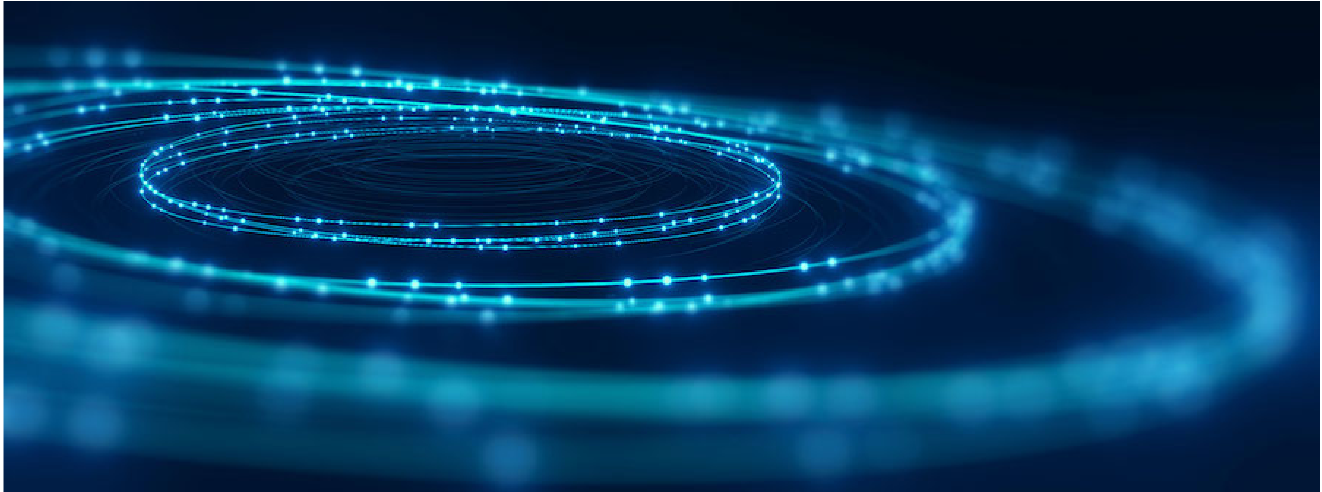
Technical Analysis of CVE-2021-1732

January 5, 2022



By Eoin Carroll on Jan 05, 2022

Introduction

In February 2021, the company Dbappsecurity discovered a sample in the wild that exploited a zero-day vulnerability on Windows 10 x64.

The vulnerability, CVE-2021-1732, is a win32k window object type confusion leading to an OOB (out-of-bounds) write which can be used to create arbitrary memory read and write capabilities within the Windows kernel (local Elevation of Privilege (EoP)). Memory exploitation generally requires a read, write, and execute primitive to bypass modern exploit mitigations such as DEP, ASLR and CFG on hardened operating systems such as Windows

10. A data-only attack requires only a read and write primitive as it does not seek to execute malicious code in memory, but rather manipulates data structures used by the operating system to its advantage (i.e., to achieve elevated privileges).

Kernel exploits are usually the most sophisticated attack as they interact directly with the Windows kernel. When such attacks are successful, they are critical because they provide high privileges to the attacker, which can be used to increase the impact of the overall exploit chain. In this case the exploit is a Local Privilege Escalation (LPE) that targets 64-bit Windows 10 version 1909. The original sample discovered was compiled in May 2020 and reported to Microsoft in December 2020. While searching for additional findings we went through a public exploit published in March of 2021 by a researcher. Having this code publicly available may raise the potential for additional threat attackers. While we have not found clear evidence demonstrating malicious use of the proof-of-concept (POC), we did discover some variants being tested and uploaded to VirusTotal.

In this blog post, McAfee Advanced Threat Research (ATR) performed a deep dive into the analysis of the vulnerability, to identify the primitives for detection and protection. The exploit is novel in its use of a new win32k arbitrary kernel memory read primitive using the GetMenuBarInfo API, which to the best of our knowledge had not been previously known publicly.

CVE-2021-1732 Deep Dive

Exploitation of CVE-2021-1732 can be divided into six stages with the end goal of escalating a process' privileges to System. The following diagram shows the stages.

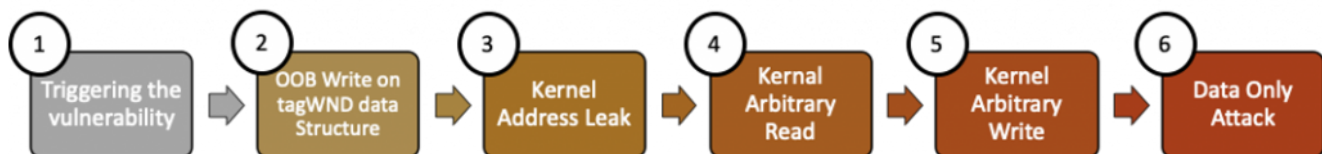


Figure 1 – Six stages of CVE-2021-1732

Before we dive into the details, we must give some background to win32k exploitation primitives which are used in the exploitation of CVE-2021-1732.

Win32K Background

Win32k is a Graphical (GUI) component of the Microsoft Windows Subsystem, most of which exists in the kernel for performance reasons. It is used for graphical print of the Windows OS desktop. However, due to the win32k architecture, the kernel component of win32k still needs to be able to make calls to user mode through user-mode callback functions to facilitate window creation and management.

Kernel user-mode callbacks have been well researched as far back as 2008 and 2010, with a very comprehensive analysis in 2011 by Mandt. A win32k kernel function such as xxxCreateWindowEx will make a callback function such as xxxClientAllocWindowClassExtraBytes through the user process PEB KernelCallbackTable.

When the user-mode callback has completed, NtCallbackReturn executes and passes the expected return parameter back to the kernel. Due to the stateless nature of these callbacks, many vulnerabilities have been discovered related to the locking mechanisms on the objects leading to use-after-free (UAF) exploitation.

Win32k has been one of the most exploited components in the Windows kernel accounting for 63% of vulnerabilities from 2010 to 2018, due to its large attack surface of syscalls relative to ntdll syscalls. Win32k vulnerabilities are generally turned into data-only attacks using a read/write kernel primitive by using a desktop object known as a tagWND data structure.

There are two aspects to data-only attacks:

1. Discovering a vulnerability.
2. Leveraging existing or new read/write primitives using specific OS APIs on object fields such as tagWND.cbWndExtra.

The tagWND data structure has two fields which make it a prime target for reading/writing within kernel memory; tagWND.cbWndExtra and tagWND.ExtraBytes. When a window is created using CreateWindowEx, it is possible to request additional bytes of memory directly after the tagWND object in memory through the cbWndExtra field in the WNDCLASSEXA structure when registering the window class.

The number of extra bytes is controlled by the cbWndExtra field, and the allocated additional memory address is located at the ExtraBytes field. The read/write primitive is created as follows:

1. Discover a vulnerability such as a UAF, which will allow you to write to a tagWND object in memory called WND0.
2. Allocate another tagWND object called WND1 near the previously corrupted WND0 in memory.
3. Overwrite WND0.cbWndExtra to a large value such as 0xFFFFFFF.
4. Call an API such as SetWindowLongPtr on WND0 which will write OOB to fields within WND1.

Win32k kernel user-mode callbacks have been exploited many times by leveraging tagWND read/write capabilities within the Windows kernel for escalation of privileges such as CVE-2014-4113, CVE-2015-0057, MS15-061, CVE-2016-7255 and CVE-2019-0808.

Win32k Exploit Primitives

Several primitives have been observed in the CVE-2021-1732 exploit used by the attackers; additionally, it is worth mentioning that some of them are new and not previously seen in the wild.

Prior to Windows RS4 it was trivial to leak tagWND kernel addresses using multiple techniques, such as calling HMValidateHandle to copy tagWND objects from the kernel to user desktop heap. The latest version of Windows 10 has been hardened against such trivial techniques.

However, using the spmenu kernel address leak technique and relative tagWND desktop heap offsets, once a vulnerability is discovered to overwrite a tagWND.cbWndExtra field, it is possible to achieve kernel read/write capabilities without leaking the actual tagWND kernel addresses. The spmenu technique in this exploit was used here and here, but we are not aware of the GetMenuBarInfo API ever being used before in a win32k exploit.

The following diagram shows the primitives used in CVE-2021-1732.

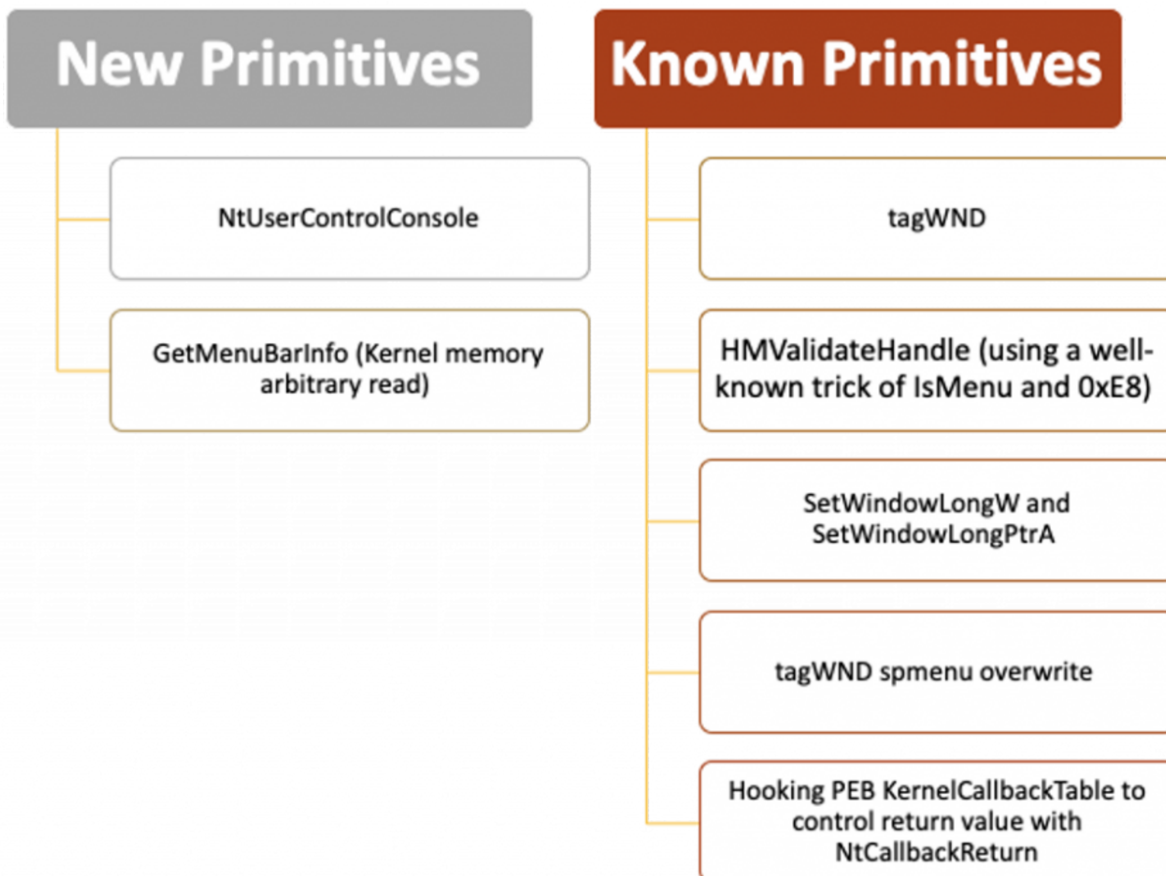


Figure 2 – CVE-2021-1732 Primitives

Existing Windows OS Mitigations

Great work has been done to harden the security of win32k against EoP attacks with new and improved mitigations by the Microsoft OSR team, Mandt, Google Project Zero, Schenk and Dabah. These mitigations include:

1. Type isolation (all same type objects tagWND being used).
2. Win32k filtering (limited to Edge browser and not process wide but since this research there have been many improvements on win32k API filtering capabilities such as the addition of _stub_UserSetWindowLong and _stub_UserSetWindowLongPtr _stub_UserGetMenuBarInfo in win32k.sys).
3. Fragmenting kernel desktop heap and removal of kernel addresses in the user desktop heap (can use relative offsets within user and desktop heaps described later in the blog).
4. Removal of data type symbols from win32k drivers (obfuscation rather than mitigation).

In the context of a malicious process exploiting CVE-2021-1732, the above mitigations provide no protection. However, it does not impact Google Chrome as it disallows win32k calls (Windows 8 and higher), or Microsoft Edge as it applies win32k filtering on the relevant APIs.

Triggering the Vulnerability and Patch Analysis

When a window is created using CreateWindowEx API, a tagWND object is created by the Windows operating system. This window, as explained above, can be created with a parameter to allocate extra memory using cbWndExtra.

During the windows creation process (CreateWindowEx API) a callback named xxxClientAllocWindowClassExtraBytes is triggered to allocate space in the user mode desktop heap for the tagWND.ExtraBytes (offset 0x128) per the tagWND.cbWndExtra (offset 0xc8) value size (see figure 3 and 4 below for WND1).

```
0: kd> dps ffffe709431e7540
ffffe709`431e7540  00000000`000202f6
ffffe709`431e7548  00000000`00000004
ffffe709`431e7550  ffffe709`44ece8a0
ffffe709`431e7558  ffffc28f`311164b0
ffffe709`431e7560  ffffe709`431e7540
ffffe709`431e7568  ffffe709`410338c0
ffffe709`431e7570  00000000`000338c0
ffffe709`431e7578  00000000`00000000
ffffe709`431e7580  00000000`00000000
ffffe709`431e7588  00000000`00000000
ffffe709`431e7590  00000000`00000000
ffffe709`431e7598  ffffe709`431e5150
ffffe709`431e75a0  ffffe709`431e8000
ffffe709`431e75a8  ffffe709`40830930
ffffe709`431e75b0  00000000`00000000
ffffe709`431e75b8  00000000`00000000
```

```
0: kd> dps ffffe709`410338C0
ffffe709`410338c0  00000000`000202f6
ffffe709`410338c8  00000000`000338c0
ffffe709`410338d0  80000700`40020019
ffffe709`410338d8  0cc00000`08000100
ffffe709`410338e0  00007ff6`851b0000
ffffe709`410338e8  00000000`00000000
ffffe709`410338f0  00000000`000010d0
ffffe709`410338f8  00000000`00000000
ffffe709`41033900  00000000`00000000
ffffe709`41033908  00000000`0002e4e0
ffffe709`41033910  00000000`00033a90
ffffe709`41033918  00000000`00000000
ffffe709`41033920  00000027`00000088
ffffe709`41033928  0000001f`00000008
ffffe709`41033930  0000001f`00000080
ffffe709`41033938  00007ff6`852188b1
```

```
0: kd> dps ffffe709`410338C0+128
ffffe709`410339e8  000002b0`bd4b9fe0
```

Figure 3 – WND1 Kernel tagWND – User mode copy located at offset 0x28

```
0: kd> dps 0x000002B0BF3538C0 L3
000002b0`bf3538c0  00000000`000202f6
000002b0`bf3538c8  00000000`000338c0
000002b0`bf3538d0  80000700`40020019
```

```
0: kd> dps 0x000002B0BF3538C0+128
000002b0`bf3539e8  000002b0`bd4b9fe0
```

```
0: kd> dps 000002b0`bd4b9fe0 L4
000002b0`bd4b9fe0  00000000`00000000
000002b0`bd4b9fe8  00000000`00000000
000002b0`bd4b9ff0  00000000`00000000
000002b0`bd4b9ff8  00000000`00000000
```

Figure 4 – WND1 User Mode tagWND

The location of this memory is stored as a user mode memory pointer to the desktop heap and placed at tagWND.ExtraBytes. It is then possible to convert the normal window to a console window using NtUserConsoleControl which will convert that user mode pointer at tagWND.ExtraBytes to an offset value which points into the kernel desktop heap (see figure 5 below for WND0). It is this change in value at tagWND.ExtraBytes (window type confusion) that can be exploited for an OOB write during the xxxClientAllocWindowClassExtraBytes callback window.

```
0: kd> dps 0x000002B0BF34E4E0 L3
000002b0`bf34e4e0  00000000`000402be
000002b0`bf34e4e8  00000000`0002e4e0
000002b0`bf34e4f0  80000700`40020019
```

```
0: kd> dps 0x000002B0BF34E4E0+128
000002b0`bf34e608  00000000`0002e4e0
```

Figure 5 – WND0 User Mode tagWND

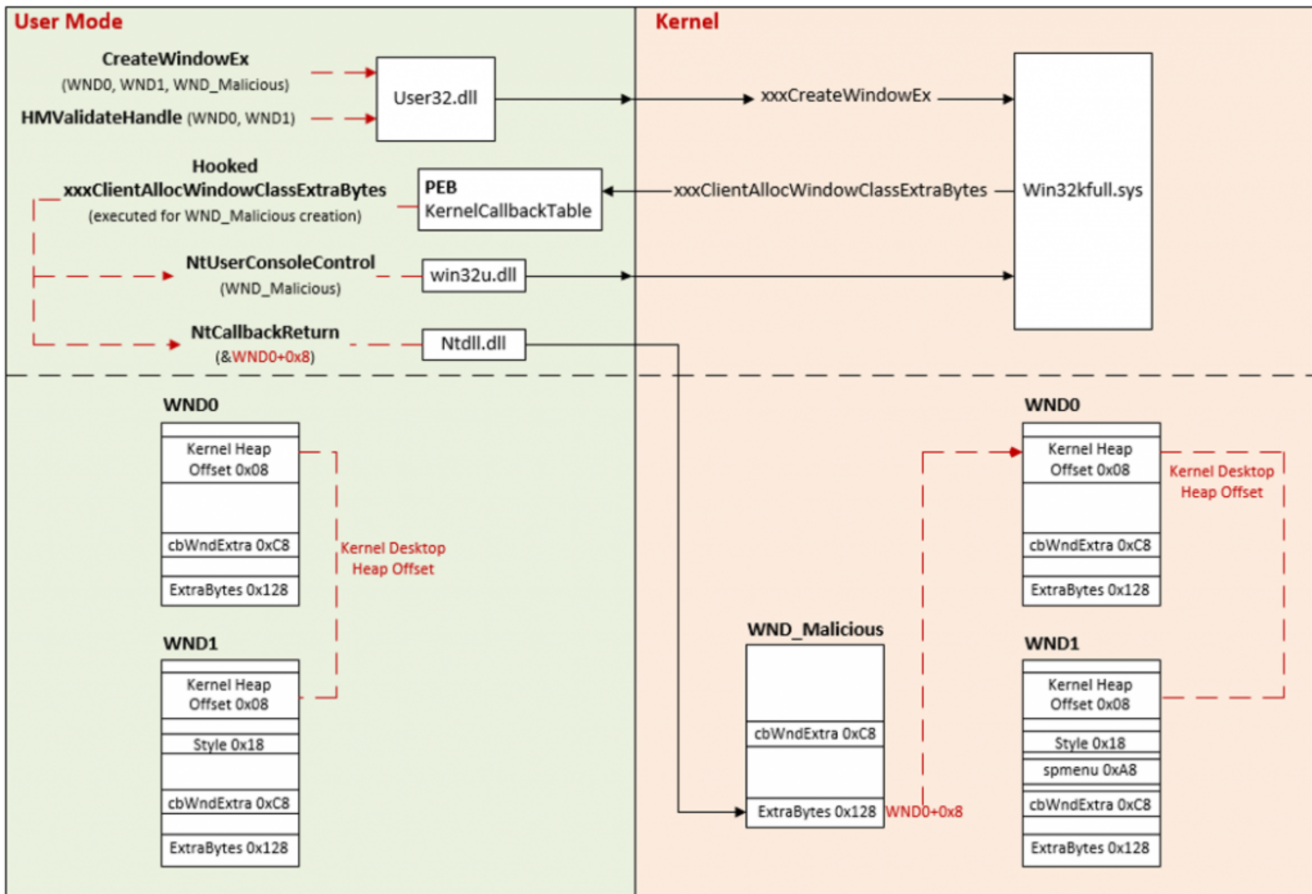


Figure 6 – Triggering the type confusion vulnerability within win32kfull!xxxCreateWindowEx

Per figure 6 above the following steps are required to trigger the vulnerability:

1. Get a pointer to the HMValidateHandle inline function within user32.dll.
2. Hook xxxClientAllocWindowClassExtraBytes within the PEB KernelCallBack table.

3. Create multiple windows (we will just use the first two WND0 and WND1 created), using the CreateWindowEx API, so that two windows are created in close memory proximity.
4. Call HMValidateHandle on WND0 and WND1 which will copy their objects from the kernel desktop heap to user desktop heap. At tagWND+0x8 an offset is stored into the desktop heap; this offset is the same for the user and kernel desktop heaps. The exploit uses these offset values to calculate the relative distance between WND0 and WND1 in the kernel desktop heap which is needed later for reading and writing OOB. Per table 1 below, by using these offsets there is no requirement to leak the actual WND0 and WND1 kernel addresses since read and writes can be done relative to the offsets (user and kernel desktop heaps have the same offsets).

Handle/Address	WND0	WND1
User Handle	402be	202f6
Desktop Heap Offset	2e4e0	338c0
User Desktop Heap	0x000002b0bf34e4e0	0x000002b0bf3538c0
Kernel Desktop Heap	0xffffe7094102e4e0	0xffffe709410338c0

Table 1 – User and Kernel Desktop heaps have the same offsets

5. WND0 is then converted to a console window by calling NtUserConsoleControl which converts WND0.ExtraBytes from a user desktop heap pointer to an offset within the kernel desktop heap. This is needed later so that WND0 can write OOB to WND1.

6. Create malicious window WND_Malicious using the CreateWindowEx API

- During the window creation the callback xxxClientAllocWindowClassExtraBytes API is executed to request user mode to allocate memory for WND_Malicious.cbWndExtra and pass the user desktop heap pointer back to the kernel function win32kfull!xxxCreateWindowEx.
- xxxClientAllocWindowClassExtraBytes has now been hooked and we do the following before returning to win32kfull!xxxCreateWindowEx:
  - Call NtUserConsoleControl to convert WND_Malicious to a console window so converting its WND_Malicious.cbWndExtra from a user desktop heap pointer to an offset within the kernel desktop heap.
  - Finally call NtCallbackReturn which completes the callback and returns a single value to xxxClientAllocWindowClassExtraBytes. Instead of passing the user desktop heap pointer as expected by xxxClientAllocWindowClassExtraBytes back to the kernel we pass the value at WND0+0x08 which is the kernel desktop heap offset to WND0 per figure 7 below. Now anytime we call SetWindowLongW on WND_Malicious we will be writing to WND0.

```
1: kd> dps fffffb45`410217f0+128
fffffb45`41021918  00000000`0002e4e0
```

Figure 7 – WND_Malicious

Patch Analysis

The vulnerability lies in the fact that win32kfull!xxxCreateWindowEx does not check whether the window type has changed between the time it initiates the xxxClientAllocWindowClassExtraBytes and gets the response from NtCallbackReturn.

When we call NtUserConsoleControl with WND_Malicious in the hook above, xxxConsoleControl checks if tagWND+0xE8 flag has been set to 0x800 to indicate a console window per figure below. As WND_Malicious was created as a normal window, xxxConsoleControl allocates memory at an offset within the kernel desktop heap and then frees the user desktop heap pointer existing at WND_Malicious.ExtraBytes (0ffset 0x128). It then places the offset to this new allocation in the kernel heap at WND_Malicious.ExtraBytes (0ffset 0x128) and sets the tagWND+0xE8 flag to 0x800 to indicate it's a console window.

```
  if ( (*(*tagWND + 0xE8i64) & 0x800) != 0 )
  {
    Offset_Allocation = (*(v19 + 0x128) + *(*(HWND_Validated + 0x18) + 0x80i64));
  }
  else
  {
    Offset_Allocation = DesktopAlloc(*(HWND_Validated + 0x18), *(v19 + 0xC8));
    if ( !Offset_Allocation )
    {
      v5 = 0xC0000017;
LABEL_33:
      ThreadUnlock1();
      return v5;
    }
    if ( *(*tagWND + 0x128i64) )
    {
      v23 = PsGetCurrentProcess(v21);
      v29 = *(*tagWND + 0xC8i64);
      v28 = *(*tagWND + 0x128i64);
      memmove(Offset_Allocation, v28, v29);
      if ( (*(v23 + 0x30C) & 0x40000008) == 0 )
        xxxClientFreeWindowClassExtraBytes(HWND_Validated, *(*(HWND_Validated + 0x28) + 0x128i64));
    }
    *(*tagWND + 0x128i64) = Offset_Allocation - *(*(HWND_Validated + 0x18) + 0x80i64);
  }
  if ( Offset_Allocation )
  {
    *Offset_Allocation = *(ConsoleCtrlInfo + 2);
    Offset_Allocation[1] = *(ConsoleCtrlInfo + 3);
  }
  *(*tagWND + 0xE8i64) |= 0x800u;
  goto LABEL_33;
```

After returning from the callback when we issued NtCallbackReturn above, xxxCreateWindowEx does not check that the window type has changed and places the WND0+0x08 at WND_Malicious.ExtraBytes per figure 9 below. The RedirectFieldpExtraBytes checks the WND_Malicious.ExtraBytes initialized value but it is too late as WND0+0x08 has already been written to WND_Malicious.ExtraBytes (offset 0x128).

```
*(*(tagWND + 5) + 0x128i64) = xxxClientAllocWindowClassExtraBytes(*(*(tagWND + 5) + 0xC8i64));
v244 = 0i64;
if ( !tagWND::RedirectedFieldpExtraBytes::operator==<unsigned __int64>(tagWND + 0x140, &v244) )
{
  if ( IsWindowBeingDestroyed(tagWND) || (*(_HMPheFromObject(tagWND) + 25) & 1) != 0 )
  {
    *(*(tagWND + 5) + 296i64) = 0i64;
    goto LABEL_478;
  }
  goto LABEL_204;
}
v197 = 2;
LABEL_478:
```

Figure 9 – win32kfull!xxxCreateWindowEx (vulnerable version)

The patched win32kfull.sys has updated xxxCreateWindowEx to now check the ExtraBytes initialized value before writing the returned value from user mode to tagWND. ExtraBytes (offset 0x128) per figure 10 below.

```
ExtraBytes = xxxClientAllocWindowClassExtraBytes(*(*(tagWND_kernel + 5) + 0xC8i64));
v278 = ExtraBytes;
if ( ExtraBytes )
{
  if ( IsWindowBeingDestroyed(tagWND_kernel) )
    goto LABEL_481;
  if ( (*(_HMPheFromObject(tagWND_kernel) + 0x19) & 1) != 0 )
    goto LABEL_481;
  v241 = 0i64;
  if ( tagWND::RedirectedFieldpExtraBytes::operator!=<unsigned __int64>(tagWND_kernel + 0x140, &v241) )
    goto LABEL_481;
  tagWND_user = *(tagWND_kernel + 5);
  if ( (*(tagWND_user + 0xE8) & 0x800) != 0 )
  {
    MicrosoftTelemetryAssertTriggeredNoArgsKM();
    tagWND_user = *(tagWND_kernel + 5);
  }
  *(tagWND_user + 0x128) = ExtraBytes;
```

Figure 10 – win32kfull!xxxCreateWindowEx (patched version)

Figure 11 below shows that tagWND. ExtraBytes is initialized to zero within xxxCreateWindowEx during normal window creation.

`*(*(tagWND + 5) + 0x128i64) = 0i64;` Figure 11 – tagWND. ExtraBytes initialization for normal window

Figure 12 below shows that tagWND. ExtraBytes is initialized to the new offset value in the kernel desktop heap within xxxConsoleControl during console window creation. RedirectFieldpExtraBytes simply checks this initialized value to determine if the window type has changed. In addition, Microsoft have also added telemetry for detecting changes to the window type flag in the patched version.

```
if ( Offset_Allocation )
{
  *Offset_Allocation = *(ConsoleCtrlInfo + 2);
  Offset_Allocation[1] = *(ConsoleCtrlInfo + 3);
}
```

Figure 12 – tagWND. ExtraBytes initialization for console window

tagWND OOB Write

The vulnerability within the xxxCreateWindowEx API allowed the WND_Malicious.ExtraBytes field be to set to a value of WND0 offset within the kernel desktop heap. Now any time SetWindowLongW is called on WND_Malicious it will write to WND0. By supplying an offset of 0xc8, the function will overwrite the WND0.cbWndExtra field to a large value of 0XFFFFFFF per figures 13 and 14 below.

This means it can write beyond its tagWND structure and ExtraBytes in kernel memory to fields within WND1. In addition, WND0.ExtraBytes is also overwritten with the offset to itself so calls to SetWindowLongPtrA on WND0 will write to an offset in kernel desktop heap relative to the start of WND0.

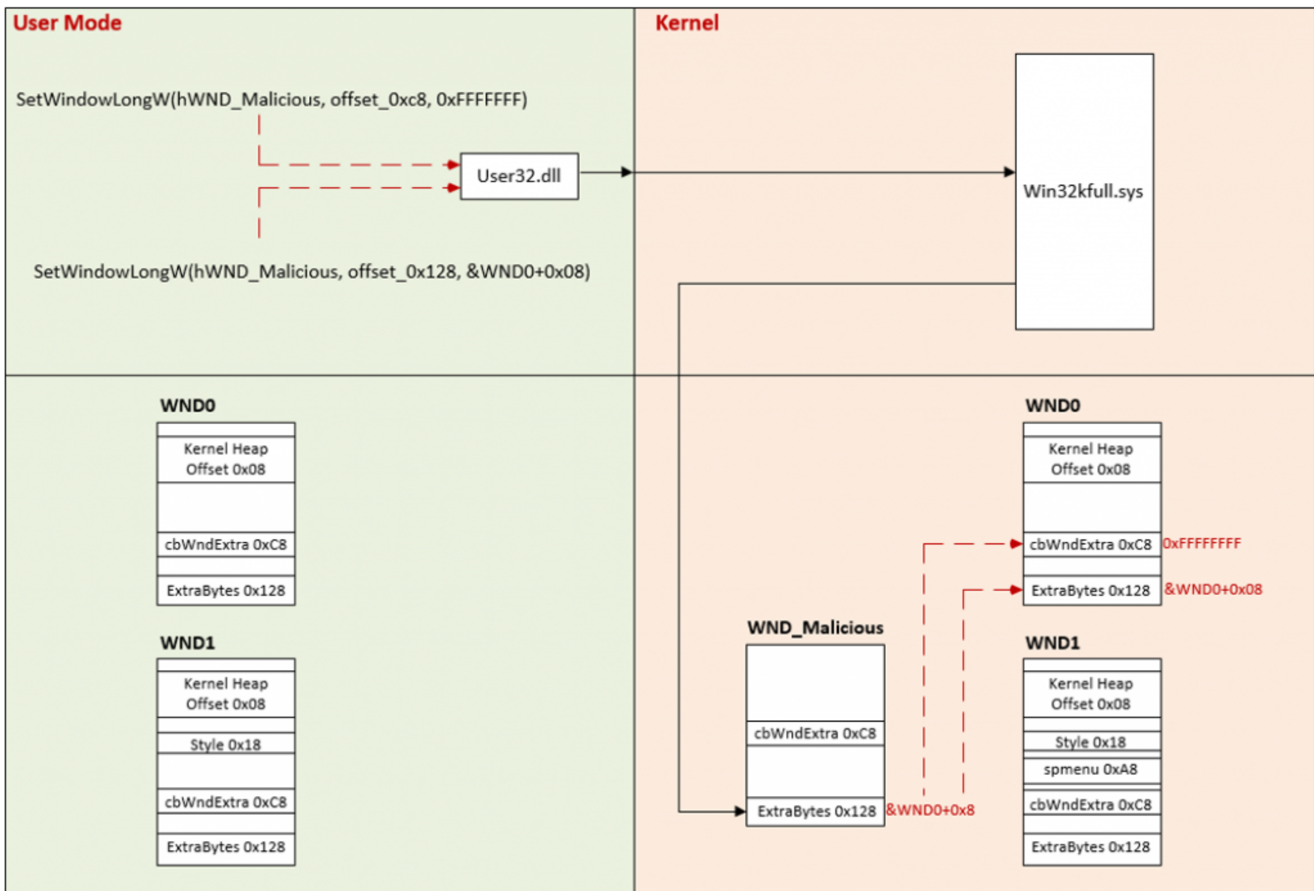


Figure 13 – OOB Write from WND_Malicious to WND0

```
0: kd> dps 0x000002B0BF34E4E0+c8
000002b0`bf34e5a8  00000000`0fffffff
```

Figure 14 – WND0 cbWndExtra overwritten with 0xFFFFFFF by WND_Malicious OOB write

Kernel Address Leak

Now that the WND0.cbWndExtra field has been set to a very large value (0xFFFFFFF), anytime SetWindowLongPtrA is called on WND0 it will write into the adjacent WND1 in kernel memory per figure 15 below. By writing to specific fields in WND1 we can create a kernel address memory leak as follows:

1. Write a value of 0x4000000000000000 to WND1 style field to temporarily change it to a child window per figures 15 and 16 below.
2. Calling SetWindowLongPtrA API on WND0 with a value of -12 (GWLP_ID) now allows the spmenu field (type tagMENU) of WND1 to be overwritten with a fake spmenu data structure since we have changed it to be a child window per figure 15 and 17 below.
3. Per SetWindowLongPtrA API documentation, the return value will give us the original value at the offset overwritten, i.e., the spmenu data structure pointer which is a kernel memory address. So, we now have leaked a pointer to a spmenu (type tagMENU) data structure in kernel memory and replaced the pointer in WND1.spmenu with a fake spmenu data structure within user desktop heap per figure 17 below.

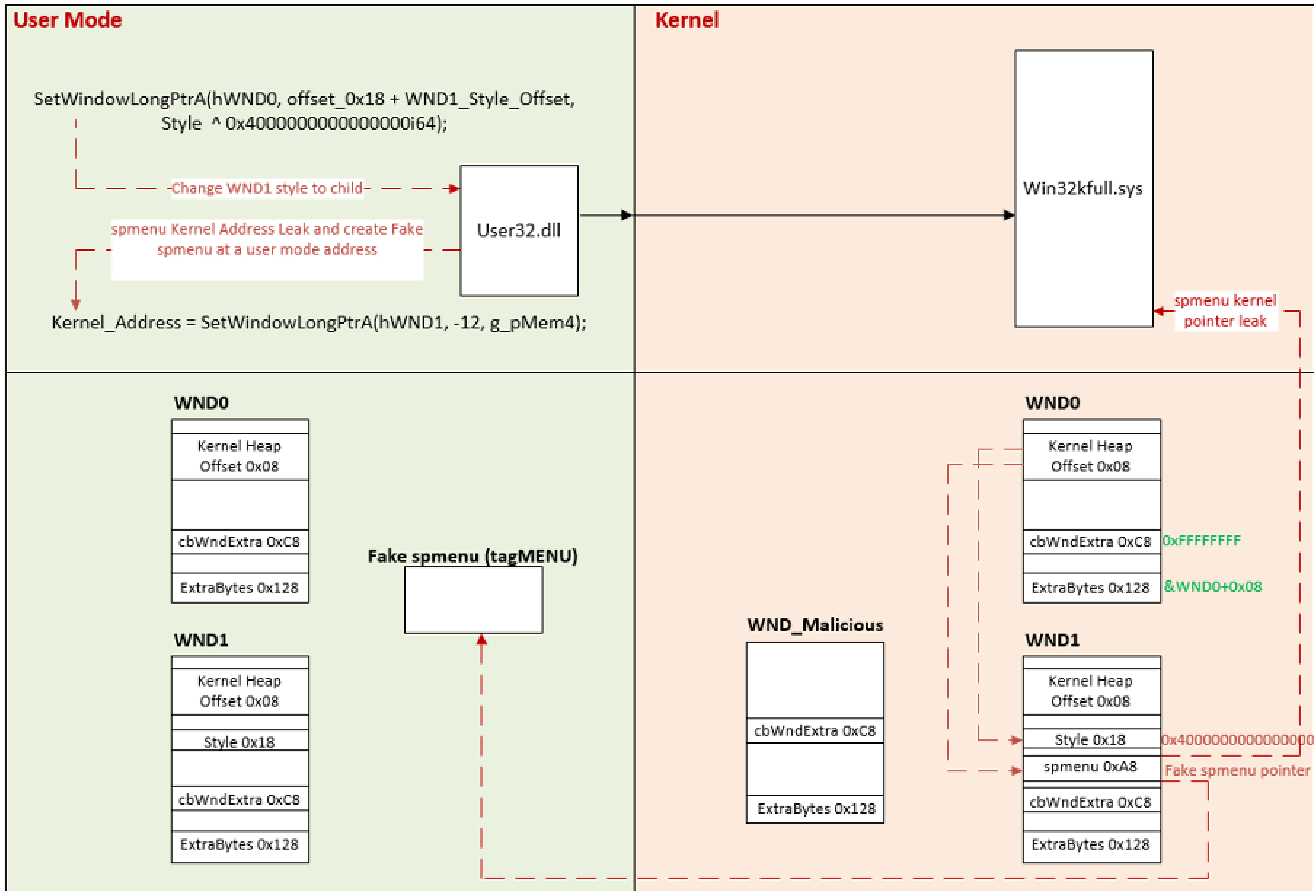


Figure 15 – OOB Write from WND0 to WND1 to Leak Kernel Address

```
0: kd> dps ffffe709`410338C0+0x18
ffffe709`410338d8  0cc00000`08000100
```

```
0: kd> dps ffffe709`410338c0+18
ffffe709`410338d8  4cc00000`08000100
```

Figure 16 – WND1 Style field before and after writing 0x4000000000000000

```
0: kd> dps ffffe709431e7540+a8
ffffe709`431e75e8  ffffe709`40824140
```

```
0: kd> dps ffffe709431e7540+a8
ffffe709`431e75e8  000002b0`bd2f6f60
```

Figure 17 – spmenu kernel memory address pointer leaked and subsequently replaced by a user mode address pointing to a fake spmenu data structure

Kernel Arbitrary Read

Using the spmenu data structure kernel pointer leaked previously we can use the layout of this data structure and the GetMenuBarInfo API logic to turn it into an arbitrary kernel memory read per figures 18,19 and 20 below.

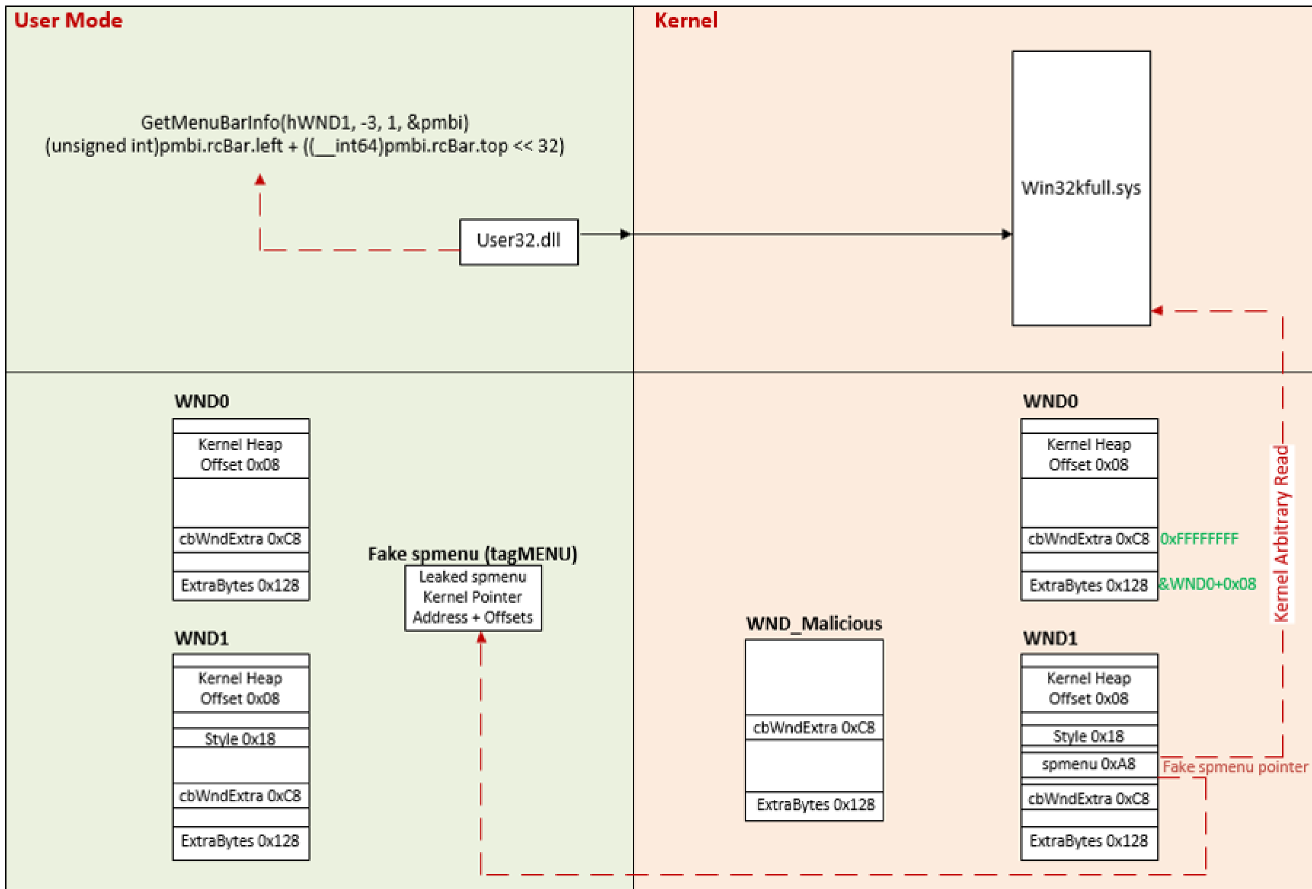


Figure 18 – Kernel Arbitrary Read using fake spmenu and GetMenuBarInfo

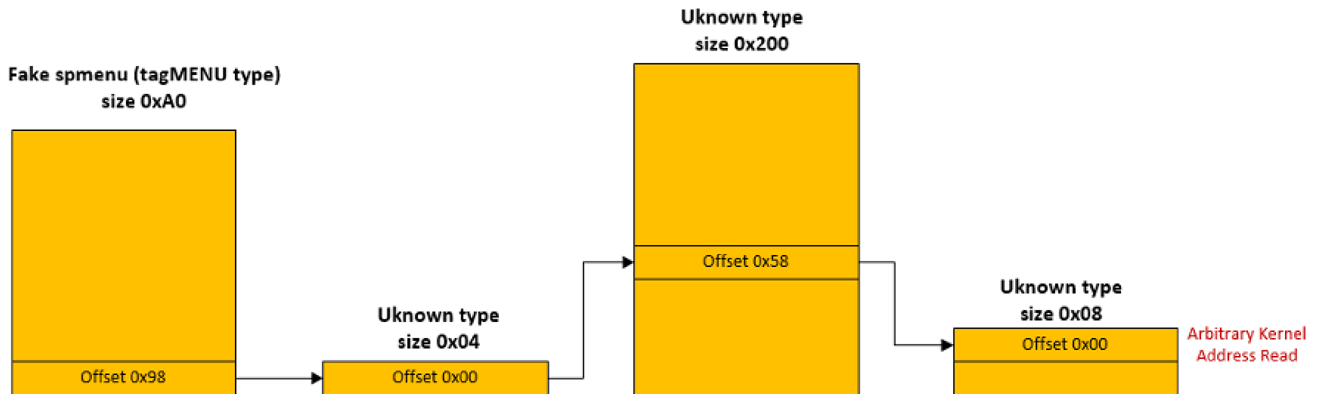


Figure 19 – Fake spmenu data structure in user desktop heap with original spmenu leaked kernel pointer at crafted location to enable arbitrary read using GetMenuBarInfo API

```
1: kd> dps poi(poi(poi(000002b0`bd2f6f60+0x98))+58)
000002b0`bd2f8ff0  ffffe709`40824150
```

Figure 20 – WinDbg command to show location within spmneu data structure that is deferenced by xxGetMenuBarInfo

As you can see from the xxxGetMenuBarInfo function in figures 21 and 22 below, by placing our leaked kernel address at the right location in our fake spmenu data structure we can create an arbitrary kernel memory read when calling GetMenuBarInfo.

```
switch ( idObject )                          // OBJID_MENU
{
  case -3:
    if ( (*(_BYTE *)(v24 + 0x1F) & 0x40) != 0 )
      goto LABEL_9;
    spmenu_ptr_kernelstruct = *(_QWORD *)(tagWND1_HND_Kernel + 0xA8);// spmenu at offset 0xA8 in Kernel tagWND1
    if ( !spmenu_ptr_kernelstruct )
      goto LABEL_9;
    v75 = 0i64;
    SmartObjStackRefBase<tagMENU>::operator=(&spmenu, spmenu_ptr_kernelstruct);
if ( v6 )
{
  tagWND1_user = *(tagWND1_HND_Kernel + 0x28);// pointer to Kernel user mode tagWND1
  v38 = 0x60 * v6;
  spmenu_0x58 = *(*spmenu + 0x58i64);   // spmenu dereference
  spmenu_kernel_leaked_address_pointer = *(0x60 * v6 + spmenu_0x58 - 0x60);// pointer to leaked kernel pointer address in fake spmenu
  if ( (*(tagWND1_user + 0x1A) & 0x40) != 0 )
  {
    v41 = *(tagWND1_user + 0x60) - *(spmenu_kernel_leaked_address_pointer + 0x40);
    *(PMENUBARINFO + 0xC) = v41;
    *(PMENUBARINFO + 4) = v41 - *(*(v38 + spmenu_0x58 - 0x60) + 0x48i64);
  }
  else
  {
    Leaked_KernelAddress = *(spmenu_kernel_leaked_address_pointer + 0x40) + *(tagWND1_user + 0x58);// Read memory at kernel pointer address in fake spmenu
    *(PMENUBARINFO + 4) = Leaked_KernelAddress;
    *(PMENUBARINFO + 12) = Leaked_KernelAddress + *(*(v38 + spmenu_0x58 - 0x60) + 0x48i64);
  }
  v43 = *(*(v38 + spmenu_0x58 - 0x60) + 68i64) + *(*(tagWND1_HND_Kernel + 0x28) + 0x5Ci64);
  *(PMENUBARINFO + 8) = v43;
  v36 = v43 + *(*(v38 + spmenu_0x58 - 0x60) + 0x4Ci64);
}
```

Figure 21 – win32kfull!xxxGetMenuBarInfo

```
1: kd> dps r9
000000ea`c5da23e8  00000040`00000030
000000ea`c5da23f0  00000088`00000044
000000ea`c5da23f8  00000000`00000090
000000ea`c5da2400  00000000`88888888
000000ea`c5da2408  00000000`00000000
000000ea`c5da2410  00000000`00000000
```

```
1: kd> dps 000000ea`c5da23e8
000000ea`c5da23e8  431e7540`00000030
000000ea`c5da23f0  431e7540`ffffe709
000000ea`c5da23f8  00000000`ffffe709
000000ea`c5da2400  00000000`88888888
000000ea`c5da2408  00000000`00000000
000000ea`c5da2410  00000000`00000000
```

Figure 22 – GetMenuBarInfo data structure populated return values per normal spmenu and fake spmenu (leaks kernel address)

Kernel Arbitrary Write

An arbitrary kernel write primitive can be easily achieved now by writing our destination address to WND1.ExtraBytes field by calling SetWindowLongPtrA on WND0 which will write OOB to WND1 relative to the offset we specify per figure 23 below

In this case the offset is 0x128 which is ExtraBytes. Then simply calling SetWindowLongPtrA on WND1 will write a specified value at the address placed in the WND1.ExtraBytes field. The arbitrary write is achieved because WND1 is a normal window (has not been converted to a console window like WND0 and WND_Malicious) and so will write to whatever address we place in WND1.ExtraBytes.

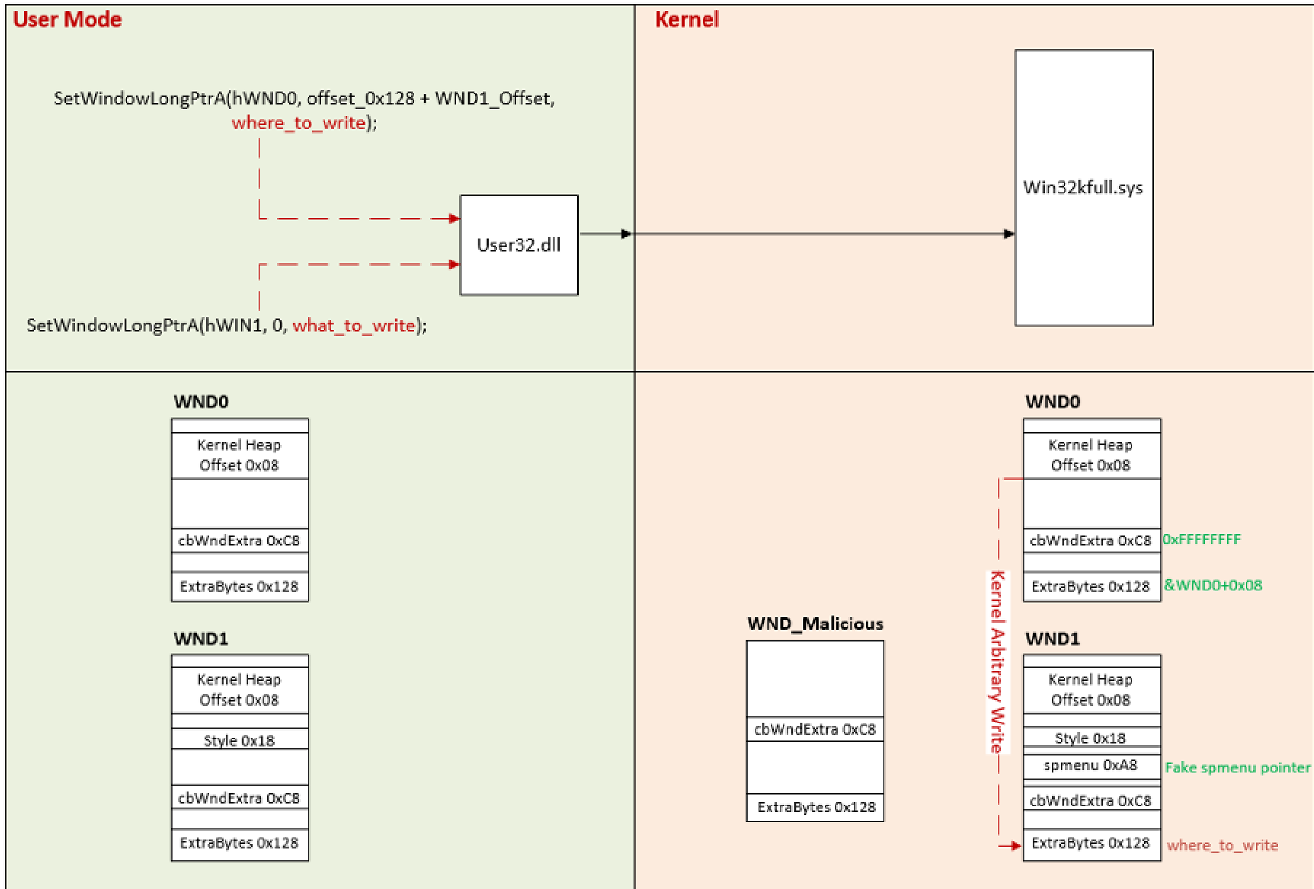


Figure 23– Kernel Arbitrary Write for What-Write-Where (WWW)

Data Only Attack

The arbitrary kernel read and write primitives can be combined to perform a data-only attack to overwrite a malicious process EPROCESS token with that of PID 4 which is System for an escalation of privilege (EoP).

The original spmenu kernel address leaked previously has a pointer to WND1 at offset 0x50 per figures 24 and 25 below. Through multiple arbitrary reads using the GetMenuBarInfo on our fake spmenu data structure with this WND1 kernel address we can eventually read the PID 4 System EPROCESS token.

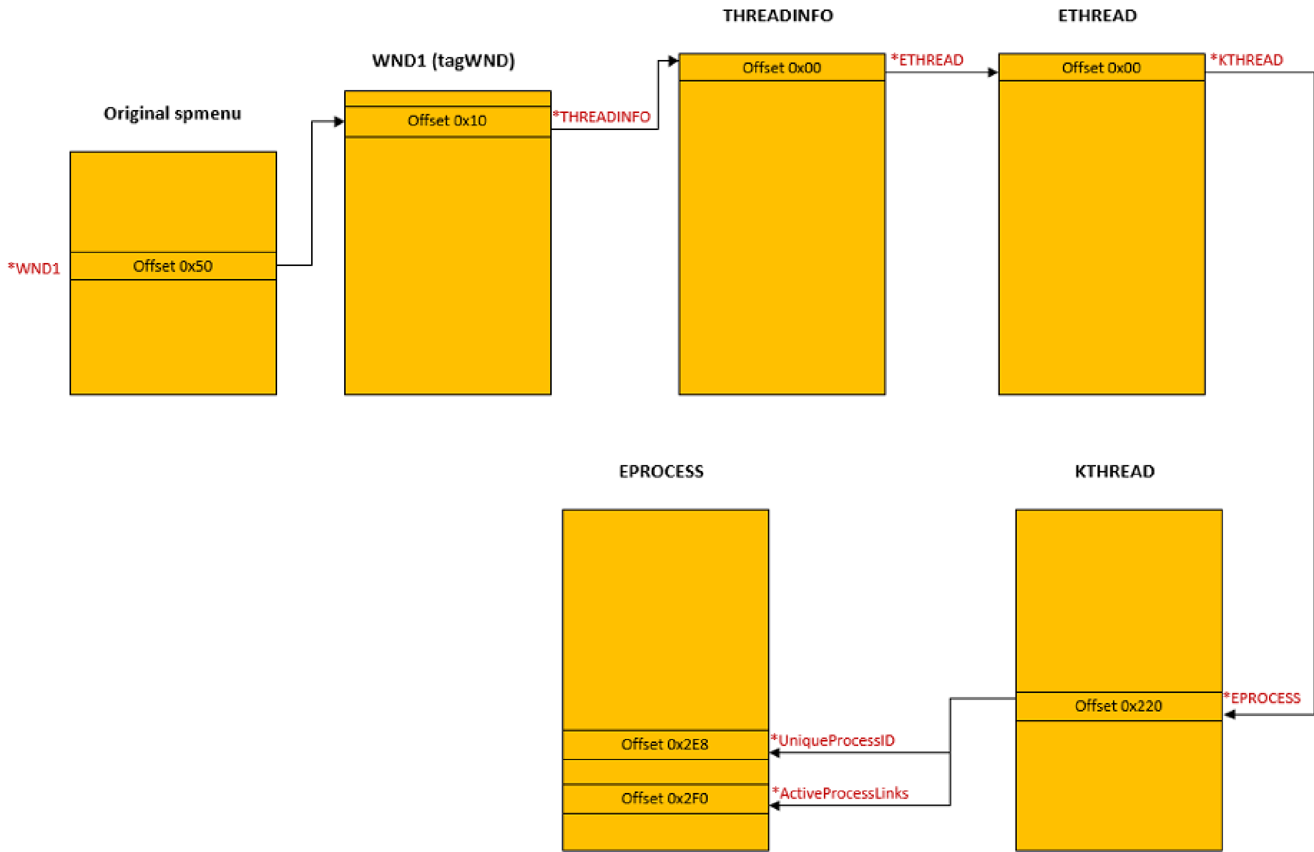


Figure 24 – Combining fake spmenu with GetMenuBarInfo arbitrary read to get PID 4 token

```
0: kd> dps ffffe709`40824140 L14
ffffe709`40824140  00000000`0005013d
ffffe709`40824148  00000000`00000001
ffffe709`40824150  00000000`00000000
ffffe709`40824158  ffffc28f`311164b0
ffffe709`40824160  ffffe709`40824140
ffffe709`40824168  ffffe709`41033870
ffffe709`40824170  00000000`00033870
ffffe709`40824178  00000000`00000000
ffffe709`40824180  00000000`00000000
ffffe709`40824188  00000000`00000000
ffffe709`40824190  ffffe709`431e7540
ffffe709`40824198  00000000`00000000
ffffe709`408241a0  00000000`00000000
ffffe709`408241a8  00000000`00000000
ffffe709`408241b0  00000000`00000000
ffffe709`408241b8  00000000`00000000
ffffe709`408241c0  00000000`00000000
ffffe709`408241c8  00000000`00000000
ffffe709`408241d0  00000000`00000000
ffffe709`408241d8  ffffe709`44d37230
```

Figure 25– Original spmenu with WND1 kernel address pointer at offset 0x50

By placing the destination address (malicious process EPROCESS token) at WND1.ExtraBytes then the subsequent call to SetWindowLongPtrA will write the value (PID 4 – System EPROCESS token) to that address per figures 26 and 27 below.

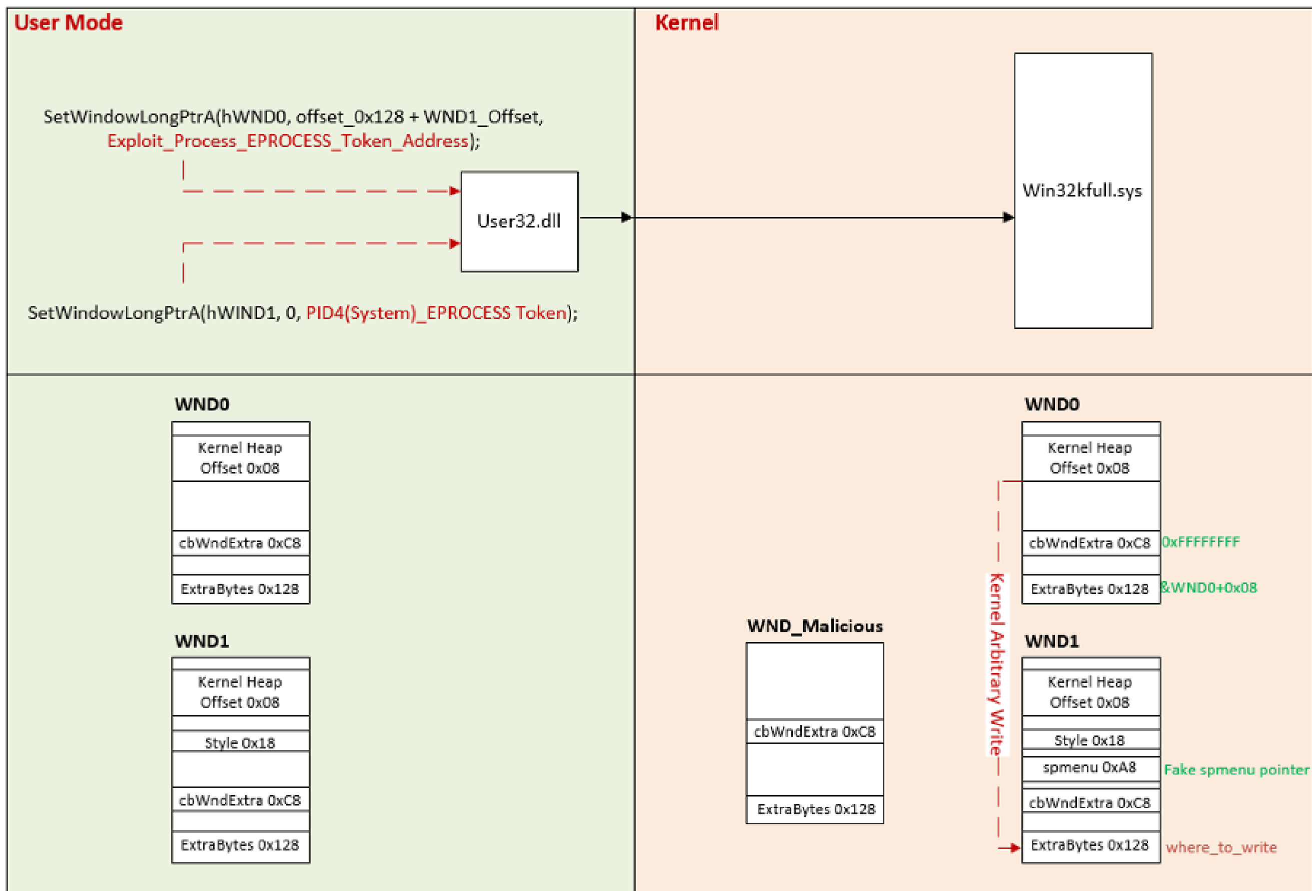


Figure 26 – EPROCESS Token swap

```
1: kd> dps ffffe709`410338c0+128
ffffe709`410339e8  000002b0`bd4b9fe0
```

```
1: kd> dps ffffe709`410338c0+128
ffffe709`410339e8  ffffc28f`33e913e0
```

Figure 27 – Overwriting WND1.ExtraBytes with address of EPROCESS token

The exploit then restores overwritten data structure values once the EoP is complete to prevent a BSOD (Blue Screen of Death).

Conclusion

In this report, we undertook a deep analysis of CVE-2021-1732 which is a Local Privilege Escalation on Windows 10. Windows kernel data-only attacks are difficult to defend against, as once a vulnerability is discovered they use legitimate and trusted code through specific APIs to manipulate data structures in kernel memory.

The win32k component has been hardened through great work by Microsoft against read/write primitives, but there are still opportunities for exploitation due to its large attack surface (syscalls and callbacks) and lack of win32k filtering on a process-wide basis. It would also be great to see a system wide win32k filtering policy capability within Windows 10.

Patching is always the best solution for vulnerabilities, but a strong defense strategy such as threat hunting is also required where patching may not be possible, and to detect variants of vulnerabilities/exploits being used by campaigns.