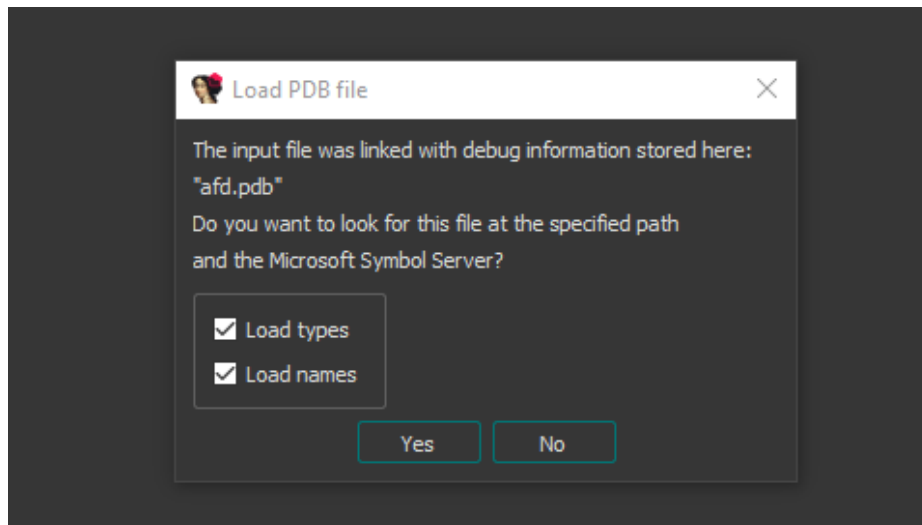


Driver Reverse Engineering 101 - Part I: Static Analysis

 eversinc33.com/2025/08/15/driver-reverse-engineering-101



Driver Reverse Engineering 101 - Part I: Static Analysis

Published : 2025-08-15 | Category : [Reverse Engineering Windows Drivers](#)

A few months ago, while hunting for vulnerable drivers to abuse for BYOVD on operations, I stumbled upon a repository of [47GBs of signed Windows drivers](#). After fuzzing them with [ioctlance](#), a symbolic execution based fuzzer, I was able to identify many previously unknown olddrivers. In this post I do not want to talk about exploiting drivers, but about how to approach reverse engineering of Windows WDM drivers. I often get asked questions on how to approach this, so I figured I might as well write it down once. The good news is: reversing IOCTL based WDM drivers (the most prevalent way drivers are developed) is very easy, as they always follow the same structure.

This is the dummy dummy explanation if your goal is to get reversing quickly. Of course I advise you to learn the basics of driver development, IOCTLs, IRPs and more, to really understand whats

happening here. But at the end of this tutorial, you should be able to get going with simple driver reverse engineering of IOCTL communications using IDA.

WTF is WDM?

Windows Driver Model ([WDM](#)) is the “old school” way of writing drivers. A lot of newer drivers are written using the Kernel Mode Driver Framework (KMDF), which takes care of lots of boilerplate code for the developer and is generally recommended. Still, a lot of drivers you will encounter in the wild are based on WDM.

A driver in the end is just a regular PE that is loaded and executed with kernel privileges, usually by creating a service. The basic skeleton of a WDM driver looks as follows:

First, we have a driver entry, where we usually create a device object and a symbolic link to it. This symbolic link can then be used by usermode processes to get a handle to our driver (e.g. by calling `CreateFile` on `\\??\BasicWdmLink`) and send messages (IOCTLs) to it to communicate (this is just one way for usermode to kernelmode communication, albeit the most common one). As usual, error handling and some code hidden for brevity:

```
#include <ntddk.h>

PDEVICE_OBJECT g_DeviceObject = NULL;
UNICODE_STRING g_DeviceName = RTL_CONSTANT_STRING(L"\\Device\\BasicWdmDevice");
UNICODE_STRING g_SymbolicLink = RTL_CONSTANT_STRING(L"\\??\\BasicWdmLink");

NTSTATUS
DriverEntry(
    _In_ PDRIVER_OBJECT DriverObject,
    _In_ PUNICODE_STRING RegistryPath
)
{
    // Create device
    status = IoCreateDevice(
        DriverObject,
        0,
        &g_DeviceName,
        FILE_DEVICE_UNKNOWN,
        0,
        FALSE,
        &g_DeviceObject
    );

    // Create symbolic link
    status = IoCreateSymbolicLink(&g_SymbolicLink, &g_DeviceName);
}
```

Usually, in this function the driver registers different dispatch routines. These describe what the driver does when its interacted with:

```
// Set dispatch routines
DriverObject->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = DispatchClose;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DispatchIoc
DriverObject->DriverUnload = DriverUnload;

return STATUS_SUCCESS;
}
```

These can be implemented as follows:

```

VOID
DriverUnload(
    _In_ PDRIVER_OBJECT DriverObject
)
{
    IoDeleteSymbolicLink(&g_SymbolicLink);
    IoDeleteDevice(DriverObject->DeviceObject);
}

NTSTATUS
DispatchCreate(
    _In_ PDEVICE_OBJECT DeviceObject,
    _Inout_ PIRP Irp
)
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

NTSTATUS
DispatchClose(
    _In_ PDEVICE_OBJECT DeviceObject,
    _Inout_ PIRP Irp
)
{
    UNREFERENCED_PARAMETER(DeviceObject);
    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

```

Most interesting is usually the IOCTL dispatcher routine though, as this handles the calls from a usermode program that sends an IOCTL via DeviceIoControl. The macro CTL_CODE is used to build a unique 32-bit value to identify an IOCTL, based on some options. More on that later. For now all you need to know is that an IOCTL code to us reverse engineers looks like a random 32 bit value, that actually encodes some information (which can be decoded e.g. with [OSR Ioctl Decoder](#)).

```

#define IOCTL_ECHO_DATA CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)

NTSTATUS
DispatchIoctl(
    _In_ PDEVICE_OBJECT DeviceObject,
    _Inout_ PIRP Irp
)
{
    // Get IOCTL code sent to our driver
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;

    switch (code) {
    case IOCTL_ECHO_DATA:
    {
        // HANDLE THE DATA FROM USERMODE
        break;
    }
    default:
        status = STATUS_INVALID_DEVICE_REQUEST;
        break;
    }

    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = info;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}

```

A usermode program can now send the ECHO_DATA IOCTL in our driver using the same CTL_CODE. In this case, we send a TestMessage string to our driver:

```

#define IOCTL_ECHO_DATA CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)

int main() {
    HANDLE hDevice = CreateFileW(
        L"\\\\.\\BasicWdmLink",
        GENERIC_READ | GENERIC_WRITE,
        0,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );

    const char input[] = "TestMessage";
    char output[64] = {0}; // buffer for the output, which the driver may write into
    DWORD bytesReturned = 0;

    DeviceIoControl(
        hDevice,
        IOCTL_ECHO_DATA,
        (LPVOID)input,
        sizeof(input),
        output,
        sizeof(output),
        &bytesReturned,
        NULL
    );
}

```

There are different methods for IOCTL communication, here METHOD_BUFFERED is used (which again, is very common). This essentially means, that a buffer is shared for both input and output of the operation.

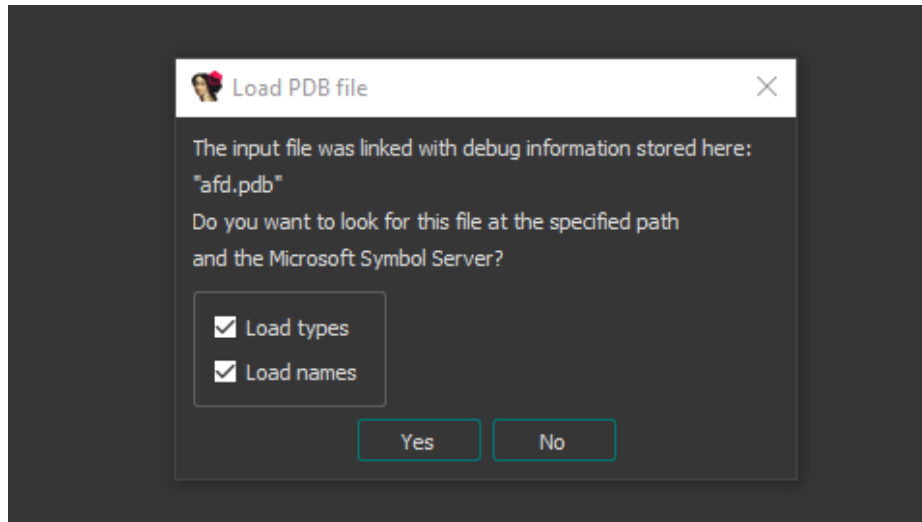
How can the driver access this? Through a huge union structure called IRP, which in turn contains another huge union, the IO_STACK_LOCATION. We will not dive deep here, but this will be important when reversing in IDA later, as we need to choose the right union depending on the method. See [here](#) and [here](#) for the struct definitions.

Essentially, this is enough knowledge to get going. So let's start disassembling a random driver:

Static Reverse Engineering

We are going to use the driver `afd.sys`, since this one will be present on your Windows version as well, so you can follow along. I chose this deliberately, but if you are wondering, this is the driver that is used for socket communication - so [malware can](#), instead of using the winsock API, talk to this driver directly via IOCTLs to create socket connections and send data.

Open `C:\Windows\system32\drivers\afd.sys` in IDA, and you will be asked if you want to resolve symbols from the MS Symbol Server:



Resolving symbols

Usually, we would say yes, because this makes reverse engineering almost like reading source code, but since our goal is to get a methodology that works regardless of the presence of symbols, we are going to say no here. After some analysis time, you should be greeted with the `DriverEntry` (think `main`) function. If you do not see pseudocode but disassembly, press `F5`:

```
IDA View-A | Pseudocode-A | Hex View-1
|NTSTATUS __stdcall DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
|{
|    sub_1C0087044();
|    return sub_1C00871F0(DriverObject, RegistryPath);
|}
```

Main decompilation

As you can see this is just some boilerplate wrapper to the actual main entry, which will be `sub_1C00871F0` in this case. You can see this, since the `DriverObject` and `RegistryPath` are passed to that function, and at least the `DriverObject` is needed for the initial setup of a classic IOCTL based driver.

If you double click this function and scroll down a little, you can see a call to the creation of a unicode string `\\Device\\Afd` and a call to `IoCreateDevice`. You can note down the device name, since this will be what we can open a handle to to send commands and potentially exploit our target driver:

```

stru_1C00293A0.CurrentIrp = 0LL;
sub_1C0039B88();
Timeout.QuadPart = -3000000000LL;
if ( MmIsVerifierEnabled(&VerifierFlags) >= 0 )
    Timeout.QuadPart *= 4LL;
NetioSetTriageBlock(2LL, &unk_1C0029270);
KeInitializeSpinLock((PKSPIN_LOCK)&stru_1C00293A0.DeviceExtension);
RtlInitUnicodeString(&DestinationString, L"\\Device\\Afd");
v3 = IoCreateDevice(DriverObject, 0, &DestinationString, 0x11u, 0x20000u, 0, &::DeviceObject);
v4 = v3;

```

IoCreateDevice

If you scroll down a bit, you will usually at one point find a block of code that looks like the following:

```

goto LABEL_44;
memset64(DriverObject->MajorFunction, (unsigned __int64)&sub_1C0054CC0, 0x1CuLL);
DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)&sub_1C005C790;
DriverObject->MajorFunction[15] = (PDRIVER_DISPATCH)&sub_1C0004A60;
DriverObject->MajorFunction[23] = (PDRIVER_DISPATCH)&sub_1C0049DC0;
DriverObject->FastIoDispatch = (PFAST_IO_DISPATCH)&unk_1C0029160;
DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_1C0040730;
::DeviceObject->Flags |= 0x10u;
::DeviceObject->StackSize = byte_1C0029253;

```

Dispatch routine registration

This is essentially the equivalent of our code block in the beginning, where we registered our dispatch routines, except that IDA does not resolve the numbers to the enums automatically.

```

// Set dispatch routines
DriverObject->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = DispatchClose;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DispatchIoc
DriverObject->DriverUnload = DriverUnload;

```

If you consult the following table, you should be able to spot the IRP_MJ_DEVICE_CONTROL (the “IOCTL Handler”) in `afd.sys`. Tip: click on a number and press `h` to convert from decimal to hex and vice versa

```

//
// Define the major function codes for IRPs.
//

#define IRP_MJ_CREATE                0x00
#define IRP_MJ_CREATE_NAMED_PIPE    0x01
#define IRP_MJ_CLOSE                 0x02
#define IRP_MJ_READ                   0x03
#define IRP_MJ_WRITE                  0x04
#define IRP_MJ_QUERY_INFORMATION     0x05
#define IRP_MJ_SET_INFORMATION       0x06
#define IRP_MJ_QUERY_EA              0x07
#define IRP_MJ_SET_EA                0x08
#define IRP_MJ_FLUSH_BUFFERS         0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
#define IRP_MJ_SET_VOLUME_INFORMATION 0x0b
#define IRP_MJ_DIRECTORY_CONTROL     0x0c
#define IRP_MJ_FILE_SYSTEM_CONTROL   0x0d
#define IRP_MJ_DEVICE_CONTROL        0x0e
#define IRP_MJ_INTERNAL_DEVICE_CONTROL 0x0f
#define IRP_MJ_SHUTDOWN              0x10
#define IRP_MJ_LOCK_CONTROL          0x11
#define IRP_MJ_CLEANUP               0x12
#define IRP_MJ_CREATE_MAILSLLOT      0x13
#define IRP_MJ_QUERY_SECURITY        0x14
#define IRP_MJ_SET_SECURITY          0x15
#define IRP_MJ_POWER                 0x16
#define IRP_MJ_SYSTEM_CONTROL        0x17
#define IRP_MJ_DEVICE_CHANGE         0x18
#define IRP_MJ_QUERY_QUOTA           0x19
#define IRP_MJ_SET_QUOTA             0x1a
#define IRP_MJ_PNP                   0x1b
#define IRP_MJ_PNP_POWER             IRP_MJ_PNP    // Obsolete....
#define IRP_MJ_MAXIMUM_FUNCTION      0x1b

```

IRP Major Function Codes

Since 0x0e resolves to 14 in decimal, our function of interest is sub_1C005C790. We can rename it to HandleIOCTL by pressing n. To add proper typing for parameters and return values, also cast it to PDRIVER_DISPATCH by pressing y:

```

goto LABEL_44;
memset64(DriverObject->MajorFunction, (unsigned __int64)&sub_1C0054CC0, 0x1CuLL);
DriverObject->MajorFunction[0xE] = HandleIOCTL;
DriverObject->MajorFunction[15] = (PDRIVER_DISPATCH)&sub_1C0004A60;
DriverObject->MajorFunction[23] = (PDRIVER_DISPATCH)&sub_1C0049DC0;
DriverObject->FastIoDispatch = (PFAST_IO_DISPATCH)&unk_1C0029160;
DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_1C0040730;
::DeviceObject->Flags |= 0x10u;
::DeviceObject->StackSize = byte_1C0029253;

```

Functions are not properly typed yet

The call to `memset64` is something you will often see: this usually just sets all routines to a stub that signals an unsupported routine, so that if an operation is not supported, the driver does not crash.

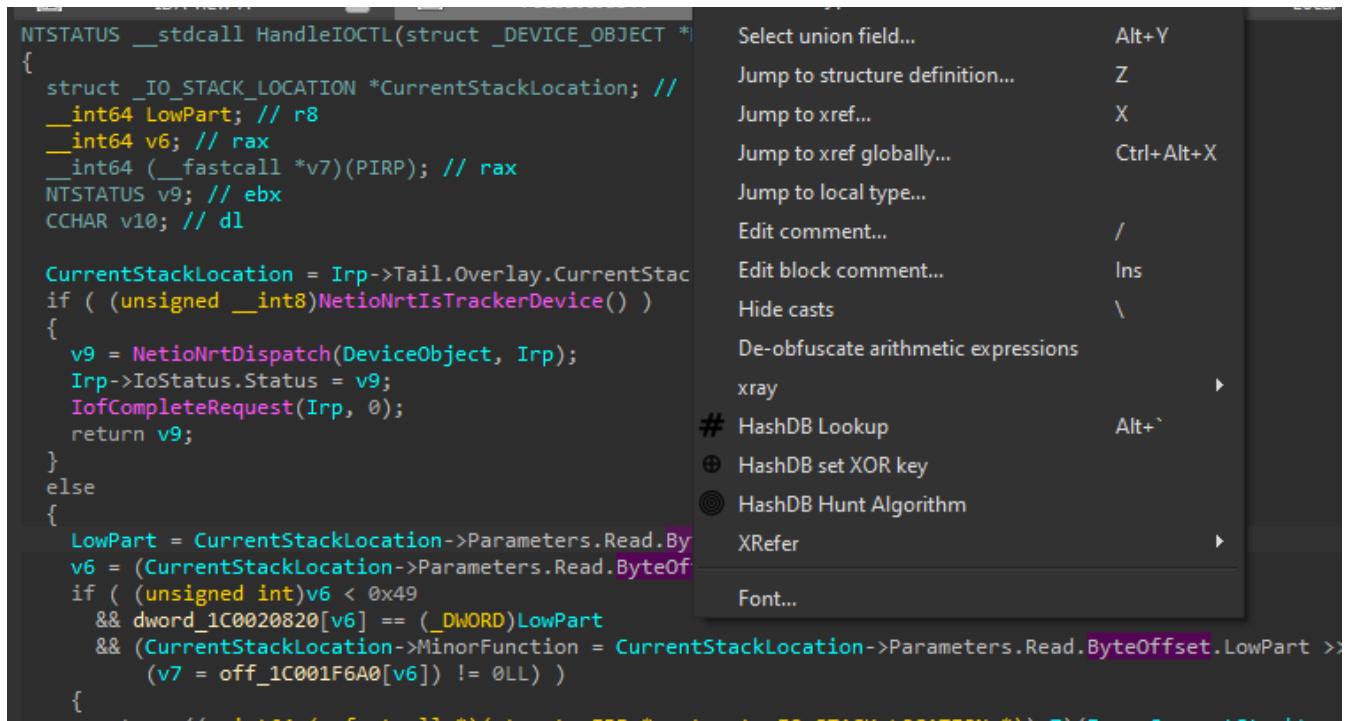
As an exercise, you can try finding out what the other functions being registered do. For now let us jump into the handler:

```
NTSTATUS __stdcall HandleIOCTL(struct _DEVICE_OBJECT *DeviceObject, struct _IRP *Irp)
{
    struct _IO_STACK_LOCATION *CurrentStackLocation; // rbx
    __int64 LowPart; // r8
    __int64 v6; // rax
    __int64 (__fastcall *v7)(PIRP); // rax
    NTSTATUS v9; // ebx
    CCHAR v10; // dl

    CurrentStackLocation = Irp->Tail.Overlay.CurrentStackLocation;
    if ( (unsigned __int8)NetioNrtIsTrackerDevice() )
    {
        v9 = NetioNrtDispatch(DeviceObject, Irp);
        Irp->IoStatus.Status = v9;
        IoCompleteRequest(Irp, 0);
        return v9;
    }
    else
    {
        LowPart = CurrentStackLocation->Parameters.Read.ByteOffset.LowPart;
        v6 = (CurrentStackLocation->Parameters.Read.ByteOffset.LowPart >> 2) & 0x3FF;
        if ( (unsigned int)v6 < 0x49
            && dword_1C0020820[v6] == (_DWORD)LowPart
            && (CurrentStackLocation->MinorFunction = CurrentStackLocation->Parameters.Read.ByteOffset.LowPart >> 2,
                (v7 = off_1C001F6A0[v6]) != 0LL) )
        {
            return ((__int64 (__fastcall *))(struct _IRP *, struct _IO_STACK_LOCATION *)v7)(Irp, CurrentStackLocation);
        }
        else
        {
            f
        }
    }
}
```

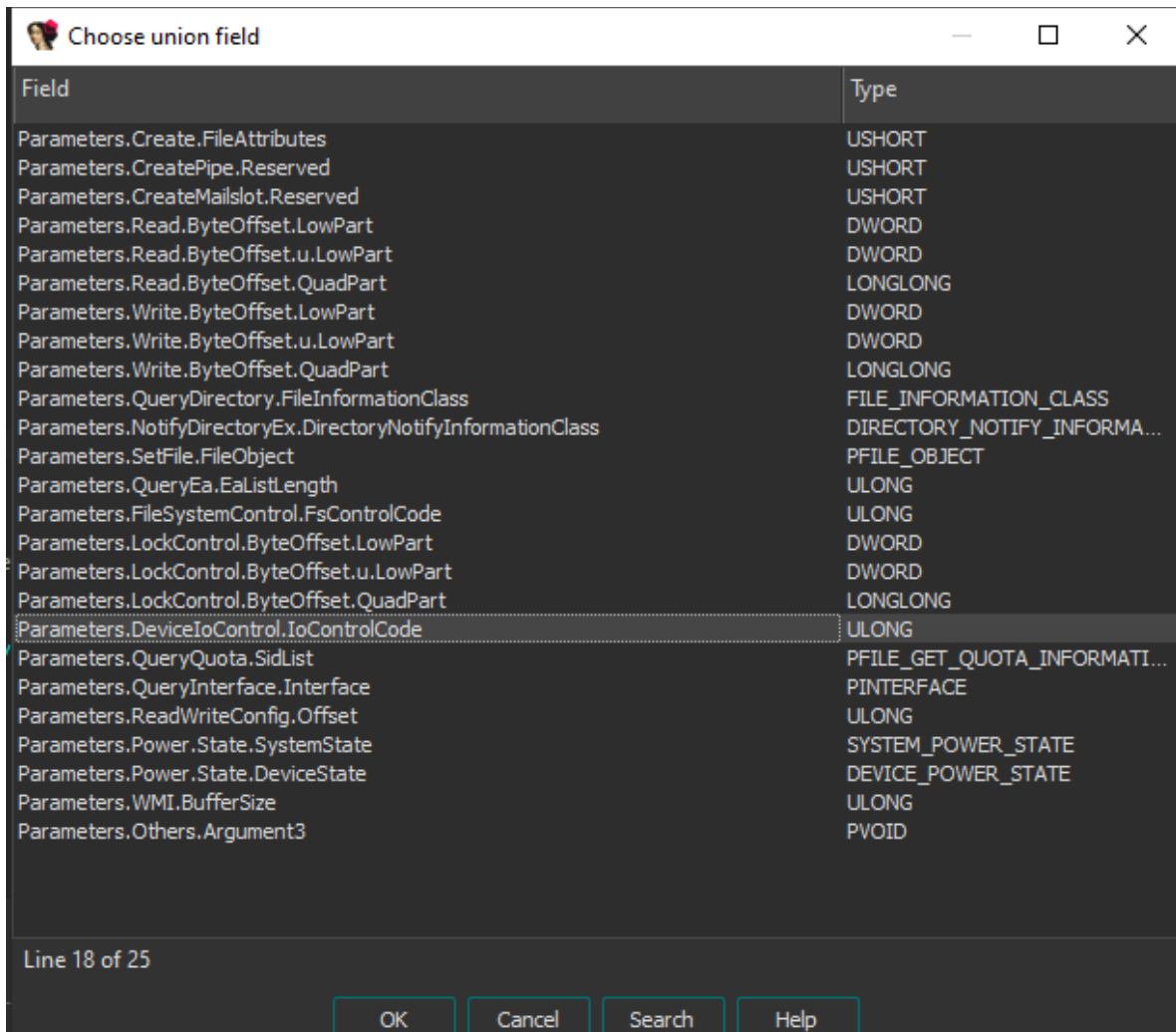
Handler

Now this might seem intimidating at first, but there is one trick which makes this a lot more readable. Do you remember how I told you earlier that an `_IO_STACK_LOCATION` is a massive union? If you are aware of unions, essentially they mean that one type can mean different things. How can IDA choose the right union? It can not, which is why you usually need to select the correct one by right clicking or pressing `alt+y` to select a union field where `CurrentStackLocation`, the `_IO_STACK_LOCATION` member of the `IRP` is used:



Selecting the right union

Since we are in an IOCTL handler, it is likely that this field references the `IoControlCode`:



Selecting the right union

Now the pseudocode makes a lot more sense. I renamed variables and added comments to explain what it does:

```

{
    ioctl_code = CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode;
    // Get function code from ioctl
    function_code = (CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode >> 2) & 0x3FF;
    if ( (unsigned int)function_code < 0x49 // bounds check
        && expected_ioctl_list[function_code] == (_DWORD)ioctl_code // verify ioctl is as expected
        && (CurrentStackLocation->MinorFunction = CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode >> 2,
            (v7 = ioctl_function_table[function_code]) != 0LL) ) // call function from function table
    {

```

Decompilation Markup

Essentially, it extracts the function code from the incoming IOCTL, verifies it is as expected by checking against a whitelist and finally calls a function from a function table. We can double click the function table you can see all the different functions (that can be called through this IOCTL handler) in an array:

```
.rdata:00000001C001F69F          db      0
.rdata:00000001C001F6A0  ioctl_function_table dq offset sub_1C0033C20
.rdata:00000001C001F6A8          ; DATA XREF: HandleIOCTL+60↓r
.rdata:00000001C001F6A8          dq offset sub_1C004E580
.rdata:00000001C001F6B0          dq offset sub_1C005BD20
.rdata:00000001C001F6B8          dq offset sub_1C005C5A0
.rdata:00000001C001F6C0          dq offset sub_1C004D270
.rdata:00000001C001F6C8          dq offset sub_1C0055E50
.rdata:00000001C001F6D0          dq offset sub_1C0057CF0
.rdata:00000001C001F6D8          dq offset sub_1C0056B60
.rdata:00000001C001F6E0          dq offset sub_1C00752B0
.rdata:00000001C001F6E8          dq offset sub_1C005CD50
.rdata:00000001C001F6F0          dq offset sub_1C0010EC0
.rdata:00000001C001F6F8          dq offset sub_1C00371F0
.rdata:00000001C001F700          dq offset sub_1C0010EC0
.rdata:00000001C001F708          dq offset sub_1C0010EC0
.rdata:00000001C001F710          dq offset sub_1C0010EC0
.rdata:00000001C001F718          dq offset sub_1C0010EC0
.rdata:00000001C001F720          dq offset sub_1C0010EC0
.rdata:00000001C001F728          dq offset sub_1C0010EC0
.rdata:00000001C001F730          dq offset sub_1C0010EC0
.rdata:00000001C001F738          dq offset sub_1C0010EC0
.rdata:00000001C001F740          dq offset sub_1C0010EC0
.rdata:00000001C001F748          dq offset sub_1C0010EC0
.rdata:00000001C001F750          dq offset sub_1C0010EC0
.rdata:00000001C001F758          dq offset sub_1C0010EC0
.rdata:00000001C001F760          dq offset sub_1C0010EC0
.rdata:00000001C001F768          dq offset sub_1C0010EC0
.rdata:00000001C001F770          dq offset sub_1C0010EC0
.rdata:00000001C001F778          dq offset sub_1C0010EC0
.rdata:00000001C001F780          dq offset sub_1C0010EC0
```

Whitelist

Now there's two ways in which IOCTL functions are called from an IOCTL handler (of course there are endless possibilities, but those are the two you will encounter most often):

- The function code is extracted from the IOCTL and used as an index to a table of functions
- A huge switch statement

Let us look at an example of the latter as well. For this we open `mountmgr.sys` and follow the exact same steps to end up in the IOCTL handler which has an `if/else/switch` statement handling different codes:

```

}
if ( ioctl <= 0x6D4020 )
{
    switch ( ioctl )
    {
        case 0x6D4020u:
ABEL_54:
            PsDetachSiloFromCurrentThread(v13);
            goto LABEL_7;
        case 0x6D0008u:
            v5 = "IOCTL_MOUNTMGR_QUERY_POINTS";
            v4 = sub_1C0009730((int)DeviceExtension, (int)Irp);
            goto LABEL_54;
        case 0x6D0030u:
            v5 = "IOCTL_MOUNTMGR_QUERY_DOS_VOLUME_PATH";
            v4 = sub_1C00091D0(DeviceExtension, Irp);
            goto LABEL_54;
        case 0x6D0034u:
            v5 = "IOCTL_MOUNTMGR_QUERY_DOS_VOLUME_PATHS";
            v4 = sub_1C0009E90(DeviceExtension, Irp);
            goto LABEL_54;
        case 0x6D003Cu:
            v5 = "IOCTL_MOUNTMGR_QUERY_AUTO_MOUNT";
            v4 = sub_1C0011568(DeviceExtension, Irp);
            goto LABEL_54;
        case 0x6D4008u:
            v5 = "IOCTL_MOUNTMGR_QUERY_POINTS_ADMIN";
            v4 = sub_1C0009730((int)DeviceExtension, (int)Irp);
            goto LABEL_54;
    }
ABEL_53:
    v5 = "IOCTL_UNKNOWN";
    v4 = -1073741808;
    goto LABEL_54;
}

```

Switch statement

From here you can either follow the function calls and see if you find vulnerabilities in themselves, or if you found a vulnerable IOCTL through fuzzing, you now know how to find it from the DriverEntry on.

If we look into one example function of this driver, there is yet again the IRP that is used to pass on information to the function call. And again, it is important to select the correct union:

```

1  __int64 __fastcall sub_1C0011568(__int64 a1, PIRP a2, __int64 a3)
2  {
3      ULONG_PTR v4; // rbx
4      struct_IRP *MasterIrp; // rdx
5      unsigned int v6; // edi
6
7      v4 = 0LL;
8      MasterIrp = a2->AssociatedIrp.MasterIrp;
9      v6 = 0;
10     if ( a2->Tail.Overlay.CurrentStackLocation->Parameters.Read.Length >= 4 )
11     {
12         LOBYTE(v4) = *(_BYTE *)(a1 + 192) != 0;
13         *(_DWORD *)&MasterIrp->Type = v4;
14         v4 = 4LL;
15     }
16     else
17     {
18         v6 = -1073741811;
19         if ( off_1C0006000 != (PDEVICE_OBJECT)&off_1C0006000 && (HIDWORD(off_1C0006000->Timer) & 1)
20             sub_1C000234C(off_1C0006000->AttachedDevice, 339LL, a3);
21     }
22     a2->IoStatus.Information = v4;
23     return v6;
24 }

```

Function decompilation

As you can see, IDA default selected MasterIrp in line 8, when accessing the IRP. However, this is usually not the right union. Most of the time, you would want to choose SystemBuffer here, which would be the buffer passed from userland and back when calling the IOCTL from userland:

```

1  __int64 __fastcall sub_1C0011568(__int64 deviceExtension, PIRP pIRP, __int64 a3)
2  {
3      ULONG_PTR v4; // rbx
4      _DWORD *SystemBuffer; // rdx
5      unsigned int v6; // edi
6
7      v4 = 0LL;
8      SystemBuffer = pIRP->AssociatedIrp.SystemBuffer;
9      v6 = 0;
10     if ( pIRP->Tail.Overlay.CurrentStackLocation->Parameters.DeviceIoControl.OutputBufferLength >
11     {
12         LOBYTE(v4) = *(_BYTE *)(deviceExtension + 192) != 0;
13         *SystemBuffer = v4;
14         v4 = 4LL;
15     }
16     else
17     {
18         v6 = 0xC000000D;
19         if ( off_1C0006000 != (PDEVICE_OBJECT)&off_1C0006000 && (HIDWORD(off_1C0006000->Timer) & 1)
20             sub_1C000234C(off_1C0006000->AttachedDevice, 339LL, a3);
21     }
22     pIRP->IoStatus.Information = v4;
23     return v6;
24 }

```

Correct union selected

You can try out different union members and see what makes sense or actually go methodological and parse the IOCTL number with a tool like OSR Ioctl Decoder and choose the right union based upon the Method:

Method (IoControlCode & 0x3)	IRP Union Member to Use
METHOD_BUFFERED (0)	Irp->AssociatedIrp.SystemBuffer
METHOD_IN_DIRECT (1)	Irp->AssociatedIrp.SystemBuffer (input), Irp->MdlAddress (output)
METHOD_OUT_DIRECT (2)	Irp->AssociatedIrp.SystemBuffer (input), Irp->MdlAddress (output)
METHOD_NEITHER (3)	Parameters.DeviceIoControl.Type3InputBuffer (input), Irp->UserBuffer (output)

Now this should be enough to get going.

If you want to actually learn exploiting drivers, I recommend playing around with [HackSys Extreme Vulnerable Driver](#).

Going Dynamic

One fallacy that novice reverse engineers (including myself) fall to early in their learnings, is that reverse engineering is looking at pseudo-C in IDA (or worse, disassembly) and renaming variables until you are basically at source code level. But most of the time, dynamic analysis is much more important, after you figured out the basic structure of the binary through static reversing.

This is a whole different topic, but there are many guides on that already. [Here](#) is a no bullshit guide on how to setup remote kernel debugging.

What you want to do is essentially:

- Setup 2 VMs (One Debugger, One Debugee)
- Enable Kernel Debugging on the Debugee
- Configure it to do remote kernel debugging (ideally via network) and connect back to your debugger VMs IP
- Run WinDbg on your Debugger VM

I hope this little intro gave you some tips on how to get started!

Happy Hacking!

[Windows Reverse Engineering Drivers](#)

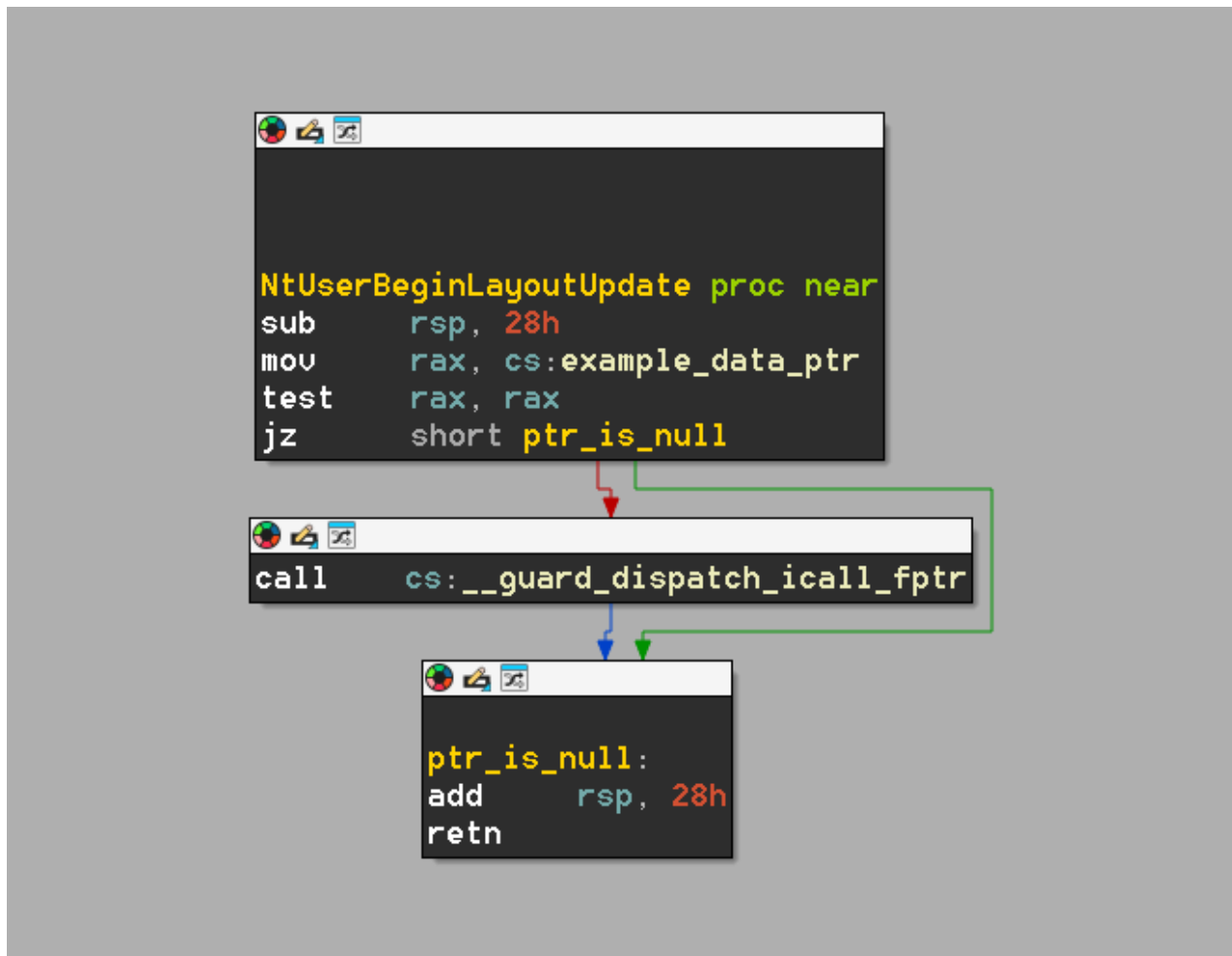
The image shows a debugger window on the left and IDA Pro on the right. The debugger window displays memory addresses and assembly instructions. A purple arrow points from a memory address in the debugger to a corresponding call instruction in IDA Pro.

```
0: kd> dp KSophon_x64+2D2388 L1
fffff806`603a2388 fffff806`60c704ee
0: kd> uf fffff806`60c704ee
KSophon_x64+0xba04ee:
fffff806`60c704ee 57          push     rdi
fffff806`60c704ef 57          push     rdi
fffff806`60c704f0 54          push     rsp
fffff806`60c704f1 51          push     rcx
fffff806`60c704f2 488d0df7be860b lea     rcx,[fffff806`6c4dc3f0]
fffff806`60c704f9 488b89382490f4 mov     rcx,qword ptr [rcx-0B6FDBC8h]
fffff806`60c70500 488d892aaa3850 lea     rcx,[rcx+5038AA2Ah]
fffff806`60c70507 48870c24    xchg   rcx,qword ptr [rsp]
fffff806`60c7050b c2180a     ret     18h
0: kd> dp fffff806`6c4dc3f0-0B6FDBC8h L1
fffff806`60dde828 fffff806`0bd35156
0: kd> ln fffff806`0bd35156+5038AA2Ah
Browse module
Set bu breakpoint
(fffff806`5c0bfb80) nt!PsGetCurrentProcess | (fffff806`5c0bfb80) nt
```

```
cs:qword_1408258C0, 0
mov     r14, rcx
jz     loc_1400F30E8
mov     r15, gs:188h
mov     rcx, r15
call   cs:qword_1402D2028
mov     rsi, rax
call   cs:qword_1402D2388
mov     rbp, rax
cmp    rsi, rax
jz     short loc_1400F3089
mov     rcx, rax
call   sub_1400D48A0
test   al, al
jz     short loc_1400F3089
call   cs:qword_1402D21B8
mov     rcx, rsi
mov     rdi, rax
call   cs:qword_1402D21C0
mov     rcx, rbp
mov     rbx, rax
```

[Prev:](#)

[Driver Reverse Engineering 101 - Part II: Unpacking a VMProtected Kernel Driver](#)



[Next:](#)

[\(Anti-\)Anti-Rootkit Techniques - Part III: Hijacking Pointers](#)