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.

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.
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; // eax82  
    int v4; // edx83  
    _DWORD *v5; // esi83  
    int v6; // edi84  

    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 );  
        IcaReferenceStack(v6);  
        else  
            goto LABEL_13;  
    }  
    v6 - 0;  
    IcaUnlockChannelTable((PERESOURCE)(a1 + 272));  
    result - v6;  
}
```

String comparison with same offset (88)

v6 is returned, so we can assume v6 is the channel pointer or handle
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 inspected the channel name and call stack.

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.

A capture of the RDP connection sequence

![A capture of the RDP connection sequence](https://www.malwaretech.com/wp-content/uploads/2019/05/WiresharkCapture.png)
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.

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.
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 rdpswx!MCSCreateDomain.
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.

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

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

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

    while ( v1 )
    {
        CompletionKey = 0;
        Overlapped = 0;
        v1 = GetQueuedCompletionStatus(
            lpThreadParameter,
            &NumberOfBytesTransferred,
            &CompletionKey,
            &Overlapped,
            0xFFFFFFF);

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

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: kc> g
Breakpoint 1 hit. MCSPortData+0x26:
```
Breakpoint hit on MCSPortData once data is sent 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…

```c
1: #define __stdcall MCSPortData(int channel_ptr, int bytes_transferred)
2: 
3:     int v2; // eax8
4:     void *v3; // eax10
5: 
6:     EnterCriticalSection((LPIT:RITICAL_SECTION)(channel_ptr + 4));
7:     if ((unsigned int)bytes_transferred >= 0xFFFFFFFF) {
8:         MCSCloseChannel(channel_ptr);
9:     } else if ( (!(* (channel_ptr + 0x20)) & 1) ) {  
10:         if ( v2 == 0) {
11:             HANDLE_DISCONNECT_PROVIDER_INDICATION(channel_ptr, bytes_transferred, channel_ptr + 116);
12:             MCSCloseChannel(channel_ptr);
13:         } else if ( v2 == 2) {
14:             HANDLE_CONNECT_PROVIDER_INDICATION(channel_ptr, bytes_transferred, channel_ptr + 116);
15:         } else {
16:             v3 = (void *)(channel_ptr + 36);
17:             if ( v3 && (ReadFile(v3, (LPVOID)(channel_ptr + 116), 0x454u, 0, (LP overlapped)(channel_ptr + 8)) || GetLastError() == 997) ) {
18:                 MCSReferenceDomain((volatile long)channel_ptr);
19:             }
20:         }
21:     }
22: 
23:     LeaveCriticalSection((LPIT:RITICAL_SECTION)(channel_ptr + 4));
24:     return MCSReferenceDomain((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 channel_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 MCSCloseChannel. Surely
A problem has been detected and Windows has been shut down to prevent damage to your computer.

BAD_POOL_CALLER

If this is the first time you’ve seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing.

If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup options, and then select Safe Mode.

Technical information:

*** STOP: 0x000000c2 (0x00000007,0x00000cd4,0x02130001,0x899aced0)

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

Bugscheck Analysis

BAD_POOL_CALLER (c2)
The current thread is making a bad pool request. Typically this is at a bad ISR level or double freeing the same allocation, etc.

Argument
Arg1 00000007 [Attempt to free pool which was already freed]
Arg2 000000c4 [reserved]
Arg3 001300d1 Memory contents of the pool block
Arg4 093ab1b0 Address of the block of pool being deallocated

Debugging Details:

POOL_ADDRESS: 093ab1b0 Nonpaged pool
FREE_POOL_TAG: Ica
BUGCHECK_STR: 0x02_2_Ica
DEFAULT_BUCKET_ID: DRIVER_FAULT
PROCESS_NAME: svchost.exe
ANALYSIS_VERSION: 6.2.9600.17336 (debugger(dbg) 150226-1500) msv4fre
LAST_CONTROL_TRANSFER: from 804f86d9 to 8052b5dc

STACK_TEXT:

b1612b69 804f86d9 00000003 b1612b78 00000100 ntRtlpBreakWithStatusInstruction
b1612b78 804f86e4 00000003 093ab1b0 093ab1b0 ntRtlBugCheckEmebgBreak+0x19
b1613c08 804f86e1 000000e1 00000007 00000cd4 ntRltBugCheckEmebgBreak+0x74
b1613c08 804f86e3 000000e3 00000007 00000cd4 ntRltBugCheckEmebgBreak+0x1b
b1613c08 b0a2f410 093ab1b0 00000000 093ab1b0 ntRtlPBreakWithStatusInstruction

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.

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.

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.