

前言

□ 对一些有趣的堆相关的漏洞的利用做一个记录，如有差错，请见谅。

□ 文中未做说明 均是指 glibc 2.23

□ 相关引用已在文中进行了标注，如有遗漏，请提醒。

简单源码分析

□ 本节只是简单跟读了一下 malloc 和 free 的源码，说的比较简单，很多细节还是要自己拿一份源代码来读。

堆中的一些数据结构

堆管理结构

```
struct malloc_state {
    mutex_t mutex;                /* Serialize access. */
    int flags;                    /* Flags (formerly in max_fast). */
#ifdef THREAD_STATS
    /* Statistics for locking. Only used if THREAD_STATS is defined. */
    long stat_lock_direct, stat_lock_loop, stat_lock_wait;
#endif
    mfastbinptr fastbins[NFASTBINS]; /* Fastbins */
    mchunkptr top;
    mchunkptr last_remainder;
    mchunkptr bins[NBINS * 2];
    unsigned int binmap[BINMAPSIZE]; /* Bitmap of bins */
    struct malloc_state *next;        /* Linked list */
    INTERNAL_SIZE_T system_mem;
    INTERNAL_SIZE_T max_system_mem;
};
```

- malloc_state结构是我们最常用的结构，其中的重要字段如下：
- fastbins：存储多个链表。每个链表由空闲的fastbin组成，是fastbin freelist。
- top：top chunk，指向的是arena中剩下的空间。如果各种freelist都为空，则从top chunk开始分配堆块。
- bins：存储多个双向链表。意义上和堆块头部的双向链表一样，并和其组成了一个双向环状空闲列表（freelist）。这里的bins位于freelist的结构上的头部，后向指针（

堆块结构

```
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;

    /* Only used for large blocks: pointer to next larger size. */
    struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
    struct malloc_chunk* bk_nextsize;
};
```

- prev_size：相邻的前一个堆块大小。这个字段只有在前一个堆块（且该堆块为normal chunk）处于释放状态时才有意义。这个字段最重要（甚至是唯一）的作用就是用于堆块释放时快速和相邻的前一个空闲堆块融合。该字段不计入当前堆块的大小计算。
- size：本堆块的长度。长度计算方式：size字段长度+用户申请的长度+对齐。libc以 size_T 长度*2 为粒度对齐。例如 32bit 以 42=8byte 对齐，64bit 以 82=0x10 对齐。因为最少以8字节对齐，所以size一定是8的倍数，故size字段的最后三位恒为0，libc用这三个bit做标志flag。比较关键的是最后一个bit（pre_inuse），用于指示 bit=1。libc判断 当前堆块是否处于free状态的方法 就是 判断下一个堆块的 pre_inuse** 是否为 1。这里也是 double free 和 null byte offset 等漏洞利用的关键。

- `fd`
`&bk`: 双向指针, 用于组成一个双向空闲链表。故这两个字段只有在堆块free后才有意义。堆块在alloc状态时, 这两个字段内容是用户填充的数据。两个字段可以造成内存shoot等效果。
- 值得一提的是, 堆块根据大小, libc使用fastbin、chunk等逻辑上的结构代表, 但其存储结构上都是malloc_chunk结构, 只是各个字段略有区别, 如fastbin相对于chunk_freelist是个单向链表。

来源

Libc堆管理机制及漏洞利用技术

Malloc 源码分析

用户调用 malloc 时会先进入 __libc_malloc

```
void *
__libc_malloc (size_t bytes)
{
  mstate ar_ptr;
  void *victim;

  void *(*hook) (size_t, const void *)
    = atomic_forced_read (__malloc_hook);
  if (__builtin_expect (hook != NULL, 0))// ██████ __malloc_hook ██████████
    return (*hook)(bytes, RETURN_ADDRESS (0));

  arena_get (ar_ptr, bytes);

  victim = _int_malloc (ar_ptr, bytes);
  return victim;
}
```

如果设置了 `__malloc_hook` 就执行它然后返回，否则进入 `_int_malloc` 这个函数就是 `malloc` 的具体实现

```
static void *
_int_malloc (mstate av, size_t bytes)
{
  /*
   * 2 * size_t 64 0x10
   * malloc(0x28) ----> nb = 0x30, 0x10 header + 0x20  + 0x8  pre_size
   */

  checked_request2size (bytes, nb);

  /*
   * malloc, sysmalloc--> mmap
   */
  if (__glibc_unlikely (av == NULL))
    {
      void *p = sysmalloc (nb, av);
      if (p != NULL)
        alloc_perturb (p, bytes);
      return p;
    }
}
```

首先把传入的 bytes 转换为 chunk 的实际大小，保存到 nb 里面。然后如果是第一次调用 malloc，就会进入 sysmalloc 分配内存。

搜索Fastbin

接着会看申请的 nb 是不是在 fastbin 里面，如果是进入 fastbin 的处理流程

```
if (((unsigned long) (nb) <= (unsigned long) (get_max_fast ()))
    {
        idx = fastbin_index (nb); // ■■■nb ■■■ fastbin ■ ■■ idx
        mfastbinptr *fb = &fastbin (av, idx); // ■■■■■ fastbin ■■■■
        mchunkptr pp = *fb;
        do
        {
            victim = pp;
            if (victim == NULL)
                break;
        }
    }
```

```

while ((pp = catomic_compare_and_exchange_val_acq (fb, victim->fd, victim))
      != victim);
if (victim != 0) //■■ fastbin ■■■■■■■■
{
    if (__builtin_expect (fastbin_index (chunksize (victim)) != idx, 0))// ■■■■■■■■ fastbin■■bin■■■■■
    {
        errstr = "malloc(): memory corruption (fast)";
        errout:
        malloc_printerr (check_action, errstr, chunk2mem (victim), av);
        return NULL;
    }
    check_reallocated_chunk (av, victim, nb);
    void *p = chunk2mem (victim);
    alloc_perturb (p, bytes);
    return p;
}
}

```

首先根据 nb 找到该大小对应的 fastbin 的项，然后看看该 fastbin 是不是为空，如果非空，就分配该 fastbin 的第一个 chunk 给用户。

分配过程还会检查待分配的 chunk 的 size 是不是满足在该 fastbin 项的限制。

```
fastbin_index (chunksize (victim)) != idx
```

搜索Smallbin

如果 fastbin 为空或者 nb 不在 fastbin 里面，就会进入 smallbin 和 largebin 的处理逻辑

```

if (in_smallbin_range (nb))
{
    idx = smallbin_index (nb);// ■■ smallbin ■■
    bin = bin_at (av, idx);
    if ((victim = last (bin)) != bin) // ■■ bin ■■■■■■ chunk
    {
        if (victim == 0) /* initialization check */
            malloc_consolidate (av);
        else
        {
            bck = victim->bk;
            if (__glibc_unlikely (bck->fd != victim)) // ■■■■
            {
                errstr = "malloc(): smallbin double linked list corrupted";
                goto errout;
            }
            set_inuse_bit_at_offset (victim, nb); //■■■■■chunk■ in_use ■
            bin->bk = bck;
            bck->fd = bin;

            if (av != &main_arena)
                victim->size |= NON_MAIN_ARENA;
            check_malloced_chunk (av, victim, nb);
            void *p = chunk2mem (victim);
            alloc_perturb (p, bytes);
            return p;
        }
    }
}

/*
■■■■■■■■■ malloc_consolidate
*/
else
{
    idx = largebin_index (nb);
    if (have_fastchunks (av))
        malloc_consolidate (av);
}

```

如果申请的 nb 位于 smallbin 的范围，就会 fastbin 一样去找对应的项，然后判断 bin 是不是为空，如果不空，分配第一个 chunk 给用户，分配之前还会校验该 chunk 是不是正确的。如果为空，就会进入 unsorted bin 的处理了。

如果 nb 不满足 smallbin , 就会触发 malloc_consolidate . 然后进入 unsorted bin

```

int iters = 0;
while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av)) // ■■ unsorted bin
{
    bck = victim->bk;
    size = chunksize (victim);

    if (in_smallbin_range (nb) &&
        bck == unsorted_chunks (av) &&
        victim == av->last_remainder &&
        (unsigned long) (size) > (unsigned long) (nb + MINSIZE))
    {
        /* split and reattach remainder */
        remainder_size = size - nb;
        remainder = chunk_at_offset (victim, nb);
        unsorted_chunks (av)->bk = unsorted_chunks (av)->fd = remainder;
        av->last_remainder = remainder;
        remainder->bk = remainder->fd = unsorted_chunks (av);
        if (!in_smallbin_range (remainder_size))
        {
            remainder->fd_nextsize = NULL;
            remainder->bk_nextsize = NULL;
        }

        set_head (victim, nb | PREV_INUSE |
                  (av != &main_arena ? NON_MAIN_ARENA : 0));
        set_head (remainder, remainder_size | PREV_INUSE);
        set_foot (remainder, remainder_size);

        check_mallocated_chunk (av, victim, nb);
        void *p = chunk2mem (victim);
        alloc_perturb (p, bytes);
        return p;
    }
}

```

```
(unsigned long) (size) > (unsigned long) (nb + MINSIZE)
```

否则就先把该 `unsorted bin` 从 `unsorted list` 中移除下来，这里用了一个类似 `unlink` 的操作，不过没有检查 `chunk` 的指针

[illegible]

```
if (in_smallbin_range (size))
{
    victim_index = smallbin_index (size);
    bck = bin at (av, victim_index);
```

```

        fwd = bck->fd;
    }
else
{
    victim_index = largebin_index (size);
    bck = bin_at (av, victim_index);
    fwd = bck->fd;
    .....
    .....

```

如果 size 在 smallbin 里就放到 smallbin , 否则就放到 large bin

搜索 Largebin

接下来就会去搜索 largebin 了

```

if (!in_smallbin_range (nb))
{
    bin = bin_at (av, idx);

    /* skip scan if empty or largest chunk is too small */
    if ((victim = first (bin)) != bin &&
        (unsigned long) (victim->size) >= (unsigned long) (nb))
    {
        victim = victim->bk_nextsize;
        while (((unsigned long) (size = chunksize (victim)) <
            (unsigned long) (nb)))
            victim = victim->bk_nextsize;

        /* Avoid removing the first entry for a size so that the skip
           list does not have to be rerouted.  */
        if (victim != last (bin) && victim->size == victim->fd->size)
            victim = victim->fd;

        remainder_size = size - nb;
        unlink (av, victim, bck, fwd);

        /* Exhaust */
        if (remainder_size < MINSIZE)
        {
            set_inuse_bit_at_offset (victim, size);
            if (av != &main_arena)
                victim->size |= NON_MAIN_ARENA;
        }
        /* Split */
        else
        {
            remainder = chunk_at_offset (victim, nb);
            /* We cannot assume the unsorted list is empty and therefore
               have to perform a complete insert here.  */
            bck = unsorted_chunks (av);
            fwd = bck->fd;
            if (__glibc_unlikely (fwd->bk != bck))
            {
                errstr = "malloc(): corrupted unsorted chunks";
                goto errout;
            }
            remainder->bk = bck;
            remainder->fd = fwd;
            bck->fd = remainder;
            fwd->bk = remainder;
            if (!in_smallbin_range (remainder_size))
            {
                remainder->fd_nextsize = NULL;
                remainder->bk_nextsize = NULL;
            }
            set_head (victim, nb | PREV_INUSE |
                (av != &main_arena ? NON_MAIN_ARENA : 0));
            set_head (remainder, remainder_size | PREV_INUSE);
            set_foot (remainder, remainder_size);

```

```

    }
    check_mallocated_chunk (av, victim, nb);
    void *p = chunk2mem (victim);
    alloc_perturb (p, bytes);
    return p;
}
}

```

使用 Top chunk

```

victim = av->top;
size = chunksize (victim);
// ■■ top chunk ■■■■■■■■ top chunk ■■■■
if ((unsigned long) (size) >= (unsigned long) (nb + MINSIZE))
{
    remainder_size = size - nb;
    remainder = chunk_at_offset (victim, nb);
    av->top = remainder;
    set_head (victim, nb | PREV_INUSE |
              (av != &main_arena ? NON_MAIN_ARENA : 0));
    set_head (remainder, remainder_size | PREV_INUSE);

    check_mallocated_chunk (av, victim, nb);
    void *p = chunk2mem (victim);
    alloc_perturb (p, bytes);
    return p;
}

/* When we are using atomic ops to free fast chunks we can get
   here for all block sizes. */
else if (have_fastchunks (av))
{
    malloc_consolidate (av);
    /* restore original bin index */
    if (in_smallbin_range (nb))
        idx = smallbin_index (nb);
    else
        idx = largebin_index (nb);
}

/*
   Otherwise, relay to handle system-dependent cases
*/
else
{
    void *p = sysmalloc (nb, av);
    if (p != NULL)
        alloc_perturb (p, bytes);
    return p;
}
}

```

如果 top chunk 的大小足够就直接切割分配，否则如果此时还有 fastbin 就触发 malloc_consolidate 重复上述流程，如果没有 fastbin 调用 sysmalloc 分配内存

Free 源码分析

_GI__libc_free

首先是 _GI__libc_free

```

void __fastcall _GI__libc_free(void *ptr)
{
    if ( _free_hook )
    {
        _free_hook(ptr, retaddr);
    }
    else if ( ptr )
    {

```

```

v1 = (unsigned __int64)ptr - 16;
v2 = *((_QWORD *)ptr - 1);
if ( v2 & 2 ) // size mmap chunk
{
    if ( !mp_.no_dyn_threshold
        && v2 > mp_.mmap_threshold
        && v2 <= 0x2000000
        && (v1 < (unsigned __int64)dumped_main_arena_start || v1 >= (unsigned __int64)dumped_main_arena_end) )
    {
        mp_.mmap_threshold = v2 & 0xFFFFFFFFFFFFFFFF8LL;
        mp_.trim_threshold = 2 * (v2 & 0xFFFFFFFFFFFFFFFF8LL);
    }
    munmap_chunk((mchunkptr)((char *)ptr - 16));
}
else
{
    av = &main_arena;
    if ( v2 & 4 )
        av = *(malloc_state **)(v1 & 0xFFFFFFFFC0000000LL);
    int_free(av, (mchunkptr)v1, 0);
}
}
}

```

如果存在 `free_hook` , 就会直接调用 `free_hook(ptr)` 然后返回。否则判断被 `free` 的内存是否是 `mmap` 获取的, 如果是则使用 `munmap_chunk` 回收内存, 否则进入 `_int_free`

`_int_free`

首先会做一些简单的检查

```

size = chunksize (p);

//
if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0)
    || __builtin_expect (misaligned_chunk (p), 0))
{
    errstr = "free(): invalid pointer";
errout:
    if (!have_lock && locked)
        (void) mutex_unlock (&av->mutex);
    malloc_printerr (check_action, errstr, chunk2mem (p), av);
    return;
}

// size >= MINSIZE
if (__glibc_unlikely (size < MINSIZE || !aligned_OK (size)))
{
    errstr = "free(): invalid size";
    goto errout;
}

// chunk inuse
check_inuse_chunk(av, p);

```

检查

- 指针是否对齐
- 块的大小是否对齐, 且大于最小的大小
- 块是否在 `inuse` 状态

进入 fastbin

```

if ((unsigned long)(size) <= (unsigned long)(get_max_fast ())) {
    if (have_lock
        || ({ assert (locked == 0);
              mutex_lock(&av->mutex);
              locked = 1;
              chunk_at_offset (p, size)->size <= 2 * SIZE_SZ // next->size <= 2 * SIZE_SZ
              || chunksize (chunk_at_offset (p, size)) >= av->system_mem; //

```



```

        >= ((char *) av->top + chunksize(av->top)), 0))
    {
errstr = "double free or corruption (out)";
goto errout;
    }
/*p ██████ inuse ██ */
if (__glibc_unlikely (!prev_inuse(nextchunk)))
    {
        errstr = "double free or corruption (!prev)";
        goto errout;
    }

nextsize = chunksize(nextchunk);
// nextsize █ [ 2 * SIZE_SZ, av->system_mem] █
if (__builtin_expect (nextchunk->size <= 2 * SIZE_SZ, 0)
|| __builtin_expect (nextsize >= av->system_mem, 0))
    {
        errstr = "free(): invalid next size (normal)";
        goto errout;
    }

free_perturb (chunk2mem(p), size - 2 * SIZE_SZ);

/* ██ p██████ free ██████████ p->pre_inused ███ */
if (!prev_inuse(p)) {
    prevsize = p->prev_size;
    size += prevsize;
    p = chunk_at_offset(p, -((long) prevsize));
    unlink(av, p, bck, fwd);
}

if (nextchunk != av->top) {
    // ██ nextchunk █████ chunk, █ pre_inused█
    nextinuse = inuse_bit_at_offset(nextchunk, nextsize);

    // ██ nextchunk █ free █████
    if (!nextinuse) {
        unlink(av, nextchunk, bck, fwd);
        size += nextsize;
    } else
clear_inuse_bit_at_offset(nextchunk, 0);

    // ██████████ unsorted bin
    bck = unsorted_chunks(av);
    fwd = bck->fd;

    // ██ unsortedbin ███
    if (__glibc_unlikely (fwd->bk != bck))
    {
        errstr = "free(): corrupted unsorted chunks";
        goto errout;
    }
    p->fd = fwd;
    p->bk = bck;
    if (!in_smallbin_range(size))
    {
        p->fd_nextsize = NULL;
        p->bk_nextsize = NULL;
    }
    bck->fd = p;
    fwd->bk = p;

    set_head(p, size | PREV_INUSE);
    set_foot(p, size);

    check_free_chunk(av, p);
}

else {

```

```

    size += nextsize;
    set_head(p, size | PREV_INUSE);
    av->top = p;
    check_chunk(av, p);
}

// ■■ free ■■■■ unsorted bin ■ size(■■■■chunk ■■■■) ■■■■ FASTBIN_CONSOLIDATION_THRESHOLD ■■■■ malloc_consolidate
if ((unsigned long)(size) >= FASTBIN_CONSOLIDATION_THRESHOLD) {
    if (have_fastchunks(av))
        malloc_consolidate(av);

    if (av == &main_arena) {
#ifdef MORECORE_CANNOT_TRIM
        if ((unsigned long)(chunksize(av->top)) >=
            (unsigned long)(mp_.trim_threshold))
            systrim(mp_.top_pad, av);
#endif
    } else {
        /* Always try heap_trim(), even if the top chunk is not
           large, because the corresponding heap might go away. */
        heap_info *heap = heap_for_ptr(top(av));

        assert(heap->ar_ptr == av);
        heap_trim(heap, mp_.top_pad);
    }

    if (! have_lock) {
        assert (locked);
        (void)mutex_unlock(&av->mutex);
    }
}
/*
   If the chunk was allocated via mmap, release via munmap().
*/

```

大概流程

- 首先做了一些检查, $p \neq \text{top_chunk}$, $p \rightarrow \text{size}$ 不能越界, 限制了 $\text{next_chunk} \rightarrow \text{size}$, p 要处于 `inuse` 状态 (通过 $\text{next_chunk} \rightarrow \text{pre_inused}$ 判断)
- 接着判断 p 的前后相邻块是不是 `free` 状态, 如果是就合并
- 根据此次拿到的 `unsorted bin` 的大小, 如果 $\text{size} \geq \text{FASTBIN_CONSOLIDATION_THRESHOLD}$ 就会触发 `malloc_consolidate`

如果 p 是通过 `mmap` 获得的, 就通过

```
munmap_chunk (p);
```

释放掉他

Check In Glibc

函数名	检查	报错信息
<code>unlink</code>	$p \rightarrow \text{size} == \text{nextchunk} \rightarrow \text{pre_size}$	<code>corrupted size vs prev_size</code>
<code>unlink</code>	$p \rightarrow \text{fd} \rightarrow \text{bk} == p$ 且 $p \rightarrow \text{bk} \rightarrow \text{fd} == p$	<code>corrupted double-linked list</code>
<code>_int_malloc</code>	当从 <code>fastbin</code> 分配内存时, 找到的那个 <code>fastbin</code> chunk 的 size 要等于其位于的 <code>fastbin</code> 的大小, 比如在 <code>0x20</code> 的 <code>fastbin</code> 中其大小就要为 <code>0x20</code>	<code>malloc():memory corruption (fast)</code>
<code>_int_malloc</code>	当从 <code>smallbin</code> 分配 chunk(victim) 时, 要求 $\text{victim} \rightarrow \text{bk} \rightarrow \text{fd} == \text{victim}$	<code>malloc(): smallbin double linked list corrupted</code>
<code>_int_malloc</code>	当迭代 <code>unsorted bin</code> 时, 迭代中的 chunk (cur) 要满足, $\text{cur} \rightarrow \text{size}$ 在 $[2 * \text{SIZE_SZ}, \text{av} \rightarrow \text{system_mem}]$ 中	<code>malloc(): memory corruption</code>
<code>_int_free</code>	当插入一个 chunk 到 <code>fastbin</code> 时, 判断 <code>fastbin</code> 的 head 是不是和 释放的 chunk 相等	<code>double free or corruption (fasttop)</code>
<code>_int_free</code>	判断 $\text{next_chunk} \rightarrow \text{pre_inuse} == 1$	<code>double free or corruption (!prev</code>

来源

[heap-exploitation](#)

各种漏洞原理及利用

通用的信息泄露思路

当 chunk 处于 free 状态时，会进入 bin 里面，其中的 fd 和 bk 可以用于信息泄露

- 分配两个 0x90 的 chunk(p0, p1)
- 释放掉 p0, p0 会进入 unsorted bin
- 分配 0x90 的 chunk,再次拿到 p0, 在 malloc 的实现中不会对这些指针进行清空，就可以泄露

如果分配后的内存被 memset 清空后，就需要利用一些其他的漏洞才能利用。

Unsorted bin 用于泄露 libc

fastbin 用于 泄露 heap 地址

Unlink 利用

原理

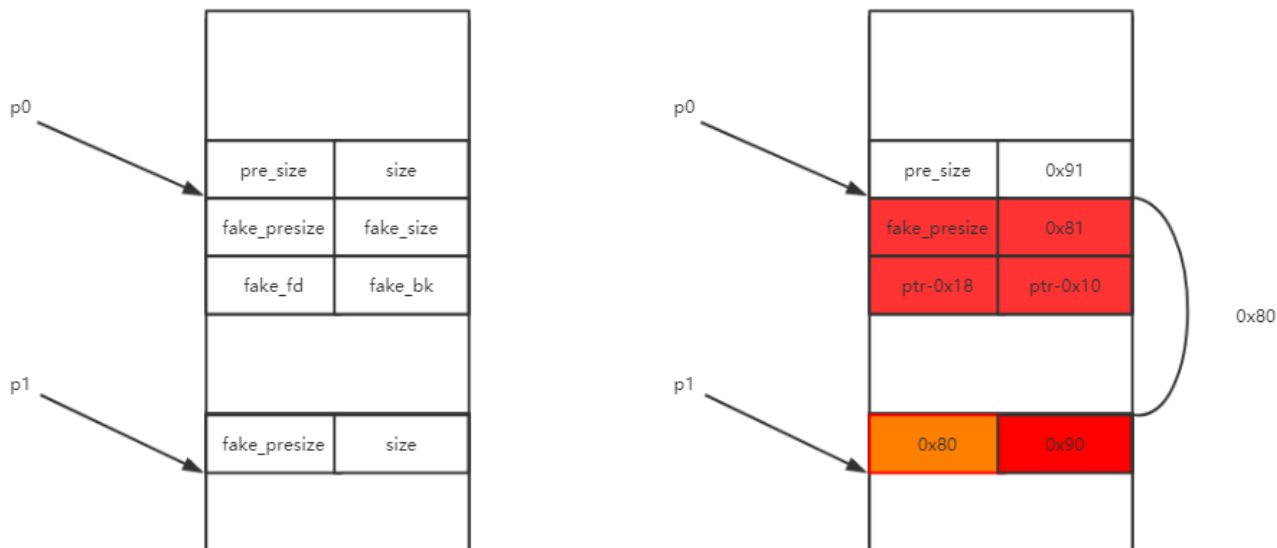
在把 chunk 从 bins 拿下来时会触发 unlink 操作

```
/* 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) {
                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;
            }
        }
    }
}
```

如果我们可以伪装 fd 和 bk 过掉 unlink 的检查，就可以实现 4 字节写

利用

首先利用其它的漏洞伪造下面的内存布局



先知社区

- `p0 = malloc(0x80), p1 = malloc(0x80), ptr = p0`
- 此时 `free(p1)` , 发现 `p1` 所在 chunk 的 `pre_size = 0` , 表明前一个 chunk 已经 `free` , 于是向前合并
- 通过 `p1 - 0x10 - 0x80` (`chunk_addr - pre_size`) , 找到前面已经释放的 chunk , 也就是 我们 伪造的 fake chunk `p1`
- 然后进行 `unlink` , 实现 `*ptr = ptr-0x18`

Fastbin Attack 总结

原理

Fastbin 在分配 chunk 时, 只检查 `p->size&0xffffffff000` 是否满足等于的 fastbin 的大小, 而且不检查指针是否对齐。所以我们只要找到 size 为 fastbin 的范围, 然后修改 位于 fastbin 的 chunk 的 fd 到这, 分配几次以后, 就可以分配到这个位置

利用方式

利用 libc 中的 现有的 数据

`__malloc_hook` 附近

64位下在 `__malloc_hook - 0x23 + 0x8` 处的值为 `p64(0x7f)` , 这些值可以通过 `gdb + hexdump` 找找

然后想办法修改 位于 `0x70` 的 fastbin 的 chunk 的 fd 为 `__malloc_hook - 0x23` , 然后分配几次 `0x70` 的 chunk 就可以修改 `__malloc_hook`

`main_aren->fastbinY` 数组

该数组用于存放 指定大小的 fastbin 的表头指针, 如果为空则为 `p64(0)` , 而堆的地址基本 是 `0x5x` 开头的 (其在内存就是 `xx xx..... 5x`) , 此时如果在 `main_aren->fastbinY` 的 相邻项为 `0x0` (相邻大小的 fastbin), 就会出现 `5x 00 00 00...` , 所以就可以出现 `0x0000000000000005x` , 可以把它作为 fastbin 的 size 进行 fastbin attack , 不过作为 fastbin attack 的 size 不能为 `0x55`

于是想办法修改 位于 `0x50` 的 fastbin 的 chunk 的 fd 为 `__malloc_hook - 0x23` , 然后分配几次 `0x50` 的 chunk 就可以分配到 `main_aren` , 然后就可以修改 `main_aren->top` 。

std* 结构体

在 `std*` 类结构体中有很多字段都会被设置为 `0x0` , 同时其中的某些字段会有 libc 的地址大多数情况下 libc 是加载在 `0x7f...` , 配合着 `std*` 中的 其他 `0x0` 的字段, 我们就可以有 `p64(0x7f)` , 然后修改 位于 `0x70` 的 fastbin 的 chunk 的 fd 为该位置即可。

```
pwndbg> x/28xg stdin
0x7ffff7dd18e0 <_IO_2_1_stdin_>:      0x00000000fbad2088      0x000055555555757042
0x7ffff7dd18f0 <_IO_2_1_stdin_+16>:    0x000055555555757043      0x000055555555757040
0x7ffff7dd1900 <_IO_2_1_stdin_+32>:    0x000055555555757040      0x000055555555757040
0x7ffff7dd1910 <_IO_2_1_stdin_+48>:    0x000055555555757040      0x000055555555757040
0x7ffff7dd1920 <_IO_2_1_stdin_+64>:    0x000055555555758040      0x0000000000000000
0x7ffff7dd1930 <_IO_2_1_stdin_+80>:    0x0000000000000000      0x0000000000000000
0x7ffff7dd1940 <_IO_2_1_stdin_+96>:    0x0000000000000000      0x0000000000000000
0x7ffff7dd1950 <_IO_2_1_stdin_+112>:   0x0000000000000000      0xffffffffffffffff
0x7ffff7dd1960 <_IO_2_1_stdin_+128>:   0x0000000000000000      0x00007ffff7dd3790
0x7ffff7dd1970 <_IO_2_1_stdin_+144>:   0xffffffffffffffff      0x0000000000000000
0x7ffff7dd1980 <_IO_2_1_stdin_+160>:   0x00007ffff7dd19c0      0x0000000000000000
0x7ffff7dd1990 <_IO_2_1_stdin_+176>:   0x0000000000000000      0x0000000000000000
0x7ffff7dd19a0 <_IO_2_1_stdin_+192>:   0x00000000ffffffff      0x0000000000000000
0x7ffff7dd19b0 <_IO_2_1_stdin_+208>:   0x0000000000000000      0x00007ffff7dd06e0
pwndbg> x/4xg 0x7ffff7dd1985-8
0x7ffff7dd197d <_IO_2_1_stdin_+157>:   0xfff7dd19c0000000      0x000000000000007f
0x7ffff7dd198d <_IO_2_1_stdin_+173>:   0x0000000000000000      0x0000000000000000
pwndbg>
```



自己构造 size

利用 unsorted bin attack 往 __free_hook 构造 size

我们知道如果我们可以 修改 unsorted bin 的 fd 和 bk, 在对 unsorted bin 拆卸的时候 我们就能实现

```
*(bk + 0x10) = main_aren->unsorted_bin
```

利用这个我们就能往任意地址写入 main_aren 的地址, 由于 libc 的地址基本都是 0x7fxxxxx, 所以写完以后我们就可以在 __free_hook 的前面构造出 p64(0x7f), 可以作为 fastbin attack 的目标, 然后修改 __free_hook

有一个小坑要注意, 在 __free_hook-0x30 开始的 0x30 个字节是 _IO_stdfile_*_lock 区域, 用于 std* 类文件的锁操作, 这个区域的内存会被经常清零。

所以 unsorted bin attack 应该往上面一点, 比如 libc.symbols['__free_hook'] - 0x50

还有一点就是在进行 unsorted bin attack 以后, unsorted bin 链表就被破坏了, 所以 就只能通过 fastbin 或者 smallbin 进行内存的分配, 所以我们应该先劫持 fastbin 的 fd 到 目标位置, 然后触发 unsorted bin attack 写入 size, 最后进行 fastbin attack, 修改 __free_hook

利用 fastbin 往 main_aren 构造 size

首先分配 0x40 的 chunk p, 然后释放掉 p, 进入 0x40 的 fastbin

然后通过一些手段, 修改 p->fd = p64(0x71)

分配 0x40 的 chunk, 会拿到 p, 此时 main_aren->fastbinY 中 0x40 大小对应的项的值为 p64(0x71)

然后分配 0x71 的 chunk p2, 释放掉

修改 p2->fd 为 main_aren->fastbinY 的相应位置, 然后分配两次, 即可分配到 main_aren->fastbinY

然后通过修改 main_aren->top, 即可分配到 malloc_hook 或者 free_hook 等

□

Unsorted bin Attack

原理

因为 unsorted bin 的取出操作没有使用 unlink 宏, 而是自己实现的几行代码

```
bck = victim->bk;
...
unsorted_chunks (av)->bk = bck;
bck->fd = unsorted_chunks (av);
```

所以当我们控制了 victim 的 bk 时, 则 bk + 0x10 会被改写成 unsorted bin 的地址, 但是 unsorted bin 的 bk 也会被破坏, 下一次再到这里时就可能因为 victim->bk->fd 不可写而造成 SIGSEGV。

所以在触发 unsorted bin attack 以后就 只能 通过 fastbin 和 smallbin 来分配内存了(否则会进入 unsorted bin 的流程, 会报错), 所以在 触发 unsorted bin attack 需要把需要的内存布局好。

利用的方式

写 stdin->_IO_buf_end

在 glibc 中 scanf, gets 等函数默认是对 stdin 结构体进行操作。以 scanf 为例

- 在调用 scanf 获取输入时, 首先会把输入的东西复制到 [_IO_buf_base, _IO_buf_end], 最大大小为 _IO_buf_end - _IO_buf_base。
- 修改 unsorted bin 的 bck 为 _IO_base_end-0x10, 就可以使 _IO_base_end=main_arena+0x88, 我们就能修改很多东西了, 而且 malloc_hook 就在这里面。

_IO_list_all 和 abort 以及 修改虚表到 _IO_wstrn_jumps

原理

绕过虚表校验

其实就是对 house of orange 在 libc2.24 里面的再利用。在 libc2.24 里对 vtable 进行了校验。

对 vtable 进行校验的函数是 IO_validate_vtable

```
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;
}
```

就是保证 vtable 要在 __stop__libc_IO_vtables 和 __start__libc_IO_vtables 之间。

这里的目标就是 _IO_wstrn_jumps, 这个也是一个 vtable, 能够满足 IO_validate_vtable 的校验。

在 _IO_wstrn_jumps 有一个有趣的函数 IO_wstr_finish, 位于 libc.symbols['_IO_wstrn_jumps'] + 0x10

```
void __fastcall IO_wstr_finish(_IO_FILE_2 *fp, int dummy)
{
    _IO_FILE_plus *fp_ = fp; // rbx
    wchar_t *io_buf_base; // rdi

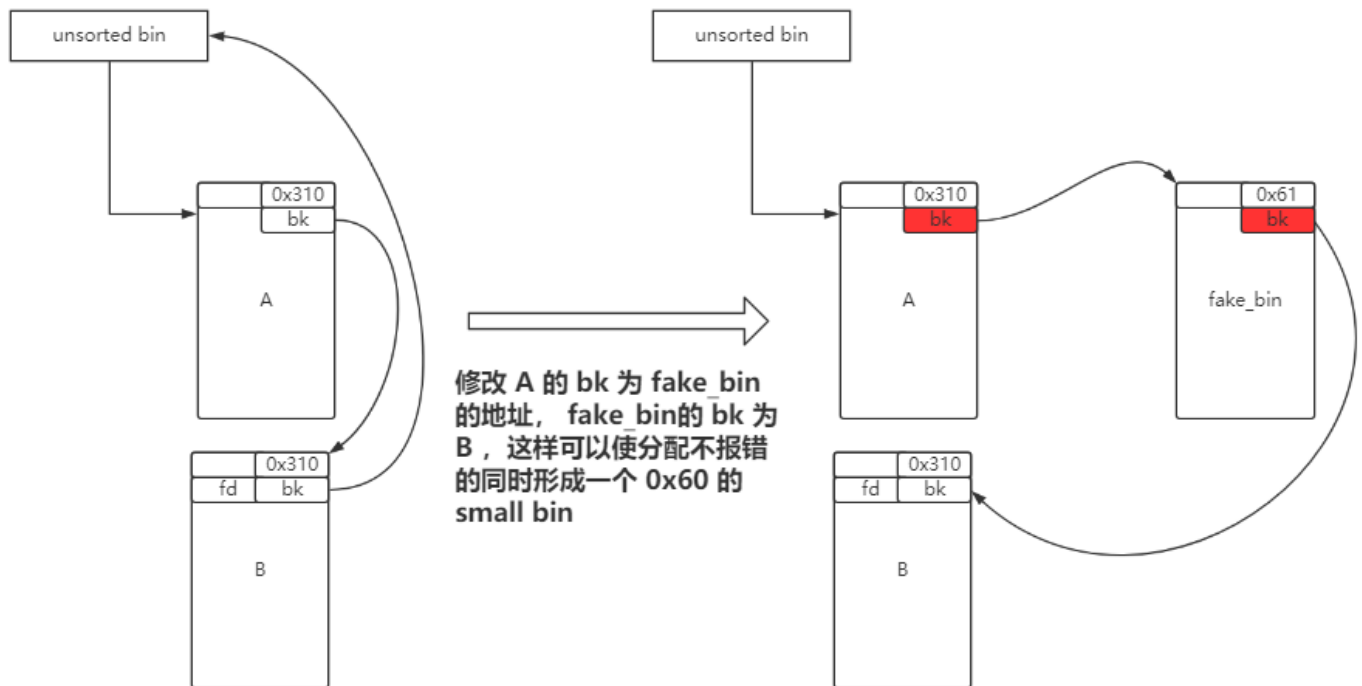
    fp_ = fp;
    io_buf_base = fp->_wide_data->_IO_buf_base;
    if (io_buf_base && !(fp->file._flags2 & 8))
        (fp->file._IO_read_ptr)(io_buf_base, *&dummy); // call    qword ptr [fp+0E8h]
    fp->file._wide_data->_IO_buf_base = 0LL;
    _GI__IO_wdefault_finish(fp_, 0);
}
```

我们把 fp->_wide_data 改成 fp, 然后设置 fp->_IO_buf_base 设置为 /bin/sh 的地址, fp->file._IO_read_ptr (fp+0xe8) 改成 system 的地址, 其他字段根据 check 设置好以便过掉检查, 之后调用该函数就会 system('/bin/sh')

利用方案举例

以 34c3ctf-300 为例, 程序限制只能分配 0x310 的 chunk, 这里利用 unsorted bin 遍历的缺陷, 伪造了一个 0x60 的 smallbin, 为后续做准备。

- 首先分配 4 个 0x310 的 chunk ■A X B K■, 释放 A , B 此时 A , B 均进入 unsorted bin ,并且通过bk 链接起来
- 修改 A->bk 为 fake_bin 的地址, 并且 设置 fake_bin->size=0x61 and fake_bin->bk = B, 此时 unsorted bin 的链表其实有 3 项。
- 分配一个 0x310 的 chunk ,此时 A 位于链表首部, 且大小刚好, 分配 A , 并且 把 fake_bin 置于链表首部
- 再次分配一个 0x310 的 chunk , 此时 fake_bin 位于链表首部, 大小不够于是把 fake_bin 放到 smallbin[4], 然后继续遍历 , 分配到 B , 至此在 smallbin[4] 就存有 fake_bin 的地址



fake_bin 的内容为 (从 chunk 的开始地址开始)

```
payload = p64(0xfbad2084)    #■■■■ File ■■■■■■■■■fp->_flag
payload += p64(0x61)
payload += p64(0xb00bfce)
payload += p64(B_addr) # bk ■■■■ B ■■■
payload += p64(0x0)        # fp->_IO_write_base
payload += p64(libc_base + sh_addr)    # fp->_IO_write_ptr
payload += p64(libc_base + sh_addr)    # fp->wide_data->buf_base
payload += "A"*60
payload += p64(0x0)        # fp->_flags2
payload += "A"*36
payload += p64(fake_bin)    # fp->_wide_data ■■■■ fake_bin, ■■ fake_bin
payload += "A"*24
payload += p64(0x0)        # fp->_mode
payload += "A"*16
payload += p64( libc.symbols['_IO_wstrn_jumps'] + 0x10 - 0x18) # fake vtable
payload += "A"*8
payload += p64(libc_base + libc.symbols['system'])    # ((_IO_strfile *) fp)->s._free_buffer
```

- 然后利用 unsorted bin attack 修改 __IO_list_all 为 main_aren+88
- 触发 abort (malloc_printerr内部会调用), 就会触发 _IO_flush_all_lockp,根据 __IO_list_all 和 __chain , 遍历调用 _IO_OVERFLOW (fp, EOF) (其实就是 (fp->vtable + 0x18)(fp, EOF)
- __IO_list_all->__chain 位于 smallbin[4] , 所以遍历第二次可以对 fake_bin 进行 _IO_OVERFLOW (fp, EOF), 此时就会调用 IO_wstr_finish , 此时 fake_bin 中的相关数据已经设置好, 最后会执行 system("/bin/sh")

参考

[34c3ctf-300](#)

[Pwn with File结构体 四](#)

组合 fastbin attack

方案一

- 把 bk 改成 global_max_fast-0x10 触发 unsorted bin attack 后, global_max_fast 会被修改成一个很大的值 (指针), 所以之后的 内存 分配和释放 都会按 fastbin 来
- 之后看情况进行 伪fastbin attack

方案二

把 bk 改成 libc.symbols['__free_hook'] - 0x50 触发 unsorted bin attack 后, free_hook 前面就会出现 p64(0x7f), 之后就可以通过 fastbin attack 修改 free_hook

参考

[0ctf-2016-zerostorage](#)

结合 largebin 和 _dl_open_hook

原理

在 遍历 unsoted bin 时, 是通过 bk 指针 进行遍历

```
for (;;)
{
    int iters = 0;
    //victim = unsorted_chunks (av)->bk
    while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av)) // ■■ unsorted bin
    {
        bck = victim->bk;
        .....
        .....
        .....
        /* remove from unsorted list */
        unsorted_chunks (av)->bk = bck; //unsorted_chunks (av)->bk = victim->bk->bk
        bck->fd = unsorted_chunks (av);
        .....
        .....
        .....
    }
}
```

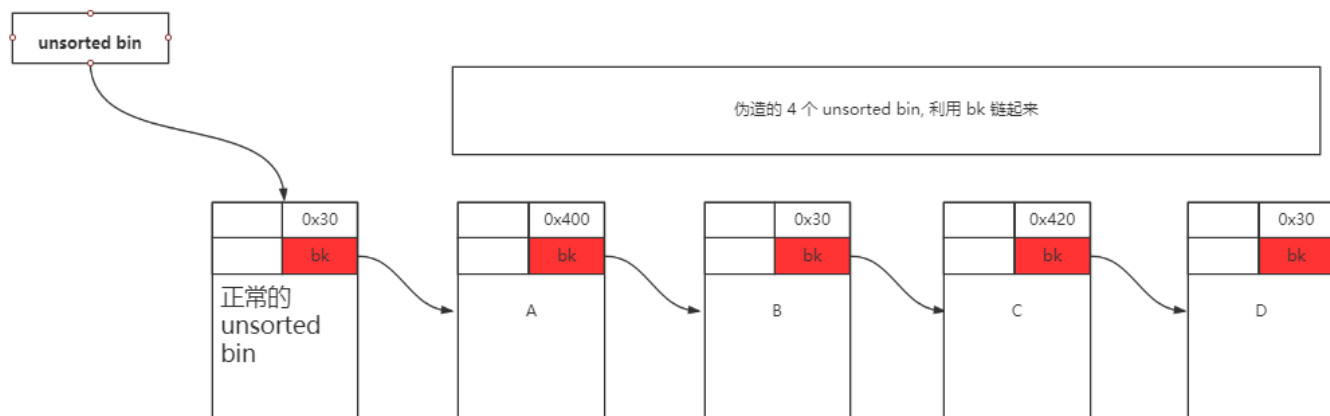
所以通过修改 bk 来伪造 unsorted bin 是可行的

同时在 遍历 unsorted bin 把 chunk 放入 largebin 的过程中, 也没有什么检查, 于是可以利用 把 chunk 放入 largebin 的过程 往 任意地址写入 chunk 的地址。

PS: 因为要伪造 unsorted bin , 需要我们可以拿到 heap 的基地址

大体的思路

- 在堆上通过修改 unsorted bin 的 bk 指针 伪造几个 unsorted bin(A B C D)■(0x400, 0x30, 0x420, 0x30)
- 分配 0x30, A 进入 largebin, B 被分配
- 修改 A->bk = _dl_open_hook - 0x10 and A->bk_nextsize = _dl_open_hook - 0x20
- 分配 0x30, C 进入 largebin, 会导致 A->bk->fd = C , A->bk_nextsize->fd_nextsize = C (其实就是 *_dl_open_hook = C)
- 此时 _dl_open_hook 指针被改成 C 的地址, 然后在 C 中设置 p64(libc.symbols['__libc_dlsym'] + 4)+p64(one_gadget)+p64(one_gadget) , 伪造 dl_open_hook 结构体。
- 后面的执行过程会调用 _dl_open_hook, 就会调用 __libc_dlsym + 4, 这里面会 跳转到 dl_open_hook 结构体偏移 8 的值处, 也就是 one_gadget 的地址



先知社区

参考

[Octl 2018 babyheap challenge exploit](#)

特定写权限的利用

可写 main_aren

通过一些 fastbin 攻击，我们可以分配到 main_aren，此时一般都是改写 main_aren->top

转换为写 __malloc_hook

malloc_hook -0x10 处存放的是指针，值很大，修改 main_aren->top 到这里，然后控制程序使得通过 top_chunk 分配，就可以分配到 malloc_hook

转换为写 __free_hook

在 free_hook-0xb58 处存放的也是一些地址，修改 main_aren->top 到这里，然后控制程序使得通过 top_chunk 分配几次内存（一次分配太多，会触发 sysmalloc，可以一次分配 0x90 多分配几次），我们就可以分配到 free_hook

可写 __malloc_hook

直接写 one_gadget

写入 one_gadget，不过触发的时候，用 malloc_printerr 来触发 malloc

此时用下面这样的 one_gadget [rsp+0x50]

```
0xef6c4 execve("/bin/sh", rsp+0x50, environ)
constraints:
[rsp+0x50] == NULL
```

这样更稳定，成功率也高

通过 __realloc_hook 中转

__malloc_hook 和 __realloc_hook 是相邻的，且 __realloc_hook 在 __malloc_hook 的前面，所以基本上可以同时修改它们。

利用 one_gadget 时，对于栈的条件会有一些要求，利用 realloc 函数内部的跳转 到 __realloc_hook 之前的栈操作，加上栈中原有的数据，可以对栈进行跳转，以满足 one_gadget 的要求

```
realloc      proc near      ; DATA XREF: LOAD:0000000000006BA0↑o

push        r15
push        r14
push        r13
push        r12
mov         r13, rsi
push        rbp
push        rbx
mov         rbx, rdi
sub         rsp, 38h
mov         rax, cs:__realloc_hook_ptr #■■■ __realloc_hook ■■■
mov         rax, [rax]
test        rax, rax
```

```

jnz     loc_848E8
test    rsi, rsi
jnz     short loc_846F5
test    rdi, rdi
jnz     loc_84960

```

代码中的 `push` 以及 `sub rsp, 38h` 都可用于对栈进行调整。

可以收先把 `__malloc_hook` 设置为 `0x6363636363636363`，当程序断下来后，查看栈的情况，然后选择跳转的位置。

最后把 `malloc_hook` 设置为选择好的位置，`realloc_hook` 设置为 `one_gadget`，触发 `malloc`

可写 `__free_hook`

直接写 `one_gadget`

改成 `system` 函数的地址

然后 释放掉 内容为 `/bin/sh\x00` 的 chunk

可写 `std*` 结构体

`std*` 类结构体 定义是 `_IO_FILE_plus`，64 为大小为 `0xe0`

修改 `vtable` 指针

`libc <= 2.23`

`_IO_FILE_plus` 的最后一个字节就是 `vtable` 指针，修改 `vtable` 指针到一个可控数据可控的地址，在地址处填上 `one_gadget`，然后在调用一些输入输出函数时，就会触发。

如果是堆类题目可以 修改 `vtable` 指针到 `heap`，或者如果是通过 `fastbin` 分配到了 `std*`，那么可以修改 `vtable` 到 `std*` 的相应位置，只要保证马上要被调用的函数指针我们可控 即可

`libc > 2.23`

一般结合 `unsorted bin attack`，改到 `libc.symbols['_IO_wstrn_jumps'] + 0x10 - 0x18`，然后触发 `abort` 会调用 `_IO_OVERFLOW (fp, EOF)` 时就会调用 `_IO_wstr_finish(fp, EOF)`，通过设置 `fp` 的数据，就可以 `system("/bin/sh")`。

(: `fp` 为文件结构体的指针

Double Free

原理

程序把指针 `free` 之后没有对指针进行清空，出现了 悬垂指针。后续还可以对该指针进行 `free` 操作。

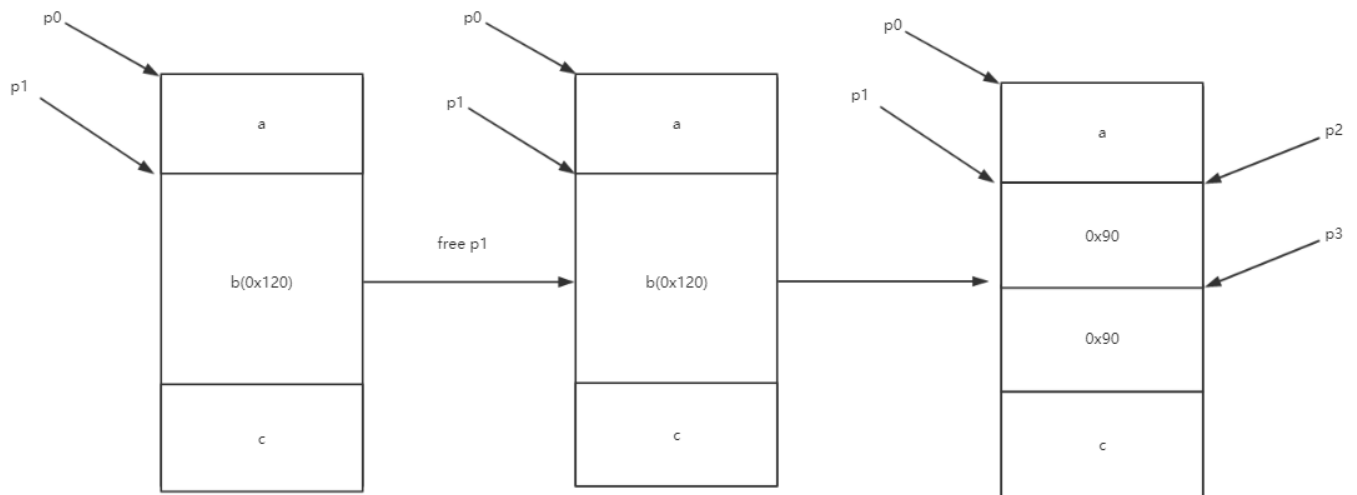
利用

基于 `pwnable.tw` 中的 `secretgard`

信息泄露

总的思路：大块 拆成 小块

- 分配一个 `0x120` 的 chunk, `p1` 指向 它。然后释放掉他
- 分配两个 `0x90` 的 chunk 重用刚刚 `free` 掉的 chunk, 可以发现此时 `p1==p2`
- 此时再次 `free(p1)`, 在 `p2->fd` 和 `p2->bk` 会写入 `main_aren` 的地址 (`free` 之后大小大于 `fastbin` 的范围，进入 `unsorted bin`)
- 然后打印 `p2` 的内容就可以拿到 `libc` 的地址



此时B是一个0x120的 unsorted bin

p2 = malloc(0x80)
p3 = malloc(0x80)

先知社区

Overlap chunk + unlink

总的思路：小块 融合成 大块

- 首先分配两个 0x90 的 chunk (p0, p1)，然后释放掉，会进行合并，形成一个 0x120 的 unsorted bin
- 然后分配一个 0x120 的 chunk (p2)，则 p0=p2，此时 p0 所在的 chunk 可以包含 p1 的 chunk
- 然后在 p0 所在的 chunk 伪造一个 free chunk，设置好 fd 和 bk，然后释放 p1 触发 unlink

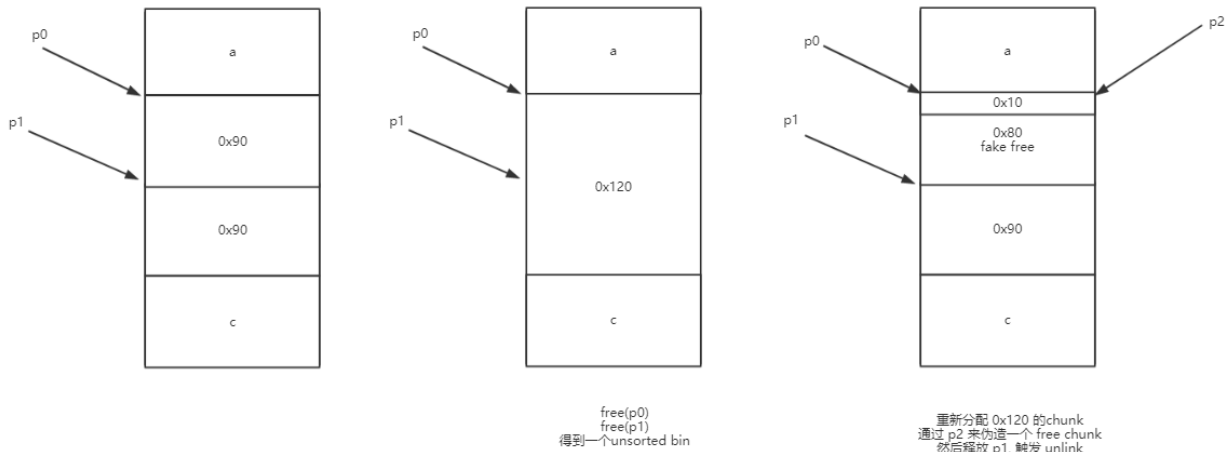
```
add(0x80) # pz
add(0x80) # p0
add(0x80) # p1
add(0x80) # px
```

```
del(1)
del(2)
```

```
add(0x110) # p2
```

```
payload = p64(0)          # p1's 0000
payload += p64(0x81)      # fake chunk size
payload += p64(ptr - 0x18) # fd, ptr--->p0 + header_size
payload += p64(ptr - 0x10) # bk
payload += 'a' * (0x80 - len(payload))
payload += p64(0x80)      # pre_size ----- 0000 chunk p1
payload += p64(0x80)      # size 00 pre_inused=0
payload += 'b' * 0x70
payload += p64(0x80)
payload += p64(0x21)      # size 00 pre_inused=1 ---- p1-->next_chunk, 00 double free 00
edit(2,payload)          # fake chunk
```

```
# p1 00 chunk->pre_inused=0, 0000
# 00 fake chunk 00 unlink
# ptr--->p0 + header_size, 00 *ptr = ptr-0x18
del(1)
```



修改 __malloc_hook

一般 malloc 触发的方式, one_gadgets 由于限制条件不满足, 很可能会失败

可以使用 malloc_printerr 触发, 此时恰好 [esp+0x50]=0

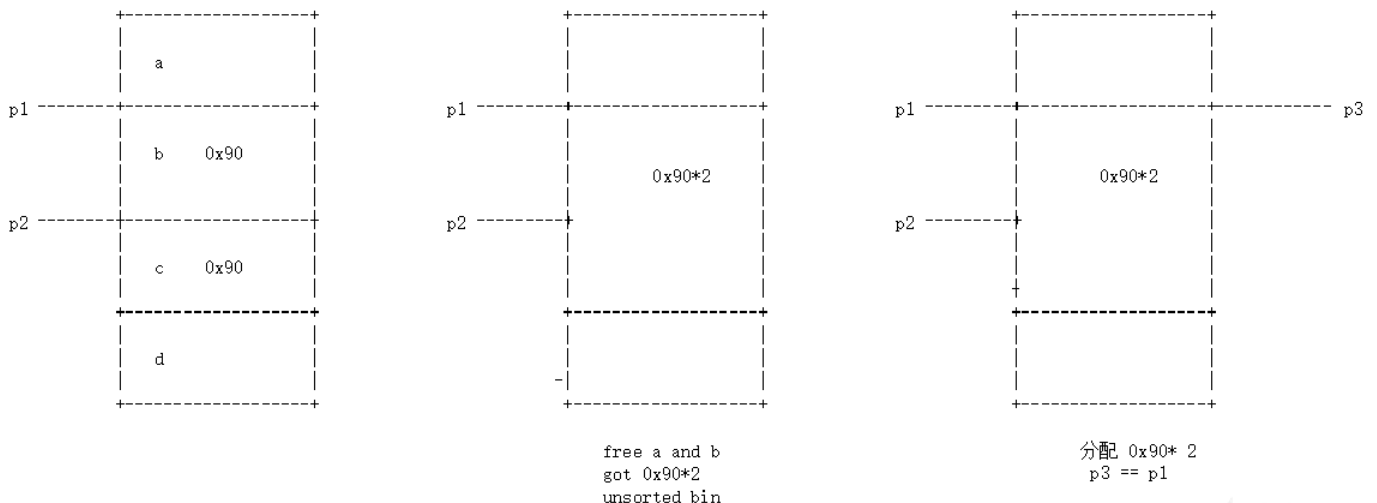
__malloc_hook - 0x23 + 0x8 的内容为 0x000000000000007f, 可以用来绕过 fastbin 分配的检查

可以 gdb + hexdump 找到类似的位置来伪造 fastbin

Overlap Chunk + Fastbin Attack

总的思路: 小块 融合成 大块, 分配大块操纵小块

- 首先分配两个 0x90 大小的 chunk (p0, p1)
- 释放掉它们, 合并成一个 0x120 的 unsorted bin
- 分配 0x120 的 chunk (p3), p3==p1, 而且此时通过 p3 可以修改 p2 的 chunk, Overlap Chunk 完成
- 修改 p->size = 0x71 p = p2-0x10, p 为 p2 所在 chunk 的地址
- 修改 p + 0x70 为 p64(0x70) + p64(0x41), 设置 pre_inused = 1, 使得后面 free(p2) 绕过 double free 检测
- 此时 free(p2), p2 进入 0x70 大小的 fastbin
- 再次 free(p1) (此时 p1 所在 chunk 的 size 为 0x120), 得到一个 0x120 的 unsorted bin
- 再次分配 0x120 的 chunk (p4), p4==p1
- 通过 p4 可以修改 p2 指向的 chunk 的 fd 为 __malloc_hook - 0x23 (此时 p2 的 chunk 已经在 0x70 的 fastbin 里面)
- Fastbin Attack 开始, 分配两次, 可以得到 p6 = __malloc_hook - 0x13
- 然后修改 __malloc_hook



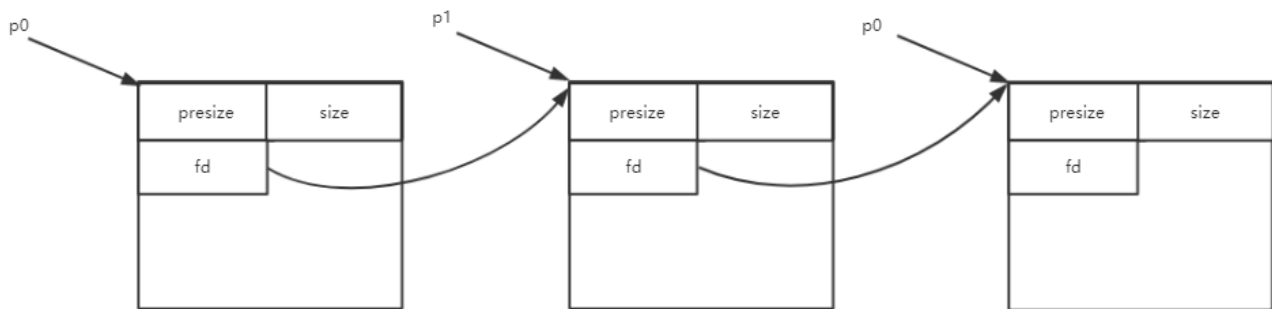
Overlap chunk + fastbin attack + 修改 top chunk

- 首先通过上面的 Overlap chunk 我们可以修改 p2 的 chunk 的内容
- 修改 chunk->size = 0x41, 注意设置好 chunk->nextchunk 的 pre_inused 位 避免过不了 double free 检查
- free(p2), 此时 p2 的 chunk 进入 0x40 的 fastbin
- free(p3), malloc(0x110), 可以再次修改 p2 chunk, 修改 chunk->size = 0x41 and chunk->fd = 0x71
- malloc(0x30), 此时 main_aren->fastbinY 中会有一项的值为 p64(0x71)
- 再次 free(p3), malloc(0x110), 修改 p2 chunk, chunk->size = 0x71
- free(p2), 此时 p2 的 chunk 进入 0x70 的 fastbin
- free(p3), malloc(0x110), 修改 p2 chunk, 设置 chunk->size = 0x71 and chunk->fd = 0x40 fastbinY 的地址附近
- 分配两次 0x70 的 chunk, 可以修改 main_aren->top 为 __malloc_hook - 0x10 (这里存的指针, 值很大)
- 然后使用 top chunk 进行分配, 就可以拿到 __malloc_hook

Fastbin dup+ Fastbin Attack

在把释放的块放进 fastbin 时, 会检测也 只检测 当前 free 的 chunk 和 fastbin 第一项 是否相同, 如果相同则报 double free 的错误。

- 首先 分配 2 个 0x70 的 chunk, p0, p1
- 释放 p0, p0 进入 0x70 大小的 fastbin, 此时 p0 为第一项
- 释放 p1, p1 进入 0x70 大小的 fastbin, 此时 p1 为第一项, p1->fd = p0
- 再次释放 p0, 此时 p1 为 fastbin 的第一项, 不会报错, p0 进入 fastbin, 此时 p0 为第一项
- 分配 0x70 的 chunk p2, p2 = p0, 设置 p2->fd = __malloc_hook - 0x23, 其实就是修改 p0->fd
- 此时 __malloc_hook - 0x23 成为 0x70 fastbin 的第 3 项
- 分配三个 0x70 的 chunk p3, p4, p5, p5 = __malloc_hook - 0x13
- 通过 p5 修改 __malloc_hook



修改 __free_hook

因为 free_hook 上方很大一块空间都是 \x00, 所以使用 fastbin attack 直接来修改它基本不可能, 可以迂回一下, 在 free_hook-0xb58 位置会存一些指针, 我们通过 fastbin attack 修改 main_aren->top, 到这里然后多用 top_chunk 分配几次, 就可以分配到 free_hook, 然后该 free_hook 为 system。

Fastbin dup + Fastbin Attack 修改 main_aren->top

- 首先利用 Fastbin dup 我们可以拿到实现修改 fastbin 中的块的 fd
- 由于在 fastbin 中 如果为空, 其在 main_aren->fastbinY 里面对应的值为 0x0, 而堆的地址基本是 0x5x 开头的 (其在内存就是 xx xx... 5x), 此时如果在 main_aren->fastbinY 的相邻项为 0x0, 就会出现 5x 00 00 00..., 所以就可以出现 0x000000000000005x, 可以把它作为 fastbin 的 size 进行 fastbin attack, 不过作为 fastbin attack 的 size 不能为 0x55
- 然后我们就可以修改 main_aren->top 为 free_hook-0xb58
- 之后多分配几次, 既可以分配到 free_hook
- 改 free_hook 为 system
- free 掉一个 内容为 /bin/sh\x00 的块

修改 _IO_FILE_plus 结构体的 vtable

在 libc 2.24 以下可修改 _IO_FILE_plus 的 vtable 指针到我们可控的位置, 进行虚表的伪造。

参考

[Pwnable.tw secretgard](https://pwnable.tw/secretgard/)

off by one

原理

在一些情况下我们可以往指定的 buf 中多写入 1 个字节的数据，这就是 off by one . 这种情况下可以进行利用的原因在于 调用 malloc 分配内存是要对齐的，64 位 0x10 字节对齐，32 位 8 字节对齐，下面均以64位进行说明。如果 malloc(0x28) 则会分配 0x30 字节的 chunk，除去 0x10 的首部，我们有 0x20 然后加上下一个 chunk 的 pre_size，我们就有 0x28 了，我们知道 pre_size 后面紧跟着就是 size，所以利用 off by one 可以 修改 下一个 chunk 的 size 字段，同时在 glibc 中的内存管理 非常依赖这个 size 字段，所以我们可以利用它做一些有趣的事情。

所以当程序中有类似这种不对齐的分配，就要小心 off by one

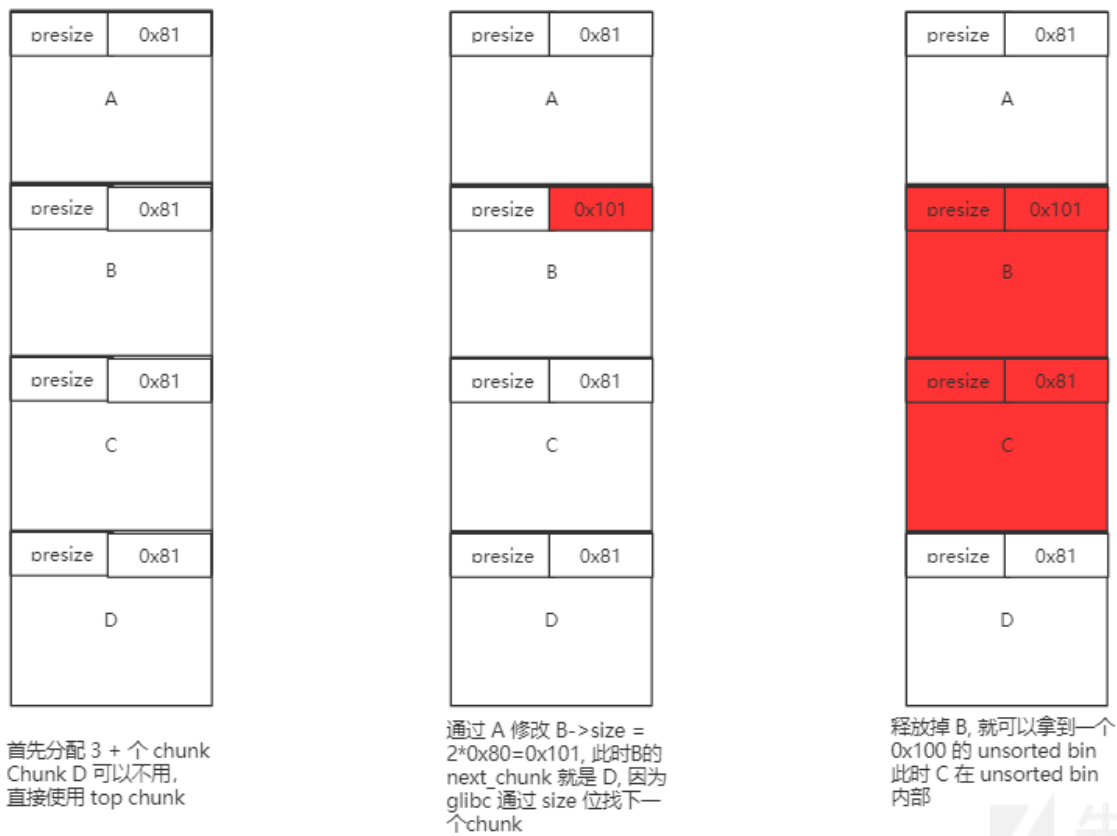
利用

普通off by one

在这种情况下，溢出的那个字节不受限制，此时的利用思路就是，多分配几个 chunk，然后利用第一个来溢出修改第二个 chunk 的 size (改大)，然后 free(chunk_2)，就可以 overlap chunk 3，要非常注意 in_used 位的设置

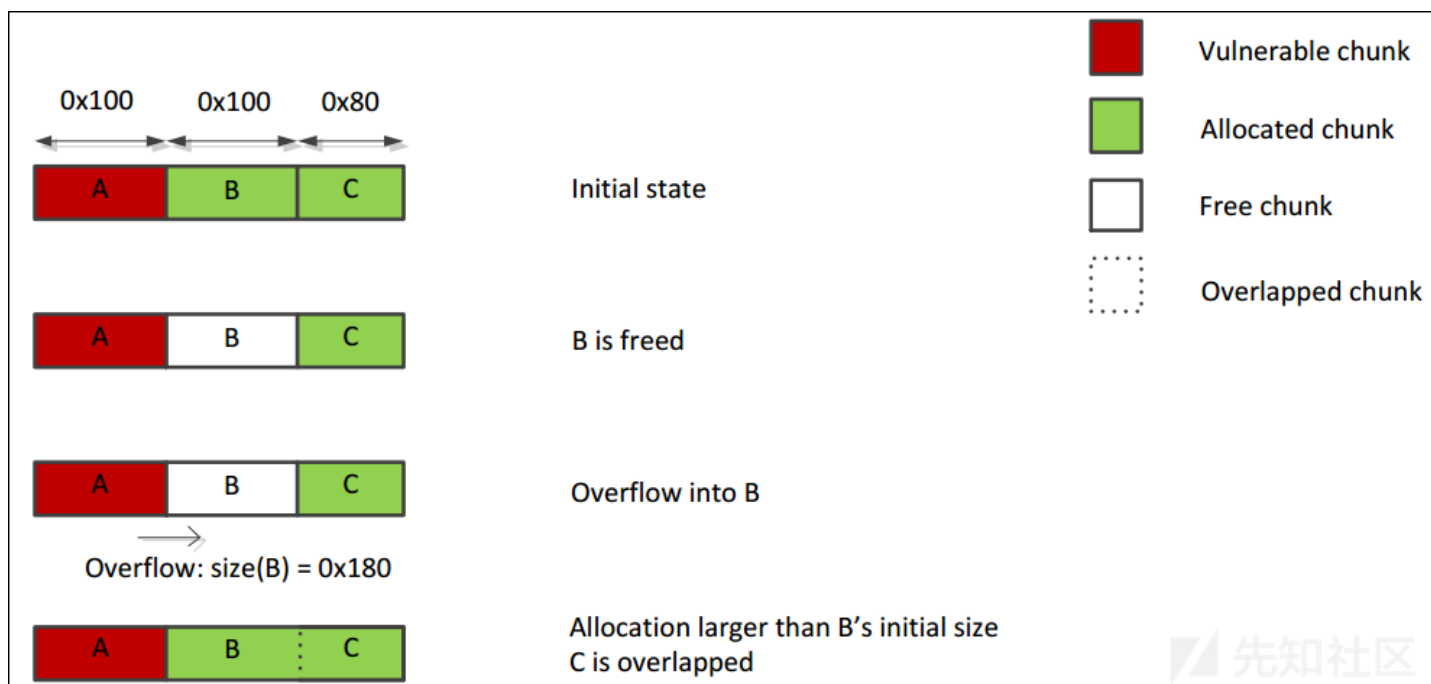
溢出 used 状态的 chunk

在 free 时可以获得包含 chunk 的 unsorted bin



溢出 free 状态的 chunk

因为malloc 再分配内存时 不会校验 unsorted bin 的 size 是否被修改



[Glibc Adventures-The Forgotten Chunks](#)

基于 Octf 2018 babyheap

信息泄露

- 首先 malloc 4 个 chunk, malloc(0x18)

```
allocate(0x18) # 0, 0x20 chunk
allocate(0x38) # 1, 0x40 chunk----> ■■■■■ 0x91
allocate(0x48) # 2, 0x50 chunk
allocate(0x18) # 3, 0x20 chunk
```

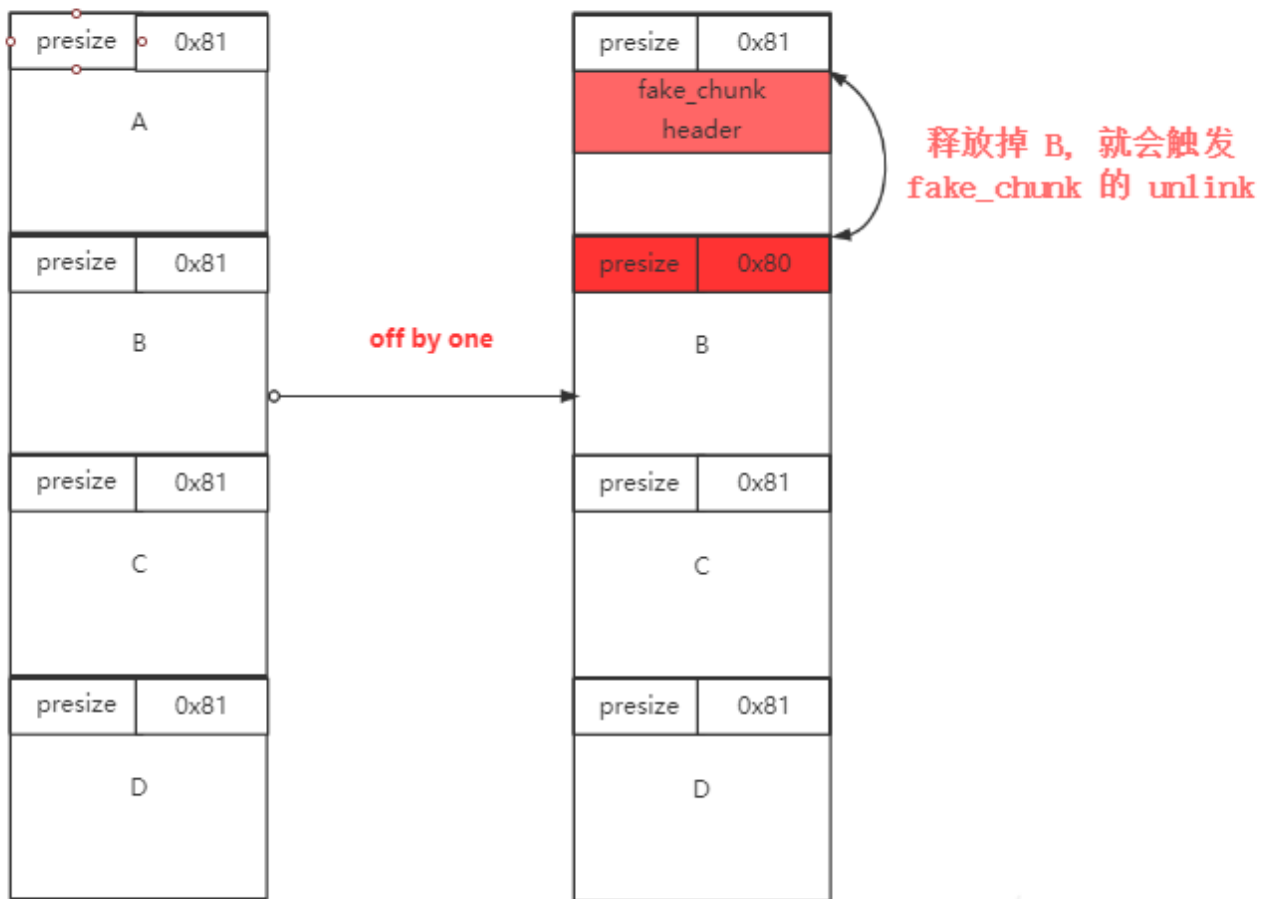
- 然后在 chunk 0 溢出一个字节, 修改 chunk 1 的 size 位为 0x91 (原来应该为 0x41), 这样一来通过 chunk 1 索引到的下一个 chunk 就是 $p + 0x90 = \text{chunk 3}$ (设 p 为 chunk 1 的地址)
- 此时释放 chunk 1, libc 会根据下一个 chunk (这里也就是 chunk 3) 的 `pre_inused` 位来检查是否 double free, 由于 chunk 2 原来并没有被释放, 所以 `pre_inused = 1`, 于是可以过掉检查, 此时得到一个 0x90 的 unsorted bin, 同时 chunk 2 在这个 unsorted bin 里面, overlap chunk 2
- 此时再次 malloc(0x38), 会使用 unsorted bin 进行切割, 所以在 chunk 2 的 fd, bk 处会写入 main_aren 的地址, 打印 chunk 2 的内容就可以 leak libc

漏洞利用

其实可以 overlap chunk 了, 就相当于获得了 ■■■■ 的能力, 我们可以任意修改 chunk 的数据, 此时可以使用 unlink, unsorted bin attack, fastbin attack。没有限制内存分配的大小, 使用 fastbin attack 即可

unlink

这种情况下的 unlink 应该比较简单, 在当前 chunk 伪造好 fd, bk 然后利用 off by one 修改下一个 chunk 的 `pre_size` (由于不对齐的分配, 这个区域其实属于当前 chunk) 和 `size` 的 `pre_inused` 为 0, 然后 free 掉下面那个 chunk, 就可以触发 unlink 了

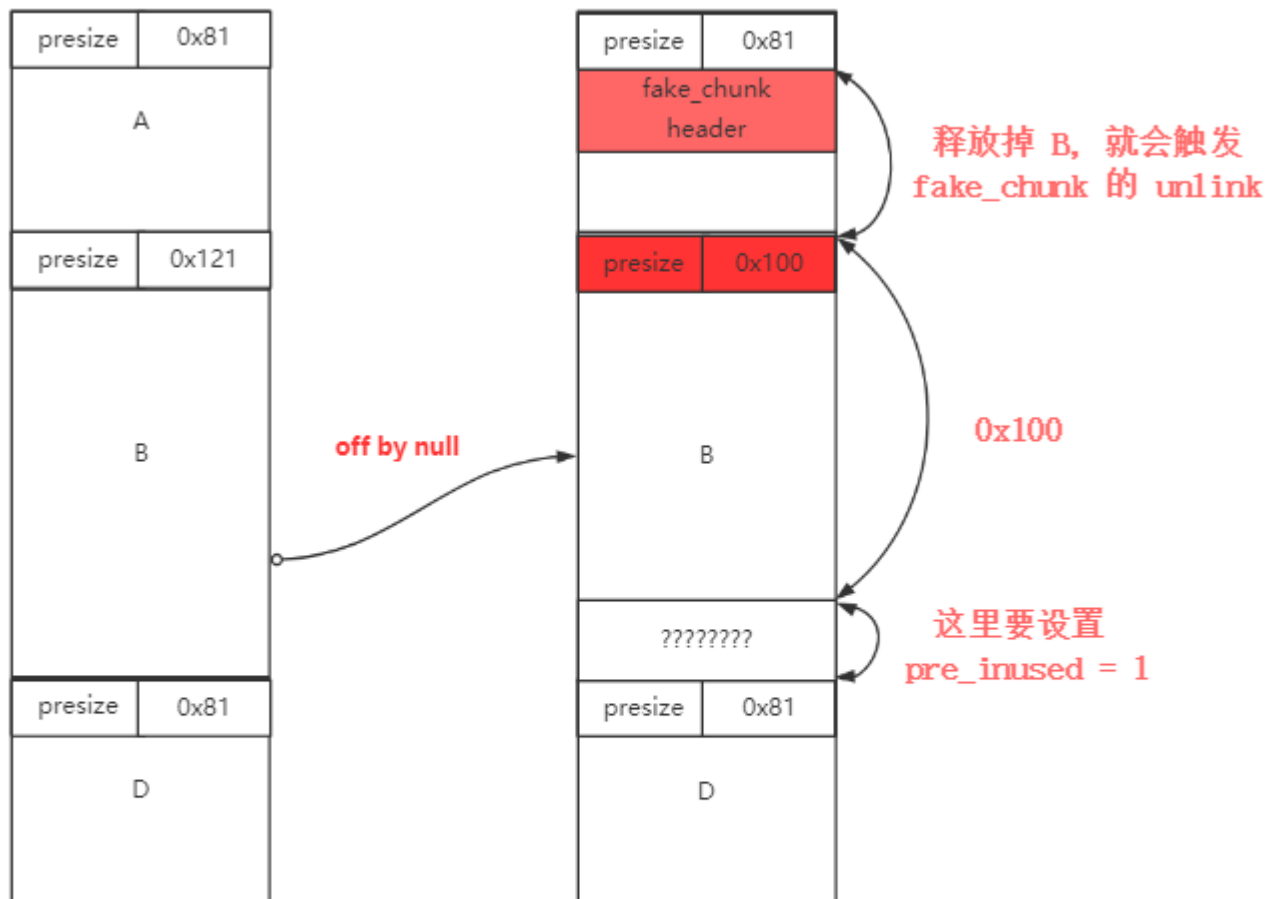


off by null

在这种情况下, 我们只能溢出 \x00 字节, 所以会把 size 变小 同时 inused 位 会被设置为 0

unlink

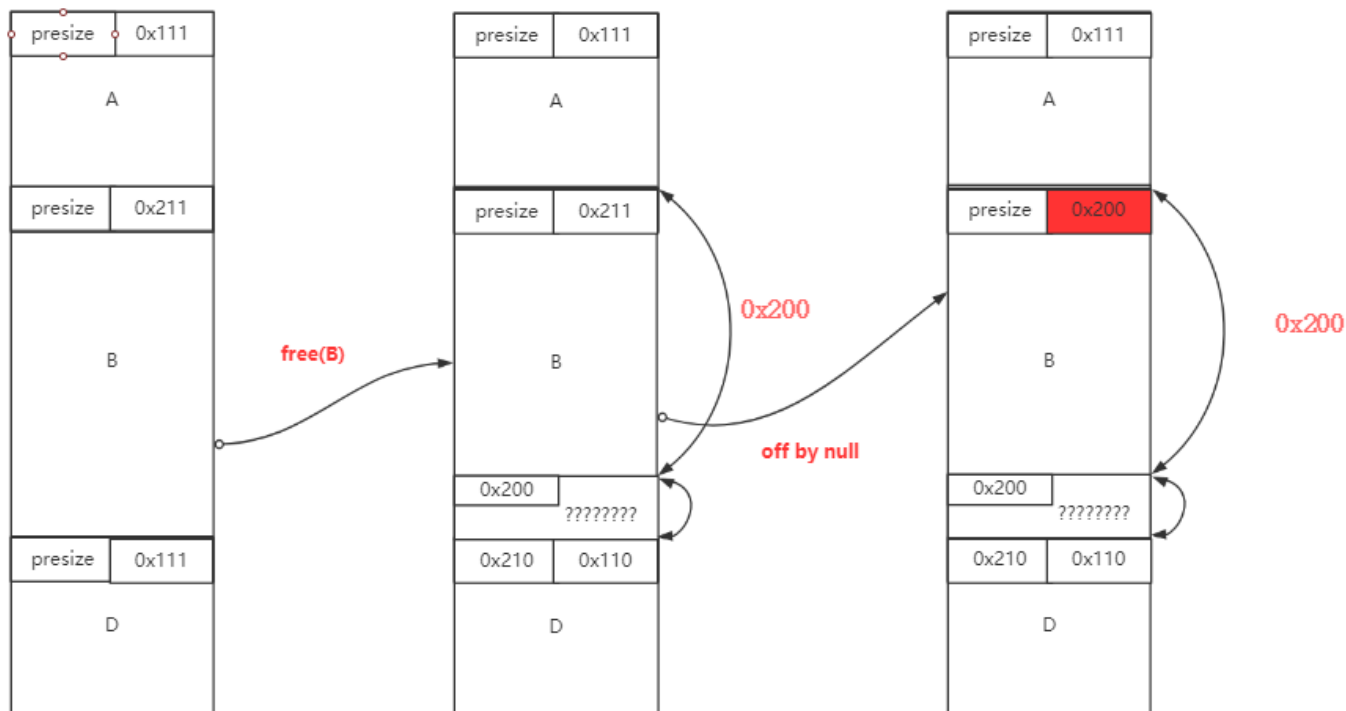
B + 0x100 处要设置好 p64(0x41) 关键是 pre_inused 位, free 的时候会检测这个位



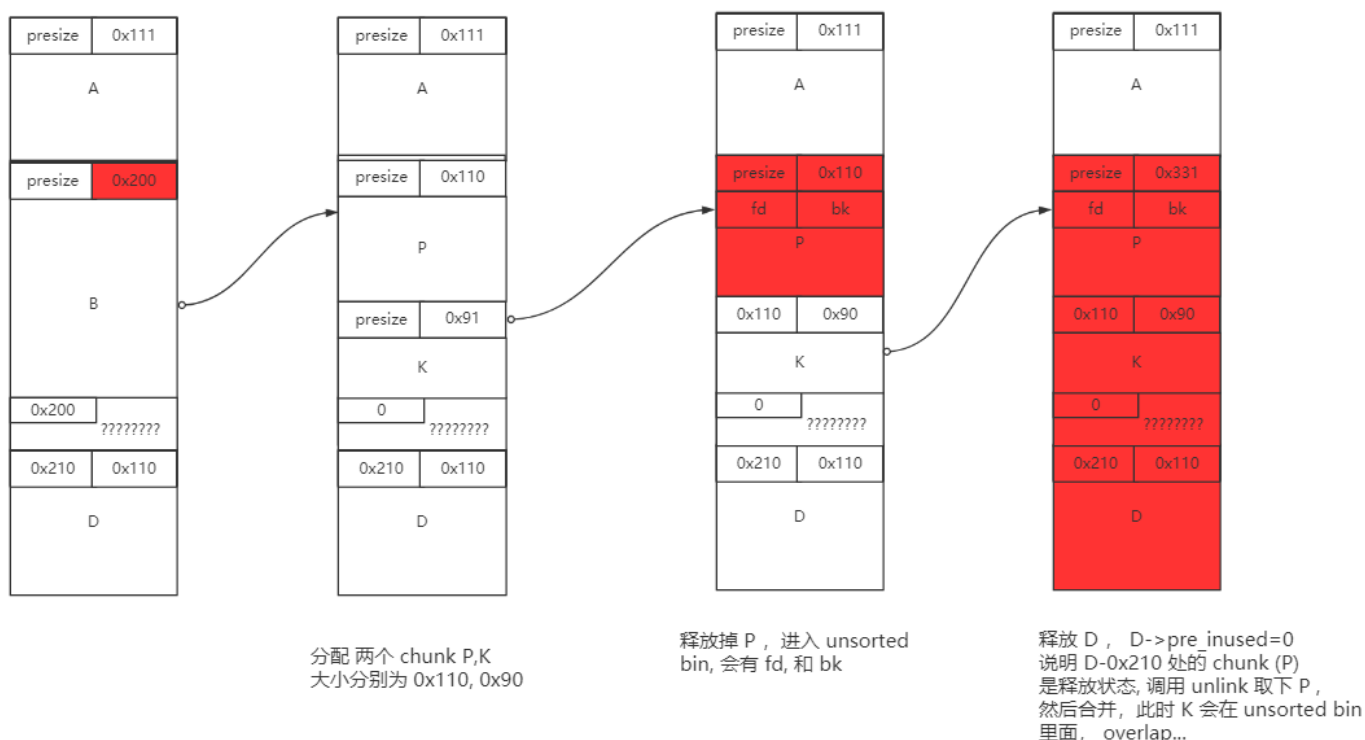
shrink free chunk size

布局过程

- 首先分配 3 个 chunk (A B D), 大小分别为 0x110 , 0x210, 0x110
- 然后 释放 B , 此时 D->pre_inused = 0 and D->pre_size = 0x210
- 修改 B+0x200 处为 p64(0x200) , 绕过新版 libc 的 chunksize(P) != prev_size(next_chunk(P)) 检查



- 然后分配两个 chunk (P, K), 大小为 0x110, 0x90
- 释放掉 P, 此时 P 会进入 unsorted bin, fd, bk 是有效的, 原因是 后面合并 D 时需要 unlink
- 释放 D, 发现 D->pre_inused=0, 说明前一个 chunk 已经 free, 需要合并。根据 pre_size 找到 P, 然后 unlink(P) 合并得到一个 0x330 的 unsorted bin, 此时 K 位于 unsorted bin 内部, overlap chunk done



布局过程中的一些 tips

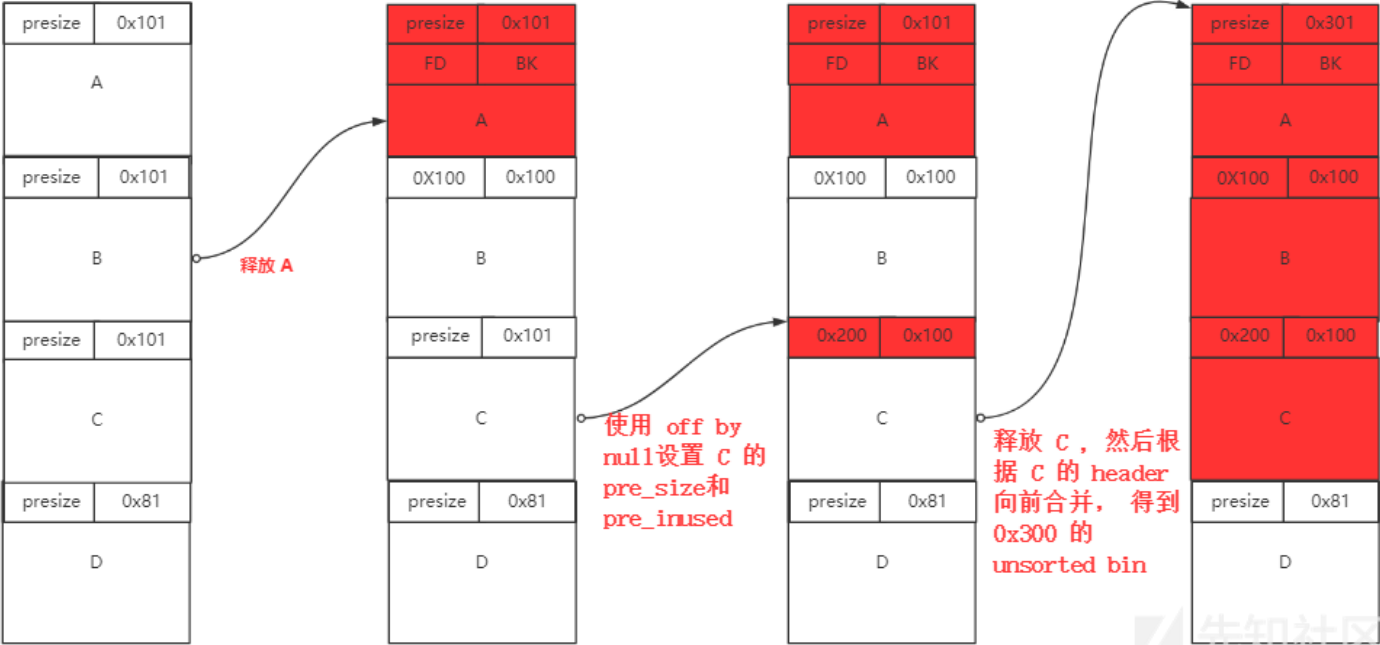
在第三步, 释放 B 之前把 B+0x200 处 设置 p64(0x200), 因为新版的 libc 会检验 `chunksz(P) != prev_size(next_chunk(P))`

off by null 缩小 B 以后, 分配 P 其大小不能再 fastbin 的范围内, 后面释放 D 需要向前合并, 会进行 unlink 操作, 所以大小 大于 fastbin, free(P)后 P 会进入 unsorted bin, 此时他的 fd, bk 都是正常的, 正常 unlink。

修改 pre_inused + 向前合并

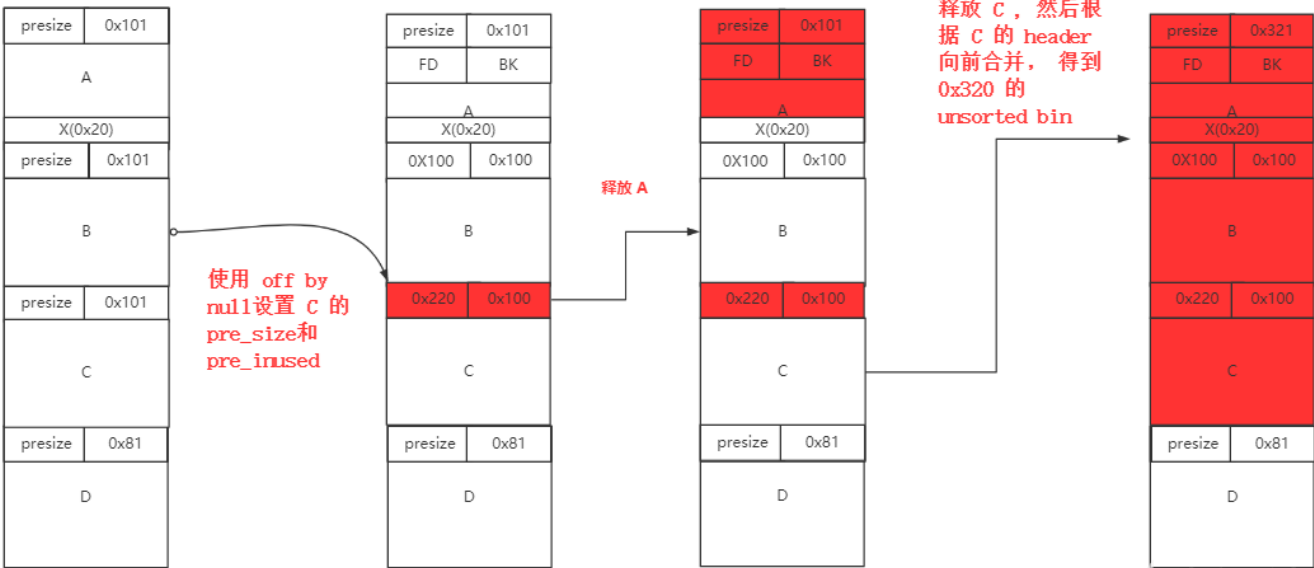
方案一

- 首先分配 4 个 chunk (A B C D), 大小分别为 0x100, 0x100, 0x100, 0x80. 最后那个用于防止 top_chunk 合并
- 然后释放 A, 此时 A 进入 unsorted bin, 生成了有效的 FD 和 BK, 为了可以在后面的融合中成功 unlink
- 然后利用 off by null, 设置 C 的 pre_size 和 pre_inused。
- 释放 C, 系统 根据 C 的 pre_size 找到 A 进行合并, 首先 unlink(A) 因为 A 已经在 unsorted bin, 不会出错, 然后就会有一个 0x300 的 unsorted bin, 此时 B 位于 该 unsorted bin 的 中间



方案二

如果程序限制只能在触发 off by null 之后才能 释放 A, 需要在 A 和 B 之间多分配一个内存块 x (0x20), 原因是 触发 off by null 后 B 被标识已经 free, 那么此时再 释放 A 就会对 B 进行 unlink, 此时 B 中 fd 和 bk 是过不了 检查的 (B 已经分配, 并已经被用来进行 off by null)。



对于堆相关的漏洞，不论是 堆溢出，double free, off by one，uaf 等其最终目的都是为了修改 chunk 的一些管理结构 比如 fd, bk, 然后在后续的堆管理程序处理中实现我们的目的（代码执行）。

堆溢出

直接可以修改 下一个 chunk 的 元数据，然后就是 unsorted bin attack, fastbin attack 等攻击手法了

double free

利用一些内存布局，可以实现 overlap chunk, 最后也是实现了 可以修改 chunk 的元数据

off by one

类似于 double free，实现 overlap chunk 然后改 chunk 元数据

点击收藏 | 7 关注 | 2

[上一篇：linux kernel pwn ...](#) [下一篇：用python继承链搞事情](#)

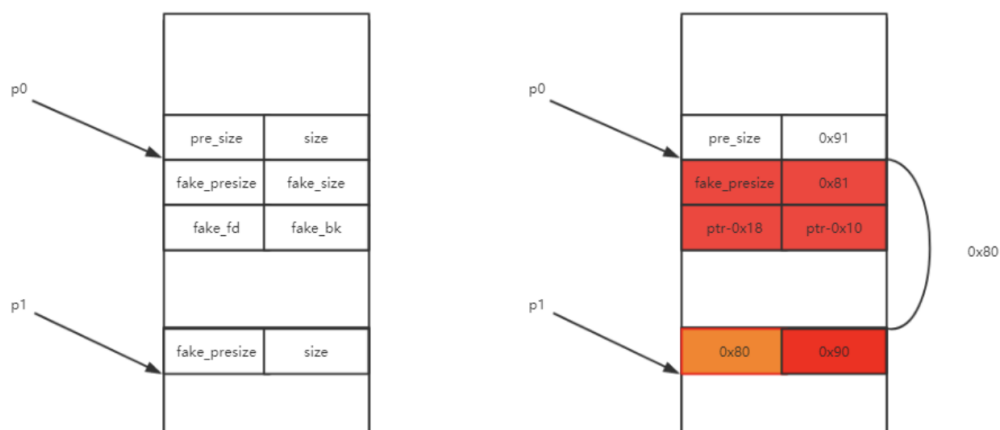
1. 1 条回复



[myhackerw****](#) 2018-10-01 00:41:53

利用

首先利用其它的漏洞伪造下面的内存布局



- `p0 = malloc(0x80)`, `p1 = malloc(0x80)`, `ptr = p0`
- 此时 `free(p1)`，发现 `p1` 所在 `chunk` 的 `pre_size = 0`，表明前一个 `chunk` 已经 `free`，于是向前合并
- 通过 `p1 - 0x10 - 0x80` (`chunk_addr - pre_size`)，找到前面已经释放的 `chunk`，也就是我们伪造的 `fake chunk p1`
- 然后进行 `unlink`，实现 `*ptr = ptr-0x18`

师傅这里不是应该表达成 `pre_size = 0x80` 才能绕过 check 吗

0 回复Ta

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

[目录](#)

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)