**RE lab 06 - ROP and ASLR**

**Lab files and setup**

Download the lab files from [here](https://pwnthybytes.ro/unibuc_re/06-lab-files.zip). The archive password is infected.

* For the Linux task, run:

$ apt-get install gdb git

$ cd

$ git clone https://github.com/longld/peda

$ echo "source ~/peda/peda.py" >> ~/.gdbinit

* Install rp++ from [here](https://github.com/0vercl0k/rp)

**pwntools utility functions**

context.arch = "amd64"

pop\_rdi\_ret = 0x400123

*#Either like this*

ropchain = p64(pop\_rdi\_ret) + p64(0x1234)

*#Or like this*

ropchain = flat([

pop\_rdi\_ret,

0x1234,

])

io.send(ropchain)

**Using gdb to find strings in memory**

gdb-peda$ find "%s" binary

Searching for '%s' in: binary ranges

Found 2 results, display max 2 items:

test : 0x402004 --> 0x3b031b0100007325

test : 0x403004 --> 0x3b031b0100007325

gdb-peda$ hexdump 0x402004

0x00402004 : 25 73 00 00 01 1b 03 3b 64 00 00 00 0b 00 00 00 %s.....;d.......

**Using gdb to find instructions in memory**

gdb-peda$ asmsearch "pop rdi; ret"

Searching for ASM code: 'pop rdi; ret' in: binary ranges

0x0040125b : (5fc3) pop rdi; ret

**Using gdb to find in-memory gadgets**

gdb-peda$ dumprop

Warning: this can be very slow, do not run for large memory range

Writing ROP gadgets to file: test-rop.txt ...

0x40115c: ret

0x40112a: repz ret

0x4011db: leave; ret

0x40125a: pop r15; ret

0x4010c0: pop rbp; ret

0x40125b: pop rdi; ret

**Using rp++ to find in-file gadgets:**

$ rp-lin-x64 -f ./test -r 1 --unique #here for length 1 gadgets

Trying to open './test'..

Loading ELF information..

FileFormat: Elf, Arch: x64

Using the Nasm syntax..

Wait a few seconds, rp++ is looking for gadgets..

**in** LOAD

55 found.

A total of 55 gadgets found.

You decided to keep only the unique ones, 30 unique gadgets found.

0x0040107e: add byte [rax], al **;** ret **;** **(**1 found**)**

0x0040107d: add byte [rax], r8L **;** ret **;** **(**1 found**)**

0x00401128: add byte [rcx], al **;** rep ret **;** **(**1 found**)**

0x00401129: add ebx, esi **;** ret **;** **(**1 found**)**

0x00401013: add esp, 0x08 **;** ret **;** **(**2 found**)**

0x00401012: add rsp, 0x08 **;** ret **;** **(**2 found**)**

0x00401241: call qword [r12+rbx\*8] **;** **(**1 found**)**

0x00401196: call qword [rax+0x4855C35D] **;** **(**1 found**)**

0x004011d9: call qword [rax+0x4855C3C9] **;** **(**1 found**)**

0x00401155: call qword [rbp+0x48] **;** **(**1 found**)**

0x00401242: call qword [rsp+rbx\*8] **;** **(**1 found**)**

0x00401010: call rax **;** **(**2 found**)**

0x00401244: fmul qword [rax-0x7D] **;** ret **;** **(**1 found**)**

0x004010b5: jmp rax **;** **(**2 found**)**

0x004011db: leave **;** ret **;** **(**1 found**)**

0x00401123: mov byte [0x0000000000404058], 0x00000001 **;** rep ret **;** **(**1 found**)**

0x0040114c: mov ebp, esp **;** call rax **;** **(**1 found**)**

0x004010b0: mov edi, 0x00404038 **;** jmp rax **;** **(**2 found**)**

0x0040123f: mov edi, ebp **;** call qword [r12+rbx\*8] **;** **(**1 found**)**

0x0040123e: mov edi, r13d **;** call qword [r12+rbx\*8] **;** **(**1 found**)**

0x0040114b: mov rbp, rsp **;** call rax **;** **(**1 found**)**

0x0040107b: nop dword [rax+rax+0x00] **;** ret **;** **(**1 found**)**

0x0040125d: nop dword [rax] **;** ret **;** **(**1 found**)**

0x00401240: out dx, eax **;** call qword [r12+rbx\*8] **;** **(**1 found**)**

0x0040125a: pop r15 **;** ret **;** **(**1 found**)**

0x004010c0: pop rbp **;** ret **;** **(**7 found**)**

0x0040125b: pop rdi **;** ret **;** **(**1 found**)**

0x0040112a: rep ret **;** **(**1 found**)**

0x00401016: ret **;** **(**15 found**)**

0x0040123d: test byte [rcx+rcx\*4-0x11], 0x00000041 **;** call qword [rsp+rbx\*8] **;** **(**1 found**)**

**Using gdb to find function offsets**

gdb-peda$ p puts

$1 = {<text variable, no debug info>} 0x7ffff7e37b10 <puts>

gdb-peda$ xinfo 0x7ffff7e37b10

0x7ffff7e37b10 (<puts>: push r13)

Virtual memory mapping:

Start : 0x00007ffff7de8000

End : 0x00007ffff7f30000

Offset: 0x4fb10

Perm : r-xp

Name : /lib/x86\_64-linux-gnu/libc-2.28.so

So the puts function is at offset 0x4fb10. If, through an information leak, we find that puts is at address:

* 0x7fe6fb1efb10, then libc was loaded at 0x7fe6fb1efb10 - 0x4fb10 = 0x00007fe6fb1a0000
* 0x7f06356adb10, then libc was loaded at 0x7f06356adb10 - 0x4fb10 = 0x00007f063565e000

**Tricks for format string vulnerability exploitation**

$ cat main.c

int main(int argc, char \*\*argv) {

//classic usage

printf("%d %d\n", 12, 34);

//indexed usage, equivalent to the above

printf("%1$d %2$d\n", 12, 34);

//switched indexes

printf("%2$d %1$d\n", 12, 34);

//reading out of bounds

printf("%1$d %2$d %3$d %4$d\n", 12, 34);

// reading out of bounds from arbitrary start

printf("%4$p %5$p %6$p %7$p %8$p %9$p %10$p %11$p \n", 12, 34);

int out;

//write number of bytes printed to "out" parameter

printf("%s %n", "TEST", &out);

printf("Written %d bytes\n", out); // "TEST " => 5 bytes

printf("%100s %n", "TEST", &out);

printf("Written %d bytes\n", out); // 100 + 1 => 101 bytes

return 0;

}

$ ./main

12 34

12 34

34 12

12 34 0 0

(nil) 0xa 0x7fffffffe178 0x155555050 0x7fffffffe170 (nil) 0x555555555240 0x7ffff7dfed0a

TEST Written 5 bytes

TEST Written 101 bytes

**Tasks**

The binaries have a trivial vulnerability as in the previous lab. However, this time, the end game is not to just print “Task X solved” but to obtain code execution. We achieve this by calling system("/bin/sh"). To this end, you will need to construct increasingly difficult ROP chains.

**Task 1: First ROP**

* Find the offset until the return address
* Find any ret instruction and construct a return sled. Step through it using gdb
* Using rp++ find a pop rdi; ret gadget.
* Call function f1 with the parameter 0xdeadbeef. **(1p)**
* Using rp++ find a pop rsi; ret gadget.
  + Is there any?
  + Relax the search term in order to find something equivalent
* Call function f1 with the parameter 0xdeadbeef and f2 with the parameters 0x1234, 0xabcd **(1p)**
* Using IDA find the address of system in the binary. Using gdb find the address of the string “/bin/sh\x00” in the binary
  + Note that not all payloads work. If you have a whitespace character such as “\n” or " " the scanf function terminates. Choose addresses according to these constraints.
* Construct a ROP chain that loads the address of “/bin/sh” as the first argument and calls system
* Exploit the service running at 45.76.91.112 10061 and read password.txt **(1p)**

**Task 2: Multi-step ROP**

* In this task, system is no longer called. However, it is possible to recover its address using a helper function.
* Call the leaky\_function and then main again. Using the address leak, calculate the base of libc. **(1p)**
* Turn the exploit into a full Remote Code Execution exploit. Use the service running at 45.76.91.112 10062 and read password.txt **(2p)**

**Task 3: format string info leak**

* Use the input to leak values from the stack (find the puts pointer stored on the stack in main) and obtain the address of libc **(3p)**
* Turn the exploit into a full Remote Code Execution exploit. Use the service running at 45.76.91.112 10063 and read password.txt **(1p)**

**Task 4: Advanced ROP (Bonus)**

* “Recreate” the leaky\_function from task2 using ROP in order to obtain the libc base and then call main again. **(2p)**
* Turn the exploit into a full Remote Code Execution exploit. Use the service running at 45.76.91.112 10064 and read password.txt **(1p)**
* Bonus++: modify the remote exploit such that main is not called a second time **(3p)**