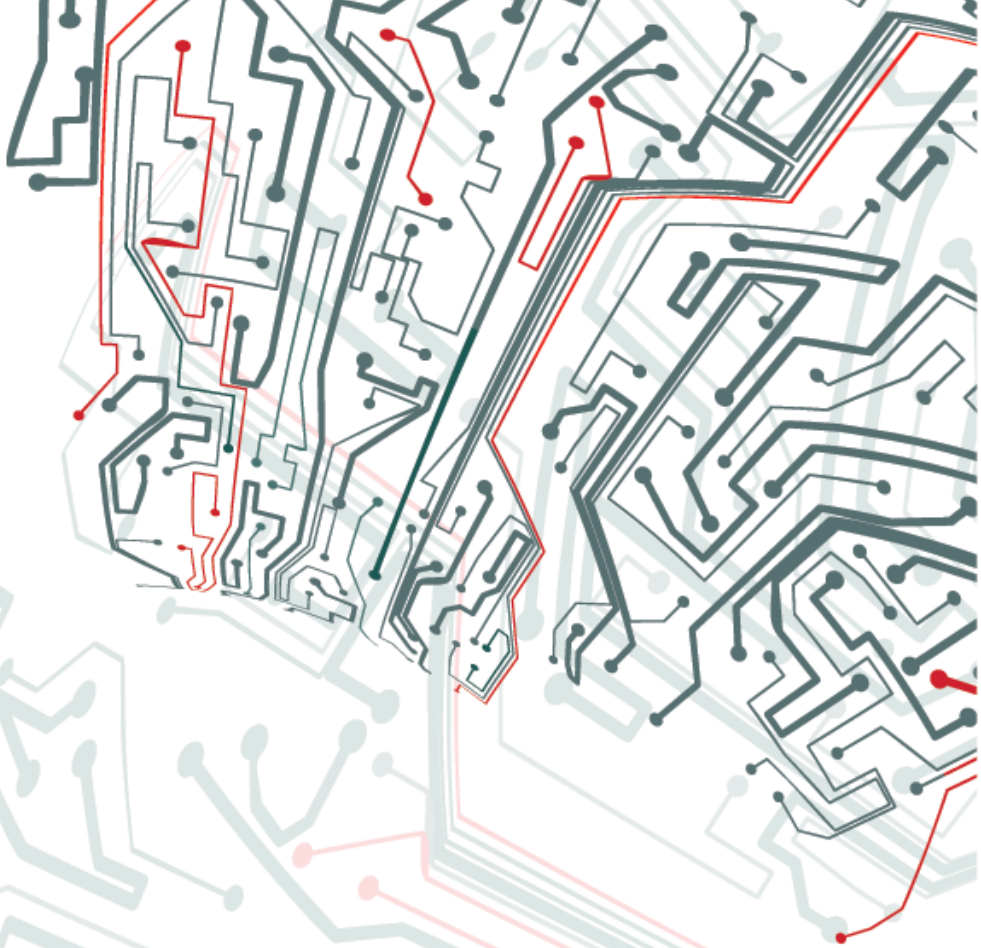
Windows Kernel Graphics Driver Attack Surface

Ilja van Sprundel
Director of Penetration Testing



IOActive
Hardware | Software | Wetware
SECURITY SERVICES

Who Am I ?

- Ilja van Sprundel
- ivansprundel@ioactive.com
- Director of Penetration Testing at IOActive
- Pen test
- Code review
- Break stuff for fun and profit ☺

IOActive™

Outline/Agenda

- What's this talk about ?
- Windows graphics drivers
- WDDM kmd Driver
  - Synchronization
  - Entrypoints
- Full userland program to talk about this stuff
- Sniffing/snooping private data
- Putting it all together
  - Fuzzing
  - Reverse engineering

IOActive™

What's This Talk About ?

- Windows® WDDM drivers
  - Implementation security
  - Kernel driver part
- Audience
  - Auditors (what to look for)
  - Graphics drivers developers
    (what not to do, and where to pay close attention)
  - Curious people that like to poke around in driver internals
- Knowledge
  - Some basic knowledge of Windows drivers (IRP's, probing, capturing, …)

IOActive™

Windows Graphics Drivers

- Old Model
  - XDDM /XPDM
  - Windows 2000/XP
  - No longer supported as of Windows 8
  - Not covered in this presentation
- WDDM (Windows Display Driver Model)
  - New Vista model
    - v1 – vista
    - v1.1 – win 7
    - v1.2 – win 8
    - V1.3 – win 8.1
  - Will only describe interesting parts from a security perspective

IOActive™

Windows Graphics Drivers

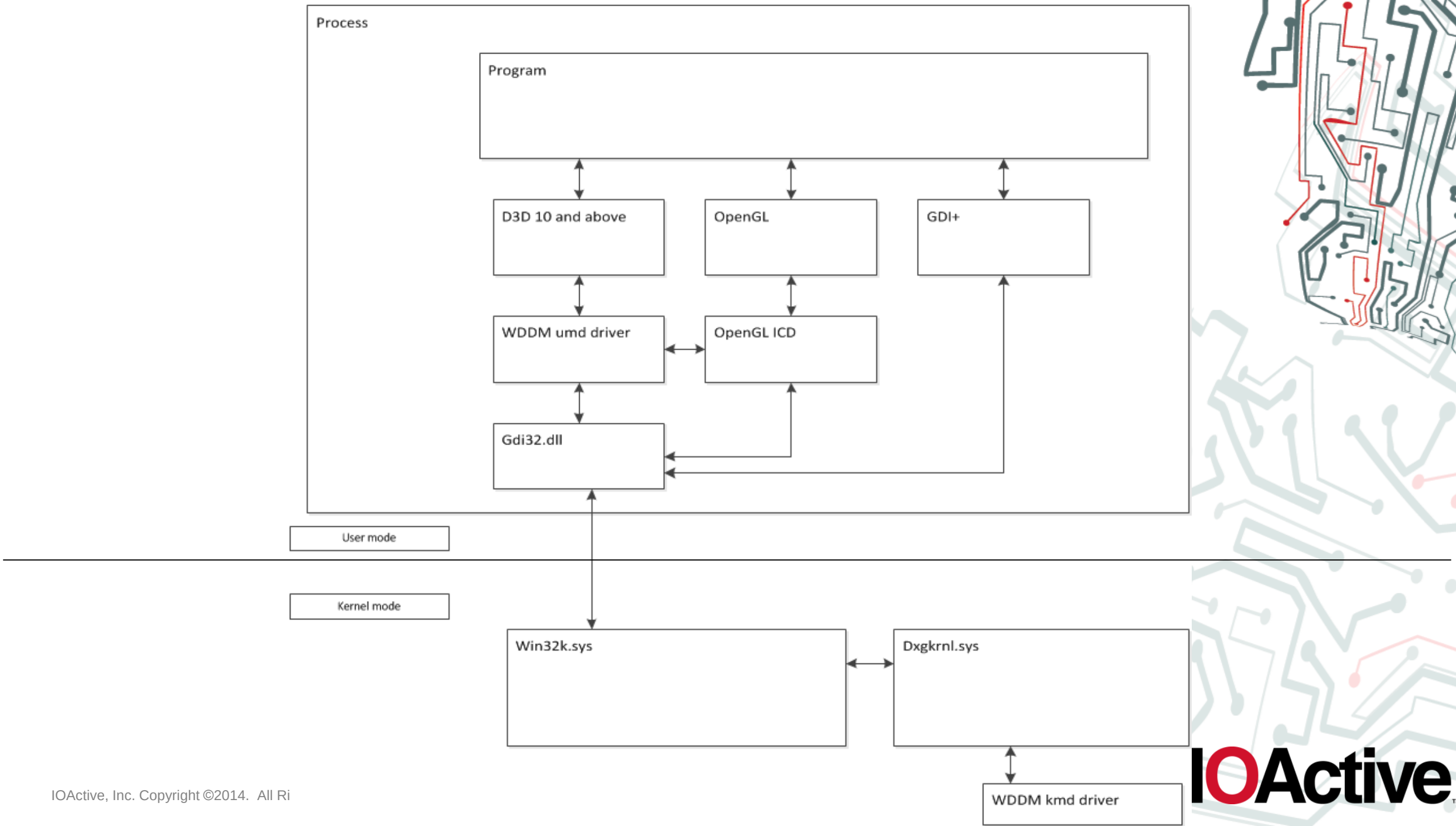
- So who makes these things and why ?
  - IHV's (Intel, NVIDIA, AMD, Qualcomm, PowerVR, VIA, Matrox, …)
    - Very rich drivers
  - Basic fallback (basic render, basic display)
    - Implements the bare minimum
  - Virtualization (VMware, Virtual Box, Parallels guest drivers)
    - Specific special purpose driver
  - Remote desktop scenario's (XenDesktop, RDP, …)
    - Specific special purpose driver
  - Virtual display (intelligraphics, extramon, …)
    - Specific special purpose driver

IOActive™

Windows Graphics Drivers

- WDDM model is split between user mode and kernel mode
- Move to user was done for stability and reliability
  - A large chunk of all blue screen prior to vista were due to graphics drivers (from MSDN): "*In Windows XP, display drivers, which are large and complex, can be a major source of system instability. These drivers execute entirely in kernel mode (i.e., deep in the system code) and hence a single problem in the driver will often force the entire system to reboot. According to the **crash analysis** data collected during the **Windows XP timeframe**, display drivers are responsible for **up to 20 percent of all blue screens***."
- User mode part runs as part of a dll in most processes
  - Still has interesting attack surface
    - Encoders / decoders
    - Binary planting
    - Some API's might be partially (and indirectly) exposed to remote attack surface (e.g. WebGL)
- Will not cover user mode part, only kernel mode.

IOActive™



WDDM kmd Driver

- So what does a WDDM kmd driver look like?

```
NTSTATUS DriverEntry( IN PDRIVER_OBJECT DriverObject,
                      IN PUNICODE_STRING RegistryPath )
{
...
            DRIVER_INITIALIZATION_DATA DriverInitializationData;
...
            DriverInitializationData.DxgkDdiEscape = DDIEscape;
...
   Status = DxgkInitialize(DriverObject,
                           RegistryPath,
                           &DriverInitializationData);
...
}
```


IOActive™

WDDM kmd Driver

- DriverEntry() is the main entry point for any kernel driver
- Fill in DRIVER_INITIALIZATION_DATA struct
  - Contains a large set of callback functions
  - ‘dynamic’ struct
    - Bigger on win7 (vs vista)
    - Even bigger on win8
    - Grown even more for win 8.1
    - All later elements appended at the end
- Call DxgkInitialize()
  - Tells dxgkernel about this driver and all its callbacks
- No IRP’s, no IOCTL’s, nothing like WDM. You don’t pass the IoManager.

IOActive™

WDDM kmd Driver

- Very similar variant of this
- Calls DxgkInitializeDisplayOnlyDriver() iso DxgkInitialize() instead
- Uses PKMDDOD_INITIALIZATION_DATA structure
- Much like the previous, but for use by a kernel mode display only driver

WDDM kmd Driver

- DRIVER_INITIALIZATION_DATA contains all sorts of callbacks
- From an attack surface perspective, we can roughly divide them into three groups:
  - Those where an attacker has no or very little control
  - Those where an attacker has some (indirect) control
  - Those where an attacker has significant input into the callback
- We’re obviously mainly concerned with the latter

IOActive™

WDDM kmd Driver – Synchronization

- WDDM has a threading model for these callbacks which basically consists of four levels (where each callback belongs to one of these):
- Three
  - Only a single thread may enter
  - GPU Has to be idle
  - No DMA buffers being processed
  - Video memory is evicted to host CPU memory
- Two
  - Same as three except for video memory eviction

IOActive™

WDDM kmd Driver – Synchronization

- One
  - Calls are categorized into classes. Only one thread of each class is allowed to call into callback simultaneously
- Zero
  - Completely reentrant
- If concurrency is allowed, no two concurrent threads may belong to the same process.
- This is important to know, since you need to keep this in mind when looking for potential race conditions scenarios.

IOActive™

WDDM kmd Driver Entrypoints

- A fairly small number of the callbacks take significant input from userland:
  - Escape
  - Render
  - Allocation
  - QueryAdapter
- Before we can get to them, we need to perform proper driver initialization
  - Look at this first
- Then look at the callbacks

WDDM kmd Driver Entrypoints – Initialization

- Need to initialize the device before entry points can be reached from userland
- Assume we come from the GDI world and we have an HDC
- Succinctly, this involves three steps:
  - Convert HDC to WDDM adapter handle
  - Get a WDDM device handle out of the adapter handle
  - Create a context for the device

WDDM kmd Driver Entrypoints – Initialization

- Convert HDC to adapter handle
- Fill in the D3DKMT_OPENADAPTERFROMHDC data structure
- Call D3DKMTOpenAdapterFromHdc

```
D3DKMT_OPENADAPTERFROMHDC
oafh;
memset(&oafh, 0x00, sizeof(oafh));
oafh.hDc = GetDC(NULL);
D3DKMTOpenAdapterFromHdc(&oafh);
```


IOActive™

WDDM kmd Driver Entrypoints – Initialization

- Get a device handle out of the adapter handle
- Fill in D3DKMT_CREATEDEVICE data structure
- Call D3DKMTCreateDevice

```
D3DKMT_CREATEDEVICE cdev;
memset(&cdev, 0x00, sizeof(cctx));
cdev.hAdapter = oafh.hAdapter;
D3DKMTCreateDevice(&cdev);
```


IOActive™

WDDM kmd Driver Entrypoints – Initialization

- Create a context for the device
- The previously obtained device handle is the handle that gets passed to most userland API's to talk to WDDM drivers.
- In order to do anything, you'll need to create a device context for the device
  - Sets up stuff like command buffers that can be passed off to a WDDM driver
  - There is some attack surface here. Allows passing arbitrary userland data (**pPrivateDriverData)** (with associated length, **PrivateDriverDataSize**) to WDDM driver.
    - It may or may not look at it. This is completely driver dependent.

IOActive™

WDDM kmd Driver Entrypoints – Initialization

- Create a context for the device
- Fill in D3DKMT_CREATECONTEXT data structure
- Call D3DKMTCreateContext

```
D3DKMT_CREATECONTEXT cctx;
memset(&cctx, 0x00, sizeof(cctx));
cctx.hDevice = cdev.hDevice;
r = pfnKTCreateContext(&cctx);
```

- DxgkDdiCreateContext kernel entry point

```
typedef struct _D3DKMT_CREATECONTEXT {
D3DKMT_HANDLE           hDevice;
UINT                    NodeOrdinal;
UINT                    EngineAffinity;
D3DDDI_CREATECONTEXTFLAGS Flags;
VOID                    *pPrivateDriverData;
UINT                    PrivateDriverDataSize;
D3DKMT_CLIENTHINT       ClientHint;
D3DKMT_HANDLE           hContext;
VOID                    *pCommandBuffer;
UINT                    CommandBufferSize;
D3DDDI_ALLOCATIONLIST   *pAllocationList;
UINT                    AllocationListSize;
D3DDDI_PATCHLOCATIONLIST *pPatchLocatio
nList; UINT                 PatchLocationListSize;
D3DGPU_VIRTUAL_ADDRESS  CommandBuff
er;
} D3DKMT_CREATECONTEXT;
```


IOActive

WDDM kmd Driver Entrypoints – Initialization

- Create a context for the device
- Some interesting output elements in struct
- Both command buffer and patchlocationlist get allocated on your behalf by WDDM
- In usermode. Used to talk to WDDM driver.

```
typedef struct _D3DKMT_CREATECONTEXT {
D3DKMT_HANDLE           hDevice;
UINT                    NodeOrdinal;
UINT                    EngineAffinity;
D3DDDI_CREATECONTEXTFLAGS Flags;
VOID                    *pPrivateDriverData;
UINT                    PrivateDriverDataSize;
D3DKMT_CLIENTHINT       ClientHint;
D3DKMT_HANDLE           hContext;
VOID                    *pCommandBuffer;
UINT                    CommandBufferSize;
D3DDDI_ALLOCATIONLIST   *pAllocationList;
UINT                    AllocationListSize;
D3DDDI_PATCHLOCATIONLIST *pPatchLocatio
nList; UINT                 PatchLocationListSize;
D3DGPU_VIRTUAL_ADDRESS  CommandBuff
er;
} D3DKMT_CREATECONTEXT;
```


IOActive

WDDM kmd Driver Entrypoints – Escape

- DxgkDdiEscape
- This is the IOCTL of graphics drivers.
- Very much like the ‘old’ extEscape
- However, no escape function is passed.
- Just a pointer to private data and a length value
- MSDN describes it as “The *DxgkDdiEscape* function shares information with the user-mode display driver.”
- Driver is free to implement this any way it sees fit.
- Data isn’t structured in any standardized way.
  - Can and will vary wildly from driver to driver.
- Threading level 2

IOActive™

WDDM kmd Driver Entrypoints – Escape

- What does DxgkDdiEscape look like?

```
NTSTATUS APIENTRY
DxgkDdiEscape(
  __in  const HANDLE hAdapter,
  __in  const DXGKARG_ESCAPE
*pEscape
)
{ ... }
```

```
typedef struct
_DXGKARG_ESCAPE {
  HANDLE              hDevice;
  D3DDDI_ESCAPEFLAGS Flags;
  VOID
*pPrivateDriverData;
  UINT
PrivateDriverDataSize;
  HANDLE              hContext;
} DXGKARG_ESCAPE;
```


IOActive™

WDDM kmd Driver Entrypoints – Escape

- pPrivateDriverData is probed and captured
- No length restrictions (e.g. could be ~4 gigs)
- Userland has complete control of its content
- Any embedded pointers in it need to be probed and only used under a try/except

IOActive™

WDDM kmd Driver Entrypoints – Escape

- How do you talk to this from userland?

```
NTSTATUS D3DKMTEscape(
_In_  const D3DKMT_ESCAPE
*pData );
```

```
typedef struct _D3DKMT_ESCAPE {
D3DKMT_HANDLE      hAdapter;
D3DKMT_HANDLE      hDevice;
D3DKMT_ESCAPETYPE  Type;
D3DDDI_ESCAPEFLAGS Flags;
VOID               *pPrivateDriverData;
UINT               PrivateDriverDataSize
; D3DKMT_HANDLE      hContext;
} D3DKMT_ESCAPE;
```

- Publicly documented function. Basically exposes a system call.

IOActive™

WDDM kmd Driver Entrypoints – Escape

```
.text:00000001800014A0 ; Exported entry 1131. D3DKMTEscape
.text:00000001800014A0
.text:00000001800014A0 ; ============== S U B R O U T I N E =======================================
.text:00000001800014A0
.text:00000001800014A0
.text:00000001800014A0                 public ZwGdiDdDDIEscape
.text:00000001800014A0 ZwGdiDdDDIEscape proc near               ; DATA XREF: .text:off_180027984↓o
.text:00000001800014A0                                          ; .pdata:000000018012036C↓o ...
.text:00000001800014A0                 mov     r10, rcx
.text:00000001800014A3                 mov     eax, 1180h
.text:00000001800014A8                 syscall
.text:00000001800014AA                 retn
.text:00000001800014AA ZwGdiDdDDIEscape endp
.text:00000001800014AA
.text:00000001800014AA ; ---------------------------------------------------------------------------
```


IOActive™

WDDM kmd Driver Entrypoints – Render

- DxgkDdiRender
- This callback is at the heart of rendering.
- Allows usermode to tell GPU to render from a command buffer
  - Will generate DMA buffer from command buffer

IOActive™

WDDM kmd Driver Entrypoints – Render

- What does DxgkDdiRender look like?

```
NTSTATUS APIENTRY
DxgkDdiRender(
  _In_     const HANDLE hContext,
  _Inout_  DXGKARG_RENDER
*pRender
)
{ ... }
```

```
typedef struct _DXGKARG_RENDER {
const VOID CONST        *pCommand;
const UINT              CommandLength;
VOID                    *pDmaBuffer;
UINT                    DmaSize;
VOID                    *pDmaBufferPrivateData;
UINT                    DmaBufferPrivateDataSize;
DXGK_ALLOCATIONLIST     *pAllocationList;
UINT                    AllocationListSize;
D3DDDI_PATCHLOCATIONLIST *pPatchLocationListIn;
UINT                    PatchLocationListInSize;
D3DDDI_PATCHLOCATIONLIST *pPatchLocationListOut;
UINT                    PatchLocationListOutSize;
UINT                    MultipassOffset;
UINT                    DmaBufferSegmentId;
PHYSICAL_ADDRESS        DmaBufferPhysicalAddress;
} DXGKARG_RENDER;
```


IOActive™

WDDM kmd Driver Entrypoints – Render

- pCommand buffer is a pointer that comes from userland
- pPatchLocationListIn is a pointer that comes from userland
- MSDN says the following about these:
  “Both the command buffer **pCommand** and the input patch-location list **pPatchLocationListIn** that the user-mode display driver generates are allocated from the user-mode address space and are passed to the display miniport driver untouched. The display miniport driver must use __try/__except code on any access to the buffer and list and must validate the content of the buffer and list before copying the content to the respective kernel buffers.”
- It goes on to give a validation sample.

IOActive™

WDDM kmd Driver Entrypoints – Render

```
__try
  {
    for (Index = 0; Index < AllocationListInSize; AllocationTable++,
          AllocationListIn++, AllocationListOut++, Index++)
    {
      D3DKMT_HANDLE AllocationHandle = AllocationListIn->hAllocation;
      . . .
    }
  }
__except(EXCEPTION_EXECUTE_HANDLER)
  {
    Status = STATUS_INVALID_PARAMETER;
    SAMPLE_LOG_ERROR(  "Exception occurred accessing … Status=0x%I64x", Status);
    goto cleanup;
  }
```


IOActive™

WDDM kmd Driver Entrypoints – Render

- Userland doesn't actually get to specify the command buffer and patch list addresses.
- Dxgkernel allocates them on your behalf when you call D3DKMTCreateContext, but does map it in userland.
- So you can unmap it (VirtualFree), behind the drivers back.
- Hence, why the try/except is needed.
- Given that both command and patch list addresses are in userland you need to watch out for double fetches.
  - Fetch one: dereference and validate
  - Userland changes data
  - Fetch two: dereference and use, double fetch bug, invalidates previous validation

IOActive™

WDDM kmd Driver Entrypoints – Render

- Example of missing try /except

```
static NTSTATUS APIENTRY DxgkDdiRenderNew( CONST HANDLE  hContext, DXGKARG_RENDER  *pRender) {
    if (pRender->CommandLength < sizeof (VBOXWDDM_DMA_PRIVATEDATA_BASEHDR))
    {
        return STATUS_INVALID_PARAMETER;
    }

    PVBOXWDDM_DMA_PRIVATEDATA_BASEHDR pInputHdr = (PVBOXWDDM_DMA_PRIVATEDATA_BASEHDR)pRender-
>pCommand;
    NTSTATUS Status = STATUS_SUCCESS;
VBOXCMDVBVA_HDR* pCmd = (VBOXCMDVBVA_HDR*)pRender->pDmaBufferPrivateData;
switch (pInputHdr->enmCmd) ← no try/except.
    {
...
    }
...
    return STATUS_SUCCESS;
}
```

WDDM kmd Driver Entrypoints – Render

- Example of double fetch

```
static NTSTATUS APIENTRY DxgkDdiRenderNew( CONST HANDLE  hContext, DXGKARG_RENDER  *pRender) {
…
   PVBOXWDDM_DMA_PRIVATEDATA_BASEHDR pInputHdr =
(PVBOXWDDM_DMA_PRIVATEDATA_BASEHDR)pRender->pCommand;
...
 PVBOXWDDM_DMA_PRIVATEDATA_UM_CHROMIUM_CMD pUmCmd = pInputHdr;
…
 PVBOXWDDM_UHGSMI_BUFFER_UI_SUBMIT_INFO pSubmUmInfo = pUmCmd->aBufInfos;
…
 if (pSubmUmInfo->offData >= pAlloc->AllocData.SurfDesc.cbSize
                    || pSubmUmInfo->cbData > pAlloc->AllocData.SurfDesc.cbSize
                    || pSubmUmInfo->offData + pSubmUmInfo->cbData > pAlloc->AllocData.SurfDesc.cbSize)
            {
                WARN(("invalid data"));
                return STATUS_INVALID_PARAMETER;
            }
…
 pSubmInfo->cbBuffer = pSubmUmInfo->cbData;
…
```

Validate

Use

WDDM kmd Driver Entrypoints – Render

- How do you talk to this from userland?

```
NTSTATUS APIENTRY
D3DKMTRender(
_Inout_  D3DKMT_RENDER *pData );
```

```
typedef struct _D3DKMT_RENDER {
  union {
    D3DKMT_HANDLE hDevice;
    D3DKMT_HANDLE hContext;
  };
  UINT                  CommandOffset;
  UINT                  CommandLength;
  UINT                  AllocationCount;
  UINT                  PatchLocationCount;
  VOID                  *pNewCommandBuffer;
  UINT                  NewCommandBufferSize;
  D3DDDI_ALLOCATIONLIST    *pNewAllocationList;
  UINT                  NewAllocationListSize;
  D3DDDI_PATCHLOCATIONLIST *pNewPatchLocationList;
  UINT                  NewPatchLocationListSize;
  D3DKMT_RENDERFLAGS       Flags;
  ULONGLONG                PresentHistoryToken;
  ULONG                 BroadcastContextCount;
  D3DKMT_HANDLE
BroadcastContext[D3DDDI_MAX_BROADCAST_CONTEXT];
  ULONG                 QueuedBufferCount;
  D3DGPU_VIRTUAL_ADDRESS   NewCommandBuffer;
  VOID                  *pPrivateDriverData;
  UINT                  PrivateDriverDataSize;
} D3DKMT_RENDER;
```


IOActive™

WDDM kmd Driver Entrypoints – Render

```
.text:000007FF7FD713F0 ; Exported entry 1151. D3DKMTRender
.text:000007FF7FD713F0
.text:000007FF7FD713F0 ; =============== S U B R O U T I N E =======================================
.text:000007FF7FD713F0
.text:000007FF7FD713F0
.text:000007FF7FD713F0                 public ZwGdiDdDDIRender
.text:000007FF7FD713F0 ZwGdiDdDDIRender proc near               ; DATA XREF: .rdata:off_7FF7FDC24B8↓o
.text:000007FF7FD713F0                                          ; .pdata:000007FF7FDCE090↓o ...
.text:000007FF7FD713F0                 mov     r10, rcx
.text:000007FF7FD713F3                 mov     eax, 1187h
.text:000007FF7FD713F8                 syscall
.text:000007FF7FD713FA                 retn
.text:000007FF7FD713FA ZwGdiDdDDIRender endp
.text:000007FF7FD713FA
.text:000007FF7FD713FA ; ---------------------------------------------------------------------------
```


IOActive™

WDDM kmd Driver Entrypoints – Allocation

- DxgkDdiCreateAllocation
- Dxgkernel calls this callback on userland's behalf to allocate memory.
- It will allocate either system or video memory, depending on flags.

IOActive™

WDDM kmd Driver Entrypoints – Allocation

- What does DxgkDdiCreateAllocation look like?

```
NTSTATUS APIENTRY DxgkDdiCreateAllocation(
  const HANDLE hAdapter,
  DXGKARG_CREATEALLOCATION
*pCreateAllocation
)
{ ... }
```

```
typedef struct
_DXGKARG_CREATEALLOCATION {
  const VOID                *pPrivateDriverData;
  UINT                      PrivateDriverDataSize;
  UINT                      NumAllocations;
  DXGK_ALLOCATIONINFO
*pAllocationInfo;
  HANDLE                    hResource;
  DXGK_CREATEALLOCATIONFLAGS
Flags;
} DXGKARG_CREATEALLOCATION;
```


IOActive™

WDDM kmd Driver Entrypoints – Allocation

- What does DxgkDdiCreateAllocation look like ? (cont.)

```
typedef struct _DXGK_ALLOCATIONINFO {
  VOID                    *pPrivateDriverData;
  UINT                    PrivateDriverDataSize;
  UINT                    Alignment;
  SIZE_T                   Size;
  SIZE_T                   PitchAlignedSize;
  DXGK_SEGMENTBANKPREFERENCE HintedBank;
  DXGK_SEGMENTPREFERENCE     PreferredSegment;
  UINT                    SupportedReadSegmentSet;
  UINT                    SupportedWriteSegmentSet;
  UINT                    EvictionSegmentSet;
  UINT                    MaximumRenamingListLength;
  HANDLE                     hAllocation;
  DXGK_ALLOCATIONINFOFLAGS   Flags;
  DXGK_ALLOCATIONUSAGEHINT   *pAllocationUsageHint;
  UINT                    AllocationPriority;
} DXGK_ALLOCATIONINFO;
```

WDDM kmd Driver Entrypoints – Allocation

- Private driver data is captured from user to kernel.
- There are NumAllocations DXGK_ALLOCATIONINFO structures that userland gets to pass.
- DXGK_ALLOCATIONINFO's private driver data is also captured from user to kernel.
- DxgkDdiOpenAllocation can't be directly called from userland, but its private driver data is the same as provided here.

IOActive™

WDDM kmd Driver Entrypoints – Allocation

- How do you talk to this from userland ?

```
NTSTATUS APIENTRY
D3DKMTCreateAllocation(
  D3DKMT_CREATEALLOCATION
*pData
);
```

```
typedef struct _D3DKMT_CREATEALLOCATION {
  D3DKMT_HANDLE                hDevice;
  D3DKMT_HANDLE                hResource;
  D3DKMT_HANDLE                hGlobalShare;
  const VOID                   *pPrivateRuntimeData;
  UINT                         PrivateRuntimeDataSize;
  const VOID                   *pPrivateDriverData;
  UINT                         PrivateDriverDataSize;
  UINT                         NumAllocations;
D3DDDI_ALLOCATIONINFO  *pAllocationInfo;
D3DKMT_CREATEALLOCATIONFLAGS Flags;
  HANDLE
hPrivateRuntimeResourceHandle;
} D3DKMT_CREATEALLOCATION;
```


IOActive™

WDDM kmd Driver Entrypoints – Allocation

```
.text:000007FF7FD74C20 ; Exported entry 1106. D3DKMTCreateAllocation2
.text:000007FF7FD74C20
.text:000007FF7FD74C20 ; =============== S U B R O U T I N E =======================================
.text:000007FF7FD74C20
.text:000007FF7FD74C20
.text:000007FF7FD74C20                 public NtGdiDdDDICreateAllocation
.text:000007FF7FD74C20 NtGdiDdDDICreateAllocation proc near     ; CODE XREF: D3DKMTCreateAllocation+B5↑p
.text:000007FF7FD74C20                                          ; DATA XREF: .rdata:off_7FF7FDC24B8↓o
.text:000007FF7FD74C20                 mov     r10, rcx
.text:000007FF7FD74C23                 mov     eax, 115Dh
.text:000007FF7FD74C28                 syscall
.text:000007FF7FD74C2A                 retn
.text:000007FF7FD74C2A NtGdiDdDDICreateAllocation endp
.text:000007FF7FD74C2A
.text:000007FF7FD74C2A ; ---------------------------------------------------------------------------
```


IOActive™

WDDM kmd Driver Entrypoints – queryadapter

- The actual type nr in user and driver different
- Dxgkernel does some kind of translation
- All have well defined format
- With well defined length
- Except for DXGKQAITYPE_UMDRIVERPRIVATE
- Driver can implement that one any way it wants

```
typedef enum _DXGK_QUERYADAPTERINFOTYPE {
  DXGKQAITYPE_UMDRIVERPRIVATE          = 0,
  DXGKQAITYPE_DRIVERCAPS               = 1,
  DXGKQAITYPE_QUERYSEGMENT             = 2,
#if (DXGKDDI_INTERFACE_VERSION >= DXGKDDI_INTERFACE_VERSION_WIN7)
  DXGKQAITYPE_ALLOCATIONGROUP          = 3,
  DXGKQAITYPE_QUERYSEGMENT2            = 4,
#endif
#if (DXGKDDI_INTERFACE_VERSION >= DXGKDDI_INTERFACE_VERSION_WIN8)
  DXGKQAITYPE_QUERYSEGMENT3            = 5,
  DXGKQAITYPE_NUMPOWERCOMPONENTS       = 6,
  DXGKQAITYPE_POWERCOMPONENTINFO       = 7,
  DXGKQAITYPE_PREFERREDGPUNODE         = 8,
#endif
#if (DXGKDDI_INTERFACE_VERSION >=
DXGKDDI_INTERFACE_VERSION_WDDM1_3)
  DXGKQAITYPE_POWERCOMPONENTPSTATEINFO  = 9,
  DXGKQAITYPE_HISTORYBUFFERPRECISION    = 10

#endif } DXGK_QUERYADAPTERINFOTYPE;
```


IOActive™

WDDM kmd Driver Entrypoints – queryadapter

- What does DxgkDdiQueryAdapterInfo look like?

```
NTSTATUS APIENTRY
DxgkDdiQueryAdapterInfo(
 HANDLE hAdapter,
 DXGKARG_QUERYADAPTERINFO
*pQueryAdapterInfo )
{ ... }
```

```
typedef struct
_DXGKARG_QUERYADAPTERINFO {
            DXGK_QUERYADAPTERINFOTYP
E Type;  VOID                *pInputData;
            UINT                InputDataSize;
            VOID                *pOutputData;
            UINT                OutputDataSize;
} DXGKARG_QUERYADAPTERINFO;
```


IOActive™

WDDM kmd Driver Entrypoints – queryadapter

- Has interesting entry- and exit points
- Entry points:
  - Data being passed in from userland.
  - Most interesting type for this is DXGKQAITYPE_UMDRIVERPRIVATE.
- Exit points:
  - With Query API's that return large structures from kernel to user, there is the risk of information leaks.
    - Usually happens when a struct is on the stack/heap, no memset is done, and part of one or more members is not initialized (e.g. fixed character buffer that holds a 0-terminated string).

IOActive™

WDDM kmd Driver Entrypoints – queryadapter

- How do you talk to this from userland ?

```
NTSTATUS
D3DKMTQueryAdapterInfo(
D3DKMT_QUERYADAPTERINFO
*pData
);
```

```
typedef struct
_D3DKMT_QUERYADAPTERINFO {
  D3DKMT_HANDLE          hAdapter;
  KMTQUERYADAPTERINFOTYPE Type;
  VOID                   *pPrivateDriverData;
  UINT                   PrivateDriverDataSize;
} D3DKMT_QUERYADAPTERINFO;
```


IOActive™

WDDM kmd Driver Entrypoints – queryadapter

```
.text:000007FF7FD789E0 ; Exported entry 1145. D3DKMTQueryAdapterInfo
.text:000007FF7FD789E0
.text:000007FF7FD789E0 ; =============== S U B R O U T I N E =======================================
.text:000007FF7FD789E0
.text:000007FF7FD789E0
.text:000007FF7FD789E0                 public NtGdiDdDDIQueryAdapterInfo
.text:000007FF7FD789E0 NtGdiDdDDIQueryAdapterInfo proc near    ; DATA XREF: .rdata:off_7FF7FDC24B8↓o
.text:000007FF7FD789E0                 mov     r10, rcx
.text:000007FF7FD789E3                 mov     eax, 1181h
.text:000007FF7FD789E8                 syscall
.text:000007FF7FD789EA                 retn
.text:000007FF7FD789EA NtGdiDdDDIQueryAdapterInfo endp
.text:000007FF7FD789EA
.text:000007FF7FD789EA ; ---------------------------------------------------------------------------
```


IOActive™

WDDM kmd Driver Entrypoints – Best Practices

- Out of bound read ← very common
  - This means bluescreen in kernel
  - Could happen, even for a single byte out of bound read
- Don’t ship debug code
  - Remove DbgPrint calls
  - And surrounding code (e.g. data that will be printed by formatstring)
    - Ends up in binary otherwise. Could contains exploitable bugs.
  - #ifdef debug
- Use kernel safe integer library routines (e.g. RtlUIntAdd)
  - Please don’t roll your own …

IOActive™

Full userland Program to Talk to this Stuff

- Slightly more difficult than it looks.
- The API's are documented on msdn, and exported from gdi32.dll.
- The data structures are documented on msdn.
- Meant for OpenGL ICD (Installable client driver) drivers
  - No headers for this stuff:
    - Need to LoadLibrary/GetProcAddress
    - There is a devkit for this, but, ..., MSDN says: "**Note** To obtain a license for the OpenGL ICD Development Kit, contact the OpenGL Issues team."
    - Given that it is documented, getting a (partially) working implementation is pretty easy.
    - Or you could use the COM APIs. ☺

IOActive™

Sniffing/snooping Private Data

- Since data send from umd to kmd is not structured in any way, we need to see what gets send to kmd under normal conditions.
- To get an idea of what the protocol looks like for any given driver
- Hook APIs:
  - D3DKMTEscape
  - D3DKMTRender
  - D3DKMTCreateAllocation
  - D3DKMTQueryAdapterInfo

IOActive™

Sniffing/snooping Private Data

- Tool/demo
- Release!
- Running against PowerPoint seems to give pretty good results.

IOActive™

Putting It all Together

- Fuzzing
- Reverse engineering

IOActive™

Putting It all Together – Fuzzing

- Mutating fuzzer
- Starting off with sniffed data (template per driver)
- Mutate data
- Loop
- Combine this with reversing
  - If (embedded_len != PrivateDataSize) bail;
  - Checksums
- ➔ Bugs!

Putting It all Together – Reverse Engineering

"If the process of reverse engineering Windows drivers could be modeled as a discrete task, 90% would be understanding how Windows works and 10% would be understanding assembly code.“ – Bruce Dang

IOActive™

WDDM kmd Driver – Reverse Engineering

- As shown before, all the driver does as part of it's initialization is call DxgkInitialize() or DxgkInitializeDisplayOnlyDriver().
- And pass it a callback table (DRIVER_INITIALIZATION_DATA)
- When looking at the driver in a disassembler no call to these functions is observed.
- These functions are inlined

IOActive™

WDDM kmd Driver – Reverse Engineering

- So what does it look like? (with symbols)

```
00406000 ; int __stdcall DriverEntry(_DRIVER_OBJECT *pDriverObject, _UNICODE_STRING *pRegistryPath)
00406000 _DriverEntry@8  proc near               ; CODE XREF: GsDriverEntry(x,x)+B↑j
00406000
00406000 InitialData     = _KMDDOD_INITIALIZATION_DATA ptr -0A4h
00406000 pDriverObject   = dword ptr  8
00406000 pRegistryPath   = dword ptr  0Ch
00406000
00406000                 push    ebp
00406001                 mov     ebp, esp
00406003                 sub     esp, 0A4h
00406009                 push    0A0h            ; size_t
0040600E                 lea     eax, [ebp+InitialData.DxgkDdiAddDevice]
00406014                 push    0               ; int
00406016                 push    eax             ; void *
00406017                 call    _memset
0040601C                 add     esp, 0Ch
0040601F                 mov     [ebp+InitialData.Version], 4002h
....
00406104                 mov     [ebp+InitialData.DxgkDdiSystemDisplayEnable], offset ?BddDdiSystemDisplayEnable@@YGJPAXIPAU_DXGKARG_SYSTEM_DI
0040610B                 mov     [ebp+InitialData.DxgkDdiSystemDisplayWrite], offset ?BddDdiSystemDisplayWrite@@YGXPAX0IIIII@Z ; BddDdiSystemD
00406112                 call    _DxgkInitializeDisplayOnlyDriver@12 ; DxgkInitializeDisplayOnlyDriver(x,x,x)
00406117                 mov     esp, ebp
00406119                 pop     ebp
0040611A                 retn    8
0040611A  DriverEntry@8  endp
```


IOActive™

```
int __stdcall DxgkInitializeDisplayOnlyDriver(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING *RegistryPath, _KMDDOD_INITIALIZATION_DATA *InitData)
{
  char v3; // bl@1
  _KMDDOD_INITIALIZATION_DATA *v4; // edi@3
  unsigned int v5; // eax@4
  int result; // eax@8
  int v7; // esi@9
  int (__stdcall *DpiInitialize)(_DRIVER_OBJECT *, _UNICODE_STRING *, _KMDDOD_INITIALIZATION_DATA *); // [sp+8h] [bp-Ch]@1
  void *FileObject; // [sp+Ch] [bp-8h]@1
  _DEVICE_OBJECT *DxgDeviceObject; // [sp+10h] [bp-4h]@1

  v3 = 0;
  FileObject = 0;
  DxgDeviceObject = 0;
  DpiInitialize = 0;
  if ( DriverObject && RegistryPath && (v4 = InitData) != 0 )
  {
    v5 = InitData->Version;
    if ( InitData->Version == 4178 || v5 == 4179 || v5 == 8197 || v5 >= 0x300E )
    {
      v7 = DlpLoadDxgkrnl((_FILE_OBJECT **)&FileObject, &DxgDeviceObject);
      if ( v7 >= 0 || v7 == -1073741554 )
      {
        v3 = v7 != -1073741554;
        v7 = DlpCallSyncDeviceIoControl(
               DxgDeviceObject,
               0x230043u,
               1,
               0,
               0,
               &DpiInitialize,
               4u,
               (unsigned int *)&InitData);
      }
      if ( FileObject )
        ObfDereferenceObject();
      if ( v7 < 0 )
      {
        if ( v3 )
          DlpUnloadDxgkrnl();
      }
      else
      {
        MiniportStartDevice = (int)v4->DxgkDdiStartDevice;
        v4->DxgkDdiStartDevice = DlpStartDevice;
        v7 = DpiInitialize(DriverObject, RegistryPath, v4);
      }
      result = v7;
    }
    else
    {
      result = -1073741735;
    }
  }
  else
  {
    result = -1073741811;
  }
  return result;
}
```

WDDM kmd Driver – Reverse Engineering

```
int __stdcall DlpLoadDxgkrnl(_FILE_OBJECT **FileObject, _DEVICE_OBJECT **DeviceObject)
{
  const wchar_t *v2; // eax@1
  void *v3; // edi@1
  int v4; // esi@2
  char v5; // bl@5
  _UNICODE_STRING DeviceName; // [sp+8h] [bp-1Ch]@5
  _UNICODE_STRING DriverServiceName; // [sp+10h] [bp-14h]@3
  signed __int64 DelayDuration; // [sp+18h] [bp-Ch]@8
  int v10; // [sp+20h] [bp-4h]@1

  v10 = 10;
  v2 = DlpGetServiceNameInSystemSpace();
  v3 = (void *)v2;
  if ( v2 )
  {
    RtlInitUnicodeString(&DriverServiceName, v2);
    v4 = ZwLoadDriver(&DriverServiceName);
    ExFreePoolWithTag(v3, 0);
    if ( v4 >= 0 || v4 == -1073741554 )
    {
      v5 = v4 != -1073741554;
      RtlInitUnicodeString(&DeviceName, L"\\Device\\DxgKrnl");
      while ( 1 )
      {
        v4 = IoGetDeviceObjectPointer(&DeviceName, 0xC0000000u, FileObject, DeviceObject);
        if ( v4 >= 0 )
          break;
        if ( v5 )
        {
          DlpUnloadDxgkrnl();
          return v4;
        }
        DelayDuration = -50000i64;
        KeDelayExecutionThread(0, 0, (PLARGE_INTEGER)&DelayDuration);
        --v10;
        if ( !v10 )
            return v4;
        }
        if ( !v5 )
          v4 = -1073741554;
      }
    }
    else
    {
      v4 = -1073741801;
    }
    return v4;
}
```

```
const wchar_t *__stdcall DlpGetServiceNameInSystemSpace()
{
  const wchar_t *result; // eax@1

  result = (const wchar_t *)ExAllocatePoolWithTag(PagedPool, 0x78u, 0x706D6C44u);
  if ( result )
    memcpy((void *)result, L"\\Registry\\Machine\\System\\CurrentControlSet\\Services\\DXGKrnl", 0x78u);
  return result;
}
```

IOActive™

WDDM kmd Driver – Reverse Engineering

- Loads Dxgkrnl.sys (it should already be loaded)
- Gets a pointer to it's device object
- Issues ioctl 0x230043 on it (video device, function 10, method neither, FILE_ANY_ACCESS)
- Hands back a function pointer to be used to register the callback
- Call that function pointer with DRIVER_INITIALIZATION_DATA or PKMDDOD_INITIALIZATION_DATA struct as argument

IOActive™

WDDM kmd Driver – Reverse Engineering

- The table itself can be created/stored several different ways
  - tabled stored globally
  - Created on the stack
  - Specific function fills in DRIVER_INITIALIZATION_DATA
- Or filled out in a local stack buffer right before calling DxgkInitialize()
- Finding the code that does this is usually pretty simple. It'll happen early on, usually in DriverEntry() or some function it calls.
- Tends to look like this:

IOActive™

WDDM kmd Driver – Reverse Engineering

```
lea     ecx, [ebp+var_A4]
push    ecx
push    edx
push    eax
mov     [ebp+var_A4], 300Eh
mov     [ebp+var_A0], offset sub_402C80
mov     [ebp+var_9C], offset sub_402D50
mov     [ebp+var_98], offset sub_403250
mov     [ebp+var_94], offset sub_4032C0
mov     [ebp+var_90], offset sub_4032E0
mov     [ebp+var_8C], offset sub_4032F0
mov     [ebp+var_88], offset sub_4036F0
mov     [ebp+var_84], offset sub_4037E0
mov     [ebp+var_80], offset sub_403840
mov     [ebp+var_7C], offset sub_4038B0
mov     [ebp+var_78], offset sub_4038C0
mov     [ebp+var_74], offset sub_4038D0
mov     [ebp+var_70], offset nullsub_1
mov     [ebp+var_6C], offset sub_4038F0
mov     [ebp+var_68], offset sub_403930
mov     [ebp+var_64], offset nullsub_2
mov     [ebp+var_60], offset sub_403950
mov     [ebp+var_5C], offset sub_4051C0
mov     [ebp+var_58], offset sub_405550
mov     [ebp+var_54], offset sub_405650
mov     [ebp+var_50], offset sub_405720
mov     [ebp+var_4C], offset sub_4061A0
mov     [ebp+var_48], offset sub_406250
mov     [ebp+var_44], offset sub_406320
mov     [ebp+var_40], offset sub_4065B0
mov     [ebp+var_3C], offset sub_4066E0
mov     [ebp+var_38], offset sub_408BA0
mov     [ebp+var_34], offset sub_4067D0
mov     [ebp+var_30], offset sub_4067E0
mov     [ebp+var_2C], offset sub_4068E0
mov     [ebp+var_28], offset sub_408170
mov     [ebp+var_24], offset sub_408180
mov     [ebp+var_20], offset sub_4083E0
mov     [ebp+var_1C], offset sub_4083F0
mov     [ebp+var_18], offset nullsub_3
mov     [ebp+var_10], offset sub_406900
call    sub_42A64C
```

```
__int64 __fastcall sub_23870(__int64 a1)
{
  __int64 result; // rax@1

  *(_DWORD *)a1 = 0x3008;
  *(_QWORD *)(a1 + 8) = DxgkDdiAddDevice;
  *(_QWORD *)(a1 + 16) = DxgkDdiStartDevice;
  *(_QWORD *)(a1 + 24) = sub_50DE0;
  *(_QWORD *)(a1 + 32) = sub_50590;
  *(_QWORD *)(a1 + 40) = sub_1C3A0;
  *(_QWORD *)(a1 + 48) = sub_1C460;
  *(_QWORD *)(a1 + 56) = sub_1C350;
  *(_QWORD *)(a1 + 64) = sub_53260;
  *(_QWORD *)(a1 + 72) = sub_53340;
  *(_QWORD *)(a1 + 80) = sub_535A0;
  *(_QWORD *)(a1 + 88) = sub_25030;
  *(_QWORD *)(a1 + 104) = sub_1C550;
  *(_QWORD *)(a1 + 112) = sub_510C0;
  *(_QWORD *)(a1 + 136) = sub_529C0;
  *(_QWORD *)(a1 + 152) = sub_1A2B0;
  *(_QWORD *)(a1 + 160) = sub_1A560;
  *(_QWORD *)(a1 + 168) = sub_197A0;
  *(_QWORD *)(a1 + 176) = sub_51080;
  *(_QWORD *)(a1 + 184) = sub_53E80;
  *(_QWORD *)(a1 + 192) = sub_53ED0;
  *(_QWORD *)(a1 + 200) = sub_53630;
  *(_QWORD *)(a1 + 208) = sub_545A0;
  *(_QWORD *)(a1 + 216) = sub_304B0;
  *(_QWORD *)(a1 + 224) = sub_31460;
  *(_QWORD *)(a1 + 232) = sub_544F0;
  *(_QWORD *)(a1 + 256) = sub_527D0;
  *(_QWORD *)(a1 + 264) = sub_1C5B0;
  result = a1;
  *(_QWORD *)(a1 + 248) = sub_30DB0;
  return result;
}
```


IOActive™

WDDM kmd Driver – Reverse Engineering

- Here's what the structure looks like in C:

```
typedef struct _DRIVER_INITIALIZATION_DATA {
  ULONG                                Version;
  PDXGKDDI_ADD_DEVICE                  DxgkDdiAddDevice;
  PDXGKDDI_START_DEVICE                DxgkDdiStartDevice;
  PDXGKDDI_STOP_DEVICE                 DxgkDdiStopDevice;
  PDXGKDDI_REMOVE_DEVICE               DxgkDdiRemoveDevice;
...
} DRIVER_INITIALIZATION_DATA, *PDRIVER_INITIALIZATION_DATA;
```

- Mapping this to IDA disassembly and renaming the functions to something meaningful is pretty easy

WDDM kmd Driver – Reverse Engineering

- Userland data passed in (PrivateData)
  - Driver gets to handle this any way it sees fit
  - Usually:
    - Feels like (simple) network protocol reverse engineering
    - Usually comes with a header
      - Type
      - Length
      - Value
    - Switch case or nested if/else to handle values for types

WDDM kmd Driver – Reverse Engineering

- Example of a typical case:

```
int __fastcall DxgkDdiEscape(PVOID hAdapter, __int64 pEscape)
{
  __int64 v2; // rbx@1
  PVOID v3; // rdi@1
  int result; // eax@3

  v2 = pEscape;
  v3 = hAdapter;
  DbgPrintEx(78i64, 3i64, qword_140015680, "Vd3dkAdapter::DxgkDdiEscape");
  if ( *(_DWORD *)(v2 + 24) < 4u )
  {
LABEL_18:
    result = 0xC000000D;
  }
  else
  {
    switch ( **(_DWORD **)(v2 + 16) )
    {
      case 3:
        result = sub_14000DE70(v3, v2);
        break;
      case 0:
        result = sub_14000C340(v3, v2);
        break;
      case 1:
        result = sub_14000C370(v3, v2);
        break;
      case 2:
        result = sub_14000C380(v3, v2);
        break;
      case 4:
        result = sub_14000FB20(v3);
        break;
      case 5:
        result = sub_14000D880(v3, v2);
        break;
      case 6:
        result = sub_14000CAC0(v3, v2);
        break;
      case 7:
        result = sub_14000CB20(v3, v2);
        break;
      case 8:
        result = sub_14000CB60(v3, v2);
        break;
      case 9:
        result = sub_14000CB70(v3, v2);
        break;
      case 0xA:
        result = sub_14000C390(v3, v2);
        break;
      case 0xB:
        result = sub_14000CFE0(v3, v2);
        break;
      case 0xC:
        result = sub_14000ED50(v3, v2);
        break;
      case 0xD:
        result = sub_14000D800(v3, v2);
        break;
      case 0xE:
        result = sub_14000EDD0(v3, v2);
        break;
      default:
        goto LABEL_18;
    }
  }
  return result;
}
```


IOActive™

WDDM kmd Driver – Reverse Engineering

- Functions (e.g. DxgkDdiEscape) tend to return STATUS_INVALID_PARAMETER (0xc000000d) when userland provided data couldn't be parsed
  - Return value gets picked by driver
  - If you see this often/constantly during fuzzing, it's usually a sign you've hit some kind of validation/checksum.
    - Dig into assembly to figure out why, and adjust your fuzzer accordingly.

Q&A

IOActive™