Windows Heap-Backed Pool

The good, the bad, and the encoded

Yarden Shafir

CrowdStrike

About Me

- Software Engineer at CrowdStrike
- Previously a Security Researcher at SentinelOne
- Circus Artist – Aerial Arts Performer & Instructor
- Windows Internals Instructor
- Former Pastry Chef
- Blogging about Windows Security stuff
  - Windows-internals.com
- Twitter: @yarden_shafir

Windows Kernel Pool

- Kernel dynamic memory – used to store data for drivers and the system
  - Similar to the user-mode heap
- Can be Paged or NonPaged
- Common target for buffer overflow attacks leading to elevation to Ring 0
- Used to have lots of information leaks from uninitialized memory buffers being copied to user-mode
  - New API zeroes out allocations by default, avoiding those bugs

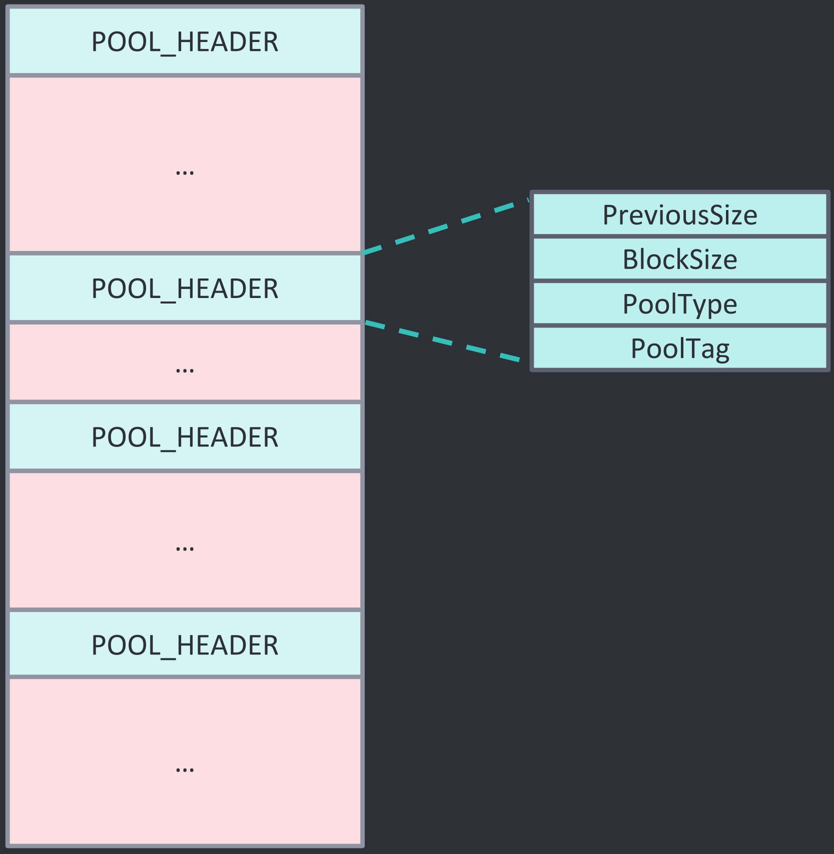
Windows Kernel Pool APIs

- ExAllocatePool
- ExAllocatePool2
- ExAllocatePool3
- ExAllocatePoolMm
- ExAllocatePoolSanityChecks
- ExAllocatePoolWithQuota
- ExAllocatePoolWithQuotaTag
- ExAllocatePoolWithTag
- ExAllocatePoolWithTagFromNode
- ExAllocatePoolWithTagPriority

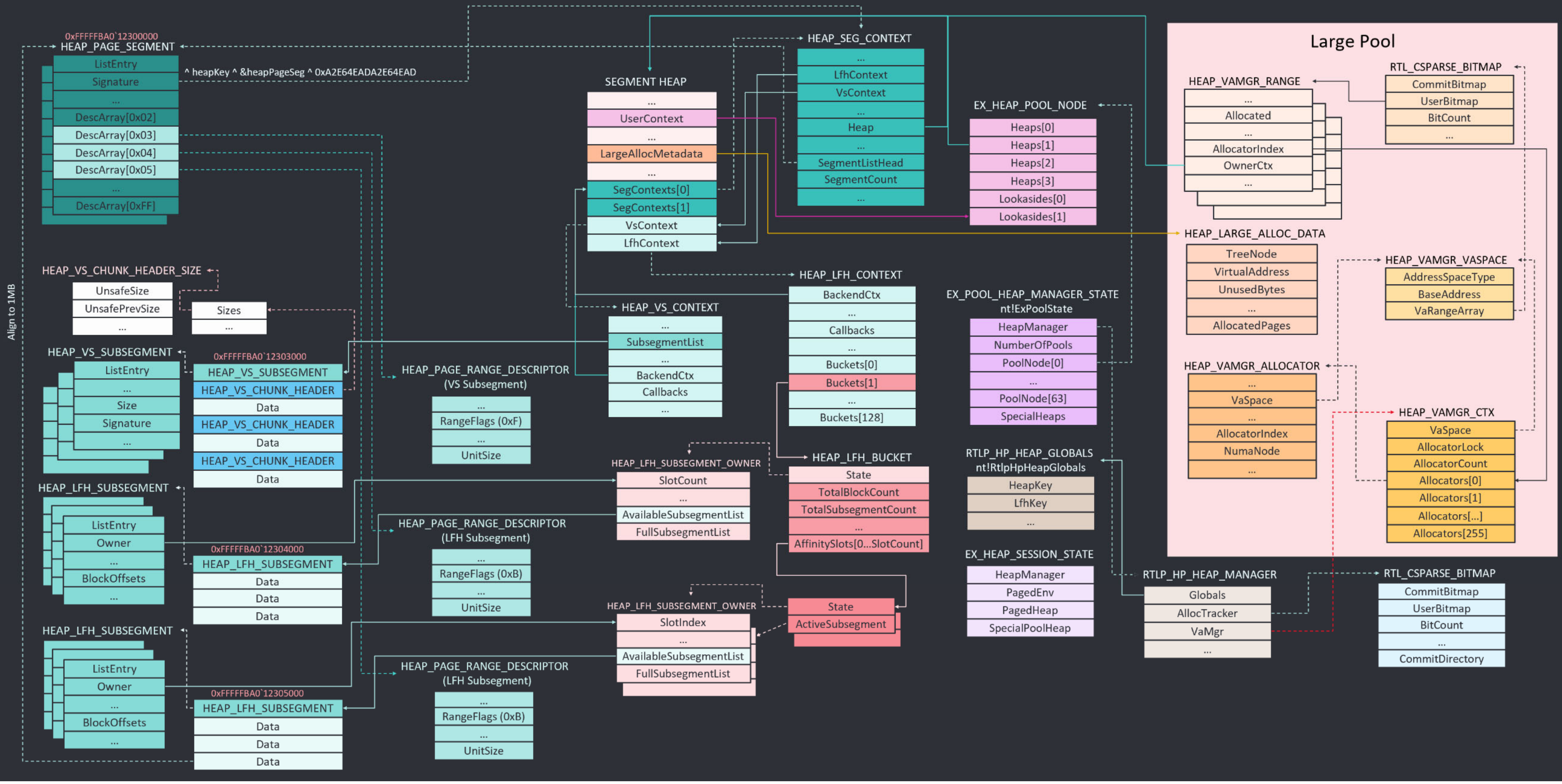
This gets even
more complicated
under the hood

Internal Kernel Pool Structure – Pre RS5

- Every pool page stands by itself
- Easy to parse
- Metadata is clearly readable
- Easy to fake when exploiting an overflow



Internal Kernel Pool Structure – RS5+

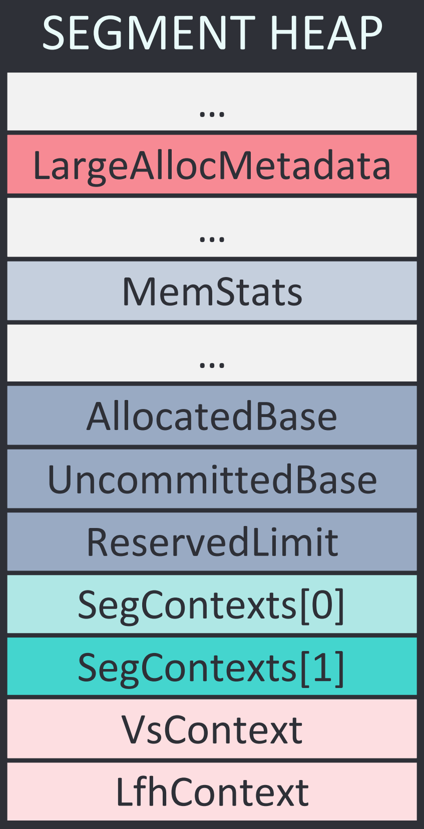


Let’s Break This Down…

- New pool design is managed through the same libraries as the user-mode heap
- Every individual pool is managed through a SEGMENT_HEAP structure
- Allocations are handled differently based on their size
  - Different mechanisms are used for different sizes
  - Large pool is still managed through the VA space
  - Looking at a single page of pool memory is no longer useful

SEGMENT_HEAP

- Manages a pool
- Contains all the metadata for the pool
  - Reserved / Committed / Free pages
  - Stats for large pages
  - Commit limit
  - Max allocation size
  - Beginning and end of committed range
- Has pointers to the structures that manage different types of pool allocations



SEGMENT HEAP

…

LargeAllocMetadata

…

MemStats

…

AllocatedBase

UncommittedBase

ReservedLimit

SegContexts[0]

SegContexts[1]

VsContext

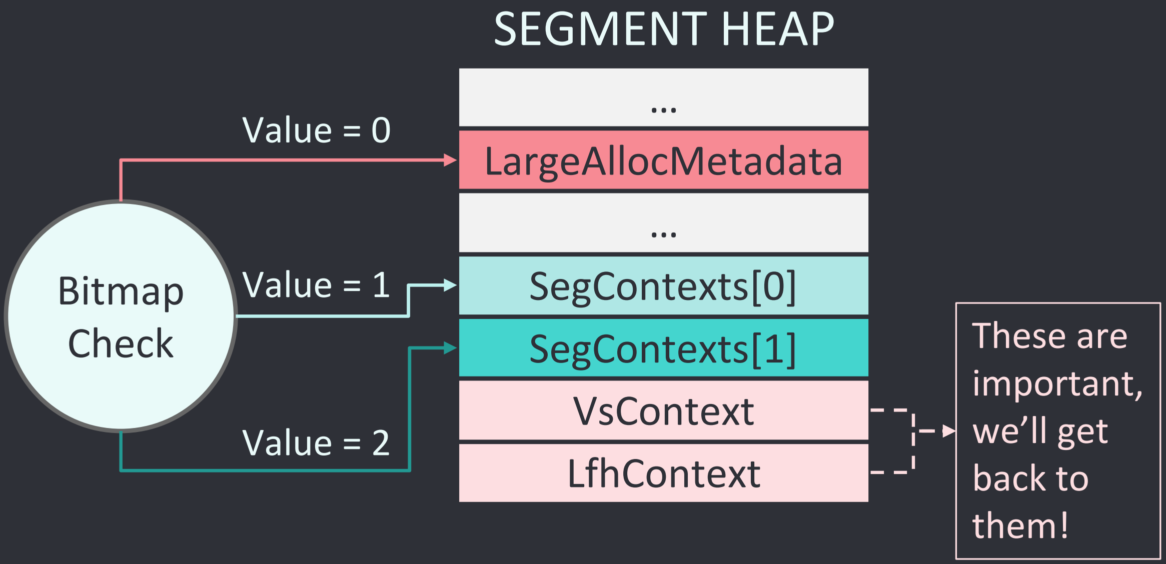
LfhContext

Pool Allocation Sizes

- Every heap can manage 3 types of allocations:
  - Up to 508KB (0-0x7F000)
  - 508KB to 8128KB (0x7F000-0x7F0000)
  - Over 8128KB
- First two types are managed by segments
  - SEGMENT_HEAP has 2 HEAP_SEG_CONTEXT structures that handle those
- Allocations over 8128KB are managed by large pool
- The kernel keeps track of allocations in a bitmap that marks their type
  - nt!ExPoolState->HeapManager->AllocTracker->AllocTrackerBitmap

Bitmap to Heap Structure

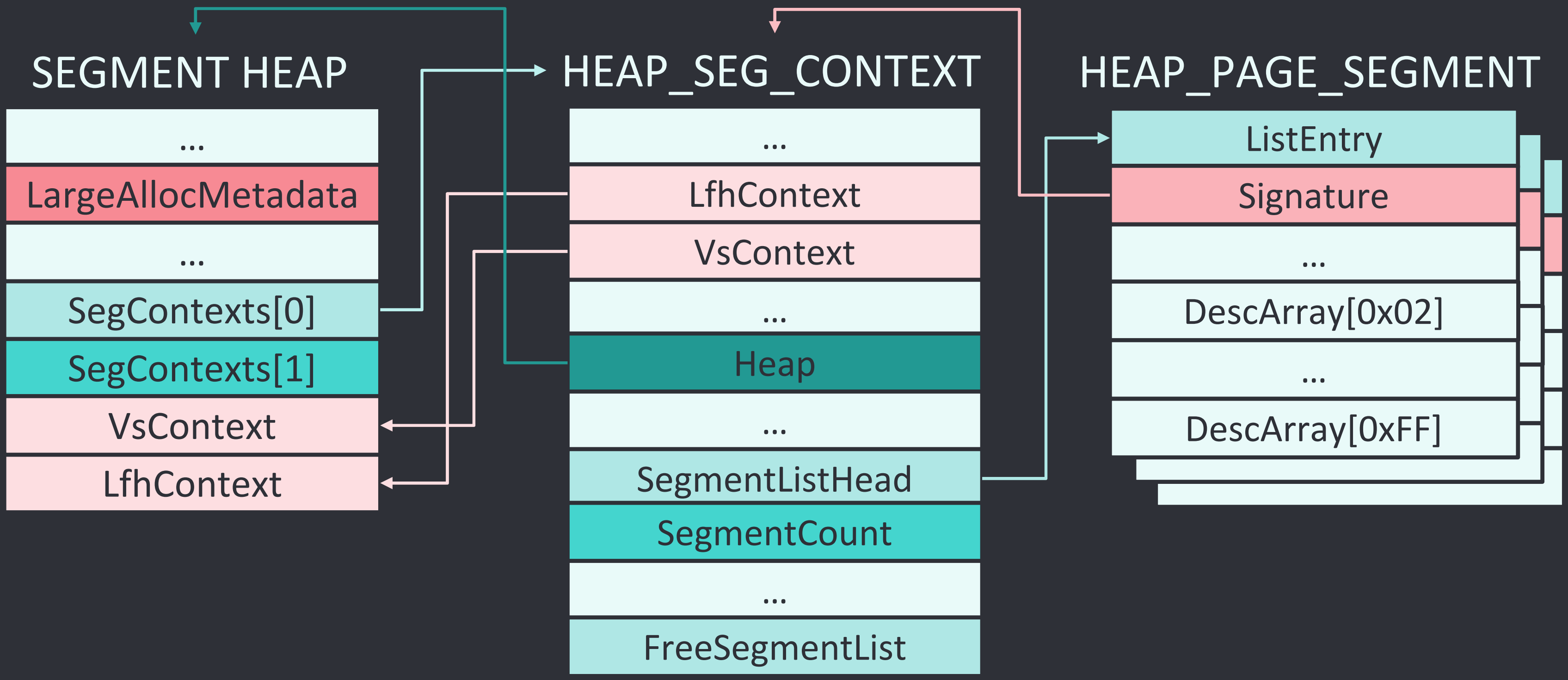
- Every two bytes represent an address in the kernel address space
- For efficiency and better use of space, bitmap has 3 layers:
  - CommitDirectory
  - CommitBitmap
  - UserBitmap



HEAP_SEG_CONTEXT

- Every HEAP_SEG_CONTEXT structure is split into segments
  - SegmentListHead links all the segments
  - FreeSegmentList links all the free segments
  - Each represented by a HEAP_PAGE_SEGMENT
- Every segment manages a set size of memory
  - SegContexts[0] handles 1MB segments in base units of 1 page
  - SegContexts[1] handles 16MB segments in base units of 16 pages
- SegmentMask field tells us how to get from pool address to its segment

Structures so far



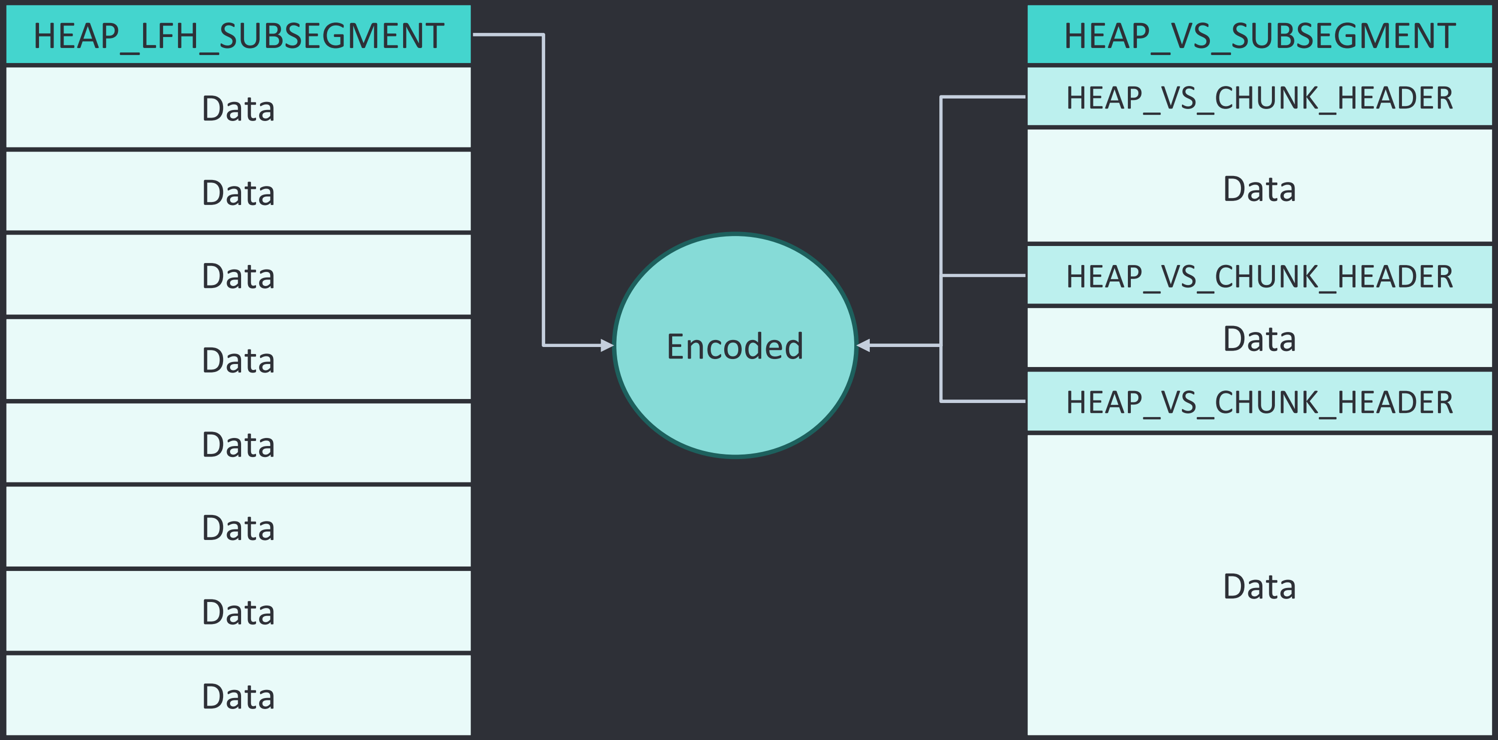
HEAP_PAGE_SEGMENT

- Has an array of 256 descriptors
  - Every descriptor describes a unit, which is part of a range
  - Describes the type of subsegment and offset of unit in it
    - And the size of the subsegment, if it’s the first unit
- ListEntry connects all segments in the SegContext
- Signature can lead back to the HEAP_SEG_CONTEXT – but not easily
  - Usually looks something like this: 0xb3a5b3d3a6f86cde
  - Needs to be decoded using a magic value, segment address and a heap key
  - HeapKey is saved in a global kernel variable, not exported

Two Types of Subsegments

- LFH – Low Fragmentation Heap
  - Used for allocations of 129 common sizes
  - All allocations in a subsegment have the same size
  - Gets created after a certain number of allocations of the same size
  - Max size is 0x4000 – only exists in SegContexts[0]
  - No need for unique header for each block – saves space
  - Block status is managed by a bitmap in the header
- VS – Variable Size
  - Manages all blocks that don’t fit LFH buckets
  - Every block has its own header to describe it

LFH vs. VS Ranges



HEAP_LFH_SUBSEGMENT

Data

Data

Data

Data

Data

Data

Data

Data

Encoded

HEAP_VS_SUBSEGMENT

HEAP_VS_CHUNK_HEADER

Data

HEAP_VS_CHUNK_HEADER

Data

HEAP_VS_CHUNK_HEADER

Data

Heap Subsegments – VS and LFH

- Structures containing block sizes are encoded
  - VS headers encode with the heap key, LFH use LFH key
  - Makes it a lot harder to parse the pool – and fake headers when exploiting an overflow
- Size of subsegments is not fixed
- Subsegments wrap several pages – have to start at beginning of subsegment to parse
  - Looking at a single page isn’t possible anymore – data will make no sense
- Allocated blocks will still have POOL_HEADER
  - But free blocks won’t, can’t rely on them to parse the pool anymore and nothing uses them

Subsegment Headers Encoding

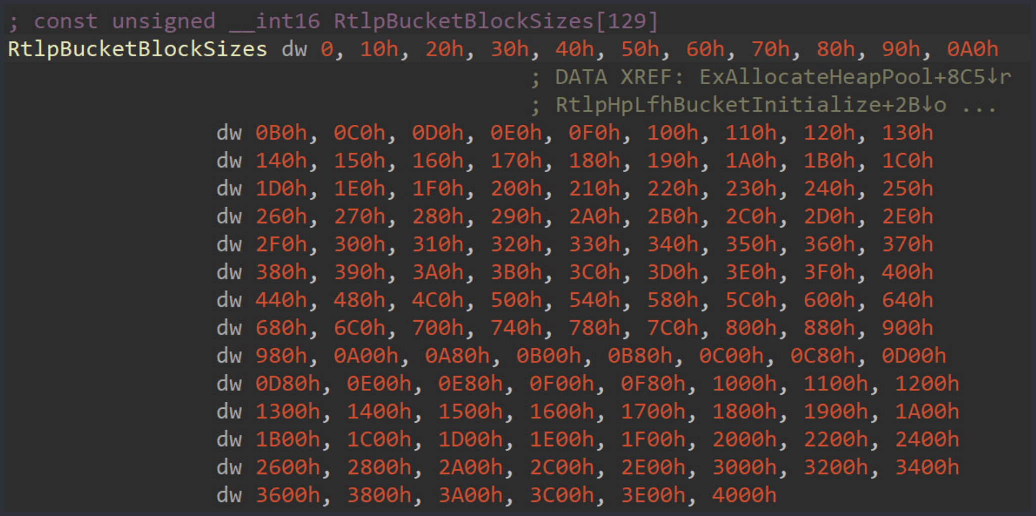
- HEAP_LFH_SUBSEGMENT
  - Block sizes and offsets is in encoded BlockOffsets field
  - Data = EncodedData ^ LfhKey ^ ((ULONG)(Subsegment) >> 12)
- HEAP_VS_SUBSEGMENT
  - Has a linked list of all subsegments – needs to be decoded with the address of current subsegment
    - LFH subsegments are also linked – but this list isn’t encoded
- HEAP_VS_CHUNK_HEADER
  - Exists for every block in VS subsegment
  - Block size and allocation status in encoded Sizes field
  - Data = Sizes.HeaderBits ^ HeapKey ^ ChunkHeader

LFH and VS Contexts

- SEGMENT_HEAP contains LFH context and VS context
- HEAP_VS_CONTEXT
  - Contains a list of all VS subsegments in the heap
  - Keeps track of committed and free chunks
  - Has suballocator callbacks – allocate, free, commit…
- HEAP_LFH_CONTEXT
  - Has array of 129 LFH buckets
    - Each bucket links all subsegments that use that size
    - Keeps data on block size, subsegment count, block count
  - Has suballocator callbacks as well

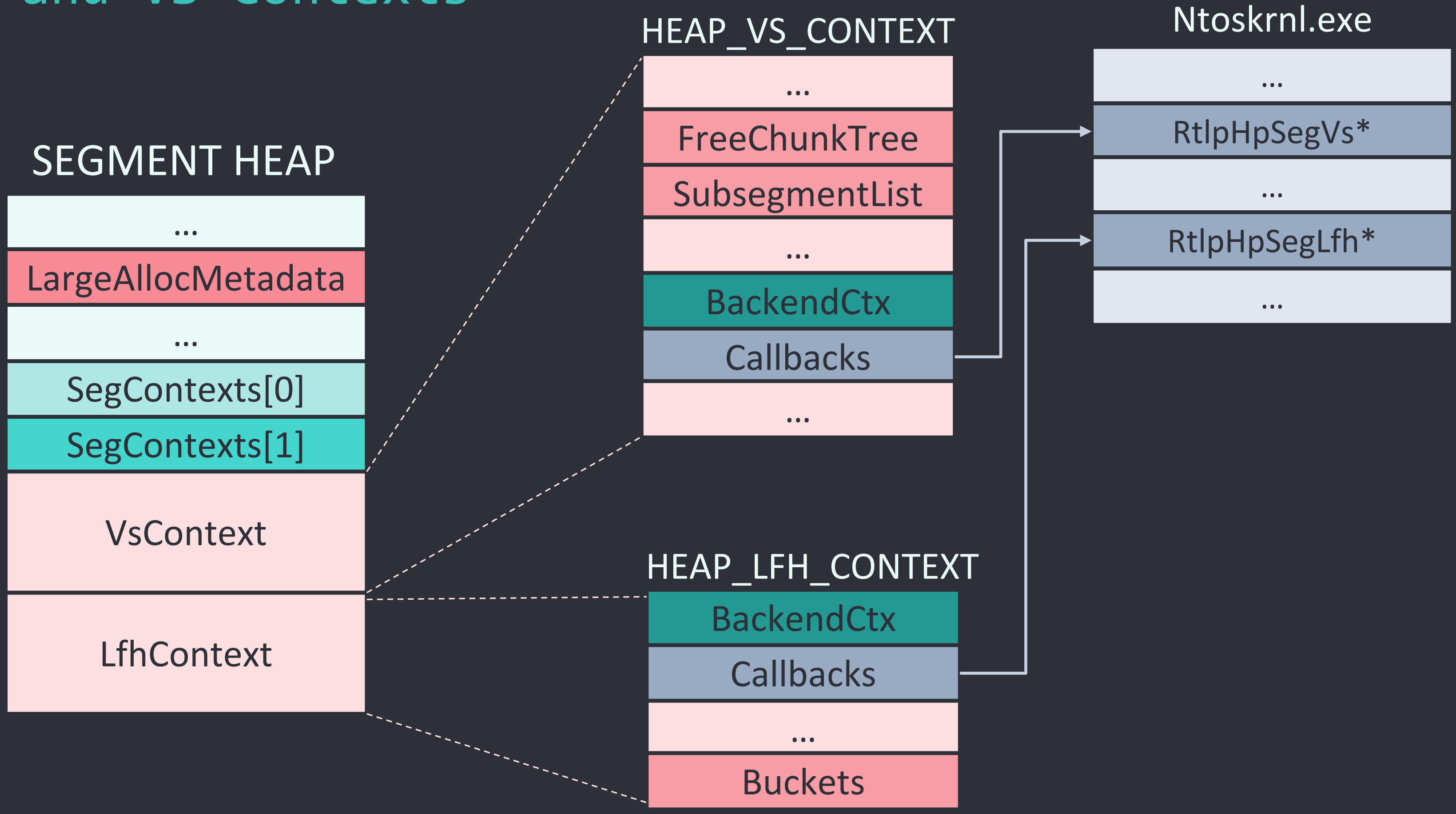
LFH Buckets

- Support common block sizes that are multiples of 0x10, 0x40, 0x80, 0x100, 0x200
- Get activated after 0x10 allocations of the same size



```
; const unsigned __int16 RtlpBucketBlockSizes[129]
RtlpBucketBlockSizes dw 0, 10h, 20h, 30h, 40h, 50h, 60h, 70h, 80h, 90h, 0A0h
                                        ; DATA XREF: ExAllocateHeapPool+8C5↓r
                                        ; RtlpHpLfhBucketInitialize+2B↓o ...
                dw 0B0h, 0C0h, 0D0h, 0E0h, 0F0h, 100h, 110h, 120h, 130h
                dw 140h, 150h, 160h, 170h, 180h, 190h, 1A0h, 1B0h, 1C0h
                dw 1D0h, 1E0h, 1F0h, 200h, 210h, 220h, 230h, 240h, 250h
                dw 260h, 270h, 280h, 290h, 2A0h, 2B0h, 2C0h, 2D0h, 2E0h
                dw 2F0h, 300h, 310h, 320h, 330h, 340h, 350h, 360h, 370h
                dw 380h, 390h, 3A0h, 3B0h, 3C0h, 3D0h, 3E0h, 3F0h, 400h
                dw 440h, 480h, 4C0h, 500h, 540h, 580h, 5C0h, 600h, 640h
                dw 680h, 6C0h, 700h, 740h, 780h, 7C0h, 800h, 880h, 900h
                dw 980h, 0A00h, 0A80h, 0B00h, 0B80h, 0C00h, 0C80h, 0D00h
                dw 0D80h, 0E00h, 0E80h, 0F00h, 0F80h, 1000h, 1100h, 1200h
                dw 1300h, 1400h, 1500h, 1600h, 1700h, 1800h, 1900h, 1A00h
                dw 1B00h, 1C00h, 1D00h, 1E00h, 1F00h, 2000h, 2200h, 2400h
                dw 2600h, 2800h, 2A00h, 2C00h, 2E00h, 3000h, 3200h, 3400h
                dw 3600h, 3800h, 3A00h, 3C00h, 3E00h, 4000h
```

LFH and VS Contexts



Exploitation Limitations

- Spraying the pool is harder
  - Need to take into account subsegment types of sprayed objects and overflowing allocation – have to be exact same size if they will use LFH
- Overflowing VS allocation will overwrite encoded VS header
  - Will need to account for header size and fake it
  - Successfully faking a header requires leaking heap key and exact address of header that will be faked

Exploitation Limitations – Cont.

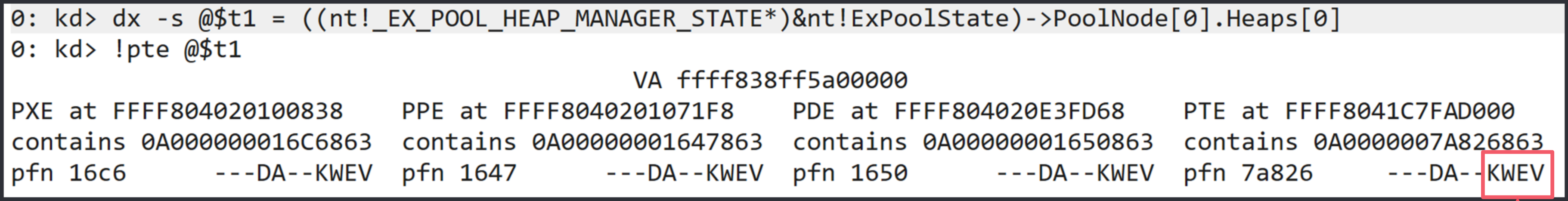
- Subsegments cross page boundaries – blocks don’t
  - Can create empty spaces at the end of pages that need to be known and accounted for in exploitation
- Overflowing the last block in a subsegment leads to overwriting next subsegment header
  - Need complex info leak to tell where it is and which subsegment type
  - Might require more encoding to fake
- And all older pool mitigations…
  - Pool allocations are zeroed out to prevent info leaks
  - Object header mitigations to break known exploit techniques

But There are Some Benefits too

- Every pool type is initialized when system starts
- Necessary structures are allocated in the pool
  - SEGMENT_HEAP – contains 2 HEAP_SEG_CONTEXT, VS context and LFH context
- This includes the executable NonPagedPool
  - This type is not in use anymore, but still exists
  - Will only have one page – the one needed for SEGMENT_HEAP and the structures it contains
  - SEGMENT_HEAP uses around 0x800 bytes – leaving half a page of unused RWX kernel memory!
  - In HVCI systems this page won’t exist
  - Alex Ionescu discovered this

RWX Kernel Pool Page

- nt!ExPoolState has array of nodes
- Each node has 4 pools:
  - NonPagedPool
  - NonPagedPoolNx
  - PagedPool
  - PagedPool (Prototype pool)



```
0: kd> dx -s @$t1 = ((nt!_EX_POOL_HEAP_MANAGER_STATE*)&nt!ExPoolState)->PoolNode[0].Heaps[0]
0: kd> !pte @$t1
                                           VA ffff838ff5a00000
PXE at FFFF804020100838    PPE at FFFF8040201071F8    PDE at FFFF804020E3FD68    PTE at FFFF8041C7FAD000
contains 0A000000016C6863  contains 0A00000001647863  contains 0A00000001650863  contains 0A0000007A826863
pfn 16c6      ---DA--KWEV  pfn 1647      ---DA--KWEV  pfn 1650      ---DA--KWEV  pfn 7a826     ---DA--KWEV
```

Writable +
Executable!

Secure Kernel Address Leaks

- Secure Pool is managed by the secure kernel and is read only for normal kernel code
  - Meant for drivers to protect their data from corruption by malicious code running in Ring 0
  - Uses the same design as normal kernel pools
- SEGMENT_HEAP exists in the beginning of the pool and contains VS and LFH Contexts
  - These contain Callbacks – pointing to SecureKernel.exe functions (discovered by Alex Ionescu and fixed)
- But the secure pool is still leaking addresses…
  - HEAP_PAGE_SEGMENT exists in each subsegment – can leak the address of the secure pool and the secure kernel metadata heap

Debugging and Analysis Tools

- !pool extension was broken for a while
  - Fixed and doing better now
- PoolViewer – GUI tool to visualize kernel heap
- PoolViewExt – debugger extension
  - Searches for pool allocation (like !pool)
  - Implements searching for pool tag

Previous Work

- Kernel heap exploitation technique: https://github.com/synacktiv/Windows-kernel-SegmentHeap-Aligned-Chunk-Confusion/blob/master/Scoop_The_Windows_10_pool.pdf
- Overview of segment heap: https://speakerdeck.com/scwuaptx/windows-kernel-heap-segment-heap-in-windows-kernel-part-1
- Segment heap internals: https://www.blackhat.com/docs/us-16/materials/us-16-Yason-Windows-10-Segment-Heap-Internals.pdf

Questions?