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

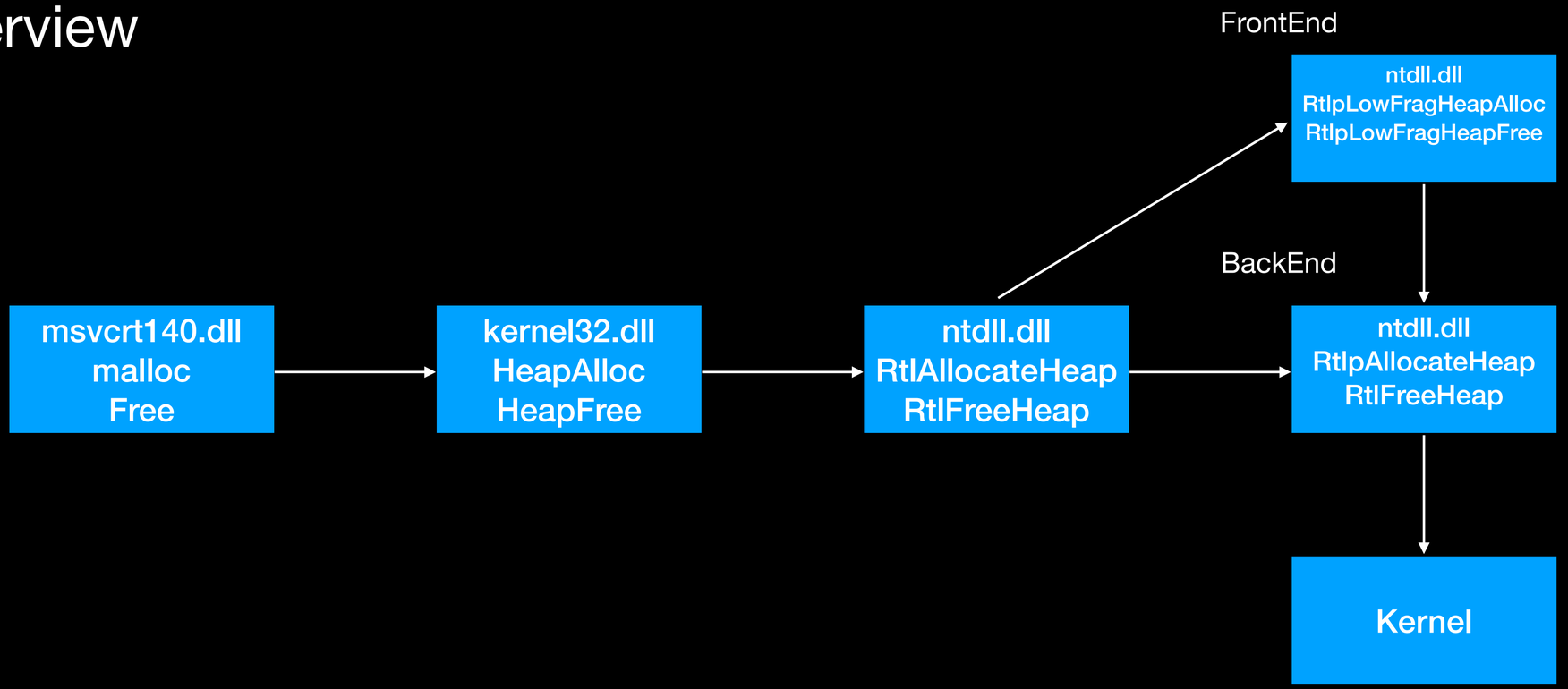
- Segment Heap
  - Used in Windows UWP programs

Windows Memory Allocator

- Tested on Windows 10 1803 64-bit

Windows Memory Allocator

- Overview



Windows Memory Allocator

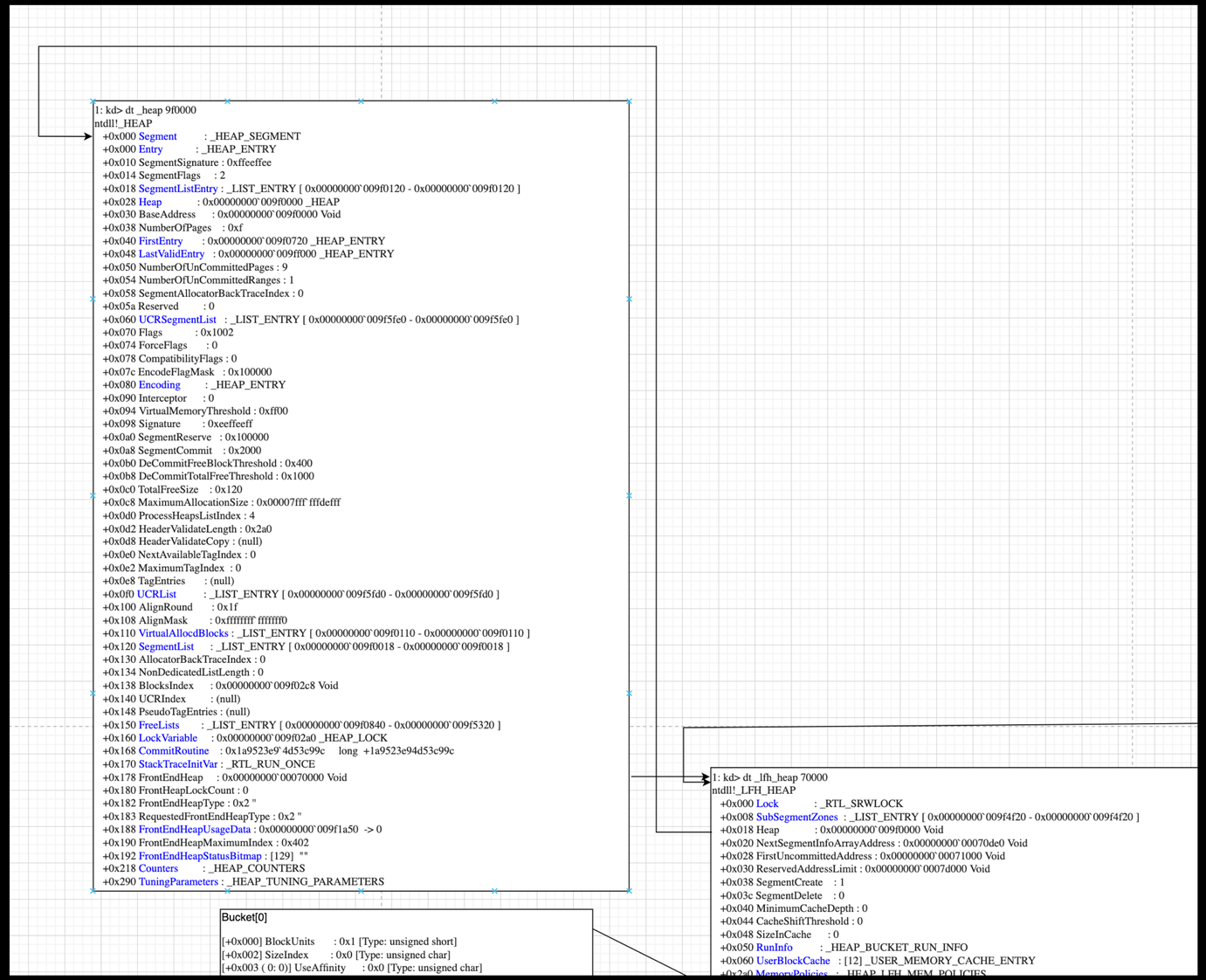
- Heap types
  - Default heap
    - Shared in process, stored in _PEB
  - CRT heap
    - malloc / new
  - Private heap
    - HeapCreate

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

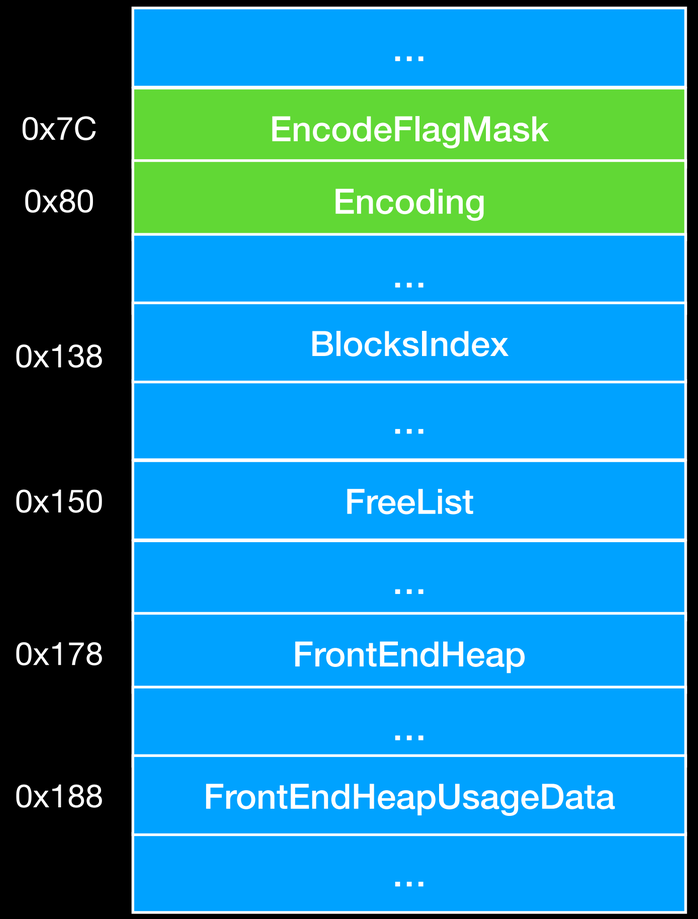
Windows Memory Allocator

- _HEAP overview



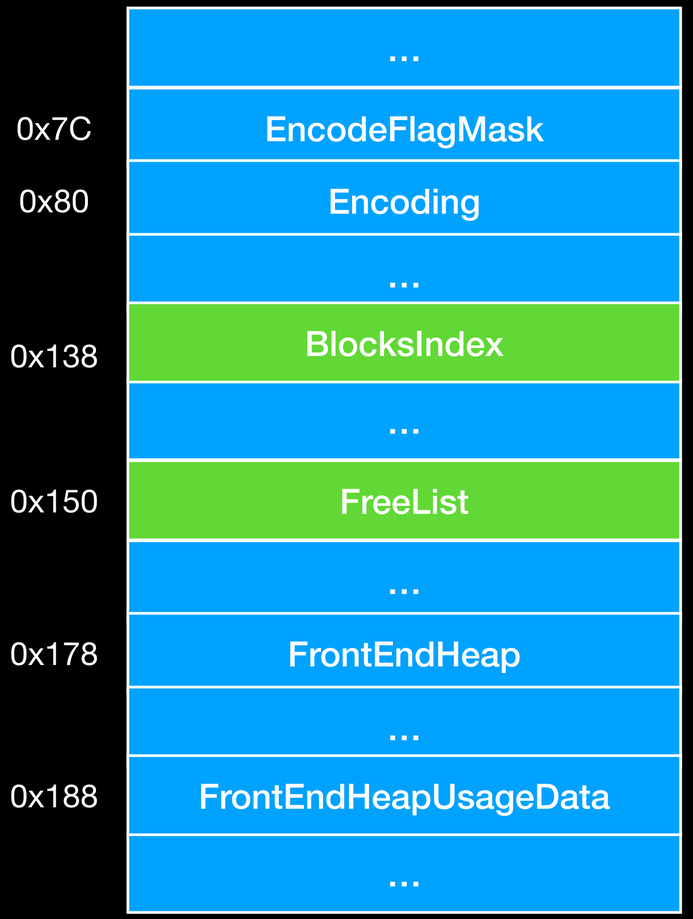
Windows Memory Allocator

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



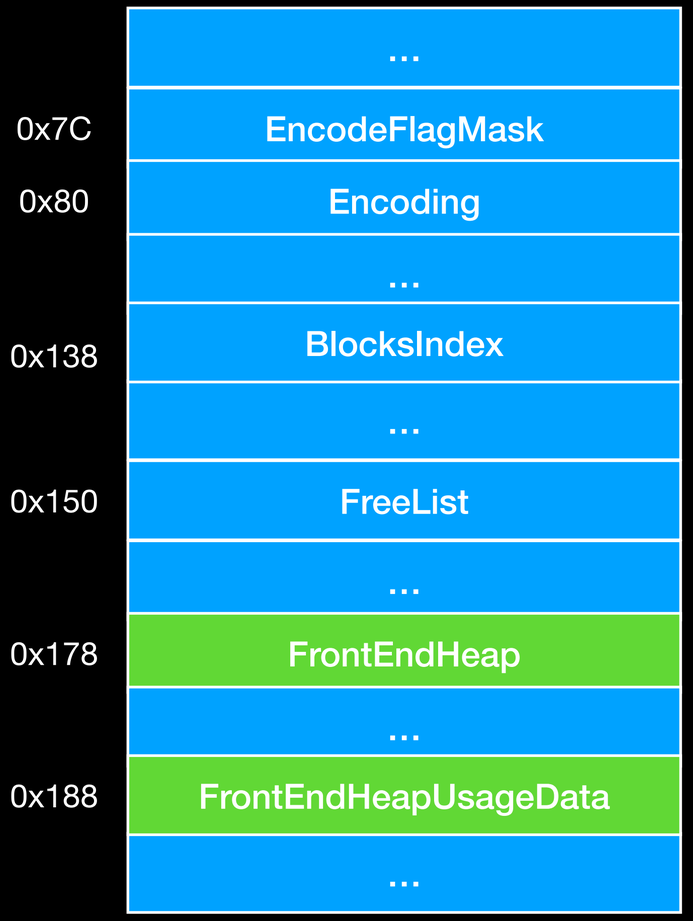
Windows Memory Allocator

- _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

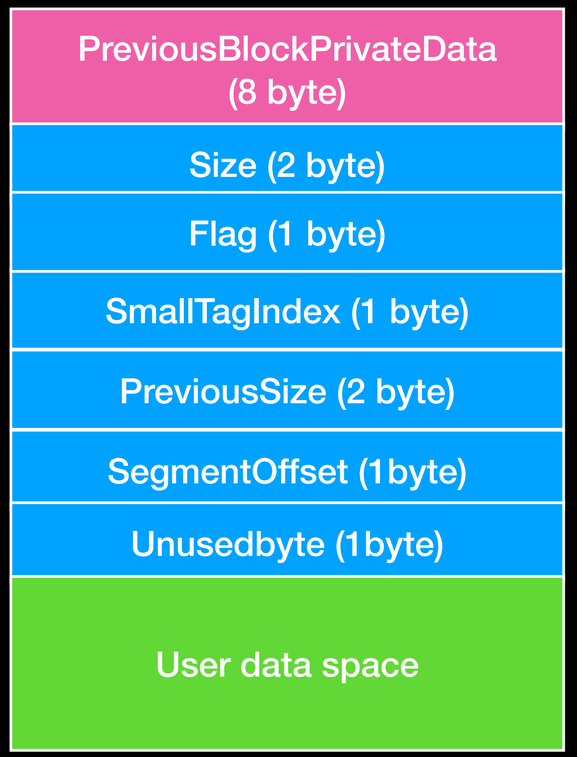
- NT heap
- Common data structure
  - BackEnd
    - Allocation mechanism
    - BackEnd heap exploitation
  - FrontEnd
    - Allocation mechanism
    - FrontEnd heap exploitation

Windows Memory Allocator

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

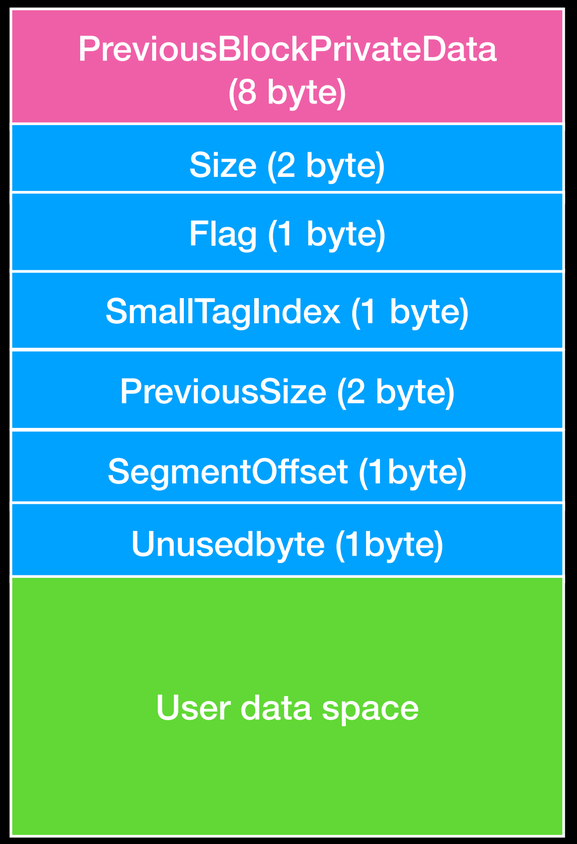
Windows Memory Allocator
in used chunk

- PreviousBlockPrivateData
  - previous chunk data
- Size
  - value = real size >> 4
- Flag
  - if inused
- SmallTagIndex
  - value = (size & 0xff) ^ (size >> 8) ^ 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

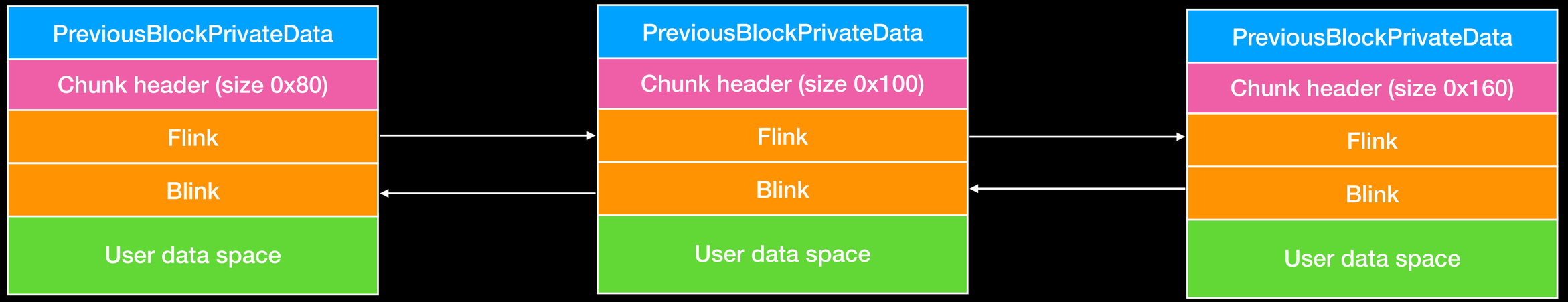
Windows Memory Allocator
in used mmap chunk

- 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: encode_header = header ^ (_HEAP->Encoding)
  - Decode check
    - val = encode_header^(_HEAP->Encoding)
    - val[3] = val[0] ^ val[1] ^ val[2] ?

Windows Memory Allocator

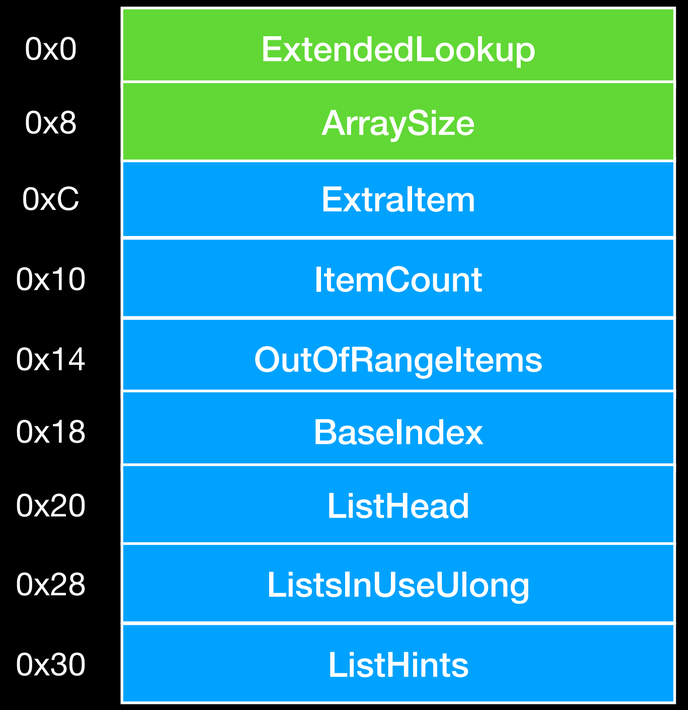
BlocksIndex

- manage various size freed chunks
  - Fast lookup

Windows Memory Allocator

BlocksIndex

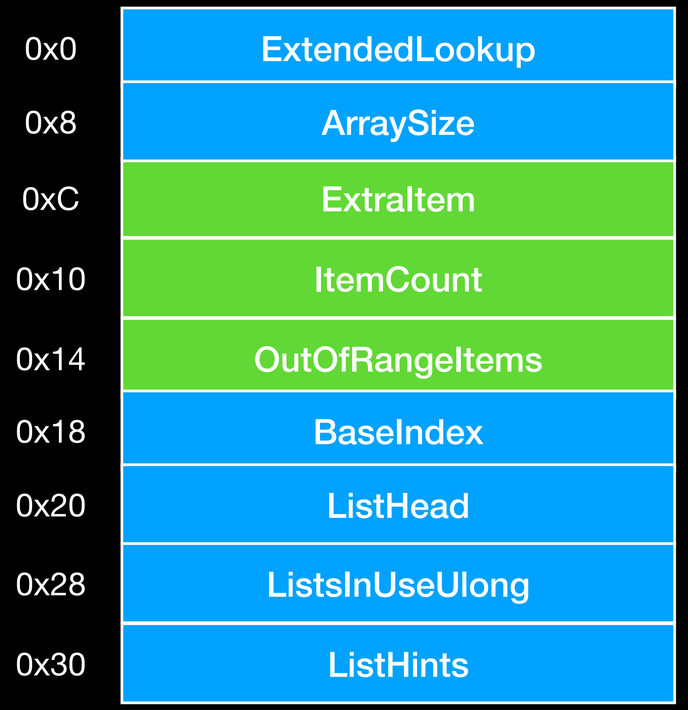
- _HEAP_LIST_LOOKUP
- ExtendedLookup(_HEAP_LIST_LOOKUP)
  - point to next ExtendedLookup
  - Usually, next ExtendedLookup manage bigger chunk
- ArraySize
  - Max chunk size in current lookup
  - Usually 0x80



Windows Memory Allocator

BlocksIndex

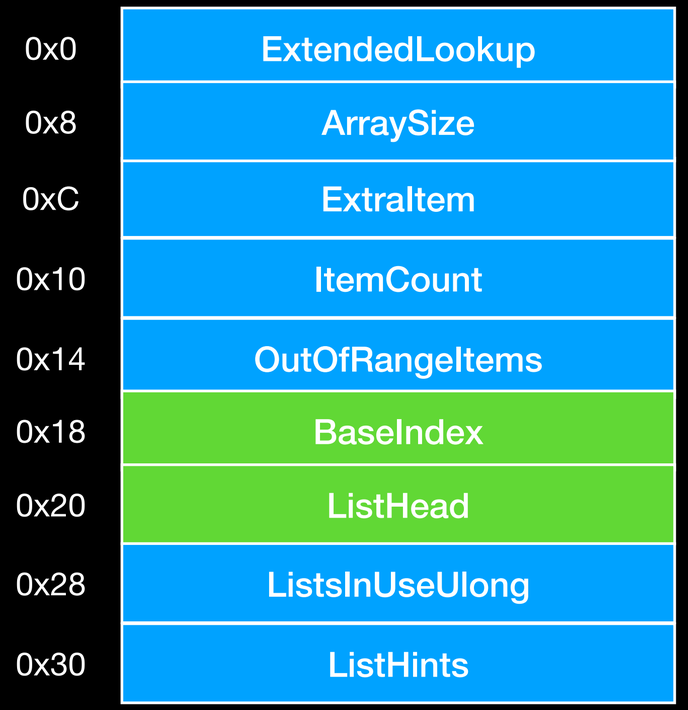
- ExtraItem
  - Indicate whether each member in the ListHints array is composed of one pointer or two pointers
- ItemCount
  - Number of chunks managed by the current structure
- OutOfRangeItems
  - Number of chunks exceeding the current structure management size



Windows Memory Allocator

BlocksIndex

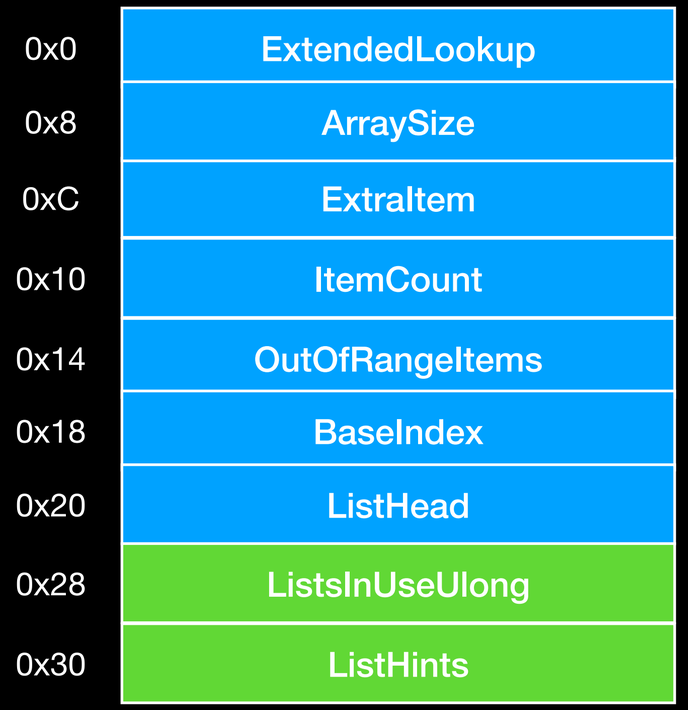
- BaseIndex
  - The starting index of the chunk managed by this structure is used to find a free chunk of suitable size from ListHint, and the next BlocksIndex will be the maximum value of this index to the Base Index
- ListHead
  - FreeList Head



Windows Memory Allocator

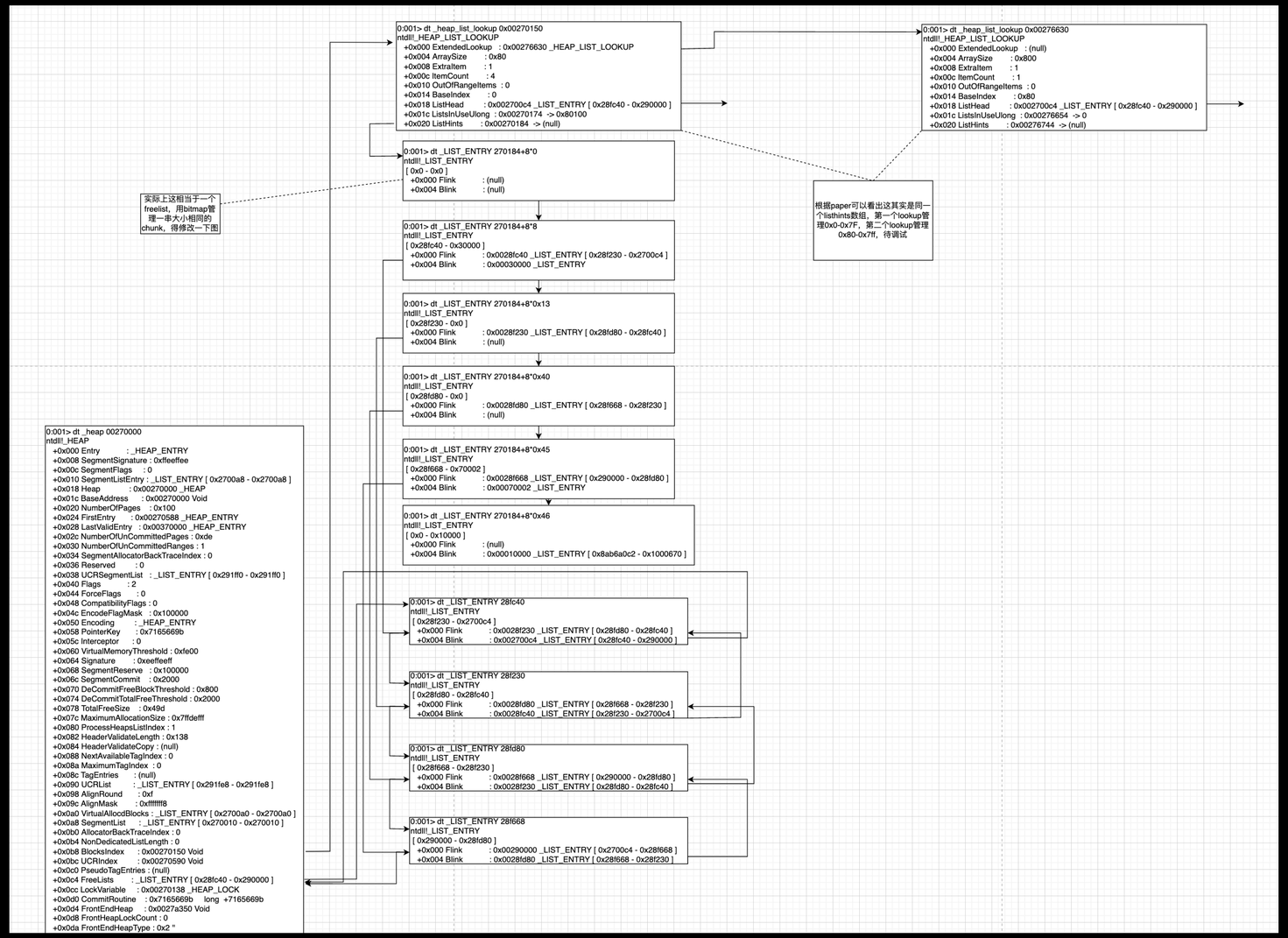
BlocksIndex

- 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 < Size <= 0xff000
- same operations as size <= 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 & pervious 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

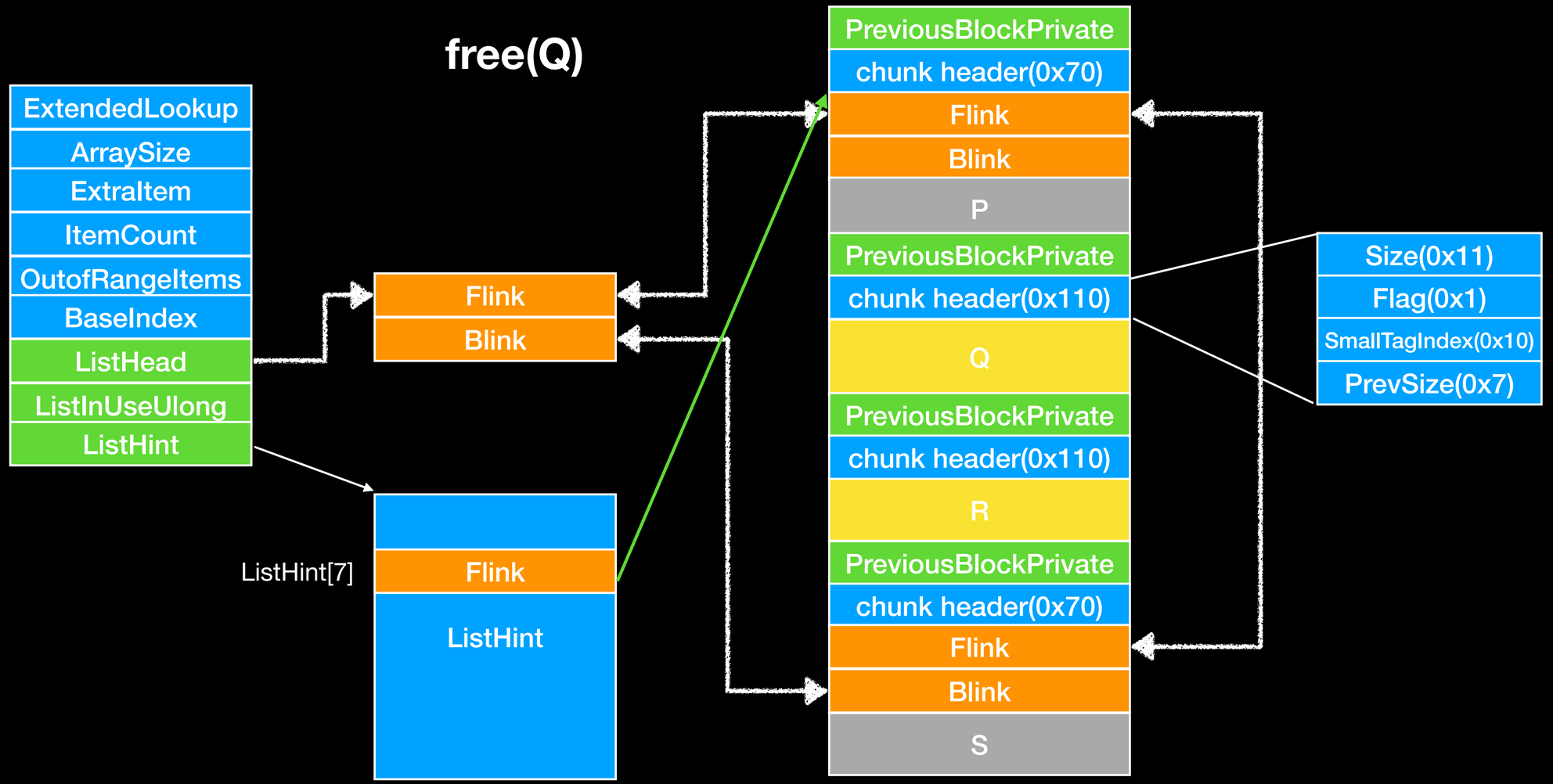
Windows Memory Allocator

BackEnd allocation mechanism

- 0xff000 < Size
  - check chunk’s linked list and remove it from _HEAP->VirtualAllocdBlocks
  - unmap this chunk by RltpSecMemFreeVirtualMemory

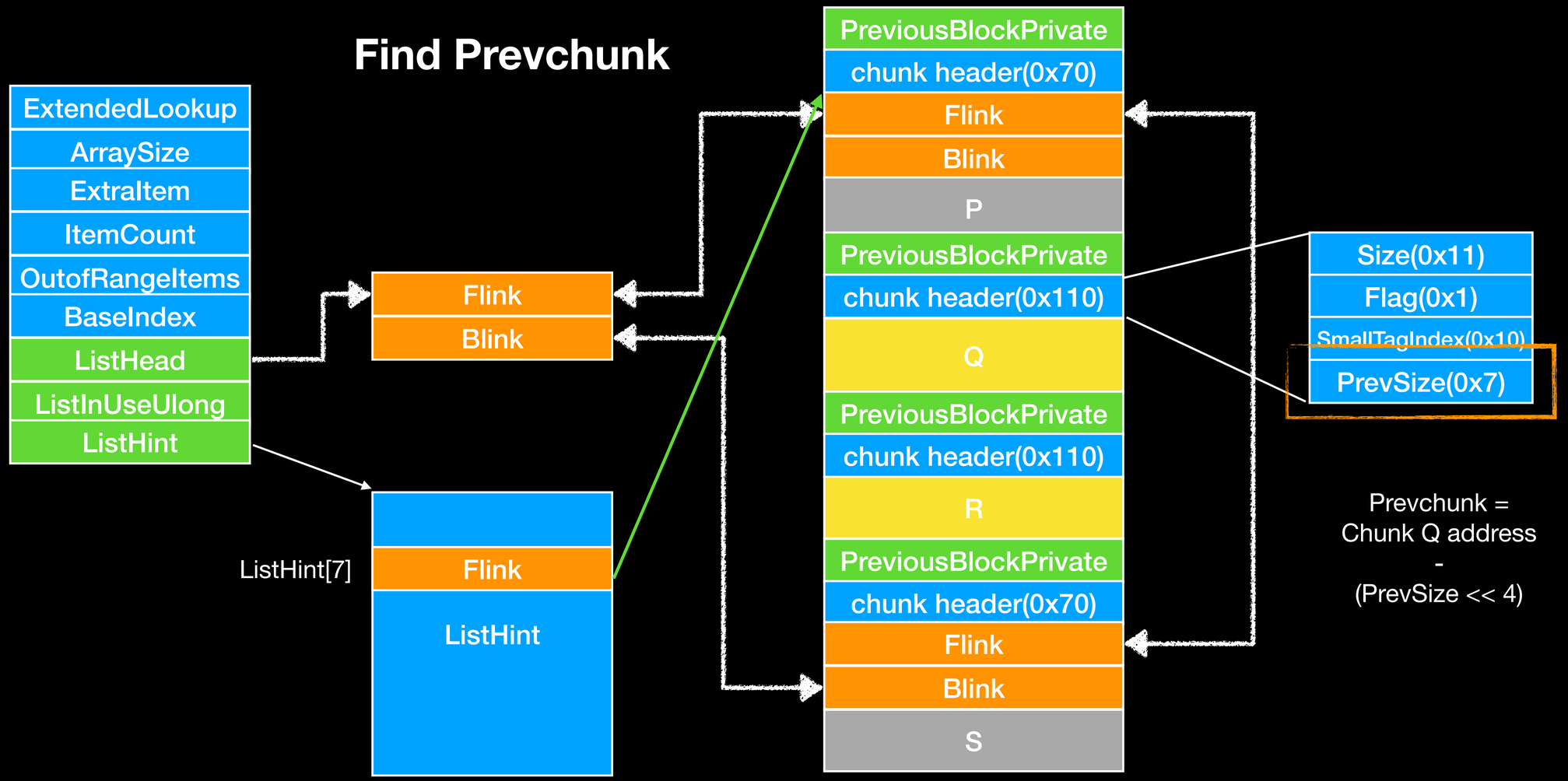
Windows Memory Allocator

merge chunk



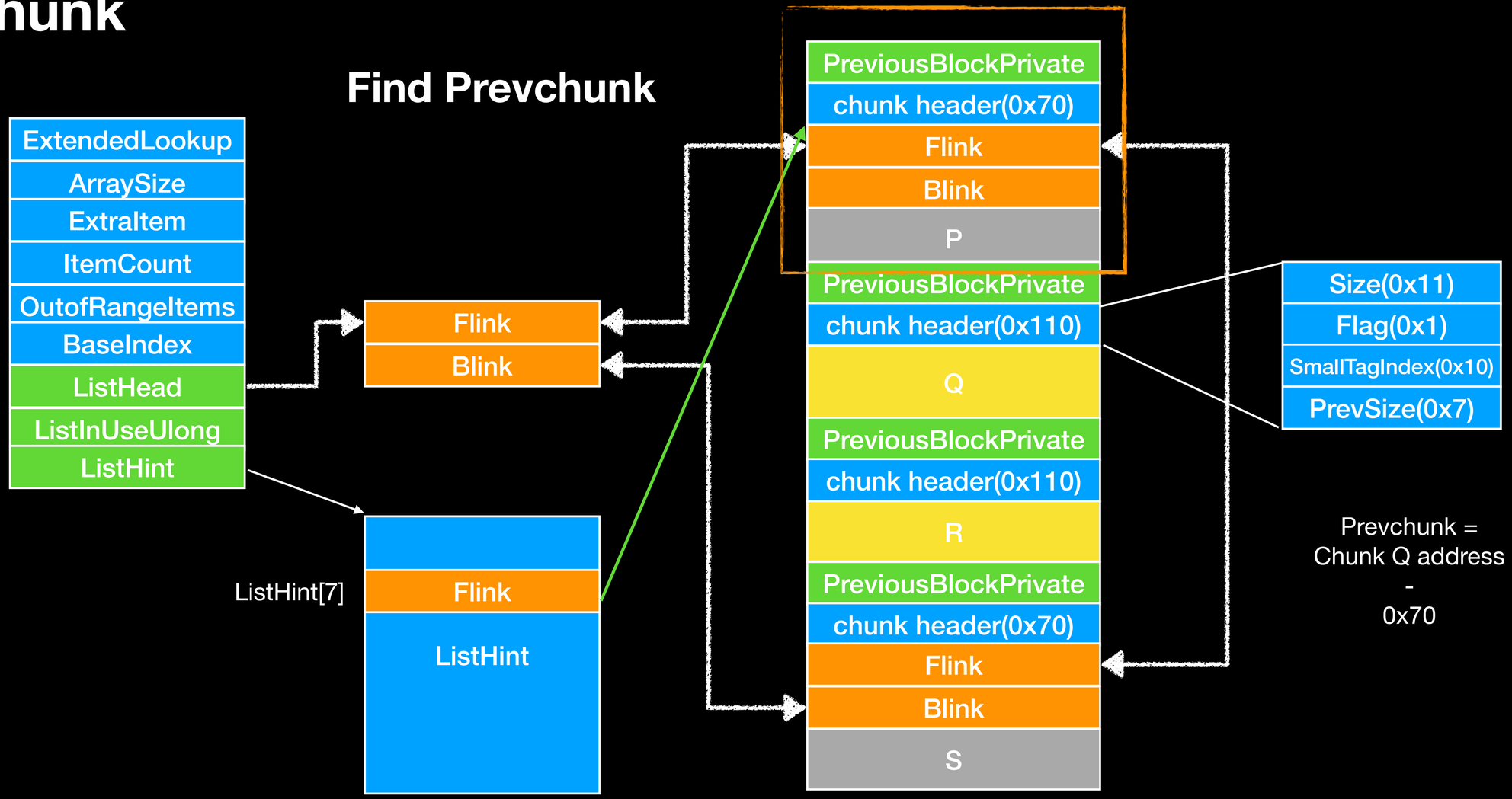
Windows Memory Allocator

merge chunk



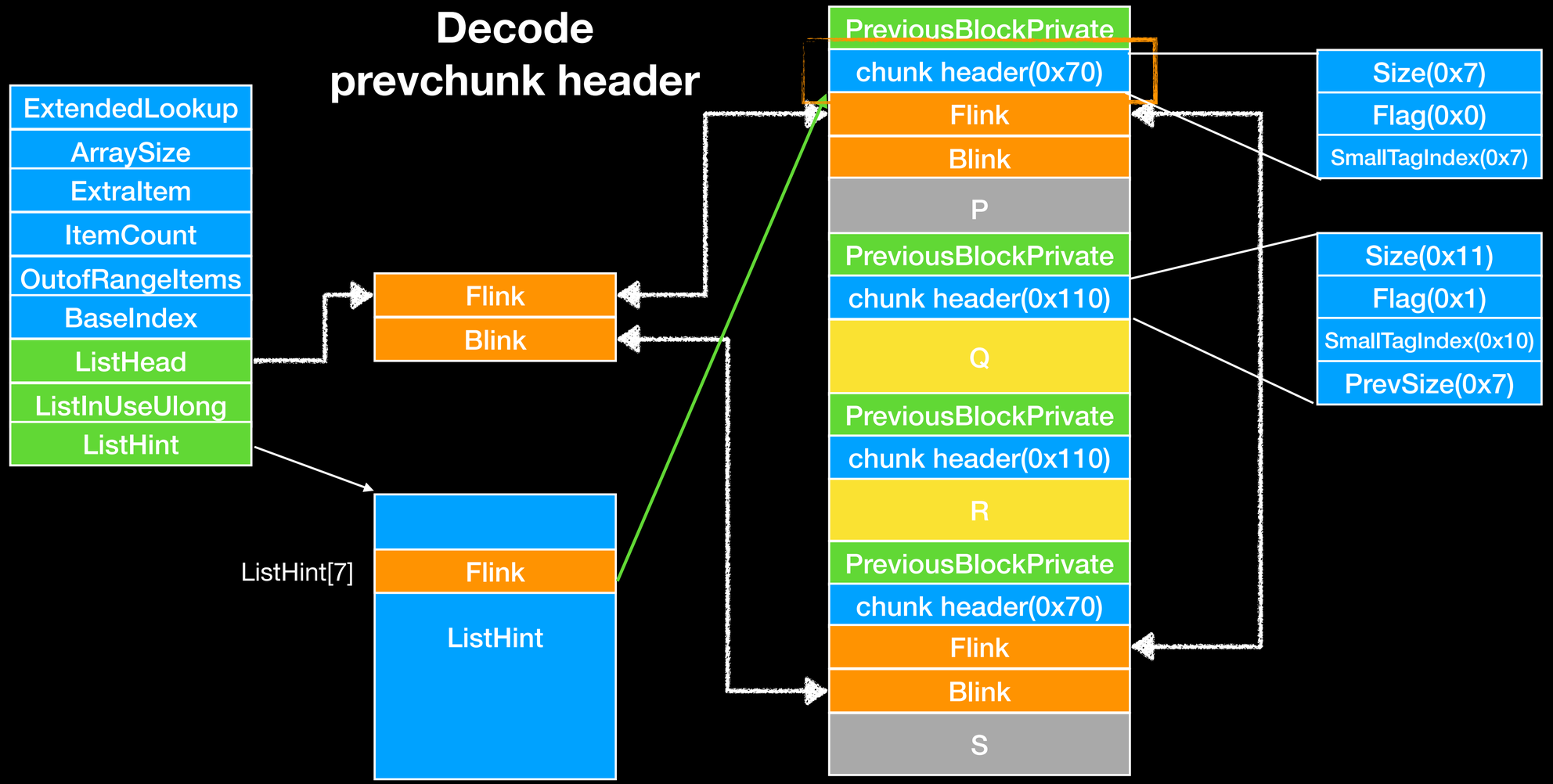
Windows Memory Allocator

merge chunk



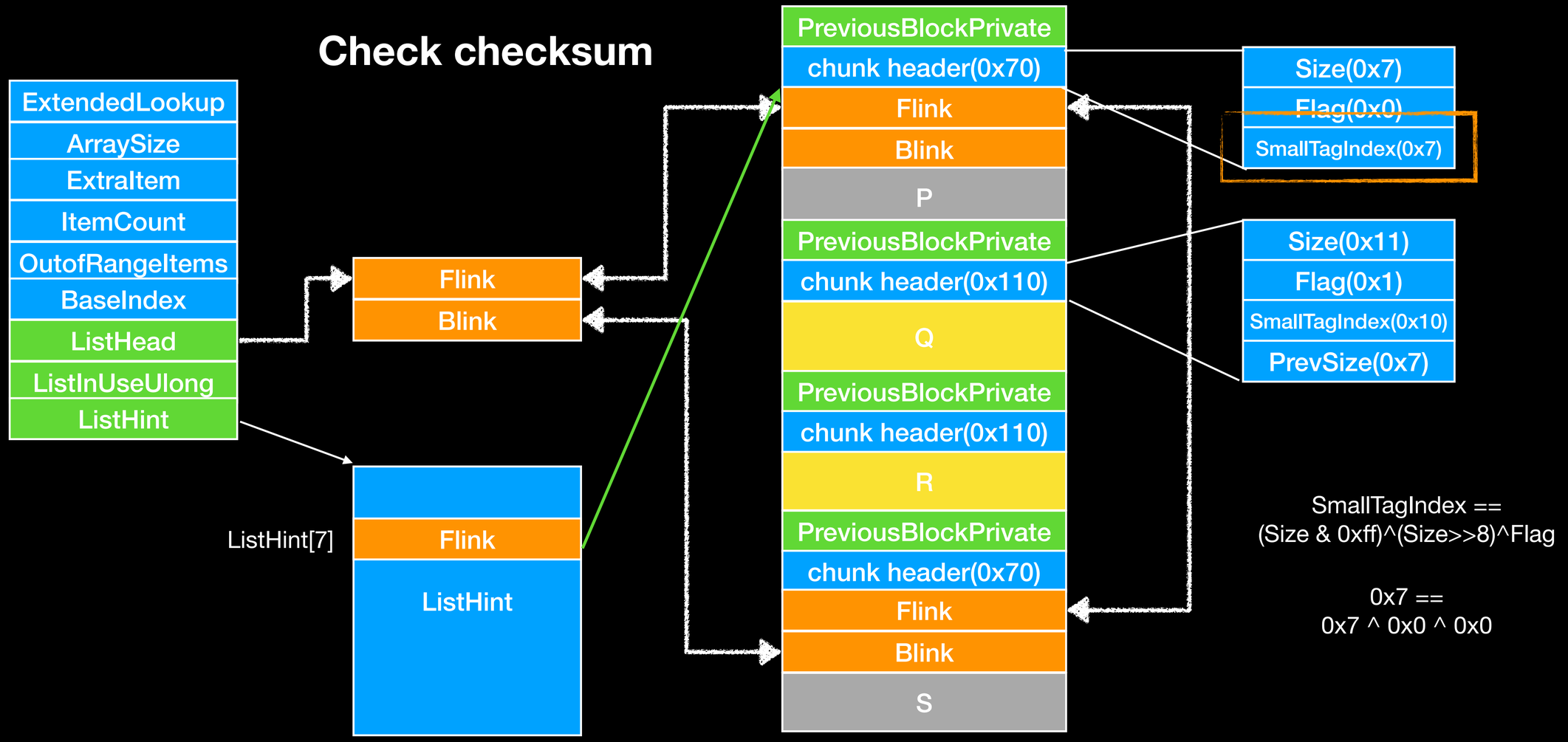
Windows Memory Allocator

merge chunk



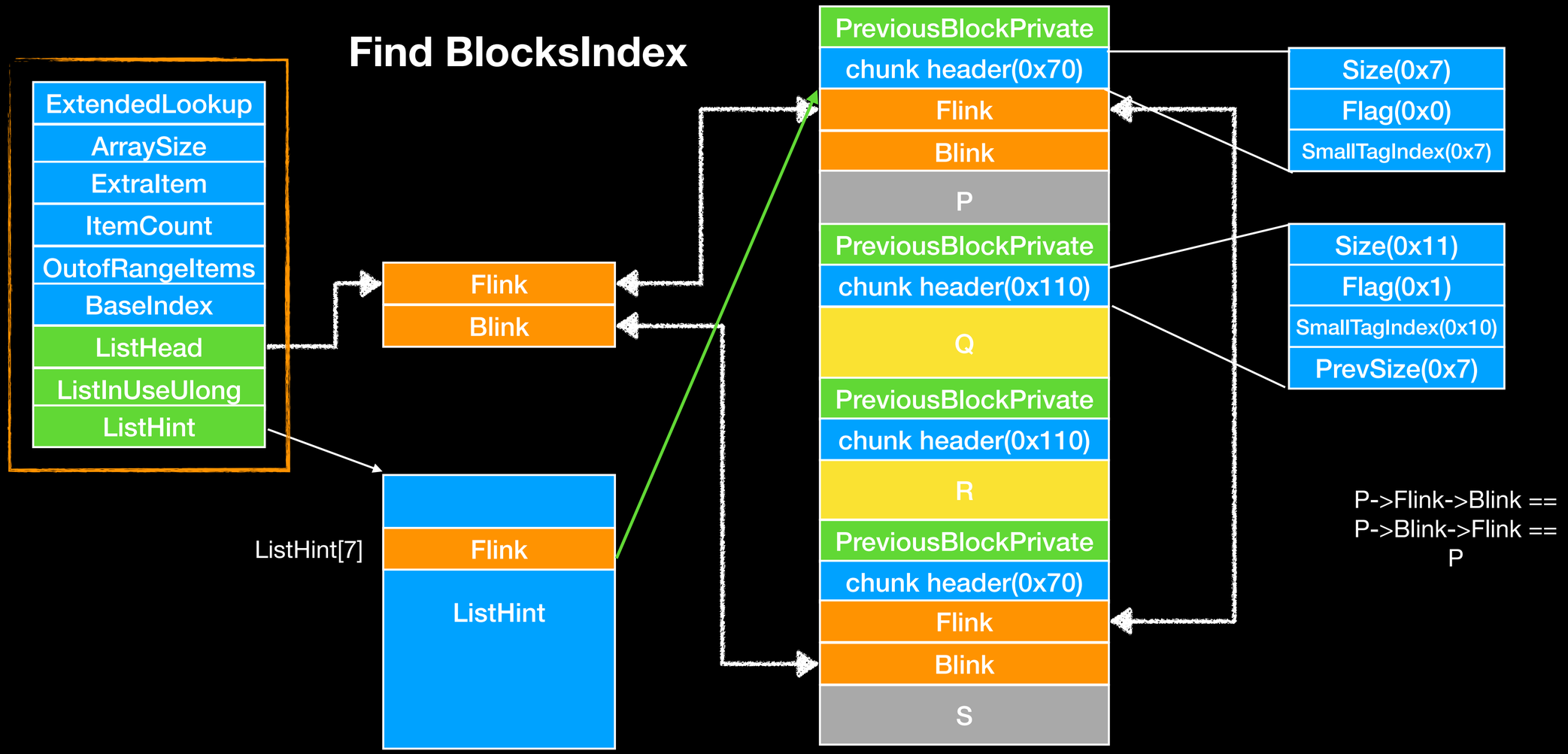
Windows Memory Allocator

merge chunk



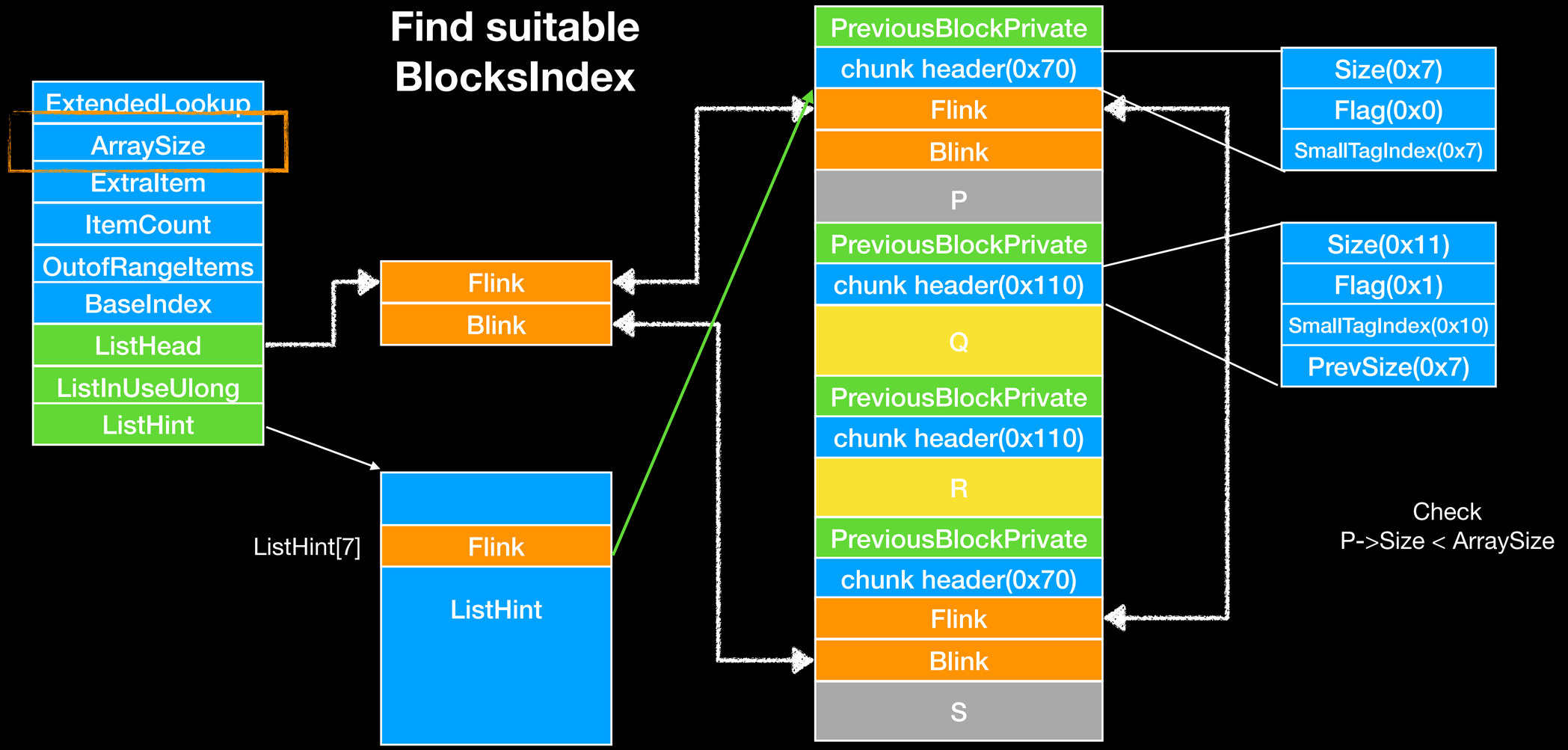
Windows Memory Allocator

merge chunk



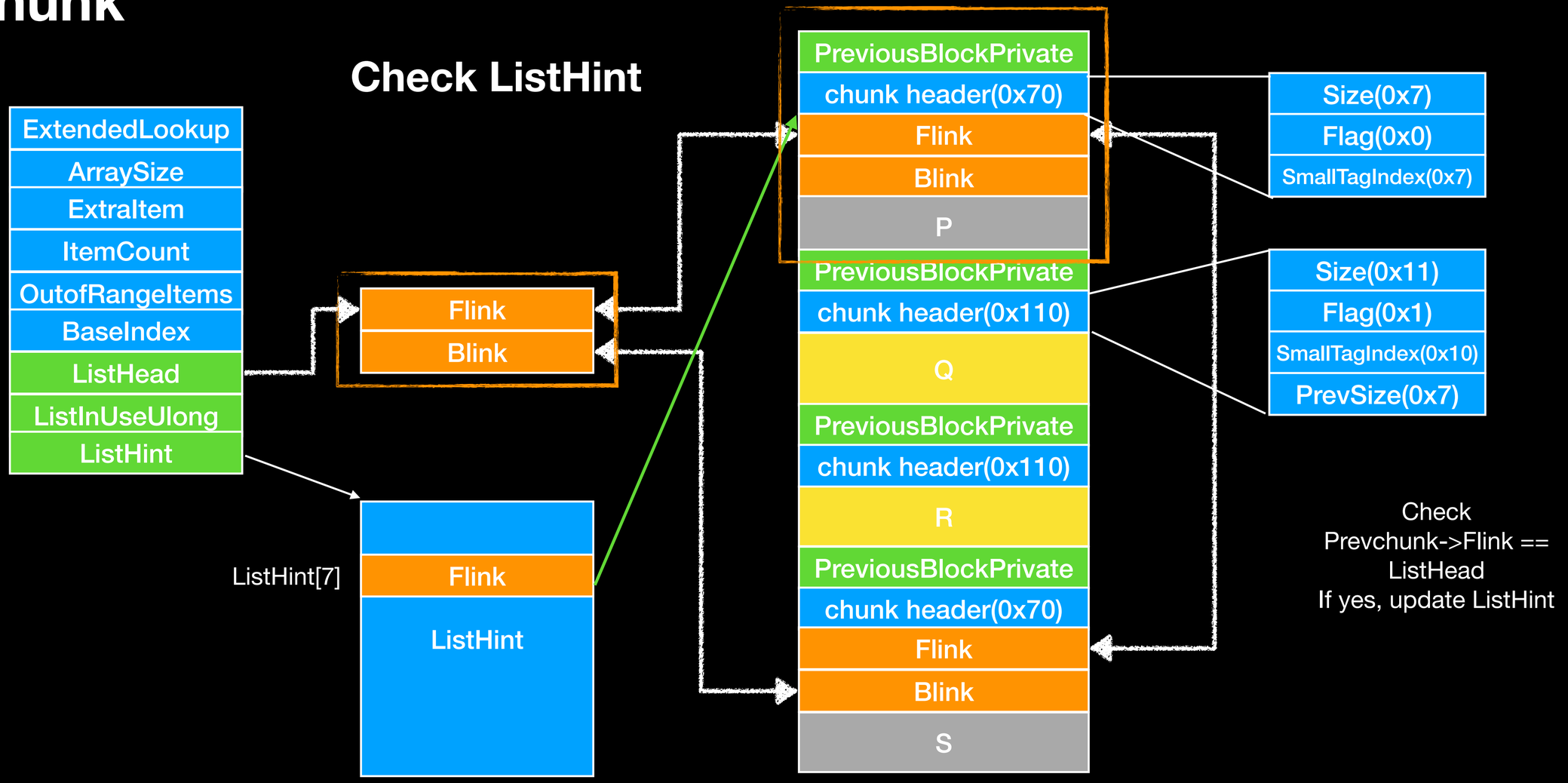
Windows Memory Allocator

merge chunk



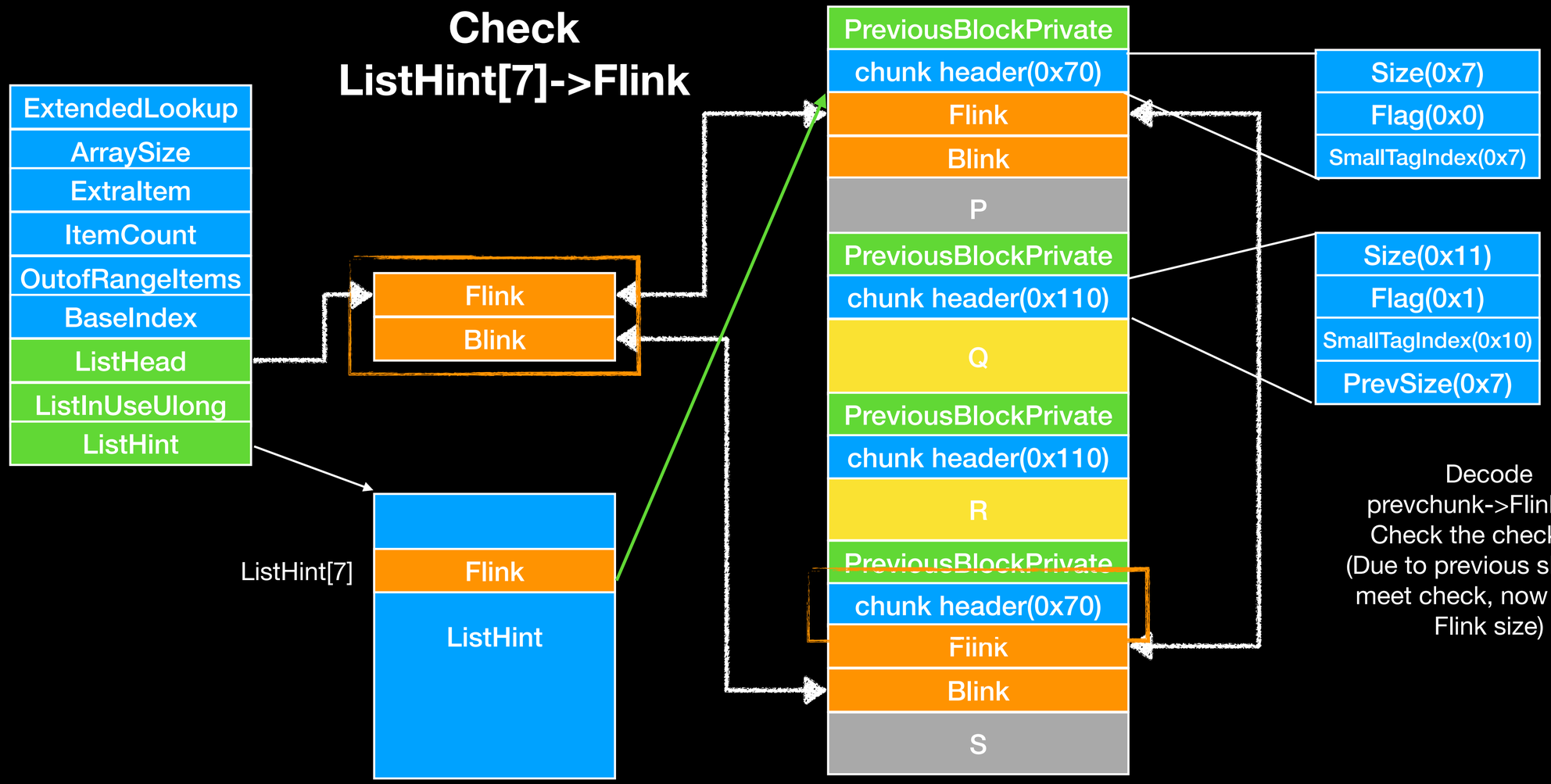
Windows Memory Allocator

merge chunk



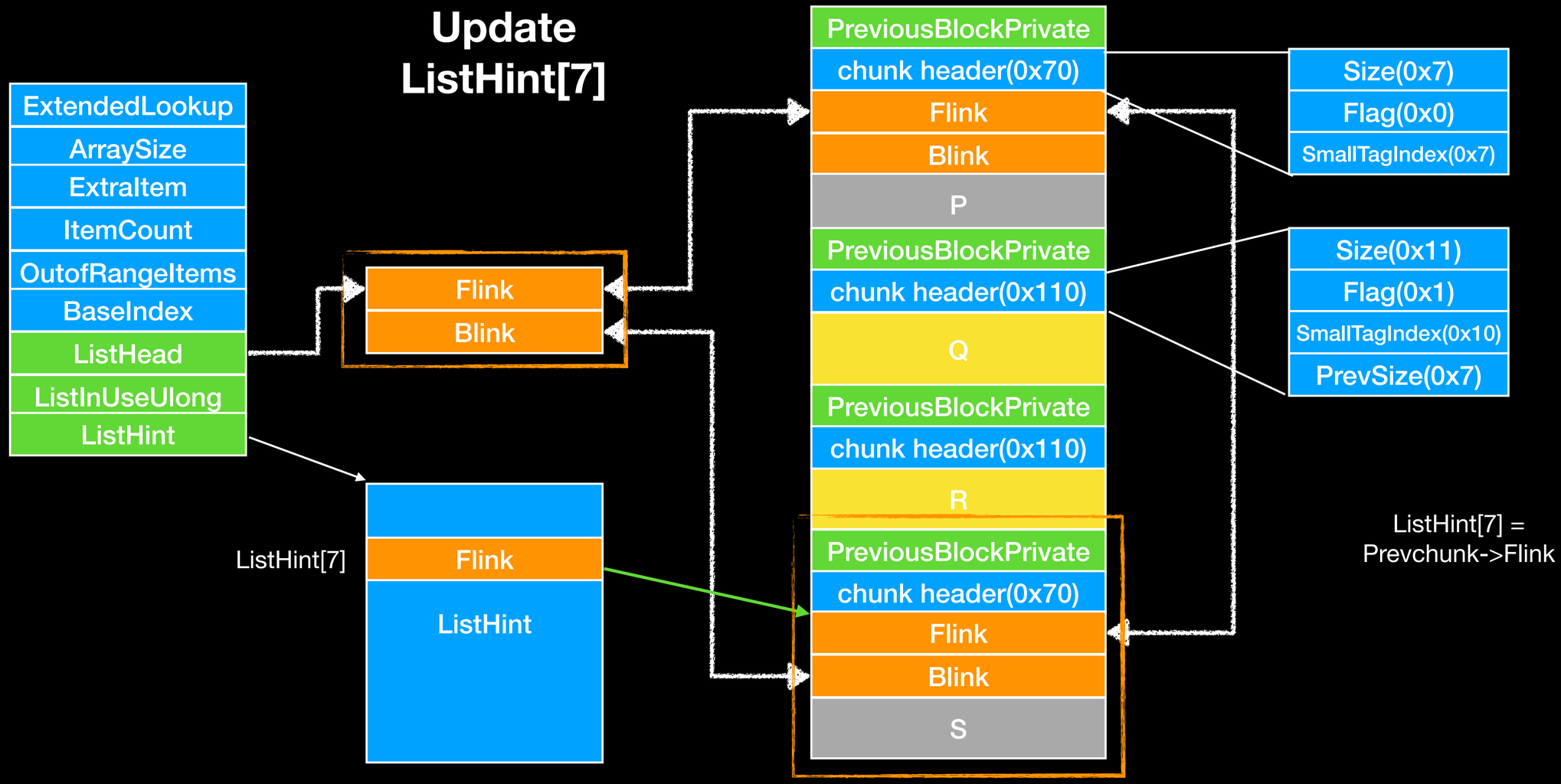
Windows Memory Allocator

merge chunk



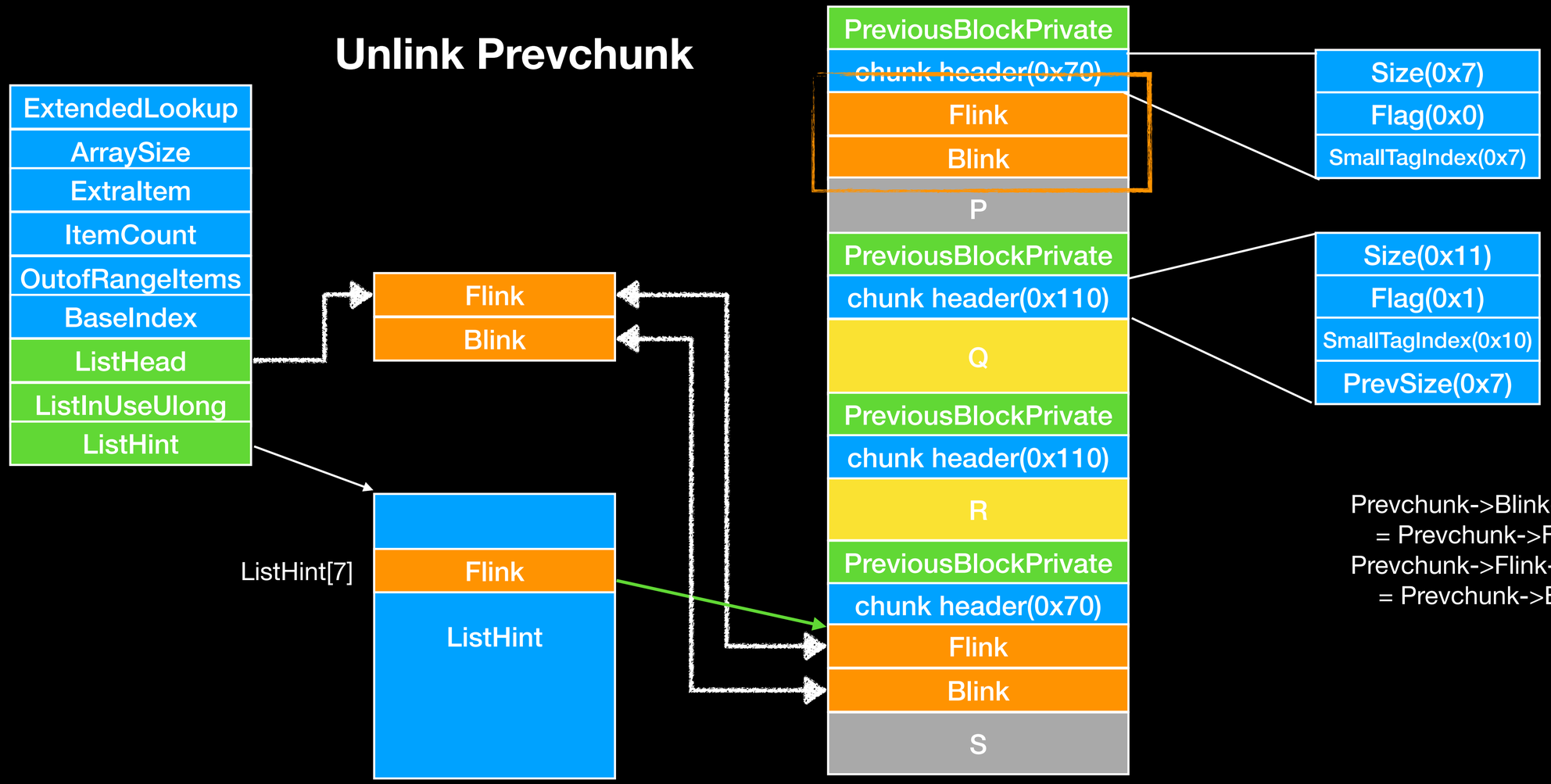
Windows Memory Allocator

merge chunk



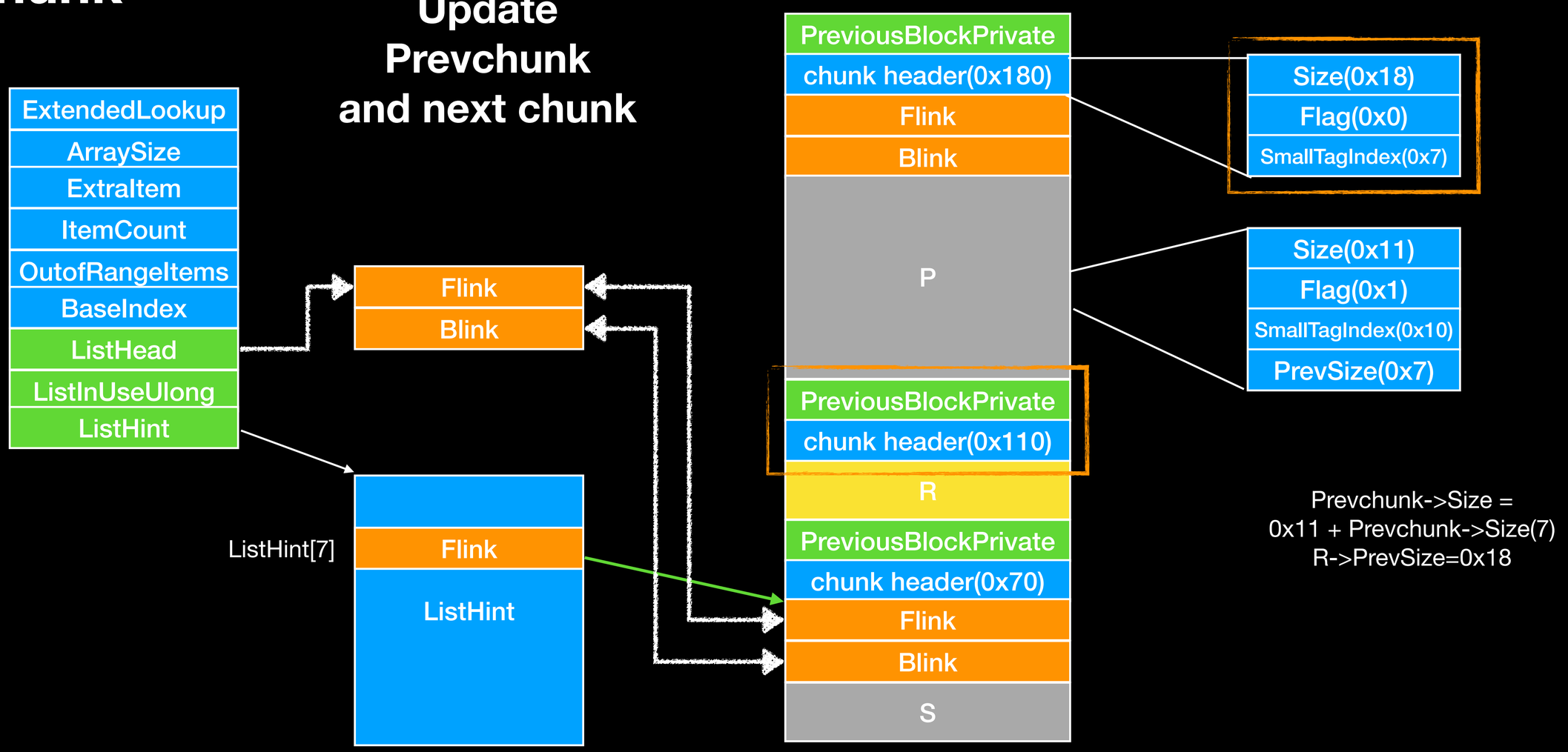
Windows Memory Allocator

merge chunk



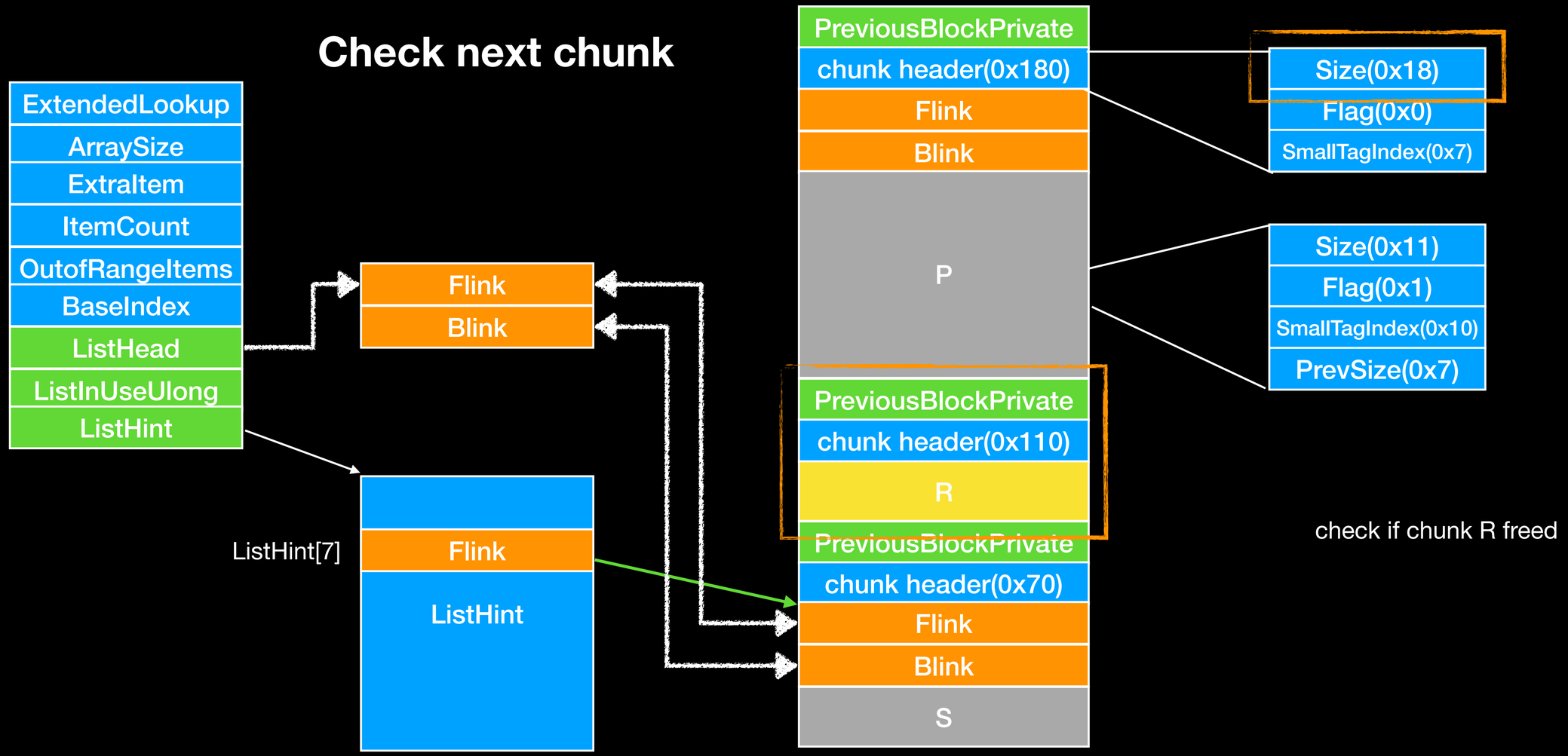
Windows Memory Allocator

merge chunk



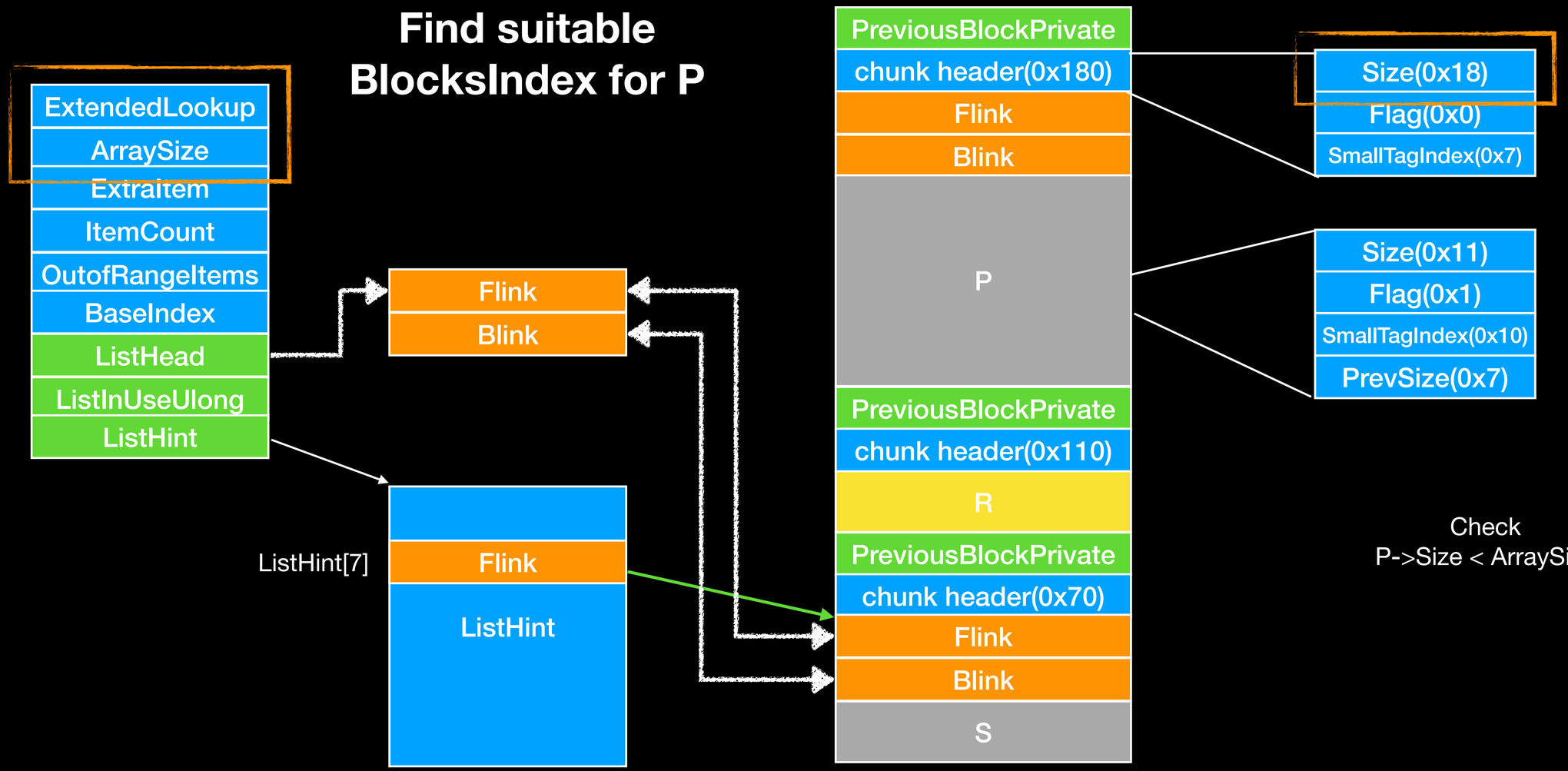
Windows Memory Allocator

merge chunk



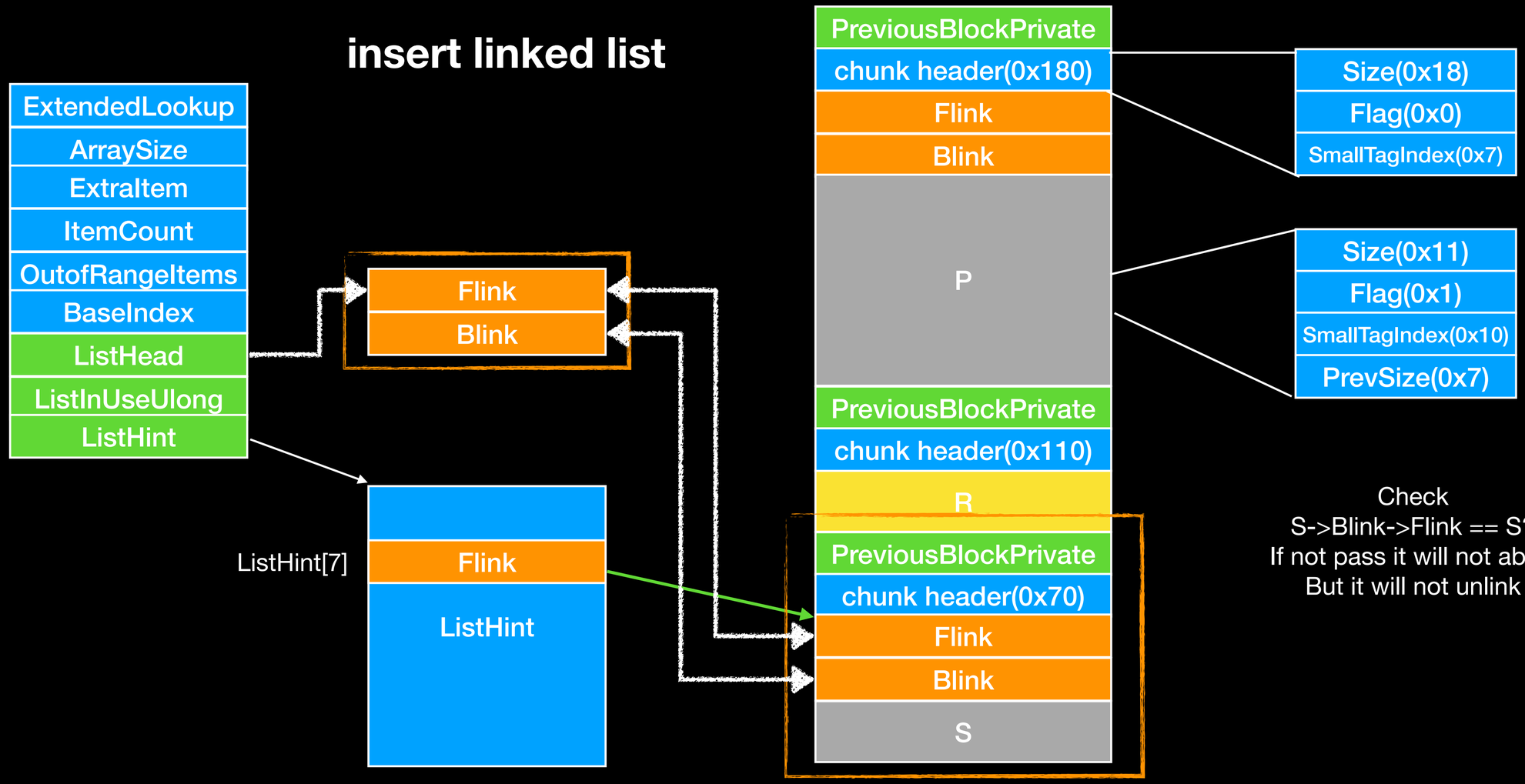
Windows Memory Allocator

merge chunk



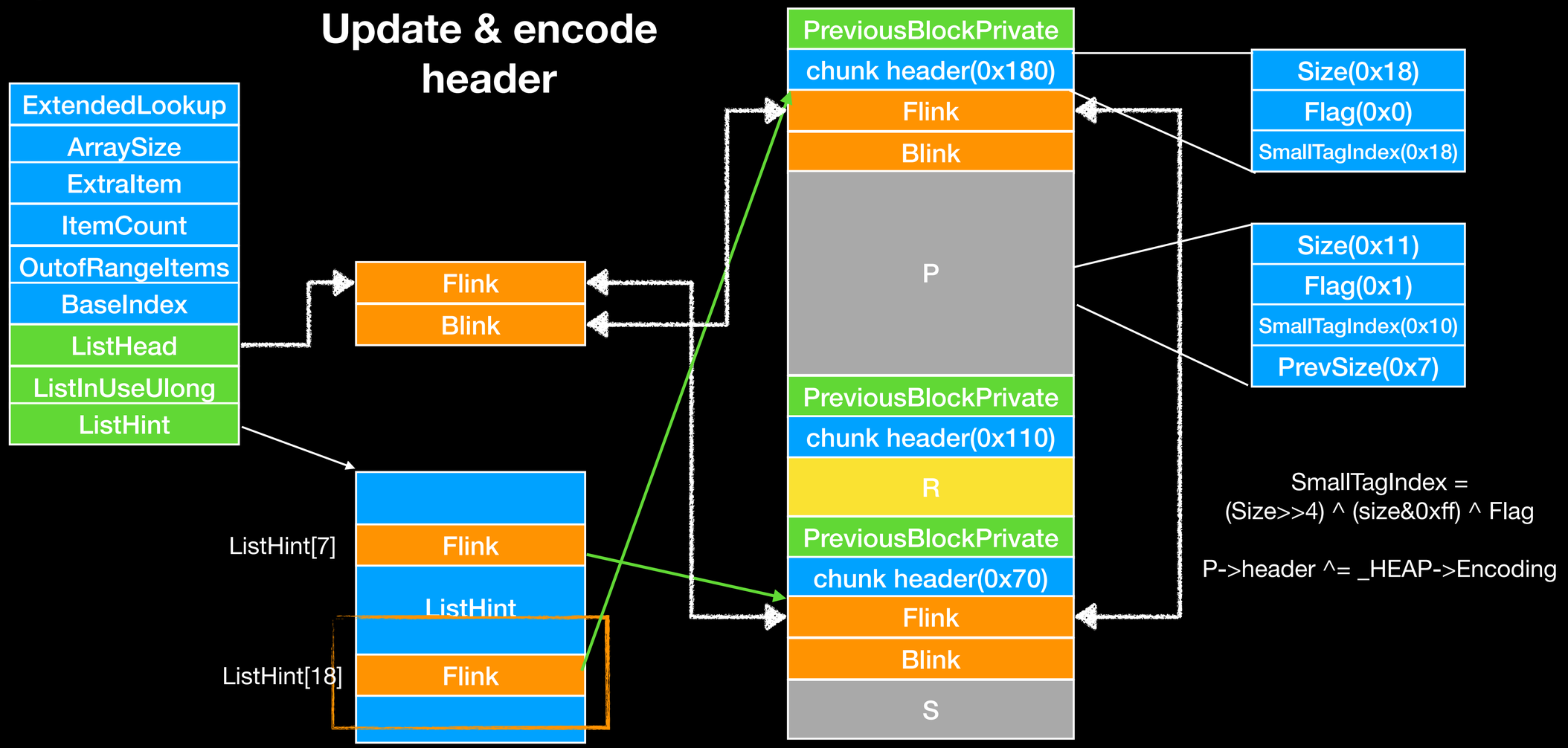
Windows Memory Allocator

merge chunk



Windows Memory Allocator

merge chunk

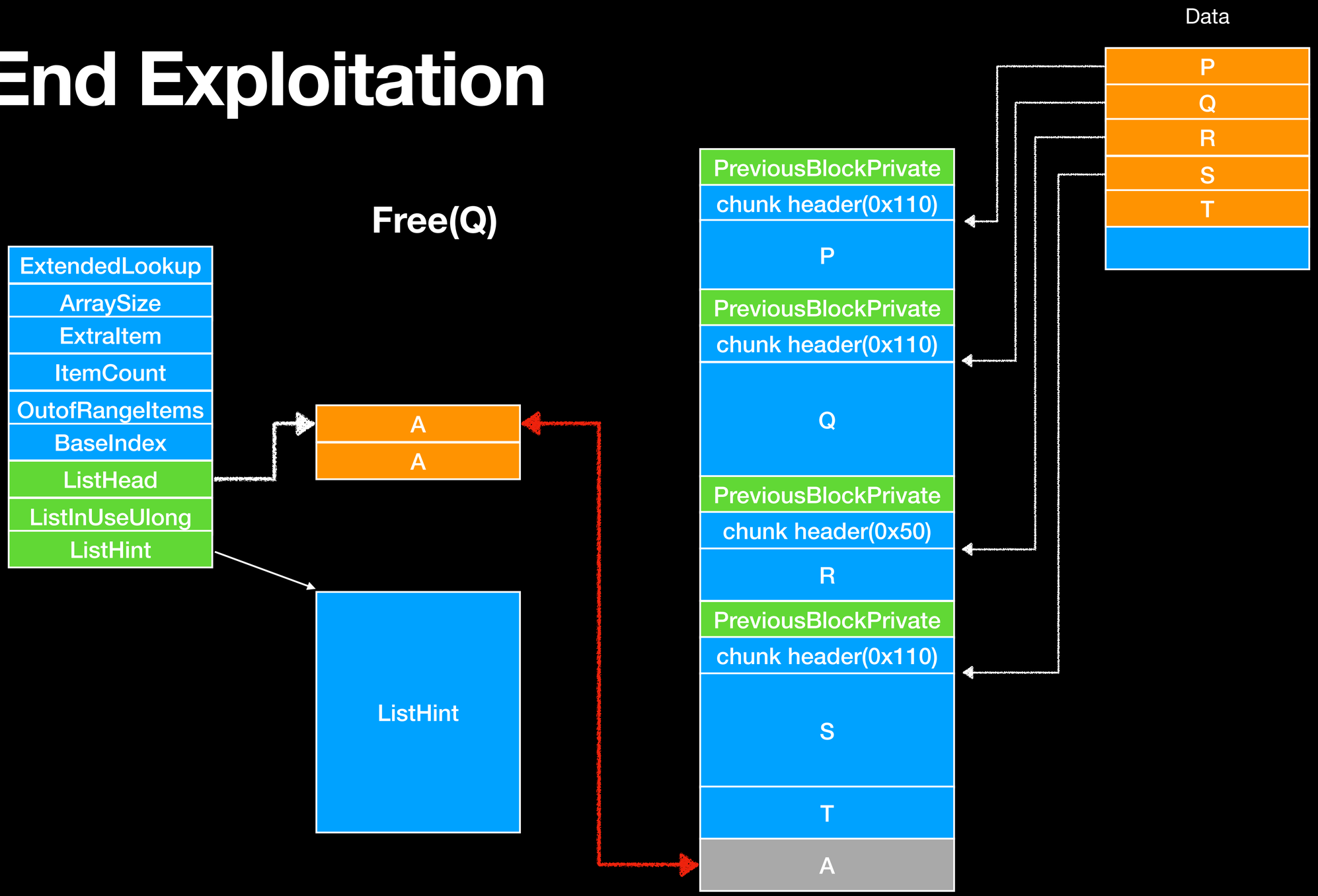


BackEnd Exploitation

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

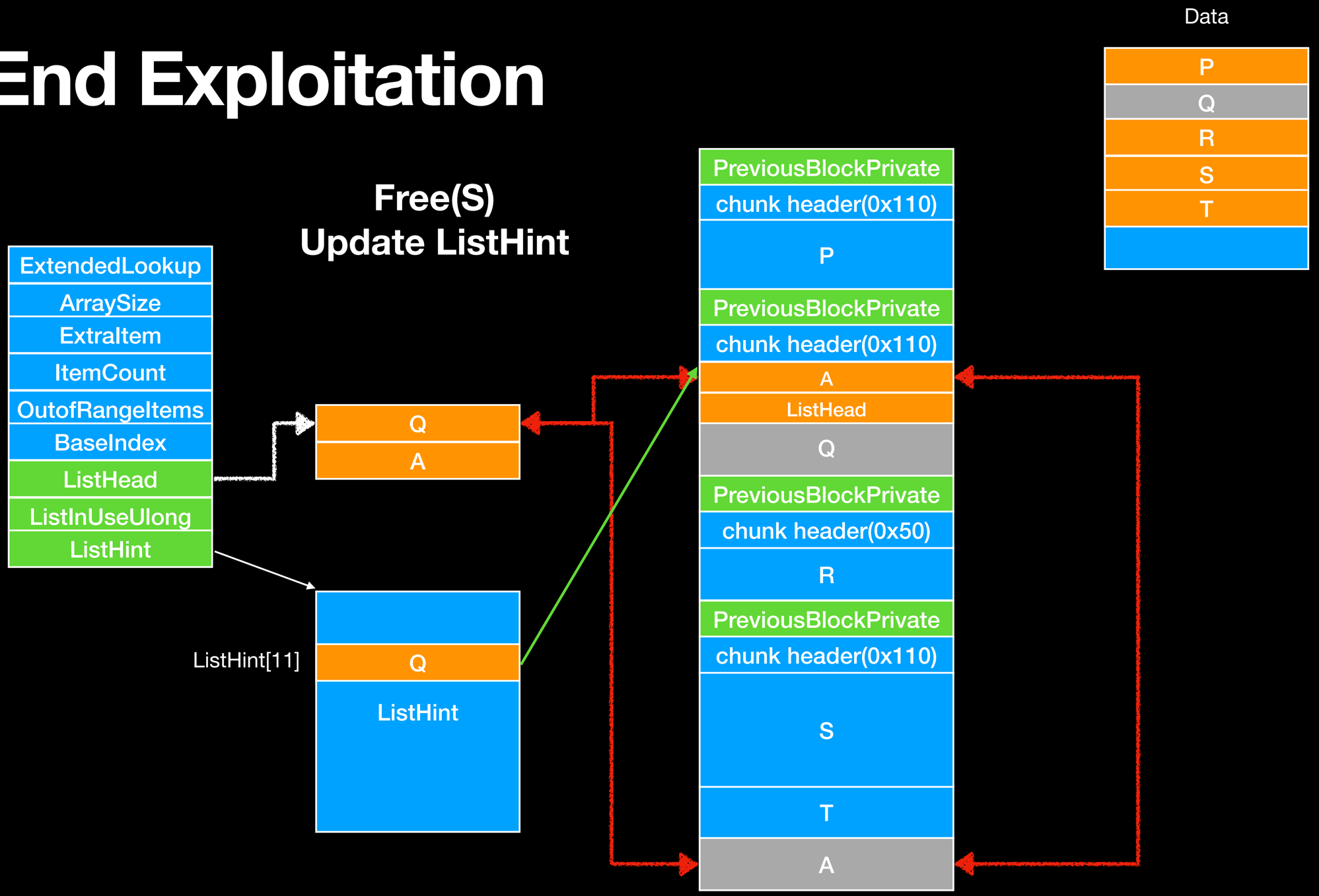
BackEnd Exploitation

UAF



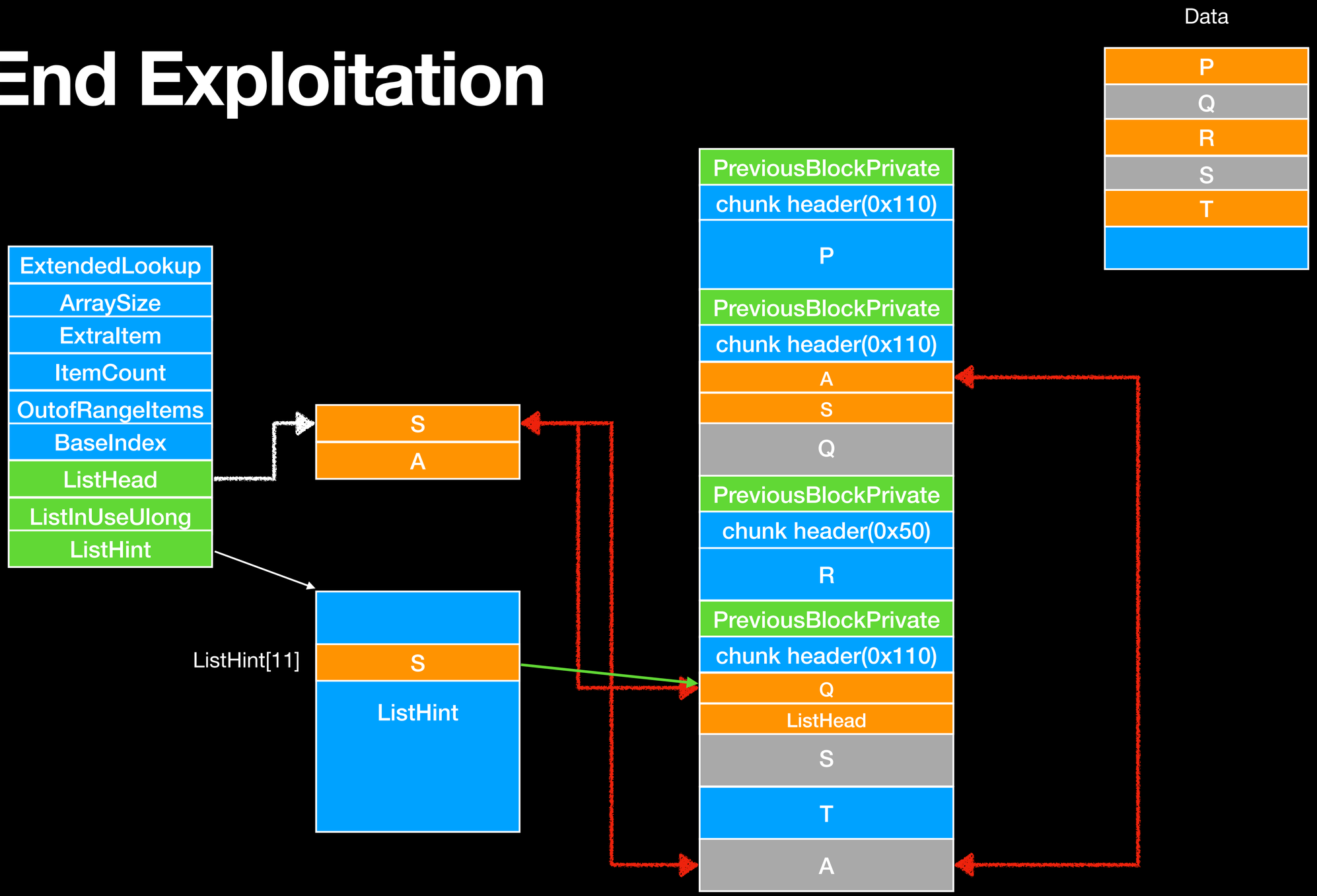
BackEnd Exploitation

UAF



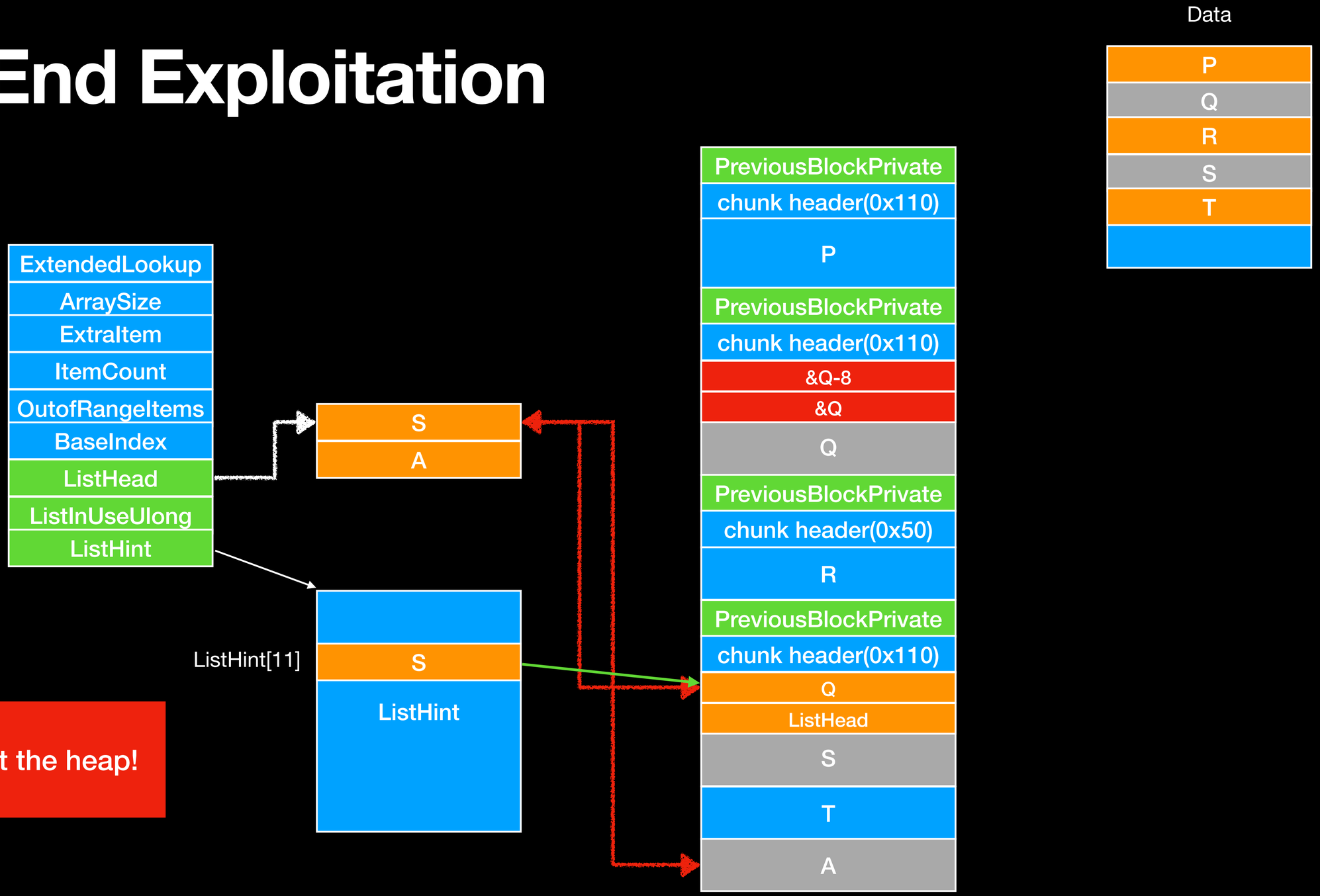
BackEnd Exploitation

UAF



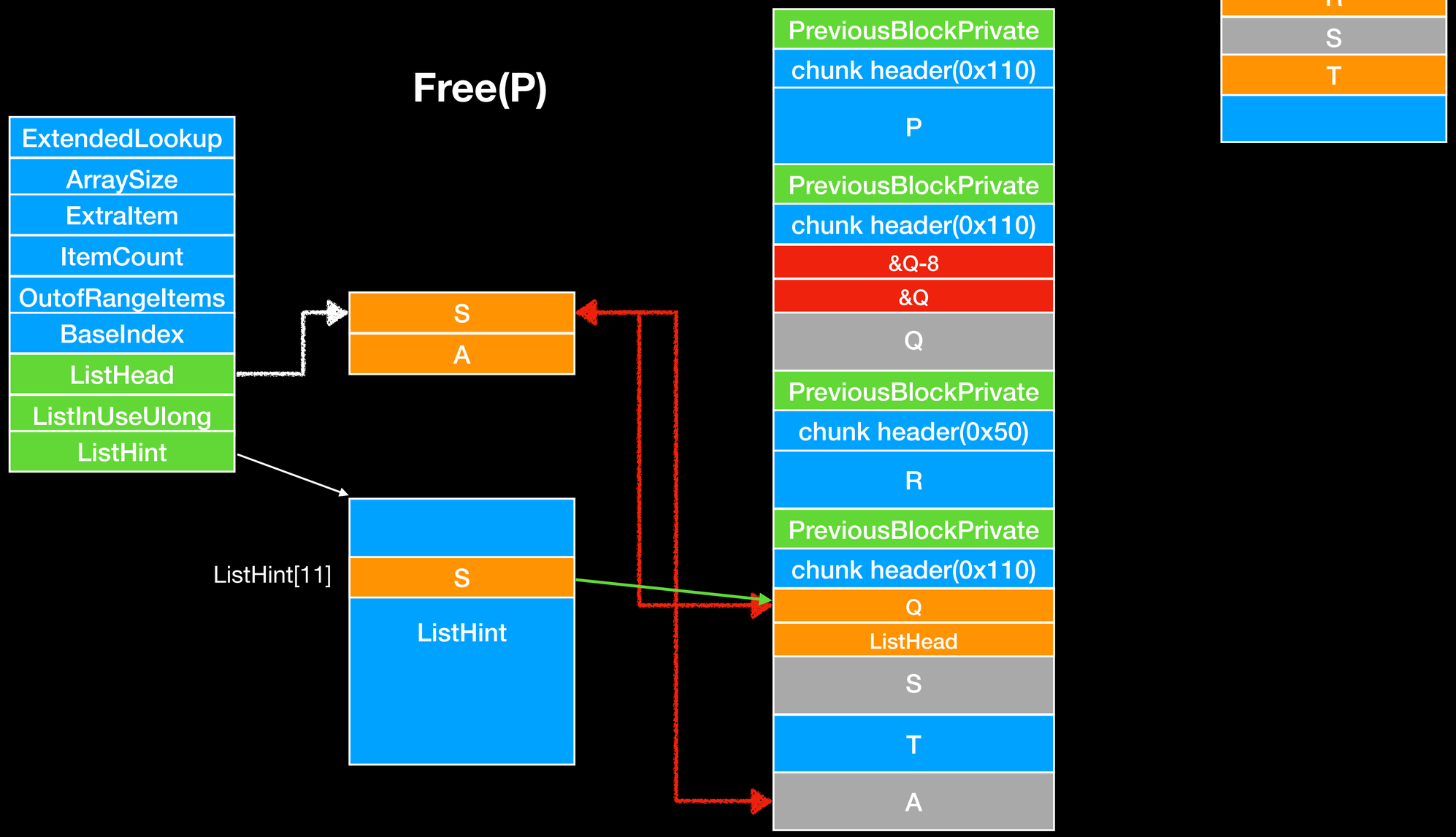
BackEnd Exploitation

UAF



BackEnd Exploitation

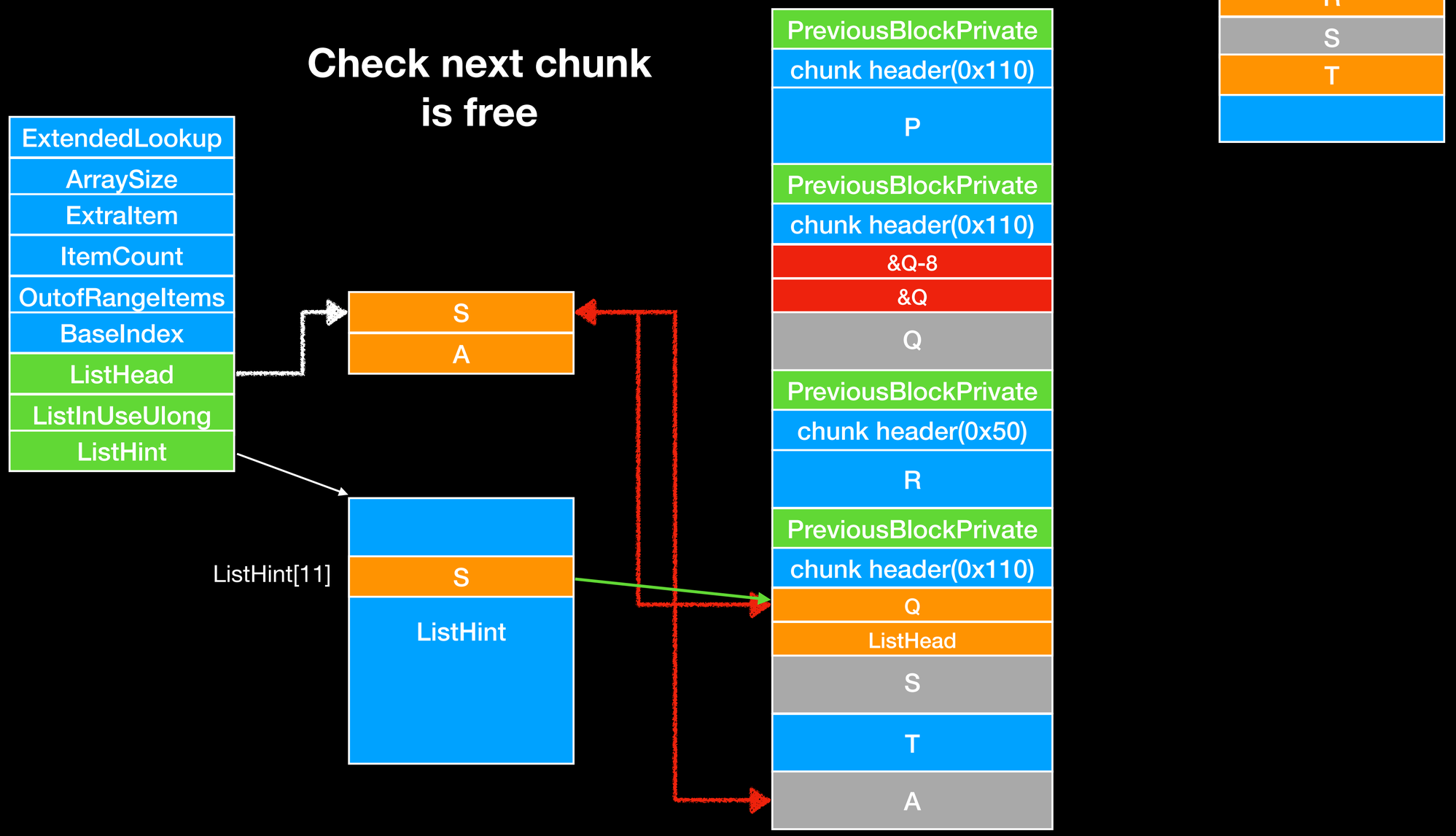
UAF



Data

BackEnd Exploitation

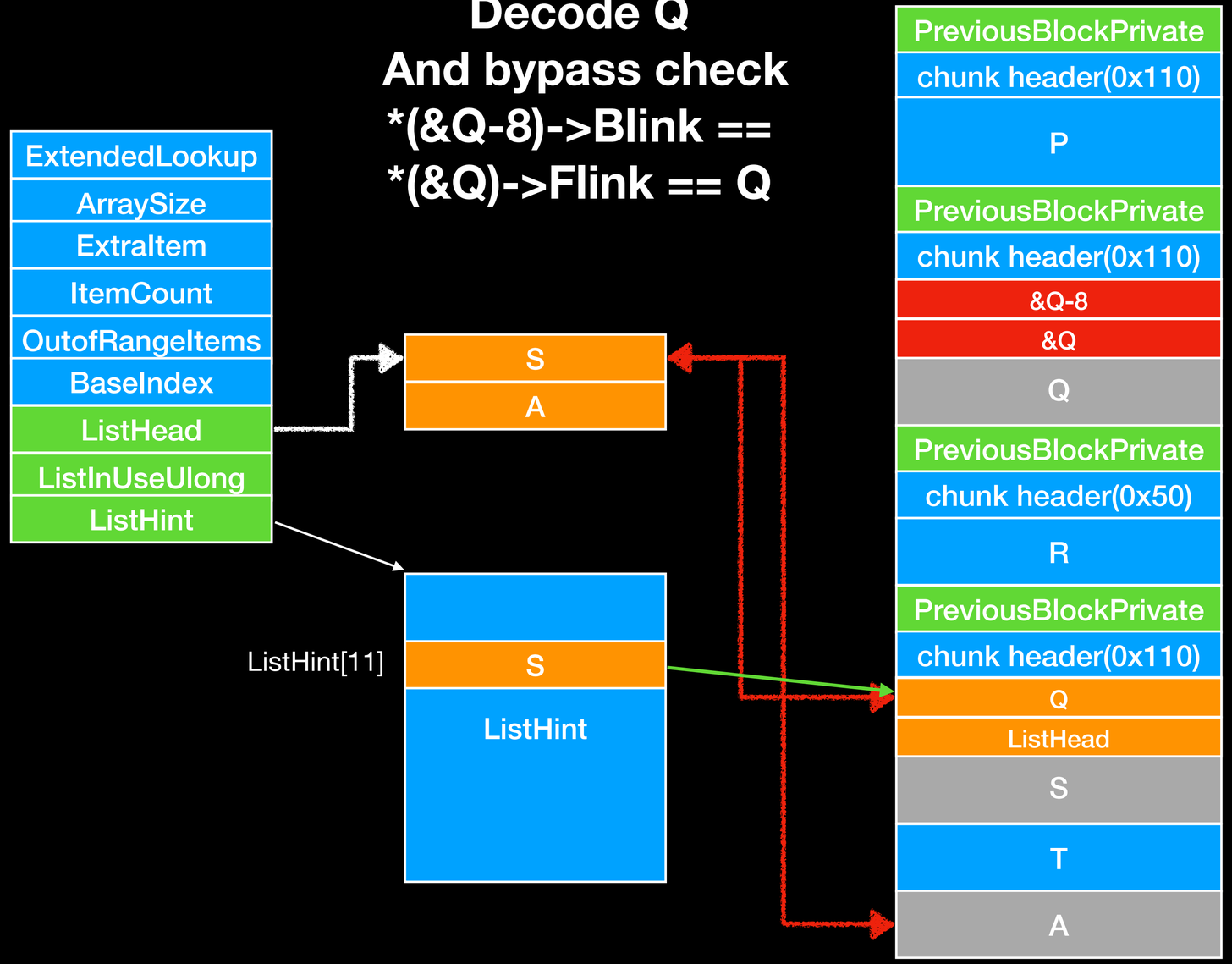
UAF



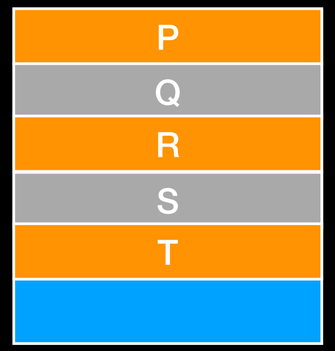
BackEnd Exploitation

UAF

Decode Q
And bypass check
***(&Q-8)->Blink ==**
***(&Q)->Flink == Q**

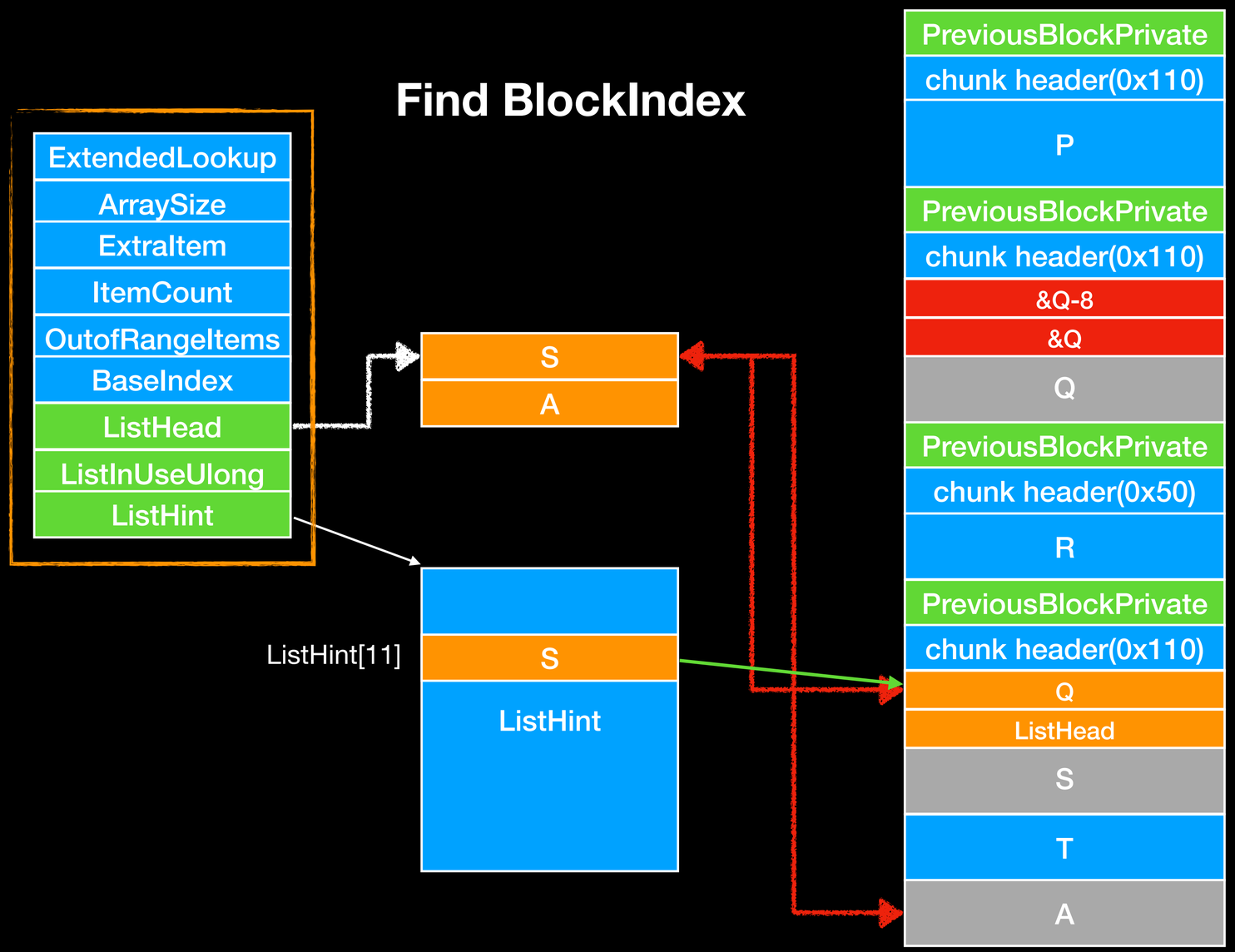


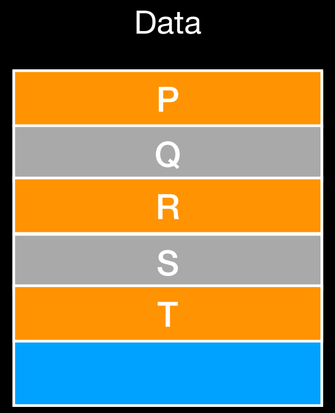
Data



BackEnd Exploitation

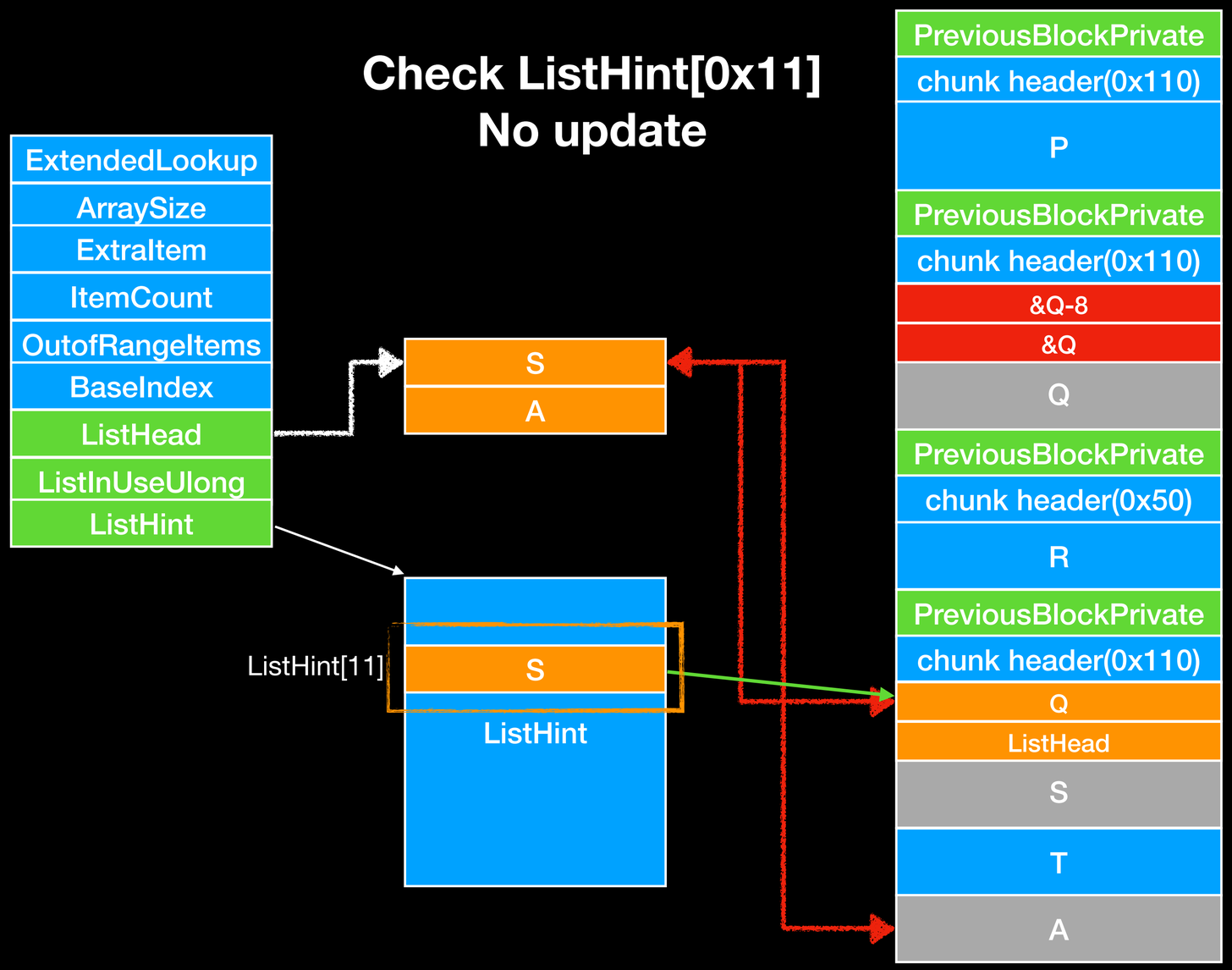
UAF

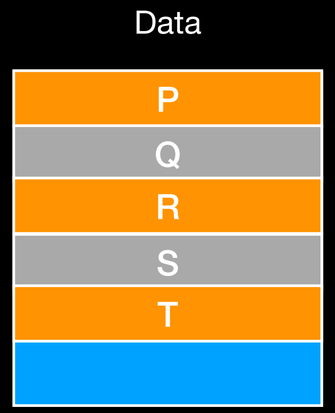




BackEnd Exploitation

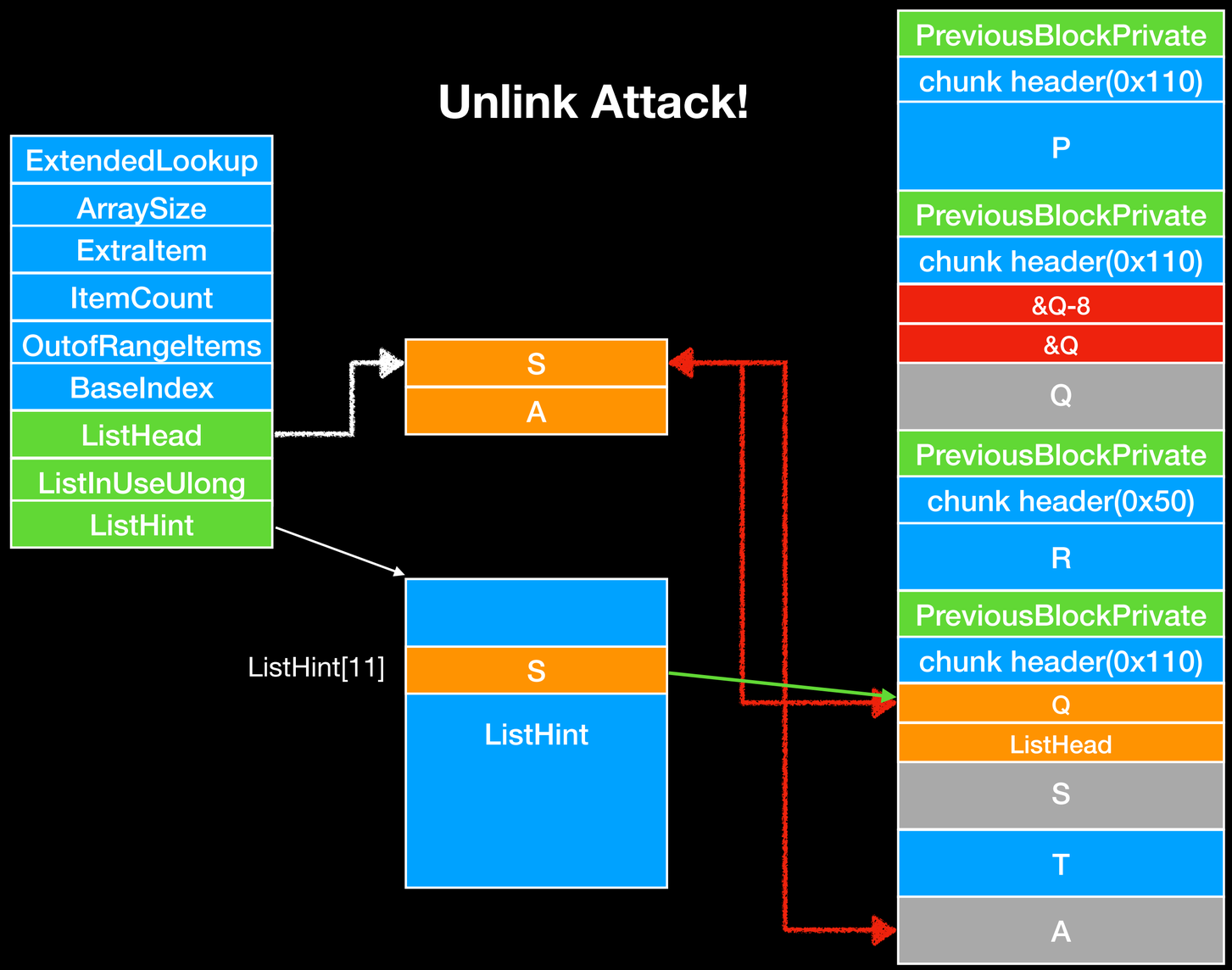
UAF

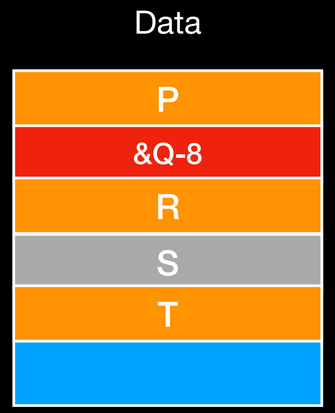




BackEnd Exploitation

UAF

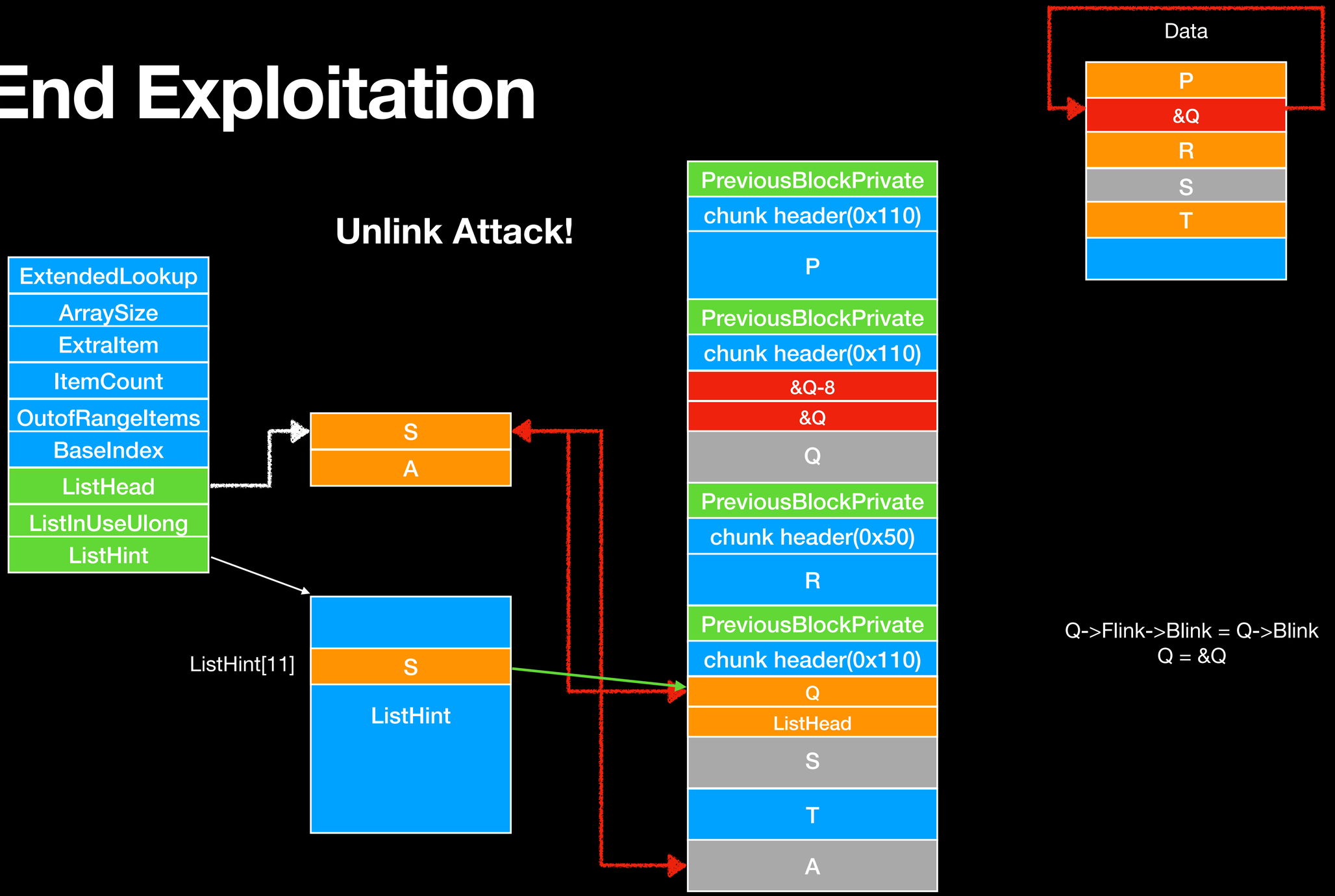




Q->Blink->Flink = Q->Flink
Q = &Q-8

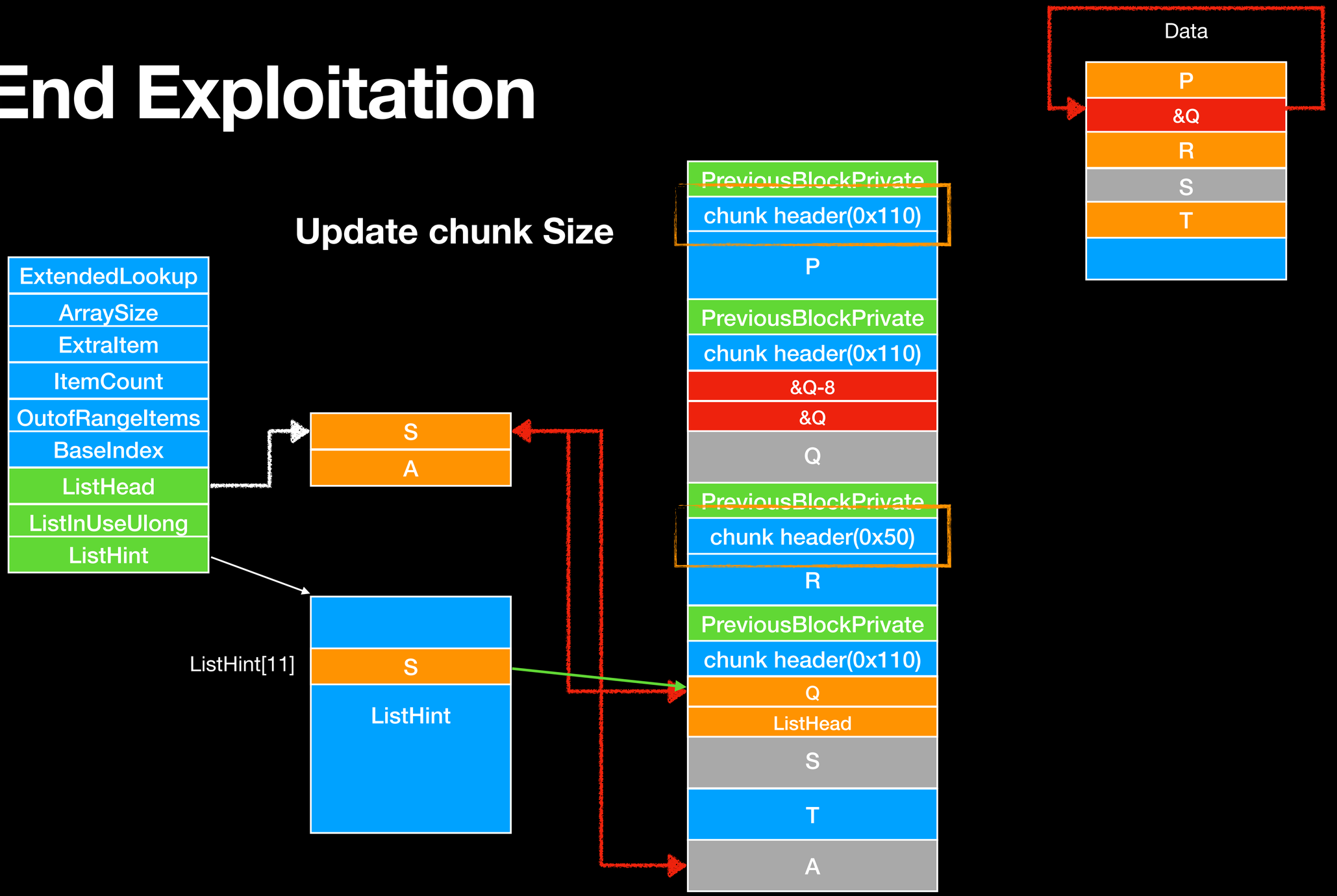
BackEnd Exploitation

UAF



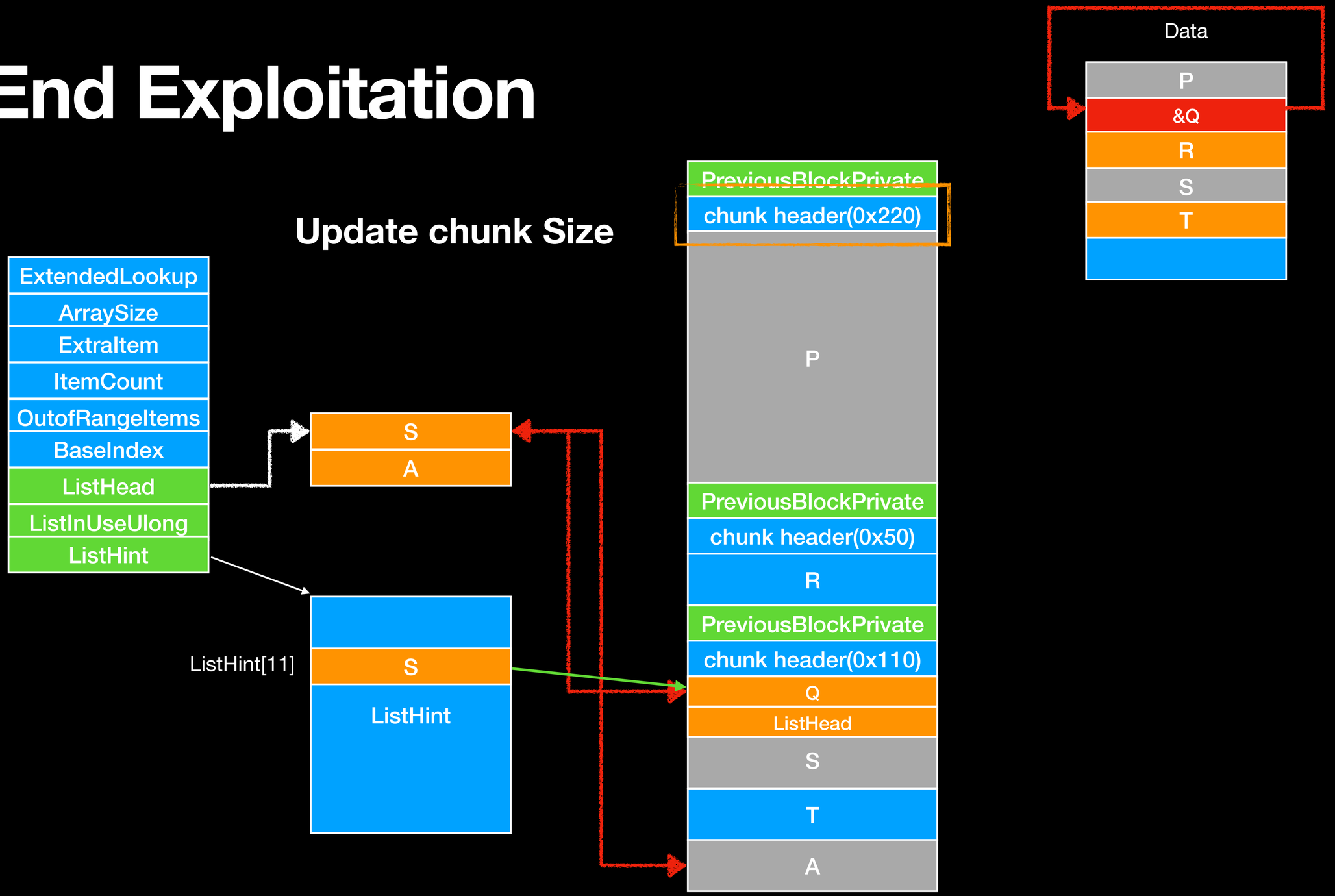
BackEnd Exploitation

UAF



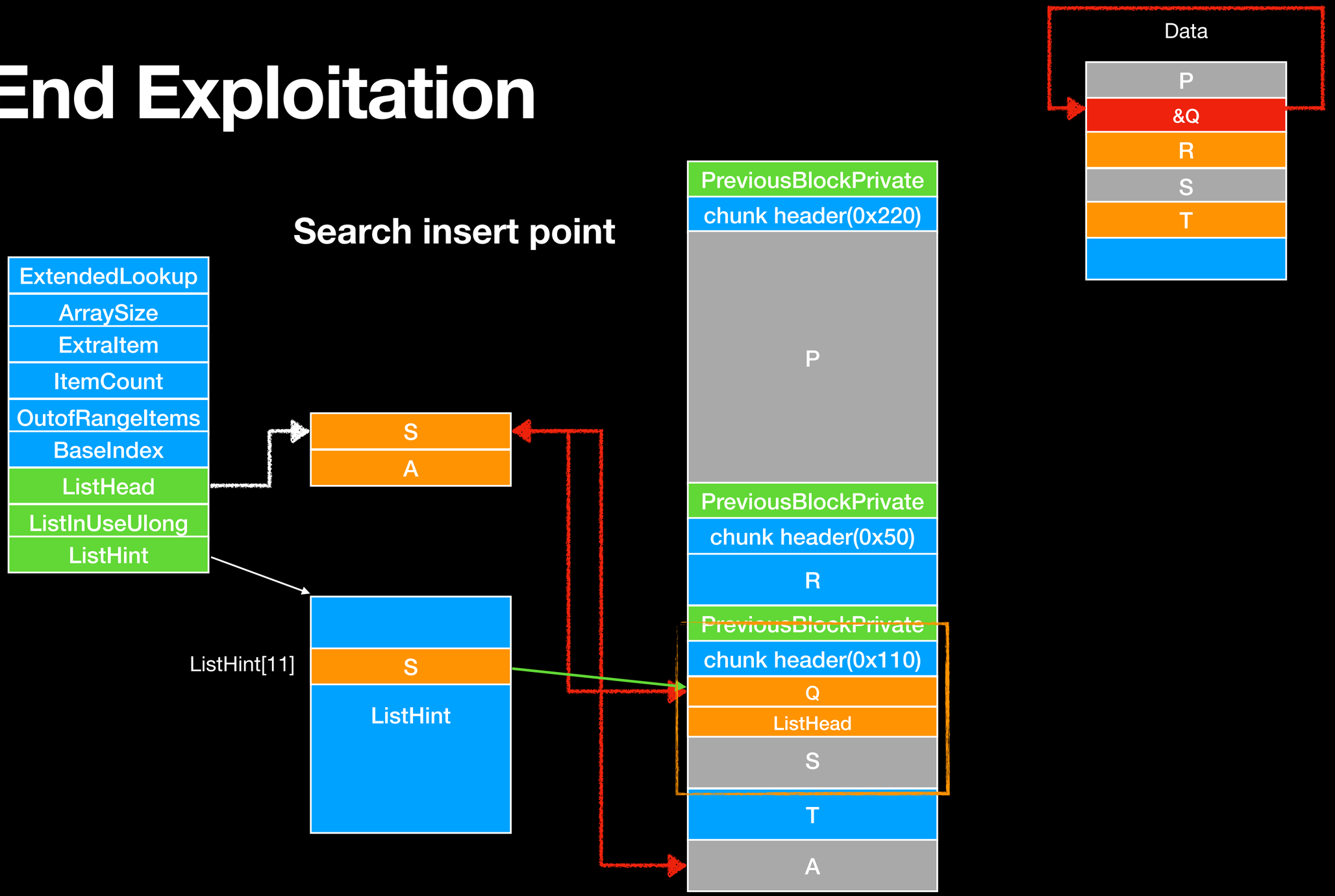
BackEnd Exploitation

UAF



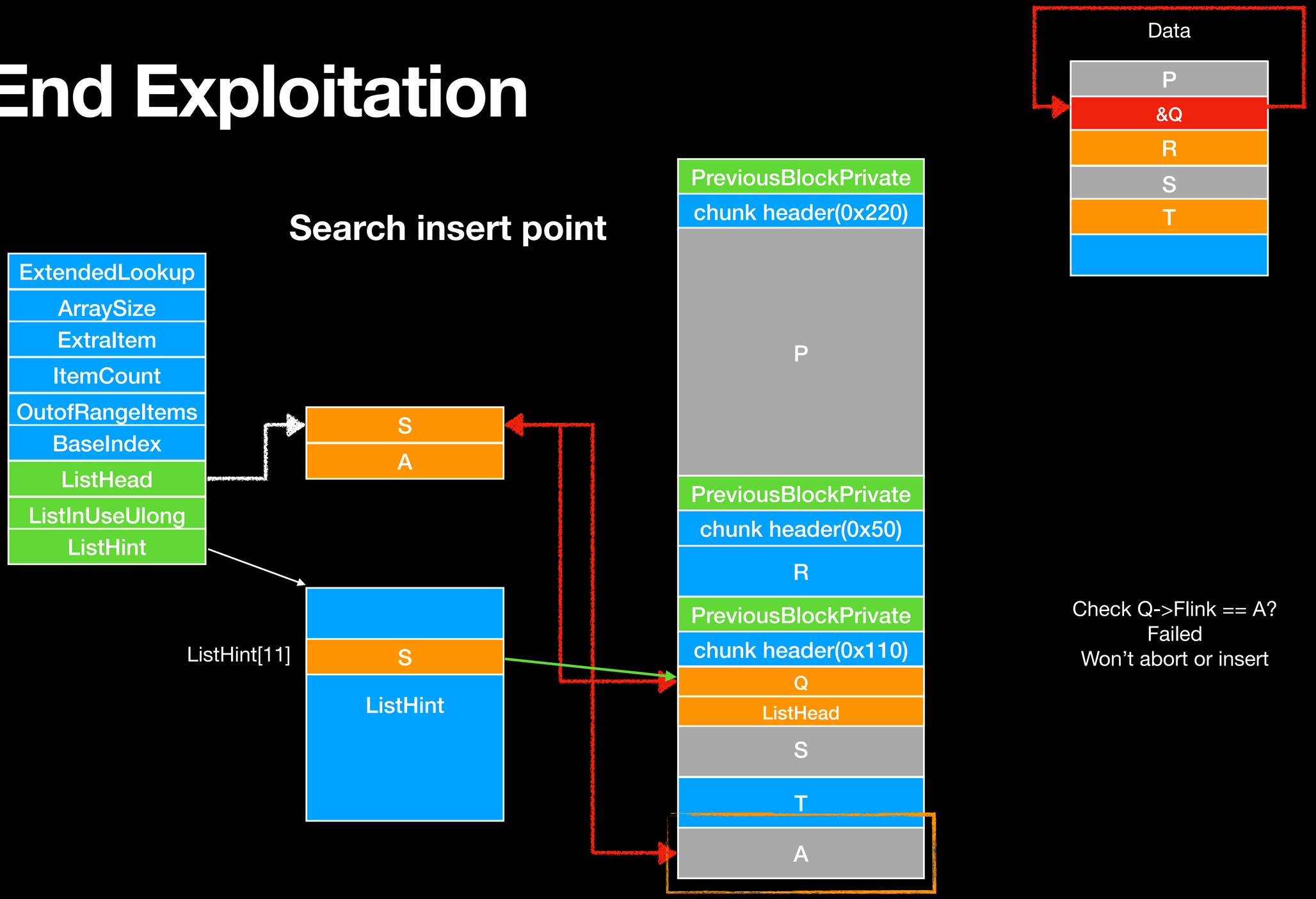
BackEnd Exploitation

UAF



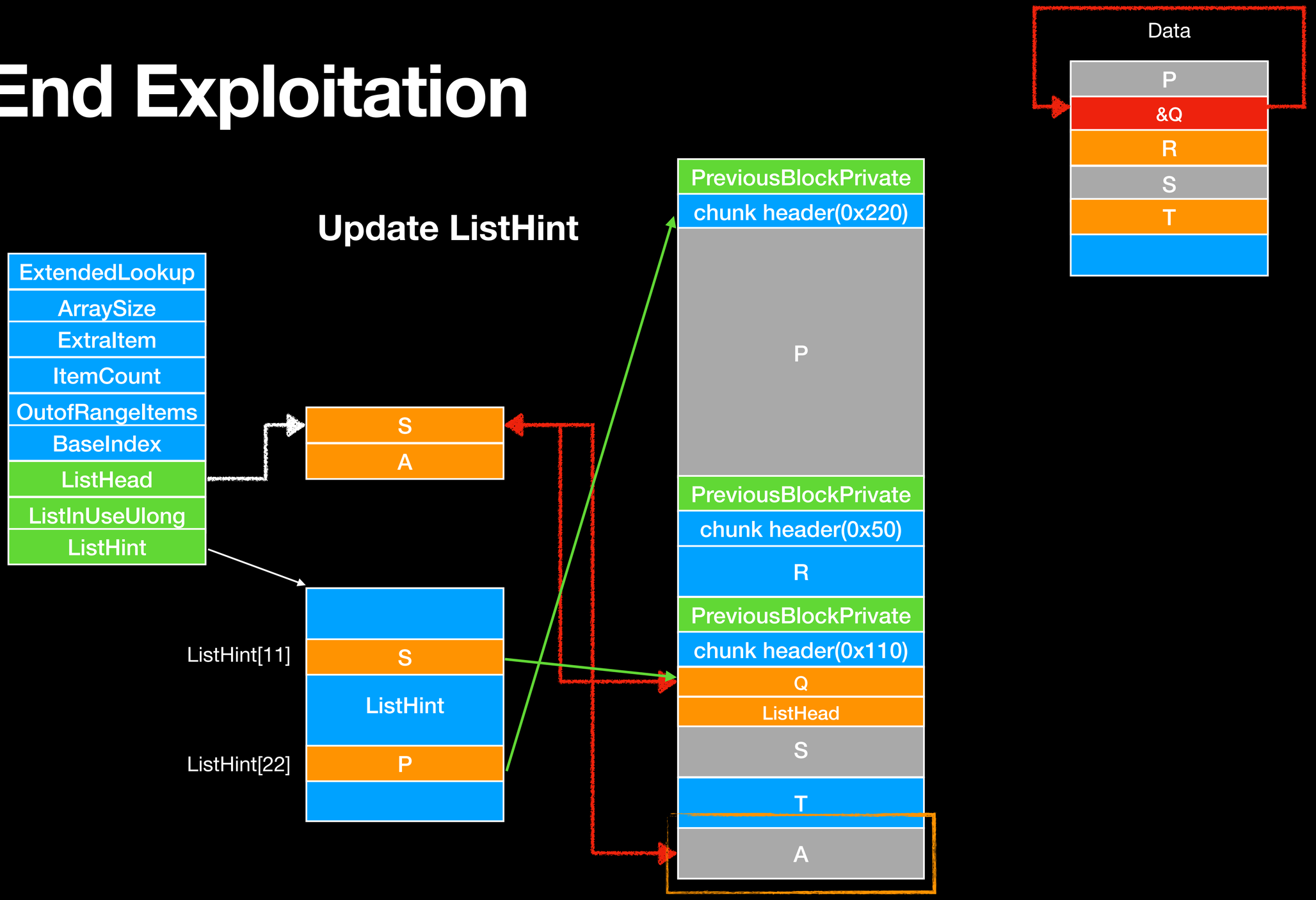
BackEnd Exploitation

UAF



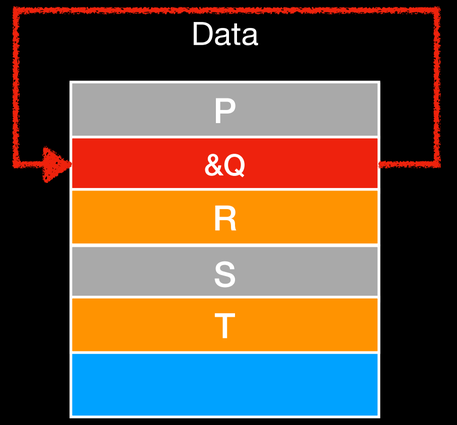
BackEnd Exploitation

UAF



BackEnd Exploitation
UAF

- Edit Q
  - Arbitrary memory reading/writing primitive



BackEnd Exploitation
UAF

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

BackEnd Exploitation
UAF

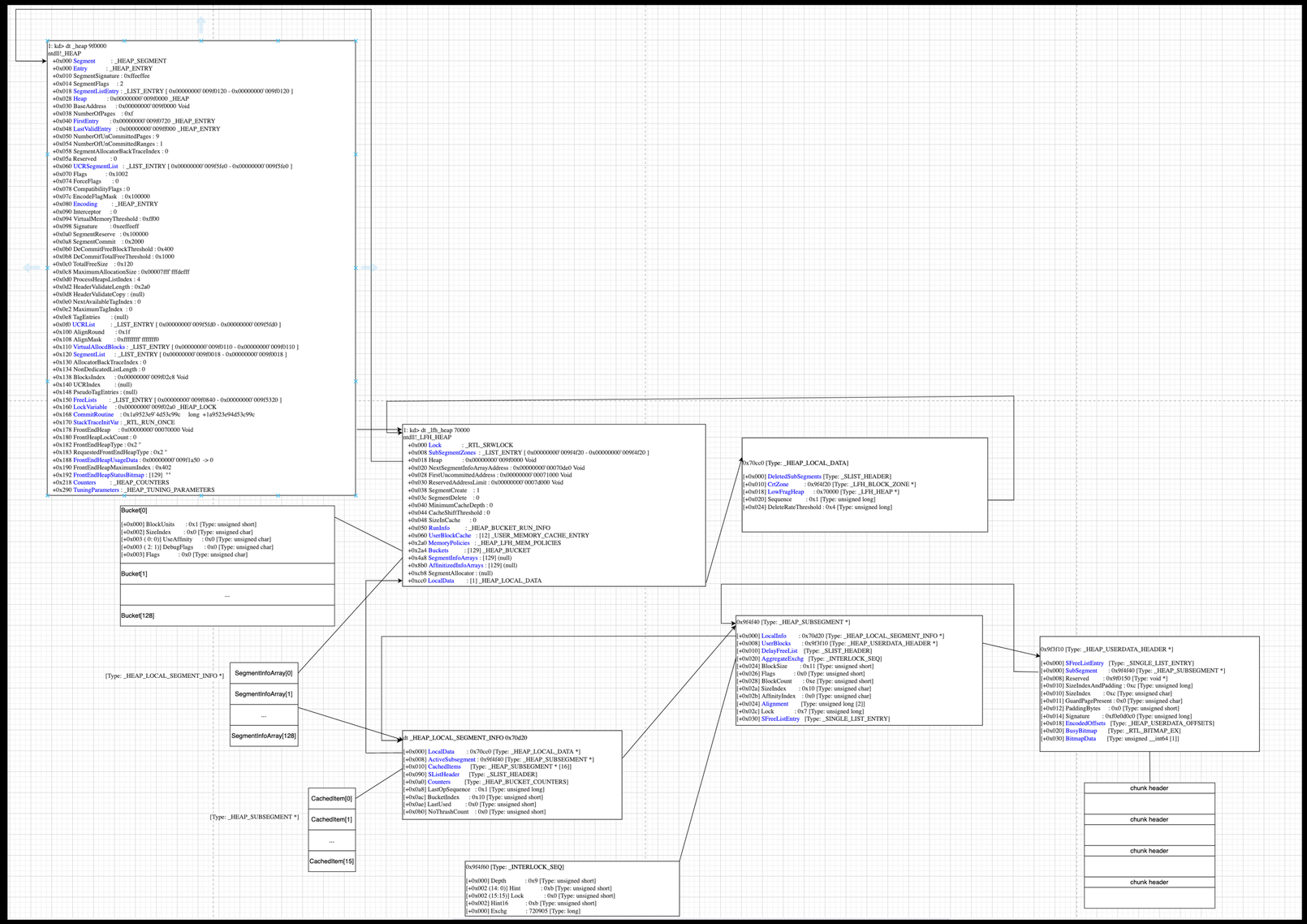
- After leaking
  - Write
    - Return address on stack
  - Control RIP
    - ROP to system(“cmd.exe”)
    - ROP to VirtualProtect/VirtualAlloc and jump to shell code

Windows memory allocator

- Nt Heap
  - FrontEnd
    - Data structure
    - Allocation mechanism

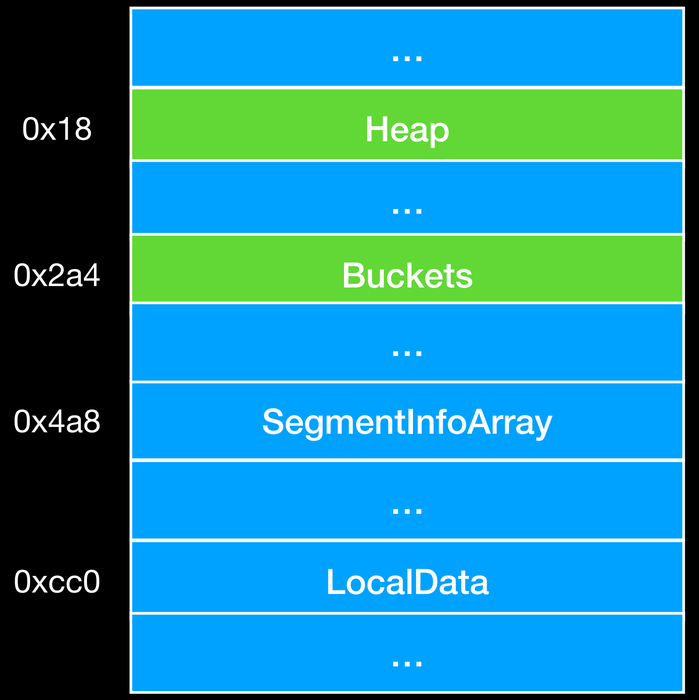
Windows Memory Allocator

- LFH Overview



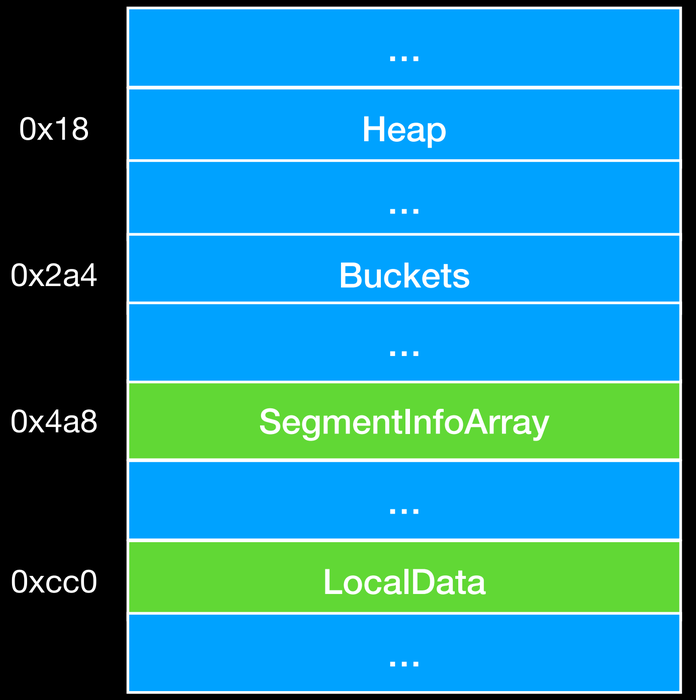
Windows Memory Allocator

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



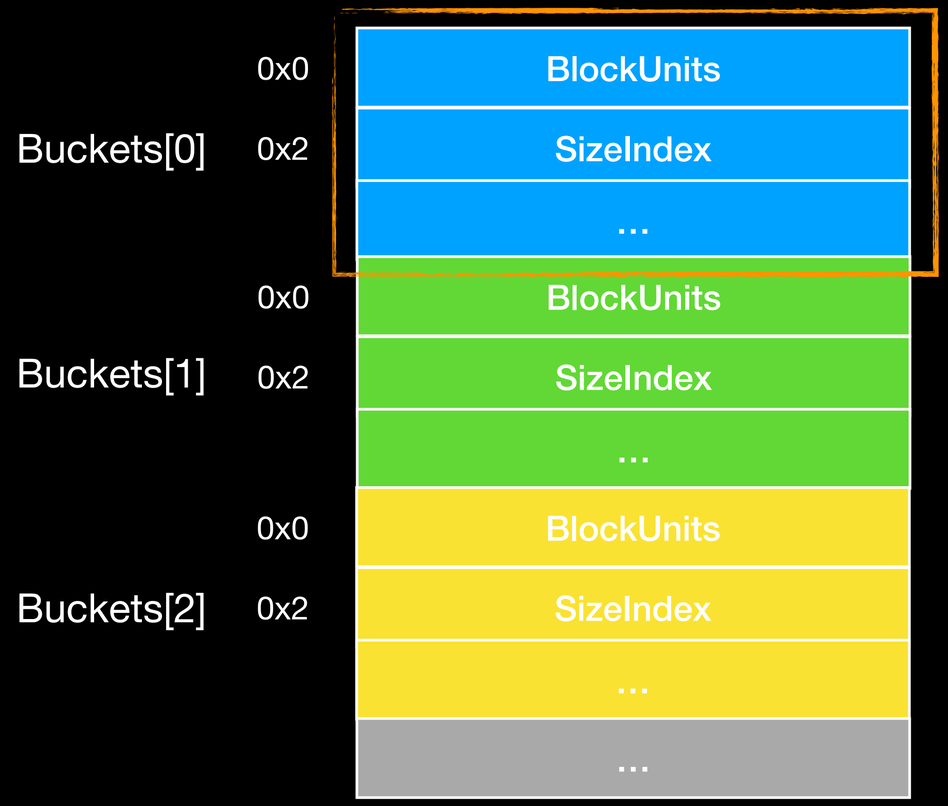
Windows Memory Allocator

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



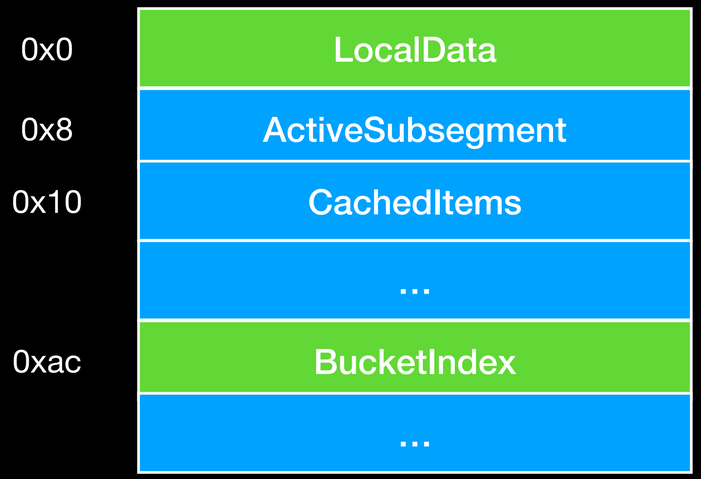
Windows Memory Allocator

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



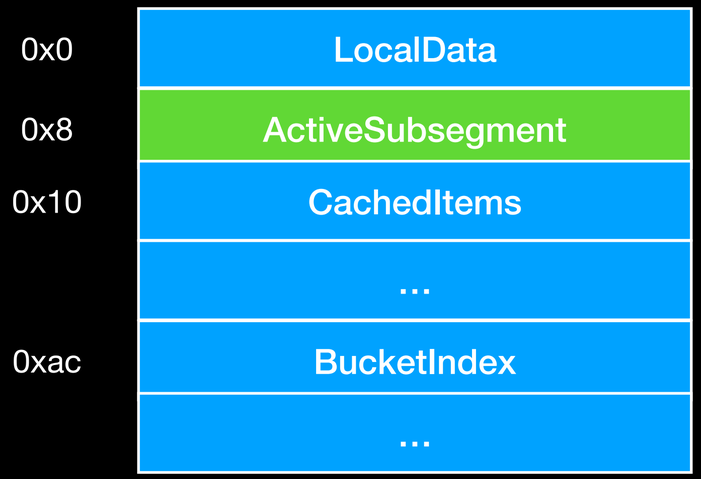
Windows Memory Allocator

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



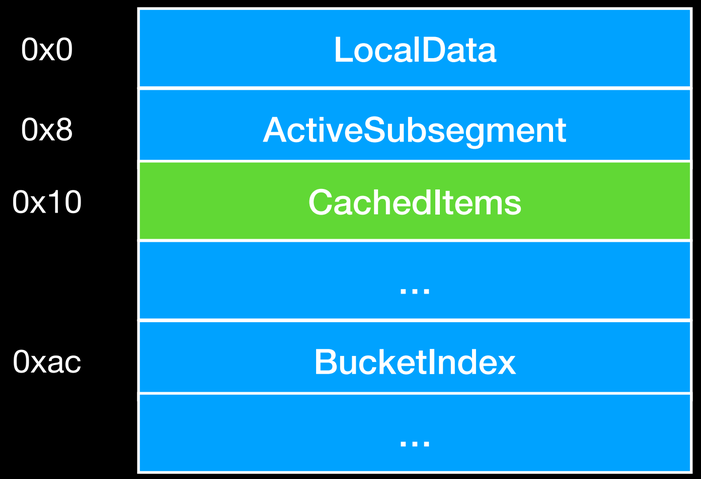
Windows Memory Allocator

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



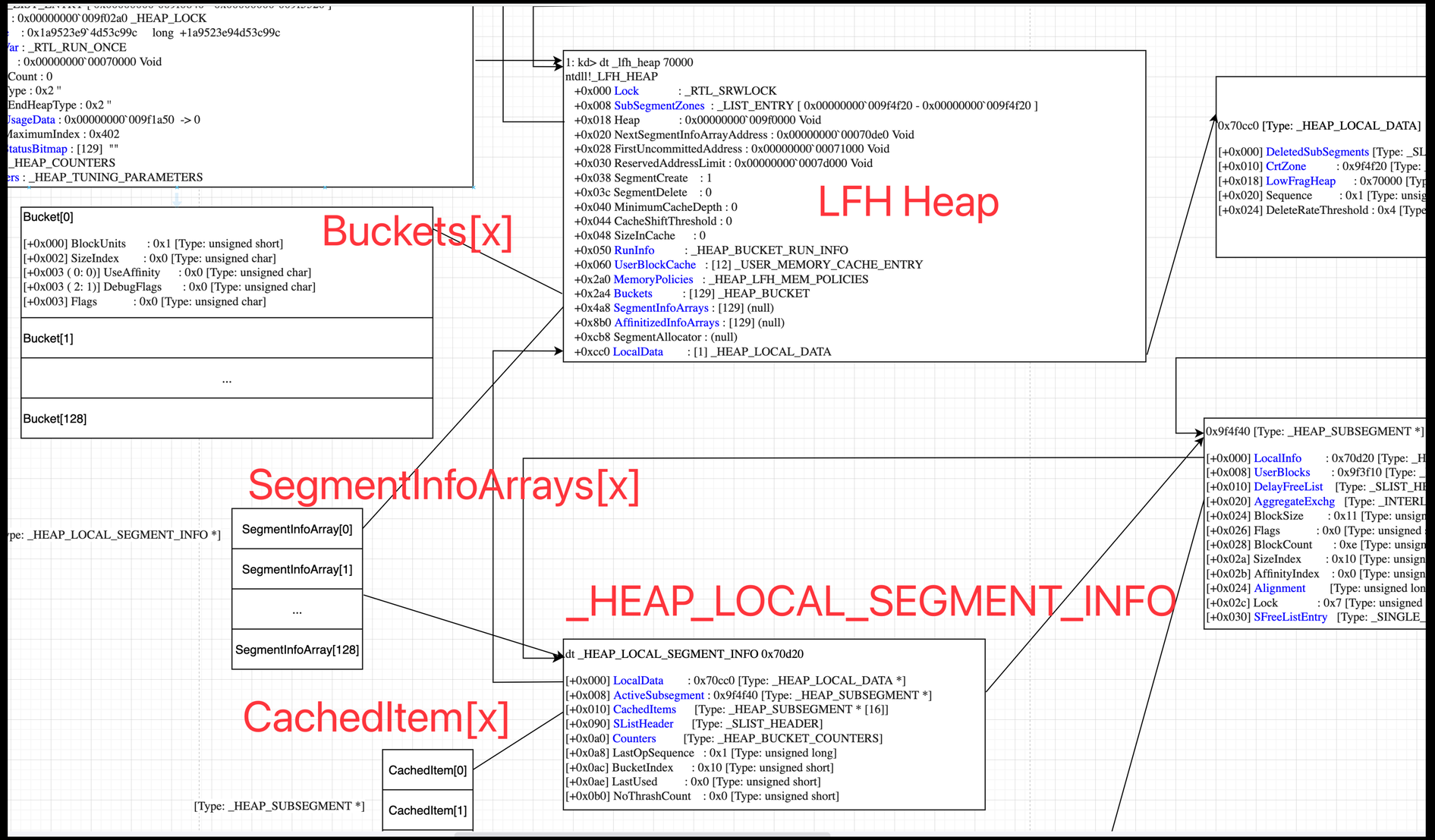
Windows Memory Allocator

- 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



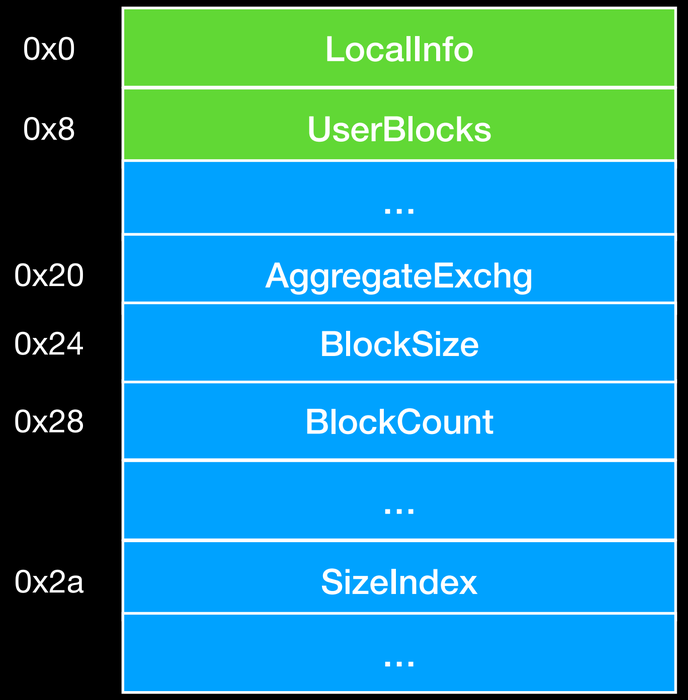
Windows Memory Allocator

LFH view1



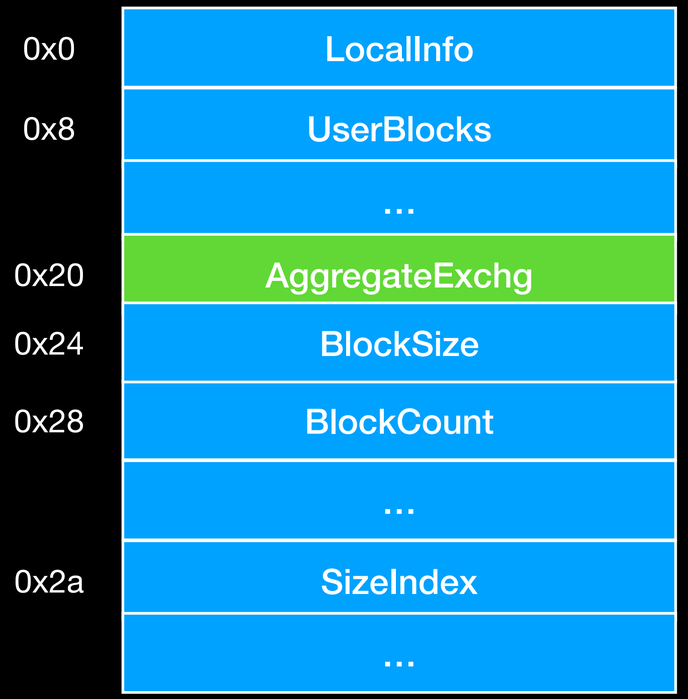
Windows Memory Allocator

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



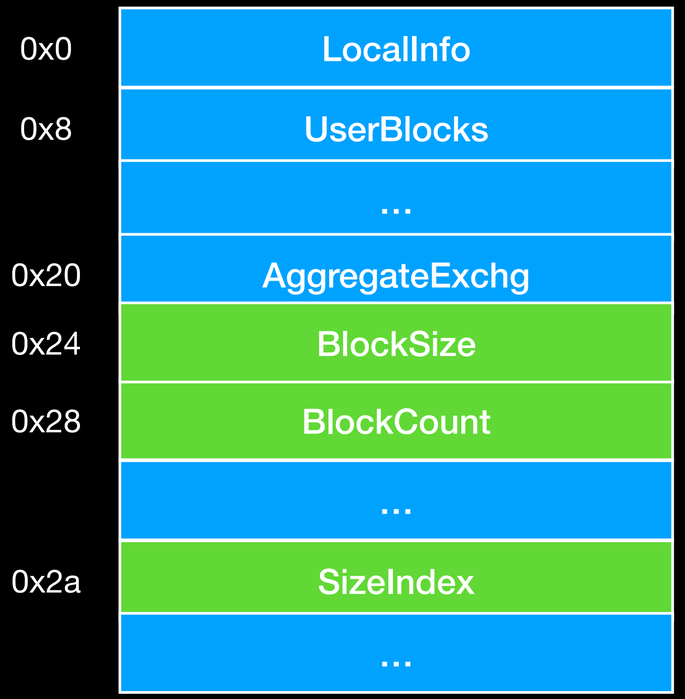
Windows Memory Allocator

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



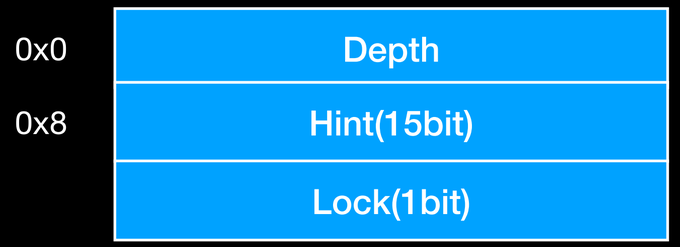
Windows Memory Allocator

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



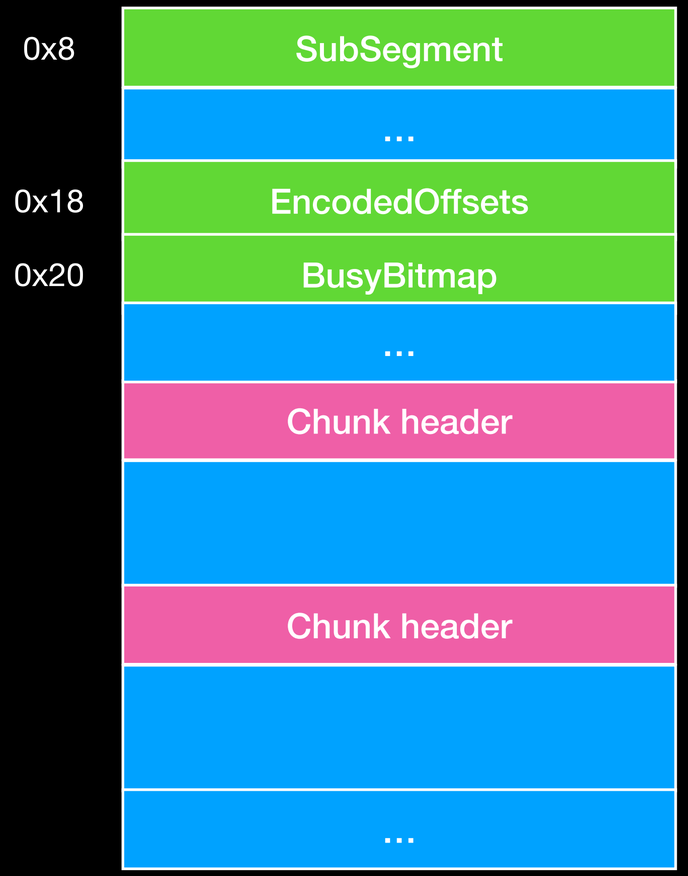
Windows Memory Allocator

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



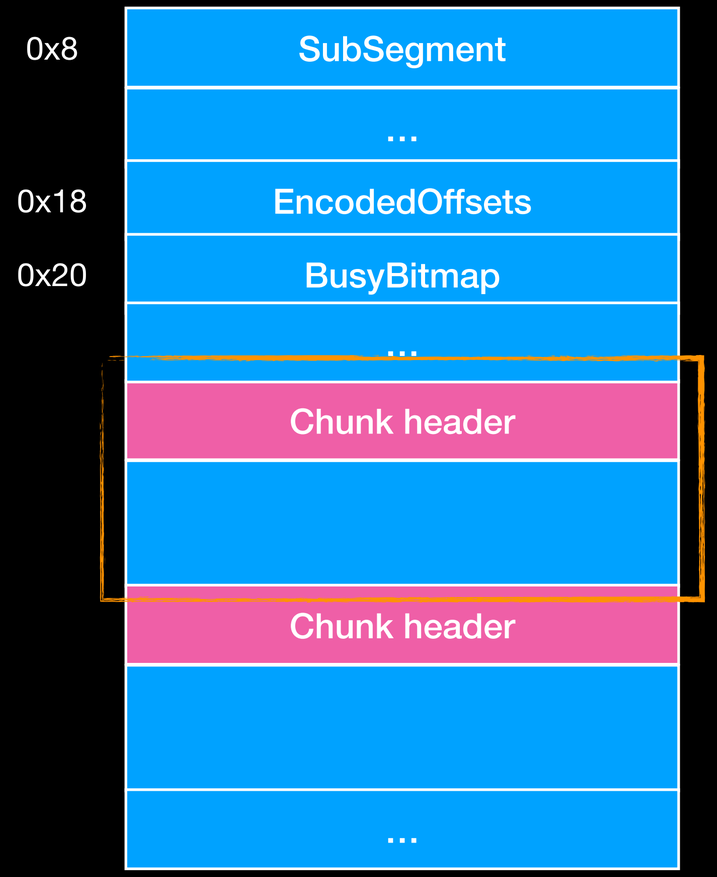
Windows Memory Allocator

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



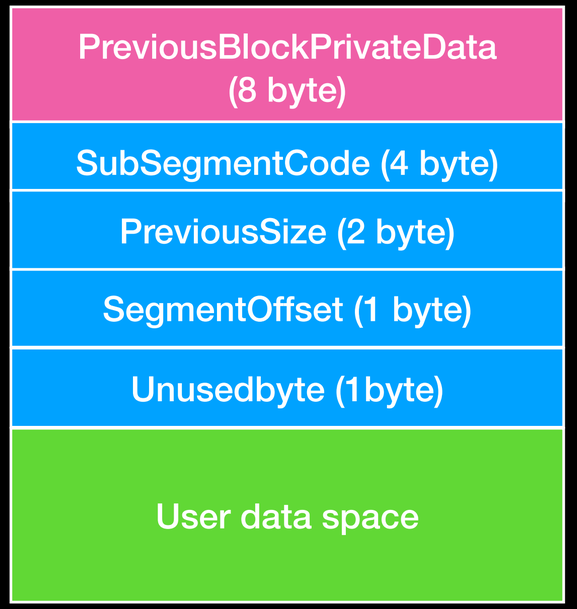
Windows Memory Allocator

- 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

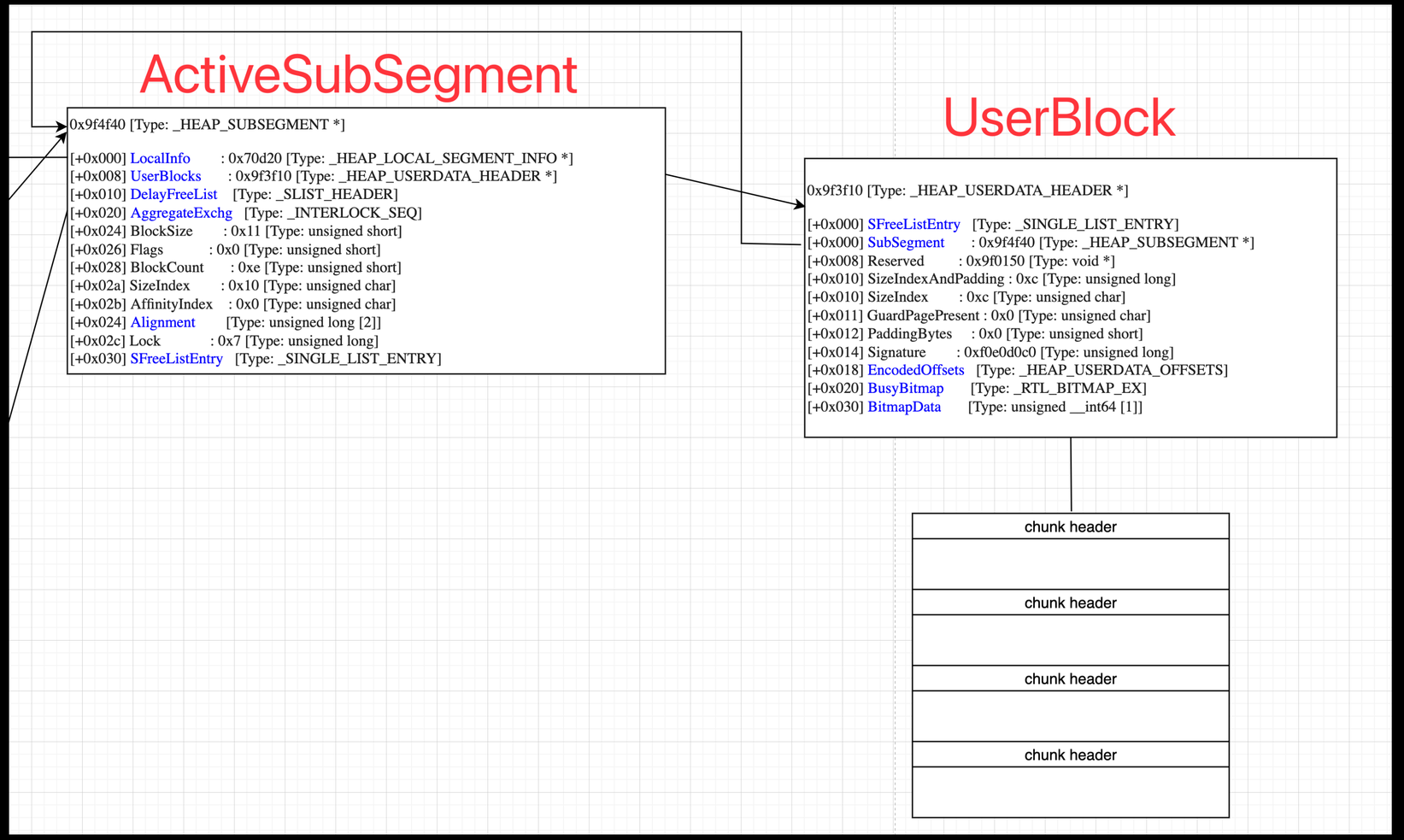
Windows memory allocator

- Remark
  - LFHkey

```
1  void *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     {
```

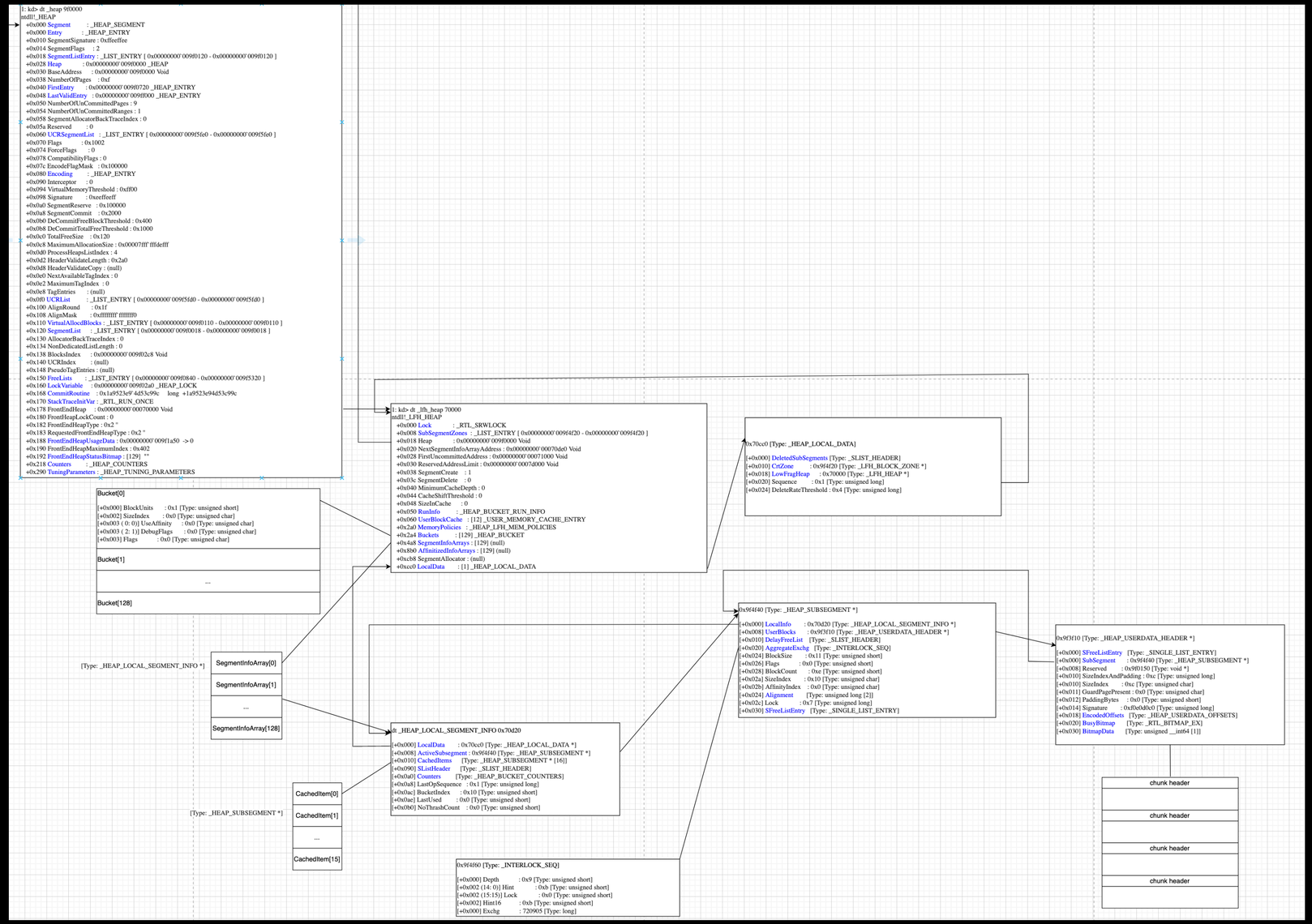
Windows Memory Allocator

LFH view2



Windows Memory Allocator

LFH Overview



Windows Memory Allocator

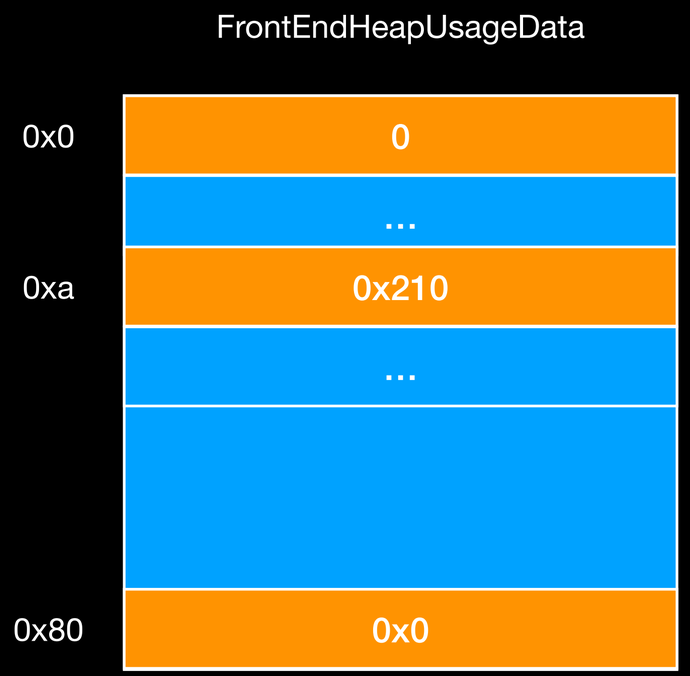
FrontEnd allocation mechanism

- 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

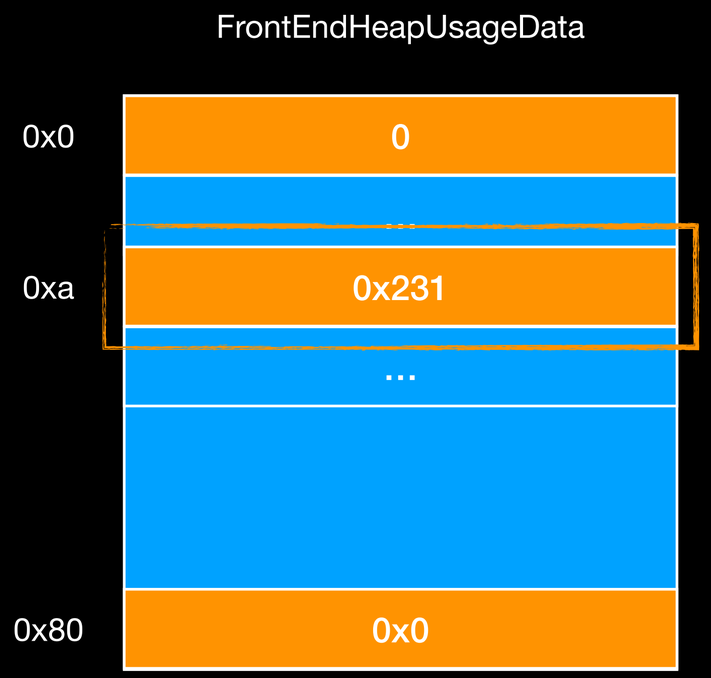
- LFH initialization
  - malloc(0x40) * 16
  - word elements



Windows Memory Allocator

FrontEnd allocation mechanism

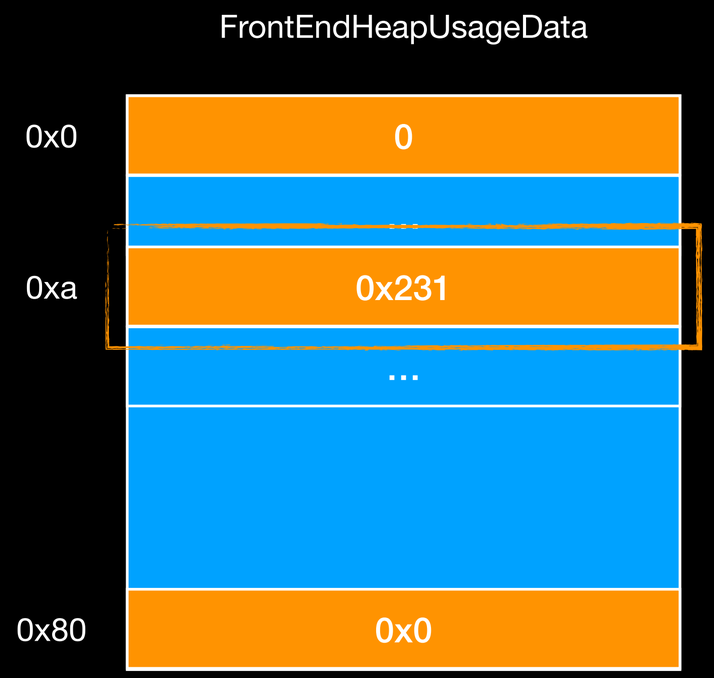
- LFH initialization
  - malloc(0x40) * 17
    - 0x231 & 0x1f > 0x10



Windows Memory Allocator

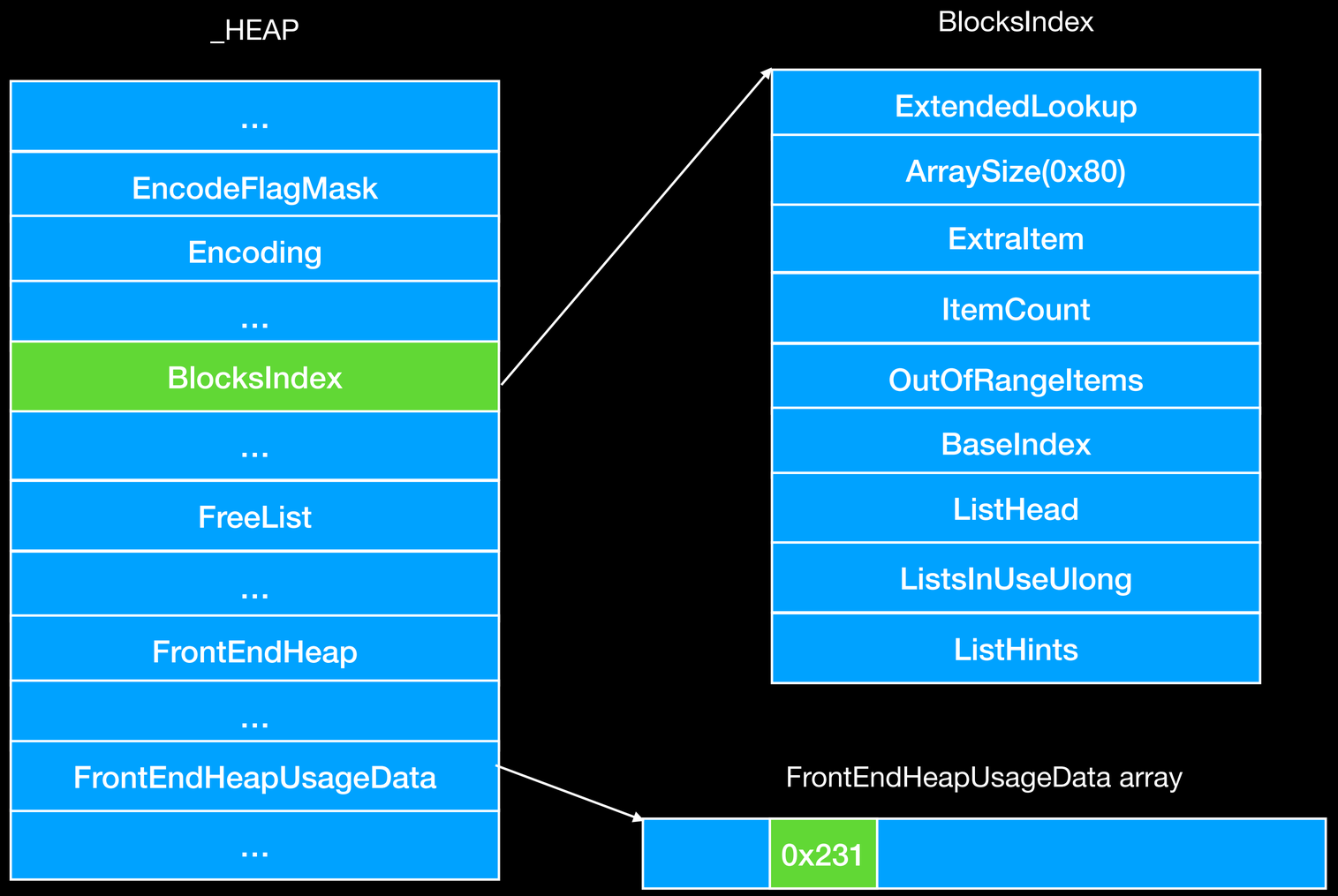
FrontEnd allocation mechanism

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



Windows Memory Allocator

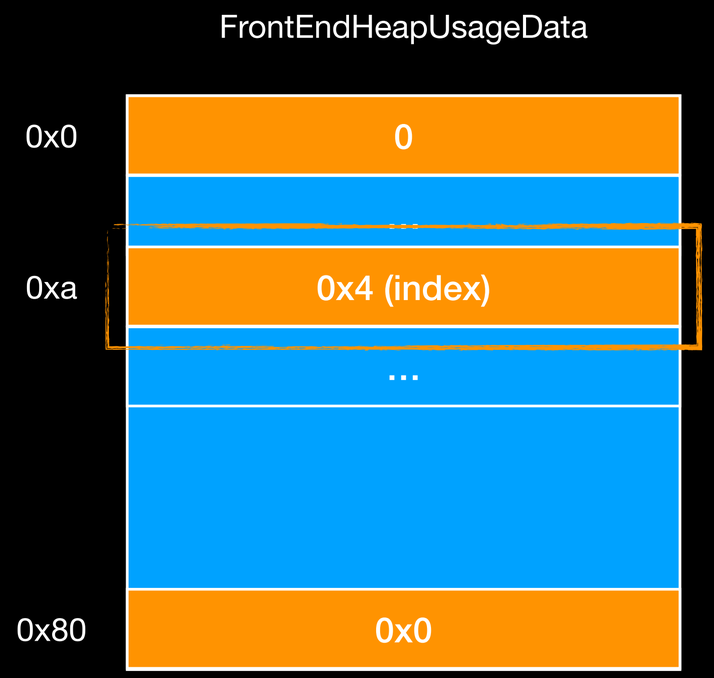
FrontEnd allocation mechanism



Windows Memory Allocator

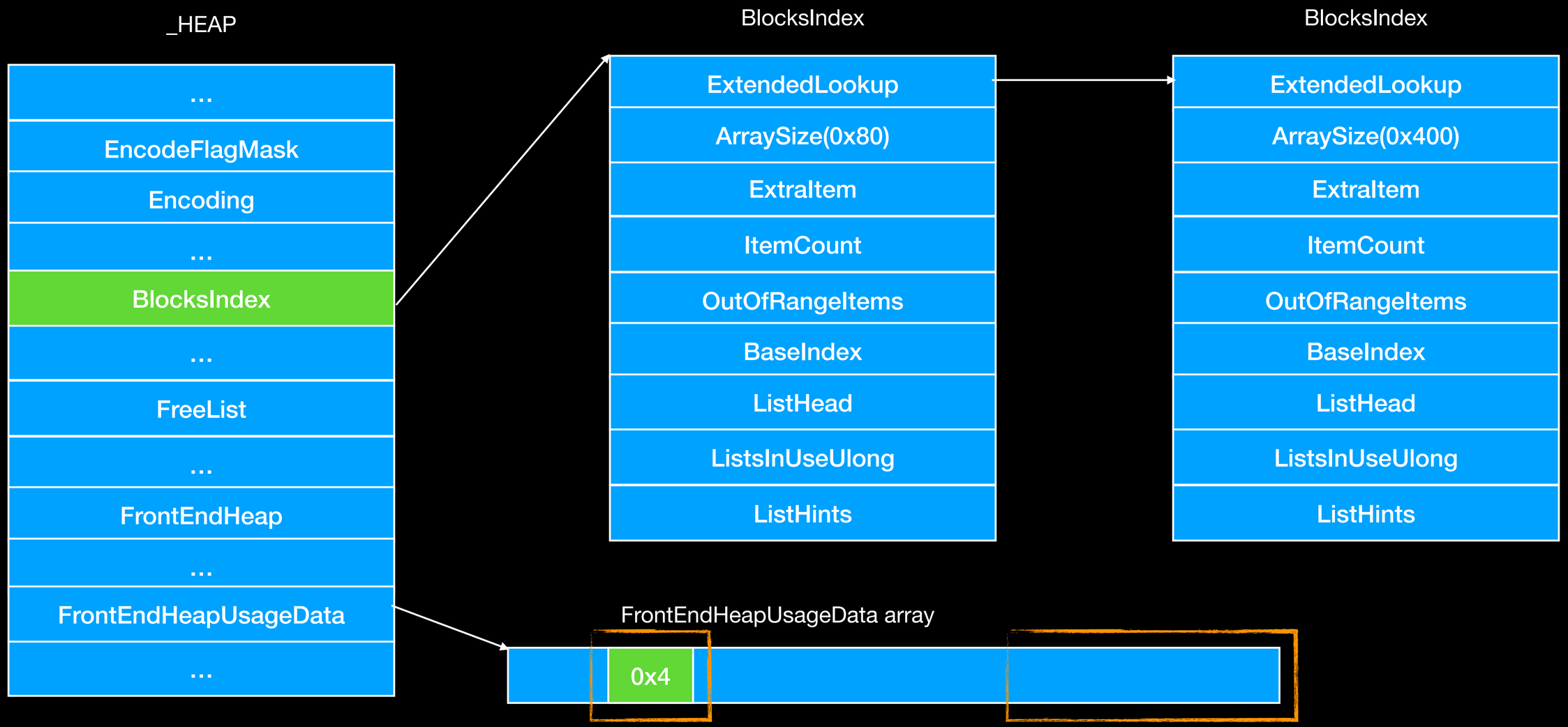
FrontEnd allocation mechanism

- 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

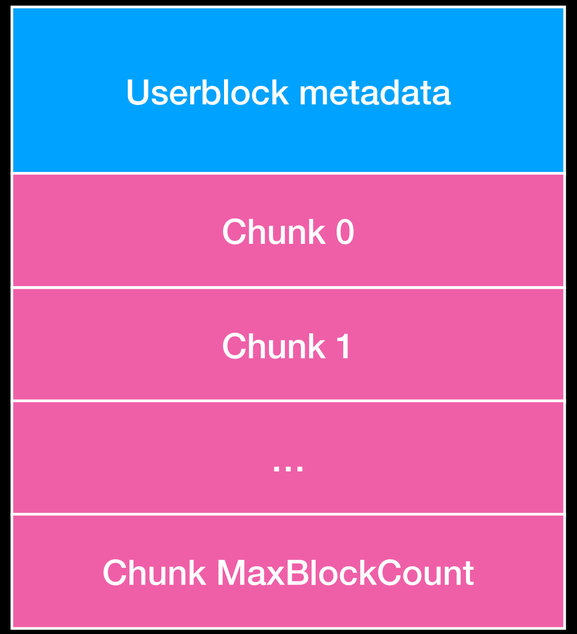
FrontEnd allocation mechanism



Windows Memory Allocator

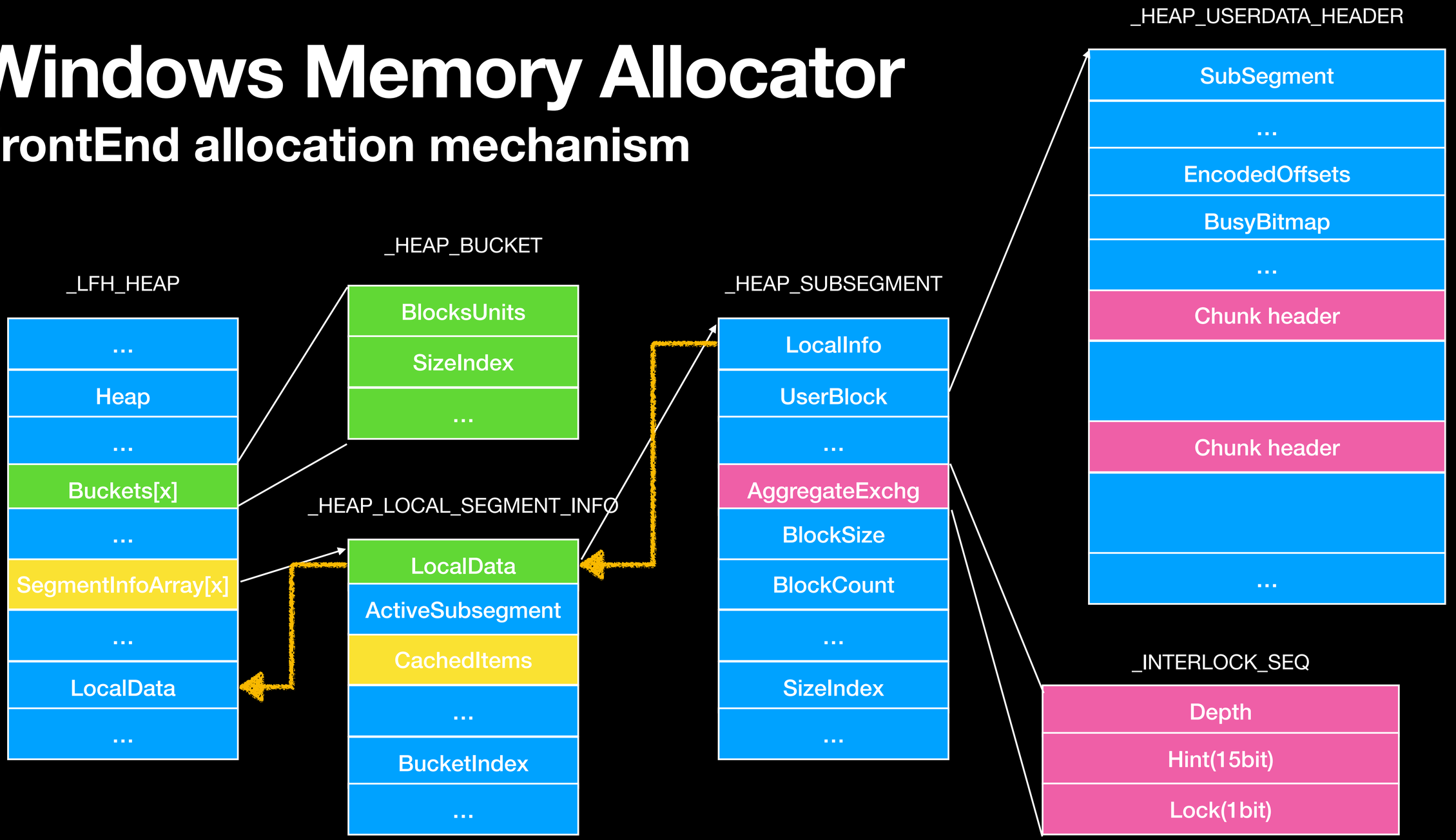
FrontEnd allocation mechanism

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



Windows Memory Allocator

FrontEnd allocation mechanism



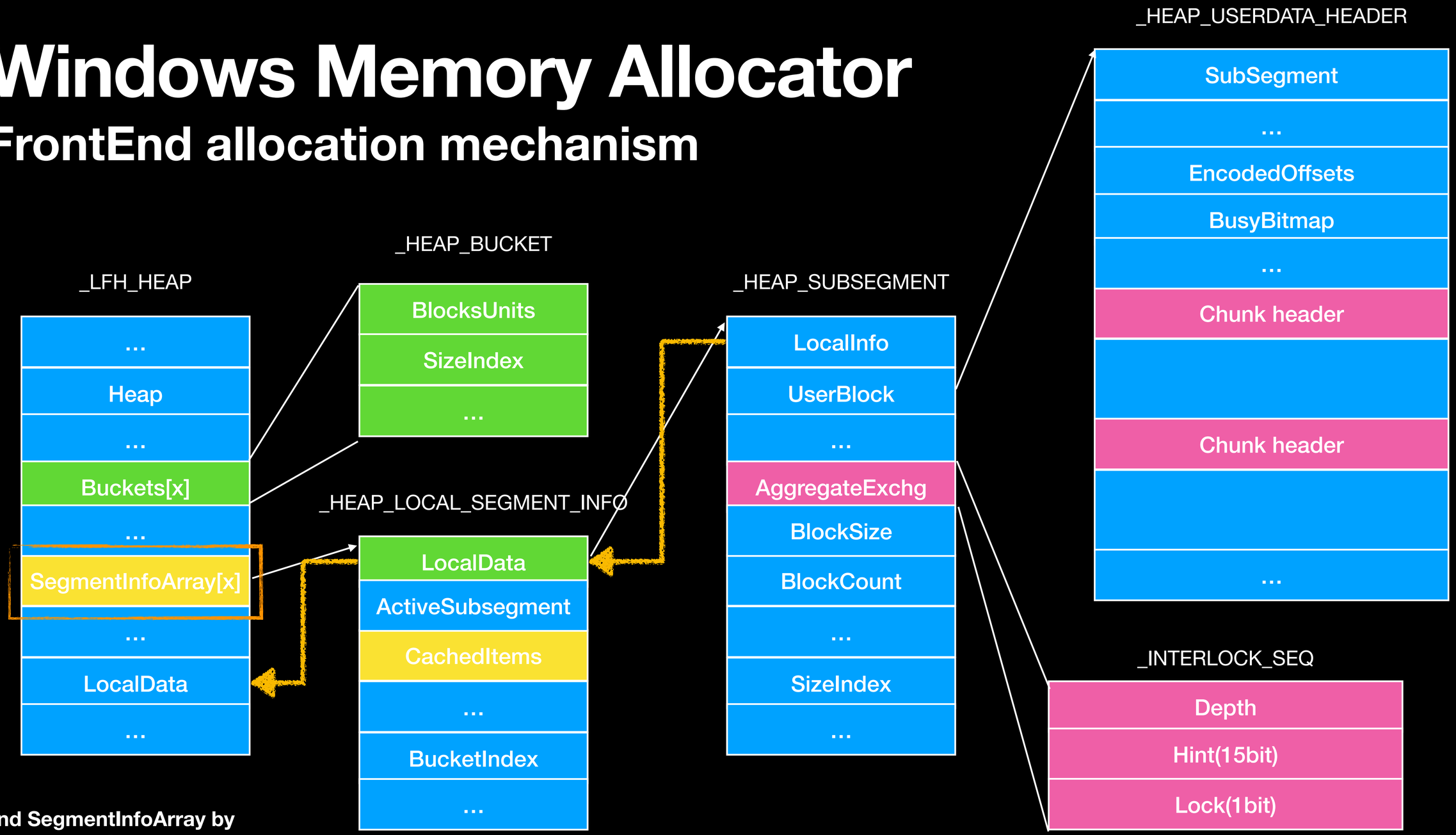
Windows Memory Allocator

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

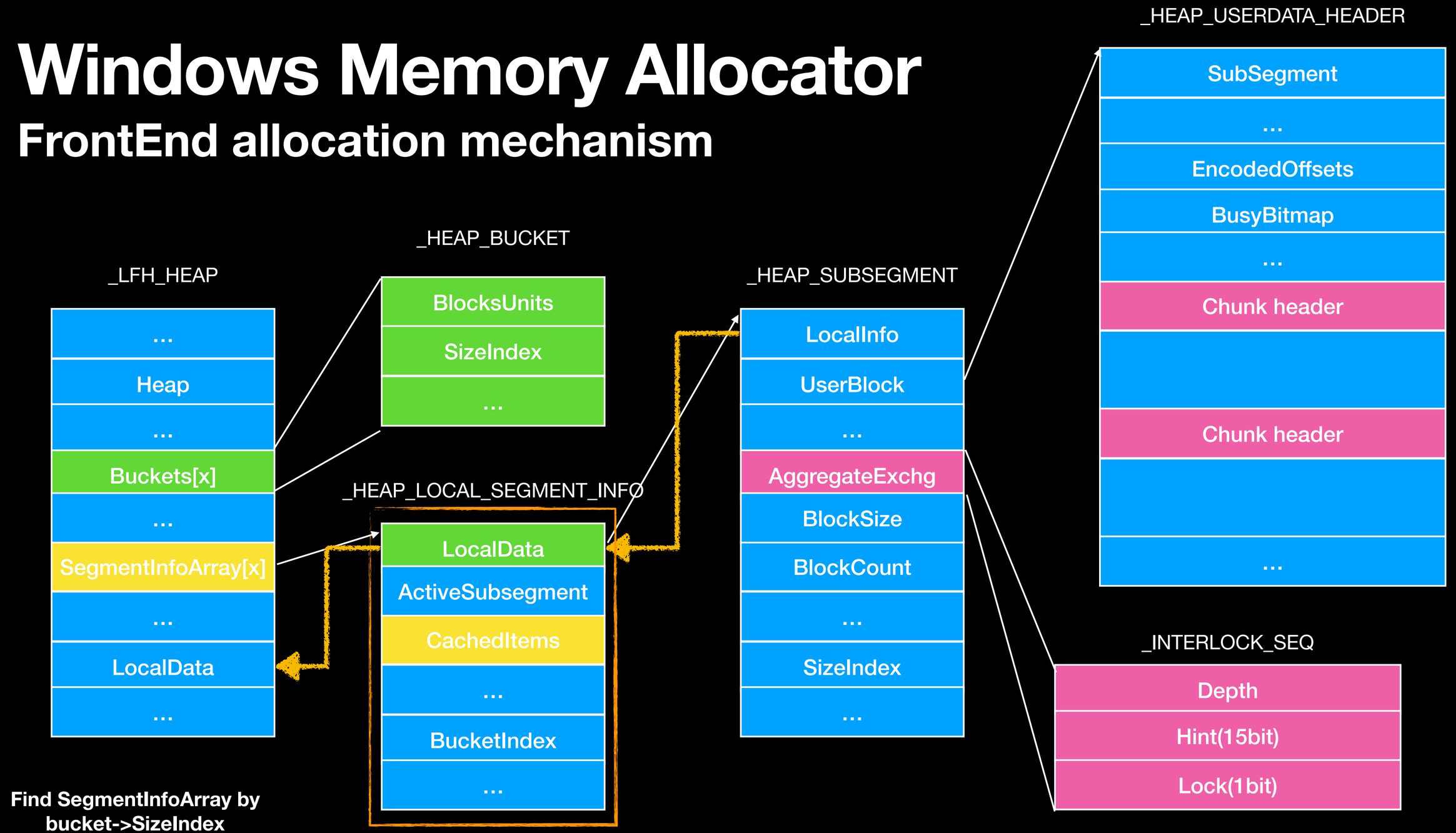
Windows Memory Allocator

FrontEnd allocation mechanism



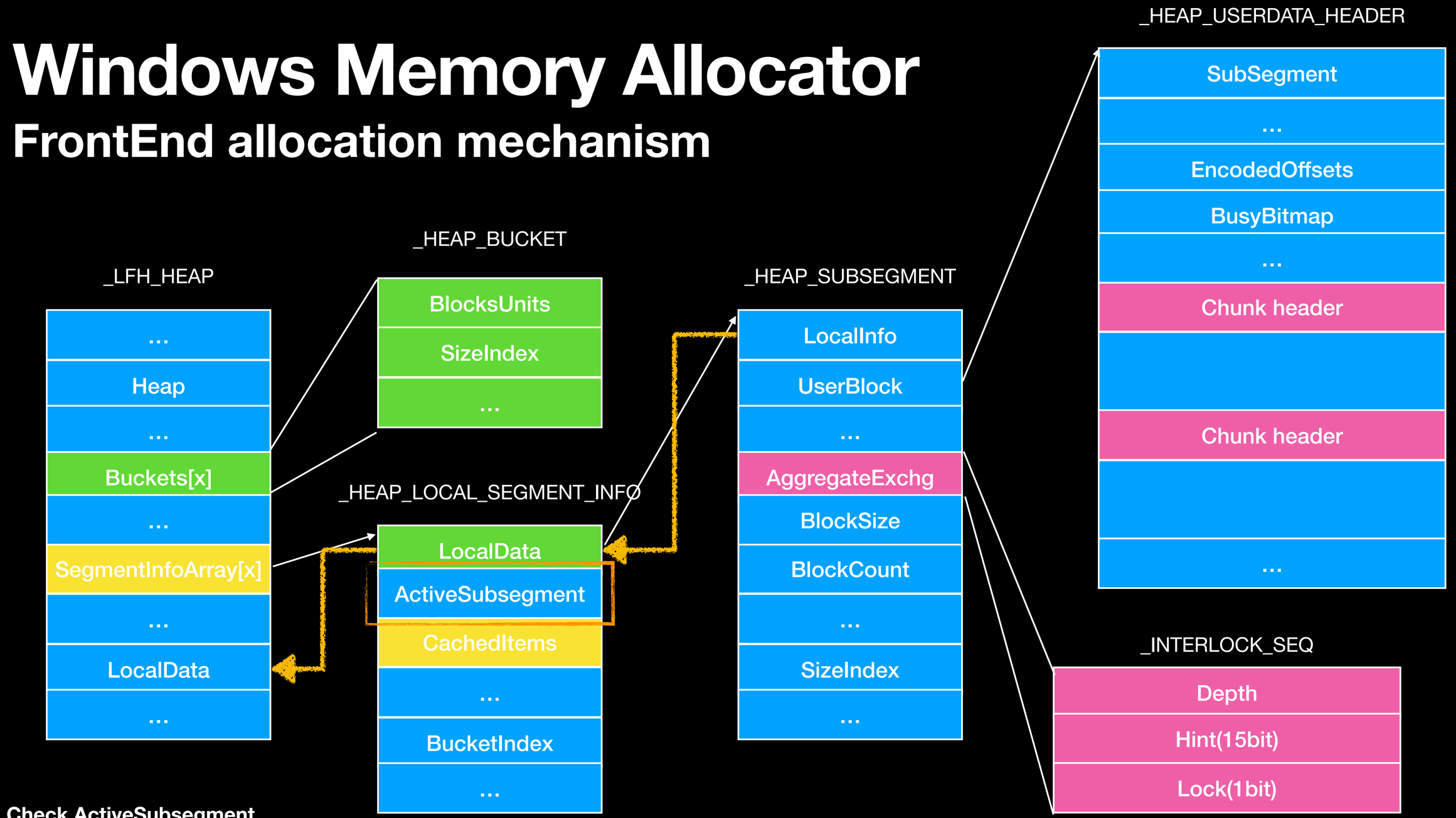
Windows Memory Allocator

FrontEnd allocation mechanism



Windows Memory Allocator

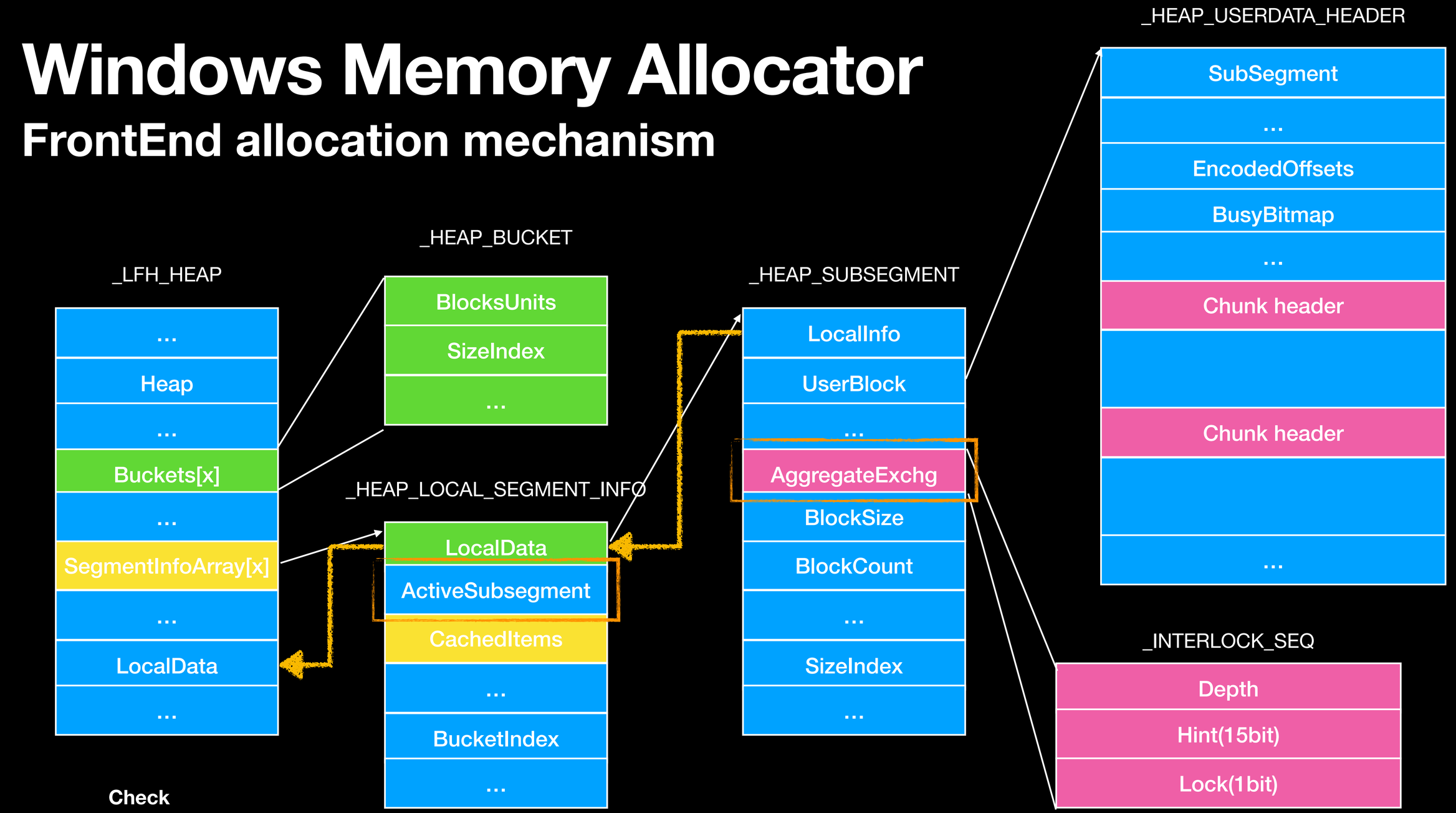
FrontEnd allocation mechanism



Check ActiveSubsegment

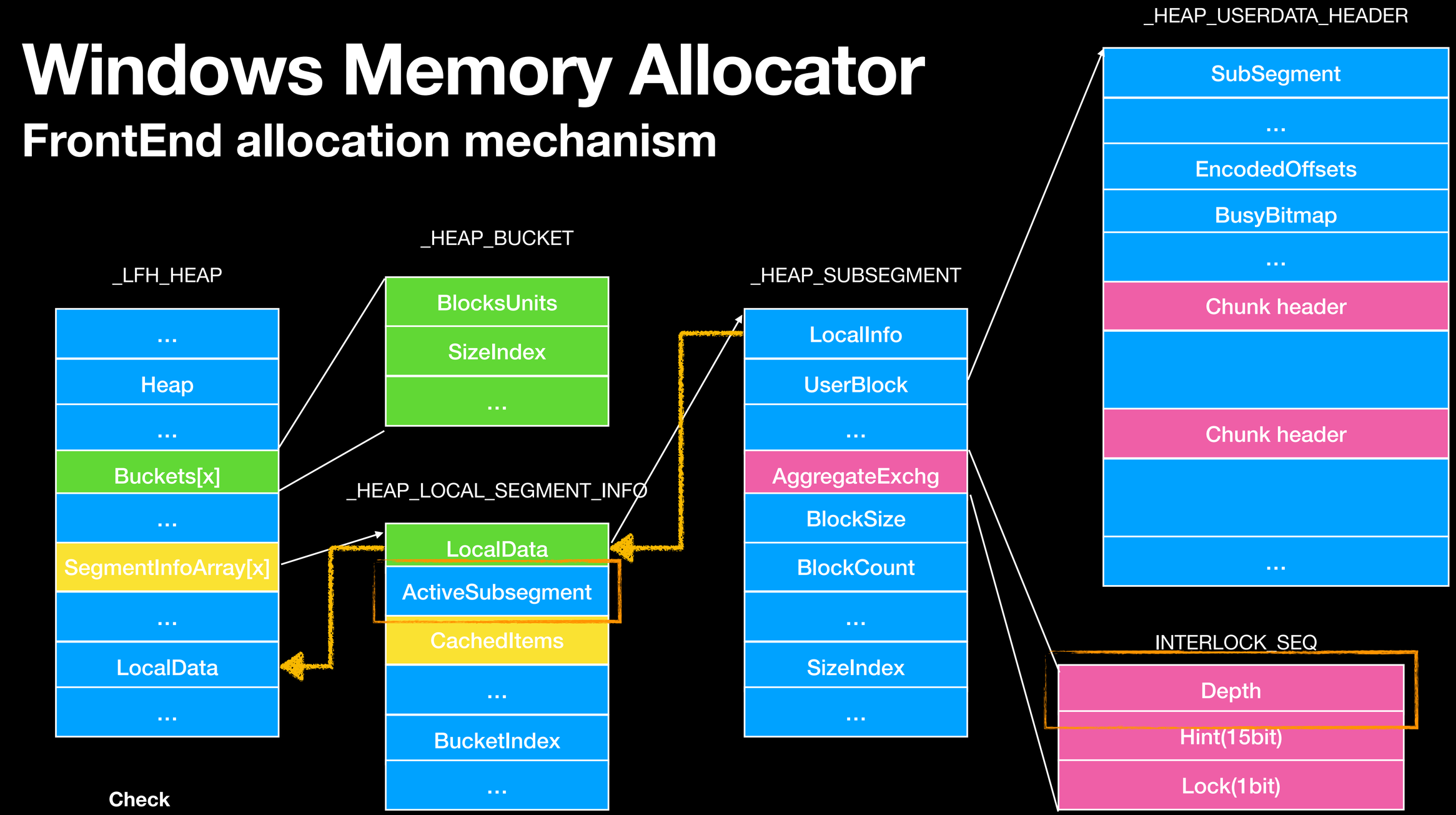
Windows Memory Allocator

FrontEnd allocation mechanism



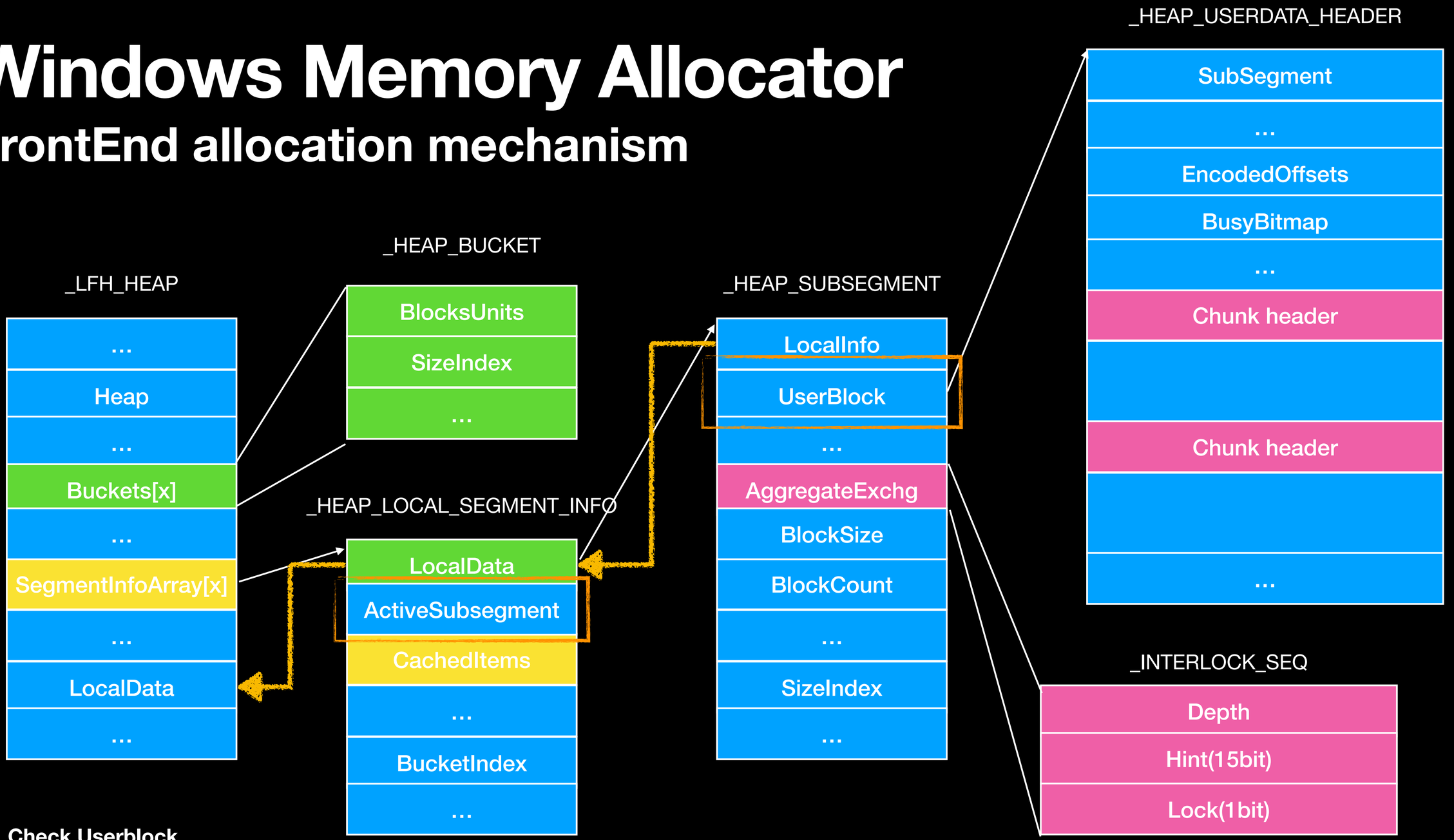
Windows Memory Allocator

FrontEnd allocation mechanism



Windows Memory Allocator

FrontEnd allocation mechanism



Windows Memory Allocator

FrontEnd allocation mechanism

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

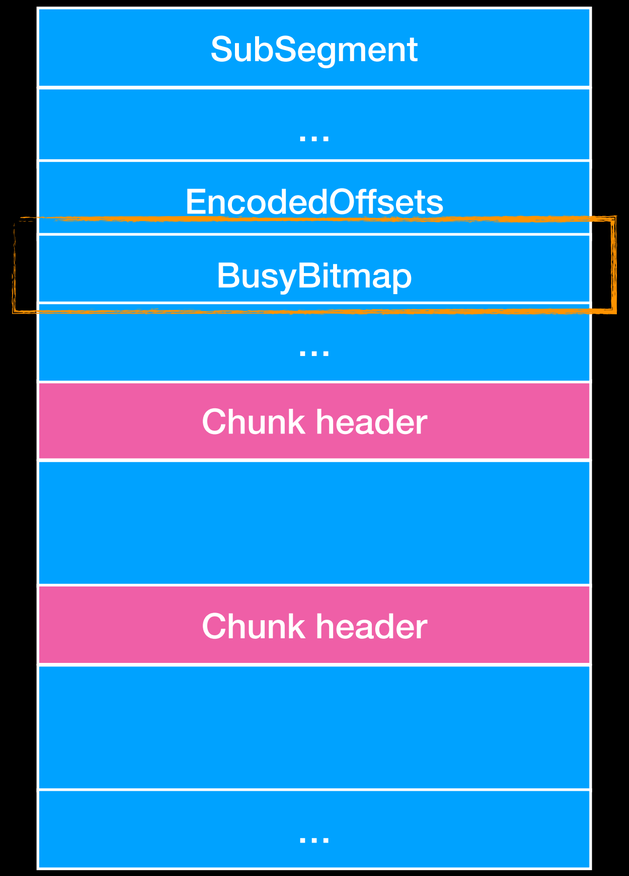
Windows Memory Allocator

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 = RtlpLowFragHeapRandomData[index] * maxidx >> 7
  - Check BusyBitmap[x]
    - If 0, take chunk
    - If 1, take next chunk
    - Config bitmap



Windows Memory Allocator

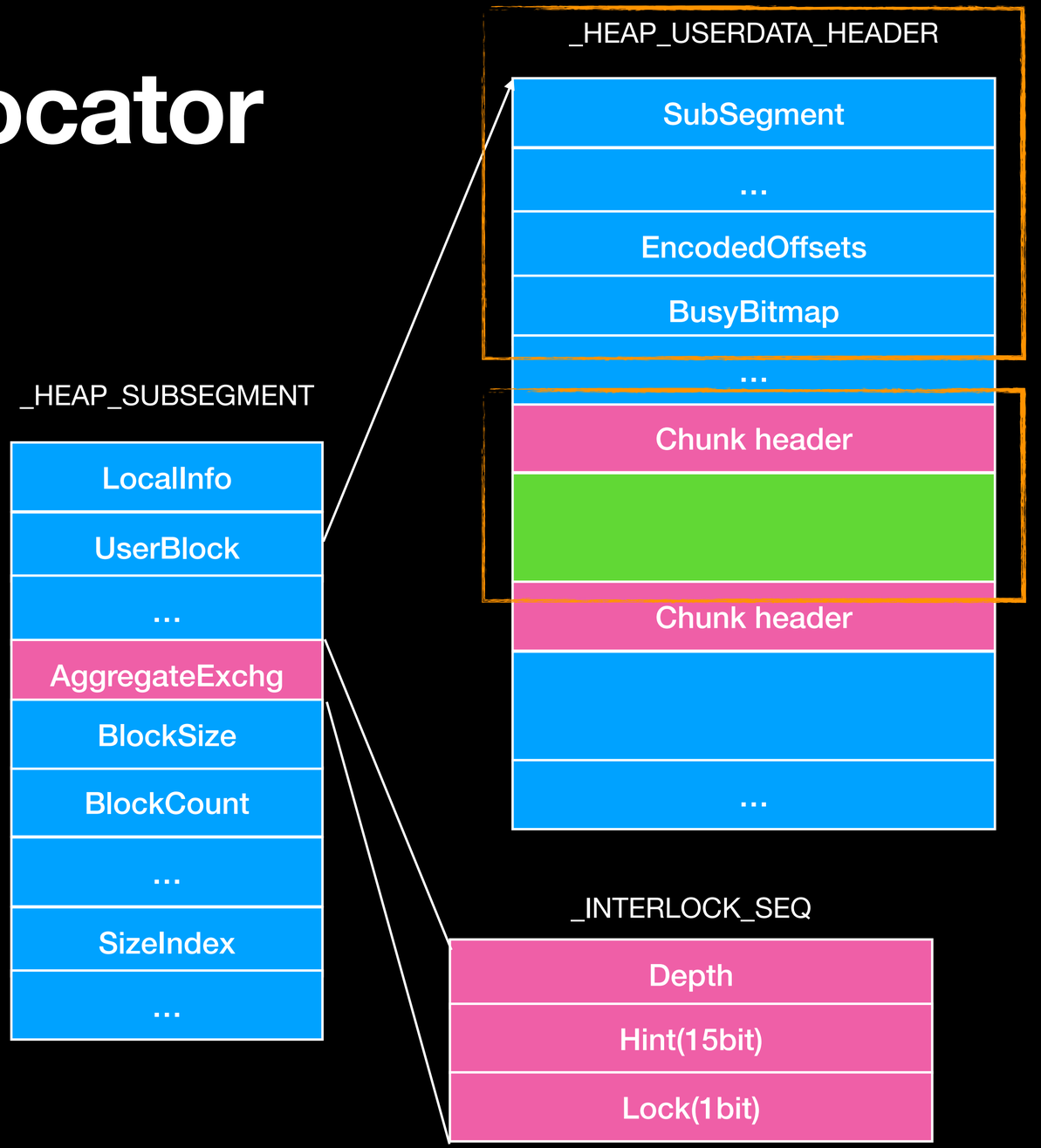
FrontEnd allocation mechanism

- 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

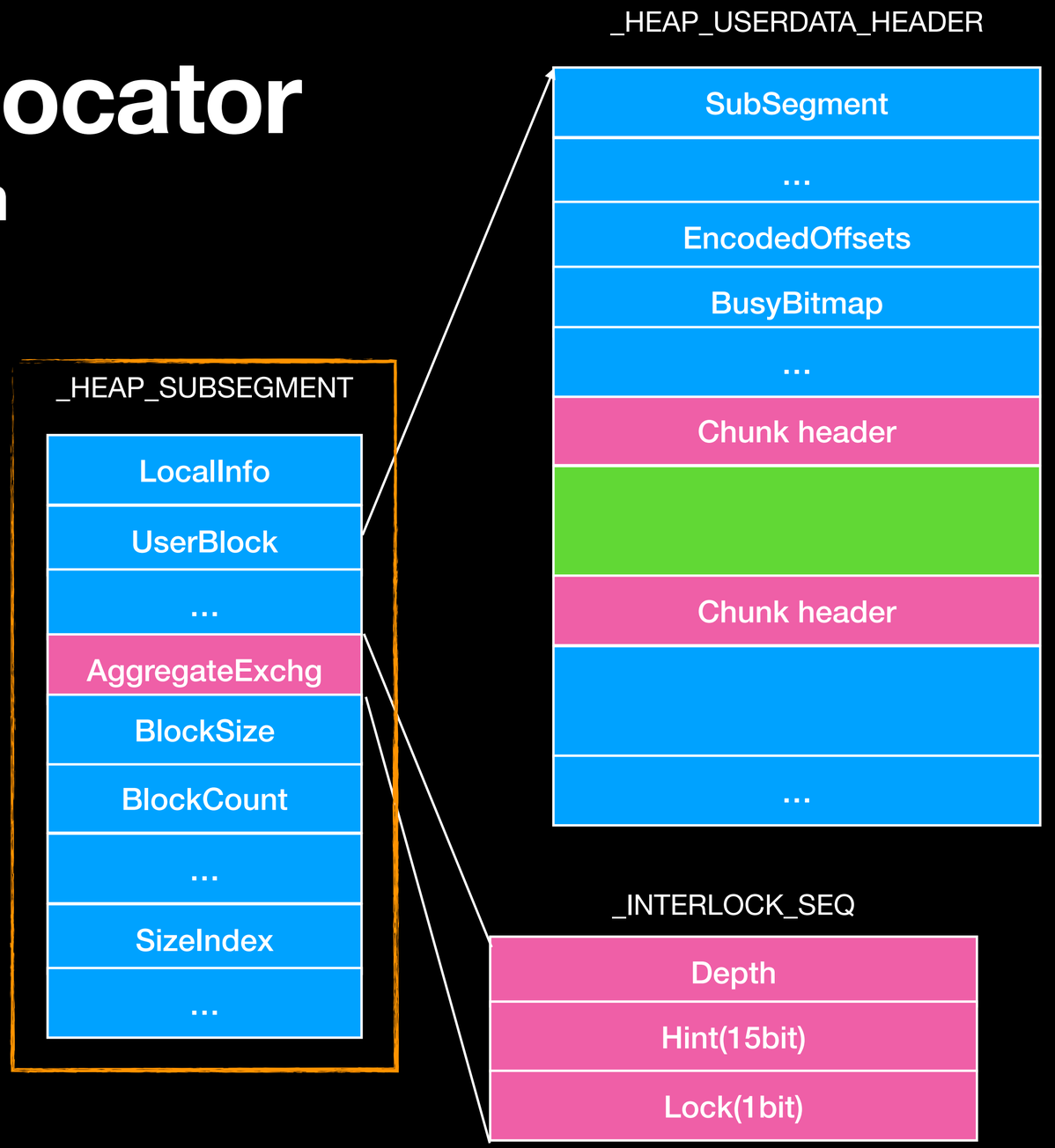
- Free
  - Chunk header->UserBlock



Windows Memory Allocator

FrontEnd allocation mechanism

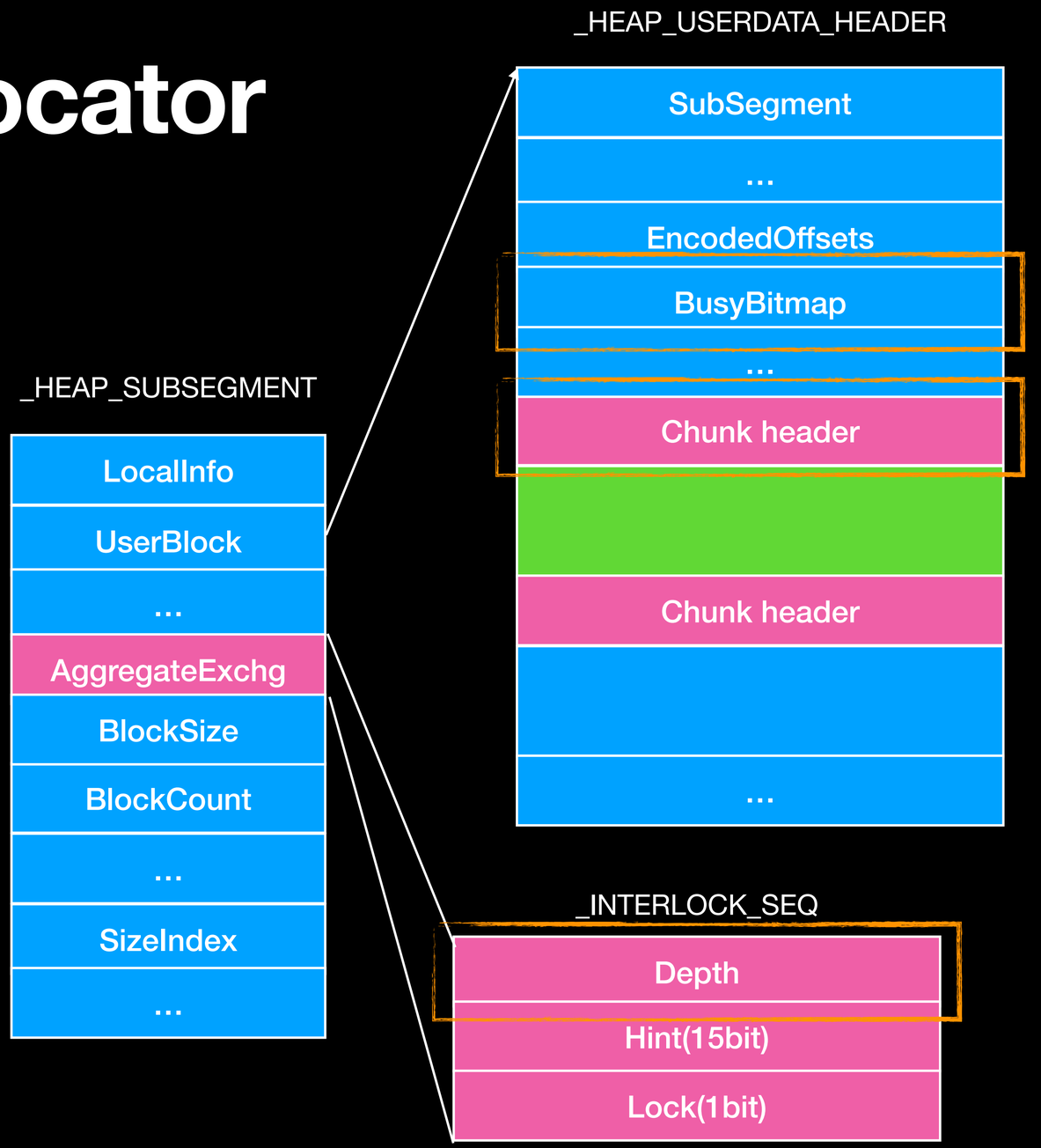
- Free
  - Chunk header->UserBlock
  - SubSegment



Windows Memory Allocator

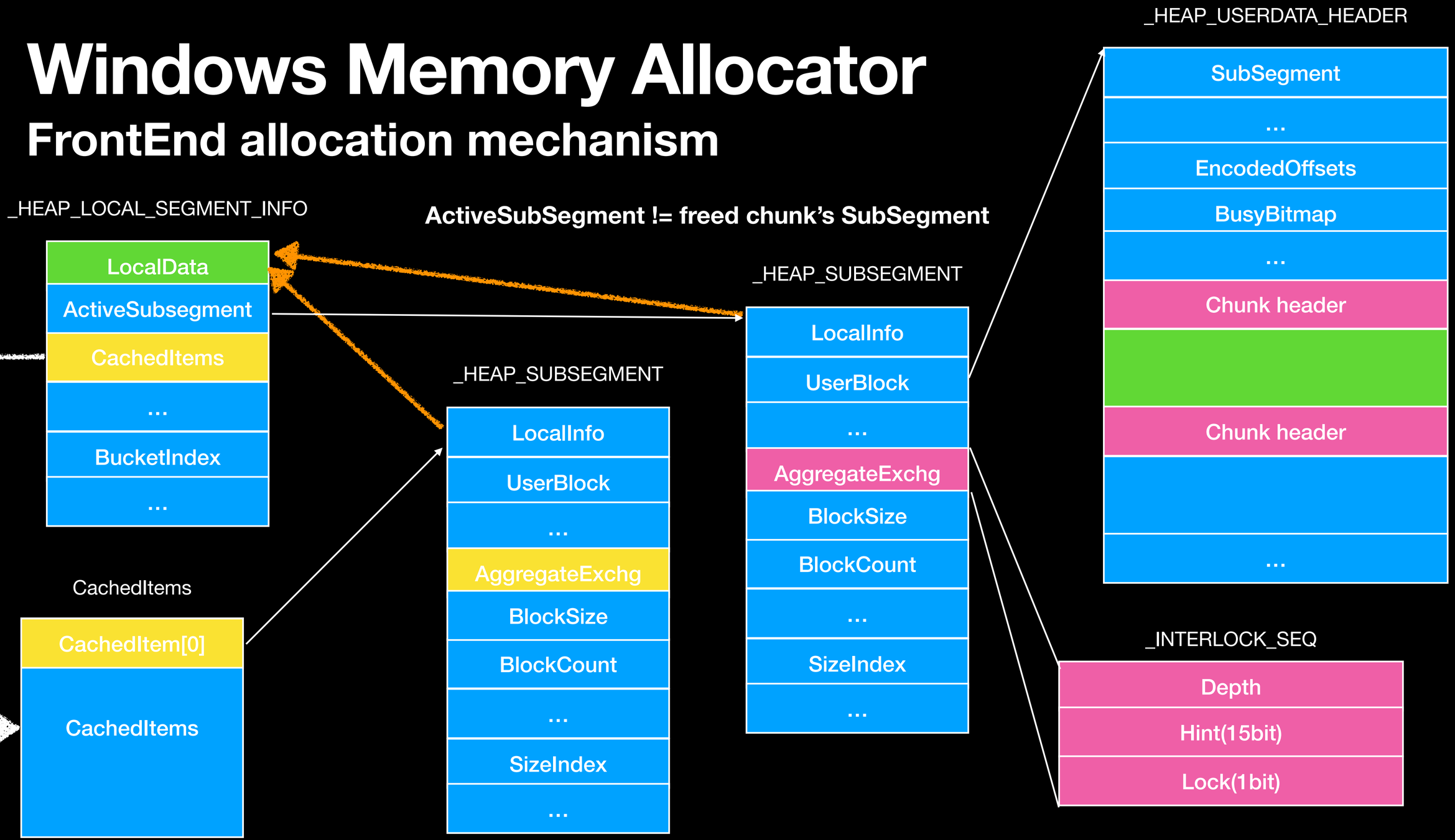
FrontEnd allocation mechanism

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



Windows Memory Allocator

FrontEnd allocation mechanism

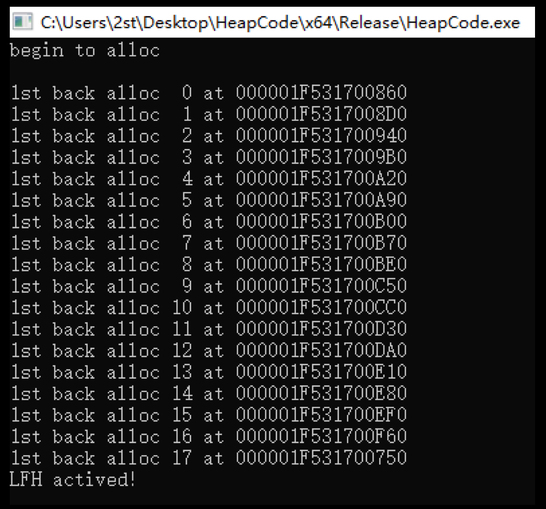


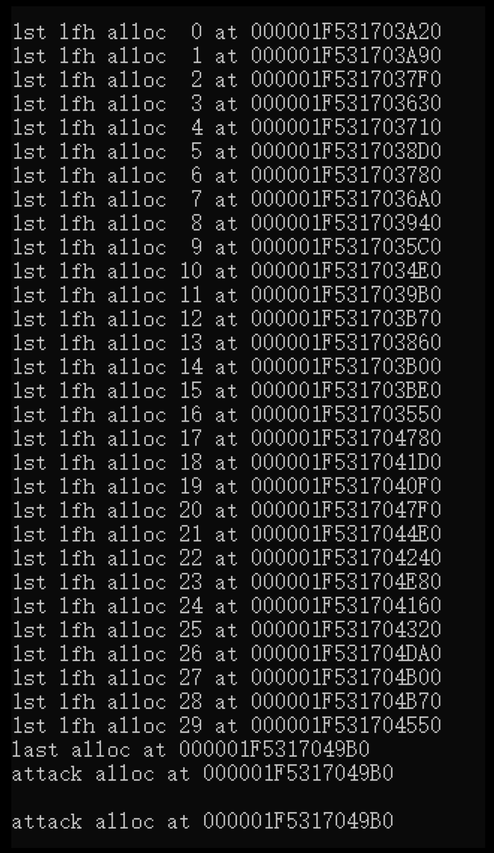
LFH Exploitation

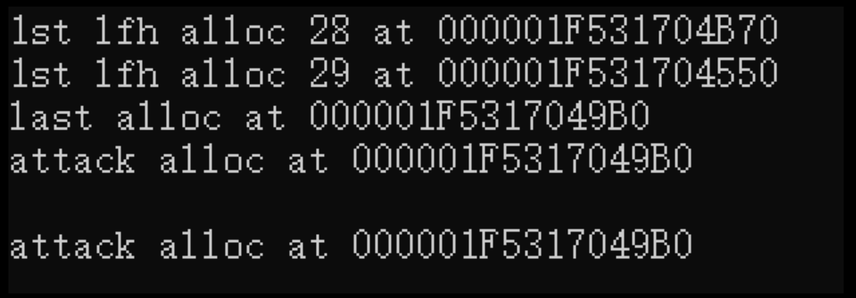
- Reuse attack
- indicate that we have UAF first
  - Due to LFH’s its randomness, we can not simply exploit memory by alloc/free
  - We can fill a UserBlock and free specific one, we can get the same LFH chunk at next allocation
  - First UserBlock has 32 chunks, Second has 64…

LFH Exploitation

- Reuse attack







Thanks