IOActive
Hardware | Software | Wetware
SECURITY SERVICES

Reverse Engineering & Bug Hunting on KMDF Drivers

Enrique Nissim

44CON
2018

ID

- Senior Consultant at IOActive
- Information System Engineer
- Infosec enthusiast (exploits, reversing, programming, pentesting, etc.)
- Conference speaking:
  - AsiaSecWest 2018
  - Ekoparty 2015-2016
  - CansecWest 2016
  - ZeroNights 2016
- @kiquenissim

Who

- Developers
  - If you write Windows drivers
- Security Consultants / Pentesters
  - If you need to audit Windows drivers
- Curious People?

What

- The focus will be on finding bugs and not on exploitation.
- This will highlight interesting functions and how to find them.
- See MSDN and references for full details on KMDF.

Why

- Several drivers were harmed during the process.
- Bugs were very easy to find.
- Some of them are in laptops since 2012.

Some bugs reported

- Intel CSI2 Host Controller:
  - 2 pool corruptions due to un-sanitized indexes
- Alps Touch Pad driver:
  - map and read from physical memory
  - read and write from IO ports
  - control over executive apis such as ObReferenceObjectByHandle
- Synaptics SynTP driver:
  - More than a dozen of kernel pointer leaks

Some bugs reported

- Intel Wireless Display:
  - Memory leak through WdfChildListAddOrUpdateChildDescriptionAsPresent
  - Out of bounds during string parsing
- Microsoft vwifibus driver:
  - Memory leak through WdfChildListAddOrUpdateChildDescriptionAsPresent
- Razer Synapse 3 – Rzudd Engine:
  - Multiple out of bounds due to bad WDF api usage.
- SteelSeries Engine ssdevfactory:
  - PDO duplication local DoS

Agenda

- Quick recap on WDM
  - Driver and Devices
  - Dispatch Routines
  - IRPs
  - IOCTLs
- Enter KMDF
  - Interfaces, IOQueues, Requests, ChildLists, Control Objects
  - kmdf-re.py
- Interesting functions and common errors
- Conclusions

Different Driver Models

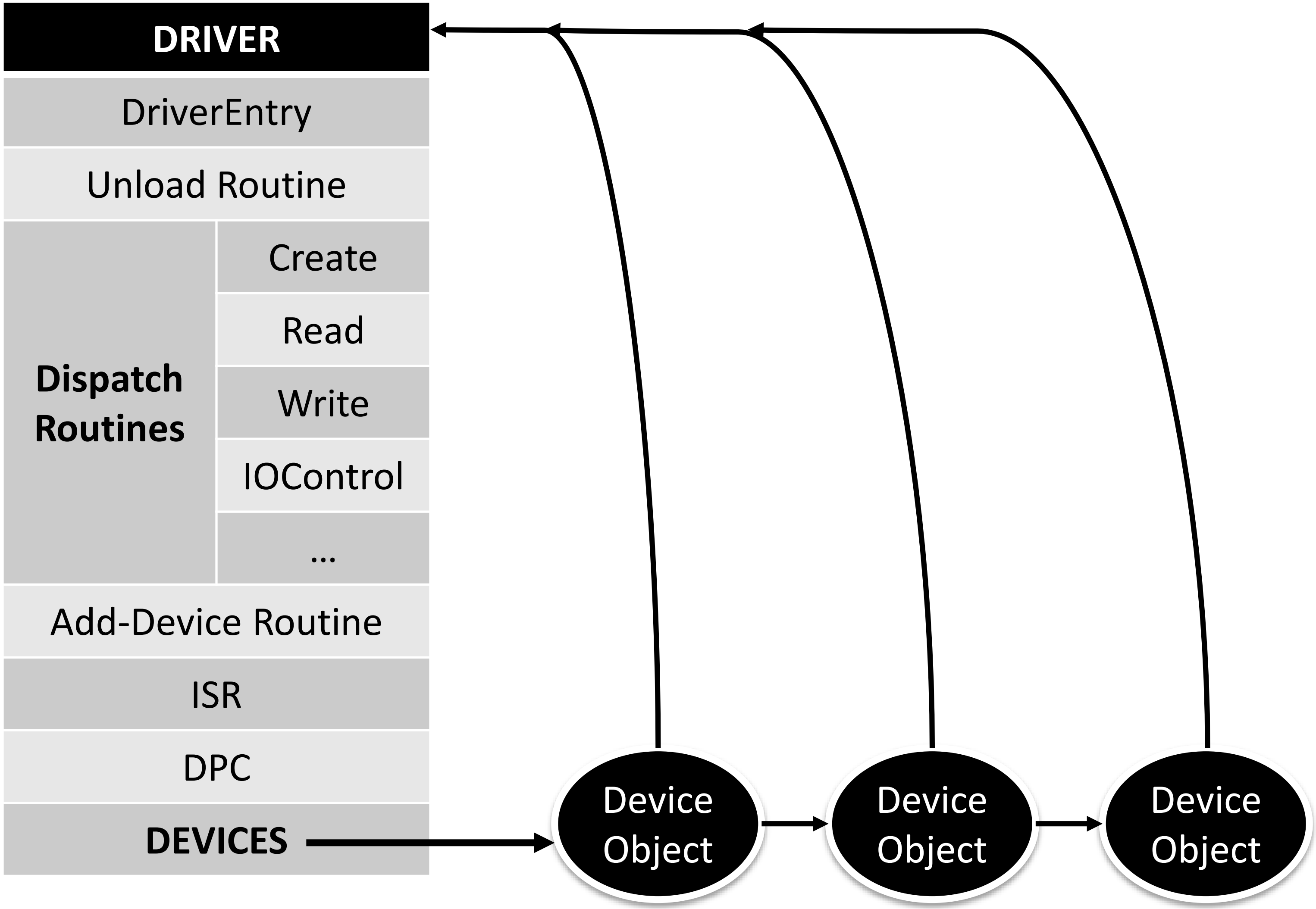
- WDM
- KMDF
- WDDM
- NDIS (miniport, filter, protocol)
- WFP
- Native 802.11
- WDI
- FileSystem and MiniFilter FS
- Portcl
- KS

Windows Driver Model

WDM

- The standard for all
- All models use WDM under the hood in one way or another
- Even though MS encourages the use of KMDF, knowledge of WDM is required to get most of it.
- Most vendors still use this one (except for bus and device drivers)

Driver and Device Objects



DRIVER

DriverEntry

Unload Routine

Dispatch Routines
- Create
- Read
- Write
- IOControl
- …

Add-Device Routine

ISR

DPC

DEVICES

Device Object

Device Object

Device Object

Creating the Device

```
NTKERNELAPI NTSTATUS IoCreateDevice(
  PDRIVER_OBJECT  DriverObject,
  ULONG           DeviceExtensionSize,
  PUNICODE_STRING DeviceName,
  DEVICE_TYPE     DeviceType,
  ULONG           DeviceCharacteristics,
  BOOLEAN         Exclusive,
  PDEVICE_OBJECT  *DeviceObject
);
```

- Most drivers specify only the **FILE_DEVICE_SECURE_OPEN** characteristic. This ensures that the same security settings are applied to any open request into the device's namespace.
- https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/controlling-device-namespace-access

Dispatch Routines

```
NTSTATUS SomeDispatchRoutine(
        PDEVICE_OBJECT DeviceObject,
        IN PIRP Irp
 );
```

- Drivers can set a single handler for all major functions and process the request based on the IRP Major code or set different dispatch routines for each case.

- The routine should validate the IRP parameters passed from user before using them blindly.

IRP Major Function Codes

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

Basic WDM Driver

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath) {
        PDEVICE_OBJECT DeviceObject = NULL;
        NTSTATUS Status = STATUS_UNSUCCESSFUL;
        UNICODE_STRING DeviceName, DosDeviceName = { 0 };

        RtlInitUnicodeString(&DeviceName, L"\\Device\\ZeroDriver");
        RtlInitUnicodeString(&DosDeviceName, L"\\DosDevices\\ZeroDriver");

        Status = IoCreateDevice(DriverObject, 0, &DeviceName, FILE_DEVICE_UNKNOWN,
                NULL, FALSE, &DeviceObject);

        DriverObject->MajorFunction[IRP_MJ_CREATE] = IrpCreateHandler;
        DriverObject->MajorFunction[IRP_MJ_READ] = IrpReadHandler;
        DriverObject->MajorFunction[IRP_MJ_WRITE] = IrpWriteHandler;
        DriverObject->MajorFunction[IRP_MJ_CLOSE] = IrpCloseHandler;
        DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IrpDeviceIoCtlHandler;
        DriverObject->DriverUnload = IrpUnloadHandler;

        // Create the symbolic link / Expose to User
        Status = IoCreateSymbolicLink(&DosDeviceName, &DeviceName);
        return Status;
}
```

Talking to the Driver

```
void TestDriver_X() {
        char bufferOut[256] = { 0 };
        char bufferIn[256] = { 0 };
        HANDLE hDevice = CreateFileW(L"\\\\.\\ZeroDriver\\",
                FILE_READ_ACCESS|FILE_WRITE_ACCESS,
                FILE_SHARE_READ|FILE_SHARE_WRITE, NULL,
                OPEN_EXISTING, 0, NULL);

        DWORD bytesRead, bytesWritten, bytesReturned;

        ReadFile(hDevice, &bufferOut, sizeof(bufferOut), &bytesRead, NULL);

        WriteFile(hDevice, bufferIn, sizeof(bufferIn), &bytesWritten, NULL);

        DeviceIoControl(hDevice, 0x88883000, bufferIn, sizeof(bufferIn),
                bufferOut, sizeof(bufferOut), &bytesReturned, NULL);

        CloseHandle(hDevice);
        return;
}
```

Syscalls to talk to Drivers (1/2)

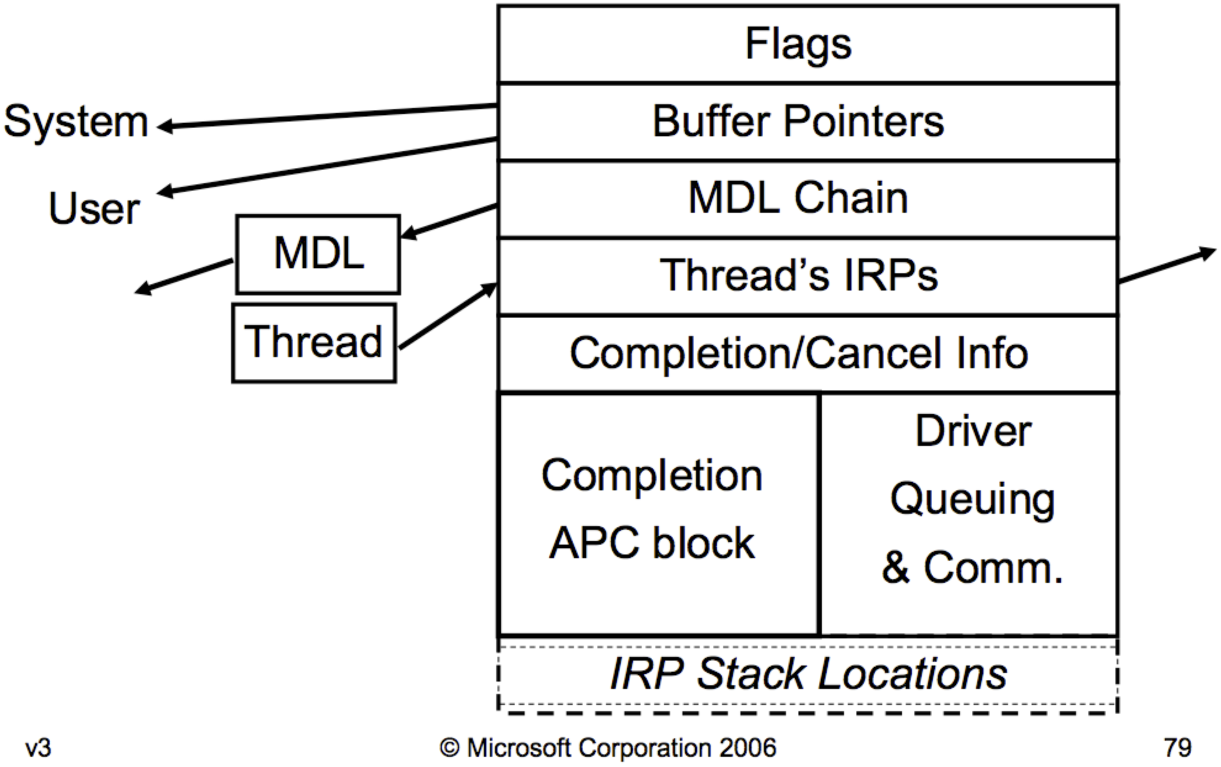
NtCreateFile	DispatchCreate
NtCreateNamedPipeFile	DispatchCreateNamedPipe
NtCloseHandle	DispatchClose
NtReadFile	DispatchRead
NtWriteFile	DispatchWrite
NtQueryInformationFile	DispatchQueryInformation
NtSetInformationFile	DispatchSetInformation
NtQueryEaFile	DispatchQueryEA
NtFlushBuffersFile	DispatchFlushBuffers
NtQueryVolumeInformationFile	DispatchQueryVolumeInformation
NtSetVolumeInformationFile	DispatchSetVolumeInformation

Syscalls to talk to Drivers (2/2)

NtQueryDirectoryFile	DispatchDirectoryControl
Ntfscontrolfile	DispatchFileSystemControl
NtDeviceIoControlFile	DispatchDeviceIOControl
NtShutdownSystem	DispatchShutdown
NtLockFile/NtUnlockFile	DispatchLockControl
NtCreateMailSlotFlie	DispatchCreateMailslot
NtQuerySecurityObject	DispatchQuerySecurity
NtSetSecurityObject	DispatchSetSecurity
NtQueryQuotaInformationFile	DispatchQueryQuota
NtSetQuotaInformationFile	DispatchSetQuota

Interrupt Request Packets

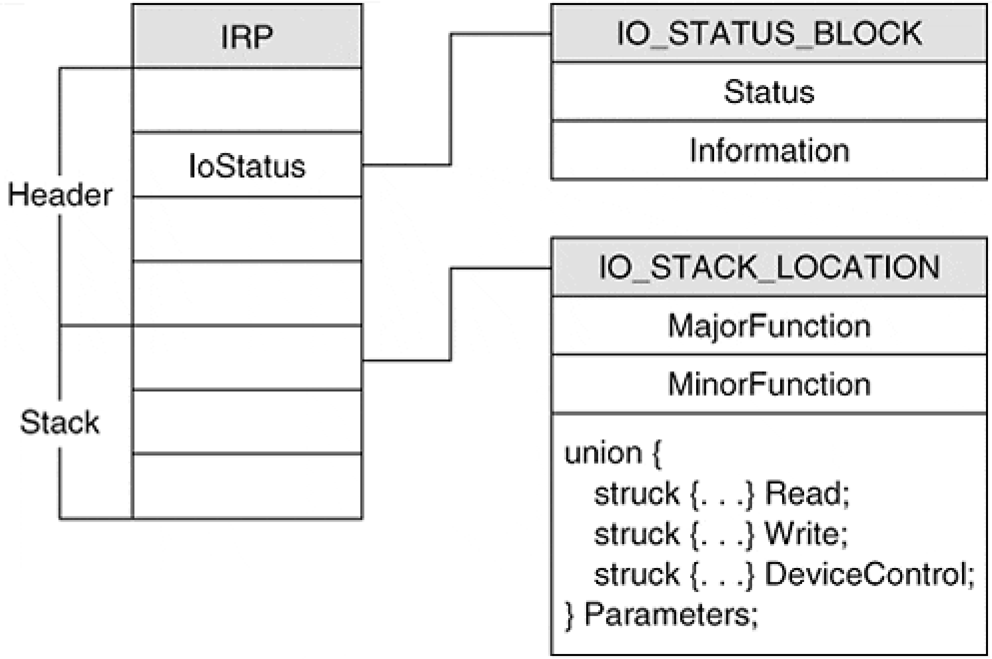
- Structure created by the IO manager that holds the information for the IO Request.



v3 © Microsoft Corporation 2006 79

Stack Locations

- The I/O manager creates an array of I/O stack locations for each IRP, with an array element corresponding to each driver in a chain of layered drivers.



```
union {
   struck {. . .} Read;
   struck {. . .} Write;
   struck {. . .} DeviceControl;
} Parameters;
```

Buffer Access Methods (1/3)

- **BUFFERED:** The IO manager creates intermediate buffers that it shares with the driver.

- **DIRECT IO:** The IO manager locks the buffer space into physical memory, and then provides the driver with direct access to the buffer space.

- **NEITHER:** The IO manager provides the driver with the virtual addresses of the request's buffer space. The IO manager does not validate the request's buffer space, so the driver must verify that the buffer space is accessible and lock the buffer space into physical memory.

Buffer Access Methods (2/3)

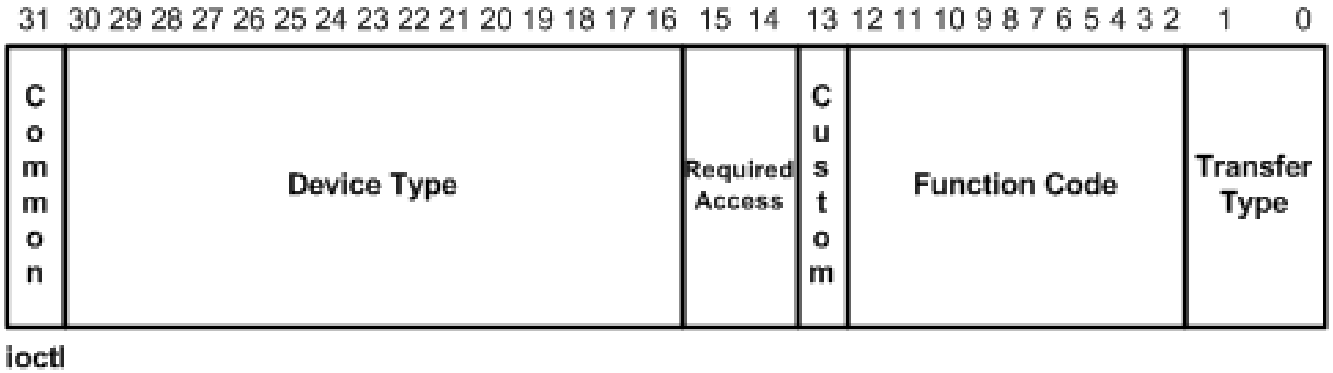
- The buffering flags affects the following operations:
  - IRP_MJ_READ
  - IRP_MJ_WRITE
  - IR_MJ_QUERY_EA
  - IR_MJ_SET_EA
  - IRP_MJ_DIRECTORY_CONTROL
  - IRP_MJ_QUERY_QUOTA
  - IRP_MJ_SET_QUOTA

Buffer Access Methods (3/3)

- For IO-Control Operations, the method is encoded in the IOCTL Code argument:
  - IRP_MJ_FILE_SYSTEM_CONTROL
  - **IRP_MJ_DEVICE_CONTROL**
  - IRP_MJ_INTERNAL_DEVICE_CONTROL

IOCTL Code

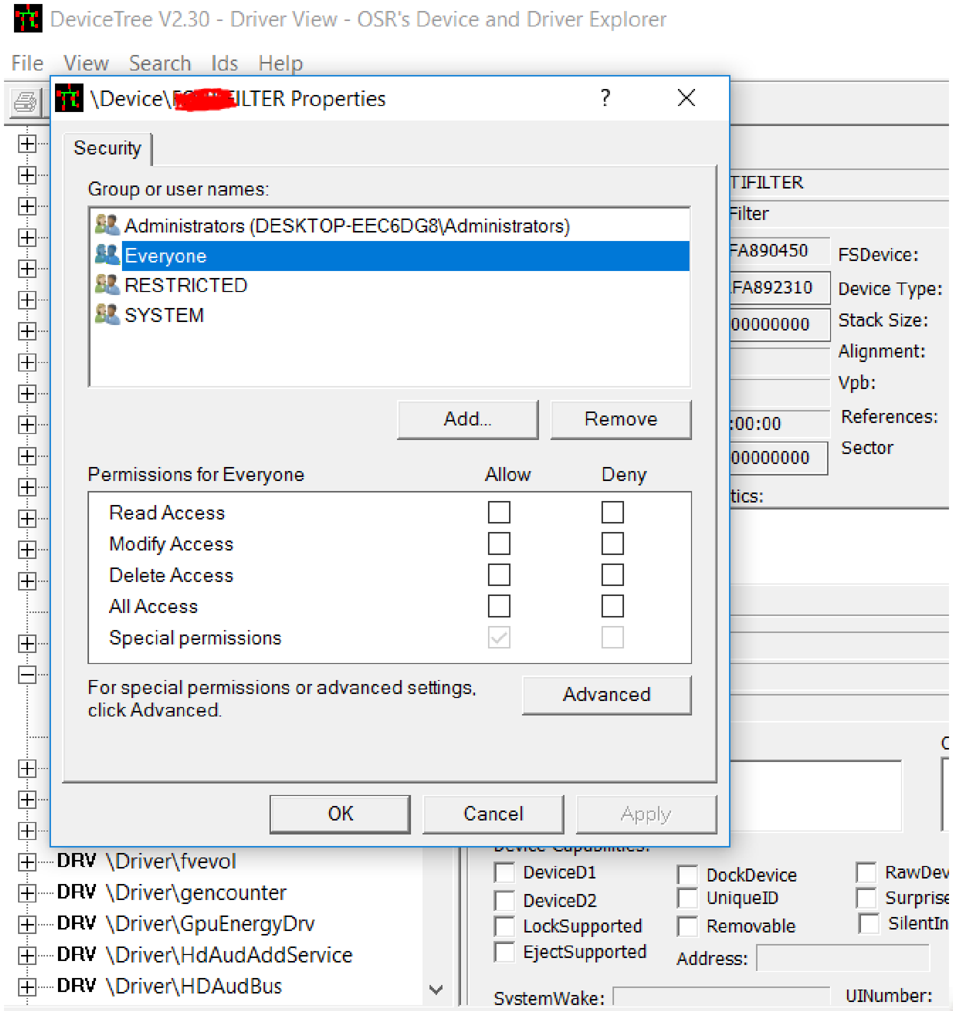
- An IOCTL code is a combination of values packed into a DWORD:



- **TransferType:** dictates how the IOManager will make the buffers available to the driver and what checks it performs on them.
- **RequiredAccess**: the right access required by the IOCTL; This is checked against the access rights we used for opening the device handle.

IOCTL Code

- #define FILE_ANY_ACCESS 0
- #define FILE_READ_ACCESS 1
- #define FILE_WRITE_ACCESS 2



\Device\[redacted]ILTER Properties

Security

Group or user names:

- Administrators (DESKTOP-EEC6DG8\Administrators)
- Everyone
- RESTRICTED
- SYSTEM

Permissions for Everyone	Allow	Deny
Read Access	☐	☐
Modify Access	☐	☐
Delete Access	☐	☐
All Access	☐	☐
Special permissions	☑	☐

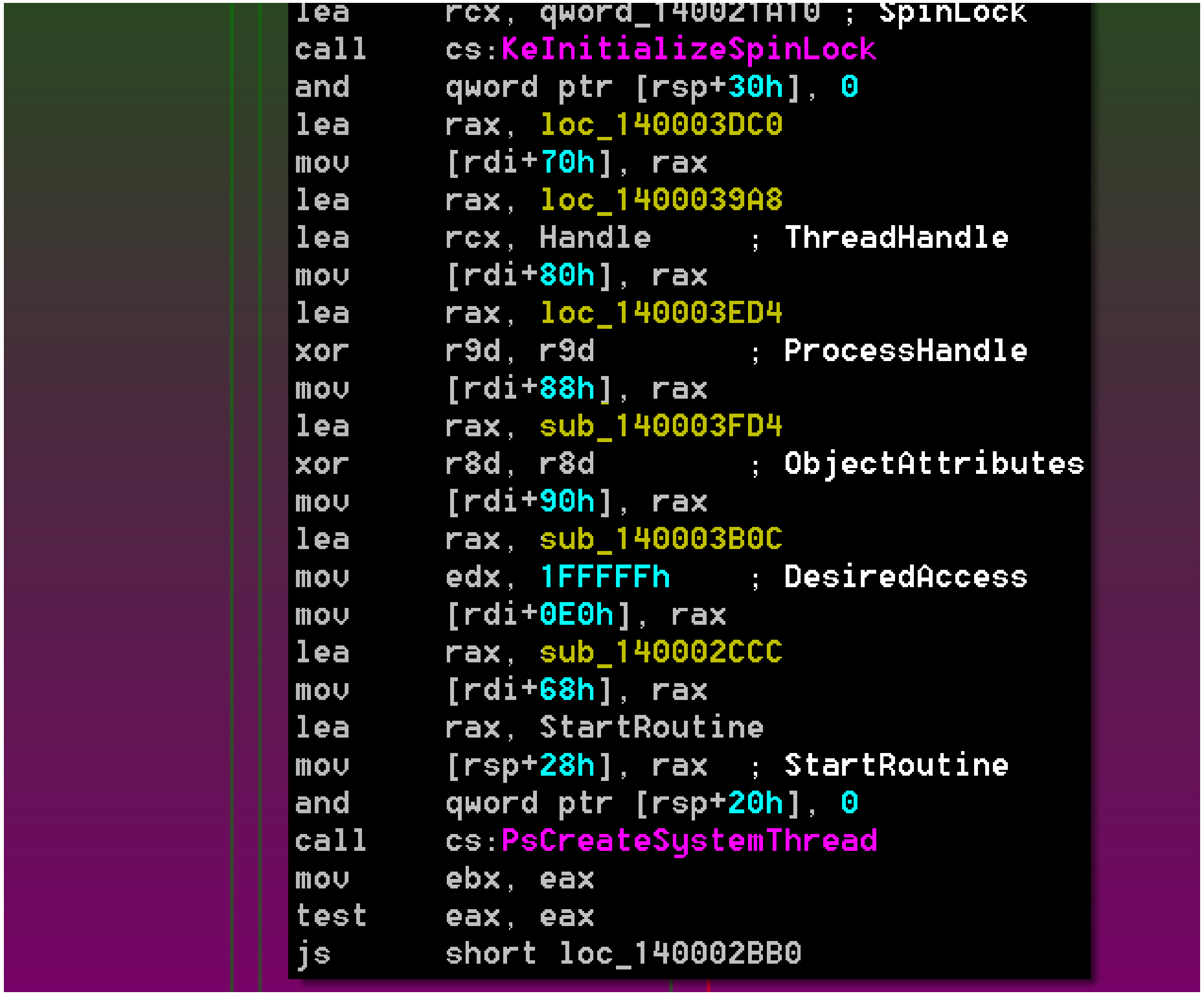
For special permissions or advanced settings, click Advanced.

TIFILTER
Filter
FA890450 FSDevice:
FA892310 Device Type:
00000000 Stack Size:
Alignment:
Vpb:
:00:00 References:
00000000 Sector
tics:

DRV \Driver\fvevol
DRV \Driver\gencounter
DRV \Driver\GpuEnergyDrv
DRV \Driver\HdAudAddService
DRV \Driver\HDAudBus

Device Capabilities:
☐ DeviceD1 ☐ DockDevice ☐ RawDev
☐ DeviceD2 ☐ UniqueID ☐ Surprise
☐ LockSupported ☐ Removable ☐ SilentIn
☐ EjectSupported Address:
SystemWake: UINumber:

Looking for Dispatch Routines



```
lea     rcx, qword_140021A10 ; SpinLock
call    cs:KeInitializeSpinLock
and     qword ptr [rsp+30h], 0
lea     rax, loc_140003DC0
mov     [rdi+70h], rax
lea     rax, loc_1400039A8
lea     rcx, Handle      ; ThreadHandle
mov     [rdi+80h], rax
lea     rax, loc_140003ED4
xor     r9d, r9d         ; ProcessHandle
mov     [rdi+88h], rax
lea     rax, sub_140003FD4
xor     r8d, r8d         ; ObjectAttributes
mov     [rdi+90h], rax
lea     rax, sub_140003B0C
mov     edx, 1FFFFFh     ; DesiredAccess
mov     [rdi+0E0h], rax
lea     rax, sub_140002CCC
mov     [rdi+68h], rax
lea     rax, StartRoutine
mov     [rsp+28h], rax   ; StartRoutine
and     qword ptr [rsp+20h], 0
call    cs:PsCreateSystemThread
mov     ebx, eax
test    eax, eax
js      short loc_140002BB0
```

Common Issues in WDM

- What can go wrong? → A lot, check "Windows Drivers Attack Surface" by Ilja Van Sprundel

Kernel Mode Driver Framework

KMDF Overview

- KMDF provides an abstraction on top of WDM that simplifies driver development.
- More difficult to find the booty from a RE perspective.
- New drivers are written using KMDF.
- It got open sourced three years ago: https://github.com/Microsoft/Windows-Driver-Frameworks

KMDF Overview

- KMDF establishes its own dispatch routines that intercept all IRPs that are sent to the driver.
- For read, write, device I/O control, and internal device I/O control requests, the driver creates one or more queues and configures each queue to receive one or more types of I/O requests.
- The framework creates a WDFREQUEST object to represent the request and adds it to the queue

IO Queue

```
typedef struct _WDF_IO_QUEUE_CONFIG {
  ULONG                                   Size;
  WDF_IO_QUEUE_DISPATCH_TYPE                  DispatchType;
  WDF_TRI_STATE                           PowerManaged;
  BOOLEAN                                 AllowZeroLengthRequests;
  BOOLEAN                                 DefaultQueue;
  PFN_WDF_IO_QUEUE_IO_DEFAULT                 EvtIoDefault;
  PFN_WDF_IO_QUEUE_IO_READ                    EvtIoRead;
  PFN_WDF_IO_QUEUE_IO_WRITE                   EvtIoWrite;
  PFN_WDF_IO_QUEUE_IO_DEVICE_CONTROL          EvtIoDeviceControl;
  PFN_WDF_IO_QUEUE_IO_INTERNAL_DEVICE_CONTROL EvtIoInternalDeviceControl;
  PFN_WDF_IO_QUEUE_IO_STOP                    EvtIoStop;
  PFN_WDF_IO_QUEUE_IO_RESUME                  EvtIoResume;
  PFN_WDF_IO_QUEUE_IO_CANCELED_ON_QUEUE       EvtIoCanceledOnQueue;
  union {
    struct {
      ULONG NumberOfPresentedRequests;
    } Parallel;
  } Settings;
  WDFDRIVER                               Driver;
} WDF_IO_QUEUE_CONFIG, *PWDF_IO_QUEUE_CONFIG;
```

KMDF-WDM Equivalents (1/2)

- Driver Object → WDFDriver
- Device Object → WDFDevice
- Device Extension → Object Context
- IRP → WDFRequest
- Dispatch Routines → IOQueue Handlers
- IO Stack Location → WDFRequest Params

KMDF-WDM Equivalents (2/2)

DispatchCleanup	EvtFileCleanup
DispatchClose	EvtFileClose
DispatchCreate	EvtDeviceFileCreate or EvtIoDefault
DispatchDeviceControl	EvtIoDeviceControl or EvtIoDefault
DispatchInternalDeviceControl	EvtIoInternalDeviceControl or EvtIoDefault
DispatchRead	EvtIoRead or EvtIoDefault
DispatchWrite	EvtIoWrite or EvtIoDefault
Others	EvtDeviceWdmIrpPreprocess

IOActive
Hardware | Software | Wetware
SECURITY SERVICES

A basic KMDF driver (1/2)

```
NTSTATUS DriverEntry(
        IN PDRIVER_OBJECT  DriverObject,
        IN PUNICODE_STRING  RegistryPath
        )   {
  WDF_DRIVER_CONFIG config;
  NTSTATUS status;

  WDF_DRIVER_CONFIG_INIT(&config, EvtDeviceAdd);
  status = WdfDriverCreate(DriverObject, RegistryPath,
              WDF_NO_OBJECT_ATTRIBUTES, &config, WDF_NO_HANDLE);

  if(!NT_SUCCESS(status))
    KdPrint((__DRIVER_NAME "WdfDriverCreate failed with status 0x%08x\n", status));

   return status;
}
```

A basic KMDF driver (2/2)

```
NTSTATUS EvtDeviceAdd( IN WDFDRIVER  Driver, IN PWDFDEVICE_INIT
DeviceInit )
{
  NTSTATUS status;
  WDFDEVICE device;
  PDEVICE_CONTEXT devCtx = NULL;
  WDF_OBJECT_ATTRIBUTES attributes;
  WDF_IO_QUEUE_CONFIG ioQConfig;

  WdfDeviceInitSetIoType(DeviceInit, WdfDeviceIoDirect);
  WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&attributes,
                                          DEVICE_CONTEXT);
  status = WdfDeviceCreate(&DeviceInit,  &attributes, &device);

[..]
```

A basic KMDF driver (3/3)

```
[..]
  devCtx = GetDeviceContext(device);
  WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(&ioQConfig,
                                 WdfIoQueueDispatchSequential);

  ioQConfig.EvtIoDefault = EvtDeviceIoDefault;
  status = WdfIoQueueCreate(
                        device,
                        &ioQConfig,
                       WDF_NO_OBJECT_ATTRIBUTES,
                       &devCtx->IoDefaultQueue);

  status = WdfDeviceCreateDeviceInterface(device,
                            &GUID_DEV_ZERO, NULL);
  return status;
}
```

KMDF DriverEntry

- Our DriverEntry will actually be wrapped by a KMDF-DriverEntry which will bind to an specific wdf library version and then call to our DriverEntry.

```
NTSTATUS
WdfVersionBind(
    __in    PDRIVER_OBJECT DriverObject,
    __in    PUNICODE_STRING RegistryPath,
    __inout PWDF_BIND_INFO BindInfo,
    __out   PWDF_COMPONENT_GLOBALS*
ComponentGlobals
    );
```

KMDF DriverEntry

```
typedef struct _WDF_BIND_INFO {
        ULONG              Size;
        PWCHAR             Component;
        WDF_VERSION        Version;
        ULONG              FuncCount;
        PVOID              FuncTable;
        PVOID    Module;
} WDF_BIND_INFO, * PWDF_BIND_INFO;
```

Device Interfaces

- As KMDF is mostly used for device drivers, and hardware can appear and disappear dynamically (PnP), it is common to create interfaces based on GUIDs rather than on names.

```
NTSTATUS WdfDeviceCreateDeviceInterface(
  WDFDEVICE          Device,
  CONST GUID         *InterfaceClassGUID,
  PCUNICODE_STRING ReferenceString
);
```

Device Interfaces

- TeeDriverW8X64
    - \\?\pci#ven_8086&dev_a13a&subsys_1c5d1043&rev_31#3&11583659&1&b0#{e2d1ff34-3458-49a9-88da-8e6915ce9be5}
- IntcAud.sys
    - \\?\hdaudio#func_01&ven_8086&dev_2809&subsys_80860101&rev_1000#4&5e29a79&0&0201#{86841137-ed8e-4d97-9975-f2ed56b4430e}\intazaudprivateinterface
- I’ve also found that WinObj doesn’t show the reference strings!,, it only shows one instance.. You need to go manually or use ObjDir.

IOActive
Hardware | Software | Wetware
SECURITY SERVICES

Device Interfaces

- The ReferenceString parameter allows to have multiple instances of an interface.
- For most device types and characteristics, the default security descriptor gives read/write/execute access to everyone.
- If explicit permissions are set, we still need to check the ACL and determine if a handle can be opened without read/write permissions and work with the IOCTLs that are ANY_ACCESS

Using Device Interfaces

```
void GetInterfaceDevicePath(GUID *guid) {
  DWORD requiredSize;
  int MemberIdx = 0;
  HDEVINFO hDeviceInfoset = SetupDiGetClassDevs(guid, NULL, 0, DIGCF_DEVICEINTERFACE | DIGCF_PRESENT);
  if (hDeviceInfoset != INVALID_HANDLE_VALUE) {
  SP_DEVICE_INTERFACE_DATA  DeviceInterfaceData = { 0 };
  DeviceInterfaceData.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);
  while (SetupDiEnumDeviceInterfaces(hDeviceInfoset, NULL, guid, MemberIdx, &DeviceInterfaceData)) {
     MemberIdx++;
     SP_DEVINFO_DATA DeviceInfoData = { 0 };
     DeviceInfoData.cbSize = sizeof(SP_DEVINFO_DATA);
     SetupDiGetDeviceInterfaceDetail(hDeviceInfoset, &DeviceInterfaceData, NULL, 0, &requiredSize, NULL);
     SP_DEVICE_INTERFACE_DETAIL_DATA *DevIntfDetailData = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
requiredSize);
     DevIntfDetailData->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
     if (SetupDiGetDeviceInterfaceDetail(hDeviceInfoset, &DeviceInterfaceData,
            DevIntfDetailData, requiredSize, &requiredSize, &DeviceInfoData)) {
       printf("DevicePath: %S\n", (TCHAR*)DevIntfDetailData->DevicePath);
     }
     HeapFree(GetProcessHeap(), 0, DevIntfDetailData);
  }
  SetupDiDestroyDeviceInfoList(hDeviceInfoset);
 }
}
```

KMDF and Buffer Access

```
NTSTATUS WdfRequestRetrieveInputBuffer(
  WDFREQUEST Request,
  size_t     MinimumRequiredLength,
  PVOID        *Buffer,
  size_t     *Length
);
```

- **Length** must be checked before dereferencing beyond MinimumRequiredLength

KMDF and Buffer Access

- *WdfRequestRetrieveInputBuffer*
- *WdfRequestRetrieveOutputBuffer*

- *WdfRequestRetrieveInputWdmMdl*
- *WdfRequestRetrieveOutputWdmMdl*

You can call WdfRequestRetrieveInputBuffer or WdfRequestRetrieveInputWdmMdl for either DIRECT OR BUFFERED TransferTypes!... What the framework does in each case depends on the TransferType. For instance, calling WdfRequetRetrieveInputBuffer when using DIRECT IO will return the VirtualAddress of the MDL allocated by the IOManager.

Calling WdfRequestRetrieveInputWdmMDL when using BUFFERED type will allocate a new MDL over the kernel pool buffer and return that to the caller.

Method NEITHER

- KMDF doesn’t want you to use method neither.

- To use it you need to access it in an EvtIoInCallerContext Callback and use:
    - ***WdfRequestRetrieveUnsafeUserInputBuffer***
    - ***WdfRequestRetrieveUnsafeUserOutputBuffer***

    - ***WdfRequestProbeAndLockUserBufferForRead***
    - ***WdfRequestProbeAndLockUserBufferForWrite***

Non-PnP KMDF

Non-PnP Drivers

- The driver set the WdfDriverInitNonPnpDriver flag in the WDF_DRIVER_CONFIG.
- Provide an EvtDriverUnload callback.
- Create a control device object

Control Device Objects (1/2)

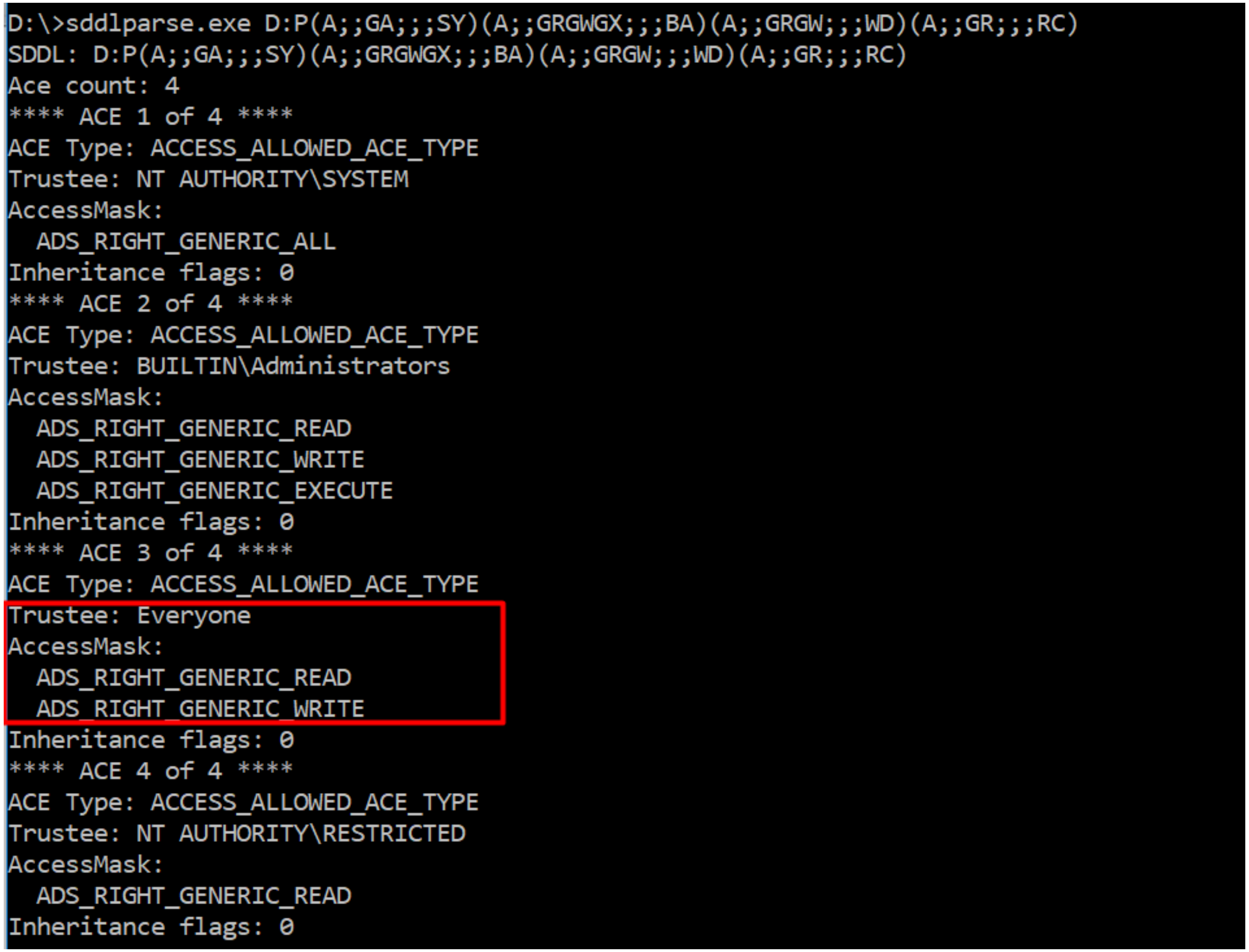
- These are used by KMDF drivers to support an extra set of IO control codes for applications
- Typical Flow:
  1. *WdfControlDeviceInitAllocate()*
  2. *WdfDeviceInitAssignName()*
  3. *WdfDeviceCreate()*
  4. *WdfDeviceCreateSymbolicLink()*
  5. *WdfIoQueueCreate()*
  6. *WdfControlFinishInitializing()*

Control Device Objects (2/2)

```
PWDFDEVICE_INIT WdfControlDeviceInitAllocate(
  WDFDRIVER          Driver,
  CONST UNICODE_STRING *SDDLString
);
```

- The SDDLString specifies the Security Descriptor to apply to the object.
- It can later be overridden with *WdfDeviceInitAssignSDDLString()*
- SDDL Parse Tool: http://blogs.microsoft.co.il/files/folders/guyt/entry70399.aspx

Decoding the SDDL



```
D:\>sddlparse.exe D:P(A;;GA;;;SY)(A;;GRGWGX;;;BA)(A;;GRGW;;;WD)(A;;GR;;;RC)
SDDL: D:P(A;;GA;;;SY)(A;;GRGWGX;;;BA)(A;;GRGW;;;WD)(A;;GR;;;RC)
Ace count: 4
**** ACE 1 of 4 ****
ACE Type: ACCESS_ALLOWED_ACE_TYPE
Trustee: NT AUTHORITY\SYSTEM
AccessMask:
  ADS_RIGHT_GENERIC_ALL
Inheritance flags: 0
**** ACE 2 of 4 ****
ACE Type: ACCESS_ALLOWED_ACE_TYPE
Trustee: BUILTIN\Administrators
AccessMask:
  ADS_RIGHT_GENERIC_READ
  ADS_RIGHT_GENERIC_WRITE
  ADS_RIGHT_GENERIC_EXECUTE
Inheritance flags: 0
**** ACE 3 of 4 ****
ACE Type: ACCESS_ALLOWED_ACE_TYPE
Trustee: Everyone
AccessMask:
  ADS_RIGHT_GENERIC_READ
  ADS_RIGHT_GENERIC_WRITE
Inheritance flags: 0
**** ACE 4 of 4 ****
ACE Type: ACCESS_ALLOWED_ACE_TYPE
Trustee: NT AUTHORITY\RESTRICTED
AccessMask:
  ADS_RIGHT_GENERIC_READ
Inheritance flags: 0
```

Demo kmdf-re.py

IRP/WDFRequest Pre-Processing

- There are two methods to do this:

1. **WdfDeviceInitSetIoInCallerContextCallback**
    - To get the WDFRequest before it gets into the IOQueue

2. **WdfDeviceInitAssignWdmIrpPreprocessCallback**
    - To get the IRP before the Framework

- If you see any of these, you need to check whether they are hooking an interesting major function.

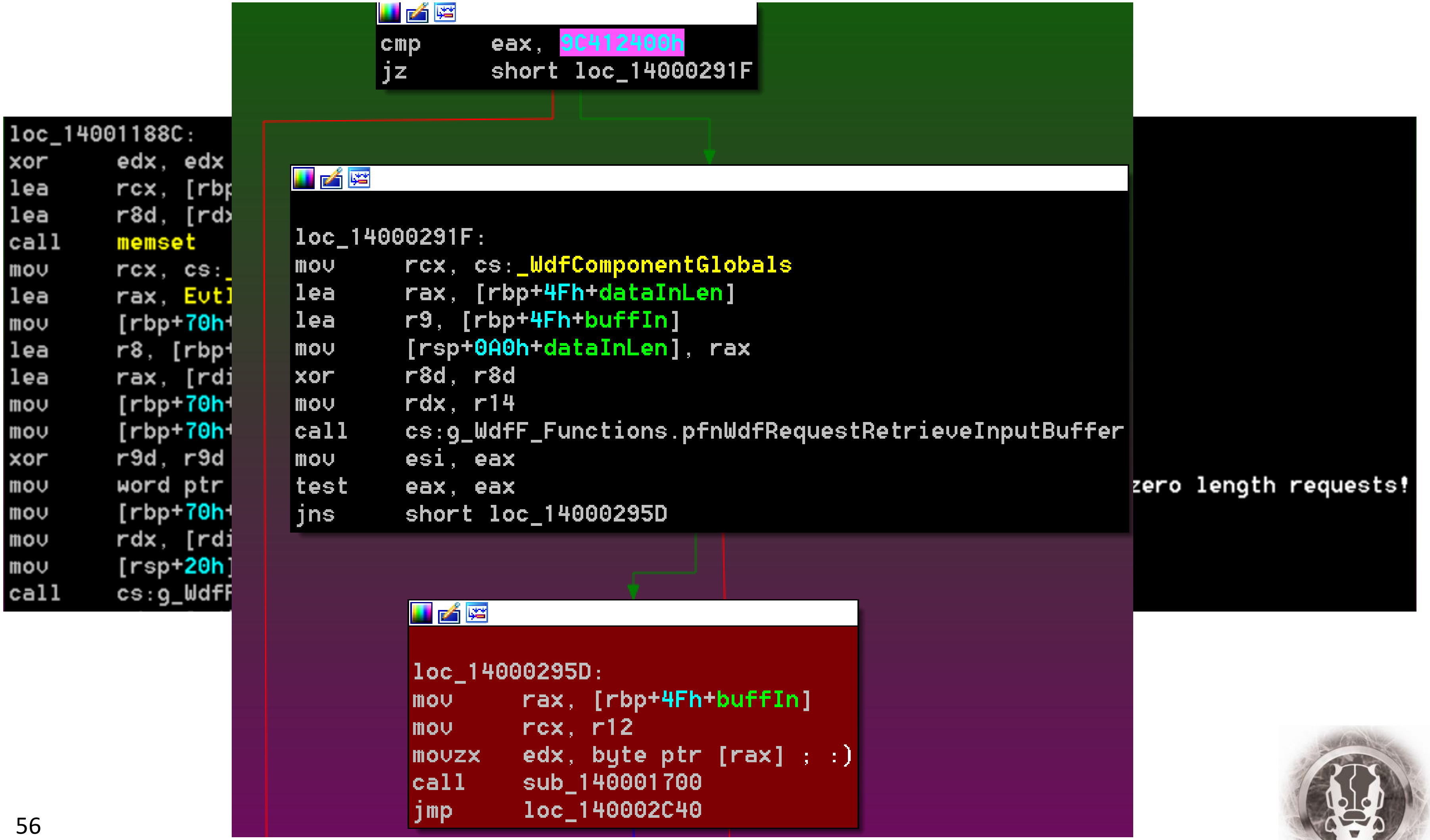
Null Buffers

- When calling DeviceIoControl with:
  - 0 for BufferLengths
  - NULL for Buffers
- This basic test used to trigger a lot of null-dereference conditions in WDM.
  - IRP->SystemBuffer = NULL
  - IRP->MdlAddress = NULL
- Still does in 2018 but not with KMDF:
  - CVE-2018-8342 - Windows NDIS Elevation of Privilege Vulnerability

Null Buffers Conditions (1/3)

```
NTSTATUS WdfRequestRetrieveInputBuffer(
  WDFREQUEST Request,
  size_t     MinimumRequiredLength, → This must be Zero
  PVOID       *Buffer,
  size_t     *Length
);
```

No issues here ☹



```
loc_14001188C:
xor     edx, edx
lea     rcx, [rbp
lea     r8d, [rdx
call    memset
mov     rcx, cs:_
lea     rax, EvtI
mov     [rbp+70h+
lea     r8, [rbp+
lea     rax, [rdi
mov     [rbp+70h+
mov     [rbp+70h+
xor     r9d, r9d
mov     word ptr
mov     [rbp+70h+
mov     rdx, [rdi
mov     [rsp+20h]
call    cs:g_WdfF
```

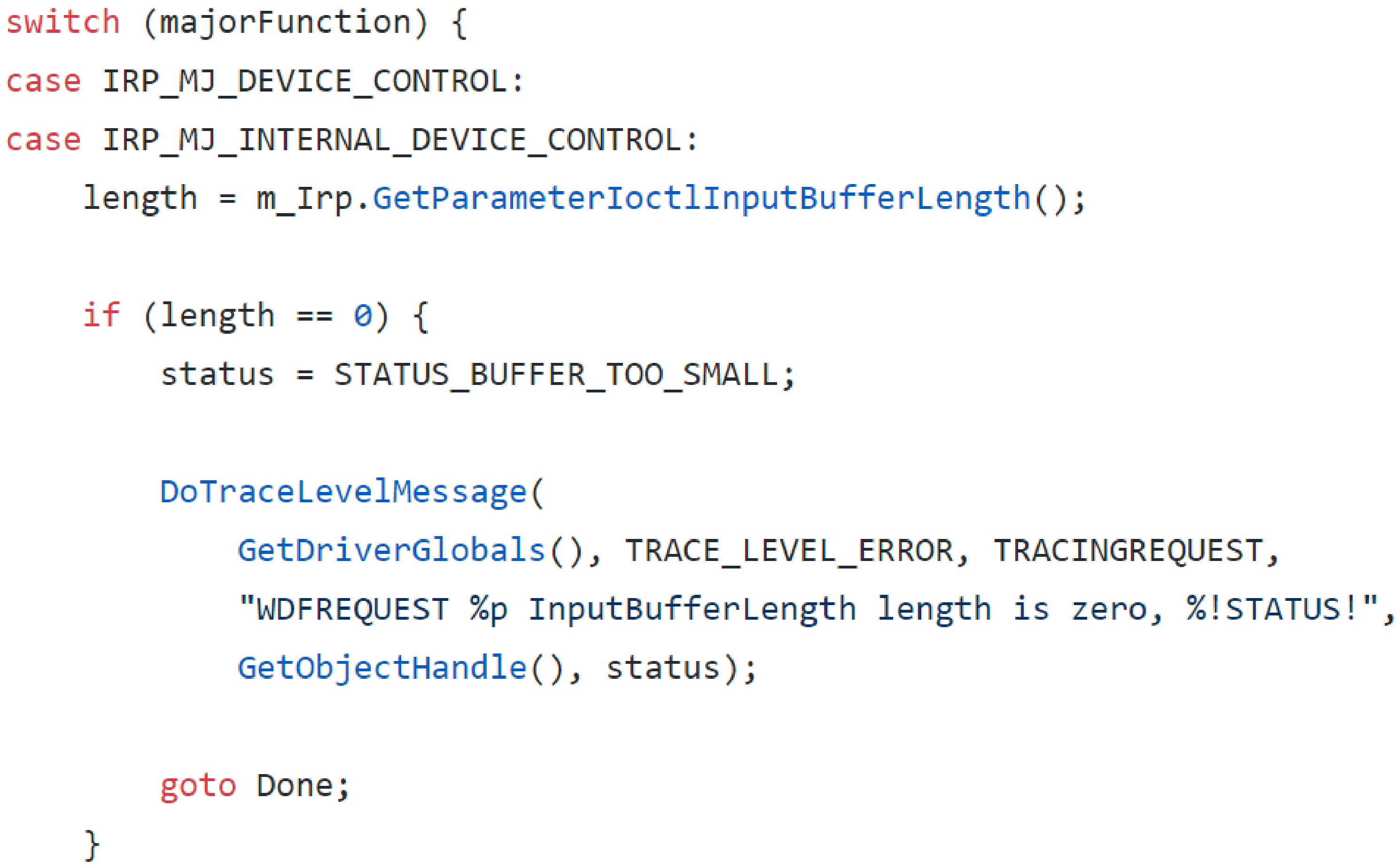
```
cmp     eax, 9C412400h
jz      short loc_14000291F
```

```
loc_14000291F:
mov     rcx, cs:_WdfComponentGlobals
lea     rax, [rbp+4Fh+dataInLen]
lea     r9, [rbp+4Fh+buffIn]
mov     [rsp+0A0h+dataInLen], rax
xor     r8d, r8d
mov     rdx, r14
call    cs:g_WdfF_Functions.pfnWdfRequestRetrieveInputBuffer
mov     esi, eax
test    eax, eax
jns     short loc_14000295D
```

```
zero length requests!
```

```
loc_14000295D:
mov     rax, [rbp+4Fh+buffIn]
mov     rcx, r12
movzx   edx, byte ptr [rax] ; :)
call    sub_140001700
jmp     loc_140002C40
```

Null Buffers Conditions (2/3)



```
switch (majorFunction) {
case IRP_MJ_DEVICE_CONTROL:
case IRP_MJ_INTERNAL_DEVICE_CONTROL:
    length = m_Irp.GetParameterIoctlInputBufferLength();

    if (length == 0) {
        status = STATUS_BUFFER_TOO_SMALL;

        DoTraceLevelMessage(
            GetDriverGlobals(), TRACE_LEVEL_ERROR, TRACINGREQUEST,
            "WDFREQUEST %p InputBufferLength length is zero, %!STATUS!",
            GetObjectHandle(), status);

        goto Done;
    }
```

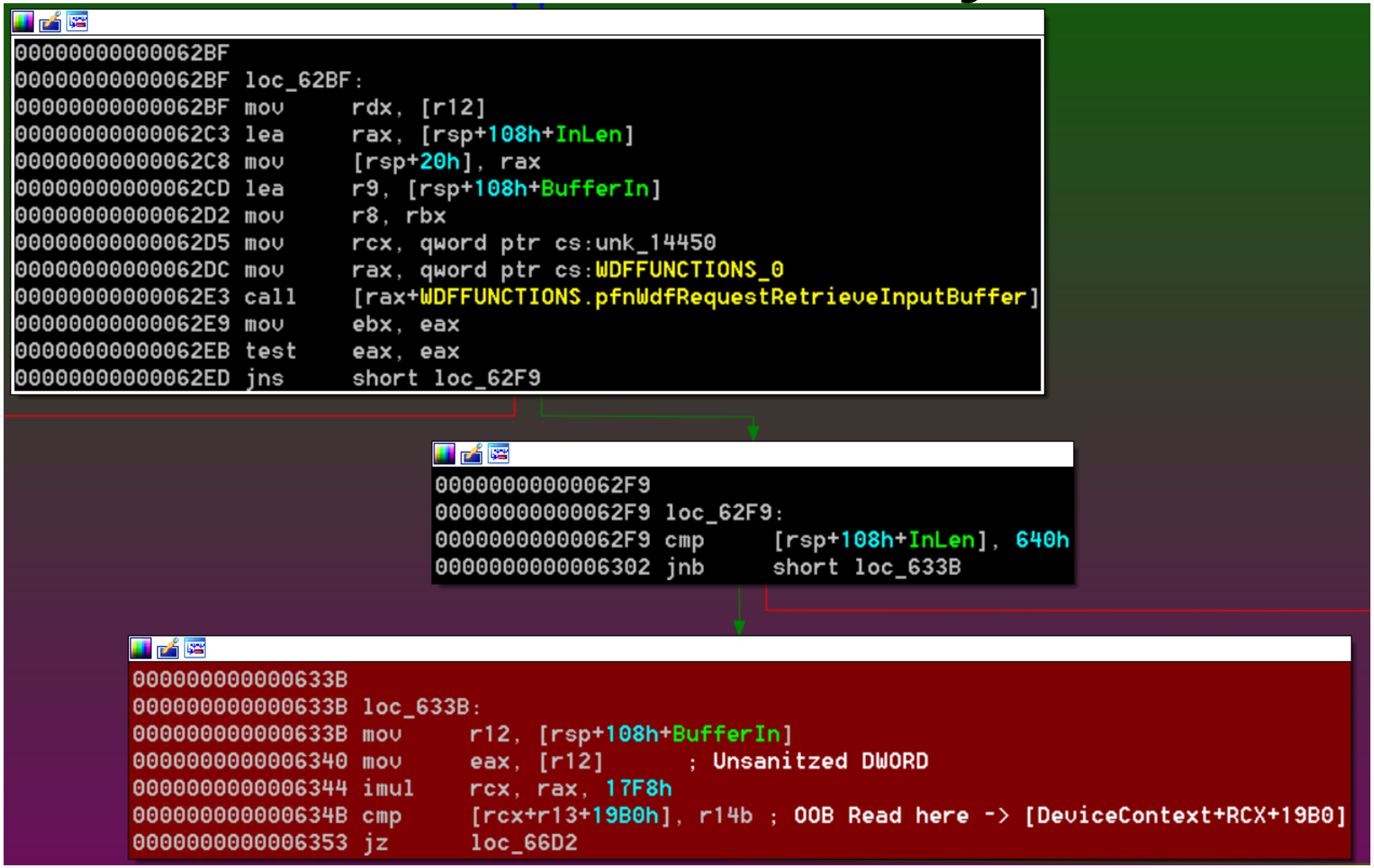
Null Buffers Conditions (3/3) – For Read/Write requests

```
typedef struct _WDF_IO_QUEUE_CONFIG {
  ULONG                                 Size;
  WDF_IO_QUEUE_DISPATCH_TYPE            DispatchType;
  WDF_TRI_STATE                         PowerManaged;
  BOOLEAN                               AllowZeroLengthRequests; → This must be True for Read/Write; Default is FALSE
  BOOLEAN                               DefaultQueue;
  PFN_WDF_IO_QUEUE_IO_DEFAULT           EvtIoDefault;
  PFN_WDF_IO_QUEUE_IO_READ              EvtIoRead;
  PFN_WDF_IO_QUEUE_IO_WRITE             EvtIoWrite;
  PFN_WDF_IO_QUEUE_IO_DEVICE_CONTROL    EvtIoDeviceControl;
  PFN_WDF_IO_QUEUE_IO_INTERNAL_DEVICE_CONTROL EvtIoInternalDeviceControl;
  PFN_WDF_IO_QUEUE_IO_STOP              EvtIoStop;
  PFN_WDF_IO_QUEUE_IO_RESUME            EvtIoResume;
  PFN_WDF_IO_QUEUE_IO_CANCELED_ON_QUEUE EvtIoCanceledOnQueue;
  union {
    struct {
      ULONG NumberOfPresentedRequests;
    } Parallel;
  } Settings;
  WDFDRIVER                             Driver;
} WDF_IO_QUEUE_CONFIG, *PWDF_IO_QUEUE_CONFIG;
```

Type of Issues

- *Unsanitized data*
    - *Indexes*
    - *Offsets*
    - *Pointers*
- *EvtIoDefault Type Confusion*
- *Privileged Operations Exposed*
    - *MSR control, IO Ports, Registry Keys, Physical Memory read/write, etc.*
- *Memory Exhaustion (Object leakage)*
- *Race conditions when using DirectIO*
- *Kernel pointers leakage in OutputBuffers*

Un-sanitized index: CSI2HostControllerDriver.sys



```
00000000000062BF
00000000000062BF loc_62BF:
00000000000062BF mov     rdx, [r12]
00000000000062C3 lea     rax, [rsp+108h+InLen]
00000000000062C8 mov     [rsp+20h], rax
00000000000062CD lea     r9, [rsp+108h+BufferIn]
00000000000062D2 mov     r8, rbx
00000000000062D5 mov     rcx, qword ptr cs:unk_14450
00000000000062DC mov     rax, qword ptr cs:WDFFUNCTIONS_0
00000000000062E3 call    [rax+WDFFUNCTIONS.pfnWdfRequestRetrieveInputBuffer]
00000000000062E9 mov     ebx, eax
00000000000062EB test    eax, eax
00000000000062ED jns     short loc_62F9
```

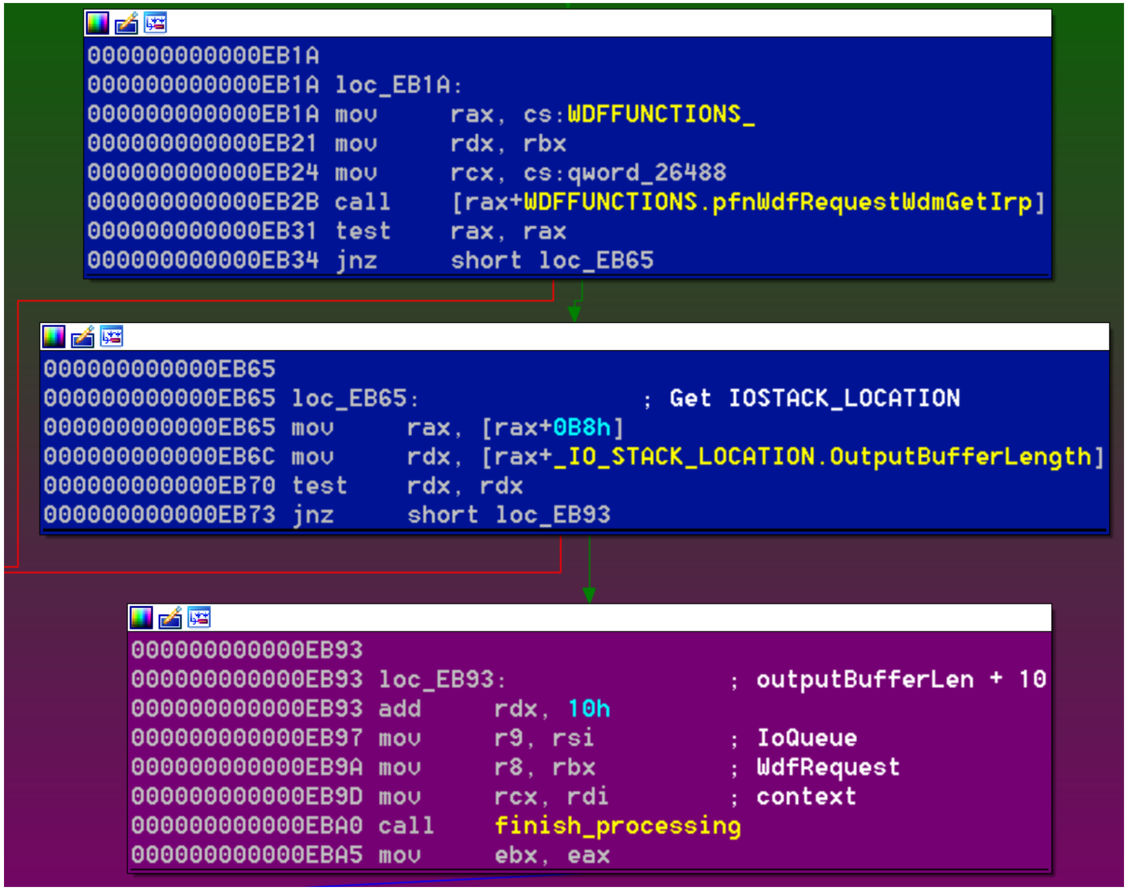
```
00000000000062F9
00000000000062F9 loc_62F9:
00000000000062F9 cmp     [rsp+108h+InLen], 640h
0000000000006302 jnb     short loc_633B
```

```
000000000000633B
000000000000633B loc_633B:
000000000000633B mov     r12, [rsp+108h+BufferIn]
0000000000006340 mov     eax, [r12]       ; Unsanitzed DWORD
0000000000006344 imul    rcx, rax, 17F8h
000000000000634B cmp     [rcx+r13+19B0h], r14b ; OOB Read here -> [DeviceContext+RCX+19B0]
0000000000006353 jz      loc_66D2
```

EvtIoDefault Type Confusion

- The framework calls an IO queue EvtIoDefault callback when a request is available and there is not a type specific callback function.
- If EvtIoDefault is used, the code should check the Request/IRP type before processing its content.

EvtIoDefault Type Confusion



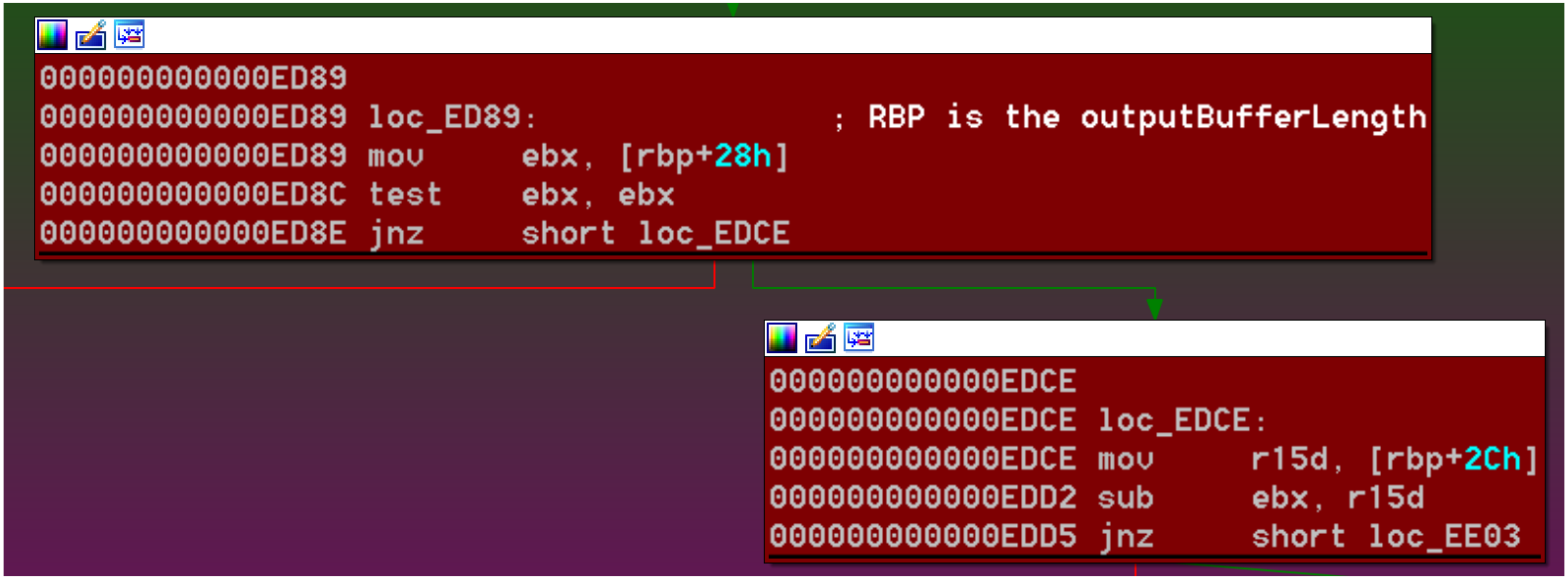
```
000000000000EB1A
000000000000EB1A loc_EB1A:
000000000000EB1A mov     rax, cs:WDFFUNCTIONS_
000000000000EB21 mov     rdx, rbx
000000000000EB24 mov     rcx, cs:qword_26488
000000000000EB2B call    [rax+WDFFUNCTIONS.pfnWdfRequestWdmGetIrp]
000000000000EB31 test    rax, rax
000000000000EB34 jnz     short loc_EB65
```

```
000000000000EB65
000000000000EB65 loc_EB65:                       ; Get IOSTACK_LOCATION
000000000000EB65 mov     rax, [rax+0B8h]
000000000000EB6C mov     rdx, [rax+_IO_STACK_LOCATION.OutputBufferLength]
000000000000EB70 test    rdx, rdx
000000000000EB73 jnz     short loc_EB93
```

```
000000000000EB93
000000000000EB93 loc_EB93:                       ; outputBufferLen + 10
000000000000EB93 add     rdx, 10h
000000000000EB97 mov     r9, rsi                 ; IoQueue
000000000000EB9A mov     r8, rbx                 ; WdfRequest
000000000000EB9D mov     rcx, rdi                ; context
000000000000EBA0 call    finish_processing
000000000000EBA5 mov     ebx, eax
```

Example: an EvtIoDefault callback that took the IRP from the WDFRequest, then grabbed the OutputBufferLength from the IO_STACK_LOCATION and added 0x10 to it to then pass it to another function.

EvtIoDefault Type Confusion



```
000000000000ED89
000000000000ED89 loc_ED89:                      ; RBP is the outputBufferLength
000000000000ED89 mov     ebx, [rbp+28h]
000000000000ED8C test    ebx, ebx
000000000000ED8E jnz     short loc_EDCE
```

```
000000000000EDCE
000000000000EDCE loc_EDCE:
000000000000EDCE mov     r15d, [rbp+2Ch]
000000000000EDD2 sub     ebx, r15d
000000000000EDD5 jnz     short loc_EE03
```

Inside that function, the code used the Length as a Pointer!

Example: privileged operation exposed + memory exhaustion (leak)

- Bus drivers report enumerated devices to the PnP Manager, which uses the information to build the device tree.

- The framework enables drivers to support dynamic enumeration by providing child-list objects.

- Each child-list object represents a list of child devices that are connected to a parent device.

Example: privileged operation exposed + memory exhaustion (leak)

- Each time a bus driver identifies a child device, it must add the child device's description to a child list and create a physical device object (PDO) for it.
  - *It does this by calling:*

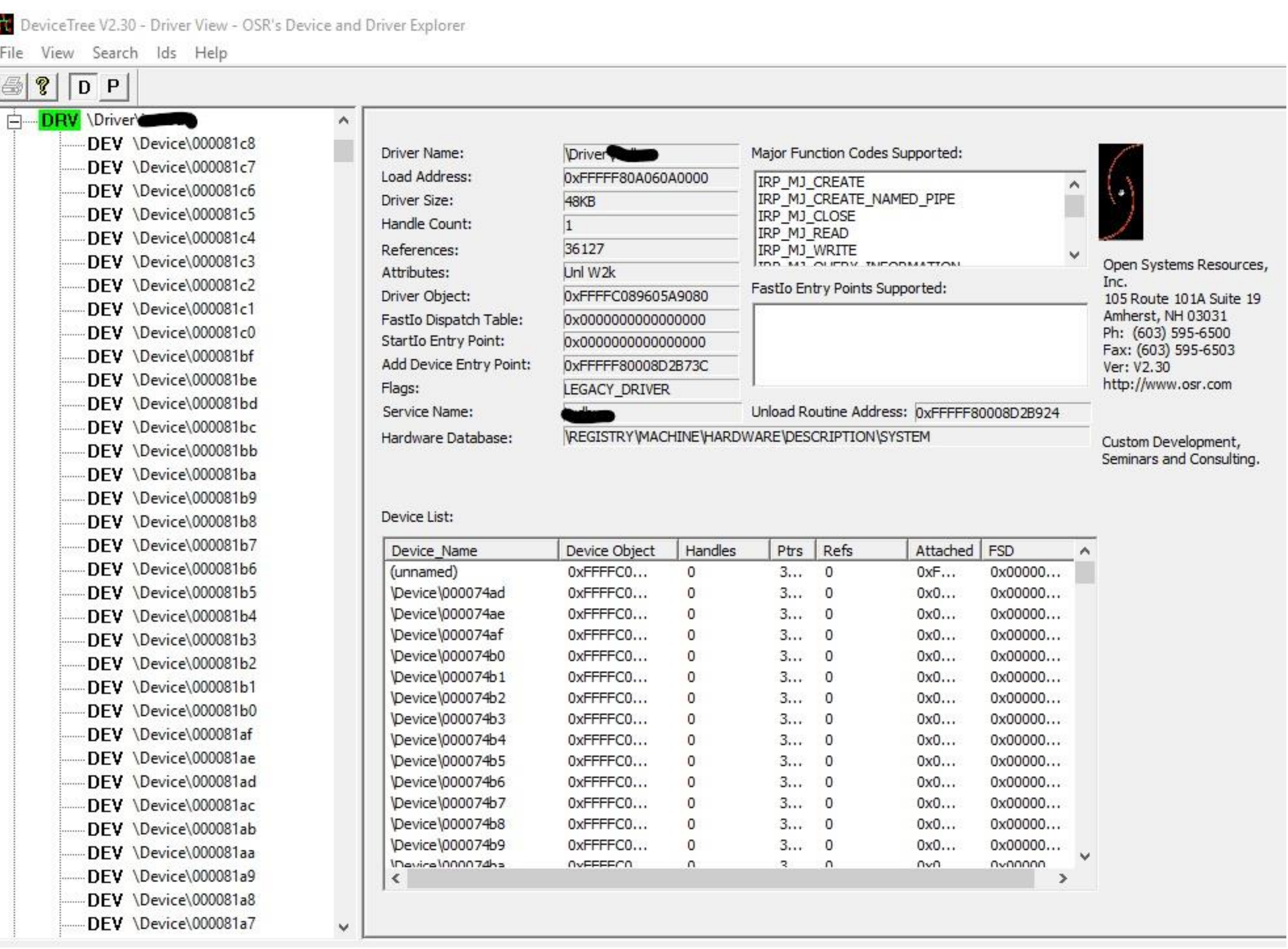
```
WdfChildListAddOrUpdateChildDescriptionAsPresent
(
  ChildList,
  IdentificationDescription,
  AddressDescription
);
```

Privileged Operations Exposed

- This API should be called in two situations:
  1. When a parent device receives an interrupt that indicates the arrival or removal of a child.
  2. When the parent device enters its working (D0) state, in the context of EvtChildListScanForChildren.

- So what happens when you expose *WdfChildListAddOrUpdateChildDescriptionAsPresent*() as an IOCTL operation?

  → The objects will leak until the system collapses



DRV \Driver\[redacted]

DEV \Device\000081c8
DEV \Device\000081c7
DEV \Device\000081c6
DEV \Device\000081c5
DEV \Device\000081c4
DEV \Device\000081c3
DEV \Device\000081c2
DEV \Device\000081c1
DEV \Device\000081c0
DEV \Device\000081bf
DEV \Device\000081be
DEV \Device\000081bd
DEV \Device\000081bc
DEV \Device\000081bb
DEV \Device\000081ba
DEV \Device\000081b9
DEV \Device\000081b8
DEV \Device\000081b7
DEV \Device\000081b6
DEV \Device\000081b5
DEV \Device\000081b4
DEV \Device\000081b3
DEV \Device\000081b2
DEV \Device\000081b1
DEV \Device\000081b0
DEV \Device\000081af
DEV \Device\000081ae
DEV \Device\000081ad
DEV \Device\000081ac
DEV \Device\000081ab
DEV \Device\000081aa
DEV \Device\000081a9
DEV \Device\000081a8
DEV \Device\000081a7

Driver Name: \Driver\[redacted]
Load Address: 0xFFFFF80A060A0000
Driver Size: 48KB
Handle Count: 1
References: 36127
Attributes: Unl W2k
Driver Object: 0xFFFFC089605A9080
FastIo Dispatch Table: 0x0000000000000000
StartIo Entry Point: 0x0000000000000000
Add Device Entry Point: 0xFFFFF80008D2B73C
Flags: LEGACY_DRIVER
Service Name: [redacted]
Hardware Database: \REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM

Major Function Codes Supported:
IRP_MJ_CREATE
IRP_MJ_CREATE_NAMED_PIPE
IRP_MJ_CLOSE
IRP_MJ_READ
IRP_MJ_WRITE

FastIo Entry Points Supported:

Unload Routine Address: 0xFFFFF80008D2B924

Open Systems Resources, Inc.
105 Route 101A Suite 19
Amherst, NH 03031
Ph: (603) 595-6500
Fax: (603) 595-6503
Ver: V2.30
http://www.osr.com

Custom Development, Seminars and Consulting.

Device List:

Device_Name	Device Object	Handles	Ptrs	Refs	Attached	FSD
(unnamed)	0xFFFFC0...	0	3...	0	0xF...	0x00000...
\Device\000074ad	0xFFFFC0...	0	3...	0	0x0...	0x00000...
\Device\000074ae	0xFFFFC0...	0	3...	0	0x0...	0x00000...
\Device\000074af	0xFFFFC0...	0	3...	0	0x0...	0x00000...
\Device\000074b0	0xFFFFC0...	0	3...	0	0x0...	0x00000...
\Device\000074b1	0xFFFFC0...	0	3...	0	0x0...	0x00000...
\Device\000074b2	0xFFFFC0...	0	3...	0	0x0...	0x00000...
\Device\000074b3	0xFFFFC0...	0	3...	0	0x0...	0x00000...
\Device\000074b4	0xFFFFC0...	0	3...	0	0x0...	0x00000...
\Device\000074b5	0xFFFFC0...	0	3...	0	0x0...	0x00000...
\Device\000074b6	0xFFFFC0...	0	3...	0	0x0...	0x00000...
\Device\000074b7	0xFFFFC0...	0	3...	0	0x0...	0x00000...
\Device\000074b8	0xFFFFC0...	0	3...	0	0x0...	0x00000...
\Device\000074b9	0xFFFFC0...	0	3...	0	0x0...	0x00000...

Example: privileged operation exposed + memory exhaustion (leak)

- Microsoft Driver Sample: toastDrv
  - ***https://github.com/Microsoft/Windows-driver-samples/blob/master/general/toaster/toastDrv/kmdf/bus/dynamic/busenum.c***
- Some concrete implementations with the same pattern:
  - *vwifibus.sys (Microsoft Virtual WiFi Bus Drv)*
  - *iwdbus.sys (Intel Wireless Display Driver)*
  - *ssdevfactory.sys (SteelSeries Engine)*

Bus driver attack surface++

- Not only that, but we also have more attack surface when this happens.
- **IdentificationDescription** and **AddressDescription** arguments are driver defined structures that are used by the internal functions registered as part of the WdfFdoInitSetDefaultChildListConfig call:
  - *EvtChildListIdentificationDescriptionCopy*
  - *EvtChildListIdentificationDescriptionDuplicate*
  - *EvtChildListIdentificationDescriptionCleanup*
  - *EvtChildListIdentificationDescriptionCompare*
  - *EvtChildListAddressDescriptionCopy*
  - *EvtChildListAddressDescriptionDuplicate*
  - *EvtChildListAddressDescriptionCleanup*

Kernel Pointers Leakage

- Synaptics Touchpad Win64 Driver
  - SynTP.sys → used by some HP, Lenovo, Acer, …?
  - The following IOCTLs returned kernel pointers:
    - 80002040h
    - 80002030h
    - 80002034h
    - 80002038h
    - 8000200ch
    - 8000203ch
    - 80002010h
    - 80002000h
    - 80002050h
    - 80002044h
    - 8000a008h
    - 80006004h
    - 80006018h
  - Synaptics informed us that not all OEM's are using the official update for different reasons.

KMDF and Miniports

- Some miniport drivers can use Kernel-Mode Driver Framework, if the port/miniport architecture allows the miniport driver to communicate with other drivers by using WDM or framework interfaces.
    - Example: NDIS Miniport
- In these cases, the driver doesn’t use the KMDF callbacks.

Finding KMDF drivers

```
for driver_name in driver_names:
    try:
        pe = pefile.PE(DRIVER_PATH + driver_name)
    except pefile.PEFormatError as message:
        print message, driver_name
    pe.parse_data_directories()
    kmdf = False
    try:
        for entry in pe.DIRECTORY_ENTRY_IMPORT:
            if entry.dll == "WDFLDR.SYS":
                kmdf = True
                sys.stdout.write("+")
                break
        if kmdf:
            final_list.append(driver_name)
except AttributeError:
        pass
```

Check your drivers!

- Third party bus drivers
- TouchPads
- Cameras
- Gamer devices
  - Mouse
  - Keyboards
  - Headsets
  - Joysticks and gamepads

Conclusions (1/2)

- KMDF does enhance security by default.
  - FILE_DEVICE_SECURE_OPEN
  - No NULL buffers
  - Probed and Locked buffers (discourages the use of METHOD_NEITHER)
  - Better APIs to access request information and check sizes

Conclusions (2/2)

- However, there are many things that can go wrong:
  - Bad ACLs for device objects is still a problem.
    - FILE_ANY_ACCESS abused in most cases.
  - Buffer’s data should be treated carefully.
  - APIs may be used in the wrong way.
  - Race conditions still apply when DirectIO is used.
  - Privileged operations shouldn’t be exposed to regular users.
  - Double check the content being written into output buffers.

References

- Windows Internals 6th Edition
- Windows 7 Device Driver
- The Windows NT Device Driver Book
- Accessing User Buffers
- Architecture of the Kernel-Mode Driver Framework
- Summary of KMDF and WDM Equivalents

get the kmdf_rf scripts here.

Thank you