IBM Global Technology Services

Practical Windows XP SP3 / 2003 Heap Exploitation

Blackhat USA 2009

John McDonald
Christopher Valasek
IBM ISS X-Force Research

IBM Internet Security Systems
Ahead of the threat.™

IBM Confidential

Introduction

All the fun of debugging subtle race conditions, without all the tedium of earning an honest living.

Introduction

- **Heap Exploitation used to be only Internet mildly hard™**
  - mov [ecx], eax
  - mov [eax+4], ecx
  - No PEB randomization
  - Could easily lead to arbitrary DWORD overwrites

- **For some reason, they made it harder**
  - Safe Un-linking
  - Heap Cookie
  - Slightly Randomized PEB
  - Programs becoming more multi-threaded
  - Vista

History

- **Windows has strong tradition of technical heap research**
  - Well, until today

- **Matt and Oded’s 4-5 talks**
  - Homework: try to track down all the different versions

- **Brett’s BH06 talk is literally 17 times better than this one**

  Very interesting details if you pay attention

- **Ben Hawkes Vista tour de force**

- **Nico Waisman is *really* good at Internet**
  - Nuff said

Things we will cover…

- **Memory Management Foundations**
  - Core data structures
  - Core algorithms
  - Security mechanisms
- **Tactics**
  - Lookaside list link overwrite
  - Bitmap attacks
  - Exploiting Freelist[0]
  - Lookaside list exception handler
  - Heap cache exploitation
- **Strategy**
- **Demo**

Fundamentals

“If ntdll ever changes, I quit.”
- Christopher Valasek

Heap Base

- **Heap Base**
  - The heap base is a 0x588 byte data structure that is at the beginning of every Windows XPSP3/2003 heap
  - Used to keep track of the memory currently being managed
  - It contains vital information about data structure identifiers, how to handle memory requests, free memory chunks, and much more
  - The following slide is a dump of the debugging information for the default Windows heap in a sample application running under XP SP3:

Heap Base Dump

```
0:001> dt _HEAP 150000
ntdll!_HEAP
   +0x000 Entry            : _HEAP_ENTRY
   +0x008 Signature        : 0xeeffeeff
   +0x00c Flags            : 2
   +0x010 ForceFlags       : 0
   +0x014 VirtualMemoryThreshold : 0xfe00
   +0x018 SegmentReserve   : 0x100000
   +0x01c SegmentCommit    : 0x2000
   +0x020 DeCommitFreeBlockThreshold : 0x200
   +0x024 DeCommitTotalFreeThreshold : 0x2000
   +0x028 TotalFreeSize    : 0x68
   +0x02c MaximumAllocationSize : 0x7ffdefff
   +0x030 ProcessHeapsListIndex : 1
   +0x032 HeaderValidateLength : 0x608
   +0x034 HeaderValidateCopy : (null)
   +0x038 NextAvailableTagIndex : 0
   +0x03a MaximumTagIndex  : 0
   +0x03c TagEntries       : (null)
   +0x040 UCRSegments      : (null)
   +0x044 UnusedUnCommittedRanges : 0x00150598 _HEAP_UNCOMMMTTED_RANGE
   +0x048 AlignRound       : 0xf
   +0x04c AlignMask        : 0xfffffff8
   +0x050 VirtualAllocdBlocks : _LIST_ENTRY [ 0x150050 - 0x150050 ]
   +0x058 Segments         : [64] 0x00150640 _HEAP_SEGMENT
   +0x158 u                : __unnamed
   +0x168 u2               : __unnamed
   +0x16a AllocatorBackTraceIndex : 0
   +0x16c NonDedicatedListLength : 0
   +0x170 LargeBlocksIndex : (null)
   +0x174 PseudoTagEntries : (null)
   +0x178 FreeLists        : [128] _LIST_ENTRY [ 0x150178 - 0x150178 ]
   +0x578 LockVariable     : 0x00150608 _HEAP_LOCK
   +0x57c CommitRoutine    : (null)
   +0x580 FrontEndHeap     : 0x00150688
   +0x584 FrontHeapLockCount : 0
   +0x586 FrontEndHeapType : 0x1  ''
   +0x587 LastSegmentIndex : 0 ''
```

Segment Base

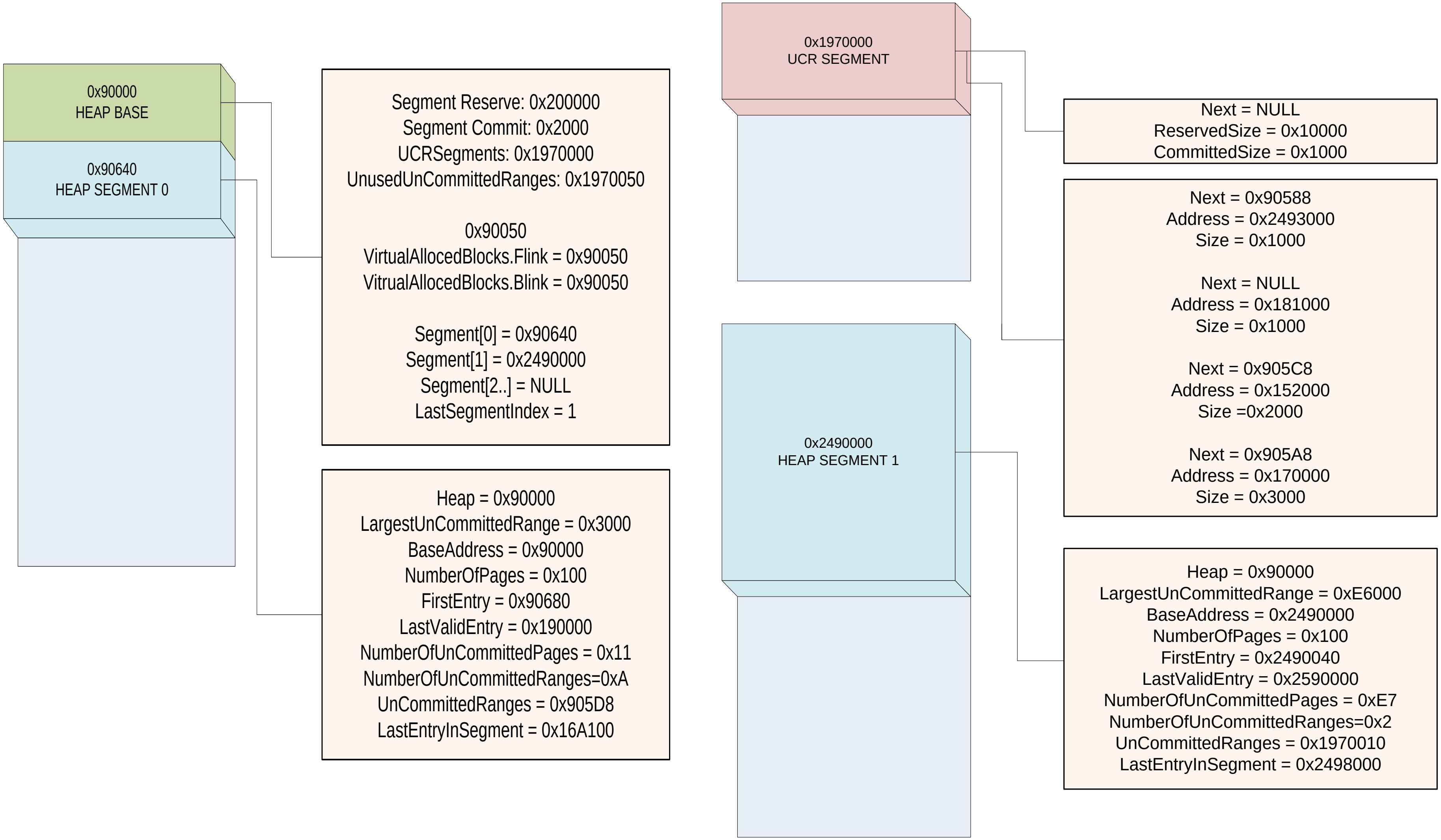
- **Segments**
  - A contiguous range of reserved virtual memory
  - Typically a fraction is committed up-front
  - The rest is committed later

- **Segment Base**
  - Each heap contains a **segment base** that is used to keep track of memory associated with heap segments
  - The **segment base** is an array of 64 _HEAP_SEGMENT structures
  - Each **_HEAP_SEGMENT** can be traversed to view information about each heap chunk contained by that segment
  - This is done by some debugging tools (Thanks Nico ☺) to provide information about heap chunks
  - The following slide contains a debugging dump of a **_HEAP_SEGMENT** structure under Windows XP SP3:

Heap Segment Dump

```
0:001> dt _HEAP_SEGMENT 150640
ntdll!_HEAP_SEGMENT
   +0x000 Entry            : _HEAP_ENTRY
   +0x008 Signature        : 0xffeeffee
   +0x00c Flags            : 0
   +0x010 Heap             : 0x00150000 _HEAP
   +0x014 LargestUnCommittedRange : 0xfc000
   +0x018 BaseAddress      : 0x00150000
   +0x01c NumberOfPages    : 0x100
   +0x020 FirstEntry       : 0x00150680 _HEAP_ENTRY
   +0x024 LastValidEntry   : 0x00250000 _HEAP_ENTRY
   +0x028 NumberOfUnCommittedPages : 0xfc
   +0x02c NumberOfUnCommittedRanges : 1
   +0x030 UnCommittedRanges : 0x00150588 _HEAP_UNCOMMMTTED_RANGE
   +0x034 AllocatorBackTraceIndex : 0
   +0x036 Reserved         : 0
   +0x038 LastEntryInSegment : 0x00153cc0 _HEAP_ENTRY
```

Memory Layout Example



Segment Reserve: 0x200000
Segment Commit: 0x2000
UCRSegments: 0x1970000
UnusedUnCommittedRanges: 0x1970050

0x90050
VirtualAllocedBlocks.Flink = 0x90050
VirtualAllocedBlocks.Blink = 0x90050

Segment[0] = 0x90640
Segment[1] = 0x2490000
Segment[2..] = NULL
LastSegmentIndex = 1

Heap = 0x90000
LargestUnCommittedRange = 0x3000
BaseAddress = 0x90000
NumberOfPages = 0x100
FirstEntry = 0x90680
LastValidEntry = 0x190000
NumberOfUnCommittedPages = 0x11
NumberOfUnCommittedRanges=0xA
UnCommittedRanges = 0x905D8
LastEntryInSegment = 0x16A100

Next = NULL
ReservedSize = 0x10000
CommittedSize = 0x1000

Next = 0x90588
Address = 0x2493000
Size = 0x1000

Next = NULL
Address = 0x181000
Size = 0x1000

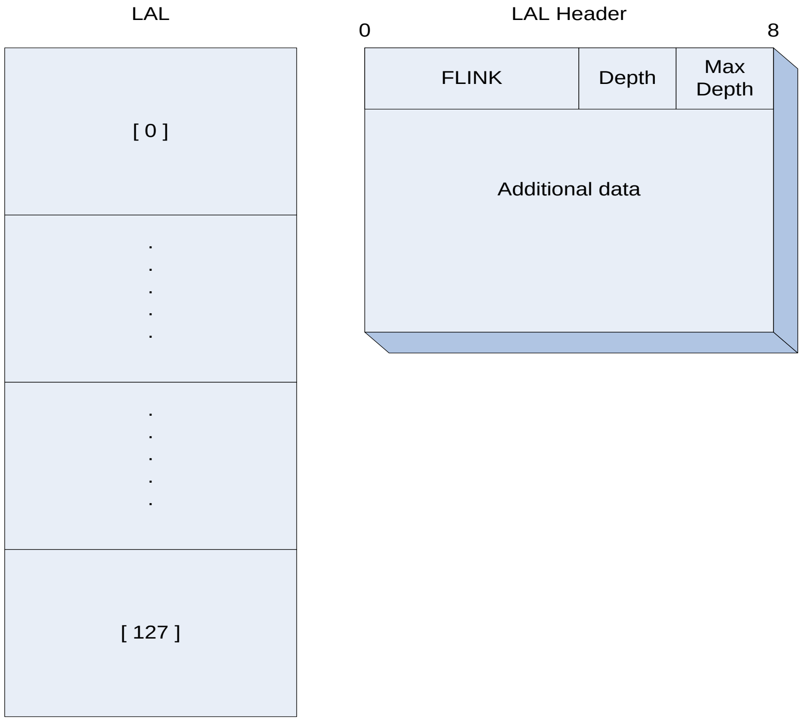
Next = 0x905C8
Address = 0x152000
Size =0x2000

Next = 0x905A8
Address = 0x170000
Size = 0x3000

Heap = 0x90000
LargestUnCommittedRange = 0xE6000
BaseAddress = 0x2490000
NumberOfPages = 0x100
FirstEntry = 0x2490040
LastValidEntry = 0x2590000
NumberOfUnCommittedPages = 0xE7
NumberOfUnCommittedRanges=0x2
UnCommittedRanges = 0x1970010
LastEntryInSegment = 0x2498000

Front-end Manager

- **Lookaside List (LAL)**
  - Array of 128 linked lists containing 48 byte data structures
  - Each bucket of the array represents free chunks of a certain size that are below 1024 bytes
  - The free chunks in each bucket are the size of the bucket, multiplied by 8
    - i.e. LAL[4] = 32 byte free chunks
    - Buckets 0 and 1 are not used because each heap chunk requires 8 bytes for the header
  - The first 8 bytes of the 48 byte LAL header is shown in detail



Front-end Manager

- **Allocation Pseudo-code**

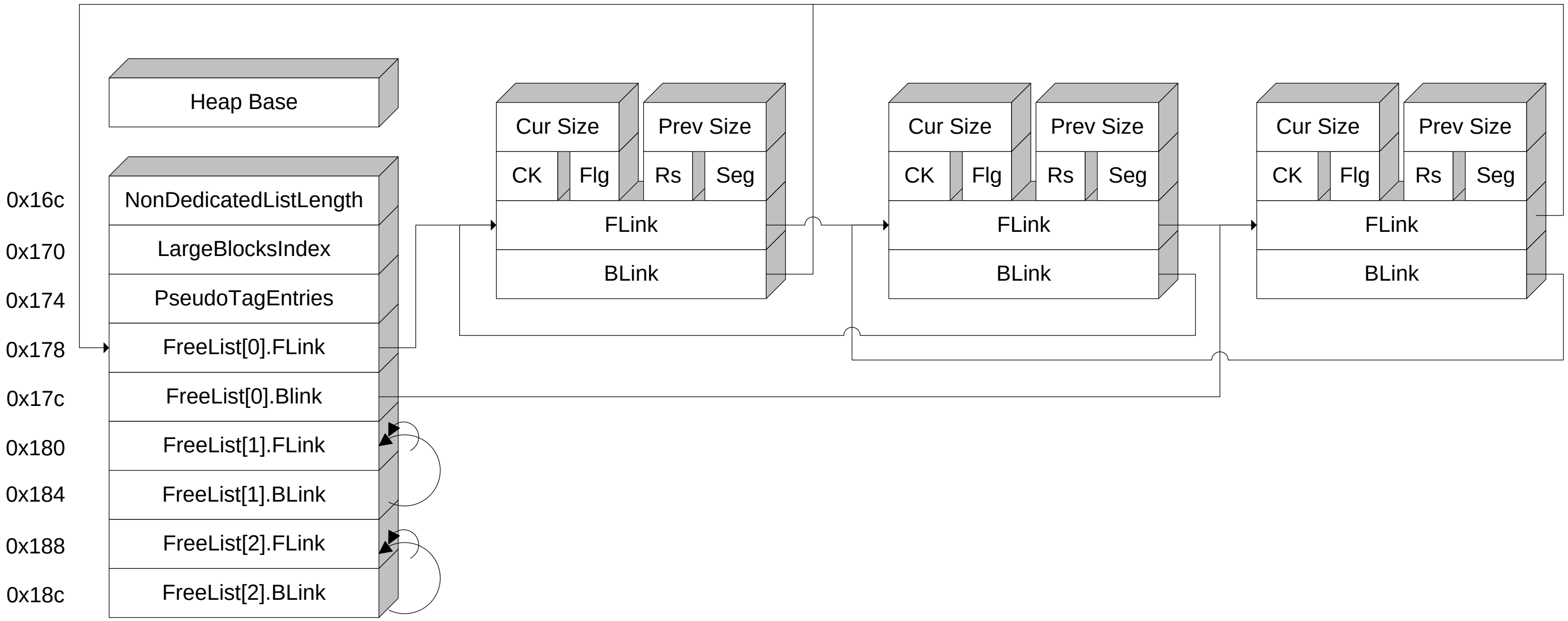
```
size = allocation_request_size;

//add 8 for the chunk header
blocksize = Round(size + 8) / 8;

//get the correct bucket via LAL header size
bucket = (HeapBase+0x688) + (blocksize * 0x30);

if(size < 0x80 && LFH == 0) {
      entry = RtlpAllocateFromHeapLookaside(bucket);
      if(entry)
            return entry;

}

//use FreeList instead
```

Back-end Manager

- **FreeList**
  - Contains **128** doubly-linked lists representing **free chunks** up to 1024 bytes
  - Each list contains a sentinel node located at the base of the heap, starting at **+0x178** from the base
  - An empty list is denoted by both pointers pointing back to the sentinel head node
  - **FreeList[0]** is a special list that contains all the free chunks that are >= 1024 bytes
  - **FreeList[0]** is sorted in order, from **smallest** to **largest**
  - **FreeList[1]** is **not** used because each chunk must contain an 8 byte header, leaving no space for user data

Back-end Manager

- **FreeList Diagram**
  - This diagram shows a populated **FreeList[0]** and an empty **FreeList[1] and FreeList[2]**



Back-end Manager

- **FreeListInUseBitmap**
  - Since not all requests to be serviced have a corresponding **FreeList** entry, the memory manager was given an optimization
  - The **FreeListInUseBitmap** is 128-bit (4 byte) value located at **+0x158** from the base of the heap
  - Each bit represents a bucket in the **FreeList** ranging from 0x00 to 0x7F
  - If the **bit** is **set**, that **FreeList** bucket contains free chunks of that size, otherwise the list is considered **empty**
  - The **FreeListInUseBitMap** is queried when there is not a **direct** match for amount of memory requested. It will then locate and use a chunk from the **next largest** list

Core Algorithms

Core Algorithms

- **Commitment / De-commitment**
  - The heap manager will initially **reserve** memory for use. This only means the address range will not be gobbled up by another thread.
  - **Committing** is the act of actually mapping and backing the **reserved** virtual memory
  - **De-committing** is the act of taking mapped memory and returning it to the reserved section
  - Processes can freely **commit** and **de-commit** memory in a reserved chunk without actually un-reserving it (Which is called **releasing** memory)
  - Read / Write operations on **un-committed** memory will result in an **access violation (AV)**
    - I've heard a lot of people refer to this as 'writing off the end of the page', which isn't entirely false, be it that memory is committed one page 0x1000 at a time. Why am I even discussing this…on with the presentation!
  - **Reserving, committing, de-committing,** and **releasing** of memory are all performed by the **VirtualAlloc()** and **VirtualFree()** functions**.**

Core Algorithms

- **FreeList Search**
  - If the request to be serviced is < 1016 bytes and the **front-end allocator** has failed to fulfill the request, a dedicated **FreeList** is used. We'll refer to this as **FreeList[n]**
    - If there are free entries on **FreeList[n]** then it is used, otherwise the **FreeListInUseBitMap** is queried to find a sufficiently sized chunk.
    - If the chunk used is more than 8 bytes larger than the requested size, it is split. The **requested** chunk is returned to the user, leaving the **remainder** to be linked back into an appropriate **FreeList**
  - If the request to be serviced is >= 1024 bytes then **FreeList[0]** is used. As discussed previously, **FreeList[0]** contains all the free chunks that are >= 1024 bytes in size.
  - Once a properly sized chunk has been found, the node is **unlinked** from the **FreeList**, updating its **FLINK/BLINK** pointers, along with that of each of its **neighbors**.
  - A more thorough discussion of details will be covered in the following pseudo-code…

Core Algorithms

- **FreeList Search Pseudo-code pt.1**
  - It will first check to see if the size is less than 1024, and attempt to use the **Lookaside List**. If that fails, it will continue its search in the **FreeList**

```
if (size<0x80)
{
      // we have an entry in the lookaside list
      if (chunk = RtlpAllocateFromHeapLookaside(size))
            return chunk;
}
```

Core Algorithms

- ## FreeList Search Pseudo-code pt.2
  - If the **Lookaside List** fails and the size is under 1024 the dedicated **FreeList[n],** which corresponds to the request side, is used. An empty **FreeList[n]** will result in the **FreeListInUseBitMap** being queried for a sufficiently sized chunk

```
if (size<0x80)
{
      // we have an entry in the free list
      if (FreeLists[size].flink != &FreeLists[size])
            return FreeLists[size].blink;


      // ok, use bitmap to find next largest entry
      if (offset=scan_FreeListsInUseBitmap(size))
      {
            return FreeLists[offset].blink;
      }

      // we didn't find an entry in the bitmap so fall through
      // to FreeLists[0]
}
```

Core Algorithms

- **FreeList Search Pseudo-code pt.3**
  - If search from the dedicated **FreeLists** fail or the request size >= 1024, **FreeList[0]** is used. It will first attempt to use the **heap cache** as an optimization

```
if (Heap->LargeBlocksIndex )      // Heap Cache active?
{
       foundentry = RtlpFindEntry(Heap, size);

       // Not found in Heap Cache
      if (&FreeLists[0] == foundentry )
              return NULL;

       // returned entry not large enough
       if (SIZE(foundentry) < size)
              return NULL;

       // we're allocing a >=4k block,
      // and the smallest block we find is >=16k.
       // flush one of the large blocks, and allocate a new
       // one for the request

       if (LargeBlocksIndex->Sequence &&
              size > Heap->DeCommitFreeBlockThreshold &&
              SIZE(foundentry) > (4*size))
       {
             RtlpFlushLargestCacheBlock(vHeap);
              return NULL;
       }

       // return entry found in Heap Cache
       return foundentry;
}
```

Core Algorithms

- **FreeList Search Pseudo-code pt.4**
  - If the **Heap Cache** is **not** active, the heap will iterate through **FreeList[0]** to find an appropriately size block

```
// Ok, search FreeList[0] - Heap Cache is not active
Biggest = (struct _HEAP *)Heap->FreeLists[0].Blink;

// empty FreeList[0]
if (Biggest == &FreeLists[0])
      return NULL;

// Our request is bigger than biggest block available
if (SIZE(Biggest)<size)
      return NULL;

walker = &FreeLists[0];

while ( 1 )
{
      walker = walker->Flink;

      if (walker == &FreeLists[0])
            return NULL;

      if ( SIZE(walker) >= size)
            return walker;
}
```

Core Algorithms

- **FreeList Unlinking**
  - When a free chunk of memory is selected to service a request it has to be removed from the **FreeList**.
  - Unlinking can be thought of as a 3 step process
    - 1. Remove the node from the **Heap Cache** (if it is active)
    - 2. A safe unlink is performed
      - Ensures that the node being unlinked doesn’t have tainted FLink/BLink
    - 3. If the node is on a dedicated FreeList (non-FreeList[0]), then the **FreeListInUseBitMap** is updated accordingly
  - Lets discuss some pseudo-code…

Core Algorithms

- **FreeList Unlinking Pseudo-code**

```
// remove block from Heap Cache (if activated)
RtlpUpdateIndexRemoveBlock(heap, block);

prevblock = block->blink;
nextblock = block->flink;

// safe unlink check
if ((prevblock->flink != nextblock->blink) ||
    (prevblock->flink != block))
{
    // non-fatal by default
    ReportHeapCorruption(…);
}
else
{
    // perform unlink
    prevblock->flink = nextblock;
    nextblock->blink = prevblock;
}

// if we unlinked from a dedicated free list and emptied it,
// clear the bitmap
if (reqsize<0x80 && nextblock==prevblock)
{
    size = SIZE(block);
    vBitMask = 1 << (size & 7);

    // note that this is an xor
    FreeListsInUseBitmap[size >> 3] ^= vBitMask;
}
```

Core Algorithms

- **FreeList Linking**
  - Linking is the process of taking a free chunk and placing it into the appropriate **FreeList**
  - Linking occurs when:
    - A block is **split**, placing the remainder back on the **FreeList**
    - A chunk is **freed**, adding it into the appropriate **FreeList**
  - Linking involves:
    - 1. Determining the correct **FreeList** for the chunk to be inserted
    - 2. Toggling the **FreeListInUseBitMap** if necessary
      - First free chunk on an empty list
    - 3. Find an appropriate block and acquire its **BLINK**
    - 4. Link in the new block, updating its pointers along with those of its neighbors
  - Let’s look at some pseudo-code…

Core Algorithms

- **FreeList Linking Pseudo-code**

```
int size = SIZE(newblock);

// we want to find a pointer to the block that will be after our block

if (size < (0x80))
{
      afterblock = FreeList[size].flink;

      //toggle bitmap if freelist is empty
      if (afterblock->flink == afterblock)
            set_freelist_bitmap(size);
}
else
{
    if (Heap->LargeBlocksIndex )    // Heap Cache active?
        afterblock = RtlpFindEntry(Heap, size);
    else
        afterblock = Freelist[0].flink;

    while(1)
    {
        if (afterblock==&FreeList[0])
            return; // we ran out of free blocks

        if (SIZE(afterblock) >= size)
            break;

        afterblock=afterblock->flink;
    }
}

// now find a pointer to the block that will be before us
beforeblock=afterblock->blink;

// we point to the before and after links
newblock->flink = afterblock;
newblock->blink = beforeblock;

// now they point to us
beforeblock->flink = newblock;
afterblock->blink = newblock;
```

Core Algorithms

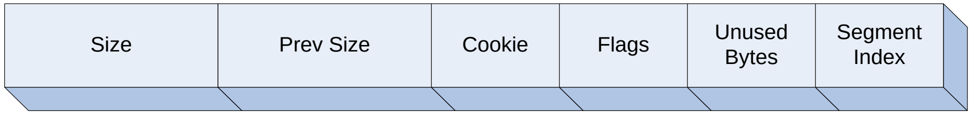
- **Coalescing**
  - When a heap chunk is inserted into a list, via freeing or block splitting, it will attempt to coalesce with its neighbors
  - This prevents a fragmented heap
    - Imagine having only free chunks of size 0x10, the memory manager would need to commit more memory for larger allocations
  - Coalescing used to be the perfect spot for an arbitrary DWORD overwrite, then the **HeapReportCorruption**() / safe-unlinking fail was introduced ☹

```
//calculates this by subtracting the current->prev_size from its
location
prev = current->blink

if(prev_chunk->flags != Flags.Busy &&
(current->size + prev->size < 0xFE00)) {

      //safe unlink check
      if((prev->flink != next->blink) || (prev->flink != current))
            RtlpHeapReportCorruption(...);
      else {
            //remove from the heap cache if necessary
            RtlpUpdateIndexRemoveBlock(...);

            //unlink the coalesced chunk
            prev->blink->flink = current;
            prev->flink->blink = prev->blink;

            //add the sizes
            current->size += prev->size;

            //update FreeListInUseBitMap if necessary
            UpdateBitMap(prev);
      }
}

//the same is done for current->next
```

Security Mechanisms

Security Mechanisms

- **Heap Cookie**
  - The heap cookie is a security mechanism introduced in Windows XP SP2
  - Checked on free
  - Below is a heap chunk header for WinXPSP2 and the algorithm used to check for a valid heap cookie
    - Note: the address of the memory chunk has a part in cookie creation
    - Note: the heap cookie is only 1-byte long, leaving it somewhat vulnerable to brute force



Size	Prev Size	Cookie	Flags	Unused Bytes	Segment Index

```
        if((&chunk  / 8) ^ chunk->cookie ^ heap->cookie ) {
              RtlpHeapReportCorruption(chunk)
        }
```

Security Mechanisms

- **Safe Unlinking**
  - As mentioned previously, **safe unlinking** assures that a heap chunk to be freed is not corrupted
  - A check is made to determine if a prev->flink and next->blink point to the same location.
  - Then it makes sure that prev->flink points to the chunk being freed
  - This made the old-school arbitrary DWORD overwrite (typically done during coalescing) the fail, instead of the win

```
if(chunk->blink->flink != chunk->flink->blink)
{
        RtlpHeapReportCorruption(chunk)
}
if(chunk->blink->flink != chunk)
{
        RtlpHeapReportCorruption(chunk)
}

//Unlink / Link the chunk
```

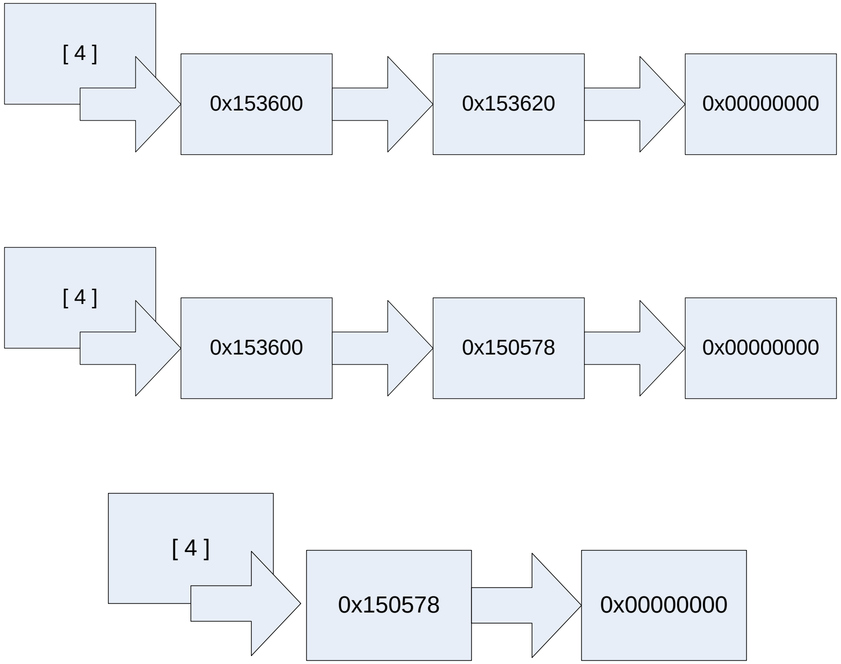
Security Mechanisms

- **Process Termination**
  - In the previous slides, we mentioned the mechanisms put in place to prevent ‘easy’ exploitation of heap overflows
  - Execution can be terminated if heap corruption is detected by the heap manager
  - This is done by setting the **HeapEnableTerminateOnCorruption** flag through the **HeapSetInformation()** API
    - This is only supported on Windows Vista and Windows Server 2008.
    - For 2003 and XP, if the image **gflag FLG_ENABLE_SYSTEM_CRIT_BREAKS** is set, the Heap Manager will call **DbgBreakPoint**() and raise an exception if the safe-unlink check fails. This is an uncommon setting, as its security properties aren’t clearly documented
  - In 2003, there are some corner-case corruptions that are detected that will raise an exception. Not safe-unlink, though.

Tactics

Best of Breed Tactic 1 – LAL Overwrite

- **Lookaside List Link Overwrite**
  - Overwrite **FLINK** in **Lookaside** List Header
  - Credited to Alexander Anisimov
  - http://www.maxpatrol.com/defeating-xpsp2-heap-protection.pdf

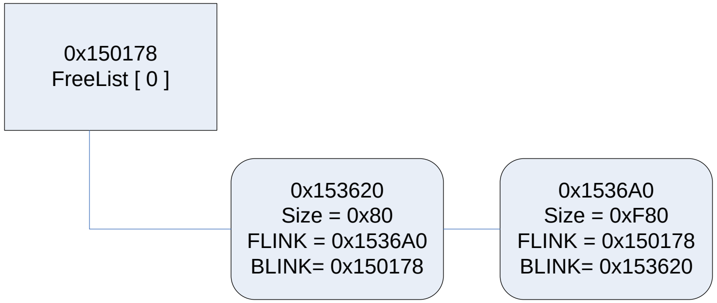


Best of Breed Tactic 2 – FL[0] Attacks

Freelist[0] multiple techniques

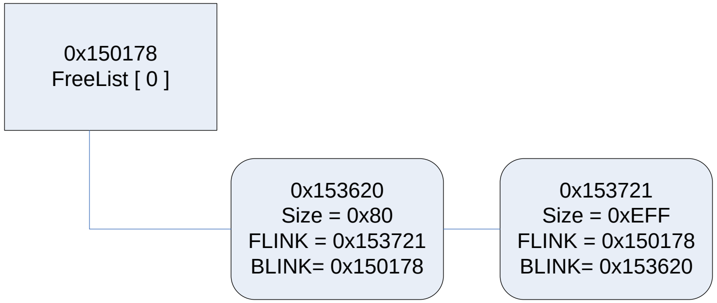
Linking is unsafe
Safe-unlinking doesn't Terminate

- Using the maintenance algorithms associated with **FreeList[0]** to achieve overwrites
- Brett Moore rules
- Heaps about Heaps / Exploitign FreeList[0] on XPSP2 -> http://www.insomniasec.com/releases/whitepapers-presentations


0x150178
FreeList [0]

0x153620
Size = 0x80
FLINK = 0x1536A0
BLINK= 0x150178

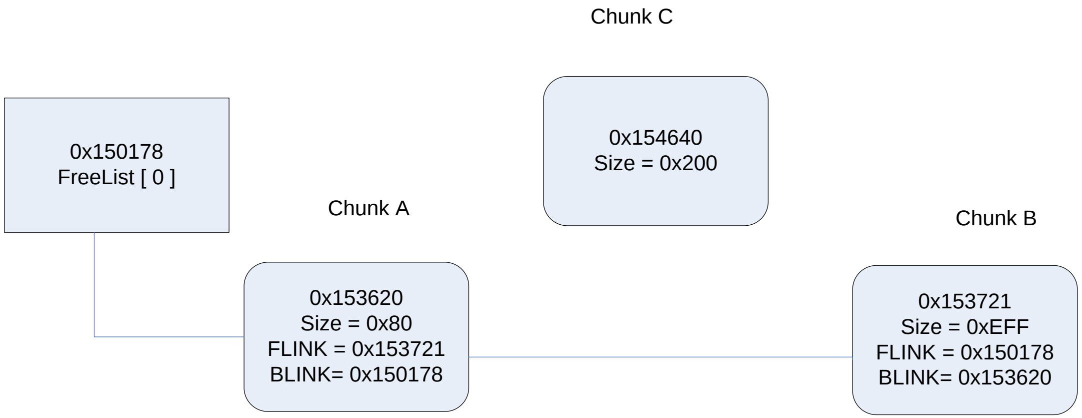
0x1536A0
Size = 0xF80
FLINK = 0x150178
BLINK= 0x153620


0x150178
FreeList [0]

0x153620
Size = 0x80
FLINK = 0x153721
BLINK= 0x150178

0x153721
Size = 0xEFF
FLINK = 0x150178
BLINK= 0x153620

FL[0] Attacks 2 - Chunk Linking



- ChunkC->FLINK = ChunkB
- ChunkC->BLINK = ChunkB->BLINK

- ChunkB->BLINK->FLINK = ChunkC
- ChunkB->BLINK = ChunkC

Best of Breed Tactic 3 – Bitmap (FL[n]) Attacks

- **Bitmap Flipping Attack**
  - Toggling the **FreeListInUseBitMap** to make empty list appear to be populated
  - Credited to **Nico Waisman**
    - (Have you caught on that we’ve jacked most of Nico’s stuff?)
  - **Heaps about Heaps**
  - http://www.insomniasec.com/releases/whitepapers-presentations

New Tactics

Contrary to recent “reports,” IBM Soylent Green is almost definitely probably not people.

New Tactics

- **Bitmap Attack Variation**
  - Nico Waisman (name drop #3 for those keeping track at home) suggested targeting the **FreeListInUseBitMap** when incrementing an arbitrary DWORD
    - http://lists.immunitysec.com/pipermail/dailydave/2007-May/004364.html
  - We've found that scenarios where a 1-4 byte overwrites into a chunk on a **FreeList / Lookaside List** == win.
  - If the user can control certain allocations, he/she can use this to toggle bits in the **FreeListInUseBitMap** making an empty **FreeList** appear to be populated
  - Let's look at some code…

New Tactics

- **Bitmap XOR Attack**

```
/* coming into here, we've found a bit in the bitmap */
/* and listhead will be set to the corresponding FreeList[n]
head*/

_LIST_ENTRY *listhead = SearchBitmap(vHeap, aSize);

/* pop Blink off list */
_LIST_ENTRY *target = listhead->Blink;

/* get pointer to heap entry (((u_char*)target) - 8) */
HEAP_FREE_ENTRY *vent = FL2ENT(target);

/* do safe unlink of vent from free list */
next = vent->Flink;
prev = vent->Blink;

if (prev->Flink != next->Blink || prev->Flink != listhead)
{
      RtlpHeapReportCorruption(vent);
}
Else
{
      prev->Flink=next;
      next->Blink=prev;
}

/* Adjust the bitmap */

// make sure we clear out bitmask if this is last entry
if ( next == prev )
{
      vSize = vent->Size;
      vHeap->FreeListsInUseBitmap[vSize >> 3] ^= 1 << (vSize &
7);
}
```

New Tactics

- **Bitmap Updating Problems**

  1. The code checks if ‘next == prev’ to determine if the recently acquired chunk is the last chunk on the **FreeList**. If it appears to be the last chunk the **FreeListInUseBitmap** will be updated
     - This is an easy scenario to forge if 16 bytes are can be overwritten, resulting in improper **FreeListInUseBitmap** updating
  2. The size used for updating the bitmap is taken directly from the heap chunk
     - Just like all the other heap metadata, the size can be overwritten resulting in updating the bitmap for a **FreeList** incorrectly
  3. **FreeListInUseBitMap** is updated by way of XOR.
     - Clearing of the bitmap should be done by setting the bit to zero, instead of XOR’ing the bit. This can lead to updating of arbitrary **FreeLists**
  4. Heap chunk size is not checked to be below 0x80
     - This means that we can toggle bits in semi-arbitrary locations past the **FreeListInUseBitmap**. This could be quite useful because the bitmap is located in the **Heap Base**.

New Tactics

- **RtlpAllocateFromHeapLookaside / ExInterlockedPopEntrySList**

```
int __stdcall RtlpAllocateFromHeapLookaside(struct LAL *lal)
{
  int result;
  try {
    result = (int)ExInterlockedPopEntrySList(&lal->ListHead);
  }
  catch {
    result = 0;
  }

  return result;
}
```

```
__fastcall ExInterlockedPopEntrySList(void *lal_head)
{
  do {
    int lock = *(lal_head + 4) - 1;
    if(lock == 0)
      return;

    flink = *(lal_head);
  }
  while (!AtomicSwap(&lal_head, flink))
}
```

New Tactics

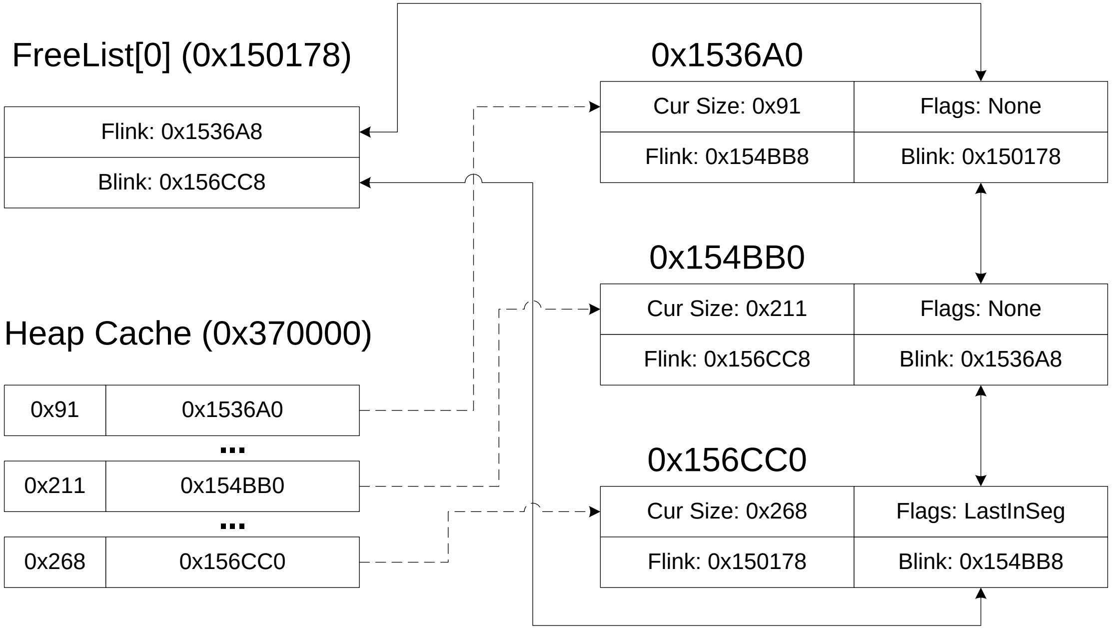
- **Lookaside List Exception Handler**
  - When allocating from the **Lookaside List** the code in ntdll wraps *ExInterlockedPopEntrySList* (used to get item off a singly linked list) in a ‘catch-all’ **exception handler**
  - The **exception handler** will return 0, leaving the **memory manager** to proceed to use the **FreeLists**
  - If an attacker can overwrite the **FLINK** of the first entry on a **Lookaside** List (or somehow get their overwritten entry to the front), then they can use it to attempt to brute force **stack addresses**, **PEB**, **TIB**, etc
    - They would also need to control the write used on a returned address
  - An attacker could also use this as a method to bypass the **Lookaside List** entirely, if using the **FreeLists** is more desirable.

Back-end Manager

- **Heap Cache (Large Block Index)**
  - **FreeList[0]** contains all the free chunks >= 1024 bytes in length
  - The **Heap Cache** is an optimization enhancement to minimize traversals of **FreeList[0]** by creating an external index for the blocks
    - NOTE: It should be noted that the Heap Manager does not move any of the blocks into the cache, the blocks are still kept in **FreeList[0]**, but the cache contains pointers into the nodes in **FreeList[0]**
  - The cache consist of an array of **896 buckets** (by default, it can be configured differently) each representing chunk sizes between 1024 and 8192
  - Each bucket contains a single pointer to the first block in **FreeList[0]** with the size represented by the bucket
  - If there is no corresponding entry in **FreeList[0]** the **Heap Cache** bucket contains a **NULL** pointer
  - The last bucket is special, as it points to the first free chunk in **FreeList[0]** >= **8192**.
    - Therefore representing all free chunks >= 8192 bytes
  - Since most buckets will be empty, there is a corresponding **bitmap** used to speed up allocations.
    - This used in the same way as the **FreeListInUseBitMap**

New Tactics

- **Heap Cache Example**



FreeList[0] (0x150178)

Flink: 0x1536A8

Blink: 0x156CC8

Heap Cache (0x370000)

0x91 0x1536A0

...

0x211 0x154BB0

...

0x268 0x156CC0

0x1536A0

Cur Size: 0x91 Flags: None

Flink: 0x154BB8 Blink: 0x150178

0x154BB0

Cur Size: 0x211 Flags: None

Flink: 0x156CC8 Blink: 0x1536A8

0x156CC0

Cur Size: 0x268 Flags: LastInSeg

Flink: 0x150178 Blink: 0x154BB8

New Tactics

- **Heap Cache Invocation**
  - Invoked during runtime to avoid frequent commitment / de-commitment of memory
  - Heap cache is activated when:
    - 32 entries simultaneously in **FreeList[0]**
      - for (i=0;i<32;i++)
      - {
      -     b1=HeapAlloc(pHeap, 0, 2048+i*8);
      -     b2=HeapAlloc(pHeap, 0, 2048+i*8);
      -     HeapFree(pHeap,0,b1);
      - }
    - 256 blocks must have been de-committed
      - for (i=0;i<256;i++)
      - {
      -     b1=HeapAlloc(pHeap, 0, 65536);
      -     HeapFree(pHeap,0,b1);
      - }

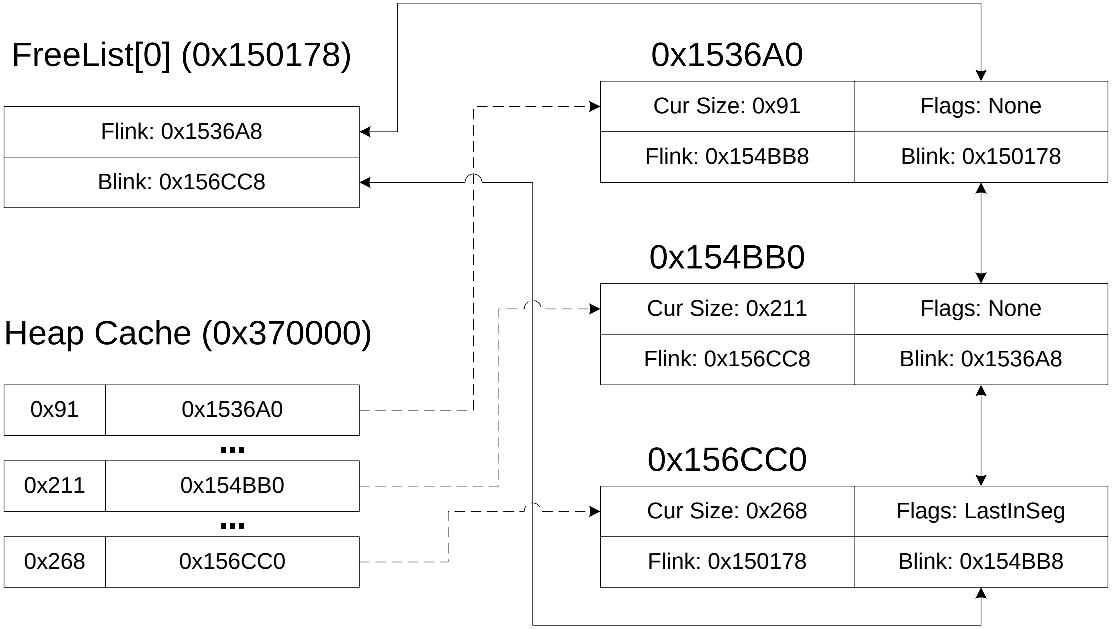
Heap Cache Attacks

- **Premise of Attacks: Chunk Size is the primary key**

- **Heap Cache De-synchronization**
  - Simplest form of attack works by corrupting the size of a heap chunk in the heap cache
  - 1-byte overflow
  - On alloc, chunk will be removed from **FreeList[0]**
  - **Heap Cache** will contain a stale pointer to this chunk

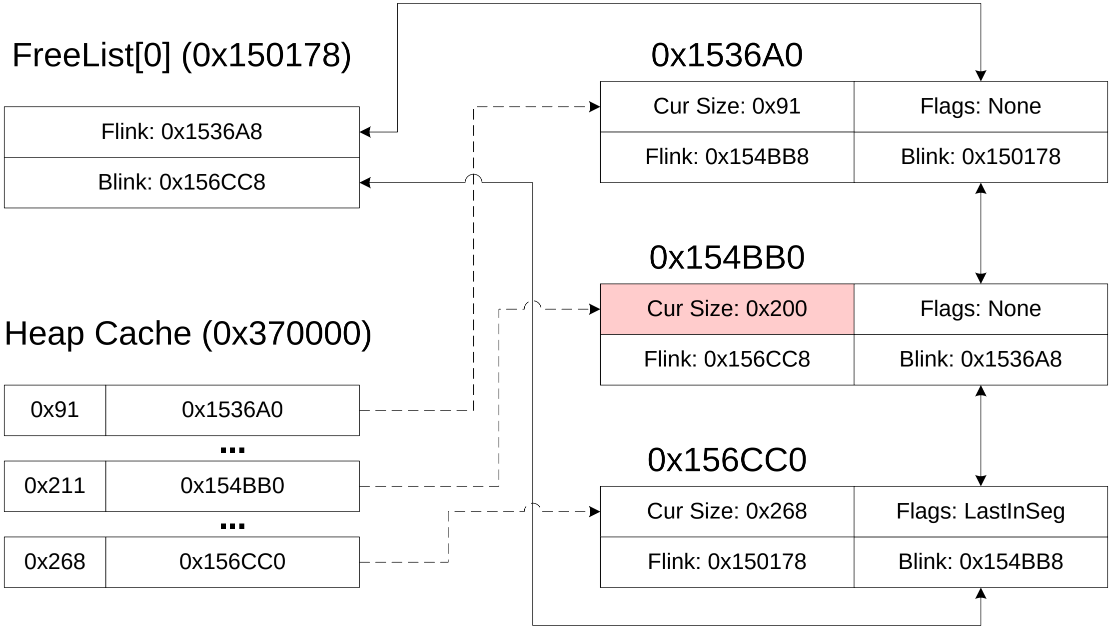
Desynch I

- **State of Nature**



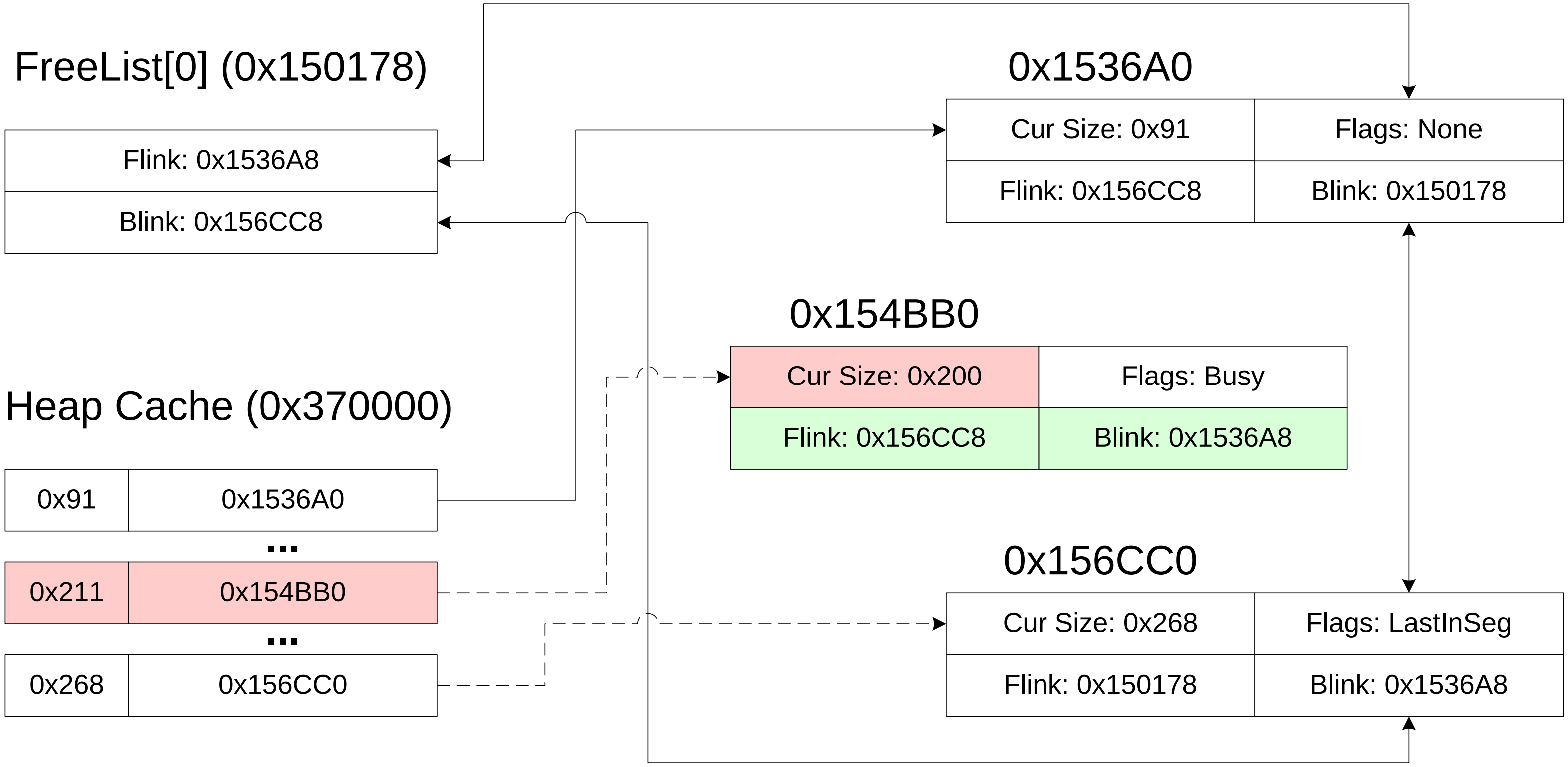
Desynch II

- **1-byte Overflow**



Desynch III

- **Buffer is alloced.**
- **Flink and Blink written readable pointers**



Desynch IV

- The result of stale pointer:
  - HeapAlloc(heap, 0, 0xFF8) returns **0x154BB8**
  - HeapAlloc(heap, 0, 0xFF8) returns **0x154BB8**
  - HeapAlloc(heap, 0, 0xFF8) returns **0x154BB8**
  - HeapAlloc(heap, 0, 0xFF8) returns **0x154BB8**

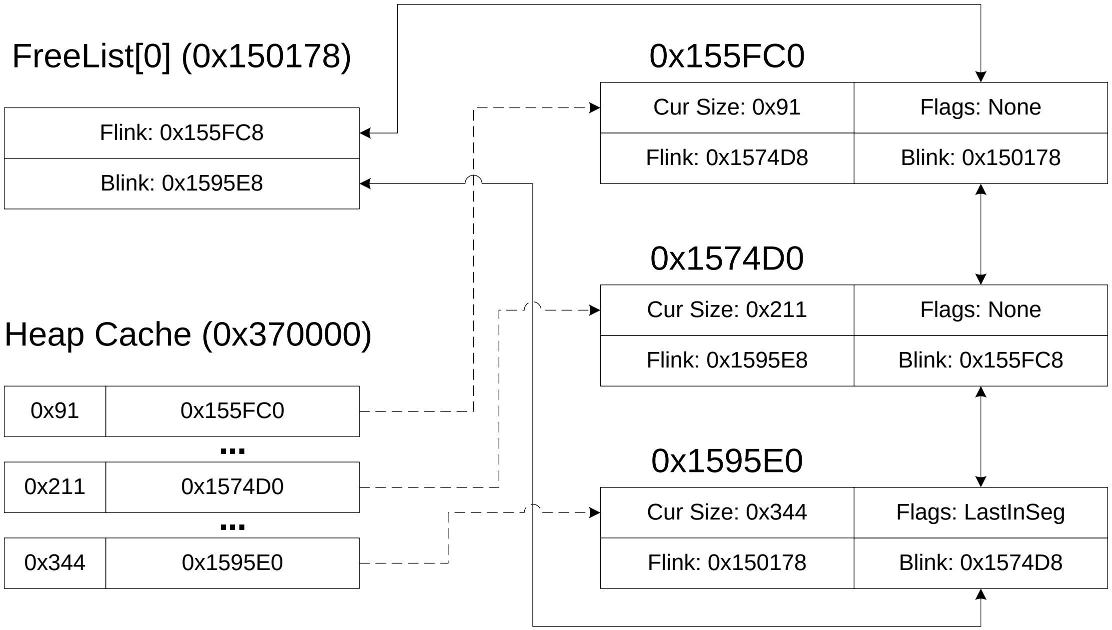
New Tactics

- **Heap Cache Insertion Attack**
  - We do desynch
  - But provide evil Flink and Blink via alloc
  - Need to control the first 8 bytes written

  - We borrow heap base pointers from Dr. Moore
    - See *Heaps about Heaps*

Heap Cache Insertion I

- **State of Nature**



FreeList[0] (0x150178)

Flink: 0x155FC8
Blink: 0x1595E8

Heap Cache (0x370000)

0x91	0x155FC0
…	
0x211	0x1574D0
…	
0x344	0x1595E0

0x155FC0

Cur Size: 0x91	Flags: None
Flink: 0x1574D8	Blink: 0x150178

0x1574D0

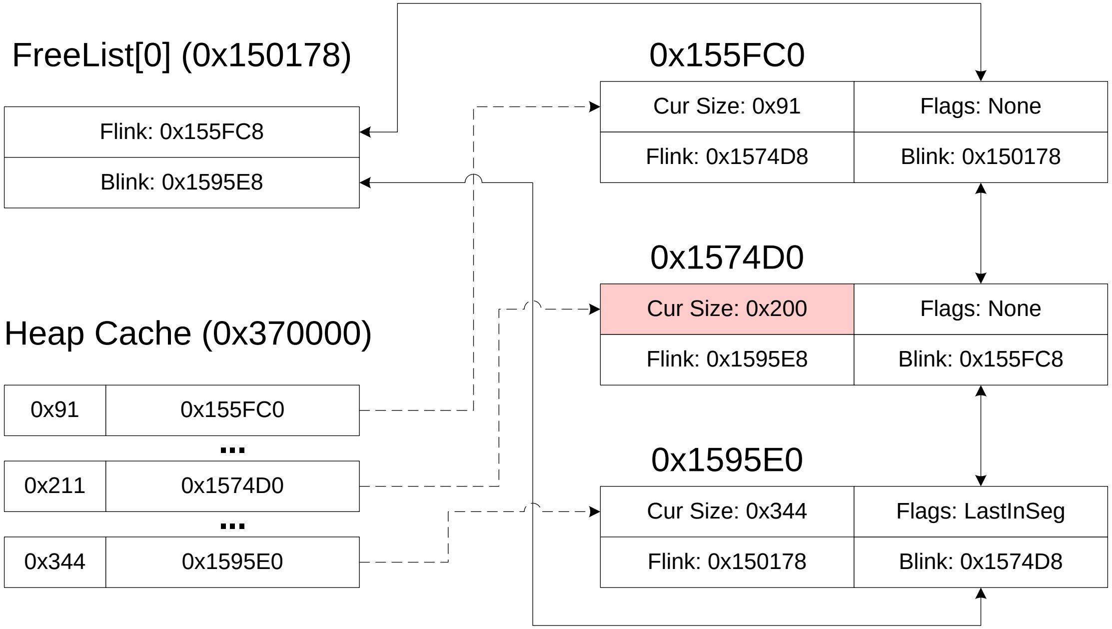
Cur Size: 0x211	Flags: None
Flink: 0x1595E8	Blink: 0x155FC8

0x1595E0

Cur Size: 0x344	Flags: LastInSeg
Flink: 0x150178	Blink: 0x1574D8

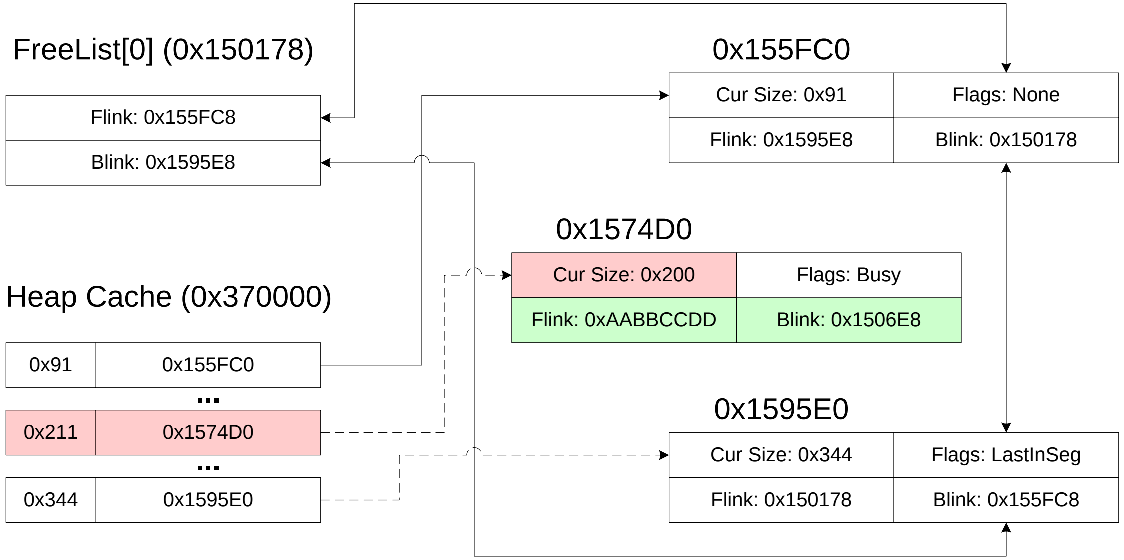
Heap Cache Insertion II

- **We overwrite 1 byte with 0**



Heap Cache Insertion III

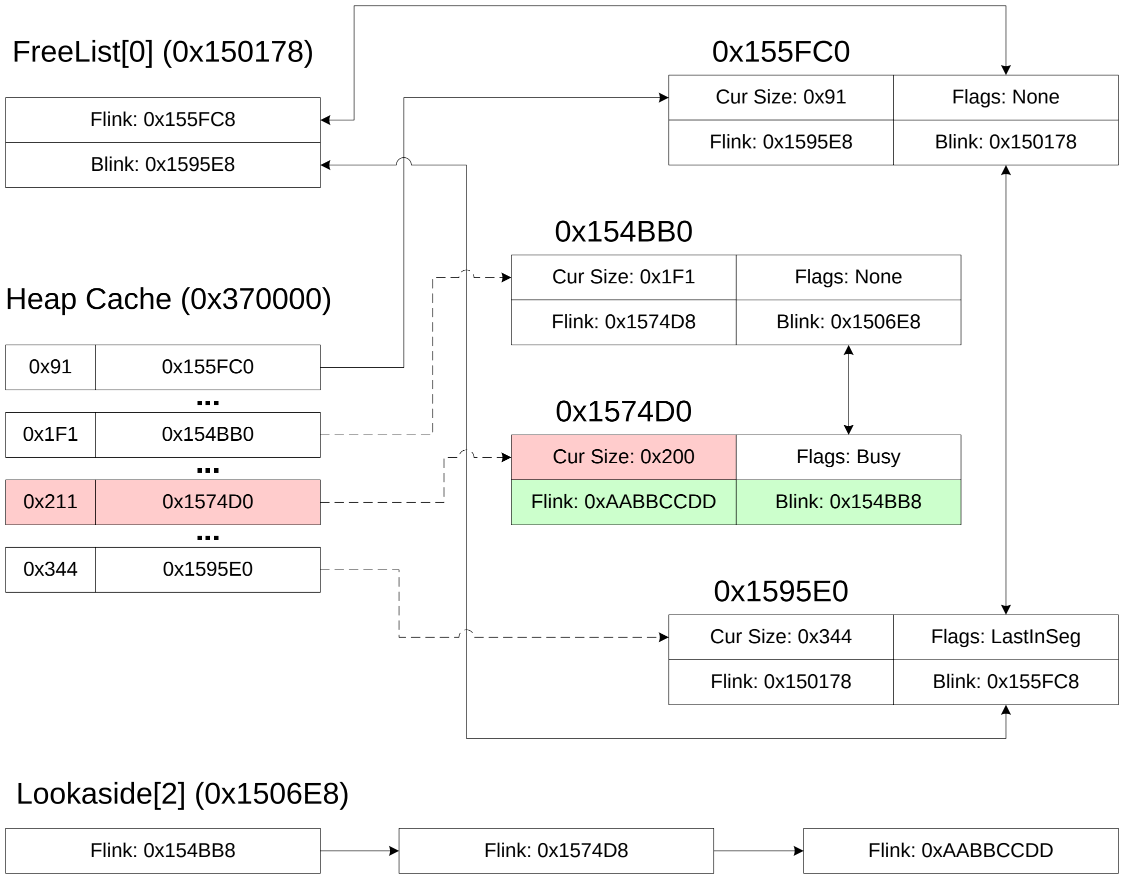
- **Our evil block is allocated and we control contents**



We provide Flink and Blink to exploit unsafe-linking

Heap Cache Insertion IV

- **Insert a block behind our evil block for the win**

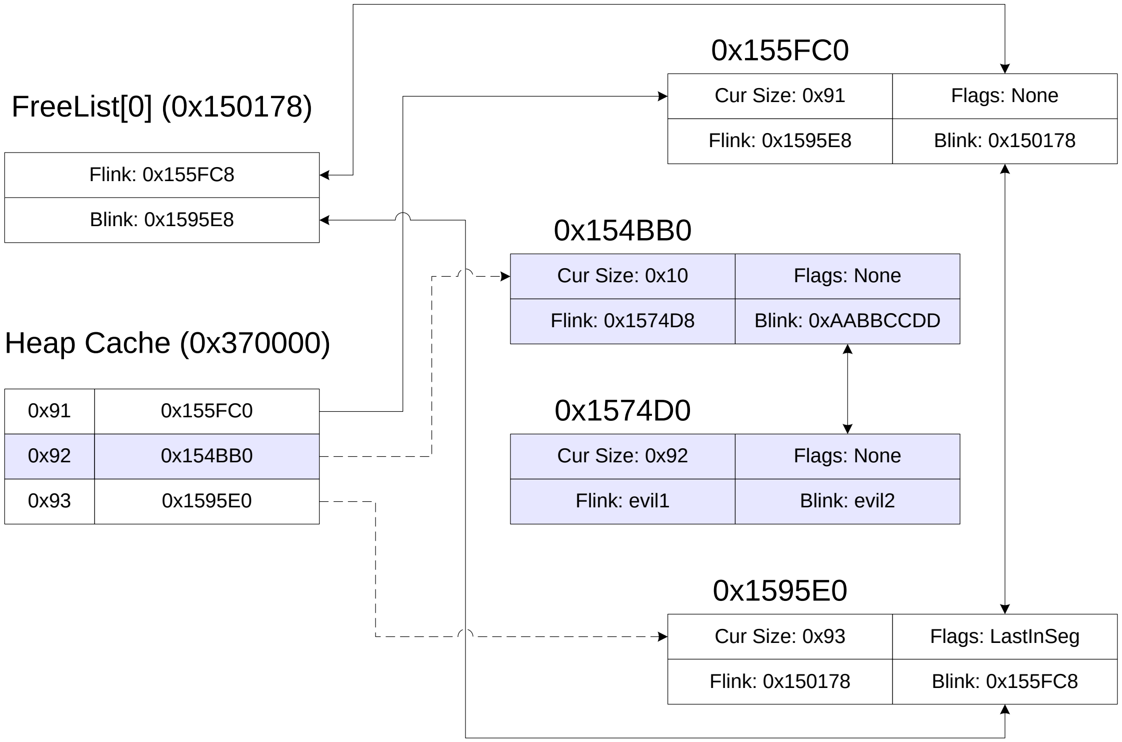


Size Targeting / Shadow FreeLists

- One of the biggest obstacles to overcome when attempting to win against the memory manager is **block splitting**
- Block splitting occurs when an allocation request is serviced by a heap block that is larger than the requested size
- It will break-up the chunk into the **result block**, which is returned to the user, and the **remainder block**, which is **coalesced** if necessary and returned to the appropriate **FreeList**
- A consistent **heap cache** will result in most allocations / linking searches being serviced by legitimate entries in **FreeList[0]**
- Creation of a **Shadow FreeList** can create a **trapdoor** for a specific allocation size, which in turn will provide more resiliency for innocuous allocations / linking searches
  - i.e. Allocations for certain amounts will be fulfilled by legitimate entries in **FreeList[0]** while allocations for other sizes can be serviced by malicious entries placed into the **heap cache**

Shadow FreeList

- **Shadow FreeList**



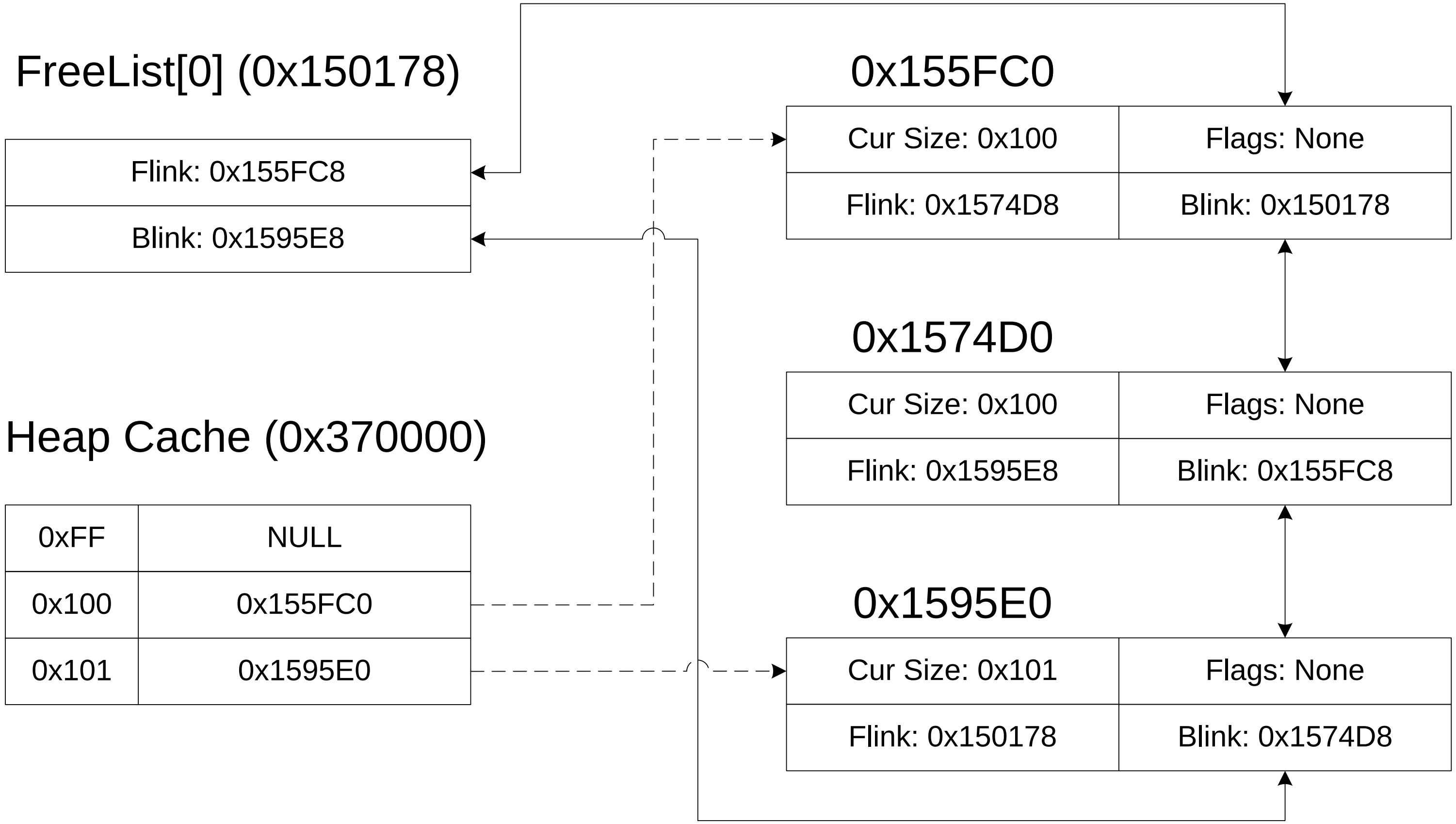
Malicious Entry

- **Malicious Heap Cache Entry**
  - Each Heap Cache bucket can refer to multiple entries
    - i.e. 10 chunks of the size 0x100 in FL[0]
  - The **FLINK** of an entry in the heap cache is followed to determine if the bucket is empty
  - If an attacker can provide a malicious **FLINK** value through memory corruption, and this value is a valid pointer to an appropriate size word, then they can get a malicious address placed into the heap cache
    - For the ‘catch-all’ bucket (chunk size >= 8192), you only need to overwrite the **FLINK** with a valid pointer that points to a size that is **>= 8192**

  - **In short, you can win if block is in Heap Cache**
  - **Or if it’s not**

Malicious Entry

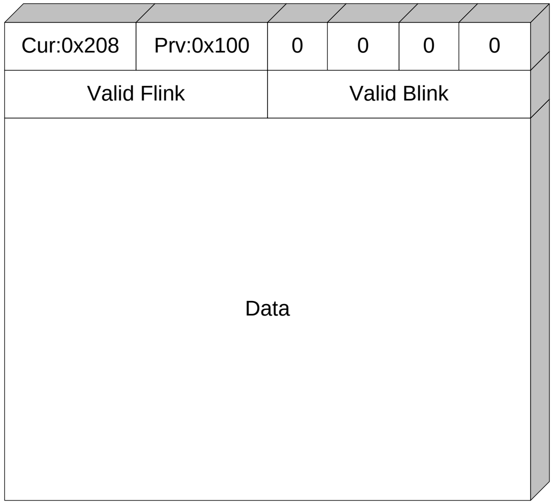
- **State of Nature**



Malicious Entry 2

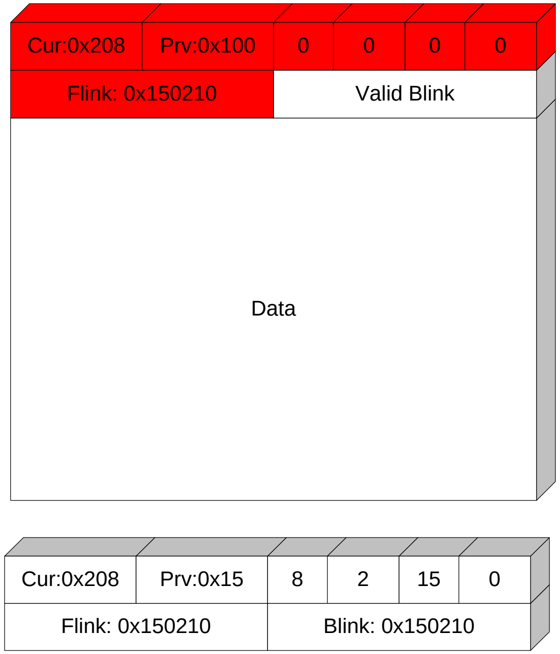
- **Dedicated Block**

Pre - Overwrite



Cur:0x208	Prv:0x100	0	0	0	0
Valid Flink			Valid Blink		
Data					

Post - Overwrite



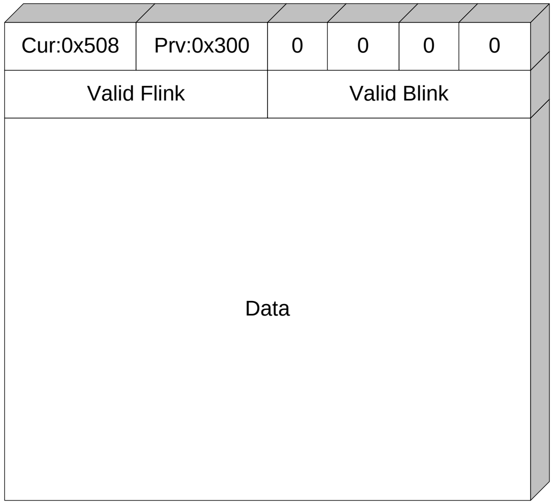
Cur:0x208	Prv:0x100	0	0	0	0
Flink: 0x150210			Valid Blink		
Data					

Cur:0x208	Prv:0x15	8	2	15	0
Flink: 0x150210			Blink: 0x150210		

Malicious Entry 3

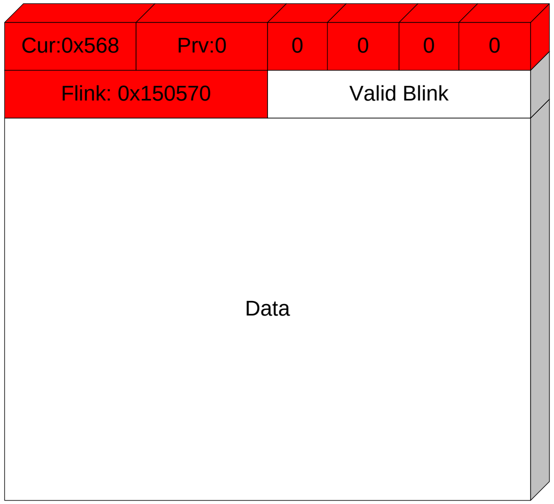
- **Catch-All Bucket**

Pre - Overwrite

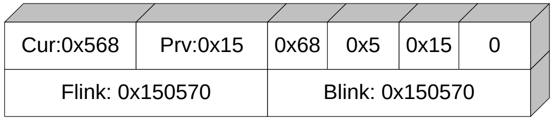


Cur:0x508	Prv:0x300	0	0	0	0
Valid Flink			Valid Blink		
Data					

Post - Overwrite



Cur:0x568	Prv:0	0	0	0	0
Flink: 0x150570			Valid Blink		
Data					



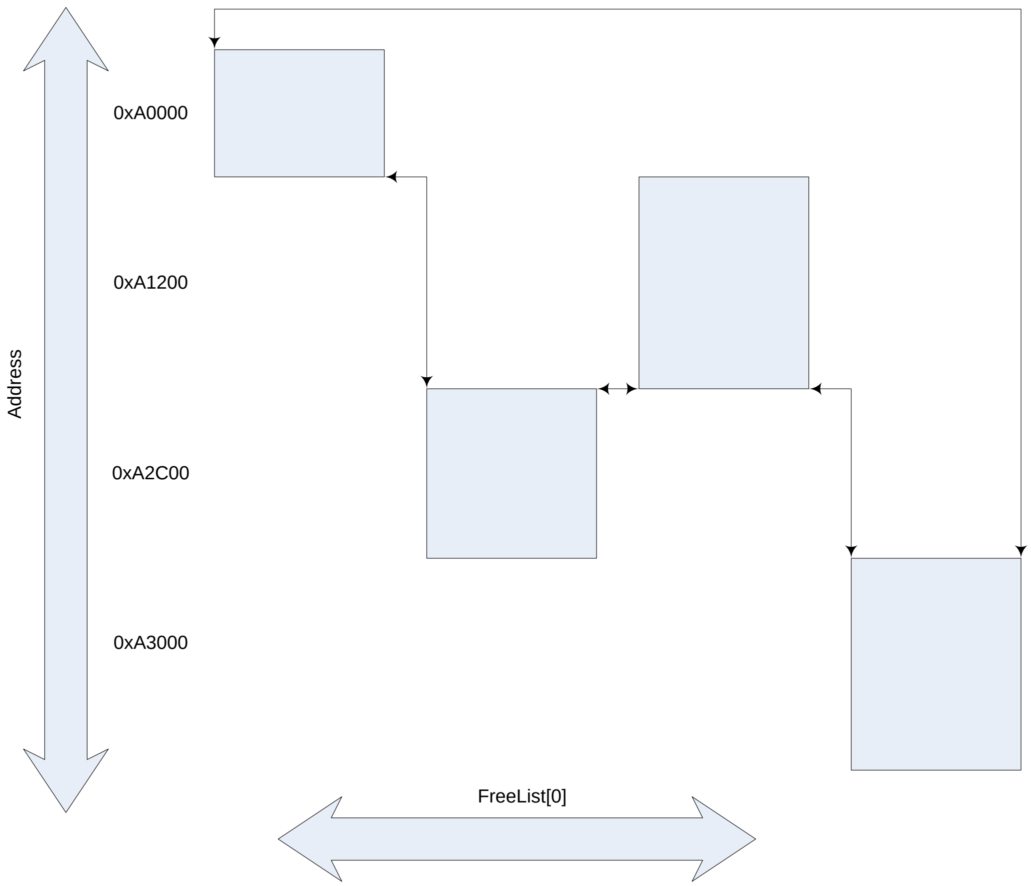
Cur:0x568	Prv:0x15	0x68	0x5	0x15	0
Flink: 0x150570			Blink: 0x150570		

Strategy

This strategy better feature an awesome screen-saver.

Strategy

- Be vague
- Use lots of metaphors
- Be multi-dimensional
- Be the dream?



Meta-data or Application-data?

- **Meta-data**
  - – you attack internal Heap data structures
    - Pro - you often know where meta-data is
      - Base of process heap
      - Heap chunk header
    - Con - heap meta-data is hardened

- **Application data**
  - – you target the data *in* heap
    - Pro – app data is uniformly soft
    - Con – can add uncertainty

Best of Breed Strategy

- **Published**
  - Dr. Waisman – Memory Leaks
  - Heap Spraying / Heap Feng Shui
    - Originally used by SkyLined for IE IFRAME vulnerability
    - Using JavaScript strings to store shellcode creating a semi-reliable return address
  - **Feng Shui** the heap to a more deterministic state
    - Alexander Sotirov: BlackHat Europe 2007
    - http://www.phreedom.org/research/heap-feng-shui/
- **Our Process**
  - 1. State of Nature – Get your bearings
  - 2. Action Correlation – Correlate user actions
  - 3. Heap Normalization – Normalize to predictable state
  - 4. Fixing in Contiguous Memory – Create necessary holes
  - 5. Fixing in Logical Lists – Create necessary
  - 6. Corruption – Invoke the attack
  - 7. Exploitation – Move to code execution

General Process

- **1. State of Nature**
  - Get your bearings in a process post-corruption
  - Questions:
    - Is the Heap Cache likely to be invoked already?
    - Is there a LAL on the Heap you are corrupting?
    - Is there an LFH?
    - How populated is the LAL?
    - How about the free lists?

General Process

- **2. Action Correlation**
  - Correlate user actions to allocation behavior
  - Find the following:
    - Permanent or long-living memory leaks
    - Allocations where you control the contents of the first bytes
    - Short-life memory leaks for timing allocations and frees
    - The ability to free a buffer of an arbitrary size at an arbitrary time
    - Allocations of arbitrary size for heap normalization and hole creation,
    - Information leaks
    - Targets!
      - Function pointers and other primitives in application data.

General Process

- **3. Heap Normalization**
  - Normalize the heap to predictable state
  - LAL
  - FreeList[n]
  - Heap Cache

- **Use Patterns!**
  - Invoke Heap Cache
  - Fill Holes

General Process

- **4. Fixing in Contiguous Memory**
  - Create necessary holes in memory
- **5. Fixing in Logical Lists**
  - Create necessary logical relationships
- **6. Corruption**
  - Invoke the attack
- **7. Exploitation**
  - Move from immediate corruption to code execution

Nico’s Timeline for XPSP2

- **1 day: Triggering the bug**
- **1-2 days: Understanding the heap layout**
- **2-5 days: Finding Soft and Hard Memleaks**
- **10-30 days: Overwriting a Lookaside Chunk**
- **1-2 days: Getting burned out, crying like a baby, trying to quit, doing group therapy**
- **2-5 days: Finding a Function pointer**
- **1-2 days: Shellcode**

Tools

Need a good, flexible programmatic debugging environment.

- Windbg/dbgeng seems like it would be a great way to implement these.
- Pydbg
- Gera’s Heap Visualizer / Tracer
- Byakugen – (visualizer unpublished)
- Flashky’s Heap tool. (We haven’t seen it.)
- Myriad tools for normal programmers (UDMH, etc)

  These run out of utility for us pretty quicly.

Immunity Debugger!@#

These are quite useful (and there’s more)

- !funsniff
- !hippie
- !heap –d (target discovery)

We’ll kick back immunity debugger changes

Or Dave can just grab them off our hard drives

Demo

(getting popcorn)

Demo

- **And now to prove that we aren’t making all of this up…**

- *** Note: proof may be in the form of a prepared video of dubious veracity.**

Conclusion

Conclusion

- Although the mitigations have been introduced to the Windows Heap Manager for some time there still exists ways to obtain reliable exploitation
- **FreeListInUseBitMap** can be leveraged to exploit 1-4 byte overflows
- The LAL exception handler can be used for brute forcing addresses to overwrite
- **Heap Cache**
  - Can be used to normalize the heap to turn seemingly impossible exploitation conditions into more workable scenarios
  - **Shadow FreeLists** can provide **trapdoors** to prevent innocuous allocations / linking searches from crashing the application before exploitation can occur
  - Just like Brett Moore's **FreeList[0]** techniques the **heap cache** can be used for address overwriting, making code execution possible in the post-safe-unlinking world.

Conclusion

- **Microsoft has taken steps in the right direction with Vista Heap Manager**
  - ASLR, heap meta-data encryption, optional process termination upon corruption & heightened focus on security
- We really hate the heap and it is the bane of our existence

FIN

(For those of you playing along from home, this is where the audience erupts into spontaneous applause and weeping)