Reverse Engineering April, 2023

Lab session 0x06

In this lab session, you will check out RE techniques and their mitigation such as:

- Address Space Layout Randomization / Position Independent Execution (ASLR/PIE);
- Stack Smashing Protections (SSP);
- Return Oriented Programming (ROP).

1 Lab files

The files for this lab session are available at:

- https://pwnthybytes.ro/unibuc_re/05-lab-files.zip
- https://pwnthybytes.ro/unibuc_re/06-lab-files.zip
- https://pwnthybytes.ro/unibuc_re/07-lab-files.zip

The password for all the zip files is *infected*.

2 Tools we use (Linux)

Today, all the work will be done in the Linux environment. Make sure you have *python3* and *pwntools* installed on your VM:

```
$ apt-get update
$ apt-get install python3-pycryptodome xinetd libffi-dev python3-wheel gcc
↪ gdb python3-setuptools python3-dev libssl-dev git libc6-dbg python3-pip make gcc-
↪ multilib socat
$ pip3 install pwntools
```

You need to install gdb and peda using the following:

```
$ apt-get install gdb git
$ cd
$ git clone https://github.com/longld/peda
$ echo "source ~/peda/peda.py" >> ~/.gdbinit
```

Then, also install rp++[1] and one_gadget[2].

Finally, check out the pwntools documentation for: opening processes/sockets and programmatic communication[3]; and packing, unpacking bytes, file I/O[4].

2.1 New gdb commands

We will use the following gdb commands.

- Investigating quad words starting from an address:

[1] https://github.com/0vercl0k/rp
[2] https://github.com/david942j/one_gadget
[3] https://github.com/Gallopsled/pwntools-tutorial/blob/master/tubes.md
[4] https://github.com/Gallopsled/pwntools-tutorial/blob/master/utility.md#packing-and-unpacking-integers

```
gdb-peda$ telescope $rsp 20
0000| 0x7fffffffde70 --> 0x401360 (<__libc_csu_init>:    push   r15)
0008| 0x7fffffffde78 --> 0x7ffff7dea09b (<__libc_start_main+235>:    mov    edi,eax)
0016| 0x7fffffffde80 --> 0x0
0024| 0x7fffffffde88 --> 0x7fffffffdf58 --> 0x7fffffffe2a2 ("/ctf/unibuc/curs_re/curs_"
  ↪ ...)
0032| 0x7fffffffde90 --> 0x100040000
0040| 0x7fffffffde98 --> 0x401208 (<main>:   push   rbp)
0048| 0x7fffffffdea0 --> 0x0
0056| 0x7fffffffdea8 --> 0xe2f3fe41bf131724
0064| 0x7fffffffdeb0 --> 0x4010c0 (<_start>:     xor    ebp,ebp)
0072| 0x7fffffffdeb8 --> 0x7fffffffdf50 --> 0x1
0080| 0x7fffffffdec0 --> 0x0
0088| 0x7fffffffdec8 --> 0x0
0096| 0x7fffffffded0 --> 0x1d0c013e24d31724
0104| 0x7fffffffded8 --> 0x1d0c117cd9751724
0112| 0x7fffffffdee0 --> 0x0
0120| 0x7fffffffdee8 --> 0x0
0128| 0x7fffffffdef0 --> 0x0
0136| 0x7fffffffdef8 --> 0x7fffffffdf68 --> 0x7fffffffe2d1 ("CLUTTER_IM_MODULE=xim")
0144| 0x7fffffffdf00 --> 0x7ffff7ffe190 --> 0x0
0152| 0x7fffffffdf08 --> 0x7ffff7fe44b6 (<_dl_init+118>:     cmp    ebx,0xffffffff)
```

- 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 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

- Investigating the protections of a binary

```
gdb-peda$ checksec
CANARY    : disabled   # No stack cookie
FORTIFY   : disabled
NX        : ENABLED    # Code is not modifiable, data is not executable
PIE       : ENABLED    # The binary is position independent
RELRO     : FULL       # The GOT table cannot be modified
```

- 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)
```

- Understand the 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)
```

- Tricks to 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 0
    ↪ x7ffff7dfed0a
TEST Written 5 bytes

    ↪                TEST Written 101 bytes
```

2.2 Tasks: working with PIE binaries

Working with a stripped PIE binary

Unfortunately, gdb has some bugs when setting breakpoints for a stripped PIE binary:

```
$ gdb ./task3
Reading symbols from ./task3...(no debugging symbols found)...done.
gdb-peda$ start
No unwaited-for children left.
Aborted (core dumped)
```

```
$ gdb ./task3
Reading symbols from ./task3...(no debugging symbols found)...done.
gdb-peda$ b *0x1338
Breakpoint 1 at 0x1338
gdb-peda$ run
Starting program: /ctf/unibuc/curs_re/curs_07/lab_07/task_03/task3
Warning:
Cannot insert breakpoint 1.
Cannot access memory at address 0x1338
```

The recommended way to set breakpoints is to disable ASLR system-wide:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Next, run the program once and get the base of the executable, and set breakpoints relative to it:

```
gdb-peda$ vmmap
Start              End                Perm  Name
0x0000555555554000 0x0000555555555000 r--p  /ctf/unibuc/curs_re/curs_07/lab_07/task_03/
    ↪ task3
0x0000555555555000 0x0000555555556000 r-xp  /ctf/unibuc/curs_re/curs_07/lab_07/task_03/
    ↪ task3
0x0000555555556000 0x0000555555557000 r--p  /ctf/unibuc/curs_re/curs_07/lab_07/task_03/
    ↪ task3
0x0000555555557000 0x0000555555558000 r--p  /ctf/unibuc/curs_re/curs_07/lab_07/task_03/
    ↪ task3
```

Note 0x0000555555555000, the .text section

```
gdb-peda$ b *0x0000555555554000 + 0x1338
Breakpoint 2 at 0x555555555338
gdb-peda$ run
Starting program: /ctf/unibuc/curs_re/curs_07/lab_07/task_03/task3
....


Breakpoint 2, 0x0000555555555338 in ?? ()
gdb-peda$ p $rip
$1 = (void (*)()) 0x555555555338
```

Working with a PIE binary that is not stripped

Breakpoints for relative addresses do not work (as seen in gdb before starting the executable or in IDA):

```
$ gdb  ./task01
Reading symbols from ./task01...
(No debugging symbols found in ./task01)
gdb-peda$ pdis main
Dump of assembler code for function main:
   0x0000000000001253 <+0>: push   rbp
   0x0000000000001254 <+1>: mov    rbp,rsp
```

```
   0x0000000000001257 <+4>: mov     eax,0x0
   0x000000000000125c <+9>: call    0x11c0 <setup>
   0x0000000000001261 <+14>:    mov     eax,0x0
   0x0000000000001266 <+19>:    call    0x11ef <vuln>
   0x000000000000126b <+24>:    mov     eax,0x0
   0x0000000000001270 <+29>:    pop     rbp
   0x0000000000001271 <+30>:    ret
End of assembler dump.
gdb-peda$ b *0x1271
Breakpoint 1 at 0x1271
gdb-peda$ run
Starting program: task01
Warning:
Cannot insert breakpoint 1.
Cannot access memory at address 0x1271
```

Breakpoints relative to known symbols work even under ASLR.

```
$ gdb -q ./task01
Reading symbols from ./task01...
(No debugging symbols found in ./task01)
gdb-peda$ pdis main
Dump of assembler code for function main:
   0x0000000000001253 <+0>: push    rbp
   0x0000000000001254 <+1>: mov     rbp,rsp
   0x0000000000001257 <+4>: mov     eax,0x0
   0x000000000000125c <+9>: call    0x11c0 <setup>
   0x0000000000001261 <+14>:    mov     eax,0x0
   0x0000000000001266 <+19>:    call    0x11ef <vuln>
   0x000000000000126b <+24>:    mov     eax,0x0
   0x0000000000001270 <+29>:    pop     rbp
   0x0000000000001271 <+30>:    ret
End of assembler dump.
gdb-peda$ set disable-randomization off
gdb-peda$ b *main+30
Breakpoint 1 at 0x1271
gdb-peda$ run
Starting program: /work/unibuc/curs_re/curs_07/lab_07/pack/task01
[============= Task 1 =============]
What is your name?
1234
Hello there, 1234
!
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x7f1dc64856e0 (<__write_nocancel+7>:  cmp     rax,0xfffffffffffff001)
RDX: 0x7f1dc6754780 --> 0x0
RSI: 0x7fff987901e0 ("Hello there, 1234\n!\n")
RDI: 0x1
RBP: 0x55915b47c280 (<__libc_csu_init>: push    r15)
RSP: 0x7fff98792908 --> 0x7f1dc63af830 (<__libc_start_main+240>:     mov     edi,eax)
RIP: 0x55915b47c271 (<main+30>: ret)
R8 : 0x7f1dc6979700 (0x00007f1dc6979700)
R9 : 0x14
R10: 0x5
R11: 0x246
R12: 0x55915b47c090 (<_start>:  xor     ebp,ebp)
R13: 0x7fff987929e0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x55915b47c25c <main+9>: call    0x55915b47c1c0 <setup>
```

```
   0x55915b47c261 <main+14>:     mov    eax,0x0
   0x55915b47c266 <main+19>:     call   0x55915b47c1ef <vuln>
   0x55915b47c26b <main+24>:     mov    eax,0x0
   0x55915b47c270 <main+29>:     pop    rbp
=> 0x55915b47c271 <main+30>:     ret
   0x55915b47c272:   nop    WORD PTR cs:[rax+rax*1+0x0]
   0x55915b47c27c:   nop    DWORD PTR [rax+0x0]
   0x55915b47c280 <__libc_csu_init>:     push   r15
   0x55915b47c282 <__libc_csu_init+2>:   mov    r15,rdx
   0x55915b47c285 <__libc_csu_init+5>:   push   r14
   0x55915b47c287 <__libc_csu_init+7>:   mov    r14,rsi
[------------------------------------stack-------------------------------------]
0000| 0x7fff98792908 --> 0x7f1dc63af830 (<__libc_start_main+240>:   mov    edi,eax)
0008| 0x7fff98792910 --> 0x1
0016| 0x7fff98792918 --> 0x7fff987929e8 --> 0x7fff987933fc ("/work/unibuc/curs_re/
    ↪ curs_07/lab_07/pack/task01")
0024| 0x7fff98792920 --> 0x1c697dca0
0032| 0x7fff98792928 --> 0x55915b47c253 (<main>:    push   rbp)
0040| 0x7fff98792930 --> 0x0
0048| 0x7fff98792938 --> 0x573f0e24e3051cf4
0056| 0x7fff98792940 --> 0x55915b47c090 (<_start>:   xor    ebp,ebp)
0064| 0x7fff98792948 --> 0x7fff987929e0 --> 0x1
0072| 0x7fff98792950 --> 0x0
0080| 0x7fff98792958 --> 0x0
0088| 0x7fff98792960 --> 0x3e2885934251cf4

[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x000055915b47c271 in main ()
gdb-peda$
```

3 Lab tasks: RE techniques

3.1 Smashing the stack (7p)

For this section, use the binaries in the 05-lab-files.zip file.

Task 3.1.1: stack-buffer overflow into data (2p)

- Do an initial analysis of the binary in IDA. Find the buffer overflow vulnerability and calculate the required input length to overwrite the pass_len variable. (0.5p)
- Starting from the template, construct an input that overflows into the pass_len variable and make the program print: (0.5p)

```
The correct password has length 12345
Unauthorized!
```

- Bypass the memcmp comparison by forcing its third parameter to be 0. (0.5p) Exploit the service running at 45.76.91.112 port 10051. (0.5p)

Task 3.1.2: stack-buffer overflow into ret addr (3p)

- Do an initial analysis of the binary in IDA. Find the buffer overflow vulnerability, look at the stack frame and calculate the required input length to overwrite the return address. (1p)
- Starting from the template, construct an input that overflows into the return address and replaces it with the address of do_login_success. (1p)

- Exploit the service running at 45.76.91.112 port 10052. (1p)

Task 3.1.3: stack-buffer overflow protection (2p)

- Starting from the template, construct an input that bypasses the stack protection but still overflows into the return address and replaces it with the address of do_login_success. (1p)
- Exploit the service running at 45.76.91.112 port 10053. (1p)

3.2 PIE tasks (7p)

For this section, use the binaries in the 07-lab-files.zip file.

Task 3.3.1: simple PIE (3p)

- Identify the binary protections and the helper function (which spawns a shell). (1p)
- Is the binary stripped? What approach is needed for breakpoints?
- Set a breakpoint on the return address in the vulnerable function:
  - run a couple of times with ASLR.
  - for each run, observe the raw 8 bytes of the (overwriteable) return address.
  - for each run, also observe the raw 8 bytes of the target (helper function) address.
  - how many bytes differ between the (overwriteable) return address and the target (helper function) address. (1p)
  - calculate the probability that a partial overwrite of the return address succeeds.
- Exploit the vulnerability by doing a partial overwrite of the return address. Remote end: 45.76.91.112 10071. (1p)

Task 3.3.2: complex PIE (4p)

- Identify the binary protections. (1p)
- Is the binary stripped? What approach is needed for breakpoints?
- Analyze the binary in IDA. What is the vulnerability present? What can it be used for? (1p)
- Use the one_gadget tool to find a couple of offsets into libc for shell spawn.
- Scan the GOT table to see which of those addresses differs the least (less bytes to overwrite => less failed tries) and calculate the probability here as well (use the same approach as in task 3.3.1). (1p)
- Exploit the vulnerability locally and remotely. Remote end: 45.76.91.112 10072. (1p)

3.3 ROP tricks (8p)

For this section, use the binaries in the 06-lab-files.zip file.

The binaries have a trivial vulnerability as in the previous section. 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 3.2.1: first ROP (3p)

- 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 3.2.2: multi-step ROP (3p)

- 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 3.2.3: format string info leak (2)

- 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. (1p)
- 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)