#### Windows 8 Heap Internals







#### Windows 8 Heap Internals

### INTRODUCTION





#### Who

- Chris Valasek (@nudehaberdasher)
  - Sr. Research Scientist
  - Coverity
- Tarjei Mandt (@kernelpool)
  - Vulnerability Researcher
  - Azimuth Security





#### What

- Windows 8 Release Preview
- Heap manager specifics
- Exploitation techniques for Windows 8 heap
- Prerequisite reading
  - "Understanding the LFH"
    - http://illmatics.com/Understanding the LFH.pdf
    - http://illmatics.com/Understanding the LFH Slides.pdf
  - "Modern Kernel Pool Exploitation"
    - http://www.mista.nu/research/kernelpool infiltrate2011.pdf
  - Kostya, Hawkes, Halvar, McDonald, Moore, etc





### Why

- Learn how the Heap Manager and Kernel Pool Allocator work (in detail)
  - PLEASE read the paper if you want full details, this presentation just touches the surface
- Heap exploits that worked on Windows 7 will most likely NOT work on Windows 8
- Let's find out why





#### Windows 8 Heap Internals

### **User Land Back-End**





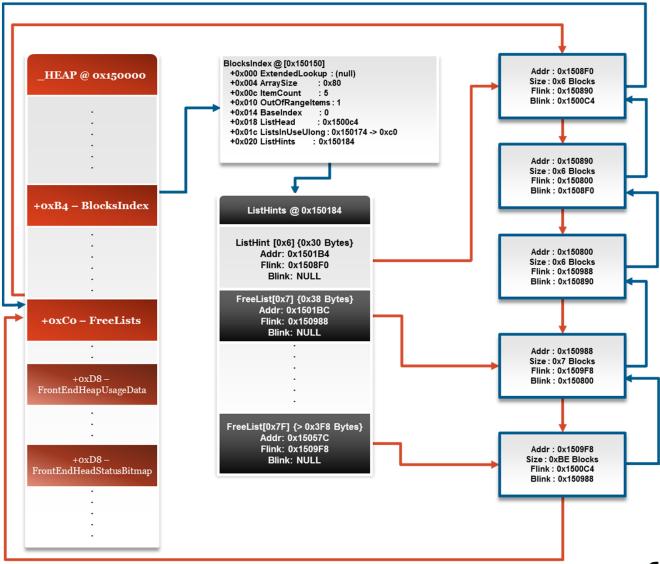
### Windows 8 Back-end

- Slightly modified version of the Windows 7 back-end [RtlpAllocateHeap()]
- Mitigations
  - Freeing of \_HEAP structures is prohibited (R.I.P Ben Hawkes tech)
  - 2. Virtually allocated chunks now have randomized locality/size





# Windows 8 Back-end (cont.)







# Back-end Mitigation I

- Prevents the freeing and subsequent allocation of a \_HEAP structure in RtlpFreeHeap().
  - https://www.lateralsecurity.com/downloads/hawkes ruxcon-nov-2008.pdf
  - Although the direct overwriting can still occur, it is unlikely
- Same holds true for RtlpReAllocateHeap()





# Back-end Mitigation I (cont.)

```
RtlpFreeHeap(_HEAP *heap, DWORD flags, void *header, void *mem)
{
    .
    .
    .
    if(heap == header)
    {
        RtlpLogHeapFailure(9, heap, header, 0, 0, 0);
        return 0;
    }
    .
    .
    .
}
```





# Back-end Mitigation II

- Chunk that exceeds the VirtualMemoryThreshold will be serviced by NtAllocateVirtualMemory()
- Previously, the allocations occurred with a potential for semipredictable locations and sizes
- Changes have been made to add a random offset to the base address when allocating large chunks in RtlpAllocateHeap()
- Hope to encapsulate virtual chunk in inaccessible memory (MEM\_RESERVE)
- Note: If safe-linking fails the application will only terminate if HeapTerminateOnCorruption has been set via HeapSetInformation(), otherwise the chunk is NOT linked in but still RETURNED





# Back-end Mitigation II

```
//VirtualMemoryThreshold set to 0x7F000 in CreateHeap()
int request size = Round(request_size)
int block size = request size / 8;
if(block size > heap->VirtualMemoryThreshold)
        int rand_offset = (RtlpHeapGenerateRandomValue32() & 0xF) << 12;</pre>
        request size += 24;
        int region size = request size + 0x1000 + rand offset;
        void *virtual base, *virtual chunk;
        int protect = PAGE READWRITE;
        if(heap->flags & 0x40000)
                 protect = PAGE_EXECUTE_READWRITE;
        //Attempt to reserve region_size bytes of memory
        if(NtAllocateVirtualMemory(-1, &virtual_base, 0, &region_size, MEM_RESERVE, protect) < 0)</pre>
                 goto cleanup and return;
        virtual chunk = virtual base + rand offset;
        if(NtAllocateVirtualMemory(-1, &virtual chunk, 0, &request size, MEM COMMIT, protect) < 0)</pre>
                 goto cleanup_and_return;
        //XXX Set headers and safe link-in
        return virtual_chunk;
```





#### Windows 8 Heap Internals

### User Land Front End





#### Windows 8 Front-End

- Major changes to allocation and free algorithms and moderate changes to integral data structures
- RtlpLowFragHeapAllocFromContext() will not be a "matched function" by BinDiff between Windows 7 and Windows 8
- Mostly the same data structures but offsets and members have changed a bit





#### Mitigations

- 1. Front-End Activation
  - Dedicated counters/index instead of ListHint->Blink
  - FrontEndHeapUsageData[] (See paper)
- 2. Front-End Allocation
  - FreeEntryOffset removed
  - Non-deterministic allocations
- 3. Fast Fail
  - RtlpLowFragHeapAllocFromZone() implements fast fail
    - Also additional checking compared to Windows 7
- 4. Guard Pages
- 5. Arbitrary Free Mitigation
- Exception Handling Removal





### Windows 7 Front-End



\_INTERLOCK\_SEQ.Hint (i.e. FreeEntryOffset) is gathered from the free chunk  $\mbox{w/o}$  validation

_HEAI	P_USERDATA_H	HEADER
_HEAP_ENTRY	FreeEntryOffset	<user data=""></user>





### Windows 7 Front-End Allocation 0

				τ	JserF	Block	S				
_HEAP_ENTRY	NextOffset 0x06	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x0C	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x12	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x18	<user data=""></user>
_HEAP_ENTRY	NextOffset 0x1E	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x24	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x2A	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x30	<user data=""></user>
_HEAP_ENTRY	NextOffset 0x36	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x3C	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x42	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x48	<user data=""></user>
_HEAP_ENTRY	NextOffset 0x4E	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x54	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x5A	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x60	<user data=""></user>

Depth = oxoF FreeEntryOffset = oxo







### Windows 7 Front-End Allocation I

	UserBlocks														
_HEAP_ENTRY	∕ <user data=""></user>		_HEAP_ENTRY	NextOffset <user da<="" th=""><th>_HEAP_ENTRY</th><th colspan="2">NextOffset <user data=""> 0x12</user></th><th>_HEAP_ENTRY</th><th>NextOffset 0x18</th><th><user data=""></user></th></user>		_HEAP_ENTRY	NextOffset <user data=""> 0x12</user>		_HEAP_ENTRY	NextOffset 0x18	<user data=""></user>				
_HEAP_ENTRY	NextOffset 0x1E	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x24	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x2A	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x30	<user data=""></user>				
_HEAP_ENTRY	NextOffset 0x36	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x3C	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x42	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x48	<user data=""></user>				
_HEAP_ENTRY	NextOffset 0x4E	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x54	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x5A	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x60	<user data=""></user>				

Depth = oxoE FreeEntryOffset = oxo6







### Windows 7 Front-End Allocation II

				Ţ	JserF	Block	S				
_HEAP_ENTRY	_HEAP_ENTRY <b><user data=""></user></b>		_HEAP_ENTRY	∕ <user data=""></user>		_HEAP_ENTRY NextOffs 0x12		<user data=""></user>	_HEAP_ENTRY	NextOffset 0x18	<user data=""></user>
_HEAP_ENTRY	NextOffset 0x1E	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x24	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x2A	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x30	<user data=""></user>
_HEAP_ENTRY	NextOffset 0x36	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x3C	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x42	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x48	<user data=""></user>
_HEAP_ENTRY	NextOffset 0x4E	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x54	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x5A	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x60	<user data=""></user>

Depth = oxoD FreeEntryOffset = oxoC







### Windows 7 Front-End Allocation III

				Į	JserF	Block	S				
_HEAP_ENTRY	HEAP_ENTRY < <b>User Data</b> >		_HEAP_ENTRY	Y <user data=""></user>		_HEAP_ENTRY	<user data=""></user>		_HEAP_ENTRY	NextOffset 0x18	<user data=""></user>
_HEAP_ENTRY	NextOffset 0x1E	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x24	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x2A	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x30	<user data=""></user>
_HEAP_ENTRY	NextOffset 0x36	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x3C	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x42	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x48	<user data=""></user>
_HEAP_ENTRY	NextOffset 0x4E	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x54	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x5A	<user data=""></user>	_HEAP_ENTRY	NextOffset 0x60	<user data=""></user>

Depth = oxoC FreeEntryOffset = ox12

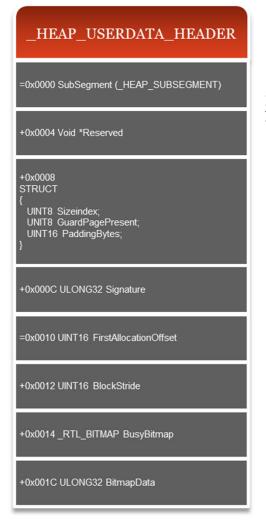




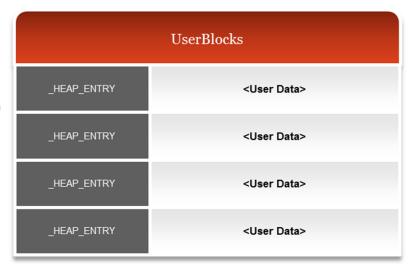


#### Windows 8 Front-End

No need to use the FreeEntryOffset as the Bitmap does all the work



EntryOffset w/in the UserBlocks I kept in \_HEAP\_ENTRY.PreviousSize









#### Windows 8 Randomization

 RtlpLowFragHeapRandomData initialized from RtlpCreateLowFragHeap and SlotIndex is updated on \_HEAP\_SUBSEGMENT creation [RtlpSubSegmentInitialize()]

```
RtlpInitializeLfhRandomDataArray()
{
    int RandIndex = 0;
    do
    {
        //ensure that all bytes are unsigned
        int newrand1 = RtlpHeapGenerateRandomValue32() & 0x7F7F7F7F;
        int newrand2 = RtlpHeapGenerateRandomValue32() & 0x7F7F7F7F;
        int newrand2 = RtlpHeapGenerateRandomValue32() & 0x7F7F7F7F;

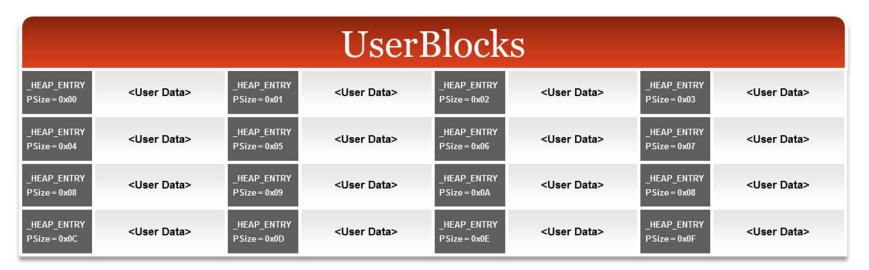
        RtlpLowFragHeapRandomData[RandIndex] = newrand1;
        RtlpLowFragHeapRandomData[RandIndex+1] = newrand2;

        RandIndex+=2;
    }
    while(RandIndex < 64)
}</pre>
```





### Windows 8 Front-End Allocation 0



Depth = oxoF





oxF OXO

	FREE															
- 1																





### Windows 8 Front-End Allocation I

			User	Blocks	5		
_HEAP_ENTRY PSize = 0x00	<user data=""></user>	_HEAP_ENTRY PSize = 0x01	<user data=""></user>	_HEAP_ENTRY PSize = 0x02	<user data=""></user>	_HEAP_ENTRY PSize=0x03	<user data=""></user>
_HEAP_ENTRY PSize = 0x04	<user data=""></user>	_HEAP_ENTRY PSize = 0x05	<user data=""></user>	_HEAP_ENTRY PSize = 0x06	<user data=""></user>	_HEAP_ENTRY PSize = 0x07	<user data=""></user>
_HEAP_ENTRY PSize = 0x08	<user data=""></user>	_HEAP_ENTRY PSize = 0x09	<user data=""></user>	_HEAP_ENTRY PSize = 0x0A	<user data=""></user>	_HEAP_ENTRY PSize = 0x08	<user data=""></user>
_HEAP_ENTRY PSize = 0x0C	<user data=""></user>	_HEAP_ENTRY PSize = 0x0D	<user data=""></user>	_HEAP_ENTRY PSize = 0x0E	<user data=""></user>	_HEAP_ENTRY PSize = 0x0F	<user data=""></user>

Depth = oxoE





Start = RandRand(o, Bitmap.SizeofBitmap); Index = CicularSearch(Bitmap, Start) UpdateBitmap(Bitmap, Index)

Index = ox5

oxF OXO

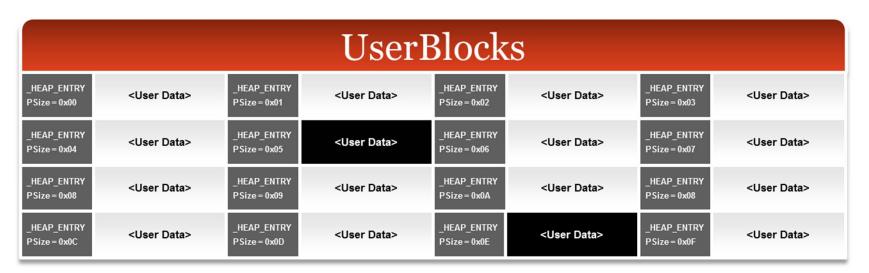
Return UserBlocks[Index]

FREI	FREE	FREE	FREE	FREE	BUSY	FREE										
------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	--





### Windows 8 Front-End Allocation II



Depth = oxoD





Start = RandRand(o, Bitmap.SizeofBitmap); Index = CicularSearch(Bitmap, Start) UpdateBitmap(Bitmap, Index) Return UserBlocks[Index]

Index = oxE

OXO









### Windows 8 Front-End Allocation III

			User	Blocks	S		
_HEAP_ENTRY PSize = 0x00	<user data=""></user>	_HEAP_ENTRY PSize = 0x01	<user data=""></user>	_HEAP_ENTRY PSize = 0x02	<user data=""></user>	_HEAP_ENTRY PSize = 0x03	<user data=""></user>
_HEAP_ENTRY PSize = 0x04	<user data=""></user>	_HEAP_ENTRY PSize = 0x05	<user data=""></user>	_HEAP_ENTRY PSize = 0x06	<user data=""></user>	_HEAP_ENTRY PSize = 0x07	<user data=""></user>
_HEAP_ENTRY PSize=0x08	<user data=""></user>	_HEAP_ENTRY PSize = 0x09	<user data=""></user>	_HEAP_ENTRY PSize = 0x0A	<user data=""></user>	_HEAP_ENTRY PSize = 0x08	<user data=""></user>
_HEAP_ENTRY PSize=0x0C	<user data=""></user>	_HEAP_ENTRY PSize = 0x0D	<user data=""></user>	_HEAP_ENTRY PSize = 0x0E	<user data=""></user>	_HEAP_ENTRY PSize = 0x0F	<user data=""></user>

Depth = oxoD





Start = RandRand(o, Bitmap.SizeofBitmap);

Index = CicularSearch(Bitmap, Start)

UpdateBitmap(Bitmap, Index)

Return UserBlocks[Index]

OXO

oxF

FRI	EE	FREE	FREE	FREE	FREE	BUSY	BUSY	FREE	BUSY	FREE						
-----	----	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------





#### Win 7 vs Win 8 Allocation

#### Windows 7

- Will sequentially allocate chunks from the UserBlock
- No validation of FreeEntryOffset, hence it can be overwritten and used as an exploitation primitive

#### Windows 8

- Randomized array used to search a bitmap
- Bitmap will select the chunk, update itself and use a different random location each time
- Heap determinism goes down significantly
- FreeEntryOffset no longer kept in user data, therefore FreeEntryOffset Overwrite technique has died ☺





- Fast Fail
  - INT 0x29 Interupt
  - Designed to ensure 'fast failing'
    - http://www.alex-ionescu.com/?p=69
  - Search "CD 29" (x86) and find instances all over ntdll.dll
  - Only one assertion in the LFH, otherwise use the RtlpLogHeapFailure() function and rely upon HeapTerminateOnCorruption flag





Bad News: Windows 8 checks LFH->SubSegmentZones

```
_HEAP_SUBSEGMENT *RtlpLowFragHeapAllocateFromZone(_LFH_HEAP *LFH, int AffinityIndex)
{
    .
    .
    .
    .
    .
    .
    LIST_ENTRY *subseg_zones = &LFH->SubSegmentZones;
    if(LFH->SubSegmentZones->Flink->Blink != subseg_zones ||
        LFH->SubSegmentZones->Blink->Flink != subseg_zones)
    __asm{int 29};
}
```

- Good News: Windows 7 has less strict checks
  - Potential for write-4 primitive ©





- Guard Pages were added between \_HEAP\_USERDATA\_HEADER objects to foil overwrites and heap spraying
- Therefore, an overflow will need to exist in the same UserBlock, potentially guarding other UserBlock containrs.
- After a certain amount of chunks exist for a certain size a guard page will be added for subsequent UserBlock creations
- If page\_shift == 0x12 || total\_blocks >= 0x400
  - Add a guard page to the allocation





```
RtlpLowFragHeapAllocFromContext()
      //determine if we should use a guard page
      set guard = false;
      //The total amount of chunks available for a _HEAP_SUBSEGMENT
      int total_block = HeapLocalSegInfo->Counters.TotalBlocks;
      if(total blocks > 0x400)
             total blocks = 0x400;
      //there are other operations here, left out for brevity
      int page shift = 7;
      int req size = total blocks * RtlpBucketBlockSizes[HeapBucket->SizeIndex] + 8;
      req_size = req_size + Round32(total_blocks) + 0x24;
      do
             page shift++;
      while(req_size >> page_shift);
      if(page shift == 0x12 || total blocks >= 0x400)
             set guard = true;
      //will allocate memory for the UserBlocks and add a guard page if necessary
      RtlpAllocateUserBlock(LFH, page shift, BucketByteSize, set guard);
```





RtlpAllocateUserBlock calls RtlpAllocateUserBlockFromHeap

```
RtlpAllocateUserBlockFromHeap(_HEAP *heap, int size, bool set_guard)
       HEAP USERDATA HEADER *user block = RtlAllocateHeap(heap, 0x800001, size - 8);
      if(set guard)
             int page_size = 0x1000;
             //get the page aligned address then caluculate the size
             //plus one page (0x1000)
             int page_end_addr = (user_block + (size - 8) + 0xFFF) & 0xFFFFF000;
             int new_size = page_end_addr - user_block + page_size;
             //reallocate with an additional page of memory appended
             user_block = RtlReAllocateHeap(heap, 0x800001, user_block, new_size);
             //make the last page of this memory PAGE NOACCESS
             ZwProtectVirtualMemory(-1, &new size, &page size, PAGE NOACCESS, &output);
             user block->GuardPagePresent = true;
       }
      return user block;
```





Low Address

Overflow Direction UserBlocks 1 for \_HEAP\_BUCKET[0x6]

Guard Page [PAGE\_NOACCESS]

UserBlocks 1 for \_HEAP\_BUCKET[ox8]

Guard Page [PAGE\_NOACCESS]

UserBlocks 2 for HEAP\_BUCKET[0x6]

**Higher Address** 

Contiguous Memory





- Ben Hawkes devised a technique to turn an overwrite of a LFH chunk into a semi-arbitrary free
  - https://www.lateralsecurity.com/downloads/hawkes ruxcon-nov-2008.pdf
  - Overwrite 'Flags' and 'Index' to point at a valid chunk within the UserBlock
  - Therefore you can taint a overflowed header, point to a legitimate, inuse chunk and free it
  - Win!
- There are checks to ensure that this will no longer work









```
if(Mem - 1 == 0x5)
{
    //this chunk was from the LFH
    if(header->UnusedBytes & 0x80)
    {
        //ensures that the header values haven't been altered
        if(!RtlpValidateLFHBlock(Heap, header))
        {
            RtlpLogHeapFailure(3, Heap, header, Mem, 0, 0);
            return 0;
        }
    }
}
```





### Windows 8 Front-End Mitigation VI

- Windows 7 wrapped
   RtlpLowFragHeapAllocFromContext() in a try/catch that would handle any exception
- I've speculated that this could be used to 'brute force' address overwrites if multiple memory corruptions were a possibility.
- This is REMOVED in Windows 8 <sup>(2)</sup>





# Summary

Primitive	Windows Vista	Windows 7	Windows 8 (RP)
Heap Handle Protection	×	×	$\overline{\checkmark}$
Virtual Memory Non-Determinism	×	×	
FrontEndStatusBitmap	×	×	$\overline{\checkmark}$
LFH Non-Determinism	×	×	$\overline{\checkmark}$
Fast Fail	×	×	$\overline{\checkmark}$
Guard Pages	×	×	$\overline{\checkmark}$
Arbitrary Free Protection	×	×	$\overline{\checkmark}$
Exception Handler Removal	×	×	





#### Windows 8 Heap Internals

### **User Land Exploitation Tech**





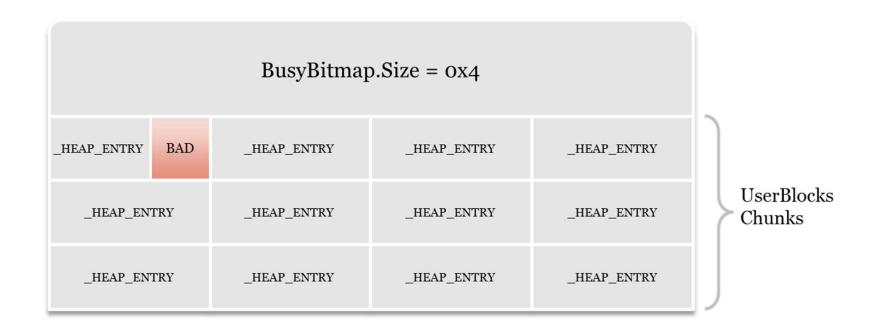
### Bitmap Flipping 2.0

- A LFH chunk's index within the UserBlock is still kept in an unencoded fashion
  - \_HEAP\_ENTRY.PreviousSize
  - Used to update the UserBlock->Bitmap
    - bittestandreset(UserBlocks->BusyBitmap->Buffer, header->PreviousSize);
  - Zero out certain bits relative to the address of the BusyBitmap
  - PROBLEMS
    - The UserBlock is taken from the \_HEAP\_SUBSEGMENT
    - SubSegment derived from chunk header
    - SubSegment = \*(DWORD)header ^ (header / 8) ^ heap ^ RtlpLFHKey;
    - UserBlocks = SubSegment->UserBlocks;
    - Corruption the chunk header (via sequential overflow) will wreck the SubSegment





### Bitmap Flipping 2.0



By making the PreviousSize of a chunk header to free larger than BusyBitmap.Size, an attacker can NULL out bits.





### \_HEAP\_USERDATA\_HEADER Attack

- Attack the new \_HEAP\_USERDATA\_HEADER structure (aka UserBlocks)
- Has a member called BlockStride, which denotes the amount of space between each chunk
  - Also FirstAllocationOffset can be targeted as well
- Used to return the proper chunk to the calling application
  - Chunk = UserBlocks + RandIndex \* BlockStride + FirstAllocationOffset
- Effectively the same as Windows 7 FreeEntryOffset overwrite
- PROBLEMS
  - Guard pages if too many allocations are made of the same size
    - Stagger allocation sizes [i.e. alloc(0x40) x 10; alloc(0x48) x 10, etc)
  - You have to position your overflow-able chunk BEFORE a \_HEAP\_USERDATA\_HEADER structure (which can be challenging)
  - Tainting the \_RTL\_BITMAP structure could cause more instability
  - if ( (ret\_chunk->UnusedBytes & 0x3F) )
    - RtlpLogHeapFailure()





## \_HEAP\_USERDATA\_HEADER Attack

\_HEAP\_USERDATA\_HEADER

<b>Memory</b>
Chunks

+0x0000 - SubSegment	+0x0004 - Reserved	+0x0006 - SizeIndexPadding	+0x000C - Signature
0x0010 - FirstAllocationOffset	0x0012 - BlockStride	+0x0014 – BusyBitmap	+0x001C - BitmapData
_HEAP_ENTRY		_HEAP_ENTRY	
_HEAP_ENTRY		_HEAP_ENTRY	
_HEAP_ENTRY		_HEAP_ENTRY	
+0x0000 - SubSegment	+0x0004 - Reserved	+0x0008 - SizeIndexPadding	+0x000C - Signature
0x0010 - FirstAllocationOffset	0x0012 - BlockStride	+0x0014 – BusyBitmap	+0x001C - BitmapData
_HEAP_ENTRY		_HEAP_ENTRY	
_HEAP_ENTRY		_HEAP_ENTRY	
_HEAP_ENTRY		_HEAP_ENTRY	

Contiguous Memory





# HEAP\_USERDATA\_HEADER Attack

\_HEAP\_USERDATA\_HEADER

Overflow Direction

+0x0000 - SubSegment	+0x0004 - Reserved	+0x0006 - SizeIndexPadding	+0x000C - Signature
0x0010 - FirstAllocationOffset	0x0012 - BlockStride	+0x0014 – BusyBitmap	+0x001C - BitmapData
_HEAP_ENTRY		_HEAP_ENTRY	
_HEAP_ENTRY		_HEAP_ENTRY	
_HEAP_ENTRY		_HEAP_ENTRY	
+0x0000 - SubSegment	+0x0004 - Reserved	+0x0008 - SizeIndexPadding	+0x000C - Signature
0x0010 - FirstAllocationOffset	0x0012 - BlockStride	+0x0014 – BusyBitmap	+0x001C - BitmapData
_HEAP_ENTRY		_HEAP_ENTRY	
_HEAP_ENTRY		_HEAP_ENTRY	
_HEAP_ENTRY		_HEAP_ENTRY	

Contiguous Memory





#### Windows 8 Heap Internals

### **Kernel Pool**





#### Kernel Pool

- Deterministic allocator
  - First chunk allocated from top of page
  - Subsequent chunks allocated bottom-up
- Uses traditional doubly linked free lists
  - Ordered by block size
- Focused on efficiency
  - Uses lookaside lists for small chunks
- Used by drivers and system components





### **Pool Types**

- Generally two types of pool memory
- Non-paged pool
  - Guaranteed to be present at any time
  - Can be accessed by any code, regardless of IRQL
- Paged pool
  - Can be paged out
  - Can only be accessed at IRQL < DPC/Dispatch level</li>





### **Pool Descriptor**

- Each pool is managed by a pool descriptor
- Primarily manages lists of free pool chunks
  - Ordered by block size
    - x86: 8 bytes
    - x64: 16 bytes
  - Used for allocations up to 4080 bytes
- Also keeps track of no. of allocations/frees, pages in use, etc.





#### Pool Header

- Each pool chunk is preceded by a pool header
  - Defines size of previous/current chunk, pool type, associated pool descriptor and process pointer
- kd> dt nt!\_POOL\_HEADER

- +0x000 PreviousSize : Pos 0, 8 Bits

-+0x000 PoolIndex : Pos 8, 8 Bits

-+0x000 BlockSize : Pos 16, 8 Bits

-+0x000 PoolType : Pos 24, 8 Bits

-+0x004 PoolTag : Uint4B

— +0x008 ProcessBilled : Ptr64 \_EPROCESS





#### Windows 8 Heap Internals

#### Windows 8 Kernel Pool





#### Windows 8 Kernel Pool

- Hardened version of the Windows 7 kernel pool
  - No significant structure changes
- Includes a lot more checks
  - Pool header validation (e.g. PoolIndex)
  - Linked list validation
  - Cookies used to protect pointers
- Introduces the non-executable pool
  - Designed to thwart spraying of executable kernel code





#### **NX Pool**

- Windows 8 introduces the non-executable (NX) non-paged pool
  - New pool type: NonPagedPoolNx (0x200)
  - Most non-paged pool allocations now use this
  - NT objects (e.g. reserve objects) can no longer be used to store shellcode
- Requires the system to have enabled DEP
  - If disabled -> nt!ExpPoolFlags & 0x800





### **NX Pool Descriptor**

- Windows 8 allocates two pool descriptors per non-paged pool
  - Executable and non-executable
- Separate non-paged NX lookaside lists
  - KPRCB.PPNxPagedLookasideList
- The kernel calls nt!MmIsNonPagedPoolNx to determine if a chunk is non-executable
  - Looks up the PTE or the PDE (large page) and checks the NX bit
  - E.g. used by the free algorithm





#### Kernel Pool Cookie

- Used to protect pointers referenced by both freed and allocated pool chunks
  - Lookaside lists
  - Process pointers
- Also used to protect certain cache aligned allocations
- Initialized upon boot (nt!InitializePool)
- Randomized with several system counters





#### Windows 8 Pool Cookie Initialization

```
ULONG PTR Value;
KPRCB * Prcb = KeGetCurrentPrcb();
LARGE INTEGER Time;
KeQuerySystemTime(&Time);
Value = rdtsc() ^
                          // tick count
       Prcb->KeSystemCalls ^ // number of system calls
       Prcb->InterruptTime ^ // interrupt time
       Time.HighPart ^ // current system time
       Time.LowPart ^
       ExGenRandom();
                            // pseudo random number
ExpPoolOuotaCookie = (Value) ? Value : 1;
```

From the Windows 8 Developer Preview





### ExGenRandom()

- Generates a pseudo random number
- Based on the Lagged Fibonacci Generator (LFG)
  - -j = 24, k = 55
  - Seeded by boot entropy in the loader parameter block (nt!KeLoaderBlock)
- Used by a number of functions
  - Image base randomization
  - Peb randomization
  - Stack cookie generation





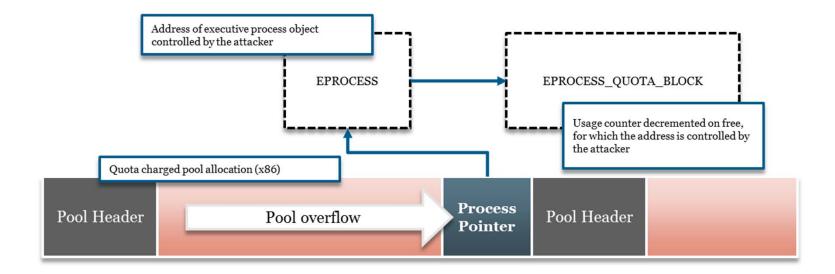
### **Boot Entropy**

- Gathered by winload from six sources
  - OslpGatherSeedFileEntropy
  - OslpGatherExternalEntropy
  - OslpGatherTpmEntropy
  - OslpGatherTimeEntropy
  - OslpGatherAcpiOem0Entropy
  - OslpGatherRdrandEntropy
- The latter uses the RDRAND instruction
  - New PRNG introduced in Ivy Bridge CPUs





#### **Process Pointer Attack**







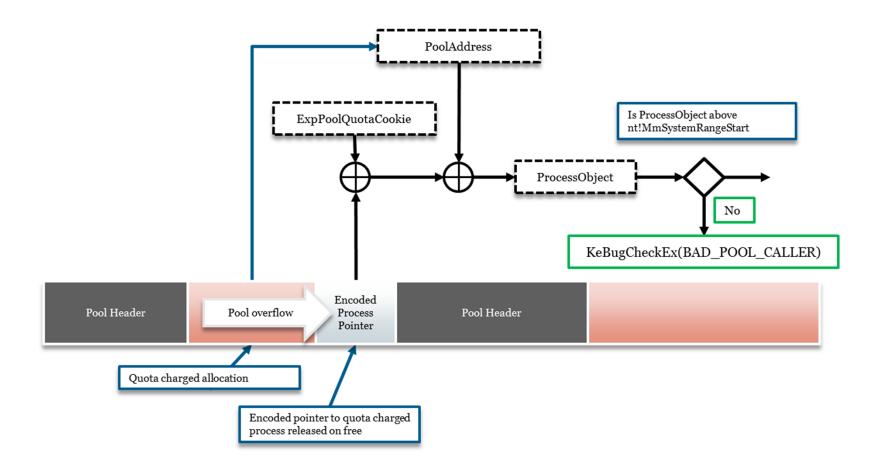
### **Process Pointer Encoding**

- Windows 8 addresses this attack by XOR encoding the process pointer
  - PoolCookie XOR PoolAddress XOR ProcessPointer
  - Also checks if the decoded pointer points into kernel address space (nt!MmSystemRangeStart)
- Checked upon pool free in nt!ExpReleasePoolQuota





### **Process Pointer Encoding**







#### Lookaside Pointer Attacks

- Lookaside lists are used for fast allocation
  - Does not require pool descriptor locking (fast!)
  - Singly linked
  - Atomic compare and swap
- In Windows 7, an attacker could overflow into a freed chunk and corrupt the lookaside list
  - Control the address of the next chunk on the list





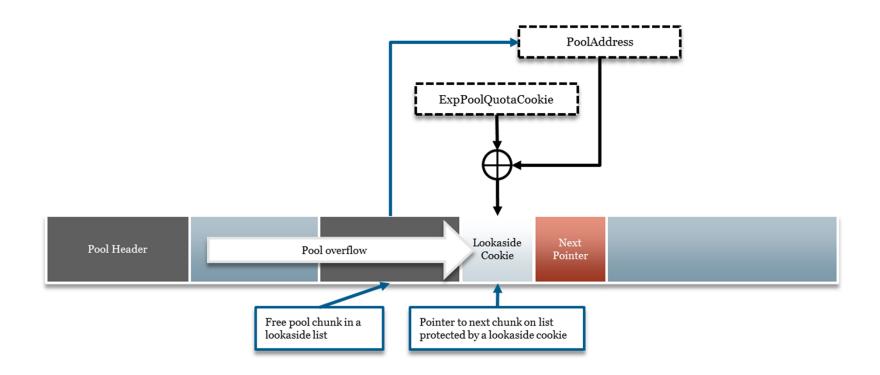
### Lookaside Pointer Encoding

- Windows 8 precedes each lookaside pointer with a randomized cookie
  - PoolCookie XOR PoolAddress
  - Checked before pool allocation
- Also used to protect entries on the deferred free list
- Note: No cookie used for protecting pool page lookaside lists





## Lookaside Pointer Encoding







### Cache Aligned Allocations

- Pool allocations can be requested to be cache aligned
  - PoolType & 4 (e.g. NonPagedPoolCacheAligned)
  - CPU cache line size indicated by nt!ExpCacheLineSize
- When requested, the pool allocator ensures that a suitable cache aligned address is found by increasing the number of bytes requested
  - Rounds up to the nearest cache line size + cache line size
- Favors performance over space usage
  - x86: 0x40 byte request -> 0xC0 byte allocation
  - Does not bother with returning unused bytes





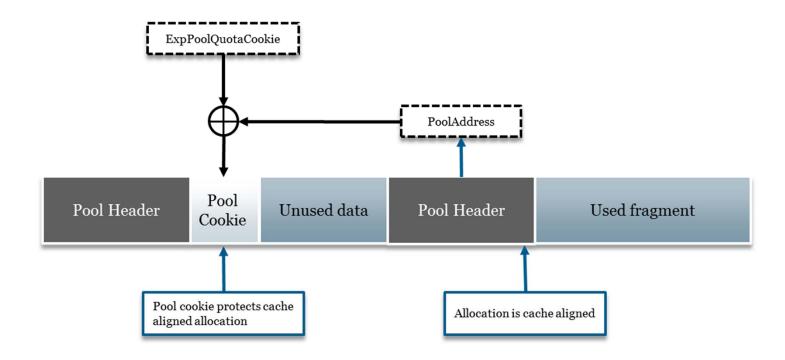
### Cache Aligned Allocation Cookie

- If a cache aligned chunk is returned from the freelists/lookasides, the allocator returns immediately
  - The cache alignment pool type (4) is masked away
  - Nothing is done with the exceeding bytes
- If an unaligned chunk is returned, the allocator returns the address at the nearest cache aligned boundary
  - Returned chunk retains the cache alignment pool type
- In the latter case, the allocator uses the unused pool chunk for storing a cookie
  - \*UnusedChunk = UsedChunk ^ PoolCookie
  - Verification is indicated by the pool type (4) on free





### Cache Aligned Allocation Cookie







## Safe Unlinking

- Introduced in the kernel pool in Windows 7
- Makes sure that adjacent elements on a doubly linked list point to the chunk being unlinked
- Checks were generally made when a chunk was <u>un</u>linked
  - No checks when linking in a pool chunk





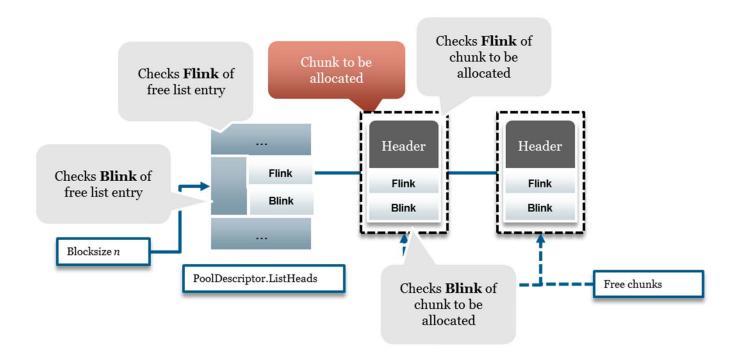
## Safe (Un)linking in Windows 8

- Performs both safe linking and unlinking
  - When allocating chunks from a free list
  - When freeing chunks to a free list
    - This also includes unused pool fragments
- Validates Flink/Blink of both pool descriptor list entry and the chunk to be allocated
  - Incomplete validation in Windows 7 allowed for Flink attacks





# Safe Unlinking in Windows 8







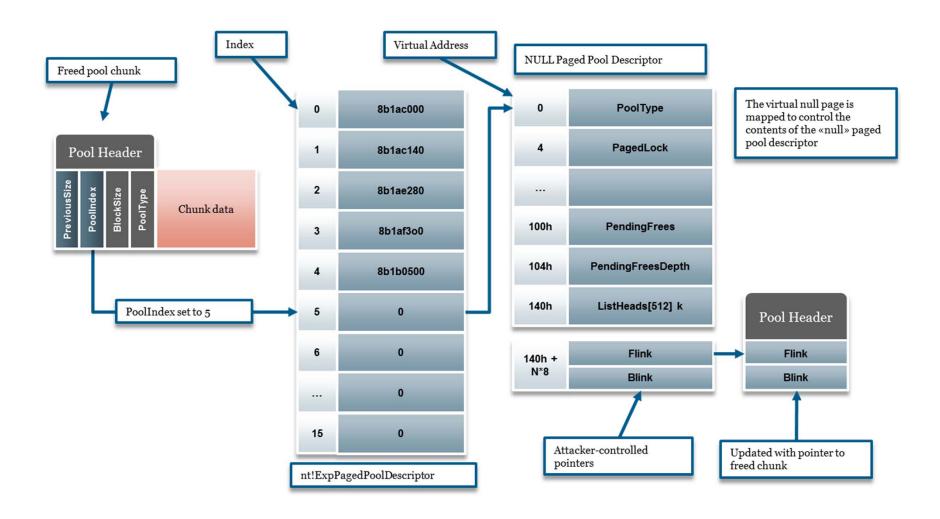
#### PoolIndex Attack

- Windows 7 didn't check the PoolIndex to the associated pool descriptor upon pool free
  - Used as array index for looking up pointer
- An attacker could overwrite the pool index to subsequently control the pool descriptor
  - Out-of-bounds entry would reference a null pointer
  - Mapping the null page allows control of the pool descriptor and where the chunk is inserted





#### PoolIndex Attack







#### PoolIndex Fix

- Windows 8 addresses the PoolIndex attack by checking the value properly before freeing
  - E.g. is PoolIndex < nt!ExpNumberOfPagedPools</p>
- Also prevents user processes from mapping the null page





# Summary

Primitive	Windows Vista	Windows 7	Windows 8 (CP)
Safe Unlinking	×	$\overline{\checkmark}$	$\overline{\checkmark}$
Safe Linking	×	×	$\overline{\checkmark}$
Pool Cookie			
Lookaside Chunks	×	×	$\overline{\checkmark}$
Lookaside Pages	×	×	×
PendingFrees List	×	×	$\overline{\checkmark}$
Cache Aligned Allocations			
PoolIndex Validation	×	×	$\overline{\checkmark}$
Encoded Process Pointer	×	×	$\overline{\checkmark}$
NX Non-Paged Pool	×	×	$\overline{\checkmark}$

<sup>\*</sup> Safe Unlinking: Windows 8 (RP) also addresses the ListHeads Flink attack





#### Windows 8 Heap Internals

#### **Block Size Attacks**





#### **Block Size Attacks**

- The pool header is still subject to attacks as no encoding is used
- Some fields can be hard to properly validate
  - How big is a pool chunk really?
- An attacker can overwrite the block size of a chunk to extend a limited overwrite to an nbyte corruption
  - BlockSize Attack
  - Split Fragment Attack





# BlockSize/PreviousSize

- Used for indicating the size of a block
- Used by the allocator in coalescing
  - Checks if adjacent chunks are free and merges to reduce fragmentation
- Also used in validation upon pool free
  - FreedChunk.BlockSize == NextChunk.PreviousSize
  - The exception to this rule is when the next chunk is on the next page (PreviousSize is <u>null</u>)





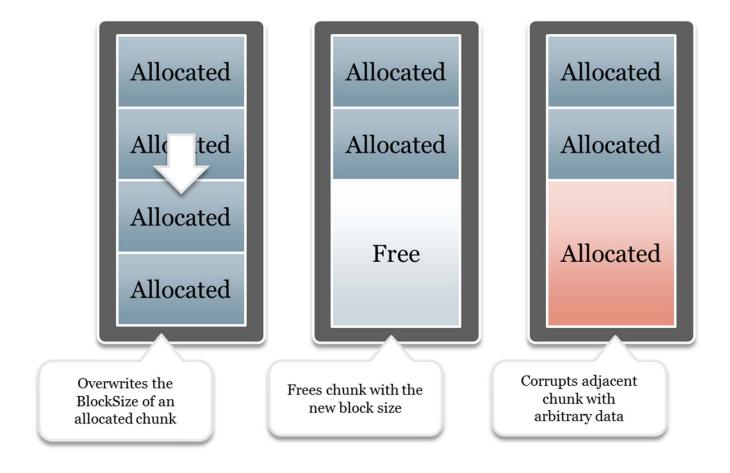
#### BlockSize Attack

- When a chunk is freed, it is put into a free list or lookaside based on its block size
- An attacker can overwrite the block size in order to put it into an arbitrary free list
- Setting the block size to cover the rest of the page avoids the BlockSize/PreviousSize check on free





#### BlockSize Attack







# BlockSize Attack Steps

- Corrupt the block size of an in-use chunk
  - Set it to fill the rest of the page
- Free the corrupted pool chunk
  - Allocator puts the chunk in the free list/lookaside for the new size
- Reallocate the freed memory using something controllable like a unicode string
  - Arbitrary pool corruption





## Split Chunk Pool Allocation

- When requesting a pool chunk, the allocator scans the free lists until a chunk is found
  - If larger than requested, splits and returns the remaining bytes
- A good amount of sanity checking
  - Validates the Flink/Blink of the chunk to be allocated
  - Validates the Flink/Blink of the free list entry
  - Validates the pool index for the allocated chunk
- No validation on block size





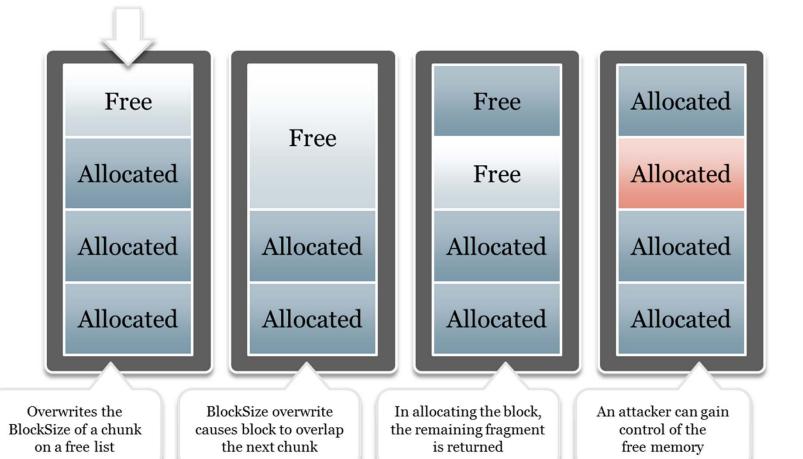
# Split Fragment Attack

- Enables an attacker to extend a 3 byte (semicontrolled) overwrite into a n-byte pool corruption
  - Targets the BlockSize of chunk in a pool descriptor free list
- If BlockSize is set to a larger value, the remaining bytes are returned to the allocator
  - Can free fragments of in-use memory





# Split Fragment Attack







## Split Fragment Attack Steps

- Corrupt the blocksize of a free chunk
  - Set it to something larger
- When the block is allocated, the allocator splits it based on the blocksize value
  - Remaining fragment is returned to the free list
- Reallocate the freed memory using something controllable like a unicode string
  - Arbitrary pool corruption





#### Windows 8 Heap Internals

### Conclusions





#### Determinism

- Unlike the Windows 8 heap, the kernel pool remains highly deterministic
  - Biased towards efficiency, e.g. in the use of lookaside lists for which no locking is required
- Allows an attacker to very accurately manipulate the state of the kernel pool
- Because of this, attacks on pool content is a likely attack vector on Windows 8





#### **Block Size Attacks**

- Block size attacks rely on pool determinism
  - Reducing it could reduce feasibility
- Some block size attacks can be addressed by improving the validation
  - E.g. check if the block size of a chunk held by a free list is of the expected size upon allocation





### **User Land Closing Notes**

- Windows 7 Exploitation tech has been addressed in Windows 8
- Determinism is at an all time low
- That being said, there are still viable attacks
  - \_HEAP\_USERDATA\_HEADER Attack
- Also, since the LFH is grouped by size, useafter-free vulnerability exploitation hasn't too drastically





### Kernel Pool Closing Notes

- Attacks previously demonstrated on Windows
   7 have (mostly) been addressed in Windows 8
  - Proper safe linking and unlinking
  - Randomized cookies used to protect pointers
- Pool header is not protected (e.g. encoded)
  - An attacker can overflow into an in-use chunk
  - No need to repair pool structures
- Pool page lookaside lists are not protected





#### Windows 8 Heap Internals

## Questions?



