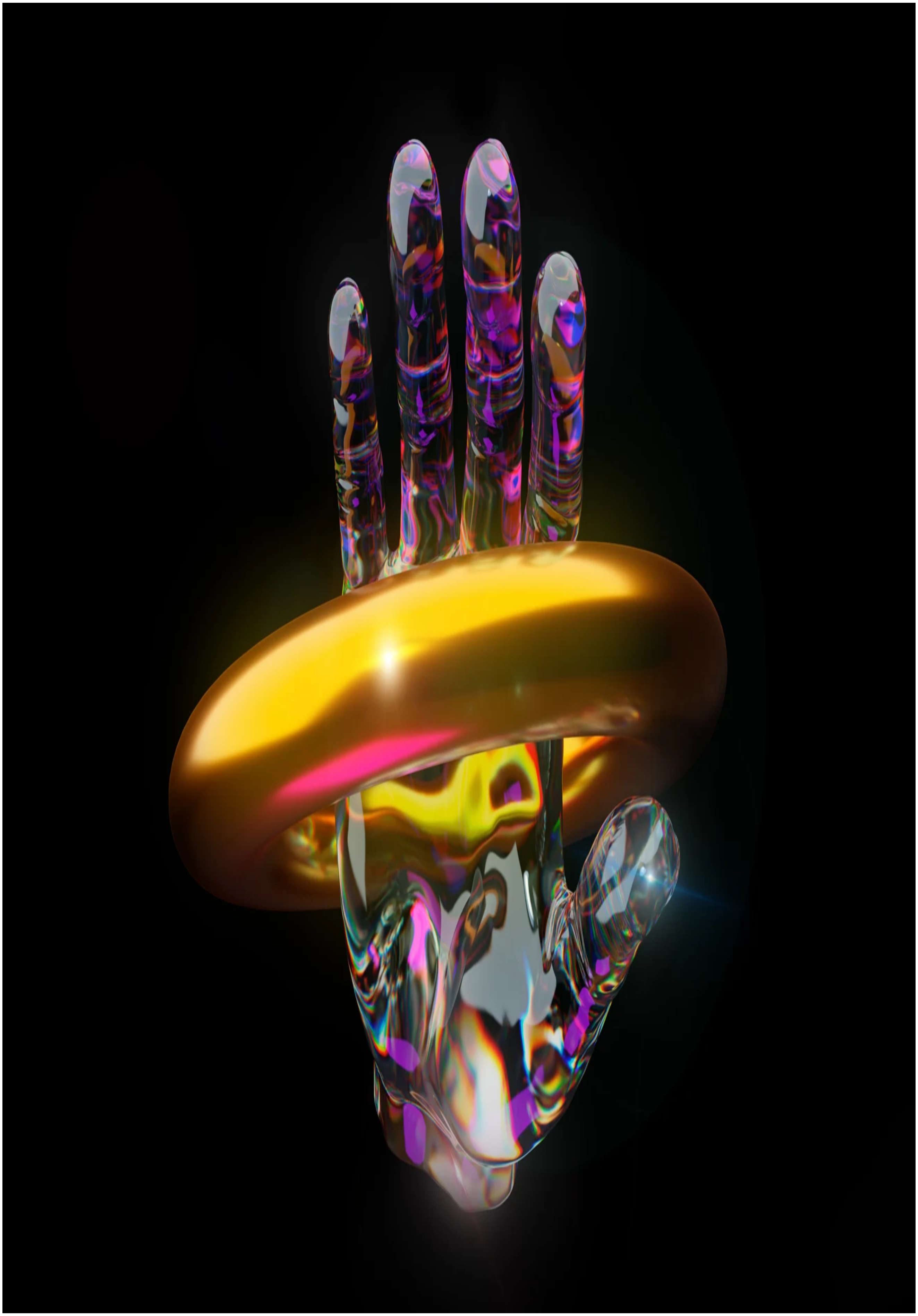
DirectX/HyperV; An Offensive View

fluidattacks.com/blog/offensive-hyperv-directx-1

A Black Hat talk follow up



This year I attended Black Hat USA. The available talks were diverse, all of them inviting and some of them particularly attractive for my current field of work, which is currently mainly focused on advanced topics on Red Teaming and Exploit Development.

One of the talks I found most interesting was DirectX: The new Hyper-V Attack surface, presented by Zhenhao Hong (@rthhh17). In that talk, four vulnerabilities were presented (CVE-2021-43219, CVE-2022-21898, CVE-2022-21912 and CVE-2022-21918) regarding bugs like Null Pointer Dereference, Arbitrary Address Read and Arbitrary Address Write, which included a few lines of the PoC (Proof-Of-Concept) code to trigger each vulnerability.

Also, it was presented an overview of the architecture of Hyper-V DirectX components and a proposed fuzzing methodology to find new vulnerabilities.

In this post(s) I will try to follow up with that research and overcome expected shortcomings of the talk due to time restrictions:

- There is no public access to the PoC codes.
- There is no public access to the fuzzing artifacts.
- The infrastructure to perform research on that specific environment was also not covered.
- Hyper-V → DirectX integration is a work-in-progress for Microsoft, so many of the things mentioned in that talk are no longer working in the current version of Windows 11.

Setting up environment

We have already covered a post to set up a basic environment to perform remote kernel debugging. It involved creating a virtual machine, enabling debug mode using a network connection and plugging in the debugger. That could be done using a single computer.

This case is different. We need to debug a DirectX GPU adapter on a Windows machine acting as hypervisor using Hyper-V with a VM running Linux. Enabling virtualized extensions (**VT-x**) in a Windows VM can enable a nested Hyper-V, but the DirectX adapter will not be visible. After testing different scenarios, I ended needing to use two laptops.

The first laptop will be the debuggee (`host1`). In that laptop, the latest version of Windows 11 was installed:

```
PS C:\Users\aroldan> hostname
host1
PS C:\Users\aroldan> Get-ComputerInfo | select WindowsBuildLabEx,OSVersion

WindowsBuildLabEx                          OsVersion
-----------------                          ---------
22000.1.amd64fre.co_release.210604-1628 10.0.22000
```

In that machine, WSL was installed along with Kali as guest VM:

```
PS C:\Users\aroldan> wsl --install -d kali-linux
Downloading: Kali Linux Rolling
Installing: Kali Linux Rolling
Kali Linux Rolling has been installed.
Launching Kali Linux Rolling...
PS C:\Users\aroldan>
```

The latest stable kernel used on WSL for the guest machines is `5.10.102.1-microsoft-standard-WSL2`. However, I wanted to use the latest version available of WSL, so I built it to use it. To the date of the exercise, the latest version was `5.15.57.1`.

```
aroldan@host1:~$ hostname
host1
aroldan@host1:~$ uname -a
Linux host1 5.15.57.1-fluidattacks-standard-WSL2
aroldan@host1:~$
```

Make sure that you have partitionable GPUs on the host using `Get-VMHostPartitionableGpu`:

```
PS C:\Users\aroldan> Get-VMHostPartitionableGpu


Name                    : \\?\PCI#VEN_8086&DEV_9B41&SUBSYS_22BE17AA&REV_02#3&11583659&0&10#{064092b3-625e-43bf-9eb5-dc84589
                          7dd59}\GPUPARAV
ValidPartitionCounts    : {32}
PartitionCount          : 32
TotalVRAM               : 1000000000
AvailableVRAM           : 1000000000
MinPartitionVRAM        : 0
MaxPartitionVRAM        : 1000000000
OptimalPartitionVRAM    : 1000000000
TotalEncode             : 18446744073709551615
AvailableEncode         : 18446744073709551615
MinPartitionEncode      : 0
MaxPartitionEncode      : 18446744073709551615
OptimalPartitionEncode  : 18446744073709551615
TotalDecode             : 1000000000
AvailableDecode         : 1000000000
MinPartitionDecode      : 0
```

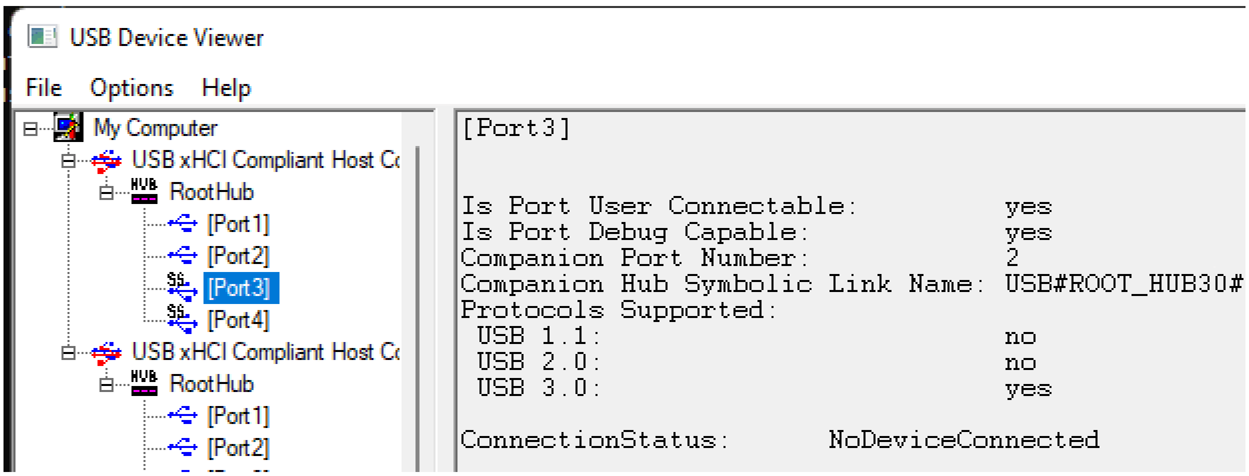
In a past article, we could be able to perform remote debugging using a network connection. I tried to do that, but failed because the physical network adapter didn't support debugging:

```
PS D:\poc> .\kdnet.exe

This Microsoft hypervisor supports using KDNET in guest VMs.
Network debugging is not supported on any of the NICs in this machine.
KDNET supports NICs from Intel, Broadcom, Realtek, Atheros, Emulex, Mellanox
and Cisco.
```

I had to use another approach. Luckily, Windows has several ways to be debugged. In this case, I chose to use USB3 debugging. To do that, I had to:

Find a USB3 port on my debuggee laptop with debugging support. That could be done using **USBView** from Windows SDK:



Enable debug options.

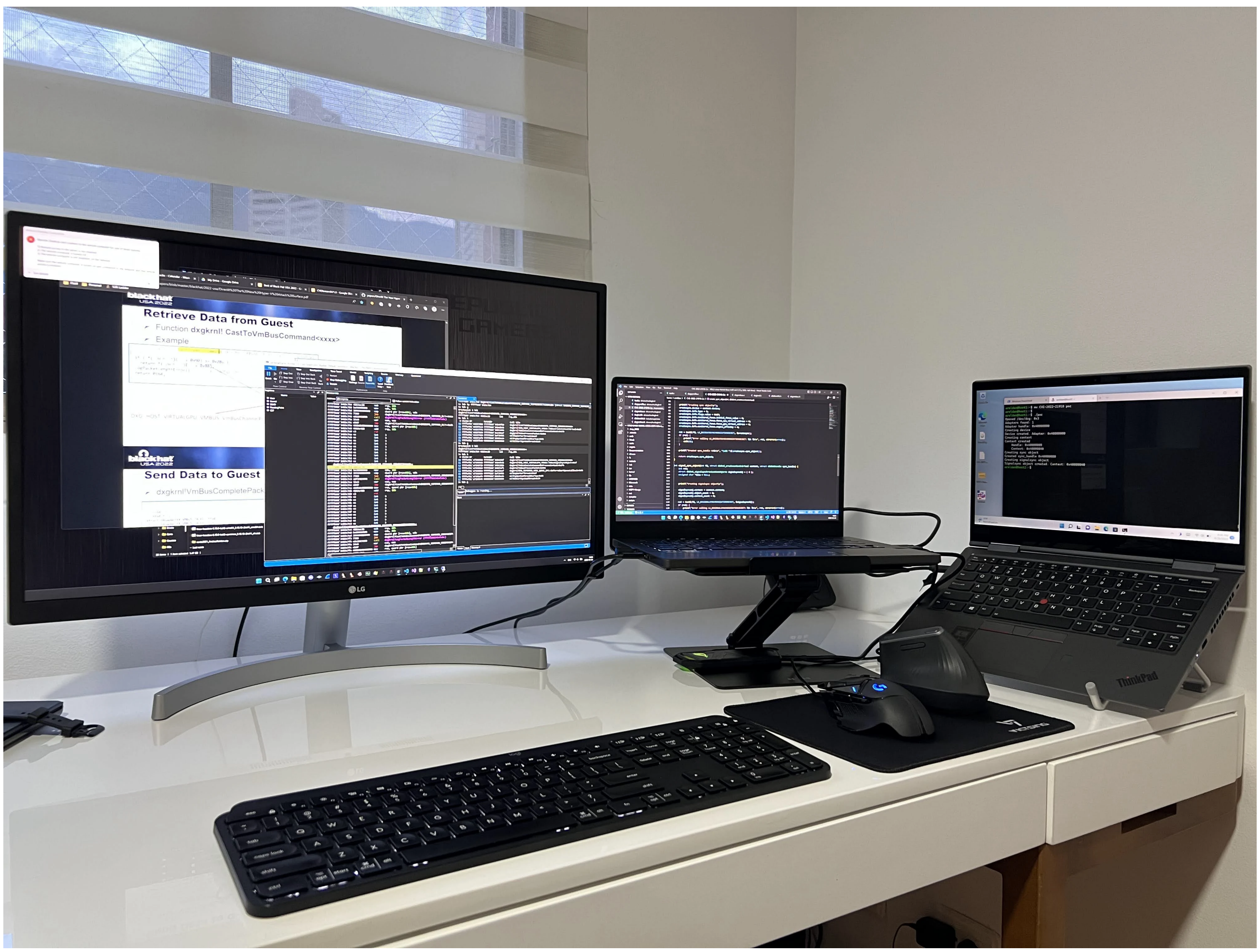
```
PS C:\Users\aroldan> bcdedit /dbgsettings usb targetname:fluidlab
The operation completed successfully.
PS C:\Users\aroldan> bcdedit /debug on
The operation completed successfully.
PS C:\Users\aroldan> bcdedit.exe /set "{dbgsettings}" busparams 45.0.0
The operation completed successfully.
PS C:\Users\aroldan>
```

Plug the debugger and the debuggee using a quality USB3 cable.

```
Using USB for debugging
Waiting to reconnect...
USB: Write opened
Connected to Windows 10 22000 x64 target at (Fri Aug 26 17:14:08.246 2022 (UTC - 5:00)), ptr64 TRUE
Kernel Debugger connection established.

************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*
OK                                             C:\ProgramData\Dbg\sym
Deferred                                       srv*https://msdl.microsoft.com/download/symbols
Deferred                                       srv*C:\ProgramData\Dbg\sym*https://msdl.microsoft.com/download/symbo
Deferred                                       srv*c:\SYMBOLS*https://msdl.microsoft.com/download/symbols
Symbol search path is: srv*;C:\ProgramData\Dbg\sym;srv*https://msdl.microsoft.com/download/symbols;srv*C:\ProgramDa
Executable search path is:
Windows 10 Kernel Version 22000 MP (8 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Edition build lab: 22000.1.amd64fre.co_release.210604-1628
Machine Name:
Kernel base = 0xfffff800`2ac00000 PsLoadedModuleList = 0xfffff800`2b8296b0
Debug session time: Fri Aug 26 17:14:01.816 2022 (UTC - 5:00)
System Uptime: 0 days 0:04:48.169
Break instruction exception - code 80000003 (first chance)
*******************************************************************************
```

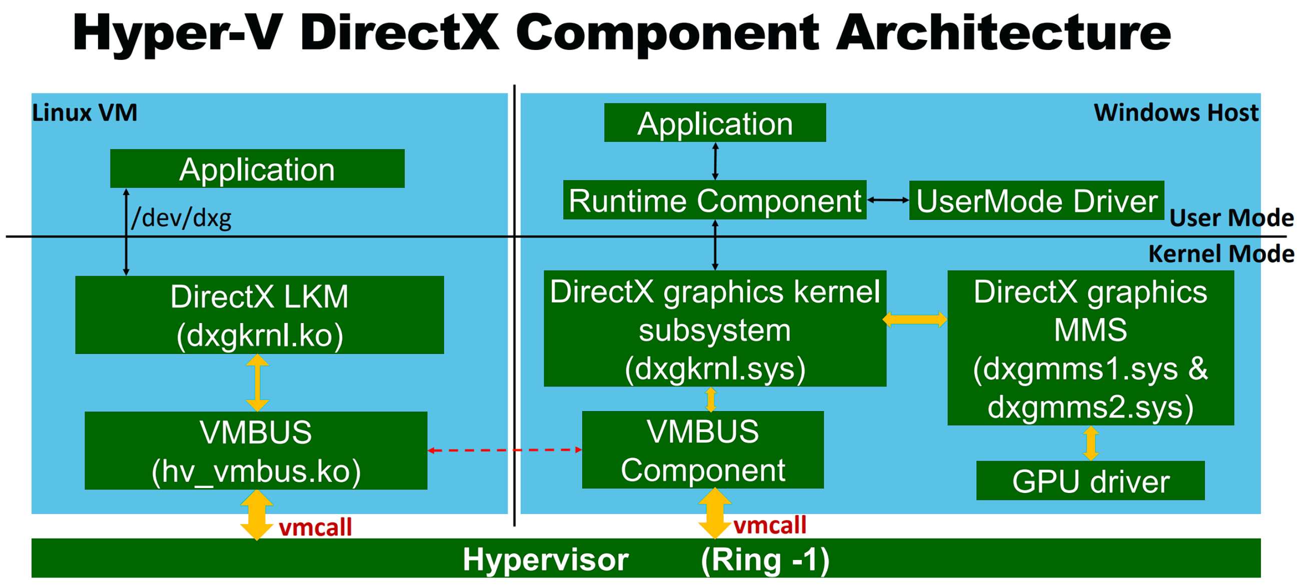
In the end, the lab environment looked like this:



We are ready now!

An updated Hyper-V DirectX data flow

The following graph was presented by Zhenhao Hong (@rthhh17) which nicely describes the DirectX components and how are they accessed by a VM on Hyper-V:



The following is a detailed and updated flow of these interactions:

1. There is a Linux driver called `dxgkrnl.ko` which exposes a set of `IOCTL` commands to interact with the host's DirectX adapters.
2. When a `IOCTL` is called, there is another driver called `hv_vmbus.ko` which uses the VMBUS to create a packet and a bus channel between the VM, the hypervisor and the kernel of the host machine.
3. The `IOCTL` payload is contained in a structure called `DXGADAPTER_VMBUS_PACKET` which contains the command (`DXGK_VMBCOMMAND`) and the command options to be sent.
4. The host machine implements the receiving and processing counterpart in the `dxgkrnl.sys` driver.
5. The procedure `dxgkrnl!VmBusProcessPacket` is the `VMBUS` receiving method that handles the `DXGADAPTER_VMBUS_PACKET` payload.
6. If the `DXGK_VMBCOMMAND` is a global command (listed on `enum dxgkvmb_commandtype_global`), a function pointer (indirect call) is set to a method with the form `dxgkrnl!DXG_HOST_GLOBAL_VMBUS::<command>`, for example `dxgkrnl!DXG_HOST_GLOBAL_VMBUS::VmBusDestroyProcess`. Otherwise, the flow skips to point 7.
7. If the `DXGK_VMBCOMMAND` is not a global command packet, it is processed by `dxgkrnl!VmBusExecuteCommandInProcessContext` which also uses indirect calls (function pointers) to compute the target handling method of that specific `IOCTL` request command. In this case, the handler has the form `dxgkrnl!DXG_HOST_VIRTUALGPU_VMBUS::<command>`, for example `dxgkrnl!DXG_HOST_VIRTUALGPU_VMBUS::VmBusCreateDevice`.

8. The handling method casts the `DXGADAPTER_VMBUS_PACKET` packet using `dxgkrnl!CastToVmBusCommand<DXGKVMB_COMMAND_<command>>` (for example, `dxgkrnl!CastToVmBusCommand<DXGKVMB_COMMAND_DESTROYPROCESS>`) to filter the data as needed to this specific command handler.
9. The handler performs boilerplate checks and perform the desired action. In some cases, it delivers the packet to a function with the pattern `dxgkrnl!*Internal` (for example, `dxgkrnl!SignalSynchronizationObjectInternal`) or `dxgkrnl!Dxgk<command>Impl` (for example, `dxgkrnl!DxgkCreateDeviceImpl`) which has the required interfaces to deliver the packet to the MMS (Microsoft Media System) components of DirectX that resides on the `dxgmms1.sys` and `dxgmms2.sys` drivers.
10. The MMS system is finally in charge to talk with the corresponding GPU driver, which exposes the adapter that can either be virtual or physical.
11. In the end, the response is sent back to the VM via `dxgkrnl!VmBusCompletePacket`.

It's a complex process if you read it, but let's look at it in action. Let's see an example performing only one command: `Create Device`. Here is the sample code.

Get started with Fluid Attacks' Red Teaming solution right now

---

```
/*
Hyper-V -> DirectX Interaction Sample Code

Compile as: cc -ggdb -Og -o sample1 sample1.c

Author: Andres Roldan <aroldan@fluidattacks.com>
LinkedIn: https://www.linkedin.com/in/andres-roldan/
Twitter: @andresroldan
*/

#define _GNU_SOURCE 1
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include "/home/aroldan/WSL2-Linux-Kernel-linux-msft-wsl-
5.15.y/include/uapi/misc/d3dkmthk.h"

int open_device() {
    int fd;
    fd = open("/dev/dxg", O_RDWR);
    if (fd < 0) {
        printf("Cannot open device file...\n");
        exit(1);
    }
    printf("Opened /dev/dxg: 0x%x\n", fd);
    return fd;
}

void create_device(int fd) {
    int ret;
    struct d3dkmt_createdevice ddd = { 0 };
    struct d3dkmt_adapterinfo adapterinfo = { 0 };
    struct d3dkmt_enumadapters3 enumada = { 0 };

    enumada.adapter_count = 0xff;
    enumada.adapters = &adapterinfo;
    ret = ioctl(fd, LX_DXENUMADAPTERS3, &enumada);
    if (ret) {
        printf("Error calling LX_DXENUMADAPTERS3: %d: %s\n", ret, strerror(errno));
        exit(1);
    }
    printf("Adapters found: %d\n", enumada.adapter_count);

    ddd.adapter = adapterinfo.adapter_handle;
    printf("Adapter handle: 0x%x\n", ddd.adapter.v);
```

```
    printf("Creating device\n");
    ret = ioctl(fd, LX_DXCREATEDEVICE, &ddd);
    if (ret) {
        printf("Error calling LX_DXCREATEDEVICE: %d: %s\n", ret, strerror(errno));
        exit(1);
    }
    printf("Device created: 0x%x\n", ddd.device);
}

int main() {
    int fd;
    struct d3dkmthandle device;

    fd = open_device();
    create_device(fd);
    close(fd);
}
```

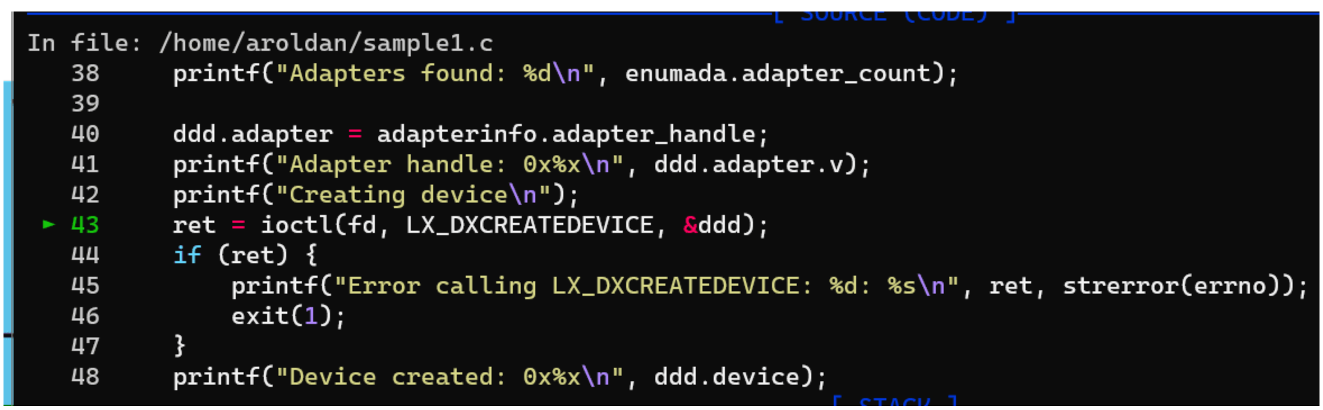
It's a straightforward code:

1. Opens a handle to `/dev/dxg`.
2. Uses that handle to enumerate the adapters available.
3. Creates the device handle.

The output on the console should be something like:

```
aroldan@host1:~$ cc -ggdb -Og -o sample1 sample1.c
aroldan@host1:~$ ./sample1
Opened /dev/dxg: 0x3
Adapters found: 2
Adapter handle: 0x40000000
Creating device
Device created: 0x40000000
```

Let's first look at the Linux side of things. First, I'm going to pause the execution on `gdb` at `*create_device+176` (`sample1.c:43`) which is when the `IOCTL` calling the command `LX_DXCREATEDEVICE` is performed:



```
In file: /home/aroldan/sample1.c
   38     printf("Adapters found: %d\n", enumada.adapter_count);
   39
   40     ddd.adapter = adapterinfo.adapter_handle;
   41     printf("Adapter handle: 0x%x\n", ddd.adapter.v);
   42     printf("Creating device\n");
 ► 43     ret = ioctl(fd, LX_DXCREATEDEVICE, &ddd);
   44     if (ret) {
   45         printf("Error calling LX_DXCREATEDEVICE: %d: %s\n", ret, strerror(errno));
   46         exit(1);
   47     }
   48     printf("Device created: 0x%x\n", ddd.device);
```

If we see the value of the variable `ddd.device` before the call, you should see something like this:

```
pwndbg> print /x ddd.device
$1 = {
  {
    {
      instance = 0x0,
      index = 0x0,
      unique = 0x0
    },
    v = 0x0
  }
}
pwndbg>
```

After the `IOCTL`, we can see that the device handle is now populated:

```
pwndbg> ni
0x00005555555552a8     43        ret = ioctl(fd, LX_DXCREATEDEVICE, &ddd);
```

```
pwndbg> print /x ddd.device
$2 = {
  {
    {
      instance = 0x0,
      index = 0x0,
      unique = 0x1
    },
    v = 0x40000000
  }
}
```

Now, let's check at the host running Windows. We should be able to witness the creation of the device handler (`0x40000000`) on a `dxgkrnl!VmBusCompletePacket` response. We're going to need to set a few breakpoints to check the flow. First, let's put a breakpoint at `dxgkrnl!VmBusProcessPacket`

```
0: kd> bp dxgkrnl!VmBusProcessPacket
0: kd> g
```

Inspecting `dxgkrnl!VmBusProcessPacket` we can see at `dxgkrnl!VmBusProcessPacket+0x568` an indirect call being performed. This is where `dxgkrnl!VmBusProcessPacket` handles `DXGK_VMBCOMMAND` global commands. You can find indirect calls (function pointers) in kernel space because they are wrapped by calls to `_guard_dispatch_icall_fptr`, which is added when the kernel is compiled with CFG.

Let's put another breakpoint there:

```
3: kd> u dxgkrnl!VmBusProcessPacket+0x568 L1
dxgkrnl!VmBusProcessPacket+0x568:
fffff804`47a53338 ff1532deddff    call    qword ptr [dxgkrnl!_guard_dispatch_icall_fptr
3: kd> bp dxgkrnl!VmBusProcessPacket+0x568
```

Now, let's put a breakpoint at `dxgkrnl!VmBusExecuteCommandInProcessContext`:

```
3: kd> bp dxgkrnl!VmBusExecuteCommandInProcessContext
```

In that function at `dxgkrnl!VmBusExecuteCommandInProcessContext+0x1f0` we can also find an indirect call being performed. We can set a new breakpoint in that place:

```
3: kd> u dxgkrnl!VmBusExecuteCommandInProcessContext+0x1f0 L1
dxgkrnl!VmBusExecuteCommandInProcessContext+0x1f0:
fffff800`6a62dc5c ff150e35deff    call    qword ptr [dxgkrnl!_guard_dispatch_icall_fptr
3: kd> bp dxgkrnl!VmBusExecuteCommandInProcessContext+0x1f0
```

Finally, a breakpoint at `dxgkrnl!VmBusCompletePacket` will be set:

```
4: kd> bp dxgkrnl!VmBusCompletePacket
```

We should now have five breakpoints as follows:

```
2: kd> bl
     0 e Disable Clear  fffff800`6a632dd0     0001 (0001) dxgkrnl!VmBusProcessPacket
     1 e Disable Clear  fffff800`6a633338     0001 (0001) dxgkrnl!VmBusProcessPacket+0x568
     2 e Disable Clear  fffff800`6a62da6c     0001 (0001) dxgkrnl!VmBusExecuteCommandInProcessContext
     3 e Disable Clear  fffff800`6a62dc5c     0001 (0001) dxgkrnl!VmBusExecuteCommandInProcessContext+0x1f0
     4 e Disable Clear  fffff800`6a31e278     0001 (0001) dxgkrnl!VmBusCompletePacket
```

I'm going to reference the steps described above in the following execution flow.

When we run the sample code again, it hits our first breakpoint (step 5):

```
0: kd> g
Breakpoint 0 hit
dxgkrnl!VmBusProcessPacket:
fffff800`6a632dd0 4053            push    rbx
```

When we resume the execution, the next breakpoint is hit at the `dxgkrnl!_guard_dispatch_icall_fptr` call, which is an indirect call to the first command's handler. In this case, the handling function was resolved as `dxgkrnl!DXG_HOST_GLOBAL_VMBUS::VmBusCreateProcess` (step 6):

```
     0 e Disable Clear  fffff800`6a632dd0     0001 (0001) dxgkrnl!VmBusProcessPacket
     1 e Disable Clear  fffff800`6a633338     0001 (0001) dxgkrnl!VmBusProcessPacket+0x
     2 e Disable Clear  fffff800`6a62da6c     0001 (0001) dxgkrnl!VmBusExecuteCommandIn
     3 e Disable Clear  fffff800`6a62dc5c     0001 (0001) dxgkrnl!VmBusExecuteCommandIn
     4 e Disable Clear  fffff800`6a31e278     0001 (0001) dxgkrnl!VmBusCompletePacket

4: kd> g
Breakpoint 0 hit
dxgkrnl!VmBusProcessPacket:
fffff800`6a632dd0 4053            push    rbx

0: kd>
```

If we resume the execution, a call to `dxgkrnl!VmBusCompletePacket` is performed to send to the caller the result of the `dxgkrnl!DXG_HOST_GLOBAL_VMBUS::VmBusCreateProcess` command:

```
0: kd> t
dxgkrnl!DXG_HOST_GLOBAL_VMBUS::VmBusCreateProcess:
fffff800`6a62af90 488bc4          mov     rax,rsp
0: kd> g
Breakpoint 4 hit
dxgkrnl!VmBusCompletePacket:
fffff800`6a31e278 4883ec28        sub     rsp,28h
```

When we resume the execution twice, first the breakpoint at `dxgkrnl!VmBusProcessPacket` is hit (step 5) as expected, but the next breakpoint hit is at `dxgkrnl!VmBusExecuteCommandInProcessContext`, which means that the incoming command is not a global command (step 7):

```
0: kd> g
Breakpoint 0 hit
dxgkrnl!VmBusProcessPacket:
fffff800`6a632dd0 4053            push    rbx
2: kd> g
Breakpoint 2 hit
dxgkrnl!VmBusExecuteCommandInProcessContext:
fffff800`6a62da6c 4c8bdc          mov     r11,rsp
```

Now, when we resume the execution, the next breakpoint is hit at `dxgkrnl!VmBusExecuteCommandInProcessContext+0x1f0` which contains the indirect call resolved to a non-global command. In this case, we see the command we sent (`LX_DXCREATEDEVICE`) for creating a device:

```
dxgkrnl!VmBusCompletePacket:
fffff800`6a31e278 4883ec28        sub     rsp,28h
0: kd> g
Breakpoint 0 hit
dxgkrnl!VmBusProcessPacket:
fffff800`6a632dd0 4053            push    rbx
2: kd> g
Breakpoint 2 hit
dxgkrnl!VmBusExecuteCommandInProcessContext:
fffff800`6a62da6c 4c8bdc          mov     r11,rsp

2: kd>
```

In that method, at `dxgkrnl!DXG_HOST_VIRTUALGPU_VMBUS::VmBusCreateDevice+0x8d` we can see a call to `dxgkrnl!CastToVmBusCommand<DXGKVMB_COMMAND_CREATEDEVICE>` which will extract the needed parts of the `DXGADAPTER_VMBUS_PACKET` (step 8):

```
2: kd> u dxgkrnl!DXG_HOST_VIRTUALGPU_VMBUS::VmBusCreateDevice+0x8d
dxgkrnl!DXG_HOST_VIRTUALGPU_VMBUS::VmBusCreateDevice+0x8d:
fffff800`6a62a29d e82228cfff      call    dxgkrnl!CastToVmBusCommand<DXGKVMB_COMMAND_CREATEDEVICE>
```

Here's the decompiled code:

```
__int64 __fastcall CastToVmBusCommand<DXGKVMB_COMMAND_CREATEDEVICE>(__int64 a1)
{
  if ( *(_DWORD *)(a1 + 144) >= 0x28u )
    return *(_QWORD *)(a1 + 136);
  LogPacketLengthError((struct DXGADAPTER_VMBUS_PACKET *)a1, 0x28ui64);
  return 0i64;
}
```

Later on that function, at `dxgkrnl!DXG_HOST_VIRTUALGPU_VMBUS::VmBusCreateDevice+0x3b0`, we can see a call to `dxgkrnl!DxgkCreateDeviceImpl` which do the dirty job (step 9):

```
4: kd> u dxgkrnl!DXG_HOST_VIRTUALGPU_VMBUS::VmBusCreateDevice+0x3b0 L1
dxgkrnl!DXG_HOST_VIRTUALGPU_VMBUS::VmBusCreateDevice+0x3b0:
fffff804`47a4a5c0 e8b328e5ff      call    dxgkrnl!DxgkCreateDeviceImpl
```

And finally, when we continue the execution, the breakpoint at `dxgkrnl!VmBusCompletePacket` is hit. According to this article, the second parameter of the function `dxgkrnl!VmBusCompletePacket` is the data to be sent back to the caller (step 11). It means that if we check the double word data pointed by the `rdx` register, we should see the device handler (`0x40000000`) returned as we saw before in the Linux VM output:

```
2: kd> t
dxgkrnl!DXG_HOST_VIRTUALGPU_VMBUS::VmBusCreateDevice:
fffff800`6a62a210 48895c2410      mov     qword ptr [rsp+10h],rbx
2: kd> g
Breakpoint 4 hit
dxgkrnl!VmBusCompletePacket:
fffff800`6a31e278 4883ec28        sub     rsp,28h
2: kd> dds rdx L1
fffffe05`59645630  40000000
```

Great!

You can download the `sample1.c` file here.

Conclusion

The Hyper-V DirectX interaction is not officially documented. You can understand most of the internals by reading the WSL code, performing reverse engineering of the Windows drivers and doing kernel debugging. In the next article, we will see that most of the `dxgkrnl` commands are not stateless and some of them depends on creating certain kernel objects first. We will also see how to leverage this architecture using an offensive approach.

If you want to follow this series and receive our next blog posts, don't hesitate to subscribe to our weekly newsletter.