Under the Hood of AFD.sys Part 3: Sending TCP packets

Mateusz Lewczak ⋮ ⋮ 30/07/2025

A deep-dive into the IOCTL_AFD_SEND Fast-I/O path: snaring AfdFastIoDeviceControl hits in WinDbg, reverse-engineering the AFD_SEND_INFO / WSABUF chain, and blasting raw TCP payloads straight from user space on Windows 11—still no Winsock, just pure AFD.sys magic.

Posted Jul 30, 2025
By *Mateusz Lewczak*

20 min read

Under the Hood of AFD.sys Part 3: Sending TCP packets

Introduction

With a word of introduction, this post is the third in a series of articles in which we take a closer look at the `AFD.sys` driver. So far we have managed to create a socket and perform a three-way handshake using only I/O request packets to `AFD.sys` with the omission of Winsock and `mswsock.dll`. Now it was time to send and receive the packet.

As tradition dictates, here is our code from Winsock for our reference:

```
1  void createTCPv4() {
2      const size_t PAYLOAD = 8;
3
4      SOCKET s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
5      if (s == INVALID_SOCKET) { std::cerr << "socket: " << WSAGetLastError() << '\n';
6  return; }
7
8      sockaddr_in dst{};
9      dst.sin_family = AF_INET;
10     dst.sin_port = htons(80);
11     InetPtonA(AF_INET, "192.168.1.1", &dst.sin_addr);
12
13     if (connect(s, reinterpret_cast<sockaddr*>(&dst), sizeof(dst)) == SOCKET_ERROR) {
14         std::cerr << "connect: " << WSAGetLastError() << '\n';
15         closesocket(s); return;
16     }
17
18     std::string big(PAYLOAD, 'A');
19
20     size_t sent = 0;
21     while (sent < big.size()) {
22         int n = send(s, big.data() + sent, static_cast<int>(big.size() - sent), 0);
23         if (n == SOCKET_ERROR) {
24             std::cerr << "send: " << WSAGetLastError() << '\n';
25             break;
26         }
27         sent += n;
28     }
```

```
29
30      closesocket(s);
    }
```

After how easy it was to intercept the communiqué between our example program from Winsock and `AFD.sys` I thought `send` and `recv` would be equally easy, but I was wrong. Setting the breakpoints to `afd!AfdSend` and `afd!AfdReceive` did nothing. The previously adopted method was not effective, in this case.

Starting with `send`, I thought at that point that maybe just maybe `AfdSend` is not the function that is actually called to send TCP packets. I started searching by available symbols and the phrase `Send`, I then hit nearly 96 different entries in the export table…

How do drivers differentiate between requests?

Unlike a normal program, where the start function is `main` (simplification), in Windows drivers such a function is `DriverEntry`. This is the place where the `DRIVER_OBJECT` is created, which is a structure describing the device being made available, you will find there such information as:

- The name of the device, which will be visible, e.g. `\Device\Afd`.
- Function setting, when the driver is initialised/deinitialised.
- Setting of the disptach function that is called when an `IRP` comes in.

In order for the driver to distinguish between specific codes there is such a thing as a dispatch function. This is a function that decodes the `IoControlCode` and passes the data and control to the next function responsible for handling that particular request. For example, below we have the pseudo C code (Binary Ninja) from the `AfdDispatchDeviceControl` function:

```
1  uint64_t AfdDispatchDeviceControl(int64_t arg1, IRP* arg2) {
2      void* Overlay = *(uint64_t*)((char*)arg2->Tail + 0x40);
3
4      if (NetioNrtIsTrackerDevice()) {
5          int32_t rax_6 = NetioNrtDispatch(arg1, arg2);
6          *(uint32_t*)((char*)arg2->IoStatus. + 0) = rax_6;
7          IofCompleteRequest(arg2, 0);
8          return (uint64_t)rax_6;
9      }
10
11     int32_t r8 = *(uint32_t*)((char*)Overlay + 0x18);
12     uint64_t rax_3 = (uint64_t)(r8 >> 2) & 0x3ff;
13
14     if (rax_3 < 0x4a && *(uint32_t*)((rax_3 << 2) + &AfdIoctlTable) == r8) {
15         *(uint8_t*)((char*)Overlay + 1) = rax_3;
16
17         if ((&AfdIrpCallDispatch)[rax_3])
18             return _guard_dispatch_icall();
19     }
20
21     if ((*(int64_t*)((char*)g_rgFastWppLevelEnabledFlags + 0xe)) & 0x10)
22         WPP_SF_D(0xb, &WPP_750cd5b025b73ac1a6ce4c47647b8469_Traceguids, r8);
23
24     *(uint32_t*)((char*)arg2->IoStatus. + 0) = 0xc0000010;
25     IofCompleteRequest(arg2, AfdPriorityBoost);
26     return 0xc0000010;
27 }
```

There are a number of ways on how to perform such a dispatch, one is simply to create a series of expressions with `if` or `switch/case` and based on the resulting `IoControlCode` value, the specific function responsible for performing the operation is called.

The second way (used in `AFD.sys`) is to create a call table (see `AfdIrpCallDispatch`). Instead of complex conditional expressions, the driver creates an array of (pointers to) functions for itself and, depending on the decoded function, the corresponding `call` is executed. A fragment of this code can be found in lines 14 to 19 in the snippet above.

We can go further and see what the content of this `AfdIrpCallDispatch` table looks like:

```
1  1c0059410  void* AfdIrpCallDispatch = AfdBind
2  1c0059418  void* data_1c0059418 = AfdConnect
3  1c0059420  void* data_1c0059420 = AfdStartListen
4  1c0059428  void* data_1c0059428 = AfdWaitForListen
5  1c0059430  void* data_1c0059430 = AfdAccept
6  1c0059438  void* data_1c0059438 = AfdReceive
7  1c0059440  void* data_1c0059440 = AfdReceiveDatagram
8  1c0059448  void* data_1c0059448 = AfdSend
9  1c0059450  void* data_1c0059450 = AfdSendDatagram
10 1c0059458  void* data_1c0059458 = AfdPoll
11 1c0059460  void* data_1c0059460 = AfdDispatchImmediateIrp
12 1c0059468  void* data_1c0059468 = AfdGetAddress
13 1c0059470  void* data_1c0059470 = AfdDispatchImmediateIrp
14 1c0059478  void* data_1c0059478 = AfdDispatchImmediateIrp
15 ...
```

We see there, for example, that operation 0 will be `AfdBind`, operation 1 will be `AfdConnect`, and we also find there that operation 7 will be `AfdSend`. And these offsets are actually reflected in how we build the `IoControlCode` to communicate with `AFD.sys`. Our control code is encoded with information about what operation we want to perform:

```
  ...
1 #define AFD_BIND                        0
2 #define AFD_CONNECT                     1
3 ...
4 #define FSCTL_AFD_BASE  FILE_DEVICE_NETWORK
5 #define _AFD_CONTROL_CODE(Request, Method) (FSCTL_AFD_BASE << 12 | (Request) << 2 |
6 (Method))
7 ...
8 #define IOCTL_AFD_BIND                          _AFD_CONTROL_CODE(AFD_BIND,
9 METHOD_NEITHER)    // 0x12003
  #define IOCTL_AFD_CONNECT                       _AFD_CONTROL_CODE(AFD_CONNECT,
  METHOD_NEITHER) // 0x12007
```

Intercepting AfdDispatchDeviceControl

So instead of creating a breakpoint on `afd!AfdSend` let's try setting one for our `afd!AfdDispatchDeviceControl` function. What I want to do at this point is simply check what `IoControlCode` values are sent to our driver and see if one of them will be `IOCTL_AFD_SEND` (`0x1201F`). To do this we will use the JavaScript below, which is supposed to read the `IoControlCode` value at each hit:

```
1  "use strict";
2
```

```
function GetIoctl(irpAddr){
    // Get _IRP object
    const irp = host.createTypedObject(irpAddr, "nt", "_IRP");
    // Get _IO_STACK_LOCATION address
    const stackPtr = irp.Tail.Overlay.CurrentStackLocation;
    // Get _IO_STACK_LOCATION object
    const isl = stackPtr.dereference();

    const code = isl.Parameters.DeviceIoControl.IoControlCode;
    return code;
}
```

Now we need to load our script and set the appropriate breakpoint, which will write us the returned value and not stop each time:

```
10: kd> .scriptrun D:\afddispatch.js;
JavaScript script successfully loaded from 'D:\afddispatch.js'
JavaScript script 'D:\afddispatch.js' has no main function to invoke!

14: kd> .foreach /pS 1 (ep { !process 0 0 afd_re.exe }) { bm /p ${ep}
afd!AfdDispatchDeviceControl "dx
Debugger.State.Scripts.afddispatch.Contents.GetIoctl(@rdx);gc;" }
 17: fffff800`515b2db0 @!"afd!AfdDispatchDeviceControl"

14: kd> g
Debugger.State.Scripts.afddispatch.Contents.GetIoctl(@rdx) : 0x120bf  //
IOCTL_AFD_TRANSPORT_IOCTL
Debugger.State.Scripts.afddispatch.Contents.GetIoctl(@rdx) : 0x12003  //
IOCTL_AFD_BIND
Debugger.State.Scripts.afddispatch.Contents.GetIoctl(@rdx) : 0x12007  //
IOCTL_AFD_CONNECT
```

Already from the obtained `IoControlCode` we can see that we only have `AfdBind` and `AfdConnect`, but where is our `AfdSend`? After many hours of reversing `AFD.sys` and `mswsock.dll` and searching the Internet for information I came across something called **Fast I/O**.

What is Fast I/O?

I will use the book *Windows® Internals Part 2 - 6th edition* (especially Chapter 11) (Allievi et al.) as one source of information here. As we can read on page 375, *Fast I/O* is Windows' mechanism for performing fast operations, bypassing all the anguish involved in generating *I/O request packets*. Our driver first checks if something can be handled as *Fast I/O*, if so it goes to another dispatch function that will handle the request. Although in the book itself the author refers to a *File system driver*, as we will see this does not apply only to file handling. One of the requirements to be able to handle *Fast I/O* is that our request must be synchronous, and our `send` function from Winsock is, after all, waiting until it receives the result - I don't know if this is the good determinant, `mswsock.dll` may handle it differently, but it's always something. Importantly, requests that can be handled as *Fast I/O* do not go to the traditional dispatch function.

Looking for send

We have some suspicion that `AFD.sys` supports `send` as *Fast I/O*, so let's start looking for confirmation in the code. Like traditional dispatch, fast dispatch is also set in `DriverEntry`:

```
1  NTSTATUS DriverEntry(DRIVER_OBJECT* arg1) {
2  ...
3      rdi_3 = __memfill_u64(&arg1->MajorFunction, AfdDispatch, 0x1c);
4      arg1->MajorFunction[0xe] = AfdDispatchDeviceControl;
5      arg1->MajorFunction[0xf] = AfdWskDispatchInternalDeviceControl;
6      arg1->MajorFunction[0x17] = AfdEtwDispatch;
7      arg1->FastIoDispatch = &AfdFastIoDispatch;
8      arg1->DriverUnload = AfdUnload;
9      void* AfdDeviceObject_1 = AfdDeviceObject;
10 ...
11 }
```

And so right next to `AfdDispatchDeviceControl` we have the `AfdFastIoDispatch` function, it is worth taking a closer look at it. Our `AfdFastIoDispatch` object is an array:

```
1 1c0065000  AfdFastIoDispatch:
2 1c0065000  e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
3 1c0065010  void* data_1c0065010 = AfdFastIoRead
4 1c0065018  void* data_1c0065018 = AfdFastIoWrite
5 1c0065020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
6 1c0065030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
7 1c0065040  void* data_1c0065040 = AfdSanFastUnlockAll
8 1c0065048                           00 00 00 00 00 00 00 00          ........
9 1c0065050  void* data_1c0065050 = AfdFastIoDeviceControl
```

In our array we can see the entry `AfdFastIoDeviceControl`, which is a dispatch function, but for *Fast I/O*. Why not throw a breakpoint in there and collect the `IoControlCode`. Except that they won't have to delve into the `_IRP` structure, the operation code is passed as one of the arguments of the `PFAST_IO_DEVICE_CONTROL` call:

```
1  typedef
2  BOOLEAN
3  (*PFAST_IO_DEVICE_CONTROL) (
4      IN struct _FILE_OBJECT *FileObject,
5      IN BOOLEAN Wait,
6      IN PVOID InputBuffer OPTIONAL,
7      IN ULONG InputBufferLength,
8      OUT PVOID OutputBuffer OPTIONAL,
9      IN ULONG OutputBufferLength,
10     IN ULONG IoControlCode,
11     OUT PIO_STATUS_BLOCK IoStatus,
12     IN struct _DEVICE_OBJECT *DeviceObject
13     );
```

So all we need to do is read the seventh argument (`@rdi`) of the call, we do this by setting such a breakpoint:

```
1  6: kd> .foreach /pS 1 (ep { !process 0 0 afd_re.exe }) { bm /p ${ep}
2  afd!AfdFastIoDeviceControl ".printf \"IoControlCode=%p\\n\", @rdi;gc;" }
3    2: fffff800`515c4c20 @!"afd!AfdFastIoDeviceControl"
4  Couldn't resolve error at 'SessionId: afd!AfdFastIoDeviceControl ".printf
5  \"IoControlCode=%p\\n\", @rdi;gc;" '
6
7  6: kd> g
8  IoControlCode=000000000001207b // IOCTL_AFD_TRANSMIT_FILE
9  IoControlCode=000000000001207b // IOCTL_AFD_TRANSMIT_FILE
10 IoControlCode=0000000000012047 // IOCTL_AFD_SET_CONTEXT
11 IoControlCode=00000000000120bf // IOCTL_AFD_TRANSPORT_IOCTL
12 IoControlCode=0000000000012047 // IOCTL_AFD_SET_CONTEXT
13 IoControlCode=0000000000012003 // IOCTL_AFD_BIND
   IoControlCode=0000000000012047 // IOCTL_AFD_SET_CONTEXT
```

```
14 IoControlCode=0000000000012007 // IOCTL_AFD_CONNECT
15 IoControlCode=0000000000012047 // IOCTL_AFD_SET_CONTEXT
   IoControlCode=000000000001201f // IOCTL_AFD_SEND
```

Ok, there we have it! Our `send` is treated as *Fast I/O*, let's try to look at the `AFD.sys` code and find what function is called when the driver receives `0x1201f`:

```
1  1c0034be0   int64_t AfdFastIoDeviceControl(struct _FILE_OBJECT* FileObject,
2  1c0034be0      BOOLEAN Wait, PVOID InputBuffer, ULONG InputBufferLength,
3  1c0034be0      PVOID OutputBuffer, ULONG OutputBufferLength, ULONG IoControlCode,
4  1c0034be0      PIO_STATUS_BLOCK IoStatus, struct _DEVICE_OBJECT* DeviceObject) {
5      ...
6  1c0034c9b                          if (IoControlCode == 0x1201f)
7  1c0034c9b                              goto label_1c0034d7d;
8      ...
9  1c0034d7d                              label_1c0034d7d:
10 1c0034d7d                              __builtin_memset(&s_2, 0, 0x14);
11 1c0034d8d                              int128_t s_3;
12 1c0034d8d                              __builtin_memset(&s_3, 0, 0x48);
13     ...
14 1c00350f7                              rbx = (uint64_t)AfdFastConnectionSend(FsContext,
15 1c00350f7                                  &s_2, rax_30, IoStatus);
16 1c00350fa                              goto label_1c003646b;
17     ...
18 1c0034be0    }
```

The code of the entire `AfdFastIoDeviceControl` is quite extensive, so I have only shown the parts related to our `0x1201f`. We can find there that if `IoControlCode == 0x1201f`, then execute `jmp` to `0x1c0034d7d`. This is where the initialisation of all necessary memory areas, variables etc. starts. And a piece further on we have a call to the `AfdFastConnectionSend` function. This could be our function responsible for sending the data. Of course, to confirm this we should now set a breakpoint there:

```
1 6: kd> .foreach /pS 1 (ep { !process 0 0 afd_re.exe }) { bm /p ${ep}
2 afd!AfdFastConnectionSend }
3   4: fffff800`515aac90 @!"afd!AfdFastConnectionSend"
4 Couldn't resolve error at 'SessionId: afd!AfdFastConnectionSend '
5
6 6: kd> g
7
8 Breakpoint 4 hit
9 afd!AfdFastConnectionSend:
  fffff800`515aac90 4053            push    rbx
```

Hit! We found our function responsible for sending data via TCP! Now it is time to analyse the input buffer. Here, as usual, our invaluable sources (killvxk), (unknowncheats.me ICoded post), (ReactOS Project), (DynamoRIO / Dr. Memory), (DeDf), (diversenok) will help us.

Analyzing retrieved data AfdFastConnectionSend

From our signature for `PFAST_IO_DEVICE_CONTROL`, we know that to the dispatch, `InputBuffer` and `InputBufferLength` are passed as arguments to the third and fourth arguments, respectively. We are not sure that they are passed to `AfdFastConnectionSend` at the same positions, but we can safely assume that they are also passed directly as arguments. So what we'll be looking for is by the values of the address registers from user-space (*canonical lower half*) and some (relatively) small buffer length value.

```
12: kd> r
rax=0000000000000002 rbx=000000c9532ff128 rcx=ffffbd8bfa8dda80
rdx=fffffce0958bcef70 rsi=0000000000000001 rdi=0000000000000000
rip=fffff800515aac90 rsp=fffffce0958bcee88 rbp=fffffce0958bcf4e0
 r8=0000000000000008  r9=fffffce0958bcf1c8 r10=fffff800bbc17c70
r11=ffff88f9d7c00000 r12=ffffbd8bfa8dda80 r13=0000000000000000
r14=0000000000000018 r15=000000000000afd1
iopl=0         nv up ei pl zr na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00040246
afd!AfdFastConnectionSend:
fffff800`515aac90 4053            push    rbx
```

Here we see that the `rbx` register stores something that may resemble an address in user-space, while `r14` looks like the size of our buffer. So let's read their value:

```
12: kd> db 000000c9532ff128 L18
000000c9`532ff128  08 f2 2f 53 c9 00 00 00-01 00 00 00 00 00 00 00  ../S............
000000c9`532ff138  00 00 00 00 00 00 00 00                         ........
```

Again we have something that resembles an address and some size, let's try to read (note on the dumpy it is *little-endian*) `0x000000c9532ff208`:

```
12: kd> db 000000c9532ff208 L18
000000c9`532ff208  08 00 00 00 00 00 00 00-e0 f2 2f 53 c9 00 00 00  ........../S....
000000c9`532ff218  00 00 00 00 00 00 00 00                         ........
```

Once again, we see some size (`0x08`), which corresponds to the `AAAAAAAA` payload we sent. Let's try another dereference and check to see what it is at `0x000000c9532ff2e0`:

```
12: kd> db 0x000000c9532ff2e0 L8
12: kd> db 0x000000c9532ff2e0 L8
000000c9`532ff2e0  41 41 41 41 41 41 41 41                         AAAAAAAA
```

We've got it! There is our payload! But the question is how are the buffers constructed? The answer to that will be found in (diversenok):

```
// ref: https://learn.microsoft.com/en-us/windows/win32/api/ws2def/ns-ws2def-wsabuf
typedef struct _WSABUF {
  ULONG len;
  CHAR  *buf;
} WSABUF, *LPWSABUF;

typedef struct _AFD_SEND_INFO {
    _Field_size_(BufferCount) LPWSABUF BufferArray;
    ULONG BufferCount;
    ULONG AfdFlags;
    ULONG TdiFlags; // TDI_RECEIVE_*
} AFD_SEND_INFO, *PAFD_SEND_INFO;
```

Breaking this down step by step, we first have a `_AFD_SEND_INFO` structure containing a pointer to an array of buffers and the number of these buffers. In each buffer, on the other hand, we have its length and a pointer to the data. A fairly good analogy for this might be the standard use of `argv` in the `main` function. There, too, we are dealing with an array for pointers to the buffers of our arguments passed to the program.

A keen eye can spot a certain inconsistency. After all, we know that the InputBuffer from Winsock is `0x18` bytes and our `_AFD_SEND_INFO` structure is `0x20` bytes. I have experimentally verified that, in principle, `TdiFlags` is optional. Presumably if we had indicated `TransportDevice` (e.g. `DeviceTcp`) when creating the socket we would have had to indicate this. This leaves the conundrum of what values can `AfdFlags` take?

According to what we have in (diversenok) this could be:

```
1 #define AFD_NO_FAST_IO 0x0001
2 #define AFD_OVERLAPPED 0x0002
```

The `AFD_NO_FAST_IO` seems to be the most interesting from the perspective of our work so far. In fact when we set `AfdFlags` to `0x0001` then `AFD.sys` goes through a classic dispatch and the breakpoint on `AfdSend` is triggered:

```
1 12: kd> .foreach /pS 1 (ep { !process 0 0 afd-networking.exe }) { bm /p ${ep}
2 afd!AfdSend }
3   6: fffff800`515a18c0 @!"afd!AfdSend"
4 Couldn't resolve error at 'SessionId: afd!AfdSend '
5 12: kd> g
6 Breakpoint 6 hit
7 afd!AfdSend:
  fffff800`515a18c0 4c8bdc          mov     r11,rsp
```

So, that's cool, we can control how this particular request will be dispatched. It's worth saving this for a later reserach on where and how this is done. What about TCPv6? Generally it looks the same, there are no big differences in sending packets. Socket created, connection established, interface to send is the same.

The question now would be how many buffers can it send, how big can they be? Does the total number of bytes count? Let's find out!

Playing with buffers

So let's perhaps start by trying to send 10 megabytes using WinSock and see if it breaks it up somehow, to get a general idea of what we're dealing with. By default, I set my breakpoint to `afd!AfdFastIoDeviceControl` and write out the `IoControlCode` to see if, for example, Winsock is splitting this data packet into multiple requests:

```
1  14: kd> .foreach /pS 1 (ep { !process 0 0 afd_re.exe }) { bm /p ${ep}
2  afd!AfdFastIoDeviceControl ".printf \"IoControlCode=%p\\n\", @rdi;gc;" }
3   10: fffff800`515c4c20 @!"afd!AfdFastIoDeviceControl"
4  Couldn't resolve error at 'SessionId: afd!AfdFastIoDeviceControl ".printf
5  \"IoControlCode=%p\\n\", @rdi;gc;" '
6  14: kd> g
7  IoControlCode=000000000001207b
8  IoControlCode=000000000001207b
9  IoControlCode=0000000000012047
10 IoControlCode=00000000000120bf
11 IoControlCode=0000000000012047
12 IoControlCode=0000000000012003
13 IoControlCode=0000000000012047
14 IoControlCode=0000000000012007
```

```
IoControlCode=0000000000012047
IoControlCode=000000000001201f
```

Despite our loop to make sure all the data was sent this Winsock managed to send 10 Megabytes at a time:

```
while (sent < big.size()) {
    int n = send(s, big.data() + sent, static_cast<int>(big.size() - sent), 0);
    std::cerr << "sent portion: " << n << '\n';
    if (n == SOCKET_ERROR) {
        std::cerr << "send: " << WSAGetLastError() << '\n';
        break;
    }
    sent += n;
}
```

And what does the buffer that is passed to `AfdFastConnectionSend` look like?

```
8: kd> .foreach /pS 1 (ep { !process 0 0 afd_re.exe }) { bm /p ${ep}
afd!AfdFastConnectionSend }
 12: fffff800`515aac90 @!"afd!AfdFastConnectionSend"
Couldn't resolve error at 'SessionId: afd!AfdFastConnectionSend '
8: kd> g
Breakpoint 12 hit
afd!AfdFastConnectionSend:
fffff800`515aac90 4053            push    rbx
8: kd> r
rax=0000000000000002 rbx=000000ce1a6ff328 rcx=ffffbd8bfa8dac00
rdx=fffffce0956db6f70 rsi=0000000000000001 rdi=0000000000000000
rip=fffff800515aac90 rsp=fffffce0956db6e88 rbp=fffffce0956db74e0
 r8=0000000006400000  r9=fffffce0956db71c8 r10=fffff800bbc17c70
r11=ffff88f9d7c00000 r12=ffffbd8bfa8dac00 r13=0000000000000000
r14=0000000000000018 r15=000000000000afd1
iopl=0         nv up ei pl zr na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00040246
afd!AfdFastConnectionSend:
fffff800`515aac90 4053            push    rbx
8: kd> dq 000000ce1a6ff328 L3
000000ce`1a6ff328  000000ce`1a6ff408 00000000`00000001
000000ce`1a6ff338  00000000`00000000
8: kd> dq 000000ce`1a6ff408 L2
000000ce`1a6ff408  00000000`06400000 00000242`e3249080
```

Everything flies in one big buffer - the same for 1 Gigabyte. So I am curious how realistically `AFD.sys` interprets these buffers. Maybe `n` buffers will be sent as `n` packets? This is already verified without using Winsock:

```
NTSTATUS sendAfdPacketTCP(HANDLE socket) {
    const int BUF_NUM = 16;
    const int BUF_SIZE = 16;
    AFD_BUFF* payload = new AFD_BUFF[BUF_NUM];
    for (int i = 0; i < BUF_NUM; i++) {
        payload[i].buf = (uint8_t*)malloc(BUF_SIZE);
        memset(payload[i].buf, 0x42, BUF_SIZE);
        payload[i].len = BUF_SIZE;
    }

    AFD_SEND_PACKET* afdSendPacket = new AFD_SEND_PACKET;
    afdSendPacket->BufferArray = payload;
    afdSendPacket->BufferCount = BUF_NUM;
    afdSendPacket->AfdFlags = AFD_NO_FAST_IO;

```

```
16      IO_STATUS_BLOCK ioStatus;
17      NTSTATUS status = NtDeviceIoControlFile(socket, NULL, NULL, NULL, &ioStatus,
18  IOCTL_AFD_SEND,
19                                              afdSendPacket, sizeof(AFD_SEND_PACKET),
20                                              NULL, NULL);
21      if (status == STATUS_PENDING) {
22          WaitForSingleObject(socket, INFINITE);
23          status = ioStatus.Status;
24      }
25      return status;
    }
```

As it turns out this changes nothing, it flies as one packet. For obvious reasons per the TCP specification the packet would be split once it exceeded `0xFFFF` bytes, but the number of buffers has no bearing on this. I checked experimentally and `AFD.sys` will also accept `1024*1024` buffers of `1024` bytes each. An important limitation, of course, remains our hardware.

Next steps

Although I originally intended to discuss both `send` and `receive` in this part, this article is long enough that it is in the next step that we will deal with receiving TCP packets.

Final code

Below you can find the full code for the current state of our knowledge:

```
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 NTSTATUS bindAfdSocket(HANDLE socket) {...}
11 NTSTATUS connectAfdSocket(HANDLE socket) {...}
12
13 // AFDFLAGS
14 #define AFD_NO_FAST_IO 0x0001
15 #define AFD_OVERLAPPED 0x0002
16
17 struct AFD_BUFF {
18     uint64_t len;
19     uint8_t* buf;
20 };
21
22 struct AFD_SEND_PACKET {
23     AFD_BUFF* buffersArray;
24     uint64_t  buffersCount;
25     uint64_t  afdFlags;
26     uint64_t  tdiFlags; // optional
27 };
28
29 NTSTATUS sendAfdPacketTCP(HANDLE socket) {
30     const int BUF_NUM  =  1;
31     const int BUF_SIZE = 16;
32     AFD_BUFF* payload = new AFD_BUFF[BUF_NUM];
33     for (int i = 0; i < BUF_NUM; i++) {
34         payload[i].buf = (uint8_t*)malloc(BUF_SIZE);
```

```
        memset(payload[i].buf, 0x42, BUF_SIZE);
        payload[i].len = BUF_SIZE;
    }

    AFD_SEND_PACKET* afdSendPacket  = new AFD_SEND_PACKET;
    afdSendPacket->buffersArray     = payload;
    afdSendPacket->buffersCount     = BUF_NUM;
    afdSendPacket->afdFlags         = 0;

    IO_STATUS_BLOCK ioStatus;
    NTSTATUS status = NtDeviceIoControlFile(socket, NULL, NULL, NULL, &ioStatus,
IOCTL_AFD_SEND,
                                            afdSendPacket, sizeof(AFD_SEND_PACKET),
                                            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;

    status = sendAfdPacketTCP(socket);
    if (!NT_SUCCESS(status)) {
        std::cout << "[-] Could not send TCP packet: " << std::hex << status <<
std::endl;
        return 1;
    }
    std::cout << "[+] Sent!" << 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_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. *Texttttntafd.h – Ancillary Function Driver Definitions*. commit 2dda0dd, Hunt & Hackett, April 2025, https://github.com/winsiderss/systeminformer/blob/master/phnt/include/ntafd.h.