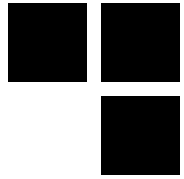


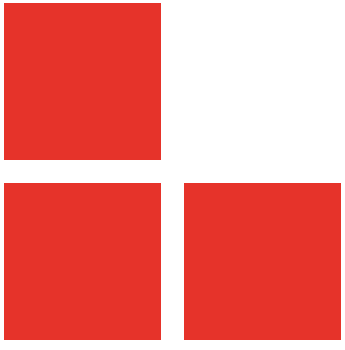
SpeedPwning VMware Workstation

Failing at Pwn2Own, but doing it fast

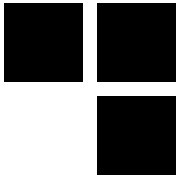
Corentin Bayet (@OnlyTheDuck) & Bruno Pujos (@BrunoPujos)

Ekoparty 2020

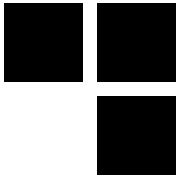
September 18, 2020



Plan

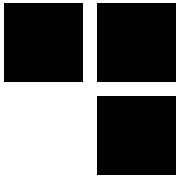


What is this ?



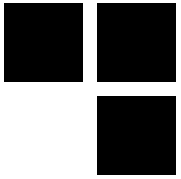
- This talk is about the research we did for Pwn2Own targeting VMware Workstation.
- We wanted to share:
  - Our methodology,
  - Our technology,
  - A funny story !
- We have a LOT to share:
  - Don't be scared by the amount of slides, everything will be ok !
  - As during the research, we will go fast, some parts lack of details.
  - We would be pleased to answer your questions during the live Q&A or after !

Who are we ?



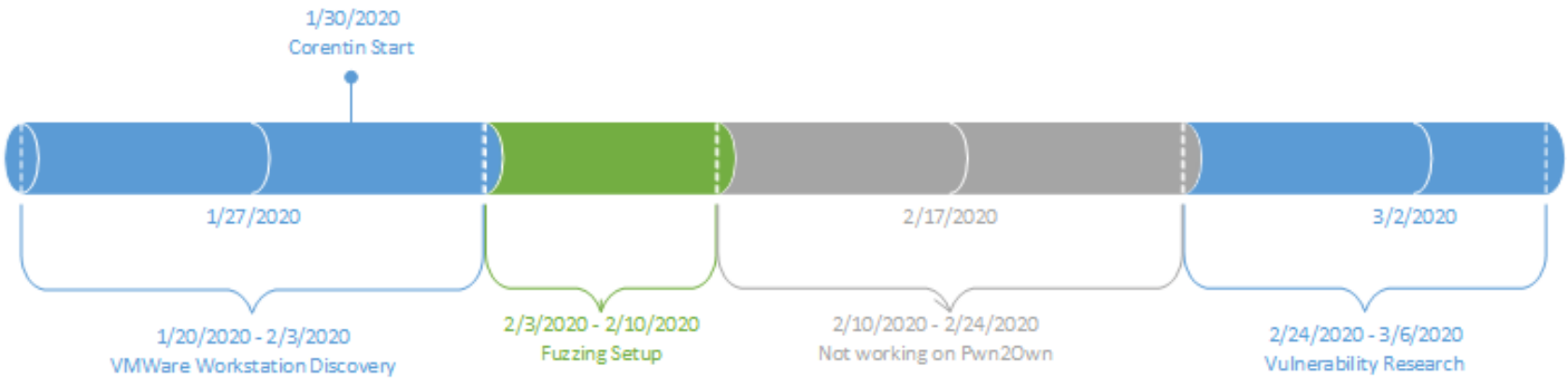
- Bruno Pujos (@BrunoPujos)
  - Security researcher & consultant at Synacktiv.
  - Previous work on UEFI and SMM.
- Corentin Bayet (@OnlyTheDuck)
  - Security researcher & consultant at Synacktiv.
  - Previous work on Windows kernel heap exploitation.
- Both interested in virtualization technologies.

Introduction

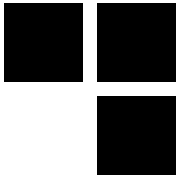


- Pwn2Own returns in 2020 with a virtualization category.
- We're interested in trying to participate in the contest.
- We have only 40 (work) days to score (20 days each).
- We picked VMware Workstation:
  - + our favorite desktop virtualization software,
  - + target previously documented,
  - + seems doable in limited time.
- We already worked on virtualization components, but never on Workstation.

Planning


1/30/2020
Corentin Start
1/27/2020
2/17/2020
3/2/2020
1/20/2020 - 2/3/2020
VMWare Workstation Discovery
2/3/2020 - 2/10/2020
Fuzzing Setup
2/10/2020 - 2/24/2020
Not working on Pwn2Own
2/24/2020 - 3/6/2020
Vulnerability Research

Plan



1 Introduction

2 Workstation Discovery
- Virtualization for dummies
- Workstation Discovery

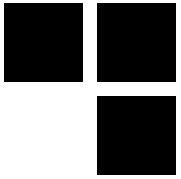
3 Vulnerability Research

4 Exploit

5 Conclusion

6 Annexes

Plan



Virtualization for dummies

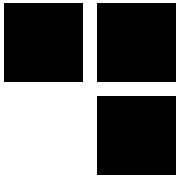
Virtualization ?

- Virtualization allows to "emulate" the hardware of a computer.
- Allows to run different(s) OS on the same hardware.
- A **guest** or Virtual Machine (VM) is an OS which run with the emulated hardware.
- The **host** is the main (real) OS of the computer.
- The **hypervisor** is all the code for handling the guest.

Security

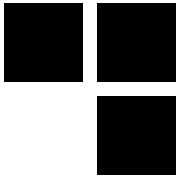
- The host should be isolated from the guests.
- A **VM Escape** allows to get code execution in the host from the guest.

Virtualization for dummies: devices

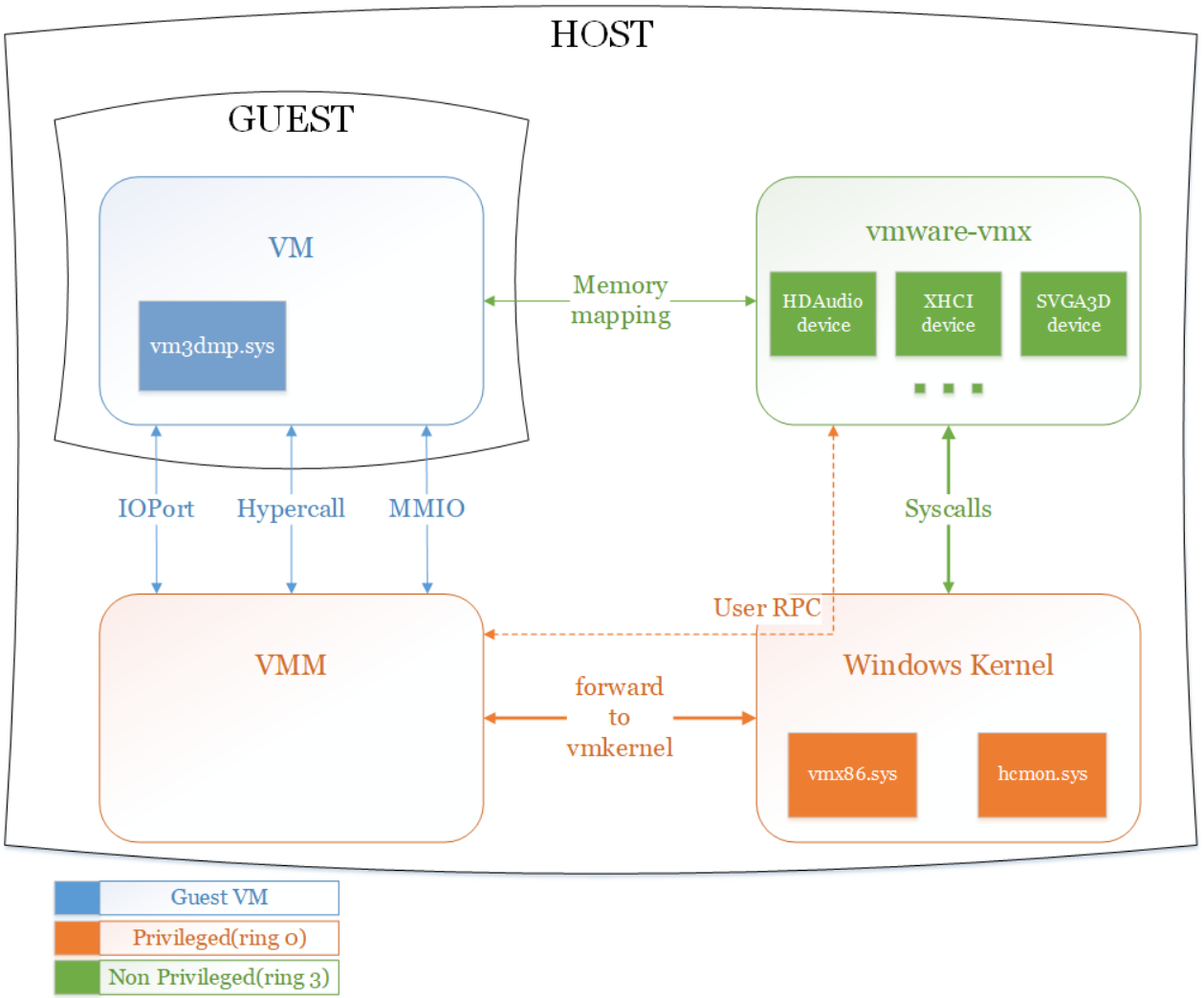


- The hypervisor (host) emulates some hardware components for the guest: the **devices**.
- "Hardware assisted virtualization" (VT-x on Intel): use hardware features for helping emulate the guest hardware.
- Devices use "traditional" technologies: PCI(e), IOPorts, MMIO, DMA…
- Devices can be: network card, USB, audio, graphic, printer, hard drive… All of this must be emulated (and more).
- Emulated devices are the main attack surface from the guest.

Plan



VMware Workstation



General Reverse

Locate user input

- Functions allowing to register IOPort, MMIO, PCI...
- Functions for read/write/map the guest memory.

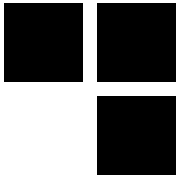
Debug Symbols

- Rename all globals setup from the configuration.
- Rename lock functions from the open-vm-tools.
- Locate and rename functions from debug strings.
- Lot of symbols from the code loading a snapshot.

Misc

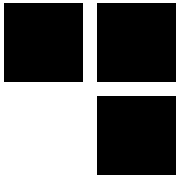
- A *vmware-vmx-debug.exe* exists: more debug strings but also more checks.
- Init. of the devices documented in *Straight Outta VMware*.

Target selection



- Our goal is to demonstrate a **VM Escape**.
- We choose 3 targets, even if we thought we would have only time to investigate 2 of them.
- Focus on target mostly implemented in *vmware-vmx.exe*.
- Pwn2own: guest and host Windows 10, with default configuration.
- Our choice was the following one:
  - USB: looked complex, vulnerability in the past.
  - Audio: no prior work at all, potential for parsing.
  - SVGA: complex, vulnerability in the past, error prone.

Plan



1 Introduction

2 Workstation Discovery

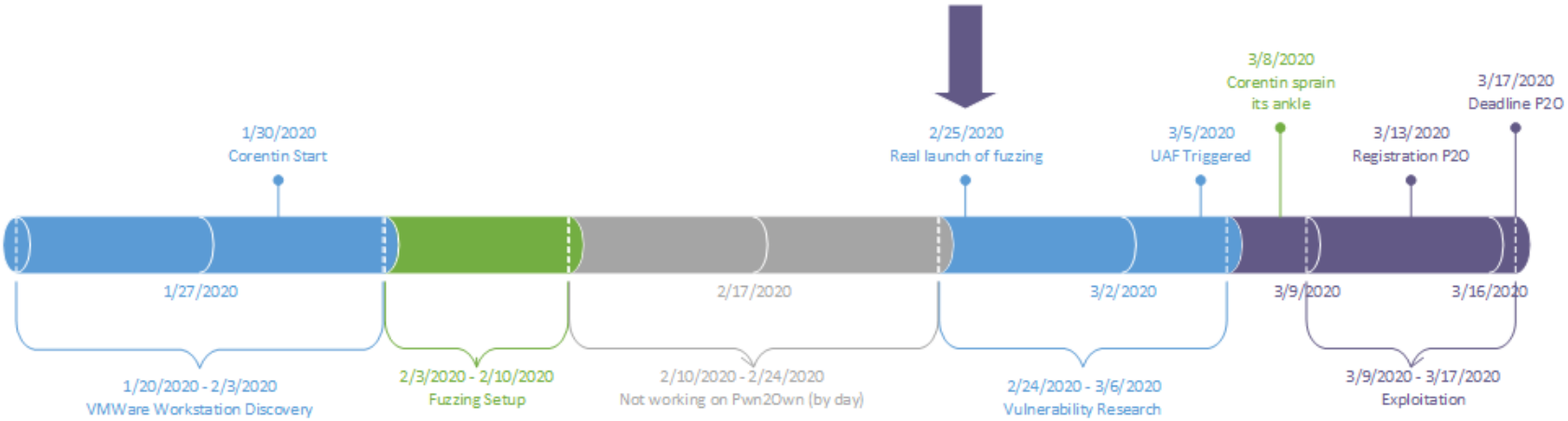
3 Vulnerability Research
- Misc research results
- SVGA2
- Vulnerability

4 Exploit

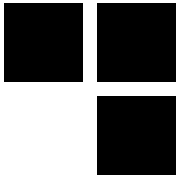
5 Conclusion

6 Annexes

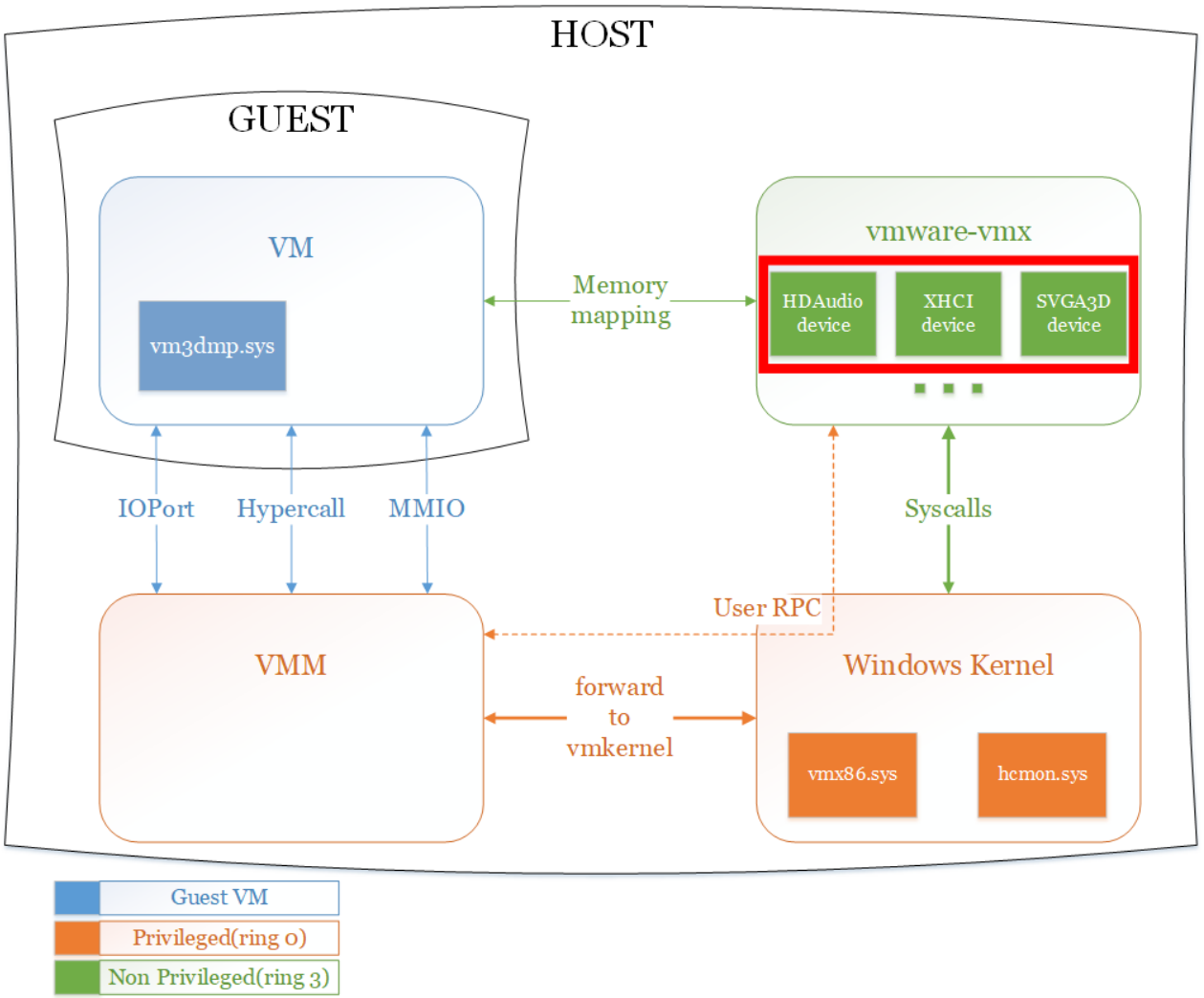
Planning


1/30/2020
Corentin Start
2/25/2020
Real launch of fuzzing
3/5/2020
UAF Triggered
3/8/2020
Corentin sprain its ankle
3/13/2020
Registration P2O
3/17/2020
Deadline P2O
1/27/2020
2/17/2020
3/2/2020
3/9/2020
3/16/2020
1/20/2020 - 2/3/2020
VMWare Workstation Discovery
2/3/2020 - 2/10/2020
Fuzzing Setup
2/10/2020 - 2/24/2020
Not working on Pwn2Own (by day)
2/24/2020 - 3/6/2020
Vulnerability Research
3/9/2020 - 3/17/2020
Exploitation

Plan



VMware Workstation architecture



Quick vulnerabity research results

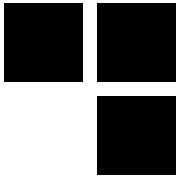
HDAudio

- Only forwards to **waveIn*** and **waveOut*** APIs of **winmm.dll**.
- No parsing, only raw data audio transmission to hardware.
- Very few code, not interesting.

XHCI (USB 3.0)

- More code than HDAudio.
- But most of it is unreachable without plugging specific USB devices, which is out of scope.
- Reachable code in the default configuration partially audited, found a useless bug (patched since).

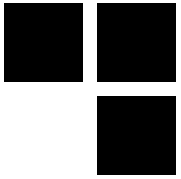
Quick vulnerabity research results



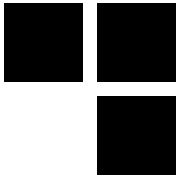
Fuzzing

- Tried to fuzz every component we audited.
- With WinAFL and Synacktiv's internal fuzzer.
- Hard to implement, took a lot of time.
- No results, big regret on spending too much time trying to fuzz.

Plan

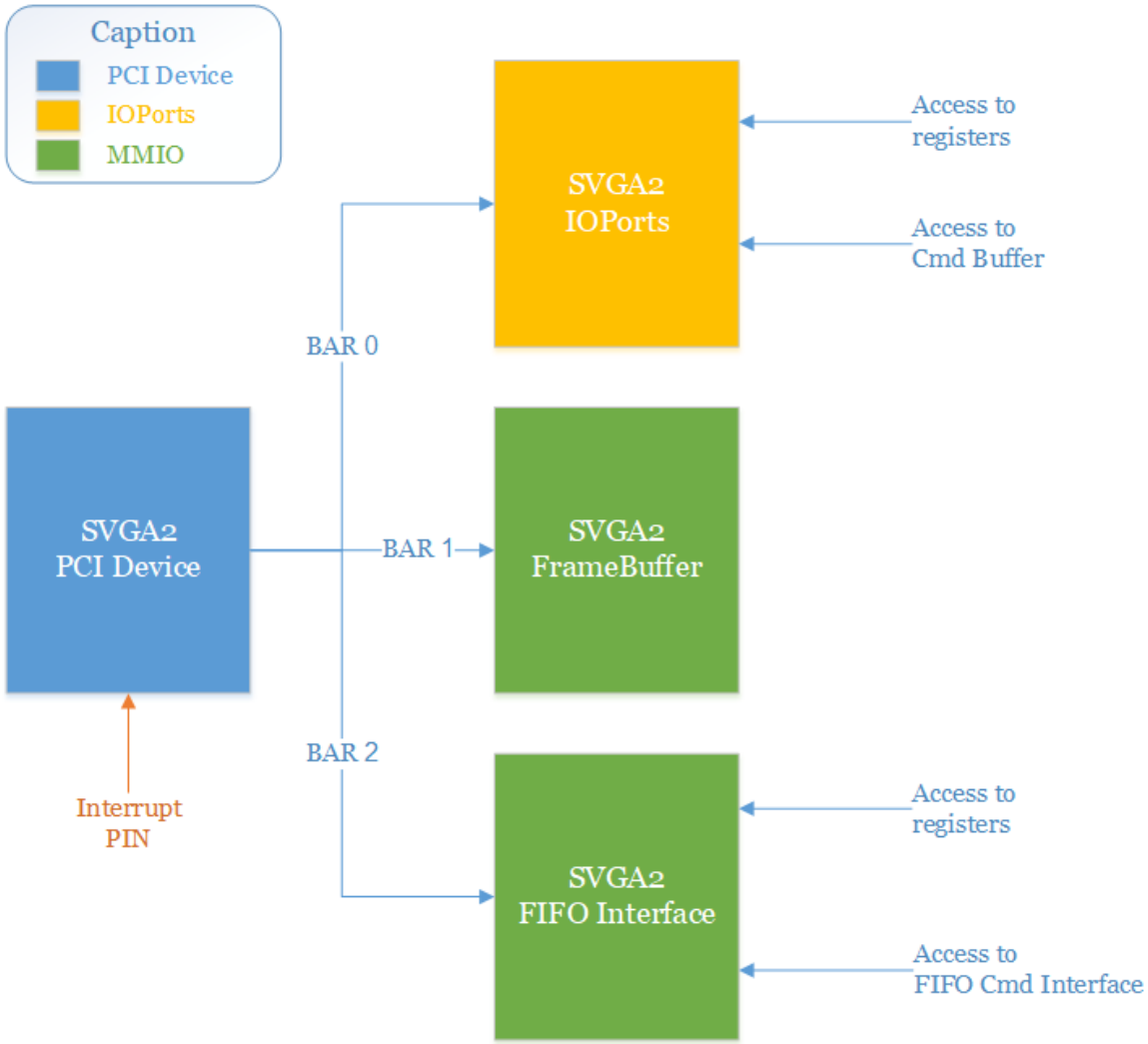


SVGA2



- Graphical interface for VMware products.
- "Para-virtualized": no physical device exists, "idealized" version of the hardware.
- The backend implementation is made for being as fast as possible and depends on the host.
- We only look at the 3D part of the API.
- Papers:
  - *GPU Virtualization on VMware's Hosted I/O Architecture* by Micah Dowty and Jeremy Sugerman,
  - *Straight outta VMware* by Zisis Sialveras.

SVGA2 guest view



Graphics object & ArrayID

Graphics object

- Objects are everything we usually see: shaders, surfaces…
- Objects are read from DMA zone in the guest and are lazy-loaded by the VMX.
- Possible to *readback*: for sync. between host and guest.

ArrayID

- Array for storing the graphical objects on the VMX side.
- Function for adding, removing and searching in those.
- When an element is added to an array, the memory for this element is allocated, and freed when removed.

Direct X API

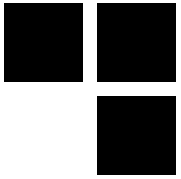
Commands & APIs

- Define graphical objects through a set of commands.
- Several APIs (sets of commands) are exposed through the commands and they can be used concurrently.
- Commands/API enabled depend on config., host, guest…
- We used two of them:
  - the GB (global ?): the "normal" one,
  - the DX (Direct X) which is specific to Windows host.

DX API

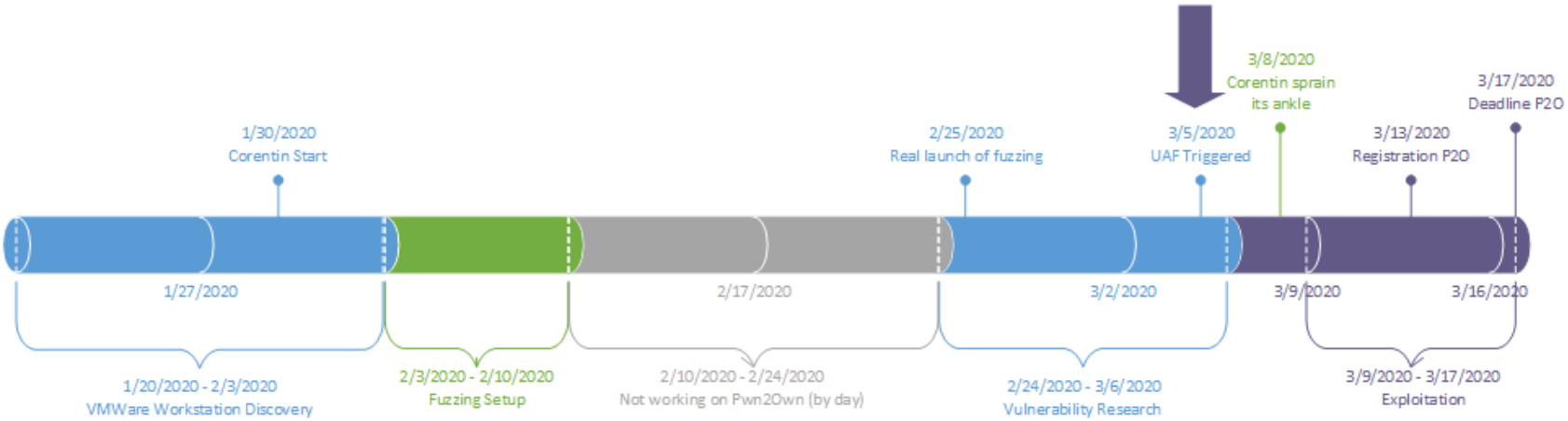
- The DX objects are always associated with a `DxContext`.
- The `DxContext` is a classical graphical object.
- Most commands of the DX API use a *current context* provided by the guest for a batch of commands.

Back-end Renderer

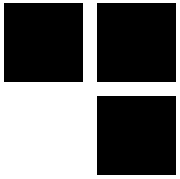


- When a "real" action is made using the objects loaded, they will be passed to a backend: the renderer.
- The selected renderer depends on host, guest, config…In our case (Windows guest and host), the DX11_Renderer is used.
- We did not reverse this (but we used it later on).
- The graphical object will be used for creating new objects for the renderer.
- We called those "Resource Container", from their name in *Straight outta VMware*.
- The "Resource Container" will then make the transition with the real DirectX API (D3D11, …).

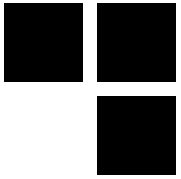
Planning


1/30/2020
Corentin Start
2/25/2020
Real launch of fuzzing
3/5/2020
UAF Triggered
3/8/2020
Corentin sprain its ankle
3/13/2020
Registration P2O
3/17/2020
Deadline P2O
1/27/2020
2/17/2020
3/2/2020
3/9/2020
3/16/2020
1/20/2020 - 2/3/2020
VMWare Workstation Discovery
2/3/2020 - 2/10/2020
Fuzzing Setup
2/10/2020 - 2/24/2020
Not working on Pwn2Own (by day)
2/24/2020 - 3/6/2020
Vulnerability Research
3/9/2020 - 3/17/2020
Exploitation

Plan



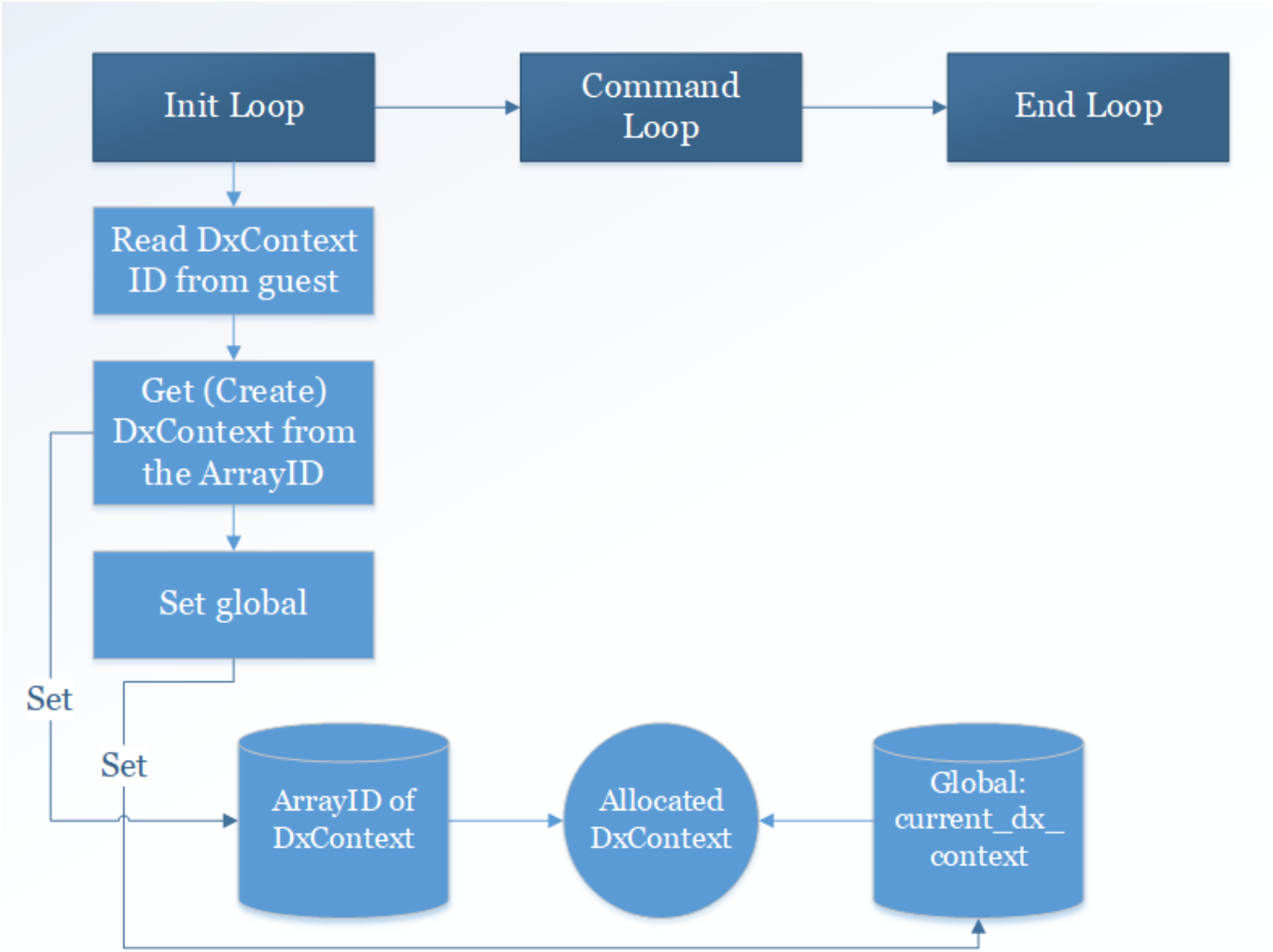
Vulnerability



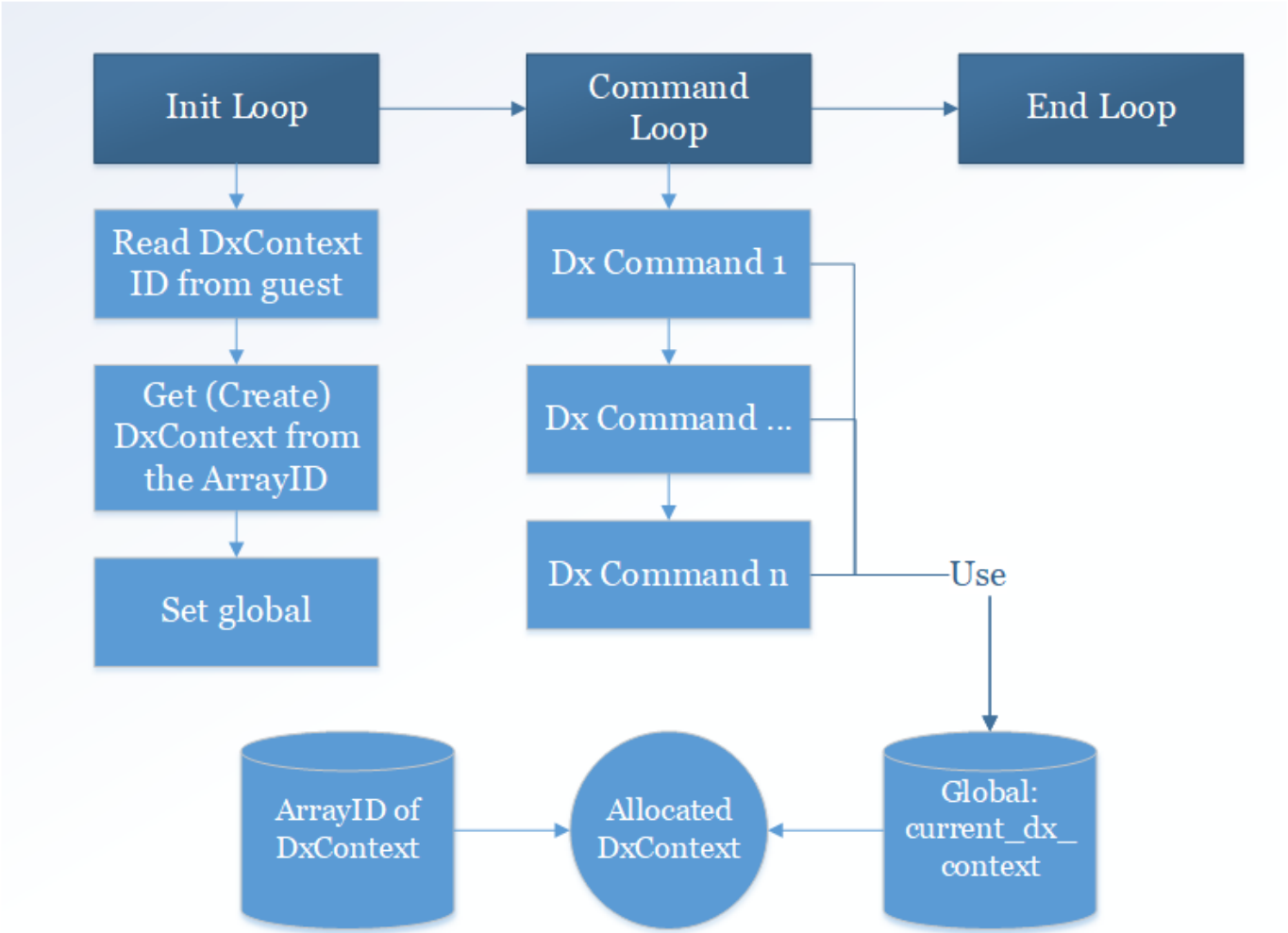
- Able to trigger the vulnerability ~14 hours before the deadline for the end of the research.
- Vulnerability identified in the handling of the `DxContext`:
  - Linked to the fact that the DX commands use a `DxContext`.
  - The `DxContext` used is fetch from a global variable: `current_dx_context`.

Let's look precisely at how the svga thread handle dx commands.

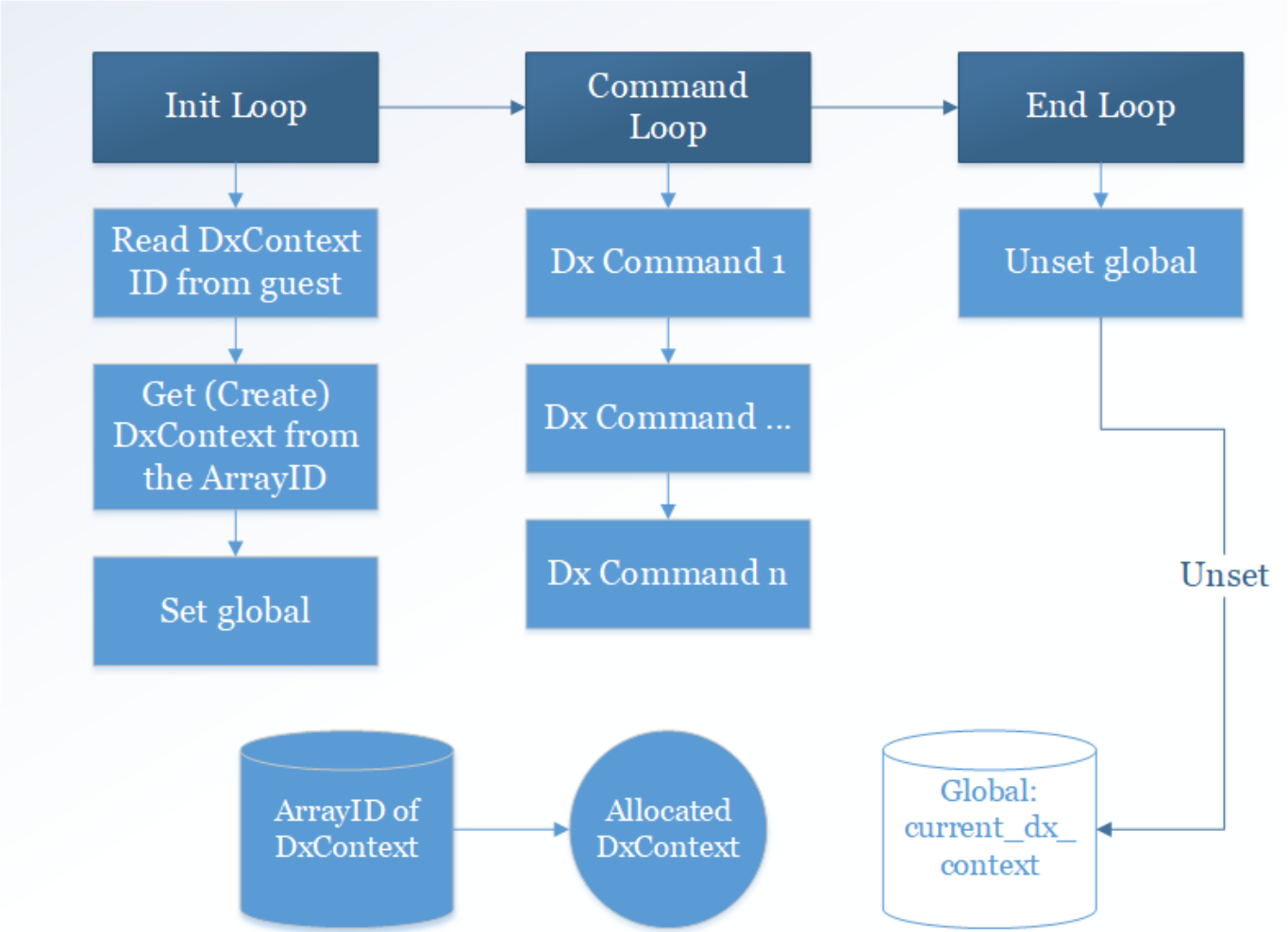
Execution of DX Commands



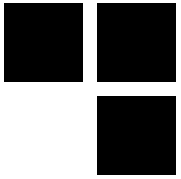
Execution of DX Commands



Execution of DX Commands

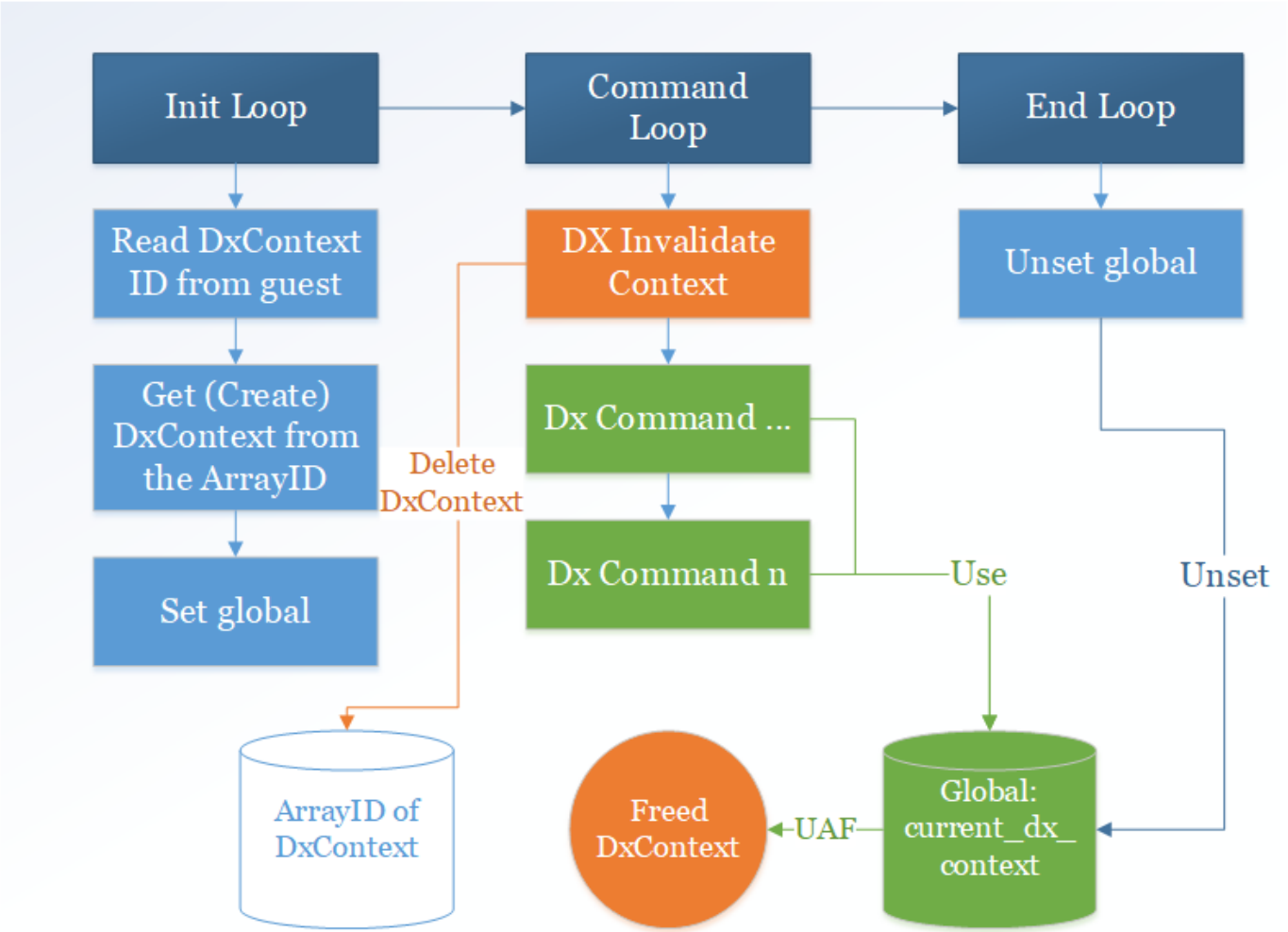


The UAF

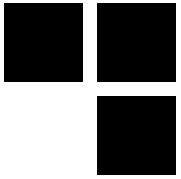


- A command `DX_INVALIDATE_CONTEXT` exist:
  1. the command calls `delete_DXContext`,
  2. `delete_DXContext` delete a `DxContext` from the `ArrayID`,
  3. the `DxContext` object is `free`.
- No other check on the `current_dx_context`!
- We got an UAF!

DxContext UAF

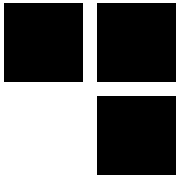


Vulnerability recap.



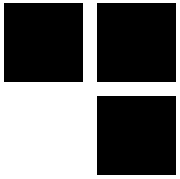
- We got an UAF on a global `current_dx_context`.
- The global pointer on the free chunk is lost when leaving the command loop:
  - if an asynchronous action is triggered,
  - if a command fails,
  - if there is no other commands.
- There is a restricted number of commands that uses the `current_dx_context`.

Of course we have time



- Registration for P2O ends in a week:
  - we have a POC for triggering,
  - setup for interfacing with the device is "not great",
  - we have no idea what most of the commands do,
  - we did not even start looking at the backend,
  - …
- Do we try it ? Of course we do :)

Plan



1 Introduction

2 Workstation Discovery

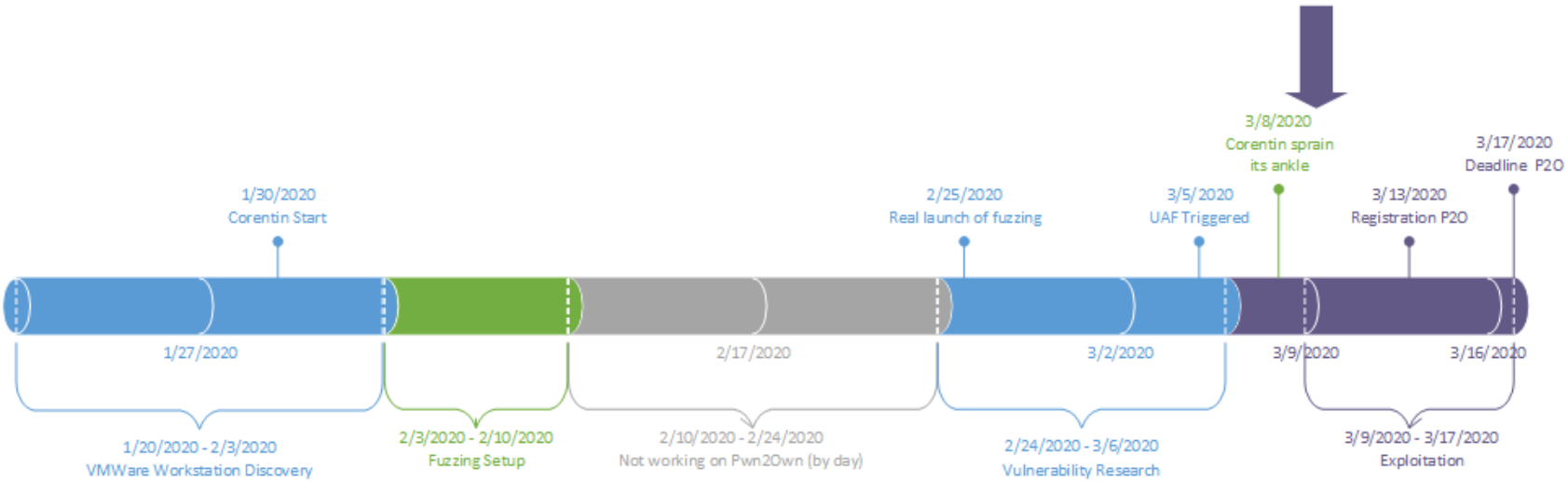
3 Vulnerability Research

4 Exploit
- Exploit Strategy
- Getting a heap leak
- Getting a .text leak
- Pop a notepad
- Exploit conclusion
- Pwn2own ?

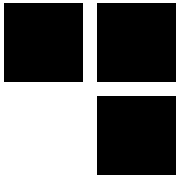
5 Conclusion

6 Annexes

Planning


3/8/2020
Corentin sprain its ankle
3/17/2020
Deadline P2O
1/30/2020
Corentin Start
2/25/2020
Real launch of fuzzing
3/5/2020
UAF Triggered
3/13/2020
Registration P2O
1/27/2020
2/17/2020
3/2/2020
3/9/2020
3/16/2020
1/20/2020 - 2/3/2020
VMWare Workstation Discovery
2/3/2020 - 2/10/2020
Fuzzing Setup
2/10/2020 - 2/24/2020
Not working on Pwn2Own (by day)
2/24/2020 - 3/6/2020
Vulnerability Research
3/9/2020 - 3/17/2020
Exploitation

Plan

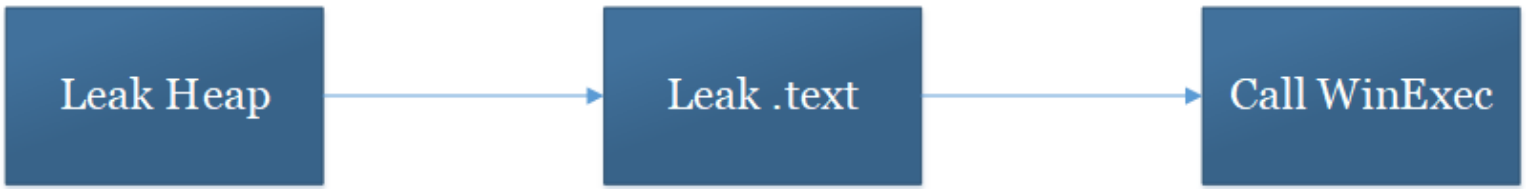


Planning


3/9/2020
Start Exploitation
3/13/2020
Registration
Deadline
3/18/2020
Send to ZDI
3/9/2020
3/10/2020
3/11/2020
3/12/2020
3/13/2020
3/14/2020
3/15/2020
3/16/2020
3/17/2020
3/17/2020
Goal was to have
working exploit

Exploitation strategy

- This is Pwn2Own, we just need to pop a calc or a notepad to demonstrate the exploit.
- We can call `WinExec` with the first argument pointing on controlled data. We need :
  - A leak of the base of vmware-vmx.exe, to know the address of the import table.
  - An arbitrary read, to read an address of a function of kernel32 to compute `WinExec`'s.
  - An arbitrary call, to jump on `WinExec`.



- Disclaimer: there is probably a better way to write this exploit… but we did not have time.

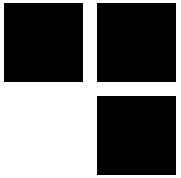
Exploitation strategy

Problem with the DxContext UAF

- Almost no pointers in the `DxContext`, or not used to read or write data;
- Size of 0x5B68, with no time to reverse all the backend and the fields.

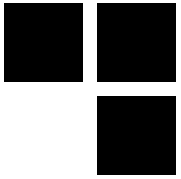
- Only thing we can do is control some `DWORDs` stored in the `DxContext` with various commands.
  - The only strategy is to realloc the chunk in UAF and overwrite some `DWORDs` using these commands.
  - With heap massaging, we can control the alignment between the UAF chunk and the target object.

Exploit - Massaging the heap



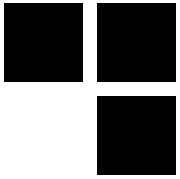
- The segment heap is not enabled, so it's the classic NT heap.
  - All alloc with size < 0x3FFF will use LFH if used enough.
  - A great slide deck on NT heap internals is available here.
- The UAF object has a size of 0x5B68, it will never be handled by LFH.
  - We can massage the heap to align the `DxContext` in UAF with other objects.
  - We need a good massaging primitive.
- The SVGA provides a perfect massaging primitive with the `GbShaders`: Command `set_gb_shader`:
  - alloc size fully controlled,
  - data in the allocation fully controlled,
  - can be freed at anytime,
  - Previously described in 2018 by @_zisis in a great paper.

Exploit - Chosing a target object



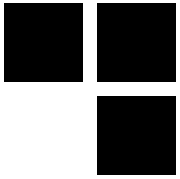
- The objects that can be targeted are very limited:
  - cannot reliably target objects with a size < 0x3FFF, because of LFH,
  - need to control its allocation and free.
- Again, no time to reverse too much of the backend.
- The `GbContext` is another object containing a lot of information:
  - size of 0x5490, won't use LFH,
  - ability to readback a part of its content,
  - can be easily allocated and freed.
- Might be a good target.

Exploit Strategy

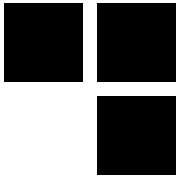


- The vulnerability can be used multiple times to obtain different primitives.
- Using this strategy, we successfully obtained:
  - a heap pointer leak,
  - an arbitrary free primitive (with some constraints),
  - an arbitrary call primitive (with some constraints).
- Those primitives are enough to get everything needed.
- Let's see how.

Plan

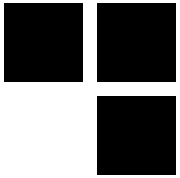


Exploit - Heap Leak



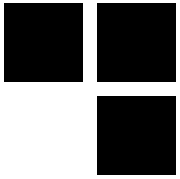
- First steps is to get a leak.
- The *readback* of the `DxContext` cannot be used because it does not use the global.
- Only two possibilities using the UAF:
  1. Leak a pointer stored in the `DxContext` using another object *readback*.
  2. Trigger another memory corruption which can provide us a leak.
- First possibility seems best.
- Wait is there even one pointer ?

Exploit - DxContext Pointer



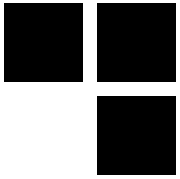
- Most front-end objects do not store the pointer of other objects but simply their ID for the `ArrayID`.
- Luckily we were able to find **one** pointer!
- Pointer for a `DxShader`:
  - the object will be added to an `ArrayID`,
  - the shader content will be allocated and the pointer is stored in the `DxContext` object!
- This would give us a leak in the heap, with partial control of the allocated size.
- Seems perfect, let's do it.

Adding a DxShader



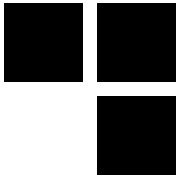
- Adding a DxShader is quite simple, the command `dx_set_shader` will do it for us.
- A `current_dx_context` must be set and the *type* of the shader is verified.
- Then it will make the following steps:
  1. fetch the `DxShader` struct from the guest,
  2. read the content of the shader from the guest,
  3. check that the content of the shader is valid,
  4. check that the shader is not present in the associated `DxContext`.
- Steps 1 & 2 are classic.
- Steps 3: we need a correct `DxShader`.

DxShader format



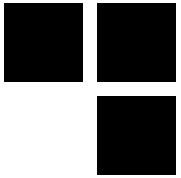
- The shader is apparently using the SML4 format.
- The galium header from Mesa provides several interesting information about it.
- The `DxShader` is decomposed into two main parts:
  1. A first header followed by an array of tokens: this is the main part of the shader.
  2. A second optional header (which does not seem documented) which contains 3 arrays of unknown elements.
- The second header can be hardcoded.
- Without using the first header we could have a maximum size of 0x3C18: perfect for a leak into the NT Heap.

Reading back the DxShader pointer



- We now need an object for re-use of the UAF allowing us to readback the pointer.
- The size of the `DxContext` object is 0x5B68.
- The size of the `GbContext` object is 0x5490.
- This looks like a good match!
- We can readback a `GbContext` and it happens that the location of the `DxShader` pointer is in a part of the `GbContext` which can be readback!

A little problem

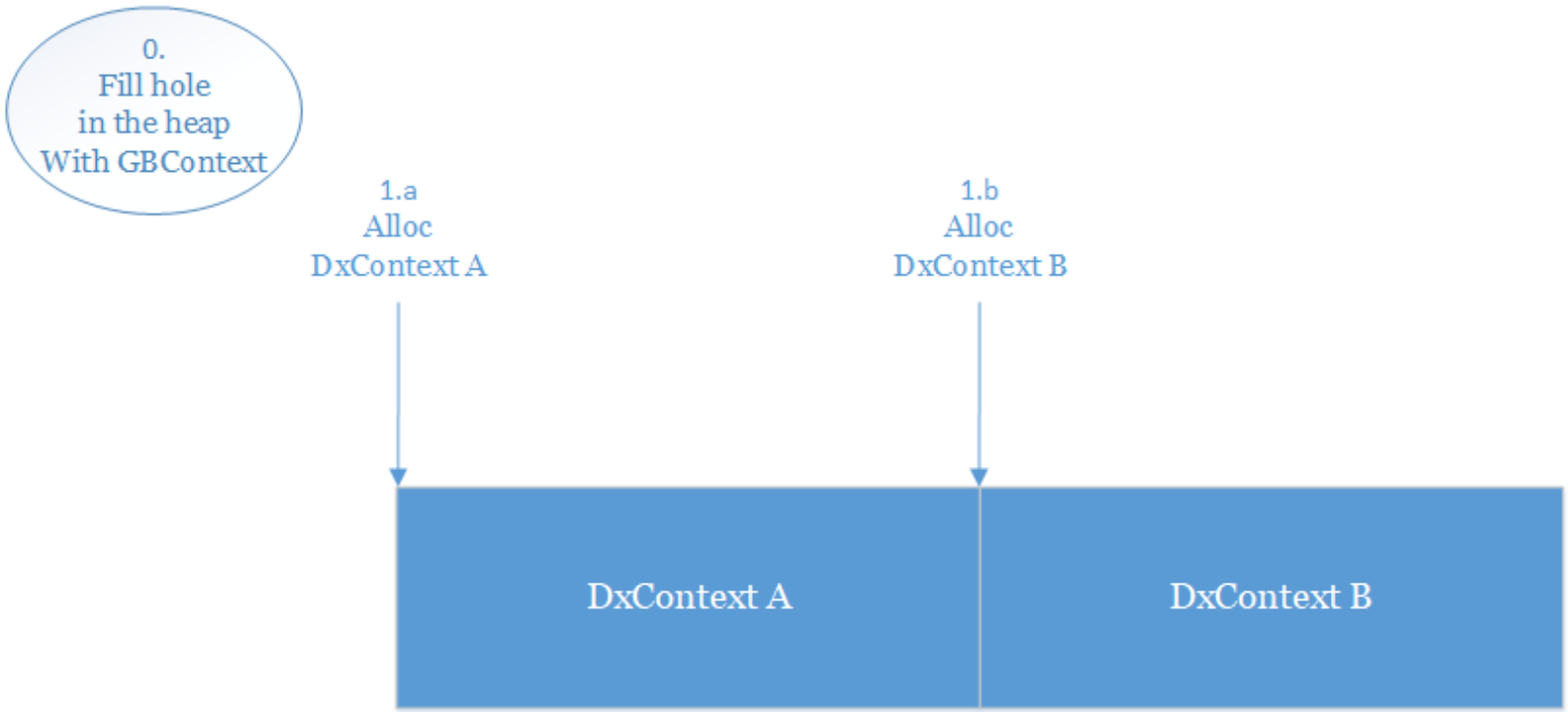


- After the parsing of a `DxShader`, the `DxContext` will be recuperated from the `ArrayID`.
- If the `DxContext` is not in the `ArrayID`, we have a NULL deref:

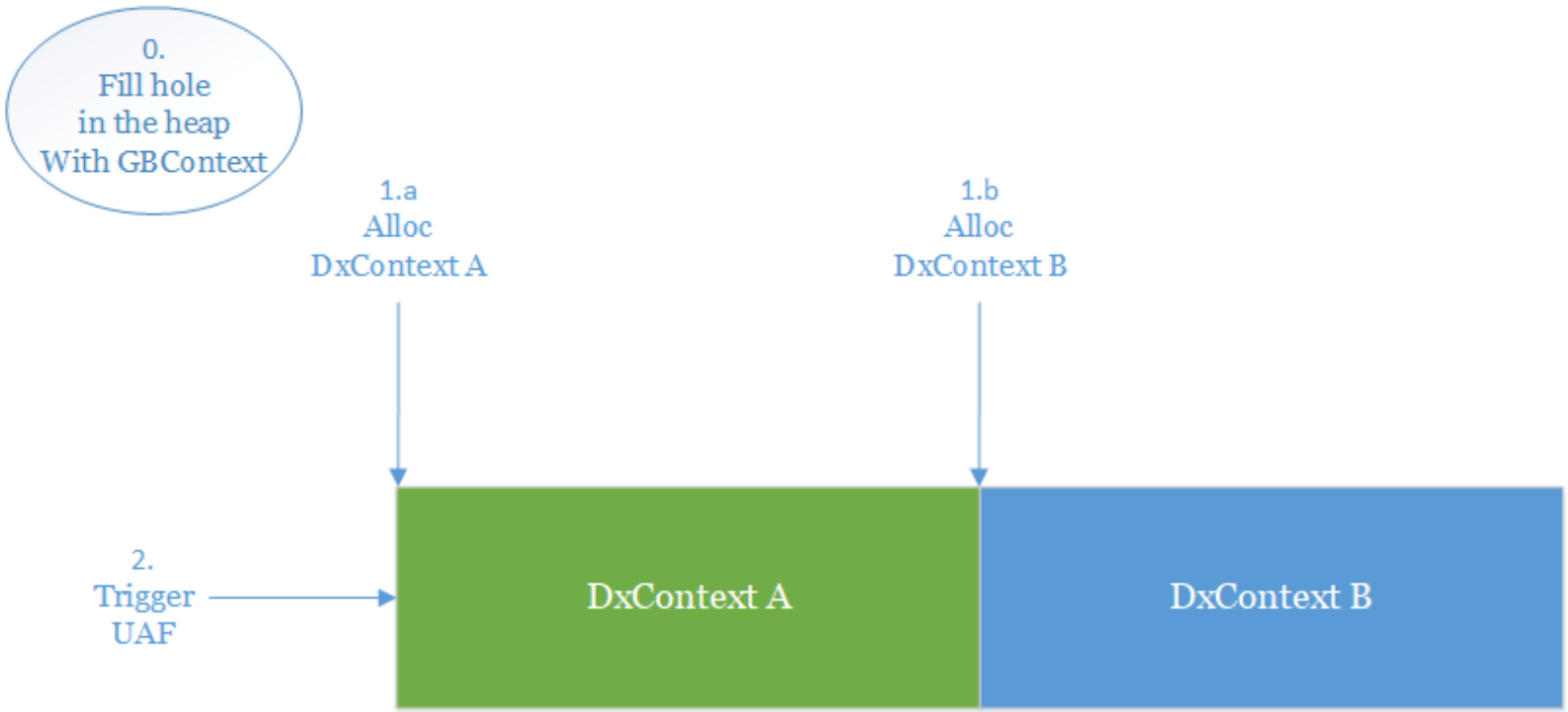
```
cid = DXContextVmxIntern_->cid; // this is the UAF DxContext
        // CID is the Context ID
// [...]
dxCtxt = ArrayId::find_value(&array_dx_context, cid, mask);
// [...]
v17 = dxCtxt->shaderState; // NULL deref if CID not in array
```

- The ID is always stored at the beginning of an object and is chosen by the guest.
- This only means we have to set the ID of the `GbContext` for the re-use to a valid ID for a valid `DxContext`.

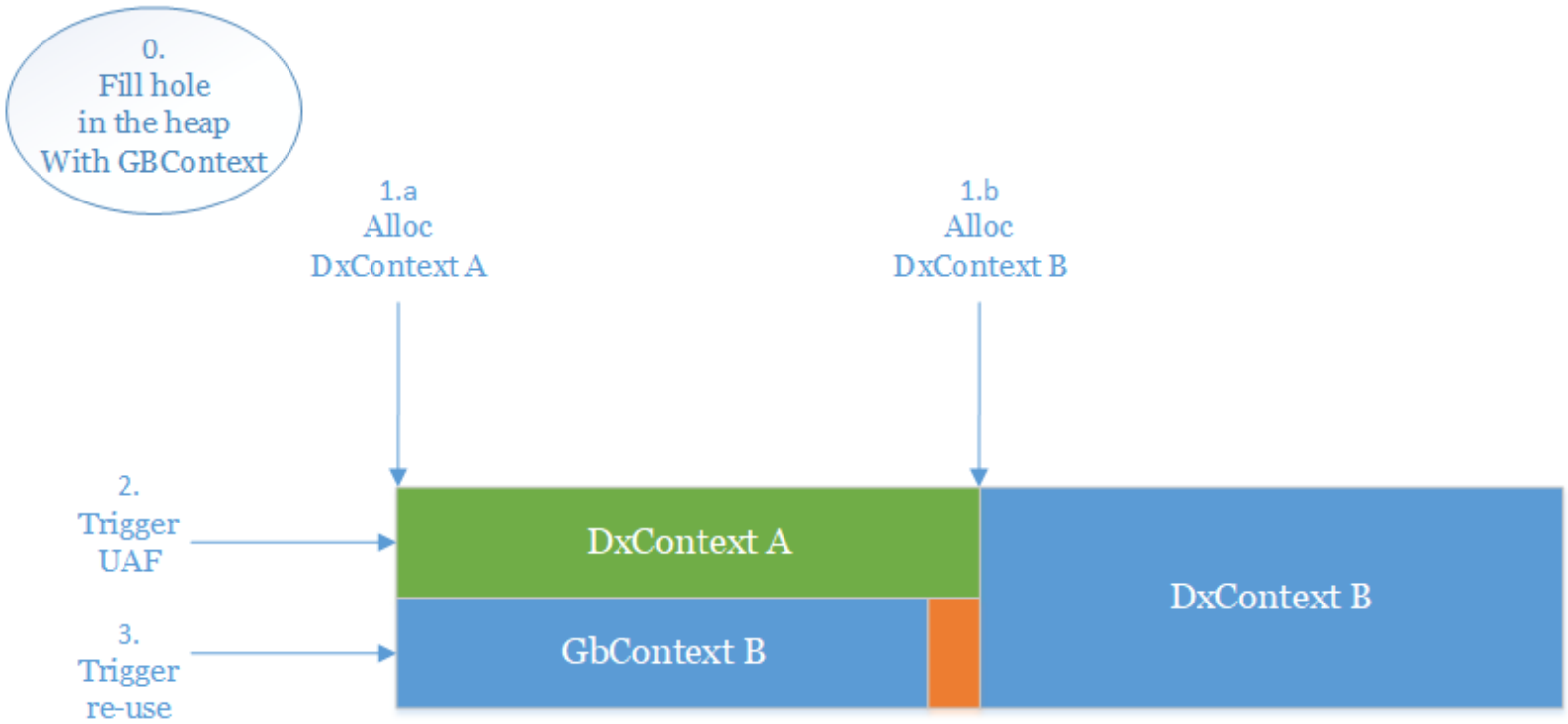
Final steps for the leak



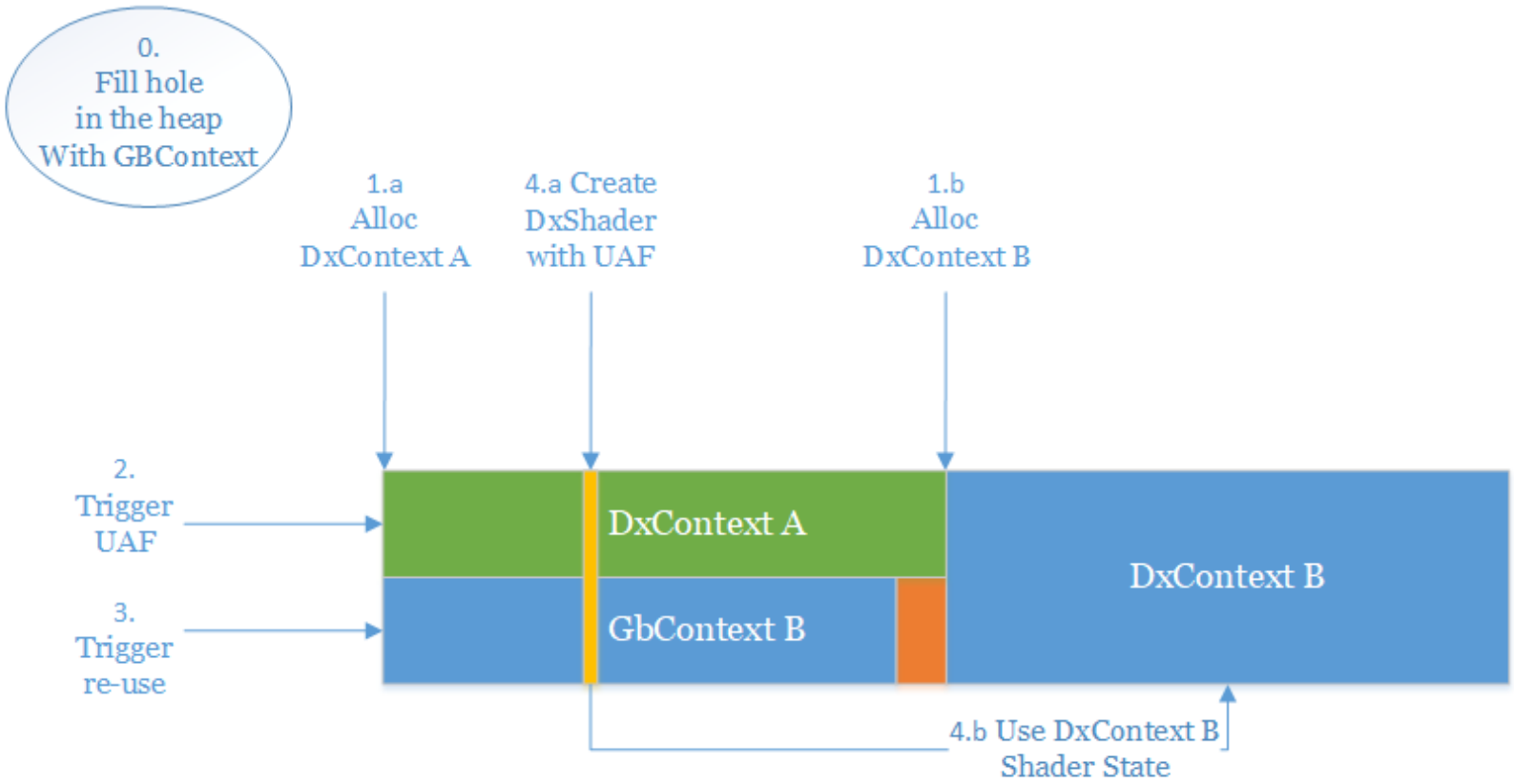
Final steps for the leak



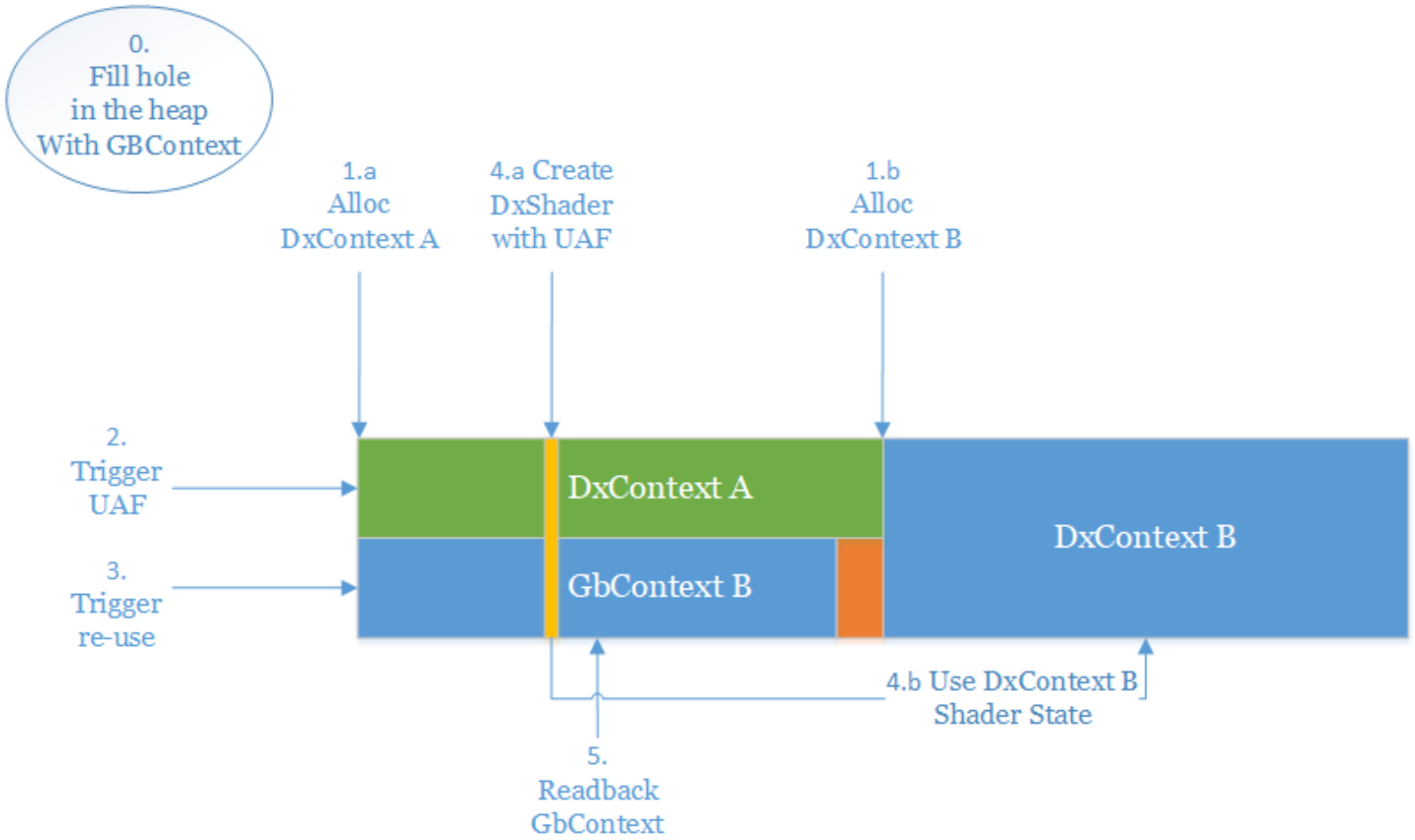
Final steps for the leak



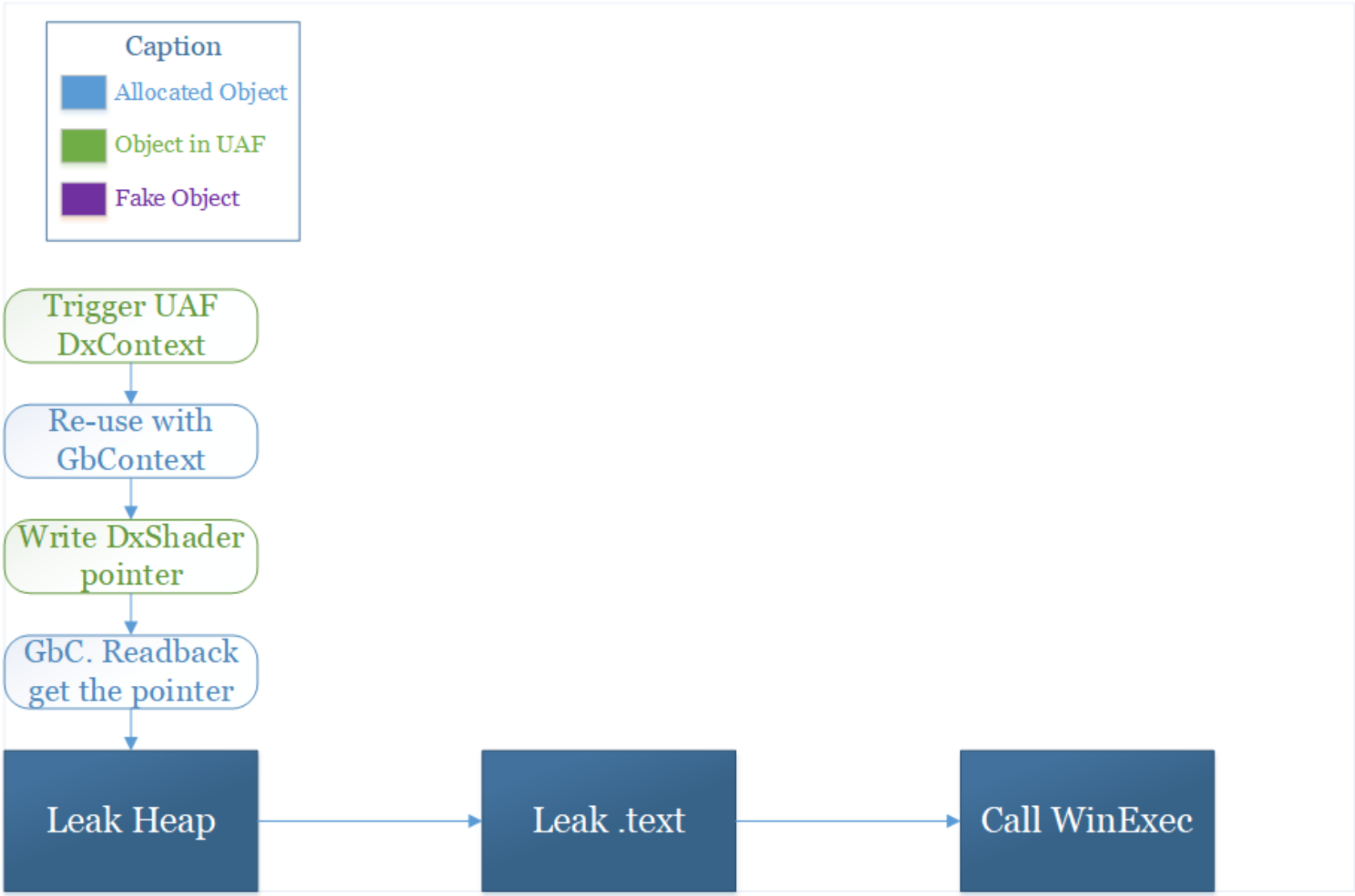
Final steps for the leak



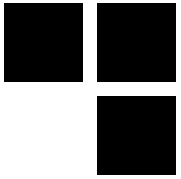
Final steps for the leak



Exploit Steps

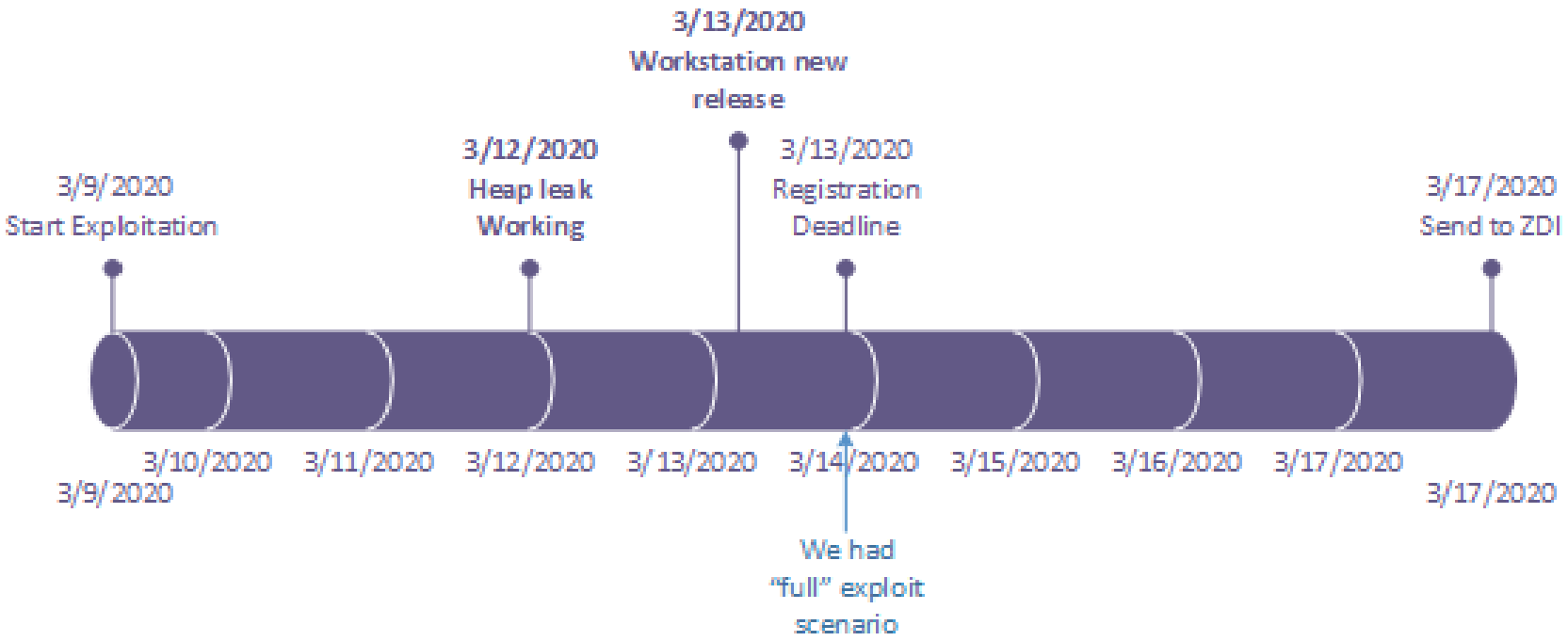


Leak conclusion

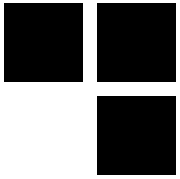


- We successfully triggered a leak on the heap:
  - + Memory can be cleaned: the `DxShader` can be freed by removing the `DxContext B`.
  - + Size is partially controlled: leak in the heap of our choice.
  - + Content is controlled with constraint: SML4 compatible.
  - + Can be triggered several times if needed.
- It uses the only pointer we found on the `DxContext`.
- Only ~6 days left…

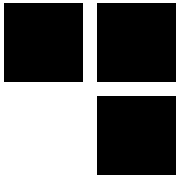
Planning


3/13/2020
Workstation new release
3/12/2020
Heap leak Working
3/13/2020
Registration Deadline
3/9/2020
Start Exploitation
3/17/2020
Send to ZDI
3/9/2020
3/10/2020
3/11/2020
3/12/2020
3/13/2020
3/14/2020
3/15/2020
3/16/2020
3/17/2020
3/17/2020
We had "full" exploit scenario

Plan

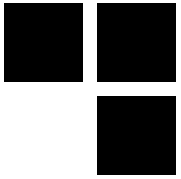


Exploit - Getting a .text leak



- We need a .text leak to control the execution flow.
- We did not find any way to do that using directly the UAF of the `DxContext`.
- So we needed a second stage:
  - We leveraged the `DxContext`'s UAF to reach an arbitrary free primitive.
  - We were able to turn it into another UAF.
  - And then use this UAF to get a .text leak.
- Let's see how.

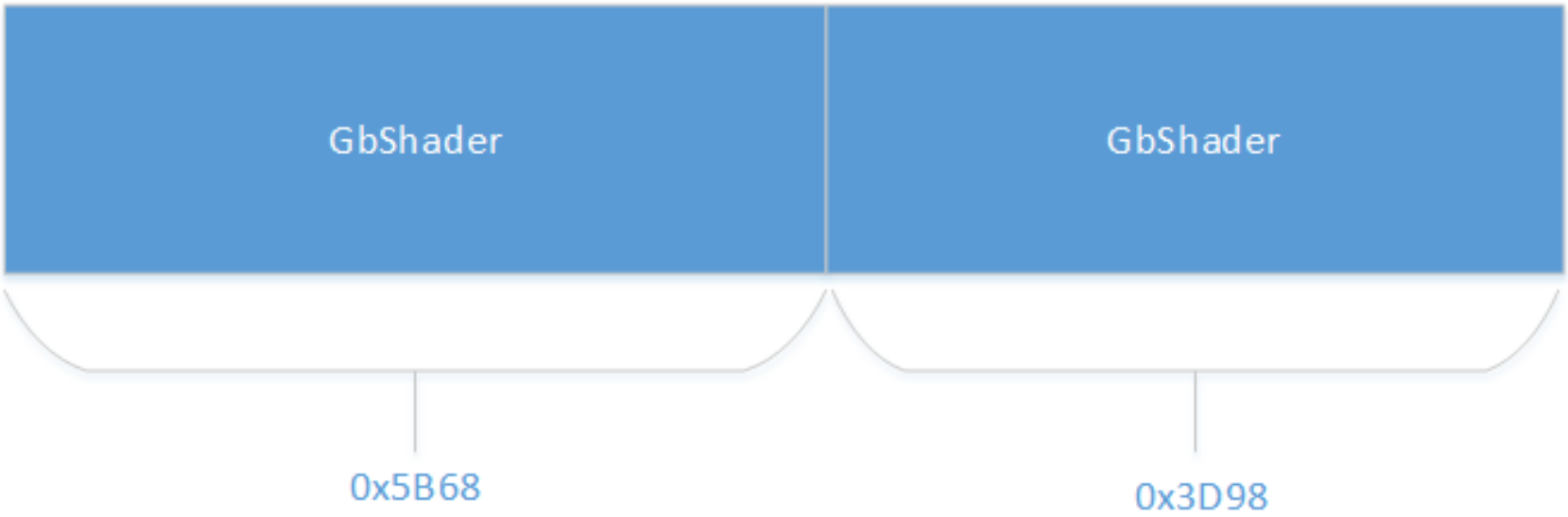
Exploit - Getting an arbitrary free/call primitive



- The `GbContext` can be linked to a Resource Container.
  - Resource Container objects are stored in a global array.
  - The `GbContext` only stores an index in this array, called `RcIndex`, at offset 4.
- We can use the `DxContext`'s UAF to control the `RcIndex` of a `GbContext`.
  - The `set_depthstencil_state` command allows to write the DWORD at offset 0x4464 of the `DxContext`.
  - Use heap massaging to align the chunks.

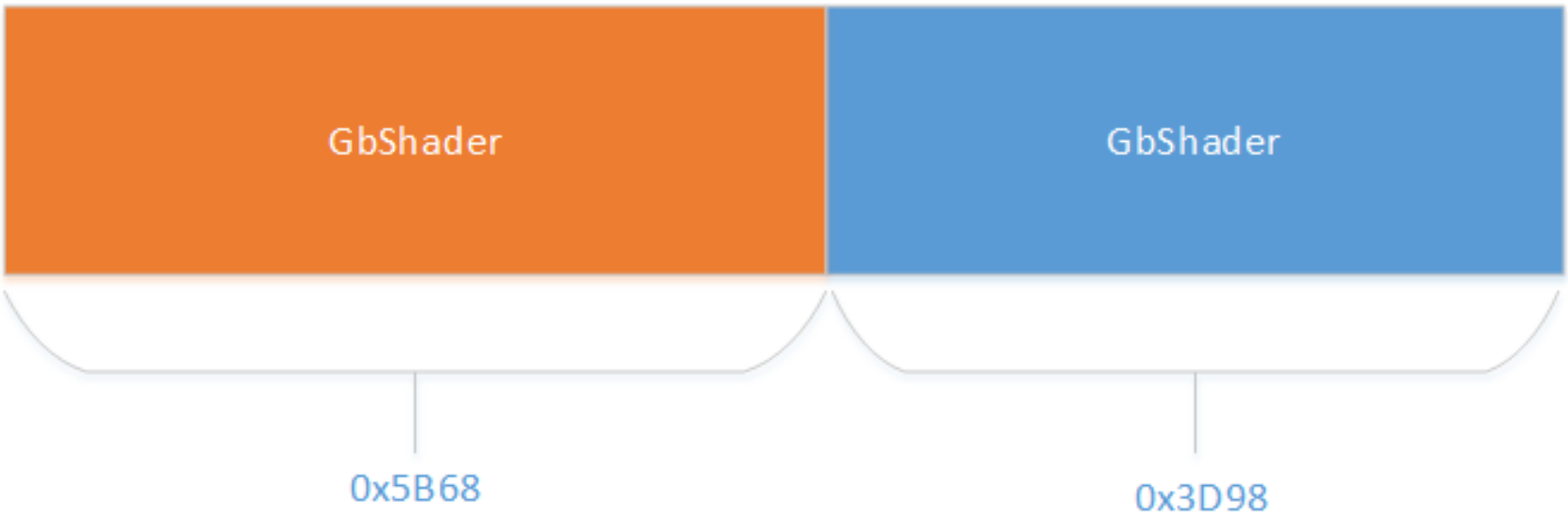
Exploit - Controlling the RcIndex

The idea is to alloc alternatively some chunks of size 0x5B68 (`sizeof(DxContext)`) and 0x3D98.



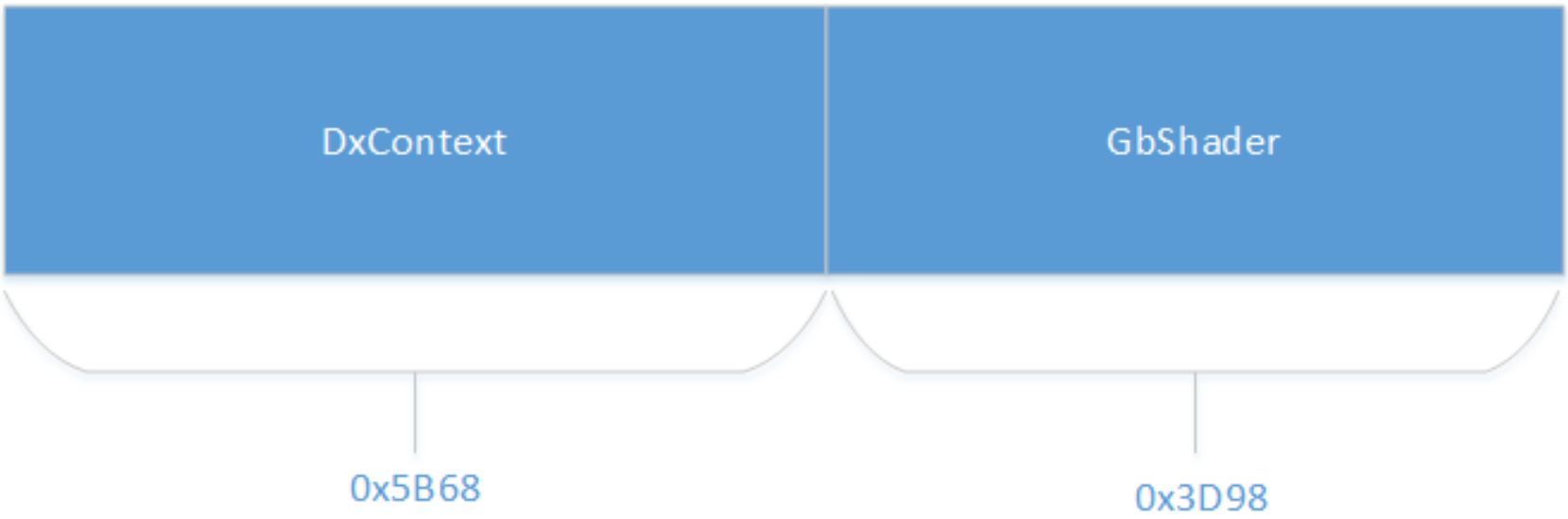
Exploit - Controlling the RcIndex

- Then, one of the 0x5b68 chunk is freed.



Exploit - Controlling the RcIndex

- The `DxContext` is allocated, and fall in the hole.

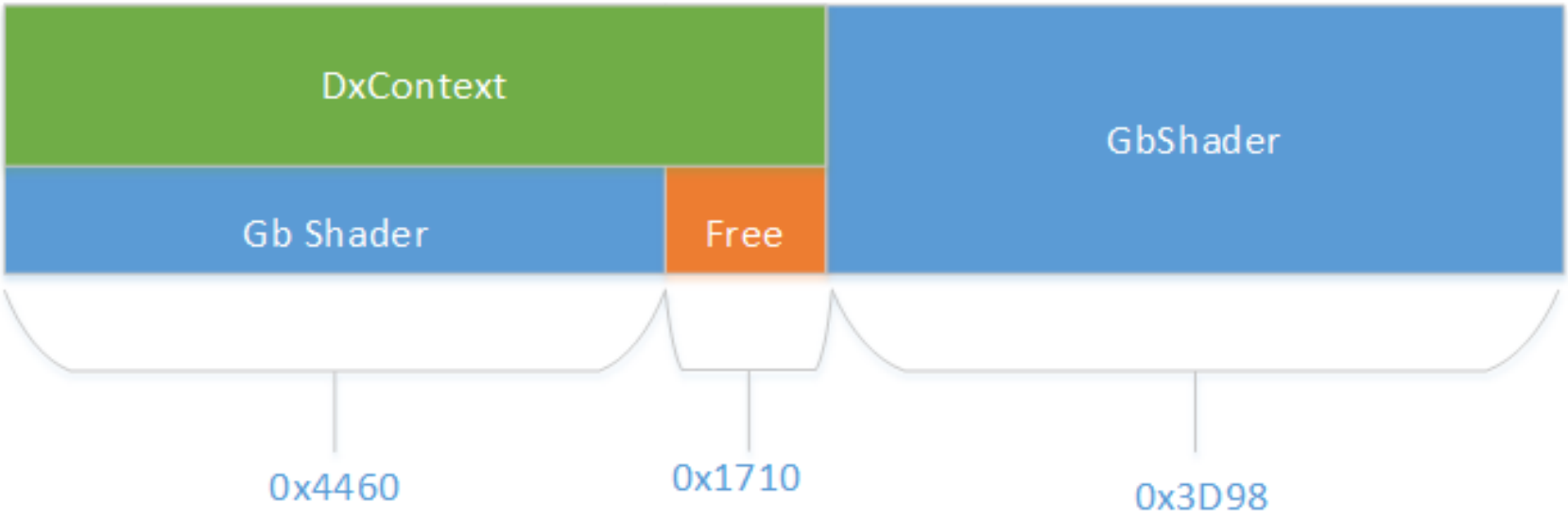


Exploit - Controlling the RcIndex

The vulnerability is triggered, freeing the `DxContext`, but still in UAF, so we can use it.

Exploit - Controlling the RcIndex

Then, allocs of 0x4450 are sprayed. One of them should reuse the hole created by the freeing of the `DxContext`. A hole of size 0x1710 remains behind this new allocation.

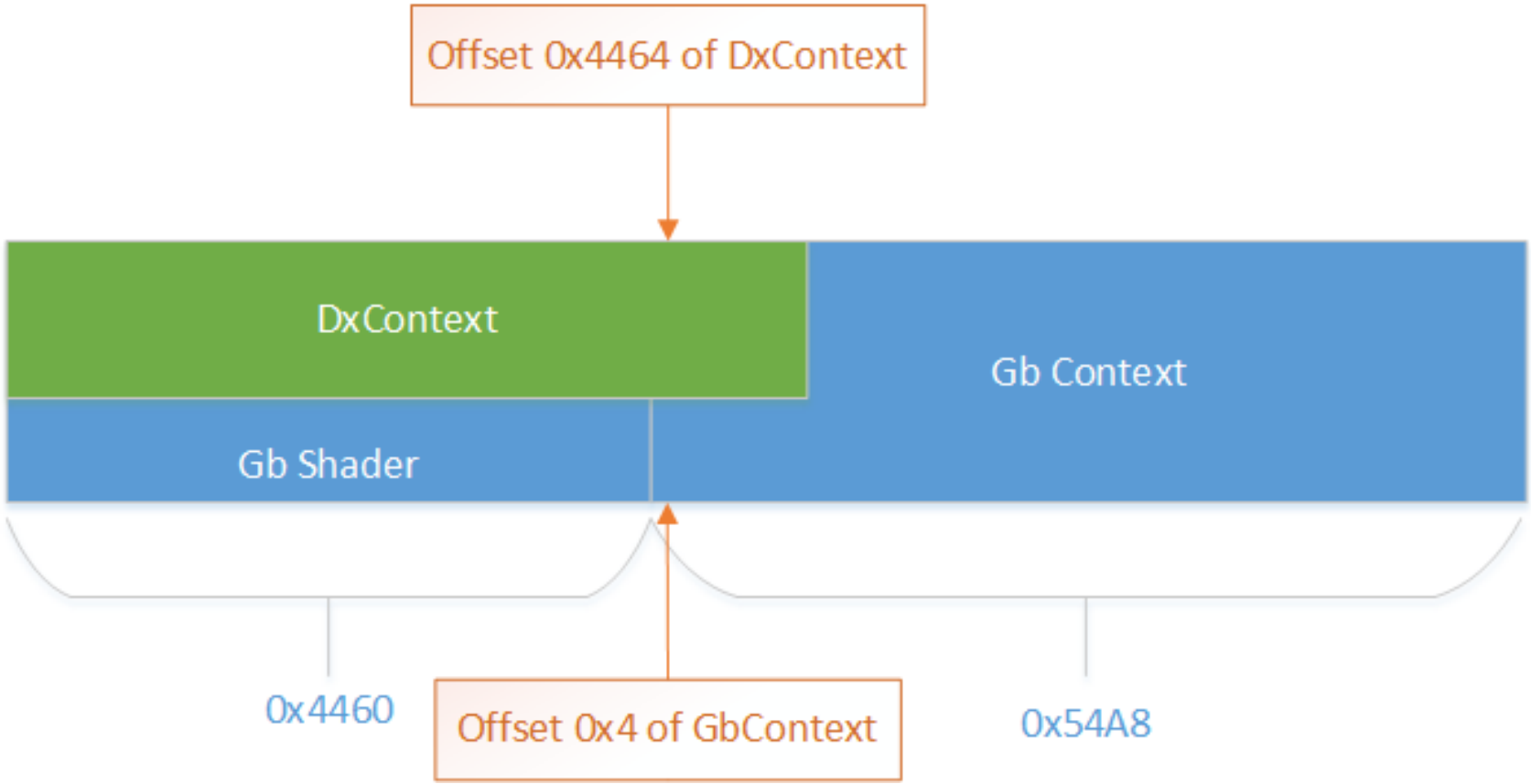


Exploit - Controlling the RcIndex

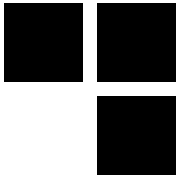
The allocations of 0x3D98 are released, creating a hole of size 0x54A8.

Exploit - Controlling the RcIndex

Finally, a `GbContext` is allocated. Since its size is 0x5490, the place needed with the header is 0x54A0, which falls perfectly with the hole just created.

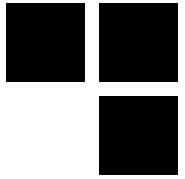


Exploit - Controlling the RcIndex



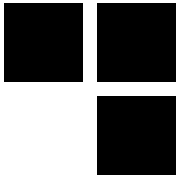
- With this layout, the offset between the start of the `DxContext` and the start of the `GbContext` is precisely 0x4460.
- Allows to rewrite the `RcIndex` of the `GbContext` with the `set_depthstencil_state` command.
- By freeing the altered `GbContext`, we can reach `SVGADestroyGbContextResourceContainer` with a controlled `RcIndex`.

Exploit - Getting an arbitrary free/call primitive



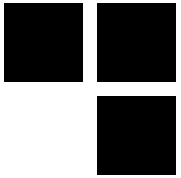
```
void __fastcall SVGADestroyGbContextResourceContainer(int RcIndex)
{
    GbContextResourceContainer * rc = g_GbContextResourceContainers[RcIndex];
    g_GbContextResourceContainers[RcIndex] = 0i64;
    [...]
    SVGACallBackendDestroy(*rc->field_B220);
    MKSMemMgr_free(Shim3DContext, rc); // [4]
}
```

Exploit - Getting an arbitrary free/call primitive



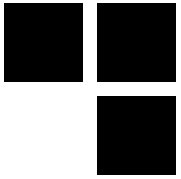
- When freeing the `GbContext`, the Resource Container is fetched and freed.
- Some data in the .data section can be controlled using various commands.
- By controlling the `RcIndex`, the rc pointer can be controlled.
- Using the previous heap leak and some shaders, the content of the rc structure can be fully controlled.

Exploit - Getting an arbitrary free/call primitive



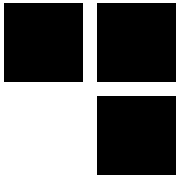
```
void __fastcall SVGACallBackendDestroy(__int64 RcBackend)
{
    counter = 0i64;
    while (counter < RcBackend->nb_callbacks)
    {
          v3 = RcBackend->callback_args[counter] ;
          if ( !v3->called )
              RcBackend->callback_ptr(v3->arg1, v3->arg2); // [1]
          counter++;
          v3->called = 1;
    }
    [...]
    free(RcBackend->callback_args); // [2]
    free(RcBackend); // [3]
}
```

Exploit - Getting an arbitrary free/call primitive



- Reaching `SVGACallBackendDestroy` with a controlled argument provides.
  1. An arbitrary call with 2 controlled arguments.
  2. An arbitrary free.
  3. **RcBackend** is freed, it must point on a valid chunk or it will crash.
  4. **rc** is freed with a special internal heap:
     - needs to fake a header for the internal heap,
     - had to free a real chunk to avoid crashing.
- Can't use the arbitrary call for now, we need a leak of the address of a function to call.

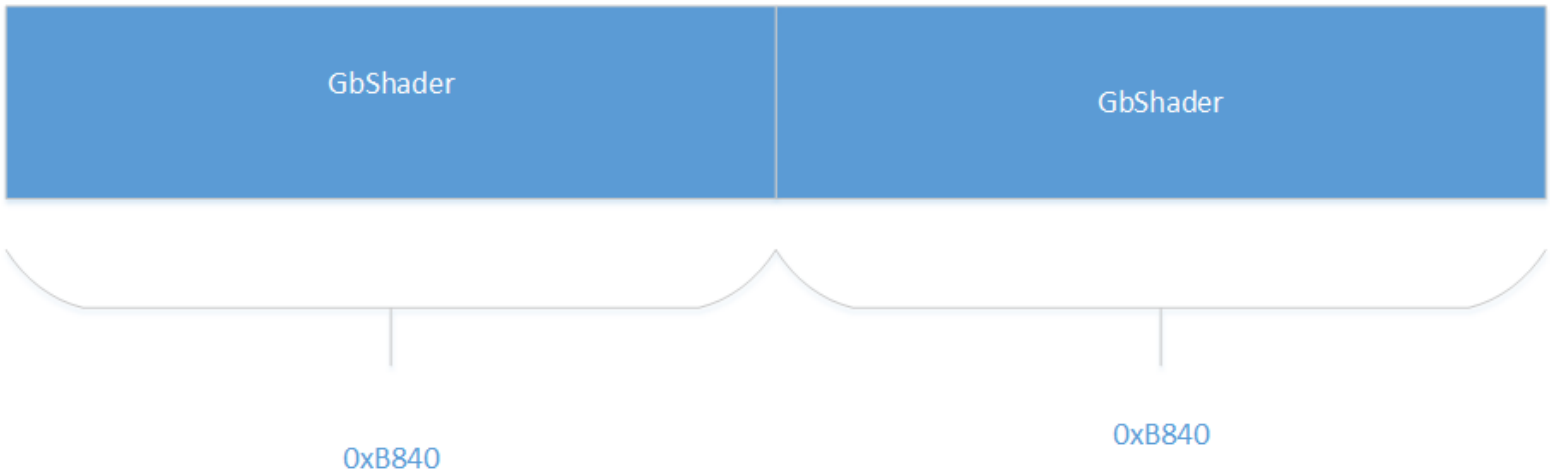
Exploit - Arbitrary free to .text leak



- Use the arbitrary free to put a `GbContext` in UAF.
  - Can already be done using the heap leak and some massaging.
- Overlap the `GbContext` with an object containing a function pointer.
  - The `GbContextResourceContainer` is the actual object supposed to be linked to the `GbContext` via the `RcIndex`.
  - Size of 0xB7E0, does not fall into LFH.
  - Contains a function pointer at offset 0xB7D8.
- Readback the `GbContext` to get the function pointer.

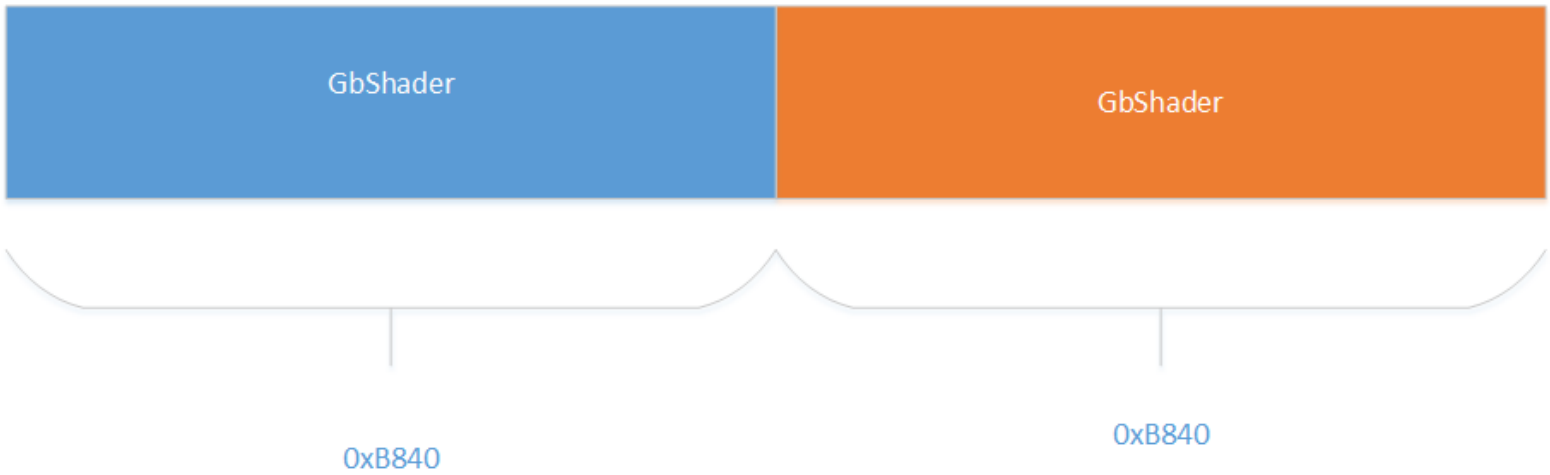
Exploit - Arbitrary free to .text leak

- First, blocks of 0xB840 are sprayed.

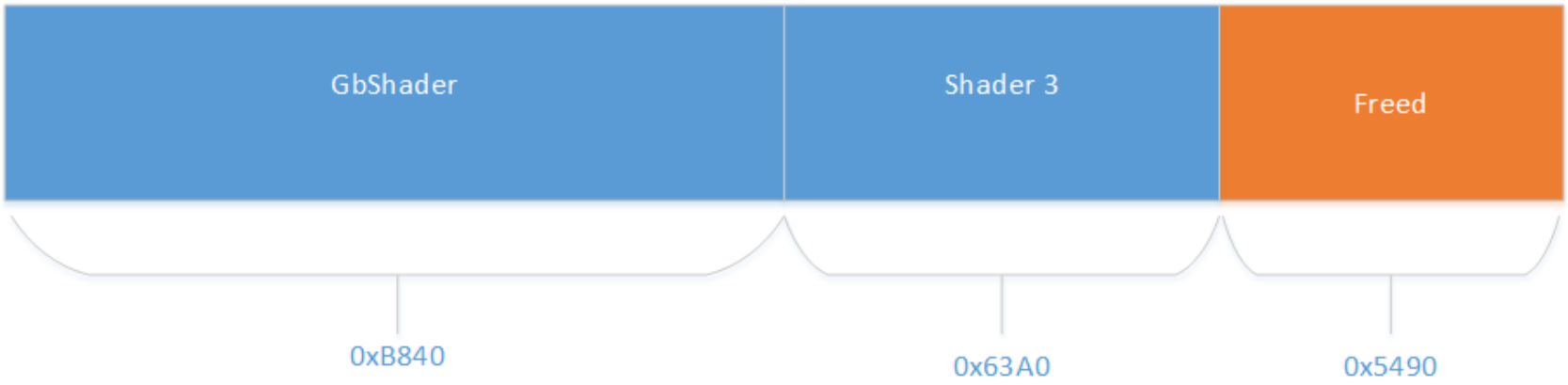


Exploit - Arbitrary free to .text leak

- One of them is freed, and some allocation of size 0xB840 - sizeof(GbContext) = 0x63A0 are sprayed.
- Split the 0xB840 into two chunks, one of 0x63A0, and one of 0x5490.

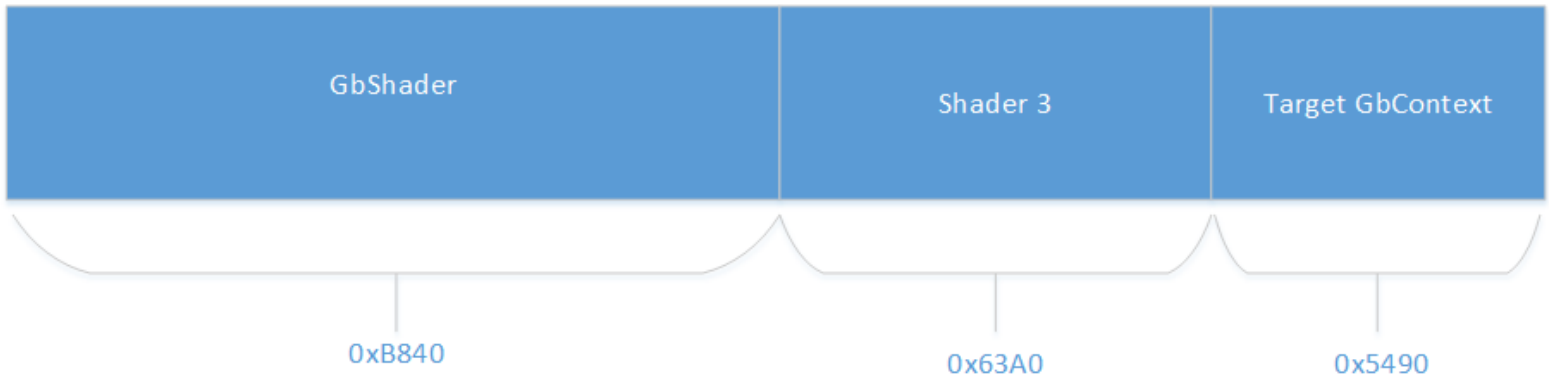


Exploit - Arbitrary free to .text leak

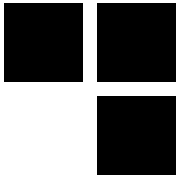


Exploit - Arbitrary free to .text leak

- After this, we should have a hole of 0x5A90, which is the size of a `GbContext`. We allocate a `GbContext`, that will fall into this hole.

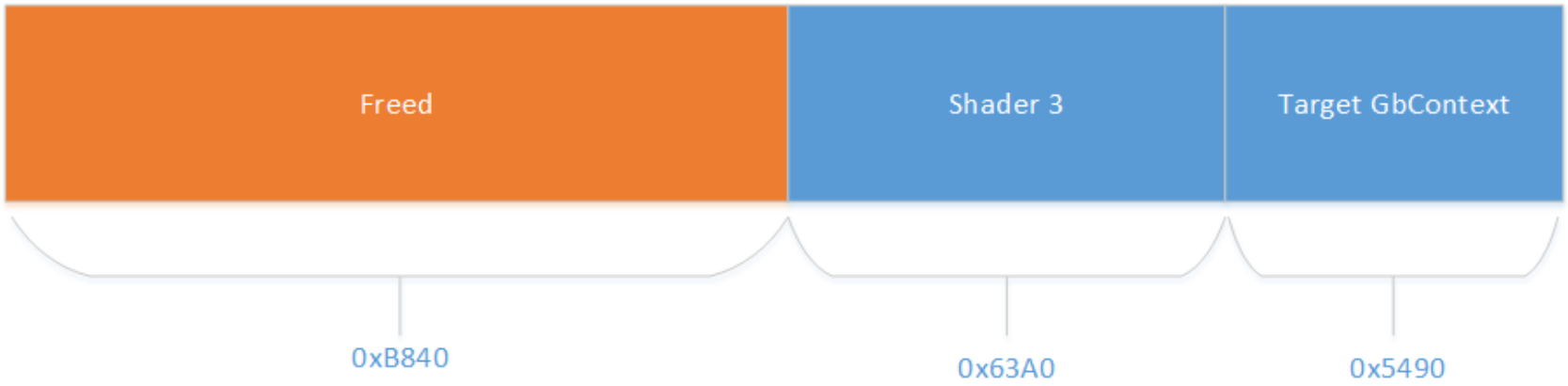


Exploit - Arbitrary free to .text leak

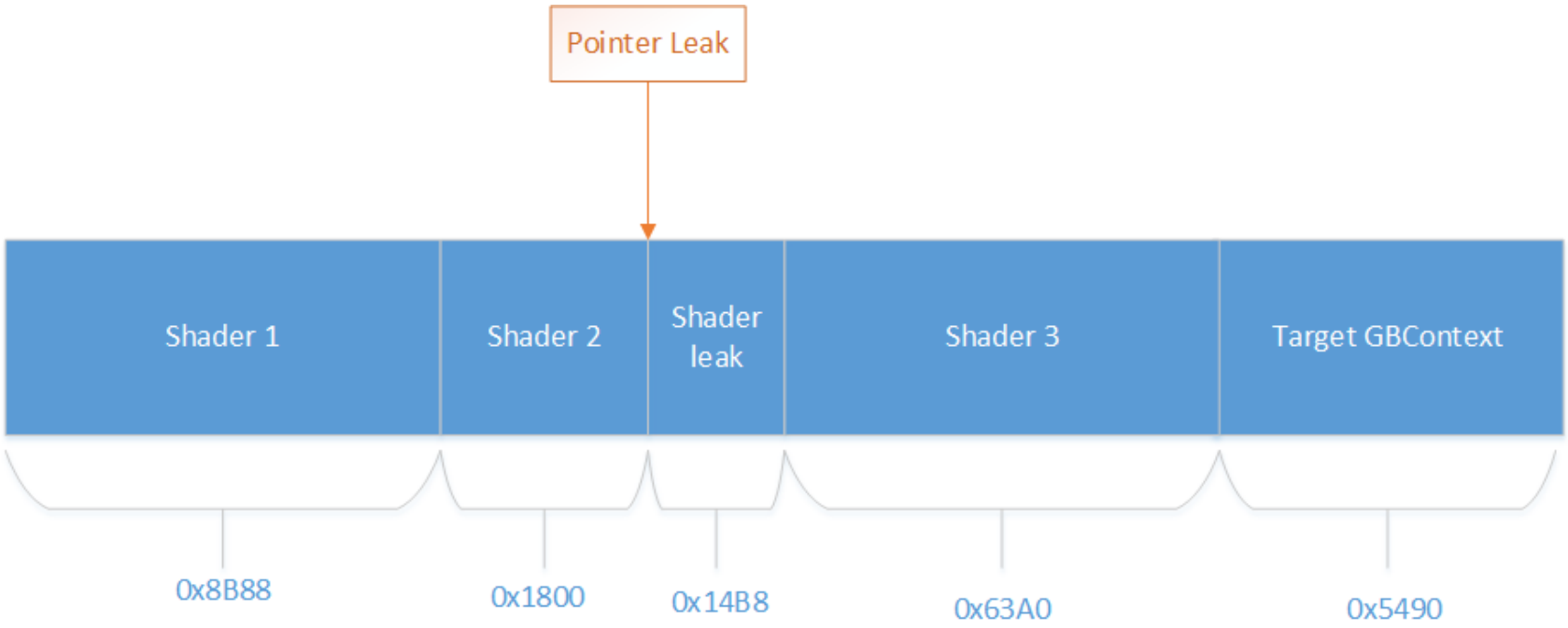


- The allocation of 0xB840 that followed the first we freed is also freed.
- Hopefully, the allocation will be just in front the setup we just did.
- This freed allocation will be split up in three different chunks:
  - One allocated chunk of size 0x8b88, called **Shader1** ;
  - One allocated chunk of size 0x1800, called **Shader2** ;
  - One freed chunk of size 0x14b8, called **ShaderLeaked**.
- Use the free chunk of 0x14b8 do allocate a `DxShader` and leak it's address.
- The address of this setup is known.

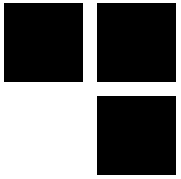
Exploit - Arbitrary free to .text leak



Exploit - Arbitrary free to .text leak

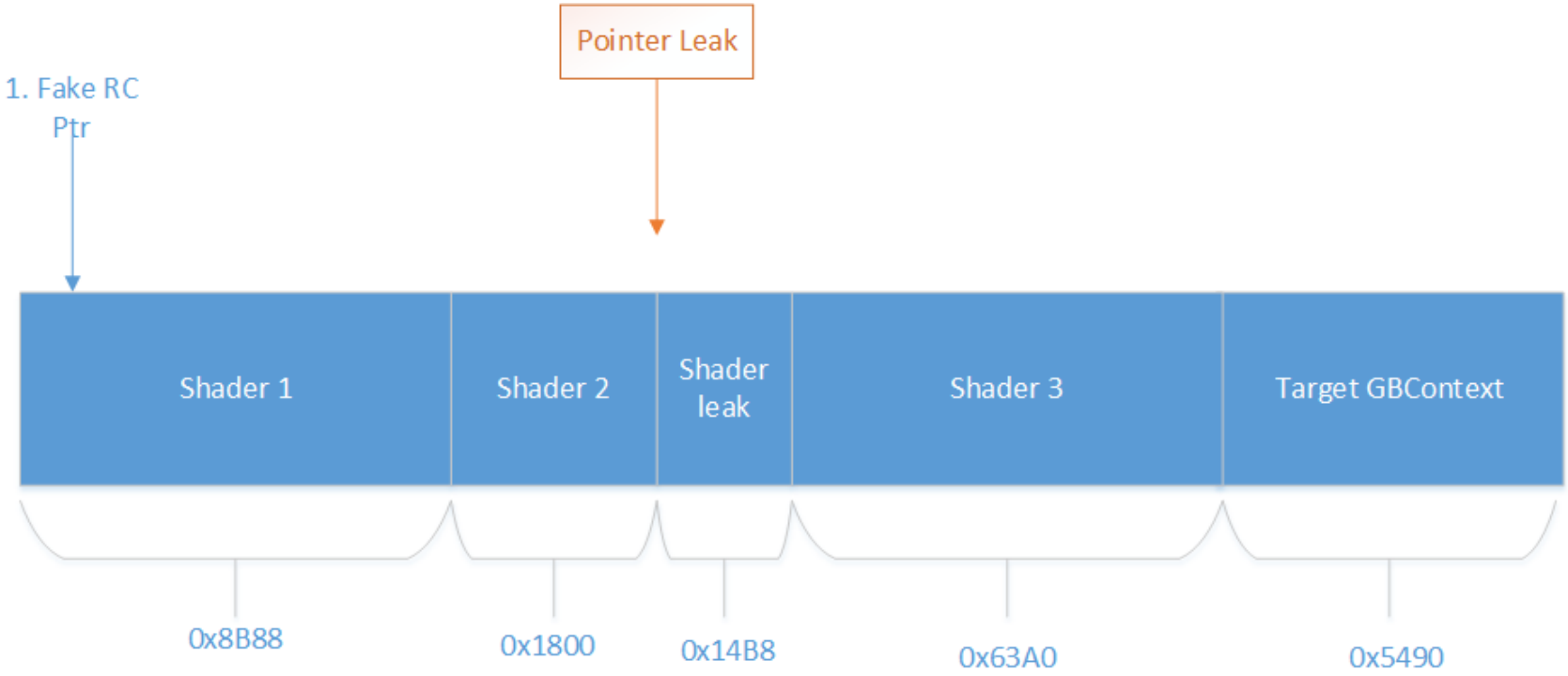


Exploit - Arbitrary free to .text leak

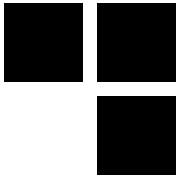


- Build a fake `GbContextResourceContainer` into the **Shader1**.
- The rc + 0xB220 will point in the middle of the **ShaderLeaked**.
- The pointer stored there will point to the beginning of the **ShaderLeaked**.
- Trigger the arbitrary free.

Exploit - Arbitrary free to .text leak

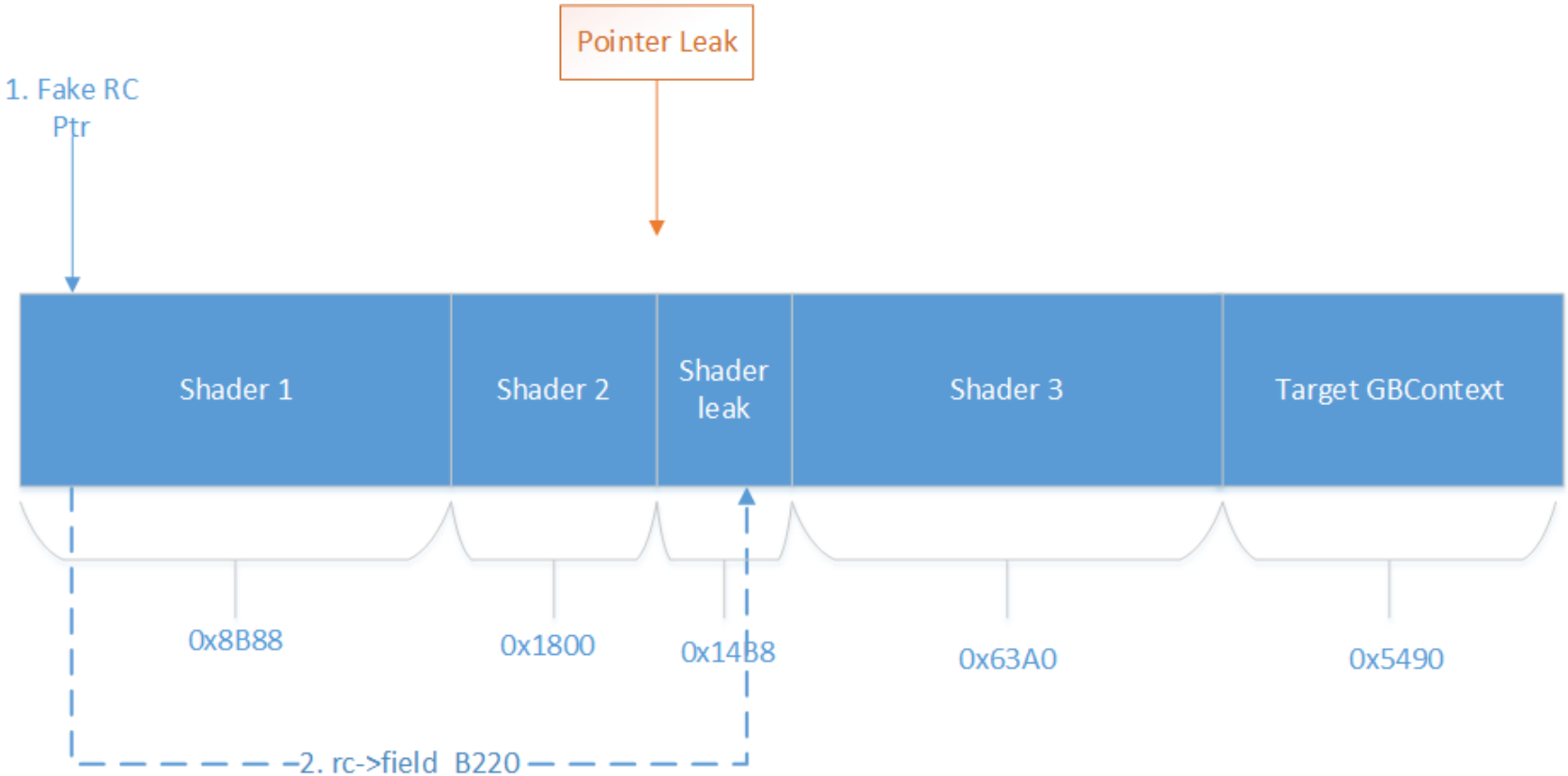


Exploit - Getting an arbitrary free/call primitive

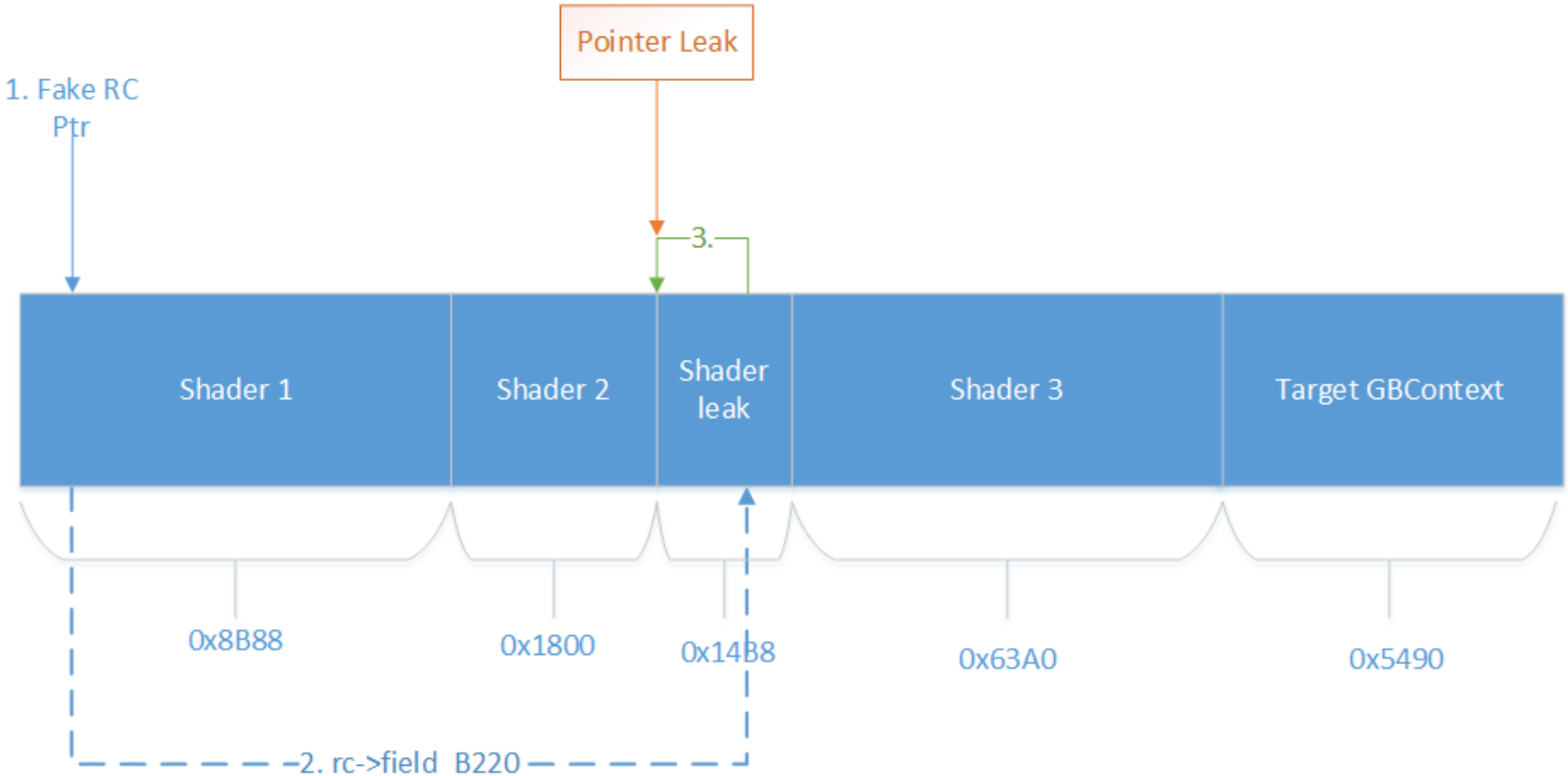


```
void __fastcall SVGADestroyGbContextResourceContainer(int RcIndex)
{
    GbContextResourceContainer * rc = g_GbContextResourceContainers[RcIndex];
    g_GbContextResourceContainers[RcIndex] = 0i64;
    [...]
    SVGACallBackendDestroy(*rc->field_B220); // Goes down this path first
    MKSMemMgr_free(Shim3DContext, rc);
}
```

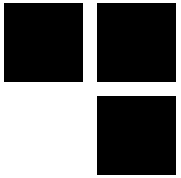
Exploit - Arbitrary free to .text leak



Exploit - Arbitrary free to .text leak

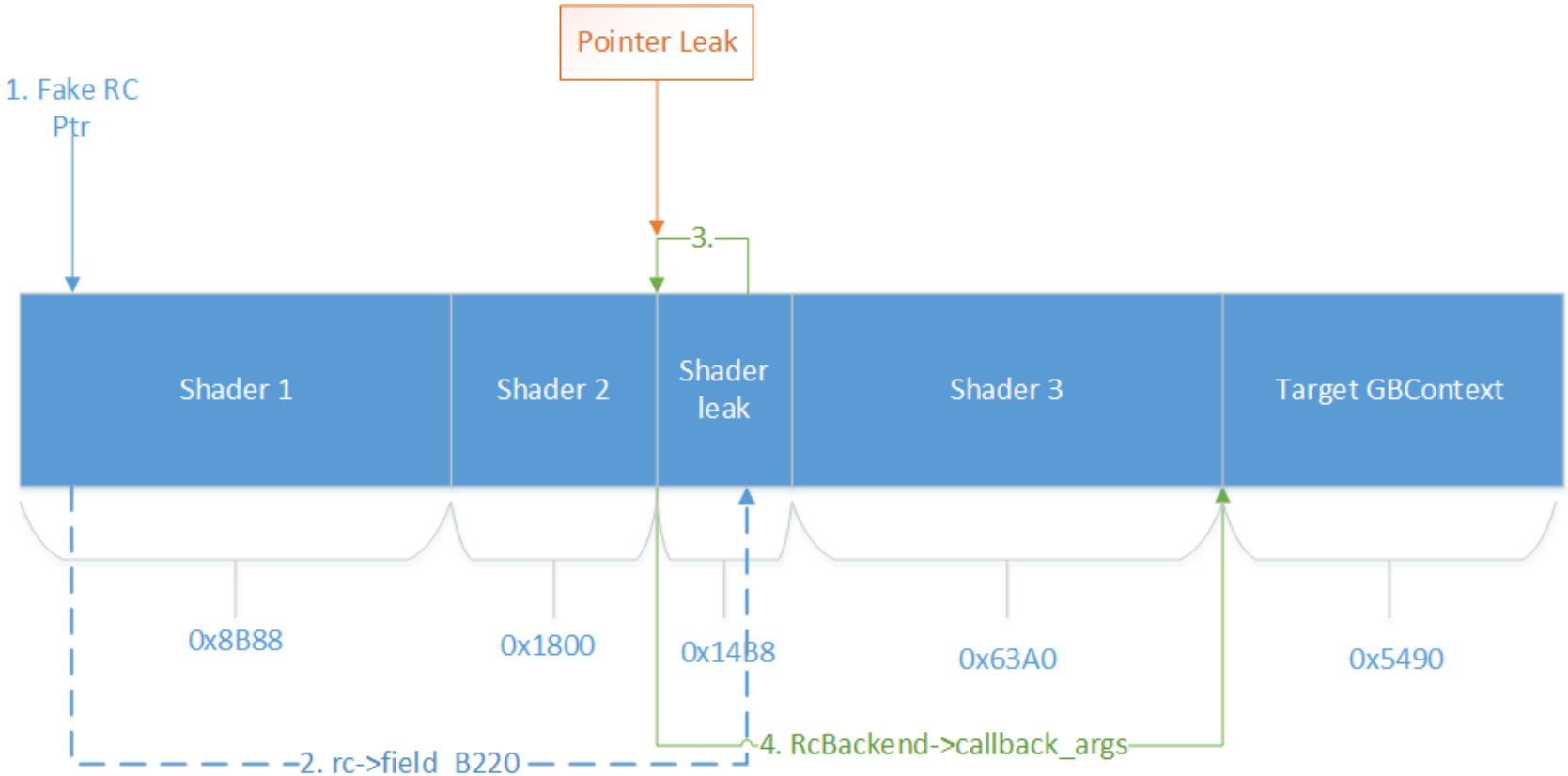


Exploit - Getting an arbitrary free/call primitive

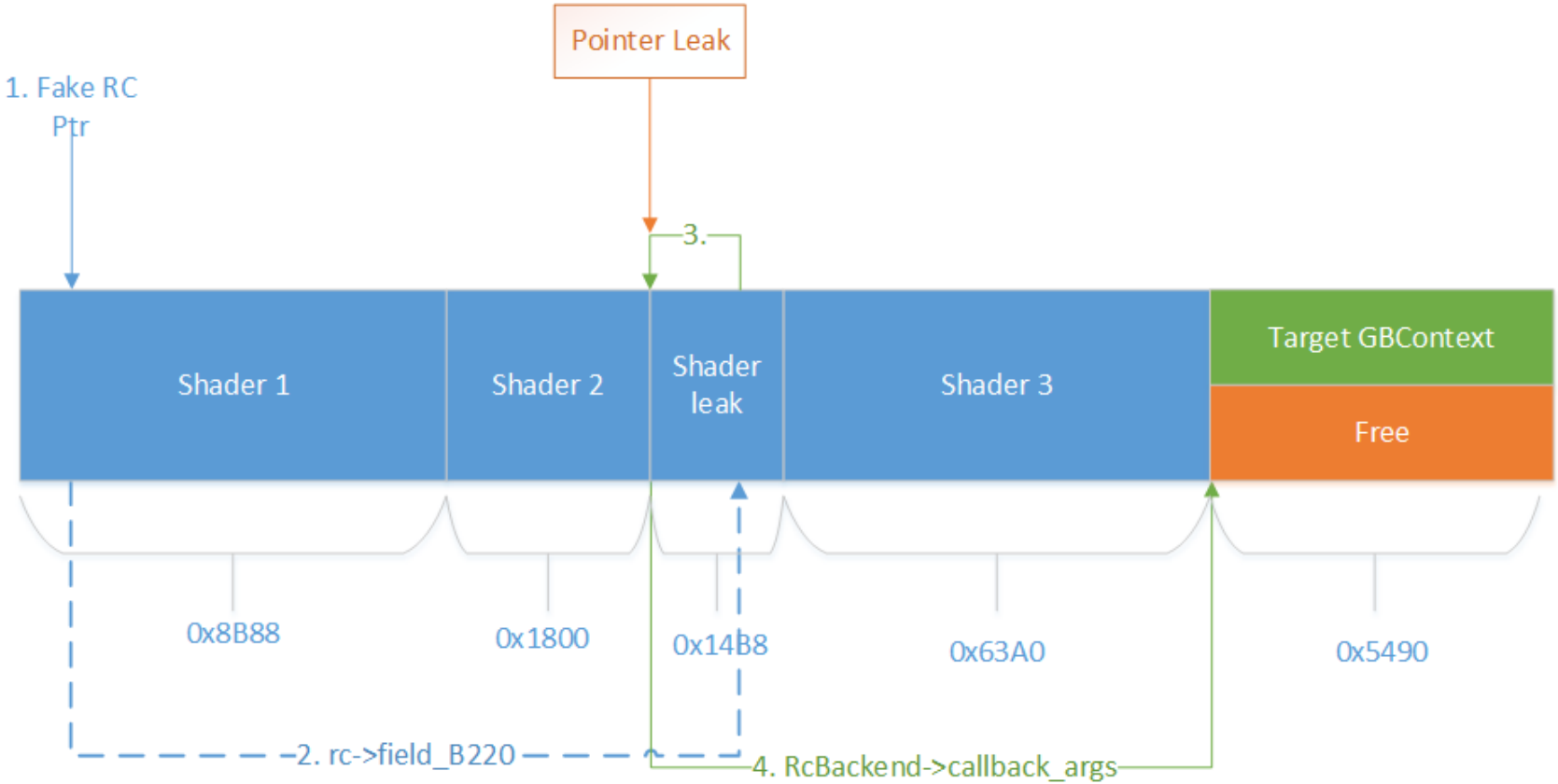


```
void __fastcall SVGACallBackendDestroy(__int64 RcBackend)
{
    counter = 0i64;
    while (counter < RcBackend->nb_callbacks)
    {
          v3 = RcBackend->callback_args[counter] ;
          if ( !v3->called )
              RcBackend->callback_ptr(v3->arg1, v3->arg2);
          counter++;
          v3->called = 1;
    }
    [...]
    free(RcBackend->callback_args); // Frees TargetGbContext
    free(RcBackend); // Frees ShaderLeaked
}
```

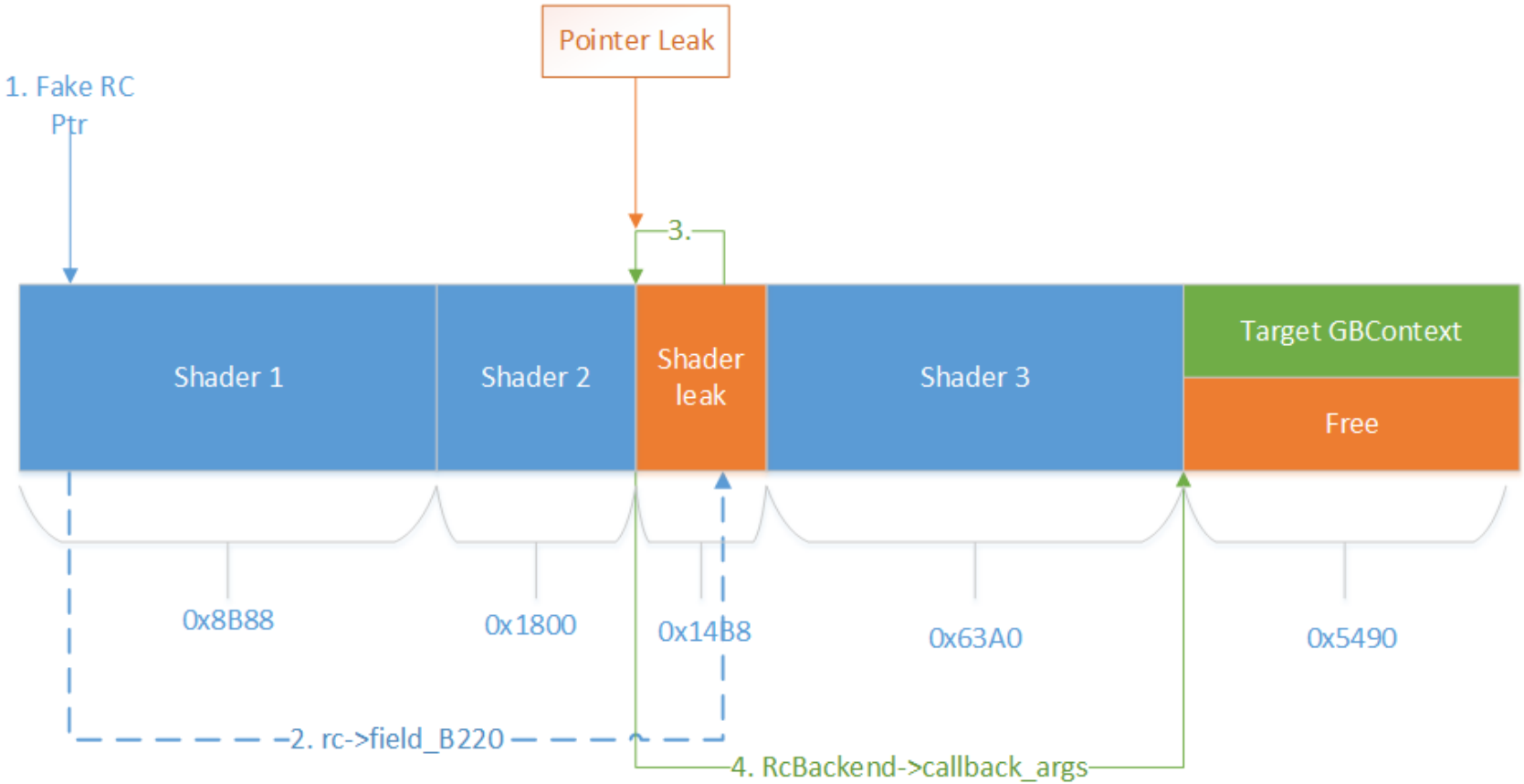
Exploit - Arbitrary free to .text leak



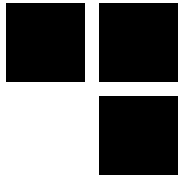
Exploit - Arbitrary free to .text leak



Exploit - Arbitrary free to .text leak

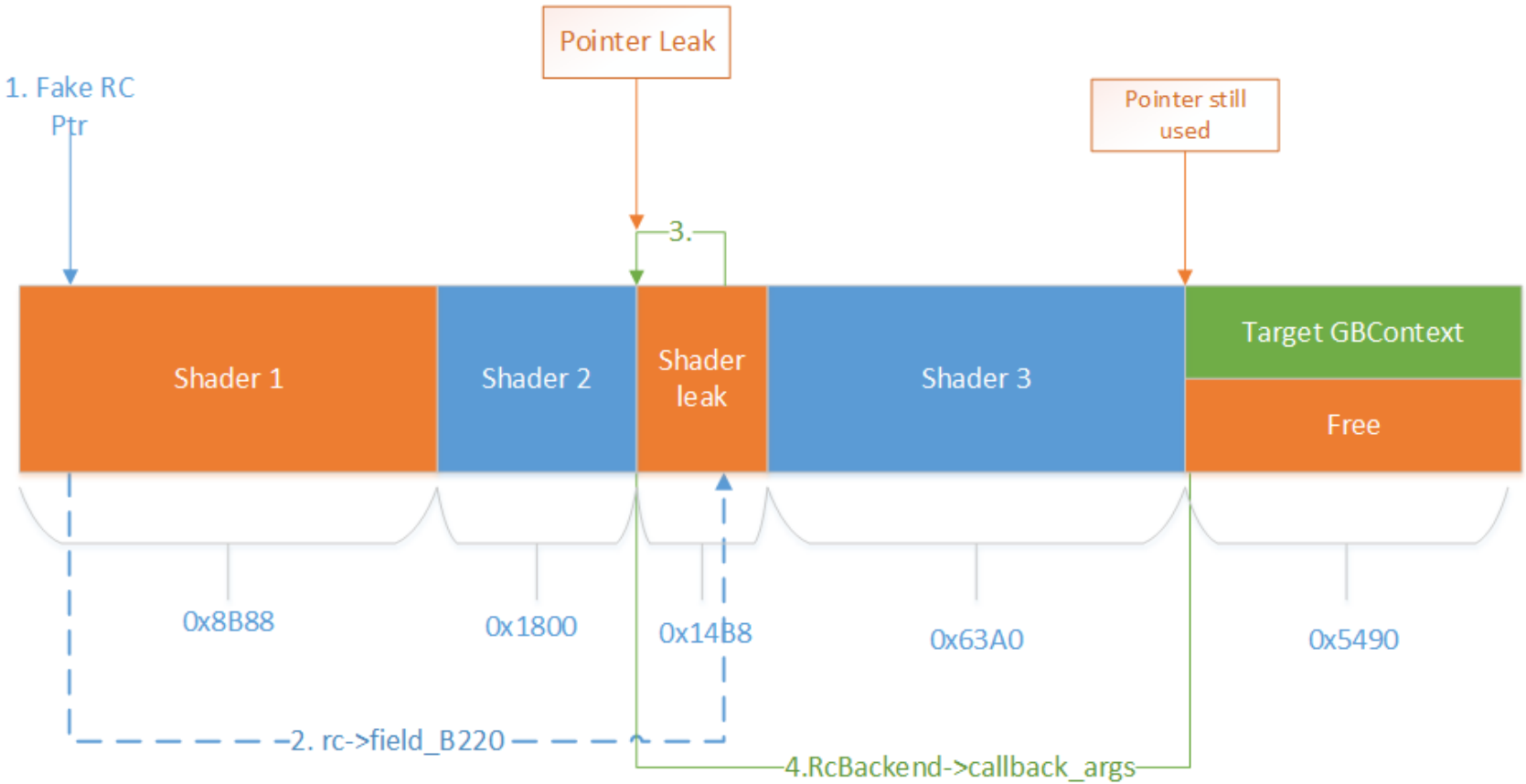


Exploit - Getting an arbitrary free/call primitive

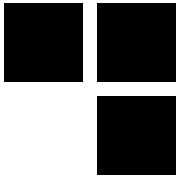


```
void __fastcall SVGADestroyGbContextResourceContainer(int RcIndex)
{
    GbContextResourceContainer * rc = g_GbContextResourceContainers[RcIndex];
    g_GbContextResourceContainers[RcIndex] = 0i64;
    [...]
    SVGACallBackendDestroy(*rc->field_B220);
    MKSMemMgr_free(Shim3DContext, rc); // Frees shader 1
}
```

Exploit - Arbitrary free to .text leak



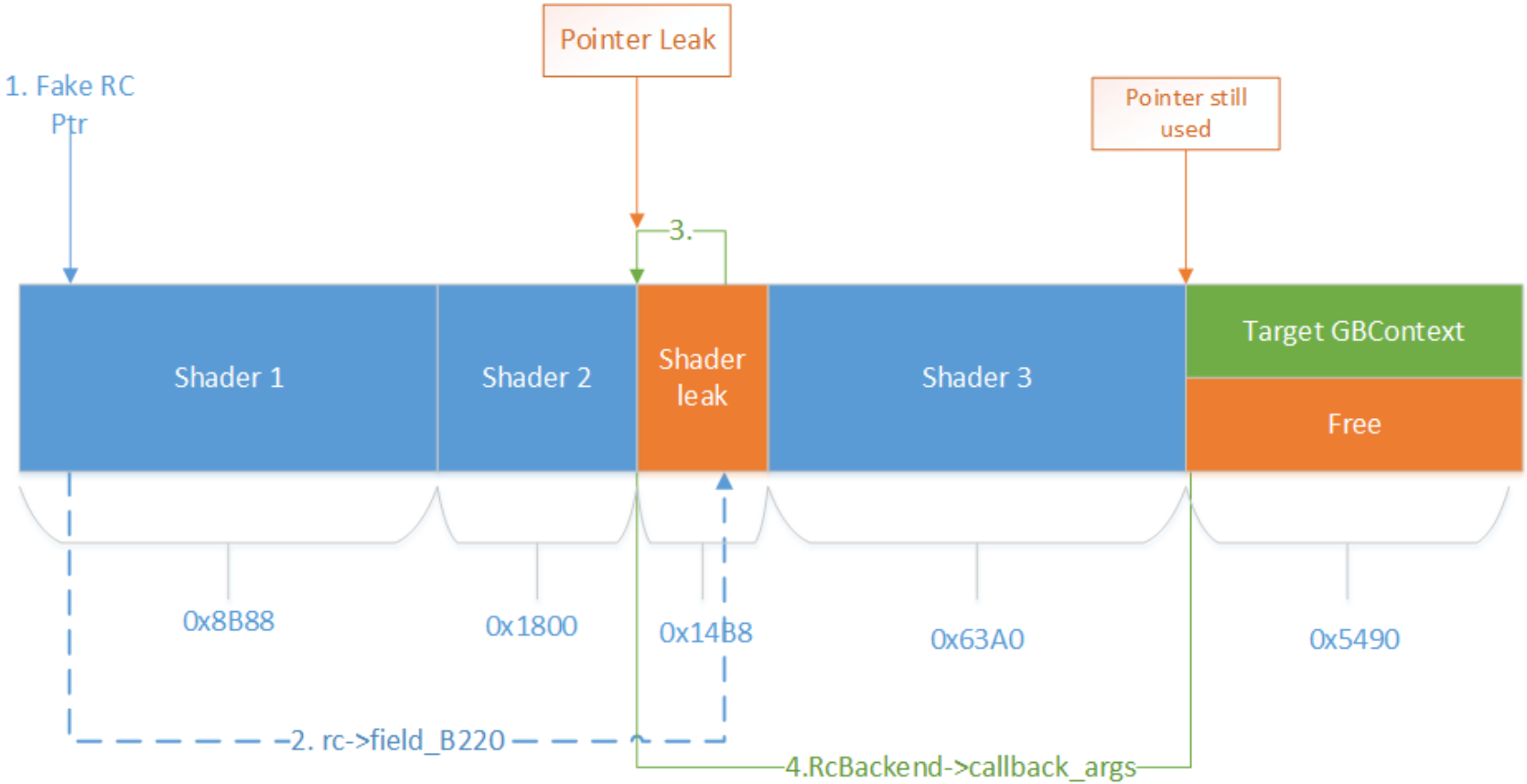
Exploit - Arbitrary free to .text leak



- **Shader1**, **ShaderLeaked**, and **TargetGbContext** are in UAF.
  - It's not a problem to have shaders in UAF as it won't crash.
- Shader 2 is necessary here to avoid the coalescing of **Shader1** and **ShaderLeaked**.
- The `GbContext` with the controlled `RcIndex` is freed too.

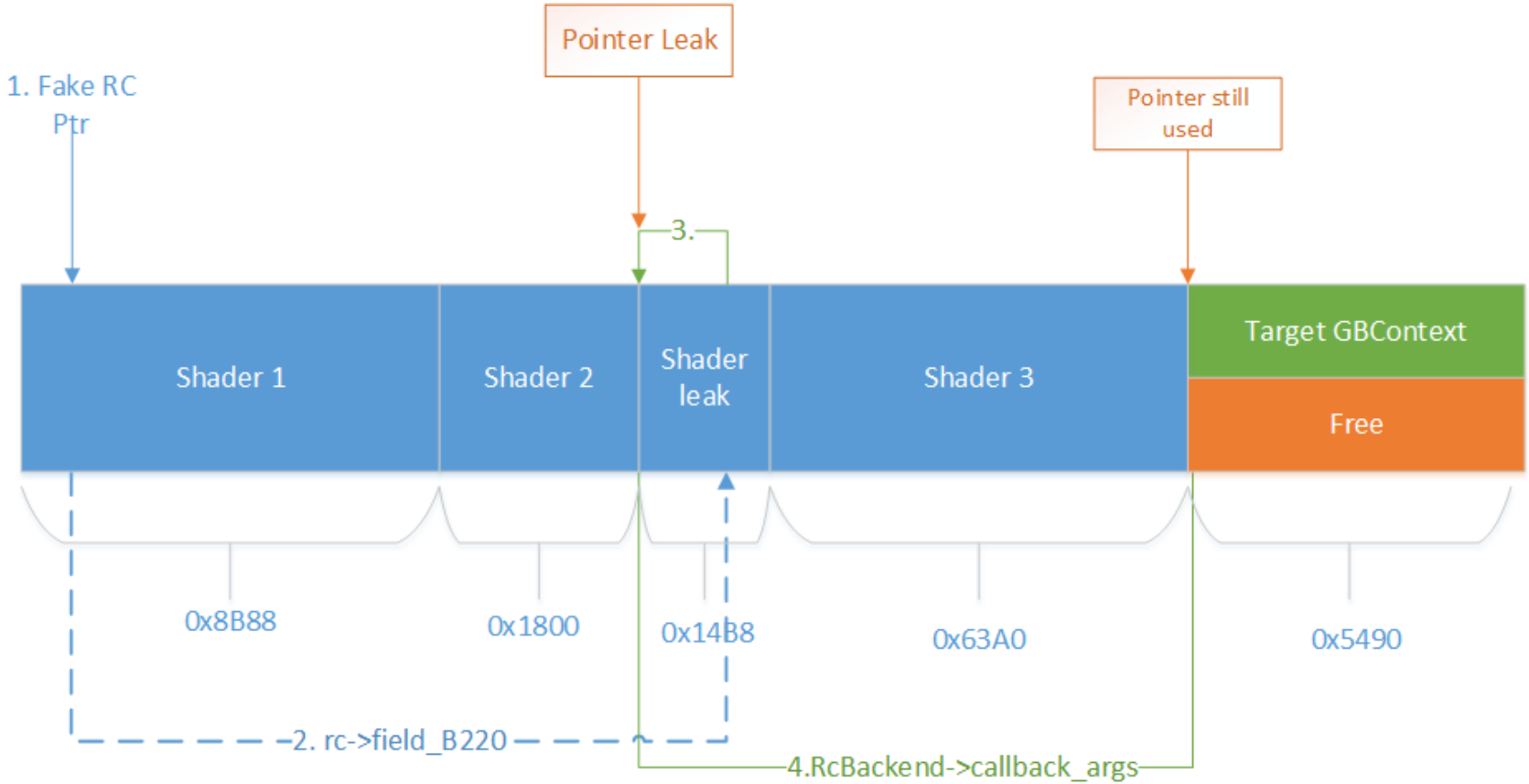
Exploit - Arbitrary free to .text leak

Reallocate the shader that were freed.

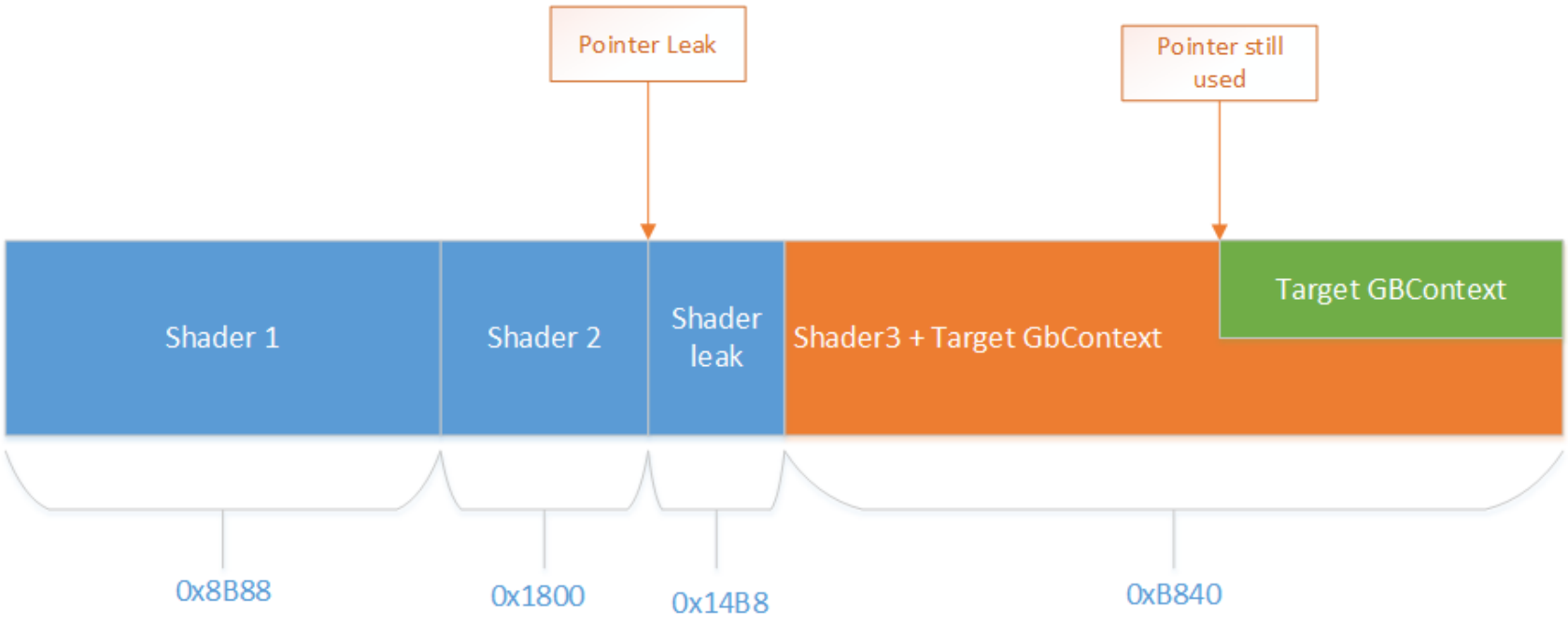


Exploit - Arbitrary free to .text leak

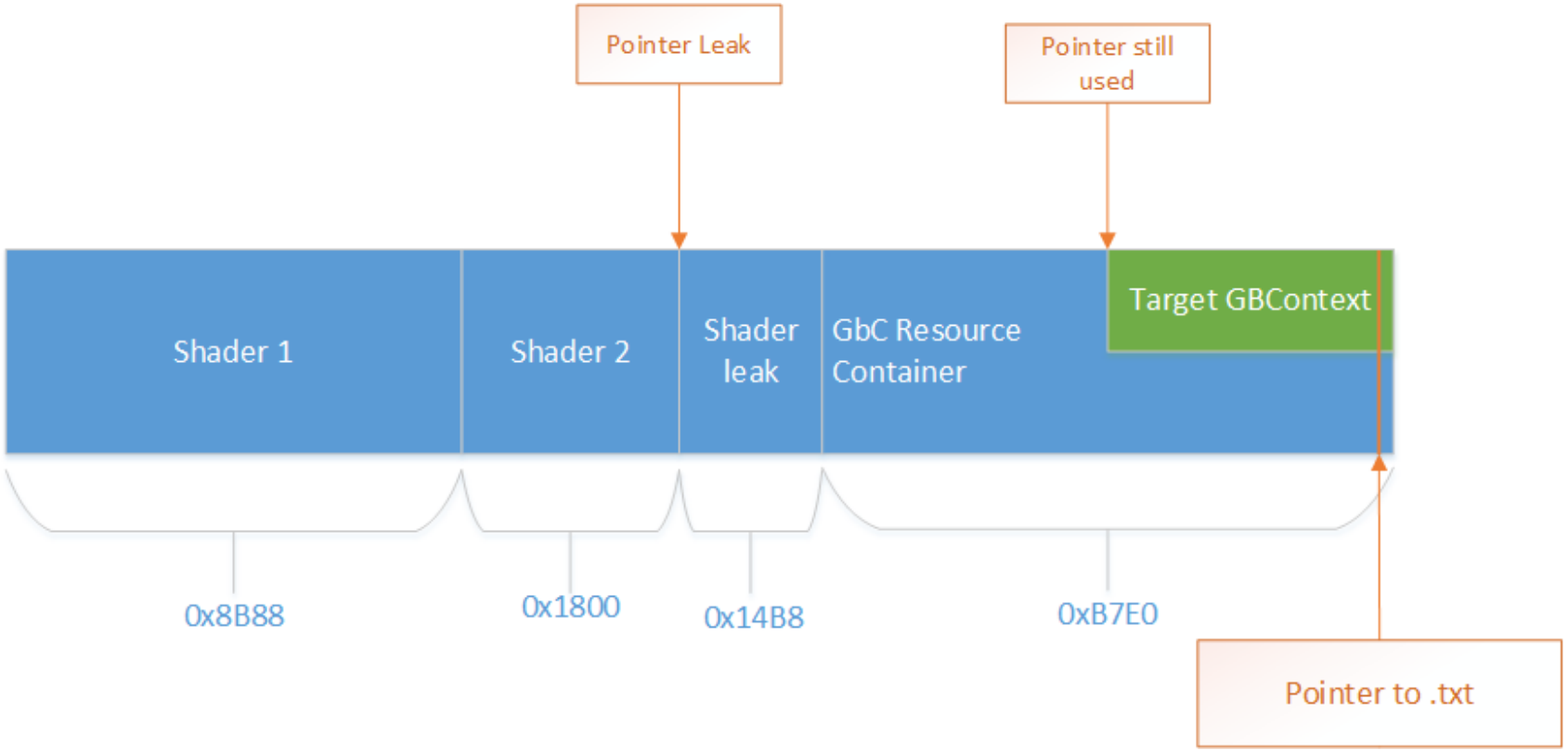
Reallocate the shader that were freed.



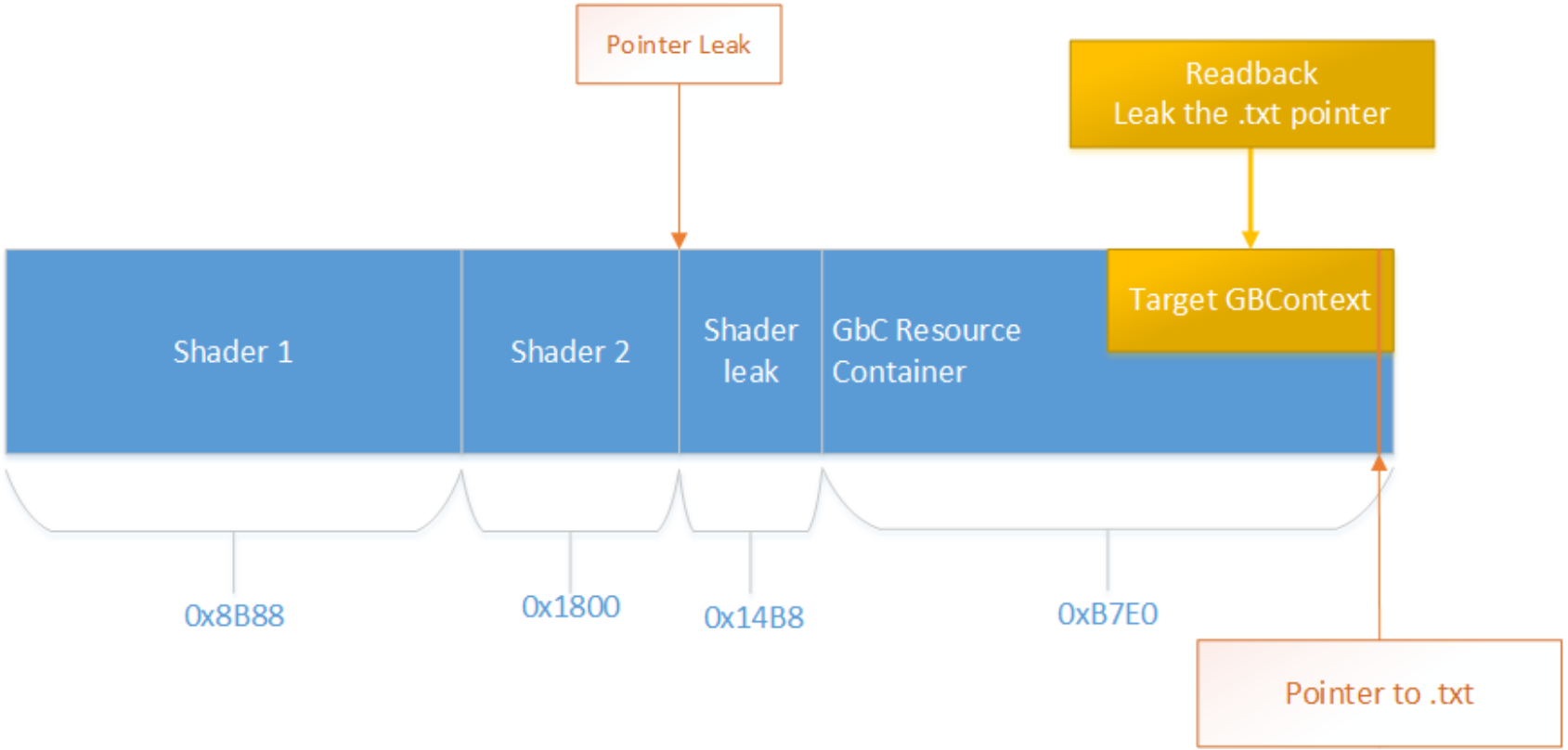
Exploit - Arbitrary free to .text leak



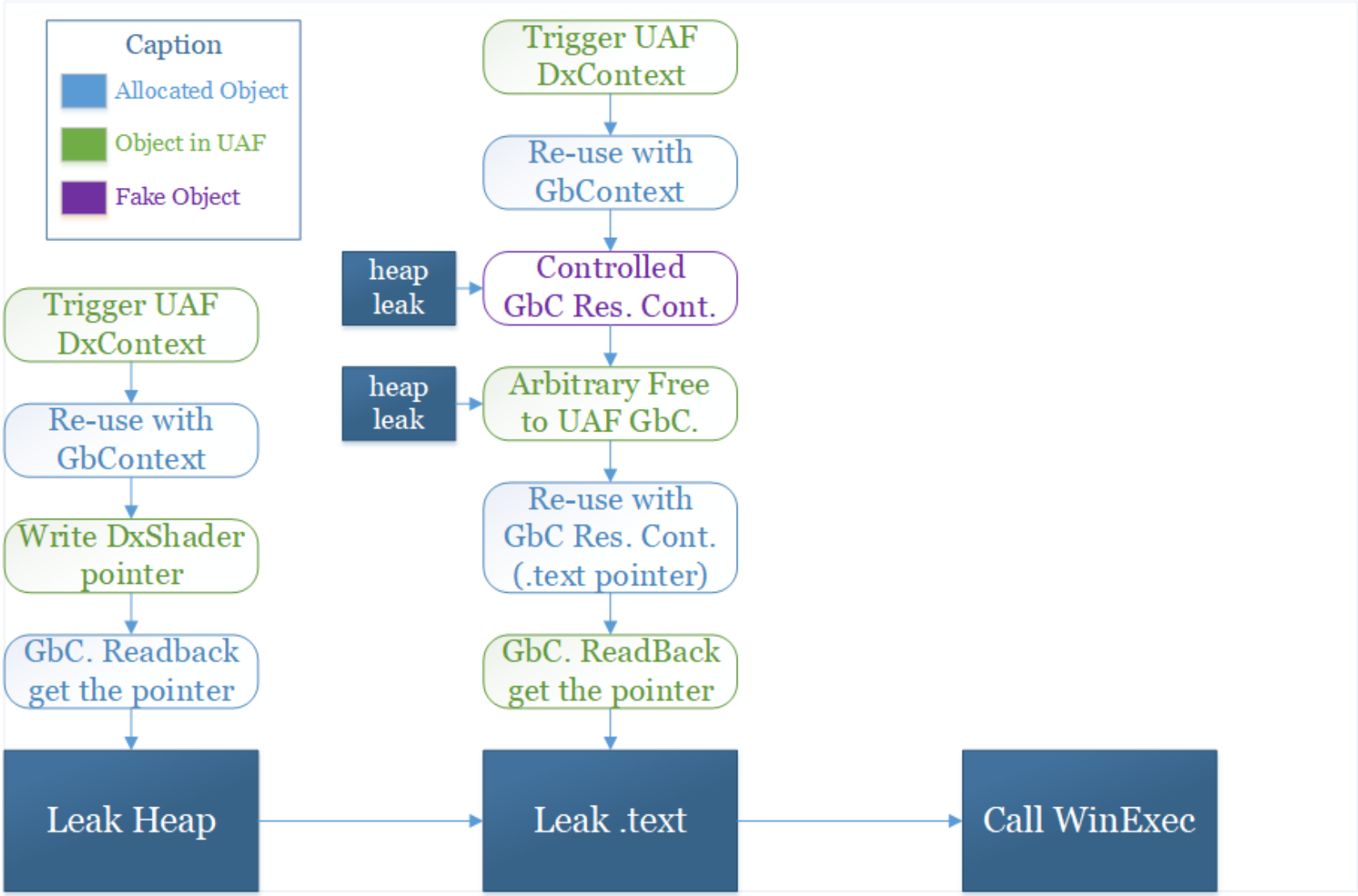
Exploit - Arbitrary free to .text leak



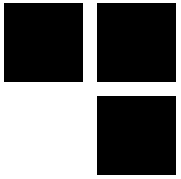
Exploit - Arbitrary free to .text leak



Exploit - Arbitrary free to .text leak



Plan

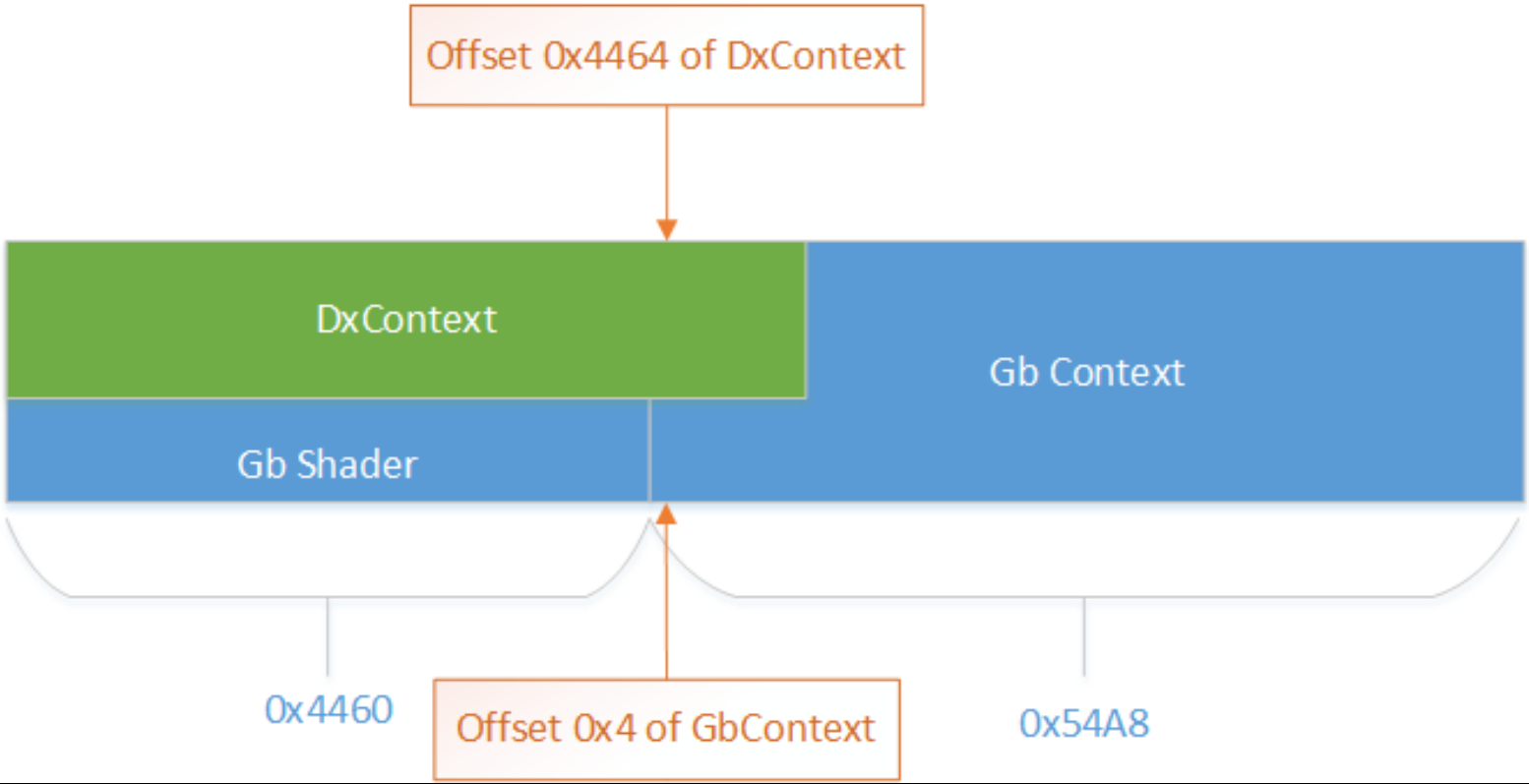


Exploit - Before leaving the command loop

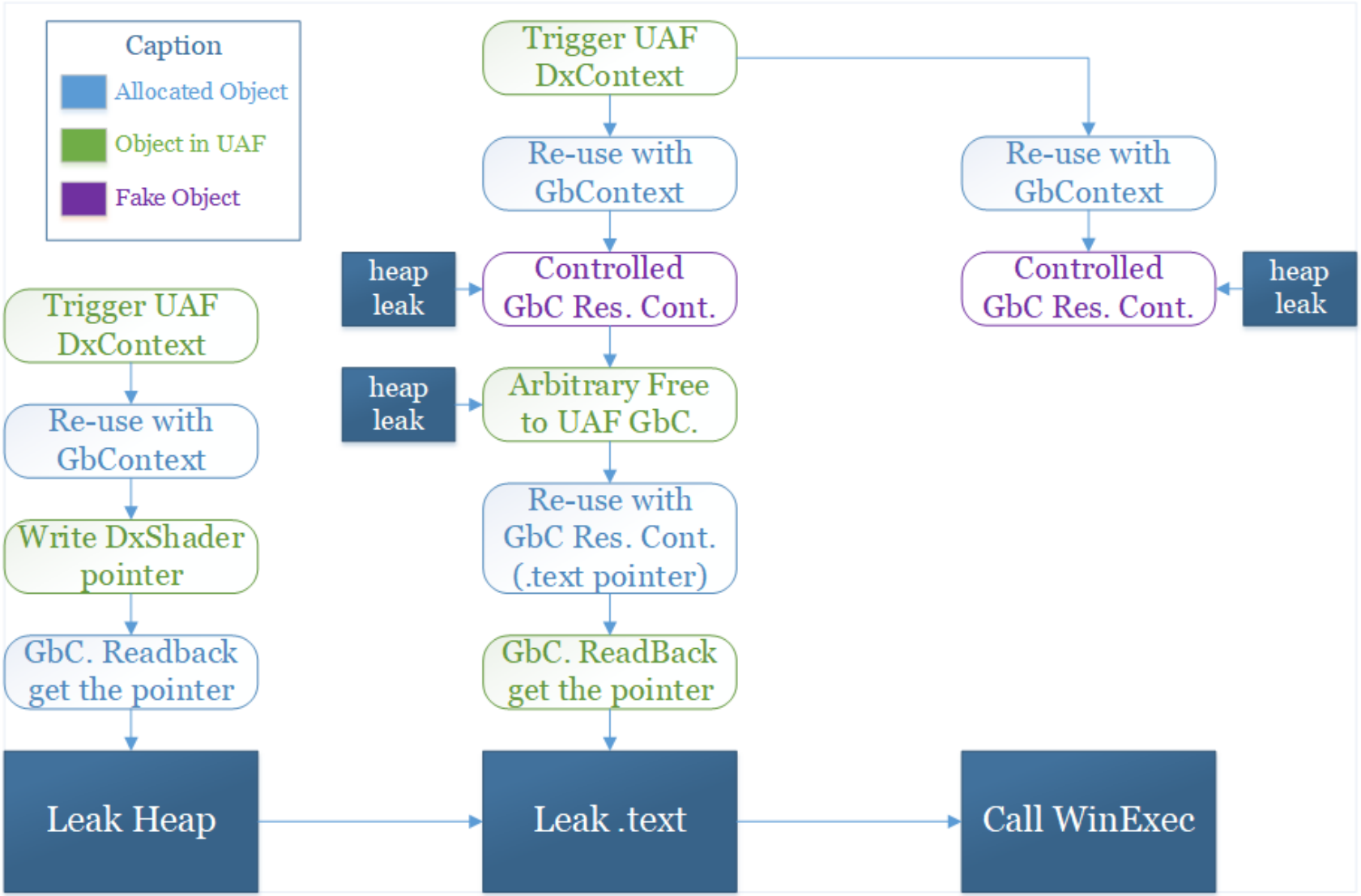
- To reach the arbitrary call, we need to reach the same code path.
- But we just freed the `GbContext` with the `RcIndex` controlled .
  - This is how we reach the arbitrary free path.

Exploit - Before leaving the command loop

- A `GbContext` can be quickly reallocated in the place of the old one.
- Write again the `RcIndex` using the `set_depthstencil_state` command.



Exploit - Before leaving the command loop



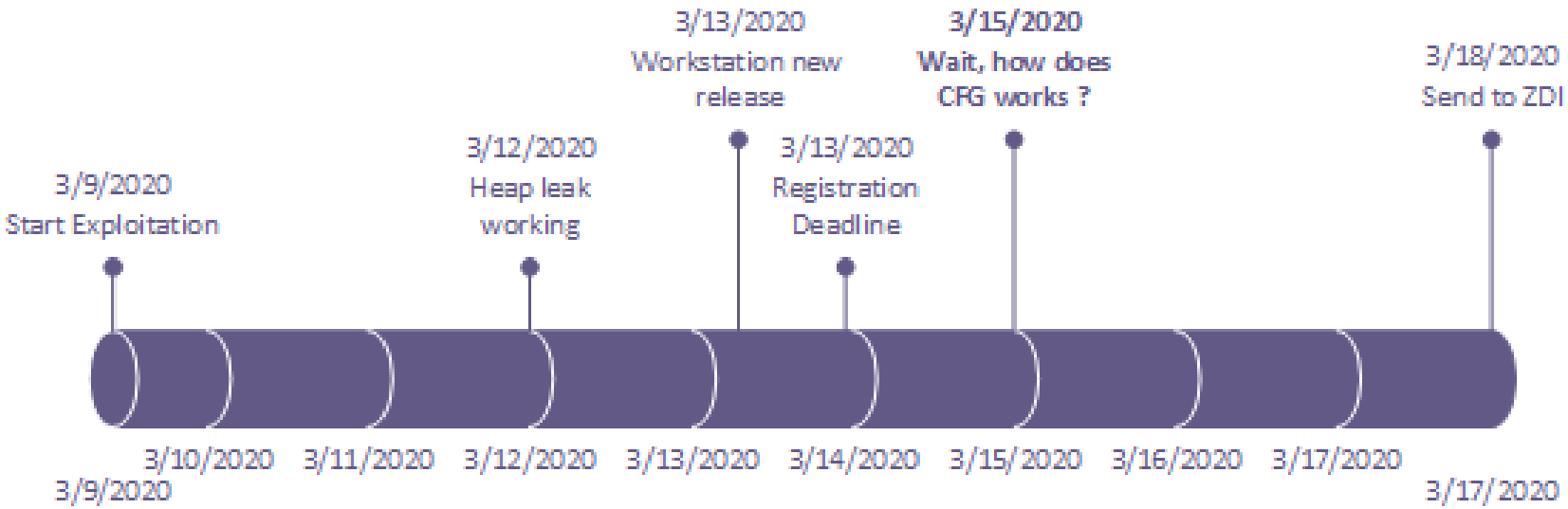
Exploit - Using the arbitrary call

- Reach the `SVGACallBackendDestroy` function, but this time to use the arbitrary call:

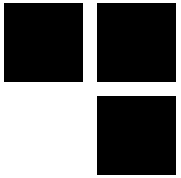
```
while (counter < RcBackend->nb_callbacks)
{
    v3 = RcBackend->callback_args[counter] ;
    if (!v3->called)
        RcBackend->callback_ptr(v3->arg1, v3->arg2);
    counter++;
    v3->called = 1;
}
```

- The loop allows to do multiple arbitrary calls.
  - Always the same function called.
  - But different arguments, always controlled.
- `RcBackend` is fully controlled and at known address.

Planning

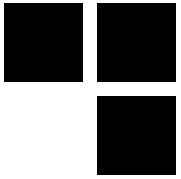

3/9/2020
Start Exploitation
3/12/2020
Heap leak working
3/13/2020
Workstation new release
3/13/2020
Registration Deadline
3/15/2020
Wait, how does CFG works ?
3/18/2020
Send to ZDI
3/9/2020
3/10/2020
3/11/2020
3/12/2020
3/13/2020
3/14/2020
3/15/2020
3/16/2020
3/17/2020
3/17/2020

Exploit - Not so arbitrary call



- 72 hours before the end: wait…how does Control Flow Guard (CFG) work ?
  - Check that addresses of indirect calls are "valid" functions.
  - Include only functions which can be called indirectly.
  - Compare with a bitmap.
  - Implementation in `ntdll!LdrpDispatchUserCallTarget`.
- Solution ?
  - Dynamic dump of the bitmap.
  - Re-implement the check in python.
  - Gather a list of all functions we can use in IDA.
  - Search for small functions, which deref. pointer, without loop and without call to other functions.
- In total, it took one of us ∼4 hours for doing all of this.

Exploit - Using the arbitrary call

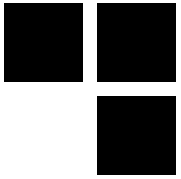


- Found an arbitrary read and write function.

```
int64 arbitrary_read_write(int64 src, QWORD *dst)
{
    result = *(_QWORD *)(src + 0x70);
    *dst = result;
    return result;
}
```

- Read a value at controlled location from argument 1.
- Stores it to a controlled location from argument 2.

Exploit - Using the arbitrary call

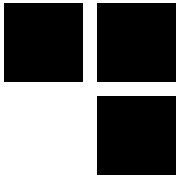


- Found an arbitrary increment with an arbitrary value on a DWORD.

```
int64 arbitrary_increment(unsigned int *src, _DWORD *dst)
{
    // [...]
    result = *src;
    *dst += result;
    return result;
}
```

- Read a value at controlled location from argument 1.
- Add it to a controlled location from argument 2.

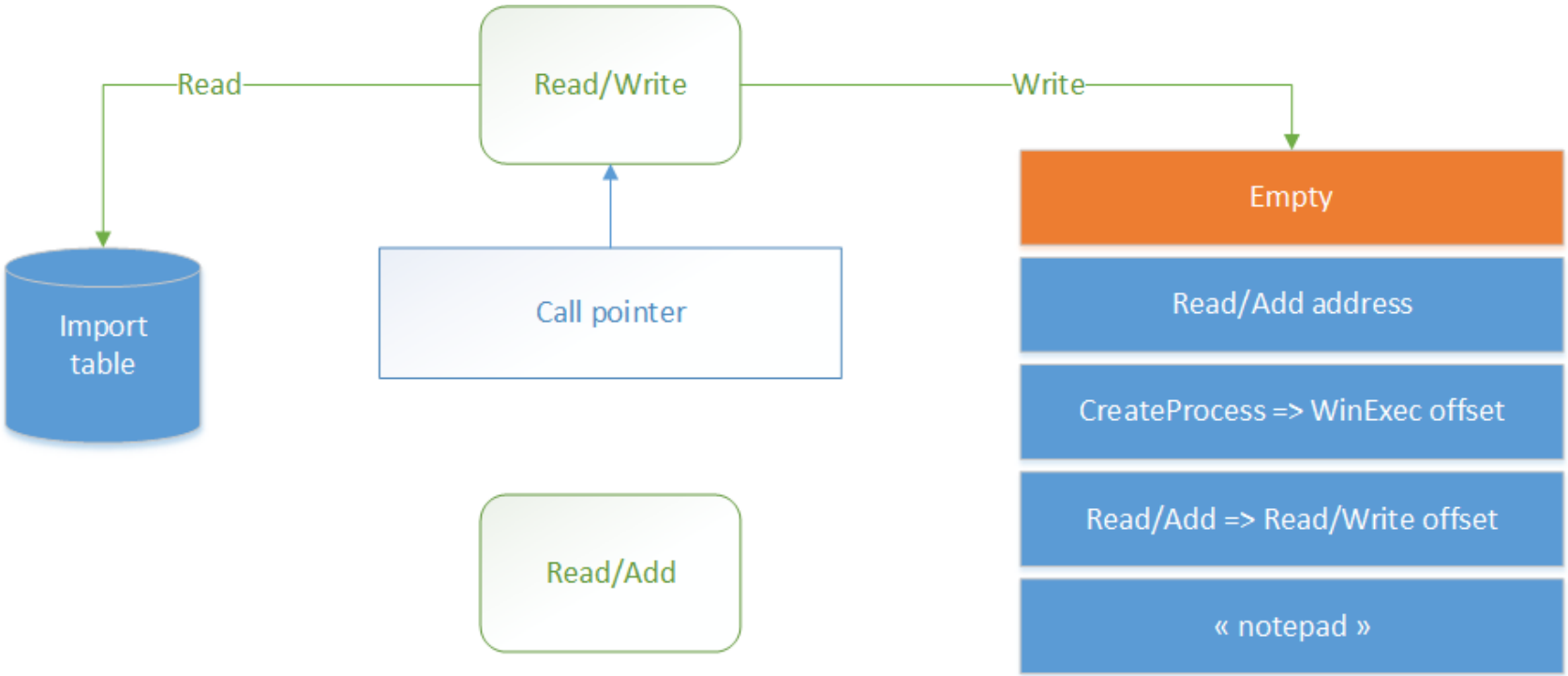
Exploit - Using the arbitrary call



- Ability to read and write in memory.
- Ability to do addition.
- Ability to control the execution flow.
  - Use the read/write to change the value of the function pointer.
- We got a Turing machine !
  - Calls can be chained to read/write and call `WinExec`.

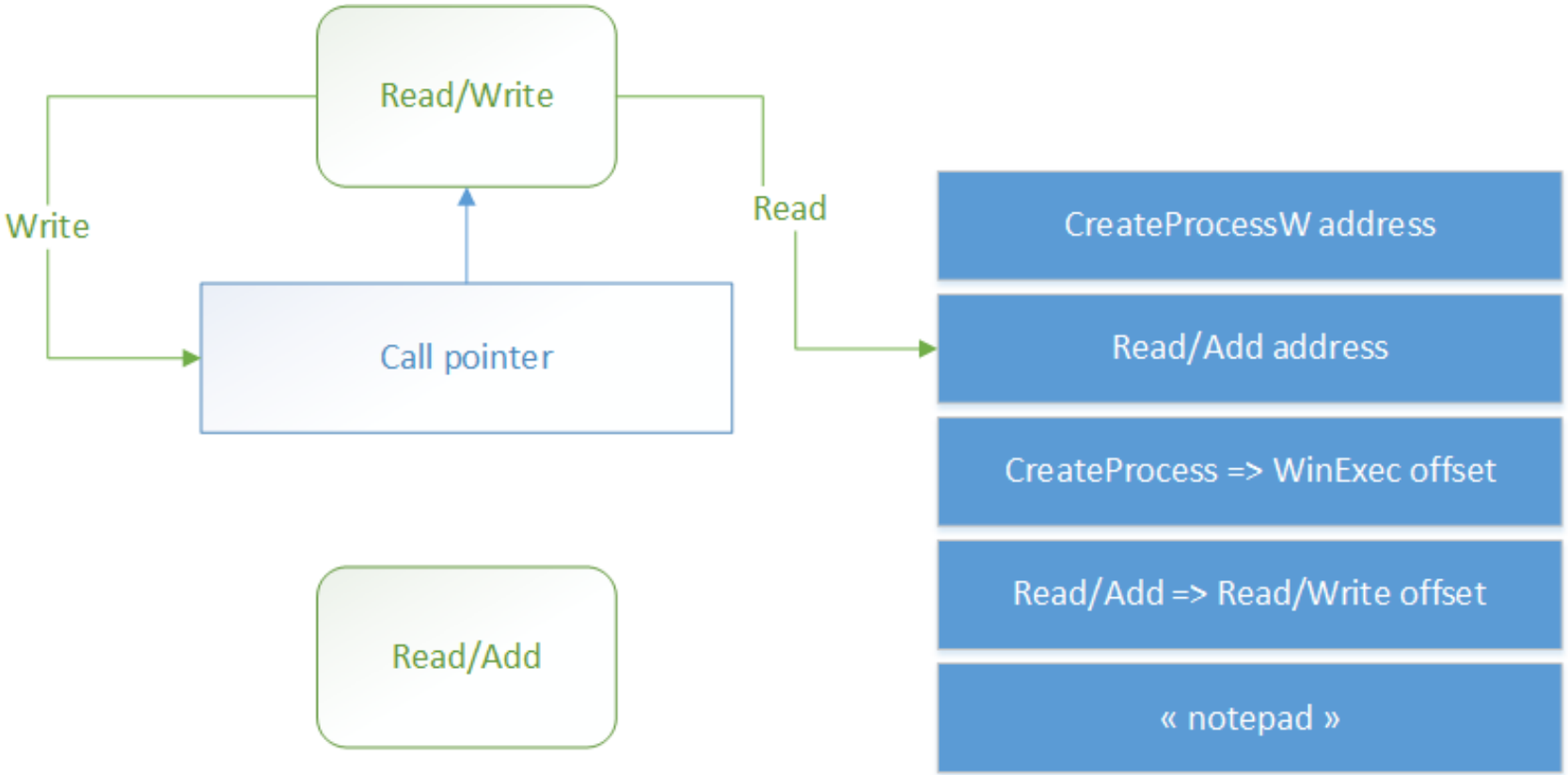
Exploit - Using the arbitrary call

1. Read the Import table address of `CreateProcessW` and write it at a known location.



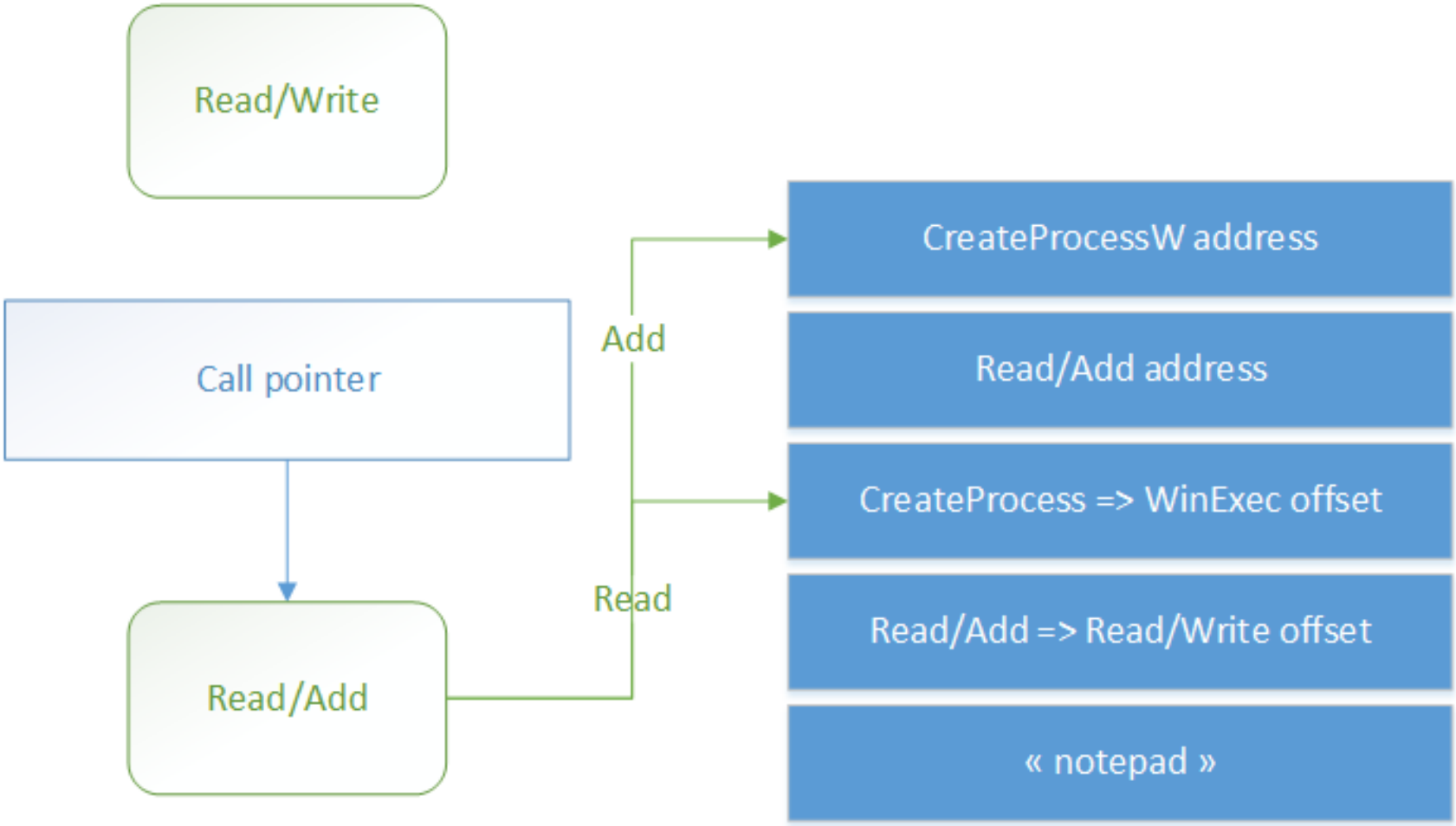
Exploit - Using the arbitrary call

2 Use the read/write to change the next call address, and change it to the arbitrary add function.



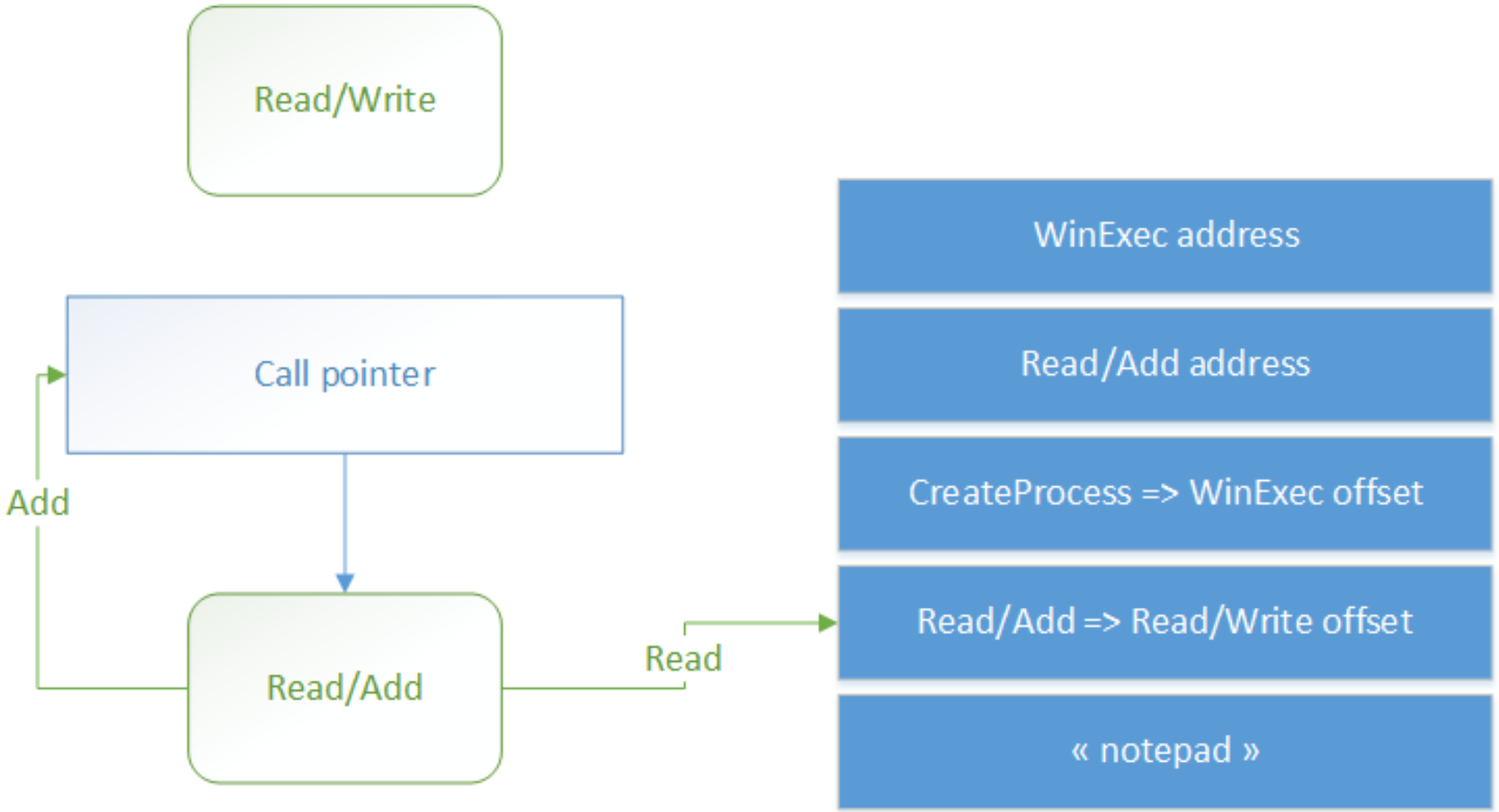
Exploit - Using the arbitrary call

3 Adds the `WinExec` offset to the `CreateProcessW` address.



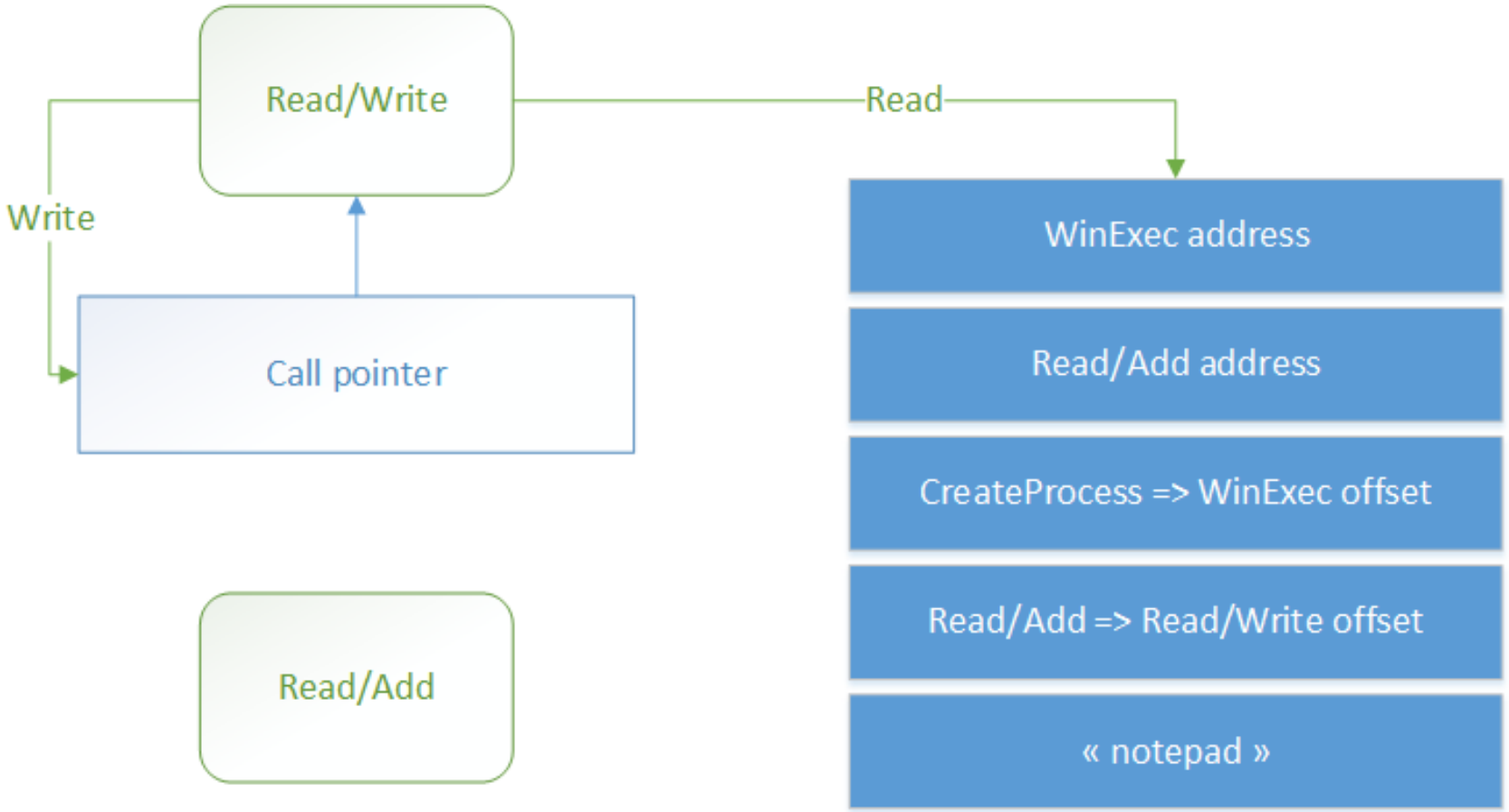
Exploit - Using the arbitrary call

4 Use the arbitrary add to add an offset to the call pointer, and change it back to the arbitrary read function.



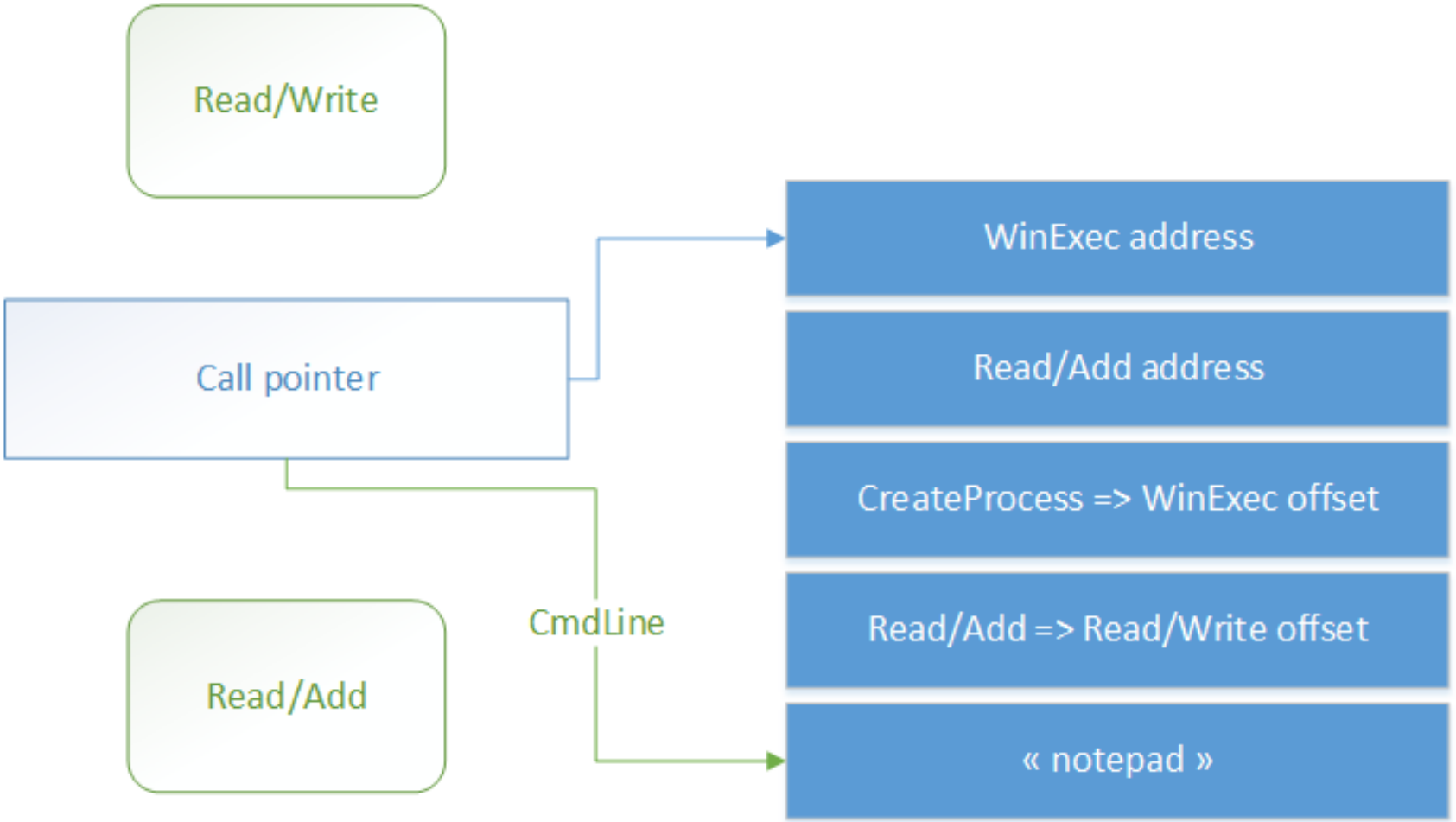
Exploit - Using the arbitrary call

5 Use the arbitrary read to readback the `WinExec` address and replace the call pointer by `WinExec`.

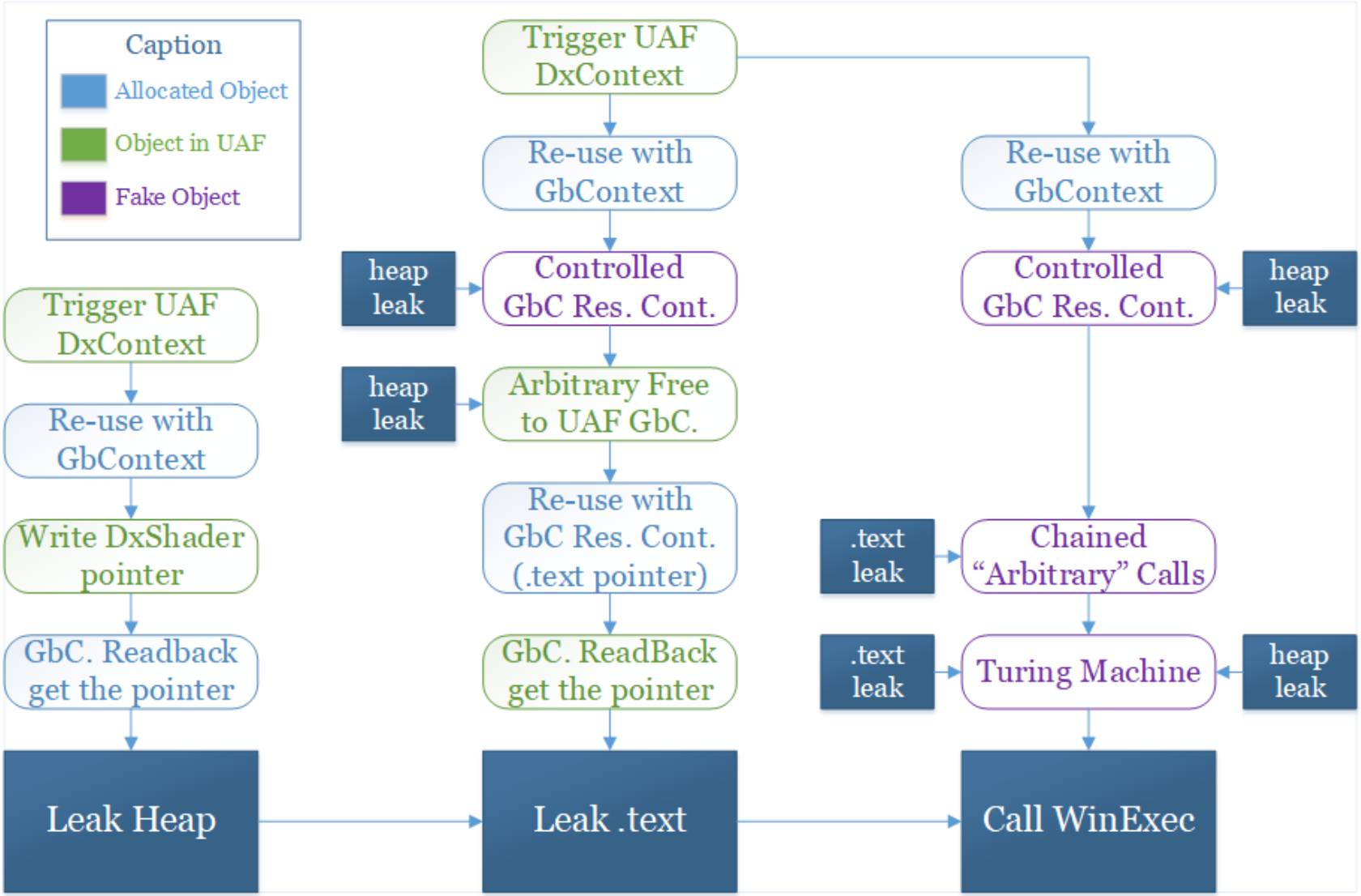


Exploit - Using the arbitrary call

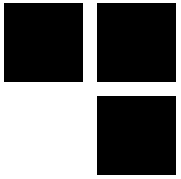
6 Call `WinExec` with controlled arguments.



Pwn2Own - Popping a notepad

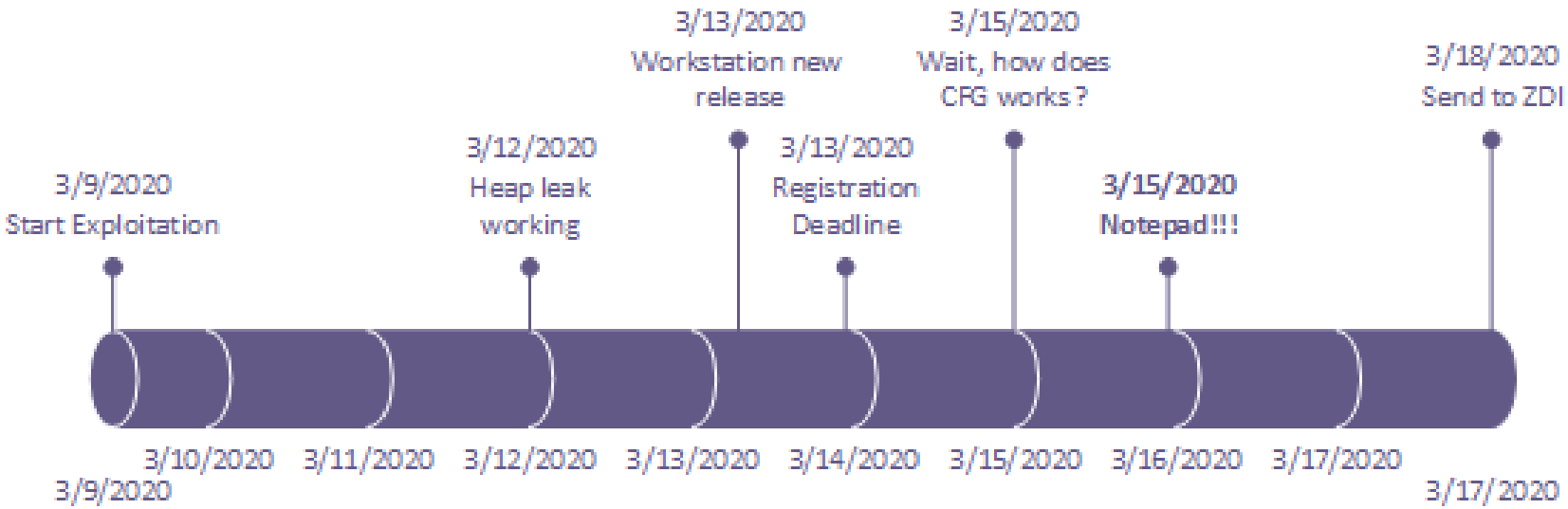


Pwn2Own - Popping a notepad

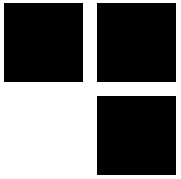


DEMO

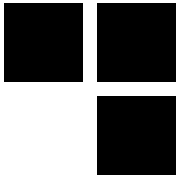
Planning


3/9/2020
Start Exploitation
3/12/2020
Heap leak working
3/13/2020
Workstation new release
3/13/2020
Registration Deadline
3/15/2020
Wait, how does CFG works?
3/15/2020
Notepad!!!
3/18/2020
Send to ZDI
3/9/2020
3/10/2020
3/11/2020
3/12/2020
3/13/2020
3/14/2020
3/15/2020
3/16/2020
3/17/2020
3/17/2020

Plan

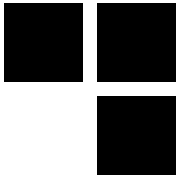


Exploit - Timeline



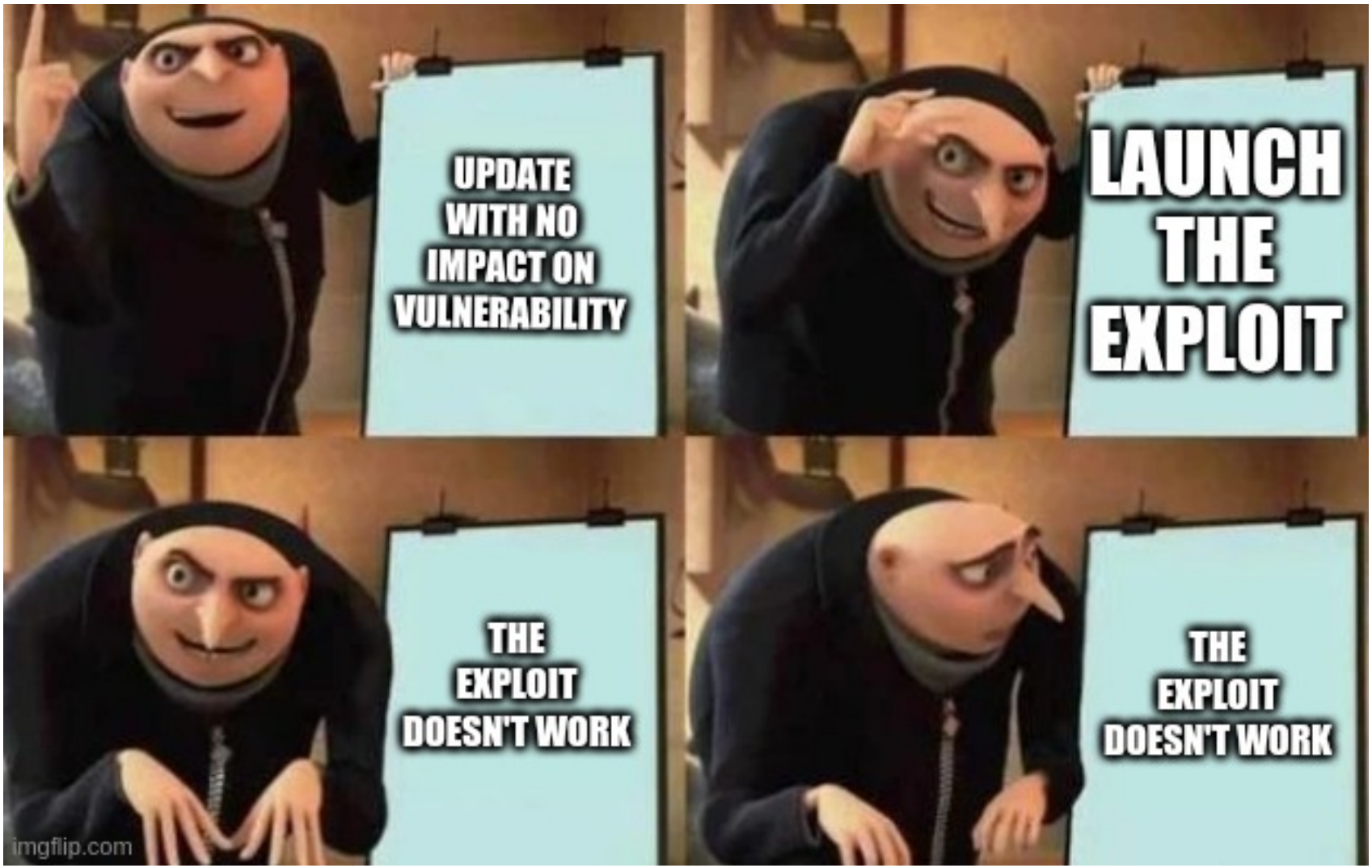
- Monday March 16th, 3AM: First notepad!
- Exploit reliability around 60-80%:
  - asynchronous SVGA2 event,
  - heap-spray failure,
  - uncompatible original device state,
  - …
- Time left: 39 hours before the contest.

Exploit - Timeline

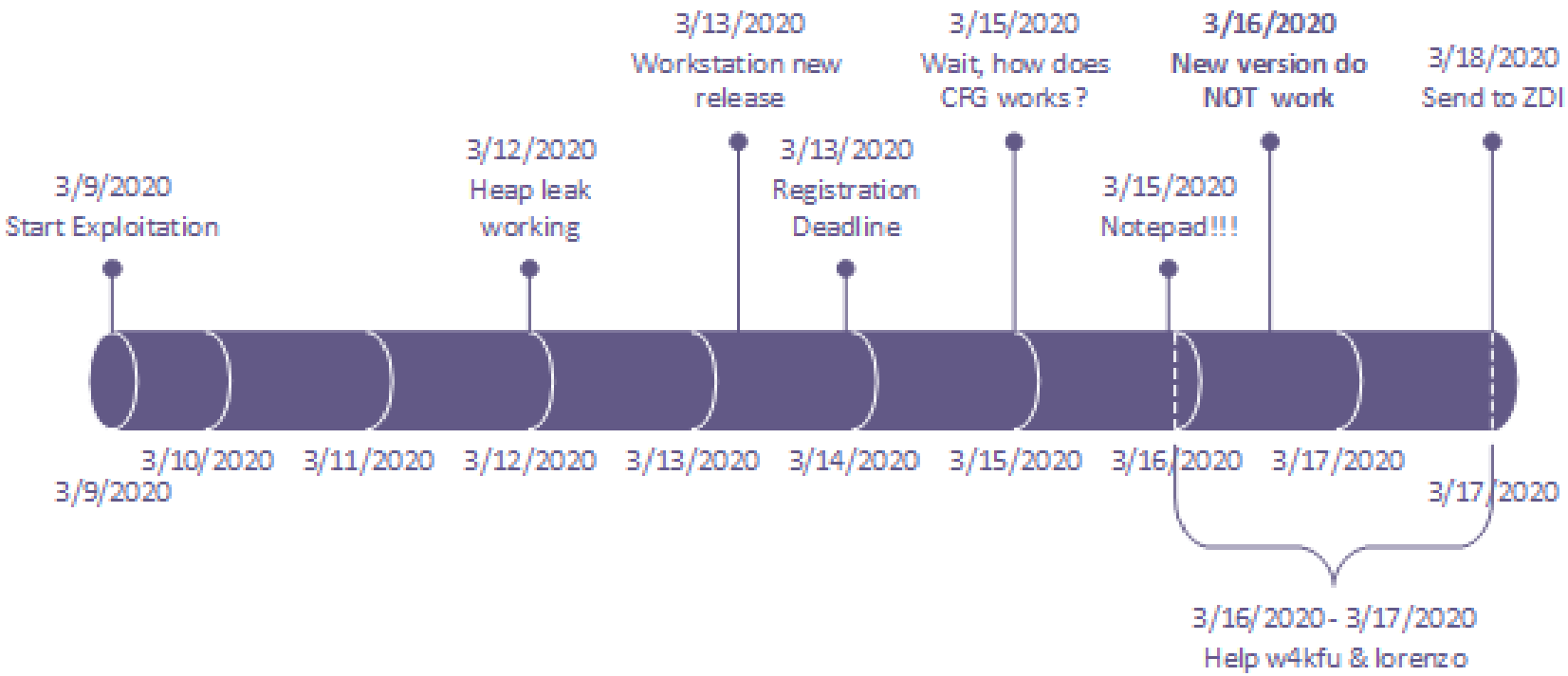


- Thursday March 12th: new version of Vmware Workstation.
  - VMSA-2020-0004
  - Of course our exploit run on previous version.
  - But no impact on our vulnerability, shouldn't change anything.
- Let's update and run our exploit.

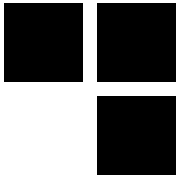
Exploit - WTF


UPDATE WITH NO IMPACT ON VULNERABILITY
LAUNCH THE EXPLOIT
THE EXPLOIT DOESN'T WORK
THE EXPLOIT DOESN'T WORK
imgflip.com

Planning

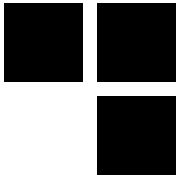

3/9/2020
Start Exploitation
3/12/2020
Heap leak
working
3/13/2020
Workstation new
release
3/13/2020
Registration
Deadline
3/15/2020
Wait, how does
CFG works?
3/15/2020
Notepad!!!
3/16/2020
New version do
NOT work
3/18/2020
Send to ZDI
3/9/2020
3/10/2020
3/11/2020
3/12/2020
3/13/2020
3/14/2020
3/15/2020
3/16/2020
3/17/2020
3/17/2020
3/16/2020- 3/17/2020
Help w4kfu & lorenzo

Exploit - WTF



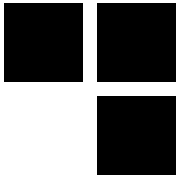
- The leak doesn't work anymore.
- Looking for the root cause.
  - 30 hours remaining before the contest.
- Some patches in the parsing of the `DxShaders`.
- More constraints on the allocation of the shader to get the leak.
  - Impacts the exploit widely.
  - All size in the massaging needs to be changed.
- Fixed Monday evening.
  - Reliability of the exploit falls to 30%.
  - 24 hours remaining before the contest.

Exploit - Last day



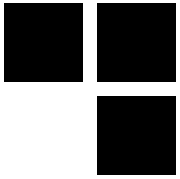
- Fix the reliability of the exploit:
  - spray more,
  - works at 80% on our setup (two different computers).
- Do the setup for the contest.
- Our exploit works in Python, need an .exe.
  - Thanks py2exe & SFX.
- Our exploit use a kernel in debug mode to get read/write on physical memory.
  - Use a vulnerable signed driver to load a custom driver.
  - access to physical memory and IOports.
- Thanks @w4kfu and @_lucas_georges_ that helped us on this.

Plan



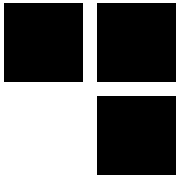
Pwn2Own - Us at Pwn2Own

Pwn2Own - aaaaand... fail



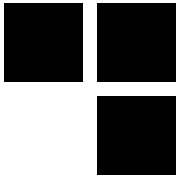
- Failed attempt:
  - The exploit failed 3 times.
  - Numerous problems we knew about and could have fix, but did not have time.
  - Of course we don't know exactly why.
  - Very frustrating.
- But ZDI did buy the vulnerability and the exploit outside of the contest.
  - Really cool from them !

Pwn2Own - Feedback

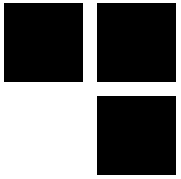


- Pwn2Own is well organized and really cool.
  - Answered quickly to our many (many !) questions.
- ZDI does everything to make your exploit works. They want you to succeed !
- Thanks to ZDI, Pwn2Own is great !

Plan

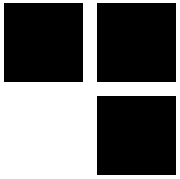


The End



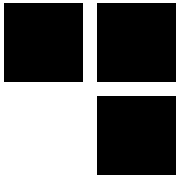
- We spent 2 months working as much as we could.
- Several bugs found:
  - One exploitable and exploited: CVE-2020-3962.
  - One hardly exploitable: CVE-2020-3969 (reported to ZDI outside of the contest).
  - Multiple bugs useless or unexploitable.
- VMWare Advisory VMSA-2020-0015.

The End



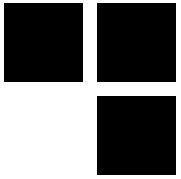
- Notes for next time:
  - 40 days is not really enough ;)
  - Less attempt of fuzzing, more reverse.
- FastPwning is challenging.
  - Might be frustrating.
  - Exhausting.
  - Awesome !!

Thanks



- @_zisis for the amazing paper *Straight outta VMware*
- @thezdi for Pwn2Own
- @hakril, @w4kfu and @_lucas_georges_ for their amazing tools and help !

Plan



1 Introduction

2 Workstation Discovery

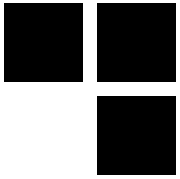
3 Vulnerability Research

4 Exploit

5 Conclusion

6 Annexes
- State of The Art
- Our Setup

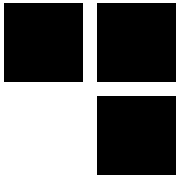
Plan



State Of The Art – publication

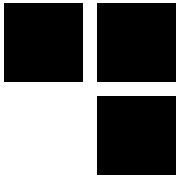
- Small number of publications (∼25) about VMware.
- Basically everything listed in: *VMware Exploitation github*.
- Some articles about specific components are interesting for those components but not in general.
- Three interesting papers:
  - *Straight outta VMware* is the most interesting paper, speak about Workstation internals and specifically about SVGA.
  - *The Great Escapes Of Vmware* provides an overview of past vulnerabilities.
  - *Wandering through the Shady Corners of VMware Workstation/Fusion* contains information helping to start the RE and things specific to the SVGA.

State Of The Art – tools & code



- Code of the Linux drivers for the emulated devices is open-source.
- Code in Mesa is interesting for the SVGA.
- *Open-vm-tools* distributed by VMware are opensource and contains lots of code, some of it is actually shared with the hypervisor.

Plan



VMware binaries

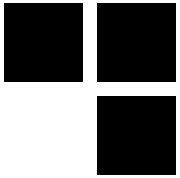
Original Binaries

- Version: 15.5.1.50853
- vmware-vmx.exe MD5:
  `D76FEB17DF9153630D00E373A6ECB99B`

Final Binaries

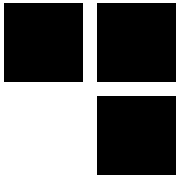
- Version: 15.5.2.54704
- vmware-vmx.exe MD5:
  `B23A9F348DA1F2DC2B0D6B2DB5D9CCA7`

Our setup



- For the debug of the hypervisor:
  - we only debugged the userland process (vm worker).
  - *PythonForWindows* (PFW) by *@hakril* for scripting.
  - Windbg for interactive.
- For interfacing with the devices from the guest (necessity to be in kernel): *LKD python3* by *@w4kfu*.
- For reverse: IDA and *Bip* by *Bruno* for scripting.
- For fuzzing: Synacktiv internal fuzzer & *winafl*. Also use *lighthouse* for coverage and analysis.

Guest driver & devices initialization



- The guest drivers are doing the initialization of the devices, those are provided by VMware.
- For not having to support the initialization (RE, re-implement, …), we just patch them for "stopping" them.
- In particular we did that for SVGA (*vm3dmp.sys*)…
- …and we regretted this choice later on :/

ANY
QUESTIONS?

THANK YOU

SYNACKTIV
DIGITAL SECURITY