

Real World Kernel Pool Exploitation

Kostya Kortchinsky
Immunity, Inc.

SyScan'08 Hong Kong



Agenda

KNOWING YOU'RE SECURE

- Introduction
- The Kernel Pool and its structures
- Allocation and Free algorithm
- Exploiting a Kernel Pool overflow
- MS08-001: IGMPv3 Kernel Pool overflow
- Conclusion



Introduction



Introduction

KNOWING YOU'RE SECURE

- Kernel vulnerabilities are more popular
 - Less protections than in userland
 - Kernel allocations are highly performance optimized
 - Less room for cookies and other roadblocks
 - Code has had less attention than userland services
- Several kernel pool overflows to exploit
 - Our example: MS08-001 (IGMPv3 Overflow)

Population

KNOWING YOU'RE SECURE

People who write exploits

People who write
Windows
overflows

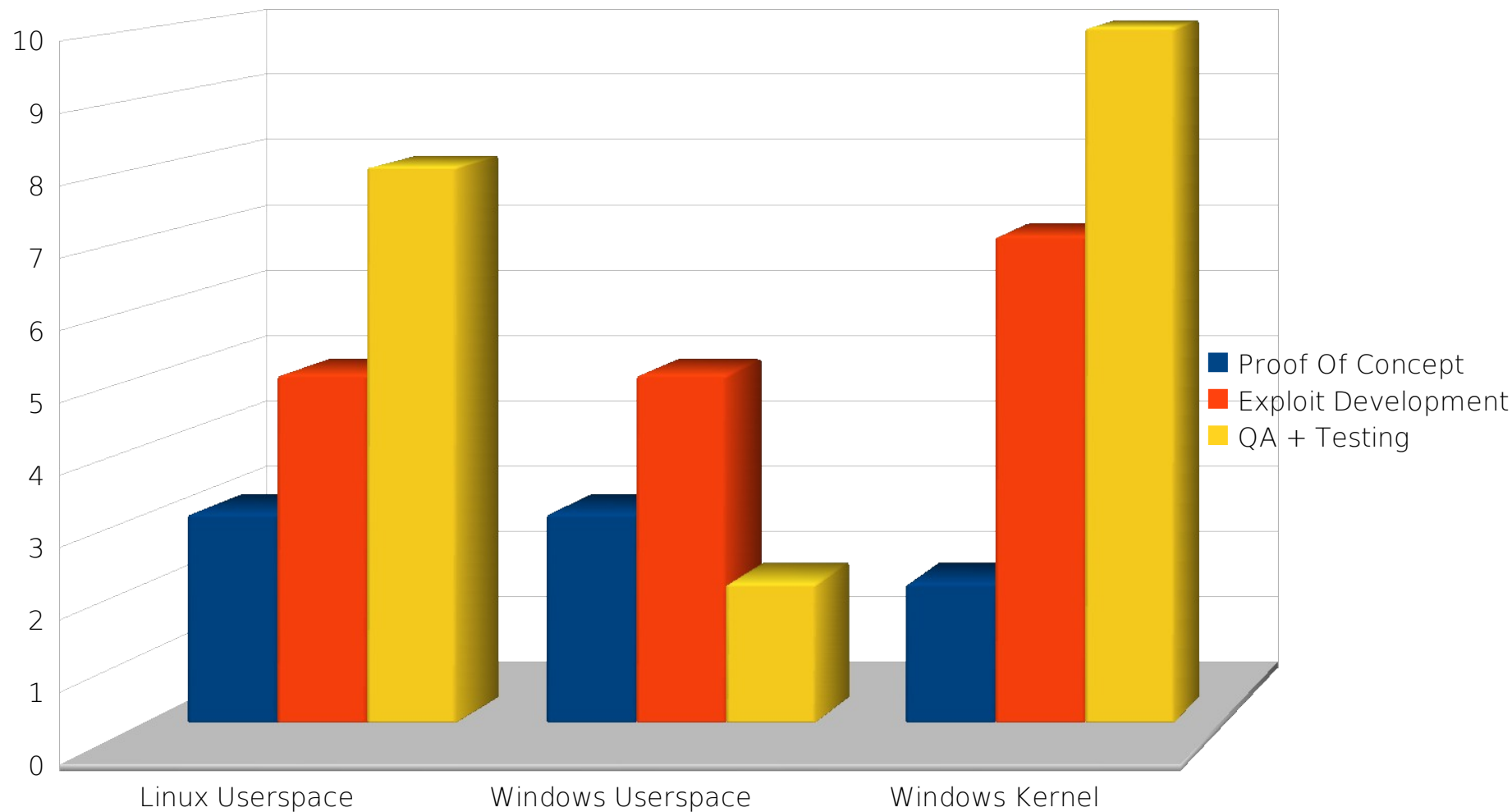
People who
write Windows
Kernel Pool
Overflows

Other considerations

KNOWING YOU'RE SECURE

- Because few people know how to write them, Windows Kernel Pool overflows are often mischaracterized as “Denial of Service”
- Huge investment in exploit development before it is known if a reliable exploit can be created for any bug
 - If it costs 100K dollars to even know if you have a risk from any particular vulnerability, can you afford to find out?

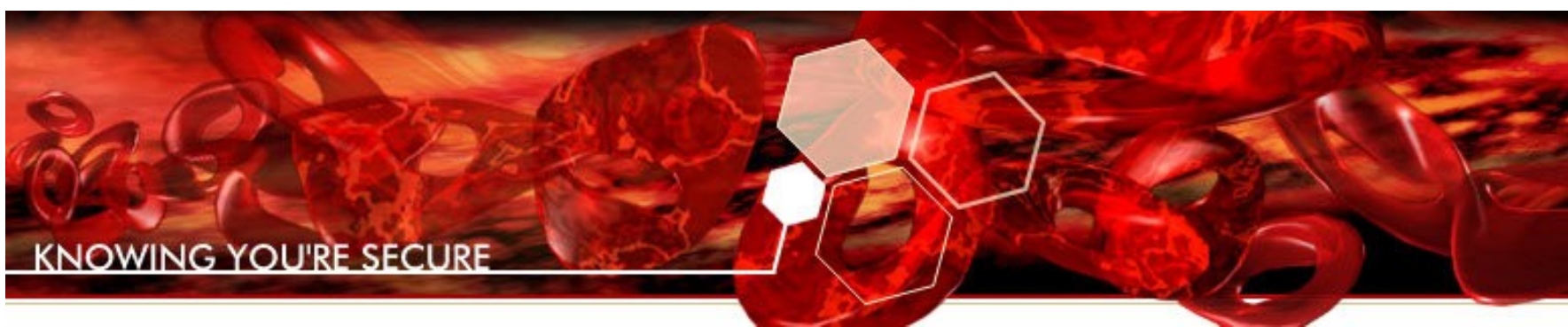
Diversity increases QA costs dramatically



Addresses May Vary

KNOWING YOU'RE SECURE

- The following slides assume:
 - Structures, constants and offsets are from ntkrnlpa.exe 5.1.2600.2180 (XP SP2 default)
 - Windows XP SP2 has 4 kernels, the one being used depends on PAE and MP
 - nt!KeNumberNodes is 1 (non NUMA architecture)
 - NUMA introduces differences in the kernel pools
 - See slide #71



The Kernel Pool and its structures

Kernel Pool vs. Userland Heap

KNOWING YOU'RE SECURE

- Quite similar
- Only a few pools for all the kernel allocations
 - Think LSASS default heap
- Kernel pool is designed to be fast
 - As few checks as possible
 - No kernel pool cookie
 - No kernel pool safe unlink



Kernel Pool

KNOWING YOU'RE SECURE

- Used by Windows for dynamic memory allocations within kernel land by functions:
 - `nt!ExAllocatePool`, `nt!ExAllocatePoolWithTag`, ...
 - `nt!ExFreePool`, `nt!ExFreePoolWithTag`, ...
- There are several kernel pools, default being:
 - One *non-paged* pool
 - Two *paged* pools
 - One *session paged* pool



Pools Definitions

KNOWING YOU'RE SECURE

- Pools are defined thanks to structure `nt!_POOL_DESCRIPTOR`
 - Described in slide #16
- They are stored in the `nt!PoolVector` array
 - Array of pointers
 - Located in the `.data` section of the kernel

Non Paged Pool

KNOWING YOU'RE SECURE

- *Non pageable* system memory
- Can be accessed from any IRQL
- Scarce resource
- Descriptor for non paged pool is static:
 - nt!NonPagedPoolDescriptor in .data section
- It is initialized in nt!InitializePool
- Pointer to the descriptor is stored in entry 0 of nt!PoolVector



Paged Pool

KNOWING YOU'RE SECURE

- *Pageable* system memory
- Number of paged pools defined by nt!
ExpNumberOfPagedPools (default is **2**)
- An array of pool descriptors is dynamically allocated and initialized in nt!InitializePool
 - NonPagedPool allocation
 - One more descriptor than number of paged pools
- Pointer to array stored in entry 1 of nt!
PoolVector and in nt!ExpPagedPoolDescriptor

Session Paged Pool

KNOWING YOU'RE SECURE

- *Pageable* system memory
- Descriptor for the session paged pool is located in the session space
 - PagedPool member of nt!MmSessionSpace structure
 - Usually `0xbf7f0000+0x244`
- Initialized in `nt!MiInitializeSessionPool`
 - used for session space allocations
 - do not use lookaside lists
- Pointer to the descriptor stored in `nt!ExpSessionPoolDescriptor`

nt!_POOL_DESCRIPTOR

KNOWING YOU'RE SECURE

```
kd> dt nt !_POOL_DESCRIPTOR
+0x000 Pool Type           : _POOL_TYPE
+0x004 Pool Index         : Uint4B
+0x008 RunningAllocs      : Uint4B
+0x00c RunningDeAllocs    : Uint4B
+0x010 Total Pages        : Uint4B
+0x014 Total BigPages     : Uint4B
+0x018 Threshold          : Uint4B
+0x01c LockAddress        : Ptr32 Void
+0x020 PendingFrees       : Ptr32 Void
+0x024 PendingFreeDepth  : Int4B
+0x028 ListHeads          : [512] _LIST_ENTRY
```



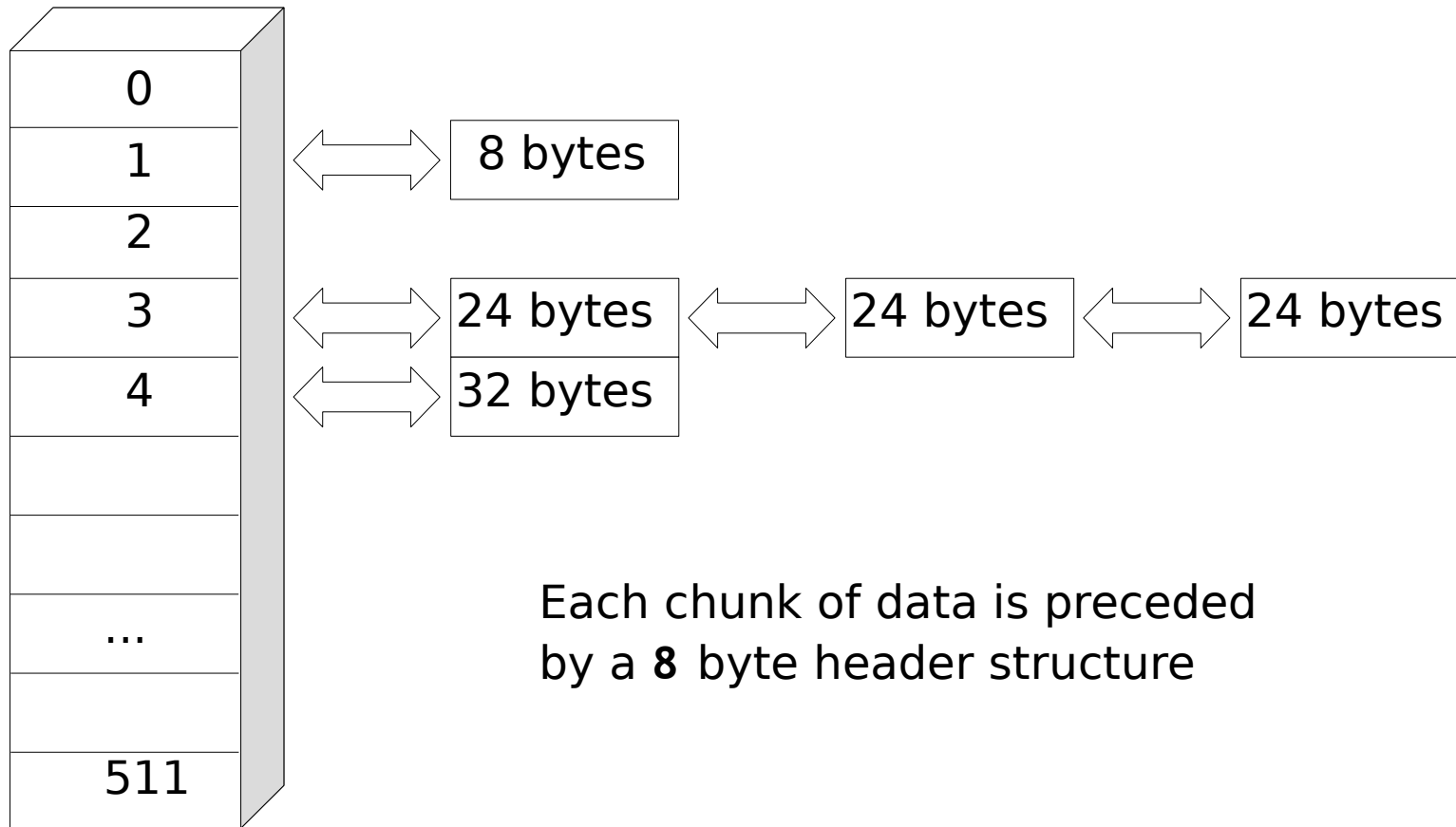

Pool Descriptor

KNOWING YOU'RE SECURE

- Some of the members:
 - *PoolType*: NonPagedPool=0, PagedPool=1, ...
 - *PoolIndex*: 0 for non-paged pool and paged session pool, index of the pool descriptor in nt!ExpPagedPoolDescriptor for paged pools
 - *ListHeads*: 512 double-linked lists of free memory chunks of the same size (8 byte granularity)
 - List number for a requested allocation size is calculated by $\text{Block Size} = (\text{Number Of Bytes} + 0xf) \gg 3$
 - Thus can be used for allocations up to 0xff0 (4080) bytes

ListHeads

KNOWING YOU'RE SECURE



nt!_POOL_HEADER

KNOWING YOU'RE SECURE

```
kd> dt nt !_POOL_HEADER
+0x000 PreviousSize      : Pos 0, 9 Bits
+0x000 PoolIndex        : Pos 9, 7 Bits
+0x002 BlockSize       : Pos 0, 9 Bits
+0x002 PoolType        : Pos 9, 7 Bits
+0x000 Ulong1          : Uint 4B
+0x004 ProcessBilled   : Ptr 32 _EPROCESS
+0x004 PoolTag         : Uint 4B
+0x004 AllocatorBackTraceIndex : Uint 2B
+0x006 PoolTagHash     : Uint 2B
```



Chunk Header

KNOWING YOU'RE SECURE

- Explanations on some of the members:
 - *PreviousSize*: *BlockSize* of the preceding chunk
 - 0 if chunk is located at the beginning of a page
 - *BlockSize*: Number of bytes requested plus header size rounded up to a multiple of 8, divided by 8
 - or $(\text{Number Of Bytes} + 0xf) \gg 3$
 - *PoolIndex*: same definition as for the descriptor
 - *PoolType*: 0 if free, $(\text{PoolType} + 1) | 4$ if allocated
 - *PoolTag*: Usually 4 printable characters identifying the code responsible for the allocation

Free Chunk Header

KNOWING YOU'RE SECURE

- When *PoolType*=0, the chunk header is followed by a `nt!_LIST_ENTRY` structure
 - This is the entry pointed to by the ListHeads double linked list
- Chunks freed to the lookaside lists remain the same, their *PoolType* is non 0

```
kd> dt nt!_LIST_ENTRY
```

```
+0x000 Flink : Ptr32 _LIST_ENTRY
```

```
+0x004 Blink : Ptr32 _LIST_ENTRY
```

Lookaside Lists

KNOWING YOU'RE SECURE

- Like for userland heaps, kernel uses lookaside lists for faster allocating and freeing of small chunks of data
 - Maximum *BlockSize* being 32 (or 256 bytes)
- They are defined in the processor control block
 - 32 entry `PPPagedLookasideList`
 - 32 entry `PPNPagedLookasideList`
- Each entry holds **2** single chained lists of nt!
`_GENERAL_LOOKASIDE` structures: one “per processor” *P*, one “system wide” *L*

nt!_GENERAL_LOOKASIDE

KNOWING YOU'RE SECURE

```
kd> dt nt !_GENERAL_LOOKASIDE
+0x000 ListHead          : _SLIST_HEADER
+0x008 Depth             : Uint2B
+0x00a MaximumDepth     : Uint2B
+0x00c TotalAllocates   : Uint4B
+0x010 AllocateMisses   : Uint4B
+0x010 AllocateHits     : Uint4B
+0x014 TotalFrees       : Uint4B
+0x018 FreeMisses       : Uint4B
+0x018 FreeHits         : Uint4B
+0x01c Type              : _POOL_TYPE
+0x020 Tag               : Uint4B
+0x024 Size              : Uint4B
+0x028 Allocate         : Ptr32 void*
+0x02c Free              : Ptr32 void
+0x030 ListEntry        : _LIST_ENTRY
+0x038 LastTotalAllocates : Uint4B
+0x03c LastAllocateMisses : Uint4B
+0x03c LastAllocateHits  : Uint4B
+0x040 Future           : [2] Uint4B
```

nt! MmNonPagedPoolFreeListHead

KNOWING YOU'RE SECURE

- Static array of 4 double linked lists for non-paged free chunks bigger than a page
 - Index in the array is (SizeOfChunkInPages-1)
 - Last entry also has everything bigger than 4 pages

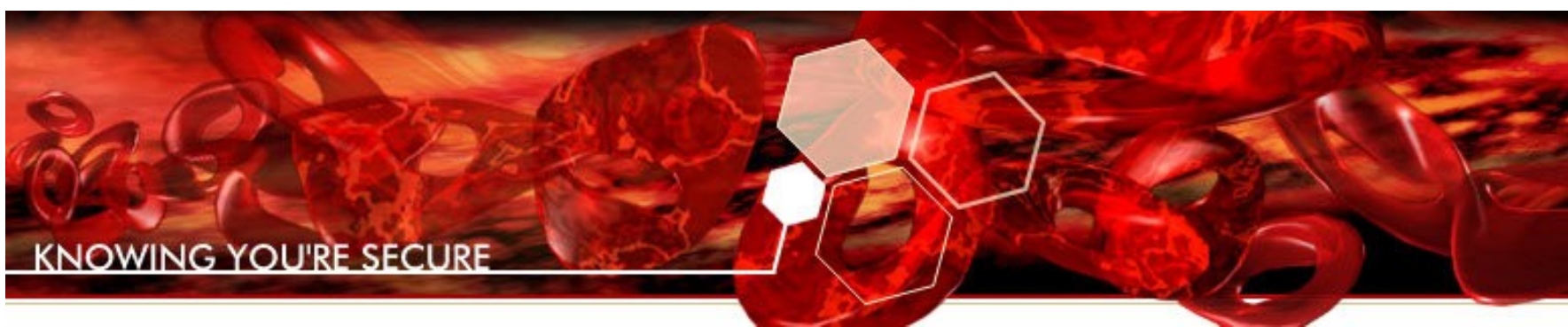
- Structure used is nt!

_MMFREE_POOL_ENTRY

```
kd> dt nt !_MMFREE_POOL_ENTRY
```

```
+0x000 List          : _LIST_ENTRY
+0x008 Size          : Uint4B
+0x00c Signature    : Uint4B
+0x010 Owner        : Ptr32 _MMFREE_POOL_ENTRY
```

- Thus free non-paged “big” chunks are linked through the pages themselves



Allocation and Free algorithms

Allocation Summary

KNOWING YOU'RE SECURE

- Windows kernel pool allocates in small chunks:
 - 8 byte granularity up to 4080 bytes (included)
 - Used to be 32 for Windows 2000
 - Page granularity above
- Makes extensive use of optimized lists:
 - Single linked Lookaside lists up to 256 bytes
 - Double linked ListHeads lists up to 4080 bytes
- Splits an entry if a chunk of the exact size cannot be found
- Expands the pool if needed

Simplified Allocation Algorithm

KNOWING YOU'RE SECURE

nt!ExAllocatePoolWithTag (1/2)

- If NumberOfBytes > 0xffff:
 - Call nt!MiAllocatePoolPages
- If PagedPool requested:
 - If BlockSize ≤ 0x20:
 - Try the “per processor” paged lookaside list
 - If failed, try the “system wide” paged lookaside list
 - Return on success
 - Try and lock a paged pool descriptor
- Else:
 - If BlockSize ≤ 0x20:
 - Try the “per processor” non-paged lookaside list

Simplified Allocation Algorithm

KNOWING YOU'RE SECURE

nt!ExAllocatePoolWithTag (2/2)

- If failed, try the “system wide” non-paged lookaside list
- Return on success
- Try and lock the non-paged pool descriptor
- Use ListHeads of currently locked pool:
 - Use 1st non empty ListHeads[n]
 - With $\text{BlockSize} \leq n < 512$
 - Split entry if bigger than needed
 - Return on success
 - If failed, expand the pool by adding a page
 - Try again!



Free Chunk Splitting

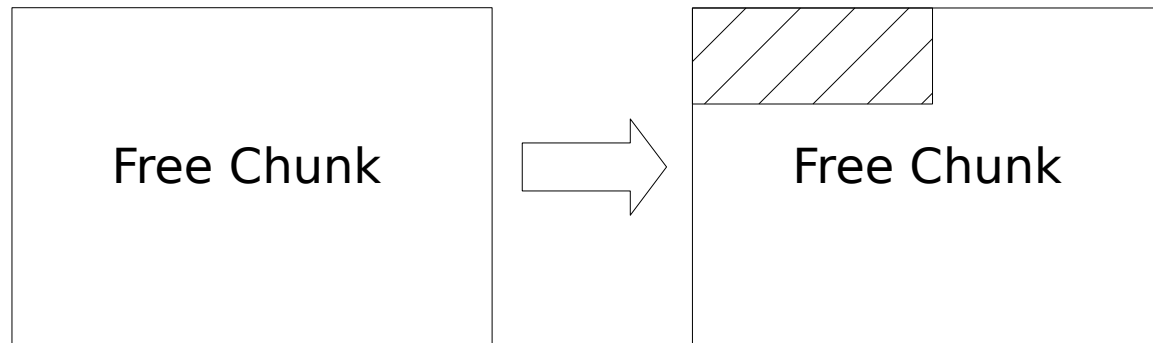
KNOWING YOU'RE SECURE

- If the algorithm has selected a chunk larger than the requested `NumberOfBytes`, it is split
 - If the chunk is at the start of a page:
 - Take the allocation from the *front* of the chunk
 - Otherwise:
 - Take the allocation from the *end* of the chunk
- The remaining part is inserted in the correct list

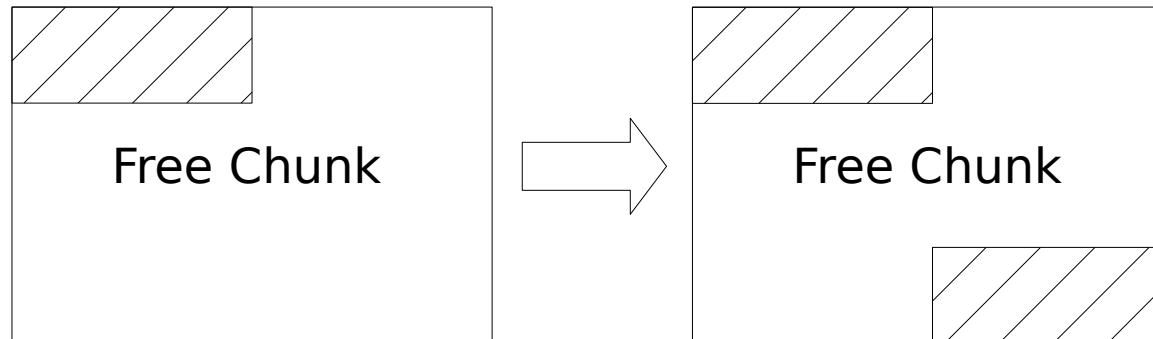
Splitting Schema

KNOWING YOU'RE SECURE

Free chunk at the beginning of a page,
allocated chunk goes at the front



Otherwise, allocated chunk goes at the end





Free Summary

KNOWING YOU'RE SECURE

- Free algorithm works pretty much as expected:
 - It will use Lookaside lists for chunks up to 256 bytes
 - If they are not full already
 - It will use ListHeads for chunks up to 4080 bytes
- Merges contiguous free chunks to lower fragmentation
- Releases pages if necessary

Simplified Free Algorithm

nt!ExFreePoolWithTag (1/2)

KNOWING YOUR SYSTEM

- If P is page aligned:
 - Call nt!MiFreePoolPages
- If BlockSize ≤ 0x20:
 - If PoolType = PagedPool:
 - Put in “per processor” paged lookaside list
 - If failed, put in “system wide” paged lookaside list
 - Return on success
 - Else:
 - Put in “per processor” non-paged lookaside list
 - If failed, put in “system wide” non-paged lookaside list
 - Return on success

Simplified Free Algorithm

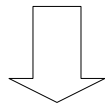
nt!ExFreePoolWithTag (2/2)

KNOWING YOUR INSIDE

- If next chunk is free and not page aligned:
 - Merge with current chunk
- If previous chunk is free:
 - Merge with current chunk
- If resulting chunk is a full page:
 - Call nt!MiFreePoolPages
- Else:
 - Add chunk to the tail of the correct ListHeads
 - Based on PoolType, PoolIndex and BlockSize of chunk



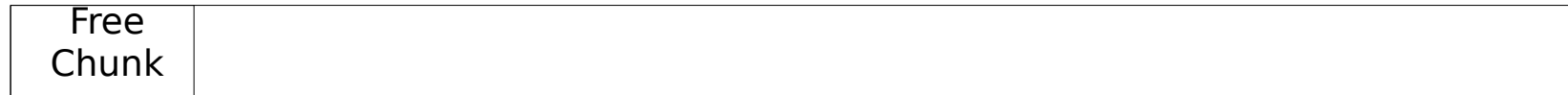
Chunk being freed



Merge #1



Merge #2





Exploiting a Kernel Pool overflow



Pool BugChecks

KNOWING YOU'RE SECURE

- Discrepancies in the kernel pool will most likely result in a BugCheck (Blue Screen)

0x19: BAD_POOL_HEADER

0x41: MUST_SUCCEED_POOL_EMPTY

0xc1: SPECIAL_POOL_DETECTED_MEMORY_CORRUPTION

0xc2: BAD_POOL_CALLER

- Some are only present in Checked Build
- Avoid those when exploiting a pool overflow!

BugCheck Example

KNOWING YOU'RE SECURE

```
Local host - VMware Server Console
File Edit View Host VM Power Snapshot Windows Help
Inventory
winXPProSP2EnCB
winXPProSP2En
winXPHomSP2Fr
winVistaUltSP0En
winXPMCE2005En
win2008EntSP0En
Home winXPMCE2005En
A problem has been detected and windows has been shut down to prevent damage
to your computer.
BAD_POOL_CALLER
If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:
Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any windows updates you might need.
If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup options, and then
select Safe Mode.
Technical information:
*** STOP: 0x000000C2 (0x00000007, 0x000000cd4, 0x41414141, 0x81800008)
Beginning dump of physical memory
Dumping physical memory to disk: 45
VMware Server 1.0.5
```

Some BugCheck Conditions

KNOWING YOU'RE SECURE

- **nt!ExFreePoolWithTag:**
 - BugCheck 0xc2, 7 if *PoolType*&4=0
 - The chunk attempted to be freed is already free
 - BugCheck 0x19, 0x20 if *PreviousSize* of next chunk is \neq *BlockSize* of current chunk
- **Checked Build:**
 - BugCheck 0x19, 3 if $(\text{Entry} \rightarrow \text{Flink}) \rightarrow \text{Blink} \neq \text{Entry}$ or $(\text{Entry} \rightarrow \text{Blink}) \rightarrow \text{Flink} \neq \text{Entry}$
 - It didn't make it to retail build, thanks Microsoft!

Exploitable Overflows?

KNOWING YOU'RE SECURE

Yes!

- Unlike in userland heaps, there is no such thing as a kernel pool *cookie*
- There is no safe unlinking in retail build

Kernel Pool Unlink

KNOWING YOU'RE SECURE

- Removing an entry 'e' from a double linked list:

```
PLI ST_ENTRY b, f ;
```

```
f = e → Flink ;
```

```
b = e → BLink ;
```

```
b → Flink = f ;
```

```
f → BLink = b ;
```

- This leads to a usual write4 primitive:

```
*(where) = what
```

```
*(what + 4) = where
```

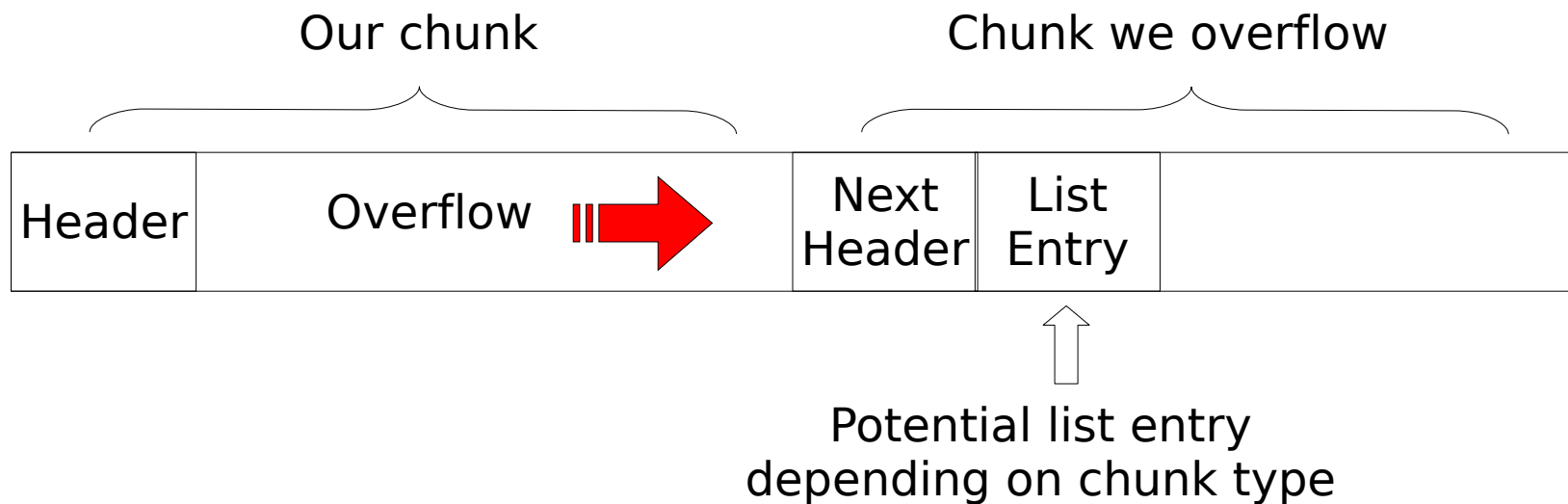
- Locating the unlinks:

- nt!ExFreePoolWithTag: when merging chunks

- nt!MiAllocatePoolPages, nt!MiFreePoolPages, ...



Kernel Pool Overflow



Different Write4

KNOWING YOU'RE SECURE

- Summary list of write4 techniques:
 - Write4 on Merge with Next
 - When freeing our chunk
 - Write4 on Merge with Previous
 - When freeing the chunk we overflowed
 - Write4 on *ListHeads* Unlink
 - If we overflowed an entry in a *ListHeads* list
 - Write4 on MmNonPagedPoolFreeListHead Unlink
 - If we overflowed an entry in a MmNonPagedPoolFreeListHead list

Write4 on Merge with Next

KNOWING YOU'RE SECURE

Case #1

- When our chunk is freed:
 - If *PreviousSize* of next chunk is = *BlockSize* of current chunk
 - To avoid BugCheck 0x19
 - If *BlockSize* > 0x20 (or lookaside lists are full)
 - To avoid a free to lookaside
 - If *PoolType* of next chunk is 0
 - If *BlockSize* of next chunk is > 1
 - Otherwise it means there is no list entry
 - Then merge with next chunk:
 - And **unlink** happens on list entry of next chunk

Exploit Case #1

KNOWING YOU'RE SECURE

- Allocate a chunk of size 256-4080 bytes
 - Works with smaller chunks if lookaside lists are full
- Craft a free header after our chunk with:
 - Correct *PreviousSize*
 - *PoolType=0*
 - Wanted *Flink* and *Blink*
 - Requires a minimum overflow of about 16 bytes
- Write4 happens when our allocated chunk is freed

Write4 on Merge with Previous

KNOWING YOU'RE SECURE

Case #2

- When the chunk we overflowed is freed:
 - If *PreviousSize* of next chunk is = *BlockSize* of current chunk
 - To avoid BugCheck 0x19
 - If *BlockSize* > 0x20 (or lookaside lists are full)
 - To avoid a free to lookaside
 - If *PoolType* of previous chunk is 0
 - If *BlockSize* of previous chunk is >1
 - Otherwise it means there is no list entry
 - Then merge with previous chunk:
 - And **unlink** happens on list entry of previous chunk

Exploit Case #2 (1/2)

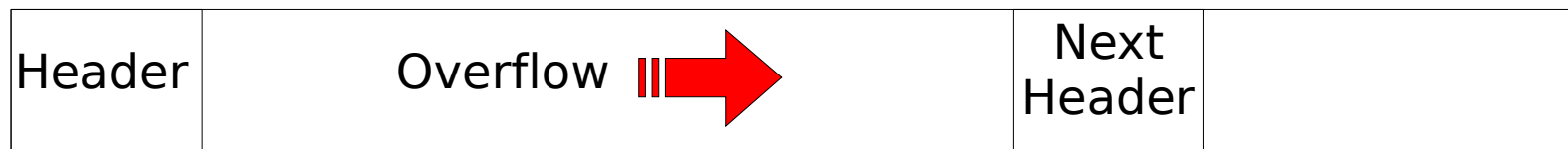
KNOWING YOU'RE SECURE

- Allocate a chunk
- Next chunk **must** be allocated and of size 256-4080 bytes (not as easy as it looks)
 - Works with smaller chunks if lookaside lists are full
- Craft a fake free header and list entry at the end of our chunk (with realistic sizes)
- Overflow *PreviousSize* of next chunk to make it point to our fake header
 - Requires a ~1 byte overflow!
 - 0x00 can work if *PreviousSize* > 0x100

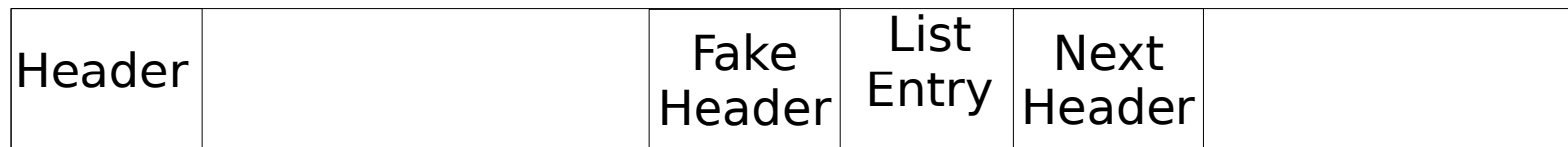
Exploit Case #2 (2/2)

KNOWING YOU'RE SECURE

- A bigger overflow would require knowledge of *BlockSize* of next chunk to avoid BugCheck
 - Or enough bytes to craft another header after (~257)
- Write4 happens when next chunk is freed
 - Also works with page aligned chunks!



Overflow PreviousSize of next chunk



Merge

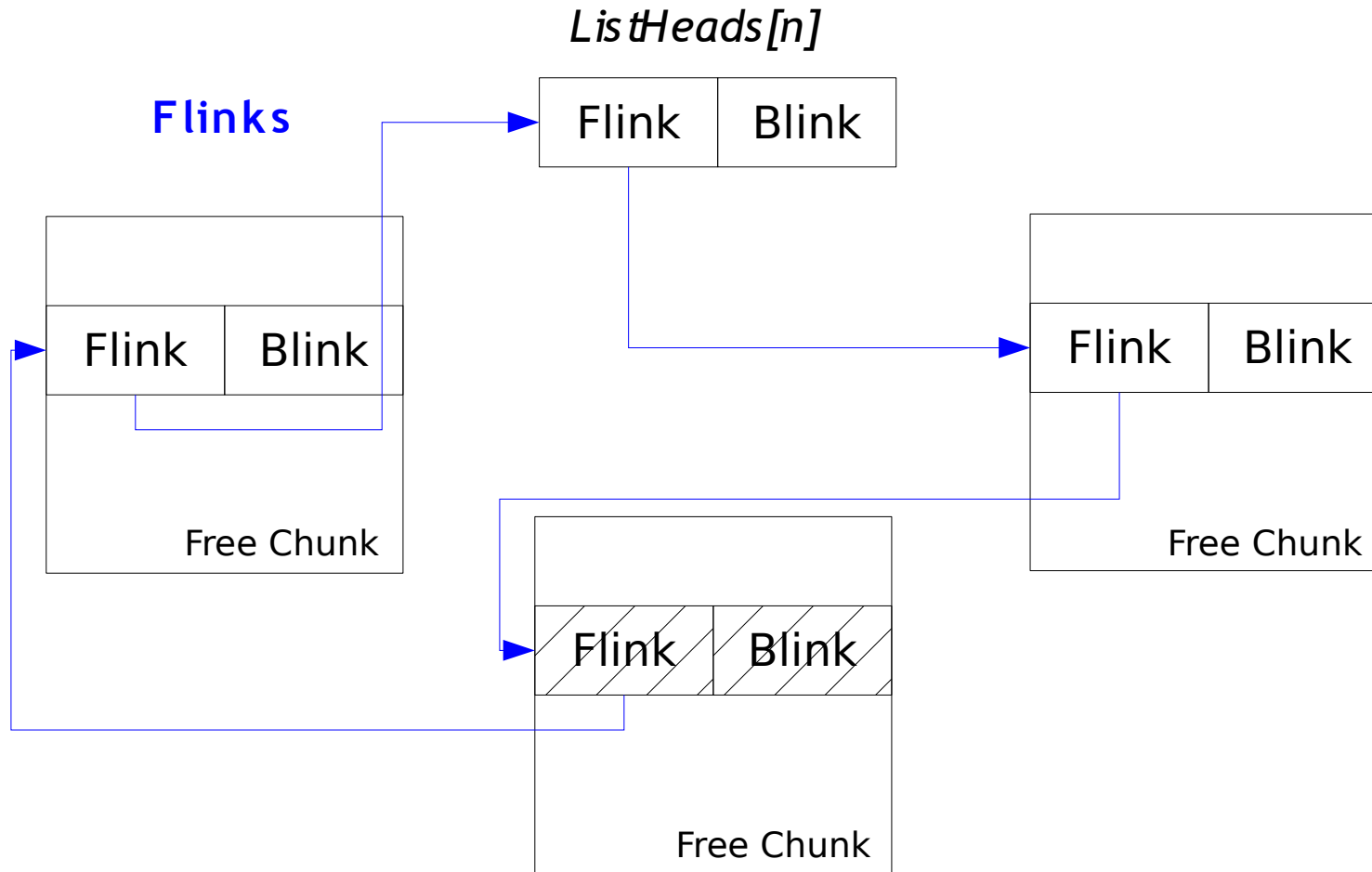
ListHeads Write4

KNOWING YOU'RE SECURE

- When the chunk we overflowed is allocated:
 - If the chunk was requested through ListHeads list
 - No other constraint on *BlockSize*, *PreviousSize*, ...
 - Then the overflowed list entry is removed from the ListHeads list ↗ Write4
 - **Variations** of the write4 exist based on operations done on the neighboring entries
 - We can end up overwriting ListHeads[*BlockSize*] → *Flink* with a pointer we control
 - Next time a chunk of *BlockSize* is requested, our pointer will be returned

ListHeads Illustrated (1/3)

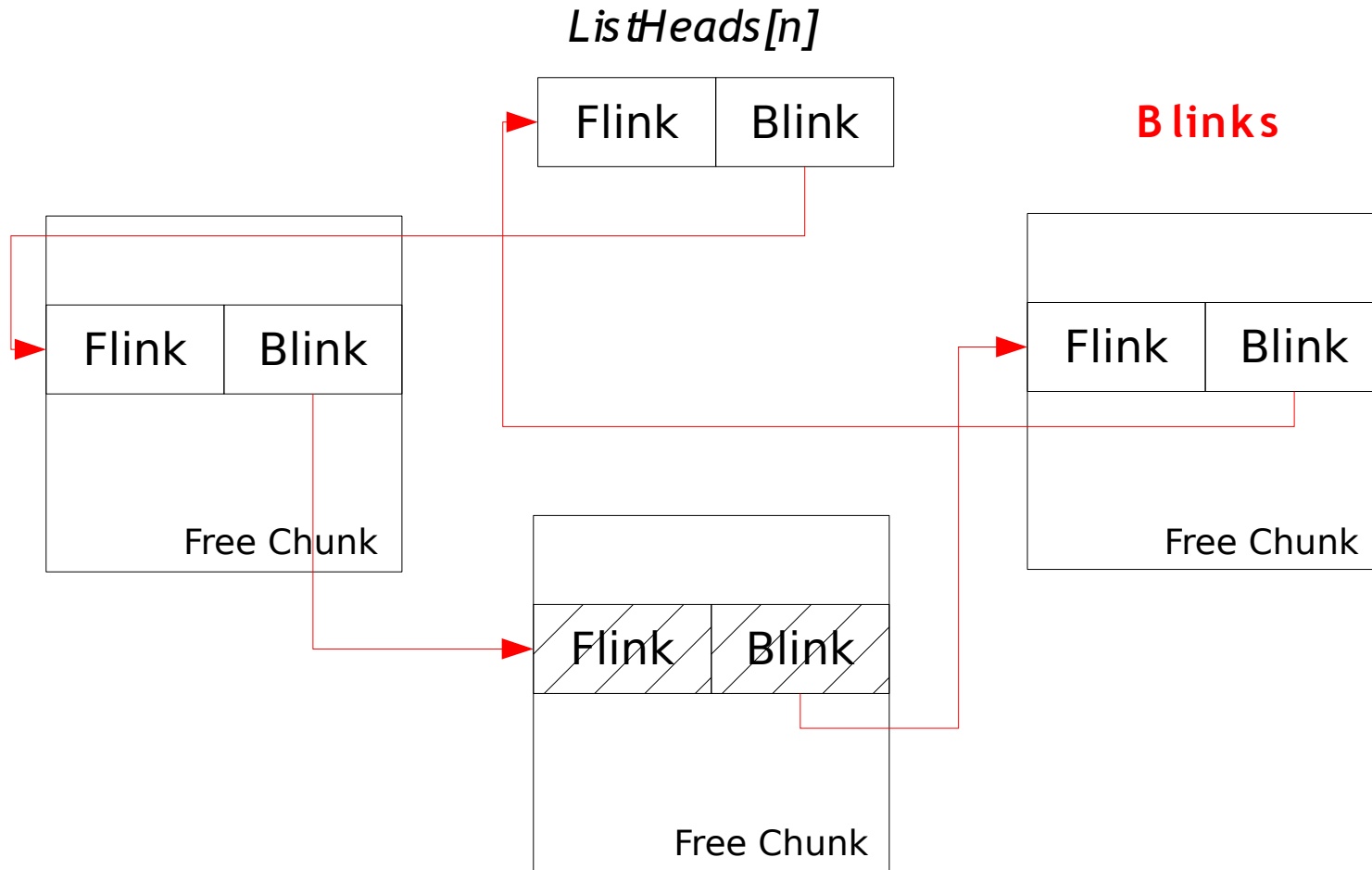
KNOWING YOU'RE SECURE



 : Overflowed list entry

ListHeads Illustrated (1/3)

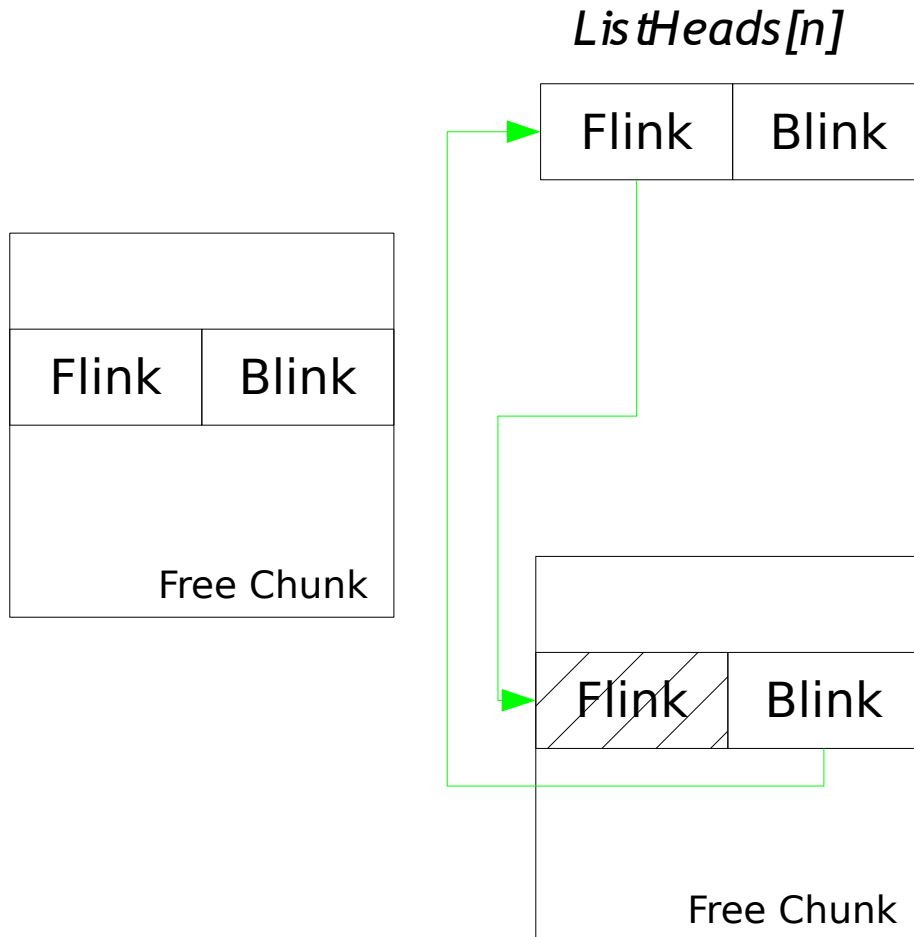
KNOWING YOU'RE SECURE



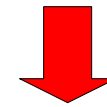
 : Overflowed list entry

ListHeads Illustrated (2/3)

KNOWING YOU'RE SECURE



Allocation of size n
unlinks $ListHeads[n] \rightarrow Flink$



```
PLIST_ENTRY b,f;  
f=ListHeads[n]→Flink→Flink;  
b=ListHeads[n]→Flink→Blink;  
b→Flink=f;  
f→Blink=b;
```

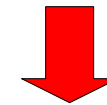
ListHeads Illustrated (3/3)

KNOWING YOU'RE SECURE

ListHeads[n]



Allocation of size n
unlinks ListHeads[n]→Flink



```

PLIST_ENTRY b,f;
f=ListHeads[n]→Flink→Flink;
b=ListHeads[n]→Flink→Blink;
b→Flink=f;
f→Blink=b;   ↩ might AV
    
```

ListHeads[n]→Flink is now under our control!

MMFREE_POOL_ENTRY

Write4

KNOWING YOU'RE SECURE

- When the chunk we overflowed is allocated:
 - If the chunk was requested through nt! MmNonPagedPoolFreeListHead list
 - Overflowed chunk is obviously page aligned
 - Then the write4 happens when the MMFREE_POOL_ENTRY structure is removed from the double linked list ↗ Write4
 - **Variations** of the write4 exist based on operations done on the neighboring list entries

What? Where?

KNOWING YOU'RE SECURE

- Most complicated with Kernel Pool overflows:
 - Only a few pools for **all** kernel allocations
 - Lot of allocation and free
 - 4 different kernels based on architecture:
 - Single processor: *ntoskrnl.exe*, *ntkrnlpa.exe*
 - Multi processors: *ntkrnlmp.exe*, *ntkrpamp.exe*
 - A lot of kernel patches for every service pack
 - ↳ Addresses tend to change, a lot
- Any access violation will most likely end up in a BugCheck ☹

Some Ideas

Non Exhaustive

KNOWING YOU'RE SECURE

- `nt!KiDebugRoutine` function pointer
 - Called by `nt!KiDispatchException` if not NULL
 - Should be kept as last resort
- Context specific function pointers
 - `tcpip!tcpxsum_routine` is called right after the overflow in the case of MS08-001
- Function pointers arrays
 - `nt!HalDispatchTable`
 - Interrupt Dispatch Table (IDT)
- Kernel instructions – page is RWE!

Write4 into the Kernel

KNOWING YOU'RE SECURE

- Before

```
mov eax, [edx+8]
mov ebx, [edx+0Ch]
mov [ebx], eax
mov [eax+4], ebx
```

Edx points to something we control

```
loc_80543F0B: ; CODE XREF: ExFreePoolWithTag(x,x)+518j
movzx edx, word ptr [edx+2]
```

- After

```
mov eax, [edx+8]
mov ebx, [edx+0Ch]
mov [ebx], eax
mov [eax+4], ebx
```

Jmp edx being 2 bytes long, we can pick the upper 2 so that the write4 doesn't trigger an access violation

```
loc_80543F0B: ; CODE XREF: ExFreePoolWithTag(x,x)+518j
jmp edx
```


Fixing the Kernel Pool

KNOWING YOU'RE SECURE

- Check for inconsistencies and fix them:
 - Lookaside lists, both “per processor” and systemwide
 - Zero out *Sequence*, *Depth* and *Next* of *ListHead* member of the given `nt!_GENERAL_LOOKASIDE` entry
 - In fact the first 8 bytes
 - ListHeads lists for involved pool descriptor(s)
 - Set `ListHeads[BlockSize]→Flink` and `ListHeads[BlockSize]→Blink` to `&ListHeads[BlockSize]`
 - `nt!MmNonPagedPoolFreeListHead` array



MS08-001: IGMPv3 Kernel Pool overflow



History

KNOWING YOU'RE SECURE

- Remote default kernel pool overflow in Windows XP when parsing IGMPv3 packets
 - Even bypasses default firewall rules!
- Released January, 8th 2008 in [MS08-001](#)
 - Along with MLDv2 vulnerability for Vista
- Reported by Alex Wheeler and Ryan Smith of [IBM – Internet Security Systems](#)
- Considered by Microsoft [SWI](#) as “unlikely” exploitable



1. Walk a single linked list to count the number of entries (using a 32 bit counter)

2. Allocate some memory:

```
loc_44197:    ; CODE XREF: Get GSI sInRecord(x, x)+18j
push 10h    ; Priority
movzx eax, dx
push 'qICT' ; Tag
lea eax, ds:8[eax*4]
push eax   ; Number Of Bytes
push ebx   ; Pool Type
call ds:__imp__ExAllocatePoolWithTagPriority@16
```

❖ Failed!

3. Copy the list entries in the allocated array by walking the list ➡ Overflow



- Send multiple IGMPv3 membership queries
 - Group address has to be valid
 - Can be sent to:
 - Unicast IP address (ie. 10.10.10.42)
 - Broadcast IP address (ie. 10.10.10.255)
 - Multicast IP address (ie. 224.0.0.1)
 - Total number of **unique** source addresses must be 65536 or greater
 - IP addresses in the 224-239 range are ignored
- Wait for the IGMPv3 report to be triggered



Issues (1/2)

KNOWING YOU'RE SECURE

- Sending a lot of IGMPv3 membership queries induces *high CPU utilization*
 - Each new source address triggers the allocation of a **20 (0x14) byte** structure
 - The linked list of structures is walked before adding a new element to check for uniqueness of IP ($O(n^2)$)
- High CPU usage leads to potential packets dropping
 - Final buffer will not be as expected

Issues (2/2)

KNOWING YOU'RE SECURE

- IGMPv3 reports are on a *random* timer
 - Can be triggered before all the queries are sent
 - Buffer will not be filled as intended
- 16 bit wrap *usually* means **huge** overflow
 - 65536 entries are put in a 0 byte buffer



Solutions

KNOWING YOU'RE SECURE

- Exponential delay between packets sending
 - Still fast enough to avoid a random report trigger
- Report trigger can be forced if $QQIC=0$
 - Empty the list before the attack
 - Trigger the overflow when all the packets are sent
- Index for the copy is on 16 bit register
 - Overflow size is actually (for $n > 0xffff$):
$$(0x10000 - (n \% 0x10000)) * 4$$
 - For $n = 0x1fffe$, allocation size is **0x40000** (page aligned allocation), overflow size is **8**

Dirty Way

KNOWING YOU'RE SECURE

$256 < (n \% 0x10000) * 4 + 8 < 4080$

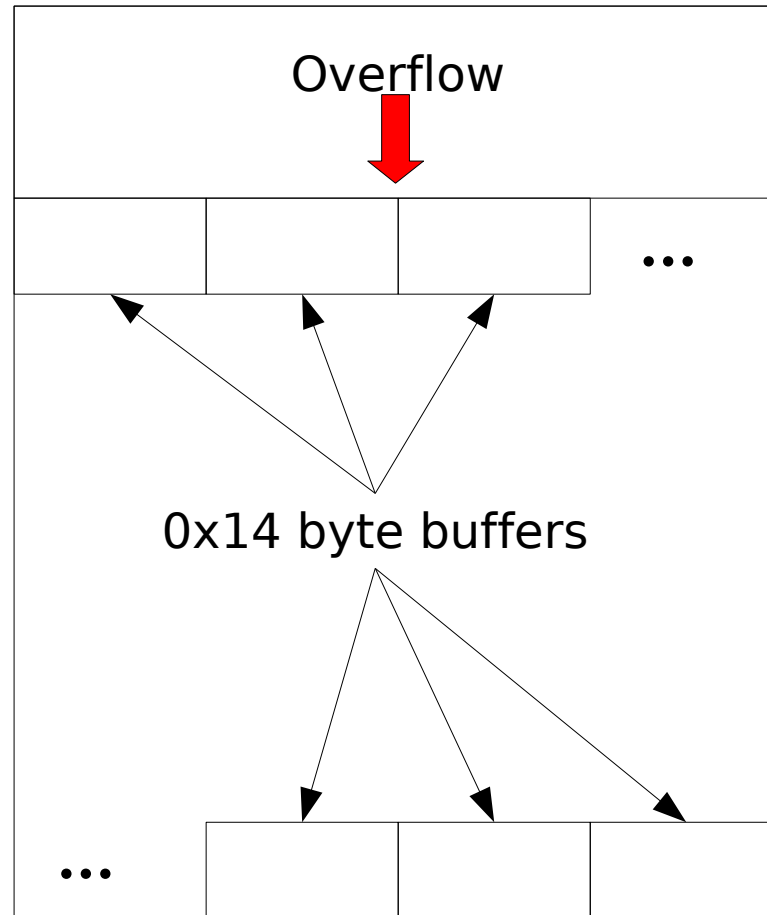
- Pros
 - Relatively small number of packets (~180)
 - Since the allocated buffer is freed after the overflow, “Write4 on Merge with Next” will always happen
 - Next chunk is easily craftable
 - We control *BlockSize* to some extent, which is interesting for the “Write4 into Kernel” technique
- Cons
 - Overflow is huge! (>0x3f000 bytes)
 - A large chunk of the kernel pool will be trashed ☹
 - Lot of pool fixing to do

Why would it work?

KNOWING YOU'RE SECURE

Kernel Pool is filled with n 0x14 byte buffers

Our buffer will be allocated before those (we pretty much exhausted all the free chunks)



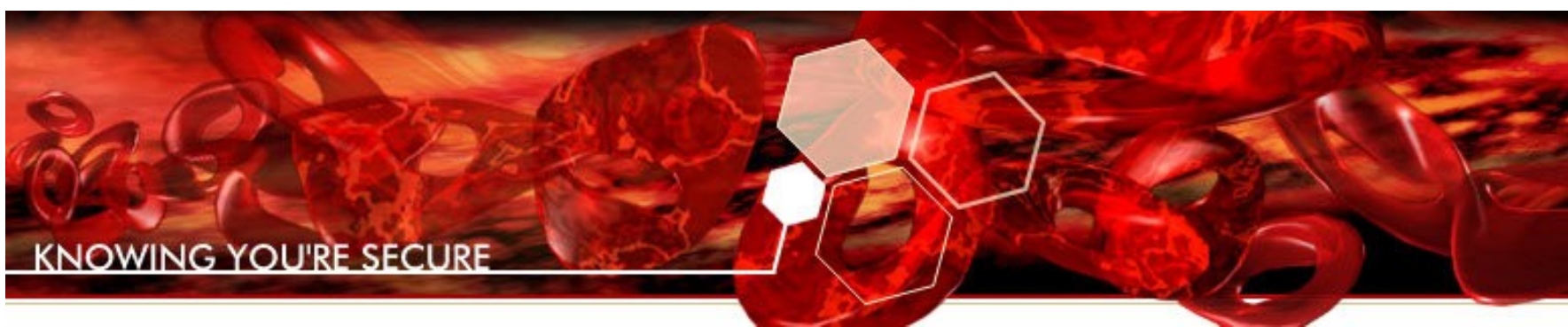
Buffer closest to our allocated buffer is the 1st one to be copied and freed

Requires a “carpet” of ~13000 contiguous 0x14 byte buffers (not too hard)

Clean Way

KNOWING YOU'RE SECURE $(0x10000 - (n \% 0x10000)) * 4 = 8$

- Pros
 - 8 byte overflow only
 - The kernel pool remains pretty clean
- Cons
 - Next chunk can be:
 - A nt!_MMFREE_POOL_ENTRY: we overflow a list entry
 - A busy nt!_POOL_HEADER
 - “Write4 on Merge with Previous” will happen on free
 - Etc
 - ⇒ Headaches
 - Higher number of packets (~300)



Conclusion



CONCLUSION

KNOWING YOU'RE SECURE

- Kernel still offers a nice playground for exploitation even in latest Windows versions
- Exploitation costs have increased dramatically
 - We're still working on MSo8-001!



- More memory allocations are NUMA aware:
 - Different pool descriptors per NUMA node
 - Up to 16 non-paged pool descriptors
 - Initial non-paged pool has separate address ranges for each node
 - Up to 16 paged pool descriptors
- Only kernel in XP SP2 modifying nt!
KeNumberNodes is *ntkrpamp.exe* (PAE & MP)

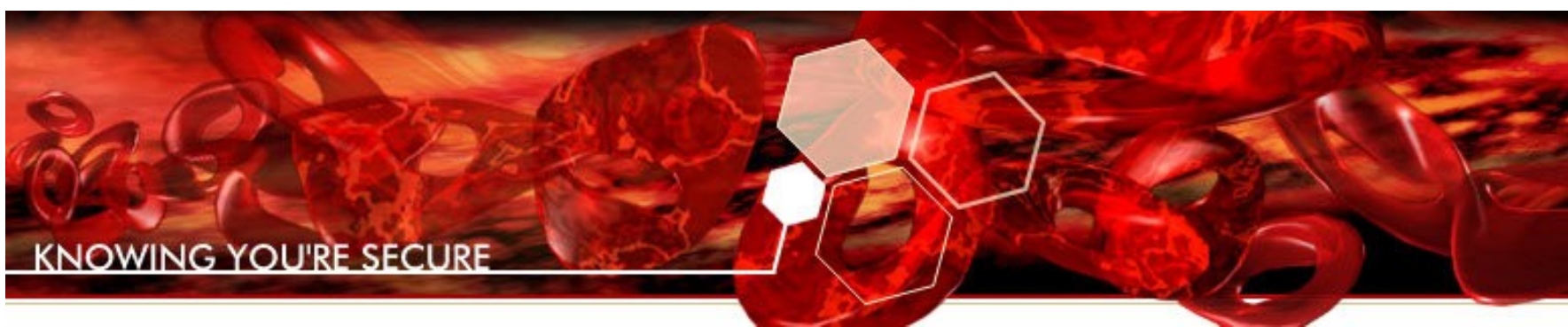
⇒ Kernel pool overflows more complex



Literature

KNOWING YOU'RE SECURE

- **How to exploit Windows kernel memory pool**
 - SoBeIt, Xcon 2005
- **Reliable Windows Heap Exploits**
 - M. Conover, O. Horovitz, CanSecWest 2004
- **Owning NonPaged pool using stealth hooking**
 - mxatone, Phrack 65



Thank you!
Questions?