Answer 1: Though Anjali doesn't use environment variables in her code, the environmental variables LD_PRELOAD and LD_LIBRARAY_PATH are used while compiling a program with dynamic linking. In dynamic linking, a part of the program's code is undecided at compilation time. So, the security of the program is at risk if the user can influence the missing code.

Consider the following example. This program (test.c) is dynamically linked and invokes the sleep function.

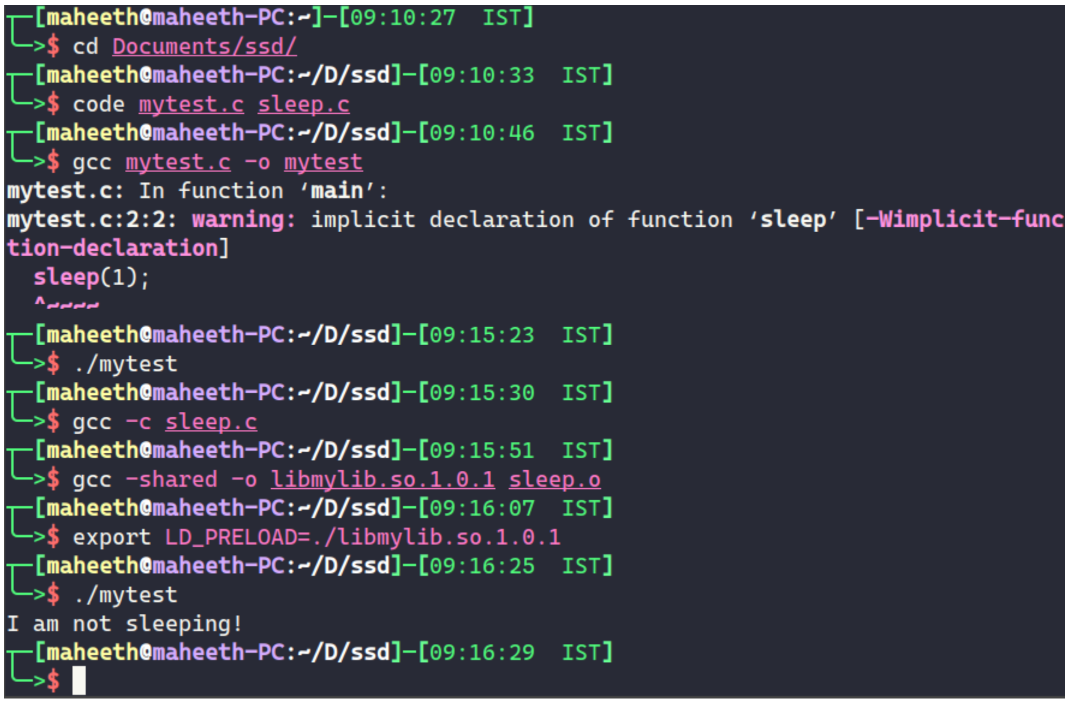
```
int main() {
    sleep(1);
    return 0;
}
```

I have written a custom sleep function below (sleep.c).

```
#include <stdio.h>

void sleep(int s) {
    printf("I am not sleeping!\n");
}
```

Now, I will create a library out of this sleep.c file and put the path to that library in the LD_PRELOAD environmental variable. So, when test.c is executed, our custom sleep() function is executed. This is shown in the screenshot here.



```
┌─[maheeth@maheeth-PC:~]─[09:10:27  IST]
└─>$ cd Documents/ssd/
┌─[maheeth@maheeth-PC:~/D/ssd]─[09:10:33  IST]
└─>$ code mytest.c sleep.c
┌─[maheeth@maheeth-PC:~/D/ssd]─[09:10:46  IST]
└─>$ gcc mytest.c -o mytest
mytest.c: In function ‘main’:
mytest.c:2:2: warning: implicit declaration of function ‘sleep’ [-Wimplicit-func
tion-declaration]
  sleep(1);
  ^~~~~
┌─[maheeth@maheeth-PC:~/D/ssd]─[09:15:23  IST]
└─>$ ./mytest
┌─[maheeth@maheeth-PC:~/D/ssd]─[09:15:30  IST]
└─>$ gcc -c sleep.c
┌─[maheeth@maheeth-PC:~/D/ssd]─[09:15:51  IST]
└─>$ gcc -shared -o libmylib.so.1.0.1 sleep.o
┌─[maheeth@maheeth-PC:~/D/ssd]─[09:16:07  IST]
└─>$ export LD_PRELOAD=./libmylib.so.1.0.1
┌─[maheeth@maheeth-PC:~/D/ssd]─[09:16:25  IST]
└─>$ ./mytest
I am not sleeping!
┌─[maheeth@maheeth-PC:~/D/ssd]─[09:16:29  IST]
└─>$ 
```

Answer 2: In this scenario, the return address of a stack frame is protected from overflow with the stack frame. But an overflow in a buffer from an adjacent stack frame will still overwrite the return pointer and allow for malicious exploitation of the buffer overflow bug. So, this suggestion only changes the method of attack, but doesn't avoid the bug completely.

Answer 3: In part B of the buffer overflow attack, some of the working return address fields could be pointing back to the bof() function, causing some sort of a recursive call to the function itself. So, in the second or third call or multiple calls, the attack succeeds.