Understanding the Low Fragmentation Heap

Blackhat USA 2010
Chris Valasek
X-Force Researcher
cvalasek@gmail.com
@nudehaberdasher

Table of Contents

Introduction

Over the years, Windows heap exploitation has continued to increase in difficulty due to the addition of exploitation counter measures along with the implementation of more complex algorithms and data structures. Due to these trends and the scarcity of comprehensive heap knowledge within the community, reliable exploitation has severely declined. Maintaining a complete understanding of the inner workings of a heap manager can be the difference between unpredictable failure and precise exploitation.

The Low Fragmentation heap has become the default front-end heap manager for the Windows operating system since the introduction of Windows Vista. This new front-end manager brought with it a different set of data structures and algorithms that replaced the **Lookaside List**. The system has also changed the way back-end memory management works as well. All of this material must be reviewed to understand the repercussions of allocating and freeing memory within an application on Windows 7.

The main goal of this paper is to familiarize the reader with the newly created logic and data structures associated with the **Low Fragmentation heap**. First, a clear and concise foundation will be provided by explaining the new data structures and their coupled purpose within the heap manager. Then detailed explanations concerning the underlying algorithms that manipulate those data structures will be discussed. Finally, some newly devised exploitation techniques will be unveiled providing practical applications from this new found knowledge.

Overview

This paper is broken into four separate sections. The first section details the core **Data Structures**, which are used throughout the heap manager to keep track of memory. A thorough understanding of these data structures is a prerequisite for understanding material presented in the remainder of the document.

The second section will discuss the newly created **Architecture** introduced in Windows Vista and carried on into Windows 7. This section shows how the use of data structures has evolved from the Windows XP code base.

The third section will divulge in-depth information on the core **Algorithms** using the Windows 7 heap. This section will detail both the **front-end** and **back-end** heap management subsystems. Understanding the material in this section will aid exploit development and provide the framework for the fourth section.

The fourth and final portion of the paper shows **Tactics** that may be employed to use the underlying heap manager to produce reliable heap manipulation, seed user supplied information, and abuse the heap meta-data in order to achieve code execution.

Prior Works

Although there may be much more information available regarding the **Low Fragmentation heap**, I am only listing the few resources that I used when doing my research. I believe this material should be viewed as prerequisite material for understanding this paper. I apologize in advance for anyone who I may have left out on this list.

- I'm still convinced Ben Hawkes had this figured out years ago. His presentation at RuxCon/Blackhat in 2008 continues to be an inspiration for my work. (Hawkes 2008)
- Nico Waisman did a tremendous job reversing Windows Vista and providing detailed information in the *libheap.py* plug-in for Immunity Debugger. (Waisman 2008).
- Brett Moore's paper *Heaps about Heaps* is, in my opinion, one of the best heap presentations ever devised. I think it will forever be used as a reference for a great deal of heap work (Moore 2007)
- Brett Moore also published material used for exploiting the link-in process for FreeList[0] in Windows XP SP2, which will be directly addressed in this paper (Moore 2005)
- Richard Johnson's presentation at ToorCon 2006 described the newly created Low Fragmentation heap for Windows Vista. This is the first (and maybe only) material that divulges detailed information about the algorithms and data structures. (Johnson 2006)
- Although David B. Probert's (Ph.D.) presentation was mainly focused on the performance benefits of the Low Fragmentation heap, it was still quite useful when attempting to understand the reasoning behind the heap implementation changes in Windows 7. (Probet)
- Adrian Marinescu gave a presentation at Blackhat 2006 on the Windows Vista heap changes. It clearly shows the reasoning for the transition away from the previous heap manager. (Marinescu 2006)
- Lastly, Lionel d'Hauenens (http://www.laboskopia.com) *Symbol Type Viewer* was an invaluable tool when analyzing the data structures used by the Windows 7 heap manager. Without it many hours might have been wasted looking for the proper structures.

Prerequisites

All of the pseudo-code and structures used in this paper were derived from 32-bit Windows 7 ntdll.dll version 6.1.7600.16385 unless otherwise stated. The structure definitions were procured from the library via the Microsoft Symbol Server using the *Symbol Type Viewer* tool and *Windbg*.

The pseudo-code representation of this code has been **heavily** edited for brevity to provide focus on the most commonly used heap management algorithms. If you feel that I have left something out or incorrectly interpreted the code please contact me at cvalasek@gmail.com. I will give you a dollar.

Terminology

Many papers have been written regarding the Windows heap, and unfortunately I have seen many different terms used throughout the material. Although the terms used in this paper may vary from those used by others, I would like to define them now for consistency throughout this document.

The term **block** or **blocks** will refer to **8-bytes** of contiguous memory. This is the unit measurement used by **heap chunk** headers when referencing their size. A **chunk** is a contiguous piece of memory that can be measured in either, **blocks** or **bytes**.

HeapBase is a pseudonym for a pointer to a **_HEAP** structure as defined by the Windows Debugging Symbols. Throughout this paper objects may be defined as being a certain offset from the **HeapBase**. Also, the **Low Fragmentation Heap** will be shortened to **LFH** in some places.

A **BlocksIndex** is another name for the **_HEAP_LIST_LOOKUP** structure. The two terms can be used interchangeably. The BlocksIndex structure that manages **Lists** holding chunks below 0x400 (1024) bytes will be referred to as the **1st BlocksIndex**, while the structure that manages **Lists** holding chunks ranging from 0x400-0x4000 (16k) will be referred to as the **2nd BlocksIndex**. Chunks larger than 16k and below the **DeCommitThreshold** and **0xFE00** blocks will be managed in *FreeList[0]-like* structure [Discussed later in this paper].

The concept of having **Dedicated FreeLists** has disappeared. The term **ListHint** or **FreeList** will be used when referring to a list which points into the Heap->FreeLists at a specific location. This will be discussed in further detail in the *Architecture* section.

Finally the term **HeapBin, Bin, or UserBlock** will be used when referring to memory allocated from the **Low Fragmentation heap** for a certain size. I know most call this a **HeapBucket** but I will refrain from doing so to reduce confusion since Microsoft's debugging symbols use the name **_HEAP_BUCKET** for a 0x4 byte data structure that is used to designate a size but not used for memory containment.

Notes

This paper is meant to serve as subsequent knowledge to the work that John McDonald and I completed for Blackhat USA 2009. Any information regarding the inner workings of doubly linked lists, Lookaside lists, etc can be found in the paper entitled Practical Windows XP/2003 Exploitation (McDonald/Valasek 2009).

Data Structures

These data structures were derived from the Windows Debugging Symbols for ntdll.dll version 6.1.7600.16385. They are used to keep track of memory within the manager, providing the user seamless access to virtual memory through abstracted function calls; mainly HeapAlloc(), HeapFree(), malloc() and free().

_HEAP

(HeapBase)

Each heap that gets created starts with an integral structure called the **HeapBase**. The HeapBase contains many vital values and pointers to structures used by the heap manager. This is the heart of every heap and must sustain its integrity to provide reliable allocation and free operations. If you are familiar with the HeapBase used in the Windows XP code base, this will look quite similar; yet some of the bolded items need further explanation. The following shows the contents for a **_HEAP** structure in 32-bit Windows 7 Service Pack 0:

Listing 1. _HEAP via windbg

```
0:001> dt _HEAP
ntdll!_HEAP
   +0x000 Entry          : _HEAP_ENTRY
   +0x008 SegmentSignature : Uint4B
   +0x00c SegmentFlags     : Uint4B
   +0x010 SegmentListEntry : _LIST_ENTRY
   +0x018 Heap           : Ptr32 _HEAP
   +0x01c BaseAddress      : Ptr32 Void
   +0x020 NumberOfPages    : Uint4B
   +0x024 FirstEntry       : Ptr32 _HEAP_ENTRY
   +0x028 LastValidEntry   : Ptr32 _HEAP_ENTRY
   +0x02c NumberOfUnCommittedPages : Uint4B
   +0x030 NumberOfUnCommittedRanges : Uint4B
   +0x034 SegmentAllocatorBackTraceIndex : Uint2B
   +0x036 Reserved         : Uint2B
   +0x038 UCRSegmentList   : _LIST_ENTRY
   +0x040 Flags          : Uint4B
   +0x044 ForceFlags       : Uint4B
```

```
+0x048 CompatibilityFlags : Uint4B
+0x04c EncodeFlagMask   : Uint4B
+0x050 Encoding         : _HEAP_ENTRY
+0x058 PointerKey       : Uint4B
+0x05c Interceptor      : Uint4B
+0x060 VirtualMemoryThreshold : Uint4B
+0x064 Signature        : Uint4B
+0x068 SegmentReserve   : Uint4B
+0x06c SegmentCommit    : Uint4B
+0x070 DeCommitFreeBlockThreshold : Uint4B
+0x074 DeCommitTotalFreeThreshold : Uint4B
+0x078 TotalFreeSize    : Uint4B
+0x07c MaximumAllocationSize : Uint4B
+0x080 ProcessHeapsListIndex : Uint2B
+0x082 HeaderValidateLength : Uint2B
+0x084 HeaderValidateCopy : Ptr32 Void
+0x088 NextAvailableTagIndex : Uint2B
+0x08a MaximumTagIndex  : Uint2B
+0x08c TagEntries       : Ptr32 _HEAP_TAG_ENTRY
+0x090 UCRList          : _LIST_ENTRY
+0x098 AlignRound       : Uint4B
+0x09c AlignMask        : Uint4B
+0x0a0 VirtualAllocdBlocks : _LIST_ENTRY
+0x0a8 SegmentList      : _LIST_ENTRY
+0x0b0 AllocatorBackTraceIndex : Uint2B
+0x0b4 NonDedicatedListLength : Uint4B
+0x0b8 BlocksIndex      : Ptr32 Void
+0x0bc UCRIndex         : Ptr32 Void
+0x0c0 PseudoTagEntries : Ptr32 _HEAP_PSEUDO_TAG_ENTRY
+0x0c4 FreeLists        : _LIST_ENTRY
+0x0cc LockVariable     : Ptr32 _HEAP_LOCK
+0x0d0 CommitRoutine    : Ptr32     long
+0x0d4 FrontEndHeap     : Ptr32 Void
+0x0d8 FrontHeapLockCount : Uint2B
+0x0da FrontEndHeapType : UChar
+0x0dc Counters         : _HEAP_COUNTERS
+0x130 TuningParameters : _HEAP_TUNING_PARAMETERS
```

- **EncodeFlagMask** – A value that is used to determine if a heap chunk **header** is encoded. This value is initially set to 0x100000 by **RtlpCreateHeapEncoding()** in **RtlCreateHeap()**.
- **Encoding** – Used in an XOR operation to encode the chunk **headers**, preventing predictable meta-data corruption.
- **BlocksIndex** – This is a **_HEAP_LIST_LOOKUP** structure that is used for a variety of purposes. Due to its importance, it will be discussed in greater detail later in this document.

- **FreeLists** – A special linked-list that contains pointers to ALL of the free chunks for this heap. It can almost be thought of as a **heap cache**, but for chunks of every size (and no single associated bitmap).
- **FrontEndHeapType** – An integer is initially set to 0x0, and is subsequently assigned a value of 0x2, indicating the use of a **LFH**. Note: Windows 7 does not actually have support for using **Lookaside Lists**.
- **FrontEndHeap** – A pointer to the associated front-end heap. This will either be NULL or a pointer to a _LFH_HEAP structure when running under Windows 7.

_HEAP_LIST_LOOKUP

(HeapBase->BlocksIndex)

Understanding the **_HEAP_LIST_LOOKUP** structure is one of the most important tasks in building a solid foundation of Windows 7 heap management. It is the keystone of allocations and frees used by the **back-end manager** and the **front-end manager**. Under *normal* conditions the 1st **_HEAP_LIST_LOOKUP** structure, which is initialized in **RtlCreateHeap()**, will be located at an offset of +0x150 from the **HeapBase**.

Listing 2. _HEAP_LIST_LOOKUP via windbg

```
0:001> dt _HEAP_LIST_LOOKUP
ntdll!_HEAP_LIST_LOOKUP
  +0x000 ExtendedLookup   : Ptr32 _HEAP_LIST_LOOKUP
  +0x004 ArraySize        : Uint4B
  +0x008 ExtraItem        : Uint4B
  +0x00c ItemCount        : Uint4B
  +0x010 OutOfRangeItems  : Uint4B
  +0x014 BaseIndex        : Uint4B
  +0x018 ListHead         : Ptr32 _LIST_ENTRY
  +0x01c ListsInUseUlong  : Ptr32 Uint4B
  +0x020 ListHints        : Ptr32 Ptr32 _LIST_ENTRY
```

- **ExtendedLookup** - A pointer to the next **_HEAP_LIST_LOOKUP** structure. The value is NULL if there is no ExtendedLookup.
- **ArraySize** – The highest **block** size that this structure will track, otherwise storing it in a *special* **ListHint**. The only two sizes that Windows 7 currently uses are **0x80** and **0x800**.
- **OutOfRangeItems** – This 4-byte value counts the number of items in the *FreeList[0]-like* structure. Each **_HEAP_LIST_LOOKUP** tracks free chunks larger than **ArraySize-1** in **ListHint[ArraySize-BaseIndex-1]**.
- **BaseIndex** – Used to find the relative offset into the **ListHints** array, since each _HEAP_LIST_LOOKUP is designated for a certain size. For example, the BaseIndex for 1st BlocksIndex would be 0x0 because it manages lists for chunks from **0x0 – 0x80**, while the 2nd BlocksIndex would have a BaseIndex of **0x80**.
- **ListHead** – This points to the same location as **HeapBase->FreeLists**, which is a linked list of all the free chunks available to a heap.

- **ListsInUseUlong** – Formally known as the **FreeListInUseBitmap**, this 4-byte integer is an optimization used to determine which **ListHints** have available chunks.
- **ListHints** – Also known as **FreeLists**, these linked lists provide pointers to free chunks of memory, while also serving another purpose. If the **LFH** is enabled for a given **Bucket** size, then the **blink** of a specifically sized **ListHint**/**FreeList** will contain the address of a **_HEAP_BUCKET** + 1.

_LFH_HEAP

(HeapBase->FrontEndHeap)

The **Low Fragmentation heap** is managed by this data structure. When activated, it will let the heap manager know what sizes it is capable of managing, along with keeping caches of previously used chunks. Although the **BlocksIndex** is capable of tracking chunks that are over **0x800** blocks in length, the **LFH** will only be used for chunks that are less than **16k**.

Listing 3. _LFH_HEAP via windbg

```
0:001> dt _LFH_HEAP
ntdll!_LFH_HEAP
   +0x000 Lock          : _RTL_CRITICAL_SECTION
   +0x018 SubSegmentZones  : _LIST_ENTRY
   +0x020 ZoneBlockSize    : Uint4B
   +0x024 Heap          : Ptr32 Void
   +0x028 SegmentChange    : Uint4B
   +0x02c SegmentCreate    : Uint4B
   +0x030 SegmentInsertInFree : Uint4B
   +0x034 SegmentDelete    : Uint4B
   +0x038 CacheAllocs      : Uint4B
   +0x03c CacheFrees       : Uint4B
   +0x040 SizeInCache      : Uint4B
   +0x048 RunInfo        : _HEAP_BUCKET_RUN_INFO
   +0x050 UserBlockCache   : [12] _USER_MEMORY_CACHE_ENTRY
   +0x110 Buckets        : [128] _HEAP_BUCKET
   +0x310 LocalData      : [1] _HEAP_LOCAL_DATA
```

- **Heap** – A pointer to the parent heap of this **LFH**.
- **UserBlockCache** – Although this won't be thoroughly discussed, it's worth mentioning that the **UserBlockCache** array keeps track of previously used memory chunks for future allocations.
- **Buckets** – An array of 0x4 byte data structures that are used for the sole purpose of keeping track of indices and sizes. This is why the term **Bin** will be used to describe the area of memory used to fulfill request for a certain **Bucket**.
- **LocalData** – This is a pointer to a large data structure which holds information about each **SubSegment**. See _HEAP_LOCAL_DATA for more information.

_LFH_BLOCK_ZONE

(HeapBase->FrontEndHeap->LocalData->CrtZone)

This data structure is used to keep track of locations in memory that are used to service allocation requests. These pointers are setup on the first request serviced by the **LFH** or after the pointer list have been exhausted.

Listing 4. _LFH_BLOCK_ZONE via windbg

```
0:000> dt _LFH_BLOCK_ZONE
ntdll!_LFH_BLOCK_ZONE
  +0x000 ListEntry      : _LIST_ENTRY
  +0x008 FreePointer    : Ptr32 Void
  +0x00c Limit          : Ptr32 Void
```

- **ListEntry** – A linked list of **_LFH_BLOCK_ZONE** structures.
- **FreePointer** – This will hold a pointer to memory that can be used by a **_HEAP_SUBSEGMENT**.
- **Limit** – The last **_LFH_BLOCK_ZONE** structure in the list. When this value is reached or exceeded, the **back-end** heap will be used to create more **_LFH_BLOCK_ZONE** structures.

_HEAP_LOCAL_DATA

(HeapBase->FrontEndHeap->LocalData)

A key structure that provides **_HEAP_LOCAL_SEGMENT_INFO** instances to the **Low Fragmentation heap**.

Listing 5. _HEAP_LOCAL_DATA via windbg

```
0:000> dt _HEAP_LOCAL_DATA
ntdll!_HEAP_LOCAL_DATA
  +0x000 DeletedSubSegments : _SLIST_HEADER
  +0x008 CrtZone        : Ptr32 _LFH_BLOCK_ZONE
  +0x00c LowFragHeap    : Ptr32 _LFH_HEAP
  +0x010 Sequence       : Uint4B
  +0x018 SegmentInfo    : [128] _HEAP_LOCAL_SEGMENT_INFO
```

- **LowFragHeap** – The **Low Fragmentation heap** associated with this structure.
- **SegmentInfo** – An array of **_HEAP_LOCAL_SEGMENT_INFO** structures representing all available sizes for this **LFH**. See _HEAP_LOCAL_SEGMENT_INFO for more information.

_HEAP_LOCAL_SEGMENT_INFO

(HeapBase->FrontEndHeap->LocalData->SegmentInfo[])

The size of the request to be serviced will determine which **_HEAP_LOCAL_SEGMENT_INFO** structure is used. This structure holds information that the heap algorithms use when determining the most efficient way to allocate and free memory. Although there are only 128 of these structures in **_HEAP_LOCAL_DATA**, all **8-byte aligned** sizes below **16k** have a corresponding _HEAP_LOCAL_SEGMENT_INFO. A special algorithm calculates a relative index so that each **Bucket** is guaranteed a dedicated structure.

Listing 6. _HEAP_LOCAL_SEGMENT_INFO via windbg

```
0:000> dt _HEAP_LOCAL_SEGMENT_INFO
ntdll!_HEAP_LOCAL_SEGMENT_INFO
  +0x000 Hint          : Ptr32 _HEAP_SUBSEGMENT
  +0x004 ActiveSubsegment : Ptr32 _HEAP_SUBSEGMENT
  +0x008 CachedItems     : [16] Ptr32 _HEAP_SUBSEGMENT
  +0x048 SListHeader     : _SLIST_HEADER
  +0x050 Counters        : _HEAP_BUCKET_COUNTERS
  +0x058 LocalData       : Ptr32 _HEAP_LOCAL_DATA
  +0x05c LastOpSequence   : Uint4B
  +0x060 BucketIndex     : Uint2B
  +0x062 LastUsed        : Uint2B
```

- **Hint** – This **SubSegment** is only set when the **LFH** frees a chunk which it is managing. If a chunk is never freed, this value will always be **NULL**.
- **ActiveSubsegment** – The **SubSegment** used for most memory requests. While initially NULL, it is set on the **first** allocation for a specific size.
- **LocalData** – The **_HEAP_LOCAL_DATA** structure associated with this structure.
- **BucketIndex** – Each **SegmentInfo** object is related to a certain **Bucket** size (or Index).

_HEAP_SUBSEGMENT

(HeapBase->FrontEndHeap->LocalData->SegmentInfo[]->Hint,ActiveSubsegment,CachedItems)

After the appropriate structures are indentified for a specific **_HEAP_BUCKET**, the front-end manager will perform a free or allocation. Since the **LFH** can be thought of as a heap manager inside a heap manager, it makes sense that the **_HEAP_SUBSEGMENT** is used to keep track of how much memory is available and how it should be distributed.

Listing 7. _HEAP_SUBSEGMENT via windbg

```
0:000> dt _HEAP_SUBSEGMENT
ntdll!_HEAP_SUBSEGMENT
  +0x000 LocalInfo       : Ptr32 _HEAP_LOCAL_SEGMENT_INFO
```

```
+0x004 UserBlocks      : Ptr32 _HEAP_USERDATA_HEADER
+0x008 AggregateExchg  : _INTERLOCK_SEQ
+0x010 BlockSize       : Uint2B
+0x012 Flags           : Uint2B
+0x014 BlockCount      : Uint2B
+0x016 SizeIndex       : UChar
+0x017 AffinityIndex   : UChar
+0x010 Alignment       : [2] Uint4B
+0x018 SFreeListEntry  : _SINGLE_LIST_ENTRY
+0x01c Lock            : Uint4B
```

- **LocalInfo** – The **_HEAP_LOCAL_SEGMENT_INFO** structure associated with this structure**.**
- **UserBlocks** – A **_HEAP_USERDATA_HEADER** structure coupled with this **SubSegment** which holds a large chunk of memory split into n-number of chunks.
- **AggregateExchg** – An **_INTERLOCK_SEQ** structure used to keep track of the current **Offset** and **Depth**.
- **SizeIndex** – The **_HEAP_BUCKET SizeIndex** for this **SubSegment**.

_HEAP_USERDATA_HEADER

(HeapBase->FrontEndHeap->LocalData->SegmentInfo[]->Hint,ActiveSubsegment,CachedItems->UserBlocks)

This header precedes the **UserBlock** chunk that is used to service all requests for the **Low Fragmentation heap**. After all the logic is performed to locate a **SubSegment**, this structure is where **committed** memory is actually manipulated.

Listing 8. _HEAP_USERDATA_HEADER via windbg

```
0:000> dt _HEAP_USERDATA_HEADER
ntdll!_HEAP_USERDATA_HEADER
  +0x000 SFreeListEntry  : _SINGLE_LIST_ENTRY
  +0x000 SubSegment      : Ptr32 _HEAP_SUBSEGMENT
  +0x004 Reserved        : Ptr32 Void
  +0x008 SizeIndex       : Uint4B
  +0x00c Signature       : Uint4
```

_INTERLOCK_SEQ

(HeapBase->FrontEndHeap->LocalData->SegmentInfo[]->Hint,ActiveSubsegment,CachedItems->AggregateExchg)

Due to the way the **UserBlock** chunks are divided, there needs to be a way to get the current offset into it; for freeing or allocating the next chunk. This process is handled by the **_INTERLOCK_SEQ** data structure.

Listing 9. _INTERLOCK_SEQ via windbg

```
0:000> dt _INTERLOCK_SEQ
ntdll!_INTERLOCK_SEQ
   +0x000 Depth          : Uint2B
   +0x002 FreeEntryOffset : Uint2B
   +0x000 OffsetAndDepth  : Uint4B
   +0x004 Sequence       : Uint4B
   +0x000 Exchg          : Int8B
```

- **Depth** – A counter that keeps track of how many chunks are left in a **UserBlock**. This number is **incremented** on a free and **decremented** on an allocation. Its value is initialized to the size of **UserBlock** divided by the **HeapBucket** size.
- **FreeEntryOffset** – This 2-byte integer holds a value, when added to the address of the **_HEAP_USERDATA_HEADER**, results in a pointer to the next location for freeing or allocating memory. This value is represented in **blocks** (0x8 byte chunks) and is initialized to 0x2, as **sizeof(_HEAP_USERDATA_HEADER)** equals 0x10. [0x2 * 0x8 == 0x10].
- **OffsetAndDepth** – Since both **Depth** and **FreeEntryOffset** are 2-bytes, they can be combined into this single 4-byte value.

_HEAP_ENTRY

(Chunk Header)

The **_HEAP_ENTRY**, also known as the heap chunk **header**, is an 8-byte value stored before every chunk of memory in the heap (even the chunks inside the **UserBlocks**). It has changed quite drastically since the Windows XP code base due to the modifications in header validation and security introduced in newer versions of Windows.

Listing 10. _HEAP_ENTRY via Windbg

```
0:001> dt _HEAP_ENTRY
ntdll!_HEAP_ENTRY
   +0x000 Size          : Uint2B
   +0x002 Flags         : UChar
   +0x003 SmallTagIndex    : Uchar
   +0x000 SubSegmentCode   : Ptr32 Void
   +0x004 PreviousSize     : Uint2B
   +0x006 SegmentOffset    : Uchar
   +0x006 LFHFlags         : Uchar
   +0x007 UnusedBytes      : Uchar
   +0x000 FunctionIndex    : Uint2B
   +0x002 ContextValue     : Uint2B
   +0x000 InterceptorValue : Uint4B
   +0x004 UnusedBytesLength : Uint2B
   +0x006 EntryOffset      : Uchar
   +0x007 ExtendedBlockSignature : Uchar
```

- **Size** – The size, in blocks, of the chunk. This includes the _HEAP_ENTRY itself.
- **Flags** – Flags denoting the state of this heap chunk. Some examples are FREE or BUSY.
- **SmallTagIndex** – This value will hold the XOR'ed checksum of the first three bytes of the _HEAP_ENTRY.
- **UnusedBytes/ExtendedBlockSignature** – A value used to hold the unused bytes or a byte indicating the state of the chunk being managed by the **LFH**.

Overview

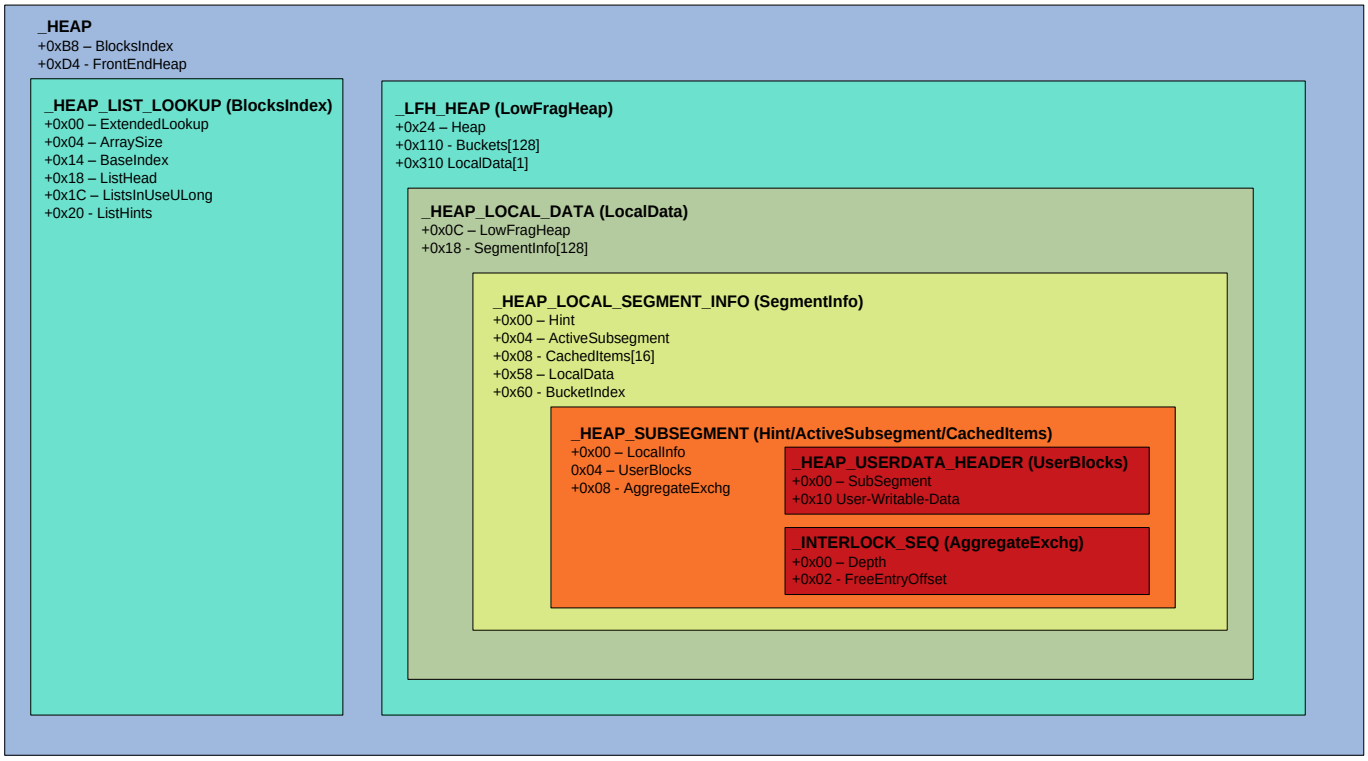

Diagram 1. Data structure overview

Architecture

The Windows 7 heap manager has changed quite drastically since the Windows XP days, so it is necessary to go over some brief architectural adjustments. Specifically, the way FreeLists work has been redesigned along with how data is stored in them needs explanation.

FreeLists

Before we can talk about the core algorithms, the current and previous **FreeList** structure must be examined. This is because the way the FreeLists operate and store data has changed since the Windows XP code base. This is an overview of the FreeList structure as defined by John McDonald and myself in a previous paper:

> There are separate lists for each possible block size below 1024 bytes, giving a total of 128 free lists (heap blocks are sized in multiples of 8.) Each doubly-linked free list has a sentinel head node located in the array at the base of the heap. Each head node contains two pointers: a forward link (FLink), and a back link (BLink). FreeList[1] is unused, and FreeList[2] through FreeList[127] are called the dedicated free lists. For these dedicated lists, all of the free blocks in the list are the same size, which corresponds to the array index * 8. All blocks higher than or equal to size 1024, however, are kept in a single free list at FreeList[0]. (This slot was available for use because there aren't any free blocks of size 0.) The free blocks in this list are sorted from the smallest block to the largest block. So, FreeList[0].Flink points to the smallest free block (of size>=1024), and FreeList[0].Blink points to the largest free block (of size>=1024.)

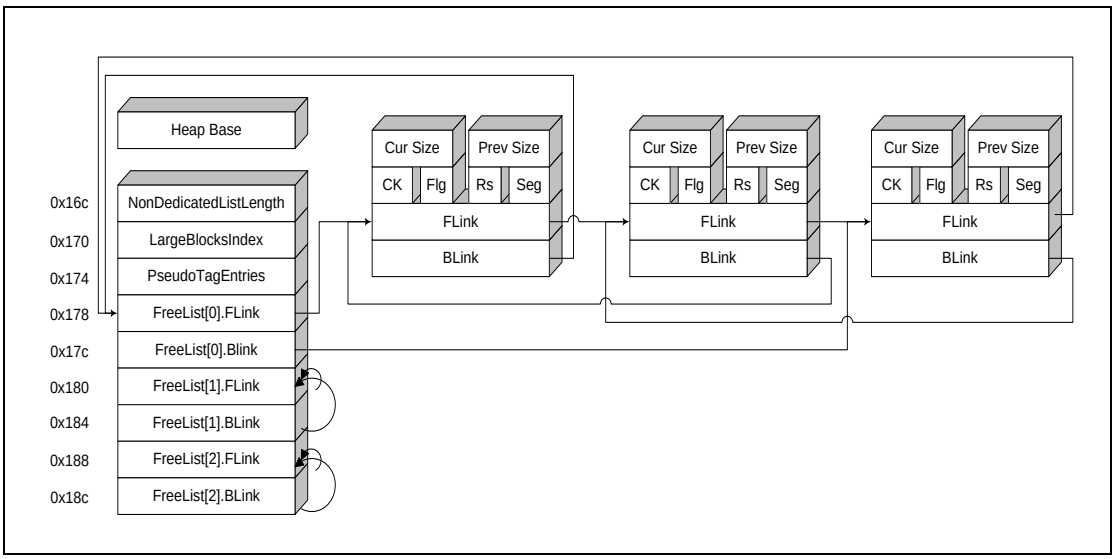

Diagram 2. Windows XP FreeList relationships

Since the **LFH** changed the way the front-end manager worked, the back-end manager had to adapt as well. There is not a single **Dedicated FreeList** now, instead each **BlocksIndex** maintains its own **ListHints**, which are initialized to NULL.

The **BlocksIndex** structure holds a pointer to what is referred to as the **ListHints**, which point into the **FreeLists** structure. It is setup quite similar to the old **FreeLists** only the 0^{th} position of the linked list is no longer used for chunks larger than 0x400 (1024) bytes; instead conditionals are required. If there is no **BlocksIndex->ExtendedLookup** then all chunks of size greater than or equal to **BlocksIndex->ArraySize - 1** will be stored in ascending order in **FreeList[ArraySize-BaseIndex – 1]**.

Although the FreeLists contain **sentinel nodes** at a calculated offset from the ListHints pointer that is where the most of the similarities end. While the **Flink** pointer still points to the next available chunk in a **FreeList**, it can span into larger Freelists as well. This lets the **Heap.FreeLists** traverse **every** free chunk available for a particular **heap**.

The **Blink** in the sentinel node has changed as well; serving a dual purpose. If the **LFH** is not enabled for a **Bucket**, then the **sentinel Blink** will hold a counter used in an **allocation heuristic**. Otherwise, it will contain the address of a **_HEAP_BUCKET + 1** (Except for the Case of ListHint[ArraySize-BaseIndex-1]).

The following diagram is an example of a sparsely populated heap to show how these new constructs interact with each other. It contains a single **BlocksIndex** that is designed to track chunks below 1024 bytes in size. There are only **five** chunks associated with this heap and they can be accessed in a variety of ways.

For example, if a request came in for an allocation of 0x30 (48) bytes the heap would attempt to use the **ListHint[0x6]**. You can see that that although there are only three chunks available for size 0x30, the **Flink** in the last free chunk for size 0x30 points to an entry that belongs to **FreeList[0x7]**. FreeList[0x7] has one entry, but like **FreeList[0x6]**, its last chunk points across the size boundary into a larger chunk.

This changes the way that list termination is done. Instead of the last node in the list pointing to its **sentinel node** on a **Dedicated FreeList**, it points to the **FreeLists** entry at **+0xC4** from the **HeapBase**.

> **Note:** When the **_HEAP_LIST_LOOKUP** structures are initialized, either in **RtlCreateHeap()** or **RtlpExtendListLookup**, the **ListHead** is set to point at **Heap.FreLists [0xC4** from **HeapBase]**. This makes both entries identical and pointing to the same region in memory.

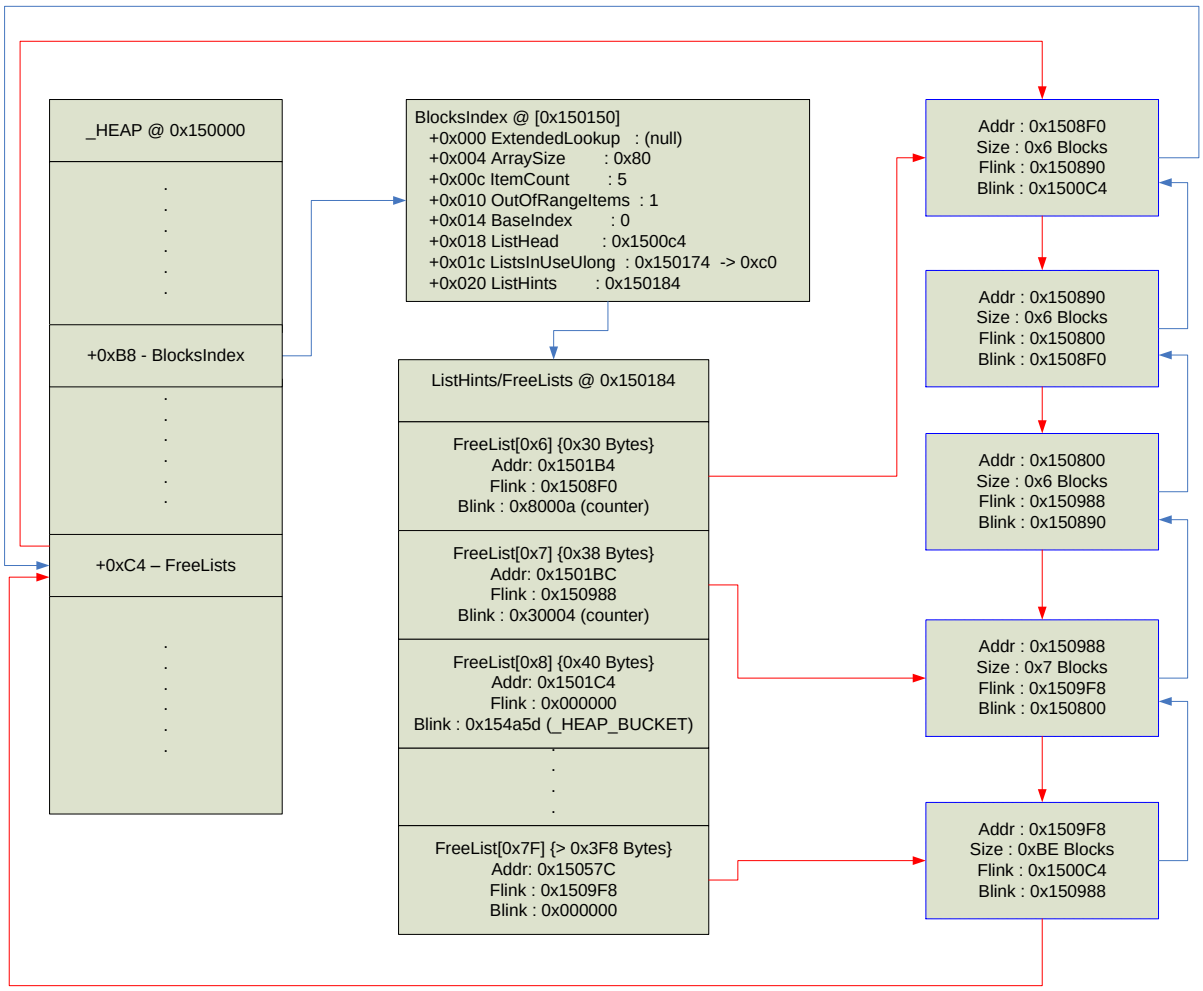

Diagram 3. New FreeList relationship

Algorithms

A basic foundation of knowledge must be laid to fully understand heap determinism and exploitation. Without this core information one can only *pray-after-free*. This section will break down the core algorithms into two parts, allocation and freeing; divided into back-end and front-end. This is done because memory manipulation performed by the front or back end can affect the state of the other.

Allocation

Allocations start at **RtlAllocateHeap()** when attempting to service requests from a calling application. The function starts by 8-byte aligning the desired allocation amount. It will then acquire an **index** into the **ListHints**. If no specific index is located, **BlocksIndex->ArraySize-1** will be used.

Listing 11. RtlAllocateHeap BlocksIndex Search

```
if(Size == 0x0)
	Size = 0x1;

//ensure that this number is 8-byte aligned
int RoundSize = Round(Size);

int BlockSize = Size / 8;

//get the HeapListLookup, which determines if we should use the LFH
_HEAP_LIST_LOOKUP *BlocksIndex = (_HEAP_LIST_LOOKUP*)heap->BlocksIndex;

//loop through the HeapListLookup structures to determine which one to use
while(BlocksSize >= BlocksIndex->ArraySize)
{
	if(BlocksIndex->ExtendedLookup == NULL)
	{
		BlocksSize = BlocksIndex->ArraySize - 1;
		break;
	}

	BlocksIndex = BlocksIndex->ExtendedLookup;
}
```

There exists a condition where the search will return a ListHint **index** of **BlocksIndex->ArraySize-1**. If this condition occurs then the **back-end** allocator will be used with a **FreeList** value of **NULL**. This will cause the back-end allocator to attempt to use the **Heap->FreeLists**. If the **FreeLists** do not contain a sufficiently sized chunk, the heap will be extended via **RtlpExtendHeap()**.

If a specific **index** has been successfully acquired, then the heap manager will attempt to use the **FreeList** for the requested size. It will then proceed to look at the **FreeList->Blink** to determine if the **Low Fragmentation heap** is active for that Bucket; otherwise, the manager will default to using the back-end:

Listing 12. RtlAllocateHeap heap manager selector

```
//get the appropriate freelist to use based on size
int FreeListIndex = BlockSize - HeapListLookup->BaseIndex;

_LIST_ENTRY *FreeList = &HeapListLookup->ListHints[FreeListIndex];
if(FreeList)
{
	//check FreeList[index]->Blink to see if the heap bucket
	//context has been populated via RtlpGetLFHContext()
	//RtlpGetLFHContext() stores the HeapBucket
	//context + 1 in the Blink
	_HEAP_BUCKET *HeapBucket = FreeList->Blink;

	if(HeapBucket & 1)
	{
		RetChunk = RtlpLowFragHeapAllocFromContext(HeapBucket-1, aBytes);

		if(RetChunk && heap->Flags == HEAP_ZERO_MEMORY)
			memset(RetChunk, 0, RoundSize);
	}
}

//if the front-end allocator did not succeed, use the back-end
if(!RetChunk)
{
	RetChunk = RtlpAllocateHeap(heap, Flags | 2, Size, RoundSize, FreeList);
}
```

Back-end Allocation

The back-end allocator is the last line in allocations; if it fails, the request for memory will not occur, returning NULL. Along with the responsibilities of servicing memory requests unable to be serviced by the front-end, the back-end also **activates** the front-end allocator based on activation heuristics. It can be thought of as being very similar to the way the **heap cache** heuristic worked in the Window XP code base.

RtlpAllocateHeap

A **_HEAP** structure, the size to allocate, and the desired **ListHint (FreeList)** are a few of the arguments passed to **RtlpAllocateHeap()**. Like **RtlAllocateHeap**, the first order of business is to round the requested size to the nearest 8-byte aligned value and determining if the **Flags** have the HEAP_NO_SERIALIZE bit set. If this bit is set, the **LFH** will not be enabled. (http://msdn.microsoft.com/en-us/library/aa366599%28v=VS.85%29.aspx)

Listing 13. RtlpAllocateHeap size rounding and Heap maintenance

```
int RoundSize;

//if the FreeList isn't NULL, the rounding has already
//been preformed
if(FreeList)
{
	RoundSize = RoundSize;
}
```

```
else
{
	int MinSize = 1;
	if(Size)
		MinSize = Size;

	//rounds to the nearest 8-byte aligned number
	RoundSize = Round(MinSize);
}

int SizeInBlocks = RoundSize / 8;

if(SizeInBlocks < 2)
{
	//RoundSize += sizeof(_HEAP_ENTRY)
	RoundSize = RoundSize + 8;
	SizeInBlocks = 2;
}

//if NOT HEAP_NO_SERIALIZE, use locking mechanisms
//LFH CANNOT be enabled if this path isn't taken
if(!(Flags & HEAP_NO_SERIALIZE))
{
	//setup locking mechanisms here

	//if we have certain compaitibilityflags
	//(either set below, or otherwise, then
	//we will call 'RtlpPerformHeapMaintenance'
	//which will activate the LFH and
	//setup an ExtendedListLookup as well
	if(Heap->CompatibilityFlags & 0x60000000)
		RtlpPerformHeapMaintenance(Heap);
}
```

Note: You will see later how the **CompatibilityFlags** are set in subsequent code. This is how the **Low Fragmentation** heap is activated. Although the LFH is the front-end manager by default it doesn't actually handle any of the memory management until a certain heuristic is triggered.

Even though the **LFH** may be ready to service requests at this point, the back-end allocator will continue for this allocation. By leaving out the code for handling Virtual Memory requests, it can be seen that **RtlpAllocateHeap()** will attempt to see if the **FreeList** argument is non-NULL. Pending a valid FreeList argument, the back-end manager will apply the heuristic used to determine if the **LFH** should be used for any later allocations:

Listing 14. RtlpAllocateHeap LFH Heuristic

```
if(FreeList != NULL)
{
	//if this freelist doesn't hold a _HEAP_BUCKET
	//update the counters and attempt to get the LFH context
	if(!(FreeList->Blink & 1))
	{
		//add a certain amount to the blink
		FreeList->Blink += 0x10002;

		//if the counter has ran more than 0x10 times OR
		//we haven't successfully entered the critical section OR
		//we've been through 0x1000 iterations
```

```
                //then attempt to set the Compatibility flags
                //(which, in turn, will call RtlpPerformHeapMaintenance())
                if(WORD)FreeList->Blink > 0x20 || FreeList->Blink > 0x10000000)
                {
                        //if the FrontEndHeapType is LFH (0x2) assign it
                        int FrontEndHeap;
                        if(Heap->FrontEndHeapType == 0x2)
                                FrontEndHeap = Heap->FrontEndHeap;
                        else
                                FrontEndHeap = NULL;

                        //this function gets a _HEAP_BUCKET
                        //stored in _LFH_HEAP->Bucket[BucketSize]
                        //if the LFH hasn't been activated yet, it will return NULL
                        char *LFHContext = RtlpGetLFHContext(FrontEndHeap, Size);

                        //if the context isn't set AND
                        //we've seen 0x10+ allocations, set the flags
                        if(LFHContext == NULL)
                        {
                                if((WORD)FreeList->Blink > 0x20)
                                {
                                        //RtlpPerformHeapMaintenance heurstic
                                        if(Heap->FrontEndHeapType == NULL)
                                                Heap->CompatibilityFlags |= 0x20000000;
                                }
                        }
                        else
                        {
                                //save the _HEAP_BUCKET in the Blink
                                //+1 == _HEAP_BUCKET
                                FreeList->Blink = LFHContext + 1;
                        }
                }
        }
}
```

Note: This is why I stated that the front and back end managers were so closely related. As you can see the **ListHint** is used to store the address of a **_HEAP_BUCKET**, which in turn is used to determine if the manager should use the LFH. This dual use may seem confusing now, but it will become clearer after the front-end allocation and free algorithms are discussed.

Now that the LFH activation flags have been set, the allocation can continue its course on the back-end. The **FreeList** is checked to see if it is populated, then a safe unlink check is performed. This ensures FreeList values maintain their integrity to prevent 4-byte overwrites when unlinking. The **ListsInUseUlong** (FreeListInUseBitmap) is then updated accordingly. Finally, the chunk that has been unlinked from the list has its header updated to show that it is BUSY and **returned**.

Listing 15. RtlpAllocateHeap ListHint allocation

```
//attempt to use the Flink
if(FreeList != NULL && FreeList->Flink != NULL)
{
	int SizeToUse;

	//saved values
	_HEAP_ENTRY *Blink = FreeList->Blink;
	_HEAP_ENTRY *Flink = FreeList->Flink;

	//get the heap chunk header by subtracting 8
	_HEAP_ENTRY *ChunkToUseHeader = Flink - 8;

	//decode the header if applicable
	if(Heap->EncodeFlagMask)
		DecodeAndValidateChecksum(ChunkToUseHeader);

	//ensure safe unlinking before acquiring this chunk for use
	if(Blink->Flink != Flink->Blink ||
		Blink->Flink != FreeList)
	{
		RtlpLogHeapFailure();
		//XXX RtlNtStatusToDosError and return
	}

	//decrement the total heap size
	Heap->TotalFreeSize -= ChunkToUseHeader->Size;

	//iterate through the BlocksIndex structures
	//If no sufficient BlocksIndex is found, use BlocksIndex->ArraySize - 1
	_HEAP_LIST_LOOKUP *BlocksIndex = Heap->BlocksIndex;
	if(BlocksIndex)
	{
		int ListHintIndex = GetListHintIndex(BlocksIndex, Size);
		int RelativeOffset = ListHintIndex - BlocksIndex->BaseIndex;

		//if there are more of the same size
		//don't update the bitmap
		if(Flink->Flink != BlocksIndex->ListHead &&
			Flink.Size == Flink->Flink.Size)
		{
			BlocksIndex->ListHints[FreeListOffset] =
				Flink->Flink;
		}
		else
		{
			BlocksIndex->ListHints[FreeListOffset] = NULL;
			BlocksIndex->ListsInUseUlong[RelativeOffset >> 5] &=
				~(1 << RelativeOffset & 0x1F);
		}
	}

	//unlink the current chunk to be allocated
	Blink->Flink = Flink;
	Flink->Blink = Blink;
}
```

Note: Although it will be talked about later, it should be noted that the updating of the bitmap no longer uses the XOR operation, but instead, the bitwise AND. This prevents the 1-byte FreeListInUseBitmap flipping attack from working (McDonald/Valasek 2009).

If the **ListHint** is **not** able to provide the application with the memory requested then back-end heap manager proceeds to use the **Heap->FreeLists**. The **FreeLists** contain all the free chunks for a heap. If a chunk of sufficient size is found it will be split if necessary and returned to the user. Otherwise, the heap will have to be extended via **RtlpExtendHeap()**:

Listing 16. RtlpAllocateHeap Heap->FreeLists allocation

```
//attempt to use the FreeLists
_LIST_ENTRY *HeapFreeLists = &Heap->FreeLists;
_HEAP_LIST_LOOKUP *BlocksIndex = Heap->BlocksIndex;
_LIST_ENTRY *ChunkToUse;

//find an appropriate chunk on the FreeLists
_HEAP_LIST_LOOKUP *CurrBlocksIndex = BlocksIndex;

while(1)
{
	//if we've ran out of structures
	//abort and we'll extend the heap
	if(CurrBlocksIndex == NULL)
	{
		ChunkToUse = CurrBlocksIndex->ListHead;
		break;
	}

	//remember that ListHead and HeapFreeLists
	//point to the same location
	CurrListHead = CurrBlocksIndex->ListHead;

	//if we've came upon an empty FreeList extend the heap
	if(CurrListHead == CurrListHead->Blink)
	{
		ChunkToUse = CurrListHead;
		break;
	}

	_HEAP_ENTRY *BlinkHeader = (CurrListHead->Blink - 8);

	//if the chunk is encoded decode it
	if(Heap->EncodeFlagMask &&
		Heap->EncodeFlagMask & BlinkHeader)
	{
		DecodeHeader(BlinkHeader);
	}

	//if the chunk can't be serviced by the
	//largest chunk extend the heap
	if(SizeInBlocks > BlinkHeader->Size)
	{
		ChunkToUse = CurrListHead;
		break;
	}

	_HEAP_ENTRY *FlinkHeader = CurrListHead->Flink-8;
	//if the chunk is encoded decode it
```

```
        if(Heap->EncodeFlagMask &&
                Heap->EncodeFlagMask & FlinkHeader)
        {
                DecodeHeader(FlinkHeader);
        }

        //if the first chunk is sufficient use it
        //otherwise loop through the rest
        if(FlinkHeader->Size >= SizeInBlocks)
        {
                ChunkToUse = CurrListHead->Flink;
                break;
        }
        else
        {
                //loop through all the BlocksIndex->ListHints
                //looking for a sufficiently sized chunk
                //then update the bitmap accordingly
        }

        //look at the next blocks index
        CurrBlocksIndex = CurrBlocksIndex->ExtendedLookup;
}
```

Overview

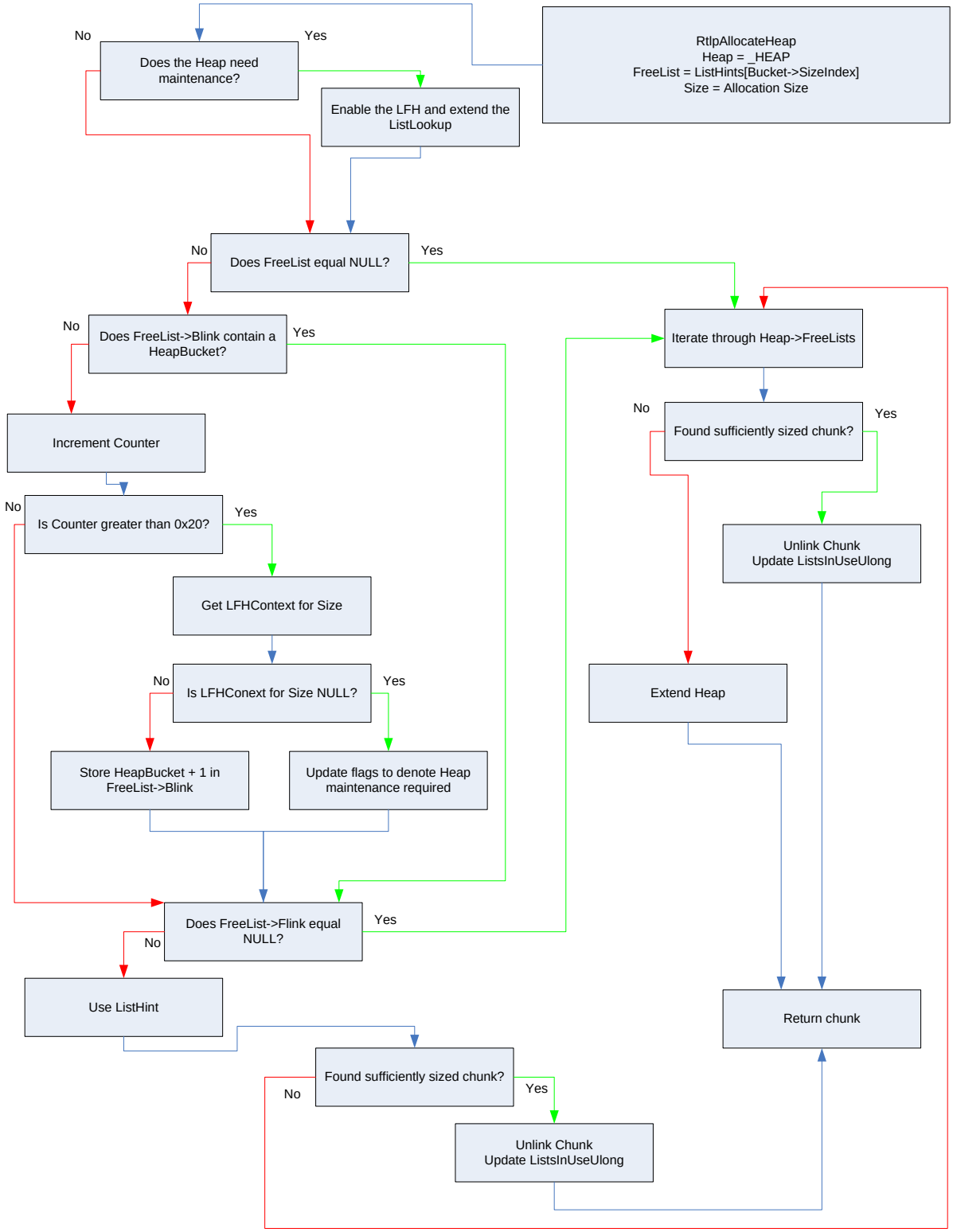

Diagram 4. Back-end allocation

Front-end Allocation

Now that we've seen how the **Low Fragmentation heap** is enabled via a heuristic in the back-end, we can go over the allocation algorithm used for the front-end manager. The **LFH** was designed with performance and reliability in mind (Marinescu 2006). These newly desired benefits come at a large cost to reverse engineers; mainly the ability to understand how the front-end allocator works. In this section I will try to illustrate how a *typical* allocation occurs when using the **Low Fragmentation heap**.

RtlpLowFragHeapAllocFromContext

As shown previously, **RtlpLowFragHeapAllocFromContext()** is only called if the **blink** for a **ListHint** has bit zero is set to 0x1. The bitwise operation determines if the **blink** contains a **HeapBucket**, indicating readiness to service the request from the **LFH**.

Allocations begin with the manager acquiring all the essential data structures for use. This includes the **_HEAP_LOCAL_DATA**, **_HEAP_LOCAL_SEGMENT_INFO** and **_HEAP_SUBSEGMENT** [The relationship between these structures can be viewed in **Diagram 1**].

The allocator will first attempt to use the **Hint SubSegment**. Pending a failure, it will then attempt to use the **ActiveSubsegment**. If the **ActiveSubsegment** fails, the allocator must setup the proper data structures for the **LFH** to continue. [To avoid redundancy, the code below only shows the pseudo-code used for the **Hint** SubSegment, but the logic can be applied to the **ActiveSubsegment** as well]

The **_INTERLOCK_SEQ** structure is queried to get the current **Depth, Offset** and **Sequence**. This information is used to get a pointer to the current free chunk as well as to calculate the **Offset** for the next available chunk. The looping logic is to guarantee that the updating of important data is done in an atomic fashion, without any changes between operations.

Listing 17. LFH SubSegment allocation

```
int LocalDataIndex = 0;

//uses the SizeIndex of the _HEAP_BUCKET to
//get the address of the LFH for this bucket
_LFH_HEAP *LFH = GetLFHFromBucket(HeapBucket);

//figure this out yourself :)
if(aHeapBucket->Affinity == 1)
{
       AllocateAndUpdateLocalDataIndex();
}

//get the LocalData and LocalSegmentInfo
//structures based on Affinity and SizeIndex
_HEAP_LOCAL_DATA *HeapLocalData =
       LFH->LocalData[LocalDataIndex];
```

```
_HEAP_LOCAL_SEGMENT_INFO *HeapLocalSegmentInfo =
       HeapLocalData->SegmentInfo[HeapBucket->SizeIndex];

//try to use the 'Hint' SubSegment first
//otherwise this would be 'ActiveSubsegment'
_HEAP_SUBSEGMENT *SubSeg = HeapLocalSegmentInfo->Hint;
_HEAP_SUBSEGMENT *SubSeg_Saved = HeapLocalSegmentInfo->Hint;

if(SubSeg)
{
       while(1)
       {
              //get the current AggregateExchange information
              _INTERLOCK_SEQ *AggrExchg = SubSeg->AggregateExchg;
              int Offset = AggrExchg->FreeEntryOffset;
              int Depth = AggrExchg->Depth;
              int Sequence = AggrExchg->Sequence

              //store the old values, to ensure atomic swapping
              _INTERLOCK_SEQ AggrExchg_Saved;
              AggrExchg_Saved.OffsetAndDepth = AggrExchg.OffsetAndDepth;
              AggrExchg_Saved.Sequence = Sequence;

              //continue only if this is a valid SubSegment
              _HEAP_USERDATA_HEADER *UserBlocks = SubSeg->UserBlocks;
              if(!Depth ||
                     !UserBlocks ||
                     SubSeg->LocalInfo != HeapLocalSegmentInfo)
              {
                     break;
              }

              //this gets the offset from the AggregateExchg
              //(block size) and creates a byte offset
              int ByteOffset =  Offset * 8;
              LFHChunk = UserBlocks + ByteOffset;

              //the next offset is store in the 1st 2-bytes
              //of the userdata (this can probably be abused ;))
              short NextOffset = UserBlocks +
                     ByteOffset +
                     sizeof(_HEAP_ENTRY));

              //store the updated offset, depth and sequence
              //new_offset = current_offset += BucketSize
              //new_depth = current_depth--
              //new_sequence = depends on current depth
              _INTERLOCK_SEQ AggrExchg_New;
              AggrExchg_New.Offset = NextOffset;
              AggrExchg_New.Depth = Depth--;
              if(AggrExchg_New.Depth == -1)
                     AggrExchg_New.Sequence = Sequence--;
              else
                     AggrExchg_New.Sequence = Sequence;

              //i.e InterlockedCompareExchange
              if(AtomicSwap(AggrExchg, AggrExchg_New))
              {
                     UpdateHeaders(LFHChunk);
                     return LFHChunk;
              }
              else
              {
```

```
                    UpdateAffinity();
                    SubSeg = SubSeg_Saved;
                }
        }
}
```

> **Note:** Although left out for formatting reasons, it should be understood that all the code in **RtlpLowFragHeapAllocFromContext()** is wrapped in a **try/catch** block. This is done so that if there is any failure in the **LFH**, **NULL** will be returned and the back-end allocator will attempt to service the memory request.

If the Hint and Active SubSegments have failed, whether from lack of initialization or being invalid, **RtlpLowFragHeapAllocFromContext()** must acquire a new **SubSegment** by allocating memory (using the back-end) and dividing a large chunk of memory to a **HeapBin**. Once this stage has been completed, the aforementioned code above will service the request via the **ActiveSubsegment**.

To service a request in the event of both SubSegments failing, a new chunk of memory must be allocated for the front-end heap. The amount of memory to be requested is not arbitrary, but based on the chunk size requested and the total amount of memory available to the heap. The following pseudo-code is something I refer to as the *Magic Formula*. It calculates how much memory to request from the back-end so that it may be split up into a **UserBlock** for a certain **HeapBucket**:

Listing 18. LFH UserBlocks allocation size algorithm

```
int TotalBlocks = HeapLocalSegmentInfo->Counters->TotalBlocks;

if(MaxRunLenReached)
       TotalBlocks = TotalBlocks / 16;

int BucketAffinity = HeapBucket->Affinity & 1;

int BucketBytesSize =  RtlpBucketBlockSizes[HeapBucket->SizeIndex];

int StartIndex = 7;

if(BucketBytesSize < 256)
       BucketAffinity--;
if ( dword_77F97594 > RtlpHeapMaxAffinity >> 1) )
       BucketAffinity++;

int BlockMultiplier = 4 - BucketAffinity;

if(TotalBlocks < (1 << BlockMultiplier))
       TotalBlocks = 1 << BlockMultiplier;

if(TotalBlocks > 1024)
       TotalBlocks = 1024;

//used to calculate cache index and size to allocate
int TotalBlockSize = TotalBlocks * (BucketBytesSize + sizeof(_HEAP_ENTRY)) +
0x18;

if(TotalBlockSize > 0x78000)
       TotalBlockSize = 0x78000;
```

```
//calculate the cahce index
//upon a cache miss, this index will determine
//the amount of memory to be allocated
if(TotalBlockSize >= 0x80)
{
	do
	{
		StartIndex++;
	}while(TotalBlockSize >> StartIndex);
}

//we will @ most, only allocate 40 pages (0x1000 bytes per page)
if((unsigned)StartIndex > 0x12)
	StartIndex = 0x12;

int UserBlockCacheIndex = StartIndex;

if(HeapBucket->Affinity & 6)
	UserBlockCacheIndex = 0x12;

//allocate space for a _HEAP_USERDATA_HEADER along with room
//for ((1 << UserBlockCacheIndex) / BucketBytesSize) heap chunks
void *pUserData =
RtlpAllocateUserBlock(LFH, UserBlockCacheIndex, BucketByteSize + 8);

_HEAP_USERDATA_HEADER *UserData = (_HEAP_USERDATA_HEADER*)pUserData;
if(!pUserData)
	return 0;
```

The *UserBlockCacheIndex* variable in the code above is used as an index into an array of cached items. If there is a cache miss, this same value is used to calculate how much memory to allocate for a **UserBlocks** chunk. This **UserBlocks** chunk will later be split up into **BucketSize** chunks. Let's see how **RtlpAllocateUserBlock**, when not using cached items, is basically a wrapper for **RtlpAllocateHeap**:

Listing 19. RtlpAllocateUserBlock without caching

```
int AllocAmount = 1 << UserBlockCacheIndex;
if(AllocAmount > 0x78000)
	AllocAmount = 0x78000;

UserBlock = RtlAllocateHeap(LFH->Heap, 0x800000, AllocAmount - 8);
if(UserBlock)
{
	LFH->CacheAllocs++;

	//Assign the _HEAP_USERDATA_HEADER->SizeIndex
	*(UserBlock+8) = UserBlockCacheIndex;

}

return UserBlock;
```

Although memory has been allocated, it is not quite ready to be used by the LFH. It must first be coupled with a **_HEAP_SUBSEGMENT**. This SubSegment is either acquired from a previously deleted one or created at an address retrieved from **_LFH_BLOCK_ZONE** list.

Listing 20. LFH Pre-SubSegment initialization setup

```
int UserDataBytesSize = 1 << UserData->AvailableBlocks;
if(UserDataBytesSize > 0x78000)
	UserDataBytesSize = 0x78000;

int UserDataAllocSize = UserDataBytesSize - 8;

//Increment SegmentCreate to denote a new SubSegment created
InterlockedExchangeAdd(&LFH->SegmentCreate, 1);

DeletedSubSegment = ExInterlockedPopEntrySList(HeapLocalData);
_HEAP_SUBSEGMENT *NewSubSegment = NULL;
if (DeletedSubSegment)
{
	// if there are any deleted subsegments, use them
	NewSubSegment = (_HEAP_SUBSEGMENT *)(DeletedSubSegment - 0x18);
}
else
{
	NewSubSegment =
		RtlpLowFragHeapAllocateFromZone(LFH, LocalDataIndex);

	//return failure use back-end
	if(!NewSubSegment)
		return 0;
}

//this function will setup the _HEAP_SUBEMENT structure
//and chunk out the data in 'UserData' to be of HeapBucket->SizeIndex chunks
RtlpSubSegmentInitialize(LFH,
	NewSubSegment,
	UserBlock,
	RtlpBucketBlockSizes[HeapBucket->SizeIndex],
	UserDataAllocSize,HeapBucket);

//each UserBlock starts with the same sig
UserBlock->Signature = 0xF0E0D0C0;
```

RtlpLowFragHeapAllocateFromZone() serves as a dual purpose function; either finding a pointer for a **_HEAP_SUBSEGMENT** or creating several **_LFH_BLOCK_ZONE** structures for future address tracking.

The function will first check to see if there are any valid **_LFH_BLOCK_ZONE** structures holding an address to be used by a **SubSegment**. If that is not the case or it has exceeded the **Limit** for which it was designed, it will then allocate 0x3F8 (1016) bytes of memory to store new **_LFH_BLOCK_ZONE** objects. The code below shows how **RtlpLowFragHeapAllocateFromZone()** typically works.

Listing 21. RtlpLowFragHeapAllocateFromZone

```
_LFH_BLOCK_ZONE *CrtZone =
LFH->LocalData[LocalDataIndex]->CrtZone;

_LFH_BLOCK_ZONE *CrtZoneFlink = NULL;

while(1)
{
	while(1)
	{
		while(1)
		{
			//Flink == NULL => create initial zones
			CrtZoneFlink = CrtZone->ListEntry->Flink;
			if(!CrtZone-Flink)
				break;

			void *FreePointer = CrtZoneFlink->FreePointer;

			//This will increment it to the next SubSegment
			void *FreePointer_New =
				FreePointer + LFH->ZoneBlockSize;

			//if we've exceeded the limit
			//create more zones
			if(FreePointer_New >= CrtZoneFlink->Limit)
				break;

			//InterlockedCompareExchange
			//loop if this fails
			if(CompareExchange(
				&CrtZoneFlink->FreePointer,
				FreePointer_New))
				return FreePointer
		}

		if (CrtZoneFlink == CrtZone->ListEntry.Flink )
			break;
	}

	//this will effectively give us 31 _LFH_BLOCK_ZONE
	//structures to use for keeping track of userdata
	void *NewLFHBlockZone =
		RtlAllocateHeap(LFH->Heap, 0x800000u, 0x3F8u);

	if(!NewLFHBlockZone)
		return 0;

	//if the CrtZone's ListEntry is empty
	if (CrtZoneFlink == CrtZone->ListEntry.Flink )
	{
		//link in the newly created structure
		//into the LFH->SubSegmentZones
		LinkInBlockZone(LFH, NewLFHBlockZone);

		//points to the end of the allocation
		NewLFHBlockZone->Limit = NewLFHBlockZone + 0x3F8;

		//sizeof(_LFH_BLOCK_ZONE) == 0x10
		char *AlignedZone =
			RoundAlign(NewLFHBlockZone + 0x10);

		NewLFHBlockZone->FreePointer = AlignedZone;
```

```
                CrtZone->ListEntry.Flink = NewLFHBlockZone;

                continue;
        }

        //if we failed, free the data
        RtlFreeHeap(LFH->Heap, 0x800000, NewLFHBlockZone);
}
```

Although the code above may seem a bit difficult to understand, it only has a few design purposes in mind. The inner most loop ensures atomic swapping of **FreePointers** to avoid race conditions when dealing with multiple threads. The outer most loop will guarantee that this function creates new **BlockZones** in the event that they are exhausted.

After an address is acquired from **RtlpLowFragHeapAllocateFromZone()**, the SubSegment can be initialized in **RtlpSubSegmentInitialize()**. As the name implies, it is responsible for initializing the **_HEAP_SUBSEGMENT**, taking a variety of arguments; such as the newly created SubSegment (NewSubSegment), the recently allocated memory (UserBlock), the amount of memory available (UserDataAllocSize), and the size of the chunks to create (HeapBucket / BucketBytesSize).

RtlpSubSegmentInitialize() will first get the **LocalSegmentInfo** and **LocalData** structures based of the **HeapBucket** size. After ensuring that a certain **Affinity** state is guaranteed, it will then proceed to calculate exactly how many chunks are going to be available for this **UserBlock**. Once the number of chunks to create is determined, it will iterate through the large chunk of memory, writing a header for each chunk. Finally the **_INTERLOCK_SEQ** has its initial values set to have the **Depth** as the **NumberOfChunks** and the **FreeEntryOffset** equal to **0x2**.

Listing 22. RtlpSubSegmentInitialize

```
void *UserBlockData = UserBlock + sizeof(_HEAP_USERDATA_HEADER);

_HEAP_LOCAL_SEGMENT_INFO *LocalSegmentInfo =
LFH->LocalData[NewSubSegment->AffinityIndex]->SegmentInfo[HeapBucket-
>SizeIndex];

_HEAP_LOCAL_DATA *LocalData =
LFH->LocalData[NewSubSegment->AffinityIndex]->Segmentinfo[HeapBucket-
>SizeIndex]->LocalData;

if (!((HeapBucket->Affinity >> 1) & 3))
{
        int TotalBucketByteSize = BucketByteSize + sizeof(_HEAP_ENTRY);
        int BucketBlockSize = TotalBucketByteSize / 8;

        //sizeof(_HEAP_USERDATA_HEADER) == 0x10
        int NumberOfChunks = (UserDataAllocSize - 0x10) /
                TotalBucketByteSize;

        //skip past the header, so we can start chunking
        void *pUserData = UserBlock + sizeof(_HEAP_USERDATA_HEADER);


        //assign the SubSegment
        UserBlock->SubSegment = NewSubSegment;
```

```
	//sizeof(_HEAP_USERDATA_HEADER) == 0x10 (2 blocks)
	int SegmentOffset = 2;

	_INTERLOCK_SEQ AggrExchg_New;
	AggrExchg_New.FreeEntryOffset = 2;

	if(NumberOfChunks)
	{
		int NumberOfChunksItor = NumberOfChunks;
		do
		{
			SegmentOffset += BucketBlockSize;
			pUserData = UserBlockData;
			UserBlockData += BucketByteSize;

			//next FreeEntryOffset
			*(WORD*)(pUserData + 8) = SegmentOffset;

			//Set _HEAP_ENTRY.LFHFlags
			*(BYTE*)(pUserData + 6) = 0x0;

			//Set _HEAP_ENTRY.UnusedBytes
			*(BYTE*)(pUserData + 7) = 0x80;

			EncodeDWORD(LFH, pUserData)
		}
		while(NumberOfChunksItor--);
	}

	//-1 indicates last chunk
	//in the UserBlock (_HEAP_USERDATA_HEADER)
	*(WORD*)(pUserData + 8) = -1;

	//Sets all the values for this subsegment
	InitSubSegment(NewSubSegment);

	//updates the bucket counter to reflect
	//NumberOfChunk as total blocks and
	//sets the SubSegmentCount
	UpdateBucketCounters(LocalSegmentInfo);

	//will be atomically assigned to the
	//NewSubSegment's _INTERLOCK_SEQ
	AggrExchg_New.Depth = NumberOfChunks;
	AggrExchg_New.Sequence = AggrExchg_Saved.Sequence + 1;

	//InterlockedCompareExchange64
	AtomicSwap(&NewSubSegment->AggregateExchg, AggrExchg_New);
}
```

Finally, after the UserBlocks have been allocated, assigned to a SubSegment and the SubSegment has been initialized, the **LFH** can set the **ActiveSubsegment** to the one that was just initialized. It will operate on some locking mechanisms, eventually atomically assigning the **ActiveSubsegment**. Lastly execution will resume at the point shown in **Listing 17**.

Listing 23. ActiveSubsegment assignment

```
//now used for LFH allocation for a specific bucket size
AtomicSwap(&HeapLocalSegmentInfo->ActiveSegment, NewSubSegment);
```

Overview

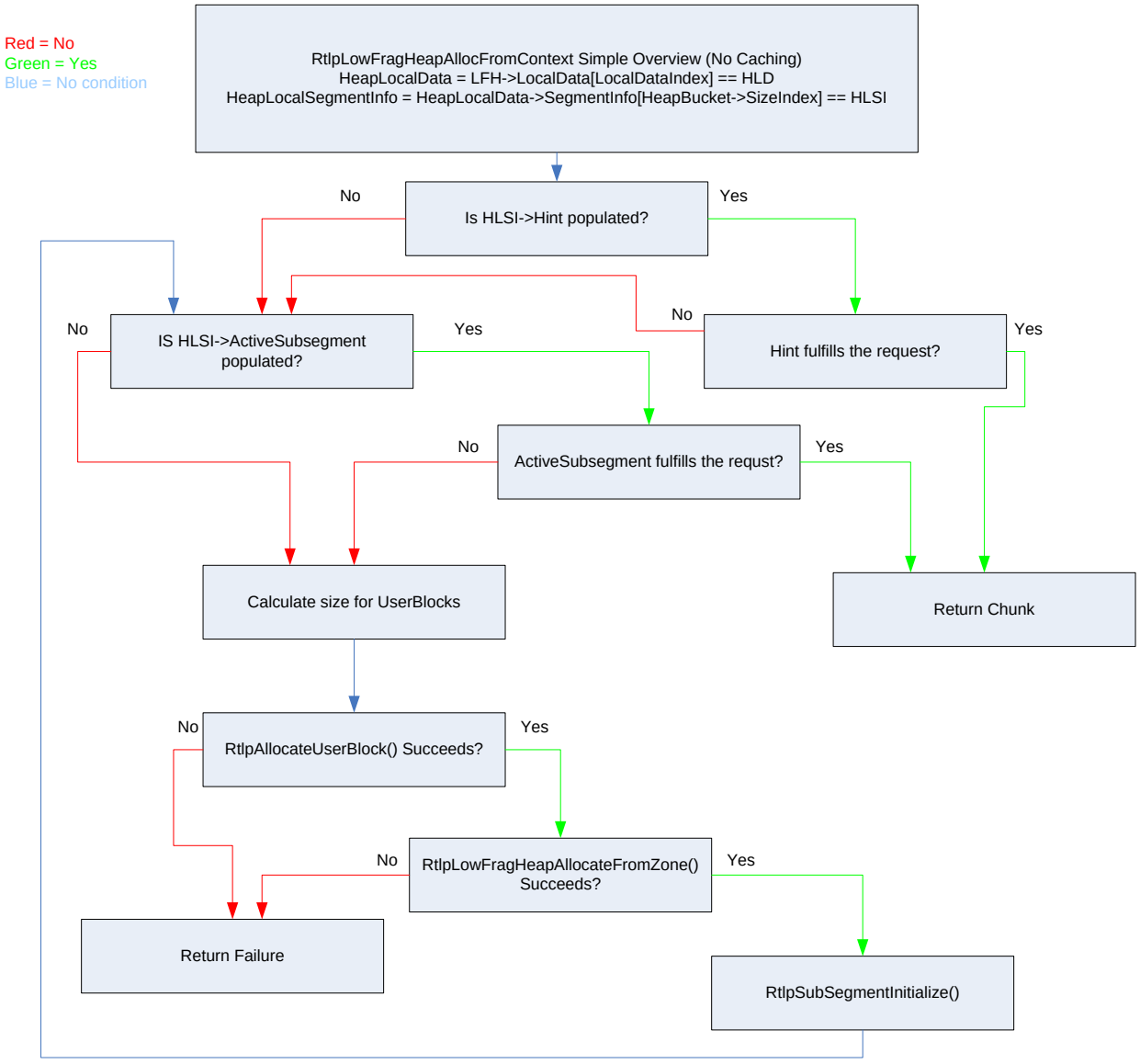

Diagram 5. Front-end allocation

Example

The best way to fully understand the allocation process is with an example. Let's assume that the **LFH** has been enabled and we're on our first allocation request that will be fulfilled by the front-end allocator. A request has been received for 0x28 (40) bytes, which after adding space for a header will be 0x30 (48) bytes (or 0x6 blocks). It will also be assumed that we're using the **ActiveSubsegment** from **SegmentInfo[0x6]** located in the **_HEAP_LOCAL_DATA** structure.

Note: LFH->LocalData[0]->SegmentInfo[0x6]->ActiveSubsegment->UserBlocks

Based on the *Magic Formula* shown above, we can deduce that there are going to be **0x2A** chunks for **0x30** bytes (i.e. the **Depth**). The initial offset will be set to **0x2** because the **_HEAP_USERDATA_HEADER** is 0x10 bytes wide.

Each chunk in the **UserBlock** will contain an 8-byte header, for which the first 4-bytes are encoded, and then n-bytes of user writable data to be returned to the calling process. The first **2-bytes** of the user-writable data hold the **next offset** to be assigned to the **_INTERLOCK_SEQ**.

Each offset is calculated from the beginning of the **UserBlock** chunk and referenced in terms of **blocks**. The byte offset of the next available chunk would be **UserBlocks + FreeEntryOffset * 0x8**.

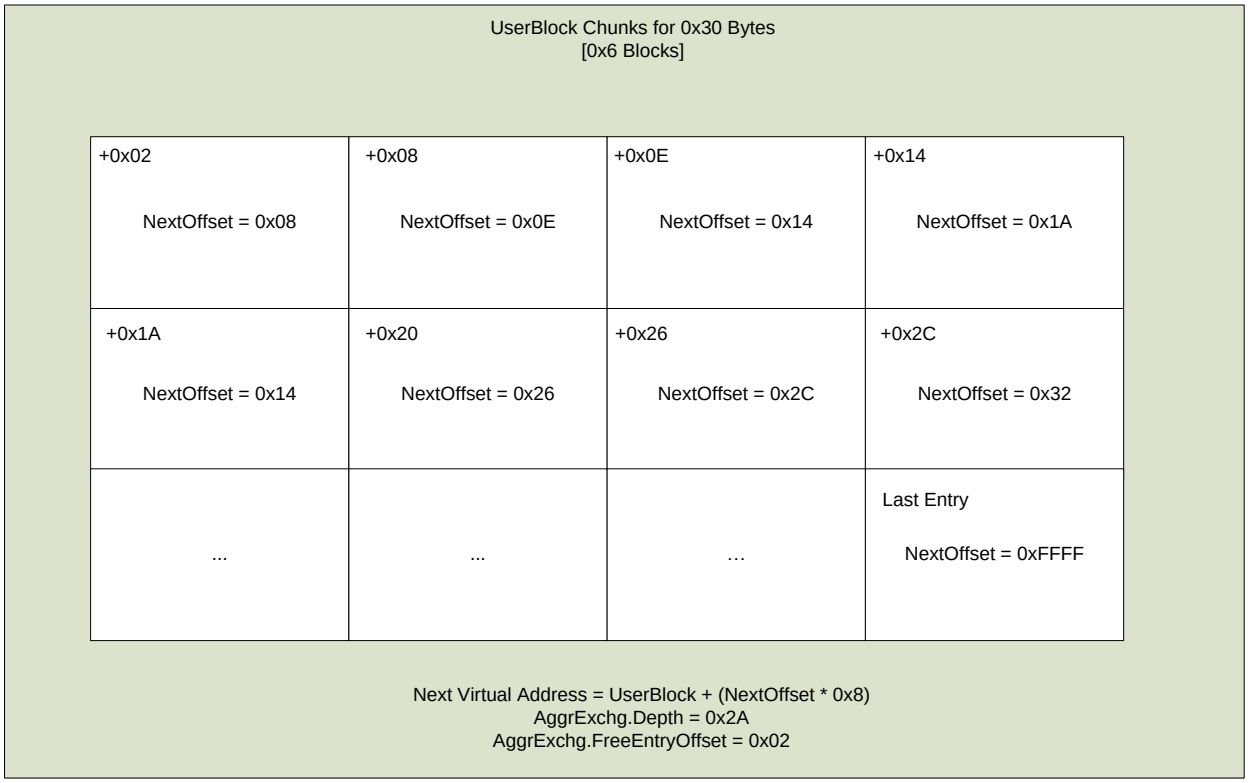

Diagram 6. Full UserBlock for Bucket 0x6

After an initial allocation is made, the Depth and offset are updated to reflect the next available chunk in the UserBlock. While the memory is not actually moved, just indexed differently, the follow diagram will show the available memory after **one** allocation. Notice how the **Offset** holds the value previously stored in a chunk (i.e. NextOffset) and the **Depth** is decremented by 0x1; indicating we have used one block and have **0x29** remaining.

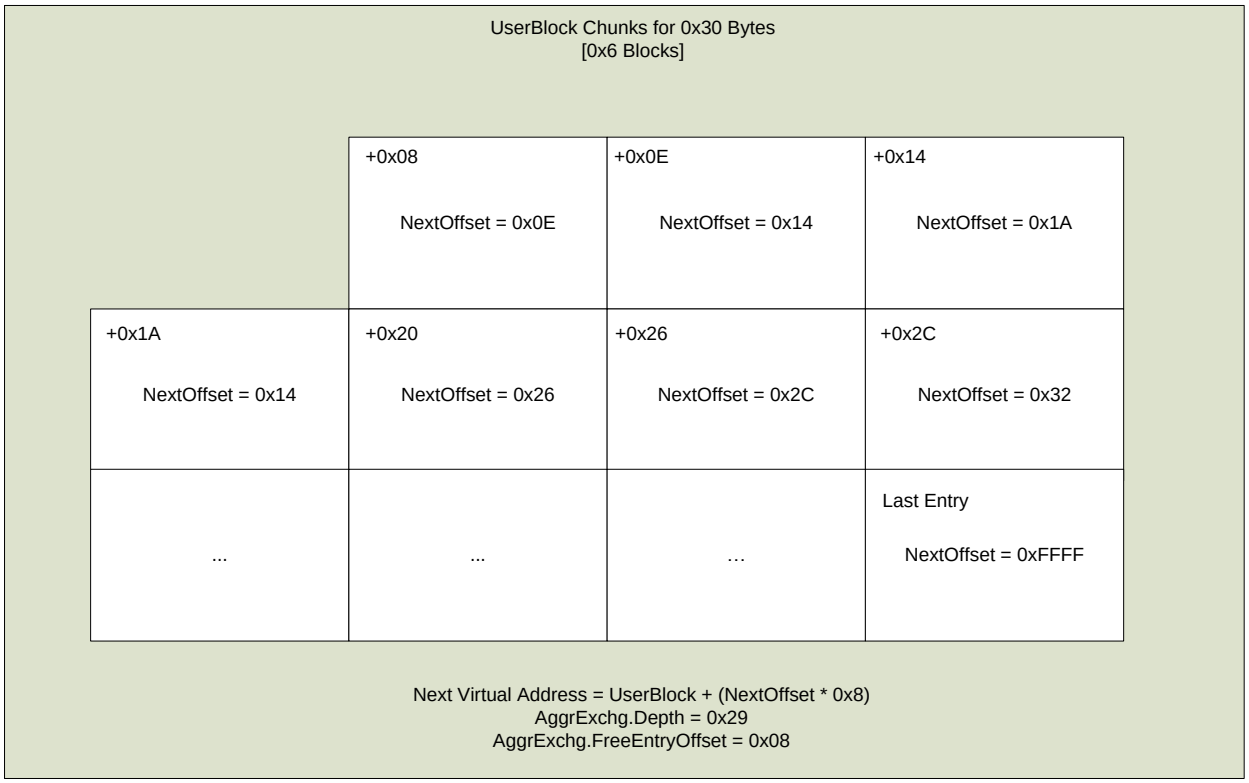

Diagram 7. UserBlock after the 1st allocation for 0x30 bytes

After the **second** allocation, offset **UserBlock+0xE** will be next free chunk. After that, **Userblock+ 0x14** will be the next free chunk, and so on. It keeps incrementing the **Offset** and decrementing the **Depth** until the Depth equals 0; indicating the need to allocate more memory for yet another **UserBlock**. The following diagram is the 2nd consecutive allocation for 0x30 bytes. We will see how these blocks are released in the *Freeing* section.

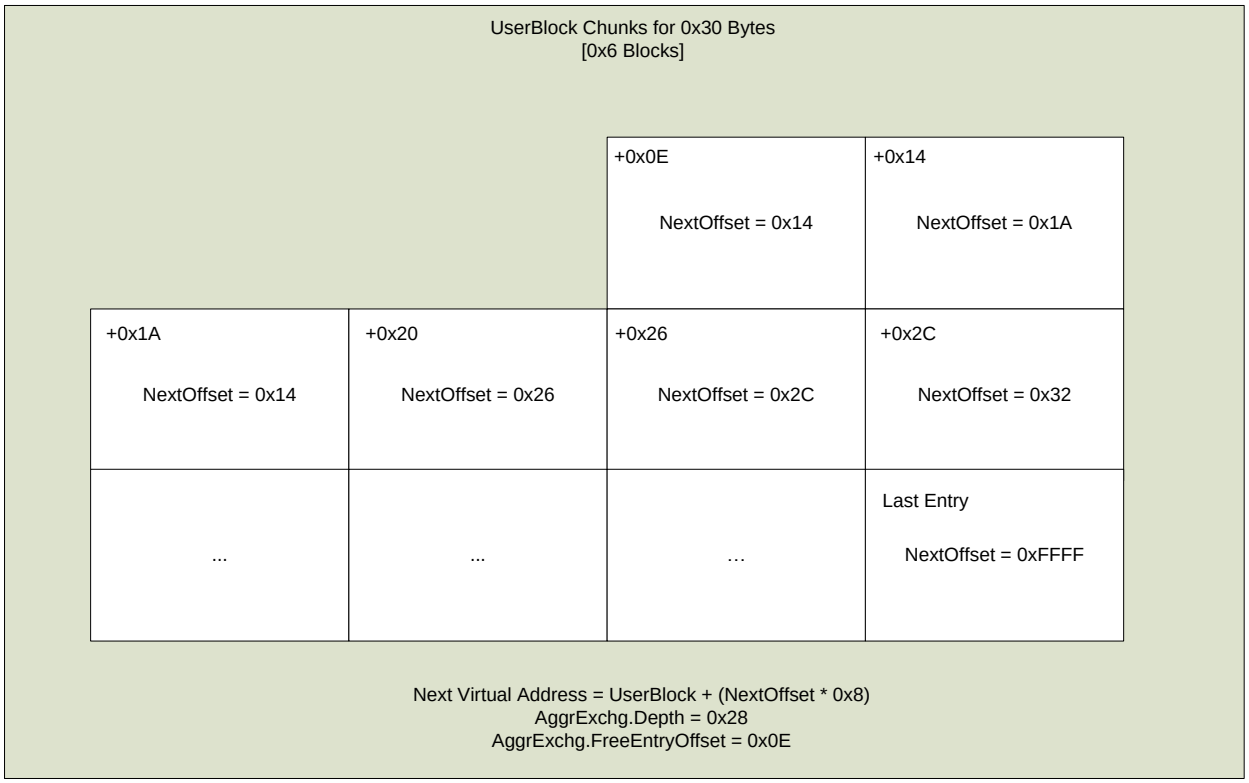

Diagram 8. UserBlock after the 2nd consecutive allocation for 0x30 bytes

Freeing

Now that a base understanding of how memory is acquired in Windows 7, I can now discuss how it is released. A chunk in use will eventually need to be freed by the application and given back to the heap manager. This process starts at **RtlFreeHeap()**, which takes the **heap**, **flags**, and the **chunk** to be freed as arguments. This function's first order of business is to figure out if this chunk is actually free-able and then inspect the **chunk's** header to determine which piece of the heap manager should be responsible for releasing it.

Listing 24. RtlFreeHeap

```
ChunkHeader = NULL;

//it will not operate on NULL
if(ChunkToFree == NULL)
	return;

//ensure the chunk is 8-byte aligned
if(!(ChunkToFree & 7))
{
	//subtract the sizeof(_HEAP_ENTRY)
	ChunkHeader = ChunkToFree - 0x8;

	//use the index to find the size
	if(ChunkHeader->UnusedBytes == 0x5)
		ChunkHeader -=
			0x8 * (BYTE)ChunkToFreeHeader->SegmentOffset;
}
else
{
	RtlpLogHeapFailure();
	return;
}

//position 0x7 in the header denotes
//whether the chunk was allocated via
//the front-end or the back-end (non-encoded ;) )
if(ChunkHeader->UnusedBytes & 0x80)
	RtlpLowFragHeapFree(Heap, ChunkToFree);
else
	RtlpFreeHeap(Heap, Flags | 2, ChunkHeader, ChunkToFree);

return;
```

Back-end Freeing

The back-end manager is responsible for handling all memory that isn't handled by the front-end heap; either due to the size or the lack of a **Low Fragmentation heap**. While all allocations over 0xFE00 blocks are handled by **VirtualAlloc()/VirtualFree()**, all memory management that is greater than 0x800 blocks is handled by the back-end; along with any memory that cannot be serviced by the front-end.

RtlpFreeHeap

RtlpFreeHeap() takes a **_HEAP**, **Flags**, **ChunkHeader**, and the **ChunkToFree** as arguments. It will first attempt to decode the chunk header if it is encoded, then proceed to find an appropriate **ListHint** within the BlocksIndex. If it cannot find a sufficient **index** it will use BlocksIndex->ArraySize-1 as the **ListHint**.

Listing 25. RtlpFreeHeap BlocksIndex search

```
if(Heap->EncodeFlagMask)
	DecodeAndValidateChecksum(ChunkHeader);

int ChunkSize = ChunkHeader->Size;
_HEAP_LIST_LOOKUP *BlocksIndex = Heap->BlocksIndex;

while(1)
{
	//if the chunk will fit in this BlocksIndex, break out
	if(ChunkSize < BlocksIndex->ArraySize)
		break;

	//if the chunk is too big for this blocksindex and there is NOT
	//and extended lookup, then free onto FreeList[BlocksIndex->ArraySize-1]
	if(!BlocksIndex->ExtendedLookup)
	{
		ChunkSize = BlocksIndex->ArraySize - 1;
		break;
	}

	//next item in the linked list
	BlocksIndex = BlocksIndex->ExtendedLookup;
}
```

Note: This process of finding a ListHint Index and returning will from here on out be referred to as *BlocksIndexSearch()*. It will take a **_HEAP_LIST_LOOKUP** and a **ChunkSize** as an input. It will walk the linked list updating the BlocksIndex argument until it has found a candidate, lastly returning a **FreeListIndex**.

Now that a **_HEAP_LIST_LOOKUP** has been found, the function can attempt to use a specific **ListHint**. The ListHint can be a specific value, such as ListHints[0x6], or, if the chunk to free is larger than what the **BlocksIndex** manages, it could reside in ListHints[BlocksIndex->ArraySize-BaseIndex-1]. (This can be thought of as a *FreeList[0] type* list.)

Listing 26. RtlpFreeHeap ListHint retrieval

```
//attempt to locate a freelist
_LIST_ENTRY *ListHint = NULL;

//if the chunk can be managed by specific BlocksIndex
if(ChunkSize < (BlocksIndex->ArraySize - 1) ||
       BlocksIndex->ExtendedLookup != 0x0 &&
       ChunkSize == (BlocksIndex->ArraySize - 1))
{
       //get the offset into the ListHints
       int BaseIndex = BlocksIndex->BaseIndex;
       int FreeListIndex =
              RelativeSize(BlocksIndex, ChunkSize - BaseIndex);

       //acquire a freelist
       ListHint = BlocksIndex->ListHints[FreeListIndex];
}
```

If a **ListHint** has been located and the **blink** does not contain a **HeapBucket**, then the back-end manager will update the value used in the **LFH** heuristic. Since a chunk is being put back on the heap, it will subtract 0x2 from the counter. This means that to actually enable the **LFH** for a given **Bucket**, you must see at least 0x11 consecutive allocations.

For example, if 0x10 requests are received for **Bucket[0x6]**, then 0x2 of those chunks are released back to the heap, followed by 0x2 allocations for the same size, the **LFH** will **NOT** be enabled for **Bucket[0x6]**. The threshold must be met before the activation heuristic will perform heap maintenance.

Listing 27. RtlpFreeHeap LFH counter decrement

```
if(ListHint != NULL)
{
       int FreeListBlink = ListHint->Blink;

       //if the blink is not populated with a HeapBucket
       //decrement the counter, as we just freed a chunk
       if( !(BYTE)FreeListBlink & 1)
       {
              if(FreeListBlink >= 2)
                     ListHint->Blink = FreeListBlink - 2;
       }
}
```

After updating the counter for **LFH** activation, **RtlpFreeHeap()** will attempt to coalesce the chunk if the heap permits doing so. Chunk coalescing is an important process in which the heap looks at the **two chunks** adjacent to the piece of memory being freed. This is done to avoid having many small free chunks residing next to each other (The Low Fragmentation heap directly addresses this problem). Although **RtlpCoalesceFreeBlocks()** is always called, chunk coalescing will only occur if an adjacent chunk is marked as **FREE**.

Once coalescing of adjacent blocks is completed a check is made to ensure that the new total block size isn't above the **Heap->DeCommitThreshold** along with ensuring that it is not required to be handled by virtual memory. Lastly, this piece of the algorithm will mark the chunk as **FREE** and having **zero** unused bytes.

Listing 28. RtlpFreeHeap Chunk coalescing and header reassignment

```
//unless the heap says otherwise, coalesce the adjacent free blocks
int ChunkSize = ChunkHeader->Size;
if( !(Heap->Flags & 0x80) )
{
       //combine the adjacent blocks
       ChunkHeader =
              RtlpCoalesceFreeBlocks(Heap, ChunkHeader, &ChunkSize, 0x0);
}

//reassign the ChunkSize if neccessary
ChunkSize = ChunkHeader->Size;

//if the coalesced chunk is bigger than the
//decommit threshold for this heap, decommit the memory
if(ChunkHeader->Size > DeCommitThreshold ||
   ChunkHeader->Size + TotalFreeBlocks > DeCommitThreshold)
{
       RtlpDeCommitFreeBlock(Heap, ChunkHeader, ChunkSize, 0x0);
       return;
}

if(ChunkSize > 0xFE00)
{
       RtlpInsertFreeBlock(Heap, ChunkHeader, ChunkSize);
       UpdateTagEntry(ChunkHeader);
       return;
}

//mark the chunk as FREE
ChunkToFreeHeader->Flags = 0x0;
ChunkToFreeHeader->UnusedBytes = 0x0;
```

A free chunk must be placed in a certain position on the FreeLists or, at the very minimum, on the *FreeList[0]* type structure held **ListHints[ArraySize-BaseIndex-1]**. The first step in this process is walking the **_HEAP_LIST_LOOKUP** looking for an insertion point. Then it will walk the **ListHead**, which if you remember, is the same pointer as **Heap->FreeLists**. This pointer starts at the smallest free chunk and links upwards towards the largest.

A loop is setup to iterate through all the _HEAP_LIST_LOOKUP structures available to this heap. The algorithm will then grab the **ListHead** and do a few initial verifications; the first being a check to see if the list is empty. If it is, the loop is terminated and execution continues. The second check is to ensure that the **chunk** to be freed will actually fit in the list. It does this by ensuring that the last item in the list (**ListHead->Blink**) has a size that is greater than the chunk to be freed.

Finally it will check the first item on the ListHead, determining if it can insert before it. If it cannot, the **FreeLists** will be walked to determine where the newly freed chunk can be linked in; starting at the **FreeListIndex** position. [Please see *Diagram 3* for more information on the FreeLists]. You can now see why these items are referred to as **ListHints** because they aren't actually dedicated lists terminated by a link to the sentinel node; instead they are pointers to a location within the overall **FreeLists**.

Listing 29. RtlpFreeHeap Insertion point search

```
//FreeList will determine where, if anywhere there is space
BlocksIndex = Heap->BlocksIndex;
_LIST_ENTRY *InsertList = Heap->FreeLists;

//attempt to find where to insert this item
//on the ListHead list for a particular BlocksIndex
if(BlocksIndex)
{
	//find a BlocksIndex for storage
	int FreeListIndex = BlocksIndexSearch(BlocksIndex, ChunkSize)

	while(BlocksIndex != NULL)
	{
		_HEAP_ENTRY *ListHead = BlocksIndex->ListHead;

		//if the ListHead is empty
		//our insert list will be the sentinel node
		if(ListHead == ListHead->Blink)
		{
			InsertList = ListHead;
			break;
		}

		//if the chunk is larger than the largest
		//entry, we'll insert it after
		if(ChunkSize > ListHead->Blink.Size)
		{
			InsertList = ListHead;
			break;
		}

		//pick the insertion point behind the 1st
		//chunk larger than the ChunkToFree
		_LIST_ENTRY *NextChunk = ListHead->Flink;
		if(NextChunk.Size > ChunkSize)
		{
			InsertList = NextChunk;
			break;
		}

		NextChunk = BlocksIndex->ListHints[FreeListIndex];
		while(NextChunk != ListHead)
		{
			//there is actually some decoding done here
			if(NextChunk.Size > ChunkSize)
			{
				InsertList = NextChunk;
				break;
			}

			NextChunk = NextChunk->Flink;
		}

		//if we've found an insertion
		//spot terminate the loop
		if(InsertList != Heap->FreeLists)
			break;

		BlocksIndex = BlocksIndex->ExtendedLookup;
	}
}
```

Note: For brevity's sake, the decoding of the chunk headers to acquire the size is left out. Please don't mistake this un-encoded header dereferencing.

With an insertion point pinpointed, **RtlpFreeHeap()** will ensure that the chunk is linked into the proper location. Once the final location for chunk insertion is located, it is **safely** linked into the FreeList. To my knowledge this functionality is new and directly addresses Brett Moore's Insertion Attack (Moore 2005). Finally the chunk is placed on the appropriate **FreeList** and the **ListsInUseUlong** is updated accordingly.

Listing 30. Safe link-in

```
while(InsertList != Heap->FreeLists)
{
	if(InsertList.Size > ChunkSize)
		break;

	InsertList = InsertList->Flink;
}

//R.I.P FreeList Insertion Attack
if(InsertList->Blink->Flink == InsertList)
{
	ChunkToFree->Flink = InsertList;
	ChunkToFree->Blink = InsertList->Blink;
	InsertList->Blink->Flink = ChunkToFree;
	InsertList->Blink = ChunkToFree
}
else
{
	RtlpLogHeapFailure();
}

BlocksIndex = Heap->BlocksIndex;
if(BlocksIndex)
{
	FreeListIndex = BlocksIndexSearch(BlocksIndex, ChunkSize);

	int RelSize = ChunkSize - BlocksIndex->BaseIndex;
	FreeListIndex =     RelativeSize(BlocksIndex, RelSize);

	_LIST_ENTRY *FreeListToUse = BlocksIndex->ListHints[FreeListIndex];

	if(ChunkSize >= FreeListToUse.Size)
	{
		BlocksIndex->ListHints[FreeListIndex] = ChunkToFree;
	}

	//bitwise OR instead of the XP XOR (R.I.P Bitmap flipping (hi nico))
	if(!FreeListToUse)
	{
		int UlongIndex = Chunkize - BlocksIndex->BaseIndex >> 5;
		int Shifter = ChunkSize - BlocksIndex->BaseIndex & 1F;
		BlocksIndex->ListsInUseUlong[UlongIndex] |= 1 << Shifter;
	}
	EncodeHeader(ChunkHeader);
}
```

Note: Notice that the **ListsInUseUlong** is using a bitwise OR, instead of the previously used XOR. This ensures that a populated list is always marked as populated and an empty list can only be marked as populated.

Hint: What would happen if **RtlpLogHeapFailure()** doesn't terminate execution? (Flink/Blink never updated…)

Overview

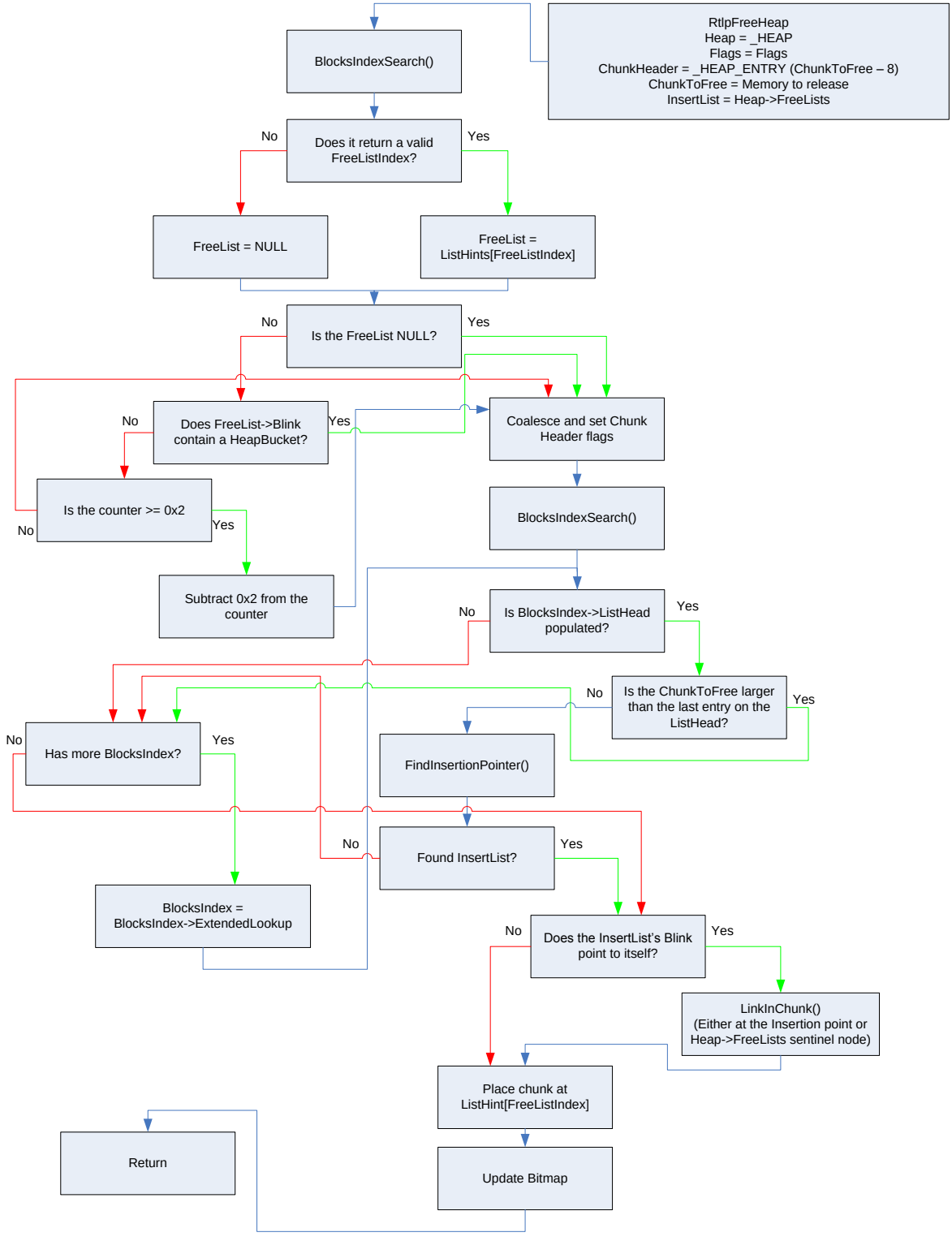

Diagram 9. RtlpFreeHeap overview

Front-end Freeing

Front-end allocation is handled by the **Low-Fragmentation heap**. While it is not used first, it is the only heap manager used once a certain heuristic is triggered. The **LFH** was designed to prevent memory fragmentation and support frequent use of specific sizes, differing greatly from the *old* front-end manager, the Lookaside List, which kept track of chunks below 1024 bytes in a linked list structure. While the **BlocksIndex** structure can track chunks that are greater than **16k** in size, the **LFH** will only be used for chunks below **16k**.

RtlpLowFragHeapFree

RtlpLowFragHeapFree() takes two arguments, a _HEAP structure and a pointer to the chunk to free. The function first checks to see if certain flags are set in the *ChunkToFree* header. If they are equal to 0x5, then an adjustment is made to change the location of the header. It then finds the associated **SubSegment**, which in turn will let it access all the members needed for memory tracking. It will also reset some values in the header to reflect a recently freed chunk.

Listing 31. RtlpLowFragHeapFree Subsegment acquisition

```
//hi ben hawkes :)
_HEAP_ENTRY *ChunkHeader = ChunkToFree - sizeof(_HEAP_ENTRY);
if(ChunkHeader->UnusedBytes == 0x5)
       ChunkHeader -= 8 * (BYTE)ChunkHeader->SegmentOffset;

_HEAP_ENTRY *ChunkHeader_Saved = ChunkHeader;

//gets the subsegment based off
//the LFHKey, Heap and ChunkHeader
_HEAP_SUBSEGMENT SubSegment = GetSubSegment(Heap, ChunkToFree);
_HEAP_USERDATA_HEADER *UserBlocks = SubSegment->UserBlocks;

//Set flags to 0x80 for LFH_FREE (offset 0x7)
ChunkHeader->UnusedBytes = 0x80;

//Set SegmentOffset or LFHFlags (offset 0x6)
ChunkHeader->SegmentOffset = 0x0;
```

> **Note:** Ben Hawkes explained how overwriting the chunk header can be used to alter the ChunkHeader pointer, leading to freeing of semi-controlled memory. You will have a maximum of *ChunkHeader – (0x8 * 0xFF)* memory address space to choose from.

A proper **chunk header** has been located, so now the function will need to calculate a new offset. The offset will be used to write into the UserBlock that is associated with the **SubSegment** (calculated above). Some initial checks are made to ensure that the SubSegment hasn't over stepped its bounds before actually freeing the chunk. If these conditions aren't made, values are set to indicate that the SubSegment needs some maintenance performed [excluded for simplicity].

Next an attempt will be made to get an **_INTERLOCK_SEQ** from the SubSegment, acquiring the current **Offset, Depth**, and **Sequence**. The **next offset** is collected from the chunk that is forwardly adjacent to the chunk being freed. As we saw in

RtlpLowFragHeapAllocFromContext(), this is stored in the first 2-bytes of the chunks data. The **Depth** is incremented by one because a chunk has just been placed back into the available bin.

An attempt to atomically swap the new and old values is made, breaking the loop on success and continuing the loop on failure. This is typical of most of the actions in the **LFH**, which are designed to work in highly multi-threaded environments.

Listing 32. RtlpLowFragHeapFree OffsetAndDepth / Sequence update

```
while(1)
{
	_INTERLOCK_SEQ AggrExchg;

	AggrExchg.OffsetAndDepth = SubSegment->AggregateExchg.OffsetAndDepth;

	AggrExchg.Sequence = SubSegment->AggregateExchg.Sequence;

	int Sequence_New = AggrExchg.Sequence + 1;
	if(AggrExchg.OffsetAndDepth >= -1)
		Sequence_New--;

	int Depth = SubSegment->AggrExchg.Depth;

	_HEAP_LOCAL_SEGMENT_INFO *LocalSegmentInfo = SubSegment->LocalInfo;
	unsigned int Sequence = SubSegment->LocalInfo->LocalData->Sequence;
	unsigned int LastOpSequence = LocalSegmentInfo->LastOpSequence;

	//setup the new INTERLOCK_SEQ
	_INTERLOCK_SEQ AggrExchg_New;
	AggrExchg_New.Sequence = Sequence_New;

	if(Depth != SubSegment->BlockCount ||
		LocalSegmentInfo->Counters.SubSegmentCounts == 1)
	{
		if(Sequence >= LastOpSequence &&
			(Sequence - LastOpSequence) < 0x20)
		{
			//set the FreeEntry Offset of ChunkToFree
			*(WORD)(ChunkHeader + 8) = AggrExchg.FreeEntryOffset;

			//Subtract the size of the block being freed
			//from the current offset; which will give
			//you the next free chunk
			int NewOffset = AggrExchg.FreeEntryOffset -
				(ChunkHeader - UserBlocks) / 8;
			AggrExchg_New.FreeEntryOffset = NewOffset;

			//increase depth because we're freeing
			AggrExchg_New.Depth = Depth + 1;

			//set the Hint in the subsegment
			Sequence = 1;
			SubSegment->LocalInfo->Hint = SubSegment;
		}
		else
		{
			Sequence = 3;
			AggrExchg_New.Depth = -1; //last entry
			AggrExchg_New.FreeEntryOffset = 0; //no offset
		}
	}
```

```
        else
        {
                Sequence = 3;
                AggrExchg_New.Depth = -1; //last entry
                AggrExchg_New.FreeEntryOffset = 0; //no offset
        }

        //_InterlockedCompareExchange64
        if(AtomicSwap(&SubSegment->AggregateExchg, AggrExchg_New, AggrExchg))
                break;

        //if something has changed since swapping, try again
        ChunkHeader = ChunkHeader_Saved;
}
```

Note: You can see this is where **SubSegment->Hint** is assigned for future use in the allocator.

Finally a check is made to see if the **Sequence** variable has been set to 0x3. If it has, this means that the **SubSegment** needs some operations performed and that the **UserBlocks** chunk can be freed (via the back-end manager). If this is not the case, 0x1 is returned to the calling function.

Listing 33. RtlpLowFragHeapFree Epilog

```
//if there are cached items handle them
UpdateCache(SubSegment);

//if we've freed every item in the list
//update the subsegment and free the UserBlock
if(Sequence == 3)
{
        PerformSubSegmentMaintenance(SubSegment);
        RtlpFreeUserBlock(LFH, SubSegment->UserBlocks);
}

return 1;
```

Note: PerformSubSegmentMaintenance() is not a real function, only a pseudonym for a series of complex instructions that will prepare the SubSegment for release or future use.

Overview

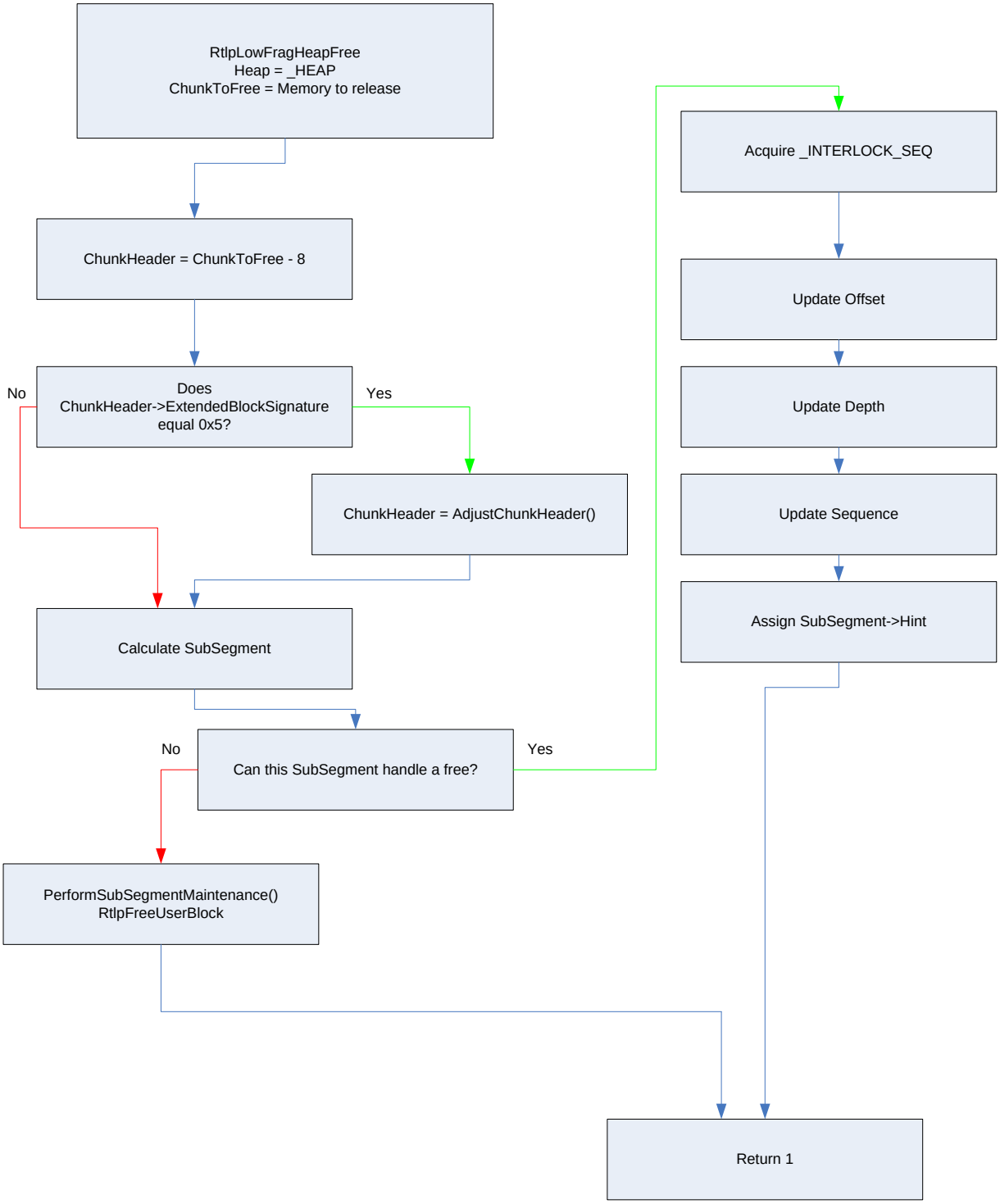

Diagram 10. RtlpLowFragHeapFree overview

Example

Continuing with the example in the *Allocation* section of the paper (above), we will make a third consecutive allocation for 0x30 bytes. This means that there are **0x27** chunks left (0x30 a piece) and the current offset to the next chunk is **0x14** from the **UserBlock**; which looks like this:

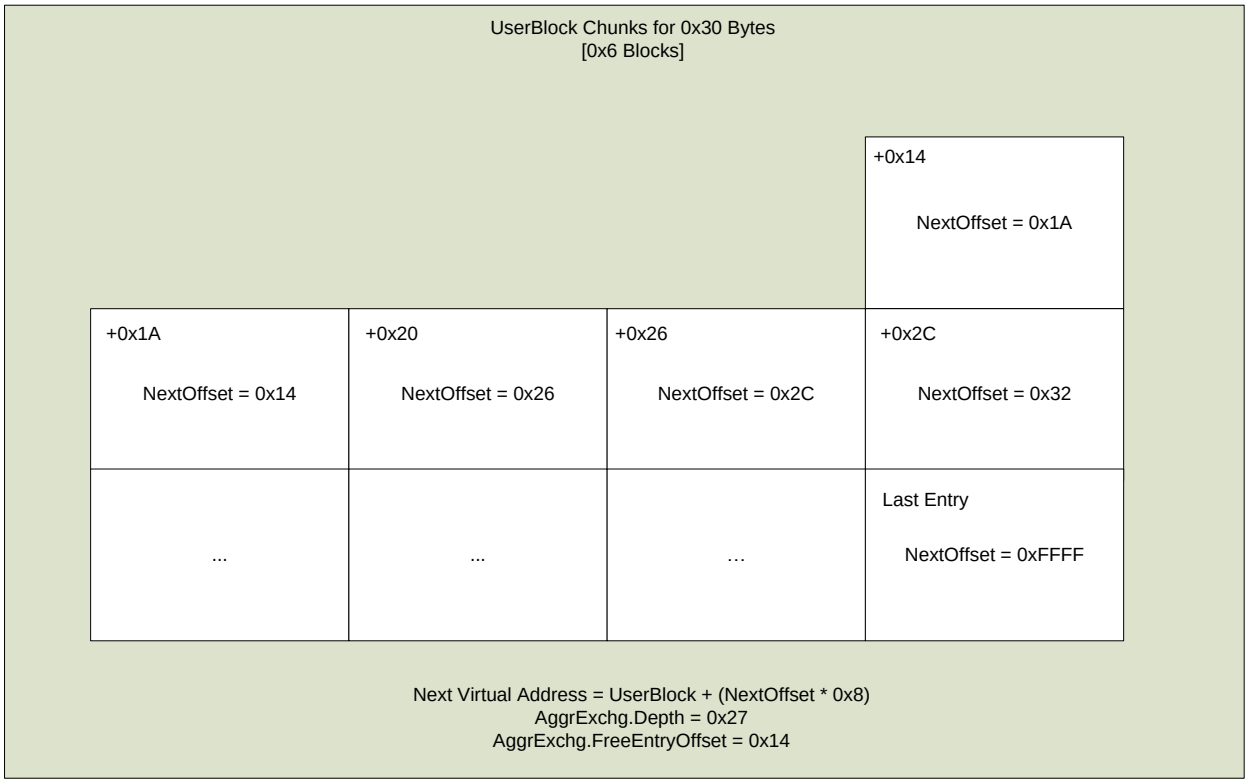

Diagram 11. UserBlock after the 3rd consecutive allocation for 0x30 bytes

So what exactly happens when memory that was allocated from the LFH is freed? As previously stated, the memory is not actually *moved* anywhere, instead an **Offset** is updated which is used as an index to the next *free* location within the UserBlocks.

Suppose that the **first** chunk that was allocated from the UserBlocks is freed. It will need to update the Offset to the location of the first chunk and increment the depth by **0x1**. The new offset is calculated by taking the address of the chunk header, subtracting the address of the UserBlock and dividing the result by 0x8. That is, the new Offset is derived from its relative position from the UserBlock (in **blocks**). The following diagram is the state of the UserBlock after the initial chunk of memory has been freed.

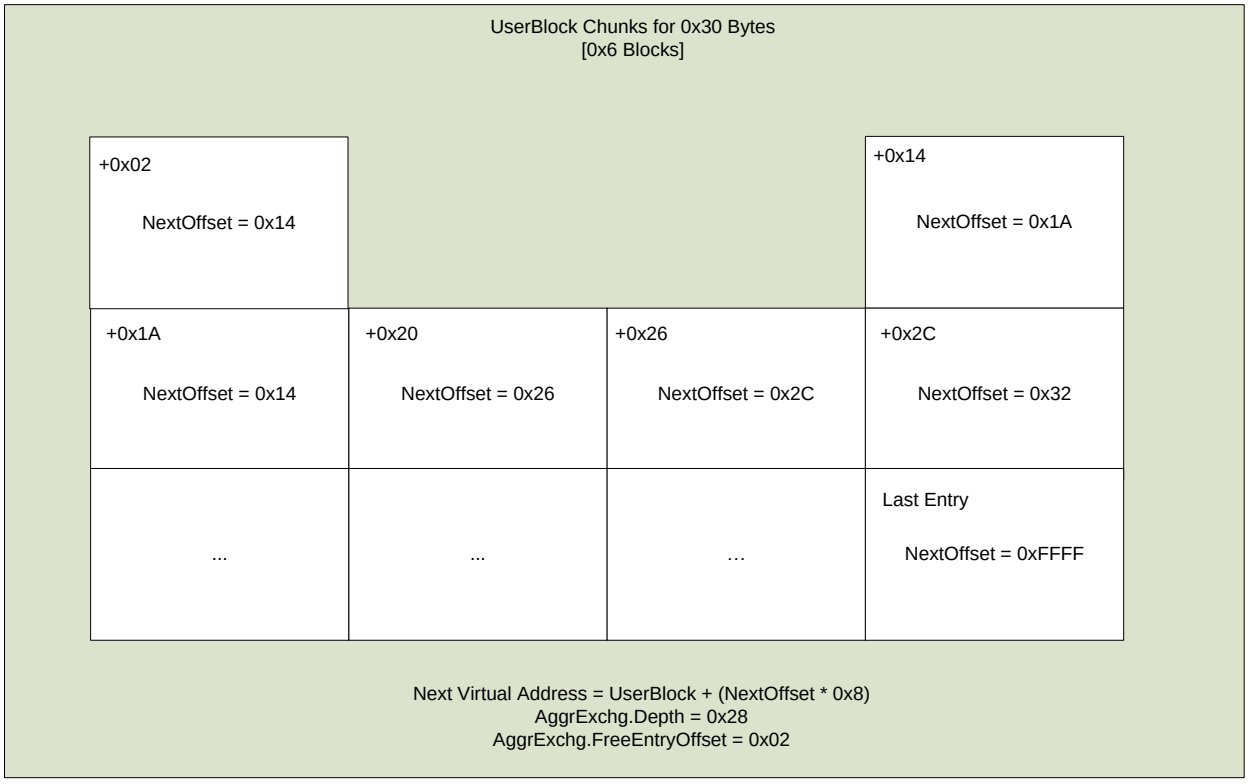

Diagram 12. Freeing of the 1st chunk allocated for 0x30 bytes

Now imagine that the second chunk that was allocated is freed (before any other allocations or frees occur). The new offset would be **0x8** because the second chunk was freed **after** the first. Even though there is a free chunk at Offset **0x2**, the next chunk to be used for an allocation resides at Offset **0x8**. This can be thought of as a linked list that updates a piece of a pointer instead of the entire address.

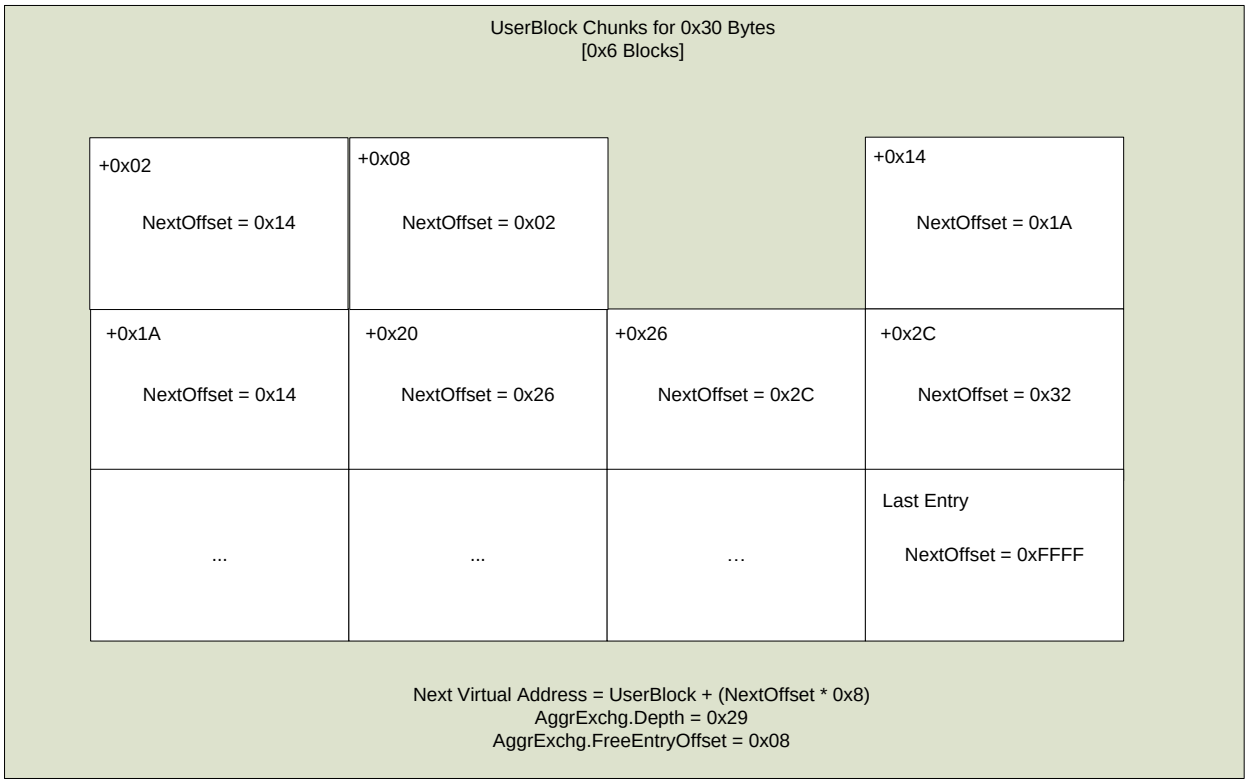

Diagram 13. Freeing of the 2nd chunk allocated for 0x30 bytes

Security Mechanisms

While most of the security mechanisms introduced in Windows XP SP2 are still intact in Windows 7 there are a few more introduced in Windows 7 (Windows Vista code base). In this section we'll discuss some of these security measures, how they are implemented and finally some thoughts about each mechanism. That being said, I think overall the protection logic introduced in the Windows Vista code base produces the hardest-to-exploit Windows heap to date.

Heap Randomization

The goal of heap randomization is to ensure that the **HeapBase** has an unpredictable address. Each time a heap is created a random value is added to the base address in attempt to prevent predictable memory addresses.

This happens in **RtlCreateHeap()** by creating a random 64k-aligned value to be added to the **HeapBase**. This value will attempt to produce a varying address each time the application is executed. Randomization can depend on the maximum size of the heap, which is calculated from arguments passed to **HeapCreate()**. The following is a code snippet from RtlCreateHeap() that attempts to randomize **HeapBase**:

Listing 34. RtlCreateHeap randomization

```
int BaseAddress = Zero;
int RandPad = Zero;

//get page aligned size to use as a random pad
int RandPad = (RtlpHeapGenerateRandomValue64() & 0x1F) << 0x10;

//if maxsize + pad wraps, null out the randpad
int TotalMaxSize = MaximumSize + RandPad;
if(TotalMaxSize < MaximumSize)
{
       TotalMaxSize = MaximumSize;
       RandPad = Zero;
}

//0x2000 = MEM_RESERVE
//0x40 = PAGE_EXECUTE_READWRITE
//0x04 = PAGE_READWRITE
//this will reserve the memory at the baseaddress
//but NOT actually commit any memory at this point
int Opts = 0x4;
if(Options & 0x40000
   Opts = 0x40;

if(NtAllocateVirtualmemory(-1, &BaseAddress, 0x0, &TotalMaxSize, 0x2000, Opts)
       return 0;

Heap = (_HEAP*)BaseAddress;

//adjust the heap pointer by randpad if possible
if(RandPad != Zero)
{
       if(RtlpSecMemFreeVirtualMemory(-1, &BaseAddress, &RandPad, 0x8000) >= 0 )
```

```
        {
                Heap = (_HEAP*)RandPad + BaseAddress;
                MaximumSize = TotalSize - RandPad;
        }
}
```

Comments

There are only a limited number of base addresses for the heap when using randomization due to the fact that they will be **64k-aligned** (5-bits of entropy). While guessing heap addresses based on the lack of entropy may not be practical, it is certainly not impossible.

Another less likely scenario stems from the fact that if *RandPad + MaximumSize* wraps, the **RandPad** will be NULL. This will effectively void the random generation of heaps. There's two specific reasons I claim that this is very unlikely. The first being the lack of control on arguments passed to HeapCreate(). While I'm sure this happens in applications, it isn't that common. The second reason is getting a MaximumSize large enough to wrap will most likely cause **NtAllocateVirtualmemory()** to return NULL, resulting in total failure.

Header Encoding/Decoding

Before Windows Vista, the only way to ensure an uncorrupted chunk was to validate the 1-byte cookie that resided in the chunk **header**. This obviously wasn't the most robust situation due to the fact that the cookie could be brute forced but more importantly, there was header data **before** it (McDonald/Valasek 2009).

This brings us to the advent of encoding heap chunk headers. The heap now encodes the **1st 4-bytes** of each _HEAP_ENTRY. This prevents the desired effect of a **Size**, **Flags** or **Checksum** overflow. The encoding is accomplished by **XOR**ing the first 3 bytes together and storing it in the **SmallTagIndex** variable, then the first 4-bytes are XORed with the **Heap->Encoding** (which is randomly created in **RtlCreateHeap()**).

Listing 35. Heap header encoding

```
EncodeHeader(_HEAP_ENTRY *Header, _HEAP *Heap)
{
        if(Heap->EncodeFlagMask)
        {
                Header->SmallTagIndex =
                        (BYTE)Header ^ (Byte)Header+1 ^ (Byte)Header+2;
                (DWORD)Header ^= Heap->Encoding;
        }
}
```

Decoding the chunk is quite similar to encoding, but it has a few extra checks before decoding is actually done. It makes sure that the chunk **header** to be decoded was encoded in the first place due to scenarios where heap chunk headers are **not** encoded.

Listing 36. Heap header decoding

```
DecodeHeader(_HEAP_ENTRY *Header, _HEAP *Heap)
{
	if(Heap->EncodeFlagMask && (Header & Heap->EncodeFlagMask))
	{
		(DWORD)Header ^= Heap->Encoding;
	}
}
```

Comments

Encoding the first 4-bytes of the chunk header makes it nearly impossible to successfully overwrite the **Size**, **Flags**, or **Checksum** fields without using an information leak (Hawkes 2008). But this doesn't stop you from overwriting other information in the header: If a header can be overwritten and used before values are check-summed, then it can potentially be used to change the flow of execution. We'll talk more about this in later sections.

The other possible evasions deal with overwriting values resulting in the heap manager believing the chunk is **not** encoded. This can be done a few ways. The first and least likely way is to NULL out the **Heap->EncodeFlagMask** (which is initialized to 0x100000). Any future decoding or encoding operation will not be performed. This does have some drawbacks due to the eminent heap instability to follow. A new heap would likely have to be created to reap the overall desired effects (Un-encoded overwrites of _HEAP_ENTRY headers).

The second and more likely scenario involves in overwriting the first 4-bytes of the chunk header so that a bitwise **AND** with the **Heap->EncodeFlagMask** returns *false*; giving you limited control over the **Size**, **Flags**, and **Checksum**. This is only truly useful for overwriting headers in the **FreeLists,** as checksum validation is done in the **allocation** routine.

Finally, an attacker could target the **last 4 bytes** of the chunk header. We have seen previously that these fields are used to determine the state of a chunk header. For example, offset 0x7 from the chunk header will be used to determine if it came from the **LFH** or the **back-end**.

Death of bitmap flipping

In Brett Moore's *Heaps about Heaps* there was a discussion on how to trick a FreeList into believing it was populated or unpopulated when it actually was not. This was, from there on out, referred to as **bitmap flipping** (Moore 2008). This exploitation attack was directly addressed in the newer versions of Windows due to a highly technical and critically acclaimed paper written in 2009 (McDonald/Valasek 2009) [**Note**: This is 100% untrue].

In the Windows XP code base, the XOR operator was used to update the bitmap. If the logic to reach the update operation was reached and the FreeList was empty, it would XOR the current bit in the bitmap with itself, therefore inversing its value.

Listing 37. Death of Bitmap Flipping

```
// if we unlinked from a dedicated free list and emptied it,clear the bitmap
if (reqsize < 0x80 && nextchunk == prevchunk)
{
    size = SIZE(chunk);
    BitMask = 1 << (size & 7);

    // note that this is an xor
    FreeListsInUseBitmap[size >> 3] ^= vBitMask;
}
```

The problem with this technique was that if the size of the chunk was corrupted, you could change the state of a dedicated FreeList when it shouldn't have been altered; resulting in the ability to overwrite critical data in the **HeapBase**.

When updating the bitmap to display an empty list, bitwise AND is used, ensuring that an **empty** list will stay empty and a populated list can only be marked as empty. The same goes for marking a list as **populated**, which uses a bitwise OR to alter the **ListsInUseUlong**. That way, an empty list can become populated but a populated list being freed to can't become **unpopulated**.

On top of all these changes, the concept of dedicated Freelists has disappeared, so an attempt to allocate from an *empty* list will only traverse the FreeList structure for a sufficiently sized chunk.

Listing 38. Death of Bitmap Flipping 2

```
//HeapAlloc
size = SIZE(chunk);
BitMask = 1 << (Size & 0x1F);
BlocksIndex->ListInUseUlong[Size >> 5] &= ~BitMask;

//HeapFree
size = SIZE(chunk);
BitMask = 1 << (Size & 0x1F);
BlocksIndex->ListInUseUlong[Size >> 5] |= BitMask;
```

Safe Linking

Safe unlinking was introduced in Windows XP SP2 to prevent 4-byte overwrites from occurring when unlinking a chunk from a FreeList (or coalescing two free chunks together). This has prevented much of the *generic* heap exploitation ever since.

Although unlinking an item from a list could no longer be used to overwrite arbitrary addresses, you could overwrite an entry's **blink** on FreeList[0] to point to an address you wanted to overwrite (Moore 2005). That way, if a chunk was to be inserted before the compromised entry, the blink would now point to the address of the chunk that was just freed.

New checks in the in the back-end manager validate a chunk's **blink** before linking in a free chunk. We saw this code above when explaining how **RtlpFreeHeap()** worked, but let's review it again. You can see that if a FreeList's Blink->Flink does **not** point to itself, it is considered corrupted and no link-in process is performed.

Listing 39. Safe Linking

```
if(InsertList->Blink->Flink == InsertList)
{
	ChunkToFree->Flink = InsertList;
	ChunkToFree->Blink = InsertList->Blink;
	InsertList->Blink->Flink = ChunkToFree;
	InsertList->Blink = ChunkToFree
}
else
{
	RtlpLogHeapFailure();
}
```

Comments

While this device prevents the overwriting of pointers via a compromised Blink, it still presents a problem if the process doesn't terminate after **RtlpLogHeapFailure()**. The code directly after this inserts the chunk into its appropriate spot in the **ListHints** without actually updating its **flink and blink**. This means that the flink and blink are completely user-controlled.

Tactics

Heap Determinism

For years researchers have been focused on corrupting heap meta-data to change the flow of execution. This once trivial task has become much more difficult. Generic write-4-attacks are extinct, heap headers are now encoded with pseudo-random values to protect their integrity, and now even a trick as old as the FreeList insertion attack is basically dead.

Now, more than ever, exploits must have a high degree of precision when attempting to setup advantageous situations. When talking about the heap, we call this **heap manipulation**. Command over chunk sizes, allocations and frees truly make a difference in modern Windows heap exploitation.

In this section I will try to discuss some common scenarios that take place when attempting to get the heap in a deterministic state using the **Low Fragmentation heap.** Such as, where will allocation X be located? What will be adjacent allocation X? If chunk X is freed what will happen?

Activating the LFH

One of the most basic pieces of information to know is how to activate the **LFH** for a specific **Bucket**. The LFH is the only front-end heap manager for Windows 7 but that doesn't mean it will handle every allocation for a specific chunk size by default. As shown in the code above, the LFH must be *activated* via a heuristic in the back-end. If you can force allocations to be made, which is quite common, then you can enable the LFH for a certain size. The LFH can be activated by making at least **0x12** consecutive allocations of the same size (or 0x11 if the LFH has previously been activated).

Listing 40. Enable the LFH for a specific size

```
//0x10 => Heap->CompatibilityFlags |= 0x20000000;
//0x11 => RtlpPerformHeapMaintenance(Heap);
//0x11 => FreeList->Blink = LFHContext + 1;
for(i = 0; i < 0x12; i++)
{
	printf("Allocation 0x%02x for 0x%02x bytes\n", i, SIZE);
	allocb[i] = HeapAlloc(pHeap, 0x0, SIZE);
}

//now that the _HEAP_BUCKET is in the
//ListHint->Blink, the LFH will be used
printf("Allocation 0x%02x for 0x%02x bytes\n", i++, SIZE);
printf("\tFirst serviced by the LFH\n");
allocb[i] = HeapAlloc(pHeap, 0x0, SIZE);
```

As you can see, activating and enabling the LFH for specific sizes is quite trivial if you have the ability to control allocations, such as allocating DOM objects. Now that the front-end manager is being used for specific sizes, steps can be taken to provide greater levels of determinism.

Defragmentation

Before we can talk about placing chunks adjacently in the LFH, the topic of **fragmentation** must be discussed. Due to the frequent allocation/de-allocation routines of most applications, the **UserBlock** chunk will be **fragmented**. This means some **defragmentation** must take place to ensure that a chunk that you are overflowing will be adjacent to the chunk you are overwriting.

In the next example we will have a prerequisite of chunks residing directly adjacent together. While this is quite trivial when dealing with a **new SubSegment** because there aren't any holes yet, this won't be the case when using a SubSegment that has been in use. The diagram below shows that a single allocation will not procure **three** adjacent objects. The **holes** must be filled for this to happen. The easiest way to do this is to have some understanding of the application's current memory layout and make some allocations.

Gray = BUSY
Blue = FREE

0x08	0x0E	0x14	0x1A	0x20	0x26	0x2C	0x32
0x38	0x3E	0x44	0x4A	0x50	0x56	0x5C	0x62

In this scenario, we only needed to make **three** allocations until we've filled all the holes, but obviously this isn't a realistic example. An attacker won't likely know exactly how many holes he/she will have to fill and will probably need to make enough allocations to completely exhaust the **UserBlocks** (Depth == 0x0). This will force the heap manager to create a **new SubSegment** which will not contain any holes. (Note: Thanks Alex/Matt).

Gray = BUSY
Blue = FREE

0x08	0x0E	0x14	0x1A	0x20	0x26	0x2C	0x32
0x38	0x3E	0x44	0x4A	0x50	0x56	0x5C	0x62

Adjacent Data

One of the most difficult tasks when attempting to exploit a heap-based buffer overflow is manipulating the heap into a well known state. You want to ensure that the chunk you are overflowing is directly adjacent to the chunk to be overwritten. Normally with the back-end heap, **coalescing** can be a problem when memory is freed, leading to unreliability when attempting exploitation.

> **Note:** This was the most difficult task when attempting to exploit the heap overflow vulnerability used during the demo for *Practical Windows XP/2003 Heap Exploitation*. The multi-threaded nature of the application would unreliably coalesce chunks, rendering most our heap manipulation ineffective.

The LFH does not coalesce chunks because they are all of the same size; hence indexed relative to their offset from the **UserBlock**. This makes placing chunks of the same size next to each other trivial. If an overflow is possible, **BUSY** and **FREE** chunks could be overwritten depending on the current state of the **UserBlocks**.

Let's say you want to overwrite information in **alloc3** and you have a **write-n** situation As long as your allocation which can be overflowed comes before **alloc3**, you will be able to overwrite the data in alloc3.

Listing 42. LFH Chunk overflow

```
EnableLFH(SIZE);
NormalizeLFH(SIZE);

alloc1 = HeapAlloc(pHeap, 0x0, SIZE);

alloc2 = HeapAlloc(pHeap, 0x0, SIZE);
memset(alloc2, 0x42, SIZE);
*(alloc2 + SIZE-1) = '\0';

alloc3 = HeapAlloc(pHeap, 0x0, SIZE);
memset(alloc3, 0x43, SIZE);
*(alloc3 + SIZE-1) = '\0';

printf("alloc2 => %s\n", alloc2);
printf("alloc3 => %s\n", alloc3);

memset(alloc1, 0x41, SIZE * 3);

printf("Post overflow..\n");
printf("alloc2 => %s\n", alloc2);
printf("alloc3 => %s\n", alloc3);
```

Listing 43. LFH Chunk overflow result

```
Result:
alloc2 => BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
alloc3 => CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

Post overflow..

alloc2 => AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
          AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACCCCCC
          CCCCCCCCC
alloc3 => AAAAAAAAAAAAAAAAAAAAAAACCCCCCCCCCCCCC
```

As you can see some of the data that was present in **alloc3** is now overwritten by the data written to **alloc1**. Another side effect that should be noted is the string length of **alloc2** after the overflow occurs. The length is actually that of **alloc2** and **alloc3** combined since the null terminator was overwritten. Please read Peter Vreugdenhil's fantastic paper (Vreugdenhil 2010) for a real world example of overwriting a null terminator to help in gaining code execution.

But what if **alloc2** is used or has its header validated before the data in **alloc3** is used? You will need to position the chunk that can be overflowed **directly before** the chunk that is being overflowed into. While this may seem trivial, fragmentation must be taken into account along with ability to control allocations and frees.

Listing 44. Chunk reuse

```
alloc1 = HeapAlloc(pHeap, 0x0, SIZE);
alloc2 = HeapAlloc(pHeap, 0x0, SIZE);
alloc3 = HeapAlloc(pHeap, 0x0, SIZE);

HeapFree(pHeap, 0x0, alloc2);

//overflow-able chunk just like alloc1 could reside in same position as alloc2
alloc4 = HeapAlloc(pHeap, 0x0, SIZE);

memcpy(alloc4, src, user_controlled_size)
```

Note: Although manipulating the heap into ordering chunks adjacently has become easier in the LFH, there is one huge drawback. The ability to place two chunks of **different sizes** has become much more complicated (Involving multiple **SubSegments** in adjacent memory). This can hinder the exploitation process if the ability to control allocation sizes is not fully controlled.

Seeding Data

At the time of writing, *use-after-free* bugs are very popular. Most of the exploits for these vulnerabilities contain code trying various allocation methods (JavaScript strings, DOM object instances, etc) in an attempt to seed data within the heap. Nico Waisman called this technique *pray-after-free* (Waisman 2010) due to the lack of understanding regarding object data.

We already know how the **LFH** stores memory in large **UserBlocks** which are divided into **BucketSize** chunks. We also know that each one of these chunks in the UserBlock can be adjacent to each other depending on the ability to control allocations and frees. Using this knowledge the contents of each chunk can be **seeded** by writing data to user-writable memory [This is contingent on the HEAP_ZERO_MEMORY flags not being set in the call to **HeapAlloc()**] (http://msdn.microsoft.com/en-us/library/aa366597%28VS.85%29.aspx).

The following example shows how memory can be copied to chunks on the LFH, subsequently freed, and then allocated again without losing much of the original data.

Listing 45. Data seeding

```
EnableLFH(SIZE);
NormalizeLFH(SIZE);

for(i = 0; i < 0x10; i++)
{
	printf("Allocation 0x%02x for 0x%02x bytes => ", i, SIZE);
	allocb[i] = HeapAlloc(pHeap, 0x0, SIZE);
	memset(allocb[i], 0x41 + i, SIZE);
	for(j=0; j<12; j++)
		printf("%.2X", allocb[i][j]);
	printf("\n");
}

printf("Freeing all chunks!\n");
for(i = 0; i < 0x10; i++)
{
	HeapFree(pHeap, 0x0, allocb[i]);
}

printf("Allocating again\n");
for(i = 0; i < 0x10; i++)
{
	printf("Allocation 0x%02x for 0x%02x bytes => ", i, SIZE);
	allocb[i] = HeapAlloc(pHeap, 0x0, SIZE);
	for(j=0; j<12; j++)
		printf("%.2X", allocb[i][j]);
	printf("\n");
}
```

Listing 46. Data seeding results

```
Result:
Allocation 0x00 for 0x28 bytes => 41414141 41414141 41414141
Allocation 0x01 for 0x28 bytes => 42424242 42424242 42424242
Allocation 0x02 for 0x28 bytes => 43434343 43434343 43434343
Allocation 0x03 for 0x28 bytes => 44444444 44444444 44444444
Allocation 0x04 for 0x28 bytes => 45454545 45454545 45454545
Allocation 0x05 for 0x28 bytes => 46464646 46464646 46464646
Allocation 0x06 for 0x28 bytes => 47474747 47474747 47474747
Allocation 0x07 for 0x28 bytes => 48484848 48484848 48484848
Allocation 0x08 for 0x28 bytes => 49494949 49494949 49494949
Allocation 0x09 for 0x28 bytes => 4A4A4A4A 4A4A4A4A 4A4A4A4A
Allocation 0x0a for 0x28 bytes => 4B4B4B4B 4B4B4B4B 4B4B4B4B
Allocation 0x0b for 0x28 bytes => 4C4C4C4C 4C4C4C4C 4C4C4C4C
Allocation 0x0c for 0x28 bytes => 4D4D4D4D 4D4D4D4D 4D4D4D4D
Allocation 0x0d for 0x28 bytes => 4E4E4E4E 4E4E4E4E 4E4E4E4E
Allocation 0x0e for 0x28 bytes => 4F4F4F4F 4F4F4F4F 4F4F4F4F
Allocation 0x0f for 0x28 bytes => 50505050 50505050 50505050

Freeing all chunks!
Allocating again

Allocation 0x00 for 0x28 bytes => 56005050 50505050 50505050
Allocation 0x01 for 0x28 bytes => 50004F4F 4F4F4F4F 4F4F4F4F
Allocation 0x02 for 0x28 bytes => 4A004E4E 4E4E4E4E 4E4E4E4E
Allocation 0x03 for 0x28 bytes => 44004D4D 4D4D4D4D 4D4D4D4D
Allocation 0x04 for 0x28 bytes => 3E004C4C 4C4C4C4C 4C4C4C4C
Allocation 0x05 for 0x28 bytes => 38004B4B 4B4B4B4B 4B4B4B4B
Allocation 0x06 for 0x28 bytes => 32004A4A 4A4A4A4A 4A4A4A4A
Allocation 0x07 for 0x28 bytes => 2C004949 49494949 49494949
```

```
Allocation 0x08 for 0x28 bytes => 26004848 48484848 48484848
Allocation 0x09 for 0x28 bytes => 20004747 47474747 47474747
Allocation 0x0a for 0x28 bytes => 1A004646 46464646 46464646
Allocation 0x0b for 0x28 bytes => 14004545 45454545 45454545
Allocation 0x0c for 0x28 bytes => 0E004444 44444444 44444444
Allocation 0x0d for 0x28 bytes => 08004343 43434343 43434343
Allocation 0x0e for 0x28 bytes => 02004242 42424242 42424242
Allocation 0x0f for 0x28 bytes => 62004141 41414141 41414141
```

> **Note:** If all the chunks inside a **HeapBin** are freed, then the entire Bin itself is freed. This means that a full *plunger* effect isn't possible like it was with the **Lookaside List**. Jay-Z suggests allocating a **large** amount of chunks before doing multiple frees; minimizing the chance of the **HeapBin** getting too small and subsequently freed.

The first allocation printout shows the memory written to each chunk in the LFH for size **0x28** (0x30 after adding space for the _HEAP_ENTRY). All the chunks are freed and then allocated in the same fashion in *step 1*. Although this time you don't see a call to **memset**, the data in the chunks is strikingly similar.

This is because none of the chunks were coalesced nor was the data cleared from them. The only piece of information that seems to have changed is the first 2 bytes of the data. This is the saved **FreeEntryOffset** that was talked about in the *Algorithms* section. The **FreeEntryOffset** is saved in the 2 bytes directly following the chunk **header** for future use by the heap manager.

It should also be noted that the chunks are in *reverse* order from when they were originally pieced out of the **UserBlock**. It's not that the memory actually inverted itself but only that the **FreeEntryOffset** was re-indexed after every **HeapFree()**. So what can you accomplish by knowing the size of an allocation and stale data? *Use-after-free* vulnerabilities are a perfect candidate for leveraging the ability to make allocations of a known size.

Let's assume that an object has a size of **0x30** bytes and has a **vtable** at offset 0x0. This object is used, garbage collected and then incorrectly used after the memory has been freed. This leaves the opportunity for overwriting of the data **before** it is used after the free; giving the attacker control (or semi-control) of what address is used as the **vtable**.

Since the size is known and we can control an allocation before the *free object* is used, this will give us complete control over what address is used for the object's vtable. Having control over this will give us the ability to change the flow of execution to our supplied address and corresponding payload.

Listing 47. Use-after-free contrived example (Sotoriv 2007)

```
//LFH is enabled

//0x30 byte object
var obj = new FakeObj();

//.
//.
//.
//obj goes out of scope & garbage collected
//.
//.
//.

var heap = new heapLib.ie();

//this will allocate the same location
//in memeory as obj
heap.alloc("A" * 0x30, "overwrite");

//vtable == 0x41414141
obj.DoStuff();
```

Note: Although this is a simplistic example, it still shows how knowledge of the heap manager can result in more precise exploitation. If you are reading this paper before Blackhat USA 2010 I HIGHLY recommend attending Nico Waisman's talk, *Aleatory Persistent Threat*. If not please take some time to read his paper of the same title. This will provide practical applications of the topics discussed in this paper. </nico_plug>

Exploitation

Numerous protection mechanisms have been created since the popularity of heap exploitation became prevalent, from safe-unlinking to encoded chunk headers; heap exploitation has continued to increase in difficulty. Meta-data corruption now is generally used to overwrite application data that resides in the heap, rather than directly using the corruption to achieve code execution. That being said, it should not be considered impossible; meta-data corruption can still be used to gain code execution, albeit difficult.

In this section I will discuss previously detailed techniques along with some new techniques for exploiting heap meta-data. While this meta-data corruption doesn't get an *easy* write-4, it will show how to manipulate data to achieve code execution, provided a few prerequisites are met.

Ben Hawkes #1

Ben Hawkes' RuxCon 2008 paper (Hawkes 2008) not only gave a great overview of Windows Vista managed memory, but also came up with new techniques for leveraging code execution through heap corruption. I recommend reading this paper as prerequisite / supplementary reading for this paper; specifically this section. While he came up with numerous techniques, most impressively his **Heap HANDLE payload**, I will only discuss one technique in this paper.

Dr. Hawkes specifically talks about corrupting chunks in the **UserBlock** managed by the **LFH**. When freeing a chunk in **RtlpLowFragHeapFree()** it checks to see if the _HEAP_ENTRY has a **UnusedBytes** (offset 0x7) equal to **0x5**. If that is the case the function will use the **SegmentOffset** (offset 0x6) as an index to a different chunk header.

Listing 48. Chunk header relocation

```
_HEAP_ENTRY *ChunkHeader = ChunkToFree - sizeof(_HEAP_ENTRY);
if(ChunkHeader->UnusedBytes == 0x5)
      ChunkHeader -= 8 * (BYTE)ChunkHeader->SegmentOffset;
```

A normal chunk on the LFH will have a header that looks like this:

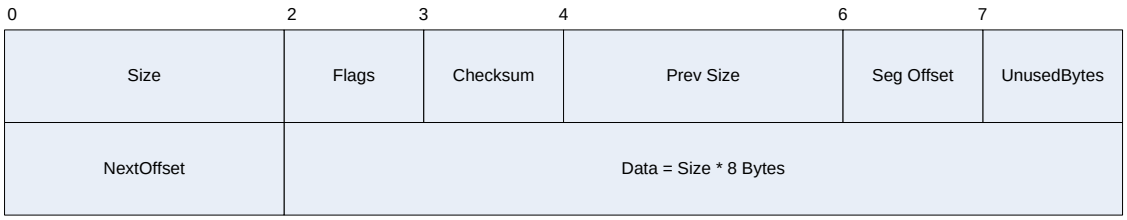

Diagram 14. HeapBin chunk

Note: The 'NextOffset' isn't actually a separate field but it resides in the first 2-bytes of the user-writeable data.

If you can place an overflow-able chunk directly before the chunk to be freed, the **UnusedBytes** can be assigned a value of **0x5** and the **SegOffset** can be a 1-byte value of your choosing. This gives you 0xFF * 8 bytes of **negative** direction to choose from.

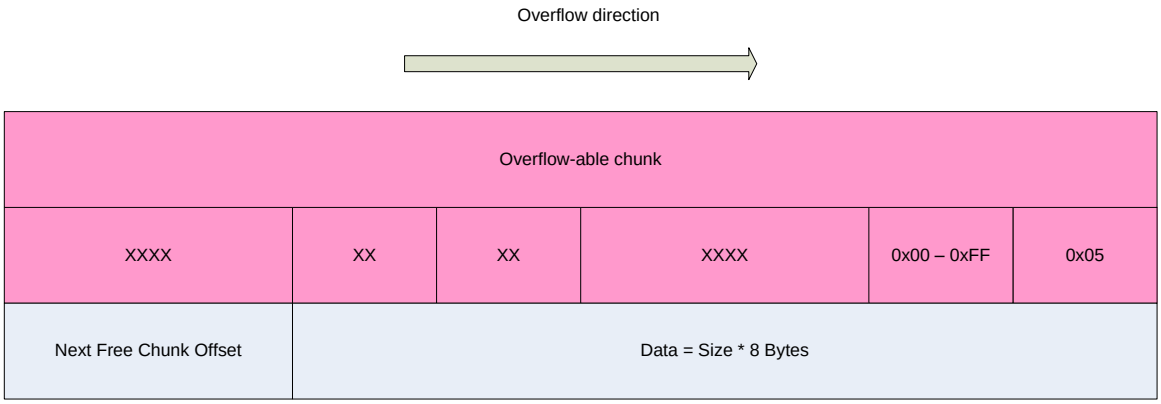

Diagram 15. Overwritten HeapBin chunk

The **ChunkHeader** will be reassigned a value based on the overwritten **SegOffset,** but must be a valid **_HEAP_ENTRY**, since RtlpLowFragHeapFree() must free it. This may seem useless at first but let's look at a code sample.

This highly simplistic, contrived example (ignore scope) takes input from a source and attempts to create objects and allocate memory for future use. There is a trivial buffer overflow due to a miscalculation in the amount of memory to read. I will show you how this overflow can lead to the overwriting of application data.

Listing 49. C++ contrived example

```
class Paser
{
public:
	Parser();
	virtual void DoThings();
	~Parser();

private:
	int Items;
	int Values;
	int Stuff;
	int Things;
};


void *buffer, *output;
int action, copy_size = 0x40;
Parser *p;

while(1)
{
	action = ReadInt();


	if(action == 0x1)
		p = new Parser();
```

```
        else if(action == 0x2)
                buffer = malloc(sizeof(Parser));
        else if(action == 0xFF)
                break;

        action = ReadInt();
        if(action == 0x3)
                p.DoThings();
        else if(action == 0x4)
                ReadBytes(buffer, copy_size);
        else if(action == 0x5)
                free(buffer);
        else if(action == 0x6)
                delete p;
        else if(action == 0x7)
                ReadBytes(buffer, 0x10);
}
```

The first order of business is to enable the LFH for **sizeof(Parser)**; then set up memory so that a Parser object is located **behind** an allocation which you can overflow. Another allocation must be made, so that the overflowed chunk headers do not get reassigned in **RtlpLowFragHeapAllocFromContext()** after an allocation. This will permit the overwriting an adjacent header with user controlled values.

Diagram 16. Chunk setup

After overwriting the header in **Alloc2** with values that will adjust the chunk **header** to point to the parser object, we then free **Alloc2**, which will actually make the memory location of the **parser object** the next available chunk.

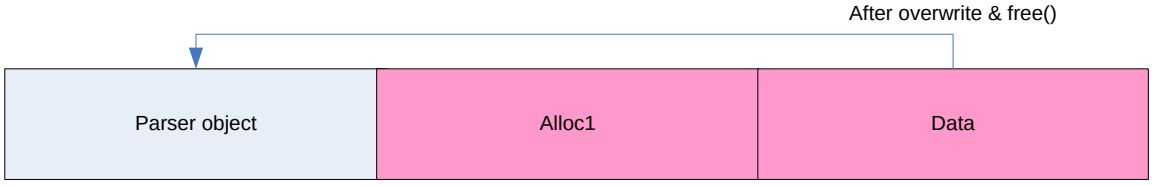

Diagram 17. Chunk overwrite and free

Now our next allocation will be given the address of the **parser object**, which after overwriting the **vtable**, will be under our control. This requires a third allocation and then a call to **p.DoThings()**.

Step-by-Step

1. Enable the LFH for sizeof(Parser)
2. Allocate a Parser object
3. Allocate a chunk of memory which can be overflowed (Alloc1)
4. Allocate a chunk of memory to be overwritten (Alloc2)
5. Overflow from Alloc1 into Alloc2 overwriting the chunk header with UnusedBytes equal to 0x5 and the SegOffset equal to the amount of blocks to point to the parser object
6. Free Alloc2
7. Allocate a chunk of memory and write your desired contents. In this scenario you will be overwriting a pointer to the parser object's vtable (Alloc3)
8. Trigger a call to the parser object (p.DoThings() for example)

Prerequisites

- Control over allocations of a certain size
- Ability to enable the LFH for specific size
- Place a legitimate LFH chunk before the chunk that is overflowed
- Overflow at least 8-bytes, altering the chunk header of an adjacent piece of memory
- Ability to free overwritten chunks

Methodology

1. Enable LFH
2. Normalize the heap
3. Alloc1
4. Alloc2
5. Alloc3
6. Overwrite Alloc3 (At least 8-btyes)
7. Free Alloc3 (Adjust header to point to Alloc1)
8. Alloc4 (Actually points to Alloc1)
9. Write data (Tainting memory in Alloc1)
10. Use Alloc1

FreeEntryOffset Overwrite

This section starts the *new* techniques devised while doing the research for this paper. In the previous section we talked about how the chunks in the **UserBlock** are managed by keeping track of the current **offset** and the offset of the **next** available chunk. The current offset is held in an **_INTERLOCK_SEQ** structure. This way, the LFH knows where to get the **next free** chunk.

The main problem with this is the next **FreeEntryOffset** is stored in the first 2 bytes of user-writable data. This means the allocator must acquire a value based off the **size** of the chunk being managed; subsequently storing it for the next iteration. Since each chunk in the LFH can be directly adjacent to one another and the chunk headers aren't verified on allocations, the **FreeEntryOffset** can be overwritten with a new *Offset*; making later allocations point to semi-arbitrary locations.

Listing 50. Try/Catch for LFH allocation

```
try
{
	//the next offset is stored in the 1st 2-bytes of userdata
	short NextOffset =
		UserBlocks + BlockOffset + sizeof(_HEAP_ENTRY));

	_INTERLOCK_SEQ AggrExchg_New;
	AggrExchg_New.Offset = NextOffset;
}
catch
{
	return 0;
}
```

With this knowledge and at least a **0x9** (**0xA** preferred) byte overwrite, you can influence the value of **NextOffset**, affecting an allocation made directly afterwards. Let's see a half-rate example.

Listing 51. FreeEntryOffset ovewrite example

```
class Dollar
{
public:
	Dollar();
	virtual void INeedDollar();

private:
	int Job = 0;
	int Dollars = 0;
	unsigned int Booze = 0xFFFFFFFF
};

int first_len, second_len, total_len;
char *dollar1, *dollar2, *dollar3, *data1, *data2;
char *statement = "I need dollar";

first_len = ReadInt();
second_len = ReadInt();

data = alloc(first_len);
ReadBytes(data, first_len);
```

```
//int wrap
total_len = first_len + second_len;

//we can write more data than this
//allocation can hold
dollar1 = alloc(total_len);
memcpy(dollar1, data, first_len);

//this will store the tainted FreeEntryOffset
//in the _INTERLOCK_SEQ
dollar2 = alloc(total_len);
memcpy(dollar2, statement, strlen(statement));

//although not on the same UserBlock, it will be
//allocated on an adjacent page in memory
Dollar dollars[0x20];
for(i = 0; i < 0x20; i++)
{
	dollars[i] = new Dollar();
}

//this will give us 0xFFFF * 0x8 bytes of
//forward range when making an allocation, more
//than enough room to overwrite any of the Dollar objects
dollar3 = alloc(first_len);
memcpy(dollar3, data, first_len);

//one or more of these can be overwritten
//with data copied into dollar3
for(i = 0; i < 0x20; i++)
{
	dollars[i].INeedDollar();
}
```

There is an obviously integer wrap followed by a potential buffer overflow at the very beginning of this simple function, but contrary to previous exploitation methods, we cannot simply overwrite some meta data and gain code execution. But, by overwriting the **FreeEntryOffset** stored in the data of **dollar2** we can control the address returned for **dollar3**.

Let's assume that the LFH is enabled for **Bucket[0x6]** (0x30 bytes) and also assume that **dollar1** is the first chunk to be allocated for Bucket[0x6] (This is just for simplicity's sake. It does not actually need to be the first allocation from the LFH for a specific size). The state of the **UserBlock** should look like this:

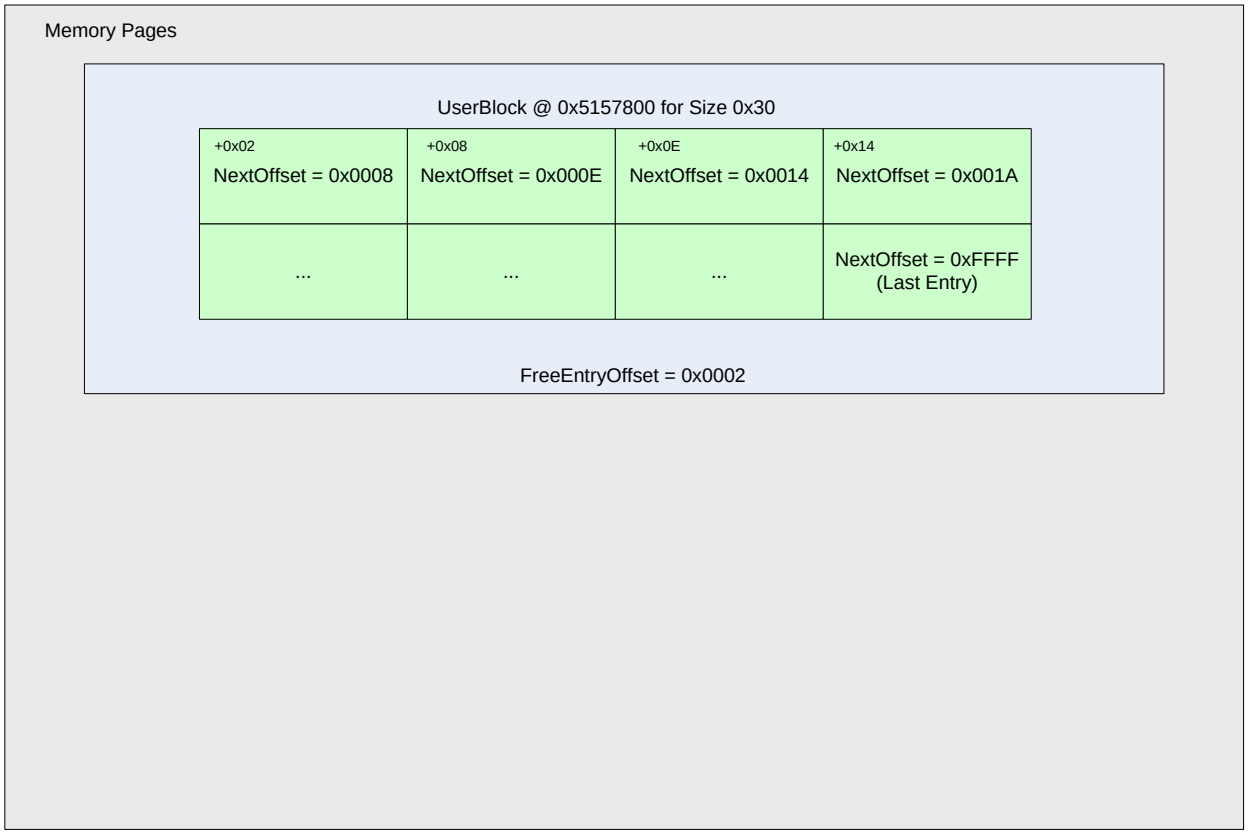

Diagram 18. Userblock after chunking

After **dollar1** is allocated and a subsequently overflowed into the next free heap chunk, the **FreeEntryOffset** is updated to the offset of the next chunk (which is 0x0008).

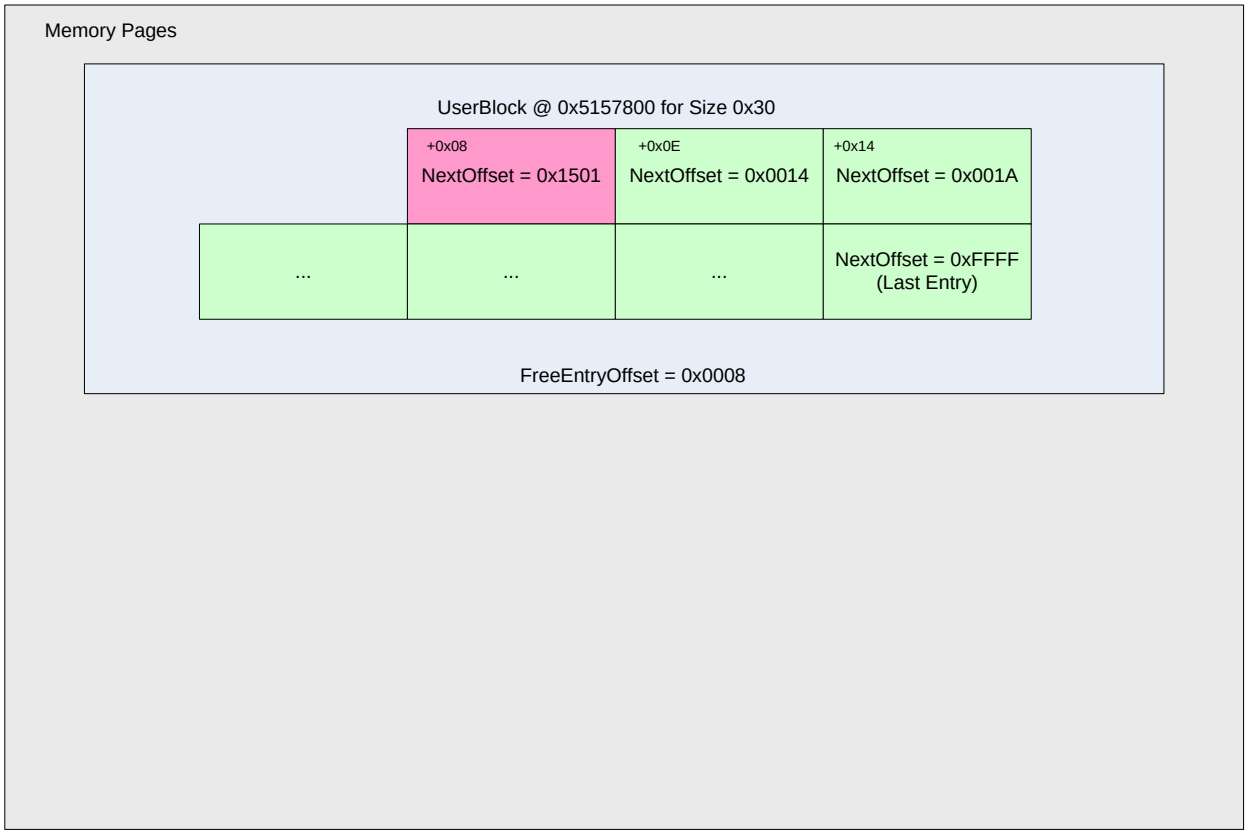

Diagram 19. FreeEntryOffset Overwrite

At this point the next offset has been overwritten with a value that we control, but to use this offset you will need to invoke another allocation; storing the value in the **_INTERLOCK_SEQ** for future use. This allocation, in this example, occurs when allocating memory for **dollar2**.

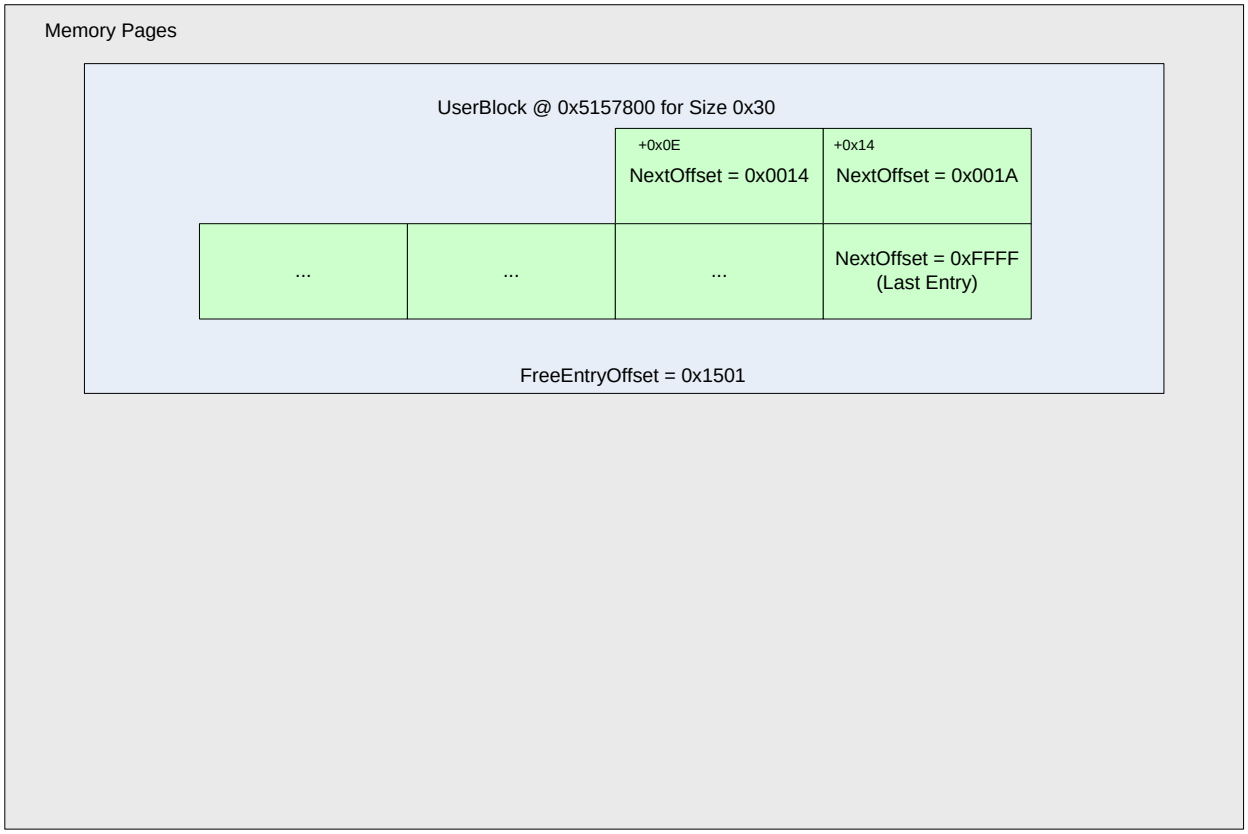

Diagram 20. Second allocation, setting overwritten FreeEntryOffset

Now we have a **FreeEntryOffset** of **0x1501**. This value will write off the page allocated for this UserBlock, so adjacent memory is required for overwriting. (This is not always the case. If there is data perfect for an overwrite in this memory page, then the overwritten offset can be any value from 0x0000-0xFFFF).

The adjacent memory in this example is created when constructing the array of 0x20 **Dollar** objects (we will assume these are 0x40 bytes wide). Once the LFH is enabled for **Bucket[0x8]** (0x40 bytes), you will have an overall memory layout that looks similar to the following:

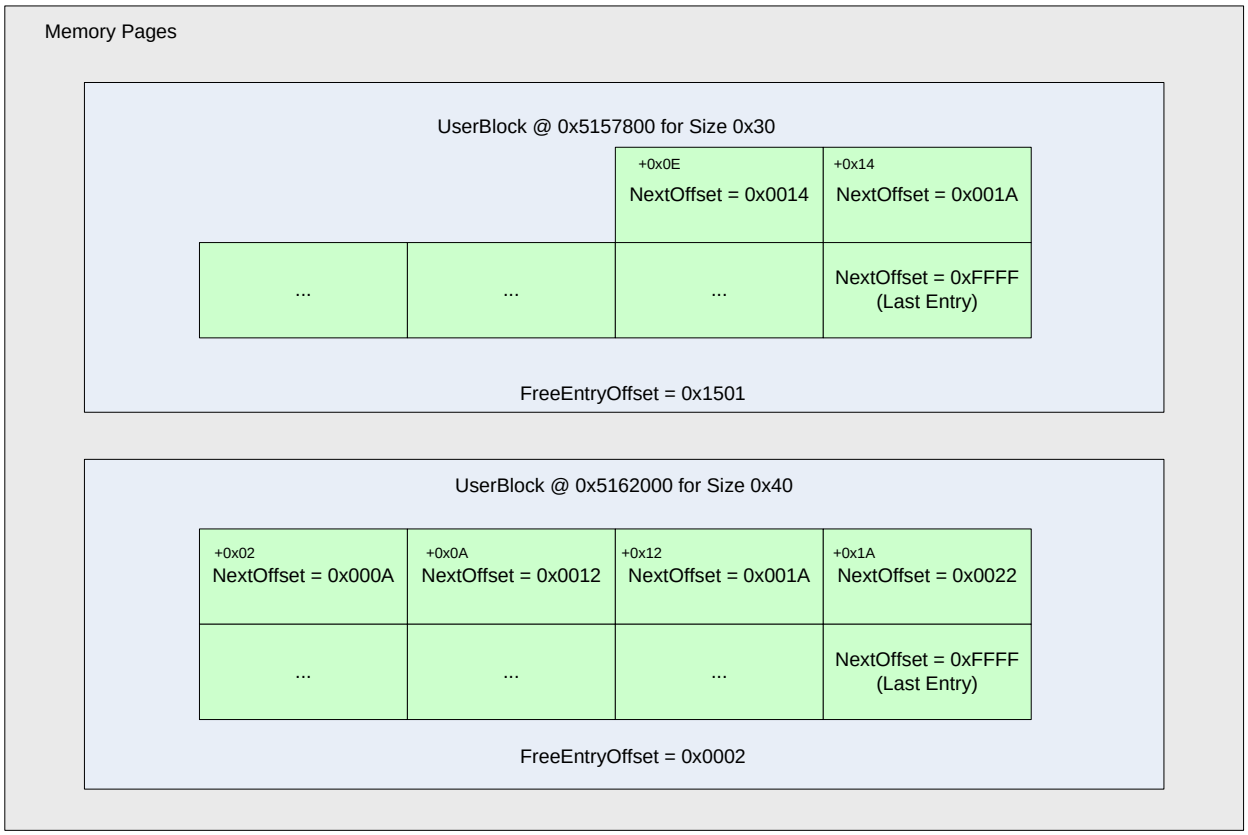

Diagram 21. Multiple UserBlocks

At this point there is a very advantageous situation for overwriting function pointers / vtables. The next allocation for 0x30 bytes will return the memory address 0x5162006, due to the next free entry being calculated by taking the address of the current UserBlock [0x5157800], adding the current offset into the UserBlocks [0x0E] (which is 2 entries of 0x6 blocks into the chunk), and finally adding the **FreeEntryOffset * 8** [0x1501].

```
NextChunk = UserBlock + Depth_IntoUserBlock + (FreeEntryOffset * 8)
NextChunk = 0x5157800 + 0x0E + (0x1501 * 8)
NextChunk = 0x5162016
```

This means once we allocate **dollar3** and write 0x30 bytes of data, we will overwrite an object in the **Dollar array**. The overwritten offset can even be tuned to point to a specific object within the array. Although this example is simplistic and attacks application data in the heap, it can still be used in varying ways to have n-byte overwrites.

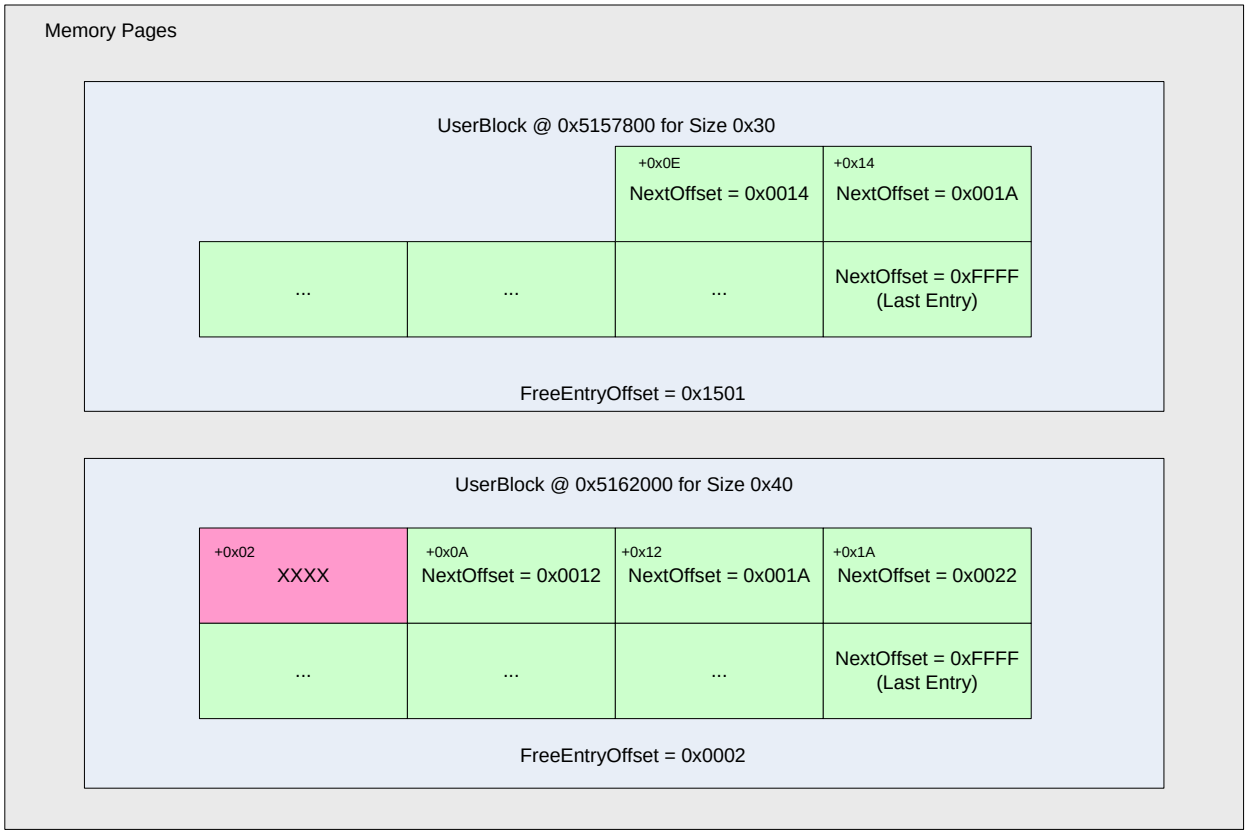

Diagram 22. Cross page overwrite

Step-by-Step

1. Enable the LFH
2. Allocate a chunk for the size you have enabled (dollar1)
3. Overflow dollar1 by at least 0x9 bytes (0xA is preferable) into a directly adjacent free chunk, which is to be allocated next
4. Allocate a chunk for the size you enabled (dollar2) [This will store the overwritten FreeEntryOffset in the _INTERLOCK_SEQ]
5. Allocate an object which is to be overwritten. This allocation needs to reside within the range available to the **FreeEntryOffset** (0x08 * 0xFFFF is the max). In this case, the Dollar objects are in an adjacent memory page.
6. Allocate a chunk for the size you enabled (dollar3) [This will return the address of your choosing based off the overwritten FreeEntryOffset]
7. Write n-bytes, overwriting an important object
8. Call overwritten function pointer / vtable

Prerequisites

- Ability to enable the LFH for a specific Bucket
- Control over allocations for a specific Bucket
- At least a 0x9 byte overwrite, 0xA is preferable
- An adjacent free block to overwrite
- An object to overwrite within the maximum range (0xFFFF * 0x8)
- A way to trigger the overwritten object

Methodology

1. Enable LFH
2. Normalize the heap
3. Alloc 1
4. Overwrite adjacent chunk's FreeEntryOffset
5. Alloc2
6. Alloc3
7. Write data to Alloc3 (Overwriting object of interest)
8. Trigger

Observations

Although the material presented in this section could very easily be put into the *Exploitation* section, I felt that it would be better placed under *Observations*. The reasoning behind this is the lack of reliability when attempting to use this technique to leverage code execution. The last thing I wanted to do was place an item of interest into the *Exploitation* section when it was in fact, as Sinan Erin would put it, ***strawberry pudding***.

SubSegment Overwrite

We saw in the *Allocation* section that the LFH will attempt to use a **SubSegment** when allocating memory. If there is no SubSegment available, it will allocate space for a **UserBlock** and then proceed to procure a SubSegment.

Listing 52. SubSegment acquisition

```
_HEAP_SUBSEGMENT *SubSeg = HeapLocalSegmentInfo->ActiveSubsegment;

if(SubSeg)
{
	while(1)
	{
		.
		.
		.
		//checks to ensure valid subsegment
		_HEAP_USERDATA_HEADER *UserBlocks =
			SubSeg->UserBlocks;
		if(!Depth ||
			!UserBlocks ||
			SubSeg->LocalInfo != HeapLocalSegmentInfo)
		{
			break;
		}
		.
		.
		.
	}
}


.
.
.

_HEAP_USERDATA_HEADER *UserData =
	RtlpAllocateUserBlock(lfh, UserBlockCacheIndex, BucketByteSize + 8);

DeletedSubSegment = ExInterlockedPopEntrySList(HeapLocalData);
if (DeletedSubSegment)
{
	// if there are any deleted subsegments, use them
	NewSubSegment = (_HEAP_SUBSEGMENT *)(DeletedSubSegment - 0x18);
}
```

```
else
{
	_HEAP_SUBSEGMENT *NewSubSegment =
		RtlpLowFragHeapAllocateFromZone(LFH, LocalDataIndex);

	//return failure use back-end
	if(!NewSubSegment)
		return 0;
}

//this function will setup the _HEAP_SUBEMENT structure
//and chunk out the data in 'UserData' to be of HeapBucket->SizeIndex chunks
RtlpSubSegmentInitialize(LFH,
	NewSubSegment,
	UserBlock,
	RtlpBucketBlockSizes[HeapBucket->SizeIndex],
	UserDataAllocSize,HeapBucket);

.
.
.
```

This flow of execution usually occurs when a **_HEAP_SUBSEGMENT** is not setup (i.e. the first allocation from the LFH for a certain **Bucket**) or if all the SubSegments have been used. This situation will give you a memory layout that looks like the following.

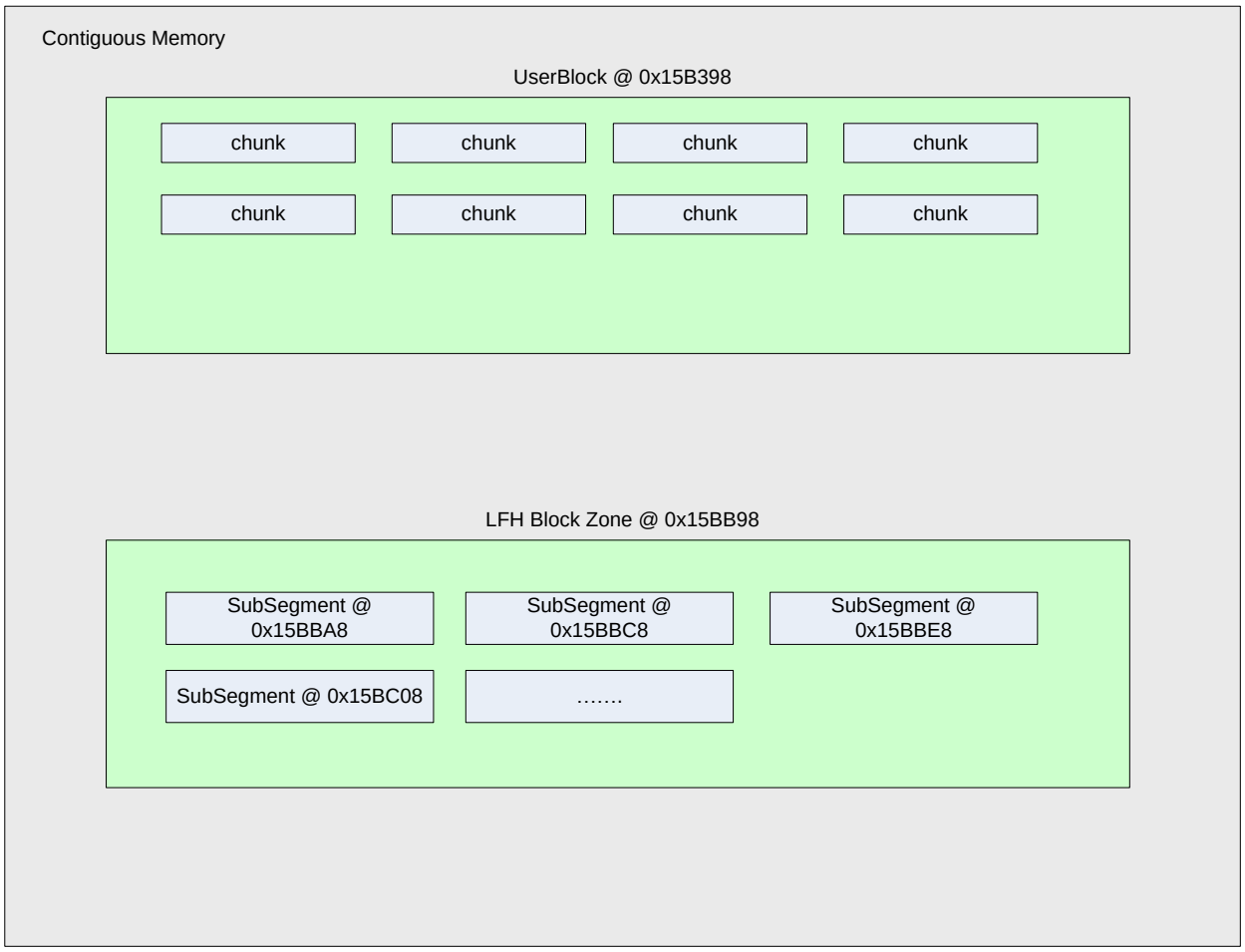

Diagram 23. UserBlock residing before SubSegment pointers in memory

As you can see, if you place the UserBlock chunk **before** the memory that is allocated for the **SubSegments**, then an overflow can overwrite pointers used by the **_HEAP_SUBSEGMENT** structures. Although Richard Johnson theorized that the **FreePointer** in the **_LFH_BLOCK_ZONE** could be compromised to write a **_HEAP_SUBSEGMENT** structure to a semi-arbitrary location (Johnson 2006), I have a different idea. You could overwrite the **SubSegment**, including the **UserBlock** pointer. Then proceed to make a subsequent allocation, giving you a **write-n** situation with a user supplied pointer.

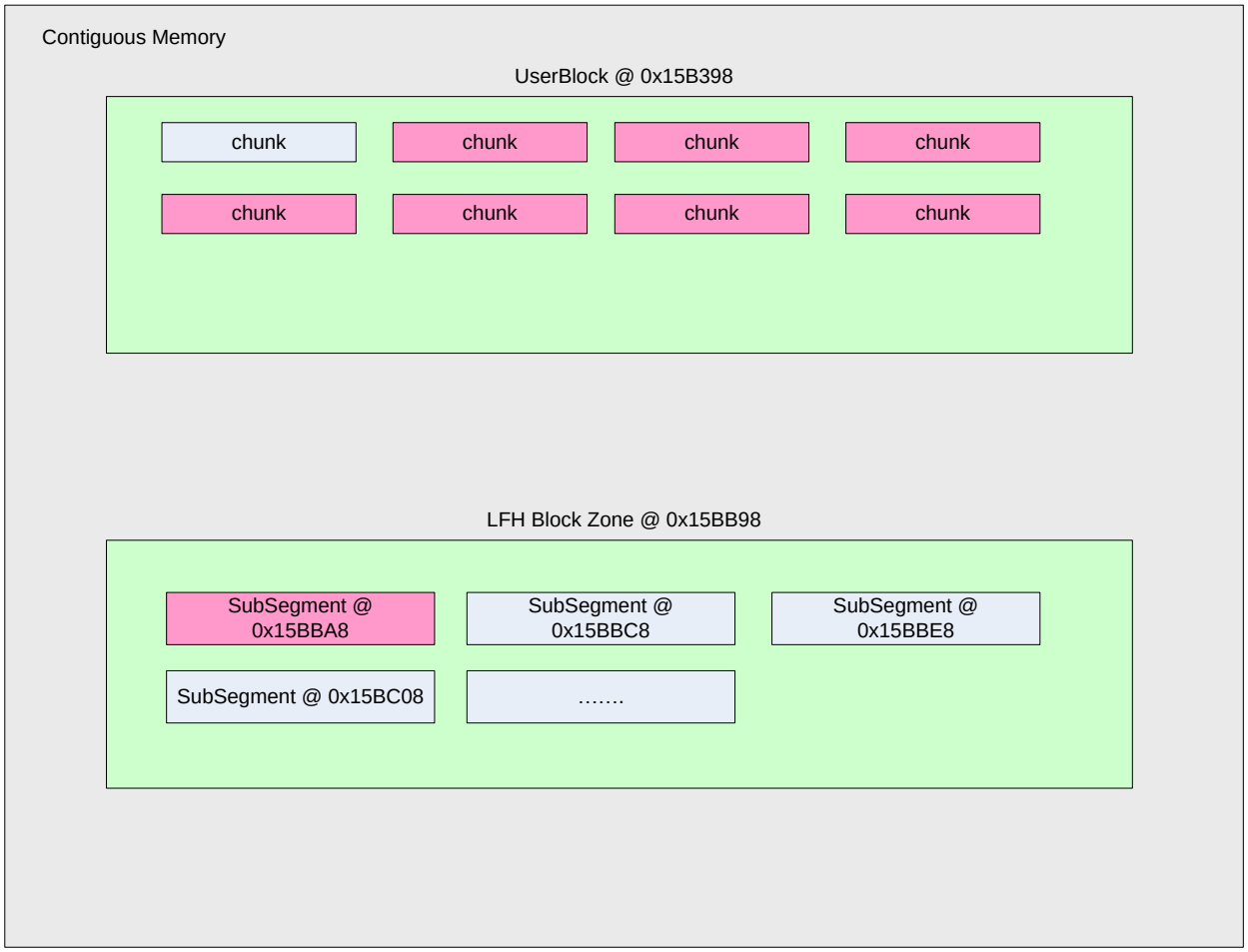

Diagram 24. Overwrite into _HEAP_SUBSEGMENT

Example

The following example shows how a LFH **Bin** can be enabled for a certain size, then a subsequent allocation can overwrite the **_HEAP_SUBSEGMENT**, resulting in tainted values used for later allocations. Note that the overwrite size of **0x200** isn't a specific value, but just used to show that all the **_LFH_BLOCK_ZONE** items can be overwritten.

Listing 53. SubSegment overwrite

```
//turn on the LFH
for(i = 0; i < 0x12; i++)
	allocb[i] = HeapAlloc(pHeap, 0x0, SIZE);

//first allocation for SIZE in LFH
alloc1 = HeapAlloc(pHeap, 0x0, SIZE);

//get closer in virtual memory
//to make overwrite easier
for(i = 0; i < 0x27; i++)
	alloc2 = HeapAlloc(pHeap, 0x0, SIZE);

//overwrite the UserBlocks pointer for this SubSegment
alloc3 = HeapAlloc(pHeap, 0x0, SIZE);
memset(alloc3, 0x41, 0x200);

//this allocation will use the tainted UserBlocks
alloc4 = HeapAlloc(pHeap, 0x0, SIZE);
```

Issues

There are two major hurdles holding this technique back. The first revolves around the ability, or lack thereof, to manipulate the heap in a certain way. As you saw from the example, a **UserBlock** chunk would need to be allocated before the memory used to store the **SubSegment** pointers (_LFH_BLOCK_ZONEs). While enabling a LFH Bin is trivial, ensuring that it resides in **contiguous** memory **before** the chunk used for SubSegment pointers will be much more difficult in a real world scenario. One just needs to take a look at Internet Explorer to see just how difficult a task this may be.

The second task is avoiding a check done in the allocation routine to guarantee SubSegment integrity. It will ensure that the _HEAP_LOCAL_SEGMENT_INFO structure for the requested size, matches that currently stored in the _HEAP_SUBSEGMENT.

Listing 54. SubSegment validation

```
_HEAP_LOCAL_SEGMENT_INFO *HeapLocalSegmentInfo =
       HeapLocalData->SegmentInfo[HeapBucket->SizeIndex];

_HEAP_SUBSEGMENT *SubSeg = HeapLocalSegmentInfo->ActiveSubsegment;

.
.

if(!Depth ||
       !UserBlocks ||
       SubSeg->LocalInfo != HeapLocalSegmentInfo)
{
       break;
}
```

While the **Depth** and **UserBlocks** conditions are easily spoofed, the check ensuring that **LocalInfo** structure is the same pointer as the one stored in the **Low Fragmentation heap** is a bit more complicated. Using a **leaked** chunk address is one possible solution to this problem, but the longer an application runs, the harder it will be to reliably predict an address. The easiest way would be to acquire the address of the **FrontEndHeap** pointer from the **HeapBase**. This would provide you with a pointer to a **_LFH_HEAP** structure then the address of the appropriate **_HEAP_LOCAL_SEGMENT_INFO** item could be inferred by the size of requested chunk.

Overall this technique provides a very easy write-n situation relying purely on heap meta-data and an address to overwrite. Unfortunately, at the time of this writing, a reliable technique has yet to be discovered. This does not mean it's impossible, it just means that I gave up because I am a huge failure.

Conclusion

Many aspects of Windows memory management have changed since the Windows XP code base. A vast majority of these changes occurred with the inception of Windows Vista and were carried over to Windows 7.

The data structures used are far more complex, providing better support for frequent memory requests via multiple threads, but also hold some familiarity due to similar usage to that of past heap implementations.

These new data structures are used in a different fashion than before. The concept of **Dedicated FreeLists** was quickly retired for more robust techniques. These new techniques provided means for the **back-end** manager to enable the **front-end** manager, which now only support the **Low Fragmentation heap**; as the **Lookaside List** has been sent out to pasture. A new structure referred to as the **UserBlock** now holds all chunks for a **HeapBucket** in one large continuous piece of memory after certain thresholds are met. This provides more efficient access to memory for frequent allocation and frees.

Although there were many security mechanisms added, such as encoded header, anti-bitmap flipping and safe linking, there were born some scenarios that could actually make heap manipulation more reliable. Now that memory of the same size can reside directly next to each other, overwrites can be more predictable and data seeding can be procured with simple control of allocations and frees.

With all the newly created data structures and algorithms came new complexity. This complexity, as Ben Hawkes has shown previously and I in this paper can be used to control the flow of execution. The new offsets and pointers can be used in simple overflow scenarios to alter the state of the heap and provide better means of reliable execution.

Finally, although meta-data overwrites can lead to a change in execution flow, they are not as useful as they once were. Heap exploitation has become increasingly complex and as time goes on, the ability to understand what you are allocating/freeing and where it resides has become far more important than *what* you are overwriting (not to mention subverting DEP and ASLR). It may seem unnecessary to discuss both the front and back-end managers in such depth, but to competently manipulate the heap you must understand exactly how it works.

- Chris Valasek 2010
    - @nudehaberdasher
    - cvalasek@gmail.com

Bibliography

Hawkes, Ben. 2008. Attacking the Vista Heap. Ruxcon 2008 / Blackhat USA 2008,
http://www.lateralsecurity.com/downloads/hawkes_ruxcon-nov-2008.pdf
http://www.blackhat.com/presentations/bh-usa-08/Hawkes/BH_US_08_Hawkes_Attacking_Vista_Heap.ppt

Immunity Inc. Immunity Debugger heap library source code. Immunity Inc.
http://debugger.immunityinc.com/update/Documentation/ref/Libs.libheap-pysrc.html

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

McDonald/Valasek. 2009. Practical Windows XP/2003 Heap Exploitation. Blackhat USA 2009,
http://www.blackhat.com/presentations/bh-usa-09/MCDONALD/BHUSA09-McDonald-WindowsHeap-PAPER.pdf

Marinescu, Adrian. 2006. Windows Vista Heap Management Enhancements. Blackhat USA 2006
http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Marinescu.pdf

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

Probert, David B. (PhD),
http://www.i.u-tokyo.ac.jp/edu/training/ss/lecture/new-documents/Lectures/16-UserModeHeap/UserModeHeapManager.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

Vreugdenhil, Peter. 2010. Windows7-InternetExplorer8. Pwn2Own 2010,
http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf

Waisman, Nico. 2010. (A)leatory (P)ersitent (T)hreat,
http://eticanicomana.blogspot.com/2010/03/aleatory-persitent-threat.html