Journey to freedom

Typhoon

CORENTIN BAYET

BRUNO PUJOS

RE TACTICS

Who am I ?

CORENTIN BAYET

Security researcher and CTO of REverse Tactics.

Specialized in low-level software reverse engineering and exploit, and in particular:

- Kernel and OS security
- Hypervisors
- Embedded Software

Who am I ?

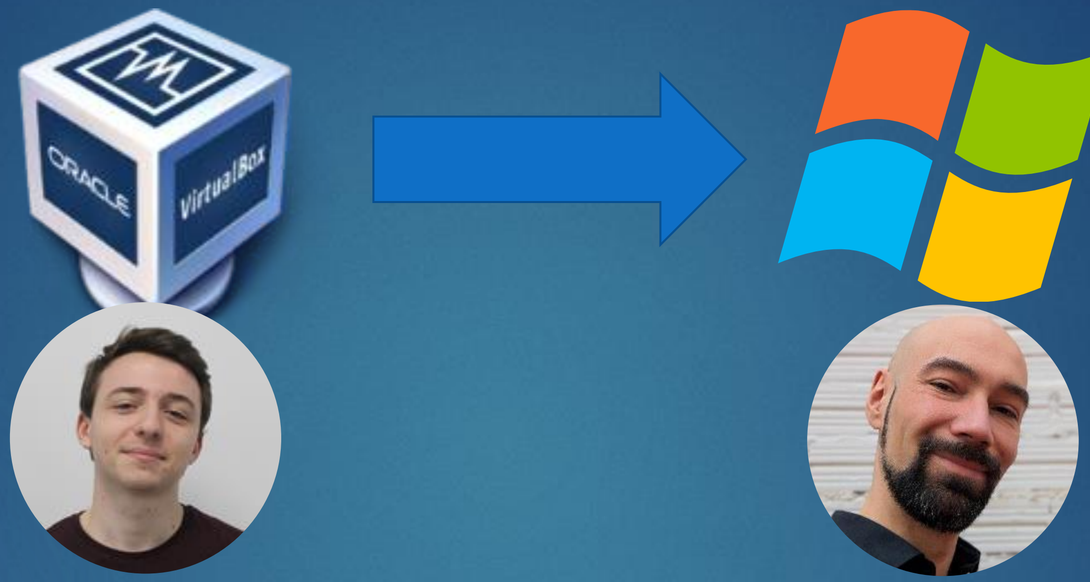
CEO and founder of REverse Tactics.

Security researcher specialized in Reverse Engineering & Vulnerability Research. In particular:

- UEFI firmware
- Kernel & virtualization.
- Embedded Software

BRUNO PUJOS

Pwn2Own Vancouver 2024



- At Pwn2Own we chained a VirtualBox VM Escape with a LPE Windows

Pwn2Own rules (virtualization)

- Exploit needs to demonstrate Virtual Machine escape (VME)
  - Start with administrator/root privileges in the guest (Linux or Windows)
  - Must demonstrate code execution on the host
    - Up-to-date Windows for Virtualbox
    - Can be chained with elevation of privileges on the host for a bonus
- About configuration
  - Virtual machines can have a great variety of configurations
    - Big impact on the available attack surface
  - Doesn’t have to target the default configuration
    - But must represent a realistic real life scenario
    - The organizer decides

Plan

01 Finding the map

02 Reaching high seas

03 Exploring the ocean

04 Hunting for gold

05 Stealing the treasure

06 Making Port

Oracle VirtualBox

- Popular hypervisor
  - Open source
  - Free
  - Easy to use
  - Working on Windows / Linux / MacOS
- Maintained by Oracle
  - No team 100% dedicated to VirtualBox's security

A few definitions

- **Hypervisor**: Software that manages one or multiple virtual machines on a single physical computer
  - Here, Virtualbox
- **Host**: Operating system running the hypervisor
  - Here, Windows is running Virtualbox
- **Guest**: Operating system running in the virtual machine
- **GPA**: Guest Physical Address
  - An address in the physical memory view of the guest
- **Paravirtualization**: virtualization technique
  - Guest OS is modified to communicate directly with the hypervisor
  - Improved performances

Communication channels

- Exchange data through shared memory
  - Direct Memory Access (DMA)
- Trigger specific actions through
  - Port mapped Input/Output (PMIO)
    - Privileged instructions: IN / OUT
  - Memory Mapped IO (MMIO)
    - Read / write in specific physical memory ranges
  - Hypercalls
    - Specific interfaces used with paravirtualized devices

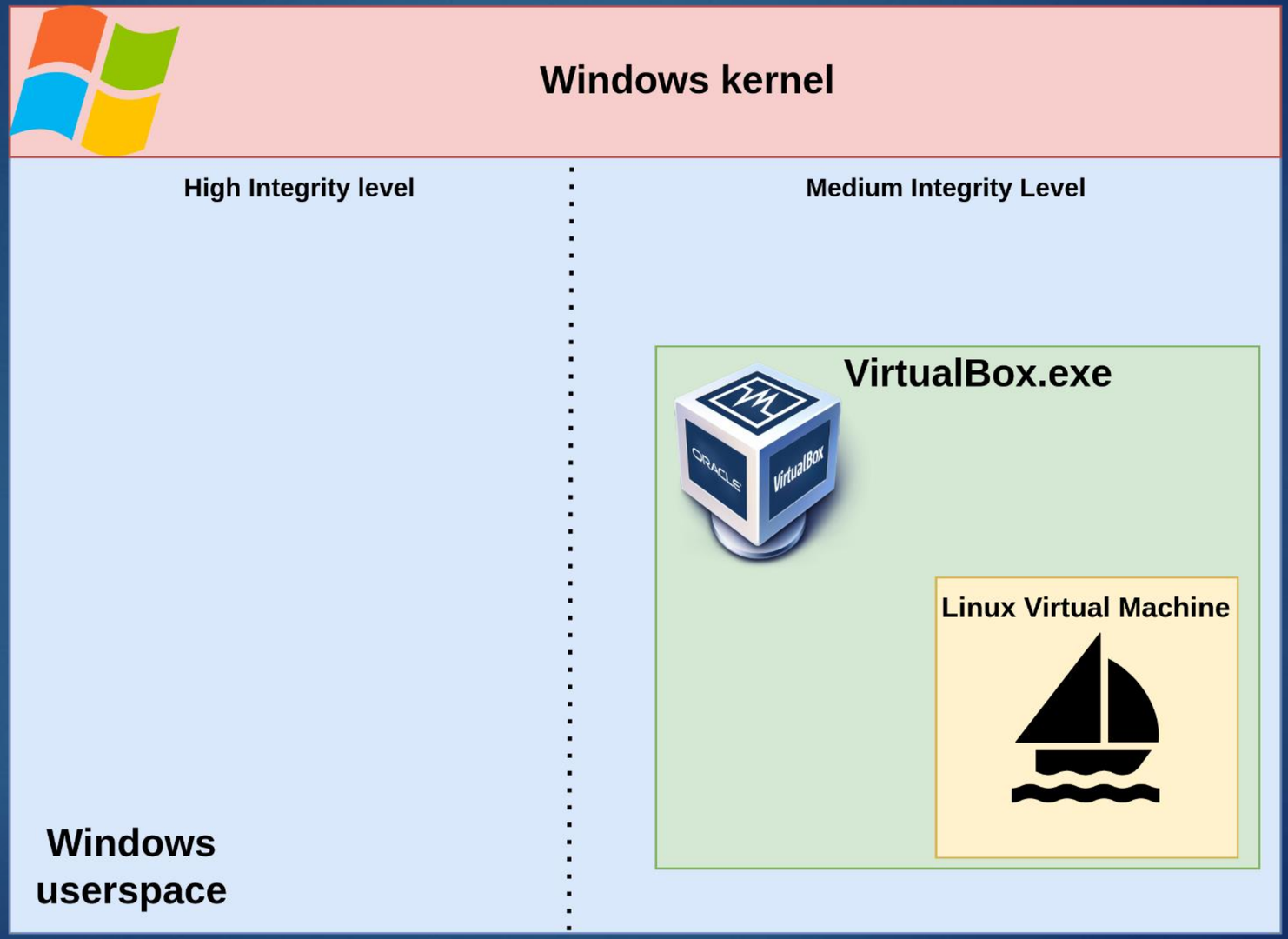
Setup

- **Chipsec**
  - Framework for testing the security of hardware or system firmware (UEFI / BIOS)
- Already developed drivers for Windows and Linux that exposes privileged operations
  - Allocate / Read / Write physical memory
  - Execute privileged instructions
    - IN / OUT (PMIO)
    - Hypercalls
  - Read / Write in PCI
- Has a Python API !
  - OS agnostic !

Setup

```python
from chipsec import chipset

cs = chipset.cs().basic_init_with_helper()

# Allocate and write into physical memory
phys_addr = cs.mem.alloc_physical_mem(0x1000, 0xffffffff)
cs.mem.write_physical_mem(phys_addr, b'A'*0x1000)

# Trigger MMIO, provide DMA address
mmio_data = phys_addr.to_bytes(4, byteorder='little')
cs.mmio.write_MMIO_reg(0xbc000000, 0, mmio_data, 4)

# read result
data = cs.mem.read_physical_mem(phys_addr, 0x1000)
```

Treasure map



Finding the map

- A good journey always start with a good state of the art
  - Very important step, not to neglect
  - MUST put time into it

- Goals:
  - Find generic information on the target
    - Public documentation
    - Source code organization
    - Is it fuzzed ?
- Study previous vulnerabilities
  - Understand common attack surfaces
  - Note exploit techniques
  - Extract **vulnerable patterns**

CVE-2023-21988

- Uninitialized memory read in VirtualBox
  - Found and exploited by @MajorTomSec of Synacktiv for Pwn2Own 2023
- Bug affecting **PGMPhysRead**
  - Function responsible for reading the physical memory of the guest to a host buffer
  - See it as an equivalent of **copy_from_user** or **memcpy**
    - The source address is a GPA

```
 * @param   pVM             The cross context VM structure.
 * @param   GCPhys          Physical address start reading from.
 * @param   pvBuf           Where to put the read bits.
 * @param   cbRead          How many bytes to read.
 * @param   enmOrigin       The origin of this call.
 */
VMMDECL(VBOXSTRICTRC) PGMPhysRead(PVMCC pVM, RTGCPHYS GCPhys, void *pvBuf, size_t cbRead, PGMACCESSORIGIN enmOrigin)
```

CVE-2023-21988

- This function will split the access page by page
  - Because each guest physical page can be located at a different place in host's memory

- It also handle MMIO accesses
  - If one of the GPA is registered as a MMIO, call the appropriate MMIO handler
  - If any error occurs during the MMIO handling fill up the output buffer and return

CVE-2023-21988

```
VMMDECL(VBOXSTRICTRC) PGMPhysRead(PVMCC pVM, RTGCPHYS GCPhys, void *pvBuf, size_t cbRead, PGMACCESSORIGIN enmOrigin)
{
    // [...] Loop on each page
    {
        size_t   cb    = GUEST_PAGE_SIZE - (off & GUEST_PAGE_OFFSET_MASK);
        if (cb > cbRead)
            cb = cbRead;

        // Is a MMIO Page
        if ( PGM_PAGE_HAS_ACTIVE_ALL_HANDLERS(pPage) || PGM_PAGE_IS_SPECIAL_ALIAS_MMIO(pPage))
        {
            // call MMIO handler
            VBOXSTRICTRC rcStrict2 = pgmPhysReadHandler(pVM, pPage, pRam->GCPhys + off, pvBuf, cb, enmOrigin);
            if (PGM_PHYS_RW_IS_SUCCESS(rcStrict2))
                PGM_PHYS_RW_DO_UPDATE_STRICT_RC(rcStrict, rcStrict2);
            else
            {
                /* Set the remaining buffer to a known value. */
                memset(pvBuf, 0xff, cb);
                PGM_UNLOCK(pVM);
                return rcStrict2;
            }
        }
        // [...]
    }
}
```

Note: code was simplified

CVE-2023-21988

```
VMMDECL(VBOXSTRICTRC) PGMPhysRead(PVMCC pVM, RTGCPHYS GCPhys, void *pvBuf, size_t cbRead, PGMACCESSORIGIN enmOrigin)
{
    // [...] Loop on each page
    {
        size_t    cb    = GUEST_PAGE_SIZE - (off & GUEST_PAGE_OFFSET_MASK);
        if (cb > cbRead)
            cb = cbRead;

        // Is a MMIO Page
        if ( PGM_PAGE_HAS_ACTIVE_ALL_HANDLERS(pPage) || PGM_PAGE_IS_SPECIAL_ALIAS_MMIO(pPage))
        {
            // call MMIO handler
            VBOXSTRICTRC rcStrict2 = pgmPhysReadHandler(pVM, pPage, pRam->GCPhys + off, pvBuf, cb, enmOrigin);
            if (PGM_PHYS_RW_IS_SUCCESS(rcStrict2))
                PGM_PHYS_RW_DO_UPDATE_STRICT_RC(rcStrict, rcStrict2);
            else
            {
                /* Set the remaining buffer to a known value. */
                memset(pvBuf, 0xff, cb);
                PGM_UNLOCK(pVM);
                return rcStrict2;
            }
        }
        // [...]
    }
}
```

- Only calls memset for the current page size.
- Remaining on the **pvBuf** buffer remains uninitialized.

CVE-2023-21988

- Bug allows to let some data uninitialized when reading from guest physical memory
  - Requires to control the GPA to trigger an error
  - This is a very common pattern
- Requires to find a code that will write back this uninitialized data to the guest
  - Found in the XHCI device
- Impact:
  - Leak uninitialized memory from the host
  - Get some stack/heap pointers and defeat ASLR

CVE-2023-21988

```
VMMDECL(VBOXSTRICTRC) PGMPhysRead(PVMCC pVM, RTGCPHYS GCPhys, void *pvBuf, size_t cbRead, PGMACCESSORIGIN enmOrigin)
{
    // [...] Loop on each page
    {
        size_t   cb    = GUEST_PAGE_SIZE - (off & GUEST_PAGE_OFFSET_MASK);
        if (cb > cbRead)
            cb = cbRead;

        // Is a MMIO Page
        if ( PGM_PAGE_HAS_ACTIVE_ALL_HANDLERS(pPage) || PGM_PAGE_IS_SPECIAL_ALIAS_MMIO(pPage))
        {
            // call MMIO handler
            VBOXSTRICTRC rcStrict2 = pgmPhysReadHandler(pVM, pPage, pRam->GCPhys + off, pvBuf, cb, enmOrigin);
            if (PGM_PHYS_RW_IS_SUCCESS(rcStrict2))
                PGM_PHYS_RW_DO_UPDATE_STRICT_RC(rcStrict, rcStrict2);
            else
            {
                /* Set the remaining buffer to a known value. */
                memset(pvBuf, 0xff, cb);
                PGM_UNLOCK(pVM);
                return rcStrict2;
            }
        }
        // [...]
    }
}
```

CVE-2023-21988 - Patched

```
VMMDECL(VBOXSTRICTRC) PGMPhysRead(PVMCC pVM, RTGCPHYS GCPhys, void *pvBuf, size_t cbRead, PGMACCESSORIGIN enmOrigin)
{
    // [...] Loop on each page
    {
        size_t    cb    = GUEST_PAGE_SIZE - (off & GUEST_PAGE_OFFSET_MASK);
        if (cb > cbRead)
            cb = cbRead;

        // Is a MMIO Page
        if ( PGM_PAGE_HAS_ACTIVE_ALL_HANDLERS(pPage) || PGM_PAGE_IS_SPECIAL_ALIAS_MMIO(pPage))
        {
            // call MMIO handler
            VBOXSTRICTRC rcStrict2 = pgmPhysReadHandler(pVM, pPage, pRam->GCPhys + off, pvBuf, cb, enmOrigin);
            if (PGM_PHYS_RW_IS_SUCCESS(rcStrict2))
                PGM_PHYS_RW_DO_UPDATE_STRICT_RC(rcStrict, rcStrict2);
            else
            {
                /* Set the remaining buffer to a known value. */
                memset(pvBuf, 0xff, cbRead);
                PGM_UNLOCK(pVM);
                return rcStrict2;
            }
        }
        // [...]
    }
}
```

CVE-2023-21988 - Patched

```
VMMDECL(VBOXSTRICTRC) PGMPhysRead(PVMCC pVM, RTGCPHYS GCPhys, void *pvBuf, size_t cbRead, PGMACCESSORIGIN enmOrigin)
{
    // [...] Loop on each page
    {
        size_t    cb    = GUEST_PAGE_SIZE - (off & GUEST_PAGE_OFFSET_MASK);
        if (cb > cbRead)
            cb = cbRead;

        // Is a MMIO Page
        if ( PGM_PAGE_HAS_ACTIVE_ALL_HANDLERS(pPage) || PGM_PAGE_IS_SPECIAL_ALIAS_MMIO(pPage))
        {
            // call MMIO handler
            VBOXSTRICTRC rcStrict2 = pgmPhysReadHandler(pVM, pPage, pRam->GCPhys + off, pvBuf, cb, enmOrigin);
            if (PGM_PHYS_RW_IS_SUCCESS(rcStrict2))
                PGM_PHYS_RW_DO_UPDATE_STRICT_RC(rcStrict, rcStrict2);
            else
            {
                /* Set the remaining buffer to a known value. */
                memset(pvBuf, 0xff, cbRead);
                PGM_UNLOCK(pVM);
                return rcStrict2;
            }
        }
        // [...]
    }
}
```

- What's happening there ?

Pushing the issue deeper

- **pgmPhysReadHandler**
  - Function that will call the appropriate MMIO handler for the given GPA
- How does a MMIO handler looks like ?
  - A lot of different devices, a lot of different MMIO handlers
  - Is supposed to fill the provided buffer depending on the given GPA
  - Are they all doing it ?

Pushing the issue deeper

- **pgmPhysReadHandler**
  - Function that will call the appropriate MMIO handler for the given GPA
- How does a MMIO handler looks like ?
  - A lot of different devices, a lot of different MMIO handlers
  - Is supposed to fill the provided buffer depending on the given GPA
  - Are they all doing it ?

```
/**
 * @callback_method_impl{FNIOMMMIONEWREAD}
 */
static DECLCALLBACK(VBOXSTRICTRC) buslogicMMIORead(PPDMDEVINS pDevIns, void *pvUser, RTGCPHYS off, void *pv, unsigned cb)
{
    RT_NOREF(pDevIns, pvUser, off, pv, cb);

    /* the linux driver does not make use of the MMIO area. */
    ASSERT_GUEST_MSG_FAILED(("MMIO Read: %RGp LB %u\n", off, cb));
    return VINF_SUCCESS;
}
```

Pushing the issue deeper

- **pgmPhysReadHandler**
  - Function that will call the appropriate MMIO handler for the given GPA
- How does a MMIO handler looks like ?
  - A lot of different devices, a lot of different MMIO handlers
  - Is supposed to fill the provided buffer depending on the given GPA
  - Are they all doing it ?

```
/**
 * @callback_method_impl{FNIOMMMIONEWREAD}
 */
static DECLCALLBACK(VBOXSTRICTRC) buslogicMMIORead(PPDMDEVINS pDevIns, void *pvUser, RTGCPHYS off, void *pv, unsigned cb)
{
    RT_NOREF(pDevIns, pvUser, off, pv, cb);

    /* the linux driver does not make use of the MMIO area. */
    ASSERT_GUEST_MSG_FAILED(("MMIO Read: %RGp LB %u\n", off, cb));
    return VINF_SUCCESS;
}
```

- Nope !

CVE-2024-21121

```
VMMDECL(VBOXSTRICTRC) PGMPhysRead(PVMCC pVM, RTGCPHYS GCPhys, void *pvBuf, size_t cbRead, PGMACCESSORIGIN enmOrigin)
{
    // [...] Loop on each page
    {
        size_t    cb    = GUEST_PAGE_SIZE - (off & GUEST_PAGE_OFFSET_MASK);
        if (cb > cbRead)
            cb = cbRead;

        // Is a MMIO Page
        if ( PGM_PAGE_HAS_ACTIVE_ALL_HANDLERS(pPage) || PGM_PAGE_IS_SPECIAL_ALIAS_MMIO(pPage))
        {
            // call MMIO handler
            VBOXSTRICTRC rcStrict2 = pgmPhysReadHandler(pVM, pPage, pRam->GCPhys + off, pvBuf, cb, enmOrigin);
            if (PGM_PHYS_RW_IS_SUCCESS(rcStrict2))
                PGM_PHYS_RW_DO_UPDATE_STRICT_RC(rcStrict, rcStrict2);
            else
            {
                /* Set the remaining buffer to a known value. */
                memset(pvBuf, 0xff, cbRead);
                PGM_UNLOCK(pVM);
                return rcStrict2;
            }
        }
        // [...]
    }
}
```

- No error during the callback
- **pvBuf** still not initialized

CVE-2024-21121

- Found a variant of the bug
  - Can use the same exploit technique as CVE-2023-21988
- Requires to find specific MMIO read handlers
  - Must return a success without fully initializing the buffer
  - Must be registered with the flag **IOMMMIO_FLAGS_READ_PASSTHRU**
    - Allow the MMIO handler to be called for any size instead of only 1/2/4
- The MMIO handler for the BusLogic device fits perfectly
  - Hard disk technology
- We have our leak !
  - And can defeat ASLR

Plan

01 Finding the map

02 Reaching high seas

03 Exploring the ocean

04 Hunting for gold

05 Stealing the treasure

06 Making Port

Reaching high seas

- Hypervisors have a HUGE code base, you can’t audit everything
  - Very time consuming to fully understand an attack surface from top to bottom
  - We don’t have this time ! How to chose where to look ?
- Use knowledge acquired during SOTA to find “interesting” code
  - Vulnerability patterns
  - Attack surfaces with a lot of past bugs
- Use tools !
  - grep
- Find a list of things to look at deeper
  - Low quality code
  - Attack surfaces not identified during SOTA

Reaching high seas

- But was not a great success on VirtualBox code base
- Too much false positives
  - Vulnerabilities only accessible in the weirdest configurations
  - Non exploitable / reachable bugs
  - Code that felt weird but was fine
- Spent too much time on those
- But allowed me to explore a lot of different code
  - Acquired knowledge on the code base
  - Found interesting attack surfaces to look at from top to bottom !

Reaching high seas

- Decided to chose the VirtIO devices implementation
  - Specification for a paravirtualization interface for multiple devices
  - Implemented in a lot of hypervisors
  - VirtualBox implements the VirtIO Disk and Network card

- VirtualBox's implementation can be compared to others
  - And the code felt a bit weird…

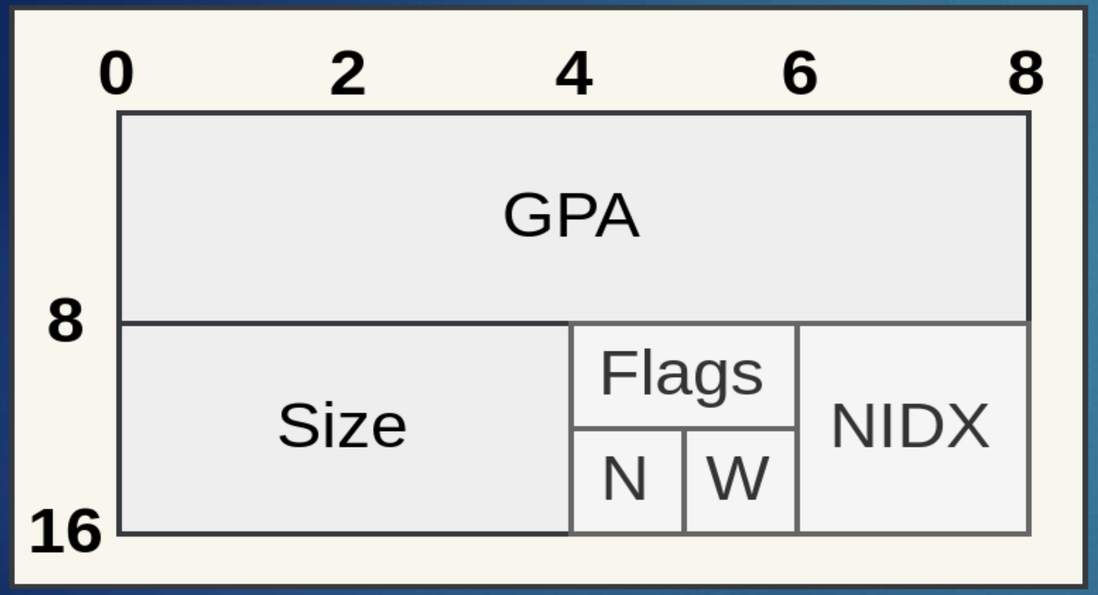
Reaching high seas

```
#ifdef VIRTIO_VBUF_ON_STACK
                PVIRTQBUF pVirtqBuf = virtioCoreR3VirtqBufAlloc();
                if (!pVirtqBuf)
                {
                    LogRel(("Failed to allocate memory for VIRTQBUF\n"));
                    break;  /* No point in trying to allocate memory for other descriptor chains */
                }
                int rc = virtioCoreR3VirtqAvailBufGet(pDevIns, &pThis->Virtio, uVirtqNbr,
                                                      pWorkerR3->auRedoDescs[i], pVirtqBuf);
#else /* !VIRTIO_VBUF_ON_STACK */
                PVIRTQBUF pVirtqBuf;
                int rc = virtioCoreR3VirtqAvailBufGet(pDevIns, &pThis->Virtio, uVirtqNbr,
                                                      pWorkerR3->auRedoDescs[i], &pVirtqBuf);
#endif /* !VIRTIO_VBUF_ON_STACK */
```

VirtIO queues

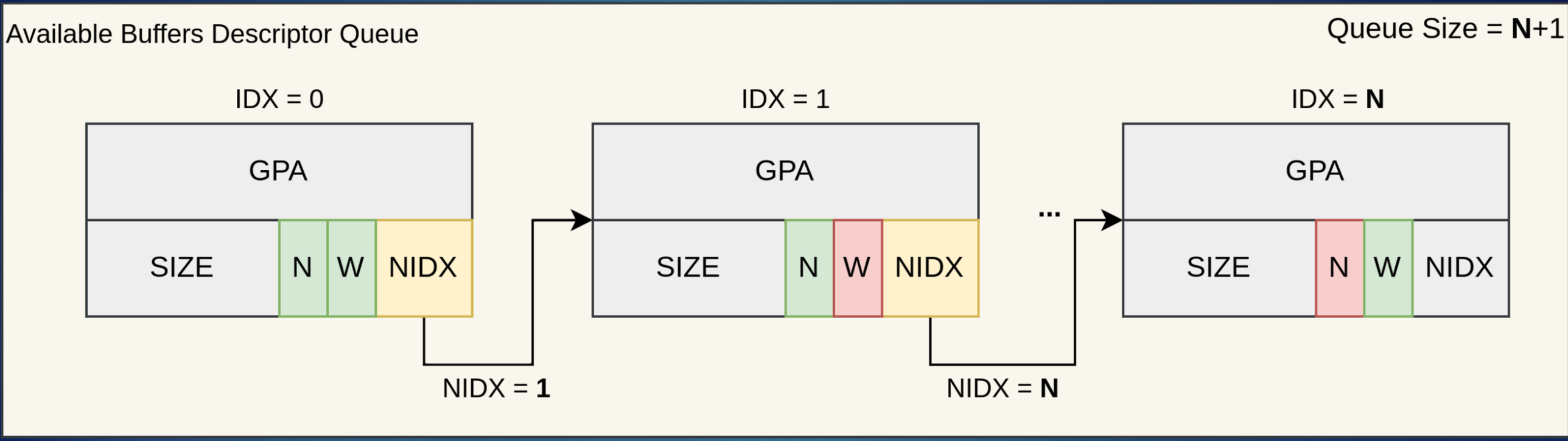
- VirtIO Queues is a mechanism to send and receive data to and from the guest
  - Implemented in the core of VirtIO
  - used by all VirtIO devices
- Problematic: want to send a lot of data between guest and host
  - Cannot use a single contiguous buffer of physical memory
- A very common way to do this is to use a queue of segment descriptors
  - A segment represents a chunk of contiguous physical memory to use
- Each segment is described by
  - A Guest Physical Address
  - A size

VirtIO queue descriptors



- Additional flags
  - **VIRTQ_DESC_F_NEXT**
    - The descriptor chain is not over
    - Get the next descriptor at index **NIDX**
  - **VIRTQ_DESC_F_WRITE**
    - The buffer must be used only for writing

VirtIO queue descriptors chain



VirtIO – VBox implementation

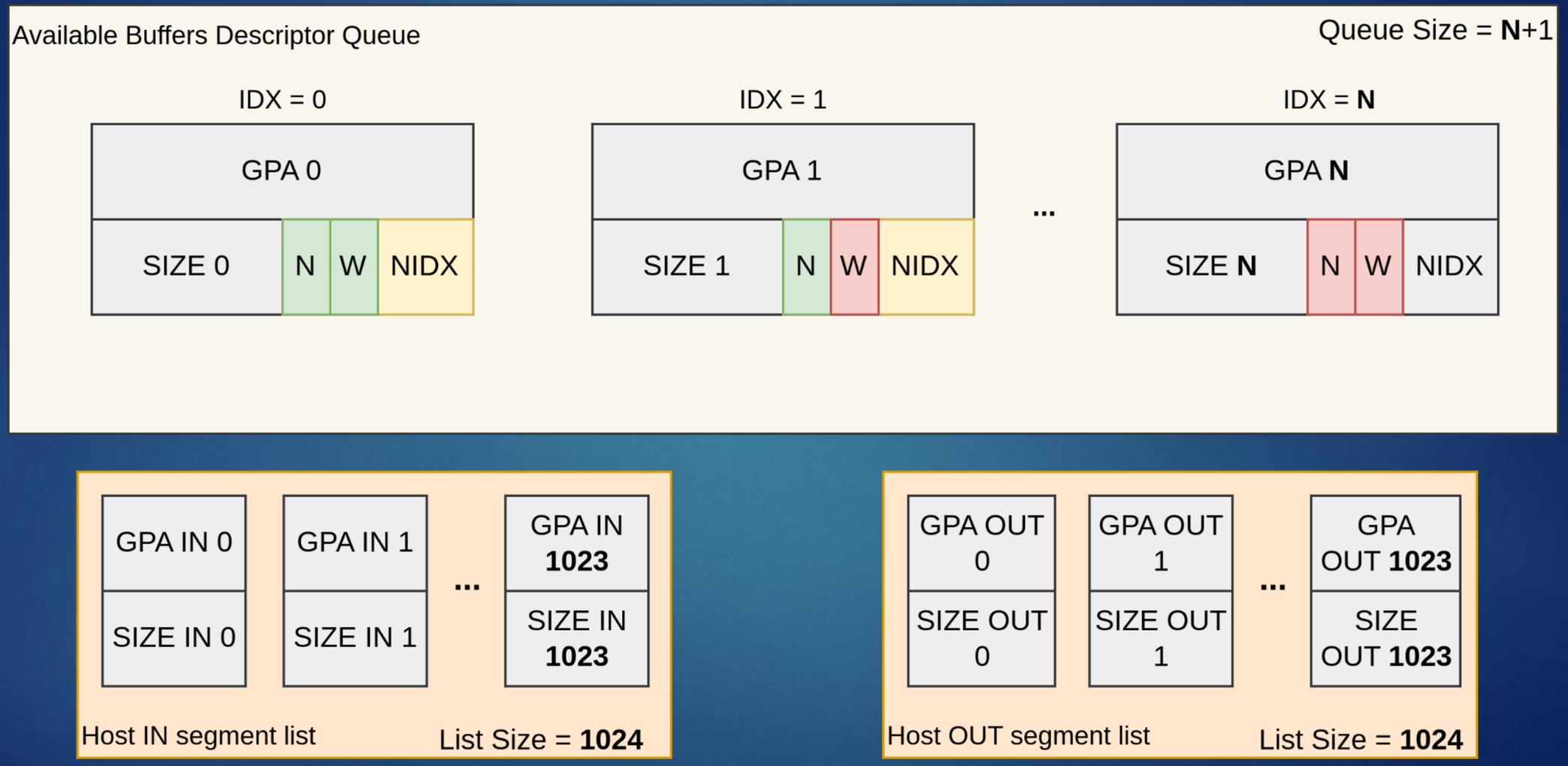
- Function **virtioCoreR3VirtqAvailBufGet**
  - Responsible for parsing a descriptor chain
  - Place it in the **VIRTQBUF** passed in parameter
    - Contains a list of segments

```
typedef struct VIRTQBUF
{
    // [...]
    VIRTIOSGSEG          aSegsIn[1024];
    VIRTIOSGSEG          aSegsOut[1024];
} VIRTQBUF_T;

typedef struct VIRTIOSGSEG /**< An S/G entry */
{
    uint64_t GCPhys; /**< Pointer to the segment buffer */
    size_t  cbSeg;  /**< Size of the segment buffer */
} VIRTIOSGSEG; // Total size : 0x10
```

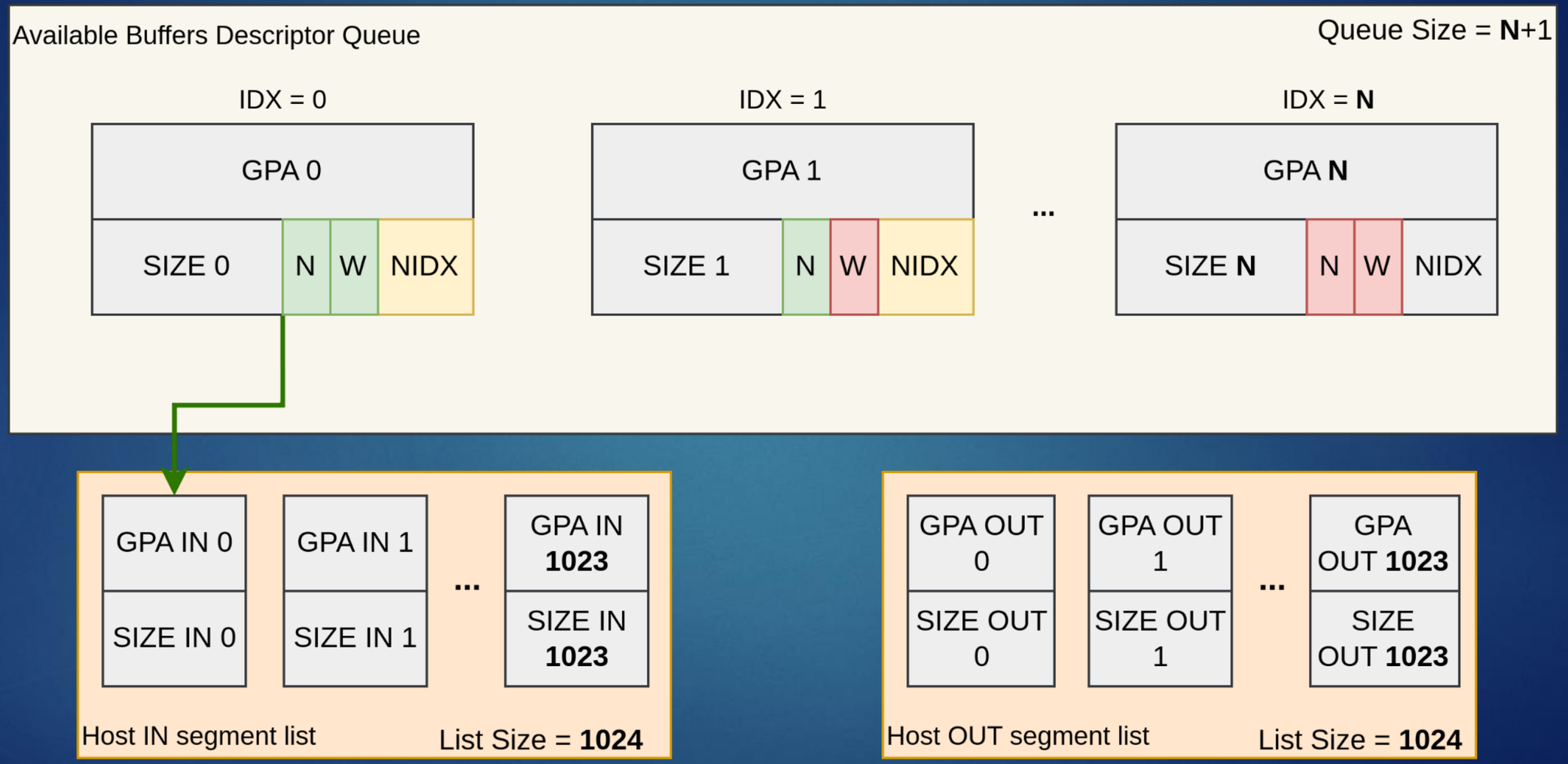
VirtIO – VBox implementation

```
int virtioCoreR3VirtqAvailBufGet(PPDMDEVINS pDevIns, PVIRTIOCORE pVirtio, uint16_t uVirtq,
    uint16_t uHeadIdx, PVIRTQBUF pVirtqBuf)
{
    // [...]
    uint32_t cSegsIn, cSegsOut  = 0;
    PVIRTIOSGSEG paSegsIn  = pVirtqBuf->aSegsIn;
    PVIRTIOSGSEG paSegsOut = pVirtqBuf->aSegsOut;

    do
    {
        PVIRTIOSGSEG pSeg;
        if (cSegsIn + cSegsOut >= pVirtq->uQueueSize)
        {
            // [...] Error log
            break;
        }

        virtioReadDesc(pDevIns, pVirtio, pVirtq, uDescIdx, &desc);

        // simplified version of the result
        if (desc.fFlags & VIRTQ_DESC_F_WRITE)
            pSeg = &paSegsIn[cSegsIn++];
        else
            pSeg = &paSegsOut[cSegsOut++];

        pSeg->GCPhys = desc.GCPhysBuf;
        pSeg->cbSeg = desc.cb;
        uDescIdx = desc.uDescIdxNext;
    } while (desc.fFlags & VIRTQ_DESC_F_NEXT);
}
```

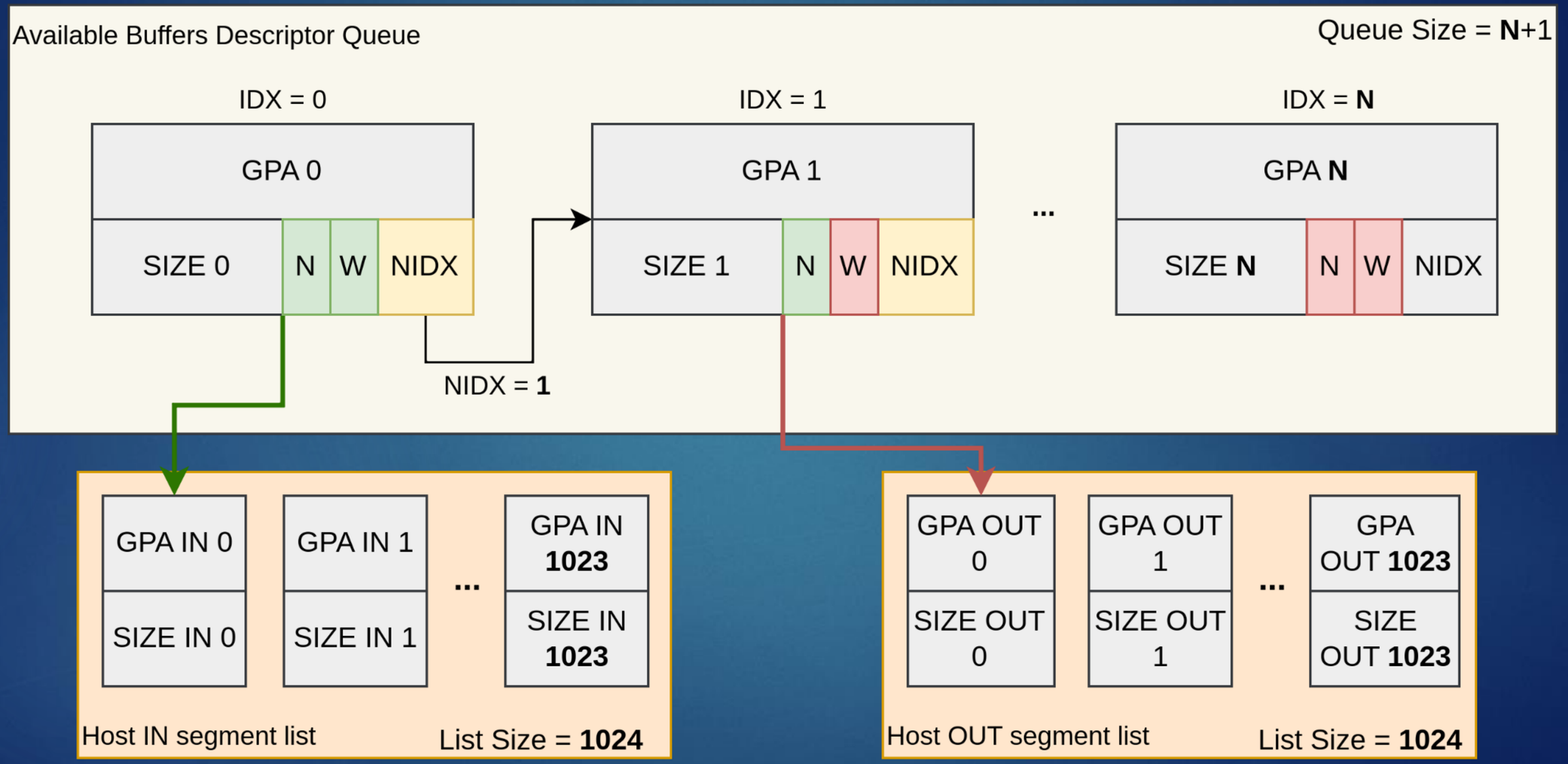
VirtIO – VBox implementation



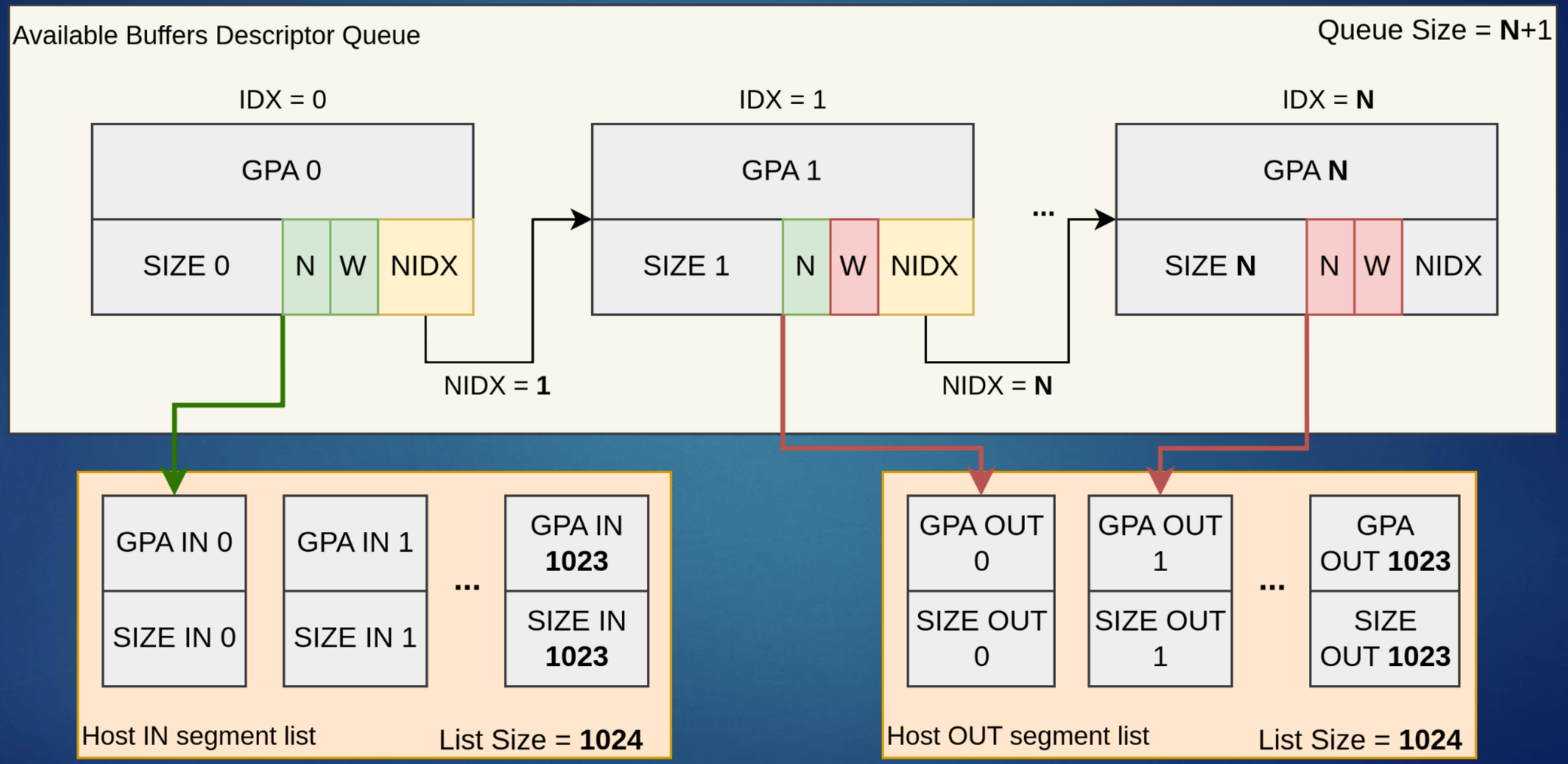
VirtIO – VBox implementation



VirtIO – VBox implementation



VirtIO – VBox implementation



VirtIO – VBox implementation

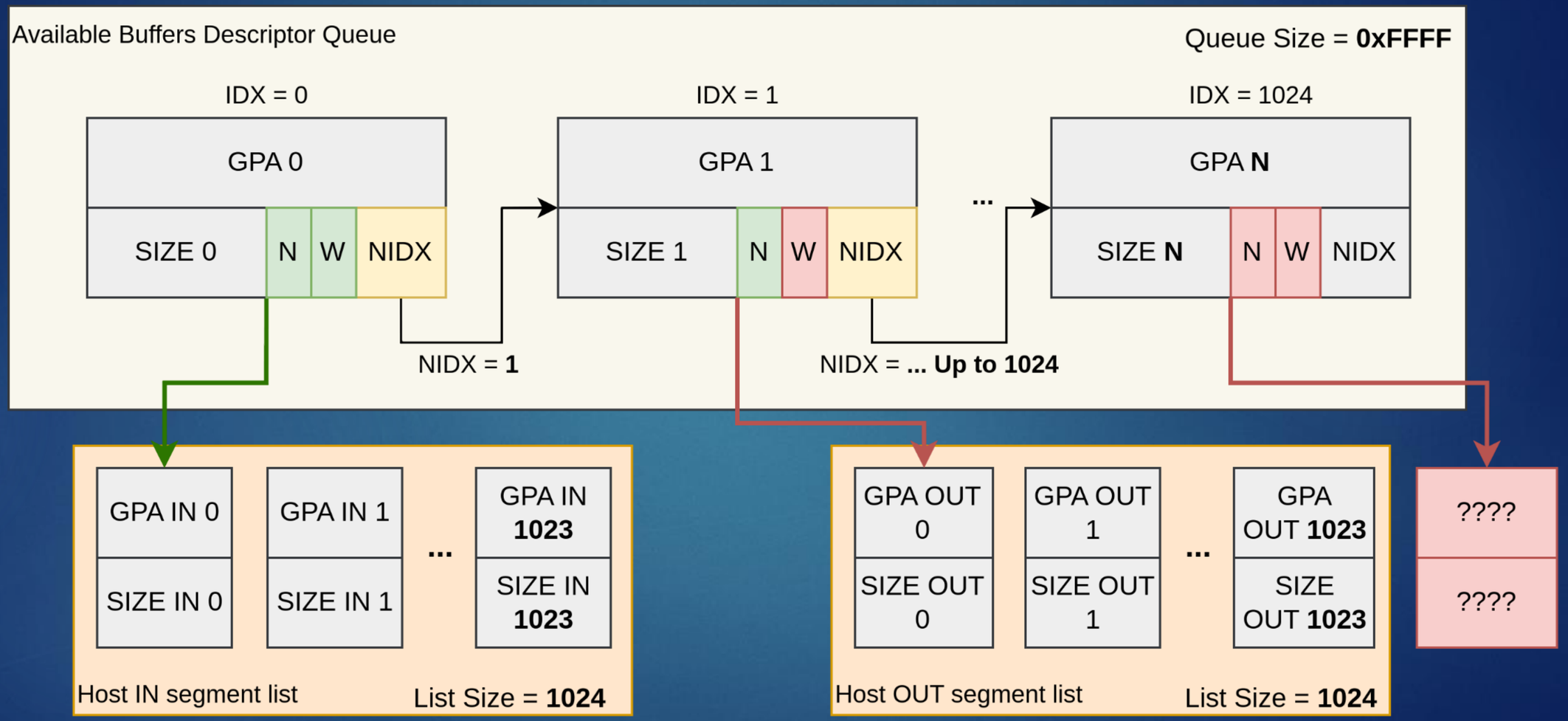
```
int virtioCoreR3VirtqAvailBufGet(PPDMDEVINS pDevIns, PVIRTIOCORE pVirtio, uint16_t uVirtq,
    uint16_t uHeadIdx, PVIRTQBUF pVirtqBuf)
{
    // [...]
    uint32_t cSegsIn, cSegsOut  = 0;
    PVIRTIOSGSEG paSegsIn  = pVirtqBuf->aSegsIn;
    PVIRTIOSGSEG paSegsOut = pVirtqBuf->aSegsOut;

    do
    {
        PVIRTIOSGSEG pSeg;
        if (cSegsIn + cSegsOut >= pVirtq->uQueueSize)
        {
            // [...] Error log
            break;
        }

        virtioReadDesc(pDevIns, pVirtio, pVirtq, uDescIdx, &desc);

        // simplified version of the result
        if (desc.fFlags & VIRTQ_DESC_F_WRITE)
            pSeg = &paSegsIn[cSegsIn++];
        else
            pSeg = &paSegsOut[cSegsOut++];

        pSeg->GCPhys = desc.GCPhysBuf;
        pSeg->cbSeg = desc.cb;
        uDescIdx = desc.uDescIdxNext;
    } while (desc.fFlags & VIRTQ_DESC_F_NEXT);
}
```

- Only error stop condition

CVE-2024-21114 – Root cause

- **uQueueSize** is NOT fixed !
  - Default is 1024…
- But can be changed by writing into the MMIO
  - To any value on 16 bits
  - Maximum **0xFFFF**

CVE-2024-21114 – Root cause



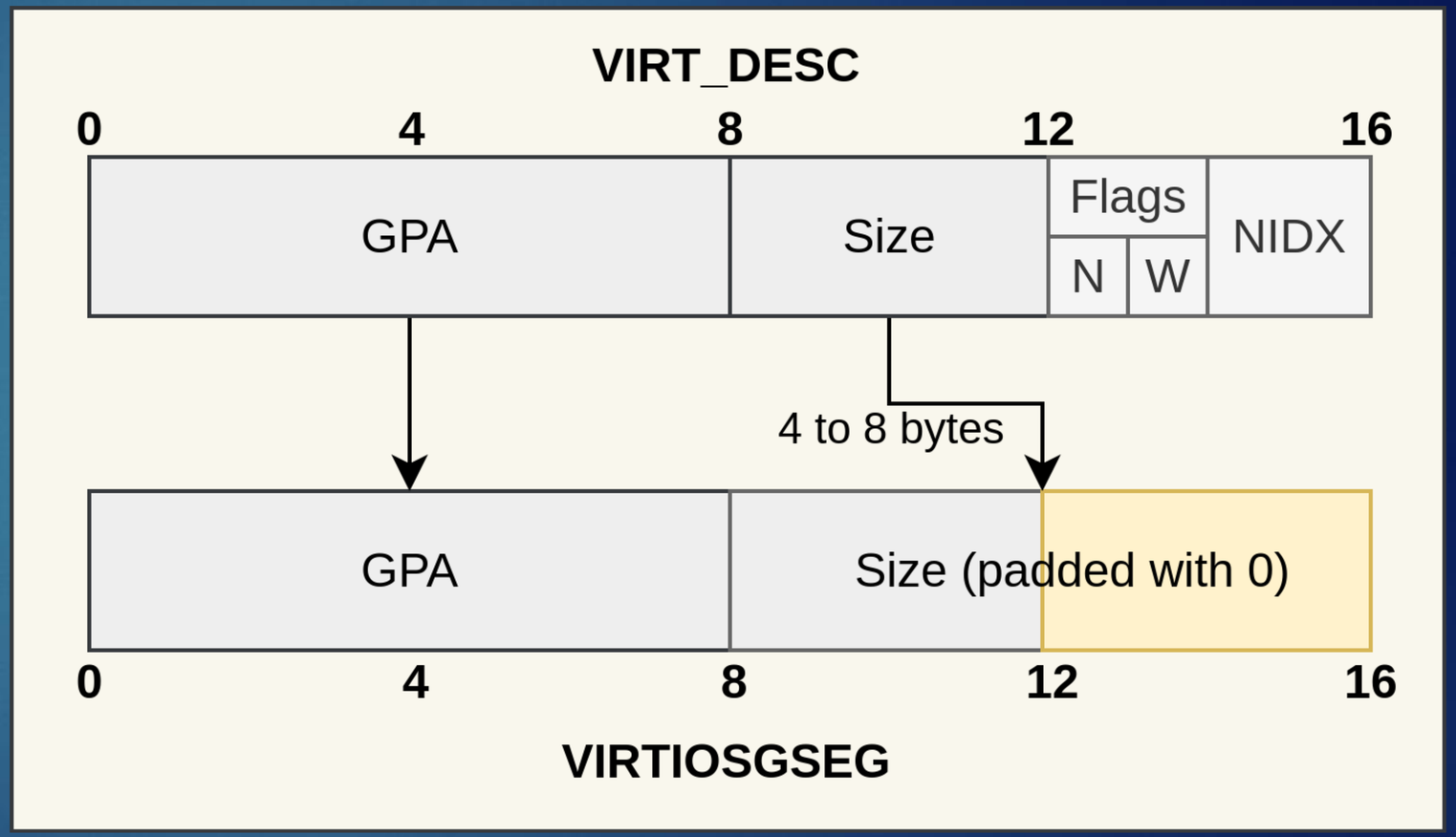
CVE-2024-21114 – Root cause

- The host fails to properly check if there are too many descriptors in the list
- Can write up to 0xFFFF segments in a list of size 1024
  - OOB write after the **VIRTQBUF** structure passed in parameter

```
typedef struct VIRTQBUF
{
    // [...]
    VIRTIOSGSEG          aSegsIn[1024];
    VIRTIOSGSEG          aSegsOut[1024];
} VIRTQBUF_T;

typedef struct VIRTIOSGSEG /**< An S/G entry */
{
    uint64_t GCPhys; /**< Pointer to the segment buffer */
    size_t  cbSeg;  /**< Size of the segment buffer */
} VIRTIOSGSEG; // Total size : 0x10
```

CVE-2024-21114 – Impact

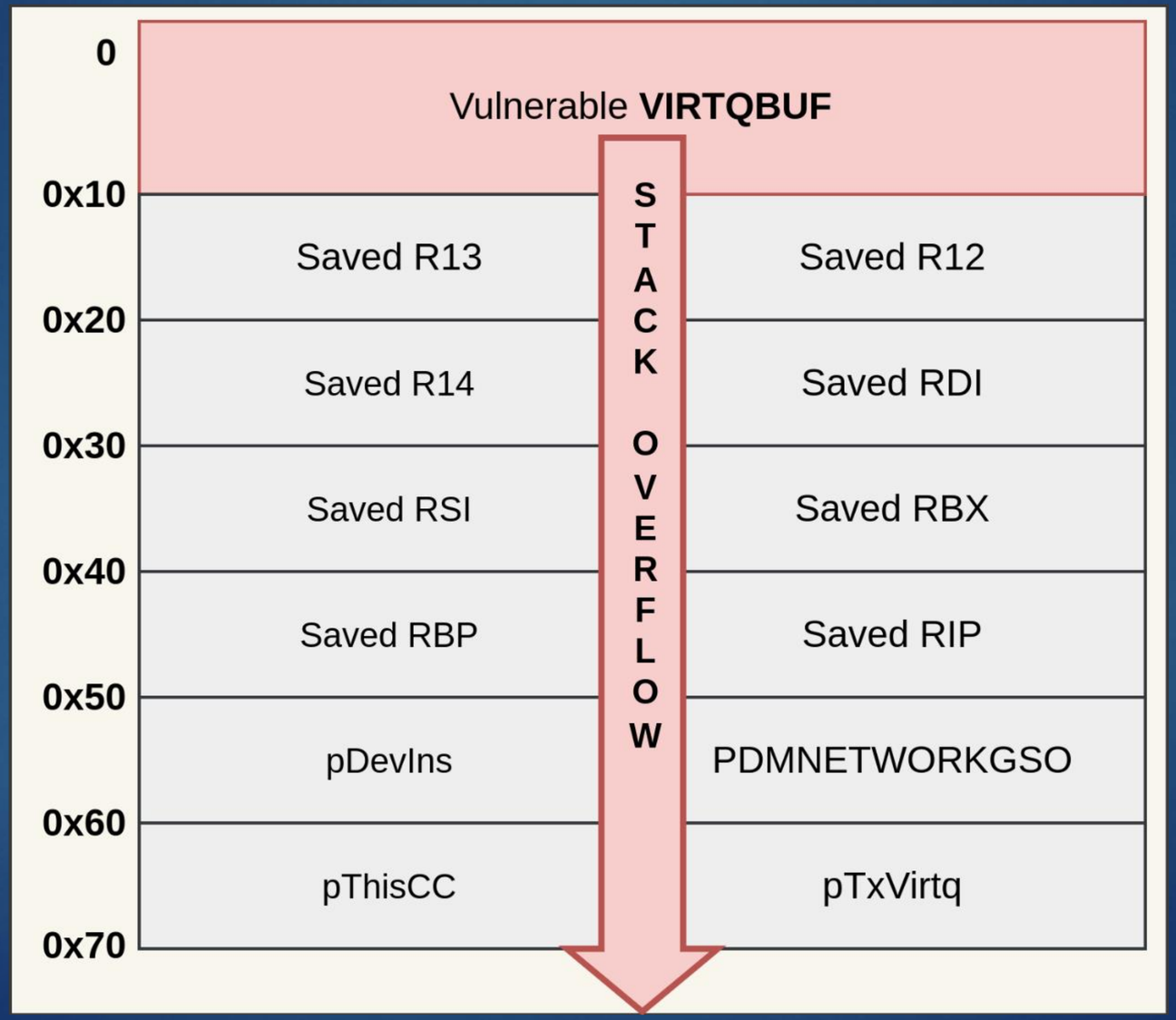
- The **VIRTQBUF** structure can be located on the stack or in the heap
  - Decide to go with the stack buffer overflow exploit
- Vulnerability allows to write chunks of 16 bytes in OOB
  - But only 12 are controlled, 4 last bytes are 0



CVE-2024-21114 – Exploit

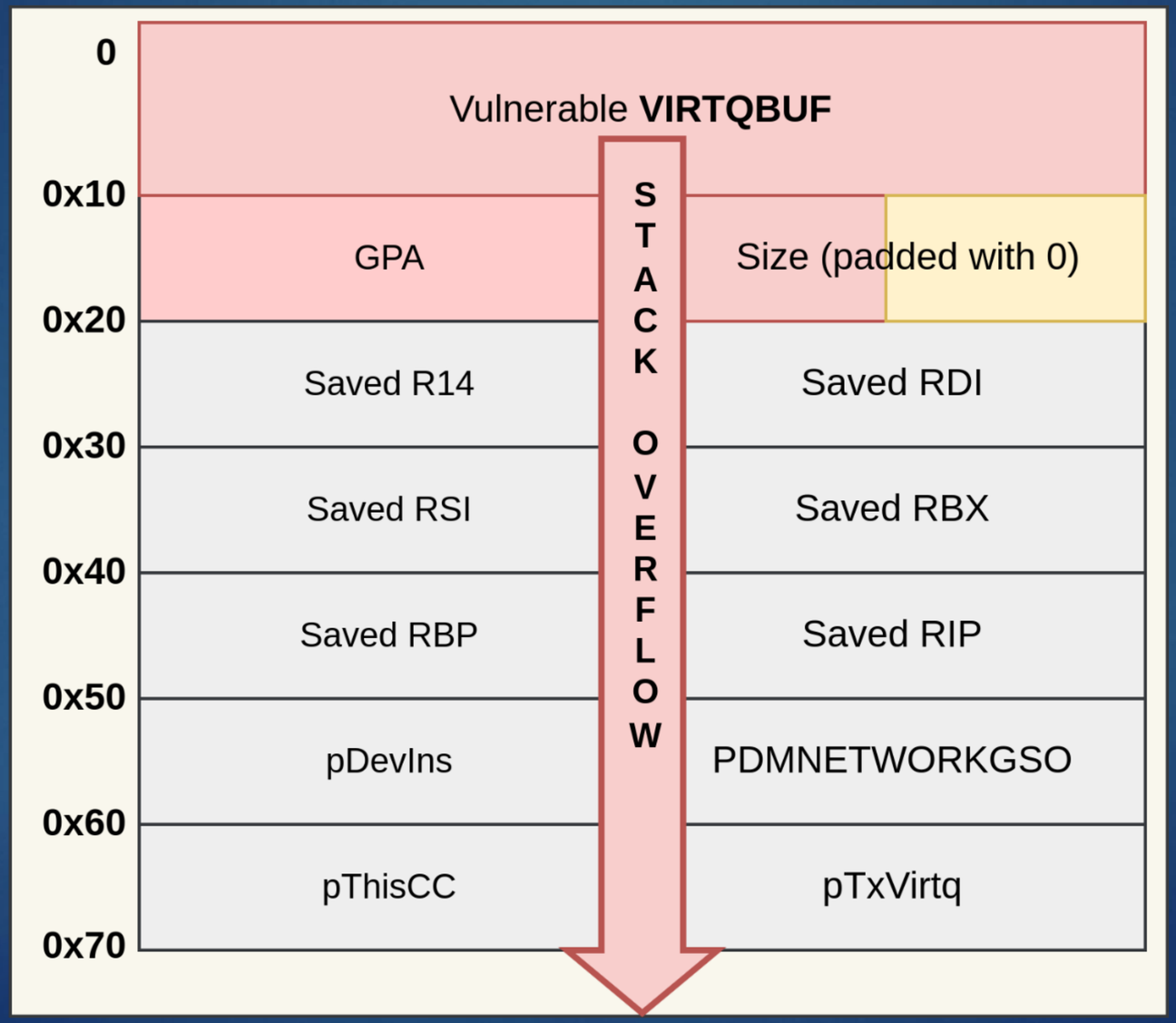
- Can be triggered from the function **virtioNetR3TransmitPkts**
  - In VirtIO network card implementation
- ASLR is defeated thanks to the exploited leak
  - CVE-2024-21121
- VirtualBox compiled without stack canaries
  - Easy win ?

CVE-2024-21114 – Exploit



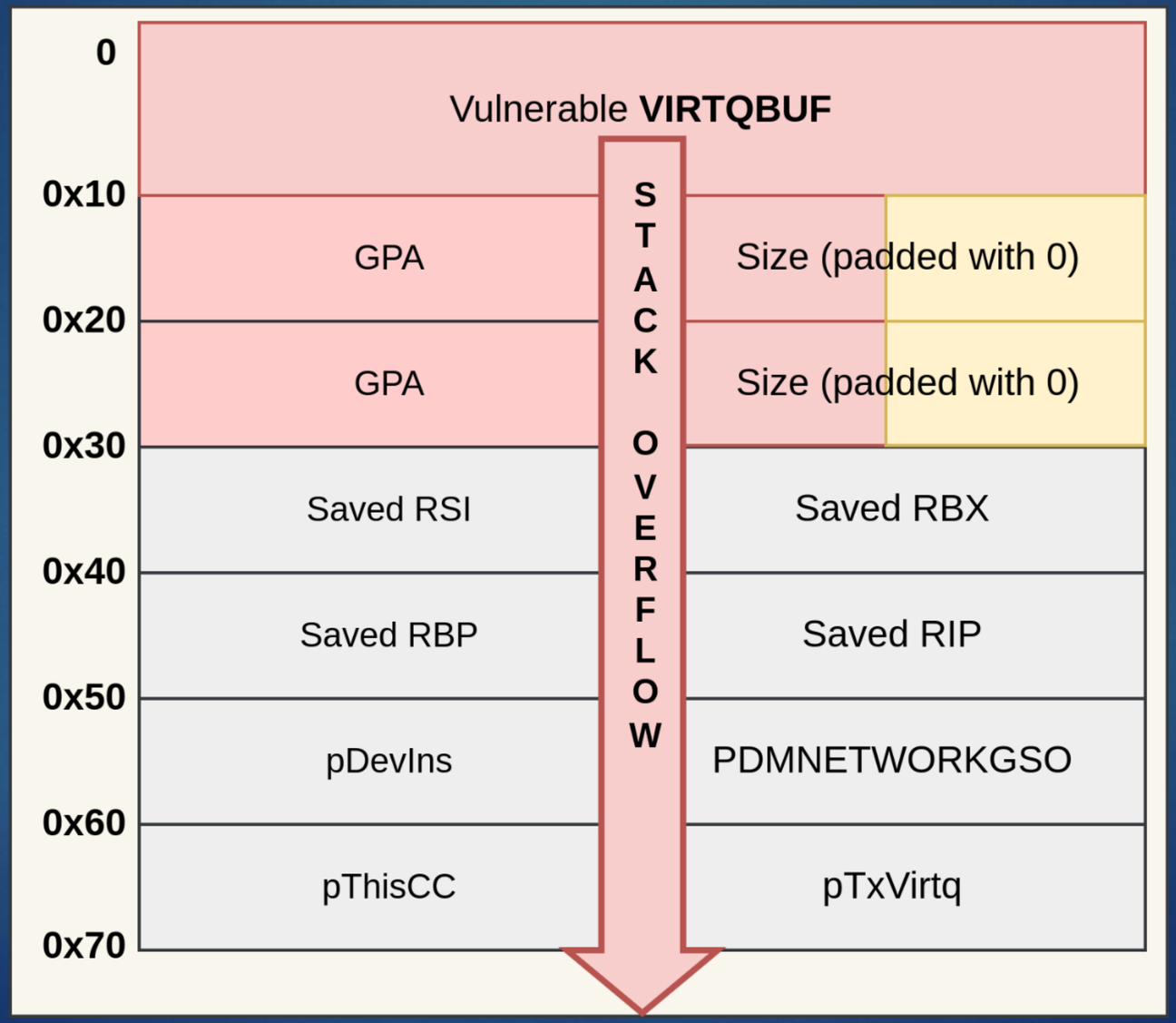
- Stack frame of **virtioNetR3TransmitPkts**

CVE-2024-21114 – Exploit



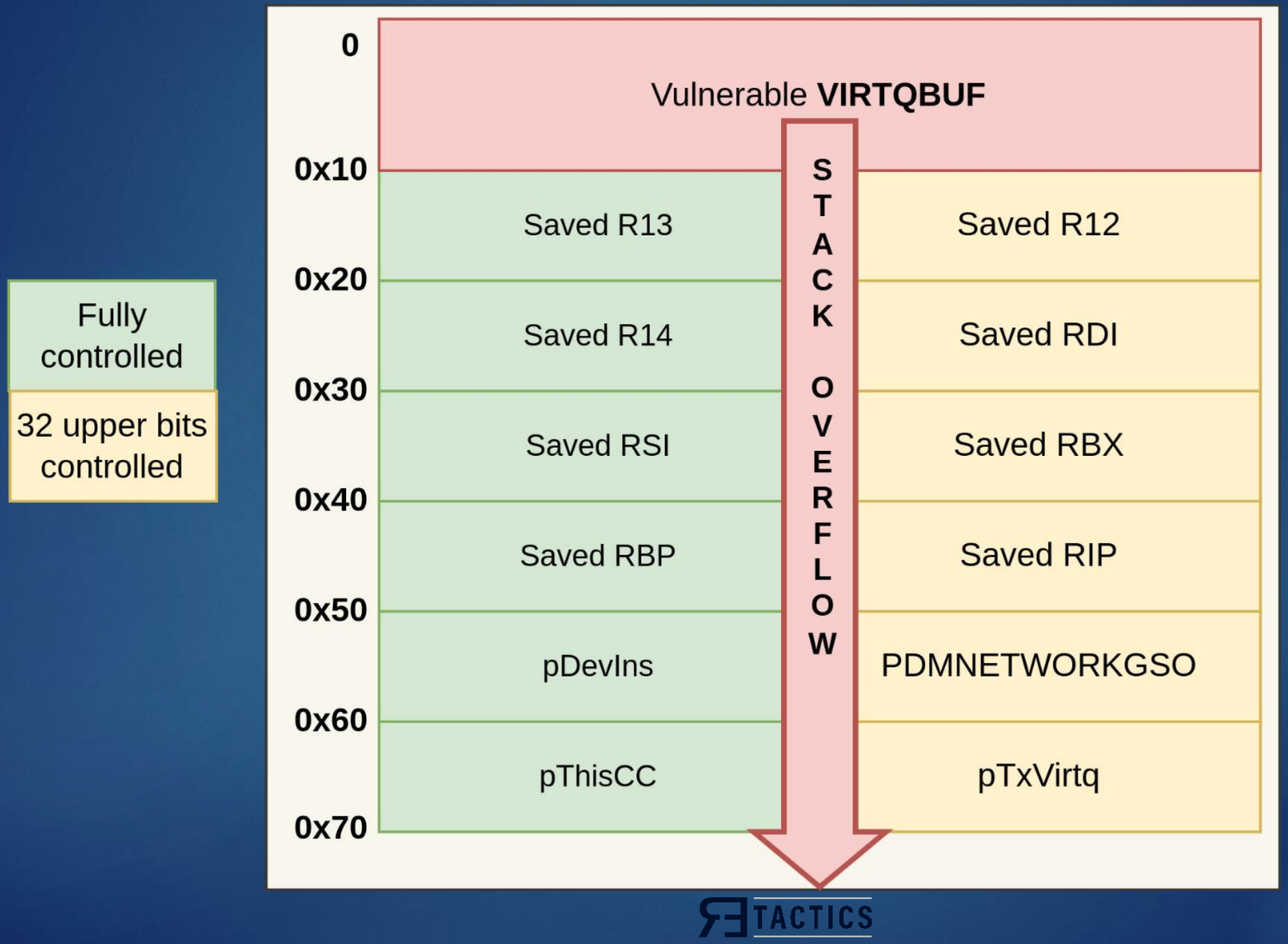
- Stack frame of **virtioNetR3TransmitPkts**

CVE-2024-21114 – Exploit



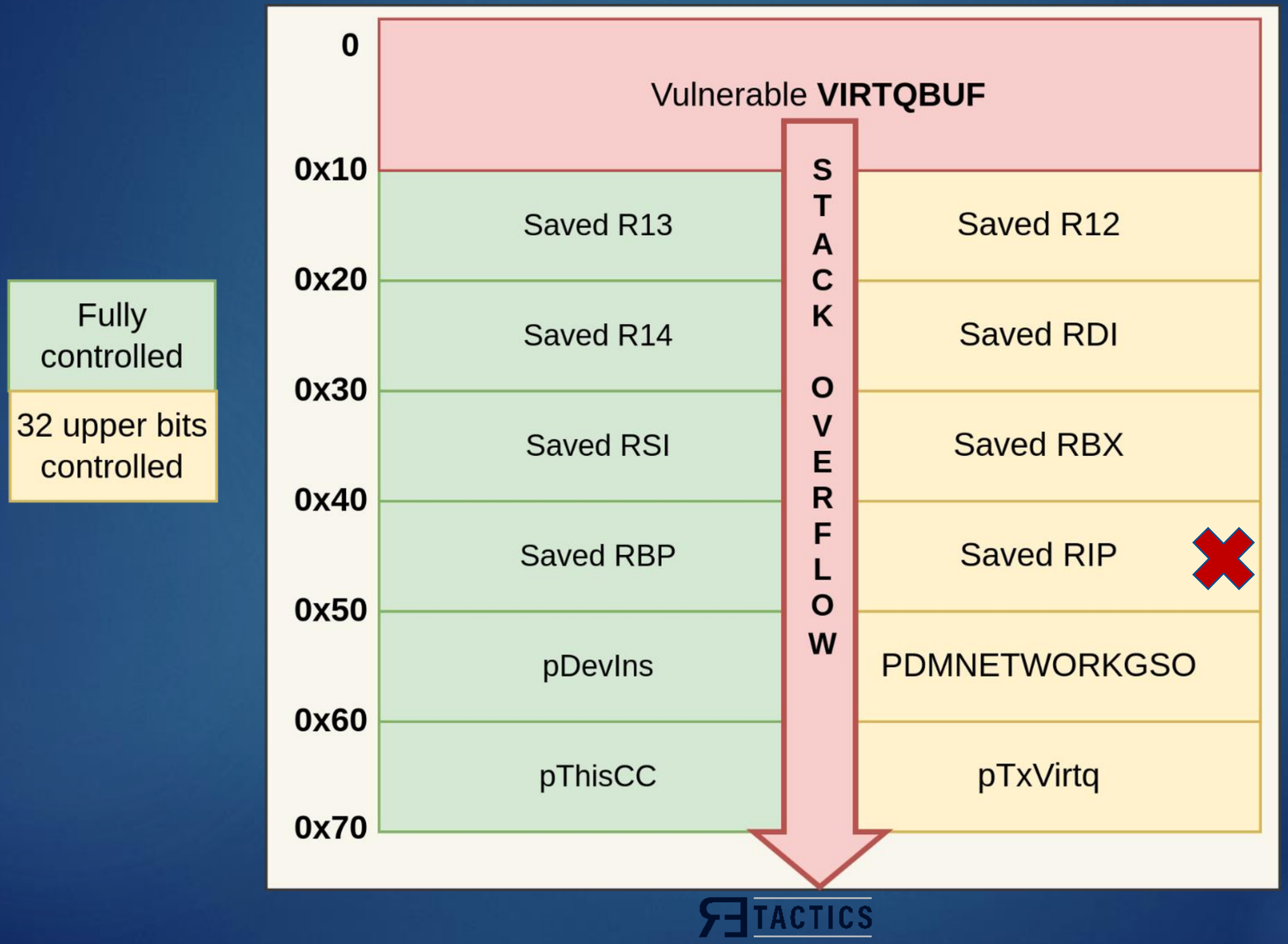
- Stack frame of **virtioNetR3TransmitPkts**

CVE-2024-21114 – Exploit



- Stack frame of **virtioNetR3TransmitPkts**

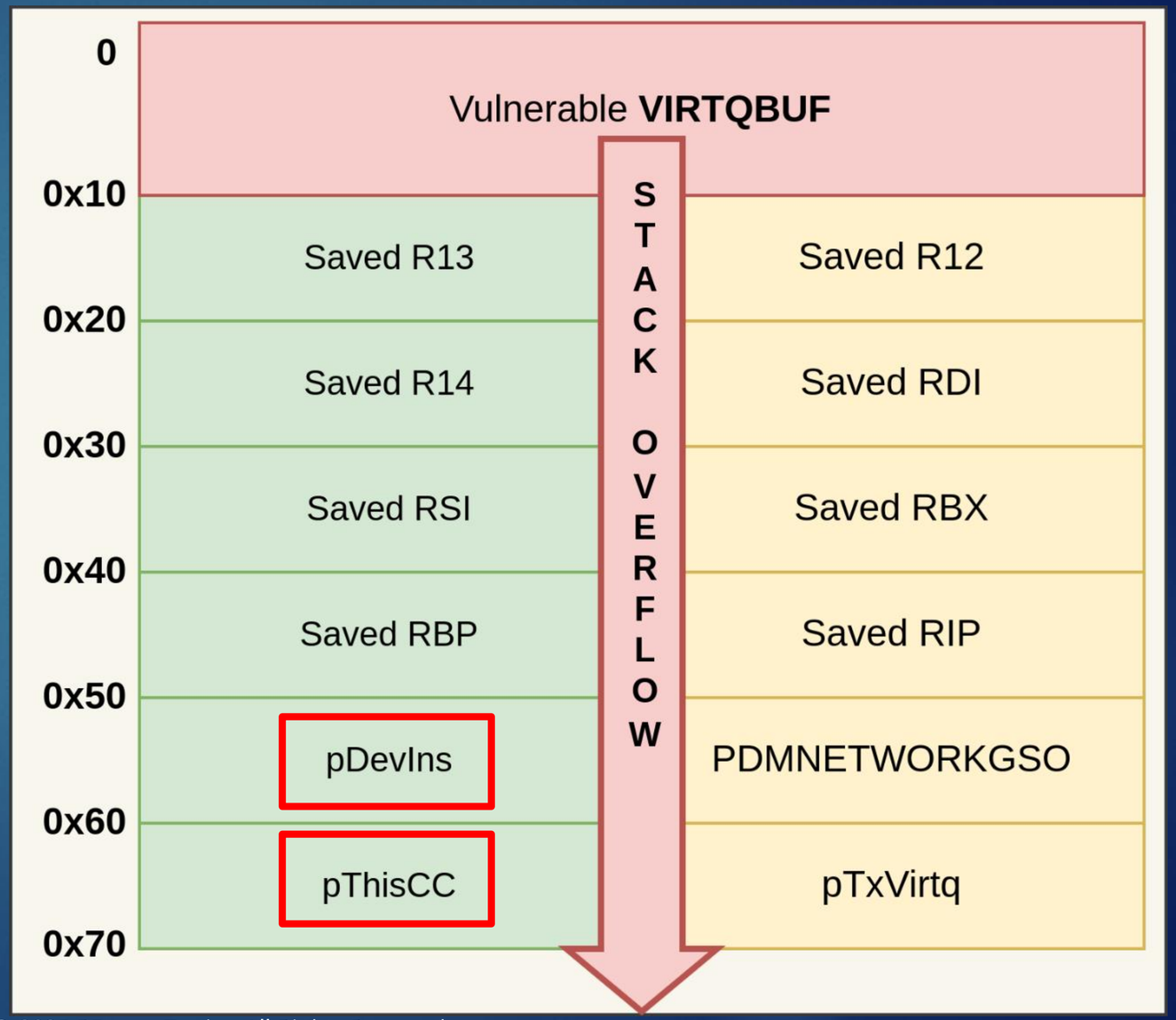
CVE-2024-21114 – Exploit



- Stack frame of **virtioNetR3TransmitPkts**
- Can not fully control RIP
- Nothing interesting to control before RIP

CVE-2024-21114 – Exploit

- But two objects interesting to control after RIP
  - **pDevIns** and **pThisCC**
  - Arguments to the function
- Can both be used to have an arbitrary call
  - Before the function returns
  - Within the limits of CFG
- But function can’t return
  - RIP has been overwritten



Exploit – Capabilities

- Stack buffer overflow to 2 arbitrary calls
  - CFG: Can only call existing functions
  - Must never return
- Strategy
  - Use the first “arbitrary” call to trigger an arbitrary write
  - Use the second “arbitrary” call to call **Sleep** forever
    - Function will never return
    - Will not crash !
- From stack buffer overflow to arbitrary write
  - Can use it only one time
  - Thread is sleeping forever

Exploit – Capabilities

- Single arbitrary write

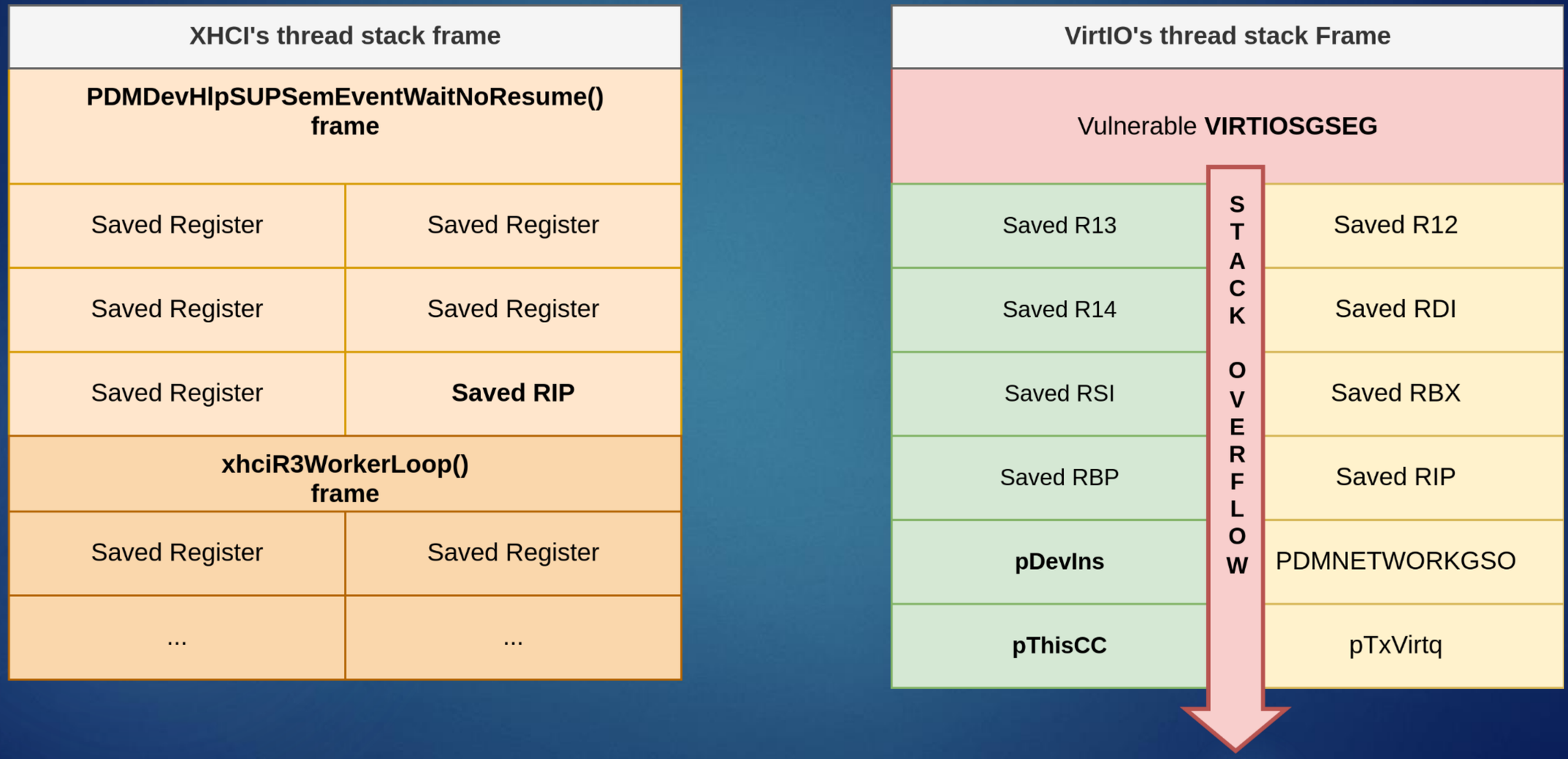
- ASLR is defeated thanks to the exploited leak
  - Can place arbitrary data at known location
  - Know the address of ROP gadgets
  - Know where the stack of the XHCI command thread is

Exploit

```
static DECLCALLBACK(int) xhciR3WorkerLoop(PPDMDEVINS pDevIns, PPDMTHREAD pThread)
{
    while (pThread->enmState == PDMTHREADSTATE_RUNNING)
    {
        // [..]
        if (!u32Tasks)
        {
            Assert(ASMAtomicReadBool(&pThis->fWrkThreadSleeping));
            rc = PDMDevHlpSUPSemEventWaitNoResume(pDevIns, pThis->hEvtProcess, RT_INDEFINITE_WAIT);
            AssertLogRelMsgReturn(RT_SUCCESS(rc) || rc == VERR_INTERRUPTED, ("%Rrc\n", rc), rc);
            if (RT_UNLIKELY(pThread->enmState != PDMTHREADSTATE_RUNNING))
                break;
            LogFlowFunc(("Woken up with rc=%Rrc\n", rc));
            u32Tasks = ASMAtomicXchgU32(&pThis->u32TasksNew, 0);
        }

        RTCritSectEnter(&pThisCC->CritSectThrd);

        if (pThis->crcr & XHCI_CRCR_CRR)
            xhciR3ProcessCommandRing(pDevIns, pThis, pThisCC);
        // [...]
    }
}
```

- Thread is waiting here
  - Semaphore
- Woke up when a command is sent by the guest

Exploit



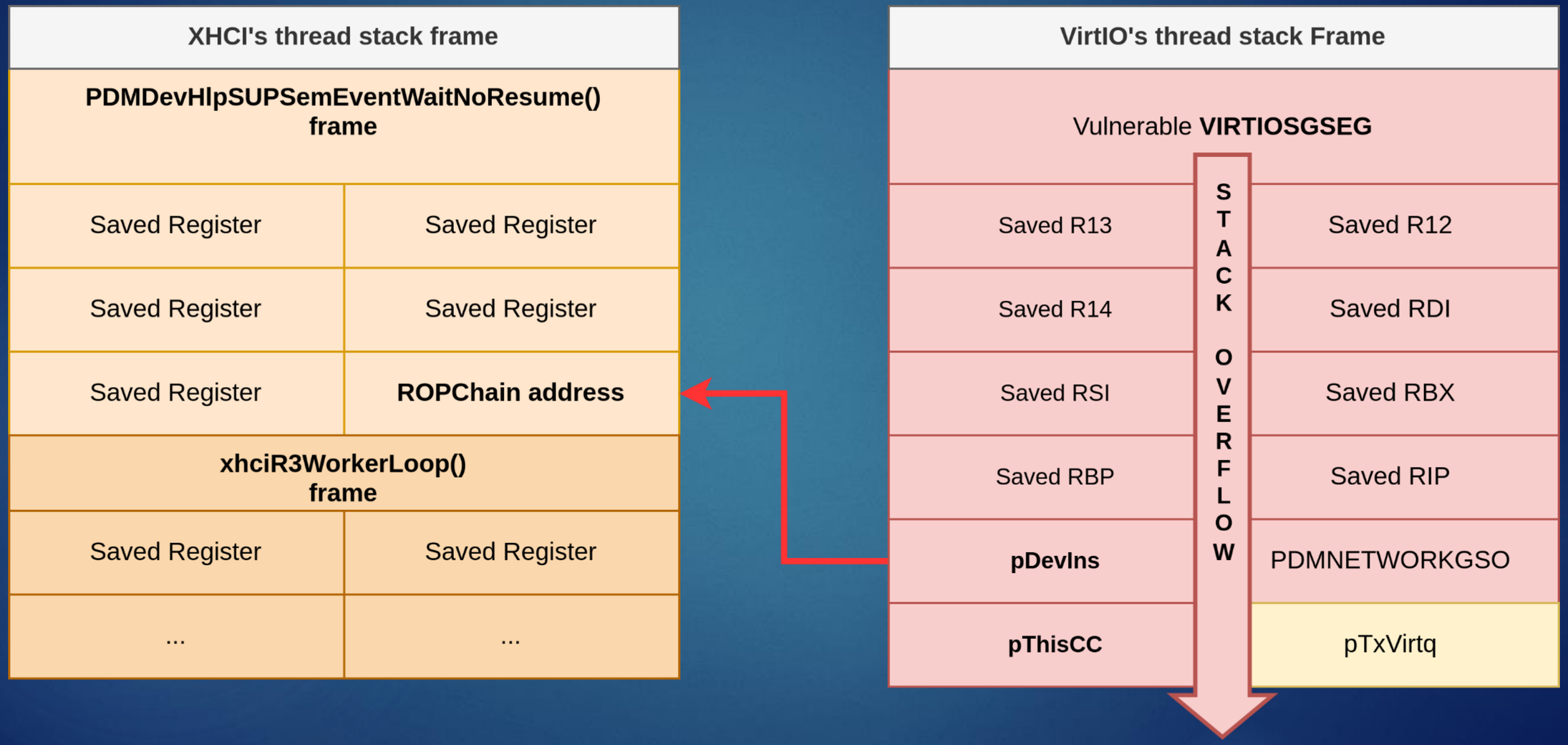
Exploit

XHCI's thread stack frame	
PDMDevHlpSUPSemEventWaitNoResume() frame	
Saved Register	Saved Register
Saved Register	Saved Register
Saved Register	**Saved RIP**
xhciR3WorkerLoop() frame	
Saved Register	Saved Register
...	...

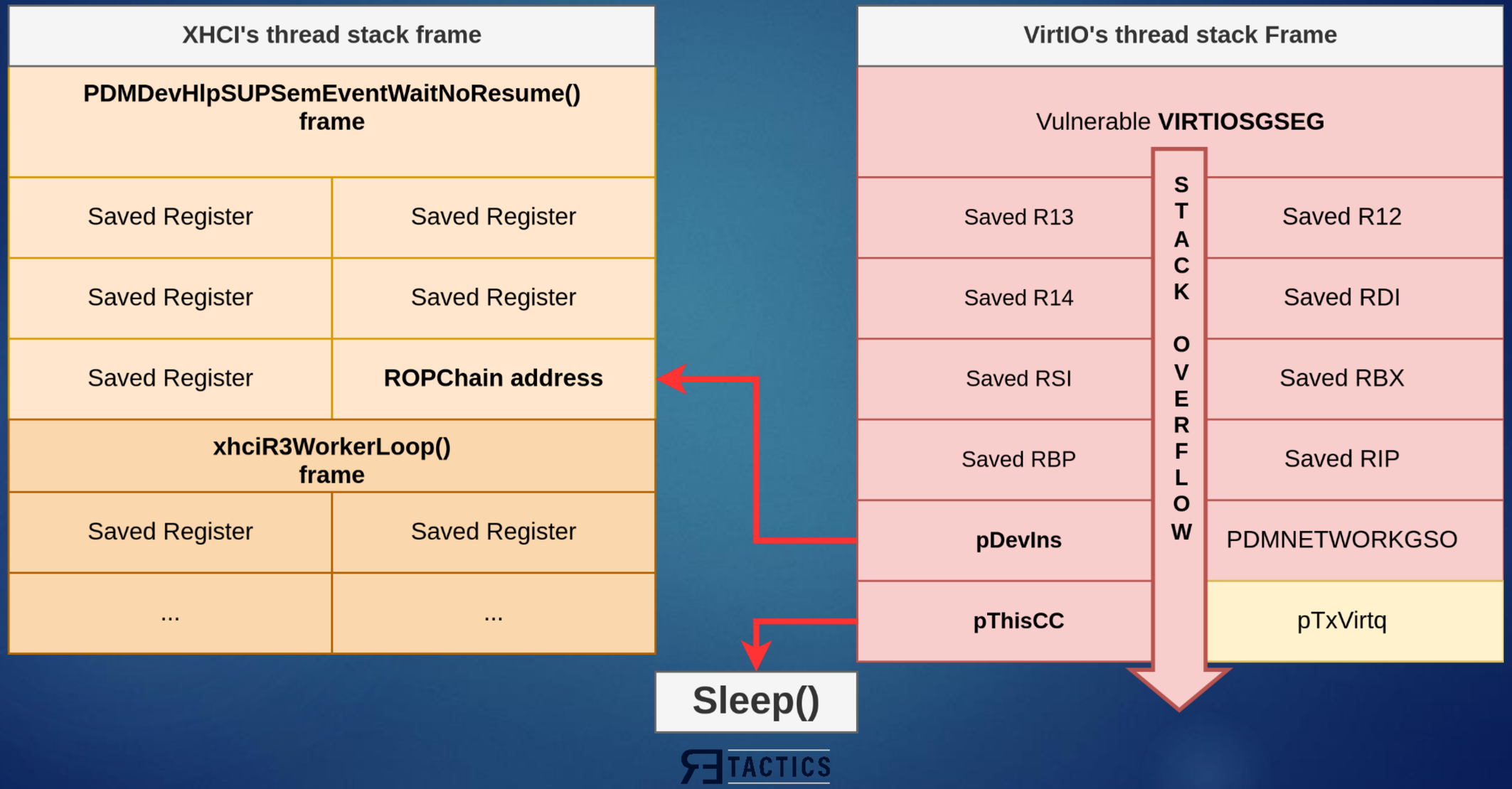
VirtIO's thread stack Frame	
Vulnerable **VIRTIOSGSEG**	
Saved R13	Saved R12
Saved R14	Saved RDI
Saved RSI	Saved RBX
Saved RBP	Saved RIP
pDevIns	PDMNETWORKGSO
pThisCC	pTxVirtq

STACK OVERFLOW

Exploit



Exploit



Pivoting to LPE

- ROP to shellcode
  - Arbitrary Code Execution with MEDIUM privileges
- Shellcode loads arbitrary executable from guest
  - Using segment descriptor queue
  - Write it on the host
- Use CreateProcess to start the chained exploit
- Triggers an exception to kill itself
  - Do not disturb the LPE !

Plan

01 Finding the map

02 Reaching high seas

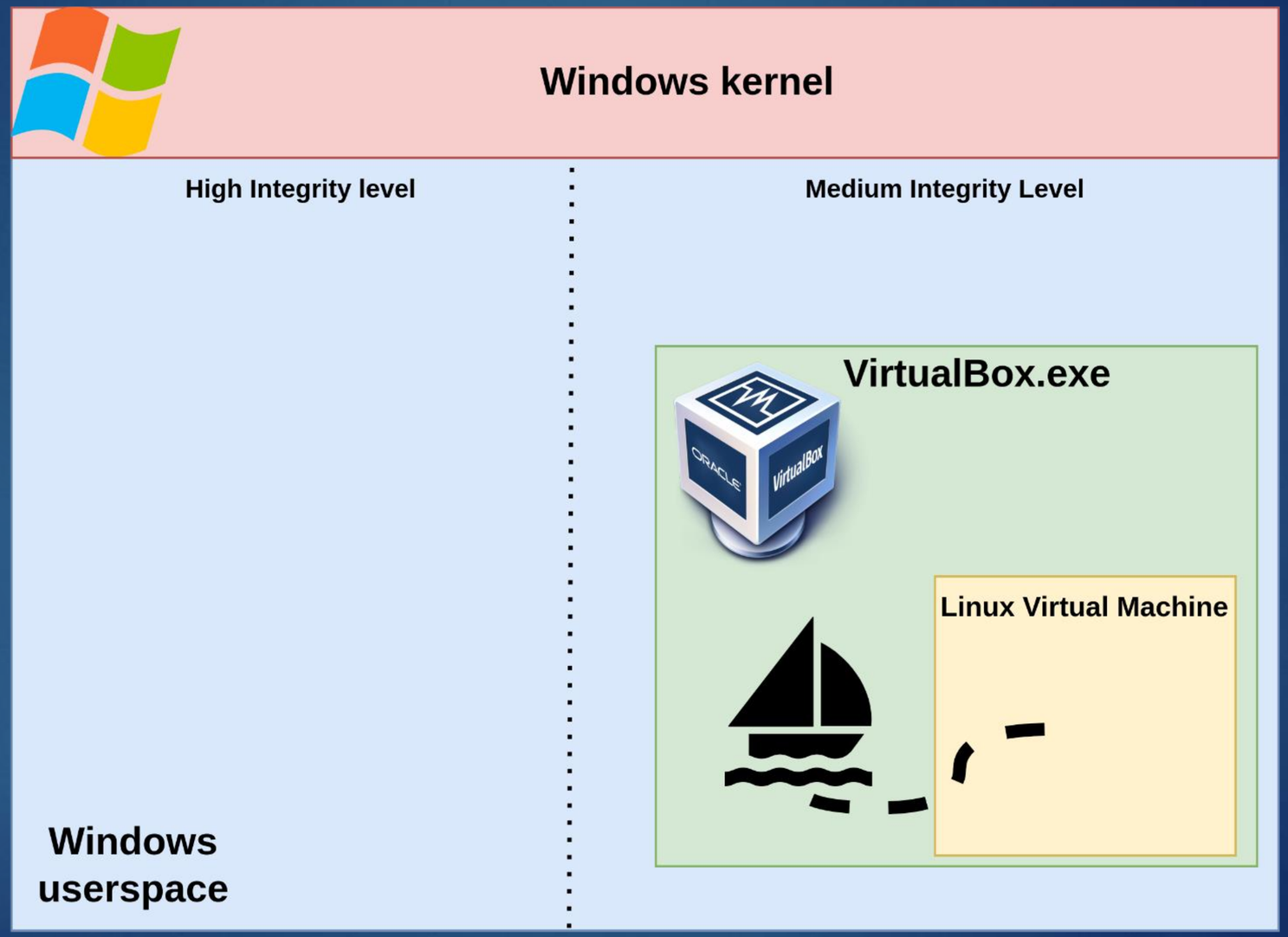
03 Exploring the ocean

04 Hunting for gold

05 Stealing the treasure

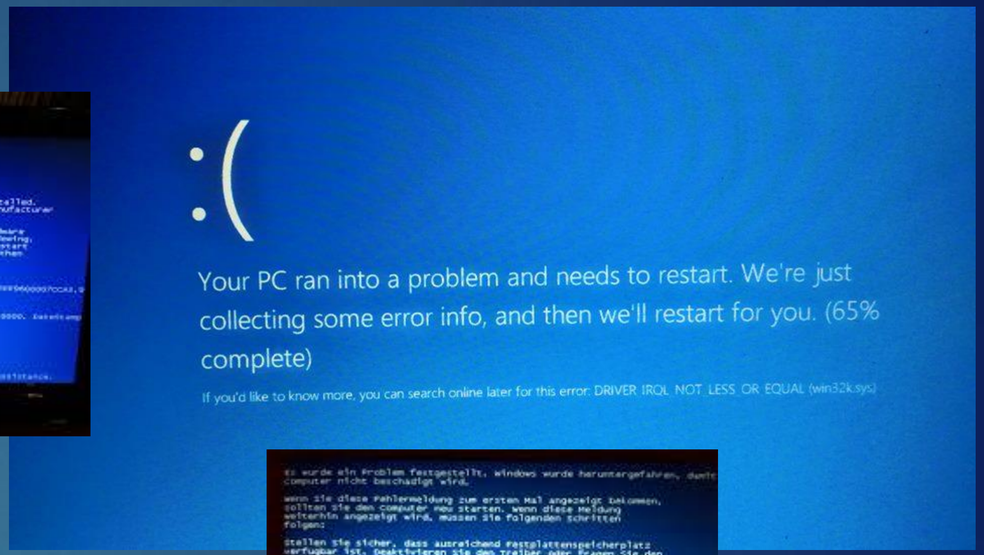
06 Making Port

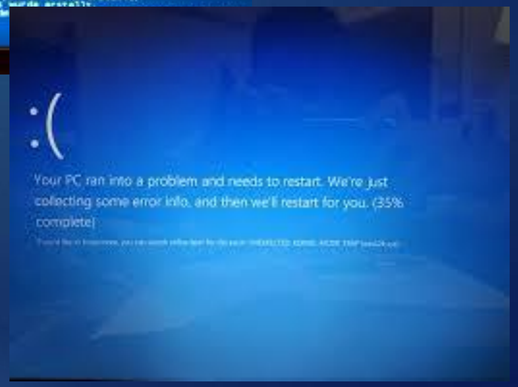
Treasure map



Vulnerability Research

- LPE from medium
  - No need for kASLR bypass



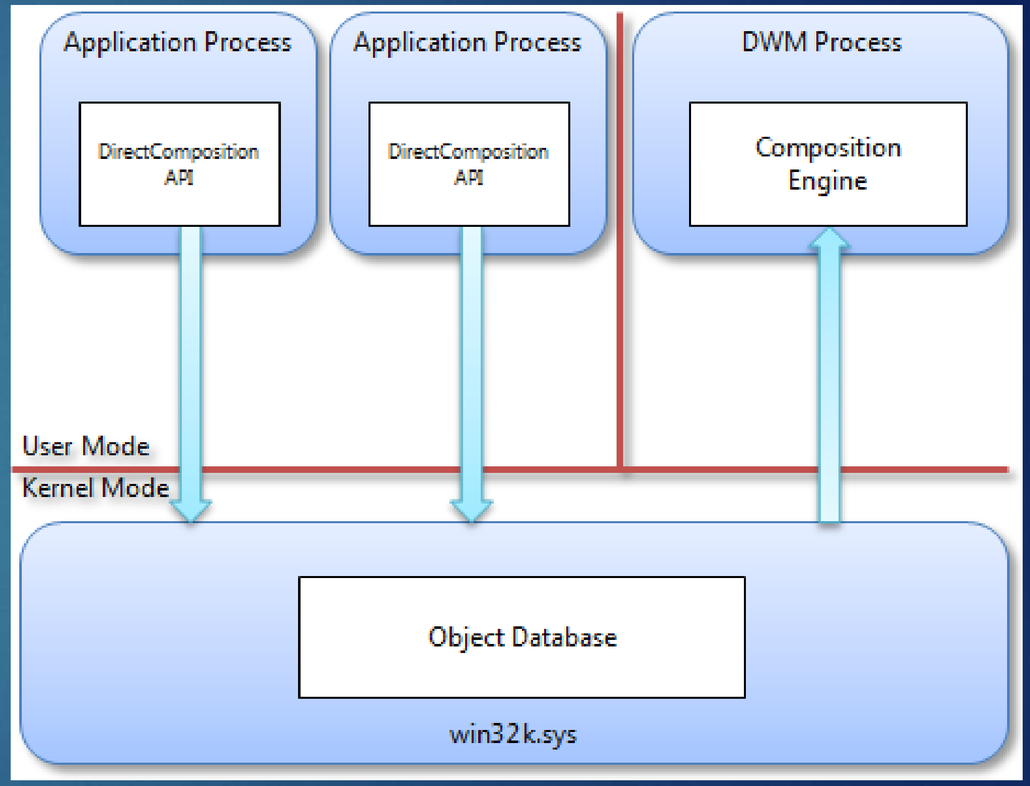


- Target choice:
  - win32k is a huge attack surface
  - Already targeted it at P2O 2022
  - Found a vulnerability in “Direct Composition” (DC)

Why not look at it again **?**

Direct Composition (DC) 101

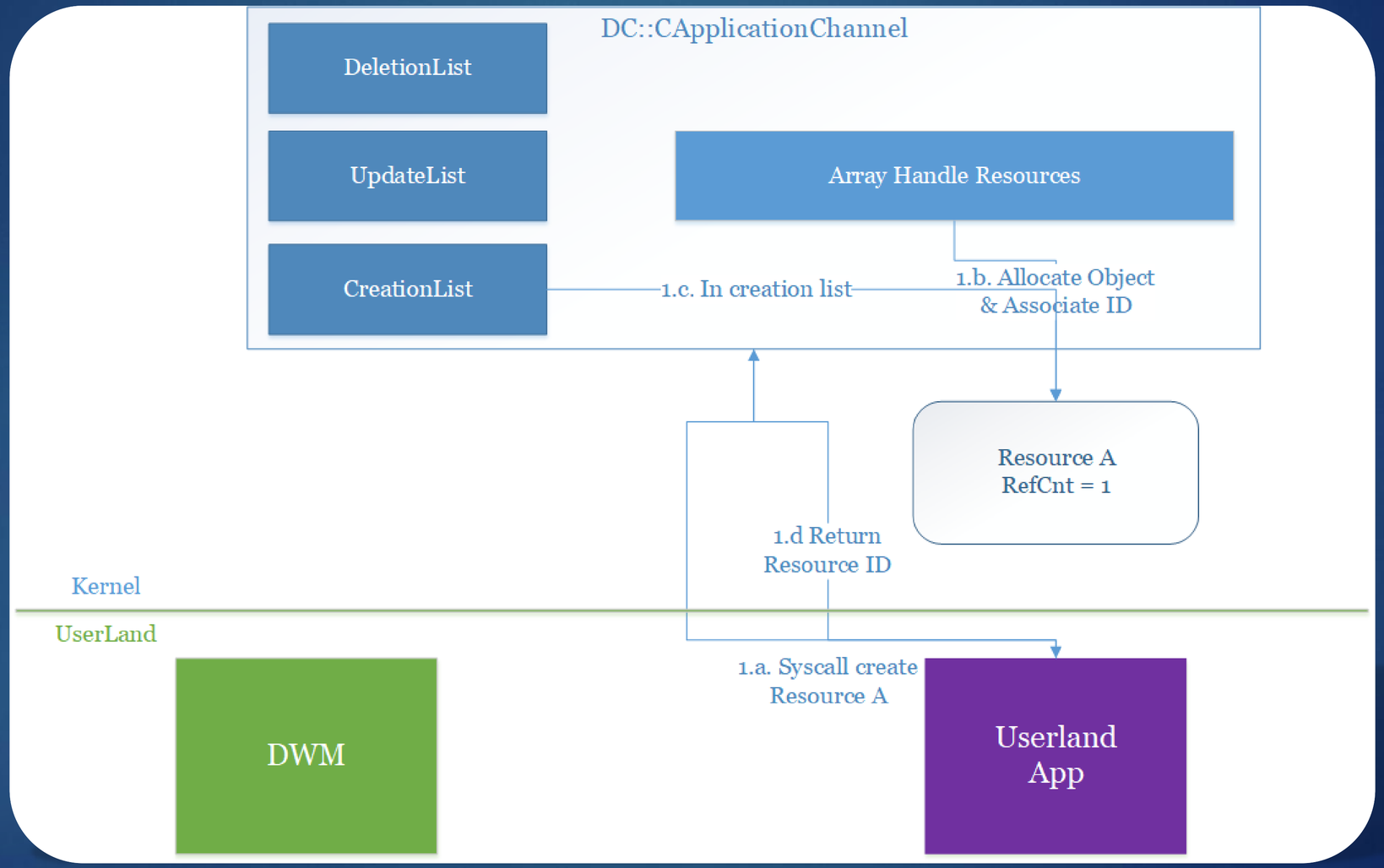
- DC is part of the win32k API
  - Not documented
  - Wrapped by dcomp.dll
- Allows userland applications to perform graphical “compositions”
  - Communicate using the syscalls NtDComposition*
  - In particular allow to send “commands” for manipulating “resources”.
- Data will be stored in the kernel and then transmitted to the privilege process dwm.exe
- Basically a store of object which are used for representing graphical elements which can be updated.



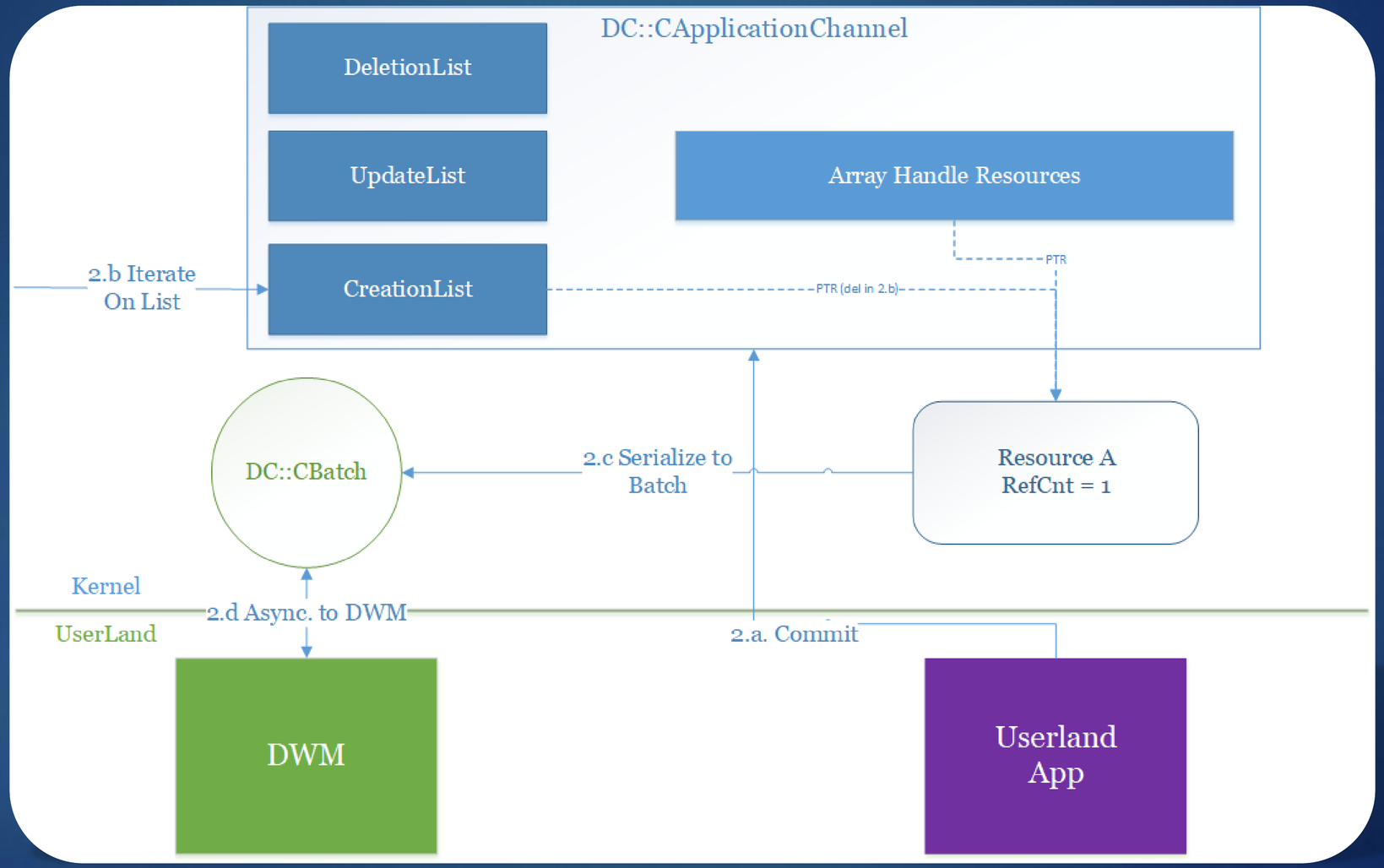
DC Resources

- DC Resources represent graphical objects, actions or interactions.
  - C++ objects inheriting from DirectComposition::CResourceMarshaler
  - Each resources is associated with a “channel” object.
  - Userland can manipulate them through a handle (an index in an array) and the handle of the channel.
  - Resources can be linked together.
- For keeping track of references to a resources, a refcount is implemented:
  - Initially at 1, as long as associated with the channel and usable.
  - Incremented by 1 if another resource/object is linked to it.
  - When reaching 0, it will be marked as being “to delete” but not actually freed yet.
- Change in resources need to be “emitted”/”committed”
  - Will allow to create a “batch” of serialized data representing the actions
  - That batch of data will be fetch by dwm at some point

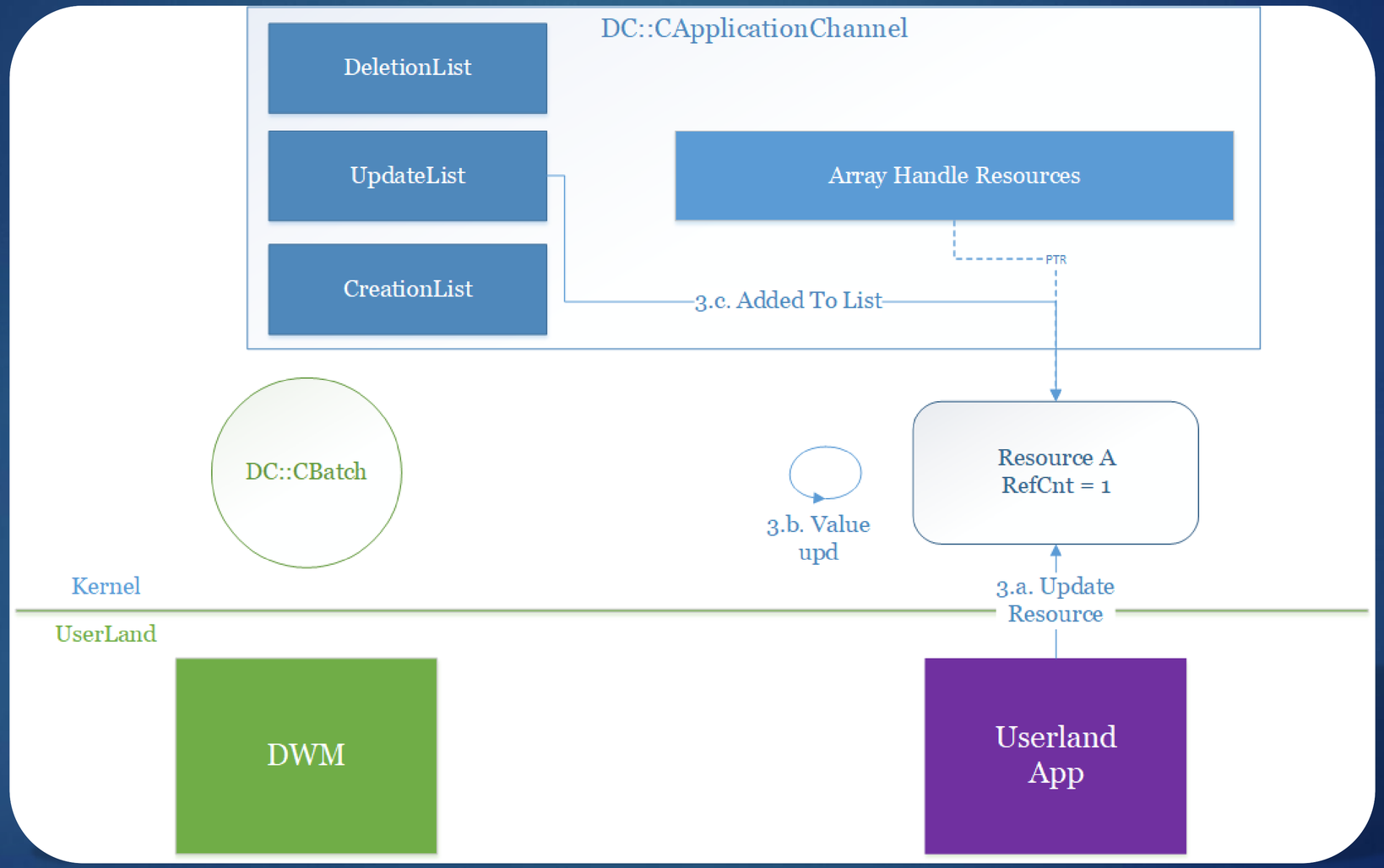
Creation of a DC Resource



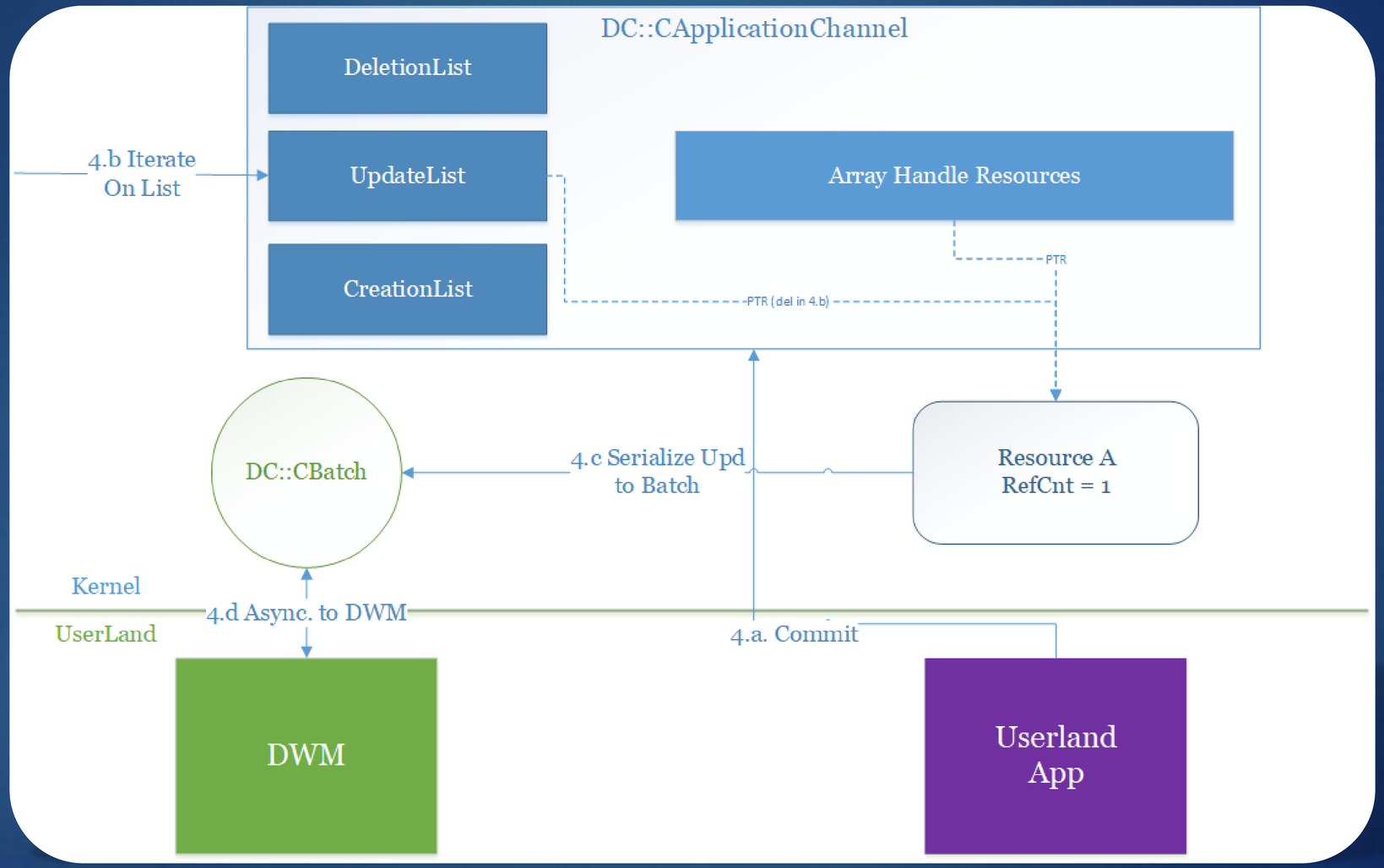
Creation of a DC Resource



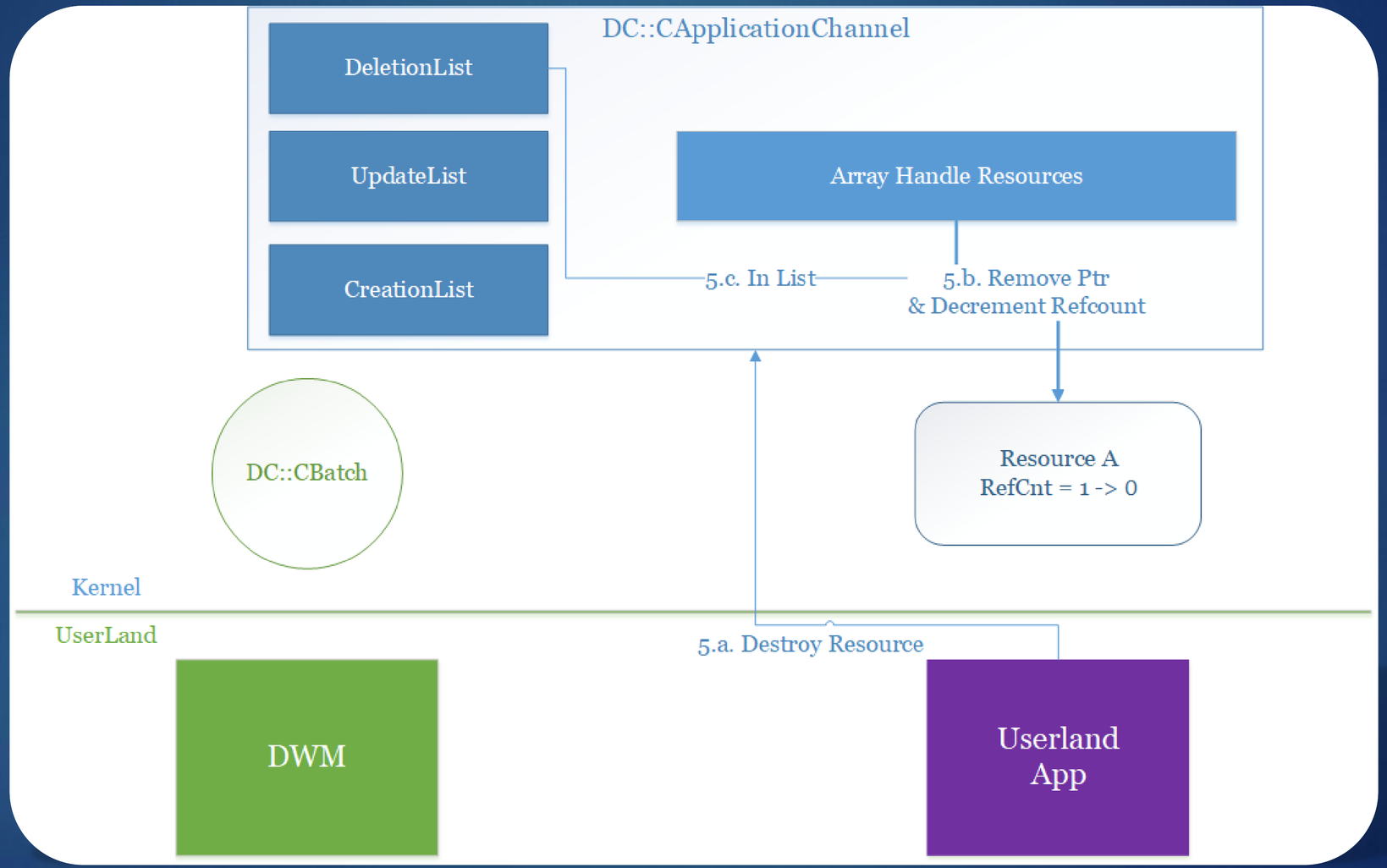
Update of a DC Resource



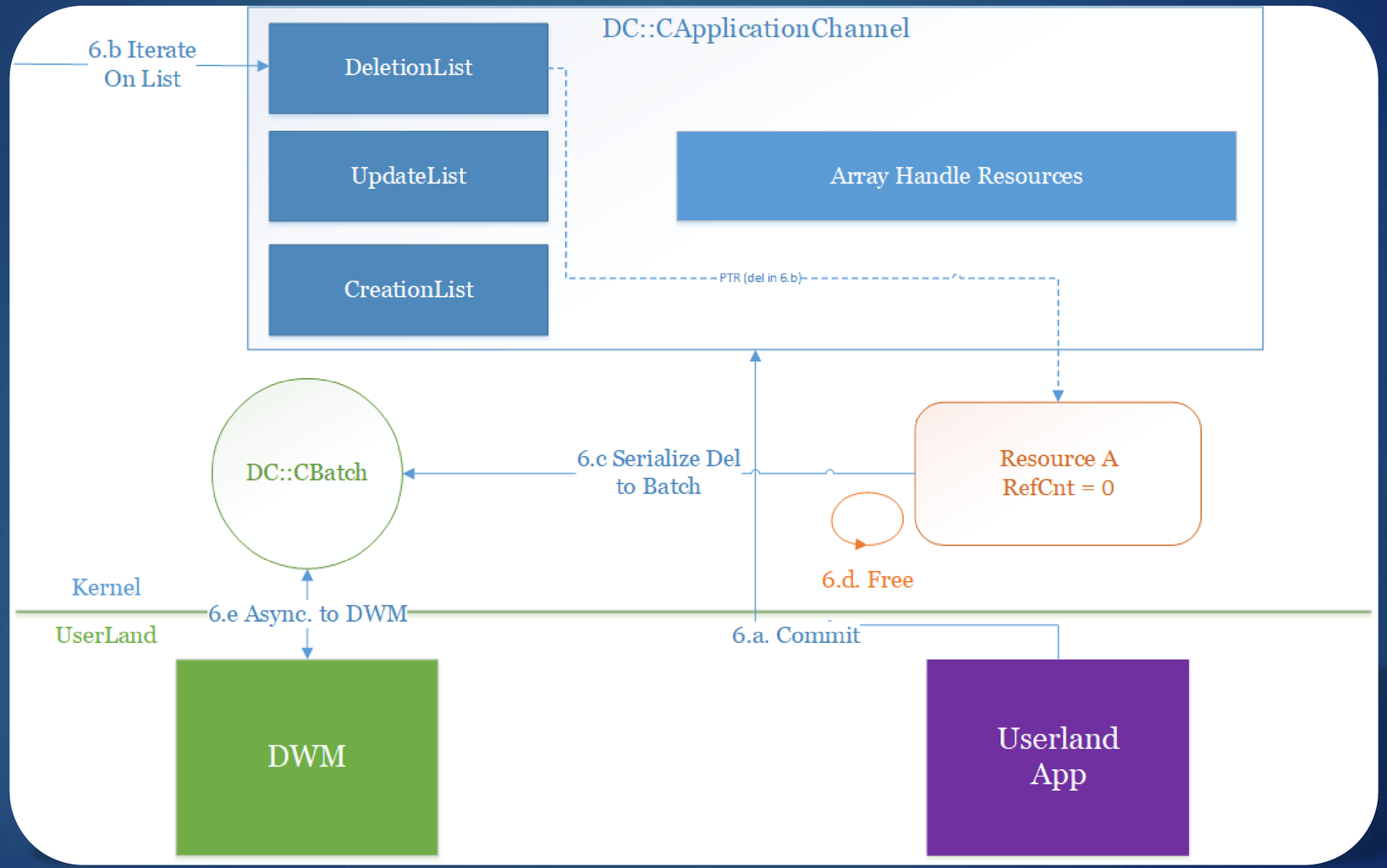
Update of a DC Resource



Deletion of a DC Resource

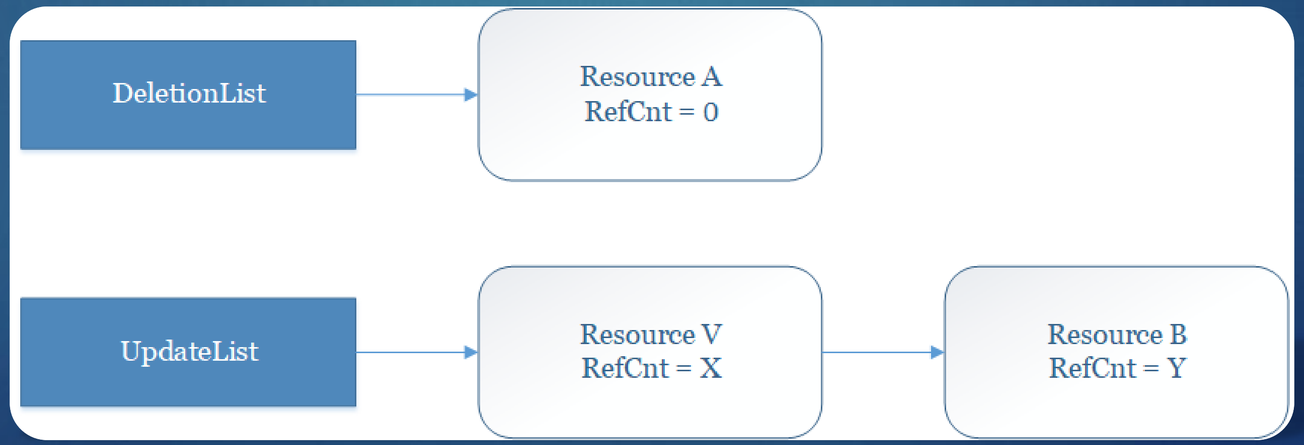


Deletion of a DC Resource



DC Update & Deletion

- The DeletionList & UpdateList are part of the CApplicationChannel
- Linked list using the SAME pointer in the resources.
  - Should never be present in both.
  - If deleted while updated: removed from update and added to deletion.
- RefCnt is at 0 for all resources in the deletion list. Will be free after commit.
- RefCnt is > 0 for all resources in the update list.



The (potential) bug

```
currResource = CApp->UpdateList.head; // [1]
if ( currResource )
{
    while ( 1 )
    {
        this->UpdateList.head = currResource->nextInList; // [2]
        vftable = currResource->vftable;
        currResource->nextInList = 0i64; // [3]
        if ( !vftable->EmitUpdateCommands(currResource, &pCBatch) ) { // [4]
            // Failure of EmitUpdateCommands
            currResource->nextInList = CApp->UpdateList.head; // [5]
            this->UpdateList.head = currResource;
            goto lbl_retFalse2; // return False from BuildBatch
        }
        // Success of EmitUpdateCommands
        currResource->flags_toEmit &= ~2u;
        currResource = CApp->UpdateList.head;
        if ( !currResource ) // nothing in update: go out
            break; // continue next part
    }
}
```

A real bug ?

- At this point we have no idea if this is a real vulnerability…
- Problematics linked to *EmitUpdateCommands* virtual function:
  - Need to release the last reference to the current resource.
  - AND to make the function fail after.
- And of course we would need to exploit after this.
- Good news: there is a LOT of different resources (more than 150)
  - Means lot of different implementation of *EmitUpdateCommands*

Plan

01 Finding the map

02 Reaching high seas

03 Exploring the ocean

04 Hunting for gold

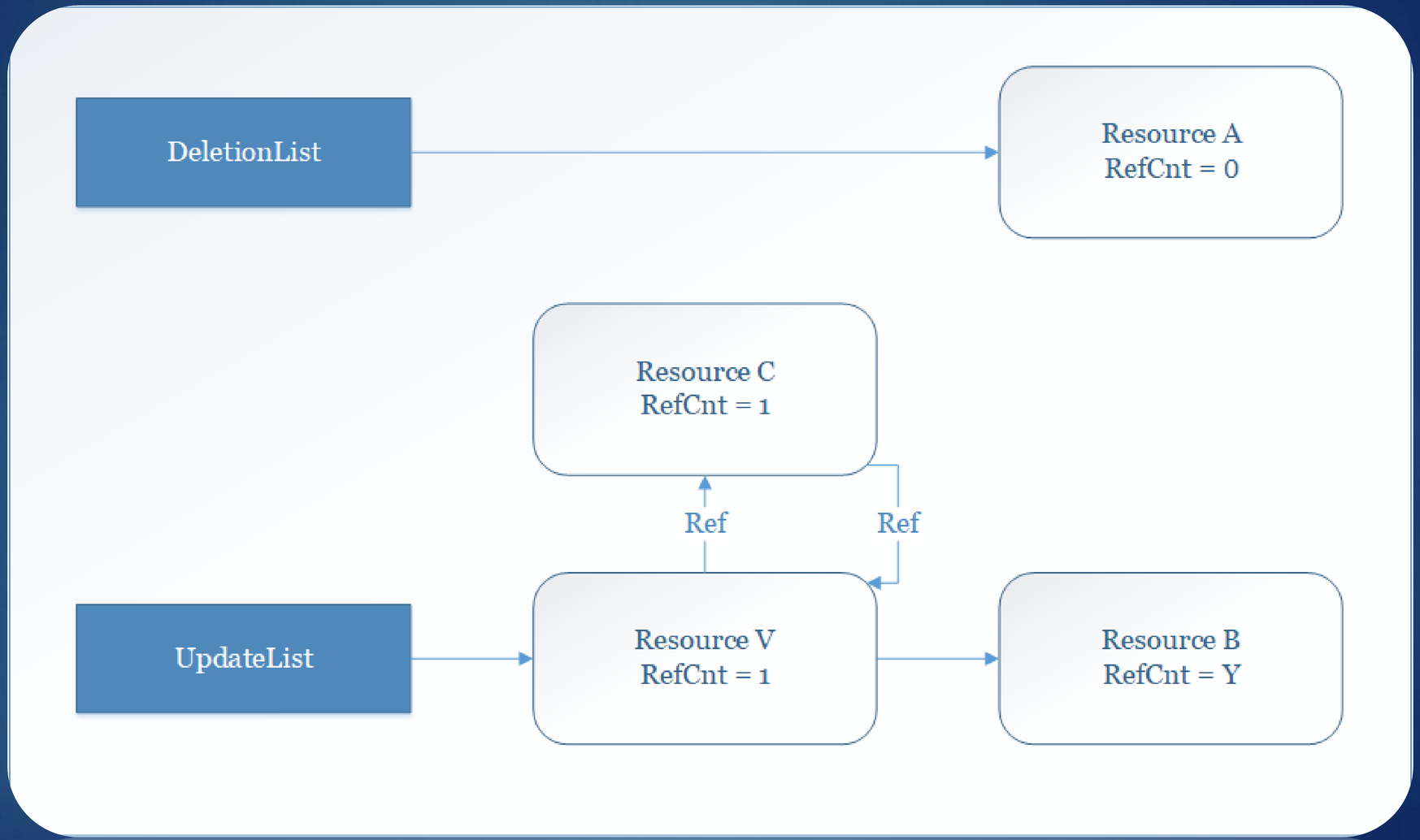
05 Stealing the treasure

06 Making Port

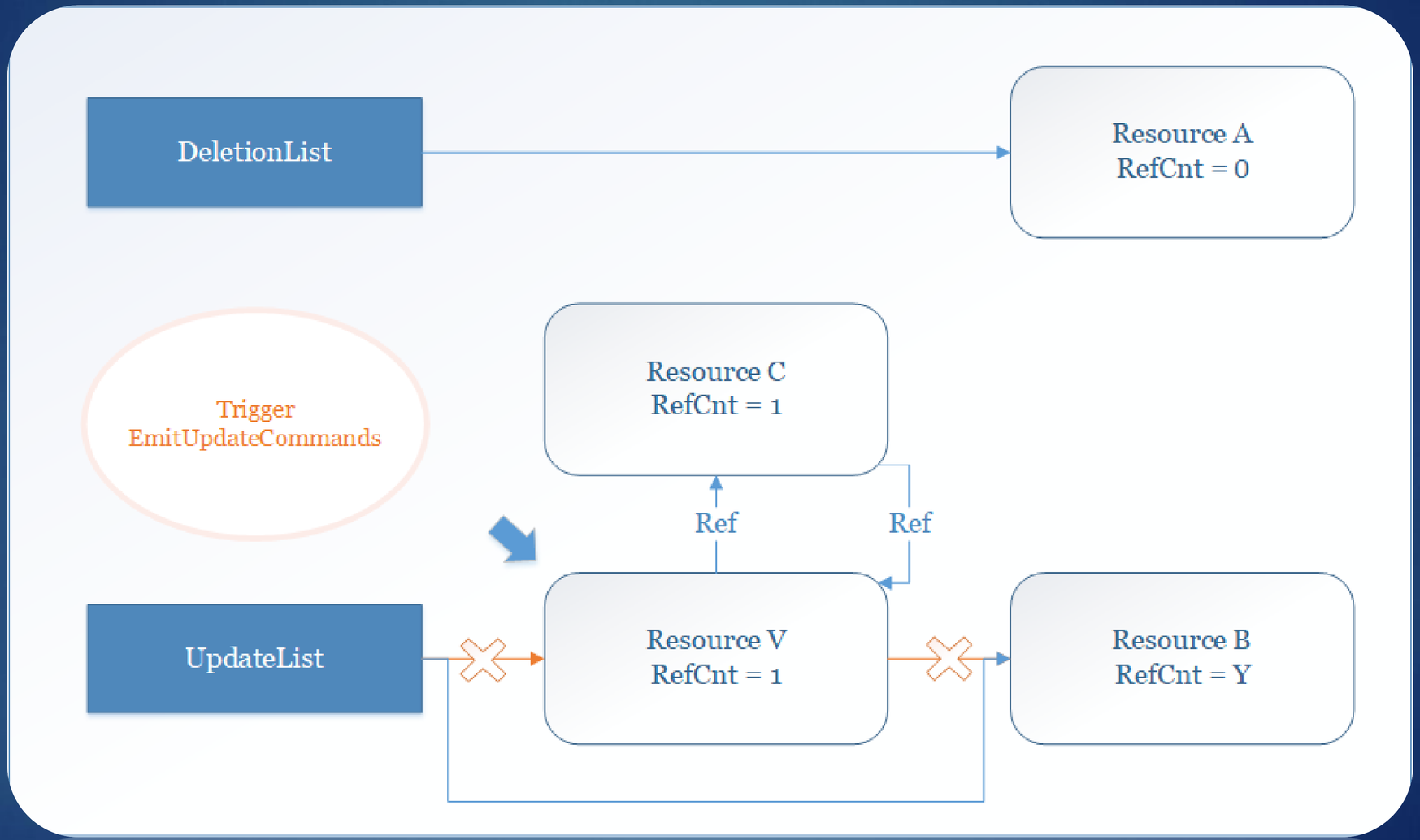
Releasing the resource

- No resource will delete the reference on themselves during the *EmitUpdateCommands*
- Some resources will release another resource.
  - In particular *visual* resources can have “children”
  - If the link between a child and its parent visual is removed, the reference to the child will be removed during the *EmitUpdateCommands*.
- Idea: use circular reference
  1. Create a visual with a child.
  2. Make the child have a reference on its parent.
  3. Remove the link between child & parent. And any reference on both child and parent.
  4. Commit: child get its refcnt at 0 which trigger the refcnt at 0.

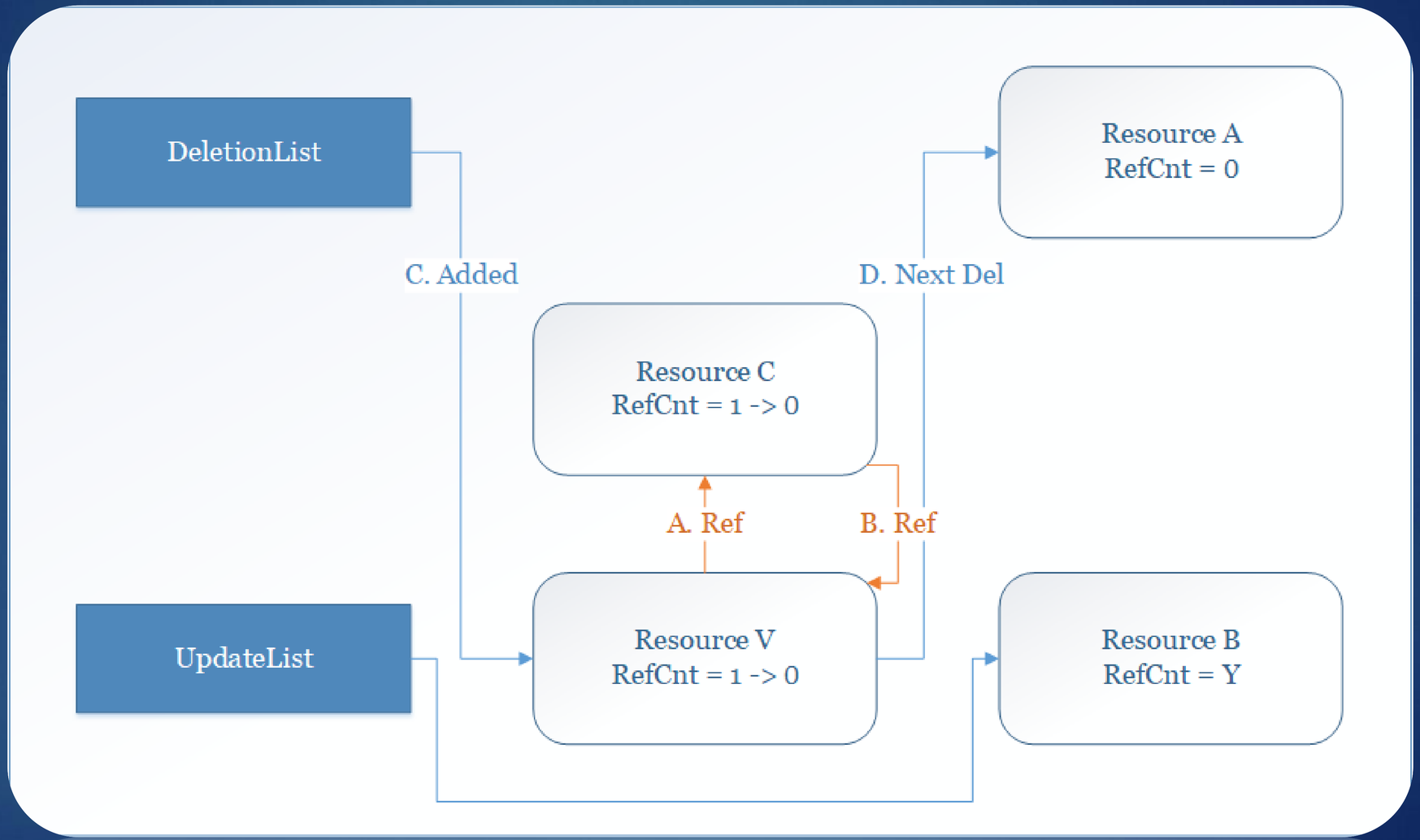
Circular References



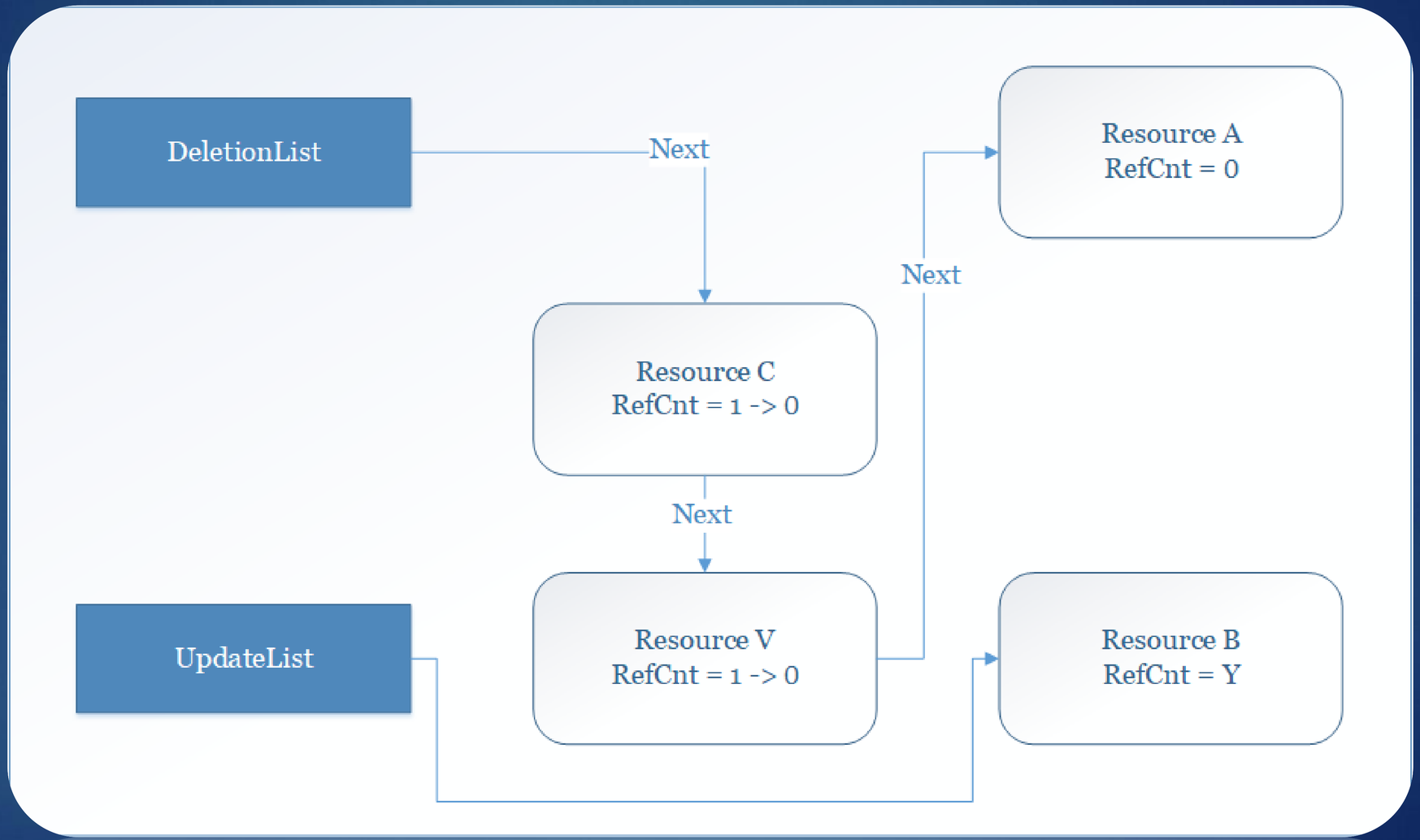
Circular References



Circular References



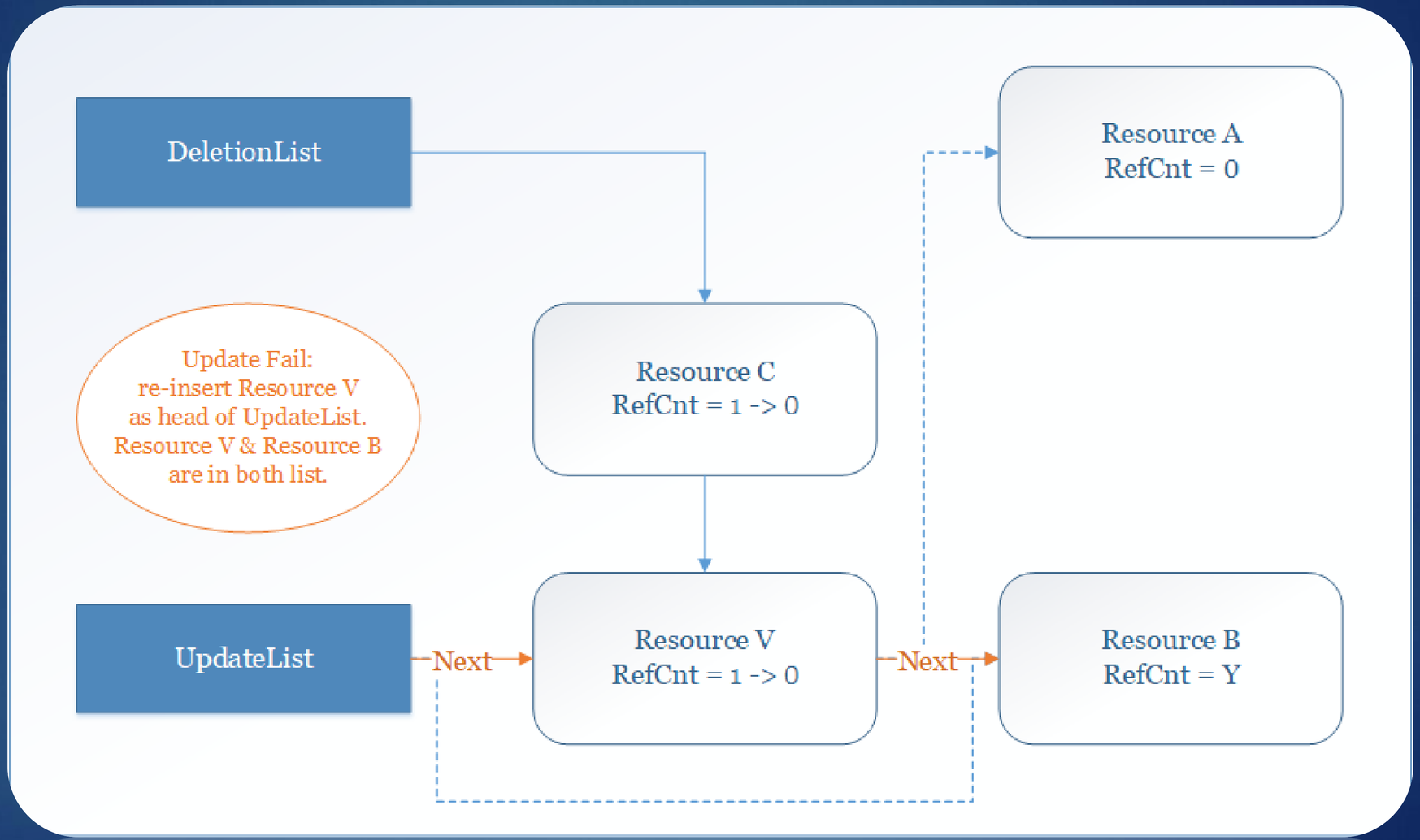
Circular References



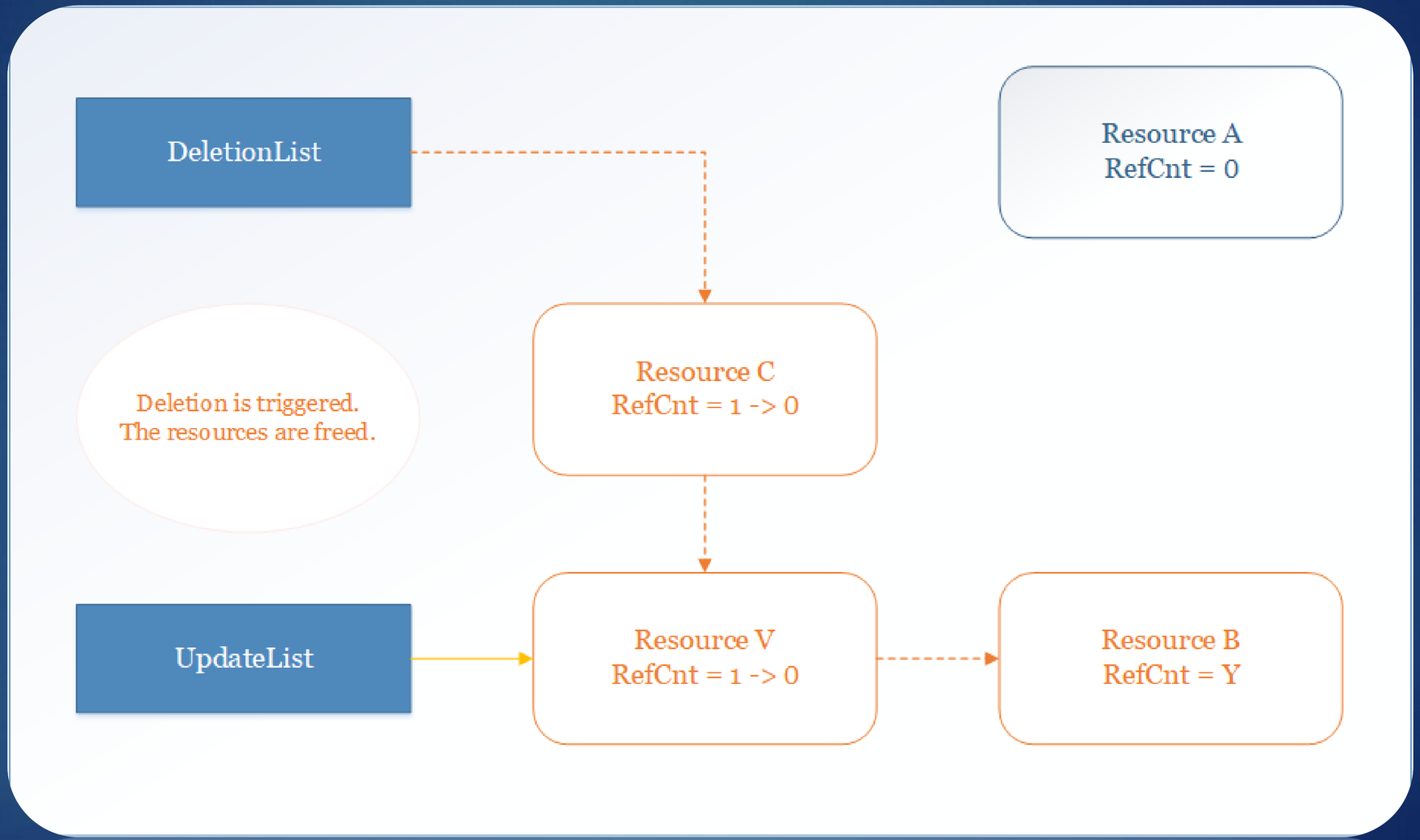
The (potential) bug

```
currResource = CApp->UpdateList.head; // [1]
if ( currResource )
{
    while ( 1 )
    {
        this->UpdateList.head = currResource->nextInList; // [2]
        vftable = currResource->vftable;
        currResource->nextInList = 0i64; // [3]
        if ( !vftable->EmitUpdateCommands(currResource, &pCBatch) ) { // [4]
            // Failure of EmitUpdateCommands
            currResource->nextInList = CApp->UpdateList.head; // [5]
            this->UpdateList.head = currResource;
            goto lbl_retFalse2; // return False from BuildBatch
        }
        // Success of EmitUpdateCommands
        currResource->flags_toEmit &= ~2u;
        currResource = CApp->UpdateList.head;
        if ( !currResource ) // nothing in update: go out
            break; // continue next part
    }
}
```

Circular References



Circular References



EmitUpdateCommands fail

Engineering a failure

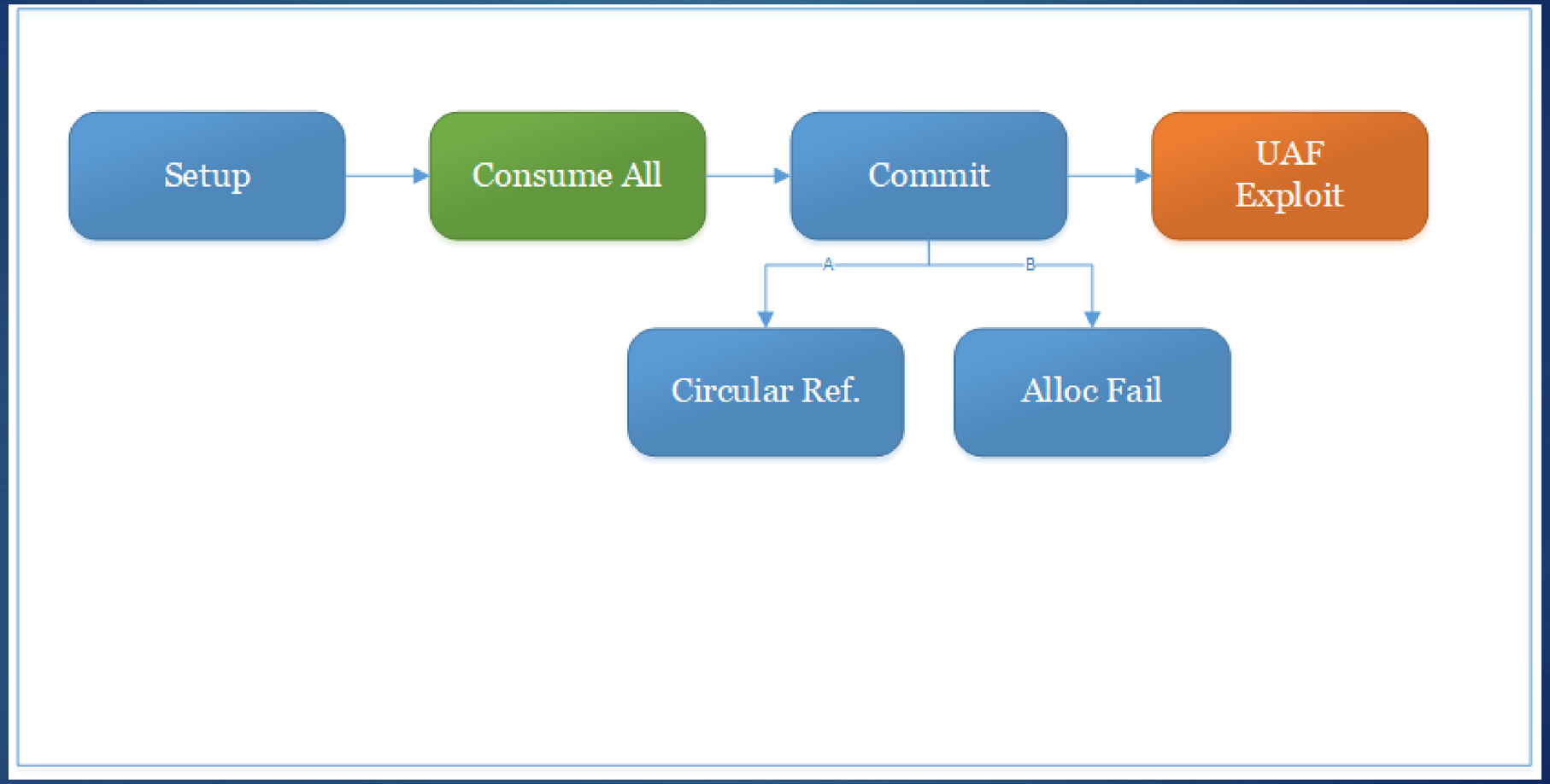
- We still need *EmitUpdateCommands* to fail.
  - But there is no easy way to do that with any resources which allow the removal of the refcount…
- One case is possible:
  - During the emit, more memory might need to be allocated for the CBatch to receive the serialized data.
  - If the allocation fails, *EmitUpdateCommands* fails

Is this even possible **?**

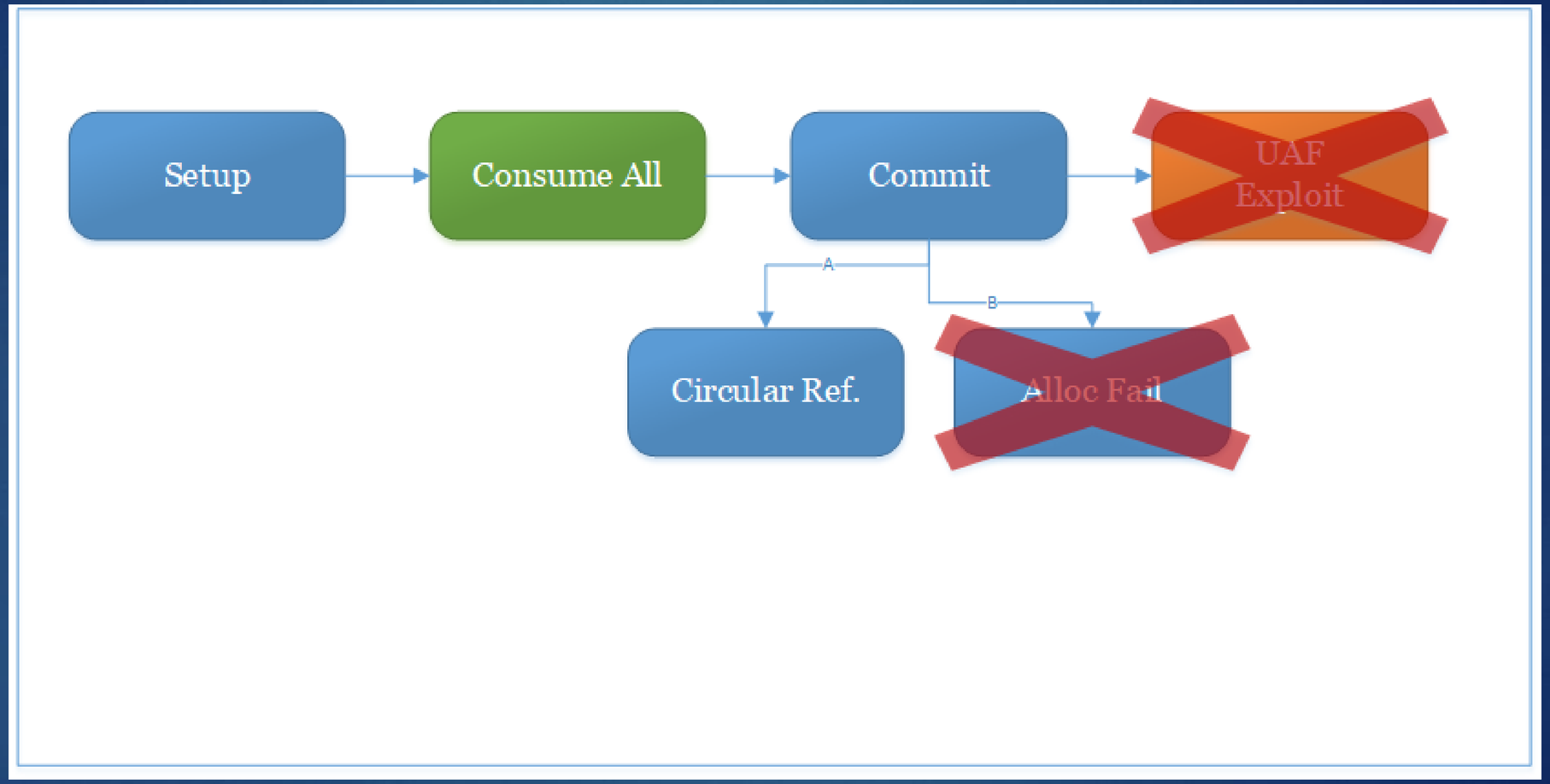
Engineering a failure

- Allocation use “section” (*MmCreateSection*)
- Section can be allocated using another syscall (*NtCreateSection*)
- In theory we can allocate all memory from the computer and make the creation fails.
- Need to calculate the good size for triggering the batch memory request at the good moment but that is easily doable.
- Can we actually exhaust the memory ? Let’s try!

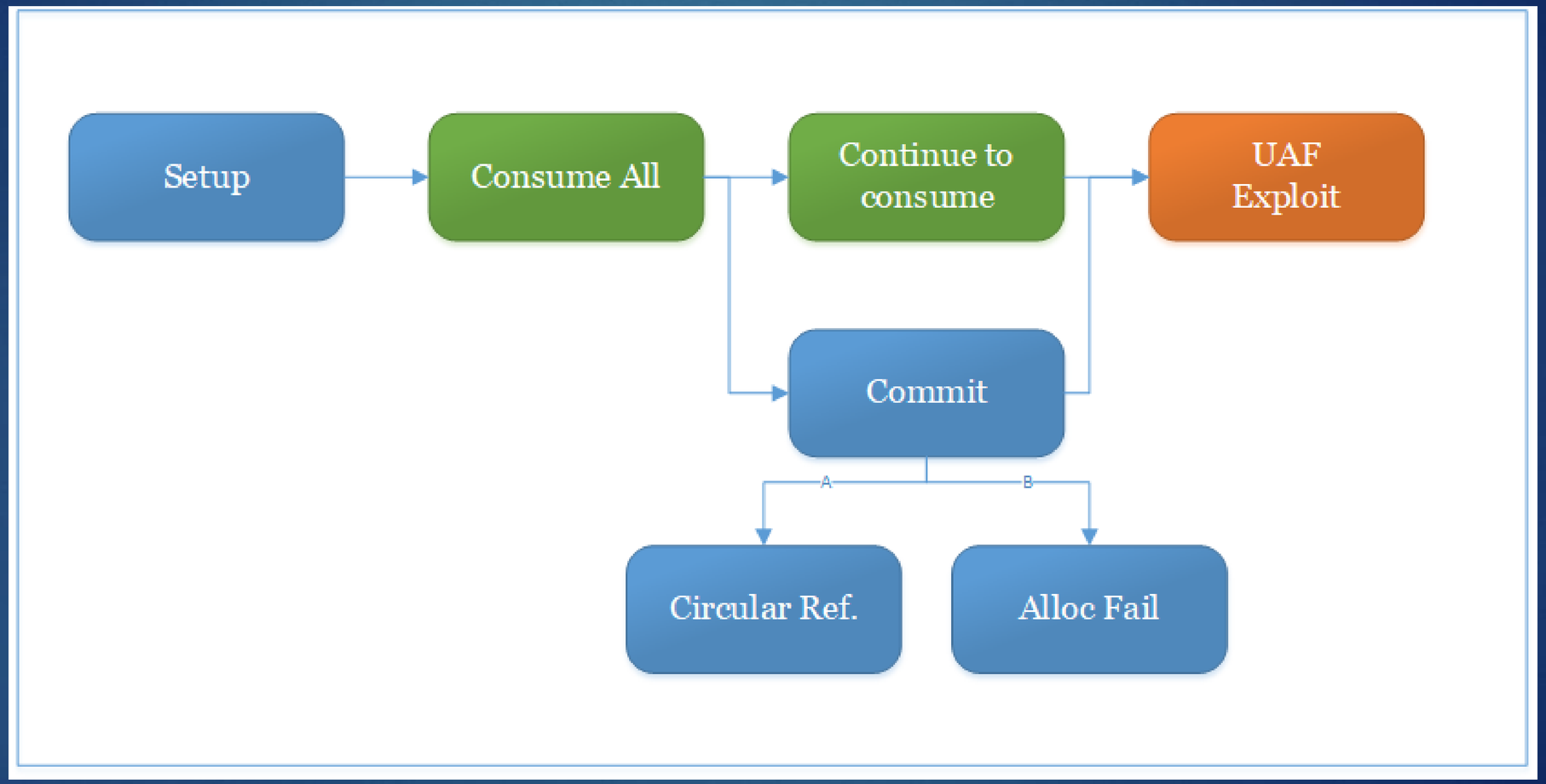
Consuming the world



Consuming the world



Consuming the world



It’s a real bug !

- Successfully trigger the UAF!
- But we still need to get away with the gold!
- Let’s see how we can exploit our UAF.

Plan

01 Finding the map

02 Reaching high seas

03 Exploring the ocean

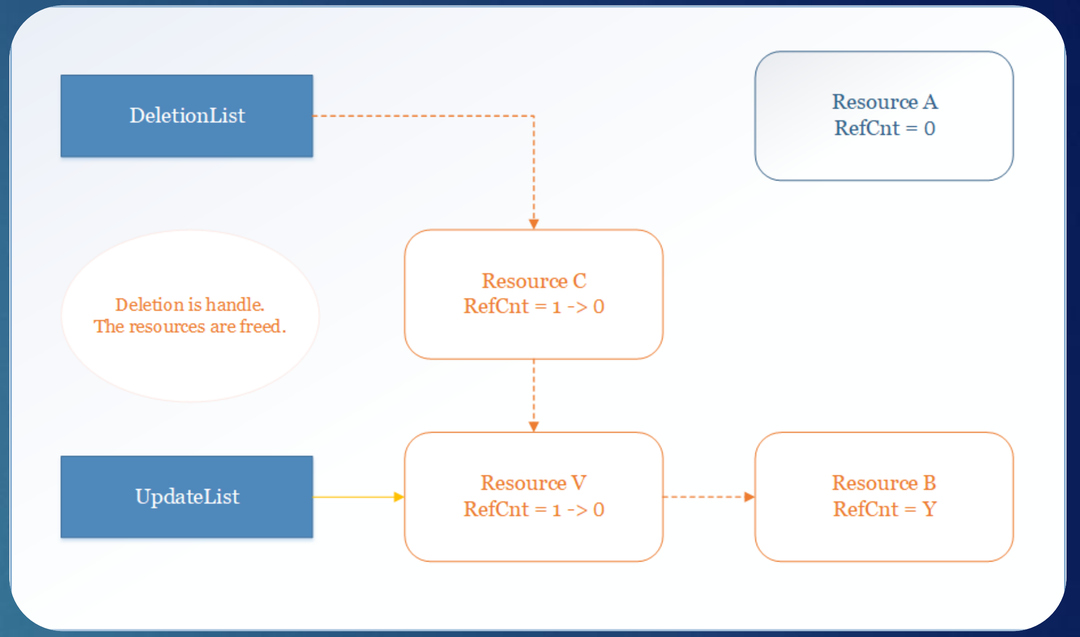
04 Hunting for gold

05 Stealing the treasure

06 Making Port

Basic UAF consideration

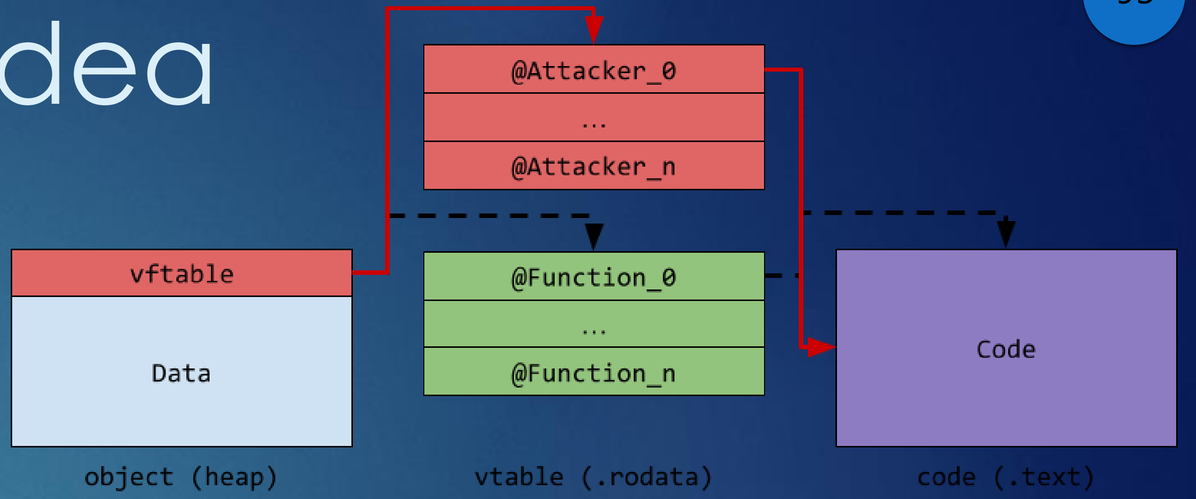
- We can put ANY resource as a UAF
  - Lot's of different object choices!
  - Lot's of different size!
- In win32k known exploitation techniques with palette objects:
  - Allocated through a syscall
  - Arbitrary size!
  - Content controlled!
- Let's go!



Interesting object

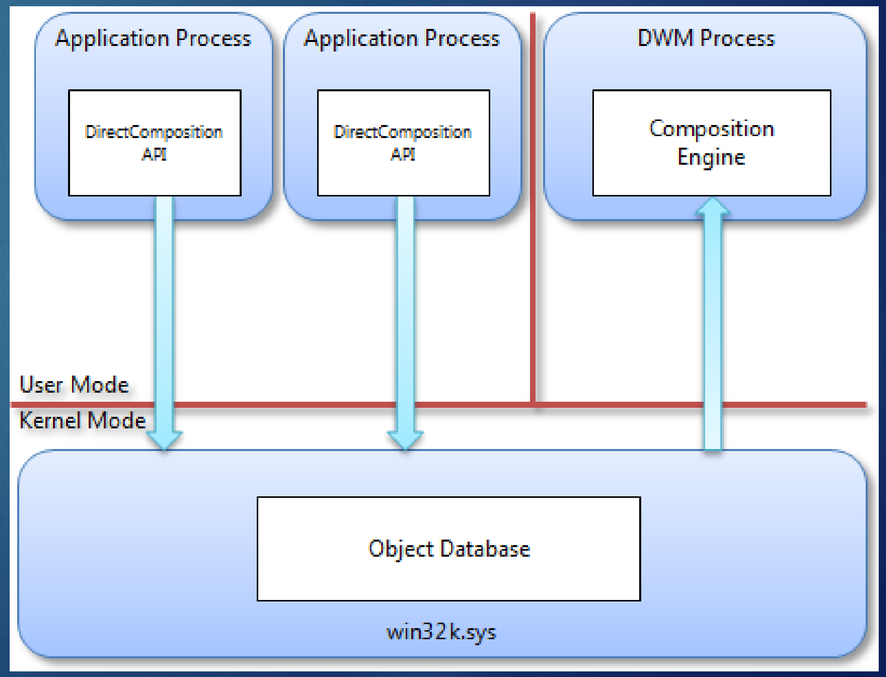
Must be of the same size.
Must allow to do something with the UAF.
Must be easy to allocate numerous time.
If possible, persistent in memory.

Initial idea

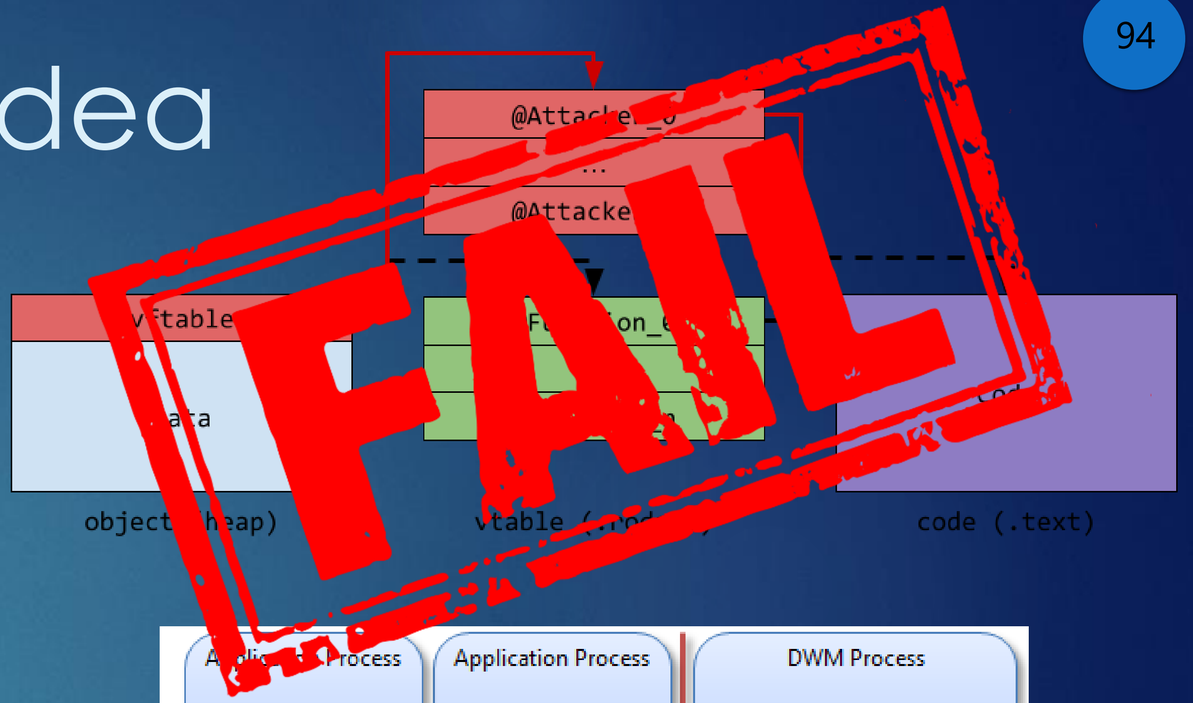


- Resources are C++ objects with vftables and Windows has no SMAP!

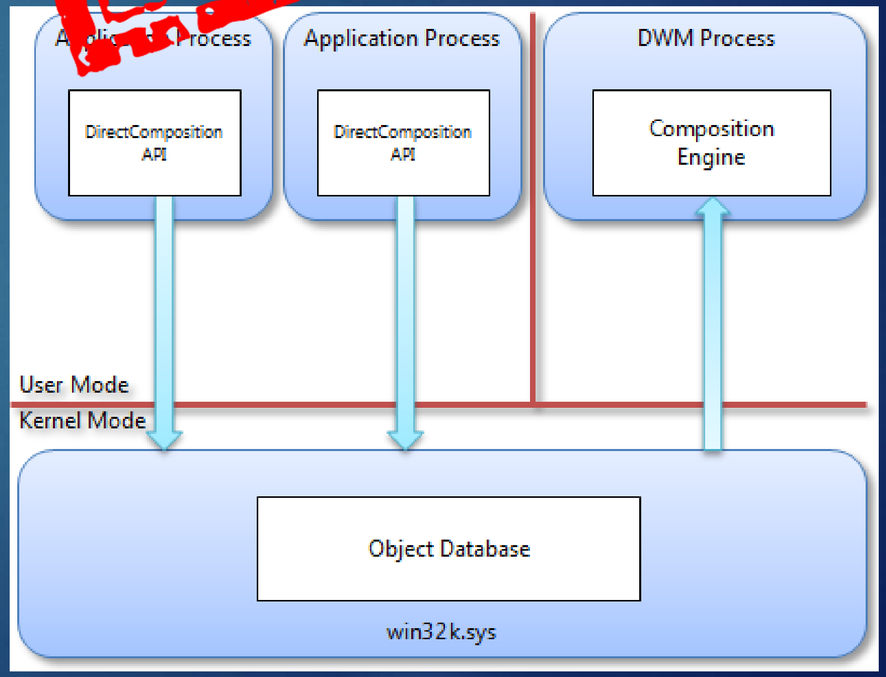
- Simple (original) idea:
  - Set our vftable in userland.
  - Arbitrary code execution each time it uses a function!



Initial idea



- Resources are C++ objects with vftables and Windows has no SMAP!
- Simple (original) idea:
  - Set our vftable in userland.
  - Arbitrary code execution each time it uses a function!
- But the object is also accessed by DWM.exe.
  - Which will trigger a crash because not the same userland memory :(



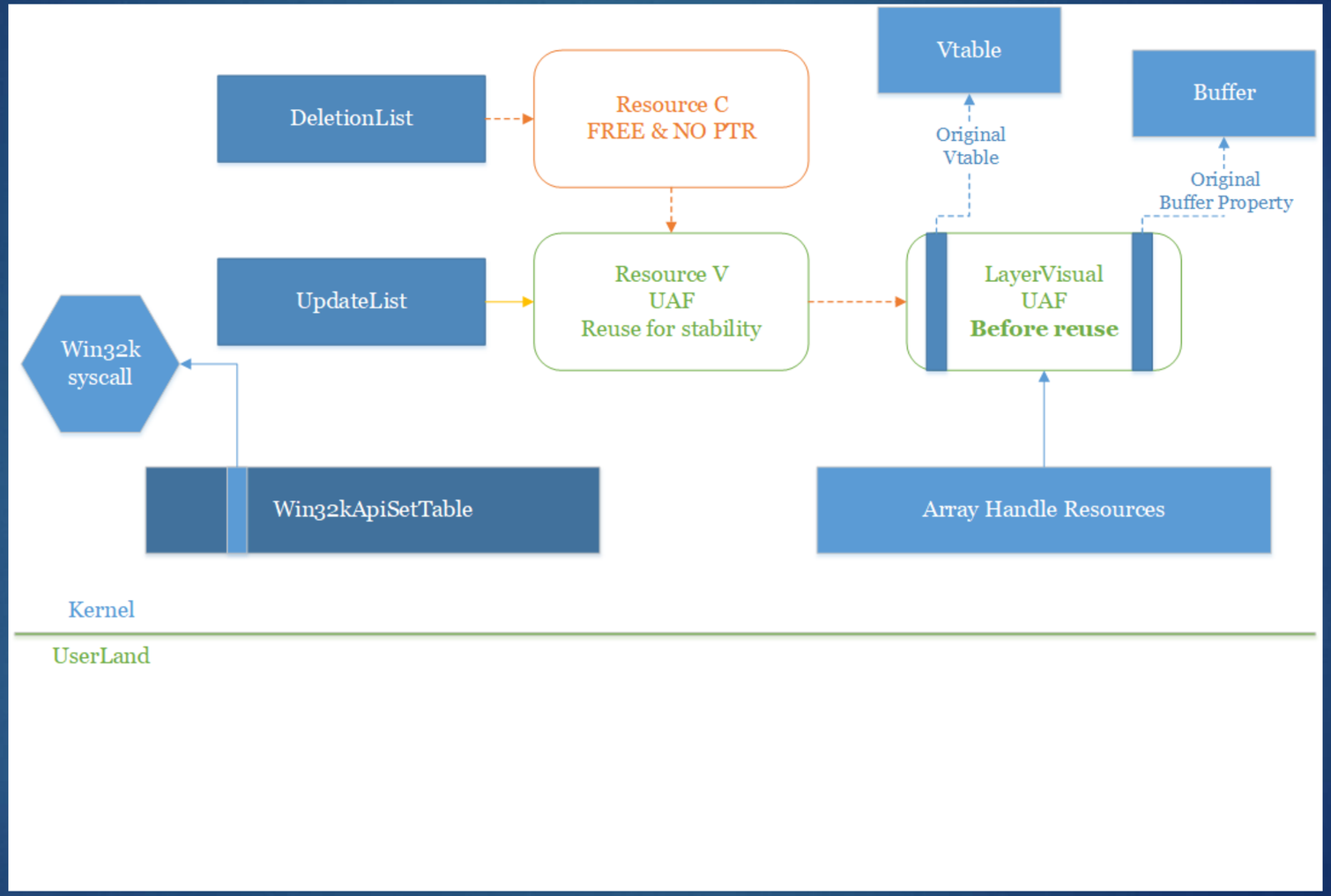
Better idea: buffer property

- Lot's of resource have a "buffer property"
  - A pointer on an allocated buffer stored in our object.
  - Possible to use from the userland for setting data.

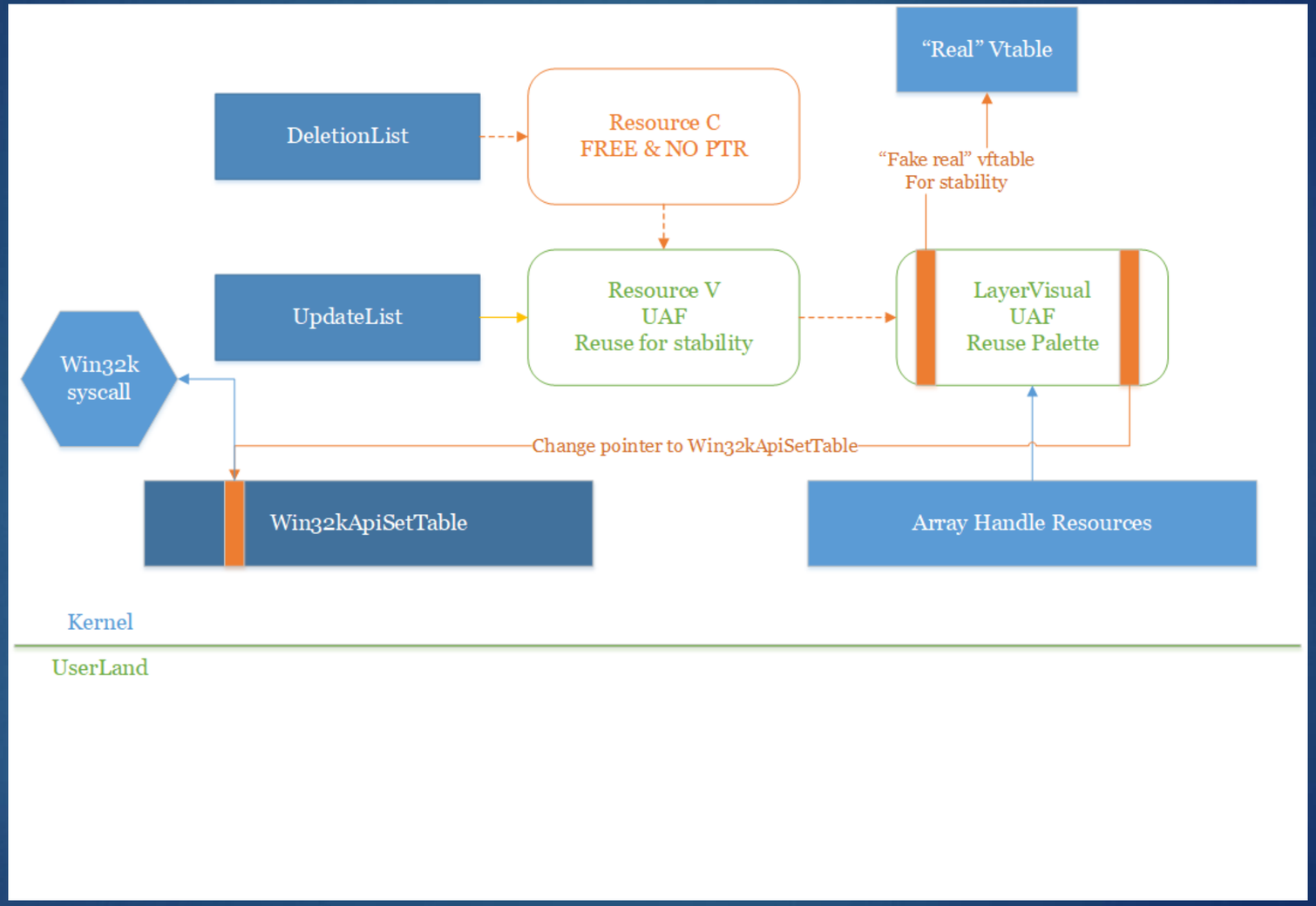
- The *LayerVisual* is a visual object with:
  - A buffer property
  - A size of 0x190

- Idea:
  1. Put the *LayerVisual* in UAF
  2. Rewrite the "buffer property" pointer with a pointer of our choice using the palette.
  3. Use the pointer (Set the content of the *LayerVisual* "buffer property"): Arbitrary write!

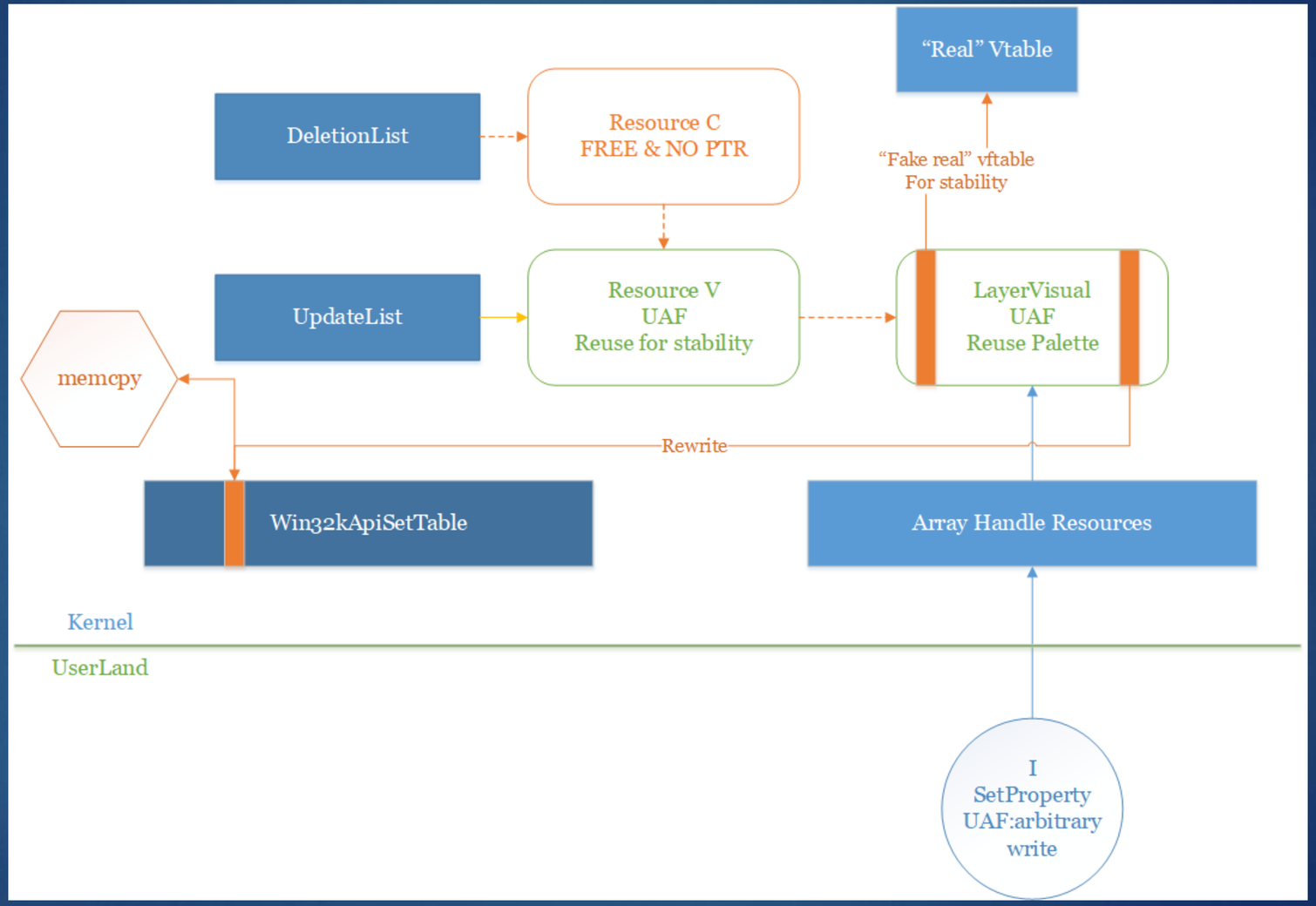
Final Exploit Overview



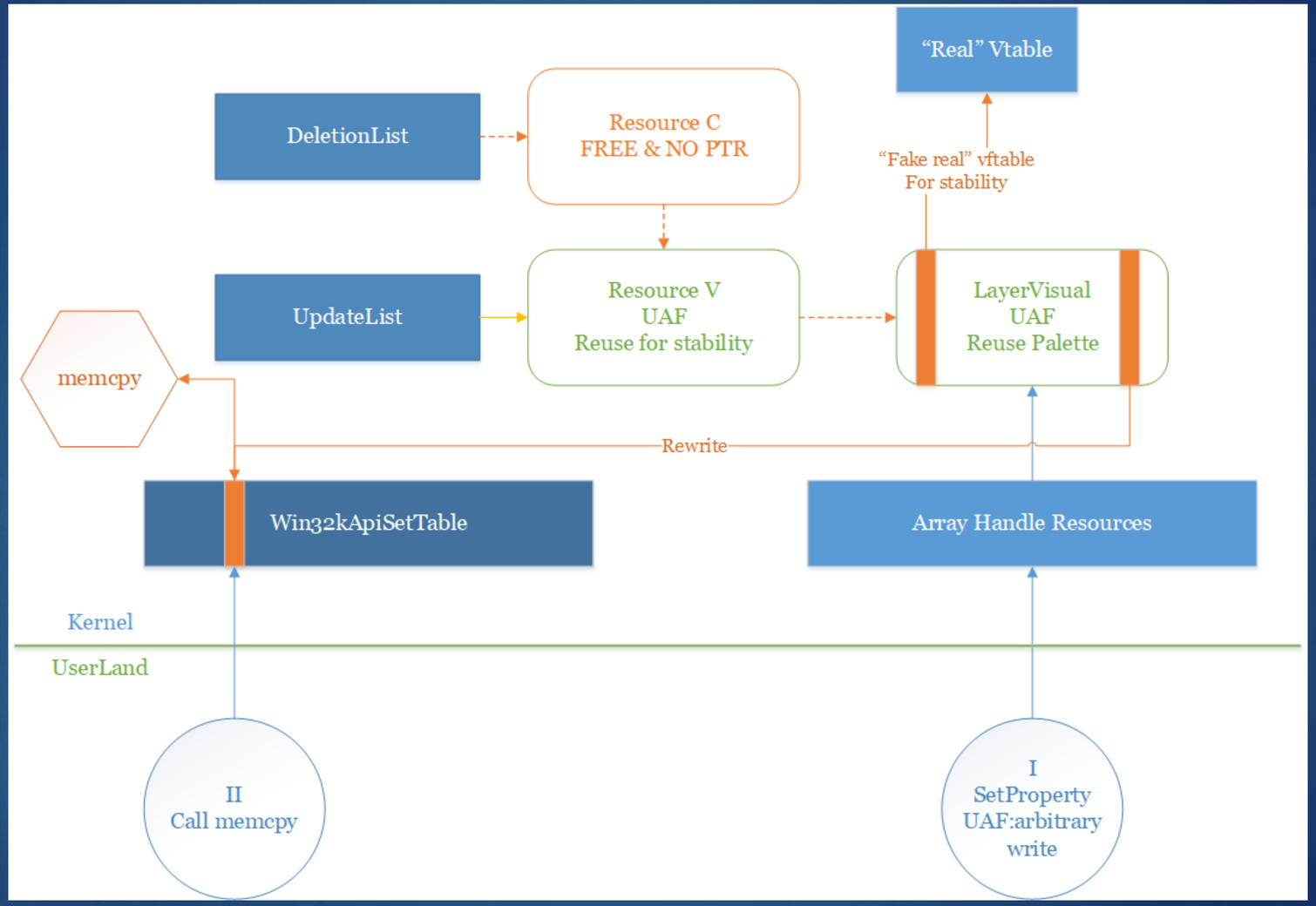
Final Exploit Overview



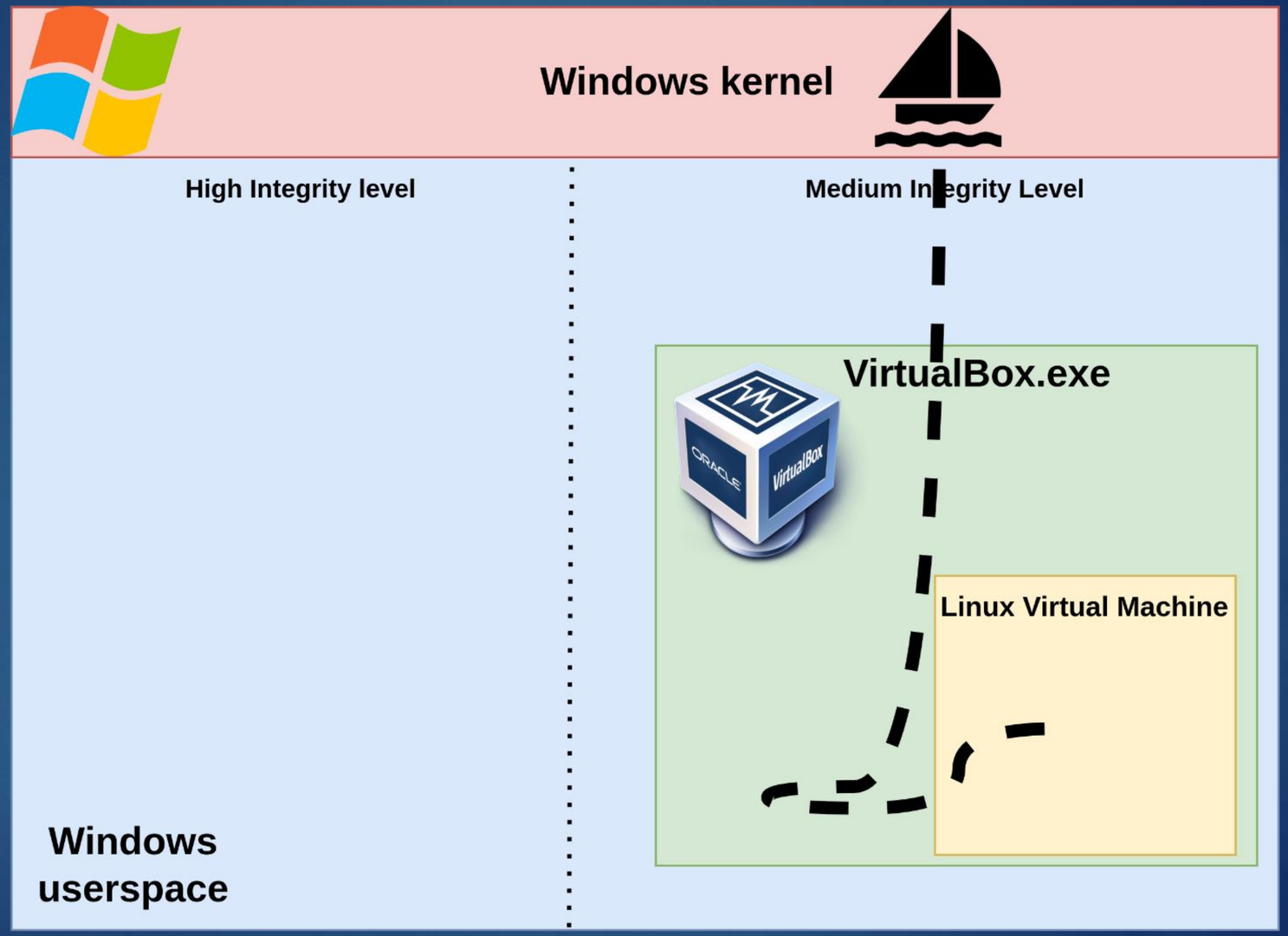
Final Exploit Overview



Final Exploit Overview



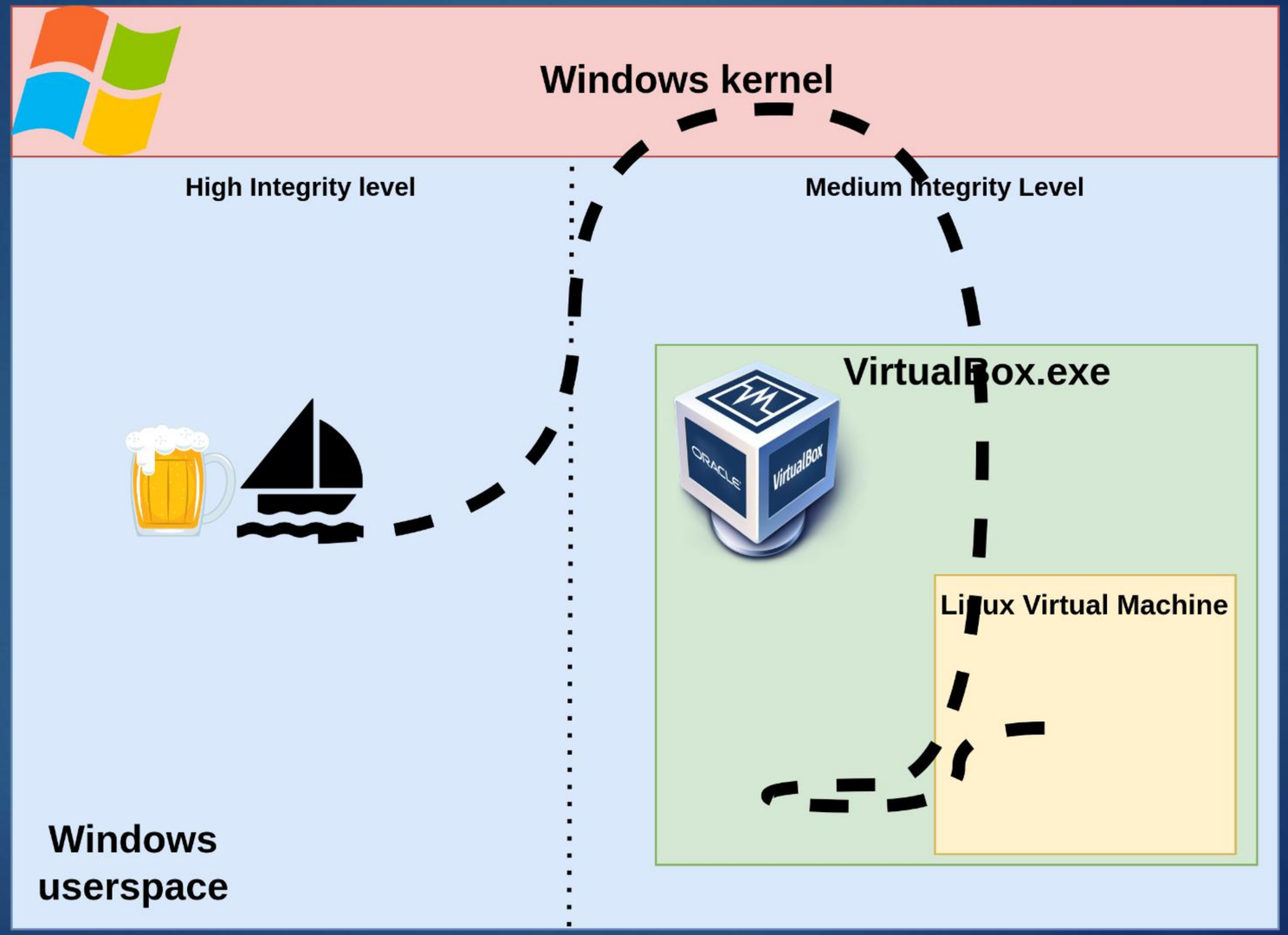
Treasure map



Post exploit

- Arbitrary read/write in kernel
- Make some cleanup for avoiding to crash because of our list.
- Then we could get code execution…
  - … or simply rewrite the token of our process to be admin.
  - Steal the token of the initial process!
- We are NT Authority / system !

Treasure map



Plan

01 Finding the map

02 Reaching high seas

03 Exploring the ocean

04 Hunting for gold

05 Stealing the treasure

06 Making Port

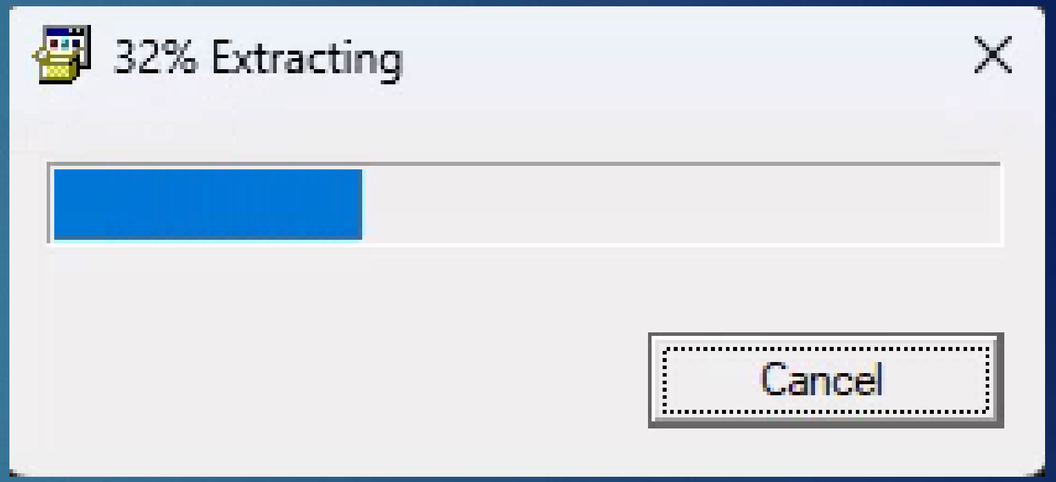
Pwn2Own Vancouver 2024

- Got lucky on the draw: first day, first on that target.
- Launched the exploit…
  - VBox exploit works and then…

Pwn2Own Vancouver 2024

- Got lucky on the draw: first day, first on that target.
- Launched the exploit…
  - VBox exploit works and then…

very_stealthy_exploit.png



Pwn2Own Vancouver 2024

- Got lucky on the draw: first day, first on that target.
- Launched the exploit…
  - VBox exploit works and then…

```
Successfully escaped VirtualBox !
Starting EoP in 5...
launcherDelDuringUpd - INFO -  Entering exploit main with rounds=0x10000!
launcherDelDuringUpd - INFO -  main: Going to elevate pid 9372
exploitDelDuringUpd - INFO -  exploit: entered with loglvl=10
exploitDelDuringUpd - INFO -  exploit: process_name is DelDuringUpd\exploit.py
exploitDelDuringUpd - INFO -  Waiting consume process to start...
exploitDelDuringUpd - INFO -  Consume process is ready, starting exploit !
```

Pwn2Own Vancouver 2024

- Got lucky on the draw: first day, first on that target.

- Launched the exploit…
  - VBox exploit works and then…
  - Wait…

```
Successfully escaped VirtualBox !
Starting EoP in 5...
launcherDelDuringUpd - INFO -  Entering exploit main with rounds=0x10000!
launcherDelDuringUpd - INFO -  main: Going to elevate pid 9372
exploitDelDuringUpd - INFO -  exploit: entered with loglvl=10
exploitDelDuringUpd - INFO -  exploit: process_name is DelDuringUpd\exploit.py
exploitDelDuringUpd - INFO -  Waiting consume process to start...
exploitDelDuringUpd - INFO -  Consume process is ready, starting exploit !
```

Pwn2Own Vancouver 2024

- Got lucky on the draw: first day, first on that target.

- Launched the exploit…
  - VBox exploit works and then…
  - Wait…
  - Wait…

```
Successfully escaped VirtualBox !
Starting EoP in 5...
launcherDelDuringUpd - INFO -  Entering exploit main with rounds=0x10000!
launcherDelDuringUpd - INFO -  main: Going to elevate pid 9372
exploitDelDuringUpd - INFO -  exploit: entered with loglvl=10
exploitDelDuringUpd - INFO -  exploit: process_name is DelDuringUpd\exploit.py
exploitDelDuringUpd - INFO -  Waiting consume process to start...
exploitDelDuringUpd - INFO -  Consume process is ready, starting exploit !
```

Pwn2Own Vancouver 2024

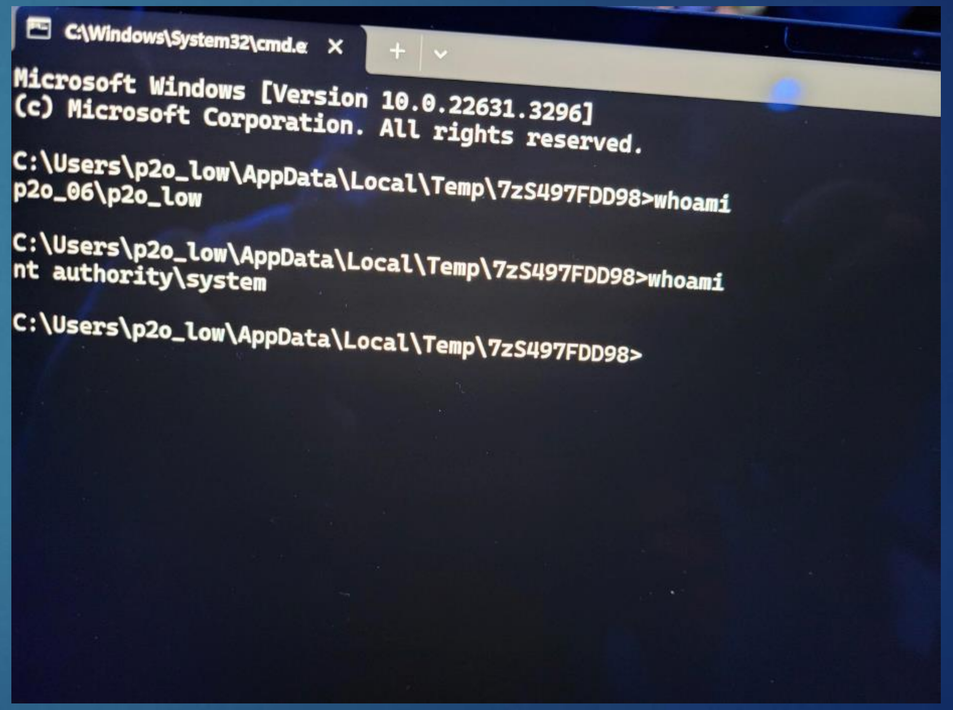
- Got lucky on the draw: first day, first on that target.

- Launched the exploit…
  - VBox exploit works and then…
  - Wait…
  - Wait…
  - Wait…

```
Successfully escaped VirtualBox !
Starting EoP in 5...
launcherDelDuringUpd - INFO -  Entering exploit main with rounds=0x10000!
launcherDelDuringUpd - INFO -  main: Going to elevate pid 9372
exploitDelDuringUpd - INFO -  exploit: entered with loglvl=10
exploitDelDuringUpd - INFO -  exploit: process_name is DelDuringUpd\exploit.py
exploitDelDuringUpd - INFO -  Waiting consume process to start...
exploitDelDuringUpd - INFO -  Consume process is ready, starting exploit !
```

Pwn2Own Vancouver 2024

- Got lucky on the draw: first day, first on that target.
- Launched the exploit…
  - VBox exploit works and then…
  - Wait…
  - Wait…
  - Wait…
  - Worked on first try!!



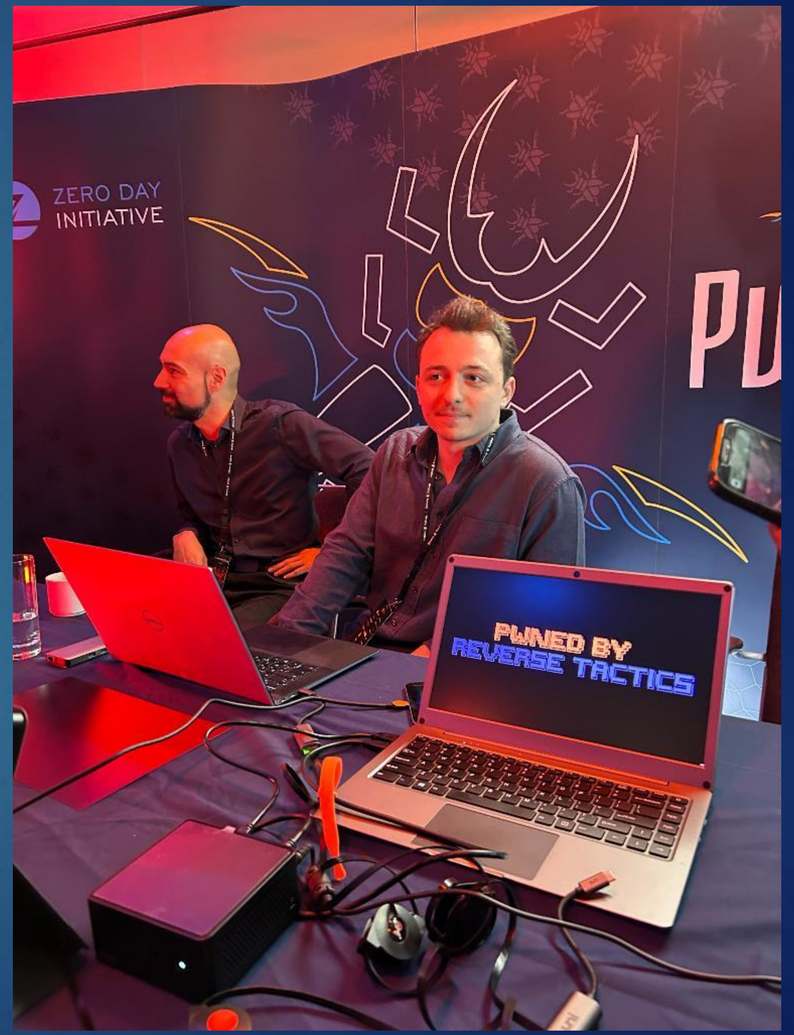
Pwn2Own Vancouver 2024

- Exploits fully written in Python + self-extracting archive
  - VirtualBox Escape 100% stable
  - Windows LPE **not** 100% stable
- Had a full win !
  - Lucky: picked first in the random draw
  - No bug collisions

SUCCESS - Bruno PUJOS and Corentin BAYET from REverse Tactics (@Reverse_Tactics) combined two Oracle VirtualBox bugs - including a buffer overflow - along with a Windows UAF to escape the guest OS and execute code as SYSTEM on the host OS. This fantastic research earns them $90,000 and 9 Master of Pwn points.

Pwn2Own Berlin 2025

- Also had an entry at Pwn2Own this year
  - This time targeting VMware ESXi



Your turn !

- Want to learn VM escapes ?
  - Available seats for our training “Bug hunting in Hypervisors”
  - **https://www.reversetactics.com/trainings/**

RE TACTICS

QUESTIONS ?

contact@reversetactics.com