

Modern Windows Pwn

Userland Heap Exploitation

2st 23/5/2022

Windows Heap Exploitation

- Windows memory allocator
- NT Heap
 - Backend
 - Alloc / Free / Exploitation
 - FrontEnd
 - Low Fragmentation Heap
 - Alloc / Free / Exploitation
- Segment Heap

Windows Memory Allocator

- NT Heap
 - Windows userland default heap
 - BackEnd
 - FrontEnd
 - Size <= 0x4000
 - Avoid fragmented memory blocks
 - In used when same size memory blocks were requested 18 times

Windows Memory Allocator

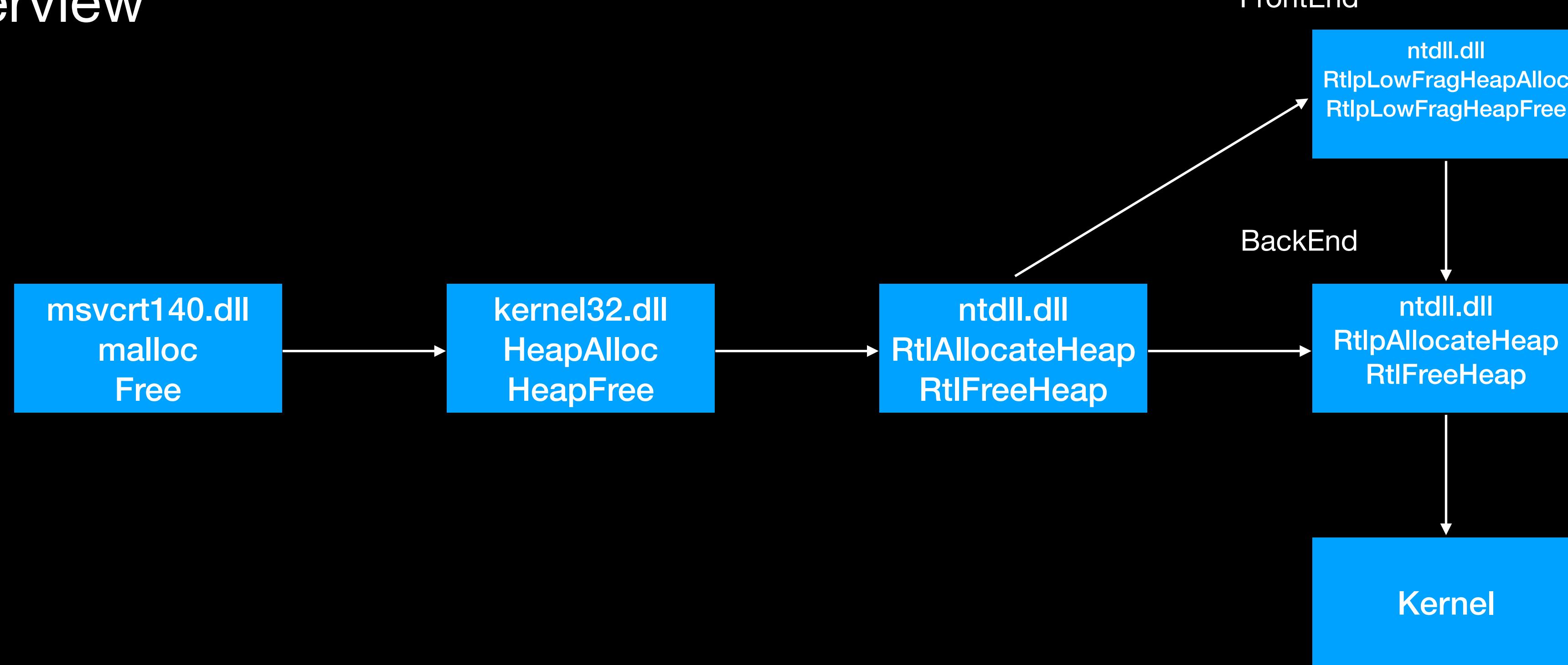
- Segment Heap
 - Used in Windows UWP programs

Windows Memory Allocator

- Tested on Windows 10 1803 64-bit

Windows Memory Allocator

- Overview



Windows Memory Allocator

- Heap types
 - Default heap
 - Shared in process, stored in _PEB
 - CRT heap
 - malloc / new
 - Private heap
 - HeapCreate

Windows Memory Allocator

- Core data structure
 - _HEAP
 - Heap manager
 - Usually at the beginning of heap
 - _HEAP_ENTRY(chunk)
 - memory block
 - common structure used in both FrontEnd and BackEnd

Windows Memory Allocator

- _HEAP overview

The image shows two windows from a debugger, likely Immunity Debugger, displaying memory dump information for Windows memory structures.

Top Window (ntdll!_HEAP):

```
1: kd> dt _heap 9f0000
ntdll!_HEAP
+0x000 Segment : _HEAP_SEGMENT
+0x000 Entry : _HEAP_ENTRY
+0x010 SegmentSignature : 0xffeffee
+0x014 SegmentFlags : 2
+0x018 SegmentListEntry : _LIST_ENTRY [ 0x00000000`009f0120 - 0x00000000`009f0120 ]
+0x028 Heap : 0x00000000`009f0000 _HEAP
+0x030 BaseAddress : 0x00000000`009f0000 Void
+0x038 NumberOfPages : 0xf
+0x040 FirstEntry : 0x00000000`009f0720 _HEAP_ENTRY
+0x048 LastValidEntry : 0x00000000`009ff000 _HEAP_ENTRY
+0x050 NumberOfUnCommittedPages : 9
+0x054 NumberOfUnCommittedRanges : 1
+0x058 SegmentAllocatorBackTraceIndex : 0
+0x05a Reserved : 0
+0x060 UCRSegmentList : _LIST_ENTRY [ 0x00000000`009f5fe0 - 0x00000000`009f5fe0 ]
+0x070 Flags : 0x1002
+0x074 ForceFlags : 0
+0x078 CompatibilityFlags : 0
+0x07c EncodeFlagMask : 0x100000
+0x080 Encoding : _HEAP_ENTRY
+0x090 Interceptor : 0
+0x094 VirtualMemoryThreshold : 0xffff
+0x098 Signature : 0xeffeff
+0x0a0 SegmentReserve : 0x100000
+0x0a8 SegmentCommit : 0x2000
+0x0b0 DeCommitFreeBlockThreshold : 0x400
+0x0b8 DeCommitTotalFreeThreshold : 0x1000
+0x0c0 TotalFreeSize : 0x120
+0x0c8 MaximumAllocationSize : 0x00007fff fffffeff
+0x0d0 ProcessHeapsListIndex : 4
+0x0d2 HeaderValidateLength : 0x2a0
+0x0d8 HeaderValidateCopy : (null)
+0x0e0 NextAvailableTagIndex : 0
+0x0e2 MaximumTagIndex : 0
+0x0e8 TagEntries : (null)
+0x0f0 UCRList : _LIST_ENTRY [ 0x00000000`009f5fd0 - 0x00000000`009f5fd0 ]
+0x100 AlignRound : 0x1f
+0x108 AlignMask : 0xffffffff ffffffff
+0x110 VirtualAllocdBlocks : _LIST_ENTRY [ 0x00000000`009f0110 - 0x00000000`009f0110 ]
+0x120 SegmentList : _LIST_ENTRY [ 0x00000000`009f0018 - 0x00000000`009f0018 ]
+0x130 AllocatorBackTraceIndex : 0
+0x134 NonDedicatedListLength : 0
+0x138 BlocksIndex : 0x00000000`009f02c8 Void
+0x140 UCRIndex : (null)
+0x148 PseudoTagEntries : (null)
+0x150 FreeLists : _LIST_ENTRY [ 0x00000000`009f0840 - 0x00000000`009f5320 ]
+0x160 LockVariable : 0x00000000`009f02a0 _HEAP_LOCK
+0x168 CommitRoutine : 0x1a9523e9`4d53c99c long +1a9523e94d53c99c
+0x170 StackTraceInitVar : _RTL_RUN_ONCE
+0x178 FrontEndHeap : 0x00000000`00070000 Void
+0x180 FrontEndHeapLockCount : 0
+0x182 FrontEndHeapType : 0x2 "
+0x183 RequestedFrontEndHeapType : 0x2 "
+0x188 FrontEndHeapUsageData : 0x00000000`009f1a50 -> 0
+0x190 FrontEndHeapMaximumIndex : 0x402
+0x192 FrontEndHeapStatusBitmap : [129] ""
+0x218 Counters : _HEAP_COUNTERS
+0x290 TuningParameters : _HEAP_TUNING_PARAMETERS
```

Bottom Window (ntdll!_LFH_HEAP):

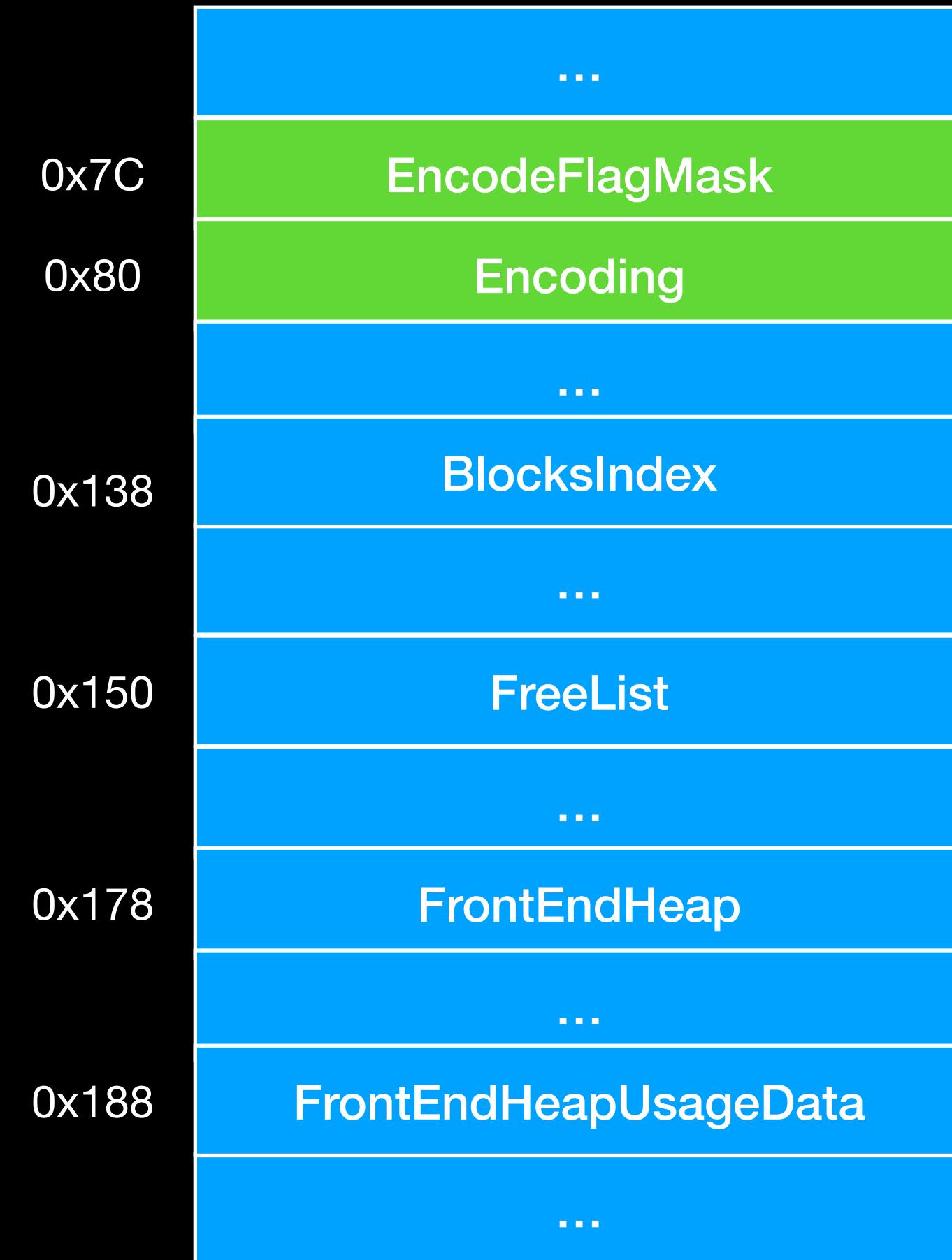
```
1: kd> dt _lfh_heap 70000
ntdll!_LFH_HEAP
+0x000 Lock : _RTL_SRWLOCK
+0x008 SubSegmentZones : _LIST_ENTRY [ 0x00000000`009f4f20 - 0x00000000`009f4f20 ]
+0x018 Heap : 0x00000000`009f0000 Void
+0x020 NextSegmentInfoArrayAddress : 0x00000000`00070de0 Void
+0x028 FirstUncommittedAddress : 0x00000000`00071000 Void
+0x030 ReservedAddressLimit : 0x00000000`0007d000 Void
+0x038 SegmentCreate : 1
+0x03c SegmentDelete : 0
+0x040 MinimumCacheDepth : 0
+0x044 CacheShiftThreshold : 0
+0x048 SizeInCache : 0
+0x050 RunInfo : _HEAP_BUCKET_RUN_INFO
+0x060 UserBlockCache : [12] _USER_MEMORY_CACHE_ENTRY
+0x2a0 MemoryPolicies : HEAP_LFH_MEM_POLICIES
```

Bucket[0] (Bottom Left):

```
[+0x000] BlockUnits : 0x1 [Type: unsigned short]
[+0x002] SizeIndex : 0x0 [Type: unsigned char]
[+0x003 ( 0: 0)] UseAffinity : 0x0 [Type: unsigned char]
```

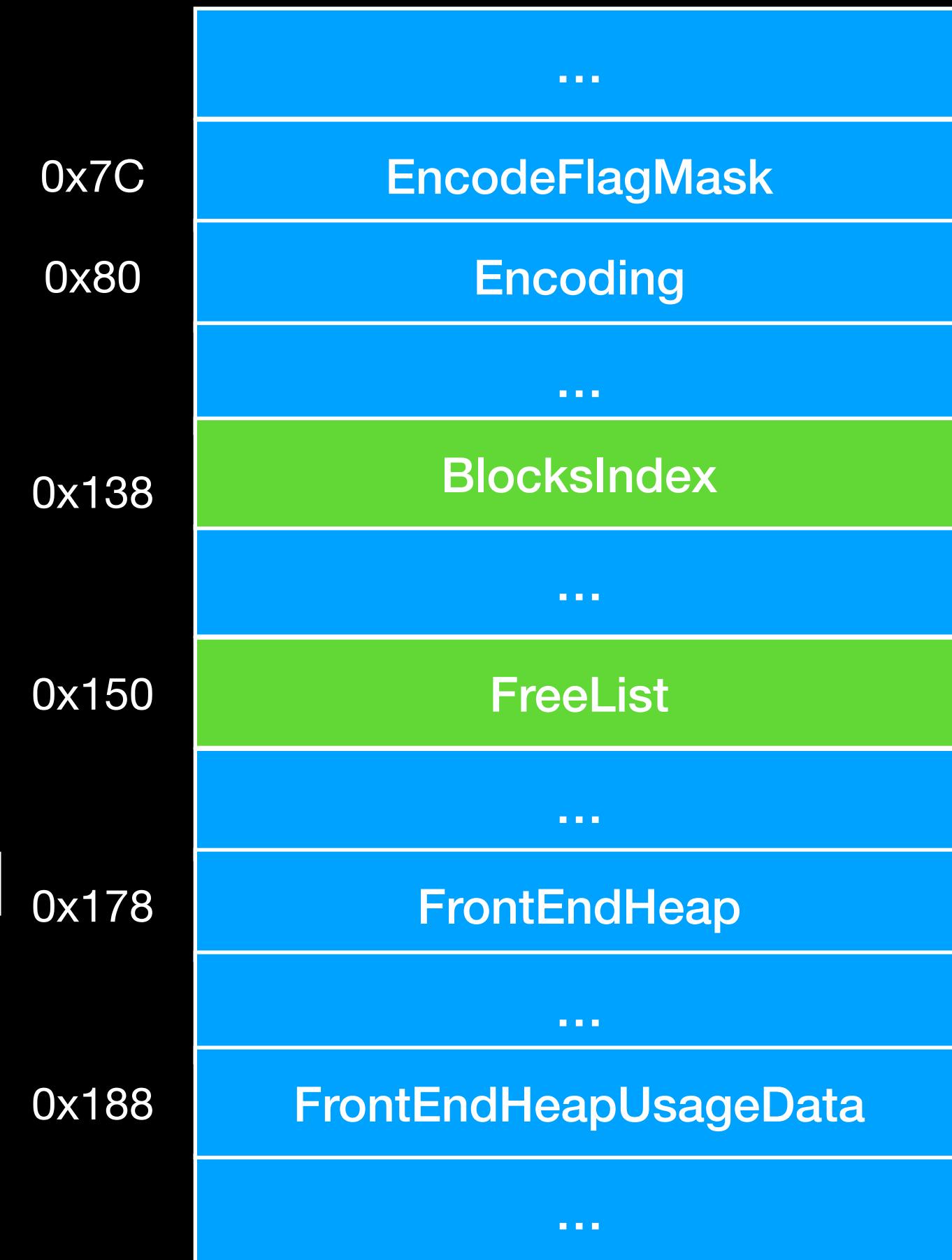
Windows Memory Allocator

- `_HEAP`
 - `EncodeFlagMask`
 - init 0x100000, decide chunks' header are encoded or not in this heap
 - `Encoding`
 - xor cookies, to encode chunk header



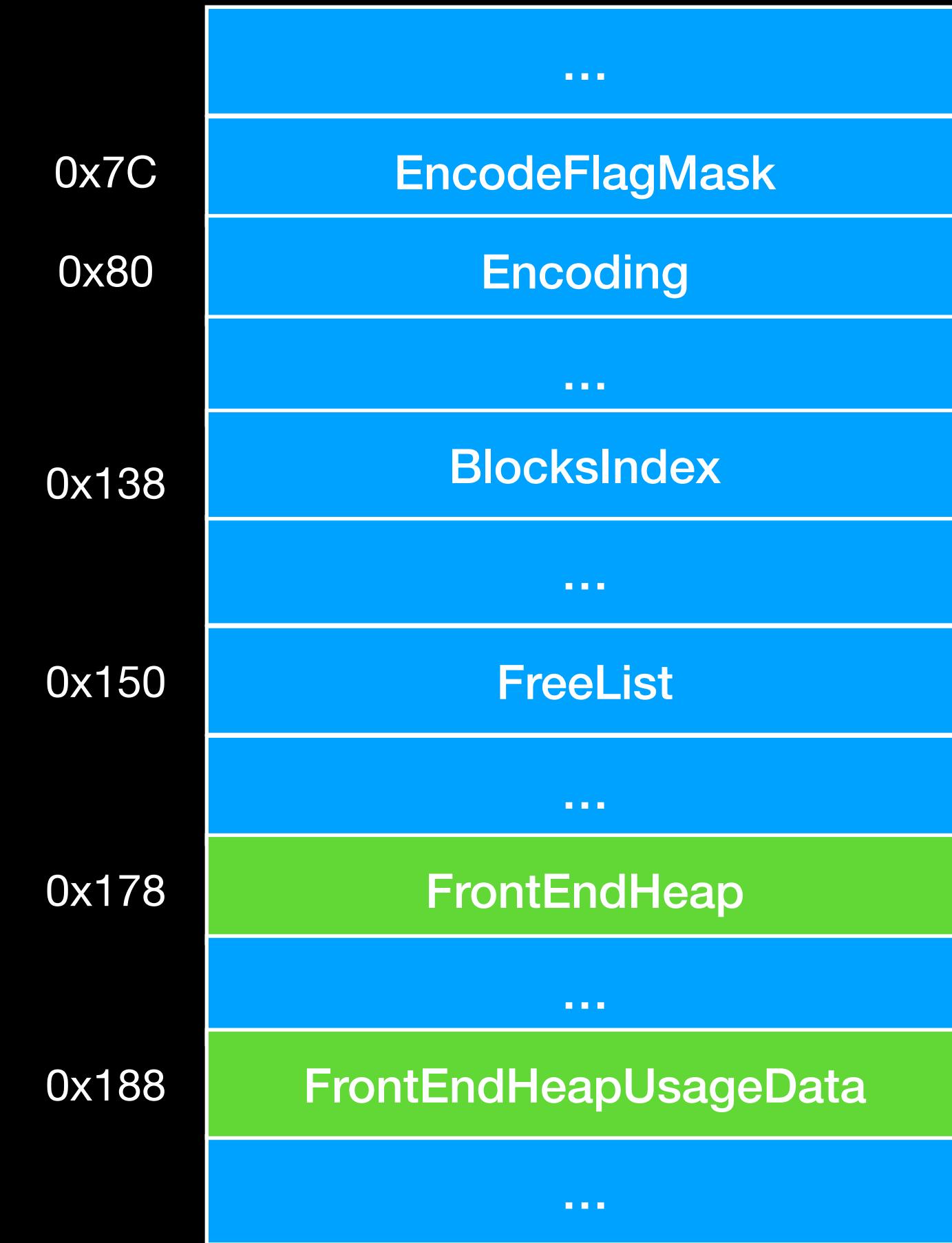
Windows Memory Allocator

- `_HEAP`
 - `BlocksIndex`
 - chunks manager in BackEnd
 - `FreeList`
 - free chunks double linked list in BackEnd
 - sorted



Windows Memory Allocator

- `_HEAP`
 - `FrontEndHeap`
 - Pointer to FrontEnd heap
 - `FrontEndHeapUsageData`
 - WORD array
 - Stores number of times a chunk of corresponding size is used, in order to enable LFH



Windows Memory Allocator

- NT heap
- Common data structure
 - BackEnd
 - Allocation mechanism
 - BackEnd heap exploitation
 - FrontEnd
 - Allocation mechanism
 - FrontEnd heap exploitation

Windows Memory Allocator

- memory chunks (`_HEAP_ENTRY`)
 - Status
 - Allocated chunk
 - Freed chunk
 - VirtualAlloc chunk (`_HEAP_VIRTUAL_ALLOC_ENTRY`)

Windows Memory Allocator

in used chunk

- PreviousBlockPrivateData
 - previous chunk data
- Size
 - value = real size $\gg 4$
- Flag
 - if inused
- SmallTagIndex
 - value = $(size \& 0xff) \wedge (size \gg 8) \wedge Flag$



Windows Memory Allocator

in used chunk

- PreviousSize
 - size of previous chunk (real size $>> 4$)
- SegmentOffset
 - To find segment
- Unusedbyte
 - Judge chunk status
 - BackEnd or FrontEnd
 - in used or free
- User data space
 - return address



Windows Memory Allocator

freed chunk

- Flink
 - point to next chunk on double linked list
- Blink
 - point to previous chunk on double linked list
- Unusedbyte
 - 0



Windows Memory Allocator

in used mmap chunk

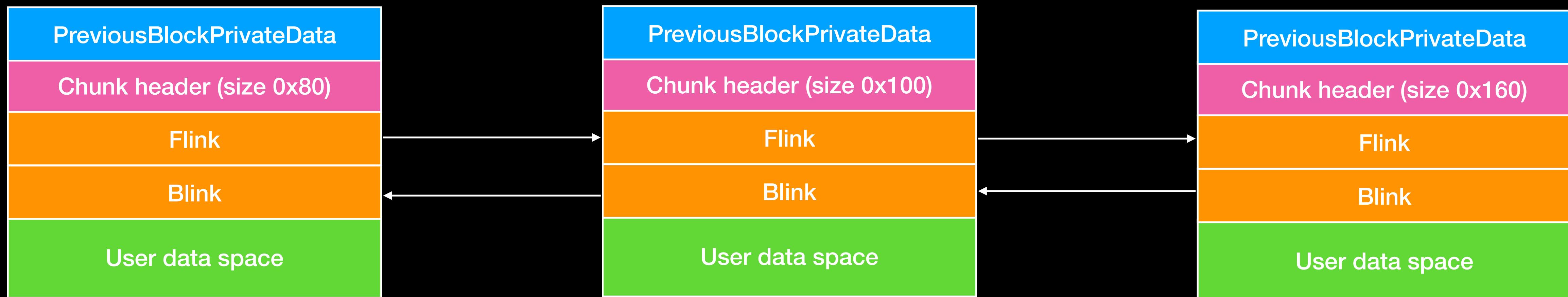
- Flink
 - point to next chunk on double linked list
- Blink
 - point to previous chunk on double linked list



Windows Memory Allocator

FreeLists(_HEAP_ENTRY)

- After a chunk is freed, it will be inserted in FreeLists by size



Windows Memory Allocator

Remark

- header encoding
 - Stored value: $\text{encode_header} = \text{header} \wedge (\text{_HEAP}\rightarrow\text{Encoding})$
 - Decode check
 - $\text{val} = \text{encode_header} \wedge (\text{_HEAP}\rightarrow\text{Encoding})$
 - $\text{val}[3] = \text{val}[0] \wedge \text{val}[1] \wedge \text{val}[2] ?$

Windows Memory Allocator

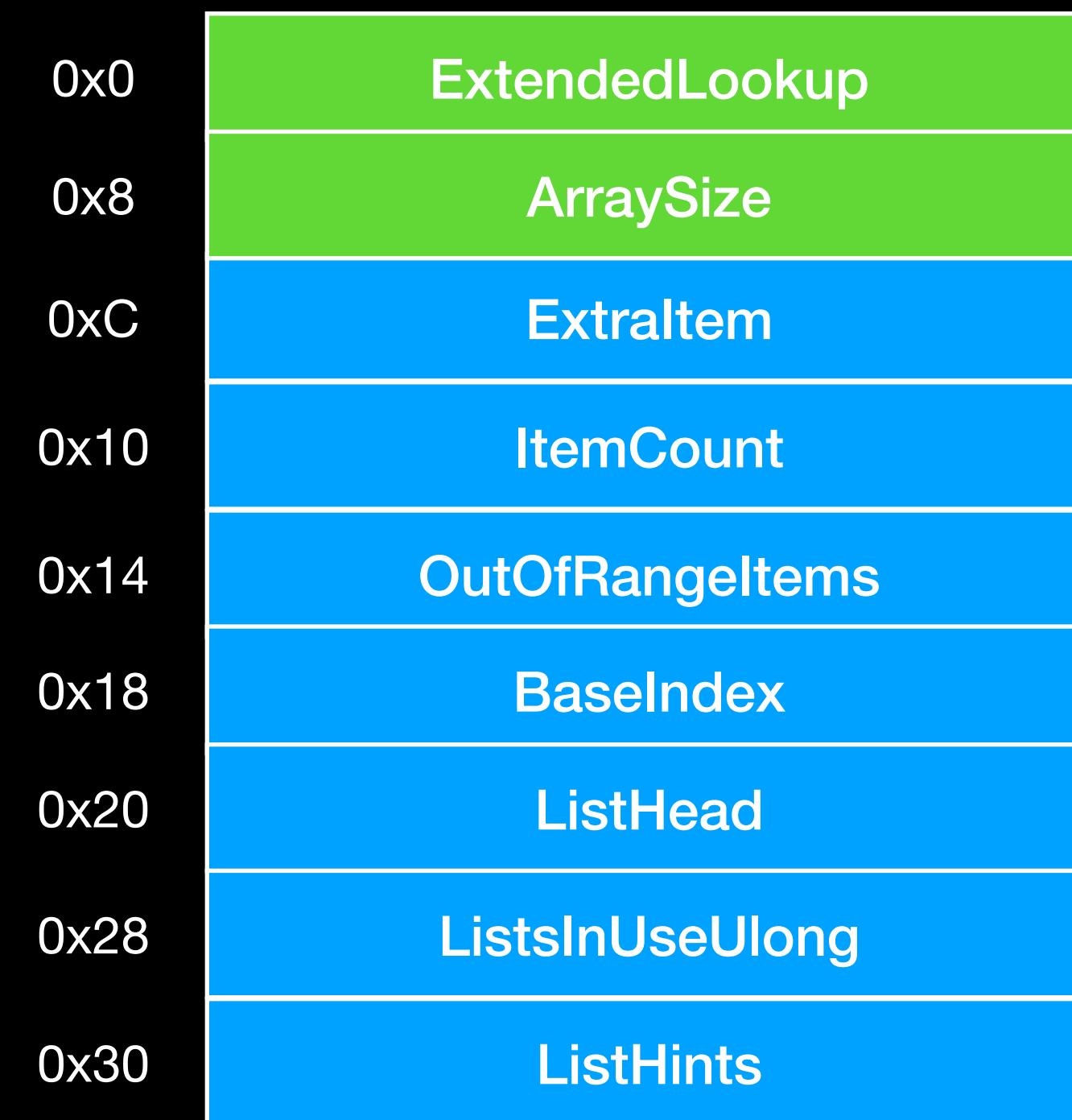
BlocksIndex

- manage various size freed chunks
 - Fast lookup

Windows Memory Allocator

BlocksIndex

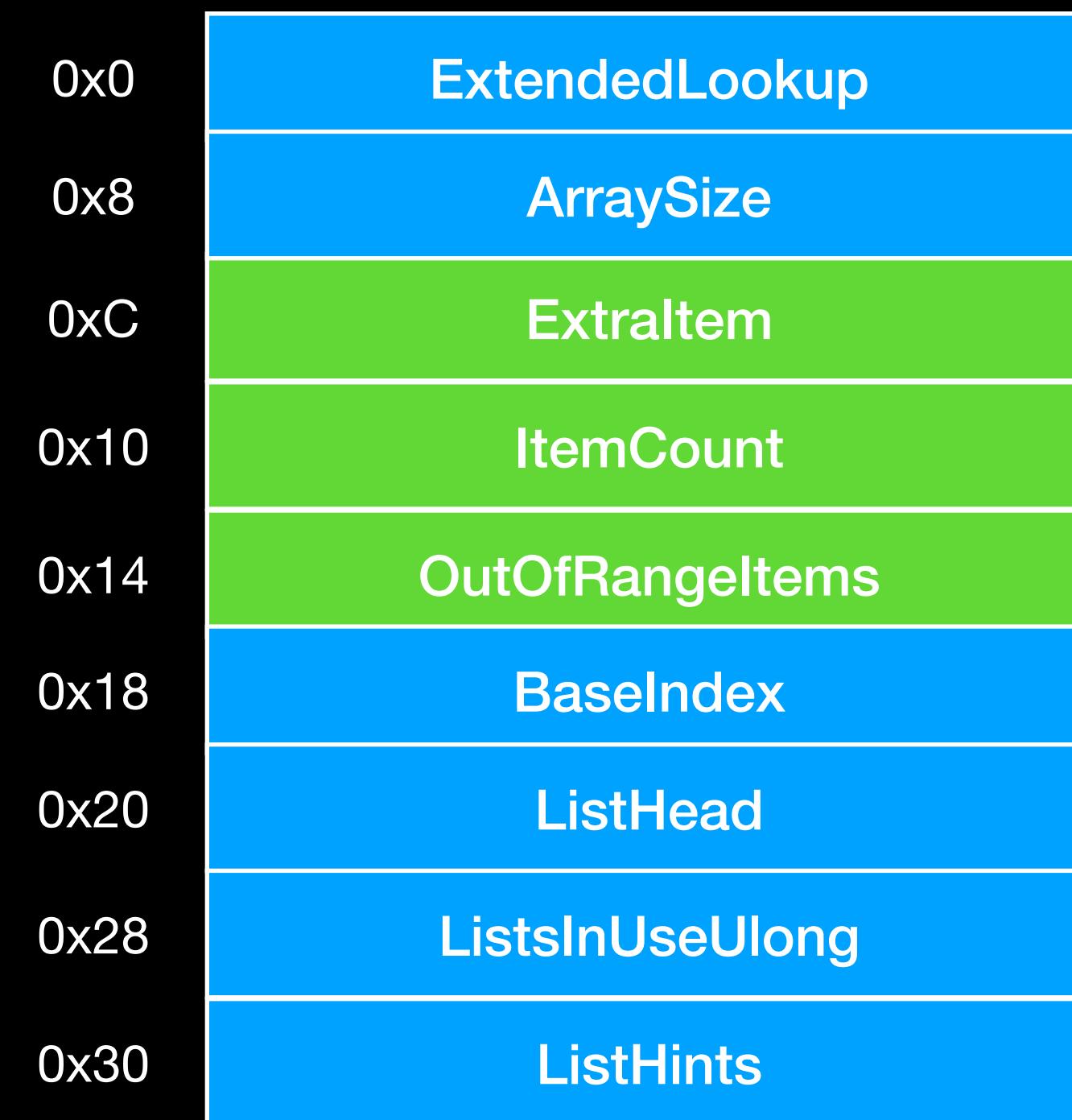
- `_HEAP_LIST_LOOKUP`
- `ExtendedLookup(_HEAP_LIST_LOOKUP)`
 - point to next `ExtendedLookup`
 - Usually, next `ExtendedLookup` manage bigger chunk
- `ArraySize`
 - Max chunk size in current lookup
 - Usually 0x80



Windows Memory Allocator

BlocksIndex

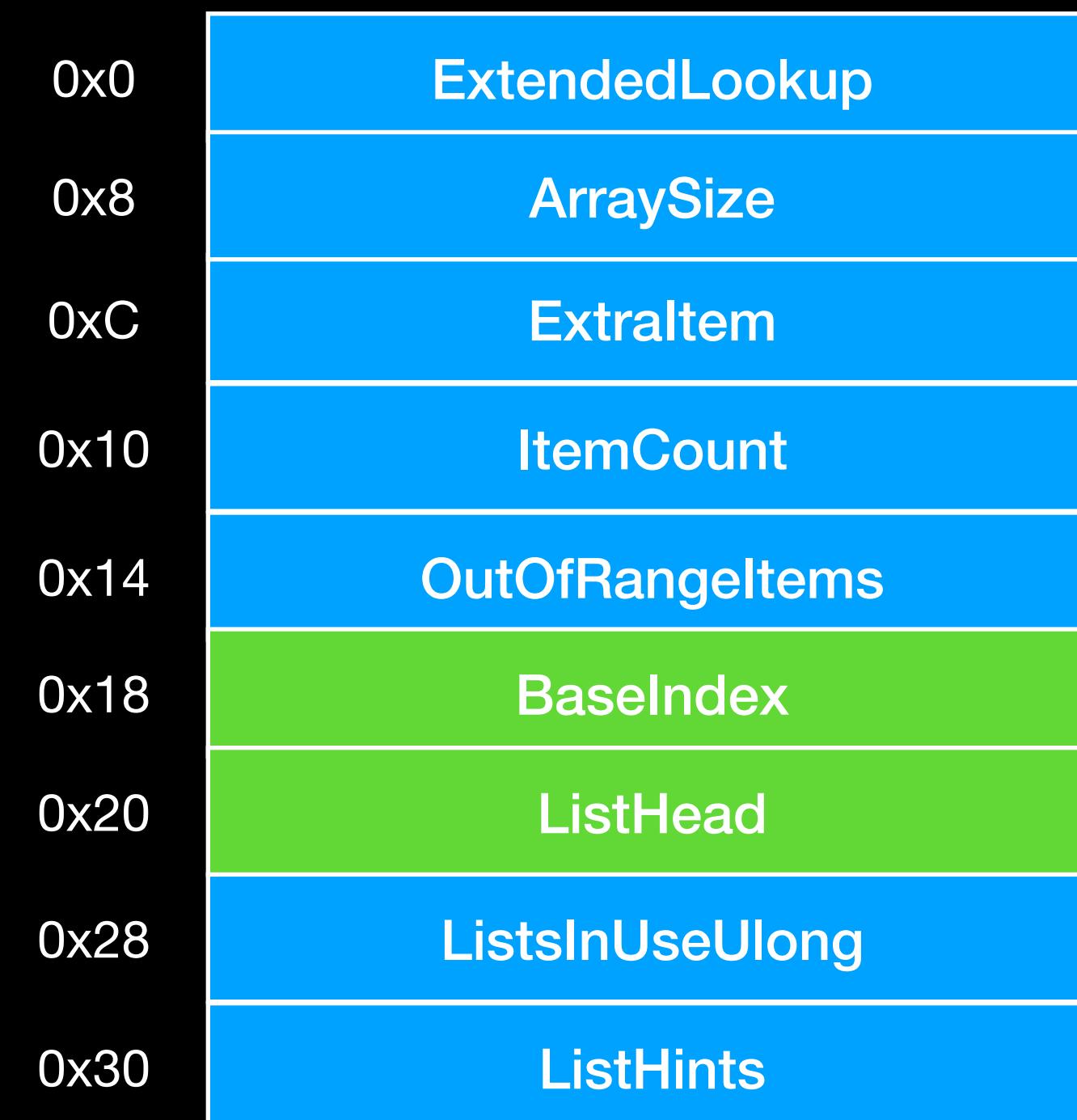
- **Extraltem**
 - Indicate whether each member in the **ListHints** array is composed of one pointer or two pointers
- **ItemCount**
 - Number of chunks managed by the current structure
- **OutOfRangItems**
 - Number of chunks exceeding the current structure management size



Windows Memory Allocator

BlocksIndex

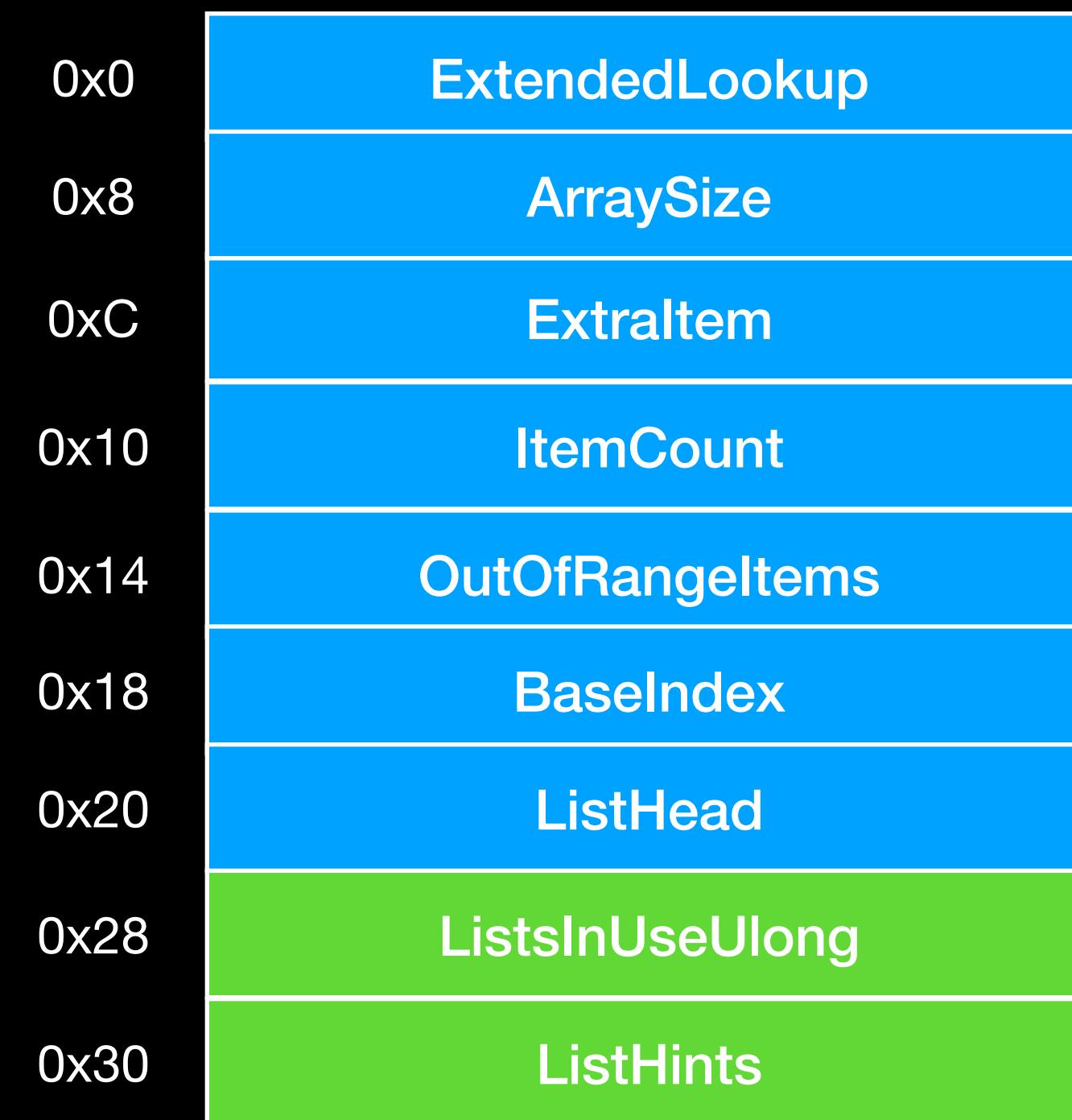
- **BaseIndex**
 - The starting index of the chunk managed by this structure is used to find a free chunk of suitable size from ListHint, and the next BlocksIndex will be the maximum value of this index to the Base Index
- **ListHead**
 - FreeList Head



Windows Memory Allocator

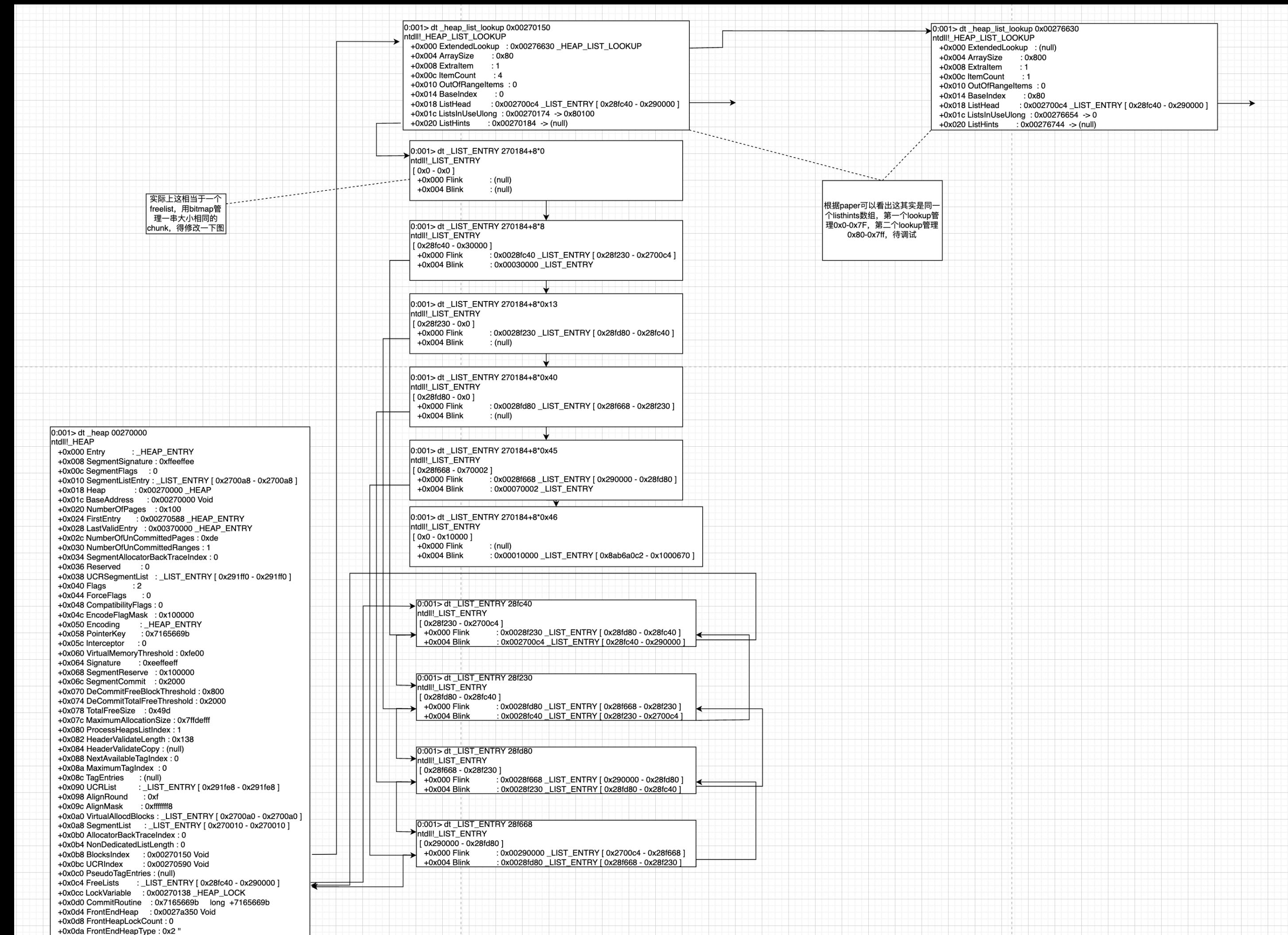
BlocksIndex

- `ListsInUseUlong`
 - Bitmap for determining whether there is a suitable size chunk in `ListHint`
- `ListHint`
 - Array of pointers (to chunk)
 - `Fast`



Windows Memory Allocator

BlocksIndex overview



Windows Memory Allocator

BackEnd allocation mechanism

- Allocate (RtlAllocateHeap)
 - Size <= 0x4000
 - 0x4000 < Size <= 0xff000
 - 0xff000 < Size

Windows Memory Allocator

BackEnd allocation mechanism

- Size <= 0x4000
 - Allocated in RtlpAllocateHeap
 - check if current size's LFH is enabled in corresponding FrontEndHeapStatusBitmap
 - if not, corresponding FrontEndHeapUsageData add 0x21
 - check if FrontEndHeapUsageData > 0xff00 or (FrontEndHeapUsageData & 0x1f) > 0x10 to enable LFH

Windows Memory Allocator

BackEnd allocation mechanism

- Size <= 0x4000
- check chunks in ListHint
 - Found suitable chunks on ListHint, remove it, check if chunk->flink the same size as chunk
 - Yes, ListHint = chunk->flink
 - No, ListHint = NULL
 - unlink this chunk to user, xor header

Windows Memory Allocator

BackEnd allocation mechanism

- Size <= 0x4000
- check chunks in ListHint
 - Not found suitable chunks on ListHint
 - Try to find in bigger ListHint
 - Found, same remove operation on ListHint, unlink it from FreeList
 - cut this chunk, try to add left part to FreeList and ListHint(xor header)
 - return chunk to user

Windows Memory Allocator

BackEnd allocation mechanism

- Size <= 0x4000
- check chunks in ListHint
 - Not found suitable chunks on ListHint
 - Try to find in bigger ListHint
 - Not found, extend heap through ExtendHeap
 - same operation as above

Windows Memory Allocator

BackEnd allocation mechanism

- $0x4000 < \text{Size} \leq 0xff000$
- same operations as $\text{size} \leq 0x4000$ except LFH operations

Windows Memory Allocator

BackEnd allocation mechanism

- 0xff000 (`_HEAP->VirtualMemoryThreshold << 4`) < Size
- `ZwAllocateVirtualMemory`
- Insert in `_HEAP->VirtualAllocdBlocks`
 - Used for `VirutalAlloc` on heap

Windows Memory Allocator

BackEnd allocation mechanism

- Free (RtlpFreeHeap)
 - Size <= 0xff000
 - 0xff000 < Size

Windows Memory Allocator

BackEnd allocation mechanism

- Size <= 0xff000
 - check alignment, check chunk status by unused byte
 - If not LFH, corresponding FrontEndHeapUsageData - 1
 - check if forward adjoin chunk1 freed
 - Yes, unlink chunk1 and remove it from ListHint chunks merge
 - remove operations same as above

Windows Memory Allocator

BackEnd allocation mechanism

- Size <= 0xff000
 - After merging, update size & previous size, check if self is the first or last chunk in freelist. Yes, insert into freelist. No, insert into & update ListHint
 - When insert happening, windows will check chunk->Flink->Blink == chunk->Blink->Flink, but this check procedure will not abort process

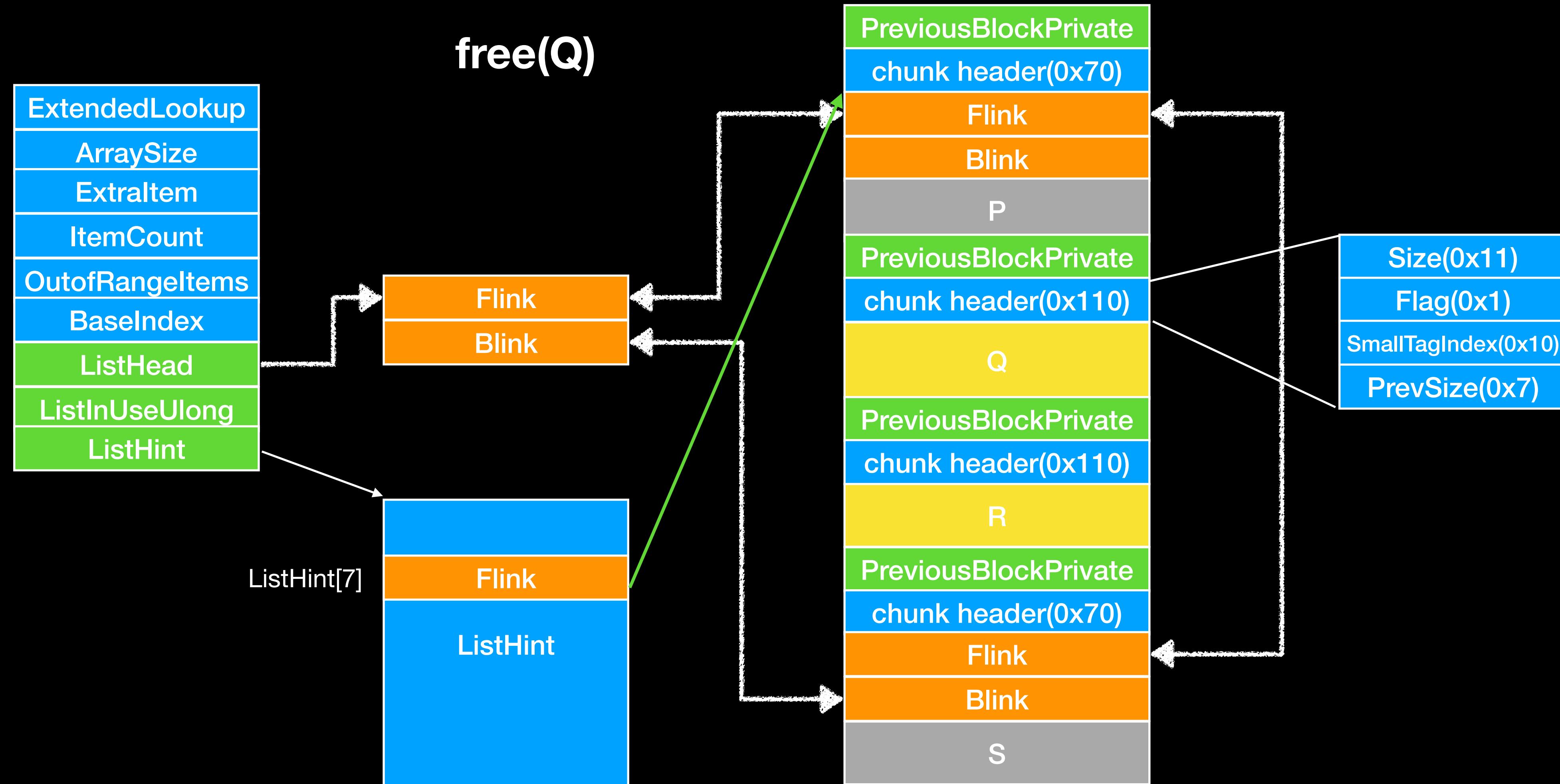
Windows Memory Allocator

BackEnd allocation mechanism

- $0\text{xff000} < \text{Size}$
 - check chunk's linked list and remove it from `_HEAP->VirtualAllocdBlocks`
 - unmap this chunk by `RltpSecMemFreeVirtualMemory`

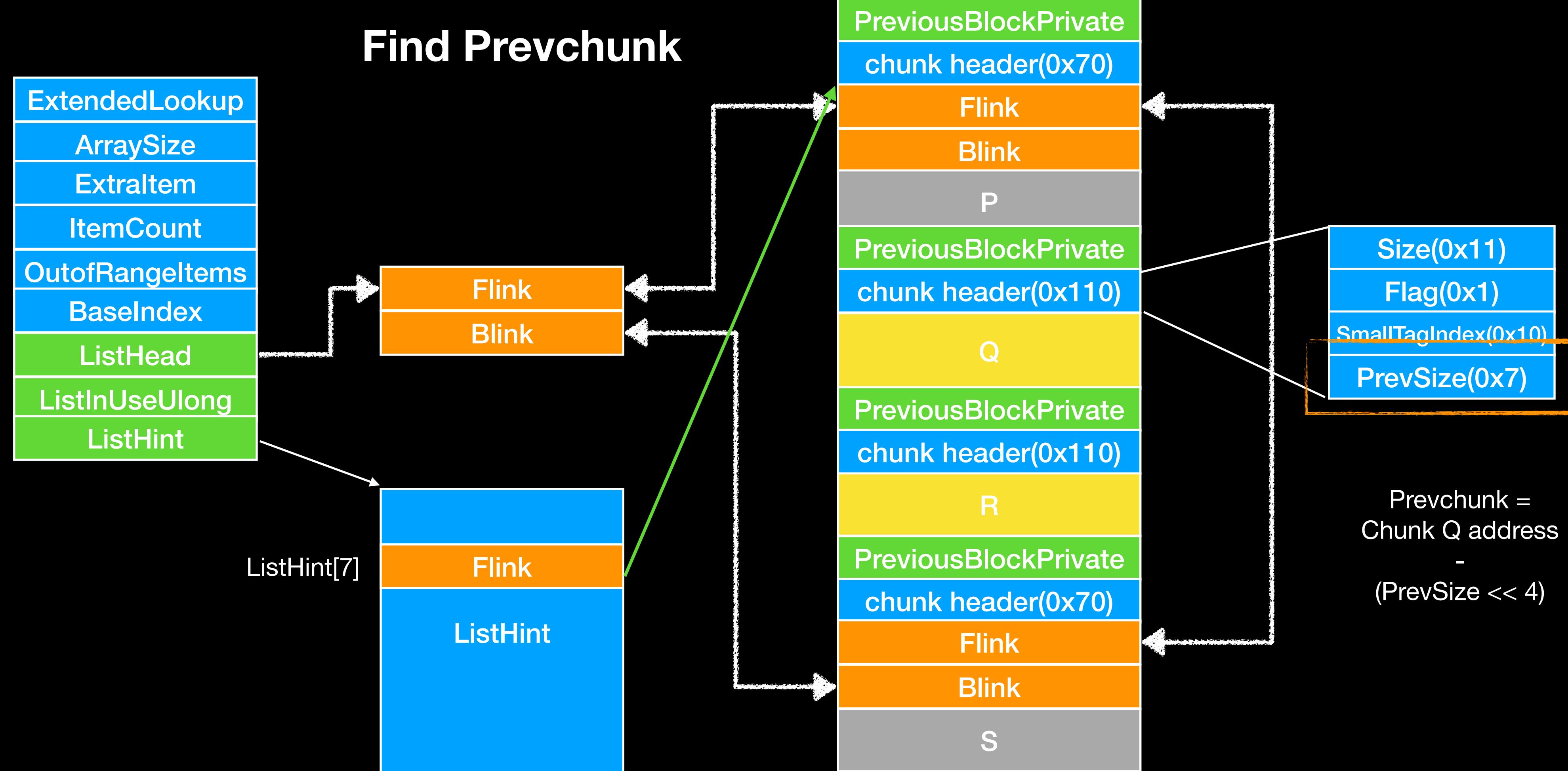
Windows Memory Allocator

merge chunk



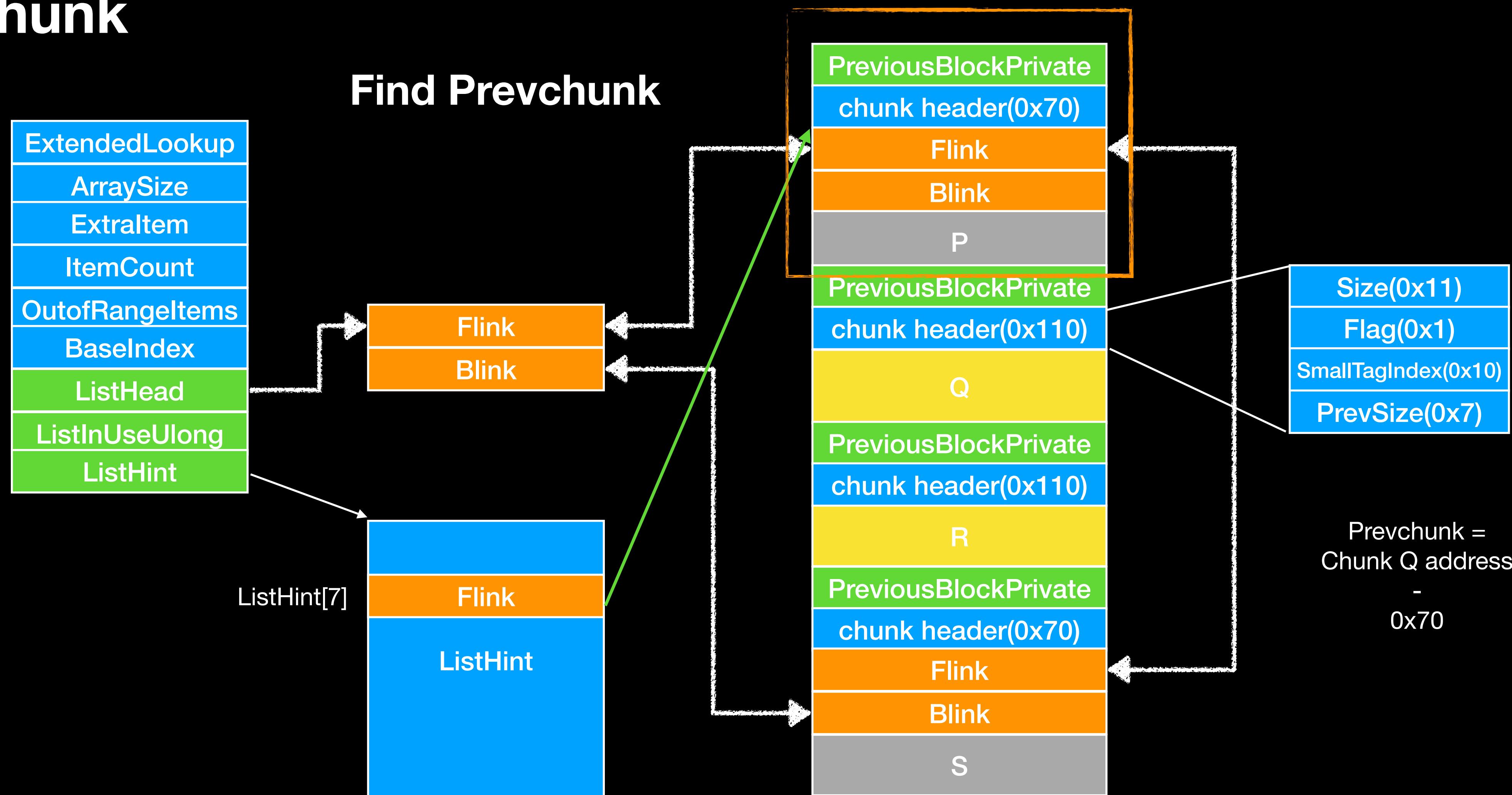
Windows Memory Allocator

merge chunk



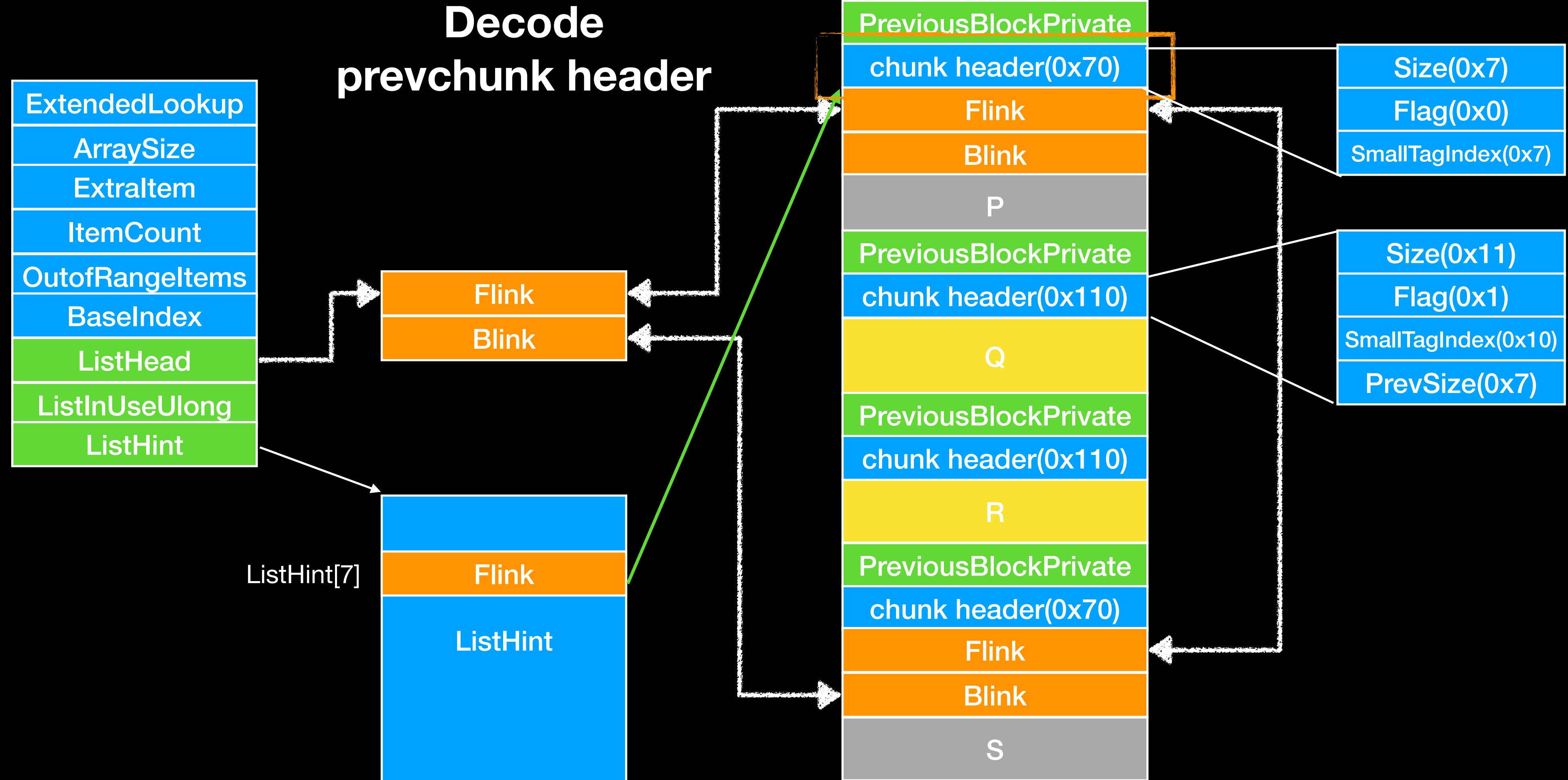
Windows Memory Allocator

merge chunk



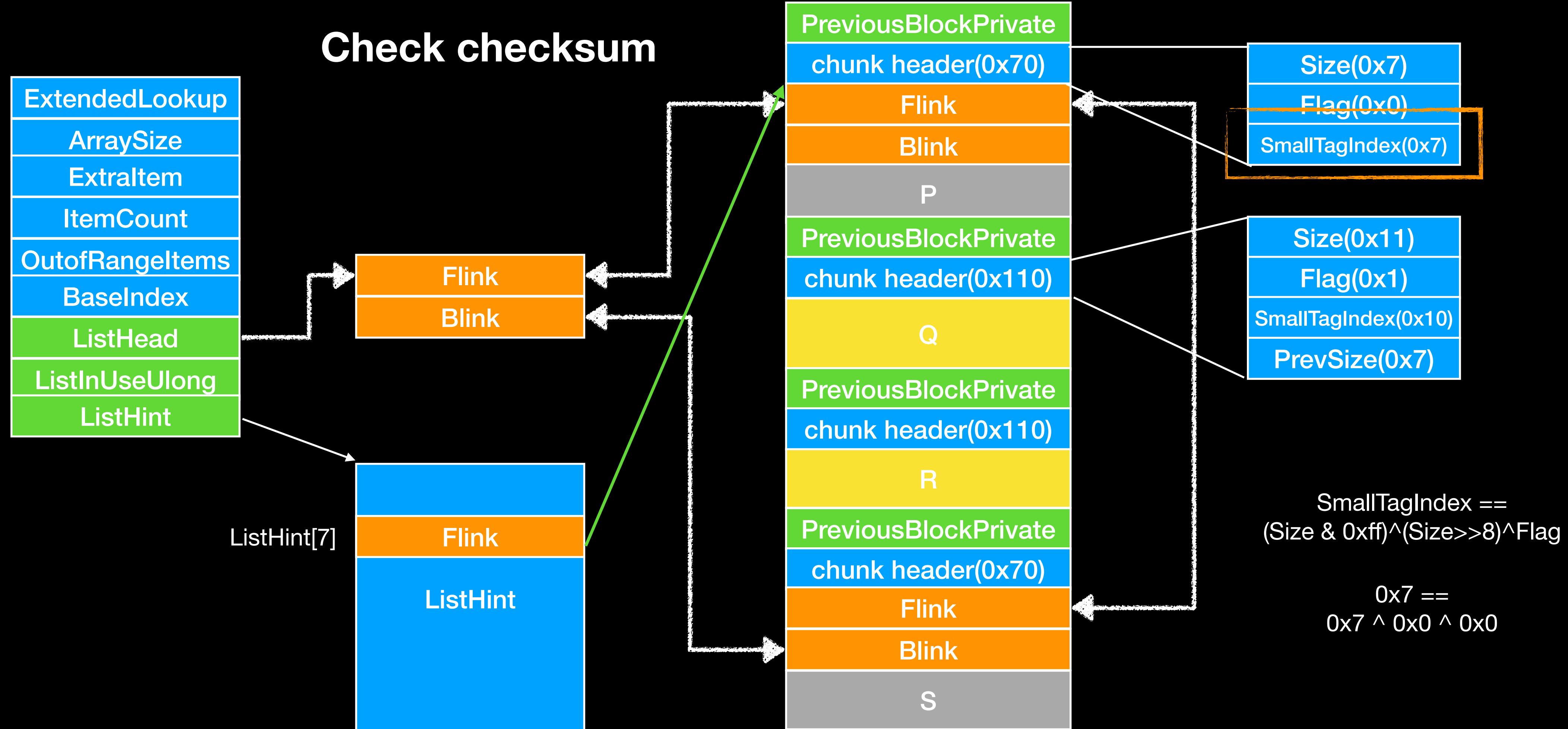
Windows Memory Allocator

merge chunk



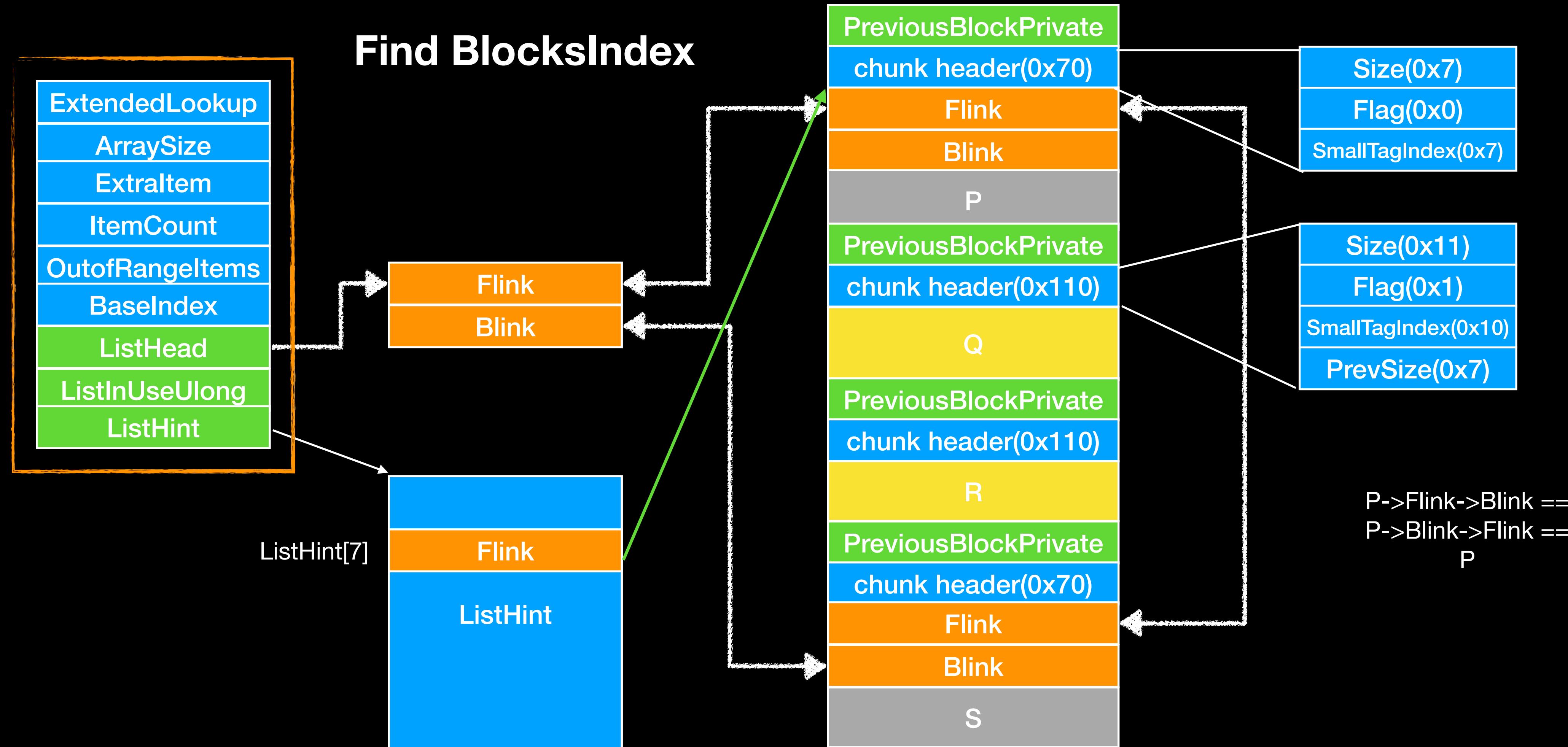
Windows Memory Allocator

merge chunk



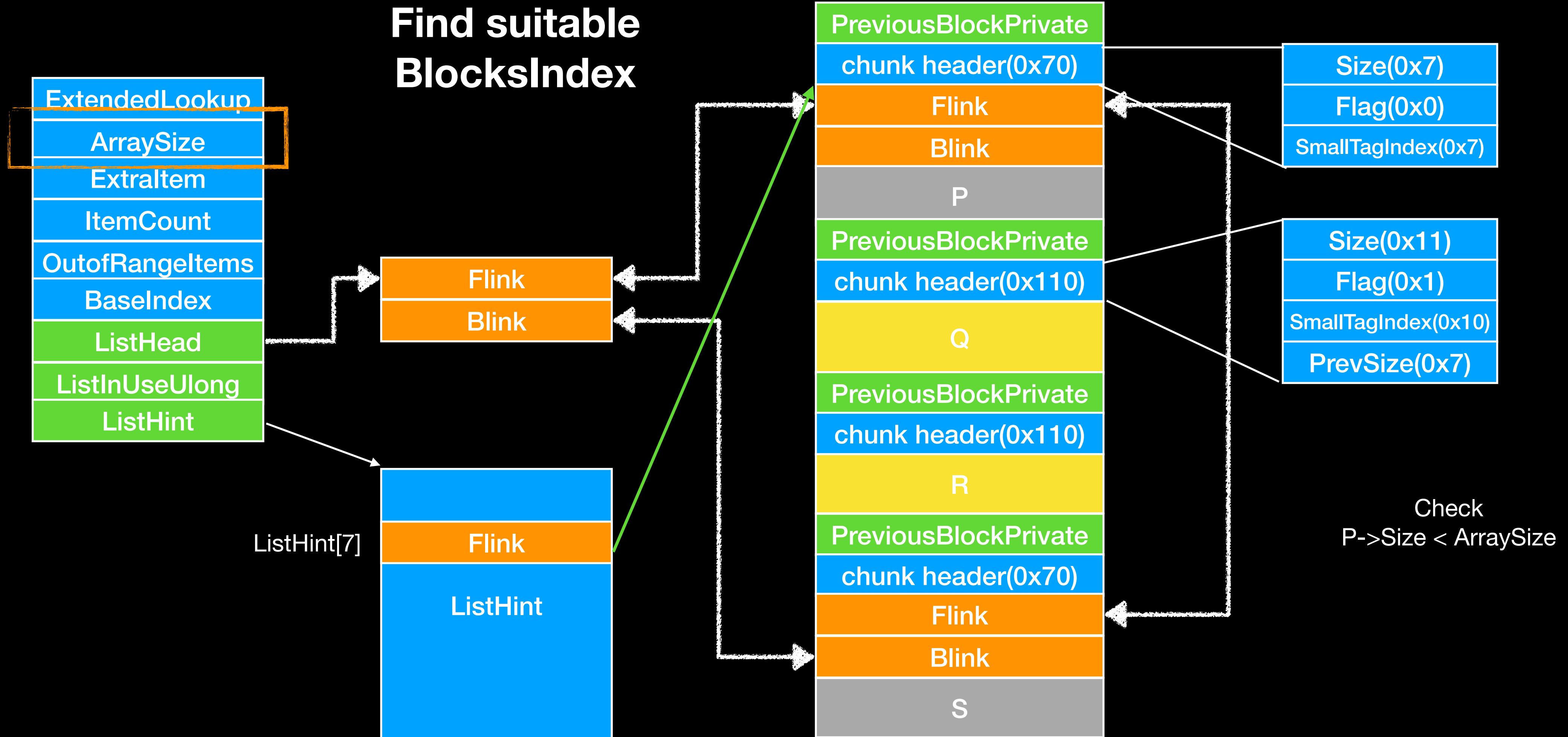
Windows Memory Allocator

merge chunk



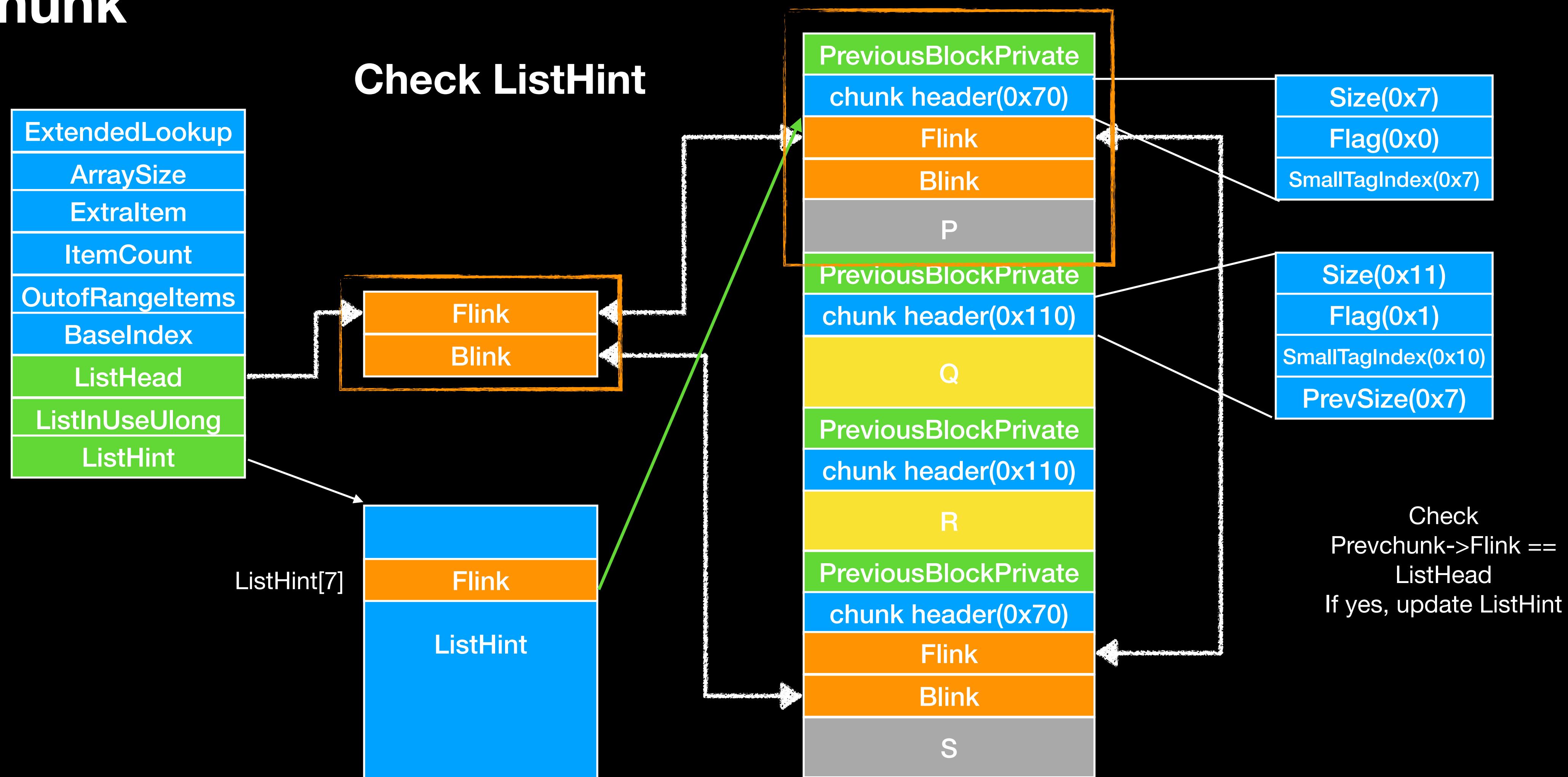
Windows Memory Allocator

merge chunk



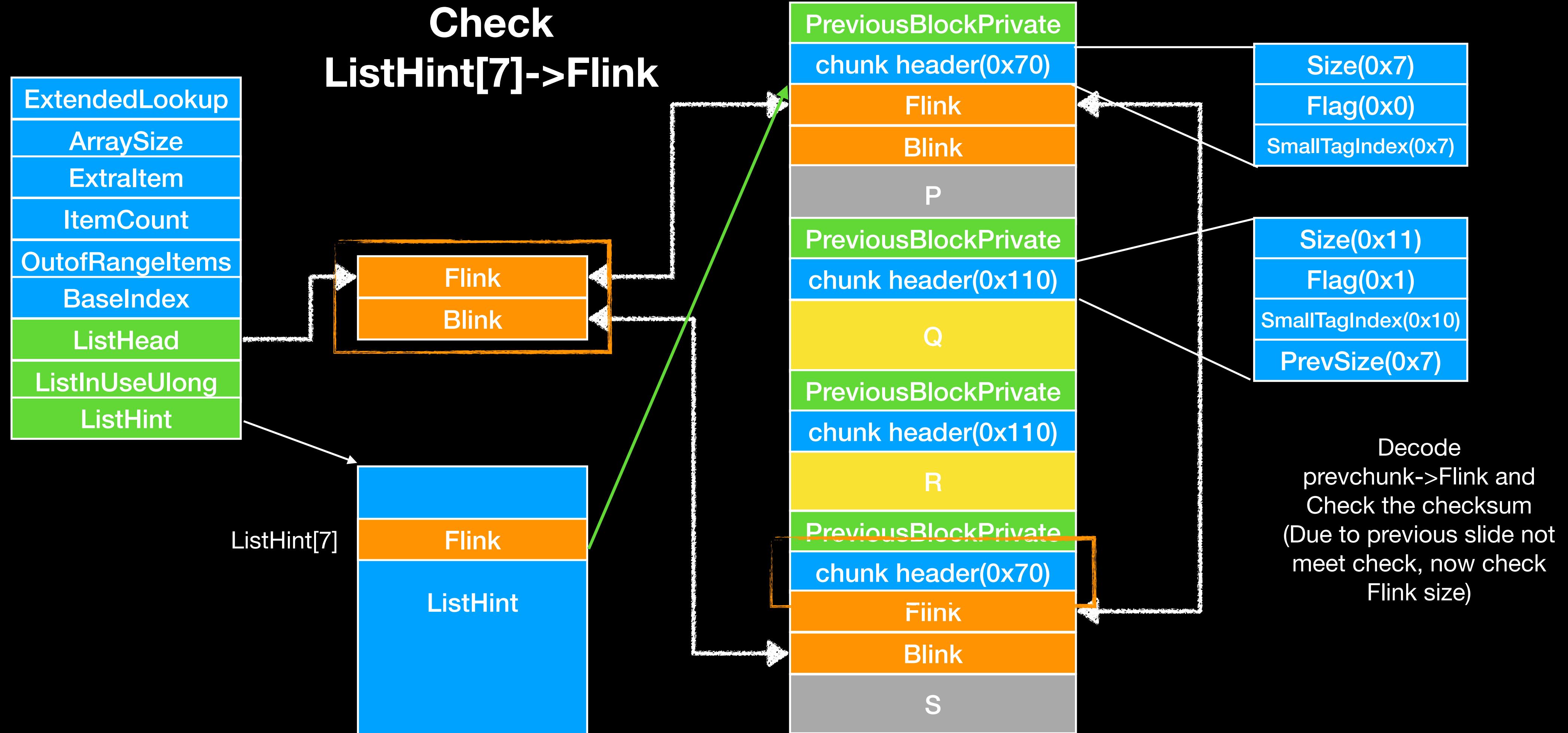
Windows Memory Allocator

merge chunk



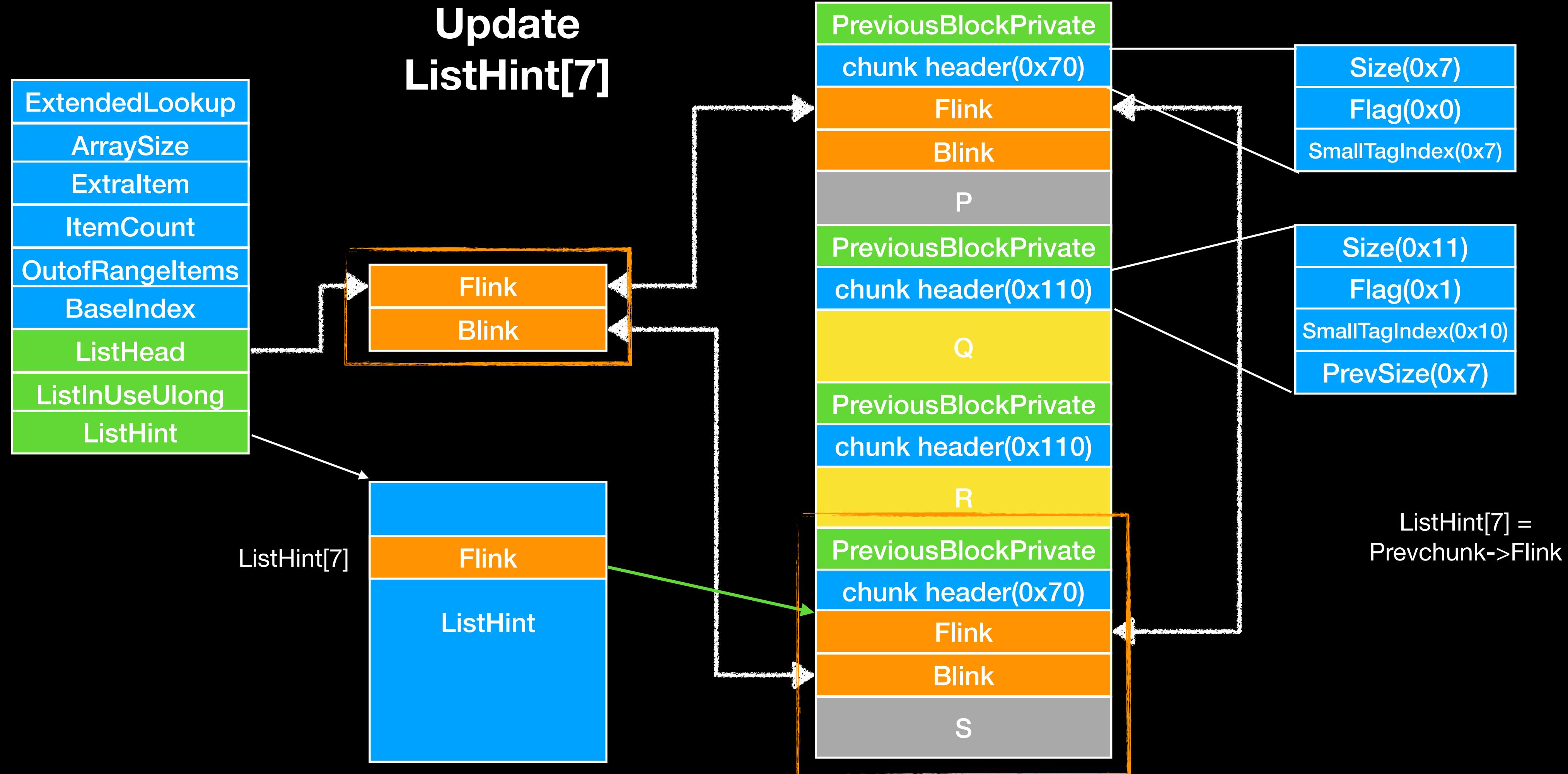
Windows Memory Allocator

merge chunk



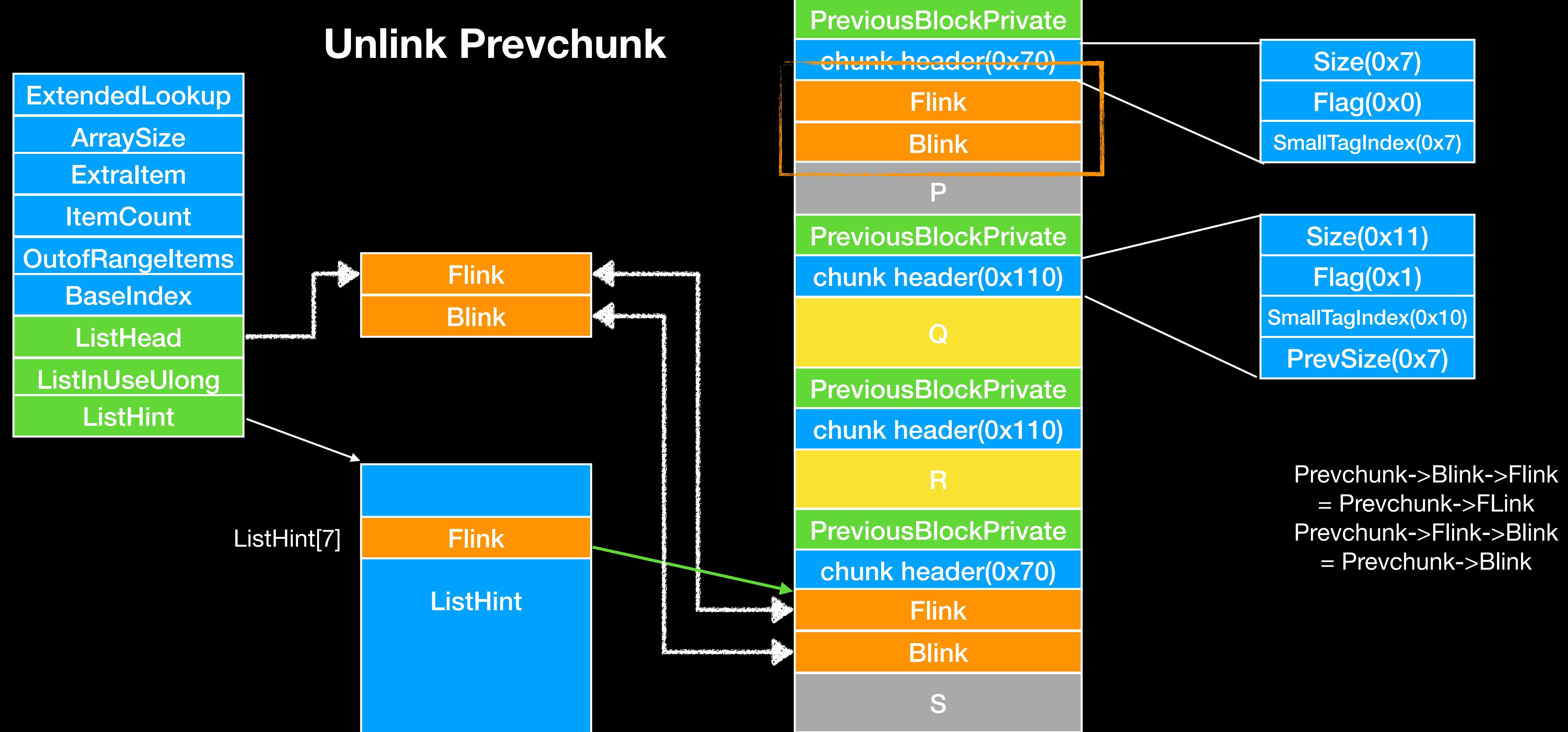
Windows Memory Allocator

merge chunk



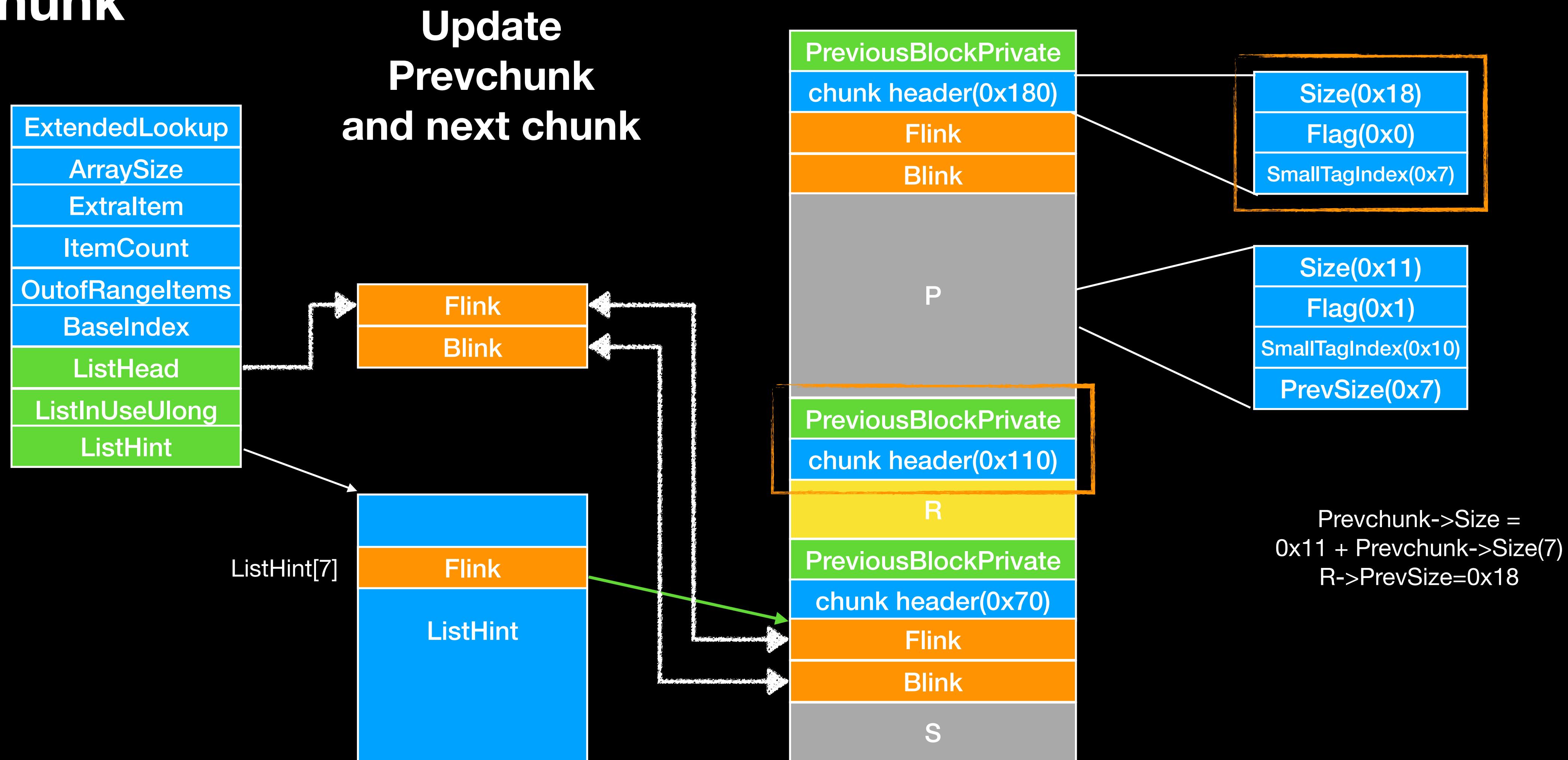
Windows Memory Allocator

merge chunk



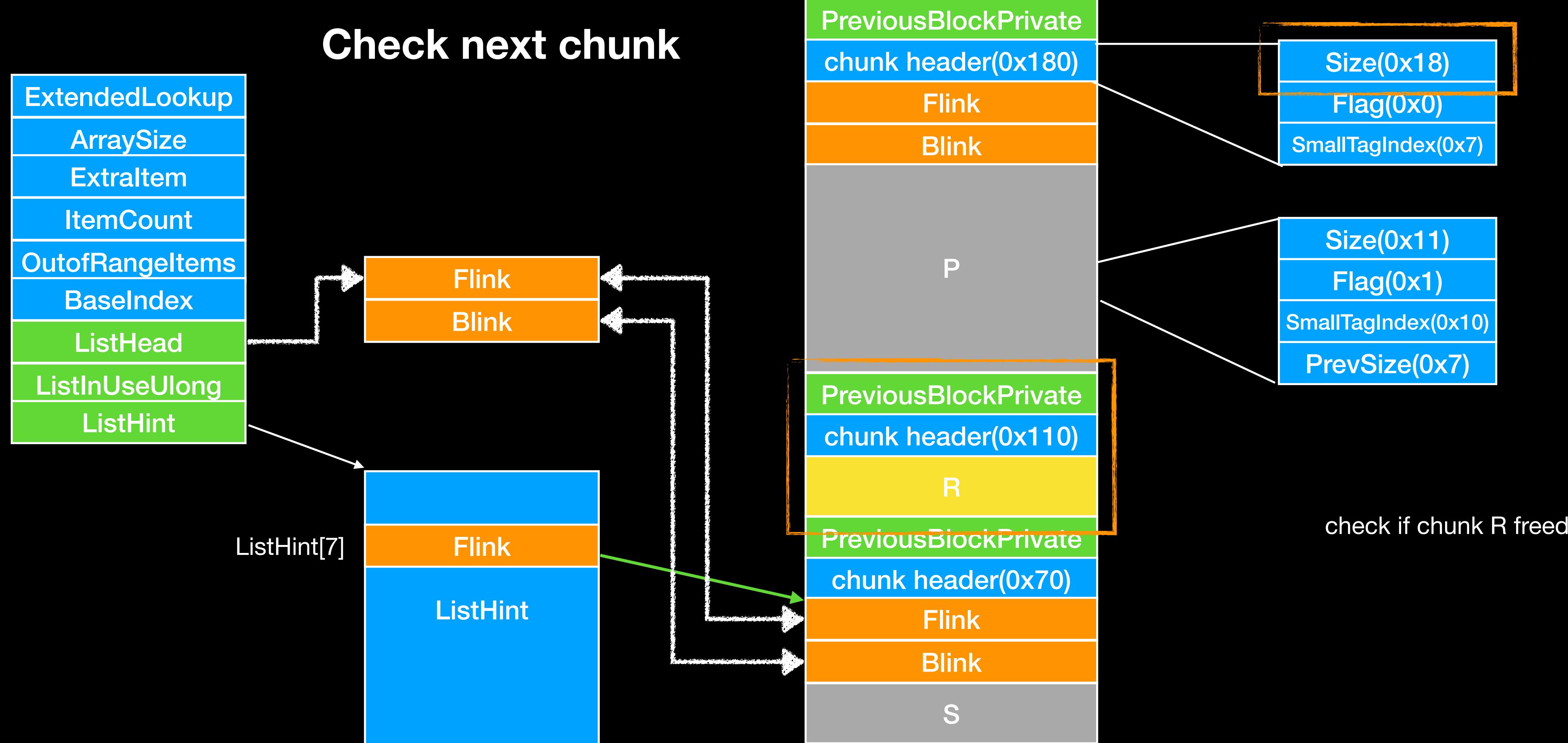
Windows Memory Allocator

merge chunk



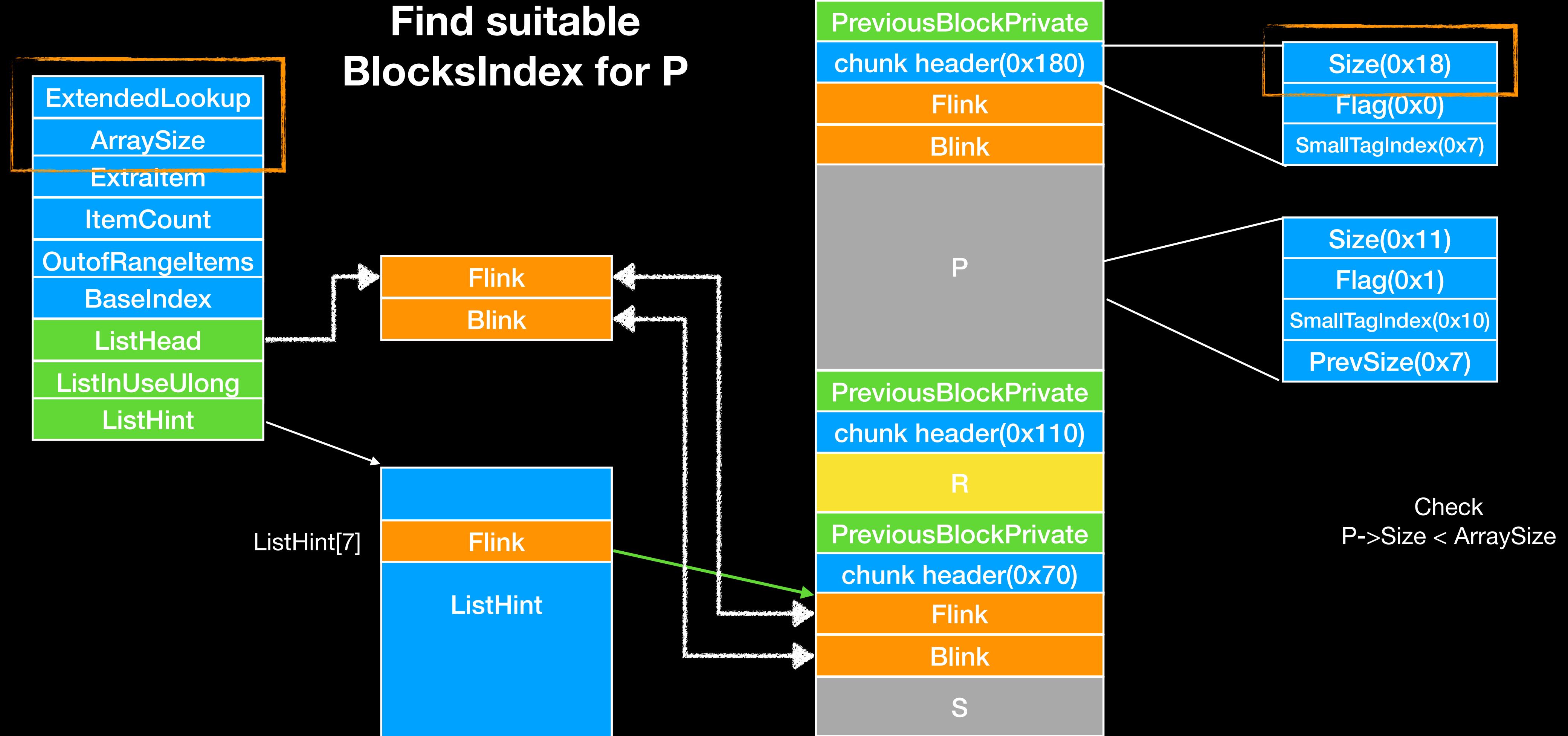
Windows Memory Allocator

merge chunk



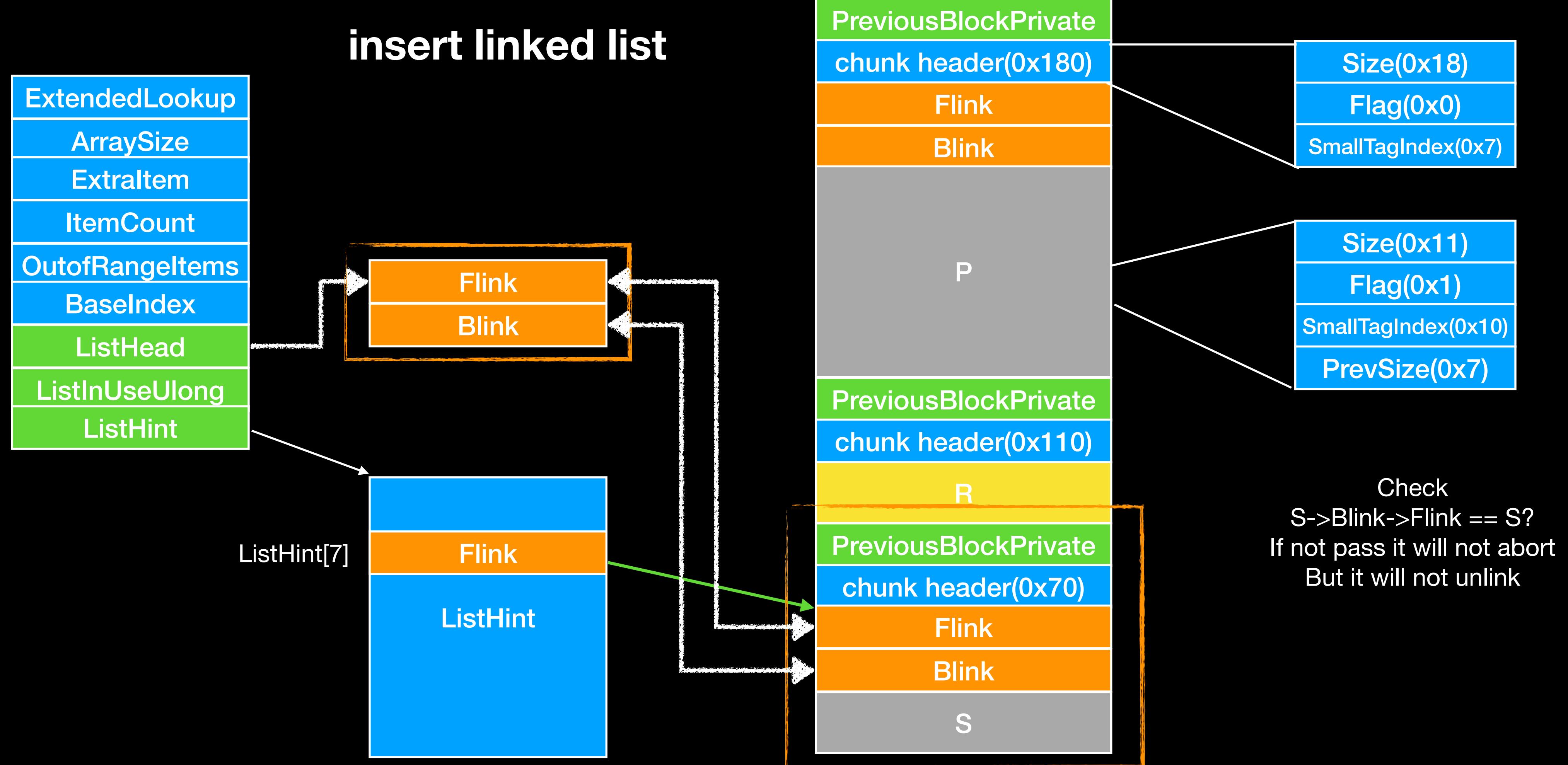
Windows Memory Allocator

merge chunk



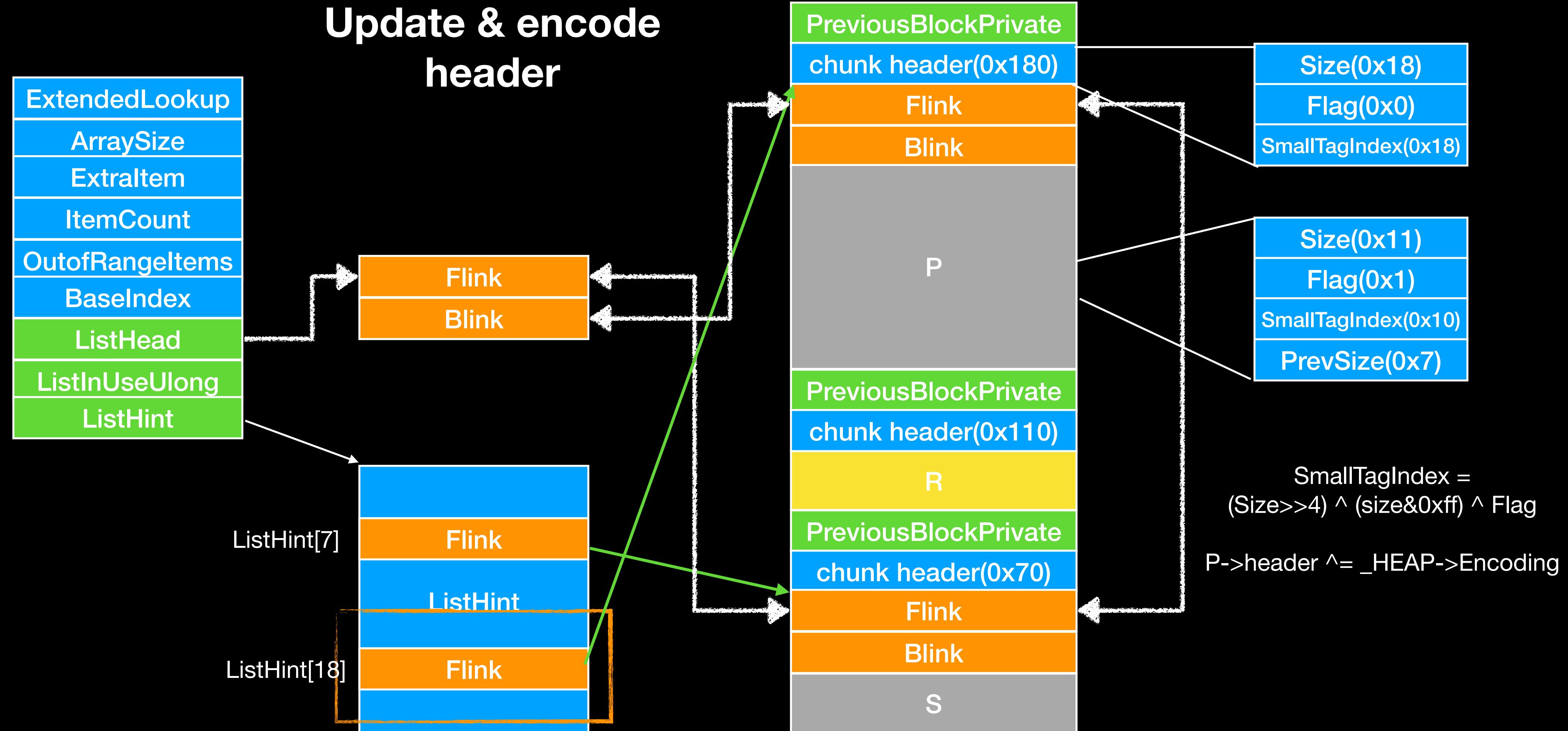
Windows Memory Allocator

merge chunk



Windows Memory Allocator

merge chunk

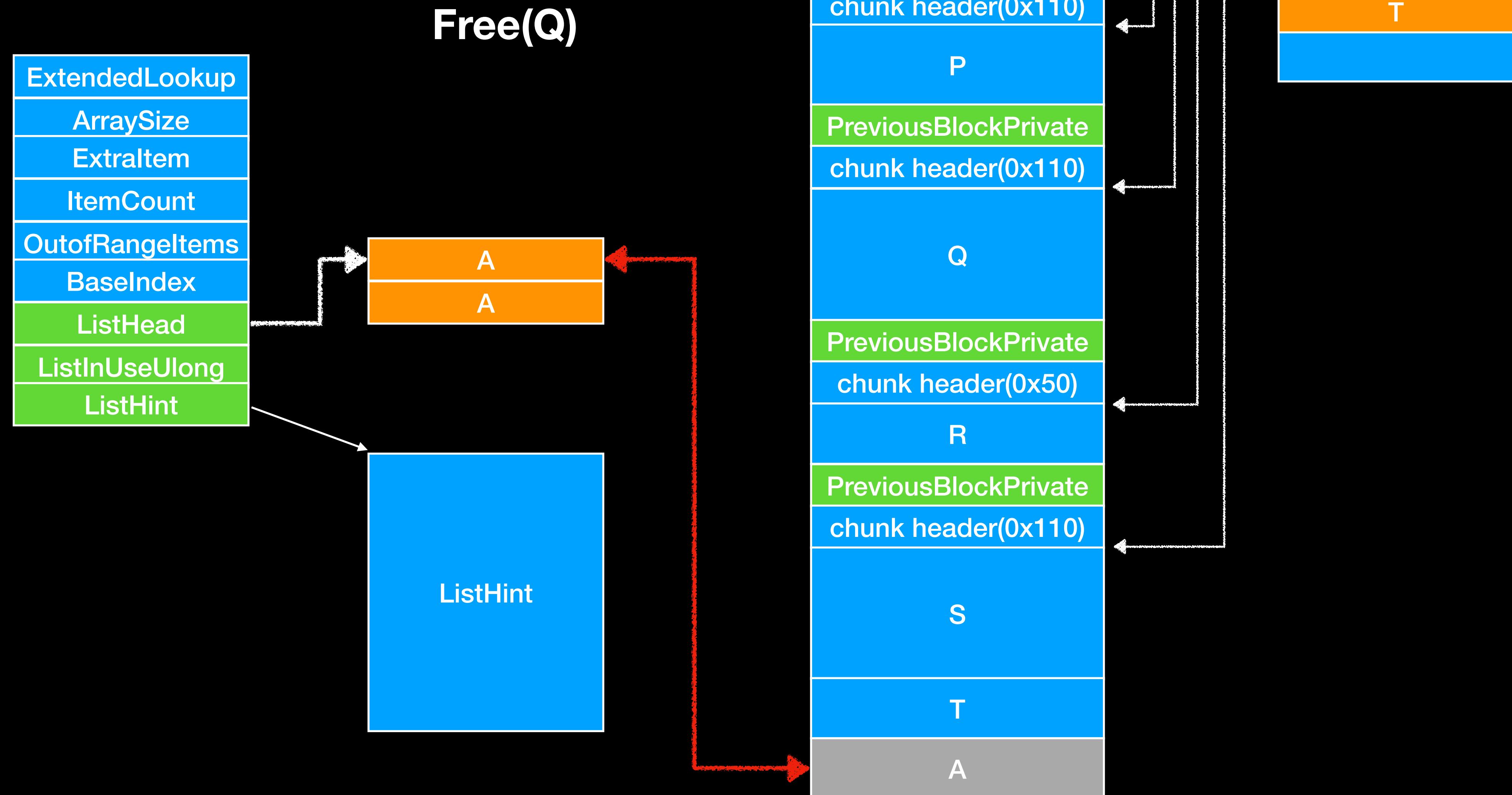


BackEnd Exploitation

- Unlink
 - like unlink in linux
 - Pass decode and checksum
 - Bypass double linked list check

BackEnd Exploitation

UAF

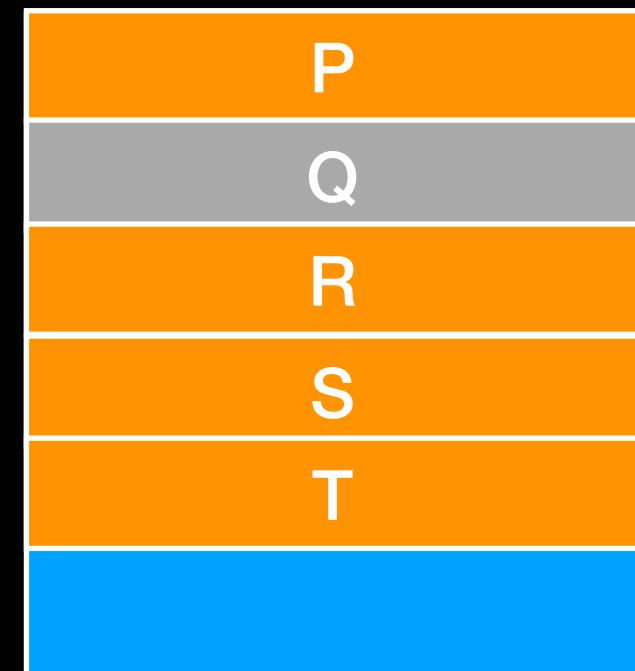
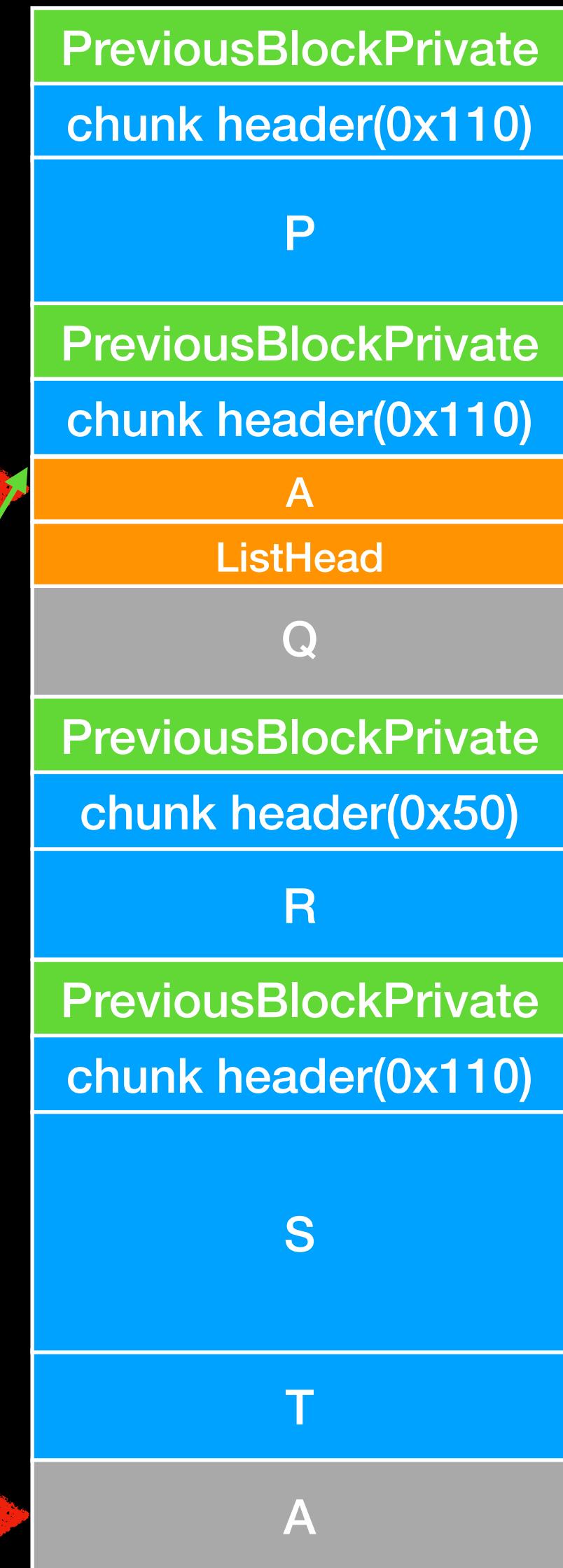
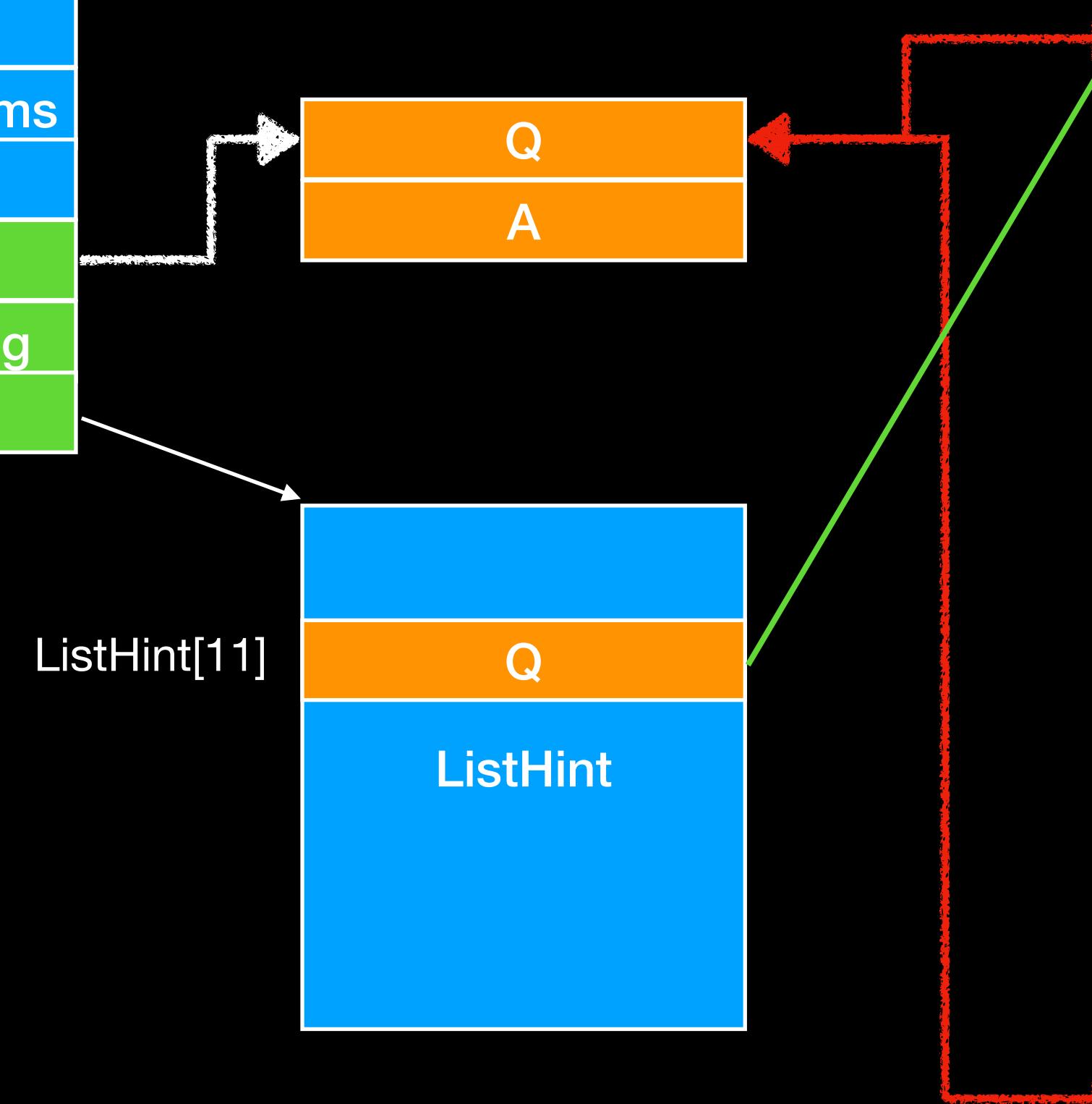


BackEnd Exploitation

UAF

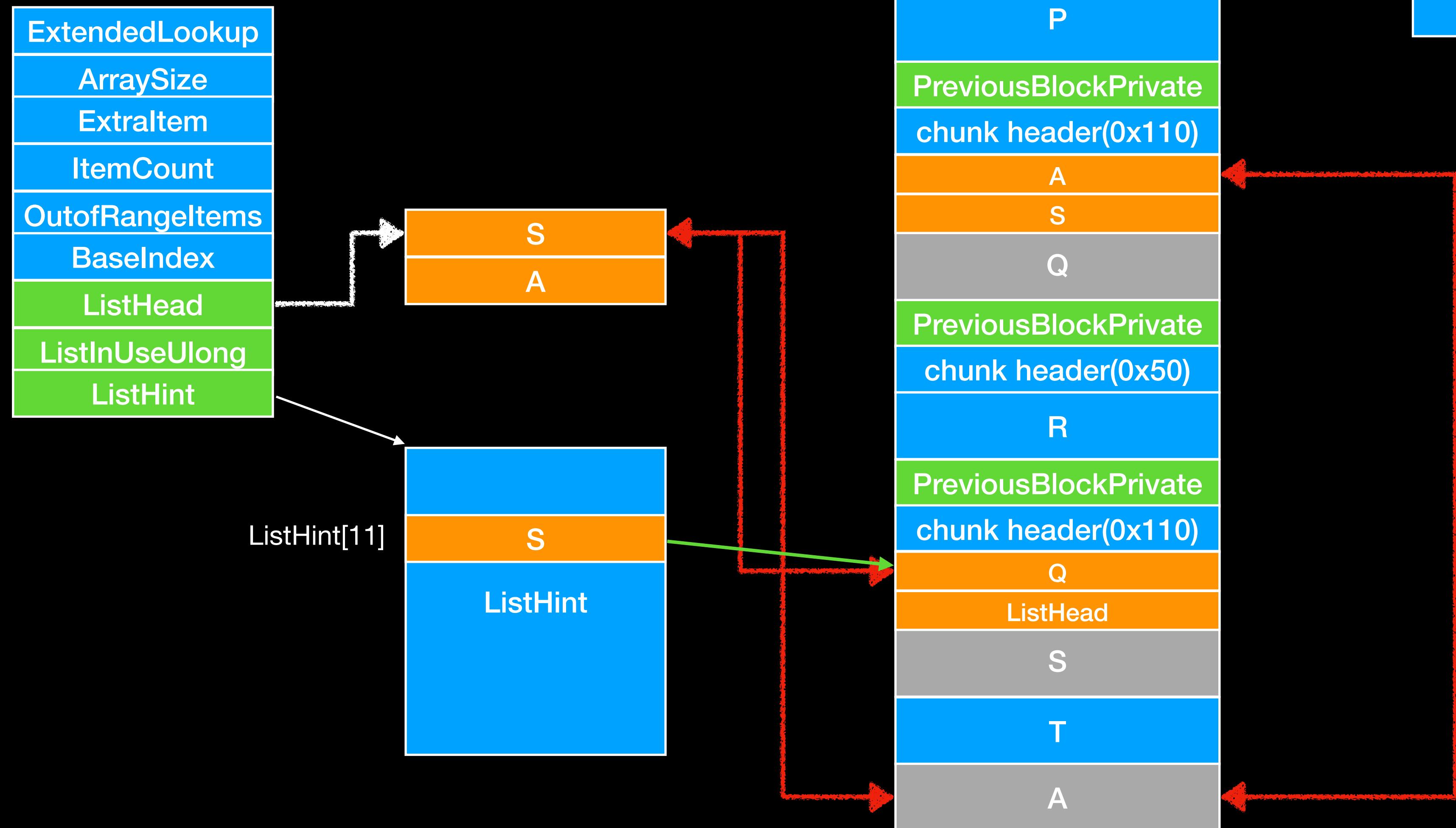
Free(S)
Update ListHint

ExtendedLookup
ArraySize
ExtralItem
ItemCount
OutofRangeItems
BaseIndex
ListHead
ListInUseUlong
ListHint



BackEnd Exploitation

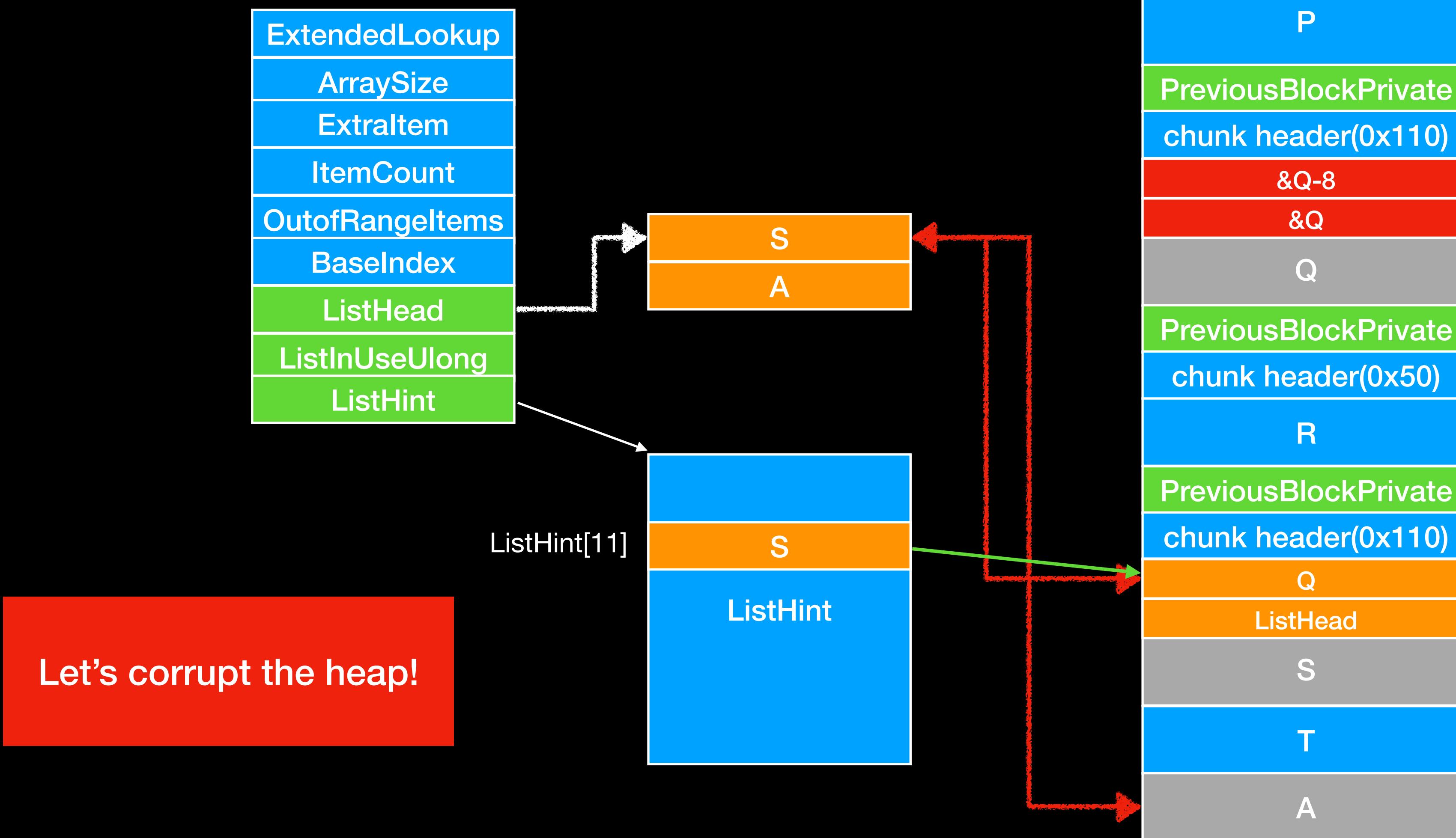
UAF



Data

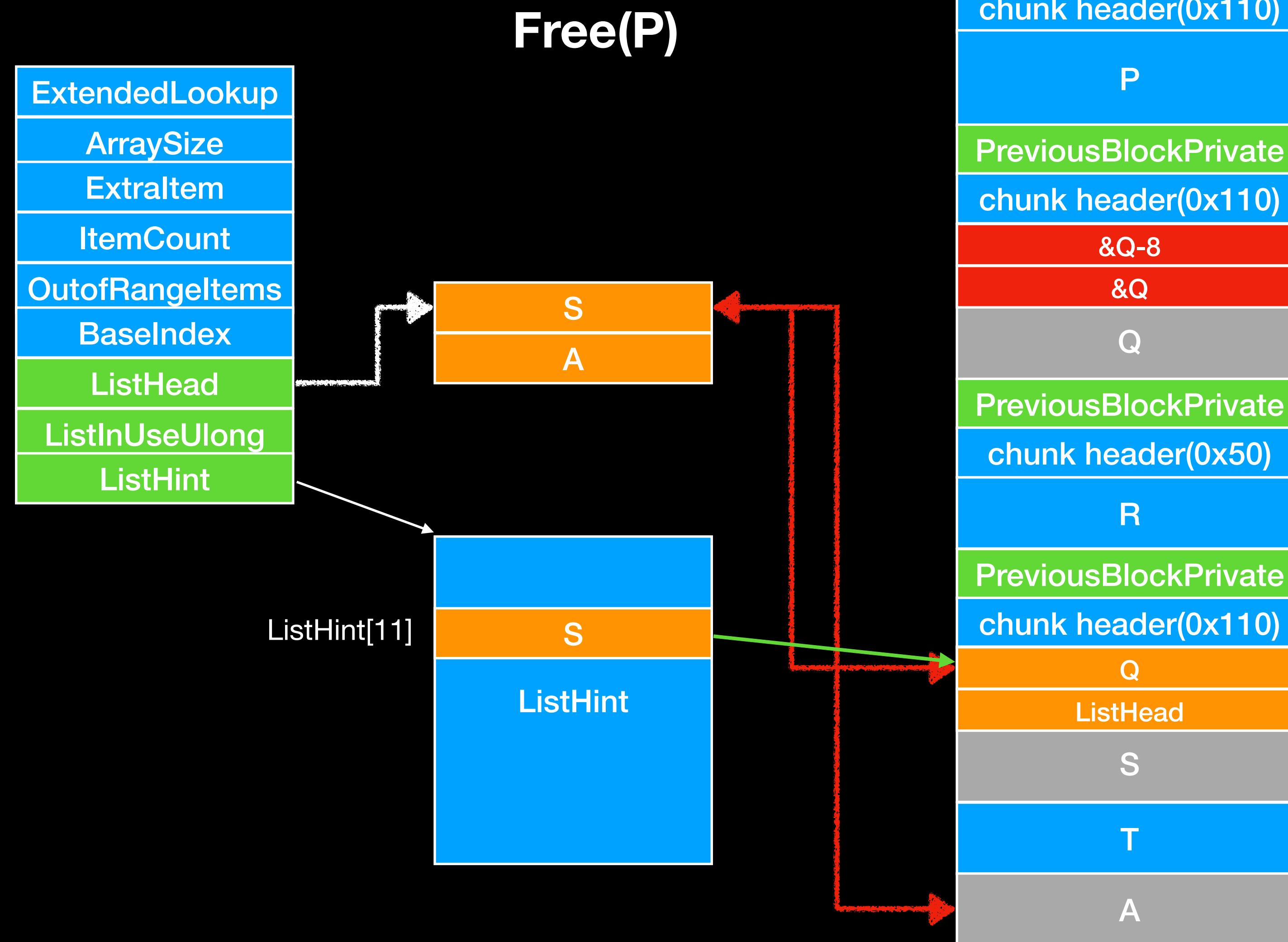
BackEnd Exploitation

UAF



BackEnd Exploitation

UAF



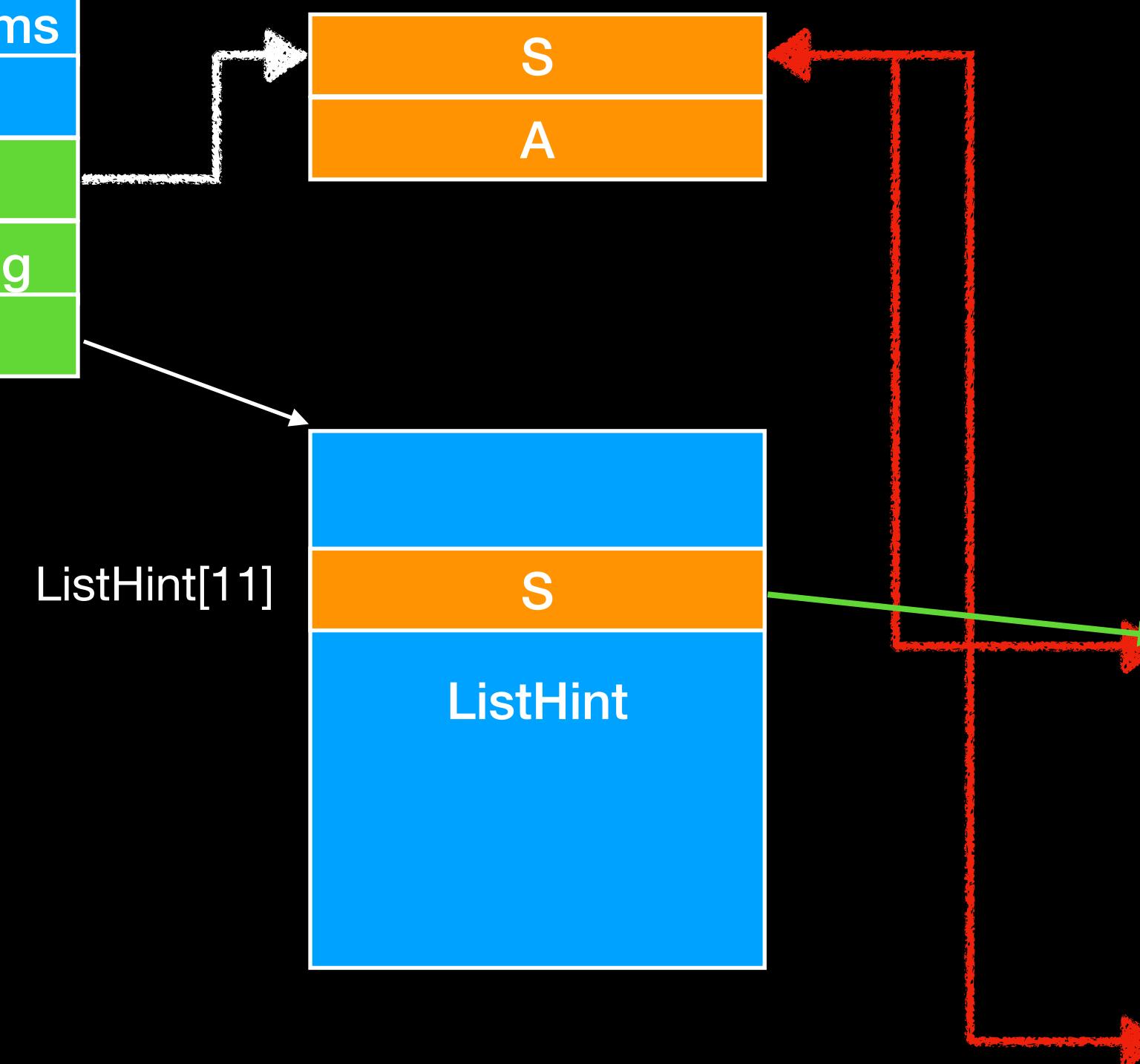
Data

BackEnd Exploitation

UAF

Check next chunk
is free

ExtendedLookup
ArraySize
ExtralItem
ItemCount
OutofRangeItems
BaseIndex
ListHead
ListInUseUlong
ListHint

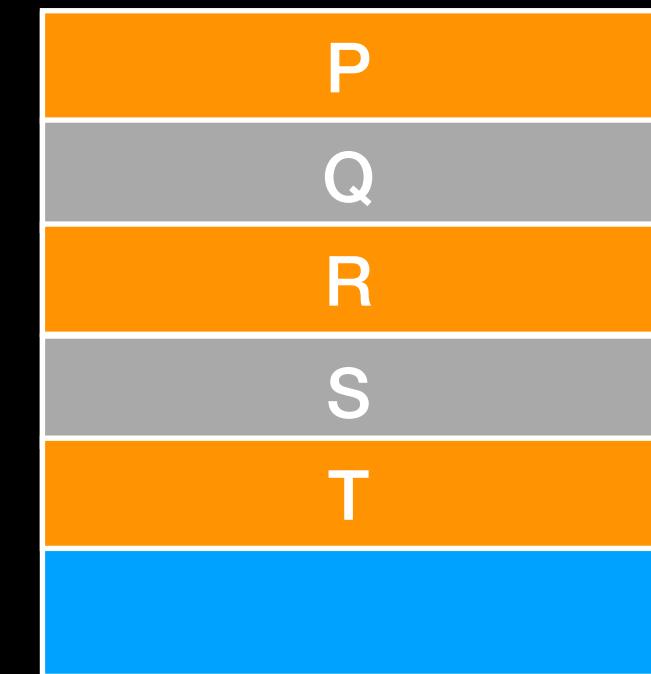
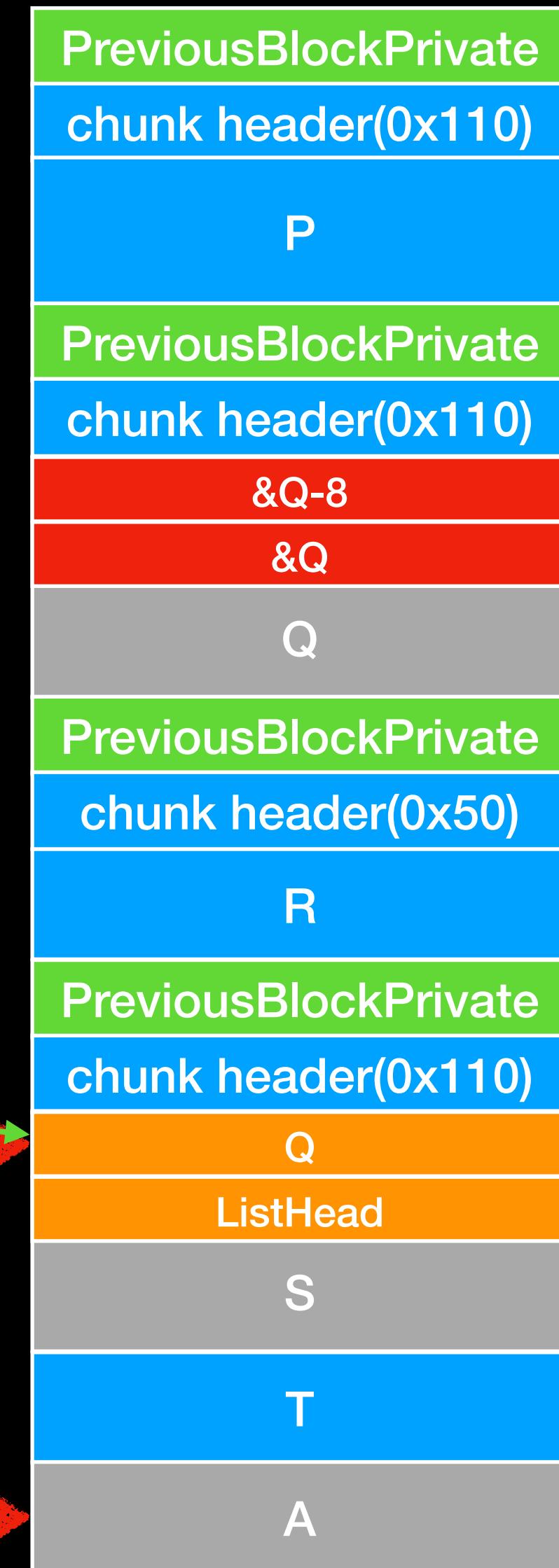
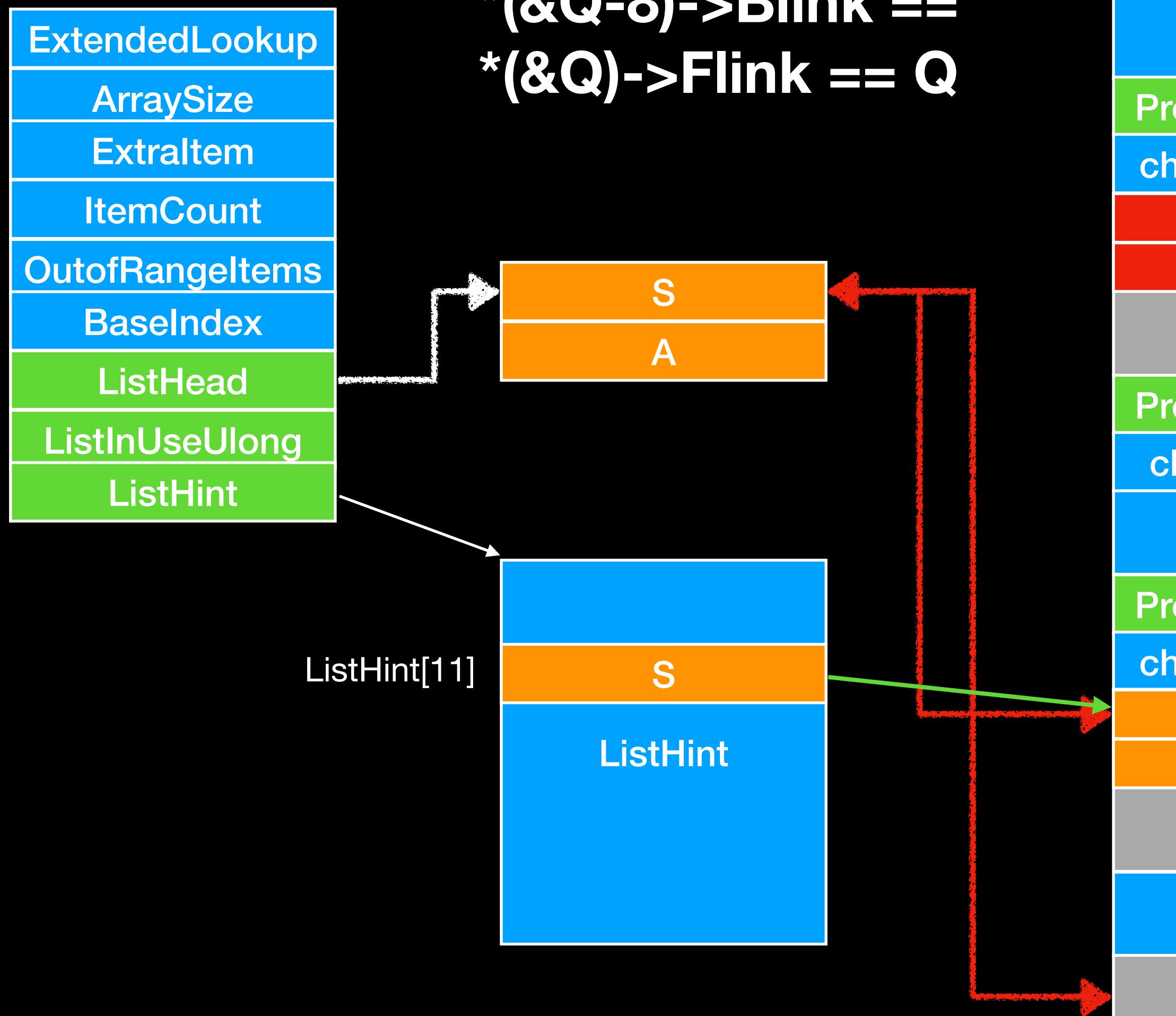


PreviousBlockPrivate
chunk header(0x110)
P
PreviousBlockPrivate
chunk header(0x110)
&Q-8
&Q
Q
PreviousBlockPrivate
chunk header(0x50)
R
PreviousBlockPrivate
chunk header(0x110)
Q
ListHead
S
T
A

BackEnd Exploitation

UAF

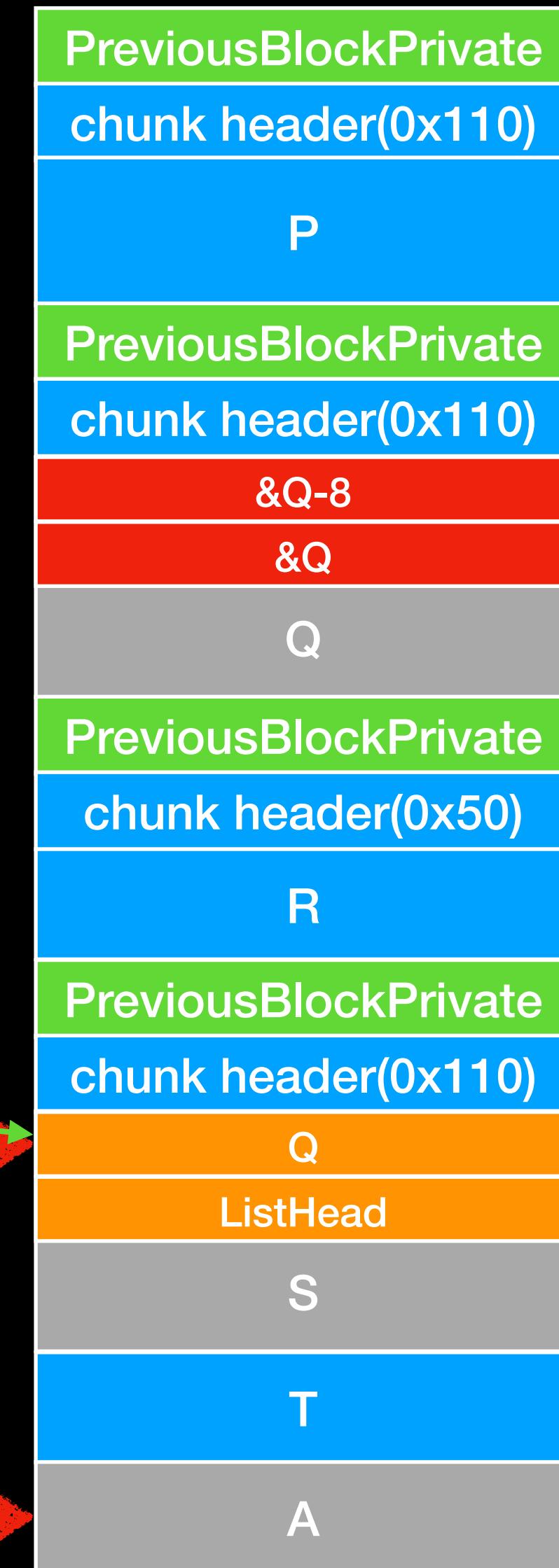
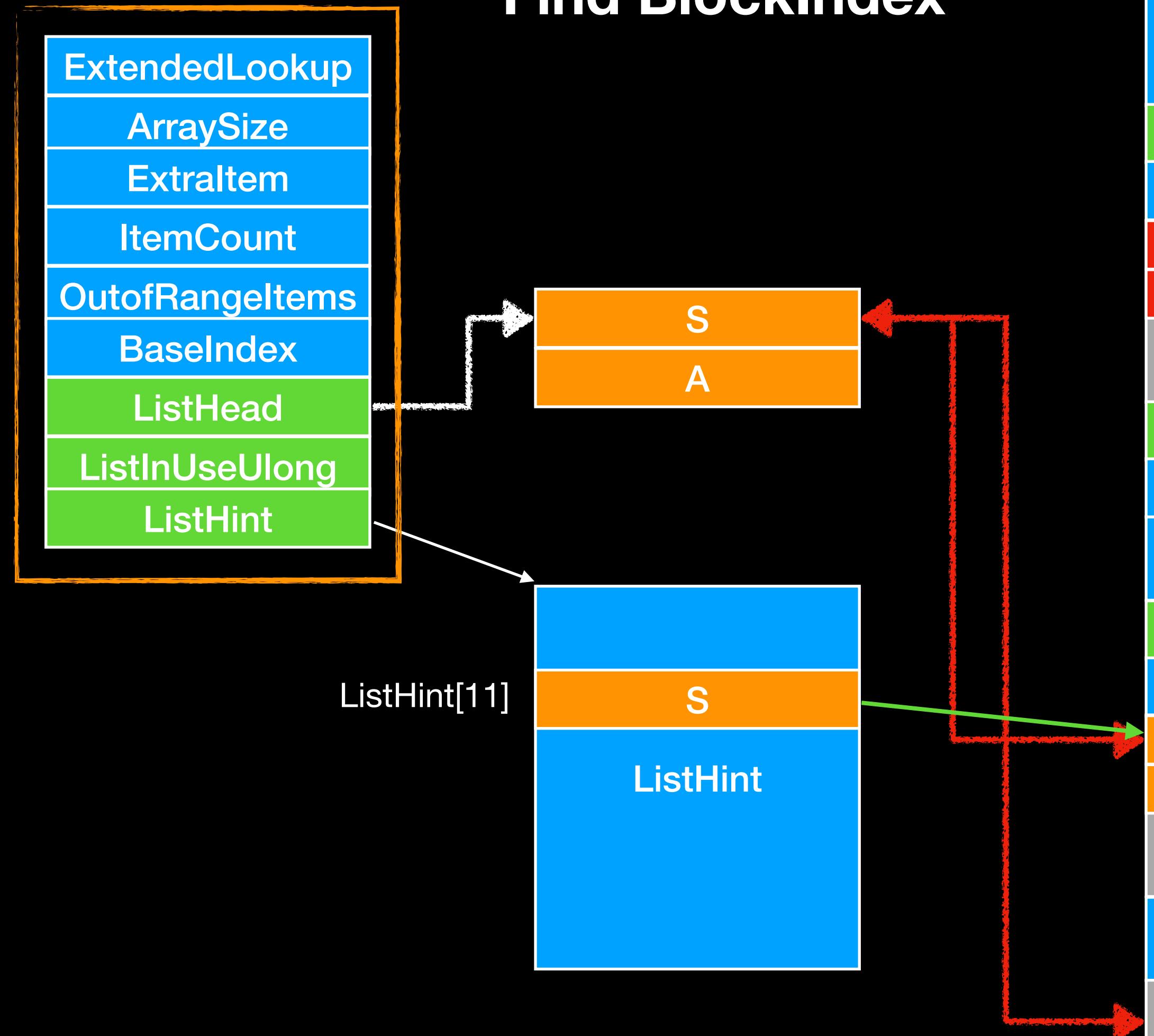
Decode Q
And bypass check
 $\ast(\&Q-8)->\text{Blink} ==$
 $\ast(\&Q)->\text{Flink} == Q$



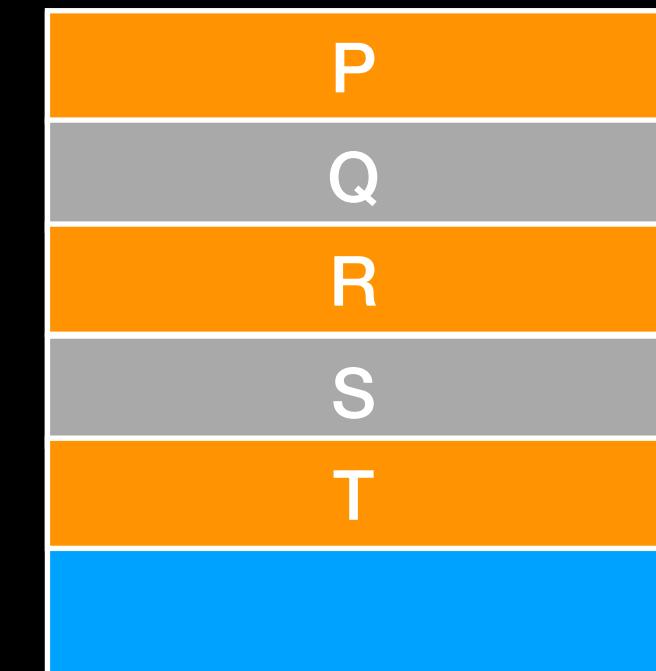
BackEnd Exploitation

UAF

Find BlockIndex



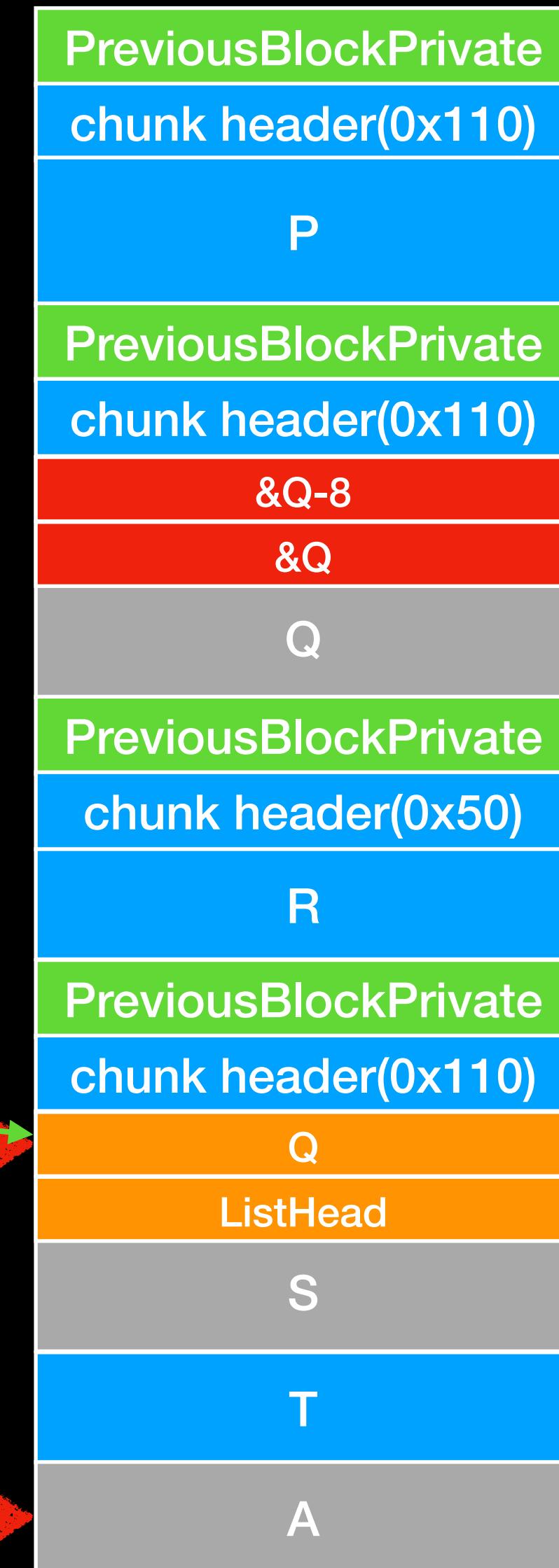
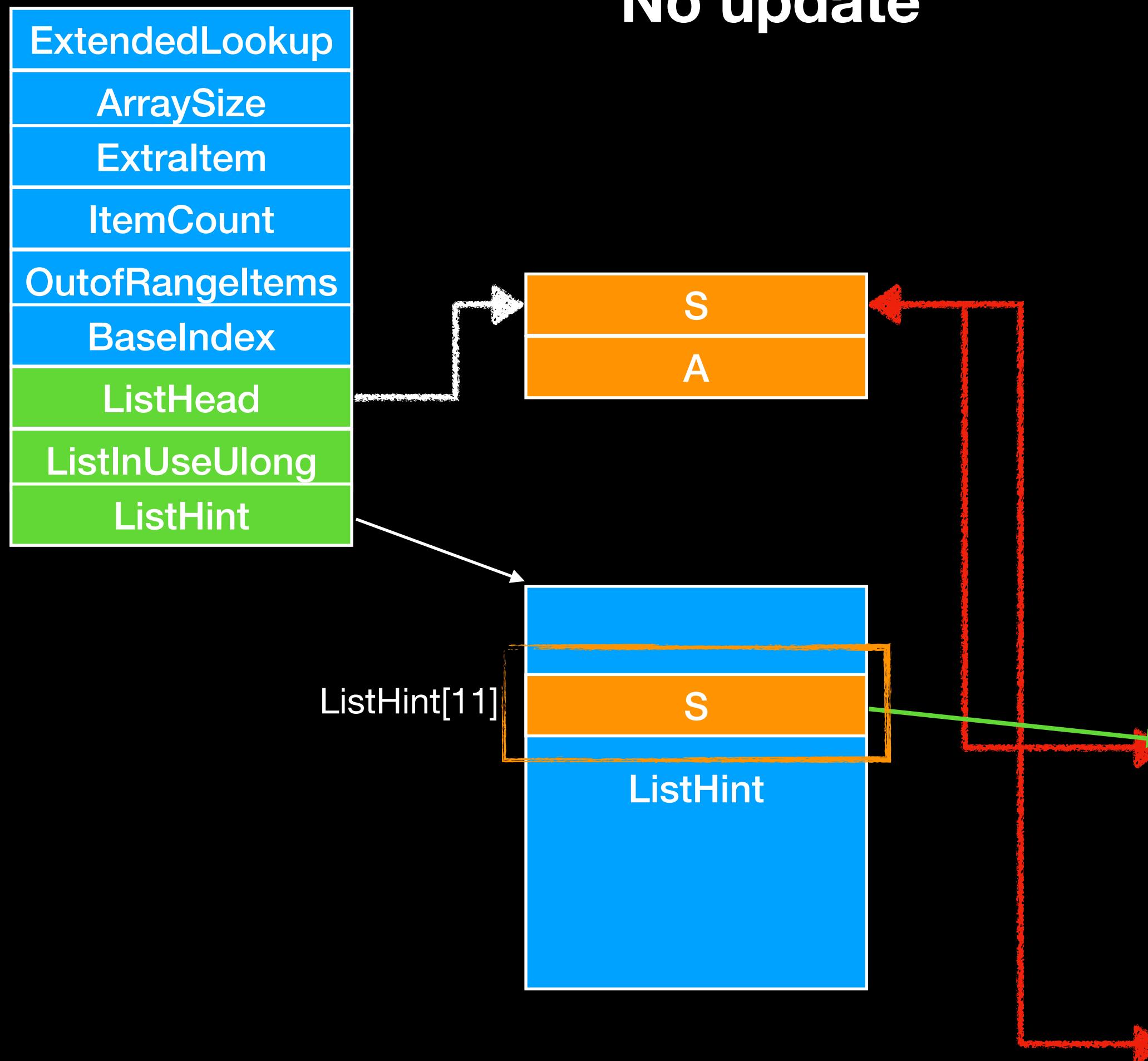
Data



BackEnd Exploitation

UAF

Check ListHint[0x11]
No update

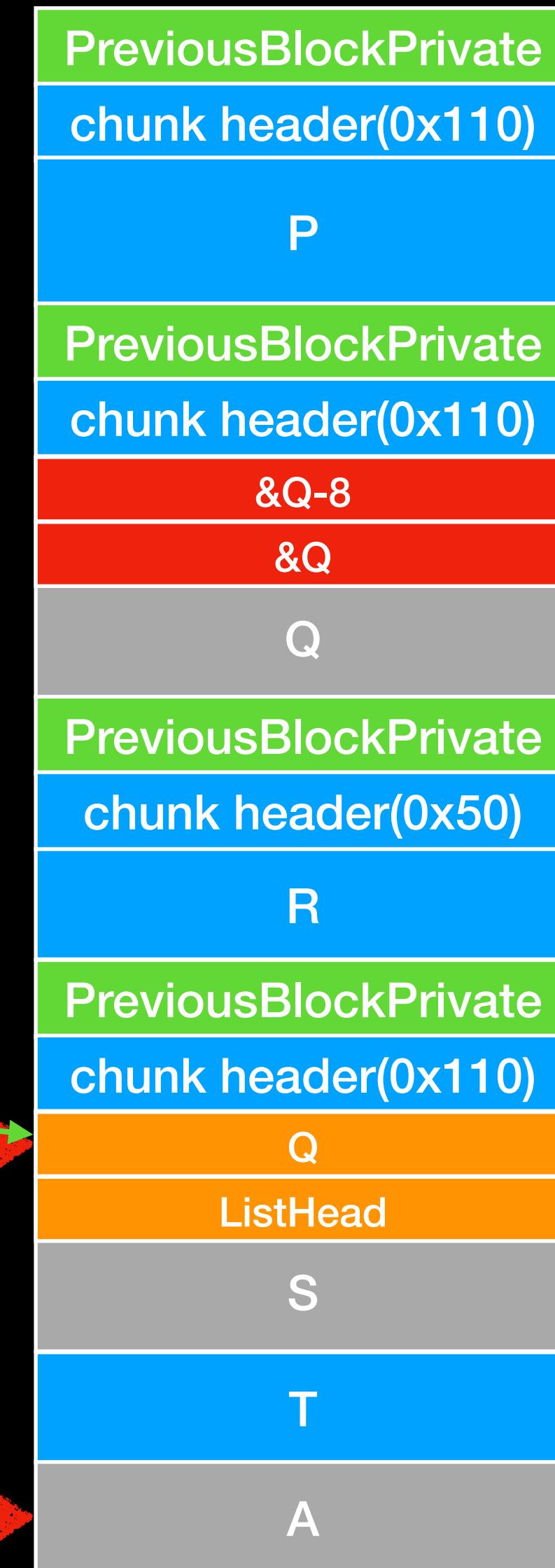
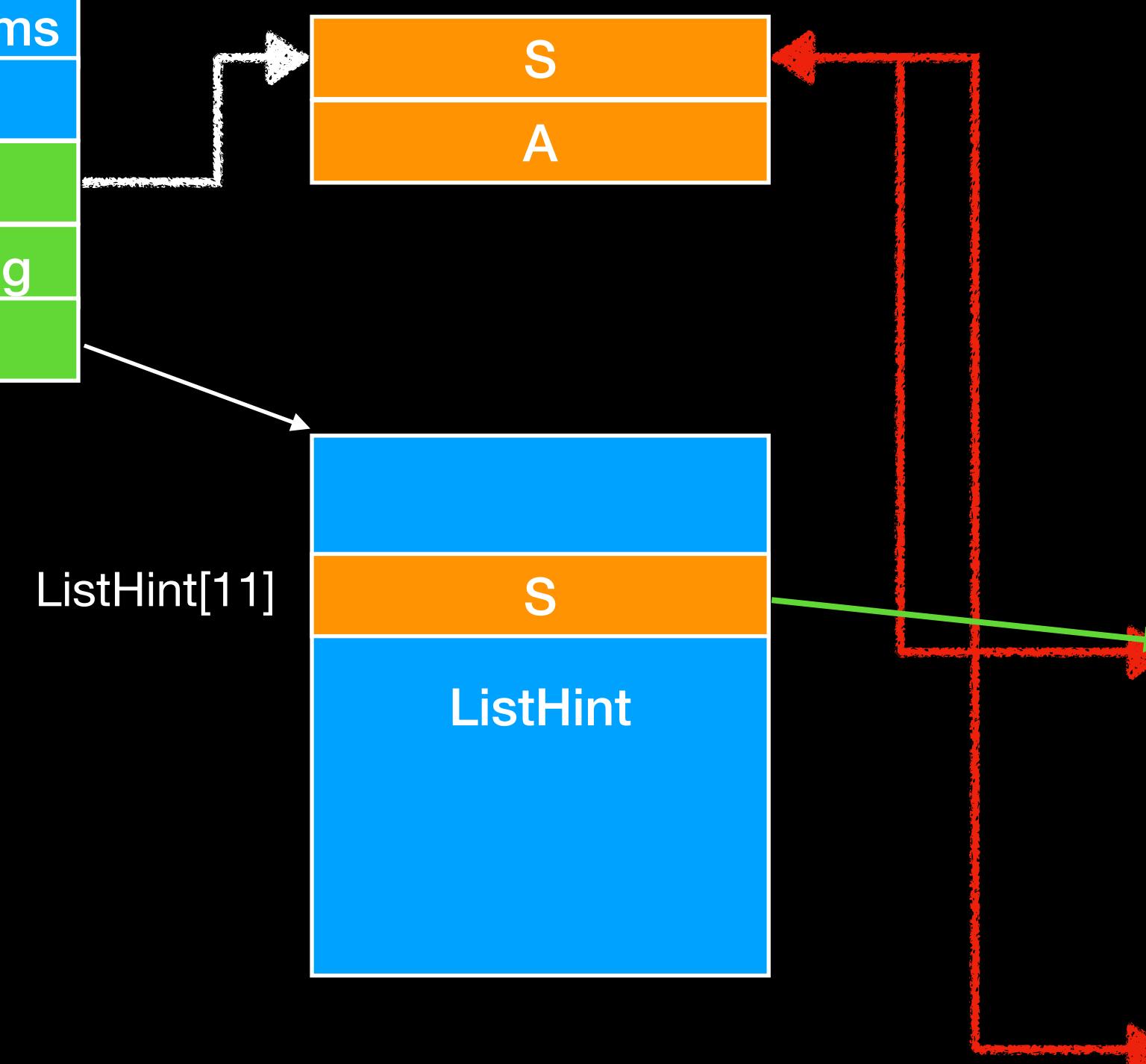


BackEnd Exploitation

UAF

Unlink Attack!

ExtendedLookup
ArraySize
ExtralItem
ItemCount
OutofRangeItems
BaseIndex
ListHead
ListInUseUlong
ListHint



Q->Blink->Flink = Q->Flink
Q = &Q-8

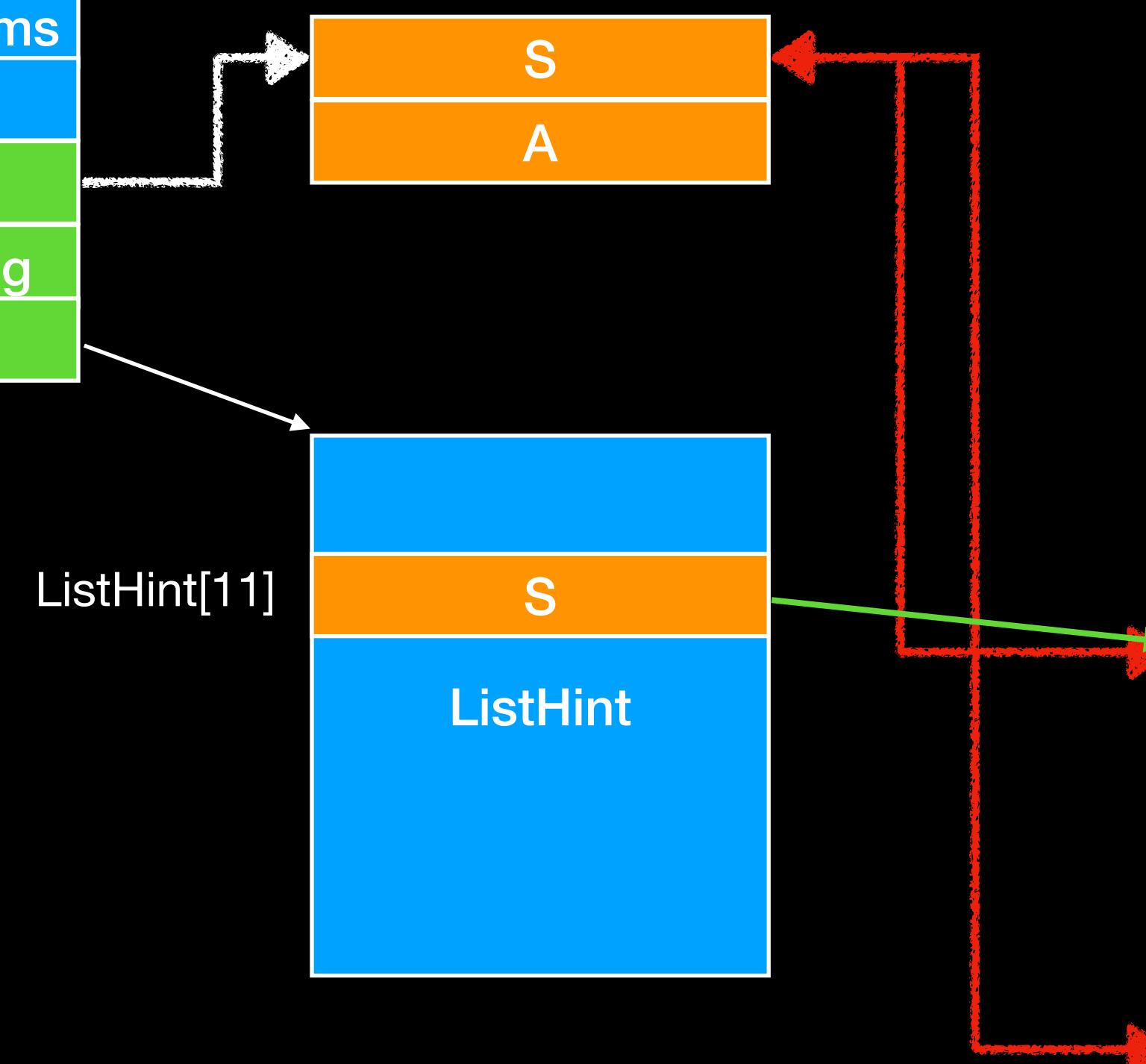
Data

BackEnd Exploitation

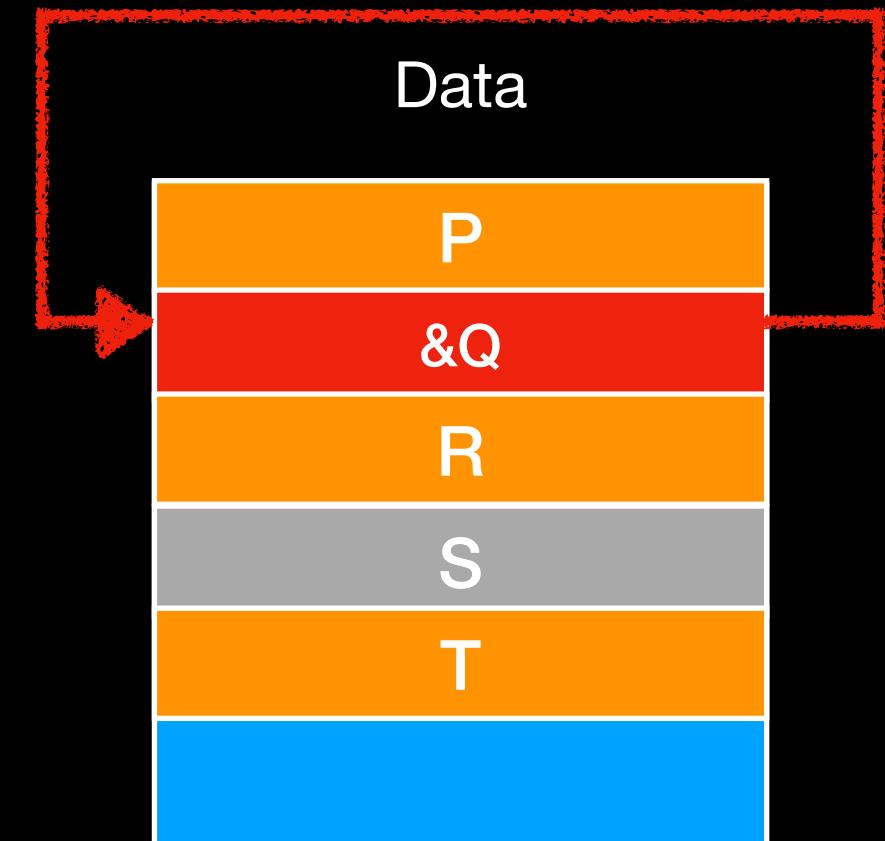
UAF

Unlink Attack!

ExtendedLookup
ArraySize
ExtralItem
ItemCount
OutofRangeItems
BaseIndex
ListHead
ListInUseUlong
ListHint



PreviousBlockPrivate
chunk header(0x110)
P
PreviousBlockPrivate
chunk header(0x110)
&Q-8
&Q
Q
PreviousBlockPrivate
chunk header(0x50)
R
PreviousBlockPrivate
chunk header(0x110)
Q
ListHead
S
T
A



Q->Flink->Blink = Q->Blink
Q = &Q

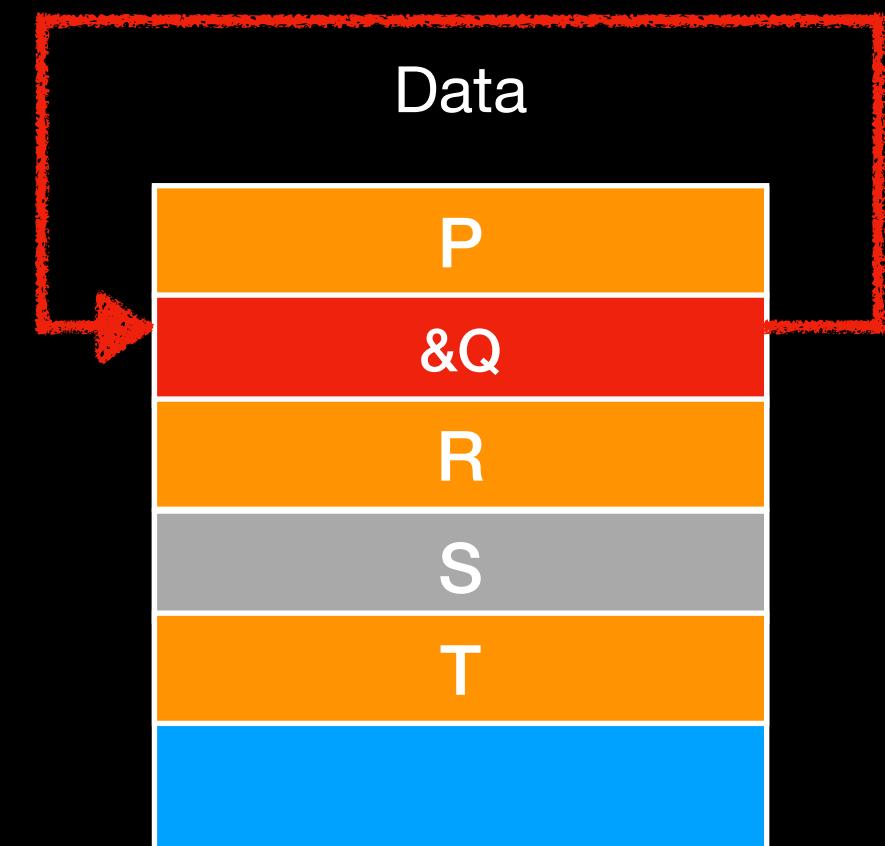
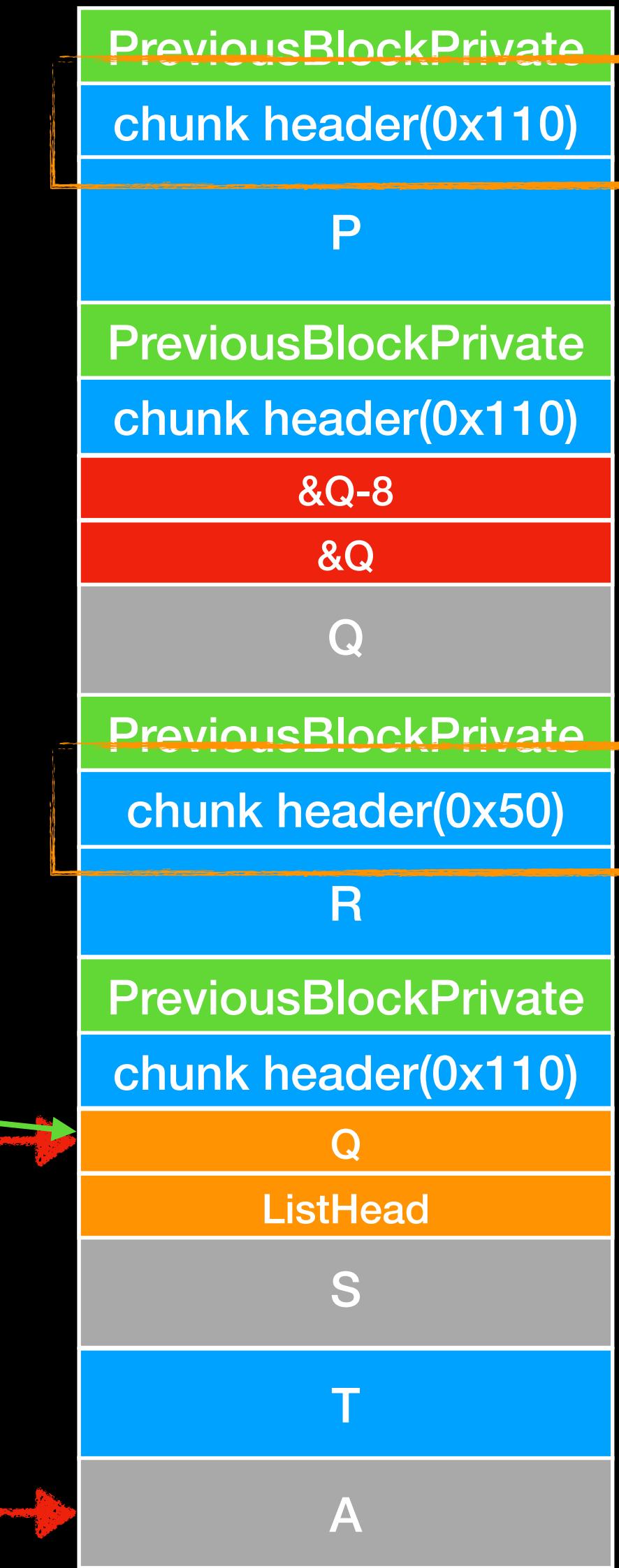
BackEnd Exploitation

UAF

Update chunk Size

ExtendedLookup
ArraySize
ExtralItem
ItemCount
OutofRangeItems
BaseIndex
ListHead
ListInUseUlong
ListHint

ListHint[11]



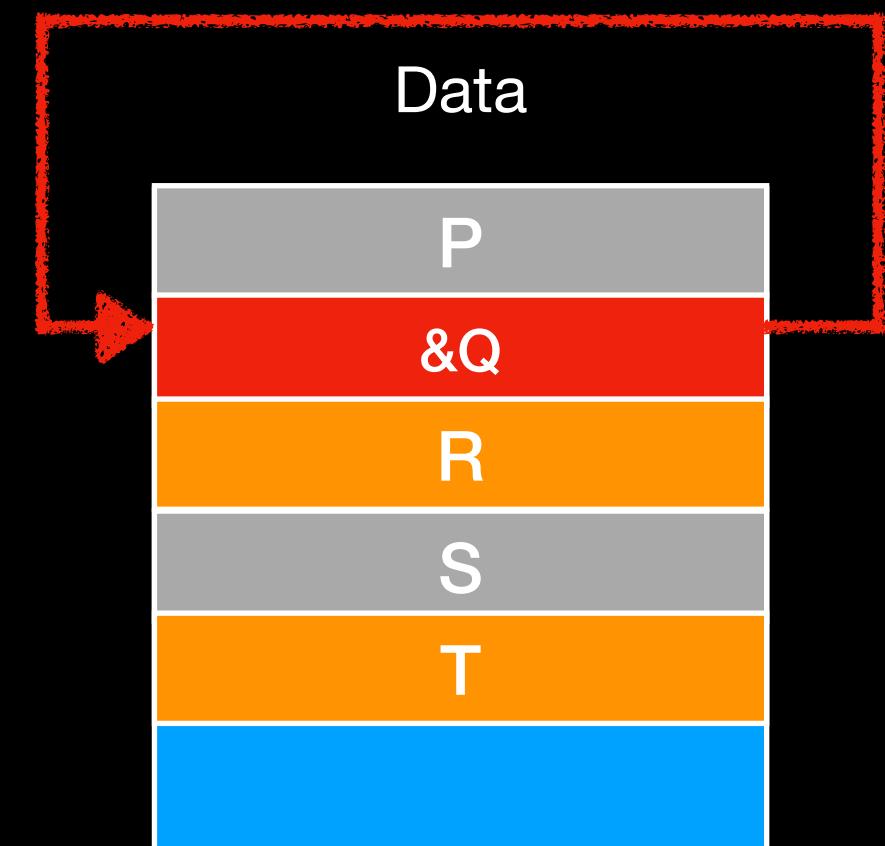
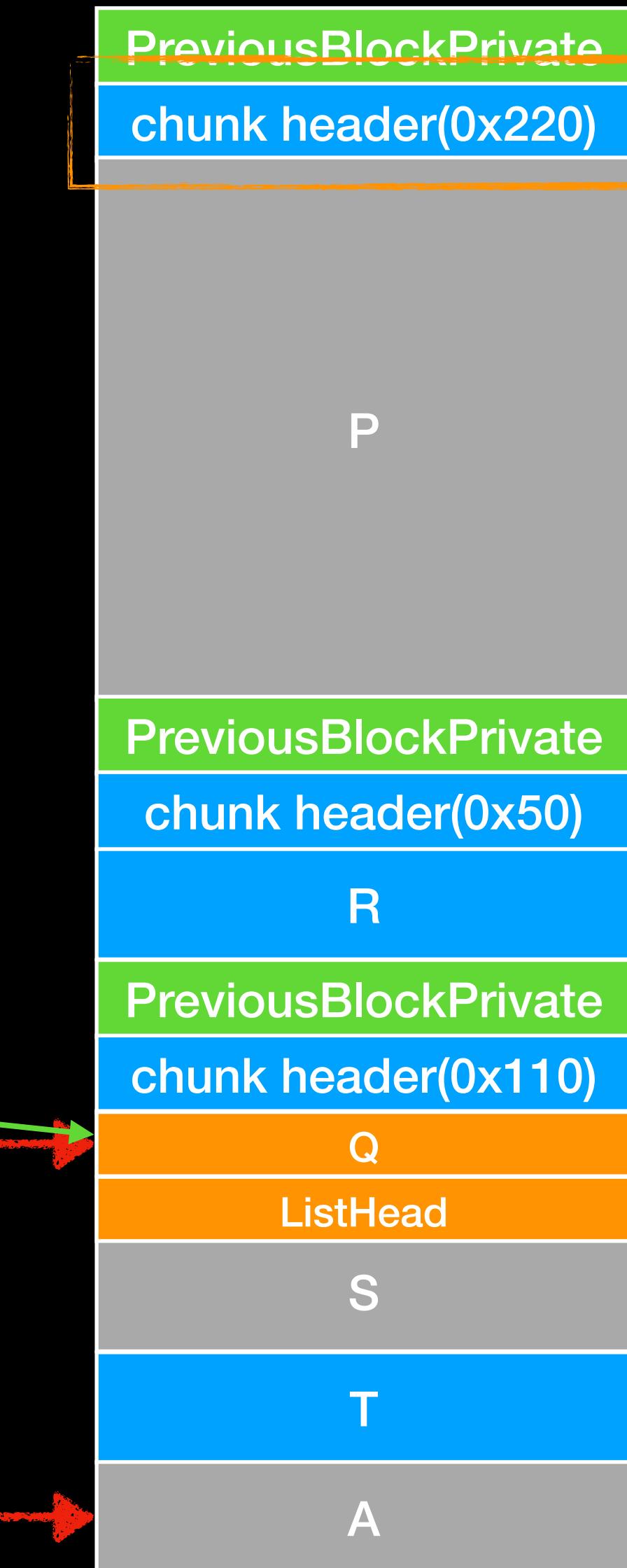
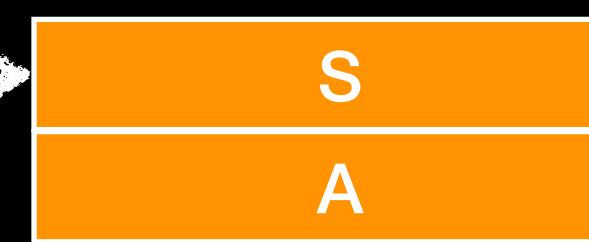
BackEnd Exploitation

UAF

Update chunk Size

ExtendedLookup
ArraySize
ExtralItem
ItemCount
OutofRangeItems
BaseIndex
ListHead
ListInUseUlong
ListHint

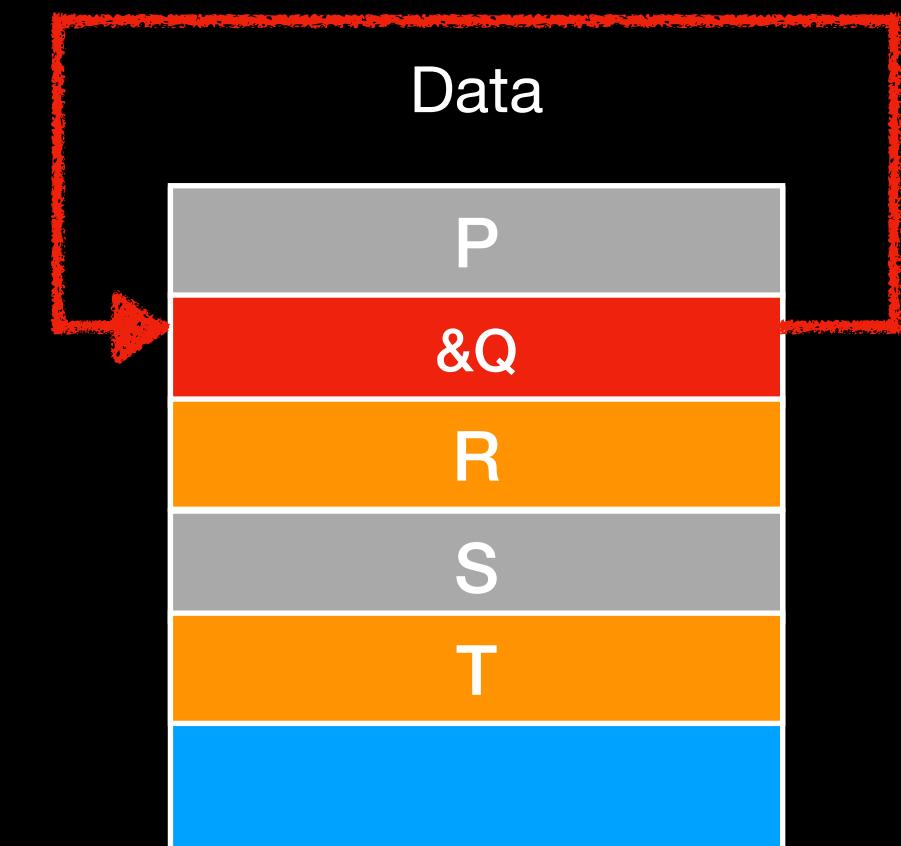
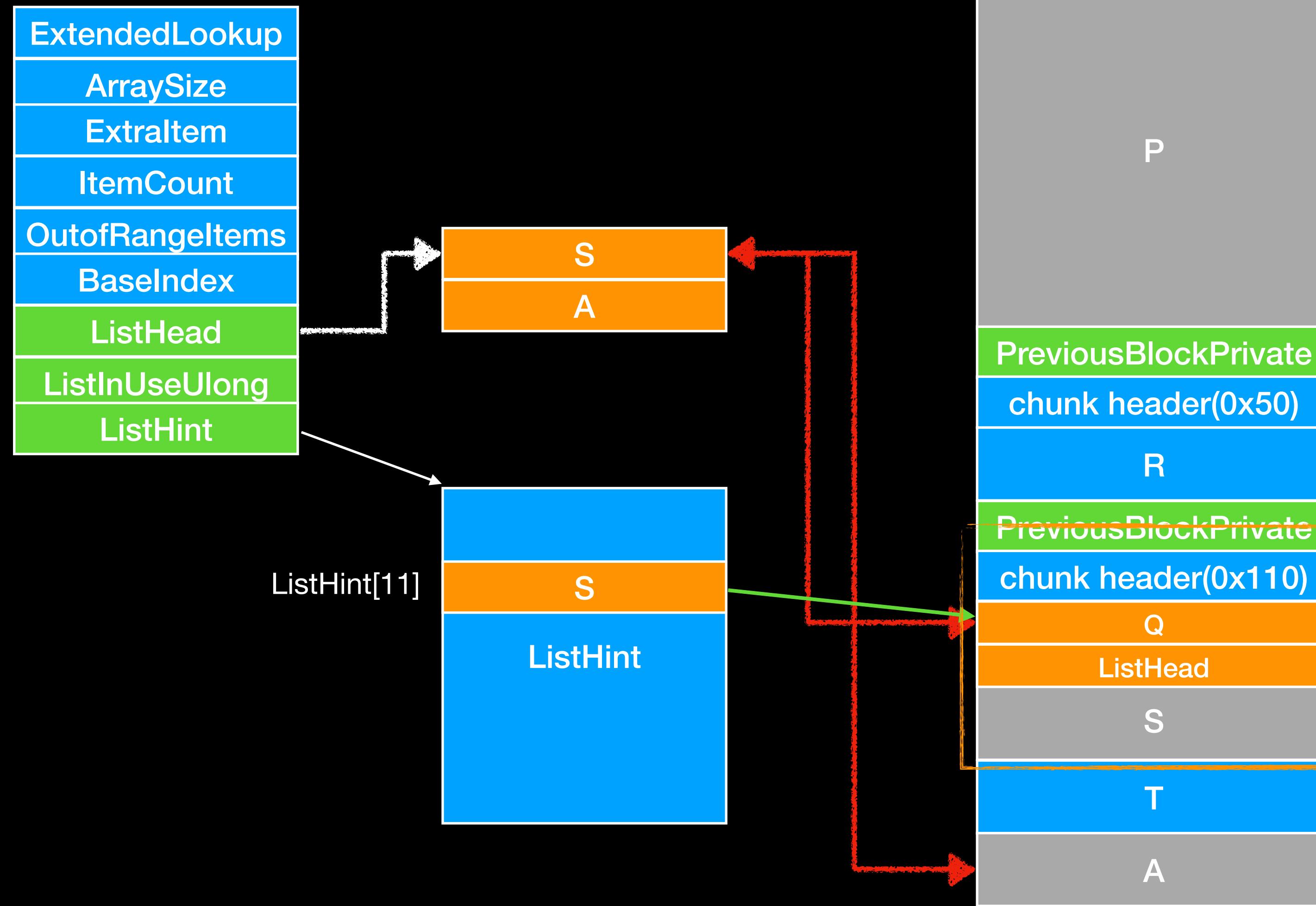
ListHint[11]



BackEnd Exploitation

UAF

Search insert point

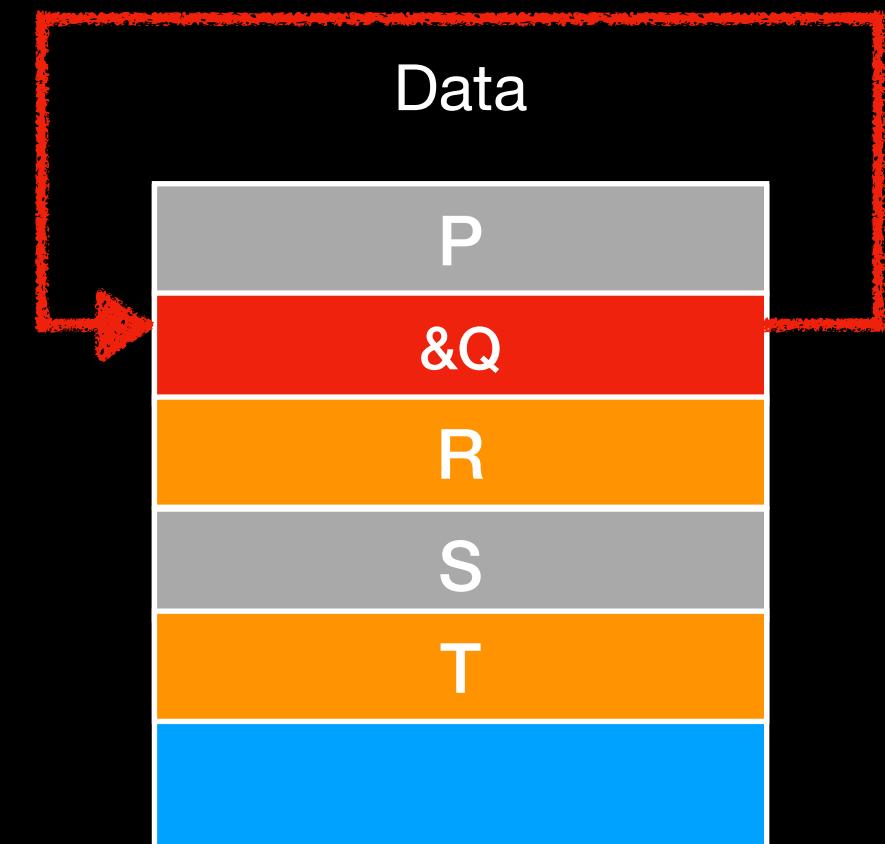
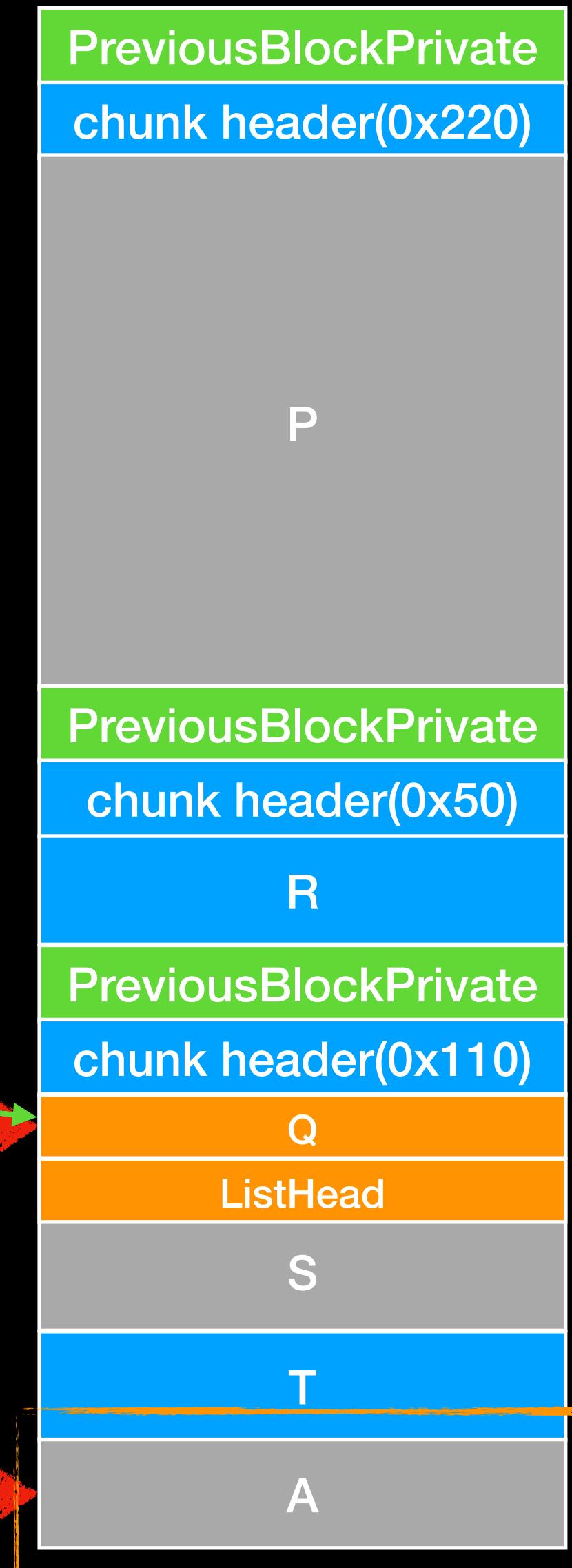
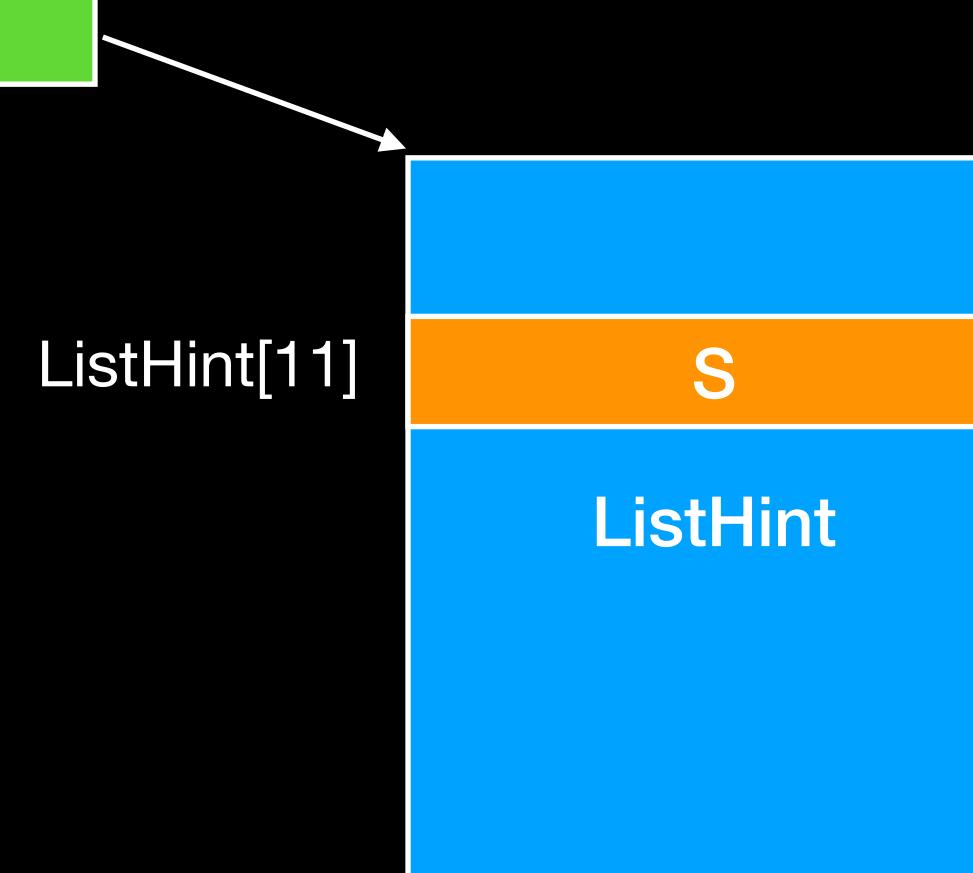


BackEnd Exploitation

UAF

Search insert point

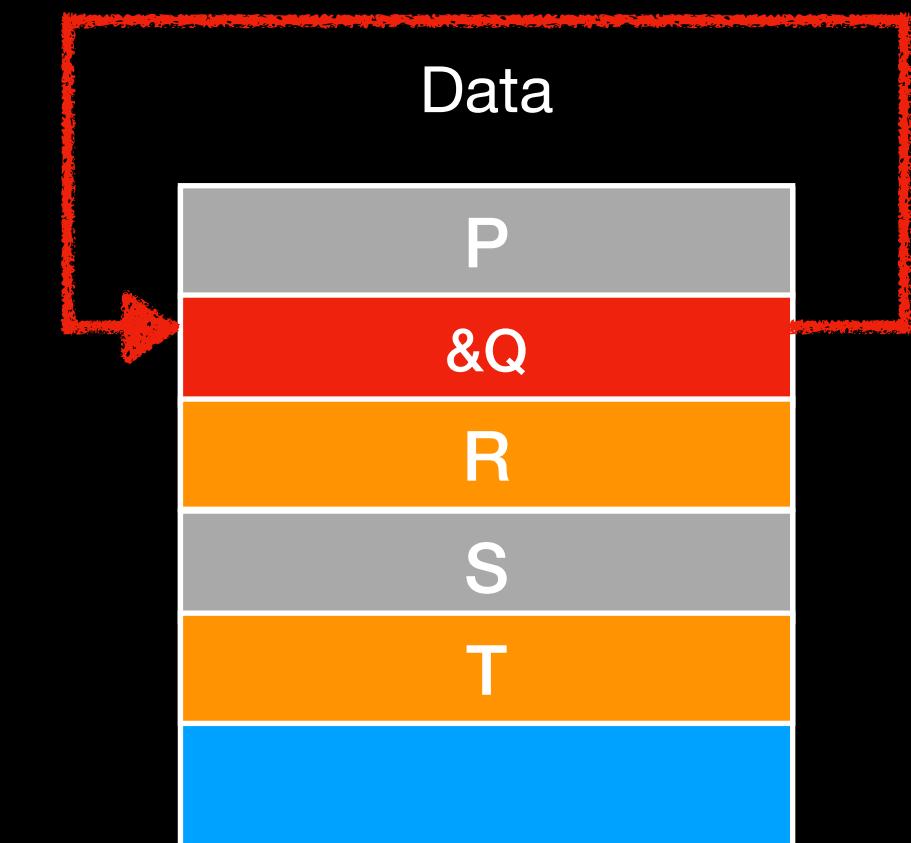
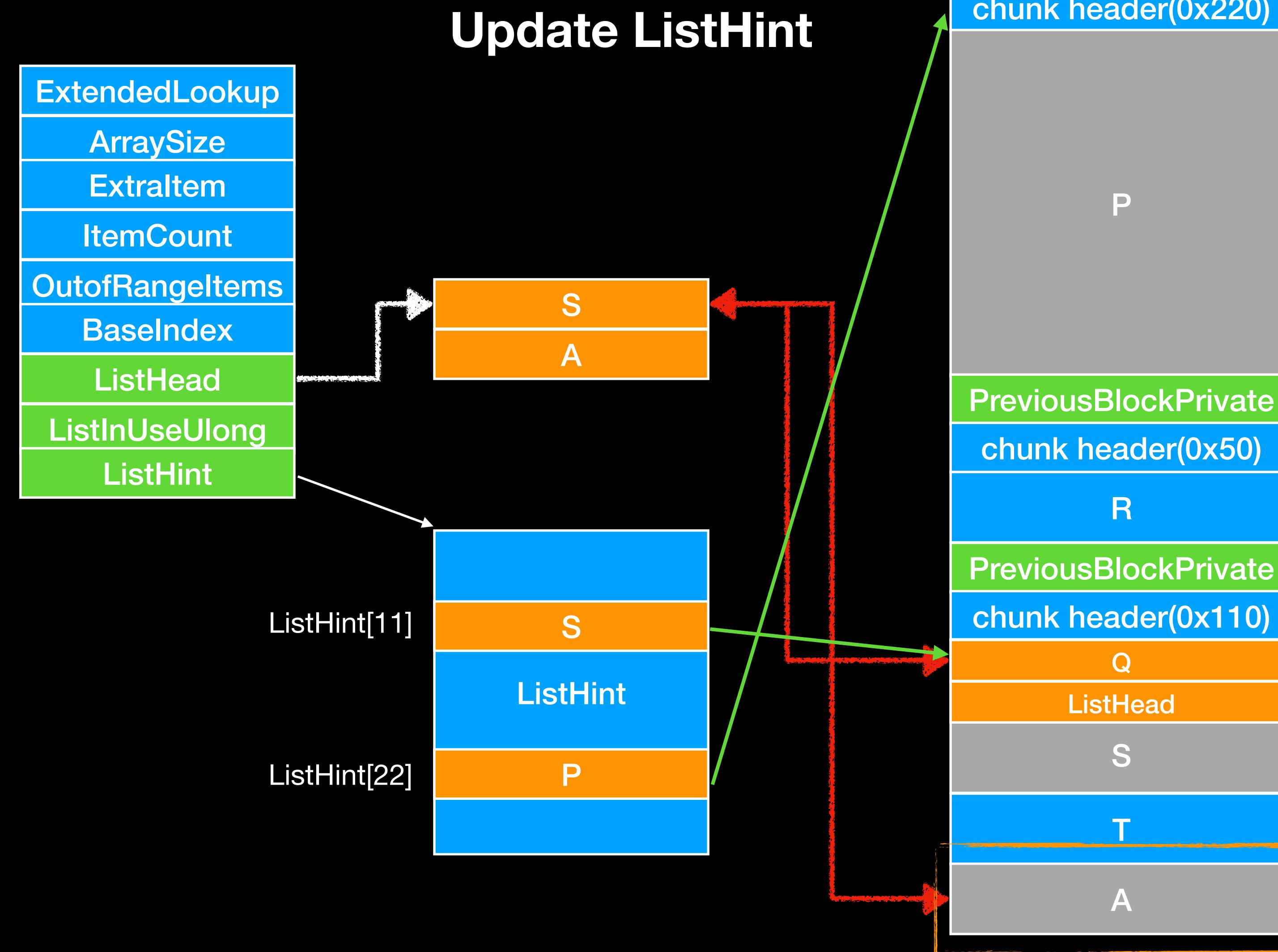
ExtendedLookup
ArraySize
ExtralItem
ItemCount
OutofRangeItems
BaseIndex
ListHead
ListInUseUlong
ListHint



Check Q->Flink == A?
Failed
Won't abort or insert

BackEnd Exploitation

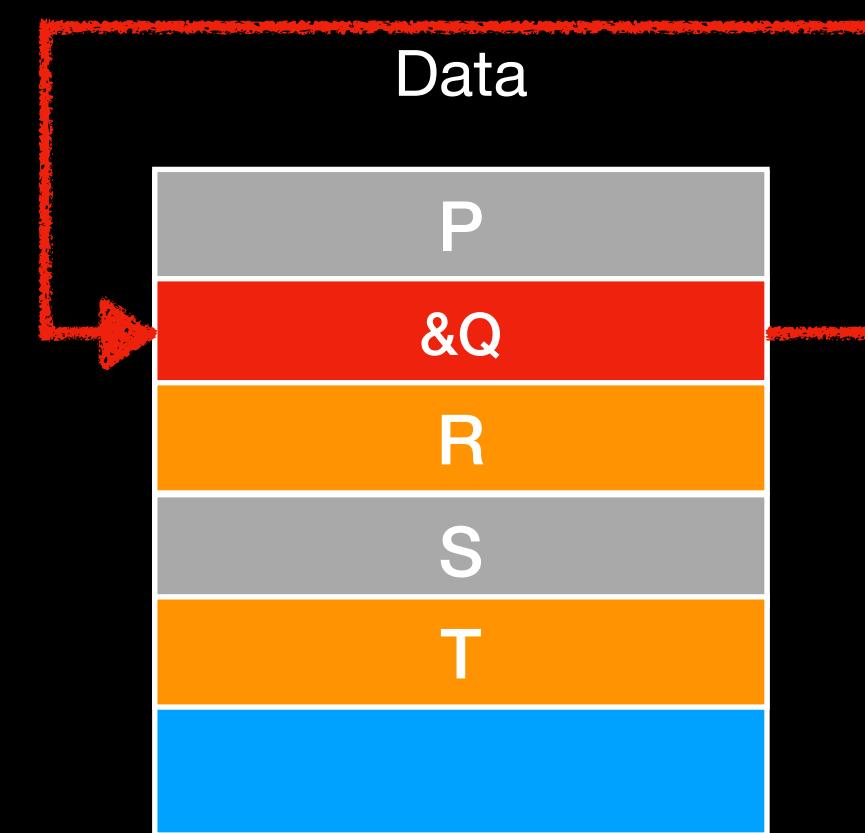
UAF



BackEnd Exploitation

UAF

- Edit Q
 - Arbitrary memory reading/writing primitive



BackEnd Exploitation

UAF

- After arbitrary memory reading/writing
 - Leak
 - kernel32.dll
 - ntdll.dll -> TIsBitMap+8 -> PEB.FIsBitmapBits
 - PEB +/- 1 page -> TEB
 - TEB -> function return address

BackEnd Exploitation

UAF

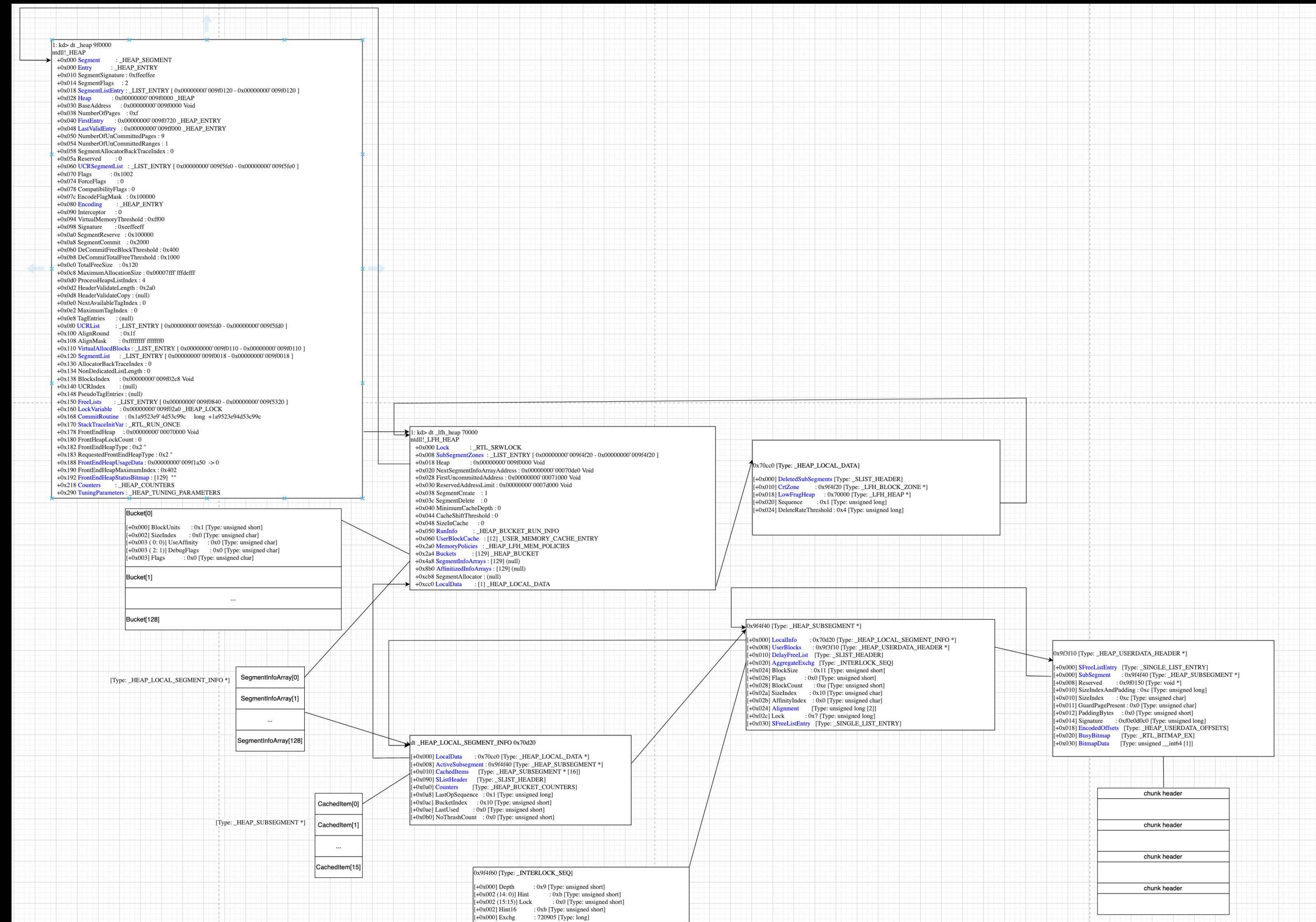
- After leaking
 - Write
 - Return address on stack
 - Control RIP
 - ROP to system("cmd.exe")
 - ROP to VirtualProtect/VirtualAlloc and jump to shell code

Windows memory allocator

- Nt Heap
 - FrontEnd
 - Data structure
 - Allocation mechanism

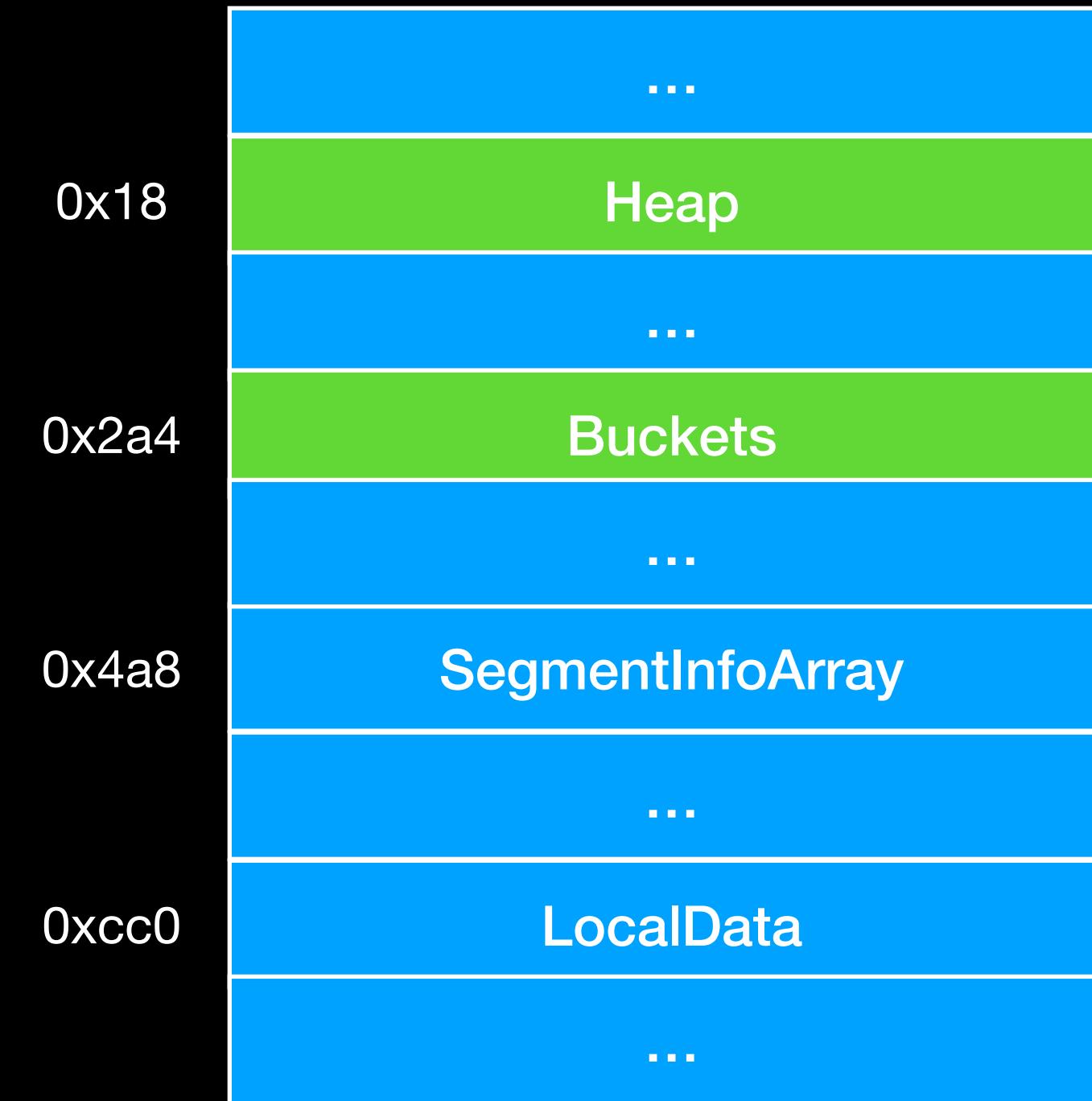
Windows Memory Allocator

- LFH Overview



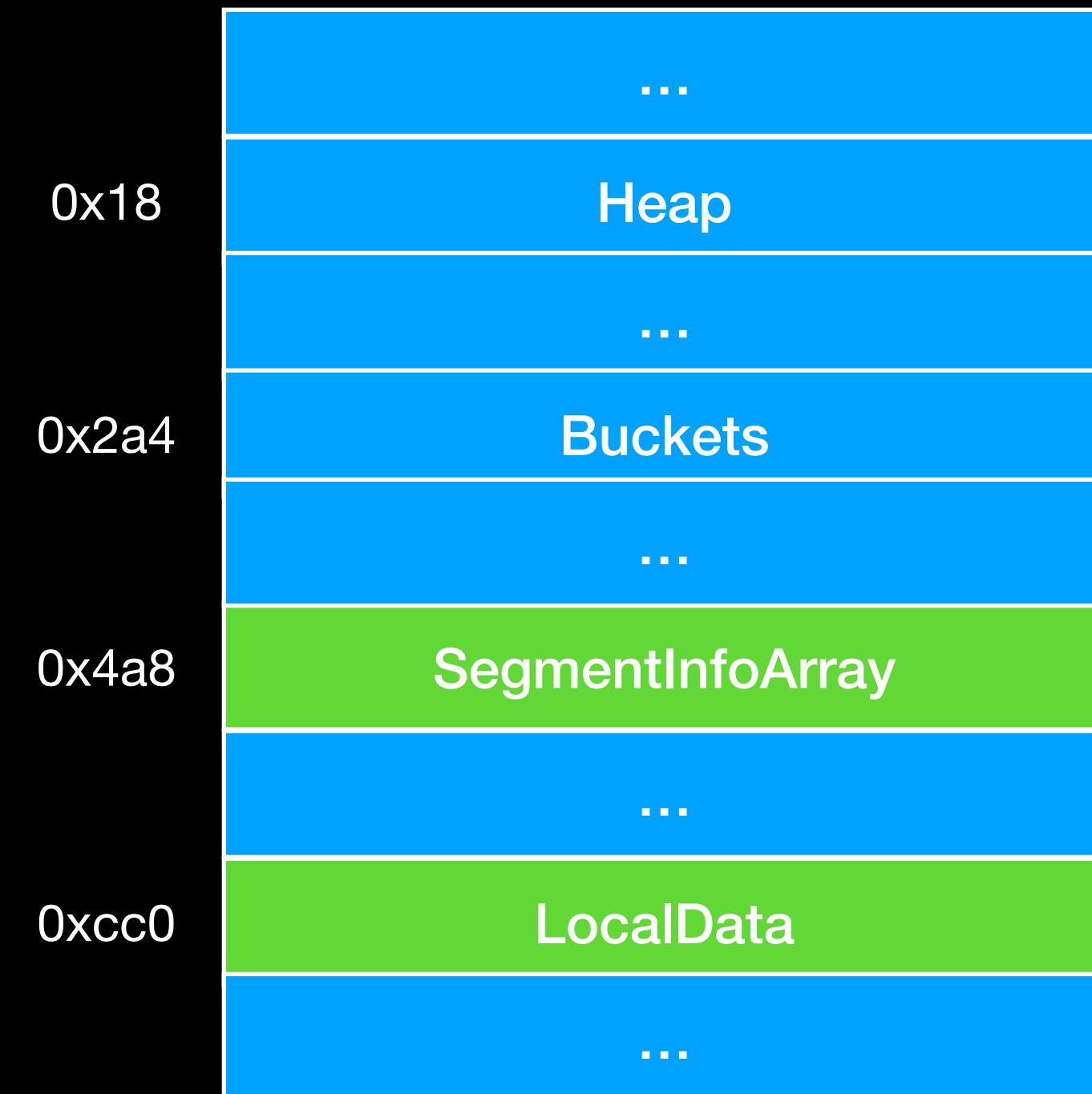
Windows Memory Allocator

- `_LFH_HEAP`
 - `Heap`
 - Point to `_HEAP`
 - `Buckets`
 - Array used to find the corresponding Block size



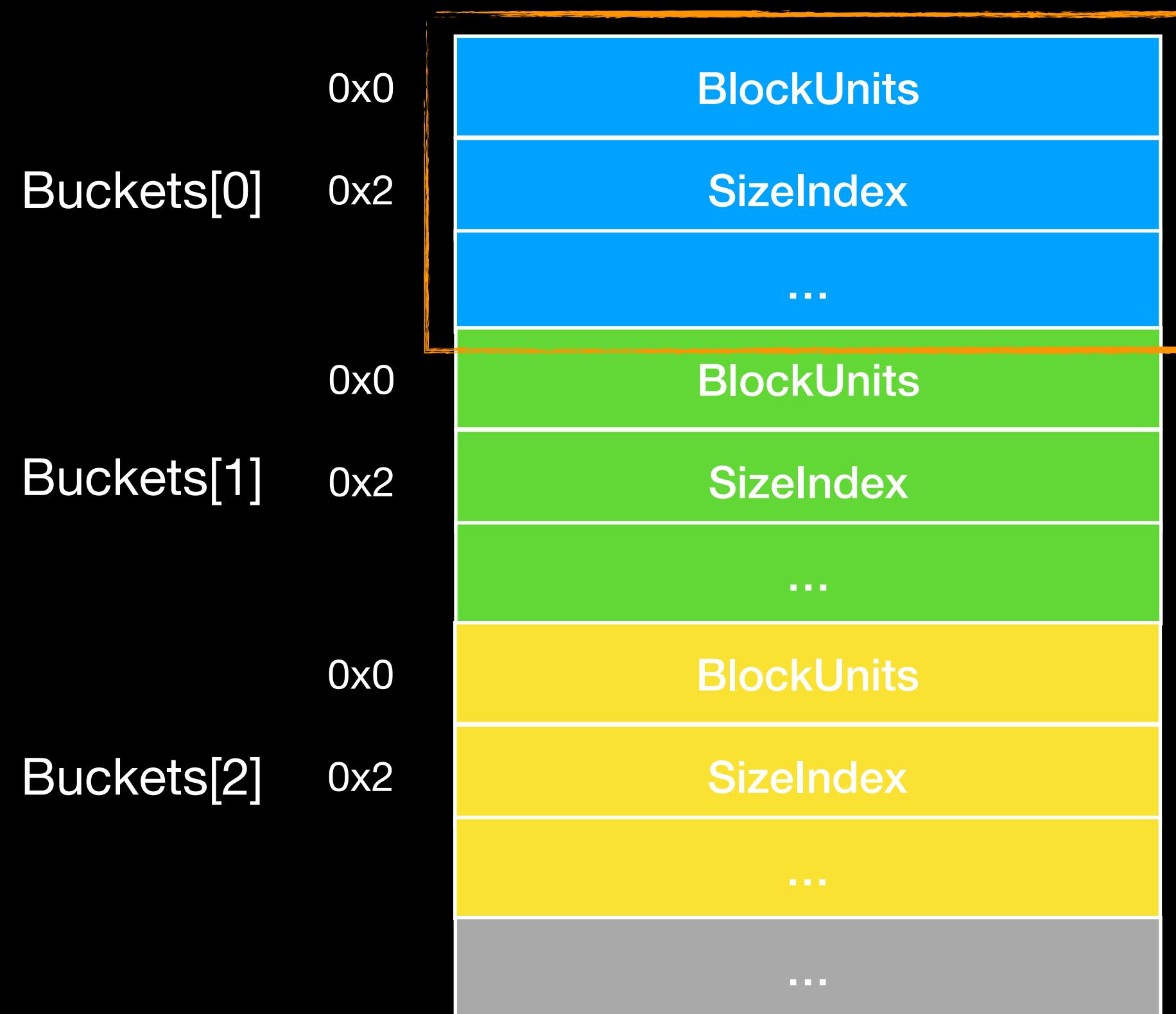
Windows Memory Allocator

- `_LFH_HEAP`
 - `SegmentInfoArray`
 - mainly managing the corresponding `SubSegment` information
 - `LocalData`
 - A field in it points to `_LFH`



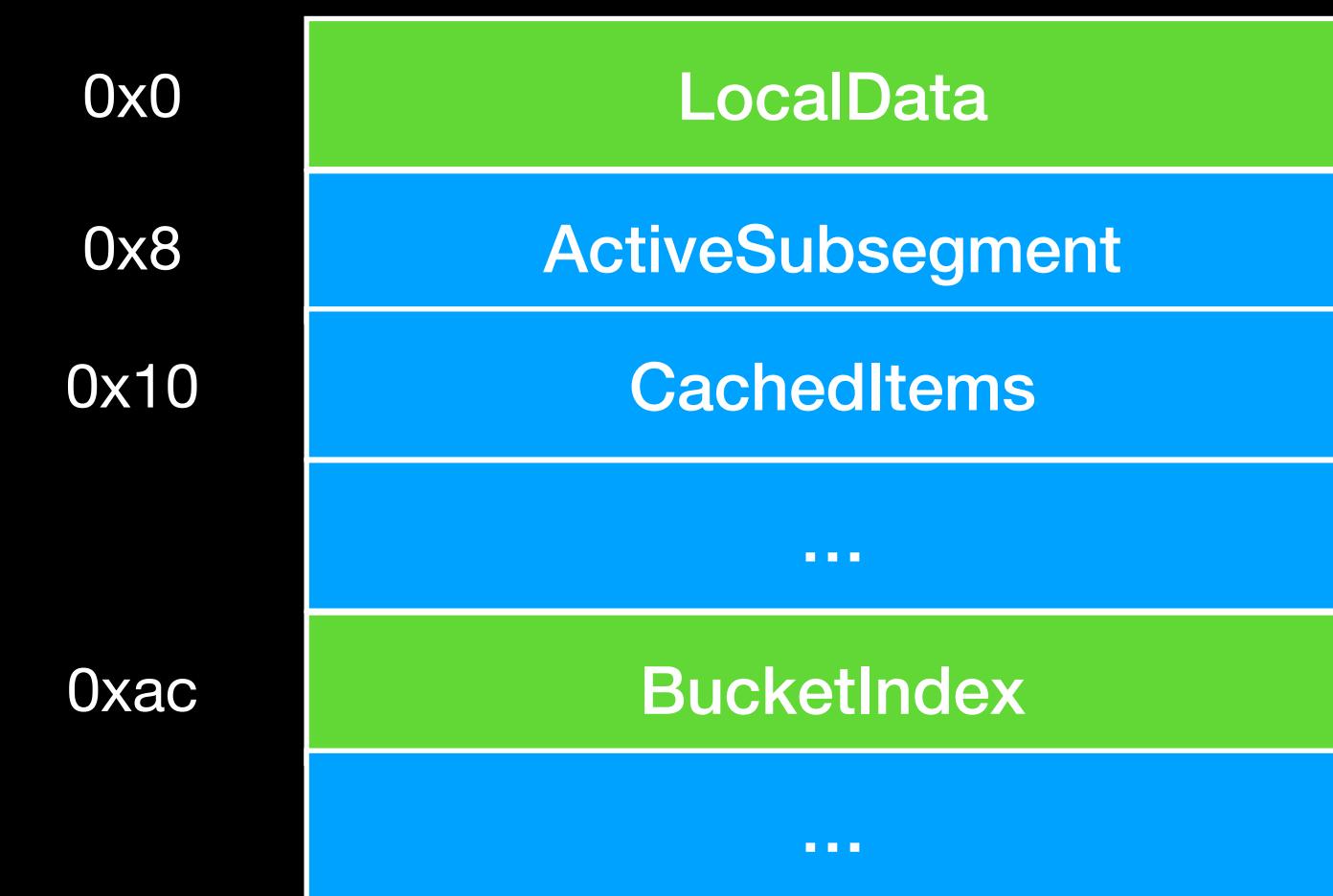
Windows Memory Allocator

- Buckets
 - Size 4 byte, array[0-128]
 - BlockUnits
 - Alloc size (real size $>> 4$)
 - Sizelndex
 - User need size (real size $>> 4$)



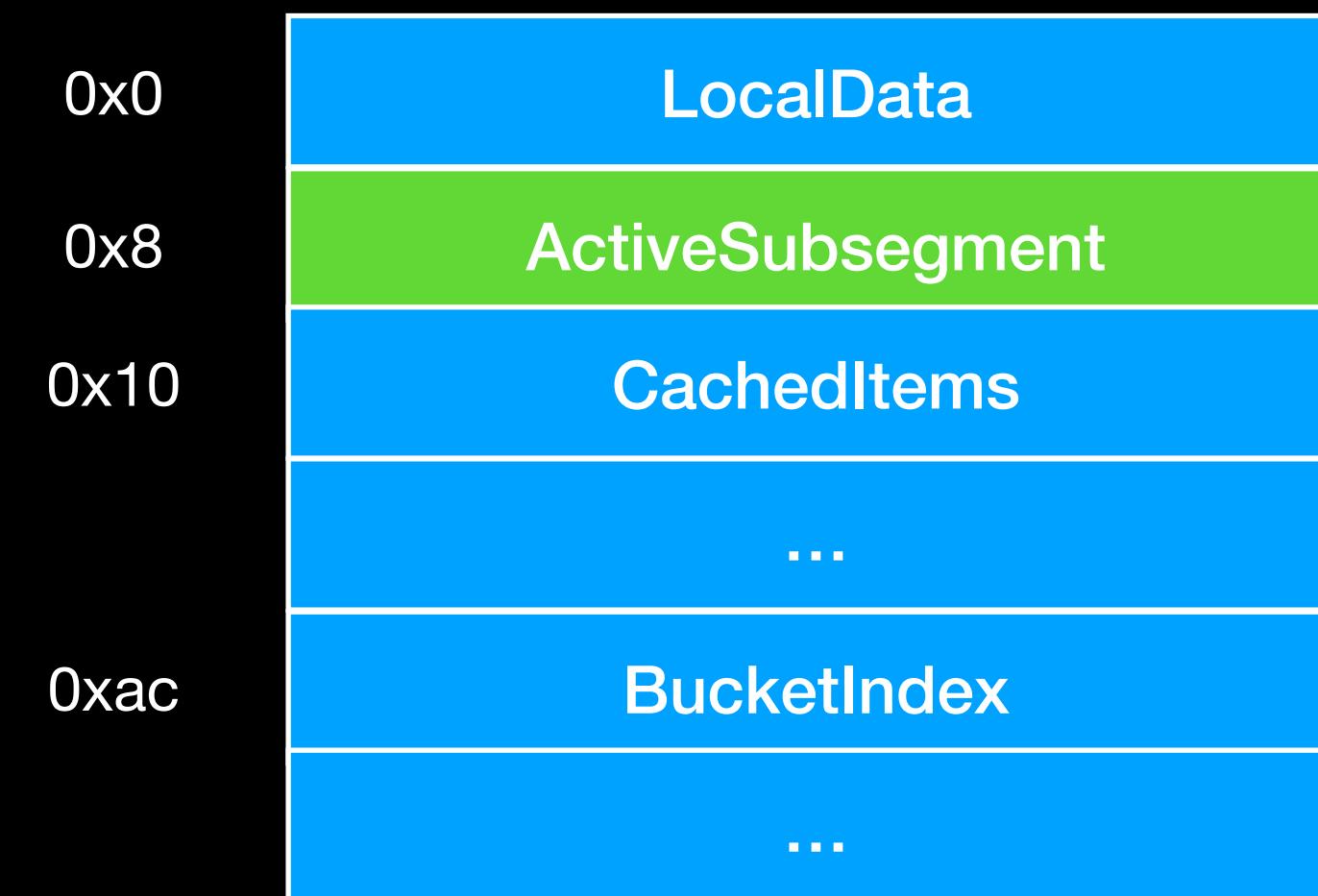
Windows Memory Allocator

- SegmentInfoArray[x]
 - LocalData
 - Point to _LFH_HEAP->LocalData
 - BucketIndex
 - Index of Buckets[x]



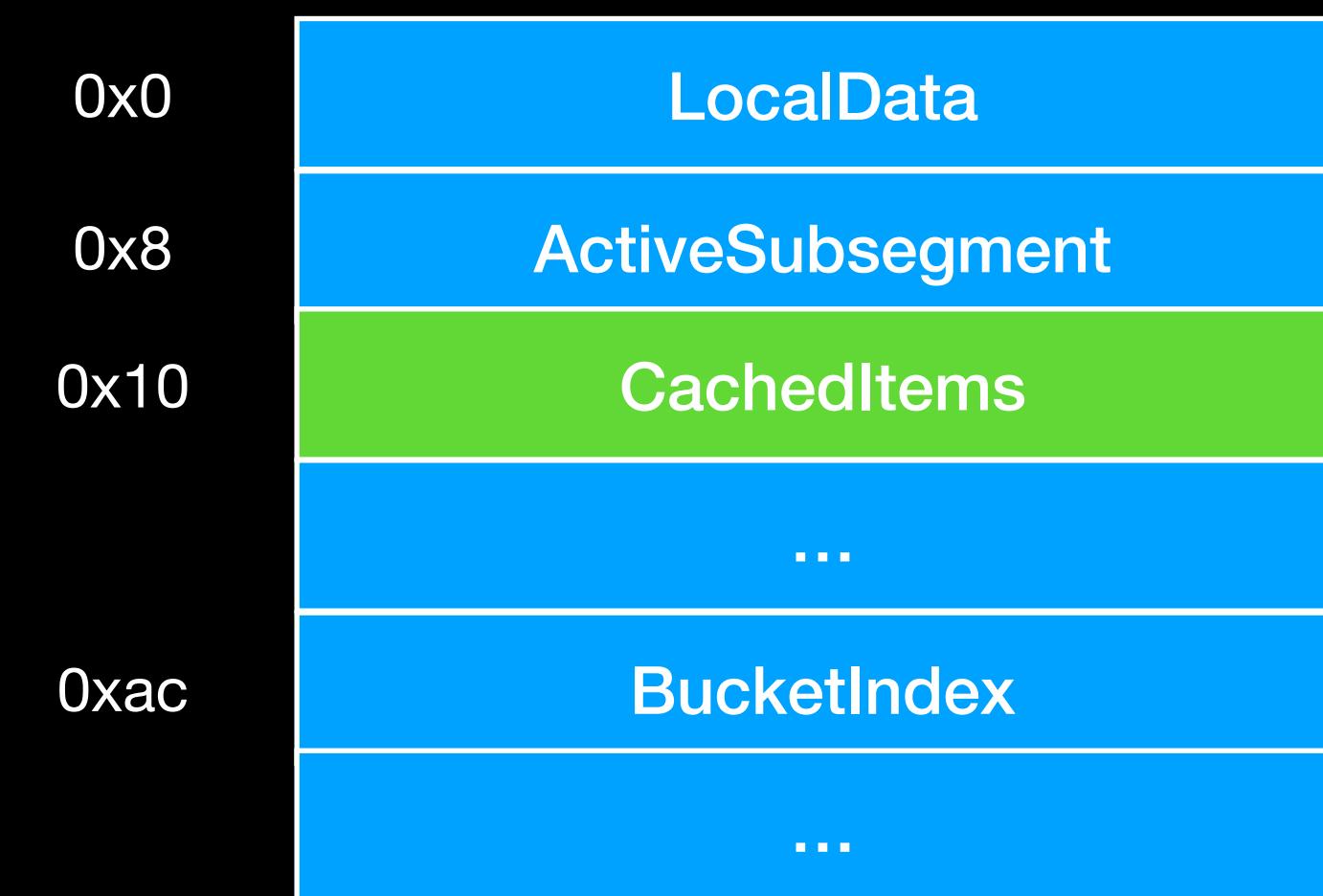
Windows Memory Allocator

- SegmentInfoArray[x]
 - ActiveSubsegment
 - Type _HEAP_SUBSEGMENT
 - Corresponding to the allocated subsegment
 - manage Userblock
 - chunk allocation info



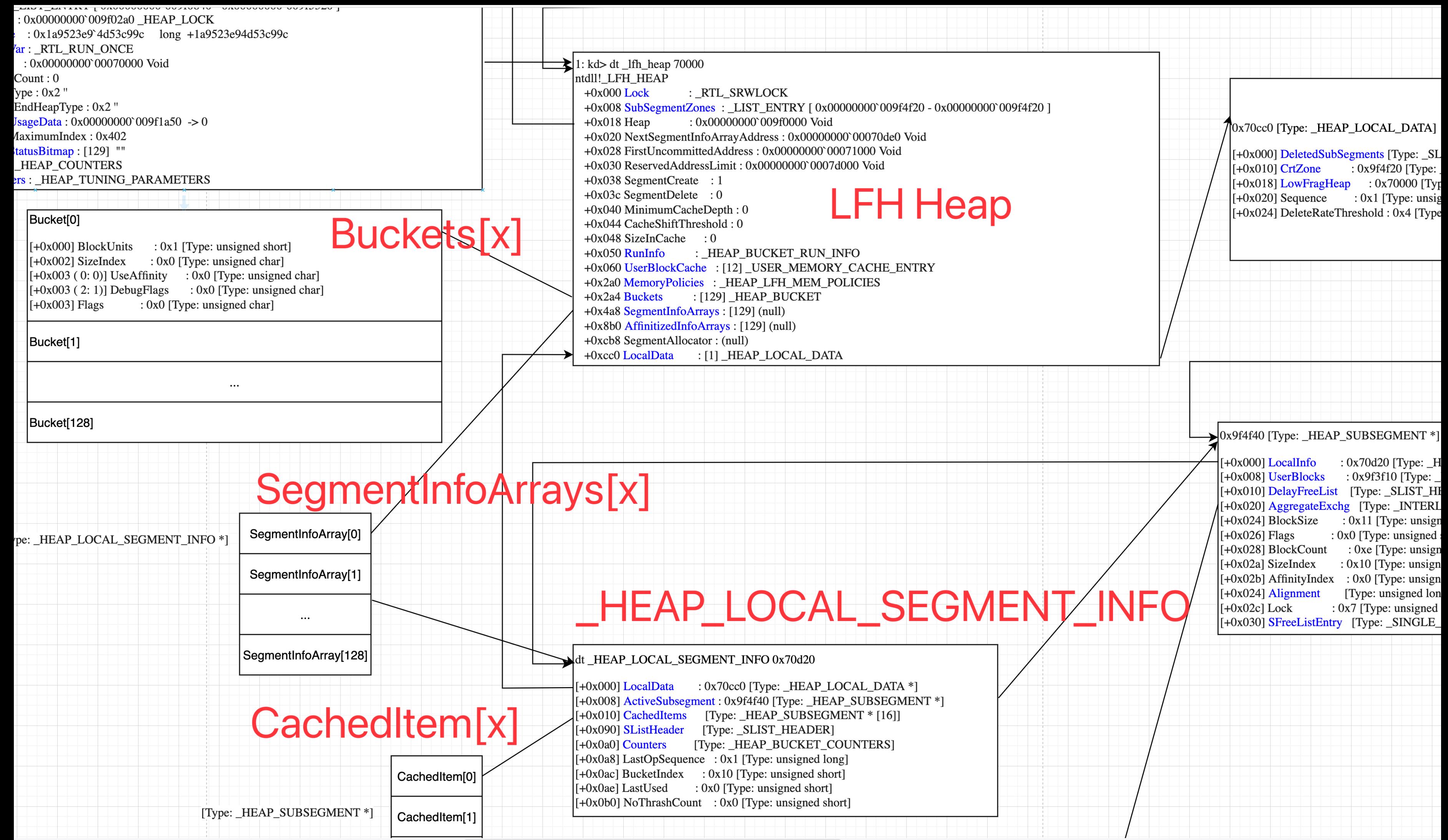
Windows Memory Allocator

- SegmentInfoArray[x]
 - CachedItems
 - `_HEAP_SUBSEGMENT` array
 - Stores the Subsegment corresponding to the SegmentInfo which can assign chunks to users
 - When ActiveSubsegment runs out, it will be filled from this field



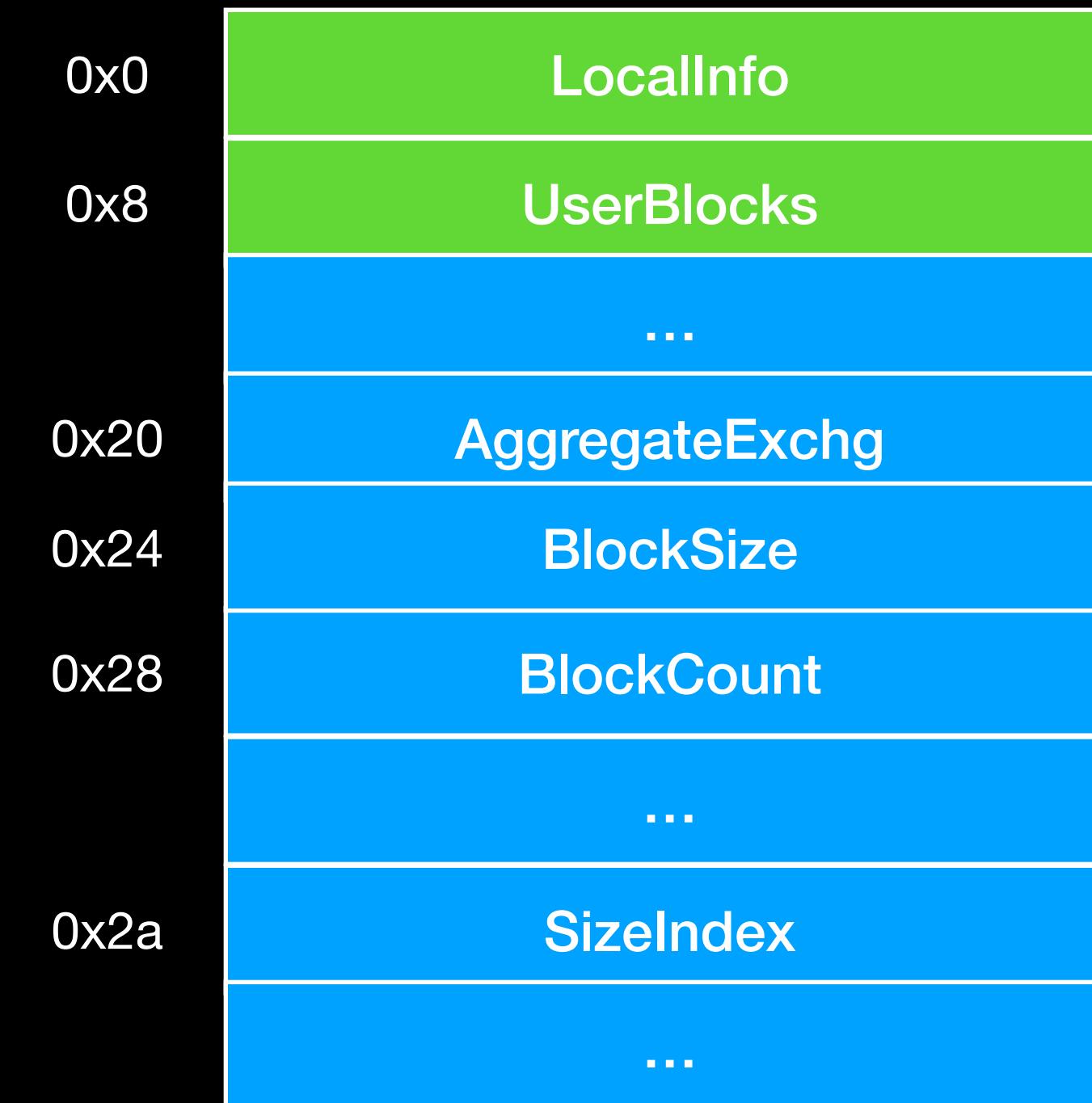
Windows Memory Allocator

LFH view1



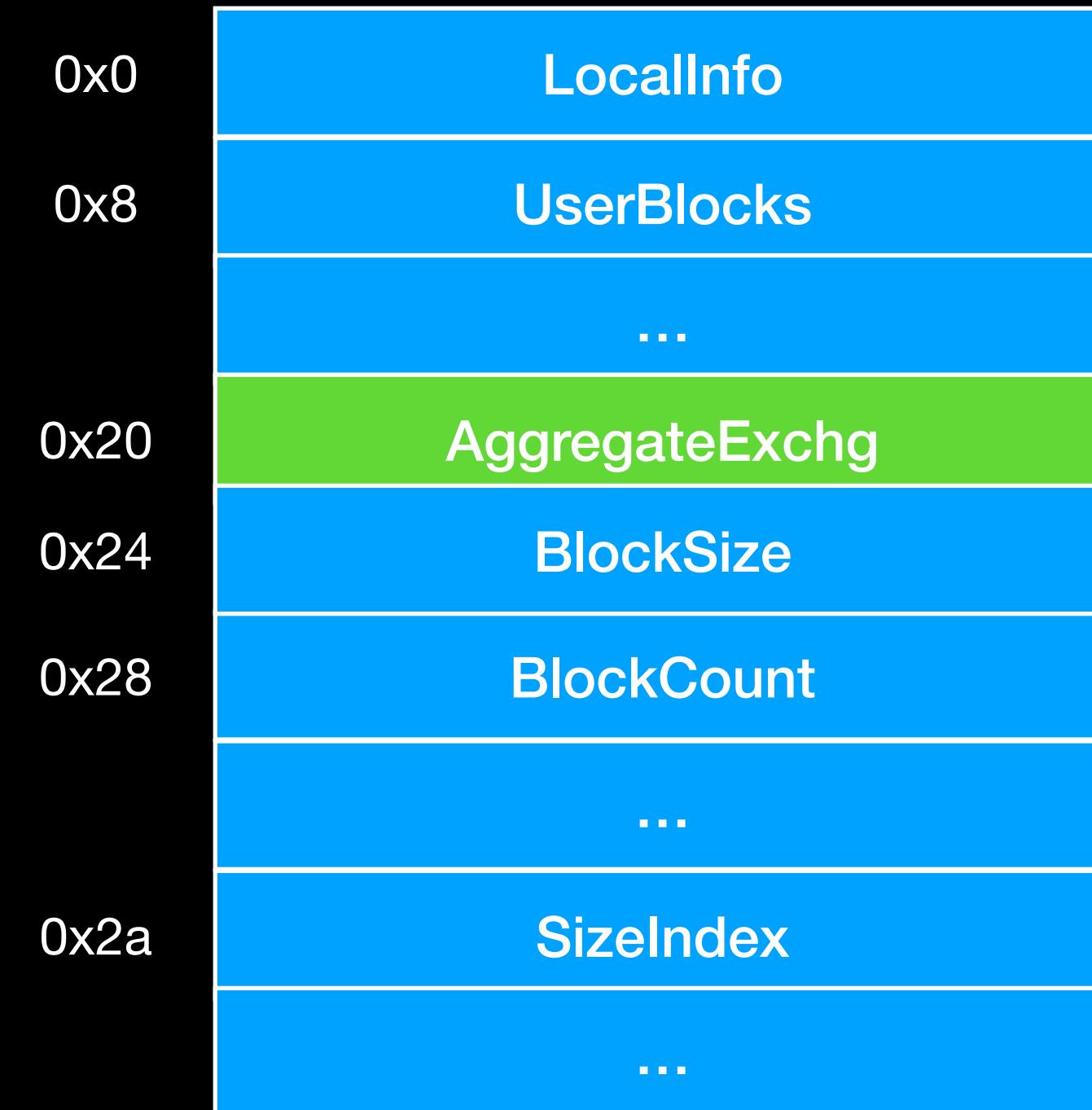
Windows Memory Allocator

- ActiveSubsegment
 - LocallInfo
 - Point to corresponding SegmentInfoArray[x].LocalData(0x0)
 - UserBlocks
 - LFH memory pool
 - metadata + chunks



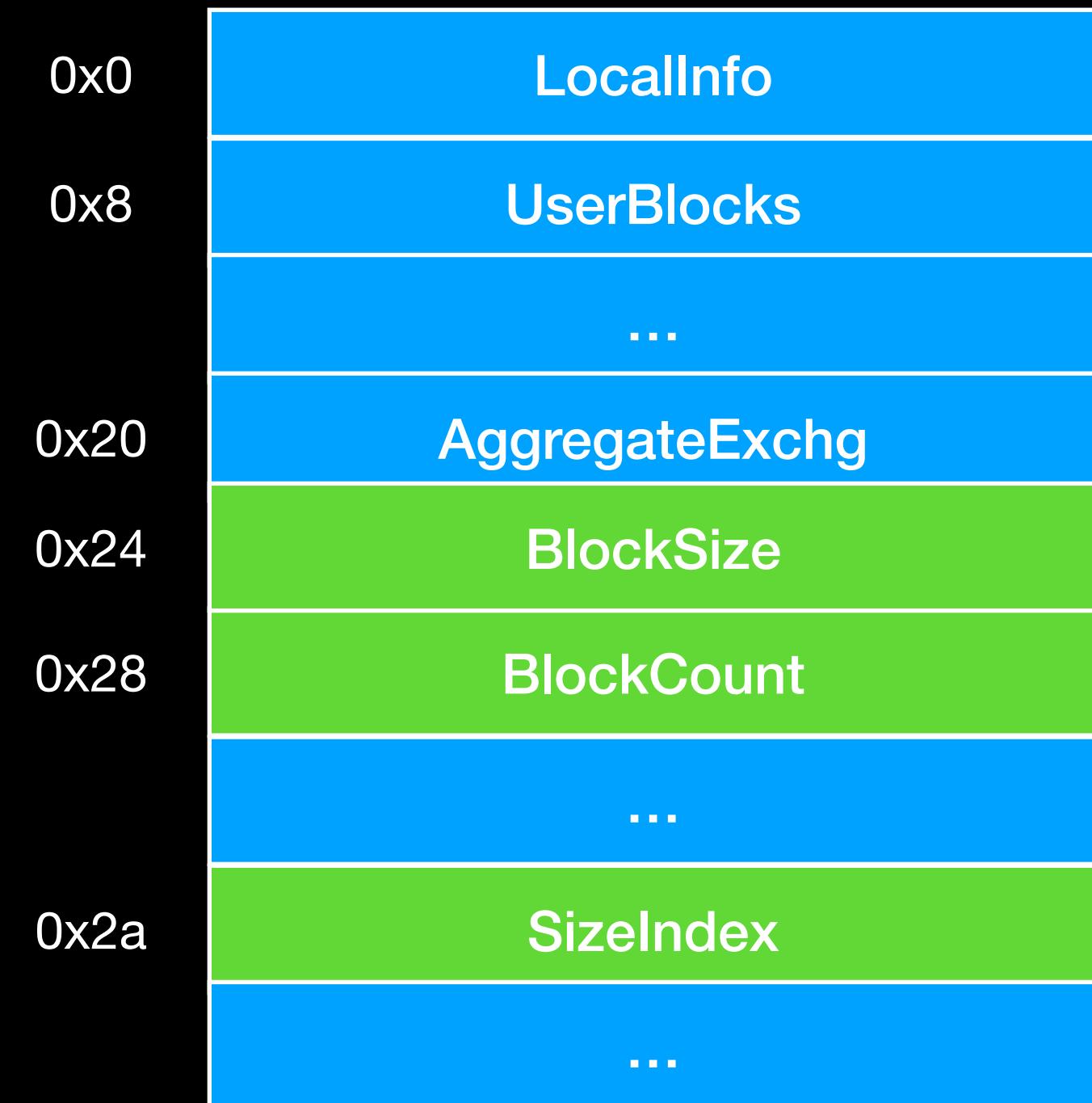
Windows Memory Allocator

- ActiveSubsegment
 - AggregateExchg
 - Num of free chunks in this UserBlock
 - LFH uses this field to determine whether to allocate from the UserBlock
 - Lock



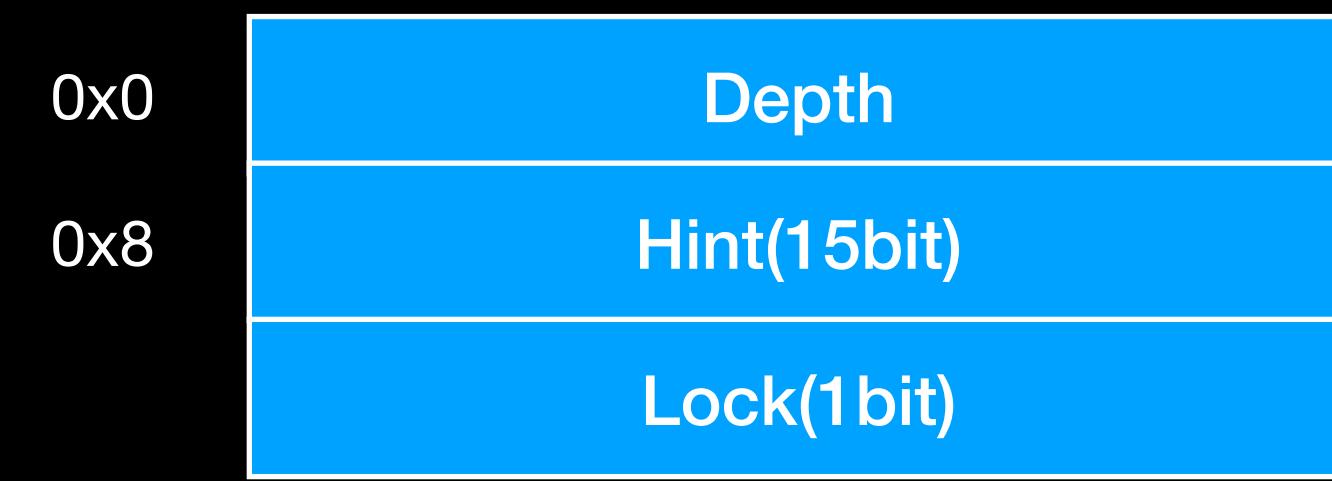
Windows Memory Allocator

- ActiveSubsegment
 - BlockSize
 - chunk(block) size in this UserBlock
 - BlockCount
 - Num of chunks in this UserBlock
 - SizelIndex
 - SizelIndex corresponding to the UserBlock



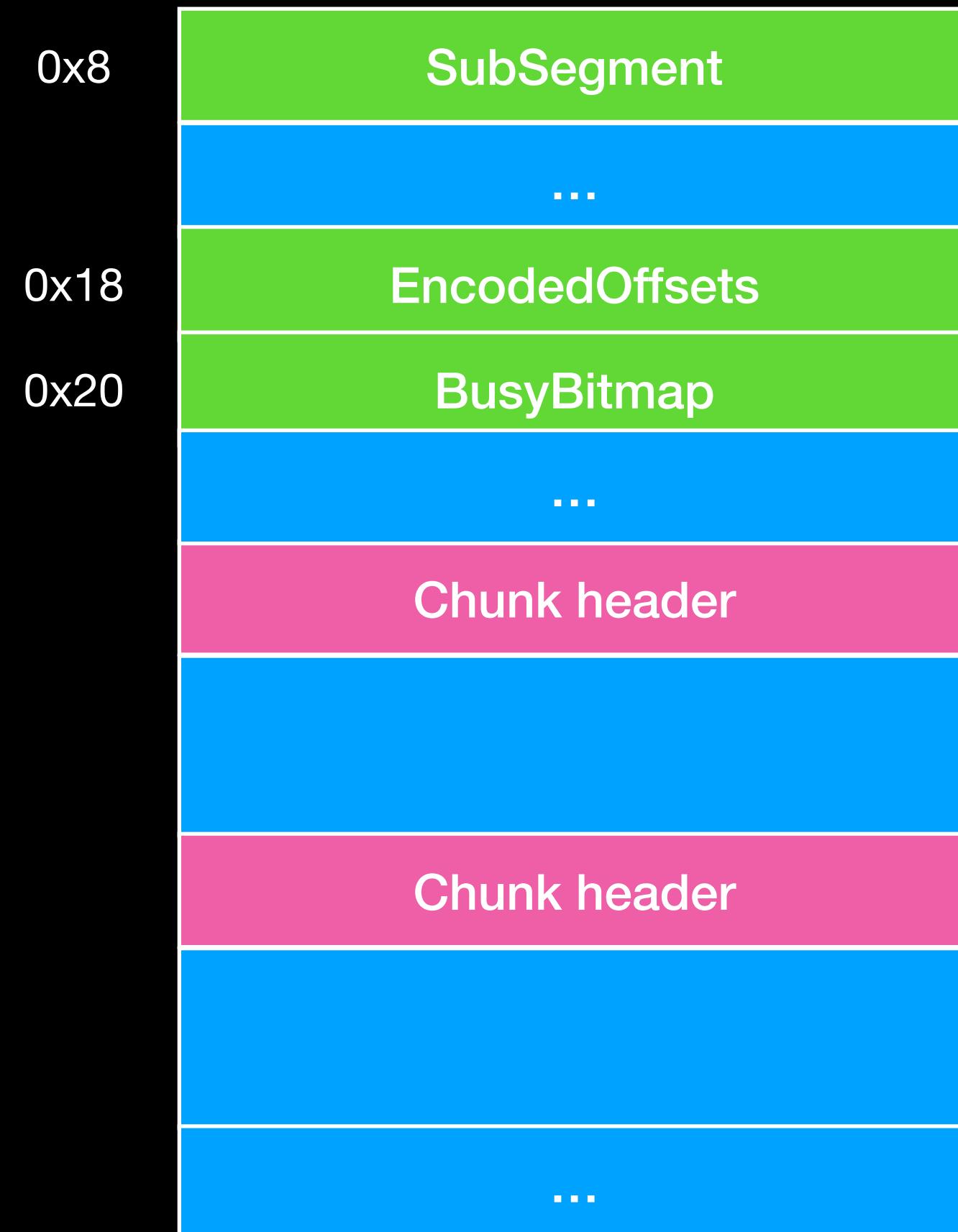
Windows Memory Allocator

- AggregateExchg
 - Depth
 - Num of free chunks in this UserBlock
 - Lock



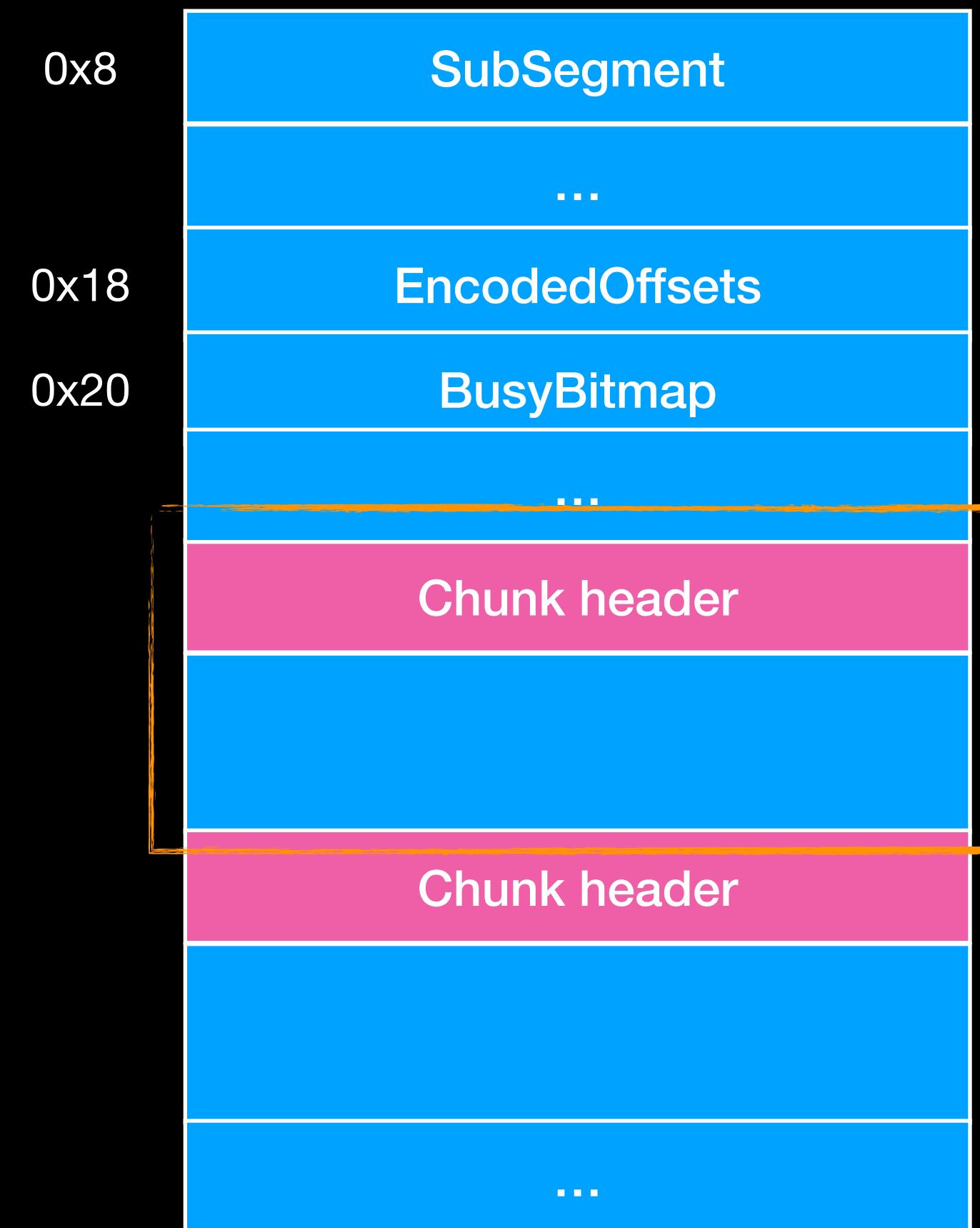
Windows Memory Allocator

- UserBlock
 - SubSegment
 - Point to ActiveSubsegment
 - EncodedOffsets
 - Verify the integrity of chunk header
 - BusyBitmap
 - Bitmap of chunk in use



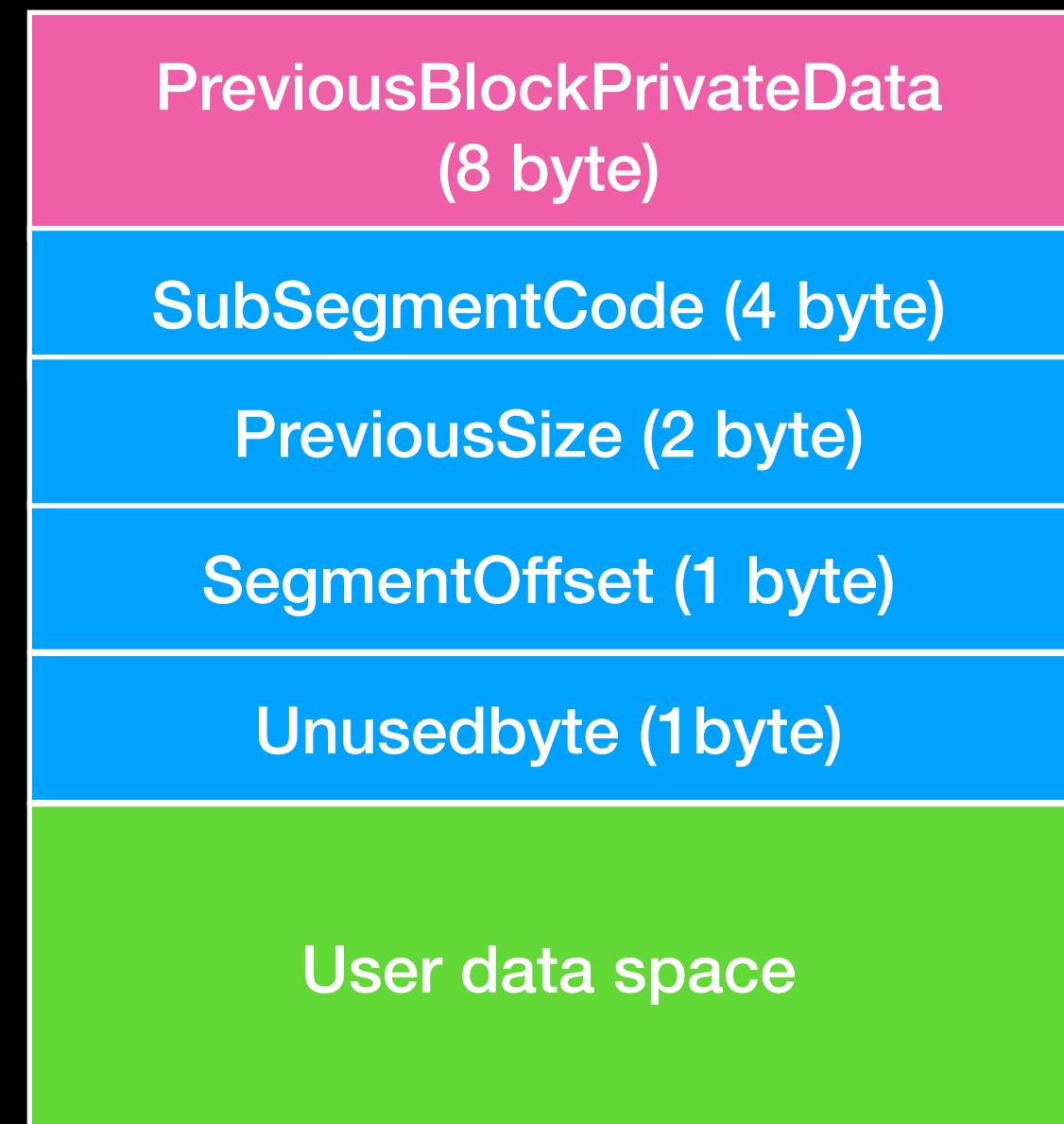
Windows Memory Allocator

- UserBlock
 - Block (chunk)
 - Chunks that LFH returns to user



Windows Memory Allocator in used LFH chunk

- SubSegmentCode
 - encoded metadata used to point UserBlock
 - SegmentOffset
 - Index of chunk in UserBlock
 - Unusedbyte
 - Unusedbyte & 0x80 is true
 - 0x80
 - Used to judge whether it is LFH



Windows memory allocator

- Remark
 - EncodedOffsets
 - xor of the following 4 value
 - (sizeof(userblock header) | (BlockUnit*0x10<<16))
 - LFHkey
 - Userblock address
 - _LFH_HEAP address

Windows memory allocator

- Remark
 - LFH header encoding
 - applied on every chunk header's initialization
 - xor of the following 4 value
 - _HEAP address
 - LFHKey
 - Chunk address >> 4
 - (Chunk address - UserBlock address) << 12

Windows memory allocator

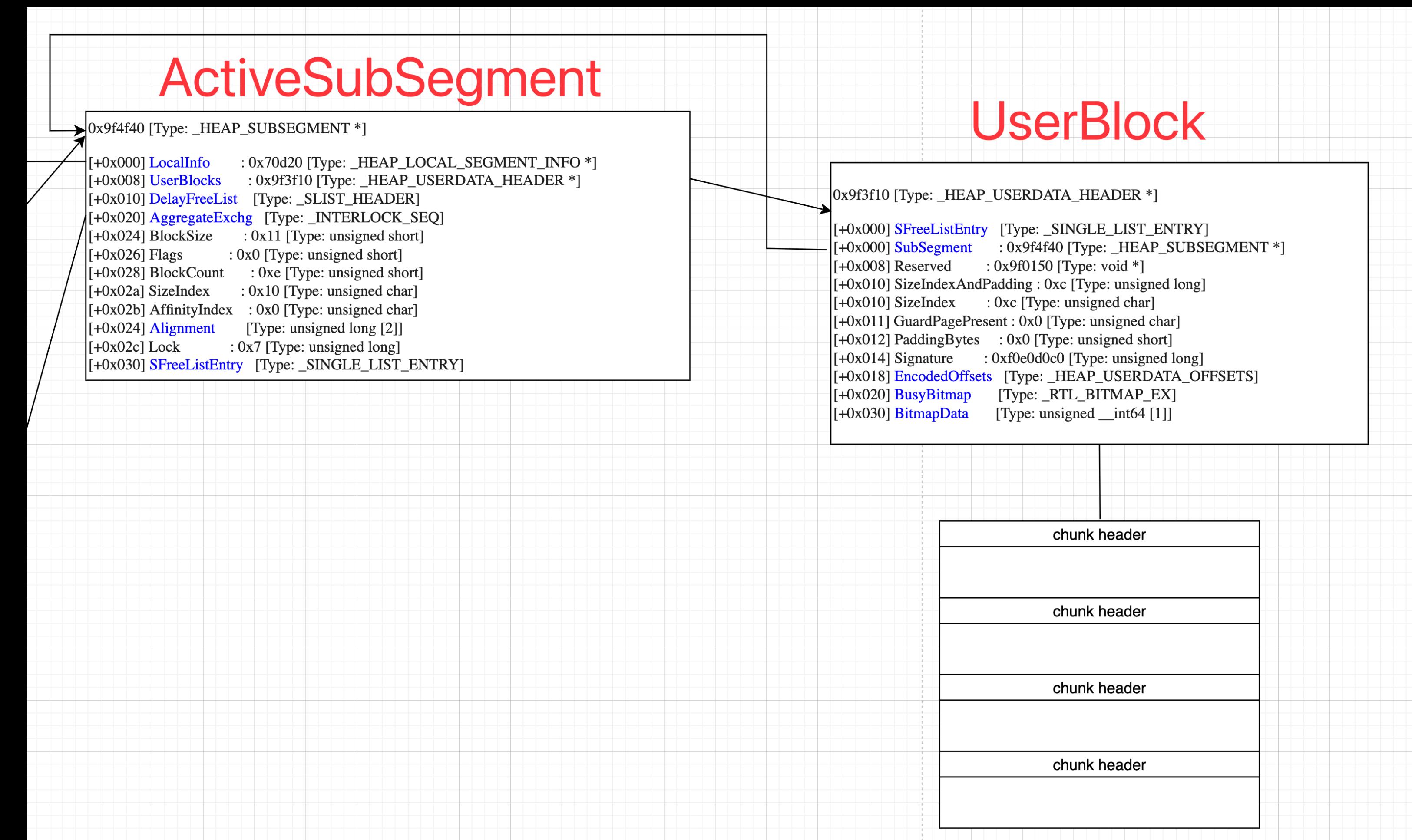
- Remark

- LFHkey

```
1void *RtlpInitializeLowFragHeapManager()
2{
3    unsigned int v0; // ebx
4    char v1; // cl
5    unsigned int v2; // ecx
6    void *result; // rax
7    char SystemInformation[56]; // [rsp+20h] [rbp-58h] BYREF
8    char v5; // [rsp+58h] [rbp-20h]
9
10   v0 = 1;
11   RtlpLFHKey = RtlpHeapGenerateRandomValue64();
12   if ( NtQuerySystemInformation(SystemBasicInformation, SystemInformation, 0x40u, 0i64) >= 0 )
13   {
14       v1 = 0;
15       if ( v5 > 1 )
16       {
```

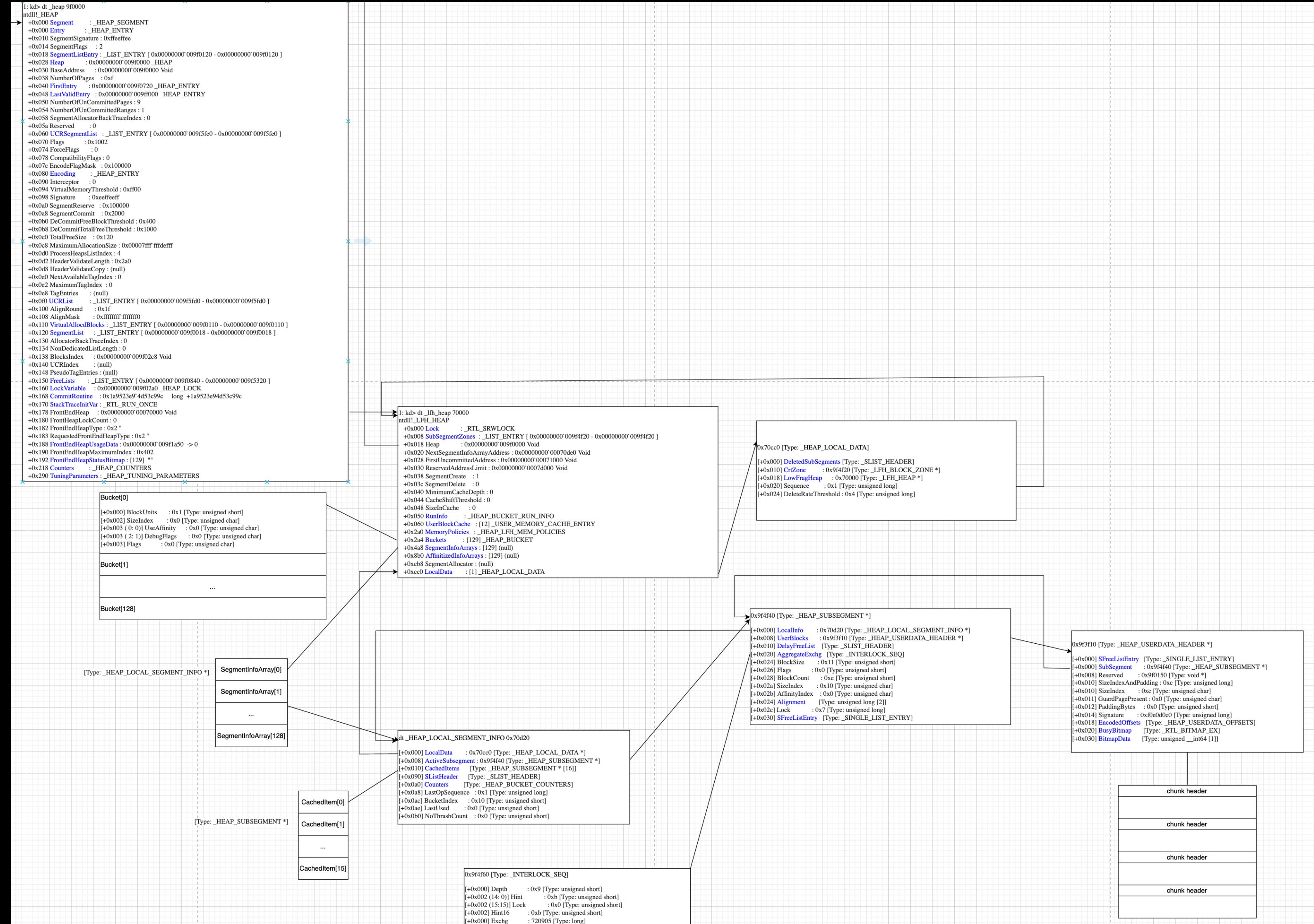
Windows Memory Allocator

LFH view2



Windows Memory Allocator

LFH Overview



Windows Memory Allocator

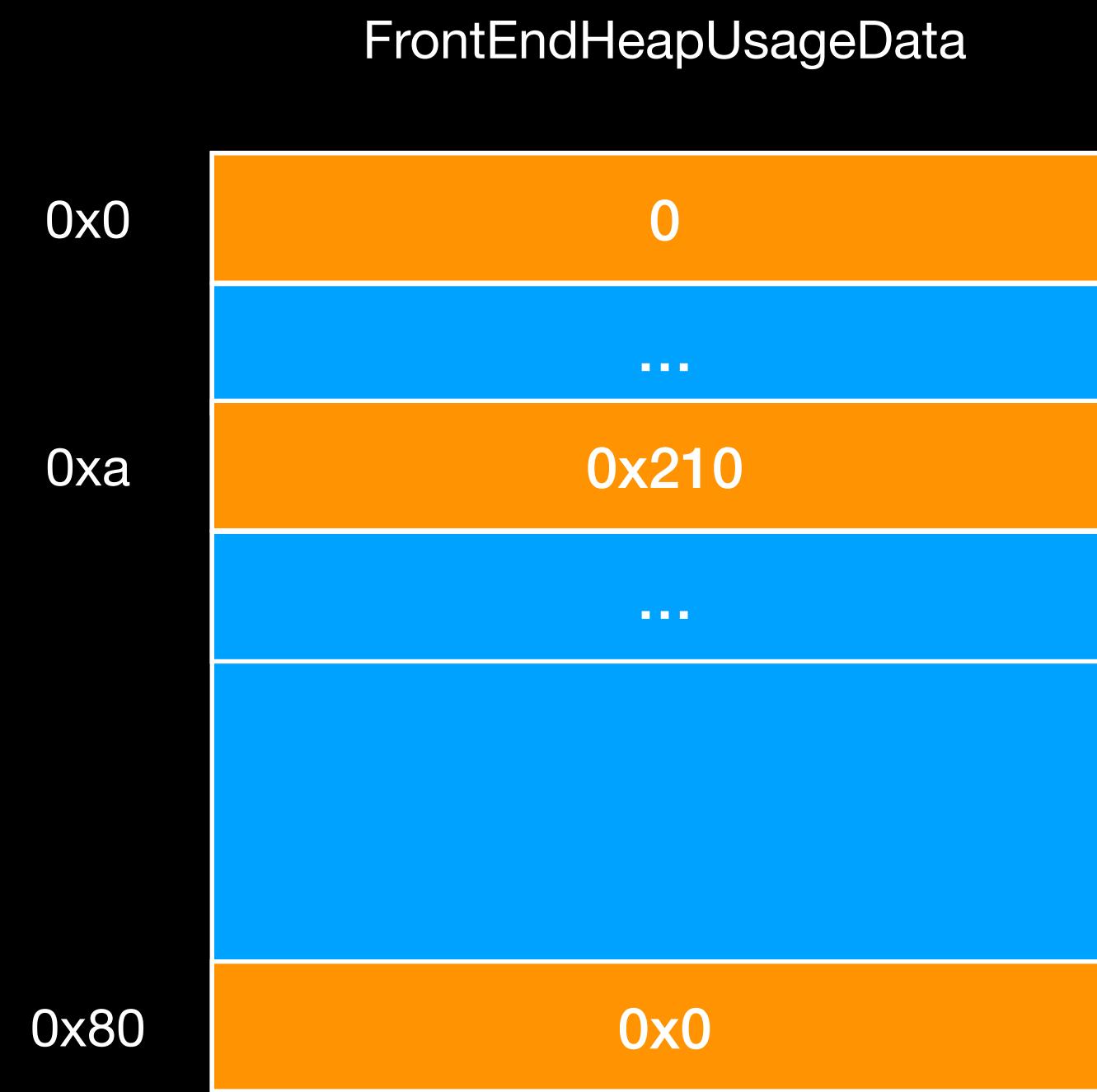
FrontEnd allocation mechanism

- FrontEnd
 - Initialization
 - When `FrontEndHeapUsageData[x] & 0x1f > 0x10`, the next allocate will initialize the LFH
 - `ExtendFrontEndUsageData`, add a larger `BlocksIndex`
 - build `FrontEndHeap`
 - initialize `SegmentInfoArrays[idx]`
 - LFH chunks will be used in next same size allocation

Windows Memory Allocator

FrontEnd allocation mechanism

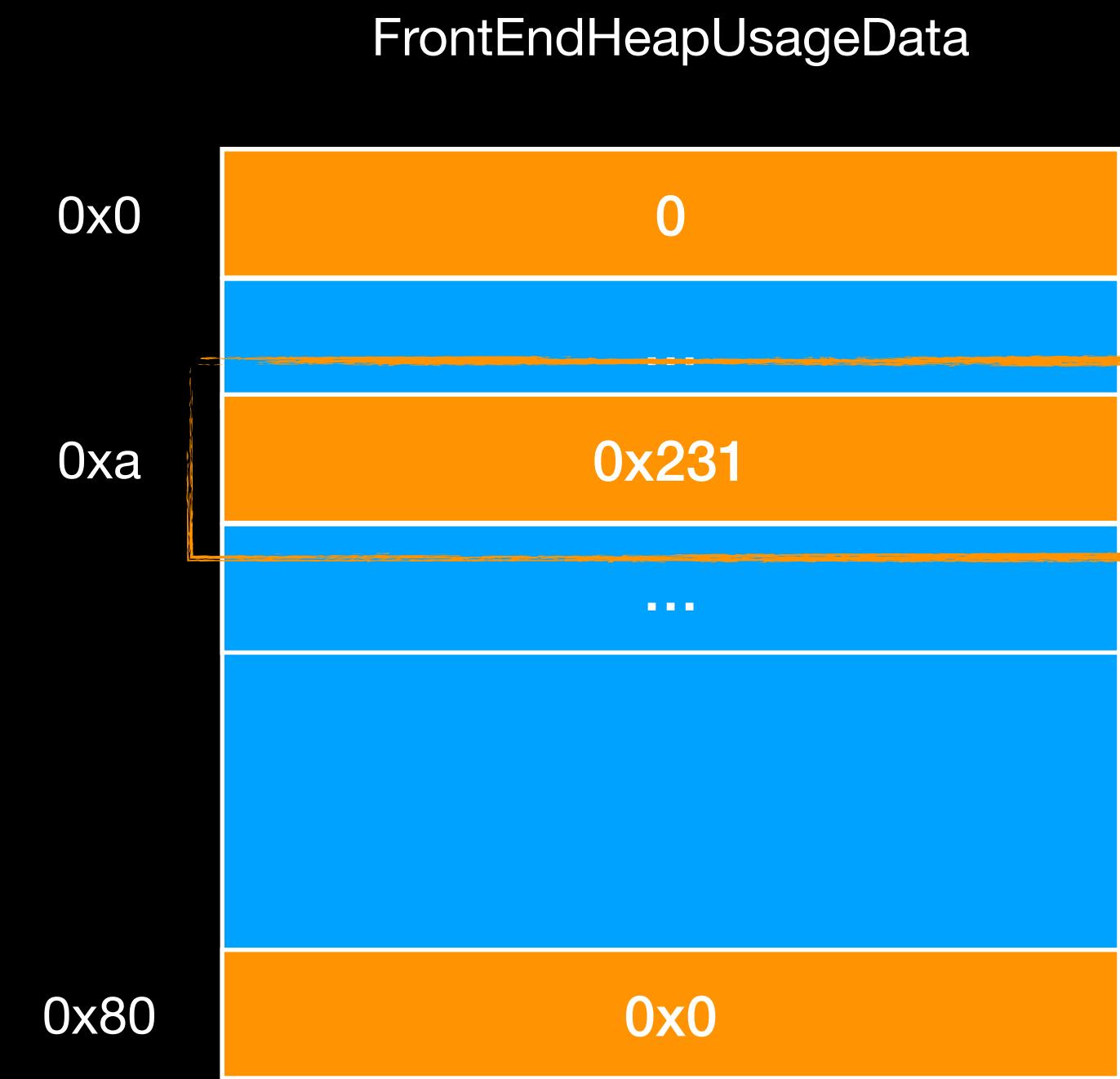
- LFH initialization
 - `malloc(0x40) * 16`
 - word elements



Windows Memory Allocator

FrontEnd allocation mechanism

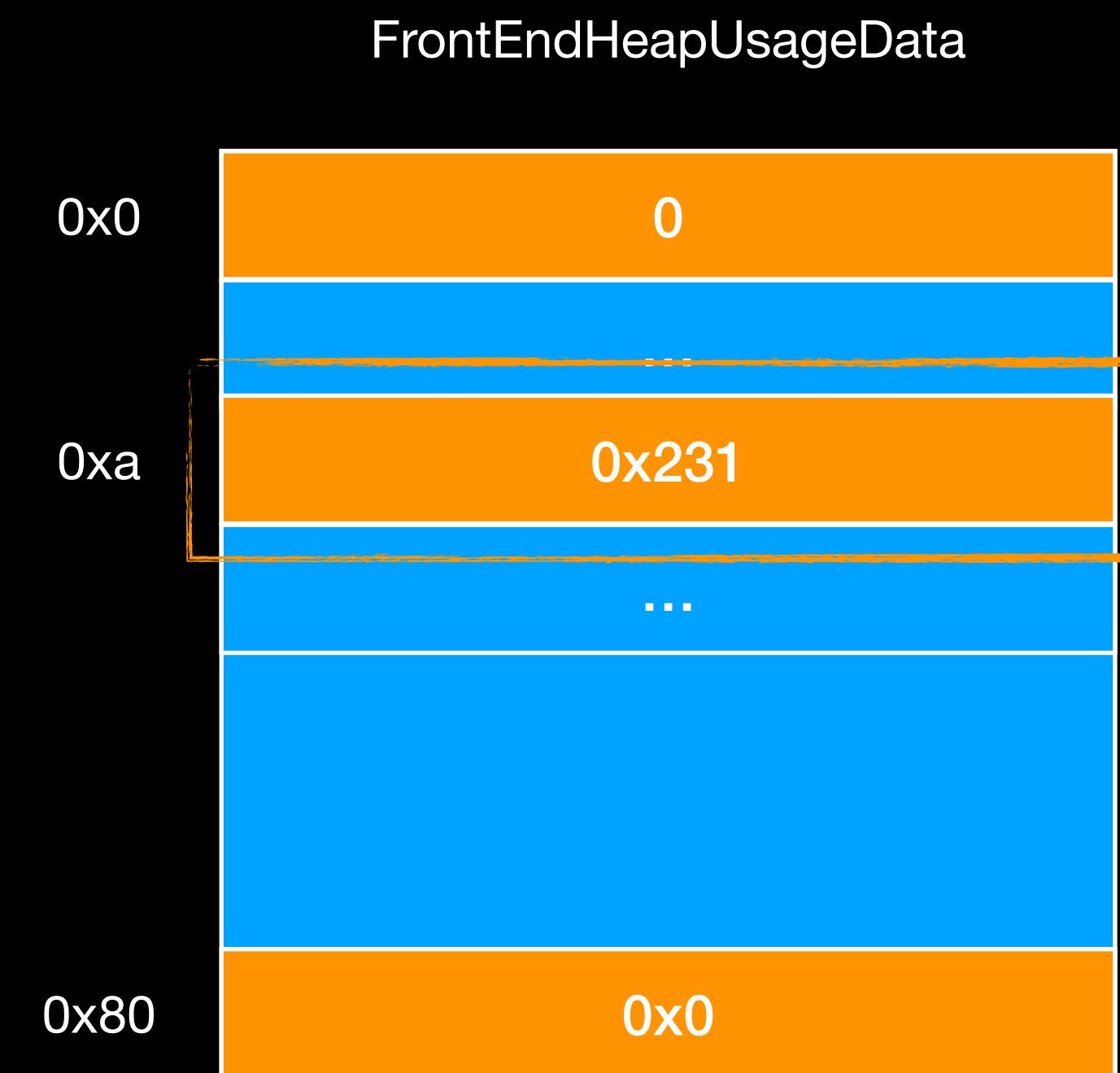
- LFH initialization
 - `malloc(0x40) * 17`
 - $0x231 \& 0x1f > 0x10$



Windows Memory Allocator

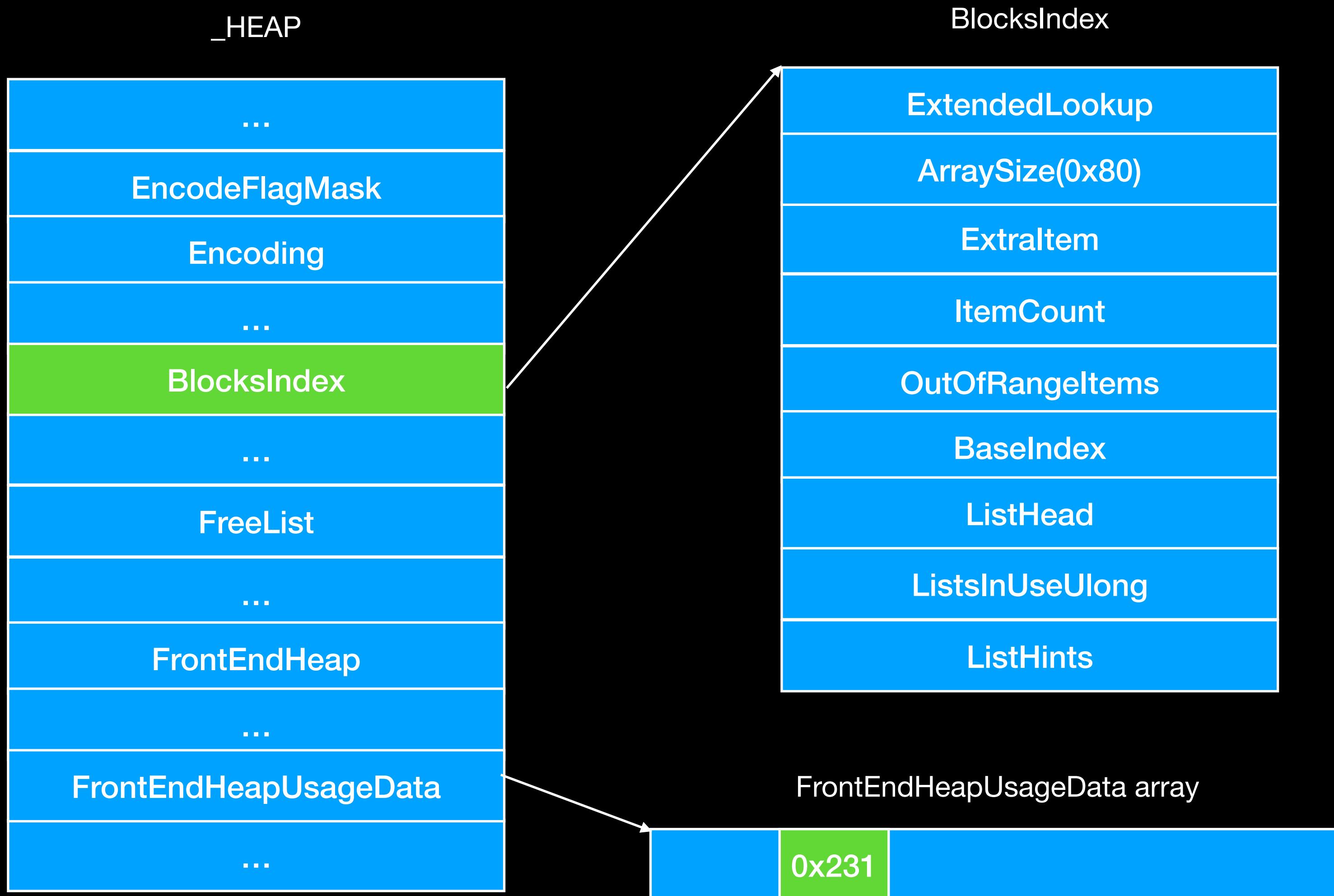
FrontEnd allocation mechanism

- LFH initialization
 - `malloc(0x40) * 17`
 - `Heap->Compatibility |= 0x20000000`
 - the next allocate will initialize the LFH



Windows Memory Allocator

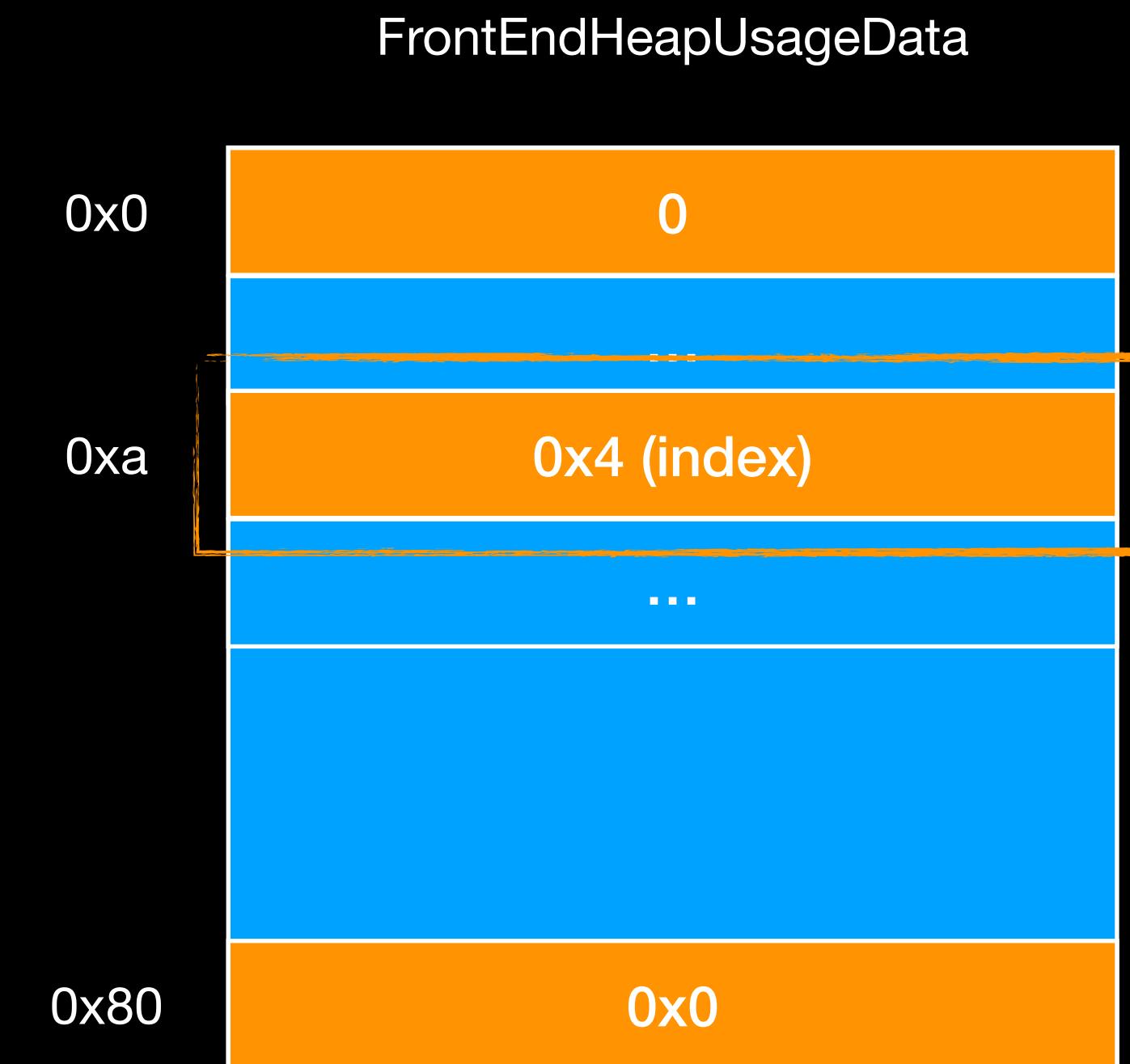
FrontEnd allocation mechanism



Windows Memory Allocator

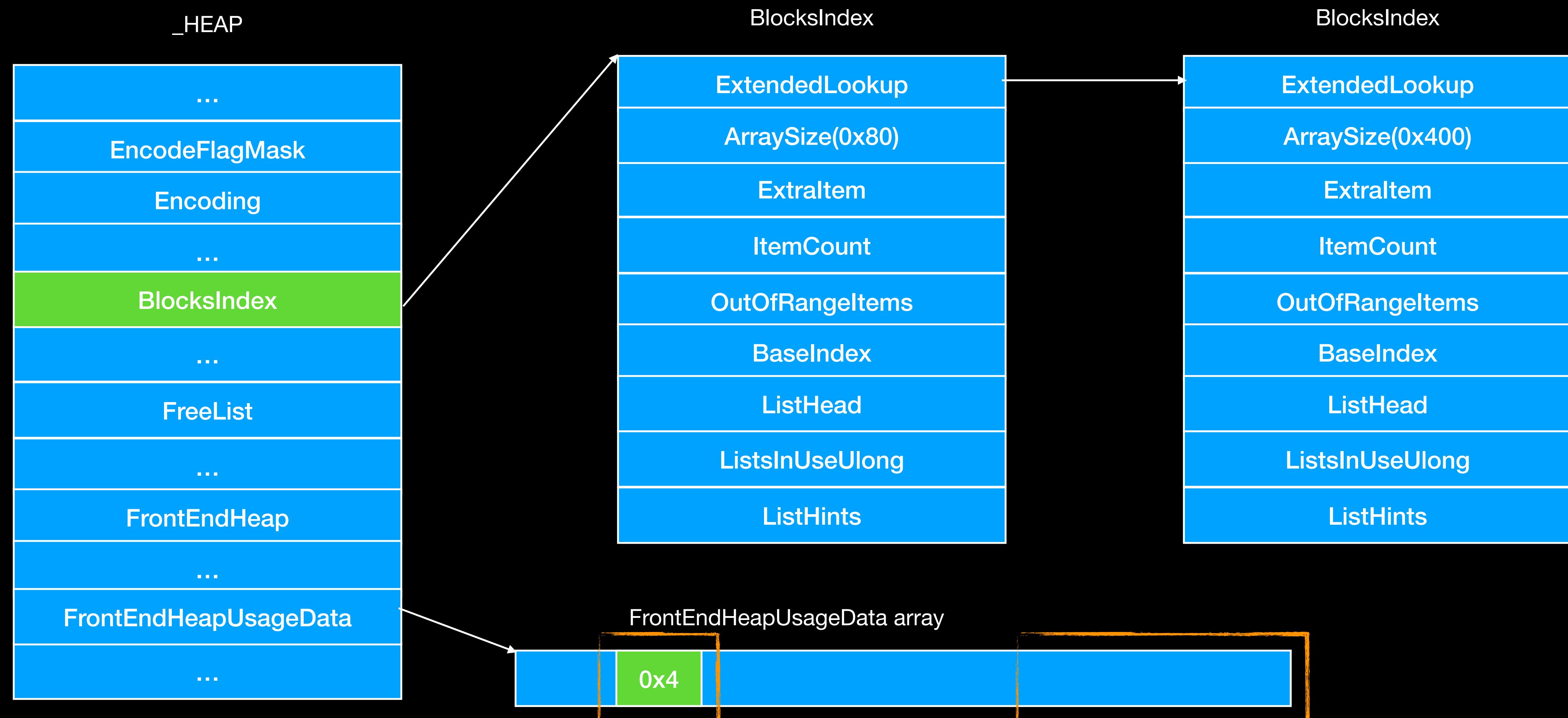
FrontEnd allocation mechanism

- LFH initialization
 - malloc(0x40) (18th)
 - ExtendFrontEndUsageData, add a larger BlocksIndex (0x80-0x400) and initialize corresponding bitmap
 - Write index on FrontEndUsageData[x]
 - initialize FrontEndHeap
 - Initialize SegmentInfoArrays[idx]
 - Write SegmentInfo to SegmentInfoArrays[idx]



Windows Memory Allocator

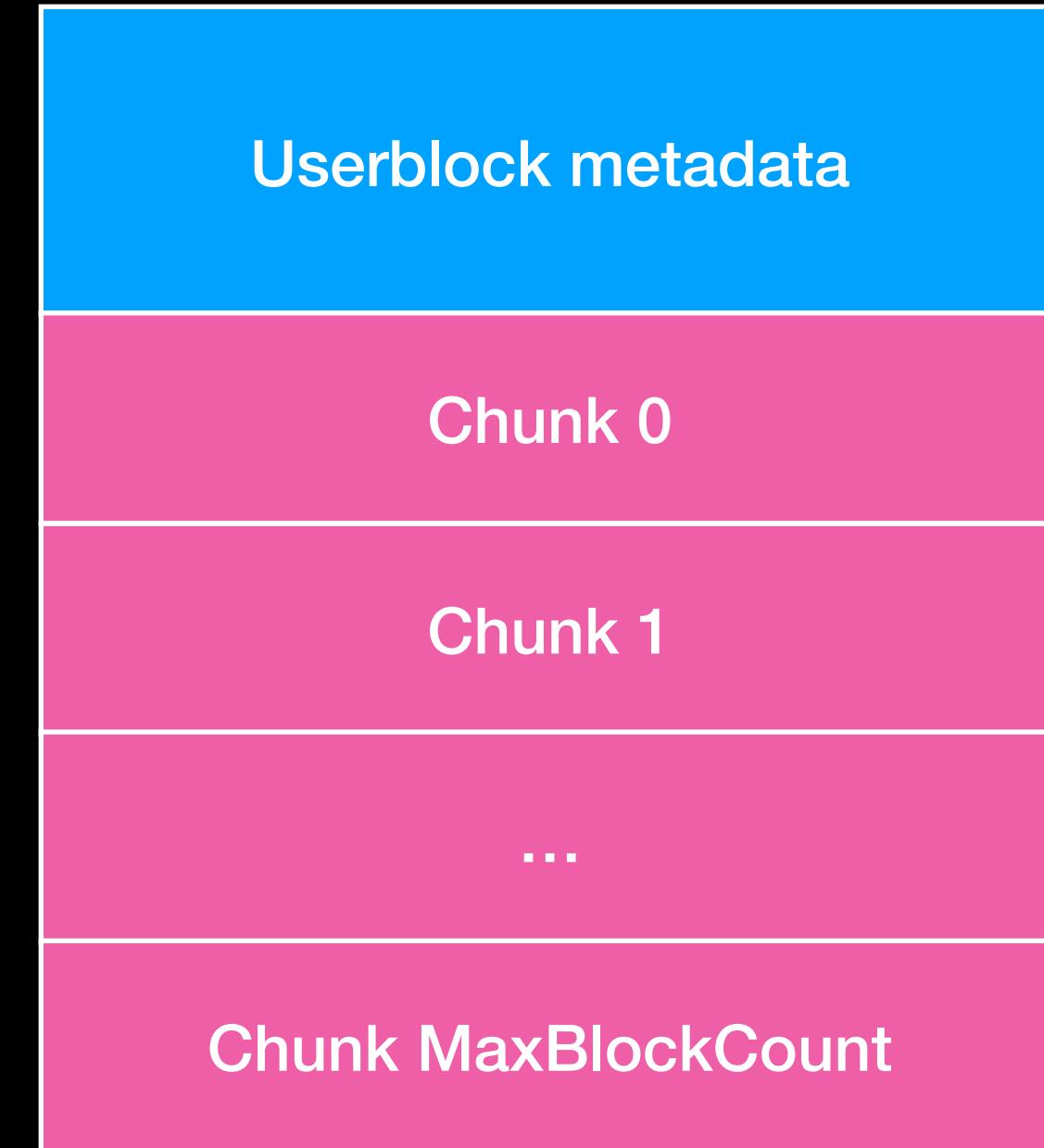
FrontEnd allocation mechanism



Windows Memory Allocator

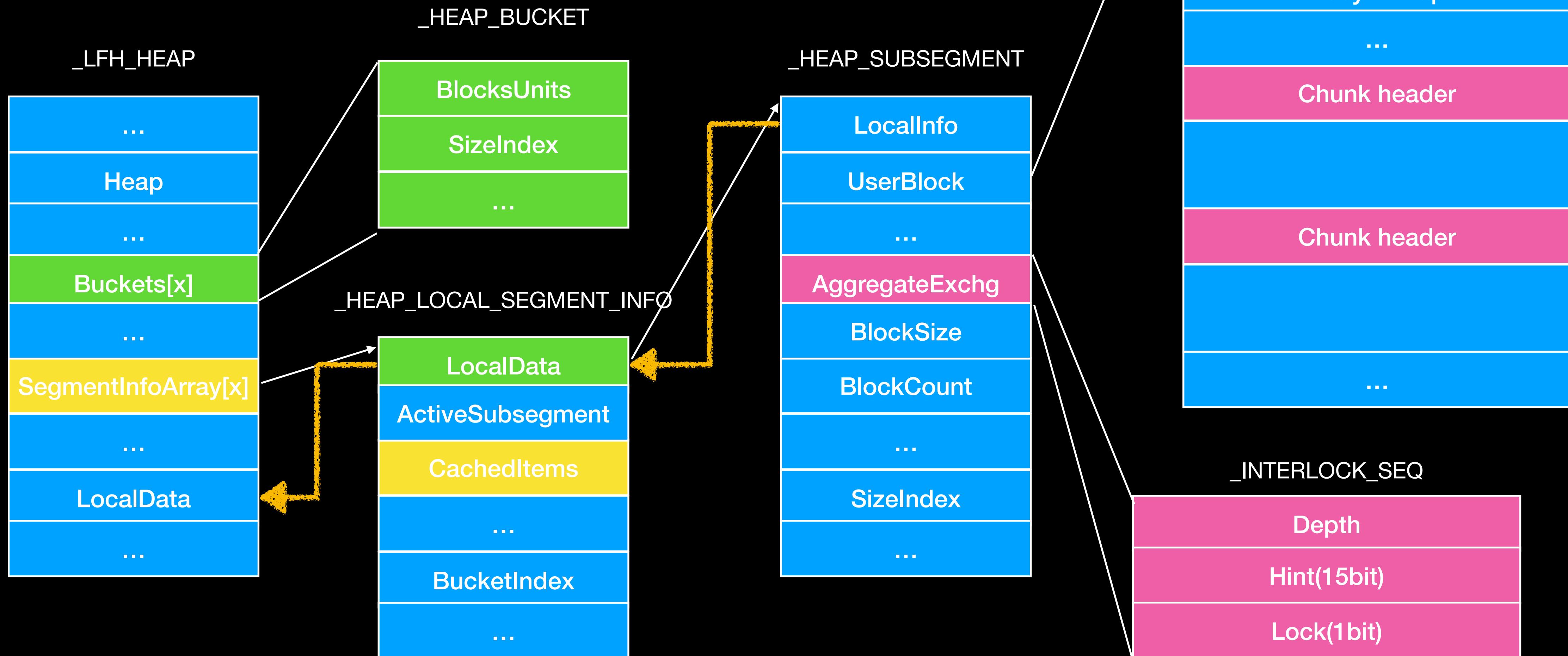
FrontEnd allocation mechanism

- LFH initialization
 - malloc(0x40) (19th)
 - Allocate & init Userblock
 - Config ActiveSubsegment
 - Return random chunk



Windows Memory Allocator

FrontEnd allocation mechanism



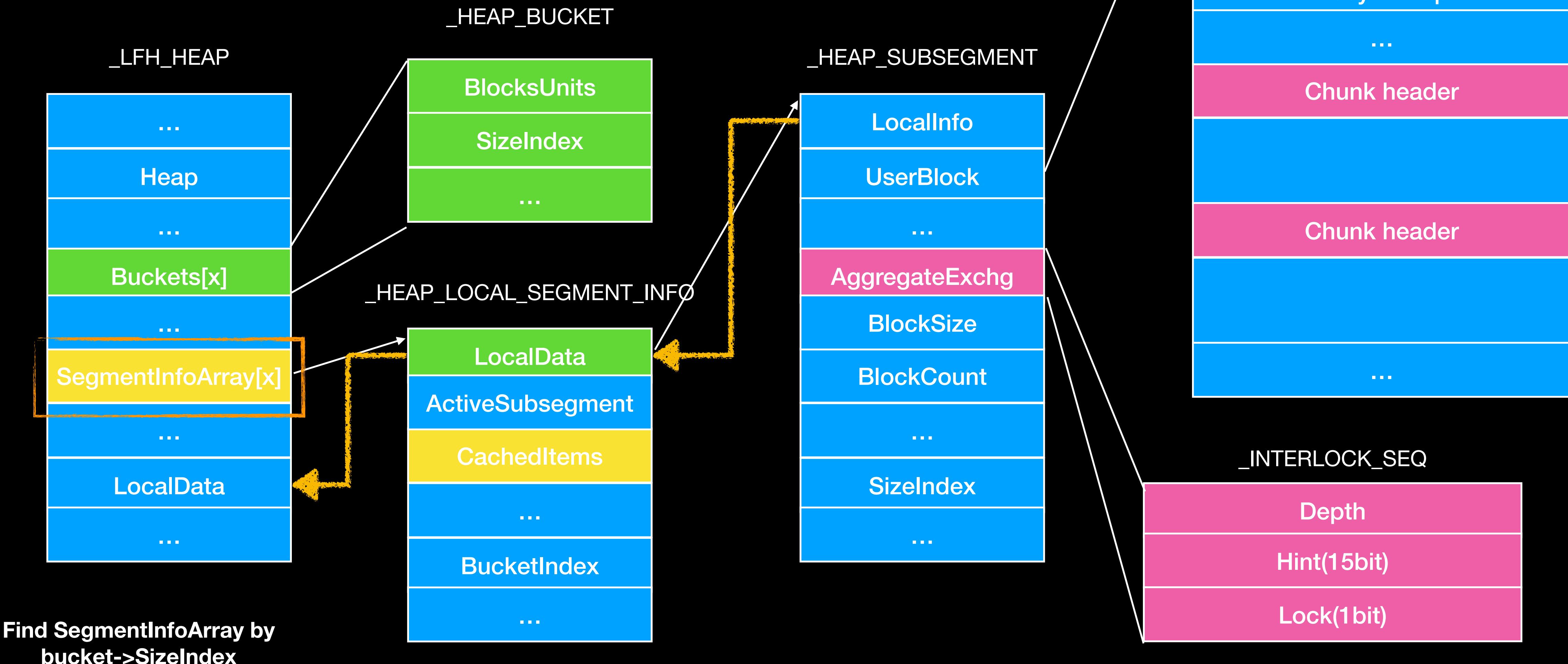
Windows Memory Allocator

FrontEnd allocation mechanism

- FrontEnd
 - Allocate (RtlpLowFragHeapAllocFromContext)
 - allocate chunk from ActiveSubsegment first
 - Check ActiveSubsegment->depth
 - allocate chunk from CachedItem
 - If found, replace the subsegment in the CachedItem with ActiveSubsegment

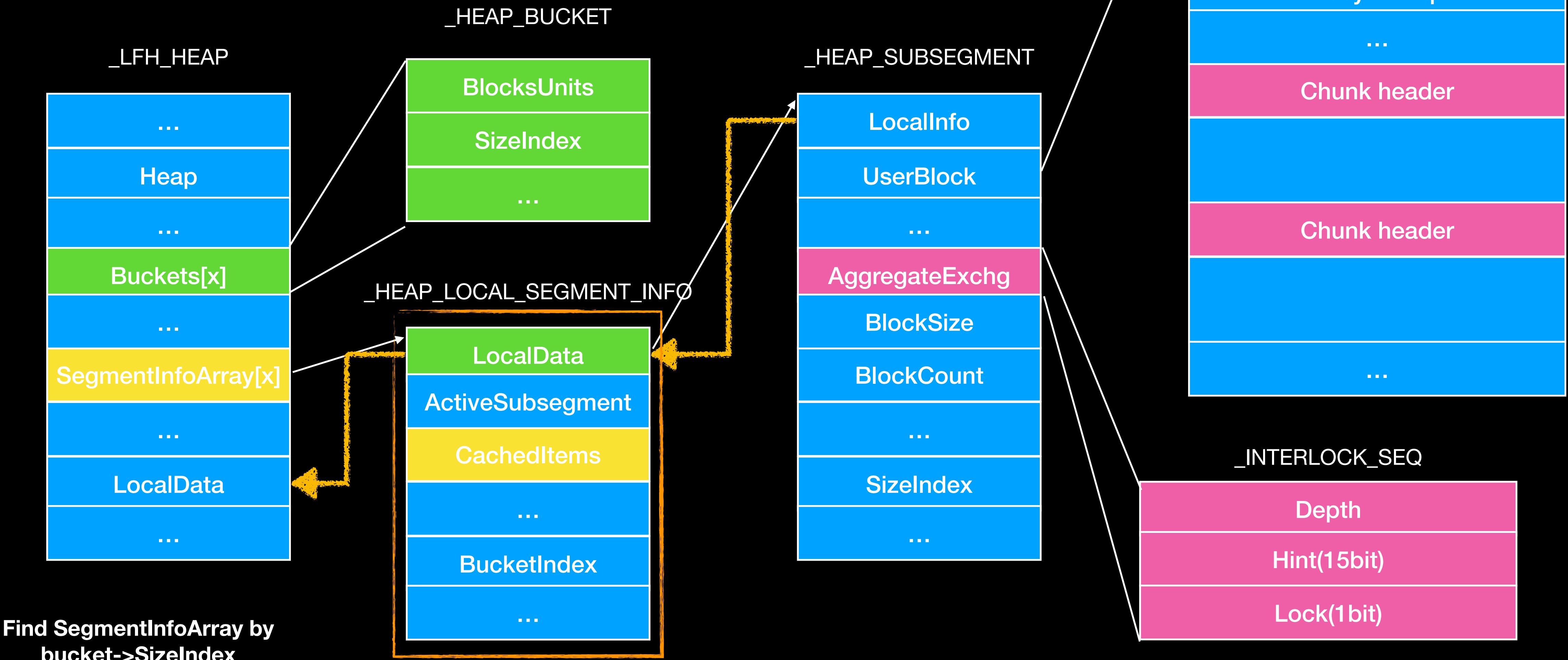
Windows Memory Allocator

FrontEnd allocation mechanism



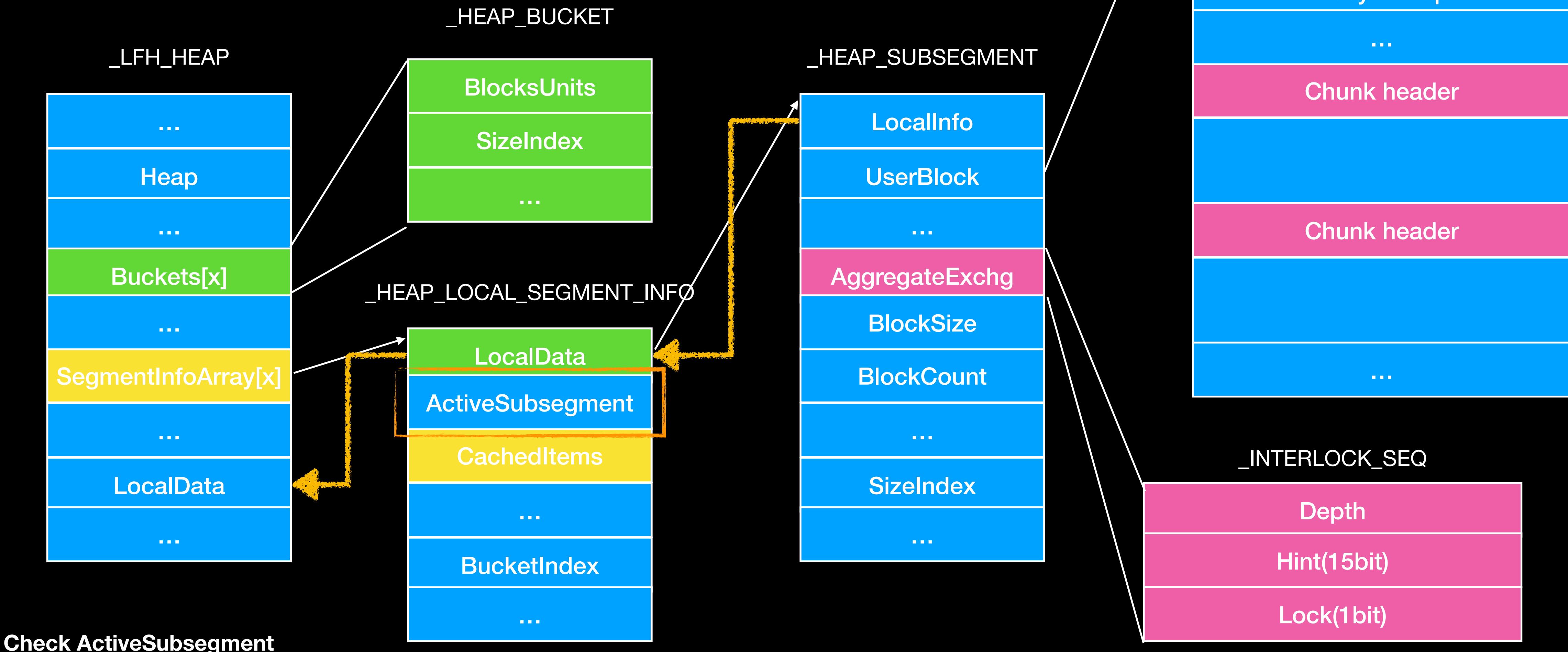
Windows Memory Allocator

FrontEnd allocation mechanism



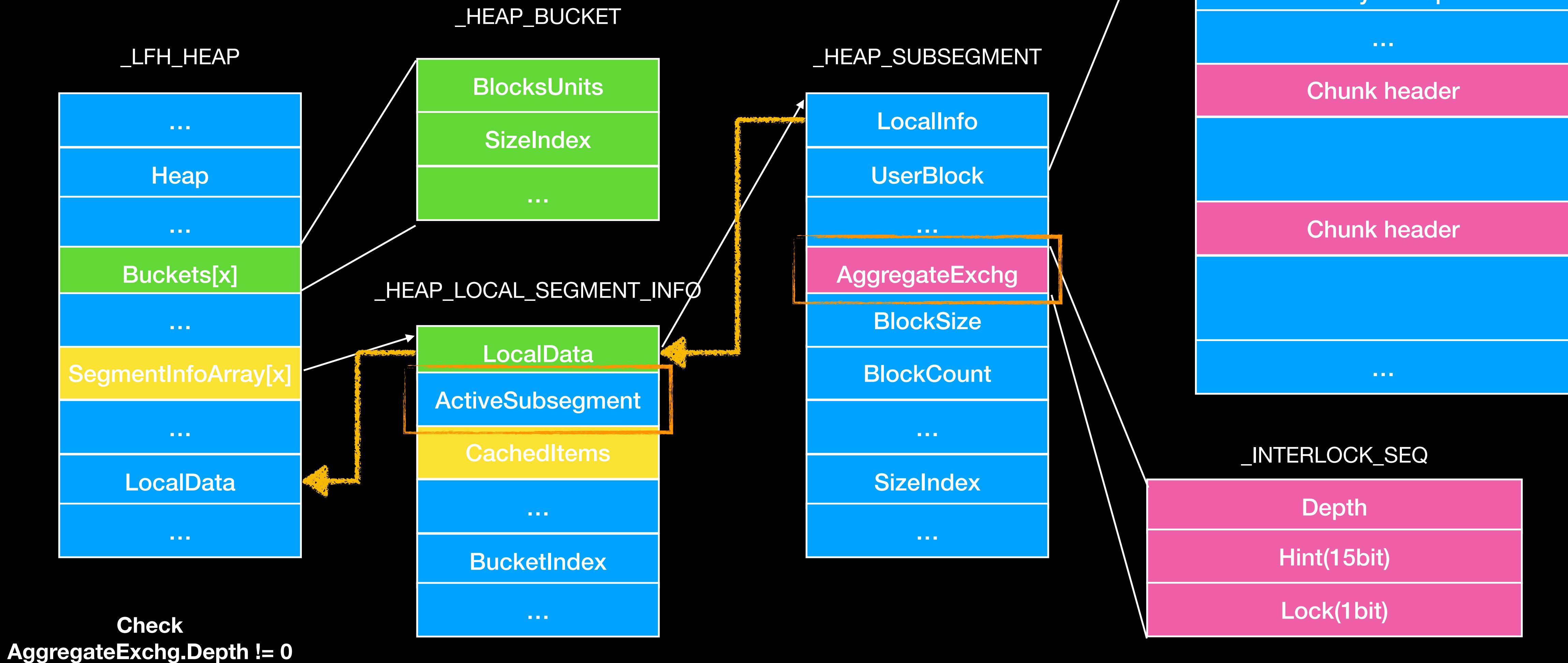
Windows Memory Allocator

FrontEnd allocation mechanism



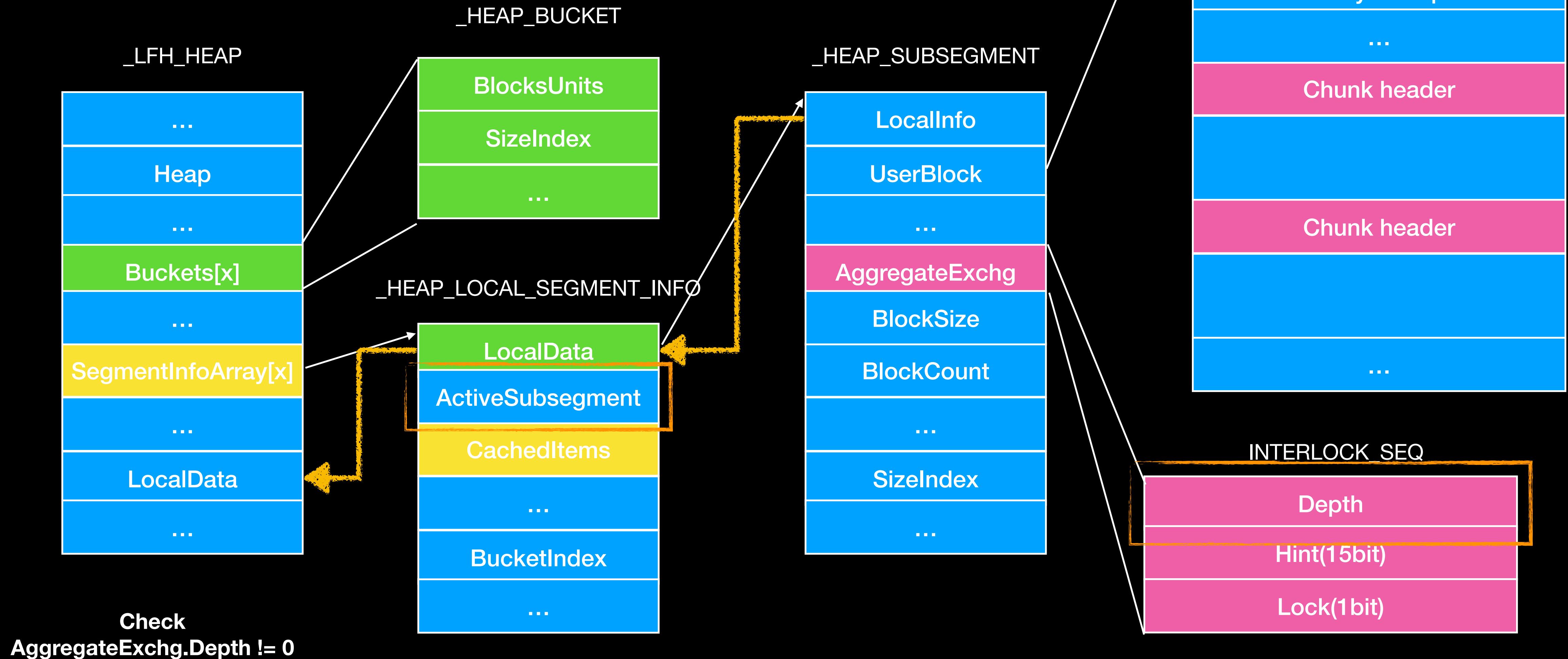
Windows Memory Allocator

FrontEnd allocation mechanism



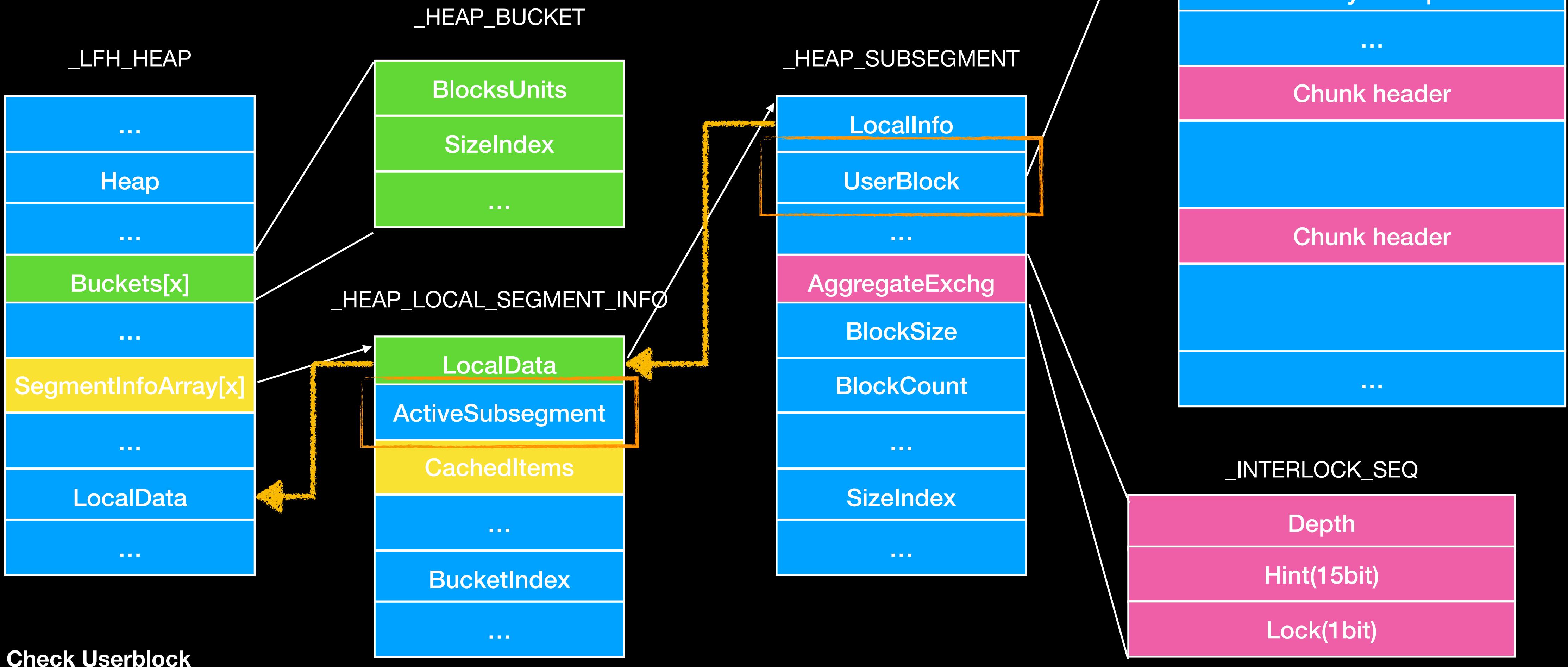
Windows Memory Allocator

FrontEnd allocation mechanism



Windows Memory Allocator

FrontEnd allocation mechanism



Windows Memory Allocator

FrontEnd allocation mechanism

- FrontEnd
 - Allocate (RtlpLowFragHeapAllocFromContext)
 - $x = \text{Teb} \rightarrow \text{LowFragHeapDataSlot}$
 - RtlpLowFragHeapRandomData len 256
 - Index = RtlpLowFragHeapRandomData[x]
 - Next index = RtlpLowFragHeapRandomData[x+1]
 - Next turn, Teb->LowFragHeapDataSlot = rand() % 256

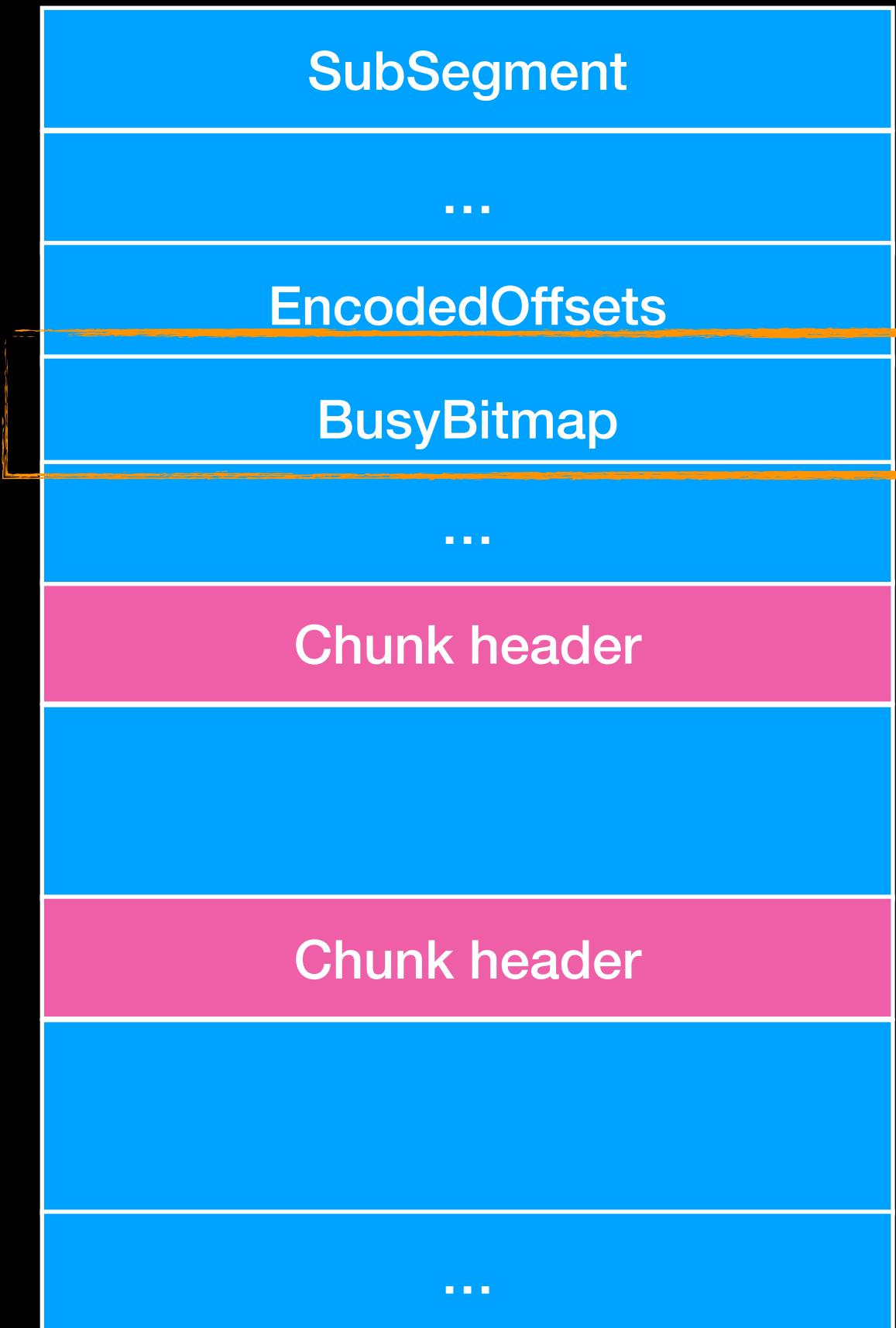
Windows Memory Allocator

FrontEnd allocation mechanism

- FrontEnd
 - Allocate (`RtlpLowFragHeapAllocFromContext`)
 - Final index = `index * maxidx >> 7`
 - If collision
 - Find nearest
 - Check (`used byte & 0x3f`) !=0 (indicating freed)
 - Set index and unused byte then return chunk to user

Windows Memory Allocator

- Get an index
 - $x = \text{RtlpLowFragHeapRandomData}[index] * \text{maxidx} >> 7$
 - Check BusyBitmap[x]
 - If 0, take chunk
 - If 1, take next chunk
 - Config bitmap



Windows Memory Allocator

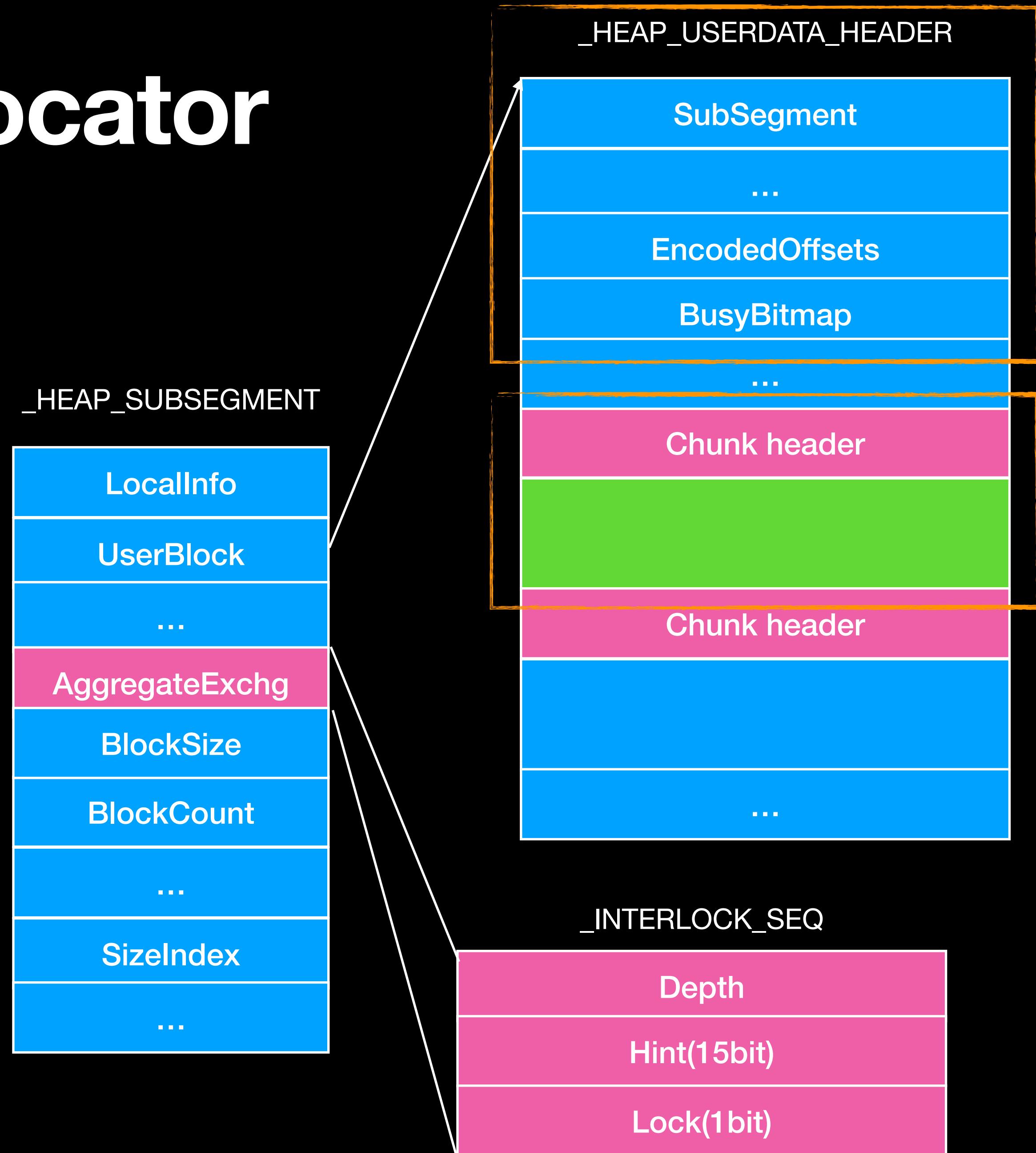
FrontEnd allocation mechanism

- FrontEnd
 - Free (RtlFreeHeap)
 - update unused byte in chunk header
 - Find the index of chunk and reset UserBlock->BusyBitmap
 - Update ActiveSubsegment->AggregateExchg
 - If freed chunk not belongs to current ActiveSubsegment, try cachedItems

Windows Memory Allocator

FrontEnd allocation mechanism

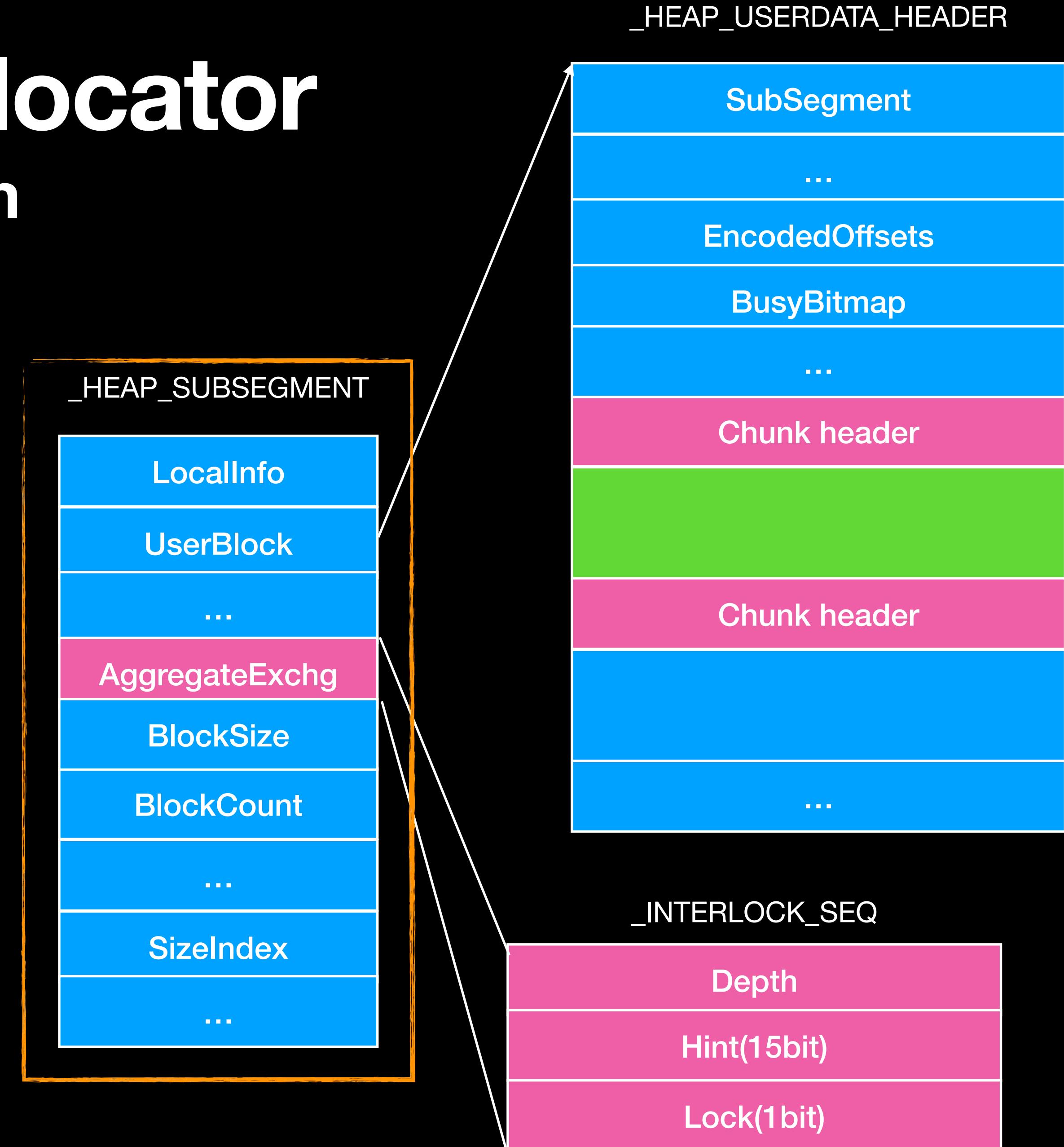
- Free
 - Chunk header->UserBlock



Windows Memory Allocator

FrontEnd allocation mechanism

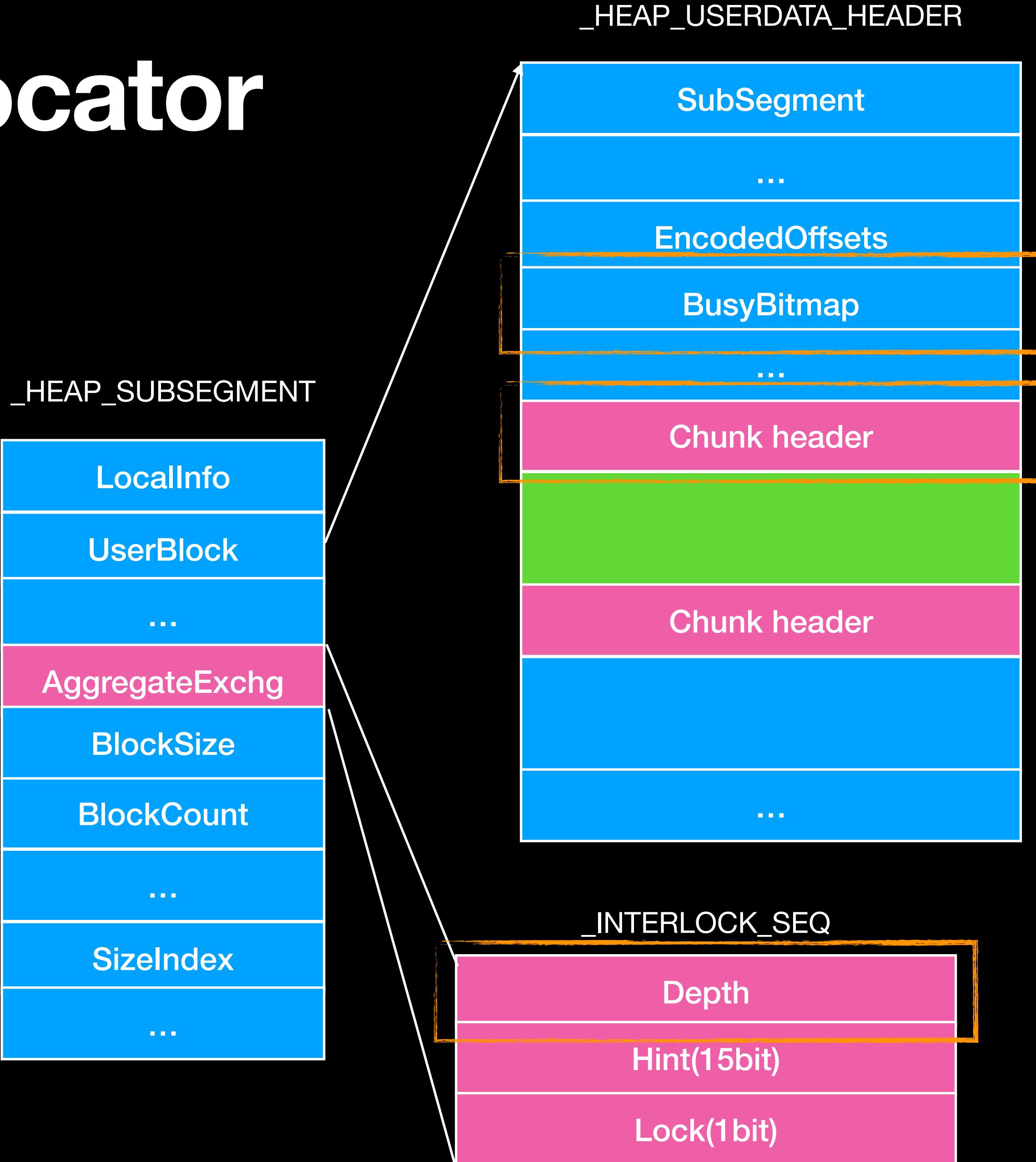
- Free
 - Chunk header->UserBlock
 - SubSegment



Windows Memory Allocator

FrontEnd allocation mechanism

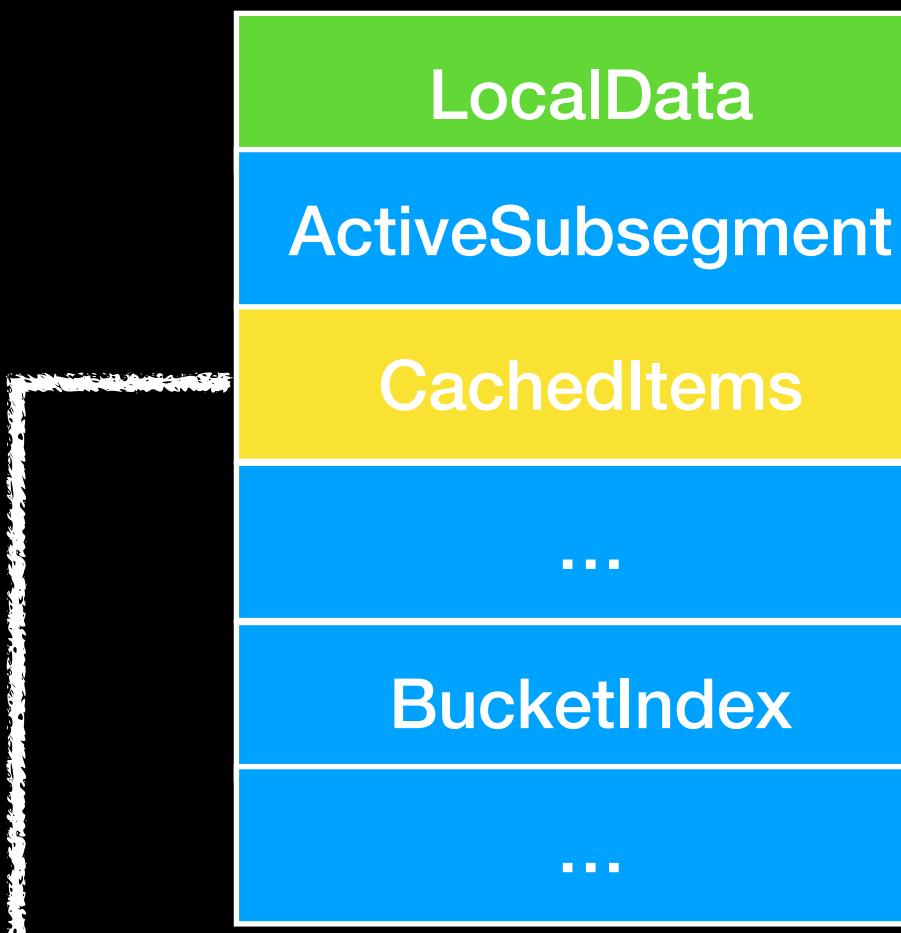
- Free
 - Chunk header->UserBlock
 - SubSegment
 - Set unused byte = 0x80
 - Clear bitmap
 - Update AggregateExchg



Windows Memory Allocator

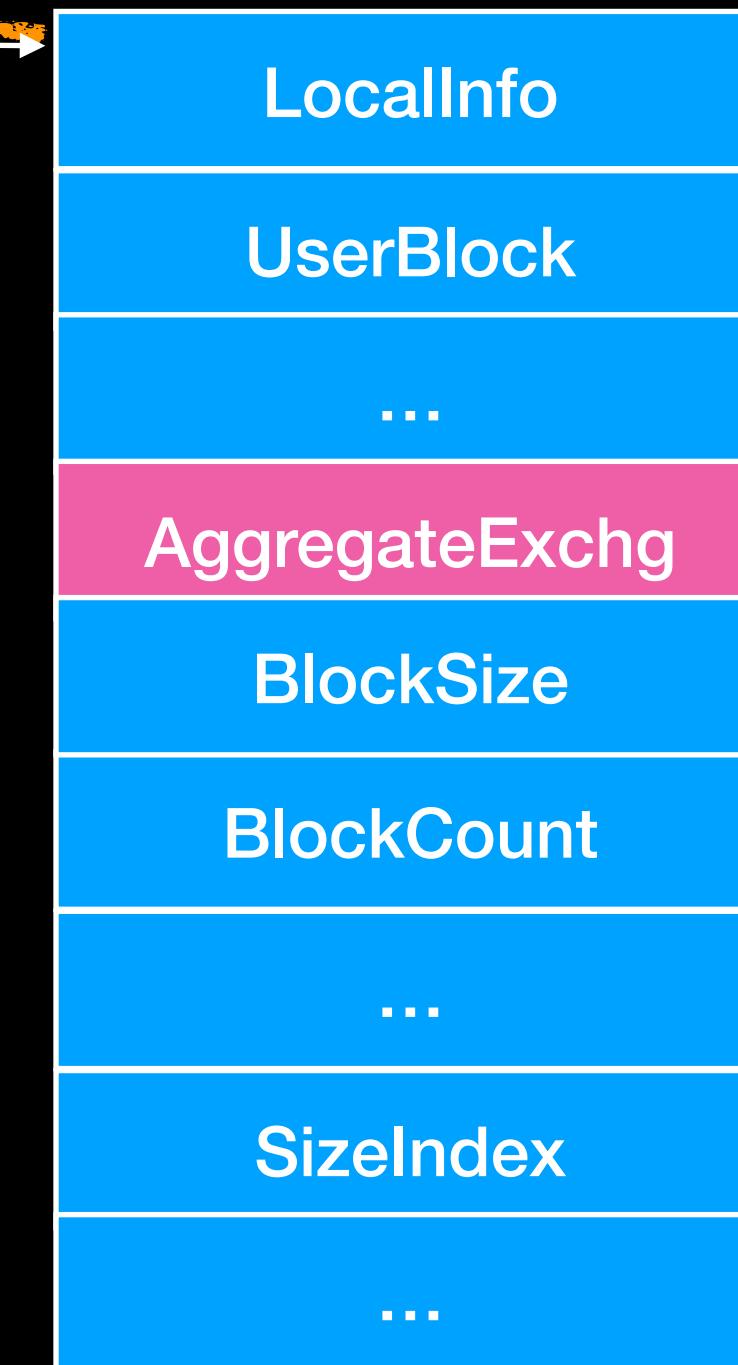
FrontEnd allocation mechanism

_HEAP_LOCAL_SEGMENT_INFO

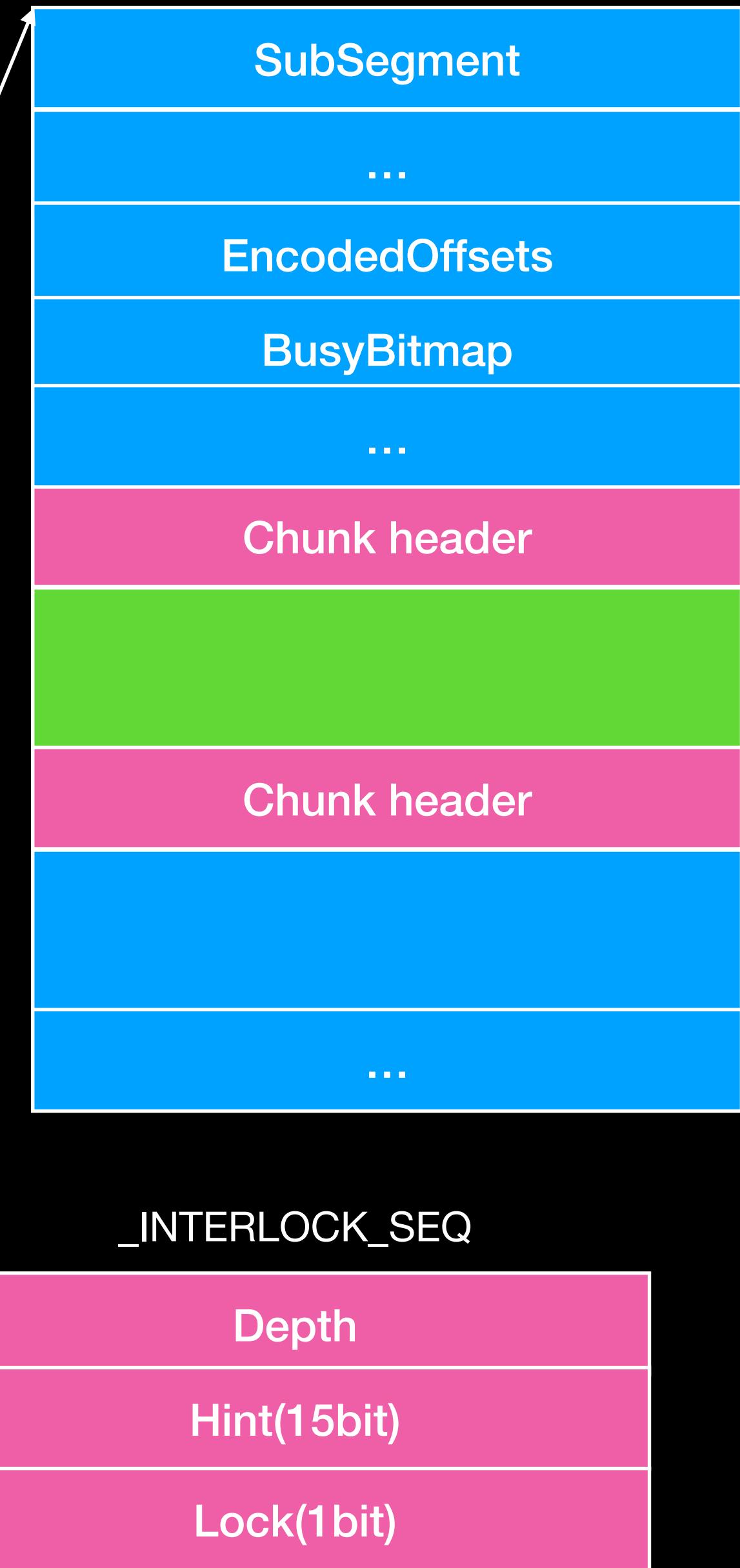


ActiveSubSegment != freed chunk's SubSegment

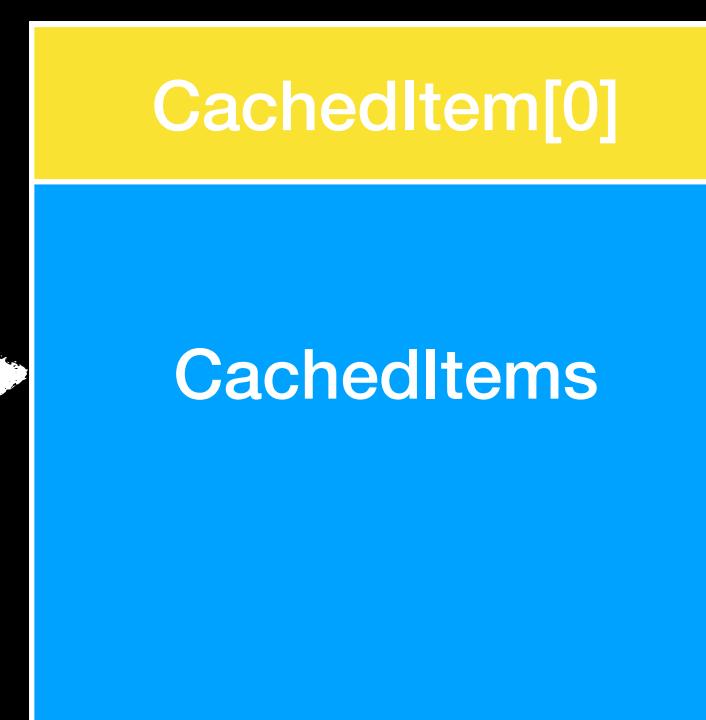
_HEAP_SUBSEGMENT



_HEAP_USERDATA_HEADER



CachedItems



LFH Exploitation

- Reuse attack
- indicate that we have UAF first
 - Due to LFH's its randomness, we can not simply exploit memory by alloc/free
 - We can fill a UserBlock and free specific one, we can get the same LFH chunk at next allocation
 - First UserBlock has 32 chunks, Second has 64...

LFH Exploitation

- Reuse attack

```
C:\Users\2st\Desktop\HeapCode\x64\Release\HeapCode.exe
begin to alloc

1st back alloc 0 at 000001F531700860
1st back alloc 1 at 000001F5317008D0
1st back alloc 2 at 000001F531700940
1st back alloc 3 at 000001F5317009B0
1st back alloc 4 at 000001F531700A20
1st back alloc 5 at 000001F531700A90
1st back alloc 6 at 000001F531700B00
1st back alloc 7 at 000001F531700B70
1st back alloc 8 at 000001F531700BE0
1st back alloc 9 at 000001F531700C50
1st back alloc 10 at 000001F531700CC0
1st back alloc 11 at 000001F531700D30
1st back alloc 12 at 000001F531700DA0
1st back alloc 13 at 000001F531700E10
1st back alloc 14 at 000001F531700E80
1st back alloc 15 at 000001F531700EF0
1st back alloc 16 at 000001F531700F60
1st back alloc 17 at 000001F531700750
LFH activied!
```

```
1st lfh alloc 0 at 000001F531703A20
1st lfh alloc 1 at 000001F531703A90
1st lfh alloc 2 at 000001F5317037F0
1st lfh alloc 3 at 000001F531703630
1st lfh alloc 4 at 000001F531703710
1st lfh alloc 5 at 000001F5317038D0
1st lfh alloc 6 at 000001F531703780
1st lfh alloc 7 at 000001F5317036A0
1st lfh alloc 8 at 000001F531703940
1st lfh alloc 9 at 000001F5317035C0
1st lfh alloc 10 at 000001F5317034E0
1st lfh alloc 11 at 000001F5317039B0
1st lfh alloc 12 at 000001F531703B70
1st lfh alloc 13 at 000001F531703860
1st lfh alloc 14 at 000001F531703B00
1st lfh alloc 15 at 000001F531703BE0
1st lfh alloc 16 at 000001F531703550
1st lfh alloc 17 at 000001F531704780
1st lfh alloc 18 at 000001F5317041D0
1st lfh alloc 19 at 000001F5317040F0
1st lfh alloc 20 at 000001F5317047F0
1st lfh alloc 21 at 000001F5317044E0
1st lfh alloc 22 at 000001F531704240
1st lfh alloc 23 at 000001F531704E80
1st lfh alloc 24 at 000001F531704160
1st lfh alloc 25 at 000001F531704320
1st lfh alloc 26 at 000001F531704DA0
1st lfh alloc 27 at 000001F531704B00
1st lfh alloc 28 at 000001F531704B70
1st lfh alloc 29 at 000001F531704550
last alloc at 000001F5317049B0
attack alloc at 000001F5317049B0

attack alloc at 000001F5317049B0
```

```
1st lfh alloc 28 at 000001F531704B70
1st lfh alloc 29 at 000001F531704550
last alloc at 000001F5317049B0
attack alloc at 000001F5317049B0

attack alloc at 000001F5317049B0
```

Thanks