Windows 8 Heap Internals

Chris Valasek
Sr. Security Research Scientist – Coverity
cvalasek@gmail.com
@nudehaberdasher

Tarjei Mandt
Sr. Vulnerability Researcher – Azimuth
kernelpool@gmail.com
@kernelpool

Contents

Introduction

Windows 8 developer preview was released in September 2011. While many focused on the Metro UI of the operating system, we decided to investigate the memory manager. Even though generic heap exploitation has been dead for quite some time, intricate knowledge of both the application and underlying operating system's memory manager have permitted reliable heap exploitation occur under certain circumstances. This paper focuses on the transition of heap exploitation mitigations from Windows 7 to Windows 8 (Release Preview) from both a user-land and kernel-land perspective. We will be examining the inner workings of the Windows memory manager for allocations, de-allocations and all additional heap-related security features implemented in Windows 8. Also, additional tips and tricks will be covered providing the readers the proper knowledge to achieve the highest possible levels of heap determinism.

Overview

This paper is broken into two major sections, each having several subsections. The first major section of the paper covers the **User Land Heap Manager**, which is default mechanism for applications that implementing dynamic memory (i.e. heap memory). The first subsection will give an overview of changes in the **Data Structures** used by the Windows 8 heap manager when tracking memory used by applications, followed by a brief update regarding an update to the overall heap manager **Architecture**. The second subsection will cover key **Algorithms** that direct the manager on how to allocate and free memory. The third subsection will unveil information about **Security Mitigations** that are new to the Windows 8 operating system, providing better overall protection for dynamically allocated memory. The fourth and final subsection will divulge information regarding **Exploitation Tactics**. Although few, still are valid against the Windows 8 Release Preview. Lastly, a conclusion will be formed about the overall state of the User Land Heap Manager.

The second major section will detail the inner workings of the Windows 8 **Kernel Pool Allocator**. In the first subsection, we briefly introduce the **Kernel Pool**, its lists and structures. The second subsection highlights the new major **Security Improvements** featured in the Windows 8 kernel pool, such as the non-executable non-paged pool and the kernel pool cookie. In the third subsection, we look at how **Prior Attacks** applicable to Windows 7 are mitigated in Windows 8 with the help of these improvements as well as by introducing more stringent security checks. In subsection four, we discuss some alternative approaches for **Attacking** the Windows 8 kernel pool, while still focusing on pool header attacks. Finally, in subsection five, we offer a conclusion of the overall state of the Kernel Pool.

Prior Works

Although the content within this document is completely original, it is based on a foundation of prior knowledge. The follow list contains some works that are recommended reading before fully divulging into this paper:

- While some of the algorithms and data structures have changed for the **Heap Manager**, the underlying foundation is very similar to the Windows 7 Heap Manager (Valasek 2010)
- Again, the vast majority of changes to the **Kernel Pool** were derived from the Windows 7 Kernel Pool which should be understood before digesting the follow material (Mandt 2011)
- Lionel d'Hauenens (http://www.laboskopia.com) Symbol Type Viewer was an invaluable tool when analyzing the data structures used by the Windows 8 heap manager. Without it many hours might have been wasted looking for the proper structures.

Prerequisites

User Land

All of the pseudo-code and data structures were acquired via the 32-bit version of Windows 8 Release Preview from **ntdll.dll (6.2.8400.0)**, which is the most recent version of the binary. Obviously, the code and data is limited to a 32-bit architecture but may have relevance to the 64-bit architecture as well.

If you have any questions, comments, or feel that any of the information regarding the **Heap Manager** is incorrect, please feel free to contact Chris at cvalasek@gmail.com.

Kernel Land

All of the pseudo-code and data structures were acquired via the 64-bit version of Windows 8 Release Preview from **ntoskrnl.exe** (**6.2.8400.0**). However, both 32- and 64-bit versions have been studied in order to identify differences in how mitigations have been implemented. This is mentioned explicitly where applicable.

If you have any questions, comments, or feel that any of the information regarding the **Kernel Pool** is incorrect, please feel free to contact Tarjei at kernelpool@gmail.com.

Terminology

Just like previous papers, this section is included to avoid any ambiguity with regards to terms used to describe objects and function of the Windows 8 heap. While the terms may not be universally agreed upon, they will be consistently used throughout this paper.

The term **block** or **blocks** will refer to 8-bytes or 16-bytes of contiguous memory for 32-bit and 64-bit architectures, respectively. 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.

A **chunk header** or **heap chunk header** is synonymous with a _HEAP_ENTRY structure and can be interchangeably used with the term **header**.

A _HEAP_LIST_LOOKUP structure is used to keep track of free chunk based on their size and will be called a **BlocksIndex** or a **ListLookup**.

A **FreeList** is a doubly linked list that is a member of the **HeapBase** structure that has a head pointing to the smallest **chunk** in the list and gets progressively larger until pointing back to itself to denote list termination. **ListHints**, on the other hand, point into the **FreeLists** at specific locations as an optimization when searching for chunks of a certain size.

The term **UserBlocks** or **UserBlock container** is used to describe the collection of individual **chunks** that are preceded by a _HEAP_USERDATA_HEADER. These individual chunks are the memory that the Low Fragmentation Heap (**LFH**) returns to the calling function. The chunks in the UserBlocks are grouped by size, or put into **HeapBuckets** or **Buckets**.

Lastly, the term **Bitmap** will be used to describe a contiguous piece of memory where each bit represents a state, such as **free** or **busy**.

User Land Heap Manager

This section examines the inner workings of Windows 8 Heap Manager by detailing the data structures, algorithms, and security mechanisms that are integral to its operation. The content is not meant to be completely exhaustive, but only to provide insight into the most important concepts applicable to Windows 8 Release Preview.

Data Structures

The following data structures come from Windows 8 Release Preview via Windbg with an **ntdll.dll** having a version of **6.2.8400.0**. These structures are used to keep track and manage free and allocated memory when an application calls functions such as free(), malloc(), and realloc().

_HEAP
(HeapBase)

A heap structure is created for each process (default process heap) and can also be created ad hoc via the **HeapCreate()** API. It serves as the main infrastructure for all items related to dynamic memory, containing other structures, pointers, and data used by the Heap Manager to properly allocate and de-allocate memory.

For a full listing please issue the *dt _HEAP* command in Windbg.

```
0:030> dt _HEAP
ntdll!_HEAP
+0x000 Entry          : _HEAP_ENTRY
…
+0x018 Heap           : Ptr32 _HEAP
…
+0x04c EncodeFlagMask   : Uint4B
+0x050 Encoding         : _HEAP_ENTRY
+0x058 Interceptor      : Uint4B
…
+0x0b4 BlocksIndex      : Ptr32 Void
…
+0x0c0 FreeLists        : _LIST_ENTRY
+0x0c8 LockVariable     : Ptr32 _HEAP_LOCK
+0x0cc CommitRoutine    : Ptr32     long
+0x0d0 FrontEndHeap     : Ptr32 Void
…
+0x0d8 FrontEndHeapUsageData : Ptr32 Uint2B
+0x0dc FrontEndHeapMaximumIndex : Uint2B
+0x0de FrontEndHeapStatusBitmap : [257] UChar
+0x1e0 Counters        : _HEAP_COUNTERS
+0x23c TuningParameters : _HEAP_TUNING_PARAMETERS
```

- **FrontEndHeap** – A pointer to a structure that is the FrontEnd Heap. In Windows 8 case, the Low Fragmentation Heap (LFH) is the only option available.

- **FrontEndHeapUsageData** – Is an array of **128 16-bit integers** that represent a counter or **HeapBucket** index. The counter denotes how many allocations of a certain size have been seen, being **incremented** on allocation and **decremented** on de-allocation. The HeapBucket **index** is used by the FrontEnd Heap to determine which **_HEAP_BUCKET** will service a request. It is updated by the BackEnd manager during allocations and frees to heuristically enable the LFH for a certain size. Windows 7 previously stored these values in the ListHint[Size]->**Blink** variable within the **BlocksIndex**.
- **FrontEndHeapStatusBitmap** – A bitmap used as an optimization when determining if a memory request should be serviced by the BackEnd or FrontEnd heap. If the bit is set then the LFH (FrontEnd) will service the request, otherwise the BackEnd (linked list based heap) will be responsible for the allocation. It is updated by the BackEnd manager during allocations and frees to heuristically enable the LFH for specific sizes.

_LFH_HEAP
(Heap->FrontEndHeap)

The _LFH_HEAP structure hasn't changed much since the Windows 7 days, only now there are **separate** arrays for **regular** InfoArrays and **Affinitized** InfoArrays. This means, unlike Windows 7, which used the LocalData member to access the proper _HEAP_LOCAL_SEGMENT_INFO structure based on Processor **Affinity**, Windows 8 has separate variables.

```
0:030> dt _LFH_HEAP
ntdll!_LFH_HEAP
   +0x000 Lock           : _RTL_SRWLOCK
   +0x004 SubSegmentZones  : _LIST_ENTRY
   +0x00c Heap           : Ptr32 Void
   +0x010 NextSegmentInfoArrayAddress : Ptr32 Void
   +0x014 FirstUncommittedAddress : Ptr32 Void
   +0x018 ReservedAddressLimit : Ptr32 Void
   +0x01c SegmentCreate    : Uint4B
   +0x020 SegmentDelete    : Uint4B
   +0x024 MinimumCacheDepth : Uint4B
   +0x028 CacheShiftThreshold : Uint4B
   +0x02c SizeInCache      : Uint4B
   +0x030 RunInfo        : _HEAP_BUCKET_RUN_INFO
   +0x038 UserBlockCache   : [12] _USER_MEMORY_CACHE_ENTRY
   +0x1b8 Buckets        : [129] _HEAP_BUCKET
   +0x3bc SegmentInfoArrays : [129] Ptr32 _HEAP_LOCAL_SEGMENT_INFO
   +0x5c0 AffinitizedInfoArrays : [129] Ptr32 _HEAP_LOCAL_SEGMENT_INFO
   +0x7c8 LocalData       : [1] _HEAP_LOCAL_DATA
```

- **SegmentInfoArrays** – This array is used when there is no affinity associated with a specific HeapBucket (i.e. size).
- **AffinitizedInfoArrays** – This array is used when a specific processor or core is deemed responsible for certain allocations. See SMP (SMP) for more information.

_HEAP_LOCAL_DATA
(Heap->FrontEndHeap->LocalData)

The only thing to notice is that due to how Affinitized and LocalInfo arrays are handled by the LFH, the _HEAP_LOCAL_DATA structure no longer needs to have a _HEAP_LOCAL_SEGMENT_INFO **array** member.

```
0:001> 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
   +0x014 DeleteRateThreshold : Uint4B
```

_HEAP_LOCAL_SEGMENT_INFO
(Heap->LFH->SegmentInfoArrays[] / AffinitizedInfoArrays[])

The structure has changed a bit since Windows 7. It no longer contains the **Hint** _HEAP_SUBSEGMENT structure as it is no longer used for allocations. Other than the removal of the Hint, the only changes are the order of the members.

```
0:001> dt _HEAP_LOCAL_SEGMENT_INFO
ntdll!_HEAP_LOCAL_SEGMENT_INFO
   +0x000 LocalData       : Ptr32 _HEAP_LOCAL_DATA
   +0x004 ActiveSubsegment : Ptr32 _HEAP_SUBSEGMENT
   +0x008 CachedItems     : [16] Ptr32 _HEAP_SUBSEGMENT
   +0x048 SListHeader     : _SLIST_HEADER
   +0x050 Counters        : _HEAP_BUCKET_COUNTERS
   +0x058 LastOpSequence   : Uint4B
   +0x05c BucketIndex     : Uint2B
   +0x05e LastUsed        : Uint2B
   +0x060 NoThrashCount    : Uint2B
```

- **ActiveSubsegment** – As you can see there is now only one _HEAP_SUBSEGMENT in the LocalInfo structure. The **Hint** Subsegment is no longer used.

_HEAP_SUBSEGMENT
(Heap->LFH->InfoArrays[]->ActiveSubsegment)

The _HEAP_SUBSEGMENT structure has only minor changes that add a singly linked list used to track chunks that could not be freed at the designated time.

```
0:001> dt _HEAP_SUBSEGMENT
ntdll!_HEAP_SUBSEGMENT
   +0x000 LocalInfo        : Ptr32 _HEAP_LOCAL_SEGMENT_INFO
   +0x004 UserBlocks       : Ptr32 _HEAP_USERDATA_HEADER
   +0x008 DelayFreeList    : _SLIST_HEADER
   +0x010 AggregateExchg   : _INTERLOCK_SEQ
   +0x014 BlockSize        : Uint2B
   +0x016 Flags            : Uint2B
   +0x018 BlockCount       : Uint2B
   +0x01a SizeIndex        : UChar
   +0x01b AffinityIndex    : UChar
   +0x014 Alignment        : [2] Uint4B
   +0x01c SFreeListEntry   : _SINGLE_LIST_ENTRY
   +0x020 Lock             : Uint4B
```

- **DelayFreeList** – This singly linked list is used to store the addresses of chunks that could not be freed at their desired time. The next time **RtlFreeHeap** is called, the de-allocator will attempt to traverse the list and free the chunks if possible.

_HEAP_USERDATA_HEADER
(Heap->LFH->InfoArrays[]->ActiveSubsegment->UserBlocks)

This data structure has gone through the biggest transformation since Windows 7. Changes were made so the heap manager would not have to blindly rely in information that could have been corrupted. It also takes into account adding guard pages for extra protection when allocating a **UserBlock** container.

```
0:001> dt _HEAP_USERDATA_HEADER
ntdll!_HEAP_USERDATA_HEADER
   +0x000 SFreeListEntry   : _SINGLE_LIST_ENTRY
   +0x000 SubSegment       : Ptr32 _HEAP_SUBSEGMENT
   +0x004 Reserved         : Ptr32 Void
   +0x008 SizeIndexAndPadding : Uint4B
   +0x008 SizeIndex        : UChar
   +0x009 GuardPagePresent : UChar
   +0x00a PaddingBytes     : Uint2B
   +0x00c Signature        : Uint4B
   +0x010 FirstAllocationOffset : Uint2B
   +0x012 BlockStride      : Uint2B
   +0x014 BusyBitmap       : _RTL_BITMAP
   +0x01c BitmapData       : [1] Uint4B
```

- **GuardPagePresent** – If this flag is set then the initial allocation for the UserBlocks will contain a guard page at the end. This prevents sequential overflows from accessing adjacent memory.
- **FirstAllocationOffset** – This SHORT is very similar to the **implied** initial value of 0x2 on Windows 7. Now the value is set explicitly to the first allocable chunk in the UserBlocks
- **BlockStride** – A value to denote the size of each chunk (which are all the same size) contained by the UserBlocks. Previously, this value was derived from the **FreeEntryOffset**.
- **BusyBitmap** – A contiguous piece of memory that contains a bitmap denoting which chunks in a UserBlock container are FREE or BUSY. In Windows 7, this was accomplished via the FreeEntryOffset and the _INTERLOCK_SEQ.Hint variables.

_RTL_BITMAP
(Heap->LFH->InfoArrays[]->ActiveSubsegment->UserBlocks->Bitmap)

This small data structure is used to determine which chunks (and their associated **indexes** in the UserBlocks) are FREE or BUSY for a parent UserBlock container.

```
0:001> dt _RTL_BITMAP
ntdll!_RTL_BITMAP
   +0x000 SizeOfBitMap     : Uint4B
   +0x004 Buffer           : Ptr32 Uint4B
```

- **SizeOfBitMap** – The size, in bytes, of the bitmap
- **Buffer** – The actual bitmap used in verification operations

_HEAP_ENTRY

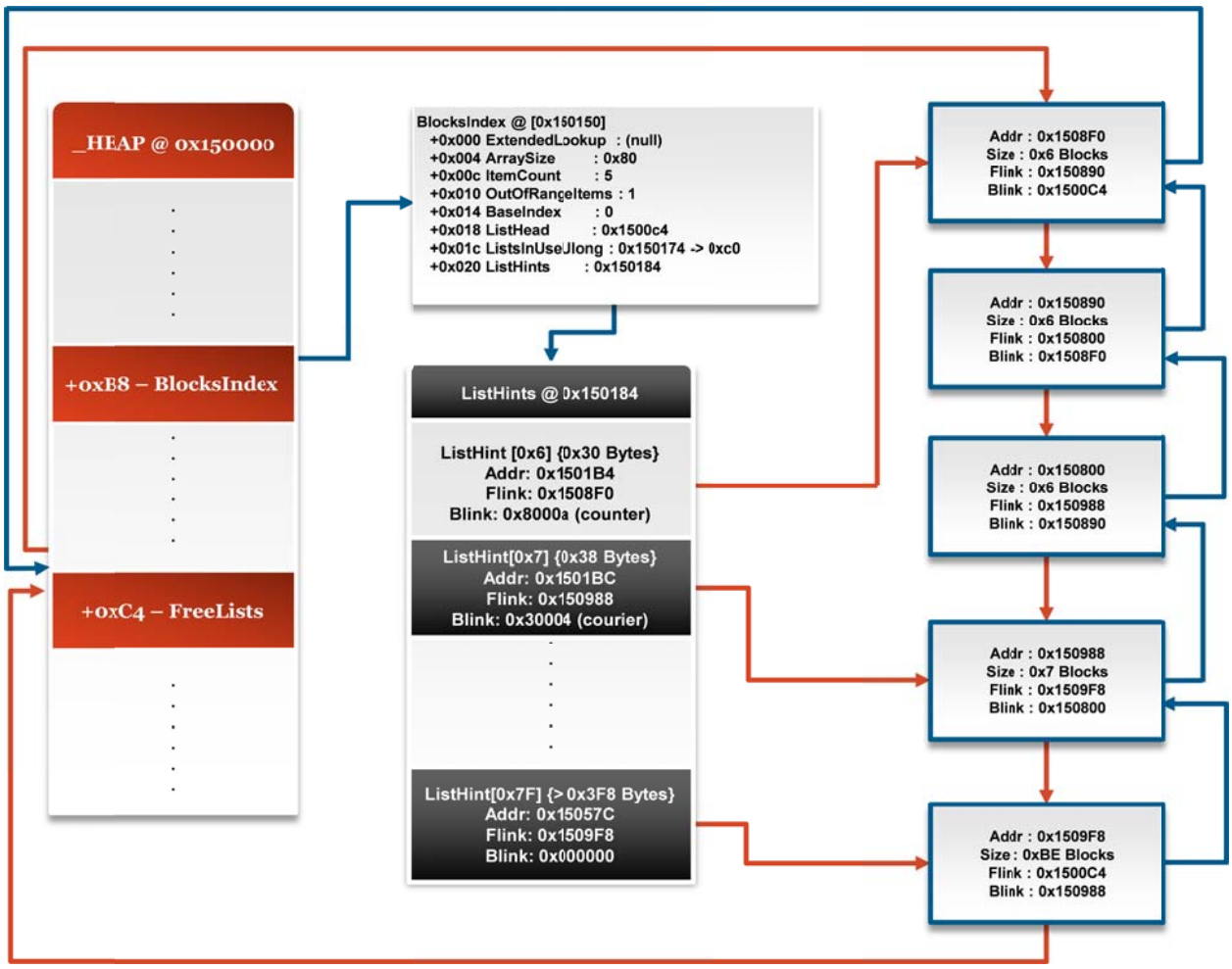
Although the structure of the _HEAP_ENTRY (aka chunk header or **header**) has remained the same, some repurposing has been done to chunks residing in the LFH.

```
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
   +0x000 Code1          : Uint4B
   +0x004 Code2          : Uint2B
   +0x006 Code3          : UChar
   +0x007 Code4          : UChar
   +0x004 Code234        : Uint4B
   +0x000 AgregateCode   : Uint8B
```

- **PreviousSize** – If the chunk resides in the LFH the PreviousSize member will contain the **index** into the **bitmap** used by the **UserBlocks**, instead of the size of the previous chunk (which makes sense, as all chunks within a UserBlock container are the same size).

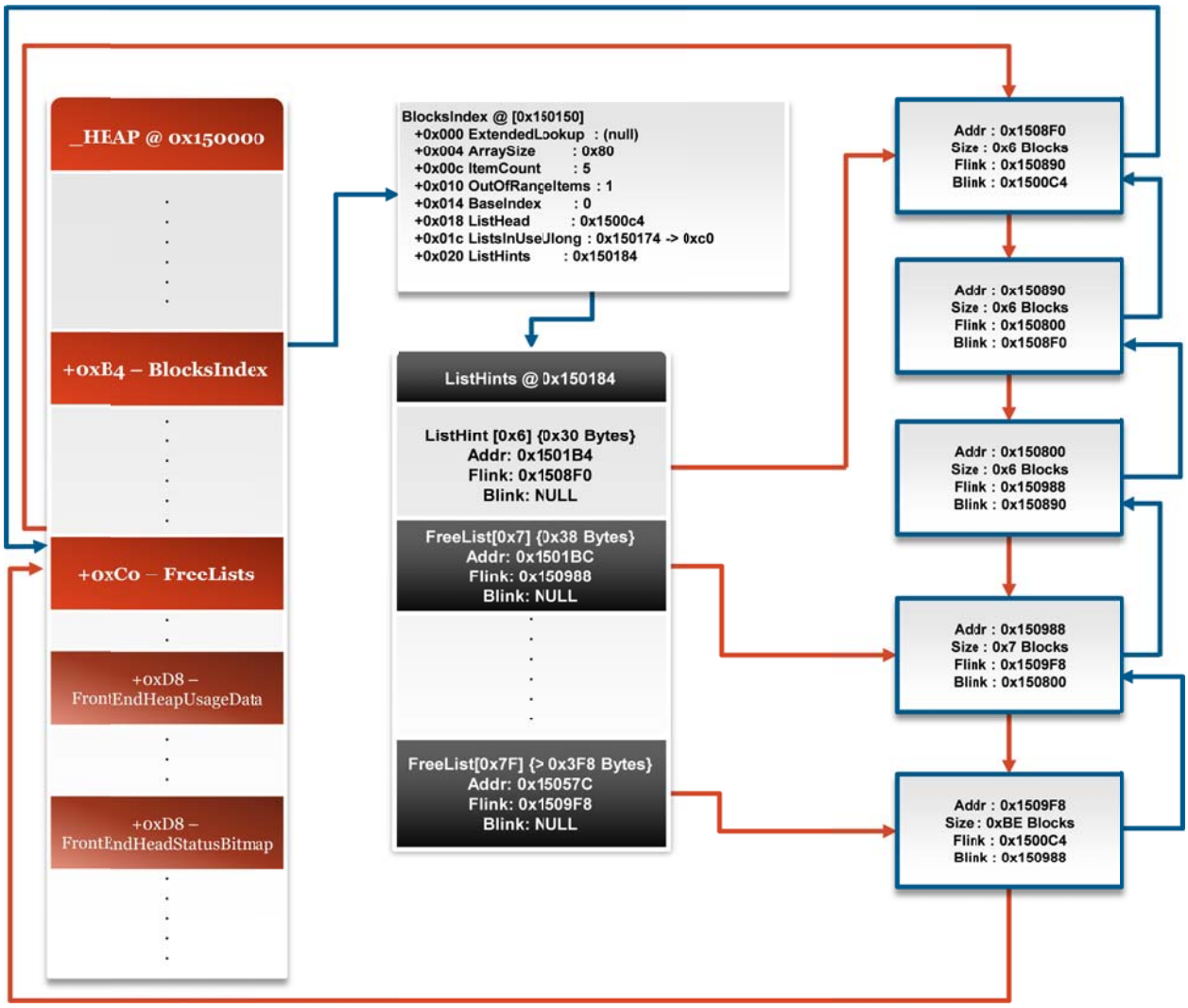
Architecture

The architecture of the Windows 8 **ListHint** and **FreeList** structures are nearly identical except for the removal of the dual purpose ListHint. Windows 7 uses the ListHint[Size]->**Blink** for two purposes. If the LFH was not enabled, then the value contained a counter. If the LFH was enabled, then the value would be the address of the **_HEAP_BUCKET** structure (plus 1). The example below shows how the doubly linked list structure served a dual purpose for counting the number of allocations and pointing to a **FreeList** entry if one existed.



I assume this was done so that a new data structure didn't need to be created and a **_LIST_ENTRY** could be purposed for multiple tasks. Unfortunately, as Ben Hawkes pointed out (Hawkes 2008), the _HEAP_BUCKET structure could be overwritten and used to **subvert** the **FrontEnd** allocator. Therefore, the Windows 8 heap team decided to add **dedicated** variables for the **allocation count** and store the _HEAP_BUCKET **index** (instead of the actual _HEAP_BUCKET), tying them all together with a **bitmap** optimization for decision making when choosing to use the FrontEnd or BackEnd heap.

You can see in the figure below that the **ListHints** no longer contain a **counter** in the **Blink**, which is set to NULL. Also, there have been members added to the **HeapBase** that track which chunk sizes should be serviced by the LFH. Not only does this speed up allocation decisions, but it also works as a mitigation for Ben Hawkes' _HEAP_BUCKET overwrite attack (Hawkes 2008).



Algorithms -- Allocation

This section will go over the allocation algorithms used by the Windows 8 Heap Manager. The first subsection will cover the Intermediate algorithm which determines whether the LFH or the BackHeap heap shall service a request. The second subsection details the BackEnd heap, as it uses heuristics to enable the LFH for chunks based on size. Lastly, the LFH allocation routine will be described in detail. While the intermediate and BackEnd algorithms are very similar to the Windows 7 versions, the FrontEnd (LFH) allocator has changed significantly.

Note: Much of the code has been left out to simplify the learning process. Please contact Chris if more in-depth information is desired

Intermediate

Before the LFH or the BackEnd heap can be used, certain fields need to be examined to determine the best course of action. The function that serves this purpose is `RtlAllocateHeap`. It has a function signature of:

```
void *RtlAllocateHeap(_HEAP *Heap, DWORD Flags, size_t Size)
```

The first thing the function does is some validation on the amount of memory being requested. If the requested size is too large (**above 2GB** on 32-bit), the call will fail. If the requested size is too small, the minimum amount of memory is requested. Then the size is rounded up to the nearest **8-byte** value, as all chunks are tracked in **Block** size, not bytes.

```
void *chunk;

//if the size is above 2GB, it won't be serviced
if(Size > 0x7FFFFFFF)
	return ERROR_TOO_BIG;

//ensure that at least 1-byte will be allocated
//and subsequently rounded (result ==> 8 byte alloc)
if(Size == 0)
	Size = 1;

//ensure that there will be at least 8 bytes for user data
//and 8 bytes for the _HEAP_ENTRY header
int RoundSize = (Size + 15) & 0xFFFFFF8;

//blocks are contiguous 8-byte chunks
int BlockSize = RoundSize / 8;
```

Next, if the size of the chunk is outside the realm that can be serviced by the LFH (less than 16k), the BackEnd heap attempts to acquire memory on behalf of the calling function. The process of acquiring memory from the BackEnd starts with locating an appropriately sized BlocksIndex structure and identifying the desired ListHint. If the BlocksIndex fails to have a sufficiently sized ListHint, then the **OutOfRange** ListHint is used (ListHints[BlocksIndex->**ArraySize-1**]). Finally, the BackEnd allocator can get the correct function parameters and attempt to allocate memory, returning a chunk on success and an error on failure.

```
//The maximum allocation unit for the LFH 0x4000 bytes
if(Size > 0x4000)
{
	_HEAP_LIST_LOOKUP *BlocksIndex;
	while(BlockSize >= BlocksIndex->ArraySize)
	{
		if(!BlocksIndex->ExtendedLookup)
		{
			BlockSize = BlocksIndex->ArraySize - 1;
			break;
		}
		BlocksIndex = BlocksIndex->ExtendedLookup;
	}

	//gets the ListHint index based on the size requested
	int Index = GetRealIndex(BlocksIndex, BlockSize);

	_LIST_ENTRY *hint = Heap->ListHints[Index];

	int DummyRet;
	chunk = RtlpAllocateHeap(Heap, Flags | 2, Size, RoundSize, Hint, &DummyRet);

	if(!chunk)
		return ERROR_NO_MEMORY;

	return chunk;
}
```

If the size requested can potentially be accommodated by the LFH, then `RtlAllocateHeap` will attempt to see if the FrontEnd is **enabled** for the **size** being requested (remember this is the **rounded size**, not the size requested by the calling function). If the bitmap indicated the LFH is servicing request for the particular size, then LFH will pursue allocation. If the LFH fails or the bitmap says that the LFH is not enabled, the routine described **above** will execute in an attempt to use the BackEnd heap.

```
else
{
	//check the status bitmap to see if the LFH has been enabled
	int BitmapIndex = 1 << (RoundSize / 8) & 7;

	if(BitmapIndex & Heap->FrontEndStatusBitmap[RoundSize >> 6])
	{
		//Get the BucketIndex (as opposed to passing a _HEAP_BUCKET)
		_LFH_HEAP LFH = Heap->FrontEndHeap;
		unsigned short BucketIndex = FrontEndHeapUsageData[BlockSize];

		chunk = RtlpLowFragHeapAllocFromContext(LFH,
			BucketIndex, Size, Flags | Heap->GlobalFlags);
	}

	if(!chunk)
		TryBackEnd();
	else
		return chunk;
}
```

Note: In Windows 7 the ListHint->Blink would have been checked to see if the LFH was activated for the size requested. The newly created bitmap and usage data array have taken over those responsibilities, doubling as an exploitation mitigation.

BackEnd

The BackEnd allocator is almost identical to the BackEnd of Windows 7 with the only exception being the newly created **bitmap** and **status arrays** are used for tracking LFH activation instead of the ListHint **Blink**. There have also been security features added to virtual allocations that prevent predictable addressing. The function responsible for BackEnd allocations is `RtlpAllocateHeap` and has a function signature of:

```
void *__fastcall RtlpAllocateHeap(_HEAP *Heap, int Flags, int Size, unsigned int
RoundedSize, _LIST_ENTRY *ListHint, int *RetCode)
```

The first step taken by the BackEnd is complementary to the Intermediate function to ensure that a minimum and maximum size is set. The maximum number of bytes to be allocated must be under 2GB and the minimum will be 16-bytes, 8-bytes for the **header** and 8-bytes for use. Additionally, it will check to see if the heap is set to use the LFH (it **can** be set to **NEVER** use the LFH) and update some heuristics.

```
void *Chunk = NULL;
void *VirtBase;
bool NormalAlloc = true;

//covert the 8-byte aligned amount of bytes
// to 'blocks' assuring space for at least 8-bytes user and 8-byte header
int BlockSize = RoundedSize / 8;
if(BlocksSize < 2)
{
	BlockSize = 2;
	RoundedSize += 8;
}

//32-bit arch will only allocate less than 2GB
if(Size >= 0x7FFFFFFF)
	return 0;

//if we have serialization enabled (i.e. use LFH) then go through some heuristics
if(!(Flags & HEAP_NO_SERIALIZE))
{
	//This will activate the LFH if a FrontEnd allocation is enabled
	if (Heap->CompatibilityFlags & 0x30000000)
		RtlpPerformHeapMaintenance(vHeap);
}
```

Next a test is made to determine if the **size** being requested is greater than the VirtualMemoryThreshold (set to 0x7F000 in `RtlCreateHeap`). If the allocation is too large, the **FreeLists** will be bypassed and virtual allocation will take place. New features added will augment the allocation with some security measures to ensure that **predictable** virtual memory **addresses** will **not** be likely. Windows 8 will generate a random number and use it as the **start** of the virtual memory **header**, which as a byproduct, will randomize the amount of total memory requested.

```
//Virtual memory threshold is set to 0x7F000 in RtlCreateHeap()
if(BlockSize > Heap->VirtualMemoryThreshold)
{
	//Adjust the size for a _HEAP_VIRTUAL_ALLOC_ENTRY
	RoundedSize += 24;

	int Rand = (RtlpHeapGenerateRandomValue32() & 15) << 12;

	//Total size needed for the allocation
	size_t RegionSize = RoundedSize + 0x1000 + Rand;

	int Protect = PAGE_READWRITE;
	if(Flags & 0x40000)
		Protect = PAGE_EXECUTE_READWRITE;

	//if we can't reserve the memory, then we're going to abort
	if(NtAllocateVirtualMemory(-1, &VirtBase, 0, &RegionSize,
		MEM_RESERVE, Protect) < 0)
		return NULL;

	//Return at an random offset into the virtual memory
	_HEAP_VIRTUAL_ALLOC_ENTRY *Virt = VirtBase + Rand;

	//If we can't actually commit the memory, abort
	if(NtAllocateVirtualMemory(-1, &Virt, 0, &RoundedSize,
		MEM_COMMIT, Protect) < 0)
	{
		RtlpSecMemFreeVirtualMemory(-1, &VirtBase, &Rand, MEM_RESET);
		++heap->Counters.CommitFailures;

		return NULL;
	}

	//Assign the size, falgs, etc
	SetInfo(Virt);

	//add the virtually allocated chunk to the list ensuring
	//safe linking in at the end of the list
	if(!SafeLinkIn(Virt))
		RtlpLogHeapFailure();

	Chunk = Virt + sizeof(_HEAP_VIRTUAL_ALLOC_ENTRY);
	return Chunk;
}
```

If a virtual chunk isn't required, the BackEnd will attempt to update heuristics used to let the Heap Manager know that the LFH can be used, if enabled and necessary.

```
//attempt to determine if the LFH should be enabled for the size requested
if(BlockSize >= Heap->FrontEndHeapMaximumIndex)
{
	//if a size that could be serviced by the LFH is requested
	//attempt to set flags indicating bucket activation is possible
	if(Size < 0x4000 && (Heap->FrontEndHeapType == 2 && !Heap->FrontEndHeap))
		Heap->CompatibilityFlags |= 0x20000000;
}
```

Immediate after **compatibility** checks, the desired **size** is examined to determine if it falls within the bounds of the **LFH**. If so, then the allocation counters are updated and attempt to see if _HEAP_BUCKET is active. Should a heap **bucket** be active, the `FrontEndHeapStatusBitmap` will be updated to tell the Heap Manager that the next allocation should come from the LFH, **not** the BackEnd. Otherwise, **increment** the allocation counters to indicate another allocation has been made, which will count towards heap bucket **activation**.

```
else if(Size < 0x4000)
{
	//Heap->FrontEndHeapStatusBitmap has 256 possible entries
	int BitmapIndex = BlockSize / 8;
	int BitPos = BlockSize & 7;

	//if the lfh isn't enabled for the size we're attempting to allocate
	//determine if we should enable it for the next go-around
	if(!((1 << BitPos) & Heap->FrontEndHeapStatusBitmap[BitmapIndex]))
	{
		//increment the counter used to determine when to use the LFH
		unsigned short Count = Heap->FrontEndHeapUsageData[BlockSize] + 0x21;
		Heap->FrontEndHeapUsageData[BlockSize] = Count;

		//if there were 16 consecutive allocation or many allocations consider LFH
		if((Count & 0x1F) > 0x10 || Count > 0xFF00)
		{
			//if the LFH has been initialized and activated, use it
			_LFH_HEAP *LFH = NULL;
			if(Heap->FrontEndHeapType == 2)
				LFH = heap->FrontEndHeap;

			//if the LFH is activated, it will return a valid index
			short BucketIndex =  RtlpGetLFHContext(LFH, Size);
			if(BucketIndex != -1)
			{
				//store the heap bucket index
				Heap->FrontEndHeapUsageData[BlockSize] = BucketIndex;

				//update the bitmap accordingly
				Heap->FrontEndHeapStatusBitmap[BitmapIndex] |= 1 << BitPos;
			}
			else if(Count > 0x10)
			{
				//if we haven't been using the LFH, we will next time around
				if(!LFH)
					Heap->CompatibilityFlags |= 0x20000000;
			}
		}
	}
}
```

With all the FrontEnd **activation** heuristics out of the way, the BackEnd can now start searching for a chunk to fulfill the allocation request. The first source examined is the **ListHint** passed to `RtlpAllocateHeap`, which is the obvious choice owing to its acquisition in `RtlAllocateHeap`. If a ListHint wasn't provided or doesn't contain any free chunks, meaning there was **not** an exact **match** for the amount of bytes desired, the **FreeLists** will be traversed looking for a sufficiently sized chunk (which is any chunk **greater than or equal** to the request size). On the off chance that there are no chunks of a suitable size, the heap must be **extended** via `RtlpExtendHeap`. The combination of a failure to find a chunk in the FreeLists and the inability to extend the heap will result in returning with error.

```
//attempt to use the ListHints to optimally find a suitable chunk
_HEAP_ENTRY *HintHeader = NULL;
_LIST_ENTRY *FreeListEntry = NULL;
if(ListHint && ListHint->Flink)
	HintHeader = ListHint - 8;
else
{
	FreeListEntry = RtlpFindEntry(Heap, BlockSize);

	if(&Heap->FreeLists == FreeListEntry)
	{
		//if the freelists are empty, you will have to extend the heap
		_HEAP_ENTRY *ExtendedChunk = RtlpExtendHeap(Heap, aRoundedSize);

		if(ExtendedChunk)
			HintHeader = ExtendedChunk;
		else
			return NULL;
	}
	else
	{
		//try to use the chunk from the freelist
		HintHeader = FreeListEntry - 8;
		if(Heap->EncodeFlagMask)
			DecodeValidateHeader(HintHeader, Heap);

		int HintSize = HintHeader->Size;

		//if the chunk isn't big enough, extend the heap
		if(HintSize < BlockSize)
		{
			EncodeHeader(HintHeader, Heap);
			_HEAP_ENTRY *ExtendedChunk = RtlpExtendHeap(Heap, RoundedSize);

			if(ExtendedChunk)
				HintHeader = ExtendedChunk;
			else
				return NULL;
		}
	}
}
```

Before returning the chunk to the user the BackEnd ensures that the item retrieved from the FreeLists is **not tainted**, returning **error** if doubly-linked list tainting has occurred. This functionality has been around since Windows XP SP2, and subsequently **killed** off **generic** heap overflow exploitation.

```
ListHint = HintHeader + 8;
_LIST_ENTRY *Flink = ListHint->Flink;
_LIST_ENTRY *Blink = ListHint->Blink;

//safe unlinking or bust
if(Blink->Flink != Flink->Blink || Blink->Flink != ListHint)
{
	RtlpLogHeapFailure(12, Heap, ListHint, Flink->Blink, Blink->Flink, 0);
	return ERROR;
}

unsigned int HintSize = HintHeader->Size;
_HEAP_LIST_LOOKUP *BlocksIndex = Heap->BlocksIndex;
if(BlocksIndex)
{
	//this will traverse the BlocksIndex looking for
	//an appropriate index, returning ArraySize - 1
	//for a chunk that doesn't have a ListHint (or is too big)
	HintSize = SearchBlocksIndex(BlocksIndex);
}

//updates the ListHint linked lists and Bitmap used by the BlocksIndex
RtlpHeapRemoveListEntry(Heap, BlocksIndex, RtlpHeapFreeListCompare,
	ListHint, HintSize, HintHeader->Size);

//unlink the entry from the linked list
//safety check above, so this is OK
Flink->Blink = Blink;
Blink->Flink = Flink;
```

Note: Header encoding and decoding has been left out to shorten the code. Just remember that decoding will need to take place before header attributes are accessed and encoded directly thereafter.

Lastly, the **header** values can be updated and the memory will be zeroed out if required. I've purposefully **left out** the block splitting process. Please see `RtlpCreateSplitBlock` for more information on chunk splitting (which will occur if the **UnusedBytes** are greater than 1).

```
if( !(HintHeader->Flags & 8) || RtlpCommitBlock(Heap, HintHeader))
{
	//Depending on the flags and the unused bytes the header
	//will set the UnusedBytes and potentially alter the 'next'
	//chunk directly after the one acquired from the FreeLists
	//which migh result in a call to RtlpCreateSplitBlock()
	int UnusedBytes = HintHeader->Size - RoundedSize;
	bool OK = UpdateHeaders(HintHeader);

	if(OK)
	{
		//We've updated all we need, MEM_ZERO the chunk
		//if needed and return to the calling function
		Chunk = HintHeader + 8;
		if(Flags & 8)
			memset(Chunk, 0, HintHeader->Size - 8);

		return Chunk;
	}
	else
		return ERROR;

}
else
{
	RtlpDeCommitFreeBlock(Heap, HintHeader, HintHeader->Size, 1);
	return ERROR;
}
```

Front End

The LFH is the sole FrontEnd allocator for Windows 8 and is capable of tracking **chunks** that have a size below **0x4000** bytes (**16k**). Like Windows 7, the Windows 8 LFH uses **UserBlocks**, which are pre-allocated containers for smaller chunks, to service requests. The similarities end there, as you will see searching for FREE chunks, allocating UserBlocks and many other tasks have changed. The function responsible for FrontEnd allocation is `RtlpLowFragHeapAllocFromContext` and has a function signature of:

```
void *RtlpLowFragHeapAllocFromContext(_LFH_HEAP *LFH, unsigned short BucketIndex, int
Size, char Flags)
```

The first thing you may notice is that a _HEAP_BUCKET pointer is **no longer** passed as a function argument, instead passing the index into the **HeapBucket** array within the LFH. It was discussed previously that this prevents an attack devised by Ben Hawkes (Hawkes 2008).

The first step is determining if we're dealing with a size that has been labeled as having **affinity** and if so, **initialize** all the variables that will be used in the forthcoming operations.

```
_HEAP_BUCKET *HeapBucket = LFH->Buckets[BucketIndex];
_HEAP_ENTRY *Header = NULL;

int VirtAffinity = NtCurrentTeb()->HeapVirtualAffinity - 1;
int AffinityIndex = VirtAffinity;
if(HeapBucket->UseAffinity)
{
	if(VirtAffinity < 0)
			AffinityIndex = RtlpAllocateAffinityIndex();

	//Initializes all global variables used for Affinity based allocations
	AffinitySetup();
}
```

After the affinity variables have been initialized the FrontEnd decides which array it is going to use to acquire a **_HEAP_LOCAL_SEGMENT_INFO** structure, which is ordered by size (and affinity if present). Then it will acquire the **ActiveSubsegment** which will be used for the upcoming allocation.

```
int SizeIndex = HeapBucket->SizeIndex;
_HEAP_LOCAL_SEGMENT_INFO *LocalSegInfo;

if(AffinityIndex)
	LocalSegInfo = LFH->AffinitizedInfoArrays[SizeIndex][AffinityIndex - 1];
else
	LocalSegInfo = LFH->SegmentInfoArrays[SizeIndex];

_HEAP_SUBSEGMENT *ActiveSubseg = LocalSegInfo->ActiveSubsegment;
```

Note: You'll notice there is no longer a check for a **Hint** Subsegment as that functionality has been removed.

Next, a check is made to ensure that the `ActiveSubsegment` is non-null, checking the **cache** for previously used _HEAP_SUBSEGMENT if the `ActiveSubsegment` is **NULL**. Hopefully the **Subsegment** will be valid and the **Depth**, **Hint**, and **UserBlocks** will be gathered. The **Depth** represents the **amount** of **chunks** left for a given Subsegment/UserBlock combo. The **Hint** was once an offset to the first **free chunk** within the UserBlocks, but no longer serves that purpose.

If the UserBlocks is not setup yet or there are not any chunks left in the UserBlock container, the cache will be examined and a new UserBlocks will be created. Think of this as checking that a swimming pool exists and full of water before diving in head first.

```
//This is actually done in a loop but left out for formatting reasons
//The LFH will do its best to attempt to service the allocation before giving up
if(!ActiveSubseg)
	goto check_cache;

_INTERLOCK_SEQ *AggrExchg = ActiveSubseg->AggregateExchg;

//ensure the values are acquired atomically
int Depth, Hint;
AtomicAcquireDepthHint(AggrExchg, &Depth, &Hint);

//at this point we should have acquired a sufficient subsegment and can
//now use it for an actual allocation, we also want to make sure that
//the UserBlocks has chunks left along w/ a matching subsegment info structures
_HEAP_USERDATA_HEADER *UserBlocks = ActiveSubseg->UserBlocks;

//if the UserBlocks haven't been allocated or the
//_HEAP_LOCAL_SEGMENT_INFO structures don't match
//attempt to acquire a Subsegment from the cache
if(!UserBlocks || ActiveSubseg->LocalInfo != LocalSegInfo)
	goto check_cache;
```

This is where the similarities to Windows 7 subside and Windows 8 shows its pretty colors. Instead of blindly using the **Hint** as an **index** into the **UserBlocks**, subsequently updating itself with another un-vetted value (**FreeEntryOffset**), it uses a random offset into the UserBlocks as a starting point.

The first step in the new process is to acquire a **random** value that was pre-populated into a global array. By using a random value instead of the next available free chunk, the allocator can avoid **determinism**, putting quite a hindrance on use-after-free and sequential overflow vulnerabilities.

```
//Instead of using the FreeEntryOffset to determine the index
//of the allocation, use a random byte to start the search
short LFHDataSlot = NtCurrentTeb()->LowFragHeapDataSlot;
BYTE Rand = RtlpLowFragHeapRandomData[LFHDataSlot];
NtCurrentTeb()->LowFragHeapDataSlot++;
```

Next the **bitmap**, which is used to determine which chunks are **free** and which chunks are **busy** in a UserBlock container, is acquired and a starting offset is chosen for identifying free chunks.

```
//we need to know the size of the bitmap we're searching
unsigned int BitmapSize = UserBlocks->BusyBitmap->SizeOfBitmap;

//Starting offset into the bitmap to search for a free chunk
unsigned int StartOffset = Rand;

void *Bitmap = UserBlocks->BusyBitmap->Buffer;

if(BitmapSize < 0x20)
	StartOffset = (Rand * BitmapSize) / 0x80;
else
	StartOffset = SafeSearchLargeBitmap(UserBlocks->BusyBitmap->Buffer);
```

Note: The StartOffset might not actually be FREE. It is only the starting point for searching for a FREE chunk.

The bitmap is then **rotated** to the right ensuring that, although we're starting at a **random** location, all possible positions will be examined. Directly thereafter, the bitmap is inverted, due to the way the assembly instruction `bsf` works. It will scan a bitmap looking for the first instance of a bit being 1. Since we're interested in FREE chunks, the bitmap must be inverted to turn all the 0s into 1s.

```
//Rotate the bitmap (as to not lose items) to start
//at our randomly chosen offset
int RORBitmap = __ROR__(*Bitmap, StartOffset);

//since we're looking for 0's (FREE chunks)
//we'll invert the value due to how the next instruction works
int InverseBitmap = ~RORBitmap;

//these instructions search from low order bit to high order bit looking for a 1
//since we inverted our bitmap, the 1s will be 0s (BUSY) and the 0s will be 1s (FREE)
//    <-- search direction
//H.O                                                   L.O
//-------------------------------------------------------
//| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0
//-------------------------------------------------------
//the following code would look at the bitmap above, starting at L.O
//looking for a bit position that contains the value of one, and storing that index
int FreeIndex;
__asm{bsf FreeIndex, InverseBitmap};
```

Now the bitmap needs **updated** and the **address** of the actual chunk of memory needs to be derived from the **index** of the bitmap, which could be different than the `StartOffset` (depending on which chunks are free and which chunks are busy). The **Depth** can be decremented by **1** and the Hint is updated, although it is not used as an allocation offset anymore.

The header of the chunk is acquired by taking the starting address of the UserBlocks and **adding** the `FirstAllocationOffset`. Then the index from the bitmap (which has a one-to-one correlation to the UserBlocks) is **multiplied** by the `BlockStride`.

```
//shows the difference between the start search index and
//the actual index of the first free chunk found
int Delta = ((BYTE)FreeIndex + (BYTE)StartOffset) & 0x1F;

//now that we've found the index of the chunk we want to allocate
//mark it as 'used'; as it previously was 'free'
*Bitmap |= 1 << Delta;

//get the location (current index into the UserBlock)
int NewHint = Delta + sizeof(_HEAP_USERDATA_HEADER) *
(Bitmap - UserBlocks->BusyBitmap->Buffer);

AggrExchg.Depth = Depth - 1;
AggrExchg.Hint = NewHint;

//get the chunk header for the chunk that we just allocated
Header = (_HEAP_ENTRY)UserBlocks + UserBlocks->FirstAllocationOffset + (NewHint *
UserBlocks->BlockStride);
```

Finally there are some checks on the **header** to guarantee that a non-corrupted chunk is returned. Lastly the chunk's header is updated and returned to the calling function.

```
//if we've gotten a chunk that has certain attributes, report failure
if(Header->UnusedBytes & 0x3F)
	RtlpReportHeapFailure(14, LocalSegInfo->LocalData->LowFragHeap->Heap,
	Header, 0, 0, 0);

if(Header)
{
	if(Flags & 8)
		memset(Header + 8, 0, HeapBucket->BlockUnits - 8);

	//set the unused bytes if there are any
	int Unused = (HeapBucket->BlockUnits * 8) - Size;
	Header->UnusedBytes = Unused | 0x80;
	if(Unused >= 0x3F)
	{
		_HEAP_ENTRY *Next = Header + (8 * HeapBucket->BlockUnits) - 8;
		Next->PreviousSize = Unused;
		Header->UnusedBytes = 0xBF;
	}

	return Header + sizeof(_HEAP_ENTRY);
}
```

Unfortunately, there are times were a **_HEAP_SUBSEGMENT** and corresponding **UserBlocks** aren't initialized, for example the first LFH allocation for a specific size. The first thing that needs to happens, as shown above, is the Subsegment **cache** needs to be searched. If a cached _HEAP_SUBSEGMENT doesn't exist, one will be created later.

```
_HEAP_SUBSEGMENT *NewSubseg = NULL;
NewSubseg = SearchCache(LocalSegInfo);
```

Note: I've narrowed down cache search functionality. Please look at the binary for more detailed information.

At this point, a **UserBlocks** needs to be **created** so chunks of the requested size will be available to the LFH. While the exact formula to determine the overall UserBlocks size is a bit complicated, it will suffice to say that it is based off the **size** requested, the **total number** of chunks that exist for that size (per _HEAP_LOCAL_SEGMENT_INFO), and **affinity**.

```
int PageShift, BlockSize;
int TotalBlocks = LocalSegInfo->Counters->TotalBlocks;

//Based on the amount of chunks allocated for a given
//_HEAP_LOCAL_SEGMENT_INFO structure, and the _HEAP_BUCKET
//size and affinity formulate how many pages to allocate
CalculateUserBlocksSize(HeapBucket, &PageShift, &TotalBlocks, &BlockSize);
```

Note: Please see the binary for much more detailed information on the UserBlocks size calculation.

The next portion of code was added during the **Consumer Preview** as a way to prevent sequential overflows from corrupting **adjacent** memory. By signaling that a **guard page** should be present if certain criteria are met, the Heap Manager can ensure that some overflows will attempt to **access invalid** memory, **terminating** the process. This guard page flag is then passed to `RtlpAllocateUserBlock` so additional memory will be accounted for when **UserBlocks allocation** takes place.

```
//If we've seen enough allocations or the number of pages
//to allocate is very large, we're going to set a guard page
//after the UserBlocks container
bool SetGuard = false;
if(PageShift == 0x12 || TotalBlocks >= 0x400)
       SetGuard = true;

//Allocate memory for a new UserBlocks structure
_HEAP_USERDATA_HEADER *UserBlock =
       RtlpAllocateUserBlock(LFH, PageShift, BlockSize + 8, SetGuard);

if(UserBlock == NULL)
       return 0;
```

The Windows 8 version of `RtlpAllocateUserBlock` is almost like its Windows 7 counterpart, albeit with one small difference. Instead of handling the **BackEnd** allocation itself, the responsibilities are passed off to a function called `RtlpAllocateUserBlockFromHeap`.

`RtlpAllocateUserBlock` has a function signature of:

```
_HEAP_USERDATA_HEADER *RtlpAllocateUserBlock(_LFH_HEAP *LFH, unsigned __int8 PageShift,
int ChunkSize, bool SetGuardPage)
```

```
_HEAP_USERDATA_HEADER *UserBlocks;
int ByteSize = 1 << PageShift;
if(ByteSize > 0x78000)
	ByteSize = 0x78000;

UserBlocks = CheckCache(LFH->UserBlockCache, PageShift);
if(!UserBlocks)
	UserBlocks = RtlpAllocateUserBlockFromHeap(LFH->Heap, PageShift,
				ChunkSize, SetGuardPage);

UpdateCounters(LFH->UserBlockCache, PageShift);

return UserBlocks;
```

`RtlpAllocateUserBlockFromHeap` serves as the **allocator** for the **UserBlock container** with the small caveat of adding a guard page if necessary. Its function signature is:

```
_HEAP_USERDATA_HEADER *RtlpAllocateUserBlockFromHeap(_HEAP *Heap, PageShift, ChunkSize,
SetGuardPage)
```

The first order of business for `RtlpAllocateUserBlockFromHeap` is to get the **proper size**, in bytes, to be allocated for the desired UserBlock container, while enforcing a maximum value. It will then allocate the UserBlocks and return NULL if there is insufficient memory.

```
int ByteSize = 1 << PageShift;
if(ByteSize > 0x78000)
	ByteSize = 0x78000;

int SizeNoHeader = ByteSize - 8;
int SizeNoHeaderOrig = SizeNoHeader;

//Add extra space for the guard page
if(SetGuardPage)
	SizeNoHeader += 0x2000;

_HEAP_USERDATA_HEADER *UserBlocks = RtlAllocatHeap(Heap, 0x800001, SizeNoHeader);
if(!UserBlocks)
	return NULL;
```

Next RtlpAllocateUserBlockFromHeap will check if the SetGuardPage variable is set to true, indicating that, indeed, additional **protection** between UserBlocks should be enforced. If additional protection is necessary, an extra page (**0x1000** bytes) of memory is added to the overall total and given access permissions of PAGE_NOACCESS. Lastly the _HEAP_USERDATA_HEADER members are updated to indicate that a **guard page** was added and the container object is returned to RtlpAllocateUserBlock.

```
if(!SetGuardPage)
{
	UserBlocks->GuardPagePresent = false;
	return UserBlocks;
}

//add in a guard page so that a sequential overflow will fail
//as PAGE_NOACCESS will raise a AV on read/write
int GuardPageSize = 0x1000;
int AlignedAddr = (UserBlocks + SizeNoHeaderOrig + 0xFFF) & 0xFFFFF000;
int NewSize = (AlignedAddr - UserBlocks) + GuardPageSize;

//reallocate the memory
UserBlocks = RtlReAllocateHeap(Heap, 0x800001, UserBlocks, NewSize);

//Sets the last page (0x1000 bytes) of the memory chunk to PAGE_NOACCESS (0x1)
//http://msdn.microsoft.com/en-us/library/windows/desktop/aa366786(v=vs.85).aspx
ZwProtectVirtualMemory(-1, &AlignedAddr, &GuardPageSize, PAGE_NOACCESS, &output);

//Update the meta data for the UserBlocks
UserBlocks->GuardPagePresent = true;
UserBlocks->PaddingBytes = (SizeNoHeader - GuardPageSize) - SizeNoHeaderOrig;
UserBlocks->SizeIndex = PageShift;

return UserBlocks;
```

From here RtlpAllocateUserBlock returns a **UserBlock container** back to RtlpLowFragHeapAllocFromContext which will eventually be associated with a **_HEAP_SUBSEGMENT** structure.

The **Subsegment** will either come from a _HEAP_SUBSEGMENT **zone**, which is a region of **pre-allocated** memory that is specifically designed to hold an array of Subsegment structures, or by acquiring a Subsegment via a previously deleted structure. If a Subsegment **cannot** be procured, the FrontEnd allocator has **failed** and will return, resulting in the BackEnd heap servicing the request.

```
//See if there are previously deleted Subsegments to use
NewSubseg = CheckDeletedSubsegs(LocalSegInfo);

if(!NewSubseg)
	NewSubseg = RtlpLowFragHeapAllocateFromZone(LFH, AffinityIndex);

//if we can't get a subsegment we can't fulfill this allocation
if(!NewSubseg)
	return;
```

`RtlpLowFragHeapAllocateFromZone` is also responsible for providing a _HEAP_SUBSEGMENT back to the FrontEnd heap. It will attempt to pull an item from a previously allocated **pool** (or **zone**), allocating a new pool if one is not present or lacks sufficient space. It has a function signature of:

```
_HEAP_SUBSEGMENT *RtlpLowFragHeapAllocateFromZone(_LFH_HEAP *LFH, int AffinityIndex)
```

The first operation that `RtlpLowFragHeapAllocateFromZone` performs is attempting to acquire a **_HEAP_SUBSEGMENT** from the pre-allocated zone stored in the **_HEAP_LOCAL_DATA** structure. If a zone doesn't exist or doesn't contain **sufficient** space, one will be **created**. Otherwise, the Subsegment is returned back `RtlpLowFragHeapAllocFromContext`.

```
int LocalIndex = AffinityIndex * sizeof(_HEAP_LOCAL_DATA);
_LFH_BLOCK_ZONE *Zone = NULL;
_LFH_BLOCK_ZONE *NewZone;
char *FreePtr = NULL;

try_zone:
//if there aren't any CrtZones allocate some
Zone = LFH->LocalData[LocalIndex]->CrtZone;
if(Zone)
{
	//this is actually done atomically
	FreePtr = Zone->FreePointer;
	if(FreePtr + 0x28 < Zone->Limit)
	{
		AtomicIncrement(&Zone->FreePointer, 0x28);
		return FreePtr;
	}
}
```

There may **not** always be a **zone** or **sufficient** space, so the function will attempt to allocate memory from the BackEnd heap to use if needed. Pending allocation success and linked list checking, the new zone will be **linked in** and a _HEAP_SUBSEGMENT will be **returned**. If the doubly linked list is corrupted, execution will immediate **halt** by triggering an interrupt.

```
//allocate 1016 bytes for _LFH_BLOCK_ZONE structs
NewZone = RtlAllocateHeap(LFH->Heap, 0x800000, 0x3F8);
if(!NewZone)
	return 0;

_LIST_ENTRY *ZoneHead = &LFH->SubSegmentZones;
if(ZoneHead->Flink->Blink == ZoneHead &&
	ZoneHeader->Blink->Flink == ZoneHead)
{
	LinkIn(NewZone);

	NewZone->Limit = NewZone + 0x3F8;
	NewZone->FreePointer = NewZone + sizeof(_LFH_BLOCK_ZONE);

	//set the current localdata
	LFH->LocalData[LocalIndex]->CrtZone = NewZone;
	goto try_zone;
}
else
{
	//fast fail!
	__asm{int 0x29};
}
```

Note: The `int 0x29` interrupt was added as a way for developers to quickly terminate execution in the event of linked list corruption. Please see the Security Mitigations section for more information.

The `RtlpLowFragHeapAllocFromContext` now has a **UserBlock** container and a viable **Subsegment** for association. Now the FrontEnd can initialize all the data in the UserBlocks and set members of the _HEAP_SUBSEGMENT, which is achieved by calling `RtlpSubSegmentInitialize`. It has a function signature of:

```
int RtlpSubSegmentInitialize(_LFH_HEAP *LFH, _HEAP_SUBSEGMENT *NewSubSeg,
_HEAP_USERDATA_HEADER *UserBlocks, int ChunkSize, int SizeNoHeader, _HEAP_BUCKET
*HeapBucket)
```

```
//Initialize the Subsegment, which will divide out the
//chunks in the UserBlock by writing a _HEAP_ENTRY header
//every HeapBucket->BlockUnits bytes
NewSubseg->AffinityIndex = AffinityIndex;
RtlpSubSegmentInitialize(LFH, NewSubseg, UserBlock,
	RtlpBucketBlockSizes[HeapBucket->SizeIndex], SizeIndex - 8, HeapBucket);
```

`RtlpSubSegmentInitialize` will first attempt to find the proper _HEAP_LOCAL_SEGMENT_INFO structure for association by using the **affinity** and allocation request **size** as inputs to the **SegmentInfoArrays**.

```
_HEAP_LOCAL_SEGMENT_INFO *SegmentInfo;
_INTERLOCK_SEQ *AggrExchg = NewSubSeg->AggregateExchg;

int AffinityIndex = NewSubSeg->AffinityIndex;
int SizeIndex = HeapBucket->SizeIndex;

//get the proper _HEAP_LOCAL_SEGMENT_INFO based on affinity
if(AffinityIndex)
	SegmentInfo = LFH->AffinitizedInfoArrays[SizeIndex][AffinityIndex - 1];
else
	SegmentInfo = LFH->SegmentInfoArrays[SizeIndex];
```

Next `RtlpSubSegmentInitialize` is going to calculate the **size** of each **chunk** that will be in the UserBlock container by taking the 8-byte **rounded** size and adding space for a chunk **header**. Once the total size of each **chunk** is determined, the **total** number of chunks can be calculated, taking into account the space for the _HEAP_USERDATA_HEADER structure. With the **total size** of chunks and the amount of memory available to a UserBlocks finalized, the address for the **first free offset** can be calculated for the UserBlocks.

```
//figure out the total sizes of each chunk in the UserBlocks
unsigned int TotalSize = ChunkSize + sizeof(_HEAP_ENTRY);
unsigned short BlockSize = TotalSize / 8;

//this will be the number of chunks in the UserBlocks
unsigned int NumOfChunks = (SizeNoHeader - sizeof(_HEAP_USERDATA_HEADER)) / TotalSize;

//Set the _HEAP_SUBSEGMENT and denote the end
UserBlocks->SfreeListEntry.Next = NewSubSeg;

char *UserBlockEnd = UserBlock + SizeNoHeader;

//Get the offset of the first chunk that can be allocated
//Windows 7 just used 0x2 (2 * 8), which was the size
//of the _HEAP_USERDATA_HEADER
unsigned int FirstAllocOffset = ((((NumOfChunks + 0x1F) / 8) & 0x1FFFFFFC) +
	sizeof(_HEAP_USERDATA_HEADER)) & 0xFFFFFFF8;

UserBlocks->FirstAllocationOffset = FirstAllocOffset;
```

Note: The `FirstAllocationOffset` was not needed in Windows 7 as the first free entry was implicitly after the 0x10 byte _HEAP_USERDATA_HEADER.

After the **size** and **quantity** are calculated, RtlpSubSegmentInitialize will iterate through the **contiguous** piece of memory that currently makes up the **UserBlocks**, writing a **_HEAP_ENTRY** header for each chunk.

```
//if permitted, start writing chunk headers every TotalSize bytes
if(UserBlocks + FirstAllocOffset + TotalSize < UserBlockEnd)
{
	_HEAP_ENTRY *CurrHeader = UserBlocks + FirstAllocOffset;

	do
	{
		//set the encoded lfh chunk header, by XORing certain
		//values. This is how a Subsegment can be derived in RtlpLowFragHeapFree
		*(DWORD)CurrHeader = (DWORD)Heap->Entry ^ NewSubSeg ^
			RtlpLFHKey ^ (CurrHeader >> 3);

		//FreeEntryOffset replacement
		CurrHeader->PreviousSize = Index;

		//denote as a free chunk in the LFH
		CurrHeader->UnusedBytes = 0x80;

		//increment the header and counter
		CurrHeader += TotalSize;
		Index++;
	}
	while((CurrHeader + TotalSize) < UserBlockEnd);
}
```

Note: You'll notice how there is no FreeEntryOffset being set, but the Index is stored in the PreviousSize field of the chunk header. The index is used to update the bitmap when freeing a chunk from the LFH.

Since there is **no** longer a **FreeEntryOffset** in each chunk, the UserBlocks must track free chunks in a different way. It does free chunk tracking by marking up an associated **bitmap** which has a **1-to-1** ratio of bits to chunks in the UserBlocks. Initially all bits will be set to **zero**, owning to every chunk in the container being **free** (it has just been created). After the bitmap is **updated**, the function will associate the _HEAP_LOCAL_SEGMENT_INFO (**SegmentInfo**) and _HEAP_USERDATA_HEADER (**UserBlocks**) with the newly acquired/created _HEAP_SUBSEGMENT (**NewSubSeg**).

```
//Initialize the bitmap and zero out its memory (Index == Number of Chunks)
RtlInitializeBitMap(&UserBlocks->BusyBitmap; UserBlocks->BitmapData, Index);

char *Bitmap = UserBlocks->BusyBitmap->Buffer;

unsigned int BitmapSize = UserBlocks->BusyBitmap->SizeOfBitMap;

memset(Bitmap, 0, (BitmapSize + 7) / 8);

//This will set all the members of this structure
//to the appropriate values derived from this func
//associating UserBlocks and SegmentInfo
UpdateSubsegment(NewSubSeg,SegmentInfo, UserBlocks);
```

Lastly, `RtlpSubSegmentInitialize` will save the new **Depth** (number of chunks) and **Hint** (offset to a free chunk) in the newly created **_INTERLOCK_SEQ** structure. Also, `RtlpLowFragHeapRandomData` will be updated, which is the array that stores unsigned random bytes used as starting points when looking for free chunks within a UserBlock container.

```
//Update the random values each time a _HEAP_SUBSEGMENT is init
int DataSlot = NtCurrentTeb()->LowFragHeapDataSlot;

//RtlpLowFragHeapRandomData is generated in
//RtlpInitializeLfhRandomDataArray() via RtlpCreateLowFragHeap
short RandWord = GetRandWord(RtlpLowFragHeapRandomData, DataSlot);
NtCurrentTeb()->LowFragHeapDataSlot = (DataSlot + 2) & 0xFF;

//update the depth to be the amount of chunks we created
_INTERLOCK_SEQ NewAggrExchg;
NewAggrExchg.Depth = Index;
NewAggrExchg.Hint = RandWord % (Index << 16);

//swap of the old and new aggr_exchg
int Result = _InterlockedCompareExchange(&NewSubSeg->AggregateExchg,
       NewAggrExchg, AggrExchg);

//update the previously used SHORT w/ new random values
if(!(RtlpLowFragHeapGlobalFlags & 2))
{
       unsigned short Slot = NtCurrentTeb()->LowFragHeapDataSlot;

       //ensure that all bytes are unsigned
       int Rand1 = RtlpHeapGenerateRandomValue32() & 0x7F7F7F7F;
       int Rand2 = RtlpHeapGenerateRandomValue32() & 0x7F7F7F7F;

       //reassign the random data so it's not the same for each Subsegment
       RtlpLowFragHeapRandomData[Slot] = Rand1;
       RtlpLowFragHeapRandomData[Slot+1] = Rand2;
}

return result;
```

`RtlpLowFragHeapAllocFromContext` has now acquired and calibrated all the information needed to service the request. The **UserBlock container** has been created based on the desired size. The **Subsegment** has been acquired through various channels and **associated** with the UserBlocks. Lastly, the large contiguous piece of memory for the UserBlocks has been separated into user digestible chunks. `RtlpLowFragHeapAllocFromContext` will skip back to the beginning where the **ActiveSubsegment** was used to service the allocation.

```
UserBlock->Signature = 0xF0E0D0C0;

LocalSegInfo->ActiveSubsegment = NewSubseg;

//same logic seen in previous code
goto use_active_subsegment;
```

Algorithms – Freeing

This section will go over the freeing algorithms used by the Windows 8 Heap Manager. The first subsection will cover the **Intermediate** algorithm which determines whether the chunk being freed will reside in LFH or the BackHeap heap. The second subsection details the **BackEnd** freeing mechanism, which is familiar, owing to its doubly linked list architecture. Finally, the **LFH** de-allocation routine will be thoroughly examined. While the intermediate and BackEnd algorithms may look strikingly similar to the Windows 7 versions, the FrontEnd (LFH) freeing mechanism has changed significantly.

Note: Much of the code has been left out to simplify the learning process. Please contact me if more in-depth information is desired

Intermediate

Before a chunk can be officially freed, the Heap Manager must decide if the responsibility lies with the **BackEnd** or **FrontHeap** heap. The function that makes this decision is `RtlFreeHeap`. It has a function signature of:

```
int RtlFreeHeap(_HEAP *Heap, int Flags, void *Mem)
```

The first step taken by `RtlFreeHeap` is to ensure that a **non-NULL** address is passed to the function. If the chunk is NULL, the function will just return. Therefore, the **freeing** of **NULL** chunks to the user land heap has no effect. Next, the flags are examined to determine if the **BackEnd** freeing routine should be used before any other validation occurs.

```
 //the user-land memory allocator won't actually
//free a NULL chunk passed to it
if(!Mem)
	return;

//the header to be used in the freeing process
_HEAP_ENTRY *Header = NULL;
_HEAP_ENTRY *HeaderOrig = NULL;

//you can force the heap to ALWAYS use the back-end manager
if(Heap->ForceFlags & 0x1000000)
	return RtlpFreeHeap(Heap, Flags | 2, Header, Mem);
```

`RtlFreeHeap` will now ensure that the memory being freed is **8-byte** aligned, as all heap memory should be 8-byte aligned. If it is not, then a heap **failure** will be reported and the function will return.

```
if(Mem & 7)
{
	RtlpLogHeapFailure(9, Heap, Mem, 0, 0, 0);
	return ERROR;
}
```

The **headers** can now be checked, which are always located 8-bytes behind the chunk of memory. The first header check will look at the **SegmentOffset** to discern if header relocation is necessary, and if so, the header will be moved backwards in memory. Then a check is made to guarantee that the adjusted header is of the right **type**, aborting if the type is incorrect.

```
//Get the _HEAP_ENTRY header
Header = Mem - 8;
HeaderOrig = Mem - 8;

//ben hawkes technique will use this adjustment
//to point to another chunk of memory
if(Header->UnusedBytes == 0x5)
	Header -= 8 * Header->SegmentOffset;

//another header check to ensure valid frees
if(!(Header->UnusedBytes & 0x3F))
{
	RtlpLogHeapFailure(8, Heap, Header, 0, 0, 0);
	Header = NULL;
}

//if anything went wrong, return ERROR
if(!Header)
	return ERROR;
```

Additional header validation mechanisms have been added to prevent an exploitation technique published by Ben Hawkes back in 2008 (Hawkes 2008). If header **relocation** has taken place and the chunk resides in the LFH, the algorithm **verifies** the adjusted header is actually meant to be freed by calling `RtlpValidateLFHBlock`. If the chunk is not in the LFH, the headers are verified the traditional way by validating that they are not tainted, returning **error** on **corruption**.

```
//look at the original header, NOT the adjusted
bool valid_chunk = false;
if(HeaderOrig->UnusedBytes == 0x5)
{
	//look at adjusted header to determine if in the LFH
	if(Header->UnusedBytes & 0x80)
	{
		//RIP Ben Hawkes SegmentOffset attack :(
		valid_chunk = RtlpValidateLFHBlock(Heap, Header);
	}
	else
	{
		if(Heap->EncodeFlagMask)
		{
			if(!DecodeValidateHeader(Heap, Header))
				RtlpLogHeapFailure(3, Heap, Header, Mem, 0, 0);
			else
				valid_chunk = true;
		}
	}

	//if it's found that this is a tainted chunk, return ERROR
	if(!valid_chunk)
		return ERROR_BAD_CHUNK;
}
```

Lastly `RtlFreeHeap` will **decode** the header (the **first 4-bytes** are **encoded**) and look at the `UnusedBytes` (**Offset 0x7**), which indicates if a chunk was allocated by the **LFH** or the **BackEnd** heap, choosing either `RtlpLowFragHeapFree` or `RtlpFreeHeap`, respectively.

```
//This will attempt to decode the header (diff for LFH and Back-End)
//and ensure that all the meta-data is correct
Header = DecodeValidateHeader(Heap, Header);

//being bitwase ANDed with 0x80 denotes a chunk from the LFH
if(Header->UnusedBytes & 0x80)
	return RtlpLowFragHeapFree(Heap, Header);
else
	return RtlpFreeHeap(Heap, Flags | 2, Header, Mem);
```

BackEnd

The Windows 8 BackEnd de-allocator is very **similar** to the Windows 7 BackEnd. It will insert a chunk being freed onto a **doubly-linked** list, but instead of updating **counters** in a **back** link, the routine will update the `FrontEndHeapUsageData` to indicate if the LFH should be used on subsequent allocations. The function responsible for BackEnd freeing is `RtlpFreeHeap` and has a signature of:

```
int RtlpFreeHeap(_HEAP *Heap, int Flags, _HEAP_ENTRY *Header, void *Chunk)
```

Before the act of freeing a chunk can be accomplished the heap manager will do some preliminary **validation** of the chunk being freed to ensure that it meets a certain level of integrity. The **chunk** is tested against the **address** of the **_HEAP** structure managing it to make sure they don't point to the same location. If that test passes, the chunk **header** will be **decoded** and **validated**. Both tests result in returning with **error** upon **failure**.

```
//prevent freeing of a _HEAP structure (Ben Hawkes technique dead)
if(Heap == Header)
{
	RtlpLogHeapFailure(9, Heap, Header, 0,0,0);
	return;
}

//attempt to decode and validate the header
//if it doesn't decode properly, abort
if(Heap->EncodeFlagMask)
	if(!DecodeValidateHeader(Header, Heap))
		return;
```

Note: The _HEAP structure check is new to Windows 8

The next step is to traverse the `BlocksIndex` structures looking for one that can **track** the chunk being freed (based on **size**). Before standard freeing occurs, the BackEnd will check to see if certain **header characteristics** exist, denoting a **virtually** allocated chunk and if so, call the virtual de-allocator.

```
//search for the appropriately sized blocksindex
_HEAP_LIST_LOOKUP *BlocksIndex = Heap->BlocksIndex;
do
{
	if(Header->Size < BlocksIndex->ArraySize)
		break;

	BlocksIndex = BlocksIndex->ExtendedLookup;
}
while(BlocksIndex);

//the UnusedBytes (offset: 0x7) are used for many things
//a value of 0x4 indicates that the chunk was virtually
//allocated and needs to be freed that way (safe linking included)
if(Header->UnusedBytes == 0x4)
	return VirtualFree(Head, Header);
```

`RtlpFreeHeap` will then update the heap's `FrontEndHeapUsageData` pending the **size** comparison of the chunk. This effectively will only **update** the **usage data** if the chunk being freed could be serviced by the **LFH** (FrontEnd). By **decrementing** the value, the **heuristic** to trigger LFH allocation for this size has been put back by one, requiring more **consecutive allocations** before the FrontEnd heap will be used.

```
//Get the size and check to see if it's under the
//maximum permitted for the LFH
int Size = Header->Size;

//if the chunk is capable of being serviced by the LFH then check the
//counters, if they are greater than 1 decrement the value to denote
//that an item has been freed, remember, you need at least 16 CONSECUTIVE
//allocations to enable the LFH for a given size
if(Size < Heap->FrontEndHeapMaximumIndex)
{
	if(!( (1 << Size & 7) & (heap->FrontEndStatusBitmap[Size / 8])))
	{
		if(Heap->FrontEndHeapUsageData[Size] > 1)
			Heap->FrontEndHeapUsageData[Size]--;
	}
}
```

Now that the validation and heuristics are out of the way, the de-allocator can attempt, pending the heap's **permission**, to **coalesce** chunks adjacent to the one being freed. What this means is that the chunk **before** and the chunk **after** are checked for being **FREE**. If either chunk is free then they will be combined into a larger chunk to avoid fragmentation (something the LFH directly addresses). If the total size of the combined chunks exceeds certain limits it will be **de-committed** and potentially added to a list of large **virtual** chunks.

```
//if we can coalesce the chunks adjacent to this one, do it to
//avoid fragmentation (something the LFH directly addresses)
int CoalescedSize;
if(!(heap->Flags 0x80))
{
	Header = RtlpCoalesceFreeBlocks(Heap, Header, &CoalescedSize, 0);

	//if the combined space is greater than the Heap->DecommittThreshold
	//then decommit the chunk from memory
	DetermineDecommitStatus(Heap, Header, CoalescedSize);

	//if the chunk is greater than the VirtualMemoryThreshold
	//insert it and update the appropriate lists
	if(CoalescedSize > 0xFE00)
		RtlpInsertFreeBlock(Heap, Header, CoalescedSize);
}
```

The chunk (which is **potentially bigger** than when it was originally submitted for freeing) is now ready to be **linked** into the **FreeLists**. The algorithm will start searching the **beginning** of the list for a chunk that is **greater than or equal** to the **size** of the chunk being freed to be used as the insertion point.

```
//get a pointer to the FreeList head
_LIST_ENTRY *InsertPoint = &Heap->FreeLists;
_LIST_ENTRY *NewNode;

//get the blocks index and attempt to assign
//the index at which to free the current chunk
_HEAP_LIST_LOOKUP *BlocksIndex = Heap->BlocksIndex;
int ListHintIndex;

Header->Flags = 0;
Header->UnusedBytes = 0;

//attempt to find the proper insertion point to insert
//chunk being freed, which will happen at the when a freelist
//entry that is greater than or equal to CoalescedSize is located
if(Heap->BlocksIndex)
	InsertPoint = RtlpFindEntry(Heap, CoalescedSize);
else
	InsertPoint = *InsertPoint;

//find the insertion point within the freelists
while(&heap->FreeLists != InsertPoint)
{
	_HEAP_ENTRY *CurrEntry = InsertPoint - 8;
	if(heap->EncodeFlagMask)
		DecodeHeader(CurrEntry, Heap);

	if(CoalescedSize <= CurrEntry->Size)
		break;

	InsertPoint = InsertPoint->Flink;
}
```

Before the chunk is **linked into** the FreeLists a check, which was introduced in Windows 7, is made to ensure that the **FreeLists** haven't been **corrupted**, avoiding the infamous **write-4** primitive (**insertion** attack).

```
//insertion attacks FOILED! Hi Brett Moore/Nico
NewNode = Header + 8;
_LIST_ENTRY *Blink = InsertPoint->Blink;
if(Blink->Flink == InsertPoint)
{
	NewNode->Flink = InsertPoint;
	NewNode->Blink = Blink;
	Blink->Flink = NewNode;
	Blink = NewNode;
}
else
{
	RtlpLogHeapFailure(12, 0, InsertPoint, 0, Blink->Flink, 0);
}
```

Lastly, the freeing routine will set the **TotalFreeSize** to reflect the overall amount of free space **gained** in this de-allocation and update the **ListHints**. Even though the FreeLists have been updated (code above) the ListHint optimizations must also be **updated** so that the FrontEnd Allocator can quickly find specifically sized chunks.

```
//update the total free blocks available to this heap
Heap->TotalFreeSize += Header->Size;

//if we have a valid _HEAP_LIST_LOOKUP structure, find
//the appropriate index to use to update the ListHints
if(BlocksIndex)
{
	int Size = Header->Size;
	int ListHintIndex;

	while(Size >= BlocksIndex->ArraySize)
	{
		if(!BlocksIndex->ExtendedLookup)
		{
			ListHintIndex = BlocksIndex->ArraySize - 1;
			break;
		}

		BlocksIndex = BlocksIndex->ExtendedLookup;
	}

	//add the current entry to the ListHints doubly linked list
	RtlpHeapAddListEntry(Heap, BlocksIndex, RtlpHeapFreeListCompare,
		NewNode, ListHintIndex, Size);
}
```

FrontEnd

The sole FrontEnd de-allocator for Windows 8 is the Low Fragmentation Heap (**LFH**), which can manage chunks that are **0x4000** bytes (**16k**) or below. Like the FrontEnd **Allocator**, the freeing mechanism puts chunks back into a **UserBlocks** but **no** longer relies on the _INTERLOCK_SEQ structure to determine the **offset** within the overall container. The new functionality that makes up the Windows 8 FrontEnd de-allocator makes freeing much more simple and secure. The function responsible for LFH freeing is `RtlpLowFragHeapFree` and has a function signature of:

```
int RtlpLowFragHeapFree(_HEAP *Heap, _HEAP_ENTRY *Header)
```

The first step in the LFH freeing process starts with deriving the **_HEAP_SUBSEGMENT** (Subsegment) and **_HEAP_USERDATA_HEADER** (UserBlocks) from the chunk being freed. While I don't officially categorize the Subsegment derivation as a security mechanism, it does **foil** the freeing of a chunk that has a corrupted chunk header (which would most likely occur through a **sequential heap overflow**).

```
//derive the subsegment from the chunk to be freed, this
//can royally screw up an exploit for a sequential overflow
_HEAP_SUBSEGMENT *Subseg = (DWORD)Heap ^ RtlpLFHKey ^ *(DWORD)Header ^ (Header >> 3);

_HEAP_USERDATA_HEADER *UserBlocks = Subseg->UserBlocks;

//Get the AggrExchg which contains the Depth (how many left)
//and the Hint (at what offset) [not really used anymore]
_INTERLOCK_SEQ *AggrExchg = AtomicAcquireIntSeq(Subseg);
```

Next, the **bitmap** must be **updated** to indicate that a chunk at a certain offset within the UserBlocks is now available for allocation, as it has just been freed. The **index** is acquired by accessing the `PreviousSize` field in the chunk header. This is quite similar to using the **FreeEntryOffset** in Windows 7 with the added protection of being protected by the encoded chunk header which precedes it.

```
//the PreviousSize is now used to hold the index into the UserBlock
//for each chunk. this is somewhat like the FreeEntryOffset used before it
//See RtlpSubSegmentInitialize() for details on how this is initialized
short BitmapIndex = Header->PreviousSize;

//Set the chunk as free
Header->UnusedBytes = 0x80;

//zero out the bitmap based on the predefined index set in RtlpSubSegmentInitialize
//via the BTR (Bit-test and Reset) x86 instruction
bittestandreset(UserBlocks->BusyBitmap->Buffer, BitmapIndex);
```

For all intents and purposes the chunk is now **FREE**, although additional actions must be performed. Any chunks that were meant to be **freed** previously but **failed** will be given another opportunity by accessing the `DelayFreeList`. Then the newly updated values of **Depth** (how many left) and **Hint** (where the next free chunk is) are assigned and updated to **reflect** the freed chunks. If the UserBlocks isn't completely FREE, that is there exists at least one chunk that is BUSY within a UserBlock container, then the **Subsegment** will be **updated** and the function will return.

```
//Chunks can be deferred for freeing at a later time
//If there are any of these chunks, attempt to free them
//by resetting the bitmap
int DelayedFreeCount;
if(Subseg->DelayFreeList->Depth)
       FreeDelayedChunks(Subseg, &DelayedFreeCount);

//now it's time to update the Depth and Hint for the current Subsegment
//1) The Depth will be increased by 1, since we're adding an item back into the UserBlock
//2) The Hint will be set to the index of the chunk being freed
_INTERLOCK_SEQ NewSeq;
int NewDepth = AggrExchg->Depth + 1 + DelayedFreeCount;
NewSeq.Depth = NewDepth;
NewSeq.Hint = BitmapIndex;

//if the UserBlocks still have BUSY chunks in it then update
//the AggregateExchg and return back to the calling function
if(!EmptyUserBlock(Subseg))
{
       Subseg->AggregateExchang = NewSeq;
       return NewSeq;
}
```

If it is determined that the **Subsegment** hosts a UserBlock container that is **no longer necessary** the freeing algorithm will **update** some of its members and proceed to mark the **Depth** and **Hint** to be **NULL**, indicating that there is **no** viable **UserBlocks** associated with the **Subsegment**.

```
//Update the list if we've freed any chunks
//that were previously in the delayed state
UpdateDelayedFreeList(Subseg);

//update the CachedItem[] array with the _HEAP_SUBSEGMENT
//we're about to free below
UpdateCache(Subseg->LocalInfo);

Subseg->AggregateExchang.Depth = 0;
Subseg->AggregateExchang.Hint = 0;

int ret = InterlockedExchange(&Subseg->ActiveSubsegment, 0);
if(ret)
       UpdateLockingMechanisms(Subseg)
```

Certain **flags** in the **_HEAP_SUBSEGMENT** might indicate that the next **page aligned address** from the **start** of the **UserBlocks** may be better off having **non-execute** permissions. The non-execute permissions will prevent memory, most likely from some sort of spray, from being used as an **executable pivot** in a potential exploit.

```
//if certain flags are set this will mark prtection for the next page in the userblock
if(Subseg->Flags & 3 != 0)
{
	//get a page aligned address
	void *PageAligned = (Subseg->UserBlock + 0x101F) & 0xFFFFF000;

	int UserBlockByteSize = Subseg->BlockCount * RtlpGetReservedBlockSize(Subseg);
	UserBlockByteSize *= 8;

	//depending on the flags, make the memory read/write or rwx
	//http://msdn.microsoft.com/en-us/library/windows/desktop/aa366786(v=vs.85).aspx
	DWORD Protect = PAGE_READWRITE;
	if(flags & 40000 != 0)
		Protect = PAGE_EXECUTE_READWRITE;

	//insert a non-executable memory page
	DWORD output;
	ZwProtectVirtualMemory(-1, &PageAligned, &UserBlockByteSize, Protect, &output);
}
```

Finally the **UserBlock** container can be **freed**, which means all the chunks within it are effectively freed (Although not freed individually).

```
//Free all the chunks (not individually) by freeing the UserBlocks structure
Subseg->UserBlocks->Signature = 0;
RtlpFreeUserBlock(Subseg->LocalInfo->LocalData->LowFragHeap, Subseg->UserBlocks);

return;
```

Security Mechanisms

This section will cover the security mechanisms introduced in Windows 8 Release Preview. These security features were added to directly address the most modern exploitation techniques employed by attackers at the time of writing. The anti-exploitation features will **start** with those residing in the **BackEnd** manager and then **continue** with mitigations present in the **FrontEnd**.

_HEAP Handle Protection

Back in 2008 Ben Hawkes proposed a payload that, if used to **overwrite** a **_HEAP** structure, could result in the execution of an attacker supplied **address** after a subsequent allocation (Hawkes 2008). Windows 8 mitigates the aforementioned exploitation technique by ensuring that a chunk being **freed** is not the **heap handle** that is freeing it. Although there may exist a corner case of a chunk being freed that belongs to a different _HEAP structure than the one freeing it, the likelihood is extremely low.

```
RtlpFreeHeap(_HEAP *heap, DWORD flags, void *header, void *mem)
{
	.
	.
	.

	if(heap == header)
	{
		RtlpLogHeapFailure(9, heap, header, 0, 0, 0);
		return 0;
	}

	.
	.
	.
}
```

Note: The same functionality exists in `RtlpReAllocateHeap()`

Virtual Memory Randomization

If an allocation request is received by `RtlpAllocateHeap` that exceeded the **VirtualMemoryThreshold** the heap manager will call `NtAllocateVirtualMemory()` instead of using the **FreeLists**. These virtual allocations have the tendency to have **predictable** memory layouts due to their infrequent use and could be used as a primitive in a memory corruption exploit. Windows 8 will now adds **randomness** to the address of each **virtual allocation**. Therefore each virtual allocation will start at a **random offset** within the overall virtual chunk, removing **predictability** of heap meta-data in the chance over a sequential overflow.

```
//VirtualMemoryThreshold set to 0x7F000 in CreateHeap()
int request_size = Round(request_size)
int block_size = request_size / 8;
if(block_size > heap->VirtualMemoryThreshold)
{
	int rand_offset = (RtlpHeapGenerateRandomValue32() & 0xF) << 12;

	request_size += 24;

	int region_size = request_size + 0x1000 + rand_offset;

	void *virtual_base, *virtual_chunk;

	int protect = PAGE_READWRITE;
	if(heap->flags & 0x40000)
		protect = PAGE_EXECUTE_READWRITE;

	//Attempt to reserve region size bytes of memory
	if(NtAllocateVirtualMemory(-1, &virtual_base, 0, &region_size,
		MEM_RESERVE, protect) < 0)
		goto cleanup_and_return;

	virtual_chunk = virtual_base + rand_offset;
	if(NtAllocateVirtualMemory(-1, &virtual_chunk, 0, &request_size,
		MEM_COMMIT, protect) < 0)
		goto cleanup_and_return;

	//XXX Set headers and safe link-in

	return virtual_chunk;
}
```

Note: The size of each virtually allocated chunk is also randomized to a certain extent providing additional protection against heap determinism.

FrontEnd Activation

Windows 7 used the **ListHints** as a **multi-purpose** data structure. The first function was to provide an optimization to the BackEnd heap when servicing allocations, instead of having to completely traverse the **FreeLists**. The second function was to use the **ListHint->Blink** for an allocation counter and data storage. If the allocation counter exceeded the threshold (**16 consecutive** allocations of the same size), the **LFH** would be activated for that **size** and the address of **_HEAP_BUCKET** would be placed in the Blink. This dual-purpose functionality has been **replaced** with a much more efficient and straight forward solution using **dedicated** counters and a **bitmap**. The new data structures are used to indicate how many allocations for a specific size have been requested and what **_HEAP_BUCKETs** are activated.

As you saw in the **BackEnd Algorithms** allocation section, the **FrontEndHeapUsageData** array is used to store the allocation **count** or the **index** into the **_HEAP_BUCKET array** within a _LFH_HEAP. These two measures make bucket activation less complicated while also **mitigating** the _HEAP_BUCKET overwrite **attack** made popular by Ben Hawkes a few years ago (Hawkes 2008).

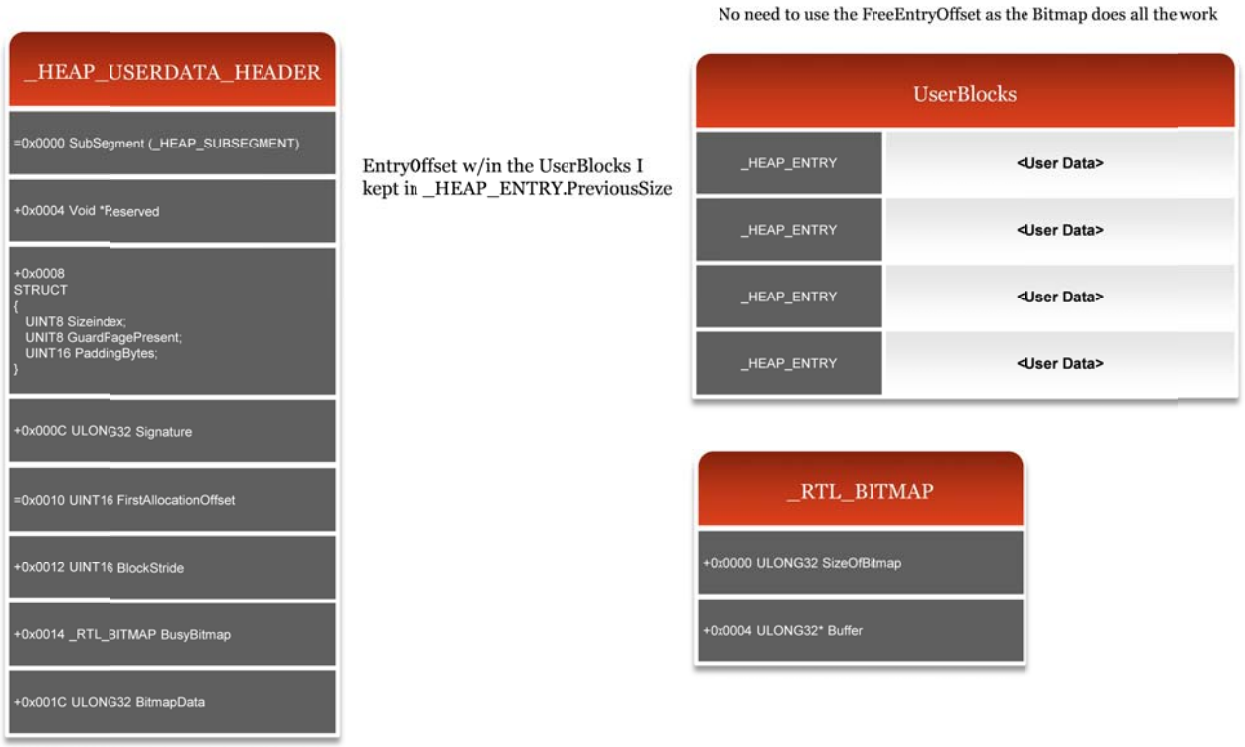
```
else if(Size < 0x4000)
{
	//Heap->FrontEndHeapStatusBitmap has 256 possible entries
	int BitmapIndex = BlockSize / 8;
	int BitPos = BlockSize & 7;

	//determine if the LFH should be activated
	if(!( (1 << BitPos) & Heap->FrontEndHeapStatusBitmap[BitmapIndex]) )
	{
		//increment the counter used to determine when to use the LFH
		int Count = Heap->FrontEndHeapUsageData[BlockSize] + 0x21;
		Heap->FrontEndHeapUsageData[BlockSize] = Count;

		//if there were 16 consecutive allocation or many allocations consider LFH
		if((Count & 0x1F) > 0x10 || Count > 0xFF00)
		{
			_LFH_HEAP *LFH = NULL;
			if(Heap->FrontEndHeapType == 2)
				LFH = heap->FrontEndHeap;

			//if the LFH is activated, it will return a valid index
			short BucketIndex =  RtlpGetLFHContext(LFH, Size);
			if(BucketIndex != -1)
			{
				//store the heap bucket index and update accordingly
				Heap->FrontEndHeapUsageData[BlockSize] = BucketIndex;
				Heap->FrontEndHeapStatusBitmap[BitmapIndex] |= 1 << BitPos;
			}
			else if (BucketIndex > 0x10)
			{
				//if we haven't been using the LFH, we will next time around
				if(!LFH)
					Heap->CompatibilityFlags |= 0x20000000;
			}
		}
	}
}
```

FrontEnd Allocation

Not only have LFH enabling data structures changed, but the way chunks are allocated from a UserBlocks have changed as well. Prior to Windows 8, the FrontEnd allocator would rely on the **_INTERLOCK_SEQ.Hint** to determine where the **next free** chunk resided and then use the **FreeEntryOffset** to update the Hint. Unfortunately for Microsoft, that routine **failed** to do any **validation** on the FreeEntryOffset which gave an attacker the ability to overwrite arbitrary memory from the **base** of the **UserBlocks** (Phrack 68).

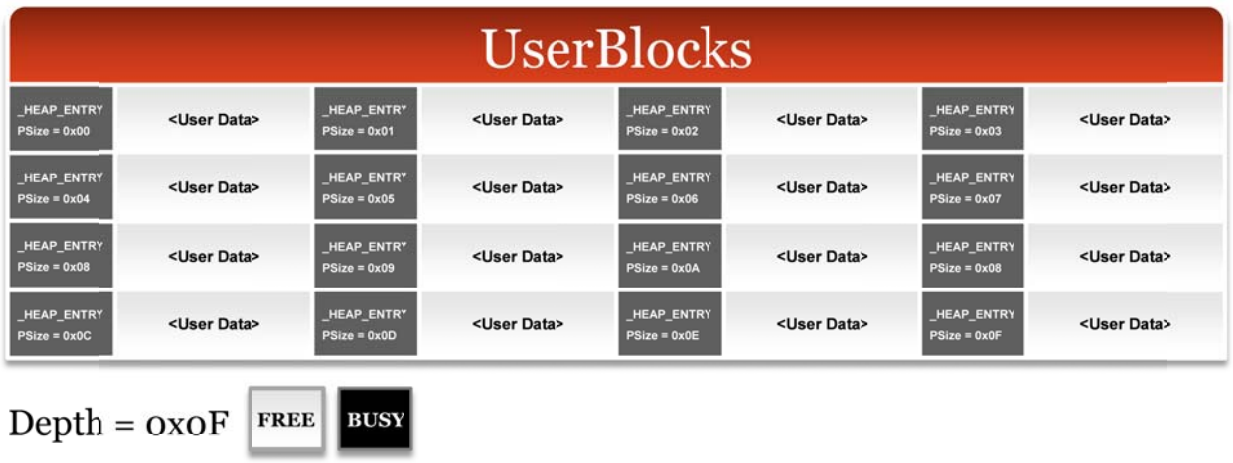
Also, since chunks were allocated in **contiguous** memory and pointed to the **next** free chunk in the container (which could be adjacent), sequential allocations had the potential to be **predictable** under certain circumstances. The result was the ability to determine which chunk within the UserBlocks would be **allocated** or **freed** next, enabling heap determinism for use-after-free bugs and sequential heap overflows.

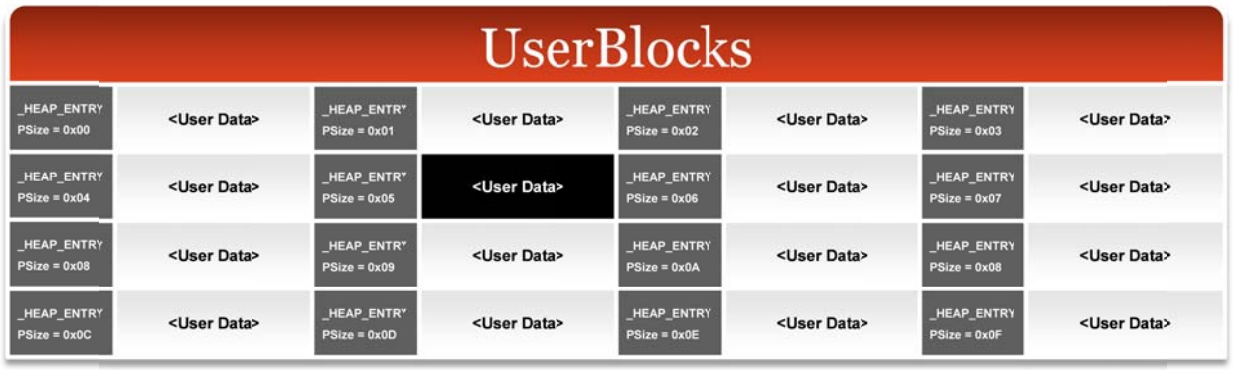
Windows 8 directly addresses both problems. First, the **FreeEntryOffset** has been completely **removed** in favor of a **bitmap** that is added to the **_HEAP_USERDATA_HEADER**. The newly created bitmap has **1-bit** representing **each chunk** in the UserBlocks. The bitmap is searched and the corresponding **index** indicates which chunk from the UserBlocks to use.



Note: Please see FrontEnd Allocation section above for corresponding code.

This brings us to the second issues of predictable memory locations being used when making allocations from the Windows 7 LFH. This problem too was remediated in Windows 8 by starting the **bitmap** search at a **random** location, instead of always selecting the free chunk based off the _INTERLOCK_SEQ.Hint variable.





Fast Fail

Linked lists are quite common for storing collections of similar objects within the Windows operating system. These same linked lists have also been the **target** of attackers since the advent of **heap overflow** exploitation. By corrupting a linked list entry (either Flink or Blink depending on the list type), an attacker can effectively **write a 4-byte** value to an arbitrary memory address. Although many checks have been implemented, staring in Windows XP SP2, there still exist link-in and unlinking code that may not behave according to security standards.

The **fast fail** interrupt was designed to give application developers the ability to **terminate** a process without having to know if the proper **flags** were **designated** on heap creation. For example, if **HeapEnableTerminationOnCorruption** is not set via the `HeapSetInformation` API then an application might not terminate even if an error function was called. Fail fail makes it very easy to halt the execution of a process by issuing a simple interrupt of `int 0x29`.

You can see `RtlpLowFragHeapAllocateFromZone` implements this new interrupt when checking the **zones** to ensure the integrity of the linked list. You can search other binaries for `int 0x29` to see all of its usage.

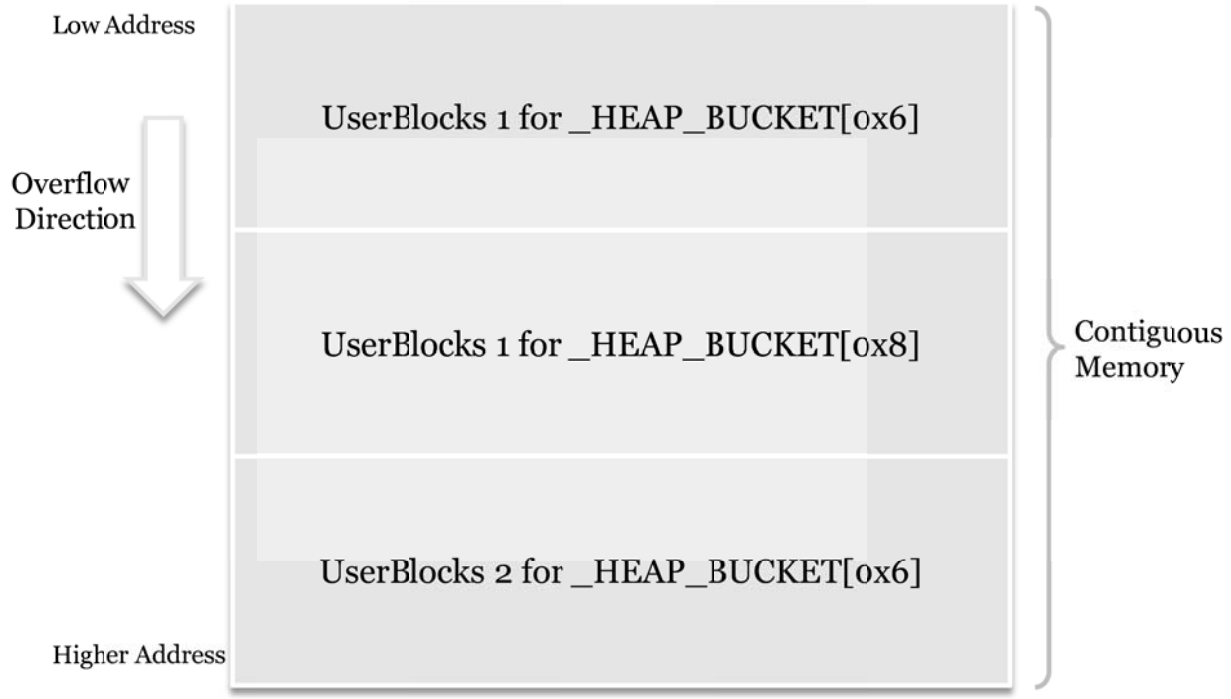
```
_HEAP_SUBSEGMENT *RtlpLowFragHeapAllocateFromZone(_LFH_HEAP *LFH, int AffinityIndex)
{
	.
	.
	.

	_LIST_ENTRY *subseg_zones = &LFH->SubSegmentZones;
	if(LFH->SubSegmentZones->Flink->Blink != subseg_zones ||
		LFH->SubSegmentZones->Blink->Flink != subseg_zones)
		__asm{int 29};

}
```

Note: A less comprehensive check was used on Windows 7

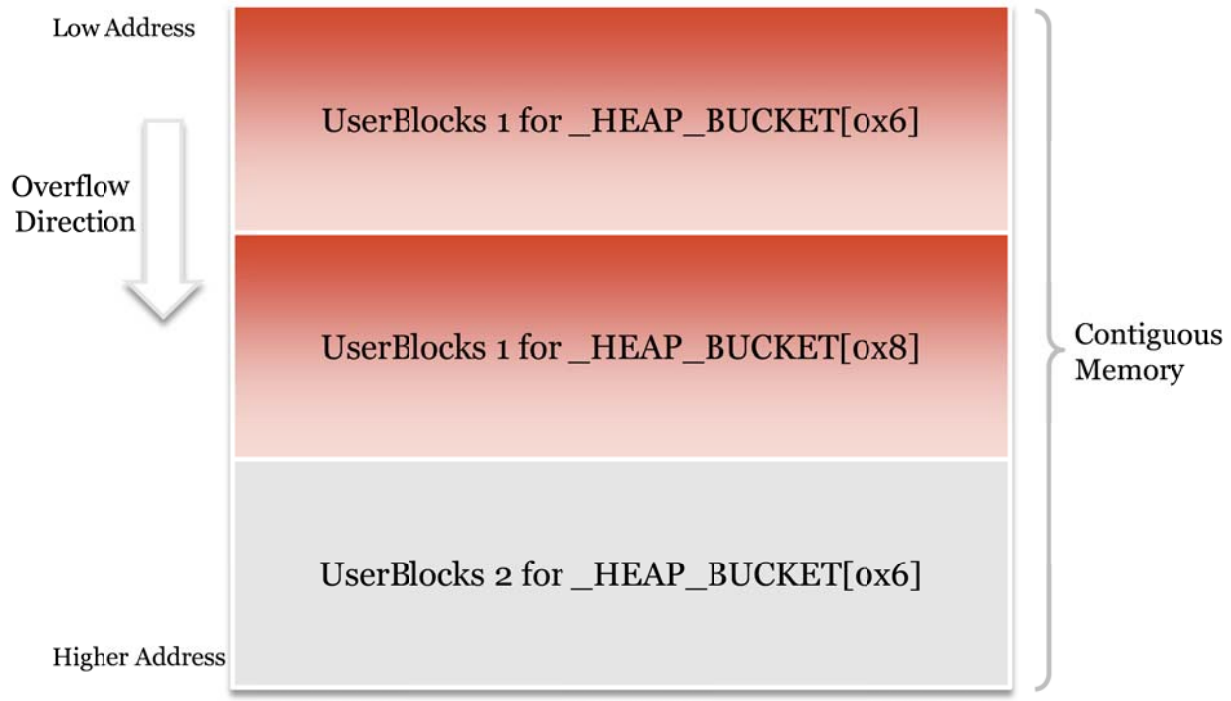
Guard Pages

The LFH was designed to be a fast, reliable way to acquire memory with minimal fragmentation. Unfortunately, there exist certain situations that can result in heap **meta-data** being **overwritten** if a sequential **overflow** occurs. If a chunk in the LFH can **overflow** into adjacent chunks within the same UserBlocks, then it may also be able to overwrite into another UserBlock **container** (which may hold the same of differently sized chunks).

In Windows 7, the address space used for UserBlocks could potentially reside in **contiguous** memory. That means that a UserBlock containing chunks of **one size** could reside in **memory** next to a UserBlocks containing chunks of another size (or the same size).



For example, in the diagram below a UserBlock container for chunks of **0x30 bytes** (0x6 blocks) could reside next to a UserBlock container hold chunks of **0x40 bytes** (0x8 blocks). A sequential overflow from a 0x6 block sized chunk could potentially overwrite the meta-data for the **_HEAP_USERDATA_HEADER** that lies before the container holding the 0x40 byte chunks (along with data inside the 0x8 block sized chunks).



Windows 8 identified the problem with sequential overflow within the UserBlocks (to a certain extent) by **randomizing** the **index** which services an allocation request. Microsoft also identified the need for **security** between UserBlock **containers** as well. If certain heuristics are triggered, such as a large number of allocations for one size, a **guard page** will be inserted after the UserBlocks, preventing sequential overflows from corrupting any data that may come **after** the UserBlocks.

Arbitrary Free

Another technique founded by Ben Hawkes was the **arbitrary freeing** of chunks in the LFH by overwriting the **Segment Offset** (Hawkes 2008). If certain flags are set, the address of the chunk header is **adjusted** to point to a new location.

```
RtlFreeHeap(_HEAP *Heap, DWORD Flags, void *Mem)
{
	.
	.

	//if the header denotes a different segment
	//then adjust the header accordingly
	_HEAP_ENTRY *Header = Mem - 8;
	_HEAP_ENTRY *HeaderOrig = Mem – 8;
	if(Header->UnusedBytes == 0x5)
		Header -= 8 * Header->SegmentOffset;

	if(!(Header->UnusedBytes & 0x3F))
	{
		//this will prevent the chunk from being freed
		RtlpLogHeapFailure(8, Heap, Header, 0,0,0);
		Header = NULL;
	}
	.
	.
}
```

In Windows 7, the newly adjusted, and potentially **dangerous**, chunk would be passed to the FrontEnd **de-allocator**. The result would be freeing a chunk of memory that may be currently **BUSY**, presenting an opportunity for an attacker to change data that is currently in use by a process.

Windows 8 has inserted **checks** into the intermediate freeing routine to ensure that if a chunk has been adjusted it is **valid chunk**, as adjusted chunks should have certain **characteristics**. If the characteristics are not met, then the function will **fail** and return.

```
if(HeaderOrig->UnusedBytes == 0x5)
{
	//this chunk was from the LFH
	if(Header->UnusedBytes & 0x80)
	{
		//ensures that the header values haven't been altered
		if(!RtlpValidateLFHBlock(Heap, Header))
		{
			RtlpLogHeapFailure(3, Heap, Header, Mem, 0, 0);
			return 0;
		}
	}
}
```

Exception Handling

The Windows 7 LFH allocator was subject to **catch-all** exception handling when attempting **allocations**. Essentially, the function would catch **any error**, such as a memory access violation, and return NULL, leaving the allocation responsibilities to the **BackEnd** allocator.

```
int RtlpLowFragHeapAllocFromContext(_HEAP_BUCKET *aHeapBucket, int aBytes)
{
	try {
		//Attempt allocatcion
	}
	catch
	{
		return 0;
	}
}
```

I **theorized** that this catch-all behavior could actually be **abused** by attackers if they had the ability to trigger multiple overflows (Valasek 2010). This brute forcing effect could permit attackers to bypass something like ASLR due to invalid address access being handled.

The Windows heap team has identified that non-specific exception handling may lead to security exposure and **removed** the **exception handler** from `RtlpLowFragHeapAllocFromContext` to prevent any malicious intent.

Exploitation Tactics

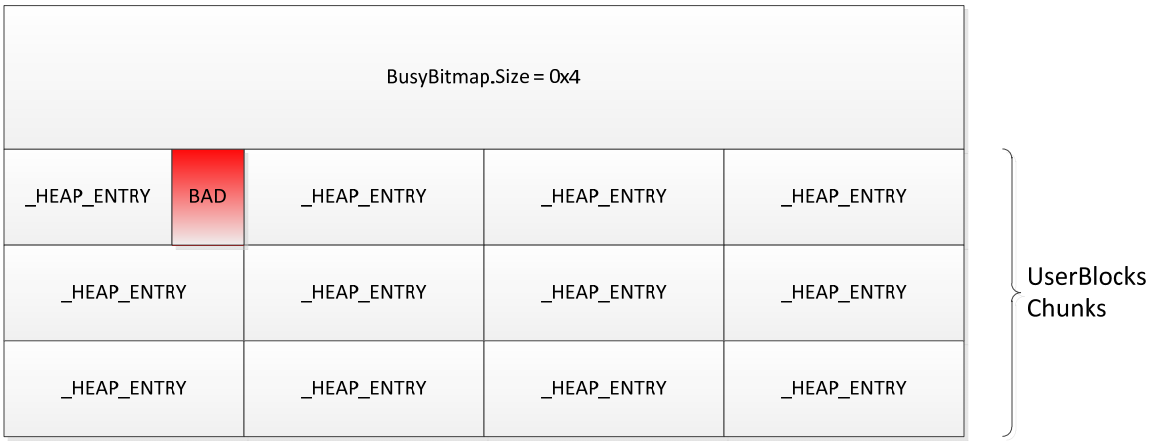
This section will examine tactics that may be used against the Windows 8 heap manager to **aid** in code execution. The Windows 8 heap manager has implemented many anti-exploitation technologies, preventing **any** of the previously **published exploitation techniques** affecting Windows 7 from working. However, new data structures and algorithms have provided small opportunities in Windows 8 to leverage meta-data in a code execution exploit.

Bitmap Flipping 2.0

We saw in the FrontEnd de-allocation section how the **UserBlocks->BusyBitmap** was used to indicate that a chunk managed by the LFH is free. The bitmap was cleared at the index provided by the **_HEAP_ENTRY.PreviousSize** member.

```
int RtlpLowFragHeapFree(_HEAP *Heap, _HEAP_ENTRY *Header)
{
        .
        .
        .
        short BitmapIndex = Header->PreviousSize;

        //Set the chunk as free
        Header->UnusedBytes = 0x80;

        bittestandreset(UserBlocks->BusyBitmap->Buffer, BitmapIndex);

        .
        .
        .
}
```

Also, if you remember the _RTL_BITMAP structure that is located off the base of the _HEAP_USERDATA_HEADER (**UserBlocks**). This provides a very small, but **theoretical**, attack surface if an attacker can corrupt the **PreviousSize** with a value that is beyond the bitmap. For example, if a bitmap has a **size** of **0x4**, but an attacker overwrites the chunk header with a size of **0x20** then a bit will be set to **zero** in a chunk relative to the UserBlocks. If a certain amount of information is known about a chunk or chunks, it may be possible to use this tactic as a way to corrupt sensitive data.



Limitations

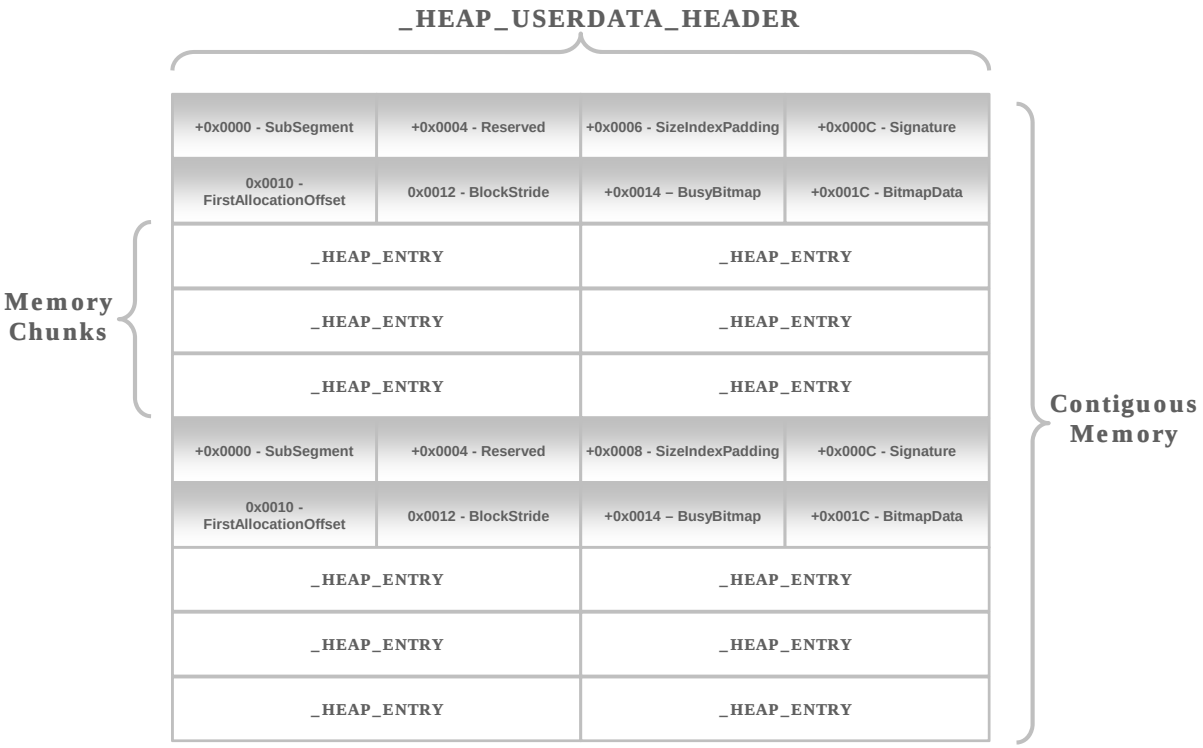
While the zeroing of memory has been previously proved to aid in the exploitation of software vulnerabilities (Moore 2008), this variation is quite limited for a few reasons.

- The UserBlock and corresponding bitmap are acquired from the _HEAP_SUBSEGMENT
- The _HEAP_SUBSEGMENT is derived from the chunk header upon de-allocation
    - SubSegment = *(DWORD)header ^ (header / 8) ^ heap ^ RtlpLFHKey;
- Therefore, a sequential overflow will most likely yield an undesirable result, terminating the processes.
- An overwrite will need to either start after the encoded chunk header or consist of a non-sequential overwrite, such as an off-by-a-few error.

_HEAP_USERDATA_HEADER Attack

The FreeEntryOffset attack **no longer works** in Windows 8 due to the additional information added to the _HEAP_USERDATA_HEADER structure that takes responsibility for locating free chunks. But like the FreeEntryOffset attack, the UserBlocks **header** can also be leveraged to give memory back to the user **outside** the **address** space reserved for UserBlocks.

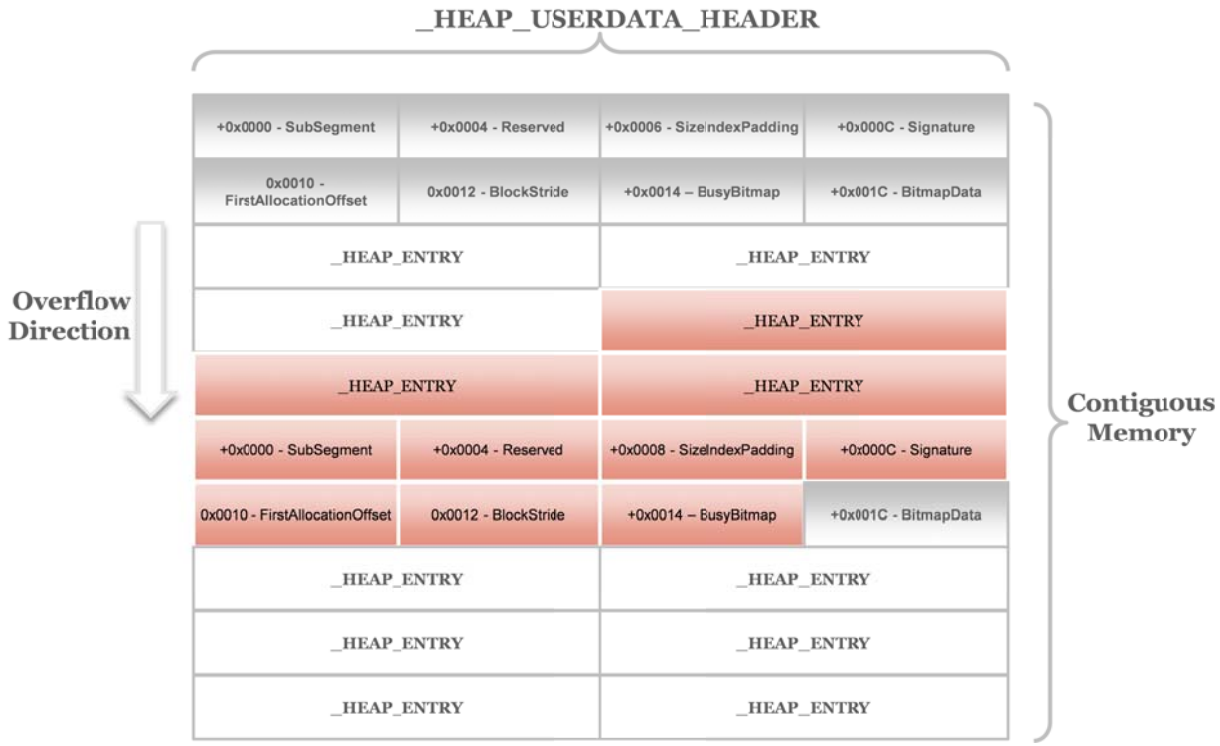
For example, it is possible to get multiple **UserBlocks** containers located **adjacently** to each other in **contiguous** heap memory. The following diagram shows how a _HEAP_USERDATA_HEADER can come **after** chunks from a different UserBlocks container.



Note: The UserBlock containers do NOT need to hold chunks of the same size

If an attacker can position a _HEAP_USERDATA_HEADER **after** a **chunk** that he can **overflowed**, then a **subsequent** allocation can point to **unknown** and potentially **dangerous** memory.

Note: This is much like the FreeEntryOffset attack, but limited to _HEAP_USERDATA_HEADERs.



Once the **FirstAllocationOffset** and **BlockStride** values have been **overwritten**, subsequent allocations made from the corrupted UserBlocks can result in a **semi-arbitrary** memory address being **returned** from the FrontEnd allocator.

```
//If the FirstAllocationOffset and/or the BlockStride are attacker controlled
//semi arbitrary memory will be returned to calling function, potential for code exec
Header = (_HEAP_ENTRY)UserBlocks + UserBlocks->FirstAllocationOffset +
       (NewHint * UserBlocks->BlockStride);
```

Limitations

- An overflowable chunk must reside in front of a _HEAP_USERDATA_HEADER.
- The size of chunks contained by the overflowed _HEAP_USERDATA_HEADER will most likely have to be known.
- The chunk that is to be returned must be FREE (!(Header->UnusedBytes & 0x3F))
- The _RTL_BITMAP structure will need to be taken into account, as invalid bitmap traversal can result in access violations if a bitmap size is too large.
- Most importantly, allocations will need to be kept to a minimum as guard pages are introduced after certain heuristics are triggered (See FrontEnd Allocation section).
    - **TIP**: Stagger the Heap Bucket sizes when priming the heap for the attack

- i.e. Alloc(0x40) x 10; Alloc (0x50) x 0x10, etc

User Land Conclusion

Windows 8 has changed quite a bit from its Windows 7 foundation. We saw that new data structures were added to assist the memory manager with reliable allocation and frees. These new data structures also advertently changed the way certain algorithms work. Instead of stuffing data into preexisting structures to work with the current algorithms, the newest version of Windows created algorithms that reaped the benefits of the new data structures.

No longer are the ListHints used for a dual purpose. New dedicated bitmaps and counters were introduced, to protect the memory manager from having its meta-data abused during an exploitation attempt. The algorithms used to manage heap memory have taken out some complexities while introducing others, but still making the overall allocation and freeing concept much easier to understand.

Microsoft has really taken security to heart with the release of Windows 8. As we've seen in the sections above, this version of Windows probably has more mitigations added than all other versions combined. It appears that all public exploitation techniques for Windows 7 have been addressed. I think this shows that Microsoft is listening to what security researchers have to say and learning how to better protect their customers by using available research as learning tools, instead of only seeing them as attacks on their operating system.

Lastly, we've seen the death of most of the heap meta-data attacks in the Heap Manage since rise to popularity starting back in the early 2000s. But are they officially dead? I think the Exploitation Tactics section proves that while extremely difficult and less abundant, heap meta-data exploitation is still a possibility when writing exploits for the Windows 8 operating system.

That being said, Windows 8 appears to have the most hardened user-land heap to date and will provide major hurdles for attackers relying on currently exploitation techniques. I don't believe the goal is to promise an un-exploitable heap manager, but make it expensive enough to where your average attacker is rendered ineffective.

Kernel Pool Allocator

This major section examines the Windows 8 kernel pool allocator and highlights the differences between the implementation in the Windows 8 Release Preview and Windows 7. Although there are no significant algorithm and structure changes since Windows 7, a number of security improvements have been introduced to address previously presented kernel pool attacks. We begin by providing a brief overview of the kernel pool as well as its key data structures, before moving on to detailing these improvements.

Fundamentals

The Windows kernel as well as third-party drivers commonly allocate memory from the kernel pool allocator. Unlike the user-mode heap, the kernel pool seeks to avoid costly operations and unnecessary page faults as it will greatly impact the performance of the system due to its abundant use in all areas of Windows. In order to service allocations in the fastest and most efficient way possible, the allocator uses several lists from which fragments of pool memory can be retrieved. Knowledge of these lists, their associated data structures, and how they are used is critical in order to understand how the kernel pool operates, and is essential in assessing its security and robustness against pool corruption attacks.

Pool Types

When requesting pool memory from the pool allocator, a driver or system component specifies a pool type. Although many different types are defined, there are really only two basic types of pool memory, the paged pool and the non-paged pool.

In order to conserve resources and memory use, the allocator allows pool memory to be paged out to disk. When subsequently accessing paged out memory, Windows triggers a page fault, pages the data back into memory and recreates the page table entry (PTE) for the specific page. However, if the kernel is already running at IRQ level above DPC/Dispatch (e.g. processing an interrupt), it cannot service the page fault interrupt in a timely manner. For this reason, the kernel also provides non-pagable memory that is guaranteed to be paged in (present) at all times. Because of this requirement, the non-paged pool is a scarce resource, limited by the physical memory in the system.

In a standard uniprocessor system, Windows sets up one (1) non-paged pool and four (4) paged pools that system components and drivers can access. These are accessed through the **nt!PoolVector** and the **nt!ExpPagedPoolDescriptor** arrays respectively. Note that each logged in user also has its own session pool, defined by the session space structure.

Pool Descriptor

In order to manage a given pool, Windows defines what is known as the *pool descriptor*. The pool descriptor defines the properties of the pool itself (such as its type), but more importantly maintains the doubly linked lists of free pool chunks. Pool memory is managed in the order of block size, a unit of 8 bytes on 32-bit systems and 16 bytes on 64-bit systems. The doubly linked lists associated with a pool descriptor maintain free memory of size up to 4080/4064 (x86/x64) bytes, hence a descriptor holds a total of 512 lists on 32-bit or 256 lists on 64-bit. The structure of the pool descriptor (**nt!_POOL_DESCRIPTOR**) on Windows 8 Release Preview (x64) is shown below.

```
kd> dt nt!_POOL_DESCRIPTOR
   +0x000 PoolType        : _POOL_TYPE
   +0x008 PagedLock       : _FAST_MUTEX
   +0x008 NonPagedLock    : Uint8B
   +0x040 RunningAllocs   : Int4B
   +0x044 RunningDeAllocs  : Int4B
   +0x048 TotalBigPages   : Int4B
   +0x04c ThreadsProcessingDeferrals : Int4B
   +0x050 TotalBytes      : Uint8B
   +0x080 PoolIndex       : Uint4B
   +0x0c0 TotalPages      : Int4B
   +0x100 PendingFrees    : _SINGLE_LIST_ENTRY
   +0x108 PendingFreeDepth : Int4B
   +0x140 ListHeads       : [256] _LIST_ENTRY
```

The pool descriptor also manages a singly-linked list of pool chunks waiting to be freed (**PendingFrees**). This is a performance optimization in order reduce the overhead associated with pool locking, needed whenever adding or removing elements on a doubly linked list. If this optimization is enabled (indicated by a flag in **nt!ExpPoolFlags**) and a pool chunk is freed, the free algorithm inserts it to the pending free lists if it's not full (32 entries). When the list is full, the algorithm calls a separate function (**nt!ExDeferredFreePool**) to lock the associated pool descriptor and free all the pool chunks back to their respective doubly-linked free lists.

Although not strictly related to the pool descriptor, the kernel pool may also attempt to use lookaside lists for smaller sized allocations (256 bytes or less), defined per processor in the processor control block (**nt!KPRCB**). Lookaside lists are singly linked, hence perform very well as allocations can be serviced without the need to lock or operate on pool management structures. However, their efficiency comes at a tradeoff, as list consistency cannot be easily validated due to the simplistic nature of the singly linked list. As such, these lists have mostly been abandoned in user-mode and replaced by more robust alternatives.

Pool Header

Another important data structure to the kernel pool, which is more relevant to memory corruption attack scenarios, is the pool header. Each allocated pool chunk is preceded by a pool header structure and notably defines the size of the previous and current chunk, its pool type, an index pointing to an array of pool descriptors, and a pointer to the associated process when dealing with quota charged allocations. On Windows 8 Release Preview (x64), the pool header (**nt!_POOL_HEADER**) is defined as follows.

```
kd> dt nt!_POOL_HEADER
   +0x000 PreviousSize    : Pos 0, 8 Bits
   +0x000 PoolIndex       : Pos 8, 8 Bits
   +0x000 BlockSize       : Pos 16, 8 Bits
   +0x000 PoolType        : Pos 24, 8 Bits
```

```
+0x000 Ulong1          : Uint4B
+0x004 PoolTag         : Uint4B
+0x008 ProcessBilled   : Ptr64 _EPROCESS
+0x008 AllocatorBackTraceIndex : Uint2B
+0x00a PoolTagHash     : Uint2B
```

Unlike the low fragmentation heap in the Windows user-mode heap allocator, the kernel pool divides pool pages into variable sized fragments for use when servicing pool allocations. As such, the pool header holds metadata on the size of a given chunk (**BlockSize**) as well as the size of the previous chunk (**PreviousSize**) in order to allow the allocator to keep track of a chunk's size and merge two blocks of free memory to reduce fragmentation. It also holds metadata needed to determine whether a pool chunk is free (a chunk is marked as busy when its **PoolType** is OR'ed with 2) and to what pool resource it belongs.

The use of unprotected metadata preceding pool allocations, which is critical to both allocation and free operations, have historically allowed for a number of different attacks. Windows 7 introduced safe unlinking in the kernel pool to address the well-known and widely discussed "write-4" attack . However, Windows 7 failed (Mandt 2011) to address several other vectors, which allowed for generic attacks by targeting metadata held by pool allocations. This included basic attacks on lookaside lists, as well as various other issues such as the lack of proper pool index validation, allowing an attacker to coerce the pool algorithms to operate on a user controlled pool descriptor.

In Windows 8, Microsoft has invested a significant effort into furthermore locking down the kernel pool. The following sections aim to highlight both the improvements and changes that were introduced to both neutralize previously discussed attacks as well as offer additional hardening to integrate better with state-of-the-art exploit mitigation technologies such SMEP (SMEP).

Windows 8 Enhancements

This section details the major security enhancements of the Windows 8 kernel pool, used to help mitigate previously presented attacks and make it more difficult for an attacker to exploit kernel pool corruption vulnerabilities.

Non-Executable (NX) Non-Paged Pool

On Windows 7 and prior versions, the non-paged pool is always backed by pages that are read, write, and executable (RWX). Although DEP has allowed Microsoft to implement NX support on kernel memory pages for a long time (and has so in the paged pool case), execution was needed in the non-paged pool for a variety of reasons. The problem in maintaining this design was that it could potentially allow an attacker to store arbitrary code in kernel-mode, and subsequently learn its location using various techniques. An example of this was described in a 2010 HITB Magazine article (Jurczyk) where the author used kernel reserve objects to store fragments of user provided shellcode in kernel-mode. Because these objects had multiple fields controllable by the user, and because the address of kernel objects could be queried from user-mode (using **NtQuerySystemInformation**), it was possible to create multiple objects in order to create a full-fledged shellcode. Windows 8 attempts to address injection of user control code into RWX memory and make it less useful to jump into the non-paged pool by introducing the Non-Executable (NX) non-paged pool.

It is fair to ask why Microsoft didn't choose to introduce the non-executable non-paged pool at an earlier stage. One fairly apparent answer is that there was no need to. Rather than going through the trouble of inserting shellcode into kernel memory (and possibly having to find its location), the attacker could simply put the shellcode in an executable buffer in user-mode, direct execution there, and call it a day. While this works perfectly well on Windows 7 and prior versions of the operating system, Windows 8 takes advantage of the new hardware mitigation introduced in Intel Ivy Bridge CPUs called Supervisor Mode Execution Protection (SMEP). In short, SMEP (or "OS Guard", which appears to be the marketing name) prevents the CPU, while running in privileged mode, from executing pages marked as "User" (indicated by the PTE), and thus effectively hinders execution of user-mode pages. The reason the non-executable non-paged pool made its appearance now is therefore more likely because SMEP could easily be bypassed if all non-paged memory was executable.

The NX non-paged pool is introduced as a new pool type (0x200), hence requires existing code to be updated for compatibility reasons. The **POOL_TYPE** enum in Windows 8 now contains the following definitions.

```
kd> dt nt!_POOL_TYPE
   NonPagedPool = 0n0
   NonPagedPoolExecute = 0n0
   PagedPool = 0n1
   NonPagedPoolMustSucceed = 0n2
   DontUseThisType = 0n3
   NonPagedPoolCacheAligned = 0n4
   PagedPoolCacheAligned = 0n5
   NonPagedPoolCacheAlignedMustS = 0n6
```

```
MaxPoolType = 0n7
NonPagedPoolBase = 0n0
NonPagedPoolBaseMustSucceed = 0n2
NonPagedPoolBaseCacheAligned = 0n4
NonPagedPoolBaseCacheAlignedMustS = 0n6
NonPagedPoolSession = 0n32
PagedPoolSession = 0n33
NonPagedPoolMustSucceedSession = 0n34
DontUseThisTypeSession = 0n35
NonPagedPoolCacheAlignedSession = 0n36
PagedPoolCacheAlignedSession = 0n37
NonPagedPoolCacheAlignedMustSSession = 0n38
NonPagedPoolNx = 0n512
NonPagedPoolNxCacheAligned = 0n516
NonPagedPoolSessionNx = 0n544
```

Most non-paged pool allocations in both the Windows kernel and system drivers such as win32k.sys now use the NX pool type for non-paged allocations. This also includes kernel objects such as the reserve object mentioned initially. Naturally, the NX pool is only relevant as long as DEP is enabled by the system. If DEP is disabled, the kernel sets the 0x800 bit in **nt!ExpPoolFlags** to inform the pool allocator that the NX non-paged pool should not be used.

Windows 8 creates two pool descriptors per non-paged pool, defining both executable and non-executable pool memory. This can be observed by looking at the function responsible for creating the non-paged pool, **nt!InitializePool**.

```
POOL_DESCRIPTOR * Descriptor;

// check if the system has multiple NUMA nodes
if ( KeNumberNodes > 1 )
{
    ExpNumberOfNonPagedPools = KeNumberNodes;

    // limit by pool index maximum
    if ( ExpNumberOfNonPagedPools > 127 )
    {
        ExpNumberOfNonPagedPools = 127;
    }

    // limit by pointer array maximum
    // x86: 16; x64: 64
    if ( ExpNumberOfNonPagedPools > EXP_MAXIMUM_POOL_NODES )
    {
        ExpNumberOfNonPagedPools = EXP_MAXIMUM_POOL_NODES;
    }

    // create two non-paged pools per NUMA node
    for ( idx = 0; idx < ExpNumberOfNonPagedPools; idx++ )
    {
        Descriptor = MmAllocateIndependentPages( sizeof(POOL_DESCRIPTOR) * 2 );
```

```
        if ( !Descriptor )
            return 0;

        ExpNonPagedPoolDescriptor[idx] = Descriptor;
        ExInitializePoolDescriptor(idx, Descriptor, NonPagedPoolNx);
        ExInitializePoolDescriptor(idx, Descriptor + 1, 0);
    }
}

// initialize the default non-paged pool descriptors
ExpTaggedPoolLock = 0;
PoolVector = &NonPagedPoolDescriptor;
ExInitializePoolDescriptor( 0, &NonPagedPoolDescriptor, NonPagedPoolNx );
ExInitializePoolDescriptor( 0, &unk_5D9740, NonPagedPool );
```

The non-executable and executable pool descriptors are located adjacently in memory, hence the kernel only needs to adjust the pointer by the size of a descriptor in order to switch between the two. For NUMA compatible systems with multiple nodes, **nt!ExpNonPagedPoolDescriptor** points to individual pairs of descriptors, whereas on uniprocessor (non-NUMA) systems, the first entry in **nt!PoolVector** points to the pair of non-paged pool descriptors located in the data section of the kernel image. Additionally, Windows 8 also defines lookaside lists for the NX non-paged pool. These are managed by the processor control block (**nt!_KPRCB**), where specifically **PPNxPagedLookasideList** defines the array of NX lookaside lists.

```
kd> dt nt!_KPRCB
...
  +0x670 LockQueue      : [17] _KSPIN_LOCK_QUEUE
  +0x780 PPLookasideList  : [16] _PP_LOOKASIDE_LIST
  +0x880 PPNxPagedLookasideList : [32] _GENERAL_LOOKASIDE_POOL
  +0x1480 PPNPagedLookasideList : [32] _GENERAL_LOOKASIDE_POOL
  +0x2080 PPPagedLookasideList : [32] _GENERAL_LOOKASIDE_POOL
  +0x2c80 PrcbPad20       : Uint8B
```

In order to distinguish between executable and non-executable non-paged pool allocations, the pool allocator does not trust the pool type in the pool header. This could potentially allow an attacker, using a pool corruption vulnerability, to inject pool chunks present in executable memory into lists managing non-executable memory. Instead, the kernel calls **nt!MmIsNonPagedPoolNx** to determine if a chunk is non-executable. This function looks up the page table entry (PTE) or the page directory entry (PDE) and checks the NX bit (0x8000000000000000) as shown below.

```
BOOL
MmIsNonPagedPoolNx( ULONG_PTR va )
{
    PMMPTE pte,pde;

    if ( MmPaeMask == 0 )
```

```
        return TRUE;

    pde = MiGetPdeAddress( va );

    // check no-execute bit in page directory entry (large page)
    if ( pde->u.Hard.NoExecute )
        return TRUE;

    pte = MiGetPteAddress( va );

    // check no-execute bit in page table entry
    if ( pte->u.Hard.NoExecute )
        return TRUE;

    return FALSE;
}
```

Kernel Pool Cookie

One of the ways attacks against pool metadata are mitigated in Windows 8 is by introducing unpredictable data into select locations such that exploitation attempts can be detected at the earliest opportunity. This random piece of data is known as the kernel pool cookie and is essentially a randomized value (combined with various other properties such as the address of the structure it seeks to protect) chosen at runtime by the operating system. Cookies are already used by the kernel in protecting against stack-based buffer overruns, and have for many years played an important role in mitigating exploitation in user-mode. As long as the attacker cannot infer the value of the kernel pool cookie, the system may detect exploitation attempts in scenarios where behavior in either allocation or free algorithms are attempted abused.

As the security of the kernel pool cookie lies in its secrecy, it's important to understand how the cookie is generated. We discuss the seeding of the cookie as well as how it is generated in the following sections.

Gathering Boot Entropy

The way in which the pool cookie is created starts at boot time when the Windows loader (Winload) collects entropy, later used for seeding the kernel provided pseudo random number generator (exposed through **nt!ExGenRandom**). This entropy is passed to the kernel through the loader parameter block (**nt!KeLoaderBlock**), which is initialized in **winload!OslInitializeLoaderBlock** upon running the Windows loader. The loader block initialization function sets up various device nodes including the disk and keyboard controller, before calling **winload!OslGatherEntropy** to gather the boot entropy itself.

The boot entropy gathered by Winload is primarily retrieved from six different sources. These sources are processed by functions in the **winload!OslpEntropySourceGatherFunctions** table, and are as follows.

- **OslpGatherSeedFileEntropy**
  Gathers entropy by looking up the value of the "Seed" registry key (REG_BINARY) in

HKEY_LOCAL_MACHINE\SYSTEM\RNG. This key is 76 bytes in size, whereas the last 64 bytes hold a unique hash used to seed the CryptoAPI PRNG.

- **OslpGatherExternalEntropy**
  Gathers entropy by looking up the value of the "ExternalEntropyCount" registry key (REG_DWORD) in HKEY_LOCAL_MACHINE\SYSTEM\RNG, indicating the number of external entropy sources (such as the TPM). It then uses this value (commonly 2 or 3) to compute a SHA512 hash (64 bytes) in order to produce the actual entropy.
- **OslpGatherTpmEntropy**
  In addition to offering facilities for securely generating and storing cryptographic keys, the Trusted Platform Module (TPM) also features its own (true) random number generator (TRNG). The TPM random number generator consists of a state machine that mixes unpredictable data with the output of a one-way hash function. If a TPM is present in the system, **winload!OslpGatherTpmEntropy** calls **winload!TpmApiGetRandom** to produce 40 bytes of random data.
- **OslpGatherTimeEntropy**
  Queries several performance counters in order to produce 56 bytes of semi-random data. This includes the performance counter (**winload!BlArchGetPerformanceCounter**), a time performance frequency counter (**winload!BlTimeQueryPerformanceCounter**), the current time (**winload!BlGetTime**), as well as the current relative time (**winload!BlTimeGetRelativeTimeEx**).
- **OslpGatherAcpiOem0Entropy**
  Calls **winload!BlUtlGetAcpiTable** to query the OEM0 ACPI table and retrieve 64 bytes of data.
- **OslpGatherRdrandEntropy**
  Intel Ivy Bridge CPUs expose a new pseudo random number generator via the RDRAND instruction. **Winload!OslpGatherRdRandEntropy** checks if the CPU supports this feature and allocates 0x6000 bytes of memory. It then fills this buffer by calling RDRAND repeatedly, generating a random 32-bit value each time. Finally, the function calls **winload!SymCryptSha512** to generate a SHA512 hash (64 bytes) of the buffer, which it uses as the final entropy.

Before querying each of these functions, **winload!OslGatherEntropy** initializes a separate buffer to keep track of the information retrieved. We describe this buffer using the following **BOOT_ENTROPY** data structure.

```
typedef struct _BOOT_ENTROPY {
        DWORD EntropyCount;
        DWORD Unknown;
        ENTROPY_INFO EntropyInfo[7];
        CHAR BootRngData[0x430];       // offset 0x2E0h
} BOOT_ENTROPY;
```

The **BOOT_ENTROPY** structure defines the number of entropy sources (**EntropyCount**) as well as information on each of the queried source (including whether status information indicating if the request was successful), using a separate **ENTROPY_INFO** buffer. We describe this specific structure as follows.

```
typedef struct _ENTROPY_INFO {
        DWORD Id;
        DWORD Unknown2;
        DWORD Unknown3;
        DWORD Unknown4;
        DWORD Code;                 // supplementary to Status
        DWORD Result;
        UINT64  TicksElapsed;       // ticks it took to query to entropy function
        DWORD Length;
        CHAR Data[0x40];            // entropy source data
        DWORD Unknown7;
} ENTROPY_INFO;
```

When gathering the entropy, Winload processes each source in a loop by passing the **ENTROPY_INFO** buffer to a function in the **winload!OslpEntropySourceGatherFunctions** table. We depict this process in the following pseudo code.

```
#define ENTROPY_FUNCTION_COUNT 6

UINT64 tickcount;

RtlZeroMemory( EntropySource, sizeof( BOOT_ENTROPY ) );

EntropySource->EntropyCount = 7

// the mismatch between EntropyCount and ENTROPY_FUNCTION_COUNT is
// intentional as the last entry is reserved (not used)
for ( i = 0; i < ENTROPY_FUNCTION_COUNT; i++ )
{
    tickcount = BlArchGetPerformanceCounter();

    ( OslpEntropySourceGatherFunctions[i] )( HiveIndex, &EntropySource->EntropyInfo[i] );

    EntropySource->TicksElapsed = BlArchGetPerformanceCounter() - tickcount;
}
```

The first argument passed to the entropy source gather functions defines the index to the system hive table entry in the **HiveTable** initialized by Winload (see **winload!OslpLoadSystemHive**). It is used to look up various keys in the registry, used by various gather functions in generating entropy. One such example can be seen in **winload!OslpGatherExternalEntropy**. This function looks up the "ExternalEntropyCount" registry key (REG_DWORD) in \\HKEY_LOCAL_MACHINE\SYSTEM\RNG and uses it to compute a SHA512 hash (64 bytes) to generate the actual entropy.

```
NTSTATUS
OslpGatherExternalEntropy( DWORD HiveIndex , ENTROPY_INFO * EntropyInfo )
{
    NTSTATUS Status;
    DWORD Code, Type;
    PVOID Root, SubKey;
    CHAR Buf[256];

    Code = 2;

    EntropyInfo->Id = 2;
    EntropyInfo->Unknown3 = 0;
    EntropyInfo->Unknown4 = 0;

    Root = OslGetRootCell( HiveIndex );

    Status = OslGetSubkey( HiveIndex, &SubKey, Root, L"RNG" );

    if ( NT_SUCCESS( Status ) )
    {
        Length = 256;

        // retrieve the value of the ExternalEntropyCount registry key
        Status = OslGetValue( HiveIndex,
                              SubKey,
                              L"ExternalEntropyCount",
                              &Type,
                              &Length,
                              &Buf );

        if ( NT_SUCCESS( Status ) )
        {
            // generate a sha512 hash of the registry key value
            SymCryptSha512( &Buf, &EntropyInfo->Data[0], Length );

            EntropyInfo->Length = 0x40;

            Status = STATUS_SUCCESS;

            Code = 4;
        }

    }

    EntropyInfo->Code = Code;
    EntropyInfo->Result = Status;

    return Status;
}
```

Once having queried all functions for the needed entropy, **winload!OslGatherEntropy** proceeds to create a SHA512 hash of all the data chunks held by the processed **ENTROPY_INFO** structures. This hash is again used to seed an internal AES-based random number generator (used by Winload specifically)

which is subsequently used to generate 0x430 bytes of random data (**BootRngData**). This data constitutes the actual boot entropy, later referenced by ntoskrnl through the loader parameter block.

```
CHAR Hash[64];
NTSTATUS Status;

Status = STATUS_SUCCESSFUL;

SymCryptSha512Init( &ShaInit );

for ( i = 0; i < 7; i++ )
{
    SymCryptSha512Append( &EntropySource->EntropyInfo[i].Data[0],
                          &ShaInit,
                          EntropySource->EntropyInfo[i].Length );
}

// generate a sha512 hash of the collected entropy
SymCryptSha512Result( &ShaInit, &Hash );

if ( SymCryptRngAesInstantiate( &RngAesInit, &Hash ) )
{
    Status = STATUS_UNSUCCESSFUL;
}
else
{
    SymCryptRngAesGenerate( 0x10, &RngAesInit, &Stack );
    SymCryptRngAesGenerate( 0x30, &RngAesInit, &EntropySource->BootRngData[0] );
    SymCryptRngAesGenerate( 0x400, &RngAesInit, &EntropySource->BootRngData[0x30] );
    SymCryptRngAesUninstantiate( &RngAesInit );
}

// clear the hash from memory
for ( i = 0; i < 64; i++ )
{
    Hash[i] = 0;
}

return Status;
```

When Winload calls ntoskrnl on boot, it passes it the loader parameter block data structure (**nt!_LOADER_PARAMETER_BLOCK**) containing the boot entropy as well as all the information necessary to initialize the kernel. This includes the system and boot partition paths, a pointer to a table describing the physical memory of the system, a pointer to the in-memory HARDWARE and SOFTWARE registry hives and so on.

The kernel performs initialization in a two-phase process, phase 0 and phase 1. During phase 0, it creates rudimentary structures that allow phase 1 to be invoked and initializes each processor, as well as internal lists and other data structures that CPUs share. Before completing the phase 0 initialization routines for the executive (**nt!ExpSystemInitPhase0**), the kernel calls **nt!ExpRngInitializeSystem** to initialize its own random number generator.

Random Number Generator

The pseudo random number generator exposed by the Windows 8 kernel is based on the Lagged Fibonacci Generator (LFG) and is seeded by the entropy information provided in the loader parameter block (**nt!KeLoaderBlock**). It is not only used in the process of generating the pool cookie, but also for a variety of other purposes such as for image base randomization, heap encoding, top-down/bottom-up allocation randomization, PEB randomization, and stack cookie generation.

The random number generator is initialized (**nt!ExpRngInitializeSystem**) by first populating **nt!ExpLFGRngState** with the boot entropy gathered by Winload. As the RNG does not use all the provided entropy, it also copies unused data into **nt!ExpRemainingLeftoverBootRngData**. After this, the function proceeds to generate the first random value (part of the kernel GS cookie) by calling **nt!ExGenRandom**. This function permutes the LFG RNG state using an additive LFG algorithm with parameters j=24 and k=55, and returns a 32-bit value. The kernel may also request to use the leftover boot RNG data (if there is any remaining) for returning random values directly by passing **ExGenRandom** a one (1) as its first arguments.

Pool Cookie Generation

In order to generate the pool cookie (**nt!ExpPoolQuotaCookie**), the kernel makes use of its random number generator as well as a number of other variables to further improve its randomness. The cookie is defined upon initializing the first non-paged kernel pool, in **nt!InitializePool** (called by **nt!MmInitNucleus**), which is part of the phase 0 kernel initialization process. This function first calls **nt!KeQuerySystemTime** to retrieve the current system time as a **LARGE_INTEGER**, after which it gets the current tick count through the RDTSC instruction. These values (time split into two 32-bit values) then become XOR'ed together, and are furthermore XOR'ed with both **KeSystemCalls** and **InterruptTime** from the processor control block (**nt!_KPRCB**). **KeSystemCalls** is a counter holding the current number of invoked system calls while **InterruptTime** is the time spent servicing interrupts. Finally, these XOR'ed values become XOR'ed once more with a pseudo random value returned by **nt!ExGenRandom**. In the reasonably unlikely event that the final value should be 0, the pool cookie is set to 1. Otherwise, the final value is used as the pool cookie.

```
ULONG_PTR Value;
KPRCB * Prcb = KeGetCurrentPrcb( );
LARGE_INTEGER Time;

KeQuerySystemTime( &Time );

Value = __rdtsc() ^               // tick count
       Prcb->KeSystemCalls ^      // number of system calls
       Prcb->InterruptTime ^      // interrupt time
       Time.HighPart ^            // current system time
       Time.LowPart ^
       ExGenRandom(0);            // pseudo random number

ExpPoolQuotaCookie = (Value) ? Value : 1;
```

Attack Mitigations

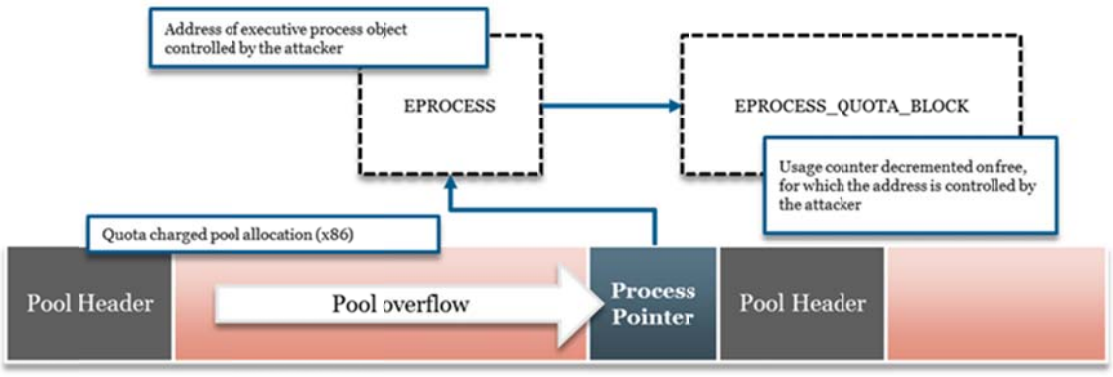
In this section we show how Windows 8 addresses the various attacks presented previously on the Windows 7 kernel pool (Mandt 2011). In each of the below subsections, we briefly detail how these attacks were performed in Windows 7 (and prior versions of Windows) and then show how they are mitigated by the changes introduced in Windows 8.

Process Pointer Encoding

A driver or system component may request pool allocations to be quota charged against the current process by calling **nt!ExAllocatePoolWithQuotaTag**. Internally, the pool allocator associates a pool allocation with a process by storing a pointer to the process object. On x86, the kernel increases the number of bytes requested by four in order to store the process pointer in the last four bytes of the pool allocation. On x64, the process pointer is stored in the last eight bytes of the pool header as the **ProcessBilled** pointer.

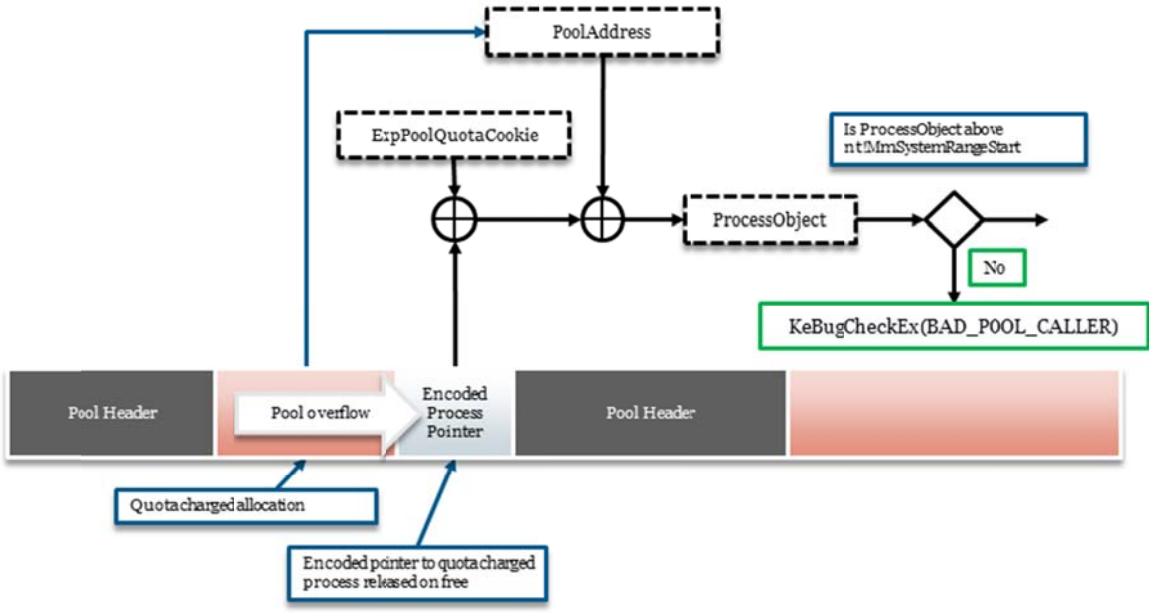
Upon freeing a quota charged allocation, the pool allocator returns the quota to the associated process. This is performed by looking up the process object pointer and by locating the pointer to the associated quota block (**nt!_EPROCESS_QUOTA_BLOCK**). This opaque structure holds the actual quota information including a value defining the amount of quota used. When a free occurs, the allocator decrements this value according to the size of the pool allocation.

Because Windows 7 and former operating system versions do not protect the process pointer, an attacker could overwrite it using a memory corruption vulnerability in order to decrement arbitrary kernel memory. Specifically, the attacker could set the process pointer to a fake process object data structure (e.g. created in user-mode) in order to control the address of the quota block structure where the decrement occurs. Moreover, on x86 there is no need to corrupt adjacent allocations, as the process pointer immediately follows the pool data. The diagram below illustrates this attack on x86 systems. Note that on x64, the attacker must overflow into the next allocation in order to reach the **ProcessBilled** pointer held by the pool header.



Windows 8 addresses the process pointer attack by XOR encoding the process pointer itself. The process pointer is first XOR'ed with the pool cookie (**nt!ExpPoolQuotaCookie**), followed by the address of the

affected pool allocation (at the beginning of the pool header). When a quota charged allocation is freed, the kernel uses the pool cookie and pool address to decode the stored pointer, and subsequently validates it by making sure it points into kernel address space (above **nt!MmSystemRangeStart**).



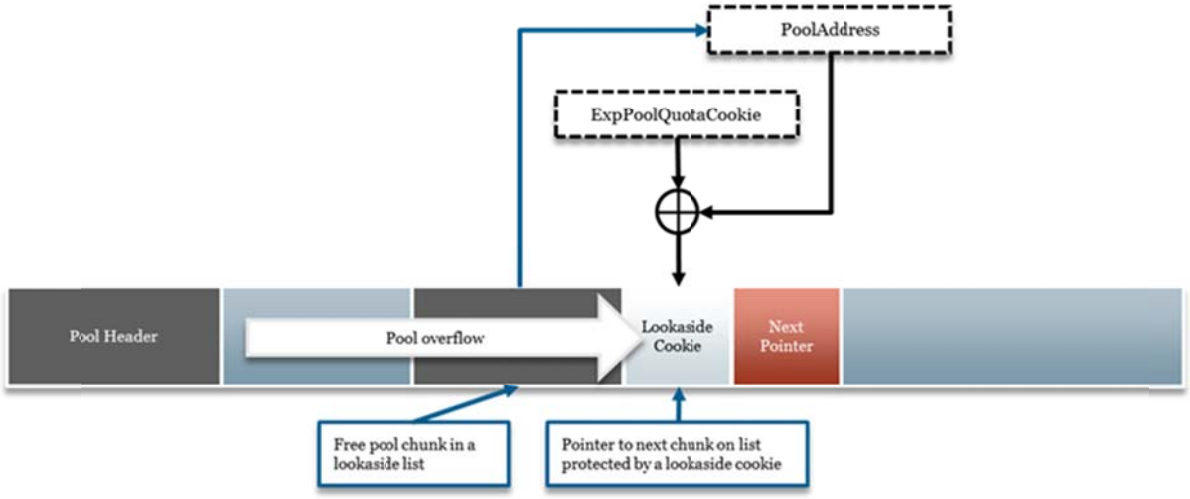
Lookaside Cookie

Due to the abundant use of the kernel pool, lookside lists play a key role in making sure the kernel pool performs well. Because lookaside lists are singly-linked and do not require locking of the pool descriptor, they can benefit from highly optimized CPU instructions such as the atomic compare and exchange used in adding (push) and removing (pop) elements from a list. However, unlike doubly-linked lists, there is no easy way of verifying the integrity of a singly-linked list. This has historically lead to a number of attacks and is part of the reason why these lists are mostly abandoned in user-mode.

In Windows 7 and former operating systems, an attacker could overwrite the next pointer held by a freed pool chunk on a lookaside list in order to control address of the next free pool chunk. The attacker could then force subsequent allocations (e.g. by creating objects of the same size) until the pool allocator services the pointer to memory controlled by the attacker. This would then allow the attacker to control the contents of the memory used by the kernel, hence could be used to extend the lookaside pointer overwrite into a more useful exploitation primitive such as an arbitrary kernel memory write.

Rather than getting rid of lookaside lists altogether, Windows 8 protects each lookaside list pointer using a randomized value, derived from the kernel pool cookie. This value is computed by taking the pool cookie and XOR encoding it with the address of the affected pool chunk (from the pool header). Although the pool header is already full (on 32-bit) and remains unchanged on Windows 8, each pool chunk always reserves space for the **LIST_ENTRY** structure, used to chain elements on a doubly linked free list. As the **LIST_ENTRY** structure contains two pointers, whereas elements on the singly linked

lookaside list only contain one, the cookie can be stored directly in front of the lookaside list next pointer. On x64, the cookie is stored in place of the **ProcessBilled** pointer in the pool header, as this pointer is not in use when an allocation is already free.



Pool cookies are also used to protect entries on the pending frees list, and these cookies are validated in the same way as lookaside lists upon processing the pending frees list (see **nt!ExDeferredFreePool**). However, it should be noted that not all singly-linked lists are protected by cookies. This includes the pool page lookaside lists as well as dedicated (task specific) lookside lists such as those that use **nt!ExAllocateFromNPagedLookasideList** and **nt!ExAllocateFromPagedLookasideList**.

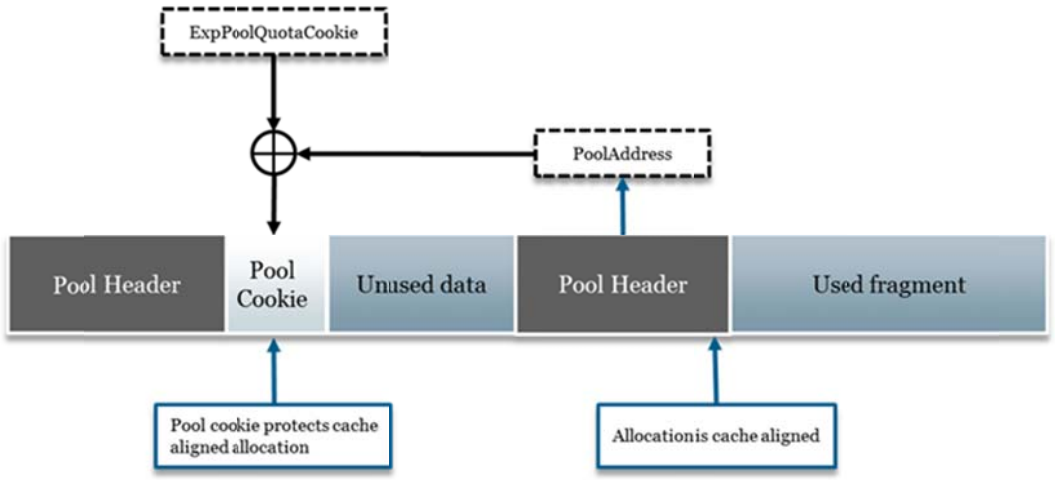
Cache Aligned Allocation Cookie

In order to improve performance and reduce the number of cache lines hit during a memory operation, pool allocations can be requested to be aligned on processor cache boundaries. Although the MSDN documentation states that cache aligned pool allocations are for internal use only, any kernel component or third-party driver can request them by choosing a **CacheAligned** pool type (4) such as **NonPagedPoolCacheAligned**. When requested, the pool allocator ensures that a suitable cache aligned address is found by rounding the number of bytes requested up to the nearest cache line size, plus the size of the cache line. The CPU cache line size is defined in **nt!ExpCacheLineSize** and is typically 64 bytes.

Cache aligned allocations greatly favor performance over space usage. As an example, 32-bit systems that request 0x40 bytes of cache aligned memory typically end up allocating 0xC0 bytes to make sure that a fragment of the requested size is found on a cache aligned boundary. As the allocator does not bother with returning the unused bytes, Windows 8 attempts to mitigate exploitation attempts by inserting a cookie in front of the cache aligned allocation.

The use of the cache aligned allocation cookie depends on the address returned by the free lists and if enough space is available in front of the pool fragment used. If a system component requests a cache aligned pool allocation and the returned chunk is already on a cache aligned boundary, the allocator masks away the **CacheAligned** pool type (4) from the pool header of the affected allocation and returns

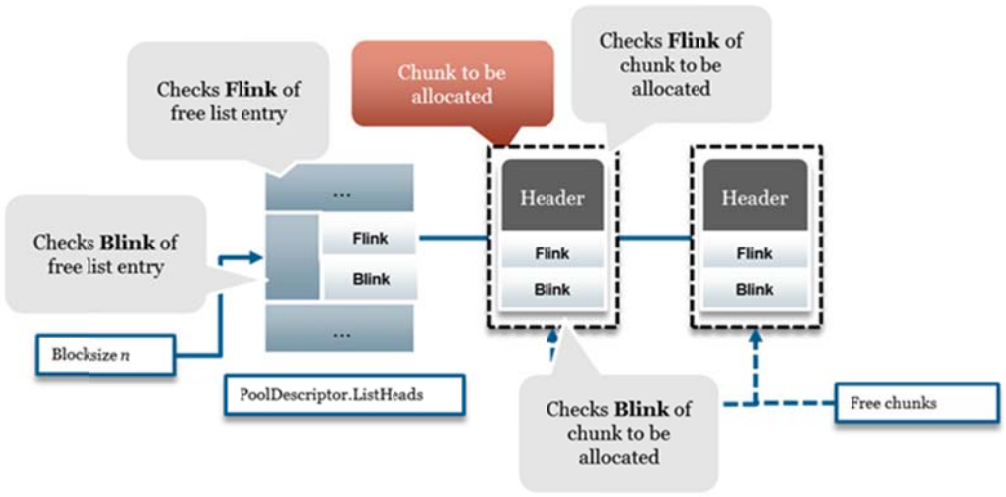
immediately. Although the allocator increases the requested size, it leaves the exceeding bytes unused. However, if an unaligned chunk is returned, the allocator adjusts the address up to the nearest cache aligned boundary. In this particular case, the returned cache aligned chunk retains the **CacheAligned** pool type if the skipped fragment of bytes is large enough (more than a single block size) to hold a cookie. This cookie is stored in a separate pool chunk and computed by XOR encoding the address of the used (cache aligned) chunk with the pool cookie generated by the kernel (**nt!ExpPoolQuotaCookie**). Thus, the free algorithm checks for the **CacheAligned** pool type in order to determine whether a cache aligned allocation cookie needs to be verified.



Safe (Un)linking

Safe unlinking was introduced in the Windows 7 kernel pool to address attacks (Kortchinsky) on **LIST_ENTRY** structures used by doubly linked lists. If an attacker was able to corrupt the forward and backward pointers held by the structure, unlinking the chunk from a linked list would result in a situation where an attacker controlled value was written to an attacker controlled location, commonly known as a "write-4" (or "write-8" on x64). However, the linked list validation performed by Windows 7 was not perfect. Specifically, safe unlinking could be circumvented in specific situations (see the List Entry Flink attack presented in (Mandt)) and the kernel pool also did not perform any validation when linking chunks into a list.

Windows 8 significantly improves linked list validation over Windows 7 and performs both safe linking and unlinking. Notably, when allocating memory, the pool allocator validates both the Flink and Blink of both the descriptor **LIST_ENTRY** as well as well as the one held by the chunk to be allocated. This effectively neutralizes the Windows 7 attack on safe unlinking in which the Flink in the head of a list (held by the pool descriptor) wasn't properly validated. The Windows 8 pool allocator also checks list consistency before linking in unused fragments of pool memory, commonly encountered when a larger chunk than the size requested is returned by a linked list.



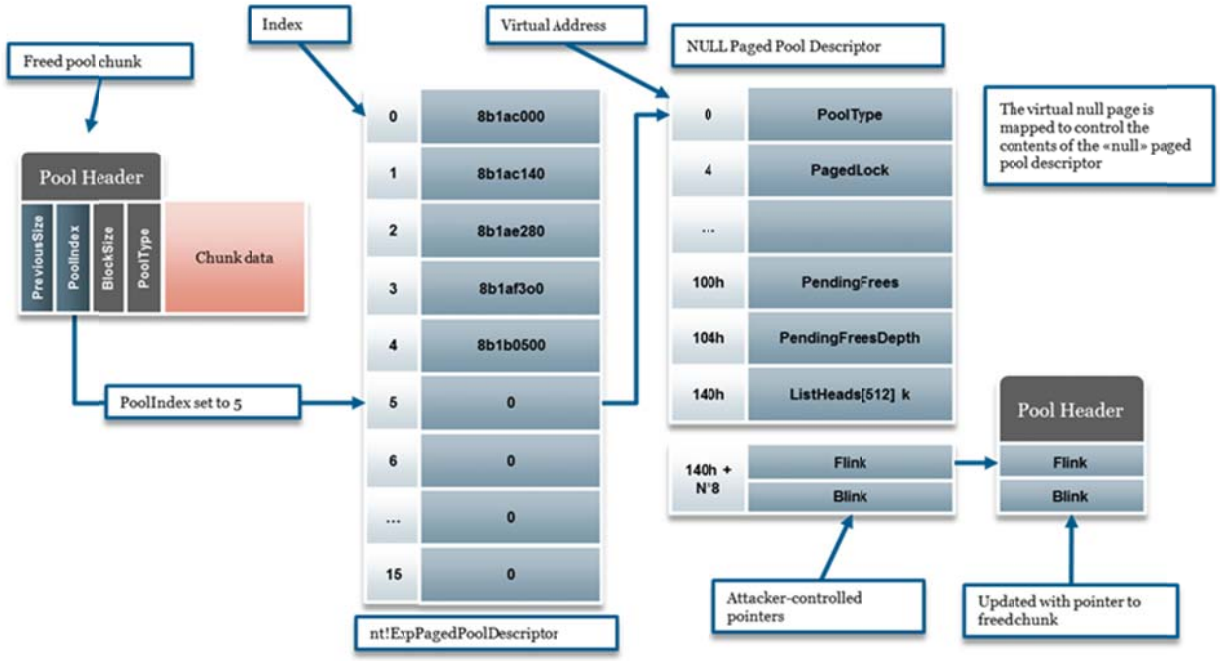
The reason for the improved linked list validation and also why there are cases where pointers are validated twice is because the Windows 8 kernel pool makes use of a new type of security assertion (Ionescu) that is recognized in assembly by the new int29h interrupt handler, calling **KiRaiseSecurityCheckFailure**. As long as **NO_KERNEL_LIST_ENTRY_CHECKS** remain undefined, **LIST_ENTRY** macros in Windows 8 automatically add the line *RtlpCheckListEntry(Entry);* to verify the linked list before any operation takes place. This makes list validation transparent to the programmer as the necessary checks are introduced upon compilation.

```
FORCEINLINE
VOID
RtlpCheckListEntry(
    _In_ PLIST_ENTRY Entry
    )
{
    if ((((Entry->Flink)->Blink) != Entry) || (((Entry->Blink)->Flink) != Entry))
    {
        FatalListEntryError(
            (PVOID)(Entry),
            (PVOID)((Entry->Flink)->Blink),
            (PVOID)((Entry->Blink)->Flink));
    }
}
```

PoolIndex Validation

Upon freeing a pool allocation, the free algorithm uses the pool type as well as the pool index defined in the pool header to determine to which pool descriptor the allocation should be returned. The pool index is used as an array index into a pool descriptor array (holding pointers to the pool descriptor structures themselves) which in the most common case will either be **nt!ExpPagedPoolDescriptor** or **nt!ExpNonPagedPoolDescriptor** if there are more than 1 non-paged pools defined.

As Windows 7 doesn't validate the pool index upon looking up the pool descriptor, an attacker could reference an out-of-bounds entry in the pool descriptor pointer array. For instance, as the paged pool descriptor array typically holds 4 pointers, the attacker could use a memory corruption vulnerability to set the pool index of a pool chunk to 5. Upon freeing the affected allocation, this would cause the kernel to dereference the null pointer immediately following the pool descriptor pointers. Hence, by mapping the null-page an attacker could fully control the pool descriptor data structure (including its free lists) to which the freed chunk is returned. Furthermore, as the attacker operates on a pool descriptor which is not really managed by the system, there are no issues concerning contention nor need for cleaning up management structures post exploitation.



Windows 8 addresses the pool index attack using a very simple fix. Whenever a pool chunk is freed, its pool index is validated to ensure that it is within bounds of the associated pool descriptor array. For paged pool allocations, the allocator checks if the pool index is less than the number of paged pools (**nt!ExpNumberOfPagedPools**). The pool index is also verified upon block allocation from the doubly linked free lists by comparing it to the index used to retrieve the pool descriptor initially. Moreover, Windows 8 prevents user applications from mapping the null page (as long process is not a VDM process), hence mitigates the PoolIndex attack in multiple ways.

Summary

In summary, the Windows 8 kernel pool addresses the shortcomings identified in prior versions of the operating system, both in terms of robustness and security. Although the pool header remains unprotected to date, generic attacks on pool metadata have become considerably more difficult due to the extensive validation performed by the pool allocator. The following table summarizes the security enhancements and mitigations introduced in the last iterations of Windows, up until the Windows 8 Release Preview.

Primitive	Windows Vista	Windows 7	Windows 8 (RP)
Safe Unlinking	☒	☑	☑
Safe Linking	☒	☒	☑
Pool Cookie	☒	☒	☑
Lookaside Chunks			☑
Lookaside Pages			☒
Pending Frees List			☑
Cache Aligned Allocations			☑
PoolIndex Validation	☒	☒	☑
Pointer Encoding	☒	☒	☑
NX Non-Paged Pool	☒	☒	☑

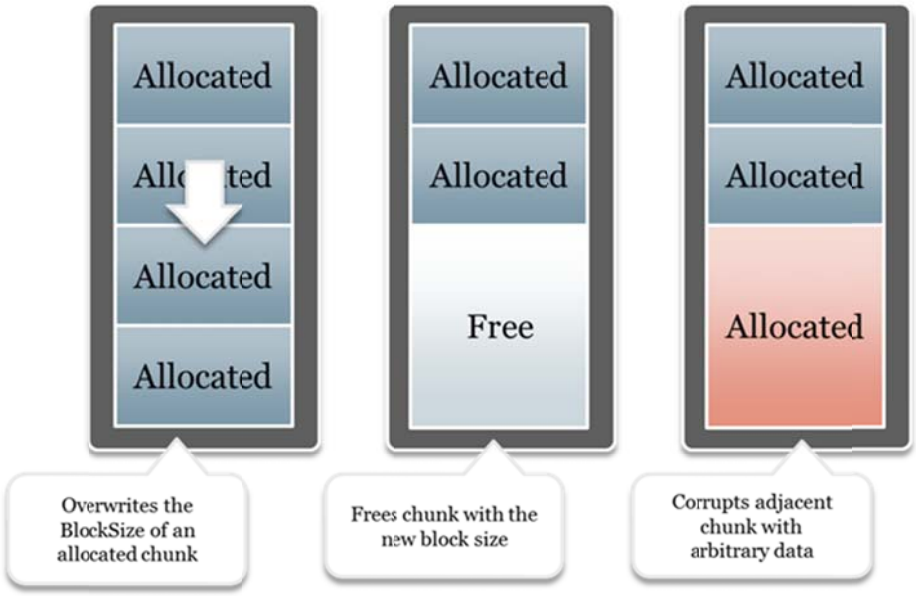
Block Size Attacks

Although the Windows 8 kernel pool addresses the attacks presented previously, it does not prevent an attacker from manipulating fields in the pool header using a pool corruption vulnerability. While the extensive validation performed by Windows 8 goes a long way, some fields can be hard to validate properly due to their lack of dependencies. This is especially true in the case of determining a chunk's size, as the pool allocator relies completely on the size information held by the pool header. In this section, we describe two attacks on block size values where an attacker may extend a limited (both in length and data written) corruption into an n-byte arbitrary data corruption.

Block Size Attack

As mentioned in the initial discussion, the pool header of a pool chunk holds two size values, the block size (**BlockSize**) and the previous size (**PreviousSize**). These fields are used by the allocator to determine the size of a given pool chunk, as well as for locating adjacently positioned pool chunks. The block size values are also used to perform rudimentary validation upon free. Specifically, **ExFreePoolWithTag** checks if the block size of the freed chunk matches the previous size of the chunk following it. The exception to this rule is when the freed chunk fills the rest of the page, as chunks at the start of a page always have their previous size set to null (there are no cross-page relationships for small allocations and therefore no guarantee that the next page is in use).

When a pool chunk is freed, it is put on a free list or lookaside list based on its block size. Thus, given a pool corruption vulnerability, an attacker can overwrite the block size in order to place it in an arbitrary free list. At this point, there are two scenarios to consider. The attacker could set the block size to a value smaller than the original value. However, this would be of little use as it would not extend the corruption, and creating an embedded pool header would have little or no benefit due to the pool header checks present. On the other hand, if the attacker sets the block size to a larger value, the corruption could be extended into adjacent pool chunks. Although the allocator performs the **BlockSize**/**PreviousSize** check on free, setting the block size to fill the rest of the page of the page avoids the check altogether. The attacker could then reallocate the freed allocation using a string or some other controllable allocation in order to fully control the contents of the bordering pool chunk(s).



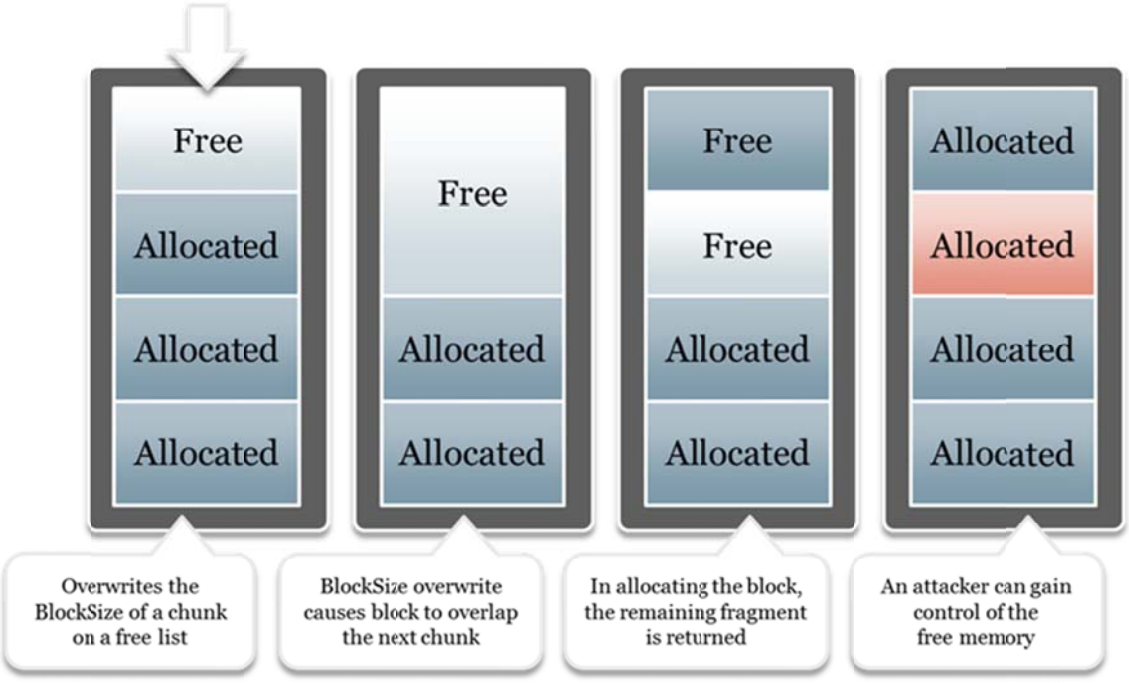
As there is no simple way for the allocator to verify the block size other than looking at the surrounding chunks, it appears to be somewhat difficult to address block size attacks without using some form of encoding on the block size information or by grouping allocations of the same size together in a approach similar to that used by the low fragmentation heap (Valasek). The practicality of the block size attack, as well as any attack dealing with targeting pool allocations in a specific state, also depends on the attacker's ability to sufficiently manipulate and control the state of the kernel pool. For instance, one of the challenging aspects of this attack is to find the block size value needed to fill the remaining fragment of a pool page. This essentially requires the attacker to selectively allocate and free data in order to obtain a reasonable probability of succeeding.

Split Fragment Attack

When requesting a pool chunk (not larger than 4080 bytes or 4064 bytes on x64) and lookaside lists cannot be used, the allocator scans the doubly linked free lists until a suitable chunk is found. If the chunk returned is larger than requested, the allocator splits the chunk and returns the unused fragment back to the free lists. The part of the chunk that is split (front or back) depends on the locality of the chunk returned, which is designed to reduce fragmentation. If the chunk is at the beginning of a page, the front of the chunk is returned to the caller while the remaining part of the chunk is returned back to the allocator. If, on the other hand, the chunk is not at the beginning of a page (say, some place in the middle), the end of the chunk is returned to the caller while the front of the chunk is returned back to the allocator.

In the process of retrieving a pool chunk from a doubly linked free list, there's a good amount of sanity checking. The allocator validates both the Flink and Blink of the chunk to be allocated, as well as the Flink and Blink of the head of the free list. It also validates the pool index for the allocated chunk to ensure it is from the expected pool descriptor. However, because there's no validation on the block size, an attacker could use a memory corruption vulnerability to trigger a block split when in fact the

allocated block is of the requested size. If the block size is set to a larger value, the remaining bytes are returned back to the allocator, hence the attacker can potentially free fragments of in use-memory.



In the above example, the attacker has sprayed allocations of the same size (e.g. executive objects) across multiple pages. By selectively freeing some of these allocations and triggering a pool corruption vulnerability, the attacker could overwrite the block size of a free chunk at the start of a page and double its size. Upon requesting this memory using something controllable like a string, the allocator splits the allocation once returned by the free list, and returns the top part of the chunk, while returning the remaining part back to the free lists. At this point, the allocator have freed a chunk that was already in use, hence have created a use-after-free like situation where the attacker can reallocate the freed memory in order to gain full control of the affected object.

The benefit from an attacker's perspective of the split fragment attack over the block size attack is that chunk positioning is less of an issue as the splitting process makes sure that the affected pool chunk headers are updated correctly. However, because the kernel still references the memory freed (e.g. in the object manager if a kernel object was targeted) in creating the split fragment, there may be a risk of collateral damage (such as double frees) unless precautionary steps are taken.

Kernel Land Conclusion

The Windows 8 kernel pool improves in many areas over previous versions of Windows and raises the bar for exploitation once again. Although there are no significant changes to its algorithms and structures, the array of security improvements now make generic kernel pool attacks somewhat a lost art of the past. Specifically, the addition of proper safe linking and unlinking, and the use of randomized cookies to encode and protect pointers prevent an attacker from targeting metadata, used to carry out simple, yet highly effective kernel pool attacks. However, as the pool header remains unprotected, there may still be situations where an attacker can target header data such as block size values in order to make less exploitable vulnerabilities somewhat more useful. Although such attacks require an attacker to manipulate the kernel pool with a high degree of control, the allocator possesses a high degree of determinism due to its continued use of lookaside lists and bias towards efficiency. That said, the increased difficulty and skillset required in reliably exploiting pool corruption vulnerabilities in Windows 8, suggests that these types of attacks will be fewer and farther between.

Thanks

We'd like to thank the following people for their help.

- Jon Larimer (@shydemeanor)
- Dan Rosenberg (@djrbliss)
- Mark Dowd (@mdowd)

Bibliography

Jurczyk, Mateusz 'j00ru' - Reserve Objects in Windows 7 (Hack in the Box Magazine)

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

Ionescu, Alex – Int 0x29
http://www.alex-ionescu.com/?p=69

Kortchinsky, Kostya – Real World Kernel Pool Exploitation
http://sebug.net/paper/Meeting-Documents/syscanhk/KernelPool.pdf

Mandt, Tarjei. 2011, "Modern Kernel Pool Exploitation"
http://www.mista.nu/research/kernelpool_infiltrate2011.pdf

Moore, Brett, 2008 "Heaps about Heaps"
http://www.insomniasec.com/publications/Heaps_About_Heaps.ppt

Phrack 68 "The Art of Exploitation: MS IIS 7.5 Remote Heap Overflow"
http://www.phrack.org/issues.html?issue=68&id=12#article

SMP, "Symmetric multiprocessing"
http://en.wikipedia.org/wiki/Symmetric_multiprocessing

Valasek, Chris. 2010, "Understanding the Low Fragmentation Heap"
http://illmatics.com/Understanding_the_LFH.pdf
http://illmatics.com/Understanding_the_LFH_Slides.pdf

Varghese George, Tom Piazza, Hong Jiang - Technology Insight: Intel Next Generation Microarchitecture Codenamed Ivy Bridge
http://www.intel.com/idf/library/pdf/sf_2011/SF11_SPCS005_101F.pdf