1. 审计程序

程序main函数是

```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  int v3; // [rsp+4h] [rbp-Ch]
  unsigned __int64 v4; // [rsp+8h] [rbp-8h]

  v4 = __readfsqword(0x28u);
  init_proc();
  while ( 1 )
  {
    while ( 1 )
    {
      menu();
      _isoc99_scanf("%d", &v3);
      if ( v3 != 3 )
        break;
      delete_note();
    }
    if ( v3 > 3 )
    {
      if ( v3 == 4 )
        exit(0);
      if ( v3 == 666 )
        backdoor();
LABEL_15:
      puts("Invalid choice");
    }
    else if ( v3 == 1 )
    {
      alloc_note();
    }
    else
    {
      if ( v3 != 2 )
        goto LABEL_15;
      edit_note();
    }
  }
}
```

1. 我们首先看下**init_proc**函数

```
ssize_t init_proc()
{
  ssize_t result; // rax
  int fd; // [rsp+Ch] [rbp-4h]

  setbuf(stdin, 0LL);
  setbuf(stdout, 0LL);
  setbuf(stderr, 0LL);
  if ( !mallopt(1, 0) )
    exit(-1);
  if ( mmap((void *)0xABCD0000LL, 0x1000uLL, 3, 34, -1, 0LL) != (void *)0xABCD0000LL )
    exit(-1);
  fd = open("/dev/urandom", 0);
  if ( fd < 0 )
    exit(-1);
  result = read(fd, (void *)0xABCD0100LL, 0x30uLL);
  if ( result != 48 )
    exit(-1);
  return result;
}
```

首先把stdin, stdout, stderr那几个函数的缓冲关闭

然后

```
mallopt(1,0)
```

我们可以看下

man mallopt

M_MXFAST (since glibc 2.3)
Set the upper limit for memory allocation requests that are satisfied using "fastbins". (The measurement unit for this parameter is bytes.) Fastbins are storage areas that hold deallocated blocks of memory of the same size without merging adjacent free blocks. Subsequent reallocation of blocks of the same size can be handled very quickly by allocating from the fastbin, although memory fragmentation and the overall memory footprint of the program can increase.

The default value for this parameter is *64*sizeof(size_t)/4* (i.e., 64 on 32-bit architectures). The range for this parameter is 0 to *80*sizeof(size_t)/4*. Setting **M_MXFAST** to 0 disables the use of fastbins.

大概是设置了这个，将fastbin关闭了

```
fd = open("/dev/urandom", 0);
if ( fd < 0 )
  exit(-1);
result = read(fd, (void *)0xABCD0100LL, 0x30uLL);
if ( result != 48 )
  exit(-1);
return result;
```

这里是读取0x30个随机字符到0xabcd0100的位置

2. 接下来我们看下alloc_note

```
unsigned __int64 alloc_note()
{
  int v1; // [rsp+0h] [rbp-10h]
  int i; // [rsp+4h] [rbp-Ch]
  unsigned __int64 v3; // [rsp+8h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  for ( i = 0; i <= 15 && note[i]; ++i )
    ;
  if ( i == 16 )
  {
    puts("full!");
  }
  else
  {
    puts("size ?");
    _isoc99_scanf("%d", &v1);
    if ( v1 > 0 && v1 <= 0xFFFFF )
    {
      note[i] = calloc(v1, 1uLL);
      note_size[i] = v1;
      puts("Done");
    }
    else
    {
      puts("Invalid size");
    }
  }
  return __readfsqword(0x28u) ^ v3;
}
```

看起来非常正常，先判断有没有空位置，有的话就要求输入chunk的size

如果size在正常范围的话，就调用calloc来分配内存，同时保存输入的size

3. edit_note

```
unsigned __int64 edit_note()
{
  int v1; // [rsp+0h] [rbp-10h]
  int v2; // [rsp+4h] [rbp-Ch]
  unsigned __int64 v3; // [rsp+8h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  puts("Index ?");
  _isoc99_scanf("%d", &v1);
  if ( v1 >= 0 && v1 <= 15 && note[v1] )
  {
    puts("Content: ");
    v2 = read(0, note[v1], (signed int)note_size[v1]);
    *((_BYTE *)note[v1] + v2) = 0;
    puts("Done");
  }
  else
  {
    puts("Invalid index");
  }
  return __readfsqword(0x28u) ^ v3;
}
```

首先要求输入index，如果index在正常范围且note也存在的话，就可以进行输入

关键就是输入之后，有一个置0截断，这里是一个漏洞，有off-by-null

4. delete_note

```
unsigned __int64 delete_note()
{
  int v1; // [rsp+4h] [rbp-Ch]
  unsigned __int64 v2; // [rsp+8h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  puts("Index ?");
  _isoc99_scanf("%d", &v1);
  if ( v1 >= 0 && v1 <= 15 && note[v1] )
  {
    free(note[v1]);
    note[v1] = 0LL;
    note_size[v1] = 0;
  }
  else
  {
    puts("Invalid index");
  }
  return __readfsqword(0x28u) ^ v2;
}
```

这里也是很正常，判断index的范围，free掉之后也有置0

5. backdoor

```
void __noreturn backdoor()
{
  char buf; // [rsp+0h] [rbp-40h]
  unsigned __int64 v1; // [rsp+38h] [rbp-8h]

  v1 = __readfsqword(0x28u);
  puts("If you can open the lock, I will let you in");
  read(0, &buf, 0x30uLL);
  if ( !memcmp(&buf, (const void *)0xABCD0100LL, 0x30uLL) )
    system("/bin/sh");
  exit(0);
}
```

这里读取30个字节，然后和0xabcd0100的进行比较，如果相同的话，就能get到shell

但是0xabcd0100那里的是随机生成的字符，不可预测，所以唯一的办法就是修改0xabcd0100那里的内容

2. 攻击思路

要控制0xabcd0100那里的内容，一种是利用unsorted bin attack之类的攻击那里，但是能攻击的数量太少了，只能攻击8个字节

fastbin也不能用，不能用fastbin attack攻击那里，唯一能想到的办法就是large bin attack了

但是题目的漏洞只有一个off-by-null，虽然比较难用，但是也足够了

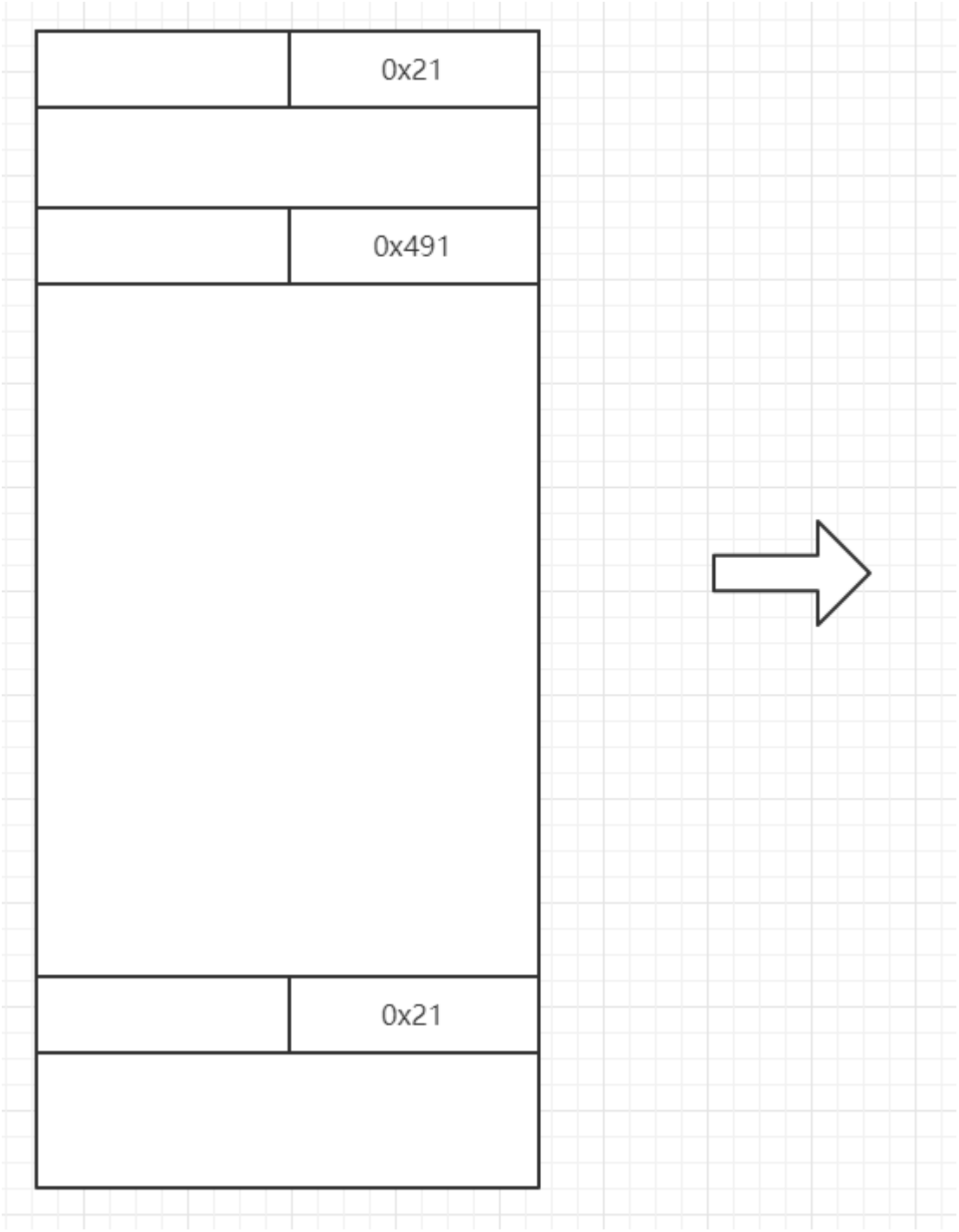
shrink chunk

首先通过chunk shrink来控制另外一个chunk，具体如下

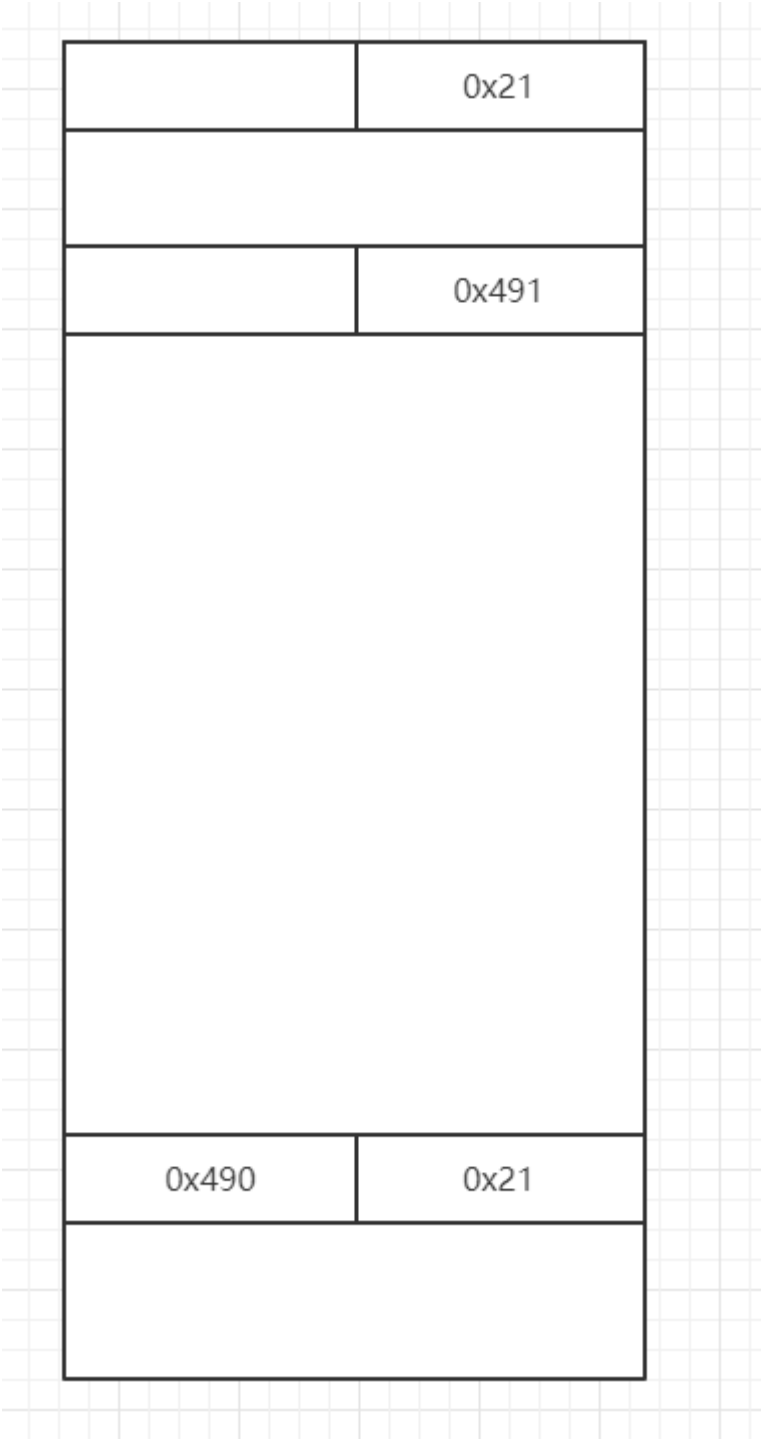
1. alloc chunk A、B、C，大小为0x18, 0x488, 0x18
2. free掉chunk B
3. 利用off by null，将chunk B的size改为0x400，在chunk B+0x400处构造一个fake chunk
4. 分配 chunk D、E，大小分别为0x18, 0x3d8，刚好占据了缩小后的chunk B的内存
5. free掉chunk D
6. free掉chunk C，这个时候会合并C、D
7. alloc一个大小为0x4a8的chunk，就能控制chunk E

图示如下

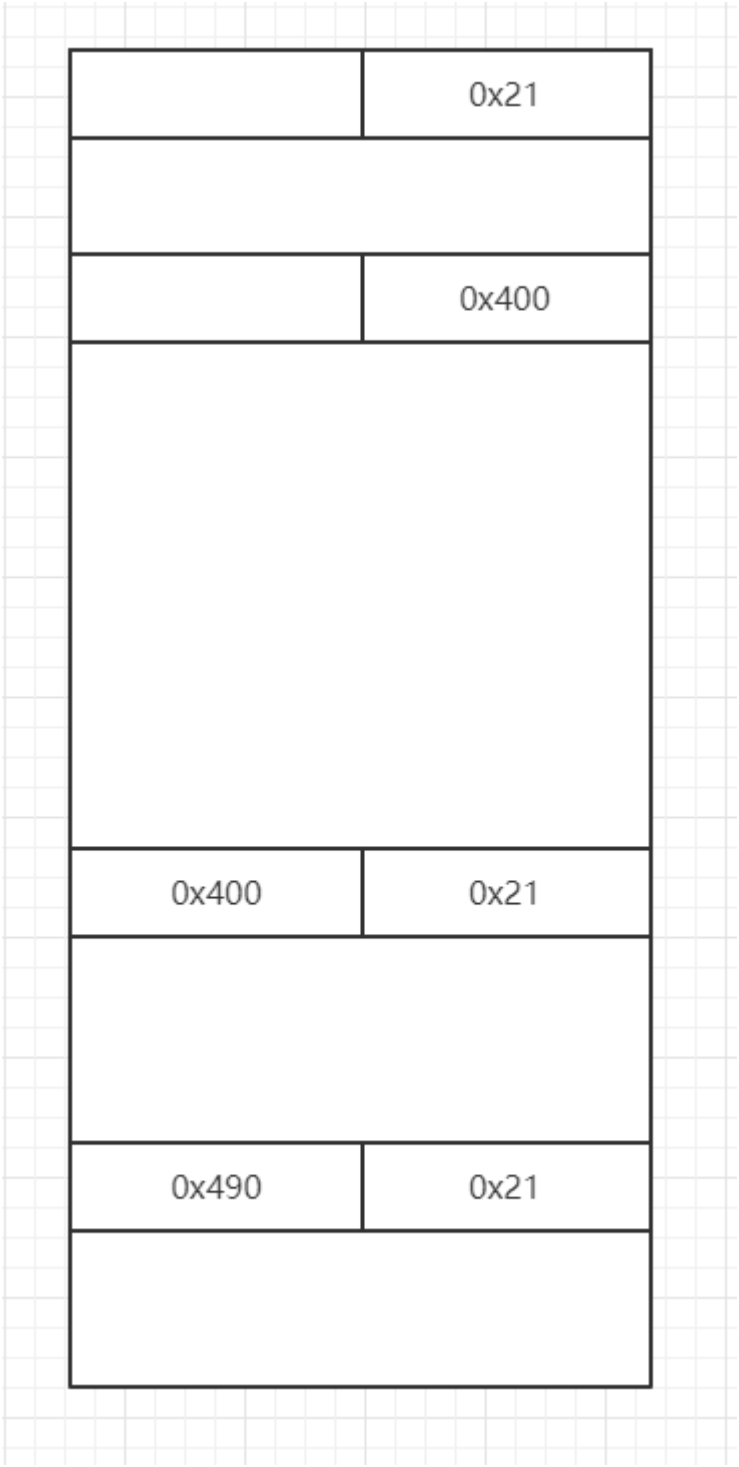
第一步之后



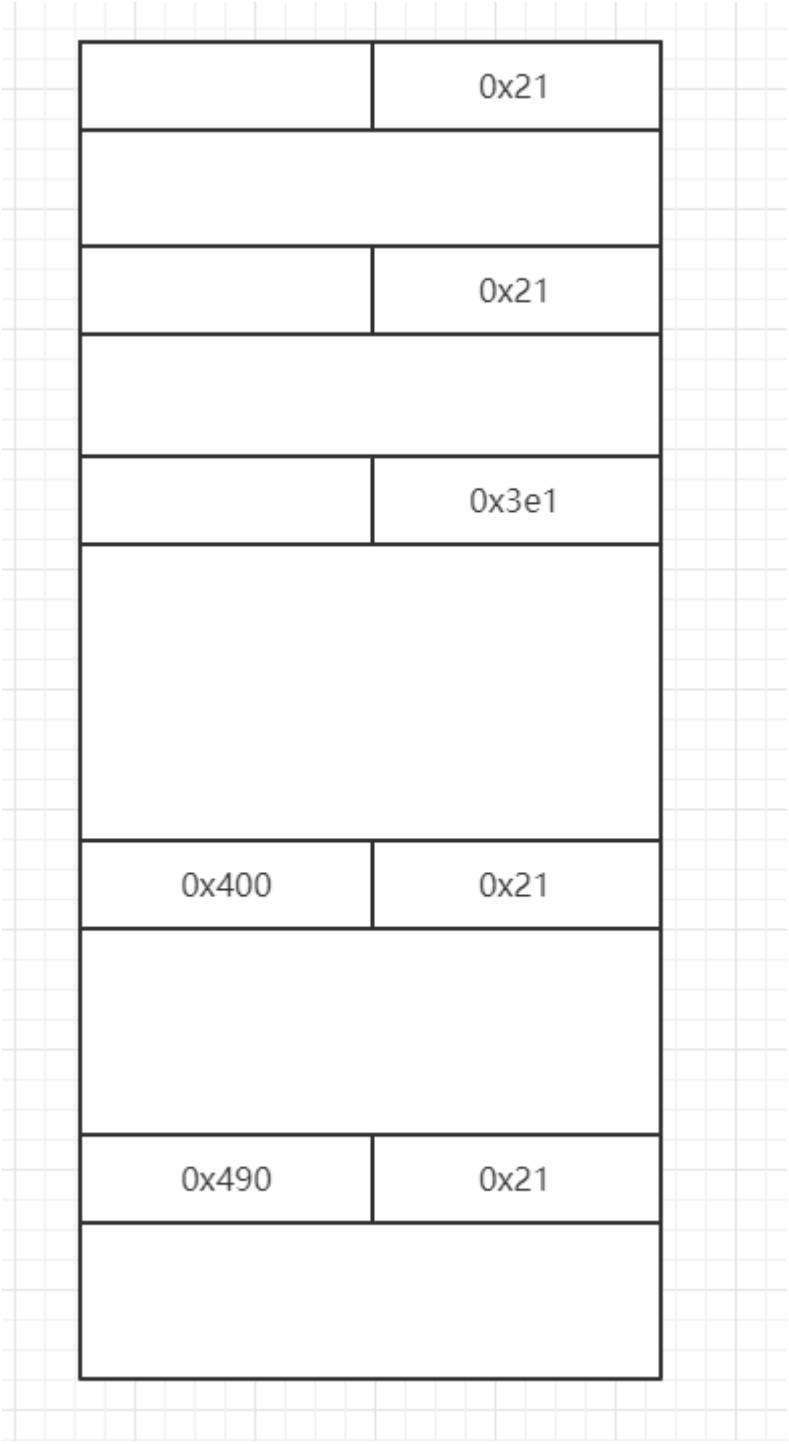
第二步之后



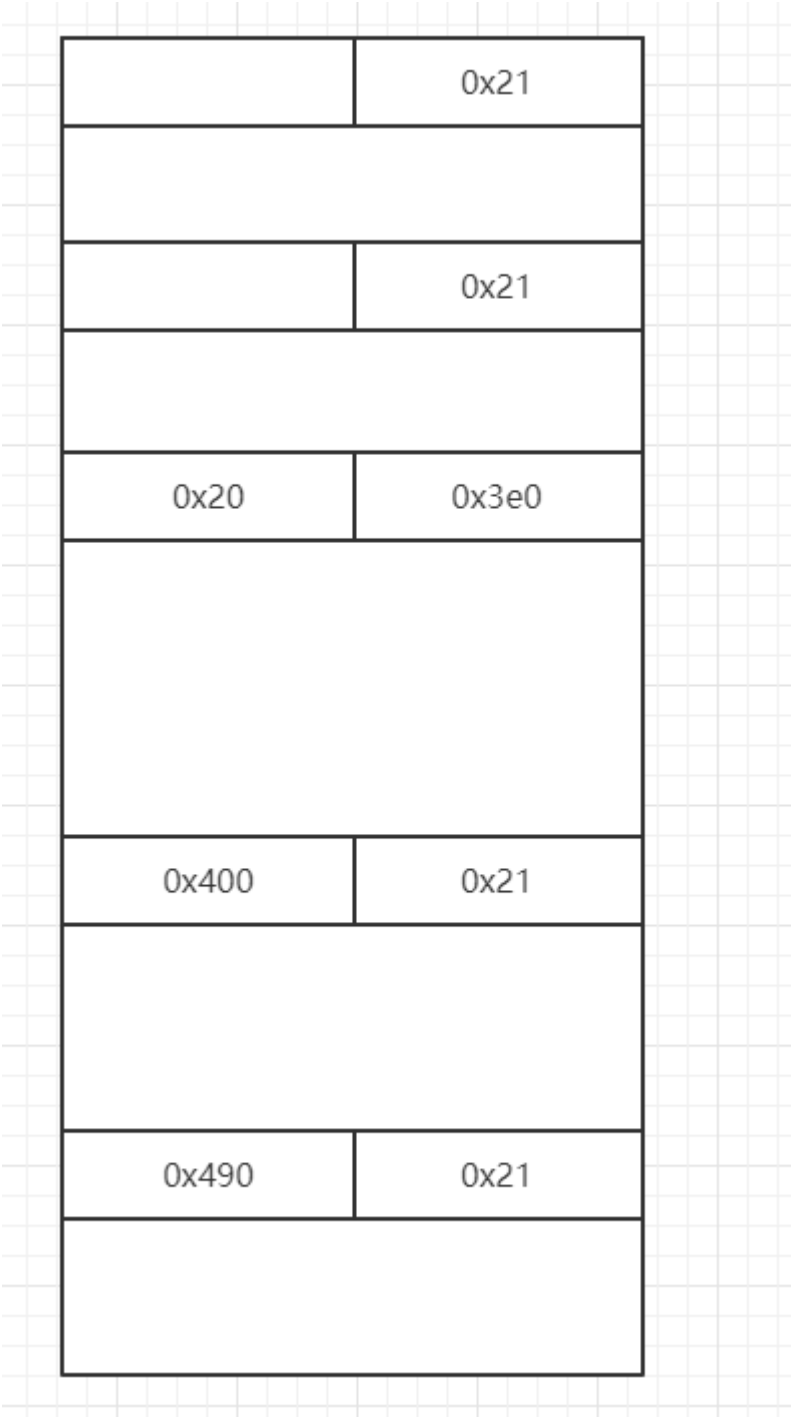
第三步之后



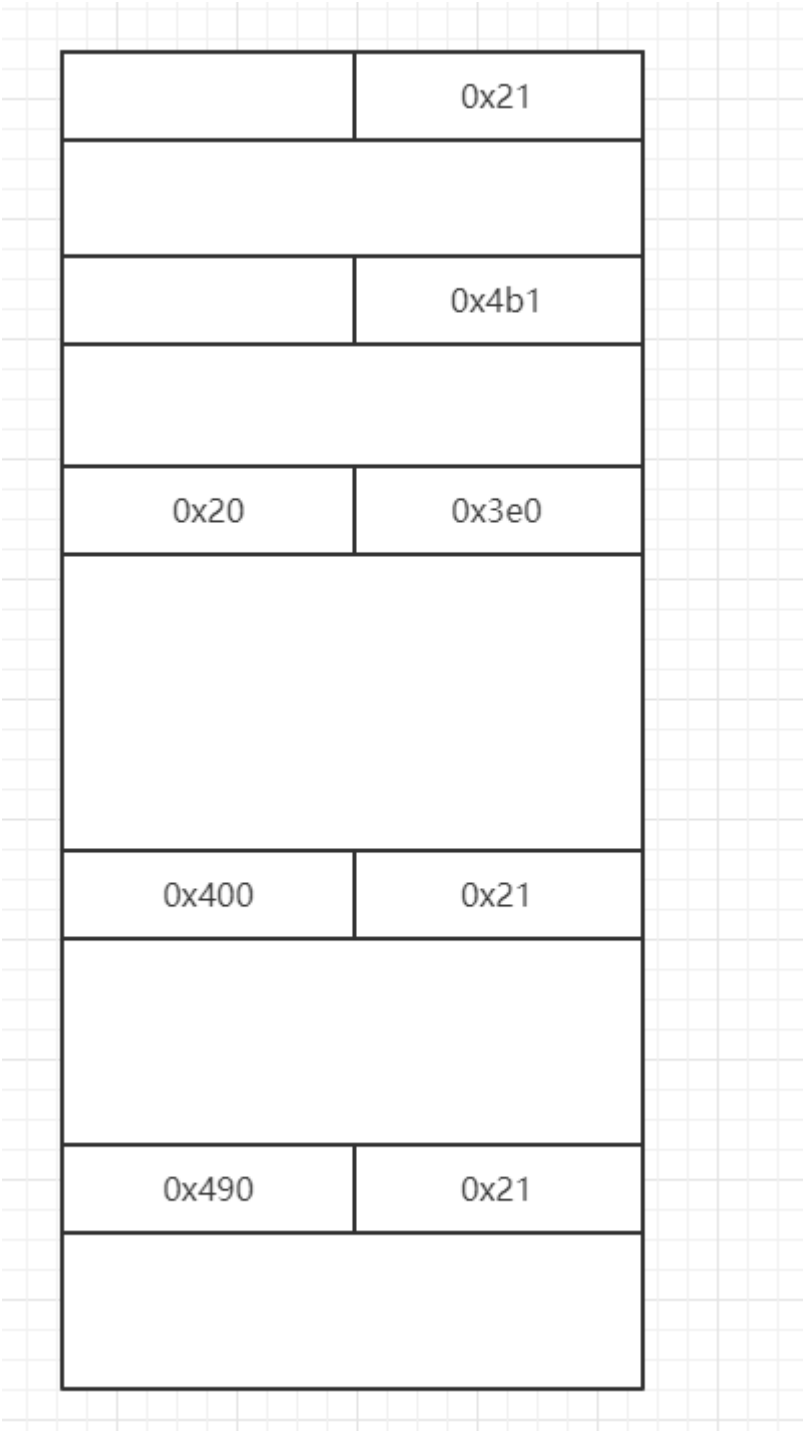
第四步之后



第五步之后



第六步之后



通过两次shrink chunk，我们就可以完全控制两个chunk

large bin attack

我们利用控制的两个chunk，修改chunk的size，然后构造一个large bin chunk，一个unsorted bin chunk

然后再控制里面的fd,bk, fd_nextsize, bk_nextsize

large bin 控制为

	0x421
0xdeadbeef	0xabcd00e0-0x10+3
0xdeadbeef	0xabcd00e0-8

unsorted bin 控制为

	0x431
0xdeadbeef	0xabcd00e0

然后这个时候alloc一个size为0x48的chunk，就可以控制0xabcd00f0之后的内容

但是为什么alloc一个0x48大小的chunk就能alloc到0xabcd00f0呢？

我们来去看下glibc源码

因为关闭了fastbin，所以我们alloc 0x48算是small bin

```
  if (in_smallbin_range (nb))
    {
      idx = smallbin_index (nb);
      bin = bin_at (av, idx);

      if ((victim = last (bin)) != bin)
        {
```

但是small bin list这个时候为空，所以第二个if就进不了去

```
/*
   If this is a large request, consolidate fastbins before continuing.
   While it might look excessive to kill all fastbins before
   even seeing if there is space available, this avoids
   fragmentation problems normally associated with fastbins.
   Also, in practice, programs tend to have runs of either small or
   large requests, but less often mixtures, so consolidation is not
   invoked all that often in most programs. And the programs that
   it is called frequently in otherwise tend to fragment.
 */

else
  {
    idx = largebin_index (nb);
    if (atomic_load_relaxed (&av->have_fastchunks))
      malloc_consolidate (av);
  }
```

这里是判断要求alloc的size是否在large bin的范围内，但是这里不在

```
for (;; )
  {
    int iters = 0;
    while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
      {
        bck = victim->bk;
        if (__builtin_expect (chunksize_nomask (victim) <= 2 * SIZE_SZ, 0)
            || __builtin_expect (chunksize_nomask (victim)
                                 > av->system_mem, 0))
          malloc_printerr ("malloc(): memory corruption");
        size = chunksize (victim);

        /*
           If a small request, try to use last remainder if it is the
           only chunk in unsorted bin.  This helps promote locality for
           runs of consecutive small requests. This is the only
           exception to best-fit, and applies only when there is
           no exact fit for a small chunk.
         */

        if (in_smallbin_range (nb) &&
            bck == unsorted_chunks (av) &&
            victim == av->last_remainder &&
            (unsigned long) (size) > (unsigned long) (nb + MINSIZE))
          {
            /* split and reattach remainder */
            remainder_size = size - nb;
            remainder = chunk_at_offset (victim, nb);
            unsorted_chunks (av)->bk = unsorted_chunks (av)->fd = remainder;
            av->last_remainder = remainder;
            remainder->bk = remainder->fd = unsorted_chunks (av);
            if (!in_smallbin_range (remainder_size))
              {
                remainder->fd_nextsize = NULL;
                remainder->bk_nextsize = NULL;
              }

            set_head (victim, nb | PREV_INUSE |
                      (av != &main_arena ? NON_MAIN_ARENA : 0));
            set_head (remainder, remainder_size | PREV_INUSE);
            set_foot (remainder, remainder_size);

            check_malloced_chunk (av, victim, nb);
            void *p = chunk2mem (victim);
            alloc_perturb (p, bytes);
            return p;
          }
```

这里就是将unsorted bin 从unsorted bin list中取出的循环

首先从unsorted bin list中取出我们构造的那个unsorted bin

```
if (in_smallbin_range (nb) &&
    bck == unsorted_chunks (av) &&
    victim == av->last_remainder &&
    (unsigned long) (size) > (unsigned long) (nb + MINSIZE))
  {
```

然后第一个判断，很明显bck不等于unsorted_chunks，所以这里就跳过这个if

```
        unsorted_chunks (av)->bk = bck;
        bck->fd = unsorted_chunks (av);
```

下面就是unsorted bin的unlink

这个时候0xabcd00e0+0x10 处就变成libc中的一个地址

```
    if (size == nb)
      {
        set_inuse_bit_at_offset (victim, size);
        if (av != &main_arena)
          set_non_main_arena (victim);
```

这里判断拿出来的unsorted bin 是否和要alloc的一样，这里明显不一样

```
/* place chunk in bin */

if (in_smallbin_range (size))
  {
    victim_index = smallbin_index (size);
    bck = bin_at (av, victim_index);
    fwd = bck->fd;
  }
else
  {
    victim_index = largebin_index (size);
    bck = bin_at (av, victim_index);
    fwd = bck->fd;

    /* maintain large bins in sorted order */
    if (fwd != bck)
      {
        /* Or with inuse bit to speed comparisons */
        size |= PREV_INUSE;
        /* if smaller than smallest, bypass loop below */
        assert (chunk_main_arena (bck->bk));
        if ((unsigned long) (size)
            < (unsigned long) chunksize_nomask (bck->bk))
          {
            fwd = bck;
            bck = bck->bk;

            victim->fd_nextsize = fwd->fd;
            victim->bk_nextsize = fwd->fd->bk_nextsize;
            fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim;
          }
        else
          {
            assert (chunk_main_arena (fwd));
            while ((unsigned long) size < chunksize_nomask (fwd))
              {
                fwd = fwd->fd_nextsize;
                assert (chunk_main_arena (fwd));
              }

            if ((unsigned long) size
                == (unsigned long) chunksize_nomask (fwd))
              /* Always insert in the second position.  */
              fwd = fwd->fd;
            else
              {
                victim->fd_nextsize = fwd;
                victim->bk_nextsize = fwd->bk_nextsize;
                fwd->bk_nextsize = victim;
```

这里是将拿出来的unsorted bin 放到对应的list中

这里很明显是large bin list

插入large bin list有三种情况，这里就不一一介绍了

我们来看关键的地方

```
else
  {
    victim->fd_nextsize = fwd;
    victim->bk_nextsize = fwd->bk_nextsize;
    fwd->bk_nextsize = victim;
    victim->bk_nextsize->fd_nextsize = victim;
  }
```

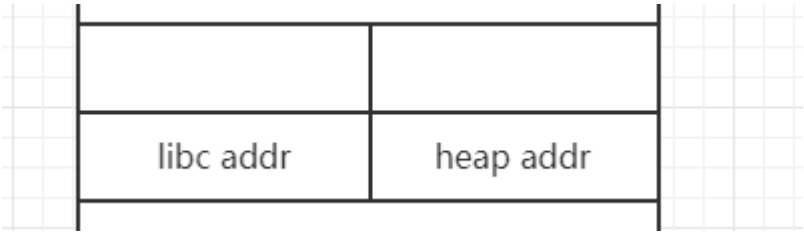
这里fwd是我们构造的large bin

```
fwd->bk_nextsize=0xabcd00e0-8
```

所以到执行

```
victim->bk_nextsize->fd_nextsize = victim;
=
(0xabcd00e0-8)->fd_nextsize = victim
=
*(0xabcd00e0+0x18) = victim
```

这个时候0xabcd00f0处就变成



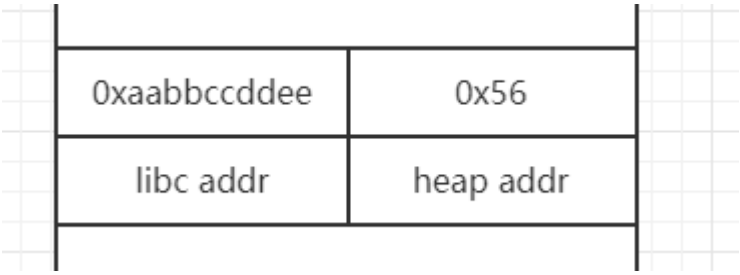
插完large bin list之后，还有一些代码

```
mark_bin (av, victim_index);
victim->bk = bck;
victim->fd = fwd;
fwd->bk = victim;
bck->fd = victim;
```

这里又是一个任意写堆地址

```
bck->fd = victim
=
(0xabcd00e0-0x10+3)->fd =victim
=
*(0xabcd00e0+3)=victim
```

然后0xabcd00e0处就变成



这样就伪造了一个0x56大小的chunk，对应alloc的大小是0x48

经过上面两个任意写堆地址之后，循环还没结束

下个unsorted bin就是我们伪造的这个chunk，所以我们alloc 0x48大小的chunk

```
  if (size == nb)
    {
      set_inuse_bit_at_offset (victim, size);
      if (av != &main_arena)
        set_non_main_arena (victim);
```

在这里就过了if判断，直接返回我们构造的chunk

这样，我们就能控制0xabcd00f0后面0x48大小的内存了

控制之后，改写里面的内容，就可以经过backdoor的校验，成功get 到shell