0.1 CVE-2021-24086

According to Microsoft[4] CVE-2021-24 086 is a denial of service vulnerability with a CVSS:3.0 score of 7.5 / 6.5, that is a base score metrics of 7.5 and a temporal score metrics of 6.5. The vulnerability affects all supported versions of Windows and Windows Server. According to an accompanied blog post published by Microsoft [6] at the same time as the patch was released, details that the vulnerable component is the Windows TCP/IP implementation, and that the vulnerability revolves around IPv6 fragmentation. The Security Update guide and the blog post also present a workaround that can be used to temporarily mitigate the vulnerability by disabling IPv6 fragmentation.

Figure out if this should be here

0.1.1 Public information

Due to the Microsoft Active Protetions Program (MAPP)[5] security software providers are given early access to vulnerability information. This information often include Proof of Concept (PoC)s for vulnerabilities to be patched, in order to aid security software providers to create valid detections for exploitation of soon-to-be patched vulnerabilities. Due to MAPP, some security software providers publish relevant information regarding recently patched vulnerabilities. However, the information is usually very vague in details, and can therefore only aid in the initial exploration of the vulnerability. For CVE-2021-24086, both McAfee[9] and Palo Alto[8] posted public information about CVE-2021-24086. However, both articles contained very limited details, and is therefore far from sufficient to reproduce the vulnerability. Before trying to rediscover the vulnerability, the following information is available:

- The vulnerability lies within the handling om fragmented packets in IPv6
- The relevant code lies within the `tcpip.sys` drivers
- The root cause of the vulnerability is a NULL pointer dereference in `Ip⌋v6ReassembleDatagram` of `tcpip.sys`
- The reassembled packet should contain around 0xFFFF (65535) bytes of extension headers, which is usually not possible

0.1.2 Binary diffing

The usage of binary diffing to gather information about patched vulnerabilities is well described in current research[7][10], and has been made popular and easy to do by tools such as Bindiff[11] and Diaphora[3].

write a little about how bindiffing works. Or don't idc.

If we look at figure 1 we can compare the function changes of the patched and not-patched `tcpip.sys`. Looking at `tcpip!Ipv6pReassembleDatagram` we can see that the similarity factor is only 0.38 telling us that a significant amount of code has been changed.

Similarity	Confid	Change	EA Primary	Name Primary	EA Secondary	Name Secondary
0.16	0.27	GI--E--	00000001C018D794	sub_00000001C018D794	00000001C015A1D6	sub_00000001C015A1D6
0.27	0.42	GI--EL-	00000001C01905B5	sub_00000001C01905B5	00000001C01568FC	IppCleanupPathPrimitive
0.31	0.73	GI--E--	00000001C0190F38	Ipv4pReassembleDatagram	00000001C0190F68	Ipv4pReassembleDatagram
0.38	0.98	GI--E--	00000001C0199FAC	Ipv6pReassembleDatagram	00000001C019A0AC	Ipv6pReassembleDatagram
0.42	0.62	-I--E--	00000001C0154959	sub_00000001C0154959	00000001C0001E42	sub_00000001C0001E42
0.54	0.96	GI-----	00000001C019A658	Ipv6pReceiveFragment	00000001C019A7F8	Ipv6pReceiveFragment

Figure 1: Primary matched functions of `tcpip.sys`

Diving into the binary diff of `tcpip!Ipv6pReassembleDatagram` as seen on listing 1, we can clearly see a change. The first many changes from line *5-39* are simply register changes and other insignificant changes due to how the compiler works. However, on line *41-42* a new comparison is made to ensure that the value of the register `edx` is less than 0xFFFF. This matches the statement given in subsection 0.1.1 (Public information), that the vulnerability is triggered by a packet of around 0xFFFF bytes.

```
--- "a/.\\unpatched tcpip.sys"
+++ "b/.\\patched tcpip.sys"
@@ -1,6 +1,4 @@
-sub     rsp, 58h        ; Integer Subtraction
+sub     rsp, 60h        ; Integer Subtraction
 movzx   r9d, word ptr [rdx+88h] ; Move with Zero-Extend
 mov     rdi, rdx
 mov     edx, [rdx+8Ch]
-mov     bl, r8b
+mov     r13b, r8b
 add     edx, r9d        ; Add
-mov     byte ptr [rsp+98h+var_70], 0
-and     [rsp+98h+var_78], 0 ; Logical AND
 mov     [rsp+98h+length], edx
 lea     eax, [rdx+28h]  ; Load Effective Address
-mov     rdx, rdi
 mov     [rsp+98h+var_68], eax
 lea     eax, [r9+28h]   ; Load Effective Address
 mov     [rsp+98h+BytesNeeded], eax
-xor     r9d, r9d        ; Logical Exclusive OR
 mov     rax, [rcx+0D0h]
-lea     rcx, IppReassemblyNetBufferListsComplete ; Load Effective
↪  Address
-mov     r13, [rax+8]
-mov     rax, [r13+0]
+mov     r12, [rax+8]
+mov     rax, [r12]
 mov     r15, [rax+28h]
 mov     eax, gs:1A4h
 mov     r8d, eax
-mov     rax, [r13+388h]
+mov     rax, [r12+388h]
 lea     rbp, [r8+r8*2]  ; Load Effective Address
-mov     r12, [rax+r8*8]
-xor     r8d, r8d        ; Logical Exclusive OR
+mov     rcx, [rax+r8*8]
 shl     rbp, 6          ; Shift Logical Left
-add     rbp, [r15+4728h] ; Add
+add     rbp, [r15+4728h] ; Add
+mov     [rsp+98h+var_58], rcx
+cmp     edx, 0FFFFh     ; Compare Two Operands
+jbe     short loc_1C019A186 ; Jump if Below or Equal (CF=1 | ZF=1)
```

Listing 1: Diff of patched and vulnerable `Ipv6pReassembleDatagram`

Looking at the raw assembly without any knowledge of what the registers contain or what parameters are passed to the function can be very confusing. To make it easier for the reader to follow, listing 2 contains the annotated

decompiled code of the vulnerable and patched `tcpip!Ipv6pReassembleDatagram` function. Here the patch is easy to spot, as the call to `tcpip!NetioAllocateAndReferenceNetBufferAndNetBufferList` is replaced with the check that we also observed in listing 1. The check is there to ensure that the total packet size is less than 0xFFFF, which is the largest 16 bit value. The packet size is calculated on line *4-6* using the fragmentable and unfragmentable parts of the reassembled packet.

```
1  --- "a/.\\unpatched tcpip.sys"
2  +++ "b/.\\patched tcpip.sys"
3   void __fastcall Ipv6pReassembleDatagram(__int64 a1, struct_datagram
    ↪  *datagram, char a3) {
4   unfragmentableHeaderLength = datagram->unfragmentableHeaderLength;
5   packetSize = unfragmentableHeaderLength + datagram->fragmentableLength;
6   BytesNeeded = unfragmentableHeaderLength + 40;
7   v6 = *(_QWORD *)(*(_QWORD *)(a1 + 208) + 8i64);
8   v7 = *(_QWORD *)(*(_QWORD *)v6 + 40i64);
9   LockArray_high = HIDWORD(KeGetPcr()[1].LockArray);
10 -v11 = NetioAllocateAndReferenceNetBufferAndNetBufferList(IppReassembly
   ↪  NetBufferListsComplete, datagram, 0i64, 0i64, 0,
   ↪  0);
11 +if ( packetSize > 0xFFFF )
```

Listing 2: Diff of patched and vulnerable `Ipv6pReassembleDatagram`

At this stage of the vulnerability rediscovery process, the following requirements are now available:

- We have to abuse IPv6 fragmentation in `tcpip!Ipv6pReassembleDatagram`
- We have to construct a single packet with around 0xFFFF bytes of extension headers
- We have to trigger a null dereference somewhere in `tcpip!Ipv6pReassembleDatagram`

The next section will give a primer into how IPv6 fragmentation works to better understand how we can fulfill the above-mentioned requirements.

0.1.3 IPv6 fragmentation primer

When the size of a packet is larger than the Maximum transmission unit (MTU) of the outbound interface, IPv6 fragmentation is used. The MTU of most standard network equipment and desktop computers is 1500 bytes. Therefore if you have an IPv6 packet that is larger than 1500 bytes, the packet must be fragmented. This is done by splitting the packet into a number of fragments, that each has to be decorated with the IPv6 fragment header. This header is a

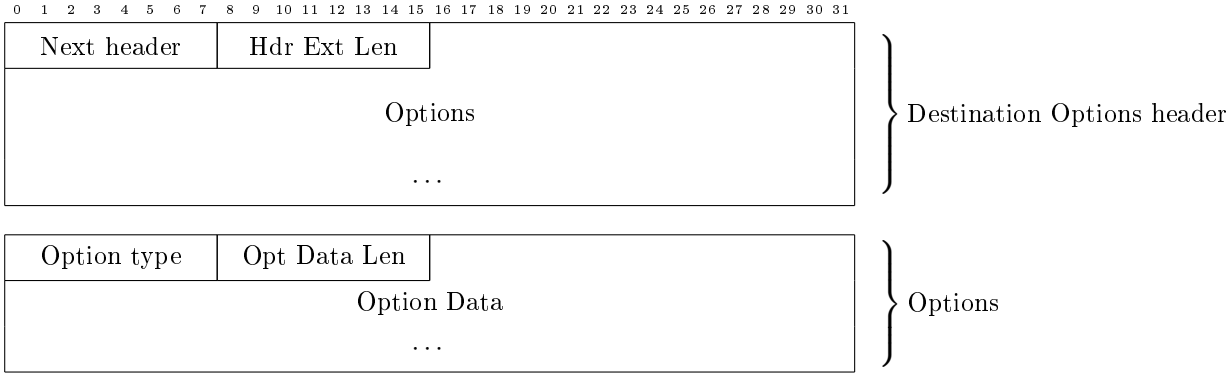
part of the specification for IPv6 Extension Headers[2, sec. 4.5]. The IPv6 Extension Headers specification specify a number of headers situated between the IPv6 header and the upper-layer header in a packet. The full list of extension headers can be seen in the following list:

1. Hop-by-Hop Options
2. *Fragment*
3. *Destination Options*
4. Routing
5. Authentication
6. Encapsulating Security Payload

As mentioned in section 0.1.1, the vulnerability is triggered when around 0xFFFF bytes of extension headers are present in the packet. Therefore, the following sections will describe both the *Destination Options* and *Fragment* extension headers in enough detail to support the exploitation of CVE-2021-24086.

IPv6 Destination Options extension header

IPv6 Destination Options are a way of defining options that should be handled by the destination node. In our case this would be the device that we are trying to attack using CVE-2021-24086. The specification can be seen on Figure 2 (IPv6 Destination Options Header [2, sec. 4.6]). The header is essentially structured as a list of options, where it is up to the receiver of a packet to support certain options.



Where

Next Header is an 8-bit selector identifying the initial header type of the Fragmentable part of the original packet.

Hdr Ext Len is an 8-bit unsigned integer describing the length of the Destination Option header in 8-octets units excluding the first 8 octets

Options is a variable-length field. See below

And

Option Type is an 8-bit identifier of the option type

Opt Data Len is an 8-bit unsigned integer describing the length of the *Data Option* field in octets

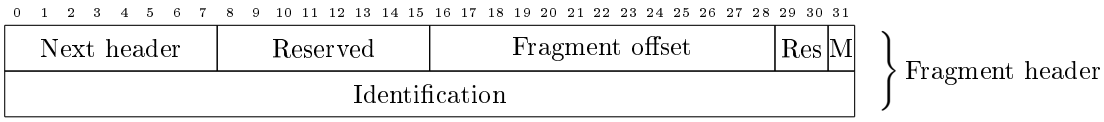
Options is a variable-length field with data specified by the option type

Figure 2: IPv6 Destination Options Header [2, sec. 4.6]

By default, only one option exist, the *PadN option*[2, sec. 4.2] which is used to create padding between two options. While this may not seem overly exciting, it is a very important part of how we can exploit CVE-2021-24086. Most other extension headers contain data that must be valid, such as routing options, which makes it hard to create a valid packet with around 0xFFFF bytes of extension headers. Destination Options does not have this limitation, as we can simply fill it with an arbitrary number of *PadN* options.

IPv6 Fragment extension header

Moving on to the IPv6 Fragment extension header, which, as mentioned earlier, is a header placed when you split an IPv6 packet into smaller fragments. IPv6 fragments are mostly used to send packets larger than the configured MTU, on either the sender or receiver side. The specification is detailed on figure Figure 3 (IPv6 Fragment Header [2, sec. 4.5]). The header contains an offset that points to where the fragment data fits into the entire packet.



Where

Next Header is an 8-bit selector identifying the initial header type of the Fragmentable part of the original packet.

Reserved is an 8-bit reserved field. Initialized to zero.

Fragment Offset is a 13-bit unsigned integer stating the offset.

Res is a 2-bit reserved field that is initialized to zero by the transmitter and ignored by the receiver.

M flag is a 1-bit boolean field describing if this is the last fragment. 1 = more fragments, 0 = last fragment.

Identificiation is a 32-bit identifier that is unique to fragments from the same package.

Figure 3: IPv6 Fragment Header [2, sec. 4.5]

Every packet that is fragmented has an unique identification, as specified in Figure 3 (IPv6 Fragment Header [2, sec. 4.5]). According to the specification[2, sec. 4.5], this identification must be different than any other fragmented packet sent recently[1].

A packet destined to be fragmented goes through two different processes, fragmentation and reassembly. Fragmentation happens on the sender side whereas reassembly is handled by the recipient of the packet.

[1] Recently is very loosely defined by RFC 8200[2] as the *"maximum likely lifetime of a packet, including transit time from source to destination and time spent awaiting reassembly with other fragments of the same packet."*[2, sec. 4.5]

Fragmentation is done by the sender and is a fairly simple concept. Looking at figure Figure 4 (IPv6 fragmentation[1]), it can be seen that an IPv6 packet contains two parts, an unfragmentable and a fragmentable part. The unfragmentable part is the IPv6 headers and the following two IPv6 extension headers, as they are processed by nodes en route:

- Hop-by-Hop Options Headers
- Routing Header

The rest of the IPv6 packet, including the Destination Options header, is handled as a fragmentable part.

Reassembly Reassembling the fragmented packet is done by the receiver and is essentially the fragmentation process in reverse. So here the receiver will convert a number of fragments into a single packet that can be handled as a standard IPv6 packet. The split of a fragmented packet can be seen on figure Figure 4 (IPv6 fragmentation[1]). Here it is easy to see that every fragment contains the unfragmentable part before any fragmented data.

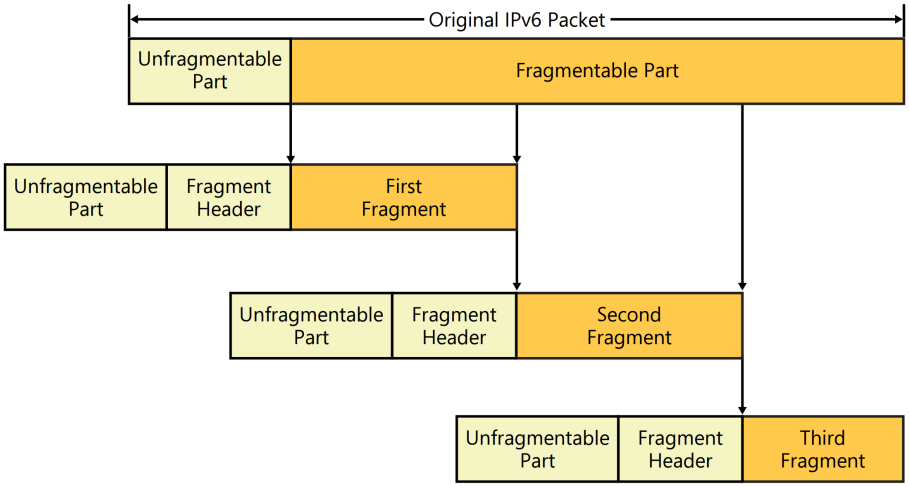


Figure 4: IPv6 fragmentation[1]

0.1.4 Root-cause analysis

At this point in the analysis the following relevant information has been presented to the reader:

1. The vulnerability happens when `tcpip.sys` reassembles a fragmented packet
2. The root cause of the vulnerability is a NULL pointer dereference in `Ip⌋ v6ReassembleDatagram` of `tcpip.sys`

3. The packet should contain around 0xFFFF bytes of extension headers
4. Extension headers can be present both in the unfragmentable and the unfragmentable part of the packet
5. The MTU limits how many bytes the unfragmentable part of the packet can contain
6. The Destination Options extension header is a good candidate for reaching 0xFFFF bytes
7. The Fragment extension header is needed to fragment the packet

To understand the root-cause of CVE-2021-24086 we must first understand how the fragmentable and unfragmentable data of the fragmented packet is handled in `Ipv6pReceiveFragment` and `Ipv6ReassembleDatagram`. If we start with `Ipv6pReceiveFragment`, we can see that a packet is reassembled when the total length of all fragment matches the expected length of the packet:

```
1  RtlCopyMdlToBuffer(netBuffer->MdlChain, netBuffer->DataOffset, v55,
   ↪  netBuffer->DataLength, &v53);
2  IppReassemblyInsertFragment(datagram, ippReassemblyLocation, NewIrql);
3  IppIncreaseReassemblySize((struct_a1 *)(Blink + 20304), datagram,
   ↪  netBuffer->DataLength + 256, netBuffer->DataLength);
4
5  if ( datagram->dataLength == datagram->fragmentableLength ) {
6      Ipv6pReassembleDatagram(a1, datagram, v21);
7  }
8  else {
9      IppCheckReassemblyQuota((PKSPIN_LOCK)(Blink + 20304));
10 }
```

Listing 3: `Ipv6pReceiveFragment` packet reassembly logic

The check can be seen on line *(5)* of listing 3 where line *(6)* shows the call to `Ipv6ReassembleDatagram`. Once inside `Ipv6pReceiveFragment` we can see that both the unfragmentable and fragmentable lengths are saved to local variables as seen on listing 4

```
1  void __fastcall Ipv6pReassembleDatagram(__int64 a1, struct_datagram
   ↪  *datagram, char a3)
2  {
3      int unfragmentableHeaderLength; // er9
4      ulong BytesNeeded; // [rsp+A8h] [rbp+10h]
5      int length; // [rsp+B8h] [rbp+20h]
6
7      ...
8
9      unfragmentableHeaderLength = datagram->unfragmentableHeaderLength;
10     length = unfragmentableHeaderLength + datagram->fragmentableLength;
11     BytesNeeded = unfragmentableHeaderLength + 40;
12
13     ...
14 }
```

Listing 4: `Ipv6pReassembleDatagram` length calculation

It's also important to notice the `BytesNeeded` variable which is equal to the size of unfragmentable header and the size of the Ipv6 header which is 40 bytes as seen on line *(11)*. To understand the root cause, it is important to understand what will happen if the unfragmentable part of the header contains around 0xFFFF bytes. The calculation of `BytesNeeded` on line *11* also shows why it is only necessary to have *around* 0xFFFF bytes in the unfragmentable part.

Tracking down where `BytesNeeded` is used leads us to the code found in listing 5. This listing contains the code for obtaining a buffer to store the data for the unfragmentable part of the header. As it can be seen on line *(9)* and *19*, this is where the `BytesNeeded` variable is used.

```
1  NetBufferList = (_NET_BUFFER_LIST *)NetioAllocateAndReferenceNetBufferA⌋
   ↪  ndNetBufferList(IppReassemblyNetBufferListsComplete, datagram,
   ↪  0i64, 0i64, 0, 0);
2  if ( !NetBufferList )
3  {
4      ...
5      goto failure;
6  }
7
8  netBuffer = NetBufferList->FirstNetBuffer;
9  if ( NetioRetreatNetBuffer(netBuffer, (unsigned __int16)BytesNeeded, 0)
   ↪  < 0 )
10 {
11     IppRemoveFromReassemblySet((PKSPIN_LOCK)(v7 + 20304),
       ↪  (__int64)datagram, a3);
12     NetioDereferenceNetBufferList(NetBufferList, 0i64);
13
14     ...
15
16     goto memory_failure;
17 }
18
19 buffer = NdisGetDataBuffer(netBuffer, BytesNeeded, 0i64, 1u, 0);
```

Listing 5: `Ipv6pReassembleDatagram` NetBuffer null reference logic

The logic for listing 5 can be explained as such:

1. The NetBufferList is retrieved by `NetioAllocateAndReferenceNetBufferA⌋ ndNetBufferList` and checked for validity
2. The first `NetBuffer` is retrieved using `NetioRetreatNetBuffer`
    - Notice the cast to a unsigned 16 bit integer on line *(9)* wich will truncate the `BytesNeeded`.
3. `NdisGetDataBuffer` is used to retrieve a buffer.
    - Notice that `BytesNeeded` is *not* truncated in this call on line *10*.

Now the question is, what happens when `NetioRetreatNetBuffer` is invoked with a smaller value than `NdisGetDataBuffer`? The answer to that question is that `NdisGetDataBuffer` returns null. Later on in the function this buffer, which is null, is written to which will demonstrate that this indeed is a null pointer dereference. At this point we are presented with the root cause of the vulnerability, and can therefore move on to the process of triggering the vulnerability by sending a packet with about 0xFFFF extension headers in the unfragmentable part of the packet.

0.1.5 Triggering the vulnerability