Under the Hood of AFD.sys Part 1: Investigating Undocumented Interfaces

Mateusz Lewczak ⋮ ⋮ 16/07/2025

A quick look at how I used WinDbg and NtCreateFile to craft a raw TCP socket via AFD.sys on Windows 11, completely skipping Winsock.

Posted Jul 16, 2025
By *Mateusz Lewczak*

18 min read

Under the Hood of AFD.sys Part 1: Investigating Undocumented Interfaces

Introduction

This is the first post in a series about my deep-dive into the `AFD.sys` driver on Windows 11. The idea is that both this write-up and the library that comes out of it will be a one-stop doc set - and a launchpad - for poking at other drivers that don't ship with an official spec.

On Windows, the go-to (and easiest) way to do network stuff is Winsock. It gives you a bunch of high-level calls for TCP/UDP and raw sockets over IPv4/IPv6. Under the hood Winsock rides on `mswsock.dll`, which is lower-level, but most apps never need to touch that because Winsock already covers 99 % of everyday networking needs.

In this first part we're focusing purely on creating the socket itself. Step #1 is to open a TCP socket to any host on the LAN using nothing but I/O requests aimed at `\Device\Afd`. Instead of the usual Winsock calls (or anything in `mswsock.dll`) we're going to slam everything through `NtDeviceIoControlFile`, hand-crafting the IRPs (I/O Request Packets) the AFD driver expects. That'll show us, in real life, how to build the call sequence, buffer layouts, and flags you need to spin up a TCP session.

The actual data exchange over that socket - the whole TCP conversation - will come in later posts.

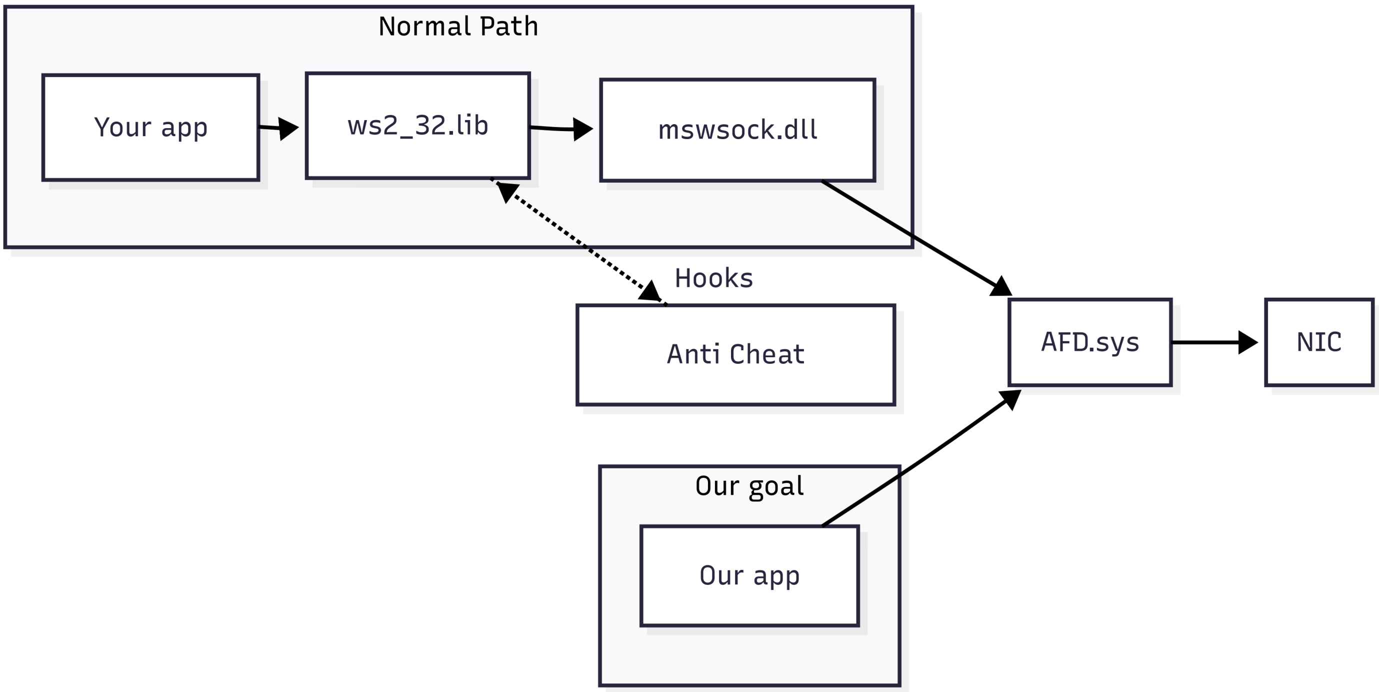
Right now I've already collected all the data to pull off the TCP three-way handshake. Took me a few evenings to get there, so I'm just jotting down what I did so far. I'll keep adding the rest as I go - at least that's the plan!

What is AFD.sys?

The **AFD.sys** - or Ancillary Function Driver - is a small but absolutely basic Windows kernel driver. It sits in C:Windows32drivers and starts up with the system, because it's the one that translates the Winsock calls of your applications (send, recv, connect…) into lower-layer intelligible `IRP` (*I/O request packet*), which tcpip.sys and co. are already taking over. If it were missing, the browser, Spotify or remote desktop wouldn't see the network - all TCP/UDP traffic would simply stop.

Rationale

The first reason for talking directly to `AFD.sys` instead of going through Winsock is to dodge the hooks used by some protection systems - like anti-cheat or anti-malware (though the latter usually rely on NDIS filters in kernel mode). A lot of these protections work by intercepting and modifying calls to functions exported by `Ws2_32.lib` - usually by injecting their own DLLs or patching stuff directly in process memory. But if you're not using Winsock, those hooks have nothing to latch onto, which makes their job way harder from a technical standpoint.



The second reason - and honestly the one that matters most to me - is the educational value. Working directly with `AFD.sys` gives you a deep look under the hood of how Windows handles networking. That kind of insight just isn't possible when you stick to high-level APIs.

The goal of this whole project is to build a library for talking directly to the `AFD.sys` driver on Windows 11, completely skipping the Winsock layer. The core will be written in C/C++ and will include all the low-level logic for building and sending IRPs. On top of that, I'm planning to add clean, easy-to-use bindings for Python - great for quick prototyping or scripting - and also for Rust.

Dumb Copy&Paste

The very first thing we have to nail down is a socket the driver will actually accept, so we can start talking on the wire. While combing the internet I ran into a PoC for CVE-2024-38193 (killvxk). That was the first real bit of code that spat out a socket for me:

```
NTSTATUS AfdCreate(PHANDLE Handle, ULONG EndpointFlags)
{
    UNICODE_STRING DevName;
```

```
    RtlInitUnicodeString(&DevName, L"\\Device\\Afd\\Endpoint");
    const wchar_t* transportName = L"\\Device\\Tcp";

    BYTE bExtendedAttributes[] = {...};

    OBJECT_ATTRIBUTES Object;
    Object = { 0 };
    Object.ObjectName = &DevName;
    Object.Length = 48;
    Object.Attributes = 0x40;

    IO_STATUS_BLOCK IoStatusBlock;

    return NtCreateFile(Handle, 0xC0140000, &Object, &IoStatusBlock, 0, 0, 3,
FILE_OPEN_IF, 0x20, &bExtendedAttributes, sizeof(bExtendedAttributes));
}
```

Right away I learned that what AFD calls a "socket" is really just a `HANDLE`. With the rest of that PoC I could bind the socket, but I still couldn't connect. So the hunt continued - was my `_EXTENDED_ATTRIBUTES` struct busted? Or was the problem somewhere else?

Next stop: a thread on the UnKoWnCheaTs blog (unknowncheats.me ICoded post). It's basically only code, no explanation, so I copied the snippet and tried to run it like this:

```
int main() {
        HANDLE socket;
        NTSTATUS status = AfdCreate(&socket, AF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (!NT_SUCCESS(status)) {
                std::cout << "[-] Could not create socket: " << std::hex << status << std::endl;
                return 1;
        }
        std::cout << "[+] Socket created!" << std::endl;

        sockaddr_in server = { AF_INET, htons(27015), {inet_addr("127.0.0.1")}, {0} };
        status = AfdBind(socket, &server);
        if (!NT_SUCCESS(status)) {
                std::cout << "[-] Could not bind: " << std::hex << status << std::endl;
                return 1;
        }
        std::cout << "[+] Socket bound!" << std::endl;

        status = AfdDoConnect(socket, &server);
        if (!NT_SUCCESS(status)) {
                std::cout << "[-] Could not connect: " << std::hex << status << std::endl;
                return 1;
        }
        std::cout << "[+] Connected!" << std::endl;
}
```

That time the socket came to life again, but `bind` flat-out failed. So I went spelunking for reversed structure definitions in publicly available code. I ran into plenty of candidates - ReactOS (ReactOS Project), Dr. Memory's AFD bits (DynamoRIO / Dr. Memory), even an old issue thread (Dr. Memory - GH issue#376). None of them truly pieced the puzzle together, so I was still stuck at `bind`.

Why's it blowing up? A few theories:

1. Different Windows builds and `AFD.sys` versions might expect slightly different structures.

2. Flags in the CVE-2024-38193 PoC are tuned for exploitation, not for my vanilla use case - so they're probably wrong here.
3. Insert literally any other reason…

Kernel Debugging Time

At this point I realized that blindly copy-pasting other people's code wasn't going to cut it - I needed to do a few experiments with WinDbg. So I spun up a Windows 11 VM and started grabbing calls that hit `AFD.sys`. The plan:

1. Find some code that makes legit requests to `AFD.sys`.
2. Capture the I/O-request buffers that code sends.
3. Re-create those buffers on my host and see if the driver is happy.
4. Reverse-engineer the structs so we actually know what each field is and which values make sense.

Side note: I'm skipping the whole "turn on kernel debugging, set up the connection" dance. Microsoft's docs and half the internet explain that step-by-step.

What's the fastest way to make a process fire off valid `AFD.sys` requests? Write a dead-simple TCP client with Winsock:

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iostream>
#pragma comment(lib,"Ws2_32.lib")

int main() {
    std::cout << "PID: " << GetCurrentProcessId() << "\nPress <Enter> to continue..."
<< std::endl;
    std::cin.get();

    WSADATA wsa;
    if (WSAStartup(MAKEWORD(2, 2), &wsa)) return 1;

    SOCKET s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (s == INVALID_SOCKET) return 1;

    sockaddr_in addr{};
    addr.sin_family = AF_INET;
    addr.sin_port = htons(80);
    InetPtonA(AF_INET, "192.168.1.1", &addr.sin_addr);

    if (connect(s, reinterpret_cast<sockaddr*>(&addr), sizeof(addr)) == SOCKET_ERROR)
{
        std::cerr << "connect error: " << WSAGetLastError() << '\n';
    } else {
        std::cout << "Connected\n";
    }

    closesocket(s);
    WSACleanup();
    return 0;
}
```

We know the very first thing Winsock does is create a socket by opening a `HANDLE` to `\Device\Afd`. So our next task is to break on `nt!NtCreateFile`. You might wonder why I print the PID and then pause - if I simply slapped a breakpoint on `NtCreateFile`, I'd hit **every** call system-wide, which is useless. I only want the calls from *this* process.

Now what's left is to run this program and set the appropriate breakpoint - of course NtCreateFile isn't just used for driver communication, so you'll have to click around a few times until you find something like `NtCreateFile("Device)`. It's probably possible to do this as an automation in WinDbg, but I don't know how - skill issue.

A więc pokolei zaczynamy działać w WinDbg:

1. Set a process-specific breakpoint on `nt!NtCreateFile`:

```
.foreach /pS 1 (ep { !process 0 0 afd_re.exe }) { bp /p ${ep} nt!NtCreateFile }
```

2. Dump the 3rd arg (Microsoft) (register r8 on x64 / Microsoft ABI (Microsoft)) as an `_OBJECT_ATTRIBUTES`.:

```
10: kd> dt nt!_OBJECT_ATTRIBUTES @r8
Breakpoint 2 hit
+0x000 Length           : 0x30
+0x008 RootDirectory    : (null)
+0x010 ObjectName       : 0x00000018`06f1f5b0 _UNICODE_STRING
"\Device\Afd\Endpoint"
+0x018 Attributes       : 0x42
+0x020 SecurityDescriptor : (null)
+0x028 SecurityQualityOfService : (null)
```

3. If `ObjectName` shows `\Device\Afd...`, bingo. Otherwise go and wait for the next hit.
4. The last two `NtCreateFile` args live on the stack. Through trial and error I found they sit at `rsp+0x50`:.

```
4: kd> dq @rsp+50 L2
fffffc04`df00f438  00000018`06f1f5c0 00000000`00000039
```

5. What we can see here is the address of the `EXTENDED_ATTRIUTES` buffer (i.e. the extra data we pass to the file/driver when creating the `HANDLE`) and its size. It is consecutively `0x1806f1f5c0` and `0x39`.
6. What is important! The address of this buffer is the address of the memory page in the context of the user process that triggered this system call - we are currently in kernel-space. So before we can start reading it, we still need to switch to that process.

```
.process /r /p @$proc
```

7. Read those `0x39` bytes:

```
4: kd> db 1806f1f5c0 L39
00000018`06f1f5c0  00 00 00 00 00 0f 1e 00-41 66 64 4f 70 65 6e 50
........AfdOpenP
00000018`06f1f5d0  61 63 6b 65 74 58 58 00-00 00 00 00 00 00 00 00
acketXX.........
00000018`06f1f5e0  02 00 00 00 01 00 00 00-06 00 00 00 00 00 00 00
................
00000018`06f1f5f0  18 ba 5a 4a 33 01 00 00-64                        ..ZJ3...d
```

8. What have we learned so far? And what is useful to us?

   1. the Winsock (or rather `mswsock.dll`) opens a handle to the `\Device\Afd\Endpoint` driver.
   2. the expected structure is `0x39` bytes in length.
9. We are left to convert this set of bytes into code in C++:

```
NTSTATUS AfdCreate(PHANDLE handle) {
 UNICODE_STRING devName;
 RtlInitUnicodeString(&devName, L"\\Device\\Afd\\Endpoint");

 BYTE bExtendedAttributes[] = {
     0x00, 0x00, 0x00, 0x00, 0x00, 0x0F, 0x1e, 0x00,
     0x41, 0x66, 0x64, 0x4F, 0x70, 0x65, 0x6E, 0x50,
     0x61, 0x63, 0x6B, 0x65, 0x74, 0x58, 0x58, 0x00,
     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
     0x02, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
     0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
     0x18, 0xba, 0x5a, 0x4a, 0x33, 0x01, 0x00, 0x00,
     0x64
 };

 OBJECT_ATTRIBUTES Object;
 Object = { 0 };
 Object.ObjectName = &devName;
 Object.Length = 48;
 Object.Attributes = 0x40;

 IO_STATUS_BLOCK IoStatusBlock;
 return NtCreateFile(handle, GENERIC_READ | GENERIC_WRITE | SYNCHRONIZE, &Object,
&IoStatusBlock, 0, 0, FILE_SHARE_READ | FILE_SHARE_WRITE, FILE_OPEN_IF, 0x20,
&bExtendedAttributes, sizeof(bExtendedAttributes));
}
```

Analyzing retrieved data

After executing this code, we get information that our `HANDLE` (i.e. socket in practice) has been successfully created. Now gathering data from publicly available code, we can reconstruct the contents of our workingly named `AFD_OPEN_PACKET_EA` structure.

I used the previously mentioned sources and (DeDf) to recreate the structure. Let’s first try to label specific portions of bytes for ourselves, and then we will create a `struct` from this:

```
BYTE bExtendedAttributes[] = {
    0x00, 0x00, 0x00, 0x00, // NextEntryOffset - 4 bytes
    0x00,                   // Flags           - 1 byte
    0x0F,                   // EaNameLength    - 1 byte
    0x1e, 0x00,             // EaValueLength   - 2 bytes
    // START AfdOpenPacketXX 0xf bytes of name + leading zero
    0x41, 0x66, 0x64, 0x4F, 0x70, 0x65, 0x6E, 0x50,
    0x61, 0x63, 0x6B, 0x65, 0x74, 0x58, 0x58, 0x00,
    // END AfdOpenPacketXX
    0x00, 0x00, 0x00, 0x00, // EndpointFlags = 0
    0x00, 0x00, 0x00, 0x00, // GroupID = 0
    0x02, 0x00, 0x00, 0x00, // AddressFamily = AF_INET
    0x01, 0x00, 0x00, 0x00, // SocketType = SOCK_STREAM
    0x06, 0x00, 0x00, 0x00, // Protocol = IPPROTO_TCP
    0x00, 0x00, 0x00, 0x00, // SizeOfTransportName

    // unknown 9 bytes
    0x18, 0xba, 0x5a, 0x4a, 0x33, 0x01, 0x00, 0x00, 0x64
};
```

So what do we have? What do we know?

1. `NextEntryOffset` - this is the offset where the next entry for `EXTENDED_ATTRIBUTES` is located. Possibly a typical field for I/O, in our case none so we have zeros.
2. `Flags` - these are some flags for our `EXTENDED_ATTRIBUTE` structure, in this case it is zero. Unknown at this point.
3. `EaNameLength` - the length of the name of our `EXTENDED_ATTRIBUTE`, which in this case is 15 bytes.
4. `EaValueLength` - a size expressed in bytes representing the size of some internal structure. This structure will be `EndpointFlags` to the end, along with unknown bytes.
5. `EndpointFlags` - more flags, but probably already relating to our sockets. Following (killvxk) we can use the enum available there. After reproducing the identical steps, but for UDP communication and the field value is `0x11`. Which would mean `AFD_ENDPOINT_FLAG_CONNECTIONLESS | AFD_ENDPOINT_FLAG_MESSAGEMODE`.

```
// 4 bytes
enum __bitmask AFD_ENDPOINT_FLAGS {
 AFD_ENDPOINT_FLAG_CONNECTIONLESS = 0x000000000001,
 AFD_ENDPOINT_FLAG_MESSAGEMODE    = 0x000000000010,
 AFD_ENDPOINT_FLAG_RAW            = 0x000000001000,
 AFD_ENDPOINT_FLAG_MULTIPOINT     = 0x000000010000,
 AFD_ENDPOINT_FLAG_CROOT          = 0x000001000000,
 AFD_ENDPOINT_FLAG_DROOT          = 0x000010000000,
 AFD_ENDPOINT_FLAG_IGNORETDI      = 0x001000000000,
 AFD_ENDPOINT_FLAG_RIOSOCKET      = 0x010000000000,
};
```

6. `GroupID` - the identifier of the socket group (Microsoft), looks like some legacy of the old fiches.
7. `AddressFamily`, `SocketType`, `Protocol` - these are standard fields describing our address family, socket type and protocol used.
8. `SizeOfTransportName` - in some instances of sockets creation I have seen authors refer to `DeviceAfd` in addition to referring to `DeviceTcp` and similar drivers. The length of this string should be specified here, whereas during debugging, not once did I see this field actually filled in.
9. `unknown 9 bytes` - this is nowhere to be found, I have not come across it anywhere before. By trial and error I figured out that the last two bytes are optional. Without any problem `AFD.sys` will accept such a buffer as well. And even more interestingly, they can take any value, this is also a valid `EXTENDED_ATTRIBUTE`.

```
BYTE bExtendedAttributes[] = {
 [SAME VALUES]
 // unknown 9 bytes, but only 7 provided
 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff
};
```

Staying with our unknown bytes, below I have examples for a few more calls of our code:

```
c8 27 ff 09 16 02 00 00 64
98 b8 85 4a a3 02 00 00 64
```

In this case, a static analysis of `mswsock.dll` would need to be carried out to better understand what they might be.

Reverseing mswsock.dll

I used Binary Ninja (free, v5.0.7) to do the reverse engineering. I started by finding a function that uses `NtCreateFile`, I found 5 functions in total and one of them is `SockSocket`:

```
uint64_t SockSocket(int32_t arg1, int32_t arg2, int32_t arg3, int1
                        AllocationSize = 0;
                        NTSTATUS rax_40 = NtCreateFile(&FileHandle,
                            DesiredAccess, &ObjectAttributes,
                            &var_f8, AllocationSize,
                            SECURITY_ANONYMOUS, var_1b8_1,
                            var_1b0_1, var_1a8_1,
                            SocketGlobalLock_7,
                            (uint32_t)rbx_3 + 0x1b);
```

At this point we know that the penultimate argument of the `NtCreateFile` call is our `AFD_OPEN_PACKET_EA` structure, and the last argument is the length of that structure. So it's worth naming them now. And additionally create a custom structure in Binary Ninja, then the analyser will interpret the operation on our structure correctly.

Change Type

Types (C syntax):

```
struct AFD_OPEN_PACKET_EA __packed
{
    uint32_t NextEntryOffset;
    uint8_t Flags;
    uint8_t EaNameLength;
    uint16_t EaValueLength;
    char EaName[0x10];
    uint32_t EndpointFlags;
    uint32_t GroupID;
    uint32_t AddressFamily;
    uint32_t SocketType;
    uint32_t Protocol;
    uint32_t SizeOfTransportName;
    uint8_t UnknownBytes[0x9];
};
```

With this, Binary Ninja generated us this Pseudo C code, which looks promising:

```
    EXTENDED_ATTRIBUTE = &var_a8;
    label_18000c6d5:
    EXTENDED_ATTRIBUTE->NextEntryOffset = 0;
    EXTENDED_ATTRIBUTE->Flags = 0;
    EXTENDED_ATTRIBUTE->EaNameLength = 0xf;
    __builtin_strcpy(&EXTENDED_ATTRIBUTE->EaName,
        "AfdOpenPacketXX");
    EXTENDED_ATTRIBUTE->EaValueLength =
        EXTENDED_ATTRIBUTE_INNER_LEN;
    EXTENDED_ATTRIBUTE->SizeOfTransportName =
        (uint32_t)TransportName.Length;

    if (!(*(uint8_t*)(BaseAddress + 0x45) & 0x10))
        memcpy(&EXTENDED_ATTRIBUTE->UnknownBytes,
            TransportName.Buffer,
            (uint64_t)TransportName.Length + 2);

    EXTENDED_ATTRIBUTE->AddressFamily = SocketGlobalLock_5;
    EXTENDED_ATTRIBUTE->SocketType = SocketType;
    EXTENDED_ATTRIBUTE->Protocol = SocketGlobalLock_4;
    EXTENDED_ATTRIBUTE->EndpointFlags = 0;
```

I also messed around with other variables that can be inferred from the context of the code such as `TransportName` etc. It remained to check where the `SockSocket` function refers to our unknown bytes. To my surprise there is only one place. The `mswsock.dll` library only operates on them when it copies `TransportName` and in no other place. So either actually these bytes don't matter much and are just added random values when not using `TransportName` or another function operates on them.

What do our sources say about this? Unfortunately I don't see any information on this, and it looks like at least seven of those odd five bytes are required for `AFD.sys` to accept a request from us to create a new sockets. I did, however, find information about what happens when we specify a `TransportName` and when we don't specify it (diversenok). But this unfortunately does not answer our question. So this is something new that we discovered during our research! On the positive side, this leaves us room for further exploration. I think we can leave it for now and possibly come back to it later when it is needed. After all we correctly managed to create a TCP socket.

What is TDI?

It's worth going one level down from AFD.sys for a moment, because underneath lies its true interface to the **TCP/IP stack - the Transport Driver Interface (TDI)** as TDI will appear in many places in later parts of our series. TDI is the "upper edge" of the transport layer in the Windows kernel - an abstraction that, back in the days of NT 3.51, unified communication with various protocols (TCP/IP, NetBIOS, AppleTalk). From a kernel-mode point of view, there are two entities:

- **Transport Provider** - the driver of the protocol itself, e.g. `\Device\Tcp`.
- **TDI Client** - anyone who sends IRPs to it with codes `TDI_SEND`, `TDI_RECEIVE`, `TDI_CONNECT`, etc.

The AFD acts as an intermediary-client: it receives our IOCTLs from user space and then ‘builds’ the corresponding IRPs (`TdiBuildSend`, `TdiBuildReceive` macros) and passes them to the transport driver. For example, if we had specified `TransportName` in our `EXTENDED_ATTRIBUTES` we would have had to communicate with `AFD.sys` given the TDI structures. Instead of `SOCKADDR` it would be `TransportAddress`.

Next steps

In the next part of this series we will focus on trying to set up a TCP handshake with localhost on port 80. For this we will use `AfdBind` and `AfdConnect`, functions provided by `AFD.sys` available as an I/O request.

Final code

Below you can find the full code that creates a socket without using any networking library.

```
#include <stdint.h>
#include <Windows.h>
#include <winternl.h>
#include <iostream>
#pragma comment(lib, "ntdll.lib")

enum AFD_ENDPOINT_FLAGS : uint32_t {
    AFD_ENDPOINT_FLAG_CONNECTIONLESS  = 0x000000000001,
    AFD_ENDPOINT_FLAG_MESSAGEMODE     = 0x000000000010,
    AFD_ENDPOINT_FLAG_RAW             = 0x000000001000,
    AFD_ENDPOINT_FLAG_MULTIPOINT      = 0x000000010000,
    AFD_ENDPOINT_FLAG_CROOT           = 0x000001000000,
    AFD_ENDPOINT_FLAG_DROOT           = 0x000010000000,
    AFD_ENDPOINT_FLAG_IGNORETDI       = 0x001000000000,
    AFD_ENDPOINT_FLAG_RIOSOCKET       = 0x010000000000,
};

struct AFD_OPEN_PACKET_EA {
    uint32_t nextEntryOffset;
    uint8_t  flags;
    uint8_t  eaNameLength;
    uint16_t eaValueLength;
    char     eaName[0x10];
    uint32_t endpointFlags;
    uint32_t groupID;
    uint32_t addressFamily;
    uint32_t socketType;
    uint32_t protocol;
    uint32_t sizeOfTransportName;
    uint8_t  unknownBytes[0x9];
};

NTSTATUS createAfdSocket(PHANDLE socket) {
    const char* eaName = "AfdOpenPacketXX";
    UNICODE_STRING devName;
    RtlInitUnicodeString(&devName, L"\\Device\\Afd\\Endpoint");

```

```
    OBJECT_ATTRIBUTES object;
    object = { 0 };
    object.ObjectName = &devName;
    object.Length = 48;
    object.Attributes = 0x40;

    AFD_OPEN_PACKET_EA afdOpenPacketEA;
    afdOpenPacketEA.nextEntryOffset  = 0x00;
    afdOpenPacketEA.flags            = 0x00;
    afdOpenPacketEA.eaNameLength     = 0x0F;
    afdOpenPacketEA.eaValueLength    = 0x1e;
    afdOpenPacketEA.endpointFlags    = 0x00;
    afdOpenPacketEA.groupID          = 0x00;
    afdOpenPacketEA.addressFamily    = AF_INET;
    afdOpenPacketEA.socketType       = SOCK_STREAM;
    afdOpenPacketEA.protocol         = IPPROTO_TCP;
    afdOpenPacketEA.sizeOfTransportName = 0x00;
    memset(afdOpenPacketEA.eaName, 0x00, 0x10);
    memcpy(afdOpenPacketEA.eaName, eaName, 0x10);
    memset(afdOpenPacketEA.unknownBytes, 0xFF, 0x9);

    IO_STATUS_BLOCK IoStatusBlock;
    return NtCreateFile(socket, GENERIC_READ | GENERIC_WRITE | SYNCHRONIZE, &object,
                        &IoStatusBlock, 0, 0, FILE_SHARE_READ | FILE_SHARE_WRITE,
FILE_OPEN_IF,
                        FILE_SYNCHRONOUS_IO_NONALERT, &afdOpenPacketEA,
sizeof(afdOpenPacketEA));
}

int main() {
    HANDLE socket;
    NTSTATUS status = createAfdSocket(&socket);
    if (!NT_SUCCESS(status)) {
        std::cout << "[-] Could not create socket: " << std::hex << status <<
std::endl;
        return 1;
    }
    std::cout << "[+] Socket created!" << std::endl;

        return 0;
}
```

References

1. Vittitoe, Steven. “Reverse Engineering Windows AFD.sys: Uncovering the Intricacies of the Ancillary Function Driver.” *Proceedings of REcon 2015*, 2015, https://doi.org/10.5446/32819.
2. killvxk. *CVE-2024-38193 Nephster PoC*. 2024, https://github.com/killvxk/CVE-2024-38193-Nephster/blob/main/Poc/poc.h.
3. unknowncheats.me ICoded post. *Native TCP Client Socket*. n.d., https://www.unknowncheats.me/forum/c-and-c-/500413-native-tcp-client-socket.html.
4. ReactOS Project. *Afd.h*. n.d., https://github.com/reactos/reactos/blob/master/drivers/network/afd/include/afd.h.
5. DynamoRIO / Dr. Memory. *afd_sharedḣ*. n.d., https://github.com/DynamoRIO/drmemory/blob/master/wininc/afd_shared.h.
6. Dr. Memory - GH issue#376. *Issue #376: AFD Support Improvements*. n.d., https://github.com/DynamoRIO/drmemory/issues/376.
7. Microsoft. *NtCreateFile Function (Winternl.h)*. n.d., https://learn.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntcreatefile.

8. ---. *x64 Calling Convention*. n.d., https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170.
9. ---. *x64 Calling Convention*. n.d., https://learn.microsoft.com/pl-pl/windows/win32/api/winsock2/nf-winsock2-wsasocketa.
10. DeDf. *AFD Repository*. n.d., https://github.com/DeDf/afd/tree/master.
11. Allievi, Andrea, et al. *Windows® Internals Part 2 - 6th Edition*. 6th ed., Microsoft Press (Pearson Education), 2022, https://learn.microsoft.com/sysinternals/resources/windows-internals.
12. diversenok. *\Texttntafd.h – Ancillary Function Driver Definitions*. commit 2dda0dd, Hunt & Hackett, April 2025, https://github.com/winsiderss/systeminformer/blob/master/phnt/include/ntafd.h.