

---

# WINDOWS 10 SEGMENT HEAP INTERNALS

**Mark Vincent Yason**

IBM X-Force Advanced Research

yasonm[at]ph[dot]ibm[dot]com

@MarkYason

## ABSTRACT

Introduced in Windows 10, Segment Heap is the native heap implementation used in Windows apps (formerly called Modern/Metro apps) and certain system processes. This new heap implementation is an addition to the well-researched and widely documented NT Heap that is still used in traditional applications and in certain types of allocations in Windows apps.

One important aspect of the Segment Heap is that it is enabled for Microsoft Edge which means that components/dependencies running in Edge that do not use a custom heap manager will use the Segment Heap. Therefore, reliably exploiting memory corruption vulnerabilities in these Edge components/dependencies would require some level of understanding of the Segment Heap.

In this presentation, I'll discuss the data structures, algorithms and security mechanisms of the Segment Heap. Knowledge of the Segment Heap is also applied by discussing and demonstrating how a memory corruption vulnerability in the Microsoft WinRT PDF library (CVE-2016-0117) is leveraged for a reliable arbitrary write in the context of the Edge content process.



**IBM Security**

© 2016 IBM Corporation

## CONTENTS

1.	Introduction .....	5
2.	Internals .....	6
2.1.	Overview .....	6
	Architecture .....	6
	Defaults and Configuration .....	6
2.2.	HeapBase, Block Allocation and Block Freeing .....	7
	_SEGMENT_HEAP Structure .....	8
	Block Allocation .....	9
	Block Freeing .....	10
2.3.	Backend Allocation .....	12
	Segment Structure .....	12
	_HEAP_PAGE_SEGMENT Structure .....	13
	_HEAP_PAGE_RANGE_DESCRIPTOR Structure .....	13
	Backend Free Tree .....	14
	Backend Allocation .....	15
	Backend Freeing .....	17
2.4.	Variable Size Allocation .....	18
	VS Subsegments .....	18
	_HEAP_VS_CONTEXT Structure .....	18
	_HEAP_VS_SUBSEGMENT Structure .....	18
	_HEAP_VS_CHUNK_HEADER Structure .....	19
	_HEAP_VS_CHUNK_FREE_HEADER Structure .....	20
	VS Free Tree .....	21
	VS Allocation .....	21
	VS Freeing .....	23
2.5.	Low Fragmentation Heap .....	24
	LFH Subsegments .....	25
	_HEAP_LFH_CONTEXT Structure .....	25
	_HEAP_LFH_ONDEMAND_POINTER Structure .....	25
	_HEAP_LFH_BUCKET Structure .....	26
	_HEAP_LFH_AFFINITY_SLOT Structure .....	26
	_HEAP_LFH_SUBSEGMENT_OWNER Structure .....	27

_HEAP_LFH_SUBSEGMENT Structure .....	27
LFH Block Bitmap .....	29
LFH Bucket Activation .....	30
LFH Allocation .....	30
LFH Freeing .....	32
2.6. Large Blocks Allocation .....	32
_HEAP_LARGE_ALLOC_DATA Structure .....	33
Large Block Allocation.....	33
Large Block Freeing.....	34
2.7. Block Padding .....	34
2.8. Summary and Analysis: Internals .....	35
3. Security Mechanisms .....	36
3.1. Fast Fail on Linked List Node Corruption.....	36
3.2. Fast Fail on RB Tree Node Corruption .....	36
3.3. Heap Address Randomization .....	37
3.4. Guard Pages .....	38
3.5. Function Pointer Encoding .....	39
3.6. VS Block Sizes Encoding .....	39
3.7. LFH Subsegment BlockOffsets Encoding .....	40
3.8. LFH Allocation Randomization .....	40
3.9. Summary and Analysis: Security Mechanisms .....	41
4. Case Study .....	42
4.1. CVE-2016-0117 Vulnerability Details .....	42
4.2. Plan for Implanting the Target Address .....	43
4.3. Manipulating the MSVCRT Heap with Chakra's ArrayBuffer .....	44
Allocation and Setting Controlled Values .....	44
LFH Bucket Activation .....	44
Freeing and Garbage Collection .....	45
4.4. Preventing Target Address Corruption .....	45
4.5. Preventing Freed Controlled Buffer Blocks from Being Coalesced .....	46
4.6. Preventing Unintended Use of Freed Controlled Buffers Blocks .....	47
4.7. Adjusted Plan for Implanting the Target Address .....	47
4.8. Successful Arbitrary Write.....	48
4.9. Analysis and Summary: Case Study.....	48

5.	Conclusion .....	50
6.	Appendix: WinDbg !heap Extension Commands for Segment Heap .....	51
	!heap -x <address> .....	51
	!heap -i <address> -h <heap> .....	51
	!heap -s -a -h <heap> .....	51
7.	Bibliography .....	53

## 1. INTRODUCTION

With the introduction of Windows 10, Segment Heap, a new native heap implementation was also introduced. It is currently the native heap implementation used in Windows apps (formerly called Modern/Metro apps) and in certain system processes while the older native heap implementation (NT Heap) is still the default for traditional applications.

From a security researcher's perspective, understanding the internals of the Segment Heap is important as attackers may leverage or exploit this new and critical component in the near future, especially because it is being used by the Edge browser. Additionally, a security researcher performing software audits may need to develop a proof-of-concept for a vulnerability in order to prove exploitability to the vendor/developer. If creating the proof-of-concept requires precise manipulation of a heap managed by the Segment Heap, an understanding of its internals will definitely help. This paper aims to help the reader have a deep understanding of the Segment Heap.

This paper is divided into three major sections. The first section (Internals) discusses in depth the different components of the Segment Heap. It includes the data structures and algorithms used by each Segment Heap component when performing their functions. The second section (Security Mechanisms) discusses the different mechanisms that make it difficult or unreliable to attack important Segment Heap metadata, and in certain cases, make it difficult to conduct precise heap layout manipulation. The third section (Case Study) is where the understanding of the Segment Heap is applied by discussing methods for manipulating the layout of a heap managed by the Segment Heap in order to reliably leverage a vulnerability for an arbitrary write.

Since the Segment Heap and NT Heap share similar concepts, the reader is encouraged to read prior works that discuss NT Heap internals [1, 2, 3, 4, 5]. These prior works and the various papers/presentations they reference also discuss the security mechanisms and attack techniques for the NT Heap which will give the reader an idea why certain heap security mechanisms are in place in the Segment Heap.

All information in this paper is based on NTDLL.DLL (64-bit) version 10.0.14295.1000 from the Windows 10 Redstone 1 Preview (Build 14295).

## 2. INTERNALS

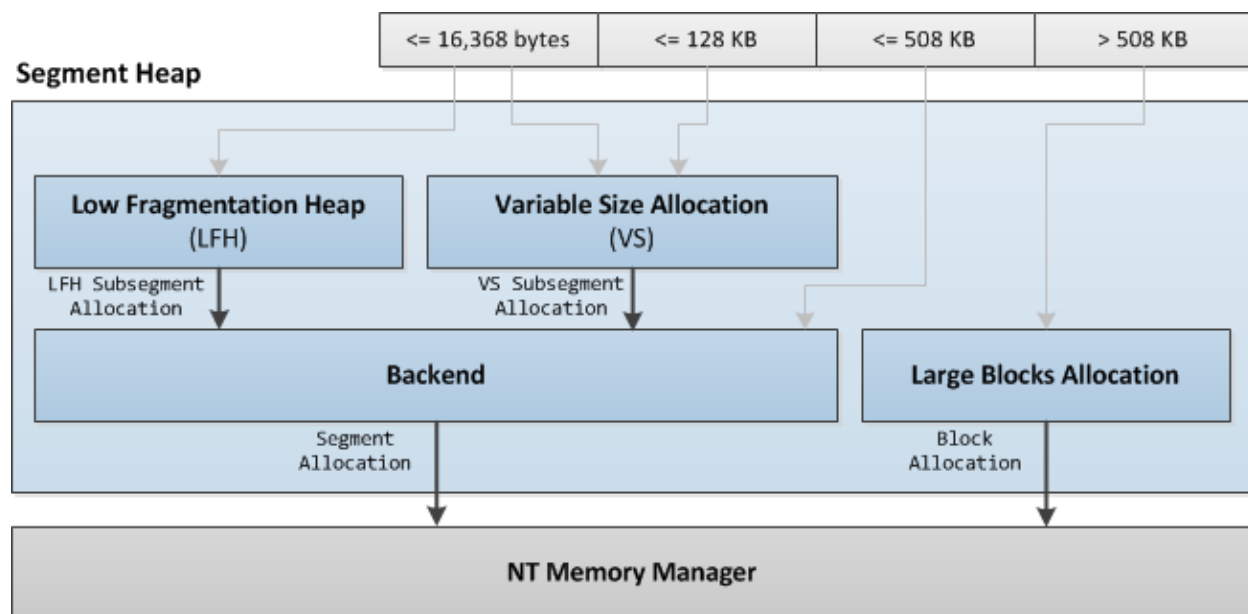
This first section discusses in depth the internals of the Segment Heap. The discussion will start with an overview of the different components of the Segment Heap and then describing the instances when the Segment Heap will be enabled in a process. After the overview, each Segment Heap component will be discussed in details in their own subsections.

Note that internal NTDLL functions discussed here may be inlined in some NTDLL builds. Therefore, the internal functions may not be seen in IDA's functions listing and a copy of the functions may be seen embedded in other functions.

### 2.1. OVERVIEW

#### Architecture

The Segment Heap consists of four components: (1) The backend which services allocation requests for >128KB to 508KB. It uses the virtual memory functions provided by the NT Memory Manager to create the segments where the backend blocks are allocated from. (2) The variable size (VS) allocation component which services allocation requests for <=128KB. It uses the backend to create the VS subsegments where VS blocks are allocated from. (3) The Low Fragmentation Heap (LFH) services allocation requests for <=16,368 bytes but only if the allocation size is detected to be commonly used in allocations. It uses the backend to create the LFH subsegments where LFH blocks are allocated from. (4) The large blocks allocation component services allocation requests for >508KB. It uses the virtual memory functions provided by the NT Memory Manager for the allocation of large blocks.



#### Defaults and Configuration

The Segment Heap is currently an opt-in feature. Windows apps are opted-in by default and executables with a name that matches any of the following (names of system executables) are also opted-in by default to use the Segment Heap:

- csrss.exe

- lsass.exe
- runtimebroker.exe
- services.exe
- smss.exe
- svchost.exe

To enable or disable the Segment Heap for a specific executable, the following Image File Execution Options (IFEO) registry value can be set.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\
Image File Execution Options\executable
FrontEndHeapDebugOptions = (DWORD)

Bit 2 (0x04): Disable Segment Heap
Bit 3 (0x08): Enable Segment Heap
```

To globally enable or disable the Segment Heap for all executables, the following registry value can be set:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Segment Heap
Enabled = (DWORD)

0      : Disable Segment Heap
(Not 0): Enable Segment Heap
```

If after all the checks it is determined that a process will use the Segment Heap, bit 0 of the global variable `RtlpHpHeapFeatures` will be set.

Note that even if Segment Heap is enabled in a process, not all heaps that will be created by the process will be managed by the Segment Heap as there are specific types of heaps that still need to be managed by the NT Heap (this will be discussed in the next subsection).

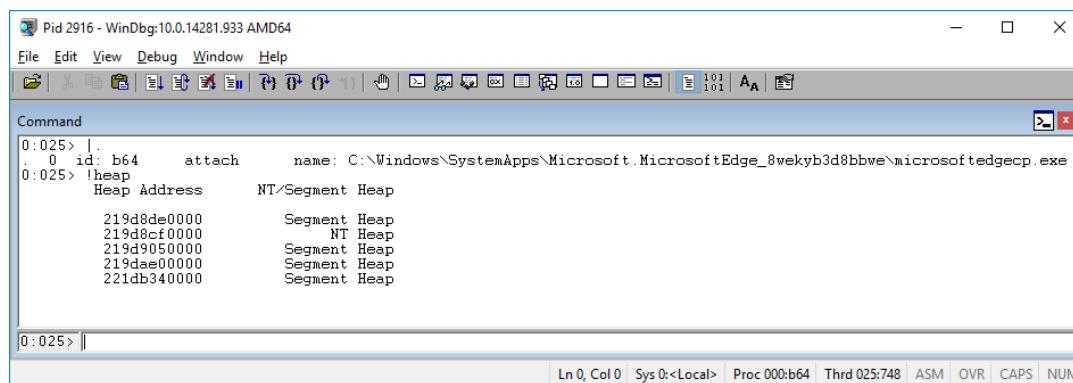
## 2.2. HEAPBASE, BLOCK ALLOCATION AND BLOCK FREEING

If the Segment Heap is enabled (bit 0 of `RtlpHpHeapFeatures` is set), the heap created by `HeapCreate()` will be managed by the Segment Heap unless the `dwMaximumSize` argument passed to it is not zero which means the heap is not growable.

If the `RtlCreateHeap()` API is directly used to create the heap, additional checks are performed to determine whether the Segment Heap or the NT Heap will manage the new heap. Specifically, all of the following should be true in order for the Segment Heap to manage the new heap:

- Heap should be growable: Flags argument passed to `RtlCreateHeap()` should have `HEAP_GROWABLE` set.
- Heap memory should not be pre-allocated (suggests a shared heap): `HeapBase` argument passed to `RtlCreateHeap()` should be `NULL`.
- If a `Parameters` argument is passed to `RtlCreateHeap()`, the following `Parameters` fields should be set to 0/`NULL`: `SegmentReserve`, `SegmentCommit`, `VirtualMemoryThreshold` and `CommitRoutine`.
- The `Lock` argument passed to `RtlCreateHeap()` should be `NULL`.

The illustration below shows the heaps created when the Edge content process (a Windows app) is initially loaded.



Four of five are managed by the Segment Heap. The first heap is the default process heap and the third heap is the MSVCRT heap (msvcrt!crtheap). The second heap is a shared heap (ntdll!CsrPortHeap), and therefore, it is managed by the NT Heap.

### \_SEGMENT\_HEAP Structure

When a heap managed by the Segment Heap is created, the heap address/handle (called HeapBase for rest of this paper) returned by HeapCreate() or RtlCreateHeap() will point to a \_SEGMENT\_HEAP structure, the counterpart of the \_HEAP structure of the NT Heap.

The HeapBase is the central location where the states of the different Segment Heap components are stored, it has the following fields:

```

windbg> dt ntdll!_SEGMENT_HEAP
+0x000 TotalReservedPages : Uint8B
+0x008 TotalCommittedPages : Uint8B
+0x010 Signature : Uint4B
+0x014 GlobalFlags : Uint4B
+0x018 FreeCommittedPages : Uint8B
+0x020 Interceptor : Uint4B
+0x024 ProcessHeapListIndex : Uint2B
+0x026 GlobalLockCount : Uint2B
+0x028 GlobalLockOwner : Uint4B
+0x030 LargeMetadataLock : _RTL_SRWLOCK
+0x038 LargeAllocMetadata : _RTL_RB_TREE
+0x048 LargeReservedPages : Uint8B
+0x050 LargeCommittedPages : Uint8B
+0x058 SegmentAllocatorLock : _RTL_SRWLOCK
+0x060 SegmentListHead : _LIST_ENTRY
+0x070 SegmentCount : Uint8B
+0x078 FreePageRanges : _RTL_RB_TREE
+0x088 StackTraceInitVar : _RTL_RUN_ONCE
+0x090 ContextExtendLock : _RTL_SRWLOCK
+0x098 AllocatedBase : Ptr64 UChar
+0x0a0 UncommittedBase : Ptr64 UChar
+0x0a8 ReservedLimit : Ptr64 UChar
+0x0b0 VsContext : _HEAP_VS_CONTEXT
+0x120 LfhContext : _HEAP_LFH_CONTEXT

```

- Signature - 0xDDEEDDEE (heap is managed by the Segment Heap).

Fields for tracking large blocks allocation state (further discussed in 2.6):

- LargeAllocMetadata - Red-black tree (RB tree) [6] of large blocks metadata.
- LargeReservedPages - Number of pages that are reserved for all large blocks allocation.



- `LargeCommittedPages` - Number of pages that are committed for all large blocks allocation.

Fields for tracking backend allocation state (further discussed in 2.3):

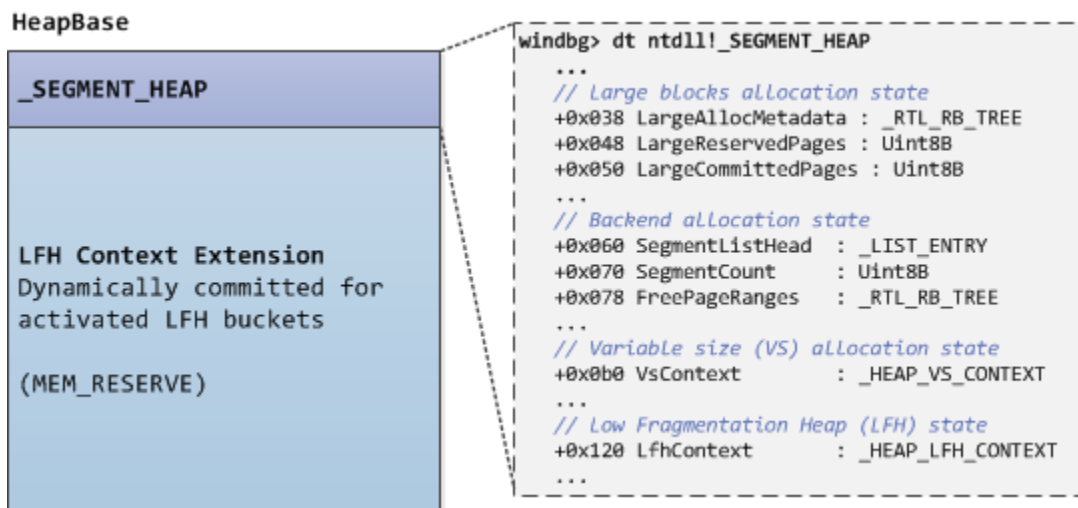
- `SegmentCount` - Number of segments owned by the heap.
- `SegmentListHead` - Linked list of segments owned by the heap.
- `FreePageRanges` - RB tree of free backend blocks.

The following substructures track the variable size allocation and the Low Fragmentation Heap states:

- `VsContext` - Tracks the state of the variable size allocation (see 2.4).
- `LfhContext` - Tracks the state of the Low Fragmentation Heap (see 2.5).

The heap is allocated and initialized via a call to `RtlpHpSegHeapCreate()`. `NtAllocateVirtualMemory()` is used to reserve and commit the virtual memory for the heap in which the reserve size varies depending on the number of processors and the commit size is the size of the `_SEGMENT_HEAP` structure.

The remaining reserved memory below the `_SEGMENT_HEAP` structure is called the LFH context extension and it is dynamically committed to store the necessary data structures for activated LFH buckets.



## Block Allocation

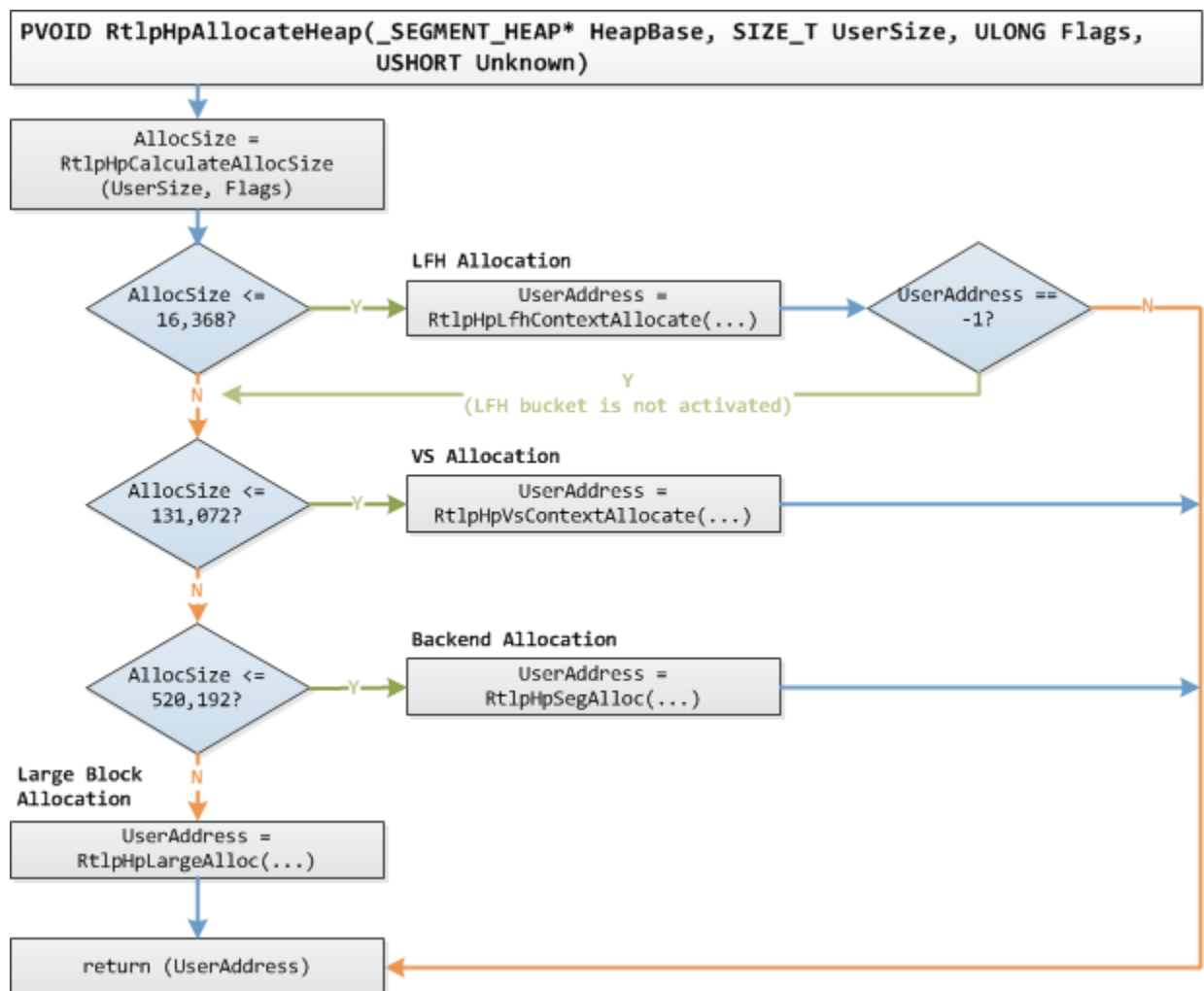
When allocating a block via `HeapAlloc()` or `RtlAllocateHeap()`, the allocation request will eventually be routed to `RtlpHpAllocateHeap()` if the heap is managed by the Segment Heap.

`RtlpHpAllocateHeap()` has the following function signature:

```
PVOID RtlpHpAllocateHeap(_SEGMENT_HEAP* HeapBase, SIZE_T UserSize, ULONG Flags, USHORT Unknown)
```

Where `UserSize` (user-requested size) is the size passed to `HeapAlloc()` or `RtlAllocateHeap()`. The return value is the pointer to the newly allocated block (called `UserAddress` for the rest of this paper).

The diagram below shows the logic of `RtlpHpAllocateHeap()`:



The purpose of `RtlpHpAllocateHeap()` is to call the allocation function of the appropriate Segment Heap component based on `AllocSize`. `AllocSize` (allocation size) is the adjusted `UserSize` depending on `Flags`, but by default, `AllocSize` will be equal to `UserSize` unless `UserSize` is 0 (if `UserSize` is 0, `AllocSize` will be 1).

Note that the logic starting where `AllocSize` is checked is actually in a separate `RtlpHpAllocateHeapInternal()` function, it is just inlined in the diagram for brevity. Also, one part to notice is that if LFH allocation returns -1, it means that the LFH bucket corresponding to `AllocSize` is not yet activated, and therefore, the allocation request will eventually be passed to VS allocation.

## Block Freeing

When freeing a block via `HeapFree()` or `RtlFreeHeap()`, the call will eventually be routed to `RtlpHpFreeHeap()` if the heap is managed by the Segment Heap.

`RtlpHpFreeHeap()` has the following function signature:

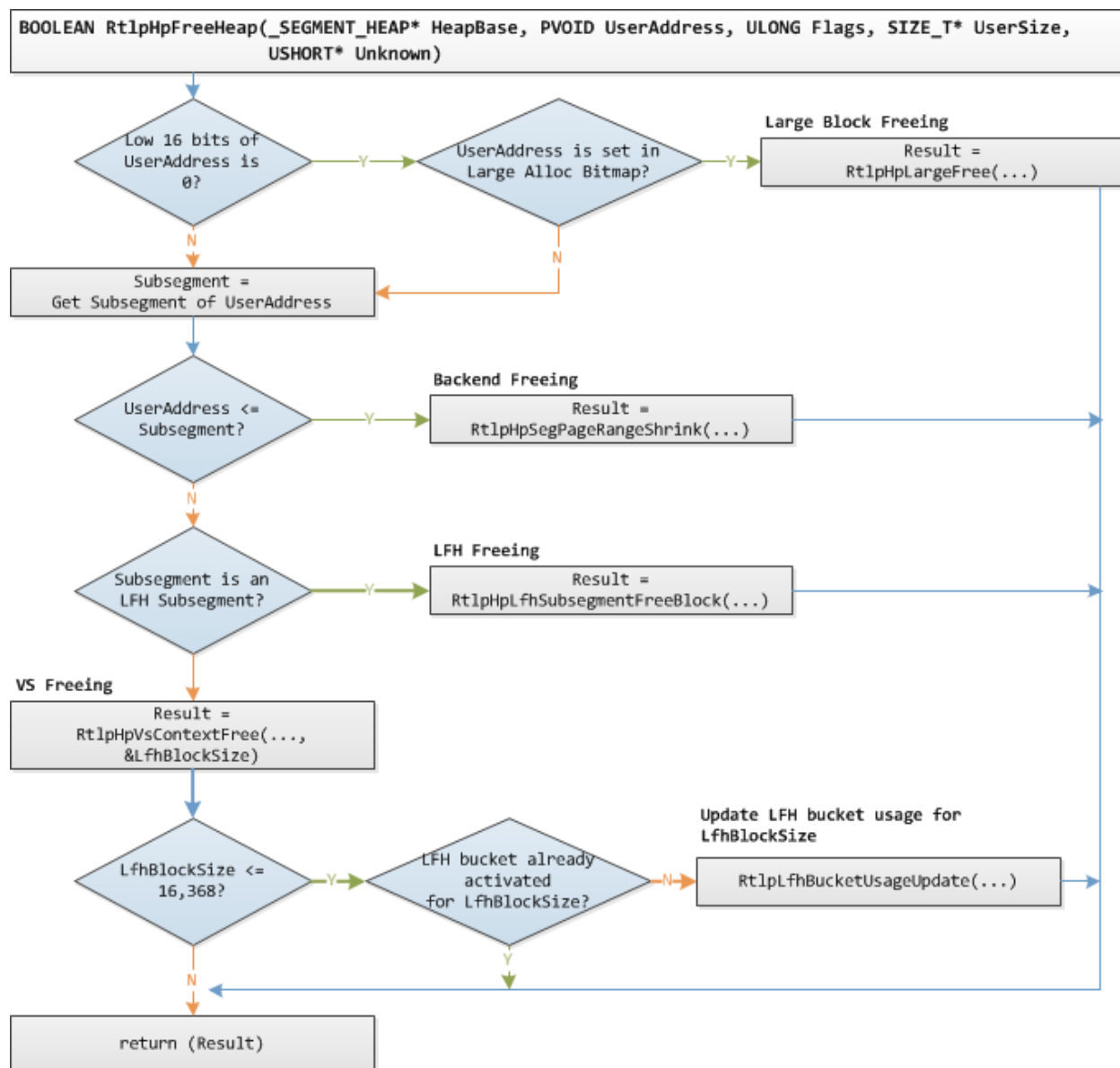
```

BOOLEAN RtlpHpFreeHeap(_SEGMENT_HEAP* HeapBase, PVOID UserAddress, ULONG Flags,
                        SIZE_T* UserSize, USHORT* Unknown)

```

Where UserAddress is the block address returned by HeapAlloc() or Rt1AllocateHeap() and the UserSize will become the user-requested size of the freed block.

The diagram below shows the freeing logic of Rt1HpFreeHeap():



The purpose of `Rt1HpFreeHeap()` is to call the freeing function of the appropriate Segment Heap component based on the value of UserAddress and what type of subsegment it is located. Subsegments will be further discussed later in this paper, but for now, subsegments are special types of backend blocks where VS or LFH blocks are allocated from.

Since the address of large allocations are 64KB aligned, a UserAddress with low 16 bits cleared is first checked against the large allocation bitmap. If the UserAddress (actually `UserAddress >> 16`) is set in the large allocation bitmap, large block freeing is called.

Next, the subsegment where `UserAddress` is located is determined. If the resulting subsegment address is less than or equal the `UserAddress`, it means that the `UserAddress` is for a backend block because the address of VS blocks and LFH blocks are above the subsegment address due to VS/LFH subsegment headers being located before the VS/LFH blocks. If `UserAddress` points to a backend block, backend freeing is called.

Finally, if the subsegment is an LFH subsegment, LFH freeing is called. Otherwise, VS freeing is called. If VS freeing is called, and if the returned `LfhBlockSize` (equivalent to the block size of the freed VS block minus `0x10`) is serviceable by LFH, the usage counter of the LFH bucket corresponding to `LfhBlockSize` is updated.

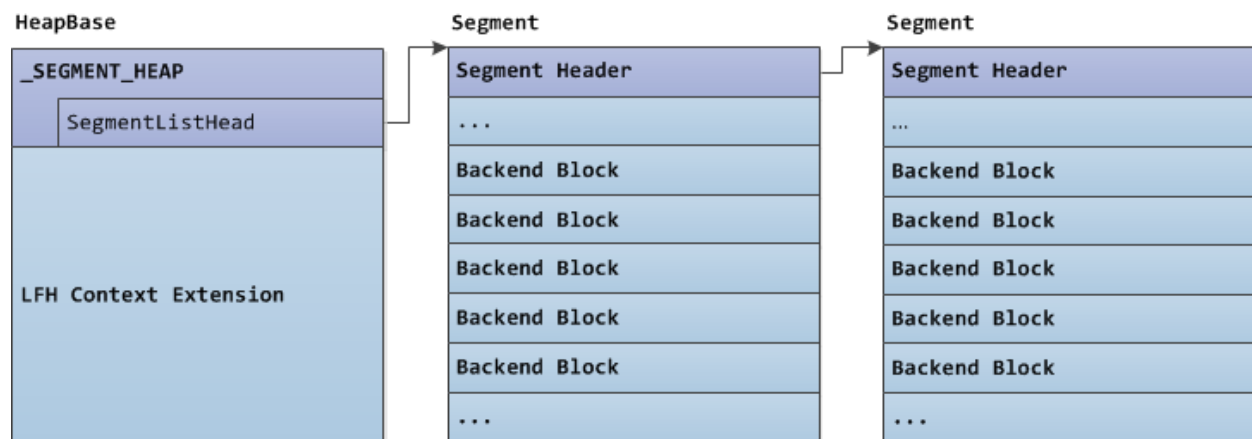
Note that the logic starting where the subsegment of `UserAddress` is derived is actually in a separate `RtlHpSegFree()` function, it was inlined in the diagram for brevity. Also, the diagram only shows the freeing logic of `RtlHpFreeHeap()`, its other functionalities were not included.

## 2.3. BACKEND ALLOCATION

The backend is used for allocations with sizes 131,073 (`0x20001`) to 520,192 (`0x7F000`) bytes. Backend blocks have a page size granularity and each does not have a block header at the beginning. In addition to allocating backend blocks, the backend is also used by the VS and LFH component for the creation of VS/LFH subsegments (special types of backend blocks) where VS/LFH blocks are allocated from.

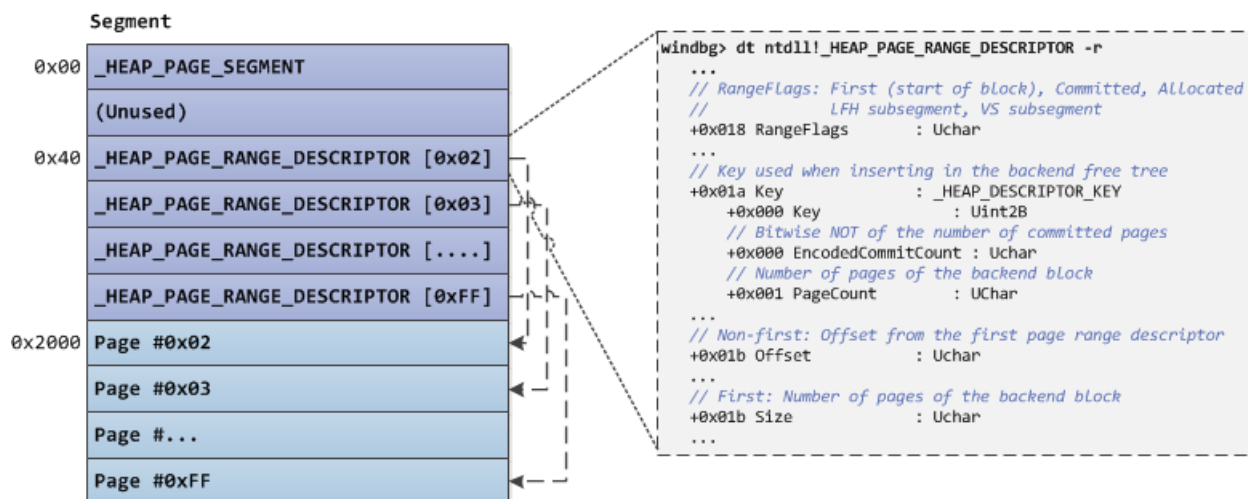
### Segment Structure

The backend operates on segment structures which are 1MB (`0x100000`) blocks of virtual memory allocated via `NtAllocateVirtualMemory()`. The segments are tracked via the `SegmentListHead` field in the `HeapBase`:



The first `0x2000` bytes of a segment is used for the segment header, while the rest is used for the allocation of backend blocks. Initially, the first `0x2000` plus an initial commit size of the segment is committed, while the rest are in the reserve state and are committed and decommitted as needed.

The segment header consists of an array of 256 page range descriptors that describe the allocation status of each page in the segment. Since the data portion of the segment starts at offset `0x2000`, the first page range descriptor is repurposed to store the `_HEAP_PAGE_SEGMENT` structure and the second page range descriptor is unused.



### \_HEAP\_PAGE\_SEGMENT Structure

As mentioned, the first page range descriptor is repurposed to store the **\_HEAP\_PAGE\_SEGMENT** structure which has the following fields:

```

windbg> dt ntdll!_HEAP_PAGE_SEGMENT
+0x000 ListEntry      : _LIST_ENTRY
+0x010 Signature      : UInt8B

```

- **ListEntry** - Each segment is a node of the heap's segments linked list (**HeapBase.SegmentListHead**).
- **Signature** - Used for verifying if an address is part of a segment. This field is computed via:  $(\text{SegmentAddress} \gg 0x14) \wedge \text{RtlpHeapKey} \wedge \text{HeapBase} \wedge 0xA2E64EADA2E64EAD$ .

### \_HEAP\_PAGE\_RANGE\_DESCRIPTOR Structure

Also mentioned are page range descriptors that describe the status of each page of the segment. Since a backend block can span multiple pages (a page range), the page range descriptor for the first page of the backend block is marked as "first", and therefore, will have additional fields set.

```

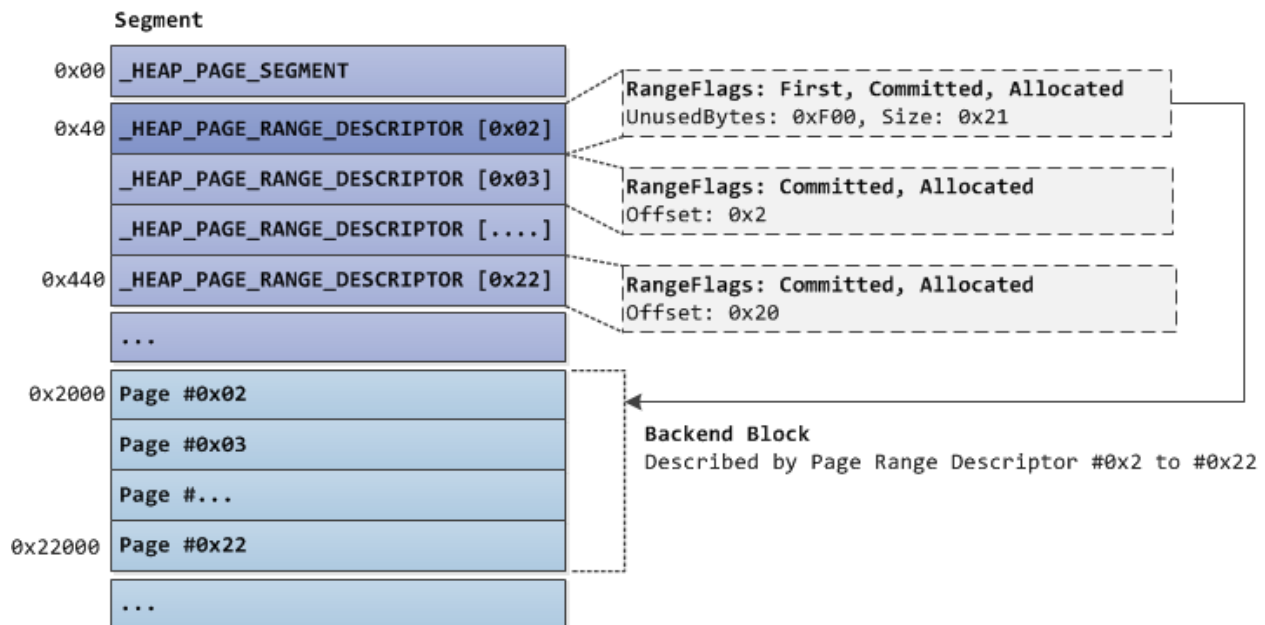
windbg> dt ntdll!_HEAP_PAGE_RANGE_DESCRIPTOR -r
+0x000 TreeNode       : _RTL_BALANCED_NODE
+0x000 TreeSignature  : UInt4B
+0x004 ExtraPresent    : Pos 0, 1 Bit
+0x004 Spare0         : Pos 1, 15 Bits
+0x006 UnusedBytes    : UInt2B
+0x018 RangeFlags     : UChar
+0x019 Spare1         : UChar
+0x01a Key            : _HEAP_DESCRIPTOR_KEY
+0x000 Key            : UInt2B
+0x000 EncodedCommitCount : UChar
+0x001 PageCount      : UChar
+0x01a Align          : UChar
+0x01b Offset         : UChar
+0x01b Size           : UChar

```

- **TreeNode** - "first" page range descriptors of free backend blocks are nodes of the backend free tree (**HeapBase.FreePageRanges**).
- **UnusedBytes** - For "first" page range descriptors. The difference between **UserSize** and the block size.
- **RangeFlags** - Bit field representing the type of the backend block and the state of the page represented by the page range descriptor.

- 0x01: PAGE\_RANGE\_FLAGS\_LFH\_SUBSEGMENT. For “first” page range descriptors. Backend block is an LFH subsegment.
- 0x02: PAGE\_RANGE\_FLAGS\_COMMITTED. Page is committed.
- 0x04: PAGE\_RANGE\_FLAGS\_ALLOCATED. Page is allocated.
- 0x08: PAGE\_RANGE\_FLAGS\_FIRST. Page range descriptor is marked as “first”.
- 0x20: PAGE\_RANGE\_FLAGS\_VS\_SUBSEGMENT. For “first” page range descriptors. Backend block is a VS subsegment.
- Key - For “first” page range descriptors of free backend blocks. This is used when a free backend block is inserted to the backend free tree.
  - Key - WORD-sized key used for the backend free tree, high BYTE is the PageCount field and the low BYTE is the EncodedCommitCount field (see below).
  - EncodedCommitCount - Bitwise NOT of the number of committed pages in the backend block. The larger number of committed pages the free backend block has, the lower EncodedCommitCount will be.
  - PageCount - Number pages of the backend block.
- Offset - For non-“first” page range descriptors. Offset of the page range descriptor from the “first” page range descriptor.
- Size - For “first” page range descriptors. Same value as Key .PageCount (overlapping fields).

Below is an illustration of a 131,328 bytes (0x20100) busy backend block and the corresponding page range descriptors, the “first” page range descriptor is highlighted:



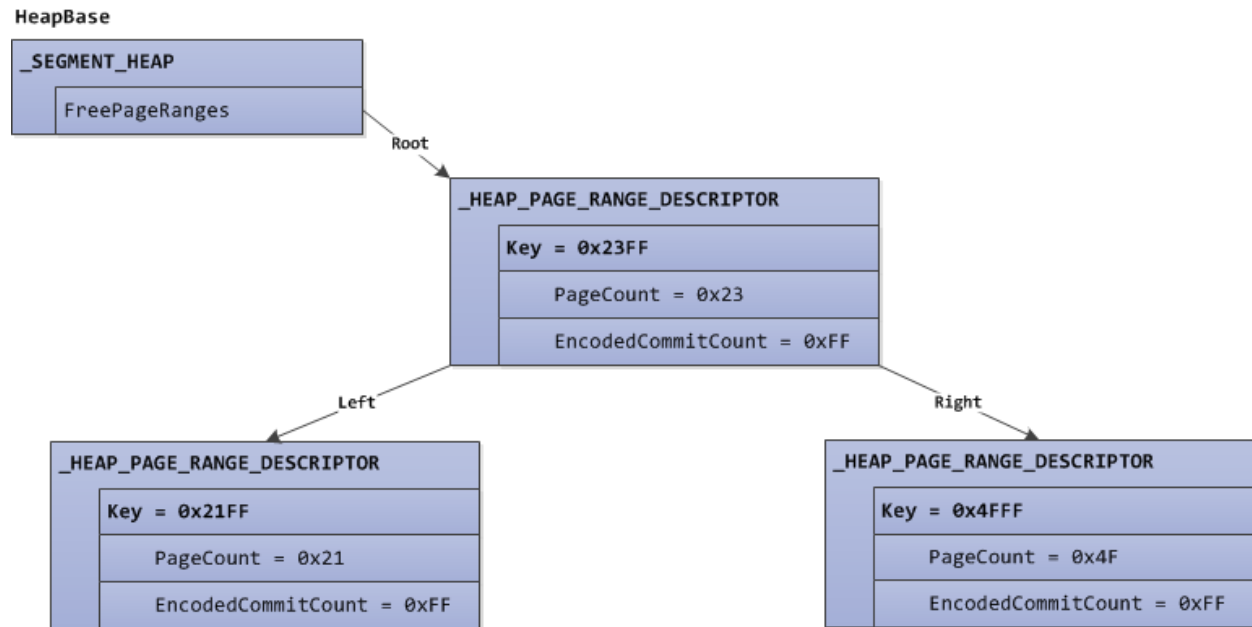
Note that because the page range descriptors that describe the backend blocks are stored at the top of the segment, it means that each backend block does not have a block header at the beginning.

## Backend Free Tree

Backend allocation and freeing uses the backend free tree for finding and storing information on free backend blocks.

The root of the backend free tree is stored in the `HeapBase.FreePageRanges` and each tree node is the “first” page range descriptor of free backend blocks. The key used for inserting nodes in the backend free tree is the “first” page range descriptor’s `Key.Key` field (see details of `Key.Key` in the previous subsection).

Below is an illustration of a backend free tree in which there are three free backend blocks with sizes `0x21000`, `0x23000` and `0x4F000`; all pages of the free blocks are decommitted (`Key.EncodedCommitCount` is `0xFF`):



## Backend Allocation

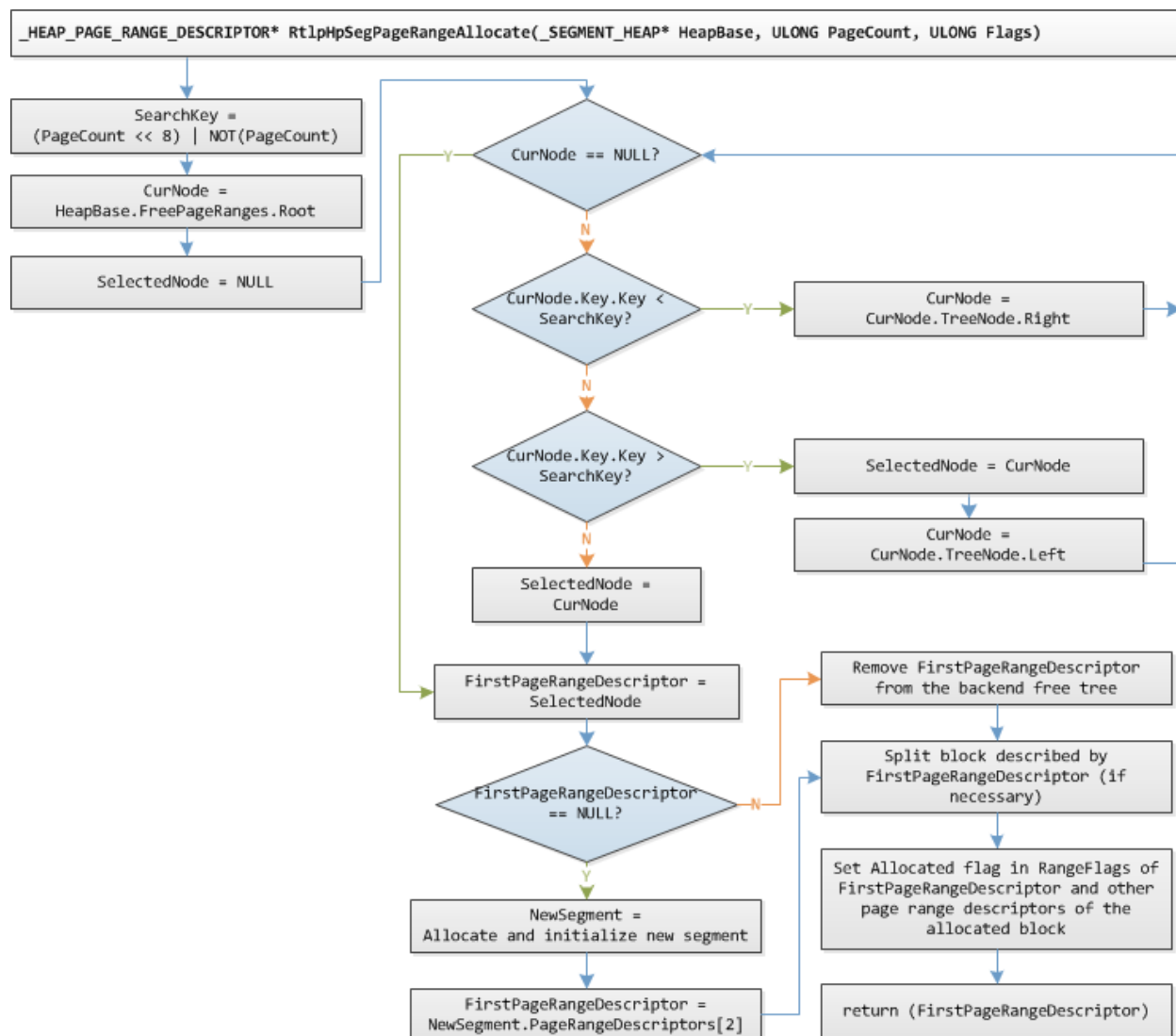
Backend allocation is performed via `RtlpHpSegAlloc()` which has the following function signature:

```
PVOID RtlpHpSegAlloc(_SEGMENT_HEAP* HeapBase, SIZE_T UserSize, SIZE_T AllocSize, ULONG Flags)
```

`RtlpHpSegAlloc()` first calls `RtlpHpSegPageRangeAllocate()` to allocate a backend block. `RtlpHpSegPageRangeAllocate()` returns the “first” page range descriptor of the allocated backend block and `RtlpHpSegAlloc()` converts it to the actual backend block address (`UserAddress`) which it will then use as the return value.

As mentioned, `RtlpHpSegPageRangeAllocate()` is the function called by `RtlpHpSegAlloc()` to find and allocate a backend block. The function accepts the number of pages to allocate and then returns the “first” page range descriptor of the allocated backend block.

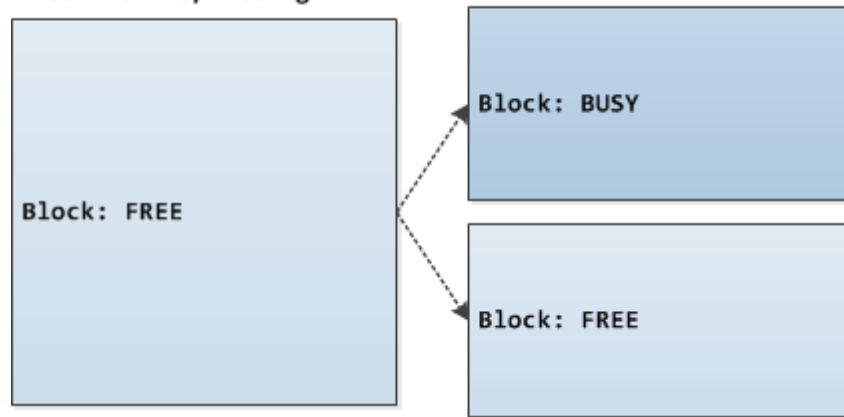
The diagram below shows the logic of `RtlpHpSegPageRangeAllocate()`:



`RtlpHpSegPageRangeAllocate()` first traverses backend free tree to find a free backend block that can fit the allocation. The search key used for finding a free backend block is a WORD-sized value in which the high BYTE is the requested number of pages and the low BYTE is the bitwise NOT of the number of requested pages which means that a best-fit search is conducted with the most committed block given preference. If any of the free backend blocks cannot fit the allocation, a new segment is created.

Since the selected free backend block can have more pages than the requested number of pages, the free block is first split if necessary via `RtlpHpSegPageRangeSplit()` and the “first” page range descriptor of the resulting remaining free block is inserted to the backend free tree.



**Free Block Splitting**

Finally, the RangeFlags field of the block's page range descriptors are updated (PAGE\_RANGE\_FLAGS\_ALLOCATED bit is set) to mark the block's pages as allocated.

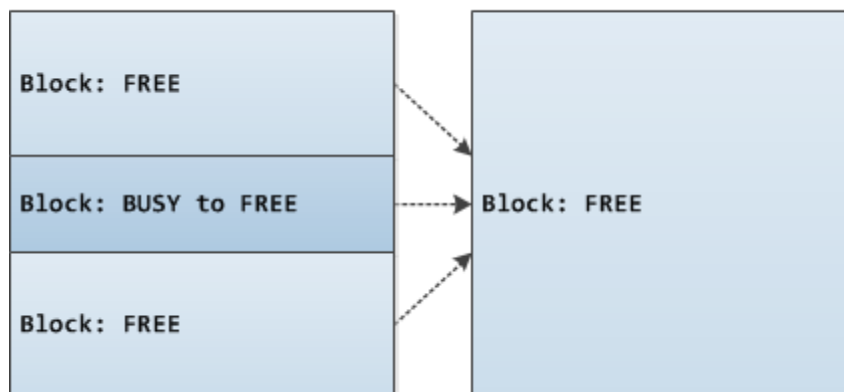
**Backend Freeing**

Backend freeing is performed via `RtlpHpSegPageRangeShrink()` which has the following function signature:

```
BOOLEAN RtlpHpSegPageRangeShrink(_SEGMENT_HEAP* HeapBase,
                                   _HEAP_PAGE_RANGE_DESCRIPTOR* FirstPageRangeDescriptor,
                                   ULONG NewPageCount, ULONG Flags)
```

Where `FirstPageRangeDescriptor` is the "first" page range descriptor of the to-be-freed backend block and `NewPageCount` is 0 which means to free the block.

`RtlpHpSegPageRangeShrink()` first clears the PAGE\_RANGE\_FLAGS\_ALLOCATED bit in the RangeFlags field of all (except the "first") page range descriptors that describes the to-be-freed backend block. It then calls `RtlpHpSegPageRangeCoalesce()` which coalesces the to-be-freed backend block with neighboring (before and after) free backend blocks and then clears the PAGE\_RANGE\_FLAGS\_ALLOCATED bit in the RangeFlags field of the "first" page range descriptor of the to-be-freed block.

**Free Blocks Coalescing**

The "first" page range descriptor of the resulting coalesced block is then inserted to the backend free tree making the coalesced free block available for allocations.

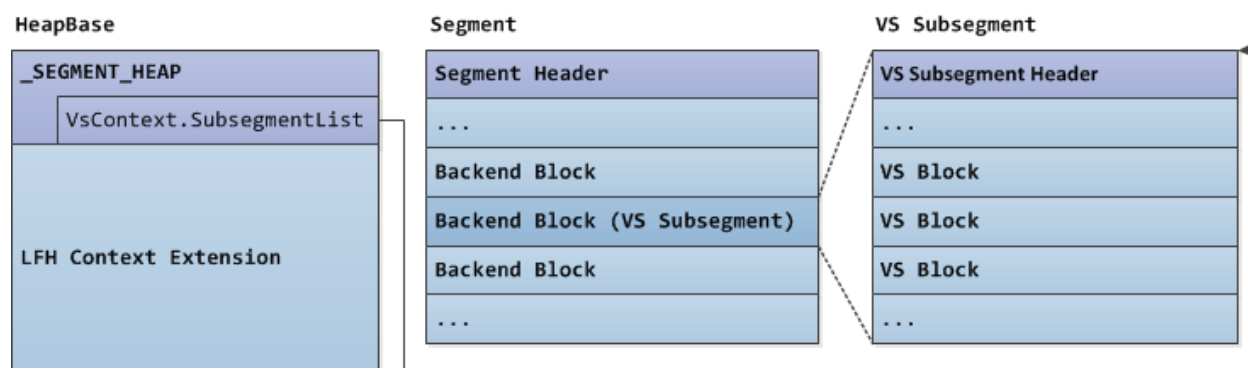
## 2.4. VARIABLE SIZE ALLOCATION

Variable size (VS) allocation is used for allocations with sizes 1 to 131,072 (0x20000) bytes. VS blocks have 16 (0x10) bytes granularity and each has a block header at the beginning.

### VS Subsegments

The VS allocation component relies on the backend for creating the VS subsegments where VS blocks are allocated from. A VS subsegment is a special type of a backend block in which the RangeFlags of the “first” page range descriptor has the PAGE\_RANGE\_FLAGS\_VS\_SUBSEGMENT (0x20) bit set.

Below is the illustration of the relationship of the HeapBase, a segment and a VS subsegment:



### \_HEAP\_VS\_CONTEXT Structure

The VS context structure tracks the free VS blocks, VS subsegments, and other information related to the VS allocation state. It is stored in the VsContext field in the HeapBase and has the following fields:

```
windbg> dt ntdll!_HEAP_VS_CONTEXT
+0x000 Lock           : _RTL_SRWLOCK
+0x008 FreeChunkTree  : _RTL_RB_TREE
+0x018 SubsegmentList : _LIST_ENTRY
+0x028 TotalCommittedUnits : UInt8B
+0x030 FreeCommittedUnits : UInt8B
+0x038 BackendCtx     : Ptr64 Void
+0x040 Callbacks      : _HEAP_SUBALLOCATOR_CALLBACKS
```

- FreeChunkTree - RB tree of free VS blocks.
- SubsegmentList - Linked list of all VS subsegments.
- BackendCtx - Pointer to the HeapBase.
- Callbacks - Encoded (see 3.5) callbacks used for the management of VS subsegments.

### \_HEAP\_VS\_SUBSEGMENT Structure

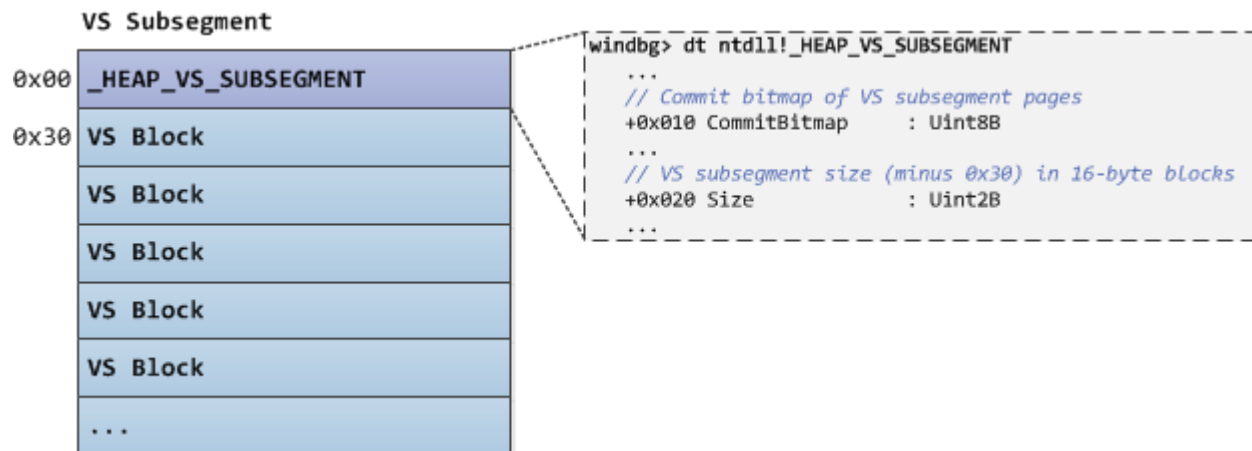
VS subsegments are where VS blocks are allocated from. VS subsegments are allocated and initialized via `RtlpHpVsSubsegmentCreate()` and will have the following **\_HEAP\_VS\_SUBSEGMENT** structure as the header:

```
windbg> dt ntdll!_HEAP_VS_SUBSEGMENT
+0x000 ListEntry      : _LIST_ENTRY
+0x010 CommitBitmap   : UInt8B
+0x018 CommitLock     : _RTL_SRWLOCK
+0x020 Size           : UInt2B
+0x022 Signature      : UInt2B
```

- Listentry - Each VS subsegment is a node of VS subsegments linked list (`VsContext.SubsegmentList`).

- CommitBitmap - Commit bitmap of VS subsegment pages.
- Size - Size of the VS subsegment (minus 0x30 for the VS subsegment header) in 16-byte blocks.
- Signature - Used for checking if the VS subsegment is corrupted. Computed via:  $\text{Size} \wedge 0xABED$ .

Below is an illustration of a VS subsegment:



The `_HEAP_VS_SUBSEGMENT` structure is at offset 0x00 while the VS blocks start at offset 0x30.

### `_HEAP_VS_CHUNK_HEADER` Structure

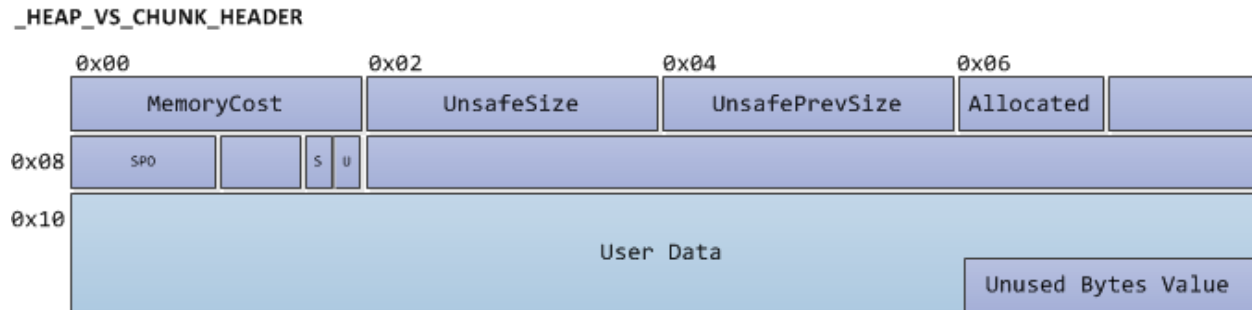
Busy VS blocks have a 16-byte (0x10) header which has following structure:

```
windbg> dt ntdll!_HEAP_VS_CHUNK_HEADER -r
+0x000 Sizes                : _HEAP_VS_CHUNK_HEADER_SIZE
+0x000 MemoryCost           : Pos 0, 16 Bits
+0x000 UnsafeSize           : Pos 16, 16 Bits
+0x004 UnsafePrevSize       : Pos 0, 16 Bits
+0x004 Allocated            : Pos 16, 8 Bits
+0x000 KeyUShort            : UInt2B
+0x000 KeyULong             : UInt4B
+0x000 HeaderBits           : UInt8B
+0x008 EncodedSegmentPageOffset : Pos 0, 8 Bits
+0x008 UnusedBytes          : Pos 8, 1 Bit
+0x008 SkipDuringWalk       : Pos 9, 1 Bit
+0x008 Spare                : Pos 10, 22 Bits
+0x008 AllocatedChunkBits   : UInt4B
```

- Sizes - Encoded (see 3.6) QWORD-sized substructure that encapsulates important size information:
  - MemoryCost - Used in free VS blocks. A value computed based on how large the committed portion of the block is. The larger the portion of the block is already committed, the lower the memory cost is. This means that if a low memory cost block is selected for allocation, the smaller amount of memory needs to be committed.
  - UnsafeSize - Size of the VS block (includes the block header) in 16-byte blocks.
  - UnsafePrevSize - Size of the previous VS block (includes the block header) in 16-byte blocks.
  - Allocated - Block is busy if value is not zero.
  - KeyULong - Used in free VS blocks. A DWORD-sized value used when inserting the free VS block to the VS free tree. The high WORD is the UnsafeSize field and the low WORD is MemoryCost field.

- EncodedSegmentPageOffset - Offset of the block from the start of the VS subsegment in pages. Encoded via:  $\text{SegmentPageOffset} \wedge \text{LOW\_8\_BITS}(\text{RtlpLFHKey}) \wedge \text{LOW\_8\_BITS}(\text{BlockAddress})$ .
- UnusedBytes - Flag that indicates whether the block has unused bytes which is the difference between the UserSize and the total block size (minus 0x10 bytes for the header). If this flag is set, the last two bytes of the VS block is treated as a 16 bit low endian value. If the number of unused bytes is 1, the high bit of this 16 bit value is set and the rest of the bits are unused, otherwise, the high bit is clear and the low 13 bits are used to store the unused bytes value.

Below is an illustration of a busy VS block (note that the first 8 bytes is encoded):

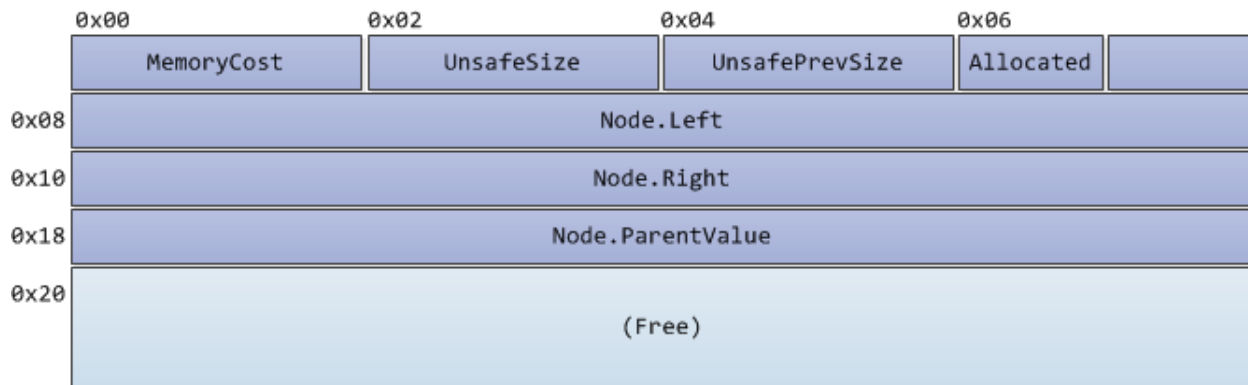


### \_HEAP\_VS\_CHUNK\_FREE\_HEADER Structure

Free VS blocks have a 32-byte (0x20) header where the first 0x8 bytes is the first 0x8 bytes of the `_HEAP_VS_CHUNK_HEADER` structure. Starting at offset +0x08 is the Node field which acts as a node in the VS free tree (`VsContext.FreeChunkTree`).

```
windbg> dt ntdll!_HEAP_VS_CHUNK_FREE_HEADER -r
+0x000 Header : _HEAP_VS_CHUNK_HEADER
+0x000 Sizes : _HEAP_VS_CHUNK_HEADER_SIZE
+0x000 MemoryCost : Pos 0, 16 Bits
+0x000 UnsafeSize : Pos 16, 16 Bits
+0x004 UnsafePrevSize : Pos 0, 16 Bits
+0x004 Allocated : Pos 16, 8 Bits
+0x000 KeyUShort : Uint2B
+0x000 KeyULong : Uint4B
+0x000 HeaderBits : Uint8B
+0x008 EncodedSegmentPageOffset : Pos 0, 8 Bits
+0x008 UnusedBytes : Pos 8, 1 Bit
+0x008 SkipDuringWalk : Pos 9, 1 Bit
+0x008 Spare : Pos 10, 22 Bits
+0x008 AllocatedChunkBits : Uint4B
+0x000 OverlapsHeader : Uint8B
+0x008 Node : _RTL_BALANCED_NODE
```

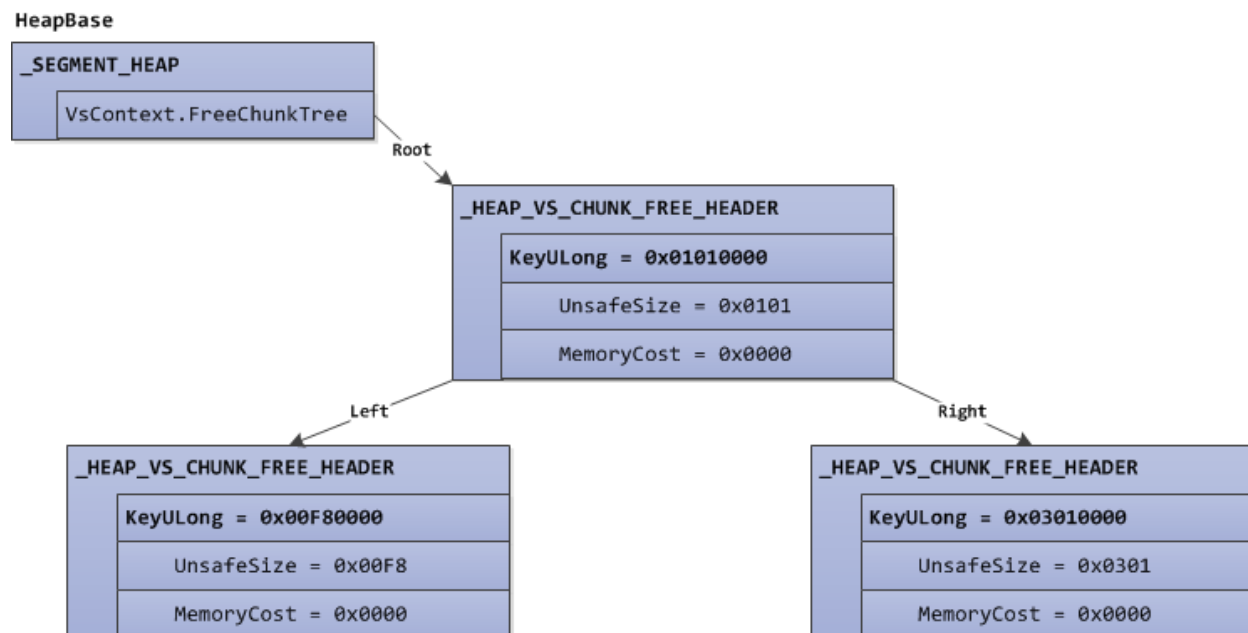
Below is an illustration of a free VS block (note that the first 8 bytes is encoded):

**\_HEAP\_VS\_CHUNK\_FREE\_HEADER****VS Free Tree**

VS allocation and freeing uses the VS free tree for finding and storing information on free VS blocks.

The root of the VS free tree is stored in `VsContext.FreeChunkTree` where each node is the `Node` field of free VS blocks. The key used for inserting nodes in the VS free tree is the free VS block's `Header.Sizes.KeyULong` field (`Sizes.KeyULong` is discussed in the “`_HEAP_VS_CHUNK_HEADER` Structure” subsection above).

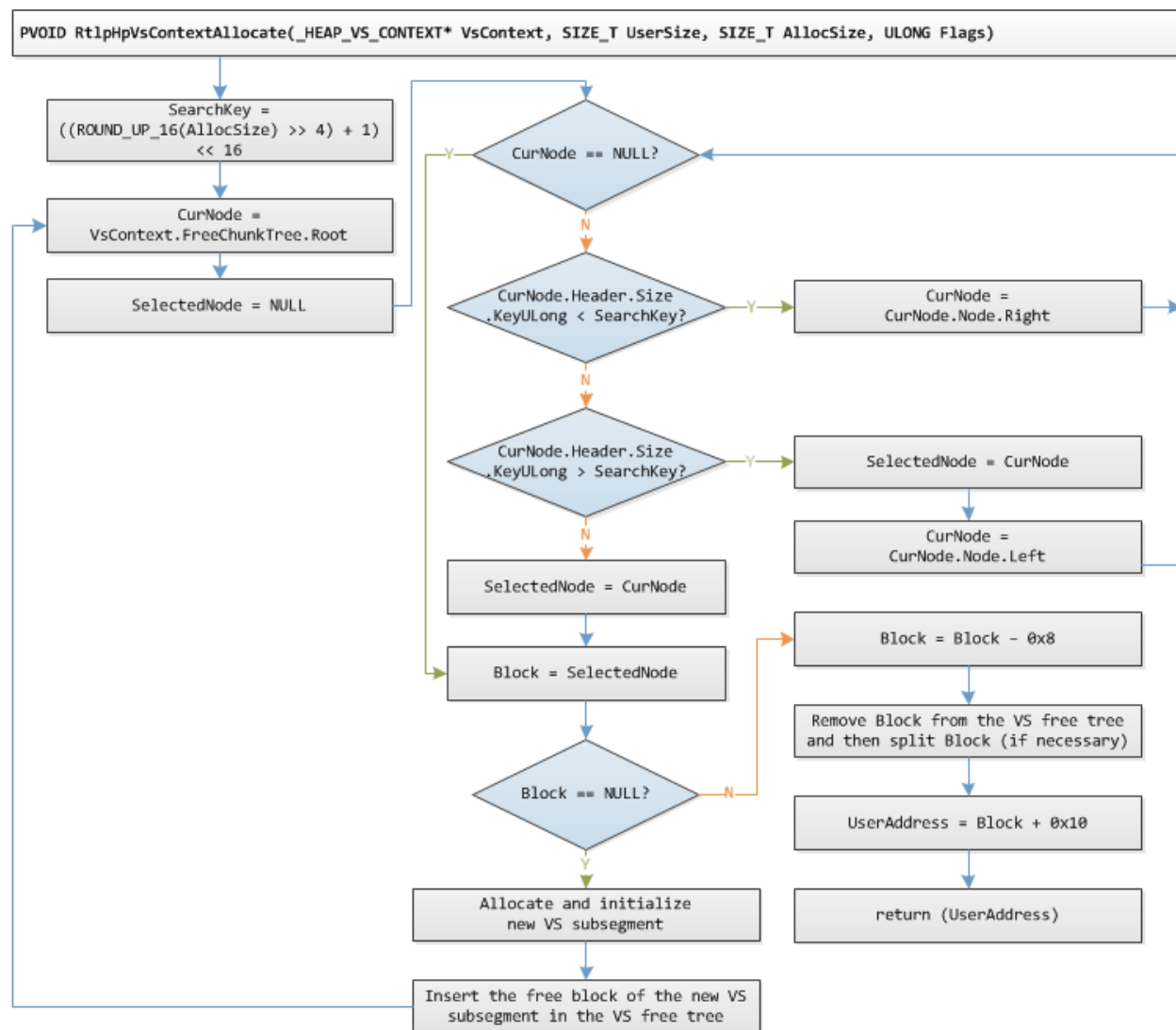
Below is an illustration of a VS free tree in which there are three free VS blocks with block sizes 0xF80, 0x1010, 0x3010 bytes; all portions of the free blocks are committed (`MemoryCost` is 0x0000):

**VS Allocation**

VS allocation is performed via `RtlpHpVsContextAllocate()` which has the following function signature:

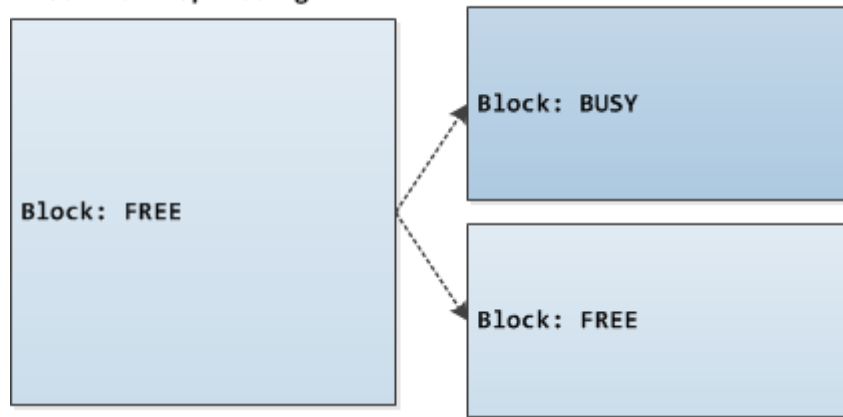
```
PVOID RtlpHpVsContextAllocate(_HEAP_VS_CONTEXT* VsContext, SIZE_T UserSize, SIZE_T AllocSize,
                              ULONG Flags)
```

The diagram bellows show the logic of `RtlpHpVsContextAllocate()`:



`RtlHpVsContextAllocate()` first traverses VS free tree to find a free VS block that can fit the allocation. The search key used for finding a free VS block is a DWORD-sized value in which the high WORD is the number of 16-byte blocks that can accommodate `AllocSize` plus one (for the block header) and the low WORD is 0 (for `MemoryCost`) which means that a best-fit search is conducted with the free VS block with the lowest memory cost (most portion of the block is committed) is given preference. If any of the free VS blocks cannot fit the allocation, a new VS subsegment is created.

Since the size of the selected free VS block can be larger than the block size that can accommodate `AllocSize`, large free VS blocks are split unless the block size of the resulting remaining block will be less than 0x20 bytes (the size of a free VS block header).

**Free Block Splitting**

The free VS block splitting is performed by `RtlpHpVsChunkSplit()`. `RtlpHpVsChunkSplit()` is also the function that removes the free VS block from VS free tree and also inserts the resulting remaining free block to the VS free tree if block splitting occurred.

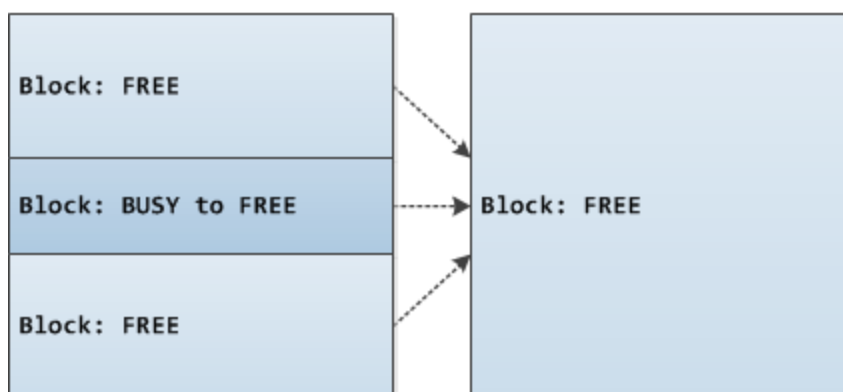
**VS Freeing**

VS freeing is performed via `RtlpHpVsContextFree()` which has the following function signature:

```
BOOLEAN RtlpHpVsContextFree(_HEAP_VS_CONTEXT* VsContext, _HEAP_VS_SUBSEGMENT* VsSubsegment,
                             PVOID UserAddress, ULONG Flags, ULONG* LfhBlockSize)
```

Where `UserAddress` is the address of the to-be-freed VS block and `LfhBlockSize` will become the block size of the to-be-freed VS block minus `0x10` (busy VS block header size). `LfhBlockSize` will be used by the caller of `RtlpHpVsContextFree()` in updating the LFH bucket usage counter corresponding to `LfhBlockSize`.

`RtlpHpVsContextFree()` first checks if the VS block is indeed allocated by checking the `Allocated` field in the block's header. It will then call `RtlpHpVsChunkCoalesce()` which will coalesce the to-be-freed block with neighboring free blocks (before and after).

**Free Blocks Coalescing**

Finally, the coalesced free block is inserted to VS free tree making it available for allocation.

## 2.5. LOW FRAGMENTATION HEAP

The Low Fragmentation Heap (LFH) is used for allocations with sizes 1 to 16,368 (0x3FF0) bytes. Similar to the LFH in the NT Heap, the LFH in the Segment Heap prevents fragmentation by using a bucketing scheme which causes similarly-sized blocks to be allocated from larger pre-allocated blocks of memory.

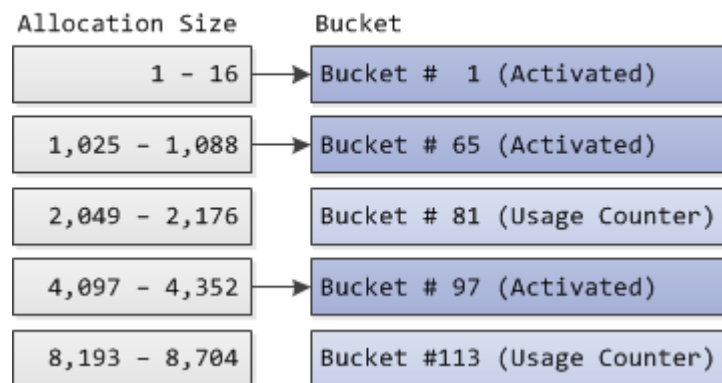
Below is a table listing the different LFH buckets, the allocation sizes distributed to the buckets, and the corresponding granularity of the buckets:

Bucket	Allocation Size	Granularity (Block Size)
1 – 64	1 – 1,024 bytes (0x1 – 0x400)	16 bytes
65 – 80	1,025 – 2,048 bytes (0x401 – 0x800)	64 bytes
81 – 96	2,049 – 4,096 bytes (0x801 – 0x1000)	128 bytes
97 – 112	4,097 – 8,192 bytes (0x1001 – 0x2000)	256 bytes
113 – 128	8,193 – 16,368 bytes (0x2001 – 0x3FF0)	512 bytes

The LFH buckets are only activated (enabled) if their corresponding allocation sizes are detected to be popular. LFH bucket activation and usage counter will be further discussed later.

Below is an illustration of a few activated buckets and non-activated buckets including their corresponding allocation sizes:

**Example Activated Buckets and Bucket Usage Counters**



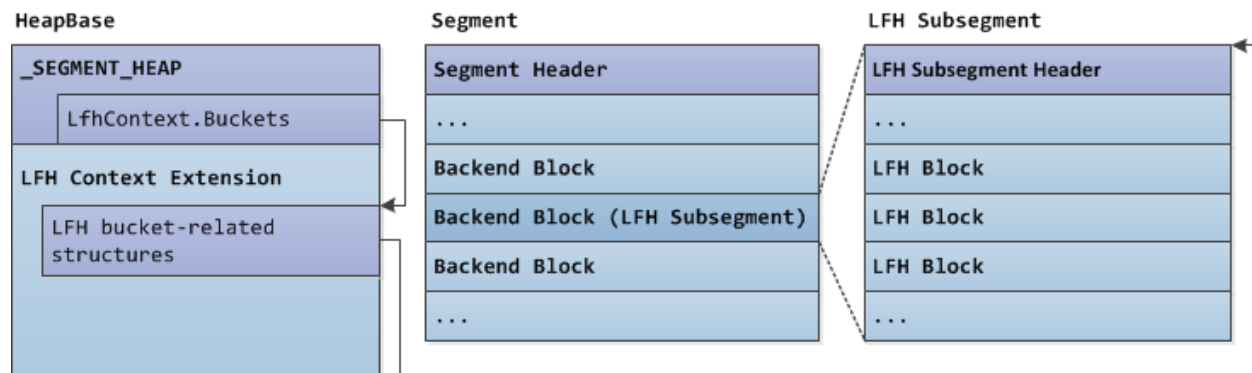
Buckets #1, #65 and #97 are activated and therefore the allocation requests for the corresponding allocation sizes will be serviced via the LFH buckets. Buckets #81 and #113 are still not activated and therefore the allocation requests for the corresponding allocation sizes will cause the usage counter for the LFH buckets to be updated. If the usage counter reaches a particular value after the update, the bucket will be activated and the allocation will be serviced via the LFH bucket, otherwise, the allocation request will eventually be passed to the VS allocation component.



## LFH Subsegments

The LFH component relies on the backend for creating the LFH subsegments where LFH blocks are allocated from. An LFH subsegment is a special type of a backend block in which the corresponding “first” page range descriptor descriptor’s RangeFlags field has the PAGE\_RANGE\_FLAGS\_LFH\_SUBSEGMENT (0x01) bit set.

Below is the illustration of the relationship of the HeapBase, a segment and an LFH subsegment:



## \_HEAP\_LFH\_CONTEXT Structure

The LFH context tracks the LFH buckets, LFH bucket usage counters and other information related to the LFH state. It is stored in the LfhContext field in the HeapBase.

```
windbg> dt ntdll!_HEAP_LFH_CONTEXT -r
+0x000 BackendCtx      : Ptr64 Void
+0x008 Callbacks       : _HEAP_SUBALLOCATOR_CALLBACKS
+0x030 SubsegmentCreationLock : _RTL_SRWLOCK
+0x038 MaxAffinity     : UChar
+0x040 AffinityModArray : Ptr64 UChar
+0x050 SubsegmentCache : _HEAP_LFH_SUBSEGMENT_CACHE
+0x000 SLists          : [7] _SLIST_HEADER
+0x0c0 Buckets         : [129] Ptr64 _HEAP_LFH_BUCKET
```

- BackendCtx - Pointer to the HeapBase.
- Callbacks - Callbacks for managing LFH subsegments and the LFH context extension.
- MaxAffinity - Maximum number of affinity slots that can be created.
- SubsegmentCache - Tracks cached (unused) LFH subsegments.
- Buckets - Array of pointers to the LFH buckets. If the bucket is activated, bit 0 of the pointer is clear and the pointer points to a `_HEAP_LFH_BUCKET` structure. Otherwise (if bit 0 is set), the pointer is actually a `_HEAP_LFH_ONDEMAND_POINTER` structure which is used for tracking LFH bucket usage.

The reserved virtual memory found after the `_SEGMENT_HEAP` structure in the HeapBase, called the LFH context extension, is dynamically committed to additionally store LFH bucket-related structures for dynamically activated LFH buckets (see previous illustration).

## \_HEAP\_LFH\_ONDEMAND\_POINTER Structure

As mentioned above, if the LFH bucket is not activated, the entry for the bucket in `LfhContext.Buckets` will be a usage counter with following structure:

```
windbg> dt ntdll!_HEAP_LFH_ONDEMAND_POINTER
+0x000 Invalid          : Pos 0, 1 Bit
+0x000 AllocationInProgress : Pos 1, 1 Bit
```

```
+0x000 Spare0      : Pos 2, 14 Bits
+0x002 UsageData   : Uint2B
+0x000 AllBits     : Ptr64 Void
```

- Invalid - Marker to determine if this pointer is an invalid `_HEAP_LFH_BUCKET` pointer (lowest bit set), and therefore, the structure is a bucket usage counter.
- UsageData – WORD-sized value describing the usage of the LFH bucket. The value represented by bit 0 to 4 is the number of active allocations for the bucket's allocations size, it is incremented on allocations and decremented on frees. The value represented by bit 5 to 15 is the number of allocation requests for the bucket's allocation size, it is incremented on allocations.

## `_HEAP_LFH_BUCKET` Structure

If the bucket is activated, the entry for the bucket in `LfhContext.Buckets` is a pointer to a `_HEAP_LFH_BUCKET` structure which has the following structure:

```
windbg> dt ntdll!_HEAP_LFH_BUCKET
+0x000 State      : _HEAP_LFH_SUBSEGMENT_OWNER
+0x038 TotalBlockCount : Uint8B
+0x040 TotalSubsegmentCount : Uint8B
+0x048 ReciprocalBlockSize : Uint4B
+0x04c Shift      : UChar
+0x050 AffinityMappingLock : _RTL_SRWLOCK
+0x058 ContentionCount : Uint4B
+0x060 ProcAffinityMapping : Ptr64 UChar
+0x068 AffinitySlots : Ptr64 Ptr64 _HEAP_LFH_AFFINITY_SLOT
```

- TotalBlockCount - Total number of LFH blocks in all LFH subsegments related to the bucket.
- TotalSubsegmentCount - Total number of LFH subsegments related to the bucket.
- ContentionCount - Number of contentions identified when allocating blocks from the LFH subsegments. Every time this global variable reaches `RtlpHpLfhContentionLimit`, a new affinity slot is created for the requesting thread's processor.
- ProcAffinityMapping - Points to an array of BYTE-sized indexes to `AffinitySlots`. This is used for dynamically assigning processors to affinity slots (discussed later). Initially, all are set to 0 which means that all processors are assigned to the initial affinity slot that was created when the bucket is activated.
- AffinitySlots - Pointer to an array of affinity slot pointers (`_HEAP_LFH_AFFINITY_SLOT*`). When the bucket is activated, only one slot is initially created, as more contentions are detected, new affinity slots are created.

## `_HEAP_LFH_AFFINITY_SLOT` Structure

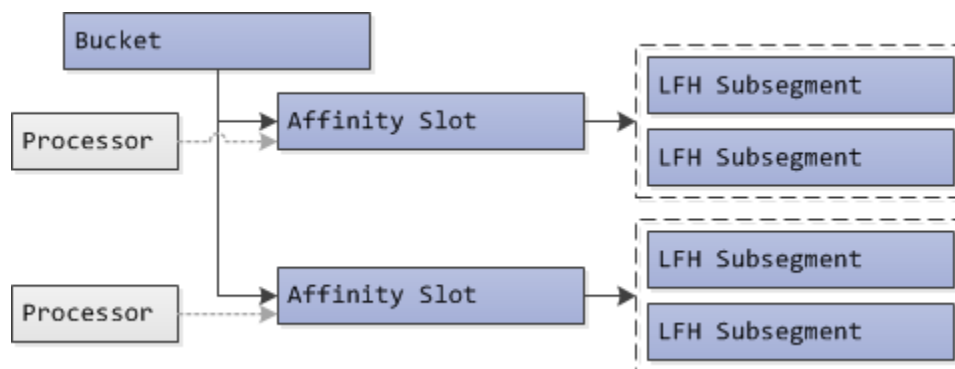
An affinity slot owns the LFH subsegments where LFH blocks are allocated from. Initially, only one affinity slot is created when the bucket is activated and all processors are assigned to the initial affinity slot.

Because only one affinity slot is initially created, it means that all processors will use the same set of LFH subsegments, and therefore, contention can occur. If too many contentions are detected, a new affinity slot is created and the requesting thread's processor is reassigned to the new affinity slot via the `ProcAffinityMapping` field in the bucket.

There is only one field in an affinity slot and its structure will be described next.

```
windbg> dt ntdll!_HEAP_LFH_AFFINITY_SLOT
+0x000 State : _HEAP_LFH_SUBSEGMENT_OWNER
```

Below is an illustration of the relationship between buckets, processors and affinity slots:



### \_HEAP\_LFH\_SUBSEGMENT\_OWNER Structure

The subsegment owner structure is used by the affinity slot (`LfhAffinitySlot.State`) to track the LFH subsegments it owns, it has the following fields:

```

windbg> dt ntdll!_HEAP_LFH_SUBSEGMENT_OWNER
+0x000 IsBucket      : Pos 0, 1 Bit
+0x000 Spare0       : Pos 1, 7 Bits
+0x001 BucketIndex  : UChar
+0x002 SlotCount    : UChar
+0x002 SlotIndex    : UChar
+0x003 Spare1       : UChar
+0x008 AvailableSubsegmentCount : UInt8B
+0x010 Lock         : _RTL_SRWLOCK
+0x018 AvailableSubsegmentList : _LIST_ENTRY
+0x028 FullSubsegmentList : _LIST_ENTRY
  
```

- `AvailableSubsegmentCount` - Number of LFH subsegments in `AvailableSubsegmentList`.
- `AvailableSubsegmentList` - Linked list of LFH subsegments that have free LFH blocks.
- `FullSubsegmentList` - Linked list of LFH subsegments that have no free LFH blocks.

### \_HEAP\_LFH\_SUBSEGMENT Structure

The LFH subsegments are where LFH blocks are allocated from. LFH subsegments are created and initialized via `RtlpHpLfhSubsegmentCreate()` and will have the following `_HEAP_LFH_SUBSEGMENT` structure as the header:

```

windbg> dt ntdll!_HEAP_LFH_SUBSEGMENT -r
+0x000 ListEntry    : _LIST_ENTRY
+0x000 Link         : _SLIST_ENTRY
+0x010 Owner        : Ptr64 _HEAP_LFH_SUBSEGMENT_OWNER
+0x010 DelayFree    : _HEAP_LFH_SUBSEGMENT_DELAY_FREE
    +0x000 DelayFree : Pos 0, 1 Bit
    +0x000 Count     : Pos 1, 63 Bits
    +0x000 AllBits   : Ptr64 Void
+0x018 CommitLock   : _RTL_SRWLOCK
+0x020 FreeCount    : UInt2B
+0x022 BlockCount   : UInt2B
+0x020 InterlockedShort : Int2B
+0x020 InterlockedLong  : Int4B
+0x024 FreeHint     : UInt2B
+0x026 Location     : UChar
+0x027 Spare        : UChar
+0x028 BlockOffsets : _HEAP_LFH_SUBSEGMENT_ENCODED_OFFSETS
    +0x000 BlockSize : UInt2B
    +0x002 FirstBlockOffset : UInt2B
    +0x000 EncodedData : UInt4B
  
```

```
+0x02c CommitUnitShift : UChar
+0x02d CommitUnitCount : UChar
+0x02e CommitStateOffset : Uint2B
+0x030 BlockBitmap      : [1] Uint8B
```

- **Listentry** - Each LFH subsegment is a node to any of the affinity slot's LFH subsegments lists (`LfhAffinitySlot.AvailableSubsegmentList` or `.FullSubsegmentList`).
- **Owner** - Pointer to the affinity slot that owns this LFH subsegment.
- **FreeHint** - Block index of recently allocated or freed LFH block. Used in the allocation algorithm when searching for a free LFH block.
- **Location** - Location of this LFH subsegment in the affinity slot's LFH subsegments lists: 0: `AvailableSubsegmentList`, 1: `FullSubsegmentList`.
- **FreeCount** - Number of free blocks in the LFH subsegment.
- **BlockCount** - Total number of blocks in the LFH subsegment.
- **BlockOffsets** - Encoded (see 3.7) DWORD-sized substructure containing the size of each LFH block and the offset of the first LFH block in the LFH subsegment.
  - **BlockSize** - Size of each LFH block in the LFH subsegment.
  - **FirstBlockOffset** - Offset the first LFH block from the LFH subsegment.
- **CommitStateOffset** - Offset of the commit state array from the LFH subsegment. An LFH subsegment is divided into multiple "commit portions"; commit state is an array of WORD-sized values that represent the commit state of each these "commit portions".
- **BlockBitmap** - Each LFH block is represented by 2 bits in this block bitmap (further discussed below).

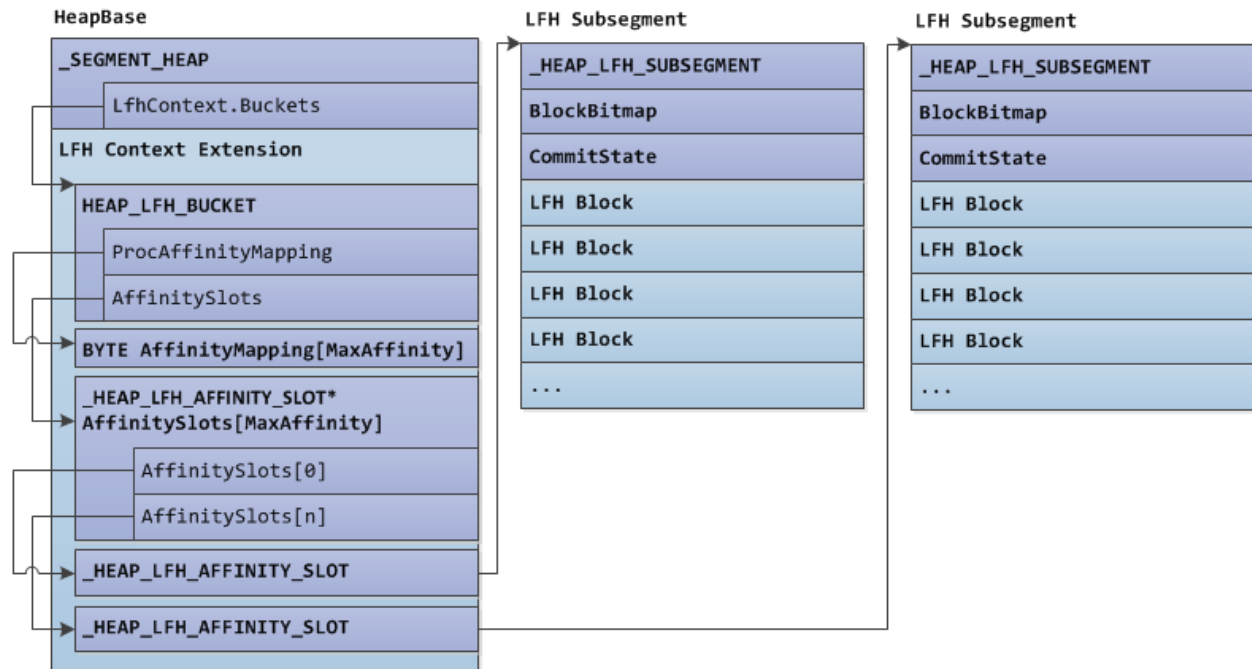
Below is an illustration of an LFH subsegment:

LFH Subsegment

<b>_HEAP_LFH_SUBSEGMENT</b>
<b>BlockBitmap</b>
<b>CommitState</b>
<b>LFH Block</b>
<b>LFH Block</b>
<b>LFH Block</b>
<b>LFH Block</b>
...

```
windbg> dt ntdll!_HEAP_LFH_SUBSEGMENT -r
...
// Number of free LFH blocks
+0x020 FreeCount      : Uint2B
...
// Total number of LFH blocks
+0x022 BlockCount     : Uint2B
...
// Size of each block and offset of first block
// from the LFH subsegment (both encoded)
+0x028 BlockOffsets   : _HEAP_LFH_SUBSEGMENT_ENCODED_OFFSETS
+0x000 BlockSize      : Uint2B
+0x002 FirstBlockOffset : Uint2B
...
// Block bitmap: 2 status bits per LFH block
+0x030 BlockBitmap    : [1] Uint8B
...
```

And below is the illustration of the different data structures and fields that support the LFH component:



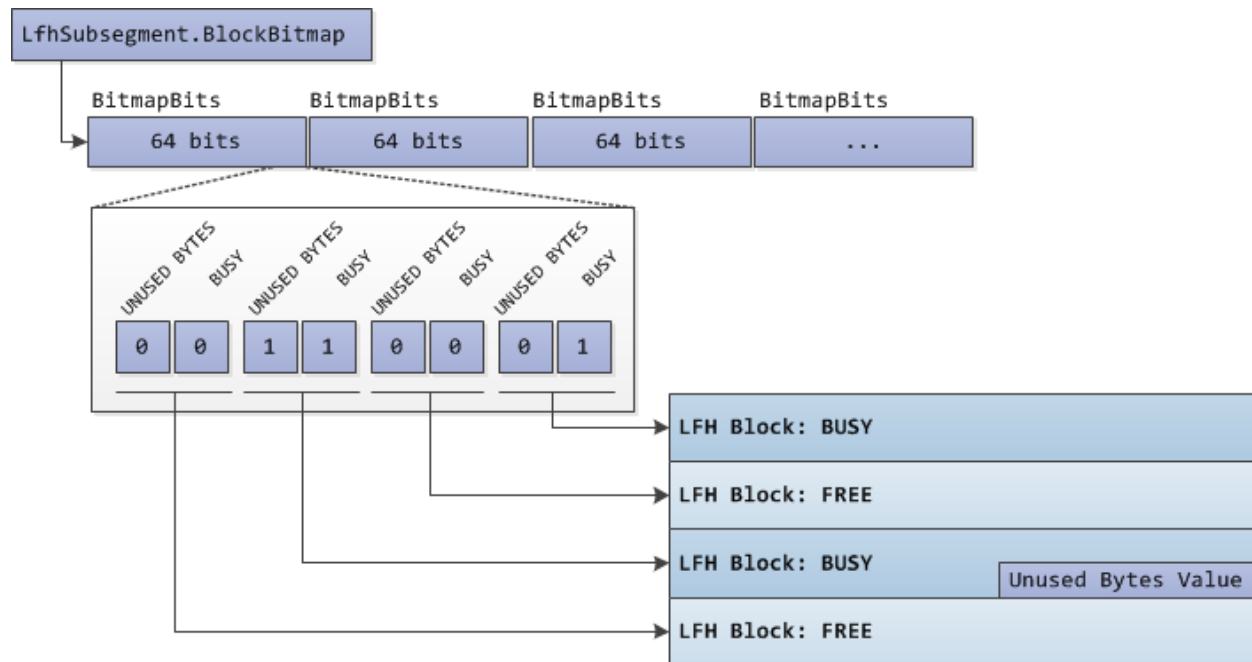
## LFH Block Bitmap

Each LFH block do not have a block header at the beginning, instead, and a block bitmap (`LfhSubsegment.BlockBitmap`) is used to track the state of each LFH block in the LFH subsegment.

Each LFH block is represented by two bits in the block bitmap. Bit 0 represents the BUSY bit and bit 1 represents the UNUSED BYTES bit. If the UNUSED BYTES bit is set, it means that there is a difference between the `UserSize` and the LFH block size, and the last two bytes of the LFH block is treated as a 16 bit low endian value to represent the difference. If the number of unused bytes is 1, the high bit of this 16 bit value is set and the rest of the bits are unused, otherwise, the high bit is clear and the low 14 bits are used to store the unused bytes value.

The block bitmap is also subdivided into QWORD-sized (64 bits) chunks, called `BitmapBits` in this paper, with each `BitmapBits` representing 32 LFH blocks.

Below is an illustration of the LFH block bitmap:



## LFH Bucket Activation

In every allocation request in which the allocation size is  $\leq 16,368$  ( $0x3FF0$ ) bytes, `RtlpHpLfhContextAllocate()` is first called to check if the bucket corresponding to the allocation size is activated. If the bucket is activated, the allocation is serviced by LFH.

If the bucket is not activated, the bucket usage counter is updated. If after the update, the bucket usage counter reaches a particular value, the bucket is activated via `RtlpHpLfhBucketActivate()` and LFH will service the allocation request. Otherwise, the VS allocation component will eventually handle the allocation request.

Bucket activation occurs if there are 17 active allocations for the bucket's allocation size. The 17<sup>th</sup> active allocation will activate the bucket, and the 17<sup>th</sup> allocation and the next allocations afterwards will be serviced by the LFH.

Bucket activation also occurs if there are 2,040 allocation requests for the bucket's allocation size, regardless if blocks from previous allocations were already freed. The 2,040<sup>th</sup> allocation will activate the bucket, and the 2,040<sup>th</sup> allocation and the next allocations afterwards will be serviced by the LFH.

## LFH Allocation

LFH allocation is performed via `RtlpHpLfhContextAllocate()` which has the following function signature:

```
PVOID RtlpHpLfhContextAllocate(_HEAP_LFH_CONTEXT* LfhContext, SIZE_T UserSize, SIZE_T AllocSize,
                                ULONG Flags)
```

The first action performed by `RtlpHpLfhContextAllocate()` is to check if the bucket corresponding to the allocation size is activated. If the bucket is not activated, the usage counter of the bucket is updated, and if the resulting bucket usage after the update is subject for activation, the bucket is activated and LFH allocation will continue.

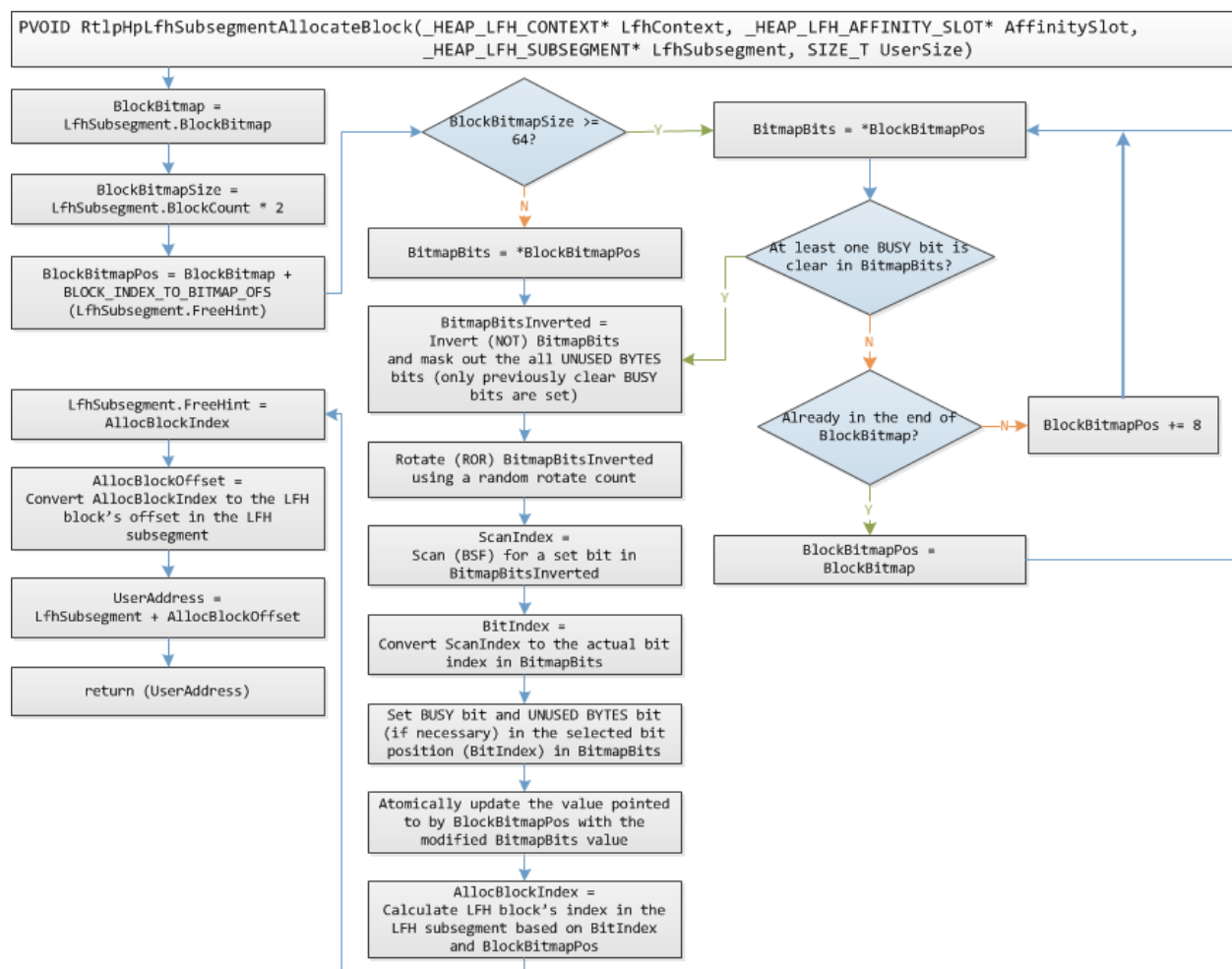
Next, the appropriate affinity slot in the bucket is selected depending on the requesting thread's processor and the processor-to-affinity slot mapping (`LfhContext.ProcAffinityMapping`). Once the affinity slot is selected,

allocation in the affinity slot's available LFH subsegment will be performed via a call to `RtlpHpLfhSlotAllocate()`.

`RtlpHpLfhSlotAllocate()`, on the other hand, first makes sure that the slot has an available LFH subsegment by creating a new LFH subsegment or re-using a cached LFH subsegment if needed. `RtlpHpLfhSlotAllocate()` will then call `RtlpHpLfhSlotReserveBlock()` to attempt to reserve a block from one of the slot's available LFH subsegments by atomically decrementing an LFH subsegment's `FreeCount` field. Too many contention detected from `RtlpHpLfhSlotReserveBlock()` will eventually cause a new affinity slot to be created for the requesting thread's processor.

If `RtlpHpLfhSlotReserveBlock()` is able to reserve a block in one of the affinity slot's LFH subsegments, `RtlpHpLfhSlotAllocate()` will call `RtlpHpLfhSubsegmentAllocateBlock()` to perform the actual allocation from the LFH subsegment where a block was reserved.

The logic of `RtlpHpLfhSubsegmentAllocateBlock()` for finding a free LFH block in the LFH subsegment is shown in the diagram below:



The bulk of the logic is from `RtlpLfhBlockBitmapAllocate()` (inlined in the diagram for brevity) which scans the block bitmap for a clear BUSY bit. The starting position of the search in the block bitmap is biased by `LfhSubsegment.FreeHint` and the selection of a clear BUSY bit is randomized.

The logic starts by pointing `BlockBitmapPos` to a `BitmapBits` in the block bitmap where `FreeHint` is (block index of recently allocated or freed LFH block). It then moves `BlockBitmapPos` forward until it finds a `BitmapBits` in which at least 1 BUSY bit is clear. If `BlockBitmapPos` reaches the end of the block bitmap, `BlockBitmapPos` is pointed to the start of the block bitmap and the search continues.

Once a `BitmapBits` is selected, the logic will randomly select a bit position in `BitmapBits` in which the BUSY bit is clear. After the bit position (`BitIndex`) is selected, the BUSY bit (and UNUSED BYTES bit if necessary) in the bit position is set, then, the value pointed to by `BlockBitmapPos` is atomically updated with the modified `BitmapBits` value. Finally, the bit position along with the value of `BlockBitmapPos` is translated into the address of the allocated LFH block (`UserAddress`). Note that the retry logic when the atomic update failed is not included in the diagram for brevity.

Below is an illustration where 8 LFH blocks are sequentially allocated from a new LFH subsegment, notice the random position of each LFH allocation:

FREE	FREE	FREE	FREE	BUSY Alloc #3	FREE	FREE	FREE
BUSY Alloc #4	FREE	FREE	BUSY Alloc #7	BUSY Alloc #5	FREE	FREE	BUSY Alloc #6
FREE	FREE	FREE	BUSY Alloc #1	FREE	FREE	FREE	FREE
BUSY Alloc #8	FREE	FREE	FREE	FREE	BUSY Alloc #2	FREE	FREE

## LFH Freeing

LFH freeing is performed via `RtlpHpLfhSubsegmentFreeBlock()` which has the following function signature:

```
BOOLEAN RtlpHpLfhSubsegmentFreeBlock(_HEAP_LFH_CONTEXT* LfhContext,
                                     _HEAP_LFH_SUBSEGMENT* LfhSubsegment,
                                     PVOID UserAddress, ULONG Flags)
```

The freeing code first computes the LFH block index of the `UserAddress` (`LfhBlockIndex`). If `LfhBlockIndex` index is less than or equal to `LfhSubsegment.FreeHint`, `LfhSubsegment.FreeHint` will be set to `LfhBlockIndex`.

Next, the corresponding BUSY and UNUSED BYTES bits of the LFH block in the block bitmap is atomically cleared, then, the LFH subsegment's `FreeCount` field is atomically incremented making the LFH block available for allocation.

## 2.6. LARGE BLOCKS ALLOCATION

Large blocks allocation is used for allocations with sizes 520,193 bytes and above ( $\geq 0 \times 7F001$ ). Large blocks have a page size granularity and each does not have a block header at the beginning. Unlike VS and LFH allocations which use the backend to create the subsegments where the VS/LFH blocks will be allocated, large blocks are allocated using the virtual memory functions provided by the NT Memory Manager.



## \_HEAP\_LARGE\_ALLOC\_DATA Structure

Each large block has a corresponding metadata with the following structure :

```
windbg> dt nt!_HEAP_LARGE_ALLOC_DATA
+0x000 TreeNode       : _RTL_BALANCED_NODE
+0x018 VirtualAddress  : Uint8B
+0x018 UnusedBytes     : Pos 0, 16 Bits
+0x020 ExtraPresent    : Pos 0, 1 Bit
+0x020 Spare           : Pos 1, 11 Bits
+0x020 AllocatedPages  : Pos 12, 52 Bits
```

- **TreeNode** - Each large block metadata is a node in the large blocks metadata tree (HeapBase.LargeAllocMetadata).
- **VirtualAddress** - Address of the block. First 16 bits are used for the UnusedBytes field.
- **UnusedBytes** - Difference between the UserSize and the committed size of the block.
- **AllocatedPages** – Committed size of the block in pages.

Interestingly, this metadata is stored in a separate heap which address is stored in the global variable RtlpHpMetadataHeap.

## Large Block Allocation

Large block allocation is performed via RtlpHpLargeAlloc() which has the following function signature:

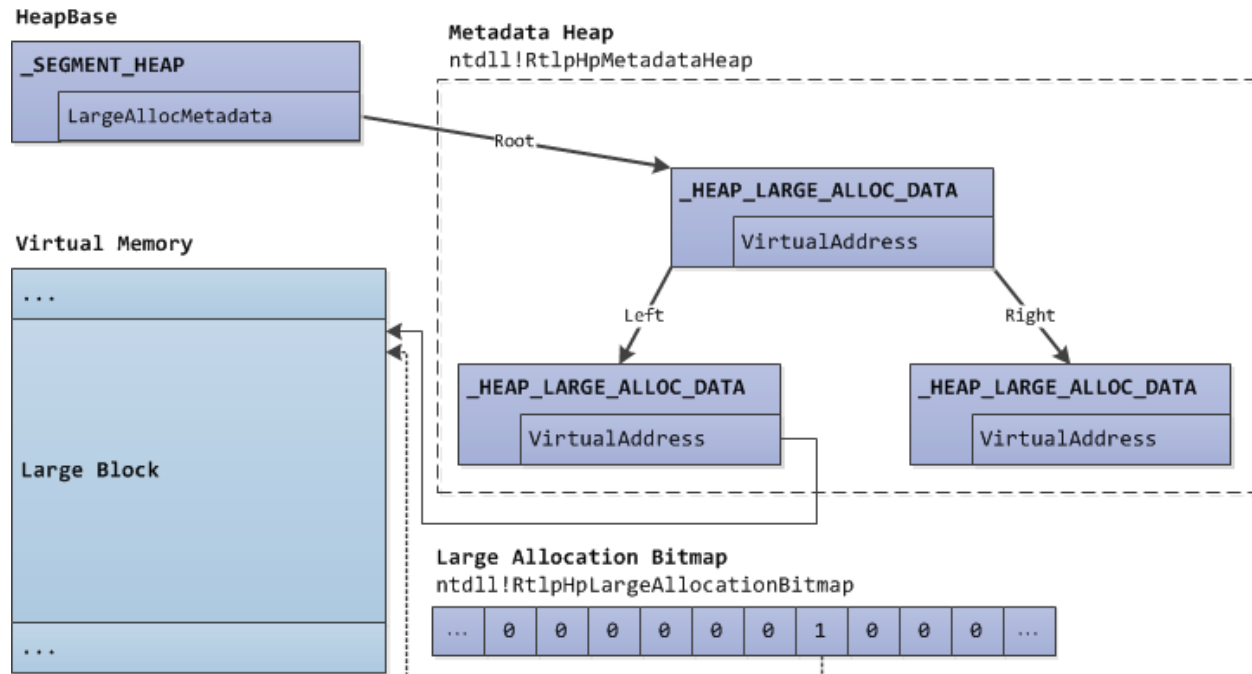
```
PVOID RtlpHpLargeAlloc(_SEGMENT_HEAP* HeapBase, SIZE_T UserSize, SIZE_T AllocSize, ULONG Flags)
```

Large block allocation is straightforward since there's no free list to consult. First, an allocation of the block's metadata from the metadata heap is done. Next, via NtAllocateVirtualMemory(), a virtual memory with a size equal to the allocation size plus 0x1000 bytes for the guard page is reserved. Then, a size equal to the allocation size is committed from the initially reserved memory, leaving the last guard page still in the reserved state.

After allocating the block, the block's metadata fields are set and the large allocation bitmap (RtlpHpLargeAllocationBitmap) is updated to mark the large block's address (actually UserAddress >> 16) as a large block allocation.

Finally, the block's metadata is inserted into the large blocks metadata tree (HeapBase.LargeAllocMetadata) using the block's address as key, then, the block's address (UserAddress) is returned to the caller.

Below is the illustration of different structures and global variables that support large blocks allocation:



## Large Block Freeing

Large block freeing is performed via `RtlpHpLargeFree()` which has the following function signature:

```
BOOLEAN RtlpHpLargeFree(_SEGMENT_HEAP* HeapBase, PVOID UserAddress, ULONG Flags)
```

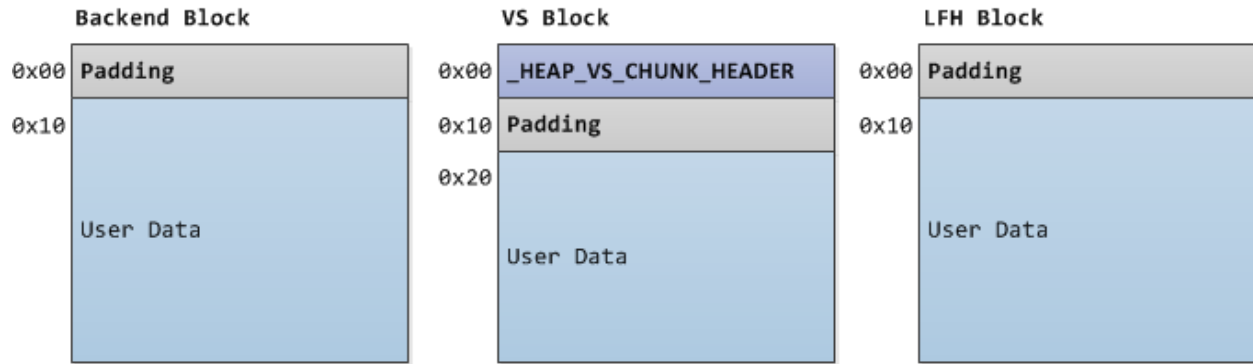
Similar to large block allocation, freeing a large block is a straightforward process. First, the metadata of the large block is first retrieved via `RtlpHpLargeAllocGetMetadata()` and then removed from the large blocks meta data tree afterwards.

Next, the large allocation bitmap is updated to unmark block's address as a large block allocation. Then, finally the virtual memory of the block is freed and the block's metadata is freed.

## 2.7. BLOCK PADDING

In applications that are not opted-in by default to use the Segment Heap (i.e.: not a Windows app and not a system executable as discussed in 2.1), an additional 16 (0x10) bytes padding is added to the block. The padding increases the total block size required for the allocation and changes the layout of backend blocks, VS blocks and LFH blocks.

Below are the layout of backend, VS and LFH blocks when padding is added:



The padding should be taken into consideration when analyzing allocated blocks, especially if the application under observation is neither a Windows app nor a system process.

## 2.8. SUMMARY AND ANALYSIS: INTERNALS

The implementation of the Segment Heap and the NT Heap are very different. The major differences can be observed in the data structures used, the use of free trees instead of linked list to track free blocks, and the use of a best-fit search algorithm with preference to most committed blocks when searching for free blocks.

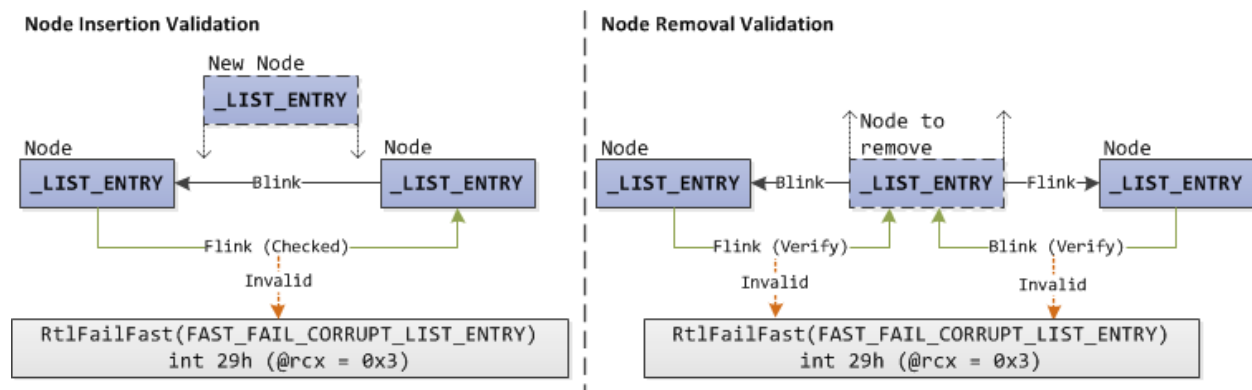
Also, although LFH in the Segment Heap and the NT Heap have the same purpose of reducing fragmentation and have the same general design, the implementation of the LFH in the Segment Heap had been overhauled. The major differences can be observed in the data structures used, the block bitmap representing the LFH blocks, and the absence of a block header at the beginning of each LFH block.

### 3. SECURITY MECHANISMS

This section discusses the different mechanisms added in the Segment Heap to make it difficult or unreliable to attack heap metadata, and in certain cases, make it unreliable to perform precise heap layout manipulation.

#### 3.1. FAST FAIL ON LINKED LIST NODE CORRUPTION

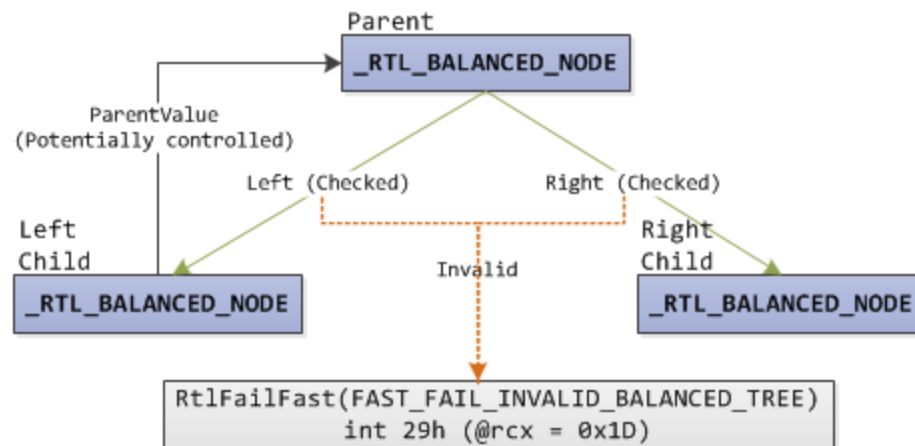
The Segment Heap uses linked lists for tracking list of segments and subsegments. Similar to older heap implementations, checks were added in the linked list node insertion and removal operations to prevent classic arbitrary memory writes due to corrupted linked list nodes. If a corrupted node is detected, the application immediately terminates via the FastFail [7] mechanism:



#### 3.2. FAST FAIL ON RB TREE NODE CORRUPTION

The Segment Heap uses RB trees for tracking free backend and VS blocks. It is also used for tracking large blocks metadata. The NTDLL-exported functions `RtlRbInsertNodeEx()` and `RtlRbRemoveNode()` performs the node insertion and removal respectively in addition to making sure that the RB tree is balanced. To prevent arbitrary writes due to corrupted tree nodes, the aforementioned functions perform validation when manipulating RB tree nodes. Similar to linked list nodes validation, failure in the validation of RB tree nodes will cause invocation of the FastFail mechanism.

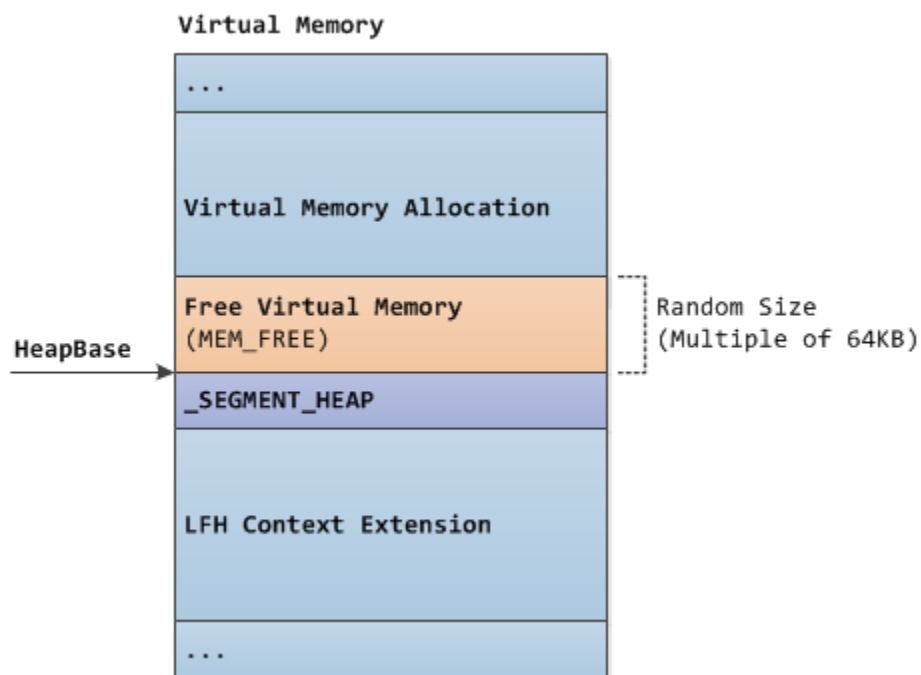
In the example validation below, the parent of the left child will be manipulated, which in turn, may lead to an arbitrary write if the left child's `ParentValue` pointer is controlled by an attacker. To prevent an arbitrary write, the parent's child nodes are checked if one of them is indeed the left child.

**Example: ParentValue Verification Before Parent Manipulation**

### 3.3. HEAP ADDRESS RANDOMIZATION

To make guessing of the heap address unreliable, randomness is added to where the heap will be located in virtual memory.

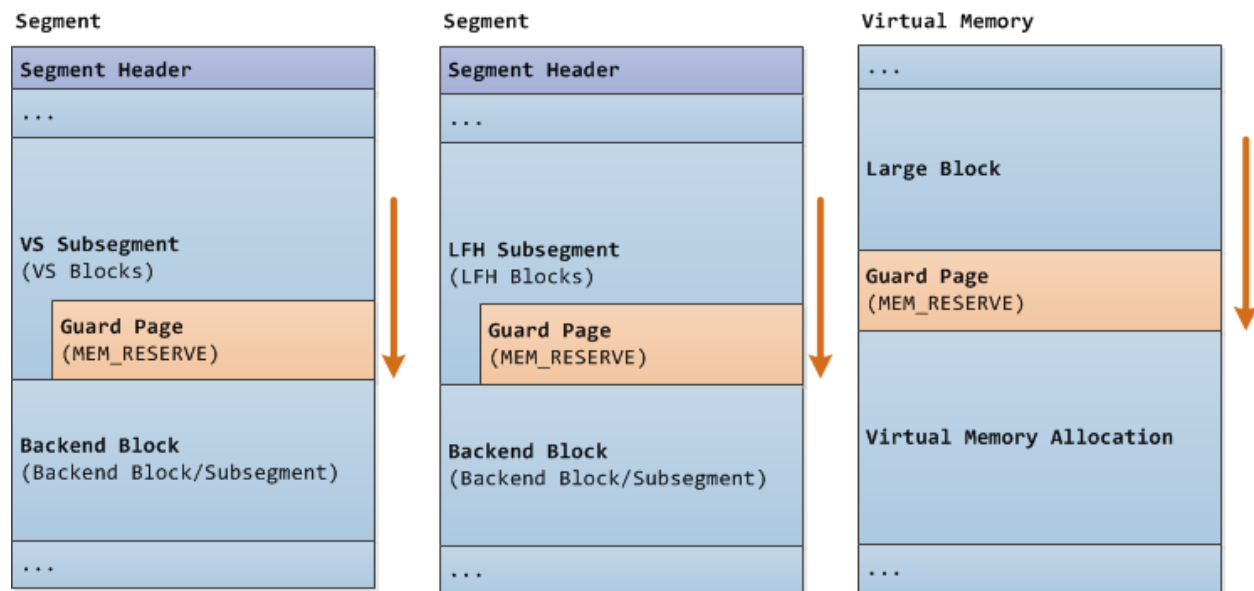
Heap address randomization is performed by `RtlpHpSegHeapAllocate()`, a function used for the creation of the heap. It is done by first reserving virtual memory with size equal to the computed size of the heap plus a randomly generated size (the random size is a multiple of 64KB). After reserving virtual memory, the beginning of the reserved virtual memory up to a size equal to the initially generated random size is released. Then, `HeapBase` is pointed to beginning of the unreleased portion of the initially reserved virtual memory.



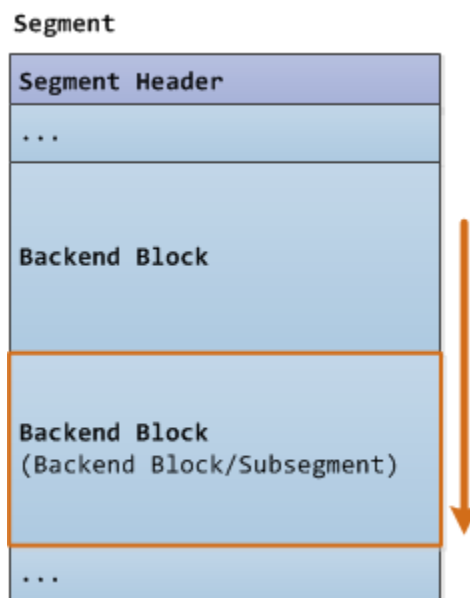
### 3.4. GUARD PAGES

When VS subsegments, LFH subsegments and large blocks are allocated, a guard page is added at the end of the subsegment/block. For VS and LFH subsegments, the subsegment size should be  $\geq 64\text{KB}$  for a guard page to be added.

The guard page prevents a sequential overflow from VS blocks, LFH blocks and large blocks from corrupting adjacent data outside the subsegment (for LFH/VS blocks) or outside the block (for large blocks).

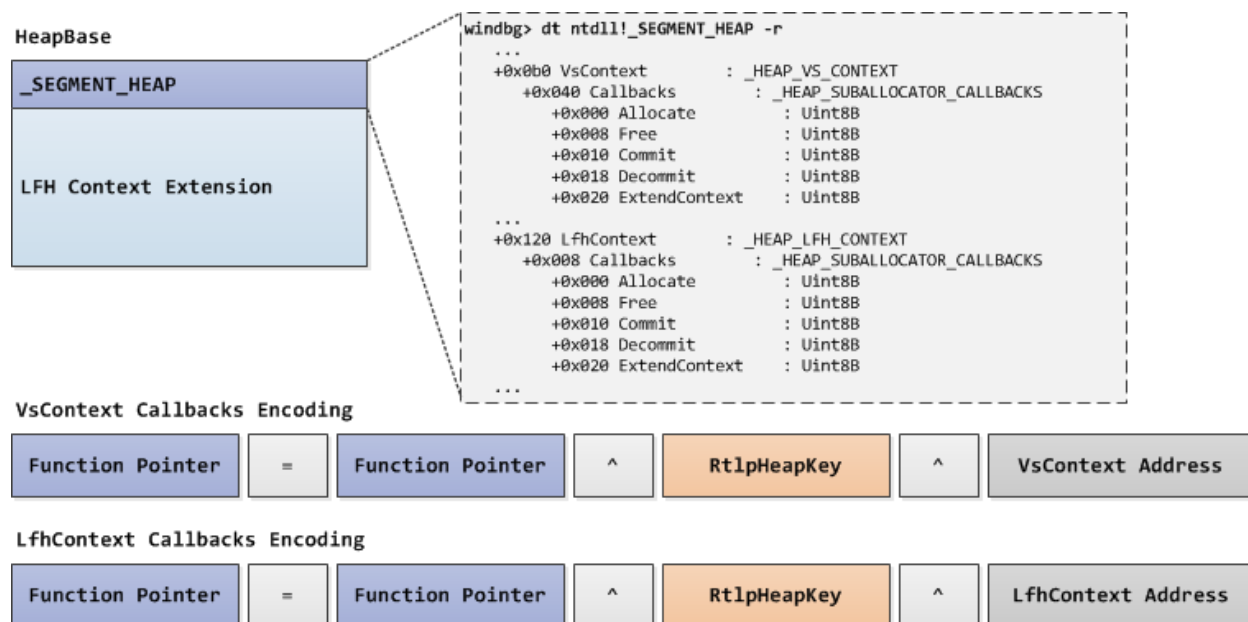


Backend blocks, on the other hand, do not have a guard page after them, allowing an overflow to corrupt adjacent data outside the block.



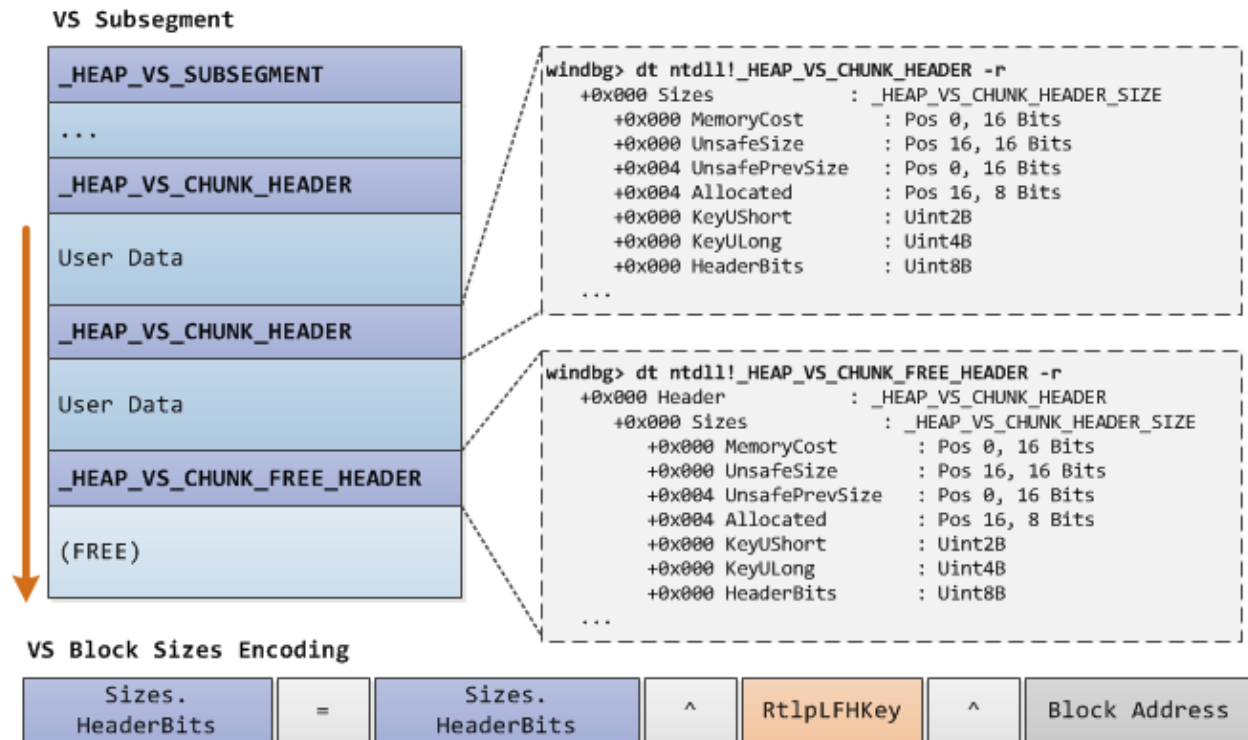
### 3.5. FUNCTION POINTER ENCODING

In cases where the attacker is able to determine the address of the heap and assuming that the attacker has a Control Flow Guard (CFG) bypass, the attacker can target the function pointers stored in the HeapBase as a way to directly control execution flow. To protect these function pointers from trivial modification, the functions pointers are encoded using the heap key and the LFH/VS context address.



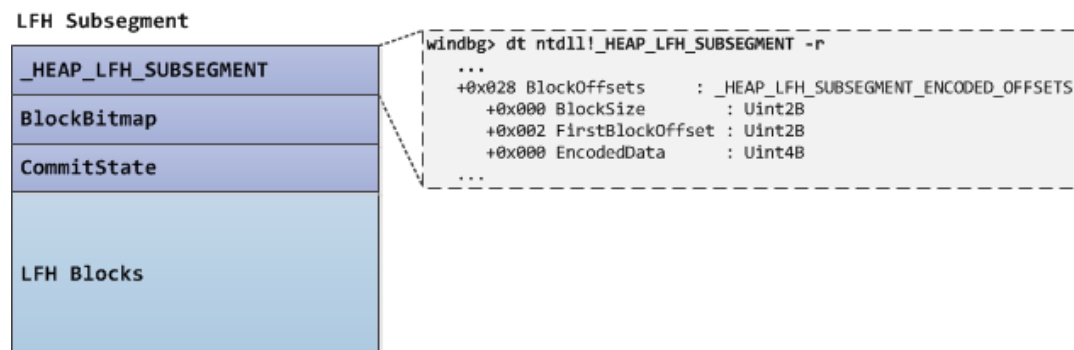
### 3.6. VS BLOCK SIZES ENCODING

Unlike backend/LFH/large blocks, VS blocks have a header at the beginning of each block which makes VS block headers a likely target in a buffer overflow. To protect important parts of the VS block header from trivial modification, they are encoded using the LFH key and the VS block address.

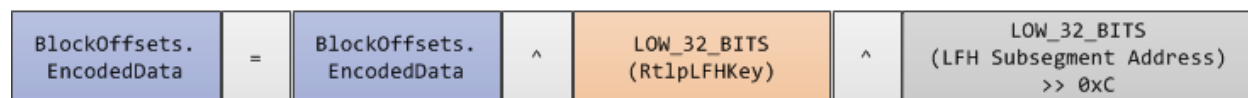


### 3.7. LFH SUBSEGMENT BLOCKOFFSETS ENCODING

To protect important LFH subsegment header fields from trivial modification, the block size field and first block offset field in the LFH subsegment header are encoded using the LFH key and the LFH subsegment address.



#### LFH Subsegment BlockOffsets Encoding




### 3.8. LFH ALLOCATION RANDOMIZATION

To make exploitation of LFH-based buffer overflows and use-after-frees unreliable, the LFH component randomly selects which free LFH block to use in allocation request. The allocation randomization makes it unreliable to place a target LFH block adjacent to an LFH block that can be overflowed, and it also makes it unreliable to reuse a



recently freed LFH block. The allocation randomization algorithm is discussed in the “LFH Allocation” subsection in 2.5.

Below is an illustration where 8 LFH blocks are sequentially allocated from a new LFH subsegment:



FREE	FREE	FREE	FREE	BUSY Alloc #3	FREE	FREE	FREE
BUSY Alloc #4	FREE	FREE	BUSY Alloc #7	BUSY Alloc #5	FREE	FREE	BUSY Alloc #6
FREE	FREE	FREE	BUSY Alloc #1	FREE	FREE	FREE	FREE
BUSY Alloc #8	FREE	FREE	FREE	FREE	BUSY Alloc #2	FREE	FREE

Notice the first allocation is on the 20<sup>th</sup> LFH block, the second allocation is on the 30<sup>th</sup> block, the third allocation is on the 5<sup>th</sup> block, and so on.

### 3.9. SUMMARY AND ANALYSIS: SECURITY MECHANISMS

The applied security mechanisms in the Segment Heap are mostly a carryover of the security mechanisms from the NT Heap, notable of which are the guard pages and the LFH allocation randomization which was new when Windows 8 was released [5, 8]. Based on this, and how important fields of the new data structures are protected, the Segment Heap is comparable with the NT heap in terms of applied security mechanisms. However, it is yet to be seen how the new Segment Heap data structures will fare when metadata attack research of the Segment Heap becomes popular.

With regards to heap layout manipulation, the best-fit search algorithm and the free block splitting mechanism of the backend and the VS component are more welcoming to heap layout manipulation compared to the LFH component which uses allocation randomization.

## 4. CASE STUDY

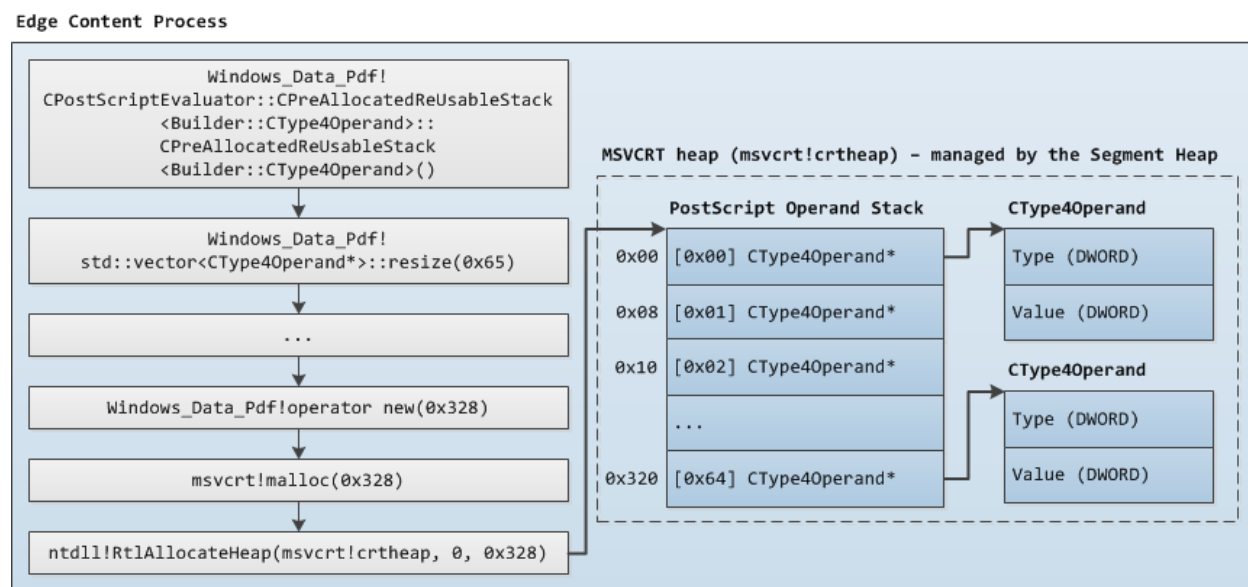
This section examines how the layout of a heap managed by the Segment Heap can be manipulated by discussing a way to leverage a memory corruption vulnerability for a reliable arbitrary write in the context of the Edge content process.

### 4.1. CVE-2016-0117 VULNERABILITY DETAILS

The vulnerability (CVE-2016-0117 [9], MS16-028 [10]) is in WinRT PDF's [11] PostScript interpreter for Type 4 (PostScript Calculator) functions [12]. PostScript Calculator functions use a subset of the PostScript language operators and these PostScript operators use the PostScript operand stack when performing their functions.

The PostScript operand stack is a vector containing 0x65 CType40operand pointers. Each CType40operand, on the other hand, is a data structure consisting of one DWORD that represents the type and one DWORD representing the value in the PostScript operand stack.

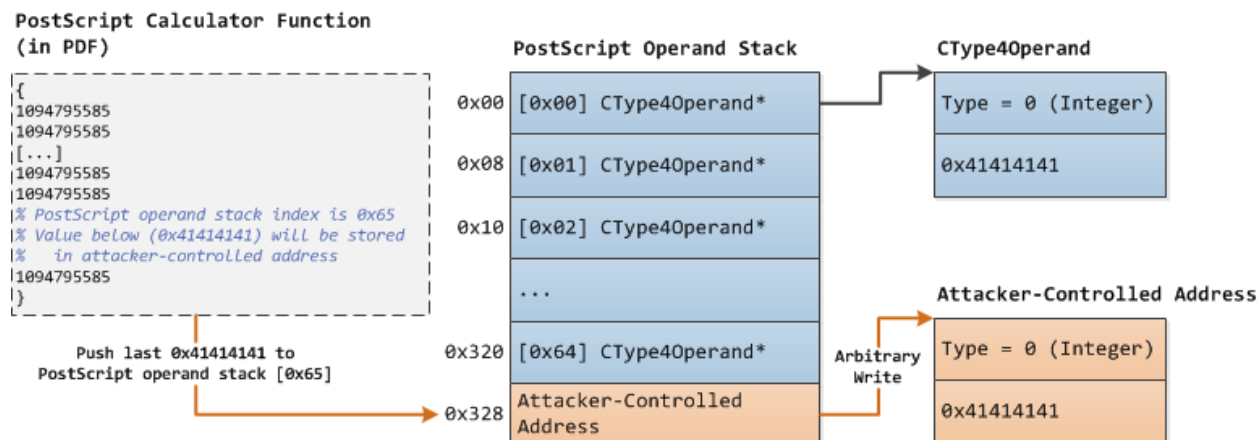
The PostScript operand stack and the CType40operands are allocated from the MSVCRT heap which is managed by the Segment Heap if WinRT PDF is loaded in the context of the Edge content process:



The issue is that the PostScript interpreter fails to validate if the PostScript operand stack index is past the end of the PostScript operand stack (PostScript operand stack index is 0x65), allowing a dereference of a CType40operand pointer located right after the end of the PostScript operand stack.

If an attacker is able to implant a target address right after the end of the PostScript operand stack, the attacker will be able to perform a memory write to the target address via a PostScript operation that pushes a value in the PostScript operand stack.

In the illustration below, multiple integers (1094795585 or 0x41414141) are pushed to the PostScript operand stack with the last 0x41414141 pushed to invalid index 0x65 of the PostScript operand stack:

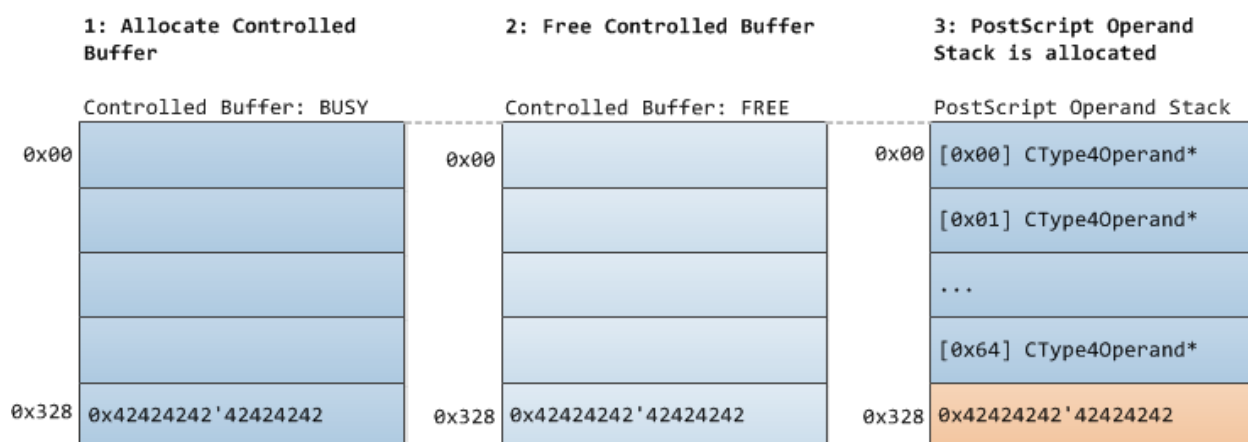


## 4.2. PLAN FOR IMPLANTING THE TARGET ADDRESS

After understanding the vulnerability, the following plan is used to implant the target address after the end of the PostScript operand stack:

1. Allocate a controlled buffer and set offset 0x328 of the controlled buffer to the target address (0x4242424242424242). For reliability, the controlled buffer and the PostScript operand stack will be VS-allocated instead of being LFH-allocated.
2. Free the controlled buffer.
3. The PostScript operand stack will be allocated in the free VS block of the freed controlled buffer.

Below is the illustration of the plan:



Executing the above plan requires the ability to manipulate the MSVCRT heap in order to reliably implant the target address after the PostScript operand stack, this includes the ability to allocate a controlled block from the MSVCRT heap and the ability to free the controlled block. In addition, there will be some issues that will affect reliability (such as free blocks coalescing) which needs to be dealt with. The next subsections will discuss the solutions to these requirements/issues.

### 4.3. MANIPULATING THE MSVCRT HEAP WITH CHAKRA'S ARRAYBUFFER

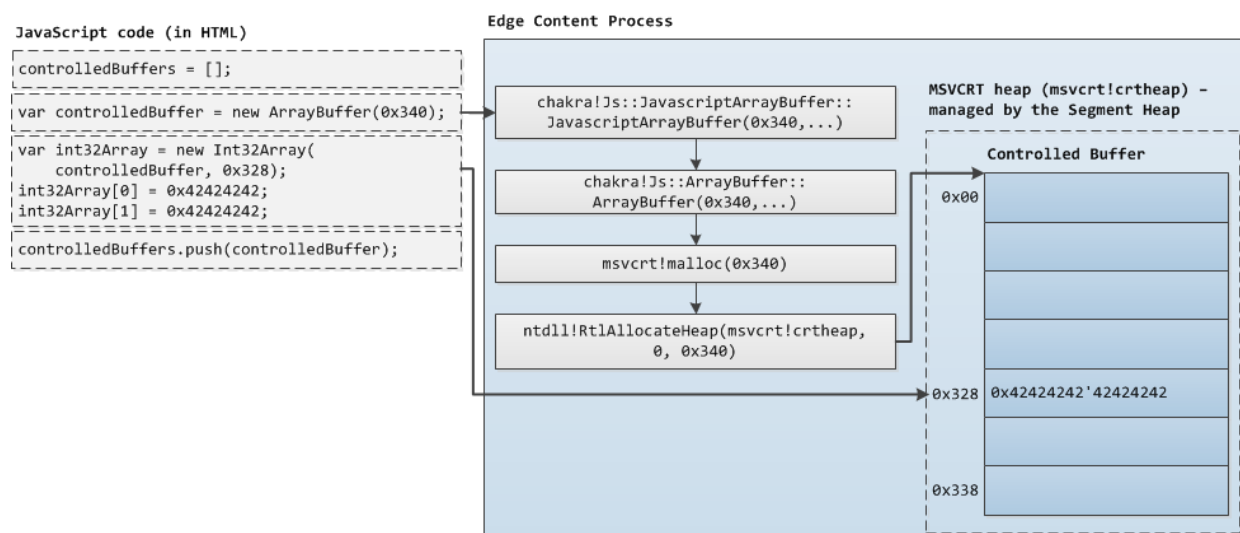
JavaScript embedded in the PDF can potentially fulfill the requirement of MSVCRT heap manipulation, but unfortunately, as of writing, WinRT PDF still does not support embedded JavaScript.

Fortunately, a solution can be found in the Chakra's (Edge's JavaScript engine) ArrayBuffer implementation. Similar to WinRT PDF's PostScript operand stack, the data buffer of Chakra's ArrayBuffer is also allocated from the MSVCRT heap via `msvcrt!malloc()` [13, 14] if the ArrayBuffer is of a certain size (i.e.: size is less than 64KB, or for size  $\geq 64$ KB, additional checks are performed).

This means that a JavaScript code in an HTML file can allocate and free the controlled buffer in the MSVCRT heap (step 1 and step 2 of the plan). Then, the JavaScript code can inject an `<embed>` element in the page which causes the PDF file containing the vulnerability trigger to be loaded by WinRT PDF. Upon loading the PDF, WinRT PDF will allocate the PostScript operand stack from the MSVCRT heap, the free VS block of the freed controlled buffer would then be returned by the heap manager to WinRT PDF to fulfill the allocation request (step 3 of the plan).

#### Allocation and Setting Controlled Values

In the illustration below, a JavaScript code in an HTML file instantiated an ArrayBuffer with a size of `0x340` which in turn lead to an allocation of a `0x340` bytes block in the MSVCRT heap; offset `0x328` of the block is then set with the target address:



#### LFH Bucket Activation

Activating the LFH bucket for a particular allocation size is also an important capability and its use in the plan will be later discussed. To activate the LFH bucket for a specific allocation size, 17 ArrayBuffer objects with the same size needs to be instantiated:

```

lfhBucketActivators = [];
for (var i = 0; i < 17; i++) {
    lfhBucketActivators.push(new ArrayBuffer(blockSize));
}

```

## Freeing and Garbage Collection

Freeing blocks involves removing references to the `ArrayBuffer` object and then triggering a garbage collection. Note that Chakra's `CollectGarbage()` is still callable but its functionality is disabled in Edge [15], therefore, another mechanism to trigger garbage collection is needed.

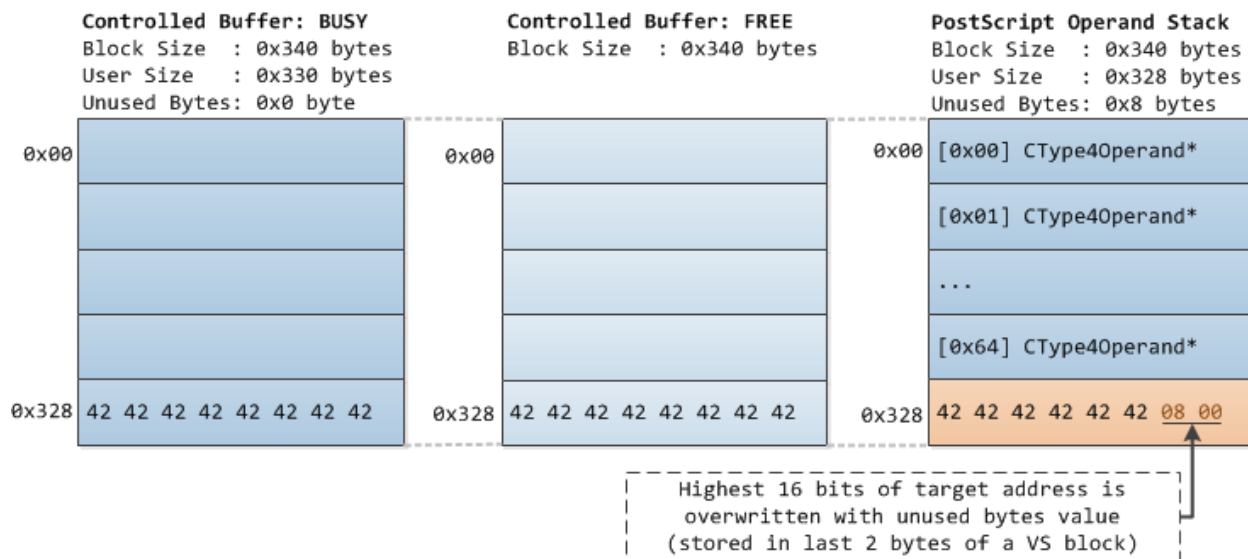
Looking again at the `ArrayBuffer` functionality, every time an `ArrayBuffer` is created, the size passed to the `ArrayBuffer` constructor is added to an internal Chakra heap manager counter [16], if that particular counter reaches  $\geq 192\text{MB}$  in machines with  $>1\text{GB}$  memory (threshold is lower for machines with lower memory), a concurrent garbage collection is triggered.

Therefore, to perform garbage collection, an `ArrayBuffer` with a size of 192MB is created, then a delay is introduced to let the concurrent garbage collection to finish, and afterwards, the succeeding JavaScript code is executed:

```
// trigger concurrent garbage collection
gcTrigger = new ArrayBuffer(192 * 1024 * 1024);
// then call afterGcCallback after some delay (adjust if needed)
setTimeout(afterGcCallback, 1000);
```

### 4.4. PREVENTING TARGET ADDRESS CORRUPTION

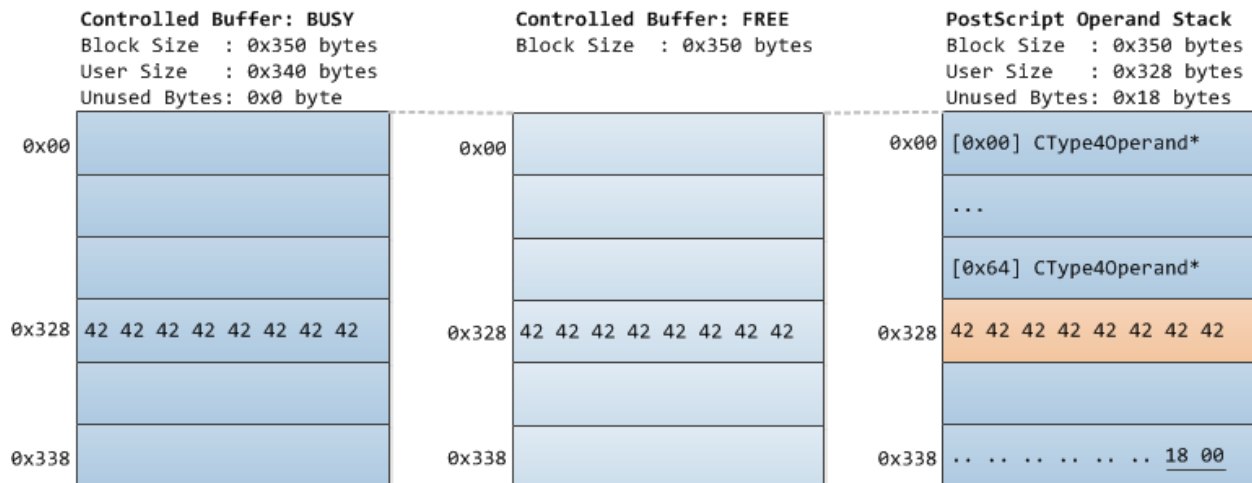
Since VS allocations are performed using a best-fit policy, the first idea that comes to mind is to VS-allocate the controlled buffer using `0x330` as the size. However, this first idea have a problem in that the highest 16 bits of the target address will be overwritten with unused bytes metadata which is stored in the last two bytes of a VS block:



To solve the issue, a property of VS chunk splitting can be leveraged. Specifically, as previously mentioned in the “VS Allocation” subsection in 2.4, large free blocks are split unless the block size of the resulting remaining block will be less than `0x20` bytes.

Therefore, if a `0x340` bytes controlled buffer (total block size including header: `0x350`) is used and that a `0x328` bytes PostScript operand stack (total block size including header: `0x340`) will be allocated in the freed controlled buffer's free VS block, the size of the remaining block after the split of will only be `0x10` bytes, thereby preventing

the split of the 0x350 bytes free VS block. And if that is the case, the unused bytes metadata will be stored at offset 0x33E of the VS block, leaving the target address unmodified:



#### 4.5. PREVENTING FREED CONTROLLED BUFFER BLOCKS FROM BEING COALESCED

To prevent the free VS block of the freed controlled buffer from being coalesced with neighboring free VS blocks, 15 instead of one controlled buffers are created sequentially, then, in an alternating manner, eight are kept busy and seven are freed.

The illustration below shows a favorable allocation pattern that prevents the free VS block of the freed controlled buffers from being coalesced:

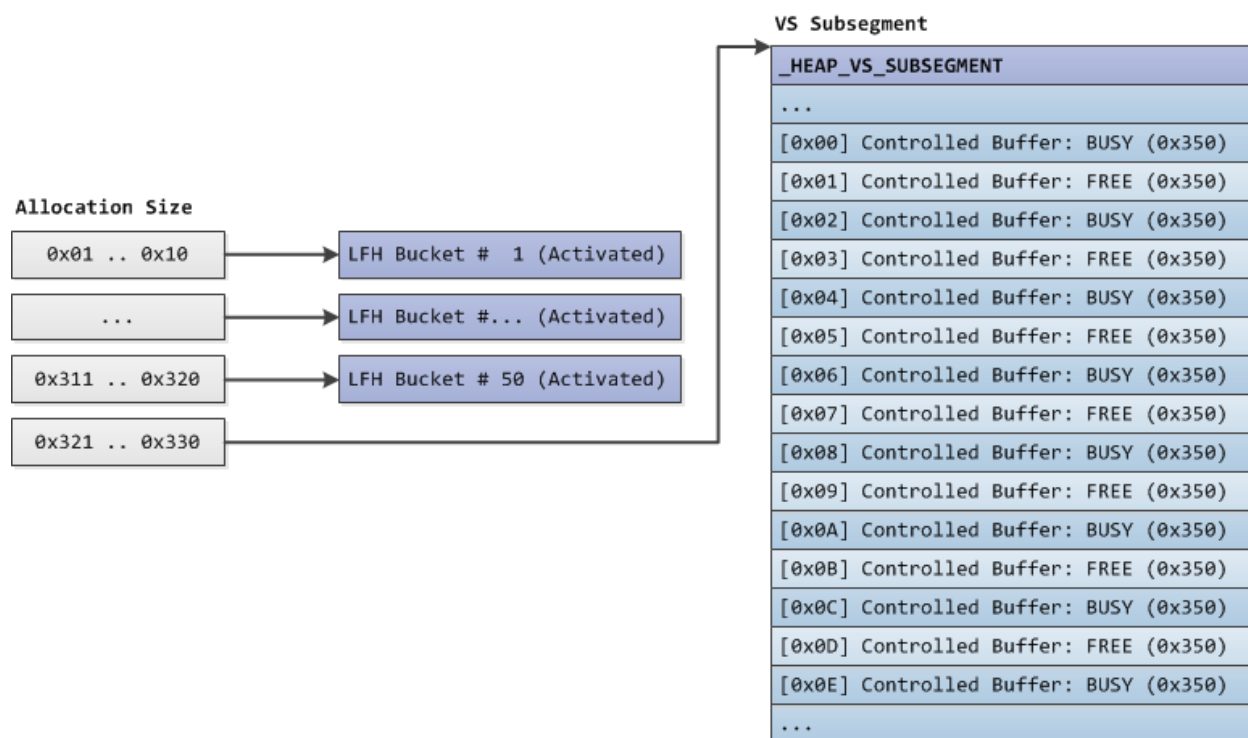
VS Subsegment

_HEAP_VS_SUBSEGMENT
...
[0x00] Controlled Buffer: BUSY (0x350)
[0x01] Controlled Buffer: FREE (0x350)
[0x02] Controlled Buffer: BUSY (0x350)
[0x03] Controlled Buffer: FREE (0x350)
[0x04] Controlled Buffer: BUSY (0x350)
[0x05] Controlled Buffer: FREE (0x350)
[0x06] Controlled Buffer: BUSY (0x350)
[0x07] Controlled Buffer: FREE (0x350)
[0x08] Controlled Buffer: BUSY (0x350)
[0x09] Controlled Buffer: FREE (0x350)
[0x0A] Controlled Buffer: BUSY (0x350)
[0x0B] Controlled Buffer: FREE (0x350)
[0x0C] Controlled Buffer: BUSY (0x350)
[0x0D] Controlled Buffer: FREE (0x350)
[0x0E] Controlled Buffer: BUSY (0x350)
...

Actual allocation patterns will not always exactly match the above illustration, such as when some of the controlled buffers are allocated from a different VS subsegment. However, multiple freed and busy controlled buffers increase the chance that at least one or more free VS blocks of the freed controlled buffers will not be coalesced.

#### 4.6. PREVENTING UNINTENDED USE OF FREED CONTROLLED BUFFERS BLOCKS

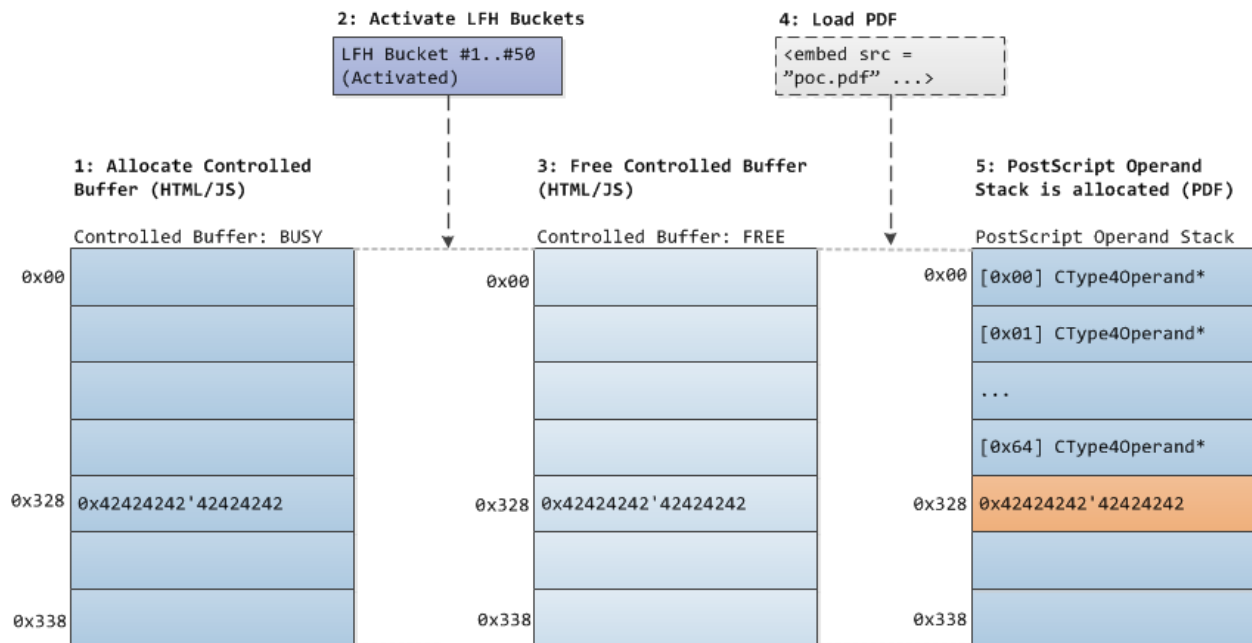
After controlled buffers are freed in step 2, their corresponding free VS blocks might be split and used for small allocations that might occur before step 3. In order to prevent the unintended use of these free VS blocks, the corresponding LFH buckets for allocation sizes 0x1 to 0x320 are activated so that allocation for those sizes will be serviced by the LFH instead of being VS allocation component:



#### 4.7. ADJUSTED PLAN FOR IMPLANTING THE TARGET ADDRESS

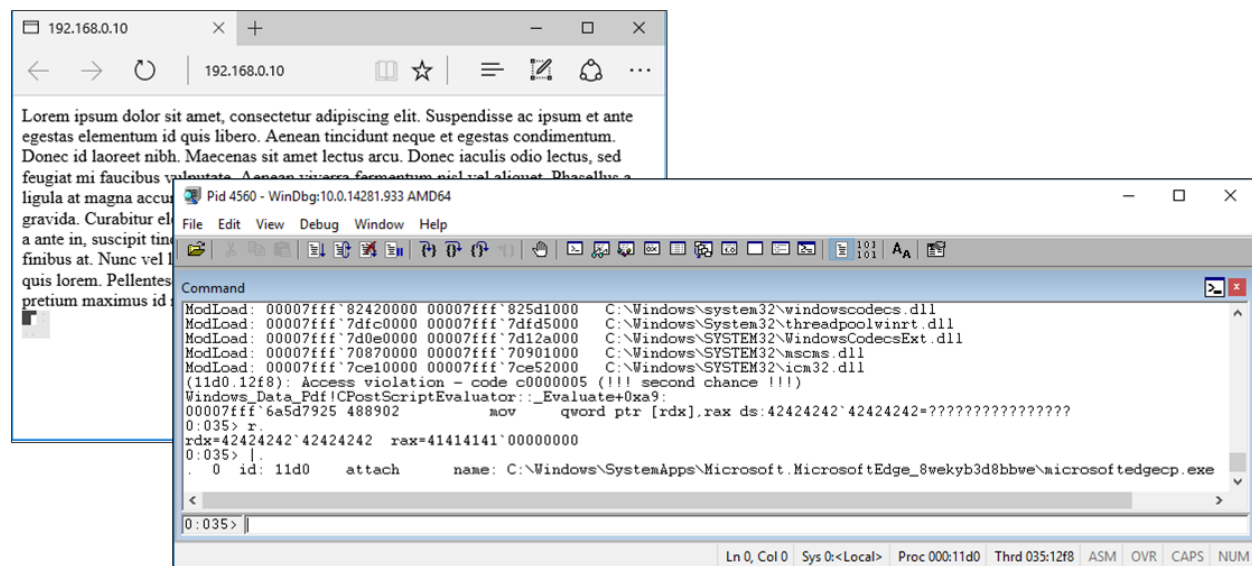
Now that the solutions to the issues are identified, the initial plan for implanting the target address is adjusted to the following:

1. HTML/JavaScript: Create 15 controlled buffers by instantiating ArrayBuffer objects with 0x340 as the size.
2. HTML/JavaScript: Activate the LFH buckets corresponding to allocation sizes 0x1 to 0x320.
3. HTML/JavaScript: In alternating manner, free seven controlled buffers and leave eight controlled buffers busy.
4. HTML/JavaScript: Inject an <embed> element to the page in order for WinRT PDF to load the PDF that triggers the vulnerability.
5. PDF: WinRT PDF will allocate the PostScript operand stack and the block by returned by the heap manager will be the free VS block of one of the freed controlled buffers.



## 4.8. SUCCESSFUL ARBITRARY WRITE

Once the target address is successfully implanted after the end of the PostScript operand stack and the vulnerability is triggered, arbitrary write is achieved:



## 4.9. ANALYSIS AND SUMMARY: CASE STUDY

The case study showed that precise layout manipulation is achievable in heaps managed by the Segment Heap. Specifically, it showed how the layout of VS allocations can be controlled and how the LFH can be used to preserve the controlled layout of VS allocations by redirecting unwanted allocation requests to activated LFH buckets.

The two main elements that allowed the precise heap layout manipulation in the case study was the scripting capability provided by the Chakra JavaScript engine and a common heap used by both Chakra's ArrayBuffer and



WinRT PDF's PostScript interpreter. Without these two elements, precise layout manipulation of the MSVCRT heap using WinRT PDF's internal allocations and freeing of objects would likely be more difficult.

Finally, when developing proof-of-concepts, one might encounter issues that seem to be unresolvable, such as the target address corruption described in the case study. In cases such as those, understanding the internals of the heap implementation will sometimes provide the solution.

## 5. CONCLUSION

The internals of the Segment Heap and the NT Heap are largely different. Although some components of the Segment Heap and the NT Heap have the same purpose, the data structures supporting the Segment Heap are mostly unlike their counterpart in the NT Heap. Consequently, these new Segment Heap data structures are interesting for metadata attack research.

Nevertheless, the security mechanisms in the initial release of the Segment Heap in Windows 10 show that previous attacks and their corresponding mitigations in the NT Heap had been taken into consideration when the Segment Heap was developed.

In terms of heap layout manipulation, the case study showed that, given a capability to perform arbitrary allocations and frees, precise layout manipulation of heaps managed by the Segment Heap is achievable. The case study also showed that in-depth knowledge of the Segment Heap can help resolve seemingly unresolvable proof-of-concept reliability/functionality issues.

Finally, I hope that this paper helped you understand Windows 10's Segment Heap.

## 6. APPENDIX: WINDBG !HEAP EXTENSION COMMANDS FOR SEGMENT HEAP

Below are some useful WinDbg !heap extension commands that work with the Segment Heap.

### !heap -x <address>

This command is useful if the heap where a block was allocated from is unknown since this command only requires the block's address. This command will show the corresponding heap, segment, subsegment, subsegment's "first" page range descriptor, type and total size of the block.

Example output for a busy VS block (user-requested size is 0x328 bytes):

```
windbg> !heap -x 00000203`6b2b6200

[100 Percent Complete]
[33 Percent Complete]
Search was performed for the following Address: 0x000002036b2b6200
Below is a detailed information about this address.
Heap Address      : 0x000002036b140000
The address was found in backend heap.
Segment Address   : 0x000002036b200000
Page range index (0-255) : 120
Page descriptor address : 0x000002036b200f00
Subsegment address : 0x000002036b278000
Subsegment Size    : 266240 Bytes
Allocation status   : Variable size allocated chunk.
Chunk header address : 000002036b2b61f0
Chunk size (bytes)  : 848
Chunk unused bytes   : 24
```

### !heap -i <address> -h <heap>

Once the corresponding heap of the block is known, this command can be used to display additional details about the block such as its user-requested size and the decoded RangeFlags field of the subsegment's "first" page range descriptor.

Example output for a busy VS block (user-requested size is 0x328 bytes, same block as the previous example):

```
windbg> !heap -i 00000203`6b2b6200 -h 000002036b140000

The address 000002036b2b6200 is in Segment 000002036b200000
Page range descriptor address: 000002036b200f00
Page range start address:    000002036b278000
Range flags (2e): First Allocated Committed VS

UserAddress:      0x000002036b2b6200
Block is :        Busy
Total Block Size (Bytes): 0x350
User Size (bytes): 0x328
UnusedBytes (bytes): 0x18
```

### !heap -s -a -h <heap>

The !heap -s command with the -a option (dump all heap blocks option) can be used for examining the layout of a heap since it displays information about each block for each Segment Heap component in sequence.

Example output:

```
0:028> !heap -s -a -h 000002036b140000

Large Allocation X-RAY.
```

Block	Address	Metadata	Address	Virtual Address	Pages Allocated	Unused Size (Bytes)
	20c7fc020c0		20c7fc020c0	20300df0000	227	2383
	20c7fc02080		20c7fc02080	20300ee0000	137	1912

**Backend Heap X-RAY.**

Segment Address	Page Range Descriptor Address	Page Range Descriptor Index	Subsegment Address	Range Flags	Subsegment Type	Pages Allocated	Unused Bytes
2036b200000	2036b200040	2	2036b202000	2e	Variable Alloc	17	4096
2036b200000	2036b200260	19	2036b213000	2e	Variable Alloc	65	4096
2036b200000	2036b200a80	84	2036b254000	f	LFH Subsegment	1	0

[...]

**Variable size allocation X-RAY**

Segment Address	Subsegment Address	Chunk Header Address	Chunk Size	Bucket Index	Status	Unused Bytes
2036b200000	2036b202000	2036b202030	752	47	Allocated	8
2036b200000	2036b202000	2036b202320	1808	77	Allocated	0
2036b200000	2036b202000	2036b202a30	272	17	Allocated	0

[...]

**LFH X-RAY**

Segment Address	Subsegment Address	Block Address	Block Size	Bucket Index	Commit Status	Busy Bytes	Free Bytes	Unused Bytes
2036b200000	2036b254000	2036b254040	256	16	Committed	256	0	0
2036b200000	2036b254000	2036b254140	256	16	Committed	256	0	0
2036b200000	2036b254000	2036b254240	256	16	Committed	256	0	0

[...]

## 7. BIBLIOGRAPHY

- [1] J. McDonald and C. Valasek, "Practical Windows XP/2003 Heap Exploitation," [Online]. Available: <https://www.blackhat.com/presentations/bh-usa-09/MCDONALD/BHUSA09-McDonald-WindowsHeap-PAPER.pdf>.
- [2] B. Moore, "Heaps About Heaps," [Online]. Available: [https://www.insomniasec.com/downloads/publications/Heaps\\_About\\_Heaps.ppt](https://www.insomniasec.com/downloads/publications/Heaps_About_Heaps.ppt).
- [3] B. Hawkes, "Attacking the Vista Heap," [Online]. Available: [http://www.blackhat.com/presentations/bh-usa-08/Hawkes/BH\\_US\\_08\\_Hawkes\\_Attacking\\_Vista\\_Heap.pdf](http://www.blackhat.com/presentations/bh-usa-08/Hawkes/BH_US_08_Hawkes_Attacking_Vista_Heap.pdf).
- [4] C. Valasek, "Understanding the Low Fragmentation Heap," [Online]. Available: [http://illmatics.com/Understanding\\_the\\_LFH.pdf](http://illmatics.com/Understanding_the_LFH.pdf).
- [5] C. Valasek and T. Mandt, "Windows 8 Heap Internals," [Online]. Available: <http://illmatics.com/Windows%208%20Heap%20Internals.pdf>.
- [6] Wikipedia, "Red-black tree," [Online]. Available: [https://en.wikipedia.org/wiki/Red-black\\_tree](https://en.wikipedia.org/wiki/Red-black_tree).
- [7] A. Ionescu, "New Security Assertions in "Windows 8"," [Online]. Available: <http://www.alex-ionescu.com/?p=69>.
- [8] K. Johnson and M. Miller, "Exploit Mitigation Improvements in Windows 8," [Online]. Available: [http://media.blackhat.com/bh-us-12/Briefings/M\\_Miller/BH\\_US\\_12\\_Miller\\_Exploit\\_Mitigation\\_Slides.pdf](http://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf).
- [9] IBM, "X-FORCE ADVISORY: Microsoft Windows WinRT PDF Renderer Library PostScript Interpreter Remote Code Execution Vulnerability," [Online]. Available: <https://exchange.xforce.ibmcloud.com/collection/X-FORCE-ADVISORY-Microsoft-Windows-WinRT-PDF-Renderer-Library-PostScript-Interpreter-Remote-Code-Execution-Vulnerability-736a7f02705bd90867262493dca2a2b7>.
- [10] Microsoft, "Microsoft Security Bulletin MS16-028 - Critical," [Online]. Available: <https://technet.microsoft.com/en-us/library/security/ms16-028.aspx>.
- [11] M. V. Yason, "WinRT PDF: A Potential Route for Attacking Edge," [Online]. Available: <https://securityintelligence.com/winrt-pdf-a-potential-route-for-attacking-edge/>.
- [12] Adobe Systems Incorporated, "Document Management - Portable Document Format - Part 1: PDF 1.7, First Edition," [Online]. Available: [http://www.adobe.com/devnet/pdf/pdf\\_reference.html](http://www.adobe.com/devnet/pdf/pdf_reference.html).
- [13] Microsoft, "GitHub: Microsoft / ChakraCore / lib / Runtime / Library / ArrayBuffer.cpp (JavascriptArrayBuffer::JavascriptArrayBuffer)," [Online]. Available: <https://github.com/Microsoft/ChakraCore/blob/447fa0df2b64717c15ae95a6e4d054ab11ce5be5/lib/Runtime>

/Library/ArrayBuffer.cpp#L761-L764.

- [14] M. Tomassoli, "IE10: Reverse Engineering IE," [Online]. Available: <http://expdev-kiuhnm.rhcloud.com/2015/05/31/ie10-reverse-engineering-ie/>.
- [15] Microsoft, "GitHub: Microsoft / ChakraCore / lib / Common / ConfigFlagsList.h (DEFAULT\_CONFIG\_CollectGarbage)," [Online]. Available: <https://github.com/Microsoft/ChakraCore/blob/5df046f68f37e8170d0fe5c6cd183fc69bbc10da/lib/Common/ConfigFlagsList.h#L479-L480>.
- [16] Microsoft, "GitHub: Microsoft / ChakraCore / lib / Runtime / Library / ArrayBuffer.cpp (JavascriptArrayBuffer::Create)," [Online]. Available: <https://github.com/Microsoft/ChakraCore/blob/447fa0df2b64717c15ae95a6e4d054ab11ce5be5/lib/Runtime/Library/ArrayBuffer.cpp#L779>.