# Introduction

The following text outlines a potential path for exploitation of CVE-2019-0708 (BlueKeep). It is certain that some people will disagree with releasing this text. Reasons why I am releasing:

- It is released in the spirit of open knowledge.
- It is an attempt to give back to the hacker community from whom I have learned so much thanks to their willingness to share information.
- The information within here is largely already available within the Chinese hacker community [1].
- The attack path that follows is geared towards Windows XP, while it may be technically possible on Windows 7 or Server 2008, it is more likely to BSOD.
- No exploit code or ring 0 to ring 3 shellcode is shared within this text. (RDP connection code is available at [2])
- Details are left out.

# Technical Analysis

CVE-2019-0708 affects Windows XP through Windows Server 2008. A use after free (UAF) condition exists within the termdd.sys RDP kernel driver. A remote, unauthenticated attacker can exploit this vulnerability by establishing an RDP connection to the target server, opening an MS_T120 virtual channel, and sending crafted data to it. Successful exploitation will result in the attacker executing arbitrary code with kernel-level privileges or causing a denial-of-service. For a full detailed analysis how to trigger the UAF condition consult [3]. The following analysis will assume a base level of knowledge from the previous article.

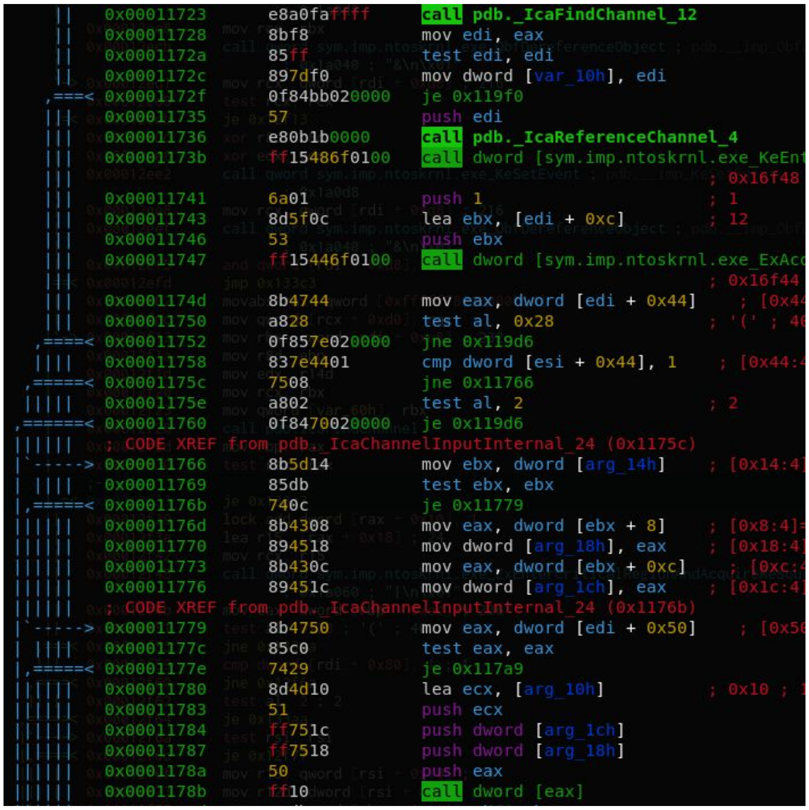
A first step after understanding how to trigger the UAF is to understand how the dangling pointer is used after it is freed. Due to our knowledge from [3] we know that the dangling pointer is returned by IcaFindChannel within the IcaChannelInputInternal function within termdd.sys. A good place to start then is analyzing the code after the IcaFindChannel call. To disassemble termdd.sys I will be utilizing radare2 [4]. This will not be a radare2 tutorial, but the following commands will help you get started if you would like to follow along (for more information on using radare2 see [5]):

Open termdd.sys with radare2:
```
# r2 termdd.sys
```
Download the debugging symbols:
```
> idpd
```
Load the debugging symbols:
```
> idp
```

Run analysis:

```
> aaa
```

After seeking to IcaChannelInputInternal and reviewing the code after IcaFindChannel we see the following:




We first see that the dangling pointer (in `eax` after return from IcaFindChannel) is moved into `edi`. Thus for now we are largely concerned with instructions that deal with `edi`. After reviewing this set of instructions something very interesting stands out immediately. At `0x11779` we see the instruction `mov, eax, dword [edi + 0x50]`, and then 8 instructions later at `0x1178b` we see `call dword [eax]`. Already we can see how we might control EIP!

It is useful to take a step back and think about the vulnerability class and how we might be able to exploit this instance of it. A use after free is exactly its name - memory is used after it has been freed. Our dangling pointer in `edi` is pointing to memory that has been returned (freed) to the memory manager. The memory manager can now allocate that same memory to another requestor. In other words, `edi` is referencing invalid data within the context of the IcaChannelInputInternal function. This will inevitably cause a blue screen of death, or arbitrary code execution if we have anything to do about it :).

Given this information we formulate a high level attack plan:

1. Establish an RDP connection with the MS_T120 virtual channel.
2. Send specific data on MS_T120 virtual channel to free channel control structure.
3. Invoke an allocation with data controlled by us to occupy the freed channel control structure memory space.
4. Control EIP

To accomplish step 3 we need to first understand a few things. Namely, what is the size of the channel control structure and what type of memory it is (paged or non-paged). This is best accomplished using Windbg. We set a breakpoint on IcaFindChannel within IcaChannelInputInternal and send data on the MS_T120 channel. We see our sent data is at `ebp+18`, and the channel control structure pointer is `0x8238ccb8`.

```
kd> dd poi(ebp+18)
e131e2db  41414141 41414141 41414141 41414141
e131e2eb  41414141 41414141 41414141 41414141
e131e2fb  41414141 41414141 41414141 41414141
e131e30b  41414141 41414141 41414141 41414141
e131e31b  41414141 41414141 41414141 41414141
e131e32b  41414141 41414141 41414141 41414141
e131e33b  41414141 41414141 41414141 41414141
e131e34b  41414141 41414141 41414141 41414141
kd> p
termdd!IcaChannelInputInternal+0xb8:
f887b728 8bf8            mov     edi,eax
kd> r eax
eax=8238ccb8
```

Next we use the handy `!pool` command to find more about this allocated memory:

```
Pool page 8238ccb8 region is Nonpaged pool
 8238c000 size:   40 previous size:    0  (Allocated)  Ntfr
 8238c040 size:    8 previous size:   40  (Free)       ...x
...
 8238cc48 size:   28 previous size:   98  (Allocated)  NtFs
 8238cc70 size:   40 previous size:   28  (Allocated)  Ntfr
*8238ccb0 size:   98 previous size:   40  (Allocated) *Ica
            Owning component : Unknown (update pooltag.txt)
 8238cd48 size:   68 previous size:   98  (Allocated)  MmCa
 8238cdb0 size:   10 previous size:   68  (Free)       CcPL
```

From this output we can see that the allocated memory region is non-paged pool. Further down we see our channel control structure memory. It starts at `0x8238ccb0` and is size `0x98`.

We have our size and memory type, but how actually to allocate the memory? Allocation requests for pool memory are typically serviced through the function call ExAllocatePoolWithTag (see [6] for more information on windows pool). We need to locate this function call, but not any one will do. We have very strict requirements:

1. Must allocate non-paged memory.
2. Must be able to allocate an arbitrary size controlled by us.

3. Must eventually contain data controlled by us.

The code base for RDP is huge and built upon many layers. There are lots of places we could potentially look. However, it is best to start simple and start the search in termdd. Going back to radare2 we first locate ExAllocatePoolWithTag:

```
[0x000116e8]> ii ~ExAllocatePoolWithTag
  5 0x00016f34    NONE    FUNC ntoskrnl.exe_ExAllocatePoolWithTag
```

Next we want to find all the references to it:

```
[0x00016f14]> axt @ 0x00016f34
pdb.__IcaLoadSdWorker_8 0x1040e [DATA] mov esi, dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb.__IcaLoadSdWorker_8 0x1049f [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb._IcaBufferAllocInternal_24 0x10ae8 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb._IcaBufferAllocInternal_24 0x10b3c [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb.__IcaCopyDataToUserBuffer_12 0x10d87 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb.__IcaRegisterVcBind_16 0x10f9b [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb._IcaChannelInputInternal_24 0x11905 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb.__IcaAllocateChannel_12 0x11c22 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb.__IcaAllocateConnection_0 0x124af [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
(nofunc) 0x1276f [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb._IcaIsSystemProcessRequest_8 0x12b7d [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb._IcaInitializeData_0 0x13012 [DATA] mov esi, dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb.__IcaAllocateStack_0 0x13419 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb.__IcaDereferenceSdLoad_4 0x13534 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
sym.TERMDD.SYS_IcaStackAllocatePoolWithTag 0x137bc [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
sym.TERMDD.SYS_IcaStackAllocatePool 0x137ee [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb.__IcaLoadSd_8 0x13c16 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
sym.TERMDD.SYS_IcaCreateThread 0x142fd [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
(nofunc) 0x147d4 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
(nofunc) 0x1496a [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
(nofunc) 0x14a73 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
(nofunc) 0x14e09 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
sym.TERMDD.SYS_IcaAllocateWorkItem 0x15646 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
sym.TERMDD.SYS_IcaTimerStart 0x15862 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
sym.TERMDD.SYS_IcaQueueWorkItemEx 0x15995 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
sym.TERMDD.SYS_IcaTimerCreate 0x15a55 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb.__IcaTraceWrite_8 0x15eb6 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb.__IcaOpenTraceFile_8 0x1620b [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
(nofunc) 0x16653 [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
entry0 0x1844e [CALL] call dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
pdb._PtEntry_8 0x1852c [DATA] mov ebx, dword sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag
```

One stands out as particularly interesting, and that is the reference within IcaChannelInputInternal. Let's review the code:

```
[0x118fe]
push 0x20616349
push eax
push ebx
call dword [sym.imp.ntoskrnl.exe_ExAllocatePoolWithTag]
mov ebx, eax
jmp 0x11911
```

Reviewing the arguments for ExAllocatePoolWithTag [7]:

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

Promising, as neither the size nor the pool type are hardcoded. Let’s place a breakpoint on this call within IcaChannelInputInternal and send variable size data on our virtual channel:

```
Breakpoint 0 hit
termdd!IcaChannelInputInternal+0x295:
f887b905 ff15340f88f8    call    dword ptr [termdd!_imp__ExAllocatePoolWithTag (f8880f34)]
kd> dd poi(ebp+18)
81c1fe82  41414141 41414141 41414141 41414141
81c1fe92  41414141 41414141 41414141 41414141
81c1fea2  41414141 41414141 41414141 41414141
81c1feb2  41414141 41414141 41414141 41414141
81c1fec2  41414141 41414141 41414141 41414141
81c1fed2  41414141 41414141 41414141 41414141
81c1fee2  41414141 9b000003 6480f002 ef030700
81c1fef2  088c8070 29000000 d6e2b8bc b4a2b278
kd> dd esp
b2a6d3c0  00000000 00000084 20616349 82143008
b2a6d3d0  00000000 e120b3a8 90e28656 80a6eee4
```

Here we see we are within IcaChannelInputInternal with the data we sent. PoolType is `0` (non-paged) and size is `0x84`. This call then meets requirement 1. Going to the next packet we sent:

```
termdd!IcaChannelInputInternal+0x295:
f887b905 ff15340f88f8    call    dword ptr [termdd!_imp__ExAllocatePoolWithTag (f8880f34)]
kd> dd poi(ebp+18)
81c1ff09  41414141 41414141 41414141 41414141
81c1ff19  41414141 41414141 41414141 41414141
81c1ff29  41414141 41414141 41414141 41414141
81c1ff39  41414141 41414141 41414141 41414141
81c1ff49  41414141 41414141 41414141 41414141
81c1ff59  41414141 41414141 41414141 41414141
81c1ff69  41414141 41414141 41414141 41414141
81c1ff79  41414141 41414141 af000003 6480f002
kd> dd esp
b2a6d3c0  00000000 00000098 20616349 82143008
b2a6d3d0  00000000 e120b3a8 79400804 51f32257
```

The next size is 0x98. It is looking very likely that we directly control the size of this allocation. This allocation is looking very promising, to find out more about how the allocated memory is used let’s place a read/write breakpoint on it and continue execution:

```
kd> r eax
eax=8237ca40
kd> ba r4 8237ca40
```

A few instructions later our breakpoint is hit within the same function:

```
kd> g
Breakpoint 1 hit
termdd!IcaChannelInputInternal+0x2d6:
f887b946 894b04          mov     dword ptr [ebx+4],ecx
kd> ub
termdd!IcaChannelInputInternal+0x2c1:
f887b931 8bc8            mov     ecx,eax
f887b933 83e103          and     ecx,3
f887b936 f3a4            rep movs byte ptr es:[edi],byte ptr [esi]
f887b938 8b7df0          mov     edi,dword ptr [ebp-10h]
f887b93b 8b751c          mov     esi,dword ptr [ebp+1Ch]
f887b93e 8d4774          lea     eax,[edi+74h]
f887b941 8b4804          mov     ecx,dword ptr [eax+4]
f887b944 8903            mov     dword ptr [ebx],eax
kd> r
eax=8205ca1c ebx=8237ca40 ecx=823d7710 edx=02e00002 esi=00000078 edi=8205c9a8
eip=f887b946 esp=b2a6d3cc ebp=b2a6d400 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
termdd!IcaChannelInputInternal+0x2d6:
f887b946 894b04          mov     dword ptr [ebx+4],ecx ds:0023:8237ca44=8237ca48
```

Looking at the previous instructions `rep movs byte ptr es:[edi], byte ptr [esi]` stands out as it is used to copy memory from one buffer to another. Let's check our pointer that was returned from the ExAllocatePoolWithTag call:

```
kd> dd 8237ca40 L50
8237ca40  8205ca1c 8237ca48 8237ca60 00000078
8237ca50  00000078 005c0065 00610048 00640072
8237ca60  41414141 41414141 41414141 41414141
8237ca70  41414141 41414141 41414141 41414141
8237ca80  41414141 41414141 41414141 41414141
8237ca90  41414141 41414141 41414141 41414141
8237caa0  41414141 41414141 41414141 41414141
8237cab0  41414141 41414141 41414141 41414141
8237cac0  41414141 41414141 41414141 41414141
8237cad0  41414141 41414141 0a050014 6d665346
```

That checks off requirement #3. IcaChannelInputInternal is truly a function sent by the RDP exploit gods. It contains everything we need for RCE.

Further filling out the attack plan we now have:

1. Establish an RDP connection with the MS_T120 virtual channel.
2. Send specific data on MS_T120 virtual channel to free channel control structure.
3. Invoke allocations via call to ExAllocatePoolWithTag in IcaChannelInputInternal such that the freed memory space is occupied with our data.
4. Control EIP via vtable call by placing function pointer to our shellcode at `[edi + 50]` within our fake allocated channel control structure.
5. Break the connection to trigger UAF
6. Obtain RCE

Items to think about:

1. Where is our shellcode located?
2. Can we run any plain old userspace shellcode?
3. Are we able to send data on a channel that we've closed?
4. If we've accomplished the above how do we exit cleanly such that we don't immediately BSOD after shellcode executes?

These are all exercises left to the reader :).

## Conclusion

The previous text outlined a potential path for exploitation of CVE-2019-0708 (BlueKeep). It is my hope (@0xeb_bp) that someone somewhere learned something. Before diving into this I had admittedly never even opened Windbg and had 0 exploitation experience with the Windows kernel. The journey through this taught me so much and I am very excited to move into more Windows kernel exploitation.

I could not have done any of this without people who have written and done so much work and shared it for everyone. Special thanks to @FuzzySec @stephenfewer @epakskape @aionescu @trufae and @kernelpool.

## References

1. https://github.com/blackorbird/APT_REPORT/blob/master/exploit_report/%23bluekeep%20RDP%20from%20patch%20to%20remote%20code%20execution.pdf
2. https://github.com/0xeb-bp/bluekeep
3. https://www.zerodayinitiative.com/blog/2019/5/27/cve-2019-0708-a-comprehensive-analysis-of-a-remote-desktop-services-vulnerability
4. https://github.com/radare/radare2
5. https://www.megabeets.net/a-journey-into-radare-2-part-1/
6. https://media.blackhat.com/bh-dc-11/Mandt/BlackHat_DC_2011_Mandt_kernelpool-wp.pdf
7. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-exallocatepoolwithtag
8. http://www.uninformed.org/?v=3&a=4&t=pdf
9. https://www.fuzzysecurity.com/tutorials/expDev/8.html