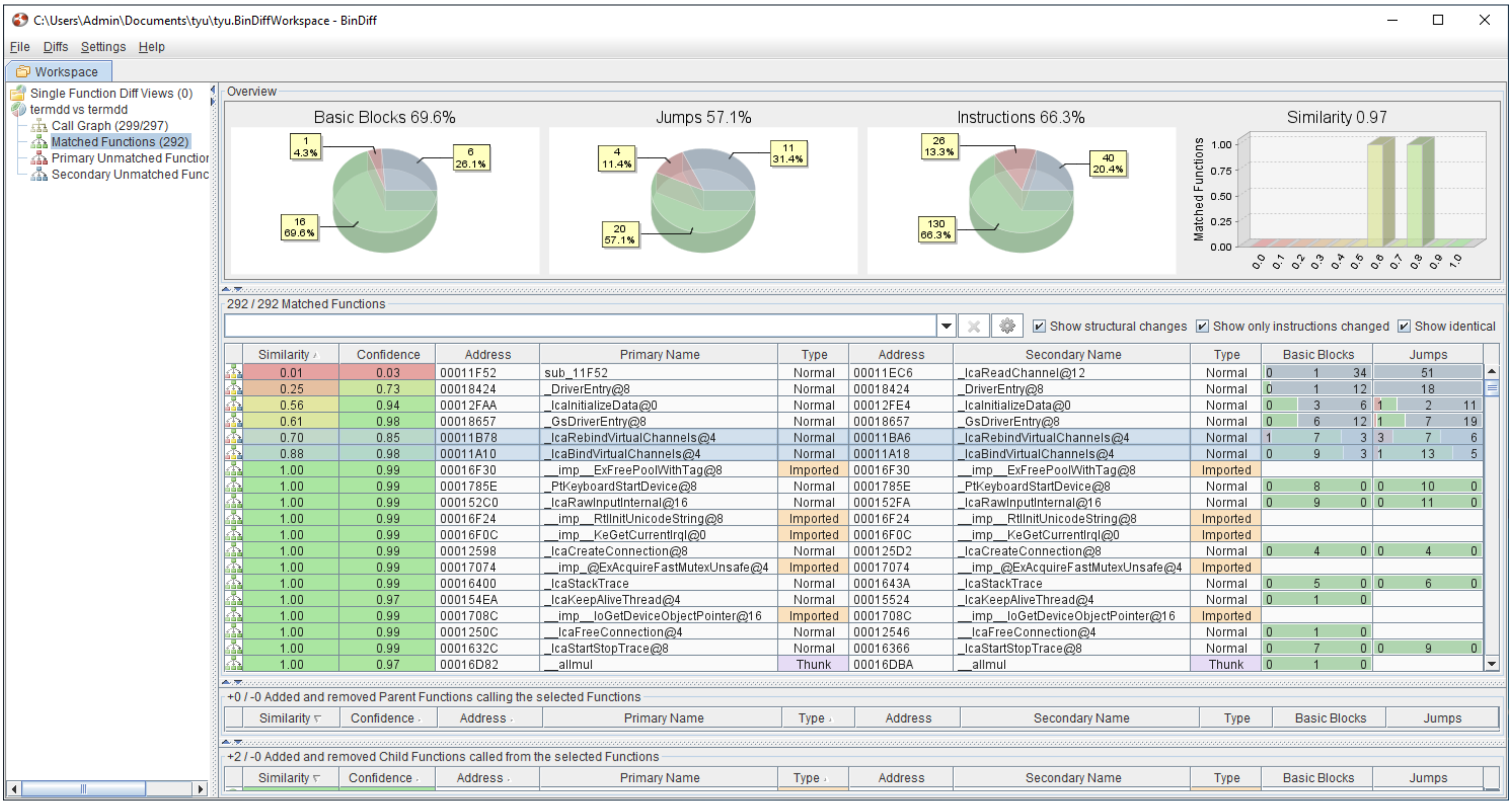
Analysis of CVE-2019-0708 (BlueKeep)

By : MalwareTech (https://www.malwaretech.com/author/malwaretech) May 31, 2019 Category : Vulnerability Research (https://www.malwaretech.com/category/vulnerability-research) Tags: kernel exploitation (https://www.malwaretech.com/tag/kernel-exploitation), reverse engineering (https://www.malwaretech.com/tag/reverse-engineering)

I held back this write-up until a proof of concept (PoC) was publicly available, as not to cause any harm. Now that there are multiple denial-of-service PoC on github, I'm posting my analysis.

Binary Diffing

As always, I started with a BinDiff of the binaries modified by the patch (in this case there is only one: TermDD.sys). Below we can see the results.



(https://www.malwaretech.com/wp-content/uploads/2019/05/BinDiff.png)

A BinDiff of TermDD.sys pre and post patch.

Most of the changes turned out to be pretty mundane, except for "_IcaBindVirtualChannels" and "_IcaRebindVirtualChannels". Both functions contained the same change, so I focused on the former as bind would likely occur before rebinding.

```
    v9 = IcaFindChannelByName(v4, (PERESOURCE)5, (char *)(v7 - 8));
    v10 = v9;
    if ( v9 )
    {
      IcaReferenceStack(v9);
      KeEnterCriticalRegion();
      ExAcquireResourceExclusiveLite((PERESOURCE)(v10 + 12), 1u);
      _IcaBindChannel(v10, 5, *(_WORD *)v7, *(_DWORD *)(v7 + 2));
      ExReleaseResourceLite((PERESOURCE)(v10 + 12));
      KeLeaveCriticalRegion();
      IcaDereferenceChannel((PVOID)v10);
      IcaDereferenceChannel((PVOID)v10);
      v4 = *(_DWORD *)(a1 - 468);
    }
    ++*(_DWORD *)(a1 - 456);
    v7 += 14;
  }
  while ( *(_DWORD *)(a1 - 456) < *(_DWORD *)(a1 - 464) );
}
```

```
    v3 = IcaFindChannelByName(v1, (PERESOURCE)5, (char *)v2 - 10);
    v4 = v3;
    if ( v3 )
    {
      IcaReferenceStack(v3);
      KeEnterCriticalRegion();
      ExAcquireResourceExclusiveLite((PERESOURCE)(v4 + 12), 1u);
      v5 = __stricmp((const char *)(v4 + 88), "MS_T120");
      v7 = *v2;
      if ( v5 )
        _IcaBindChannel(v4, 5, *((_WORD *)v2 - 1), v7);
      else
        _IcaBindChannel(v4, 5, 31, v7);
      ExReleaseResourceLite((PERESOURCE)(v4 + 12));
      KeLeaveCriticalRegion();
      IcaDereferenceChannel((PVOID)v4);
      IcaDereferenceChannel((PVOID)v4);
      v1 = v15;
    }
```

(https://www.malwaretech.com/wp-content/uploads/2019/05/IcaBindVirtualChannels.png)

Original IcaBindVirtualChannels is on the left, the patched version is on the right.

New logic has been added, changing how _IcaBindChannel is called. If the compared string is equal to "MS_T120", then parameter three of _IcaBindChannel is set to the 31.

Based on the fact the change only takes place if v4+88 is "MS_T120", we can assume that to trigger the bug this condition must be true. So, my first question is: what is "v4+88"?.

Looking at the logic inside IcaFindChannelByName, i quickly found my answer.

```
int __stdcall IcaFindChannelByName(int a1, PERESOURCE Resource, char *a3)
{
  int result; // eax@2
  int v4; // ebx@3
  _DWORD *v5; // esi@3
  int v6; // edi@4

  if ( Resource == (PERESOURCE)5 )
  {
    IcaLockChannelTable((PERESOURCE)(a1 + 272));
    v4 = a1 + 80;
    v5 = *(_DWORD **)(a1 + 80);
    if ( v5 == (_DWORD *)(a1 + 80) )
      goto LABEL_13;
    do
    {
      v6 = (int)(v5 - 25);
      if ( *(v5 - 4) == 5 && !__stricmp((const char *)(v6 + 88), a3) )
        break;
      v5 = (_DWORD *)*v5;
    }
    while ( v5 != (_DWORD *)v4 );
    if ( v5 != (_DWORD *)v4 )
      IcaReferenceStack(v6);
    else
LABEL_13:
      v6 = 0;
    IcaUnlockChannelTable((PERESOURCE)(a1 + 272));
    result = v6;
  }
  else
  {
    result = IcaFindChannel(a1, Resource, 0);
  }
  return result;
}
```

string comparison with same offset (88)

v6 is returned, so we can assume v6 is the channel pointer or handle

(https://www.malwaretech.com/wp-content/uploads/2019/05/IcaFindChannelByName.png)

content/uploads/2019/05/IcaFindChannelByName.png)

Inside of IcaFindChannelByName

Using advanced knowledge of the English language, we can decipher that IcaFindChannelByName finds a channel, by its name.

The function seems to iterate the channel table, looking for a specific channel. On line 17 there is a string comparison between a3 and v6+88, which returns v6 if both strings are equal. Therefore, we can assume a3 is the channel name to find, v6 is the channel structure, and v6+88 is the channel name within the channel structure.

Using all of the above, I came to the conclusion that "MS_T120" is the name of a channel. Next I needed to figure out how to call this function, and how to set the channel name to MS_T120.

I set a breakpoint on IcaBindVirtualChannels, right where IcaFindChannelByName is called. Afterwards, I connected to RDP with a legitimate RDP client. Each time the breakpoint triggered, I inspecting the channel name and call stack.

```
2: kd> u eip
termdd!IcaBindVirtualChannels+0xcb:
baa69adb e828f7ffff      call    termdd!IcaFindChannelByName (baa69208)
baa69ae0 8bf8            mov     edi,eax
baa69ae2 85ff            test    edi,edi
baa69ae4 7447            je      termdd!IcaBindVirtualChannels+0x11d (baa69b2d)
baa69ae6 57              push    edi
baa69ae7 e85a170000      call    termdd!IcaReferenceStack (baa6b246)
baa69aec ff1548efa6ba    call    dword ptr [termdd!_imp__KeEnterCriticalRegion (baa6ef48)]
baa69af2 6a01            push    1
2: kd> da poi(esp+8)
b1ce6508  "MS_T120"
2: kd> k
ChildEBP RetAddr
b1ce66cc baa6c867 termdd!IcaBindVirtualChannels+0xcb
b1ce6c14 baa6acfa termdd!IcaDeviceControlStack+0x1a5
b1ce6c28 baa6af8a termdd!IcaDeviceControl+0x26
b1ce6c40 804ef18f termdd!IcaDispatch+0x13a
b1ce6c50 8057f982 nt!IopfCallDriver+0x31
b1ce6c64 805807f7 nt!IopSynchronousServiceTail+0x70
b1ce6d00 80579274 nt!IopXxxControlFile+0x5c5
b1ce6d34 8054161c nt!NtDeviceIoControlFile+0x2a
b1ce6d34 7c90e4f4 nt!KiFastCallEntry+0xfc
009fe8d4 7c90d26c ntdll!KiFastSystemCallRet
009fe8d8 74f71173 ntdll!ZwDeviceIoControlFile+0xc
009fe914 74f7157e ICAAPI!IcaIoControl+0x29
009fe940 74f715c3 ICAAPI!_IcaStackIoControlWorker+0x64
009fe968 7246558e ICAAPI!IcaStackIoControl+0x29
009fe99c 7610ed48 rdpwsx!WsxIcaStackIoControl+0x2a
009fe9c8 74f7160d termsrv!WsxStackIoControl+0x43
009fe9f8 74f71806 ICAAPI!_IcaStackIoControl+0x33
009fefe0 74f71ec8 ICAAPI!_IcaStackWaitForIca+0x3e
009ff5e8 760fce31 ICAAPI!IcaStackConnectionAccept+0x153
009fff90 760fd5c0 termsrv!TransferConnectionToIdleWinStation+0x416
009fffb4 7c80b713 termsrv!WinStationTransferThread+0x69
009fffec 00000000 kernel32!BaseThreadStart+0x37
```

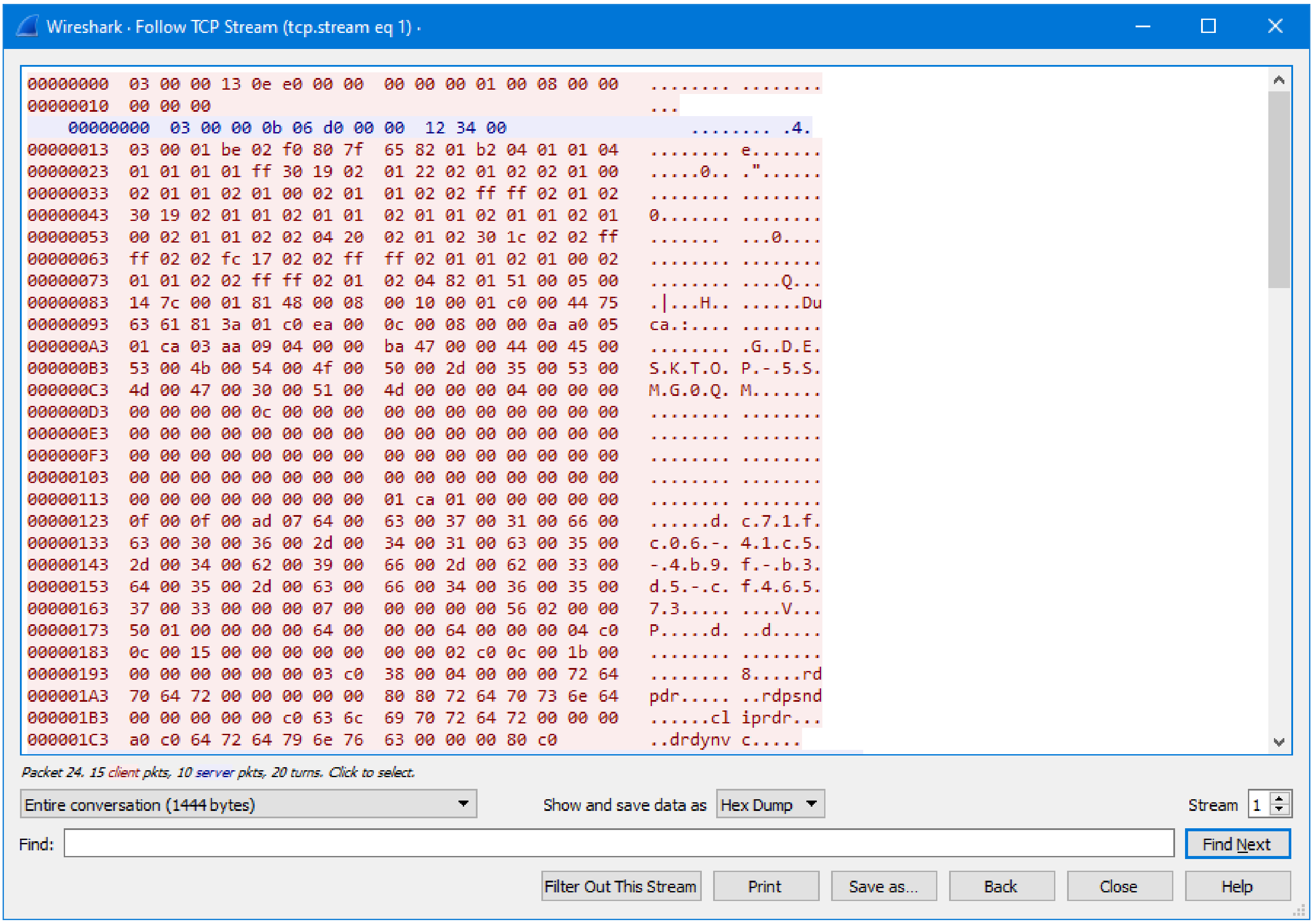
(https://www.malwaretech.com/wp-content/uploads/2019/05/Callstack1.png)

The callstack and channel name upon the first call to IcaBindVirtualChannels

The very first call to IcaBindVirtualChannels is for the channel i want, MS_T120. The subsequent channel names are "CTXTW ", "rdpdr", "rdpsnd", and "drdynvc".

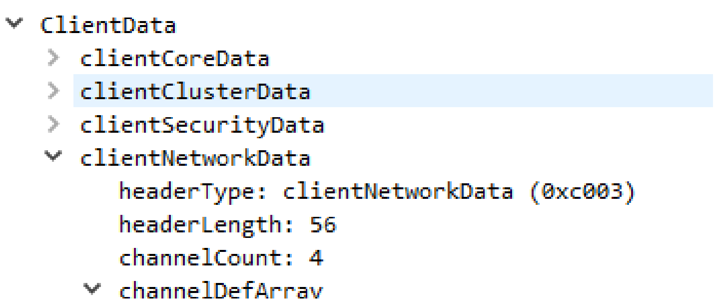
Unfortunately, the vulnerable code path is only reached if FindChannelByName succeeds (i.e. the channel already exists). In this case, the function fails and leads to the MS_T120 channel being created. To trigger the bug, i'd need to call IcaBindVirtualChannels a second time with MS_T120 as the channel name.

So my task now was to figure out how to call IcaBindVirtualChannels. In the call stack is IcaStackConnectionAccept, so the channel is likely created upon connect. Just need to find a way to open arbitrary channels post-connect… Maybe sniffing a legitimate RDP connection would provide some insight.



(https://www.malwaretech.com/wp-content/uploads/2019/05/WiresharkCapture.png)

A capture of the RDP connection sequence



```
channelDef
    name: rdpdr
  > options: 0x80800000
channelDef
    name: rdpsnd
  > options: 0xc0000000
channelDef
    name: cliprdr
  > options: 0xc0a00000
channelDef
    name: drdynvc
  > options: 0xc0800000
Frame (frame), 500 bytes
```

(https://www.malwaretech.com/wp-content/uploads/2019/05/ChannelArray.png)

The channel array, as seen by WireShark RDP parser

The second packet sent contains four of the six channel names I saw passed to IcaBindVirtualChannels (missing MS_T120 and CTXTW). The channels are opened in the order they appear in the packet, so I think this is just what I need.

Seeing as MS_T120 and CTXTW are not specified anywhere, but opened prior to the rest of the channels, I guess they must be opened automatically. Now, I wonder what happens if I implement the protocol, then add MS_T120 to the array of channels.

After moving my breakpoint to some code only hit if FindChannelByName succeeds, I ran my test.

```
2: kd> g
Breakpoint 1 hit
termdd!IcaBindVirtualChannels+0xd6:
baa69ae6 57              push    edi
```

(https://www.malwaretech.com/wp-content/uploads/2019/05/VulnerableCodePath.png)

Breakpoint is hit after adding MS_T120 to the channel array

Awesome! Now the vulnerable code path is hit, I just need to figure out what can be done…

To learn more about what the channel does, I decided to find what created it. I set a breakpoint on IcaCreateChannel, then started a new RDP connection.

```
Breakpoint 1 hit
termdd!IcaCreateChannel:
baa69d76 8bff            mov     edi,edi
3: kd> k
ChildEBP RetAddr
b1ab2a1c baa6ae21 termdd!IcaCreateChannel
b1ab2a44 baa6af4d termdd!IcaCreate+0xbd
b1ab2a5c 804ef18f termdd!IcaDispatch+0xfd
b1ab2a6c 805831fa nt!IopfCallDriver+0x31
b1ab2b4c 805bf444 nt!IopParseDevice+0xa12
b1ab2bc4 805bb9d0 nt!ObpLookupObjectName+0x53c
```

```
b1ab2c18 80576033 nt!ObOpenObjectByName+0xea
b1ab2c94 805769aa nt!IopCreateFile+0x407
b1ab2cf0 805790b4 nt!IoCreateFile+0x8e
b1ab2d30 8054161c nt!NtCreateFile+0x30
b1ab2d30 7c90e4f4 nt!KiFastCallEntry+0xfc
0252e838 7c90d09c ntdll!KiFastSystemCallRet
0252e83c 74f71207 ntdll!NtCreateFile+0xc
0252e898 74f7142b ICAAPI!_IcaOpen+0x59
0252e8b8 74f72184 ICAAPI!_IcaStackOpen+0x78
0252e8dc 7246684e ICAAPI!IcaChannelOpen+0x41
0252e90c 7246610d rdpwsx!MCSCreateDomain+0x84
0252e928 72463700 rdpwsx!GCCConferenceInit+0x24
0252e944 724640da rdpwsx!TSrvBindStack+0x19
0252e95c 72463c77 rdpwsx!TSrvAllocInfo+0x42
0252e978 724656e1 rdpwsx!TSrvStackConnect+0x26
0252e99c 7610ed48 rdpwsx!WsxIcaStackIoControl+0x17d
0252e9c8 74f7160d termsrv!WsxStackIoControl+0x43
0252e9f8 74f71806 ICAAPI!_IcaStackIoControl+0x33
0252efe0 74f71ec8 ICAAPI!_IcaStackWaitForIca+0x3e
0252f5e8 760fce31 ICAAPI!IcaStackConnectionAccept+0x153
0252ff90 760fd5c0 termsrv!TransferConnectionToIdleWinStation+0x416
0252ffb4 7c80b713 termsrv!WinStationTransferThread+0x69
0252ffec 00000000 kernel32!BaseThreadStart+0x37
```

(https://www.malwaretech.com/wp-content/uploads/2019/05/IcaCreateChannelCallstack.png)

The call stack when the IcaCreateChannel breakpoint is hit

Following the call stack downwards, we can see the transition from user to kernel mode at ntdll!NtCreateFile. Ntdll just provides a thunk for the kernel, so that's not of interest.

Below is the ICAAPI, which is the user mode counterpart of TermDD.sys. The call starts out in ICAAPI at IcaChannelOpen, so this is probably the user mode equivalent of IcaCreateChannel.

Due to the fact IcaOpenChannel is a generic function used for opening all channels, we'll go down another level to rdpwsx!MCSCreateDomain.

```
signed int __stdcall MCSCreateDomain(int a1, int a2, int a3, ULONG_PTR *a4)
{
  LPVOID v4; // eax@1
  ULONG_PTR v5; // esi@1
  HANDLE *v6; // ebx@3
  int v8; // [sp+Ch] [bp-Ch]@5
  int v9; // [sp+10h] [bp-8h]@5
  LPCRITICAL_SECTION lpCriticalSection; // [sp+14h] [bp-4h]@2

  *a4 = 0;
  v4 = HeapAlloc(g_hTShareHeap, 8u, 0x4A8u);
  v5 = (ULONG_PTR)v4;
  if ( !v4 )
    return 11;
  lpCriticalSection = (LPCRITICAL_SECTION)((char *)v4 + 4);
  if ( RtlInitializeCriticalSection((PRTL_CRITICAL_SECTION)((char *)v4 + 4)) )
  {
LABEL_8:
    HeapFree(g_hTShareHeap, 0, (LPVOID)v5);
    return 11;
  }
  EnterCriticalSection((LPCRITICAL_SECTION)(v5 + 4));
  *(_DWORD *)(v5 + 28) = a1;
  *(_DWORD *)(v5 + 32) = a2;
  *(_DWORD *)(v5 + 40) = a3;
  *(_DWORD *)(v5 + 48) = 0;
  *(_DWORD *)(v5 + 100) = 0;
  *(_DWORD *)(v5 + 96) = 0;
  *(_DWORD *)(v5 + 92) = 0;
```

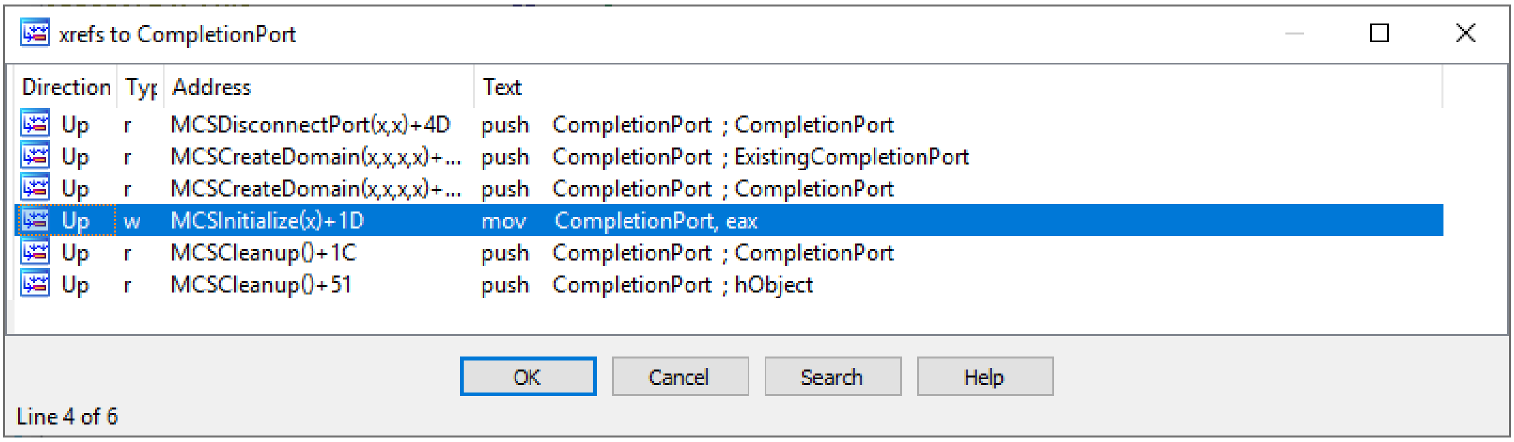
```
  *(_DWORD *)v5 = 0;
  MCSReferenceDomain((volatile LONG *)v5);
  v6 = (HANDLE *)(v5 + 36);
  if ( IcaChannelOpen(a1, 5, "MS_T120", v5 + 36) < 0 )
  {
LABEL_7:
    LeaveCriticalSection(lpCriticalSection);
    DeleteCriticalSection(lpCriticalSection);
    goto LABEL_8;
  }
  if ( !CreateIoCompletionPort(*v6, CompletionPort, v5, 0)
    || (v8 = 0, v9 = 20, IcaStackIoControl(a2, 3675139, &v8, 8, 0, 0, 0) < 0) )
  {
    IcaChannelClose(*v6);
    goto LABEL_7;
  }
  *a4 = v5;
  MCSReferenceDomain((volatile LONG *)v5);
  PostQueuedCompletionStatus(CompletionPort, 0xFFFFFFFD, v5, 0);
  LeaveCriticalSection(lpCriticalSection);
  return 0;
}
```

(https://www.malwaretech.com/wp-content/uploads/2019/05/MCSCreateDomain.png)

The code for rdpwsx!MCSCreateDomain

This function is really promising for a couple of reasons: Firstly, it calls IcaChannelOpen with the hard coded name “MS_T120”. Secondly, it creates an IoCompletionPort with the returned channel handle (Completion Ports are used for asynchronous I/O).

The variable named “CompletionPort” is the completion port handle. By looking at xrefs to the handle, we can probably find the function which handles I/O to the port.



(https://www.malwaretech.com/wp-content/uploads/2019/05/XrefsToCompletionPort.png)

All references to “CompletionPort”

Well, MCSInitialize is probably a good place to start. Initialization code is always a good place to start.

```
signed int __stdcall MCSInitialize(int a1)
{
  HANDLE v1; // eax@1
  HANDLE v2; // eax@2
  signed int result; // eax@3

  dword_72474194 = (int (__stdcall *)(_DWORD, _DWORD, _DWORD, _DWORD))a1;
  v1 = CreateIoCompletionPort((HANDLE)0xFFFFFFFF, 0, 0, 0);
  CompletionPort = v1;
  if ( v1 && (v2 = CreateThread(0, 0, IoThreadFunc, v1, 0, &ThreadId), (dword_7247418C = v2) != 0) )
  {
```

```
    SetThreadPriority(v2, 2);
    g_bInitialized = 1;
    result = 0;
  }
  else
  {
    result = 11;
  }
  return result;
}
```

(https://www.malwaretech.com/wp-content/uploads/2019/05/MCSInitialize.png)

The code contained within MCSInitialize

Ok, so a thread is created for the completion port, and the entrypoint is IoThreadFunc. Let's look there.

```
DWORD __stdcall IoThreadFunc(LPVOID lpThreadParameter)
{
  BOOL v1; // eax@1
  DWORD NumberOfBytesTransferred; // [sp+0h] [bp-Ch]@1
  LPOVERLAPPED Overlapped; // [sp+4h] [bp-8h]@1
  unsigned __int32 CompletionKey; // [sp+8h] [bp-4h]@1

  while ( 1 )
  {
    do
    {
      CompletionKey = 0;
      Overlapped = 0;
      v1 = GetQueuedCompletionStatus(
             lpThreadParameter,
             &NumberOfBytesTransferred,
             &CompletionKey,
             &Overlapped,
             0xFFFFFFFF);
    }
    while ( !v1 && !Overlapped );
    if ( CompletionKey == -1 )
      break;
    if ( v1 )
      MCSPortData((struc_1 *)CompletionKey, NumberOfBytesTransferred);
    else
      MCSDereferenceDomain((volatile LONG *)CompletionKey);
  }
  SetEvent(hObject);
  return 0;
}
```

(https://www.malwaretech.com/wp-content/uploads/2019/05/IoThreadFunc.png)

The completion port message handler

GetQueuedCompletionStatus is used to retrieve data sent to a completion port (i.e. the channel). If data is successfully received, it's passed to MCSPortData.

To confirm my understanding, I wrote a basic RDP client with the capability of sending data on RDP channels. I opened the MS_T120 channel, using the method previously explained. Once opened, I set a breakpoint on MCSPortData; then, I sent the string "MalwareTech" to the channel.

```
1: kd> g
Breakpoint 1 hit
rdpwsx!MCSPortData+0x26:
```

```
001b:724666e0 8b4704          mov     eax,dword ptr [edi+4]
1: kd> db edi
00f124ec  4d 61 6c 77 61 72 65 54-65 63 68 00 01 00 00 00  MalwareTech.....
1: kd>
```

(https://www.malwaretech.com/wp-content/uploads/2019/05/MCSPortDataBreakpoint.png)

Breakpoint hit on MCSPortData once data is sent the the channel.

So that confirms it, I can read/write to the MS_T120 channel.

Now, let's look at what MCSPortData does with the channel data…

```
LONG __stdcall MCSPortData(int channel_ptr, int bytes_transfered)
{
  int v2; // eax@3
  void *v3; // eax@10

  EnterCriticalSection((LPCRITICAL_SECTION)(channel_ptr + 4));
  if ( (unsigned int)bytes_transfered >= 0xFFFFFFF0 )
  {
    if ( bytes_transfered == 0xFFFFFFFE )
      MCSChannelClose(channel_ptr);
  }
  else if ( !(*(_BYTE *)(channel_ptr + 0x2C) & 1) )
  {
    v2 = *(_DWORD *)(channel_ptr + 120);
    if ( v2 )
    {
      if ( v2 == 2 )
      {
        HandleDisconnectProviderIndication(channel_ptr, bytes_transfered, channel_ptr + 116);
        MCSChannelClose(channel_ptr);
      }
    }
    else
    {
      HandleConnectProviderIndication(channel_ptr, bytes_transfered, channel_ptr + 116);
    }
    *(_DWORD *)(channel_ptr + 120) = -1;
  }
  v3 = *(void **)(channel_ptr + 36);
  if ( v3
    && (ReadFile(v3, (LPVOID)(channel_ptr + 116), 0x434u, 0, (LPOVERLAPPED)(channel_ptr + 84)) || GetLastError() == 997)
  {
    MCSReferenceDomain((volatile LONG *)channel_ptr);
  }
  LeaveCriticalSection((LPCRITICAL_SECTION)(channel_ptr + 4));
  return MCSDereferenceDomain((volatile LONG *)channel_ptr);
}
```

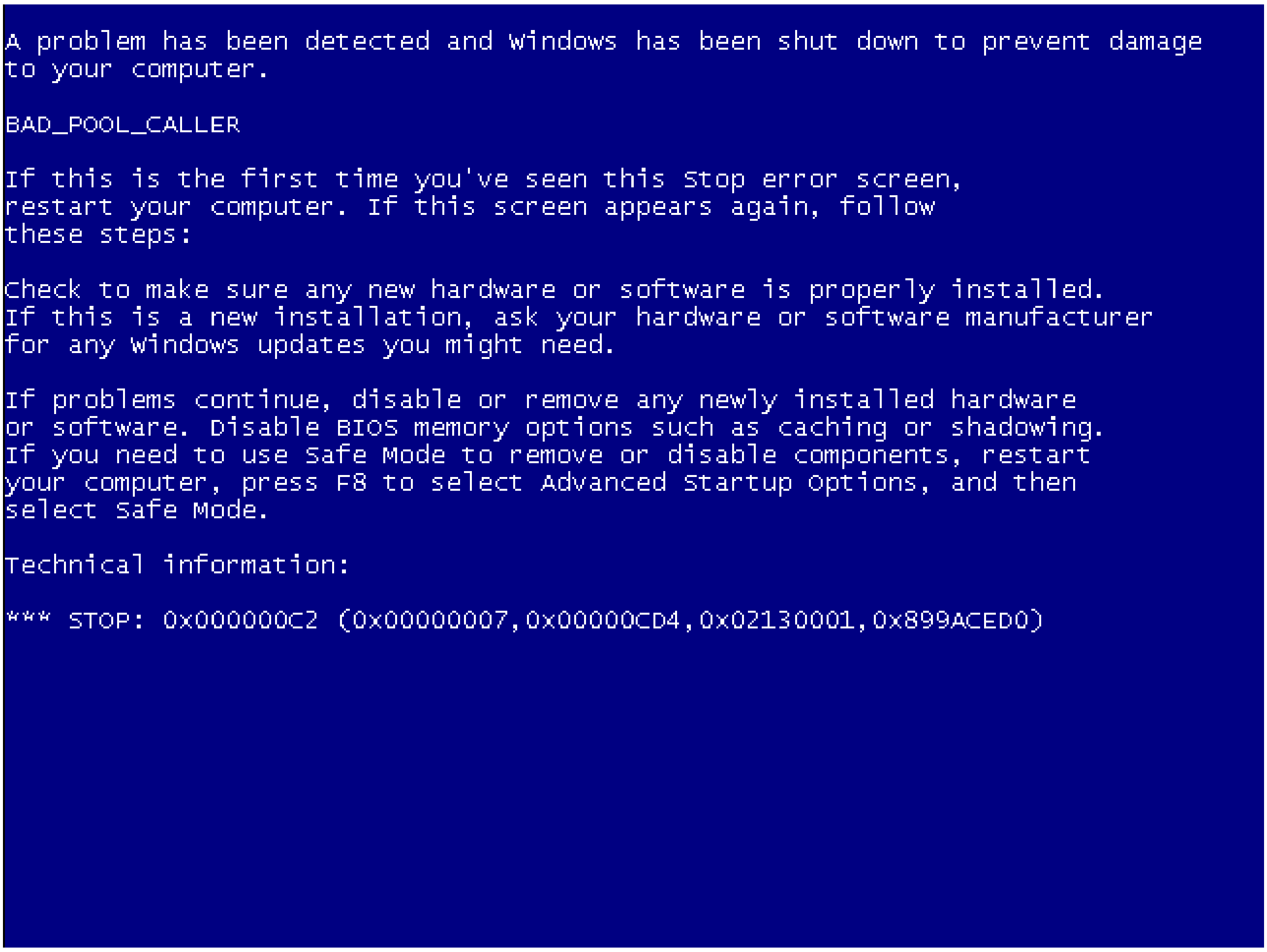
MCSPortData buffer handling code

ReadFile tells us the data buffer starts at channel_ptr+116. Near the top of the function is a check performed on chanel_ptr+120 (offset 4 into the data buffer). If the dword is set to 2, then the function calls HandleDisconnectProviderIndication and MCSCloseChannel.

Well, that's interesting. The code looks like some kind of handler to deal with channel connects/disconnect events. After looking into what would normally trigger this function, I realized MS_T120 is an internal channel and not normally exposed externally.

I don't think we're supposed to be here…

Being a little curious, i sent the data required to trigger the call to MCSChannelClose. Surely prematurely closing an internal channel couldn't lead to any issues, could it?



Oh, no. We crashed the kernel!

Whoops! Let's take a look at the bugcheck to get a better idea of what happened.

```
1: kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

BAD_POOL_CALLER (c2)
The current thread is making a bad pool request.  Typically this is at a bad IRQL level or double freeing the same allocation, etc.
Arguments:
Arg1: 00000007, Attempt to free pool which was already freed
Arg2: 00000cd4, (reserved)
Arg3: 02130001, Memory contents of the pool block
Arg4: 893ab110, Address of the block of pool being deallocated

Debugging Details:
------------------


POOL_ADDRESS:  893ab110 Nonpaged pool

FREED_POOL_TAG:  Ica

BUGCHECK_STR:  0xc2_7_Ica

DEFAULT_BUCKET_ID:  DRIVER_FAULT

PROCESS_NAME:  svchost.exe

ANALYSIS_VERSION: 6.3.9600.17336 (debuggers(dbg).150226-1500) amd64fre

LAST_CONTROL_TRANSFER:  from 804f8df9 to 8052b5dc

STACK_TEXT:
b18f2f9c 804f8df9 00000003 b18f32f8 00000000 nt!RtlpBreakWithStatusInstruction
b18f2fe8 804f99e4 00000003 893ab118 893ab108 nt!KiBugCheckDebugBreak+0x19
b18f33c8 804f9f33 000000c2 00000007 00000cd4 nt!KeBugCheck2+0x574
b18f33e8 8054b583 000000c2 00000007 00000cd4 nt!KeBugCheckEx+0x1b
b18f3438 baa88f82 893ab110 00000000 893ab110 nt!ExFreePoolWithTag+0x2a3
b18f344c baa892c7 893ab110 893ab110 893bf9e8 termdd!_IcaFreeChannel+0x3c
```

```
b18f3468 baa899f0 89c7ed28 893bf9e8 00000000 termdd!IcaDereferenceChannel+0x41
b18f34a8 baa8a46b 893bf9e8 00000005 0000001f termdd!IcaChannelInputInternal+0x380
b18f34d0 b159fa16 899a9cf4 00000005 0000001f termdd!IcaChannelInput+0x41
b18f3508 b159fa82 e13ad540 893bf9e8 899a9ce0 RDPWD!SignalBrokenConnection+0x40
b18f3520 baa8a48f e12f6008 00000004 00000000 RDPWD!MCSIcaChannelInput+0x58
b18f3548 babfa2f7 899e715c 00000004 00000000 termdd!IcaChannelInput+0x65
b18f3d90 baa8c22f 003b4ca0 00000000 893b2ac0 TDTCP!TdInputThread+0x481
b18f3dac 805cff64 893b2d20 00000000 00000000 termdd!_IcaDriverThread+0x51
b18f3ddc 805460de baa8c1de 89bd0290 00000000 nt!PspSystemThreadStartup+0x34
00000000 00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x16
```

It seems that when my client disconnected, the system tried to close the MS_T120 channel, which I'd already closed (leading to a double free).

Due to some mitigations added in Windows Vista, double-free vulnerabilities are often difficult to exploit. However, there is something better.

```c
int __stdcall SignalBrokenConnection(int a1)
{
  int v1; // ST00_4@3
  int result; // eax@3
  int v3; // [sp+4h] [bp-10h]@3
  int v4; // [sp+8h] [bp-Ch]@3
  int v5; // [sp+Ch] [bp-8h]@3
  int v6; // [sp+10h] [bp-4h]@3

  if ( *(_DWORD *)(a1 + 200) == 4 )
  {
    if ( *(_BYTE *)(a1 + 16) & 1 )
    {
      v1 = *(_DWORD *)a1;
      v4 = 2;
      v3 = 0;
      v5 = 0;
      v6 = 0;
      IcaChannelInput(v1, 5, 31, 0, (int)&v3, 16);
      result = DisconnectProvider(a1, 0, 0);
    }
  }
  return result;
}
```

Internals of the channel cleanup code run when the connection is broken

Internally, the system creates the MS_T120 channel and binds it with ID 31. However, when it is bound using the vulnerable IcaBindVirtualChannels code, it is bound with another id.

external channels are bound with a system generated id

```c
      v9 = IcaFindChannelByName(v4, (PERESOURCE)5, (char *)(v7 - 8));
      v10 = v9;
      if ( v9 )
      {
        IcaReferenceStack(v9);
        KeEnterCriticalRegion();
        ExAcquireResourceExclusiveLite((PERESOURCE)(v10 + 12), 1u);
        _IcaBindChannel(v10, 5, *(_WORD *)v7, *(_DWORD *)(v7 + 2));
        ExReleaseResourceLite((PERESOURCE)(v10 + 12));
        KeLeaveCriticalRegion();
        IcaDereferenceChannel((PVOID)v10);
        IcaDereferenceChannel((PVOID)v10);
        v4 = *(_DWORD *)(a1 - 468);
      }
      ++*(_DWORD *)(a1 - 456);
      v7 += 14;
    }
    while ( *(_DWORD *)(a1 - 456) < *(_DWORD *)(a1 - 464) );
  }
```

patch hardcodes channel id to 31 for MS_T120

```c
      v3 = IcaFindChannelByName(v1, (PERESOURCE)5, (char *)v2 - 10);
      v4 = v3;
      if ( v3 )
      {
        IcaReferenceStack(v3);
        KeEnterCriticalRegion();
        ExAcquireResourceExclusiveLite((PERESOURCE)(v4 + 12), 1u);
        v5 = __stricmp((const char *)(v4 + 88), "MS_T120");
        v7 = *v2;
        if ( v5 )
          _IcaBindChannel(v4, 5, *((_WORD *)v2 - 1), v7);
        else
          _IcaBindChannel(v4, 5, 31, v7);
        ExReleaseResourceLite((PERESOURCE)(v4 + 12));
        KeLeaveCriticalRegion();
        IcaDereferenceChannel((PVOID)v4);
        IcaDereferenceChannel((PVOID)v4);
        v1 = v15;
      }
```

The difference in code pre and post patch

Essentially, the MS_T120 channel gets bound twice (once internally, then once by us). Due to the fact the channel is bound under two different ids, we get two separate references to it.

When one references is used to close the channel, the reference is deleted, as is the channel; however, the other reference remains (known as a use-after-free). With the remaining reference, it is now possible to write kernel memory which no longer belongs to us.