Attacking hypervisors: A practical case

GreHack

New is not always better............

ESCAPING FROM VIRTUALBOX

RE TACTICS

Who am I ?

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

CORENTIN BAYET

Last year talk

Pwn2Own Vancouver 2024

- Participated at Pwn2Own in March
  - Hacking contest organized by ZDI
  - Rewarded for demonstrating 0-day exploits on popular targets
  - Has a virtualization category

Target	Prize	Master of Pwn Points	Eligible for Add-on Prize
Oracle VirtualBox	$40,000	4	Yes
VMware Workstation	$80,000	8	Yes
VMware ESXi	$150,000	15	No
Microsoft Hyper-V Client	$250,000	25	Yes

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

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

Plan

- **01** Definitions
- **02** Vulnerability research
- **03** Exploit Development
- **04** Conclusion

Plan

01

Definitions

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

Plan

01

Definitions

02

Vulnerability research

Step 0: Setup

- Need a way to easily debug the Hypervisor
  - For Virtualbox: GDB / Windbg
  - Not much to say
- Need a way to easily test things from the guest
  - And reach interesting code paths of the hypervisor
- **How do we easily communicate with the hypervisor from the guest ?**

How to reach vulnerable code

- Communications channels through MMIO, PMIO, DMA, Hypercalls
  - Read/write access to physical memory
  - Execute privileged instructions
- You need ring-0 privilege
  - **So you are supposed to write kernel drivers**

- Kernel drivers
  - Written in compiled and low-level languages (usually C)
  - Hell to compile
  - Dependent of the operating system
    - Dependent of the operating system **VERSION**

- **I don't want to do this every time I want to test something**

How to reach vulnerable code

- **Chipsec**
  - Framework originally developed 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 !

How to reach vulnerable code

```
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)
```

RE TACTICS

Step 1: State of the art

- Very important step, not to neglect
  - MUST put time into it
- Find generic information on the target
  - Public documentation
  - Source code organization
    - Architecture
  - Is it fuzzed ?
    - How ?

Step 1: State of the art

- Prior related security work

- Study previous vulnerabilities
  - Understand common attack surfaces
  - Note exploit techniques
    - What kind of vulnerabilities are actually exploitable
    - Might be useful later
  - Extract **vulnerable patterns**
    - The kind of bugs that can be found in code base
  - Take time to really understand the bugs
    - Even reproduce them if needed
    - Might find some variants…

- This phase should give you list a of ideas
  - Write a list !

State of the art: CVE-2023-21988

- Uninitialized memory read in VirtualBox
  - Found and exploited by @MajorTomSec 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.

RE TACTICS

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

```c
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

```c
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

Step 2: Finding the needles

- 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

Step 2: Finding the needles

- 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 !

Step 3: Targeted research

- 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…

Step 3: Targeted research

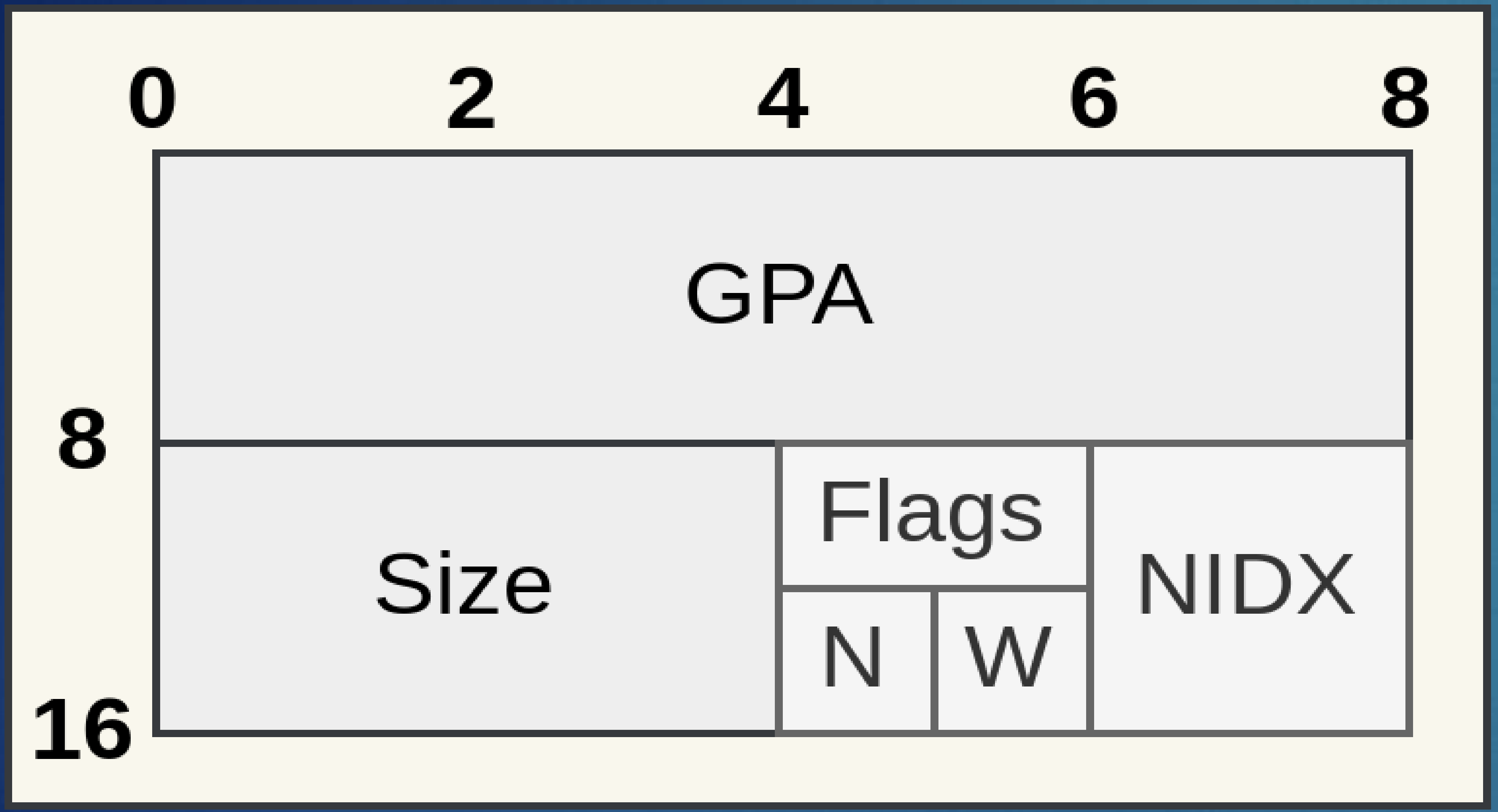
```
#ifdef VIRTIO_VBUF_ON_STACK
                PVIRTQBUF pVirtqBuf = virtioCoreR3VirtqBufAlloc();
                if (!pVirtqBuf)
                {
                    LogRel(("Failed to allocate memory for VIRTQBUF\n"));
                    break;  /* No point in trying to allocate memory for other 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 */
```

RE TACTICS

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

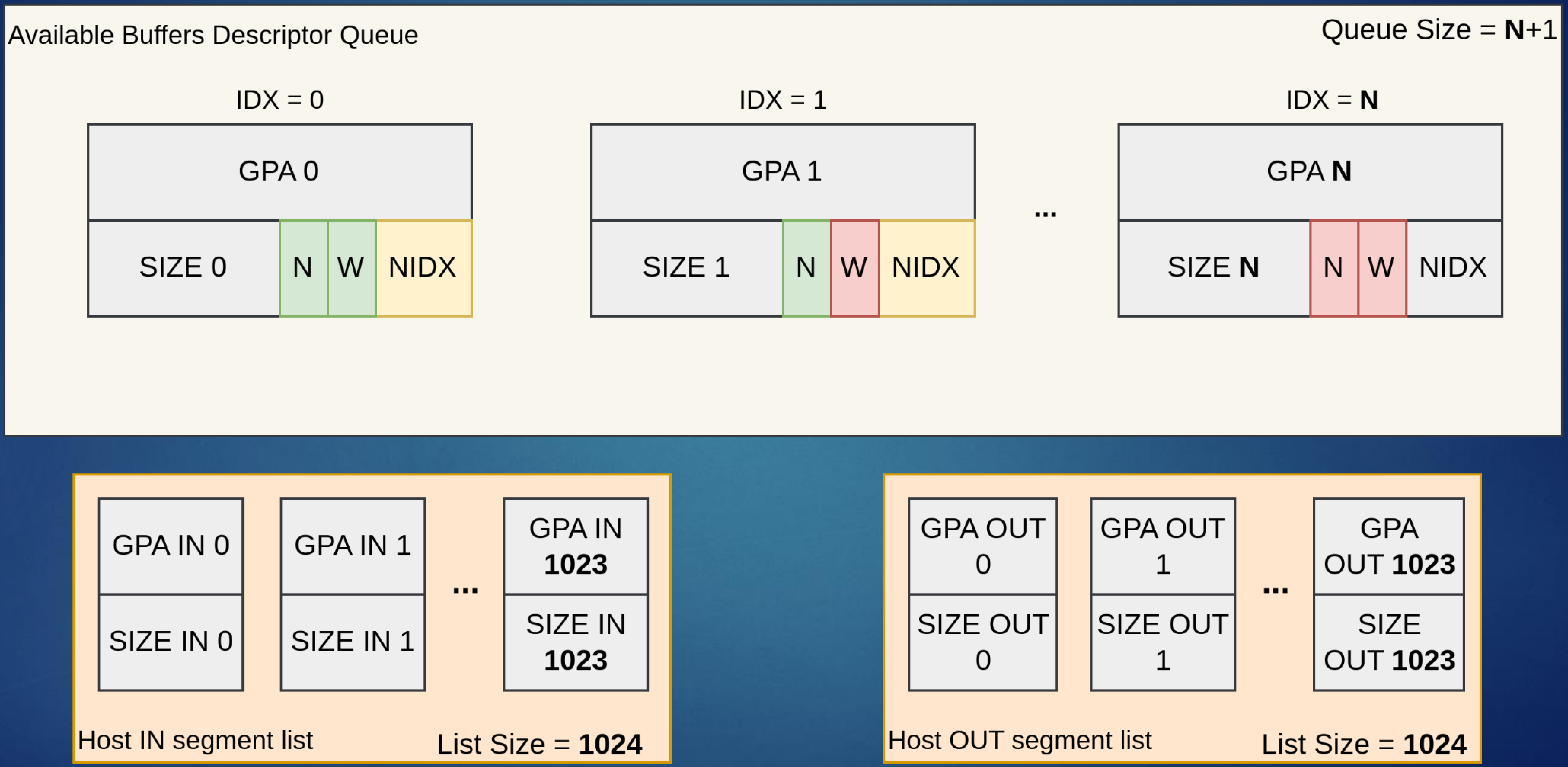
- Function **virtioCoreR3VirtqAvailBufGet**
  - Responsible for parsing a descriptor chain
  - Place it in the **VIRTQBUF** passed in parameter
    - Contains a list of segments

```
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
```

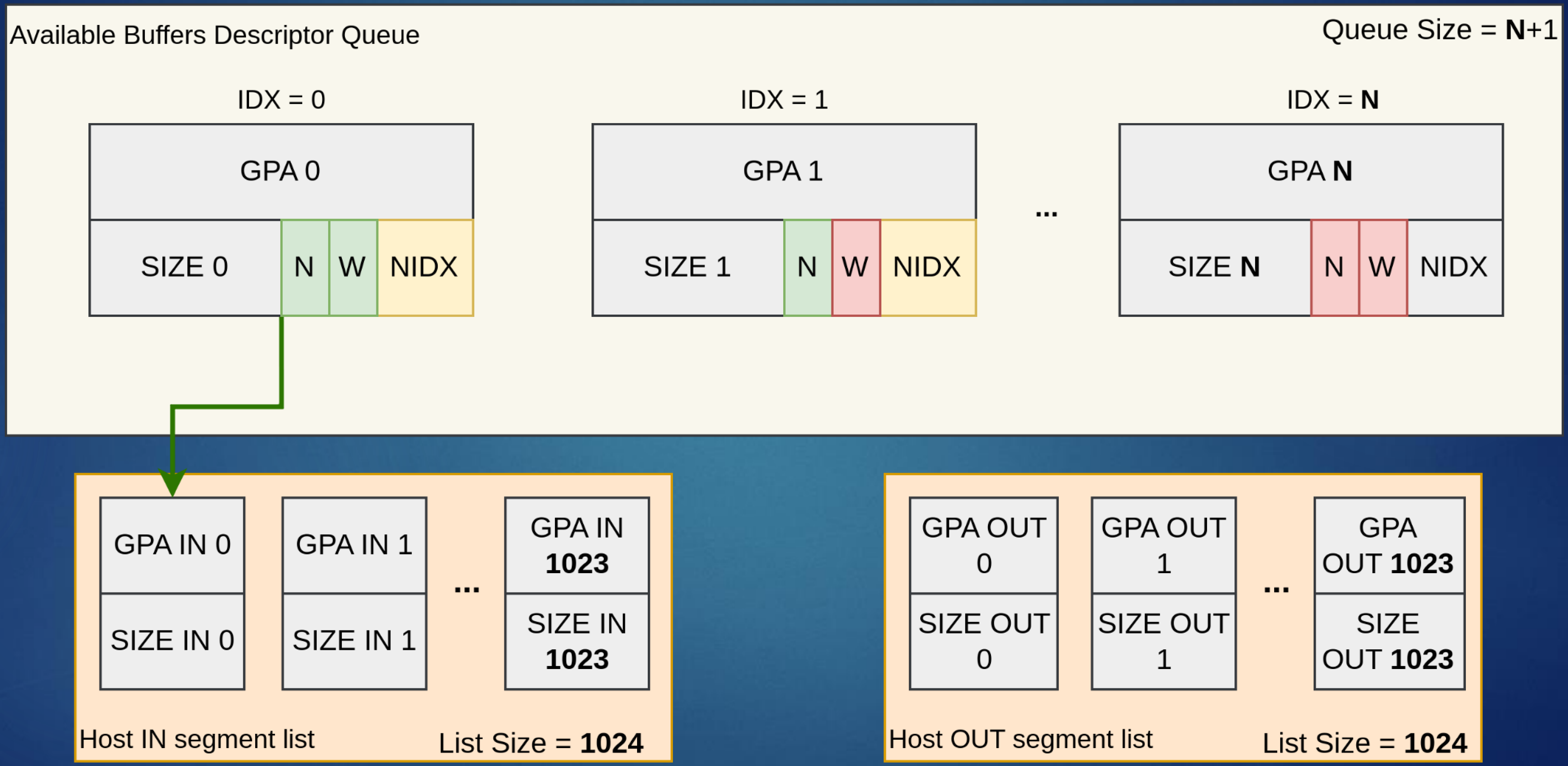
VirtIO – VBox implementation

```
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);
}
```

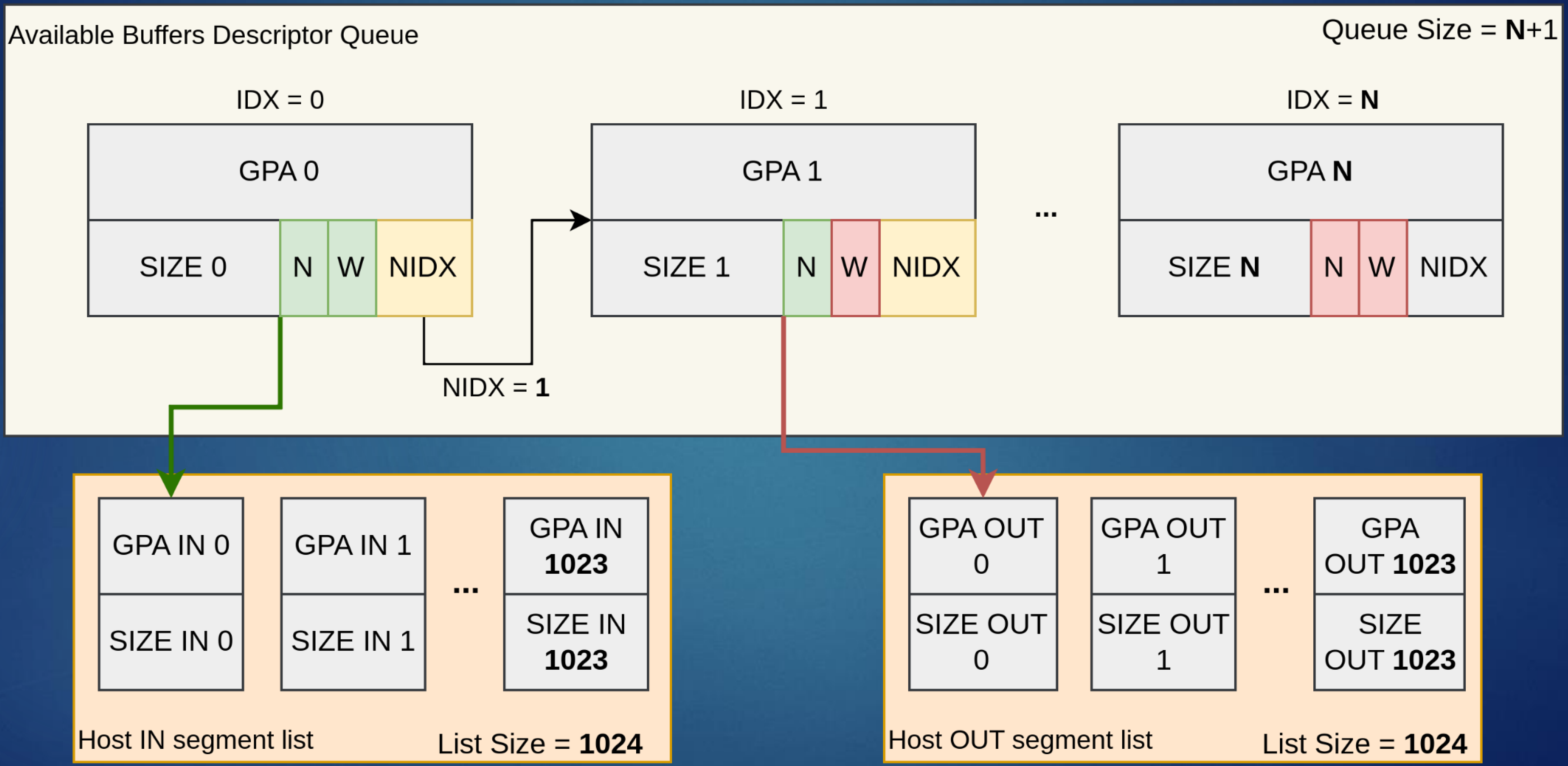
VirtIO – VBox implementation



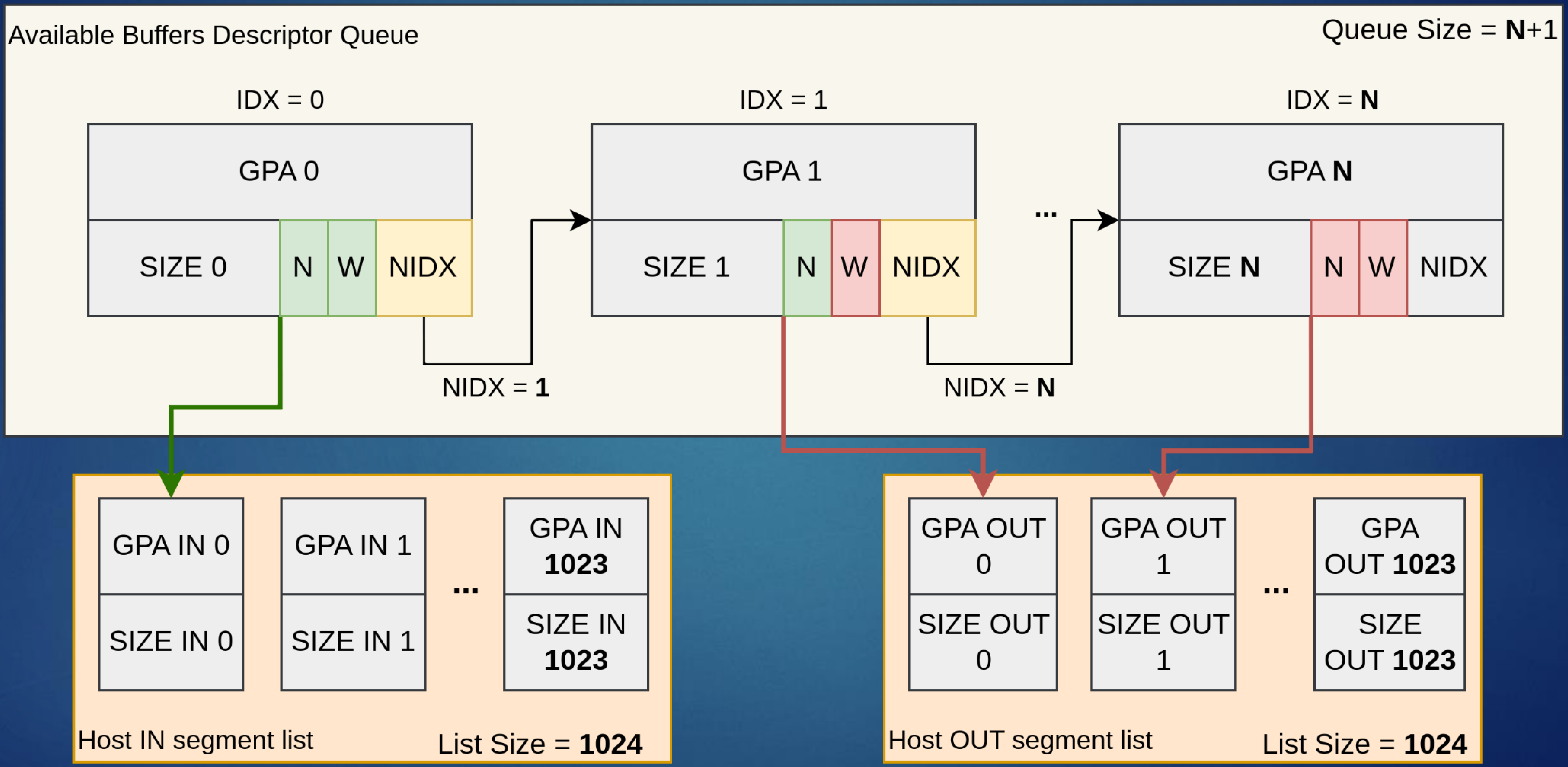
VirtIO – VBox implementation



VirtIO – VBox implementation



VirtIO – VBox implementation



VirtIO – VBox implementation

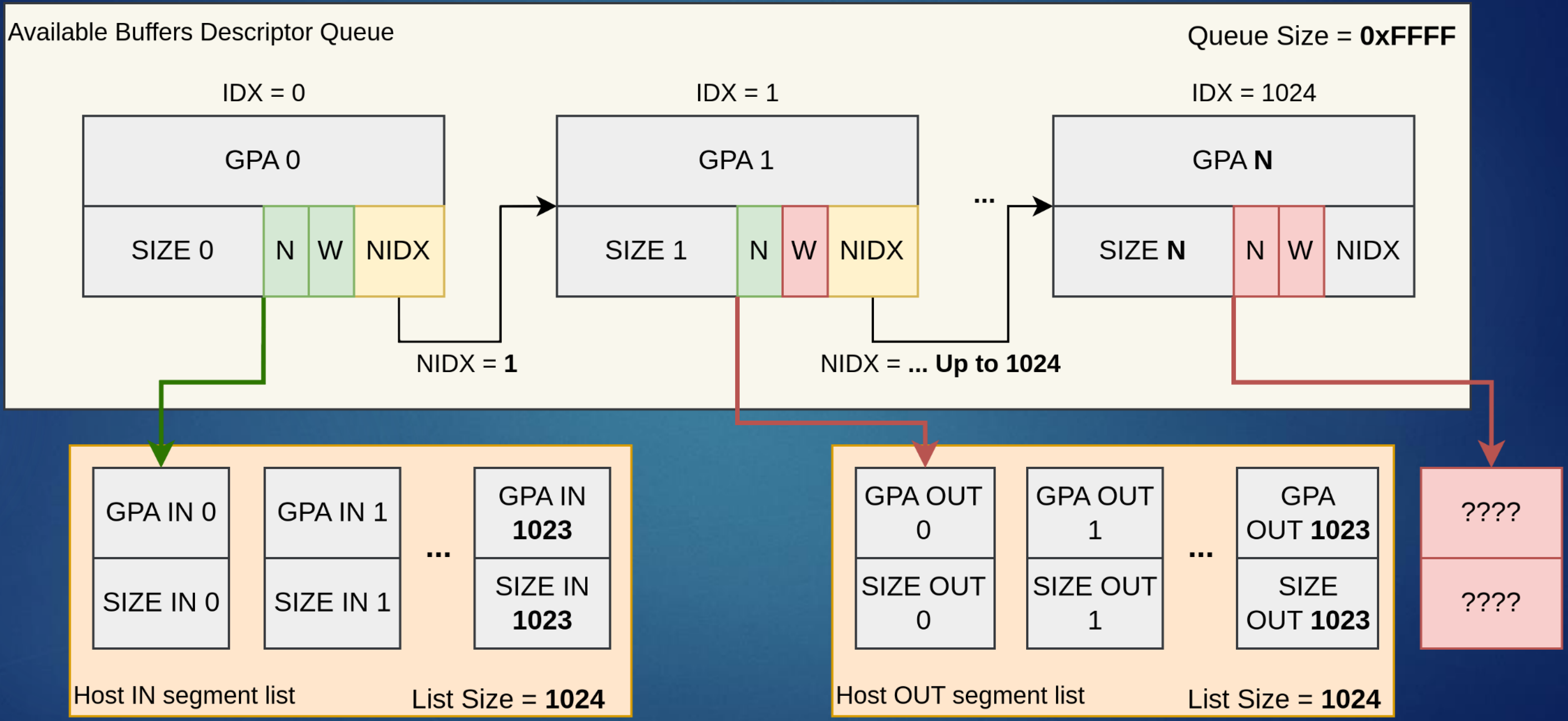
```
int virtioCoreR3VirtqAvailBufGet(PPDMDEVINS pDevIns, PVIRTIOCORE pVirtio, uint16_t uVirtq,
    uint16_t uHeadIdx, PVIRTQBUF pVirtqBuf)
{
    // [...]
    uint32_t cSegsIn, cSegsOut  = 0;
    PVIRTIOSGSEG paSegsIn  = pVirtqBuf->aSegsIn;
    PVIRTIOSGSEG paSegsOut = pVirtqBuf->aSegsOut;

    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
```

Plan

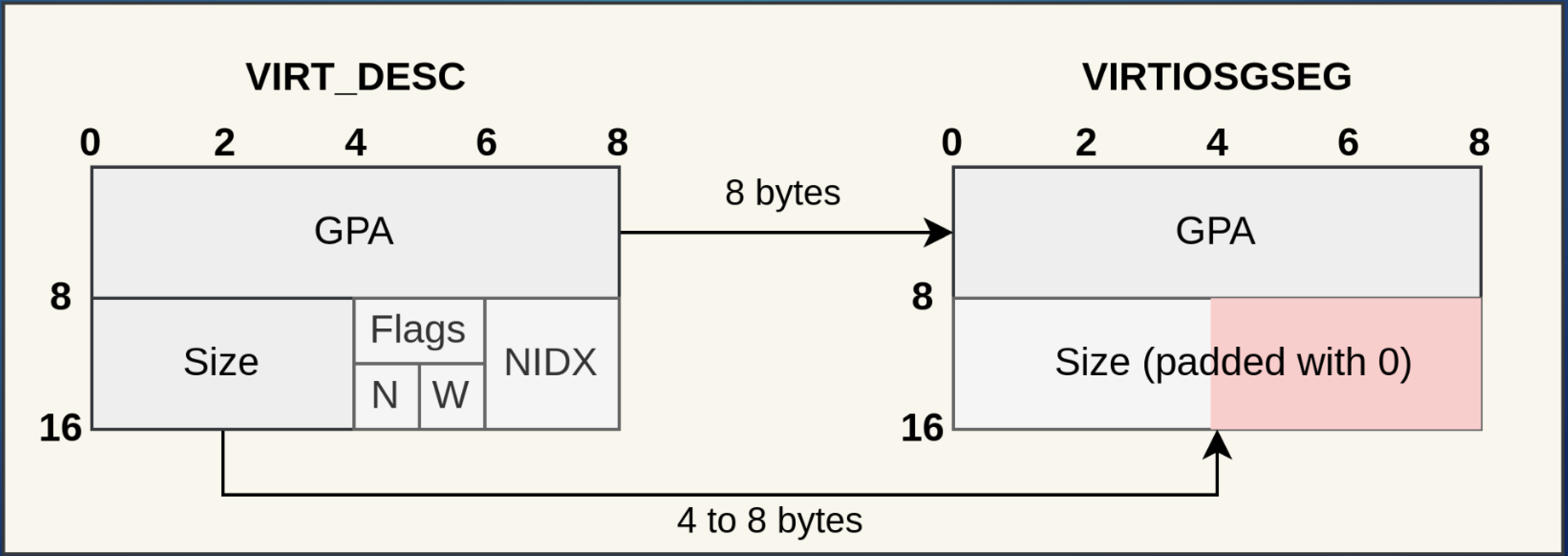
01
Definitions

02
Vulnerability research

03
Exploit Development

CVE-2024-21114 – Impact

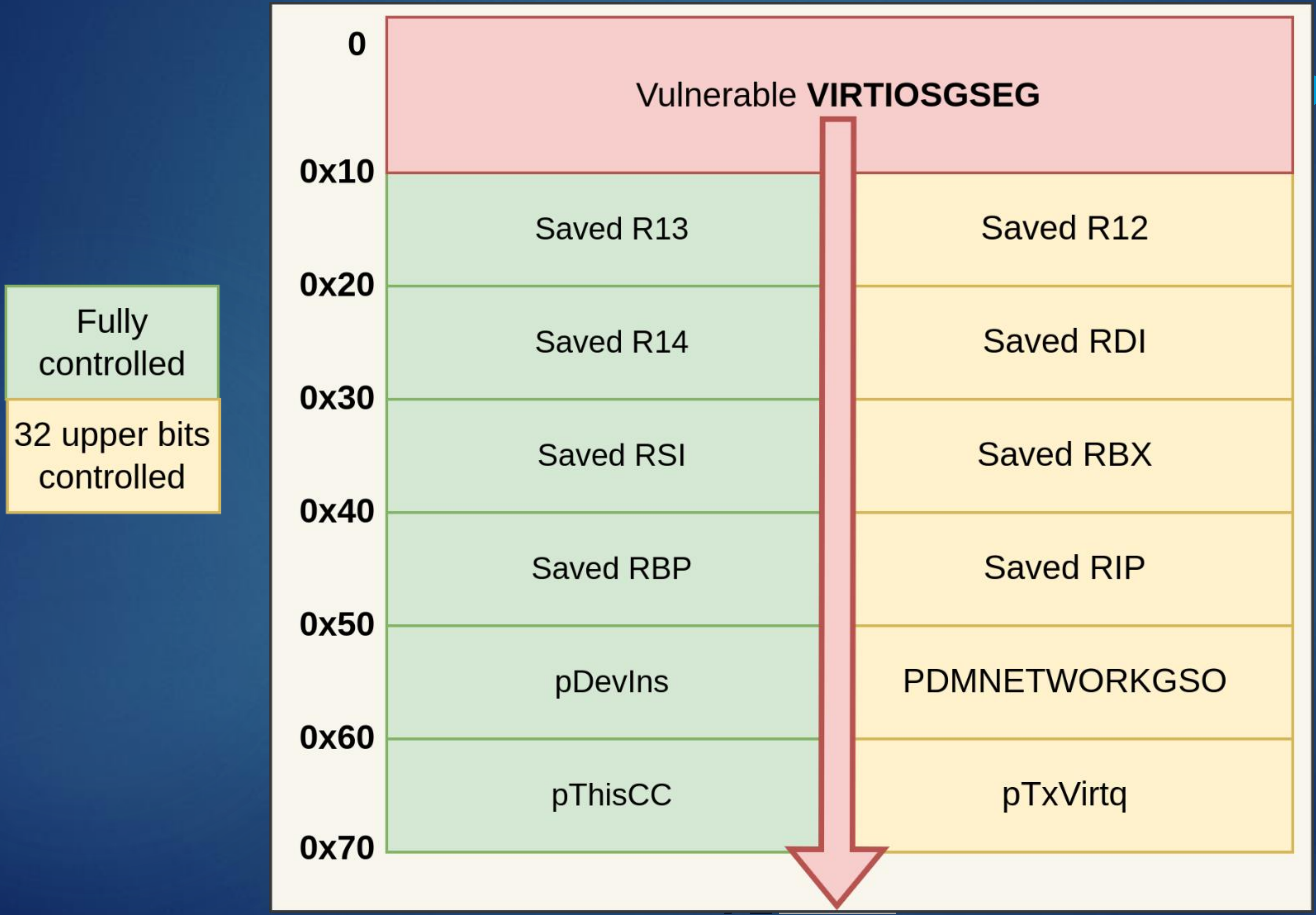
- The **VIRTQBUF** structure can be located on the stack or in the heap
  - VirtIO disk allocate it on the heap
  - VirtIO network card place it on the stack
    - Decide to go with the stack buffer overflow exploit
- Vulnerability allows to write chunks of 0x10 bytes in OOB
  - But only 0xC are controlled, 4 last bytes are 0



CVE-2024-21114 – Exploit

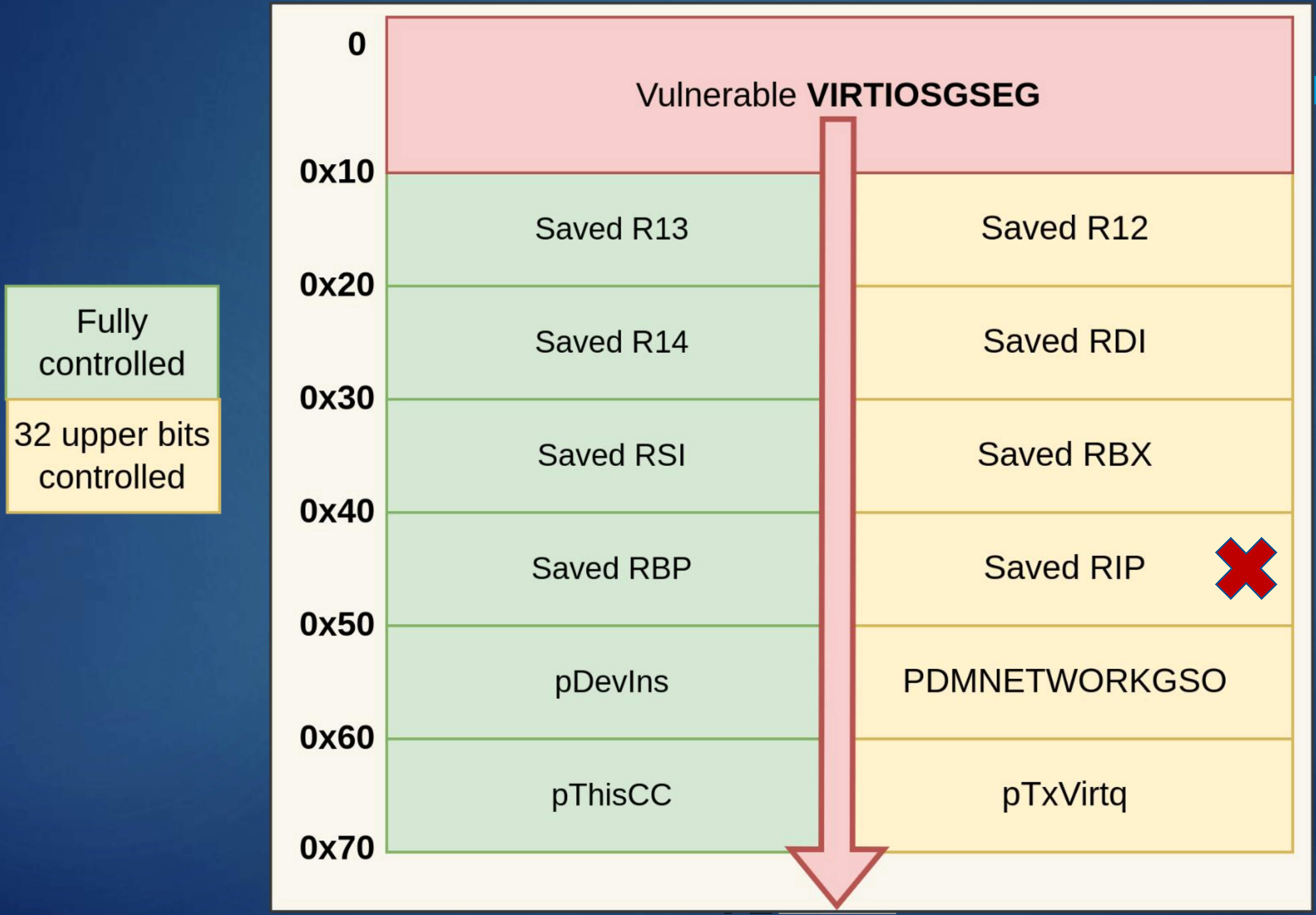
- Can be triggered from the function **virtioNetR3TransmitPkts**
  - In VirtIO network card implementation
- ASLR is defeated thanks to the exploited leak
  - CVE-2024-21121
- VirtualBox compiled without stack canaries
  - Easy win ?

CVE-2024-21114 – Exploit



- Stack frame of **virtioNetR3TransmitPkts**

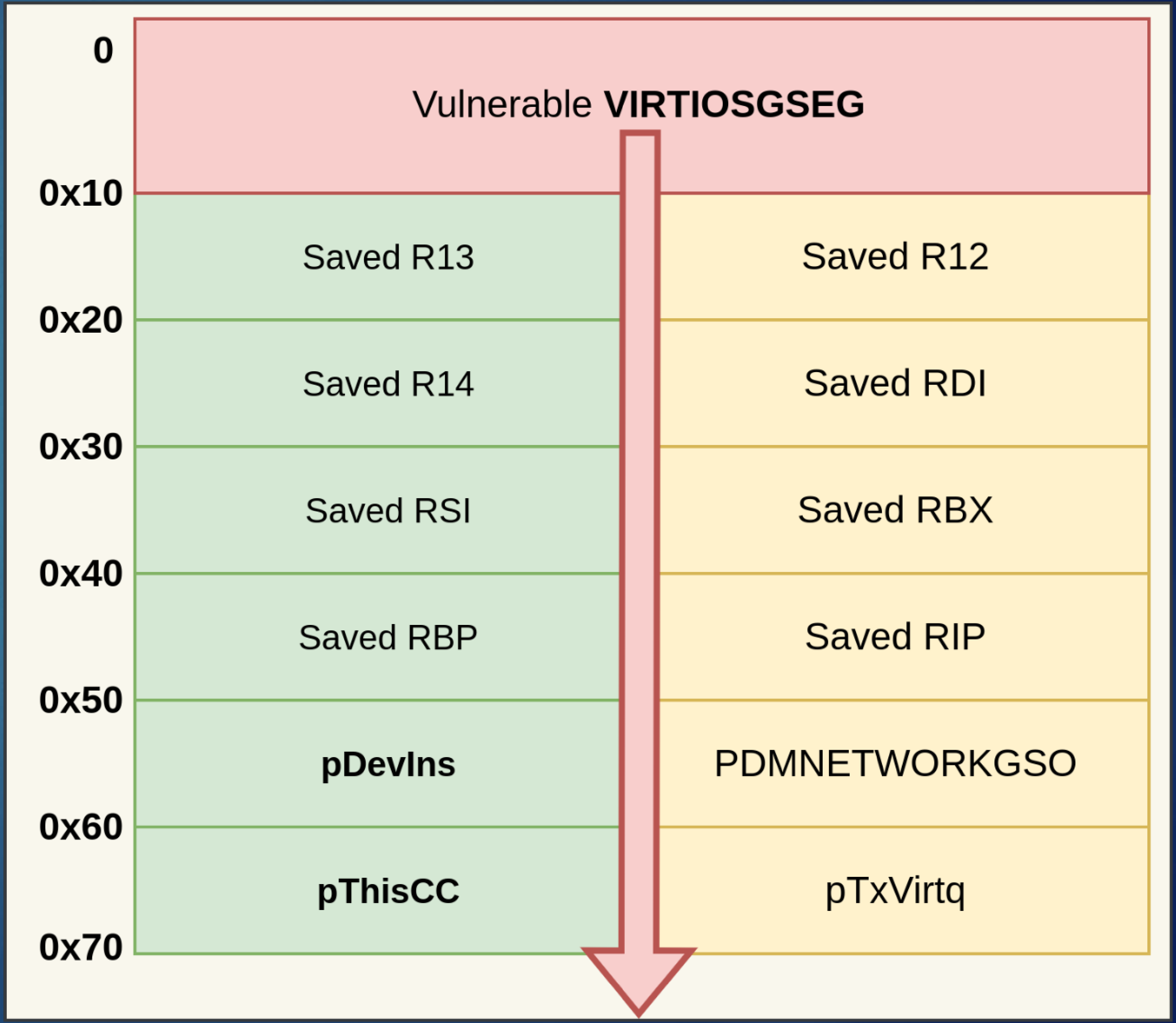
CVE-2024-21114 – Exploit



- Stack frame of **virtioNetR3TransmitPkts**
- 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

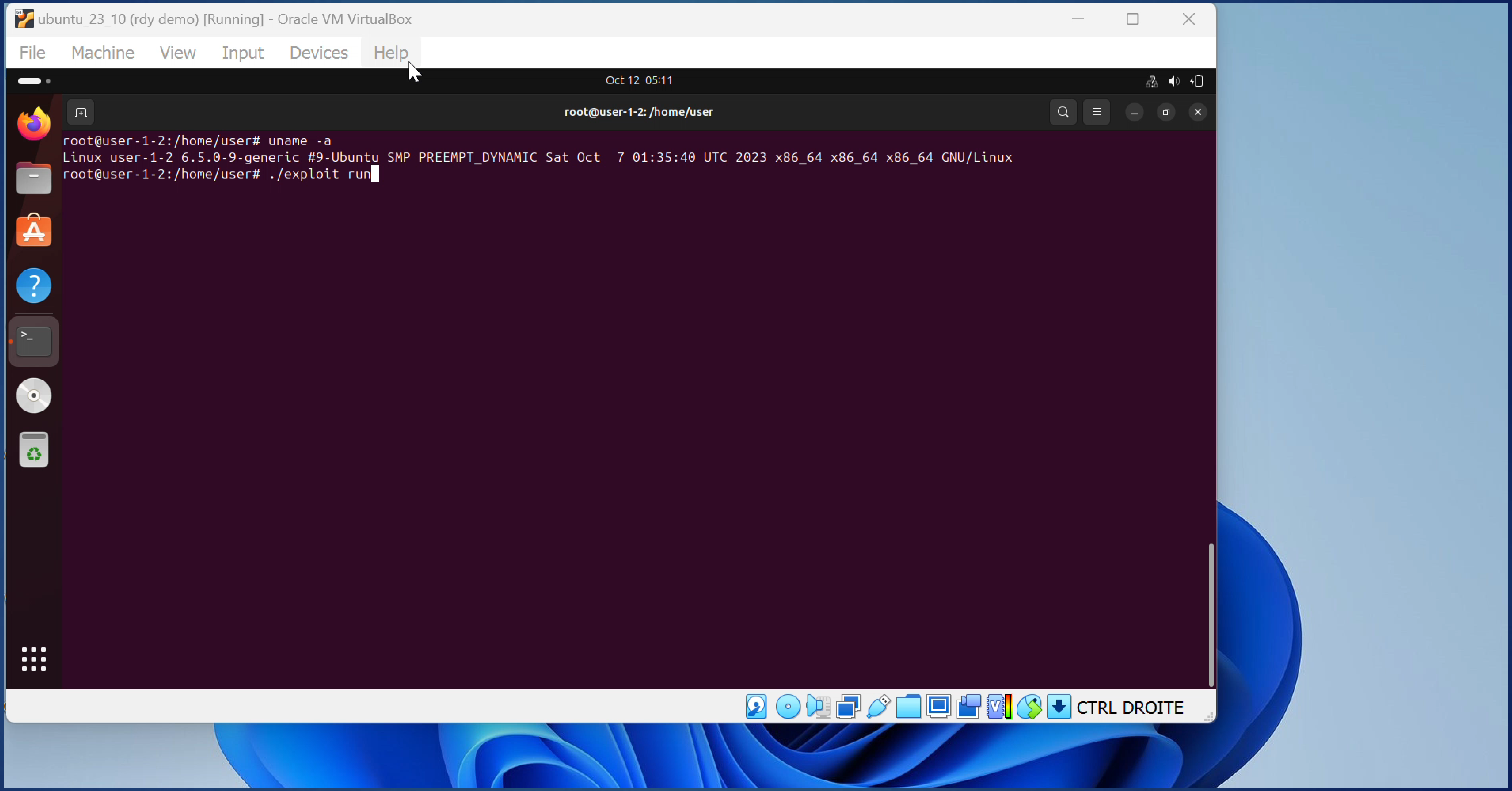
```
static DECLCALLBACK(int) xhciR3WorkerLoop(PPDMDEVINS pDevIns, PPDMTHREAD pThread)
{
    while (pThread->enmState == PDMTHREADSTATE_RUNNING)
    {
        // [..]
        if (!u32Tasks)
        {
            Assert(ASMAtomicReadBool(&pThis->fWrkThreadSleeping));
            rc = PDMDevHlpSUPSemEventWaitNoResume(pDevIns, 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

- Use arbitrary write to overwrite the XHCI thread's stack
  - Target the stack frame of the function waiting on the semaphore
  - Overwrite the saved RIP
- Trigger the wake up of the XHCI thread by sending a command
  - Thread jumps to arbitrary location
  - Bypass CFG
    - Only controls dynamic calls
    - Not the saved RIP on the stack
- ROP to shellcode !
  - WIN !

Demo


ubuntu_23_10 (rdy demo) [Running] - Oracle VM VirtualBox
File
Machine
View
Input
Devices
Help
Oct 12 05:11
root@user-1-2: /home/user
root@user-1-2:/home/user# uname -a
Linux user-1-2 6.5.0-9-generic #9-Ubuntu SMP PREEMPT_DYNAMIC Sat Oct 7 01:35:40 UTC 2023 x86_64 x86_64 x86_64 GNU/Linux
root@user-1-2:/home/user# ./exploit run
CTRL DROITE

Plan

01

Definitions

02

Vulnerability research

03

Exploit Development

04

Conclusion

Pwn2Own Vancouver 2024

- Exploit fully written in Python
  - 100% stable
- Chained with a Windows privilege escalation for Pwn2Own
  - 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.

Conclusion

- Fast and fun project
  - Lasted a month in total
  - Learned a lot on virtualization
  - Improved my tooling
- VirtualBox is a great software to learn about VM escapes
  - Open source and easy to read code
  - There is still some bugs to found
  - Can win a nice bounty at Pwn2Own !

THANK YOU!

contact@reversetactics.com