Wenbin Li SUID: 687150735

# Return-to-libc Attack Lab Report

## Task 1: Exploiting the Vulnerability

```
Terminal
[10/10/2016 11:14] seed@ubuntu:~/Desktop/lab4$ export MYSHELL=/bin/sh
[10/10/2016 11:15] seed@ubuntu:~/Desktop/lab4$ ./printMyshell
shell addr is bffffe7e
[10/10/2016 11:15] seed@ubuntu:~/Desktop/lab4$
```

Fig 1.1 export MYSHELL value

makefile | exploit.c | retlib.c | printMyshell.c

```
1  /* This program has a buffer overflow vulnerabi
2  /* Our task is to exploit this vulnerability */
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <string.h>
6
7  int bof(FILE *badfile)
8  {
9      char buffer[12];
10     /* The following statement has a buffer ove
11     fread(buffer, sizeof(char), 40, badfile);
12     return 1;
13 }
14
15 int main(int argc, char **argv)
16 {
17     FILE *badfile;
18     badfile = fopen("badfile", "r");
19     bof(badfile);
20     printf("Returned Properly\n");
21     fclose(badfile);
22     return 1;
23 }
24
```

Fig 1.2 retlib.c code

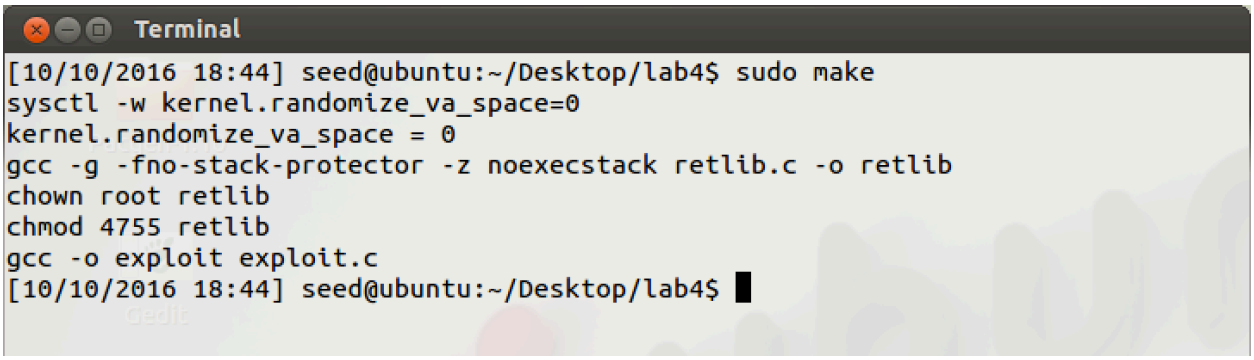



Fig 1.3 turn off addr space off and stack guard and turn on non-exe and make it set-uid program.

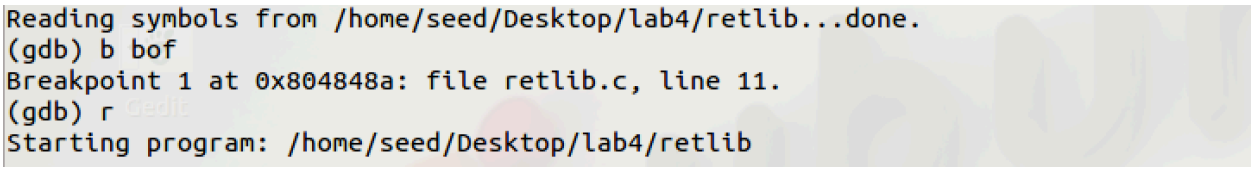



Fig 1.4 start gdb retlib

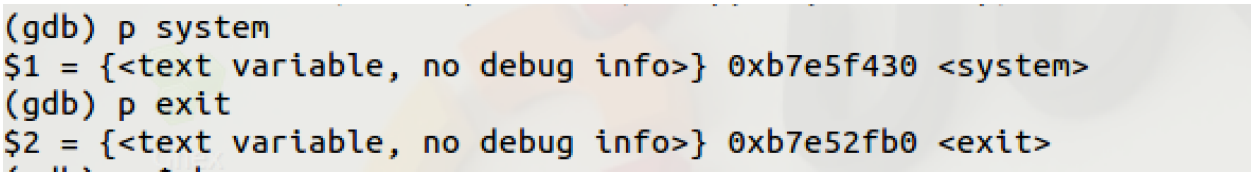



Fig 1.5 get system and exit address

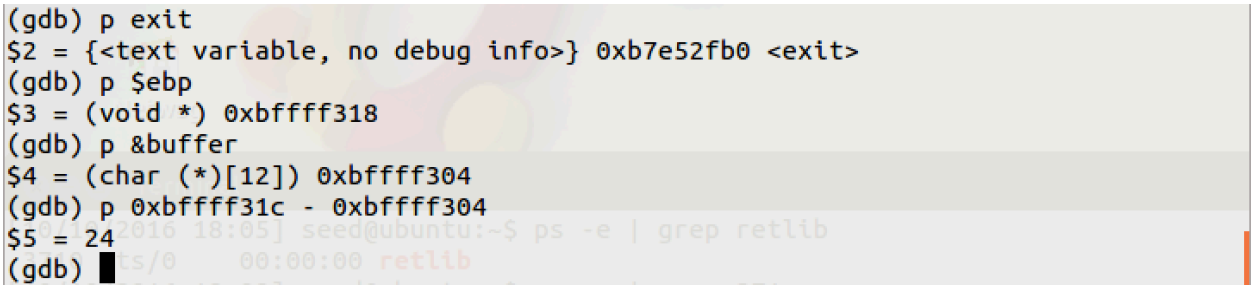



Fig 1.6 distance between buffer and return address

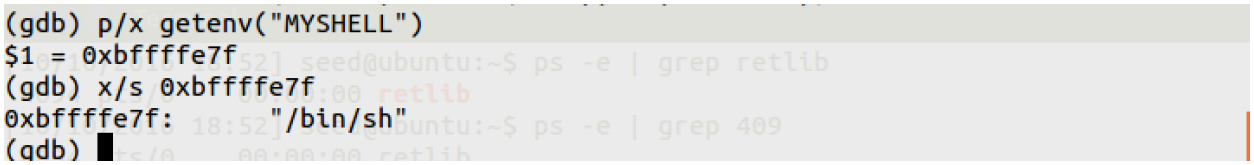



Fig 1.7 address of MYSHELL

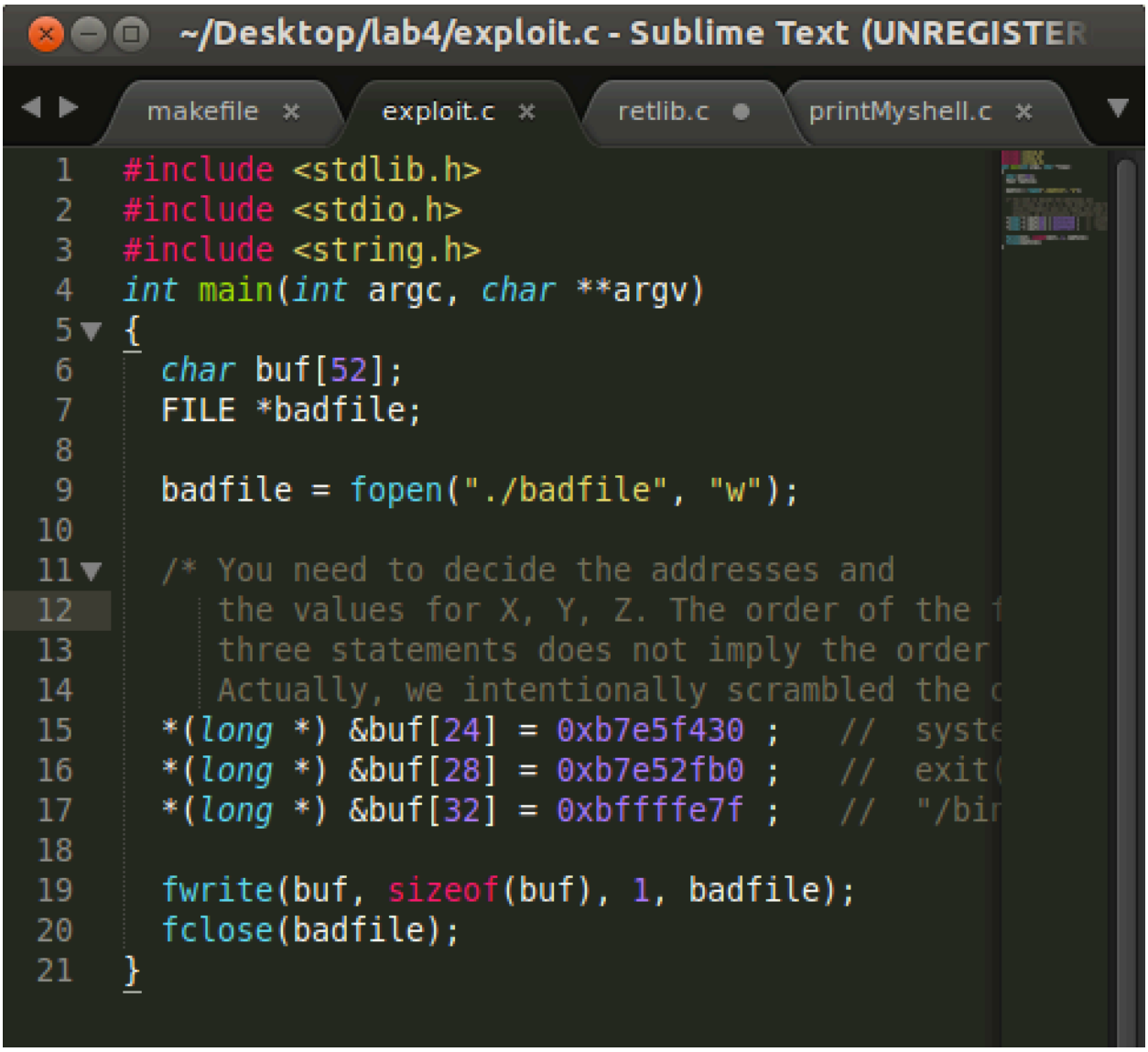



Fig 1.8 exploit.c doe according to result of gdb debug, to construct badfile

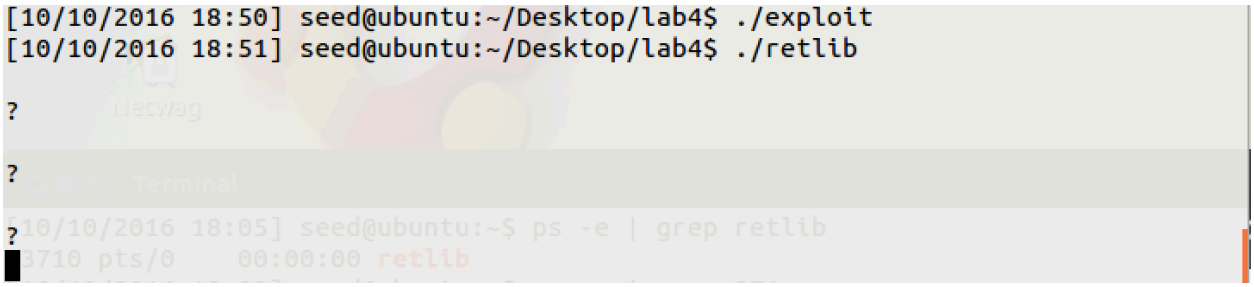



Fig 1.9 result of run attack

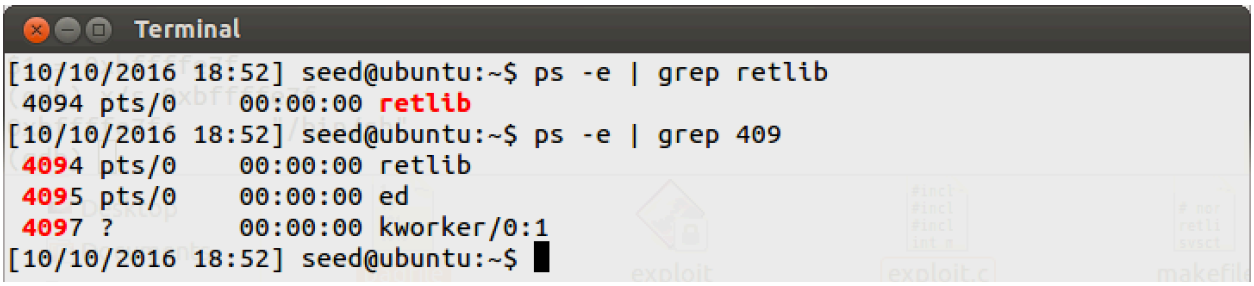



Fig 1.10 grep what is the result.

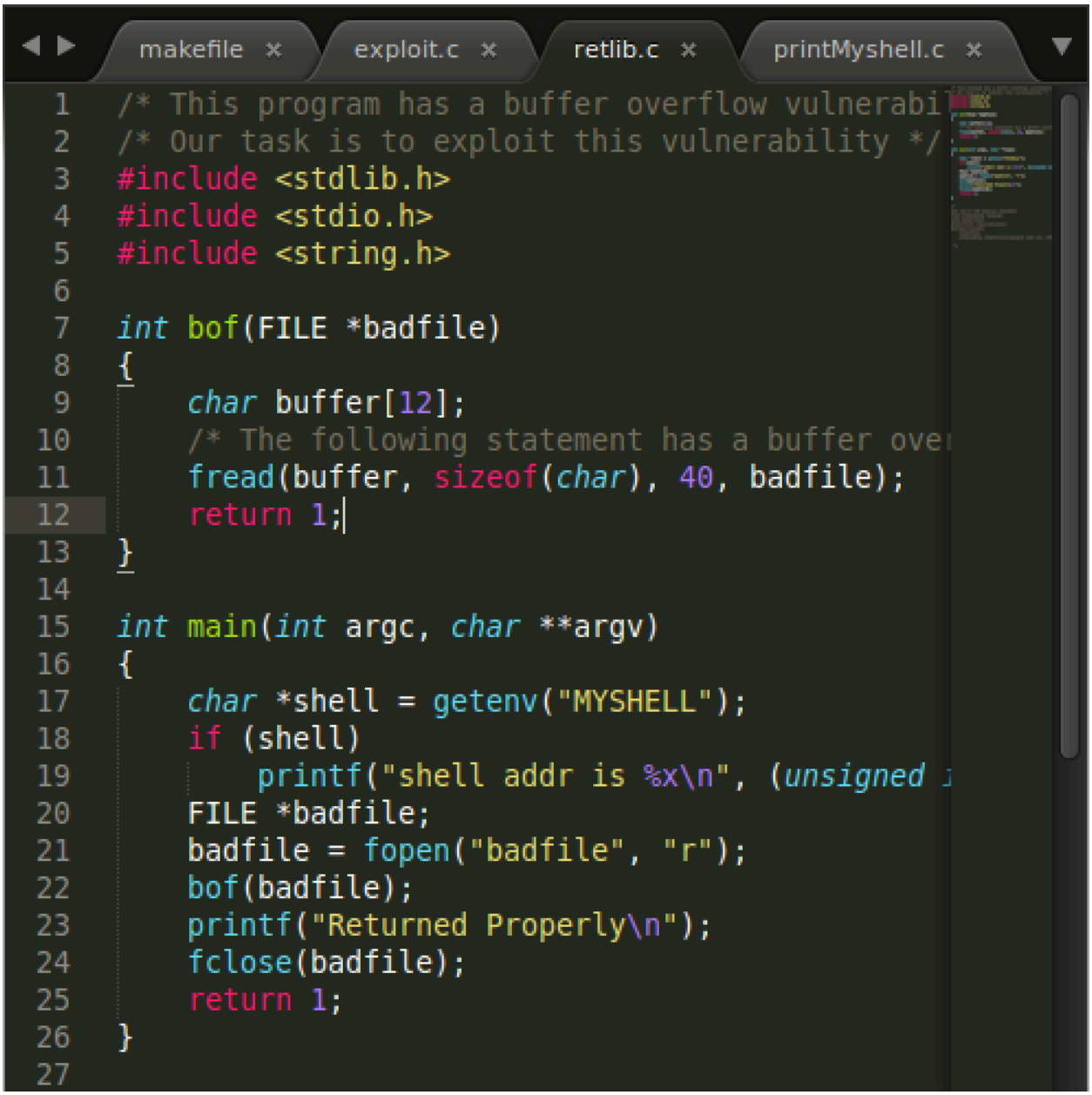



Fig 1.11 modify retlib.c code to print addr of MYSHELL

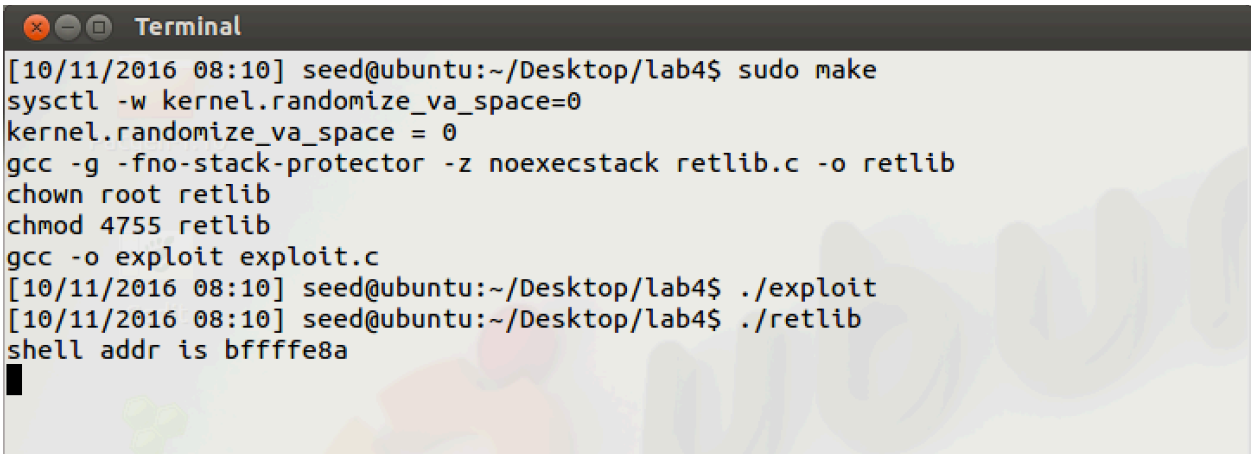



Fig 1.12 print out addr of MYSHELL

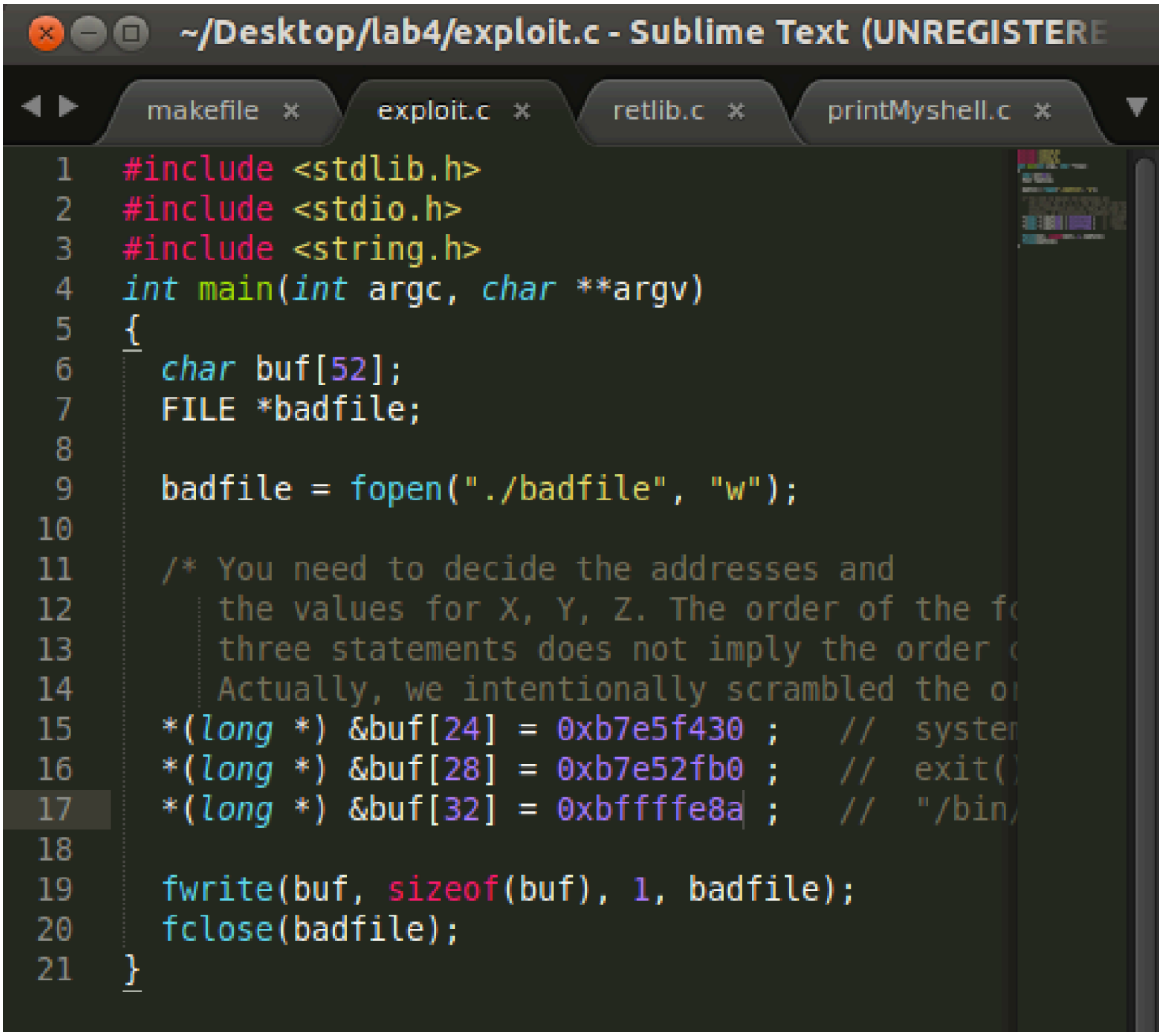



Fig 1.13 modify exploit.c code to match MYSHELL addr

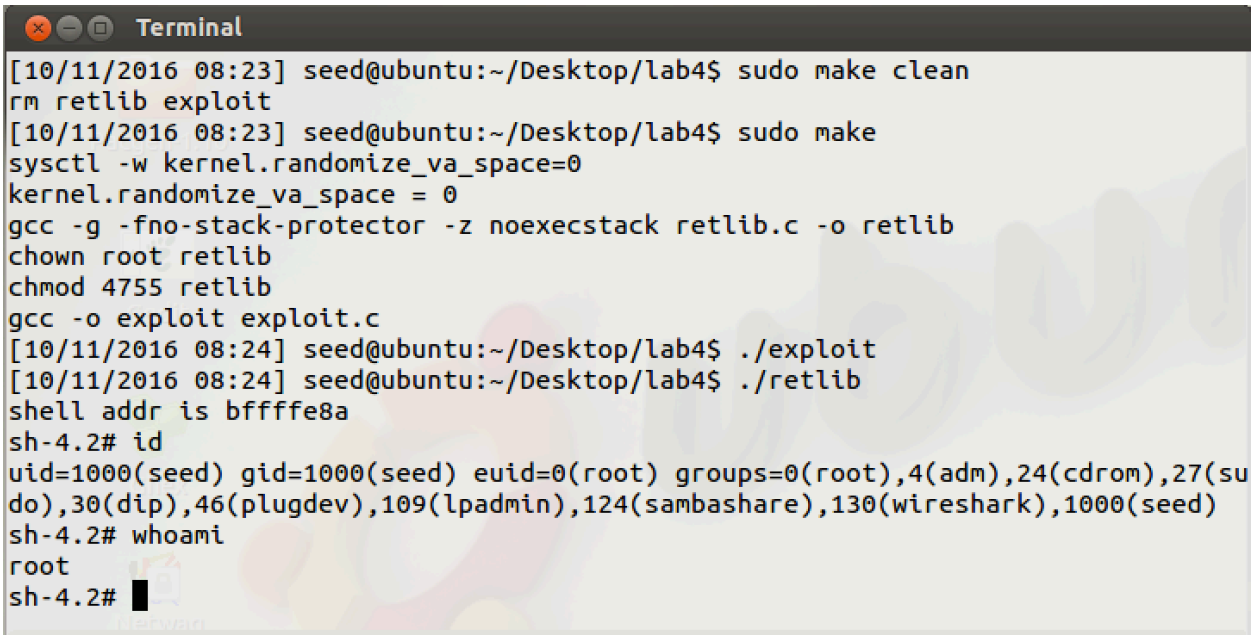



Fig 1.14 hold an attack again

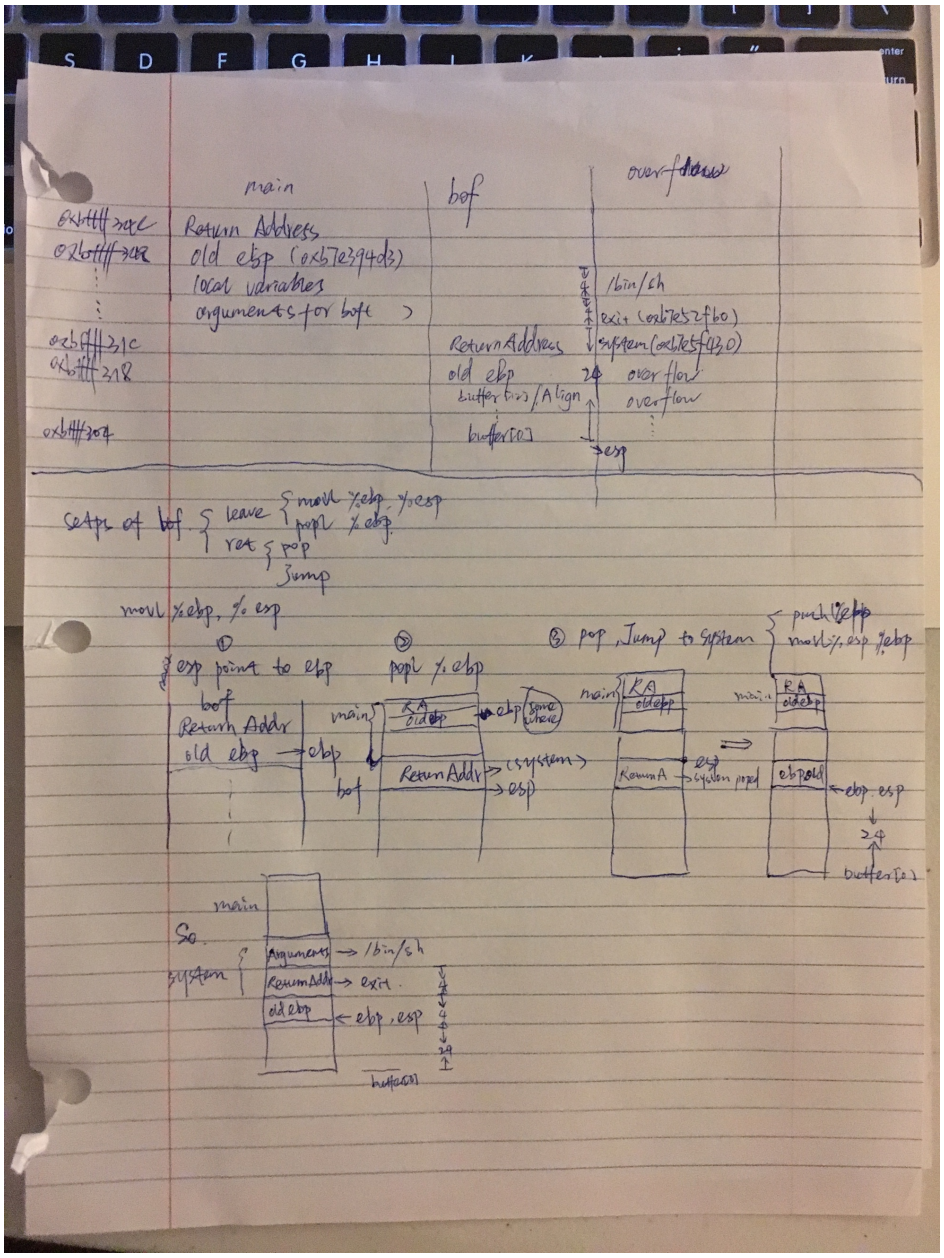
**Observation:**

1. As fig 1.1, I exported MYSHELL=/bin/sh
2. Like fig 1.3, I turned on no-executable stack
3. Like fig 1.5, I used gdb get system, exit address.
4. Like fig 1.6, I got distance between return address and buffer.
5. As fig 1.7 and fig 1.12, I got MYSHELL address. They are different.
6. As result of 1.7, I constructed badfile, like fig 1.8.
7. As fig 1.9, the attack was failed.
8. As fig 1.10, ed program is called by retlib.
9. When I used result of fig 1.12, I construct badfile like fig 1.13.
10. As fig 1.14, the attack was successful.

**Explanation:**

1. shell variable MYSHELL is used to pass string /bin/sh to system() to make the attack.

2. No-executable stack is used, because I would use system library to make the attack.
3. I could use gdb debugging to get address of system and exit. And I could use them in my attack.
4. I used distance from buffer to return address to decide value of buf[24](in exploit.c) to point to system address.
5. I got two different MYSHELL addresses. Perhaps when I used gdb, the layout of shell variables was different from without gdb debugging.
6. return address locates in: &buffer + 24
   return addr should point to system(0xb7e5f430)
   exit address locates in: &buffer + 24 + 4
   exit should point to (0xb7e52fb0)
   /bin/sh should locates in: &buffer + 24 + 4 + 4
   So I constructed badfile like fig 1.8
7. The attack was failed, because I used the MYSHELL address from gdb debugging. This address was different from process without gdb.
8. As fig 1.10, I could find that ed process was called in this attack. So the system function used the wrong value.
9. As result from fig 1.12, I constructed badfile. I changed the value to 0xbffffe8a locating at buf + 32. This woul point to /bin/sh.
10. Like fig 1.14, this attack was successful which means that all the construction and calculate were right. The process of overflow and jumping to system is displayed by the following picture.


main
bof
Return Address
old ebp (0xb7e394d3)
local variables
arguments for bof
Return Address
old ebp
buffer[0]
/bin/sh
exit (0xb7e2fbf0)
system (0xb7e5f430)
overflow
overflow
esp
setps of bof
leave
movl %ebp, %esp
popl %ebp
ret
pop
Jump
movl %ebp, %esp
① esp point to ebp
② popl %ebp
③ pop, Jump to system
push %ebp
movl %esp, %ebp
bof
Return Addr
old ebp
ebp
main
RA
old ebp
Some where
ReturnAddr
system
esp
ebp, esp
So
system
main
Arguments
ReturnAddr
old ebp
/bin/sh
exit
ebp, esp
buffer[0]

## Task 2: Address Randomization

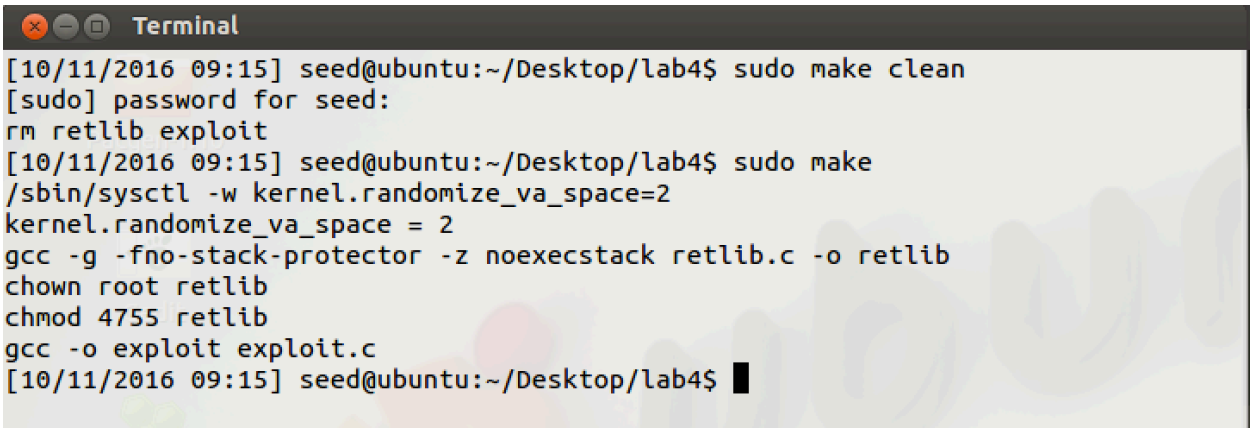



Fig 2.1 turn on address space randomization

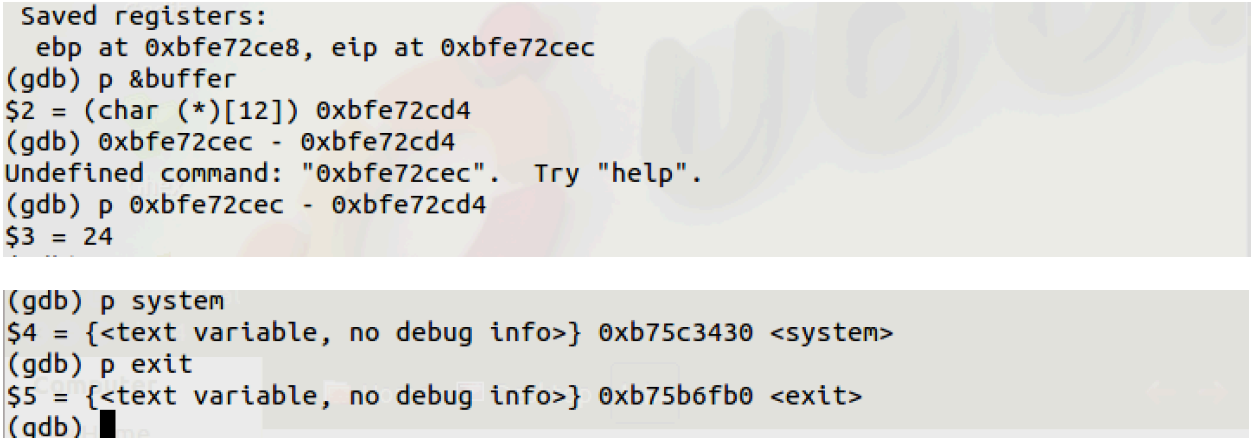



Fig 2.2 system and exit address

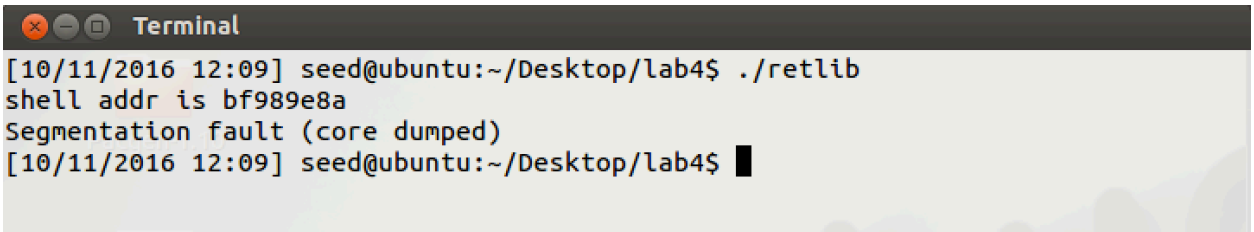



Fig 2.3 get MYSHELL address


Fig 2.4 construct the badfile according to above information

```
[10/11/2016 12:09] seed@ubuntu:~/Desktop/lab4$ gcc -o exploit exploit.c
[10/11/2016 12:12] seed@ubuntu:~/Desktop/lab4$ ./exploit
[10/11/2016 12:12] seed@ubuntu:~/Desktop/lab4$ ./retlib
shell addr is bfb13e8a
Segmentation fault (core dumped)
[10/11/2016 12:12] seed@ubuntu:~/Desktop/lab4$
```

Fig 2.5 make the attack

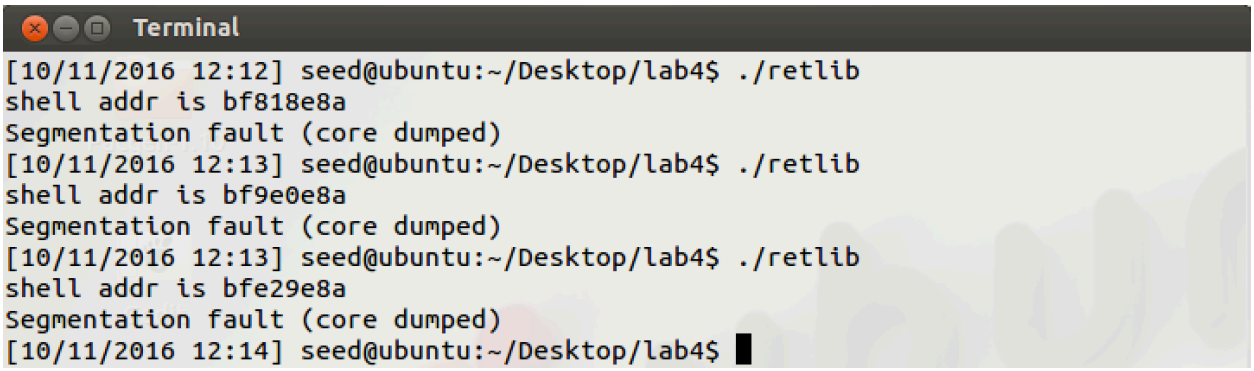



Fig 2.6 display the random address for MYSHELL

**Observation:**

1. As fig 2.2, I got system and exit address which was as same as the task 1 got.
2. When I run ./retlib, I got MYSHELL address 0xbf989e8a, so I constructed the badfile using program as fig 2.4 showed.
3. Like fig 2.5, I made an attack, but it was failed. The error message was "Segmentation fault (core dumped)".
4. As fig 2.6, I run ./retlib for several times, and as picture showed, I got several different address of MYSHELL.

**Explanation:**

1. As fig 2.2, I know that address space randomization does not affect address of system and exit function, which are located in c library.
2. return address locates in: &buffer + 24, according to fig 2.2
   return addr should point to system(0xb7e5f430)
   exit address locates in: &buffer + 24 + 4
   exit should point to (0xb7e52fb0)
   /bin/sh should locates in: &buffer + 24 + 4 + 4
   So I constructed badfile like fig 2.4
3. Because address space randomization in system was turned on. So the buffer address in stack is randomized. So the address I got from process that first time running is not same as the second time I run it again. So system function

could not get MYSHELL value, then /bin/sh would not be invoked. And process crashed reasons are following: 1. Protected address then process would crash; 2. Address for invalid instruction then process would crash; 3 the virtual address was not mapped then process would crash.

4. Because of address space randomization, MYSHELL address would change once I run or debug retlib process, like the picture 2.6 showed.

**Task 3: Stack Guard Protection**

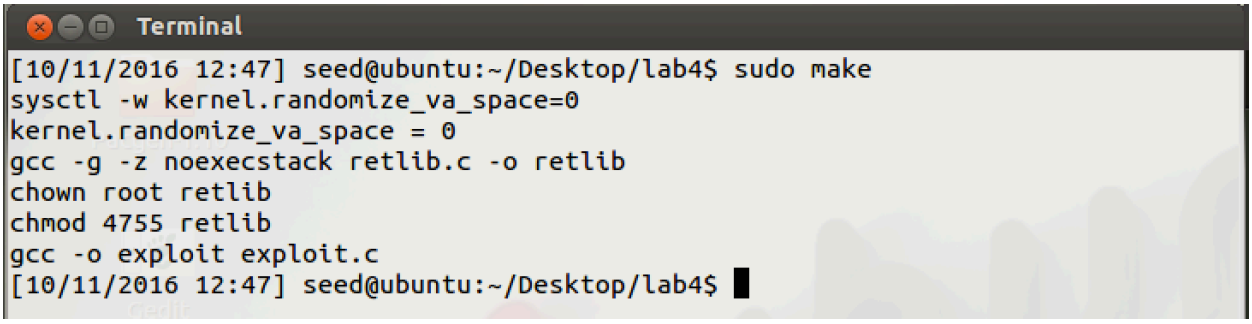



Fig 3.1 turn on stack guard and no-executable, then turn off randomization

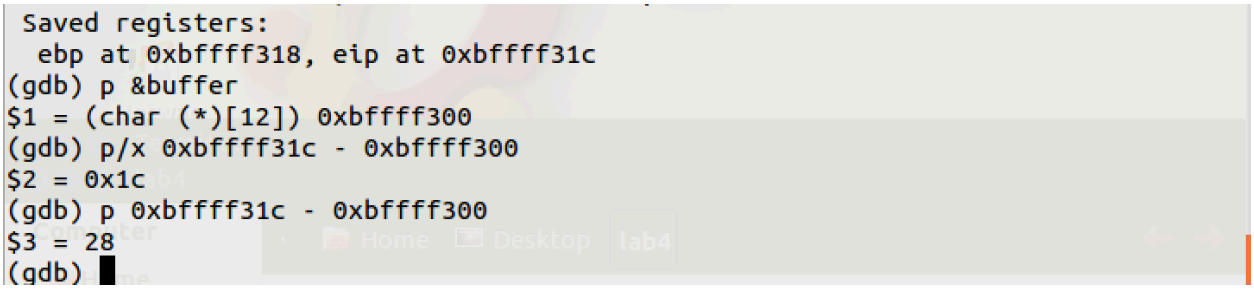



Fig 3.2 get distance from buffer to return address

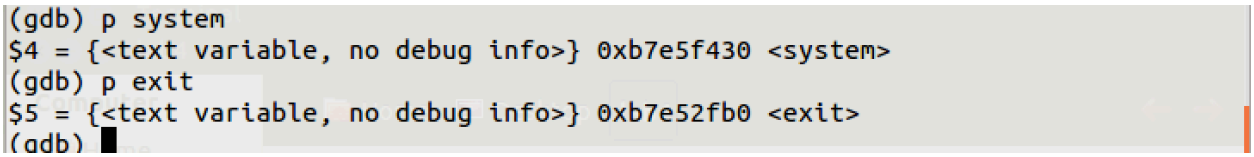



Fig 3.3 got address of system and exit

```
[10/11/2016 12:53] seed@ubuntu:~/Desktop/lab4$ ./retlib
shell addr is bffffe8a
*** stack smashing detected ***: ./retlib terminated
```

Fig 3.4 get MYSHELL address 0xbffffe8a

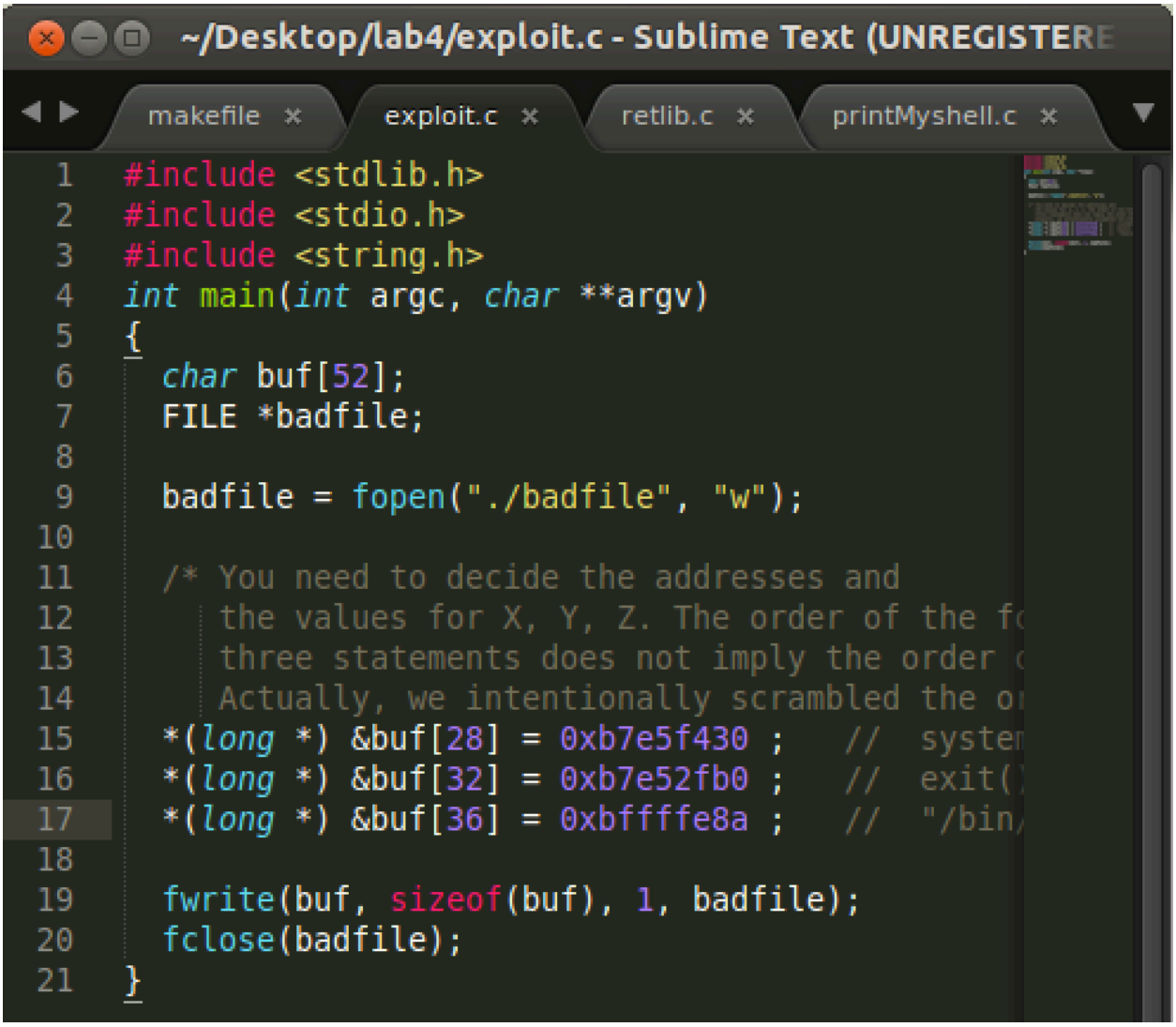



Fig 3.5 construct badfile with exploit.c code.

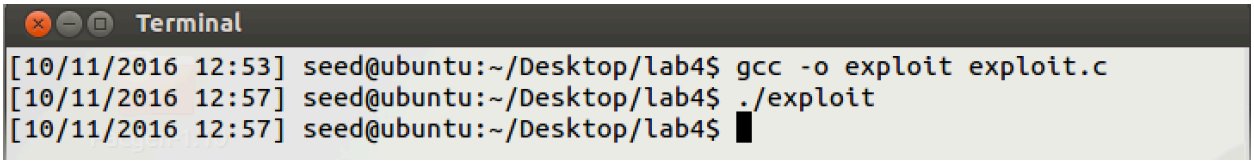



Fig 3.6 construct badfile

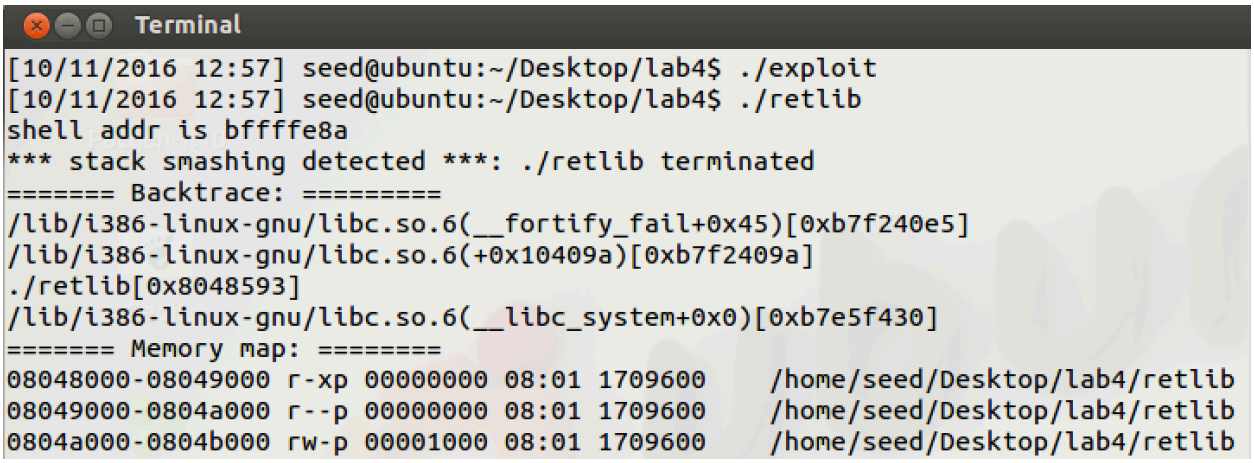



Fig 3.7 process attack

**Observation:**

1. The attack was failed, and the error message was printed out, “*** stack smashing detected ***: ./retlib terminated”.

**Explanation:**

2. GCC compiler made a security “Stack Guard” to prevent buffer overflow. As this protection buffer overflow would not work well. Once overflow was detected, the error message “*** stack smashing detected ***” would be printed and this process was terminated