学号 2019300003008

密级

武汉大学本科毕业论文

面向跨版本堆管理器的堆漏洞利用机理与方法研究

院（系）名　称：国家网络安全学院

专　业　名　称：信息安全

学　生　姓　名：邓浚泉

指　导　教　师：苏璞睿　研究员（校外）

严　飞　副教授（校内）

二〇二三年五月

郑 重 声 明

本人呈交的学位论文，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。本学位论文的知识产权归属于培养单位。

本人签名：邓浚泉　　日期：2023.06.01

摘　　要

堆作为进程运行的基本内存结构，其安全问题一直是研究者关注的焦点。黑客围绕着堆管理器的实现细节发明了大量的堆利用技术，制造了极具破坏力的堆利用攻击，但堆管理器的缓解措施却作用有限。学术界对堆利用的研究缺乏系统性，大量的堆漏洞利用分散在黑客的博客、文章、邮件列表中。而且堆利用依赖于堆管理器版本，但面向跨版本堆管理器的安全研究却十分缺乏。本文旨在调研整理堆漏洞利用的机理与方法，考察现有的堆安全机制，同时为跨版本堆管理器的堆安全研究提供工具和数据支持。

本文的主要工作与贡献如下：

1. 针对堆利用技术缺乏整理的问题，本文调研了知名的 25 种堆利用技巧和 42 种最新版本堆管理器中存在的安全检查，阐释了它们的机理，归纳了它们的方法。
2. 针对堆利用研究缺乏系统的问题，本文提出了堆利用的利用流模型，系统阐释了堆利用攻击的方法和流程，为堆漏洞利用提供了通用的抽象。
3. 针对跨版本堆管理器的堆安全研究缺乏工具的问题，本文开发了一套面向跨版本堆管理器的堆利用测试套件。套件的三个组件 libccompile、libcbench 和 cheapsec 分别解决了 glibc 的深度耦合问题、切换 libc 库困难的问题和难以获知堆管理器安全属性的问题。
4. 针对跨版本堆管理器的堆安全研究缺乏数据的问题，本文使用上述测试套件在 23 个版本的堆管理器中对 25 个利用技巧和 42 个安全检查做出了测试，获得了跨版本堆管理器的安全属性数据。

关键词：堆；漏洞利用；堆管理器；堆利用

ABSTRACT

As a fundamental memory structure for processes to run, the heap has always been a focus of researcher attention. Hackers have invented a large number of heap exploitation techniques around the implementation details of the heap allocator, resulting in highly destructive heap exploitation attacks. However, the heap allocator's mitigating measures are limited. The academic community's research on heap exploitation is lack of systematicality, with a large number of heap vulnerabilities scattered in hackers' blogs, articles, and email lists. Moreover, heap exploitation depends on heap allocator version, but there is very limited security research targeting across-version heap allocators. The aim of this paper is to research, organize, and examine the mechanisms and methods of heap vulnerability exploitation, examine the existing heap security mechanisms, and provide tool and data support for heap security research targeting across-version heap allocators.

The main work and contributions of this paper are as follows:

1. To address the lack of organization and analysis of heap exploitation techniques, this paper investigates and organizes 25 famous heap exploitation techniques and 42 heap security checks present in the latest versions of heap allocators, explaining their mechanisms and summarizing their methods.
2. To address the lack of systematic research on heap exploitation, this paper proposes a heap exploitation flow model, which systematically explains the methods and processes of heap exploitation attacks, providing a general abstract for heap vulnerability exploitation.
3. To address the lack of tools for studying heap security across versions, this paper develops a heap exploitation test suite for across-version heap allocators. The suite's three components libccompile, libcbench, and cheapsec solve the deep coupling problem of glibc, the difficulty of switching libc libraries, and the problem of difficulty in obtaining heap allocator security attributes.
4. To address the lack of data for studying heap security across versions, this paper uses the above test suite to test 25 heap exploitation techniques and 42 heap security

checks in 23 versions of heap allocators, obtaining security attribute data for across-version heap allocators.

Keywords: heap; exploitation; heap allocator; heap exploitation

目　　录

1 引言

1.1 研究背景与研究目的

随着软件规模的增长和复杂度的提高，软件漏洞问题已经成为计算机安全领域中的一个重要挑战。其中，堆漏洞问题尤为突出，据微软表示，堆漏洞在其产品的安全问题中占 53%[1]。利用这些漏洞的方法被称为堆利用技术。与基于栈的利用技术相比，堆利用技术通常非常强大，因为攻击者可以通过使用它来绕过现代操作系统的内存安全保护措施，为进一步攻击提供条件。例如，黑客可以使用一次简单的空字节溢出覆盖 ptmalloc2 的元数据，导致 Chrome OS 中的权限提升[2]。虽然堆管理器开发人员已经采取了一些加固机制来检测堆元数据破坏，并且在发现破坏时中止程序，以应对堆漏洞问题。但是这些措施都必须考虑其对性能的影响，需要进行艰难的权衡。这导致这些措施的影响力有限，几乎没有减少黑客可用的大量堆利用技术。

然而，对堆利用的系统性研究却缺乏工作，大多数相关资料都散布在黑客的技术博客和邮件列表中。而且，堆利用技术与堆管理器版本高度相关，但面向跨版本堆管理器安全研究却缺乏有力的工具和可供参考的数据。因此亟需加强对相关技术和工具的研发。本文就旨在面向不同版本的堆管理器，探索堆漏洞利用的机理和方法，研究现有的堆安全机制，以期为进一步提高抵御堆漏洞攻击的能力提供帮助。

1.2 国内外研究现状

本文的研究对象是 GNU/Linux 中使用最广泛的 glibc。对于 glibc 中存在的大量堆利用技巧和缓解措施，已经有一些相关的总结与梳理工作。2018 年，裴中煜等[3] 按历史梳理了早期的 5 种利用方法和现代的 6 种攻击手段，分析了它们的特征和局限性，并讨论了操作系统通用的内存安全措施。但选取的堆利用样本较少，而且缺少对堆管理器实现中的安全机制的关注。2019 年，Benjamin[4] 总结了 3 种常见的堆利用技巧，并讨论了现代堆管理器在安全设计上做出的努力。2022 年，0x432b 在他的博客中[5] 梳理了 glibc 2.34 中已被修复的 14 种利用技巧和仍然可用的 16 种利用技巧，并且给出了它们适用的堆管理器版本。但是这些利用技巧

并不全是堆技巧，而且只是被罗列了出来。在跨版本堆管理器的堆利用测试方面，Rørvik 在它的论文中[6] 中测试了 8 种堆利用技巧在 5 个 glibc 版本上的可行性。本文则选取了 25 个堆利用技巧，按照攻击手法分为了 5 类，并提出了它们的利用流模型。然后梳理了 42 个堆管理器中的安全检查，使用测试套件测试了这些利用技巧和安全检查在 23 个版本的堆管理器中的可行性。

有许多工作也将建立堆利用模型作为其中的一部分。堆利用原语是黑客经常使用的概念，Repel 等在文章中[7] 提到了使用原语为堆利用建模的方法。2020 年，Yun 等在他们的工作中[8] 提出了堆利用的四种漏洞原语和四种攻击原语。本文将其发展为了五种初级原语、五种中级原语和两种高级原语，并阐述了原语之间的关系。

另一个不可忽视的主题是堆利用的自动化挖掘和堆漏洞的自动化利用。在可利用性挖掘方面，2018 年，Eckert 等人[9] 将有界模型检查引入堆管理器的可利用性分析，开发了 HeapHopper 工具，在三个堆管理器中验证了其挖掘能力。为了解决 HeapHopper 符号执行的状态爆炸问题，2020 年，Yun 等人[8] 将模糊测试的思想引入堆利用挖掘，开发了 ARCHEAP，发现了 glibc 中的五种新堆利用技巧。在自动化利用方面，2018 年，Wu 等人[10] 使用内核模糊测试和符号执行技术开发了自动利用框架 FUZE，在 15 个内核堆漏洞中成功利用了 12 个。2018 年，Wang 等人[11] 提出了一种内存布局导向的模糊测试方法，以生成堆漏洞的利用程序。在 19 个 CTF 挑战的测试中，直接生成代码的比例达到 47‰。2021 年，Wang 等人[12] 还提出了一个名为 MAZE 的方案，该方案使用堆操作语义和丢番图方程求解来自动生成堆布局。这些工作大多基于特定的堆利用模型，使用了符号执行和模糊测试技术。它们是本研究进一步工作的有益参考。

1.3 本文主要工作

以上这些相关的工作有四个突出问题：

- 许多堆漏洞利用技术分散在黑客的文章、博客、邮件列表中，尤其是较新的且尚未修复的利用技术，很少得到学术界的关注和考察。
- 现有的堆利用总结多为对堆利用技巧的罗列，缺少成体系的归纳和整理，也很少建立统一的利用模型。
- 堆中的安全机制是实施堆利用不可绕开的主题，但很少有对堆安全机制的考

察和整理。

- 堆管理器的版本也是实施堆利用的关键因素，面向跨版本堆管理器的堆漏洞利用却鲜有研究。

本研究由此入手，做了四个方面的工作：

- 调研黑客的文章和邮件，考察并复现了知名的 25 种堆漏洞利用技巧；阅读 glibc 的源码，整理出了最新版本堆管理器中的 42 种堆安全检查。阐释了它们的机理，归纳了它们的方法，展示了最新的堆利用与防护技术。
- 根据以上的调研成果，提出了堆利用的利用流模型。基于利用原语的概念，系统性地阐释了堆利用攻击的方法和流程，为堆漏洞利用提供了通用的抽象。
- 为跨版本堆管理器的堆漏洞利用开发了一套面向跨版本堆管理器的堆利用测试套件。这套套件由 libccompile、libcbench 和 cheapsec 三个组件组成，为堆安全研究提供了有力的工具支持。
- 使用该套件在 23 个版本的堆管理器上实际测试了 25 个堆利用技巧和 42 个堆安全检查，得到的测试数据描摹出了现今使用的各版本堆管理器的安全属性，为堆安全研究提供了有益的数据参考。

1.4 论文结构

本文将按照背景知识、理论研究、工具研发的顺序介绍以上这些工作。第二章“背景知识”将介绍本文的研究对象——GNU/Linux 中的 ptmalloc2 堆管理器，包括 ptmalloc2 的实现以及与实现相关的安全问题。第三章“堆利用机理与方法”将阐述 25 种堆利用技巧，并据此提出堆利用的利用流模型。第四章“堆安全机制”将介绍操作系统和堆管理器为阻止堆利用做出的努力。第五章“堆利用测试套件”将介绍本研究开发的堆利用测试套件，包括 libccompile、libcbench 和 cheapsec 三个组件。最后，第六章“总结与讨论”将总结本研究的工作并探讨堆利用与防护的未来发展趋势。此外，本文还有两个附录，它们分别是堆管理器中存在的 42 个安全检查（附录 A），以及跨版本堆管理器的测试数据（附录 B），供研究者参考使用。

2 背景知识

本文的研究对象——堆管理器是一种复杂而精致的软件，它在程序的运行过程中频繁地发挥作用。因此，它的性能和安全至关重要。在设计堆管理器时，开发者需要充分考虑性能和安全之间的平衡。为了实现高性能，堆管理器通常会采用复杂的数据结构和分配算法。同时，为了提高安全性，堆管理器又会引入许多额外的安全检查。在研究堆管理器的利用技术之前，研究者通常需要对这些数据结构和分配算法有基本的了解。

2.1 GNU/Linux 中的堆管理器

本文选择 GNU/Linux 操作系统作为研究平台。这是基于以下两点考虑：首先，GNU/Linux 是世界上最受欢迎的操作系统之一，它是许多重要服务的基础设施。因此，对 GNU/Linux 的堆安全性进行研究具有重要的意义。其次，GNU/Linux 以及在其之上运行的堆管理器都是开源的，这为研究的进行提供了极大的便利。

在 GNU/Linux 操作系统中,Linux 内核和 GNU 操作系统工具集是分离的[13]。堆管理器由 GNU C Library（glibc）在用户空间实现，并在程序运行时动态加载，与 Linux 内核无关。这也意味着 GNU/Linux 中的堆管理器不是唯一的。除了 glibc 的堆实现（ptmalloc2）外，还有 Google 的 tcmalloc[14]，以及被 Firefox 使用的 jemalloc[15] 等堆实现。本文将主要研究 glibc 实现的堆管理器 ptmalloc2，它是 GNU/Linux 上使用最广泛的堆管理器。

GNU/Linux 中的堆管理器主要负责管理进程的动态内存。一个进程的内存空间结构如图 2.1 所示[16]。

Linux 内核提供了两个系统调用：brk 和 mmap，用于动态内存分配。brk 调用可以将堆段的上边界往上延伸，而 mmap 调用可以在栈和堆的中间创建内存映射页面，它们都能够扩充程序的动态地址空间。在 GNU/Linux 系统中，堆管理器通常混合使用 brk 和 mmap 来创建和扩大堆。

在为堆开辟了空间后，堆管理器会将堆按一定结构组织起来，这些结构以*元数据*的方式记录在堆上。堆漏洞利用通常通过污染这些元数据来达到攻击效果。

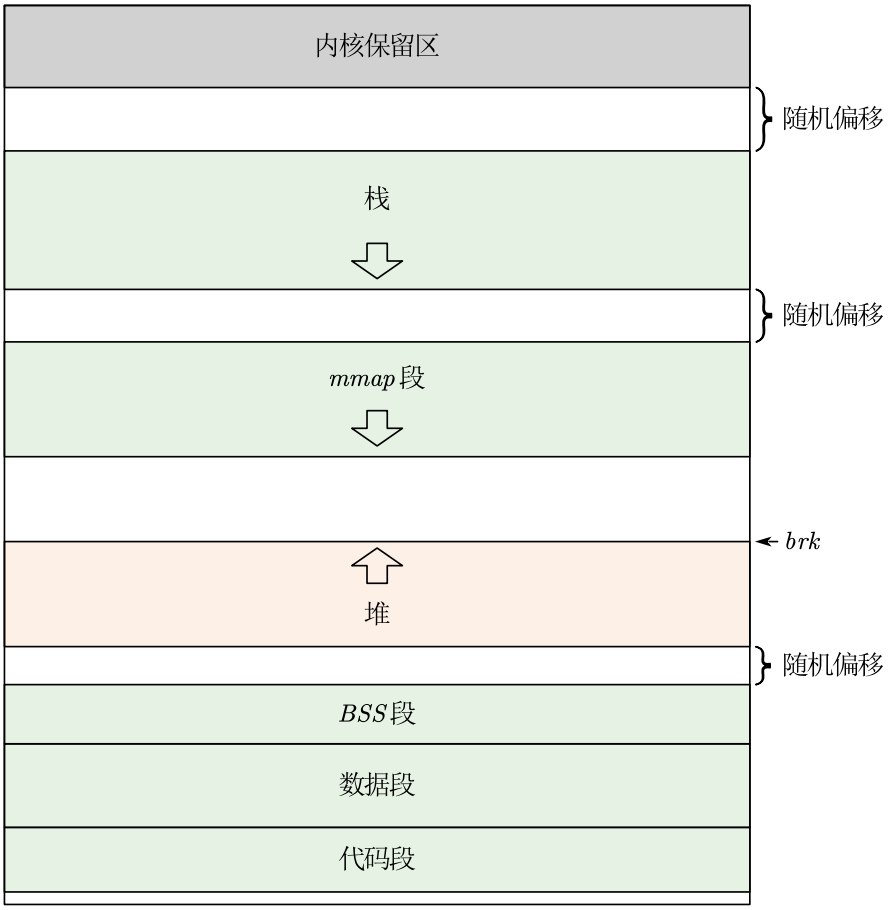

图 2.1　进程的内存结构

2.2　GNU C Library 的堆实现

glibc 当前版本中的堆实现是由 Wolfram Gloger 开发的 ptmalloc2（pthreads malloc 2）修改而来的。ptmalloc2 又源自 Doug Lea 于 1987 年开发的经典的 dlmalloc（Doug Lea malloc）[17]。

ptmalloc2 遵循 POSIX 标准，为用户提供了以下标准 POSIX 动态内存管理函数：

- malloc：从堆中分配一段新的动态内存区域。
- free：将一段动态内存区域释放给堆管理器。
- realloc：重新分配一段已分配的动态内存区域，使其扩大或者缩小。
- calloc：分配一段新的动态内存区域给数组，并将其初始化成 0.

为了提高多线程环境下堆的性能，ptmalloc2 允许进程包含多个堆区，且每个堆都在其地址空间中增长。ptmalloc2 采用了以下堆结构来达到这个目的:

- Arena：Arena 是一个堆 (Heap) 的集合，未进行操作时由多个线程共享，进行堆操作时由某个线程独占。Arena 包含多个 Heap 的引用，以及这些 Heap 中共有的 Bin。
- Heap：Heap 是一段连续的内存区域，隶属于某个 Arena，被用来分割为 Chunk 返回给用户。
- Bin：Bin 是用于组织空闲 Chunk 的结构，隶属于某个 Arena。空闲 Chunk 根据大小和释放时间被存储在各种不同的 Bin 中，以便堆管理器可以快速地找到适合分配请求的 Chunk。
- Chunk：Chunk 是内存分配的基本单位。用户每次的 malloc 或 free 都是对一个 Chunk 进行的。Chunk 可以被分配 (由用户拥有)、释放 (由堆管理器拥有)、分割成小的 Chunk 或与相邻的 Chunk 合并成更大的 Chunk。每个 Chunk 都存在于一个 Heap 中，并属于一个 Arena。

接下来将按由小到大的顺序对这些堆结构逐一介绍。

2.2.1 堆结构

(1) Chunk

Chunk 对应的是 malloc_chunk 结构。在 glibc 2.32 中，它的定义如下：

```
struct malloc_chunk {
  size_t      prev_size;
  size_t      size;
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};
```

malloc_chunk 结构中包含的字段有：

- prev_size：与当前 Chunk 物理相邻的上一个 Chunk 的大小。
- size：当前 Chunk 的大小。

  在 X86_64 架构中，由于每个 Chunk 的大小和起始地址都是 16 字节对齐的，size 的低 4 位应当总为 0. 它们事实上不被用来表示 Chunk 大小，而是用来表

示一些标志。按从低到高，这四个比特的含义是：

1. P：物理相邻的前一个 Chunk 是否空闲。
2. M：当前 Chunk 是否由 mmap 分配。
3. N：当前 Chunk 是否隶属于主线程的 Arena。
4. 未定义。

- fd：Bin 中的下一个 Chunk。
- bk：Bin 中的上一个 Chunk。
- fd_nextsize：Large Bin 中下一个不同大小的 Chunk。
- bk_nextsize：Large Bin 中上一个不同大小的 Chunk。

在 Chunk 数据结构中，一个正在使用的 Chunk 只使用 prev_size 和 size 字段，fd 字段及以下用于存储用户数据。空闲的 Chunk 被放入 Bin 中，因此会使用 fd 字段和 bk 字段。对于被放入 Large Bin 的空闲 Chunk，它会使用 fd_nextsize 和 bk_nextsize 字段。这些字段在 Bin 中的用法将在下文讨论。

需要注意的是，Chunk 的大小是 16 字节对齐的，这意味着它一定是 0x10 的倍数。如果 malloc 申请的大小不是 0x10 的倍数，堆管理器将补齐剩余的部分。另外，由于 malloc_chunk 结构体至少占据了 Chunk 中 0x10 大小的空间 (包括 prev_size 和 size 字段)，因此 Chunk 的最小大小是 0x20，并且满足 $size = 0x20 + 0x10 \times N$，其中 N 是大于等于 0 的正整数。

在 glibc 中，还存在一个特殊的 Chunk，称为 Top Chunk，它位于 Heap 的顶部。在堆被初始化时，Heap 只有一个 Top Chunk。当 malloc 请求获得新的 Chunk 时，Top Chunk 会被拆分以创建新 Chunk 所需的大小。malloc 拆分 Top Chunk 的情况将在下文讨论。

(2) Bin[18]

Bin 是一种空闲列表（freelist），用来收集和组织空闲的 Chunk，一个 Chunk 被释放就会加入 Bin。在 glibc 中，Bin 使用链表实现。一共有五种类型的 Bin，可以分为两组：常规 Bin 和缓存 Bin。

常规 Bin 使用双向链表实现，Chunk 通过 fd 和 bk 指针互相连接，而且遵循先进先出（FIFO）规则。它们是：

- Unsorted Bin：Unsorted Bin 是一个中间站。在 Chunk 加入 Small Bin 和 Large

Bin 之前，首先会被放到 Unsorted Bin 中。Unsorted Bin 中的 Chunk 是不按大小排列的。

- Small Bin：Small Bin 用来容纳较小的空闲 Chunk。每个 Small Bin 中的 Chunk 大小都相同，这样有助于快速查找到合适的小 Chunk。在 X86_64 架构中，一共有 62 个 Small Bin，其中容纳的 Chunk 大小范围从 0x20 到 0x3f0。
- Large Bin：Large Bin 用来容纳其它的空闲 Chunk。每个 Large Bin 能容纳一定范围大小的 Chunk，其中的 Chunk 按从大到小的顺序排列。此外，为了提高查找速度，Large Bin 中的 Chunk 还会使用 fd_nextsize 和 bk_nextsize 指针，它们指向 Large Bin 中下一个或上一个大小不同的 Chunk。在 X86_64 架构中，一共有 63 个 Large Bin，其中容纳的 Chunk 大小范围从 0x400 到无限大。

以 Unsorted Bin 为例，常规 Bin 的结构如图 2.2 所示。

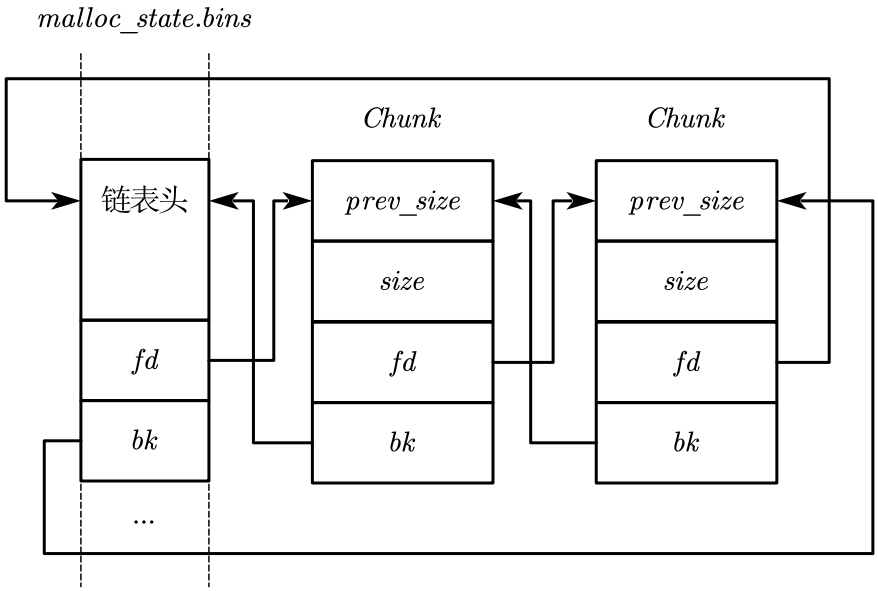

图 2.2　Unsorted Bin 的结构

缓存 Bin 使用单向链表实现，Chunk 只使用 fd 指针互相链接，遵循后进先出（LIFO）规则。它们被用来组织一些最近释放的非常小的空闲 Chunk，使得堆管理器能在遍历常规 Bin 之前快速找到它们。小于一定大小的 Chunk 在被释放时直接加入缓存 Bin，并且通过 size 字段的 P 比特标记为仍在使用的状态，这样它就不会与其它 Chunk 合并。在分配时，缓存 Bin 也会最先被堆管理器考虑。两个缓存 Bin 是：

- Fast Bin：每个 Fast Bin 中的 Chunk 大小都相同。在 X86_64 架构中，一共有 10 个 Fast Bin，能容纳的 Chunk 大小范围从 0x20 到 0x80.
- Tcache：Tcache 是 glibc 2.26 新加入的缓存 Bin。和 Fast Bin 不同之处在于，Tcache 是每个线程独有的，而 Fast Bin 是多线程共享的。在 X86_64 架构中，一共有 64 个 Tcache，范围从 0x20 到 0x410。每个 Tcache 可以容纳 7 个 Chunk，这 7 个 Chunk 大小相同。如果有更多的 Chunk，就会落入 Fast Bin 和 Unsorted Bin 中。

以 Fast Bin 为例，缓存 Bin 的结构如图 2.3 所示。

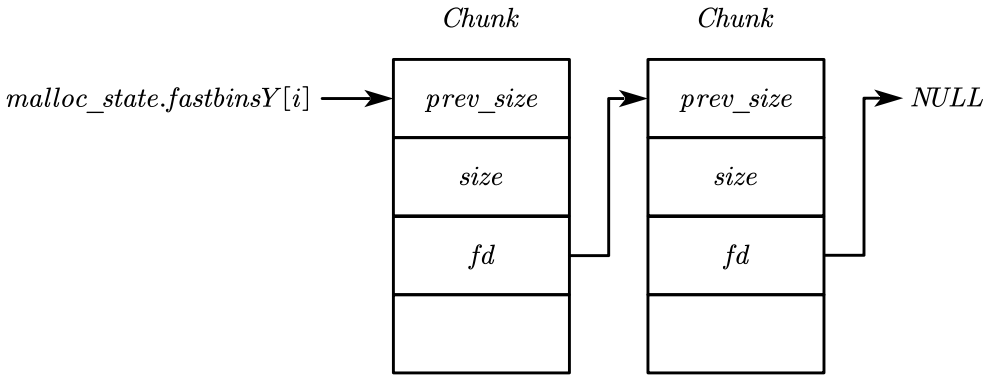

图 2.3　Fast Bin 的结构

(3) Heap 和 Arena

在 glibc 堆中，一个进程有多个 Arena，每个 Arena 在线程使用时都是上锁的。其中主线程使用*主 Arena*，其它的线程为了避免资源紧张，可以创建和使用*非主 Arena*。每个 Arena 的状态由 malloc_state 结构体定义，其中主 Arena 的 malloc_state 是一个全局变量，存储在 libc.so 的数据段上。其它线程的 malloc_state 存储在对应 Arena 的起始位置。malloc_state 在 glibc 2.32 中的定义如下：

```
struct malloc_state
{
  int flags;
  mfastbinptr fastbinsY[NFASTBINS];
  mchunkptr top;
  mchunkptr last_remainder;
  mchunkptr bins[NBINS * 2 - 2];
  unsigned int binmap[BINMAPSIZE];
  struct malloc_state *next;
  struct malloc_state *next_free;
  INTERNAL_SIZE_T attached_threads;
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
};
```

这个结构体存储了一些 Arena 的标志和属性，以及上文所述的各种 Bin 的链表头。

一个非主 Arena 可以包含多个 Heap，一个 Heap 是使用 mmap 分配的一段内存空间。在 X86_64 架构中，Heap 的起始地址一定是 0x4000000 对齐的。也就是说，每个 heap 的起始地址的低 26 位都是 0。如图 2.4 所示。

这一机制主要是用来确定一个 Chunk 是属于哪个 Arena 的：每个 Heap 的地址最低处都有一个 heap_info 结构，其中包含了指向 Heap 所属的 Arena 的指针。如果堆管理器需要知道一个 Chunk 属于哪个 Arena，首先通过 size 字段的 N 比特判断是否属于非主 Arena，然后将 Chunk 的低 26 位置 0，就能得到指向所属 Arena 的指针。

2.2.2 堆操作

(1) malloc

在 glibc 源码中，内存分配在 malloc/malloc.c 的 __libc_malloc 和 _int_malloc 两个函数中实现。它的整体流程图 2.5 所示。

程序在调用 malloc 时，首先会进入 __libc_malloc 函数。在 __libc_malloc 函数中，glibc 检查线程的 Tcache 中是否有满足要求的空闲 Chunk。如果没有，才会尝

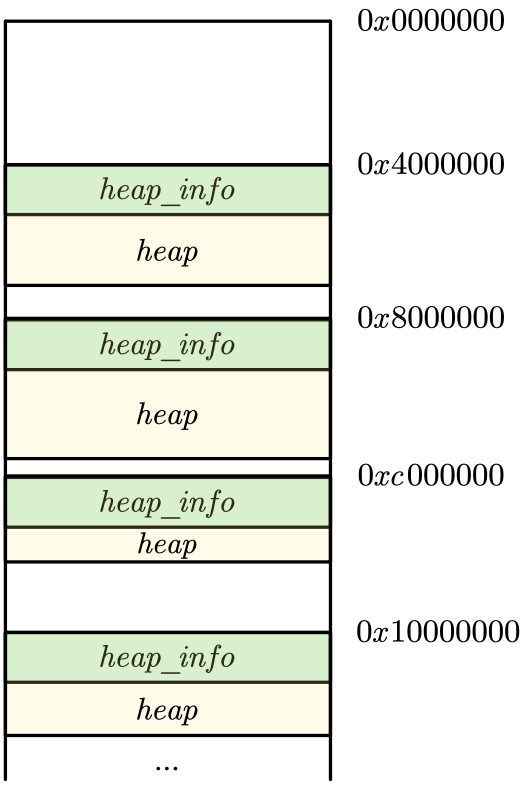

图 2.4　Heap 的结构

试获取一个未加锁的 Arena，执行常规分配流程。也就是调用 _int_malloc。

_int_malloc 的情况比较复杂。总的来说分为五步：

1. 从 Fast Bin 和 Small Bin 中寻找恰好满足要求的空闲 Chunk，如果有就直接返回。
2. 遍历 Unsorted Bin，将其中的 Chunk 移入 Small Bin 和 Large Bin。同时看 Unsorted Bin 中是否有恰好满足要求的空闲 Chunk，如果有就直接返回。
3. 从 Large Bin 中寻找合适的空闲 Chunk，如果有就把它分割并返回。
4. 运行到此处，说明 Bin 中没有恰好满足要求的空闲 Chunk。于是遍历 Small Bin 和 Large Bin 寻找比所需大小大的最小 Chunk。如果有，把它分割返回给用户。
5. 运行到此处，意味着所有的空闲 Chunk 都不能满足要求，只能从 Top Chunk 上分割。如果 Top Chunk 都不能满足，调用 sysmalloc 扩大堆。

这其中有几个重要的额外步骤：

如果申请的内存太大，超出了阈值 mmap_threshold，glibc 将使用 mmap 函数创

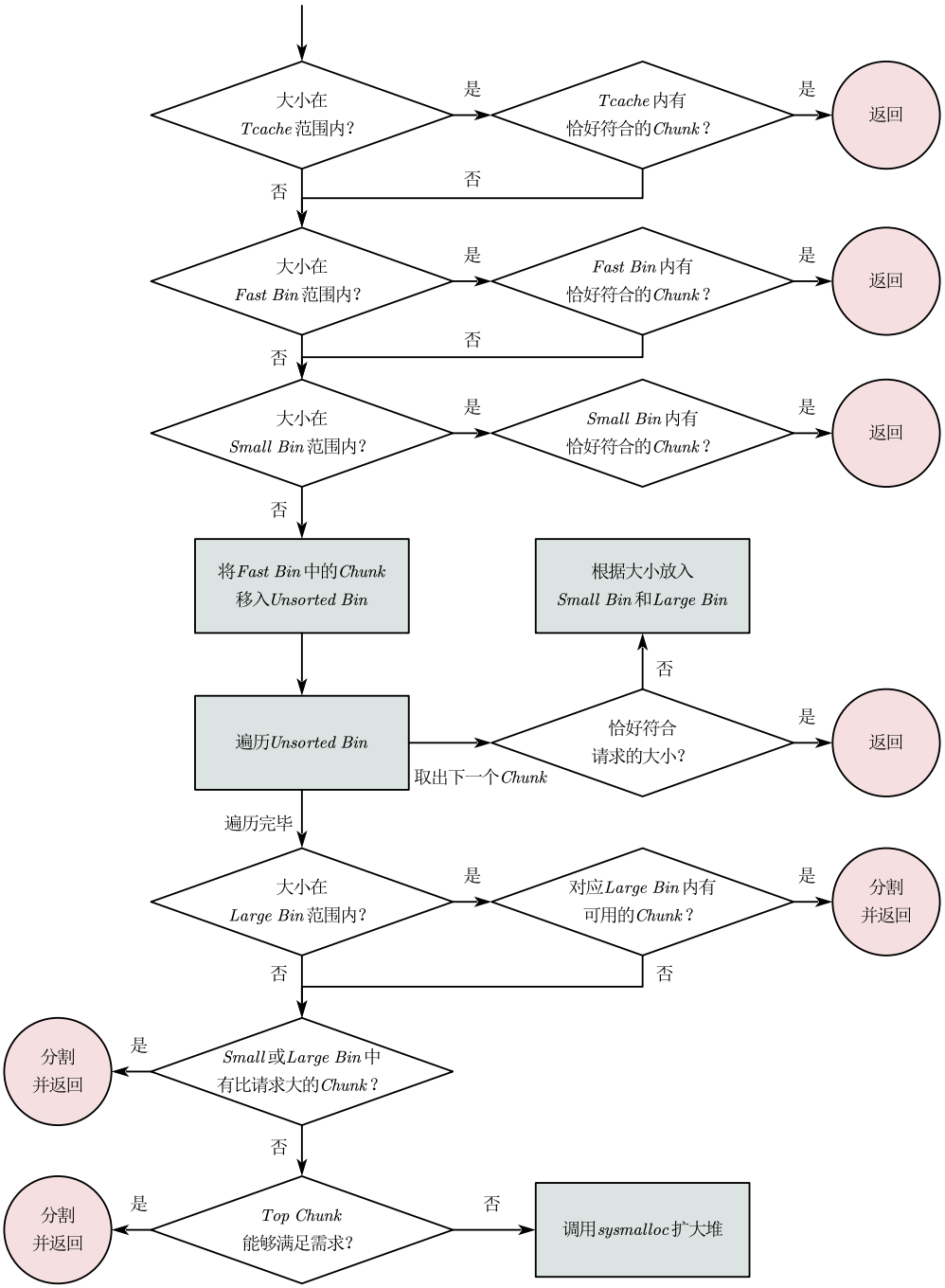

图 2.5　malloc 的执行流程

建一个独立的内存区域并将其返回给用户。这个由 mmap 创建的 Chunk 独立于上文描述的堆结构之外，不属于任何 Heap 或 Arena。这样的 Chunk 的 size 字段的 M 比特将被置为 1。默认情况下，mmap_threshold 在 X86_64 架构中设置为 128KB。该值会动态变化，没有上限。每次释放由 mmap 分配的大 Chunk 时，mmap_threshold 都会与其进行比较，并更新为两个之中的较大值。

在遍历 Fast Bin 和 Small Bin 之后、遍历 Unsorted Bin 之前，glibc 会执行一次 malloc_consolidate 函数。该函数会遍历 Fast Bin，并将其中的所有 Chunk（如果可合并）进行前后合并，最终将它们放入 Unsorted Bin 中，以改善堆的碎片化情况。

在遍历 Fast Bin、Small Bin 或者 Unsorted Bin 时如果遇到了恰好满足要求的空闲 Chunk，glibc 不会立即返回，而是将其中剩余的相同大小的 Chunk 移入 Tcache 后返回。

(2) free

与分配类似，内存释放在 malloc/malloc.c 的 __libc_free 和 _int_free 两个函数中实现。它的整体流程如图 2.6。

在 __libc_free 中，如果目标 Chunk 是由 mmap 分配的 Chunk，glibc 会调用 munmap 将其直接释放。然后才调用常规释放流程，执行 _int_free。

_int_free 相对比较简单，执行了以下三个步骤：

1. 如果 Chunk 落入了 Tcache 的范围，并且 Tcache 没有满，直接放入 Tcache。
2. 如果落入了 Fast Bin 的范围，放入 Fast Bin。
3. 尝试与后向相邻的 Chunk 合并，然后尝试与前向相邻的 Chunk 合并。合并完成后，如果 Chunk 没有被合入 Top Chunk，就把它放入 Unsorted Bin。

2.3 堆脆弱性

从上文的叙述可以看出，堆上的操作都十分依赖于堆元数据。分配算法和释放算法的每一步都涉及到对 Chunk 上大小、链表指针等的判断和操作。然而，为了提高堆管理的性能和效率，这些元数据都分散在各个 Chunk 之中，这使得用户代码中的不经意错误就可能会导致元数据的破坏。此外，如果这种破坏是来自攻击者的蓄意行为，堆管理器的操作就会被攻击者引导，所造成的后果将具有极大的破坏性。

能够造成堆元数据破坏的程序错误主要有以下几种：

- 双重释放（Double Free）：程序在调用 free 释放了一段动态内存后又对同一段内存调用了 free。在 glibc 的堆实现中，这可能会导致同一个 Chunk 在 Bin 上出现两次。
- 堆溢出（Heap Overflow）：程序在操作动态内存时没有做合适的边界检查，导

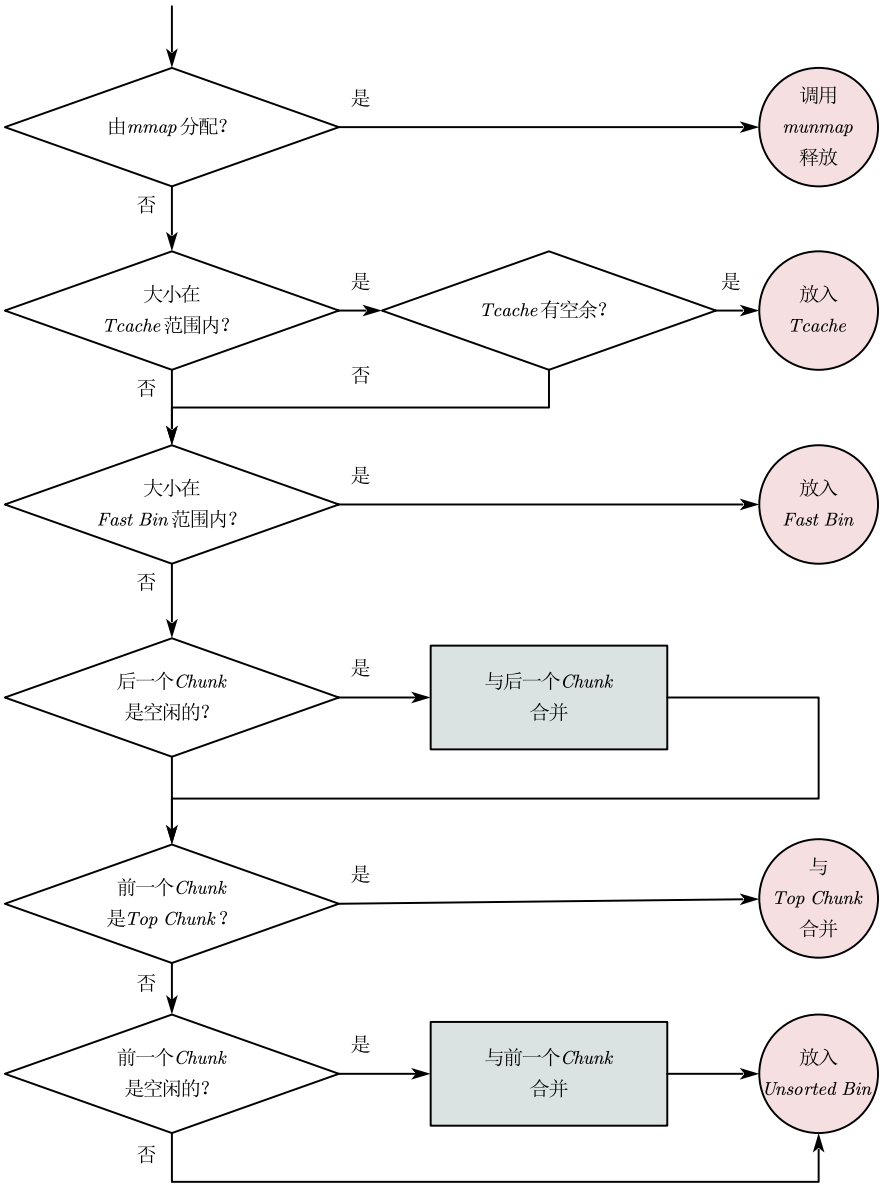

图 2.6　free 的执行流程

致数据溢出到了边界以外的位置。在 glibc 中，这可能会导致对相邻 Chunk 的写入，从而破坏相邻 Chunk 的大小、链表指针等。

- 释放后使用（Use After Free）：程序在调用 free 释放了一段动态内存后又使用了这段内存。在 glibc 中，这可能会导致对当前 Chunk 链表指针的破坏。
- 任意释放（Arbitrary Free）：程序对一段非 malloc 分配的内存调用了 free。在 glibc 中，这可能会导致不属于堆的内存进入 Bin 中。

攻击者可以利用这些程序错误实施以下攻击手法：

- 双重释放：将一个 Chunk 释放两次。

- 控制释放：对指定的内存调用 free。
- 污染大小：污染 Chunk 的 size 字段或 prev_size 字段，使得堆管理器错误判断 Chunk 的大小。
- 污染链表：污染 Chunk 的 fd、bk、fd_nextsize 或 bk_nextsize 指针，使得堆管理器错误判断 Bin 中的元素。
- 污染标志位：污染 Chunk size 字段的低三位（P、M、N 比特），使得堆管理器错误判断 Chunk 类型。

所谓堆漏洞利用，就是指攻击者将以上攻击手法与堆管理器机制结合，通过精心构造的元数据污染，在恰到好处的时机利用堆管理器的错误判断和操作达成攻击目的。

2.4 本章小结

本章首先讨论了 GNU/Linux 操作系统中堆管理器的概念，包括 Linux 内核所提供的动态内存管理接口和在此基础上构建的多种堆管理器。

接下来，本研究将目光聚焦于 glibc 实现的堆管理器——ptmalloc2，并逐一介绍了 Chunk、Bin、Heap、Arena 等堆结构以及 malloc、free 等堆操作。

最后，本文讨论了堆的脆弱性。用户程序可能存在双重释放、堆溢出、释放后使用以及任意释放等程序漏洞，而黑客可以利用这些漏洞实施多种攻击手法。堆漏洞利用是将攻击手法与堆管理器机制有机结合的过程。

3 堆利用机理与方法

3.1 堆利用技巧

前一章讨论了堆的脆弱性，并指出了攻击者可以利用的五种攻击手法：双重释放、控制释放、污染大小、污染链表以及污染标志位。接下来，本文将结合 25 个著名的堆利用技巧，深入探讨堆利用的实施方法。

这 25 个堆利用技巧主要来源于 how2heap 开源项目[19]、黑客博客、文章和邮件，时间跨度从 2005 年到 2022 年。其中一些已经无法确定提出者和提出时间。它们涵盖了公开发表的大部分堆利用技巧。在选取这些堆利用技巧时，采取了以下原则：

1. 它是一个与堆相关的利用技巧，而不是一个涉及到栈、GOT 劫持、Vtable 劫持等复合技巧的综合攻击。
2. 它不过于陈旧，在近十年的堆管理器上应该仍然可行。

3.1.1 双重释放

双重释放攻击手法依赖于用户程序一次错误的 free 操作。如果对一段已经释放的内存再次执行 free 操作，就可能导致一个 Chunk 在 Bin 上引用两次。

如果攻击者再次通过 malloc 获得这个 Chunk，由于该 Chunk 仍然被视为释放状态（因为在 Bin 中还有一份引用），对该 Chunk 的修改将导致释放后使用漏洞。此外，如果两次 malloc 获得了同一个 Chunk，则会造成 Chunk 重叠，从而进一步扩大攻击的影响范围。这些后果都可以被用来制造更大的泄漏或写入等攻击。

(1) fast bin dup

fast bin dup 是一种简单的绕过 Fast Bin 中双重释放检测的技巧。

在将 Chunk 加入 Fast Bin 时，glibc 会检查对应 Bin 顶部的 Chunk 是否等于将要被释放的 Chunk。而 fast bin dup 通过在两次释放 Fast Bin Chunk 之间释放一个同样大小的 Fast Bin Chunk 来绕过这个检查。

这种利用技巧会导致同一个 Fast Bin Chunk 在 Fast Bin 上被引用两次。

(2) fast bin dup consolidate

在两次释放 Fast Bin Chunk 之间，攻击者会申请分配一个较大的 Chunk（使得 Fast Bin 和 Small Bin 都无法满足要求），从而触发 malloc_consolidate 函数。这个函数会将上一次释放的 Chunk 从 Fast Bin 中移出，然后加入到 Unsorted Bin 中。这样就能够绕过前文提到的双重释放检测。

这种利用技巧同样会导致同一个 Fast Bin Chunk 在 Fast Bin 上被引用两次。

(3) tcache dup

tcache dup 是一种直接的 Tcache Chunk 双重释放技巧。

在 glibc ≥ 2.26 版本引入了 Tcache，但在 glibc ≤ 2.29 的版本中没有任何关于 Tcache 的双重释放检查。因此直接对一个 Tcache Chunk 双重释放不会发生任何问题。

这种利用技巧会导致同一个 Tcache Chunk 在 Tcache 上被引用两次。

(4) house of kauri

house of kauri[20] 是 awarau 在 2020 年提出的一种 Tcache Chunk 双重释放技巧。

本文将在后文详细讨论 glibc > 2.29 版本中的 Tcache 双重释放检查。house of kauri 通过简单地修改 Tcache Chunk 的 size 字段（通常需要利用堆溢出等漏洞来完成）来实现 Tcache Chunk 的双重释放。

这种利用技巧会导致同一个 Tcache Chunk 在 Tcache 上被引用两次。

(5) house of botcake

house of botcake 是另一种 Tcache Chunk 双重释放技巧。

这个技巧在 Tcache 已满时释放一次目标 Chunk (Chunk 会进入 Unsorted Bin)，在 Tcache 未满时释放一次目标 Chunk (Chunk 会进入 Tcache)。这样，目标 Chunk 就被双重释放了。

这种利用技巧会导致同一个 Chunk 在 Tcache 上和 Unsorted Bin 上同时被引用。

3.1.2 控制释放

控制释放是一种攻击手法，其假设攻击者可以控制 free 函数的参数，从而使堆管理器释放攻击者构造的伪 Chunk（Fake Chunk）。这可能会导致 Fake Chunk 被链接到 Bin 上，为之后的进一步攻击制造条件。

(1) house of spirit

house of spirit 是由 Phantasmal Phantasmagoria 在他著名的《Malloc Maleficarum》[21] 中发明的一种将 Fake Chunk 加入 Fast Bin 的利用技巧。

这个技巧试图构造一个合适的 Fake Chunk，使其被 free 调用后进入 Fast Bin。由于 glibc 会对进入 Fast Bin 的 Chunk 做一些基础的大小和对齐检查，因此使用这个技巧需要注意构造 Fake Chunk 的大小和对齐方式。

这种利用技巧会导致 Fake Chunk 被链接到 Fast Bin 上。

(2) tcache house of spirit

tcache house of spirit 是 Tcache 中的 house of spirit，也就是将 Fake Chunk 加入 Tcache 的利用技巧。

将 Chunk 加入 Tcache 的 tcache_put 函数没有对加入的 chunk 实施过多检查，除了在 glibc≥2.32 中引入的 Safe-Linking 检查（将在后文详细描述）。只要 Fake Chunk 的大小在 tcache 的范围内（0x20 - 0x410），它就可以被释放并加入 Tcache。

这种利用技巧会导致 Fake Chunk 被链接到 Tcache 上。

3.1.3 污染大小

污染大小旨在控制 Chunk 的 size 和 prev_size 字段，这通常需要溢出才能达到。但得益于 glibc 中为了提高空间利用率采取的一个特殊机制，对攻击者而言这并不是一件难事。

在 glibc 的堆实现中，两个 Chunk A 和 B 如果是物理相邻的，那么 A 可以使用 B 的 prev_size 字段存储它的数据部分。这是因为 B 的 prev_size 字段只有在 A 为空闲时才会发挥作用。这个机制意味着 Chunk A 可以直接访问和修改 B 的 prev_size 字段，而并不会造成越界。而且，如果希望篡改 B 的 size 字段，也只需要在 A 中制造单字节溢出（off-by-one）即可。

如果一个 Chunk（或者与之相邻的 Chunk）的大小被篡改，堆管理器将错误地判断 Chunk 之间的位置关系，某些操作可能造成 Chunk 重叠，甚至可能达到任意地址写入的效果。

（1）overlapping chunks

overlapping chunks 是由 François Goichon 在 2015 年描述的[22]。它是一种通过篡改 size 字段来使 Chunk 重叠的技巧。

当三个 Chunk A、B 和 C 物理相邻时，如果 B 被释放，并且 A 溢出到 B 的 size 字段，使其变大，那么下次 malloc 得到的 Chunk D 就会与 C 重叠。在 D 上的读写操作可能会导致对 C 的读写。

这种利用技巧会导致 Chunk 重叠。

（2）mmap overlapping chunks

mmap overlapping chunks 是由 Maxwell Dulin 提出的[23]，它是 mmap 版本的 overlapping chunks。

与 overlapping chunks 中的普通 Chunk 不同，Mmap Chunk 具有不同的内部结构，因此需要使用不同的方法。Mmap Chunk 包含 prev_size 和 size 字段。size 表示 Chunk 的大小，prev_size 表示 Chunk 剩余的空间（由于页面对齐等原因会产生一些剩余空间）。

如果 A 和 B 是相邻的两个 Mmap Chunk，A 的 size 或 prev_size 字段被溢出扩大。然后 A 被释放，这样 A 和 B 都将被归还给内核空间（munmap 函数将根据 prev_size + size 来确定要释放的内存映射区大小）。接下来，如果分配一个大的 Mmap Chunk C，则它将与 B 重叠。

这种利用技巧会导致 Chunk 重叠。

（3）poison null byte（Shrink Free Chunk 版）

poison null byte（Shrink Free Chunk 版本）也是由 François Goichon 在 2015 年提出的[22]。这是一种使用 off-by-null 溢出来导致 Chunk 重叠的技巧。

off-by-null 是指将一个 0x00 字节溢出到边界之外。这种溢出通常是由于在数组末尾写 '\0' 结束字符时，边界判断错误导致的。

Shrink Free Chunk 涉及多次堆操作，其流程如图 3.1。[24]

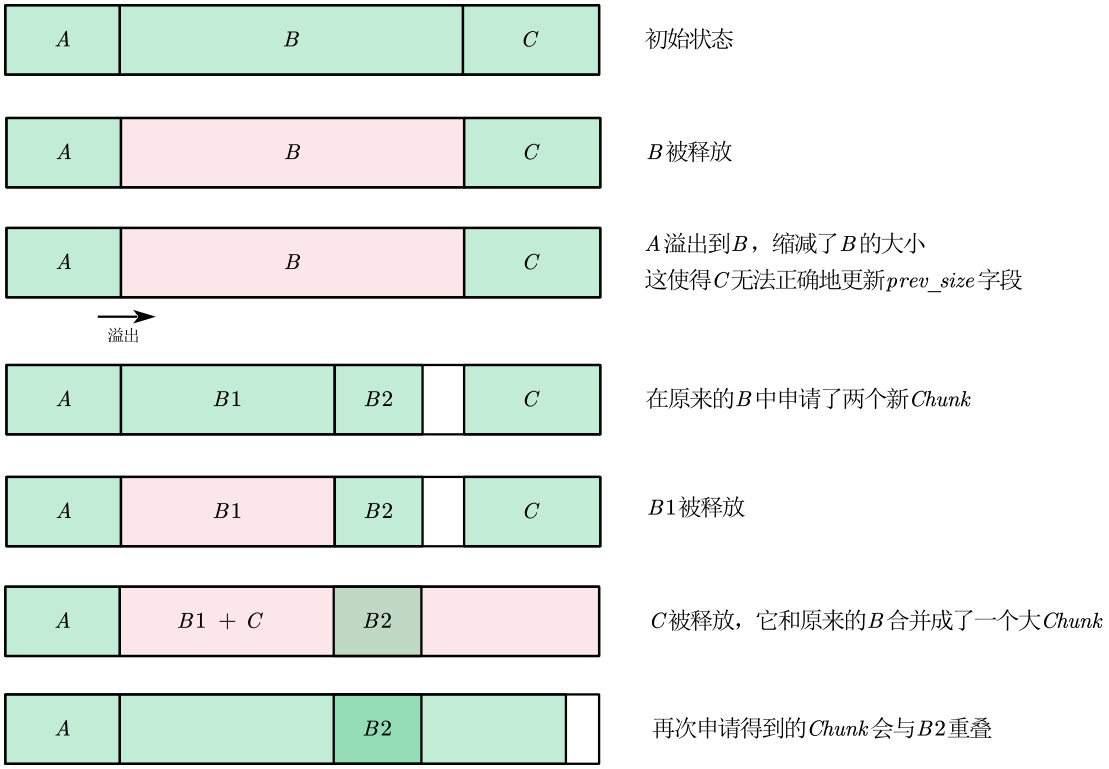

图 3.1　Shrink Free Chunk 的流程

1. 首先分配三个 Chunk A、B 和 C（需要落入 Unsorted Bin，因为 Fast Bin 和 Tcache 中的 Chunk 不会被分割）。
2. 然后释放 B。
3. 接着 A 的单个 0x00 字节溢出使 B 的 size 字段的最低字节置零，相当于缩小了 B 的 size。
4. 在 B 上重新分配了 B1 和 B2。此时，C 的 prev_size 应该更新，但实际上没有更新。堆管理器不认为 B2 和 C 之间有空闲块。
5. 接下来释放 B1。
6. 最后释放 C。此时 C 发现它减去 prev_size 位置的块（也就是 B1）已经空闲了，于是将其合并，产生了一个 B+C 的空闲 Chunk。这个空闲 Chunk 中包含仍在使用的 B2。
7. 在空闲 Chunk 上分配一个较大的 Chunk，它将与 B2 重叠。

这种利用技巧会导致 Chunk 重叠。

(4) house of force

house of force 也是在《Malloc Maleficarum》[21] 中提出的，它是一种覆写 Top Chunk 的 size，控制 malloc 返回值的技巧。

当 Top Chunk 的 size 被溢出为 -1（0xffffffffffffffff）时，它将包含整个虚拟地址空间。如果攻击者巧妙地构造 malloc(request_size)，就可以使下一次 malloc 的返回地址为地址空间内的任意值。这个技巧的关键在于 request_size 的计算。要构造 request_size，使得 malloc 返回攻击者希望的值，需要知道目标位置与 Top Chunk 的距离。

这种利用技巧会导致 malloc 返回指定的值。

3.1.4 污染链表

污染链表的目标是 Chunk 在 Bin 上时使用的链表指针，在 Fast Bin 和 Tcache 中是 fd，在 Unsorted Bin 和 Small Bin 中是 fd 和 bk，在 Large Bin 中是 fd、bk、fd_nextsize 和 bk_nextsize。

对 Chunk 中这些字段的污染主要是通过释放后使用达到的（当然也可以通过堆溢出）。这是因为对使用中的 Chunk 而言，链表指针都位于它的数据段之中。

被污染的链表指针会让堆管理器错误地判断 Bin 中包含的元素。攻击者使用此攻击手法主要是为了将 Fake Chunk 链接到 Bin 上，同时也可以实现一些其他的目的。

(1) fast bin dup into stack

这种利用方法也被称为 fast bin attack，它是通过修改 Fast Bin 中 Chunk 的 fd 指针将 Fake Chunk 添加到 Fast Bin 中的技巧。

如果能够控制 Fast Bin 上的一个已释放 Chunk，通过构造该 Chunk 的 fd 指针指向 Fake Chunk，就可以将 Fake Chunk 添加到 Fast Bin 中。为了绕过 glibc 在 Fast Bin 中的检查，需要精心设计 Fake Chunk 的大小。

自 glibc 版本 2.32 开始，引入了 Safe-Linking 检查，使得这种技巧变得十分困难。后文将详细描述该检查的实现。

这种利用技巧会导致 Fake Chunk 被链接到 Fast Bin 中。

(2) tcache poisoning

tcache poisoning 是修改 Tcache 中 Chunk 的 next 指针将 Fake Chunk 添加到 Tcache 中的技巧。

Tcache 并未对其中的 Chunk 做太多检查，攻击者只需要将某个 Tcache Chunk 的 next 指针指向一个 Fake Chunk，就可以将其加入 Tcache。同样地，Safe-Linking 检查也使得这种技巧变得十分困难。

这种利用技巧会导致 Fake Chunk 被链接到 Tcache 中。

(3) house of lore

house of lore 也是《Malloc Maleficarum》[21] 的一部分。最初，它是针对 Small Bin Chunk 和 Large Bin Chunk 的 bk 指针污染和 Fake Chunk 链接技巧，但是 Large Bin 实现的变化使得 house of lore 的 Large Bin 版本失效了。因此，此节只讨论其 Small Bin 版本。而其 Large Bin 版本最终发展成了下文的 large bin attack。

与 Fast Bin 和 Tcache 不同，Small Bin 是 LIFO 的双向链表。glibc 会从 Small Bin 的尾部取出 Chunk，并将取出 Chunk 的 bk 指向的 Chunk 链接在 Small Bin 的尾部。这意味着，如果想将 Fake Chunk 加入 Small Bin，应将其链接到 Small Bin Chunk 的 bk 指针上。

由于 glibc 对双向链表的有效性进行了检查，攻击者需要绕过该检查才能污染链表。具体来说，需要确保 Chunk →bk →fd 等于 Chunk 本身。为了绕过此检查，攻击者需要使 Fake Chunk 的 fd 指向 Small Bin Chunk，这意味着需要获得堆上的地址。然而，堆通常启用了 ASLR 机制，这将增加攻击者的难度。

这种利用技巧会导致 Fake Chunk 被链接到 Small Bin 中。

(4) unsorted bin attack

unsorted bin attack 是修改 Unsorted Bin Chunk 的 bk 指针，将 Fake Chunk 加入 Unsorted Bin 的技巧。

与 Small Bin 类似，要想将 Fake Chunk 加入 Unsorted Bin，需要篡改一个 Unsorted Bin Chunk 的 bk 指针指向 Fake Chunk。

但值得指出的一点是，当 Unsorted Bin Chunk 被取出时，Chunk→bk →fd 会被更新为 Unsorted Bin 链表头的地址。如果 Chunk→bk 是攻击者的 Fake Chunk，这

意味着 Fake Chunk →fd 处将会被写为 Unsorted Bin 链表头的地址。这可以用来实现任意地址写入指定值。虽然攻击者不能控制写入的值，但这仍然十分有用。一个常见用例是将某个阈值覆盖为一个极大值（Unsorted Bin 链表头地址可以看作一个大整数），以引发进一步的越界读/写问题。而且，写入的值是堆中的一个地址，攻击者可以用它来覆盖另一个指针，从而导致进一步的溢出和破坏。

自 glibc 版本 2.28 引入的 Unsorted Bin 双向链表有效性检查（Chunk →bk →fd = Chunk）使得该技巧的实施变得困难。

这种利用技巧会导致 Fake Chunk 被链接到 Unsorted Bin 中，同时还会导致任意地址写入指定值。

（5）large bin attack（Smaller Chunk 版）

large bin attack（Smaller Chunk 版）是修改 Unsorted Bin Chunk 的 bk 指针和 bk_nextsize 指针，实现任意地址写入指定值的技巧。

这个堆利用的过程是，修改一个 Large Bin 中 Chunk（假设叫做 Controlled Chunk）的 bk 和 bk_nextsize 指针，使它们分别指向两个目标地址。然后再令一个 Chunk（假设叫做 Victim Chunk）加入同一个 Large Bin。Victim Chunk 插入 Large Bin 双向链表时，会触发指针更新操作，导致这两个目标位置被写为 Victim Chunk 的地址。这个过程中 Large Bin 的指针更新较为复杂，不再花篇幅介绍。

所谓的“Smaller Chunk 版”指的是 Controlled Chunk 比之后加入 Large Bin 引发漏洞利用的 Victim Chunk 小。因为 Large Bin 中的 Chunk 是从大到小排序的。Controlled Chunk 与 Victim Chunk 的相对大小将改变 glibc 的执行路径，引发不同的指针更新。

这种利用技巧会导致任意地址写入指定值。但自 glibc 版本 2.30 引入的 Large Bin 双向链表有效性检查使 Smaller Chunk 版的 large bin attack 不再可能。

（6）large bin attack（Larger Chunk 版）

large bin attack（Larger Chunk 版）是 Controlled Chunk 比 Victim Chunk 大的 large bin attack。由于执行路径的不同，这个版本的 large bin attack 只能修改 Controlled Chunk 的 bk 指针，从而将一个目标位置写为 Victim Chunk 的地址。但该执行路径下的 large bin attack 至今（glibc 版本 2.37）仍未修复。

这种利用技巧会导致任意地址写入指定值。

(7) tcache stashing unlink attack

tcache stashing unlink attack 是利用 Small Bin 命中将 Fake Chunk 链入 Tacche 的技巧。

malloc 中匹配到合适的 Small Bin Chunk 时，glibc 会将 Small Bin 中剩余的 Chunk 放入 Tcache。这个技巧就是想借此将 Fake Chunk 加入 Tcache。攻击者会先将 Fake Chunk 链接到 Small Bin Chunk 的 bk 指针上，然后触发 Small Bin Chunk 移入 Tcache。

值得指出的是，在将 Small Bin Chunk 移入 Tcache 的过程中会将 Small Bin Chunk →bk →fd 置为 Small Bin 链表头的地址。这又能达到任意地址写入指定内容的效果。

这种利用技巧会导致 Fake Chunk 加入 Tcache，还会导致任意地址写入指定值。

(8) fast bin reverse into tcache

fast bin reverse into tcache 是 Fast Bin 版本的 tcache stashing unlink attack，它是利用 Fast Bin 命中将 Fake Chunk 链入 Tache 的技巧。

malloc 中匹配到合适的 Fast Bin Chunk 时，glibc 会将 Fast Bin 中剩余的 Chunk 放入 Tcache。攻击者将 Fake Chunk 链接到 Fast Bin Chunk 的 fd 指针上，然后触发上述过程，就能将 Fake Chunk 移入 Tcache。

在将 Fake Chunk 插入 Tcache 头部时，会更新 Fake Chunk 的 next 指针指向下一个 Tcache Chunk。因此该利用技巧也有任意地址写入指定内容的效果。

这种利用技巧会导致 Fake Chunk 加入 Tcache，还会导致任意地址写入指定值。

(9) house of rabbit

house of rabbit 于 2017 年提出[25]，它是一种类似于 house of force，但无需修改 Top Chunk 的大小，而是通过链接 Fake Chunk 实施的堆利用技巧。

攻击者将 Fake Chunk 的 size 设为 0xffffffffffffffff（整个虚拟地址空间的大小），然后篡改一个 Fast Bin Chunk 的 fd 指针，使其指向 Fake Chunk。接下来触发 malloc_consolidate，将 Fake Chunk 移入最后一个 Large Bin（该 Large Bin 中 Chunk 的

大小范围从 0x20000 到无限大）。最后，构造 malloc(request_size)，就可以使下次 malloc 的返回地址为地址空间内的任意值。

与 house of force 相比，这种方法不需要知道 Top Chunk 的地址（由于 ASLR 机制，通常很难获得），只需要知道 Fake Chunk 的地址即可构造 request_size。

这种利用技巧会导致 malloc 返回指定的值。

（10）house of gods

house of gods 是 Milo 在 2022 年提出的一种新堆利用技巧[26]，它非常具有破坏性，虽然实施起来相对困难。它是一种构造 Fake Arena 的技巧。

详细阐述 house of gods 的利用过程需要耗费大量篇幅。简言之，这个技巧使用了以下思想：

1. 将 Arena 结构视为一个 Fake Chunk（假设叫做 Arena Chunk），通过篡改 Unsorted Bin Chunk 的 bk 指针将 Arena Chunk 加入 Unsorted Bin。
2. 从 Unsorted Bin 中分配得到 Arena Chunk，修改其中的 next 指针（正常情况下指向下一个可用的 Arena）指向 Fake Arena。
3. 通过 malloc 两次极大值触发 arena_get_retry 函数，使线程尝试从 next 指针中获取可用的 Arena。这样，Fake Arena 就会被当作可用的 Arena 提供给线程使用。

这种技巧会导致对 Arena 的劫持。由于 Arena 是最大的堆结构，包含了所有其他堆结构，因此一旦 house of gods 实施成功，攻击者将获得极其强大的能力。

3.1.5 污染标志位

标志位是 Chunk 中 size 字段的低三个比特，它们分别是 P 比特（物理相邻的前一个 Chunk 是否空闲）、M 比特（当前 Chunk 是否由 mmap 分配）以及 N 比特（当前 Chunk 是否隶属于主线程的 Arena）。污染标志位的攻击手法主要通过 off-by-one 实现，因为溢出的一个字节已经足以污染下一个 Chunk 的标志位。

对 P 比特的污染通常用来误导堆管理器，使其认为与 Chunk 相邻的上一个 Chunk 处于空闲状态，可以被合并。对 N 比特的污染通常旨在令 Chunk 进入伪造的 Arena。但目前为止还没有发现针对 M 比特进行污染的堆利用方法。

(1) unsafe unlink

unsafe unlink 是一个流行了二十多年的经典堆利用方法。它是一种通过对 Fake Chunk 触发 unlink 实现任意地址写入的技巧。

总的来说，unsafe_unlink 试图通过 off-by-null 漏洞将相邻下一个 Chunk 的 P 比特清零，然后在当前 Chunk 的数据段中伪造一个 Fake Chunk。这样，在下一个 Chunk 释放时，Fake Chunk 就会被认为是空闲的，并被执行后向合并。在后向合并过程中，被合并的 Chunk 需要从原来的 Bin 链表上移除，这个操作由 unlink 函数实现。unlink 函数会对链表指针进行如下操作：

```
1  Chunk->fd->bk = Chunk->bk
2  Chunk->bk->fd = Chunk->fd
```

然而，在 unlink 函数中，glibc 也会检查链表的有效性，即 Chunk →fd →bk = Chunk 和 Chunk →bk →fd = Chunk。攻击者需要精心构造 Fake Chunk 的 fd 和 bk 指针，使其指向栈上或 BSS 段上的某个位置。这样不仅可以绕过链表有效性检查，还能够利用 unlink 的指针操作实现任意地址写入的效果。

这种技巧会导致任意地址写入任意值。

(2) poison null byte（Unlink Fake Chunk 版）

poison null byte（Unlink Fake Chunk 版）与 unsafe_unlink 十分相似。不同的是，unsafe_unlink 利用栈区构造假想 Chunk 来绕过链表有效性检查，而 Unlink Fake Chunk 使用堆区构造精心的 Chunk 绕过。该技巧通过对 Fake Chunk 触发 unlink 实现 Chunk 重叠的效果。

简单来说，Unlink Fake Chunk 攻击利用了从 Large Bin 中取出 Chunk 时 fd_nextsize 和 bk_nextsize 不会被清空的机制。如果在 Chunk 的数据段起始部分构造 Fake Chunk，那么 fd_nextsize 和 bk_nextsize 恰好处于 Fake Chunk 的 fd 和 bk 位置。攻击者将三个 Chunk 在 Large Bin 上恰当地连接并取下，就可以使得 Fake Chunk →fd →bk = Fake Chunk 且 Fake Chunk →bk →fd = Fake Chunk。

在绕过链表有效性检查后，攻击者同样使用 off-by-null 漏洞将相邻下一个 Chunk 的 P 比特清零，从而触发 Fake Chunk 的 unlink 操作。这样，Fake Chunk 就会被合并到一个大 Chunk 中。下次 malloc 分配得到的 Chunk 可能会和 Fake Chunk 重叠。

这种技巧会导致 Chunk 重叠。

(3) house of einherjar

house of einherjar 由 st4g3 在 2016 年提出[27]。house of einherjar 也是污染 P 比特使其置零的技巧，但它旨在控制 Top Chunk 的指针，使其指向 Fake Chunk。

当一个 Chunk 被释放时，会先进行后向合并再进行前向合并。如果这个 Chunk 前向紧邻着 Top Chunk，它就会和 Top Chunk 合并。如果攻击者能够使用 off-by-null 漏洞溢出清空这个 Chunk 的 P 比特，使其认为它后向的块是空闲的，那么就能触发后向合并。

通过设置 prev_size 的大小，使得后向合并刚好将 Chunk 合并到攻击者构造的 Fake Chunk 的位置，最终就能将 Fake Chunk 合并入 Top Chunk 中。这样，下一次执行 malloc 时，就会返回 Fake Chunk。

这种技巧可以导致 malloc 返回指定的值。

(4) house of mind fast bin

house of mind 最初由 Phantasmal Phantasmagoria 在《Malloc Maleficarum》[21] 中提出，但原始方法已经无法使用。其现代版本——house of mind fast bin，则是由 Maxwell Dulin 在 2021 年提出的[28]。它是一种污染 N 比特，使 Chunk 进入 Fake Arena 的利用技巧。

在研究基础一章中，已经介绍过 Heap 和 Arena 的组织方式。heap_info 结构总在内存中 0x4000000 的倍数处，而非主 Arena Chunk 则通过将低 26 位设为零来获取 heap_info。

house of mind fast bin 首先伪造 heap_info 结构体，使其指向攻击者希望写入的目标地址附近（假设那里有一个 Fake Arena）。然后通过堆喷射等方法将伪造的 heap_info 结构置于内存中与 0x4000000 对齐的位置。接下来，利用 off-by-one 漏洞将一个 Chunk 的 N 比特污染为 1，并释放该 Chunk，它会被加入 Fake Arena 的 Fast Bin 中。因此，目标地址（Fake Arena 的 Fast Bin 链表处）就会被写为 Chunk 的地址。

这种技巧可以导致任意地址写入指定值。

3.2 堆利用模型

通过以上分析，已经可以对堆利用的技巧和方法产生较为全面的理解。目前，攻击者拥有一些堆利用技巧，这些技巧可能导致 Fake Chunk 被链接或 Chunk 重叠等。但对攻击者而言，这还不足以实现一次完整的攻击。一次完整的堆利用攻击通常会产生比 Chunk 重叠更严重的后果，例如控制流劫持或数据泄露。那么，完整的攻击是如何实现的？为此，本文提出了堆利用的利用流模型。

3.2.1 基本概念

为了说明该模型的含义，首先需要澄清一些基本概念。这些概念大部分已经是漏洞利用行业的通用术语，但很少得到清楚的定义。[29] 在本文的语境中，它们的定义如下：

(1) 漏洞

漏洞（Vulnerability）是程序中的编码或设计错误，它可能导致潜在的安全问题。

(2) 利用原语

利用原语（Exploit Primitive）是攻击者实施利用的过程中获得的基本能力。一些常见的利用原语包括溢出、任意内存读/写、任意函数调用等。

(3) 利用技巧

利用技巧（Exploit Technique）是一种可重复使用的通用策略，用于将一个漏洞利用原语转换为另一个漏洞利用原语，转换后的利用原语通常比原始原语更强大。本文前一节所描述的都是针对堆的利用技巧。

(4) 利用流

利用流（Exploit Flow）是一系列利用技巧，它从漏洞提供的初始利用原语出发，一步步产生更强的原语，最终达到攻击的目标。

3.2.2 堆利用原语

为了建立堆的利用模型，首先需要区分出堆利用中存在的原语。但是，堆利用原语并没有固定的标准，Insu Yun 等在他们的文章[8] 中为他们的堆漏洞模型提出了四种漏洞原语和四种攻击原语。然而，由于原语之间存在相互蕴含和推导的关系，简单地将它们分成漏洞导致和攻击导致是不恰当的。因此，本文在此基础上提出了堆漏洞的五种初级原语、五种中级原语和两种高级原语，以更好地描述堆利用攻击的过程。

(1) 初级原语

五种堆利用初级原语如表 3.1 所示。

表 3.1 初级原语

初级原语	含义
DOUBLE-FREE (DF)	对同一个 Chunk 调用两次 free
USE-AFTER-FREE (UAF)	读写一个已经释放的 Chunk
OVERFLOW (OF)	溢出数据到其它 Chunk
ARBITRARY-FREE (AF)	free 任意地址
HEAP-ADDRESS-LEAK (HAL)	获知堆上的地址

其中 *OVERFLOW* 原语下有两个子原语，见表 3.2。

表 3.2 *OVERFLOW* 的子原语

初级原语	含义
OFF-BY-ONE (O1)	溢出一个字节到相邻 Chunk
OFF-BY-NULL (O1N)	溢出一个 0x00 字节到相邻 Chunk

大多数情况下，这七种初级原语由程序漏洞产生，被堆利用技巧作为初始条件。

(2) 中级原语

五种中级原语如表 3.3 所示。

大多数情况下，中级原语由初级原语产生，是堆利用完成攻击的中间步骤。

表 3.3　中级原语

中级原语	含义
CHUNK-LINKED-TWICE (CLT)	同一个 Chunk 在 Bin 上被引用两次
FAKE-CHUNK-LINKED (FCL)	伪造的 Fake Chunk 被放入 Bin
CHUNK-OVERLAPPING (CO)	两个 Chunk 重叠
ARENA-HIJACKING (AH)	使线程使用伪造的 Arena
MALLOC-HIJACKING (MH)	控制 malloc 的返回值

(3) **高级原语**

表 3.4 是两种高级原语。

表 3.4　高级原语

高级原语	含义
WRITE-WHAT-WHERE (3W)	向任意地址写任意内容
WRITE-WHERE (2W)	向任意地址写指定内容

高级原语是堆利用能提供的最强原语，可以作为影响范围更大的综合攻击的条件。

虽然初级原语、中级原语和高级原语在强大程度和利用路径上存在递进的关系，但它们之间并不是简单的初级推导出中级、中级推导出高级。实际上，在堆利用攻击的过程中，它们之间可以相互转换，这取决于攻击者的最终目的。

3.2.3　堆利用技巧

利用技巧是一种将一个利用原语转换为另一个利用原语的策略。本文考察了 25 种堆利用技巧，这些技巧通常需要满足某些实施条件才能实现特定的攻击效果。这些实施条件既有公共的，也有基于特定利用原语的。本文提出的堆利用模型认为所有堆利用技巧都拥有以下两点公共实施条件：

1. 攻击者应当可以以任意顺序分配任意大小的内存，并以任意顺序释放这些内存。这意味着攻击者能够不限制次数地调用 malloc，同时 malloc 的参数可以是任意值。他还能够按任何顺序调用 free。
2. 攻击者应当可以在合法的内存区域写入任意数据。这意味着攻击者能够完全控制他分配的内存。

除了公有条件，堆利用技巧所依赖的其它条件便都是基于利用原语的。每个堆利用技巧都有它依赖的利用原语和它实施后能产生的利用原语。

以前文描述的 overlapping chunks 为例，它通过篡改下一个 Chunk 的 size 字段来造成 Chunk 重叠。篡改下一个 Chunk 的 size 字段通常需要一个 off-by-one 漏洞，它的实施结果是 Chunk 重叠。因此，overlapping chunks 的依赖原语是 *OFF-BY-ONE*，结果原语是 *CHUNK-OVERLAPPING*。通过这个示例可以看出，利用原语为利用技巧的输入和输出提供了抽象，将繁琐的利用细节隐藏其中。这对于思考攻击流程的攻击者而言是十分有益的。

本文整理了所有 25 种利用技巧的依赖原语和结果原语，如表 3.5 所示。

表 3.5　堆利用技巧的原语

堆利用技巧	依赖原语	结果原语
fast bin dup	*DF*	*CLT*
fast bin dup consolidate	*DF*	*CLT*
tcache dup	*DF*	*CLT*
house of kauri	*(OF/O1)&DF*	*CLT*
house of botcake	*DF*	*CLT*
house of spirit	*AF*	*FCL*
tcache house of spirit	*AF*	*FCL*
overlapping chunks	*O1*	*CO*
mmap overlapping chunks	*OF*	*CO*
poison null byte（Shrink Free Chunk 版）	*O1N*	*CO*
house of force	*(OF/UAF)&HAL*	*MH&3W*
fast bin dup into stack	*OF/UAF*	*FCL*
tcache poisoning	*OF/UAF*	*FCL*
house of lore	*(OF/UAF)&HAL*	*FCL*
unsorted bin attack	*OF/UAF*	*FCL&2W&HAL*
large bin attack（Smaller Chunk 版）	*OF/UAF*	*2W&HAL*
large bin attack（Larger Chunk 版）	*OF/UAF*	*2W&HAL*
tcache stashing unlink attack	*OF/UAF*	*FCL&2W&HAL*
fast bin reverse into tcache	*OF/UAF*	*FCL&2W&HAL*
house of rabbit	*OF/UAF*	*MH&3W*
house of gods	*(OF/ UAF)&HAL*	*AH*
unsafe unlink	*O1N*	*3W*
poison null byte（Unlink Fake Chunk 版）	*O1N*	*CO*
house of einherjar	*O1N&HAL*	*MH*
house of mind fast bin	*OF&HAL*	*2W*

3.2.4 堆利用流

(1) 原语的蕴含

原语之间的关系并非简单的相互独立，一些原语可以很容易地转化为另一种原语。

例如，如果攻击者有一个 *CHUNK-LINKED-TWICE* 的原语，即某个 Chunk 在 Bin 上被引用了两次。那么如果攻击者再次通过 malloc 获得这个 Chunk，由于该 Chunk 仍然被视为已释放状态（因为在 Bin 中还有一份引用），对该 Chunk 的修改将产生 *USE-AFTER-FREE* 原语。

然而，*CHUNK-LINKED-TWICE* 原语显然不能与 *USE-AFTER-FREE* 原语划等号。从 *CHUNK-LINKED-TWICE* 到 *USE-AFTER-FREE* 的推导只能说明 *USE-AFTER-FREE* 被 *CHUNK-LINKED-TWICE* 包含。这种原语之间的包含关系称为原语的蕴含。

经过考察，本文提出了 10 种堆利用原语中存在的蕴含关系，它们如表 3.6 所示。这些蕴含关系大多是中级原语对低级原语的蕴含，也有一些中级原语之间的蕴含。

表 3.6 原语的蕴含关系

原语	被蕴含原语
CHUNK-LINKED-TWICE	*MALLOC-HIJACKING*
CHUNK-LINKED-TWICE	*USE-AFTER-FREE*
CHUNK-LINKED-TWICE	*CHUNK-OVERLAPPING*
FAKE-CHUNK-LINKED	*MALLOC-HIJACKING*
FAKE-CHUNK-LINKED	*HEAP-ADDRESS-LEAK*
OVERFLOW	*OFF-BY-ONE*
OFF-BY-ONE	*OFF-BY-NULL*
CHUNK-OVERLAPPING	*HEAP-ADDRESS-LEAK*
AREAN-HIJACKING	*MALLOC-HIJACKING*
AREAN-HIJACKING	*HEAP-ADDRESS-LEAK*

(2) 原语的发展

很容易注意到，表 3.5 中，某些堆利用技巧的结果原语是其它利用技巧的依赖原语。以 fast bin dup 为例，这个利用技巧能产生一个 *CHUNK-LINKED-TWICE* 原

语，而 *CHUNK-LINKED-TWICE* 原语蕴含了 *USE-AFTER-FREE* 原语，这恰好是实施 fast bin duo into stack 技巧所依赖的。因此，fast bin dup 能够为 fast bin dup into stack 的实施创造条件。

利用流就是以这样的方式工作的：攻击者从小的漏洞出发，通过一系列利用技巧逐步制造更强的利用原语，最终达到攻击目的。此处的利用技巧不仅仅包括堆利用技巧。两个高级原语 *WRITE-WHAT-WHERE* 和 *WRITE-WHERE* 是堆利用可以提供的最强大的原语。一次成功的攻击却通常做得更多。譬如劫持程序的控制流，以致达到运行 shellcode 弹出 shell 的效果。这意味着需要综合使用除堆利用以外的其他利用技巧。例如，如果攻击者通过堆利用获得了一个 *WRITE-WHAT-WHERE* 原语，他可以使用该原语篡改 GOT 表，以使程序运行到 shellcode 所在的区域。

3.3 本章小结

本章首先按照 5 种不同的攻击手法讨论了 25 种堆利用技巧的机理。这 5 种攻击手法是双重释放、控制释放、污染大小、污染链表和污染标志位。

经过这些讨论，已经可以对堆利用如何实施产生基本的概念。但是，堆利用技巧是零散的、细节的，从技巧的罗列中并不能获知完整的堆利用图景。于是，本文提出了堆利用的利用流模型。

堆利用模型通过漏洞、利用原语、利用技巧和利用流的概念勾勒出了堆利用攻击的整体轮廓。一次完整的堆利用攻击从漏洞出发，提取初始原语，并综合使用诸多利用技巧，以达到攻击目的。这个过程就是堆利用的利用流。

本文还总结了堆利用中的原语，并提出了 5 个初级原语、5 个中级原语和 2 个高级原语，以及它们之间的蕴含和推导关系。此外，本文还整理了 25 个堆利用技巧的依赖和结果原语。这为堆利用图景提供了必要的细节。

4 堆安全机制

前一章已经对堆利用的机理与方法进行了详尽的讨论，其中也涉及到绕过堆中的某些安全检查的技巧。实际上，理解现有机制的设计，并发现其中存在的缺陷，也是漏洞利用的攻击者关注的重要内容。在现实中成功利用堆漏洞，必须理解阻止堆利用的安全机制，并设计对应的绕过方法。本章将对这些堆安全机制进行梳理，其中包括操作系统中通用的内存保护机制，以及 glibc 最新版本（2.37）中包含的 42 个堆安全检查。

这 42 个堆安全检查可以被分为 5 类：阻止双重释放、确保参数合理、防止篡改大小、加固双向链表和加固单向链表。与前一章提到的 5 种攻击手法的对应如表 4.1 所示。

表 4.1 攻击手法与安全检查的对应关系

攻击手法	glibc 的应对措施
双重释放	阻止双重释放
控制释放	确保参数合理
污染大小	防止篡改大小
污染链表	加固双向链表、加固单向链表
污染标志位	防止篡改大小

本文将归纳出这 5 类安全检查的一般性方法，并挑选一些典型样例予以阐释。为了方便研究者参考，完整的 42 个安全检查可在附录 A 中查阅。

4.1 通用内存保护机制

现代操作系统中引入了许多通用的内存保护机制，这些机制并不是专门为阻止堆漏洞利用而开发的，但它们有效地增加了攻击者攻击的难度。这些常见的内存保护机制包括：

4.1.1 ASLR 地址空间布局随机化

ASLR 是操作系统的一种内存保护机制，旨在确保进程运行过程中各内存段的地址不可预测，从而使与这些地址相关的漏洞更加难以利用。

在 GNU/Linux 中，进程的堆区和 mmap 区都默认启用了地址随机化（见图 2.1）。这使获得堆上的地址变得困难。

4.1.2 NX 不可执行位

NX 位是内存页上的一个标识，用于区分可执行代码和不可执行数据区。通过将程序数据区的 NX 位设为不可执行，可以防止试图在数据区构造代码的漏洞利用方法。

在 GNU/Linux 中，只有程序的代码段是可执行的，其余的数据段、栈和堆都是不可执行的。这意味着无法从堆上执行 shellcode 等恶意代码。

4.1.3 RELRO 重定位只读

RELRO 是 GNU/Linux 中特有的保护机制。在 ELF 可执行文件中，有.dynamic、.got、.plt 等负责外部函数重定位的区段，其中所有外部库的函数地址都写在 GOT 表中。RELRO 机制将重定位段设置为只读，从而防止黑客篡改外部函数重定位地址控制程序流程。对于堆利用攻击而言，这意味着利用堆漏洞篡改 GOT 表的方法不再可行。

4.2 阻止双重释放

长期以来，为了提升堆的性能，glibc 没有引入强有力的防止双重释放的措施。这些一般性的方法只在某些最容易出现双重释放的程序执行路径下执行检查，其覆盖范围有限，容易被攻击者绕过。直到 glibc 2.29 版本中，堆管理器开发者为 Tcache 引入了一种新机制，可以极其有效地防止针对 Tcache 的双重释放攻击。

4.2.1 一般性方法

一个 Chunk 被释放后有三种去向：

1. 进入 Fast Bin 或 Tcache 等缓存 Bin。
2. 与其前向或后向相邻的 Chunk 合并。
3. 无法合并，直接进入 Unsorted Bin。

对于 Fast Bin 而言，在将空闲 Chunk 加入其中时，glibc 会检查它是否等于 Fast Bin 的头部 Chunk。这可防止连续释放同一块 Fast Bin Chunk。

对于合并的 Chunk 而言，glibc 只会检查已释放 Chunk 是否与 Top Chunk 重叠。这能阻止合并入 Top Chunk 的空闲 Chunk 被双重释放。

对于无法合并的 Chunk 而言，双重释放检查非常简单。glibc 只需检查其物理相邻的下一个 Chunk 的 P 比特是否为 0。

4.2.2 Tcache 的 key 机制

从 glibc 2.26 版本引入 Tcache 开始，一直到 glibc 2.28 版本，对于 Tcache 的双重释放都没有任何保护措施。然而，在 glibc 2.29 中，glibc 为 Tcache Chunk 结构体引入了一个新字段，用于防止双重释放：

```
1  typedef struct tcache_entry
2  {
3    struct tcache_entry *next;
4    /* This field exists to detect double frees.  */
5    struct tcache_perthread_struct *key;
6  } tcache_entry;
```

这个名为 key 的字段会在 Chunk 从 Tcache 取出时置为 NULL，而在 Chunk 进入 Tcache 时指向线程的 tcache_perthread_struct 结构体，起到标识 Chunk 在该线程的 Tcache 中的作用。

每次将 Chunk 加入 Tcache 之前，glibc 都会检查 Chunk 的 key 字段，看它是否等于当前线程的 tcache_perthread_struct 地址。如果相等，则说明这个 Chunk 已经在 Tcache 中存在过了，它是被双重释放的。

这个检查既简单又有效，但仍然有绕过的方法。前文提到的 house of botcake 就是其中一种方法，它将 Tcache Chunk 先释放进入 Tcache，再次释放进入 Unsorted Bin，从而混淆堆管理器。

4.3 确保参数合理

控制释放的攻击手法需要能让用户程序对任意地址调用 free。但对于 free 函数而言，它应该只接受之前调用 malloc 返回的指针作为参数。跟踪每次 malloc 的返回值会带来极大的性能开销，因此 glibc 并不打算这么做。尽管如此，它仍然对 free、munmap 和 realloc 等函数的参数进行了基本的合理性检查。

举例来说，“free(): invalid pointer” 检查会检查 free 的参数是否是 16 字节对齐的。前文已提到，所有由 glibc 分配的 Chunk，其大小和地址都是 16 字节对齐的。对齐检查虽然不够充分，但仍然极大地增加了攻击者的难度。

除了对齐检查以外，glibc 还会检查被释放的 Chunk 物理相邻的前后 Chunk 是否合理。这意味着攻击者通常需要在 Fake Chunk 的前后构造一些辅助的 Chunk，从而进一步提高了攻击的难度。

4.4 防止篡改大小

Chunk 的大小和标志位都存储在 size、prev_size 两个字段中，但这两个字段却如此容易被溢出，以致于 glibc 中为此引入了 12 个安全检查。

其中最经典的是一系列 “corrupted size vs. prev_size” 检查。它使用了一个简单的思想，即检查 Chunk 的 size 字段是否等于物理相邻的下一个 Chunk 的 prev_size 字段，该检查遍布堆管理器各处，包括 unlink 函数、malloc_consolidate 函数、Unsorted Bin 遍历等。

此外，类似 house of force 这样的利用技巧会将 Chunk 的大小置为-1，从而使 Chunk 包含整个 X86_64 虚拟地址空间。为此，glibc 也引入了对 Chunk 大小范围的检查，要求 Chunk 的大小在 $[2 * sizeof(size\_t), system\_mem]$ 区间。其中前边界是 Chunk 的最小大小，后边界是堆的总内存。

对 Chunk 大小的严格检查是困难的，堆管理器只能进行一些基本的合理性检查。只要攻击者稍有技巧，就可以为 Chunk 构造一个合理的大小，使这些检查失去作用。

4.5 加固双向链表

污染链表也是攻击者非常热衷的攻击手法，glibc 为此引入了 16 个检查，其中 10 个是针对 Unsorted Bin、Small Bin、Large Bin 等双向链表的。

glibc 对双向链表的检查只有一种方法，那就是看 Chunk →fd →bk 和 Chunk →bk →fd 是否等于 Chunk 本身。这被称为 “corrupted double-linked list” 检查，它们会在 Chunk 取出或加入双向链表时执行。

这些检查虽然简单，但效率很高。这反映了 glibc 的堆设计原则：尽可能安全，同时尽量减少性能损失。攻击者为了绕过这些检查，发展出了极其复杂的技巧，如

unsafe unlink、house of einherjar 和 poison null byte（Unlink Fake Chunk 版）都是其中的经典案例。

4.6 加固单向链表

长期以来，glibc 中的单向链表都没有任何保护机制，攻击者可以轻易地将 Fake Chunk 链接到 Fast Bin 和 Tcache 链表中。直到 2020 年，Check Point 组织提出了 Safe-Linking 技术[30]，旨在改变单向链表缺乏保护的现状。这一技术如今被广泛应用于现代堆管理器，包括 ptmalloc2、tcmalloc 和 dlmalloc 等堆管理器都已经集成了 Safe-Linking。glibc 在 2.32 版本中首次引入该技术。

简而言之，Safe-Linking 做了两件事：

1. 利用 ASLR 随机地址与单链表中的 next 指针异或，使攻击者不能轻易伪造 next 指针。
2. 在操作单链表 Chunk 的各个地方加入对齐性检查，使攻击者不能轻易找到合适的目标地址。

4.6.1 加密 next 指针

glibc 的源码中引入了以下两个宏：

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

这两个宏用于对 next 指针进行加密和解密。其中 ptr 是 next 指针将要被赋的值，pos 是 next 指针的地址。所谓 next 指针的地址是指其在内存中的位置（也就是堆元数据的存储位置），这个地址必然位于堆上。加密后的 ptr 等于 pos 右移 12 位后与原 ptr 地址异或（见图 4.1[30]）。

这样的做法是为了利用 ASLR 随机数。ASLR 随机数是在程序运行时生成的，攻击者很难猜测到这个值。对堆区启用 ASLR 时，其随机基地址（所有堆地址中共同的部分）的掩码为：

```
random_base = ((1 << rndbits) - 1) << PAGE_SHIFT)
```

在 X86_64 架构中，rndbits 的值是 28，PAGE_SHIFT 的值是 12。将 next 指针

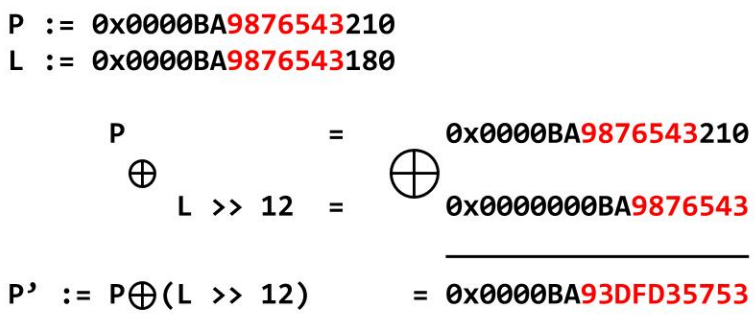

图 4.1　Safe-Linking 的加密过程
（P 表示 next 指针，L 表示 next 指针的地址）

的地址右移 12 位，这样它的最低位就是 ASLR 随机数的起始位。然后将其与原始的 next 指针进行异或运算，就起到了使用 ASLR 随机数加密 next 指针的效果。

要解密这个值，只需要再进行一遍相同的加密操作即可，因为两次异或运算会得到原始数据。

Safe-Linking 在 Chunk 加入单向链表时对链表指针执行了加密，在遍历单向链表和将 Chunk 从单向链表上取出时执行解密。由于加密和解密只需要一次移位运算和一次异或运算，它对性能的影响微乎其微。

4.6.2　对齐检查

加密 next 指针固然是一个不错的办法，但是如果一个无效的指针被错误地当作加密指针，那么它解密后得到的指针也应该是无效的。在这种情况下，如果不进行检验就直接使用该指针，将会导致无法预测的后果。因此，Safe-Linking 增加了一种方法来检验解密后的 next 指针是否合法。

正如前文所述，在 X86_64 架构中，glibc 中的所有 Chunk 都需要 16 字节对齐，这是为了提高一些指令的执行速度。这意味着 Chunk 地址的末 4 位必须为 0。基于这个特点，Safe-Linking 引入了一系列“unaligned chunk detected”检查，在每次解密时都会检查解密后的 next 指针是否符合 16 字节对齐要求。

Safe-Linking 技术大大降低了攻击者利用 Tcache 和 Fast Bin 的可能性，使覆盖或伪造 next 指针变得非常困难。此外，攻击者还很难找到合适的目标地址，因为目标地址必须满足 16 字节对齐。

4.7 本章小结

本章对堆中存在的安全机制进行了探讨。这包括 ASLR、NX 位、RELRO 等通用的内存安全机制，也包括 glibc 中引入的诸多安全检查。这些安全检查可以被分为 5 类：阻止双重释放、确保参数合理、防止篡改大小、加固双向链表和加固单向链表。它们是对前文所述的五种堆攻击方式的应对措施。本文按类别逐一介绍了这些安全检查的工作方式，值得特别注意的是 Tcache 中防止双重释放的 key 机制和加固单向链表的 Safe-Linking 技术。

然而，上述堆利用技巧和堆安全检查非常依赖于堆管理器的版本。许多安全检查在某些 glibc 版本中存在，在其他版本中不存在。此外，不同版本的 glibc 对堆的实现也有所不同。这导致堆利用技巧总是只能针对特定版本开发。如果堆安全研究者想要对堆利用攻击进行研究，堆管理器的版本将成为一个令人头痛的问题。这些问题将在下一章中得到解决。

5　堆利用测试套件

堆利用几乎总是针对特定版本堆管理器的堆利用。它巧妙安排的操作顺序取决于堆管理器的执行路径，精心构造的地址偏移依赖于堆管理器的数据结构。它试图绕过的安全机制在某些版本的堆管理器中空缺，在另一些版本中又被加强。堆管理器版本成为攻击者实施堆利用无法绕开的问题，也是安全研究人员必须考虑的因素。

为了解决这个问题，本研究开发了一套*面向跨版本堆管理器的堆利用测试套件*。该套件包含三个组件：libccompile、libcbench 和 cheapsec。其中，libccompile 用于编译各个版本的 glibc，libcbench 用于在各个版本的 glibc 中对二进制文件进行测试，而 cheapsec 则用于检查 glibc 的堆安全属性。它们之间的关系如图 5.1 所示。

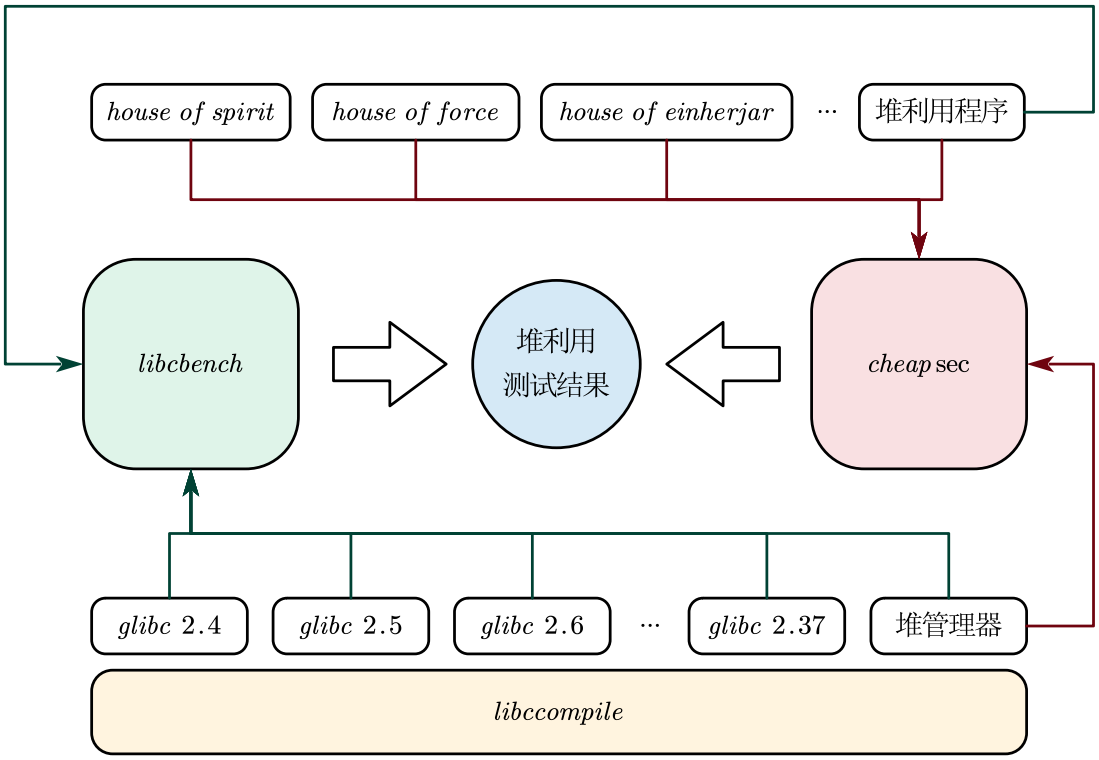

图 5.1　面向跨版本堆管理器的堆利用测试套件

libccompile 是进行跨版本堆管理器测试的基础。它能够简单地编译出研究者需要的堆管理器版本，无需过多关注 libc 与操作系统和编译器的深度耦合问题。

在构建堆利用攻击程序时，cheapsec 能提供极大的帮助。它允许研究者快速调查当前 glibc 的安全属性，包括其中包含的安全检查以及可实施的堆利用技巧。

如果研究者开发出了新的堆利用攻击，或者希望验证已存在的攻击，libcbench 让研究者可以无需修改源代码，直接在各版本的堆管理器上测试堆利用程序，从而快速评估堆利用攻击的影响范围。

使用该堆利用测试套件，本研究测试了 25 个堆利用技巧在 2.15 - 2.37 版本的 glibc 中的可行性，以及 42 个堆安全检查在 2.15 - 2.37 版本的 glibc 中的存在性。具体测试结果见附录 B。这些测试结果为堆安全研究提供了极其重要的参考。

目前，面向跨版本堆管理器的堆利用测试套件已经在 Gitee 平台上开源：

- libccompile： https://gitee.com/pvz122/libccompile
- libcbench： https://gitee.com/pvz122/libcbench
- cheapsec： https://gitee.com/pvz122/cheapsec

5.1 libccompile

5.1.1 开发目的

glibc 作为操作系统的核心组件，和所有其他底层软件一样与操作系统深度耦合，难以编译和安装。从源代码编译 glibc 需要复杂的步骤和配置[31]，还必须解决众多依赖问题。更糟糕的是，由于 glibc 本身就是编译 glibc 所需的依赖项，因此在现代操作系统中，高版本的 glibc 无法用于编译低版本的 glibc。如果研究者需要低版本 glibc 的二进制文件，他们可能需要花费数小时安装虚拟机，寻找可以编译目标版本 glibc 的特定操作系统，并在编译过程中解决各种错误。实际上，这正是本文作者在进行本研究时所经历的挑战。

一个可能的质疑是：难道 GNU/Linux 发行版的软件包仓库没有提供各版本 glibc 的二进制包吗? 事实上，的确有一些依赖于各版本 glibc 的安全工具，如 Libc-Searcher[32] 和 glibc-all-in-one[33]，它们通过直接拉取发行版软件仓库中的 glibc 二进制包来获取各个版本的 glibc。然而，发行版的 glibc 软件包存在两个问题：

- 软件包中不仅包含 glibc 的动态链接库，还包含许多发行版特定的文件，不适用于对 GNU/Linux 上堆利用的普遍研究。
- 发行版的软件仓库是根据该发行版自己的策略进行更新的，因此并未包含所有版本的 glibc，也不会包含较为古老的版本。

基于以上原因，本研究开发了 libccompile 作为跨版本堆利用测试的基础组件。它只需要简单的一个命令，就可以编译出从 2.4 版本（2006 年发布）到至今为止最新的 2.37 版本（2023 年发布）的 glibc。之所以无法编译 2.4 版本以前的 glibc，是因为在此之前 glibc 没有支持 X86_64 架构。

此外，libccompile 还支持编译研究者自己修改后的 glibc 源码。在堆安全研究中，经常需要对 glibc 源码进行个性化的修改以测试安全特性。使用 libccompile，可以自动推断 glibc 源码所基于的官方版本，并尝试在相应的环境下进行编译。

5.1.2 技术原理

让使用者无需关心操作系统环境和软件依赖的方法是容器化。容器化可以将应用程序与操作系统解耦，但却不会像虚拟机一样以损失性能为代价。这正是构建 glibc 编译工具的绝佳技术。

libccompile 基于 Docker，它将不同版本的 glibc 源代码分配到预先配置好的容器中进行编译。通过完全还原当初 glibc 开发时的环境，编译过程不会产生任何错误。而且，这些容器可以分发到任何运行 Docker 的操作系统中。

5.1.3 程序设计

(1) Docker 镜像

libccompile 是使用 Python、Perl 和 Dockerfile 开发的。为了编译各种版本的 glibc，它创建了 6 个 Docker 镜像，用于构建这些镜像的 Dockerfile 文件可以在 GitHub 仓库的 pvz122/libccompile/docker 目录中找到。表 5.1 描述了这些镜像的基本信息。

表 5.1 libccompile 创建的 Docker 镜像

镜像名称	编译的 glibc 版本范围	基于的发行版	gcc 版本	ld 版本
libccompile:04-10	2.4 - 2.10	Debian 4.0 Etch	4.1.2	2.17
libccompile:11-15	2.11 - 2.15	Debian 6.0.8 Squeeze	4.4.5	2.20.1
libccompile:16-22	2.16 - 2.22	Debian 7.3 Wheezy	4.7.2	2.22
libccompile:23-29	2.23 - 2.29	Debian 8.0 Jessie	5.3.0	2.25
libccompile:30-34	2.30 - 2.34	Debian 10.0 Buster	9.1.0	2.31.1
libccompile:35-37	2.35 - 2.37	Debian 11.0 Bullseye	12.2.0	2.35.2

① **满足先决条件**

编译 glibc 有两个先决条件：适当的系统环境和完整的依赖软件。表格中所列出的操作系统、编译器和链接器版本是编译 glibc 所需的系统环境。这些发行版和编译器版本是通过以下方式确定的：首先，glibc 的源代码包含一个 INSTALL 文件，其中记录了在开发该版本的 glibc 时使用的编译器和链接器版本。此外，glibc 官网的发行日志中包含了各版本 glibc 的发布时间。按照这个时间，可以选择相应的 GNU/Linux 发行版。随后，需要尝试组合这些条件，以找到能够完美编译相关 glibc 版本的系统环境。

编译 glibc 所需的依赖软件因版本而异，这些信息也在 INSTALL 文件中有说明。只需要在 Dockerfile 中使用发行版的包管理器安装相关依赖即可。需要注意的是，这些镜像中的发行版大多已经停止支持，因此必须修改它们的包管理器配置文件，以正确地下载所需的依赖包。

② **获取源码并编译**

解决了系统环境和依赖软件问题后，glibc 满足了所有必要的编译前置条件。接下来需要获取源码并进行编译。令人欣慰的是，glibc 源码使用 git 进行版本管理。只需要一份 glibc 源码，就可以使用 git checkout 快速切换到任意版本的 glibc 源码中。需要指出的是，在 Debian 4.0 Etch 发行的年代，git 刚刚诞生，包管理器中并没有包含 git 软件包。因此，在 libccompile:04-10 镜像的 Dockerfile 中，还需要额外添加 git 的编译和安装过程。

glibc 的编译也需要经过复杂的流程。因此，这些 Docker 镜像中还添加了自动配置和编译的脚本。这些编译和配置脚本是使用 Python 编写的，并可在 GitHub 仓库的 pvz122/libccompile/docker/compile.py 路径下找到。然而，在 libccompile:04-10 镜像中，由于当时 Python 3 尚未诞生，该脚本改用 Perl 编写，可以在 GitHub 仓库的 pvz122/libccompile/docker/compile.pl 路径下找到。这个脚本会执行以下几个步骤：

1. 从命令行读取需要编译的 glibc 版本。
2. 调用 git checkout 命令将源码切换到目标版本。
3. 创建 build 临时目录。
4. 运行 glibc 源码中的 configure 脚本，为其指定安装路径等参数，完成编译前

配置。

5. 运行 make，使用多线程编译。
6. 运行 make install，将编译好的二进制文件安装到指定目录。
7. 将二进制文件使用 tar 打包。

需要指出的是，为了提高鲁棒性，并在出现错误时帮助用户进行调试，该脚本记录了每个步骤的日志，并具备错误恢复功能。每次运行该脚本时，编译过程中的状态都会被写入到 build_meta.json 文件中。这样，在下一次运行脚本时，无需从头开始编译，而是可以直接从 build_meta.json 记录的状态中恢复。

以上就是 Docker 镜像中包含的所有内容。以 libccompile:11-15 镜像为例，它的 Dockerfile 形式如下：

```
FROM debian:6.0.8

# install dependencies
RUN \
echo \
'deb http://archive.debian.org/debian-archive/debian/'\
' squeeze main'\
>/etc/apt/sources.list && \
apt-get update && \
apt-get install -y --force-yes\
binutils gcc g++ make gawk realpath git-core python3 && \
apt-get clean

# copy script and src
WORKDIR /glibc
COPY glibc-src.tar.gz /glibc
COPY compile.py /glibc
RUN \
cd /glibc && \
tar xf glibc-src.tar.gz && \
mv glibc-src src && \
rm -rf glibc-src.tar.gz && \
chmod +x compile.py

# entrypoint
ENTRYPOINT ["/glibc/compile.py"]
```

这些镜像已经被构建好，并上传到了 Docker 的官方仓库： https://hub.docker.com/repository/docker/pvz122/libccompile。libccompile 主程序会在运行时根据用户需要编译的 glibc 版本自动拉取相应的镜像。

(2) 主程序

libccompile 的主程序也是使用 Python 开发的，其源代码可在 GitHub 仓库的 pvz122/libccompile/src/libccompile.py 路径下找到。主程序的两个核心功能是调用 Docker 镜像编译指定版本的 glibc，以及编译用户自定义的 glibc 源代码。其它附带

功能，譬如检查依赖、清理编译文件、构建 Docker 镜像等就不再赘述。

① **编译指定版本的** glibc

前文构建了 6 个 Docker 镜像，它们分别能够编译特定范围内的 glibc 版本。主程序从命令行获取用户希望编译的 glibc 版本范围（假设为 2.20 - 2.30），然后需要调用这些 Docker 镜像进行编译。具体而言，2.20 - 2.22 版本应由 libccompile:16-22 镜像编译，2.23 - 2.29 版本应由 libccompile:23-29 镜像编译，2.30 版本应由 libccompile:30-34 镜像编译。

主程序会进入一个循环中，每次循环都调用某个镜像编译对应范围的 glibc。为了确定每次编译时使用的镜像，实现了一个迭代器函数 version_range_selector。每次调用该函数时，它会返回当前所需编译的 glibc 版本以及要使用的 Docker 镜像，直到编译完成用户想要的所有版本。

一旦确定了每次编译的 glibc 版本和使用的镜像，主程序就会使用相应的镜像创建容器，并将编译任务分配给容器。前文提到，编译脚本会将编译后的二进制文件打包成 tar 包。因此，在容器完成编译后，主程序只需使用 docker cp 命令取回 tar 包即可。在所有版本的 glibc 编译完成后，使用 tar 的优势就显现了出来——可以使用 tar –concatenate 命令简单地将它们合并在一起。最后，主程序调用 gzip 将包含所有版本的 glibc 的 tar 包压缩成 tar.gz 格式，用户就得到了最终结果。

② **编译自定义的** glibc **源码**

编译用户自定义的 glibc 源代码是通过 Docker 容器的挂载功能实现的。用户指定要编译的 glibc 源码路径，libccompile 会读取源码中的 version.h 文件来确定源码所基于的 glibc 版本。之后，程序使用相应的镜像启动容器，并使用 -v 参数将 glibc 源码挂载到容器中。容器内的编译脚本会协同工作，不再使用 git checkout 切换源码版本，而是直接编译并将其打包给用户。

5.1.4 运行测试

libccompile 运行在命令行，提供了六个命令（具体的使用方法可以参见 GitHub 仓库中的 README 文件）：

- libccompile all：编译 libccompile 支持的所有版本的 glibc。

- libccompile range：编译指定版本范围内的 glibc。
- libccompile current：编译用户自定义的 glibc 源码。
- libccompile build：手动构建 Docker 镜像。
- libccompile clean：清理 Docker 容器和镜像。
- libccompile help：显示帮助信息。

以编译 2.20 - 2.30 版本的 glibc 为例，运行 libccompile range 2.20-2.30，libccompile 将会拉取镜像、执行编译（见图 5.2）。

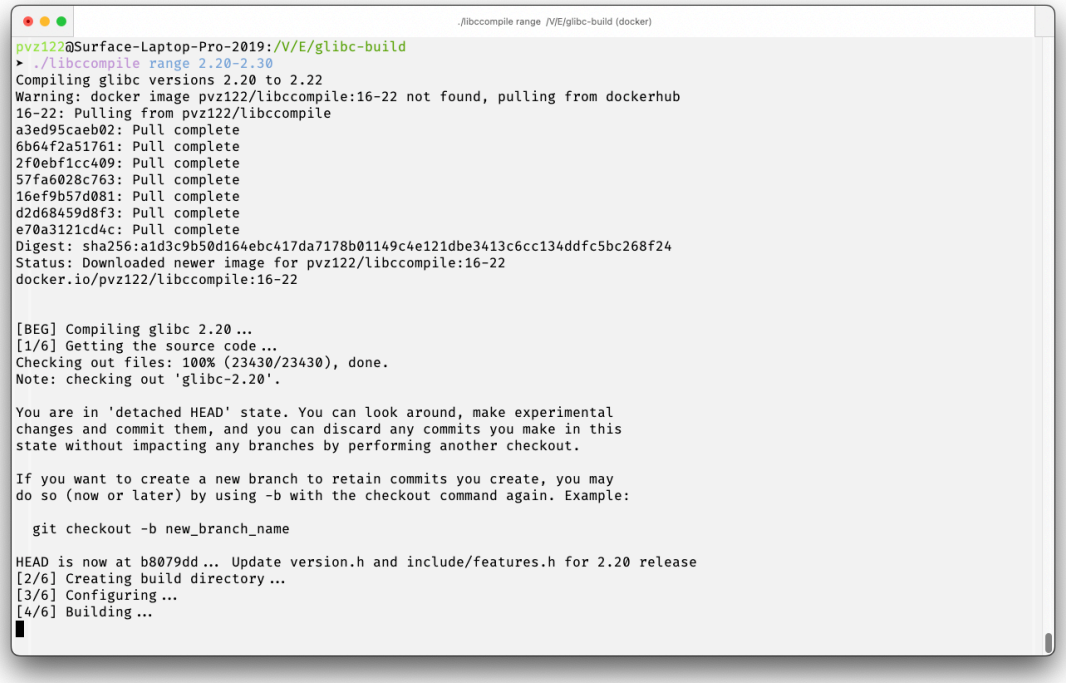

图 5.2　使用 libccompile 编译 glibc 2.20 - 2.30

等待编译结束后，会得到打包好的 glibc_2.20-2.30.tar.gz 文件，将其解压即可得到编译好的 glibc 动态链接库（见图 5.3）。

这些文件将可以直接被 libcbench 使用，也可以用于其它研究用途。

5.2　libcbench

5.2.1　开发目的

堆安全研究人员经常有在某个特定版本的堆管理器上测试堆利用程序的需求。有时候，他还需要了解堆利用攻击在不同版本堆管理器上的可行性，以此来确定

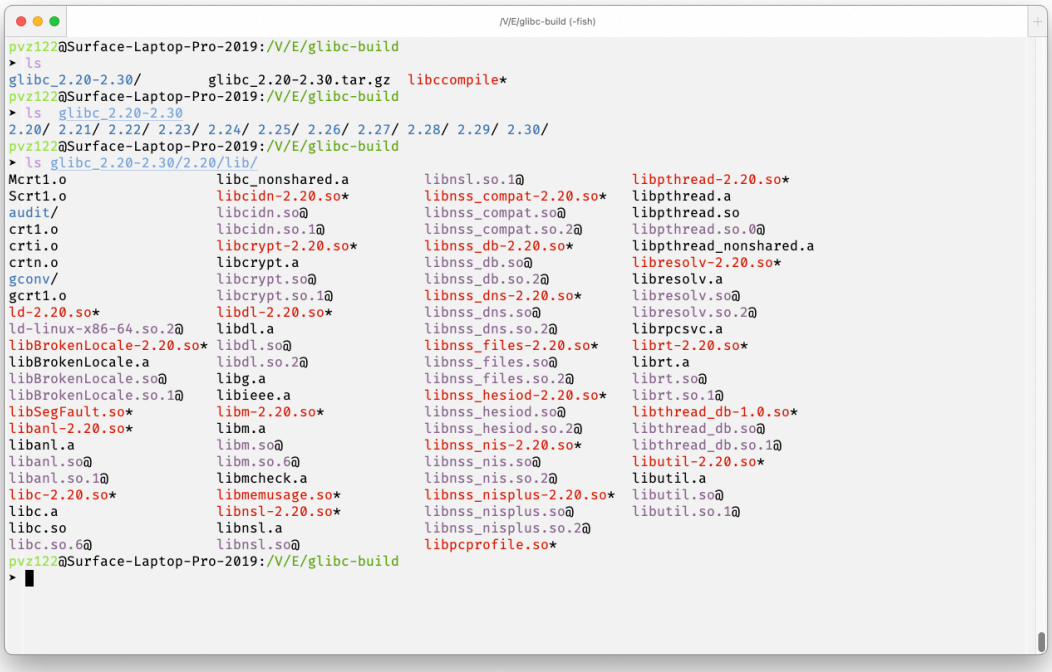

图 5.3　编译好的 glibc 2.20 - 2.30

攻击的影响范围。但是，和编译各版本的 glibc 一样，在各个版本的 glibc 上运行程序也不是一件容易的事情，这是由于以下两个原因：

- glibc 是操作系统的核心组件，包装了 Linux 内核提供的系统调用，几乎所有的应用程序都依赖它运行。在 GNU/Linux 上安装非系统自带版本的 glibc 会破坏这些应用程序的正常运行。
- glibc 与普通的动态链接库不同，它还包含了 ld 动态链接器，会参与到 ElF 文件在内存中装载的过程。而 ld 在 ELF 文件中是硬编码的，由编译器生成。更换程序的 glibc 并不容易。

libcbench 是为了解决这一问题而开发的。它包含两个 CLI 工具：libcset 和 libcbench。libcset 可以将可执行文件设置为在指定版本的 glibc 上运行；而 libcbench 则可以使用 libccompile 编译出的所有 glibc 版本测试可执行文件，并提供测试结果。

5.2.2　技术原理

由于上述两个原因，要想切换可执行文件的 glibc 版本，只有两种方法：从源码重新编译，在编译时指定 glibc 路径；或修改 ELF 文件中硬编码的特定结构，使

其指向希望使用的 glibc。

由 NixOS 开发的 patchelf[34] 提供了修改 ELF 文件的方法。简而言之，它通过修改 ELF 文件的.dynamic 节更换动态链接器，然后修改其中的 RPATH 以指定可执行文件搜索动态库的目录。将这两个值修改为目标版本 glibc 的 ld.so 和 libc.so 所在路径即可在指定的 glibc 版本上运行可执行文件。

libcbench 基于 patchelf 的方法开发了自动化测试脚本，将测试结果统计并展示给使用者。

5.2.3 程序设计

libcbench 是使用 Shell 脚本语言编写的，它包含两个脚本：libcset 和 libcbench。

(1) libcset

libcset 用于设置可执行文件链接的 glibc 版本，其源代码可以在 GitHub 仓库的 pvz122/libcbench/libcset 路径下找到。它接收两个命令行参数，第一个是目标可执行文件，第二个是 glibc 版本号。然后，它会在 /opt/libcbench/glibc（libcbench 的安装路径）中搜索相应的 glibc 版本，找到后执行以下命令来修改可执行文件的动态链接器和 RPATH：

```
patchelf --set-interpreter \
/opt/libcbench/glibc/${GLIBC_VERSION}/lib/ld-linux-x86-64.so.2\
--set-rpath \
/opt/libcbench/glibc/${GLIBC_VERSION}/lib "${EXECUTABLE_PATH}"
```

(2) libcbench

libcbench 用于在所有版本的 glibc 中测试可执行文件，其源代码可以在 GitHub 仓库的 pvz122/libcbench/libcbench 路径下找到。它的核心功能是测试可执行文件、记录运行输出、统计运行结果。

libcbench 会遍历可用的 glibc，对每一个 glibc 版本调用 libcset 修改二进制文件，然后运行它：

```
# for each version, run `libcset executable_path version`
# and then run the executable
for version in $avai_vers; do
  libcset "${exe_path}" ${version}
  # ...
  timeout ${timeout_val} "${exe_path}"
```

值得注意的是，在 glibc 2.26 版本之前，如果程序触发了安全检查或 glibc 出现错误，它会在显示错误信息后卡住，直到用户手动结束它。这在自动化测试中是一个问题，因此 libcbench 会使用 timeout 命令为程序运行设置超时时间。如果程序卡住，libcbench 会认为运行失败并直接结束它，以便继续进行测试。

如果记录功能被开启，程序运行时标准输出流和标准错误流的输出还会被保存到 log 文件中：

```
timeout ${timeout_val} "${exe_path}" 2>&1 | tee -a ${log_file}
```

该命令的含义是将标准错误流重定向到标准输出流（2>&1），然后通过管道同时输出到屏幕上和 log 文件中（| tee）。

统计功能是通过程序返回值实现的。libcbench 会将程序返回值记录在数组中，并判断返回值代表的状态。0 代表运行成功，124 代表超时，其他值则代表运行失败。

5.2.4 运行测试

将前文中使用 libccompile 编译得到的 glibc.tar.gz 文件放置于 libcbench 源代码目录中，运行 install.sh 即可将 libcbench 和打包的 glibc 安装至系统中。libcbench 提供了四个参数：

- –log：记录测试时的程序输出。
- –timeout：自定义程序超时时间。
- -l：列出系统中安装的所有 glibc 版本。
- -h：显示帮助信息。

使用以下 print_glibc_version 程序测试 libcbench：

```c
#include <stdio.h>
#include <string.h>
#include <gnu/libc-version.h>

int main()
{
    printf("The version of libc is: %s\n",\
    gnu_get_libc_version());
    return strcmp(gnu_get_libc_version(), "2.22");
}
```

这个程序的功能非常简单，它会在屏幕上打印 glibc 的版本。如果 glibc 版本是 2.22 就返回 0，不是就返回 1. 编译，使用 libcbench 测试的结果如图 5.4 所示。

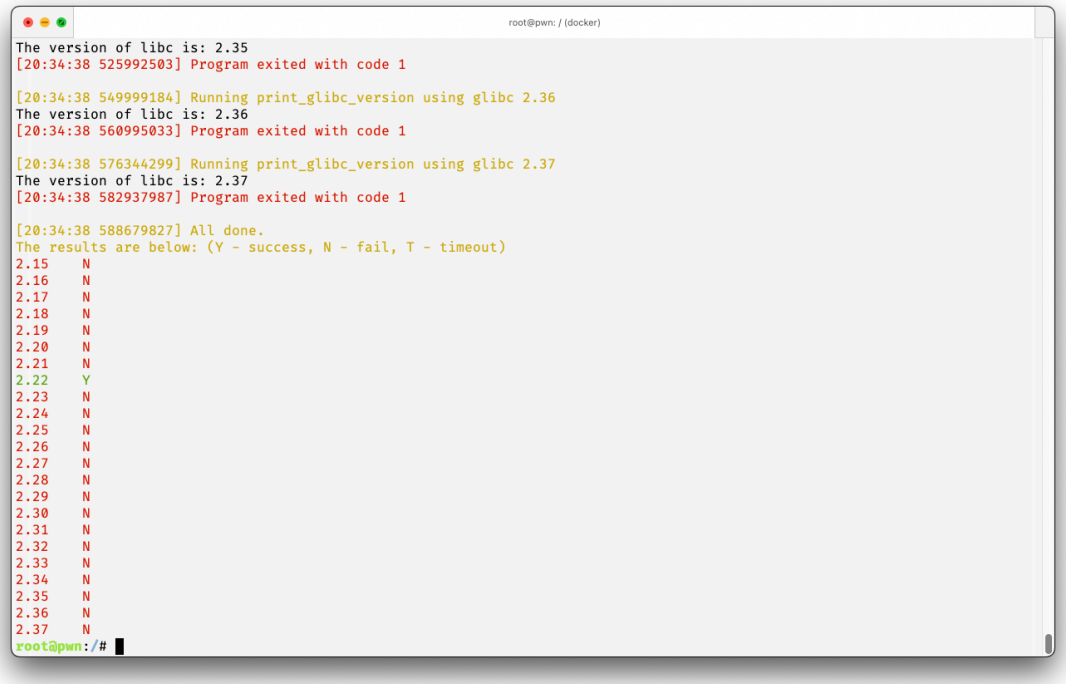

图 5.4　使用 libcbench 测试 print_glibc_version 程序

5.3 cheapsec

5.3.1 开发目的

面对一个二进制文件，堆安全研究人员首先需要弄清的是它使用的堆管理器所包含的安全属性。只有了解目标堆管理器中存在的安全检查以及现有堆利用技巧在堆管理器上的可行性，研究者才能尝试对其发起堆利用攻击。

cheapsec 就是为了解决这个问题而开发的。cheapsec 是 Check Heap Security 的缩写，它与著名的 checksec 工具[35] 非常相似。checksec 用于检查 GNU/Linux 中二进制文件的安全属性，如 PIE、RELRO、ASLR 等安全机制是否开启。而本研究开发的 cheapsec 则用于检查二进制文件的堆安全属性。检查内容包括 42 个堆安全检查是否存在，以及 25 个堆利用技巧是否能够施行。

5.3.2 技术原理

cheapsec 所使用的技术方法是在目标堆管理器上实际测试相关堆利用技巧和堆安全检查。为此，本研究编写了 25 个堆利用技巧的 POC 代码以及 42 个堆安全检查的触发程序，其中部分 POC 改编自 how2heap 开源项目[19]。

cheapsec 识别出二进制文件所链接的 glibc 路径，将以上触发程序的源代码使用目标 glibc 编译和链接，然后运行得到测试结果。这些触发程序中的适当位置加入了断言函数，如果测试不可行就会返回失败的返回值。cheapsec 最后将这些返回值进行统计和展示。

5.3.3 程序设计

cheapsec 工具是用 C 和 Python 开发的，它主要包含两个部分：堆利用和堆检查的触发程序，以及 cheapsec 主程序。

(1) 堆利用和堆检查的触发程序

cheapsec 包含了 25 个堆利用 POC 程序和 42 个触发堆检查的程序。这些程序的源代码可以在 GitHub 仓库的 pvz122/cheapsec/testing 目录下找到。其中有一些堆利用 POC 程序有多个版本，以适应不同的 glibc 版本。

堆利用 POC 程序的原理是实施完堆利用后，使用断言函数（assert）对目标效果进行检测。如果没有达到攻击效果，或者利用实施过程中触发了 glibc 安全检查，

程序就会异常退出，返回值将为非 0. 反之，如果利用成功，程序就会返回 0. 以简单的 fastbin dup 利用为例，在 glibc 2.26 版本及以下的 POC 代码如下：

```
#include <stdlib.h>
#include <assert.h>
int main()
{
  int *a = malloc(8);
  int *b = malloc(8);
  int *c = malloc(8);
  free(a);
  free(b);
  free(a);

  a = malloc(8);
  b = malloc(8);
  c = malloc(8);
  assert(a == c);
  return 0;
}
```

程序逻辑非常清晰，在双重释放 a 后，如果两次分配得到的是同一个 Chunk（assert(a == c)），就说明利用成功了。

堆安全检查触发也类似。以简单的 “free(): invalid size” 触发程序为例：

```
void *p = malloc(0x100);
chunk_p c = (chunk_p)((char *)p - 0x10);
c->size = 0x8 | 0x1; // c->size < MINSIZE
free(p); // free(): invalid size
return 0;
```

它将一个 Chunk 的 size 字段改为一个不可能的极小值，对它调用 free，从而触发 “free(): invalid size” 检查。如果检查被触发了，程序就会异常退出。反之，程序会正常返回 0.

(2) 主程序

cheapsec 的主程序用于测试 glibc 的安全属性，其源代码可以在 GitHub 仓库的 pvz122/cheapsec/src/cheapsec.py 路径下找到。cheapsec 主程序大致上就是通过前述触发程序的返回值来确定利用是否可行或者检查是否存在的。当然，每次都编译并运行几十个 POC 是耗时的。在默认情况下，cheapsec 以查表模式工作，它会推断出目标 glibc 的版本并在测试结果数据库中查找相应的安全属性。只有在研究者自行修改了 glibc，或者面对无法判断版本的 glibc 时，才需要手动指定 cheapsec 工作在编译测试模式下。

① 编译测试模式

工作在编译测试模式的 cheapsec 会在指定的 glibc 上编译并运行触发程序。使用指定的 glibc 编译程序的方法如下：

```
# compile using specified glibc
def compile_with_glibc(src_path: str, glibc_path: str, \
flags: str = "") -> bool:
    ret_code = subprocess.run("gcc \"{}\" -Wl,--rpath={}"
    " -Wl,--dynamic-linker={}/ld-linux-x86-64.so.2"
    " -o cheapsec_test {}".format(
        src_path, glibc_path, glibc_path, flags
    ), shell=True, stdout=subprocess.DEVNULL,\
    stderr=subprocess.DEVNULL).returncode
    return ret_code == 0
```

实际上，这种方法也是通过设置可执行文件的动态链接器和 RPATH 来实现的，只是与 patchelf 修改可执行文件相比，改为了在编译时设置。这能够让程序的兼容性更好。

而后，cheapsec 就会运行编译好的可执行文件，并通过返回值和错误信息判断利用是否成功或者检查是否触发。值得指出的是，在 glibc 2.26 版本以前，glibc 打印错误信息使用的 __libc_message 函数会直接往 /dev/tty 终端上写错误信息，而不是使用程序的标准错误流。这使得 cheapsec 无法从错误信息中获知是否触发了安全检查，而且还会对自动化测试的信息输出造成干扰。因此，在测试小于等于 2.26 版本的 glibc 安全属性时，cheapsec 会使用创建伪终端的方式运行触发程序，这是

使用 pty 库实现的。

将所有的触发程序编译测试完成后，cheapsec 会将结果统计并输出到终端。

② **查表模式**

在查表模式下，cheapsec 基于预先完成的测试数据（详见附录 B），这些数据存储在 data 目录的 JSON 文件中。有了这些数据，检查已知 glibc 的安全属性就变得非常方便。只需要推断出目标 glibc 的版本，然后加载 JSON 数据文件，进行查表即可。

推断 glibc 版本的方法是使用待检查的 glibc 编译程序，在程序中调用 glibc 的私有函数 gnu_get_libc_version 获得 glibc 声明的版本号。

5.3.4 运行测试

cheapsec 提供 6 个选项（具体的使用方法可以参见 GitHub 仓库中的 README 文件）：

- 默认：以查表的方式检查目标二进制文件或 glibc 库的堆安全属性。
- -l：查找指定版本 glibc 的安全属性。
- -t：以编译测试的方式检查目标二进制文件或 glibc 库的堆安全属性。
- -te：以编译测试的方式检查目标二进制文件或 glibc 库的堆可利用性。
- -ts：以编译测试的方式检查目标二进制文件或 glibc 库中存在的堆安全检查。
- -h：显示帮助信息。

在 Ubuntu 22.04（glibc 版本 2.36）中，让 cheapsec 以编译测试的方式检查可执行文件的安全属性，结果如图 5.5。

使用查表的方法检查 glibc 2.22 的安全属性，如图 5.6。

5.4 测试堆安全属性

本研究开发的面向跨版本堆管理器的堆利用测试套件，能够轻松地测试各版本 glibc 的安全属性，包括它们的可利用性和安全检查存在性。这些跨版本堆管理器的安全属性数据将为堆安全的研究提供极大的参考价值。因此，本研究设计实验在 glibc 2.15 - 2.37 版本中测试了 25 个堆利用技巧和 42 个堆安全检查。

glibc 2.15 是在 2012 年发布的。由于本研究选取的堆利用技巧都是在近十年的

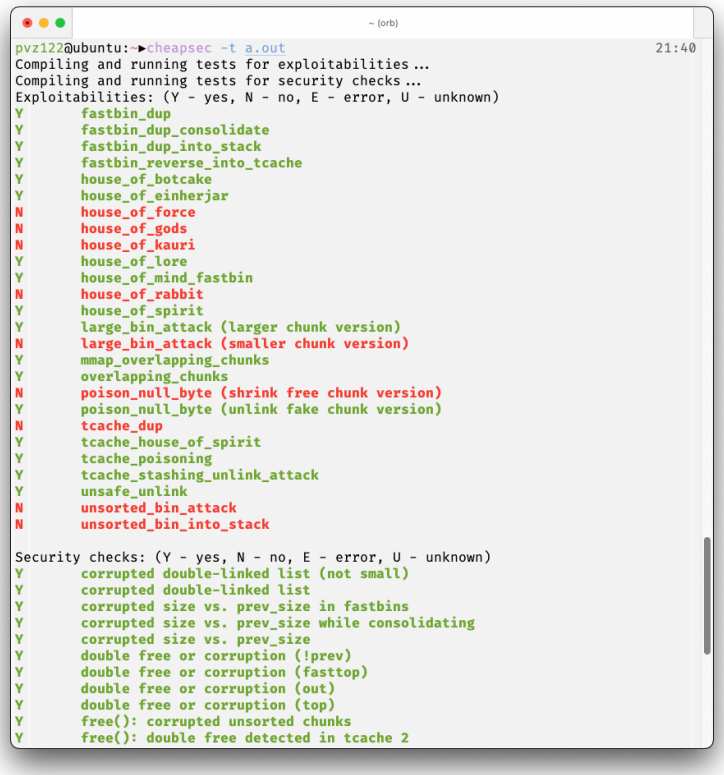

图 5.5　使用 cheapsec 测试可执行文件的堆安全属性

堆管理器中得到验证的，因此没有再测试更早的 glibc 版本。这些更早的版本由于其实现的不同难以被现代技巧利用。而且它们几乎不再有人使用，缺少研究价值。

测试的方法是，首先使用 libccompile 工具编译 glibc 2.15 - 2.37 的动态链接库文件，然后使用 cheapsec 分别测试每个 glibc 版本中可以实施的堆利用技巧和存在的安全检查。使用的 glibc 源码均来自 GNU C Library 官网，相关 POC 和触发代码部分来自 how2heap 开源项目[19]，另一部分由本研究编写。测试结果数据详见附录 B。

测试结果表明，最新版本的 glibc 2.37 中仍然有 17 种可用的堆利用技巧，而最早版本的 glibc 2.15 只有 16 种可用技巧。这在很大程度上要归因于 glibc 2.26 为了提高多线程性能引入的 Tcache 机制。在最初被引入时，该机制几乎没有任何防护措施。因此，在 glibc 2.26 中可用的堆利用技巧达到了 23 种之多。

在堆安全检查方面，glibc 2.29 是一次重大更新，其安全检查数量从 23 种增加到了 31 种。而 glibc 2.32 则也是一次重要的安全更新，其安全检查数量从 33 种增加到了 39 种。正是在该版本中，Safe-Linking 机制被引入。

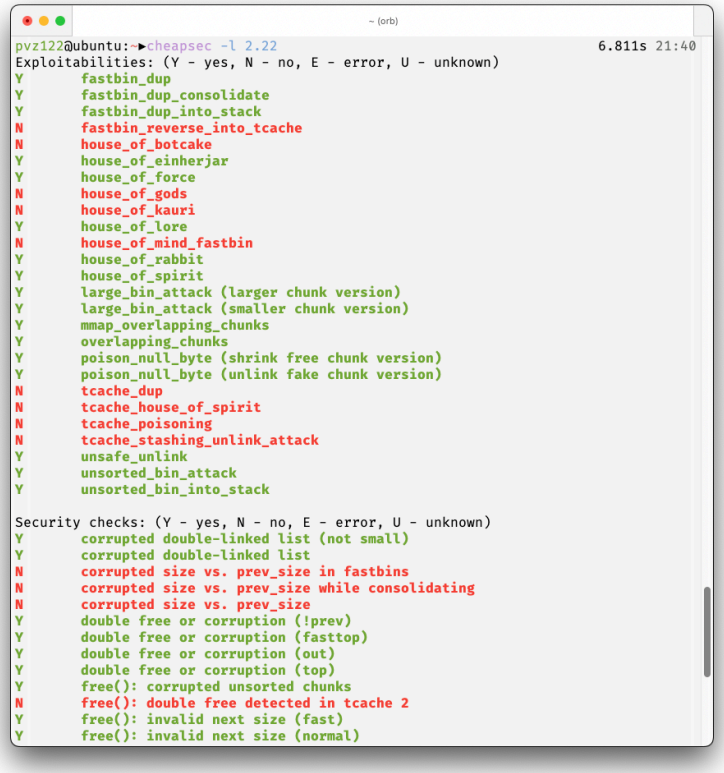

图 5.6　使用 cheapsec 查找 glibc 2.22 的堆安全属性

5.5　本章小结

在本章中，本研究介绍了自行研发的堆利用测试套件。这一套件旨在解决堆安全研究中对堆管理器版本信息的依赖。该套件由三个组件构成：libccompile、libcbench 和 cheapsec。

libccompile 是编译 glibc 的工具，解决了编译 glibc 的深度耦合和复杂依赖问题。libcbench 是使用各版本 glibc 测试可执行文件的工具，解决了切换可执行文件的 libc 库困难的问题。cheapsec 是检查堆管理器安全属性的工具，解决了研究者难以获知堆管理器可利用性的问题。

本研究使用这一套件在 23 个堆管理器版本上测试了 25 个堆利用技巧和 42 个堆安全检查，这些测试结果能为堆安全的研究提供极大的参考，详细数据见附录 B。

6　总结与讨论

本文的工作为现有的堆利用机理与方法勾勒出了一幅完整的图景，从中可以看到将来堆攻防的发展方向。这一章将总结本文所做的工作，并尝试讨论堆利用与防护的未来发展，以期为提高堆安全性提供有益的参考。

6.1　工作总结与展望

本文的工作主要包含理论研究和工具研发两部分。

在理论研究方面，本研究首先选取了 25 种广泛使用的堆利用技巧，详细考察了它们的机理，复现了它们的方法。这些利用技巧在实施过程中需要花费许多精力来绕过堆中的安全检查。因此，本研究进一步阅读了 glibc 的源代码，调查了最新版本堆管理器中存在的 42 个安全检查。这些调研工作揭示了堆的脆弱性、堆利用技巧和堆安全检查之间的对应关系。本文将其整理为 4 种漏洞、5 种攻击手法和 5 种缓解措施，如图6.1所示。本文在“堆利用机理与方法”和“堆安全机制”两章中，按照这一分类方法对调研工作进行了介绍。

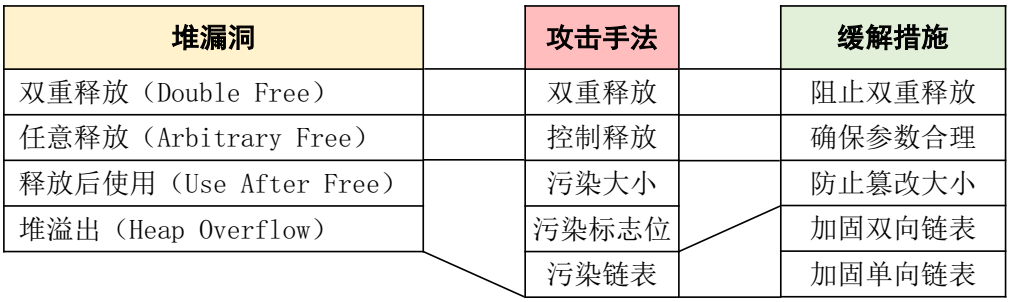

图 6.1　堆漏洞、攻击手法与缓解措施

接下来，本研究提出了堆利用的利用流模型，旨在为堆利用过程提供一个通用的抽象。该模型的核心是利用原语，也就是攻击者在实施攻击的过程中获得的基本能力。经过考察，本研究提出了 12 种堆利用原语，并将其分为初级、中级和高级三类。这些原语之间具有推导和蕴含关系。推导指的是实施堆利用技巧导致的原语转换，而蕴含则指较强原语包含较弱原语的情况。每一种堆利用技巧都有依赖的原语和产生的原语，即堆利用技巧需要的条件和导致的效果，本研究将它们归纳在了表 3.5 中。一次完整的堆利用攻击被称为堆利用流，它从漏洞出发，提

取出初始原语，然后使用各种利用技巧逐步转换为更高级的原语，直到达到攻击效果。

在工具研发方面，本研究着眼于跨版本堆管理器的堆利用测试难题，开发了一套面向跨版本堆管理器的堆利用测试套件。该套件包含三个部分：libccompile、libcbench 和 cheapsec。其中，libccompile 解决了编译 glibc 时的困难，libcbench 解决了在不同版本堆管理器上执行测试的问题，而 cheapsec 则解决了调查可执行文件的堆安全属性的问题。利用这三个组件，本研究设计了实验，在 23 个版本的堆管理器上测试了 25 个堆利用技巧和 42 个堆安全检查，测试数据在附录 B 中。这些数据对跨版本堆安全研究具有重要参考价值。

当然，本研究的工作仍然有许多可以改进的方面。例如，本文的研究对象为 X86_64 架构中的 ptmalloc2 堆管理器，这是综合考虑了研究难度和必要性的决定。实际上，本研究也可以扩充到其他架构和堆管理器中，譬如传统的 X86 架构和移动平台的 ARM 架构。如果可能，本研究开发的堆利用测试套件也可以扩展支持这些架构。另外，本研究提出的堆利用模型不局限于解释现有的堆利用方法，还可以在此基础上开发自动化原语挖掘工具。本文提出的初级原语直接对应了程序分析工具产生的结果，而后就可以尝试进行组合和推导，从而挖掘出更强的原语。这些都将是未来的工作。

6.2 堆利用与防护的未来发展

本文的研究能帮助了解堆攻防的现状。但攻击者不是一成不变的，堆利用和防护都将继续发展。

在“堆安全机制”一章中，可以看到 glibc 的安全措施主要是在特定位置引入元数据的有效性检查。这样的检查有两个问题。首先，由于 glibc 的实现中缺少校验元数据完整性的机制，它无法真正意义上地确定元数据是否被污染，而只能通过类似于 Chunk →fd →bk 是否等于 Chunk 本身这样的间接方式检查其有效性。这意味着攻击者只需要增加构造的精巧程度，最终总能绕过这些检查。其次，这些检查只在特定函数的特定位置进行，不是全面的、系统的，攻击者经常能找到某些不包含检查的特殊执行路径。如果 glibc 不进行大的设计改变，那么增加更多堆安全检查——发展更复杂堆利用技巧的循环将一直重复下去。

对攻击者而言，堆利用的空间是广阔的。这不仅是因为上文提到的 glibc 设计

问题，还因为堆中的数据动态、复杂，提供了许多尚未探索的攻击可能性。目前绝大多数的堆利用方法都集中在污染链表和大小上，如污染 M 比特这样另辟蹊径的方法甚至还未出现，但它显然是极具发展可能性的。2022 年新发明的 house of gods[26] 就是一种另辟蹊径的方法，它提出了一种从未想过的思路：将 Arena 当作 Fake Chunk 链上链表，从而获得 Arena 的控制权。今后，这样的堆利用创新必然将持续出现。

对堆管理器的开发者而言，堆安全威胁的态势是严峻的。大量出现的堆利用技巧迫使开发者不得不进一步增加堆安全检查，但增加的安全检查不久后就无济于事，而且使代码仓库变得碎片和混乱。作为一个被数亿台设备使用的堆管理器，glibc 并没有太多选择。因为没有人能忍受一个安全但缓慢的系统组件。可是，如果 glibc 不能进行实质性的创新，那么它很有可能像许多其它包袱沉重的基础软件一样被历史淘汰。如今已经出现了相当多的 glibc 替代品。

堆管理器需要更优秀的安全机制，2020 年提出的 Safe-Linking[30] 就是一个例子。它能够极大地提高安全性，却不会造成太大的性能损失。这些堆安全设计的创新也在不断产生。2010 年 Novark 等提出的 DieHarder 堆管理器[36] 通过将堆元数据和用户数据隔离等安全措施，减轻了堆溢出，还能阻止释放后使用。2017 年 Silvestro 等提出的 FreeGuard[37] 能够防止堆溢出、释放后使用、双重释放和任意释放，同时能达到和 glibc 类似的速度。但这些注重安全的堆管理器总是需要额外的内存来放置安全结构。2019 年 Beichen 等提出的 SlimGuard[38] 在保证安全的基础上尽量减少了内存占用。这些新的堆安全设计将成为堆攻防中的下一代技术。

总的来说，未来新的堆利用技巧将持续出现，堆安全威胁有增不减。而 glibc 的修补效果寥寥，亟需产生堆管理器设计上的安全进步。堆安全的研究者需要为提高堆安全性做出长期的努力。

参考文献

[1] EPAKSKAPE. Microsoft Research[EB/OL]. https://twitter.com/epakskape/status/984481101937651713, 2023.

[2] GOOGLE. Chrome OS exploit: one byte overflow and symlinks[EB/OL]. https://googleprojectzero.blogspot.com/2016/12/chrome-os-exploit-one-byte-overflow-and.html, 2023.

[3] 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(1): 1-15.

[4] SIMMONDS B. Protecting the heap[EB/OL]. https://www.bencode.net/papers/mine/2019-simmonds-exploitdev-heap-allocators.pdf, 2019.

[5] 0X434B. Overview of GLIBC heap exploitation techniques[EB/OL]. https://0x434b.dev/overview-of-glibc-heap-exploitation-techniques, 2022.

[6] RØRVIK M F. Investigation of x64 glibc heap exploitation techniques on linux[D]. 2019.

[7] REPEL D, KINDER J, CAVALLARO L. Modular synthesis of heap exploits[C]// Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security. 2017: 25-35.

[8] YUN I, KAPIL D, KIM T. Automatic techniques to systematically discover new heap exploitation primitives[C]//Proceedings of the 29th USENIX Conference on Security Symposium. 2020: 1111-1128.

[9] ECKERT M, BIANCHI A, WANG R, et al. Heaphopper: Bringing bounded model checking to heap implementation security[C]//27th USENIX Security Symposium (USENIX Security 18). 2018: 99-116.

[10] WU W, CHEN Y, XU J, et al. Fuze: Towards facilitating exploit generation for kernel use-after-free vulnerabilities[C]//27th USENIX Security Symposium (USENIX Security 18). 2018: 781-797.

[11] YAN W, CHAO Z, XIANG X, et al. Revery: From proof-of-concept to exploitable [C]//the 2018 ACM SIGSAC Conference. 2018.

[12] WANG Y, ZHANG C, ZHAO Z, et al. Maze: Towards automated heap feng shui. [C]//USENIX Security Symposium. 2021: 1647-1664.

[13] STALLMAN R. Linux and the gnu system[EB/OL]. https://www.gnu.org/gnu/linux-and-gnu.en.html, 2021[2021-11-02].

[14] GOOGLE. tcmalloc[EB/OL]. https://google.github.io/tcmalloc/overview.html, 2023.

[15] FREEBSD. jemalloc[EB/OL]. https://jemalloc.net/, 2023.

[16] BOVET D P, CESATI M. Understanding the linux kernel: from i/o ports to process management[M]. " O'Reilly Media, Inc.", 2005.

[17] GLOGER W. Wolfram Gloger's malloc homepage[EB/OL]. http://www.malloc.de/en, 2006.

[18] HACKLIZA. Heap analysis with radare2[EB/OL]. https://hackliza.gal/en/posts/r2heap, 2022.

[19] SHELLPHISH. how2heap[EB/OL]. https://github.com/shellphish/how2heap, 2023.

[20] AWARAU. House of Kauri[EB/OL]. https://awaraucom.wordpress.com/2020/06/20/house-of-kauri, 2020.

[21] PHANTASMAGORIA P. The malloc maleficarum[EB/OL]. 2005.

[22] GOICHON F. Glibc adventures: The forgotten chunks[C]//Proc. Context Inf. Secur.: volume 28. 2015: 1-35.

[23] DULIN M. how2heap-mmap_overlapping_chunks[EB/OL]. https://github.com/shellphish/how2heap/blob/master/glibc_2.35/mmap_overlapping_chunks.c, 2023.

[24] KAPIL D. Dhavalkapil/heap-exploitation[EB/OL]. https://doi.org/10.5281/zenodo.6450612, 2022.

[25] SHIFT_CROPS. House of Rabbit - Heap exploitation technique bypassing ASLR - [en][EB/OL]. https://shift-crops.hatenablog.com/entry/2017/09/17/213235, 2017.

[26] MILO-D. house-of-gods[EB/OL]. https://github.com/Milo-D/house-of-gods/blob/master/rev2/HOUSE_OF_GODS.TXT, 2023.

[27] ST4G3R. House-of-Einherjar-CB2016[EB/OL]. https://github.com/st4g3r/House-of-Einherjar-CB2016, 2016.

[28] DULIN M. House of Mind - Fastbin Variant Revived[EB/OL]. https://maxwelldulin.com/BlogPost?post=2257705984, 2023.

[29] GOOGLE. A survey of recent iOS kernel exploits[EB/OL]. https://googleprojectzero.blogspot.com/2020/06/a-survey-of-recent-ios-kernel-exploits.html, 2020.

[30] ITKIN E. Safe-Linking - Eliminating a 20 year-old malloc() exploit primitive - Check Point Research[EB/OL]. https://research.checkpoint.com/2020/safe-linking-eliminating-a-20-year-old-malloc-exploit-primitive, 2020.

[31] GNU. Configuring and compiling (The GNU C Library)[EB/OL]. https://www.gnu.org/software/libc/manual/html_node/Configuring-and-compiling.html, 2023.

[32] LIEANU. LibcSearcher[EB/OL]. https://github.com/lieanu/LibcSearcher, 2023.

[33] MATRIX1001. glibc all in one[EB/OL]. https://github.com/matrix1001/glibc-all-in-one, 2023.

[34] NIXOS. patchelf[EB/OL]. https://github.com/NixOS/patchelf, 2023.

[35] KLEIN T. checksec.sh[EB/OL]. https://www.trapkit.de/tools/checksec, 2020.

[36] NOVARK G, BERGER E D. Dieharder: securing the heap[C]//Proceedings of the 17th ACM conference on Computer and communications security. 2010: 573-584.

[37] SILVESTRO S, LIU H, CROSSER C, et al. Freeguard: A faster secure heap allocator[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017: 2389-2403.

[38] LIU B, OLIVIER P, RAVINDRAN B. Slimguard: A secure and memory-efficient heap allocator[C]//Proceedings of the 20th International Middleware Conference. 2019: 1-13.

致 谢

本研究的完成离不开许多人的帮助。首先，我要衷心感谢我的导师苏璞睿老师和贾相堃老师，他们的卓越指导让我在研究的道路上更加笃定，在遇到困难时给予了我很多有益的建议和启示。如果没有他们百忙之中为我提供的无私帮助，我不可能完成这项工作。同时，我也要特别感谢我的校内导师严飞老师，他的耐心指导和细致解答使我受益匪浅。我还要感谢四年之中所有教导过我的老师们，我将永远铭记他们的教诲和关怀，感激他们对我的深厚影响。

其次，我要由衷地感谢我的父母。他们一直以来都是我的坚强后盾和支持者，在我人生的每一个阶段都给予了我最无私的关心和帮助。如果说我在学业上取得了任何成绩，那与他们一直以来的支持和鞭策密不可分。他们从不计较付出和牺牲，默默地为我做着种种准备和支持。因此，我要向我的父母表达最真挚的感谢和敬意。

寥寥数语，难以表示我的感激之情。最后，我要向抽出宝贵时间对本文进行评审并提出意见的专家们表达我的真诚谢意！

附录 A　42 个堆安全检查

这一节枚列了 glibc 2.37 版本中存在的 42 个堆安全检查，它们按在 glibc 中出现的位置排序，以便于快速查找。表A.1提供了这些安全检查的总览，并描述了它们的检查目的。

(1) corrupted size vs. prev_size

```
1  if (chunksize (p) != prev_size (next_chunk (p)))
2  malloc_printerr ("corrupted size vs. prev_size");
```

p 是被 unlink 的 Chunk，检查 p 的 size 字段是否等于物理相邻的下一个 Chunk 的 prev_size 字段。

(2) corrupted double-linked list

```
1  mchunkptr fd = p->fd;
2  mchunkptr bk = p->bk;
3
4  if (__builtin_expect (fd->bk != p || bk->fd != p, 0))
5  malloc_printerr ("corrupted double-linked list");
```

p 是被 unlink 的 Chunk，检查 p→fd→bk 是否等于 p，p→bk→fd 是否等于 p。

(3) corrupted double-linked list (not small)

```
1  if (!in_smallbin_range (chunksize_nomask (p)) && p->fd_nextsize\
2  != NULL)
3  {
4    if (p->fd_nextsize->bk_nextsize != p || \
5    p->bk_nextsize->fd_nextsize != p)
6  malloc_printerr ("corrupted double-linked list (not small)");
```

p 是被 unlink 的 Chunk，如果 p 在 Large Bin 中，检查 p→fd_nextsize→bk_nextsize 是否等于 p，p→bk_nextsize→fd_nextsize 是否等于 p。

(4) munmap_chunk(): invalid pointer

```
uintptr_t mem = (uintptr_t) chunk2mem (p);
uintptr_t block = (uintptr_t) p - prev_size (p);
size_t total_size = prev_size (p) + size;
if (__glibc_unlikely ((block | total_size) & (pagesize - 1))\
!= 0
  || __glibc_unlikely (!powerof2 (mem & (pagesize - 1))))
malloc_printerr ("munmap_chunk(): invalid pointer");
```

p 是被 munamp 的 Chunk，检查 block、total size、mem 是否与页面对齐。

(5) malloc(): unaligned tcache chunk detected

```
/* Caller must ensure that we know tc_idx is valid
and there's available chunks to remove.  */
static __always_inline void *
tcache_get (size_t tc_idx)
{
  tcache_entry *e = tcache->entries[tc_idx];
  if (__glibc_unlikely (!aligned_OK (e)))
  malloc_printerr ("malloc(): unaligned tcache chunk detected");
  tcache->entries[tc_idx] = REVEAL_PTR (e->next);
  --(tcache->counts[tc_idx]);
  e->key = 0;
  return (void *) e;
}
```

Safe-Linking 检查的一部分。在取出 Tcache Chunk 时检查 Tcache 头部的 Chunk 地址是否是 16 字节对齐的。

(6) tcache_thread_shutdown(): unaligned tcache chunk detected

```
    /* Free all of the entries and the tcache itself back to
    the arena heap for coalescing.  */
    for (i = 0; i < TCACHE_MAX_BINS; ++i)
  {
    while (tcache_tmp->entries[i])
    {
    tcache_entry *e = tcache_tmp->entries[i];
    if (__glibc_unlikely (!aligned_OK (e)))
    malloc_printerr ("tcache_thread_shutdown(): "
       "unaligned tcache chunk detected");
    tcache_tmp->entries[i] = REVEAL_PTR (e->next);
    __libc_free (e);
    }
  }
```

Safe-Linking 检查的一部分。在销毁线程 Tcache 时检查 Tcache 头部的 Chunk 地址是否是 16 字节对齐的。

(7) realloc(): invalid pointer

```
  /* chunk corresponding to oldmem */
  const mchunkptr oldp = mem2chunk (oldmem);
  /* its size */
  const INTERNAL_SIZE_T oldsize = chunksize (oldp);

  /* Little security check which won't
  hurt performance: the allocator never wrapps around
  at the end of the address space.  Therefore we can
  exclude some size values which might appear here by
  accident or by "design" from some intruder.  */
  if ((__builtin_expect ((uintptr_t) oldp > (uintptr_t) -oldsize,\
  0)
     || __builtin_expect (misaligned_chunk (oldp), 0)))
    malloc_printerr ("realloc(): invalid pointer");
```

检查被 realloc 的 Chunk 的 size 是否合理以及 Chunk 指针是否 16 字节对齐。

(8) malloc(): unaligned fastbin chunk detected

```
#define REMOVE_FB(fb, victim, pp)       \
  do                 \
  {                \
  victim = pp;           \
  if (victim == NULL)         \
  break;              \
  pp = REVEAL_PTR (victim->fd);   \
  if (__glibc_unlikely (pp != NULL && misaligned_chunk (pp)))\
  malloc_printerr (\
 "malloc(): unaligned fastbin chunk detected"); \
  }                \
  while ((pp = \
  catomic_compare_and_exchange_val_acq (fb, pp, victim)) \
  != victim);
```

Safe-Linking 检查的一部分。在从 Fast Bin 中取出头部 Chunk 时，检查解密后的 Chunk→fd 是否 16 字节对齐。值得注意的是，REMOVE_FB 只在多线程运行时被调用，它使用专为多线程优化的原子操作。

(9) malloc(): unaligned fastbin chunk detected 2

```
  if (victim != NULL)
{
  if (__glibc_unlikely (misaligned_chunk (victim)))
  malloc_printerr (\
 "malloc(): unaligned fastbin chunk detected 2");
```

Safe-Linking 检查的一部分。在从 Fast Bin 中取出头部 Chunk 时，检查 Chunk 是否 16 字节对齐。

(10) malloc(): memory corruption (fast)

```
    idx = fastbin_index (nb);

    size_t victim_idx = fastbin_index (chunksize (victim));
    if (__builtin_expect (victim_idx != idx, 0))
  malloc_printerr ("malloc(): memory corruption (fast)");
```

在从 Fast Bin 中取出头部 Chunk 时，检查 Chunk 的 size 是否等于对应 Fast Bin 的 size。

(11) malloc(): unaligned fastbin chunk detected 3

```
  /* While bin not empty and tcache not full, copy chunks.  */
  while (tcache->counts[tc_idx] < mp_.tcache_count
   && (tc_victim = *fb) != NULL)
  {
    if (__glibc_unlikely (misaligned_chunk (tc_victim)))
  malloc_printerr (\
  "malloc(): unaligned fastbin chunk detected 3");
```

Safe-Linking 检查的一部分。在从 Fast Bin 移出 Chunk 到 Tcache 时，检查 Fast Bin 头部 Chunk 是否 16 字节对齐。

(12) malloc(): smallbin double linked list corrupted

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

    if ((victim = last (bin)) != bin)
    {
    bck = victim->bk;
    if (__glibc_unlikely (bck->fd != victim))
    malloc_printerr (\
    "malloc(): smallbin double linked list corrupted");
```

从 Small Bin 中取出 Chunk 时，检查 Chunk→bk→fd 是否等于 Chunk 本身。

(13) malloc(): invalid size (unsorted)

```
for (;; )
{
  int iters = 0;
  while ((victim = unsorted_chunks (av)->bk) !=\
  unsorted_chunks (av))
  {
    bck = victim->bk;
    size = chunksize (victim);
    mchunkptr next = chunk_at_offset (victim, size);

    if (__glibc_unlikely (size <= CHUNK_HDR_SZ)
      || __glibc_unlikely (size > av->system_mem))
    malloc_printerr ("malloc(): invalid size (unsorted)");
```

Unsorted Bin 增强检查的一部分。在循环遍历 Unsorted Bin 时，检查取出的 Chunk 的 size 是否合理 ([2 * size_t, sysmtem_mem])。

(14) malloc(): invalid next size (unsorted)

```
for (;; )
{
  int iters = 0;
  while ((victim = unsorted_chunks (av)->bk) !=\
  unsorted_chunks (av))
  {
    bck = victim->bk;
    size = chunksize (victim);
    mchunkptr next = chunk_at_offset (victim, size);

    ...
    if (__glibc_unlikely (chunksize_nomask (next) <\
    CHUNK_HDR_SZ)
      || __glibc_unlikely (chunksize_nomask (next) >\
      av->system_mem))
    malloc_printerr ("malloc(): invalid next size (unsorted)");
```

Unsorted Bin 增强检查的一部分。在循环遍历 Unsorted Bin 时，检查 Chunk 物理相邻的下一个 Chunk 的 size 是否合理 ([2 * size_t, sysmtem_mem])。

（15）malloc(): mismatching next->prev_size (unsorted)

```
for (;; )
{
  int iters = 0;
  while ((victim = unsorted_chunks (av)->bk) !=\
  unsorted_chunks (av))
  {
    bck = victim->bk;
    size = chunksize (victim);
    mchunkptr next = chunk_at_offset (victim, size);

        ...
    if (__glibc_unlikely ((prev_size (next) & ~(SIZE_BITS))\
    != size))
    malloc_printerr (\
    "malloc(): mismatching next->prev_size (unsorted)");
```

Unsorted Bin 增强检查的一部分。在循环遍历 Unsorted Bin 时，检查 Chunk 的 size 是否等于物理相邻的下一个 Chunk 的 prev_size。

(16) malloc(): unsorted double linked list corrupted

```
for (;; )
{
  int iters = 0;
  while ((victim = unsorted_chunks (av)->bk) !=\
  unsorted_chunks (av))
  {
    bck = victim->bk;
    size = chunksize (victim);
    mchunkptr next = chunk_at_offset (victim, size);

    ...
    if (__glibc_unlikely (bck->fd != victim)
      || __glibc_unlikely (victim->fd != unsorted_chunks (av)))
    malloc_printerr (\
    "malloc(): unsorted double linked list corrupted");
```

Unsorted Bin 增强检查的一部分。在循环遍历 Unsorted Bin 时，取头部 Chunk，检查 Chunk→bk→fd 是否等于 Chunk 本身，并且检查 Chunk→fd 是否等于 Unsorted Bin head。

(17) malloc(): invalid next->prev_inuse (unsorted)

```
for (;; )
{
  int iters = 0;
  while ((victim = unsorted_chunks (av)->bk) !=\
  unsorted_chunks (av))
  {
    bck = victim->bk;
    size = chunksize (victim);
    mchunkptr next = chunk_at_offset (victim, size);

    ...
    if (__glibc_unlikely (prev_inuse (next)))
    malloc_printerr (\
    "malloc(): invalid next->prev_inuse (unsorted)");
```

Unsorted Bin 增强检查的一部分。在循环遍历 Unsorted Bin 时，检查 Chunk 物理相邻的下一个 Chunk 是否将前一个 Chunk 标记为 prev_inuse (size 字段的末比特为 1)。

(18) malloc(): corrupted unsorted chunks 3

```
/* remove from unsorted list */
if (__glibc_unlikely (bck->fd != victim))
malloc_printerr ("malloc(): corrupted unsorted chunks 3");
unsorted_chunks (av)->bk = bck;
bck->fd = unsorted_chunks (av);
```

在将 Unsorted Chunk 取出时，检查 Chunk→bk→fd 是否等于 Chunk 本身。这和 glibc >= 2.19 引入的 malloc(): Unsorted double linked list corrupted 重复了。因此在 glibc >= 2.19 中它不可能被触发。

(19) malloc(): largebin double linked list corrupted (nextsize)

```
/* Always insert in the second position.  */
fwd = fwd->fd;
else
{
  victim->fd_nextsize = fwd;
  victim->bk_nextsize = fwd->bk_nextsize;
  if (__glibc_unlikely (fwd->bk_nextsize->fd_nextsize != fwd))
  malloc_printerr (\
  "malloc(): largebin double linked list corrupted (nextsize)");
  fwd->bk_nextsize = victim;
  victim->bk_nextsize->fd_nextsize = victim;
}
```

这个检查发生在将一个 Unsorted Chunk 加入 Large Bin (Binning)，更新 Large Bin 中的 nextsize 指针时。其中 victim 是要加入 Large Bin 的 Unsorted Chunk，fwd 是比 Large Bin 中比 victim 小的 Chunk，它将在 victim 的前方。

它会检查 fwd→bk_nextsize→fd_nextsize 是否等于 fwd 本身。

(20) malloc(): largebin double linked list corrupted (bk)

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

这个检查发生在将一个 Unsorted Chunk 加入 Large Bin (Binning) 时。其中 victim 是要加入 Large Bin 的 Unsorted Chunk，fwd 是比 Large Bin 中比 victim 小的 Chunk，它将在 victim 的前方。

它会检查 fwd→bk→fd 是否等于 fwd 本身。

(21) malloc(): corrupted unsorted chunks

```
    /* 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))
  malloc_printerr ("malloc(): corrupted unsorted chunks");
      remainder->bk = bck;
      remainder->fd = fwd;
      bck->fd = remainder;
      fwd->bk = remainder;
```

如果请求是 Large request，对应的 Large Bin 不为空，glibc 会在 Large Bin 中从小到大找到第一个合适的 Chunk，分割它，并把剩余部分 (remainder) 放入 Unsorted Bin。这个检查就发生在分割放入 Unsorted Bin 的过程中。

它会检查 Unsorted Bin head→fd→bk 是否等于 Unsorted Bin head 本身。

(22) malloc(): corrupted unsorted chunks 2

```
/* 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))
malloc_printerr ("malloc(): corrupted unsorted chunks 2");
  remainder->bk = bck;
  remainder->fd = fwd;
  bck->fd = remainder;
  fwd->bk = remainder;
```

如果请求的大小不能直接被任何 Bin 满足，必须查找比当前 Bin 更大的 Fast Bin，Small Bin 或者 Large Bin。找到比请求大小大的最小 Chunk，分割它，并把剩余部分 (remainder) 放入 Unsorted Bin。这个检查就发生在分割放入 Unsorted Bin 的过程中。

它会检查 Unsorted Bin head→fd→bk 是否等于 Unsorted Bin head 本身。

(23) malloc(): corrupted top size

```
use_top:
victim = av->top;
size = chunksize (victim);

if (__glibc_unlikely (size > av->system_mem))
  malloc_printerr ("malloc(): corrupted top size");
```

如果请求的大小不能被任何空闲 Chunk 满足，就需要从 Top Chunk 中分割。这个检查就发生在分割 Top Chunk 之前。

它会检查 Top Chunk 的 size 字段是否合理 (< system_mem)。

(24) free(): invalid pointer

```
/* Little security check which won't hurt performance: the
   allocator never wrapps around at the end of the address space.
   Therefore we can exclude some size values which might appear
   here by accident or by "design" from some intruder.  */
if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0)
    || __builtin_expect (misaligned_chunk (p), 0))
malloc_printerr ("free(): invalid pointer");
```

在进入 free 函数时检查被 free 的 Chunk 是否 16 字节对齐。

(25) free(): invalid size

```
/* We know that each chunk is at least MINSIZE bytes
   in size or a multiple of MALLOC_ALIGNMENT.  */
if (__glibc_unlikely (size < MINSIZE || !aligned_OK (size)))
malloc_printerr ("free(): invalid size");
```

在进入 free 函数时检查被 Free Chunk 的 size 是否合理，即它是否大于最小大小 (2 * size_t)，是否是 16 字节的倍数。

(26) free(): too many chunks detected in tcache

```
/* This test succeeds on double free.  However, we don't 100%
   trust it (it also matches random payload data at a 1 in
   2^<size_t> chance), so verify it's not an unlikely
   coincidence before aborting.  */
if (__glibc_unlikely (e->key == tcache_key))
  {
  tcache_entry *tmp;
  size_t cnt = 0;
  LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
  for (tmp = tcache->entries[tc_idx];
   tmp;
   tmp = REVEAL_PTR (tmp->next), ++cnt)
    {
  if (cnt >= mp_.tcache_count)
    malloc_printerr (\
    "free(): too many chunks detected in tcache");
```

这是 Tcache 双重释放检查和 Safe-Linking 检查的一部分。如果 Free Chunk 的 tcache_key 恰好等于当前线程的 tcache_key，怀疑它是被双重释放的 Chunk。一个个遍历 Tcache 中的 Chunk 做检查。

这一个检查是在遍历 Tcache 时，如果遍历到的 Chunk 序号比 Tcache 最大能存储的 Chunk 数量还大，怀疑 Tcache 被污染了。

(27) free(): unaligned chunk detected in tcache 2

```
/* This test succeeds on double free.  However, we don't 100%
   trust it (it also matches random payload data at a 1 in
   2^<size_t> chance), so verify it's not an unlikely
   coincidence before aborting.  */
if (__glibc_unlikely (e->key == tcache_key))
  {
  tcache_entry *tmp;
  size_t cnt = 0;
  LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
  for (tmp = tcache->entries[tc_idx];
   tmp;
   tmp = REVEAL_PTR (tmp->next), ++cnt)
    {
  ...
  if (__glibc_unlikely (!aligned_OK (tmp)))
    malloc_printerr (\
    "free(): unaligned chunk detected in tcache 2");
```

这是 Tcache 双重释放检查和 Safe-Linking 检查的一部分。如果 Free Chunk 的 tcache_key 恰好等于当前线程的 tcache_key，怀疑它是被双重释放的 Chunk。一个个遍历 Tcache 中的 Chunk 做检查。

这一个检查是在遍历 Tcache 时，如果遍历到的 Chunk 不与 16 字节对齐，怀疑 Tcache 被污染了。

(28) free(): double free detected in tcache 2

```
/* This test succeeds on double free.  However, we don't 100%
   trust it (it also matches random payload data at a 1 in
   2^<size_t> chance), so verify it's not an unlikely
   coincidence before aborting.  */
if (__glibc_unlikely (e->key == tcache_key))
  {
  tcache_entry *tmp;
  size_t cnt = 0;
  LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
  for (tmp = tcache->entries[tc_idx];
   tmp;
   tmp = REVEAL_PTR (tmp->next), ++cnt)
    {
  ...
  if (tmp == e)
    malloc_printerr (\
    "free(): double free detected in tcache 2");
```

这是 Tcache 双重释放检查和 Safe-Linking 检查的一部分。如果 Free Chunk 的 tcache_key 恰好等于当前线程的 tcache_key，怀疑它是被双重释放的 Chunk。一个个遍历 Tcache 中的 Chunk 做检查。

这一个检查是在遍历 Tcache 时，如果遍历到的 Chunk 和被释放的 Chunk 相同，那就是 Chunk 被双重释放了。

(29) free(): invalid next size (fast)

```
if (!have_lock)
  {
  __libc_lock_lock (av->mutex);
  fail = (chunksize_nomask (chunk_at_offset (p, size)) <=\
  CHUNK_HDR_SZ
    || chunksize (chunk_at_offset (p, size)) >= \
    av->system_mem);
  __libc_lock_unlock (av->mutex);
  }

if (fail)
  malloc_printerr ("free(): invalid next size (fast)");
  }
```

在将 Free Chunk 加入 Fast Bin 时检查与 Chunk 物理相邻的下一个 Chunk 的 size 字段是否合理 ([2 * size_t, sysmtem_mem])。

(30) double free or corruption (fasttop)

```
/* Atomically link P to its fastbin: P->FD = *FB; *FB = P;  */
mchunkptr old = *fb, old2;

if (SINGLE_THREAD_P)
  {
/* Check that the top of the bin is not
 the record we are going to
    add (i.e., double free).  */
if (__builtin_expect (old == p, 0))
  malloc_printerr ("double free or corruption (fasttop)");
```

在将 Free Chunk 加入 Fast Bin 时检查它是否等于 Fast Bin 的头部 Chunk。

(31) invalid fastbin entry (free)

```
/* Check that size of fastbin chunk at the top is the same as
   size of the chunk that we are adding.  We can dereference OLD
   only if we have the lock, otherwise it might have already
   been allocated again.  */
if (have_lock && old != NULL
&& __builtin_expect (fastbin_index (chunksize (old)) != idx, 0))
  malloc_printerr ("invalid fastbin entry (free)");
}
```

在将 Free Chunk 加入 Fast Bin 时检查 Fast Bin 头部 Chunk 的 size 是否等于 Free Chunk 的 size。

值得指出的是，此处要求 have_lock == 1。但 __libc_free 调用 _int_free 时不会设 have_lock 为 1，因此此处的检查通常不会触发。

(32) double free or corruption (top)

```
/* Lightweight tests: check whether the block is already the
   top block.  */
if (__glibc_unlikely (p == av->top))
  malloc_printerr ("double free or corruption (top)");
```

检查落在 Fast Bin 以外的 Free Chunk 是否就是 Top Chunk。这通常在并入 Top Chunk 的 Unsorted Chunk 被双重释放时发生。

(33) double free or corruption (out)

```
/* Or whether the next chunk is beyond the boundaries
of the arena.  */
if (__builtin_expect (contiguous (av)
          && (char *) nextchunk
          >= ((char *) av->top + chunksize(av->top)), 0))
malloc_printerr ("double free or corruption (out)");
```

检查落在 Fast Bin 以外的 Free Chunk，看它物理相邻的下一个 Chunk 是否超过了 Top Chunk 的边界。这通常在并入 Top Chunk (但 Top Chunk 已扩张) 的 Unsorted

Chunk 被双重释放时发生。

(34) double free or corruption (!prev)

```
/* Or whether the block is actually not marked used.  */
if (__glibc_unlikely (!prev_inuse(nextchunk)))
  malloc_printerr ("double free or corruption (!prev)");
```

检查落在 Fast Bin 以外的 Free Chunk，看它物理相邻的下一个 Chunk 的 prev_inuse 位是否为 0。这通常在未并入 Top Chunk 的 Unsorted Chunk 被 double free 时发生。

(35) free(): invalid next size (normal)

```
nextsize = chunksize(nextchunk);
if (__builtin_expect (chunksize_nomask (nextchunk) \
<= CHUNK_HDR_SZ, 0)
|| __builtin_expect (nextsize >= av->system_mem, 0))
  malloc_printerr ("free(): invalid next size (normal)");
```

检查落在 Fast Bin 以外的 Free Chunk，看它物理相邻的下一个 Chunk 的 size 是否合理 ([2 * size_t, sysmtem_mem])。

(36) corrupted size vs. prev_size while consolidating

```
/* consolidate backward */
if (!prev_inuse(p)) {
prevsize = prev_size (p);
size += prevsize;
p = chunk_at_offset(p, -((long) prevsize));
if (__glibc_unlikely (chunksize(p) != prevsize))
  malloc_printerr (\
  "corrupted size vs. prev_size while consolidating");
unlink_chunk (av, p);
}
```

后向合并时检查后向相邻的 Chunk 的 size 是否等于当前 Chunk 的 prev_size。

(37) free(): corrupted unsorted chunks

```
  /*
Place the chunk in unsorted chunk list. Chunks are
not placed into regular bins until after they have
been given one chance to be used in malloc.
  */

  bck = unsorted_chunks(av);
  fwd = bck->fd;
  if (__glibc_unlikely (fwd->bk != bck))
malloc_printerr ("free(): corrupted unsorted chunks");
  p->fd = fwd;
  p->bk = bck;
```

将 free Chunk 加入 Unsorted Bin 时检查 Unsorted Bin head→fd→bk 是否等于 Unsorted Bin head 本身。

(38) malloc_consolidate(): unaligned fastbin chunk detected

```
maxfb = &fastbin (av, NFASTBINS - 1);
fb = &fastbin (av, 0);
do {
p = atomic_exchange_acquire (fb, NULL);
if (p != 0) {
  do {
{
  if (__glibc_unlikely (misaligned_chunk (p)))
    malloc_printerr ("malloc_consolidate(): "
              "unaligned fastbin chunk detected");
```

Safe-Linking 检查的一部分。将 Fast Bin Chunk 取出时检查它是否是 16 字节对齐的。

(39) malloc_consolidate(): invalid chunk size

```
unsigned int idx = fastbin_index (chunksize (p));
if ((&fastbin (av, idx)) != fb)
malloc_printerr ("malloc_consolidate(): invalid chunk size");
```

将 Fast Bin Chunk 取出时检查它的 size 是否等于对应 Fast Bin 规定的 size。

(40) corrupted size vs. prev_size in fastbins

```
if (!prev_inuse(p)) {
  prevsize = prev_size (p);
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  if (__glibc_unlikely (chunksize(p) != prevsize))
    malloc_printerr (\
    "corrupted size vs. prev_size in fastbins");
  unlink_chunk (av, p);
}
```

将取出的 Fast Bin Chunk 后向合并，检查后向相邻的 Chunk 的 size 是否等于当前 Chunk 的 prev_size。

(41) realloc(): invalid old size

```
/* oldmem size */
if (__builtin_expect (chunksize_nomask (oldp) <=\
CHUNK_HDR_SZ, 0)
  || __builtin_expect (oldsize >= av->system_mem, 0)
  || __builtin_expect (oldsize != chunksize (oldp), 0))
malloc_printerr ("realloc(): invalid old size");
```

检查被 realloc 的 Chunk 原来的 size 是否合理 ([2 * size_t, sysmtem_mem])。

(42) realloc(): invalid next size

```
next = chunk_at_offset (oldp, oldsize);
INTERNAL_SIZE_T nextsize = chunksize (next);
if (__builtin_expect (chunksize_nomask (next) <=\
CHUNK_HDR_SZ, 0)
  || __builtin_expect (nextsize >= av->system_mem, 0))
malloc_printerr ("realloc(): invalid next size");
```

检查与被 realloc 的 Chunk 物理相邻的下一个 Chunk 的 size 是否合理 ([2 * size_t, sysmtem_mem])。

表 A.1　glibc 中的安全检查

安全检查	所处函数	目的
corrupted size vs. prev_size	unlink_chunk	防止篡改大小
corrupted double-linked list	unlink_chunk	加固双向链表
corrupted double-linked list (not small)	unlink_chunk	加固双向链表
munmap_chunk(): invalid pointer	munmap_chunk	确保参数合理
malloc(): unaligned tcache chunk detected	tcache_get	加固单向链表
tcache_thread_shutdown(): unaligned tcache chunk detected	tcache_thread_shutdown	加固单向链表
realloc(): invalid pointer	__libc_realooc	确保参数合理
malloc(): unaligned fastbin chunk detected	_int_malloc	加固单向链表
malloc(): unaligned fastbin chunk detected 2	_int_malloc	加固单向链表
malloc(): memory corruption (fast)	_int_malloc	防止篡改大小
malloc(): unaligned fastbin chunk detected 3	_int_malloc	加固单向链表
malloc(): smallbin double linked list corrupted	_int_malloc	加固双向链表
malloc(): invalid size (unsorted)	_int_malloc	防止篡改大小
malloc(): invalid next size (unsorted)	_int_malloc	防止篡改大小
malloc(): mismatching next->prev_size (unsorted)	_int_malloc	防止篡改大小
malloc(): unsorted double linked list corrupted	_int_malloc	加固双向链表
malloc(): invalid next->prev_inuse (unsorted)	_int_malloc	防止篡改大小
malloc(): corrupted unsorted chunks 3	_int_malloc	加固双向链表
malloc(): largebin double linked list corrupted (nextsize)	_int_malloc	加固双向链表
malloc(): largebin double linked list corrupted (bk)	_int_malloc	加固双向链表
malloc(): corrupted unsorted chunks	_int_malloc	加固双向链表
malloc(): corrupted unsorted chunks 2	_int_malloc	加固双向链表
malloc(): corrupted top size	_int_malloc	防止篡改大小
free(): invalid pointer	_int_free	确保参数合理
free(): invalid size	_int_free	确保参数合理
free(): too many chunks detected in tcache	_int_free	阻止双重释放
free(): unaligned chunk detected in tcache 2	_int_free	阻止双重释放
free(): double free detected in tcache 2	_int_free	阻止双重释放
free(): invalid next size (fast)	_int_free	防止篡改大小
double free or corruption (fasttop)	_int_free	阻止双重释放
invalid fastbin entry (free)	_int_free	防止篡改大小
double free or corruption (top)	_int_free	阻止双重释放
double free or corruption (out)	_int_free	阻止双重释放
double free or corruption (!prev)	_int_free	阻止双重释放
free(): invalid next size (normal)	_int_free	防止篡改大小
corrupted size vs. prev_size while consolidating	_int_free	防止篡改大小
free(): corrupted unsorted chunks	_int_free	加固双向链表
malloc_consolidate(): unaligned fastbin chunk detected	malloc_consolidate	加固单向链表
malloc_consolidate(): invalid chunk size	malloc_consolidate	防止篡改大小
corrupted size vs. prev_size in fastbins	malloc_consolidate	防止篡改大小
realloc(): invalid old size	_int_realloc	确保参数合理
realloc(): invalid next size	_int_realloc	确保参数合理

附录 B　测试结果

本节是使用面向跨版本堆管理器的堆利用测试套件完成的测试结果。包含两部分：25 个堆利用技巧在 2.15 - 2.37 版本的 glibc 中的可行性，以及 42 个堆安全检查在 2.15 - 2.37 版本的 glibc 中的存在性。

其中表 B.1、表 B.2、表 B.3 是堆利用技巧在各版本 glibc 中的可行性；表 B.4、表 B.5、表 B.6、表 B.7 是堆安全检查在各版本 glibc 中的存在性。

表 B.1　堆利用技巧在各版本 glibc 中的可行性
（版本 2.15 - 2.22）

堆利用技巧/glibc 版本	2.15	2.16	2.17	2.18	2.19	2.20	2.21	2.22
fastbin_dup	Y	Y	Y	Y	Y	Y	Y	Y
fastbin_dup_consolidate	Y	Y	Y	Y	Y	Y	Y	Y
fastbin_dup_into_stack	Y	Y	Y	Y	Y	Y	Y	Y
fastbin_reverse_into_tcache	N	N	N	N	N	N	N	N
house_of_botcake	N	N	N	N	N	N	N	N
house_of_einherjar	Y	Y	Y	Y	Y	Y	Y	Y
house_of_force	Y	Y	Y	Y	Y	Y	Y	Y
house_of_gods	N	N	N	N	N	N	N	N
house_of_kauri	N	N	N	N	N	N	N	N
house_of_lore	Y	Y	Y	Y	Y	Y	Y	Y
house_of_mind_fastbin	N	N	N	N	N	N	N	N
house_of_rabbit	Y	Y	Y	Y	Y	Y	Y	Y
house_of_spirit	Y	Y	Y	Y	Y	Y	Y	Y
large_bin_attack (larger chunk version)	Y	Y	Y	Y	Y	Y	Y	Y
large_bin_attack (smaller chunk version)	Y	Y	Y	Y	Y	Y	Y	Y
mmap_overlapping_chunks	Y	Y	Y	Y	Y	Y	Y	Y
overlapping_chunks	Y	Y	Y	Y	Y	Y	Y	Y
poison_null_byte (shrink free chunk version)	Y	Y	Y	Y	Y	Y	Y	Y
poison_null_byte (unlink fake chunk version)	Y	Y	Y	Y	Y	Y	Y	Y
tcache_dup	N	N	N	N	N	N	N	N
tcache_house_of_spirit	N	N	N	N	N	N	N	N
tcache_poisoning	N	N	N	N	N	N	N	N
tcache_stashing_unlink_attack	N	N	N	N	N	N	N	N
unsafe_unlink	Y	Y	Y	Y	Y	Y	Y	Y
unsorted_bin_attack	Y	Y	Y	Y	Y	Y	Y	Y

表 B.2　堆利用技巧在各版本 glibc 中的可行性
（版本 2.23 - 2.30）

堆利用技巧/glibc 版本	2.23	2.24	2.25	2.26	2.27	2.28	2.29	2.30
fastbin_dup	Y	Y	Y	Y	Y	Y	Y	Y
fastbin_dup_consolidate	Y	Y	Y	Y	Y	Y	Y	Y
fastbin_dup_into_stack	Y	Y	Y	Y	Y	Y	Y	Y
fastbin_reverse_into_tcache	N	N	N	Y	Y	Y	Y	Y
house_of_botcake	N	N	N	Y	Y	Y	Y	Y
house_of_einherjar	Y	Y	Y	Y	Y	Y	Y	Y
house_of_force	Y	Y	Y	Y	Y	Y	N	N
house_of_gods	Y	Y	Y	N	N	N	N	N
house_of_kauri	N	N	N	Y	Y	Y	Y	Y
house_of_lore	Y	Y	Y	Y	Y	Y	Y	Y
house_of_mind_fastbin	Y	Y	Y	N	Y	Y	Y	Y
house_of_rabbit	Y	Y	Y	Y	Y	Y	N	N
house_of_spirit	Y	Y	Y	Y	Y	Y	Y	Y
large_bin_attack (larger chunk version)	Y	Y	Y	Y	Y	Y	Y	Y
large_bin_attack (smaller chunk version)	Y	Y	Y	Y	Y	Y	Y	N
mmap_overlapping_chunks	Y	Y	Y	Y	Y	Y	Y	Y
overlapping_chunks	Y	Y	Y	Y	Y	Y	Y	Y
poison_null_byte (shrink free chunk version)	Y	Y	Y	Y	Y	Y	N	N
poison_null_byte (unlink fake chunk version)	Y	Y	Y	Y	Y	Y	Y	Y
tcache_dup	N	N	N	Y	Y	Y	N	N
tcache_house_of_spirit	N	N	N	Y	Y	Y	Y	Y
tcache_poisoning	N	N	N	Y	Y	Y	Y	Y
tcache_stashing_unlink_attack	N	N	N	Y	Y	Y	Y	Y
unsafe_unlink	Y	Y	Y	Y	Y	Y	Y	Y
unsorted_bin_attack	Y	Y	Y	Y	Y	N	N	N

表 B.3　堆利用技巧在各版本 glibc 中的可行性（版本 2.31 - 2.37）

堆利用技巧/glibc 版本	2.31	2.32	2.33	2.34	2.35	2.36	2.37
fastbin_dup	Y	Y	Y	Y	Y	Y	Y
fastbin_dup_consolidate	Y	Y	Y	Y	Y	Y	Y
fastbin_dup_into_stack	Y	Y	Y	Y	Y	Y	Y
fastbin_reverse_into_tcache	Y	Y	Y	Y	Y	Y	Y
house_of_botcake	Y	Y	Y	Y	Y	Y	Y
house_of_einherjar	Y	Y	Y	Y	Y	Y	Y
house_of_force	N	N	N	N	N	N	N
house_of_gods	N	N	N	N	N	N	N
house_of_kauri	Y	N	N	N	N	N	N
house_of_lore	Y	Y	Y	Y	Y	Y	Y
house_of_mind_fastbin	Y	Y	Y	Y	Y	Y	Y
house_of_rabbit	N	N	N	N	N	N	N
house_of_spirit	Y	Y	Y	Y	Y	Y	Y
large_bin_attack (larger chunk version)	Y	Y	Y	Y	Y	Y	Y
large_bin_attack (smaller chunk version)	N	N	N	N	N	N	N
mmap_overlapping_chunks	Y	Y	Y	Y	Y	Y	Y
overlapping_chunks	Y	Y	Y	Y	Y	Y	Y
poison_null_byte (shrink free chunk version)	N	N	N	N	N	N	N
poison_null_byte (unlink fake chunk version)	Y	Y	Y	Y	Y	Y	Y
tcache_dup	N	N	N	N	N	N	N
tcache_house_of_spirit	Y	Y	Y	Y	Y	Y	Y
tcache_poisoning	Y	Y	Y	Y	Y	Y	Y
tcache_stashing_unlink_attack	Y	Y	Y	Y	Y	Y	Y
unsafe_unlink	Y	Y	Y	Y	Y	Y	Y
unsorted_bin_attack	N	N	N	N	N	N	N

表 B.4　堆安全检查在各版本 glibc 中的存在性
（版本 2.15 - 2.20）

堆安全检查/glibc 版本	2.15	2.16	2.17	2.18	2.19	2.20
corrupted double-linked list (not small)	N	N	N	N	N	N
corrupted double-linked list	Y	Y	Y	Y	Y	Y
corrupted size vs. prev_size in fastbins	N	N	N	N	N	N
corrupted size vs. prev_size while consolidating	N	N	N	N	N	N
corrupted size vs. prev_size	N	N	N	N	N	N
double free or corruption (!prev)	Y	Y	Y	Y	Y	Y
double free or corruption (fasttop)	Y	Y	Y	Y	Y	Y
double free or corruption (out)	Y	Y	Y	Y	Y	Y
double free or corruption (top)	Y	Y	Y	Y	Y	Y
free(): corrupted unsorted chunks	Y	Y	Y	Y	Y	Y
free(): double free detected in tcache 2	N	N	N	N	N	N
free(): invalid next size (fast)	Y	Y	Y	Y	Y	Y
free(): invalid next size (normal)	Y	Y	Y	Y	Y	Y
free(): invalid pointer	Y	Y	Y	Y	Y	Y
free(): invalid size	Y	Y	Y	Y	Y	Y
free(): too many chunks detected in tcache	N	N	N	N	N	N
free(): unaligned chunk detected in tcache 2	N	N	N	N	N	N
invalid fastbin entry (free)	Y	Y	Y	Y	Y	Y
malloc(): corrupted top size	N	N	N	N	N	N
malloc(): corrupted unsorted chunks 2	Y	Y	Y	Y	Y	Y
malloc(): corrupted unsorted chunks 3	N	N	N	N	N	N
malloc(): corrupted unsorted chunks	Y	Y	Y	Y	Y	Y
malloc(): invalid next size (unsorted)	N	N	N	N	N	N
malloc(): invalid next->prev_inuse (unsorted)	N	N	N	N	N	N
malloc(): invalid size (unsorted)	N	N	N	N	N	N
malloc(): largebin double linked list corrupted (bk)	N	N	N	N	N	N
malloc(): largebin double linked list corrupted (nextsize)	N	N	N	N	N	N
malloc(): memory corruption (fast)	Y	Y	Y	Y	Y	Y
malloc(): mismatching next->prev_size (unsorted)	N	N	N	N	N	N
malloc(): smallbin double linked list corrupted	N	N	N	N	N	N
malloc(): unaligned fastbin chunk detected 2	N	N	N	N	N	N
malloc(): unaligned fastbin chunk detected 3	N	N	N	N	N	N
malloc(): unaligned fastbin chunk detected	N	N	N	N	N	N
malloc(): unaligned tcache chunk detected	N	N	N	N	N	N
malloc(): unsorted double linked list corrupted	N	N	N	N	N	N
malloc_consolidate(): invalid chunk size	N	N	N	N	N	N
malloc_consolidate(): unaligned fastbin chunk detected	N	N	N	N	N	N
munmap_chunk(): invalid pointer	Y	Y	Y	Y	Y	Y
realloc(): invalid next size	Y	Y	Y	Y	Y	Y
realloc(): invalid old size	Y	Y	Y	Y	Y	Y
realloc(): invalid pointer	Y	Y	Y	Y	Y	Y
tcache_thread_shutdown(): unaligned tcache chunk detected	N	N	N	N	N	N

表 B.5　堆安全检查在各版本 glibc 中的存在性
（版本 2.21 - 2.26）

堆安全检查/glibc 版本	2.21	2.22	2.23	2.24	2.25	2.26
corrupted double-linked list (not small)	Y	Y	Y	Y	Y	Y
corrupted double-linked list	Y	Y	Y	Y	Y	Y
corrupted size vs. prev_size in fastbins	N	N	N	N	N	N
corrupted size vs. prev_size while consolidating	N	N	N	N	N	N
corrupted size vs. prev_size	N	N	N	N	N	Y
double free or corruption (!prev)	Y	Y	Y	Y	Y	Y
double free or corruption (fasttop)	Y	Y	Y	Y	Y	Y
double free or corruption (out)	Y	Y	Y	Y	Y	Y
double free or corruption (top)	Y	Y	Y	Y	Y	Y
free(): corrupted unsorted chunks	Y	Y	Y	Y	Y	Y
free(): double free detected in tcache 2	N	N	N	N	N	N
free(): invalid next size (fast)	Y	Y	Y	Y	Y	Y
free(): invalid next size (normal)	Y	Y	Y	Y	Y	Y
free(): invalid pointer	Y	Y	Y	Y	Y	Y
free(): invalid size	Y	Y	Y	Y	Y	Y
free(): too many chunks detected in tcache	N	N	N	N	N	N
free(): unaligned chunk detected in tcache 2	N	N	N	N	N	N
invalid fastbin entry (free)	Y	Y	Y	Y	Y	Y
malloc(): corrupted top size	N	N	N	N	N	N
malloc(): corrupted unsorted chunks 2	Y	Y	Y	Y	Y	Y
malloc(): corrupted unsorted chunks 3	N	N	N	N	N	N
malloc(): corrupted unsorted chunks	Y	Y	Y	Y	Y	Y
malloc(): invalid next size (unsorted)	N	N	N	N	N	N
malloc(): invalid next->prev_inuse (unsorted)	N	N	N	N	N	N
malloc(): invalid size (unsorted)	N	N	N	N	N	N
malloc(): largebin double linked list corrupted (bk)	N	N	N	N	N	N
malloc(): largebin double linked list corrupted (nextsize)	N	N	N	N	N	N
malloc(): memory corruption (fast)	Y	Y	Y	Y	Y	Y
malloc(): mismatching next->prev_size (unsorted)	N	N	N	N	N	N
malloc(): smallbin double linked list corrupted	N	N	N	N	N	Y
malloc(): unaligned fastbin chunk detected 2	N	N	N	N	N	N
malloc(): unaligned fastbin chunk detected 3	N	N	N	N	N	N
malloc(): unaligned fastbin chunk detected	N	N	N	N	N	N
malloc(): unaligned tcache chunk detected	N	N	N	N	N	N
malloc(): unsorted double linked list corrupted	N	N	N	N	N	N
malloc_consolidate(): invalid chunk size	N	N	N	N	N	N
malloc_consolidate(): unaligned fastbin chunk detected	N	N	N	N	N	N
munmap_chunk(): invalid pointer	Y	Y	Y	Y	Y	Y
realloc(): invalid next size	Y	Y	Y	Y	Y	Y
realloc(): invalid old size	Y	Y	Y	Y	Y	Y
realloc(): invalid pointer	Y	Y	Y	Y	Y	Y
tcache_thread_shutdown(): unaligned tcache chunk detected	N	N	N	N	N	N

表 B.6　堆安全检查在各版本 glibc 中的存在性
（版本 2.27 - 2.32）

堆安全检查/glibc 版本	2.27	2.28	2.29	2.30	2.31	2.32
corrupted double-linked list (not small)	Y	Y	Y	Y	Y	Y
corrupted double-linked list	Y	Y	Y	Y	Y	Y
corrupted size vs. prev_size in fastbins	N	N	Y	Y	Y	Y
corrupted size vs. prev_size while consolidating	N	N	Y	Y	Y	Y
corrupted size vs. prev_size	Y	Y	Y	Y	Y	Y
double free or corruption (!prev)	Y	Y	Y	Y	Y	Y
double free or corruption (fasttop)	Y	Y	Y	Y	Y	Y
double free or corruption (out)	Y	Y	Y	Y	Y	Y
double free or corruption (top)	Y	Y	Y	Y	Y	Y
free(): corrupted unsorted chunks	Y	Y	Y	Y	Y	Y
free(): double free detected in tcache 2	N	N	Y	Y	Y	Y
free(): invalid next size (fast)	Y	Y	Y	Y	Y	Y
free(): invalid next size (normal)	Y	Y	Y	Y	Y	Y
free(): invalid pointer	Y	Y	Y	Y	Y	Y
free(): invalid size	Y	Y	Y	Y	Y	Y
free(): too many chunks detected in tcache	N	N	N	N	N	N
free(): unaligned chunk detected in tcache 2	N	N	N	N	N	Y
invalid fastbin entry (free)	Y	Y	Y	Y	Y	Y
malloc(): corrupted top size	N	N	Y	Y	Y	Y
malloc(): corrupted unsorted chunks 2	Y	Y	Y	Y	Y	Y
malloc(): corrupted unsorted chunks 3	N	Y	N	N	N	N
malloc(): corrupted unsorted chunks	Y	Y	Y	Y	Y	Y
malloc(): invalid next size (unsorted)	N	N	Y	Y	Y	Y
malloc(): invalid next->prev_inuse (unsorted)	N	N	Y	Y	Y	Y
malloc(): invalid size (unsorted)	N	N	Y	Y	Y	Y
malloc(): largebin double linked list corrupted (bk)	N	N	N	Y	Y	Y
malloc(): largebin double linked list corrupted (nextsize)	N	N	N	Y	Y	Y
malloc(): memory corruption (fast)	Y	Y	Y	Y	Y	Y
malloc(): mismatching next->prev_size (unsorted)	N	N	Y	Y	Y	Y
malloc(): smallbin double linked list corrupted	Y	Y	Y	Y	Y	Y
malloc(): unaligned fastbin chunk detected 2	N	N	N	N	N	Y
malloc(): unaligned fastbin chunk detected 3	N	N	N	N	N	Y
malloc(): unaligned fastbin chunk detected	N	N	N	N	N	Y
malloc(): unaligned tcache chunk detected	N	N	N	N	N	N
malloc(): unsorted double linked list corrupted	N	N	Y	Y	Y	Y
malloc_consolidate(): invalid chunk size	Y	Y	Y	Y	Y	Y
malloc_consolidate(): unaligned fastbin chunk detected	N	N	N	N	N	Y
munmap_chunk(): invalid pointer	Y	Y	Y	Y	Y	Y
realloc(): invalid next size	Y	Y	Y	Y	Y	Y
realloc(): invalid old size	Y	Y	Y	Y	Y	Y
realloc(): invalid pointer	Y	Y	Y	Y	Y	Y
tcache_thread_shutdown(): unaligned tcache chunk detected	N	N	N	N	N	Y

表 B.7　堆安全检查在各版本 glibc 中的存在性
（版本 2.33 - 2.37）

堆安全检查/glibc 版本	2.33	2.34	2.35	2.36	2.37
corrupted double-linked list (not small)	Y	Y	Y	Y	Y
corrupted double-linked list	Y	Y	Y	Y	Y
corrupted size vs. prev_size in fastbins	Y	Y	Y	Y	Y
corrupted size vs. prev_size while consolidating	Y	Y	Y	Y	Y
corrupted size vs. prev_size	Y	Y	Y	Y	Y
double free or corruption (!prev)	Y	Y	Y	Y	Y
double free or corruption (fasttop)	Y	Y	Y	Y	Y
double free or corruption (out)	Y	Y	Y	Y	Y
double free or corruption (top)	Y	Y	Y	Y	Y
free(): corrupted unsorted chunks	Y	Y	Y	Y	Y
free(): double free detected in tcache 2	Y	Y	Y	Y	Y
free(): invalid next size (fast)	Y	Y	Y	Y	Y
free(): invalid next size (normal)	Y	Y	Y	Y	Y
free(): invalid pointer	Y	Y	Y	Y	Y
free(): invalid size	Y	Y	Y	Y	Y
free(): too many chunks detected in tcache	Y	Y	Y	Y	Y
free(): unaligned chunk detected in tcache 2	Y	Y	Y	Y	Y
invalid fastbin entry (free)	Y	Y	Y	Y	Y
malloc(): corrupted top size	Y	Y	Y	Y	Y
malloc(): corrupted unsorted chunks 2	Y	Y	Y	Y	Y
malloc(): corrupted unsorted chunks 3	N	N	N	N	Y
malloc(): corrupted unsorted chunks	Y	Y	Y	Y	Y
malloc(): invalid next size (unsorted)	Y	Y	Y	Y	Y
malloc(): invalid next->prev_inuse (unsorted)	Y	Y	Y	Y	Y
malloc(): invalid size (unsorted)	Y	Y	Y	Y	Y
malloc(): largebin double linked list corrupted (bk)	Y	Y	Y	Y	Y
malloc(): largebin double linked list corrupted (nextsize)	Y	Y	Y	Y	Y
malloc(): memory corruption (fast)	Y	Y	Y	Y	Y
malloc(): mismatching next->prev_size (unsorted)	Y	Y	Y	Y	Y
malloc(): smallbin double linked list corrupted	Y	Y	Y	Y	Y
malloc(): unaligned fastbin chunk detected 2	Y	Y	Y	Y	Y
malloc(): unaligned fastbin chunk detected 3	Y	Y	Y	Y	Y
malloc(): unaligned fastbin chunk detected	Y	N	Y	Y	Y
malloc(): unaligned tcache chunk detected	Y	Y	N	Y	Y
malloc(): unsorted double linked list corrupted	Y	Y	Y	Y	Y
malloc_consolidate(): invalid chunk size	Y	Y	Y	Y	Y
malloc_consolidate(): unaligned fastbin chunk detected	Y	Y	Y	Y	Y
munmap_chunk(): invalid pointer	Y	Y	Y	Y	Y
realloc(): invalid next size	Y	Y	Y	Y	Y
realloc(): invalid old size	Y	Y	Y	Y	Y
realloc(): invalid pointer	Y	Y	Y	Y	Y
tcache_thread_shutdown(): unaligned tcache chunk detected	Y	Y	Y	Y	Y