



A journey of fuzzing Nvidia graphic driver leading to LPE exploitation

Quarkslab

/HEXACON/

14th & 15th of October 2022



Motivation

- Two fuzzing projects released in 2021: **WTF** & **Rewind**
- Both offer to easily target kernel components
- Wanted to get familiar with both of them
- Needed a target
 - Tried various ~~victims~~ candidates
 - Decided to go for the graphical driver developed by Nvidia

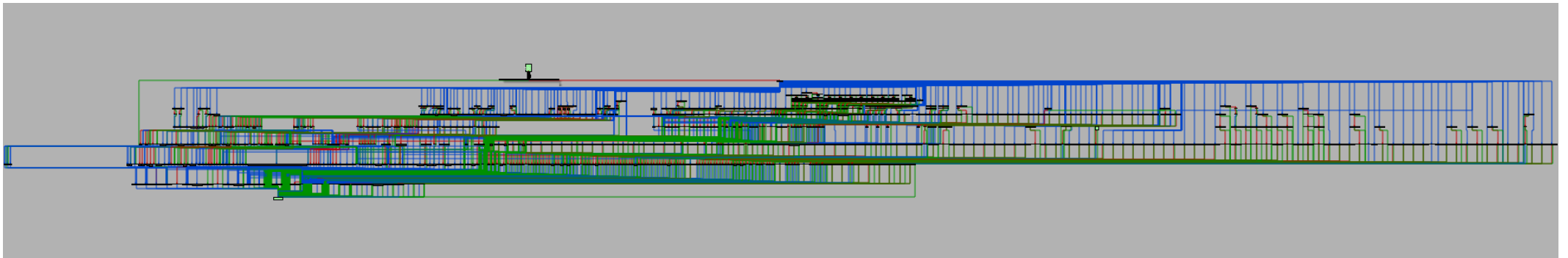


Why Nvidia Graphic Driver?

- Simple entry point...

```
NTSTATUS DxgkddiEscape(  
    IN_CONST_HANDLE hAdapter,  
    IN_CONST_PDXGKARG_ESCAPE pEscape  
)
```

- With an interesting attack surface





Previous Works

- [Attacking the Windows NVIDIA Driver](#)
Blogpost 2017, Project Zero
- [Evolutionary Kernel Fuzzing](#)
BlackHat 2017, Richard Johnson
- [Direct X – Direct way to Microsoft Windows Kernel](#)
Zeronights 2011, Nikita Tarakanov



Snapshot Fuzzing

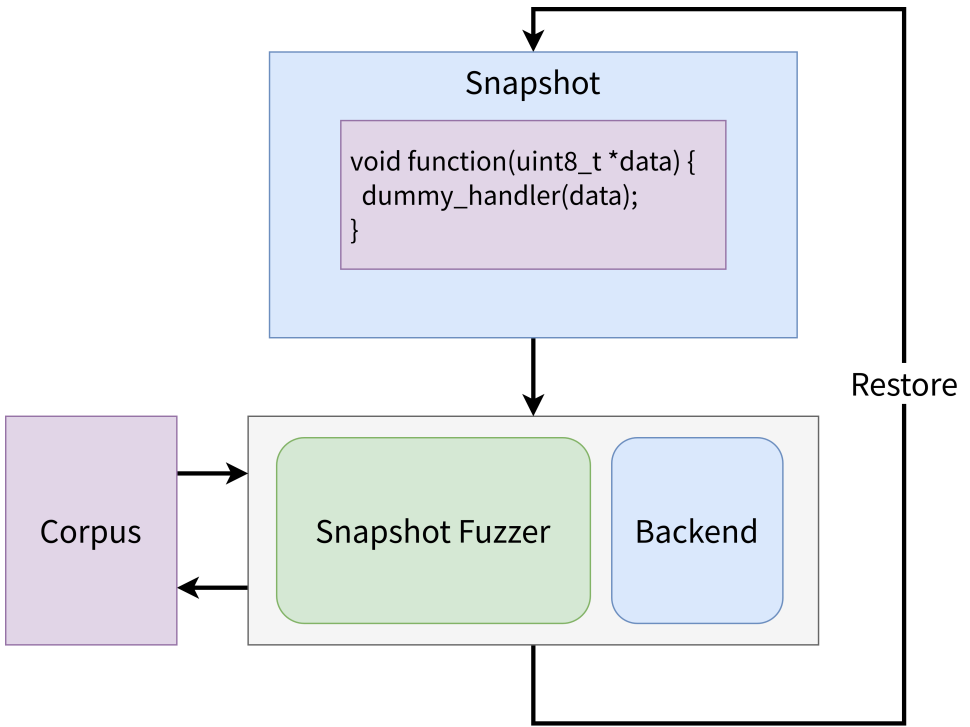
WTF

- By Overcl0k
- <https://github.com/Overcl0k/wtf>

Rewind

- By Erynian
- <https://github.com/quarkslab/rewind>

- Both use Hyper-V, BochsCPU and KVM backend





DxgkDdiEscape Interface

```
NTSTATUS DxgkddiEscape(  
    IN_CONST_HANDLE hAdapter,  
    IN_CONST_PDXGKARG_ESCAPE pEscape  
)
```

- Entry point arguments
 - hAdapter: adapter handle
 - pEscape: documented structure
 - Contains the message sent to the interface (pPrivateDriverData)
 - The format is constructor specific!



DxgkDdiEscape Escape Structure

```
typedef struct _DXGKARG_ESCAPE {
    [in] HANDLE hDevice;
    [in] D3DDDI_ESCAPEFLAGS Flags;
    [in/out] VOID *pPrivateDriverData;
    [in/out] UINT PrivateDriverDataSize;
    [in] HANDLE hContext;
    HANDLE hKmdProcessHandle;
} DXGKARG_ESCAPE;
```

- The handles are optional except for hDevice
- The command message is constructor dependant



First Fuzzing Iteration

Corpus Generation

- Record the command messages sent to the graphic driver
- Generate activities using a benchmarking tool

Result

- Barely 40% of the driver handlers covered

We have to build a better corpus



Corpus Generation



Private Buffer Format

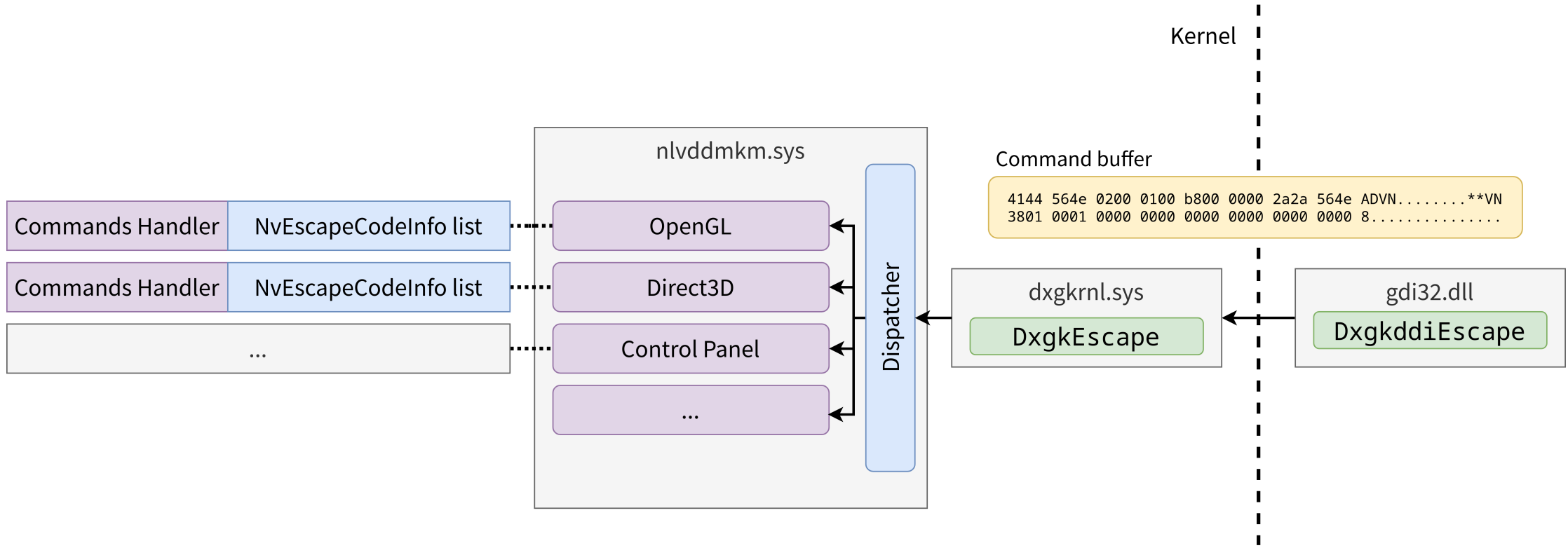
- Starts with a generic header
- Followed by the actual content
 - Specific format for each functionality

```
// sizeof(NvPrivateDataHeader) == 0x10
struct NvPrivateDataHeader {
    DWORD magic_tag;
    WORD major_version;
    WORD minor_version;
    DWORD private_data_size;
    DWORD caller_tag;
}

struct NvPrivateData {
    UINT EscapeCode;
    ...
}
```



Driver Architecture





Generic Attribute Validation

```
typedef struct _NvEscapeCodeInfo {
    UINT EscapeCode;
    UINT Size;
    BYTE Unk_2[0x8];
    UINT AdminPrivRequired;
    UINT Flags_1;
    ...
    PVOID ValidationFunction;
} NvEscapeCodeInfo;
```

- The `Flag_1` value gives information about the handle(s) to provide
 - `0x1`: A device handle is required
 - `0x2`: Device and context handles are required



Specific Message Callbacks

- Callbacks may give interesting information about the format

```
bool validation_function_1000151(DXGKARG_ESCAPE Escape, /* ... */) {
    PrivateData1000151 *msg = Escape->pPrivateDriverData;
    if (RtlCompareMemory(msg->guid_1, GUID_E7A07B48, sizeof(GUID)) ||
        RtlCompareMemory(msg->guid_2, GUID_7F03FC51, sizeof(GUID)) ||
        RtlCompareMemory(msg->guid_3, GUID_C50F93EF, sizeof(GUID))) {
        return true;
    }
    return false;
}
```



Inputs Generation

- IDA scripting
 - Parsing NvEscapeCodeInfo structures to generate message header

```
00000000: 0000 0000 0000 0000 4144 564e 0200 0100 .....ADVN....  
00000010: 4001 0000 2a2a 564e 0300 0001 0000 0000 @...**VN.....  
00000020: 0000 0000 0000 0000 0000 0000 0000 0000 .....  
00000030: 0000 0000 0000 0000 0000 0000 0000 0000 .....
```



Inputs Generation

- IDA scripting
 - Parsing NvEscapeCodeInfo structures to generate message header
- Dynamic Symbolic Execution
 - Generate inputs that pass the validation

```
00000000: 0000 0000 0000 0000 4144 564e 0200 0100 .....ADVN....  
00000010: 4001 0000 2a2a 564e 0300 0001 0000 0000 @...**VN.....  
00000020: 0000 0000 0000 0000 0000 0000 0000 0000 .....  
00000030: 0000 0000 0000 0000 0000 0000 0000 0000 .....
```

- **Triton** - <https://github.com/JonathanSalwan/Triton>
 - Dynamic binary analysis library with Python bindings
 - Allows to easily cover all the edges of a function



Inputs Generation

- Most of the time, only the first argument is used
- Both `DXGKARG_ESCAPE.Flags` and `DXGKARG_ESCAPE.PrivateDataBuffer` are symbolized
- Cover all the edges of the function
 - For each jump instruction, we look for a symbolized value that inverts it

```
{'isTaken': False,  
  'srcAddr': 5382793397, 'dstAddr': 5382793426,  
  'constraint': (((~(ref_724) & 0x1) & (~(ref_728) & 0x1)) == 0x1)}
```

```
New inputs: {52: SymVar_52:8 = 0x0, 53: SymVar_53:8 = 0x0, 55: SymVar_55:8 = 0x4,  
             54: SymVar_54:8 = 0x0}
```



Inputs Generation

- At the end of the function
 - Try to resolve the symbolic value of rax with it equal to TRUE as a constraint

```

rax = (((((((0x1 == 0x1) and not (((~(ref_724) & 0x1) & (~(ref_728) & 0x1)) == 0x1))
      and not (((~(ref_733) & 0x1) & (~(ref_737) & 0x1)) == 0x1))
      and not (((~(ref_742) & 0x1) & (~(ref_746) & 0x1)) == 0x1))
      and not (((~(ref_751) & 0x1) & (~(ref_755) & 0x1)) == 0x1))
      and not (ref_760 == 0x0)) and (0x1 == 0x1))

```

Valid escape buffer:

```

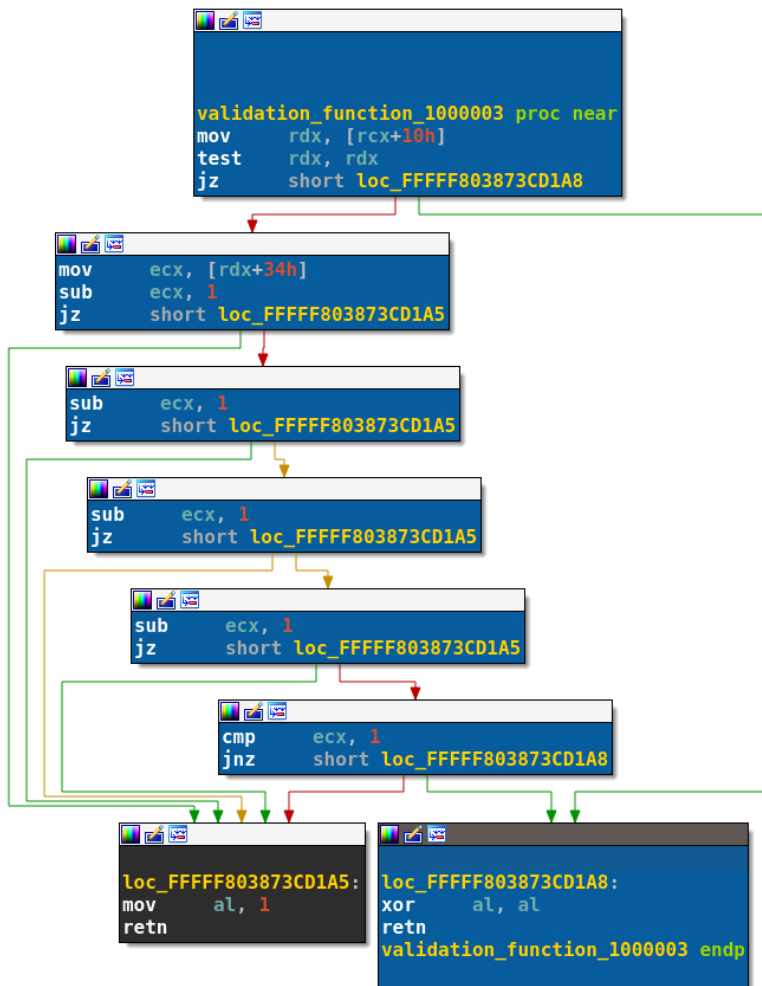
...
00000040: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 .....
00000050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
...

```



- Input generated by the IDA script

00000000:	0000	0000	0000	0000	4144	564e	0200	0100ADVN....
00000010:	4001	0000	2a2a	564e	0300	0001	0000	0000	@...**VN.....
00000020:	0000	0000	0000	0000	0000	0000	0000	0000
00000030:	0000	0000	0000	0000	0000	0000	0000	0000



- Input generated by the IDA script

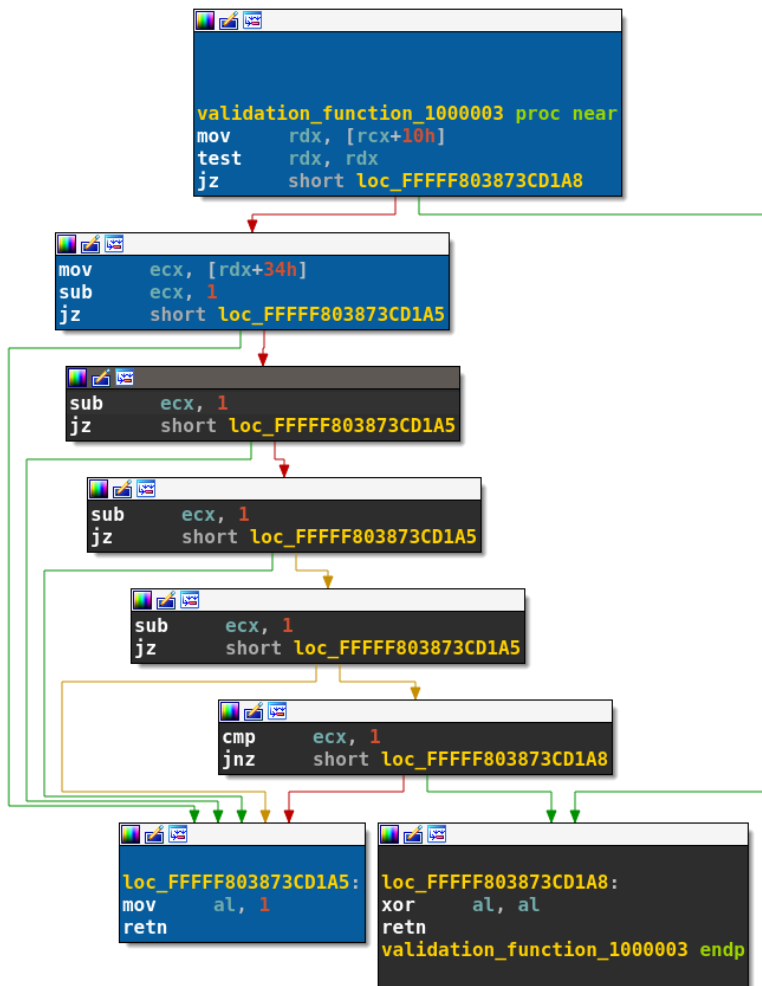
```

00000000: 0000 0000 0000 0000 4144 564e 0200 0100 .....ADVN....
00000010: 4001 0000 2a2a 564e 0300 0001 0000 0000 @...**VN.....
00000020: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000030: 0000 0000 0000 0000 0000 0000 0000 0000 .....

```



- First valid file generated with DSE

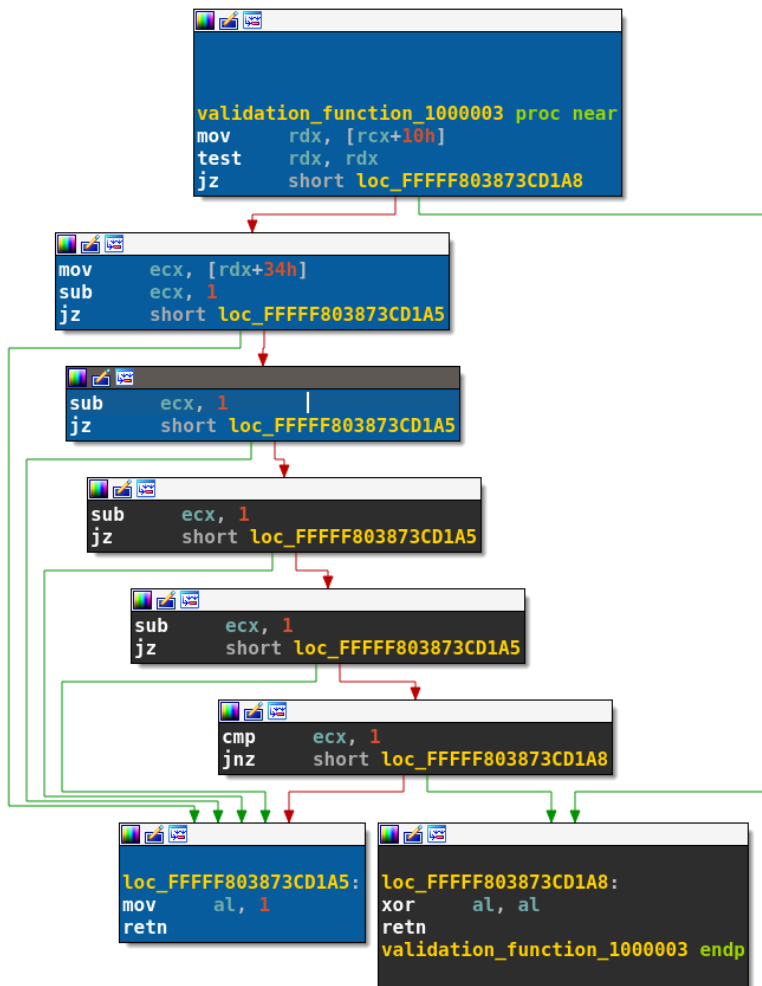


```

00000000: 0000 0000 0000 0000 4144 564e 0200 0100 .....ADVN....
00000010: 4001 0000 2a2a 564e 0300 0001 0000 0000 @...**VN.....
00000020: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000030: 0000 0000 0000 0000 0000 0000 0100 0000 .....
  
```



- Second valid file generated with DSE



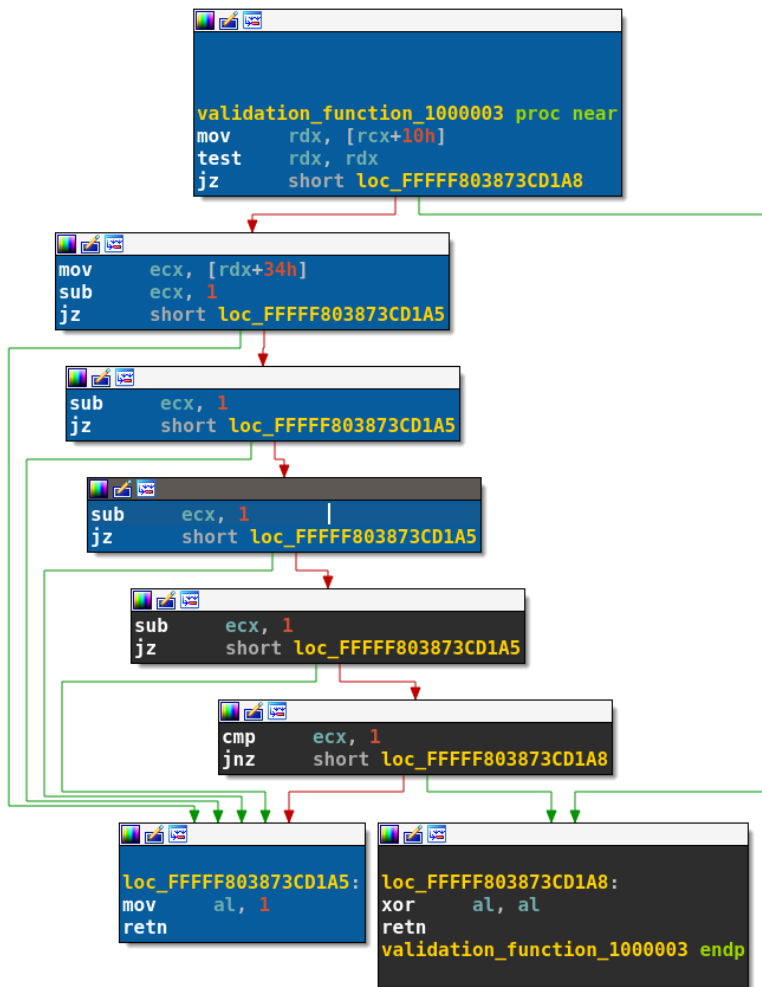
```

00000000: 0000 0000 0000 0000 4144 564e 0200 0100 .....ADVN....
00000010: 4001 0000 2a2a 564e 0300 0001 0000 0000 @...**VN.....
00000020: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000030: 0000 0000 0000 0000 0000 0000 0200 0000 .....

```



- Third valid file generated with DSE



00000000:	0000	0000	0000	0000	4144	564e	0200	0100ADVN....
00000010:	4001	0000	2a2a	564e	0300	0001	0000	0000	@...**VN.....
00000020:	0000	0000	0000	0000	0000	0000	0000	0000
00000030:	0000	0000	0000	0000	0000	0000	0300	0000



New Corpus and Coverage

- Coverage with the new corpus: 40% -> 80%

Limitation

- Some callbacks access objects in memory and cannot be emulated
 - Need a way to link the script with the memory dump



Fuzzing Harness

- Quite simple harness
- Implementation of 2 functions:
 - `Init`: setup stop addresses
 - `InsertTestcase`: called at every iteration
 - Allows to set up the test case files
- Discards invalid buffers
 - When important data (MagicCode, size, etc.) is corrupted

We remove all the mutation strategies in WTF that could impact the buffer size



Fuzzing Results



Identified Bugs

Command ID	Bug	Description
0x1000083	Out of bounds write	Out of bound write in the Adapter object leading to a privilege escalation vulnerability
0x100006b	Out of bounds read	Out of bound read in the data section
0x100002f	Out of bounds read	Out of bound read bug. <i>Not exploitable.</i>
0x7000013		
0x700010b		



0x100006b: Out of Bounds Read

- Offset read without any check and used in a memory copy

```
uint32_t offset = EscapeBuffer->Offset;
data_size = 0x2FC - offset;
if (data_size > 0x1DC) {
    data_size = 0x1DC;
}
src_ptr = DataSectionArray + offset;
memcpy(EscapeBuffer->OutputData, src_ptr, data_size);
```

- Allows to copy 476 bytes after an array stored in the .data section
- Function pointer and stack cookie present in this section
 - Can help to bypass KASLR or to exploit a stack buffer overflow (if any)



0x1000083: Out of Bounds Write

- Access to an array located in the Adapter object with an untrusted index
 - 32-bit value retrieved from the private escape buffer

```
uint8_t oob_write(void *p_adapter, /*...*/, uint32_t val_3, uint32_t index) {  
    // ...  
    if(val_3 & 0x3000) {  
        pAdapter->UnkByte_1 = val_1;  
        pAdapter->UnkByte_2 = val_2;  
        pAdapter->ArrayOffset4E94[index] = val_3;  
    }  
    // ...  
}
```



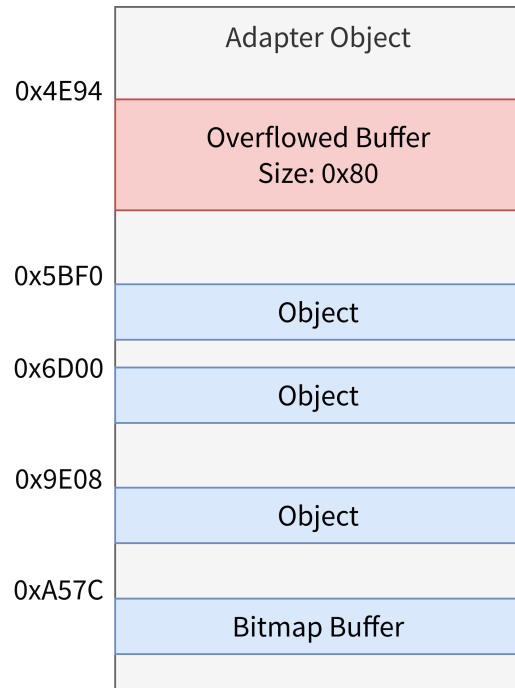
0x1000083: Limitations

- Allows to write a partially controlled value in the 4GB of memory after the array stored in the Adapter object
- Presence of an annoying cache
 - Registration of the Userland process PID
 - Vulnerable code skipped if the same process calls the escape feature twice
 - No simple way to remove the PID from the cache without administrative rights
- Create a new process each time we want to corrupt
- Some limitation on the corrupted value
 - Setting the 12th and 13th bits changes the execution path



Adapter Object Layout

```
> !pool ffff9d0f220eb000  
ffff9d0f220eb000 : large page allocation, tag is NvDI, size is 0x14000 bytes
```



- Several objects can be corrupted
- Gaining a R/W primitive is possible
 - But no CFG protection
 - Corrupting pointers is easier
- Corruption of 32-bit at a time
 - Need to trigger the bug twice
 - Choose something not heavily used by the driver



Tracing Memory Access

- Leverage the fuzzer corpus to follow memory accesses and find our target
- Bochscpu allows to easily add callbacks on the execution
 - BOCHSCPU_HOOK_MEM_EXECUTE
 - BOCHSCPU_HOOK_MEM_READ
 - BOCHSCPU_HOOK_MEM_RW
 - BOCHSCPU_HOOK_MEM_WRITE
- bochscpu_backend.cc modification to trace memory accesses



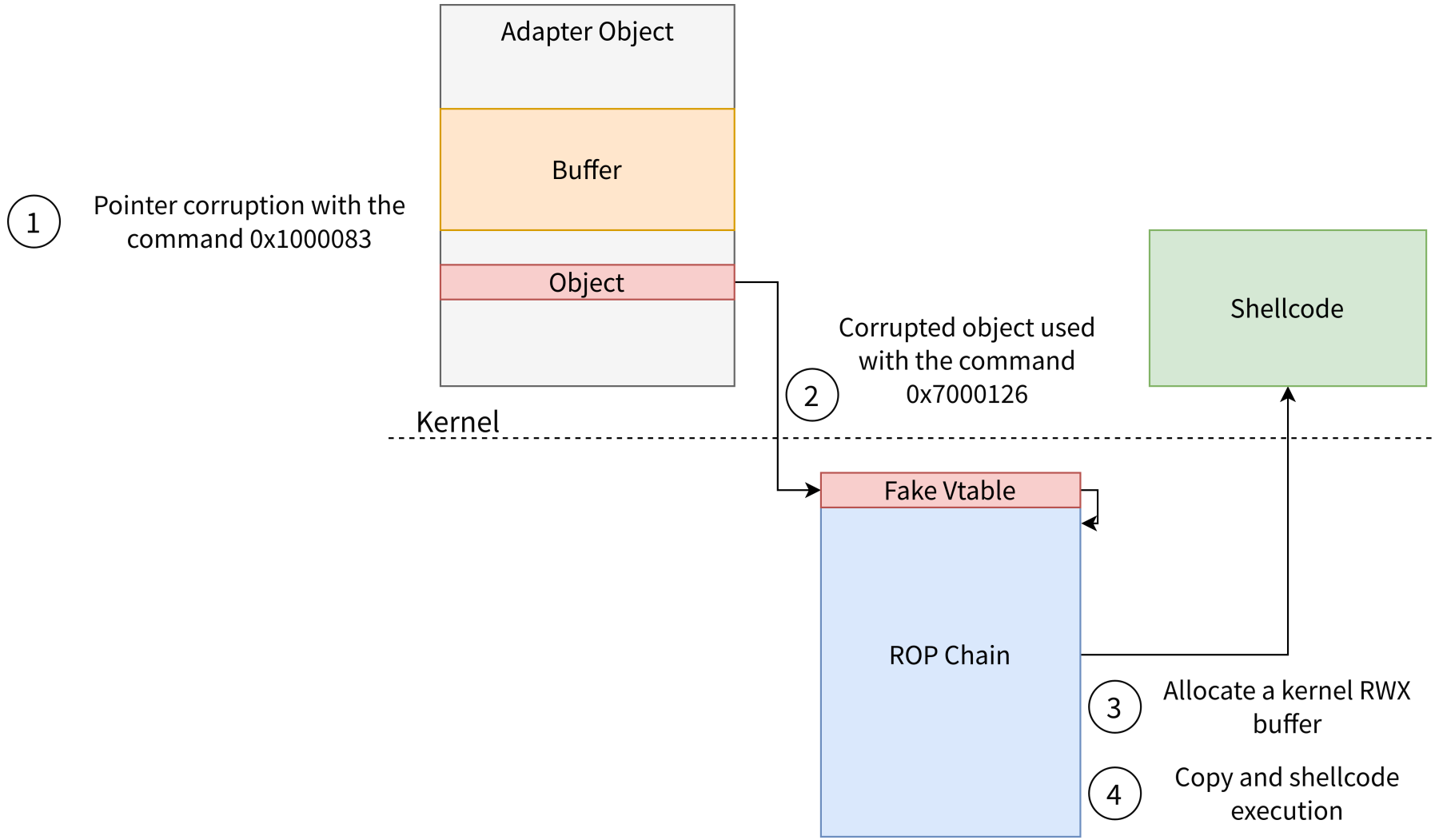
Finding a Pointer to Corrupt

- Looking for a specific pattern:
 - 8-byte read access in the Adapter object memory that can be controlled
 - Followed by another access to the value that has been read previously

```
Read 0xffff8083c4535000 at offset: 0x9e08 (pc: 0xffff803866678bf)  
Access 0xffff8083c4535000 at 0xffff80387174019
```

```
Read 0xffff8083c5d30fe0 at offset: 0x5bf0 (pc: 0xffff80387173f5c)  
Access 0xffff8083c5d30fe0 at 0xffff80387173f63
```

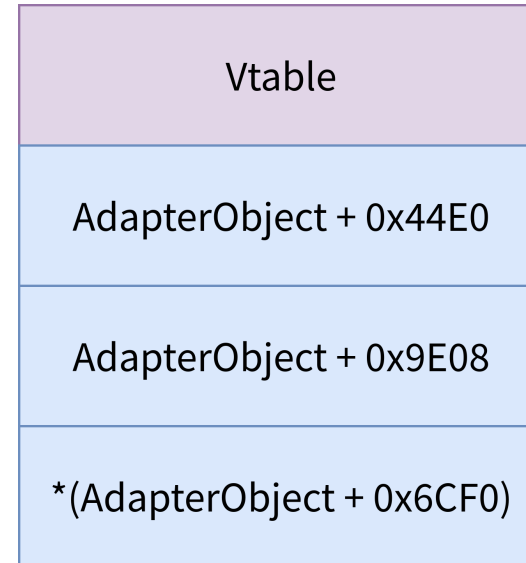
```
Read 0xffff8083c2022000 at offset: 0x6cf0 (pc: 0xffff803871009d0)  
Access 0xffff8083c2022000 at 0xffff803873a852d
```





Restoring the Object

- Need to restore the corrupted object
 - Cannot leak the pointer before overwriting it :(
 - Need to reconstruct it
- Vtable pointer retrievable in an easy way
 - Driver base address is known
- Other fields require a leak of the Adapter object address





Transforming the Corruption into a Memory Leak

- Some Nvidia requests return data to the user
 - Leverage the Adapter object corruption to leak memory
- Bypass the validation by modifying the object before the data is copied in the output buffer
- Reuse of the memory tracing capability offered by Bochscpu backend
 - Record of every read access to the object followed by a copy in the output buffer



Transforming the Corruption into a Memory Leak

We are looking for this kind of pattern:

- Read memory access from the escape buffer
 - we can control something
- Read memory in the controlled part of the Adapter object
- Write memory access in the escape buffer
 - something is returned to the user

```
Adapter Read 0x00000004 to GVA 0xffff8083c20575d0 (Offset: 0x135d0)
```

```
PrivateBuffer Read 0x00000002 to GVA 0xffffd70b83110030 at 0xfffff803870fa488
```

```
PrivateBuffer Read 0x00000002 to GVA 0xffffd70b83110030 at 0xfffff803870fa490
```

```
Adapter Read 0x000000009ac31d0a to GVA 0xffff8083c204c9b0 (Offset: 0x89b0)
```

```
PrivateBuffer Write 8 bytes to GVA 0xffffd70b83110034 at 0xfffff803870fa4b5
```



Transforming the Corruption into a Memory Leak

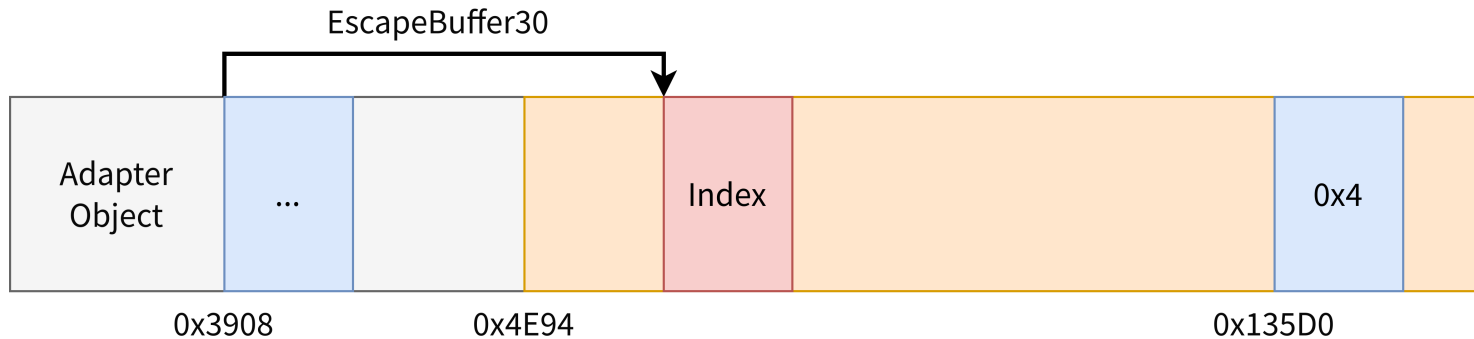
```
// Private Escape Code 0x2000041
uint32_t EscapeBuffer30 = EscapeBuffer->index;
if (EscapeBuffer30 <= AdapterObject->Offset135D0MaxIndex) {
    uint32_t index = AdapterObject->Offset3908Array[EscapeBuffer30];
    uint32_t value = AdapterObject->Offset89B0Array[index * 0x18];
    // ...
    EscapeBuffer->OutputValue = value;
    // ...
}
```





Transforming the Corruption into a Memory Leak

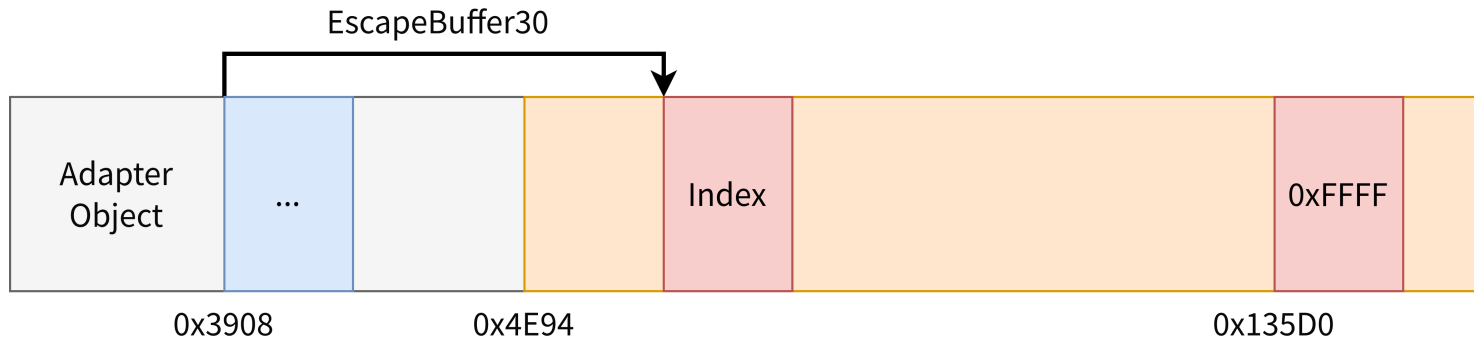
```
// Private Escape Code 0x2000041
uint32_t EscapeBuffer30 = EscapeBuffer->index;
if (EscapeBuffer30 <= AdapterObject->Offset135D0MaxIndex) {
    uint32_t index = AdapterObject->Offset3908Array[EscapeBuffer30];
    uint32_t value = AdapterObject->Offset89B0Array[index * 0x18];
    // ...
    EscapeBuffer->OutputValue = value;
    // ...
}
```





Transforming the Corruption into a Memory Leak

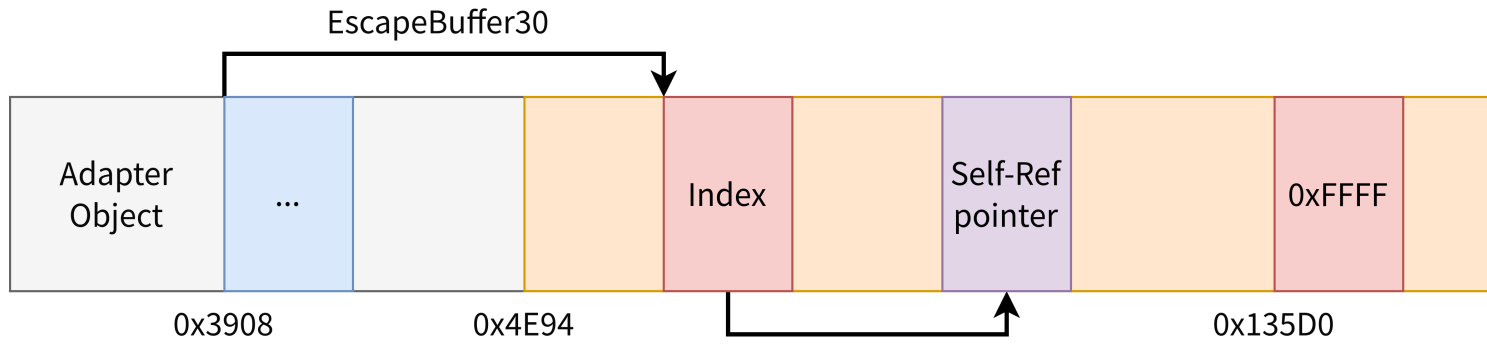
```
// Private Escape Code 0x2000041
uint32_t EscapeBuffer30 = EscapeBuffer->index;
if (EscapeBuffer30 <= AdapterObject->Offset135D0MaxIndex) {
    uint32_t index = AdapterObject->Offset3908Array[EscapeBuffer30];
    uint32_t value = AdapterObject->Offset89B0Array[index * 0x18];
    // ...
    EscapeBuffer->OutputValue = value;
    // ...
}
```





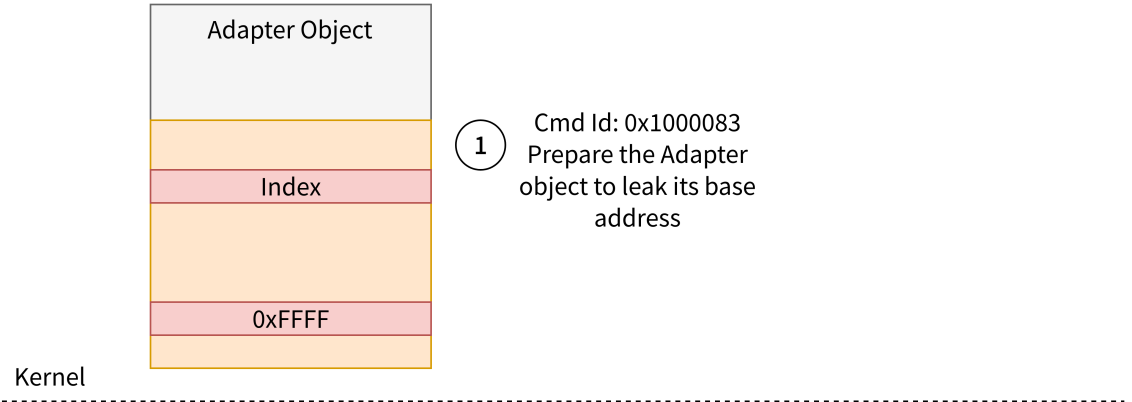
Transforming the Corruption into a Memory Leak

```
// Private Escape Code 0x2000041
uint32_t EscapeBuffer30 = EscapeBuffer->index;
if (EscapeBuffer30 <= AdapterObject->Offset135D0MaxIndex) {
    uint32_t index = AdapterObject->Offset3908Array[EscapeBuffer30];
    uint32_t value = AdapterObject->Offset89B0Array[index * 0x18];
    // ...
    EscapeBuffer->OutputValue = value;
    // ...
}
```



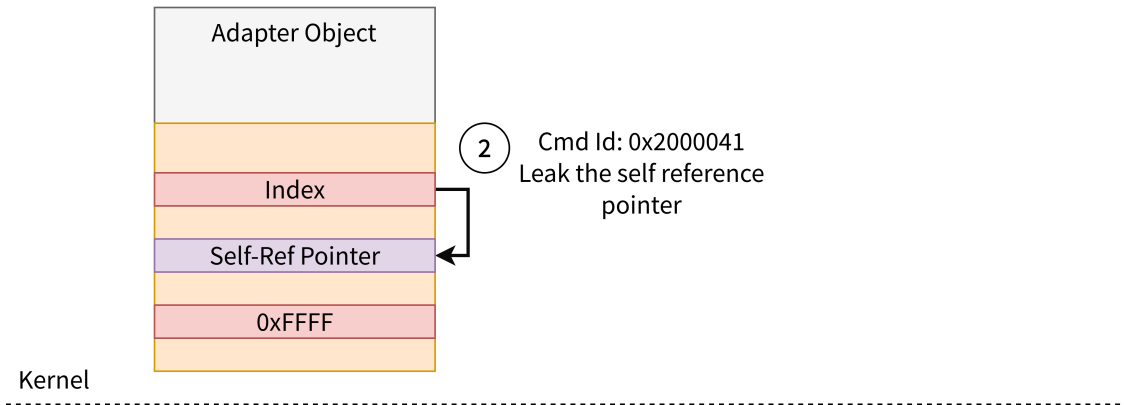


Final Exploitation Step



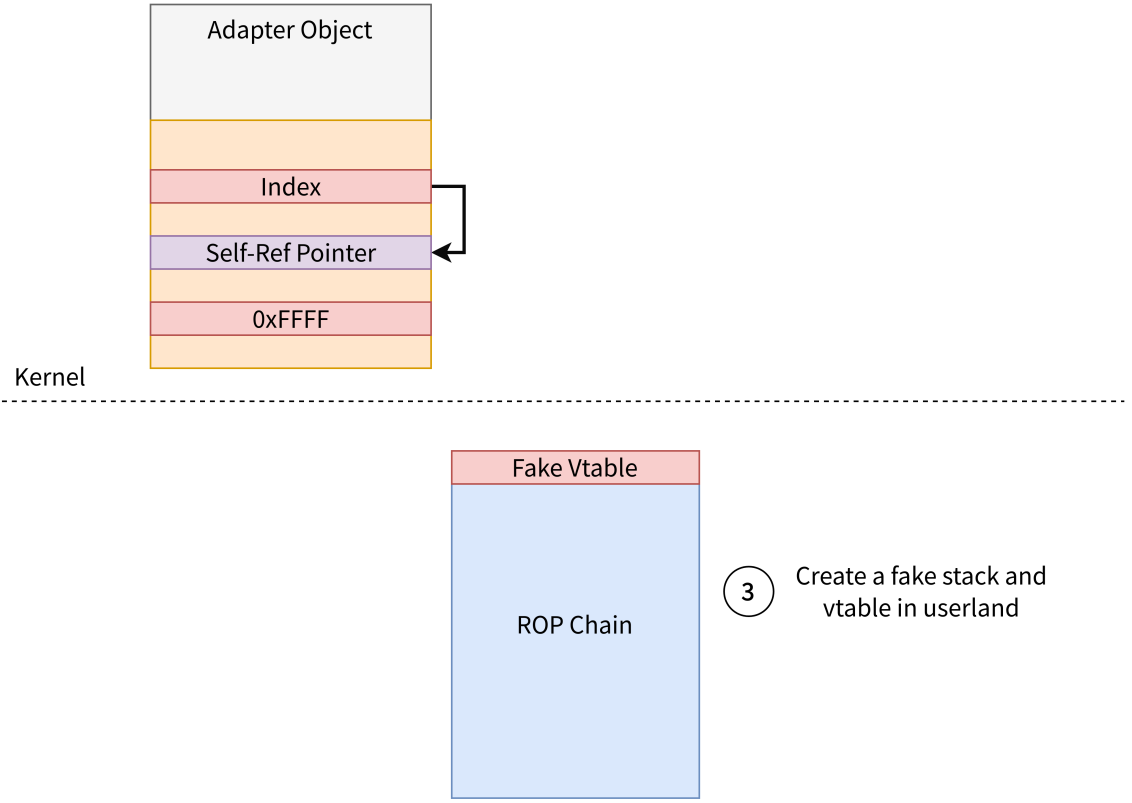


Final Exploitation Step



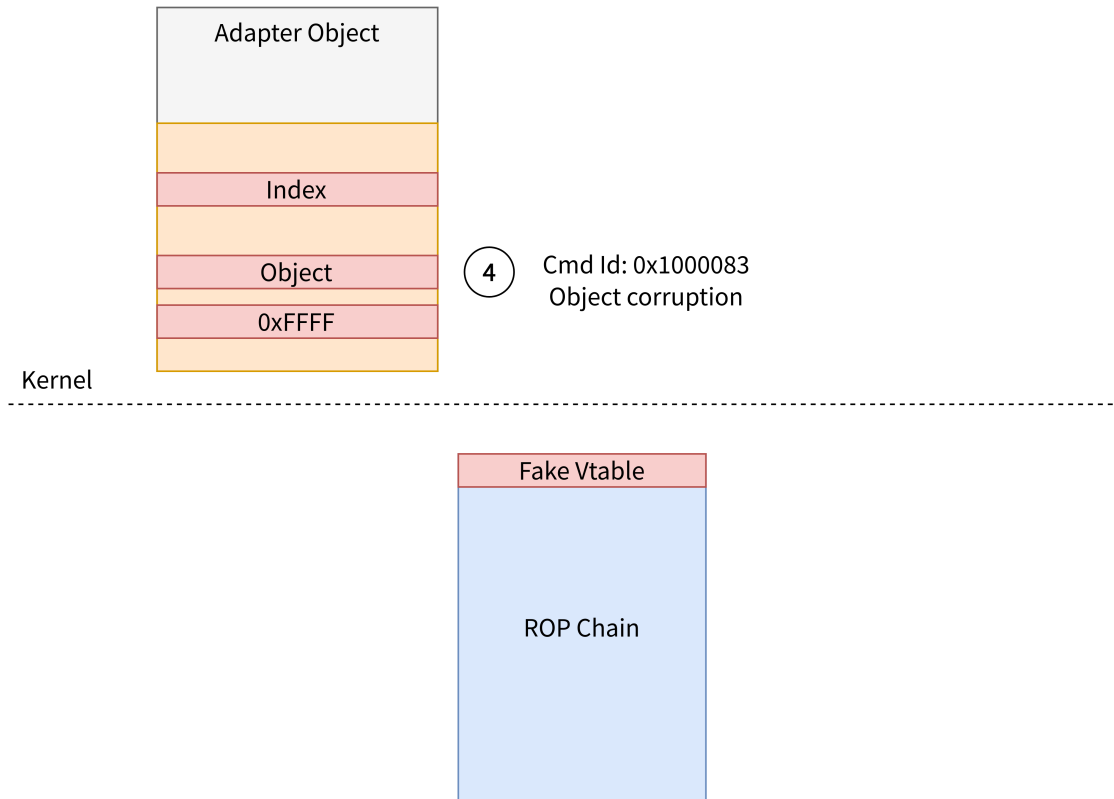


Final Exploitation Step



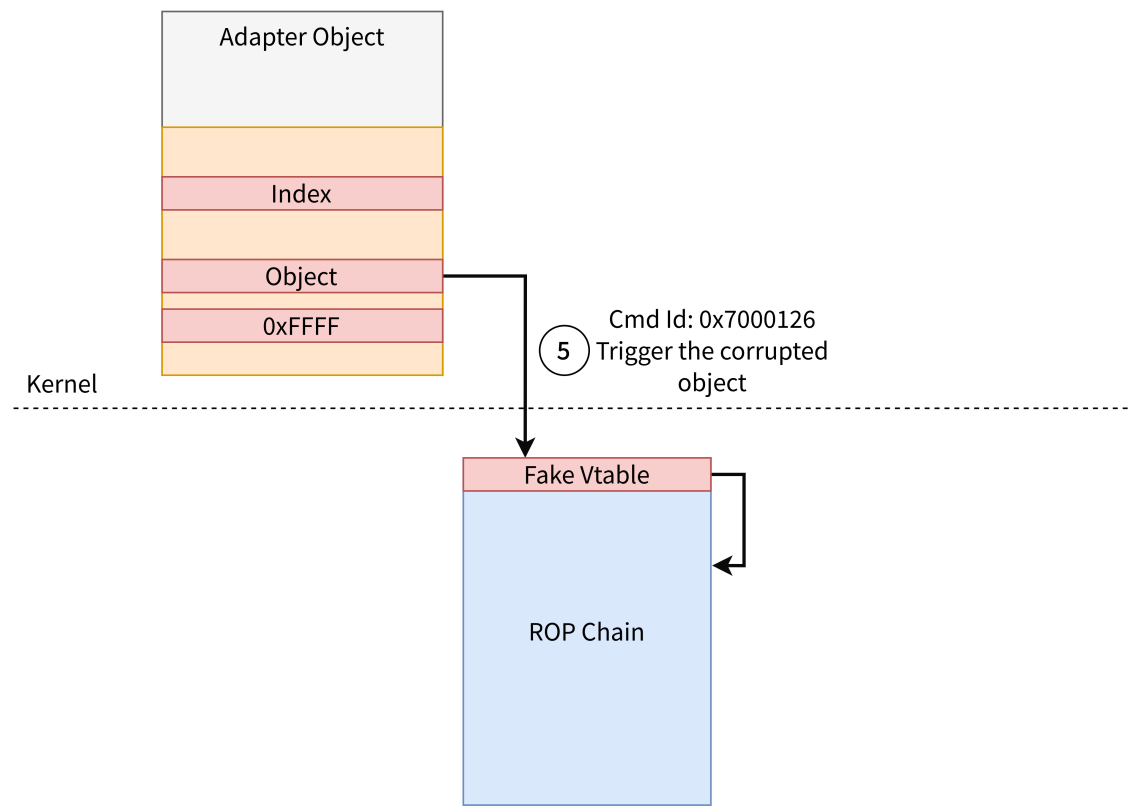


Final Exploitation Step



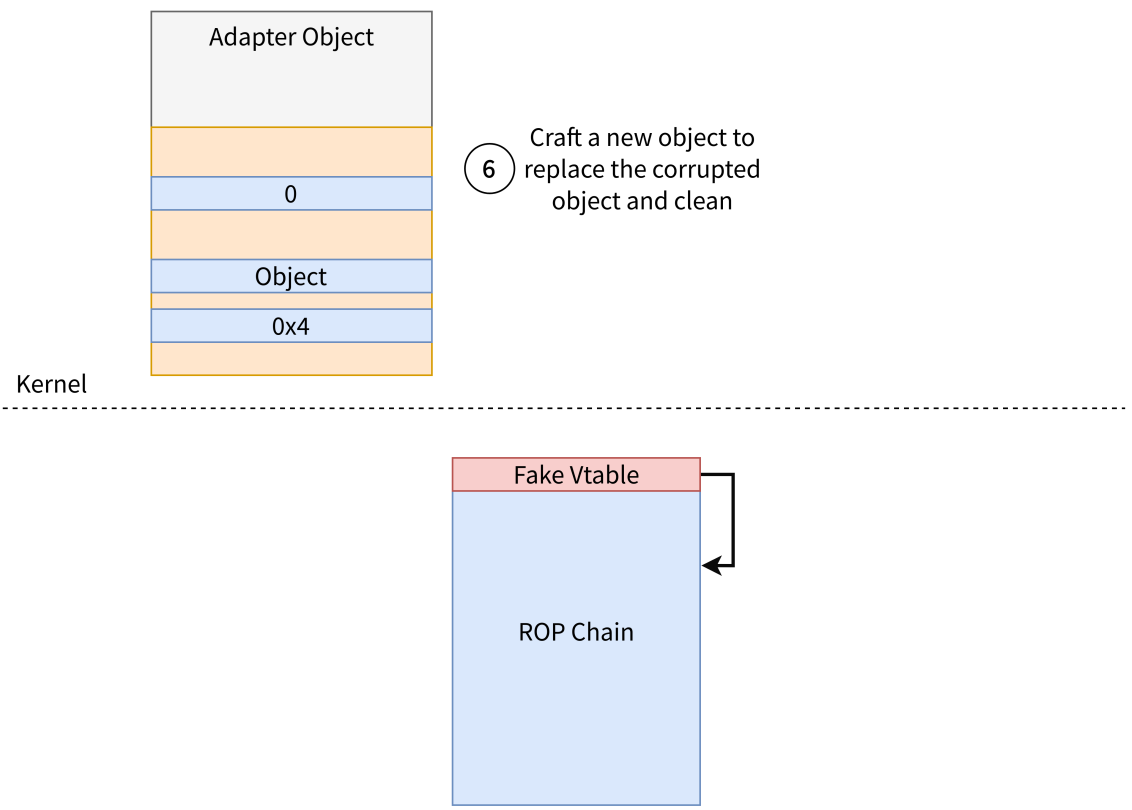


Final Exploitation Step



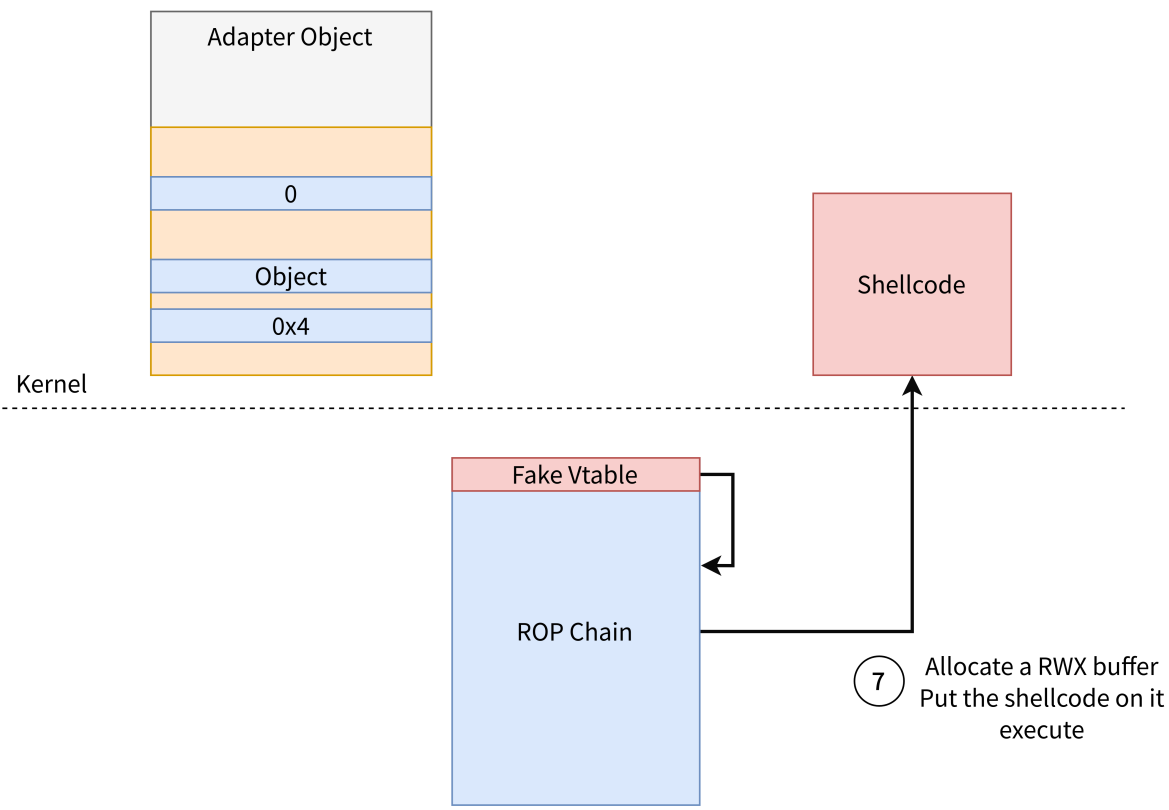


Final Exploitation Step





Final Exploitation Step





```
PS D:\build> whoami.exe
desktop-2rqdc6d\pissenlit
PS D:\build> .\exploit.exe
[Main] Sleeping for 10s ...
[Main] Nvidia driver base address: 0xfffff803b0cd0000
[OOB Write] write 0x22 at offset 0xa778
[OOB Write] write 0x100 at offset 0x135d0
[OOB Write] write 0x4 at offset 0x135d0
[OOB Write] write 0x0 at offset 0xa778
[Main] Adapter object address: 0xffff880410ae9000
[Main] Allocate the fake Object Address at 0x0000001000000000
[Main] Object pointer corruption
[OOB Write] write 0x0 at offset 0x5bf0
[OOB Write] write 0x10 at offset 0x5bf4
Microsoft Windows [Version 10.0.19044.2130]
(c) Microsoft Corporation. All rights reserved.

D:\build>whoami
nt authority\system

D:\build>)
```



Timeline

- 3 May 2022: Disclose the vulnerabilities to Nvidia
- 17 May 2022: Notice us that they manage to validate the findings and plan to release a patch in August
- 2 August 2022: All the bugs disclosed have been patched (CVE-2022-31606, CVE-2022-31612, CVE-2022-31616, and CVE-2022-31617)

Security Bulletin: NVIDIA GPU Display Driver - August 2022

https://nvidia.custhelp.com/app/answers/detail/a_id/5383



Questions?

Quarkslab

/HEXACON/