1 Windows 堆的历史

Windows 的堆是内存中一块神秘的地方、一个耐人寻味的地方， 也是一个“乱糟糟”的地方。 微软并没有完全公开其操作系统中堆管理的细节。目前为止，对 Windows 堆的了解主要基 于技术狂热者、黑客、安全专家、逆向工程师等的个人研究成果。通过无数前辈们的努力工作， 现在， Windows NT4\2000 sp4 上的堆管理策略已经“基本”上被研究清楚了。 这里的“基本”是指堆管理中与攻击相关的数据结构和算法。出于堆固有的复杂多变的特性，要想真正搞清楚微软堆中的所有细节，还要寄希望于微软的共享精神，光靠黑客们的逆向、 试验和猜测是远远不够的。 在众多研究 Windows 堆的前辈中，有几位以他们精湛的技术、坚韧的耐心和优秀的共享精 神在安全领域而闻名。

（1） Halvar Flake： 2002 年的 black hat 上，他在演

讲“Third Generation Exploitation”中首 次挑战 Windows 的堆溢出，并揭秘了堆中一些重要的数据结构和算法。

（2） David Li tchfield： David 应该是安全技术界的传奇人物。除了他曾经发现的那些被横 扫世界的蠕虫所利用的 Oday漏洞外，他还是著名的安全咨询公司 NGS(Next Generation Security）的创始人。 David 在 2004 年 black hat 上演讲的“Windows Heap Overflows”首次比较全面地介 绍了 Windows 2000 平台下堆溢出的技术细节，包括了重要数据结构、堆分配算法、利用思路、 劫持进程的方法、执行 shellcode 时会遇到的问题等。那次演讲的白皮书（White paper）几乎是 所有研究 Windows 堆溢出人员的必读文献。

（3） Matt Conover：其演讲的“XP SP2 Heap Exploitation”中除了全面揭示了 Windows 堆 中与溢出相

关的所有数据结构和分配策略之外，最重要的是，他还提出了突破 Windows XP SP2 平台下重重安全机制的防护进行堆溢出的方法。在本书的写作过程中，我有幸得到了 Matt 的 热情帮助，他关于堆的深刻见解为本书增色不少。 本章内容来源于这些前辈们关于 Windows 堆管理机制研究成果的总结与整理。了解这些精 髓的知识除了对理解堆溢出利用至关重要外，对研究操作系统、文件系统的实现等也会有很大 的帮助。 现代操作系统在经过了若干年的演变后，目前使用的堆管理机制兼顾了内存有效利用、分 配决策速度、健壮性、安全性等因素，这使得堆管理变得异常复杂。本书关注的主要是 Win32 平台的堆管理策略。微软操作系统堆管理机制的发展大致可以分为三个阶段。

（1） Windows 2000～Windows XP SP1：堆管理系统只考虑了完成分配任务和性能因素，丝 毫没有考虑安全因素，可以比较容易发被攻击者利用。 （2） Windows XP 2～Windows 2003：加入了安全因素，比如修改了块首的格式并

加入安全 cookie，双向链表结点在删除时会做指针验证等。这些安全防护措施使堆溢出攻击变得非常困 难，但利用一些高级的攻击技术在一定情况下还是有可能利用成功。

（3） Windows Vista～Windows 7：不论在堆分配效率上还是安全与稳定性上，都是堆管理算 法的一个里程碑。 本书将主要讨论 Windows 2000～Windows XP SP1 平台的堆管理策略。

2 堆与栈的区别

第 2 章中提到过，程序在执行时需要两种不同类型的内存来协同配合。

一种是前面所讨论的系统栈。经过对栈溢出利用的学习，我们应该明白栈空间是在程序设计时已经规定好怎么使用，使用多少内存空间的。典型的栈变量包括函数内部的普通变量、数组等。栈变量在使用的时候不需要额外的申请操作，系统栈会根据函数中的变量声明自动在函数栈帧中给其预留空间。栈空间由系统维护，它的分配（如 sub esp ， xx ;）和回收（如 add esp,

xxx）都由系统来完成，最终达到栈平衡。所有的这些对程序员来说都是透明的。

另外一种内存就是本章将讨论的堆。从程序员的角度来看，堆具备以下特性。

（1）堆是一种在程序运行时动态分配的内存。所谓动态是指所需内存的大小在程序设计时不能预先决定，需要在程序运行时参考用户的反馈。

（2）堆在使用时需要程序员用专用函数进行申请，如 C 语言中的 malloc 等函数、 C++中的new 函数等都是最常见的分配堆内存的函数。堆内存申请有可能成功，也有可能失败，这与申请内存的大小、机器性能和当前运行环境有关。

（3）一般用一个堆指针来使用申请得到的内存，读、写、释放都通过这个指针来完成。

（4）使用完毕后需要把堆指针传给堆释放函数回收这片内存，否则会造成内存泄露。典型的释放函数包括 free、delete 等。

堆内存与栈内存的比较如表 5-1-1 所示。

栈只有 pop 和 push 两种操作，总是在“线性”变化，其管理机制也相对简单，所以，栈溢出的利用很容易掌握。

与“整齐”的栈不同，堆往往显得“杂乱无章”，所以堆溢

出的利用是内存利用技术的一个转折点。对堆利用技术的讨论也是安全技术界长久不衰的热门话题。

表 5-1-1　堆内存与栈内存的比较

	堆内存	栈内存
典型用例	动态增长的链表等数据结构	函数局部数组
申请方式	需要用函数申请，通过返回的指针使用。如 p=malloc(8)；	在程序中直接声明即可，如 char buffer[8]；
释放方式	需要把指针传给专用的释放函数，如 free	函数返回时，由系统自动回收
管理方式	需要程序员处理申请与释放	申请后直接使用，申请与释放由系统自动完成，最后达到栈区平衡
所处位置	变化范围很大 0x0012XXXX	
增长方向	由内存低址向高址排列（不考虑碎片等情况）	由内存高址向低址增加

3 堆的数据结构与管理策略

操作系统一般会提供一套 API 把复杂的堆管理机制屏蔽掉。因此，如果不是技术狂热者，普通的程序员是没有必要知道堆分配细节的。然而，要理解堆溢出利用技术，就必须适当了解一些堆的知识。为了不至于一下掉进二进制的技术细节，本小节先从宏观上介绍一下堆管理机制的原理，这些知识将有助于您更好地理解后续的技术细节，甚至启发您自己去挖掘堆中更深层次的东西。如果您是计算机系科班出身，那一定对本节的内容不陌生，因为这听起来更像是操作系统课程中的一个章节。

程序员在使用堆时只需要做三件事情：申请一定大小的内

存，使用内存，释放内存。我们下面将站在实现一个堆管理机制的设计者角度，来看看怎样才能向程序员提供这样透明的操作。

对于堆管理系统来说，响应程序的内存使用申请就意味着要在“杂乱”的堆区中“辨别”出哪些内存是正在被使用的，哪些内存是空闲的，并最终“寻找”到一片“恰当”的空闲内存区域，以指针形式返回给程序。

（1）“杂乱”是指堆区经过反复的申请、释放操作之后，原本大片连续的空闲内存区可能呈现出大小不等且空闲块、占用块相间隔的凌乱状态。

（2）“辨别”是指堆管理程序必须能够正确地识别哪些内存区域是正在被程序使用的占用块，哪些区域是可以返回给当前请求的空闲块。

（3）“恰当”是指堆管理程序必须能够比较“经济”地分配空闲内存块。如果用户申请使用 8 个字节，而返回给用户一片 512 字节的连续内存区域并将其标记成占用状态，这将造成大量的内存浪费，以致出现明明有内存却无法满足申请请求的情况。

为了完成这些基本要求，必须设计一套高效的数据结构来配合算法。现代操作系统的堆数据结构一般包括堆块和堆表两

类。

堆块： 出于性能的考虑，堆区的内存按不同大小组织成块，以堆块为单位进行标识，而不是传统的按字节标识。一个堆块包括两个部分：块首和块身。块首是一个堆块头部的几个字节，用来标识这个堆块自身的信息，例如，本块的大小、本块空闲还是占用等信息；块身是紧跟在块首后面的部分，也是最终分配给用户使用的数据区。

堆表： 堆表一般位于堆区的起始位置，用于索引堆区中所有堆块的重要信息，包括堆块的位置、堆块的大小、空闲还是占用等。堆表的数据结构决定了整个堆区的组织方式，是快速检索空闲块、保证堆分配效率的关键。堆表在设计时可能会考虑采用平衡二叉树等高级数据结构用于优化查找效率。现代操作系统的堆表往往不止一种数据结构。

堆的内存组织如图 5.1.1 所示

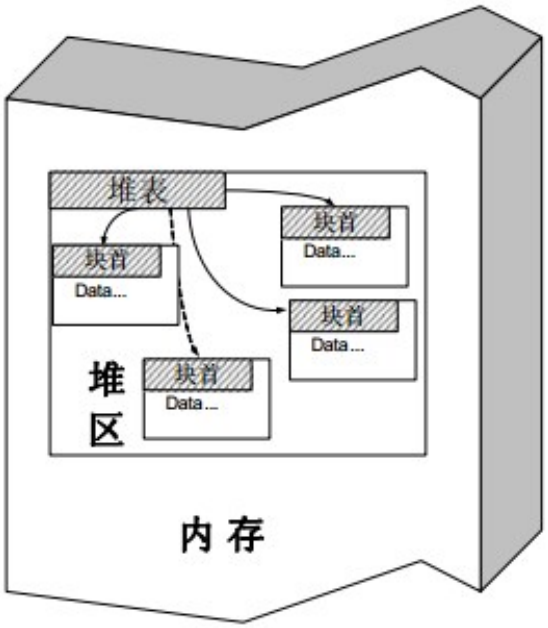

图 5.1.1　堆的内存组织

在 Windows 中，占用态的堆块被使用它的程序索引，而堆表只索引所有空闲态的堆块。其中，最重要的堆表有两种：空闲双向链表 Freelist（简称空表）和快速单向链表 Lookaside（简称快表）。

1.　空表

空闲堆块的块首中包含一对重要的指针，这对指针用于将空闲堆块组织成双向链表。按照堆块的大小不同，空表总共被分为 128 条。

堆区一开始的堆表区中有一个 128 项的指针数组，被称做空表索引（Freelist array）。该数组的每一项包括两个指针，用于标识一条空表。

如图 5.1.2 所示，空表索引的第二项（free[1]）标识了堆

中所有大小为 8 字节的空闲堆块，之后每个索引项指示的空闲堆块递增 8 字节，例如，free[2]标识大小为 16 字节的空闲堆块，free[3]标识大小为 24 字节的空闲堆块，free[127]标识大小为 1016 字节的空闲堆块。因此有：空闲堆块的大小＝索引项（ID） ×8（字节）

把空闲堆块按照大小的不同链入不同的空表，可以方便堆管理系统高效检索指定大小的空闲堆块。需要注意的是，空表索引的第一项（free[0]）所标识的空表相对比较特殊。这条双向链表链入了所有大于等于 1024 字节的堆块（小于 512KB）。这些堆块按照各自的大小在零号空表中升序地依次排列下去。

2. 快表

快表是 Windows 用来加速堆块分配而采用的一种堆表。这里之所以把它叫做“快表”是因为这类单向链表中从来不会发生堆块合并（其中的空闲块块首被设置为占用态，用来防止堆块合并）。

快表也有 128 条，组织结构与空表类似，只是其中的堆块按照单链表组织。快表总是被初 始化为空，而且每条快表最多只有 4 个结点，故很快就会被填满。

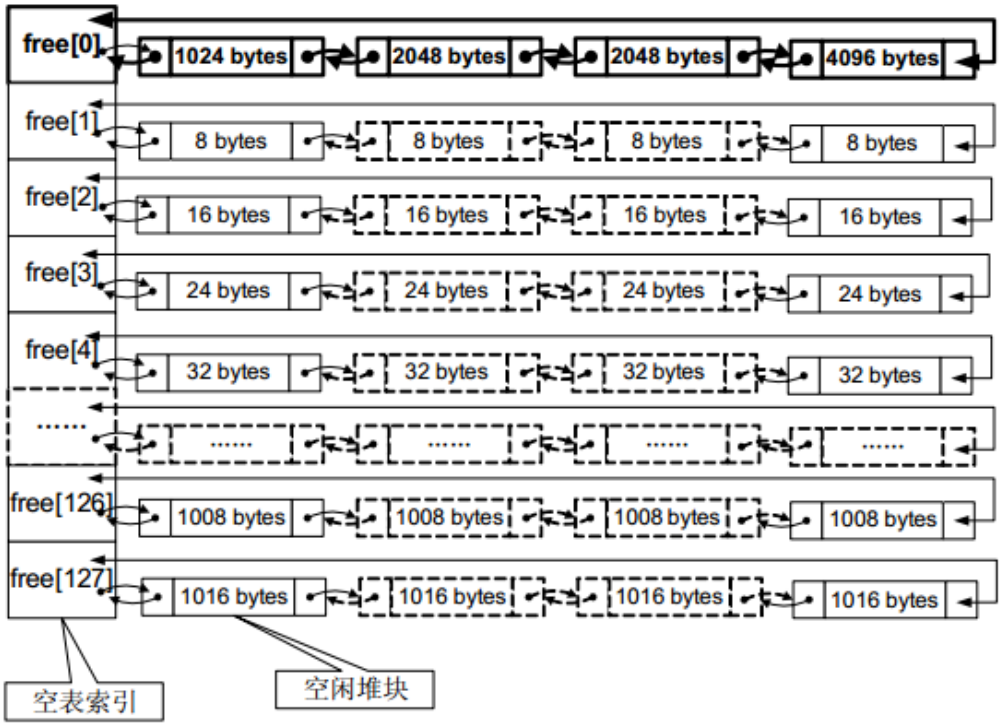

图 5.1.2　空闲双向链表（Freelist）

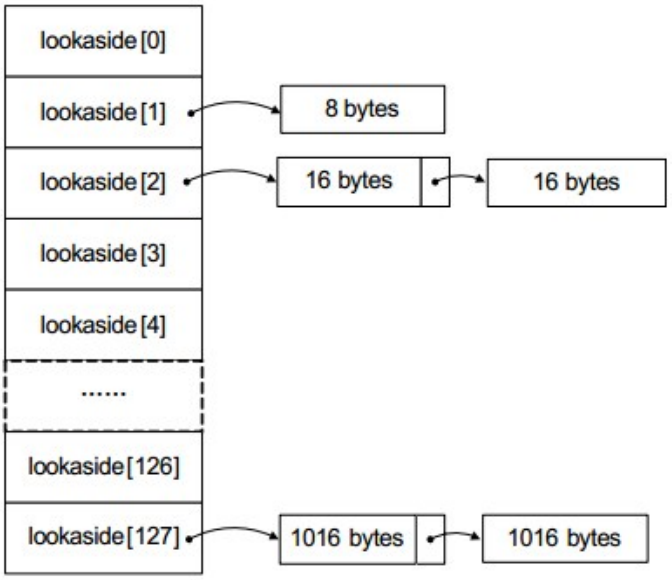

图 5.1.3　快速单向链表（Lookaside）

堆中的操作可以分为堆块分配、堆块释放和堆块合并（Coalesce）三种。其中，“分配”和“释放”是在程序提交申请和执行的，而堆块合并则是由堆管理系统自动完成的。

1.堆块分配

堆块分配可以分为三类：快表分配、普通空表分配和零号空表（free[0]）分配。

从快表中分配堆块比较简单，包括寻找到大小匹配的空闲堆块、将其状态修改为占用态、把它从堆表中“卸下”、最后返回一个指向堆块块身的指针给程序使用。

普通空表分配时首先寻找最优的空闲块分配，若失败，则寻找次优的空闲块分配，即最小的能够满足要求的空闲块。

零号空表中按照大小升序链着大小不同的空闲块，故在分配时先从 free[0]反向查找最后一个块（即表中最大块），看能否满足要求，如果能满足要求，再正向搜索最小能够满足要求的空闲堆块进行分配（这就明白为什么零号空表要按照升序排列了）。

堆块分配中的“找零钱”现象：当空表中无法找到匹配的“最优”堆块时，一个稍大些的块会被用于分配。这种次优分配发生时，会先从大块中按请求的大小精确地“割”出一块进行分配，然后给剩下的部分重新标注块首，链入空表。这里体现的就是堆管理系统的“节约”原则：买东西的时候用最合适的钞票，如果没有，就要找零钱，决不会玩大方。

由于快表只有在精确匹配时才会分配，故不存在“找钱”现象。

注意： 这里没有讨论堆缓存（heap cache）、低碎片堆（LFH）和虚分配。

2. 堆块释放

释放堆块的操作包括将堆块状态改为空闲，链入相应的堆表。所有的释放块都链入堆表的末尾，分配的时候也先从堆表末尾拿。另外需要强调，快表最多只有 4 项。

3. 堆块合并

经过反复的申请与释放操作，堆区很可能变得“千疮百孔”，产生很多内存碎片。为了合理有效地利用内存，堆管理系统还要能够进行堆块合并操作，如图 5.1.4 所示。

当堆管理系统发现两个空闲堆块彼此相邻的时候，就会进行堆块合并操作。

堆块合并包括将两个块从空闲链表中“卸下”、合并堆块、调整合并后大块的块首信息（如大小等）、将新块重新链入空闲链表。

题外话：实际上，堆区还有一种操作叫做内存紧缩（shrink the compact），由 RtlCompactHeap执行，这个操作的效果与磁盘碎片整理差不多，会对整个堆进行调整，尽量合并可

用的碎片。

在具体进行堆块分配和释放时，根据操作内存大小的不同， Windows 采取的策略也会有所不同。可以把内存块按照大小分为三类：

小块 ： SIZE<1KB

大块： 1KB≤SIZE<512KB

巨块： SIZE≥512KB

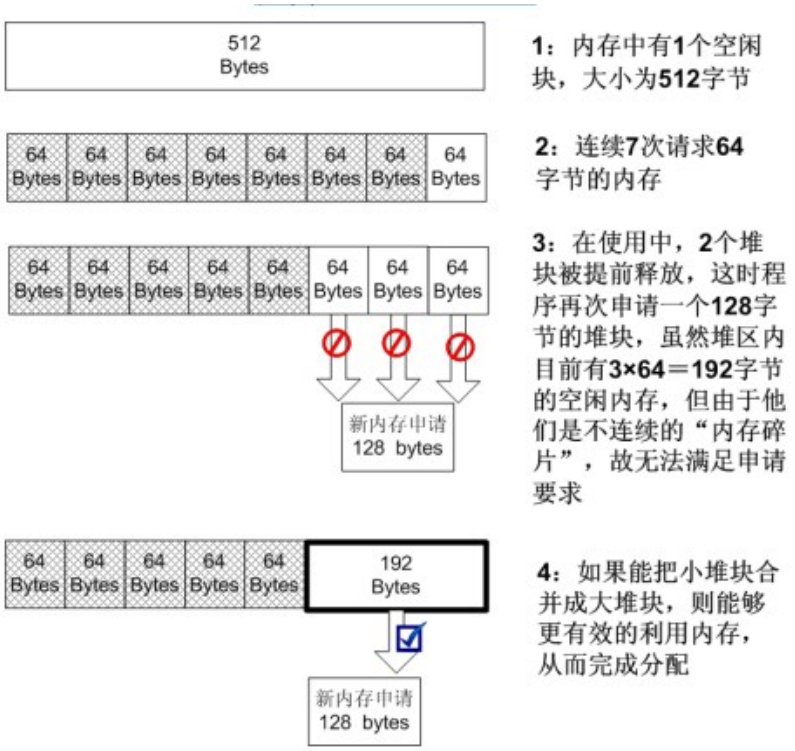

图 5.1.4 内存紧缩示意图

对应的分配和释放算法也有三类，我们可以通过表 5-1-2 来理解 Windows 的堆管理策略。

表 5-1-2　分配和释放算法

	分　　配	释　　放
小块	首先进行快表分配； 若快表分配失败，进行普通空表分配； 若普通空表分配失败，使用堆缓存（heap cache）分配； 若堆缓存分配失败，尝试零号空表分配（freelist[0]） 若零号空表分配失败，进行内存紧缩后再尝试分配； 若仍无法分配，返回 NULL	优先链入快表（只能链入 4 个空闲块）； 如果快表满，则将其链入相应的空表
大块	首先使用堆缓存进行分配； 若堆缓存分配失败，使用 free[0]中的大块进行分配	优先将其放入堆缓存 若堆缓存满，将链入 freelists[0]
巨块	一般说来，巨块申请非常罕见，要用到虚分配方法（实际上并不是从堆区分配的）。 这种类型的堆块在堆溢出利用中几乎不会遇到，本书中讨论暂不涉及这种情形	直接释放，没有堆表操作

最后，再强调一下 Windows 堆管理的几个要点。

（1）快表中的空闲块被设置为占用态，故不会发生堆块合并操作。

（2）快表只有精确匹配时才会分配，不存在“搜索次优解”和“找零钱”现象。

（3）快表是单链表，操作比双链表简单，插入删除都少用很多指令。

（4）综上所述，快表很“快”，故在分配和释放时总是优先使用快表，失败时才用空表。

（5）快表只有 4 项，很容易被填满，因此空表也是被频繁使用的。

综上所述， Windows 的堆管理策略兼顾了内存合理使用、分配效率等多方面的因素。