



# Hunting and Exploiting Recursive MMIO Flaws in QEMU/KVM

Qiuhao Li, Gaoning Pan, Hui He, Chunming Wu

*Harbin Institute of Technology*

*Zhejiang University*

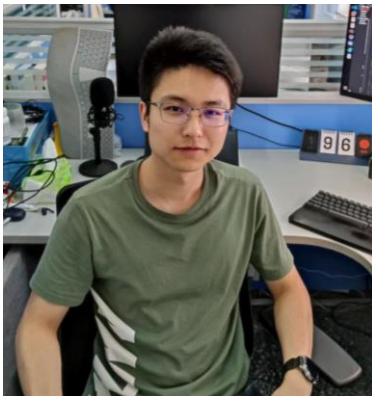
# About Us

## Qiu hao Li (李秋豪)

Graduate Student at Harbin Institute of Technology

Research Areas: Cloud Security and Fuzzing

Twitter: [@Qiu hao Li](https://twitter.com/Qiu hao Li)



## Gaoning Pan (潘高宁)

Ph.D. Candidate at Zhejiang University

Research Areas: System and Virtualization Security

Twitter: [@hades24495092](https://twitter.com/hades24495092)

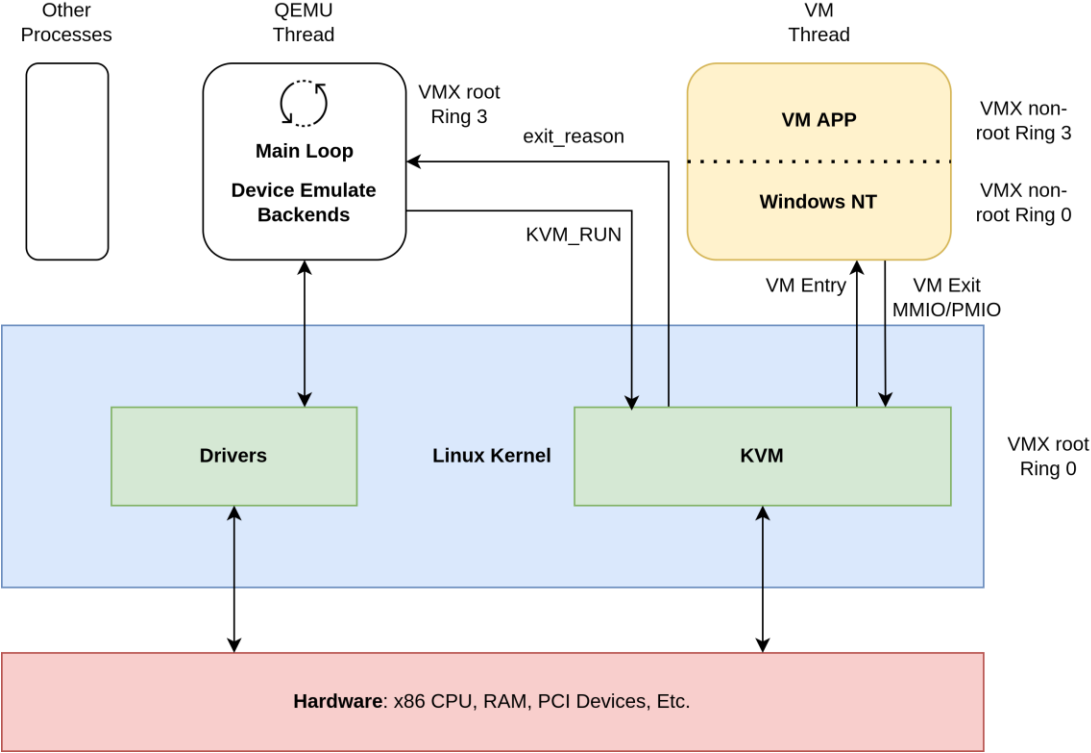


# Outline

1. **Introduction**
2. Root Causes
3. Hunting Flaws
4. Case Study
5. Exploitation
6. Mitigations
7. Thoughts



# QEMU/KVM Architecture



# Outline

1. Introduction
2. **Root Causes**
3. Hunting Flaws
4. Case Study
5. Exploitation
6. Mitigations
7. Thoughts



# Recursive MMIO

MMIO: EPT Misconfig

DMA: memcpy()

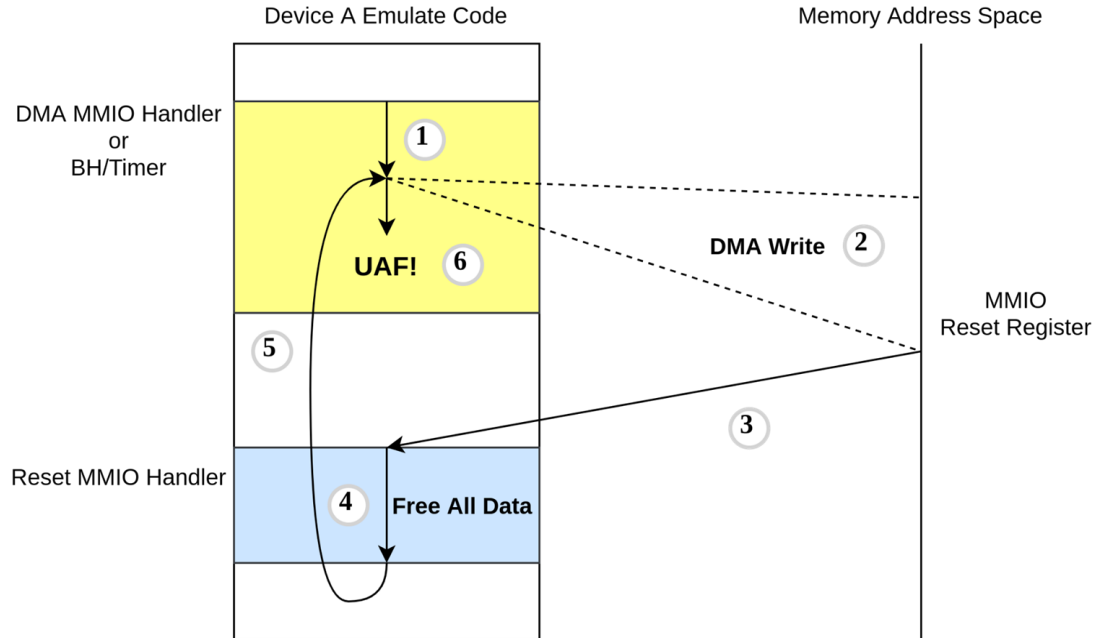
Device → Device

E.g. P2PDMA

A → B



A → A



# Recursive MMIO

MMIO: EPT Misconfig

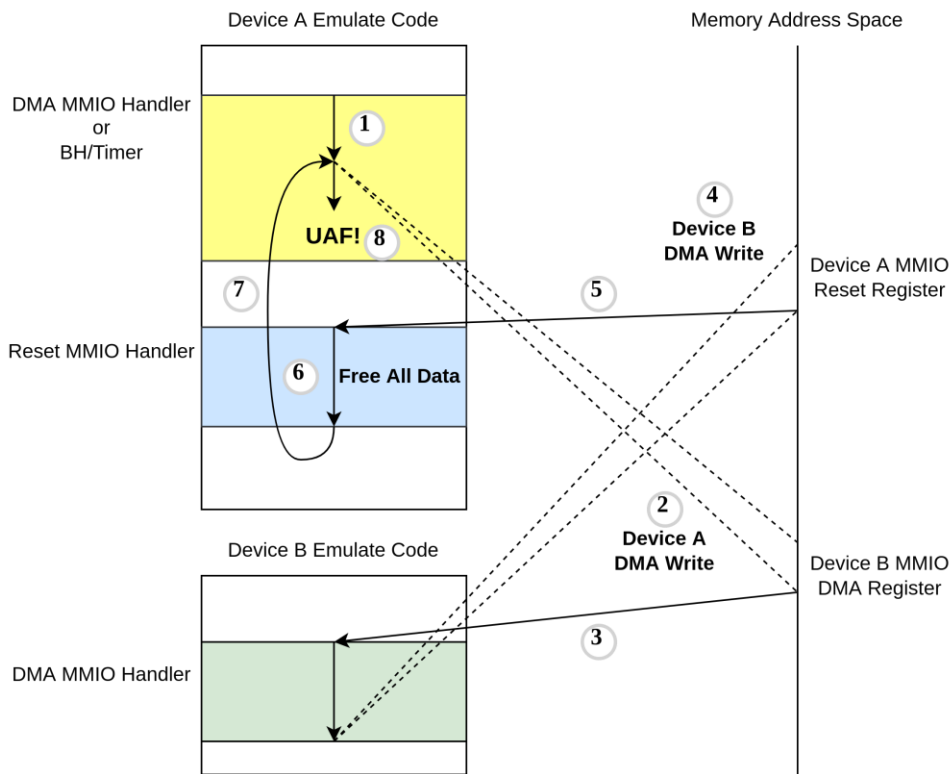
DMA: memcpy()

Device → Device

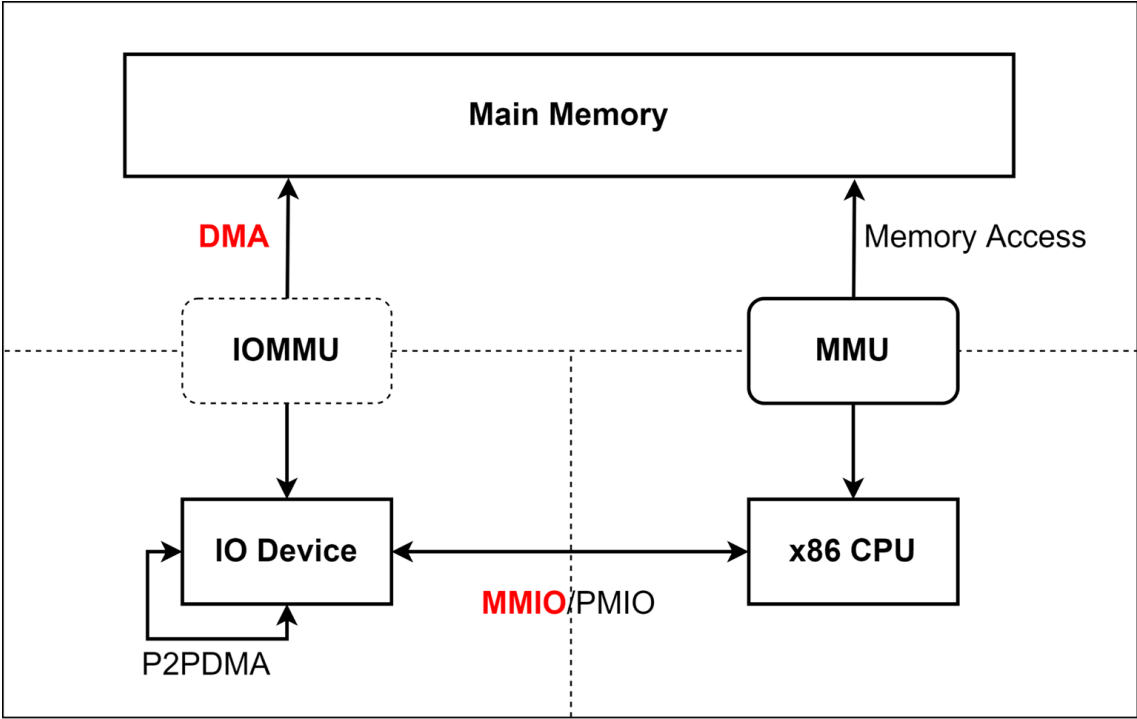
E.g. P2PDMA

A → B → C ✓

A → B → A ✗



# Recursive MMIO





# Common Consequences

Break device's state machine:

- 1. MMIO Reset Register → Free Data → Use Freed Data.**
2. Corrupt structures like transfer size and data index in global buffer.
3. Endless recursive call (stack overflow).

# Common Consequences

Break device's state machine:

1. MMIO Reset Register → Free Data → Use Freed Data.
- 2. Corrupt structures like transfer size and data index in global buffer.**
3. Endless recursive call (stack overflow).

# Common Consequences

Break device's state machine:

1. MMIO Reset Register → Free Data → Use Freed Data.
2. Corrupt structures like transfer size and data index in global buffer.
3. **Endless recursive call (stack overflow).**

# Things on VirtualBox

In VirtualBox, most write-to-physical-address primitives, like `PDMDevHlpPCIPhysWrite()`, `PDMDevHlpPhysWrite()`, `dmaR3WriteMemory()`, Etc. are end to call `PGMPPhysWrite()` -- 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 guess, we attached gdb to the VirtualBoxVM process and changed the `GCPhys` (physical address, controlled by the guest) to `0xf02000cc` (in MMIO region of the `pcnet` device), successfully triggered a recursive MMIO.

# Outline

1. Introduction
2. Root Causes
- 3. Hunting Flaws**
4. Case Study
5. Exploitation
6. Mitigations
7. Thoughts



# Recursive Paths

1. **MMIO Handler** → Write-to-Phys-Address API (DMA) → **MMIO Handler**
2. **BH/Timer Callback** → Write-to-Phys-Address API (DMA) → **MMIO Handler**

# Recursive Paths

## MMIO Write Handler

```
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);  
    /* ..... */  
};
```

# Recursive Paths

## MMIO Handler CodeQL

```
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()
    )
  }
}
```



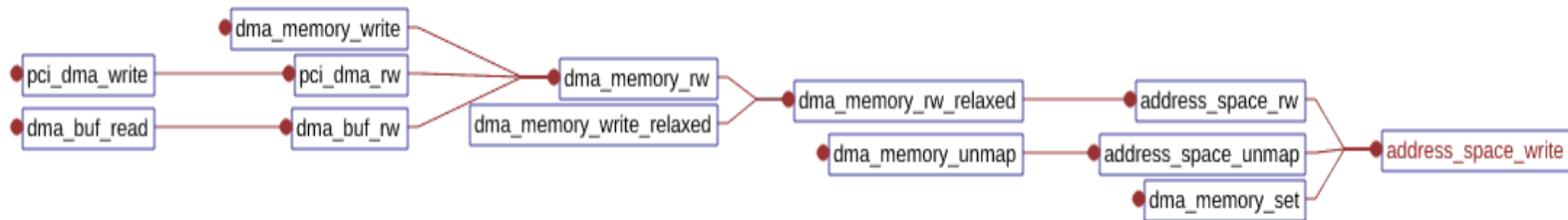
# Recursive Paths

## BH/Timer Callback CodeQL

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

# Recursive Paths

## Write-to-Phys-Address Gadgets Relationship



[docs/devel/loads-stores.rst](https://docs.kernel.org/devel/loads-stores.rst)

# Recursive Paths

## Write-to-Phys-Address Gadgets CodeQL

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

    .regexMatch("address_space_write|dma_memory_write|stb_dma|stl_be_dma|stl_le_dma|stq_be_dma|stq_le_dma|stw_be_dma|stw_le_dma|pci_dma_write|dma_buf_read|...")
  }
}
```

# Recursive Paths

## Nodes + Edges

```
/**  
 * @kind path-problem  
 */  
  
query predicate edges(Function a, Function b) { a.calls(b) }  
  
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()
```

# Recursive Paths

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

MMIO to pci\_dma\_write in MegaRAID SAS 8708EM2

# Free Primitives

MMIO write handler -> reset/free gadget

```
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.*")
    )
  }
}
```

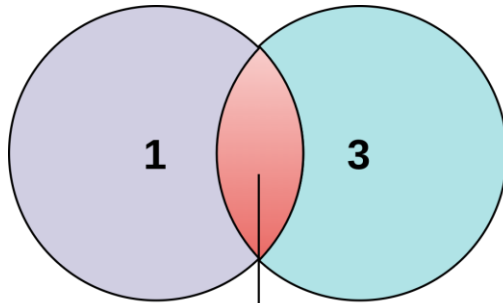
# Free Primitives

MMIO -> Free: from ehci_opreg_write to ehci_free_queue	hcd-ehci.c:604:13
Path	
1 ehci_opreg_write	hcd-ehci.c:1018:13
2 ehci_reset	hcd-ehci.c:847:6
3 ehci_queues_rip_all	hcd-ehci.c:677:13
4 ehci_free_queue	hcd-ehci.c:604:13

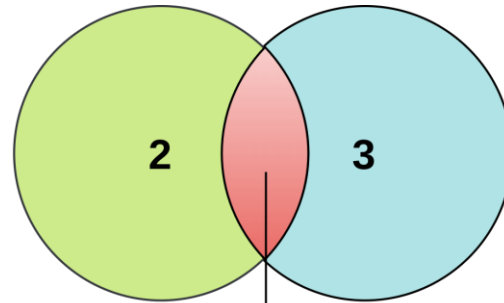
MMIO to Free in EHCI Controller

# UAF Bugs

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



**Vulnerable Devices:**  
MMIO -> Free -> UAF



**Vulnerable Devices:**  
BH/Timer -> Free -> UAF



# Possible Malloc Primitives

1. After the control flow returns from the MMIO reset handler, a malloc that allocates chunk the same size as freed chunk will be called.
2. We can write to another device's MMIO 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](#) is common in devices, it's easy to find vulnerable devices that do multiple write-to-physical-address actions.

# Malloc Pattern

1. There should be a `free()` before the `allocate` call, or the guest keep allocating chunks and crash the hypervisor.
2. The allocated chunk is composed of multiple smaller units (predefined structures, constant size), so we can control the whole chunk's size to occupy the freed chunk.
3. There should be a `read-from-physical-address` 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.

`g_free(buf)` → `buf = g_malloc(constant * nonconstant)` → `read_from_guest(buf)`

# Malloc Pattern

1. `g_free(buf)`
2. `buf = g_malloc(constant * nonconstant)`

```
Message
├── hw/audio/intel-hda.c intel-hda.c:473:15
│   └── Path
│       ├── 1 call to g_free intel-hda.c:472:5
│       ├── 2 ExprStmt intel-hda.c:473:5
│       ├── 3 sizeof(bpl) intel-hda.c:473:24
│       ├── 4 st intel-hda.c:473:38
│       ├── 5 bentries intel-hda.c:473:42
│       ├── 6 ... * ... intel-hda.c:473:24
│       └── 7 call to g_malloc intel-hda.c:473:15
└── hw/net/net_rx_pkt.c net_rx_pkt.c:81:20
```

Malloc Primitive in Intel HDA Device

# Malloc Primitive in Intel HDA

```
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); // There are 8 streams so we can malloc 1~8 chunks
    st->bpl = g_malloc(sizeof(bpl) * st->bentries); // 16n
    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);
    }
    /* ..... */
}
```

# Outline

1. Introduction
2. Root Causes
3. Hunting Flaws
4. **Case Study**
5. Exploitation
6. Mitigations
7. Thoughts



# NVMe: CVE-2021-3929

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

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

# NVMe: CVE-2021-3947

```
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);
}
```

# Outline

1. Introduction
2. Root Causes
3. Hunting Flaws
4. Case Study
- 5. Exploitation**
6. Mitigations
7. Thoughts





# Plan: Hijack the callback in cq->timer

1. In the guest OS, we **construct a fake timer and prepare a buffer for the MMIO write operations 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** in the VM.
3. Since CVE-2021-3929 and CVE-2021-3947 both are on the path of processing the Changed Namespace List command, and the content of the MMIO write operations in CVE-2021-3929 is from the buffer (nslist) in CVE-2021-3947, we can **make the source of MMIO write operations from a buffer in the guest's RAM (step 1).**
4. By leveraging CVE-2021-3929 and CVE-2021-3947, we trigger 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 of the main thread.**

## Plan (cont.)

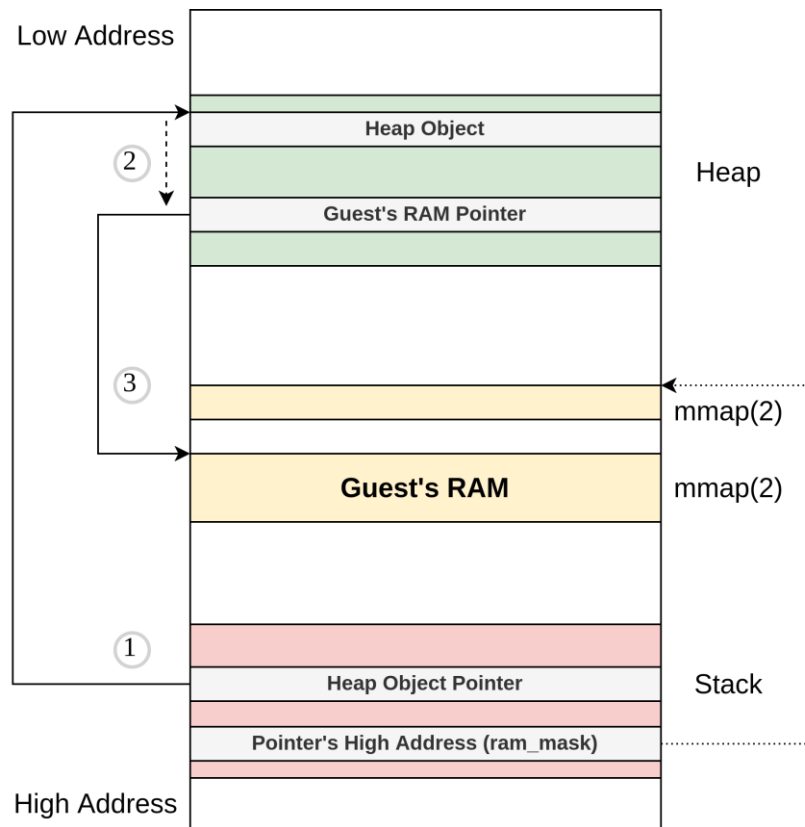
5. In the same DMA context, the second recursive MMIO write is to the reset register of the NVMe controller, **releasing the cq structure (put on the main thread's tcache)**.
6. In the same DMA context, the last MMIO write is to the malloc primitive in the HDA device, **thus occupying the timer pointer 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).

# Bypass ASLR: CVE-2021-3947

## Leak Guest's RAM

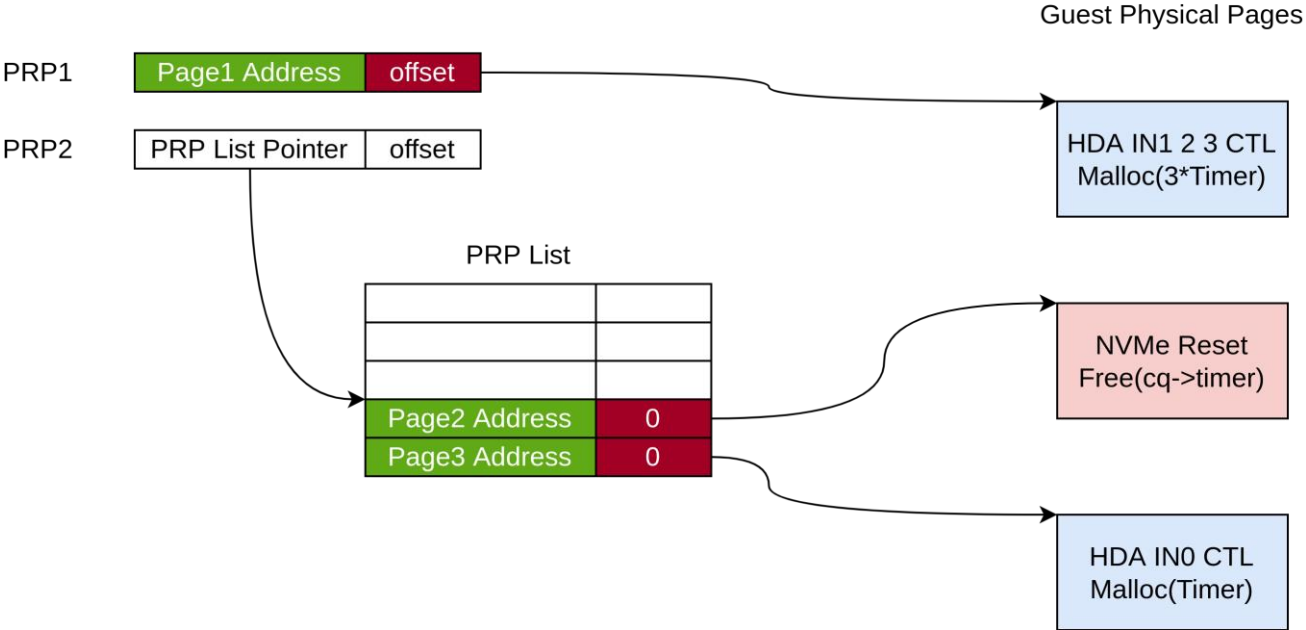
Search Patterns:

```
uint64_t tmp = *(heap + search_index);  
(tmp & 0x1ffffff) /* 0xe00000 */ == 0;  
(tmp & ~0x00007ffffffffffff) == 0;  
(tmp & ram_mask) == ram_mask;
```



# Hijack Control Flow: CVE-2021-3929

## PRP Structure



# Hijack Control Flow: CVE-2021-3929

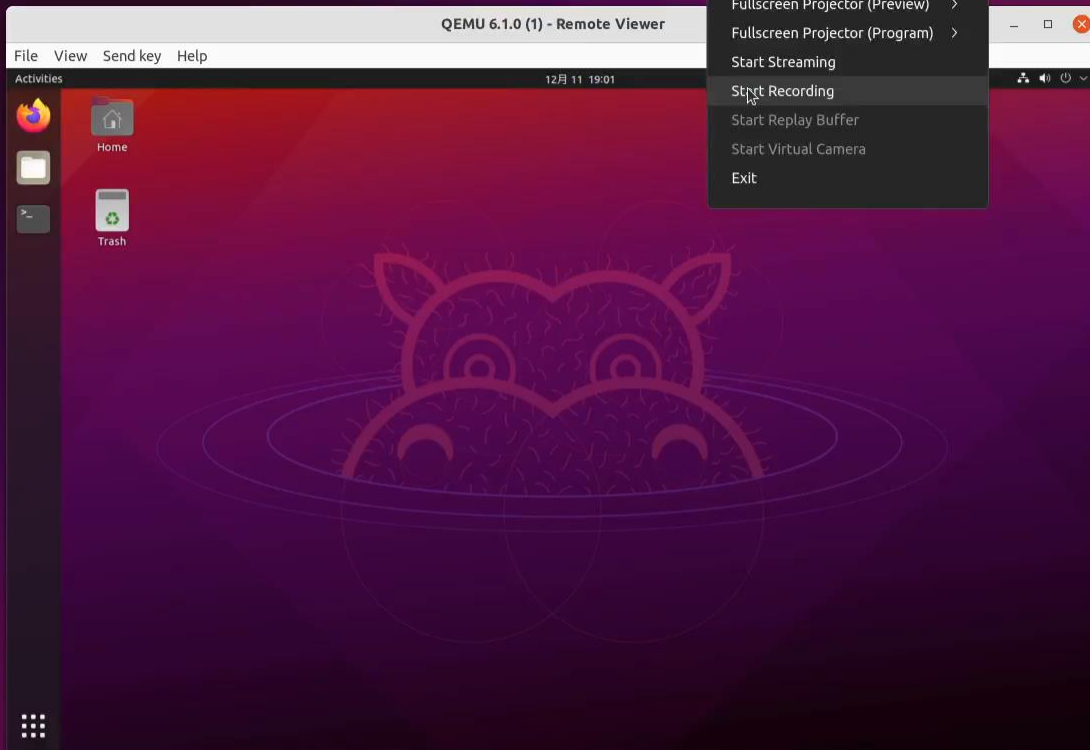
## Fake TimerList Callback

```
struct QEMUTimer {  
    int64_t expire_time;  
    QEMUTimerList *timer_list;  
    QEMUTimerCB *cb;  
    void *opaque;  
    QEMUTimer *next;  
    int attributes;  
    int scale;  
};
```

```
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();  
    }  
}
```



Home

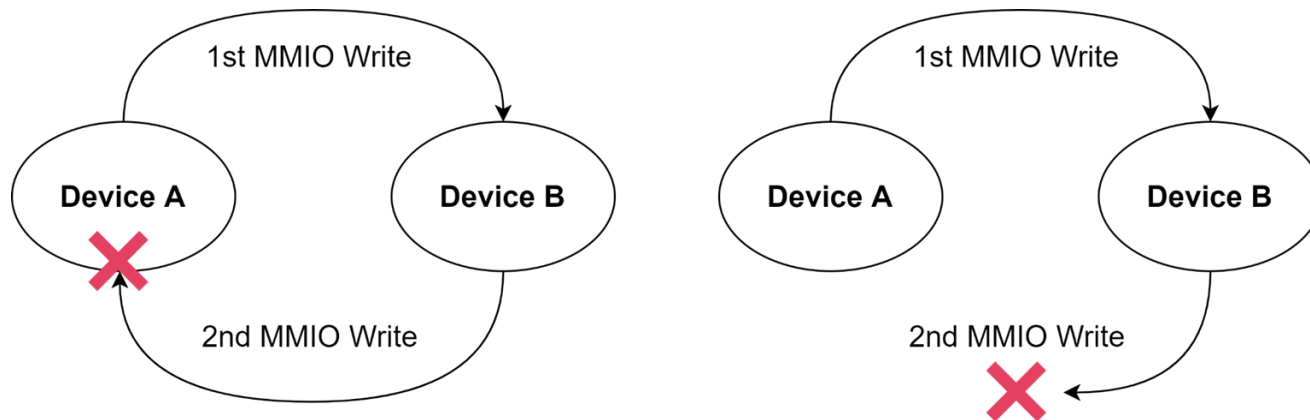


# Outline

1. Introduction
2. Root Causes
3. Hunting Flaws
4. Case Study
5. Exploitation
6. **Mitigations**
7. Thoughts



# Check in Device or Log on Bus



[\[RFC PATCH\] memory: Fix dma-reentrancy issues at the MMIO level](#)

[\[RFC PATCH v3 0/3\] physmem: Have flavier API check bus permission](#)



# Outline

1. Introduction
2. Root Causes
3. Hunting Flaws
4. Case Study
5. Exploitation
6. Mitigations
7. **Thoughts**



# Thoughts

1. This design flaw has a large impact and should be fixed ASAP. Any virtual **device that can perform DMA at a controlled address and has an MMIO region may be affected.**
2. The device reentry / recursive MMIO defect has existed in QEMU for at least ten years, or more than one year just considering the large-scale appearance. For these bugs, 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 :)**
3. When auditing the hypervisors, we should **pay attention to the different behaviors between virtualization software and real hardware.**



# Thank You

We will release the **white paper** and **PoC** on May 20 at:

<https://github.com/QiuhaoLi/CVE-2021-3929-3947>