Resurrecting Zombies
Leveraging advanced techniques of DMA reentrancy to escape QEMU

Ao Wang | Security Research Expert | DBAPPSecurity WeBin Lab
Ao Wang (@arayz)

- Security research at DBAPPSecurity WeBin Lab
- Hunting and exploiting vulnerabilities in critical products
- Mobile/Browser/Virtualization
- Pwned Safari for multiple times with callback related vulnerabilities
- Mainly focus on QEMU-KVM currently
Agenda

- Introduce
- Challenges
- DMA Oriented Programing
- Exploitation
- DEMO Time
- Conclusion
Agenda

- Introduce
- Challenges
- DMA Oriented Programming
- Exploitation
- DEMO Time
- Conclusion
Related Work

- BlackHat Asia 2022, *Hunting and Exploiting Recursive MMIO Flaws in QEMU/KVM*
  - Root Cause
  - Hunting And Exploitation
  - Mitigation

- QEMU Community, **Fix DMA MMIO reentrancy issues**
  - Fundamentally solve DMA Reentrancy problem
  - Known vulnerabilities
  - Mostly found by fuzzing
DMA Reentrancy Issue

- Make destination of DMA operation overlaps with MMIO region of the peripherals modules to invoke function call access to MMIO handlers
- Caused by difference of hypervisor and real hardware
- No defenses in the code of QEMU except for fixed vulnerabilities
- Hard to fix, still got some known vulnerabilities in latest version, and there are still some hidden vulnerabilities
- 2 types of patches
- Besides QEMU, some other hypervisors may also be affected (VirtualBox)
- Most will crash with infinite reentrancy, there are prequesites for exploiting
Agenda

- Introduce
- **Challenges**
- DMA Oriented Programming
- Exploitation
- DEMO Time
- Conclusion
Prerequisite - Case 1

```c
static Vulnerable Function()
{
    if (Ref(obj) == NULL) {
        return;
    }
    if (Recursion Condition) {
        DMA_Write();
    }
    Free(obj);
    Clear_Ref(obj);
};
```
Prerequisite - Case 1

Construct Primitives

- Make destination of DMA operation overlaps with MMIO region of the device
- Re-enter the vulnerable function to free the object twice
- Occupy the freed chunk to prevent crash after exiting the re-entrancy

Now we got an object that has been already freed
Prerequisite – Case 1

```c
bool prepare_mmio_access(MemoryRegion *mr) {
    bool release_lock = false;

    if (!qemu_mutex_iotrace_locked()) {
        qemu_mutex_lock_iotrace();
        release_lock = true;
    }

    if (mr->flush_coalesced_mmio) {
        qemu_flush_coalesced_mmio_buffer();
    }

    return release_lock;
}
```

- I/O thread is locked until exiting MMIO handler
- With glibc 2.31+, each thread corresponds to an independent arena and tcache

We can’t occupy the freed chunk with another thread or I/O request, we must do this in the same DMA context which triggers the vulnerability
Prerequisites – Case 1

Exiting I/O context safely
- Occupy the object to prevent crash
- Change the recursion condition to prevent infinite reentrancy

Occupy the freed chunk stably
- Clear the tcache before re-enter the vulnerable function

Re-enter the vulnerable function to free the object

Some other necessary context settings also require DMA Write Operations
Prerequisites – Case 1

Vulnerable Function
NULL Pointer Check
Recursion Condition
DMA Write
Free
Clear Pointer

We need more than 10 DMA write operations before exiting the MMIO context

However, we usually only have one or two chances of DMA writing
Prerequisite One: 
Need Scatter-Gathered DMA Operations
Prerequisite - Case 2

```c
static Vulnerable_Function(int* data) {
    int* obj = GetFromContext();
    DMA_Write(data);
    Use(obj);
};
```

To trigger the UAF, we must leverage the DMA Write Operation to send a specific value to the specific handler of MMIO region and free the object from the context

```c
static Free_The_Object (int val) {
    if (val & FREE_CONDITION_BIT) {
        Free(obj);
    }
}
```

The content of DMA write operation originally sends to the guest can’t be controlled by the guest
Prerequisites - Case 2

DMA Write Operation needs 3 parameters to go:

- `addr`: Destination Address
- `buf`: Contents To Write
- `len`: Length Of Contents

To trigger the vulnerability or to exploit it, we want to control them all:

- Control the `addr` for controlling which handler it sends to
- Control the `len` for getting into the correct handler
- Control the `buf` for getting into the correct branch

```c
/**
 * pci_dma_write: Write to address space from PCI device.
 * *
 * Return a MemTxResult indicating whether the operation succeeded
 * or failed (e.g. unassigned memory, device rejected the transaction,
 * IOMMU fault).
 * *
 * @dev: #PCIDevice doing the memory access
 * @addr: address within the #PCIDevice address space
 * @buf: buffer with the data transferred
 * @len: the number of bytes to write
 */
static inline MemTxResult pci_dma_write(PCIDevice *dev, dma_addr_t addr, const void *buf, dma_addr_t len)
{
    return pci_dma_rw(dev, addr, (void *) buf, len,
            DMA_DIRECTION_FROM_DEVICE, MEMTXATTRS_UNSPECIFIED);
}
```
Prerequisite Two:
Gain control of the three parameters of the DMA write operation
Prerequisites – Case 3

static Addr_overlaps_mmio(void* addr)
{
    MemoryRegion *mr = GetDeviceMMIORegion();
    return belongToMR(addr);
}

DMA operation for local access to the region of MMIO memory is guarded. Can't DMA access handlers of a device from the device itself.
Prerequisite Three:
The destination where the DMA operation is located can be reached
Agenda

- Introduce
- Challenges
- DMA Oriented Programming
- Exploitation
- DEMO Time
- Conclusion
I/O Request Handler Model

I/O Request

Device

addr condition

data condition

size condition

Switch

data condition

Handler 1
DMA Write Data1
Handler 2
DMA Write Data2
Handler 3
DMA Write Data3
I/O Request Handler Model

I/O Request -> Switch

Device

Handler 1
DMA Write Data1
Handler 2
DMA Write Data2
Handler 3
DMA Write Data3

I/O Response
DMA Reflection

Device

Handler A

DMA Write Data

...... 0x42 0x80 0x23 ......

Handler B

Guard

Reset

val & 1 << 5 ?
DMA Reflection

Device A

DMA Write Data

...... 0x42 0x80 0x23 ......

Device B

val & 1 << 9 ?

Reset
DMA Reflection

**Source Device**

**DMA Write Data**

```
...... 0x42 0x80 0x23 ......
```

**Target Device**

**Reset**

```
val & 1 << 4 ?
```

**Springboard 1**

**DMA Write Data**

```
...... 0x33 0x08 0xA2 ......
```

**Springboard 2**

**DMA Write Data**

```
...... 0x33 0x08 0xA2 ......
```

```
val & 1 << 3 ?
```
Network Loopback Mode

- Totally controllable content
- Synchronization Processing
Network Loopback Mode

```c
static const MemoryRegionOps rtl8139_io_ops = {
    .read = rtl8139_ioport_read,
    .write = rtl8139_ioport_write,
    .impl = {
        .min_access_size = 1,
        .max_access_size = 4,
    },
    .endianness = DEVICE_LITTLE_ENDIAN,
};

static void rtl8139_io_writeb(void *opaque, uint8_t addr, uint32_t val)
{
    switch (addr)
    {
    ......
    case TxPoll:
        if (val & (1 << 6))
        {
            rtl8139_cplus_transmit(s);
        }
        break;
    ......
    }
```
Network Loopback Mode

static void rtl8139_cplus_transmit(RTL8139State *s)
{
    int txcount = 0;

    while (txcount < 64 && rtl8139_cplus_transmit_one(s))
    {
        ++txcount;
    }
...

static int rtl8139_cplus_transmit_one(RTL8139State *s)
{
    int descriptor = s->currCPlusTxDesc;
    dma_addr_t cplus_tx_ring_desc = rtl8139_addr64(s->TxAddr[0], s->TxAddr[1]);
    cplus_tx_ring_desc += 16 * descriptor;
    ......
    uint32_t val, txdw0, txdw1, txbufLO, txbufHI;
    pci_dma_read(d, cplus_tx_ring_desc, (uint8_t *)&val, 4);
    txdw0 = le32_to_cpu(val);
    pci_dma_read(d, cplus_tx_ring_desc+4, (uint8_t *)&val, 4);
    txdw1 = le32_to_cpu(val);
    pci_dma_read(d, cplus_tx_ring_desc+8, (uint8_t *)&val, 4);
    txbufLO = le32_to_cpu(val);
    pci_dma_read(d, cplus_tx_ring_desc+12, (uint8_t *)&val, 4);
    txbufHI = le32_to_cpu(val);
    ......
static ssize_t qemu_net_queue_deliver(NetQueue *queue, 
   NetClientState *sender, 
   unsigned flags, 
   const uint8_t *data, 
   size_t size)
{
    ssize_t ret = -1; 
    struct iovec iov = {
        .iov_base = (void *)data, 
        .iov_len = size 
    };

    queue->delivering = 1;
    ret = queue->deliver(sender, flags, &iov, 1, queue->opaque);
    queue->delivering = 0;

    return ret;
}

We can’t re-deliver the packet while the queue is delivering
But it can still be used to DMA Write to another device
Network Loopback Mode

```c
static ssize_t rtl8139_do_receive(NetClientState *nc, const uint8_t *buf, size_t size_, int do_interrupt)
{
    ....
    dma_addr_t rx_addr = rtl8139_addr64(rxbufLO, rxbufHI);
    if (dot1q_buf) {
        pci_dma_write(d, rx_addr, buf, 2 * ETH_ALEN);
        pci_dma_write(d, rx_addr + 2 * ETH_ALEN, 
                       buf + 2 * ETH_ALEN + VLAN_HLEN, 
                       size - 2 * ETH_ALEN);
    } else {
        //Highlighted code
        pci_dma_write(d, rx_addr, buf, size);
    }
    ....
}
```

Totally Controllable Scatter-Gathered DMA Write Operation
DMA Refraction

Network Loopback Handler

val & 1 << 6 ?

Scatter-Gathered DMA

DMA Write Operation

Target Handler 1

Target Handler 2

Target Handler N

Source Handler

DMA Write Operation
DMA Oriented Programming

- Handler 1: DMA Write
- Handler 2: DMA Write
- Handler 3: DMA Write
- Handler 4: DMA Write
- Handler 5: DMA Write
- Handler 6: DMA Write
- Handler 7: Scatter-Gathered DMA Write

Target Handler N:
- Context Setting

Target Handler N+1:
- Reset

Target Handler N+2:
- Occupy
DMA Oriented Programming

- Base on the data which the DMA Write Operation provides, find a path and leverage DMA Reflection to connect it into the `Scatter-Gathered DMA Operation Network` to regain control.

- We can build the entire DMA network for constructing DMAOP-Chain conveniently.

- Leveraging DMA Refraction to transform the DMA Write Operation into nearly a callback function, each DMA Write Operation may be a potential chance for attackers to regain control without exiting the I/O context.

- In addition to break through the aforementioned prerequisites, DMA-OP can be used to construct some novel exploit techniques.
Agenda

- Introduce
- Challenges
- DMA Oriented Programing
- **Exploitation**
- DEMO Time
- Conclusion
Primitive

Vulnerable Function
NULL Pointer Check
Recursion Condition
DMA Write
Free
Clear Pointer

Clear Tcache
Disable Recursion
Re-entrancy
Occupy The Chunks

Object A
Object B
Primitive

- Vulnerable Function
- NULL Pointer Check
- Recursion Condition
- DMA Write
- Free
- Clear Pointer

Refraction

- Clear Tcache
- Disable Recursion
- Re-entrancy
- Occupy The Chunks

Object A

Object B
Uaf-After-uaF

- Now we got 2 objects which were already freed while we still hold the pointers, and we could free them again
- 144-byte chunk\(X\) and 64-byte chunk\(\text{Small } X\)
- To leak information from the host, occupy \(X\) with an object which could write its content to the guests
- To hijack the control flow, occupy \(\text{Small } X\) with another timer, overwrite the callback pointer with function address of `system`
Stability Optimization

Attacker → Alloc X

Driver → Alloc Small X

Attacker → Alloc Small X

X

X

Small X
Stability Optimization

Attacker → Refraction → X
- Alloc X
- Alloc Small X

Driver → Alloc Small X → X
- X
- Small X
Info Leak

- To leak the function address of `system`, leak the base address of libc first
- To use Unsorted-Bin-Leak trick to leak base address of libc, free an object and throw it into the unsorted bin from main-arena
- To hijack the control flow properly, the `timer_list` pointer must be leaked
- To place arguments of `system` function, leak an address of a controllable buffer
static uint16_t nvme_zone_mgmt_recv(NvmeCtrl *n, Nvmerequest *req)
{
    if (data_size < sizeof(NvmeZoneReportHeader)) {
        return NVME_INVALID_FIELD | NVME_DNR;
    }
    
    buf = g_malloc0(data_size);
    zone = &ns->zone_array[zone_idx];
    for (i = zone_idx; i < ns->num_zones; i++) {
        if (partial && nr_zones >= max_zones) {
            break;
        }
        if (nvme_zone_matches_filter(zrasf, zone++)) {
            nr_zones++;
        }
    }
    header = buf;
    header->nr_zones = cpu_to_le64(nr_zones);

    z->zt = zone->d.zt;

    status = nvme_c2h(n, (uint8_t *)buf, data_size, req);
    g_free(buf);
    return status;
}
Info Leak

static MemTxResult dma_buf_rw(void *buf, dma_addr_t len, dma_addr_t *residual,
QEMUSGList *sg, DMADirection dir,
MemTxAttrs attrs)
{

... 

while (len > 0) {
    ScatterGatherEntry entry = sg->sg[sg_cur_index++];
    dma_addr_t xfer;  
    But we only have value 0x01 and 0x02 :(  
    res |= dma_memory_rw(sg->as, entry.base, ptr, xfer, dir, attrs);
    ptr += xfer;  
    We must reflect the value to the netcard  
    len -= xfer;  
    to construct DMA Refraction here
    xresidual -= xfer;
}
... 
}
static void xhci_doorbell_write(void *ptr, hwaddr reg, uint64_t val, unsigned size)
{
    reg >>= 2;
    if (reg == 0) {
        ......
    } else {
        epid = val & 0xff;
        streamid = (val >> 16) & 0xffff;
        if (reg > xhci->numslots) {
            DPRINTF("xhci: bad doorbell %d\n", (int)reg);
        } else if (epid == 0 || epid > 31) {
            DPRINTF("xhci: bad doorbell %d write: 0x%x\n", 
            (int)reg, (int32_t)val);
        } else {
            xhci_kick_ep(xhci, reg, epid, streamid);
        }
    }
}

static void xhci_kick_epctx(XHCIEPContext *epctx, unsigned int streamid)
{
    if (epctx->nr_pstreams) {
        ......
        xhci_set_ep_state(xhci, epctx, stctx,
        EP_RUNNING);
    } else {
        ring = &epctx->ring;
        streamid = 0;
        xhci_set_ep_state(xhci, epctx, NULL, 
        EP_RUNNING);
    }
}

DMA jump to RTL8139 (Refraction Chance)
struct QEMUTimer {
    int64_t expire_time;
    QEMUTimerList **timer_list;
    QEMUTimerCB *cb;
    void *opaque;
    QEMUTimer *next;
    int attributes;
    int scale;
};

This must be leaked
Overwrite this to hijack control flow
This points to the context of the timer
Overwrite it to a controllable buffer
Info Leak

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

static void hda_audio_input_timer(void *opaque) {
    HDAAudioStream *st = opaque;
}
```
Info Leak

- We can only alloc `buf` to occupy X since it must be above 64 bytes, it can’t be Small X

- Since the `nvme_zone_mgmnt_recv` use `g_malloc0` to alloc `buf`, and it must be called in the timer thread, must be located in the main-arena

- To leak `timer_list` and the controllable buffer, slice the X with timers in the unsorted-bin

- To avoid X to be merged when we throw it into the unsorted-bin, we need to place X in a hole
Place X In A Hole

epctx 1
epctx 2
epctx 3
epctx 4
epctx 5
epctx 6
Place X In A Hole

<table>
<thead>
<tr>
<th>epctx 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>epctx 2</td>
</tr>
<tr>
<td>epctx 3</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>epctx 5</td>
</tr>
<tr>
<td>epctx 6</td>
</tr>
</tbody>
</table>
Shuttle Between Threads

I/O Thread
  Code
  Alloc
  Non-Main-Arena

Main Thread
  Code
  DMA Jump
  Main-Arena
Shuttle Between Threads

I/O Thread
- Code
- Free
- Alloc
- X
- Tcache 2

Main Thread
- Code
- Free
- X
- Tcache 1
Shuttle Between Threads

I/O Thread
  Code
  Free  Alloc
  X
  Tcache 2 (Full)

Main Thread
  Code
  Free
  Main Arena
  Tcache 1
Shuttle Between Threads And Devices

- **Guest**
  - timer_list
  - controllable buffer
  - libc base address

- **I/O Thread**

- **Main Thread**
  - NVME Alloc
  - Intel HDA Free, Spray

- **Main Arena**
  - XHCI Fill
  - buf
  - Main Arena
  - Tcache (Full)
Shuttle Between Threads And Devices

- Guest
  - timer_list
  - controllable buffer
  - libc base address

- I/O Thread
  - X

- Main Thread
  - NVME
    - Alloc
  - Intel HDA
    - Free
    - Spray
  - XHCI
  - Fill
  - buf
  - Main Arena
  - Tcache
DMA-OP Chain

NVME
buf(X)
0x01 0x02

XHCI
0x80

RTL8139
Refraction

Intel HDA
Free  Occupy

Guest
libc base address

controllable buffer
timer_list

0x1 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
0x2 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
......
Hijack Control Flow

- Re-allocate a `epctx->kick_timer` on Small X
- Overwrite the `cb` function pointer to the function address of `system`
- Fix the `timer_list` with the leaked real `timer_list`
- Fix the `opaque` with the leaked controllable buffer address
- Fill the leaked controllable buffer with the command line we want `system` function to execute
- Kick the timer in the XHCI controller to escape from QEMU
Agenda

- Introduce
- Challenges
- DMA Oriented Programing
- Exploitation
- DEMO Time
- Conclusion
DEMO Time
Agenda

- Introduce
- Challenges
- DMA Oriented Programing
- Exploitation
- DEMO Time
- Conclusion
Conclusion

- Use Refraction to gather multiple I/O requests in one I/O context to avoid interference from system driver's I/O requests
- Change the thinking, regard DMA operations in the code as a callback function that can regain control, make the exploitation flexible, and audit TOCTOU related issues
- The community is preparing a patch to fix almost every DMA Reentrancy issue, but DMA Oriented Programing will not be affected
- To defense DMA-OP effectively, permission need to be added for DMA operations, this requires extensive auditing
- Creating a graph of `Scatter-Gathered DMA Operation Network`, which can effectively help construct a DMA-OP chain
- DMA-OP in other hypervisors need to be audited such as VMware, VirtualBox
Thank you!