Understanding the Low Fragmentation Heap

Chris Valasek, Researcher, X-Force Advanced R&D
cvalasek@gmail.com / @nudehaberdasher
Blackhat USA 2010
Introduction

“What. Are. You……?”
Introduction

• Much has changed since Windows XP
• Data structures have been added and altered
• Memory management is now a bit more complex
• New security measures are in place to prevent meta-data corruption
• Heap determinism is worth more than it used to be
• Meta-data corruption isn’t entirely dead
The Beer List

• Core data structures
  • _HEAP
  • _LFH_HEAP
  • _HEAP_LIST_LOOKUP

• Architecture
  • FreeLists

• Core Algorithms
  • Back-end allocation (RtlpAllocateHeap)
  • Front-end allocation (RtlpLowFragHeapAllocFromContext)
  • Back-end de-allocation (RtlpFreeHeap)
  • Front-end de-allocation (RtlpLowFragHeapFree)

• Tactics
  • Heap determinism
    • LFH specific heap manipulation
  • Exploitation
    • Ben Hawkes #1
    • FreeEntry Offset
    • Observations
Prerequisites

• All pseudo-code and data structures are taken from Windows 7 ntdll.dll version 6.1.7600.16385 (32-bit)
  • Yikes! I think there is a new one…

• Block/Blocks = 8-bytes

• Chunk = contiguous piece of memory measured in blocks or bytes

• HeapBase = _HEAP pointer

• LFH = Low Fragmentation Heap

• BlocksIndex = _HEAP_LIST_LOOKUP structure
  • 1st BlocksIndex manages chunks from 8 to 1024 bytes
    • ListHint[0x7F] = Chunks >= 0x7F blocks
  • 2nd BlocksIndex manages chunks from 1024 bytes to 16k bytes
    • ListHint[0x77F] = Chunks >= 0x7FF blocks

• Bucket/HeapBucket = _HEAP_BUCKET structure used as size/offset reference

• HeapBin/UserBlocks = Actually memory the LFH uses to fulfill requests
Core Data Structures

“Ntdll changed, surprisingly I didn’t quit”
### _HEAP (HeapBase)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x04c</td>
<td>EncodeFlagMask</td>
<td>Uint4B</td>
</tr>
<tr>
<td>+0x050</td>
<td>Encoding</td>
<td>_HEAP_ENTRY</td>
</tr>
<tr>
<td>+0x0b8</td>
<td>BlocksIndex</td>
<td>Ptr32 Void</td>
</tr>
<tr>
<td>+0x0c4</td>
<td>FreeLists</td>
<td>_LIST_ENTRY</td>
</tr>
<tr>
<td>+0x0d4</td>
<td>FrontEndHeap</td>
<td>Ptr32 Void</td>
</tr>
<tr>
<td>+0x0da</td>
<td>FrontEndHeapType</td>
<td>UChar</td>
</tr>
</tbody>
</table>

- **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 in the next slide.  
- **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)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>ExtendedLookup : Ptr32 _HEAP_LIST_LOOK培</td>
</tr>
<tr>
<td>0x004</td>
<td>ArraySize : Uint4B</td>
</tr>
<tr>
<td>0x010</td>
<td>OutOfRangeItems : Uint4B</td>
</tr>
<tr>
<td>0x014</td>
<td>BaseIndex : Uint4B</td>
</tr>
<tr>
<td>0x018</td>
<td>ListHead : Ptr32 _LIST_ENTRY</td>
</tr>
<tr>
<td>0x01c</td>
<td>ListsInUseUlong : Ptr32 Uint4B</td>
</tr>
<tr>
<td>0x020</td>
<td>ListHints : Ptr32 Ptr32 _LIST_ENTRY</td>
</tr>
</tbody>
</table>

- **ExtendedLookup** - A pointer to the next _HEAP_LIST_LOOK培 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 items in the FreeList[0]-like structure. Each _HEAP_LIST_LOOK培 tracks free chunks larger than ArraySize-1 in ListHint[ArraySize-BaselIndex-1].
- **BaseIndex** – Used to find the relative offset into the ListHints array, since each _HEAP_LIST_LOOK培 is designated for a certain size. For example, the BaseIndex for 1st BlockIndex would be 0x0 because it manages lists for chunks from 0x0 – 0x80, while the 2nd BlockIndex 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.
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.
_LFH_HEAP
(HeapBase->FrontEndHeap)

- **Heap** – A pointer to the parent heap of this LFH.
- **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** size.
- **LocalData** – This is a pointer to a large data structure which holds information about each **SubSegment**. See _HEAP_LOCAL_DATA for more information.

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x024 Heap</td>
<td>Ptr32 Void</td>
</tr>
<tr>
<td>+0x110 Buckets</td>
<td>[128] _HEAP_BUCKET</td>
</tr>
<tr>
<td>+0x310 LocalData</td>
<td>[1] _HEAP_LOCAL_DATA</td>
</tr>
</tbody>
</table>
## _HEAP_LOCAL_DATA

(HeapBase->FrontEndHeap->LocalData)

<table>
<thead>
<tr>
<th>Offsets</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x00c LowFragHeap : P:rt32 LFH HEAP</td>
<td></td>
</tr>
<tr>
<td>+0x018 SegmentInfo : [128] HEAP_LOCAL_SEGMENT_INFO</td>
<td></td>
</tr>
</tbody>
</table>

- **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. This structure type will be discussed in later sections.
**_HEAP_LOCAL_SEGMENT_INFO_**

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x000</td>
<td>Hint : Ptr32 _HEAP_SUBSEGMENT</td>
</tr>
<tr>
<td>+0x004</td>
<td>ActiveSubsegment : Ptr32 _HEAP_SUBSEGMENT</td>
</tr>
<tr>
<td>+0x058</td>
<td>LocalData : Ptr32 _HEAP_LOCAL_DATA</td>
</tr>
<tr>
<td>+0x060</td>
<td>BucketIndex : Uint2B</td>
</tr>
</tbody>
</table>

- **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)

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x000</td>
<td>LocalInfo – The _HEAP_LOCAL_SEGMENT_INFO structure associated with this structure.</td>
</tr>
<tr>
<td>+0x004</td>
<td>UserBlocks – A _HEAP_USERDATA_HEADER structure coupled with this SubSegment which holds a large chunk of memory split into n-number of chunks.</td>
</tr>
<tr>
<td>+0x008</td>
<td>AggregateExchg – An _INTERLOCK_SEQ structure used to keep track of the current Offset and Depth.</td>
</tr>
<tr>
<td>+0x016</td>
<td>SizeIndex – The _HEAP_BUCKET SizeIndex for this SubSegment.</td>
</tr>
<tr>
<td>+0x000 LocalInfo</td>
<td>Ptr32 _HEAP_LOCAL_SEGMENT_INFO</td>
</tr>
<tr>
<td>+0x004 UserBlocks</td>
<td>Ptr32 _HEAP_USERDATA_HEADER</td>
</tr>
<tr>
<td>+0x008 AggregateExchg</td>
<td>_INTERLOCK_SEQ</td>
</tr>
<tr>
<td>+0x016 SizeIndex</td>
<td>UChar</td>
</tr>
</tbody>
</table>
### _HEAP_USERDATA_HEADER

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x000</td>
<td>SubSegment</td>
<td>Ptr32 _HEAP_SUBSEGMENT</td>
</tr>
<tr>
<td>+0x004</td>
<td>Reserved</td>
<td>Ptr32 Void</td>
</tr>
<tr>
<td>+0x008</td>
<td>SizeIndex</td>
<td>Uint4B</td>
</tr>
<tr>
<td>+0x00c</td>
<td>Signature</td>
<td>Uint4</td>
</tr>
<tr>
<td>+0x010</td>
<td>User Writable Data</td>
<td>XXXX</td>
</tr>
</tbody>
</table>
**INTERLOCK_SEQ**

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

- **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) is 0x10. [0x2 * 0x8 == 0x10].

- **OffsetAndDepth** – Since both Depth and FreeEntryOffset are 2-bytes, are combined into this single 4-byte value.
_HEAP_ENTRY
(Chunk Header)

- **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.
Architecture

“The winner of the BIG award is…”
Once upon a time there were dedicated FreeLists which were terminated with pointers to sentinel nodes. Empty lists would contain a Flink and Blink pointing to itself.
Win7 FreeLists

- The concept of dedicated FreeLists have gone away. FreeList or ListHints will point to a location within Heap->FreeLists.

- They Terminate by pointing to &HeapBase->FreeLists. Empty lists will be NULL or contain information used by the LFH.

- Only Heap->FreeLists initialized to have Flink/Blink pointing to itself.

- Chunks >= ArraySize-1 will be tracked in BlocksIndex->ListHints[ArraySize-BaseIndex-1]

- If the LFH is enabled for a specific Bucket then the ListHint->Blink will contain the address of a _HEAP_BUCKET + 1. Otherwise, ListHint->Blink can contain a counter used to enable the LFH for that specific _HEAP_BUCKET.

- LFH can manage chunks from 8-16k bytes.

- FreeLists can track 16k+ byte chunks, but will not use the LFH.
Circular Organization of Chunk Headers (COCHs)
Algorithms: Allocation

"@hzon Do you remember any of the stuff we did last year?"
Alignment

• **RtlAllocateHeap: Part I**
  • It will round the size to be 8-byte aligned then find the appropriate BlocksIndex structure to service this request. Using the *FreeList[0]* like structure if it cannot service the request.

```c
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)
    {
        BlockSize = BlocksIndex->ArraySize - 1;
        break;
    }
    BlocksIndex = BlocksIndex->ExtendedLookup;
}
```

* The above searching now will be referred to as: **BlocksIndexSearch()**
• **RtlAllocateHeap: Part II**
  - The ListHints will now be queried look for an optimal entry point into the FreeLists. A check is then made to see if the LFH or the Back-end should be used.

```c
//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)
}
```
“Working in the library? Everyday day I’m Hustlin’!”
Allocation: Back-end

• **RtlpAllocateHeap: Part I**
  • The size is rounded if necessary and **RtlpPerformHeapMaintenance()** based on the **CompatibilityFlags**. This is what will actually enables the LFH.

```c
int RoundSize = aRoundSize;

//if the FreeList isn't NULL, the rounding has already been performed
if (!FreeList)
{
    RoundSize = Round(Size)RoundSize;
}

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))
{
    if (Heap->CompatibilityFlags & 0x60000000)
        RtlpPerformHeapMaintenance(Heap);
}
```
• **RtlpAllocateHeap: Part II**

  - If there is a FreeList and it doesn’t hold a _HEAP_BUCKET update the flags used to enable the LFH. If the LFH is already enabled assign the _HEAP_BUCKET to the blink.

```c
//if this freelist doesn't hold a _HEAP_BUCKET
if(FreeList != NULL && !(FreeList->Blink & 1))
{
    //increment the counter
    FreeList->Blink += 0x10002;

    //on the 0x10th time, try to get a _HEAP_BUCKET
    if((WORD)FreeList->Blink > 0x20 || FreeList->Blink > 0x10000000)
    {
        int FrontEndHeap;
        if(Heap->FrontEndHeapType == 0x2)
            FrontEndHeap = Heap->FrontEndHeap;
        else
            FrontEndHeap = NULL;

        //gets _HEAP_BUCKET in LFH->Bucket[BucketSize]
        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 heuristic
                if(Heap->FrontEndHeapType == NULL)
                    Heap->CompatibilityFlags |= 0x20000000;
            }
        }
        else
        {
            //save the _HEAP_BUCKET in the Blink
            FreeList->Blink = LFHContext + 1;
        }
    }
}
```
• **RtlpAllocateHeap: Part III**
  • If we’ve found a chunk in one of the **FreeLists** it can now be safely **unlinked** from the list and the **ListsInUseUlong** will be updated if necessary. The chunk will then be returned to the calling process.

```c
//attempt to use the Flink
if (FreeList != NULL && FreeList->Flink != NULL) {
    //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;
    DecodeAndValidateChecksum(ChunkToUseHeader);

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

    //update the bitmap if needed
    _HEAP_LIST_LOOKUP *BlocksIndex = Heap->BlocksIndex;
    if (BlocksIndex) {
        int FreeListOffset = GetFreeListOffset();

        //if there are more of the same size
        //don't update the bitmap
        if (!LastInList(BlocksIndex, FreeListOffset))
            BlocksIndex->ListHints[FreeListOffset] = Flink->Flink;
        else
            UpdateBitmap(BlocksIndex->ListsInUseUlong); //bitwise AND
    }

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

**RtlpAllocateHeap: Part IV**

- If the `ListHints` weren't successful, attempt to use the `Heap->FreeList / BlocksIndex->ListHead`. If successful it will return `ChunkToUse`, otherwise the heap will need to be extended via `RtlpExtendHeap()`.

```c
//BI->ListHead == Heap->FreeLists
_LIST_ENTRY *HeapFreeLists = &Heap->FreeLists;
_LIST_ENTRY *ChunkToUse;
_HEAP_LIST_LOOKUP *BI = Heap->BlocksIndex;

while(1)
{
    //bail if the list is empty
    if(BI == NULL || BI->ListHead == BI->ListHead)
    {
        ChunkToUse = BI->ListHead;
        break;
    }

    _HEAP_ENTRY *BlinkHeader = DecodeHeader(BI->ListHead->Blink - 8);

    //if the requested size is too big, extend the heap
    if(SizeInBlocks > BlinkHeader->Size)
    {
        ChunkToUse = BI->ListHead;
        break;
    }

    _HEAP_ENTRY *FlinkHeader = DecodeHeader(BI->ListHead->Flink - 8);

    //if the first chunk is sufficient use it
    //otherwise loop through the rest
    if(FlinkHeader->Size >= SizeInBlocks)
    {
        ChunkToUse = CurrListHead->Flink;
        break;
    }
    else
        FindChunk(BlocksIndex->ListHints, SizeInBlocks);

    //look at the next blocks index
    BI = BI->ExtendedLookup;
}
```
Algorithms: Allocation : Front-End

“Dr. Raid will take your pizza, fo sho“
• RtlpLowFragHeapAllocFromConext: Part I

• A _HEAP_SUBSEGMENT is acquired based off the _HEAP_BUCKET passed to the function. The Hint SubSegment is tried first, proceeding to the ActiveSubsegment pending a failure. If either of these succeed in the allocation request, the chunk is returned.

```c
// gets the data structures based off the SizeIndex (affinity left out)
_LFH_HEAP *LFH = GetLFHFromBucket(HeapBucket);
_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;

        // attempt different subsegment if this one is invalid
        _HEAP_USERDATA_HEADER *UserBlocks = SubSeg->UserBlocks;
        if(!Depth || !UserBlocks || SubSeg->LocalInfo != HeapLocalSegmentInfo)
            break;

        int ByteOffset = Offset * 8;
        LFHChunk = UserBlocks + ByteOffset;

        // the next offset is stored in the 1st 2-bytes of the user data
        short NextOffset = UserBlocks + ByteOffset + sizeof(_HEAP_ENTRY);

        if(AtomicUpdate(AggrExchg, NextOffset, Depth--)
            return LFHChunk;
        else
            SubSeg = SubSeg_Saved;
    }
}
```
**Allocation: Front-end**

- **RtlpLowFragHeapAllocFromConext: Part II**
  - If a SubSegment wasn’t able to fulfill the allocation, the LFH must create a new SubSegment along with an associated **UserBlock**. A **UserBlock** is the chunk of memory that holds individual chunks for a specific _HEAP_BUCKET_. A certain formula is used to calculate how much memory should actually be acquired via the back-end allocator.

```c
//assume no bucket affinity
int TotalBlocks = HeapLocalSegmentInfo->Counters->TotalBlocks;
int BucketBytesSize = RtlpBucketBlockSizes[HeapBucket->SizeIndex];
int StartIndex = 7;
int BlockMultiplier = 5;

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)) +
sizeof(_HEAP_USERDATA_HEADER) + sizeof(_HEAP_ENTRY);

if(TotalBlockSize > 0x78000) { TotalBlockSize = 0x78000; }

//calculate the cache 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;

//allocate ((1 << UserBlockCacheIndex) / BucketBytesSize) chunks on a cache miss
void *pUserData = RtlpAllocateUserBlock(lfh, UserBlockCacheIndex, BucketByteSize + 8);
_HPP_USERDATA_HEADER *UserData = (_HEAP_USERDATA_HEADER*)pUserData;
if(!pUserData)
    return 0;
```
• **RtlpLowFragHeapAllocFromConext: Part III**

  • Now that a **UserBlock** has been allocated, the LFH can acquire a **_HEAP_SUBSEGMENT**. If a SubSegment has been found it will then initialize that SubSegment along with the UserBlock; otherwise the **back-end** will have to be used to fulfill the allocation request.

```c
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);

_HEAP_SUBSEGMENT *NewSubSegment = NULL;
DeletedSubSegment = ExInterlockedPopEntrySList(HeapLocalData);
if (DeletedSubSegment)
    NewSubSegment = (_HEAP_SUBSEGMENT *)(DeletedSubSegment - 0x18);
else
{
    NewSubSegment = RtlpLowFragHeapAllocateFromZone(LFH, LocalDataIndex);
    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;

//now used for LFH allocation for a specific bucket size
NewSubSegment = AtomicSwap(&HeapLocalSegmentInfo->ActiveSegment, NewSubSegment);
```
**Allocation: Front-end**

- **RtlpLowFragHeapAllocFromConext**: Part IV [RtlpSubSegmentInitialize]
  - The **UserBlock** chunk is divided into **BucketBlockSize** chunks followed by the **SubSegment** initialization. Finally, this new SubSegment is ready to be assigned to the HeapLocalSegmentInfo->ActiveSubsegment.
### Allocation: Front-End : Example 1

#### UserBlock Chunks for 0x30 Bytes

![Diagram showing allocation of UserBlock Chunks for 0x30 Bytes](image)

- **NextOffset**
  - +0x02: NextOffset = 0x08
  - +0x08: NextOffset = 0x0E
  - +0x0E: NextOffset = 0x14
  - +0x14: NextOffset = 0x1A
  - +0x1A: NextOffset = 0x14
  - +0x20: NextOffset = 0x26
  - +0x26: NextOffset = 0x2C
  - +0x2C: NextOffset = 0x32
  - **Last Entry**
    - NextOffset = 0xFFFF

- **Next Virtual Address**
  - NextVirtualAddress = UserBlock + (NextOffset * 0x8)

- **AggrExchg.D/AggrExchg.D/AggrExchg.D/AggrExchg.D**
  - Depth = 0x2A
  - FreeEntryOffset = 0x02

---

© 2010 IBM Corporation
Allocation : Front-End : Example II
Allocation : Front-End : Example III

UserBlock Chunks for 0x30 Bytes
[0x6 Blocks]

+0x0E
NextOffset = 0x14

+0x1A
NextOffset = 0x14

+0x20
NextOffset = 0x26

+0x26
NextOffset = 0x2C

+0x14
NextOffset = 0x1A

+0x2C
NextOffset = 0x32

...                      ...                      ...

<table>
<thead>
<tr>
<th>Last Entry</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Last Entry
NextOffset = 0xFFFF

Next Virtual Address = UserBlock + (NextOffset * 0x8)
AggrExchg.Depth = 0x28
AggrExchg.FreeEntryOffset = 0x0E
Algorithms: Freeing

“How can you go wrong? (re: Dogs wearing sunglasses)”
Freeing

- **RtlFreeHeap**
  - RtlFreeHeap will determine if the chunk is free-able. If so it will decide if the LFH or the **back-end** should be responsible for releasing the chunk.

```c
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;
```
“Spencer Pratt explained this to me”
• **RtlpFreeHeap: Part I**
  • The back-end manager will first look for a ListHint index to use as an insertion point. It will then attempt to update the counter used in the LFH heuristic.

```c
//returns ArraySize-1 on miss & no ExtendedLookup
_HEAP_LIST_LOOKUP *BlocksIndex = Heap->BlocksIndex;
ChunkSize = SearchBlocksIndex(BlocksIndex, ChunkHeader->Size);

//attempt to locate a FreeList
_LIST_ENTRY *ListHint = NULL;

//if the chunk can fit on a blocksindex OR
//BlocksIndex[ArraySize-BaseIndex-1] can hold the chunk
if(FitsInBlocksIndex(BlocksIndex, ChunkSize))
{
  int FreeListIndex = ChunkSize - BlocksIndex->BaseIndex;

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

if(ListHint != NULL)
{
  //If no HEAP_BUCKET adjust counter
  if( !(BYTE)ListHint->Blink & 1)
  {
    if(ListHint->Blink >= 2)
    {
      ListHint->Blink = ListHint->Blink - 2;
    }
  }
}
```
Freeing : Back-End

- **RtlpFreeHeap: Part II**
  - The header values are set for the chunk being freed and it is **coalesced** if necessary. While the function may be called every iteration, it will only combine chunks that are adjacently **FREE**.

```c
//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 necessary
ChunkSize = ChunkHeader->Size;

//XXX Decomit or Give to Virtual Memory if exceeding the Thresholds

//mark the chunk as FREE
ChunkHeader->Flags = 0x0;
ChunkHeader->UnusedBytes = 0x0;
```
• **RtlpFreeHeap: Part III**
  
  • Now the heap manager will find which **BlocksIndex** and corresponding **ListHint** will manage this chunk. It will ensure that **ListHead** isn't empty and can insert this chunk before the largest chunk residing on the list.

```c
BlocksIndex = Heap->BlocksIndex;
_LIST_ENTRY *InsertList = Heap->FreeLists.Flink;

//attempt to find where to insert this item
//on the ListHead list for a particular BlocksIndex
if(BlocksIndex)
{
    int FreeListIndex = BlocksIndexSearch(BlocksIndex, ChunkSize)
    while(BlocksIndex != NULL)
    {
        //abort if the list is empty or too large to fit on this list
        _HEAP_ENTRY *ListHead = BlocksIndex-ListHead;
        if(ListHead == ListHead->Blink || ChunkSize > ListHead->Blink.Size)
        {
            InsertList = ListHead;
            break;
        }
        //start at the beginning of the ListHead pick the insertion point behind the lst
        //chunk larger than the ChunkToFree
        _LIST_ENTRY *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 point, break
        if(InsertList != Heap->FreeLists.Flink)
        break;
        BlocksIndex = BlocksIndex->ExtendedLookup;
    }
}
```
Freeing: Back-End

- **RtlpFreeHeap: Part IV**
  - Finally the chunk is **safely** linked into the list and `ListInUseUlong` is updated.

```c
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();
}

if (BlocksIndex)
{
    FreeListIndex = BlocksIndexSearch(BlocksIndex, ChunkSize);
    _LIST_ENTRY *FreeListToUse = BlocksIndex->ListHints[FreeListIndex];
    if (ChunkSize >= FreeListToUse->Size)
        BlocksIndex->ListHints[FreeListIndex] = ChunkToFree;

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

“Omar! Omar! Omar comin’!”
Freeing : Front-End

• **RtlpLowFragHeapFree: Part I**
  • The chunk *header* will be checked to see if a relocation is necessary. Then the
    chunk to be freed will be used to get the *SubSegment*. Flags indicating the chunk
    is now *FREE* are also set.

```c
//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 from 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;
```
Freeing : Front-End

- **RtlpLowFragHeapFree: Part II**
  - The Offset and Depth can now be updated. The *NewOffset* should point to the chunk that was recently freed and the depth will be incremented by 0x1.

```c
while(1)
{
    // update Offset and Depth
    int Depth = SubSegment->AggregateExchg.Depth;
    int Offset = SubSegment->AggregateExchg.FreeEntryOffset;

    _INTERLOCK_SEQ AggrExchg_New;
    AggrExchg_New.Sequence = UpdateSeq(SubSegment->AggregateExchg);

    if(!MaintanenceNeeded(SubSegment))
    {
        // set the FreeEntry Offset ChunkToFree
        *(WORD)(ChunkHeader + 8) = Offset;

        // Get the next free chunk, based off the offset from the UserBlocks
        // add 0x1 to the depth due to freeing
        int NewOffset = Offset - ((ChunkHeader - UserBlocks) / 8);
        AggrExchg_New.FreeEntryOffset = NewOffset;
        AggrExchg_New.Depth = Depth + 1;

        // this is where Hint is set :)
        SubSegment->LocalInfo->Hint = SubSegment;
    }
    else
    {
        PerformSubSegmentMaintenance(SubSegment);
        RtlpFreeUserBlock(LFH, SubSegment->UserBlocks);
        break;
    }

    // _InterlockedCompareExchange64
    if(AtomicSwap(&SubSegment->AggregateExchg, AggrExchg_New))
        break;
    else
        ChunkHeader = ChunkHeader_Saved;
}
```
Freeing : Front-End : Example I

UserBlock Chunks for 0x30 Bytes
[0x6 Blocks]

+0x14
NextOffset = 0x1A

+0x2C
NextOffset = 0x32

+0x2C

Last Entry
NextOffset = 0xFFFF

Next Virtual Address = UserBlock + (NextOffset * 0x8)
AggrExchg.Depth = 0x27
AggrExchg.FreeEntryOffset = 0x14
Freeing : Front-End : Example II

UserBlock Chunks for 0x30 Bytes
[0x6 Blocks]

<table>
<thead>
<tr>
<th>+0x02</th>
<th>+0x14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>NextOffset = 0x14</td>
<td>NextOffset = 0x1A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>+0x1A</th>
<th>+0x20</th>
<th>+0x26</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NextOffset = 0x14</td>
<td>NextOffset = 0x26</td>
<td>NextOffset = 0x2C</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Entry</td>
<td>NextOffset = 0x32</td>
<td></td>
</tr>
</tbody>
</table>

Next Virtual Address = UserBlock + (NextOffset * 0x8)
AggrExchg.Depth = 0x28
AggrExchg.FreeEntryOffset = 0x02
Freeing : Front-End : Example III

UserBlock Chunks for 0x30 Bytes
[0x6 Blocks]

+0x02  +0x08  +0x14
  NextOffset = 0x14  NextOffset = 0x02  NextOffset = 0x1A

+0x1A  +0x20  +0x26  +0x2C
  NextOffset = 0x14  NextOffset = 0x26  NextOffset = 0x2C  NextOffset = 0x32

...  ...  ...  Last Entry
  NextOffset = 0xFFFF

Next Virtual Address = UserBlock + (NextOffset * 0x8)
AggrExchg.Depth = 0x29
AggrExchg.FreeEntryOffset = 0x08
Security Mechanisms

“@shydemeanor I think I’m using too much code in the slides.”
Security Mechanisms : Heap Randomization

```c
int RandPad = (RtlpHeapGenerateRandomValue64() & 0x1F) << 0x10;

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

if(NtAllocateVirtualmemory(-1, &BaseAddress....))
    return 0;

heap = (_HEAP*)BaseAddress;
MaximumSize = TotalMaxSize;

// if we used a random pad, adjust the heap pointer and free the memory
if(RandPad != Zero)
{
    if(RtlpSecMemFreeVirtualMemory())
    {
        heap = (_HEAP*)RandPad + BaseAddress;
        MaximumSize = TotalSize - RandPad;
    }
}
```

• Information
  • 64k aligned
  • 5-bits of entropy
  • Used to avoid the same HeapBase on consecutive runs

• Thoughts
  • Not impossible to brute force
  • If TotalMaxSize wraps, there will be no RandPad
  • Hard to influence HeapCreate()
  • Unlikely due to NtAllocateVirtualmemory() failing
Security Mechanisms : Header Encoding/Decoding

- Information
  - Size, Flags, CheckSum encoded
  - Prevents predictable overwrites w/o information leak
  - Makes header overwrites much more difficult

- Thoughts
  - NULL out Heap->EncodeFlagMask
    - I believe a new heap would be in order.
  - Overwrite first 4 bytes of encoded header to break Header & Heap->EncodeFlagMask
    (Only useful for items in FreeLists)
  - Attack last 4 bytes of the header

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

```c
DecodeHeader(_HEAP_ENTRY *Header, _HEAP *Heap)
{
  if(Heap->EncodeFlagMask && (Header & Heap->EncodeFlagMask))
  {
    (DWORD)Header ^= Heap->Encoding;
  }
}```
Security Mechanisms: Death of Bitmap Flipping

- Information
  - XOR no longer used
  - OR for population
  - AND for exhaustion

- Thoughts
  - SOL
  - Not as important as before because FreeLists/ListHints aren't initialized to point to themselves.

```c
// 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;
}

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

// Heap Free
size = SIZE(chunk);
BitMask = 1 << (size & 0x1F);
BlocksIndex->ListInUseUlong[Size >> 5] |= BitMask;
```
Security Mechanisms : Safe Linking

```c
if (InsertList->Blink->Flink == InsertList)
{
    ChunkToFree->Flink = InsertList;
    ChunkToFree->Blink = InsertList->Blink;
    InsertList->Blink->Flink = ChunkToFree;
    InsertList->Blink = ChunkToFree
}
else
{
    RtlpLogHeapFailure();
}
if (BlocksIndex)
{
    FreeListIndex = BlocksIndexSearch(BlocksIndex, ChunkSize);
    _LIST_ENTRY *FreeListToUse = BlocksIndex->ListHints[FreeListIndex];

    //ChunkToFree.Flink/Blink are user controlled
    if (ChunkSize >= FreeListToUse.Size)
    {
        BlocksIndex->ListHints[FreeListIndex] = ChunkToFree;
    }
}
```

- **Information**
  - Prevents overwriting a FreeList->Blink, which when linking a chunk in can be overwritten to point to the chunk that was inserted before it
  - Brett Moore Attacking FreeList[0]

- **Thoughts**
  - Although it prevents Insertion attacks, if it doesn’t terminate, the chunk will be placed in one of the **ListHints**
  - The problem is the Flink/Blink are fully controlled due to no **Linking** process
  - You still have to deal with **Safe Unlinking**, but it’s a starting point.
Tactics

“You do not want to pray-after-free – Nico Waisman”
Tactics : Heap Determinism : Activating the LFH

• 0x12 (18) consecutive allocations will guarantee LFH enabled for SIZE
• 0x11 (17) if the _LFH_HEAP has been previously activated

```c
//Without the LFH activated
//0x10 => Heap->CompatibilityFlags |= 0x20000000;
//0x11 => RtlpPerformHeapMaintenance(Heap);
//0x11 => FreeList->Blink = LFHContext + 1;
for (i = 0; i < 0x12; i++)
    HeapAlloc(pHeap, 0x0, SIZE);
```
Tactics : Heap Determinism : Defragmentation

Gray = BUSY
Blue = FREE

A game of filling the holes
Easily done by making enough allocations to create a new **SubSegment** with associated **UserBlock**
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);

Result:
alloc2 => BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
alloc3 => CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

Post overflow..

alloc2 => AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACCCCCC
CCCCCCCC
alloc3 => AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACCCCCCCCCCCCC

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, SIZE)

• Overwrite into adjacent chunks (requires normalization)

• Can overwrite NULL terminator (Vreugdenhil 2010)

• Ability to use data in a recently freed chunk with proper heap manipulation
Tactics : Heap Determinism : Data Seeding

EnableLFH(SIZE);
 NormalizeLFH(SIZE);

for(i = 0; i < 0x4; i++)
 {
 allocb[i] = HeapAlloc(pHeap, 0x0, SIZE);
 memset(allocb[i], 0x41 + i, SIZE);
 }

printf("Freeing all chunks!\n");
for(i = 0; i < 0x4; i++)
 {
 HeapFree(pHeap, 0x0, allocb[i]);
 }

printf("Allocating again\n");
for(i = 0; i < 0x4; i++)
 {
 allocb[i] = HeapAlloc(pHeap, 0x0, SIZE);
 }

Result:
Allocation 0x00 for 0x28 bytes =&gt; 41414141 41414141 41414141
Allocation 0x01 for 0x28 bytes =&gt; 42424242 42424242 42424242
Allocation 0x02 for 0x28 bytes =&gt; 43434343 43434343 43434343
Allocation 0x03 for 0x28 bytes =&gt; 44444444 44444444 44444444

Freeing all chunks!
Allocating again
Allocation 0x00 for 0x28 bytes =&gt; 0E004444 44444444 44444444
Allocation 0x01 for 0x28 bytes =&gt; 08004343 43434343 43434343
Allocation 0x02 for 0x28 bytes =&gt; 02004242 42424242 42424242
Allocation 0x03 for 0x28 bytes =&gt; 62004141 41414141 41414141

• Saved FreeEntryOffset resides in 1st 2 bytes
• Influence the LSB of vtable
• Good for use-after-free
• See Nico Wasiman’s 2010 BH Presentation / Paper
• NICO Rules!
Tactics : Exploitation

“For the Buristicati, By the Buristicati”
RtlpLowFragHeapFree() will adjust the _HEAP_ENTRY if certain flags are set.

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

<table>
<thead>
<tr>
<th>Flags</th>
<th>Checksum</th>
<th>Prev Size</th>
<th>Seg Offset</th>
<th>UnusedBytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Size</td>
<td>Flags</td>
<td>Checksum</td>
<td>Prev Size</td>
<td>Seg Offset</td>
</tr>
</tbody>
</table>

If you can overflow into a chunk that will be freed, the `SegmentOffset` can be used to point to another valid _HEAP_ENTRY.

This could lead to controlling data that was previously allocated (Think C++ objects)
**Prerequisites**
- Ability to allocate SIZE
- Place legitimate a chunk before a chunk to be overflowed
- Overflow at least 8-bytes
- Ability to free overwritten chunk

**Methodology**
1. Enable LFH
2. Normalize LFH
3. Alloc1
4. Alloc2
5. Overwrite Alloc2's header to point to an object of interest
6. Free Alloc2
7. Alloc3 (will point to the object of interest)
8. Write data
Tactics : Exploitation : FreeEntryOffset Overwrite: Part I

All code in RtlpLowFragHeapAllocFromContext() is wrapped in
try/catch{} . All exceptions will return 0, letting the back-end handle the
allocation.

```c
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;
}
```

As we saw, the FreeEntryOffset is stored in the 1st 2 bytes of user-
writeable data within each chunk in a UserBlock.

This will be used to get the address of the next free chunk used for
allocation. What if we overflow this chunk?
Assume a **full** UserBlock for 0x30 bytes (0x6 blocks). Our first allocation will update the **FreeEntryOffset** to **0x0008**. (Stored in the _INTERLOCK_SEQ_.FreeEntryOffset)

Memory Pages

<table>
<thead>
<tr>
<th>+0x02</th>
<th>+0x08</th>
<th>+0x0E</th>
<th>+0x14</th>
</tr>
</thead>
<tbody>
<tr>
<td>NextOffset = 0x0008</td>
<td>NextOffset = 0x000E</td>
<td>NextOffset = 0x0014</td>
<td>NextOffset = 0x001A</td>
</tr>
</tbody>
</table>

... ... ... NextOffset = 0xFFFF
(Last Entry)

FreeEntryOffset = 0x0002
If an overflow of at least 0x9 bytes (0xA preferable) is made. The saved FreeEntryOffset of the adjacent chunk can be overwritten. This gives the attacker a range of 0xFFFF * 0x8 (Offsets are stored in blocks and converted to byte offsets.)
An allocation for the overwritten block must be made next to store the tainted offset in the _INTERLOCK_SEQ. In this example, we will have a 0x1501 * 0x8 jump to the next ‘free chunk’.
Tactics : Exploitation : FreeEntryOffset Overwrite: Part V

Since it's possible to get SubSegments adjacent to each other in memory, you can write into other forwardly adjacent memory pages (Control over allocations is required). This gives you the ability to overwrite data that is in a different _HEAP_SUBSEGMENT than the one which you are overflowing.

<table>
<thead>
<tr>
<th>Memory Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserBlock @ 0x5157800 for Size 0x30</td>
</tr>
<tr>
<td>+0x0E NextOffset = 0x0014 +0x14 NextOffset = 0x001A</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>FreeEntryOffset = 0x1501</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserBlock @ 0x5162000 for Size 0x40</td>
</tr>
<tr>
<td>+0x02 NextOffset = 0x000A +0x0A NextOffset = 0x0012 +0x12 NextOffset = 0x001A +0x1A NextOffset = 0x0022</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>FreeEntryOffset = 0x0002</td>
</tr>
</tbody>
</table>
Prerequisites

- Enabled the LFH
- Normalize the heap
- Control allocations for SIZE
- 0x9 – 0xA byte overflow into an adjacent chunk
- Adjacent chunk must be FREE
- Object to overwrite within the range (0xFFFF * 0x8 = max)

Methodology

1. Enable LFH
2. Normalize LFH
3. Alloc1
4. Overwrite into free chunk from Alloc1
5. Alloc2 (contains overwritten header)
6. Alloc3 (Uses overwritten FreeEntryOffset)
7. Write data to Alloc3 (which will be object of your choosing w/in 0xFFFF * 0x8)

NextChunk = UserBlock + Depth_IntoUserBlock + (FreeEntryOffset * 8)
NextChunk = 0x5157800 + 0x0E + (0x1501 * 8)
NextChunk = 0x5162016
Tactics : Exploitation : Observation

“Strawberry Pudding? Psst, this is a five course meal.”
If the SubSegment can not be used, it will create a new **UserBlock** and assign it to a new **SubSegment**.

**RtlpLowFragHeapAllocateFromZone** will create space for new SubSegments if they have all been exhausted.
This provides a memory layout where the UserBlock data resides before the _LFH_BLOCK_ZONE structures (which hold pointers for SubSegment initialization).
Tactics : Exploitation : SubSegment Overwrite: Part III

An overflow past the end of the UserBlock will result in the overwriting of SubSegment information. The item of most concern is the pointer to the UserBlocks structure inside the SubSegment. If this value can be overwritten, then a subsequent allocation will result in a write-n to a user-supplied address.
Tactics : Exploitation : SubSegment Overwrite: Part IV

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

**Issues**

1. The **UserBlock** that can be overflowed MUST reside before the space allocated for the _HEAP_SUBSEGMENT. This is not trivial, due to most applications not having a deterministic **BlockZone->Limit**. You won’t know how many pointers are left.

2. **SubSeg->LocalInfo != HeapLocalSegmentInfo**. The address of the _HEAP_LOCAL_SEGMENT_INFO structure for a specific Bucket is required. The easiest way to determine this value would be a leak of the _LFH_HEAP pointer. (There are probably other ways as well)

3. A guard page could mitigate the effects of an overflow into an adjacent SubSegment.
Conclusion

“I know that most of the audience will be fast asleep by now.”
Conclusion

• Data structures have become far more complex

• Dedicated FreeLists / Lookaside List are dead
  • Replaced with new FreeList structure and LFH

• Many security mechanisms added since Win XP SP2

• Meta data corruption now leveraged to overwrite application data

• Heap normalization more important than ever

• Much more work to be done…
What’s next?

• Developing reliable exploits specifically for Win7
• Abusing Un-encoded header information
• Look at Virtual / Debug allocation/free routines
• Caching mechanisms
• Continuing to come up with heap manipulation techniques
• Figuring out information leaks (heap addresses)
• HeapCON?
Thanks to all the BISTICATI for their help!

- Jon Larimer
- Ryan Smith
- Nico Waisman
- Ben Hawkes
- Matt Miller
- Alex Sotirov
- Dino Dai Zovi
- Mark Dowd
- John McDonald
- @jmpesp
- Matthieu Suiche
Demo

“Fin.”