Kernel Attacks through User-Mode Callbacks

Tarjei Mandt

Norman Threat Research
tarjei.mandt@norman.com

Abstract. 15 years ago, Windows NT 4.0 introduced Win32k.sys to address the inherent limitations of the older client-server graphics subsystem model. Today, win32k still remains a fundamental component of the Windows architecture and manages both the Window Manager (User) and Graphics Device Interface (GDI). In order to properly interface with user-mode data, win32k makes use of user-mode callbacks, a mechanism allowing the kernel to make calls back into user-mode. User-mode callbacks enable a variety of tasks such as invoking application-defined hooks, providing event notifications, and copying data to/from user-mode. In this paper, we discuss the many challenges and problems concerning user-mode callbacks in win32k. In particular, we show how win32k's dependency on global locks in providing a thread-safe environment does not integrate well with the concept of user-mode callbacks. Although many vulnerabilities related to user-mode callbacks have been addressed, their complex nature suggests that more subtle flaws might still be present in win32k. Thus, in an effort to mitigate some of the more prevalent bug classes, we conclusively provide some suggestions as to how users may protect themselves against future kernel attacks.

Keywords: win32k, user-mode callbacks, vulnerabilities

1 Introduction

In Windows NT, the Win32 environment subsystem allows applications to interface with the Windows operating system and interact with components such as the Window Manger (User) and the Graphics Device Interface (GDI). The subsystem provides a set of functions, collectively known as the Win32 API, and follows a client-server model in which client applications communicate with a more privileged server component.

Traditionally, the server side of the Win32 subsystem was implemented in the client-server runtime subsystem (CSRSS). In order to provide optimal performance, each thread on the client side had a paired thread on the Win32 server waiting in a special interprocess communication facility called Fast LPC. As transitions between paired threads in Fast LPC did not require a scheduling event in the kernel, the server thread could run for the remaining time slice of the client thread before taking its turn in the preemptive thread scheduler. Additionally, shared memory was used for both large data transfers and providing clients read-only access to server managed data structures to minimize the need for transitions between clients and the Win32 server.

In spite of the performance optimizations made to the traditional Win32 subsystem, Microsoft decided with the release of Windows NT 4.0 to migrate a large part of the server component into kernel-mode. This lead to the introduction of Win32k.sys, a kernel mode driver managing both the Window Manager (User) and the Graphics Device Interface (GDI). The move to kernel-mode greatly reduced the overhead associated with the old subsystem design, by having far less thread and context switches (and using the much faster user/kernel transition) and reducing memory requirements. However, as user/kernel transitions are relatively slow compared to direct code/data access within the same privilege level, some old tricks such as caching of management structures in the user-mode portion of the client's address space were still maintained. Moreover, some management structures were stored exclusively in user-mode in order to avoid ring transitions. As Win32k needed a way to access this information and also support basic functionality such as windows hooking, it required a way to pass control to the user-mode client. This was realized through the user-mode callback mechanism.

User-mode callbacks allow win32k to make calls back into user-mode and perform tasks such as invoking application-defined hooks, providing event notifications, and copying data to/from user-mode. In this paper, we discuss the many challenges and problems concerning user-mode callbacks in win32k. In particular, we show how win32k's design in preserving data integrity (such as in relying on global locking) does not integrate well with the concept of user-mode callbacks. Recently, MS11-034 [7] and MS11-054 [8] addressed several vulnerabilities in an effort to address multiple bug classes related to user-mode callbacks. However, due to the complex nature of some of these issues and the prevalence of user-mode callbacks, more subtle flaws are likely to still be present in win32k. Thus, in an effort to mitigate some of the more prevalent bug classes, we conclusively discuss some ideas as to what both Microsoft and end-users might do to further mitigate the risk of future attacks in the win32k subsystem.

The rest of the paper is organized as follows. In Section 2, we review background material necessary to understand the remained of the paper, focused on user objects and user-mode callbacks. In Section 3, we discuss function name decoration in win32k and present several vulnerability classes peculiar to win32k and user-mode callbacks. In Section 4, we evaluate the exploitability of vulnerabilities triggered by user-mode callbacks, while we in Section 5 attempt to address these attacks by proposing mitigations for prevalent vulnerability classes. Finally, in Section 6 we provide thoughts and suggestions on the future of win32k and in Section 7 we provide a conclusion of the paper.

2 Background

In this section, we review the background information necessary to understand the remainder of the paper. We begin by briefly introducing Win32k and its architecture, before moving onto more specific components such as the Window Manager (focused on user objects) and the user-mode callback mechanism.

2.1 Win32k

Win32k.sys was introduced as part of the changes made in Windows NT 4.0 to increase graphics rendering performance and reduce the memory requirements of Windows applications [10]. Notably, the Windows Manager (User) as well as the Graphics Device Interface (GDI) were moved out of the client-server runtime subsystem (CSRSS) and implemented into a kernel module of its own. In Windows NT 3.51, graphics rendering and user interface management were performed by CSRSS using a fast form of interprocess communication between the application (client) and the subsystem server process (CSRSS.EXE). Although this design was optimized for performance, the graphics intensive nature of Windows lead developers to move to a kernel based design with the much faster system calls.

Win32k essentially consists of three major components: the Graphics Device Interface (GDI), the Window Manager (User), and thunks to DirectX APIs to support both the XP/2000 and Longhorn (Vista) display driver models (sometimes also considered to be a part of GDI). The Window Manager is responsible for managing the Windows user interface, such as controlling window displays, managing screen output, collecting input from mouse and keyboard, and passing messages to applications. GDI, on the other hand, is mostly concerned with graphics rendering and implements GDI objects (brushes, pens, surfaces, device contexts, etc.), the graphics rendering engine (Gre), printing support, ICM color matching, a floating point math library, and font support.

As the traditional subsystem design of CSRSS was built around having one process per user, each user session has its own mapped copy of win32k.sys. The concept of sessions also allows Windows to provide a more strict separation between users (otherwise known as session isolation). As of Windows Vista, services were also moved into their own non-interactive session [2] to avoid the array of problems associated with shared sessions such as shatter attacks [12] and vulnerabilities in privileged services. Moreover, User Interface Privilege Isolation (UIPI) [1] implements the concept of integrity levels and ensures that low privilege processes cannot interact (e.g. pass messages to) processes of a higher integrity.

In order to properly interface with the NT executive, win32k registers several callouts (`PsEstablishWin32Callouts`) to support GUI oriented objects such as desktops and window stations. Importantly, win32k also registers callouts for threads and processes to define per-thread and per-process structures used by the GUI subsystem.

GUI Threads and Processes As not all threads make use of the GUI subsystem, allocating GUI structures up front for all threads would be a waste of space. Hence, all threads on Windows start as non-GUI threads (12 KB stack). If a thread accesses any of the USER or GDI system calls (number >= 0x1000), Windows promotes the thread to a GUI thread (`nt!PsConvertToGuiThread`) and calls the process and thread callouts. Notably, a GUI thread has a larger thread stack to better deal with the recursive nature of win32k as well as support user-mode callbacks which may require additional stack space[1] for trap frames and other metadata.

When the first thread of a process is converted to a GUI thread and calls `W32pProcessCallout`, win32k calls `win32k!xxxInitProcessInfo` to initialize the per-process `W32PROCESS`/`PROCESSINFO`[2] structure. Specifically, this structure holds GUI-related information specific to each process such as the associated desktop, window station, and user and GDI handle counts. The function allocates the structure itself in `win32k!AllocateW32Process` before the USER related fields are initialized in `win32k!xxxUserProcessCallout` followed by the GDI related fields initialized in `GdiProcessCallout`.

Additionally, win32k also initializes a per-thread `W32THREAD`/`THREADINFO` structure for all threads that are converted to GUI threads. This structure holds thread specific information related to the GUI subsystem such as information on the thread message queues, registered windows hooks, owner desktop, menu state, and so on. Here, `W32pThreadCallout` calls `win32k!AllocateW32Thread` to allocate the structure, followed by `GdiThreadCallout` and `UserThreadCallout` to initialize information peculiar to the GDI and USER subsystems respectively. The most important function in this process is `win32k!xxxCreateThreadInfo`, which is responsible for initializing the thread information structure.

2.2 Window Manager

One of the important functions of the Window Manager is to keep track of user entities such as windows, menus, cursors, and so on. It does this by representing such entities as *user objects* and maintains its own handle table to keep track of their use within a user session. Thus, when an application requests an action to be performed on a user entity, it provides its handle value which the handle manager efficiently maps to the corresponding object in kernel memory.

User Objects User objects are separated into types and thus have their own type specific structures. For instance, all window objects are defined by the `win32k!tagWND` structure, while menus are defined by the `win32k!tagMENU` structure. Although object types are structurally different, they all share a common header known as the `HEAD` structure (Listing 1).

The `HEAD` structure holds a copy of the handle value (`h`) as well as a lock count (`cLockObj`), incremented whenever an object is being used. When the object is no longer being used by a particular component, its lock count is decremented. At the point where the lock count reaches zero, the Window Manager knows that the object is no longer being used by the system and frees it.

[1] On Vista and later, user-mode callbacks use dedicated kernel thread stacks.

[2] `W32PROCESS` is a subset of `PROCESSINFO`, and deals with the GDI subsystem while `PROCESSINFO` also contains information specific to the USER subsystem.

```
typedef struct _HEAD {
    HANDLE       h;
    ULONG32      cLockObj;
} HEAD, *PHEAD;
```

Listing 1: HEAD structure

Although the HEAD structure is fairly small, objects many times use the larger thread or process specific header structures such as THRDESKHEAD and PROCDESKHEAD. These structures provide additional fields such as the pointer to the thread information structure tagTHREADINFO and the pointer to the associated desktop object (tagDESKTOP). In providing this information, Windows can restrict objects on other desktops from being accessed and thus provide isolation between desktops. Similarly, as objects are always owned by a thread or process, isolation between threads or processes that coexist on the same desktop can be achieved as well. For instance, a given thread cannot destroy the window objects of other threads by simply calling DestroyWindow. Instead, it would need to send a window message which is subject to additional validation such as integrity level checks. However, as the object isolation is not provided in a uniform and centralized manner, any function not performing the required checks could allow an attacker to bypass this restriction. This was undeniably one of the reasons for the session separation (Vista and later) between privileged services and the logged in user session. As all processes and threads in the same session share the same user handle table, a low-privileged process could potentially pass messages to or interact with objects owned by a high-privileged process.

Handle Table All user objects are indexed into a per-session handle table. The handle table is initialized in win32k!Win32UserInitialize, invoked whenever a new instance of win32k is loaded. The handle table itself is stored at the base of a shared section (win32k!gpvSharedBase), also set up by Win32UserInitialize. This section is subsequently mapped into every new GUI process, thus allowing processes to access handle table information from user-mode without having to resort to a system call. The decision to map the shared section into user-mode was seen as a performance benefit and was also used in the non-kernel based Win32 subsystem design to prevent excessive context switching between client applications and the client-server runtime subsystem process (CSRSS). On Windows 7, a pointer to the handle table is stored in the shared information structure (win32k!tagSHAREDINFO). A pointer to this structure is available from both user-mode (user32!gSharedInfo[3]) and kernel-mode (win32k!gSharedInfo).

[3] Windows 7 only

```
typedef struct _HANDLEENTRY {
    struct _HEAD* phead;
    VOID*        pOwner;
    UINT8        bType;
    UINT8        bFlags;
    UINT16       wUniq;
} HANDLEENTRY, *PHANDLEENTRY;
```

Listing 2: `HANDLEENTRY` structure

Each entry in the user handle table is represented by a `HANDLEENTRY` structure, as shown in Listing 2. Specifically, this structure contains information on the object specific to a handle, such as the pointer to the object itself (`phead`), its owner (`pOwner`), and the object type (`bType`). The owner field is either a pointer to a thread or process information structure, or a null pointer in which case it is considered a session-wide object. An example of this would be the monitor or keyboard layout/file object, which are considered global to a session.

The actual type of the user object is defined by the `bType` value, and is under Windows 7 a value ranging from 0 up until 21 (Table 1). `bFlags` defines additional object flags, and is commonly used to indicate if an object has been destroyed. This may be the case if an object was requested destroyed, but is still kept in memory because its lock count its lock count is non-zero. Finally, the `wUniq` value is used as a uniqueness counter for computing handle values. A handle value is computed as `handle` = `table_entry_id` | (`wUniq` << 0x10). When an object is freed the counter is incremented to avoid subsequent objects from immediately reusing the previous handle. It should be noted that this mechanism cannot be seen as a security feature as the `wUniq` counter is only 16 bits, hence will wrap around when enough objects have been allocated and freed.

In order to validate handles, the Window Manager may call any of the `HMValidateHandle` APIs. These functions take the handle value as well as the handle type as parameters and look up the corresponding entry in the handle table. If the object is of the requested type, the object pointer is returned by the function.

User Objects in Memory In Windows, user objects and their associated data structures can reside in the desktop heap, the shared heap or the session pool. The general rule is that objects associated with a particular desktop are stored in the desktop heap, and the remaining objects are stored in the shared heap or the session pool. However, the actual locality of each object type is defined by a table known as the *handle type information table* (`win32k!ghati`). This table holds properties specific to each object type, used by the handle manager

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
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	Touch (Win 7)	Thread	Session Pool
21	Gesture (Win 7)	Thread	Session Pool

Table 1. Owner and locality of user objects

when allocating or freeing user objects. Specifically, each entry in the handle type information table is defined by an opaque structure (not listed) that holds the object allocation tag, type flags, and a pointer to a type-specific destroy routine. The latter is called whenever the lock count of an object reaches zero, in which case the Window Manager calls the type-specific destroy routine to properly free the object.

Critical Section Unlike objects managed by the NT executive, the Window Manager does not exclusively lock each user object. Instead, it implements a global lock per session using a critical section (resource) in win32k. Specifically, each kernel routines that operates on user objects or user management structures (typically `NtUser` system calls) must first enter the user critical section (i.e. acquire the `win32k!gpresUser` resource). For instance, functions that update kernel-mode structures must first call `UserEnterUserCritSec` and acquire the user resource for exclusive access before modifying data. In order to reduce the amount of lock contention in the Window Manager, system calls that only perform read operations enter the *shared* critical section (`EnterSharedCrit`). This allows win32k to achieve some sort of parallelism despite the global lock design, as multiple threads may be executing `NtUser` calls concurrently.

2.3 User-Mode Callbacks

Win32k is many times required to make calls back into user-mode for performing tasks such as invoking application-defined hooks, providing event notifications, and copying data to/from user-mode. Such calls are commonly referred to as user-mode callbacks [11][3]. The mechanism itself is implemented in `KeUserModeCallback` (Listing 3), exported by the NT executive, and operates much like a reverse system call.

```
NTSTATUS KeUserModeCallback (
    IN ULONG ApiNumber,
    IN PVOID InputBuffer,
    IN ULONG InputLength,
    OUT PVOID *OutputBuffer,
    IN PULONG OutputLength );
```

Listing 3: KeUserModeCallback

When win32k makes a user-mode callback, it calls `KeUserModeCallback` with the `ApiNumber` of the user-mode function it wants to call. Here, `ApiNumber` is an index into a function pointer table (`USER32!apfnDispatch`) whose address is copied to the Process Environment Block (`PEB.KernelCallbackTable`) during initialization of USER32.dll in a given process (Listing 4). Win32k provides the input parameters to the respective callback function by filling the `InputBuffer`, and receives the output from user-mode in `OutputBuffer`.

```
0:004> dps poi($peb+58)
00000000'77b49500  00000000'77ac6f74 USER32!_fnCOPYDATA
00000000'77b49508  00000000'77b0f760 USER32!_fnCOPYGLOBALDATA
00000000'77b49510  00000000'77ad67fc USER32!_fnDWORD
00000000'77b49518  00000000'77accb7c USER32!_fnNCDESTROY
00000000'77b49520  00000000'77adf470 USER32!_fnDWORDOPTINLPMSG
00000000'77b49528  00000000'77b0f878 USER32!_fnINOUTDRAG
00000000'77b49530  00000000'77ae85a0 USER32!_fnGETTEXTLENGTHS
00000000'77b49538  00000000'77b0fb9c USER32!_fnINCNTOUTSTRING
...
```

Listing 4: User-mode callback function dispatch table in USER32.dll

Upon invoking a system call, nt!KiSystemService or nt!KiFastCallEntry stores a trap frame (TRAP_FRAME) on the kernel thread stack to save the current thread context and be able to restore registers upon returning to user-mode. In order to make the transition back to user-mode in a user-mode callback, KeUserModeCallback first copies the input buffer to the user-mode stack using the trap frame information held by the thread object. It then creates a new trap frame with EIP set to ntdll!KiUserCallbackDispatcher, replaces the thread object's TrapFrame pointer, and finally calls nt!KiServiceExit to return execution to the user-mode callback dispatcher.

As user-mode callbacks need a place to store the thread state information such as the trap frame, Windows XP and 2003 would grow the kernel stack in order to ensure that enough space was available. However, because stack space can quickly be exhausted by calling callbacks recursively, Vista and Windows 7 moved to create a new kernel thread stack on each user-mode callback. In order to keep track of the previous stacks and so on, Windows reserves space for a KSTACK_AREA structure (Listing 5) at the base of the stack, followed by the forged trap frame.

```
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
```

Listing 5: Stack area and stack control structures

Once a user-mode callback has completed, it calls NtCallbackReturn (Listing 6) to resume execution in the kernel. This function copies the result of the callback back to the original kernel stack and restores the original trap frame (`PreviousTrapFrame`) and kernel stack by using the information held in the `KERNEL_STACK_CONTROL` structure. Before jumping to the location where it previously left off (in `nt!KiCallUserMode`), the kernel callback stack is deleted.

```
NTSTATUS NtCallbackReturn (
    IN PVOID Result OPTIONAL,
    IN ULONG ResultLength,
    IN NTSTATUS Status );
```

Listing 6: NtCallbackReturn

As recursive or nested callbacks could cause the kernel stack to grow infinitely (XP) or create an arbitrary number of stacks, the kernel keeps track of the callback depth (kernel stack space used by user-mode callbacks in total) for every running thread in the thread object structure (`KTHREAD->CallbackDepth`). Upon each callback, the bytes already used on a thread stack (stack base - stack pointer) are added to the `CallbackDepth` variable. Whenever the kernel attempts to migrate to a new stack, `nt!KiMigrateToNewKernelStack` ensures that the total `CallbackDepth` never exceeds 0xC000 bytes, or otherwise returns a `STATUS_STACK_OVERFLOW` error code.

3 Kernel Attacks through User-Mode Callbacks

In this section, we present several attack vectors that may allow an adversary to perform privilege escalation attacks from user-mode callbacks. We begin by looking at how user-mode callbacks deal with the user critical section, before discussing each attack vector in more detail.

3.1 Win32k Naming Convention

As described in Section 2.2, the Window Manager uses critical sections and global locking when operating on internal management structures. As user-mode callbacks could potentially allow applications to freeze the GUI subsystem, win32k always leaves the critical section before calling back into user-mode. This way, win32k may perform other tasks while user-mode code is being executed. Upon returning from the callback, win32k re-enters the critical section before the function resumes execution in the kernel. We can observe this behavior in any function that calls `KeUserModeCallback`, such as the one shown in Listing 7.

Upon returning from a user-mode callback, win32k must ensure that referenced objects and data structures are still in the excepted state. As the critical

```
call    _UserSessionSwitchLeaveCrit@0
lea     eax, [ebp+var_4]
push    eax
lea     eax, [ebp+var_8]
push    eax
push    0
push    0
push    43h
call    ds:__imp__KeUserModeCallback@20
call    _UserEnterUserCritSec@0
```

Listing 7: Leaving the critical section before a user-mode callback

section is left before entering a callback, user-mode code is free to alter the properties of objects, reallocate arrays, and so on. For instance, a callback could call `SetParent()` to change the parent of a window. If the kernel stores a reference to the parent before invoking a callback and continues to operate on it after returning without performing the proper checks or object locking, this could open up to security vulnerabilities.

As it's very important to keep track of the functions that potentially make calls back to user-mode (in order for developers to take the necessary precautions), win32k.sys uses its own function naming convention. In particular, functions are prefixed `xxx` or `zzz` depending on how they may invoke a user-mode callback. Functions prefixed `xxx` will in most cases leave the critical section and invoke a user-mode callback. However, in some cases the function might require a specific set of arguments in order to branch to the path where the callback is actually invoked. This is why you'll sometimes see non-prefixed functions call `xxx` functions, because the arguments they provide to the `xxx` function never results in a callback.

Functions prefixed `zzz` invoke asynchronous or deferred callbacks. This is typically the case with certain types of window events that for various reasons cannot or should not be processed immediately. In this case, win32k calls `xxxFlushDeferredWindowEvents` to flush the event queue. An important thing to note about `zzz` functions is that they require `win32k!gdwDeferWinEvent` to be non-null before calling `xxxWindowEvent`. If this is not the case, the callback is processed immediately.

The problem with the naming convention used by win32k is the lack of consistency. Several functions in win32k invoke callbacks, but are not labeled as they should. The reason for this is unclear, but one possible explanation can be that functions have been modified over time without the necessary updates made to the function names. Consequently, developers may be led into thinking that a function may never actually invoke a callback, hence avoids making the seemingly unnecessary validation (e.g. objects remain unlocked and pointers are not

Windows 7 RTM	Windows 7 (MS11-034)
`MNRecalcTabStrings`	`xxxMNRecalcTabStrings`
`FreeDDEHandle`	`xxxFreeDDEHandle`
`ClientFreeDDEHandle`	`xxxClientFreeDDEHandle`
`ClientGetDDEFlags`	`xxxClientGetDDEFlags`
`ClientGetDDEHookData`	`xxxClientGetDDEHookData`

Table 2. Functions prefixed properly as a result of MS11-034

revalidated). In addressing the vulnerabilities of MS11-034 [7], several function names were updated to properly reflect their use of user-mode callbacks (Table 2).

3.2 User Object Locking

As explained in Section 2.2, user objects implement reference counting to keep track of when objects are used and should be freed from memory. As such, objects expected to be valid after the kernel leaves the user critical section must be locked. Generally, there's two forms of locking – thread locking and assignment locking.

Thread Locking Thread locking is generally used to lock objects or buffers within a function. Each thread locked entry is stored in a thread lock structure (`win32k!_TL`) in a singly linked thread lock list, pointed to by the thread information structure (`THREADINFO.ptl`). The thread lock list operates much like a FIFO queue in which entries are pushed and pop'ed off the list. In win32k, thread locking is usually performed inline, and can be recognized by inlined pointer increments, normally before an `xxx` function is called (Listing 8). When a given function in win32k no longer needs the object or buffer, it calls `ThreadUnlock()` to remove the locked entry from the thread lock list.

In the event that objects have been locked but not unlocked properly (e.g. due to a process termination while processing a user-mode callback), win32k processes the thread lock list to release any remaining entries on thread termination. This can be observed in the `xxxDestroyThreadInfo` function in making the call to `DestroyThreadsObjects`.

Assignment Locking Unlike thread locking, assignment locking is used for more long-term locking of user objects. For instance, when creating a window inside a desktop, win32k assignment locks the desktop object at the proper offset in the window object structure. Rather than operating on lists, assignment locked entries are simply pointers (to the locked object) stored in memory. If a pointer already exists at the place where win32k needs to assignment lock an object, the module unlocks the existing entry before locking and replacing it with the one requested.

```
mov     ecx, _gptiCurrent
add     ecx, tagTHREADINFO.ptl ; thread lock list
mov     edx, [ecx]
mov     [ebp+tl.next], edx
lea     edx, [ebp+tl]
mov     [ecx], edx      ; push new entry on list
mov     [ebp+tl.pobj], eax ; window object
inc     [eax+tagWND.head.cLockObj]
push    [ebp+arg_8]
push    [ebp+arg_4]
push    eax
call    _xxxDragDetect@12 ; xxxDragDetect(x,x,x)
mov     esi, eax
call    _ThreadUnlock1@0 ; ThreadUnlock1()
```

Listing 8: Thread locking and release in win32k

The handle manager provides functions for assignment locking and unlocking. In locking an object, win32k calls `HMAssignmentLock(Address,Object)` and similarly `HMAssignmentUnlock(Address)` for releasing the object reference. Notably, assignment locking does not provide the safety net that thread locking does. Should a thread be terminated in a callback, the thread or user object cleanup routine itself is responsible for releasing these references individually. Failure to do so could result in memory leaks or the reference count could overflow[4] if the operation can be repeated arbitrarily.

Window Object Use-After-Free (CVE-2011-1237) In installing a computer-based training (CBT) hook, an application may receive various notifications about window handling, keyboard and mouse input, and message queue processing. For instance, before a new window is created, the `HCBT_CREATEWND` callback allows an application to inspect and modify the parameters used in determining the size and orientation of the window using the provided `CBT_CREATEWND`[5] structure. This structure also allows the application to choose the z-order of the window, by providing the handle to the window after which the new window will be inserted (`hwndInsertAfter`). In setting this handle, `xxxCreateWindowEx` obtains the corresponding object pointer and later uses it when linking the new window into the z-order chain. However, as the function failed to properly lock this pointer, an attacker could destroy the window provided in `hwndInsertAfter`

[4] On 64-bit platforms, this seems practically infeasible because of 64-bit length of the object `PointerCount` field.

[5] http://msdn.microsoft.com/en-us/library/ms644962(v=vs.85).aspx

in a subsequent callback and coerce win32k to operate on freed memory upon return.

In Listing 9, `xxxCreateWindowEx` calls `PWInsertAfter` to get the window object pointer (using `HMValidateHandleNoSecure`) for the `hwndInsertAfter` handle provided in the `CBT_CREATEWND` hook structure. The function then stores the object pointer in a local variable.

```
.text:BF892EA1       push    [ebp+cbt.hwndInsertAfter]
.text:BF892EA4       call    _PWInsertAfter@4 ; PWInsertAfter(x)
.text:BF892EA9       mov     [ebp+pwndInsertAfter], eax ; window object
```

Listing 9: Getting window object from CBT structure

As win32k does not lock `pwndInsertAfter`, an attacker could free the window supplied in the CBT hook in a subsequent callback (e.g. by calling `DestroyWindow`). At the end of the function (Listing 10), `xxxCreateWindowEx` uses the window object pointer and attempts to link it into (via `LinkWindow`) the window z-order chain. As the window object may no longer exist, this becomes a use-after-free vulnerability which may allow an attacker to execute arbitrary code in the context of the kernel. We discuss exploitation of user-after-free vulnerabilities affecting user objects in Section 4.

```
.text:BF893924        push    esi             ; parent window
.text:BF893925        push    [ebp+pwndInsertAfter]
.text:BF893928        push    ebx             ; new window
.text:BF893929        call    _LinkWindow@12  ; LinkWindow(x,x,x)
```

Listing 10: Linking into z-order chain

Keyboard Layout Object Use-After-Free (CVE-2011-1241) Keyboard layout objects are used in setting the active keyboard layout for a thread or process. In loading a keyboard layout, an application calls `LoadKeyboardLayout` and specifies the name of the input local identifier to load. Windows also provides the undocumented `LoadKeyboardLayoutEx` function, which takes an additional keyboard layout handle argument that win32k first attempts to unload before loading the new layout. In providing this handle, win32k failed to lock the corresponding keyboard layout object. Thus, an attacker could unload the provided keyboard layout in a user-mode callback and trigger a use-after-free condition.

In Listing 11, `LoadKeyboardLayoutEx` takes the handle of the keyboard layout to first unload and calls `HKLToPKL` to get the keyboard layout object pointer. `HKLtoPKL` traverses the list of active keyboard layouts (`THREADINFO.spklActive`) until it finds the one matching the supplied handle. `LoadKeyboardLayoutEx` then stores the object pointer in a local variable on the stack.

```
.text:BF8150C7        push    [ebp+hkl]
.text:BF8150CA        push    edi
.text:BF8150CB        call    _HKLtoPKL@8   ; get keyboard layout object
.text:BF8150D0        mov     ebx, eax
.text:BF8150D2        mov     [ebp+pkl], ebx  ; store pointer
```

Listing 11: Converting keyboard object handle to pointer

As `LoadKeyboardLayoutEx` did not sufficiently lock the keyboard layout object pointer, an attacker could unload the keyboard layout in a user-mode callback and thus free the object. This was possible as the function subsequently called `xxxClientGetCharsetInfo` to retrieve character set information from user-mode. In Listing 12, `LoadKeyboardLayoutEx` continues to use the keyboard layout object pointer previously stored, hence could be operating on freed memory.

```
.text:BF8153FC        mov     ebx, [ebp+pkl] ; KL object pointer

.text:BF81541D        mov     eax, [edi+tagTHREADINFO.ptl]
.text:BF815423        mov     [ebp+tl.next], eax
.text:BF815426        lea     eax, [ebp+tl]
.text:BF815429        push    ebx
.text:BF81542A        mov     [edi+tagTHREADINFO.ptl], eax
.text:BF815430        inc     [ebx+tagKL.head.cLockObj] ; freed memory ?
```

Listing 12: Using keyboard layout object pointer after user-mode callbacks

3.3 Object State Validation

In order to keep track of how objects are used, win32k associates several flags as well as pointers with user objects. Objects assumed to be in a certain state, should always have their state validated. User-mode callbacks could potentially alter the state and update properties of objects, such as changing the parent of a window, causing a drop down menu to no longer be active, or terminating the partner in a DDE conversation. Lack of state checking could result in bugs such as `NULL` pointer dereferences and use-after-frees, depending on how win32k uses the object.

DDE Conversation State Vulnerabilities The Dynamic Data Exchange (DDE) protocol is a legacy protocol using messages and shared memory to exchange data between applications. A DDE conversation is internally represented by the Window Manger as a DDE conversation object, one defined for both the sender and receiver. In order to keep track of which and to whom objects are engaged in a conversation with, the conversation object structure (undocumented) holds a pointer to the conversation object of the opposite party (using assignment locking). Thus, if either window or thread owning the conversation object terminates, its assignment locked pointer in the partner object is unlocked (cleared).

As DDE conversations store data in user-mode, they rely on user-mode callbacks to copy data to and from user-mode. Upon sending a DDE message, win32k calls `xxxCopyDdeIn` to copy the data in from user-mode. Similarly, in receiving a DDE message, win32k calls `xxxCopyDDEOut` to copy the data back out to user-mode. After the copy has taken place, win32k may notify the partner conversation object to act on the data, e.g. if it expects a response.

```
.text:BF8FB8A7        push    eax
.text:BF8FB8A8        push    dword ptr [edi]
.text:BF8FB8AA        call    _xxxCopyDdeIn@16
.text:BF8FB8AF        mov     ebx, eax
.text:BF8FB8B1        cmp     ebx, 2
.text:BF8FB8B4        jnz     short loc_BF8FB8FC

.text:BF8FB8C5        push    0               ; int
.text:BF8FB8C7        push    [ebp+arg_4]     ; int
.text:BF8FB8CA        push    offset _xxxExecuteAck@12
.text:BF8FB8CF        push    dword ptr [esi+10h] ; conversation object
.text:BF8FB8D2        call    _AnticipatePost@24
```

Listing 13: Absent check in conversation object handling

After processing user-mode callbacks to copy data in or out from user-mode, several functions failed to properly revalidate the partner conversation object. An attacker could terminate a conversation in a user-mode callback and thus unlock the partner conversation object from the sender's or receiver's object structure. In Listing 13, we see that the a callback may be invoked in `xxxCopyDdeIn`, but the function fails to revalidate the partner conversation object pointer before passing it to `AnticipatePost`. This in turn results in a `NULL` pointer dereference and allows an attacker to control the conversation object in mapping the null page (see Section 4.3).

Menu State Handling Vulnerabilities Menu management is one of the most complex components of win32k and holds uncharted code presumably dating back to the early days of the modern Windows operating system. Although menu objects (`tagMENU`) themselves are fairly simplistic and only contain information related to the actual menu items, menu handling as a whole depends on multiple fairly complex functions and structures. For instance, in creating popup menus, applications call `TrackPopupMenuEx`[6] to create a menu classed window in which the menu content is displayed. The menu window then processes message input through a system-defined menu window class procedure (`win32k!xxxMenuWindowProc`), in order to handle various menu specific messages. Moreover, in order to keep track of how a menu is used, win32k also associates a menu state structure (`tagMENUSTATE`) with the currently active menu. This way, functions can be aware of whether a menu is involved in a drag and drop operation, inside a menu loop, about to be terminated, and so on.

```
push    [esi+tagMENUSTATE.pGlobalPopupMenu]
or      [esi+tagMENUSTATE._bf4], 200h ; fInCallHandleMenuMessages
push    esi
lea     eax, [ebp+var_1C]
push    eax
mov     [ebp+var_C], edi
mov     [ebp+var_8], edi
call    _xxxHandleMenuMessages@12 ; xxxHandleMenuMessages(x,x,x)
and     [esi+tagMENUSTATE._bf4], 0FFFFFDFFh ; <-- may have been freed
mov     ebx, eax
mov     eax, [esi+tagMENUSTATE._bf4]
cmp     ebx, edi
jz      short loc_BF968B0B ; message processed?
```

Listing 14: Use-after-free in menu state handling

[6] http://msdn.microsoft.com/en-us/library/ms648003(v=vs.85).aspx

In processing various types of menu messages, win32k did not properly validate menus after user-mode callbacks. Specifically, in closing a menu (e.g. by sending the `MN_ENDMENU` message to the menu window class procedure) while processing a callback, win32k would in many cases fail to properly check if the menu state was still active or if the object pointers referenced by related structures such as the popup menu structure (`win32k!tagPOPUPMENU`) were non-null. In Listing 14, win32k attempts to handle certain types of menu messages by calling `xxxHandleMenuMessages`. As this function may invoke a callback, subsequent use of the menu state pointer (`ESI`) would cause win32k to operate on freed memory. This particular case could have been avoided by locking the menu state using the `dwLockCount` variable of the `tagMENUSTATE` structure (not listed).

3.4 Buffer Reallocation

Many user objects have item arrays or other forms of buffers associated with them. Item arrays where elements are added or removed are usually resized to conserve memory. For instance, if the number of elements go above or below a certain threshold the buffer is reallocated with a more suitable size. Similarly, if an array is emptied, the buffer is freed. Importantly, any buffer that can be reallocated or freed during a callback must be rechecked upon return (Figure 1). Any function failing to do this could potentially be operating on freed memory, hence allow an attacker to control assignment locked pointers or corrupt the memory of subsequent allocations.

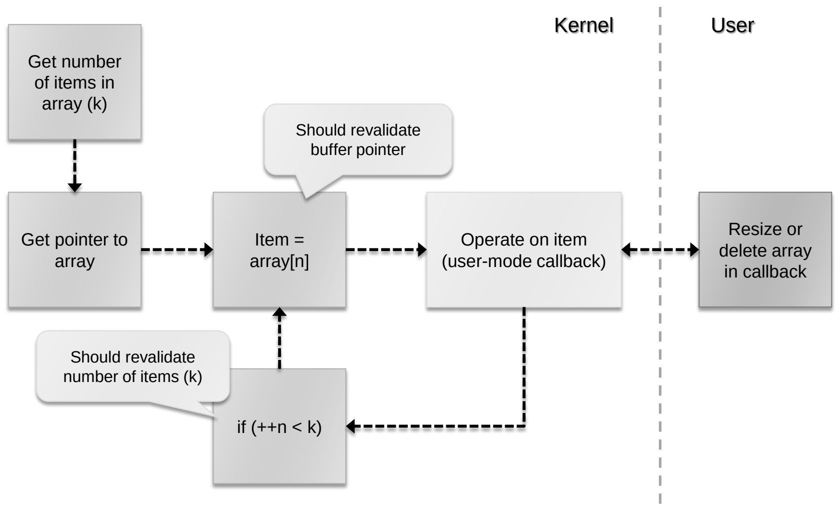
Kernel
User
Get number of items in array (k)
Should revalidate buffer pointer
Get pointer to array
Item = array[n]
Operate on item (user-mode callback)
Resize or delete array in callback
Should revalidate number of items (k)
if (++n < k)

Fig. 1. Buffer reallocation

Menu Item Array Use-After-Free In order to keep track of the menu items held by popup or drop down menu, menu objects (`win32k!tagMENU`) define a pointer (`rgItems`) to the array of menu items. Each menu item (`win32k!tagITEM`) defines properties such as the displayed text string, embedded image, pointer to submenu and so on. The menu object structure keeps track of the number of items contained by the array in the `cItems` variable, and also how many items that may fit in the allocated buffer in the `cAlloced` variable. In adding or removing elements from the menu items array, for instance by calling `InsertMenuItem()` or `DeleteMenu()`, win32k attempts to resize the array if it notices that `cAlloced` is about to become less than `cItems` (Figure 2), or if the difference between `cItems` and `cAllocated` is more than 8 items.

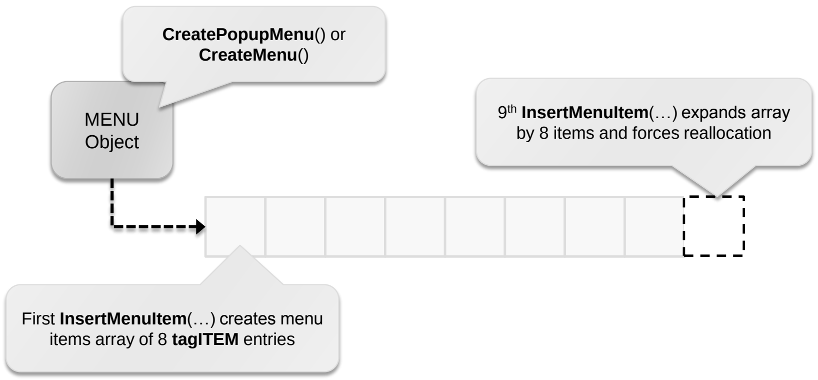
CreatePopupMenu() or **CreateMenu**()
MENU Object
9th **InsertMenuItem**(…) expands array by 8 items and forces reallocation
First **InsertMenuItem**(…) creates menu items array of 8 **tagITEM** entries

Fig. 2. Menu items array reallocation

Several functions inside win32k did not sufficiently validate the menu item array buffer after user-mode callbacks. As there is no way to "lock" a menu item, such as the case is with user objects, any function that could invoke a callback would be required to revalidate the menu item array. This also applies to functions that take menu items as arguments. If the menu item array buffer is reallocated in a user-mode callback, subsequent code could be operating on freed memory or data controlled by the attacker.

`SetMenuInfo` allows applications to set various properties of a specified menu. In setting the `MIM_APPLYTOSUBMENUS` flag mask value in the provided menu information structure (`MENUINFO`), win32k also applies the updates to all of a menu's submenus. This behavior can be observed in `xxxSetMenuInfo` as the function iterates over each menu item entry and recursively processes each submenu to propagate the updated settings. Before processing the menu items array and making any recursive calls, `xxxSetMenuInfo` stores the number of menu items (`cItems`) as well as the menu items array pointer (`rgItems`) in local variables/registers (Listing 15).

```
.text:BF89C779        mov     eax, [esi+tagMENU.cItems]
.text:BF89C77C        mov     ebx, [esi+tagMENU.rgItems]
.text:BF89C77F        mov     [ebp+cItems], eax
.text:BF89C782        cmp     eax, edx
.text:BF89C784        jz      short loc_BF89C7CC
```

Listing 15: Storing number of menu items and array pointer

Once `xxxSetMenuInfo` has reached the innermost menu, the recursion stops and the entry is processed. At this point, the function may invoke a user-mode callback in calling `xxxMNUpdateShownMenu`, hence could possibly allow the menu item array to be resized. However, as `xxxMNUpdateShownMenu` returns and upon returning from the recursive call, `xxxSetMenuInfo` fails to sufficiently validate the menu item array buffer as well as the number of items held by the array. If an attacker resizes the menu items array by calling `InsertMenuItem()` or `DeleteMenu()` from within the callback invoked by `xxxMNUpdateShownMenu`, `ebx` in Listing 16 may point to freed memory. Moreover, as `cItems` reflects the number of elements contained by the array at the point where the function was called, `xxxSetMenuInfo` may operate on items outside the allocated array.

```
.text:BF89C786        add     ebx, tagITEM.spSubMenu

.text:BF89C789        mov     eax, [ebx]        ; spSubMenu
.text:BF89C78B        dec     [ebp+cItems]
.text:BF89C78E        cmp     eax, edx
.text:BF89C790        jz      short loc_BF89C7C4
...
.text:BF89C7B2        push    edi
.text:BF89C7B3        push    dword ptr [ebx]
.text:BF89C7B5        call    _xxxSetMenuInfo@8 ; xxxSetMenuInfo(x,x)
.text:BF89C7BA        call    _ThreadUnlock1@0  ; ThreadUnlock1()
.text:BF89C7BF        xor     ecx, ecx
.text:BF89C7C1        inc     ecx
.text:BF89C7C2        xor     edx, edx
...
.text:BF89C7C4        add     ebx, 6Ch          ; next menu item
.text:BF89C7C7        cmp     [ebp+cItems], edx ; more items ?
.text:BF89C7CA        jnz     short loc_BF89C789
```

Listing 16: Insufficient buffer validation after user-mode callbacks

In order to address vulnerabilities involving processing of menu items, Microsoft introduced the new `MNGetpItemFromIndex` function in win32k. This function takes the menu object pointer and requested menu item index as arguments and returns the item based on the information provided in the menu object.

SetWindowPos Array Use-After-Free Windows allows applications to defer window position updates such that multiple windows can be updated at the same time. For this, Windows uses a special SetWindowsPos object that holds a pointer to an array of window position structures. The SWP object as well as this array is initialized when the application calls `BeginDeferWindowPos()`. This function takes the number of array elements (window position structures) to preallocate. Window position updates are then deferred by calling `DeferWindowPos()`, in which the next available position structure is filled. Should the requested number of deferred updates exceed the number of preallocated entries, win32k reallocates the array with a more suitable size (4 additional entries). Once all the requested window position updates have been deferred, the application calls `EndDeferWindowPos()` to process the list of windows to update.

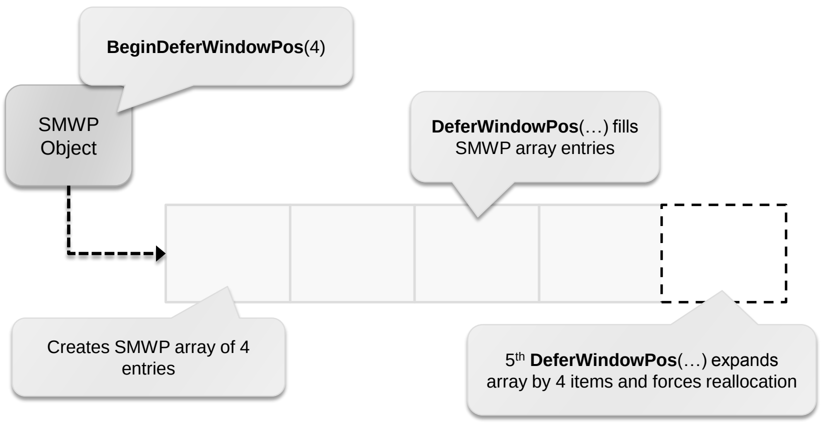
BeginDeferWindowPos(4)
SMWP Object
DeferWindowPos(…) fills SMWP array entries
Creates SMWP array of 4 entries
5th DeferWindowPos(…) expands array by 4 items and forces reallocation

Fig. 3. SMWP array reallocation

In operating on the SMWP array, win32k did not always properly validate the array pointer after user-mode callbacks. In calling `EndDeferWindowPos` to process the multiple window position structure, win32k calls `xxxCalcValidRects` to calculate the position and size of each window referenced in the SMWP array. This function iterates over each entry and performs various operations such as notifying each window that its position is changing (`WM_WINDOWPOSCHANGING`). As this message may invoke a user-mode callback, an attacker could make multiple `DeferWindowPos` calls on the same SWP object in order to cause the SMWP array to be reallocated (Listing 17). This would in turn result in a use-after-free as `xxxCalcValidRects` writes the window handle back into the original buffer.

```
.text:BF8A37B8        mov     ebx, [esi+14h]  ; SMWP array
.text:BF8A37BB        mov     [ebp+var_20], 1
.text:BF8A37C2        mov     [ebp+cItems], eax ; SMWP array count
.text:BF8A37C5        js      loc_BF8A3DE3    ; exit if no entries
...
.text:BF8A3839        push    ebx
.text:BF8A383A        push    eax
.text:BF8A383B        push    WM_WINDOWPOSCHANGING
.text:BF8A383D        push    esi
.text:BF8A383E        call    _xxxSendMessage@16 ; user-mode callback
.text:BF8A3843        mov     eax, [ebx+4]
.text:BF8A3846        mov     [ebx], edi        ; window handle
...
.text:BF8A3DD7        add     ebx, 60h          ; get next entry
.text:BF8A3DDA        dec     [ebp+cItems]      ; decrement cItems
.text:BF8A3DDD        jns     loc_BF8A37CB
```

Listing 17: Insufficient pointer and size validation in `xxxCalcValidRects`

Unlike menu items, vulnerabilities involving SMWP array handling were addressed by disallowing buffer reallocation while the SMWP array is being processed. This can be seen in `win32k!DeferWindowPos`, where the function now checks for a "being processed" flag and only allows entries to be added if it doesn't result in a buffer reallocation.

4 Exploitability

In this section, we evaluate the exploitability of vulnerabilities triggered by user-mode callbacks. As we're mostly concerned with two vulnerability primitives – use-after-frees and `NULL` pointer dereferences – we'll focus on how an attacker may be able to leverage such bug classes in exploiting win32k vulnerabilities. Assessing their exploitability is vital in order to propose reasonable mitigations or workarounds in Section 5.

4.1 Kernel Heap

As mentioned in Section 2.2, user objects and their associated data structures are either stored in the session pool, the shared heap, or the desktop heap. Objects and data structures stored in the desktop heap or the shared heap are managed by the *kernel heap allocator*. The kernel heap allocator can be considered a stripped down version of the user-mode heap allocator, and uses familiar functions such as `RtlAllocateHeap` and `RtlFreeHeap` exported by the NT executive in managing heap blocks.

Although the user and kernel heaps are strikingly similar, there are some key differences. Unlike the user-mode heap, kernel heaps as used by win32k do not employ any front end allocators. This can be observed by looking at the ExtendedLookup value of the HEAP_LIST_LOOKUP structure, referenced by the heap base (HEAP). When set to null, the heap allocator does not use any lookaside lists or low fragmentation heaps [13]. Furthermore, in dumping the heap base structure (Listing 18), we can observe that no encoding or obfuscation of heap management structures is used as both EncodingFlagMask and PointerKey are set to null. The former decides if heap header encoding should be used, while the latter is used for encoding the CommitRoutine pointer, called whenever the heap needs to be extended.

```
kd> dt nt!_HEAP fea00000
   ...
   +0x04c EncodeFlagMask   : 0
   +0x050 Encoding         : _HEAP_ENTRY
   +0x058 PointerKey       : 0
   ...
   +0x0b8 BlocksIndex      : 0xfea00138 Void
   ...
   +0x0c4 FreeLists        : _LIST_ENTRY [ 0xfea07f10 - 0xfea0e4d0 ]
   ...
   +0x0d0 CommitRoutine    : 0x93a4692d  win32k!UserCommitDesktopMemory
   +0x0d4 FrontEndHeap     : (null)
   +0x0d8 FrontHeapLockCount : 0
   +0x0da FrontEndHeapType : 0 ''

kd> dt nt!_HEAP_LIST_LOOKUP fea00138
   +0x000 ExtendedLookup   : (null)
   ...
```

Listing 18: Desktop heap base and BlocksIndex structures

When dealing with kernel heap corruptions such as use-after-frees, it is vital to know exactly how the kernel heap manager works. There are many great papers detailing the inner workings of the user-mode heap implementation [13][6][9] which may be used as reference when studying the kernel heap. For the purpose of this discussion, it is sufficient to understand that the kernel heap is one contiguous piece of memory that can be extended or shrunk depending on the amount of memory allocated. As no front-end managers are used, all the free blocks are indexed into a single free list. As a general rule, the heap manager always tries to allocate the most recently freed block (e.g. through the use of list hints) in order to better make use of the CPU cache.

4.2 Use-After-Free Exploitation

In order to exploit use-after-free vulnerabilities in win32k, an attacker needs to be able to reallocate the freed memory and to a certain degree control its content. Because user objects and associated data structures are stored together with strings, it is possible to force arbitrarily sized allocations and fully control the content of recently freed memory by setting object properties that are stored as Unicode strings. As long as `WORD NULL`s are avoided (except for the string terminator), any byte combination can be used in manipulating memory accessed as objects or data structures.

For use-after-free vulnerabilities in the desktop heap, an attacker may set the text of a window's title bar using `SetWindowTextW` to force arbitrarily sized desktop heap allocations. Similarly, arbitrarily sized session pool allocations can be triggered by calling `SetClassLongPtr` and specifying `GCLP_MENUNAME` to set the menu name string of a menu resource associated with a window class.

```
eax=41414141 ebx=00000000 ecx=ffb137e0 edx=8e135f00 esi=fe74aa60 edi=fe964d60
eip=92d05f53 esp=807d28d4 ebp=807d28f0 iopl=0         nv up ei pl nz na pe cy
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010207
win32k!xxxSetPKLinThreads+0xa9:
92d05f53 89700c   mov   dword ptr [eax+0Ch],esi ds:0023:4141414d=????????

kd> dt win32k!tagKL @edi -b
   +0x000 head            : _HEAD
      +0x000 h               : 0x41414141
      +0x004 cLockObj        : 0x41414142
   +0x008 pklNext         : 0x41414141
   +0x00c pklPrev         : 0x41414141
   ...
```

Listing 19: String as keyboard layout object (CVE-2011-1241)

In Listing 19 (showing the vulnerability described in Section 3.2), the keyboard layout object has been replaced by a user controlled string allocated in the desktop heap. In this particular case, the keyboard layout object has been freed, but win32k attempts to link it into the keyboard layout object list. This allows the attacker to choose the address where `esi` is written by controlling the `pklNext` pointer of the freed keyboard layout object.

As objects often contain pointers to other objects, win32k uses assignment locking to ensure that object dependencies are satisfied. As such, use-after-frees affecting objects whose body contain an assignment locked pointer may allow an attacker to decrement an arbitrary address as win32k attempts to release the object reference. One possible way of leveraging this is a variation of an attack described in [11], in which a destroyed menu handle index was returned from a user-mode callback. Upon thread termination, this lead to the destroy routine of the free type (0) to be called. As the free type does not define a destroy routine, win32k would call the null page which users are allowed to map on Windows (see Section 4.3).

```
eax=deadbeeb ebx=fe954990 ecx=ff910000 edx=fea11888 esi=fea11888 edi=deadbeeb
eip=92cfc55e esp=965a1ca0 ebp=965a1ca0 iopl=0         nv up ei ng nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010286
win32k!HMUnlockObject+0x8:
92cfc55e ff4804          dec     dword ptr [eax+4]    ds:0023:deadbeef=????????

965a1ca0 92cfc9e0 deadbeeb 00000000 fe954978 win32k!HMUnlockObject+0x8
965a1cb0 92c60cb1 92c60b8b 004cfa54 002dfec4 win32k!HMAssignmentLock+0x45
965a1cc8 92c60bb3 965a1cfc 965a1cf8 965a1cf4 win32k!xxxCsDdeInitialize+0x67
965a1d18 8284942a 004cfa54 004cfa64 004cfa5c win32k!NtUserDdeInitialize+0x28
965a1d18 779864f4 004cfa54 004cfa64 004cfa5c nt!KiFastCallEntry+0x12a
```

Listing 20: String as DDE object (CVE-2011-1242)

As an attacker may infer the address of the user handle table in kernel memory, he or she could decrement the type (`bType`) value of a window object handle table entry (1). Upon destroying the window, this would result in destroy routine for the free type (0) to be called and allow for arbitrary kernel code execution. In Listing 20, the attacker controls the assignment unlocked pointer, leading to arbitrary kernel decrement.

4.3 Null Pointer Exploitation

Unlike other platforms such as Linux, Windows (in staying true to backwards compatibility) allows non-privileged users to map the null page within the context of a user process. As kernel and user-mode components share the same virtual address space, an attacker may potentially be able to exploit kernel null dereference vulnerabilities by mapping the null page and controlling the dereferenced data. In order to allocate the null page on Windows, an application may simply call `NtAllocateVirtualMemory` and request a base address larger than null but less than the size of a page. Applications may also memory map the null page by calling `NtMapViewOfSection` using such a base address and the `MEM_DOS_LIM` compatibility flag to enable page aligned sections (x86 only).

Null pointer vulnerabilities in win32k are many times caused by insufficient checks in regards to user object pointers. Hence, the attacker may be able to exploit such vulnerabilities by creating fake null page objects and subsequently

```
pwnd = (PWND) 0;
pwnd->head.h = hWnd; // valid window handle
pwnd->head.pti = NtCurrentTeb()->Win32ThreadInfo;
pwnd->bServerSideWindowProc = TRUE;
pwnd->lpfnWndProc = (PVOID) xxxMyProc;
```

Listing 21: Setting up a fake window object at the null page

trigger arbitrary memory writes or control the value of a function pointer. For instance, as many of the recent null pointer vulnerabilities in win32k are concerned with window object pointers, an attacker could position a fake window object at the null page and define a custom server-side window procedure (Listing 21). This would allow the attacker to obtain arbitrary kernel code execution if any messages are later passed to the null object.

5 Mitigations

In this section, we evaluate ways of mitigating the vulnerability classes discussed in Section 4.

5.1 Use-After-Free Vulnerabilities

As mentioned in the previous section, use-after-free exploitability relies on the attacker's ability to reallocate and control the contents of the previously freed memory. Unfortunately, attempting to mitigate use-after-free vulnerabilities is very difficult as the CPU has no legitimate way of telling whether memory belongs to a particular object or data structure, as these are just abstractions made by the operating system. If we look more closely at the problem, these issues essentially boil down to the attacker being able to free an object or buffer while processing a callback, and then reallocate that memory before it is again used by win32k.sys upon callback return. Thus, it may be possible to mitigate exploitability of use-after-frees by reducing the predictability of kernel pool or heap allocations or by isolating certain allocations such that easily controllable primitivies such as strings are not allocated from the same resource as, say, user objects.

As the system always is aware of whenever callbacks are active (e.g. through `KTHREAD.CallbackDepth`), a delayed free approach can be used while processing a user-mode callback. This would prevent an attacker from immediately reusing the freed memory. However, such a mechanism would not counter exploitation in situations where multiple consecutive callbacks are invoked before the use-after-free condition is triggered. Additionally, as the user-mode callback mechanism is not implemented in win32k.sys, additional logic would have to be implemented upon callback return to perform the necessary delayed free list processing.

Rather than attempting to address use-after-free exploitation by focusing on allocation predictability, we can also look at how exploitation would typically be performed. As discussed in Section 4, unicode strings as well as allocations where a large portion of the data can be controlled (e.g. window objects with `cbWndExtra` defined) are very useful to an attacker. Hence, isolating such allocations could be used to prevent an attacker from using flexible primitives (e.g. strings) for easily reallocating the memory of freed objects.

5.2 Null Pointer Vulnerabilities

In order to address null pointer exploitation on Windows we need to deny user mode applications the ability to map and control the contents of the null page. Although there are multiple ways to approach this problem such as through system call hooking[7] or page table entry (PTE) modification, using virtual address descriptors (VADs) appears to be a more well suited solution [5]. As VADs describe the process memory space and provide Windows with the information needed to set up page table entries correctly, they can be used to prevent null page mappings in a uniform and generic way. However, preventing null page mappings also comes at the cost of backwards compatibility, as the NTVDM subsystem in 32-bit versions of Windows relies on this ability to properly support 16-bit executables.

6 Remarks

As we've shown in this paper, user-mode callbacks appear to have caused many problems and introduced many vulnerabilities in the win32k subsystem. This is partly because win32k, or the Window Manager specifically, was designed to use a global locking mechanism (the user critical section) to allow the module to be thread-safe. Although addressing these vulnerabilities on a case-by-case basis may suffice as a short-term solution, win32k will at some point require an overhaul in order to better support multicore architectures and provide better performance in window management. In the current design, no two threads in the same session can process their message queues simultaneously, even if they are in two separate applications on separate desktops. Ideally, win32k should follow the much more consistent design of the NT executive, and perform mutual exclusion on a per-object or per-structure basis.

An important step in mitigating exploitation in win32k and kernel exploitation in general on Windows, is to get rid of the shared memory sections between user and kernel-mode. Traditionally, these were seen as optimizations in that the Win32 subsystem would not need to resort to a system call, hence avoid the

[7] System call hooking is discouraged by Microsoft and cannot easily be used on 64-bit platforms due to the integrity checks enforced by Kernel Patch Protection.

overhead associated with them. Since this design decision was made, system calls no longer use the slower interrupt based approach, hence the performance gain is probably minimal. Although shared sections may still be preferred in some cases, the information shared should be kept at a bare minimum. Currently, the win32k subsystem provides an adversary with a tremendous amount of kernel address space information and also opens up to additional attack vectors as illustrated in the exploitation of a recent CSRSS vulnerability [4]. Because memory in the subsystem is shared between processes regardless of their privilege level, an attacker has the ability to manipulate the address space of a privileged process from a non-privileged process.

7 Conclusion

In this paper, we've discussed the many challenges and problems concerning user-mode callbacks in win32k. In particular, we've shown that the global locking design of the Window Manager does not integrate well with the concept of user-mode callbacks. Although a large amount of vulnerabilities involving insufficient validation around the use of user-mode callbacks have been addressed, the complex nature of some of these issues suggests that more subtle flaws are likely to still be present in win32k. Thus, in an effort to mitigate some of the more prevalent bug classes, we conclusively discussed some ideas as to what both Microsoft and end-users might do to reduce the risk of future attacks in the win32k subsystem.

References

[1] Edgar Barbosa: Windows Vista UIPI. `http://www.coseinc.com/en/index.php?rt=download&act=publication&file=Vista_UIPI.ppt.pdf`

[2] Alex Ionescu: Inside Session 0 Isolation and the UI Detection Service. `http://www.alex-ionescu.com/?p=59`

[3] ivanlef0u: You Failed! `http://www.ivanlef0u.tuxfamily.org/?p=68`

[4] Matthew 'j00ru' Jurczyk: CVE-2011-1281: A story of a Windows CSRSS Privilege Escalation vulnerability. `http://j00ru.vexillium.org/?p=893`

[5] Tarjei Mandt: Locking Down the Windows Kernel: Mitigating Null Pointer Exploitation. `http://mista.nu/blog/2011/07/07/mitigating-null-pointer-exploitation-on-windows/`

[6] John McDonald, Chris Valasek: Practical Windows XP/2003 Heap Exploitation. Black Hat Briefing USA 2009. `https://www.blackhat.com/presentations/bh-usa-09/MCDONALD/BHUSA09-McDonald-WindowsHeap-PAPER.pdf`

[7] Microsoft Security Bulletin MS11-034. Vulnerabilities in Windows Kernel-Mode Drivers Could Allow Elevation of Privilege. `http://www.microsoft.com/technet/security/bulletin/ms11-034.mspx`

[8] Microsoft Security Bulletin MS11-054. Vulnerabilities in Windows Kernel-Mode Drivers Could Allow Elevation of Privilege. `http://www.microsoft.com/technet/security/bulletin/ms11-054.mspx`

[9] Brett Moore: Heaps About Heaps. `http://www.insomniasec.com/publications/Heaps_About_Heaps.ppt`

[10] MS Windows NT Kernel-mode User and GDI White Paper. http://technet.microsoft.com/en-us/library/cc750820.aspx

[11] mxatone: Analyzing Local Privilege Escalations in Win32k. Uninformed Journal vol. 10. http://uninformed.org/?v=10&a=2

[12] Chris Paget: Click Next to Continue: Exploits & Information about Shatter Attacks. https://www.blackhat.com/presentations/bh-usa-03/bh-us-03-paget.pdf

[13] Chris Valasek: Understanding the Low Fragmentation Heap. Black Hat Briefings USA 2010. http://illmatics.com/Understanding_the_LFH.pdf