NORMAN®

Kernel Attacks through User-Mode Callbacks

Tarjei Mandt

Black Hat USA 2011

Who am I

- **Security Researcher at Norman**
  - Malware Detection Team (MDT)
- **Interests**
  - Vulnerability research
  - Operating system internals
- **Past Work**
  - Kernel Pool Exploitation on Windows 7
  - Mitigating NULL Pointer Exploitation on Windows

About this Talk

- **Several vulnerability classes related to windows hooks and user-mode callbacks**
  - Null pointer dereferences
  - Use-after-frees
- **Resulted in 44 patched privilege escalation vulnerabilities in MS11-034 and MS11-054**
  - Several unannounced vulnerabilities were also addressed as part of the variant discovery process
- **Requires understanding of several mechanisms specific to NT and win32k**

Agenda

- **Introduction**
- **Win32k**
  - Window Manager
  - User-Mode Callbacks
- **Vulnerabilities**
- **Exploitability**
- **Mitigations**
- **Conclusion**

Introduction

Win32k and User-Mode Callbacks

Win32k

- **The Windows GUI subsystem was traditionally implemented in user-mode**
  - Used a client-server process model
- **In NT 4.0, a large part of the server component (in CSRSS) was moved to kernel-mode**
  - Introduced Win32k.sys
- **Today, Win32k manages both the Window Manager (USER) and the Graphics Device Interface (GDI)**

User-Mode Callbacks

- **Allows win32k to make calls back into user-mode and operate on user-mode data**
  - Invoke application defined hooks
  - Provide event notifications
  - Read and set properties in user-mode structures
- **Implemented in the NT executive**
  - nt!KeUserModeCallback
  - Works like a reverse system call

NORMAN

Win32k vs. User-Mode Callbacks

- **Win32k uses a global locking design in creating a thread-safe environment**
  - Presumably remnants of the old subsystem design
- **Callbacks "interrupt" kernel execution and allow win32k structures and object properties to be modified**
- **Insufficient checks or validation may result in numerous vulnerabilities**
  - Use-after-frees
  - NULL pointer dereferences
  - ++

NORMAN

Previous Work

- **Mxatone - Analyzing local privilege escalations in win32k (Uninformed vol.10)**
  - Insufficient validation of data returned from user-mode callbacks
- **Win32k Window Creation Vulnerabilities**
  - CVE-2010-0484 (MS10-032)
    - Window parent not revalidated after callbacks
  - CVE-2010-1897 (MS10-048)
    - Pseudo handle provided in callback not sufficiently validated
- **Stefan Esser - State of the Art Post Exploitation in Hardened PHP Environments (BlackHat USA 2009)**
  - Interruption vulnerabilities

Goals

- Show how user-mode callbacks without very stringent checks may introduce several subtle vulnerabilities
- Show how such vulnerabilities may be exploited using pool and kernel heap manipulation
- Propose a method to generically mitigate exploitability of NULL pointer dereference vulnerabilities

NORMAN

Win32k

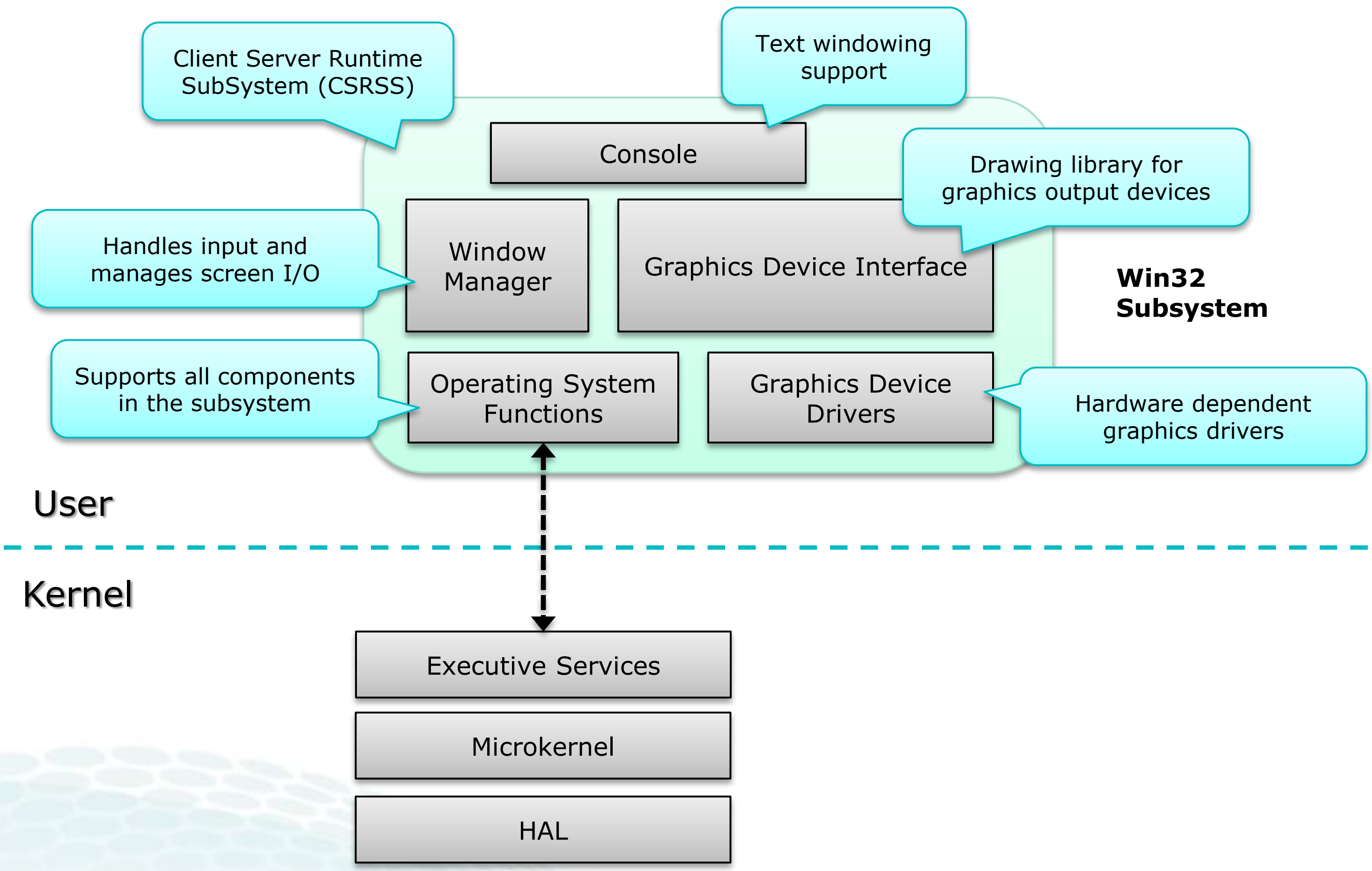
Architecture and Design

Windows NT 3.51

- **Modified microkernel design**
  - File systems, network protocols, IPC, and drivers are implemented in kernel mode
- **Followed a more pure microkernel approach in its implementation of the GUI subsystem**
  - Window Manager and GDI implemented in the Client-Server Runtime SubSystem (CSRSS)
- **Optimized for performance**
  - Shared memory design
  - Paired threads between client and server (FastLPC)

NORMAN

Windows NT 3.51 Win32 Subsystem



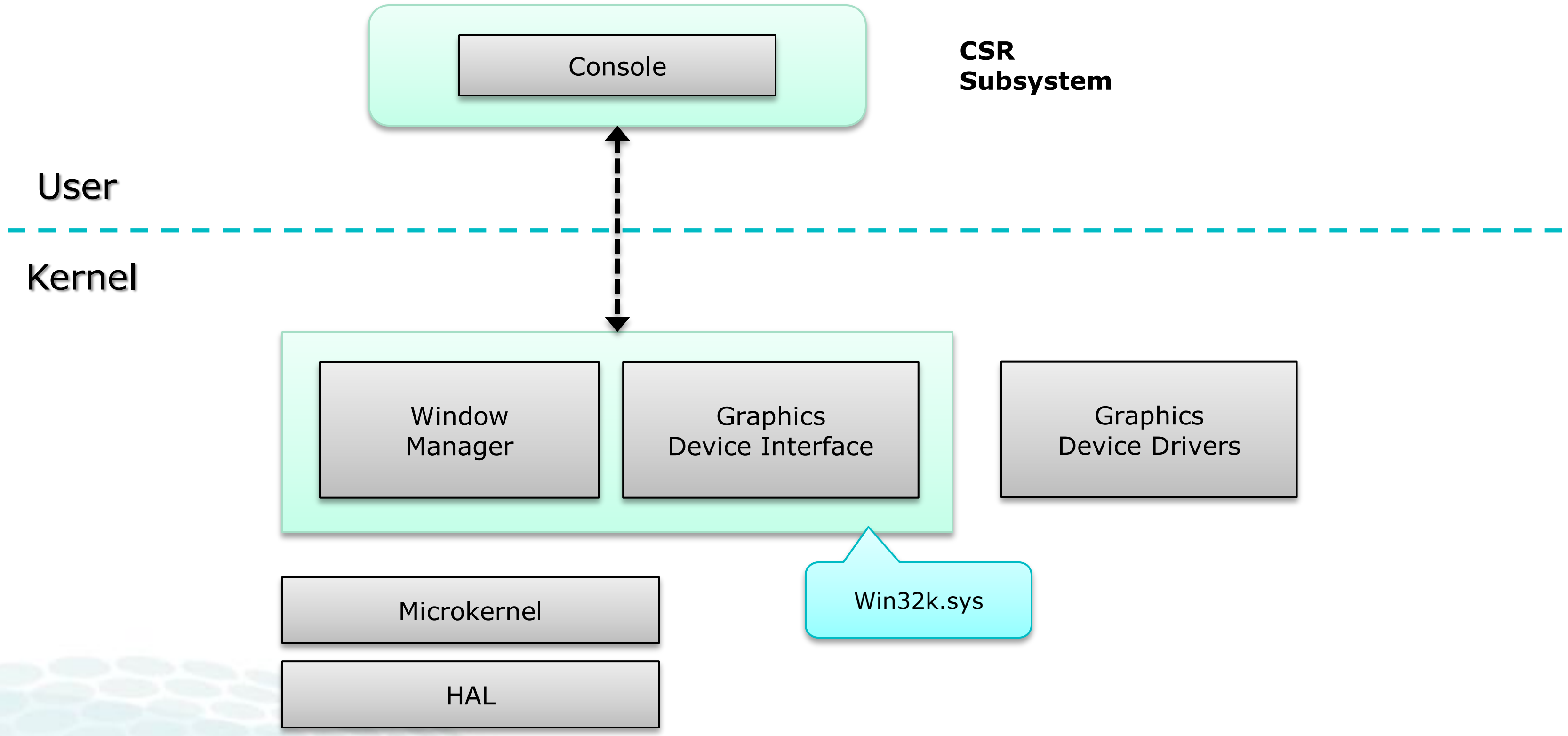
Drawbacks of the NT 3.51 Design

- **Graphics and windowing subsystem have a very high rate of interaction with hardware**
  - Video drivers, mouse, keyboard, etc.
- **Client-server interaction involves excessive thread and context switching**
  - Greatly affects graphics rendering performance
- **High memory requirements**
  - Uses 64K shared memory buffer to accumulate and pass parameters between the client and server

Windows NT 4.0

- **Moved the Window Manager, GDI and graphics device drivers to kernel-mode**
  - Introduced win32k.sys
- **Eliminated the need for shared buffers and paired threads**
  - Results in fewer thread and context switches
  - Reduces memory requirements
- **Some old performance tricks were still maintained**
  - E.g. caching of management structures in the user mode portion of the client's address space

Win32k.sys in Windows NT 4.0



Win32k

- **Kernel component of the Win32 subsystem**
- **Implements the kernel side of**
  - Window Manager (USER)
  - Graphics Device Interface (GDI)
- **Provides thunks to DirectX interfaces**
- **Has it’s own system call table**
  - More than 800 entries on Windows 7
  - win32k!W32pServiceTable

Window Manager (USER)

- **Several responsibilities**
  - Controls window displays
  - Manages screen output
  - Collects input from keyboard, mouse, etc.
  - Calls application-defined hooks
  - Passes user messages to applications
  - Manages user objects
- **The component this talk will focus on**

NORMAN

Graphics Device Interface (GDI)

- **Manages the graphics output and rendering**
  - Library of functions for graphics output devices
  - Includes functions for line, text, and figure drawing and for graphics manipulation
  - Manages GDI objects such as brushes, pens, DCs, paths, regions, etc.
  - Provides APIs for video/print drivers
- **Slow compared to Direct2D/DirectWrite**
  - Will probably be replaced at some point

NORMAN

DirectX Thunks

- **Entry point thunks for DirectX support**
  - NtGdiDd* or NtGdiDDI*
- **Calls corresponding functions in the DirectX driver**
  - dxg.sys (XDDM) or dxgkrnl.sys (WDDM) depending on the display driver model used
- **Display drivers hook DXG interfaces to hardware accelerate or punt back to GDI**

Window Manager

User Objects and Thread Safety

User Objects

- **All user handles for entities such as windows and cursors are backed by their own object**
  - Allocated in win32k!HMAllocateObject
- **Each object type is defined by a unique structure**
  - win32k!tagWND
  - win32k!tagCURSOR
- **User objects are indexed into a dedicated handle table maintained by win32k**
- **Handle values are translated into object pointers using the handle manager validation APIs**
  - win32k!HMValidateHandle(..)

User Object Header

- **Every user object starts with a HEAD structure**
- **kd> dt win32k!_HEAD**
  - +0x000 h : Ptr32 Void // handle value
  - +0x004 cLockObj : Uint4B // lock count
- **The lock count tracks object use**
  - An object is freed when the lock count reaches zero
- **Additional fields are defined if the object is owned by a thread or process, or associated with a desktop**
  - win32k!_THRDESKHEAD
  - win32k!_PROCDESKHEAD

User Handle Table

- **All user objects are indexed into a per-session handle table**
  - Initialized in win32k!Win32UserInitialize
- **Pointer to the user handle table is stored in the win32k!tagSHAREDINFO structure**
  - user32!gSharedInfo (Win 7) or win32k!gSharedInfo
- **kd> dt win32k!tagSHAREDINFO**
  - +0x000 psi : Ptr32 tagSERVERINFO
  - **+0x004 aheList : Ptr32 _HANDLEENTRY**
  - +0x008 HeEntrySize : Uint4B
  - +0x00c pDispInfo : Ptr32 tagDISPLAYINFO
  - +0x010 ulSharedDelta : Uint4B

NORMAN

User Handle Table Entries

- **Each entry in the user handle table is represented by a HANDLEENTRY structure**
- **kd> dt win32k!_HANDLEENTRY**
  - +0x000 phead : Ptr32 _HEAD
  - +0x004 pOwner : Ptr32 Void
  - +0x008 bType : Uchar
  - +0x009 bFlags : Uchar
  - +0x00a wUniq : Uint2B
- **Holds pointers to the object, its owner, type, flags, and a unique seed for the handle values**
  - handle = handle_table_index | (wUniq << 0x10)
  - *wUniq* is incremented on object free

User Handle Table Entries

object	owner	bType	bFlags	wUniq
0	0	0	0	0
ff9d1d28	0	c	0	1
ffbbd498	ffb09678	1	40	1
ffb658f0	ffbbc958	3	0	1
ff650618	ffb09678	1	0	1
ffb64918	ffbbc958	3	0	1

Pointer to object in kernel memory

Pointer to owner (THREADINFO or PROCESSINFO)

Object type (e.g. window, cursor, menu, etc.)

Object flags (e.g. being destroyed)

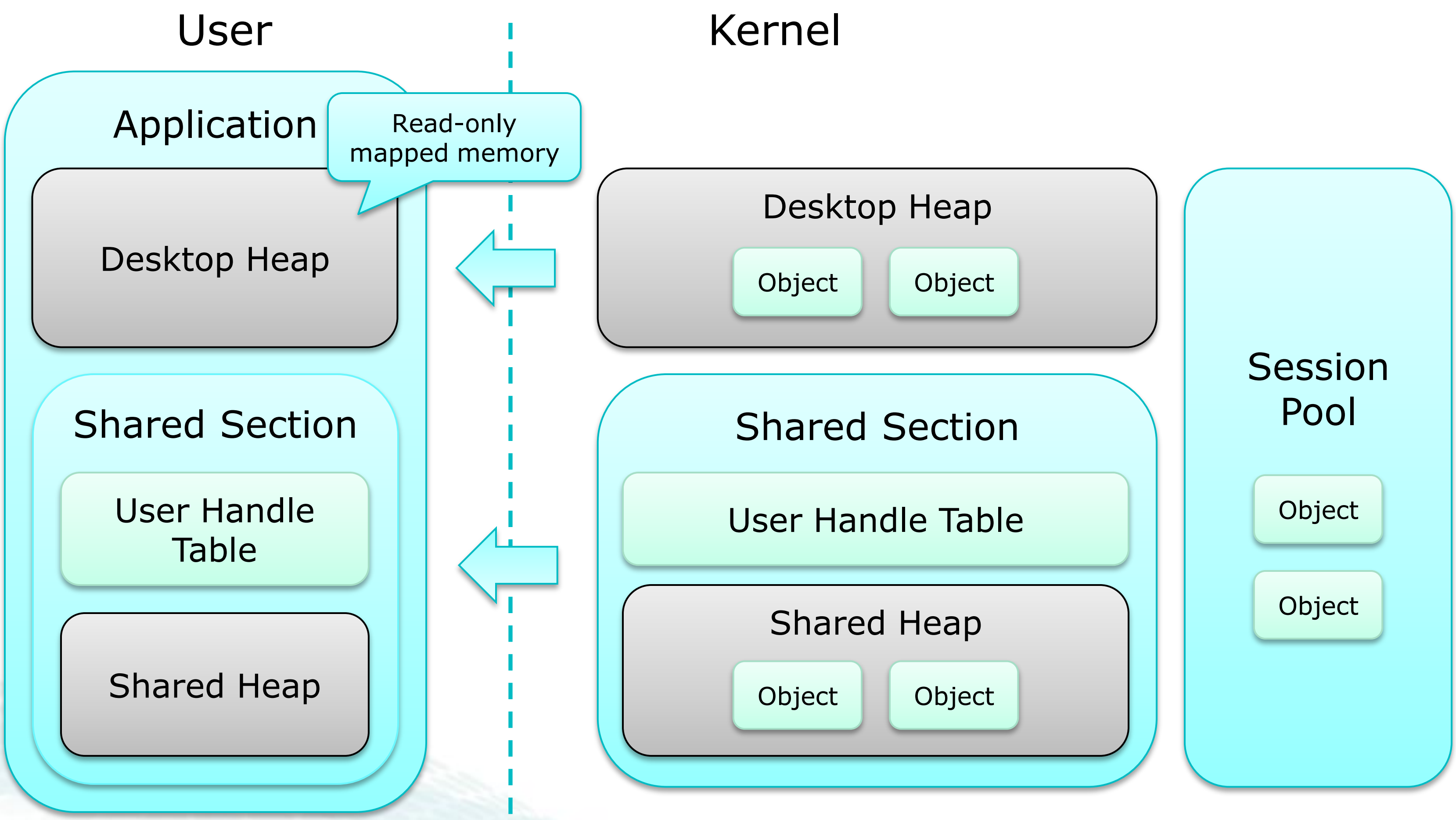
Unique counter

NORMAN

User Objects In Memory

- **User objects are stored in the *session pool*, the *desktop heap* or the *shared heap***
  - Set in the handle type information table (win32k!gahti)
- **The desktop heap and shared heap are read-only mapped into user address space**
  - Used to avoid kernel transitions
- **Objects associated with a particular desktop are stored on the desktop heap**
- **Remaining objects are stored in the shared heap or the session pool**

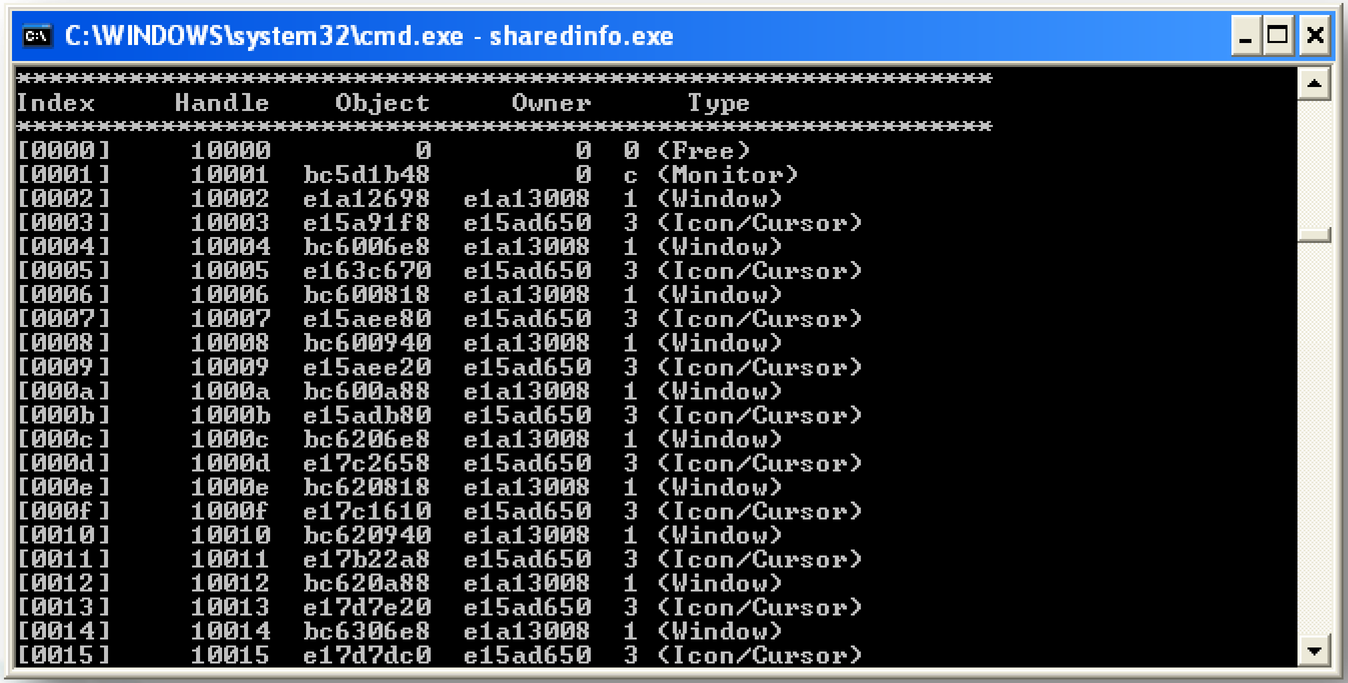
Handle Table & Objects In Memory



Shared Section User Mapping

- **The shared section is mapped into a GUI process upon initializing the client Win32 subsystem**
  - Essentially means loading user32.dll
  - Mapping itself is performed by CSRSS in calling NtUserProcessConnect (InitMapSharedSection)
- **The user handle table, at the base of the shared section, can be obtained in at least two ways**
  - From user32!gSharedInfo (exported on Windows 7)
  - From the connection information buffer returned by **CsrClientConnectToServer** upon specifying USERSRV_SEVERDLL_INDEX (3)

Handle Table From User-Mode



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

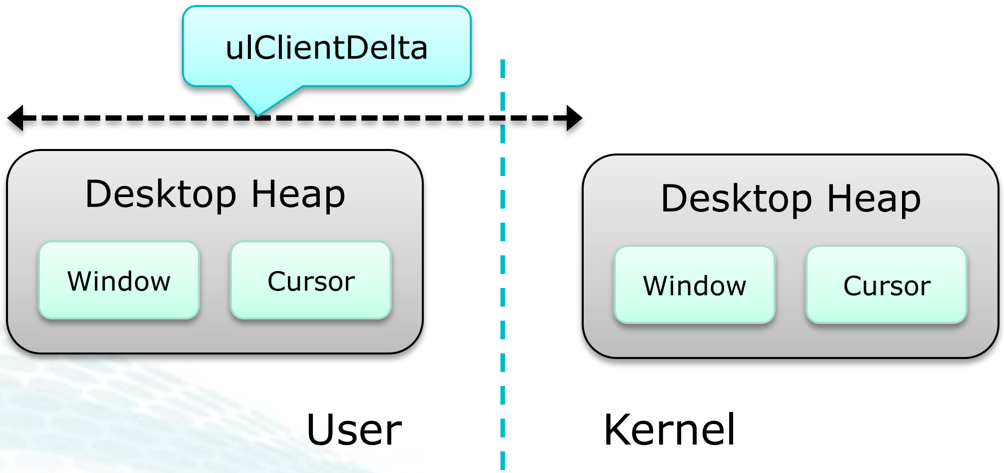
```
***********************************************************************
Index      Handle     Object      Owner       Type
***********************************************************************
[0000]      10000          0           0   0 (Free)
[0001]      10001   bc5d1b48           0   c (Monitor)
[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)
```

Desktop Heap User Mapping

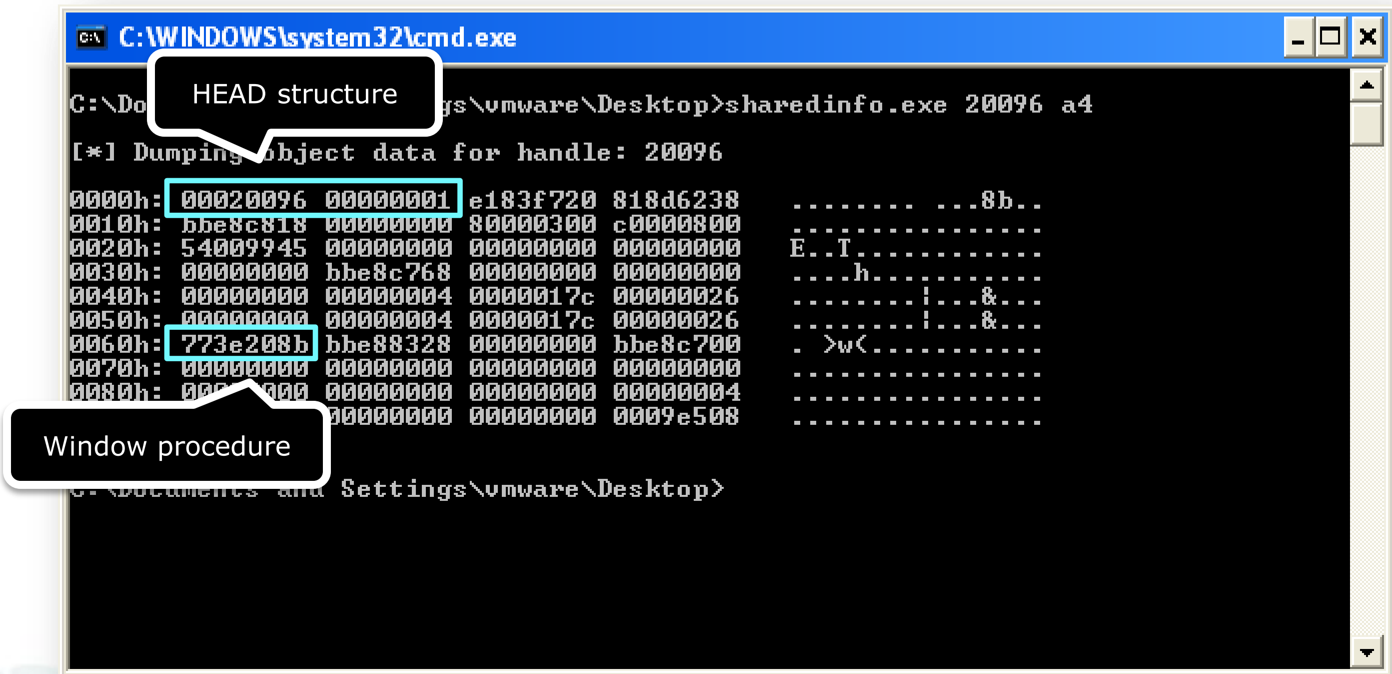
- **For each GUI thread, win32k maps the associated desktop heap into the user-mode process**
  - Performed by win32k!MapDesktop
- **Information on the desktop heap is stored in the desktop information structure**
  - Holds the kernel address of the desktop heap
  - Accessible from user-mode
    - NtCurrentTeb()->Win32ClientInfo.pDeskInfo
- **kd> dt win32k!tagDESKTOPINFO**
  - +0x000 pvDesktopBase : Ptr32 Void
  - +0x004 pvDestkopLimit : Ptr32 Void

Kernel-Mode -> User-Mode Address

- **User-space address of desktop heap objects are computed using ulClientDelta**
  - NtCurrentTeb()->Win32ClientInfo.ulClientDelta
- **User-space address of shared heap objects are computed using ulSharedDelta**
  - Defined in win32k!tagSHAREDINFO



User Object From User-Mode



User Object Types

- **On Windows 7, there are 21 different user object types (22 including the ‘free’ type)**
  - Includes ‘touch’ and ‘gesture’ objects
- **Information on each type is stored in the handle type information table**
  - win32k!ghti (undocumented structure)
  - Defines the destroy routines for each type
  - Defines target memory location (desktop/shared heap, session pool)

User Object Types #1

ID	TYPE	OWNER	MEMORY
0	Free		
1	Window	Thread	Desktop Heap / Session Pool *
2	Menu	Process	Desktop Heap
3	Cursor	Process	Session Pool
4	SetWindowPos	Thread	Session Pool
5	Hook	Thread	Desktop Heap
6	Clipboard Data		Session Pool
7	CallProcData	Process	Desktop Heap
8	Accelerator	Process	Session Pool
9	DDE Access	Thread	Session Pool
10	DDE Conversation	Thread	Session Pool

* Stored on the desktop heap if the window is associated with a desktop

NORMAN

User Object Types #2

ID	TYPE	OWNER	MEMORY
11	DDE Transaction	Thread	Session Pool
12	Monitor		Shared Heap
13	Keyboard Layout		Session Pool
14	Keyboard File		Session Pool
15	Event Hook	Thread	Session Pool
16	Timer		Session Pool
17	Input Context	Thread	Desktop Heap
18	Hid Data	Thread	Session Pool
19	Device Info		Session Pool
20 (Win 7)	Touch	Thread	Session Pool
21 (Win 7)	Gesture	Thread	Session Pool

User Critical Section

- **Unlike NT, the Window Manager does not exclusively lock each user object**
  - Implements a global lock per session
- **Each kernel routine that operates on win32k structures or objects must first acquire a lock on win32k!gpresUser**
  - Exclusive lock used if write operations are involved
  - Otherwise, shared lock is used
- **Clearly not designed to be multithreaded**
  - E.g. two separate applications in the same session cannot process their message queues simultaneously

NORMAN

Shared and Exclusive Locks

```
; Attributes: bp-based frame

; __stdcall NtUserCheckDesktopByThreadId(x)
_NtUserCheckDesktopByThreadId@4 proc near

var_4= dword ptr -4
arg_0= dword ptr  8

mov     edi, edi
push    ebp
mov     ebp, esp
push    ecx
push    esi
call    _EnterSharedCrit@0 ; EnterSharedCrit()
mov     esi, eax
call    ds:__imp__PsGetCurrentProcess@0 ; PsGetCurrentProc
push    eax
call    _IsProcessDwm@4 ; IsProcessDwm(x)
test    eax, eax
jnz     short loc_BF844222
```

Acquire shared lock

```
; Attributes: bp-based frame

; int __stdcall NtUserSwitchDesktop(HANDLE Handle, int)
_NtUserSwitchDesktop@8 proc near

var_10= byte ptr -10h
var_4= dword ptr -4
Handle= dword ptr  8
arg_4= dword ptr  0Ch

mov     edi, edi
push    ebp
mov     ebp, esp
sub     esp, 10h
push    esi
call    _UserEnterUserCritSec@0 ; UserEnterUserCritSec()
mov     eax, _gptiCurrent
xor     esi, esi
test    dword ptr [eax+0
jz      short loc_BF8198
```

Acquire exclusive lock

NORMAN

User-Mode Callbacks

Kernel to User Interaction

User-Mode Callbacks

- **In interacting with user-mode data, win32k is required to make calls back into user-mode**
  - Lead to the concept of user-mode callbacks
- **Implemented in nt!KeUserModeCallback**
  - Works like a reverse system call
  - Previously researched by Ivanlef0u and mxatone, among others
- **Used extensively in user object handling**
  - Some user objects store data in user-mode

NORMAN

KeUserModeCallback

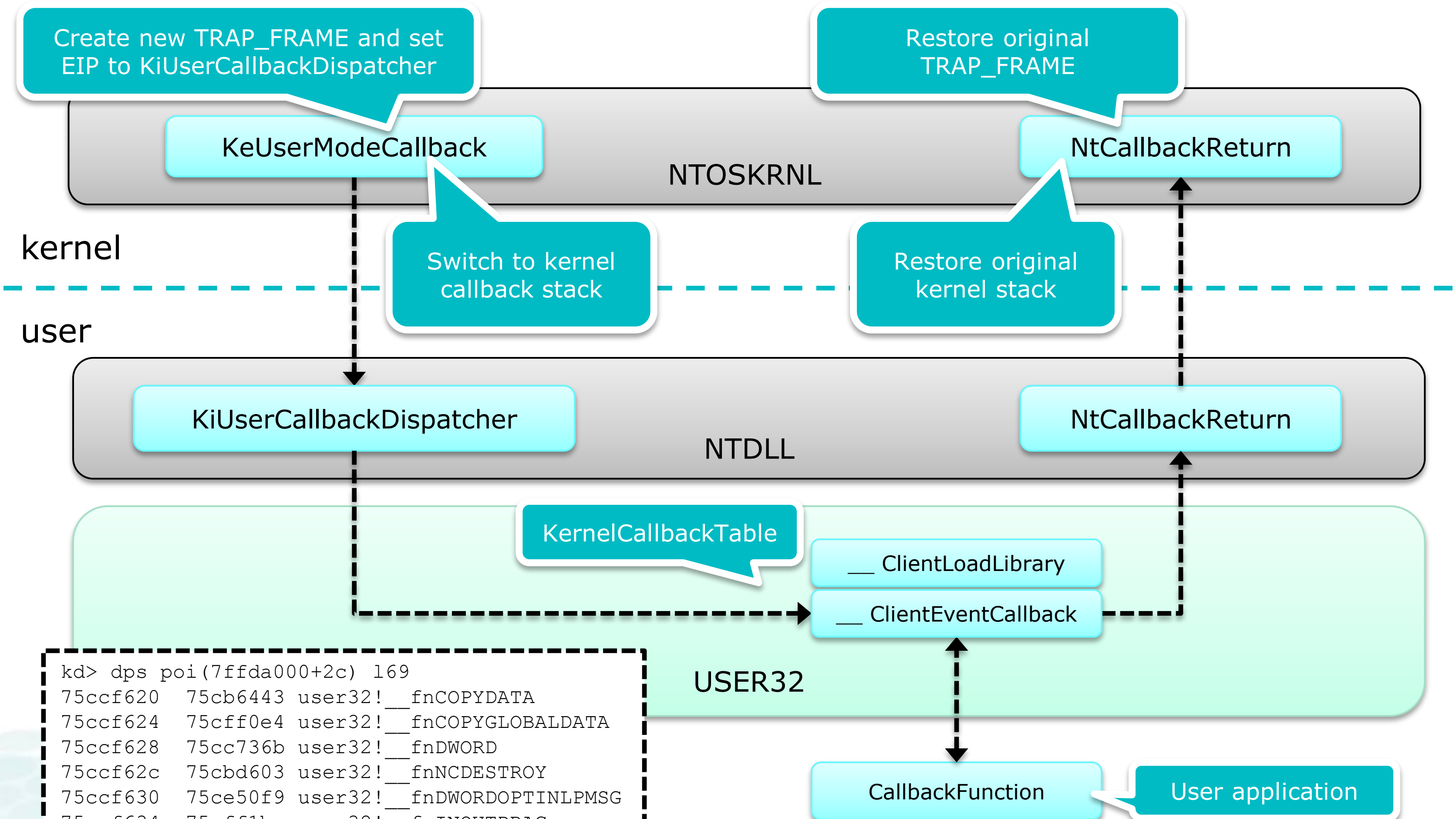
- **NTSTATUS KeUserModeCallback (**
  IN ULONG ApiNumber,
  IN PVOID InputBuffer,
  IN ULONG InputLength,
  OUT PVOID * OutputBuffer,
  IN PULONG OutputLength **);**
- ***ApiNumber* is an index into the user-mode callback function table**
  - Copied to the Process Environment Block (PEB) during the initialization of USER32.dll in a given process
- **kd> dt nt!_PEB KernelCallbackTable**
  - +0x02c KernelCallbackTable : Ptr32 Void

NORMAN

KeUserModeCallback Internals

- **In a system call, a trap frame is stored on the kernel thread stack by KiSystemService or KiFastCallEntry**
  - Used to save thread context and restore registers upon returning to user-mode
- **KeUserModeCallback creates a new trap frame (KTRAP_FRAME) before invoking KiServiceExit**
  - Sets EIP to ntdll!KiUserCallbackDispatcher
  - Replaces TrapFrame pointer of the current thread
- **Input buffer is copied to the user-mode stack**

KeUserModeCallback




```
kd> dps poi(7ffda000+2c) l69
75ccf620  75cb6443 user32!__fnCOPYDATA
75ccf624  75cff0e4 user32!__fnCOPYGLOBALDATA
75ccf628  75cc736b user32!__fnDWORD
75ccf62c  75cbd603 user32!__fnNCDESTROY
75ccf630  75ce50f9 user32!__fnDWORDOPTINLPMSG
75ccf634  75cff1be user32!__fnINOUTDRAG
75ccf638  75ce6cd0 user32!__fnGETTEXTLENGTHS
75ccf63c  75cff412 user32!__fnINCNTOUTSTRING
```

NORMAN

Kernel Callback Stack

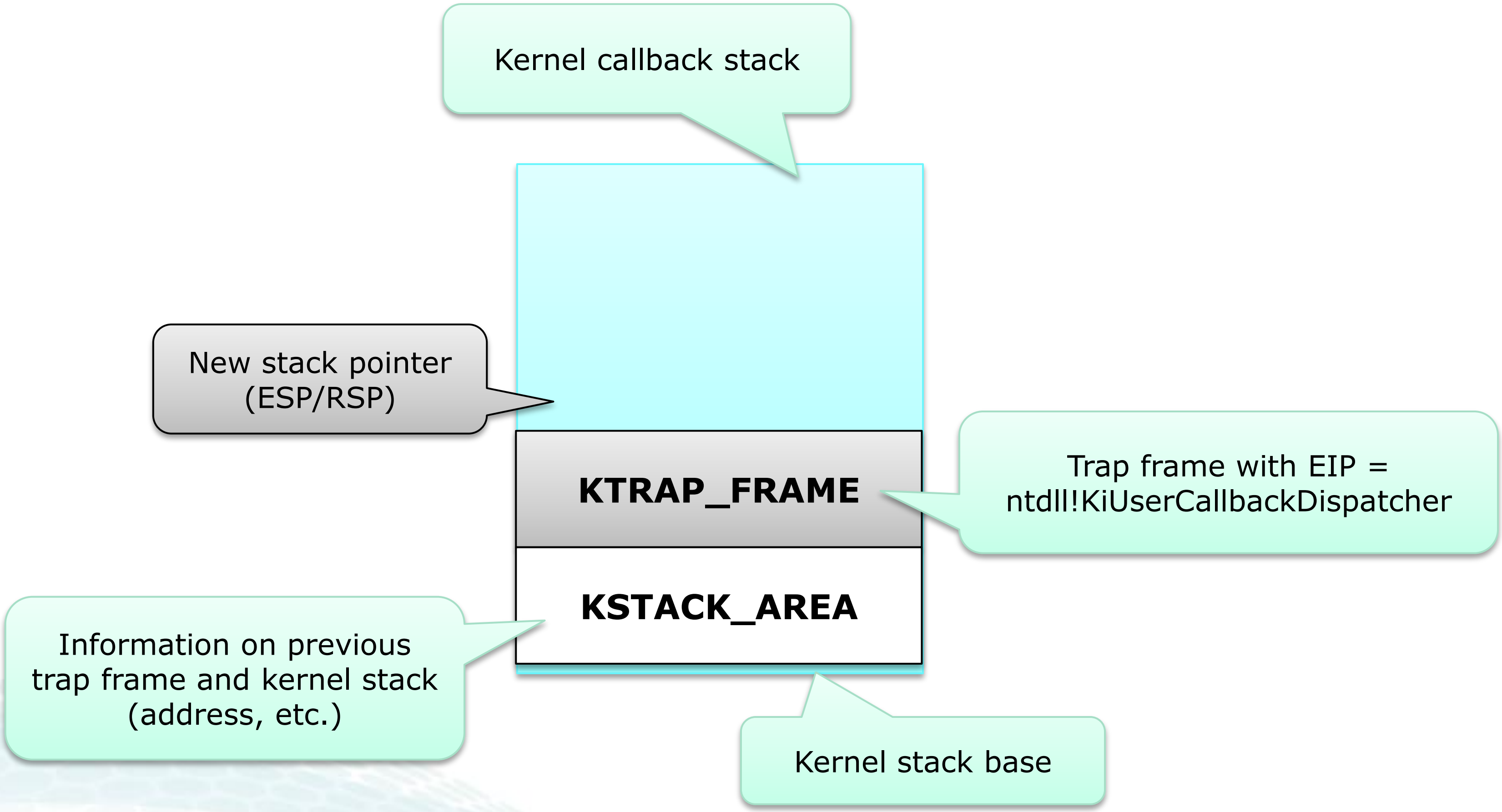
- **On Vista/Windows 7, the kernel creates a new kernel thread stack for use during the user-mode callback**
  - Windows XP would simply grow the existing stack
- **The new trap frame is stored on the new kernel stack**
- **Information on the previous kernel stack is stored in a KSTACK_AREA structure**
  - Stored at the base of every kernel thread stack

```
kd> dt nt!_KSTACK_AREA
   +0x000 FnArea              : _FNSAVE_FORMAT
   +0x000 NpxFrame            : _FXSAVE_FORMAT
   +0x1e0 StackControl        : _KERNEL_STACK_CONTROL
   +0x1fc Cr0NpxState         : Uint4B
   +0x200 Padding             : [4] Uint4B
```

```
kd> dt nt!_KERNEL_STACK_CONTROL -b
   +0x000 PreviousTrapFrame : Ptr32
   +0x000 PreviousExceptionList : Ptr32
   +0x004 StackControlFlags : Uint4B
   +0x004 PreviousLargeStack : Pos 0, 1 Bit
   +0x004 PreviousSegmentsPresent : Pos 1, 1 Bit
   +0x004 ExpandCalloutStack : Pos 2, 1 Bit
   +0x008 Previous          : _KERNEL_STACK_SEGMENT
      +0x000 StackBase         : Uint4B
      +0x004 StackLimit        : Uint4B
      +0x008 KernelStack       : Uint4B
      +0x00c InitialStack      : Uint4B
      +0x010 ActualLimit       : Uint4B
```

NORMAN

Kernel Callback Stack Layout



NtCallbackReturn

- **NTSTATUS NtCallbackReturn (**
  IN PVOID Result OPTIONAL,
  IN ULONG ResultLength,
  IN NTSTATUS Status **);**
- **Used to resume execution in the kernel after a user-mode callback**
- **Copies the result of the callback back to the original kernel stack**
- **Restores original trap frame and kernel stack by using the information held in the KSTACK_AREA**
- **Deletes the kernel callback stack upon completion**

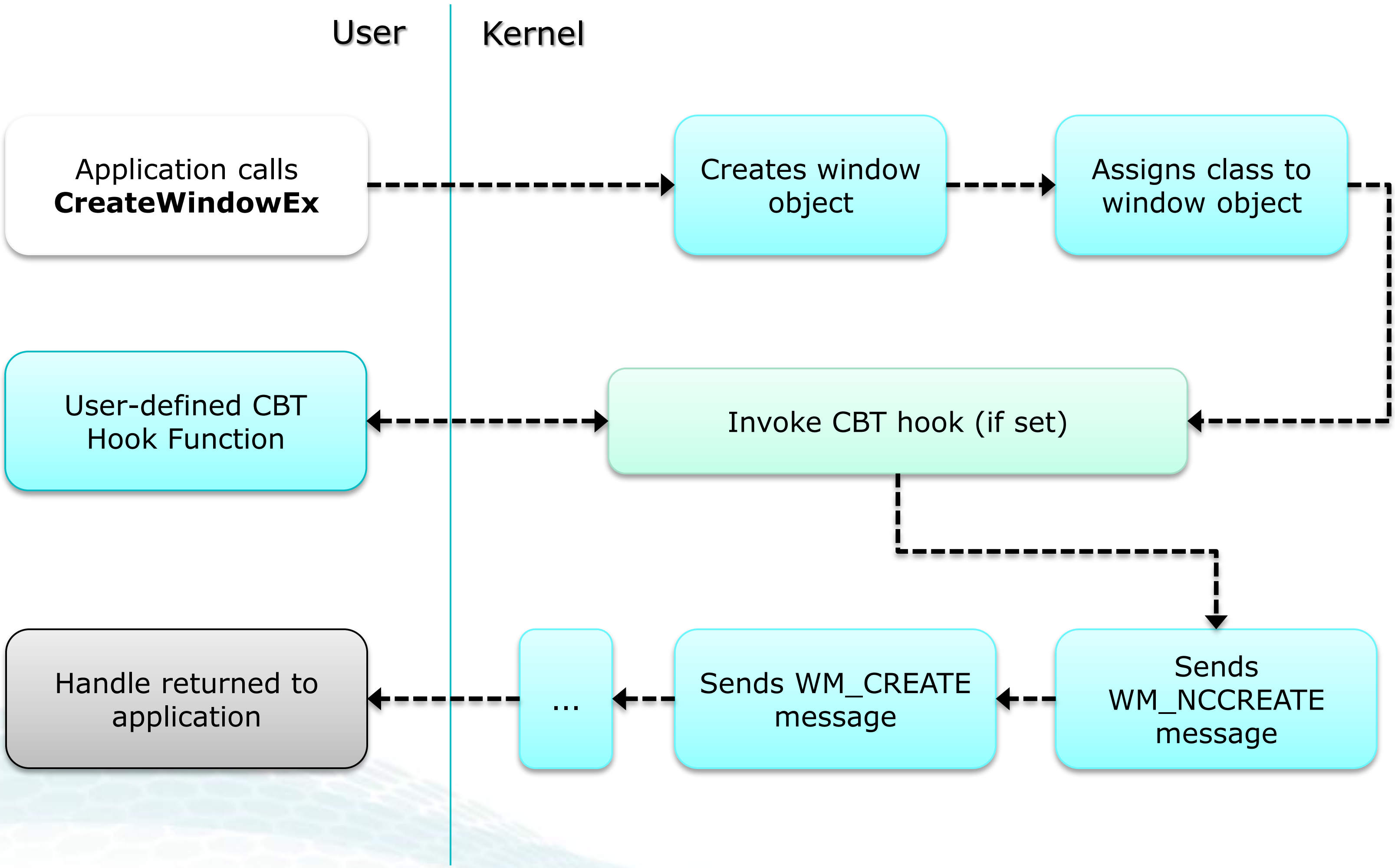
Applications of User-Mode Callbacks

- **User-mode callbacks allow win32k to perform a variety of tasks**
  - Invoke application-specific windows hooks
  - Provide event notification
  - Copy data to and from user-mode (e.g. for DDE)
- **Hooks allow users to execute code in response to certain actions performed by win32k**
  - Calling a window procedure
  - Creating or destroying
  - Processing keyboard or mouse input

Windows Hooks

- **Set using the SetWindowsHook APIs**
  - Invoked by the kernel through calls to xxxCallHook
- **Typically used to monitor certain system events and their associated paramters**
- **May alter function parameters depending on the type of hook**
  - E.g. change the z-ordering of a window in a create window hook
- **Processed synchronously**
  - The user-mode hook is called immediately at the time when the appropriate conditions are met

CreateWindow CBT Hook Example



Event Hooks

- **Set using the SetWinEventHook APIs**
  - Invoked by the kernel through calls to xxxWindowEvent
- **Used to notify a user-mode process that a certain event occured or is about to occur**
  - E.g. inform that a new window has been created
- **Can be processed both synchronously and asynchronously (deferred events)**
  - In the latter case, the kernel calls xxxFlushDeferredWindowEvents to flush the event queue

Kernel Attacks through User-Mode Callbacks

Vulnerabilities in Win32k

User Critical Section vs. Callbacks

- **Whenever a callback is executed, the kernel leaves the win32k user critical section**
  - Allows win32k to perform other tasks while user-mode code is being executed
- **Upon returning from a callback, win32k must ensure that referenced objects are still in the expected state**
  - E.g. a callback could call SetParent() to update the parent of a window
- **Insufficient checks may lead to vulnerabilities**

Function Name Decoration

- **Win32k.sys uses function name decoration to keep track of functions that leave the critical section**
  - Prefixed “xxx” and “zzz”
- **Functions prefixed “xxx” may leave the critical section and invoke a user-mode callback**
  - May sometimes require a specific argument or set of arguments to trigger the actual callback
- **Functions prefixed “zzz” typically invoke a deferred event callback**
  - However, if win32k!gdwDeferWinEvent is null, an immediate callback is performed

NORMAN

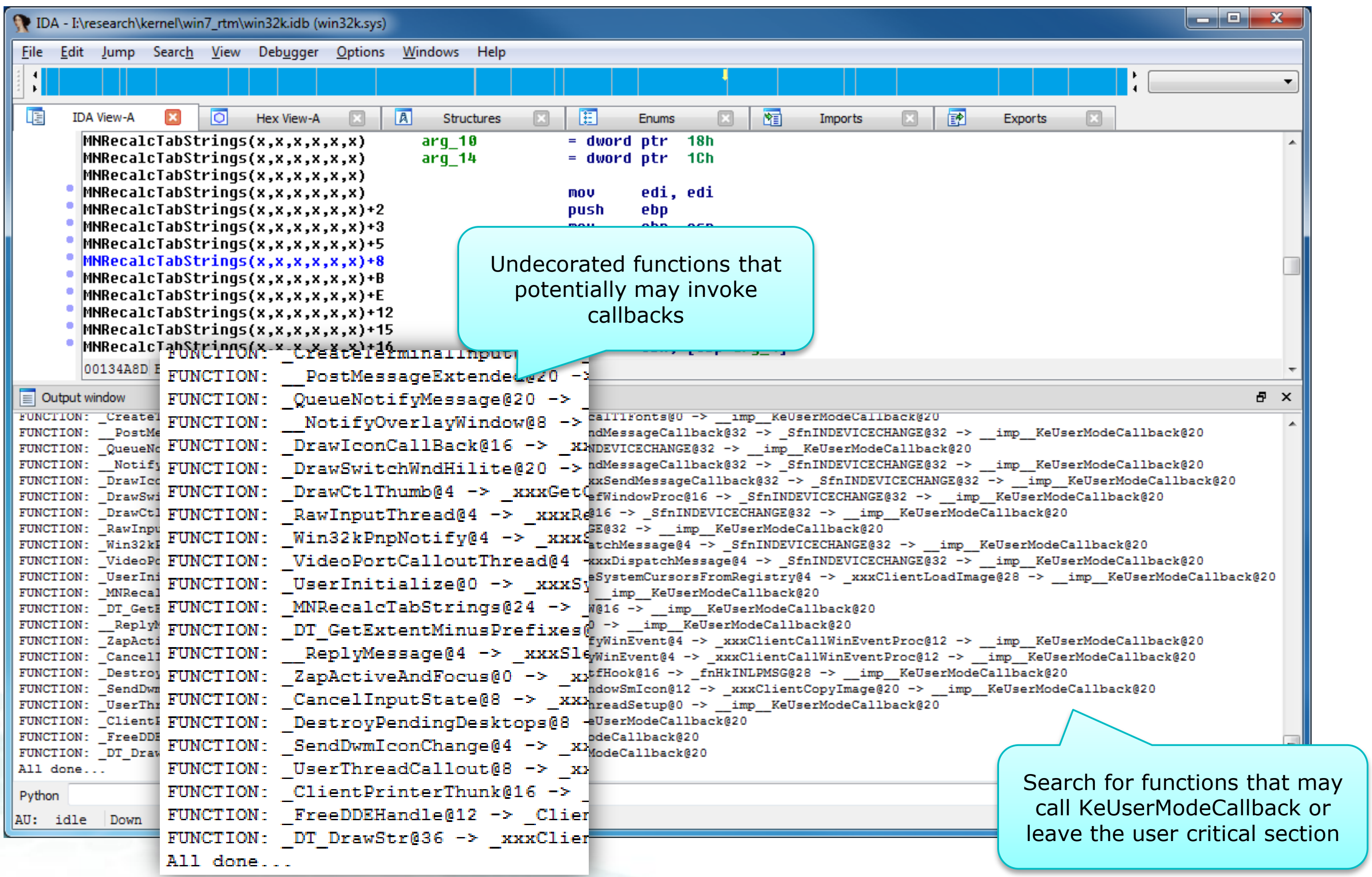
Function Name Decoration Issues

- **Functions that leave the critical section and invoke user-mode callbacks are not always prefixed**
  - Could lead to invalid assumptions by the programmer
  - Easy to spot using IDAPython and cross referencing
- **Lack of consistency in behavior of "zzz" functions**
  - Some "zzz" functions seem to increment gdwDeferWinEvent while others do not

Windows 7 RTM	Windows 7 (MS11-034)
MNRecalcTabStrings	xxxMNRecalcTabStrings
FreeDDEHandle	xxxFreeDDEHandle
ClientFreeDDEHandle	xxxClientFreeDDEHandle

NORMAN

Locating Undecorated Functions



Object Locking

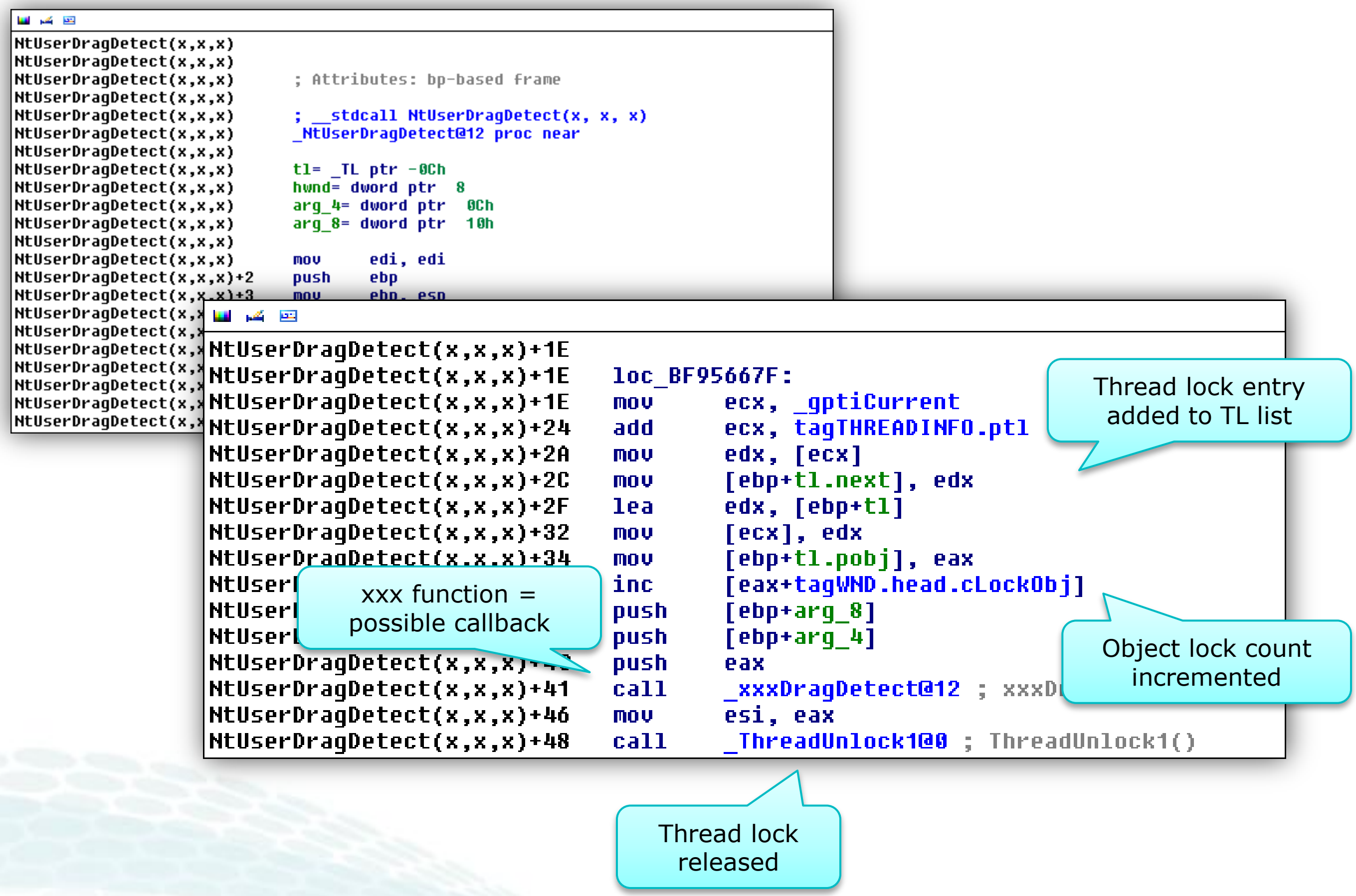
- **Objects expected to be valid after the kernel leaves the user critical section, must be *locked***
  - The cLockObj field of the common object header stores the object reference count
- **Two forms of locking**
  - Thread locking
  - Assignment locking

NORMAN

Thread Locking

- **Used to lock objects or buffers within the context of a thread**
  - ThreadLock* (inlined mostly) and ThreadUnlock*
- **Each thread locked entry is stored as a TL structure**
  - kd> dt win32k!_TL
  - +0x000 next : Ptr32 _TL
  - +0x004 pobj : Ptr32 Void
  - +0x008 pfnFree : Ptr32 Void
- **Pointer to the thread lock list is stored in the THREADINFO structure of a thread object**
- **Upon thread termination, the thread lock list is processed to release any outstanding entries**
  - xxxDestroyThreadInfo -> DestroyThreadsObjects

Thread Locking By Example



Assignment Locking

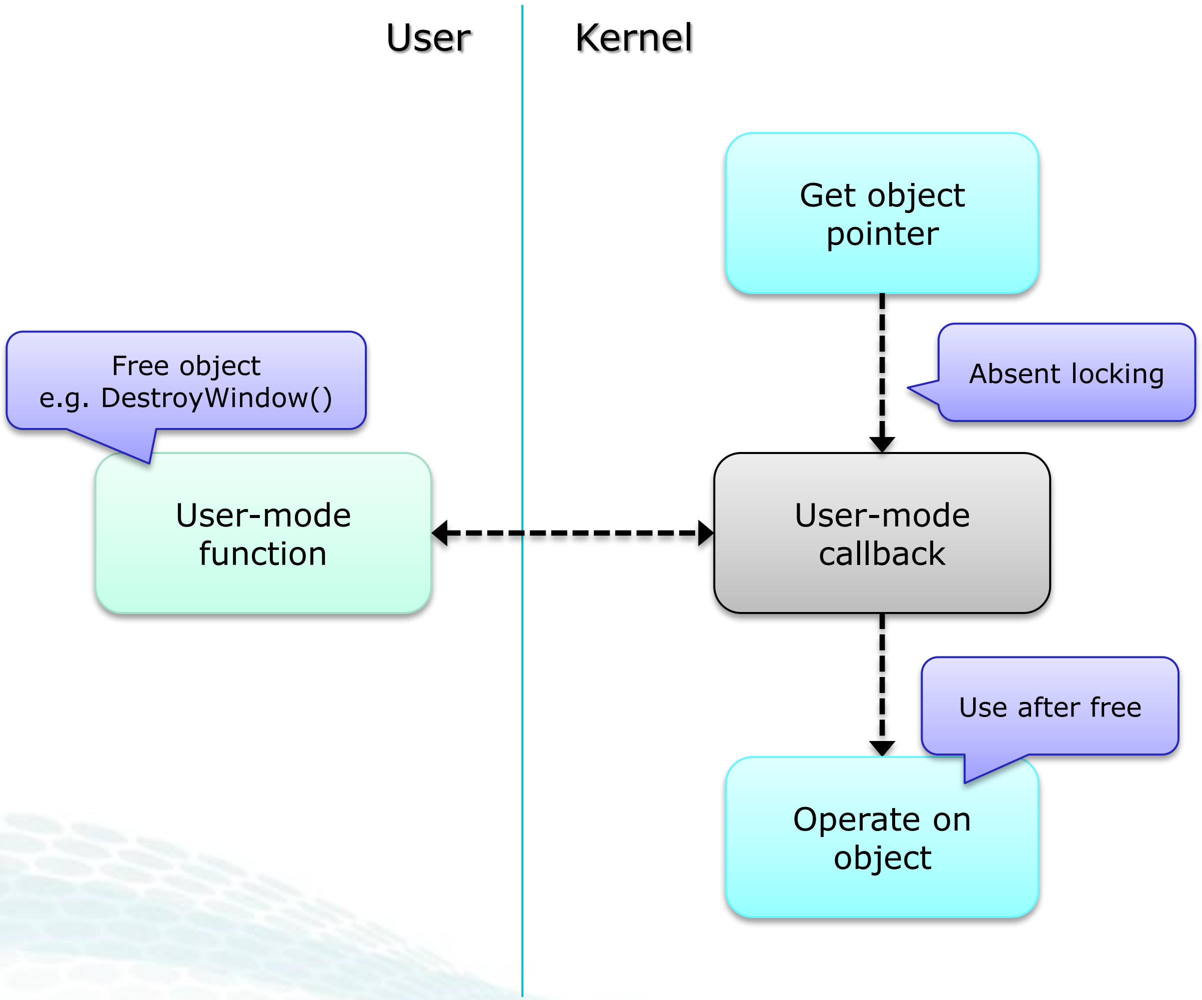
- **The handle manager provides functions for thread independent locking of objects**
  - HMAssignmentLock(Address,Object)
  - HMAssignmentUnlock(Address)
- **Assignment locking an object to an address with an initialized pointer, releases the existing reference**
- **Does not provide the safety net thread locking does**
  - E.g. if a thread termination occurs in a callback, the thread cleanup code must release these references

NORMAN

Object Locking Vulnerabilities

- **Any object expected to be valid after a user-mode callback should be locked**
- **Similarly, any object that no longer is used by a particular component should be released**
- **Mismanagement in the locking and release of objects could result in the following**
  - No retention: An object could be freed too early
  - No release: An object could never be freed, or the reference count (e.g. 32-bit on x86) could wrap

Object Use-After-Free



Window Object Use-After-Free

- **In creating a window, an application can adjust its orientation and z-order using a CBT hook**
  - Z-order is defined by providing the handle to the window after which the new window is inserted
- **win32k!xxxCreateWindowEx failed to properly lock the provided z-order window**
  - Only stored a pointer to the object in a local variable
- **An attacker could destroy the window in a subsequent user-mode callback and trigger a use-after-free**

Window Object Use-After-Free

```
lea     eax, [ebp+cstruct]
mov     [ebp+cbt.lpcs], eax
mov     [ebp+cbt.hwndInsertAfter], edi
push    5
lea     eax, [ebp+cbt]
push    eax
push    dword ptr [ebx]
push    HCBT_CREATEWND
call    _xxxCallHook@16 ;
test    eax, eax
jnz     loc_BF89365C
```

```
mov     eax, [ebp+cstruct.x]
mov     [ebp+pos_x], eax
mov     eax, [ebp+cstruct.y]
mov     [ebp+pos_y], eax
mov     eax, [ebp+cstruct._cx]
mov     [ebp+x_size], eax
mov     eax, [ebp+cstruct.cy]
mov     [ebp+y_size], eax
push    [ebp+cbt.hwndInsertAfter]
call    _PWInsertAfter@4 ; PWInsertAfter(x)
mov     [ebp+pwndInsertAfter], eax ; window object pointer
jmp     short loc_BF892EB1
```

Get object pointer from handle (cbt.hwndInsertAfter)

User-mode callback(s)

DestroyWindow(hwnd)

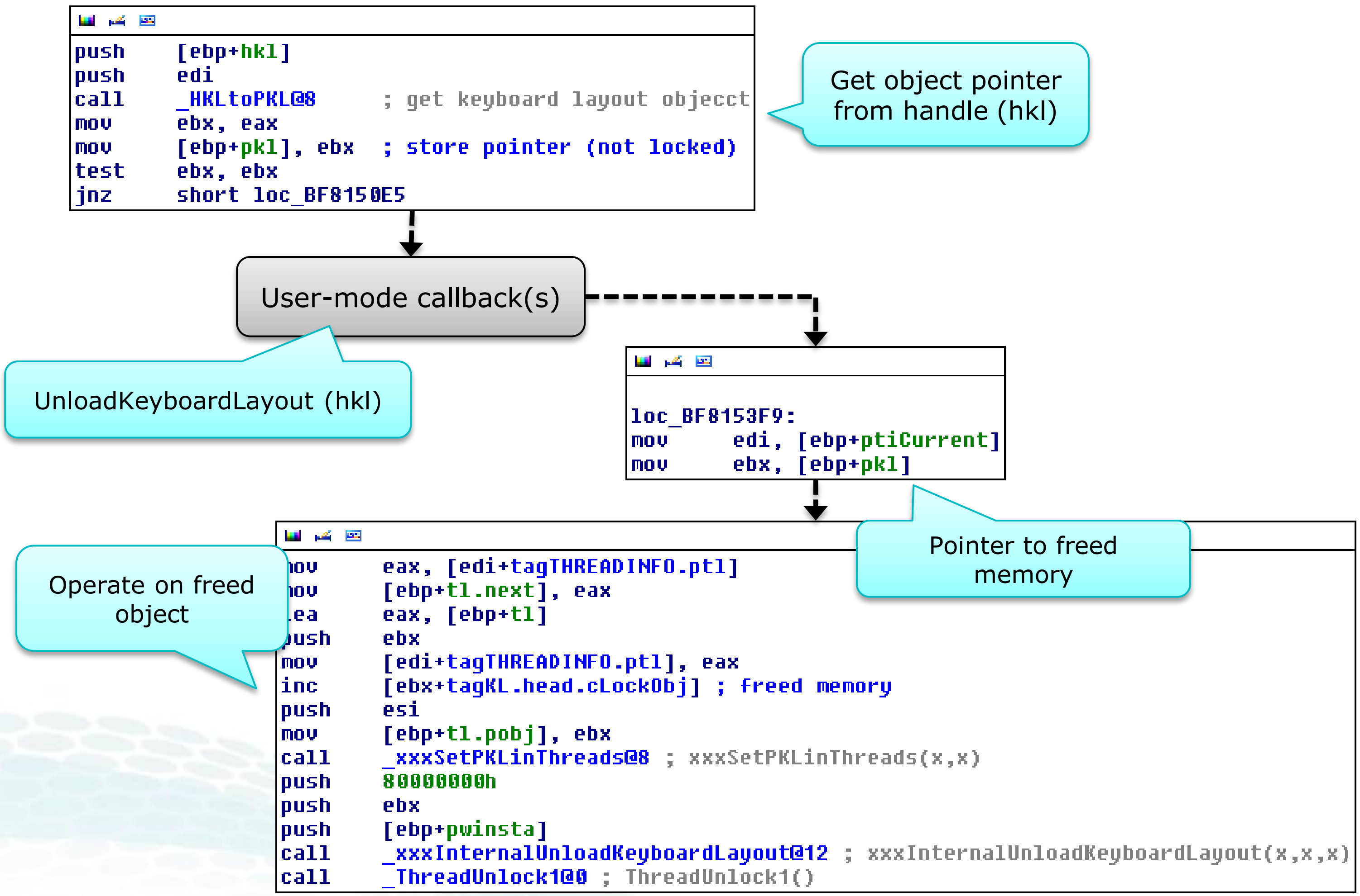
Operate on freed object

```
lo   F893924:
push    esi
push    [ebp+pwndInsertAfter]
push    ebx
call    _LinkWindow@12   ; LinkWindow(x,x,x)
push    0
push    10h
push    esi
call    _zzzLockDisplayAreaAndInvalidateDCCache@12 ; zzzLockDisplayAreaAndInvalidateDCCache(x,x,x)
mov     eax, [ebp+wndParent]
```

Keyboard Layout Object Use-After-Free

- **In loading a keyboard layout, win32k!xxxLoadKeyboardLayoutEx did not lock the keyboard layout object**
  - Pointer stored in local variable
- **An attacker could unload the keyboard layout in a user-mode callback and thus free the object**
- **Subsequently, upon using the object pointer the kernel would operate on freed memory**

Keyboard Layout Object Use-After-Free



NORMAN

Object State Validation

- **Objects assumed to be in a certain state should always have their state validated**
  - Usually involves checking for initialized pointers or flags
- **User-mode callbacks could alter the state and update properties of objects**
  - A drop down menu is no longer active
  - The parent of a window has changed
  - The partner in a DDE conversation terminated

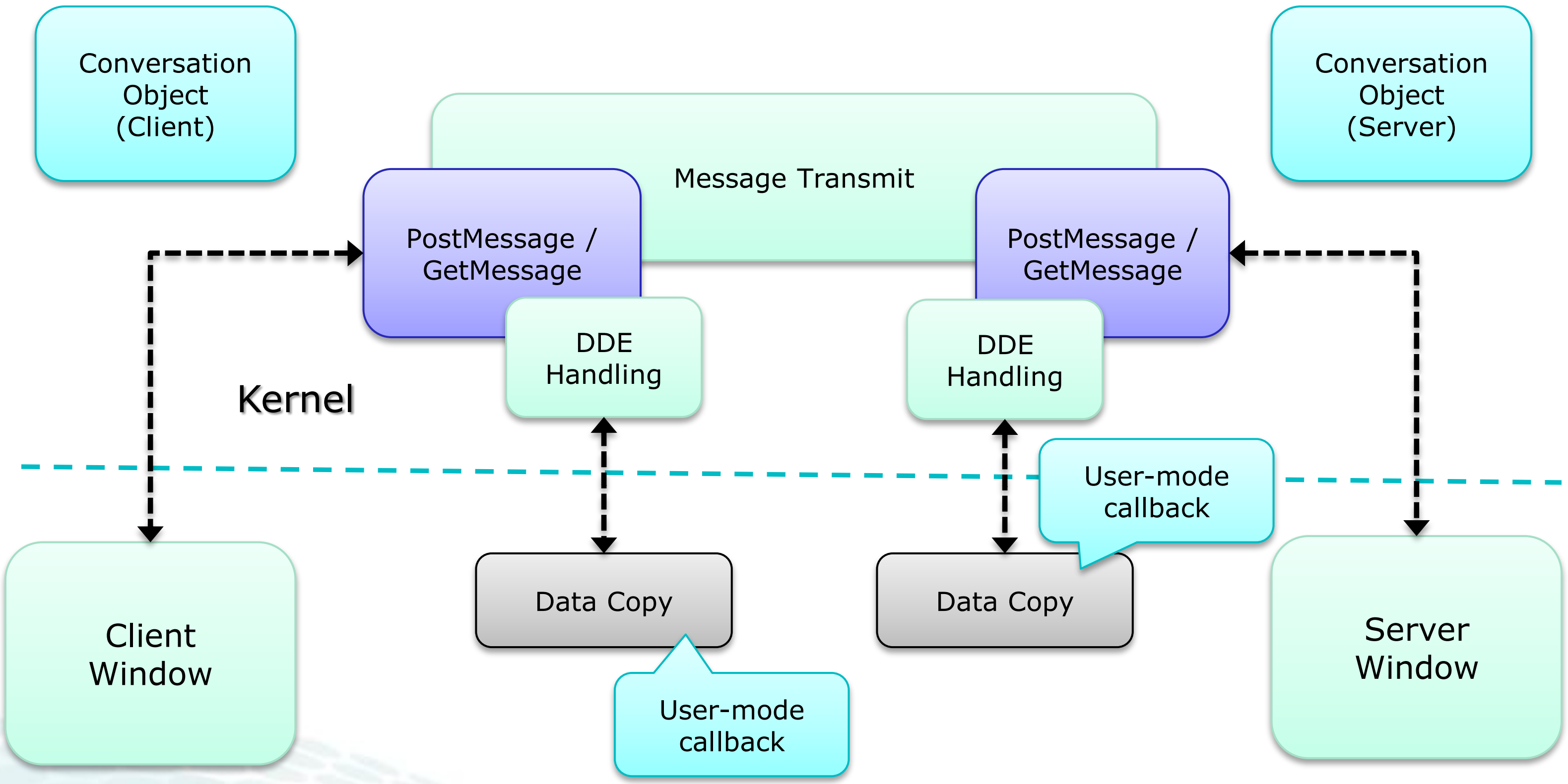
NORMAN

DDE Conversation State Vulnerabilities

- **Dynamic Data Exchange (DDE)**
  - Legacy protocol using messages and shared memory to exchange data between applications
- **Several functions did not sufficiently validate DDE conversation objects after user-mode callbacks**
  - Used to copy data in and out from user-mode
- **An attacker could terminate a conversation in a user-mode callback and thus unlock the partner conversation object**
  - Could result in a NULL pointer dereference as the function did not revalidate the conversation object pointer

NORMAN

DDE Conversation Message Handling



DDE Conversation Object NULL Dereference

```
loc_BF8FB899:
mov     edi, [ebp+arg_4]
lea     eax, [ebp+P]
push    eax
lea     eax, [ebp+arg_4]
push    eax
lea     eax, [ebp+var_4]
push    eax
push    dword ptr [edi]
call    _xxxCopyDdeIn@16 ; xxxCopyDdeIn(x,x,x,x)
mov     ebx, eax
cmp     ebx, 2
        short loc_BF8FB8FC
```

Terminate the conversation in a user-mode callback

User-mode callback(s)

Copy data to be sent in from user-mode

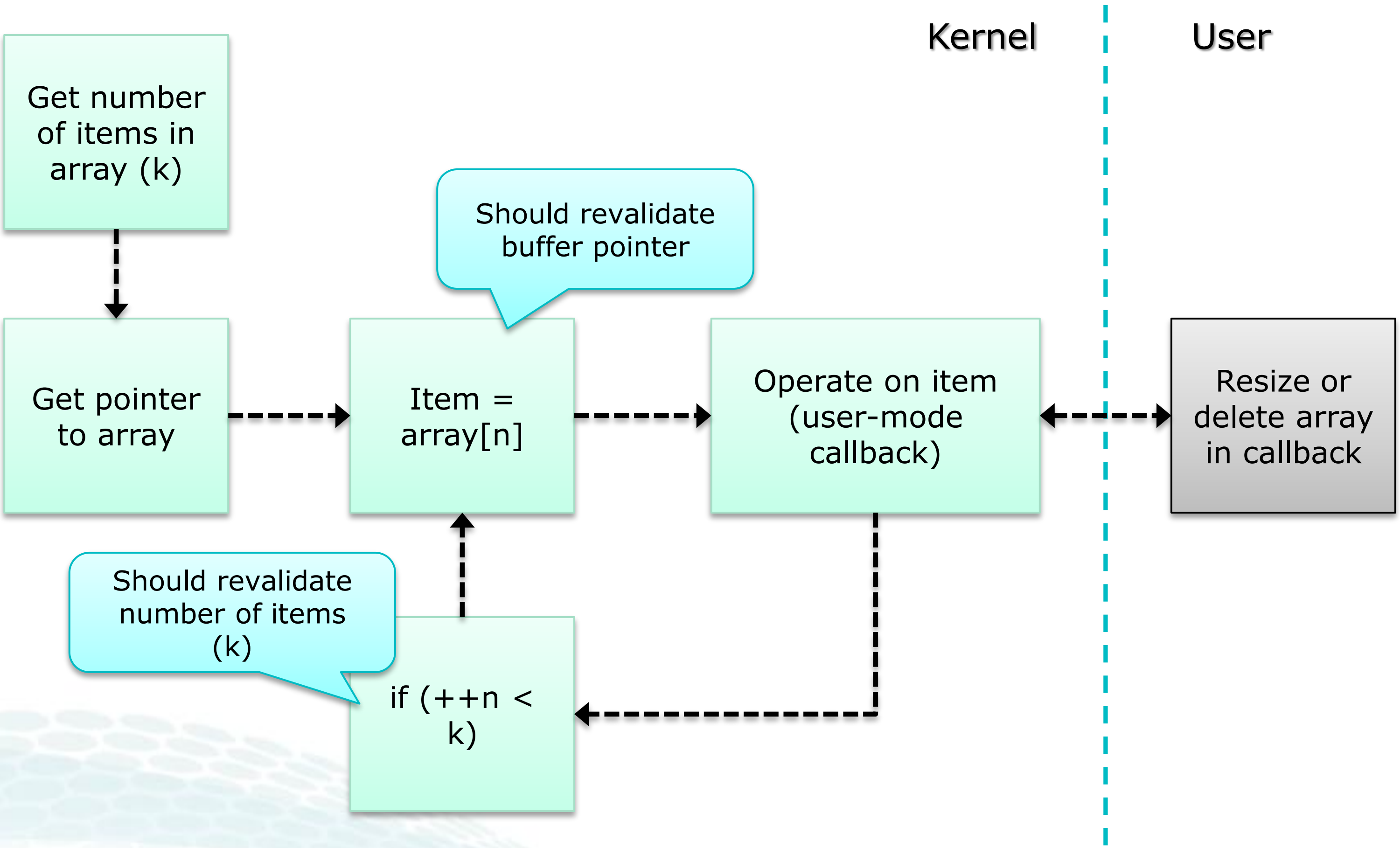
```
push    [ebp+var_4]      ; int
mov     eax, [ebp+arg_0]
push    [ebp+P]          ; P
or      dword ptr [eax], 80000000h
push    0                ; int
push    [ebp+arg_4]      ; int
push    offset _xxxExecuteAck@12 ; int
push    dword ptr [esi+10h] ; conversation object pointer
call    _AnticipatePost@24 ; AnticipatePost(x,x,x,x,x,x)
mov     [edi], eax
test    eax, eax
jz      short loc_BF8FB8F9
```

Possible NULL pointer dereference

Buffer Reallocation

- **Many user objects have item arrays or other forms of buffers associated with them**
  - E.g. menu items array
- **Item arrays where elements are added or removed are often resized to conserve memory**
  - Buffer freed if the array is empty
  - Buffer reallocated if elements is above or below a certain threshold
- **Any buffer that can be reallocated or freed during a callback must be checked upon return**
  - Failure to do so could result in use-after-free

Buffer Reallocation

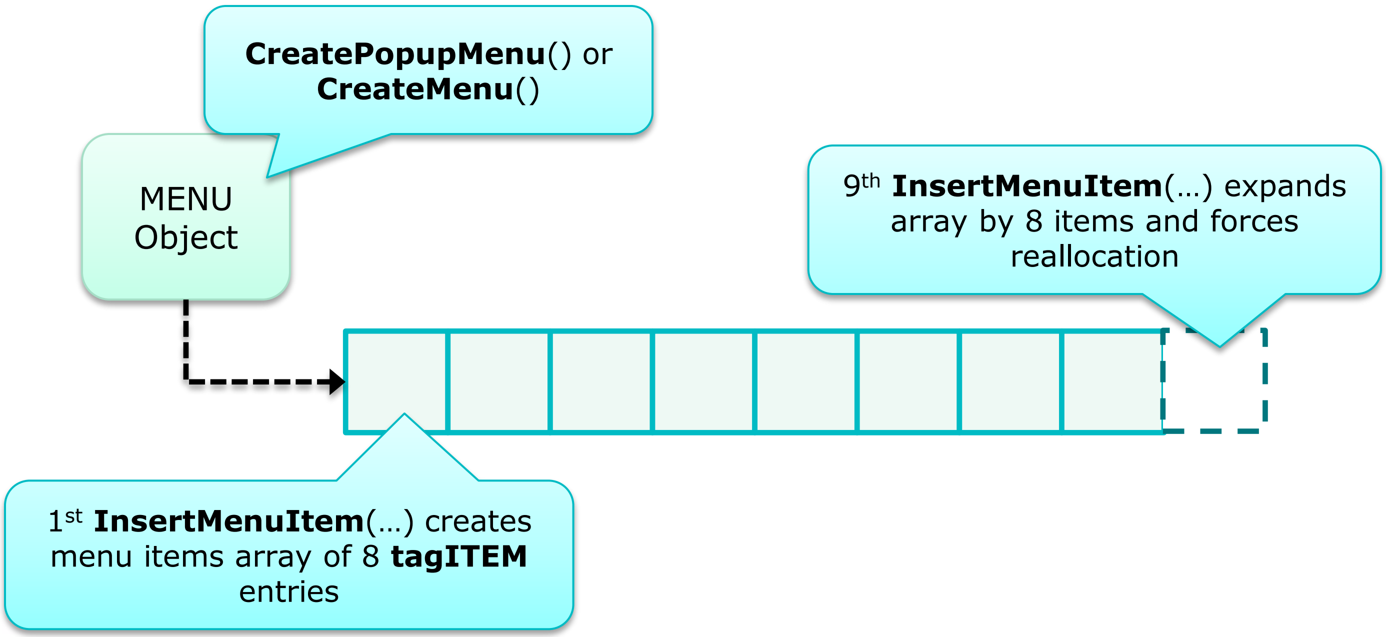


Menu Item Array Use-After-Frees

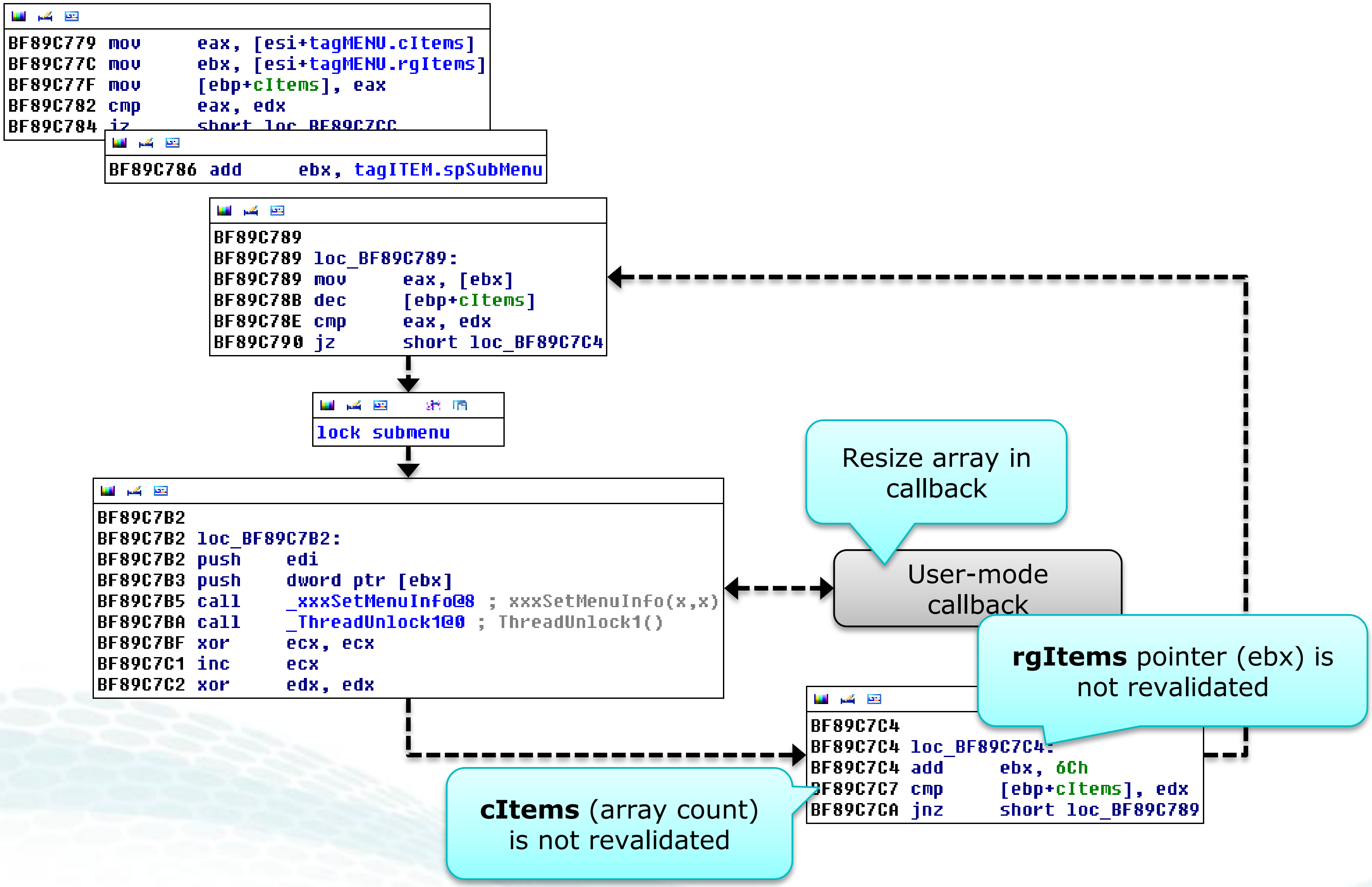
- **Menus may hold an arbitrary number of menu items**
  - Stored in a dynamically sized array pointed to by the menu object structure (win32k!tagMENU)
- **Win32k did not revalidate the menu items array pointer after user-mode callbacks**
  - No way to “lock” a menu item
  - Any ‘xxx’ function operating on menu items was potentially vulnerable
- **An attacker could cause the buffer to be reallocated in a callback and trigger a use-after-free**

NORMAN

Menu Item Array Reallocation



Menu Item Processing Use-After-Free



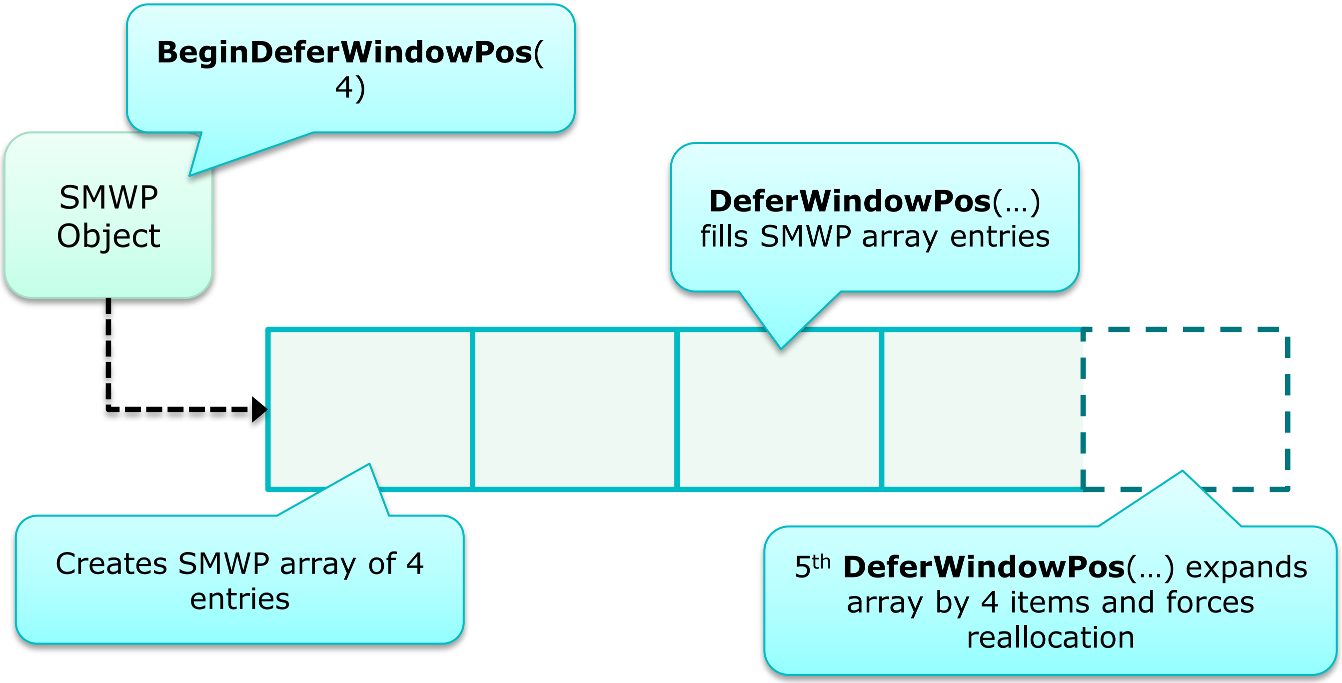
NORMAN

SetWindowPos Array Use-After-Frees

- **SMWP objects are used to update the position of multiple windows at once**
  - Created in **BeginDeferWindowPos(** int dwNum **)**
  - Hold a dynamically sized array of multiple window position structures
- **In operating on the SMWP array, win32k did not revalidate the array pointer after user-mode callbacks**
- **An attacker could force the array to be reallocated by inserting entries using DeferWindowPos(…) and trigger a use-after-free**

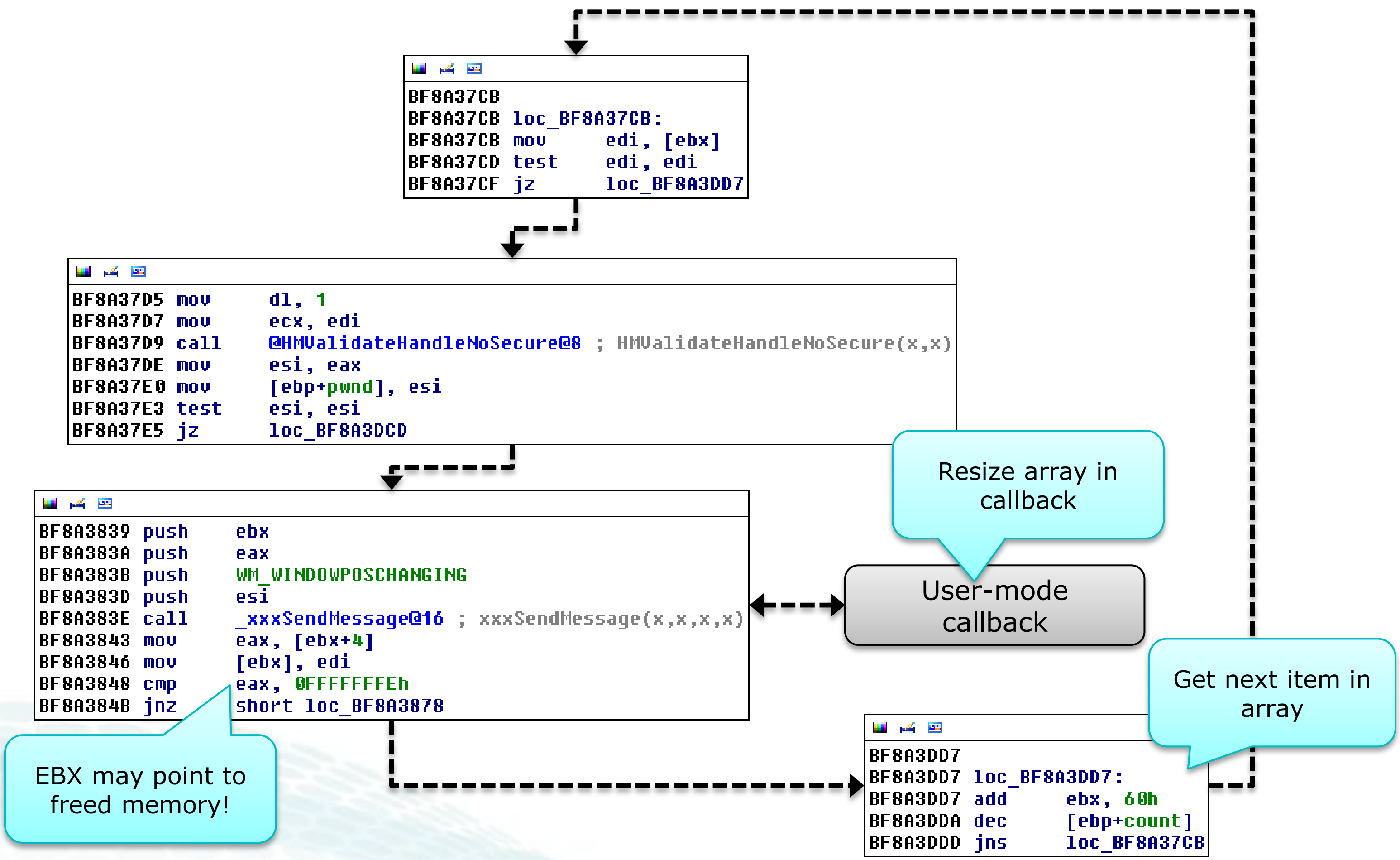
NORMAN

SetWindowPos Array Reallocation



NORMAN

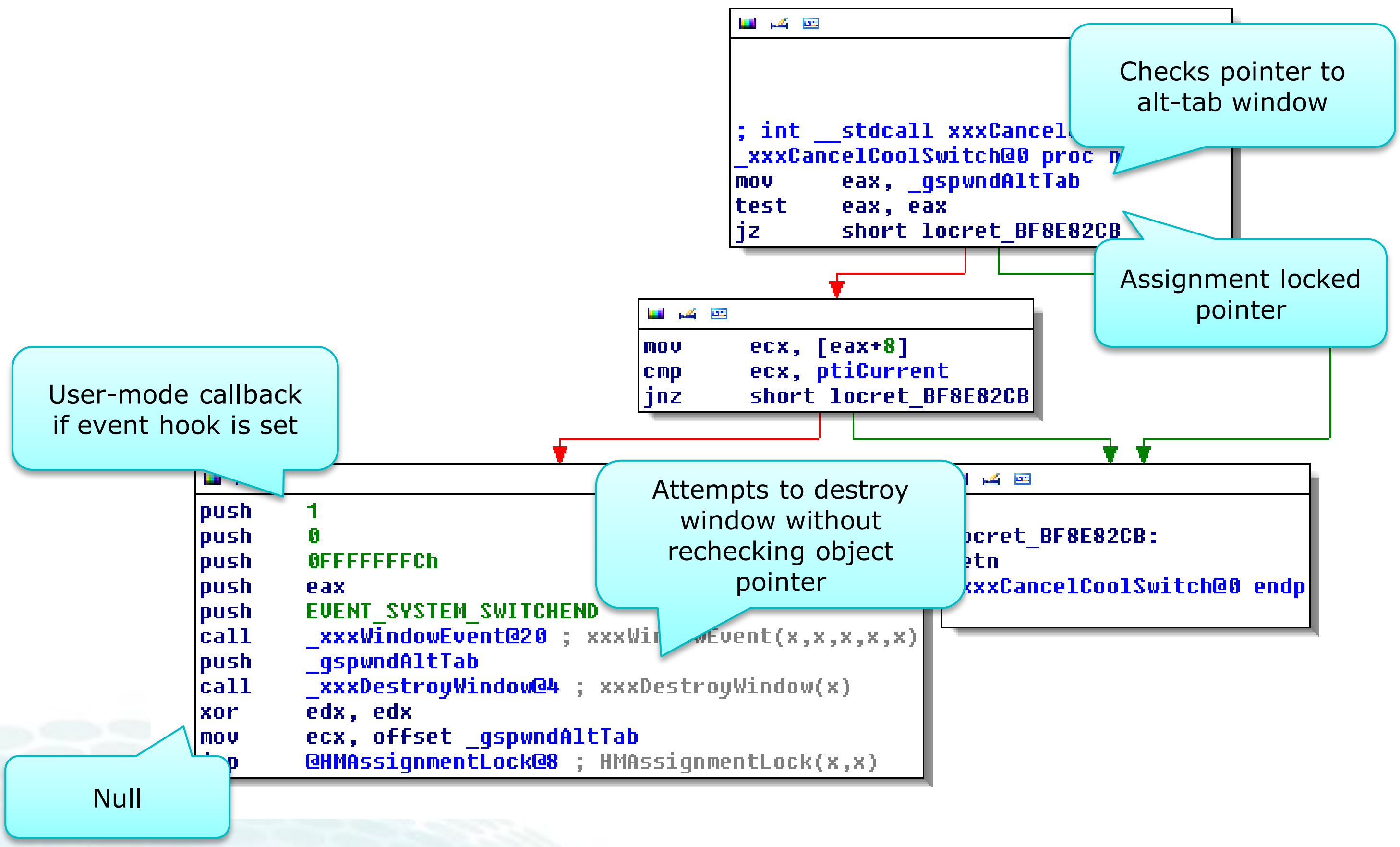
SMWP Item Processing Use-After-Free



Time-of-Check-to-Time-of-Use

- **The user critical section is generally used to prevent TOCTTOU issues in user object handling**
  - User-mode callbacks may allow an attacker to manipulate an object or global value before it is used
- **Can be particularly dangerous in clean up routines**
  - May invoke callbacks after checks have been made
  - Could result in stale references to objects or buffers
- **Values that may have changed must always be (re)checked after a callback has taken place**

Time-of-Check-to-Time-of-Use



Handle Validation

- **Required to validate handles, their type, and retrieve the corresponding object pointers**
  - HMValidateHandle() and friends
- **Generic handle validation should be avoided unless the structure of the object is irrelevant**
  - Only checks handle table entry and ignores type
- **Functions that revalidate handles after callbacks, may no longer be operating on the same object**
  - The uniqueness counter designed to provide handle entropy is only 16-bit

Insufficient Handle Validation

```
xxxGetMenuBarInfo(x,x,x,x)+297  mov     edi, [ebp+pwnd]
xxxGetMenuBarInfo(x,x,x,x)+29A  push    ecx
xxxGetMenuBarInfo(x,x,x,x)+29B  push    ecx
xxxGetMenuBarInfo(x,x,x,x)+29C  push    1E1h
xxxGetMenuBarInfo(x,x,x,x)+2A1  push    edi
xxxGetMenuBarInfo(x,x,x,x)+2A2  call    _xxxSendMessage@16 ; xxxSendMessage(x,x,x,x)
xxxGetMenuBarInfo(x,x,x,x)+2A7  mov     edx, _aheList    ; handle table pointer
xxxGetMenuBarInfo(x,x,x,x)+2AD  mov     ecx, eax         ; eax: user handle
xxxGetMenuBarInfo(x,x,x,x)+2AF  and     ecx, 0FFFFh
xxxGetMenuBarInfo(x,x,x,x)+2B5  lea     ecx, [ecx+ecx*2]
xxxGetMenuBarInfo(x,x,x,x)+2B8  mov     ecx, [edx+ecx*4]
xxxGetMenuBarInfo(x,x,x,x)+2BB  test    ecx, ecx         ; ecx: object pointer
xxxGetMenuBarInfo(x,x,x,x)+2BD  jz      loc_BF91E14B
```

Function did not check handle type nor validate index in handle table

Function did not check that object was an image (icon/cursor)

```
xxxClientLoadImage(x,x,x,x,x,x,x)+16D  or      dl, 0FFh          ; TYPE_GENERIC
xxxClientLoadImage(x,x,x,x,x,x,x)+170  mov     ecx, edi          ; handle from user-mode callback
xxxClientLoadImage(x,x,x,x,x,x,x)+172  call    @HMValidateHandleNoRip@8 ; HMValidateHandleNoRip(x,x)
xxxClientLoadImage(x,x,x,x,x,x,x)+177  mov     edi, eax
```

NORMAN

Exploitability

Use-After-Frees and NULL Pointer Dereferences

Vulnerability Primitives

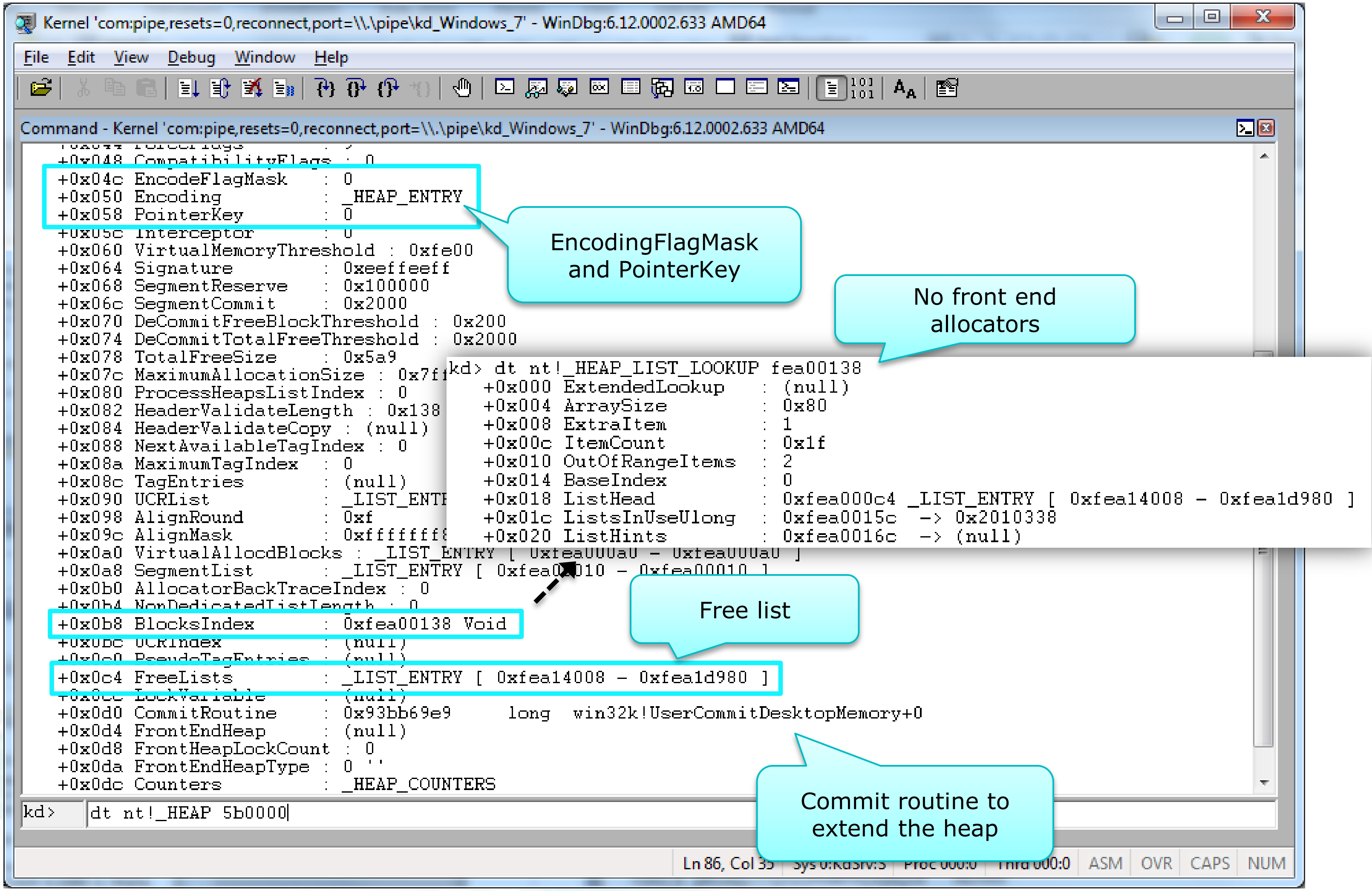
- **Mainly dealing with two vulnerability primitives**
  - Use-After-Frees
  - Null-Pointer Dereferences
- **Exploitability may depend on the attacker's ability to manipulate heap and pool memory**
  - Kernel Pool Exploitation on Windows 7 (BH DC '11)
  - Not much public information on the kernel heap
- **Hooking user-mode callbacks is easy**
  - NtCurrentPeb()->KernelCallbackTable

NORMAN

Kernel Heap

- **The kernel has a stripped down version of the user-mode heap allocator**
  - nt!RtlAllocateHeap, nt!RtlFreeHeap, etc.
  - Used by the shared and desktop heaps
- **Neither heaps employ any front end allocators**
  - ExtendedLookup == NULL
  - No low fragmentation heap or lookaside lists
- **Neither heaps encode or obfuscate heap management structures**
  - HEAP.EncodeFlagMask == 0

Desktop Heap Base



NORMAN

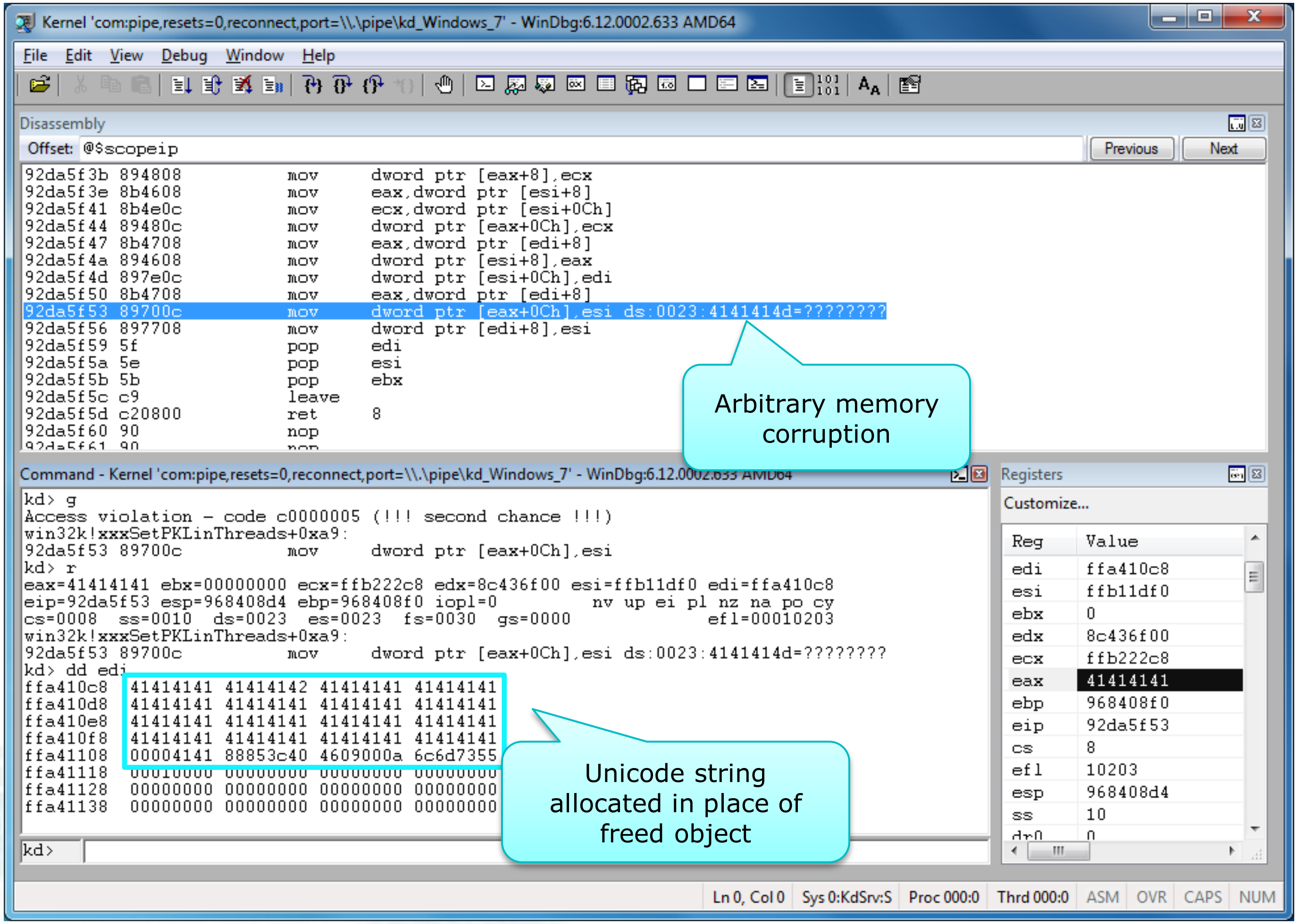
Kernel Heap Management

- **Freed memory is indexed into a single free list**
  - Ordered by block size
  - *ListHints* used to optimize list lookup
- **Requested memory is always pulled from the front of an oversized heap chunk**
  - Remaining fragment is put back into the free list
- **If the heap runs out of committed memory, win32k calls the *CommitRoutine* to extend the heap**
  - Attempts to commit memory from the reserved range
  - E.g. win32k reserves 0xC00000 bytes by default (adjustable by user) for desktop heaps

Use-After-Free Exploitation

- **Unicode strings can be used to reallocate freed memory from within user-mode callbacks**
  - Allows control of the contents and size of the heap block
  - Caveat: Cannot use WORD NULLs and last two bytes must be NULL to terminate the string
- **Desktop heap**
  - `SetWindowTextW(hWnd,String);`
- **Session pool**
  - `SetClassLongPtr(hWnd,GCLP_MENUNAME,(LONG)String);`

Strings As User Objects

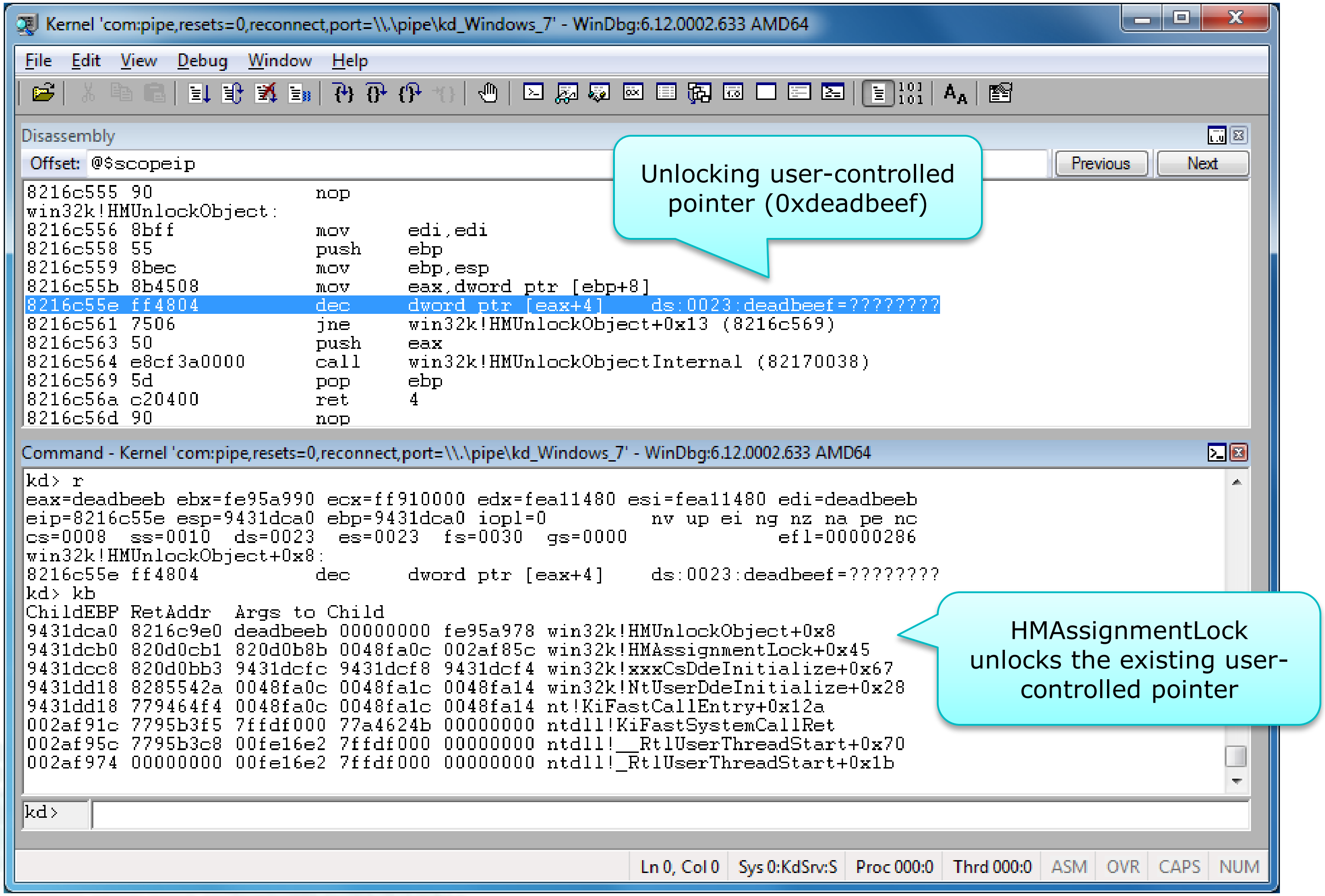


NORMAN

Exploiting Object Locking Behavior

- **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**
  - HANDLEENTRY.bType
    - Decrement the type of a window handle table entry (1)
    - Destroy routine for free type (0) is null (mappable by user)
  - KAPC.ApcMode
    - Execute code with kernel-mode privileges by decrementing UserMode (1) to KernelMode (0)

Exploiting Object Locking Behavior



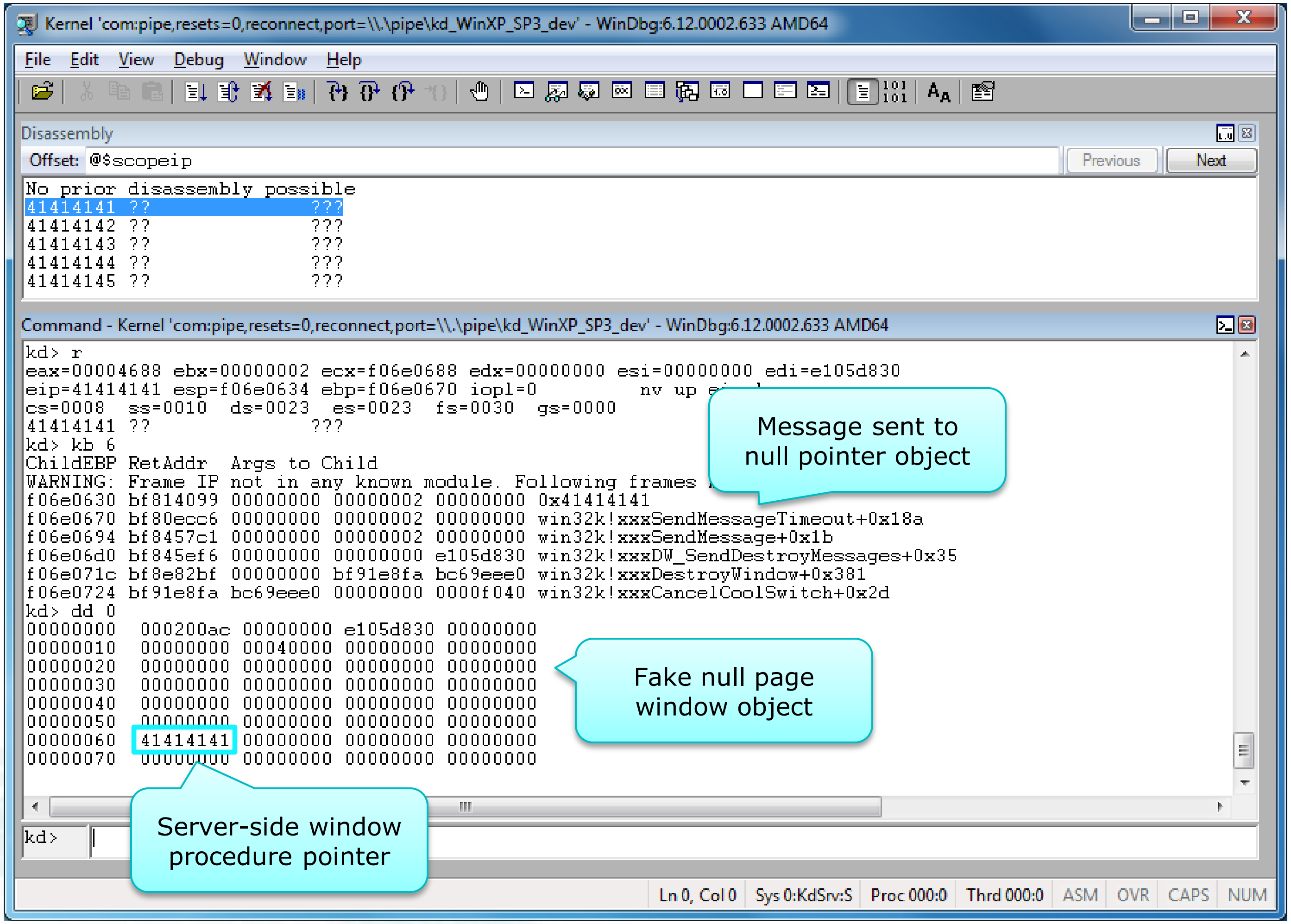
NORMAN

NULL Pointer Vulnerabilities

- **Potentially exploitable on the Windows platform**
  - Non-privileged users can map the null page, e.g. via NtAllocateVirtualMemory or NtMapViewOfFile
- **Many NULL pointer vulnerabilities are concerned with window object pointers**
- **An attacker could map the null page and set up a fake window object**
  - E.g. define a server-side window procedure and handle messages with kernel level privileges

NORMAN

NULL Pointer Object Exploitation



Demo

- **Window Object Use-After-Free (CVE-2011-1237)**
  - Arbitrary kernel code execution via HANDLEENTRY corruption

Mitigations

Protecting Against Privilege Escalation Vulnerabilities

NORMAN

Mitigating Use-After-Free Exploitation

- **Need to address an attacker's ability to reallocate the freed memory before use**
- **Some approaches**
  - Delayed frees while processing a callback
  - Dedicated free lists for user objects
  - Isolate strings used in reallocating memory
  - Track allocations between ring transitions, e.g. pointers on the stack before a callback
- **Generally hard to mitigate without significantly impacting performance**

NORMAN

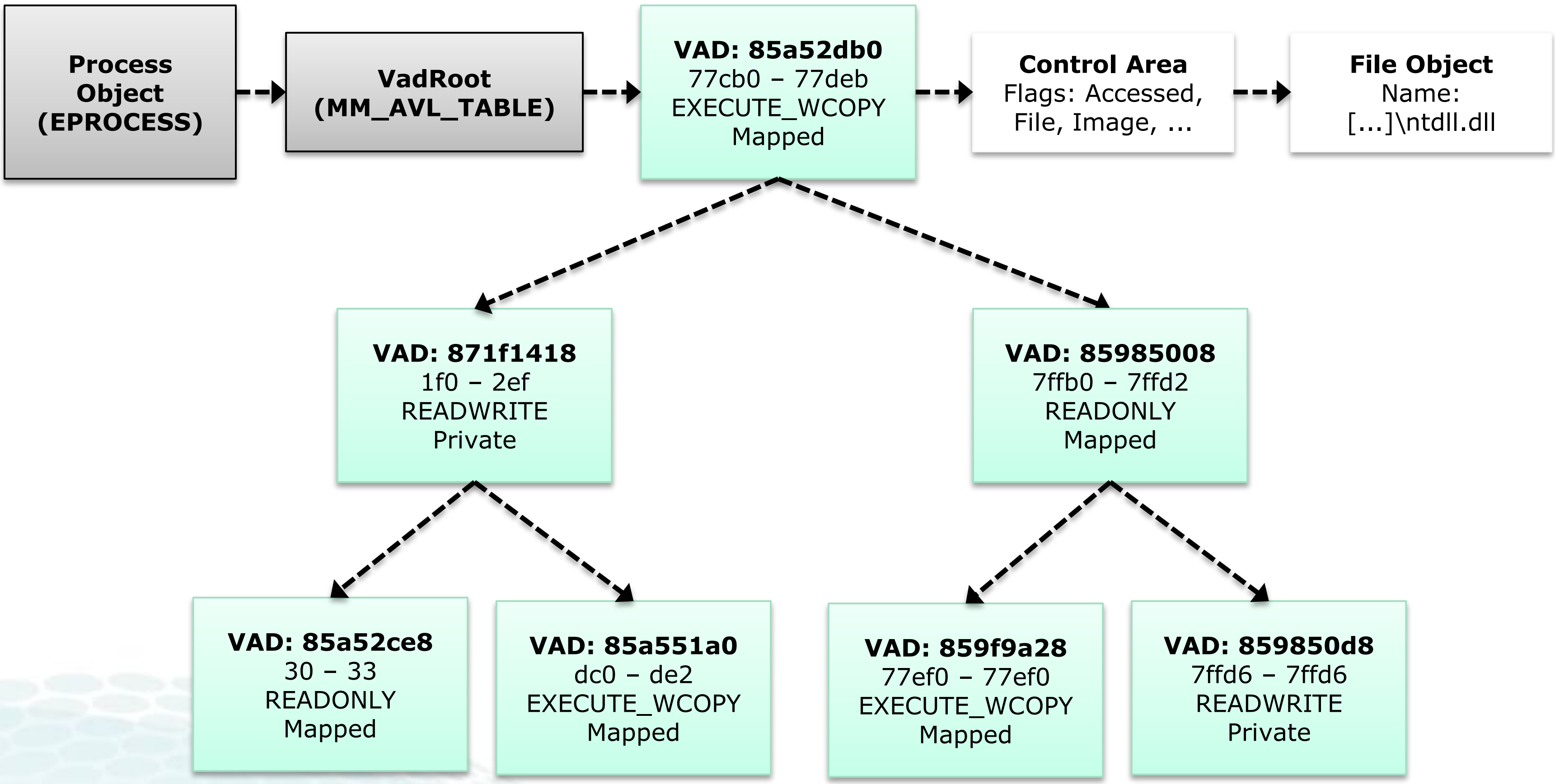
Mitigating NULL Pointer Exploitation

- **We can address null pointer exploitation by denying users the ability to map the null page**
- **Some potential ways of addressing null page mappings**
  - System call hooking
  - Page Table Entry (PTE) modification
  - VAD manipulation
- **System call hooking not supported on x64**
- **PTE modification requires page to be mapped**

VAD Manipulation

- **User mode process space is described using Virtual Address Descriptors (VADs)**
  - Structured in self-balanced AVL trees
- **VADs are always checked before PTEs are created**
  - E.g. used to implement the NO_ACCESS protection
- **VADs are used to secure memory, e.g. made non-deletable**
  - PEBs and TEBs
  - KUSER_SHARED_DATA section

VAD Tree

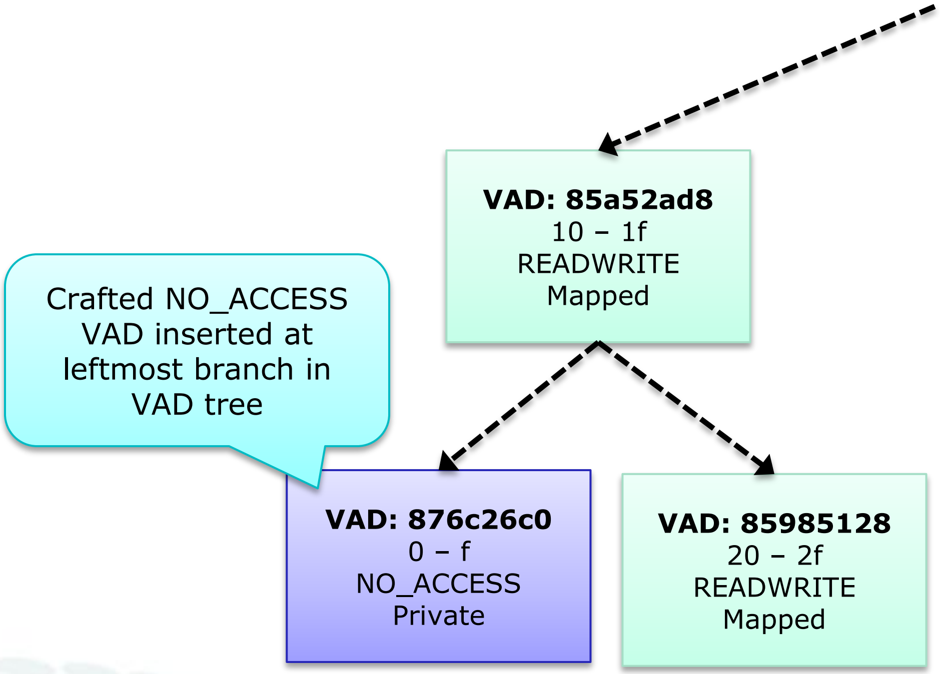


NORMAN

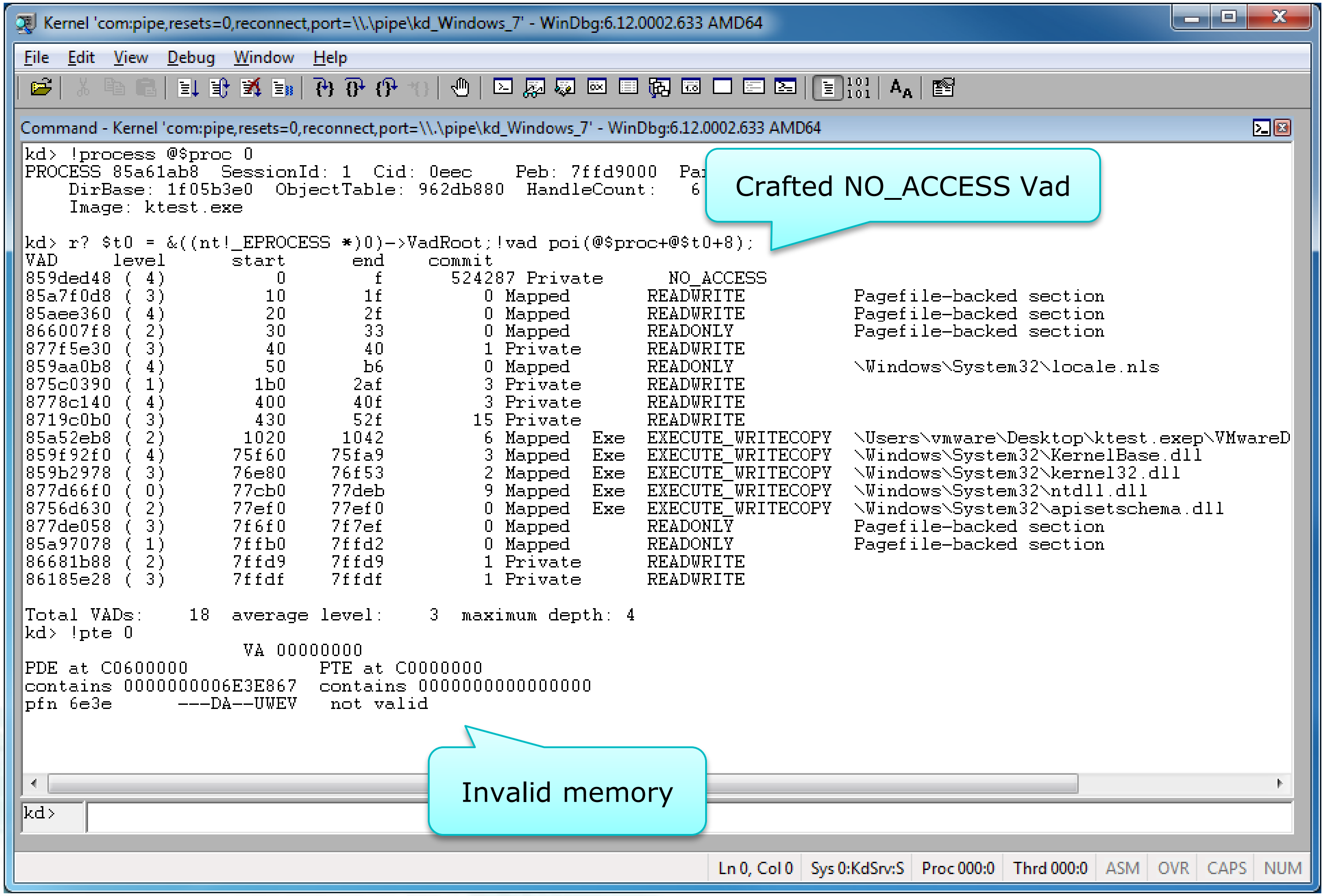
Restricting Null Page Access

- **We insert a crafted VAD entry to restrict null page access**
  - Ring3 code cannot modify the VAD entry
- **Avoid deletion using the same method employed by PEBs and TEBs**
  - Secure address range from 0 up to 0xFFFF
  - Set protection to NO_ACCESS
- **Use a special VAD flag to prevent memory commits**
  - Protection cannot be changed on uncommitted memory!

VAD Tree /w Crafted Entry



Manipulated Process VAD Tree



Mitigation Results

Function	Addr	Type	Protection	Result
NtAllocateVirtualMemory	1	MEM_RESERVE	READONLY	0xC0000018
NtAllocateVirtualMemory	1	MEM_COMMIT	READONLY	0xC0000018
NtMapViewOfSection	1	MEM_DOS_LIM*	READONLY	0xC0000018
NtProtectVirtualMemory	0		READWRITE	0xC000002D
NtProtectVirtualmemory	0		READONLY	0xC0000045
NtFreeVirtualMemory	0	MEM_RELEASE		0xC0000045

0xC0000018	STATUS_CONFLICTING_ADDRESSES
0xC000002D	STATUS_NOT_COMMITTED
0xC0000045	STATUS_INVALID_PAGE_PROTECTION

*Allows section mapping on page boundary on x86 platforms

NORMAN

Demo

- Null page mapping mitigation

Conclusion

Remarks and Conclusion

Future of the Win32k Subsystem

- **Win32k needs a much more consistent and security oriented design**
  - It should not be necessary for the kernel to make direct calls back into user-mode
  - Reconsider performance benefit of shared user and kernel-mode memory mappings
- **The Window Manager should provide mutual exclusion on a per-object basis**
  - Better suited towards multicore architectures
  - Similar to what is done in GDI and the NT executive

NORMAN

Conclusion

- **Legacy components constitute the most vulnerable parts of an operating system**
  - Security is not usually part of the original design
  - Win32k is built around very old GUI subsystem code
- **Kernel exploitation requires knowledge about the kernel address space**
  - Limiting access to such information is important
- **Although hard, mitigating Windows kernel exploitation is possible**

NORMAN

References

- **Windows Kernel Internals: Win32K.sys**
  - David B. Probert, Microsoft
- **Analyzing Local Privilege Escalations in win32k**
  - mxatone (Uninformed Vol. 10)
- **Windows Creation Vulnerability (MS10-048)**
  - Nicolás Economou
- **Pointers and Handles: A Story of Unchecked Assumptions in the Windows Kernel**
  - Alex Ionescu
- **Understanding the Low Fragmentation Heap**
  - Chris Valasek

Questions ?

- Email: kernelpool@gmail.com
- Blog: http://mista.nu/blog
- Twitter: @kernelpool
- Norman MDT Blog: http://blogs.norman.com/category/malware-detection-team

NORMAN