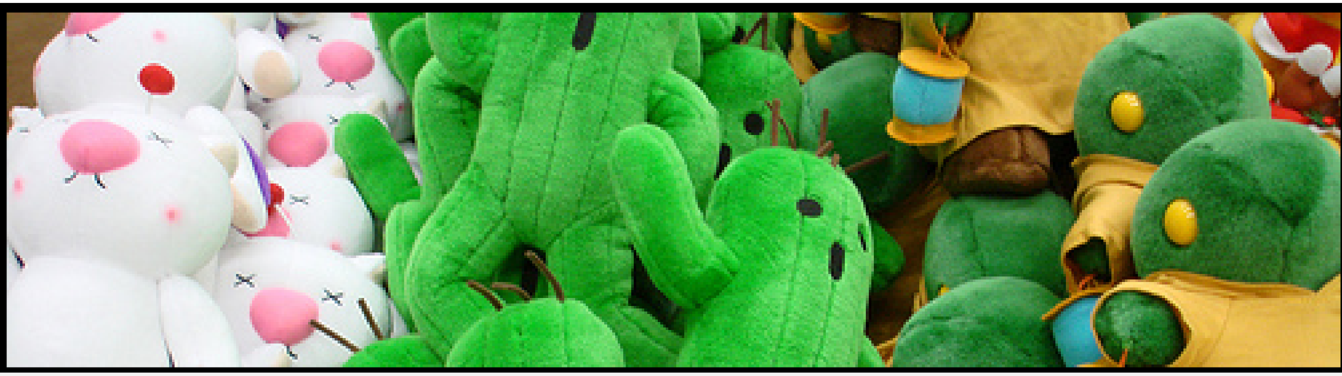
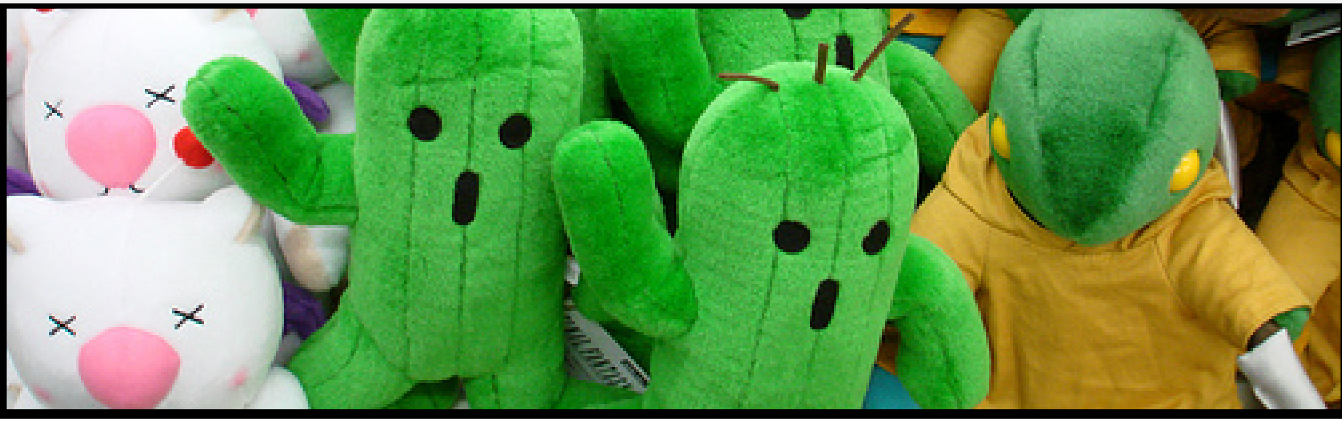


HEAPS OF DOOM



Heaps of Doom

INTRODUCTION

Who

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

What

- **Windows 8 Consumer 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, etc

Why

- This will be the first public examination of the Windows 8 Heap Manager and Kernel Pool Allocator
- Heap exploits that worked on Windows 7 will most likely NOT work on Windows 8
- Let’s find out why

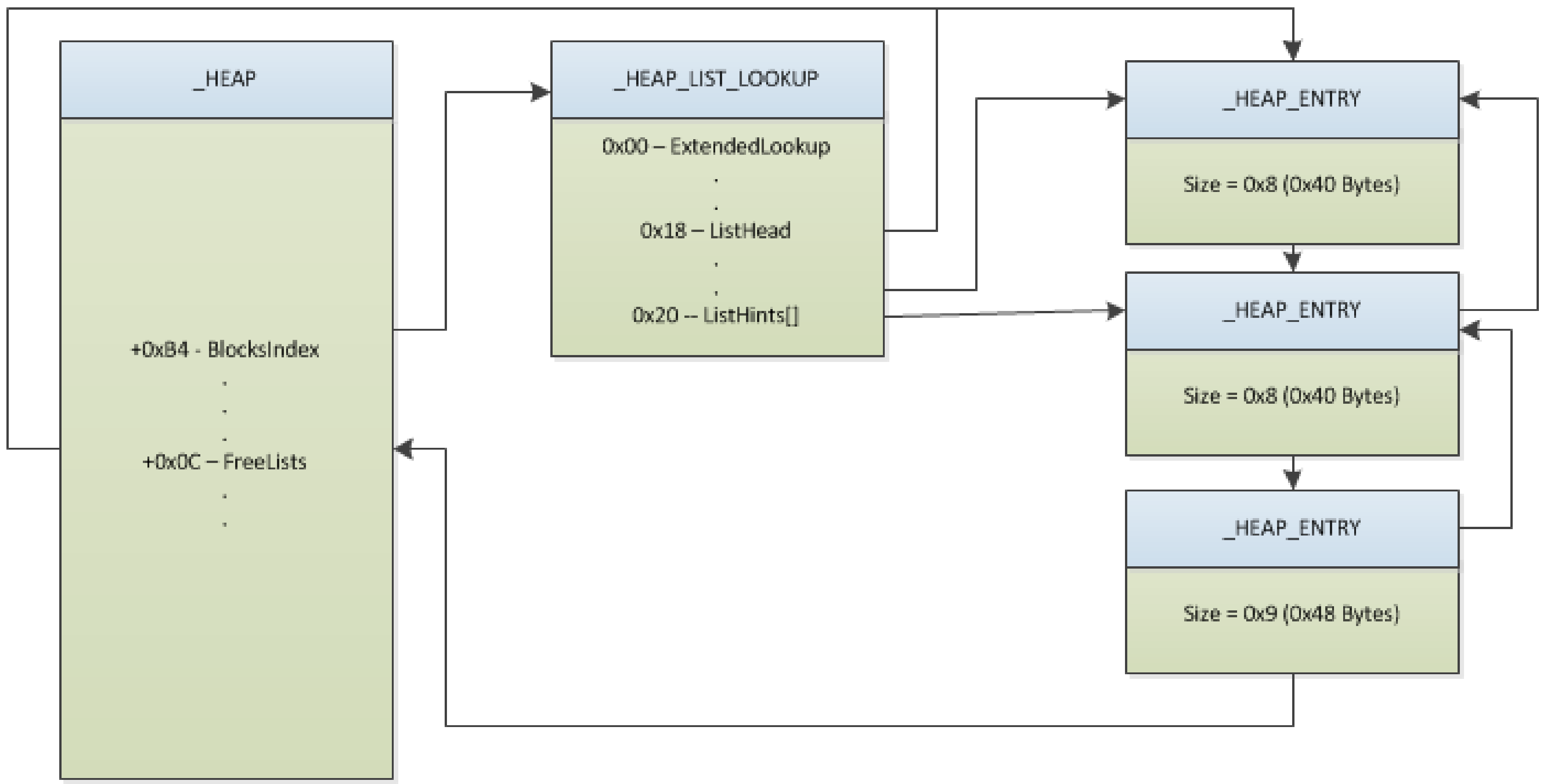
Heaps of Doom

USER LAND BACK-END

Windows 8 Back-end

- Slightly modified version of the Windows 7 back-end [**RtlpAllocateHeap()**]
- Mitigations
  1. Freeing of _HEAP structures is prohibited (R.I.P Ben Hawkes)
  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 semi-predictable locations and sizes
- Changes have been made to add a random offset to the base address when allocating large chunks in **RtlpAllocateHeap**()
- **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;
        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)
                goto cleanup_and_return;

        virtual_chunk = virtual_base + rand_offset;
        if(NtAllocateVirtualMemory(-1, &virtual_chunk, 0, &request_size, MEM_COMMIT, protect) < 0)
                goto cleanup_and_return;

        //XXX Set headers and safe link-in

        return virtual_chunk;
}
```

Heaps of Doom

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 match in BinDiff between Windows 7 and Windows 8
- Mostly the same data structures but offfsets and members have changed a bit

Windows 8 Front-End Mitigations

- Mitigations
  1. FrontEndHeapStatusBitmap[] instead of ListHint->Blink holding the _HEAP_BUCKET
  2. Non-deterministic LFH allocations
     - RtlpLowFragHeapRandomData
     - FreeEntryOffset no longer used
  3. RtlpLowFragHeapAllocFromZone()
     - LFH->SubSegmentZones list check
     - NOT in Windows 7 ☺
  4. ‘Fast Fail’ (Hi Alex Ionescu) used to ensure clean termination
  5. Guard pages between UserBlocks
  6. Segment Offset Free Protection (Sorry Dr. Hawkes)
  7. No more try/catch around Allocator, brute forcing no longer an option

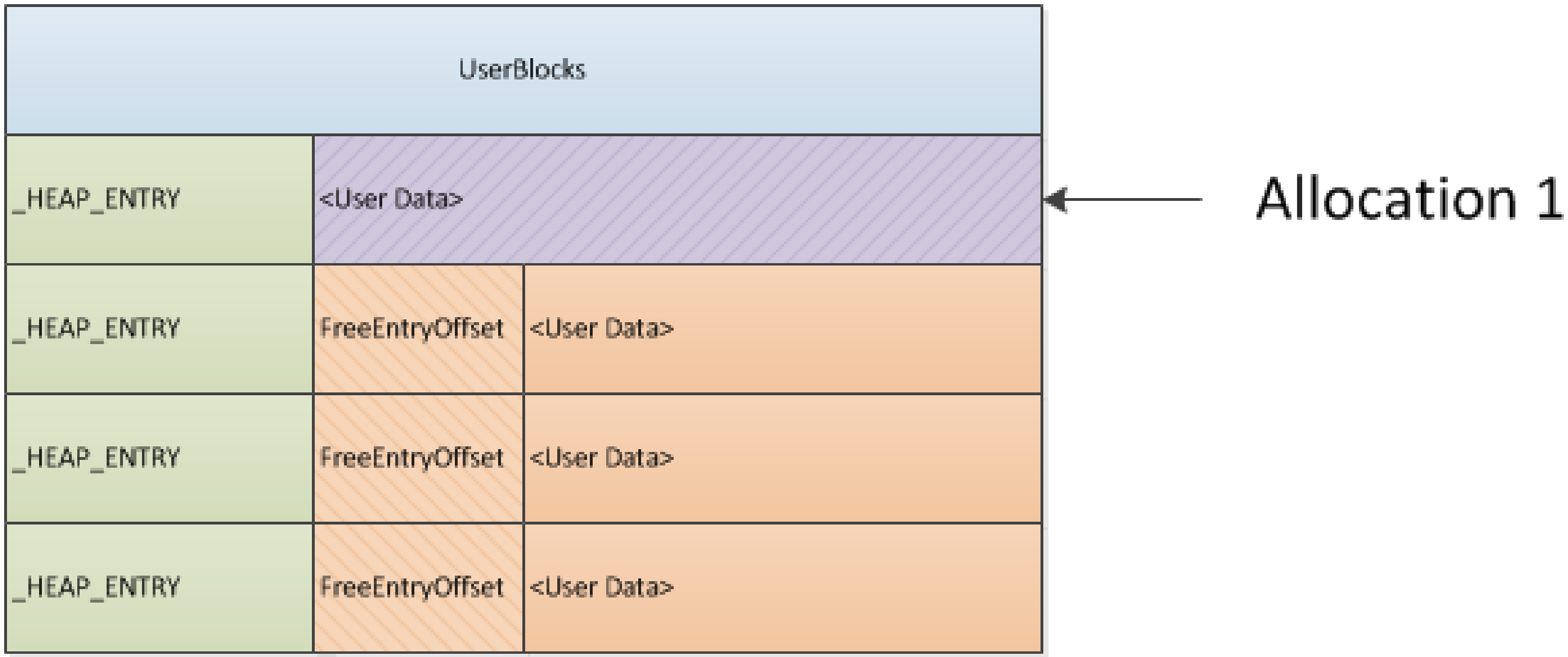
Windows 7 Front-End

_HEAP_USERDATA_HEADER
+0x0000 SubSegment (_HEAP_SUBSEGMENT)
+0x0004 Void *Reserverd
+0x0008 ULONG32 SizeIndex
+0x000C ULONG32 Signature

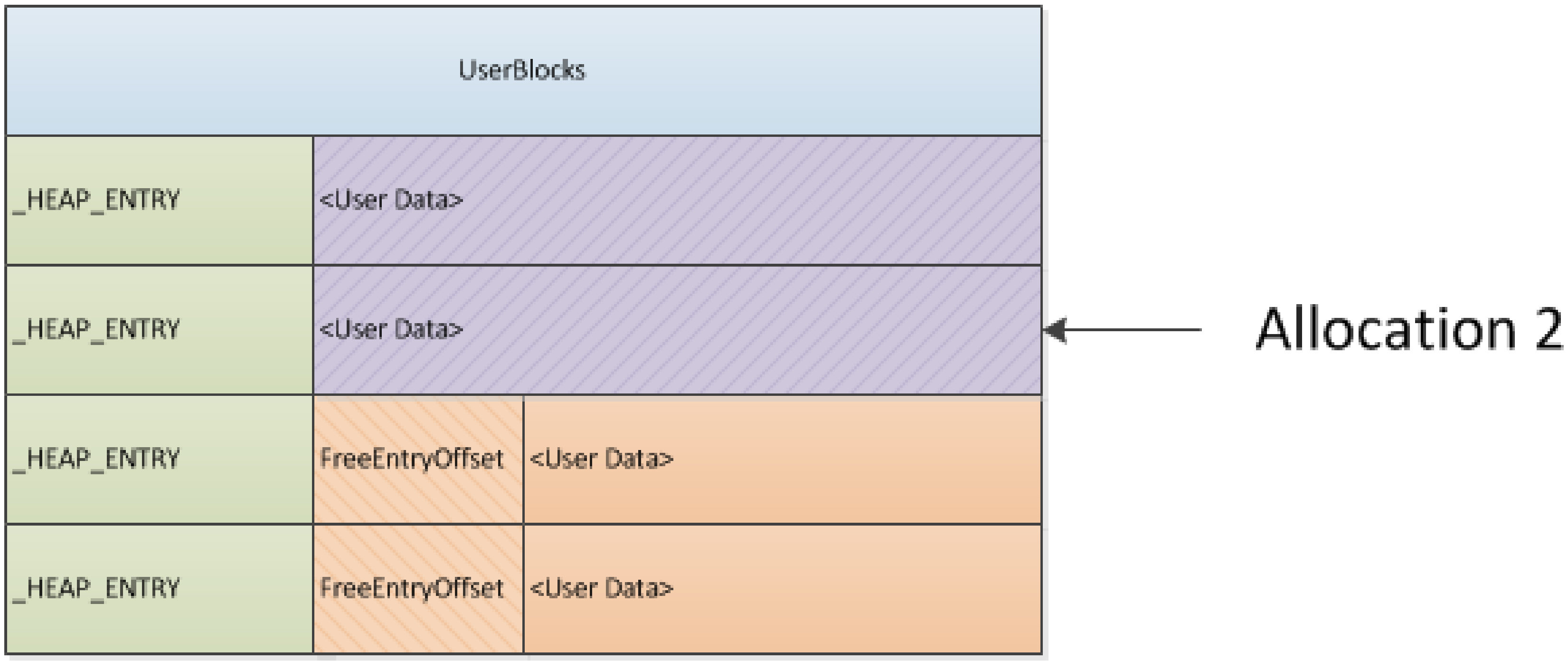
_INTERLOCK_SEQ.Hint (i.e FreeEntryOffset) is gathered from the free chunk w/o validation

UserBlocks		
_HEAP_ENTRY	FreeEntryOffset	<User Data>
_HEAP_ENTRY	FreeEntryOffset	<User Data>
_HEAP_ENTRY	FreeEntryOffset	<User Data>
_HEAP_ENTRY	FreeEntryOffset	<User Data>

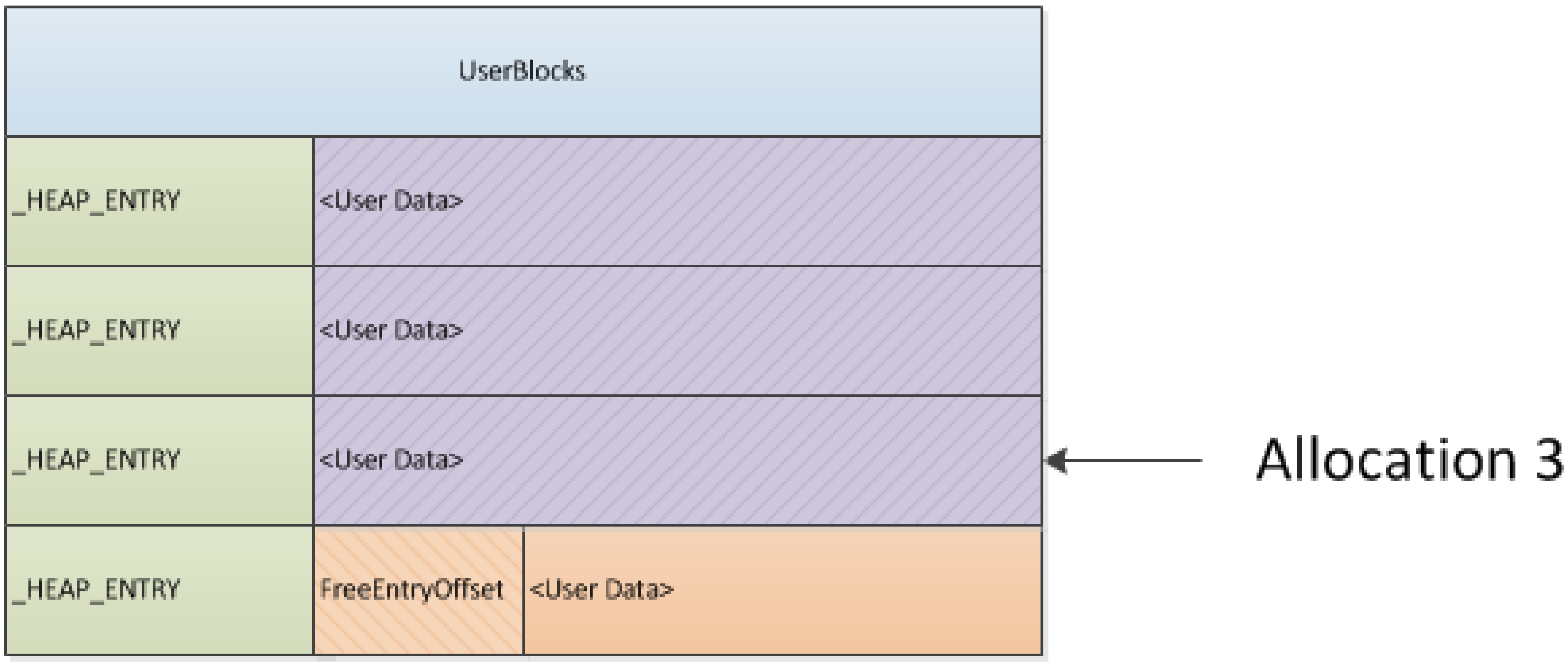
Windows 7 Front-End Allocation I



Windows 7 Front-End Allocation II



Windows 7 Front-End Allocation III



Windows 8 Front-End

_HEAP_USERDATA_HEADER
+0x0000 SubSegment (_HEAP_SUBSEGMENT)
+0x0004 Void *Reserverd

+0x0008
STRUCT
{
UINT8 SizeIndex;
UINT8 GuardPagePresent;
UINT16 PaddingBytes;
}

+0x000C ULONG32 Signature
+0x0010 UINT16 FirstAllocationOffset
+0x0012 UINT16 BlockStride
+0x0014 _RTL_BITMAP BusyBitmap
+0x001C ULONG32 BitmapData

EntryOffset w/in the UserBlocks is kept in _HEAP_ENTRY.PreviousSize

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

UserBlocks	
_HEAP_ENTRY	<User Data>
_HEAP_ENTRY	<User Data>
_HEAP_ENTRY	<User Data>
_HEAP_ENTRY	<User Data>

_RTL_BITMAP
+0x0000 ULONG32 SizeOfBitmap
+0x0004 ULONG32* Buffer

Windows 8 Randomization

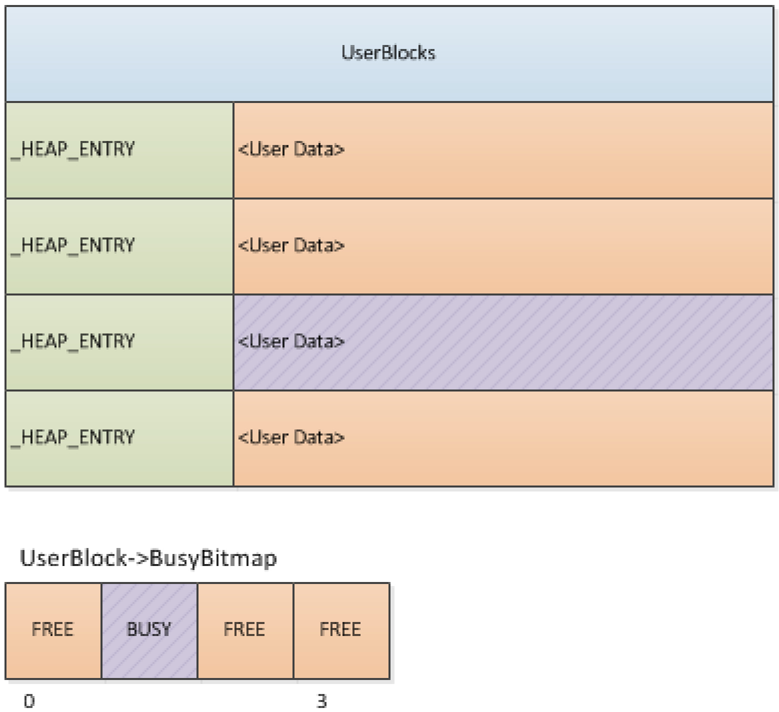
- **RtlpLowFragHeapRandomData** initialized from **RtlpCreateLowFragHeap** and **SlotIndex** is updated on _HEAP_SUBSEGMENT creation [**RtlpSubSegmentInitialize**()]

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

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

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

Windows 8 Front-End Allocation I



Allocation 1 (Simplified)

Slot_index = NtCurrentTeb->LowFragHeapDataSlot

Index = RtlpLowFragHeapRandomData[slot_index]

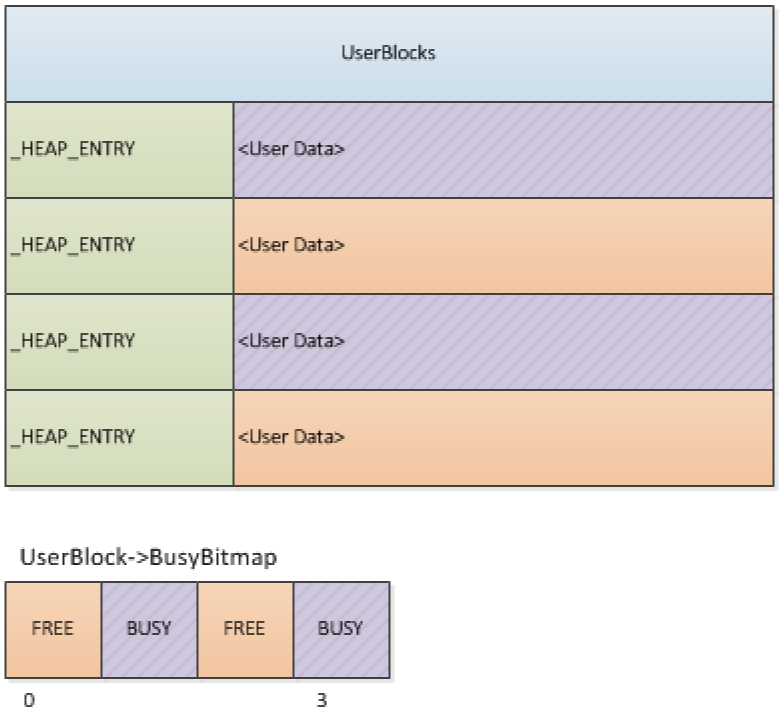
NtCurrentTeb->LowFragHeapDataSlot += 1

Bitmap_index = (index * UserBlock->Bitmap->SizeOfBitmap) >> 7

UpdateBitmap(UserBlock->BusyBitmap)

//If bitmap_index is BUSY then it looks a the next chunk
Return GetChunk(UserBlock->BusyBitmap, bitmap_index)

Windows 8 Front-End Allocation II



Allocation 2 (Simplified)

Slot_index = NtCurrentTeb->LowFragHeapDataSlot

Index = RtlpLowFragHeapRandomData[slot_index]

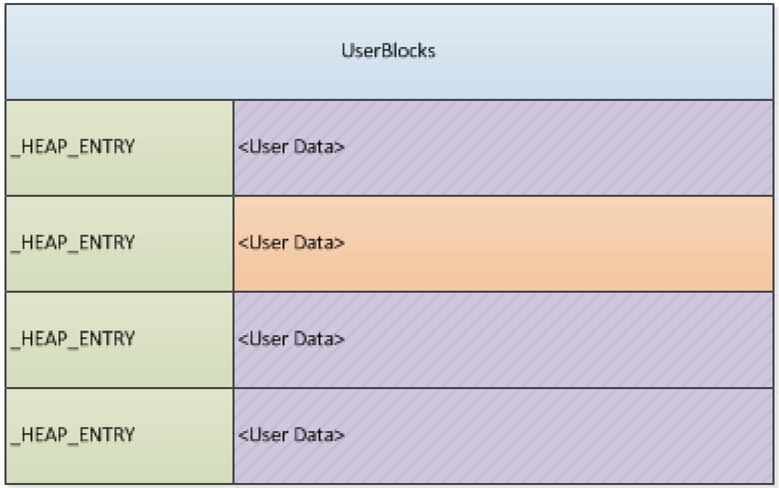
NtCurrentTeb->LowFragHeapDataSlot += 1

Bitmap_index = (index * UserBlock->Bitmap->SizeOfBitmap) >> 7

UpdateBitmap(UserBlock->BusyBitmap)

//If bitmap_index is BUSY then it looks a the next chunk
Return GetChunk(UserBlock->BusyBitmap, bitmap_index)

Windows 8 Front-End Allocation III



UserBlock->BusyBitmap

Allocation 3 (Simplified)

Slot_index = NtCurrentTeb->LowFragHeapDataSlot

Index = RtlpLowFragHeapRandomData[slot_index]

NtCurrentTeb->LowFragHeapDataSlot += 1

Bitmap_index = (index * UserBlock->Bitmap->SizeOfBitmap) >> 7

UpdateBitmap(UserBlock->BusyBitmap)

//If bitmap_index is BUSY then it looks a the next chunk
Return GetChunk(UserBlock->BusyBitmap, bitmap_index)

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 ☹

Windows 8 Front-End Mitigation II

- INT 0x29
  - 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**() and depend on **HeapTerminateOnCorruption** flag

Windows 8 Front-End Mitigation II

- 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 DOES NOT check LFH->SubSegmentZones
  - Speculation in ‘Understanding the LFH’ (paper)
  - Attack LFH->SubSegmentZones for Write-4 primitive ☺

Windows Front-End Mitigation III

- 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, guarding UserBlocks of different sizes
- 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

Windows Front-End Mitigation III

```
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);

       .
       .
}
```

Windows Front-End Mitigation III

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;
}
```

Windows Front-End Mitigation IV

- 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, in-use chunk and free it
  - Win!
- There are checks to ensure that this will no longer work ☹

Windows Front-End Mitigation IV

```
RtlFreeHeap(_HEAP *Heap, DWORD Flags, void *Mem)
{
	.
	.

	//if the header denotes a different segment
	//then adjust the header accordingly
	_HEAP_ENTRY *header = Mem - 8;
	if(Mem - 1 == 0x5)
		header -= 8 * header->SegmentOffset;

	if(!(header->UnusedBytes & 0x3F))
	{
		//this will prevent the chunk from being freed
		RtlpLogHeapFailure(8, Heap, header, 0,0,0);
		header = NULL;
	}

	.
	.
}
```

Windows Front-End Mitigation IV

```
	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 V

- 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 ☹

Summary

Primitive	Windows Vista	Windows 7	Windows 8 (CP)
Reject _HEAP Free	☒	☒	☑
Anti-Determinsm in Back-end	☒	☒	☑
Anti-Determinism in LFH	☒	☒	☑
FrontEndStatusBitmap	☒	☒	☑
Removed try/catch in LFH	☒	☒	☑
Fast Fail	☒	☒	☑
SubSegmentZones Check	☒	☒	☑
Randomized Virtual Alloc	☒	☒	☑
Guard Pages	☒	☒	☑
Segment Offset Free Protection	☒	☒	☑

Are heap meta data attacks dead? – Most are, but there are still some left.

Heaps of Doom

ATTACK OBSERVATIONS

Observation I

- A LFH chunk's index within the UserBlock is still kept in an un-encoded 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
    - Off-by-a-few index write?

Observation II

- 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()

Observation III

- Determinism can still be achieved, but now there exists corner cases
  - Control allocations
  - Have a sequential overflow
  - Only have to worry about the size of the overflow and the chunk that is being overflowed
  - Most likely WON'T be the last chunk in the UserBlock, therefore will overwrite another chunk of the same size, corrupting its data
  - A sequential overwrite stemming from the LAST chunk in a UserBlock w/ a guard page will result in immediate termination ☹
- Also, use-after-free bugs still work (relatively) the same since LFH is grouped by size.
  - Unfortunately due to anti-determinism, the same chunk might not be immediately be returned
  - Make multiple allocation requests or attempt to completely populate a UserBlock, leaving only one chunk

Heaps of Doom

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

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

Heaps of Doom

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
- 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
- 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

ExpPoolQuotaCookie = (Value) ? Value : 1;
```

From the Windows 8 Developer Preview

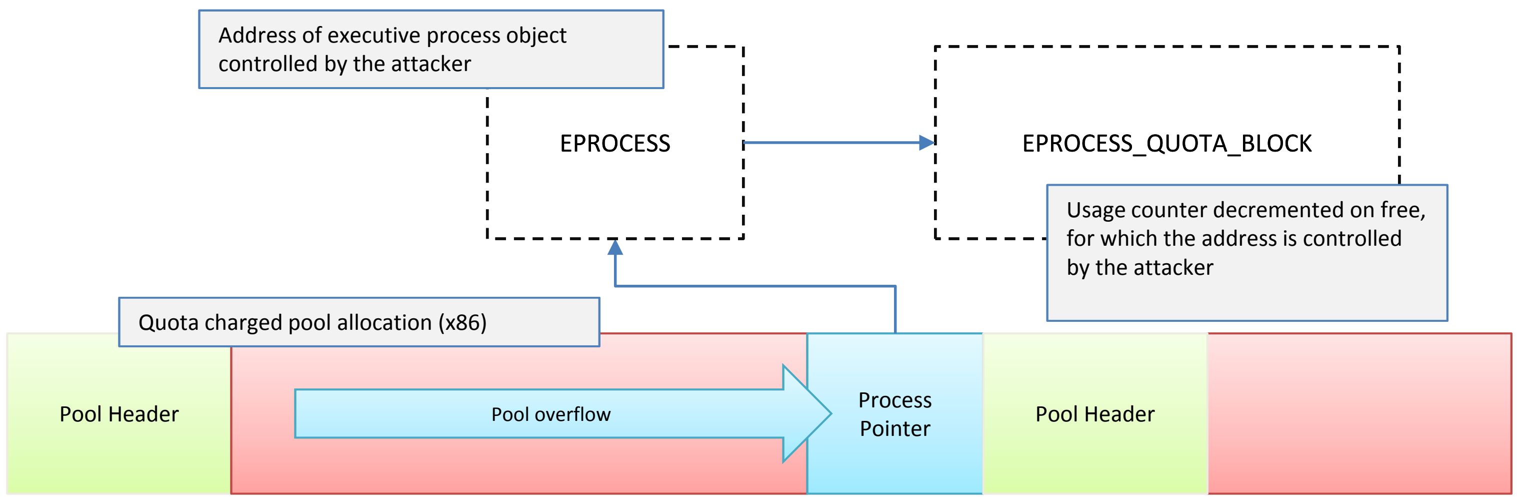
ExGenRandom()

- 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

Process Pointer Attack

- Quota charged allocations store a pointer to associated process object
  - X86: Last 4 bytes of the pool allocation
  - X64: Last 8 bytes of the pool header
- When an allocation is freed, the used quota is returned to the process
- On Windows 7, overwriting the process pointer could allow an attacker to decrement arbitrary memory

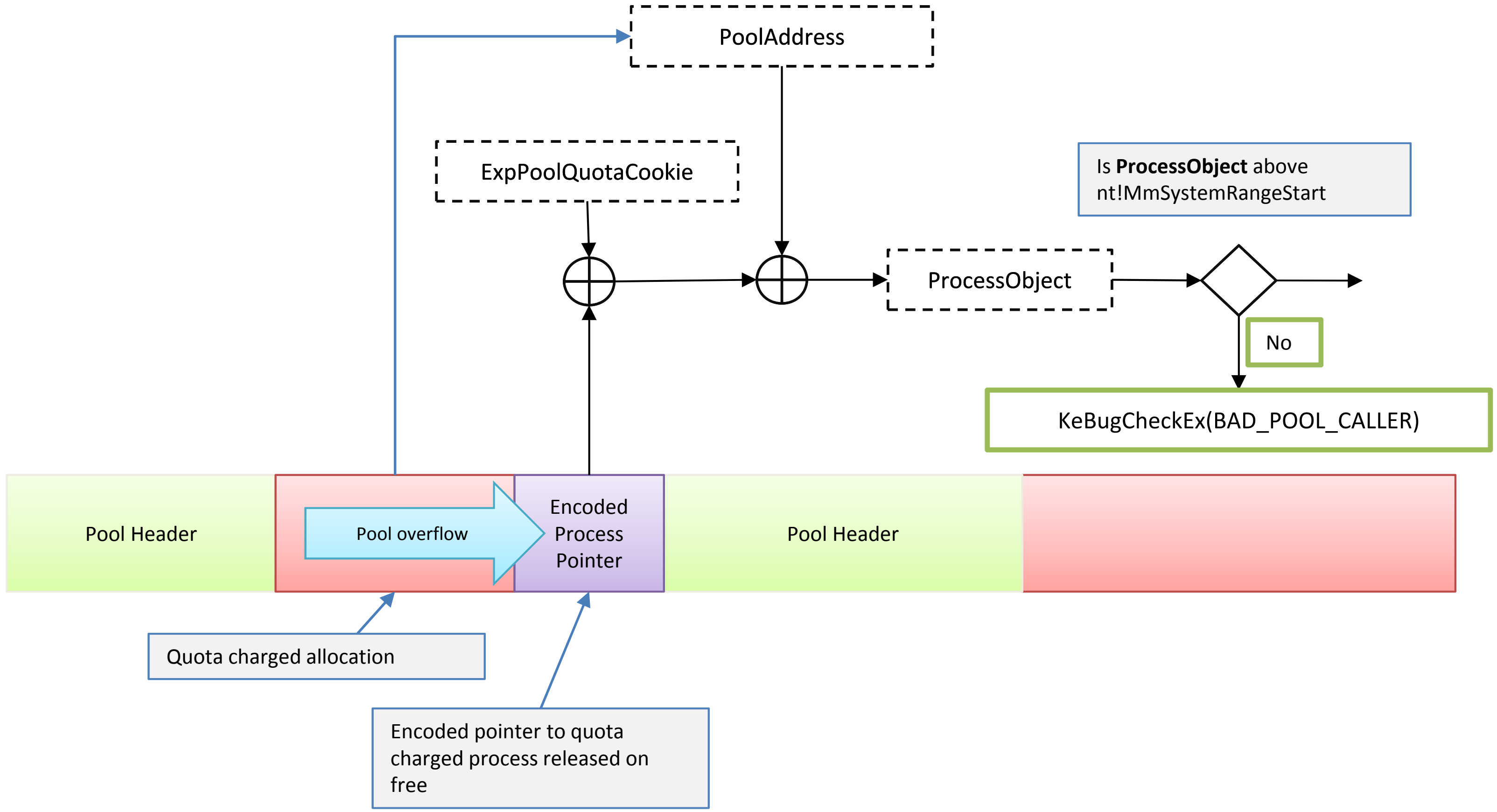
Process Pointer Attack



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



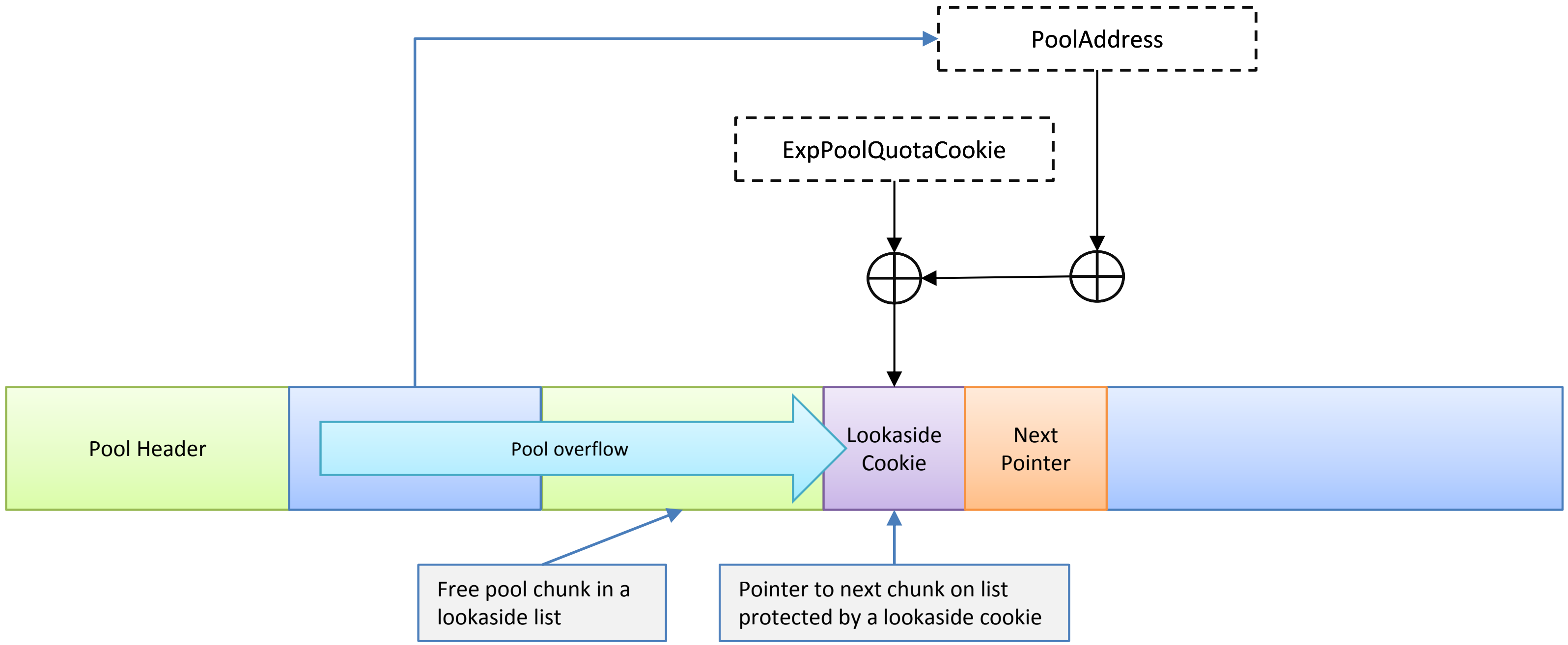
Lookaside Pointer Attacks

- 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



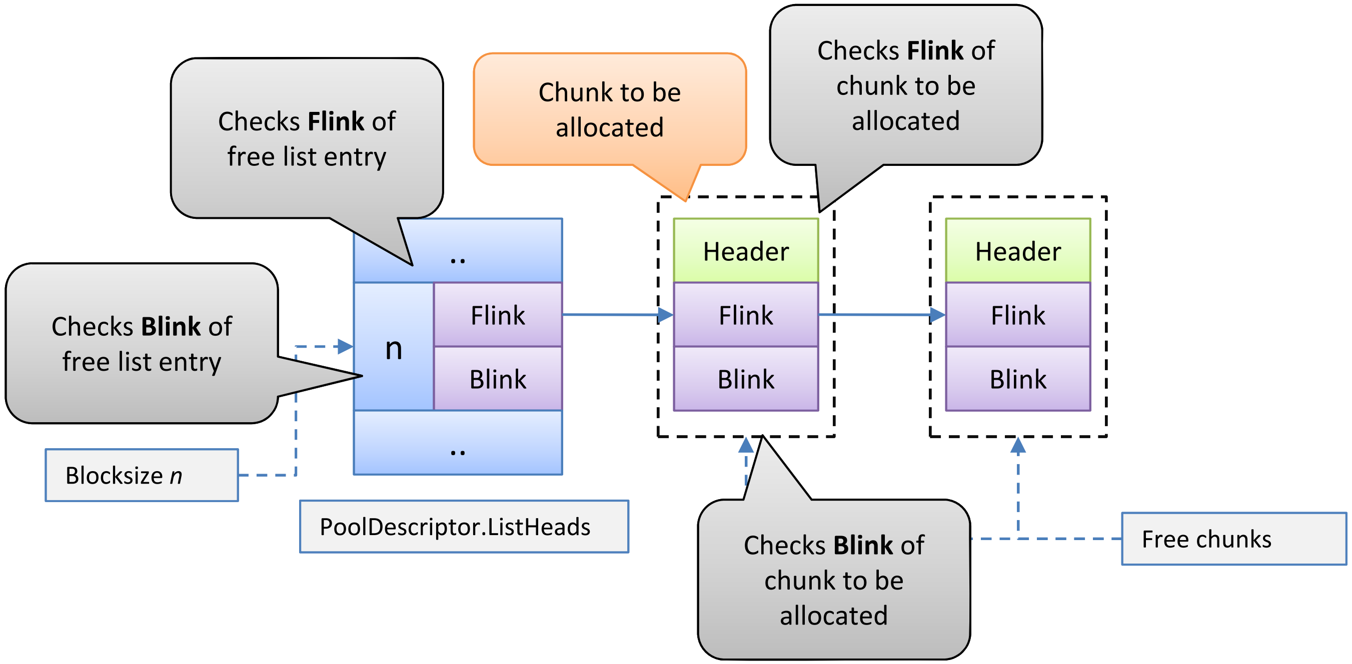
Safe Unlinking

- 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

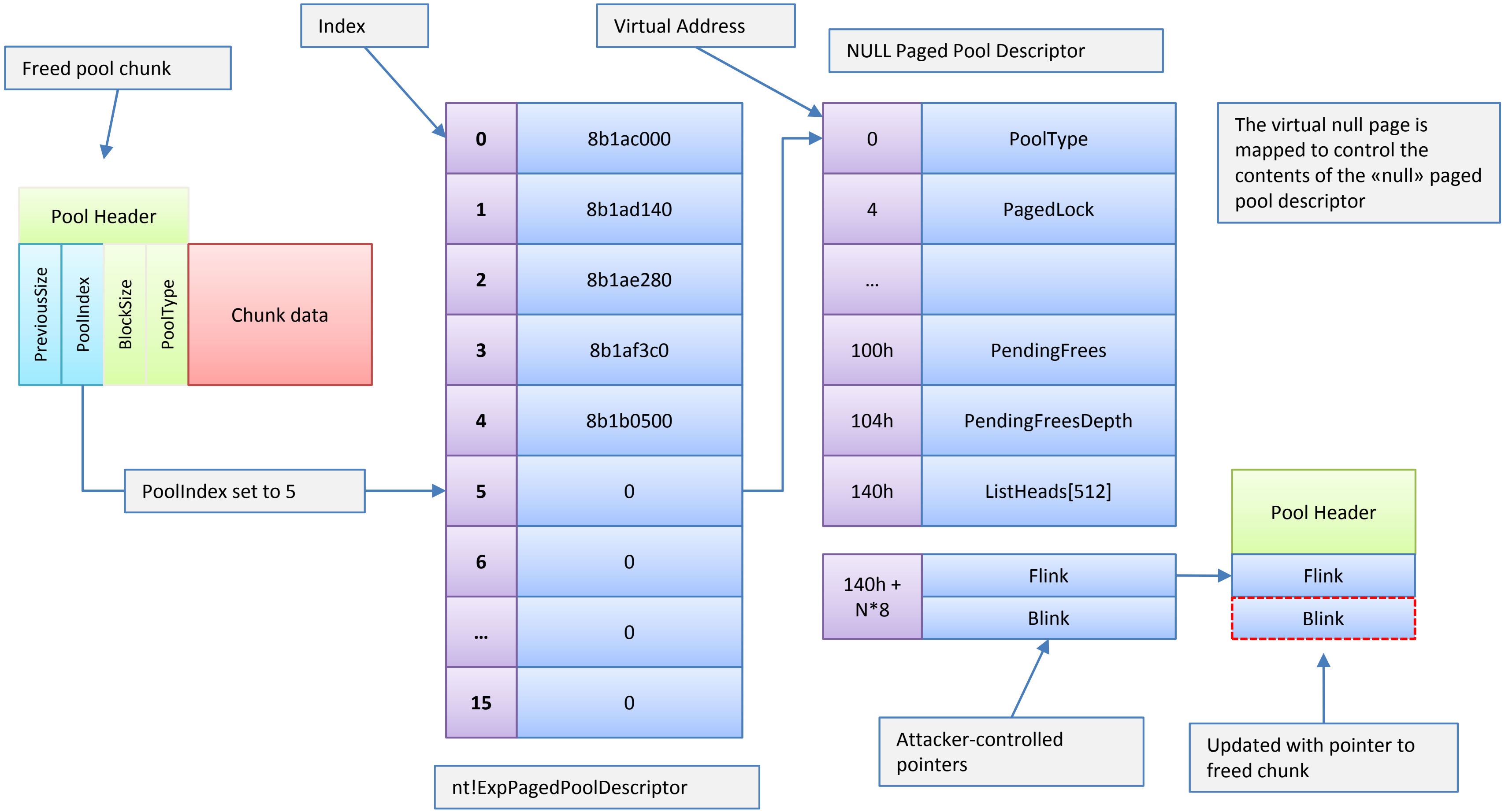
Safe Unlinking in Windows 8



PoolIndex Attack

- Windows 7 didn't check the PoolIndex of 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
- Also prevents user processes from mapping the null page

Summary

Attack	Windows Vista	Windows 7	Windows 8 (CP)
Write-4	☑	☒	☒
List Entry Flink	☑	☑	☒
Lookaside Pointer			
Pool Chunks	☑	☑	☒
Pool Pages	☑	☑	☑
PoolIndex Overwrite	☑	☑	☒
Quota Process Pointer	☑	☑	☒
PendingFrees Pointer	☑	☑	☒

Are pool header attacks dead?

- To a certain degree, yes

Heaps of Doom

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 n-byte corruption
  - BlockSize Attack
  - Split Fragment Attack

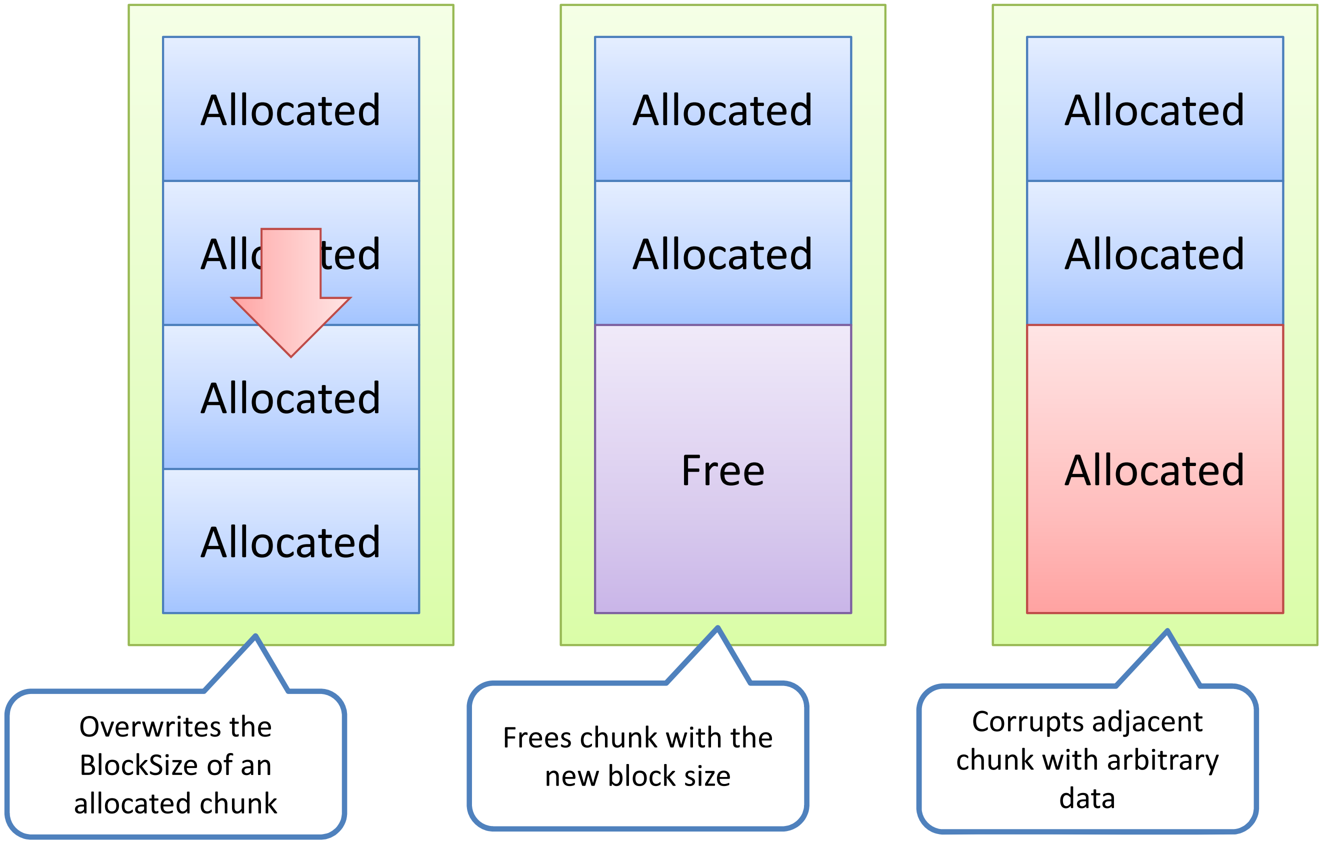
BlockSize/PreviousSize

- 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 null)

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

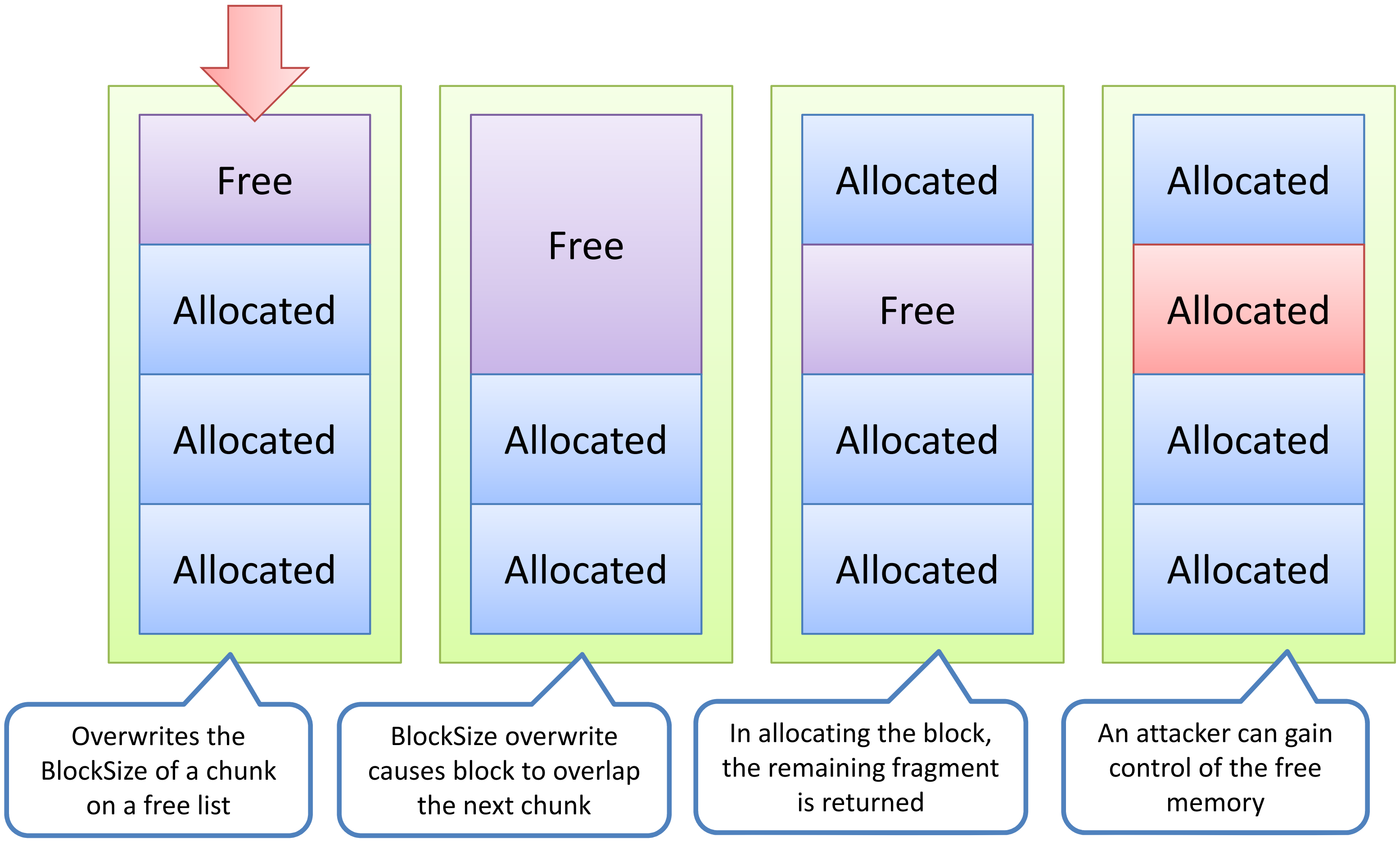
Split Chunk Pool Allocation

- 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 (semi-controlled) 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

Heaps of Doom

CONCLUSION

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

- Most of the techniques in Windows 7 have been remediated
- Determinism is at an all time low
- That being said, there are still viable attacks
  - _HEAP_USERDATA_HEADER
- Also, since the LFH is group by size, use-after-free vulnerability exploitation hasn’t changed much

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