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.

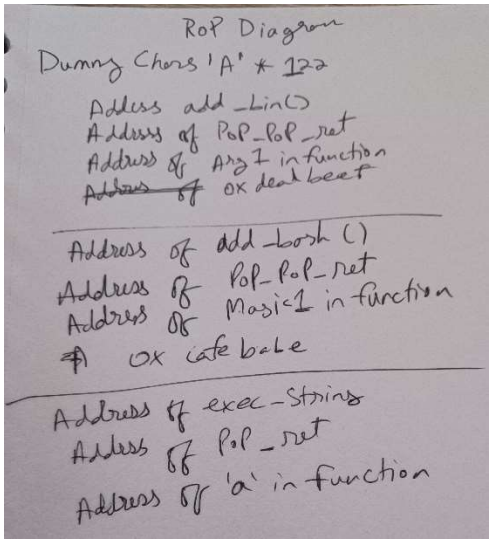
To do this, we will need to know the ret address from the vulnerable_function(). We do this by first dissembling the main function then disassembling the vulnerable_fucntion then setting up a break point at the ret address as shown in Image 1. The ret address is : 0x08048500. This is important because 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

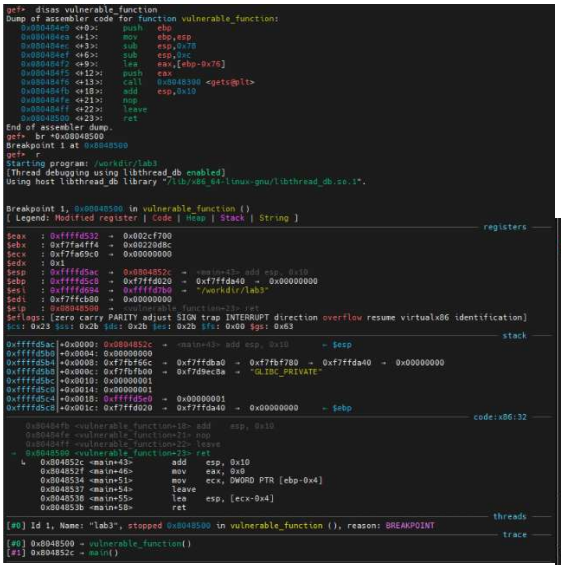
does happen, we can then redirect it to another function. With the ROP attack we will jump to all the functions at once. The /bin/bash command helps us get into the shell code, with the lab3 project using the system(string) it directly inserts into the terminal and as seen in the add_bin and add_bash 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:



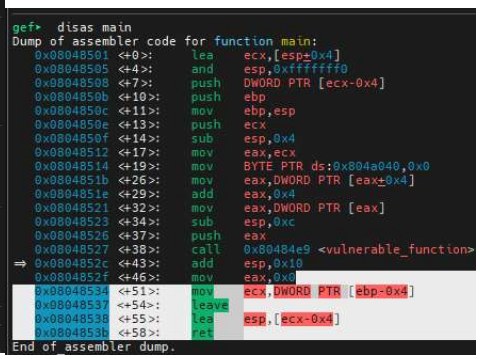
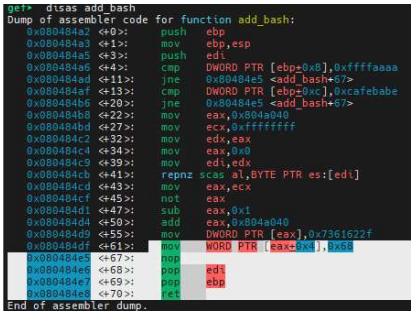


Image 1 *Image 2*



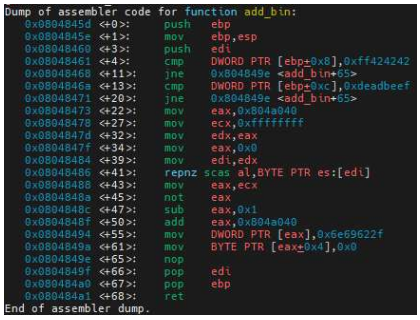
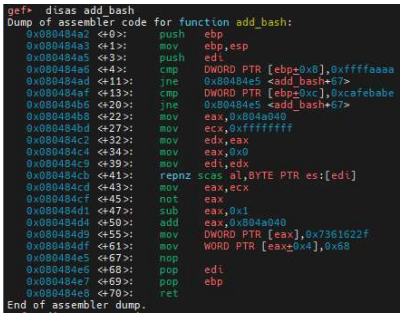


Image 3 *Image 4*



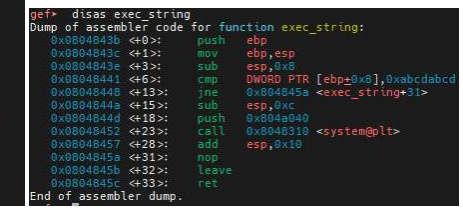


Image 5 *Image 6*

```
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()
```

Image 7

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

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