Windows Kernel Graphics Driver Attack Surface

Ilja van Sprundel
Director of Penetration Testing
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 😊
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
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, …)
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
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
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.
WDDM kmd Driver

• So what does a WDDM kmd driver look like?

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

- Very similar variant of this
- Calls DxgkInitializeDisplayOnlyDriver() instead of 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
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
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.
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

```c
D3DKMT_OPENADAPTERFROMHDC oafh;
memset(&oafh, 0x00, sizeof(oafh));
oafh.hDc = GetDC(NULL);
D3DKMTOpenAdapterFromHdc(&oafh);
```
WDDM kmd Driver Entrypoints – Initialization

- Get a device handle out of the adapter handle
- Fill in D3DKMT_CREATEDEVICE data structure
- Call D3DKMTCREATEDEVICE

```c
D3DKMT_CREATEDEVICE cdev;
memset(&cdev, 0x00, sizeof(cctx));
cdev.hAdapter = oafh.hAdapter;
D3DKMTCREATEDEVICE(&cdev);
```
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.
WDDM kmd Driver Entrypoints – Initialization

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

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

- DxgkDdiCreateContext kernel entry point
WDDM kmd Driver Entry Points – 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.

```c
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 *pPatchLocationList;
    UINT PatchLocationListSize;
    D3DGPU_VIRTUAL_ADDRESS CommandBuffer;
} D3DKMT_CREATECONTEXT;
```
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
WDDM kmd Driver Entrypoints – Escape

• What does DxgkDdiEscape look like?

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

```c
typedef struct
    _DXGKARG_ESCAPE {
        HANDLE    hDevice;
        D3DDDI_ESCAPEFLAGS Flags;
        VOID     *pPrivateDriverData;
        UINT     PrivateDriverDataSize;
        HANDLE    hContext;
    } DXGKARG_ESCAPE;
```
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
WDDM kmd Driver Entrypoints – Escape

• How do you talk to this from userland?

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

• Publicly documented function. Basically exposes a system call.

```c
typedef struct _D3DKMT_ESCAPE {
    D3DKMT_HANDLE     hAdapter;
    D3DKMT_HANDLE     hDevice;
    D3DKMT_ESCAPE_TYPE Type;
    D3DDDI_ESCAPEFLAGS Flags;
    VOID              *pPrivateDriverData;
    UINT              PrivateDriverDataSize;
    D3DKMT_HANDLE     hContext;
} D3DKMT_ESCAPE;
```
WDDM kmd Driver Entrypoints – Escape

public ZwGdiDdDDIEscape
ZwGdiDdDDIEscape proc near
    mov   r10, rcx
    mov   eax, 11B0h
    syscall
    ret

ZwGdiDdDDIEscape endp
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
WDDM kmd Driver Entrypoints – Render

- What does DxgkDdiRender look like?

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

define 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;
```
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.
__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%x", Status);
    goto cleanup;
}
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 D3DKMTCreatContext, 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
WDDM kmd Driver Entrypoints – Render

- Example of missing try /except

```c
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

```c
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;

        WARN("invalid data");
    return STATUS_INVALID_PARAMETER;

    pSubmInfo->cbBuffer = pSubmUmInfo->cbData;
    return STATUS_SUCCESS;
}
```
WDDM kmd Driver Entrypoints – Render

- How do you talk to this from userland?

```c
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;
```
WDDM kmd Driver Entrypoints – Render

PUBLIC ZwGdiDdDDIRender

ZwGdiDdDDIRender proc near
;

mov    r10, rcx
mov    eax, 1187h
syscall
retn

ZwGdiDdDDIRender endp
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.
WDDM kmd Driver Entry Points – Allocation

- What does DxgkDdiCreateAllocation look like?

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

```c
typedef struct _DXGKARG_CREATEALLOCATION {
    const VOID *pPrivateDriverData;
    UINT PrivateDriverDataSize;
    UINT NumAllocations;
    DXGK_ALLOCATIONINFO *pAllocationInfo;
    HANDLE hResource;
    DXGK_CREATEALLOCATIONFLAGS Flags;
} DXGKARG_CREATEALLOCATION;
```
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.
WDDM kmd Driver Entrypoints – Allocation

- How do you talk to this from userland?

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

```c
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;
```
WDDM kmd Driver Entrypoints – Allocation

.text:00007FF7FD74C20 ; Exported entry 1106. D3DKMTCreateAllocation2
.text:00007FF7FD74C20
.text:00007FF7FD74C20 ; =============== SUBROUTINE ====================
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20
.text:00007FF7FD74C20

public MtGdiDdDDICreateAllocation

MtGdiDdDDICreateAllocation proc near ; CODE XREF: D3DKMTCreateAllocation+B5p

mov r10, rcx
mov eax, 115Dh
syscall
retn

MtGdiDdDDICreateAllocation endp
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,
    #endif
    #if (DXGKDDI_INTERFACE_VERSION >= DXGKDDI_INTERFACE_VERSION_WIN8)
        DXGKQAITYPE_QUERYSEGMENT2 = 4,
    #endif
    #if (DXGKDDI_INTERFACE_VERSION >= DXGKDDI_INTERFACE_VERSION_WDDM1_3)
        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;
WDDM kmd Driver Entrypoints – queryadapter

• What does DxgkDdiQueryAdapterInfo look like?

```c
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;
```
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). 

WDDM kmd Driver Entrypoints – queryadapter

- How do you talk to this from userland?

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

```c
typedef struct
_D3DKMT_QUERYADAPTERINFO {
D3DKMT_HANDLE hAdapter;
KMTQUERYADAPTERINFOTYPE Type;
VOID *pPrivateDriverData;
UINT PrivateDriverDataSize;
} D3DKMT_QUERYADAPTERINFO;
```
WDDM kmd Driver Entrypoints – queryadapter

.text:00007FF7FD789E0 ; Exported entry 1145. D3DKMTQueryAdapterInfo
.text:00007FF7FD789E0
.text:00007FF7FD789E0
.text:00007FF7FD789E0
.text:00007FF7FD789E0
.text:00007FF7FD789E0
.text:00007FF7FD789E0
.text:00007FF7FD789E0

public _MtGdiDdDDIQueryAdapterInfo

_MtGdiDdDDIQueryAdapterInfo proc near ; DATA XREF: .rdata:off_7FF7FDC2488↓

    mov r10, rcx
    mov eax, 1181h
    syscall
    ret

_MtGdiDdDDIQueryAdapterInfo endp

.text:00007FF7FD789EA
.text:00007FF7FD789EA
.text:00007FF7FD789EA


WDDM kmd Driver Entrypoints – Best Practices

• Out of bound read \(\rightarrow\) 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 …
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. 😊
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
Sniffing/snooping Private Data

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

- Fuzzing
- Reverse engineering
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!
"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.\textendash; Bruce Dang
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
WDDM kmd Driver – Reverse Engineering

- So what does it look like? (with symbols)
int _stdcall DbgInitializeDisplayOnlyDriver(_DRIVER_OBJECT *DriverObject, _UNICODE_STRING *RegistryPath, _KMODD_INITIALIZATION_DATA *InitData) 
{
    char w0; // b101
    _KMODD_INITIALIZATION_DATA w4; // edi00
    unsigned int w5; // eax04
    int result; // edx08
    int w7; // esi09
    int ( _stdcall *pInitDevice)( _DRIVER_OBJECT *, _UNICODE_STRING *, _KMODD_INITIALIZATION_DATA *); // [sp+8h] [bp-Ch]01
    void *pFileObject; // [sp+Ch] [bp-8h]01
    _DEVICE_OBJECT *DkgDeviceObject; // [sp+10h] [bp-4h]01

    w3 = 0;
    FileObject = 0;
    DkgDeviceObject = 0;
    pInitDevice = 0;
    if (DriverObject && RegistryPath && (w4 = InitData) != 0 )
    {
        w5 = InitData->Version;
        if (InitData->Version == 0x78 || w5 == 0x79 || w5 == 0x0)
        {
            w7 = DblogDkgRemove(_FILE_OBJECT **)FileObject, &DkgDeviceObject);
            if (w7 >= 0) w7 = -10/3745554;
            if (FileObject)
                DerefReferenceObject();
            if (w7 < 0)
                DblogUnloadDkg();
            else
            {
                MiniportStartDevice = (int)(w4->DkgDdiStartDevice);
                w4->DkgDdiStartDevice = DblogStartDevice;
                w7 = pInitDevice(DriverObject, RegistryPath, w4);
            }
        }
        else
        {
            result = w7;
        }
        return result;
    }
}
WDDM kmd Driver – Reverse Engineering

```c
int __stdcall DplLoadDxgkrnl(FILE_OBJECT **FileObject, _DEVICE_OBJECT **DeviceObject)
{
    const wchar_t *w2; // eax81
    void *w3; // edi81
    int w4; // esi92
    char w5; // bl95
    UNICODE_STRING DeviceName; // [sp-80] [bp-10h]95
    UNICODE_STRING DriverServiceName; // [sp+10h] [bp-14h]93
    Signed __int64 DelayDuration; // [sp+14h] [bp-Ch]98
    int w6; // [sp+2Ch] [bp-4h]92
    uint w7 = 10;
    w2 = DpGetServiceNameInSystemSpace();
    w3 = (void *)&w7;
    if ( w2 )
    {
        RtlInitUnicodeString(&DriverServiceName, w2);
        w4 = 2;//loadDriver(&DriverServiceName);
        ExFreePoolWithTag(w5, 0);
        if ( w4 >= 0 || w4 == -1073741554 )
        { w5 = w4 + 1073741554;
          RtlInitUnicodeString(&DeviceName, L"\Device\Dxgkrnl\"");
          while ( 1 )
          {
            w4 = IoGetDeviceObjectPointer(&DeviceName, 0xC0000000u, FileObject, DeviceObject);
            if ( w4 > 0 )
              break;
            if ( w5 )
            { DplUnloadDxgkrnl();
              return w4;
            }
            DelayDuration = -500000000000;
            KeDelayExecutionThread(0, 0, (PLARGE_INTEGER)&DelayDuration);
            ---198;
            if ( w4 )
              return w4;
            if ( w5 )
            { w4 = -1073741554;
              return w4;
            }
            else
            { w4 = -1073741881;
              return w4;
            }
        }
    }
```
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
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:
WDDM kmd Driver – Reverse Engineering

lea ecx, [ebp+var_04]
push ecx
push edx
push eax
mov [ebp+var_04], 3006h
mov [ebp+var_08], offset sub_403500
mov [ebp+var_0c], offset sub_403505
mov [ebp+var_10], offset sub_402250
mov [ebp+var_14], offset sub_402250
mov [ebp+var_18], offset sub_402250
mov [ebp+var_1c], offset sub_402250
mov [ebp+var_20], offset sub_402250
mov [ebp+var_24], offset sub_402250
mov [ebp+var_28], offset sub_402250
call sub_406900

__int64 __fastcall sub_23870(__int64 a1)
{
__int64 result; // rax@1
/*_QUARD */a1 = 0x3006;
/*_QUARD */(a1 + 8) = DxF0d11d0fDD;
/*_QUARD */(a1 + 16) = DxF0d11d123Device;
/*_QUARD */(a1 + 24) = sub_403500;
/*_QUARD */(a1 + 32) = sub_403505;
/*_QUARD */(a1 + 40) = sub_402250;
/*_QUARD */(a1 + 48) = sub_402250;
/*_QUARD */(a1 + 56) = sub_402250;
/*_QUARD */(a1 + 64) = sub_402250;
/*_QUARD */(a1 + 72) = sub_402250;
/*_QUARD */(a1 + 80) = sub_402250;
/*_QUARD */(a1 + 88) = sub_402250;
/*_QUARD */(a1 + 96) = sub_402250;
/*_QUARD */(a1 + 104) = sub_402250;
/*_QUARD */(a1 + 112) = sub_402250;
/*_QUARD */(a1 + 120) = sub_402250;
/*_QUARD */(a1 + 128) = sub_402250;
/*_QUARD */(a1 + 136) = sub_402250;
/*_QUARD */(a1 + 144) = sub_402250;
/*_QUARD */(a1 + 152) = sub_402250;
/*_QUARD */(a1 + 160) = sub_402250;
/*_QUARD */(a1 + 168) = sub_402250;
/*_QUARD */(a1 + 176) = sub_402250;
/*_QUARD */(a1 + 184) = sub_402250;
/*_QUARD */(a1 + 192) = sub_402250;
/*_QUARD */(a1 + 200) = sub_402250;
/*_QUARD */(a1 + 208) = sub_402250;
/*_QUARD */(a1 + 216) = sub_402250;
/*_QUARD */(a1 + 224) = sub_402250;
/*_QUARD */(a1 + 232) = sub_402250;
/*_QUARD */(a1 + 240) = sub_402250;
/*_QUARD */(a1 + 248) = sub_402250;
result = a1;
/*_QUARD */(a1 + 248) = sub_300600;
return result;
WDDM kmd Driver – Reverse Engineering

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

```c
typedef struct _DRIVER_INITIALIZATION_DATA {
    ULONG Version;
    PDXGKDDI_ADD_DEVICE DxgkDdiAddDevice;
    PDXGKDDI_START_DEVICE DxgkDdiStartDevice;
    PDXGKDDI_STOP_DEVICE DxgkDdiStopDevice;
    PDXGKDDI_REMOVEDEVICE 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:
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