Under the Hood of AFD.sys Part 4: Receiving TCP packets

Mateusz Lewczak ⋮ ⋮ 06/08/2025

A hands-on foray into the IOCTL_AFD_RECEIVE Fast-I/O path: stalking AfdFastConnectionReceive in WinDbg, decoding the AFD_SENDRECV_INFO / WSABUF triad, flipping TDI flags for peek-and-poke tricks, and slurping raw TCP responses straight out of AFD.sys—zero Winsock, pure kernel-level packet sorcery.

Posted Aug 6, 2025
By *Mateusz Lewczak*

11 min read

Under the Hood of AFD.sys Part 4: Receiving TCP packets

Introduction

Ok, the time has come, we can finally receive some data in our sockets. If you haven't seen the previous batches, I encourage you to check them out, so far we've managed to create a socket and send TCP packets. We still have a long way to go to fully understand how networking works by communicating directly with the `AFD.sys` driver, but there will be time for that yet. No need to procrastinate, let's go!

Looking for recv

As before, `recv` is handled as *Fast I/O* (unless we set the flags differently). A good reference for this will be `IOCTL_AFD_RECEIVE` going into our `AfdFastIoDeviceControl` function. By default, as before, our reference will be this code using Winsock:

```
void createTCPv4() {
    const size_t PAYLOAD = 1024;
    SOCKET s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (s == INVALID_SOCKET) { std::cerr << "socket: " << 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;
    }

    std::string big(PAYLOAD, 'A');

    size_t sent = 0;
    while (sent < big.size()) {
        int n = send(s, big.data() + sent, static_cast<int>(big.size() - sent), 0);
        if (n == SOCKET_ERROR) {
            break;
        }
        sent += n;
    }

    char buf[4096];
    int n = 0;
    size_t received = 0;
    std::string response;

    while ((n = recv(s, buf, static_cast<int>(sizeof(buf)), 0)) > 0) {
        response.append(buf, n);
        received += n;
    }

    if (n == SOCKET_ERROR) {
        std::cerr << "recv: " << WSAGetLastError() << '\n';
    }

    std::cout << "Received " << received << " bytes\n";
    std::cout << "----- RESPONSE BEGIN -----\n"
        << response << '\n'
        << "----- RESPONSE END -----\n";

    closesocket(s);
}
```

As we connect to the HTTP server and send garbage we get a `Bad request` in response - a clear case. To confirm that we are indeed `recv` hitting the driver as *Fast I/O* we will use a command like this in WinDbg:

```
14: kd> .foreach /pS 1 (ep { !process 0 0 afd_re.exe }) { bm /p ${ep}
afd!AfdFastIoDeviceControl ".printf \"IoControlCode=%p\\n\", @rdi;gc;" }
  0: fffff801`6d9d4c20 @!"afd!AfdFastIoDeviceControl"
Couldn't resolve error at 'SessionId: afd!AfdFastIoDeviceControl ".printf
\"IoControlCode=%p\\n\", @rdi;gc;" '
14: kd> g
IoControlCode=000000000001207b // IOCTL_AFD_GET_INFORMATION
IoControlCode=000000000001207b // IOCTL_AFD_GET_INFORMATION
IoControlCode=0000000000012047 // IOCTL_AFD_SET_CONTEXT
IoControlCode=00000000000120bf // IOCTL_AFD_TRANSPORT_IOCTL
IoControlCode=0000000000012047 // IOCTL_AFD_SET_CONTEXT
IoControlCode=0000000000012003 // IOCTL_AFD_BIND
IoControlCode=0000000000012047 // IOCTL_AFD_SET_CONTEXT
IoControlCode=0000000000012007 // IOCTL_AFD_CONNECT
IoControlCode=0000000000012047 // IOCTL_AFD_SET_CONTEXT
IoControlCode=000000000001201f // IOCTL_AFD_SEND
IoControlCode=0000000000012017 // IOCTL_AFD_RECEIVE
IoControlCode=0000000000012017 // IOCTL_AFD_RECEIVE
```

As we can see our request `IOCTL_AFD_RECEIVE` appears twice. We can explain this by the fact that in our code, the `recv` function is executed in a loop. In practice, we retrieve the response in packets of `4096` bytes until we have received the entire TCP response. The first time we received the entire `HTTP` response and presumably `AFD.sys` returned information about how many bytes we actually received. And the second call with which we wanted to retrieve the rest returned us zero bytes, so no more requests were sent - a simple matter.

It's time to find a direct function that is responsible for handling this request, as in `AfdFastConnectionSend`. Let's check this statically using Binary Ninja:

```
1c0034be0    int64_t AfdFastIoDeviceControl(struct _FILE_OBJECT* FileObject,
1c0034be0      BOOLEAN Wait, PVOID InputBuffer, ULONG InputBufferLength,
1c0034be0      PVOID OutputBuffer, ULONG OutputBufferLength, ULONG IoControlCode,
1c0034be0      PIO_STATUS_BLOCK IoStatus, struct _DEVICE_OBJECT* DeviceObject) {
...
1c00354a6                   rbx = (uint64_t)AfdFastConnectionReceive(FsContext, &s,
1c00354a6                       rax_51, IoStatus);
...
1c0034be0    }
```

This time in the code we don't find a condition that directly checks if `IoControlCode == 0x12017`, what's more, before calling our target function we also have a number of checks that for now we don't know what they do. Let's take a breakpoint on this function:

```
12: kd> .foreach /pS 1 (ep { !process 0 0 afd_re.exe }) { bm /p ${ep}
afd!AfdFastConnectionReceive ".printf \"HIT!\\n\";gc;" }
  4: fffff801`6d9d3280 @!"afd!AfdFastConnectionReceive"
Couldn't resolve error at 'SessionId: afd!AfdFastConnectionReceive ".printf
\"HIT!\\n\";gc;" '
12: kd> g
HIT!
```

We only have one hit despite the fact that two `IOCTL_AFD_RECEIVE` requests went, this could mean that these check functions before calling `AfdFastConnectionReceive` check if, for example, the internal response buffer for the socket is empty.

We now turn to examining what our input buffer looks like for this request. Here, as usual, our invaluable sources (killvxk), (unknowncheats.me ICoded post), (ReactOS Project), (DynamoRIO / Dr. Memory), (DeDf), (diversenok) will help us.

```
10: kd> .foreach /pS 1 (ep { !process 0 0 afd_re.exe }) { bm /p ${ep}
afd!AfdFastConnectionReceive }
  6: fffff801`6d9d3280 @!"afd!AfdFastConnectionReceive"
Couldn't resolve error at 'SessionId: afd!AfdFastConnectionReceive '
10: kd> g
Breakpoint 6 hit
afd!AfdFastConnectionReceive:
fffff801`6d9d3280 4c894c2420      mov     qword ptr [rsp+20h],r9
4: kd> r
rax=0000000000000002 rbx=00000001ac3ae028 rcx=ffff8b05eaffb340
rdx=fffff58d13a4ef10 rsi=0000000000000001 rdi=0000000000000000
rip=fffff8016d9d3280 rsp=fffff58d13a4ee88 rbp=fffff58d13a4f4e0
 r8=0000000000001000  r9=fffff58d13a4f1c8 r10=fffff801d8817c70
r11=ffffb1fcd3800000 r12=ffff8b05eaffb340 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!AfdFastConnectionReceive:
fffff801`6d9d3280 4c894c2420      mov     qword ptr [rsp+20h],r9
ss:0018:fffff58d`13a4eea8=0000000000000003
4: kd> dq 00000001ac3ae028 L3
00000001`ac3ae028  00000001`ac3ae108 00000000`00000001
00000001`ac3ae038  00000000`00000020
4: kd> dq 00000001`ac3ae108 L2
00000001`ac3ae108  00000000`00001000 00000001`ac3ae260
4: kd> dq 00000001`ac3ae260 L10
00000001`ac3ae260  cccccccc`cccccccc cccccccc`cccccccc
```

```
00000001`ac3ae270  cccccccc`cccccccc cccccccc`cccccccc
00000001`ac3ae280  cccccccc`cccccccc cccccccc`cccccccc
00000001`ac3ae290  cccccccc`cccccccc cccccccc`cccccccc
00000001`ac3ae2a0  cccccccc`cccccccc cccccccc`cccccccc
00000001`ac3ae2b0  cccccccc`cccccccc cccccccc`cccccccc
00000001`ac3ae2c0  cccccccc`cccccccc cccccccc`cccccccc
00000001`ac3ae2d0  cccccccc`cccccccc cccccccc`cccccccc
```

We can see that essentially the structure of the input buffer is identical to the one we use to send packets (see part 3). With a slight difference, in our structure we have `AfdFlags`, which are flags describing our buffer. When they are set to `0x00` as in the case of sending then `AFD.sys` treats them as *send buffer*.

Analyzing retrieved data AfdFastConnectionReceive

Earlier we were guided by (diversenok) to guess what values the flags can take and nowhere there was a value `0x20`. We can instead look at (unknowncheats.me ICoded post), there we find such definitions:

```
#define TDI_RECEIVE_BROADCAST                          0x4
#define TDI_RECEIVE_MULTICAST                          0x8
#define TDI_RECEIVE_PARTIAL                                    0x10
#define TDI_RECEIVE_NORMAL                                     0x20
#define TDI_RECEIVE_EXPEDITED                          0x40
#define TDI_RECEIVE_PEEK                                       0x80
#define TDI_RECEIVE_NO_RESPONSE_EXP                    0x100
#define TDI_RECEIVE_COPY_LOOKAHEAD                     0x200
#define TDI_RECEIVE_ENTIRE_MESSAGE                     0x400
#define TDI_RECEIVE_AT_DISPATCH_LEVEL          0x800
#define TDI_RECEIVE_CONTROL_INFO                       0x1000
#define TDI_RECEIVE_FORCE_INDICATION           0x2000
#define TDI_RECEIVE_NO_PUSH                                    0x4000
```

That is, `0x20` would imply a normal reception of data from the driver. But there is a detail, according to (unknowncheats.me ICoded post), these flags apply to the `TdiFlags` field:

```
NTSTATUS AfdRecv(HANDLE SocketHandle, PVOID Buffer, ULONG_PTR BufferLength, PULONG_PTR pBytes)
{
        NTSTATUS Status;
        IO_STATUS_BLOCK IoStatus;
        AFD_SENDRECV_INFO RecvInfo;
        HANDLE Event;
        AFD_WSABUF AfdBuffer;

        Status = NtCreateEvent(&Event, EVENT_ALL_ACCESS, NULL, NotificationEvent, FALSE);
        if (NT_SUCCESS(Status))
        {
        ///
                AfdBuffer.len = (ULONG)BufferLength;

                RecvInfo.BufferArray = &AfdBuffer;
                RecvInfo.BufferCount = 1;
                RecvInfo.TdiFlags = TDI_RECEIVE_NORMAL;
                RecvInfo.AfdFlags = 0;
        ///
        }
        return Status;
}
```

Which in our case is not quite true. The buffer sent to `AFD.sys` is `0x18` in size, i.e. it has three fields of `0x8` bytes. And this third field (in our case `AfdFlags`) is just set to `0x20`. I have experimentally checked and our version is the one that works. Of course, I am not saying that the (unknowncheats.me ICoded post) version does not work, it just does not apply in our case.

With all this in mind, let us create a working proof-of-concept using everything we already have:

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

NTSTATUS createAfdSocket(PHANDLE socket) {/**/}
NTSTATUS bindAfdSocket(HANDLE socket) {/**/}
NTSTATUS connectAfdSocket(HANDLE socket) {/**/}
NTSTATUS sendAfdPacketTCP(HANDLE socket) {/**/}

NTSTATUS receiveAfdPacketTCP(HANDLE socket) {
    const int BUF_NUM = 1;
    const int BUF_SIZE = 1000;
    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, 0x00, BUF_SIZE);
        payload[i].len = BUF_SIZE;
    }

    AFD_SEND_PACKET* afdSendPacket = new AFD_SEND_PACKET;
    afdSendPacket->buffersArray = payload;
    afdSendPacket->buffersCount = BUF_NUM;
    afdSendPacket->afdFlags = 0x20; // RECEIVE_NORMAL

    IO_STATUS_BLOCK ioStatus;
    NTSTATUS status = NtDeviceIoControlFile(socket, NULL, NULL, NULL, &ioStatus,
IOCTL_AFD_RECEIVE,
                                            afdSendPacket, sizeof(AFD_SEND_PACKET),
                                            nullptr, 0);
    if (status == STATUS_PENDING) {
        WaitForSingleObject(socket, INFINITE);
        status = ioStatus.Status;
    }

    std::cout << "[+] SERVER RESPONSE: " << std::endl;
    std::cout << payload[0].buf << std::endl;

    return status;
}

int main() {
    HANDLE socket;
    // 1. Create socket
    // 2. Bind socket
    // 3. Connect to remote host
    // 4. Send 1000x'A'

    status = receiveAfdPacketTCP(socket);
    if (!NT_SUCCESS(status)) {
        std::cout << "[-] Could not receive TCP packet: " << std::hex << status <<
std::endl;
        return 1;
    }
    std::cout << "[+] Received!" << std::endl;

    return 0;
}
```

What we do. We allocate our buffers to store the received response somewhere, set `AfdFlags` to `0x20` (normal reception), and then send the request to `AFD.sys`. What more do you need?

Well, it would be useful to somehow find out how much of this data we have received. At first, I thought that maybe `AFD.sys` would modify the buffer structure and change its size. However, this did not happen. The answer was much simpler. The `IO_STATUS_BLOCK` structure has an `Information` field:

```
// ref: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_io_status_block
typedef struct _IO_STATUS_BLOCK {
  union {
    NTSTATUS Status;
    PVOID    Pointer;
  };
  ULONG_PTR Information;
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

And it is in the `Information` field that we get a return on how many bytes have been read, but we do not know how many are actually left to read. To check this we would now have to send another request to `AFD.sys` and check if `Informtaion` is equal to `0x0`. Then only then would we know if this is all there is.

Other receive flags

This question may be best answered by documentation from Microsoft. All the flags are very nicely described there. Although some of them are explained in terminology familiar to driver developers. I tried to reproduce some of them in Winsock and 'make up' my own explanation:

```
// Receive normal packets
afdSendPacket->afdFlags = TDI_RECEIVE_NORMAL;
// Receive normal packet, but don't clear AFD.sys input queue
afdSendPacket->afdFlags = TDI_RECEIVE_NORMAL | TDI_RECEIVE_PEEK;
// Receive normal packet, but wait for all data, equivalent of MSG_WAITALL in Winsock
afdSendPacket->afdFlags = TDI_RECEIVE_NORMAL | TDI_RECEIVE_NO_PUSH;
// Receive packets with tcp.flags.urg == 1
afdSendPacket->afdFlags = TDI_RECEIVE_EXPEDITED;
// Receive packets with tcp.flags.urg == 1, but don't clear AFD.sys input queue
afdSendPacket->afdFlags = TDI_RECEIVE_EXPEDITED | TDI_RECEIVE_PEEK;
// Receive packets with tcp.flags.urg == 1, but wait for all data, equivalent of MSG_WAITALL in Winsock
afdSendPacket->afdFlags = TDI_RECEIVE_EXPEDITED | TDI_RECEIVE_NO_PUSH;
```

Some of our flags relate to UDP and we will certainly look at this when the opportunity arises.

Next steps

At this point, we already have the necessary functionality to be able to create a simple TCP client. In the next batches we will look more at socket operations. How to change its parameters, how to close a connection, how to close a socket, how to handle other types of TCP messages.

Final code

Essentially, the content of the final code is no different from what you can find in the proof-of-concept above. Of course, the code for IPv6 will look identical.

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.