Breaking Out of the Box
Technical analysis of VirtualBox VM escape with Windows LPE

13 October 2023

Synacktiv

Thomas Bouzerar and Thomas Imbert

Agenda

1 Introduction

2 VirtualBox

3 Windows

4 Conclusion

About us

Thomas Bouzerar

- @MajorTomSec
- Security researcher at Synacktiv

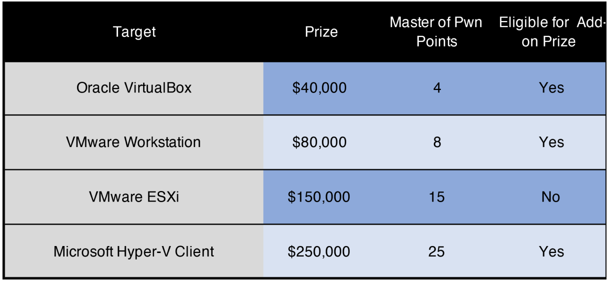
Thomas Imbert

- @masthoon
- Security researcher at Synacktiv

- Synacktiv is hiring!
  - Offensive security company
  - Pentest, Reverse engineering, Development, Incident response
  - Offices in Paris, Toulouse, Rennes, Lyon, Lille

Pwn2Own

- Ethical hacking contest organized by Zero Day Initiative (ZDI)
- Edition Pwn2Own Vancouver 2023 in March
  - Targets: Virtualization, browsers, OS, Tesla, ...



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

* Add-on prize: Additional price for chaining with a Windows LPE

Pwn2Own - Rules

VirtualBox escape with Windows LPE

- 2 months to prepare
- 3 attempts of 10 minutes maximum
- Exploit chain:
  - *VirtualBox* Virtual Machine to Host code execution
  - *Windows* host unprivileged user to *SYSTEM* account
- Total prize: $90,000

Agenda

1 Introduction

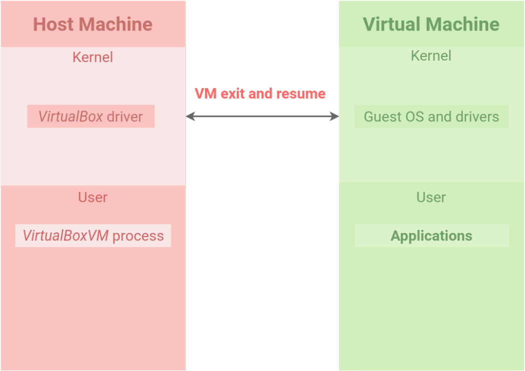
2 VirtualBox

3 Windows

4 Conclusion

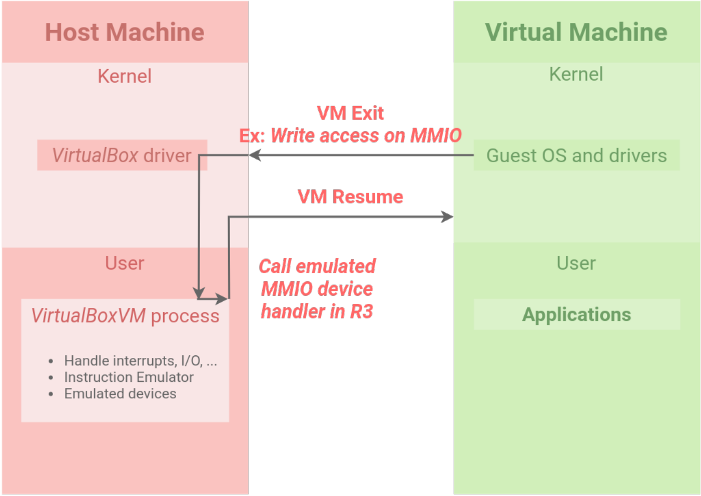
Introduction to VirtualBox

- Type 2 hypervisor
- Open-source



Virtual Box Components

Introduction to VirtualBox (2)



Virtual Box Attack Surface

VirtualBox

- Quite large codebase
  - No prior knowledge of the target
  - Where do we start ?

VirtualBox

- Quite large codebase
  - No prior knowledge of the target
  - Where do we start ?

- Latest version when we started looking at VirtualBox was:

VirtualBox 7.0.6

- Released January 17 2023
- Latest major update was VirtualBox 7.0.0 (released October 10 2022)
  - Introduces new virtual devices (IOMMU, TPM)
  - EHCI/XHCI open-sourcing
  - EFI supports Secure Boot

VirtualBox

- Quite large codebase
  - No prior knowledge of the target
  - Where do we start ?

- Latest version when we started looking at VirtualBox was:

VirtualBox 7.0.6

- Released January 17 2023
- Latest major update was VirtualBox 7.0.0 (released October 10 2022)
  - Introduces new virtual devices (IOMMU, TPM)
  - EHCI/XHCI open-sourcing
  - EFI supports Secure Boot

- According to Pwn2Own rules, target guest OS is now Windows 11
  - TPM might be a device of interest here

VirtualBox - TPM

- Trusted Platform Module (TPM)

Wikipedia

Trusted Platform Module is an international standard for a secure cryptoprocessor, a dedicated microcontroller designed to secure hardware through integrated cryptographic keys.

The term can also refer to a chip conforming to the standard.
One of Windows 11’s system requirements is TPM 2.0.

VirtualBox

- TPM is mandatory since Windows 11
- Easy to interact with

VirtualBox

- TPM is mandatory since Windows 11
- Easy to interact with
- Looks like a good first device to look at

VirtualBox

- TPM is mandatory since Windows 11
- Easy to interact with

- Looks like a good first device to look at

- `grep` for ”TPM” in the code base
  - Most interesting results are:
    - `./src/libs/libtpms/*`
    - `./src/VBox/Devices/Security/DevTpm.cpp`
    - `./src/VBox/Devices/Security/DrvTpmEmu.cpp`
    - `./src/VBox/Devices/Security/DrvTpmEmuTpms.cpp`
    - `./src/VBox/Devices/Security/DrvTpmHost.cpp`

VirtualBox

- TPM is mandatory since Windows 11
- Easy to interact with
- Looks like a good first device to look at

- `grep` for ”TPM” in the code base
  - Most interesting results are:
    - `./src/libs/libtpms/*`
    - `./src/VBox/Devices/Security/DevTpm.cpp`
    - `./src/VBox/Devices/Security/DrvTpmEmu.cpp`
    - `./src/VBox/Devices/Security/DrvTpmEmuTpms.cpp`
    - `./src/VBox/Devices/Security/DrvTpmHost.cpp`
- `libtpms` is an open-source library capable of emulating TPM in hypervisors, also used by QEMU

VirtualBox - TPM

- Time to dig in the code

VirtualBox - TPM

- Time to dig in the code

- `./src/VBox/Devices/Security/DevTpm.cpp`
  - Manages the TPM virtual device

VirtualBox - TPM

- Time to dig in the code

- `./src/VBox/Devices/Security/DevTpm.cpp`
  - Manages the TPM virtual device

- `./src/VBox/Devices/Security/DrvTpmEmu.cpp`
  - Implementation of a virtual TPM using swtpm (yet another library)

VirtualBox - TPM

- Time to dig in the code

- `./src/VBox/Devices/Security/DevTpm.cpp`
  - Manages the TPM virtual device
- `./src/VBox/Devices/Security/DrvTpmEmu.cpp`
  - Implementation of a virtual TPM using swtpm (yet another library)
- `./src/VBox/Devices/Security/DrvTpmHost.cpp`
  - TPM bridge to the host TPM chip

VirtualBox - TPM

- Time to dig in the code

- `./src/VBox/Devices/Security/DevTpm.cpp`
  - Manages the TPM virtual device
- `./src/VBox/Devices/Security/DrvTpmEmu.cpp`
  - Implementation of a virtual TPM using swtpm (yet another library)
- `./src/VBox/Devices/Security/DrvTpmHost.cpp`
  - TPM bridge to the host TPM chip
- `./src/VBox/Devices/Security/DrvTpmEmuTpms.cpp`
  - TPM emulator using libtpms

VirtualBox - TPM

- Time to dig in the code

- `./src/VBox/Devices/Security/DevTpm.cpp`
  - Manages the TPM virtual device
- `./src/VBox/Devices/Security/DrvTpmEmu.cpp`
  - Implementation of a virtual TPM using swtpm (yet another library)
- `./src/VBox/Devices/Security/DrvTpmHost.cpp`
  - TPM bridge to the host TPM chip
- `./src/VBox/Devices/Security/DrvTpmEmuTpms.cpp`
  - TPM emulator using libtpms

- Reading through the code, we can quickly focus on `DevTpm.cpp` and `DrvTpmEmuTpms.cpp`
  - Responsible for emulating and interacting with the **default** virtual TPM device

VirtualBox - DevTpm

- `DevTpm.cpp` creates a new virtual TPM and binds it to a VM each time it boots

VirtualBox - DevTpm

- `DevTpm.cpp` creates a new virtual TPM and binds it to a VM each time it boots
- Set-up of the device is done in `tpmR3Construct` for the Ring-3 side
  - Registers a new MMIO region for the VM at a fixed location (`0xFED40000`)
  - Read and write handlers to the MMIO region are `tpmMmioRead` and `tpmMmioWrite`

VirtualBox - DevTpm

- `DevTpm.cpp` creates a new virtual TPM and binds it to a VM each time it boots
- Set-up of the device is done in `tpmR3Construct` for the Ring-3 side
  - Registers a new MMIO region for the VM at a fixed location (`0xFED40000`)
  - Read and write handlers to the MMIO region are `tpmMmioRead` and `tpmMmioWrite`

- Most of the TPM emulator logic is done in R3
- Invoked methods from R0 will often jump to the R3 implementation

VirtualBox - DevTpm

- `DevTpm.cpp` creates a new virtual TPM and binds it to a VM each time it boots
- Set-up of the device is done in `tpmR3Construct` for the Ring-3 side
  - Registers a new MMIO region for the VM at a fixed location (`0xFED40000`)
  - Read and write handlers to the MMIO region are `tpmMmioRead` and `tpmMmioWrite`

- Most of the TPM emulator logic is done in R3
- Invoked methods from R0 will often jump to the R3 implementation

- So let's look into those MMIO handlers!

VirtualBox - tpmMmioRead

```c
static DECLCALLBACK(VBOXSTRICTRC) tpmMmioRead(PPDMDEVINS pDevIns, void *pvUser, RTGCPHYS off, void *pv, unsigned cb)
{
    /* ...*/
    uint64_t u64;
    rc = tpmMmioFifoRead(pDevIns, pThis, pLoc, bLoc, uReg, &u64, cb);
    /* ... */
}
```

VirtualBox - tpmMmioFifoRead

```c
static VBOXSTRICTRC tpmMmioFifoRead(PPDMDEVINS pDevIns, PDEVTPM pThis, PDEVTPMLOCALITY pLoc,
                                    uint8_t bLoc, uint32_t uReg, uint64_t *pu64, size_t cb)
{
    /* ... */
    if (pThis->offCmdResp <= pThis->cbCmdResp - cb)
    {
        memcpy(pu64, &pThis->abCmdResp[pThis->offCmdResp], cb);
        pThis->offCmdResp += (uint32_t)cb;
    }
    else
        memset(pu64, 0xff, cb);
    /* ... */
}
```

VirtualBox - tpmMmioFifoRead

```
static VBOXSTRICTRC tpmMmioFifoRead(PPDMDEVINS pDevIns, PDEVTPM pThis, PDEVTPMLOCALITY pLoc,
                                    uint8_t bLoc, uint32_t uReg, uint64_t *pu64, size_t cb)
{
    /* ... */
    if (pThis->offCmdResp <= pThis->cbCmdResp - cb)
    {
        memcpy(pu64, &pThis->abCmdResp[pThis->offCmdResp], cb);
        pThis->offCmdResp += (uint32_t)cb;
    }
    else
        memset(pu64, 0xff, cb);
    /* ... */
}
```

- No check on cb !

VirtualBox - tpmMmioFifoRead

```
memcpy(pu64, &pThis->abCmdResp[pThis->offCmdResp], cb);
```

- Stack buffer overflow with controlled data
    - `pu64` points to a stack allocated 64-bit integer
    - `abCmdResp` is a shared buffer for input commands and response data
    - `cb` is the size of the read as requested by the VMEXIT trap

VirtualBox - tpmMmioFifoRead

```
memcpy(pu64, &pThis->abCmdResp[pThis->offCmdResp], cb);
```

- Stack buffer overflow with controlled data
  - `pu64` points to a stack allocated 64-bit integer
  - `abCmdResp` is a shared buffer for input commands and response data
  - `cb` is the size of the read as requested by the VMEXIT trap
- So, how do we trigger it ?

VirtualBox - tpmMmioFifoRead

```
memcpy(pu64, &pThis->abCmdResp[pThis->offCmdResp], cb);
```

- Stack buffer overflow with controlled data
  - `pu64` points to a stack allocated 64-bit integer
  - `abCmdResp` is a shared buffer for input commands and response data
  - `cb` is the size of the read as requested by the VMEXIT trap
- So, how do we trigger it ?
- A few ideas:
  - Instructions which trigger atomic loads of >8 bytes
    - AVX instructions
    - x87 instructions (FRSTOR, ...)
  - DMA

VirtualBox - tpmMmioFifoRead

```
memcpy(pu64, &pThis->abCmdResp[pThis->offCmdResp], cb);
```

- Stack buffer overflow with controlled data
  - `pu64` points to a stack allocated 64-bit integer
  - `abCmdResp` is a shared buffer for input commands and response data
  - `cb` is the size of the read as requested by the VMEXIT trap
- So, how do we trigger it ?
- A few ideas:
  - Instructions which trigger atomic loads of >8 bytes
    - AVX instructions
    - x87 instructions (FRSTOR, ...)
  - DMA
- But we don't even understand the architecture of the hypervisor yet!

VirtualBox - Going deeper

- No information leak so far
  - Can we make our own ?

First approach

- Windows DLL base addresses are aligned on 0x10000
- Partial RIP overwrite
  - We need control over the size of the overflow
  - Overwrite part of the response buffer with host pointers
  - Trigger the bug a second time for code execution

VirtualBox - Going deeper

- We need control over the size of the overflow
- VirtualBox exposes multiple API methods for interacting with the guest physical memory:
  - `PGMPhysRead`
  - `PGMPhysWrite`

VirtualBox - Going deeper

- We need control over the size of the overflow
- VirtualBox exposes multiple API methods for interacting with the guest physical memory:
  - `PGMPhysRead`
  - `PGMPhysWrite`
- They go through the MMIO handlers in case of MMIO addresses!

VirtualBox - Going deeper

- We need control over the size of the overflow
- VirtualBox exposes multiple API methods for interacting with the guest physical memory:
  - `PGMPhysRead`
  - `PGMPhysWrite`
- They go through the MMIO handlers in case of MMIO addresses!

VirtualBox - Going deeper

- We need control over the size of the overflow
- VirtualBox exposes multiple API methods for interacting with the guest physical memory:
  - `PGMPhysRead`
  - `PGMPhysWrite`
- They go through the MMIO handlers in case of MMIO addresses!
- Many wrappers around them: `pdmR3DevHlp_PhysRead`, `pdmR0DevHlp_PhysRead`, ...

VirtualBox - Going deeper

- We need control over the size of the overflow
- VirtualBox exposes multiple API methods for interacting with the guest physical memory:
  - `PGMPhysRead`
  - `PGMPhysWrite`
- They go through the MMIO handlers in case of MMIO addresses!
- Many wrappers around them: `pdmR3DevHlp_PhysRead`, `pdmR0DevHlp_PhysRead`, ...
  - Or around those wrappers themselves: `PDMDevHlpPhysRead`, `PDMDevHlpPhysReadMeta`, ...

VirtualBox - Going deeper

- We need control over the size of the overflow
- VirtualBox exposes multiple API methods for interacting with the guest physical memory:
  - `PGMPhysRead`
  - `PGMPhysWrite`
- They go through the MMIO handlers in case of MMIO addresses!
- Many wrappers around them: `pdmR3DevHlp_PhysRead`, `pdmR0DevHlp_PhysRead`, ...
  - Or around those wrappers themselves: `PDMDevHlpPhysRead`, `PDMDevHlpPhysReadMeta`, ...

VirtualBox - Going deeper

- We need control over the size of the overflow
- VirtualBox exposes multiple API methods for interacting with the guest physical memory:
  - `PGMPhysRead`
  - `PGMPhysWrite`
- They go through the MMIO handlers in case of MMIO addresses!
- Many wrappers around them: `pdmR3DevHlp_PhysRead`, `pdmR0DevHlp_PhysRead`, ...
  - Or around those wrappers themselves: `PDMDevHlpPhysRead`, `PDMDevHlpPhysReadMeta`, ...
- Basically, `grep` for `PhysRead` or `PhysWrite`

VirtualBox - Going deeper

- We need control over the size of the overflow
- VirtualBox exposes multiple API methods for interacting with the guest physical memory:
  - `PGMPhysRead`
  - `PGMPhysWrite`
- They go through the MMIO handlers in case of MMIO addresses!
- Many wrappers around them: `pdmR3DevHlp_PhysRead`, `pdmR0DevHlp_PhysRead`, ...
  - Or around those wrappers themselves: `PDMDevHlpPhysRead`, `PDMDevHlpPhysReadMeta`, ...
- Basically, `grep` for `PhysRead` or `PhysWrite`
  - Most of those methods end up calling `PGMPhysRead`/`PGMPhysWrite`

VirtualBox - Going deeper

VMMDev device

- VMMDev is a virtual device used for Host <-> Guest communication
  - Most features are disabled by default, but the device itself is enabled!

VirtualBox - Going deeper

VMMDev device

- VMMDev is a virtual device used for Host <-> Guest communication
  - Most features are disabled by default, but the device itself is enabled!

HGCM Requests

- Host-Guest Communication Manager
- The guest can send requests to the host
  - Simple RPC protocol
  - Format well documented by other researchers
- Call parameters may be integers/buffers
  - Read from the guest memory (DMA)

VirtualBox - Controlled overflow size

Guest physical read with arbitrary size

- Use HGCM calls as a DMA read oracle around `PGMPhysRead`
  - Remap the MMIO region to a virtual address using `MmMapIoSpace`
  - Make a dummy HGCM call with a `VMMDevHGCMParmType_LinAddr` buffer parameter
    - Address of the parameter is the remapped virtual address
    - Arbitrary size can be given

VirtualBox - Controlled overflow size

Guest physical read with arbitrary size

- Use HGCM calls as a DMA read oracle around `PGMPhysRead`
  - Remap the MMIO region to a virtual address using `MmMapIoSpace`
  - Make a dummy HGCM call with a `VMMDevHGCMParmType_LinAddr` buffer parameter
    - Address of the parameter is the remapped virtual address
    - Arbitrary size can be given

- We get an arbitrary physical read in the guest with controlled size!
  - Proof of Concept gives us RIP control (full or partial overwrite)

VirtualBox - Controlled overflow size

Guest physical read with arbitrary size

- Use HGCM calls as a DMA read oracle around `PGMPhysRead`
  - Remap the MMIO region to a virtual address using `MmMapIoSpace`
  - Make a dummy HGCM call with a `VMMDevHGCMParmType_LinAddr` buffer parameter
    - Address of the parameter is the remapped virtual address
    - Arbitrary size can be given

- We get an arbitrary physical read in the guest with controlled size!
  - Proof of Concept gives us RIP control (full or partial overwrite)

Initial approach

- Create our own infoleak (partial RIP overwrite)
  - Overwrite part of the response buffer with host pointers

VirtualBox - Controlled overflow size

Guest physical read with arbitrary size

- Use HGCM calls as a DMA read oracle around `PGMPhysRead`
  - Remap the MMIO region to a virtual address using `MmMapIoSpace`
  - Make a dummy HGCM call with a `VMMDevHGCMParmType_LinAddr` buffer parameter
    - Address of the parameter is the remapped virtual address
    - Arbitrary size can be given

- We get an arbitrary physical read in the guest with controlled size!
  - Proof of Concept gives us RIP control (full or partial overwrite)

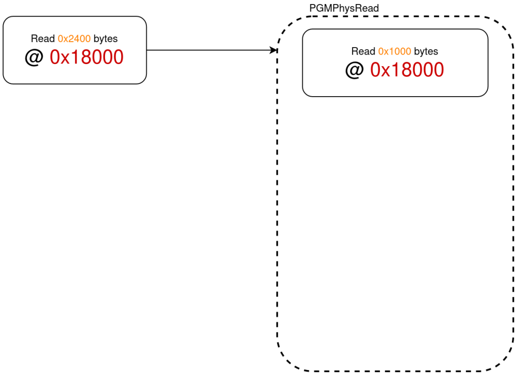
Initial approach

- Create our own infoleak (partial RIP overwrite)
  - Overwrite part of the response buffer with host pointers
- No suitable gadget candidate :-(

VirtualBox - PGMPhysRead

Read 0x2400 bytes
@ 0x18000

VirtualBox - PGMPhysRead

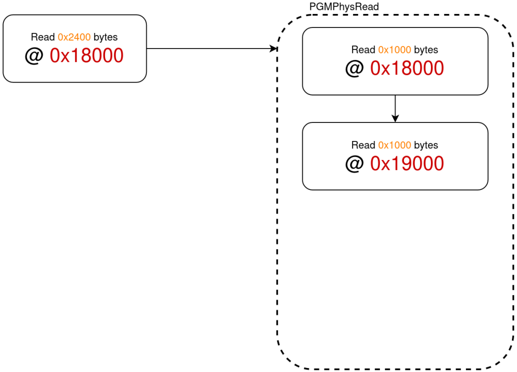


Read 0x2400 bytes
@ 0x18000

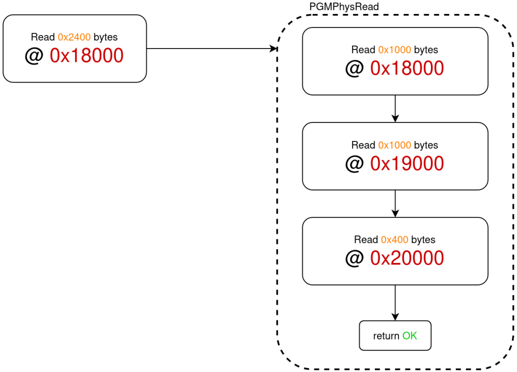
PGMPhysRead

Read 0x1000 bytes
@ 0x18000

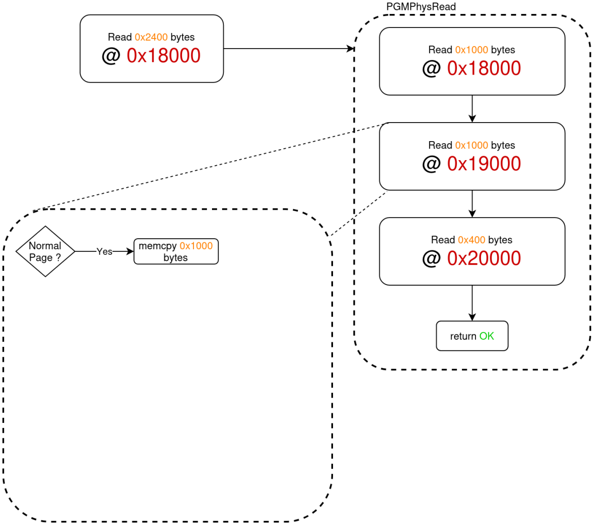
VirtualBox - PGMPhysRead



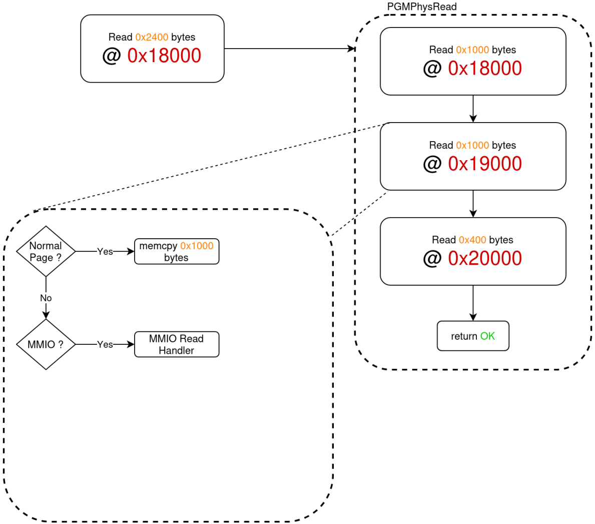
VirtualBox - PGMPhysRead



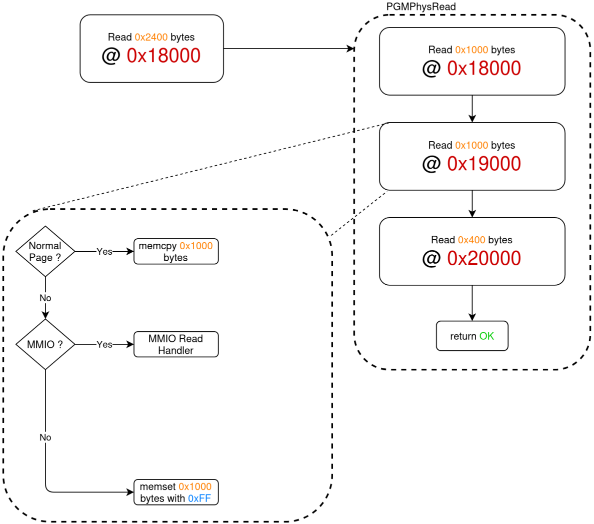
VirtualBox - PGMPhysRead



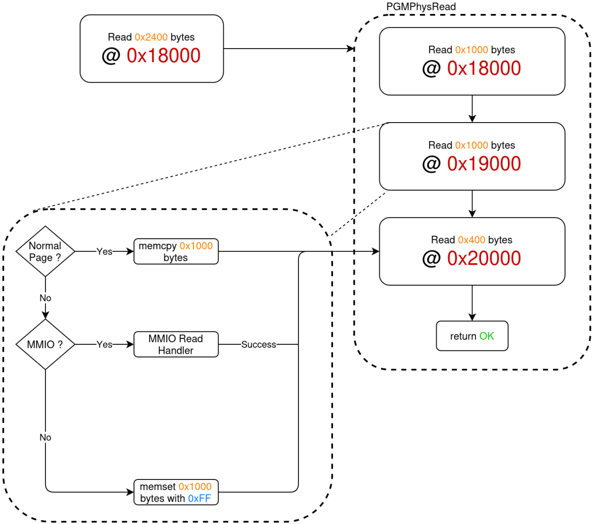
VirtualBox - PGMPhysRead



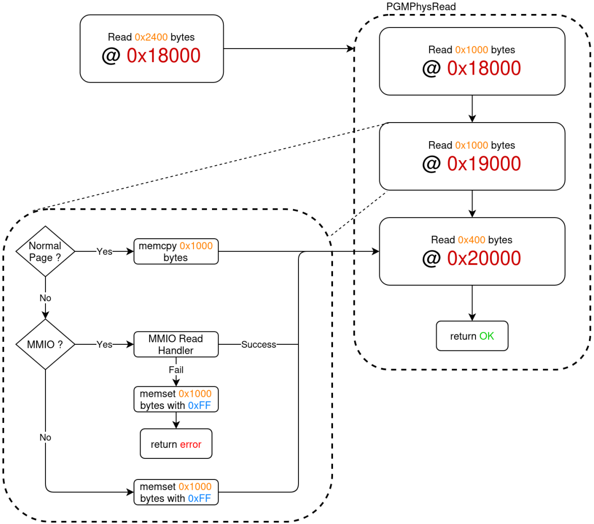
VirtualBox - PGMPhysRead



VirtualBox - PGMPhysRead



VirtualBox - PGMPhysRead



VirtualBox - PGMPhysRead

- Any call to `PGMPhysRead` which *does not* validate its return value would potentially leak data
  - We can leak any kind of data!

Uninitialized memory read in low level API

VirtualBox - PGMPhysRead

Finding a good leak candidate

- Need to find a call to `PGMPhysRead` from a *default device* which:
  - Reads in a stack buffer
  - Does not validate the return value
  - Writes back the data at a known location

VirtualBox - PGMPhysRead

Finding a good leak candidate

- Need to find a call to `PGMPhysRead` from a *default device* which:
  - Reads in a stack buffer
  - Does not validate the return value
  - Writes back the data at a known location

eXtensible Host Controller Interface (xHCI)

- Does a lot of physical memory read/write accesses
- Copies data from arbitrary physical addresses to other arbitrary physical addresses

VirtualBox - xHCI

```
static unsigned xhciR3ConfigureDevice(PPDMDEVINS pDevIns, PXHCI pThis, uint64_t uInpCtxAddr, uint8_t uSlotID, bool fDC)
{
    /* ... */
    XHCI_DEV_CTX   dc_inp; // sizeof(XHCI_DEV_CTX) = 0x400
    XHCI_DEV_CTX   dc_out;
    /* ... */
    PDMDevHlpPCIPhysReadMeta(pDevIns, GCPhysInpSlot, &dc_inp, num_inp_ctx * sizeof(XHCI_DS_ENTRY));
    /* ... */
    for (uDCI = 2; uDCI < 32; ++uDCI)
    {
        /* ... */
        dc_out.entry[uDCI].ep = dc_inp.entry[uDCI].ep;
        /* ... */
    }
    /* ... */
    PDMDevHlpPCIPhysWriteMeta(pDevIns, GCPhysOutSlot, &dc_out, num_out_ctx * sizeof(XHCI_DS_ENTRY));
    /* ... */
}
```

- Almost 0x400 bytes of Uninitialized stack memory read!

VirtualBox - Exploitation

- Information leak allows reading:
  - Return values
  - Stack canaries

VirtualBox - Exploitation

- Information leak allows reading:
  - Return values
  - Stack canaries
- Spoiler: for performance reasons, there is no stack canary in VirtualBox...

VirtualBox - Exploitation

- Information leak allows reading:
  - Return values
  - Stack canaries

- Spoiler: for performance reasons, there is no stack canary in VirtualBox...

VirtualBox - Exploitation

- Information leak allows reading:
    - Return values
    - Stack canaries
- Spoiler: for performance reasons, there is no stack canary in VirtualBox...
- Defeat ASLR, build a ROP-chain, execute a shellcode

VirtualBox - Exploitation

- Information leak allows reading:
  - Return values
  - Stack canaries
- Spoiler: for performance reasons, there is no stack canary in VirtualBox...
- Defeat ASLR, build a ROP-chain, execute a shellcode

Shellcode

- Use exported method `RTLdrGetSystemSymbol` from `VBoxRT.DLL` to resolve external symbols
- Call `PGMPhysRead` to read PE file from guest memory
- Write PE file in `%ProgramData%\a.exe`
- Call `WinExec` to execute stage 2

VirtualBox - Exploitation

- Information leak allows reading:
  - Return values
  - Stack canaries
- Spoiler: for performance reasons, there is no stack canary in VirtualBox...
- Defeat ASLR, build a ROP-chain, execute a shellcode

Shellcode

- Use exported method `RTLdrGetSystemSymbol` from `VBoxRT.DLL` to resolve external symbols
- Call `PGMPhysRead` to read PE file from guest memory
- Write PE file in `%ProgramData%\a.exe`
- Call `WinExec` to execute stage 2

- 100% reliable VM escape!

Agenda

1 Introduction

2 VirtualBox

3 Windows

4 Conclusion

Windows LPE

Exploit chain

- *VirtualBox* escape exploit
  - *VirtualBox* VM process runs as unprivileged user with Medium Integrity Level
- *Windows* **L**ocal **P**rivilege **E**scalation
  - Large *Windows* attack surface
  - Pwn2Own requires **kernel** mode vulnerability

Research

Objective

- Find a quick and stable bug in a Windows driver
- Exploit it and spawn a **SYSTEM** command prompt

Finding a target

- Static analysis of random drivers in `System32\drivers`
  - Pick ones with interesting imports: `%Probe%`
- Review *IOCTL* handlers for memory corruption or logic bugs
- Many drivers cannot be loaded without administrator access

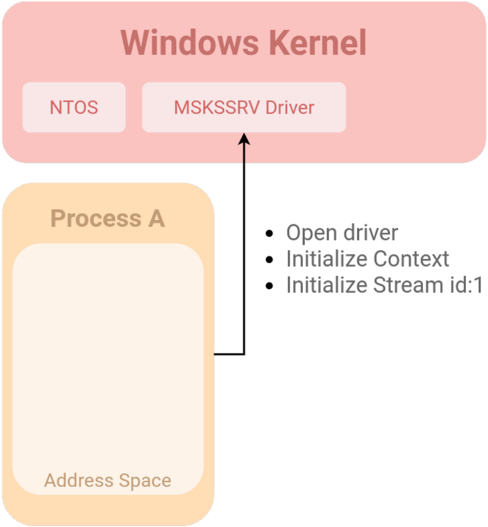
MSKSSRV

- Part of **Microsoft Streaming** component
- Content Streaming between two processes
  - Implemented as shared memory

- Driver automatically loaded on **demand**
  - **Without administrator access**
  - Device path:

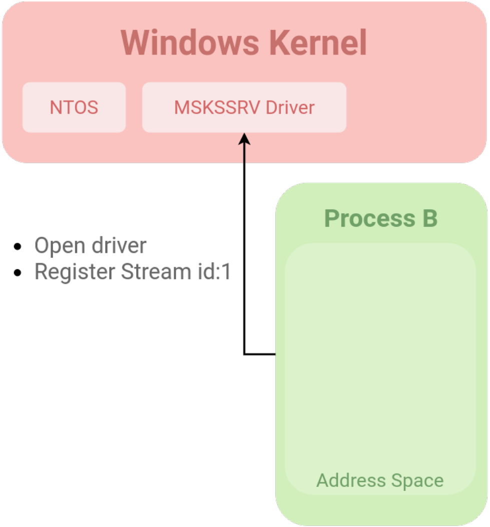
    `\\?\root#system#0000#{3c0d501a-140b-11d1-b40f-00a0c9223196}\{96e080c7-143c-11d1-b40f-00a0c9223196}&{3c0d501a-140b-11d1-b40f-00a0c9223196}`

MSKSSRV - Initialization A



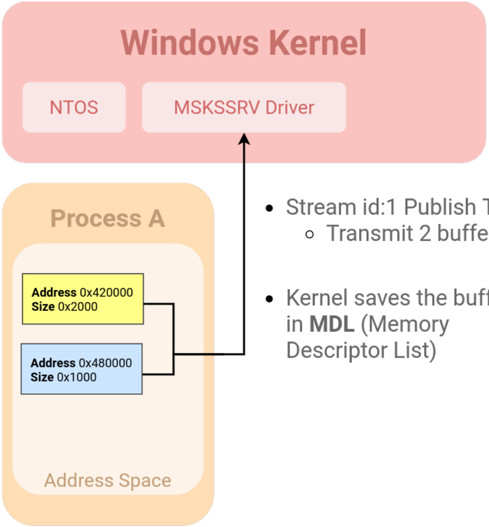
- Open driver
- Initialize Context
- Initialize Stream id:1

MSKSSRV - Initialization B



- Open driver
- Register Stream id:1

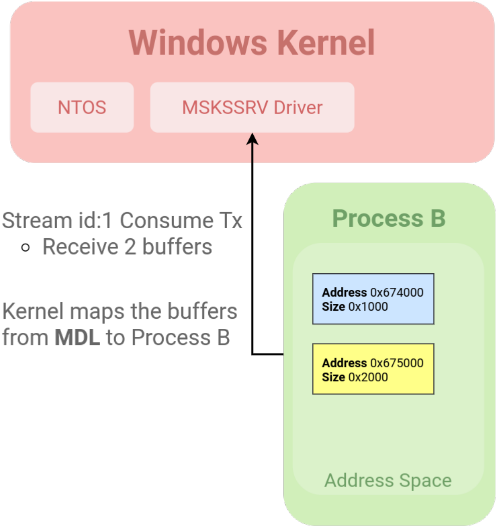
MSKSSRV - Stream Publish



- Stream id:1 Publish Tx
  - Transmit 2 buffers

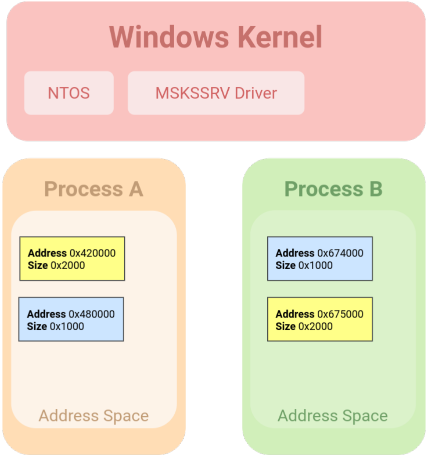
- Kernel saves the buffers in **MDL** (Memory Descriptor List)

MSKSSRV - Stream Consume



- Stream id:1 Consume Tx
  - Receive 2 buffers
- Kernel maps the buffers from **MDL** to Process B

MSKSSRV - Shared Memory



The same buffer in Process A and B is mapped to the same physical address (**Shared Memory**)

MSKSSRV Vulnerability

- MSKSSRV **does NOT validate the address of the buffer**
  - Any virtual address can be mapped **even Kernel mode memory**

```
// Vulnerability in the function FsAllocAndLockMdl (from IOCTL 0x2F0408)
Mdl = IoAllocateMdl(InputAddress, InputSize, 0, 0, NULL);
/*
  MmProbeAndLockPages Invalid Access Mode
    * KernelMode used instead of UserMode
    * The kernel will not check (called Probe) if the address belongs in userland
*/
MmProbeAndLockPages(Mdl, KernelMode, IoWriteAccess);
```

Vulnerability Outcome

- Arbitrary kernel virtual memory may be mapped to user-mode with read and write access
- → **Arbitrary kernel read and write**

MSKSSRV Exploitation

Locate the TOKEN

- Kernel *TOKEN* object describes the security context of the process
- The kernel-mode address of the current process token can be obtained using `NtQuerySystemInformation`

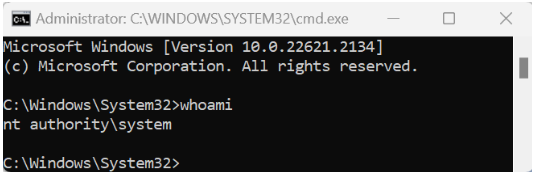
Corrupt the TOKEN

- Map the *TOKEN* to user-mode using the vulnerability
- **Overwrite** the *TOKEN* **privileges bit-field to gain all privileges**

Escalate to SYSTEM

- Using the `SeDebugPrivilege`, hijack a *SYSTEM* process
- Run *SYSTEM* command prompt !

MSKSSRV Result



```
Microsoft Windows [Version 10.0.22621.2134]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\System32>whoami
nt authority\system

C:\Windows\System32>
```

SYSTEM Command Prompt !

- Exploit takes less than 1 second
- 100% stable bug
  - Missing probe are powerful bugs

Agenda

1 Introduction

2 VirtualBox

3 Windows

4 **Conclusion**

Conclusion

Simple bugs

- There are still low hanging fruits

Conclusion

Simple bugs

- There are still low hanging fruits
- There are also deeper bugs

Conclusion

Simple bugs

- There are still low hanging fruits
- There are also deeper bugs

Conclusion

Simple bugs

- There are still low hanging fruits
- There are also deeper bugs

No real mitigation

- No stack canary in VirtualBox

Conclusion

Simple bugs

- There are still low hanging fruits
- There are also deeper bugs

No real mitigation

- No stack canary in VirtualBox
  - Smash the stack like it’s 2010

Conclusion

Simple bugs

- There are still low hanging fruits
- There are also deeper bugs

No real mitigation

- No stack canary in VirtualBox
  - Smash the stack like it's 2010
- Relatively weak mitigations in Windows

Conclusion

Simple bugs

- There are still low hanging fruits
- There are also deeper bugs

No real mitigation

- No stack canary in VirtualBox
  - Smash the stack like it's 2010
- Relatively weak mitigations in Windows

Conclusion

Simple bugs

- There are still low hanging fruits
- There are also deeper bugs

No real mitigation

- No stack canary in VirtualBox
  - Smash the stack like it's 2010
- Relatively weak mitigations in Windows

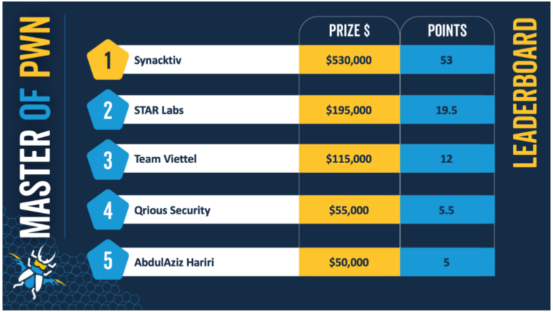
Disable AV

- Defender blocked our first attempt

Conclusion

- 3-bugs chain
  - 2 unique bugs, 1 bug collision (TPM stack buffer overflow)

- We won Pwn2Own!



MASTER OF PWN

LEADERBOARD

		PRIZE $	POINTS
1	Synacktiv	$530,000	53
2	STAR Labs	$195,000	19.5
3	Team Viettel	$115,000	12
4	Qrious Security	$55,000	5.5
5	AbdulAziz Hariri	$50,000	5

- Try it, it's fun!

QUESTIONS?

THANKS FOR YOUR ATTENTION

SYNACKTIV