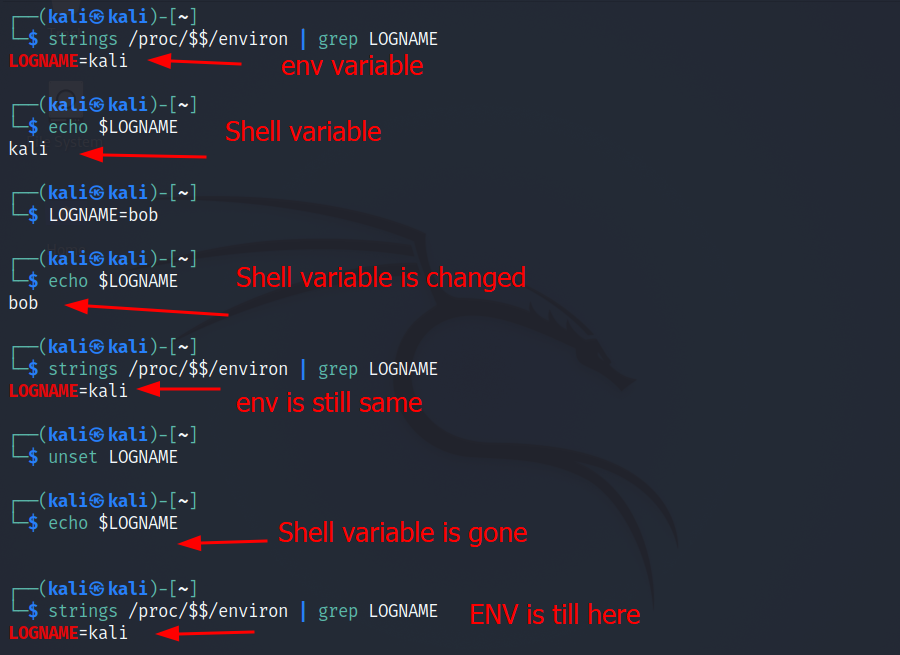
When we type env in shell prompt, it will create a child process.



Shell and environmental variables are different.

When a shell program starts, it copies the environmental variables to its own shell variables.

Changes made to the shell variables will not reflect on the environmental variables.



**ATTACKS VIA DYNAMIC LINKER –**

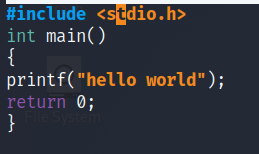
Linking finds external library code referenced in the program.

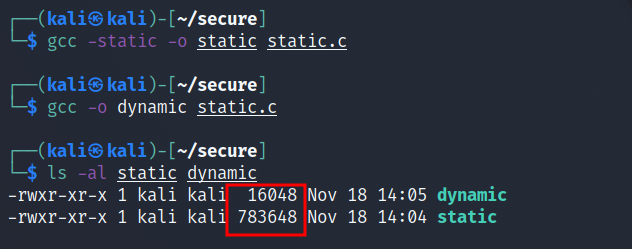
Linking is done during compile time or run time

- Dynamic Linking -> Uses env variables which becomes the attack surface

**EXAMPLE –**

**STATIC LINKING :-**

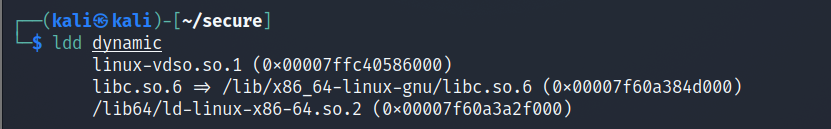




We can see that the size of static compiled program is 100 times larger than a dynamic program

**DYNAMIC LINKING :-**

We can use “ldd” command to see what shared libraries a program depends on



Here,

linux-vdso.so.1 -> for system calls

libc.so.6 -> The libc library contains functions like printf() and sleep().

/lib64/ld-linux-x86-=64.so.2 -> here the dynamic linker itself is in a shared library.It is invoked before the main function gets invoked .

**ATTACKS VIA DYNAMIC LINKER CASE STUDY -1**

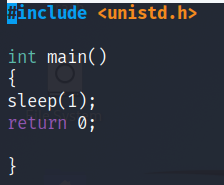
- LD\_PRELOAD contains a list of shared libraries which will be searched first by every linker.

- If all functions are not found the linker will search among several lists of folder including the one specified by LD\_LIBRARY\_PATH.

- If that programs were SET-UID programs , it may lead to security breaches.

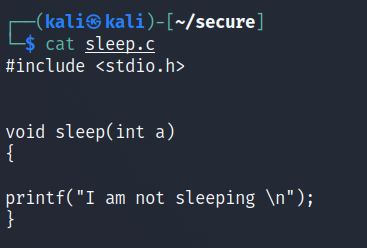
**EXAMPLE 1 –NORMAL PROGRAMS**

Program calls sleep function which is dynamically linked.

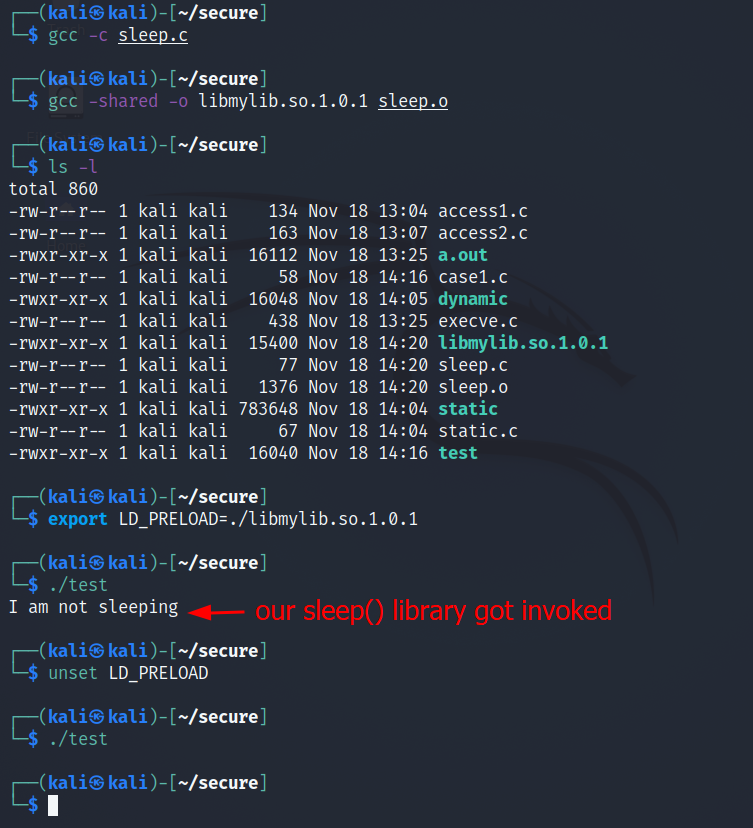


OUTPUTS -



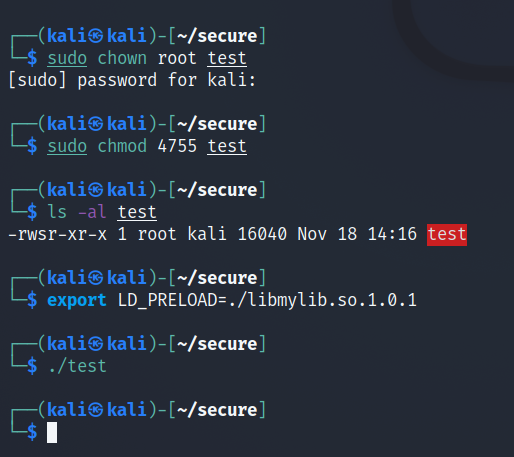


Now we implement our own sleep(). First we need to compile the above code, create a shared library and add the shared library to LD\_PRELOAD env variable.



**EXAMPLE 2- SET-UID Programs**

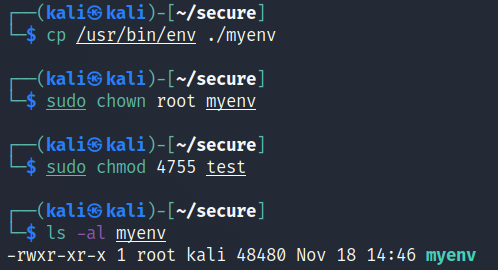
- If the technique in example 1 works for set-uid program ,it can be very dangerous. Now lets convert the above example code to SETUID.

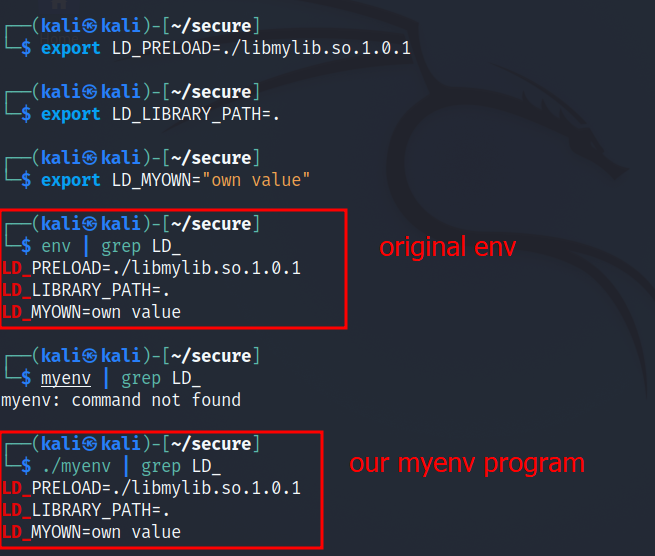


Here our sleep() was not invoked.. This is because there is a countermeasure implemented by dynamic linker. It ignores the LD\_PRELOAD and LD\_LIBRARY\_PATH env variables when euid and ruid differ.

LETS VERiFY THIS COUNTERMEASURE

- Make a copy of env program and make it setuid program.





**ATTACKS VIA DYNAMIC LINKER : CASE STUDY 2 –**

**CASE STUDY : os x dynamic linker**

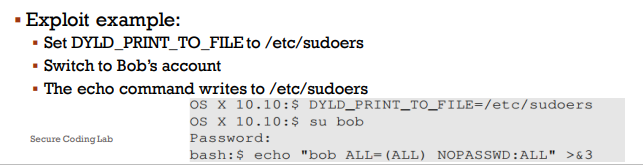
Apple OS X 10.10 introduced a new environment variable without analyzing its security implications perfectly.

▪ DYLD\_PRINT\_TO\_FILE

▪ Ability for users to supply filename for dyld

▪ If it is a Set-UID program, users can write to a protected file

▪ Capability leak – file descriptor not closed



**ATTACKS VIA EXTERNAL PROGRAMS -**

An application may invoke an external program.

▪ The application itself may not use environment variables, but the invoked external program might.

▪ Typical ways of invoking external programs:

- exec() family of function which call execve(): runs the program directly

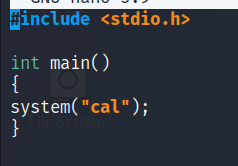
- system() calls /bin/sh to run .

**ATTACKS VIA EXTERNAL PROGRAMS : CASE STUDY**

Shell programs behavior is affected by many environment variables, the most common of which is the PATH variable.

▪ When a shell program runs a command and the absolute path is not provided, it uses the PATH variable to locate the command.

▪ Consider the following code:

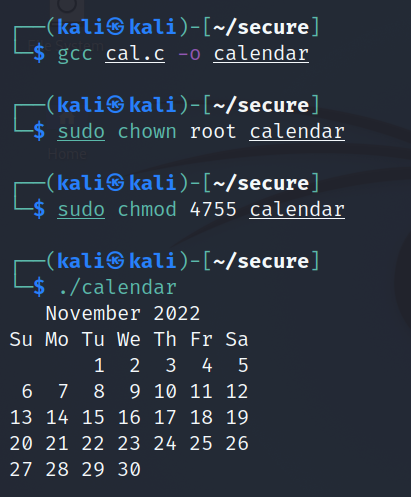


Here in the above example, full path is not provided so we can use this to manipulate the path variable.

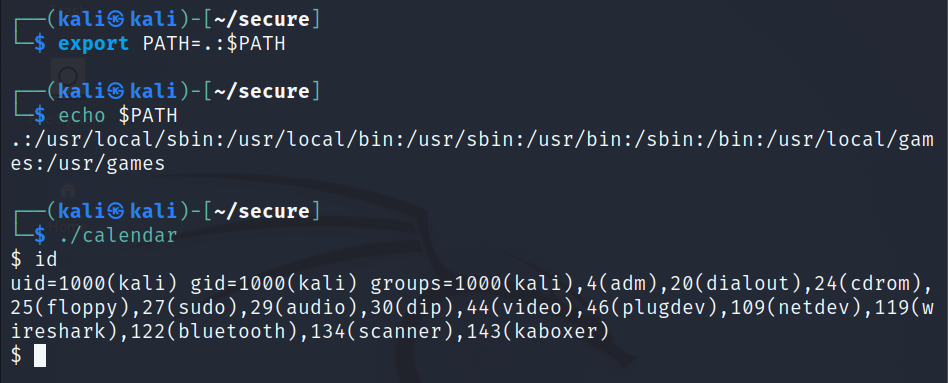
We will trick the above program to execute the below program.



First we will run the program without doing the attack.



Now change the PATH environment variable.



**ATTACKS VIA APPLICATION CODE –**

Programs often use functions from external libraries. If these functions use environment variables, they add to the attack surface.

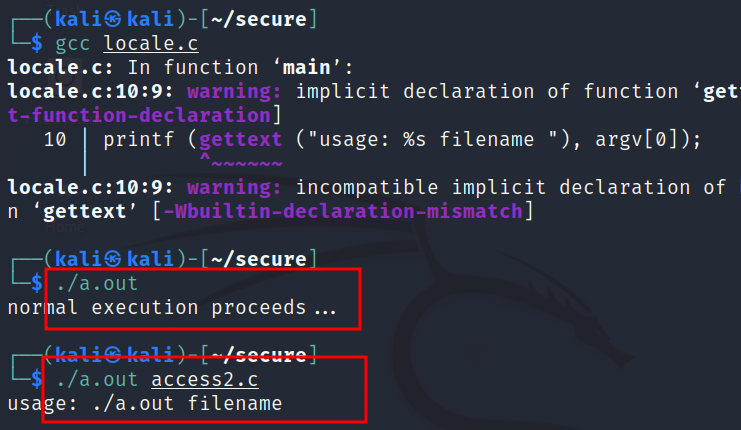
▪ Every time a message needs to be printed out, the program uses the provided library functions for the translated message ▪

Unix uses the gettext() and catopen() in the libc library ▪

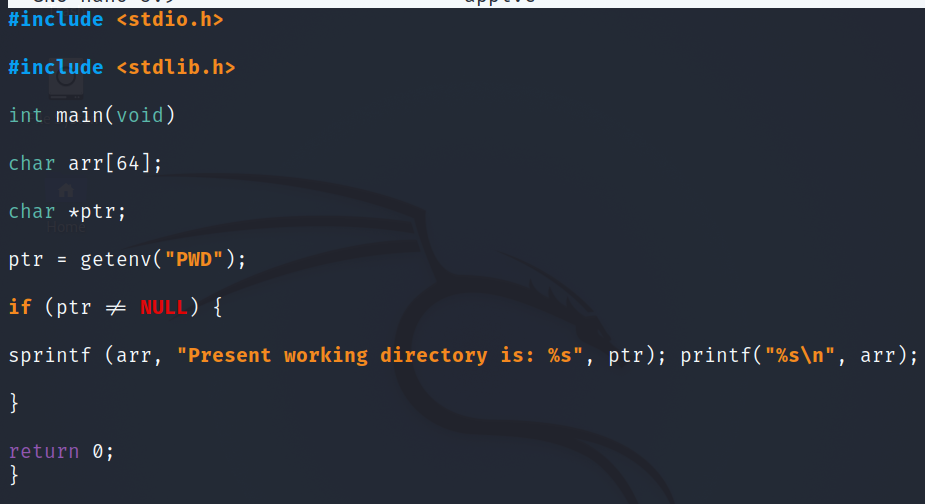
The following code shows how a program can use locale subsystem :



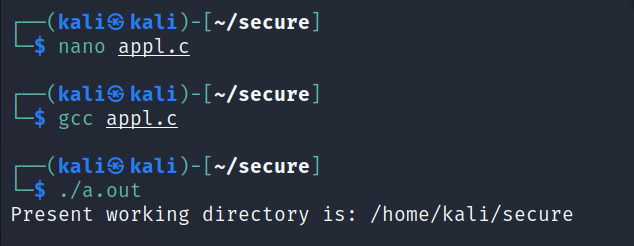
OUTPUT -



**ATTACKS VIA APPLICATION CODE –**



Output -



Programs may directly use environmental variables. If these are priviledged programs, it may result in untrusted inputs.

**EXAMPLE 2 –**

▪ The program uses getenv() to know its current directory from the PWD environment variable

▪ The program then copies this into an array “arr”, but forgets to check the length of the input. This results in a potential buffer overflow.

▪ Value of PWD comes from the shell program, so every time we change our folder the shell program updates its shell variable.

▪ We can change the shell variable ourselves.

