



Heap Cache Exploitation - White Paper by IBM Internet Security Systems

*Written by: John McDonald
IBM X-Force® Researcher - IBM Internet Security Systems (ISS)*

Contents	
Abstract	1
Introduction	1
Prior Work	1
Background	4
Heap Cache	8
Technical Findings	11
Mitigation	33
Conclusion	34
Appendix A - Heap Cache Internals	36
Appendix B - Free List Algorithm Pseudo-code	40
Appendix C - Heap Cache Algorithm Pseudo-code	47
Appendix D - De-committing Policy	56
Appendix E - Historical XP Attacks	60

Abstract

The *large block index*, or *heap cache*, is an undocumented data structure used by the Windows Heap Manager in Microsoft® Windows® 2000, XP, and Server® 2003 to improve heap performance under certain specific load conditions. The processes and code that support this data structure have certain notable security implications, which can contribute to the exploitability of memory corruption vulnerabilities. In this paper, we examine the security properties of this system, and highlight how it can alter the risk profile of certain classes of memory corruption vulnerabilities.

Specifically, leveraging the heap cache can *increase heap determinism, allow exploitation of large free block corruption, allow exploitation of stale pointers to large free blocks, and provide a mechanism for exploiting limited 1-2 byte encryption of large blocks.*

Introduction

Studying heap exploitation from a general perspective can be difficult, as it's important to maintain perspective as to which pre-conditions have relevance in real-world situations. To put it succinctly, it's important to avoid the strawberry pudding predicament, enunciated by Sinan Eren as, "*I can make a strawberry pudding with so many prerequisites.*"

This work was born out of grappling with multiple memory corruption vulnerabilities involving sparsely populated heaps with large blocks. While the more intricate attack techniques documented within are useful in these situations, this paper has a simpler goal: to document an aspect of the heap that can affect heap determinism and alter the exploitability of large block code. The heap cache is tied to simple run-time performance metrics and its invocation comes in concert with changes to the somewhat counter-intuitive de-commitment policy of the heap. Consequently, this information should prove useful in efforts to better understand, predict, and model heap behavior.

Prior Work

General Heap Exploitation

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 two presentations by Matt Conover and Oded Horovitz. They each cover slightly different ground, but are both very informative, and provide insight to many undocumented aspects of the heap:

- *Windows Heap Exploitation (Conover and Horovitz 2004 SyScan)*
- *Reliable Windows Heap Exploits (Conover and Horovitz 2004 X'Con)*

There is a free chapter from the excellent book, “Advanced Windows Debugging,” which details how to use **windbg** to explore heap internals:

- *Advanced Windows Debugging Sample Chapter (Hewardt and Pravat 2008)*

Alexander Sotirov’s Heap Fung Shei library and paper are very informative, as they are chiefly concerned with heap determinism:

- *Heap Fung Shui in JavaScript (Sotirov 2007)*

Immunity also has an excellent paper with a similar focus on determinism and practical exploitation. Their python ID heap code is very informative, as it encapsulates a lot of hard-won knowledge about the system’s internals:

- *Understanding and bypassing Windows Heap Protection (Waisman 2007)*
- *Source code for Module Libs.libheap (Immunity 2009)*

Brett Moore’s papers look specifically at exploitation in the face of technical countermeasures. They represent the current and 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.

- *Exploiting Freelist[0] on XP Service Pack 2 (Moore 2005)*
- *Heaps About Heaps (Moore 2008)*

Finally, Ben Hawkes has done a considerable amount of research on attacking the Vista Heap Manager. Windows Vista® has similarly purposed data structures in ***_Heap!BlocksIndex***, though the underlying implementation is sufficiently different.

- *Attacking the Vista Heap (Hawkes 2008)*

Heap Cache Exploitation

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 first talk on the exploitation of the Windows heap (Conover and Horovitz 2004 XCon, 22-26). 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 (Moore 2005) (Moore 2008) and Nicolas Waisman's (Waisman 2007) research and are very similar in nature, and we also detail attacks similar to Ben Hawkes' work against Windows Vista (Hawkes 2008).

Terminology

We borrow the name "heap cache" from Matt and Oded's (Conover and Horovitz 2004 XCon, 22) talk, which is also in line with various clues taken from published symbols and debugger support. The pointer in the ***WinXP SP3 _HEAP*** structure in the ***ntdll.dll*** pdb is actually called the ***LargeBlocksIndex***. Unfortunately, it lacks type definitions in the public and checked pdbs, and is specified as a void pointer. The functions that work directly with this data structure indicate that it is known internally as both the heap cache and the large block index. There is some very rough support for it in the ***windbg "heap -s"*** extension, but it relies on private symbol data that aren't available.

Background

In this paper, we'll look at the heap cache data structure in-depth and see how it can be manipulated by attackers. We'll also look at the performance metrics and how the de-committing policy changes in reaction to certain heuristics.

Large Block Example

Let's start off with some code of dubious quality:

```
b1=HeapAlloc(pHeap,0,41952);
b2=HeapAlloc(pHeap,0,41952);
b3=HeapAlloc(pHeap,0,41952);

HeapFree(pHeap,0,b1);
HeapFree(pHeap,0,b2);
HeapFree(pHeap,0,b3);

((unsigned char *)b2)[35000]=0;
```

This simple code causes an access violation, as the memory backing **b2** has been de-committed back to the operating system. While this may seem like a form of justice for such clearly misanthropic code, it is a somewhat counter-intuitive result for those of us used to Unix® memory semantics. In practice, this behavior can prove quite restrictive when analyzing memory corruption issues involving large blocks, especially when there are necessarily several large de-allocations in a row or if there are stale pointers to large blocks.

Now, consider the following code:

```
for (i=0;i<300;i++)
{
    b1=HeapAlloc(pHeap,0,65536);
    HeapFree(pHeap,0,b1);
}

b1=HeapAlloc(pHeap,0,41952);
b2=HeapAlloc(pHeap,0,41952);
b3=HeapAlloc(pHeap,0,41952);

HeapFree(pHeap,0,b1);
HeapFree(pHeap,0,b2);
HeapFree(pHeap,0,b3);

((unsigned char *)b2)[35000]=0;
```

This code doesn't cause an access violation, and the system cheerfully writes over the semantically invalid memory. What's different here?

Well, Windows did something dangerously bordering on clever. The repeated allocation and de-allocation of large heap blocks has not gone unnoticed, and the Heap Manager has enacted two changes in order to improve performance:

- *The heap cache has been activated to improve large block performance*
- *The de-committing policy has been changed to favor populating the free list instead of de-committing large blocks*

Windows Memory

We'll briefly touch on Windows memory semantics and basic heap memory management before we start looking at the heap cache in detail.

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

VirtualAlloc() and *VirtualFree()* functions.

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 0x100000 bytes of memory when creating a new heap segment, and commits at least 0x2000 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 datum 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.

Uncommitted Range 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 **UnCommitted Range** (UCR) 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.

Free Lists

The 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 at the base of the heap. There are separate free 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 and a back link. For most free lists, all of the blocks in the list are the same size, which corresponds to the position of the list in the FreeList array.

All blocks higher than or equal to size 1024, however, are kept in a single free list at **FreeList[0]**. (This slot is available 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 .)

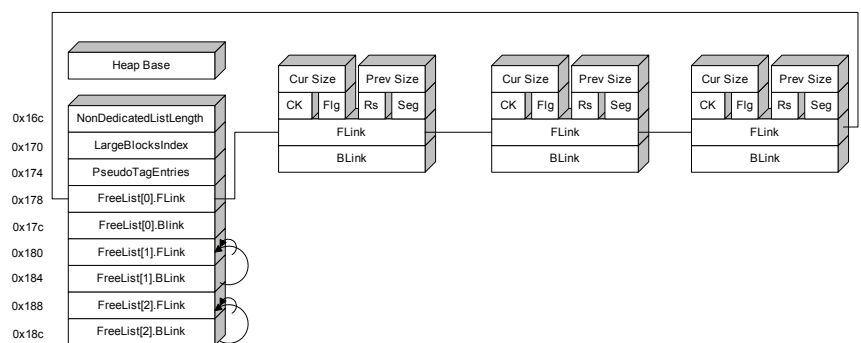


Figure 1 - Free Lists

The free lists also have a corresponding bitmap, called the **FreeListsInUseBitmap**, which is used for quickly scanning through

the *FreeList* arrays. 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.

Algorithms

The free lists contain pointers to all of the free blocks that exist in the heap, and much of the operation of the Heap Manager is concerned with managing these lists and using them to locate available blocks. Due to coalescing and block-splitting, both allocation and de-allocation can add and remove blocks to and from the free lists. Considering both allocation and de-allocation, we've isolated three core free list algorithms that are used repeatedly throughout heap code: *searching for blocks in the free lists*, *linking blocks into the free lists*, and *unlinking blocks from the free lists*. We've included pseudo-code for these algorithms in Appendix B, which will be useful for following our specific attack techniques.

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)

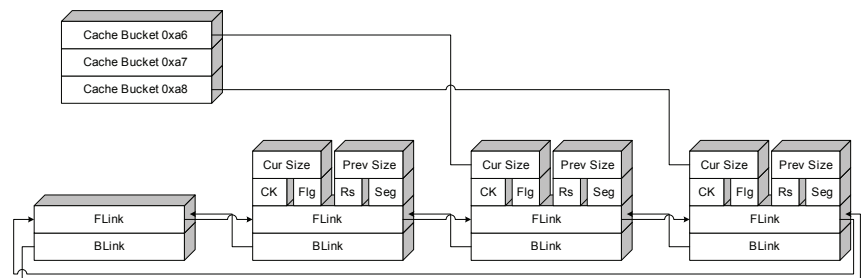


Figure 2 – Heap Cache and FreeList[0]

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.

In Appendix A, you will find more information about the heap cache data structure, including details of how its size is calculated and how it is initialized. In Appendix C, you will find pseudo-code for the algorithms that operate on the heap cache.

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()**, and is further detailed in A.3.

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

The de-committing policy is nuanced, and we cover it in more detail in Appendix D. 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.

Technical Findings

Heap Cache Invocation

The first finding is the most technically straightforward, yet arguably the most relevant in terms of overall impact.

Certain memory corruption vulnerabilities occur in large blocks, as they are tied to corruption of data structures of a large fixed size or large variable size. Exploitation of these vulnerabilities can be problematic with the default behavior of the heap manager, as blocks that are de-committed have a handful of undesirable properties. The following three are the most notable: corrupted data that exists within the blocks can be discarded before being processed, various holes can be created in virtual memory that may not lend themselves to a deterministic heap state, and the memory can simply no longer be valid, meaning that dereferencing of stale pointers will cause access violations and undesirable exceptions.

There are certainly mechanisms by which technical attackers can overcome or minimize these issues, and their effectiveness generally depends on the particular circumstances of the vulnerability and the degree to which the attacker has control of the program's run-time environment. While a clever allocation pattern or sequence of inputs can often yield windows of vulnerability, there are situations where it is a non-trivial problem to overcome (In fact, two of these situations directly led to this research.).

So, our first finding is simply that the intentional creation of the heap cache data structure can have a normalizing effect on process memory processing behavior, and can convert otherwise difficult heap corruption scenarios involving de-committing of heap blocks into more straightforward data corruption problems. The overall behavior and impact of the heap cache is also worth considering even in the more common case of memory corruption

involving small blocks. De-committing of coalesced free blocks can lead to seemingly intermittent deviations that can hinder attempts to create deterministic heap state via heap spraying or other well-documented methods.

As we previously discussed, there are two ways of causing the Heap Manager to instantiate the heap cache, which should provide multiple viable options for attackers. Essentially an attacker needs to cause one of the following conditions to become true:

- *32 free blocks need to exist in the **FreeList[0]** data structure simultaneously*

-or-

- *256 blocks need to have been de-committed since the program was initiated*

Invocation via 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:

```
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.

Prerequisites

The attacker must have some ability to influence allocation and de-allocation

behavior of the target software. This could involve the ability to open multiple simultaneous connections, or the ability to control allocation sizes and make various allocation patterns in sequence.

Invocation via 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 more than 256 blocks to de-commit 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 freeing of a buffer of size 64k or higher, 256 times. Here's a simple example:

```
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.

Prerequisites

The attacker must have some ability to influence allocation and de-allocation behavior of the target process. This could involve making multiple requests in sequence, or providing specially formatted input engineered to cause large allocations and de-allocations.

Index De-synchronization

As we've established, the heap cache is a supplemental index into the existing ***FreeList[0]*** doubly-linked list data structure. Our second finding 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:

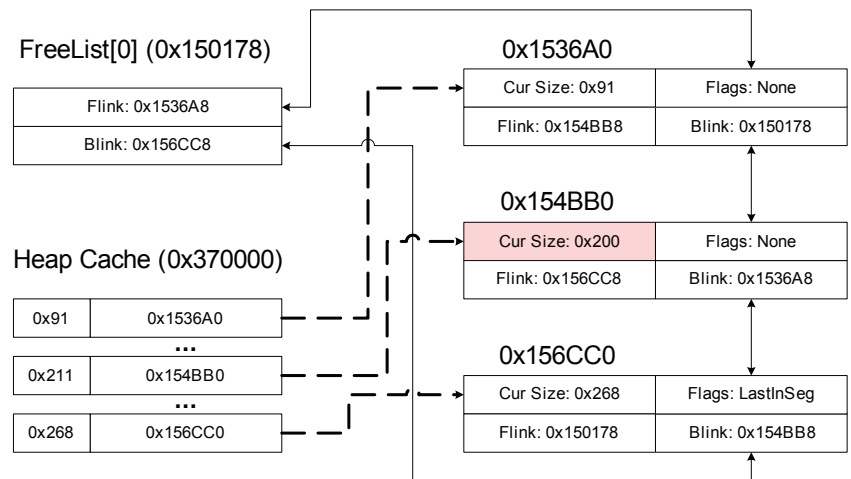


Figure 3 – Basic De-synchronization Attack Step 1

In the above diagram, we have a **FreeList[0]** with three 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 NULL 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:

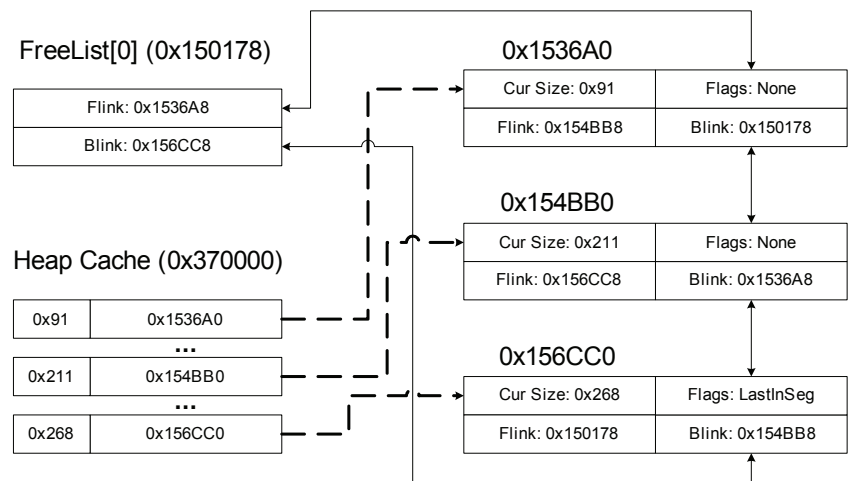


Figure 4 – Basic De-synchronization Attack Step 2

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 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. In general, we are only discussing large blocks, so any time we specify a size less than 0x400, we are talking about a block size, which is 1/8th of the actual size in bytes.

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. (You can follow the logic in our pseudo-code for the search algorithm in Appendix B.1.)

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 (see our pseudo-code for the unlink algorithm in Appendix B.2). The unlink will call **RtlpUpdateIndexRemoveBlock()**, passing it our block. If you are following along in the pseudo-code for **RtlpUpdateIndexRemoveBlock()** (Appendix C.3), you will see that it 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:

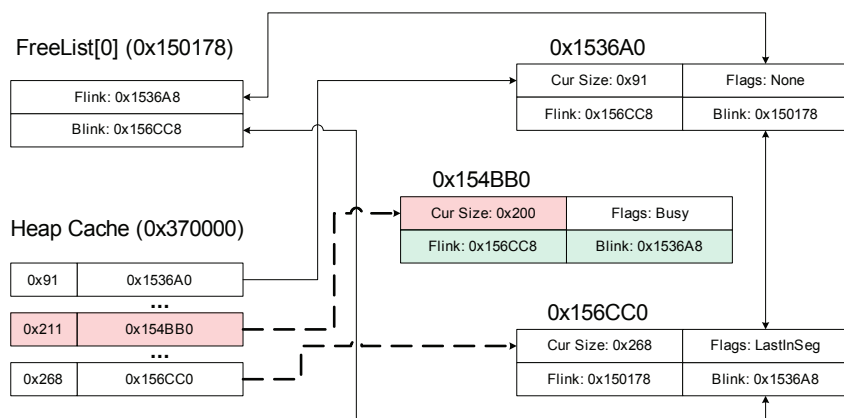


Figure 5 – Basic De-synchronization Attack Step 3

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 ***fblink*** 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 ***fblink*** and ***blink*** fields. If ***fblink*** and ***blink*** are overwritten with invalid addresses, the Heap Manager will cause an exception when it attempts to de-reference them. If the ***fblink*** 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 Microsoft Windows versions prior to Windows Server 2008 and Microsoft Vista. For Microsoft Server 2003 and Windows 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:

```
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 - One-byte Overflow of Cached Free Block

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

To help put our findings in context, we've compiled a summary of the existing attack techniques that target the XP Heap Manager. This list, located in **Appendix E**, represents a distillation of the references listed in the Prior Work section.

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 (1-4 byte Overflows)

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 listed in Appendix E as the **Bitmap Flipping Attack / Bitmap XOR Attack**. Moore's *Heaps about Heaps* documents this attack and credits it to Nicolas Waisman (Moore 2008, 21).

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 *fblink* and *blink* fields with two identical pointer values that are safe to de-reference. This attack, outlined in Moore's *Heaps about Heaps*, is applicable to free blocks of size ≤ 1024 bytes (Moore 2008, 21).

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 *fblink* and *blink* fields and some degree of prediction or preparation of memory that these pointers will reference (Moore 2008, 23-29).

De-synchronization Insert Attack

We saw that when a cache bucket is desynchronized from **FreeList[0]**, data supplied by the application can end up being interpreted as the *fblink* 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 eight bytes written into the newly allocated memory can be incorrectly interpreted as **fblink** and **blink** pointers by the Heap Manager.

If the attacker can control what the application writes to these first eight bytes, he can intentionally provide malicious **fblink** and **blink** pointers. In *Heaps About Heaps*, Moore documents several attacks against the Heap Manager that are predicated on corrupting **fblink** and **blink** pointers (Moore 2008 23-25). 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, and is documented in Appendix C.2 in pseudo-code. **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 de-reference 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 than 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 ***fblink*** 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 ***fblink*** 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:

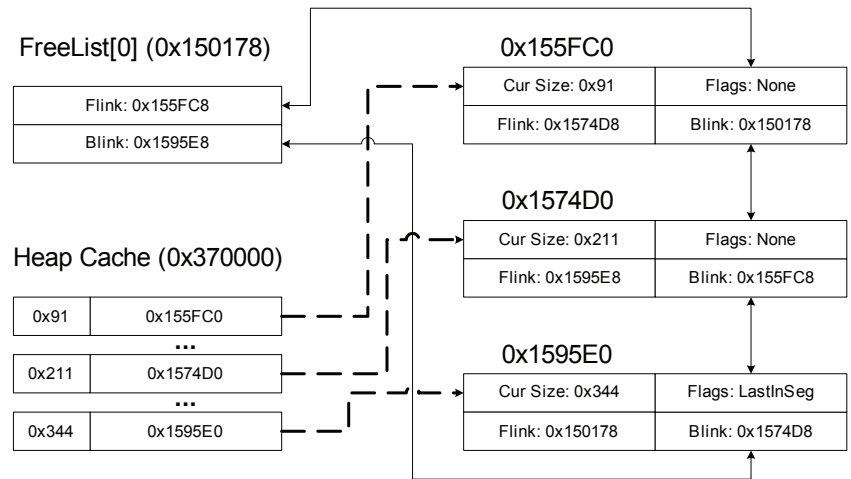


Figure 6 – Insert Attack Step 1

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:

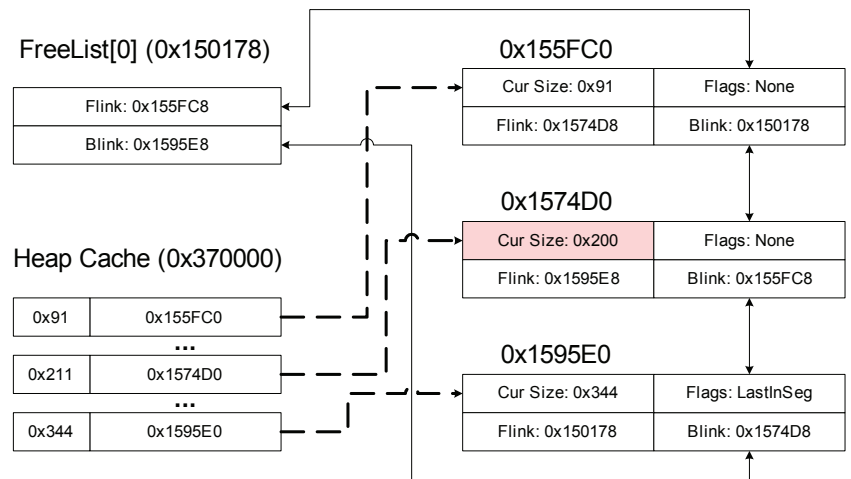


Figure 7 – Insert Attack Step 2

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 an 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()**.

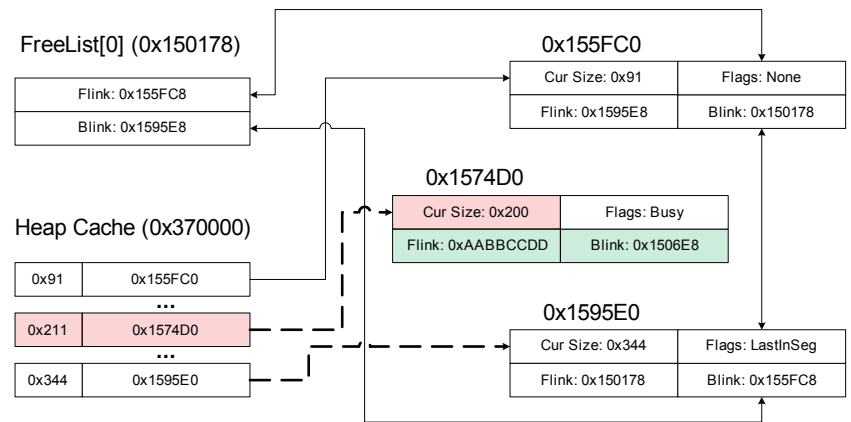


Figure 8 – Insert Attack Step 3

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 (Moore 2008, 23-25). 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 greater 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.

(Note: 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. You might want to consult the pseudo-code for the linking logic in Appendix B.3. What will happen is the following:

```

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 lookaside-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:

```

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:

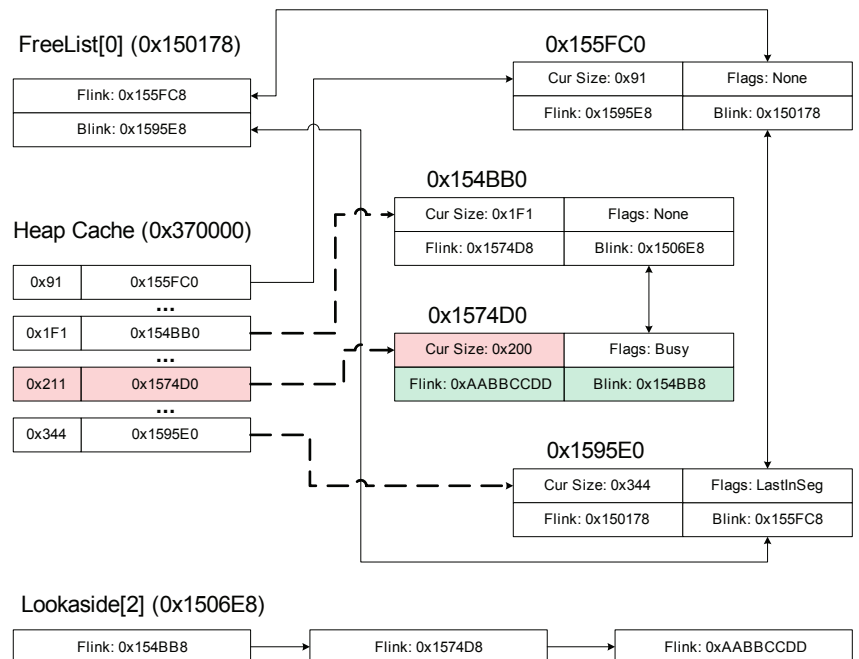


Figure 9 – Insert Attack Step 4

Summary – Insert Attack w/One-byte Overflow

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, 23-25)*, 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 **fblink** 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, 23-29).

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:

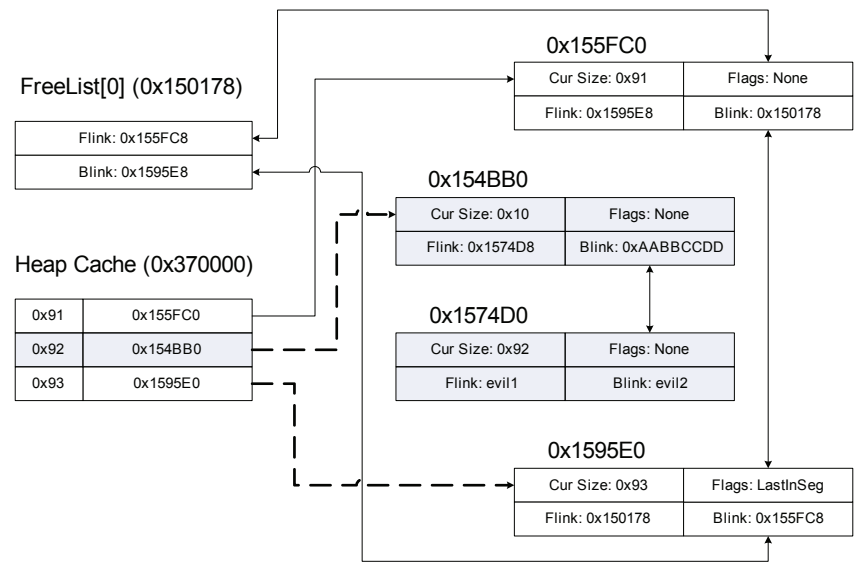


Figure 10 – De-synchronization Size Targeting

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 be 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 $\leq 0x91$, 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 $\leq 0x91$, 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, 26-27). 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:

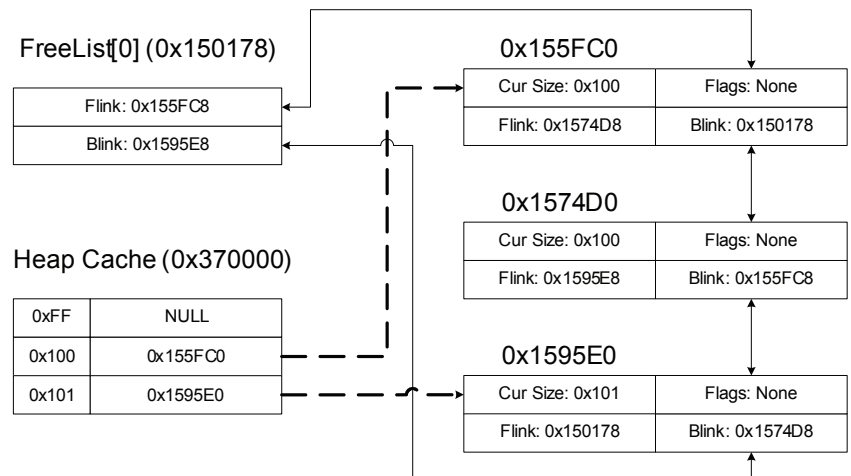


Figure 11 – Malicious Cache Entry Attack Diagram

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 de-reference 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:

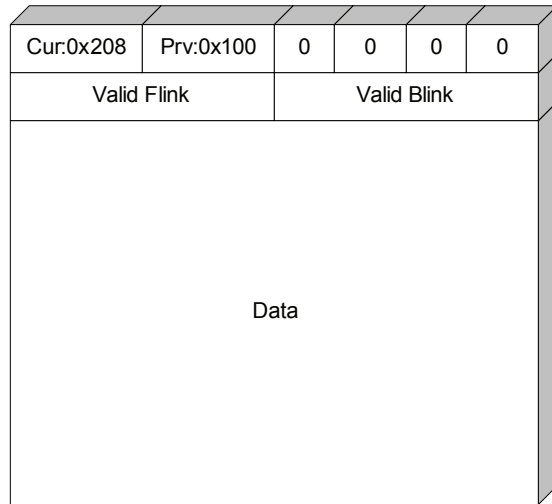


Figure 12 – Malicious Cache Entry Attack Dedicated Bucket Step 1

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

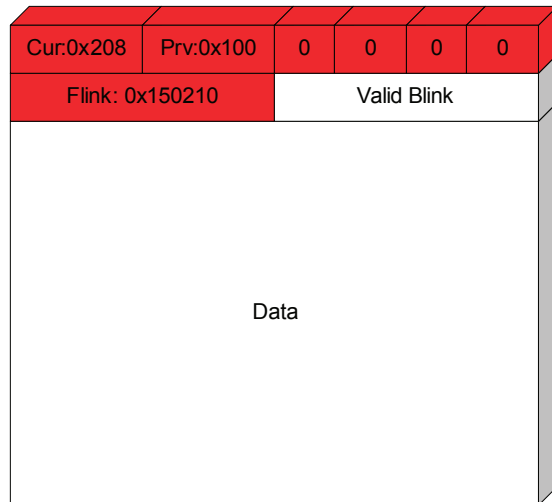


Figure 13 – Malicious Cache Entry Attack Dedicated Bucket Step 2

Essentially, the attacker didn't change anything beyond pointing the *f*link 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:

Cur:0x208	Prv:0x15	8	2	15	0
Flink: 0x150210			Blink: 0x150210		

Figure 14 – 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:

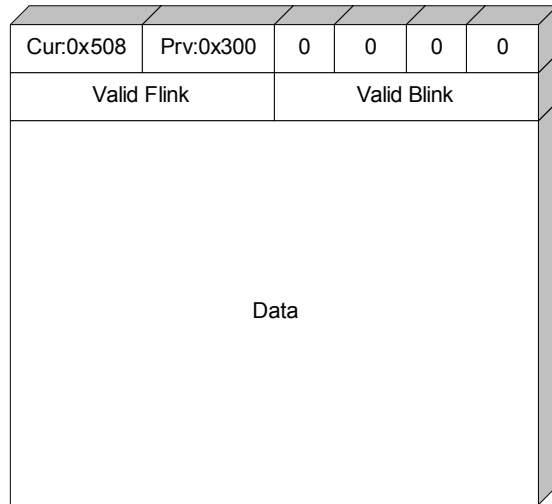


Figure 15 – Malicious Cache Entry Attack Catch-all Bucket Step 1

And you supplied these values:

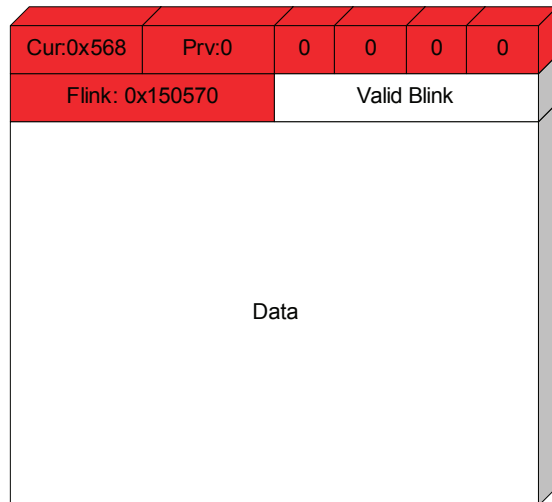


Figure 16 – Malicious Cache Entry Attack Catch-all Bucket Step 2

Assuming that you could handle coalescing by fortuitous **BUSY** flags or other planning, and every block of size $\geq 0x400$ (8192/3) was allocated before your block, your poisoned **f**link 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:

Cur:0x568	Prv:0x15	0x68	0x5	0x15	0
Flink: 0x150570			Blink: 0x150570		

Figure 17 – Malicious Cache Entry Attack Catch-all Bucket Block

Summary – Malicious Cache Entry Attack

If the attacker can overwrite the ***f*link** 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 ***f*link** 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.

Existing Attacks

This is, in essence, a variation of Moore’s attack against ***FreeList[0]*** Allocation. The interesting property here is that the corruption of the ***FreeList[0]*** isn’t necessarily as severe because any free list searching behavior using the heap cache will be able to pick up the remnants of the ***FreeList[0]*** past the point where it was corrupted (Moore 2008, 26-27). If we are corrupting an interior block that isn’t visible to the heap cache, our corrupt node may actually never be accessed until its entry into the index.

Mitigation

For existing implementations, there are some specific technical changes that could be made to make the heap cache more robust in the face of attack. The more straightforward preventative measures are as follows:

- *The back-end Heap Manager should check if it is returning a block marked as **BUSY** to an allocation request and consider that to be an error condition*
- *If **RtlpFindEntry()** returns a pointer to a block that is too small to service the request, this should be treated as an error condition that is indicative of corruption and/or an attack*
- *Alternatively, **RtlpFindEntry()** could be modified to de-reference and check the pointer it returns in order to sanity check the block size. This would be slightly more robust, as **RtlpFindEntry()** would know which bucket size the block came from and would be able to detect corruption that the caller couldn't.*
- *A default behavior of process termination on heap corruption would materially increase the difficulty of performing subtle linking attacks*

There are other technical changes that could be considered, but are more costly in terms of performance or impact to the existing system. These mitigations are as follows:

- ***RtlpUpdateIndexRemoveBlock()** could perform a sanity check to ensure that the pointer it was passed is actually in the bucket for the corresponding size, which would involve traversing list nodes. This could potentially be expensive performance-wise, and too severely penalize the performance of the heap cache.*
- *Any code that walks **FreeList[0]** could maintain a running size value and ensure that block sizes are monotonically increasing as it traverses the list*
- *The heap index could be sanity checked against **FreeList[0]** in a number of ways. This would be an expensive operation, but could be*

factored in to the flushing algorithm or other periodic code to amortize its cost.

- *Pointers to heap blocks could potentially be range checked against the segment at various points in the system, which could add a layer of defense (at the cost of adding an additional data structure.)*

Strategically, moving to a non-deterministic, difficult to predict ASLR heap implementation with encrypted meta-data is a sound approach. Naturally, this is exactly what Microsoft has done in newer versions of the operating system. From a security perspective, the Vista Heap Manager is certainly moving in the right direction, with increased non-determinism via ASLR, protection of heap meta-data with encryption, optional process termination upon corruption, and an overall heightened focus on security. For more information about security enhancements to the newer versions of the Heap Manager, please consult the following BlackHat presentation by Adrian Marienscu:

<http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Marinescu.pdf>

Moving forward, IBM ISS will focus on the Microsoft Vista and Microsoft Windows 7 OS' back-end heap manager implementations and the low-fragmentation heap front-end, paying particular attention to sub-systems affected by run-time heuristics.

Conclusion

The heap cache is an interesting dynamic component of the Windows Heap Manager, which, along with its associated changes to memory de-committing policies, can play a large role in determining overall heap behavior. We've studied how the heap cache works, how it fits into the larger system as a whole, and provided documentation that should prove useful in integrating this knowledge into existing security tools and resources. Our primary focus has been an analysis of the heap cache from an application security perspective, and, to this end, we've documented several notable properties that affect the overall security of a running process.

Our primary finding is that the heap cache can be intentionally invoked by an attacker in order to make otherwise difficult memory corruption and stale pointer attacks involving large blocks exploitable. This can change the risk

profile of large block memory corruption vulnerabilities.

The rest of our efforts focused specifically on the resilience of the heap cache's internal data structures, and showed how the index can be desynchronized or corrupted by an attacker as part of an exploit. These specific technical attacks each carry various pre-conditions and requirements, and aren't intended to be general indictments of the system as a whole. They show essentially how an attacker would approach attacking the system technically, but, in practice, the attacks documented within would need considerable effort to be applied in real-world situations.

We finished our analysis by considering several specific technical mitigations that could be applied to existing systems, and the strategic relevance of our observations looking towards the newer versions of the Windows Heap Manager.

Appendix A – Heap Cache Internals

Note: Many of the details below have been inferred based on the binary and will not match the actual internal, private names used by Microsoft.

A.1 Structure Definition

The heap cache consists of an 0x68 byte header, an array of buckets (*NumEnts* * 4 bytes), and a bitmap for fast traversal (*NumEnts* / 8 bytes). *NumEnts* is 0x380 by default. The total size is rounded up to the next 0x1000 byte (1 page) boundary for the *VirtualAlloc()*. The header contains:

```
struct HeapCache
{
    ULONG NumBuckets;
    int CommittedSize;
    LARGE_INTEGER CounterFrequency;
    LARGE_INTEGER AverageAllocTime;
    LARGE_INTEGER AverageFreeTime;
    int SampleCounter;
    int field_24;
    LARGE_INTEGER AllocTimeRunningTotal;
    LARGE_INTEGER FreeTimeRunningTotal;
    int AllocTimeCount;
    int FreeTimeCount;
    int Depth;
    int HighDepth;
    int LowDepth;
    int Sequence;
    int ExtendCount;
    int CreateUCRCount;
    int LargestHighDepth;
    int HighLowDifference;
    unsigned __int8 *pBitmap;
    HEAP_FREE_ENTRY **pBuckets;

    HEAP_FREE_ENTRY *Buckets[NumBuckets];

    unsigned int Bitmap[NumBuckets/32];
};
```

A.2 Structure Fields

NumBuckets - the number of buckets in the heap cache

CommittedSize - the size of the virtual memory allocated for the heap cache

CounterFrequency - the frequency of the high performance counter

AverageAllocTime - every 0x64 allocs, ***AverageAllocTime*** is ***AllocTimeRunningTotal/AllocTimeCount***

AverageFreeTime - every 0x64 frees, ***AverageFreeTime*** is ***FreeTimeRunningTotal/FreeTimeCount***

SampleCounter - counter that invokes timing code every 0x200 samples (allocation or free)

AllocTimeRunningTotal - running total of time spent in allocation

FreeTimeRunningTotal - running total of time spent in free

AllocTimeCount - counts 0x64 allocs, then used to determine ***AverageAllocTime***

FreeTimeCount - counts 0x64 frees, then used to determine ***AverageFreeTime***

Depth - tracks length of catch-all bucket

HighDepth - set to same as ***Depth*** initially. incremented when heap is extended, and tracks highest value of ***Depth***. set back to ***Depth*** on flush

LowDepth - set to same as ***Depth*** initially. tracks lowest value of ***Depth***. set back to ***Depth*** on flush

Sequence - used as simple recursion mechanism. If it's set, we update it with every large block action and track ***HighDepth*** and ***LowDepth***. If ***Sequence*** gets above 0x400 on a ***RtlpDeCommitFreeBlock()***, we invoke the global cache balancing function.

ExtendCount – number of times heap has been extended when the cache is active

CreateUCRCount – number of times UCR has been created when cache is active

LargestHighDepth – Largest HighDepth witnessed over lifetime of cache

HighLowDifference –Largest difference between High and Low depths witnessed over lifetime of cache

pBitmap - pointer to the bitmap used for quick navigation of the index

pBuckets - points to the index: the array of buckets

Buckets are an array of NumBuckets pointers, each one representing a different size.

Bitmap is used for quick scanning of the cache array. This works identically to the bitmap used by the **FreeList** implementation, with one bit representing each bucket.

A.3 Initialization

A pointer to the heap cache is present at offset 0x170 from the base of the heap on Microsoft Windows XP SP3. This is the **LargeBlockIndex** in the **_HEAP** type definition in the public *ntdll.dll* pdb provided by Microsoft. It is initially **NULL** and is only set if the heap cache is instantiated during the program runtime.

```
struct HeapCache *RtlpInitializeListIndex(PHEAP aHeap)
```

This function is responsible for initializing the heap cache data structure and performing the initial synchronization of the index array with the FreeList[0] list.

The heap cache size is calculated based on the **_HEAP.DeCommitFreeBlockThreshold** setting for the heap. The

number of entries in the heap cache is calculated by taking the ***DeCommitFreeBlockThreshold***, adding 0x180, and rounding the number up to the next 0x20 byte boundary.

This can be set as a parameter when the heap is created with ***RtlCreateHeap()***. If it is not set as an argument, it is set to the value in the ***PEB: _PEB.HeapDeCommitFreeBlockThreshold*** (>>>3). It can be set in the PE header, and defaults to 0x1000; the size of one page. The ***HeapDeCommitTotalFreeThreshold*** defaults to 0x10000.

The heap cache is located in a separate virtual memory range that is allocated by ***RtlInitializeListIndex*** using ***NtAllocateVirtualMemory()***.

The initial setup is done by walking through ***FreeList[0]***, starting at the smallest block at the front of the linked list. For each entry in ***FreeList[0]*** it adds a pointer back to the entry in the appropriate bucket in the heap cache, assuming the bucket has not already filled. This works to have the bucket point to the smallest appropriate element in the ***FreeList[0]***, as the traversal is performed from smallest to largest.

Appendix B – Free List Algorithm Pseudo-code

Note that the following pseudo-code and algorithm descriptions are slightly abstracted and represent logical functions, but are not what you will find verbatim in *ntdll.dll*. In the actual assembly, you will see these algorithms implemented in multiple places, with some small changes in each instantiation. It's also important to note that we're talking about the back-end Heap Manager specifically. The front-end, comprised of the look-aside lists or, optionally, the low fragmentation heap, has its own algorithms. Large blocks aren't handled by the look-aside front-end, and, while the low fragmentation heap handles blocks up to 16k, it is optional and isn't commonly used on XP. (Alexander Sotirov informed us of one rather notable exception: Internet Explorer 7 now uses the low fragmentation heap on XP.)

B.1 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'll cover the linking-related searches in B.3 below.

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 this search algorithm is: 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:

Searching Pseudo-code Part 1

```
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.

Note that if we can affect the bitmap, we can cause the system to potentially return a pointer to the base of the heap in response to an allocation. (If we flip the bit for an empty free list, allocations will return an empty free list head's blink, which will point back to the sentinel node.) This is a useful property for attacks, which was documented in Brett Moore's presentation Heaps about Heaps, and credited to Nicolas Waisman (Moore 2008, 21).

Searching Pseudo-code Part 2

```
if (Heap->LargeBlocksIndex ) // Heap Cache active?
{
    foundentry = RtlpFindEntry(Heap, size);

    // Not found in Heap Cache
    if (&FreeLists[0] == foundentry )
        return NULL;

    // returned entry not large enough
    if (SIZE(foundentry) < size)
        return NULL;

    // we're allocating a >=4k block, and the smallest block we find
    // is >=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;
    }

    // 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 very large block allocation request only 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. We'll look at the heap cache's searching implementation in Appendix C.

Searching Pseudo-code Part 3

```
// 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;

// Our request is bigger than biggest block available
if (SIZE(Biggest)<size)
    return NULL;

walker = &FreeLists[0];

while ( 1 )
{
    walker = walker->Flink;

    if (walker == &FreeLists[0])
        return NULL;

    if ( SIZE(walker) >= size)
        return walker;
}
```

If the heap cache isn't active, we need to search **FreeList[0]** manually. The system starts with the first free block in **FreeList[0]**, at **FreeList[0].flink**, and 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.

B.2 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*:

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 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.)

B.3 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:

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. (This can be useful for the attack outlined in B.1).

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.)

Appendix C – Heap Cache Algorithm Pseudo-code

We've documented how the heap cache is integrated into the core operation of the Heap Manager in Appendix B. In that coverage, we listed several functions that form the core API to the heap cache. The three main functions are: ***RtlpFindEntry()*** for searching, ***RtlpUpdateIndexInsertBlock()*** for linking, and ***RtlpUpdateIndexRemoveBlock()*** for unlinking. There are additionally two functions used for flushing that we will briefly examine.

C.1 Searching

```
_LIST_ENTRY *RtlpFindEntry(PHEAP Heap, size_t Size)
```

The heap cache is queried with the routine ***RtlpFindEntry()***, which takes a pointer to the heap and a size parameter. It searches the heap cache for the first block in ***FreeList[0]*** that is the same size or bigger than the size parameter, and returns a pointer to that block. If it can't find a suitable block, it returns a pointer to the head node of ***FreeList[0]***. Here is pseudo-code for this function:

RtlpFindEntry Pseudo-code

```
FreeList0 = &Heap->FreeLists[0];  
Biggest = Heap->FreeLists[0].Blink;  
  
// empty freelist[0]  
if ( Biggest == FreeList0 )  
    return FreeList0;  
  
// biggest chunk in freelist isn't big enough, so just return f1[0]  
if (SIZE(Biggest) < Size )  
    return FreeList0;  
  
result = FreeList0->Flink;  
  
// if first chunk in free list is big enough, just return it  
if (Size <= SIZE(result))  
    return result;  
  
Cache = Heap->LargeBlocksIndex;  
  
Index = Size - 128;  
if ( Index >= Cache->NumBuckets )  
    Index = Cache->NumBuckets - 1;
```

```
// user wants a big block - not tracked in cache
if ( Index == Cache->NumBuckets - 1 )
{
    // the linked list is 8 bytes in, but the ptr refers to the head
    walker = LISTPTR(Cache->pBuckets[Index]);
    while ( walker != FreeList0 )
    {
        // walk through big guys until we find one big enough
        if ( SIZE(walker) >= Size )
            return walker;
        walker = walker->Flink;
    }
}

// ok, use bitmap to find next largest entry
if (offset=scan_HeapCacheBitmap(HeapCache, Size))
{
    return LISTPTR(Cache->pBuckets[offset]);
}
else
{
    DbgPrint("Index not found into the bitmap %08lx\n", Size);
    result = (_LIST_ENTRY *)FreeList0;
    return result;
}
```

The search algorithm works very similarly to the free list search algorithm. It uses the size parameter to calculate the index into the array. It then uses a bitmap, where one bit represents each bucket in the array. Starting at the calculated bucket, it uses the bitmap to scan the data structure for the next available free block.

There is special case code if the size corresponds to the last bucket, which contains all blocks that are too large to have dedicated buckets. In this situation, it walks through the linked list looking for a suitable block, starting at the initial pointer in the last bucket.

Note that for allocation-related searches for a free block to service a request, the

block returned by this function represents the end of the search. If the block is too small somehow (a result of our de-synchronization attack), the allocation code will simply forego the free lists and extend the heap to create a new block suitable for the request.

However, for linking related searches, the pointer returned from **RtlpFindEntry()** is used as a starting point for the search. So, if the block returned isn't large enough, the linking code keeps walking through the doubly-linked list to find a block that is large enough. Since it gets the **flink** pointer from the returned block, this provides us some extra flexibility when we corrupt or desynchronize blocks that are indexed by the heap cache.

C.2 Linking

```
RtlpUpdateIndexInsertBlock(PHEAP Heap, _HEAP_ENTRY *Chunk)
```

RtlpUpdateIndexInsertBlock() is used to add a new free block to the heap cache, if appropriate. This also keeps track of simultaneous entries in **FreeList[0]**, and will enable the heap cache if it sees 32 of them.

RtlpUpdateIndexInsertBlock Pseudo-code

```
if (SIZE(Chunk)<0x80u)
    return;

++Heap->NonDedicatedListLength;

HCache = Heap->LargeBlocksIndex;

if (!HCache)
{
    // 0x20 simultaneous entries in freelist[0]
    if ( Heap->NonDedicatedListLength >= 0x20 )
        RtlpInitializeListIndex(Heap);

    return;
}

Index=SIZE(Chunk)-0x80;

// cap it at catch-all bucket
if (Index >= HCache->NumEntries)
```

```
Index=HCache->NumEntries-1;

BucketPtr=&HCache->pBlockIndex[Index]; // location of the bucket
CacheEnt=*BucketPtr;                  // current bucket entry

// update so it points to us
// smallest size logic is for catch-all
if (!CacheEnt || SIZE(Chunk)<=SIZE(CacheEnt))
    *BucketPtr = Chunk;

// set the corresponding bit in the bitmap
if (!CacheEnt)
    HCache->pBitmap[Index >> 3] |= 1 << (Index & 7);

if (Index==(HCache->NumEntries-1)) // last index
{
    // another big guy added
    ++HCache->Depth;
    if (HCache->Sequence )
    {
        HCache->Sequence++;
        if (HCache->Depth > HCache->HighDepth)
            HCache->HighDepth = HCache->Depth;
    }
}
```

If the chunk is smaller than 0x80, return as it doesn't belong in the heap cache.

Increment *heap->NonDedicatedListLength*. This keeps track of how many simultaneous blocks exist in *FreeList[0]*. (We activate the heap cache when we see 32 simultaneous free large blocks.)

If the heap cache isn't enabled, we check to see if *NonDedicatedListLength* is 32. If it is, we activate the heap cache by calling *RtlpInitializeListIndex()*. Otherwise, return.

If the cache is present, the index is calculated as *Size - 0x80*, with a maximum index for the catch-all bucket for the biggest chunks.

If it's within the heap cache, it checks to see if the bucket is set. If the bucket is set, it replaces the existing entry with this block. The replace only happens if the new entry is smaller than or equal to the existing entry, which is present for

the purpose of the catch-all entry.

If the bucket isn't set, it sets the bucket and then sets the appropriate bitmap.

Finally, if we add an entry to the catch-all bucket, we increase the *Depth*, increment *Sequence*, and update *HighDepth* if necessary.

C.3 Unlinking

```
int RtlpUpdateIndexRemoveBlock(PHEAP aHeap, _HEAP_ENTRY *Chunk)
```

This looks up the bucket based on the size, and if the bucket is occupied and the entry corresponds to the pointer, it looks and sees if it can pull out the next entry. It also decrements the simultaneous free block counter.

RtlpUpdateIndexRemoveBlock Pseudo-code

```
Size = SIZE(Chunk);  
HeapCache=Heap->LargeBlocksIndex;  
  
if (Size<0x80)  
    return;  
  
--Heap->NonDedicatedListLength;  
  
if (!HeapCache)  
    return;  
  
Index=Size-128;  
  
// highest bucket  
if (Index>=HeapCache->NumEntries )  
    Index=HeapCache->NumEntries-1;  
  
//Next is ptr to next chunk in freelist[0]  
if (Chunk->Flink!=&Heap->FreeLists[0])  
    Next = Chunk->Flink;  
else  
    Next = NULL;  
  
//bucket is pointer to index entry  
Bucket = &HeapCache->pBlockIndex[Index];  
  
// we are in index
```

```
if (*Bucket==Chunk)
{
// if its a biggie, we don't care if next size is different
if (Index>=HeapCache->NumEntries-1)
{
if (!Next)
{
*Bucket = 0;
HeapCache->pBitmap[Index >> 3] ^= 1 << (Index & 7);
}
else
*Bucket = Next;
}
else
{
// its not a biggie, so dont replace with mismatched size
if ( !Next || Next.Size != Size )
{
*Bucket=0;
HeapCache->pBitmap[Index>>3] ^= 1 << (Index & 7);
}
else
*Bucket = Next;
}
}

// if it's a biggie
if (Index==(HeapCache->NumEntries - 1))
{
// decrease the depth
if ( --HeapCache->Depth < 0 )
DbgPrint("Invalid Cache depth\n");

// if sequence is set, increment and track lowdepth
if (HeapCache->Sequence)
{
HeapCache->Sequence++;
if ((signed int)HeapCache->Depth < HeapCache->LowDepth)
HeapCache->LowDepth = HeapCache->Depth;
}
}
```

If the chunk is smaller than 0x80, return as it's not in the heap cache.

Decrement *heap->NonDedicatedListLength*. This keeps track of how many simultaneous blocks exist in *FreeList[0]*.

The index is calculated as *Size* - 0x80, with a maximum index for the catch-all bucket for the biggest chunks.

We check the bucket for the size of the chunk we are given, and if the entry in the bucket is not our chunk, then we are done.

If our chunk is in the bucket, we update our bucket. Basically, we follow our chunks *flink*, and if the size is the same, we store the *flink* in the bucket. The catch-all bucket is a special case, and we update it if the next chunk is simply of a larger size. If the *flink* isn't the right size, we store *NULL* in the bucket and toggle the bitmap.

If we added an entry to the catch-all bucket, we decrease the *Depth*, increment *Sequence*, and update *LowDepth* if necessary. We print a warning if the *Cache Depth* has gone below zero.

C.4 Flushing

```
int RtlpFlushLargestCacheBlock(PHEAP aHeap)
```

This function takes the largest available block on the free list and attempts to de-commit it.

Flushing of the largest block is invoked sometimes in allocation, if the requested size is sufficiently large and the retrieved matching block is at least four times the size of the requested size.

```
HeapCache = Heap->LargeBlocksIndex;  
  
if (!HeapCache)  
    return;
```

```
if (!(Sequence = HeapCache->Sequence))
    return;

Biggest=Heap->FreeLists[0].Blink;

if ( Biggest==&Heap->FreeLists[0] )
    return;

// this prevents lowdepth and highdepth from being altered
// also prevents recursive flushing
HeapCache->Sequence = 0;

// remove biggest chunk from the cache map
BiggestChunk=ENTPTR(Biggest);

// invokes our unlink algorithm on the biggest block
Do_Unlink(BiggestChunk);

// go ahead and purge any pages we can behind this guy
Biggest->Flags |= BUSY;
Heap->TotalFreeSize -= BiggestChunk->Size;
RtlpDeCommitFreeBlock(aHeap, BiggestChunk, BiggestChunk->Size);

// restore saved sequence
HeapCache->Sequence = Sequence;

void RtlpFlushCacheContents(PHEAP aHeap)
```

The sequence is checked at de-commit. If the *Sequence* hits 0x400, *RtlpFlushCacheContents()* is called. This is a more complicated routine that adjusts the size of the cache.

First, the difference between *HighDepth* and *LowDepth* is calculated. This is the amount that the catch-all bucket has grown since initialization or the last flush operation. If the current *Depth* is less than or equal to the difference, then *HighDepth* is set to *Depth*, *LowDepth* is set to *Depth*, *Sequence* is set back to 1, and the function returns.

After the first 0x400 large block events post-initialization, let's say that *LowDepth* is 0, *HighDepth* is 5, and *Depth* is 3. *Depth* is less than 5, so no flushing is performed and *LowDepth=HighDepth=Depth=3*. Now, let's say 0x400 large block operations later, *LowDepth* is 3, *HighDepth* is 4, and *Depth*

is 3. **Depth**(3) is going to be higher the difference of 1 (4-3), so we'll perform the flush this time.

The flushing algorithm works by starting at the first entry in the catch-all bucket and walking through each entry until it hits the beginning of **FreeList[0]**. So, all entries above size 8192 are subject to being flushed.

It calculates the number of entries it needs to flush in order to bring the **Depth** back down to within range of the calculated difference. So, in our previous example, if **Depth** was 3 and **DepthDifference** was 1, it would free the last two blocks present in **FreeList[0]** in order to bring **Depth** down to 1.

As it walks through the blocks, it sets the **NO_COALESCE** flag on any free block that it advances past until it reaches the first block it wants to free. Also, if it sees any block with the **NO_COALESCE** flag set, it will go ahead and flush that block. So, any free blocks > size 8192 will be flushed during the second global flush operation that they witness.

Once it reaches the blocks that it wants to free, it safely unlinks them from **FreeList[0]** and places them in a temporary list.

Once the end of **FreeList[0]** is reached, the flushing code walks through the temporary list and calls **RtlpDecommitFreeBlock()** on each one. Since sequence is temporarily set to 0, this will cause the block to be de-committed.

Finally, **HighDepth** and **LowDepth** are both set to **Cache**, **Sequence** is set back to 1, and the function returns.

Appendix D – De-committing Policy

The de-committing logic is split into two parts: the selection of candidate blocks and the actual de-committing function. The selection algorithm does not change when the heap cache is activated. The de-committing function, however, changes considerably with the presence of the heap cache.

- *Selection algorithm summary:*

If the block being freed is larger than or equal to 1 page in size, and the total number of free bytes in the heap after the free() will be higher than 64k, then select the block to be de-committed. Otherwise, the block is added to the free lists.

- *De-committing algorithm summary:*

If the heap cache is not invoked, then increment Heap->DecommitCount. This is a counter that counts the number of de-commit operations since the beginning of the process. If the counter is below 256, we de-commit the block. If the counter reaches 256, then we go ahead and initialize and create the heap cache.

If the heap cache is present, then there are numerous checks, but we generally insert the block into the free lists instead of de-committing it. There are a couple of cases to be aware of involving small free lists, which are noted below.

D.1 Selection Algorithm

If a block is freed that meets the following conditions, it is subject to the de-committing logic. The following pseudo-code shows how a block is determined to be a candidate for de-committing.

(DeCommitFreeBlockThreshold typically defaults to 0x2000.

DeCommitTotalFreeThreshold typically defaults to 0x2000.)

De-commit Selection Pseudo-code

```
if (Size >= 0x80)
{
    if (Size >= DeCommitFreeBlockThreshold)
    {
        if (Size + heap->totalFreeSize >=
deCommitTotalFreeThreshold)
        {
            RtlpDeCommitFreeBlock(Heap, Block);
            return;
        }
    }

    if (Size + heap->totalFreeSize > DeCommitTotalFreeThreshold)
    {
        if (!(rtlpDisableHeapLookaside & 0x2))
        {
            if (Size >= 0x200)
            {
                if (PREVSIZE(Block) == 0)
                {
                    RtlpDeCommitFreeBlock(Heap, Block);
                    return;
                }
                if (Flags(Block) & LAST_BLOCK)
                {
                    RtlpDeCommitFreeBlock(Heap, Block);
                    return;
                }
            }
        }
    }
}

// add to the free list and don't decommit
AddToFreeList(Block);
```

D.2 RtlpDeCommitFreeBlock

The de-committing routine, ***RtlpDeCommitFreeBlock()***, is where the logic involving the heap cache comes into play.

The first thing the function does is take the block it was given and separate it into three parts: a clean set of pages aligned on a page boundary, the extra data that was before these pages and the extra data after these pages. It then attempts to do coalescing on both the before and after chunks, which will potentially merge them with adjacent free chunks.

If there is more than 10 UCRs in use by the segment (i.e. holes in the segment), and the chunk is either the first chunk in the segment or the last chunk in the segment, then we go ahead and de-commit the block.

If, after coalescing, either of the border chunks is actually big enough that it's larger than the ***DeCommitFreeBlockThreshold***, ***RtlDecommitFreeBlock()*** will actually call itself recursively.

Assuming that this doesn't happen, the system now checks to see if the heap cache is active. If it is not, we do the following:

```
v_heapcache_ = Heap->LargeBlocksIndex;
if ( !v_heapcache_ )
{
  ++Heap->DecommitCount;
  if ( Heap->DecommitCount == 256 )
  {
    if ( LOBYTE(Heap->Flags) & 2 )
    {
      if ( !(RtlpDisableHeapLookaside & 2) )
        RtlpInitializeListIndex(Heap);
    }
  }
  goto LABEL_CreateUCR;
}
```

If the heap cache is present, then we check the sequence to see if it's zero (indicating a forced de-commit due to a potentially recursive or mutually recursive call), or if we are the only chunk in the segment (no previous or next neighbors), then we go ahead and de-commit.

Otherwise, we perform one more check against the heap cache.

If the heap depth is higher than or equal to the ***HighDepth***, then we are going

to slightly alter our insertion into the free list. (Every time the heap is extended, the *HighDepth* is incremented, so this can happen even if we haven't removed any entries from the catch-all bucket.) Specifically, if *FreeList[0]* is empty or the largest block in the *FreeList[0]* is smaller than us or only 512 bytes larger, then we de-commit. Otherwise, we insert the block into the free list, but we also flush the largest block from the cache.

Assuming we pass the depth check, we simply insert the block into the free list.

There is one last check after everything is complete: the sequence number is checked against a large predetermined value (0x400), and if we've hit that value, we call *RtlpFlushCacheContents()*, which may reset the *Sequence* value or flush a portion of the blocks higher than size 8192. This is documented in C.4, and depends on how the catch-all bucket of the heap cache has been utilized over the last 0x400 large block operations.

D.3 Debugging

(based on Alexander Sotirov's breakpoints (Sotirov 2007))

```
bu ntdll!NtAllocateVirtualMemory ".printf \"
valloc(addr=%x,size=%x,alloc=%x,prot=%x)\\n\", poi(poi(esp+8)),
poi(poi(esp+10)), poi(esp+14), poi(esp+18); g"
```

```
bu ntdll!NtFreeVirtualMemory ".printf \"
vfree(addr=%x,size=%x,type=%x)\\n\", poi(poi(esp+8)), poi(poi(esp+c)),
poi(esp+10); g"
```

```
bu ntdll!RtlpDeCommitFreeBlock ".printf \"
decommit(heap=%x,ent=%x,size=%x)\\n\", poi(esp+4), poi(esp+8),
poi(esp+c); g"
```

```
bu ntdll!RtlpInsertFreeBlock ".printf \"
insfree(heap=%x,ent=%x,size=%x)\\n\", poi(esp+4), poi(esp+8),
poi(esp+c); g"
```

Appendix E – Historical XP Attacks

This list summarizes the existing published techniques for exploiting heap corruption on Windows XP. The list includes techniques that are useful for achieving a write-4/write-self/4-to-nbyte primitive, but does not include techniques that are used as part of the secondary stage of a corruption attack (e.g. overwriting segment pointer with an unsafe unlink, or remapping heap cache.) This list does not include Ben Hawkes' research targeting the Vista Heap Manager (Hawkes 2008), though many of his techniques should be backwards-portable.

E.1 Pre-SP2

Valloc Unlink Attack – overwrite a busy chunk with fields indicating it is a virtual alloc chunk, and provide malicious values for *fblink* and *blink*. This was addressed by safe unlinking (Halvar 2002, 21-24).

Coalesce Unlink Attack – overwrite chunk with fields indicating it is a free block, providing malicious values for *fblink* and *blink*. Coalesce will cause overwrite chunk to be unlinked. This was addressed by safe unlinking (Litchfield 2004, 16) (Conover and Horovitz 2004).

Coalesce Unlink Double Attack – overwrite active chunk with fields indicating it is a busy block, providing malicious values for *fblink* and *blink*, as well as prev and cur size fields pointing to fake pre-constructed blocks. Coalesce will cause overwrite chunk and fake chunk to be unlinked, leading to multiple arbitrary overwrites. This was addressed by safe unlinking (Conover and Horovitz 2004).

Double Free Attack – Between first and second free, attacker must be able to manipulate freed header to perform a malicious coalesce. This was partially addressed by safe unlinking (Conover and Horovitz 2004).

One-byte Overflow Coalesce Unlink Attack – overwrite current size field of next chunk with *LSB* of 0 and supply a fake chunk in the source buffer. This was addressed by safe unlinking (Conover and Horovitz 2004).

E.2 Post-SP2

Cookie Brute Force – in a vacuum, brute forcing cookie value (checked upon *free()* of a busy block), will be correct 1 in 255 times. (Conover and Horovitz 2004)

Critical Section Unlinking – Small data structures on the Process Heap linked from Critical Sections contain a doubly-linked list that can be targeted. (Falliere 2005)

Safe Unlinking Defeat – if attacker can overflow a chunk on a free list, and knows the size of the chunk, and knows that the free list for that size only contains that one chunk, then the attacker can provide skewed *flink* and *blink* values that still point to the free list head. The second allocation for the size corresponding to that free list will end up returning a pointer to the base of the heap, allowing for straightforward exploitation. (Conover and Horovitz 2004 SyScan)

Bitmap Attack – if the Free List bitmap can be disrupted, if a bit corresponding with an empty `FreeList[]` is toggled to be set, it leads to an exploitable situation. Essentially an allocation for the size corresponding to that bit will return an address at the base of the heap, which can result in arbitrary code execution given some ability to control what is written there. The commit function pointer is one of the more straightforward ways to achieve this end. (Moore 2008, 11-13) (Credited to Nicolas Waisman)

Lookaside Attack – if the *flink* of a chunk freed to a look-aside list can be overwritten, this value will be propagated to the head of the look-aside, causing a specific allocation by the application to return the attacker provided address. (Anisimov 2004, 3-7)

Double Free Attack – If one chunk is freed to look-aside and other to free lists, after allocation from look-aside but before allocation from free lists, we can alter *flink/blink*. If chunk is freed twice to look-aside, we can alter *flink* in between allocations to have arbitrary address populate the look-aside head. (Conover 2007)

FreeList Head Attack – if the *blink* field of any *FreeList* entry at the base of the heap can be overwritten, the address of the next block to be freed will be

written to the attacker-supplied address. (Moore)

Bitmap XOR Attack – the free list bitmap is updated via an XOR operation, so if the system tries to clear the wrong bit, it can accidentally toggle a free bit into a set bit. This results in the Bitmap Attack outlined above. This can be caused by an attacker overwriting the current size field of a block on an existing **FreeList[]** for a size <0x80. The attacker either needs to be overwriting the only chunk on a **FreeList[]** (so **fblink**==**blink** naturally), or the attacker needs to overwrite **fblink** and **blink** manually and set them equal to each other and readable. (Moore 2008, 21) (Credited to Nicolas Waisman)

Busy Chunk Manipulation – The attacker changes the size field of a chunk that will be freed by the application. This allows them to free the list to an arbitrary **FreeList[]** or look-aside list. (Moore 2008, 22)

FreeList[0] Insert Attack – The attacker overwrites a chunk on **FreeList[0]**. A chunk is then linked into **FreeList[0]** immediately before the overwritten chunk. The address of the chunk being linked in will be written to the attacker supplied pointer in overwritten.blink. (Moore 2008, 23-25)

FreeList[0] Searching Attack – The attacker overwrites a chunk on **FreeList[0]**. This chunk provides a **fblink** value that points to a fake chunk. The next allocation request will cause this fake chunk to be returned to the application. The straightforward way to exploit this is for the attacker to choose a **fblink** value at the base of the heap that will end up returning an address in the middle of the **FreeList[]** array. (Moore 2008, 26-27)

FreeList[0] Relinking Attack – The attacker overwrites a chunk on **FreeList[0]**. This chunk provides a **fblink** value that points to a fake chunk. The next allocation request will cause the overwritten chunk to be split and returned. The remainder chunk will be inserted in the list prior to the provided fake **fblink** chunk, causing the address of the remainder chunk to be written to the fake chunk's **blink**. One potential exploitation technique is to overwrite the pointer to the front end heap manager with a pointer to the re-link chunk. (Moore 2008, 28-29)

Appendix F – 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](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



Acknowledgements

IBM Internet Security Systems would like to recognize and thank the following individuals for their efforts in both reviewing and providing feedback for this paper:

- *Marisa Mack*
- *Brett Moore*
- *Ben Hawkes*
- *Nicolas Waisman*

© Copyright IBM Corporation 2009

IBM Corporation

New Orchard Road
Armonk, NY 10504
U.S.A.

Produced in the United States of America

07-09

All Rights Reserved

IBM, the IBM logo, ibm.com, Internet Security Systems, Proventia and SiteProtector are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at ibm.com/legal/copytrade.shtml

Microsoft, Windows, Vista, Server, and XP are all trademarks or registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others. References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates.