Practical Windows XP/2003 Heap Exploitation

Blackhat USA 2009

John McDonald (jrmcdona@us.ibm.com)
Chris Valasek (cvalasek@us.ibm.com)
IBM ISS X-Force Research

Table of Contents

Introduction

The era of straightforward heap exploitation is now well behind us. Heap exploitation has steadily increased in difficulty since its genesis in Solar Designer's ground-breaking Bugtraq post in July of 2000. This trend towards increasingly complicated exploitation is primarily a result of the widespread implementation of technical heap counter-measures in modern systems software. The effort required to write reliable heap exploits has steadily increased due to other factors as well: applications have become increasingly multi-threaded to take advantage of trends in hardware, and — in certain code — memory corruption vulnerabilities have become more nuanced and unique as a result of common, straightforward vulnerability patterns slowly but surely being audited out of existence.

The end result of all these defensive machinations is that now, more than ever, you need a fluid, application-aware approach to heap exploitation. The building blocks of such an approach are an extensive working knowledge of heap internals, an understanding of the contributing factors in heap determinism, various tactics for creating predictable patterns in heap memory, and, naturally, a collection of techniques for exploiting myriad different specific types of memory corruption in heap memory.

This paper is chiefly concerned with developing this foundational knowledge, focusing on the practical challenges of heap exploitation on Windows™ XP SP3 and Server 2003. Our first goal is to bring the reader up to speed on Windows Heap Manager internals and the current best of breed exploitation techniques. Once this foundation is established, we introduce new techniques and original research, which, at the end of the day, can turn seemingly bleak memory corruption situations into exploitable conditions. We close out the discussion by looking at Windows heap exploitation from a more general perspective, and discuss leveraging existing tools and techniques as part of one’s approach.

Overview

This paper is divided into three sections. The first section is **Fundamentals**, which covers the Windows Heap Manager in detail in order to provide foundational knowledge required to perform subtle heap exploitation.

The second section is **Tactics**, which covers multiple specific techniques for leveraging heap meta-data corruption. We start by covering the currently published best of breed techniques, and then move on to demonstrate a few new tricks and tactics.

The paper then moves on to **Strategy**. Here, we consider heap exploitation from a more general perspective, and look at tools and processes that prove useful when attacking complicated real-world software.

This inquiry is limited in scope to Windows XP SP3 and Windows Server 2003. We make heavy use of existing published tools and research, which we reference throughout the paper.

Prior Work

There are several excellent resources covering the security of the Windows Heap Manager. The following list, while not comprehensive, should provide the reader with sufficient background to follow this discussion.

- An excellent starting point are the three presentations by Matt Conover and Oded Horovitz. They each cover slightly different ground, but are all very informative, and provide insight to many undocumented aspects of the heap. (Conover and Horovitz 2004)
- There is a free chapter from the excellent book "Advanced Windows Debugging," which details how to use **windbg** to explore heap internals (Hewart and Pravat 2008).
- Alexander Sotirov's Heap Fung Shei library and paper are very informative, as they are chiefly concerned with heap determinism. (Sotirov 2007)
- Immunity also has an excellent paper with a similar focus on determinism and practical exploitation (Waisman 2007) (Waisman 2009). Their python ID heap code is very informative, as it encapsulates a lot of hard-won knowledge about the system's internals. (Immunity 2009)
- Brett Moore's papers look specifically at exploitation in the face of technical countermeasures. They represent the current most effective publicly documented attacks against the XPSP3 Heap Manager. Our specific technical

attacks complement and build on Moore’s attacks in both concept and execution. (Moore 2005) (Moore 2008)

- Finally, Ben Hawkes has done a considerable amount of research on attacking the Vista Heap Manager. (Hawkes 2008)

Fundamentals

Our first goal in this paper is to bring the reader up to speed on Windows Heap Manager internals. Any serious student of heap exploitation will need to spend time studying **ntdll.dll** up close, as in practice, questions will invariably arise about the heap's logical corner cases. Hopefully, we can save you some time and effort with our coverage of the internals, while giving you enough of a foundation by which to comprehend this paper.

The Windows Heap Manager is a sub-system used throughout Windows to provision dynamically allocated memory. It resides on top of the virtual memory interfaces provided by the kernel, which are typically accessed via **VirtualAlloc()** and **VirtualFree()**. Basically, the Heap Manager is responsible for providing a high-performance software layer such that software can request and release memory using the familiar **malloc()/free()/new/delete** idioms. **RtlAllocateHeap() / RtlFreeHeap()** and **NtAllocateVirtualMemory() / NtFreeVirtualMemory()** are the actual functions that form the interface around the Heap Manager.

Note: There are many pieces of software that implement wrappers on top of the Windows Heap Manager, and one should always pay careful attention to these. Specifically, the (msvc++) CRT heaps, pool / cache allocators in browsers, and custom allocators such as horde are of interest. We don't have enough time to examine these in this paper, but one should always study intermediate allocation wrappers for two reasons:

1. It is likely there is exploitable or malleable meta-data if the allocator isn't as hardened as the underlying system libraries.

2. Understanding allocator wrapper semantics are a pre-requisite to understanding the allocation behavior and patterns of a program.

Architecture

Each process typically has multiple heaps, and software can create new heaps as required. There is a default heap for the process, known as the *process heap*, and a pointer to this heap is stored in the PEB (*Process Environment Block*). All of the heaps in a process are kept in a linked list hanging off of the PEB.

In Windows XP SP3 and Win2K3, the Heap Manager is divided into two components, which are architecturally separate. The Front-End Heap Manager is a high-performance first-line subsystem, and the Back-End Heap Manager is a robust, general-purpose heap implementation.

Front-End Manager

The front-end manager has the first crack at servicing memory allocation and de-allocation requests. There are three choices for a front-end heap manager in XPSP3/2K3: none, *LAL* (*Look-aside Lists*), or *LFH* (*Low-Fragmentation Heap*).

The LAL front-end is responsible for processing allocation requests for byte amounts below 1024. It uses a series of singly linked-list data structures to keep track of various free chunks between sizes 0 – 1024. The LFH front-end handles requests between sizes 0 – 16k, with everything >= 16k being handled by the back-end.

The front-end managers exist to optimize performance of the heap. LAL and LFH are implemented using low-contention, lock-free / atomic-swap based operations, which allow for good performance on highly concurrent systems. LAL operates on specific block sizes, and doesn't perform coalescing or block-splitting in order to keep operations as quick as possible. LFH is considerably more complicated, as it addresses heap fragmentation on top of concurrency performance. LFH does this with the goal of improving memory and cache utilization. We discuss the front-ends in further detail later in the paper.

Back-End Manager

The back-end manager is used when the front-end heap manager isn't present, or isn't able to service a request. This can happen if the front-end doesn't have an appropriate chunk to service a given request, or if the system determines via heuristics that the performance advantages of the front-end are outweighed. The back-end manager is also responsible for allocations of larger chunks of memory (>1024 bytes when LAL is engaged, >16k when LFH is engaged). The back-end provides a general purpose, yet well-performing implementation of a heap. It is primarily concerned with keeping track of free memory chunks, and it contains optimizations and run-time heuristics for performance (such as the heap cache). We will discuss the inner workings of the back-end manager in detail in this paper.

Virtual Memory

The Heap Manager is built on top of virtual memory, so we'll briefly consider the Windows virtual memory semantics.

Reservation and Commitment

Windows distinguishes between *reserved memory* and *committed memory* (Microsoft ™ 2009).

Logically, a process first reserves a range of memory, which causes the kernel to mark off that range of virtual memory addresses as unavailable. Reserving memory doesn't actually map anything to the virtual addresses, so writes, reads, and executes of reserved memory will still cause an access violation. The kernel does not attempt to guarantee or pre-arrange backing memory for reserved memory either. Reserving is essentially just a mechanism for protecting a range of addresses from being allocated out from under the user.

After reserving a portion of the address space, the process is free to *commit* and *de-commit* memory within it. Committing memory is the act of actually mapping and backing the virtual memory. Processes can freely commit and de-commit memory within a chunk of reserved memory without ever un-reserving the memory (called *releasing* the memory).

In practice, many callers reserve and commit memory at the same time, and de-commit and release memory at the same time. Reserving, committing, de-committing, and releasing of memory are all performed by the **NtAllocateVirtualMemory()** and **NtFreeVirtualMemory()** functions.

Heap Base

Every heap that is created contains a vital data structure which we call the *heap base*. The heap base contains many critical data structures that are used by the memory manager to track of free and busy chunks. The following shows the contents of the default process heap in a process created under Windows XP Service Pack 3:

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 ''
```

Many of the data structures listed above can be leveraged by a clever attacker for the purposes of acquiring code execution. We'll cover several of the fields in detail as part of the Tactics section of the paper.

Note: It's worth pointing out that the heap base starts with a valid **_HEAP_ENTRY** structure. Ben Hawkes was able to leverage this in a clever multi-step attack against Vista, and this is certainly merits further exploration on XPSP3/2K3. (Hawkes 2008)

Memory Management

The Heap Manager organizes its virtual memory using *heap segments*, and uses UCR entries and segments to track uncommitted memory.

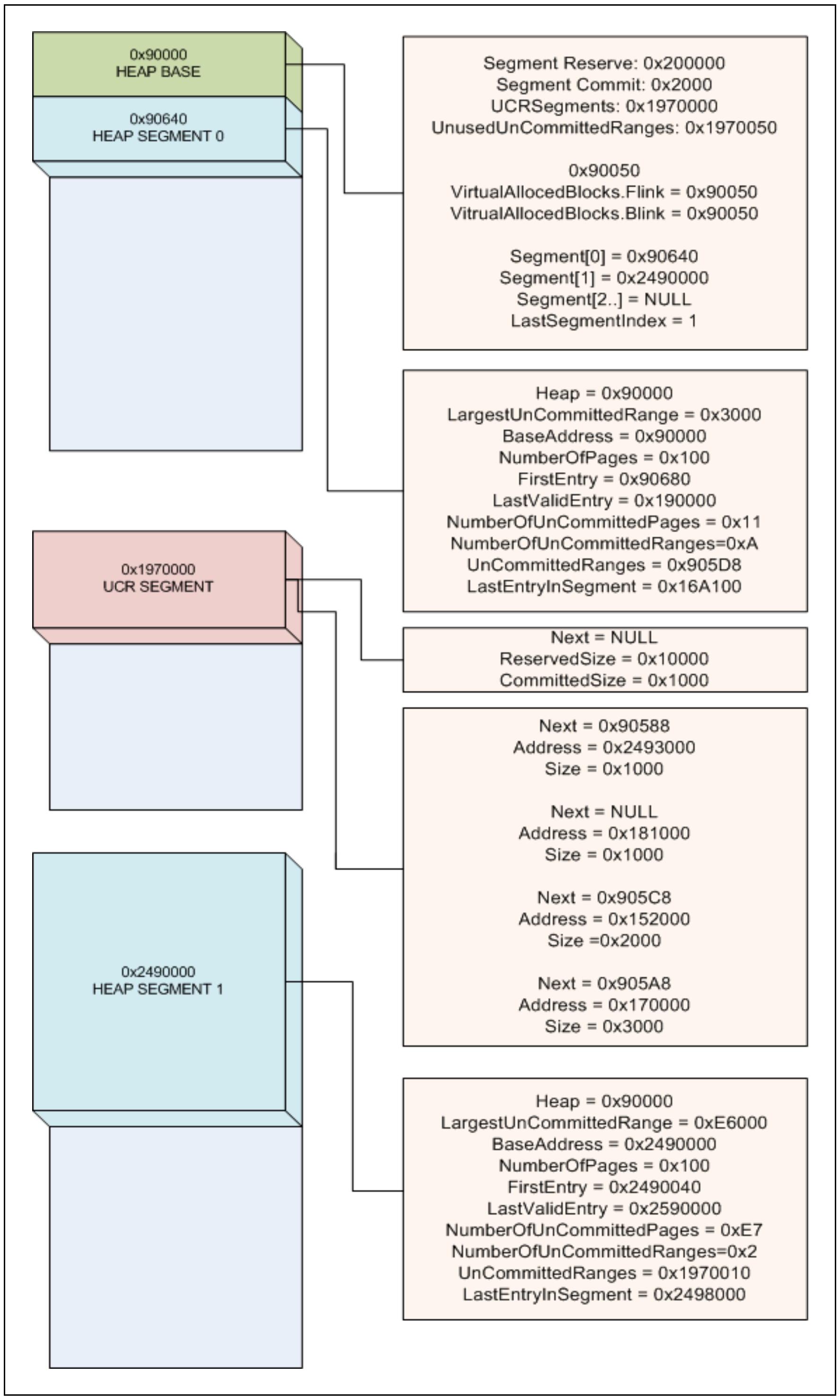

Figure 1 - Heap Segments and UCRs

Heap Segments

The back-end Heap Manager organizes its memory by segments, where each segment is a block of contiguous virtual memory that is managed by the system. (This is an internal Heap Manager data structure and not related to x86 segmentation.) When possible, the system will use committed memory to service requests, but if there isn't sufficient memory available, the Heap Manager will attempt to commit reserved memory within the heap's existing segments in order to fulfill the request. This could involve committing reserved memory at the end of the segment or committing reserved memory in holes in the middle of the segment. These holes would be created by previous de-committing of memory.

By default, the system reserves a minimum of 0x10000 bytes of memory when creating a new heap segment, and commits at least 0x1000 bytes of memory at a time. The system creates new heap segments as necessary and adds them to an array kept at the base of the heap. The first piece of data in a heap segment is typically the segment header, though the segment header for the base of the heap's segment comes after the heap header. Each time the heap creates a new segment, it doubles its reserve size, causing it to reserve larger and larger sections of memory.

Segment Base

Each heap contains an array of up to 64 pointers to segments at +0x58 from the base of the heap. This array of 64 **_HEAP_SEGMENT** structure pointers contains information about all of the segments associated with a given heap. Each segment represents a contiguous block of memory that is managed by the system. An empty segment structure is denoted by **NULL** (0x00000000) in the array. Each **_HEAP_SEGMENT** structure contains the following information:

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
```

As you can see amongst all the information include the heap associated with this segment, the **FirstEntry** in the segment, and the **LastValidEntry** in the segment. You can use this information to walk the heap to get all the metadata for each heap chunk, which is exactly what is done by the heap library provided with Immunity Debugger (Immunity 2009).

UCR Tracking

Each heap has a portion of memory set aside to track uncommitted ranges of memory. These are used by the segments to track all of the holes in their reserved address ranges. The segments track this with small data structures called UCR (*Un-committed Range*) entries. The heap keeps a global list of free UCR entry structures that the heap segments can request, and it dynamically grows this list to service the needs of the heap segments. At the base of the heap, **UnusedUnCommittedRanges** is a linked list of the empty UCR structures that can be used by the heap segments. **UCRSegments** is a linked list of the special UCR segments used to hold the UCR structures.

When a segment uses a UCR, it removes it from the heap's **UnusedUnCommittedRanges** linked list and puts it on a linked list in the segment header called **UnCommittedRanges**. The special UCR segments are allocated dynamically. The system starts off by reserving 0x10000 bytes for each UCR segment, and commits 0x1000 bytes (one page) at a time as additional UCR tracking entries are needed. If the UCR segment is filled and all 0x10000 bytes are used, the heap manager will create another UCR segment and add it to the **UCRSegments** list.

Note: Assuming that you're sufficiently mischievous, the above paragraph probably made you wonder about getting a UCR segment allocated contiguously with a virtual allocation that you can write past the end of. Check out Ben Hawkes' presentations for some more thoughts on this attack (Hawkes 2008).

Front End Manager

Look-Aside List (LAL)

The look-aside list array can typically be found in its own heap chunk at +0x688 from the base of the heap, which is pointed to by **FrontEndHeap** in the heap base. The LAL data structure is an array of 128 entries, each of size 0x30 bytes. Each entry in the array contains various performance-related variables, including the current length, a maximum length, and, most importantly, a pointer to the singly linked list of heap chunks that correspond to the bucket index. If there are no free chunks for a given bucket, the pointer is **NULL**. Similarly, the end of the singly linked list is denoted by a forward link pointer of **NULL**.

When an allocation occurs, a node is popped off of the head of the list. This can be performed with an atomic compare and swap operation in a concurrency-informed obstruction-free manner. The same goes for de-allocation; when a chunk is freed, it is pushed onto the head of the singly linked list, and the pointers are updated.

Note: When a block is placed into the LAL, the flags for the chunk are marked as **BUSY**, which prevents the back-end Heap Manager from splitting or coalescing the block. This may be counter-intuitive, as the block is technically free, but it is under the purview of the LAL front-end.

If a user's memory request is for less than (1024-8) bytes, it may be fulfilled by the LAL front-end. The diagram below shows an LAL front-end that has one entry for a size of 1016. This size corresponds to bucket number 127, which would service a user request for 1008 bytes. The LAL has no free chunks for the bucket for size 544 (entry number 68). Note that the buckets for size 0 and size 1 aren't used because every chunk requires at least 8 bytes for the chunk header.

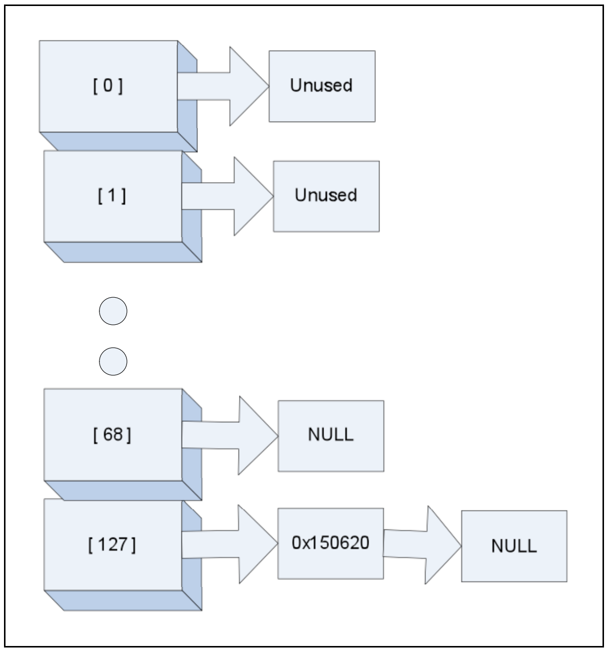

Figure 2 – Look-aside List Front End Example

Low Fragmentation Heap (LFH)

The LFH is a complicated heap front-end that addresses heap fragmentation as well as concurrent performance. This code is undocumented, though it has a more central role in later versions of Windows. Even with the LFH active, requests involving sizes > 16384 should automatically go to the heap back-end. The best source for further information is the Immunity Debugger source code. (Immunity 2009). Matt and Oded also cover it in further detail in their presentation. (Conover and Horovitz 2004).

Back End Manager

Freelists

The Back-end Heap Manager maintains several doubly linked lists to track free blocks in the heap. These are collectively referred to as the *free lists*, and they reside in an array at the base of the heap, starting at offset +0x178. There are separate lists for each possible block size below 1024 bytes, giving a total of 128 free lists (heap blocks are sized in multiples of 8.) Each doubly-linked free list has a sentinel head node located in the array at the base of the heap. Each head node contains two pointers: a forward link (**FLink**), and a back link (**BLink**).

FreeList[0] is special, which we'll discuss shortly. **FreeList[1]** is unused, and **FreeList[2]** through **FreeList[127]** are called the *dedicated free lists*. For these dedicated lists, all of the free blocks in the list are the same size, which corresponds to the array index * 8.

All blocks higher than or equal to size 1024, however, are kept in a single free list at **FreeList[0]**. (This slot is available for use because there aren't any free blocks of size 0.) The free blocks in this list are sorted from the smallest block to the largest block. So, **FreeList[0].Flink** points to the smallest free block (of size>=1024), and **FreeList[0].Blink** points to the largest free block (of size>=1024.)

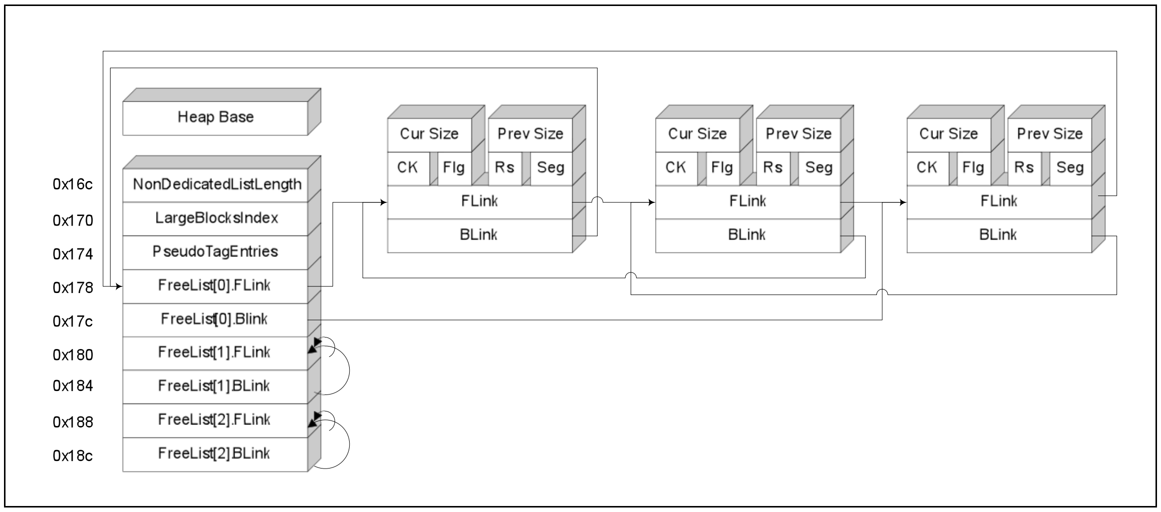

Figure 3 - Free Lists

Freelist Bitmap

The free lists also have a corresponding bitmap, called the **FreeListsInUseBitmap**, which is used for quickly scanning through the **FreeList** array. Each bit in the bitmap corresponds to one free list, and the bit is set if there are any free blocks in the corresponding list. The bitmap is located at +0x158 from the base of the heap, and it provides an optimized path for the system to service an incoming allocation request from the dedicated free lists. There are 128 bits in the bitmap (4 DWORDS), corresponding to the 128 free lists that handle allocations of size <1016.

For a given allocation request for a size < 1016, the front-end is first given a chance to service it. Assuming the LAL or LFH doesn't exist or doesn't service the request, the system then looks directly at the **FreeList[n]** linked list for the given size. Note that in this case, the bitmap is not consulted. If the **FreeList[n]** entry corresponding to the size requested by the user is empty, then the system proceeds to use the bitmap. From this point, it searches through the bitmap to find a set bit, which will cause it to find the next largest free block in the free lists. If the system runs out of bitmap, it then tries to pull a block from **FreeList[0]**.

For example, if a user requests 32 bytes from the heap and there isn't a chunk in **LAL[5]**, and **FreeList[5]** is empty, the bitmap is used to locate a chunk in the dedicated free lists that is greater than 40 bytes (starting the bitmap search at **FreeList[6]**).

Heap Cache

As we've discussed, all free blocks with a size greater than or equal to 1024 are stored in **FreeList[0]**. This is a doubly linked list, sorted by size from smallest to largest, with no additional enhancements for speed. Consequently, if **FreeList[0]** grows to hold a large number of blocks, the heap manager will need to traverse multiple list nodes every time it searches the list.

The *heap cache* is a performance enhancement that attempts to minimize the cost of frequent traversals of **FreeList[0]**. It does this by creating an external index for the blocks in **FreeList[0]**. *It's important to note that the Heap Manager doesn't actually move any free blocks into the cache.* The free blocks are still all kept in **FreeList[0]**, but the cache contains several pointers into the nodes within **FreeList[0]**,which are used as short-cuts to speed up traversal.

The cache is a simple array of buckets, where each bucket is **intptr_t** bytes in size, and contains either a **NULL** pointer or a pointer to a block in **FreeList[0]**. By default, the array contains 896 buckets, which accounts for each possible block size between 1024 and 8192. This is a configurable size, which we'll refer to here as the *maximum cache index.*

Each bucket contains a single pointer to the first block in **FreeList[0]** with the size represented by that bucket. If there is no entry in **FreeList[0]** with that exact size, the bucket contains NULL. The last bucket in the heap cache is unique: instead of representing the specific size 8192, it represents all sizes greater than or equal to the maximum cache index. So, it will point to the first block in **FreeList[0]** that is larger than the maximum size. (e.g. >=8192 bytes)

Most buckets are empty, so there is an additional bitmap that is used for fast searching of the array. This bitmap works just like the bitmap used to accelerate the free list.

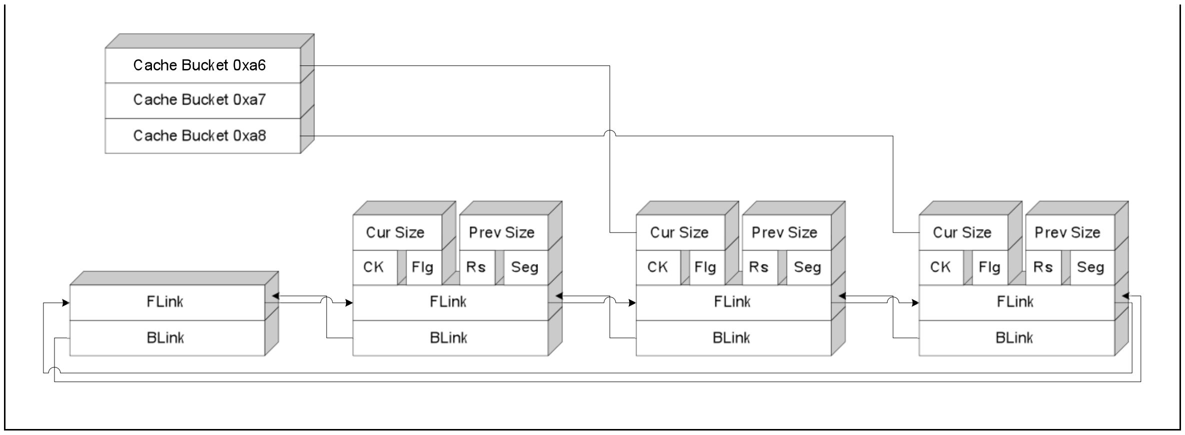

Figure 4 – Heap Cache and FreeList[0]

Virtual Alloc List

Each heap has a Virtual Allocation threshold *VirtualMemoryThreshold*. By default, this is 0xfe00, corresponding to memory chunks of size 508k or higher. Busy virtually allocated blocks are kept in a doubly-linked list at the base of the heap. When the blocks are returned back to the system, they are directly released to the kernel by the back-end Heap Manager. (**VirtualAllocdBlocks** at +0x50 and +0x54.)

Core Algorithms

Allocation Search

The free lists are searched for two reasons: to find a free block to service an allocation request, and to find the correct place to link in a free block. We cover linking related searches below.

If an allocation request is below 0x80 blocks (1024 bytes), and the front-end doesn't handle the request then, the free lists are searched. If the allocation search finds a free block to service a request, the block is unlinked from the free lists and then processed. This processing can involve splitting the block, coalescing the remainder block with its neighbors (this can involve unlinking of consumed neighbor blocks), and linking the remainder block to the free list.

The basic goal for the search algorithm is this: given a particular block size, find the first appropriate free block in the free lists with that size. If there aren't any with that exact size, then find the next largest available block. Let's look at some pseudo-code:

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]
}
```

If the size is below 1024 (0x80 * 8), the system goes directly to the free list at the base of the heap corresponding to the block size. If that free list has any elements, the searching algorithm will return a pointer to the last element on that doubly linked list.

If the free list for the requested size is empty, then the system needs to find the next largest block available. It then scans the free list bitmap, looking for a bit set corresponding to a larger block-size free list. (We abstracted the bitmap scanning code into a function for clarity.) If it finds a set bit in the bitmap, then the search returns the **BLink** of the corresponding free list.

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;
}
```

If the requested block size is >= 1024, or the system doesn't find an appropriate block using the bitmap, then it proceeds to search through the free blocks stored in **FreeList[0]**. As you recall, all free blocks higher than or equal to size 1024 are kept in this doubly linked list, sorted by size from smallest to largest. The above code queries the heap cache if it's present. It has a special case for a large block allocation request being fulfilled by a much larger free block. This will keep large collections of >16k free blocks from forming if there aren't free blocks of 4k or higher.

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;
}
```

If the heap cache isn't active, we need to search **FreeList[0]** manually. After checking to make sure **FreeList[0]** has at least one free block of sufficient size, the system starts with the first block in **FreeList[0]**. The system fetches this block from **FreeList[0].FLink**, and then walks through the linked list until an appropriately sized block is found. If the system walks all the way through the list and ends up back at the **FreeList[0]** head node, it knows that there are no suitable free blocks that meet the search query.

Unlinking

Unlinking is removing a particular free block from the free lists. This operation was the classic mechanism by which attackers exploited heap corruption vulnerabilities, so it now includes additional security checks. This is called *safe unlinking*.

Unlinking is used in allocations to pull an appropriate block off a free list in order to service a request. This is typically preceded by a search. Unlinking is also used when the heap manager obtains a pointer to a block through a different mechanism. This typically occurs during coalescing operations, as neighboring blocks that are subject to consolidation may need to be removed from the free lists. Coalescing can happen as part of both allocation and free operations. Finally, unlinking is used in allocation if the heap is extended, in order to remove the newly created free block.

Here is the basic pseudo-code for unlinking, assuming that the block that one wants to unlink is in the pointer **block**:

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;
}
```

This is basically standard code to unlink a node from a doubly linked list, with a few additions. First, there is a call to the heap cache that is used both for performance based metrics and to instruct the cache to purge an entry if necessary. Then, the safe unlink check is performed. Note that if this fails, the unlinking operation isn't performed, but it generally will fail without causing an exception, and the code will proceed.

After the block is unlinked, the system attempts to update the bitmap for the free list if necessary. Note that this performs an XOR (exclusive or) to toggle the bit, which can be another useful property for an attacker. Specifically, if the unlinking fails, but we have a **prevblock** that is equal to **nextblock**, it will toggle the corresponding bit in the bitmap. (This property was also noted in Brett Moore's *Heaps about Heaps* presentation and credited to Nicolas Waisman.)

Linking

Linking is taking a free block that is not on any list and placing it into the appropriate place in the free lists. In certain situations, the linking operation will first need to search the free lists to find this appropriate place. Linking is used in allocations when a block is split up and its remainder is added back to the free lists. It is also used in free operations to add a free block to the free lists. Let's look at some pseudo-code for the linking operation:

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)
            break;

        afterblock=afterblock->flink;
    }
}

// now find a pointer to the block that will be before us
beforeblock=afterblock->blink;
```

```
// 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);
```

This code does a simple search for the correct place to insert the block. If the size is <1024, it will insert the block onto the head of the appropriate free list. It will toggle the bitmap bit if the free list is empty.

If the size is >=1024, it will find the correct place in **FreeList[0]** to insert the block by walking through the doubly linked list. If the heap cache is present, it will use it to find the best place in the list to start the search. (Note this allows us more flexibility when we desynchronize the heap cache.)

Coalescing

Coalescing is performed before the back-end heap manager adds a chunk to the free lists. This checks the chunk's neighbors in contiguous memory to see if there are any immediately bordering free blocks that can be merged with the chunk being linked. This helps prevent heap fragmentation and reduces the amount of inline meta-data. When a chunk is passed to **HeapFree()**, if the front-end doesn't broker the request, the chunk is eventually passed to **RtlpCoalesceFreeBlocks()**. Let's look at pseudo-code for this function:

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;
```

As you can see, the code attempts to merge the chunk with both the preceding chunk in contiguous memory, and the subsequent chunk in contiguous memory. It unlinks any blocks subsumed by the coalescing from the free lists and Heap Cache. It will update the **FreeListBitMap** if **FLink** and **BLink** are equal, making it the last node on the list.

Note: This coalescing process can prove to be a significant hurdle when performing real-world attacks against heap meta-data. One general way to handle this is to seed the heap with data that specifies chunk flags indicating that neighboring blocks in contiguous memory are unavailable for coalescing process.

Security Mechanisms

Heap Cookie

The heap cookie is a security mechanism introduced in Windows XP Service Pack 2. It consists of a single byte value embedded in the chunk header. This value is checked when a busy block is freed via **RtlHeapFree()**. It is *not* checked when free blocks are allocated. The cookie uses information that would prove difficult to guess, and the intent is for the attacker to necessarily have to alter the cookie in order to corrupt the flags, segment index, or application data. The heap cookie is only 1-byte, hence giving an attacker a 1 in 256 chance of guessing the correct cookie value. The diagram below shows the chunk header with the heap cookie on Windows XP Service Pack 3:

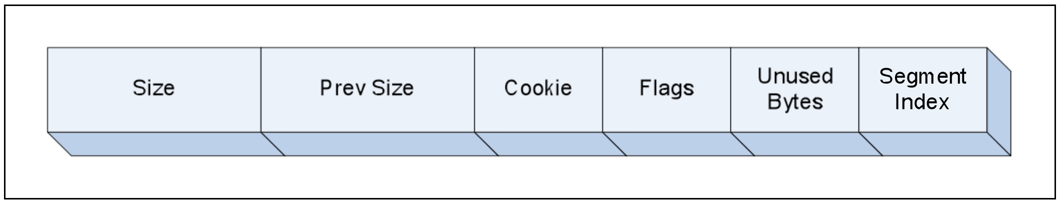

Figure 5 - Heap Cookie in Chunk Header

The address of the heap chunk determines the value of the heap cookie. The algorithm to check for cookie corruption is as follows:

Listing 10 – Cookie Check

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

If the heap cookie has been altered, **RtlpHeapReportCorruption** is called. If **HeapEnableTerminateOnCorruption** is set in newer Windows versions, the process will terminate.

Note: Practically, heap termination on corruption only works in newer versions of Windows, such as Window 2008 and Vista. We discuss this further below.

Safe Unlinking

Safe unlinking is another security mechanism introduced in Windows XP Service Pack 2. The code ensures that a block being unlinked actually belongs to the linked list before taking further action. Let's look at some pseudo-code:

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;
}
```

Safe unlinking checks require you to provide a pointer to a doubly-linked list node that you wish to have removed from the list. It first checks that **prevblock->FLink** points to the same address as **nextblock->BLink**. Then, it makes sure that both of those pointers point to the address of the block being removed from the doubly linked list. This works out to be difficult to subvert in practice, though Matt and Oded came up with a clever attack which, unfortunately, has some limiting pre-conditions. (Conover and Horovitz 2004)

If the security check fails, **RtlpHeapReportCorruption()** is called. In later versions of Windows (post-win2k3), if **HeapEnableTerminateOnCorruption** is set, the process will be terminated. Otherwise, for XP and Server 2003, execution occurs without the chunk being unlinked.

Process Termination

As mentioned above, if the heap algorithm detects that the heap meta-data has been corrupted, it will call **RtlpHeapReportCorruption**(). In Vista and 2008 Server, if **HeapSetInformation**() has been called for the heap and **HeapEnableTerminateOnCorruption** has been set, then the process will terminate. It should be noted that setting **HeapEnableTerminateOnCorruption** on Windows XP or Windows 2003 will not actually do anything. It is only currently supported on Windows 2008 Server and Windows Vista. Please see http://msdn.microsoft.com/en-us/library/aa366705(VS.85).aspx for more information on **HeapSetInformation**().

If the system fails a safe unlink of a block from the free lists due to the chunk being corrupted, the behavior depends a lot on what the chunk has been corrupted with. If the **FLink** or **BLink** pointers are invalid, it will cause an access violation when they are de-referenced, which will be terminal unless the calling function has some generous exception handling. If the **FLink** and **BLink** pointers are overwritten with readable addresses, then the block will fail the safe-unlink check.

Failing the safe-unlink check or the cookie check generally doesn’t impede an attack, as a failure doesn’t cause an unhandled exception to be raised or otherwise cause the process to terminate. As mentioned previously, the **HeapSetInformation() HeapEnableTerminationOnCorruption** option isn’t supported in Windows versions prior to Server 2008 and Vista. For 2003 and XP, if the image *gflag* **FLG_ENABLE_SYSTEM_CRIT_BREAKS** is set, the Heap Manager will call

DbgBreakPoint() and raise an exception if the safe-unlink check fails. This is an uncommon setting, as its security properties aren’t clearly documented.

Tactics

Lookaside List Link Overwrite

As we covered previously, the LAL front-end maintains 128 singly-linked lists of free chunks. There is a list for every valid request size below 1024 bytes (including the 8-byte chunk header). The LAL front-end is the first stop when allocating or freeing a chunk of data, and it adjusts its behavior according to run-time heuristics. Below is an example diagram of a **LAL[4]** that contains two free chunks (marked as BUSY to prevent the back-end from operating on them). **LAL[4]** corresponds to a block size of 4 (32 bytes), which will be used by the system to resolve 24 byte allocation requests (due to the 8-byte header).

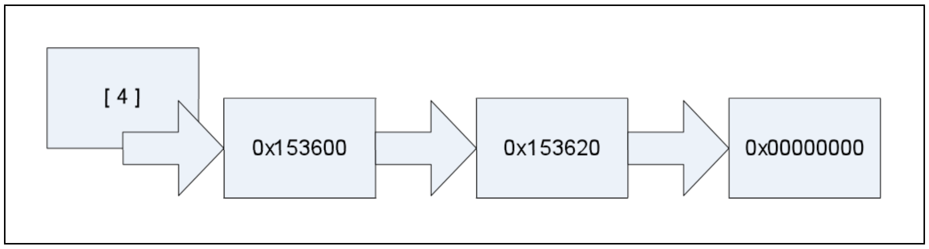

Figure 6- LAL Before Corruption

If you were to overflow into the chunk starting at 0x153620, overwriting the **FLink** pointer, you can co-opt the singly linked list for your own ends. (Conover and Horovitz 2004) If the attacker can sufficiently control a series of allocations and memory writes, they can change the flow of execution and ultimately gain arbitrary code execution. The diagram below shows the same list after an overflow into the block at 0x153620 occurs:

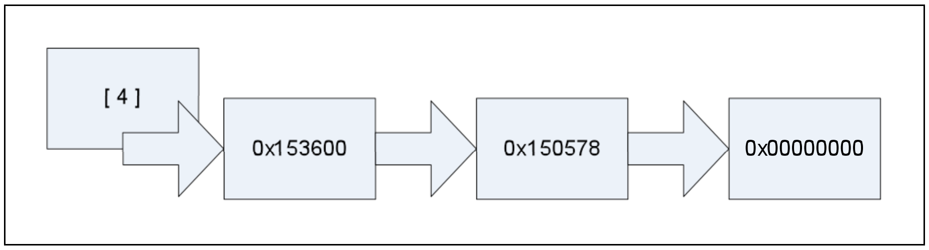

Figure 7 – LAL After Corruption

You can see that the address has changed to that of the **LockVariable** for the heap based at 0x150000. If you make an allocation for 24 bytes, you will receive the chunk at 0x153600, making the list look like:

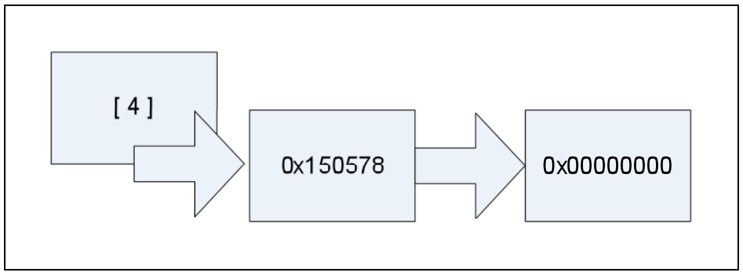

Figure 8 – LAL After Allocation

If a second allocation request can be made for 24 bytes, the memory manager will return 0x150578. As mentioned previously, this is the location of the **LockVariable** for the heap at 0x150000. If user controlled data can be written 4 bytes past this location it will overwrite the **CommitRoutine**, which can result in a call to a user supplied function pointer when the heap commits more memory. This is just one mechanism for abusing LAL chunk corruption, but there are naturally many options open to the attacker given the primitive of writing N arbitrary bytes to an arbitrary location X.

Bitmap Flipping Attack

Brett Moore's excellent presentation *Heaps about Heaps* (Moore 2007) covers bitmap flipping attacks in detail. To trace the evolution of this attack, we start In 2007, where a riddle was posted to the DailyDave mailing list by Nicolas Waisman:

Listing 12 – Nico's Riddle

Lets have a fun riddle to cheer up the spirit (Mate at 11pm, its all night insomnia.)

The riddle: Let said you are trying to exploit a remote service on an old Windows 2000 (whatever SP you want) and the primitive is the following:

inc [**edi**]   // you control **edi**

What would be the best option for **edi**?

This vulnerability is perfect for a very simple *Bitmap Flipping Attack* due to incrementing a value at a user controlled address. If the free list bitmap could be tricked into believing it has free chunks in a certain **FreeList** bucket that doesn't have any then one could overwrite data in the heap base. As discussed previously a **FreeList** bucket contains two pointers, a **FLink** and a **BLink**. If the bucket is empty, the pointers point back to the sentinel nodes located in the heap base. Let's look at an example **FreeList** for a heap starting at 0x150000:

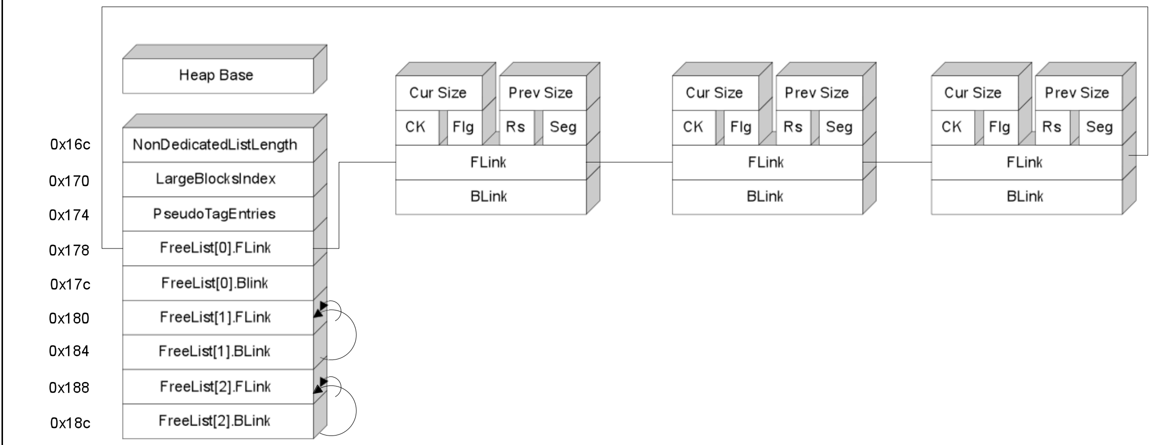

Figure 9 – Flipping Free List Example

You can see that the bucket for **FreeList[2]** is empty, hence both pointers are pointing to 0x150188. If the bitmap could be tricked into thinking that **FreeList[2]** contains free chunks, then it would return what it believe to be a free chunk starting at

0x150188. If user supplied data could then be written past this address, metadata on the heap base could be overwritten, resulting in code execution.

FreeList[0] Techniques

FreeList[0] is a doubly-linked list that contains all the free chunks that are greater than or equal to 1024 bytes in size. The blocks are ordered from smallest to largest in size. The forward links and back links are used when traversing the list, removing a node from the list, or adding a node to the list. Overwriting **FLink** and **BLink** values can alter the way these algorithms work. We'll briefly go over **FreeList[0]** searching, linking, and unlinking. For more detailed information, please read *'Exploiting FreeList[0] on XPSP2'* by Brett Moore. (Moore 2005)

Searching

When searching **FreeList[0]** for sufficiently sized chunks, the algorithm will first ensure that the last node in **FreeList[0]** is large enough to hold the request. If it is, it will start searching from the beginning of the list, otherwise more memory is committed to service the request. The search algorithm will then proceed to traverse the list until is finds a chunk large enough for the requested size, unlink that node, and return it to the user. If one can overwrite the **FLink** of an entry on **FreeList[0]** to a location that meets the requirements of the request, that address will be returned (Note. There are inner workings of the allocation algorithm intentionally left out for brevity). Lets look at an example **FreeList[0]**:

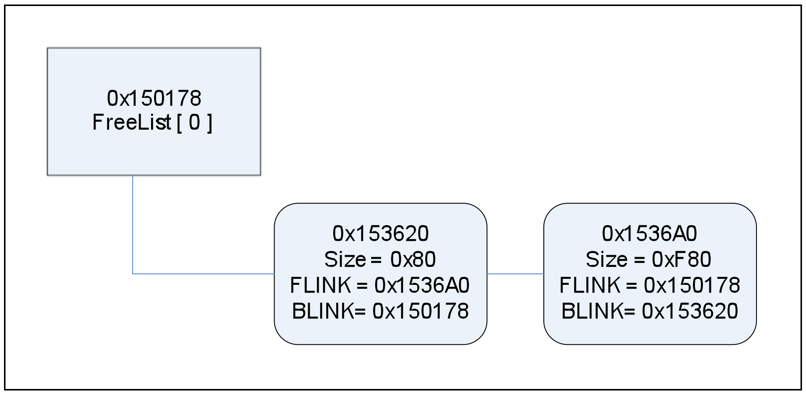

Figure 10 – FreeList[0] Searching

FreeList[0] contains two entries, one of size 0x80 and another with a size of 0xF80. If an allocation request arrives for 1032 (size includes the 8 byte heap chunk header) the search algorithm would see that the last block is large enough and start traversing at the beginning of the list. The first node in the list would not be large enough to service the request, so its **FLink** would be followed, leading to the chunk located at 0x1536A0. Since this block has a size of 0xF80, it would be split returning 1024 bytes (1032 – 8 byte chunk header) and put the remainder back on the free list. Leaving **FreeList[0]** looking like:

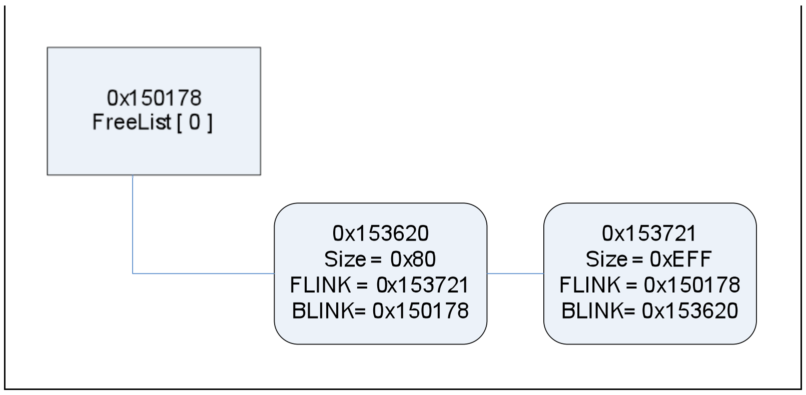

Figure 11 – FreeList[0] Searching II

If the **FLink** at 0x153620 were overwritten with a size smaller than the next allocation request and a **FLink** that appeared to be a valid heap chunk who's size is larger than that of the request then the request would attempt to split that block if necessary, unlink it (and fail but continue execution), and return.

Note: Think heap base.

Linking

Entries need to be inserted and deleted from the doubly-linked free list according to their usage. The memory manager takes care of this by changing their forward link and back link. This does not actually move the memory around, instead changing the chunk's pointers.

Without going into all the details, when a chunk of memory needs to be inserted into the **FreeList[0]** the memory manager will find a free chunk that is larger than the one to be freed and insert before it. The diagram below show 'Chunk C' that needs to be inserted between 'Chunk A' and 'Chunk B':

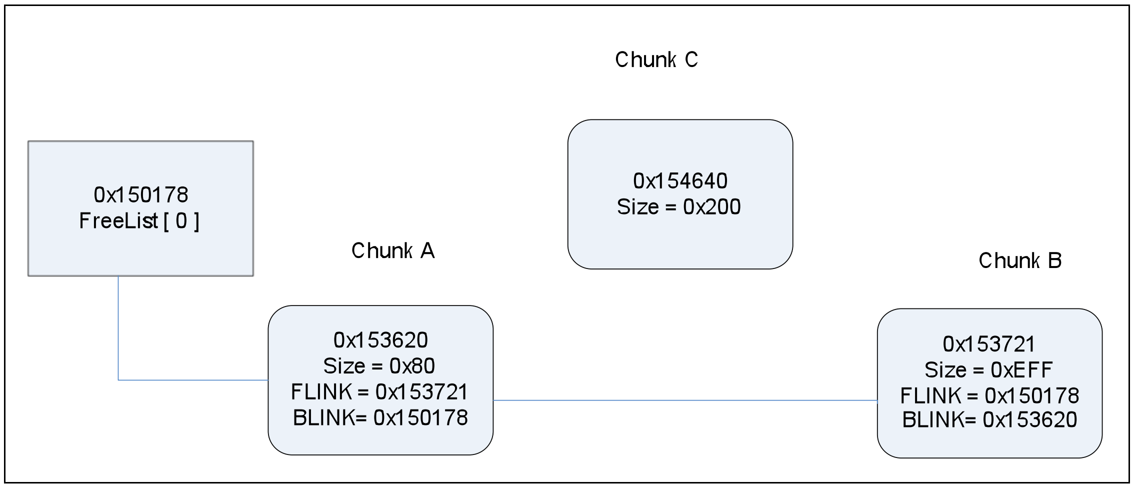

Figure 12 – FreeList[0] Linking

‘Chunk C’ is larger than ‘Chunk A’ and smaller than ‘Chunk B’, hence ‘Chunk C’ will be inserted in between the two. The way this is done is represented in the following pseudo-code:

Listing 13 – Linking Logic

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

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

This code links ‘Chunk C’ in and updates the **FLink** for ‘Chunk A’ and the **BLink** for ‘Chunk B’. If an overflow occurred into ‘Chunk B’ and kept the size larger than ‘Chunk C’ and the **BLink** was changed to an address of the attacker’s choosing then the address of the **BLink** would be overwritten with the address of ‘Chunk C’.

Note: Again, think heap base.

Tactics - New Techniques

Heap Cache

Overview

Public mentions of the heap cache are rare, as it's an undocumented internal data structure that is enabled dynamically at run-time. From an attacker's perspective, it is essentially an advisory subsystem that can often be avoided or disabled.

The heap cache is really only covered in one public resource: Horovitz and Conover's Syscan talk on the exploitation of the Windows heap (Horovitz and Conover 2004 Syscan). They cover it well, and the talk and accompanying code proved very useful in our efforts to understand the system. Our examination of the heap cache has very much built on their work, though we've observed a handful of key differences in the implementation that are probably the result of technology drift.

It's also worth noting that many of our specific technical attacks build on Brett Moore's and Nicolas Waisman's research and are very similar in nature, and we also detail attacks similar to Ben Hawkes' work against Vista.

Heap Cache Invocation

The heap cache isn't activated until the Heap Manager observes significant utilization of the **FreeList[0]** data structure at run-time. The actual initialization and synchronization with **FreeList[0]** is performed by the function **RtlpInitializeListIndex()**.

There are two performance metrics used by the Heap Manager, either of which will cause the heap cache to be instantiated:

1. 32 blocks must exist in **FreeList[0]** simultaneously

-or-

2. A total of 256 blocks must have been de-committed

Simultaneous Free Blocks

The first heuristic looks for signs of a fragmented **FreeList[0]**. Every time the Heap Manager adds a free block to the **Freelist[0]** doubly-linked list, it calls the function **RtlpUpdateIndexInsertBlock()**. Similarly, when it removes a free block from this linked list, it calls the function **RtlpUpdateIndexRemoveBlock()**.

Before the heap cache is invoked, these two functions simply maintain a counter that the Heap Manager uses to track the relative demand being placed on **FreeList[0]**. After the system observes a situation where there are 32 simultaneous entries in **FreeList[0]**, it then activates the heap cache by calling **RtlpInitializeListIndex()**.

Cumulative De-committing

The second heuristic is present in the **RtlpDeCommitFreeBlock()** function, which implements much of the logic that drives the de-committing process. Here, if the system de-commits a total of 256 blocks from the beginning of the process lifetime, it will activate the heap cache.

When the heap cache is activated by either heuristic, it triggers changes in the system's de-commitment policy. The essence of these changes is to perform much less de-commitment and instead save large free blocks in the free list.

De-committing Policy

For the purposes of understanding the basic logic, this brief and slightly inaccurate summary should suffice:

When the heap cache is turned off, the Heap Manager will generally de-commit free blocks above 1 page in size, assuming there is at least 64k of free blocks sitting in the free lists. (The block being freed counts towards the 64k, so a block of size 64k +/- 8k would necessarily be de-committed upon free.)

When the heap cache is turned on, the Heap Manager will generally avoid de-committing memory and instead save the blocks to the free list.

De-committing works by taking a large block and splitting it into three pieces: a piece leading up to the next page boundary, a set of whole pages encompassed cleanly by the block, and a piece containing any data past the last clean page boundary. The partial page pieces are coalesced and typically placed on the free lists (unless they coalesce to a large size), and the wholly encompassed set of contiguous pages is de-committed back to the kernel.

Simultaneous Entries

If an attacker has some control over the allocation and de-allocation within the target program, they can likely find a pattern of requests or activity that will lead to this heuristic being triggered. For example, in a relatively clean heap, the following pattern will cause the heap cache to be created after roughly 32 times through the loop:

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);
}
```

This works by creating free blocks that are surrounded by busy blocks. Each time through the loop, the allocation size is increased so that the existing holes in the heap won't be filled. In an active heap, a pattern like this should eventually engage the simultaneous block heuristic, if given sufficient iterations.

De-committing

For some applications, this may be easier for an attacker to utilize. In order to trigger this heuristic, the attacker needs to cause over 256 blocks to be de-committed over the lifetime of a process.

In order for a block to be de-committed, there needs to be at least 64k of free data in the heap (the block being freed will count towards this total). Also, the block has to be bigger than a page.

The simplest way to cause this to happen is to cause an allocation and free of a buffer of size 64k or higher 256 times. Here's a simple example:

Listing 15 – De-committing Threshold

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

Smaller buffers can be used as well if the heap total free size is already close to the 64k mark or can be grown there artificially. Coalescing behavior can be used if necessary to get sufficiently large blocks to be freed and de-committed.

De-synchronization

As we've established, the heap cache is a supplemental index into the existing **FreeList[0]** doubly-linked list data structure. One of the interesting findings is that the index data structure itself can be desynchronized from the other heap data structures. This can lead to multiple subtle attacks that can be initiated via various types of corruption of heap meta-data.

The basic idea of these attacks is to get the heap cache to point at semantically invalid memory for a particular bucket.

You can desynchronize the heap cache by altering the current size of any free chunk present in the heap cache. Depending on your ability to position yourself adjacent to a free buffer in memory (present in the cache index), this can be performed with a limited one-byte overflow in which you didn't have much control over the content.

The chief property exploited by these attacks is that when the heap cache code goes to remove an entry from the cache, it looks up that entry using the size as an index. So, if you change the size of a block, the heap cache can't find the corresponding array entry and fails open without removing it. This leaves a stale pointer that typically points to memory that is handed back to the application.

This stale pointer is treated like a legitimate entry in **FreeList[0]** for a particular size, which can allow multiple attacks. We'll cover a few different techniques for leveraging this situation and compare them with existing attacks.

Basic De-synchronization Attack

The simplest form of this attack works by corrupting the size of a large block that has already been freed and is resident in one of the 896 heap cache buckets. Let's look at a diagram of a potential set of free blocks:

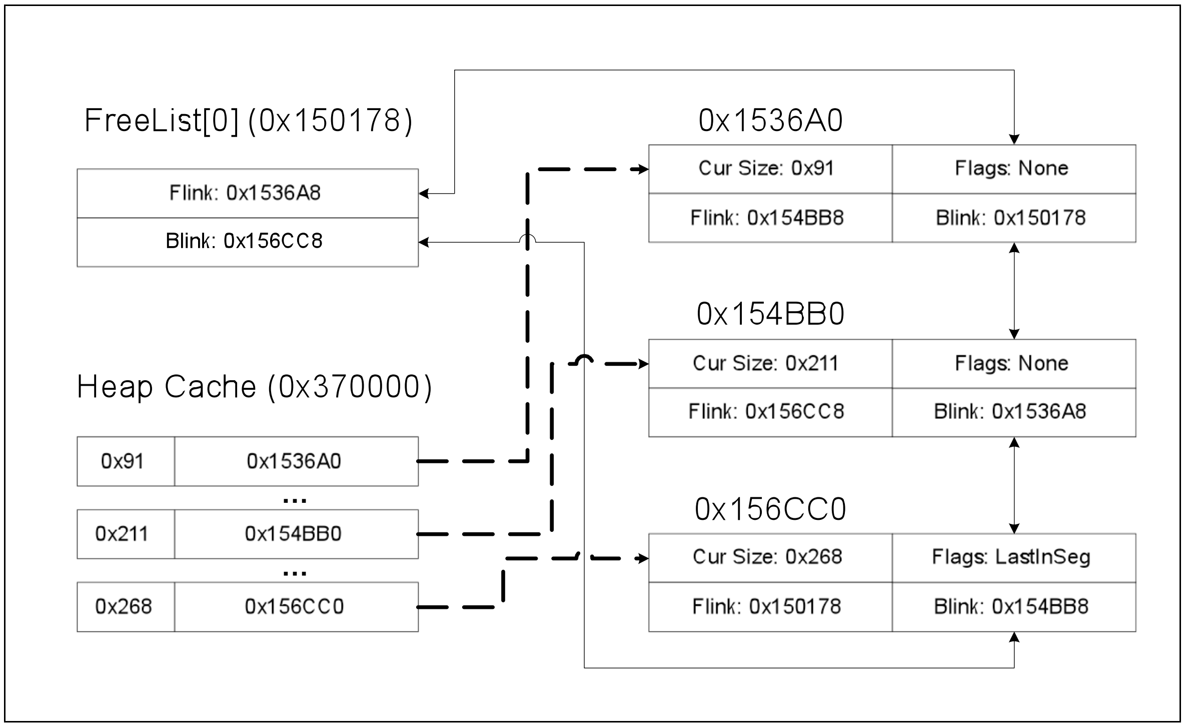

Figure 13 – Basic De-synchronization Attack Step 1

In the above diagram, we have a **FreeList[0]** with 3 blocks on it, of sizes 0x91 (0x488 bytes), 0x211 (0x1088 bytes), and 0x268 (0x1340 bytes). The heap cache is instantiated, and we see that it has entries in the three buckets corresponding to our blocks.

Let's assume that we can do a one-byte overflow of a NUL into the current size field of the free block at 0x154BB0. This will change the block size from 0x211 to 0x200, shrinking the block from 0x1088 bytes to 0x1000 bytes. This will look like the following:

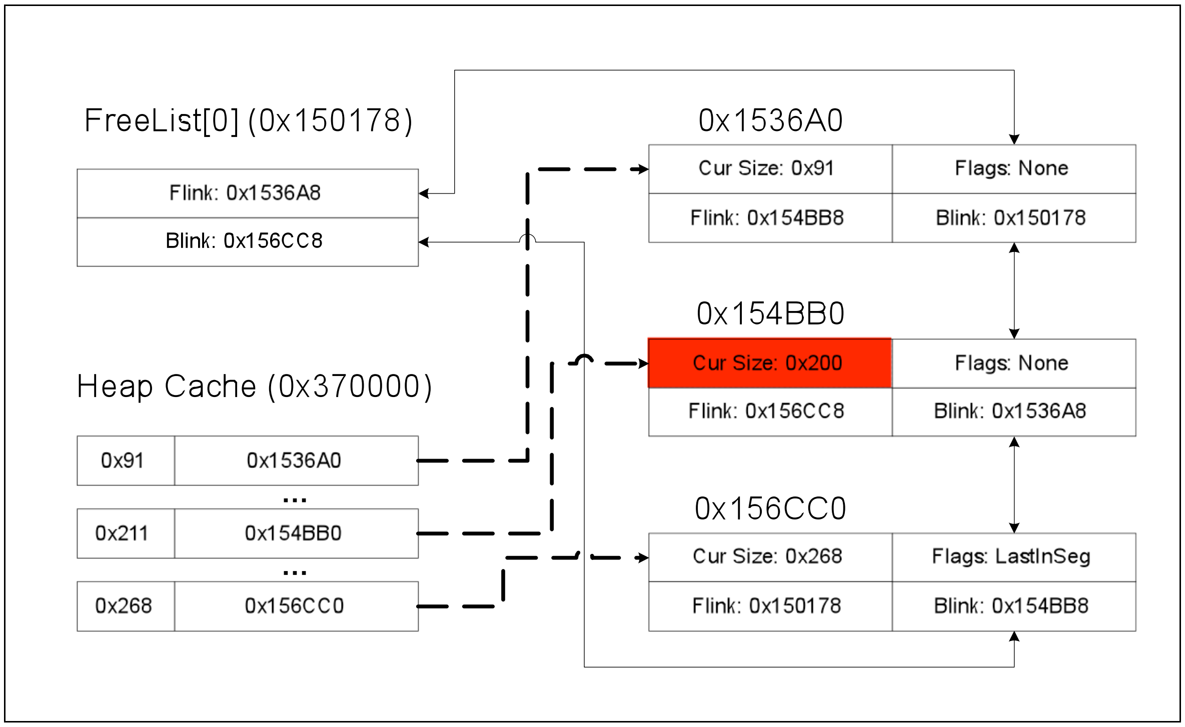

Figure 14 – Basic De-synchronization Attack Step 2

Now, we've changed the size of the free chunk at 0x154BB0, which has desynchronized our **FreeList[0]** with the index maintained by the heap cache. Currently, the bucket for block size 0x211 is pointing to a free block that is actually of block size 0x200.

Note: Throughout the rest of the heap cache discussion, we will refer to sizes in terms of "block size," which we define as 1/8th of the size in bytes. This corresponds to the size values that are actually stored in memory in the current/previous size fields, and used as indexes in the look-aside, cache, and free lists.

For the simplest form of the attack, let's assume that the next memory operation the application does is an allocation for block size 0x200 (0x1FF taking the 8 byte header into account.) First, the Heap Manager does a search for a size of 0x200.

The system will go to the heap cache, see that the bitmap field for 0x200 indicates that it is empty, and then it will scan the heap cache's bitmap. It will find our entry at 0x211, and return the pointer to the chunk at 0x154BB0.

Now, the allocation routine receives its answer from the search, and verifies it is large enough to service the allocation. It is, so the Heap Manager proceeds to perform an unlink. The unlink will call **RtlpUpdateIndexRemoveBlock()**, passing it our block, which will pull out the size 0x200 from our block, and check the heap cache to see if the bucket for 0x200 points to our block. It does not since it's empty, and the function will return without doing anything.

The unlinking will work since the block is correctly linked into **FreeList[0]**, but the heap cache will not be updated. Since, for simplicity, we chose an allocation size of 0x200 (4096 bytes), the block will be the perfect size and there won't be any block splitting or re-linking. So, no errors will be fired, and the system will return 0x154BB8 back to the application, leaving the system in the following state:

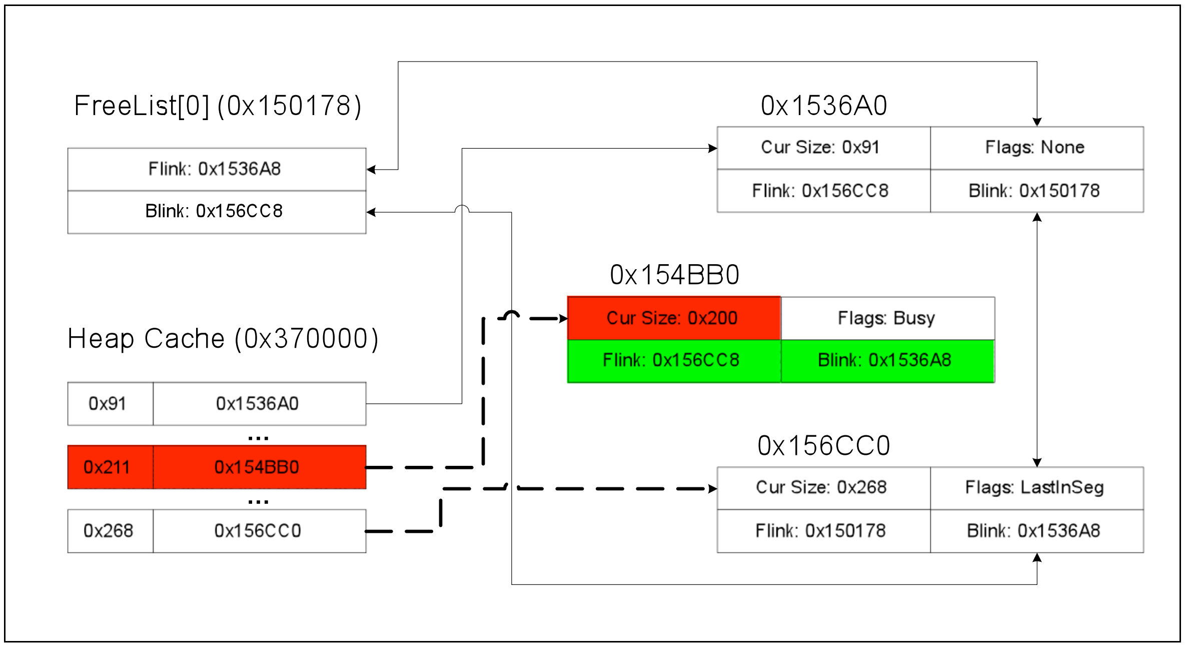

Figure 15 – Basic De-synchronization Attack Step 3

You can see that the **FreeList[0]** now contains only two blocks: 0x1536A0 and 0x156CC0. The heap cache, however, contains a stale entry to 0x154BB0, which is now a block that is marked as busy by the system. Since it is a busy block, the application will start writing its data where the **FLink** and **BLink** entries are.

For the simplest form of this attack, we'll just assume that from here, the application does multiple allocations for size 0x200 (4096 bytes). Each time this happens, the system will go to the heap cache. The heap cache will find the stale entry at 0x211, and the system will see that the block at 0x154BB0 is big enough to service the request. (It never checks the flags to ensure that the block is actually free.)

Now, the system will attempt to do a safe unlink of the stale block from **FreeList[0]**. This could cause an access violation depending on what the application fills in for the **FLink** and **BLink** fields. If **FLink** and **BLink** are overwritten with invalid addresses, the Heap Manager will cause an exception when it attempts to dereference them. If the **FLink** and **BLink** pointers are untouched, or are overwritten with readable addresses, then the stale block will fail the safe-unlink check.

Failing the safe-unlink check generally doesn't impede an attack, as a failure doesn't cause an unhandled exception to be raised or otherwise cause the process to terminate. (The **HeapSetInformation() HeapEnableTerminationOnCorruption** option isn't supported in Windows versions prior to Server 2008 and Vista. For 2003 and XP, if the image *gflag* **FLG_ENABLE_SYSTEM_CRIT_BREAKS** is set, the Heap Manager will call **DbgBreakPoint()** and raise an exception if the safe-unlink check fails. This is an uncommon setting, as its security properties aren't clearly documented.)

The end result of the attack technique is that multiple independent allocations will return the same address to the application:

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
```

Summary

If the attacker can change the current size field of a block that is pointed to by the heap cache, that block won't be properly removed from the heap cache, and a stale pointer will remain.

This attack looked at the simplest form of this situation. The result of the attack is that every time the application attempts to allocate a particular size, it receives the same pointer to a block of memory already in use. The exploitability of this would depend specifically on what the application did with this memory. In general, you'd look for a situation where a pointer to an object or function was at the same logical location as a field that was based on attacker-supplied data. Then, you'd try to create a sequence of events where the pointer would be initialized, the user-malleable data would be stored, and then the now corrupt pointer would be used.

Prerequisites

- The attacker must be able to write into the current size field of a block that is free and present in the heap cache.
- The attacker must be able to anticipate a future allocation size that the application will request.
- The attacker must avoid allocations that cause splitting or re-linking of the corrupt block (or anticipate and plan for them).
- **HeapAlloc()** incorrectly returning the same address for independent requests must create an exploitable condition in the application.

Existing Attacks

The primary prerequisite for our first de-synchronization attack is the ability to corrupt the current size field of a large, free block pointed to by the heap cache. This corruption can be caused with a one or two byte limited-control overflow, which makes it somewhat unique among the known attack techniques. To see how this might be useful, let's briefly review the current set of attacks:

Size Field Corruption

Assuming that we can only overwrite 1-4 bytes, there are a few existing attacks that may be useful. Specifically, if an attacker can overwrite a free chunk that is the only entry on a specific dedicated free list, they can cause the Free List bitmap to be improperly updated. This would only work for blocks smaller than or equal to 1024 bytes, and, presupposing a 1-4 byte overflow situation, the overwritten block would need to be the only entry in its dedicated free list. This attack is referred to as the **Bitmap Flipping Attack / Bitmap XOR Attack**. Moore's *Heaps about Heaps* documents this attack and credits it to Nicolas Waisman. (Moore 2008)

Controlled 16+ Byte Overflows

If you relax our pre-condition to include situations where the attacker can overwrite and control 16 or more bytes of chunk meta-data, then there are other alternative attack vectors that have been previously published.

Nicolas Waisman's bitmap flipping attack can be applied to blocks that are on populated dedicated freelists, but this requires overwriting the **FLink** and **BLink** fields with two identical pointer values that are safe to dereference. This attack, outlined in Moore's *Heaps about Heaps*, is applicable to free blocks of size <=1024 bytes.

Brett Moore has identified multiple attacks against the free list maintenance algorithms, which can also be applied in this situation. Moore's attacks should work for large block exploitation as well, making them viable alternatives to heap cache de-synchronization. Specifically, the **FreeList[0] Insert**, **FreeList[0] Searching**, and **FreeList[0] Re-linking** attacks should be applicable, though each have different technical prerequisites and trade-offs. These attacks generally require writing specific valid pointers to the **FLink** and **BLink** fields and some degree of prediction or preparation of memory that these pointers will reference. (Moore 2008)

De-synchronization

We saw that when a cache bucket is desynchronized from **FreeList[0]**, data supplied by the application can end up being interpreted as the **FLink** and **BLink** pointers of a **FreeList[0]** node. This is because the stale pointer handed to the application still points to memory that the heap cache considers to be a free block. Consequently, the first 8 bytes written into the newly allocated memory can be incorrectly interpreted as **FLink** and **BLink** pointers by the Heap Manager.

If the attacker can control what the application writes to these first 8 bytes, they can intentionally provide malicious **FLink** and **BLink** pointers. In *Heaps About Heaps*, Brett Moore documents several attacks against the Heap Manager that are predicated on corrupting **FLink** and **BLink** pointers. His attacks posit a buffer overflow being the primary cause of the corrupt pointers, but, with some subtle adjustments, we can re-apply them in this context as well.

Traversal and the Cache

Before we look at specific attacks, it's important to understand how the presence of the heap cache subtly changes the **FreeList[0]** traversal algorithms. Instead of starting at the head of **FreeList[0]** and traversing the linked list using the forward links, the Heap Manager first consults the heap cache. It will get a result from the heap cache, but depending on the context, it will either use the result directly, discard it, or use it as the starting point for future searching.

To be more specific, the allocation and linking algorithms both use the **RtlpFindEntry()** function to query the heap cache, but they use the pointer returned from the function differently. **RtlpFindEntry()** accelerates searches of **FreeList[0]** using the heap cache. **RtlpFindEntry()** is passed a size parameter, and it returns a pointer to the first free block it finds in **FreeList[0]** that is the same size or larger.

Allocation

The allocation algorithm is looking for a block on the free list that it can unlink from the list, parcel up as necessary, and return back to the application. The code will consult the heap cache with **RtlpFindEntry()** for the requested size. If the bucket for that size has an entry, **RtlpFindEntry()** will simply return it without explicitly checking its internal size in the chunk header. **RtlpFindEntry()** generally won't dereference any pointer in a bucket and check its size until it gets to the point where it has to look at the catch-all block (typically >= 8192 bytes.) It will then search through the **FreeList[0]** manually, starting at the block pointed to in the catch-all bucket.

The allocation code in **RtlAllocateHeap()** that calls **RtlpFindEntry()** looks at the block it gets back, and, if it notices that the block is too small, it changes its strategy entirely. Instead of trying to traverse the list to find a bigger block, it will just give up on the free list approach entirely, and extend the heap to service the request. This is an uncommon situation that is typically only brought about by our intentional de-synchronization, but it doesn't cause any debug messages or errors.

Linking

The linking algorithm is more amenable towards attacker manipulation. In general, what the linking code wants to do is find a block that is the same size or bigger, and use that block's **BLink** pointer to insert itself into the doubly linked list. The linking code will call **RtlpFindEntry()** in order to find a block that is the same size or greater as the one it is linking. If the linking code calls **RtlpFindEntry()** and notices that the returned block is too small, it will keep traversing the list looking for a larger block instead of giving up or signaling an error.

Insert Attack

So, if we've indirectly corrupted the **FLink** of a large block in **FreeList[0]** and it is consulted during an allocation search, there is no real harm done if we've intentionally made the size smaller than the bucket's intended contents. The allocation code will simply extend the heap and not disturb the free list or heap cache (beyond some temporary additions of blocks representing the newly committed memory.)

During linking searches, however, our malicious pointers will be further searched. So, if the application does an allocation and gets back one of our desynchronized stale pointers, and we can get it to write **FLink** and **BLink** values that we can control or predict, then we're in a relatively advantageous situation. The following diagram shows what this looks like in memory:

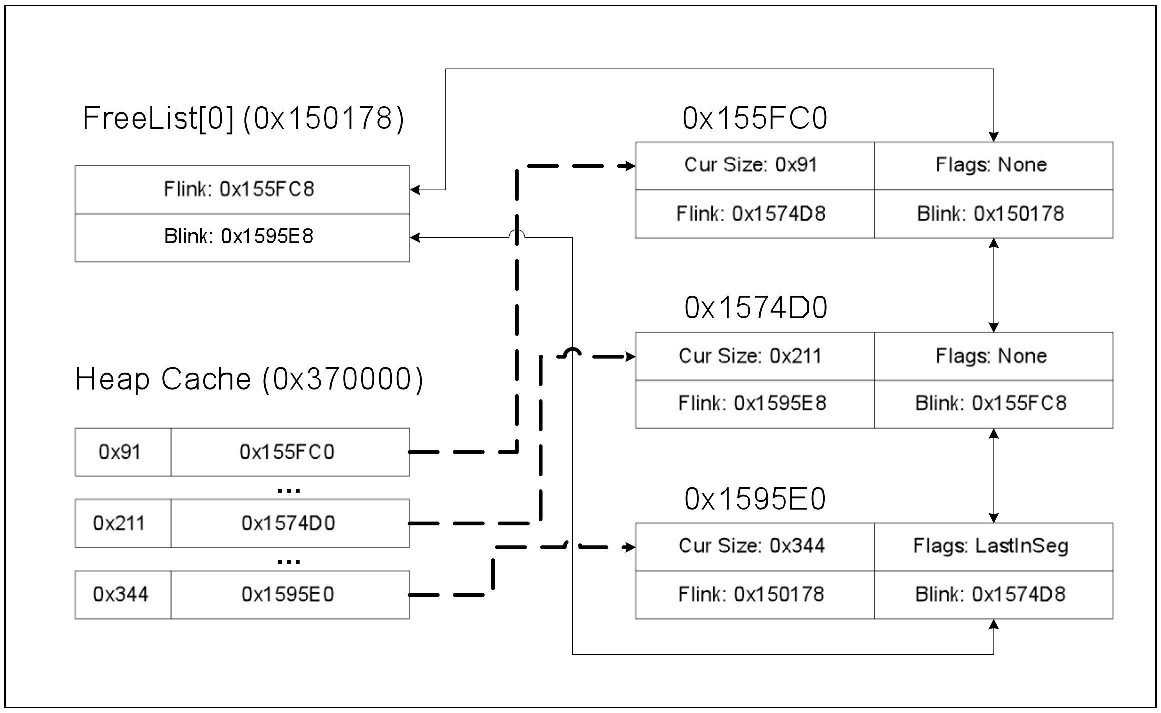

Figure 16 – Insert Attack Step 1

We've got a valid set of free blocks, in a valid **FreeList[0]**, all with entries in the heap cache. We'll do a 1-byte overflow of a NUL into the block at 0x1574D0:

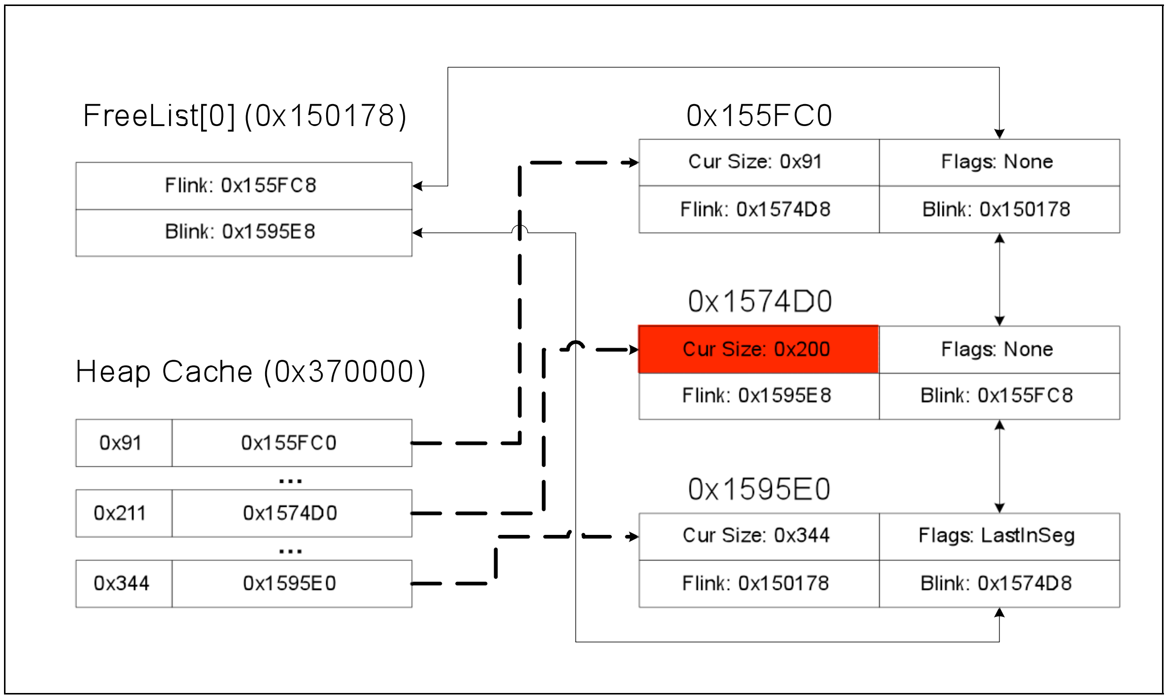

Figure 17 – Insert Attack Step 2

This does the corruption that we'd expect, slightly changing the size of the block in the 0x211 heap cache bucket. Let's assume the application allocates a 0x1FF (x8) sized buffer. This is proceeding similarly to our first attack method, but this time, we'll assume that the attacker has control over the first few bytes written into the buffer it just got back from **HeapAlloc()**.

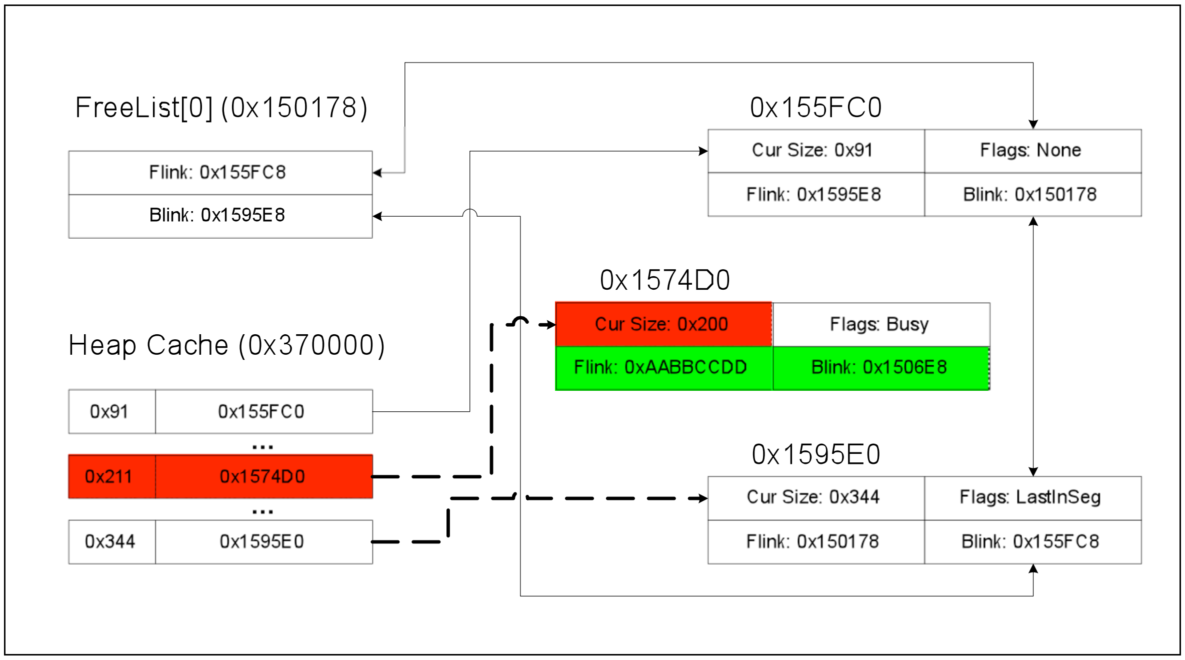

Figure 18 – Insert Attack Step 3

So, two useful things have happened. First, our corrupted block has been removed from the real valid **FreeList[0]**, as its linkage pointers were correct when the allocation occurred. Second, the heap cache entry for size 0x211 is incorrect and is pointing to a buffer that is only of size 0x200.

Our goal now is to perform an attack against unsafe linking, which, once we are at this point, parallels the **FreeList[0] Insertion** attack outlined by Brett Moore. Ideally, the next thing we'd need to happen would be for the application to free a block of a size less than 0x200, but higher than 0x91. This will cause the block being linked in to the free list to be placed right before our corrupted block, which isn't actually on the real **FreeList[0]**. For the payload of this attack, we will target a look-aside list. **BLink** has been set to 0x1506E8, which is the base of the look-aside list for block size 0x2.

(We're making a few assumptions as to the application's subsequent allocation and free behavior, but it's worth noting that the system doesn't necessarily have to free a block at this point, as an allocation that split a block and left the correct post-coalesce remainder would accomplish the same thing.)

To keep things straightforward, let's assume that the application frees a block of size 0x1f1. What will happen is the following:

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
```

The heap manager will write the address of our block to the base look-aside list pointer at 0x1506e8. This will replace any existing look-aside list with a singly-linked list of our own construction. It will look like this:

Listing 18 – Look-aside Representation

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

Thus, three allocations from the corrupted look-aside list will cause our arbitrary address, 0xAABBCCDD, to be returned to the application. That will look like the following:

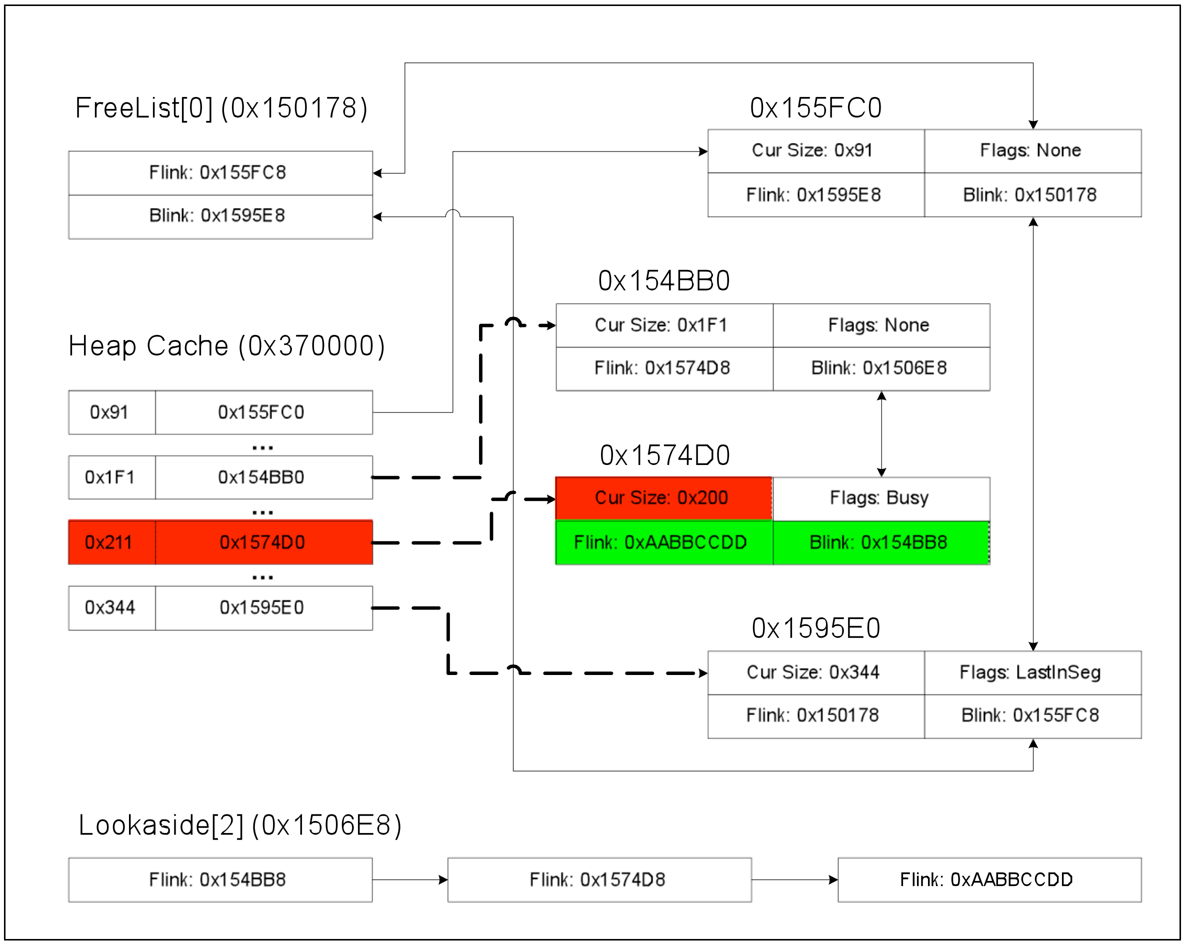

Figure 19 – Insert Attack Step 4

Summary

If the attacker can change the current size field of a block that is pointed to by the heap cache, that block won't be properly removed from the heap cache, and a stale pointer will remain. If the attacker can cause the application to allocate a buffer from that stale pointer, and the attacker can control the contents of what is stored in that buffer, he/she can provide malicious forward and back links.

This attack uses malicious forward and back links designed to overwrite the base of a look-aside list when a new block is linked in to **FreeList[0]**. It is an adaptation of the insertion attack described by Brett Moore in *Heaps about Heaps* (Moore 2008), altered to use the heap cache de-synchronization technique. The attacker causes a new block to be inserted immediately before the stale heap cache entry, which means that the new block's address will be written to the attacker-controlled **BLink** pointer. By pointing **BLink** at the base of a look-aside list, the attacker can provide their own singly-linked list, causing the attacker-supplied arbitrary **FLink** pointer to eventually be used to service an allocation.

The end-result of the attack is that the attacker can get attacker-controlled data written to an arbitrary address, by explicitly controlling the address returned to an allocation request.

Prerequisites

- The attacker must be able to write into the current size field of a block that is free and present in the heap cache.
- The attacker must be able to predict subsequent allocations that the application will make.
- The attacker must avoid allocations that cause splitting or re-linking of the corrupt block, or prepare/inherit a buffer that prevents coalescing.
- The attacker must control the contents of the first two DWORDS of what is written into the buffer allocated via the heap cache.

Existing Attacks

This attack is unique in its ability to allow for the exploitation of a 1-4 byte overflow for blocks of a size higher than 1024. Beyond this unique property and its setup using the heap cache, it can be considered to be in the same class of attacks as the insertion, searching, and re-linking attacks described by Brett Moore in his presentation *Heaps about Heaps*. (Moore 2008)

De-synchronization Size Targeting

One problem that occurs when attacking the heap cache in practice is that there is a lot of linking and unlinking traffic against the free lists in general. This activity can complicate multi-step attacks and conspire to make them probabilistic and non-deterministic.

Outside of multi-threading scenarios, one simple cause of unexpected free list activity is *block splitting*. Block splitting occurs because most larger allocation requests will not perfectly correspond in size with a free block resident in **FreeList[0]**. Instead, a free block that is overly large will be selected and then split into two blocks: a *result block*, and a *remainder block*. The result block services the allocation request from the application, so it is unlinked from **FreeList[0]**, marked as busy, and handed up to the caller. The remainder block holds the excess bytes that were unused when fulfilling the allocation request. It will have a new chunk header synthesized, be coalesced with its neighbors, and then be linked into the appropriate **FreeList[n]**.

Given some control of the application’s allocation and free behavior, there are a few ways an attacker can increase the resiliency of these attacks. We’ll briefly look at one technique, which involves creating a hole in the heap cache for a specific allocation size, and using entries to defend that hole from spurious activity.

Shadow Free Lists

The general approach for handling variance in the execution flow in a real-world program is to try and maintain a mostly innocuous, consistent heap cache. This means that most requests should end up pointing at valid **FreeList[0]** blocks, and the system should largely function correctly. For an attack targeting one particular allocation size, one can set up what is essentially a shadow **FreeList[0]** and dial in sizes that cause a specific trapdoor to be created in the heap cache. Consider the following three buckets in the heap cache:

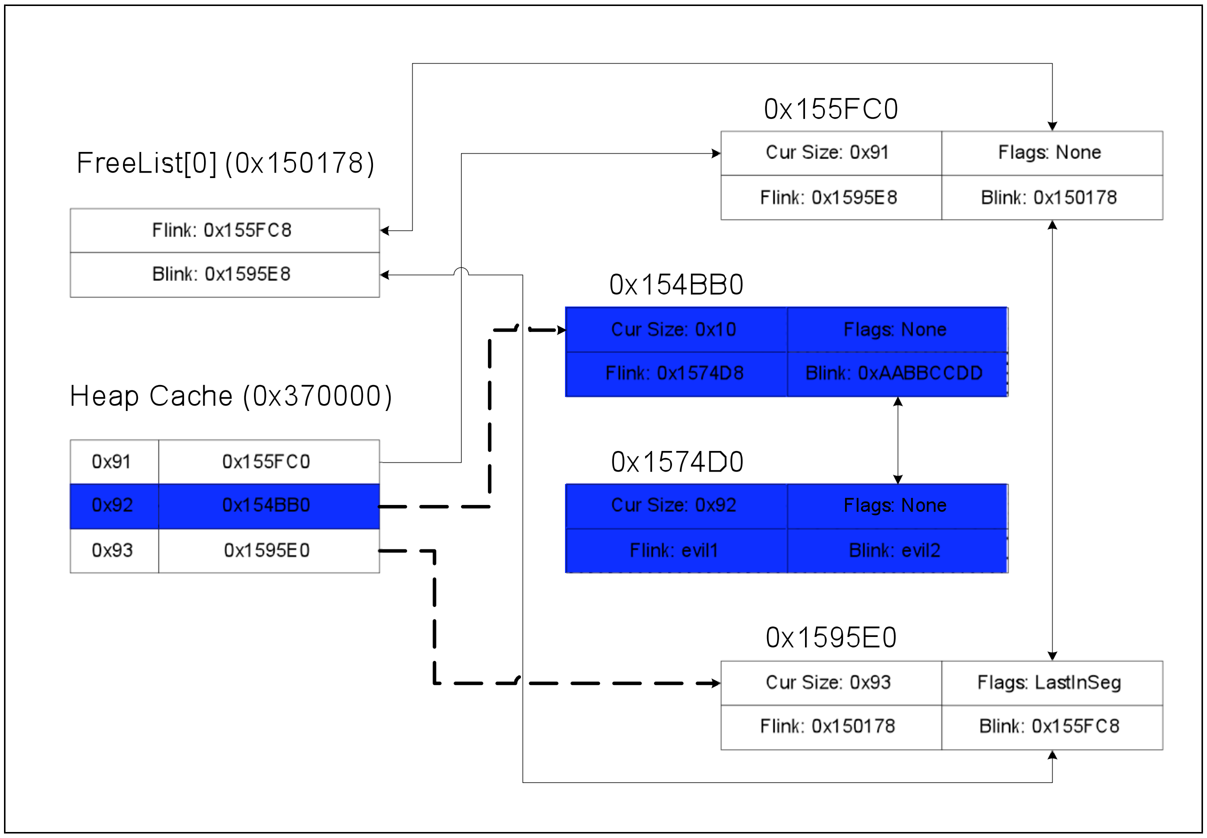

Figure 20 – De-synchronization Size Targeting

Here, we have a **FreeList[0]** with a head node and two entries (the white nodes in 0x155FC0 and 0x1595E0). These are valid and self-consistent, and synchronized with their corresponding cache bucket entries. Now, we have a stale desynchronized bucket (bucket 0x92 in the heap cache). It is pointing at the shadow **FreeList[0]**, which is logically consistent except for not having a head node.

Building such a shadow list is relatively straightforward, depending on the attacker's ability to control allocation and de-allocation. Once you do a de-synchronization and further allocation that selects the desynchronized block, you will have a stale pointer in the heap cache, but **FreeList[0]** will be valid in and of itself. The index will be wrong, but the list will still be coherent. From there, if you link new free entries by selecting the poisoned entry out of the heap cache with the linking algorithm, the inserted entries will form a shadow **FreeList[0]**. This list can only by reached through the heap cache, and won't be accessible via a normal traversal of **FreeList[0]**.

Allocation

To see why this could be useful, let's first consider allocation. Let's assume that the bucket at 0x92 is the critical size we are using to exploit the system, and we want to tightly govern which requests modify its state. If you recall, a search for an appropriately sized buffer is going to skim through the cache buckets using the bitmap for fast resolution. Here, we've defended against this somewhat by causing a valid free entry to exist in bucket 0x91. Let's consider possible activity:

- If an allocation comes in for a size <=0x91, the valid entry in bucket 0x91 will be selected and used. If the attacker arranges for multiple 0x91 entries to be in the **FreeList[0]**, they can be used as a stopgap to protect the malicious entry.
- If an allocation for 0x92 comes in, it will attempt to use the evil free list chunk, but see that its size is too small to handle the request. Consequently, it will forego the fake free lists entirely and just extend the heap and use new memory to service the allocation request. (This happens because we set the block size to a small value intentionally.)
- If an allocation for 0x93 comes in, it will use the valid free list entry in that bucket.

Linking Searches

Now, let's consider linking searches.

- If the search is for a size <=0x91, the valid free list entry in bucket 0x91 will be returned.
- If the search is for 0x93, the valid free list entry will be used, which should be innocuous.
- If the search is for exactly 0x92, the malicious free list chunk will be used. For linking, it will see that the size is too small, but then follow the malicious free list's **FLink**. From this point on, the system will be operating on the shadow free list that was provisioned entirely by the attacker. This can be used to perform the insertion/linking attacks described previously.

Malicious Cache Entry Attack

So far, we've looked at attacks centered around creating a stale pointer in the heap cache. There is a slightly different attack method, which aims to get an attacker-controlled pointer directly into the heap cache. When a valid block is removed from the heap cache, the code that updates the cache trusts the **FLink** value in the block, which can lead to exploitable conditions if the **FLink** pointer has been corrupted.

This attack is very similar to Moore's attack on **FreeList[0] Searching**, which splices the **FreeList[0]** in order to set up an exploitable situation (Moore 2008). The heap cache changes the dynamics of the situation slightly, such that an attacker can make a less-pronounced change to the data structure and alter a particular subset of **FreeList[0]**.

When the heap cache removes a block of a given size, it updates the bucket for that size with a pointer to the next appropriate block in **FreeList[0]**. If there is no such appropriate block, it sets the pointer to **NULL** and clears the associated bit in its bitmap. Normally, every possible block size between 1024 to 8192 bytes has its own bucket in the heap cache, and blocks higher than or equal to size 8192 bytes all go into the last bucket. Buckets that represent a specific size – under normal conditions – will only point to blocks of that size, and the last bucket will just point to the first block in **FreeList[0]** that is too big for the heap cache to index. The following figure shows a normal heap situation with the heap cache:

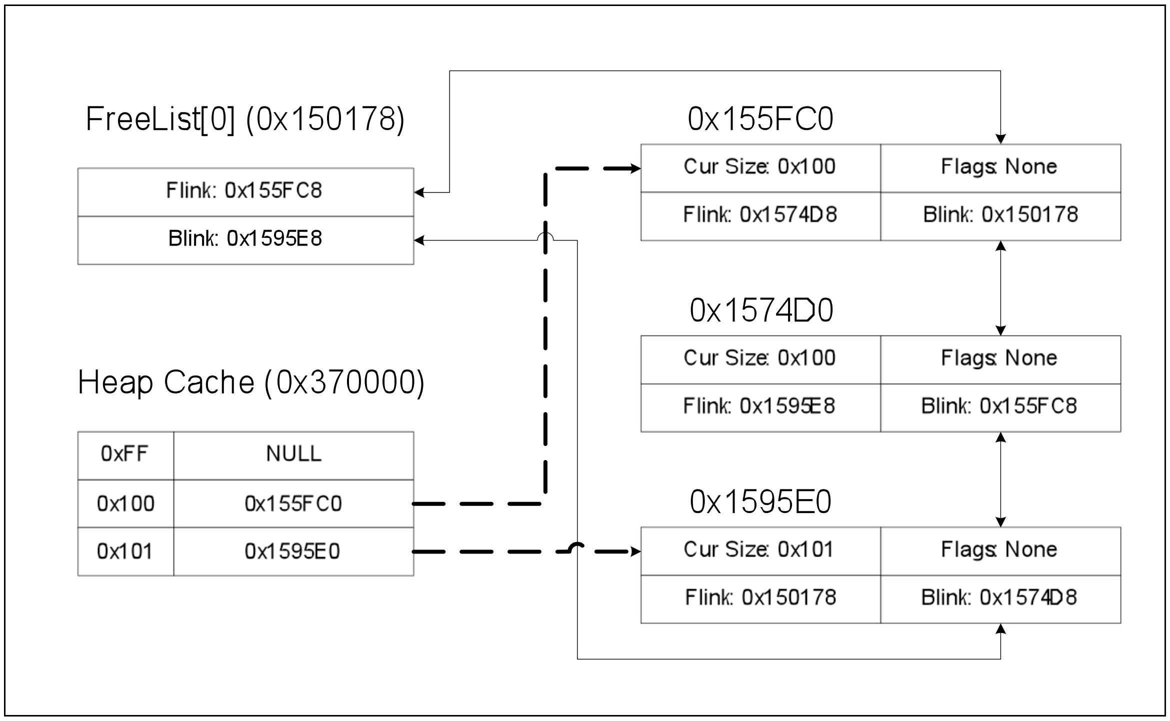

Figure 21 – Malicious Cache Entry Attack Diagram

Here, we see a small part of the heap cache, and we can see that the bucket for size 0x100 points to the block at 0x155FC0. There is a second block of size 0x100 in the free list, at 0x1574D0, which is not pointed to by the heap cache. There is also a block of size 0x101 at 0x1595E0, which is in the heap cache.

So, if block 0x155FC0 is removed from the heap cache, the bucket for size 0x100 will need to be updated. In the above situation, it will be updated to point to 0x1574D0. If 0x1574D0 was later removed from the cache, the bucket for size 0x100 would be set to NULL.

The removal algorithm works by using the **FLink** pointer of the block it is removing to find the next block in **FreeList[0]**. If that block is of the appropriate size, it sets the heap cache entry to it. For the catch-all bucket, it doesn't dereference the **FLink** pointer since it doesn't need to check that the sizes match. (It only needs to make sure it's not the very last block in **FreeList[0]**.)

So, if an attacker can provide a malicious **FLink** value through memory corruption, and this value is a valid pointer to an appropriate size word, then they can get a malicious address placed into the heap cache. In the previous attacks, we altered the size of a free chunk so that it would never be removed from the heap cache, causing stale pointers to be returned back to the application. In this attack, we are attempting to corrupt the **FLink** pointer of a free chunk, and to get our corrupt value to actually be placed into the heap cache. Once our corrupt and arbitrary value is in the heap cache for a particular size, it will be returned to the application, allowing for a controllable write to arbitrary memory.

Dedicated Bucket

For entries not in the catch-all bucket, you generally would need to predict the size of the entry you are overwriting, and provide a pointer that points to two bytes equal to that size. If you get the size wrong, the heap cache won't be updated, and you will essentially be in a desynchronized state similar to the initial stages of the first attacks we outlined. However, let's assume that we can predict the target block's size with some regularity. For example, say you are overwriting a chunk with the following values:

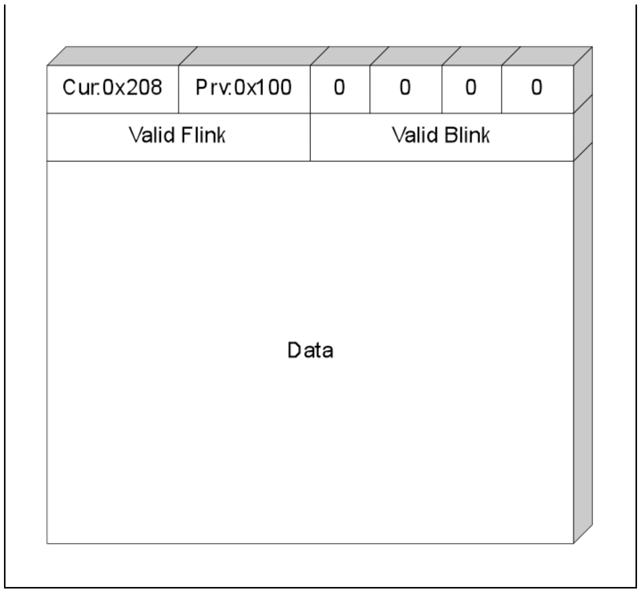

Figure 22 – Malicious Cache Entry Attack Dedicated Bucket Step 1

Assume that the attacker knows the size of the chunk that is being corrupted, and does the following overwrite:

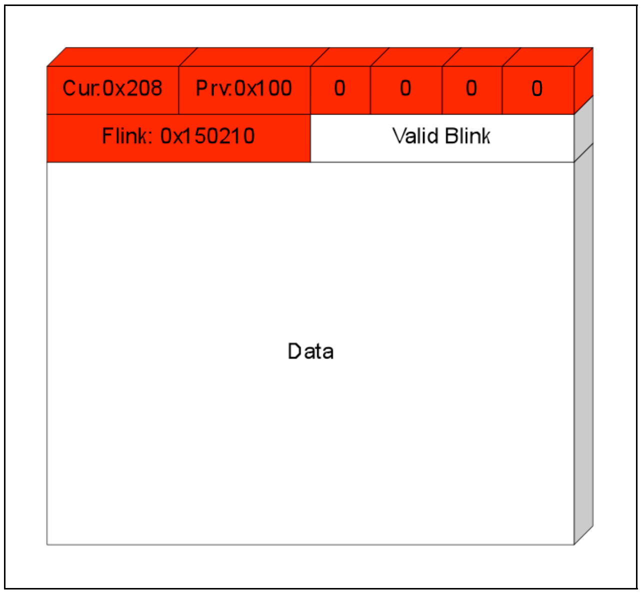

Figure 23 – Malicious Cache Entry Attack Dedicated Bucket Step 2

Essentially, the attacker didn't change anything beyond pointing the **FLink** at a free list head node at the base of the heap. This takes advantage of the situation that the attacker knows that an empty free list head node will point at itself, thus the "block" at 0x150208 will be interpreted as the following:

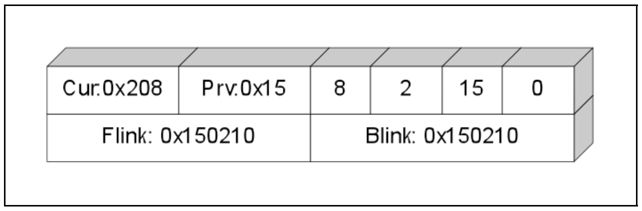

Figure 24 – Malicious Cache Entry Attack Dedicated Bucket Block

Now, the attacker would cause the application to allocate memory until the poisoned value 0x150210 was in the heap cache entry for size 0x208. Note that the size of the corrupt block being freed is 0x208, and the size of the block at its **FLink** pointer, 0x150208 is 0x208. Thus, when the corrupted block is removed from the heap cache, it will pass the size check, and the heap cache will be updated to point to 0x150208.

The next allocation for block size 0x208 would cause 0x150210 to be returned to the application, which would allow the attacker to potentially overwrite several heap header data structures. The simplest target would be the commit function pointer at 0x15057c, which would be called the next time the heap was extended.

Catch-all Bucket

It isn't necessary to predict the sizes when attacking a block in the catch-all block, which, by default, contains any block larger than or equal to 8192 bytes in size. Here, the primary requirement is to ensure that the blocks of size greater than or equal to 8192 bytes -- yet less than the attack size you choose -- are allocated before your overwritten block. This will ensure that your entry will make it into the heap cache for the last bucket entry, and the next large allocation should return the address you provide. For example, if you overwrote the following chunk:

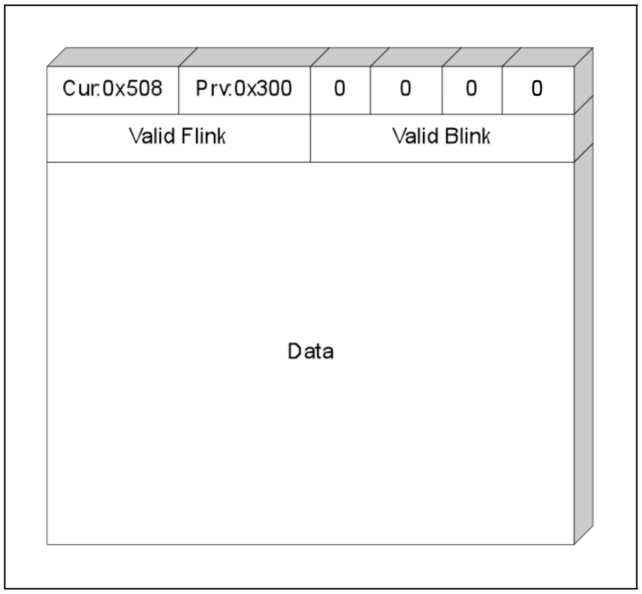

Figure 25 – Malicious Cache Entry Attack Catch-all Bucket Step 1

And you supplied these values:

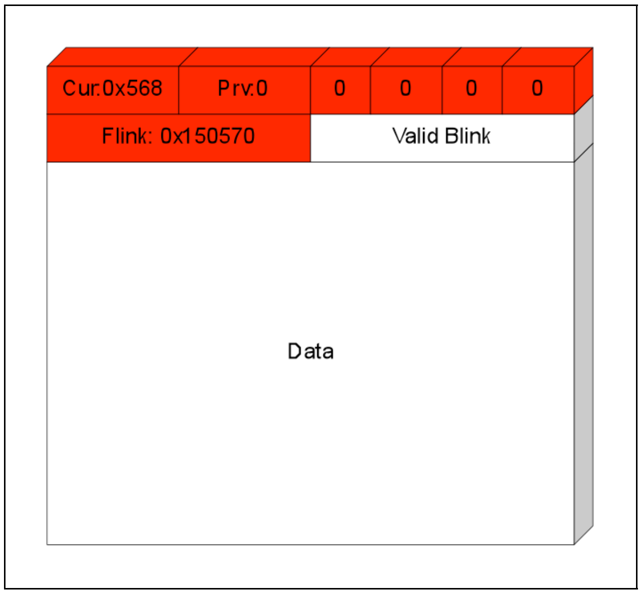

Figure 26 – Malicious Cache Entry Attack Catch-all Bucket Step 2

Assuming that you could handle coalescing by fortuitous **BUSY** flags or other planning, and every block of size >=0x400 (8192/3) was allocated before your block, your poisoned **FLink** of 0x150570 would be promoted to the entry in the cache bucket. Then, the next allocation between 8192 and 11072 bytes would return 0x150578, allowing you to potentially cause the application to write to 0x15057c and corrupt the commit function pointer. The size will be checked by **RtlAllocateHeap()**, which will interpret the block contents as:

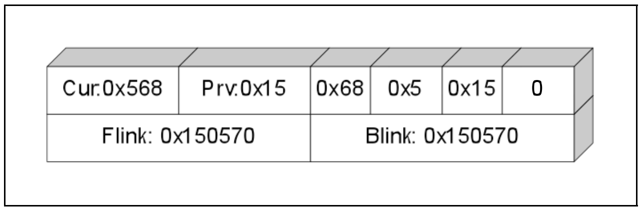

Figure 27 – Malicious Cache Entry Attack Catch-all Bucket Block

Summary

If the attacker can overwrite the **FLink** pointer of a large block that is in **FreeList[0]**, the corrupted value can eventually be propagated directly to the heap cache entry itself. When the application next attempts to allocate a block of that size, it will get an attacker controlled pointer instead of a safe piece of memory.

Prerequisites

- The attacker must be able to overwrite the **FLink** pointer of a free block.
- The attacker must be able to cause allocations to occur that promote this allocation to the heap cache.
- The application must make a predictable allocation that can be targeted by corrupting a heap cache entry.

Bitmap XOR Attack

We previously discussed a theoretical **Bitmap Attack** earlier in the paper. If there were to be a scenario where you could increment an arbitrary **DWORD**, you could attack the **FreeListInUseBitmap** and trick the memory manager into believing there were free chunks in a **FreeList** when was not rightfully so.

So, directly altering the bitmap is an interesting and useful attack primitive, but it's not a generalized heap technique. This is because in the majority of memory corruption vulnerabilities, you don't get to pick exactly where your corruption will occur. So, you generally won't be able to easily reach and modify the bitmap. However, it turns out that we can use various implementation flaws in the Heap Manager to cause it to corrupt its own bitmap. These were published in Moore's *Heaps about Heaps*, and credited to Nicolas Waisman.

Let's look at some pseudo-code to see how the **FreeListInUseBitmap** is actually used to locate a sufficient free chunk:

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);
}
```

The code above assumes that we have found a bit in the bitmap and that the **FreeList** has been set to the appropriate bucket. It then proceeds to safe-unlink the node from the list and XOR the bitmap if this was the last entry in the list. That is, the **FreeListInUseBitmap** should be cleared if the list is empty. Unfortunately / fortunately there are some problems with the code above.

- The first problem is that the algorithm checks to see if ‘next == prev’ to determine if the FreeList is empty. This is an easy condition to fake if you have a 16-byte overwrite and will still continue executing regardless of the safe-unlinking failing.

- The second problem is that the code takes the size from the chunk retrieved from **FreeList[n]** and uses it as an index when updating the **FreeListInUseBitmap** (**vSize = vent->Size;**). The problem with this is that, just like the ‘next’ and ‘prev’ values, the size can also be forged when performing an overwrite of the heap chunk metadata. The can lead to a de-synchronized state where the memory manager believes a **FreeList** entry to be populated / unpopulated when it is not the case.

- The third error is that instead of directly setting the **FreeListInUseBitmap** to 0 when **FreeList[n]** is empty, it performs an XOR operation. This means we can toggle arbitrary bits if we overwrite a chunk’s size. This can permit us to setup circumstances that are similar to the attack that Nico hypothesized as previously discussed.

- The fourth and final error is that the size is not verified to be below 0x80 before indexing into the **FreeListInUseBitmap**. This means that we can toggle bits in semi-arbitrary locations past the **FreeListInUseBitmap**. This could be quite useful because the bitmap is located in the **Heap Base**.

As a result of all these errors, a one-byte heap overflow can turn into an exploitable condition, as long the free chunk being overflowed is of a size that is less than 1024 bytes and the last free chunk of its size on its **FreeList**.

Secondly, if a full 16-byte overflow can occur, the ‘prev’ and ‘next’ pointers can be made to be the same value, resulting in an XOR of the **FreeListInUseBitmap** regardless if the **FreeList** bucket is empty or not.

Lastly is the issue of the code not checking if the heap chunk’s size is less than 0x80. This can result in the attacker specifying a size greater than 0x80, hence toggling bits at a semi-arbitrary location past the **FreeListInUseBitmap** on the **Heap Base**. This is limited by the width of the size (SHORT) and the operation on that size. Essentially this means that we can toggle any of the bits from 0x150158 to 0x152157.

Avoiding Crashes

Although it may seem trivial for exploitation to occur, it is not the case. There are many problems associated with having the memory manager treat arbitrary locations as valid heap chunks.

The first problem is ensuring that the 8-bytes before the memory being referenced by the memory manager are interpreted as a 'valid' chunk header. For example, the **FLink** and **BLink** of the address must be readable address.

The second and most common (read: painful experience) is block splitting. For example, say that **FreeList[3]** is empty and you have just flipped a bit in the bitmap to show that **FreeList[4]** is populated when it is actually empty:

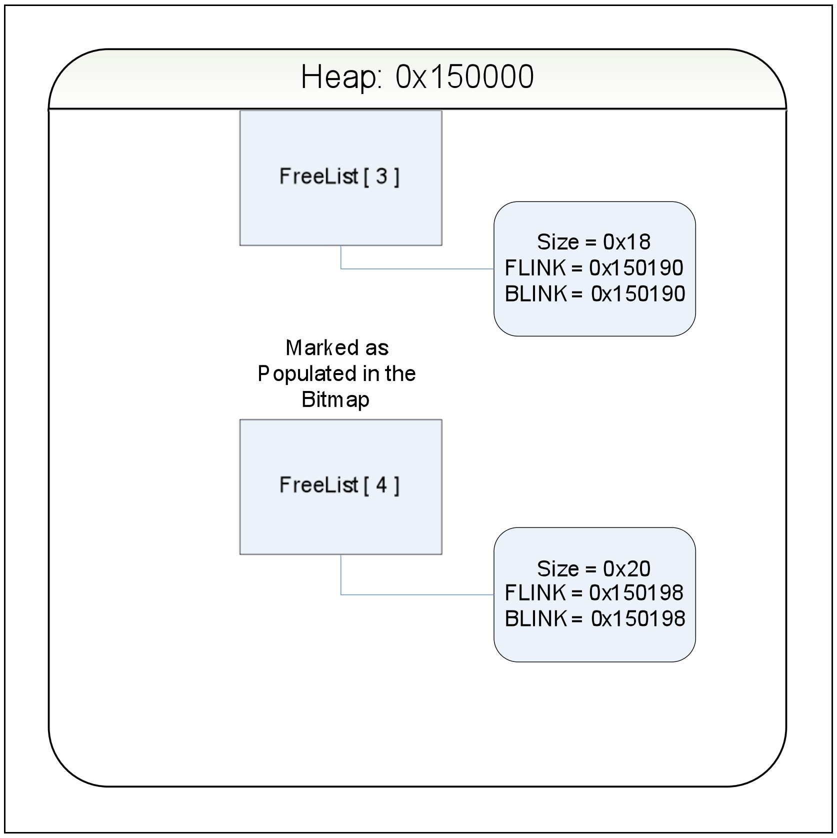

Figure 28 – Avoiding Crashes

Assume that the **Lookaside List** is empty. If a request came in for 16 bytes (24 total with the 8-byte chunk header), the memory manager would see that there are no free chunks in **FreeList[3]** and proceed to check the **FreeListInUseBitmap**, finding the bit set for **FreeList[4]**. Since it will treat the 8-bytes (**FLink** / **BLink**) as a valid heap chunk header, it will have the following values:

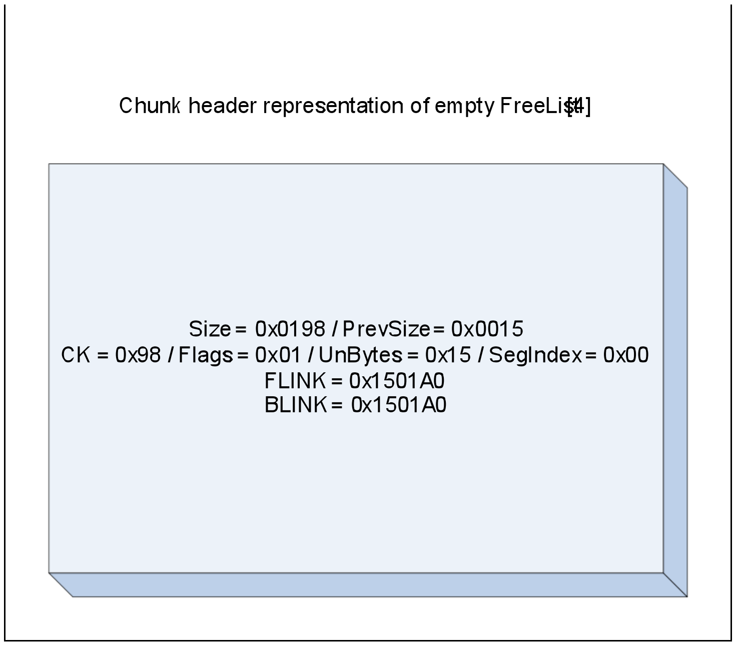
Chunk header representation of empty FreeList[4]
Size = 0x0198 / PrevSize = 0x0015
CK = 0x98 / Flags = 0x01 / UnBytes = 0x15 / SegIndex = 0x00
FLINK = 0x1501A0
BLINK = 0x1501A0

Figure 29 – Avoiding Crashes II

When the request for 24 bytes arrives, it will be fulfilled by the 'free' block in an empty **FreeList[4]**. The memory manager will subtract 24 bytes from the size and see that there are more than 8 bytes remaining, leading to a split of the memory. This gives the remaining chunk a size of 0x0195, which would reference 0x195 x 8 bytes down the heap to see to the next chunk (which isn't actually a real heap chunk). This can lead to access violations when attempting to dereference addresses that aren't readable. A great way to prevent block splitting would be to ensure that the flags are set to 0x10, making it the last entry.

Lookaside List Exception Handler

The LAL has an interesting behavior that may prove useful in the context of further exploitation. Specifically, it is wrapped by a catch-call exception handler when it performs dereferencing of the linkage pointer in the singly linked list. This means if the **FLink** pointer in an LAL list is invalid, and causes an access violation when it is de-referenced, the system will gracefully handle the condition, falling back to the back-end manager without printing a warning.

Let's look at some pseudo-code for LAL allocation:

Listing 20 – LAL Allocation

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

  return result;
}
```

Now we can look at the pseudo-code for *ExInterlockedPopEntrySList*:

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))
}
```

You can see that dereferencing of the **Lookaside List** head is performed in the ExInterlockedPopEntrySList function. If the **FLink** pointer is a non-readable address an access voiloation will occur. This condition would be caught by the catch-all exception handler setup in **RtlpAllocateFromHeapLookaside**, which could prove useful for guessing addresses. An attacker could brute force readable addresses to overwrite (i.e TIB/PEB, thread stack, heap address guessing).

Here is a theoretical example (**warning**: potentially huge bowl of strawberry pudding). Imagine an attacker could massage the LAL into something that resembles the following:

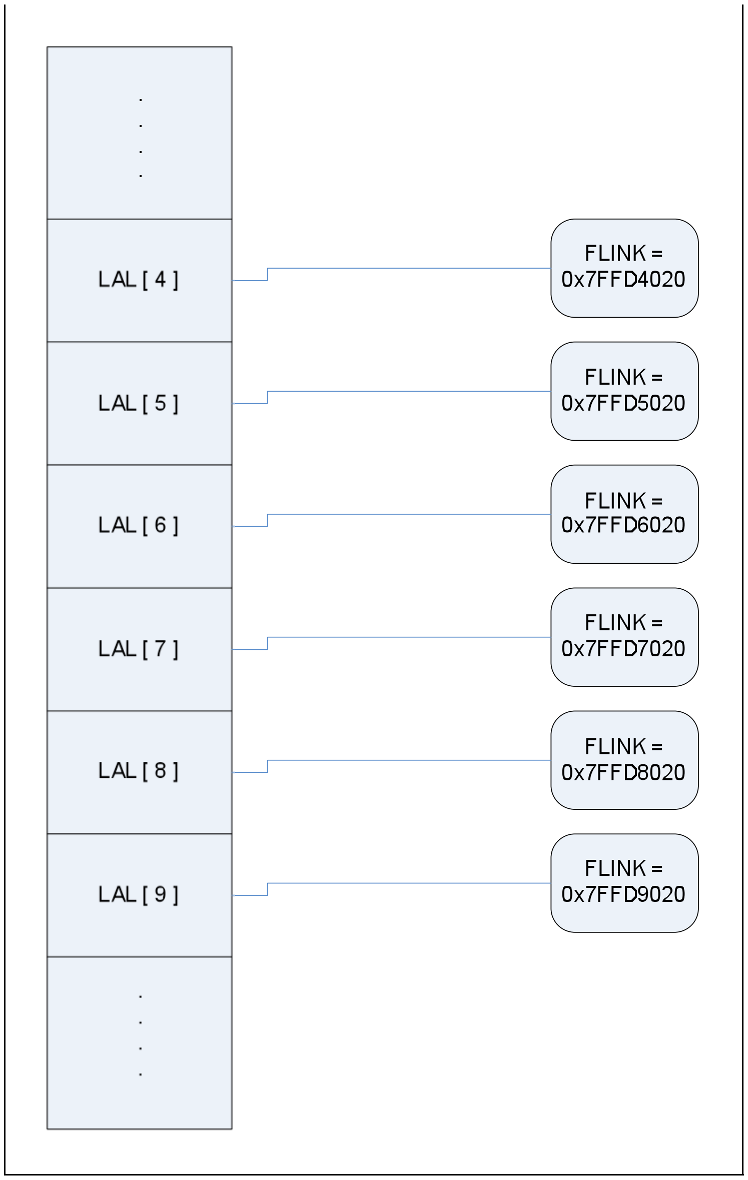

Figure 30 – LAL Exploitation

We will also assume that an attacker can control allocations and the source of the data copied:

Listing 22 – LAL Example

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

In this case, the attacker can attempt to use the values on the LAL to overwrite values in the PEB. If the address is not readable the exception handler will handle the error and the allocation will be serviced by the back-end allocator, resulting in a valid write. If the address is readable the LAL will return the address and the attacker will have an opportunity to overwrite the address he/she was attempting to guess. This technique can also be used to bypass a heavily populated LAL if using the back-end allocator is more desirable.

Note: Auditors should always look into the reasoning and functionality behind exception handlers, especially when they are designed to catch all exceptions.

Strategy

Heap exploitation is hard, even if you're good at it.

Now, it's not always hard. Sometimes things work out swimmingly, and the planets align in your favor, and you wonder why every one makes such a big deal about it. Secretly, you probably even suspect this is yet another data point that you are a genius of unparalleled technical ability, who is capable of pulling off amazing feats of right-brained intuition that would make Einstein cry at the mere beauty of their elegance, assuming you could simplify it enough such that his merely normal-genius level mind could comprehend it.

Or maybe that's just Chris.

At any rate, eventually you'll run into one that is properly hard. Hopefully, this section will help you think of ways to make forward progress if you find yourself flailing.

Application or Heap-Meta?

So, first, let's talk about meta-data exploitation vs. application data exploitation.

Meta-data exploitation is where you attack the internal data structures used by the Heap Manager itself in order to gain control of the target process. We've looked at quite a few different tricks and tactics for pulling this off.

One of the primary advantages of this approach is that you can often be very certain – a priori – of where heap meta-data will be relative to the memory corruption. You also have a good idea what it will contain, and what data structures can be simulated by repurposing the Heap Manager's pointers, arrays, and lists. One key downside, of course, is that heap meta-data is increasingly hardened against attack.

Application data exploitation is where you target the data in the actual heap buffers, which can involve necessarily trashing in-line heap meta-data. This has a parsimonious appeal, and we've personally gone back and forth on preferring this technique in a vacuum. It is particularly effective if you can isolate corruption within the same heap chunk or into a neighboring chunk that is naturally allocated with a target block.

Ultimately, you don't really have to choose which one you think is the "one true way," as the approaches are meeting in the middle in practice. Heap meta-data is getting hardened over time, requiring the coordination of multiple heap blocks in order to set up pathological conditions. Application data overwrites targeting a second or third block will require essentially the same thought processes and tactics in order to set up vulnerable situations.

Multiple Dimensions

Let’s consider one quick observation before we start discussing strategy:

There are multiple dimensions on which we need to conceptualize heap state when we try to think of it abstractly.

First, we have the contiguous memory layout to consider. These are tracked primarily with segments and UCR entries. The system keeps track of which chunk is last in a particular segment, and sets a specific flag in the chunk (0x10). The Heap Manager also maintains a pointer at the base of the heap to the last chunk in the segment.

Second, we have the data structures that model relations between buffers to consider. These depend on the attack technique being used, but can comprise the LAL, LFH, **FreeList[n]**, **FreeList[0]**, relevant bitmaps, and heap cache.

Note: This last chunk pointer update may be useful in furthering corruption as part of the second-stage of an attack. (Conover and Horovitz 2004)

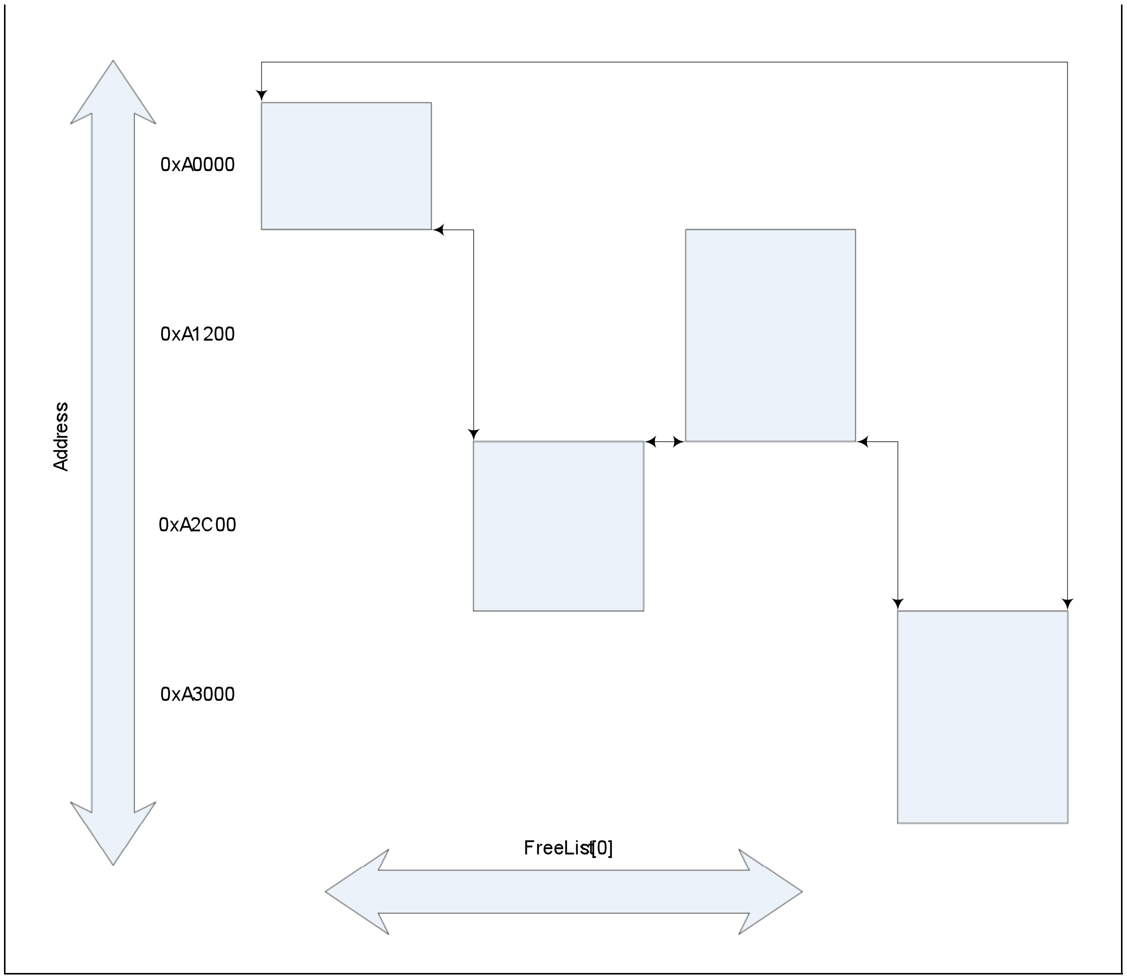

Figure 31 – Two dimensional Visualization of Heap

Determinism

Heap Spraying

Heap spraying was originally developed by Blazed and Skylined for use in browser exploits. (Sotirov 2007) It is a technique that involves using JavaScript to fill memory with a large, attacker-provided payload. This is performed with JavaScript strings or arrays, though it could be performed with other primitives or browser functionality. The string with the payload is repeated many times in memory to increase the probability that a given address range contains the attacker controlled data. After this is performed, the attacker can guess a valid return address, which can be used when corrupting a function (or object) pointer as part of a vulnerability. The guessed address is in the middle of the large heap grown heap, and will very likely contain the NOP sled and shell-code. Let’s look at an example:

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;
}
```

The code above creates a large NOP sled and proceeds to concatenate the shell-code. This is done so that the JavaScript string will allocate new memory. It then repeats this string by storing it in an array **num_of_blocks** times. The attacker would then attempt to overwrite a return address with **0x0c0c0c0c** and hope that the payload is present at that address. This technique involves certain trade-offs as far as reliability vs. memory utilization, but it’s definitely not one to discard lightly as it’s extremely useful.

The essence of this strategy is to tackle non-determinism by growing the heap such that there is a very high likelihood that the contents will be predictable and useful.

Heap Feng Shui

Heap Feng Shui is a library of techniques created by Alexander Sotirov to control heap determinism under Internet Explorer using JavaScript. (Sotirov 2007) It was originally presented at BlackHat Europe in 2007. Alex explored how Internet Explorer allocates memory for JavaScript strings off of the system heap, which involved a specific allocator wrapper, OLEAUT32.

Alex's techniques were highly tied to the idiosyncratic allocator wrapper, but he ultimately isolated the following low-level atomic actions:

- Allocate Buffer of Arbitrary Size w/ Arbitrary Content

  String Allocations in JS

- Free Buffer of Arbitrary Size w/ Arbitrary Content

  Intentionally invoke JS Garbage Collection

- Programmatic Control of Allocations and Frees

  Ah, isn't browser exploitation awesome?

From these basic idioms, he tackled the difficulties introduced by the allocation wrapper and developed:

- The Plunger Technique

  Works around presence of a 6-block intermediate cache

The next layer of functionality in Alex's code targets the Windows Heap Manager functionality:

- freelist()

  Used to add blocks to a freelist, making sure they don't get placed on the LAL and aren't coalesced.

- lookaside()

  Used to add blocks to a lookaside list.

- lookasideAddr()

  returns the address of an LAL head pointer (heapbase + X)

- vtable()

  Sets up a fake vtable that contains shellcode in memory

He further details the following process that ties together these tools:

- Defragment the heap

  Perform a large number of allocations in order to normalize the heap state

- Put blocks on free list

  Alex discusses this in the context of an un-initialized data flaw, but it is essentially his idiom for fixing a block in both the contiguous dimension and the logical dimension (the free list)

- Empty lookaside

  Empty out a look-aside entry that we will use to construct a specific shadow c++ object/vtable data structure

- Free to the lookaside

  Utilize LAL singly-linked list to double as a malicious object pointer / vtable.

Browser exploitation is an interesting situation, as you have programmatic control over allocation, which generally gives you a significant advantage. We'll build on some the same building blocks that Alex choose, but first, let's study the works of the formidable Nicolas Waisman.

Memory Leaks

There are two kinds of memory leaks that Nicolas Waisman isolates as part of his process for heap exploitation. (Waisman 2008). *Hard memory leaks* are indefinite memory leaks, in the same sense of the classic programming term, as most developers understand it. *Soft memory leaks* are those where the buffer is eventually correctly freed, but in the interim, the attacker has a degree of influence over its life-cycle. A good example of a soft memory leak would be a buffer that is allocated when a connection is opened and freed when a connection is closed. If the attacker can keep multiple simultaneous connections open, this can be a very useful primitive, especially if the buffer is freed immediately when the connection is closed.

The essential observation is that we are interested in predicting and controlling buffer life-cycles. A buffer with an interminable life-cycle, or a relatively very long life-cycle, is useful for tying up free buffers and LAL/free list entries with data to prevent them from interfering with our attempts to pre-destine buffers (either in contiguous memory or logically.)

A buffer with a short life-cycle that we can control is useful for causing actions to occur with subtle timing, or to create patterns in memory when its use is interspersed with the invocation of a buffer with a longer life-cycle.

General Process

We present a general process here, which is comprised of the following steps:

1. State of Nature – Get your bearings in a process post-corruption

2. Action Correlation – Correlate user actions to allocation behavior

3. Heap Normalization – Normalize the heap to predictable state

4. Fixing in Contiguous Memory – Create necessary holes in memory

5. Fixing in Logical Lists – Create necessary logical relationships

6. Corruption – Invoke the attack

7. Exploitation – Move from immediate corruption to code execution

This process is useful for both attacking application-specific data and attacking heap meta-data.

1. State of Nature

The first thing that you want to do is get your bearings. Examine the process virtual address space, and try to get a feel for how the heaps are being used. Essentially, you want to have a general idea as to the state of the process when your attacker supplied data is introduced.

Questions:

- Is the Heap Cache likely to be invoked already?
- Is there a LAL on the Heap you are corrupting?
- Is there an LFH?
- How populated is the LAL?
- How about the free lists?

If it’s a long-running process that handles user requests, chances are that you won’t be able to make too many claims as to its state at any given moment. If it’s a new process, then you can probably presuppose quite a bit more about the state of the system.

2. Action Correlation

In order to make progress against difficult heap vulnerabilities, you need to do a fair bit of studying of the process.

First, make the assumption that the corruption will have to happen on the same heap. i.e. you aren’t planning to jump across heap segments. (This assumption is just to make the task easier, but you should by all means discard it if you can successfully attack the process at the virtual memory/segment level.)

So, you want to study the exposed attack surface for interactions with the heap. This is an iterative process where you find an action that may be useful, and then write the necessary code and infrastructure to leverage that action.

If you get beyond where you are looking at the immediate corruption and are starting to think about priming the heap with data structures or patterns, then you can look at the other heaps to see if there’s a more straightforward mechanism to spray or prime the heap.

Pay close attention to chunk life-cycles. Two things that will prove extremely useful are soft and hard memory leaks. This is a good place to start. After that, you can look for other atomic actions that may prove useful, or start working on wrangling the heap. Here are some ideas for actions to isolate:

- You want to be able to have a permanent or long-living memory leak that you can use to allocate buffers of an arbitrary size.
- You want to be able to allocate a buffer where you control the contents of the first X bytes of that buffer, with X being between 4 and 32
- Short-life memory leaks are quite useful as well, as you can (ideally) use these to time allocations and frees.
- The ability to free a buffer of an arbitrary size at an arbitrary time is useful for de-synchronization attacks.
- Allocations where you control the size are useful for heap normalization and hole creation, especially if you can find exposed routines that have very different buffer lifetimes.
- An information leak, naturally, can tell you all kinds of useful data.
- Targets! You probably need something good to corrupt, possibly even at a specific offset of something already on the heap. So, naturally, you should study the process to find function pointers and other primitives that lend themselves towards seizing arbitrary code.

When frustrated, it’s useful to sometimes take a break and spend time refining or looking for new atomic actions, as they might prove useful later in exploitation. We generally tend to do this analysis statically, but dynamic analysis can be just as effective. (if not more-so.) One of the difficulties with this is having visibility into the code. The following tools may prove useful:

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

If there are numerous allocations and frees that happen as a result of one user action, it can be useful to look for the period of the function. For example, if one user action causes 20 heap chunks to be manipulated, it’s a good idea to fully log the heap activity (thread IDs and call stacks are extra bonus information.) Then, lay out the chunks on a spreadsheet, with each row having 20 allocations / frees. You can then correlate the actions to the attacker-provided input, and look for things like buffer lengths and/or buffer contents being derived from these inputs.

3. Heap Normalization

Heap normalization is getting the heap from a relatively unknown state into a predictable, manageable state.

Once you can do this with any regularity, you have a reasonable position from which to work forward. Too often with heap exploitation, you end up fully developing an attack vector that you can only initially trigger with 25% or 20% reliability. It is easy to go down this road, as you can make tangible progress even though you’re locking in uncertainty at the beginning.

It’s always useful to take a step back, and see if you can figure out mechanisms for first getting the heap into a predictable state. Then, from that position, manipulating the heap for further subterfuge is far less frustrating.

So, one possible goal for a predictable heap is one where there are nearly no available free blocks.

- **LAL –** Multiple allocations of the same size can empty a LAL bucket. Be careful though, as if they are soft memory leaks and you use a solid pattern for the allocations, you could end up freeing a whole bunch more chunks to the LAL when your connection closes or if the exploit fails gracefully.

- **FreeList[n] –** Again, multiple allocations of the same size can empty a **FreeList[n]**. This gets a little trickier, as you end up bouncing off of the Heap Extend size as it grows heaps to service your request.

- **Heap Cache –** The Heap Cache can be a great ally when exploiting heap vulnerabilities. First, you may need to perform several allocations and frees in order to demonstrably trigger the runtime heuristics. Once the heap cache is in place, you can use similar techniques as shown above to try and clean out the entries for a certain range of buckets. Again, this can be somewhat subtle when you start bouncing off the Heap Extend size with relinking.

Let’s look at a couple of patterns that may prove useful for heap normalization. The first one is for creating a hole in a normalized heap:

Listing 24 – Hole Creation

```
void create_hole(int size)
{
       hardmemleak_alloc(size);
       softmemleak_alloc(“holeB”, size);
       hardmemleak_alloc(size);

       softmemleak_free(“holeB”);
}
```

This is a pattern to empty an LAL bucket with hard memory leaks:

Listing 25 – Empty LAL Bucket

```
void empty_lal_bucket(int size)
{
       int j;

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

This is a pattern to fill an LAL bucket with soft memory leaks:

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”);
       }
}
```

This is a pattern to empty the LAL with soft memory leaks:

Listing 27 – Empty LAL

```
void empty_lal(void)
{
        int i;

        for (i=1024-8; i>=16; i-=8)
                empty_lal_bucket(i);
}
```

This is a pattern to fill the LAL with soft memory leaks:

Listing 28 – Fill LAL

```
void fill_lal(void)
{
        int i;

        for (i=1024-8; i>=16; i-=8)
                fill_lal_bucket(i);
}
```

This is a pattern to empty the FreeList[n] with hard memory leaks:

Listing 29 – Empty Freelist[n]

```
void empty_freelist(int size)
{
        int j;

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

This is a pattern to empty the free lists with hard memory leaks:

Listing 30 – Empty FreeList[]

```
void empty_freelists(void)
{
        int i;

        for (i=1024-8; i>=16; i-=8)
                empty_freelist(i);
}
```

For normalizing layout:

Listing 31 – Normalization Pattern

```
empty_lal();
empty_freelist();
fill_lal();
```

4. Fixing in Contiguous Memory

Use the above patterns to leave holes in memory, and work with block splitting and block coalescing.

5. Fixing in Logical Lists

Time your requests in order to get a sequence of chunks in a logical data structure to be predictable and/or exploitable.

6. Corruption

Memory corruption for modern vulnerabilities is usually fairly idiosyncratic. Writing a fixed value at a strange offset or even adding a fixed value to a few bytes past the semantic end of a piece of memory are relatively common situations these days. (This may be somewhat dependent on the types of vulnerabilities for which you and your colleagues are auditing / researching.)

So, the first rule of memory corruption on the heap is “Yes, it is totally, absolutely, definitely, almost certainly, completely exploitable. There are so many ways I can indirectly influence the heap state that the burden is on me to figure out how to succeed.”

The second rule is that “Ok, we may have overstated the case in rule 1, as this is really hard. It’s ok to fake it and/or hope for luck.” It’s ok to brute force it or fuzz some things and try to discover new behaviors. You don’t have to tell anyone about any fortuitous accidents after the exploit is written.

The third rule, which we may have stolen from Kurt Vonnegut, is “When in doubt, castle.” When you’re at a loss, do something non-linear. Go look for new atomic actions or try a different approach. Try and divide the problem into smaller steps, and see if you can gain some ground.

Nicolas Waisman, in his speech about writing exploits in the face of modern countermeasures, has a brilliant distillation of the experience of writing exploits circa 2008 (Waisman 2008). Here’s Nico’s timeline for a heap vulnerability in Windows 2003/XP SP2:

Listing 32 – Nico’s Timeline

1 day:	Triggering the bug
1-2 days:	Understanding the heap layout
2-5 days:	Finding Soft and Hard Memleaks
10-30 days:	Overwriting a Lookaside Chunk
1-2 days:	Getting burned out, crying like a baby, trying to quit, doing group therapy
2-5 days:	Finding a Function pointer
1-2 days:	Shellcode

This isn’t a bad roadmap, depending on if things cut in your favor and your heap experience, debugging, and reversing skills.

Note: Group therapy probably won’t help much unless Nico is in your group.

7. Exploitation

The LAL, if present, can be a very useful device for extending a limited form of corruption into a write-n bytes to an arbitrary location primitive. The basic idea behind the attack is to use safe-linking to overwrite a LAL head with a list comprised of pointers under the attacker’s purview. If the attacker can then intentionally plumb the corrupted LAL, he/she can come pretty close to making arbitrary changes to a processes run-time state.

It’s worth keeping in mind that if an invalid address is placed on a LAL head, the system will gracefully handle the situation and not crash due to exception handling behavior. This probably can prove useful for address guessing or potentially as part of a memory leak.

The Commit function pointer at the base of the heap is another useful device that is quite useful when performing attacks against the **FreeList[]** or heap cache. Often times, you can get a pointer to the base of the heap to be returned to the application through subtle de-synchronization of heap data structures. The commit function pointer is at base +0x57c, and it is called when the heap is extended. You can usually trigger this just by doing a large single allocation.

Note: When targeting the commit routine function pointer, you often must necessarily trample the critical section **LockVariable**, which is used to synchronize the heap. You’ll need to provide a valid pointer here to survive until the commit routine pointer is called. (check **NULL**)

Conclusion

Windows heap exploitation has necessarily evolved in sophistication over the last decade, commensurate with the increasingly hardened state of the Windows OS and the non-deterministic nature of modern concurrent software. We have revisited the best-of-breed published tactics, tracing a narrative from Microsoft's mitigations in Windows XP SP2 to the present XP SP3 and Server 2003 Heap Manager. Our contribution to the discussion is a handful of specific technical attacks, which can significantly change the risk profile of certain classes of memory corruption vulnerabilities. Beyond our specific attacks and counter-measures, the take-away from our research is ultimately something that most security professionals know intuitively: it's better to err on the side of caution when evaluating the exploitability of memory corruption flaws. In short, the heap is such a fascinatingly rich and intricate system that even the bleakest, most-constrained memory corruption vulnerabilities may eventually prove to be reliably exploitable by an attacker with sufficient resources.

Modern exploitation naturally requires focusing on traditional tactics, such as specific meta-data corruption techniques, shellcode and control-flow transition tricks, and return-into-libc attacks against DEP/NX. Beyond this, exploitation increasingly requires a thorough understanding of both the vulnerability and the target software's execution and general behavior. Much like with vulnerability research, effort invested into understanding how software works at a fundamental level will pay dividends for both attacking and defense. When attempting to exploit a heap overflow you can think of the process as a series of steps leading to a win. An understanding of the vulnerability scope and underlying Heap Management is first, followed by heap normalization and corruption. Only then can one worry about actual exploitation.

With this in mind, we've outlined a general strategy for evaluating the risk of heap vulnerabilities, which is an abstract multiple-step process. Exploitation is essentially a non-linear, creative task, so any such process will naturally have limitations, but hopefully we've collected some useful tidbits and ideas for making progress. We covered the normalization of the process state to address non-determinism, defensive and pre-emptive tactics for increasing robustness, fixation within multiple logical dimensions, and general techniques for increasing awareness of process behavior. We underscored the usefulness of isolating a set of software-specific atomic idioms that are useful in constructing larger patterns. As Nicolas Waisman has pointed out, memory leaks are notably useful for pre-destining allocations and avoiding difficulties like block-splitting and chunk coalescing. Ultimately, heap exploitation is the art of understanding the allocation and de-allocation behavior of the target application and how one can influence that behavior.

Finally, it's worth appreciating that Microsoft has made considerable strides in heap security and hardening in newer versions of Windows. We limited our scope to Windows XP and Server 2003, and the majority of the specific technical tactics outlined in this talk do not directly apply to Vista and subsequent Windows versions.

This is due to both considerable reworking of the underlying Heap Manager internals, as well as specific security hardening and technical counter-measures. Security enhancements in the newer versions include: ASLR/heap base randomization, heap meta-data encryption, termination on heap corruption, and NX/DEP. There is still a nuanced, rich attack surface, as Ben Hawkes' recent work has demonstrated, but there is no doubt that these changes have made heap exploitation much more difficult. In general, attackers will take the path of least resistance, and Microsoft's recent progress has convincingly shown that attacking application meta-data is going to be increasingly prevalent as the Heap Manager becomes more and more resilient.

Bibliography

Anisimov, Alexander. 2004. Defeating Microsoft Windows XP SP2 Heap Protection and DEP bypass. *Positive Technologies White Paper*, http://www.maxpatrol.com/defeating-xpsp2-heap-protection.pdf

Conover, Matt and Oded Horovitz. 2004. Reliable Windows Heap Exploits. *CanSecWest* 2004, http://www.cybertech.net/~sh0ksh0k/projects/winheap/CSW04%20-%20Reliable%20Heap%20Exploitation.ppt

Conover, Matt and Oded Horovitz. 2004. Reliable Windows Heap Exploits. *X'Con* 2004, http://xcon.xfocus.org/XCon2004/archives/14_Reliable%20Windows%20Heap%20Exploits_BY_SHOK.pdf

Conover, Matt and Oded Horovitz. 2004. Windows Heap Exploitation (Win2KSP0 through WinXPSP2). *SyScan* 2004, http://www.cybertech.net/~sh0ksh0k/projects/winheap/XPSP2%20Heap%20Exploitation.ppt

Conover, Matt. 2007. Double Free Vulnerabilities - Part 1. *Symantec Security Blog*, http://www.symantec.com/connect/blogs/double-free-vulnerabilities-part-1

Falliere, Nicolas. 2005. A new way to bypass Windows heap protections. *SecurityFocus White Paper*, http://www.securityfocus.com/infocus/1846

Flake, Halvar. 2002. Third Generation Exploitation. *Blackhat USA* 2002, http://www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt

Hawkes, Ben. 2008. Attacking the Vista Heap. *Ruxcon* 2008, http://www.lateralsecurity.com/downloads/hawkes_ruxcon-nov-2008.pdf

Hewardt, Mario and Daniel Pravat. 2008. Advanced Windows Debugging. New Jersey: Addison-Wesley. (Sample Chapter: http://advancedwindowsdebugging.com/ch06.pdf)

Immunity Inc. Immunity Debugger heap library source code. Immunity Inc. http://debugger.immunityinc.com/update/Documentation/ref/Libs.libheap-pysrc.html (accessed June 1, 2009)

Johnson, Richard. 2006. Windows Vista: Exploitation Countermeasures. *Toorcon* 8, http://rjohnson.uninformed.org/Presentations/200703%20EuSecWest%20-%20Windows%20Vista%20Exploitation%20Countermeasures/rjohnson%20-%20Windows%20Vista%20Exploitation%20Countermeasures.ppt

Litchfield, David. 2004. Windows Heap Overflows. *Blackhat USA* 2004, http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt

Microsoft. 2009. Virtual Memory Functions. *MSDN Online*, http://msdn.microsoft.com/en-us/library/aa366916(VS.85).aspx

Moore, Brett. 2005. Exploiting Freelist[0] on XP Service Pack 2. *Security-Assessment.com White Paper*, http://www.insomniasec.com/publications/Exploiting_Freelist%5B0%5D_On_XPSP2.zip

Moore, Brett. 2008. Heaps About Heaps. *SyScan* 2008, http://www.insomniasec.com/publications/Heaps_About_Heaps.ppt

Sotirov, Alexander. 2007. Heap Feng Shui in JavaScript. *Black Hat Europe* 2007, http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf

Waisman, Nicolas. 2007. Understanding and bypassing Windows Heap Protection. *SyScan* 2007, http://www.immunityinc.com/downloads/Heap_Singapore_Jun_2007.pdf

Waisman, Nicolas. 2008. Apology of 0days. *H2HC*, http://www.immunitysec.com/downloads/ApologyofOdays.pdf