Under the Hood of AFD.sys Part 2: TCP handshake

Mateusz Lewczak ⋮ ⋮ 23/07/2025

A walk-through of the bind + connect IOCTLs: capturing AFD.sys IRPs with WinDbg, reverse-engineering the buffers for IPv4/IPv6, and completing a manual TCP three-way handshake on Windows 11—still zero Winsock involved.

Posted Jul 23, 2025
By *Mateusz Lewczak*

15 min read

Under the Hood of AFD.sys Part 2: TCP handshake

Plan for today

This is the second part in a series of posts concerning `AFD.sys`. If you have not seen the previous one you can find it here. Familiarity with the first part will be key to understanding the content of this post, I will not duplicate the kernel debugging steps, but will immediately show here the contents of the buffers directed to `NtDeviceIoControlFile`. In this part we will look at the `bind` and `connect` operations. Although normally when we use Winsock we don't need to perform the `bind`, underneath `mswsock.dll` actually performs this `bind` for us, so it will be crucial for us to understand how we can establish a TCP handshake.

IOCTL for bind command

So let's start with the `bind` operation. In the previous part I focused mainly on TCP, this time we will perform some operations for TCP, UDP with IPv4 and IPv6. So that we can better understand what we are dealing with and what is what. However, as before, for the reconstruction of the structures, I will rely on what can be found on the Internet (killvxk), (unknowncheats.me ICoded post), (ReactOS Project), (DynamoRIO / Dr. Memory), (Dr. Memory - GH issue#376), (DeDf).

This time I will use such code using Winsock, it might be worthwhile in the future to port these programs directly to `mswsock.dll`, but the problem is that they are documented (we have signatures of the available functions), but examples of actual use are missing.

```
1  #define WIN32_LEAN_AND_MEAN
2  #include <windows.h>
3  #include <winsock2.h>
4  #include <ws2tcpip.h>
5  #include <iostream>
6  #pragma comment(lib,"Ws2_32.lib")
7
8  void createTCPv4() {
```

```
9       SOCKET s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
10      if (s == INVALID_SOCKET) { std::cerr << WSAGetLastError() << '\n'; return; }
11
12      sockaddr_in bindAddr{};
13      bindAddr.sin_family = AF_INET;
14      bindAddr.sin_port = htons(27015);
15      bindAddr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
16      if (bind(s, reinterpret_cast<sockaddr*>(&bindAddr), sizeof(bindAddr)) ==
17  SOCKET_ERROR) {
18          std::cerr << "bind: " << WSAGetLastError() << '\n'; closesocket(s); return;
19      }
20
21      closesocket(s);
22  }
23
24  void createUDPv4() {/*SAME FOR UDPv4*/}
25  void createTCPv6() {/*SAME FOR TCPv6*/}
26  void createUDPv6() {/*SAME FOR UDPv6*/}
27
28  int main() {
29      std::cout << "PID: " << GetCurrentProcessId() << "\nPress <Enter> to continue..."
30  << std::endl;
31      std::cin.get();
32
33      WSADATA wsa;
34      if (WSAStartup(MAKEWORD(2, 2), &wsa)) return 1;
35
36      createTCPv4();
37      createUDPv4();
38      createTCPv6();
39      createUDPv6();
40
41      WSACleanup();
        return 0;
    }
```

Before we start collecting data, it is worth mentioning here that socket operations within `AFD.sys` are performed using the `NtDeviceIoControlFile` (missing reference). Among other things, the `IoControlCode` parameter is passed there, with which the operations (this is a simplification, there is much more information behind it) that we want to perform are identified. The driver then performs a dispatch based on the value of this parameter. So, in addition to the data itself being passed to `AFD.sys`, we need to collect this control code.

Exactly the same as for debugging `NtCreateFile`, here we also set the corresponding breakpoint, but on `adf!AdfBind`:

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

Now we need to read the information passed to the driver, i.e. the *I/O request packet*, known as `IRP`(missing reference), for this we can use the following command in WinDbg:

```
1  4: kd> !irp @rcx 1
2  Irp is active with 4 stacks 4 is current (= 0xffffa70db3ef9718)
3   No Mdl: No System Buffer: Thread ffffa70db39f2080:  Irp stack trace.
4  Flags = 00060000
5  ThreadListEntry.Flink = ffffa70db39f25c0
6  [...]
7  >[IRP_MJ_DEVICE_CONTROL(e), N/A(0)]
8              5  0 ffffa70dac138d40 ffffa70db718fd20 00000000-00000000
9                 \Driver\AFD
```

```
10                          Args: 00000010 00000014 0x12003 75118ff2d0
11
```

Already from this information we can learn quite a lot about the arguments of this call (TCPv4 variant):

1. `00000010` - output buffer length - is also important, if we do not send a large enough buffer then `AFD.sys` will return an error,
2. `00000014` - input buffer length,
3. `0x12003` - control code a.k.a. `IoControlCode`,
4. `75118ff2d0` - input buffer address in source process a.k.a. `Type3InputBuffer`. The subsequent steps for reading the buffer are exactly the same as in the `NtCreateFile` cases.

Let's focus for a moment on the `IoControlCode`, its value is `0x12003`, it would be nice if we had some way to build these values depending on the function we need. And here a very good source for us could be (diversenok). The data we obtained actually matches what we were able to get:

```
  ...
1 #define AFD_BIND                        0
2 ...
3 #define FSCTL_AFD_BASE  FILE_DEVICE_NETWORK
4 #define _AFD_CONTROL_CODE(Request, Method) (FSCTL_AFD_BASE << 12 | (Request) << 2 |
5 (Method))
6 ...
7 #define IOCTL_AFD_BIND                          _AFD_CONTROL_CODE(AFD_BIND,
  METHOD_NEITHER) // 0x12003
```

So you we confidently use these definitions to build our tool. Or at least for now, because you never know.

Collected data

During the data collection, I considered a total of eight cases. All in order to best be able to distinguish specific pieces of data and their roles in the overall bind process. The first table shows the implicit `bind` that mswsock performs at connect if we have not previously performed a `bind`. The second one with explicit bind, where I chose `127.0.0.1` as the source address for variants with IPv4 and `::1` for variants with IPv6. In both cases I additionally selected the port `27015`.

Implicit bind()

Variant	Input buffer (hex)	Input length (hex)
TCP v4	`02 00 00 00 02 00 00 00 00 00 00 00 FF FF FF FF FF FF FF FF`	`0x14`
UDP v4	`02 00 00 00 02 00 00 00 00 00 00 00 F3 03 00 00 00 00 00 00`	`0x14`
TCP v6	`02 00 00 00 17 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`	`0x20`
UDP v6 *(identical to TCP v6)*		`0x20`

Explicit bind(loopback, 27015)

Variant	Input buffer (hex)	Input length (hex)
TCP v4	`00 00 00 00 02 00 69 87 7F 00 00 01 00 00 00 00 00 00 00 00`	`0x14`
UDP v4 *(identical to TCP v4)*		`0x14`
TCP v6	`00 00 00 00 17 00 69 87 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00`	`0x20`
UDP v6 *(identical to TCP v6)*		`0x20`

Analyzing retrieved data

TCPv4 and UDPv4

Let's focus for now on all the TCP protocol variants and try to deduce what the field is based on what we see on the collected sources. The explicit bind for IPv4 can tell us the most at this point. Let's change this to an array in C++ first:

```
unsigned char input[] = {
    0x00, 0x00, 0x00, 0x00, // Some flags
    0x02, 0x00,             // Address Family, AF_INET == 0x0002
    0x69, 0x87,             // Source port (big-endian) 27015 == 0x6987
    0x7F, 0x00, 0x00, 0x01, // Source addres 127.0.0.1 == 0x7f000001
    // unknown 8 bytes
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
```

The fields for `ADDRESS_FAMILY`, `SOURCE_PORT` and `SOURCE_ADDRESS` seem pretty clear, we have specified our data and have a direct mapping of it in the buffer. For further confirmation of `ADDRESS_FAMILY` we can look at TCPv6, where in place of `0x0002` we have `0x0017`, which is `AF_INET6`. Moving on, what might the flags be?

Looking at the definitions of the structures in (diversenok), we can see that there they are properly defined and again correspond to what we can observe. The value `0x0000` indicates the normal use of the address (explicit bind). `0x0002`, on the other hand, I assume is supposed to indicate that `AFD.sys` - or further components involved in communication - should infer from which available address they are to establish a connection.

```
#define AFD_NORMALADDRUSE    0
#define AFD_REUSEADDRESS     1
#define AFD_WILDCARDADDRESS  2
#define AFD_EXCLUSIVEADDRUSE 3
```

Mając te informacje możemy częściowo wywnioskować, że mamy do czynienia ze strukturą `SOCKADDR`, która przechowuje `AddressFamily`, `Port`, `Address`. Bardziej meaningful może być tutaj definicja struktury `SOCKADDR_IN`:

```
typedef struct sockaddr_in {

#if(_WIN32_WINNT < 0x0600)
    short   sin_family;
#else //(_WIN32_WINNT < 0x0600)
    ADDRESS_FAMILY sin_family;
#endif //(_WIN32_WINNT < 0x0600)

```

```
9       USHORT sin_port;
10      IN_ADDR sin_addr;
11      CHAR sin_zero[8];
12 } SOCKADDR_IN, *PSOCKADDR_IN;
```

It also explains to us the meaning of the last eight bytes, this is simply padding. I was able to experimentally confirm that they have no meaning on the `AfdBind` call, so the final form that our `AFD_BIND` structure can take can look like the following (similar to (diversenok)):

```
1 struct AFD_BIND_SOCKET {
2     uint32_t flags;
3     SOCKADDR address;
4 }
```

To be sure, I have forced a fixed number of bits for `flags` here (we will have to be aware of structure packing in memory). This structure will look identical for UDPv4 as for TCPv4.

TCPv6 and UDPv6

To analyse `AfdBind` for IPv6 we will find it useful to know that an address in IPv6 is 128 bits long, so let's break up our buffer as an array in C++:

```
1  unsigned char input[] = {
2      0x00, 0x00, 0x00, 0x00 // flags
3      0x17, 0x00             // AF_INET6
4      0x69, 0x87             // 27015
5      0x00, 0x00, 0x00, 0x00 // unknown 4 bytes
6      // ::1
7      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
8  0x00, 0x01
9      0x00, 0x00, 0x00, 0x00 // unknown 4 bytes
   };
```

Basically, here we can already stop and partially deduce that our structure for IPv6 will simply use the `SOCKADDR_IN6` structure, which is the equivalent of `SOCKADDR` but for IPv6:

```
1 struct AFD_BIND_SOCKET6 {
2     uint32_t flags;
3     SOCKADDR_IN6 address;
4 }
```

Why so? Because the structure `SOCKADDR_IN6` further defines `sin6_flowinfo` and `sin6_scope_id` between which our address is located, and this actually corresponds to what we see.

IOCTL for connect command

As with bind, it is useful to create code that generates valid calls to `AfdConnect`, in which case we can skip the explicit `bind` and do `connect` straight away. This time, for an obvious reason, we will only focus on TCP.

```
1  #define WIN32_LEAN_AND_MEAN
2  #include <windows.h>
3  #include <winsock2.h>
4  #include <ws2tcpip.h>
5  #include <iostream>
```

```
#pragma comment(lib,"Ws2_32.lib")

void createTCPv4() {
    SOCKET s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (s == INVALID_SOCKET) { std::cerr << WSAGetLastError() << '\n'; return; }

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

    if (connect(s, reinterpret_cast<sockaddr*>(&dst), sizeof(dst)) == SOCKET_ERROR) {
        std::cerr << "connect: " << WSAGetLastError() << '\n';
        closesocket(s); return;
    }

    closesocket(s);
}

void createTCPv6() {/*SAME FOR TCPv6*/}

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;

    createTCPv4();
    createTCPv6();

    WSACleanup();
    return 0;
}
```

This is simple code to simply establish a connection to `192.168.1.1` on port `80` for IPv4 and `::1` on port `80` for IPv6.

Analyzing retrieved data

We will skip the data collection stage here, as it is identical to that of `bind`, and go straight to presentation and analysis. It is worth starting with the fact that the `IoControlCode` for `AfdConnect` is `0x12007`, which again corresponds to what we have in (diversenok). Below are the collected buffers:

Variant	Input buffer (hex)	Input length (hex)
TCP v4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 f0 19 b5 c8 5c 02 00 00 02 00 00 50 c0 a8 01 01 00 00 00 00 00 00 00 00	0x28
TCP v6	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 a0 ed b5 c8 5c 02 00 00 17 00 00 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00	0x34

In both cases, the matter seems quite simple when represented as an array in C++:

```
// IPv4
unsigned char inputv4[] = {
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0xf0,0x19,0xb5,0xc8,0x5c,0x02,0x00,0x00,
```

```
6       // SOCKADDR
7       0x02,0x00,                                  // AF_INET
8       0x00,0x50,                                  // 80
9       0xc0,0xa8,0x01,0x01,                        // 127.0.0.1
10      0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00     // sin_zero
11  };
12
13  // IPv6
14  unsigned char inputv6[] = {
15      0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
16      0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
17      0xa0,0xed,0xb5,0xc8,0x5c,0x02,0x00,0x00,
18      // SOCKADDR_IN6
19      0x17,0x00,                                  // AF_INET
20      0x00,0x50,                                  // 80
21      0x00,0x00,0x00,0x00,                        // sin6_flowinfo
22      // ::1
23      0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x01,
24      0x00,0x00,0x00,0x00                         // sin6_scope_id
25  }
```

The presence of `SOCKADDR_IN6` seems quite reasonable and simple to deduce from the contents of this buffer. The first three fields of our structure remain a mystery. Based on what is in (diversenok) the first three fields are:

1. `BOOLEAN SanActive`
2. `HANDLE RootEndpoint`
3. `HANDLE ConnectEndpoint`

We know nothing more, we can guess by the names. It is possible that `SAN` refers to Storage Area Network (missing reference), and the other two `HANDLE` could indicate that `AFD.sys` allows us to communicate directly with specific sockets, maybe within a process, maybe across multiple processes? Definitely a topic for further analysis. Interestingly, although in this case we have specified a value for `ConnectEndpoint`, if we specify only zeros there, `AFD.sys` will also accept such a buffer and perform the correct handshake.

We will certainly come back to this, it will be worth considering for malicious use!

Next steps

In next parts we will lean into sending and receiving data from our socket using the TCP protocol. While sending seems fairly straightforward, we will probably have to dig deeper.

Final code

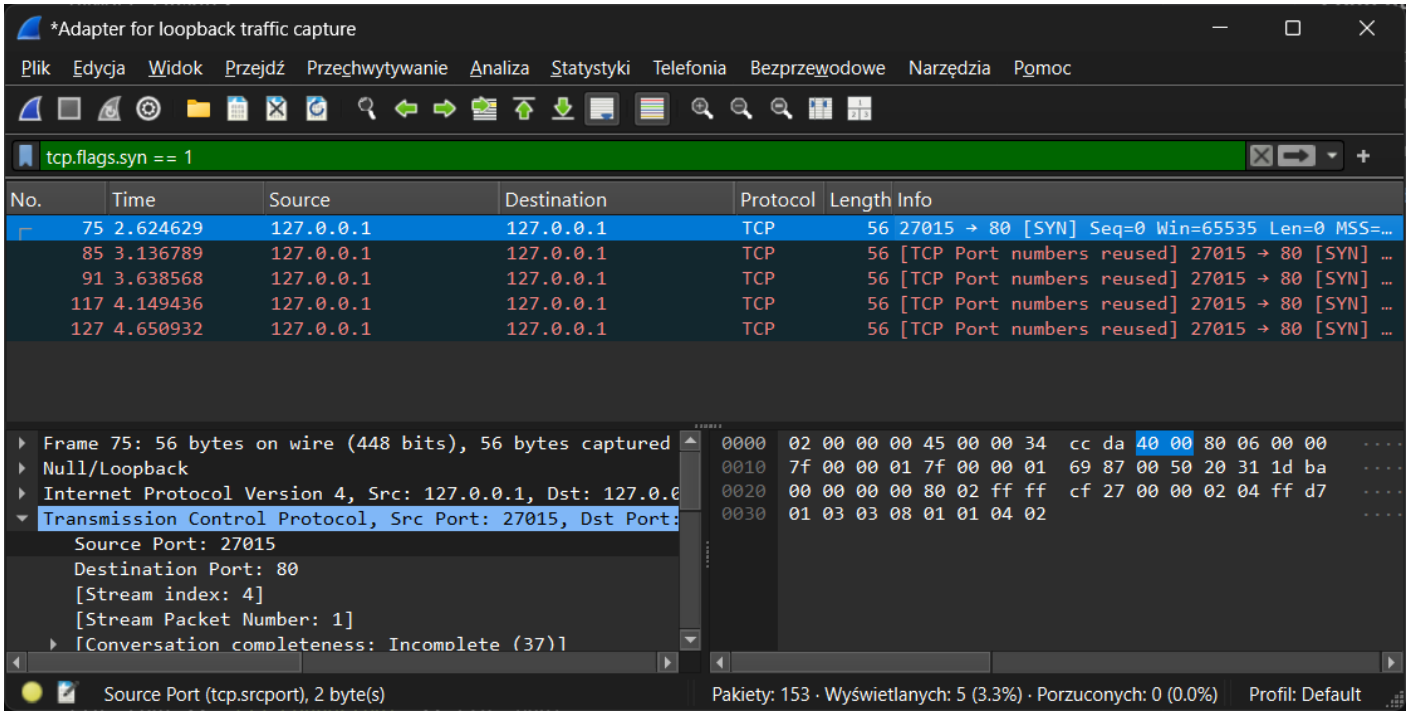
Below you can find the full code that creates a socket without using any networking library. This definitely requires additional helpers to allow us to convert IP and port to the appropriate fields in `SOCKADDR`, but without using functions from Winsock.

```
1   #include <stdint.h>
2   #include <Windows.h>
3   #include <winternl.h>
4   #include <iostream>
5   #include "afd_defs.h"
```

```
6  #include "afd_ioctl.h"
7  #pragma comment(lib, "ntdll.lib")
8
9  NTSTATUS createAfdSocket(PHANDLE socket) {...}
10
11 #define AFD_NORMALADDRUSE    0
12 #define AFD_REUSEADDRESS     1
13 #define AFD_WILDCARDADDRESS  2
14 #define AFD_EXCLUSIVEADDRUSE 3
15
16 struct AFD_BIND_SOCKET {
17     uint32_t flags;
18     SOCKADDR address;
19 };
20
21 NTSTATUS bindAfdSocket(HANDLE socket) {
22     AFD_BIND_SOCKET afdBindSocket    = { 0 };
23     afdBindSocket.flags              = AFD_NORMALADDRUSE;
24     afdBindSocket.address.sa_family  = AF_INET;
25     // PORT == 27015
26     afdBindSocket.address.sa_data[0] = 0x69;
27     afdBindSocket.address.sa_data[1] = 0x87;
28     // ADDRESS == 127.0.0.1
29     afdBindSocket.address.sa_data[2] = 0x7F;
30     afdBindSocket.address.sa_data[3] = 0x00;
31     afdBindSocket.address.sa_data[4] = 0x00;
32     afdBindSocket.address.sa_data[5] = 0x01;
33
34     uint8_t outputBuffer[0x10];
35
36     IO_STATUS_BLOCK ioStatus;
37     NTSTATUS status = NtDeviceIoControlFile(socket, NULL, NULL, NULL, &ioStatus,
38 IOCTL_AFD_BIND,
39                                             &afdBindSocket, sizeof(AFD_BIND_SOCKET),
40                                             outputBuffer, 0x00000010);
41     if (status == STATUS_PENDING) {
42         WaitForSingleObject(socket, INFINITE);
43         status = ioStatus.Status;
44     }
45     return status;
46 }
47
48 struct AFD_CONNECT_SOCKET {
49     uint64_t sanActive;
50     uint64_t rootEndpoint;
51     uint64_t connectEndpoint;
52     SOCKADDR address;
53 };
54
55 NTSTATUS connectAfdSocket(HANDLE socket) {
56     AFD_CONNECT_SOCKET afdConnectSocket = { 0 };
57     afdConnectSocket.sanActive       = 0x00;
58     afdConnectSocket.rootEndpoint    = 0x00;
59     afdConnectSocket.connectEndpoint = 0x00;
60     afdConnectSocket.address.sa_family = AF_INET;
61     // PORT == 80
62     afdConnectSocket.address.sa_data[0] = 0x00;
63     afdConnectSocket.address.sa_data[1] = 0x50;
64     // ADDRESS == 127.0.0.1
65     afdConnectSocket.address.sa_data[2] = 0x7F;
66     afdConnectSocket.address.sa_data[3] = 0x00;
67     afdConnectSocket.address.sa_data[4] = 0x00;
68     afdConnectSocket.address.sa_data[5] = 0x01;
69
70     IO_STATUS_BLOCK ioStatus;
71     NTSTATUS status = NtDeviceIoControlFile(socket, NULL, NULL, NULL, &ioStatus,
72 IOCTL_AFD_CONNECT,
```

```
                                                &afdConnectSocket,
sizeof(AFD_CONNECT_SOCKET),
                                                NULL, NULL);
    if (status == STATUS_PENDING) {
        WaitForSingleObject(socket, INFINITE);
        status = ioStatus.Status;
    }
    return status;
}

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;

    status = bindAfdSocket(socket);
    if (!NT_SUCCESS(status)) {
        std::cout << "[-] Could not bind: " << std::hex << status << std::endl;
        return 1;
    }
    std::cout << "[+] Socket bound!" << std::endl;

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

        return 0;
}
```

After executing this code, we can see that we are actually trying to set up a handshake from port `27015` to port `80` on `localhost`:



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_shared.h*. 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. *\Textttntafd.h – Ancillary Function Driver Definitions*. commit 2dda0dd, Hunt & Hackett, April 2025, https://github.com/winsiderss/systeminformer/blob/master/phnt/include/ntafd.h.