BlueKeep: A Journey from DoS to RCE (CVE-2019-0708)

Due to the serious risk of a BlueKeep based worm, I've held back this write-up to avoid advancing the timeline. Now that a proof-of-concept for RCE (remote code execution) has been release as part of Metasploit, i feel it's now safe for me to post this.

This article will be a follow on from my previous analysis (https://www.malwaretech.com/2019/05/analysis-of-cve-2019-0708-bluekeep.html).

Be free

As I mentioned in the previous article, we are able to free the data structure associated with the MS_T120 channel. Freeing the structure alone isn't of much use, but controlling its content is. With a UAF (use-after-free), the goal is to free an object, then allocate a fake one in its place. By replacing the content of a real object with our own data, we gain more extensive control over the code utilizing it. What we can do with our fake channel structure depends entirely on what the structure is used for (we'll get to this later).

A hole in my soul

After the UAF is triggered, the MS_T120 channel structure is deallocated, but still usable. To exploit this UAF, we must allocate new data at the same address where the channel structure was previously located.

Channel structures are allocated in the Non-paged Pool. In order to hijack the freed structure, we need to be able to allocate arbitrary data in the Non-paged Pool. To figure out a way, I analyzed calls to variants of ExAllocatePool (the kernel equivalent of malloc).

```
220  pool_size = channel_data_size + 0x38i64;
221  if ( pool_size < channel_data_size || pool_size < 0x38 )
222    pool_chunk = 0i64;
223  else
224    pool_chunk = ExAllocatePoolWithTag(0, pool_size, 'ciST');
225  if ( pool_chunk )
226  {
227    *((_DWORD *)pool_chunk + 6) = channel_data_size;
228    *((_DWORD *)pool_chunk + 7) = channel_data_size;
229    *((_QWORD *)pool_chunk + 2) = (char *)pool_chunk + 0x38;
230    memmove((char *)pool_chunk + 0x38, channel_data, channel_data_size);
```

(https://www.malwaretech.com/wp-content/uploads/2019/07/HeapSpray.png)

An ideal looking Non-paged Pool allocation inside IcaChannelIntputInternal.

This is basically the holy grail of heap spraying. Data sent to the channel is stored on the Non-paged Pool, so we can perform Non-paged pool allocations of any size we want, containing any data we

want!

The only caveat is this allocation is for an IOCP queue, which the channel handlers processes messages from. Once a message is read, it is removed from the queue and deallocated. However, a couple of researcher found that a certain channel doesn't ever read messages from the queue, leaving them allocated until the connection is closed.

Now, if we spam allocations of the same size as the channel structure, it's likely one will land in the hole left by the freed channel (this is known as heap spraying). To increase likelihood of a successful allocation, we can perform something known as heap grooming to ensure nothing else is allocated at that address; however, this is a complex topic with plenty of write-ups, so I'll refrain from getting into that here.

Assuming Control

The channel structure which we're able to replace is returned by the functions IcaFindChannel and IcaFindChannelByName. In order to find what can be done by manipulating this channel structure, I needed to find where it was used. After doing an xref of IcaFindChannel, I found the following code.

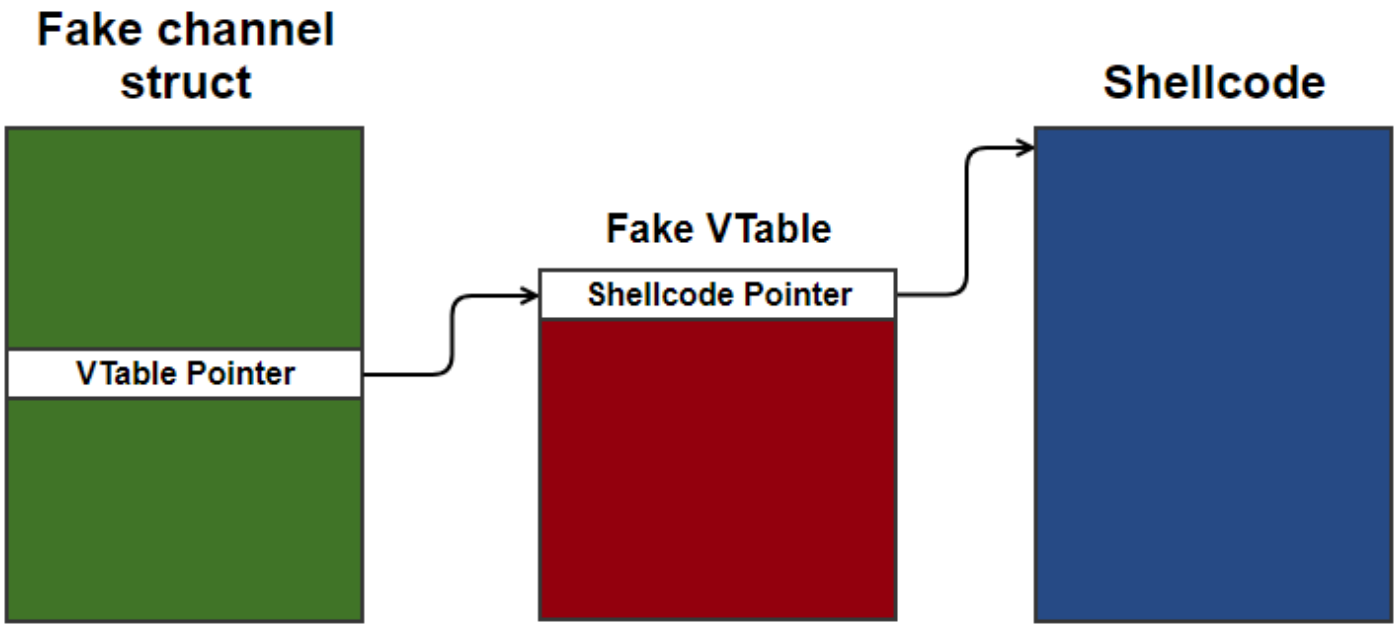
```
 76  v16 = *(_RDI + 0x1D);
 77  v45 = *(_RDI + 0x1D);
 78  channel_struct = IcaFindChannel(v45, a2, v9);
 79  if ( !channel_struct )
 80  {
 81LABEL_77:
 82    if ( v8 )
 83      ExFreePoolWithTag(v8, 0);
 84    return 0i64;
 85  }
 86  _InterlockedAdd(&channel_struct->field_10, 1u);
 87  v19 = &channel_struct->field_18;
 88  ExEnterCriticalRegionAndAcquireResourceExclusive(&channel_struct->field_18);
 89  v20 = HIDWORD(channel_struct->field_E8);
 90  if ( v20 & 0x28 || *(_RDI + 32) == 1 && !(v20 & 2) )
 91  {
 92    ExReleaseResourceAndLeaveCriticalRegion(v19);
 93    IcaDereferenceChannel(channel_struct);
 94    IcaDereferenceChannel(channel_struct);
 95    goto LABEL_77;
 96  }
 97  if ( v8 )
 98  {
 99    v7 = *(v8 + 2);
100    v6 = *(v8 + 6);
101  }
102  vtable = channel_struct->vtable;
103  if ( vtable )
104  {
105    (*vtable)(channel_struct->vtable, v7, v6, &v44);
106    if ( v8 )
107      ExFreePoolWithTag(v8, 0);
108    v8 = v44;
109    v7 = *(v44 + 16);
110    v6 = *(v44 + 24);
111  }
```

(https://www.malwaretech.com/wp-content/uploads/2019/07/IcaChannelInputInternal1.png)

A snipped from IcaChannelInputInternal, the function which handles virtual channel input.

The above code checks if the “vtable” field of the channel struct is non-zero, then if so, dereferences the address and performs an indirect call to it. To exploit this, we’d need to point channel_struct->vtable to some memory we control, then store a pointer to our shellcode in said memory.

The memory layout for RCE would need to look like this.



(https://www.malwaretech.com/wp-content/uploads/2019/07/IndirectVtableCall.png)

An indirect call via a fake VTable.

Unfortunately, there are some problems.

1. We need the fake channel structure to point to the fake VTable, but we don’t have the address of the fake vtable.
2. We need the fake VTable to point to the shellcode, but we don’t have the address of the shellcode.

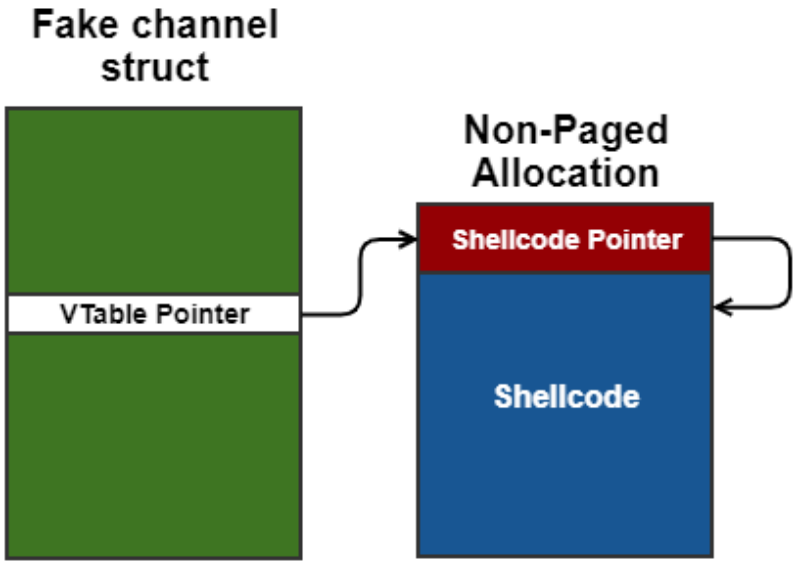
But fear not, there’s always a way!

No Execute!!! Ok, maybe just a little.

As someone who only works with Windows 10, I initially went down more complex path of trying to bypass both ASLR (Address Space Layout Randomization) and DEP (Data Execution Prevention AKA No Execute) at the same time. That was when Ryan Hanson (@ryHanson) gave me the following hint “There is something even more special about that spray. Keep in mind it is Win7”.

Of course! It’s Windows 7! There is no DEP when it comes to non-paged kernel memory. We can literally just write shellcode directly into the kernel (remember the heap spray method I explained allows us to allocate arbitrary data on the Non-Paged Pool).

The refined memory layout now looks something like this.



(https://www.malwaretech.com/wp-content/uploads/2019/08/IndirectVtableCall2.png)

Both the VTable and the Shellcode can be in the same memory buffer, because it's RWX (Readable, Writable, Executable).

We don't need a ROP chain to write the shellcode to executable memory like we would in user mode, we can just spray it to the Non-paged Pool. Only one last problem, how do we get the address of the allocation to set the VTable pointer in the channel struct?

Spray and Pray

On Windows 7 and below, the Non-paged Pool starts at a fixed address (Nt!MmNonPagedPoolStart). We know where the Non-paged Pool starts, but we don't know where within the Non-pagedPool our shellcode resides….But we can control that somewhat.

Non-paged memory is sacred because its size is limit due to lack of paging. As a result, the kernel generally avoids using it for anything that doesn't absolutely need to be there. That is, the memory utilization is fairly low.

What we can do is pick an address (say 500 MB into the Non-paged Pool) which is unlikely to be un-allocated due to low memory utilization. Next, we spray enough copies of the shellcode to fill the Non-Paged Pool past our chosen address, leading to a high probability a copy of the shellcode will reside there.

Due to the fact most objects frequently allocated on the Non-paged Pool are very small, we can create lots of holes smaller than our shellcode size (prior to the chosen address), ensuring other allocations are allocated there instead.

We then can set the VTable pointer in our fake channel structure to our chosen address, resulting in

the shellcode being executed whenever the channel receives a message.

Conclusion

Part of the reason this vulnerability is so dangerous is the lack of real mitigations on the affected system. Absence of DEP in the kernel, and lots of well documented static addresses, means the need for an address leak + ROP chain is eliminated. Furthermore, there presence of a VTable already provides an arbitrary execute primitive, making for very little work to turn the DoS into RCE.

Shout out to zerosum0x0 (https://twitter.com/zerosum0x0) and Ryan Hanson (https://twitter.com/ryHanson) for their great work on BlueKeep, as well as everyone who held back information on the vulnerability to give organizations months to patch.