BLOG

WEDNESDAY, AUGUST 14, 2013

UNDERSTANDING THE WINDOWS ALLOCATOR: A REDUX

If you are interested in Windows heap allocator behaviour, and you haven't already read Chris Valasek's papers Understanding the LFH and Windows 8 Heap Internals (credit also due to Tarjei Mandt), I strongly suggest you do so. There are few more detailed and accurate accounts of the modern (post-XP) allocators' behaviour. However, as I was working on our crashdump analytic software (codenamed Major Myer), I found that there is some additional depth of functionality not addressed in these two papers; I present it here.

First, some prerequisites. If you haven't skimmed the papers above, go do so now. You may also be wondering about a few fine points regarding how it all fits together, and perhaps some definitions. After that, I will break apart the illusion of 1-to-1 hierarchical linkage in the LFH structures, and point out a couple of unmentioned semantics (after the break; this is going to be long). Any exploits which may exist with regard to these structures I will leave as an exercise.

LISTS

_LIST_ENTRY is a particularly opaque struct, but it is foundational to understanding how most Windows kernel data fits together. It is effectively what its definition implies: a forward link pointer, and a backward link pointer, with no data. This struct is designed for inclusion in other structs, which is why it has no data members of its own. You have likely noticed that, in the heap structs, it appears at the beginning occasionally, but sometimes in the middle. This is because _LIST_ENTRY structures are included in a class to make them part of circular fixed-sentinel doubly-linked lists.

By convention, when a _LIST_ENTRY appears at the top of a struct, that struct is the data type of the linked list (with one notable exception which will be covered later on). This is so that the pointers in the list are actually pointers to the appropriate struct, theoretically minimizing offset computation requirements. However, when a _LIST_ENTRY appears in the middle of a struct, it is best understood as a pointer to the beginning and end nodes of a linked list. Unfortunately this idiom of Microsoft's results in the data type of the linked list being redacted from the symbols (and it not being a syntactic requirement), but it's usually fairly easy to guess what the data type might be based on the name of the _LIST_ENTRY member.

This is complicated slightly more, however: these pointer-style _LIST_ENTRY members are included in the circular linked list, but are not validly dereferenceable as the list's data type. Thus, they act in effect as sentinel nodes as well as data pointers, with the drawback that

SEARCH

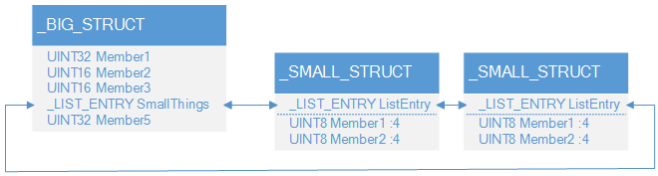
search by **AUTHOR**

Falcon Momot (2)
Ryan O'Neill (1)
Ron Bowes (1)
Joel Voss (2)
Steve Manzuik (3)
James Arlen (1)
Paul Brodeur (2)
Mikhail Davidov (1)
David Kane-Perry (3)
Frank Heidt (2)

search by **TAG**

802.11 (1)
eap (1)
enterprise (1)
hacking (1)
Microsoft (1)
mobile (1)
peap (1)
radius (1)
wifi (1)
Windows (1)
wireless (1)
Fraud (1)
NFC (1)
PCI (1)
RFID (1)
Risk (2)
Commentary (1)
DBIR (1)
Risk Advisory Services (1)
Android (1)
Java (1)

they cannot be dereferenced, and nothing besides context identifies them as sentinels. The correct way to parse such a list, then, is to remember the offset of the _LIST_ENTRY in the "parent" structure, and halt traversal when it is encountered. Generically, it looks like this:



There is also a second type of pointer-style nodes: hints. These are nodes which are not included in the list, but have forward and back links pointing to some node in the list, and are used in the ListHints array among other places. It is important to be aware of these when validating the integrity of a list, and when traversing it (they are not sentinel nodes).

That's basically the same as any other linked list, but heed the typing and the positions of the _LIST_ENTRY member. The sentinel address is that of the SmallThings member of _BIG_STRUCT. With that out of the way, it should be considerably easier to read and understand Valasek's diagrams and structures, as well as the public symbols.

BACKEND FREELISTS

The **FreeList** is one instance of atypical _LIST_ENTRY use. It is quite simply a list of nodes which refer to all the free chunks in the backend allocator. There is one per heap (and not one per segment, or one per size). A **ListHint** is a pointer into the freelist (actually a special _LIST_ENTRY which does not form part of the linked list but only points into it), and can essentially be understood as a pointer to a sublist of the heap's FreeList for a particular chunk size; the sublist is terminated either by the start of the next sublist or the end of the list. ListHint structures are stored in an array pointed to by the ListHints member of the **_HEAP_LIST_LOOKUP** ("lookup") structure (this element is more properly a pointer to an array than a double-pointer); though the array's size is stored in ArraySize (with a caveat, which we will come to), it is always 0×80 for the first lookup. If there is no chunk in the FreeList of an appropriate size for the ListHint, the ListHint's forward link will be null. Note that the type of a ListHint is _LIST_ENTRY, but the Blink member is not a backlink for a circular linked list; it is (as Valasek identifies) a triple-use member indicating whether the LFH is enabled (the least significant bit), and depending upon that either a counter used internally, or a pointer to the appropriate LFH _HEAP_BUCKET.

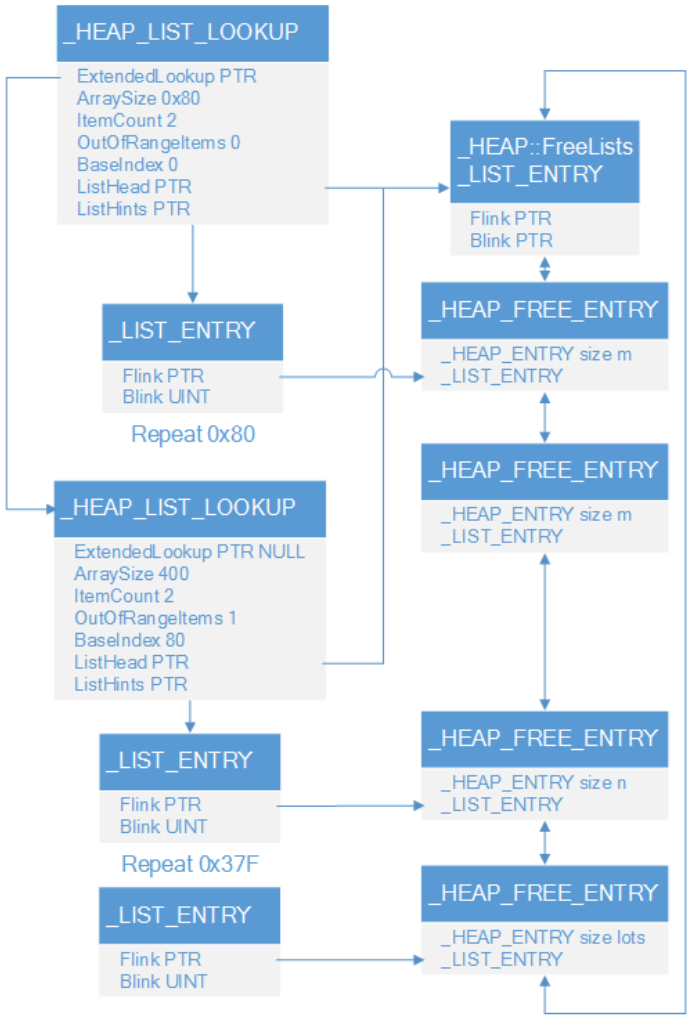
It is worth noting that the FreeList node used to reference a free chunk is stored at what would otherwise be the first user data address of the free chunk it refers to, similar to how ptmalloc stores the location of the previous and next free chunks. The header of the free chunk can therefore be determined using basic arithmetic.

In actuality, it is not necessarily the case that chunks of sizes greater than 0x7E are all stored in the **0x7F**th ListHint. Rather, they are stored in the final ListHint of the final _HEAP_LIST_LOOKUP; you can get there by traversing each successive ExtendedLookup pointer. This is where it gets strange; ArraySize does not actually refer to the size of the ListHint array in the current _HEAP_LIST_LOOKUP, but rather to the total number of ListHints in the current and all previous lookups. The number of FreeList structures in the array is actually **ArraySize – BaseIndex**. **ItemCount** is the number of chunks referenced in the current _HEAP_LIST_LOOKUP (including oversized ones); **OutOfRangeItems** also refers strictly to the current lookup (and will be 0 for all but the final lookup). The purpose of **ExtraItem** is not immediately clear, but it appears to always be 1.

search by **DATE**

December 2013 (1)
November 2013 (1)
August 2013 (3)
May 2013 (1)
April 2013 (1)
May 2012 (1)
April 2012 (2)
February 2012 (2)
January 2012 (2)
October 2010 (1)
September 2010 (3)

A typical value for the second _HEAP_LIST_LOOKUP **ArraySize** is 0×400 (the value does not appear constrained to 0×80 and 0×800 as Valasek identifies). It will often look like this:



The diagram doesn't depict clearly that the _LIST_ENTRY in each _HEAP_FREE_ENTRY points to the _LIST_ENTRY member, not to the _HEAP_FREE_ENTRY start address. **_HEAP_FREE_ENTRY** is a class derived from _HEAP_ENTRY; it simply has a _LIST_ENTRY for the freelist where the user blocks would otherwise start were it busy. This is similar to the FreeEntryOffset member in free LFH chunks. Also, for clarification, the ListHints member of _HEAP_LIST_LOOKUP is pointing to the head element of an array of _LIST_ENTRY of size ArraySize-BaseIndex, and the element pointed to by the ListHead members is within the _HEAP (and is the sentinel node).

THE BACKEND

The backend chunks themselves are contained in an array (I use the term loosely here, as each array element is of a different size, but the memory structure is array-like) whose bounds and parameters are defined by **_HEAP_SEGMENT** structures. Valasek identifies those members relevant to finding and decoding the LFH. Other members of interest

are **FirstEntry** and **LastValidEntry**, which bound the range of addresses at which chunks exist.

_HEAP_SEGMENT structures start with a member **Entry**. This records the chunk used to store the segment header (of type _HEAP or _HEAP_SEGMENT), and is essentially a normal chunk, so that everything in a heap segment is a valid chunk (the size of which could be used to identify whether it contains a _HEAP or merely a _HEAP_SUBSEGMENT). However, the FirstEntry member points to the second chunk in the heap, being the one after the header; it therefore doesn't point to the first entry. LastValidEntry is also a misnomer; it actually points to the first address *not* in the segment. This will generally either be uncommitted memory or the start of another heap segment's header (so the typing of the pointer is technically correct, but probably exists only for syntactic comparability in C). Interestingly, the PreviousSize field of the initial chunk (the one containing the segment header) when decoded is always 0.

Another interesting point about the heap is that each heap can have multiple segments. Also, _**HEAP** is directly castable to _**HEAP_SEGMENT** (the latter is simply a truncation of the former). A linked list of segments for a particular heap is provided in **SegmentListEntry**; in yet another list parsing oddity, the sentinel of the linked list is _HEAP's **SegmentList** member, which always points to the _HEAP's own SegmentListEntry member. To parse this list, offset computation is required, as there are members in _HEAP_SEGMENT which precede the _LIST_ENTRY. The _HEAP_SEGMENT address referred to in the list is two DWORDs and a _HEAP_ENTRY header prior to the address actually pointed to, because the _LIST_ENTRY members point to each other and not to the parent struct. Each segment refers to a range of addresses containing chunks, and has its own sublist of uncommitted ranges. They do not have their own **Encoding**, inheriting that of the _HEAP.

Within each segment, there are ranges of memory which have been designated as part of the heap in virtual memory, but have not yet been committed. These are the uncommitted ranges in the **UCRSegmentList**. Walking the chunks in a heap by their size will imply that there is a chunk header at the start of an uncommitted range; this is not the case, and there is no sentinel. Instead, a list of uncommitted ranges (in _**HEAP_UCR_DESCRIPTOR** structures) is stored in each heap. As mentioned before, this list is exceptional; it has two _LIST_ENTRY elements at its start. The first is a node in the heap-scoped list of uncommitted ranges, _HEAP's **UCRList** member. The second is a node in the segment-scoped list UCRSegmentList; UCRSegmentList is therefore not a hint-style _LIST_ENTRY, but an actual list in its own right. Both of these members act as the sentinel node for their respective lists; the segment-scoped list requires arithmetic to dereference (subtract the size of _LIST_ENTRY). I will spare a diagram, because it would only make it look more complex than it is. The descriptor contains the first **Address** not committed, and the **Size** of the range in bytes. The descriptor entry will always be stored in the last chunk before the uncommitted range, so a backend chunk flagged last will likely contain only a _HEAP_UCR_DESCRIPTOR and be of that size (0×40 on 64-bit windows). These ranges can be skipped as though they were a chunk, when walking the backend heap.

VIRTUAL ALLOCATIONS

The **VirtualAllocdBlocks** (a _LIST_ENTRY in _HEAP) are chunks allocated by the backend which are too large to be stored in the normal heap manager (frontend or backend). Instead, they are allocated by requesting a new virtual memory allocation from the kernel, and handing this off to the user. These blocks are headed up by a _**HEAP_VIRTUAL_ALLOC_ENTRY**. You won't find this in the public symbols for most

OSes, so I'll present the structure here in 64 bits (with a nod to Nir Sofer of NirSoft, who extracted this structure from windows vista 32-bit):

As a result of this, VMM allocations will predictably have user blocks starting at offset 0×40 from an aligned address (on 64-bit, aligned to 0×1000, or 4096-byte aligned), and so it is possible to infer with a degree of certainty that a block has been allocated from the VMM from its user address. Additionally, VMM block addresses will be allocated sequentially at the next appropriately-aligned virtual address, and it is the VMM which services requests for additional heaps and segments from the backend heap manager. The structure which keeps track of which of these structures are free is not evident and is probably within the kernel. However, a _LIST_ENTRY **Entry** gives a circular linked list of VMM-allocated chunks associated with each heap. **ReserveSize** is the requested size of the allocation; **CommitSize** is the actually committed size and will under most circumstances be the same as the ReserveSize.

The list of VMM-allocated chunks will often contain LFH subsegments, because these are large enough that they would typically be serviced by the VMM instead of the normal backend heap. On 64-bit windows 7 it appears that the threshold size after which the VMM becomes responsible for allocations, which is stored in the appropriate _HEAP's **VirtualMemoryThreshold**, is always **0xFF00** blocks.

The **Entry** member of this struct is a normal chunk header, encoded against the _HEAP's Encoding member. Interestingly, WinDbg's !heap command will not identify these chunks. Additionally, **Size** will be set appropriately, but **PreviousSize** will not (it will be 0, regardless of encoding, probably to avoid key disclosure).

The **Flags** member of the Entry substruct will typically be 0×3, or **busy | extra present**. Though the **_HEAP_ENTRY_EXTRA** subclass is present, it is included in the _HEAP_VIRTUAL_ALLOC_ENTRY and so the flag does not require any additional math to be done when computing the start of the user blocks.

It bears mentioning that the backend chunk flag **0×08**, commonly called **HEAP_ENTRY_VIRTUAL_ALLOC**, is actually an indicator that the chunk is used internally by the heap manager, and does not indicate that the entry is a virtual memory allocation. It will not be found on these chunks. WinDbg identifies chunks with this flag as "busy internal" (the flag is almost always 0×09).

THE LFH

Now, moving on to the low fragmentation heap. There is a lot of depth to the LFH which isn't immediately obvious from reading the work that is out there. In particular, I will shed new light on the LFH flags (UnusedBytes), the meaning of the CRTZone structures, and how to find all the subsegments.

The LFH flags are somewhat of a peculiarity, because whether a chunk is busy is inferred from the **UnusedBytes** member of the heap entry. Commonly, it is said that the flags 0×18 interchangeably indicate that a chunk is busy. However, in many cases (possibly generated

by reallocation), the flag 0×20 also indicates a busy chunk. The flags 0×03 allegedly indicate "top" and 0×04 is usually unidentified. However, this is incorrect.

The LFH flags should be read with a mask of 0x3F. The upper two bits are always **0b10**, a fact which appears to be validated only as of windows 8. The remaining bits are the number of "wasted" bytes in the allocation. This includes 8 bytes for the chunk header (which is effectively 8 bytes on both 32 and 64 bit builds of windows, for reasons I will get into later). Because of the added 0×08, it happens that if the chunk is busy it is guaranteed to have at least one of the flag bits 0×38 set, so it is provably accurate to use these as interchangeable flag bits to indicate the busy state. For free chunks the UnusedBytes member is always 0×80, which corresponds to 0 wasted bytes. For an allocation which corresponds exactly to the maximum size serviceable by the LFH subsegment in question, it will be 0×88. Thusly, the requested size of the allocation can be inferred from the UnusedBytes field by subtracting 0×88 (or, more pedantically, masking to 0x3F and then subtracting 0×08), and then subtracting the resultant value from the size of the chunk in bytes. However, on 64-bit systems, there is a caveat to that method.

In the 64-bit implementation of the heap, each _HEAP_ENTRY begins with a new qword member **PreviousBlockPrivateData**. This member is space which is reserved for user data from the previous chunk, and allocations (may) include the size of this space. Accordingly, single-block allocations (effectively an allocation of enough space for only the chunk header) make sense in 64-bit windows because they can service requests for 8 bytes or less. The LFH therefore includes "0-block" allocations; when this happens, the subsegment is effectively an array of _HEAP_ENTRY, and all the data is stored in the PreviousBlockPrivateData member of the succeeding block. Subsegments (and heap segments) are allocated with spacing which appears to be intended to accommodate this. In this way, it is possible for a new user allocation to have the same address as a _HEAP_ENTRY struct, or alternatively an address 0×10 bytes prior to another user allocation. In the first chunk in a segment or subsegment, or special uses of _HEAP_ENTRY (such as Encoding), PreviousBlockPrivateData is unused, but tends to be initialized to 0.

The **SubsegmentZones** member of the _HEAP_LOCAL_DATA is a list of _LFH_BLOCK_ZONE structures. These are (as identified by Valasek) for keeping track of subsegments. That is to say, **CRTZone**s are arrays of **_HEAP_SUBSEGMENT** structures. **FreePointer** is a pointer to the first address not in that array (eg. an index of size+1 into the array). The actual subsegments, containing the user blocks from which allocations are drawn, are pointed to from these, and are kept in other chunks. Those chunks are allocated by the backend as well, and are often contiguous.

The important thing here is that while looking at the appropriate **_HEAP_LOCAL_SEGMENT_INFO** structure will in fact give you the address of the subsegment that will be used for allocations, there are other subsegments associated with the segment that aren't referred to there at all. Subegments become the **ActiveSubsegment** when they are new, and when chunks are freed back into them, they become the **Hint** and sometimes the **CachedItems** (which is an array of pointers to 16 _HEAP_SUBSEGMENTs, not necessarily without duplication). However, when they are filled up and nothing is freed into them, they simply pass from memory as a new ActiveSubsegment is allocated; the _HEAP_LOCAL_SEGMENT_INFO is only reminded of them once a chunk in them becomes free.

The user allocations do not actually come from the memory allocated for the CRTZone that refers to it. Instead, a block zone merely contains an array of **_HEAP_SUBSEGMENT**s, which are references to the appropriate subsegment (more specifically to its **_HEAP_USERDATA_HEADER**, which is immediately followed by the user chunk array).

The allocations for the user data itself ultimately tend to come from the VMM. Such blocks (as contain LFH userdata) look like normal VMM-allocated blocks in that they start with a structure the same length as _HEAP_VIRTUAL_ALLOC_ENTRY and have a valid chunk header at the end which is encoded against the parent heap's encoding, but all members besides the chunk header are often null; the flags are "busy internal" as expected and the requested size is at least that of the subsegment. However, this is not the case for all subsegments, and so it is probable that occasionally more than one subsegment might be stored inside one VMM allocation. Additionally, subsegments small enough to be allocated without using the VMM appear to be allocated normally from the backend. In the majority of cases this means that the first chunk in an LFH subsegment (of whatever size) will be immediately preceded by a _HEAP_ENTRY structure.

The _HEAP_USERDATA_HEADER has only a couple members that appear meaningful. **SubSegment** is a backlink to the _HEAP_SUBSEGMENT where this _HEAP_USERDATA_HEADER is the UserBlocks member. **SizeIndex** is unrelated to the size of the chunk and is not the same as the SizeIndex in the parent _HEAP_SUBSEGMENT. **Reserved** appears uninitialized. Signature is always **0xF0E0D0C0**.

Hopefully this serves to demystify the structure of the windows heap to a greater extent than before. Though this is by no means completes the available description of the windows heap and there is lots of stuff in it that remains unclear, it should make working within it substantially easier.

Posted by FALCON MOMOT *Comments* 0 *Tags*

add COMMENT

Comment

SUBMIT