#HITB2023AMS https://conference.hitb.org/

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 mutiple times with callback related vulnerabilites
- Mainly focus on QEMU-KVM currently

Agenda

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

Agenda

- **Indroduce <<**
- Challenges
- DMA Oriented Programing
- 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

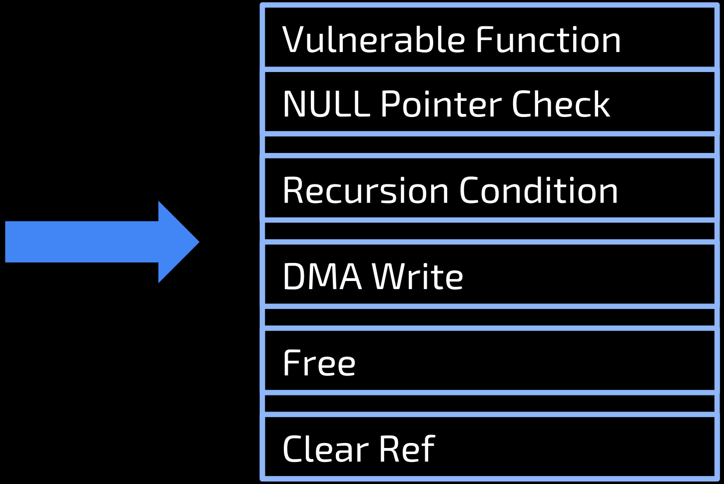
- Make destination of DMA operation overlaps with MMIO region of the peripherals modules to invoke function call access to MMIO handlers
- Caused by difference of hypervisor and real hardware
- No defenses in the code of QEMU except for fixed vulnerabilities
- Hard to fix, still got some known vulnerabilities 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

- Indroduce
- **Challenges** <<
- DMA Oriented Programing
- Exploitation
- DEMO Time
- Conclusion

Prerequesite – Case 1

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



Vulnerable Function

NULL Pointer Check

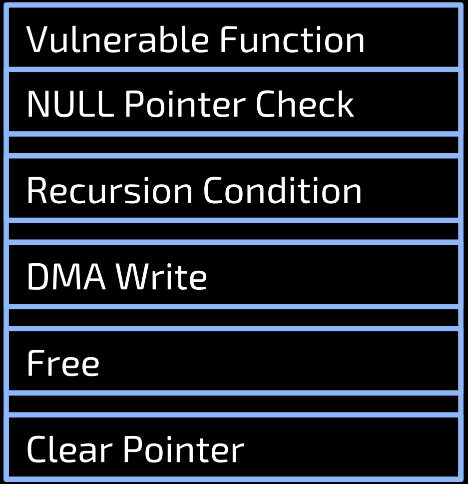
Recursion Condition

DMA Write

Free

Clear Ref

Prerequesite - Case 1



Vulnerable Function

NULL Pointer Check

Recursion Condition

DMA Write

Free

Clear Pointer

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

Prerequesite - Case 1

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

    if (!qemu_mutex_iothread_locked()) {
        qemu_mutex_lock_iothread();
        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

Prerequesite – Case 1

Vulnerable Function

NULL Pointer Check

Recursion Condition

DMA Write

Free

Clear Pointer

Exiting I/O context safely

- Occupy the obejct to prevent crash
- Change the recursion condition to prevent infinite reentancy

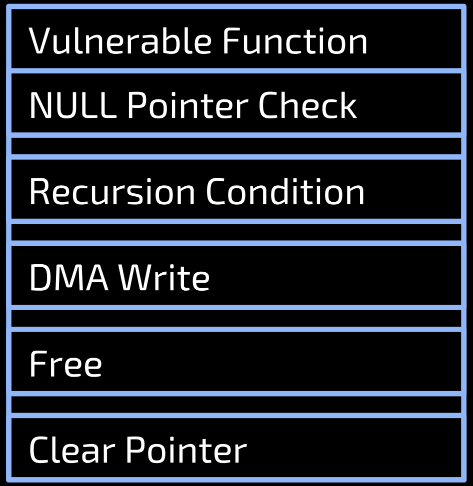
Occupy the freed chunk stably

- 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

Prerequesite – 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

Prerequesite One:
Need Scatter-Gathered DMA Operations

Prerequesite - Case 2

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

```
static Free_The_Object (int val)
{
   if (val & FREE_CONDITION_BIT) {
      Free(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

The content of DMA write operation originally sends to the guest can't be controlled by the guest

Prerequesite - Case 2

```
/**
 * pci_dma_write: Write to address space from PCI device.
 *
 * Return a MemTxResult indicating whether the operation succeeded
 * or failed (eg 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);
}
```

DMA Write Operation needs 3 paramters 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

Prerequesite Two:

Gain control of the three parameters of the DMA write operation

Prerequesite – 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 gurarded. Can't DMA access handlers of a device from the device itself.

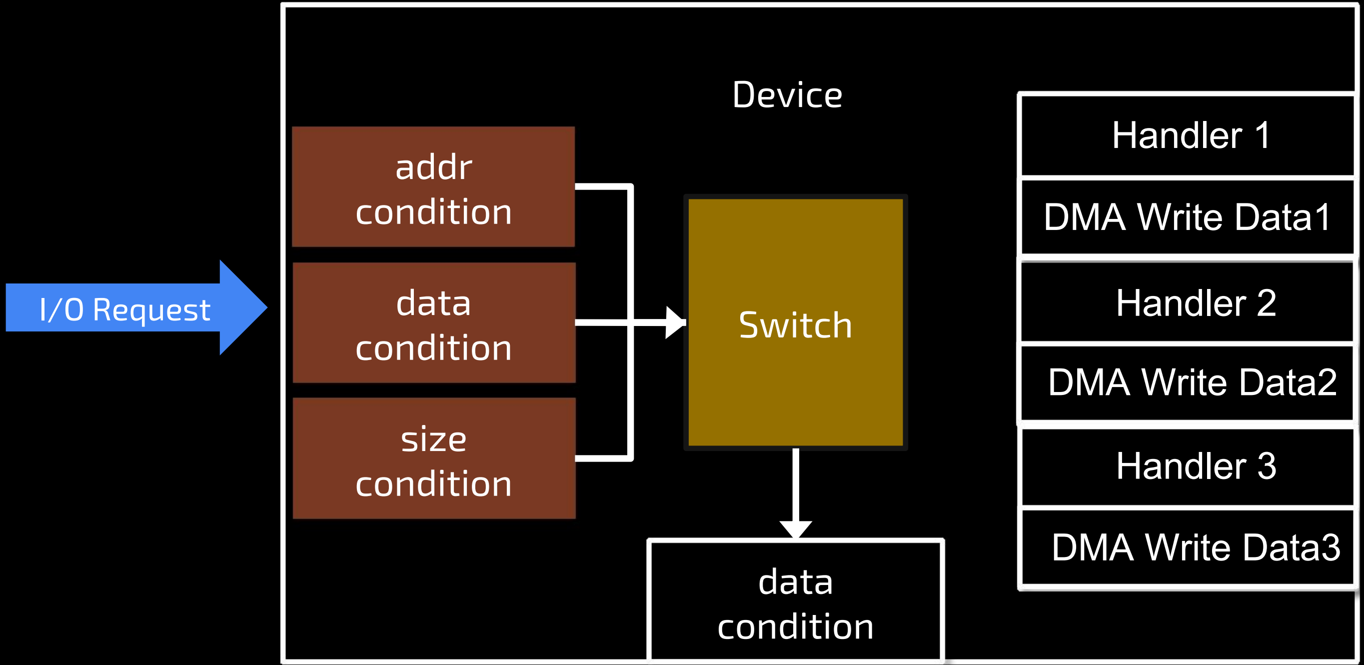
Prerequesite Three:

The destination where the DMA operation is located can be reached

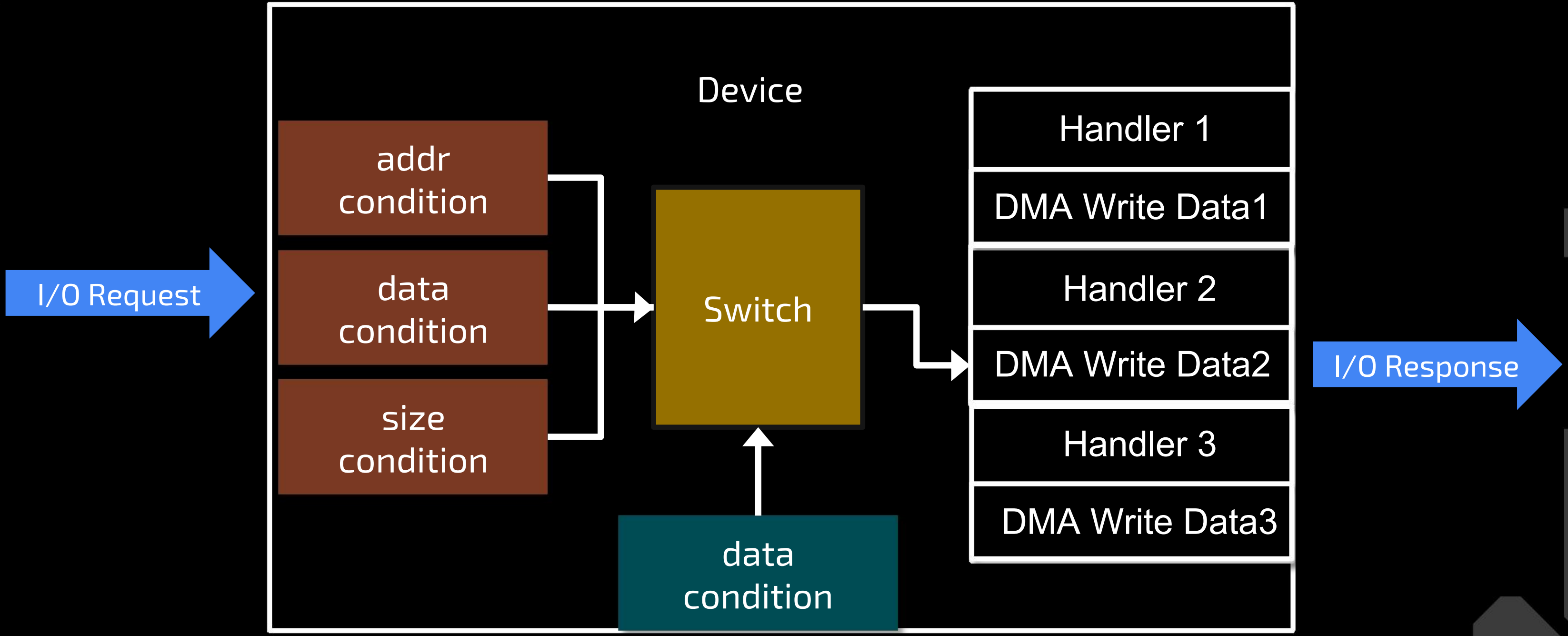
Agenda

- Indroduce
- Challenges
- **DMA Oriented Programing** **<<**
- Exploitation
- DEMO Time
- Conclusion

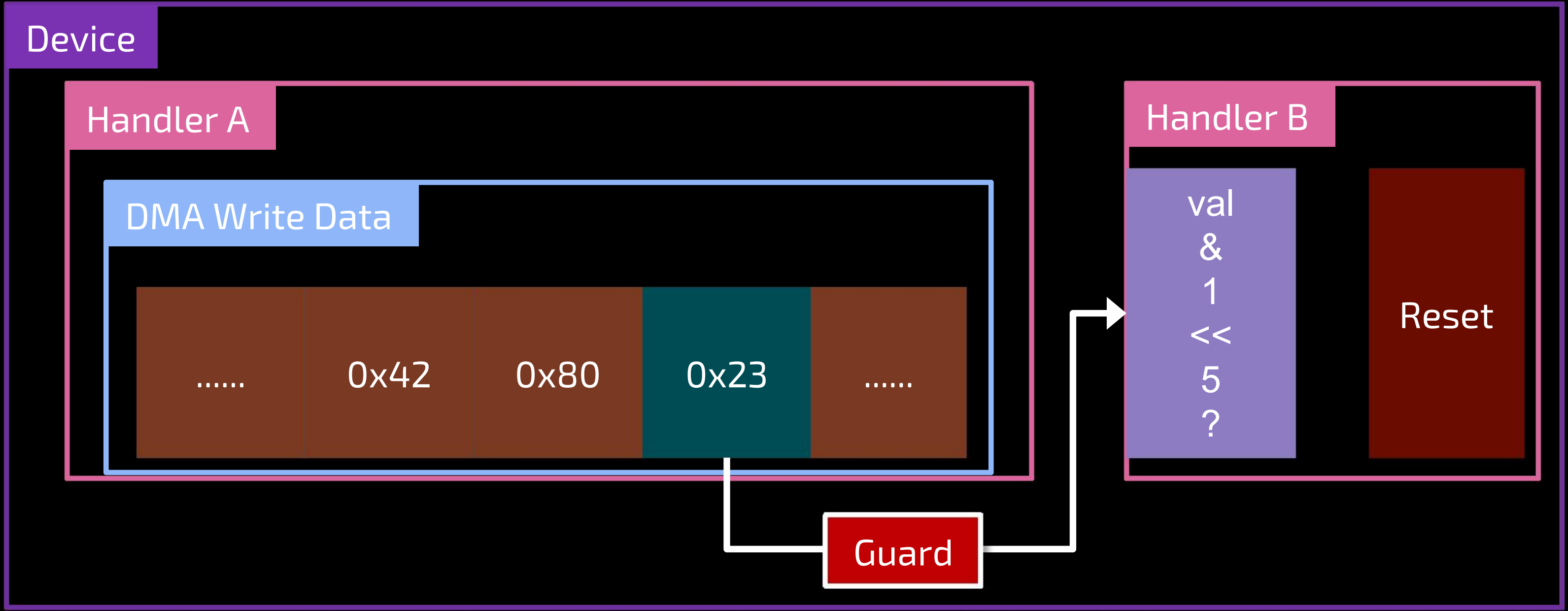
I/O Request Handler Model



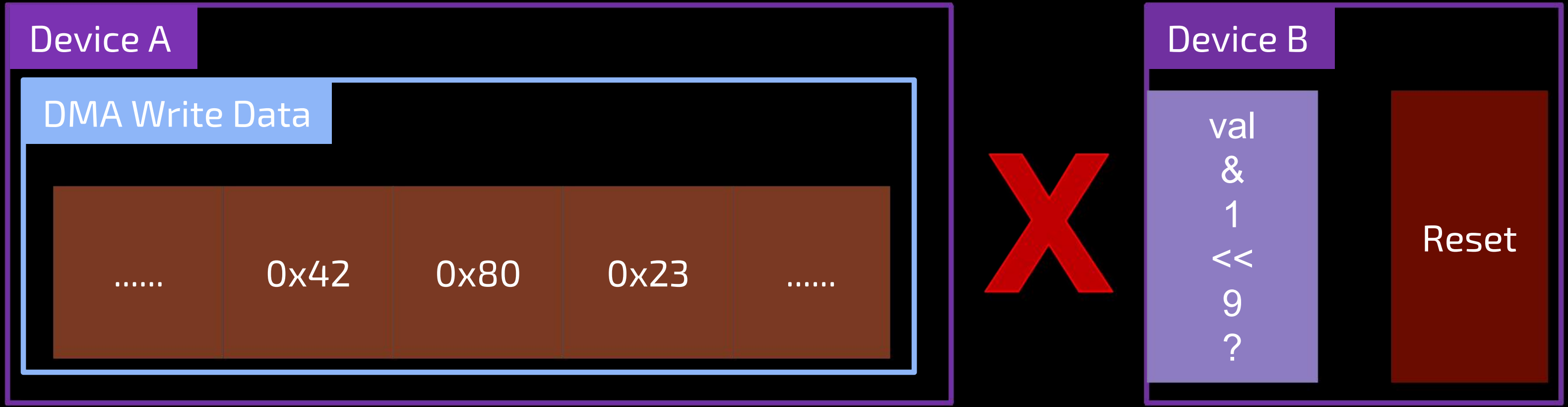
I/O Request Handler Model



DMA Reflection



DMA Reflection



Device A

DMA Write Data

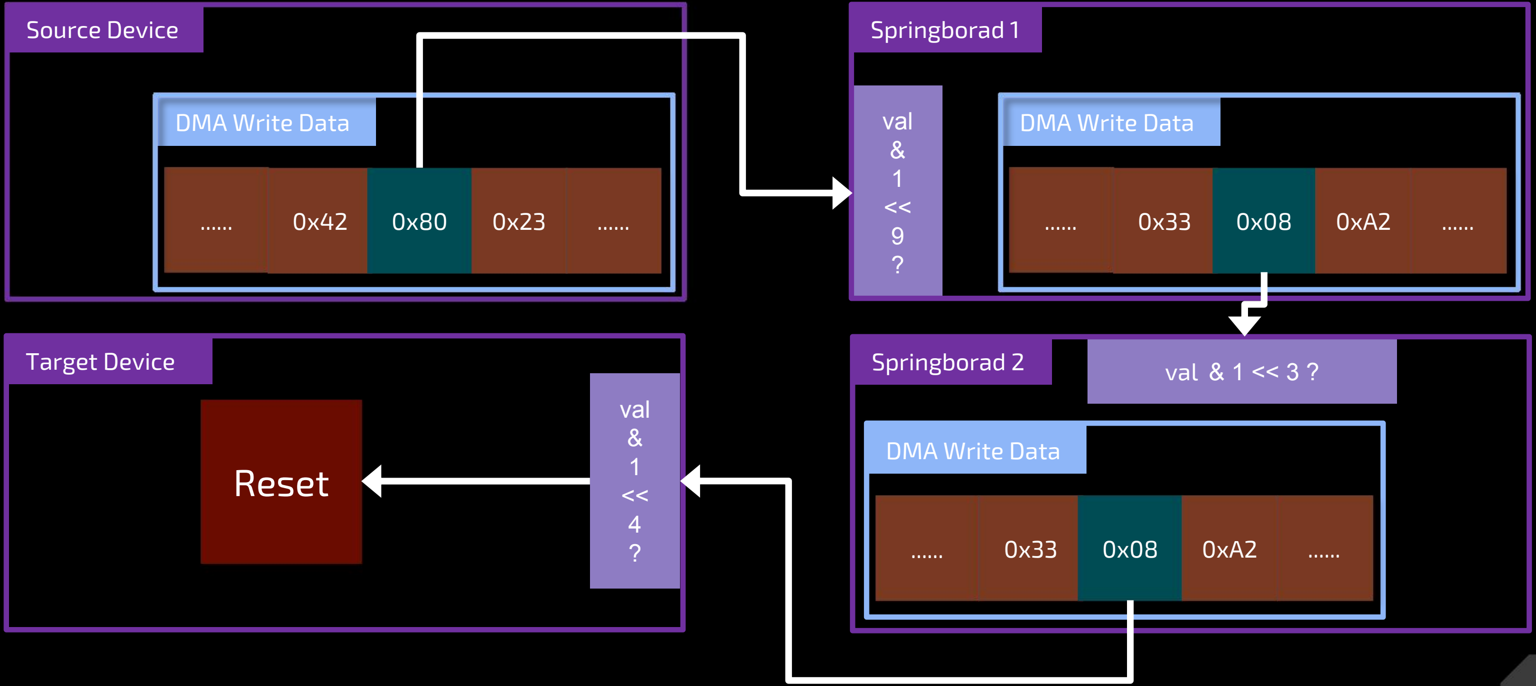
……	0x42	0x80	0x23	……

Device B

val
&
1
<<
9
?

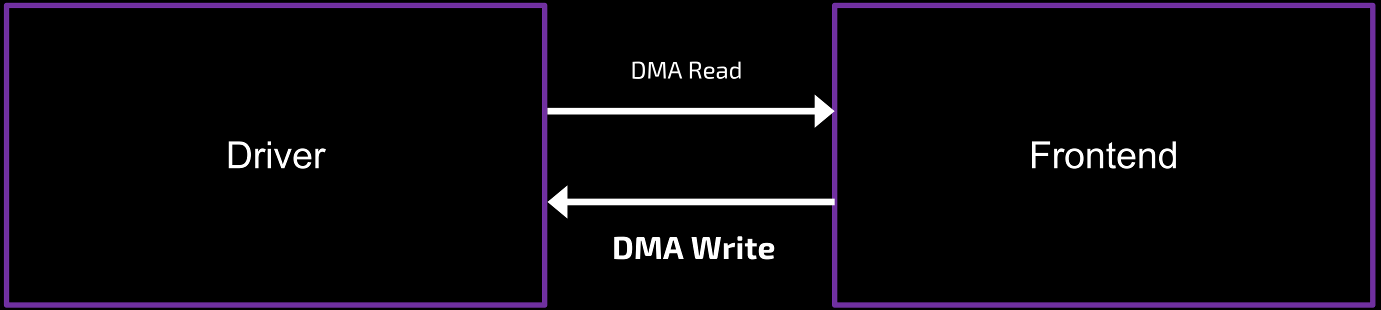
Reset

DMA Reflection



Network Loopback Mode

- Totally controllable content
- Synchronization Processing



Network Loopback Mode

```
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,
};
```

`size` Condition

```
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;
            ......
}
```

`data` Condition

Network Loopback Mode

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

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

Up to 64 times of
DMA Write Operation

```
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);
   ......
```

Subsequent code flow
relies on the preset
context, which is no
longer related to the
parameters provided by
the DMA operation

Resurrecting Zombies

```
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

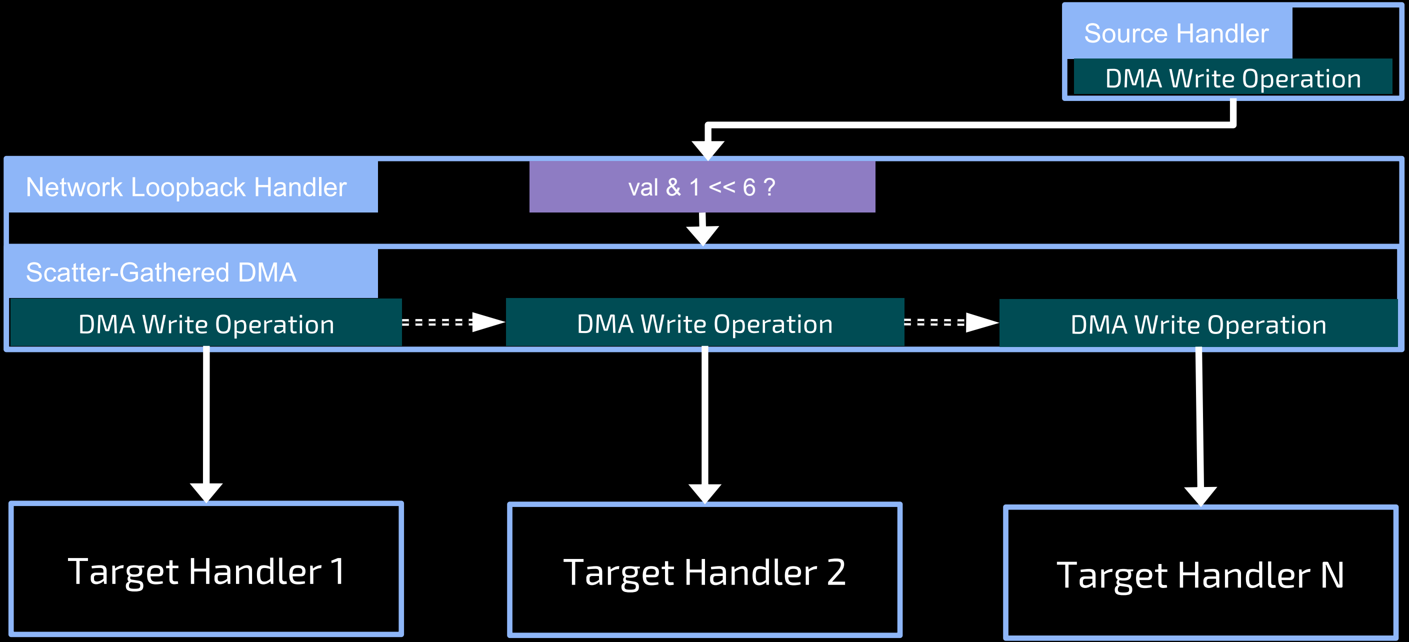
But it can still be used to DMA Write to another device

Network Loopback Mode

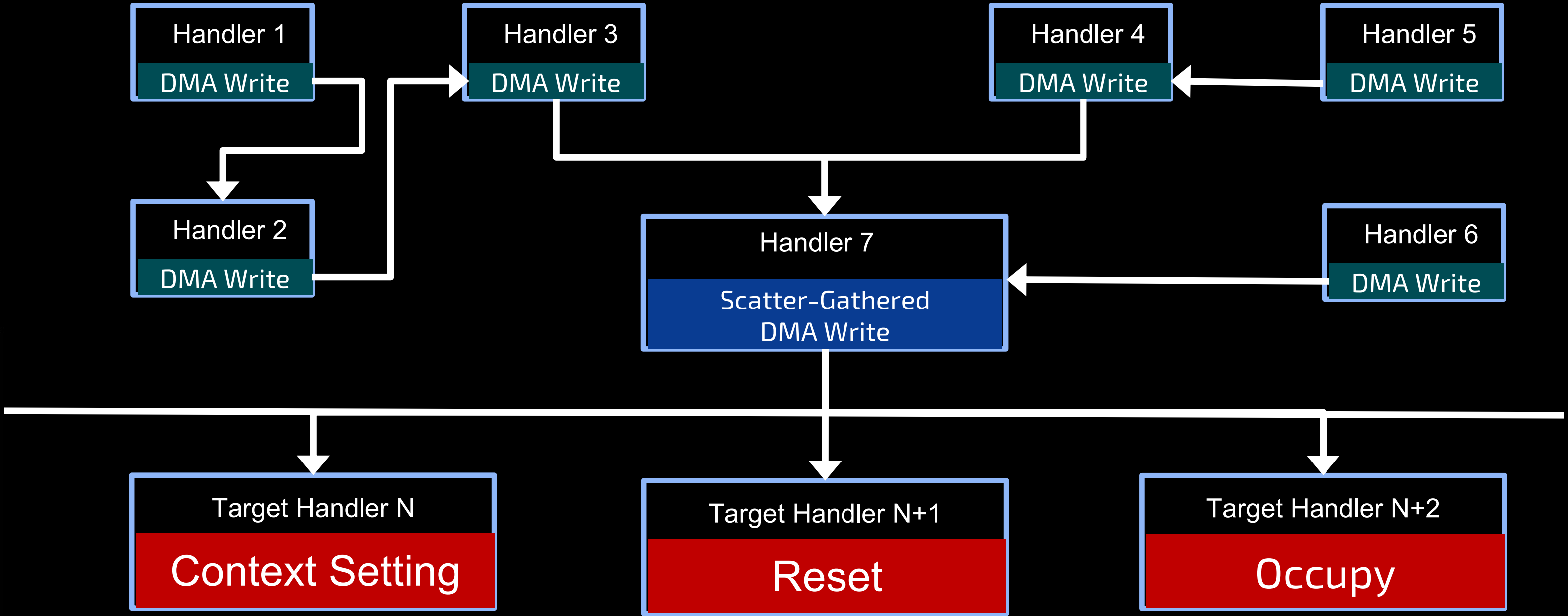
```
static ssize_t rtl8139_do_receive(NetClientState *nc, const uint8_t *buf, size_t size_, int
do_interrupt)
{
        ......
        dma_addr_t rx_addr = rtl8139_addr64(rxbufLO, rxbufHI);
        if (dot1q_buf) {
            pci_dma_write(d, rx_addr, buf, 2 * ETH_ALEN);
            pci_dma_write(d,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

Totally Controllable
Scatter-Gathered
DMA Write Operation

DMA Refraction



DMA Oriented Programing


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

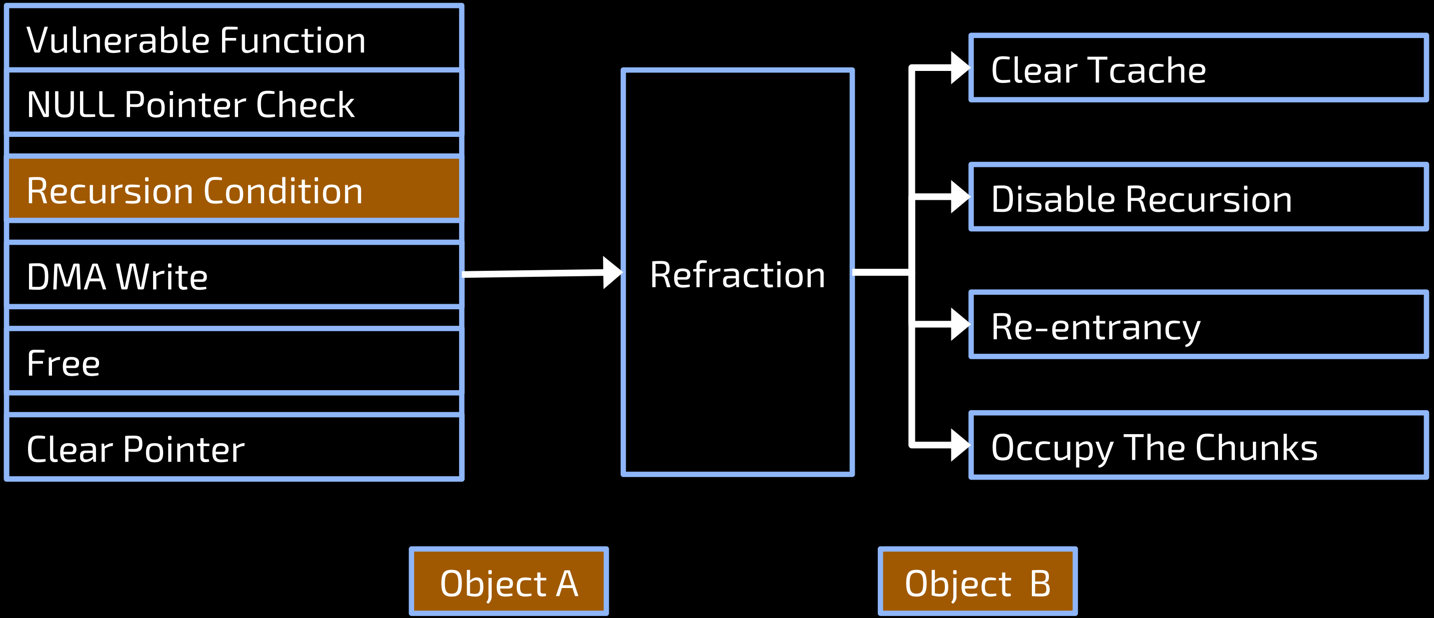
DMA Oriented Programing

- 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

- Indroduce
- Challenges
- DMA Oriented Programing
- **Exploitation** <<
- DEMO Time
- Conclusion

Primitive



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

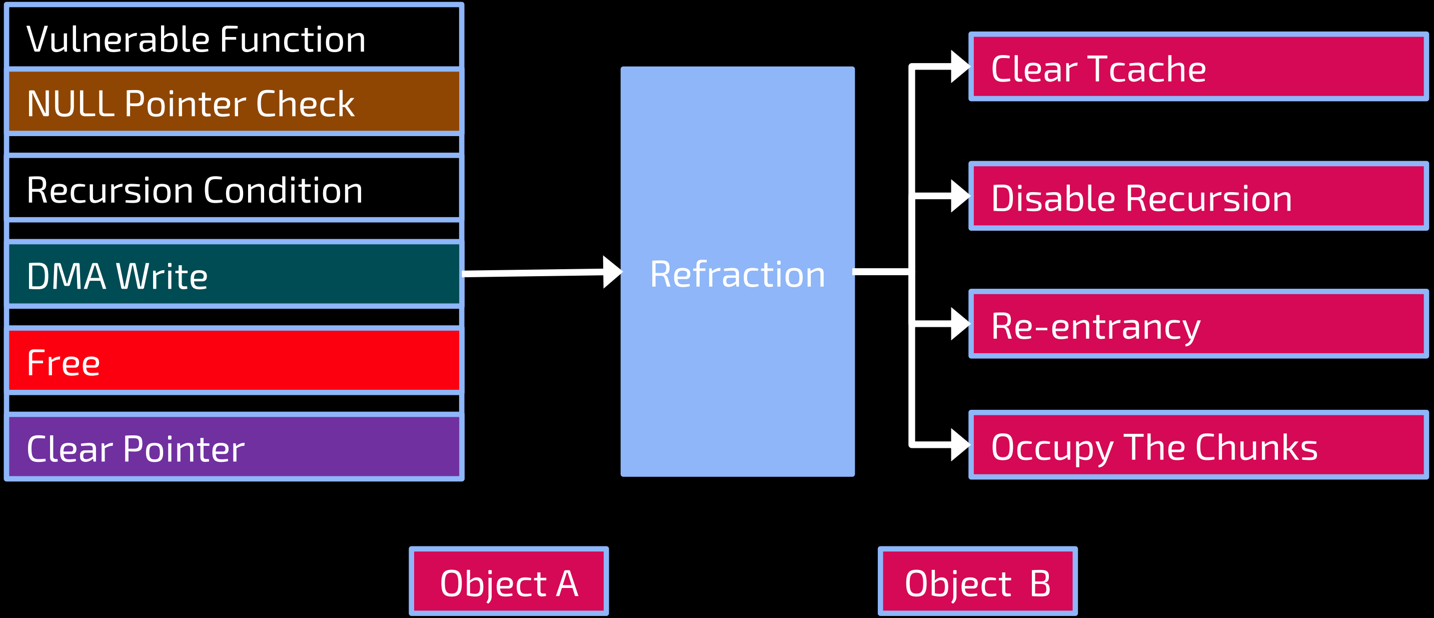
Refraction

Clear Tcache
Disable Recursion
Re-entrancy
Occupy The Chunks

Object A

Object B

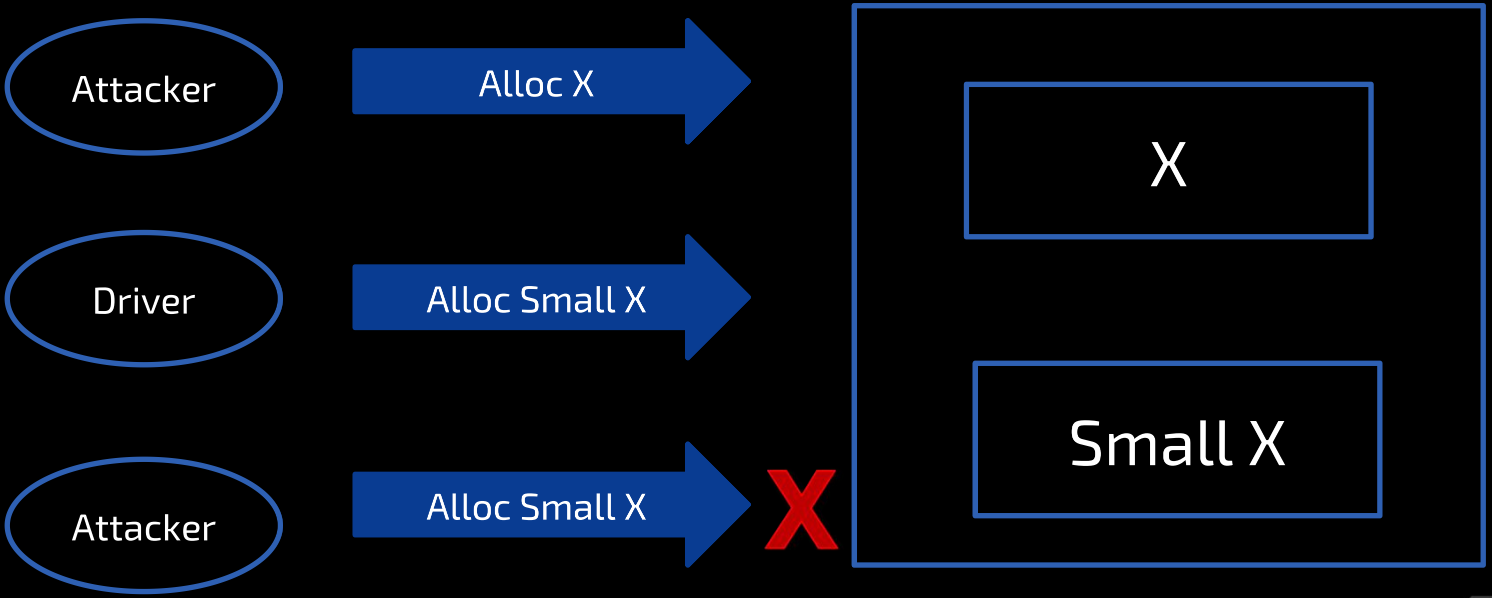
Primitive


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

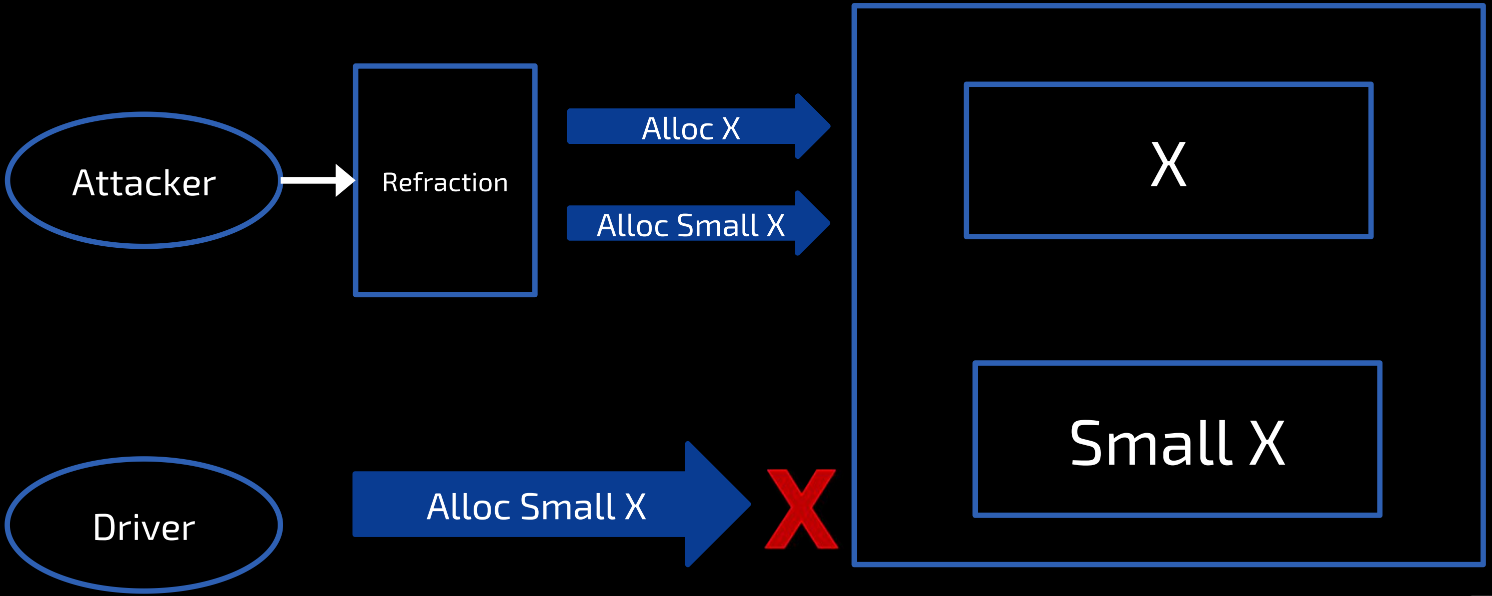
Uaf-After-uaF

- 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(Small X)
- To leak infomation from the host, occupy X with an object which could write its content to the guests
- To hijack the control flow, occupy Small X with another timer, overwrite the callback pointer with function address of `system`

Stability Optimization



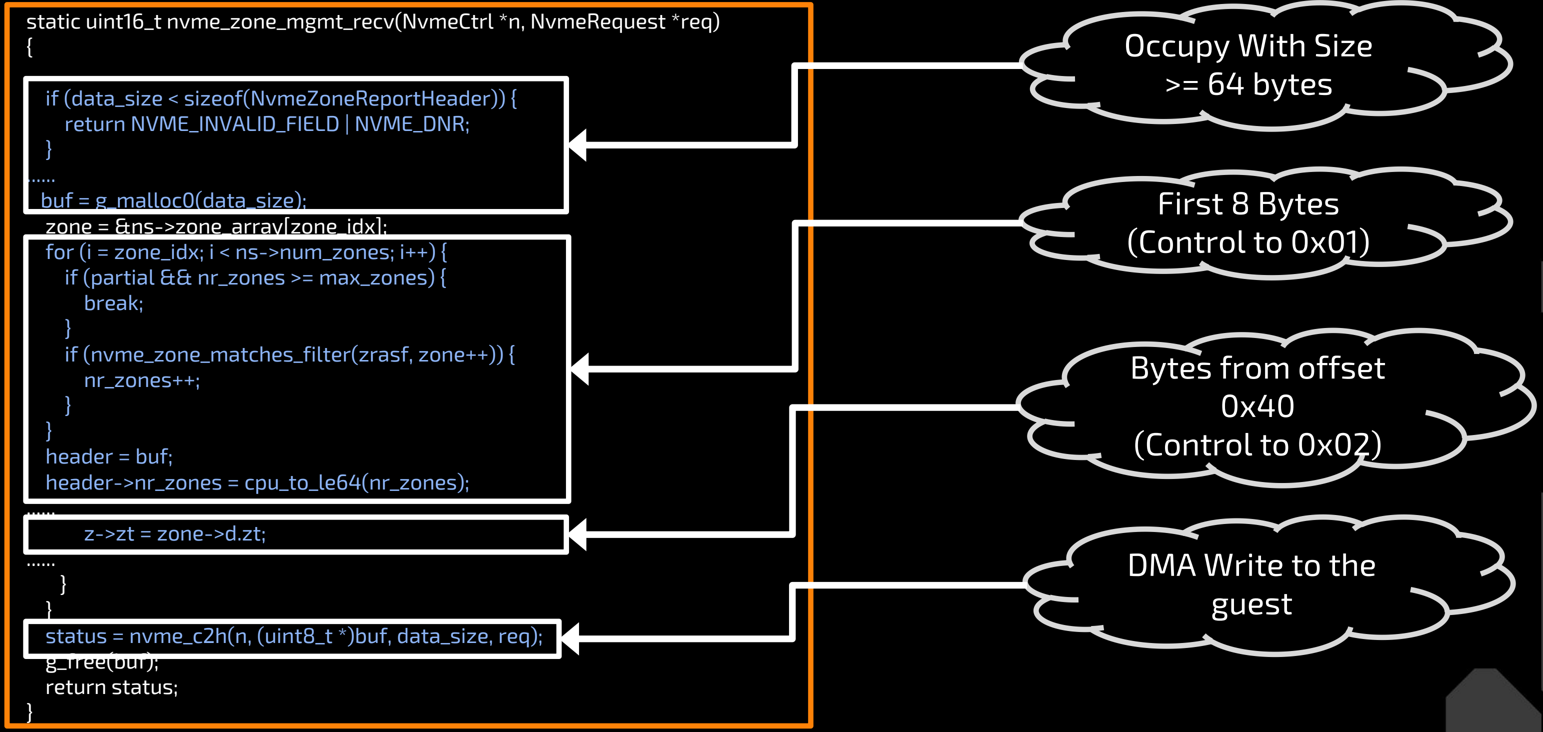
Stability Optimization



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

Info Leak



```
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

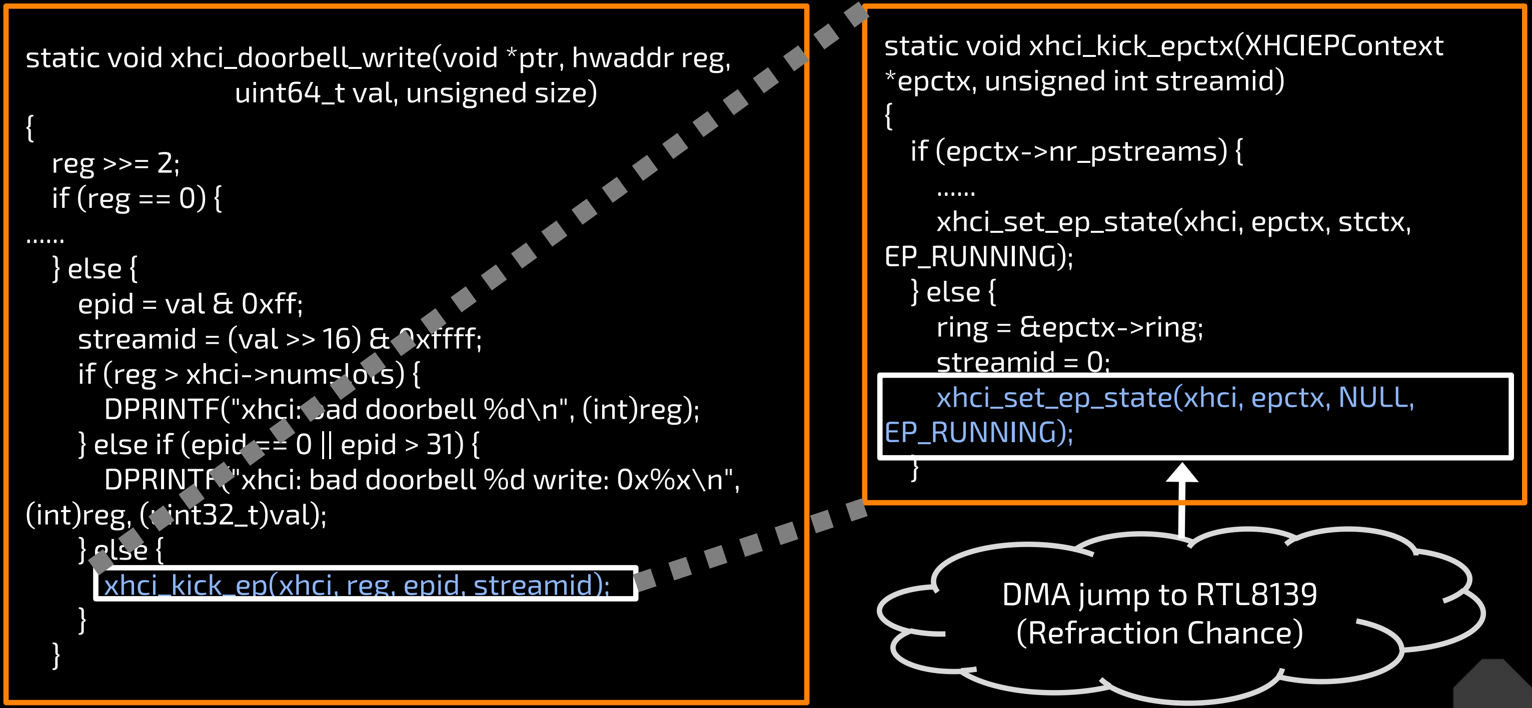
```
static MemTxResult dma_buf_rw(void *buf, dma_addr_t len, dma_addr_t
*residual,
                              QEMUSGList *sg, DMADirection dir,
                              MemTxAttrs attrs)
{
......
  while (len > 0) {
    ScatterGatherEntry entry = sg->sg[sg_cur_index++];
    dma_addr_t xfer
    res |= dma_memory_rw(sg->as, entry.base, ptr, xfer, dir, attrs);
    ptr += xfer;
    len -= xfer;
    xresidual -= xfer;
  }
......
}
```

Scatter-Gathered
DMA Write :)

But we only have value 0x01 and 0x02 :(

We must reflect the value to the netcard
to construct DMA Refraction here

Info Leak

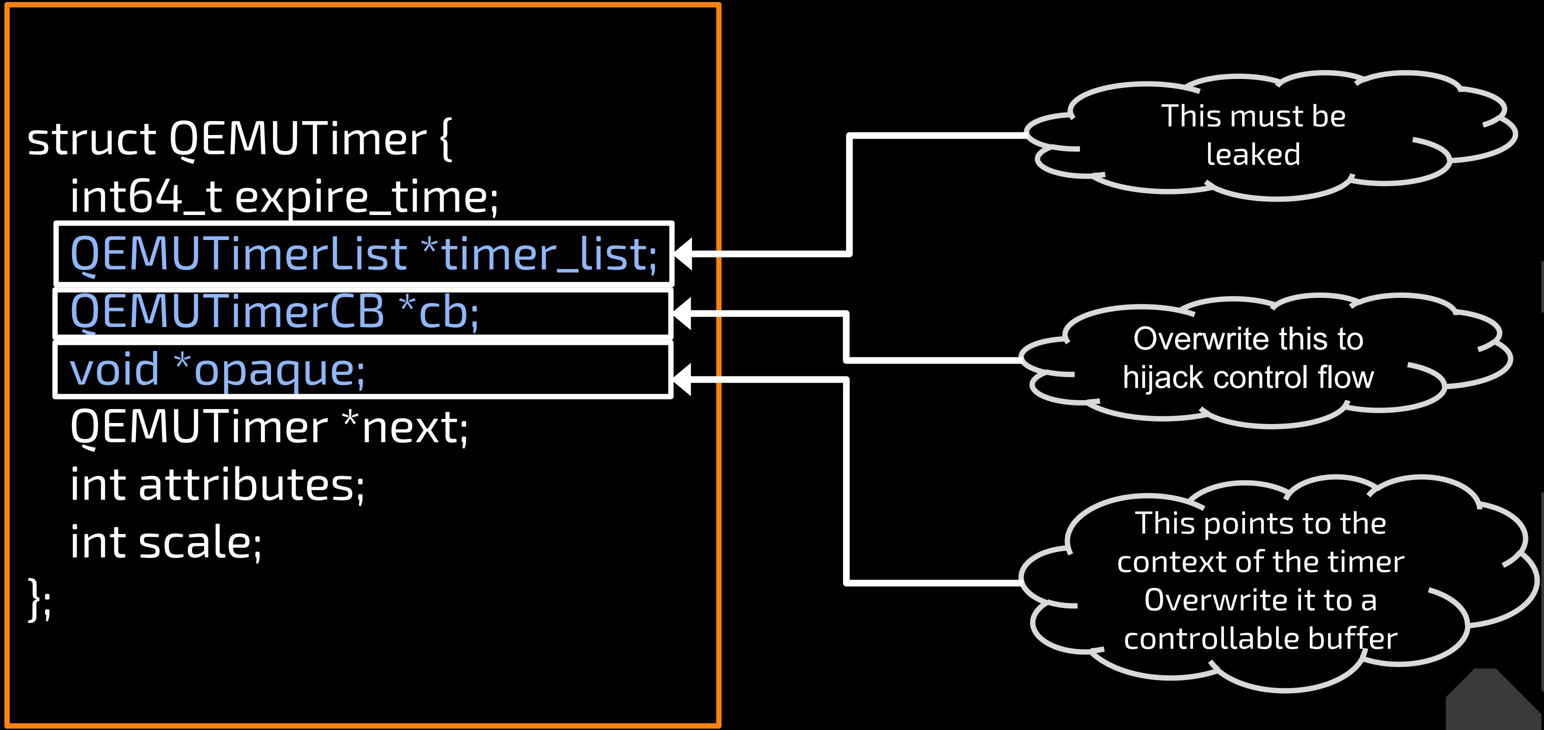


```
static void xhci_doorbell_write(void *ptr, hwaddr reg,
                                uint64_t val, unsigned size)
{
   reg >>= 2;
   if (reg == 0) {
......
   } else {
      epid = val & 0xff;
      streamid = (val >> 16) & 0xffff;
      if (reg > xhci->numslots) {
         DPRINTF("xhci: bad doorbell %d\n", (int)reg);
      } else if (epid == 0 || epid > 31) {
         DPRINTF("xhci: bad doorbell %d write: 0x%x\n",
(int)reg, (uint32_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)

Info Leak



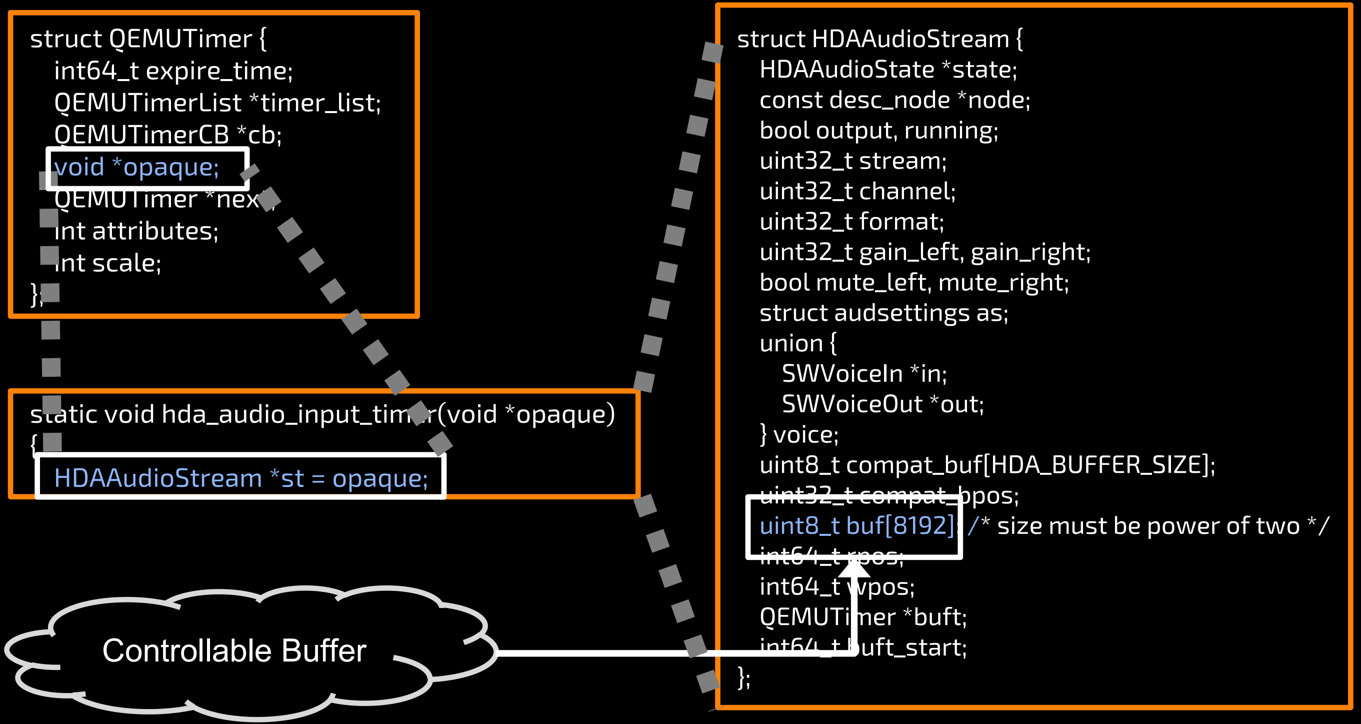
```
struct QEMUTimer {
    int64_t expire_time;
    QEMUTimerList *timer_list;
    QEMUTimerCB *cb;
    void *opaque;
    QEMUTimer *next;
    int attributes;
    int scale;
};
```

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



```
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;
```

```
struct HDAAudioStream {
    HDAAudioState *state;
    const desc_node *node;
    bool output, running;
    uint32_t stream;
    uint32_t channel;
    uint32_t format;
    uint32_t gain_left, gain_right;
    bool mute_left, mute_right;
    struct audsettings as;
    union {
        SWVoiceIn *in;
        SWVoiceOut *out;
    } voice;
    uint8_t compat_buf[HDA_BUFFER_SIZE];
    uint32_t compat_bpos;
    uint8_t buf[8192]; /* size must be power of two */
    int64_t rpos;
    int64_t wpos;
    QEMUTimer *buft;
    int64_t buft_start;
};
```

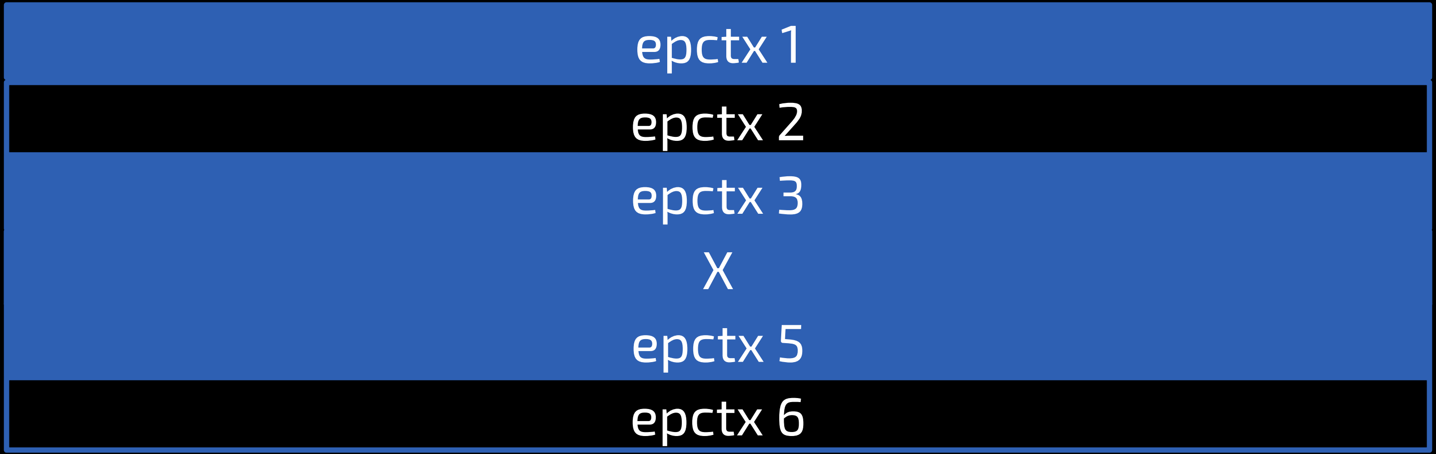
Info Leak

- 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_mgmt_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

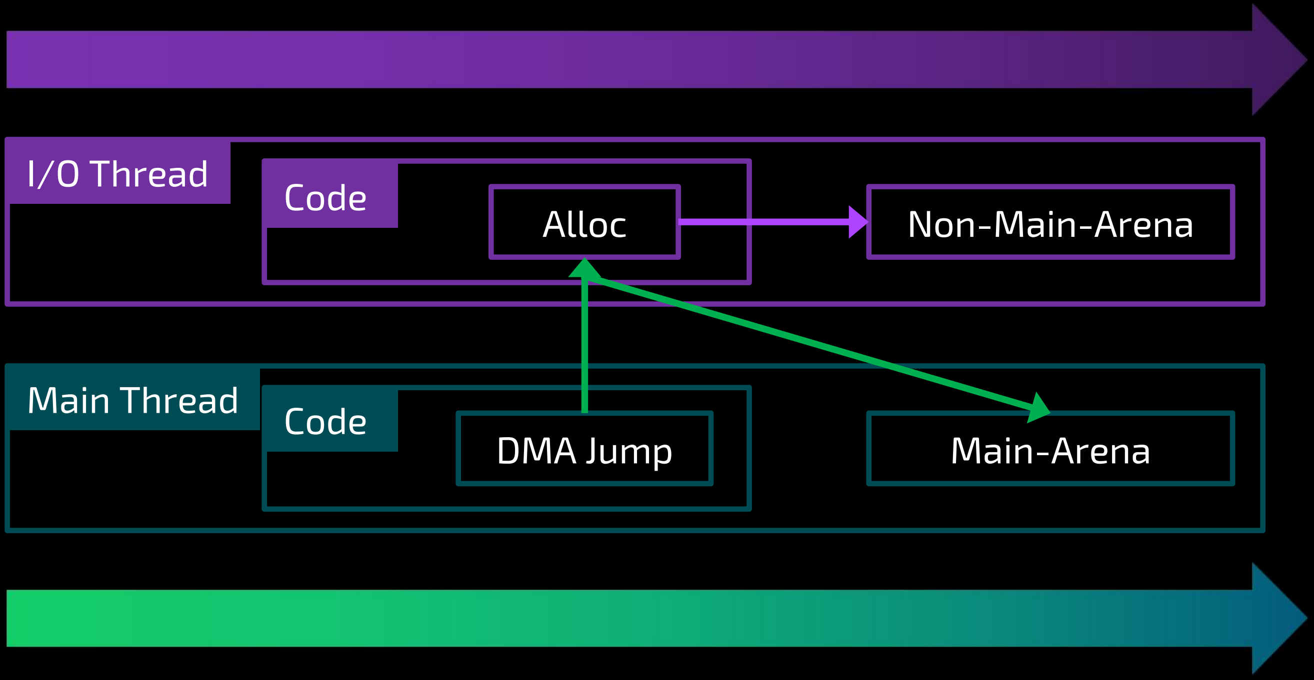
Place X In A Hole

epctx 1
epctx 2
epctx 3
epctx 4
epctx 5
epctx 6

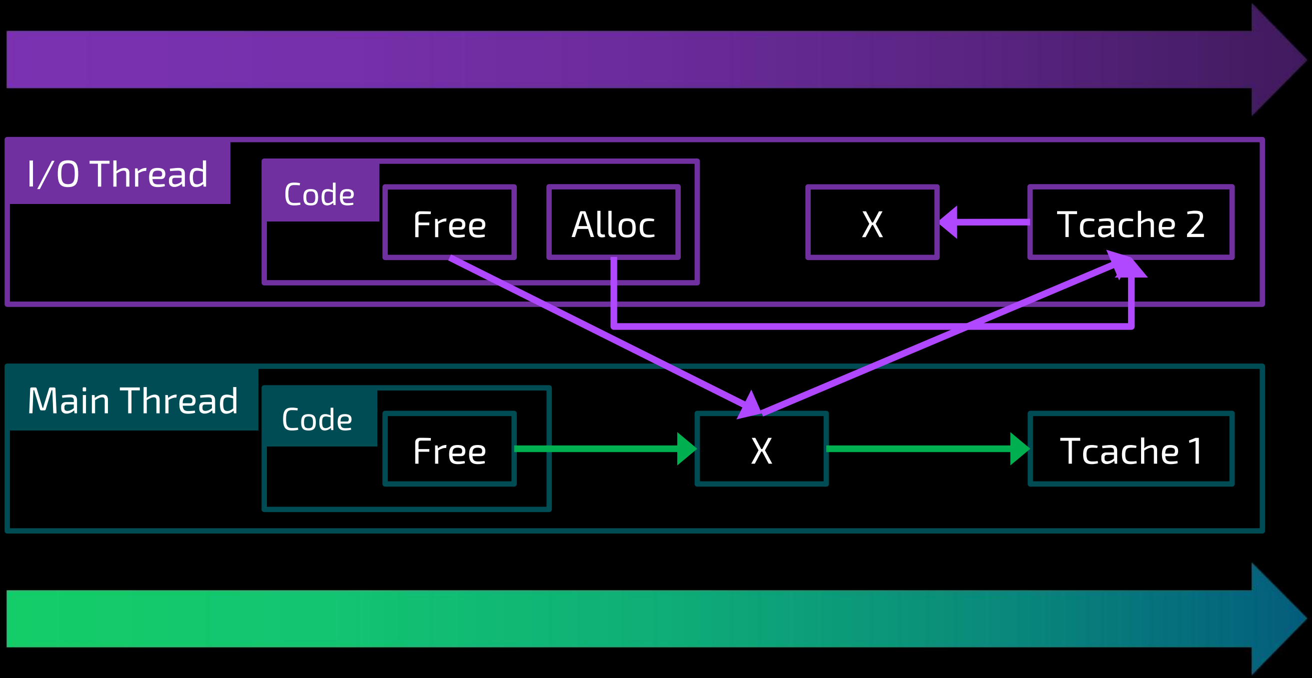
Place X In A Hole



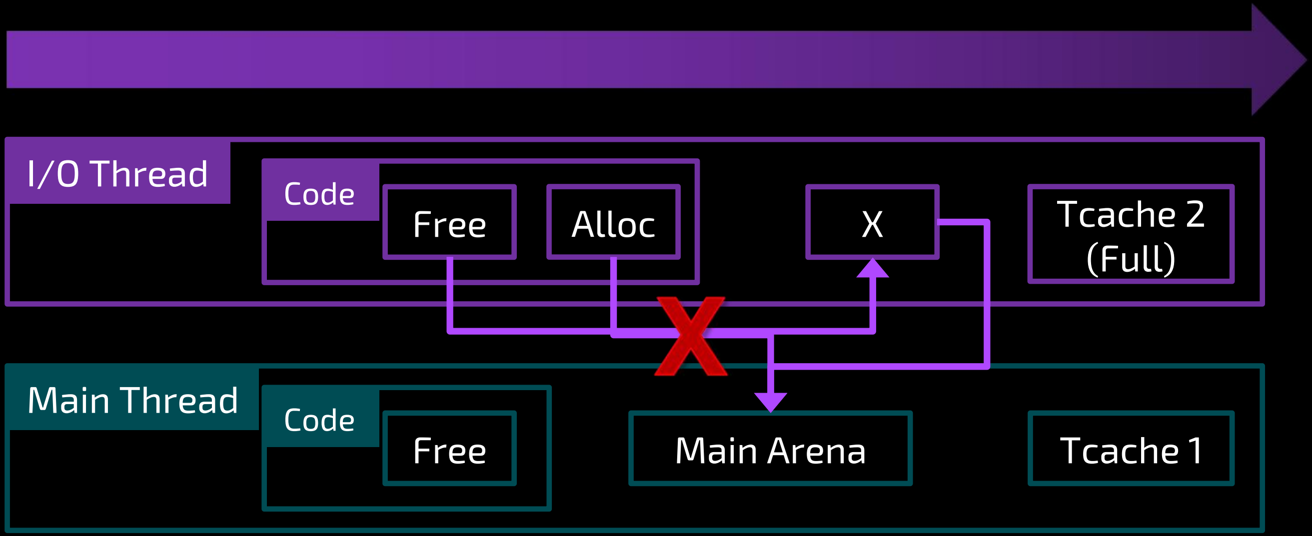
Shuttle Bettwen Threads



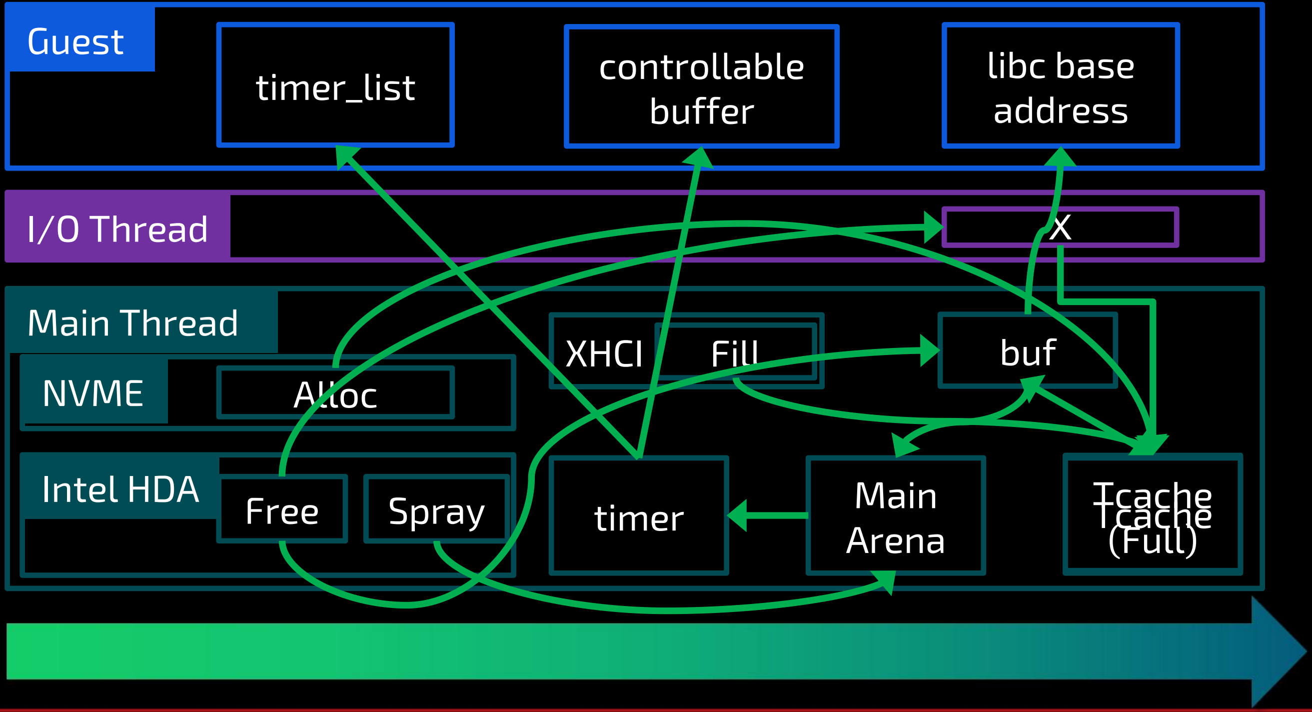
Shuttle Bettwen Threads



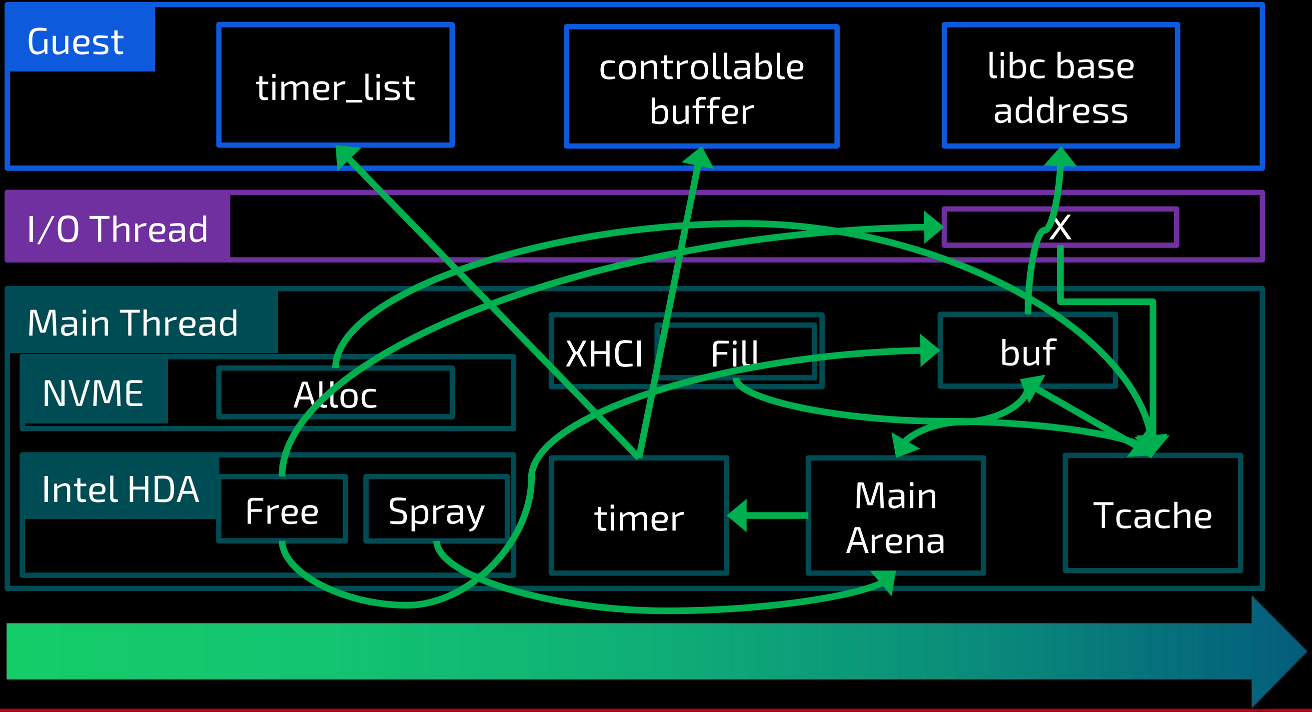
Shuttle Bettwen Threads



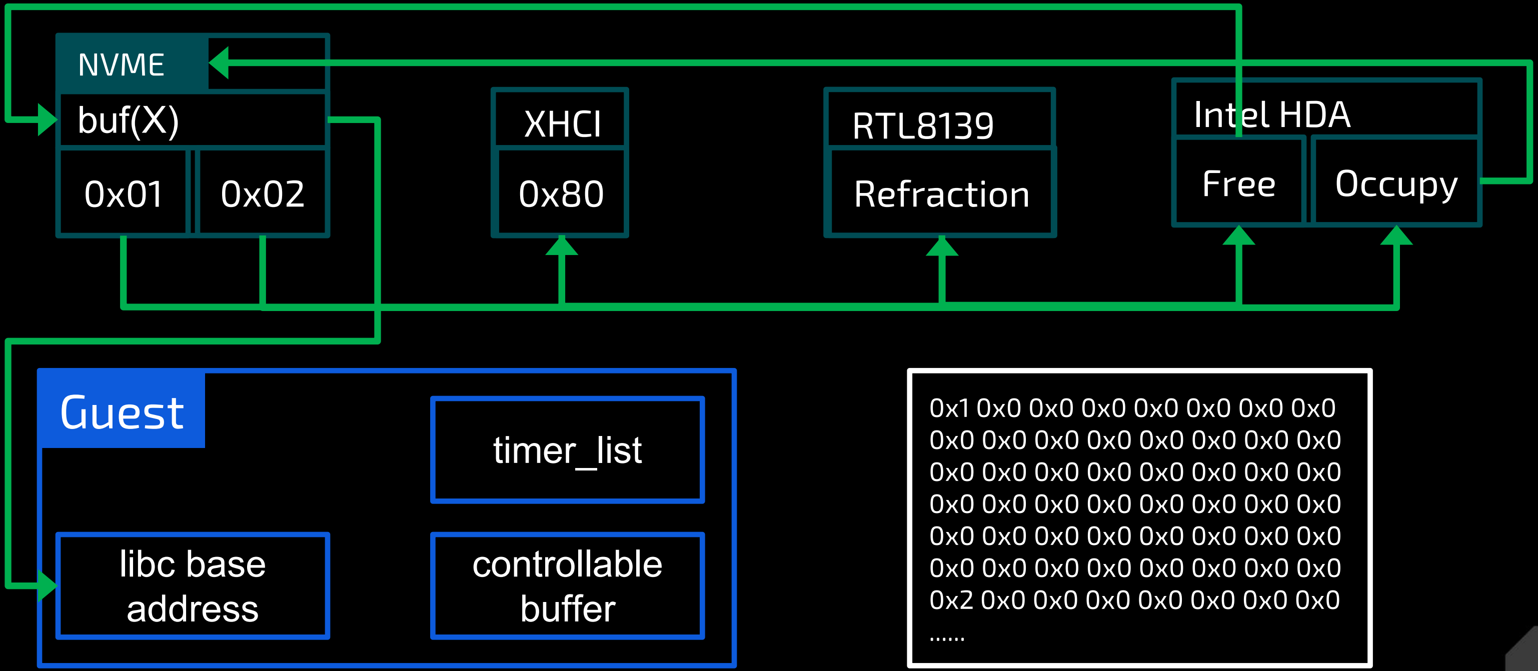
Shuttle Bettwen Threads And Devices



Shuttle Bettwen Threads And Devices



DMA-OP Chain



NVME

buf(X)

0x01 0x02

XHCI

0x80

RTL8139

Refraction

Intel HDA

Free Occupy

Guest

timer_list

libc base address

controllable buffer

```
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
0x2 0x0 0x0 0x0 0x0 0x0 0x0 0x0
……
```

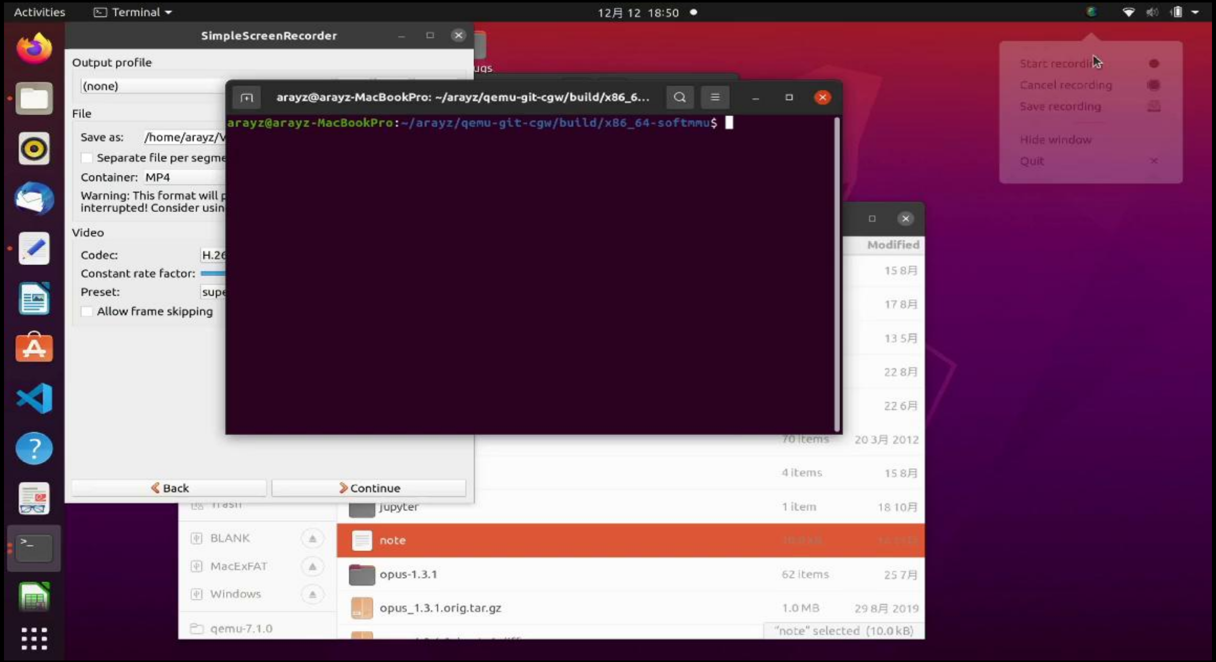
Hijack Control Flow

- 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

- Indroduce
- Challenges
- DMA Oriented Programing
- Exploitation
- **DEMO Time** <<
- Conclusion

DEMO Time



Output profile
(none)

File
Save as: /home/arayz/V
☐ Separate file per segme
Container: MP4
Warning: This format will p
interrupted! Consider usin

Video
Codec: H.26
Constant rate factor:
Preset: supe
☐ Allow frame skipping

arayz@arayz-MacBookPro: ~/arayz/qemu-git-cgw/build/x86_6...

```
arayz@arayz-MacBookPro:~/arayz/qemu-git-cgw/build/x86_64-softmmu$ 
```

		15 8月
		17 8月
		13 5月
		22 8月
		22 6月
	70 items	20 3月 2012
	4 items	15 8月
jupyter	1 item	18 10月
note		
opus-1.3.1	62 items	25 7月
opus_1.3.1.orig.tar.gz	1.0 MB	29 8月 2019

"note" selected (10.0 kB)

Agenda

- Indroduce
- 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!