



Windows Kernel Information Leak

Investigating Microsoft Vulnerability CVE-2019-1169	2
Prerequisite Knowledge	2
Advisory Analysis	2
Analyzing the Public Advisories	3
NULL Pointer Dereference – Causes and Effects	3
Why More Versions of Windows Are Not Vulnerable	5
Verifying the NULL Page Mitigation Was Backported	6
Target Setup	10
Environment Setup – Snapshots and Folders	10
Symbol Path Setup	11
Setting Up VirtualBox for Kernel Debugging	12
Patch Diffing and Initial Analysis	17
Diaphora Analysis	17
xxxMNDragOver() Patch Analysis	23
Conclusion	25

Investigating Microsoft Vulnerability CVE-2019-1169

VerSprite Security Researchers are often learning new technologies and products to perform security testing for client. Part of this work involves researching past vulnerabilities to understand how various products, such as Windows, can be attacked.

As part of these efforts, [VerSprite's Research practice](#), VS-Labs, recently investigated CVE-2019-1169, a NULL pointer dereference vulnerability in `win32k.sys` that Microsoft fixed in the August 2019 patch update. This led to the creation of a working exploit which can successfully leak data from arbitrary kernel addresses on affected Windows 7 machines.

This report will walk through how VS-Labs created this exploit, starting with setting up a testing environment, before then moving on to analyzing the patches with Diaphora, and finally creating the exploit using C++ code. At the end of this report, readers should have a solid understanding of CVE-2019-1169.

Prerequisite Knowledge

Required:

- Knowledge of C/C++
- Knowledge of x86 assembly

Recommended:

- Prior experience with Windows userland exploitation.

Advisory Analysis

Analyzing the Public Advisories

To start off the analysis of CVE-2019-1169, the VS-Labs analyzed two separate advisories: one from Microsoft and one from ZDI. Upon initial analysis, VS-Labs immediately identified discrepancies between these two advisories.

Microsoft's [advisory page for CVE-2019-1169](#) listed the vulnerability as an arbitrary code execution bug affecting Windows 7, Windows 7 SP1, Windows Server 2008, and Windows Server 2008 R2 (note that Microsoft does not list affected operating systems which are no longer supported, such as Windows XP, although they are likely to be affected by the same vulnerability).

On the other hand, ZDI's advisory for [ZDI-19-709](#) described the vulnerability as a NULL pointer dereference vulnerability in `xxxMNDragOver()`.

This advisory also mentioned that the vulnerability can be triggered by destroying a menu during a callback, granting an attacker the ability to read kernel memory from user mode code.

At this point, one must wonder, "*Which of these two advisories is correct?*".

By conducting further research, VS-Labs was able to determine that ZDI's advisory was actually correct and Microsoft's advisory had actually bundled multiple advisories together and labeled them by the one with the worst severity, which may have given users the false impression that CVE-2019-1169 results in an attacker elevating their privileges.

Before discussing how VS-Labs conducted this analysis, it is necessary to provide a bit of background on how NULL pointer dereference bugs work.

NULL Pointer Dereference – Causes and Effects

NULL pointer dereference bugs occur because developers do not check whether a pointer is NULL before dereferencing the pointer to retrieve the data it points to. This is usually the result of a developer forgetting that one of the code paths within their code can alter an object or pointer into an unexpected state. This can result in the developer making incorrect assumptions about which checks are necessary to appropriately protect their program from malicious input.

The severity of NULL pointer dereference vulnerabilities depends on how the application uses the pointer once it has been dereferenced. If the data pointed to by the pointer is used as the location for a write operation, this will grant an attacker the ability to write to arbitrary memory, which can lead to code execution.

Similarly, if the pointer is used to determine where to read data from, an attacker may only be able to conduct an arbitrary read, which would make the only potential attack an information leak. In both these cases, an attacker needs to allocate the NULL page so that when the pointer is dereferenced, the affected program will reference attacker-controlled data within the allocated NULL page. This data will then be used within the affected application, which can allow an attacker to control program behavior.

When crafting the NULL page, the attacker must ensure the contents of the NULL page match the structure of the data that is being pointed to by the NULL pointer, or else they won't be able to control the application's data. It should be noted that this means that there is no generic approach to crafting the data for the NULL page; each affected application will likely require the NULL page to be populated with data unique to that application.

Finally, it is important to note that in this blog post only kernel mode NULL pointer dereference vulnerabilities will be discussed, however the same concepts apply to user mode NULL pointer dereference vulnerabilities. Of these two, the only difference is that kernel mode NULL pointer dereference vulnerabilities have a higher chance of allowing privilege escalation, as kernel mode code can read and write to any address on the system, whereas with user mode NULL pointer dereference bugs, it is only possible to read from or write to addresses within the user mode address space.

Why More Versions of Windows Are Not Vulnerable

An interesting observation can be made by reading the Microsoft advisory again: the **bug doesn't affect Windows 8, Windows 8.1, or Windows 10**.

The reason why these versions are not affected can be found on page 33 of a [2012 presentation](#) by Matt Miller of MSRC, which notes that on Windows 8 and later prevent users from being able to utilize the first 64KB of memory (`0x00000000` to `0x0000FFFF`), in an attempt to prevent NULL pointer dereference vulnerabilities from being exploitable.

This mitigation works as if kernel mode tries to allocate memory in this region, an access violation will be raised which will result in a BSOD, which can easily alert system administrators to the presence of an attacker within their networks.

Similarly, user mode applications will not be able to allocate memory in these regions and will instead just return an error.

A very important point to note here is that this mitigation **was backported** to 64-bit versions of Windows Vista and later. Therefore, it is only possible to exploit NULL pointer dereferences on the 32-bit versions of most Windows operating systems. But how does one verify whether this is indeed the case? With a little bit of programming of course!

Verifying the NULL Page Mitigation Was Backported

To test whether the NULL page mitigation had been backported to Windows 7 x64, VS-Labs used the following program, which attempts to allocate the NULL page by using `ZwVirtualAllocMemory()`.

NULLPageAllocation.cpp

```
// NULLPageAllocation.cpp: This file contains the 'main' function.
// Program execution begins and ends there.

#include <stdio.h>
#include <Windows.h>
#include <bcrypt.h> // Needed to solve the issue "Function returning function
                  // is not allowed". Might be because it defines
                  // NTSTATUS and some other data structures.

// Definition taken from NtAllocateVirtualMemory function (ntifs.h) - Windows
drivers
typedef NTSTATUS(WINAPI* ZwAllocateVirtualMemory)(HANDLE ProcessHandle, PVOID*
BaseAddress, ULONG_PTR ZeroBits, PSIZE_T RegionSize, ULONG AllocationType, ULONG
Protect);

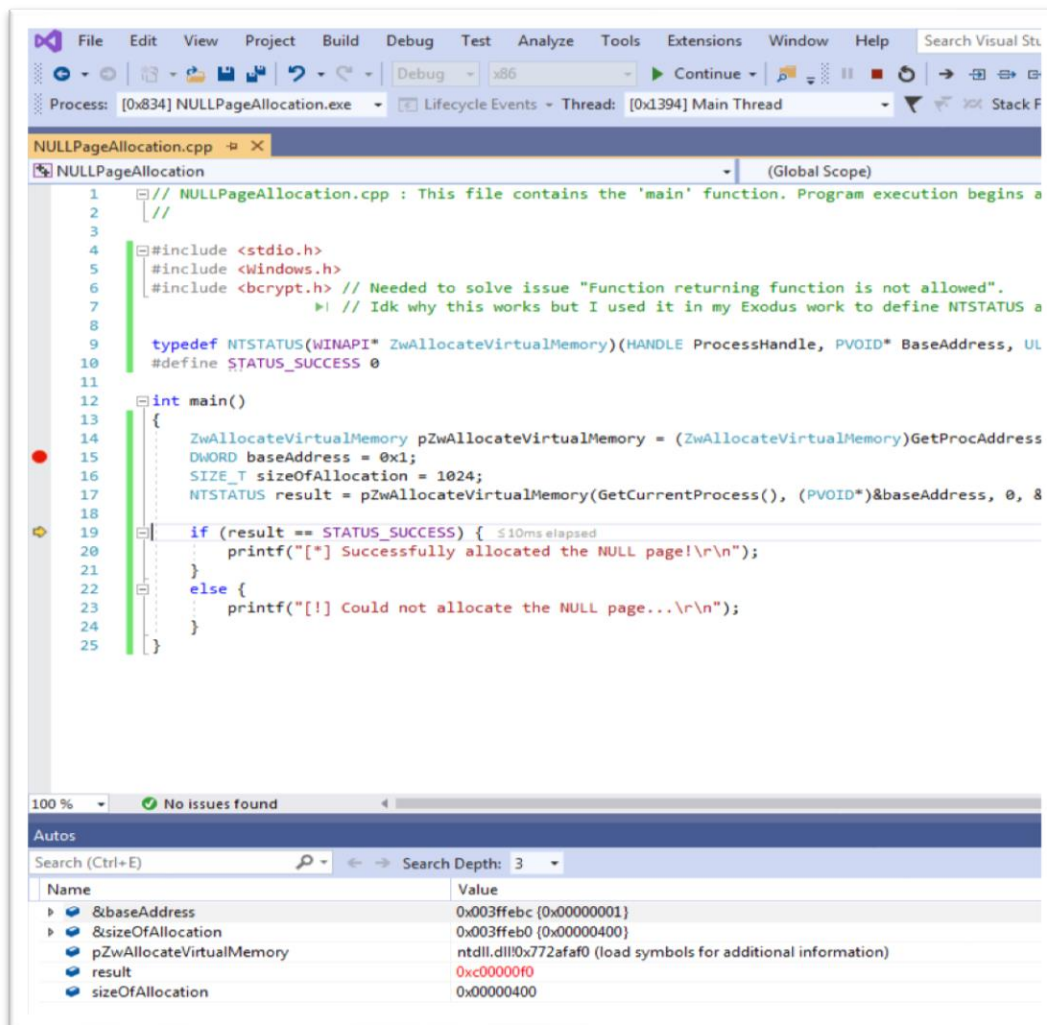
#define STATUS_SUCCESS 0

int main()
{
    // Get the address of NtAllocateVirtualMemory()
    // which is exported from ntdll.dll
    ZwAllocateVirtualMemory pZwAllocateVirtualMemory =
(ZwAllocateVirtualMemory)GetProcAddress(GetModuleHandle(L"ntdll.dll"),
"ZwAllocateVirtualMemory");
    DWORD baseAddress = 0x1;
    SIZE_T sizeOfAllocation = 1024;
    NTSTATUS result = pZwAllocateVirtualMemory(GetCurrentProcess(),
(PVOID*)&baseAddress, 0, &sizeOfAllocation, MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);

    if (result == STATUS_SUCCESS) {
        printf("[*] Successfully allocated the NULL page!\r\n");
    }
    else {
        printf("[!] Could not allocate the NULL page...\r\n");
    }
}
```

In this example, `0x1` is used as the base address because `ZwVirtualAllocMemory()` does not allow the address parameter to be `0x0`. However, if `0x1` is provided, `ZwVirtualAllocMemory()` will internally round `0x1` down to the next page aligned address, which is `0x0`. As a result, an attacker can request `ZwVirtualAllocMemory()` to try to allocate the NULL page without violating `ZwVirtualAllocMemory()`'s parameter requirements.

Note that attempting to do this trick with other memory allocation functions will not work as `ZwVirtualAllocMemory()` and its NT equivalent, `NtVirtualAllocMemory()`, have the unique ability to allocate memory at the NULL page, whereas other calls, like `VirtualAlloc()`, will **reject addresses smaller than a certain address** even after rounding them down to the nearest page boundary. Figure 1 shows the results of VS-Lab attempting to run this program on a Windows 7 SP1 x64 machine.



```

1  // NULLPageAllocation.cpp : This file contains the 'main' function. Program execution begins a
2  //
3
4  #include <stdio.h>
5  #include <windows.h>
6  #include <bcrypt.h> // Needed to solve issue "Function returning function is not allowed".
7  // Idk why this works but I used it in my Exodus work to define NTSTATUS a
8
9  typedef NTSTATUS(WINAPI* ZwAllocateVirtualMemory)(HANDLE ProcessHandle, PVOID* BaseAddress, UL
10 #define STATUS_SUCCESS 0
11
12 int main()
13 {
14     ZwAllocateVirtualMemory pZwAllocateVirtualMemory = (ZwAllocateVirtualMemory)GetProcAddress
15     DWORD baseAddress = 0x1;
16     SIZE_T sizeOfAllocation = 1024;
17     NTSTATUS result = pZwAllocateVirtualMemory(GetCurrentProcess(), (PVOID*)&baseAddress, 0, 8
18
19     if (result == STATUS_SUCCESS) {
20         printf("[*] Successfully allocated the NULL page!\r\n");
21     }
22     else {
23         printf("[!] Could not allocate the NULL page...\r\n");
24     }
25 }

```

100 % No issues found

Autos

Search (Ctrl+E) Search Depth: 3

Name	Value
&baseAddress	0x003ffebc {0x00000001}
&sizeOfAllocation	0x003ffeb0 {0x00000400}
pZwAllocateVirtualMemory	ntdll.dll!0x772afaf0 (load symbols for additional information)
result	0xc0000000
sizeOfAllocation	0x00000400

Figure 1

Note that the error code which was returned, which was saved in the `result` variable, was `0xC00000F0`, or `STATUS_INVALID_PARAMETER_2`. By referring to Microsoft's [NTSTATUS page](#), VS-Labs determined that this error code means that the second parameter passed to `ZwAllocateVirtualMemory()`, which was the base address of the memory to be allocated, was invalid.

Some readers may wonder what would happen if one attempted to allocate memory outside of the NULL page itself, but still within the first 64KB of memory. Such attempts will also fail, albeit with a somewhat different error code. In the example shown below, an attempt is made to allocate memory at the address `0xF80C` which resides within the first 64KB of memory.

This results in `ZwAllocateVirtualMemory()` returning the same error code, `0xC00000F0`, or `STATUS_INVALID_PARAMETER_2`, to indicate that the memory address specified is invalid, as can be seen in Figure 2.

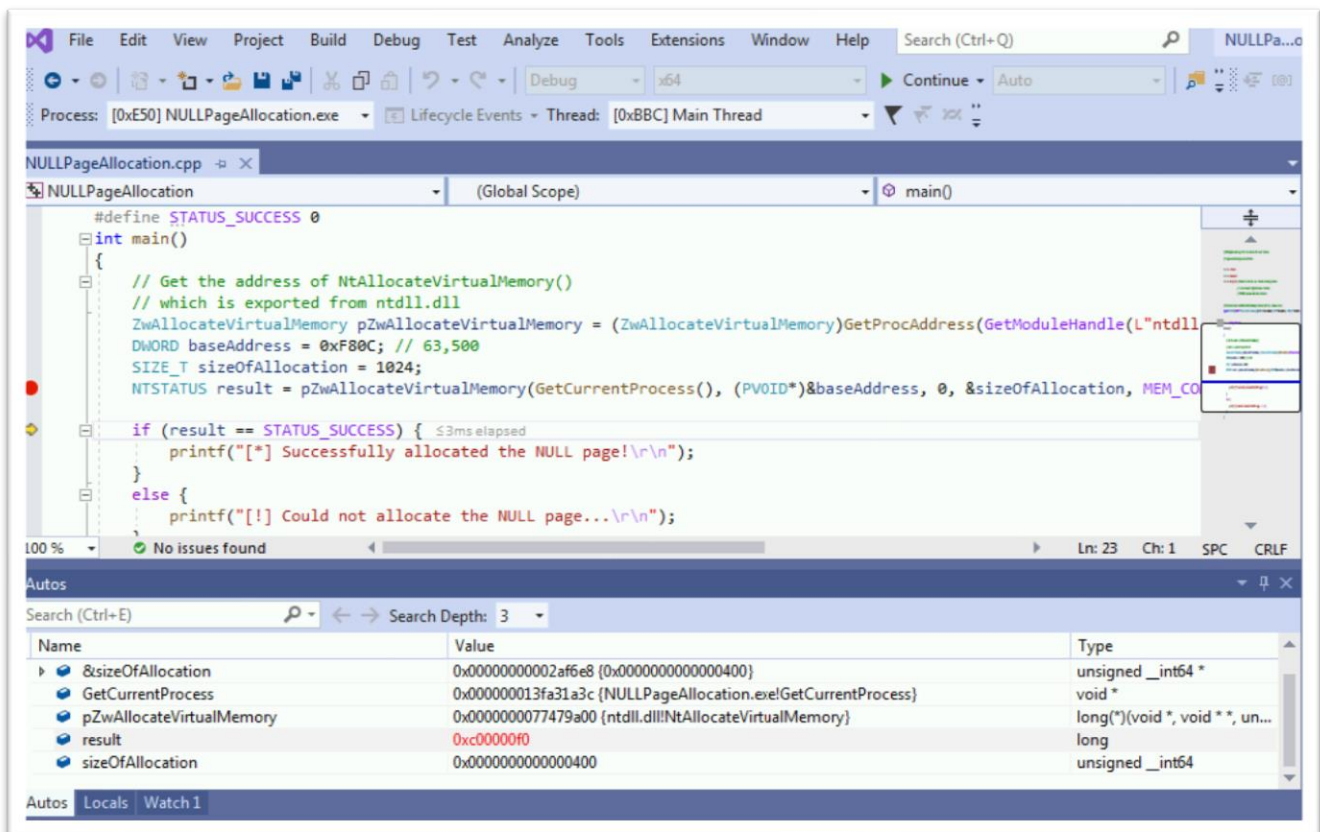


Figure 2

Finally, if a user attempts to allocate memory at a low-valued memory address that is not within the first 64KBs of memory, **ZwAllocateVirtualMemory()** will succeed, as can be seen in Figure 3.

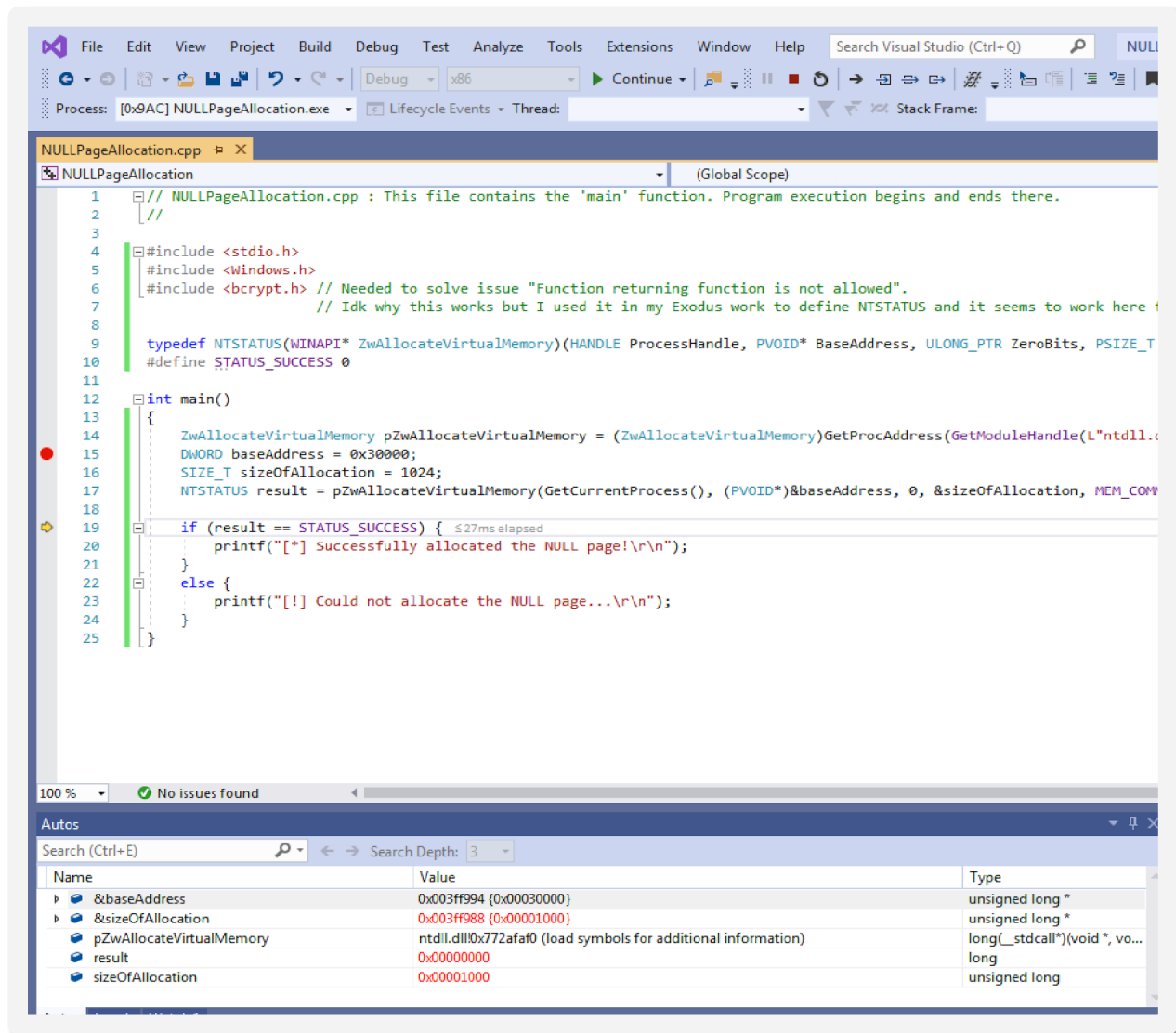


Figure 3

From the outcome of the last three tests, it was possible to confirm that **ZwAllocateVirtualMemory()** was changed internally, but only to ensure that it cannot allocate memory in the NULL page or within the first 64KB of memory.

Conducting the same tests on **NtVirtualAllocateMemory()** revealed similar results, indicating that it was also patched in an identical manner. This allowed VS-Labs to confirm that CVE-2019-1169 would only affect Windows 7 x86 and prior, and not Windows 8 and later or Windows 7 x64.

Target Setup

Environment Setup – Snapshots and Folders

Now that VS-Labs knew which systems were impacted by CVE-2019-1169, it was time to set up an environment to analyze the patches. For this task, VS-Labs used a VirtualBox VM running Windows 7 SP1 x86. Two snapshots were then taken on top of this clean image.

The first snapshot that was taken, named “*Windows 7 July 2019 Patches*”, was taken after all the patches prior to the CVE-2019-1169 patch were installed on a clean Windows 7 SP1 image. This was achieved by installing the July 2019 Monthly Rollup patch (known as [KB4507449](#)).

Monthly Rollup patches include all the patches for a given month and the months prior, which makes it very easy for researchers to set up a system with the correct set of patches applied.

The second snapshot that was taken was after [KB4512486](#) was installed, which contains the patch for CVE-2019-1169. This snapshot was named “*Windows 7 August 2019 Patches*”.

To confirm that this was the correct update package, VS-Labs looked at the [Microsoft CVE-2019-1169 advisory page](#) which specified that the Security Only update package for August 2019, the month CVE-2019-1169 was fixed, is [KB4512486](#). Figure 4 shows VirtualBox’s snapshot view after these actions were performed.

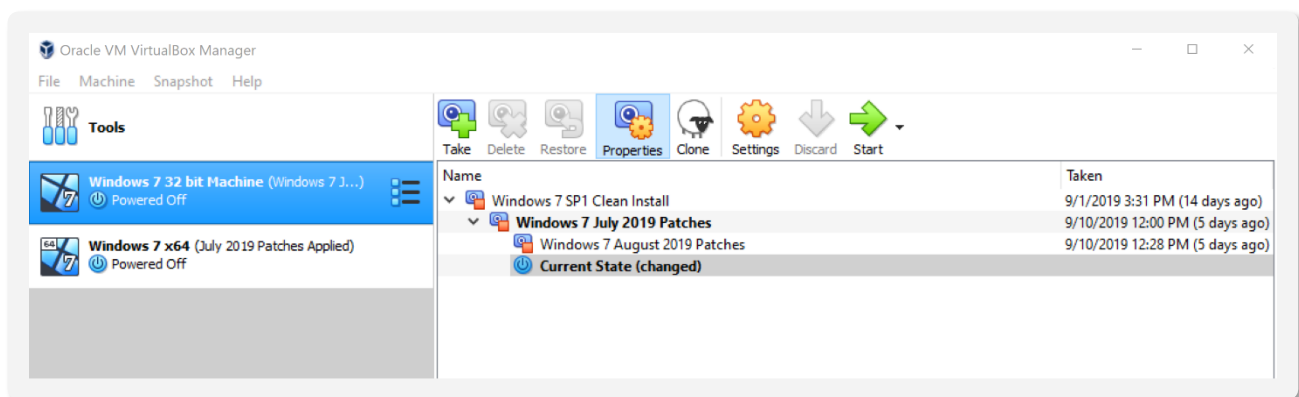


Figure 4

Once the patches were installed and the snapshots had been taken, VS-Labs extracted the **win32k.sys** files from both snapshots and copied them to the host machine.

The unpatched **win32k.sys** was placed into a folder named **old** and the patched **win32k.sys** was placed into a folder named **New**. This folder structure is shown in Figure 5:

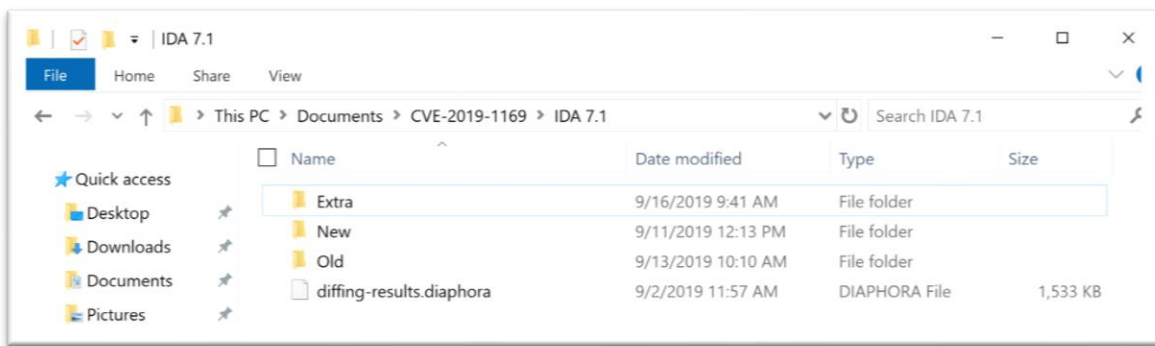


Figure 5

VS-Labs then loaded the two **win32k.sys** files into IDA Pro for analysis and saved the resulting IDA Pro database files (**.i64** files) in the corresponding local folders (**New** for the patched **win32k.sys**, **old** for the non-patched **win32k.sys**).

Symbol Path Setup

The next step VS-Labs needed to perform to investigate CVE-2019-1169 was to configure the symbol path on the host. The symbol path is an environment variable named **_NT_SYMBOL_PATH**, which usually contains two parts: the path to a local folder on your system that is used as a cache for downloaded PDB files, and a URL that points to a server to download PDB files that don't already exist in the cache. For most users, the path will point to **C:\Symbols** and the URL will point to Microsoft's symbol server.

VS-Labs needed to set this environment variable as without it Windows programs such as WinDBG that rely on this variable to locate the corresponding PDB files for various binaries won't be able to locate the appropriate PDB files. PDBs are Microsoft's version of the Unix symbol files, trying to debug programs without the information about the corresponding function names, arguments, return values, data types and structure layouts associated with a given binary would have been very

difficult. This is especially true when it comes to doing kernel research as often components are not documented and the only place to locate the corresponding information is within the PDB files.

To set up the symbol path, VS-Labs utilized the following command lines in an administrator-level PowerShell command prompt.

<i>PowerShell Symbol Path Setup Script</i>
--

<pre><i>mkdir c:\MySymbols\</i></pre>

<pre><i>[Environment]::SetEnvironmentVariable("_NT_SYMBOL_PATH", "cache*c:\MySymbols;srv*https://msdl.microsoft.com/download/symbols", "Machine")</i></pre>

The first command created the folder `C:\MySymbols\`. After this folder was created, the second command created a system environment variable named `_NT_SYMBOL_PATH` with the value `cache*c:\MySymbols;srv*https://msdl.microsoft.com/download/symbols` to specify that the system should store PDB files in `C:\MySymbols\` and that any PDB files which are not stored locally should be downloaded from the Microsoft Symbol Server before being saved into `C:\MySymbols\`.

Setting Up VirtualBox for Kernel Debugging

Once the symbol path was set up, the next step for VS-Labs was to configure VirtualBox for kernel debugging.

There are several options to do this; the most popular methods are COM and KDNET. [KDNET](#), which allows kernel debugging over a compatible network adapter, allows for much faster debugging than COM (refer to [MSDN](#) for more info) with less setup and lag, however it only works when the machine being debugged is running Windows 8 or later.

Unfortunately, since CVE-2019-1169 doesn't work on Windows 8, VS-Labs was forced to use normal COM communications over named pipes to kernel debug the Windows 7 machine chosen for testing.

To set up named pipes in VirtualBox, VS-Labs started by navigating to the VirtualBox Manager. After selecting the Windows 7 VM, VS-Labs selected the settings icon, which should show up as a yellow gear. The screen shown in Figure 6 was then displayed:

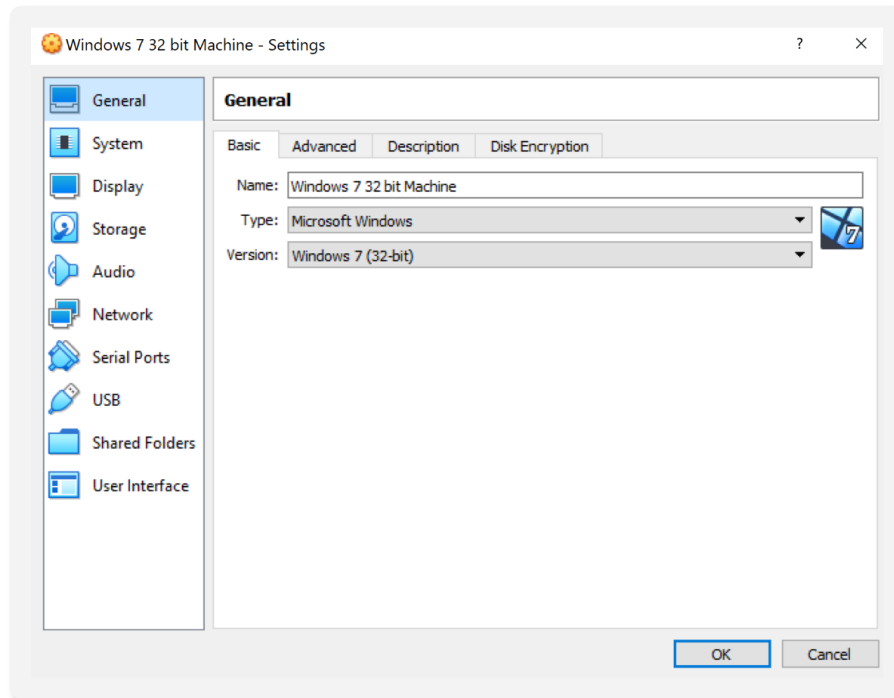


Figure 6

VS-Labs then selected the **Serial Ports** option and under the **Port 1** menu option, selected the **Enable Serial Port** checkbox along with the following options:

- Port: **COM1**
- Port Mode: **Host Pipe**
- Unchecked the **Connect to Existing Pipe/Socket** so that VirtualBox will create the pipe rather than connecting to an existing one.
- Path/Address: **\\.\pipe\Win7Kernel**

Figure 7 shows how these settings appeared once they had been entered correctly.

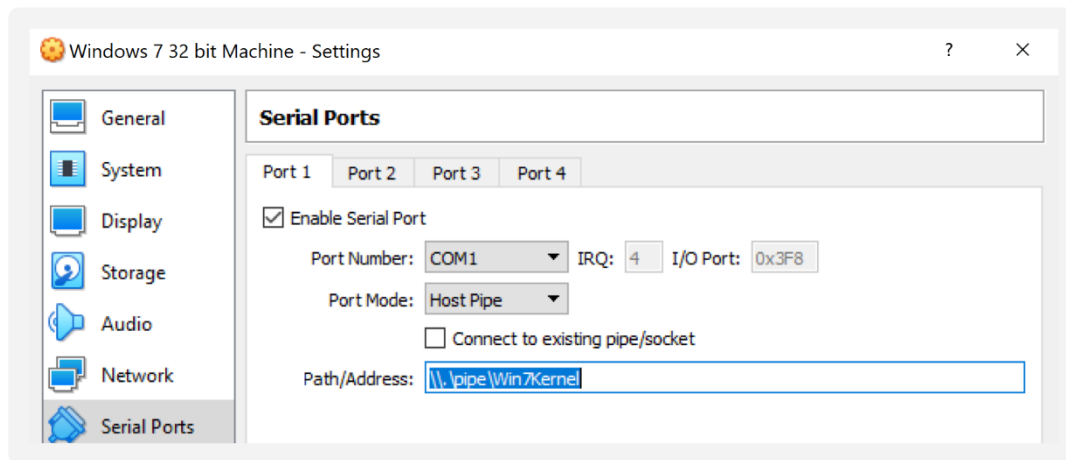


Figure 7

These settings will ask VirtualBox to create a COM pipe named `\\.\pipe\Win7Kernel` when the machine is started. This COM pipe will be associated with the serial port `COM1`.

Once this is done, VS-Labs booted up the snapshot to be debugged (which in this case was the machine with the July 2019 patches installed) and entered the following commands into an Administrator-level command prompt:

<i><code>bcdedit Commands to Enable Kernel Debugging on the Target</code></i>
<i><code>bcdedit /debug on</code></i>
<i><code>bcdedit /dbgsettings serial debugport:1 baudrate:115200</code></i>

These commands enabled debug mode on the target machine, which allows a remote machine to kernel debug the target machine, and sets the debug settings so that it will use serial port 1 (`COM1`), with a signal/baud rate of `115200`, a commonly used signal/baud rate.

Once this was complete, VS-Labs shut the guest machine down normally (not via VirtualBox) and then launched WinDBG Preview and selected **Attach to Kernel**. VS-Labs then checked the **Pipe**, **Reconnect**, and **Initial Break** buttons, set the **Baud Rate** to **115200**, and set the **Port** to **\\.\pipe\Win7Kernel**. Figure 8 shows what WinDBG Preview looked like once these settings had been entered correctly.

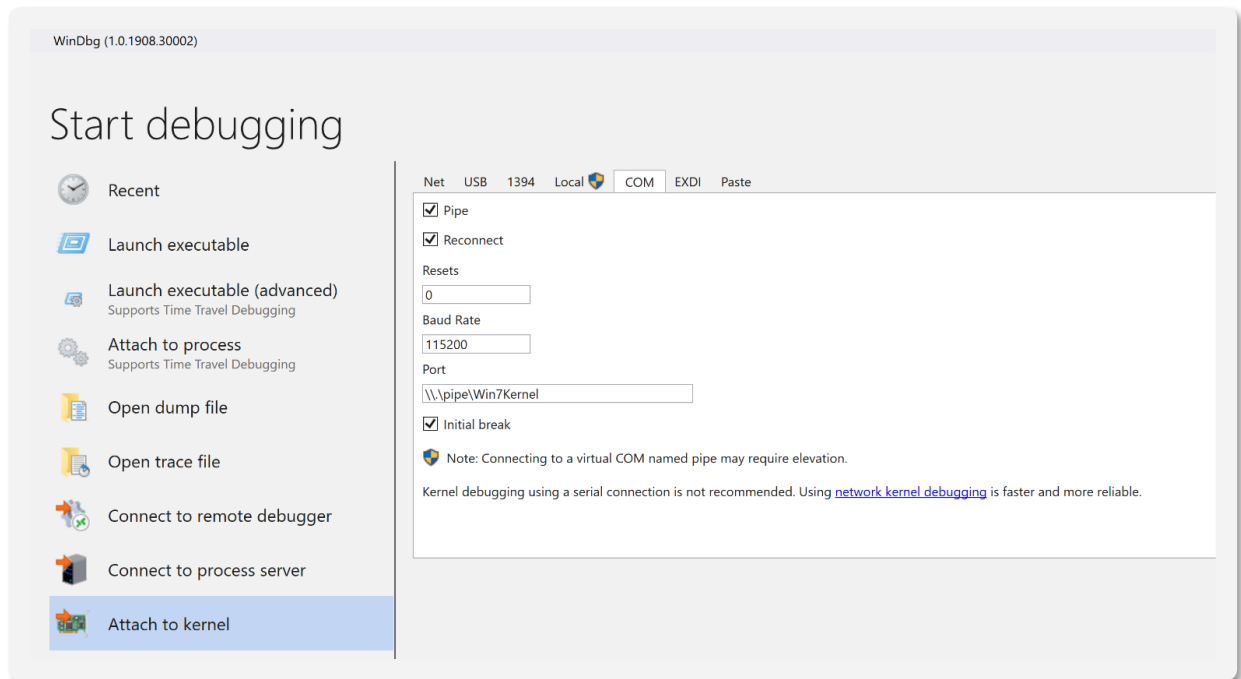


Figure 8

VS-Labs then pressed the **OK** button and WinDBG Preview displayed a window showing that it was waiting for the target machine to reconnect to the debugger. This can be seen in Figure 9.

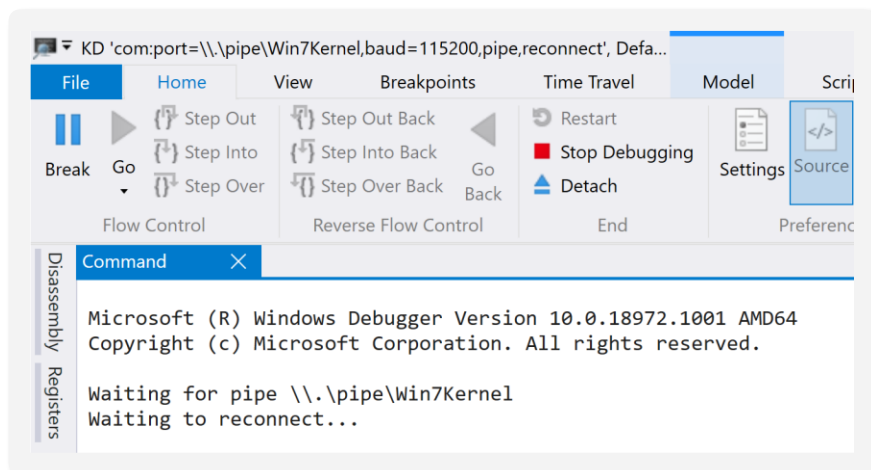


Figure 9

Once this message appeared, VS-Labs booted up the target machine in VirtualBox and was greeted with a confirmation from WinDBG Preview that the client had connected and that the symbols had been set up correctly. This can be seen in Figure 10.

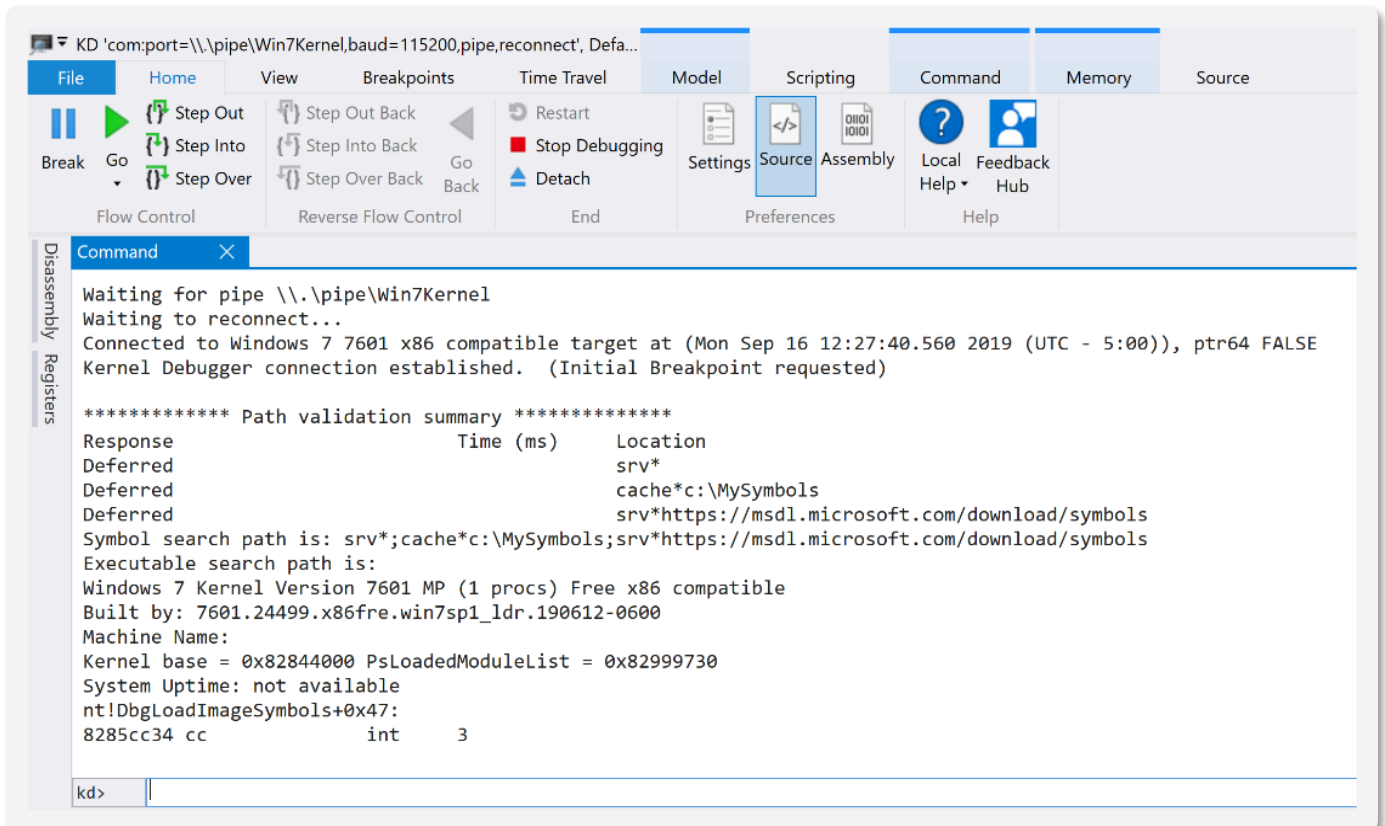


Figure 10

Patch Diffing and Initial Analysis

Diaphora Analysis

Once the environment was set up and the IDA Pro 64-bit (.i64) files were created, VS-Labs' next step was to perform patch diffing to see what had changed between the two versions of *win32k.sys*.

For this step, a diffing program was required. VS-Labs favorite tool for performing patch analysis is Diaphora. Diaphora is a well-known patch diffing tool which has a variety of heuristics and algorithms which make it much more accurate at identifying changes in Windows binaries than most of its competitors. It is available for free on GitHub at the [Diaphora repository](#) although users can also quickly acquire the current version of the repository as a ZIP file by using [this link](#) and unzipping its contents to a directory.

Once Diaphora had been cloned from GitHub, VS-Labs opened up the .i64 file corresponding to the patched *win32k.sys* file in IDA Pro and ran Diaphora on it by selecting **File→Run Script** and navigating to the location of the file diaphora.py, which was located in the root of the folder the Diaphora repository was cloned to. Once this was done, the dialog shown in Figure 11 appeared.

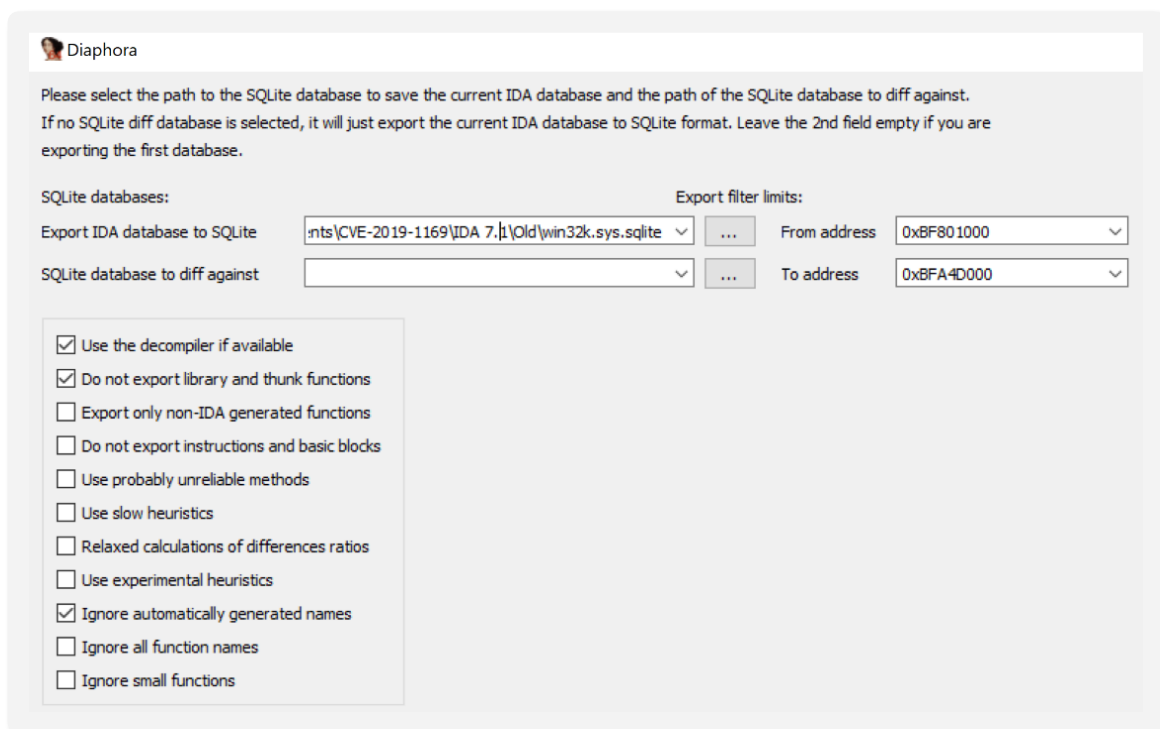


Figure 11

In most scenarios, analysts can safely accept the defaults that Diaphora provides. Note that the option *Ignore* automatically generated names will cause IDA to ignore any functions starting with *sub_*, which will be the case for functions without a supplied name.

This is acceptable for *win32k.sys* since symbols exist for Windows 7 SP1 x86, which will populate all the functions IDA Pro detects with function names. However, when symbols are not available or are incomplete, it is advisable to disable this option to ensure that all functions are appropriately analyzed.

For the sake of completeness however, VS-Labs decided to uncheck the Ignore automatically generated names option to ensure that all functions, regardless of their name, would be analyzed.

Once this was done, VS-Labs hit the *OK* button and Diaphora started to export function information corresponding to the currently loaded *.i64* file into an *.sqlite* file. This can be seen in Figure 12.

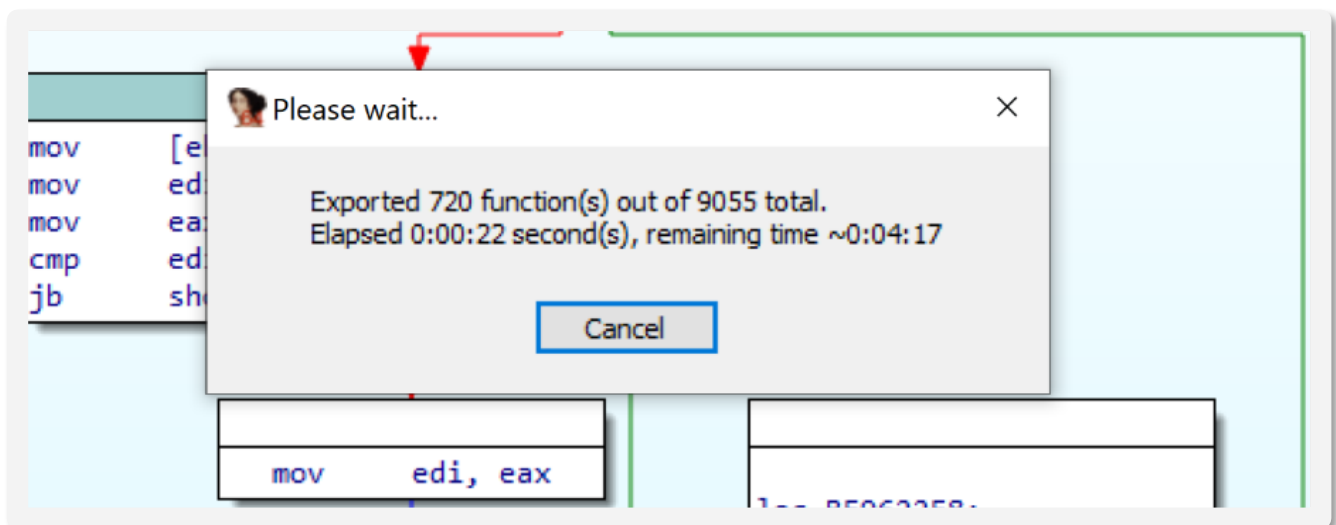


Figure 12

Once the analysis was complete, Diaphora closed this dialog to indicate that the export process was complete and output some information into the output log confirming that it had successfully exported the database information to an `.sqlite` database file.

Once this happens, VS-Labs closed IDA Pro, and reopened it with the `.i64` file corresponding to the patched version of `win32k.sys`. Diaphora was once again run against this file using the same settings, as can be seen in Figure 13.

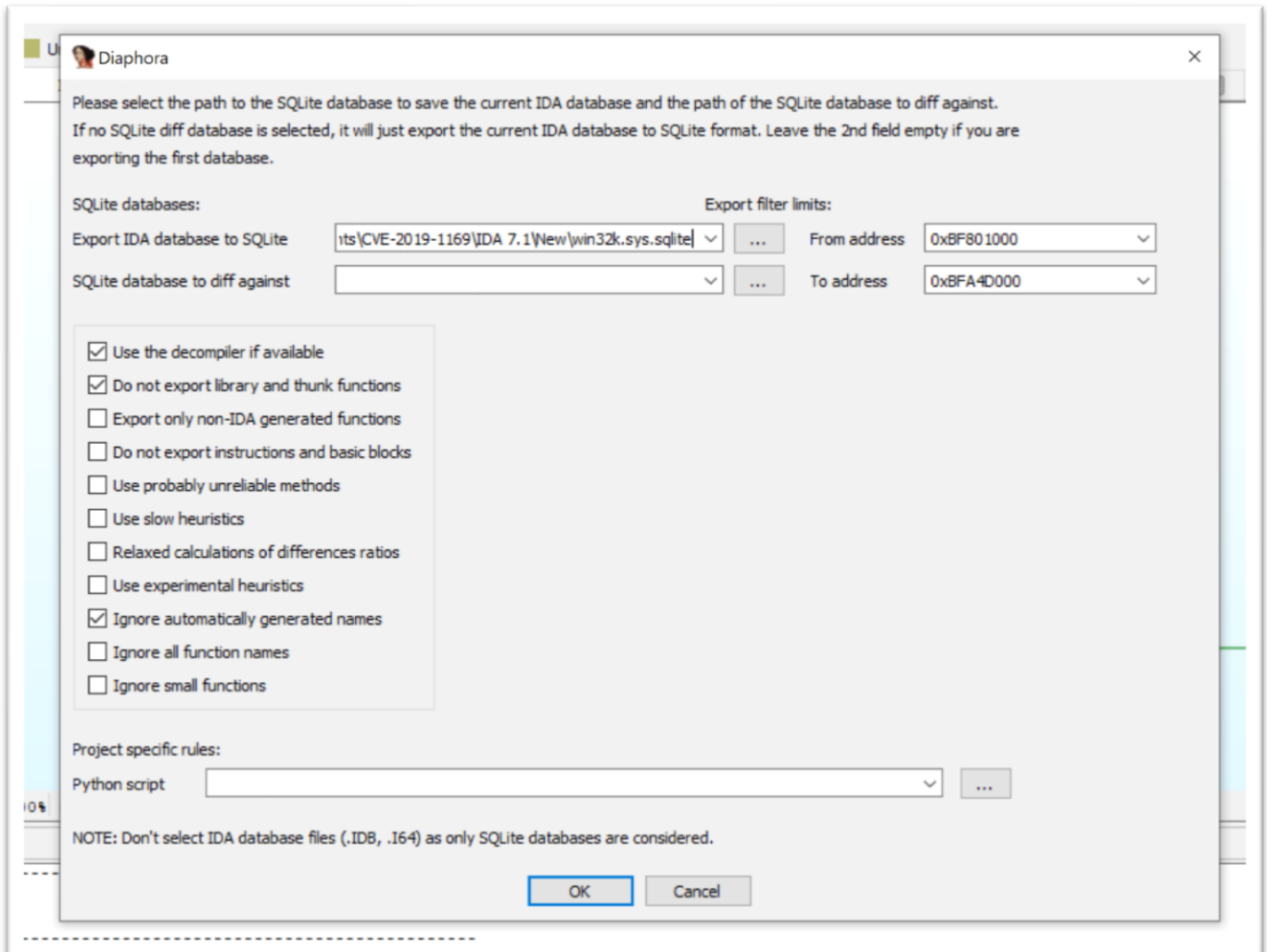


Figure 13

Once Diaphora had finished analyzing both files, VS-Labs was left with two `.sqlite` files: one for the non-patched `win32k.sys` file and one for the patched `win32k.sys` file. At this point, VS-Labs then closed IDA Pro down and then reopened IDA Pro and loaded the `.i64` file corresponding to the unpatched version of `win32k.sys`.

Diaphora was then run once more, however this time VS-Labs changed the `SQLite database to diff against` option so that it pointed to the path of the `.sqlite` file corresponding to the patched `win32k.sys` file as shown in Figure 14.

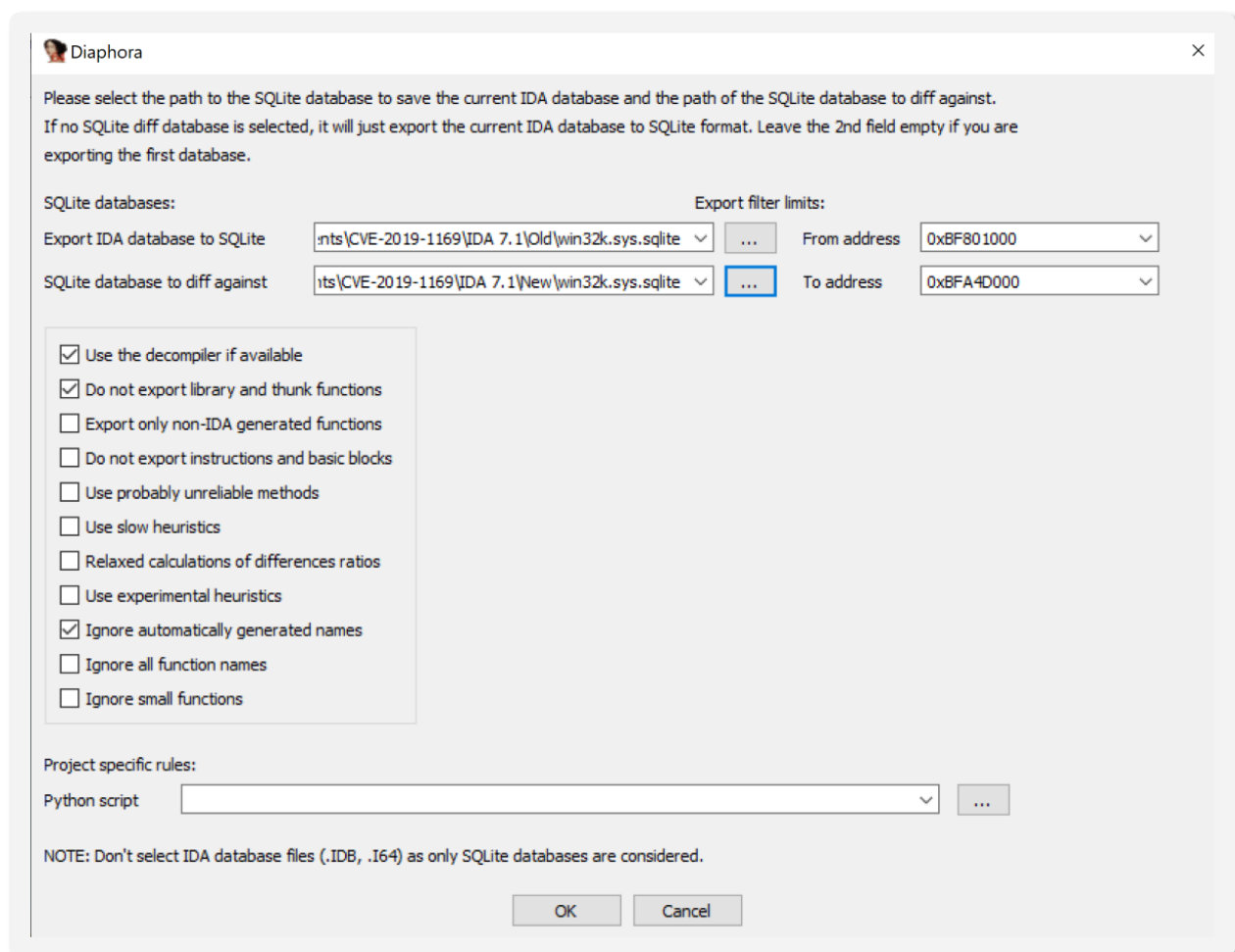


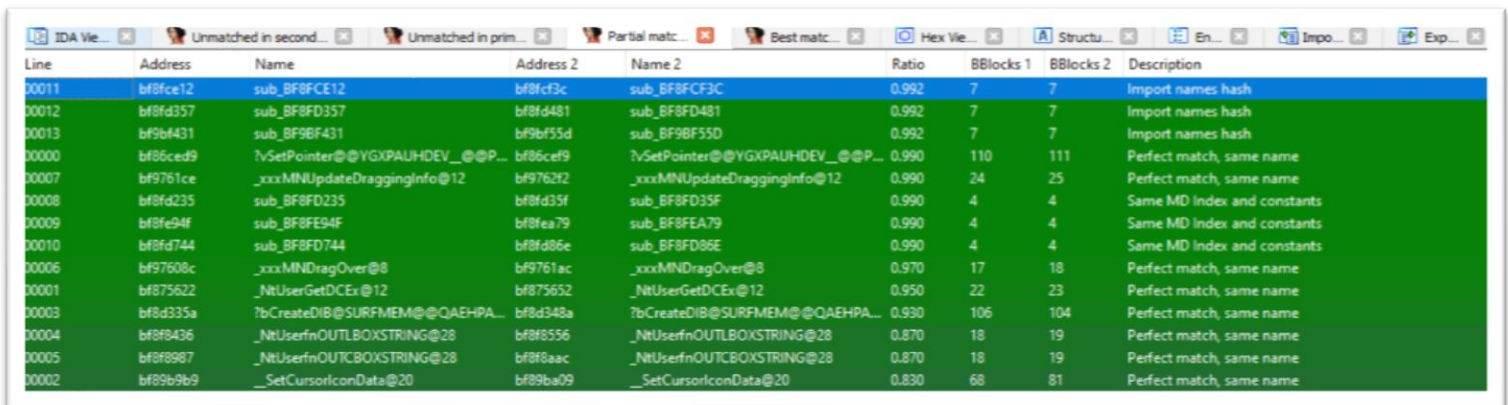
Figure 14

Once this was done, the **OK** button was pressed, and a popup menu was displayed asking if the existing `.sqlite` file should be overwritten. At this point VS-Labs clicked **No** on this prompt as Diaphora had already exported the required information for both versions of `win32k.sys` to `.sqlite` files; there was no reason to repeat this work again.

Diaphora then started running multiple threads to analyze the differences between the two files, applying various heuristics and algorithms along the way.

After roughly 20 minutes, the analysis completed, and a popup appeared asking if VS-Labs wanted to save the results straight away. Since VS-Labs did not know at this point what results were relevant, the **OK** button was pressed to ignore this popup and continue to the results.

From here the **Partial Matches** tab was selected. This resulted in the output shown in Figure 15.



Line	Address	Name	Address 2	Name 2	Ratio	BBlocks 1	BBlocks 2	Description
00011	bf8fce12	sub_BF8FCE12	bf8fc3c	sub_BF8FC3C	0.992	7	7	Import names hash
00012	bf8fd357	sub_BF8FD357	bf8fd481	sub_BF8FD481	0.992	7	7	Import names hash
00013	bf9bf431	sub_BF9BF431	bf9bf55d	sub_BF9BF55D	0.992	7	7	Import names hash
00000	bf86ced9	7vSetPointer@@YGXPAUHDEV_@@P...	bf86cef9	7vSetPointer@@YGXPAUHDEV_@@P...	0.990	110	111	Perfect match, same name
00007	bf9761ce	_xxxMNUpdateDraggingInfo@12	bf9762f2	_xxxMNUpdateDraggingInfo@12	0.990	24	25	Perfect match, same name
00008	bf8fd235	sub_BF8FD235	bf8fd35f	sub_BF8FD35F	0.990	4	4	Same MD index and constants
00009	bf8fe94f	sub_BF8FE94F	bf8fea79	sub_BF8FEA79	0.990	4	4	Same MD index and constants
00010	bf8fd744	sub_BF8FD744	bf8fd86e	sub_BF8FD86E	0.990	4	4	Same MD index and constants
00006	bf97608c	_xxxMNDragOver@8	bf9761ac	_xxxMNDragOver@8	0.970	17	18	Perfect match, same name
00001	bf875622	_NtUserGetDCEx@12	bf875652	_NtUserGetDCEx@12	0.950	22	23	Perfect match, same name
00003	bf8d335a	7bCreateDIB@SURFMEM@@QAEHPA...	bf8d348a	7bCreateDIB@SURFMEM@@QAEHPA...	0.930	106	104	Perfect match, same name
00004	bf8f8436	_NtUserfnOUTLBOXSTRING@28	bf8f8556	_NtUserfnOUTLBOXSTRING@28	0.870	18	19	Perfect match, same name
00005	bf8f8987	_NtUserfnOUTCBOXSTRING@28	bf8f8aac	_NtUserfnOUTCBOXSTRING@28	0.870	18	19	Perfect match, same name
00002	bf89b9b9	_SetCursorIconData@20	bf89ba09	_SetCursorIconData@20	0.830	68	81	Perfect match, same name

Figure 15

VS-Labs immediately realized that Diaphora may have detected some extra functions as having been changed, as several `sub_XXX` functions that appeared were marked as having been changed, yet the ratio of code changed was very small.

The reason this happens is because Diaphora can occasionally flag functions as being different even though they may have functionally similar code, such as when a function uses different registers to perform the same operations, or when code is altered so that it performs the same set of operations with more efficient instructions.

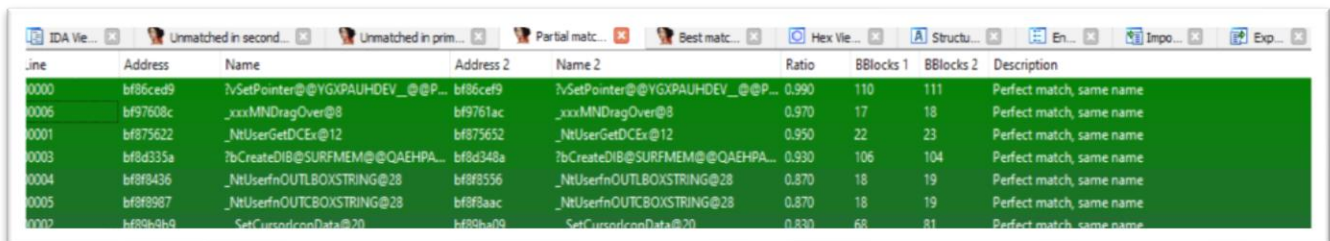
As Diaphora currently is not able to automatically detect these types of changes, reverse engineers must manually mark invalid functions as false positives.

After some quick analysis, VS-Labs determined that the **sub_XXX** functions in Diaphora's **Partial Matches** tab were false positives since each function was functionally the same, however the two **win32k.sys** files both referenced the same global variable using different methods.

VS-Labs then proceeded to remove these erroneous differences from the results, by selecting each **sub_XXX** function so that it was highlighted in blue, and then pressing **DEL** to remove it.

Note that due to a slight bug, the updates may not be shown immediately. If no updates occur after hitting Delete, it is recommended to right-click and select **Refresh** to update the results and see the changes.

Once all the **sub_XXX** functions were removed VS-Labs had a clean **Partial Matches** tab in Diaphora, as shown in Figure 16.



Line	Address	Name	Address 2	Name 2	Ratio	BBlocks 1	BBlocks 2	Description
0000	bf86ced9	!vSetPointer@@YGXPAUHDEV_@@P...	bf86cef9	!vSetPointer@@YGXPAUHDEV_@@P...	0.990	110	111	Perfect match, same name
0006	bf97608c	!xxxMNDragOver@8	bf9761ac	!xxxMNDragOver@8	0.970	17	18	Perfect match, same name
0001	bf875622	!NtUserGetDCEx@12	bf875652	!NtUserGetDCEx@12	0.950	22	23	Perfect match, same name
0003	bf8d335a	!bCreateDIB@SURFMEM@@QAEHPA...	bf8d348a	!bCreateDIB@SURFMEM@@QAEHPA...	0.930	106	104	Perfect match, same name
0004	bf8f8436	!NtUserfnOUTLBOXSTRING@28	bf8f8556	!NtUserfnOUTLBOXSTRING@28	0.870	18	19	Perfect match, same name
0005	bf8f8987	!NtUserfnOUTCBOXSTRING@28	bf8f8aac	!NtUserfnOUTCBOXSTRING@28	0.870	18	19	Perfect match, same name
0002	bf89b049	!SetCursorData@20	bf89b099	!SetCursorData@20	0.830	68	81	Perfect match, same name

Figure 16

xxxMNDragOver() Patch Analysis

Upon reviewing the ZDI advisory, VS-Labs noted that it mentioned the affected function was **xxxMNDragOver()**, which is listed in Figure 16 as one of the functions that Diaphora detected as being partially changed. By right clicking on the **xxxMNDragOver()**, entry within shown in Figure 16 and selecting **Diff assembly** in a graph, VS-Labs was able to obtain the graph shown in Figure 17.

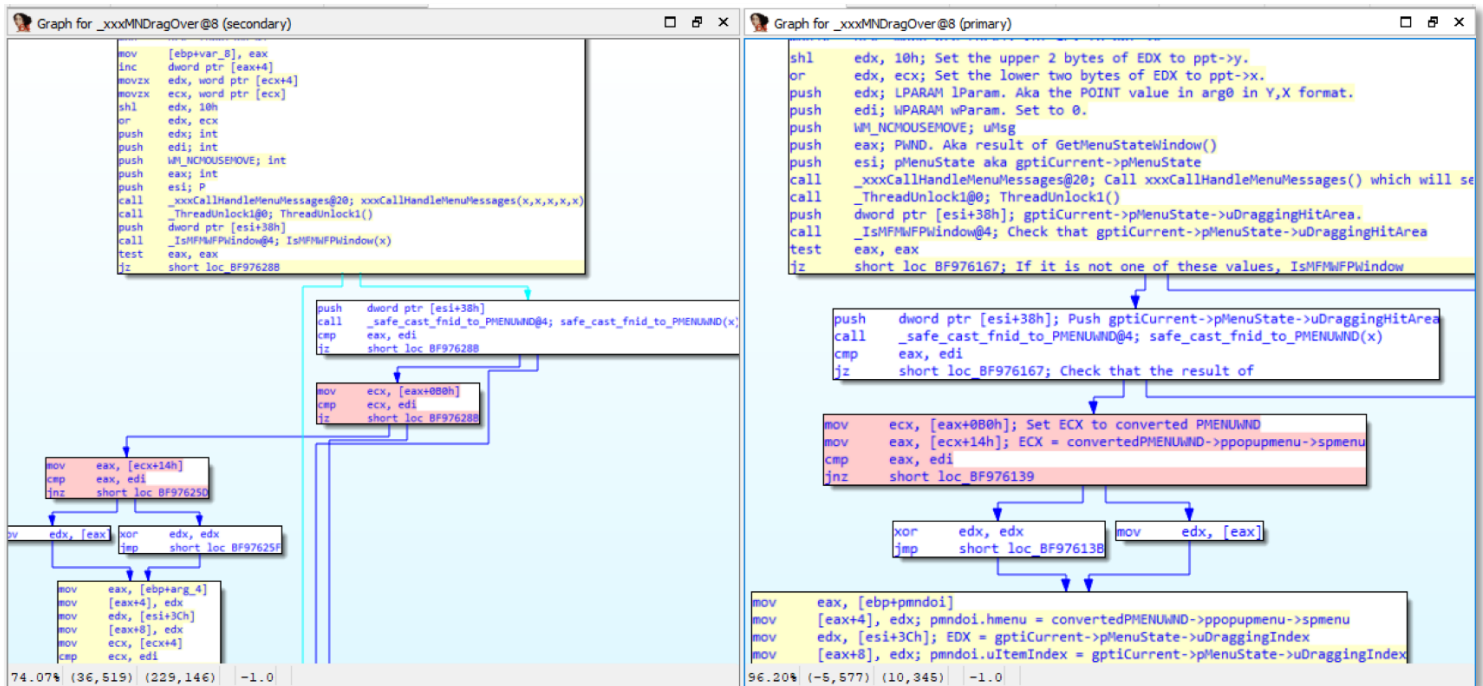


Figure 17

Note that the output shown in Figure 17 shows comments that were added by VS-Labs after several iterations of analysis; these were not added automatically by IDA Pro or Diaphora.

The right side of the image shows the unpatched code, whilst the left side of the image shows the updated code. By inspecting the code on the right side of the graph, VS-Labs noticed that there was a call to **_safe_cast_fnid_to_PMENUWND()**.

The result of this call was then compared to **EDI**, which was found to be set to **0x0** during testing.

This code had not been changed between the two versions, however there were several lines of code that operated on the result of **_safe_cast_fnid_to_PMENUWND()** which appeared to have been altered by the patch.

Since the function name is `_safe_cast_fnid_to_PMENUWND()` VS-Labs reasoned that the output of `_safe_cast_fnid_to_PMENUWND()` would be a `PMENUWND` structure. The definition for a `PMENUWND` structure is [defined on GitHub](#) and can be seen below.

<i>tagMENUWND Structure</i>
<pre>typedef struct tagMENUWND { WND wnd; PPOPUPMENU ppopupmenu; } MENUWND, *PMENUWND;</pre>

With some idea about what data the function might be operating on, VS-Labs reexamined the old version of the code once more, which is in red in the graph on the right of Figure 17.

This code first sets `ECX` to the value of the `ppopupmenu` pointer within the `PMENUWND` object returned by `_safe_cast_fnid_to_PMENUWND()`.

Once this is done, the following instruction will set `EAX` to the value `spmenu` field within the `POPUPMENU` structure that `ppopupmenu` points to. `EAX` is then compared to `EDI`, which will hold a value of `0x0`, thereby ensuring that the `spmenu` field is not NULL. The location of the `spmenu` field within `PPOPUPMENU` was confirmed by VS-Labs by looking at the `tagPOPUPMENU` structure in WinDBG:

<i>tagMENU Location Within tagPOPUPMENU</i>
<pre>1: kd> dt win32k!tagPOPUPMENU ... +0x014 spmenu : Ptr32 tagMENU ...</pre>

Observant readers may have noticed that there is an issue with this code however, as no checks are made to ensure that the `PMENUWND` object returned by `_safe_cast_fnid_to_PMENUWND()` contains a `ppopupmenu` field which is not NULL.

Therefore, it is possible that `ppopupmenu` could be a NULL pointer, which would result in `EAX` being set to the value of the 32 bits at location `0x14` in memory. As a result, the attacker could control the value of `EAX`, which would allow them to potentially alter `xxxMNDragOver()`'s execution.

All the attacker would need to do is be able to allocate the NULL page in memory, which users can do on Windows 7 x86 by calling function such as `ZwAllocateVirtualMemory()` or `NtAllocateVirtualMemory()`, and then set offset `0x14` of this page to a value of their choice. This would then allow them to control `EAX` when `EAX` is set to the value of the `spmenu` field of `ppopupmenu`.

The patched code, shown in the graph on the left half of Figure 17, fixed this vulnerability by ensuring that the `PMENUWND` object returned by `_safe_cast_fnid_to_PMENUWND()` contains a `ppopupmenu` field that is not NULL. If the `ppopupmenu` field is NULL, `xxxMNDragOver()` will stop processing the object and will prematurely terminate, preventing the attacker from being able to control the program's behavior.

Conclusion

By now, it should be clear that an attacker can exploit CVE-2019-1169 to gain at least some level of control over how `xxxMNDragOver()` executes.

However, some questions remain, such as “*What exactly can the attacker control?*” and “*Does this lead to a useful information leak?*”

In part two we will investigate these questions and examine exactly how [VerSprite's Research practice](#), VS-Labs, was able to determine the usefulness of CVE-2019-1169 and verify that it can be exploited by an attacker to leak kernel address. [Subscribe to our resource library to stay tuned.](#)