

# He Said, She Said - Poisoned RDP

## Offense and Defense

Black Hat USA 2019

[Eyal Itkin](#), Vulnerability Researcher, Check Point Research

[Dana Baril](#), Security Software Engineer, Microsoft Defender ATP

### Overview

Used by thousands of IT professionals and security researchers worldwide, the Remote Desktop Protocol (RDP) is usually considered a safe and trustworthy application to connect to remote computers. Whether it is used to help those working remotely or to work in a safe VM environment, RDP clients are an invaluable tool.

However, our research discovered multiple critical vulnerabilities in the commonly used Remote Desktop Protocol (RDP) that allow a malicious actor to reverse the usual direction of communication and infect the IT professional or security researcher's computer.

In this research we present the vulnerabilities that we found in popular RDP clients (focusing on `mstsc.exe`). From the defensive perspective, we introduce behavioral detections, leveraging basic telemetry, such as Windows built-in traces.

In addition, we discuss Microsoft's fix to the vulnerability in `mstsc.exe`, which has been made available for Windows 10 in July 2019.

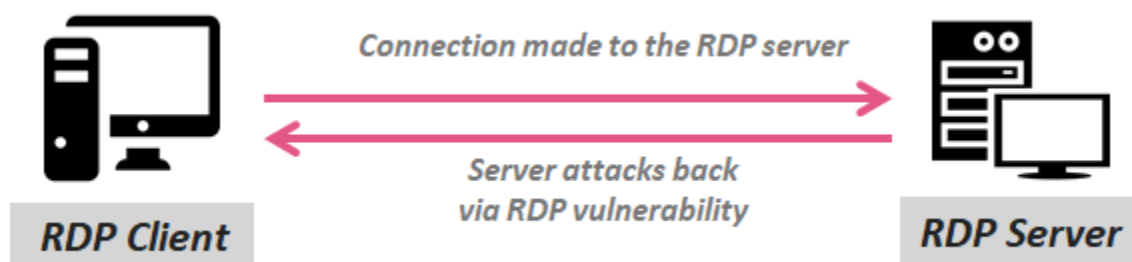
# Offense

## Introduction

The Remote Desktop Protocol (RDP), also known as `mstsc` (named after Microsoft's built-in RDP client), is a proprietary protocol developed by Microsoft that is commonly used by technical users and IT staff to connect to / work on a remote computer. There are also some popular open-source clients for the RDP protocol that are used mainly by Linux and Mac users.

RDP offers multiple complex features, such as compressed video streaming, clipboard sharing and several encryption layers. In this research we focused on finding vulnerabilities in the protocol and its popular implementations.

A common scenario is connecting an RDP client to a remote RDP server, that is installed on the remote computer. After a successful connection, the client gains access to the remote server and can control it, depending on the user's permissions. However, what happens if this can be accomplished in the opposite direction. We investigated this scenario, where a remote server can attack and gain control over a client machine.



**Figure 1:** Attack scenario for the RDP protocol.

There are several common scenarios in which an attacker can gain elevated network permissions by deploying such an attack, thus advancing his lateral movement inside an organization:

1. Attacking an IT member that connects to an infected workstation inside the corporate network, thus gaining higher permission levels and greater access to the network systems.
2. Attacking a malware researcher that connects to a remote sandboxed virtual machine that contains a tested malware. This allows the malware to escape the sandbox and infiltrate the corporate network.

Now that we decided on our attack vector, it is time to introduce our targets, the most commonly used RDP clients:

- `mstsc.exe` – Microsoft’s built-in RDP client application.
- [FreeRDP](#) – The most popular and mature open-source RDP client on Github.
- [rdesktop](#) – An older open-source RDP client, which is installed by default in Kali-linux distros.

**Fun fact:** Since `rdesktop` is the built-in client in Kali-linux (which broadly used by red teams for penetration testing) we came up with another attack scenario; Blue teams can install organizational honeypots and attack red teams that try to connect to them through the RDP protocol.

## Open-Source RDP clients

As a best practice, we investigated open source tools first, in order to get familiar with the protocol. In addition, the assumption was that common vulnerabilities from the two open source applications, could potentially apply to Microsoft’s client as well. In a recon check, `rdesktop` seems smaller than `FreeRDP` (has fewer lines of code) and therefore `rdesktop` was selected it as our first target.

**Note:** We decided to perform an old-fashion manual code audit instead of using any fuzzing technique. The motivation was avoiding the overhead of writing a dedicated fuzzer for the complex RDP protocol, along with the fact that using AFL for a protocol with several compression and encryption layers didn’t look like a good idea.

## rdesktop

**Tested version:** v1.8.3

The decision to manually search for vulnerabilities paid off quickly. We soon found several vulnerable patterns in the code, which enabled a good understanding of the code, and pinpointed possible vulnerabilities.

We found 11 vulnerabilities with a major security impact out of 19 vulnerabilities that were found overall in the library. For the full list of CVEs for `rdesktop`, see Appendix A. Instead of a technical analysis of all the CVEs, we will focus on two common vulnerable code patterns that we found.

**Note:** An additional recon showed that the [xrdp](#) open-source RDP server is based on the code of `rdesktop`. Based on our findings, it appears that similar vulnerabilities can be found in `xrdp` as well.

### Remote Code Executions – CVEs 2018-20179 – 2018-20181

Throughout the code of the client, there is an assumption that the server sent enough bytes to the client to process. One example for this assumption can be found in the following code snippet in Figure 2:

```
void
channel_process(STREAM s, uint16 mcs_channel)
{
    uint32 length, flags;
    uint32 thislength;
    VCHANNEL *channel = NULL;
    unsigned int i;
    STREAM in;

    ...
    in_uint32_le(s, length);
    in_uint32_le(s, flags);
    if ((flags & CHANNEL_FLAG_FIRST) && (flags & CHANNEL_FLAG_LAST))
    {
        /* single fragment - pass straight up */
        channel->process(s);
    }
    ...
}
```

**Figure 2:** Parsing 2 fields from stream `s` without first checking its size.

As we can see, the fields `length` and `flags` are parsed from the stream `s`, without checking that `s` indeed contains the required 8 bytes for this parsing operation. While this usually only leads to an Out-Of-Bounds read, we can combine this vulnerability with an additional vulnerability in several of the inner channels and achieve a much more severe effect.

There are three logical channels that share a common vulnerability:

- `lspci`
- `rdpsnddbg` – yes, this debug channel is always active
- `seamless`

```
/* Process new data from the virtual channel */
static void
lspci_process(STREAM s)
{
    unsigned int pkglen;
    static char *rest = NULL;
    char *buf;

    pkglen = s->end - s->p;
    /* str_handle_lines requires null terminated strings */
    // EI-DBG: In case we over-read here, we can do the following:|
    // EI-DBG: 1. buf = xmalloc(0)
    // EI-DBG: 2. STRNCPY(buf, s->p, 0)
    // EI-DBG: a) strncpy(buf, s->p, -1)
    // EI-DBG: b) buf[-1] = '\0'
    // EI-DBG: And s->p can be controlled using data from the previous packet
    buf = xmalloc(pkglen + 1);
    STRNCPY(buf, (char *) s->p, pkglen + 1);
    ...

    str_handle_lines(buf, &rest, lspci_process_line, NULL);
    xfree(buf);
}
```

**Figure 3:** Integer-Underflow when calculating the remaining `pkglen`.

By reading too much data from the stream, i.e. sending a chopped packet to the client, the invariant `s->p <= s->end` breaks. This leads to an Integer-Underflow when calculating `pkglen`, and to an additional Integer-Overflow when allocating `xmalloc(pkglen + 1)` bytes for our buffer, as can be seen in my comment above the call to `xmalloc`.

Together with the proprietary implementation of `STRNCPY`, seen in Figure 4, we can trigger a massive heap-based buffer overflow when copying data to the tiny allocated heap buffer.

```
#define STRNCPY(dst,src,n) { strncpy(dst,src,n-1); dst[n-1] = 0; }
```

**Figure 4:** proprietary implementation of the `strncpy` function.

By chaining together these two vulnerabilities, found in three different logical channels, we now have three remote code execution vulnerabilities.

## CVE 2018-8795 – Remote Code Execution

Another classic vulnerability is an Integer-Overflow when processing the received bitmap (screen content) updates, as can be seen in Figure 5:

```

/* Process bitmap updates */
void
process_bitmap_updates(STREAM s)
{
    uint16 num_updates;
    uint16 left, top, right, bottom, width, height;
    uint16 cx, cy, bpp, Bpp, compress, bufsize, size;
    uint8 *data, *bmpdata;
    int i;

    in_uint16_le(s, num_updates);

    for (i = 0; i < num_updates; i++)
    {
        in_uint16_le(s, left);
        in_uint16_le(s, top);
        in_uint16_le(s, right);
        in_uint16_le(s, bottom);
        // EI-DBG: Here we control width (16bit), height (16bit), and bpp (13bit)
        in_uint16_le(s, width);
        in_uint16_le(s, height);
        in_uint16_le(s, bpp);
        Bpp = (bpp + 7) / 8;
        ...
        in_uint8p(s, data, size);
        // EI-DBG: A nice Integer-Overflow: width * height * Bpp > 4GB
        // EI-DBG: Since the decompression methods stop on illegal opcode,
        // EI-DBG: this is a controllable heap-based Buffer-Overflow
        bmpdata = (uint8 *) xmalloc(width * height * Bpp);
        if (bitmap_decompress(bmpdata, width, height, data, size, Bpp))
        {
            ui_paint_bitmap(left, top, cx, cy, width, height, bmpdata);
        }
        else
        {
            DEBUG_RDP5(("Failed to decompress data\n"));
        }
    }
}

```

Figure 5: Integer-Overflow when processing bitmap updates.

Although `width` and `height` are only 16 bits each, by multiplying them together with `Bpp` (bits-per-pixel), we can trigger an Integer-Overflow. Later on, the bitmap decompression will process our input and break on any decompression error, giving us a controllable heap-based buffer-overflow.

**Note:** This tricky calculation can be found in several places throughout the code of `rdesktop`, so we marked it as a potential vulnerability to check for in `FreeRDP`.

# FreeRDP

**Tested version:** 2.0.0-rc3

After finding multiple vulnerabilities in `rdesktop`, we approached `FreeRDP` with some trepidation; perhaps only `rdesktop` had vulnerabilities when implementing RDP? We still can't be sure that every implementation of the protocol will be vulnerable.

And indeed, at first glance, the code seemed much better: there are minimal size checks before parsing data from the received packet, and the code "feels" more mature. It is going to be a challenge. However, after a deeper examination, we started to find cracks in the code, and eventually discovered critical vulnerabilities in this client as well. We found 5 vulnerabilities with major security impact, and 6 vulnerabilities overall in the library. For the full list of CVEs for `FreeRDP`, see Appendix B.

During our research, we developed a PoC exploit for CVE 2018-8786, as can be seen in this video:

<https://youtu.be/eogkRQtcM6U>

**Note:** An additional recon showed that the RDP client `NeutrinoRDP` is a fork of an older version (1.0.1) of `FreeRDP` and therefore probably suffers from the same vulnerabilities.

## CVE 2018-8787 – Same Integer-Overflow

As we saw earlier in `rdesktop`, calculating the dimensions of a received bitmap update is susceptible to Integer-Overflows. As expected, `FreeRDP` shares the same vulnerability:



```

static BOOL gdi_Bitmap-Decompress(rdpContext* context, rdpBitmap* bitmap,
                                const BYTE* pSrcData, UINT32 DstWidth, UINT32 DstHeight,
                                UINT32 bpp, UINT32 length, BOOL compressed,
                                UINT32 codecId)
{
    UINT32 SrcSize = length;
    UINT32 SrcFormat;
    rdpGdi* gdi = context->gdi;
    bitmap->compressed = FALSE;
    bitmap->format = gdi->dstFormat;
    // EI-DBG: Same IOF as in rdesktop:
    // EI-DBG: DstWidth - controlled 16 bits
    // EI-DBG: DstHeight - controlled 16 bits
    // EI-DBG: 1 <= bpp <= 0x20
    // EI-DBG: without decompression, we can simply have a heap buffer-overflow
    bitmap->length = DstWidth * DstHeight * GetBytesPerPixel(bitmap->format);
    bitmap->data = (BYTE*) _aligned_malloc(bitmap->length, 16);
}

```

Figure 6: Same Integer-Overflow when processing bitmap updates.

## Remote Code Execution – CVE 2018-8786

```

BITMAP_UPDATE* update_read_bitmap_update(rdpUpdate* update, wStream* s)
{
    UINT32 i;
    BITMAP_UPDATE* bitmapUpdate = calloc(1, sizeof(BITMAP_UPDATE));

    if (!bitmapUpdate)
        goto fail;

    if (Stream_GetRemainingLength(s) < 2)
        goto fail;

    Stream_Read_UINT16(s, bitmapUpdate->number); /* numberRectangles (2 bytes) */
    WLog_Print(update->log, WLOG_TRACE, "BitmapUpdate: %"PRIu32"", bitmapUpdate->number);

    if (bitmapUpdate->number > bitmapUpdate->count)
    {
        UINT16 count;
        BITMAP_DATA* newdata;
        // EI-DBG: Taking a 16 bit value, multiplying by 2, and storing it back in a 16 bit (?) variable
        // EI-DBG: count < number ==> (partially) controlled heap based buffer overflow
        count = bitmapUpdate->number * 2;
        newdata = (BITMAP_DATA*) realloc(bitmapUpdate->rectangles,
                                        sizeof(BITMAP_DATA) * count);

        if (!newdata)
            goto fail;
    }
}

```

Figure 7: Integer-Truncation when processing bitmap updates.

As can be seen in Figure 7, there is an Integer-Truncation when trying to calculate the required capacity for the bitmap updates array. Later, rectangle structs will be parsed from our packet and into the memory of the too-small allocated buffer.

The vulnerable code snippet is followed by a controlled amount (`bitmapUpdate->number`) of heap allocations (with a controlled size) when the rectangles are parsed and stored to the array, granting the attacker a great heap-shaping primitive. The downside of this vulnerability is that most of the rectangle fields are only 16 bits wide and are upcasted to 32 bits to be stored in the array. Despite that, we managed to exploit this CVE in our PoC. Even this partially controlled heap-based buffer-overflow is enough for remote code execution.

## Mstsc.exe – Microsoft’s RDP client

**Tested version:** Build 18252.rs\_prerelease.180928-1410

After we finished checking the open source implementations, we felt that we had a pretty good understanding of the protocol and can move forward to reverse engineer Microsoft’s RDP client. As a first step, we need to find which binaries contain the logic we want to examine. The `*.dll` files and `*.exe` files we chose to focus on:

- `rdpbase.dll` – Protocol layer for the RDP client.
- `rdpserverbase.dll` – Protocol layer for the RDP server.
- `rdpcore.dll` / `rdpcorets.dll` – Core logic for the RDP engine.
- `rdpclip.exe` – An `.exe` we found and that we will introduce later on.
- `mstscax.dll` – Mostly the same RDP logic, used by `mstsc.exe`.

## Testing prior vulnerabilities

We started by testing our PoCs for the vulnerabilities in the open-source clients. Unfortunately, all of them caused the client to close the connection, without any crash. Running out of excuses, we opened IDA and started to track the flow of the messages. Soon enough, we realized that Microsoft’s implementation is much better than the implementations we tested previously. Actually, it seems like Microsoft’s code is better by several orders of magnitude, as it contains:

- Several optimization layers for efficient network streaming of the received video.
- Robust input checks.

- Robust decompression checks, to guarantee that no byte will be written past the destination buffer.
- Additional supported clipboard features.
- ...

Needless to say, the code includes checks for Integer-Overflows when processing bitmap updates.

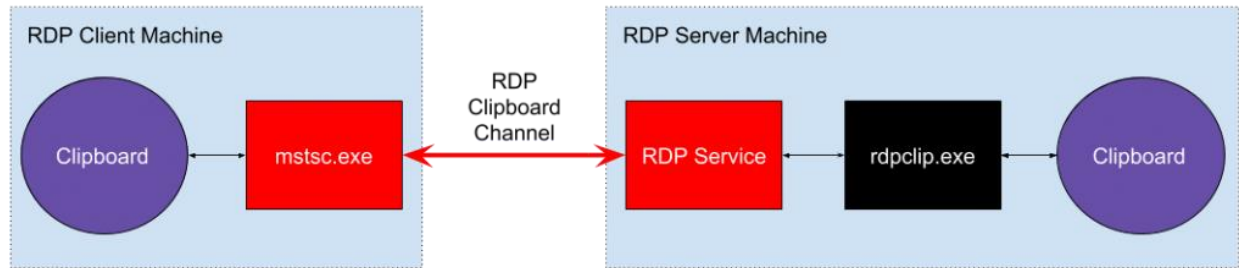
## Wait a minute, they share a clipboard?

When we checked `rdesktop` and `FreeRDP`, we found several vulnerabilities in the clipboard sharing channel (every logical data layer is called a channel). However, at the time, we didn't pay much attention to it because they shared only two formats: raw text and Unicode text. In this case, Microsoft supports several more data formats, as the switch table we saw in the code was much bigger than before.

After reading more about the different formats in [MSDN](#), one format immediately attracted our attention: `CF_HDROP`. This format is responsible for "Drag & Drop" (hence the name `HDROP`), which is in our case, the "Copy & Paste" feature. It's possible to simply copy a group of files from one computer and paste them in another computer. For example, a malware researcher might want to copy the output log of his script from his remote VM to his desktop.

It was roughly at this point, while I was trying to figure out the flow of the data, Omer ([@GullOmer](#)) asked me if and where [PathCanonicalizeA](#) is called. If the client fails to properly canonicalize and sanitize the file paths it receives, it could be vulnerable to a path-traversal attack, allowing the server to drop arbitrary files in arbitrary paths on the client's computer, a very strong attack primitive. After failing to find imports for the canonicalization function, we dug in deeper, trying to figure out the overall architecture for this data flow.

Figure 8 summarizes our findings:



**Figure 8:** Architecture of the clipboard sharing in Microsoft's RDP.

This is where `rdpclip.exe` comes into play. It turns out that the server accesses the clipboard through a broker, which is `rdpclip.exe`. In fact, `rdpclip.exe` is just a normal process (we can kill / spawn it ourselves) that interacts with the RDP service using a dedicated virtual channel API.

At this point, we installed [ClipSpy](#) (and later on switched to use [InsideClipboard](#)), and started to dynamically debug the clipboard's data handling that is done inside `rdpclip.exe`.

These are our conclusions regarding the data flow in an ordinary "Copy & Paste" operation in which a file is copied from the server to the client:

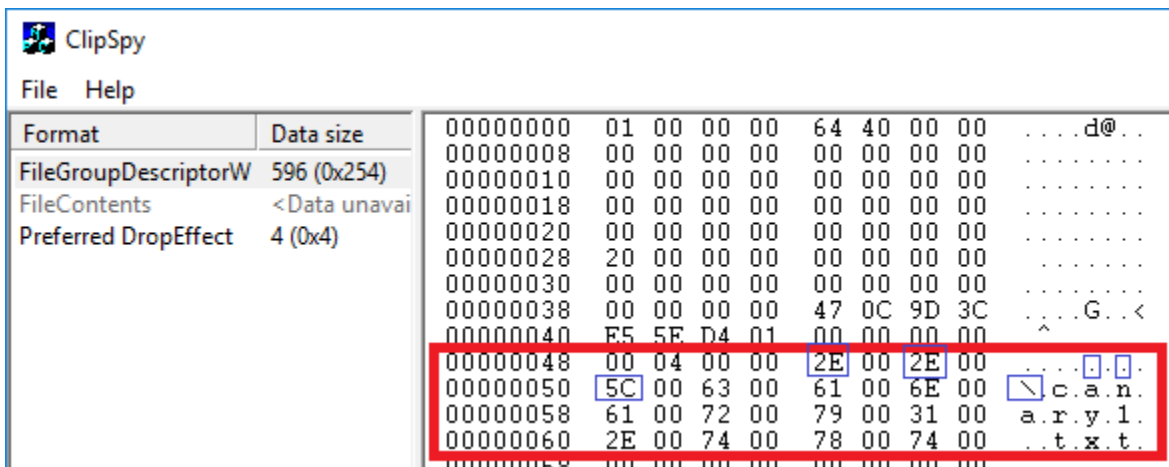
1. On the server, "copy" operation creates multiple Clipboard formats, including:
  - a. `CF_HDROP`
  - b. `CF_FileGroupDescriptorW`
2. When the "paste" operation is performed in the client's computer, a chain of events is triggered.
3. The `rdpclip.exe` process on the server is asked for the Clipboard's `CF_FileGroupDescriptorW` format, and internally converts it into a request for format `CF_HDROP` from its own Clipboard.
4. The Clipboard `CF_HDROP` format is then converted into `CF_FileGroupDescriptorW` (FGDw): the metadata of the files is added to the descriptor one at a time, using the `HdropToFgdConverter::AddItemToFgd()` function.
5. After completion, the FGDw blob is sent to the RDP service on the server.
6. The server simply wraps it and sends it to the client.

7. The client unwraps it and stores it in its own clipboard.
8. A “paste” event is sent to the process of the focused window (for example, `explorer.exe`).
9. This process handles the event and reads the data from the clipboard.
10. The content of the files is received over the RDP connection itself.

## Path Traversal over the shared RDP clipboard

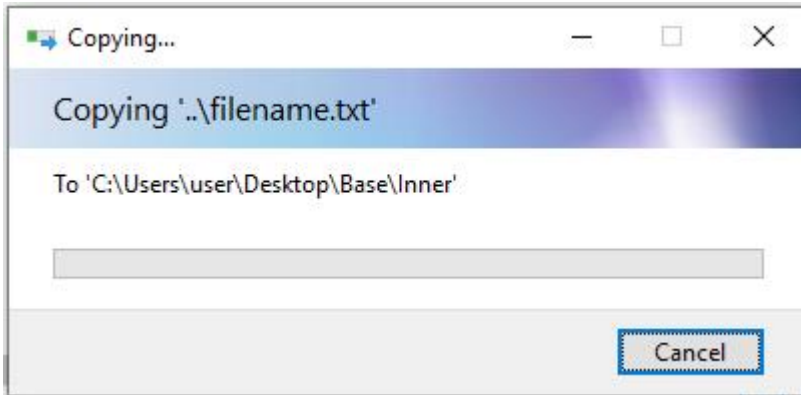
If we look back on the steps performed on the received clipboard data, we notice that the client doesn't verify the received `FGDw` blob that came from the RDP server. And indeed, if we modify the server to include a path traversal path of the form:

`..\canary1.txt`, ClipSpy shows us (see Figure 9) that it was stored “as is” on the client's clipboard:



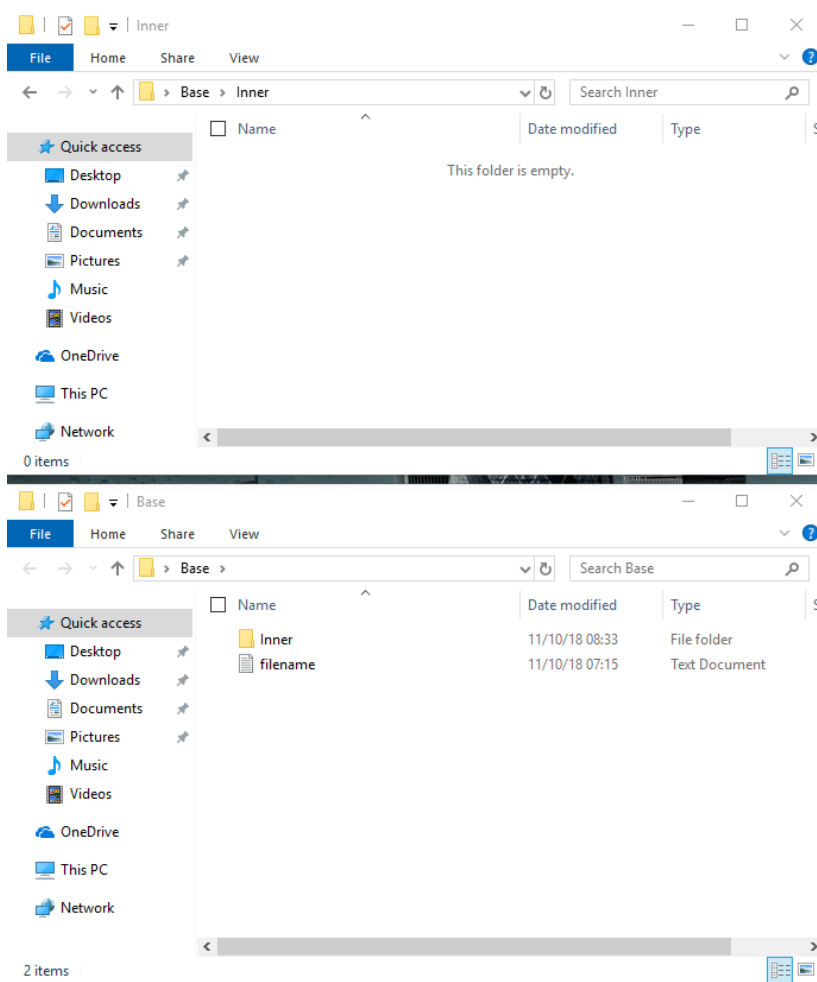
**Figure 9:** `FGDw` with a path-traversal was stored on the client's clipboard

In Figure 10, we can see how `explorer.exe` treats a path traversal of `..\filename.txt`:



**Figure 10:** FGDw with a path-traversal as `explorer.exe` handles it.

Just to make sure, after the “paste” operation in folder “Inner”, the file is stored to “Base” instead:



**Figure 11:** Folders after a successful path traversal attack.

And that’s practically it.

If a client uses the “Copy & Paste” feature over an RDP connection, a malicious RDP server can transparently drop arbitrary files to arbitrary file locations on the client’s computer, limited only by the permission of the client. For example, one can drop malicious scripts to the client’s “Startup” folder, and after a reboot they will be executed on his computer, providing full control. Here is a video of our PoC exploit:

[https://youtu.be/F70FGv\\_QxDY](https://youtu.be/F70FGv_QxDY)

**Note:** In this exploit, we simply killed `rdpclip.exe`, and spawned our own process to perform the path traversal attack by adding additional malicious file to every “Copy & Paste” operation. The attack was performed with “user” permissions, and does not require the attacker to have “system” or any other elevated permission.

## Taking it one step further

Every time a clipboard is updated on either side of the RDP connection, a [CLIPRDR FORMAT LIST](#) message is sent to the other side, to notify it about the new clipboard formats that are now available. We can think of it as a complete sync between the clipboards of both parties (except for a small set of formats that are treated differently by the RDP connection itself). This means that our malicious server is notified whenever the client copies something to his “local” clipboard, and it can now query the values and read them. In addition, the server can notify the client about a fake clipboard “update”, without an actual “copy” operation inside the RDP window, thus completely controlling the client’s clipboard without being noticed.

### Attack Scenario #1:

A malicious RDP server can eavesdrop on the client’s clipboard – this is a **feature**, not a bug. For example, the client locally copies an admin password, and the server can read it.

### Attack Scenario #2:

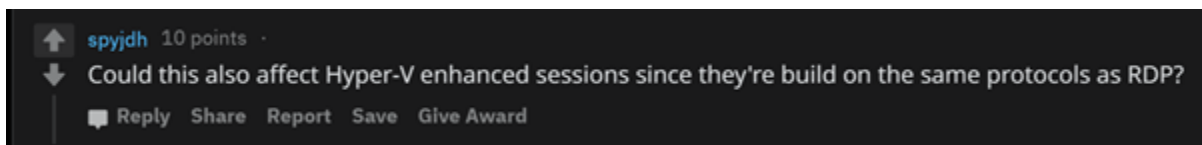
A malicious RDP server can modify any clipboard content used by the client, even if the client does not initiate a “copy” operation from the RDP window. If you click “paste”

when an RDP connection is open, you are vulnerable to this kind of attack. For example, if you copy a file on your computer, the server can modify your (executable?) file / piggy-back your copy to add additional files / path-traversal files using the previously shown PoC.

**Note:** The content of the synced clipboard is subject to [Delayed Rendering](#). This means that the clipboard's content is sent over the RDP connection only after a program actively requests it, usually by clicking "paste". Until then, the clipboard only holds the list of formats that are available, without holding the content.

## The Hyper-V Connection

Following our [initial publication](#), we received numerous comments asking if the vulnerabilities in Microsoft's RDP client can affect Microsoft's Hyper-V product. Here is one such question, from the [reddit discussion](#) over the publication:



**Figure 12:** A reddit comment asking about the Hyper-V implications of the vulnerabilities.

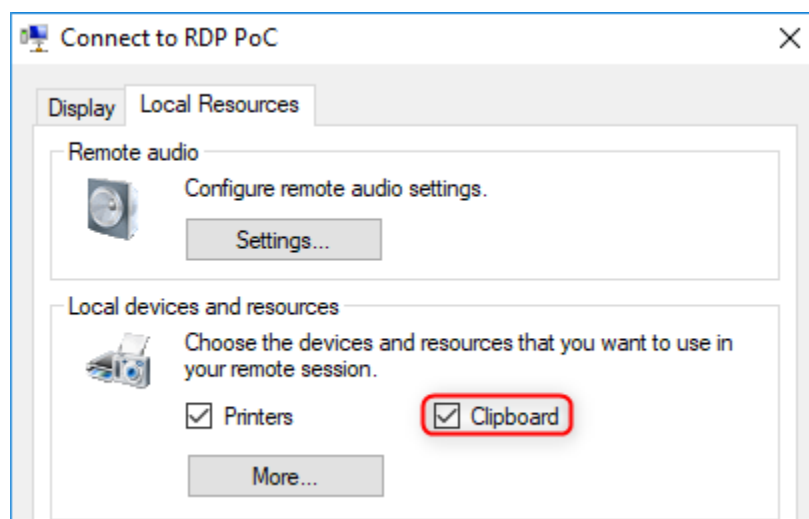
Microsoft's Hyper-V is a virtualization technology that is used in Microsoft's [Azure](#) cloud, and offered as a virtualization product on top of the Windows 10 operating system. Like other virtualization technologies, Hyper-V includes a graphical user interface that enables managing the local / remote virtual machine (VM).

This information on Hyper-V raises the question: Does Hyper-V use RDP? And the answer could be found in the term "enhanced sessions", which is Microsoft's equivalent of [VMWare](#)'s VM tools. Enhanced Sessions offer an extended functionality that includes **clipboard synchronization** between the guest and the host. As you may recall, we already found a path-traversal vulnerability in the clipboard synchronization implemented by Microsoft's RDP client. We decided to check if the same vulnerability will work out-of-the-box for the Hyper-V case.

To perform the tests, we installed Hyper-V on our machine, and converted our initial virtual machine into a Hyper-V machine. We then used the Hyper-V Manager program



and connected to our virtual machine, and soon enough we stumbled upon the familiar settings window seen in Figure 13.



**Figure 13:** Settings window for Hyper-V VM. The same window as in MSTSC.exe.

As can be seen, this is exactly the same GUI window that is used for the settings of an RDP connection when using `mstsc.exe`. This looks promising. Especially since the Clipboard resource is shared by default, meaning that this is an enhanced session. Feeling motivated, we run the same script we initially submitted to Microsoft in the `mstsc.exe` case, and it **worked**. We just found a **Hyper-V guest-to-host VM escape** over the control interface, using the RDP vulnerability! Here is a video of a paste-only attack for the Hyper-V scenario:

<https://youtu.be/nSGIMJqQEh0>

## RDP & Hyper-V - Explained

It turns out that RDP is used behind the scenes as the control plane for Hyper-V. Instead of re-implementing screen-sharing, remote keyboard and synchronized clipboard features, Microsoft decided that all of these features are already implemented as part of RDP, so why not use it in this case as well? In addition, further investigation revealed that other RDP versions are used for Windows Defender Application Guard (WDAG).

While it was hard for any security researcher to miss Microsoft's effort to test and improve the security of its Hyper-V technology, we can learn an important lesson from this research. As the saying goes: your system is as strong as its weakest link. In other words, by being dependent on other software libraries, Hyper-V Manager inherits security vulnerabilities that are found in RDP, and in other libraries it uses.

This lesson might sound trivial. However, the problem of updating versions and keeping track of vulnerabilities in external software dependencies is known to be notoriously hard.

### **Note on WDAG:**

Windows Defender Application Guard (WDAG) is a virtualization solution for "Edge" browser, that is used when browsing to "risky" websites. WDAG uses RDP to present the secured browser, and it uses `hvsirdpclient.exe` instead of `mstsc.exe`.

Due to the limited Clipboard functionality that is needed when communicating with the secured browser (Javascript is limited to a basic subset of clipboard formats),

`hvsirdpclient.exe` limits the Clipboard feature in multiple levels:

- The clipboard is off by default
- When enabled, the clipboard supports only 2 format types: Text & Images

HVSI has a hooking point inside `CFormatNamePacker::IsExcludedFormat()`:

- Proprietary filtering logic is invoked **instead** of the format ID / format Name blacklists
- The hooking point calls:

```
CHvsiApi::IsHvsiClipboardFileTypeExcluded()
```

# Defense

## Detection

Having identified RDP vulnerabilities, we set out to ensure users are protected.

However, fixing the vulnerability and releasing a patch takes time, and users do not always maintain an up-to-date system. Thus, users remain vulnerable until the patch is developed, released and installed. Our research set out to discover how can we secure users without any updates.

We developed a post-breach detection, leveraging built-in indications. This detection is able to notify the users in real time if this vulnerability has been exploited on their machine.

In order to create an effective detection, we laid these assumptions:

1. The detection may use existing optics, available to all Windows 10 versions; we realized that requiring any application update contradicts the purpose of this exercise.
2. The detection logic should detect the threat from the “victim” perspective; this lateral movement vulnerability involves two machines. A remote, compromised machine and a host client that initiated the RDP connection. The detection should detect files that are transformed from the compromised machine to the client machine, and therefore should be implemented from the client’s perspective. This means that we must rely solely on telemetry that is triggered on the client machine.
3. RDP anomaly will not detect the threat; since the RDP connection is initiated by the victim machine, and more specifically by the user, we don’t expect an abnormal connection to occur. User behavior is normal, according to anomaly detection, and therefore won’t allow detection in this scenario.

## Event Tracing for Windows (ETW)

[Event Tracing for Windows \(ETW\)](#) is an efficient kernel-level tracing facility that lets you log kernel or application-defined events to a log file. You can consume the events in real

time or from a log file and use them to debug an application or to determine where performance issues are occurring in the application. This is a built-in feature in Windows 10.

Initially, the purpose of tracing was performance tracking, quality assurance and cross application communication. Recently, security products began consuming various traces for detection purposes. This is not trivial, since the traces were not created, nor designed, to support security scenarios, a gap we will observe in this research.

In addition, events don't include private information from the application, and therefore remain mostly general.

ETW events can be consumed using Windows Event Viewer application.

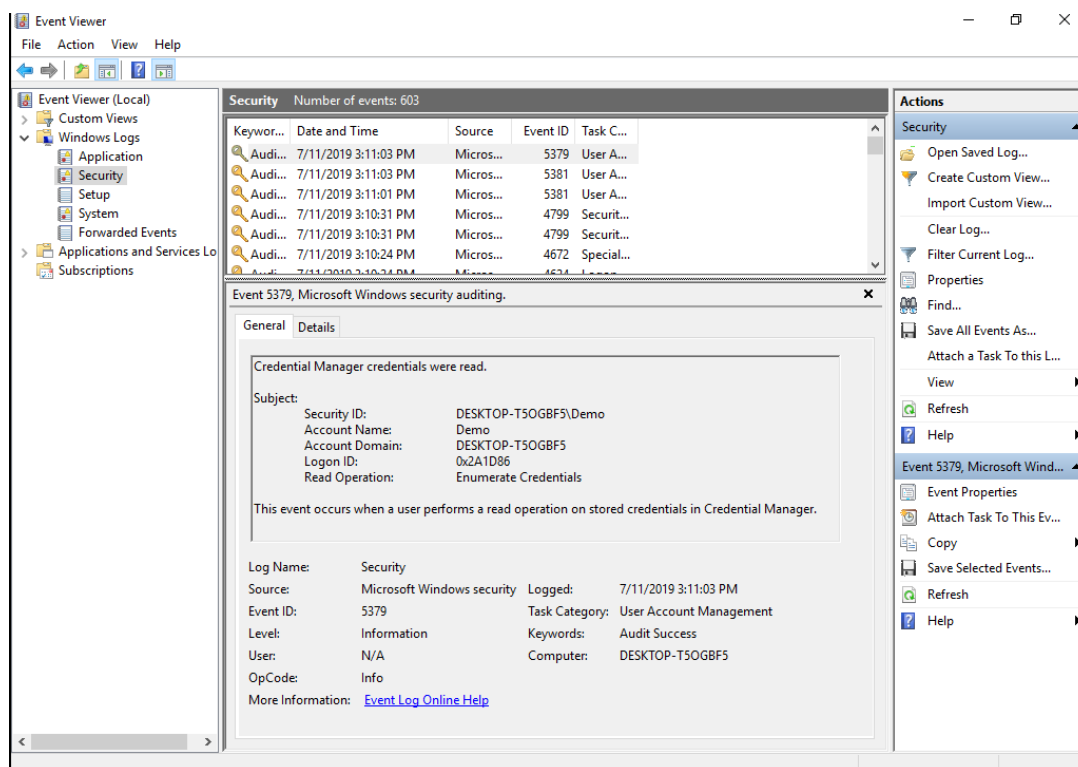


Figure 14: Windows Event Viewer – Security logs.

In order to consume ETW events, the application must subscribe to the relevant ETW provider. For some providers, a manifest file is available and includes information about the events that can be consumed within. However, a manifest file is not mandatory, and some providers do not provide this information. For our detection, we used RDP and clipboard providers.

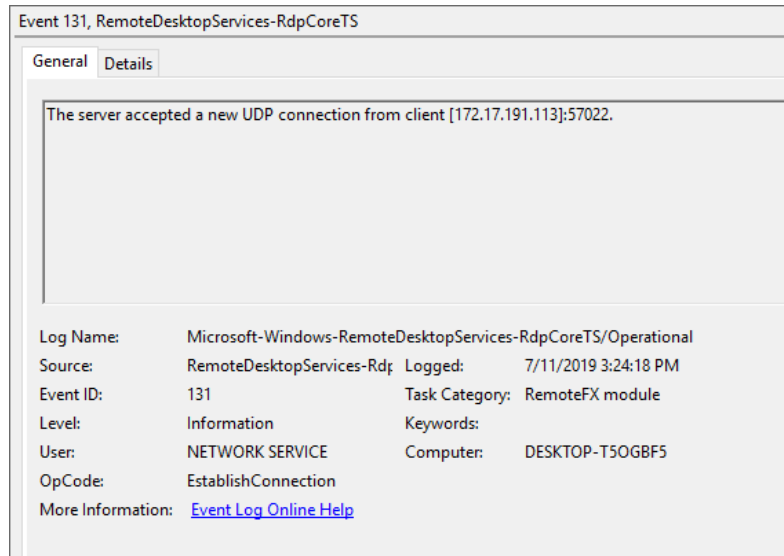
## RDP Connection Events

**Provider name:** Microsoft-Windows-RemoteDesktopServices-RdpCoreTS

**Provider guid:** 1139c61b-b549-4251-8ed3-27250a1edec8

This is a manifested provider, that includes two relevant events:

1. Event 131 – Triggered when a new RDP connection is accepted.
2. Event 132 – A keep alive indication for an RDP channel.



**Figure 15:** RDP event 131 – accepted a new connection.

## Clipboard Events

**Provider name:** Microsoft.Windows.OLE.Clipboard

**Provider guid:** 3e0e3a92-b00b-4456-9dee-f40aba77f00e

This is a non-manifested provider, tracing clipboard API usages.

The relevant task name for our detection is OLE\_Clipboard\_MethodDiagnostics

Message/PartC	
ApiName:	CClipDataObject::GetData
CLIPFORMAT:	Performed DropEffect
ClipboardDataObjectTask:	0x0
HRESULT:	0x80040064
MatchFormatetc:	1
STGMEDIUM:	0x9BDA00
m_pDataObject:	0x6404478
tymed:	1

**Figure 16:** Clipboard.

The event is triggered when one of the clipboard API is called, and includes the following properties:

- **ApiName:** Called API; can be SetData, GetData, Create, and other methods that relate to the clipboard functionality.
- **CLIPFORMAT:** The returned clipboard format (bitmap, text, Unicode text, etc.).
- **HRESULT:** Indicates whether the API call ended successfully.
- **MatchFormatetc:** Indicates whether a match for the requested format was found. Possible values:
  - FORMAT\_NOTFOUND = 1
  - FORMAT\_BADMATCH = 2
  - FORMAT\_GOODMATCH = 4

This information is relevant for cases where the user is trying to paste items to a different format. For example: copying files into text editor, copying text to desktop, etc.

- **STGMEDIUM:** Represents a generalized global memory handle used for data transfer operations by some interfaces.
- **Tymed:** Indicates the type of storage medium being used in a data transfer (file, stream, etc.).

**Note:** The clipboard content is undisclosed on the event, for obvious reasons:

- The clipboard serves different formats; including the content requires parsing of the different formats into one format.
- This is private information. The clipboard content could potentially include text with passwords, which should not be available to other applications.

A demo of the telemetry triggering for the attack scenario can be found here:

[https://www.youtube.com/watch?v=q9Lox\\_rfqvw](https://www.youtube.com/watch?v=q9Lox_rfqvw)

## Basic Detection Logic

Based on the existing ETW telemetry, we can create basic detection logic that should cover this attack scenario. However, this detection is too generic, and covers benign scenarios as well.

### Detection 1: ETW events only

1. Observe RDP session event.

2. Observe multiple files being pasted in a short period of time.
3. Trigger scan.

**Details:** We can use events 131 and 132 to identify RDP connection in action. If one of these events has been triggered, this indicates an active RDP connection.

If we recognize multiple files pasting in a short period of time, we should suspect that the vulnerability is exploited. Since this is a very broad scenario (after all, copying multiple files could be a non-malicious user scenario), the recommendation would be to scan the machine in suspicious mode.

**Note:** This is a theoretical detection logic that can be implemented using basic optics. However, for the average machine, with average usage, this will cause over detection. Therefore, we developed an improved logic.

### File Creation Event

Most security vendors track new files creation on machines. There are multiple ways to implement this indication. A well-known example is [ProcMon](#), which enables a new file creation indication. This capability is important for preventive purposes, like triggering a file scan or blocking known malicious files by signature.

File creation events should include the following basic information:

1. File name (including full path).
2. Creation time.

Some vendors include more data in this event, but this basic data suffices to detect this scenario.

### Detection Logic

RDP events, clipboard events and file creation events provide enough data to cover this RDP vulnerability scenario.

### **Detection 2: ETW events and file creation**

1. Observe RDP session event.
2. Observe multiple files being pasted in a short period of time.
3. Correlate file creation and pasting timestamps.
4. If the correlated files are in different directories – alert!

In this detection, like in Detection 1, we recognize RDP sessions using events 131 and 132. Like in detection 1, we recognize cases of multiple files pasting in a short period (within a few seconds). In this detection, we can correlate the new created files with the pasting timestamps. Depending on the implementation of the file creation event, we might observe a slight time difference between the paste event time and the file creation time.

The next step is to compare the file's path and search for anomalies. If the files are pasted into different directories, we discover an anomaly case that should be flagged.

**Note:** Since a user can copy not only a single file, but a whole directory, in some cases we see multiple files under the same sub-directories. This is not a malicious scenario. Therefore, files creation under sub-directories should not be detected.

When detecting a new attack scenario, we try to identify the generic malicious behavior this scenario introduces, and develop a detection as general as possible. This would also help counter corner cases, and account for tweaks to the attack scenario. For this reason, we identified additional detection logics that cover this attack scenario end-to-end, focusing on the behaviors that the attack introduces:

1. Startup folder as a destination: Since the goal of the initial attack scenario was to drop files in the startup folder, we can explicitly monitor this folder when new files are created under it. This can include an anomaly detection for file creation events under the startup folder, using multiple features such as the file signature, creation process, etc. In addition, the file can be verified using known scanning capabilities, such as known file malware, static analysis, etc.
2. Clipboard as an attack vector:
  - a. **File pasting anomaly:** ML based detection can recognize files that are pasted to different directories in a short period of time, in a similar way to Detection Logic 2, but in a heuristic manner. The anomaly features can be the number of pasted files or the files directories.
3. Malicious files dropping:
  - a. **File creation anomaly:** ML based detection can recognize an anomaly in file creation path. The anomaly features can be file path, creation time and



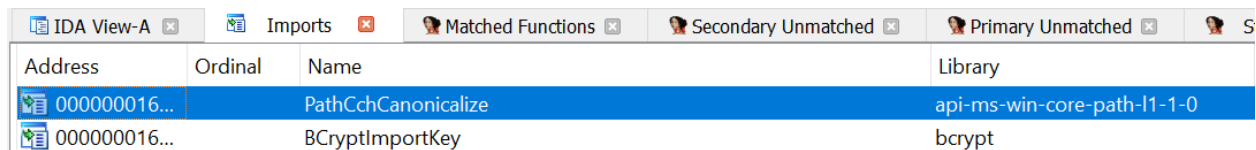
file name. This detection covers a broader scenario, dropping malicious files on “victim” machine, without considering the specific method.

## Patch

Immediately after the disclosure of the Hyper-V implications of the vulnerability to MSRC, they began investigating. Within a few weeks an MSRC ticket was opened for it, later leading to an official CVE, [CVE-2019-0887](#), and a [patch](#).

Before we start analyzing the patch, let’s recap the vulnerability. A malicious RDP server can send a crafted file transfer clipboard content, that will cause a Path Traversal on the client’s machine. Therefore, we expect to see that `mstscax.dll` will check the received `FileGroupDescriptorW` clipboard format, and sanitize each file path contained within.

We used [BinDiff](#) to analyze Microsoft’s fix, but unfortunately, it found dozens of modified functions. Sifting through this, we examined the import section and found a new entry, shown in Figure 17.

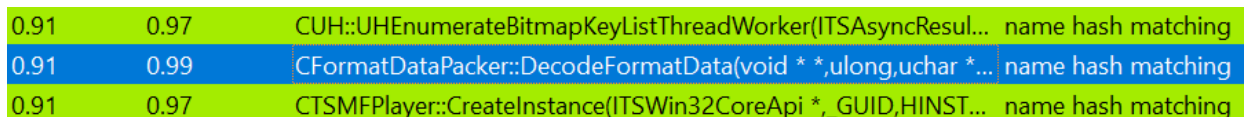


The screenshot shows the IDA Pro interface with the 'Imports' window open. The window title is 'Imports' and it contains a table of imported functions. The table has four columns: 'Address', 'Ordinal', 'Name', and 'Library'. The first row is highlighted in blue and shows the address '00000016...', ordinal '0', name 'PathCchCanonicalize', and library 'api-ms-win-core-path-l1-1-0'. The second row shows the address '00000016...', ordinal '0', name 'BCryptImportKey', and library 'bcrypt'. The window also shows tabs for 'Matched Functions', 'Secondary Unmatched', and 'Primary Unmatched'.

Address	Ordinal	Name	Library
00000016...	0	PathCchCanonicalize	api-ms-win-core-path-l1-1-0
00000016...	0	BCryptImportKey	bcrypt

**Figure 17:** `PathCchCanonicalize()` is now used by the patched `.dll` file.

According to [MSDN](#), the function `PathCchCanonicalize()` “Converts a path string into a canonical form.” This is precisely what was missing in the first place, now let’s look who invokes this function. Soon enough, we were able to spot the function that introduces the fix:



The screenshot shows a table of function changes from BinDiff. The table has three columns: 'Address', 'Ordinal', and 'Name'. The first row is highlighted in green and shows address '0.91', ordinal '0.97', and name 'CUH::UHEnumerateBitmapKeyListThreadWorker(ITSAsyncResul... name hash matching'. The second row is highlighted in blue and shows address '0.91', ordinal '0.99', and name 'CFormatDataPacker::DecodeFormatData(void \* \*,ulong,uchar \*... name hash matching'. The third row is highlighted in green and shows address '0.91', ordinal '0.97', and name 'CTSMFPlayer::CreateInstance(ITSWin32CoreApi \*,\_GUID,HINST... name hash matching'.

Address	Ordinal	Name
0.91	0.97	CUH::UHEnumerateBitmapKeyListThreadWorker(ITSAsyncResul... name hash matching
0.91	0.99	CFormatDataPacker::DecodeFormatData(void * *,ulong,uchar *... name hash matching
0.91	0.97	CTSMFPlayer::CreateInstance(ITSWin32CoreApi *,_GUID,HINST... name hash matching

**Figure 18:** `CFormatDataPacker::DecodeFormatData()` was changed, as shown in BinDiff.

The change is presented in Figure 19:

```

0000000016AB893FC CFormatDataPacker::DecodeFormatData(void **,ulong,uchar *,ulong)
0000000016AB89640 call      ?FileDescriptorW@CClipFormatTypes@@QEAAIXZ
0000000016AB89645 xor      b4 r9d, b4 r9d
0000000016AB89648 mov      b4 r8d, b4 ebp           // unsigned int
0000000016AB8964B cmp      b4 ebx, b4 eax
0000000016AB8964D mov      rdx, r14                // unsigned __int8 *
0000000016AB89650 lea     rax, ss:[rsp+arg_10]
0000000016AB89655 setz    b1 r9b                   // int
0000000016AB89659 mov     ss:[rsp+var_28], rax      // int *
0000000016AB8965E call    ?ValidateFilePaths@CFormatDataPacker@@AEAAJPEAEKHPEAH@Z
0000000016AB89663 mov     b4 ebx, b4 eax
0000000016AB89665 test    b4 eax, b4 eax
0000000016AB89667 jns     0x16AB896AD

```

**Figure 19:** A call to CFormatDataPacker::ValidateFilePaths() was added to the flow.

When handling incoming FileGroupDescriptorW file formats, the client transfers the format to a new function that verifies the blob's structure. This new function checks that the data is structured correctly, and then calculates the canonicalized form for each filename:

```

pszFilename = pCurrentFileRecord->szFilename;
status_code = PathCchCanonicalize(&pszPathOut, 0x104ui64, pszFilename);
if ( (status_code & 0x80000000) != 0 )
{

```

**Figure 20:** Calculate the canonicalized form for every filename.

If successful, the canonicalized output is compared to the original filename, and any mismatch between the two will result in an error. This means that if the filename contains strings of the form "." or "..", it will be changed when converted to the canonicalized form, thus failing the validity check.

It appears that the fix matches our initial expectations from this feature, and therefore our Path Traversal vulnerability is now fixed.

# Conclusion

## Design lesson: Think twice before connecting different modules

Originally, Clipboards were designed to be used locally. This feature is not defined as a trusted feature, as all applications have access to the clipboard content. However, according to [Microsoft](#), The clipboard is user-driven. A window should transfer data to or from the clipboard only in response to a command from the user. A window must not use the clipboard to transfer data without the user's knowledge. For this reason, even if the clipboard is not officially trusted, it is referred to as trusted, especially within the same machine.

When Microsoft created machine-sharing features, user experience dictated to enable clipboard sharing. However, this exposed machines to clipboards they can no longer trust. Our research revealed this important design lesson, that when a feature was developed under certain circumstances, when the environment changes, features must be reconsidered. This is further exacerbated, as this RDP vulnerability was then replicated across multiple applications.

## Windows telemetry is an important tool in the defender's toolbox

ETW events are not designed for security purposes. Nevertheless, they can be used for security purposes. Our research presents an example of how to leverage ETW information to detect malicious behavior. This is a powerful tool that enables the creation of new detections without an OS update. We believe that defenders should be aware of this tool when developing defense strategy. In addition, this tool should be considered by attackers; we already see attackers who clear event logs to cover their tracks. The next phase for attackers is to consider attack techniques based on available traces, avoiding components that are well traced.

## Our industry can benefit from cross-community collaborations

This cross-company, cross-continent research demonstrates the potential impact of collaboration within the security community. We were able to discover new vulnerabilities, fix them and secure users efficiently, while learning important lessons

that we can share with the industry. This is the result of transparency, trust and knowledge-sharing in the security research community.

## Appendix A – CVEs found in `rdesktop`:

- [CVE 2018-8791](#): `rdesktop` versions up to and including v1.8.3 contain an Out-Of-Bounds Read in function `rdpdr_process()` that results in an information leak.
- [CVE 2018-8792](#): `rdesktop` versions up to and including v1.8.3 contain an Out-Of-Bounds Read in function `cssp_read_tsrequest()` that results in a Denial of Service (segfault).
- [CVE 2018-8793](#): `rdesktop` versions up to and including v1.8.3 contain a Heap-Based Buffer Overflow in function `cssp_read_tsrequest()` that results in a memory corruption and probably even a remote code execution.
- [CVE 2018-8794](#): `rdesktop` versions up to and including v1.8.3 contain an Integer Overflow that leads to an Out-Of-Bounds Write in function `process_bitmap_updates()` and results in a memory corruption and possibly even a remote code execution.
- [CVE 2018-8795](#): `rdesktop` versions up to and including v1.8.3 contain an Integer Overflow that leads to a Heap-Based Buffer Overflow in function `process_bitmap_updates()` and results in a memory corruption and probably even a remote code execution.
- [CVE 2018-8796](#): `rdesktop` versions up to and including v1.8.3 contain an Out-Of-Bounds Read in function `process_bitmap_updates()` that results in a Denial of Service (segfault).
- [CVE 2018-8797](#): `rdesktop` versions up to and including v1.8.3 contain a Heap-Based Buffer Overflow in function `process_plane()` that results in a memory corruption and probably even a remote code execution.
- [CVE 2018-8798](#): `rdesktop` versions up to and including v1.8.3 contain an Out-Of-Bounds Read in function `rdpsnd_process_ping()` that results in an information leak.
- [CVE 2018-8799](#): `rdesktop` versions up to and including v1.8.3 contain an Out-Of-Bounds Read in function `process_secondary_order()` that results in a Denial of Service (segfault).

- [CVE 2018-8800](#): rdesktop versions up to and including v1.8.3 contain a Heap-Based Buffer Overflow in function `ui_clip_handle_data()` that results in a memory corruption and probably even a remote code execution.
- [CVE 2018-20174](#): rdesktop versions up to and including v1.8.3 contain an Out-Of-Bounds Read in function `ui_clip_handle_data()` that results in an information leak.
- [CVE 2018-20175](#): rdesktop versions up to and including v1.8.3 contains several Integer Signedness errors that leads to Out-Of-Bounds Reads in file `mcs.c` and result in a Denial of Service (segfault).
- [CVE 2018-20176](#): rdesktop versions up to and including v1.8.3 contains several Out-Of-Bounds Reads in file `secure.c` that result in a Denial of Service (segfault).
- [CVE 2018-20177](#): rdesktop versions up to and including v1.8.3 contain an Integer Overflow that leads to a Heap-Based Buffer Overflow in function `rdp_in_unistr()` and results in a memory corruption and possibly even a remote code execution.
- [CVE 2018-20178](#): rdesktop versions up to and including v1.8.3 contain an Out-Of-Bounds Read in function `process_demand_active()` that results in a Denial of Service (segfault).
- [CVE 2018-20179](#): rdesktop versions up to and including v1.8.3 contain an Integer Underflow that leads to a Heap-Based Buffer Overflow in function `lspci_process()` and results in a memory corruption and probably even a remote code execution.
- [CVE 2018-20180](#): rdesktop versions up to and including v1.8.3 contain an Integer Underflow that leads to a Heap-Based Buffer Overflow in function `rdpsnddbg_process()` and results in a memory corruption and probably even a remote code execution.
- [CVE 2018-20181](#): rdesktop versions up to and including v1.8.3 contain an Integer Underflow that leads to a Heap-Based Buffer Overflow in function `seamless_process()` and results in a memory corruption and probably even a remote code execution.

- [CVE 2018-20182](#): rdesktop versions up to and including v1.8.3 contain a Buffer Overflow over the global variables in function `seamless_process_line()` that results in a memory corruption and probably even a remote code execution.

## Appendix B – CVEs found in FreeRDP:

- [CVE 2018-8784](#): FreeRDP prior to version 2.0.0-rc4 contains a Heap-Based Buffer Overflow in function `zgfx_decompress_segment()` that results in a memory corruption and probably even a remote code execution.
- [CVE 2018-8785](#): FreeRDP prior to version 2.0.0-rc4 contains a Heap-Based Buffer Overflow in function `zgfx_decompress()` that results in a memory corruption and probably even a remote code execution.
- [CVE 2018-8786](#): FreeRDP prior to version 2.0.0-rc4 contains an Integer Truncation that leads to a Heap-Based Buffer Overflow in function `update_read_bitmap_update()` and results in a memory corruption and probably even a remote code execution.
- [CVE 2018-8787](#): FreeRDP prior to version 2.0.0-rc4 contains an Integer Overflow that leads to a Heap-Based Buffer Overflow in function `gdi_Bitmap-Decompress()` and results in a memory corruption and probably even a remote code execution.
- [CVE 2018-8788](#): FreeRDP prior to version 2.0.0-rc4 contains an Out-Of-Bounds Write of up to 4 bytes in function `nsc_rle_decode()` that results in a memory corruption and possibly even a remote code execution.
- [CVE 2018-8789](#): FreeRDP prior to version 2.0.0-rc4 contains several Out-Of-Bounds Reads in the NTLM Authentication module that results in a Denial of Service (segfault).