# Modern Kernel Pool Exploitation: Attacks and Techniques

Tarjei Mandt | Infiltrate 2011

# About Me

- Security Researcher at Norman
  - Malware Detection Team (MDT)
  - Focused on exploit detection / mitigation
- Interests
  - Vulnerability research
  - Operating systems internals
  - Low-level stuff
- Found some kernel bugs recently
  - MS10-073, MS10-098, MS11-012, ...
  - Some in MS11-034

# Agenda

- Introduction
- Kernel Pool Internals
- Kernel Pool Attacks
- Case Study / Demo
- Kernel Pool Hardening
- Conclusion

# Introduction

Modern Kernel Pool Exploitation:
Attacks and Techniques

# Introduction

- Exploit mitigations such as DEP and ASLR do not prevent exploitation in every case
  - JIT spraying, memory leaks, etc.
- Privilege isolation is becoming an important component in confining application vulnerabilities
  - Browsers and office applications employ “sandboxed” render processes
  - Relies on (security) features of the operating system
- In turn, this has motivated attackers to focus their efforts on privilege escalation attacks
  - Arbitrary ring0 code execution → OS security undermined

# The Kernel Pool

- Resource for dynamically allocating memory
- Shared between all kernel modules and drivers
- Analogous to the user-mode heap
  - Each pool is defined by its own structure
  - Maintains lists of free pool chunks
- Highly optimized for performance
  - No kernel pool cookie or pool header obfuscation
- The kernel executive exports dedicated functions for handling pool memory
  - **ExAllocatePool*** and **ExFreePool*** (discussed later)

# Kernel Pool Exploitation

- An attacker's ability to leverage pool corruption vulnerabilities to execute arbitrary code in ring 0
  - Similar to traditional heap exploitation
- Kernel pool exploitation requires careful modification of kernel pool structures
  - Access violations are likely to end up with a bug check (BSOD)
- Up until Windows 7, kernel pool overflows could be generically exploited using write-4 techniques
  - SoBeIt[2005]
  - Kortchinsky[2008]

# Previous Work

- Primarily focused on XP/2003 platforms
- How To Exploit Windows Kernel Memory Pool
  - Presented by SoBeIt at XCON 2005
  - Proposed two write-4 exploit methods for overflows
- Real World Kernel Pool Exploitation
  - Presented by Kostya Kortchinsky at SyScan 2008
  - Discussed four write-4 exploitation techniques
  - Demonstrated practical exploitation of MS08-001
- All the above exploitation techniques were addressed in Windows 7 (Beck[2009])

# Contributions

- Elaborate on the internal structures and changes made to the Windows 7 (and Vista) kernel pool
- Identify weaknesses in the Windows 7 kernel pool and show how an attacker may leverage these to exploit pool corruption vulnerabilities
- Propose ways to thwart the discussed attacks and further harden the kernel pool

# Kernel Pool Internals

Modern Kernel Pool Exploitation:
Attacks and Techniques

# Kernel Pool Fundamentals

- Kernel pools are divided into types
  - Defined in the **POOL_TYPE** enum
  - Non-Paged Pools, Paged Pools, Session Pools, etc.
- Each kernel pool is defined by a *pool descriptor*
  - Defined by the **POOL_DESCRIPTOR** structure
  - Tracks the number of allocs/frees, pages in use, etc.
  - Maintains lists of free pool chunks
- The initial descriptors for paged and non-paged pools are defined in the **nt!PoolVector** array
  - Each index points to an array of one or more descriptors

# Kernel Pool Descriptor (Win7 RTM x86)

- kd> **dt nt!_POOL_DESCRIPTOR**
  - +0x000 PoolType : _POOL_TYPE
  - +0x004 PagedLock : _KGUARDED_MUTEX
  - +0x004 NonPagedLock : Uint4B
  - +0x040 RunningAllocs : Int4B
  - +0x044 RunningDeAllocs : Int4B
  - +0x048 TotalBigPages : Int4B
  - +0x04c ThreadsProcessingDeferrals : Int4B
  - +0x050 TotalBytes : Uint4B
  - +0x080 PoolIndex : Uint4B
  - +0x0c0 TotalPages : Int4B
  - +0x100 PendingFrees : Ptr32 Ptr32 Void
  - +0x104 PendingFreeDepth: Int4B
  - +0x140 ListHeads : [512] _LIST_ENTRY

# Non-Paged Pool

- Non-pagable system memory
  - Guaranteed to reside in physical memory at all times
- Number of pools stored in **nt!ExpNumberOfNonPagedPools**
- On uniprocessor systems, the first index of the **nt!PoolVector** array points to the non-paged pool descriptor
  - kd> **dt nt!_POOL_DESCRIPTOR poi(nt!PoolVector)**
- On multiprocessor systems, each node has its own non-paged pool descriptor
  - Pointers stored in **nt!ExpNonPagedPoolDescriptor** array

# Paged Pool

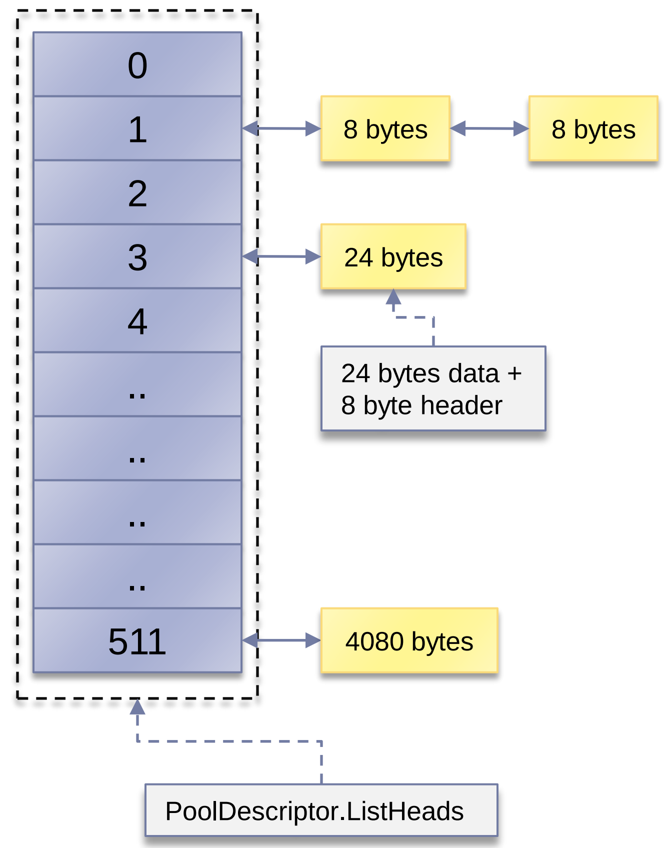
- Pageable system memory
  - Can only be accessed at IRQL < DPC/Dispatch level
- Number of paged pools defined by **nt!ExpNumberOfPagedPools**
- On uniprocessor systems, four (4) paged pool descriptors are defined
  - Index 1 through 4 in **nt!ExpPagedPoolDescriptor**
- On multiprocessor systems, one (1) paged pool descriptor is defined per node
- One additional paged pool descriptor is defined for prototype pools / full page allocations
  - Index 0 in **nt!ExpPagedPoolDescriptor**

# Session Paged Pool

- Pageable system memory for session space
  - E.g. Unique to each logged in user
- Initialized in **nt!MiInitializeSessionPool**
- On Vista, the pool descriptor pointer is stored in **nt!ExpSessionPoolDescriptor** (session space)
- On Windows 7, a pointer to the pool descriptor from the current thread is used
  - KTHREAD->Process->Session.PagedPool
- Non-paged session allocations use the global non-paged pools

# Pool Descriptor Free Lists (x86)

- Each pool descriptor has a *ListHeads* array of 512 doubly-linked lists of free chunks of the same size
  - 8 byte granularity
  - Used for allocations up to 4080 bytes
- Free chunks are indexed into the ListHeads array by block size
  - BlockSize: (NumBytes+0xF) >> 3
- Each pool chunk is preceded by an 8-byte pool header



# Kernel Pool Header (x86)

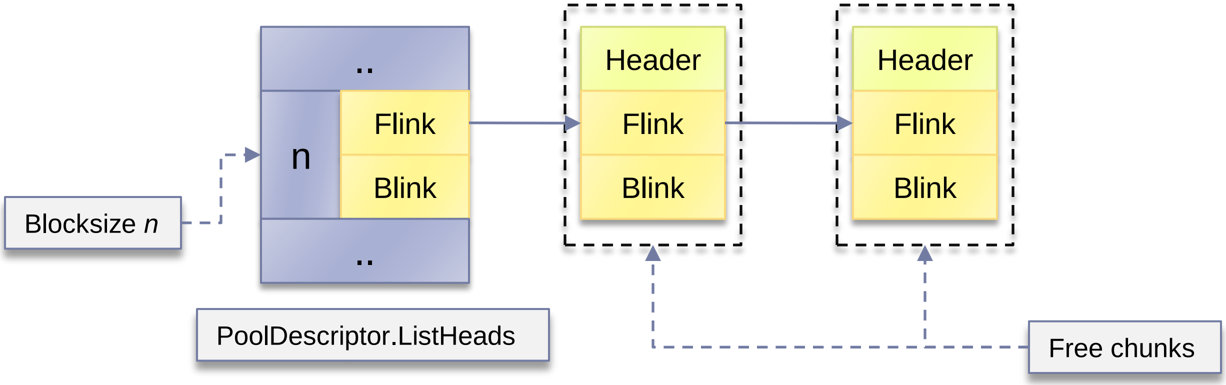
- kd> **dt nt!_POOL_HEADER**
  - +0x000 PreviousSize : Pos 0, 9 Bits
  - +0x000 PoolIndex : Pos 9, 7 Bits
  - +0x002 BlockSize : Pos 0, 9 Bits
  - +0x002 PoolType : Pos 9, 7 Bits
  - +0x004 PoolTag : Uint4B
- *PreviousSize*: BlockSize of the preceding chunk
- *PoolIndex*: Index into the associated pool descriptor array
- *BlockSize*: (NumberOfBytes+0xF) >> 3
- *PoolType*: Free=0, Allocated=(PoolType|2)
- *PoolTag*: 4 printable characters identifying the code responsible for the allocation

# Kernel Pool Header (x64)

- 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
- *BlockSize*: (NumberOfBytes+0x1F) >> 4
  - 256 ListHeads entries due to 16 byte block size
- *ProcessBilled*: Pointer to process object charged for the pool allocation (used in quota management)

# Free Pool Chunks

- If a pool chunk is freed to a pool descriptor ListHeads list, the header is followed by a **LINK_ENTRY** structure
  - Pointed to by the ListHeads doubly-linked list
  - kd> **dt nt!_LIST_ENTRY**
    +0x000 Flink : Ptr32 _LIST_ENTRY
    +0x004 Blink : Ptr32 _LIST_ENTRY



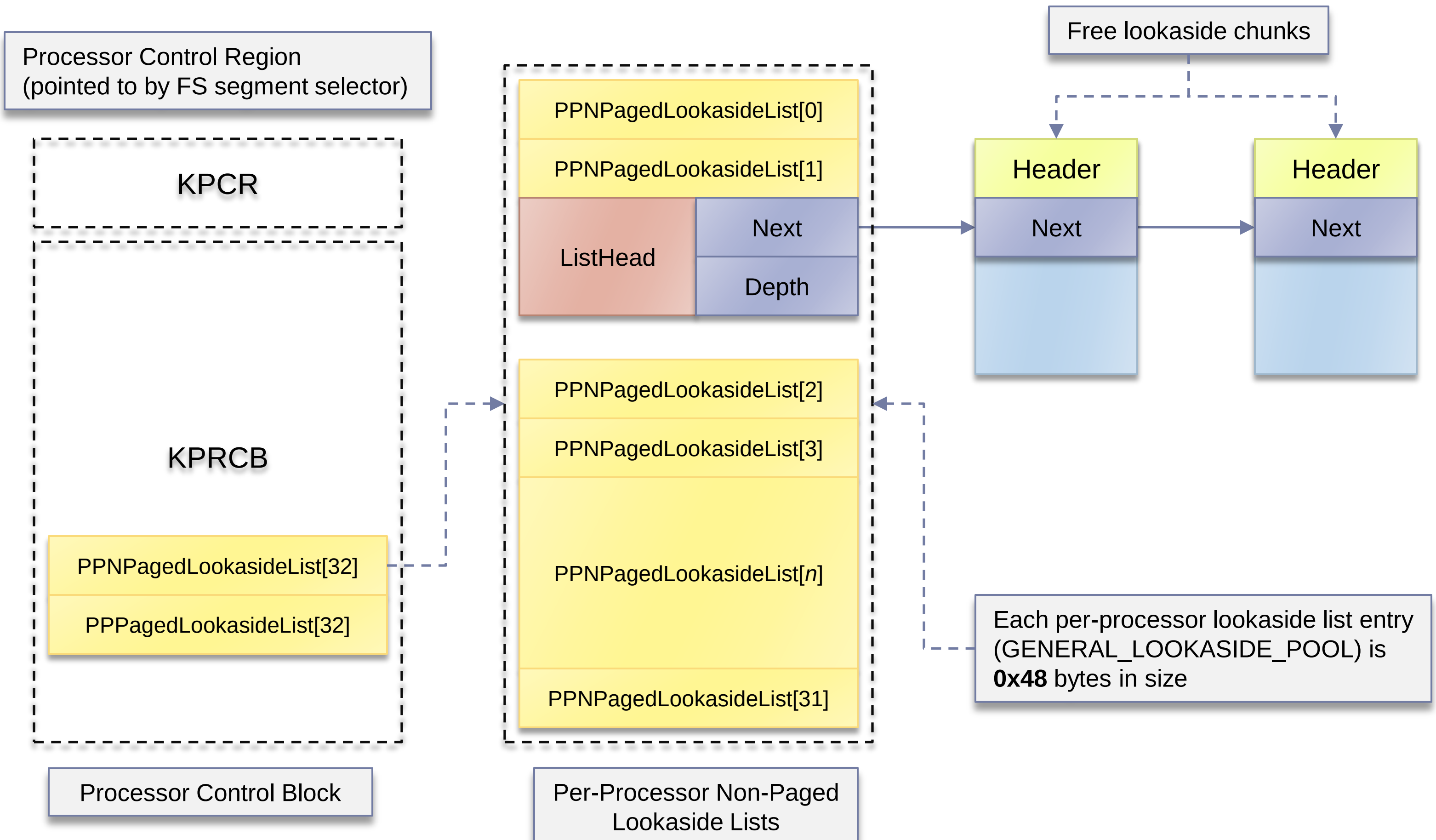
# Lookaside Lists

- Kernel uses *lookaside lists* for faster allocation/deallocation of small pool chunks
  - Singly-linked LIFO lists
  - Optimized for performance – e.g. no checks
- Separate per-processor lookaside lists for pagable and non-pagable allocations
  - Defined in the Processor Control Block (KPRCB)
  - Maximum BlockSize being 0x20 (256 bytes)
  - 8 byte granularity, hence 32 lookaside lists per type
- Each lookaside list is defined by a **GENERAL_LOOKASIDE_POOL** structure

# General Lookaside (Win7 RTM x86)

- kd> **dt _GENERAL_LOOKASIDE_POOL**
  - +0x000 ListHead : _SLIST_HEADER
  - +0x000 SingleListHead : _SINGLE_LIST_ENTRY
  - +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
  - […]

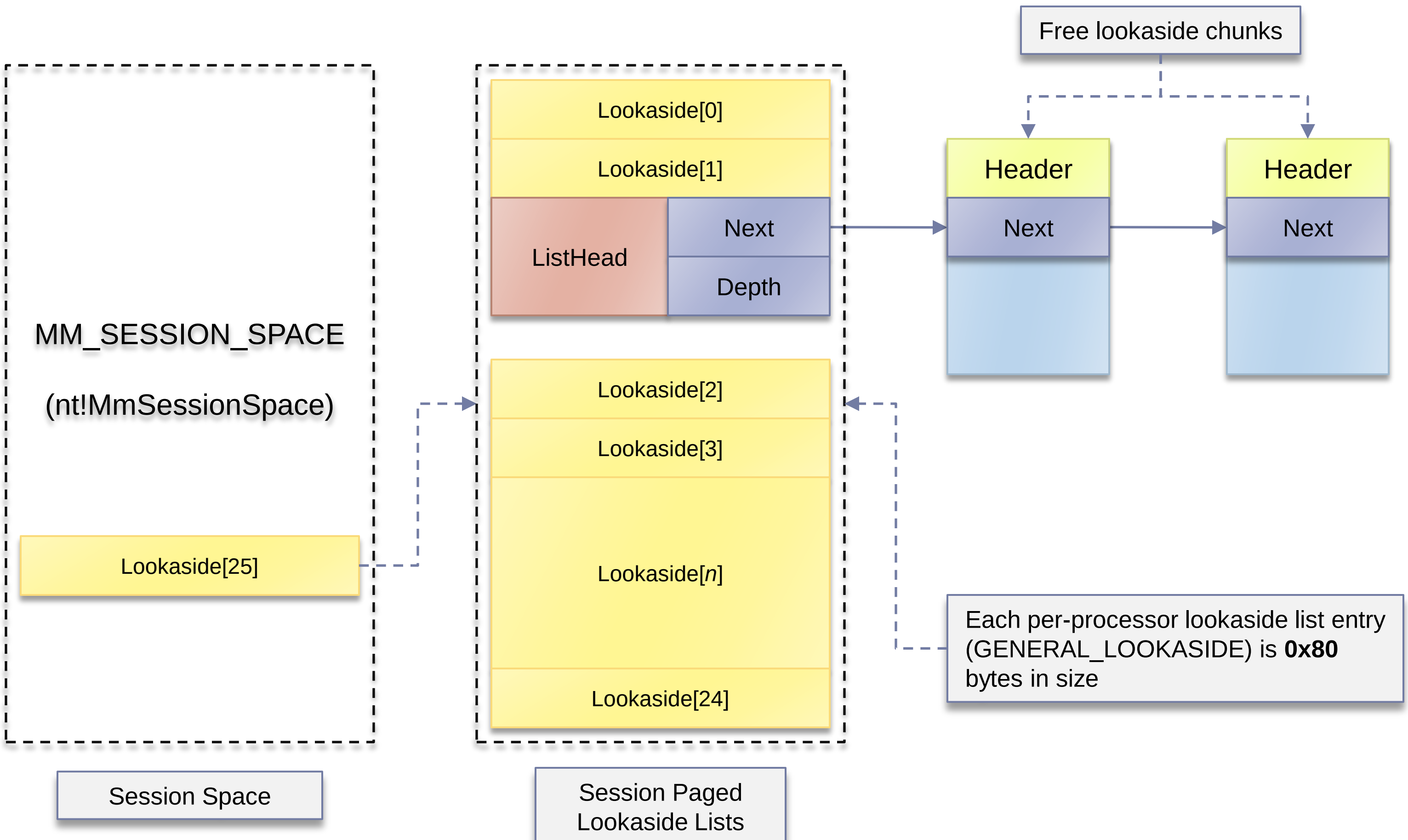
# Lookaside Lists (Per-Processor)



# Lookaside Lists (Session)

- Separate per-session lookaside lists for pagable allocations
  - Defined in session space (**nt!ExpSessionPoolLookaside**)
  - Maximum BlockSize being 0x19 (200 bytes)
  - Uses the same structure (with padding) as per-processor lists
  - All processors use the same session lookaside lists
- Non-paged session allocations use the per-processor non-paged lookaside list
- Lookaside lists are disabled if *hot/cold separation* is used
  - **nt!ExpPoolFlags** & 0x100
  - Used during system boot to increase speed and reduce the memory footprint

# Lookaside Lists (Session)



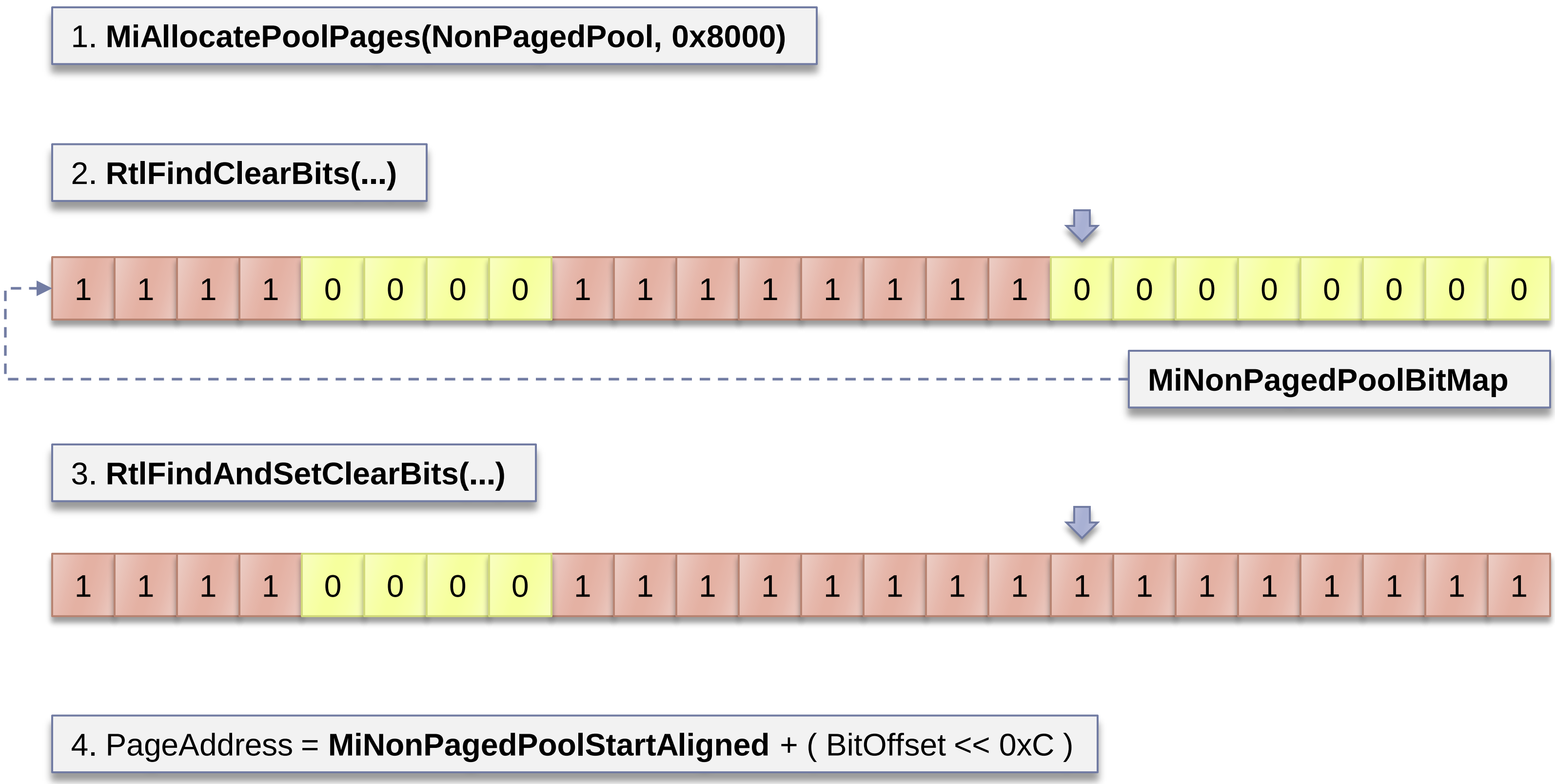
# Large Pool Allocations

- Allocations greater than 0xff0 (4080) bytes
- Handled by the function **nt!ExpAllocateBigPool**
  - Internally calls **nt!MiAllocatePoolPages**
    - Requested size is rounded up to the nearest page size
  - Excess bytes are put back at the end of the appropriate pool descriptor ListHeads list
- Each node (e.g. processor) has 4 singly-linked lookaside lists for big pool allocations
  - 1 paged for allocations of a single page
  - 3 non-paged for allocations of page count 1, 2, and 3
  - Defined in **KNODE** (KPCR.PrcbData.ParentNode)

# Large Pool Allocations

- If lookaside lists cannot be used, an *allocation bitmap* is used to obtain the requested pool pages
  - Array of bits that indicate which memory pages are in use
  - Defined by the **RTL_BITMAP** structure
- The bitmap is searched for the first index that holds the requested number of unused pages
- Bitmaps are defined for every major pool type with its own dedicated memory
  - E.g. **nt!MiNonPagedPoolBitMap**
- The array of bits is located at the beginning of the pool memory range

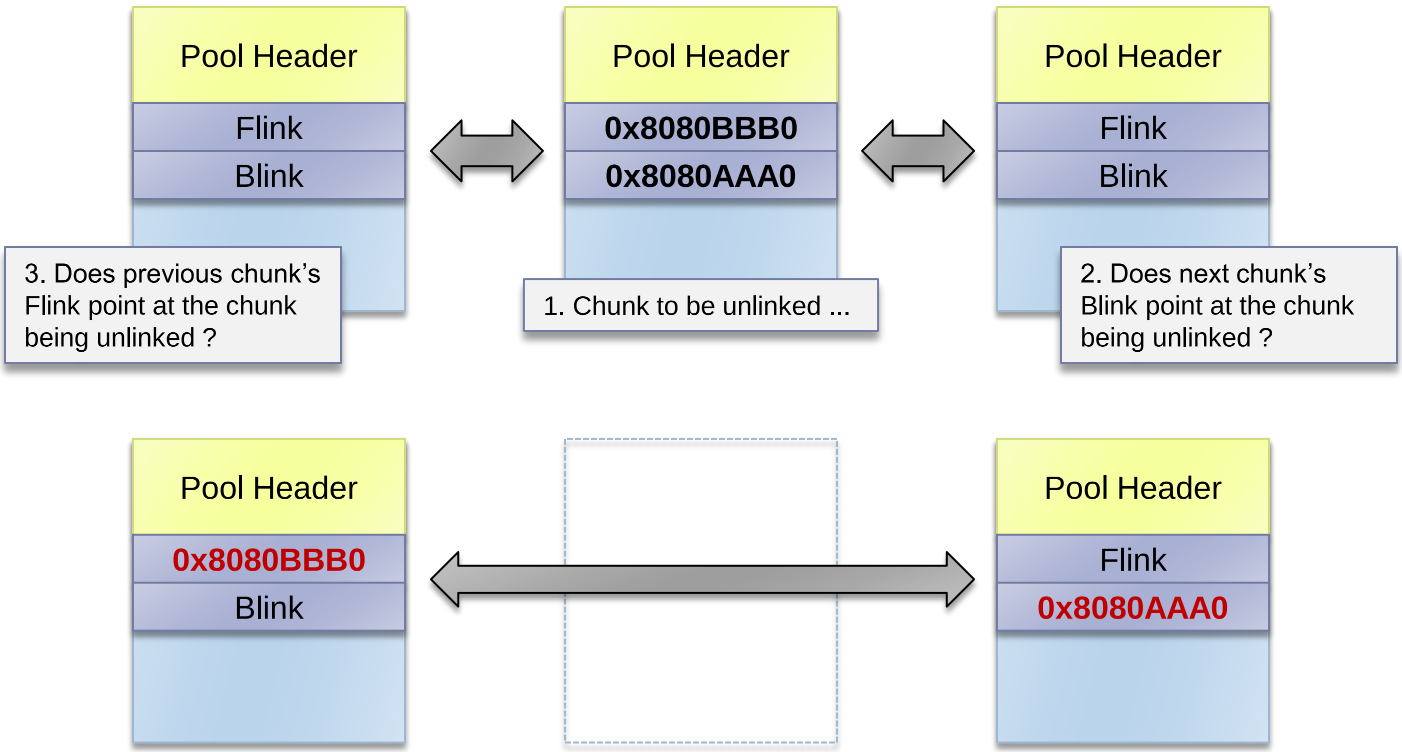
# Bitmap Search (Simplified)



# Allocation Algorithm

- The kernel exports several allocation functions for kernel modules and drivers to use
- All exported kernel pool allocation routines are essentially wrappers for **ExAllocatePoolWithTag**
- The allocation algorithm returns a free chunk by checking with the following (in order)
  - Lookaside list(s)
  - ListHeads list(s)
  - Pool page allocator
- Windows 7 performs *safe unlinking* when pulling a chunk from a free list (Beck[2009])

# Safe Pool Unlinking



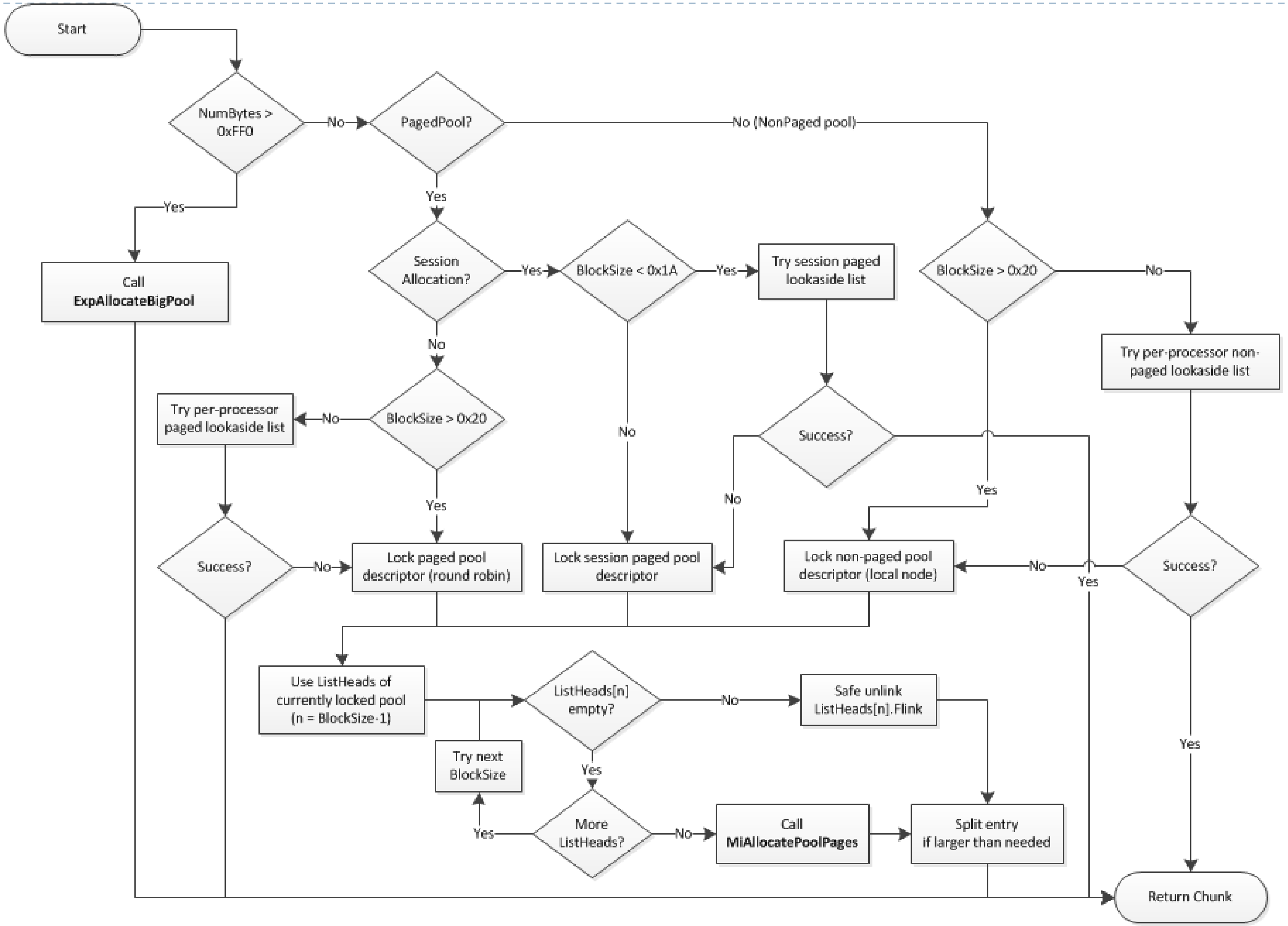
# ExAllocatePoolWithTag (1/2)

- **PVOID ExAllocatePoolWithTag(POOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG Tag)**
- If NumberOfBytes > 0xff0
  - Call **nt!ExpAllocateBigPool**
- If PagedPool requested
  - If (PoolType & SessionPoolMask) and BlockSize <= 0x19
    - Try the session paged lookaside list
    - Return on success
  - Else If BlockSize <= 0x20
    - Try the per-processor paged lookaside list
    - Return on success
  - Lock (session) paged pool descriptor (round robin)

# ExAllocatePoolWithTag (2/2)

- Else (NonPagedPool requested)
  - If BlockSize <= 0x20
    - Try the per-processor non-paged lookaside list
    - Return on success
  - Lock non-paged pool descriptor (local node)
- Use ListHeads of currently locked pool
  - For n in range(BlockSize,512)
    - If ListHeads[n] is empty, try next BlockSize
    - Safe unlink first entry and split if larger than needed
    - Return on success
  - If failed, expand the pool by adding a page
    - Call **nt!MiAllocatePoolPages**
    - Split entry and return on success

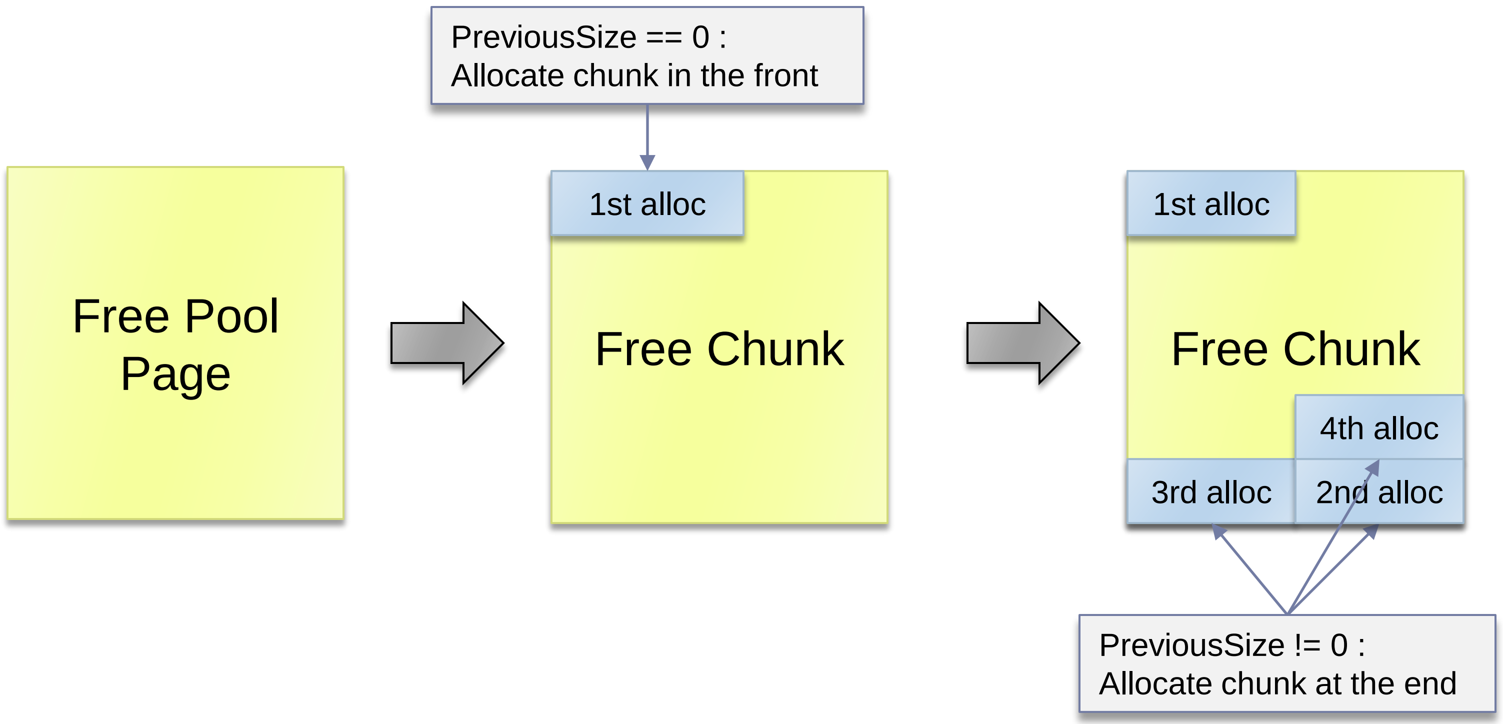
# ExAllocatePoolWithTag



# Splitting Pool Chunks

- If a chunk larger than the size requested is returned from ListHeads[n], the chunk is split
  - If chunk is page aligned, the requested size is allocated from the front of the chunk
  - If chunk is not page aligned, the requested size is allocated at the end of the chunk
- The remaining fragment of the split chunk is put at the tail of the proper ListHeads[n] list

# Splitting Pool Chunks



# Free Algorithm

- The free algorithm inspects the pool header of the chunk to be freed and frees it to the appropriate list
  - Implemented by **ExFreePoolWithTag**
- Bordering free chunks may be merged with the freed chunk to reduce fragmentation
  - Windows 7 uses safe unlinking in the merging process

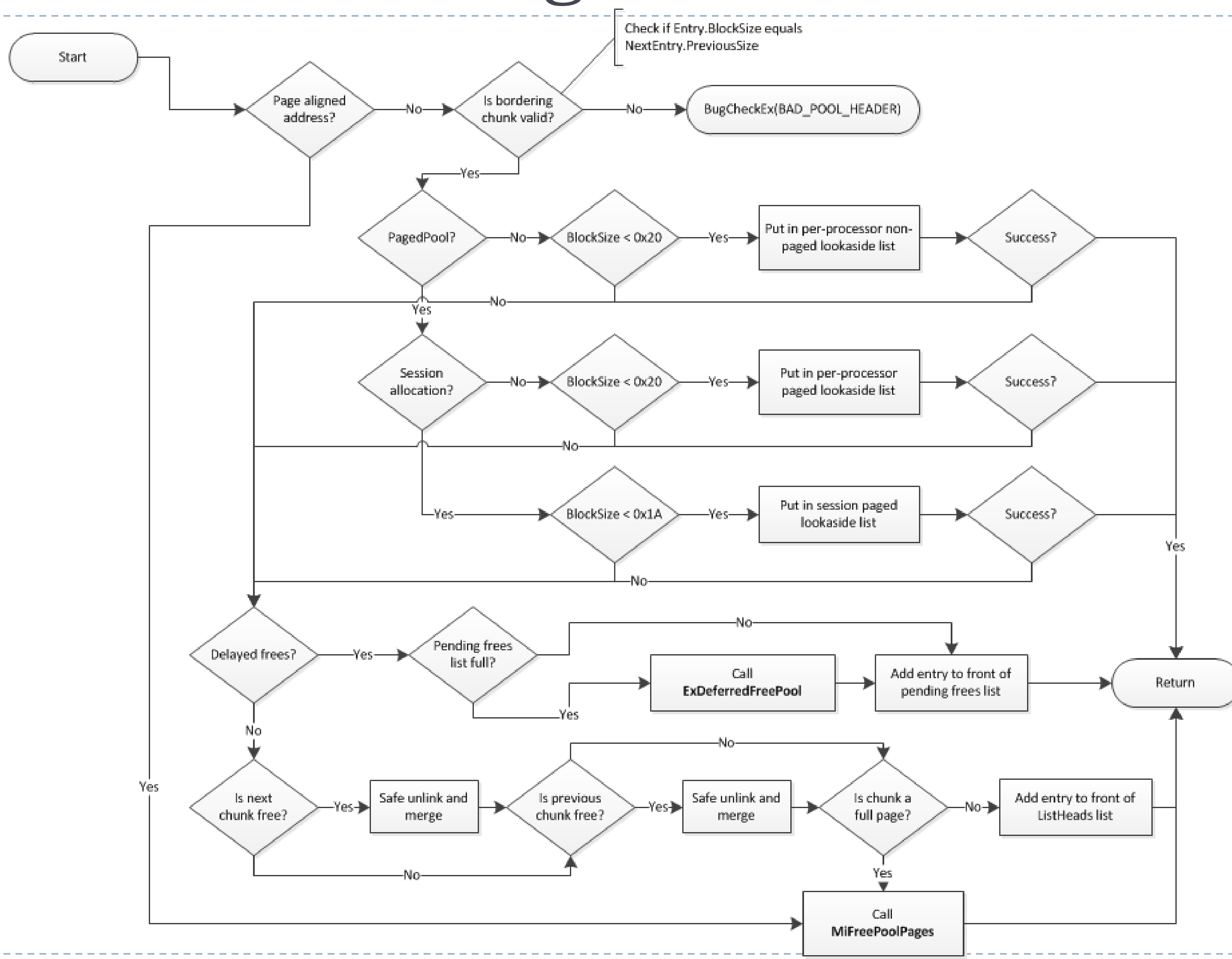
# ExFreePoolWithTag (1/2)

- **VOID ExFreePoolWithTag(PVOID Address, ULONG Tag)**
- If Address (chunk) is page aligned
  - Call **nt!MiFreePoolPages**
- If Chunk->BlockSize != NextChunk->PreviousSize
  - BugCheckEx(BAD_POOL_HEADER)
- If (PoolType & PagedPoolSession) and BlockSize <= 0x19
  - Put in session pool lookaside list
- Else If BlockSize <= 0x20 and pool is local to processor
  - If (PoolType & PagedPool)
    - Put in per-processor paged lookaside list
  - Else (NonPagedPool)
    - Put in per-processor non-paged lookaside list
- Return on sucess

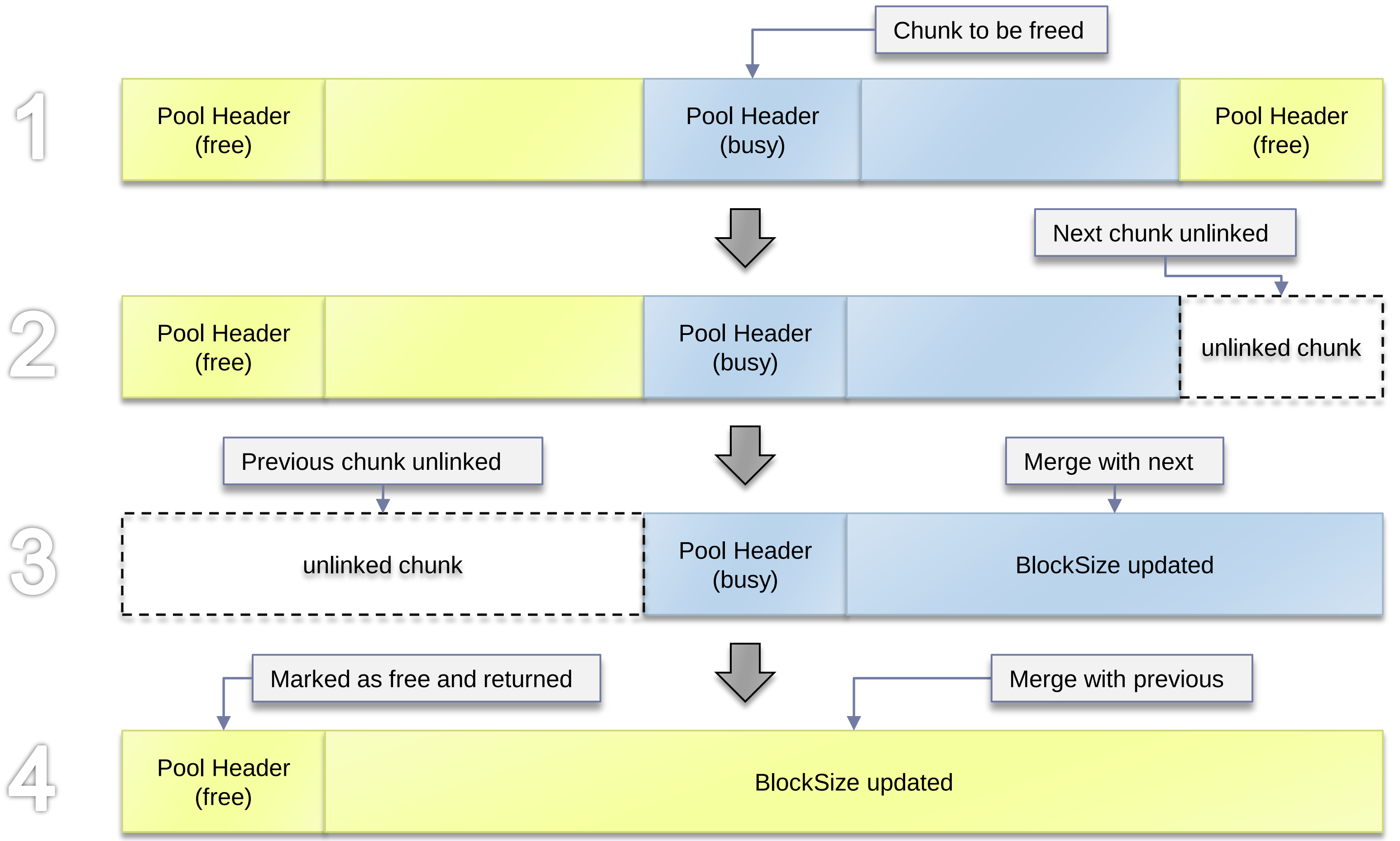
# ExFreePoolWithTag (2/2)

- If the DELAY_FREE pool flag is set
  - If pending frees >= 0x20
    - Call **nt!ExDeferredFreePool**
  - Add to front of pending frees list (singly-linked)
- Else
  - If next chunk is free and not page aligned
    - Safe unlink and merge with current chunk
  - If previous chunk is free
    - Safe unlink and merge with current chunk
  - If resulting chunk is a full page
    - Call **nt!MiFreePoolPages**
  - Else
    - Add to front of appropriate ListHeads list

# ExFreePoolWithTag



# Merging Pool Chunks



# Delayed Pool Frees

- A performance optimization that frees several pool allocations at once to amortize pool acquisition/release
  - Briefly mentioned in mxatone[2008]
- Enabled when **MmNumberOfPhysicalPages** >= 0x1fc00
  - Equivalent to 508 MBs of RAM on IA-32 and AMD64
  - **nt!ExpPoolFlags** & 0x200
- Each call to **ExFreePoolWithTag** appends a pool chunk to a singly-linked deferred free list specific to each pool descriptor
  - Current number of entries is given by **PendingFreeDepth**
  - The list is processed by the function **ExDeferredFreePool** if it has 32 or more entries

# ExDeferredFreePool

- **VOID ExDeferredFreePool(PPOOL_DESCRIPTOR PoolDescriptor, BOOLEAN bMultiThreaded)**
- For each entry on pending frees list
  - If next chunk is free and not page aligned
    - Safe unlink and merge with current chunk
  - If previous chunk is free
    - Safe unlink and merge with current chunk
  - If resulting chunk is a full page
    - Add to full page list
  - Else
    - Add to front of appropriate ListHeads list
- For each page in full page list
  - Call **nt!MiFreePoolPages**

# Free Pool Chunk Ordering

- Frees to the lookaside and pool descriptor ListHeads are always put in the front of the appropriate list
  - Exceptions are remaining fragments of split blocks which are put at the tail of the list
  - Blocks are split when the pool allocator returns chunks larger than the requested size
    - Full pages split in **ExpBigPoolAllocation**
    - ListHeads[n] entries split in **ExAllocatePoolWithTag**
- Allocations are always made from the most recently used blocks, from the front of the appropriate list
  - Attempts to use the CPU cache as much as possible

# Kernel Pool Attacks

Modern Kernel Pool Exploitation:
Attacks and Techniques

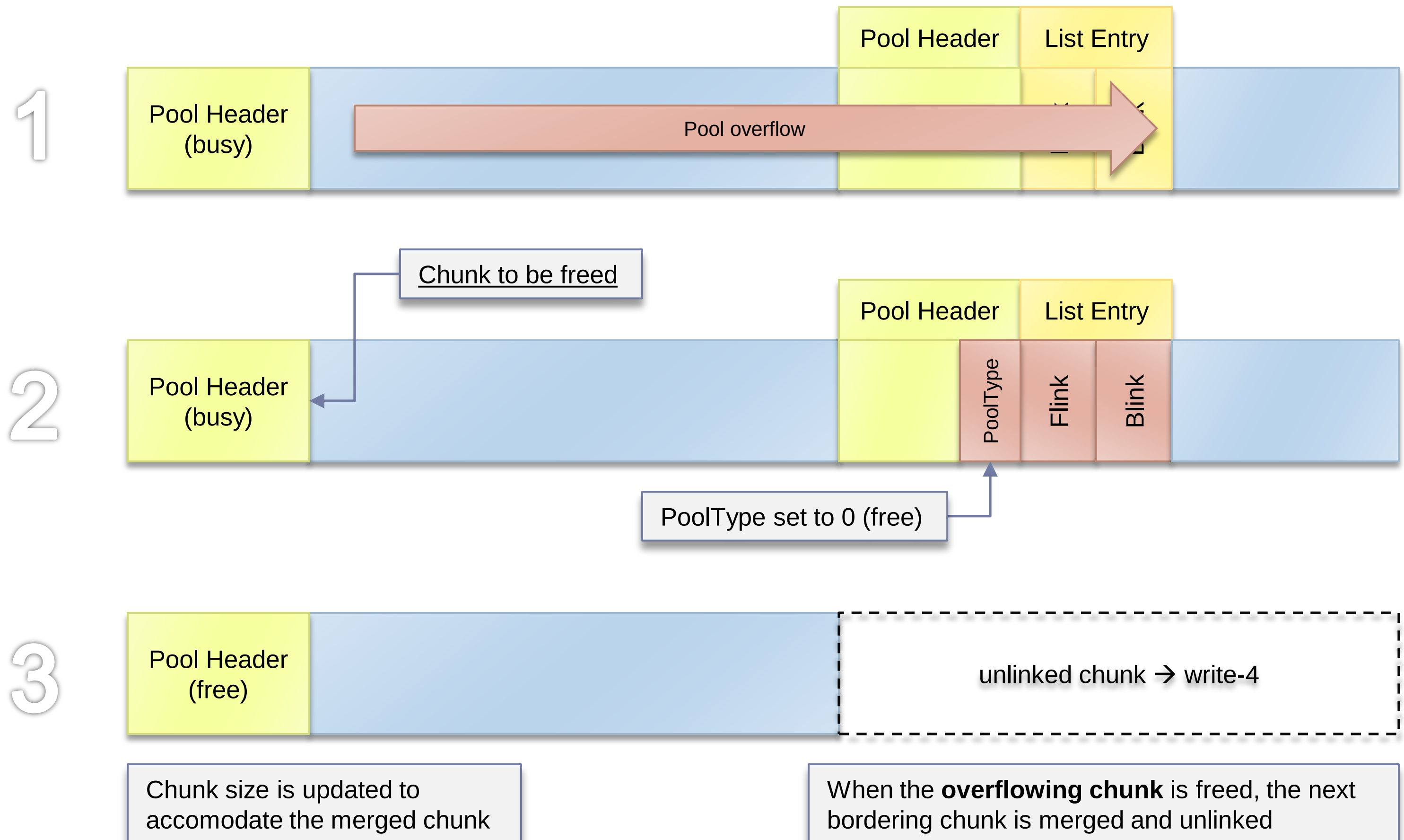
# Overview

- Traditional ListEntry Attacks (< Windows 7)
- ListEntry Flink Overwrite
- Lookaside Pointer Overwrite
- PoolIndex Overwrite
- PendingFrees Pointer Overwrite
- Quota Process Pointer Overwrite

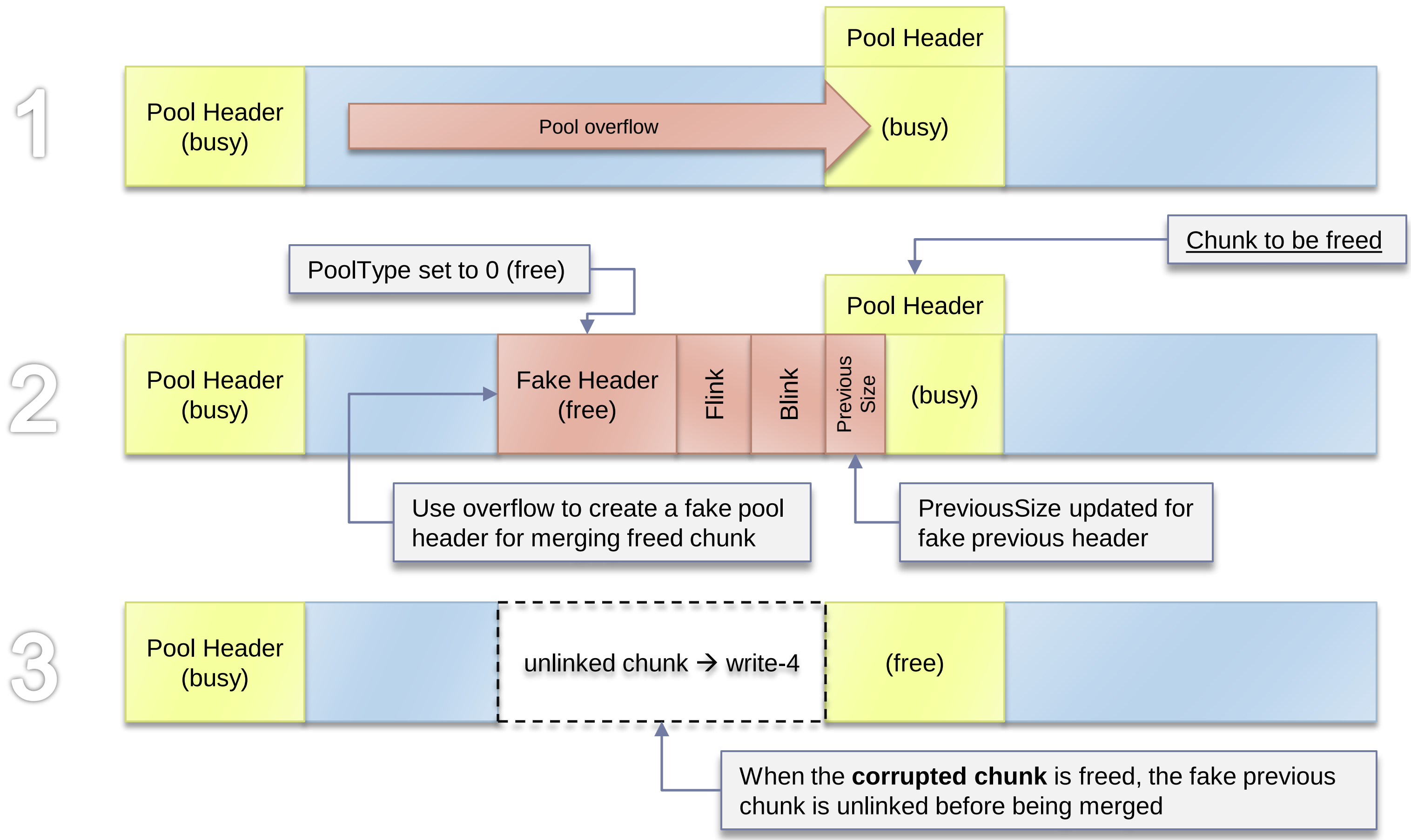
# ListEntry Overwrite (< Windows 7)

- All free list (ListHeads) pool chunks are linked together by LIST_ENTRY structures
- Vista and former versions do not validate the structures' forward and backward pointers
- A ListEntry overwrite may be leveraged to trigger a write-4 in the following situations
  - Unlink in merge with next pool chunk
  - Unlink in merge with previous pool chunk
  - Unlink in allocation from ListHeads[n] free list
- Discussed in Kortchinsky[2008] and SoBeIt[2005]

# ListEntry Overwrite (Merge With Next)



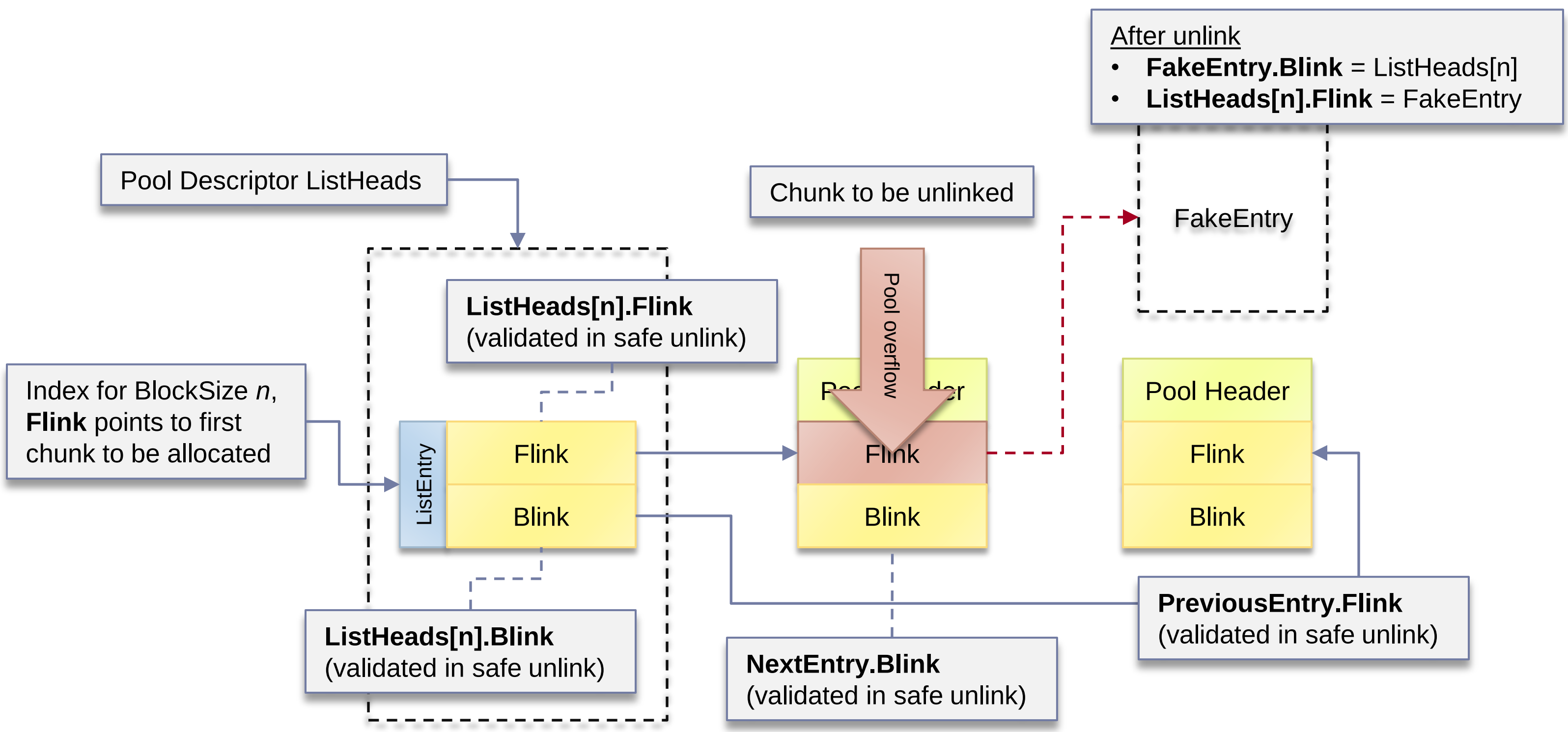
# ListEntry Overwrite (Merge With Previous)



# ListEntry Flink Overwrite

- Windows 7 uses safe unlinking to validate the LIST_ENTRY pointers of a chunk being unlinked
- In allocating a pool chunk from a ListHeads free list, the kernel fails to properly validate its forward link
  - The algorithm validates the ListHeads[n] LIST_ENTRY structure instead
- Overwriting the <u>forward link</u> of a free chunk may cause the address of ListHeads[n] to be written to an attacker controlled address
  - Target ListHeads[n] list must hold at least two free chunks

# The Not So Safe Unlink



# ListEntry Flink Overwrite

- In the following output, the address of ListHeads[n] (**esi**) in the pool descriptor is written to an attacker controlled address (**eax**)
- Pointers are not sufficiently validated when allocating a pool chunk from the free list

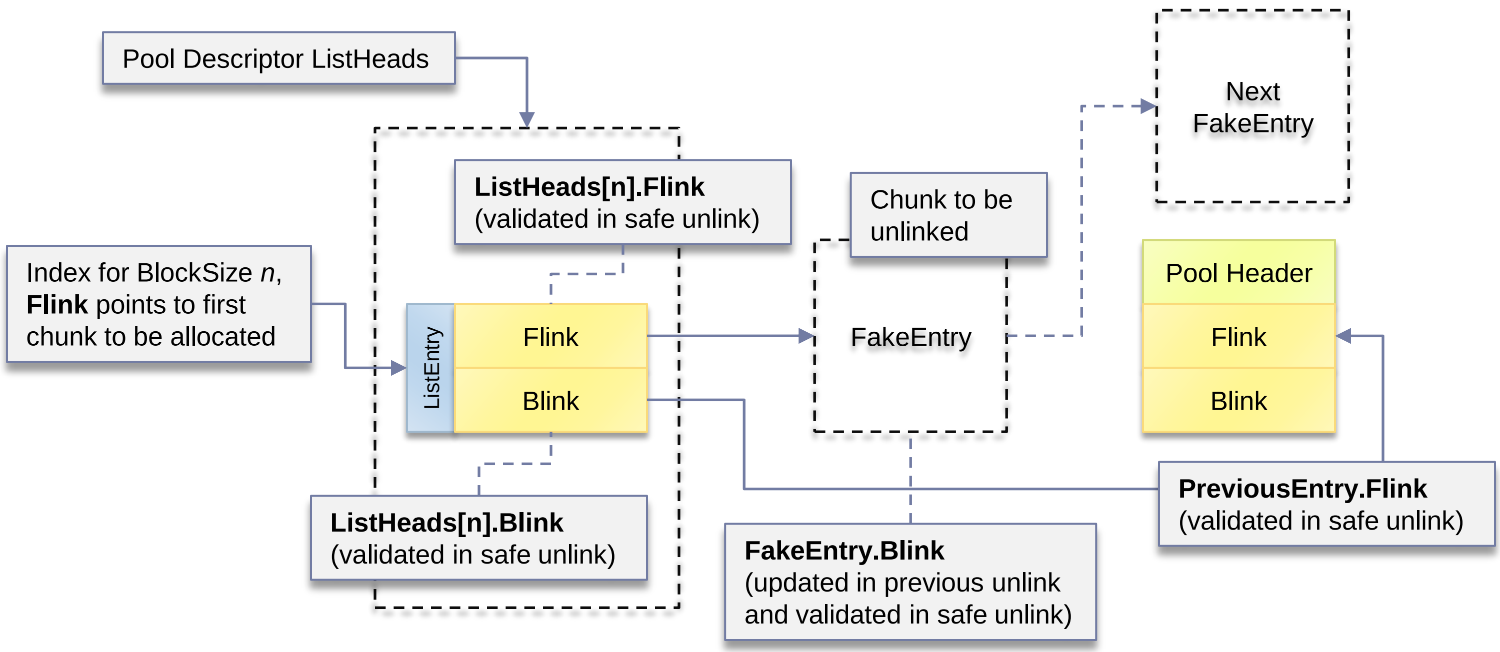
```
eax=80808080 ebx=829848c0 ecx=8cc15768 edx=8cc43298 esi=82984a18 edi=829848c4
eip=8296f067 esp=82974c00 ebp=82974c48 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010246

nt!ExAllocatePoolWithTag+0x4b7:
8296f067 897004          mov     dword ptr [eax+4],esi ds:0023:80808084=????????
```

# ListEntry Flink Overwrite

- After unlink, the attacker may control the address of the next allocated entry
  - **ListHeads[n].Flink** = FakeEntry
- FakeEntry can be safely unlinked as its blink was updated to point back to ListHeads[n]
  - **FakeEntry.Blink** = ListHeads[n]
- If a user-mode pointer is used in the overwrite, the attacker could fully control the contents of the next allocation

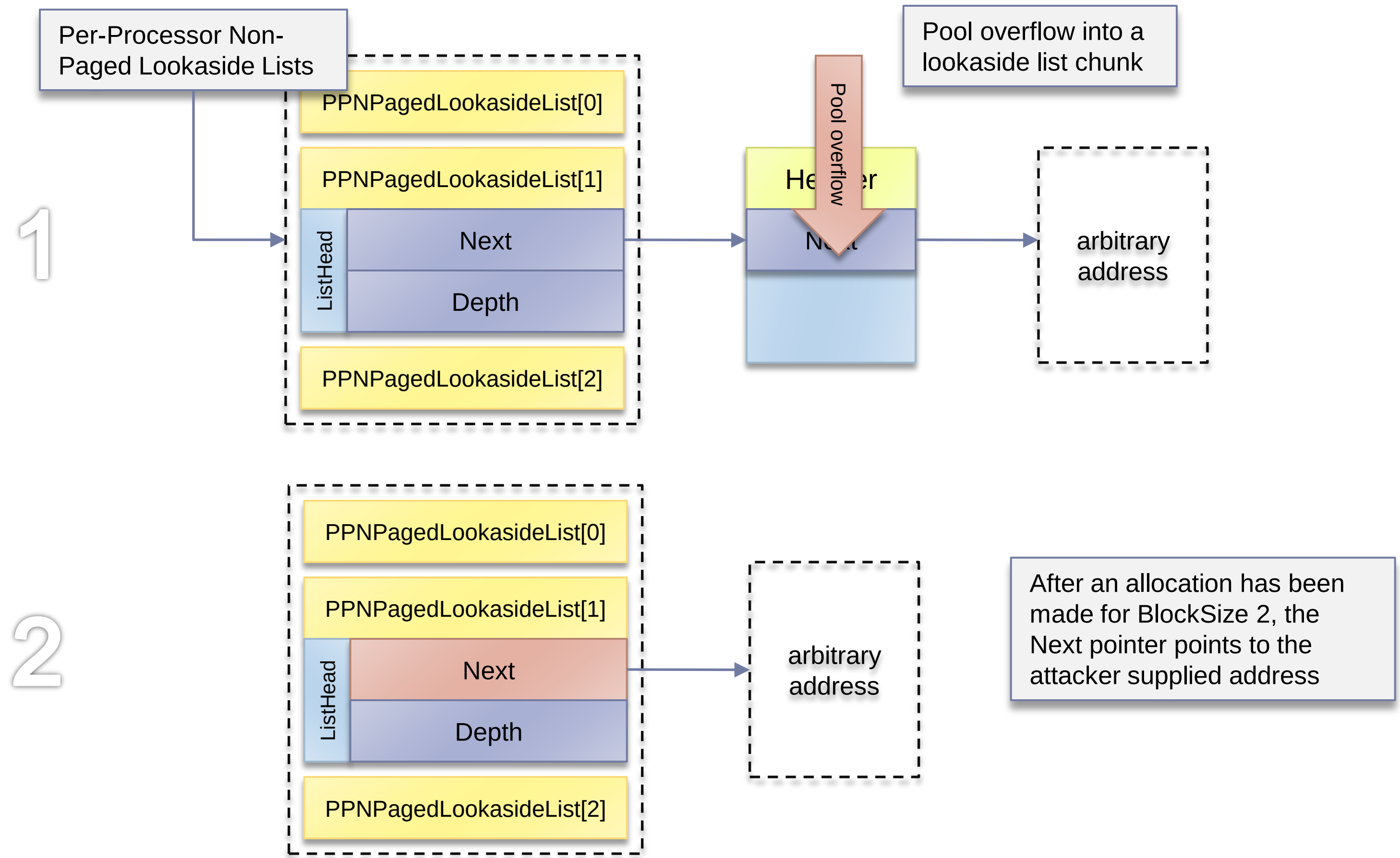
# ListEntry Flink Overwrite



# Lookaside Pointer Overwrite

- Pool chunks and pool pages on lookaside lists are singly-linked
  - Each entry holds a pointer to the next entry
  - Overwriting a next pointer may cause the kernel pool allocator to return an attacker controlled address
- A pool chunk is freed to a lookaside list if the following hold
  - BlockSize <= 0x20 for paged/non-paged pool chunks
  - BlockSize <= 0x19 for paged session pool chunks
  - Lookaside list for target BlockSize is not full
  - Hot/cold page separation is not used

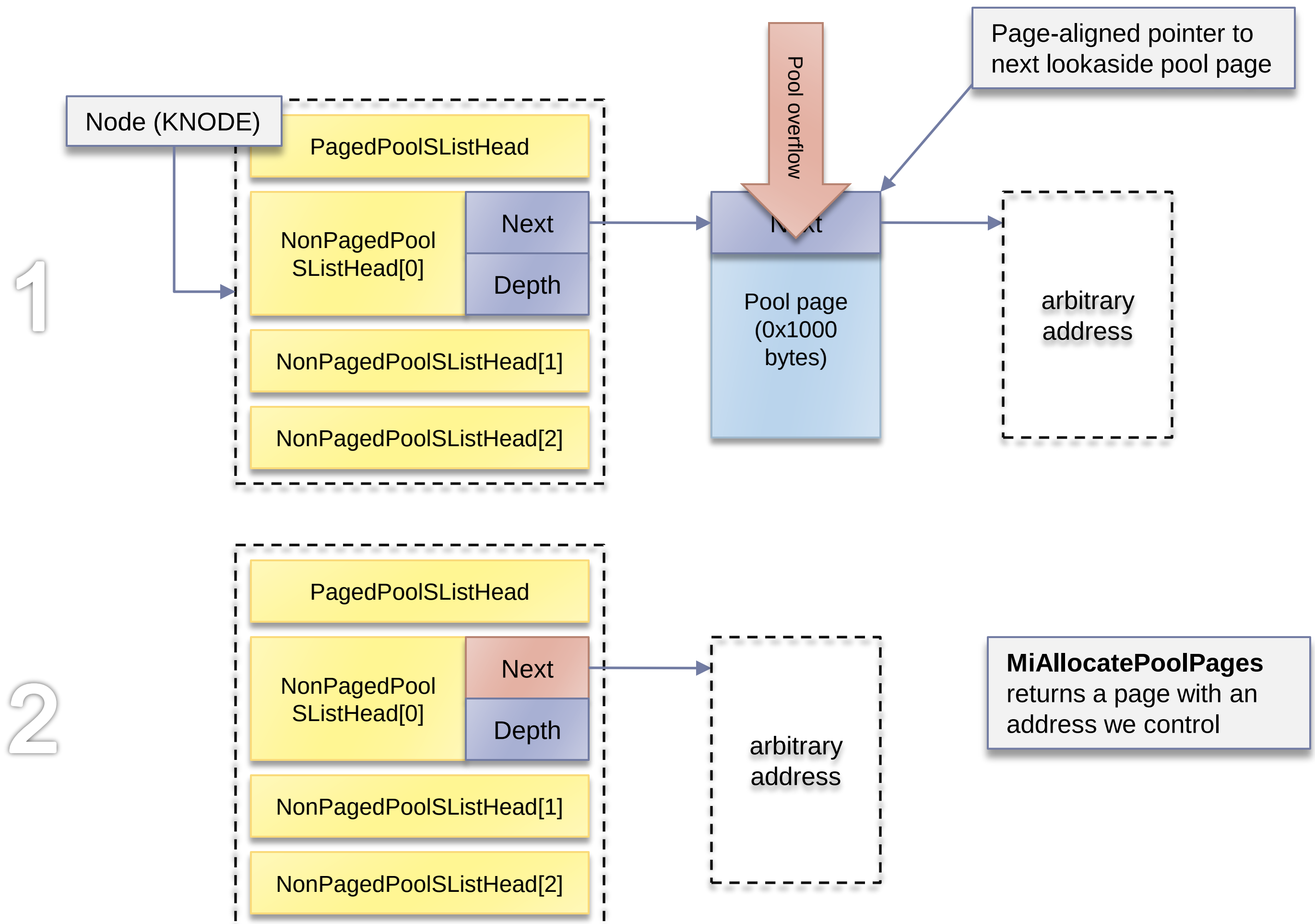
# Lookaside Pointer Overwrite (Chunks)



# Lookaside Pointer Overwrite (Pages)

- A pool page is freed to a lookaside list if the following hold
  - NumberOfPages = 1 for paged pool pages
  - NumberOfPages <= 3 for non-paged pool pages
  - Lookaside list for target page count is not full
    - Size limit determined by physical page count in system
- A pointer overwrite of lookaside pages requires at most a pointer-wide overflow
  - No pool headers on free pool pages!
  - Partial pointer overwrites may also be sufficient

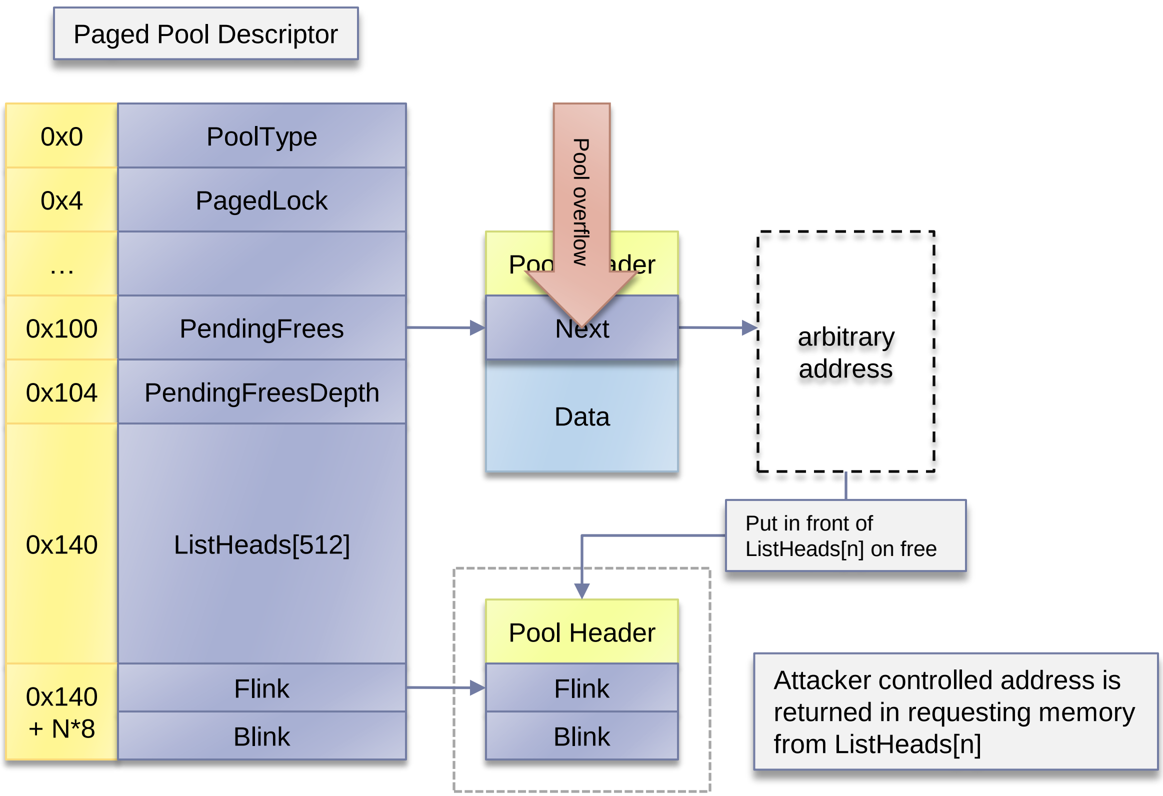
# Lookaside Pointer Overwrite (Pages)



# PendingFrees Pointer Overwrite

- Pool chunks waiting to be freed are stored in the pool descriptor deferred free list
  - Singly-linked (similar to lookaside list)
- Overwriting a chunk's next pointer will cause an arbitrary address to be freed
  - Inserted in the front of ListHeads[n]
  - Next pointer must be NULL to end the linked list
- In freeing a user-mode address, the attacker may control the contents of subsequent allocations
  - Must be made from the same process context

# PendingFrees Pointer Overwrite



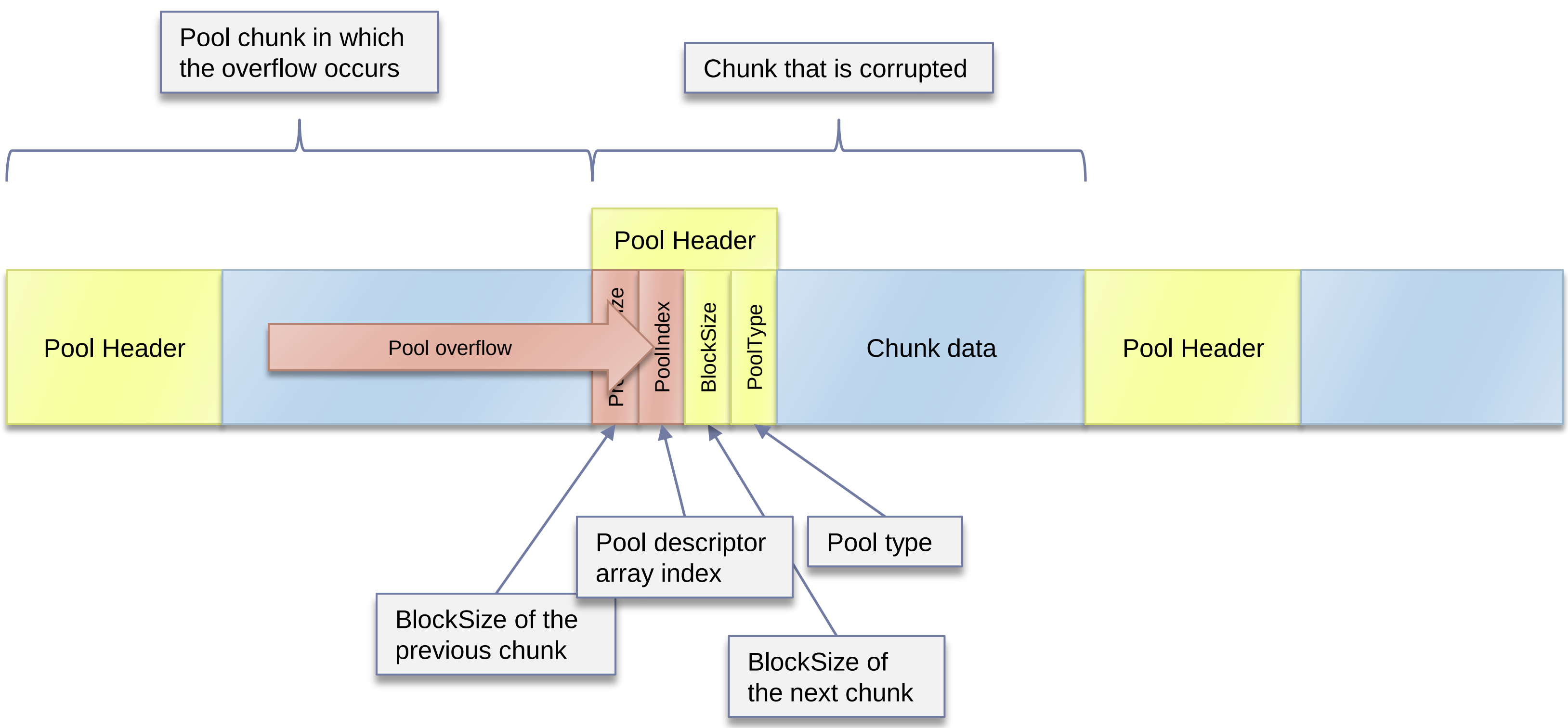
# PendingFrees Pointer Overwrite Steps

- Free a chunk to the deferred free list
- Overwrite the chunk's next pointer
  - Or any of the deferred free list entries (32 in total)
- Trigger processing of the deferred free list
  - Attacker controlled pointer freed to designated free list
- Force allocation of the controlled list entry
  - Allocator returns user-mode address
- Corrupt allocated entry
- Trigger use of corrupted entry

# PoolIndex Overwrite

- A pool chunk's PoolIndex denotes an index into the associated pool descriptor array
- For paged pools, PoolIndex always denotes an index into the **nt!ExpPagedPoolDescriptor** array
  - On checked builds, the index value is validated in a compare against **nt!ExpNumberOfPagedPools**
  - On free (retail) builds, the index is not validated
- For non-paged pools, PoolIndex denotes an index into **nt!ExpNonPagedPoolDescriptor** when there are multiple NUMA nodes
  - PoolIndex is not validated on free builds

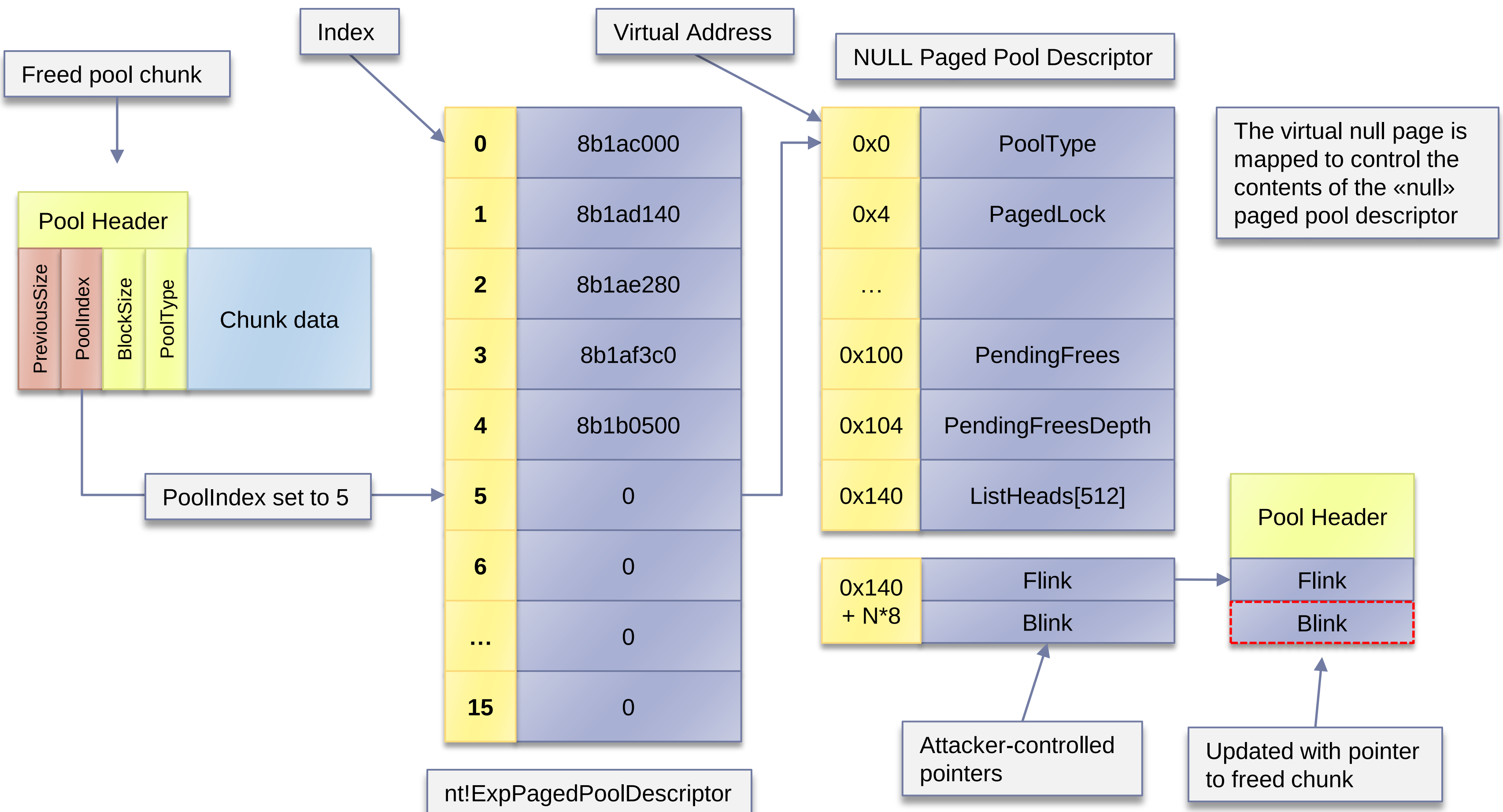
# PoolIndex Overwrite



# PoolIndex Overwrite

- A malformed PoolIndex may cause an allocated pool chunk to be freed to a null-pointer pool descriptor
  - Controllable with null page allocation
  - Requires a 2 byte pool overflow
- When linking in to a controlled pool descriptor, the attacker can write the address of the freed chunk to an arbitrary location
  - No checks performed when “linking in”
  - All ListHeads entries are fully controlled
  - **ListHeads[n].Flink->Blink** = FreedChunk

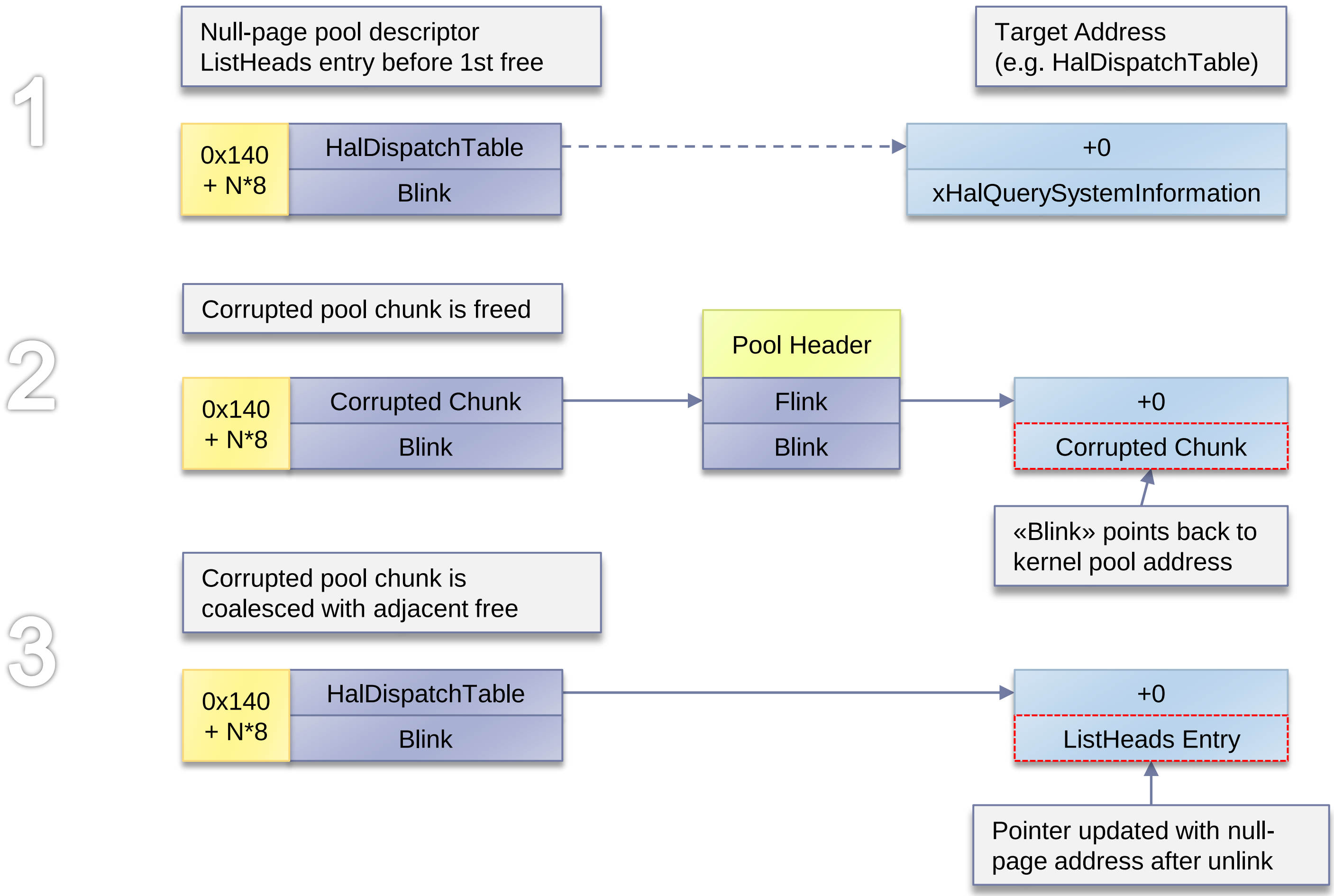
# PoolIndex Overwrite



# PoolIndex Overwrite + Coalescing

- If delayed frees are not used, the PoolIndex attack writes a kernel pool address to an arbitrary location
  - **ListHeads[n].Flink->Blink** = FreedChunk
- We can extend this to an arbitrary write of a null-page address by coalescing the freed (corrupted) chunk
  - E.g. free an adjacent pool chunk
- This will cause the initial freed chunk to be unlinked from the free list and update the **Blink** with a pointer back to the ListHeads entry (null-page)

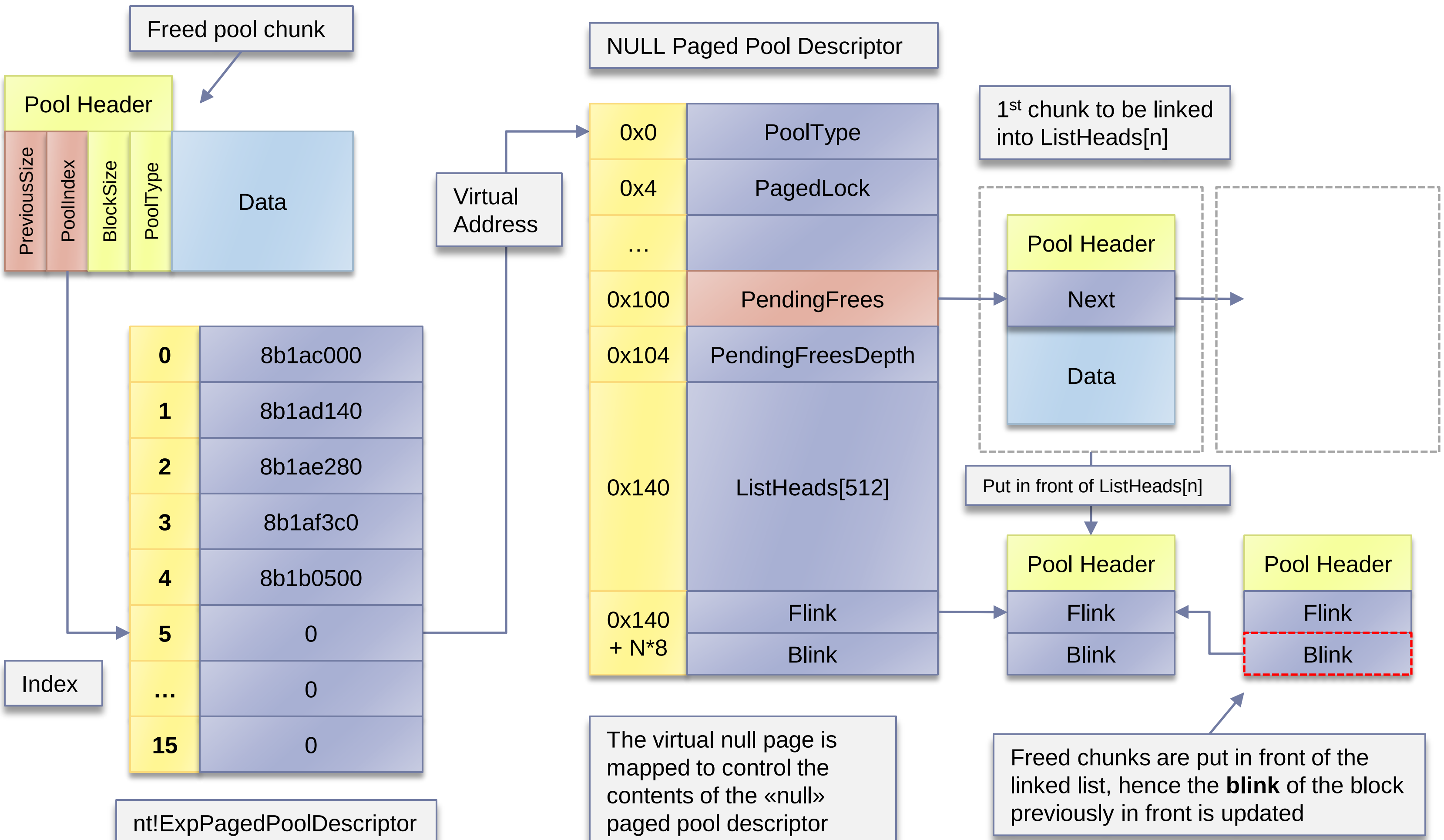
# PoolIndex Overwrite + Coalescing



# PoolIndex Overwrite (Delayed Frees)

- If delayed pool frees is enabled, the same effect can be achieved by creating a fake PendingFrees list
  - First entry should point to a user crafted chunk
- The **PendingFreeDepth** field of the pool descriptor should be >= 0x20 to trigger processing of the PendingFrees list
- The free algorithm of **ExDeferredFreePool** does basic validation on the crafted chunks
  - Coalescing / safe unlinking
  - The freed chunk should have busy bordering chunks

# PoolIndex Overwrite (Delayed Frees)



# PoolIndex Overwrite (Example)

- In controlling the PendingFrees list, a user-controlled virtual address (**eax**) can be written to an arbitrary destination address (**esi**)
- In turn, this can be used to corrupt function pointers used by the kernel to execute arbitrary code

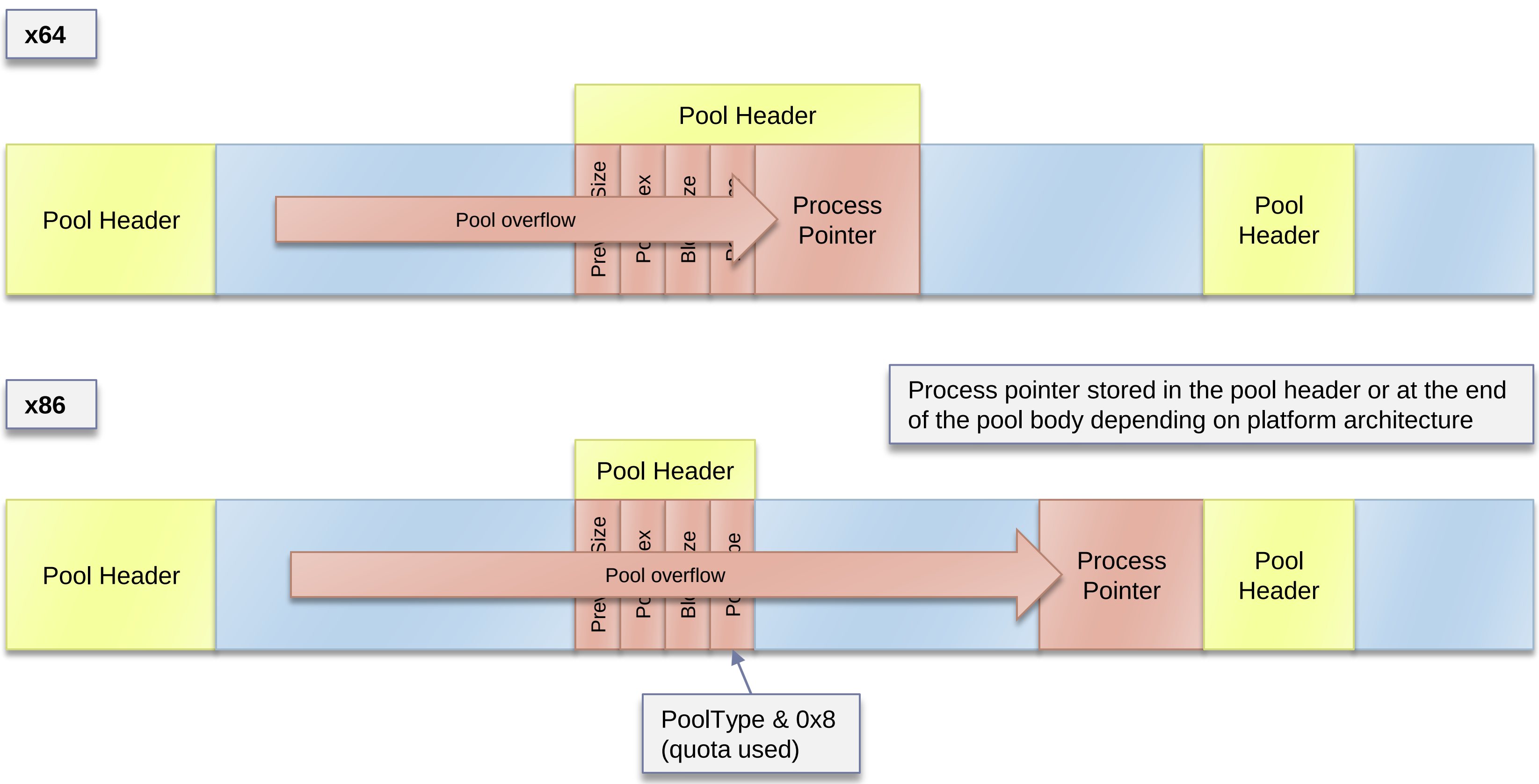
```
eax=20000008 ebx=000001ff ecx=000001ff edx=00000538 esi=80808080 edi=00000000
eip=8293c943 esp=9c05fb20 ebp=9c05fb58 iopl=0         nv up ei pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010202

nt!ExDeferredFreePool+0x2e3:
8293c943 894604          mov     dword ptr [esi+4],eax ds:0023:80808084=????????
```

# Quota Process Pointer Overwrite

- Quota charged pool allocations store a pointer to the associated process object
  - **ExAllocatePoolWithQuotaTag(…)**
  - x86: last four bytes of pool body
  - x64: last eight bytes of pool header
- Upon freeing a pool chunk, the quota is released and the process object is dereferenced
  - The object's reference count is decremented
- Overwriting the process object pointer could allow an attacker to free an in-use process object or corrupt arbitrary memory

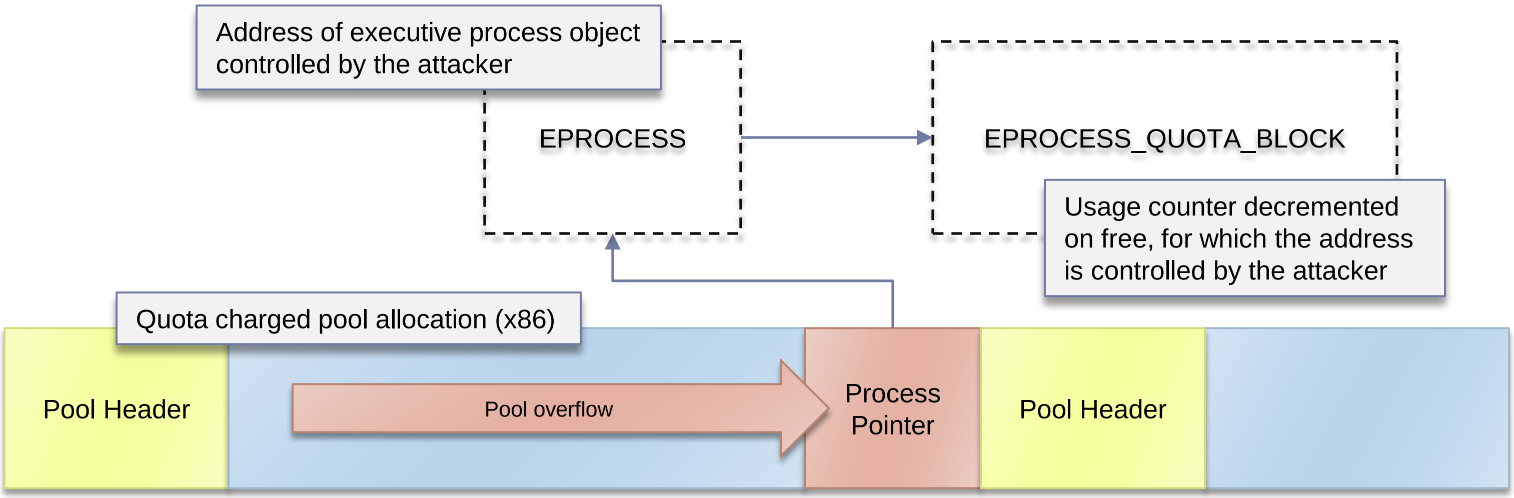
# Quota Process Pointer Overwrite



# Quota Process Pointer Overwrite

- Quota information is stored in a EPROCESS_QUOTA_BLOCK structure
  - Pointed to by the EPROCESS object
  - Provides information on limits and how much quota is being used
- On free, the charged quota is returned by subtracting the size of the allocation from the quota used
  - An attacker controlling the <u>quota block pointer</u> could decrement the value of an arbitrary address
  - More on this later!

# Arbitrary Pointer Decrement



# Summary of Attacks

- Corruption of busy pool chunk
  - BlockSize <= 0x20
    - PoolIndex + PoolType/BlockSize Overwrite
    - Quota Process Pointer Overwrite
  - BlockSize > 0x20
    - PoolIndex (+PoolType) Overwrite
    - Quota Process Pointer Overwrite
- Corruption of free pool chunk
  - BlockSize <= 0x20
    - Lookaside Pointer Overwrite
  - BlockSize > 0x20
    - ListEntry Flink Overwrite / PendingFrees Pointer Overwrite

# Case Studies

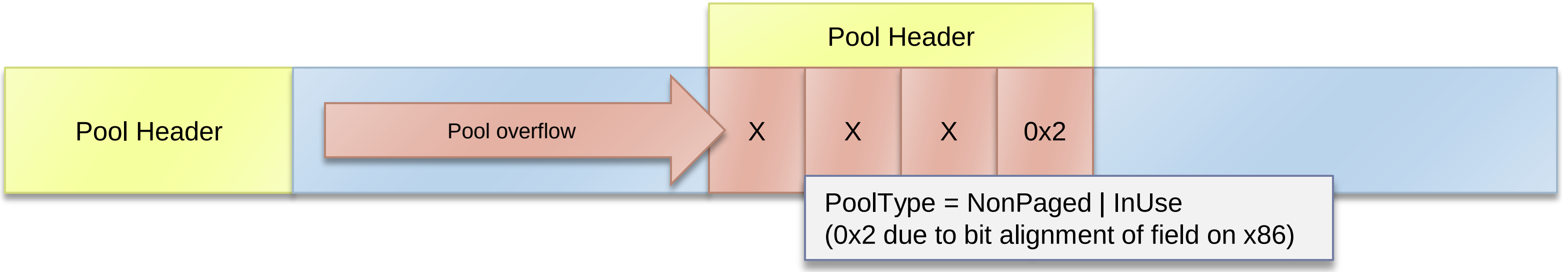
Modern Kernel Pool Exploitation:
Attacks and Techniques

# Case Study Agenda

- Two pool overflow vulnerabilities
  - Both perceived as difficult to exploit
- CVE-2010-3939 (MS10-098)
  - Win32k CreateDIBPalette() Pool Overflow Vulnerability
- CVE-2010-1893 (MS10-058)
  - Integer Overflow in Windows Networking Vulnerability

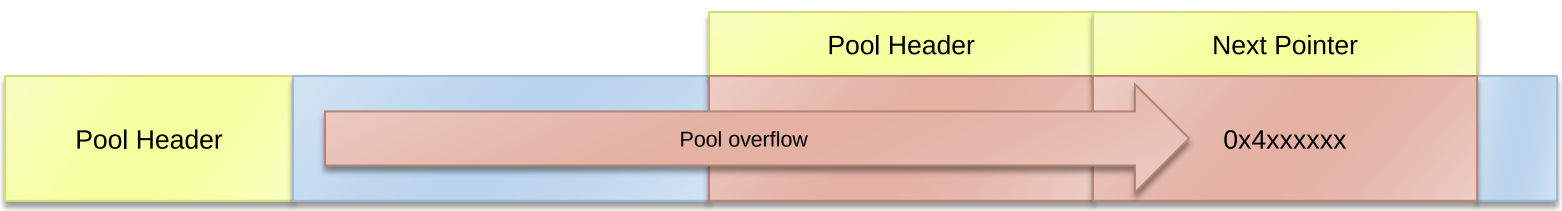
# CVE-2010-3939 (MS10-098)

- Pool overflow in win32k!CreateDIBPalette()
  - Discovered by Arkon
- Function did not validate the number of color entries in the color table used by a bitmap
  - BITMAPINFOHEADER.biClrUsed
- Every fourth byte of the overflowing buffer was set to 0x4
  - Can only reference 0x4xxxxxx addresses (user-mode)
  - PoolType is always set to NonPaged



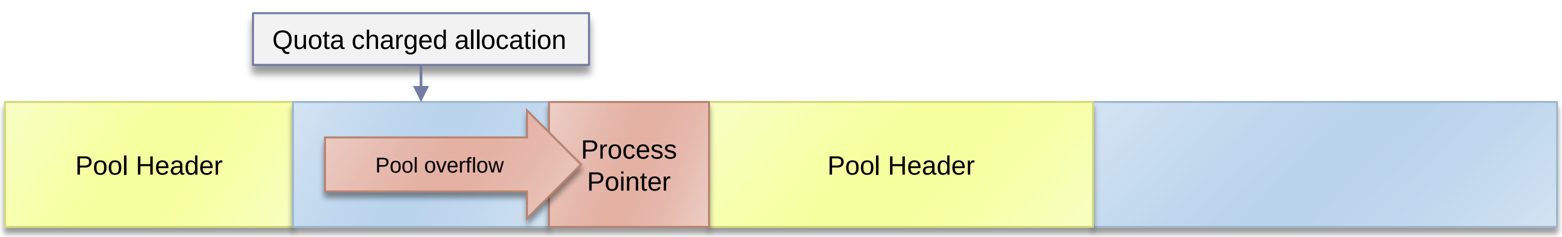
# CVE-2010-3939 (MS10-098)

- The attacker could coerce the pool allocator to return a user-mode pool chunk
  - ListEntry Flink Overwrite
  - Lookaside Overwrite
- Requires the kernel pool to be cleaned up in order for execution to continue safely
  - Repair/remove broken linked lists



# CVE-2010-3939 (MS10-098)

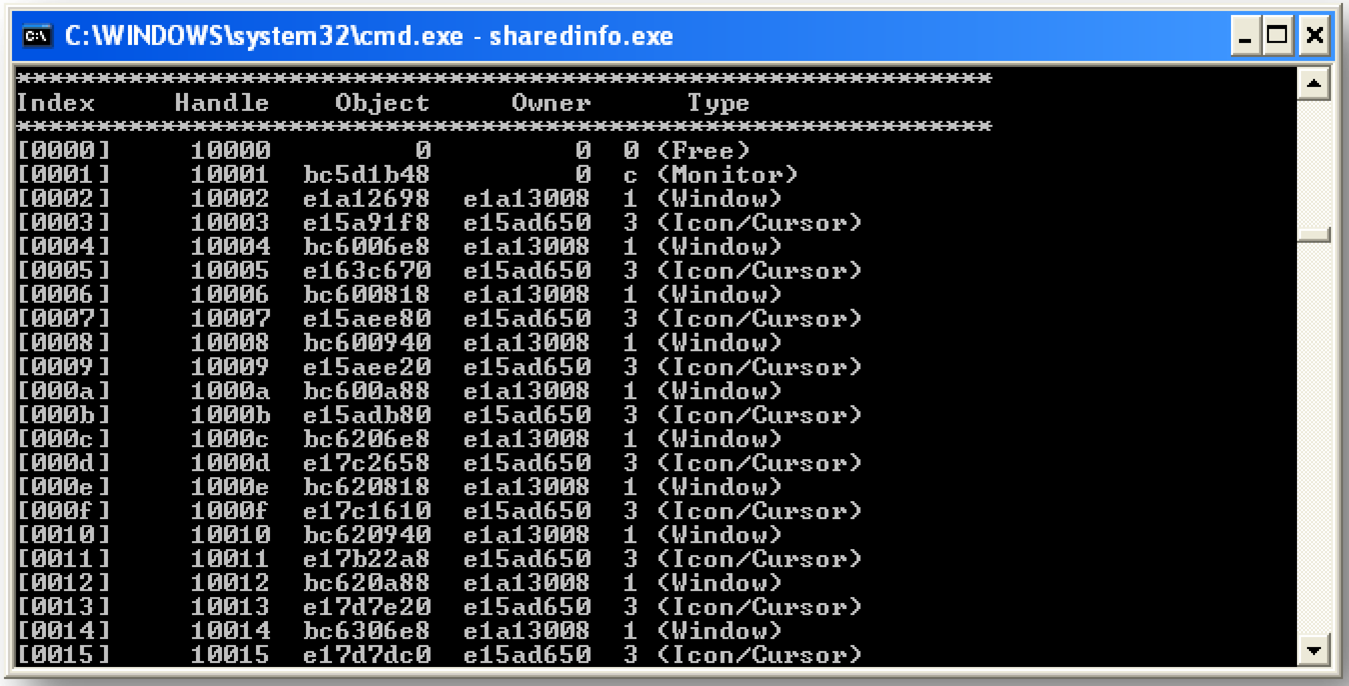
- Vulnerable buffer is also quota charged
  - Can overwrite the process object pointer (x86)
  - No pool chunks are corrupted (clean!)
- Tactic: Decrement the value of a kernel-mode window object procedure pointer
  - Trigger the vulnerability n-times until it points to user-mode memory and call the procedure



# Locating Window Objects

- Via Window Manager (USER) Handle Table
  - CsrClientConnectToServer (USERSRV_INDEX)
  - Windows 7: user32!gSharedInfo
  - Windows XP: user32!UserRegisterWowHandlers
- Via User-Mode Mapped Window Object
  - NtUserCallOneParam(…) → win32k!_MapDesktopObject
  - Patch any routine that calls user32!ValidateHwnd to return the window object pointer (user-mode)
    - E.g. IsServerSideWindow()

# Handle Table From User-Mode

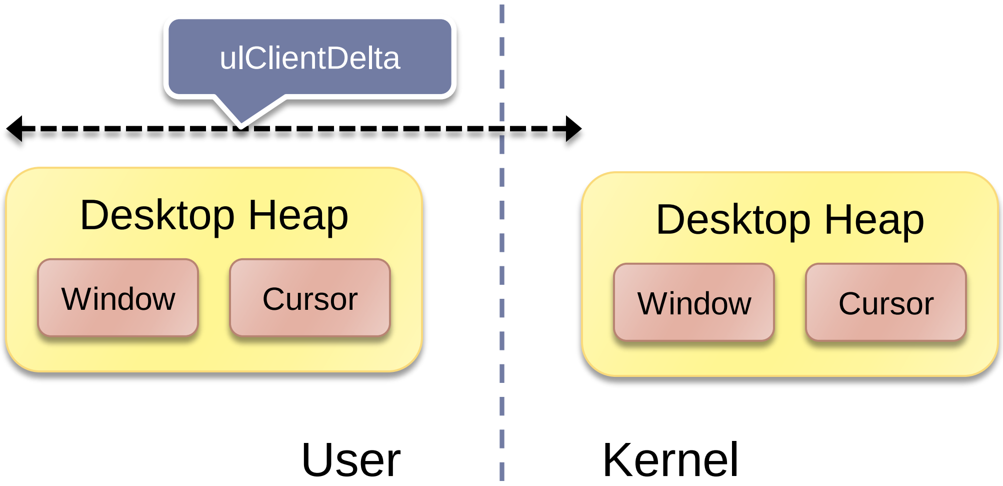


C:\WINDOWS\system32\cmd.exe - sharedinfo.exe

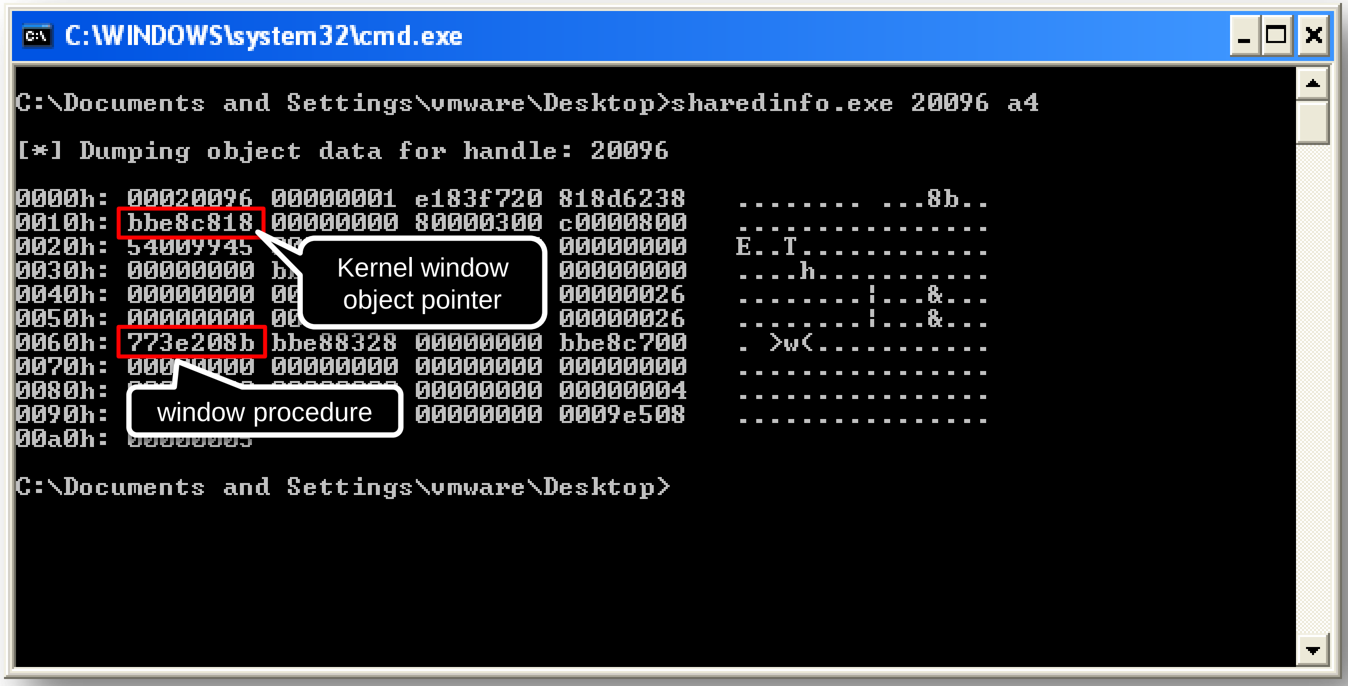
```
***************************************************************
Index       Handle      Object      Owner       Type
***************************************************************
[0000]      10000            0           0  0 (Free)
[0001]      10001  bc5d1b48              0  c (Monitor)
[0002]      10002  e1a12698   e1a13008  1 (Window)
[0003]      10003  e15a91f8   e15ad650  3 (Icon/Cursor)
[0004]      10004  bc6006e8   e1a13008  1 (Window)
[0005]      10005  e163c670   e15ad650  3 (Icon/Cursor)
[0006]      10006  bc600818   e1a13008  1 (Window)
[0007]      10007  e15aee80   e15ad650  3 (Icon/Cursor)
[0008]      10008  bc600940   e1a13008  1 (Window)
[0009]      10009  e15aee20   e15ad650  3 (Icon/Cursor)
[000a]      1000a  bc600a88   e1a13008  1 (Window)
[000b]      1000b  e15adb80   e15ad650  3 (Icon/Cursor)
[000c]      1000c  bc6206e8   e1a13008  1 (Window)
[000d]      1000d  e17c2658   e15ad650  3 (Icon/Cursor)
[000e]      1000e  bc620818   e1a13008  1 (Window)
[000f]      1000f  e17c1610   e15ad650  3 (Icon/Cursor)
[0010]      10010  bc620940   e1a13008  1 (Window)
[0011]      10011  e17b22a8   e15ad650  3 (Icon/Cursor)
[0012]      10012  bc620a88   e1a13008  1 (Window)
[0013]      10013  e17d7e20   e15ad650  3 (Icon/Cursor)
[0014]      10014  bc6306e8   e1a13008  1 (Window)
[0015]      10015  e17d7dc0   e15ad650  3 (Icon/Cursor)
```

# Kernel-Mode -> User-Mode Address

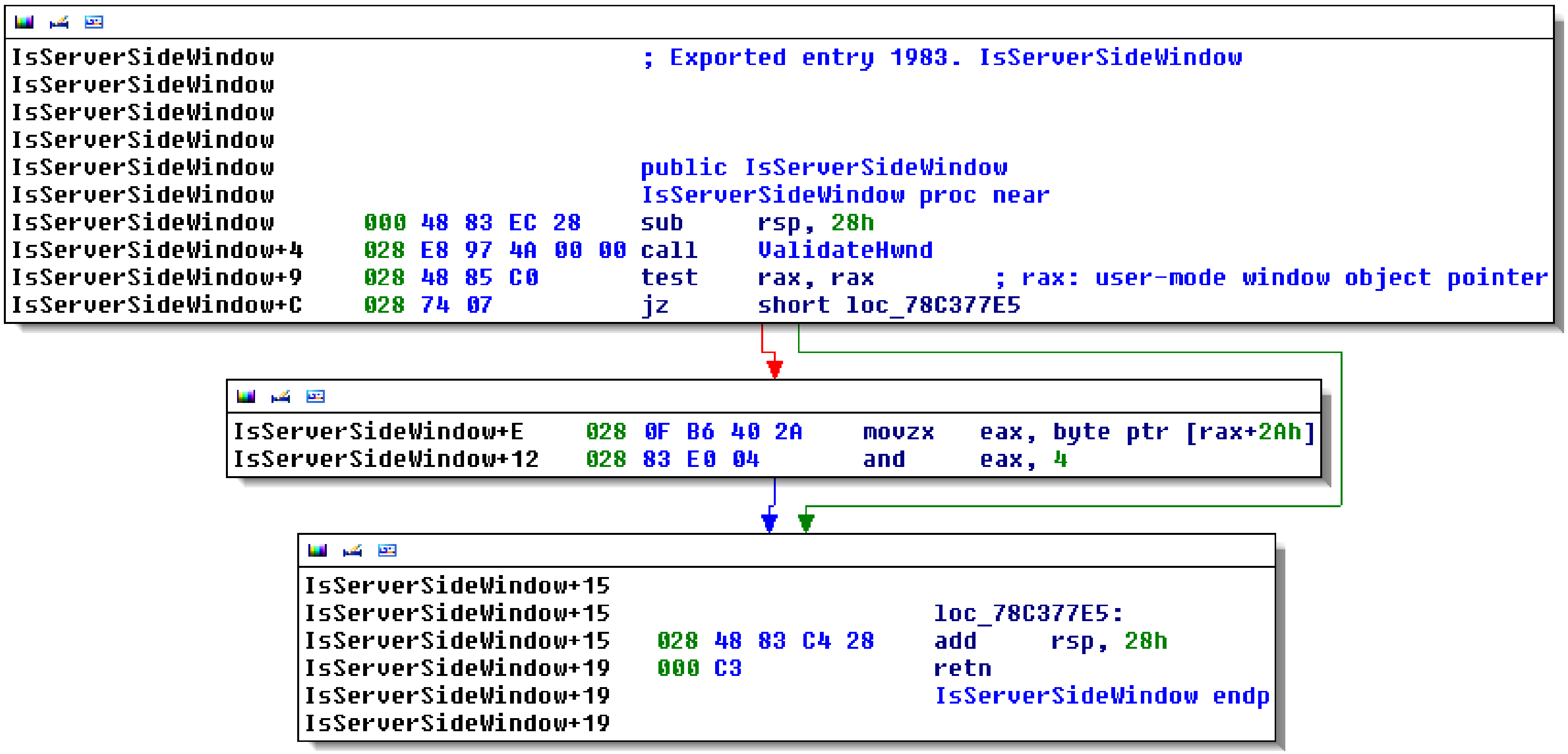
- User-space address of desktop heap objects are computed using **ulClientDelta**
  - NtCurrentTeb()->Win32ClientInfo->ulClientDelta



# Window Objects from User-Mode



# Retrieving Window Object Pointer



# Steps

- Create a default procedure window
  - win32k!xxxDefWindowProc
- Locate the window object in kernel memory
- Corrupt the window procedure pointer
- SendMessage(hwnd,…)

# CVE-2010-3939 (MS10-098)

- Quota Process Pointer Overwrite
  - Demo

# CVE-2010-1893 (MS10-058)

- Integer overflow in tcpip!IppSortDestinationAddresses()
  - Discovered by Matthieu Suiche
  - Affected Windows 7/2008 R2 and Vista/2008
- Function did not use safe-int functions consistently
  - Could result in an undersized buffer allocation, subsequently leading to a pool overflow

# IppSortDestinationAddresses()

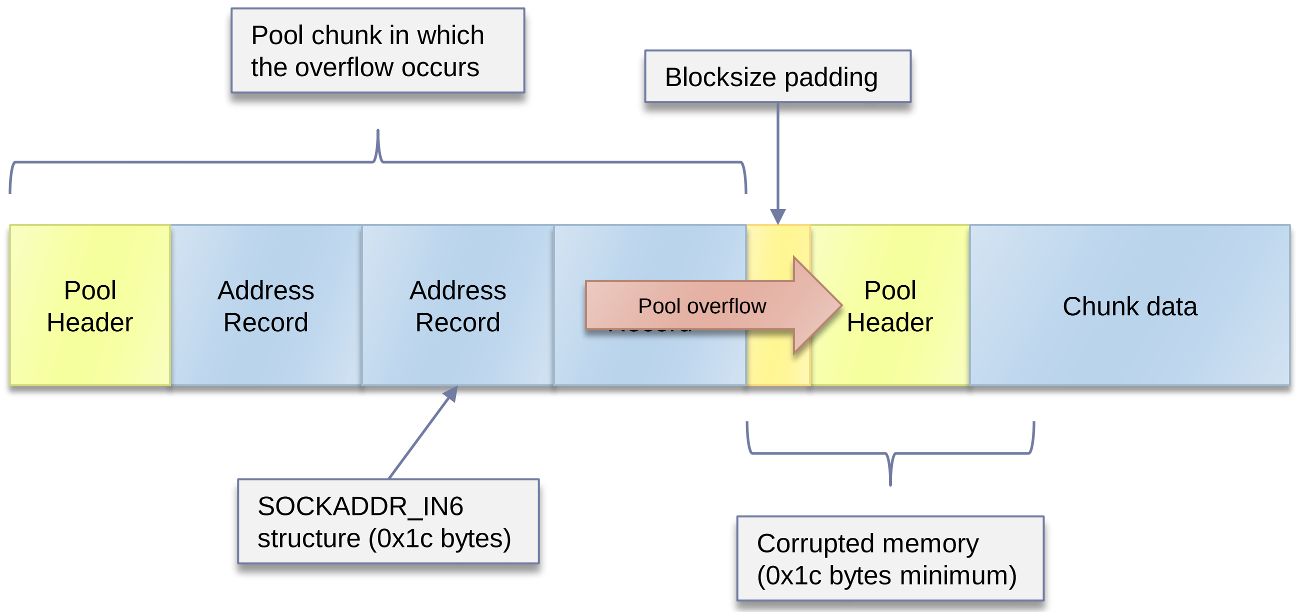
- Sorts a list of IPv6 and IPv4 destination addresses
  - Each address is a SOCKADDR_IN6 record
- Reachable from user-mode by calling WSAIoctl()
  - Ioctl: SIO_ADDRESS_LIST_SORT
  - Buffer: SOCKET_ADDRESS_LIST structure
- Allocates buffer for the address list
  - iAddressCount * sizeof(SOCKADDR_IN6)
  - No overflow checks in multiplication

```
typedef struct _SOCKET_ADDRESS_LIST {
  INT            iAddressCount;
  SOCKET_ADDRESS Address[1];
} SOCKET_ADDRESS_LIST, *PSOCKET_ADDRESS_LIST;
```

# IppFlattenAddressList()

- Copies the user provided address list to the allocated kernel pool chunk
- An undersized buffer could result in a pool overflow
  - Overflows the next pool chunk with the size of an address structure (0x1c bytes)
- Stops copying records if the size != 0x1c or the protocol family != AF_INET6 (0x17)
  - Possible to avoid trashing the kernel pool completely
- The protocol check is done after the memcpy()
  - We can overflow using any combination of bytes

# Pool Overflow



# Exploitation Tactics

- Can use the PoolIndex attack to extend the pool overflow to an arbitrary memory write
  - Must overwrite a busy chunk
- Overwritten chunk must be freed to ListHeads lists
  - BlockSize > 0x20
  - Or… fill the lookaside list
- To overflow the desired pool chunk, we must defragment and manipulate the kernel pool
  - Allocate chunks of the same size
  - Create “holes” by freeing every other chunk

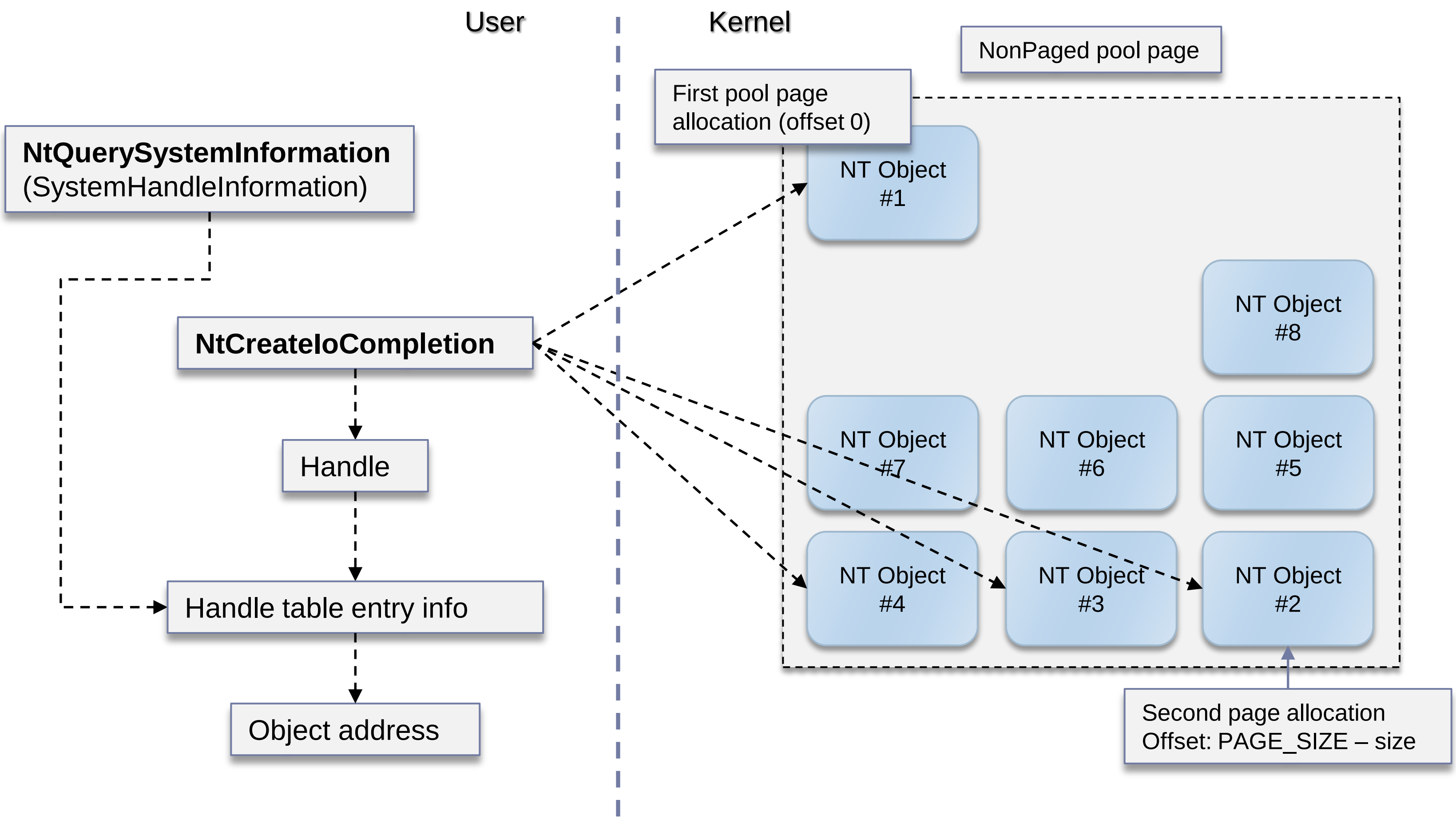
# Filling the Kernel Pool

- What do we use to fill the pool ?
  - Depends on the pool type
  - Should be easy to allocate and free
- NonPaged Pool
  - NT objects (low overhead)
- Paged Pool
  - Unicode strings (e.g. object properties)
- Session Paged Pool
  - Window Manager (USER) and GDI objects

# Kernel Objects in Pool Manipulation

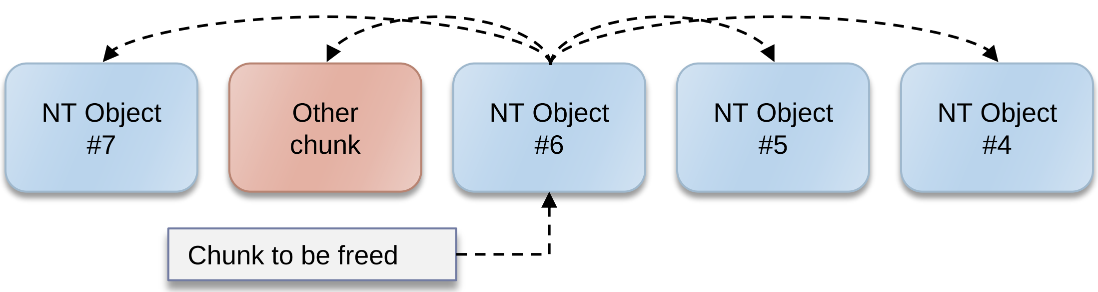
- Trivial to obtain the kernel pointers for executive, window manager, and GDI objects
  - Allows precise control in manipulating kernel pools
- Window Manager (USER) Objects
  - CsrClientConnectToServer(USERSRV_INDEX)
  - Windows 7: user32!gSharedInfo
- GDI Objects
  - Peb()->GdiSharedHandleTable
- NT Objects
  - NtQuerySystemInformation(SystemHandleInfo…)

# Filling the Kernel Pool (NT Objects)



# Enumerating Object Addresses

- For NT objects, we use NtQuerySystemInformation to enumerate the objects' kernel addresses
  - SystemHandleInformation
- Before creating any holes (using NtClose), we ensure that we control the surrounding chunks
  - Avoid coalescing or corruption of uncontrolled chunks



# Kernel Pool Manipulation

- If delayed frees are used (most systems), we can create holes for every second allocation
  - The vulnerable buffer is later allocated in one of these holes
- Freeing the remaining allocations after triggering the vulnerability mounts the PoolIndex attack

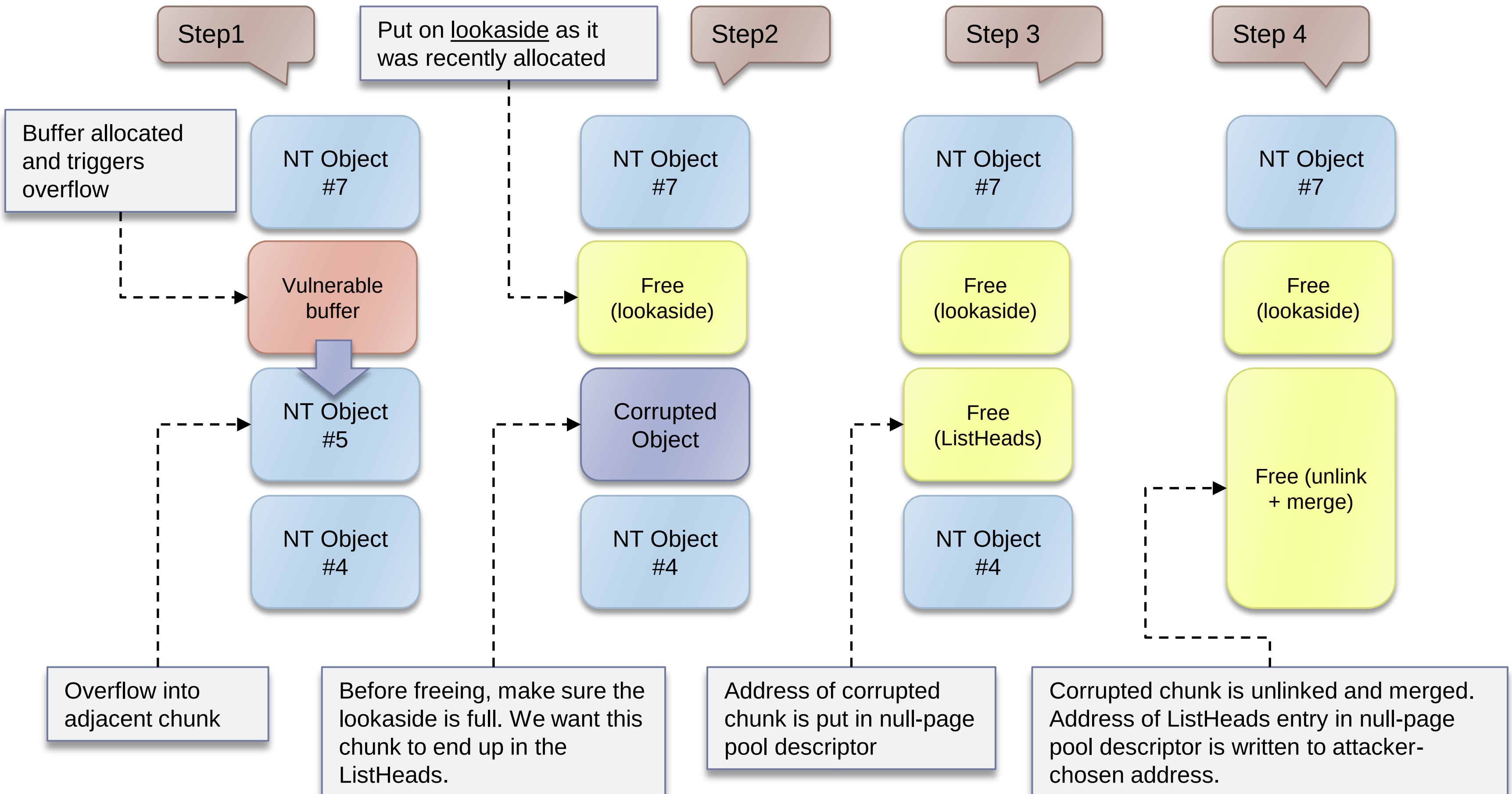
```
kd> !pool @eax
 Pool page 976e34c8 region is Nonpaged pool

 976e32e0 size: 60 previous size: 60 (Allocated) IoCo (Protected)
 976e3340 size: 60 previous size: 60 (Free) IoCo
 976e33a0 size: 60 previous size: 60 (Allocated) IoCo (Protected)
 976e3400 size: 60 previous size: 60 (Free) IoCo
 976e3460 size: 60 previous size: 60 (Allocated) IoCo (Protected)
*976e34c0 size: 60 previous size: 60 (Allocated) *Ipas
        Pooltag Ipas : IP Buffers for Address Sort, Binary : tcpip.sys
 976e3520 size: 60 previous size: 60 (Allocated) IoCo (Protected)
 976e3580 size: 60 previous size: 60 (Free) IoCo
 976e35e0 size: 60 previous size: 60 (Allocated) IoCo (Protected)
 976e3640 size: 60 previous size: 60 (Free) IoCo
```

# Coalescing for Fun and Profit

- If delayed frees are not used, we end up writing a kernel pointer to an arbitrary location
  - The address of the corrupted pool chunk
- We use the coalescing trick to write a pointer back to our null-page descriptor instead
  - Trigger an unlink of the chunk that was linked into our crafted pool descriptor
- Requires three sequentially allocated objects
  - One for our vulnerable buffer to fall into (after free)
  - One that will be corrupted
  - One that will be merged with the corrupted chunk

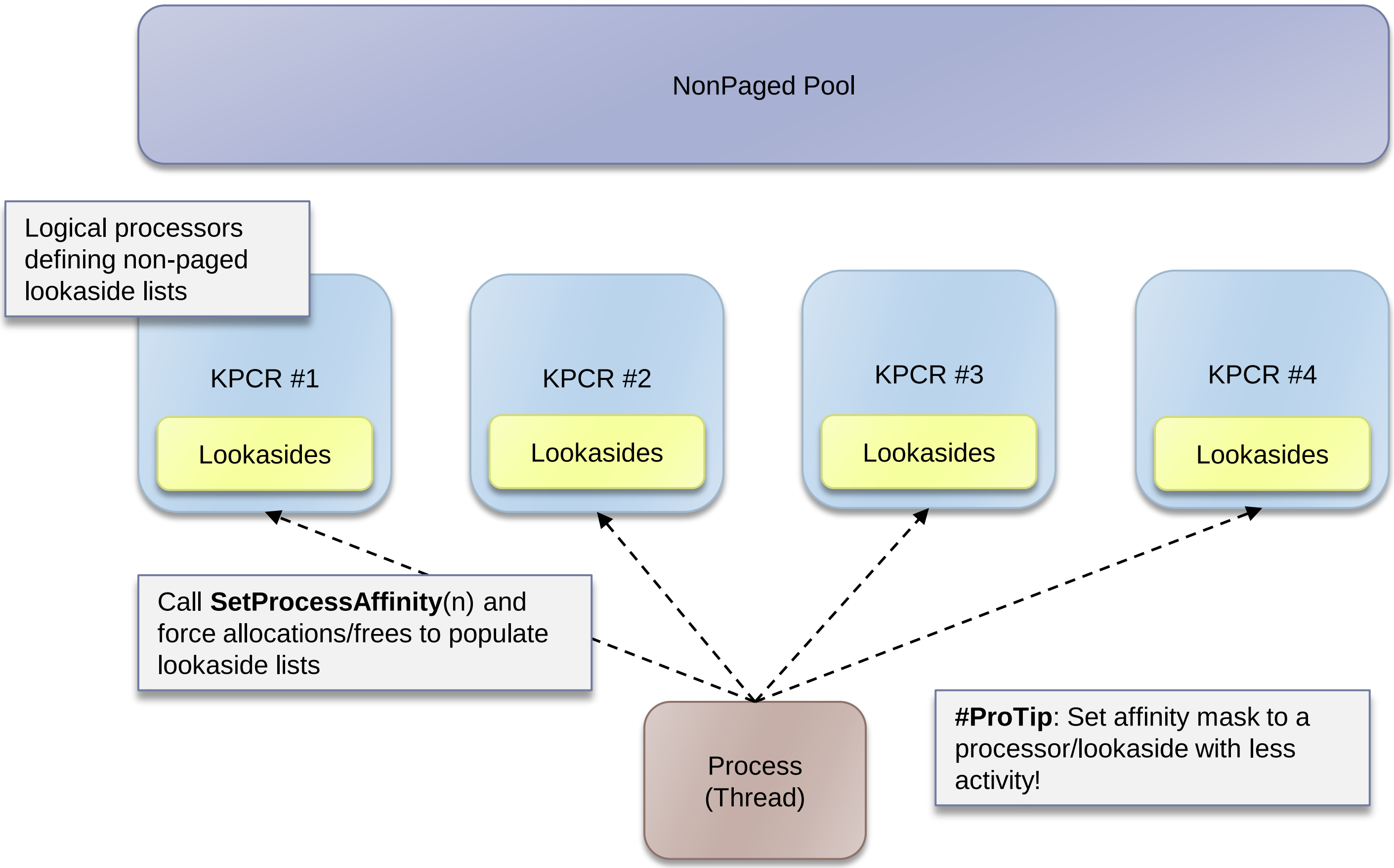
# Coalescing for Fun and Profit



# Addressing Multi-Core Systems

- On multi-core systems, multiple cores/threads can be operating on the same pool
  - E.g. only one non-paged pool
- We can reduce operations on free lists by populating the lookasides of each logical processor
  - SetProcessAffinityMask() / SetThreadAffinityMask()
- Lookasides are periodically sized according to their activity by the *balance set manager*
  - Determined by allocate/free hits and misses
  - Increasing the size can reduce the chance of other threads interfering with the pool manipulation

# Populating Lookaside Lists



# Lookaside List Information

- Can be obtained via NtQuerySystemInformation() using SystemLookasideInformation
  - Returns information on all the lookaside lists
  - Can be used to measure lookaside list activity
- Each entry is represented as a SYSTEM_LOOKASIDE_INFORMATION structure
  - Ordered by (logical) processor

```
typedef struct _SYSTEM_LOOKASIDE_INFORMATION
{
    USHORT CurrentDepth;
    USHORT MaximumDepth;
    ULONG TotalAllocates;
    ULONG AllocateMisses;
    ULONG TotalFrees;
    ULONG FreeMisses;
    ULONG Type;
    ULONG Tag;
    ULONG Size;
} SYSTEM_LOOKASIDE_INFORMATION, *PSYSTEM_LOOKASIDE_INFORMATION;
```

```
[*] Usermode pool address: 0x1b0000
[*] Lookaside 1 - Allocs: 2721 Depth: 3/4
[*] Lookaside 2 - Allocs: 2015 Depth: 0/4
[*] Lookaside 3 - Allocs: 2879 Depth: 0/4
[*] Lookaside 4 - Allocs: 15036 Depth: 21/54
[*] Filling lookasides...
[*] Lookaside 1 - Allocs: 2977 Depth: 4/4
[*] Lookaside 2 - Allocs: 2271 Depth: 4/4
[*] Lookaside 3 - Allocs: 3135 Depth: 4/4
[*] Lookaside 4 - Allocs: 15292 Depth: 54/54
```

# Possible Reliability Issues (1)

- 1. Corrupted chunk is freed to a lookaside, thus breaking the PoolIndex attack
  - Even if we fill the lookaside, there may still be preempted threads that allocate from it
- Can be addressed by maximizing the depth of the list while waiting for the balance set manager to reduce its limit
  - The lookaside list will have more entries than it can hold
  - Lookasides could also be avoided altogether by using a larger block size

# Possible Reliability Issues (2)

- 2. Buffer we overflow from uses a pool chunk not freed by us
  - Could happen if unanticipated frees were made to the lookaside list while filling
  - Less likely to happen on multi-core systems as we have multiple lookaside lists for each block size
  - Exploit reliability may improve with additional cores!

# Possible Reliability Issues (3)

- 3. Buffer we overflow from (after free) is reallocated by a different process and coalesced with the corrupted chunk
  - Triggers an unlink referencing the null-page (not mapped)
- Can be addressed by overflowing from the end of a page into a new page
  - Requires two sequentially allocated objects on the beginning of the next page

# Page Boundary Pool Allocation

- We can improve reliability by only creating holes at the end of a pool page

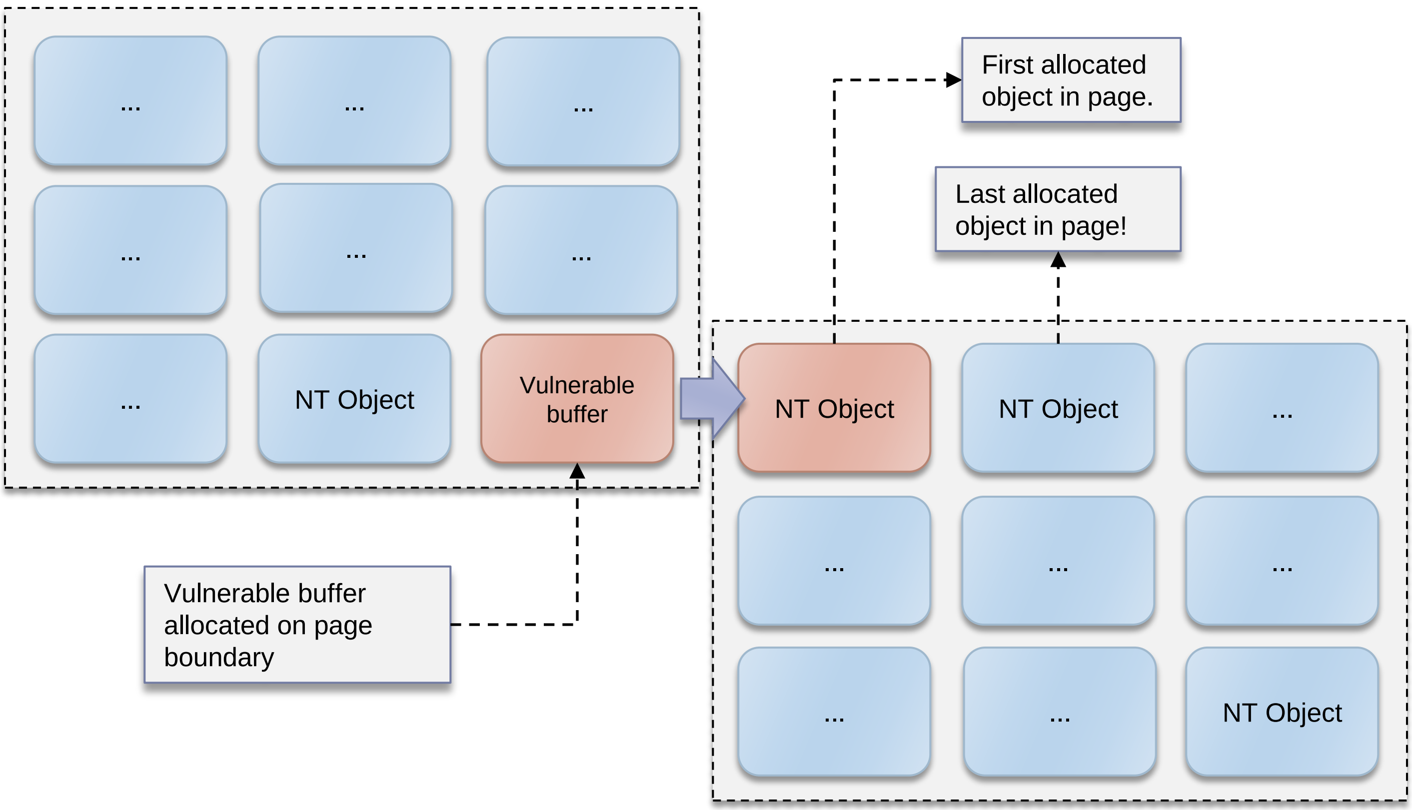
```
kd> !pool @eax
Pool page 8b518fc8 region is Nonpaged pool
 8b518000 size:   40 previous size:    0  (Allocated)  Even (Protected)
 8b518040 size:   40 previous size:   40  (Allocated)  Even (Protected)
 …
 8b518f00 size:   40 previous size:   40  (Allocated)  Even (Protected)
 8b518f40 size:   40 previous size:   40  (Allocated)  Even (Protected)
 8b518f80 size:   40 previous size:   40  (Allocated)  Even (Protected)
*8b518fc0 size:   40 previous size:   40  (Allocated) *Ipas
                    Pooltag Ipas : IP Buffers for Address Sort, Binary : tcpip.sys

 8b519000 size:   40 previous size:    0  (Allocated)  Even (Protected)
 8b519040 size:   40 previous size:   40  (Allocated)  Even (Protected)
 8b519080 size:   40 previous size:   40  (Allocated)  Even (Protected)
 8b5190c0 size:   40 previous size:   40  (Allocated)  Even (Protected)
```

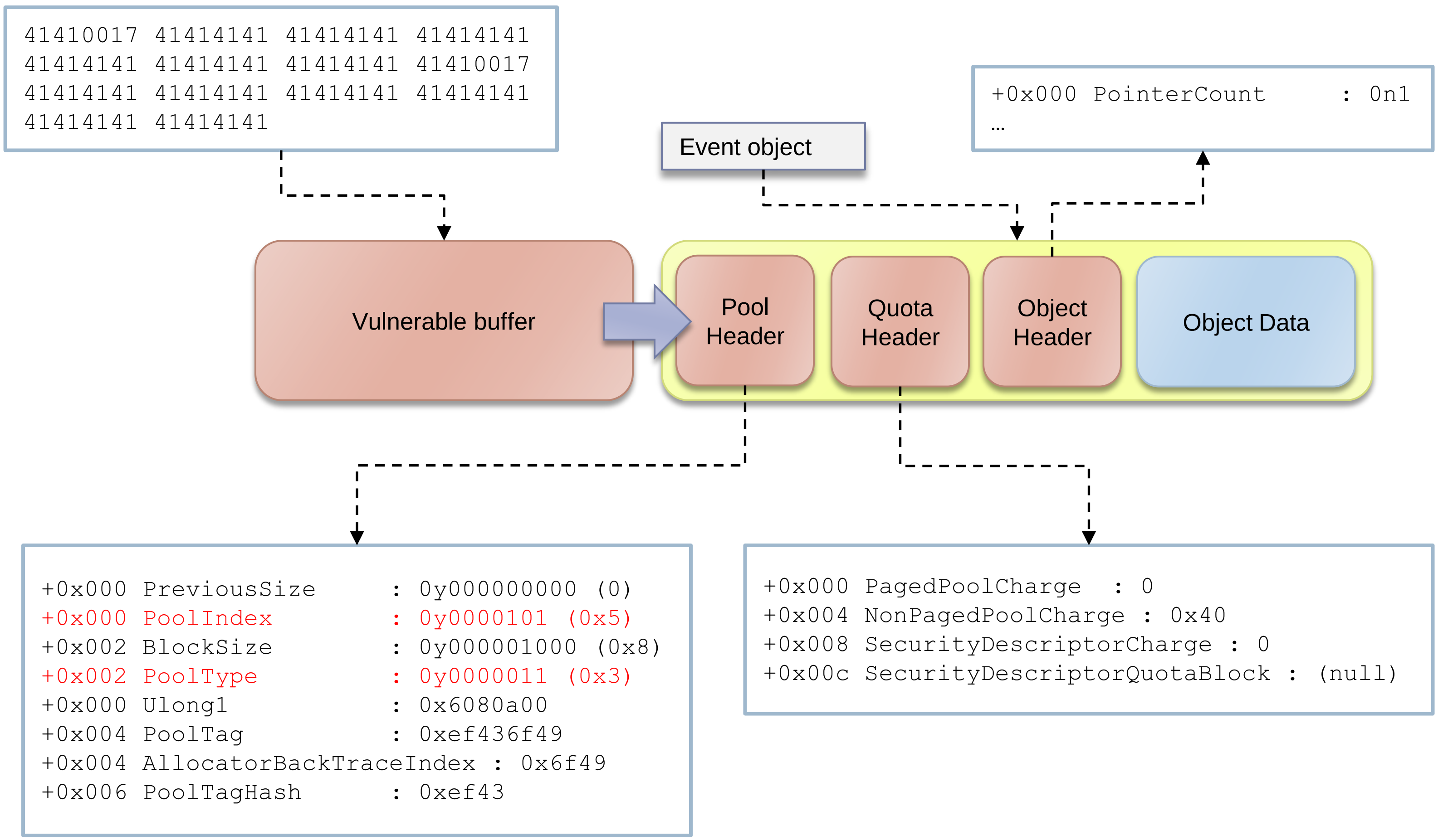
Next page

Does not merge with the previous chunk

# Page Boundary Pool Overflow



# Pool Corruption Details



# CVE-2010-1893 (MS10-058)

- Kernel pool manipulation + PoolIndex overwrite
  - Demo

# Kernel Pool Hardening

Modern Kernel Pool Exploitation:
Attacks and Techniques

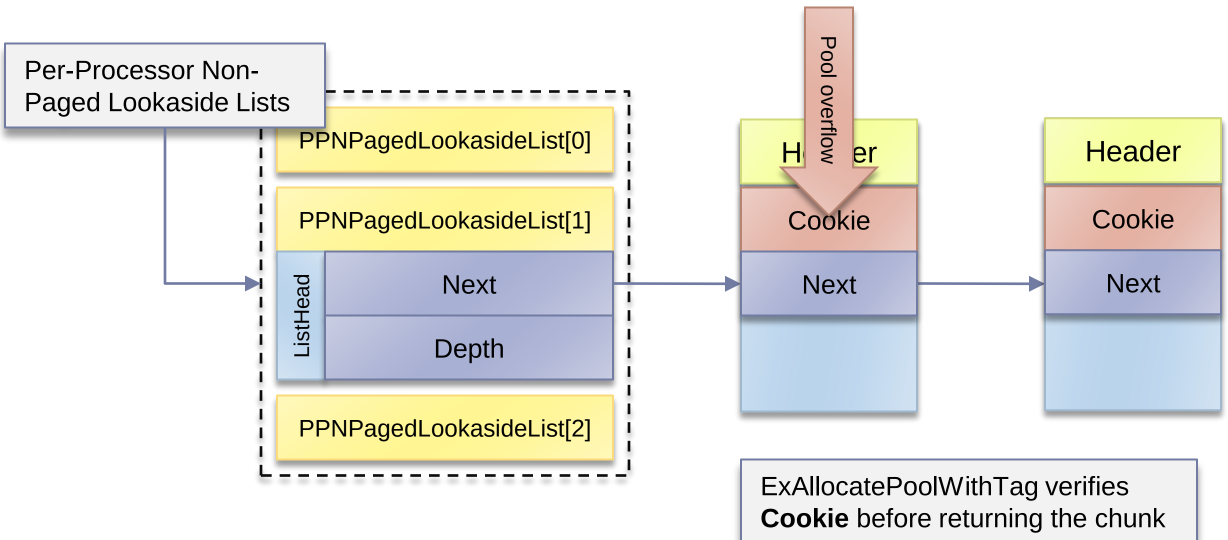
# ListEntry Flink Overwrites

- Can be addressed by properly validating the flink and blink of the chunk being unlinked
  - Yep, that's it...

# Lookaside Pointer Overwrites

- Lookaside lists are inherently insecure
  - Unchecked embedded pointers
- All pool chunks must reserve space for at least the size of a LIST_ENTRY structure
  - Two pointers (flink and blink)
- Chunks on lookaside lists only store a single pointer
  - Could include a cookie for protecting against pool overflows
- Cookies could also be used by PendingFrees list entries

# Lookaside Pool Chunk Cookie



# PoolIndex Overwrites

- Can be addressed by validating the PoolIndex value before freeing a pool chunk
  - E.g. is PoolIndex > nt!ExpNumberOfPagedPools ?
- Also required the NULL-page to be mapped
  - Could deny mapping of this address in non-privileged processes
  - Would probably break some applications (e.g. 16-bit WOW support)

# Quota Process Pointer Overwrites

- Can be addressed by encoding or obfuscating the process pointer
  - E.g. XOR'ed with a constant unknown to the attacker
- Ideally, no pointers should be embedded in pool chunks
  - Pointers to structures that are written to can easily be leveraged to corrupt arbitrary memory

# Conclusion

Modern Kernel Pool Exploitation:
Attacks and Techniques

# Future Work

- Pool content corruption
  - Object function pointers
  - Data structures
- Remote kernel pool exploitation
  - Very situation based
  - Kernel pool manipulation is hard
  - Attacks that rely on null page mapping are infeasible
- Kernel pool manipulation
  - Becomes more important as generic vectors are addressed

# Conclusion

- The kernel pool was designed to be fast
  - E.g. no pool header obfuscation
- In spite of safe unlinking, there is still a big window of opportunity in attacking pool metadata
  - Kernel pool manipulation is the key to success
- Attacks can be addressed by adding simple checks or adopting exploit prevention features from the userland heap
  - Header integrity checks
  - Pointer encoding
  - Cookies

# References


- **SoBeIt[2005]** – SoBeIt
  How to exploit Windows kernel memory pool,
  X'con 2005
- **Kortchinsky[2008]** – Kostya Kortchinsky
  Real-World Kernel Pool Exploitation,
  SyScan 2008 Hong Kong
- **Mxatone[2008]** – mxatone
  Analyzing Local Privilege Escalations in win32k,
  Uninformed Journal, vol. 10 article 2
- **Beck[2009]** – Peter Beck
  Safe Unlinking in the Kernel Pool,
  Microsoft Security Research & Defense (blog)

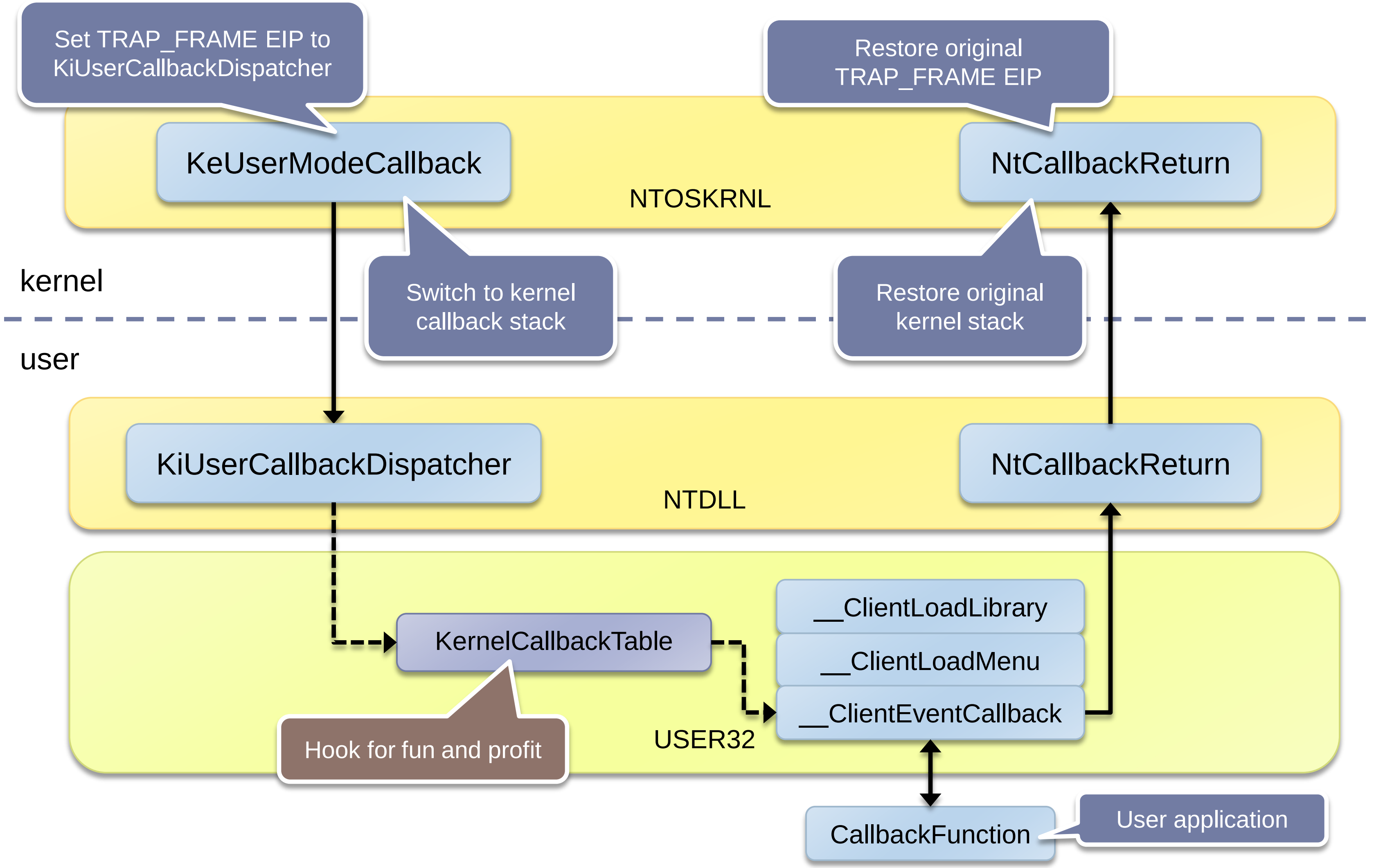
# MS11-034

Modern Kernel Pool Exploitation:
Attacks and Techniques

# Overview

- All the vulnerabilities addressed by this bulletin were related to user-mode callbacks
  - Locking issues
  - Null pointer dereferences
- Invoking user-mode callbacks
  - Event hooks (SetWinEventHook)
  - Window hooks (SetWindowsHook)
  - Some functions call back into user-mode regardless of hooks
- Pointer to callback function table stored in the PEB
  - Peb()->KernelCallbackTable
  - Hook this to do whatever during callbacks

# nt!KeUserModeCallback



# Use After Free Vulnerabilities

- All Window Manager (USER) objects are preceded by a HEAD structure
  - Defines handle value and lock count
- Whenever a callback occurs, objects subsequently used has to be locked
  - E.g. if a window is insufficiently locked, a user could call DestroyWindow to free it
- Similarly, any buffer that can be reallocated or freed (e.g. an array used by an object) has to be checked upon callback return
  - E.g. menu items array

# Ex #1: Window Object Use-After-Free

- Microsoft previously patched two vulnerabilities in win32k!xxxCreateWindowEx
  - Window Creation Vulnerability (MS10-032)
  - Function Callback Vulnerability (MS10-048)
- Both issues dealt with improper validation of changes occurring during callbacks
- None of the patches ensured that the window object returned by the CBT hook was properly locked
- Hence, an attacker could destroy the window object (in a subsequent callback) and coerce the kernel into operating on freed memory

# Ex #2: Cursor Object Use-After-Free

- In using a drag cursor while dragging an object, win32k!xxxDragObject did not lock the original cursor
- An attacker could destroy the original cursor in a user-mode callback such as an event hook
- Consequently, the kernel would operate on freed memory upon restoring the original cursor

# Exploitability

- In most cases, the attacker can allocate and control the bytes that are freed
  - E.g. using APIs that allocate strings
- Embedded object pointers in the freed object may allow an attacker to increment (lock) or decrement (unlock) an arbitrary address
  - Common behavior of locking routines
- Some targets
  - KTHREAD.PreviousMode
    - kernel trusts argument pointers when PreviousMode == 0
  - HANDLEENTRY.bType
    - destroy routine for free type (0) is null (mappable by user)

# Questions ?

- Email: kernelpool@gmail.com
- Blog: http://mista.nu/blog
- Slides/Paper: http://mista.nu/research
- Twitter: @kernelpool