CASE STUDY VM ESCAPE

VirtualBox Bug Hunting & Exploitation

鄭炳忠 (Billy)
Security Research
@st424204

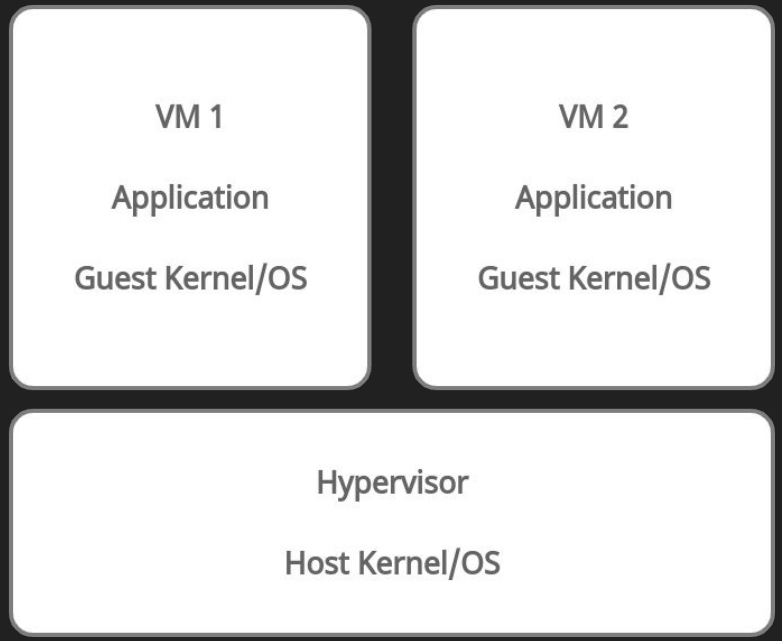
Muhd. Ramdhan
Security Research
@n0psledbyte

Outline

- What is VM Escape?
- Oracle Virtualbox Overview
- Attack Surface
- CVE-2021-2321: OOB Read Information Disclosure Vulnerability
- CVE-2021-2250: VirtualBox SLIRP Heap-based Overflow
- Demo

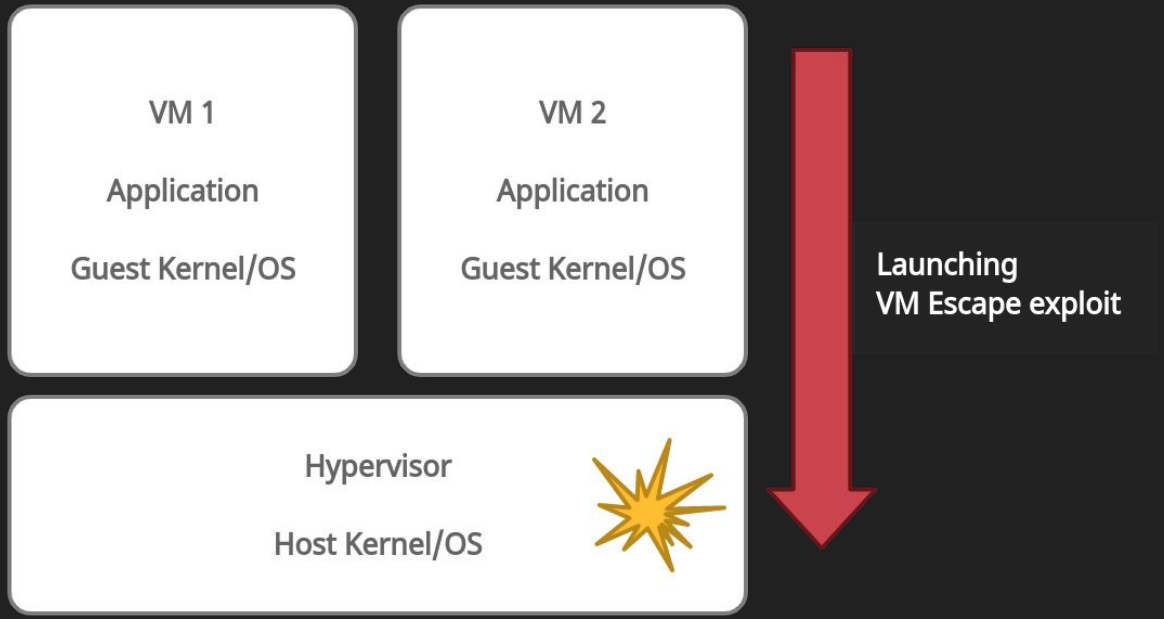
What is VM?

- Just like a normal computer that can run operating systems on it
- But this “computer” is running on the real/physical computer, we called this “computer” is VM or guest system
- Running VMs is emulated or managed by hypervisor that running on the host context



What is VM Escape?

Goals : Running arbitrary code execution in hypervisor/host context



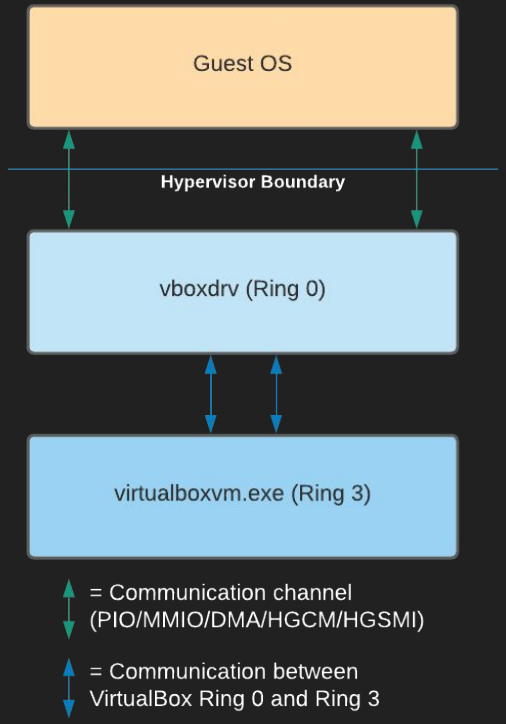
VM Escape Bug & Exploit In The Past

- CVE-2015-3456 : VENOM: buffer-overflow in QEMU virtual floppy disk controller
- CVE-2018-6981 : Uninitialized stack memory in VMware vmxnet3 virtual network adapter
- CVE-2019-2722 : Integer underflow in Oracle VirtualBox e1000 virtual network device
- CVE-2019-14378 : Heap pointer miscalculation in SLiRP (default user network backend used by QEMU)
- CVE-2021-29657 : Double fetch/TOCTOU in KVM's AMD implementation

Oracle Virtualbox Overview

- One of the popular virtualization software for desktop
- C/C++, Free, Open source (It's good for code auditing and find some memory corruption bugs ;))
- VirtualBox Extensions shipped as binaries, to support USB 2.0 and 3.0 devices, VirtualBox RDP, etc.
- Virtualbox is using Hardware-assisted Virtualization based on Intel VT-X and AMD-V

Oracle Virtualbox Architecture Overview



- For performance reason it don't switch to R3 directly
- Ring 0 mostly handling VT-X/AMD-V code and some small amount of code that handling I/O interaction from guest
- The bigger amount of code run in Ring 3
- The code that handling request from some communication channel should be interesting for attacker

Attack Surface

- Emulated devices is the most interesting attack surface, based on vulnerability in the past
- Virtualbox have a lot of emulated devices, some of them is not enabled by default:
  - Networking : E1000, virtio-net, SLiRP
  - Audio : Intel HDA, AC97
  - Graphic : VGA Device
  - USB : OHCI, EHCI, xHCI
  - Storage : AHCI
- There are other interesting attack surface through such as HGCM (Host Guest Communication Manager) to interact with specific virtualbox service such as Shared Folder, Shared Clipboard, etc

Bug Hunting

- Attack surface in network emulated devices always used in Pwn2Own 2019, 2020
- So we choose network devices first for hunting VM escape bug
- We found two bugs by code auditing that can be used for VM escape
  - CVE-2021-2321: Oracle VirtualBox e1000 Out-Of-Bounds Read Information Disclosure Vulnerability
  - CVE-2021-2250: Oracle VirtualBox SLiRP Networking Heap-based Overflow Privilege Escalation Vulnerability

CVE-2021-2321: OOB Read Information Disclosure Vulnerability

- Bug resides in code that handling e1000 frame
- Found by code auditing
- Inspired by CVE-2020-2894: e1000 Out-Of-Bounds Read Vulnerability (Shout out to Pham Hong Phi!)
- Have some (quite) complex logic bug, that can be turned into Out-Of-Bounds Read Vulnerability

CVE-2020-2894: e1000 Out-Of-Bounds Read Vulnerability

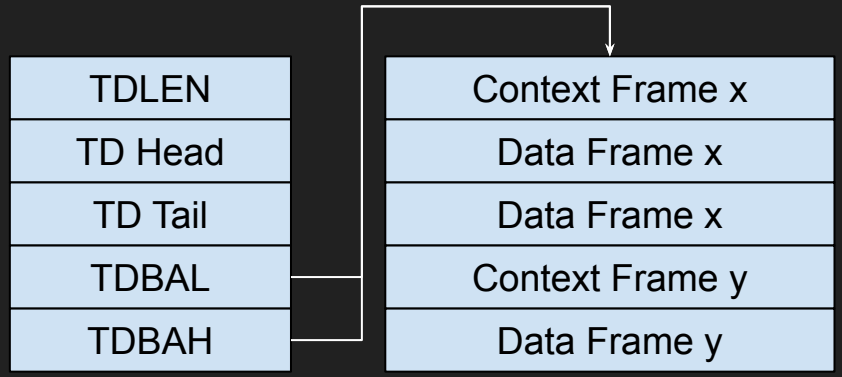
- No check against the maximum value of cse. So we can calculate checksum from OOB data.
- We can leak data after pPkt by calculating the difference between two checksums.

```
static void e1kInsertChecksum(PE1KSTATE pThis, uint8_t *pPkt, uint16_t
u16PktLen, uint8_t cso, uint8_t css, uint16_t cse)
{
    ...
    if (cse == 0)                                   // [1]
        cse = u16PktLen - 1;
    else if (cse < css)                             // [2]
    {
        E1kLog2(("%s css(%X) is greater than cse(%X), checksum is not
inserted\n",
                 pThis->szPrf, css, cse));
        return;
    }

    uint16_t u16ChkSum = e1kCSum16(pPkt + css, cse - css + 1);
    E1kLog2(("%s Inserting csum: %04X at %02X, old value: %04X\n", pThis-
>szPrf,
             u16ChkSum, cso, *(uint16_t*)(pPkt + cso)));
    *(uint16_t*)(pPkt + cso) = u16ChkSum;
}
```

CVE-2021-2321: OOB Read Information Disclosure Vulnerability

E1000 Basic



- One packet can contain one context frame followed by multiple data frames
- Last data frame in packet have End Of Packet flag

```
struct E1kTDData
{
    uint64_t u64BufAddr;                        /**< Address of data buffer */
    struct TDDCmd_st
    {
        unsigned u20DTALEN : 20; /** The total length of data pointed to by this
descriptor. */
        unsigned u4DTYP : 4; /** The descriptor type - E1K_DTYP_DATA (1). */
        unsigned fEOP : 1; /** End of packet.  Note TSCTFC update.  */
        unsigned fIFCS : 1; /** Insert Ethernet FCS/CRC (requires fEOP to be set). */
        unsigned fTSE : 1; /** Use the TSE context when set and the normal when clear.
*/
        unsigned fRS : 1; /** Report status (dw3.STA). */
        unsigned fRPS : 1; /** Reserved. 82544GC/EI defines this report packet set
(RPS).  */
        unsigned fDEXT : 1; /** Descriptor extension, must be set for this descriptor
type. */
        /** VLAN enable, requires CTRL.VME, auto enables FCS/CRC.
        *  Insert dw3.SPECIAL after ethernet header. */
        unsigned fVLE : 1;
        unsigned fIDE : 1; /** Interrupt delay enable. */
    } cmd;
    struct TDDDw3_st
    {
        unsigned fDD : 1;                       /**< Descriptor done. */
        unsigned fEC : 1;                      /**< Excess collision. */
        unsigned fLC : 1;                        /**< Late collision. */
        unsigned fTURSV : 1; /** Reserved, except for the usual oddball (82544GC/EI)
where it's called TU. */
        unsigned u4RSV : 4;                  /**< Reserved field, MBZ. */
        unsigned fIXSM : 1;                   /**< Insert IP checksum. */
        unsigned fTXSM : 1;                /**< Insert TCP/UDP checksum. */
        unsigned u6RSV : 6;                       /**< Reserved, MBZ. */
        unsigned u16Special : 16;   /**< VLAN: Id, Canonical form, Priority. */
    } dw3;
};
```

CVE-2021-2321: OOB Read Information Disclosure Vulnerability

- `e1kXmitPacket` will processing one packet frame

```
while (e1kLocateTxPacket(pThis))
{
    fIncomplete = false;
    /* Found a complete packet, allocate it. */
    rc = e1kXmitAllocBuf(pThis, pThisCC, pThis->fGSO);
    /* If we're out of bandwidth we'll come back later. */
    if (RT_FAILURE(rc))
        goto out;
    /* Copy the packet to allocated buffer and send it. */
    rc = e1kXmitPacket(pDevIns, pThis, fOnWorkerThread, &txdc);
    /* If we're out of bandwidth we'll come back later. */
    if (RT_FAILURE(rc))
        goto out;
}
```

```
static int e1kXmitPacket(PPDMDEVINS pDevIns, PE1KSTATE pThis, bool fOnWorkerThread, PE1KTXDC pTxdc)
{
    PE1KSTATECC pThisCC = PDMDEVINS_2_DATA_CC(pDevIns, PE1KSTATECC);
    int rc = VINF_SUCCESS;

        /* iterate frame */
    while (pThis->iTxDCurrent < pThis->nTxDFetched)
    {
        E1KTXDESC *pDesc = &pThis->aTxDescriptors[pThis->iTxDCurrent];
        E1kLog3(("%s About to process new TX descriptor at %08x%08x, TDLEN=%08x, TDH=%08x, TDT=%08x\n",
                 pThis->szPrf, TDBAH, TDBAL + pTxdc->tdh * sizeof(E1KTXDESC), pTxdc->tdlen, pTxdc->tdh, pTxdc->
>tdt));
                /* processing frame */
        rc = e1kXmitDesc(pDevIns, pThis, pThisCC, pDesc, e1kDescAddr(TDBAH, TDBAL, pTxdc->tdh), fOnWorkerThread);
                ...
        ++pThis->iTxDCurrent;
        if (e1kGetDescType(pDesc) != E1K_DTYP_CONTEXT && pDesc->legacy.cmd.fEOP)
            break;
    }

    return rc;
}
```

CVE-2021-2321: OOB Read Information Disclosure Vulnerability

- Useless logic error? no

```
e1kXmitPacket -> e1kXmitDesc ->
-> e1kFallbackAddToFrame -> e1kFallbackAddSegment
```

```
#define E1K_MAX_TX_PKT_SIZE    0x3fa0
...
/** TX: Transmit packet buffer use for TSE fallback and loopback. */
uint8_t     aTxPacketFallback[E1K_MAX_TX_PKT_SIZE];
```

```
static int e1kFallbackAddSegment(PPDMDEVINS pDevIns, PE1KSTATE pThis, RTGCPHYS PhysAddr, uint16_t u16Len, bool
fSend, bool fOnWorkerThread)
{
    int rc = VINF_SUCCESS;
    PE1KSTATECC pThisCC = PDMDEVINS_2_DATA_CC(pDevIns, PE1KSTATECC);
    /* TCP header being transmitted */
    struct E1kTcpHeader *pTcpHdr = (struct E1kTcpHeader *)(pThis->aTxPacketFallback + pThis->contextTSE.tu.u8CSS);
    /* IP header being transmitted */
    struct E1kIpHeader *pIpHdr = (struct E1kIpHeader *)(pThis->aTxPacketFallback + pThis->contextTSE.ip.u8CSS);

    AssertReturn(pThis->u32PayRemain + pThis->u16HdrRemain > 0, VINF_SUCCESS);

    if (pThis->u16TxPktLen + u16Len <= sizeof(pThis->aTxPacketFallback)) // [1]
        PDMDevHlpPhysRead(pDevIns, PhysAddr, pThis->aTxPacketFallback + pThis->u16TxPktLen, u16Len);
    pThis->u16TxPktLen += u16Len; // [2]
    ...
}
```

CVE-2021-2321: OOB Read Information Disclosure Vulnerability

- We don't have buffer overflow
- But, we have OOB at `e1kInsertChecksum`
- How to pass large `u16Len` ?

```
static int e1kFallbackAddSegment(PPDMDEVINS pDevIns, PE1KSTATE pThis, RTGCPHYS PhysAddr,
uint16_t u16Len, bool fSend, bool fOnWorkerThread)
{
    int rc = VINF_SUCCESS;
    PE1KSTATECC pThisCC = PDMDEVINS_2_DATA_CC(pDevIns, PE1KSTATECC);
    /* TCP header being transmitted */
    struct E1kTcpHeader *pTcpHdr = (struct E1kTcpHeader *)(pThis->aTxPacketFallback +
pThis->contextTSE.tu.u8CSS);
    /* IP header being transmitted */
    struct E1kIpHeader *pIpHdr = (struct E1kIpHeader *)(pThis->aTxPacketFallback + pThis-
>contextTSE.ip.u8CSS);

    AssertReturn(pThis->u32PayRemain + pThis->u16HdrRemain > 0, VINF_SUCCESS);

    if (pThis->u16TxPktLen + u16Len <= sizeof(pThis->aTxPacketFallback))
        PDMDevHlpPhysRead(pDevIns, PhysAddr, pThis->aTxPacketFallback + pThis-
>u16TxPktLen, u16Len);
    pThis->u16TxPktLen += u16Len;
    ...

    if (fSend)
    {
        ...
        e1kInsertChecksum(pThis, pThis->aTxPacketFallback, pThis->u16TxPktLen,
                pThis->contextTSE.ip.u8CSO,
                pThis->contextTSE.ip.u8CSS,
                pThis->contextTSE.ip.u16CSE); // [2]
        ...
        e1kInsertChecksum(pThis, pThis->aTxPacketFallback, pThis->u16TxPktLen,
                pThis->contextTSE.tu.u8CSO,
                pThis->contextTSE.tu.u8CSS,
                pThis->contextTSE.tu.u16CSE); // [3]
    }
}
```

CVE-2021-2321: OOB Read Information Disclosure Vulnerability

- How to pass large `u16Len` ?

```
static int e1kFallbackAddToFrame(PPDMDEVINS pDevIns, PE1KSTATE pThis, E1KTXDESC *pDesc, bool fOnWorkerThread)
{
    uint16_t u16MaxPktLen = pThis->contextTSE.dw3.u8HDRLEN + pThis->contextTSE.dw3.u16MSS;
    int rc = VINF_SUCCESS;
    do
    {
        /* Calculate how many bytes we have left in this TCP segment */
        uint16_t cb = u16MaxPktLen - pThis->u16TxPktLen;
        if (cb > pDesc->data.cmd.u20DTALEN)
        {
            /* This descriptor fits completely into current segment */
            cb = (uint16_t)pDesc->data.cmd.u20DTALEN; /* u20DTALEN at this point is guarantied to fit into 16 bits. */
            rc = e1kFallbackAddSegment(pDevIns, pThis, pDesc->data.u64BufAddr, cb, pDesc->data.cmd.fEOP /*fSend*/, fOnWorkerThread); // [1]
        }
        else
        {
            rc = e1kFallbackAddSegment(pDevIns, pThis, pDesc->data.u64BufAddr, cb, true /*fSend*/, fOnWorkerThread); // [2]
            /*
             * Rewind the packet tail pointer to the beginning of payload,
             * so we continue writing right beyond the header.
             */
            pThis->u16TxPktLen = pThis->contextTSE.dw3.u8HDRLEN;
        }

        pDesc->data.u64BufAddr    += cb;
        pDesc->data.cmd.u20DTALEN -= cb;
    } while (pDesc->data.cmd.u20DTALEN > 0 && RT_SUCCESS(rc));

    ...
    return VINF_SUCCESS; /// @todo consider rc;
}
```

CVE-2021-2321: OOB Read Information Disclosure Vulnerability

- How to pass large `u16Len` ?

We need pass cb more than 0x3fa0 to get OOB

```
static int e1kFallbackAddToFrame(PPDMDEVINS pDevIns, PE1KSTATE pThis, E1KTXDESC *pDesc, bool fOnWorkerThread)
{
    uint16_t u16MaxPktLen = pThis->contextTSE.dw3.u8HDRLEN + pThis->contextTSE.dw3.u16MSS;
    int rc = VINF_SUCCESS;
    do
    {
        /* Calculate how many bytes we have left in this TCP segment */
        uint16_t cb = u16MaxPktLen - pThis->u16TxPktLen;
        if (cb > pDesc->data.cmd.u20DTALEN)
        {
            /* This descriptor fits completely into current segment */
            cb = (uint16_t)pDesc->data.cmd.u20DTALEN; /* u20DTALEN at this point is guarantied to fit into 16 bits. */
            rc = e1kFallbackAddSegment(pDevIns, pThis, pDesc->data.u64BufAddr, cb, pDesc->data.cmd.fEOP /*fSend*/, fOnWorkerThread); // [1]
        }
        else
        {
            rc = e1kFallbackAddSegment(pDevIns, pThis, pDesc->data.u64BufAddr, cb, true /*fSend*/, fOnWorkerThread); // [2]
            /*
             * Rewind the packet tail pointer to the beginning of payload,
             * so we continue writing right beyond the header.
             */
            pThis->u16TxPktLen = pThis->contextTSE.dw3.u8HDRLEN;
        }

        pDesc->data.u64BufAddr    += cb;
        pDesc->data.cmd.u20DTALEN -= cb;
    } while (pDesc->data.cmd.u20DTALEN > 0 && RT_SUCCESS(rc));

    ...
    return VINF_SUCCESS; /// @todo consider rc;
}
```

CVE-2021-2321: OOB Read Information Disclosure Vulnerability

- How to pass large `u16Len` ?

```
static int e1kFallbackAddToFrame(PPDMDEVINS pDevIns, PE1KSTATE pThis, E1KTXDESC *pDesc, bool fOnWorkerThread)
{
    uint16_t u16MaxPktLen = pThis->contextTSE.dw3.u8HDRLEN + pThis->contextTSE.dw3.u16MSS;
    int rc = VINF_SUCCESS;
    do
    {
        /* Calculate how many bytes we have left in this TCP segment */
        uint16_t cb = u16MaxPktLen - pThis->u16TxPktLen;
        if (cb > pDesc->data.cmd.u20DTALEN)
        {
            /* This descriptor fits completely into current segment */
            cb = (uint16_t)pDesc->data.cmd.u20DTALEN; /* u20DTALEN at this point is guarantied to fit into 16 bits. */
            rc = e1kFallbackAddSegment(pDevIns, pThis, pDesc->data.u64BufAddr, cb, pDesc->data.cmd.fEOP /*fSend*/, fOnWorkerThread); // [
        }
        else
        {
            rc = e1kFallbackAddSegment(pDevIns, pThis, pDesc->data.u64BufAddr, cb, true /*fSend*/, fOnWorkerThread); // [2]
            /*
             * Rewind the packet tail pointer to the beginning of payload,
             * so we continue writing right beyond the header.
             */
            pThis->u16TxPktLen = pThis->contextTSE.dw3.u8HDRLEN;
        }

        pDesc->data.u64BufAddr    += cb;
        pDesc->data.cmd.u20DTALEN -= cb;
    } while (pDesc->data.cmd.u20DTALEN > 0 && RT_SUCCESS(rc));

    return VINF_SUCCESS; /// @todo consider rc;
}
```

We need pass cb more than 0x3fa0 to get OOB

We can control `u16MaxPktLen` but it never more than 0x3fa0

CVE-2021-2321: OOB Read Information Disclosure Vulnerability

- How to pass large `u16Len` ? another problem ..

```
static int e1kFallbackAddToFrame(PPDMDEVINS pDevIns, PE1KSTATE pThis, E1KTXDESC *pDesc, bool fOnWorkerThread)
{
    uint16_t u16MaxPktLen = pThis->contextTSE.dw3.u8HDRLEN + pThis->contextTSE.dw3.u16MSS;
    int rc = VINF_SUCCESS;
    do
    {
        /* Calculate how many bytes we have left in this TCP segment */
        uint16_t cb = u16MaxPktLen - pThis->u16TxPktLen;
        if (cb > pDesc->data.cmd.u20DTALEN)
        {
            /* This descriptor fits completely into current segment */
            cb = (uint16_t)pDesc->data.cmd.u20DTALEN; /* u20DTALEN at this point is guarantied to fit into 16 bits. */
            rc = e1kFallbackAddSegment(pDevIns, pThis, pDesc->data.u64BufAddr, cb, pDesc->data.cmd.fEOP /*fSend*/, fOnWorkerThread); // [
        }
        else
        {
            rc = e1kFallbackAddSegment(pDevIns, pThis, pDesc->data.u64BufAddr, cb, true /*fSend*/, fOnWorkerThread); // [2]
            /*
             * Rewind the packet tail pointer to the beginning of payload,
             * so we continue writing right beyond the header.
             */
            pThis->u16TxPktLen = pThis->contextTSE.dw3.u8HDRLEN;
        }

        pDesc->data.u64BufAddr    += cb;
        pDesc->data.cmd.u20DTALEN -= cb;
    } while (pDesc->data.cmd.u20DTALEN > 0 && RT_SUCCESS(rc));

    return VINF_SUCCESS; /// @todo consider rc;
}
```

We need pass cb more than 0x3fa0 to get OOB

We can control `u16MaxPktLen` but it never more than 0x3fa0

Somehow, we need to make `pThis->u16TxPktLen` to create **int overflow**, so we can pass large `cb`

CVE-2021-2321: OOB Read Information Disclosure Vulnerability

- Another problem is to make `pThis->u16PktLen` bigger than `u16MaxPktLen` (`pThis->contextTSE.dw3.u16MSS + pThis->contextTSE.dw3.u8HDRLEN`)
- While processing one packet frame, there's no way to make `pThis->u16PktLen` bigger than `u16MaxPktLen`
- We already know, we can control `u16MaxPktLen` but it can't have more than `E1K_MAX_TX_PKT_SIZE` (0x3fa0)
- Somehow, we control `pThis->u16PktLen` to some value, and then in the next packet frame we control `u16MaxPktLen` to be less than `pThis->u16PktLen`

CVE-2021-2321: OOB Read Information Disclosure Vulnerability

- Can we control `pThis->u16PktLen` for the next processing packet frame?
- Last data frame contain `fEOP` enabled, and seems it will always clear `pThis->u16PktLen`, is there a way to make `pThis->u16PktLen` still alive for the next frame?

```
if (pDesc->data.cmd.u20DTALEN == 0 || pDesc->data.u64Buf
{
    E1kLog2(("% Empty data descriptor, skipped.\n", pThi
    if (pDesc->data.cmd.fEOP)
    {
        e1kTransmitFrame(pDevIns, pThis, pThisCC, fOnWor
        pThis->u16TxPktLen = 0;
    }
}
```

```
if (pDesc->data.cmd.fEOP)
{
    /* End of packet, next segment will contain header. */
    if (pThis->u32PayRemain != 0)
        E1K_INC_CNT32(TSCTFC);
    pThis->u16TxPktLen = 0;
    e1kXmitFreeBuf(pThis, PDMDEVINS_
```

```
else if (pDesc->legacy.cmd.fEOP)
{
    e1kXmitFreeBuf(pThis, pThisCC);
    pThis->u16TxPktLen = 0;
}
```

```
if (pDesc->data.cmd.fEOP)
{
    ...
    pThis->u16TxPktLen = 0;
```

CVE-2021-2321: OOB Read Information Disclosure Vulnerability

- We can just set `fDD` enabled in last data frame to avoid those checks!

```
static int e1kXmitDesc(PPDMDEVINS pDevIns, PE1KSTATE pThis, PE1KSTATECC pThisCC,
E1KTXDESC *pDesc,
        RTGCPHYS addr, bool fOnWorkerThread)
{
    int rc = VINF_SUCCESS;

    e1kPrintTDesc(pThis, pDesc, "vvv");

    if (pDesc->legacy.dw3.fDD)
    {
        E1kLog(("%s e1kXmitDesc: skipping bad descriptor ^^^\n", pThis->szPrf));
        e1kDescReport(pDevIns, pThis, pDesc, addr);
        return VINF_SUCCESS;
    }
    ...
    /* some check will happen to clear pThis->u16TxPktLen
     * if pDesc is last data frame */
}
```

CVE-2021-2321: OOB Read Information Disclosure Vulnerability

Recap

- Logic error in mishandling `fDD` flag that allowed us to make int overflow at `e1kFallbackAddToFrame`
- Using int overflow, we can pass large value to `e1kFallbackAddSegment`
- Because of some missing check handling in `e1kFallbackAddSegment`, we can make `pThis->u16PktLen` large than its buffer, and it allowed us to create OOB Read in `e1kInsertCheckSum`

CVE-2021-2321: OOB Read Information Disclosure Vulnerability

- We can pass pass large `u16Len` to make OOB at **[1]** and **[2]**

```
static int e1kFallbackAddSegment(PPDMDEVINS pDevIns, PE1KSTATE pThis, RTGCPHYS PhysAddr, uint16_t u16Len, bool fSend,
bool fOnWorkerThread)
{
  ...

  pThis->u16TxPktLen += u16Len;
  ...

  if (fSend)
  {
    ...
    pIpHdr->chksum = 0;
    e1kInsertChecksum(pThis, pThis->aTxPacketFallback, pThis->u16TxPktLen,
            pThis->contextTSE.ip.u8CSO,
            pThis->contextTSE.ip.u8CSS,
            pThis->contextTSE.ip.u16CSE); // [1]

    ...
    e1kInsertChecksum(pThis, pThis->aTxPacketFallback, pThis->u16TxPktLen,
            pThis->contextTSE.tu.u8CSO,
            pThis->contextTSE.tu.u8CSS,
            pThis->contextTSE.tu.u16CSE); // [2]

    // send packet to localhost (to retreive the checksum information)
  }
}
```

```
static void e1kInsertChecksum(PE1KSTATE pThis, uint8_t *pPkt, uint16_t u16PktLen, uint8_t
cso, uint8_t css, uint16_t cse, bool fUdp = false)
{
    ...
    int16_t u16ChkSum = e1kCSum16(pPkt + css, cse - css + 1);
    if (fUdp && u16ChkSum == 0)
        u16ChkSum = ~u16ChkSum;     /* 0 means no checksum computed in case of UDP (see
@bugref{9883}) */
    E1kLog2(("%s Inserting csum: %04X at %02X, old value: %04X\n", pThis->szPrf,
                u16ChkSum, cso, *(uint16_t*)(pPkt + cso)));
    *(uint16_t*)(pPkt + cso) = u16ChkSum;
}
```

CVE-2021-2321: OOB Read Information Disclosure Vulnerability

Recap (next)

- `e1kInsertCheckSum` can calculate checksum from data out of the bound from its buffer
- Using checksum value information we can leak two bytes behind the buffer at a time by calculating the difference between two checksums
- We can retrieve VBoxDD base address to bypass ASLR and building payload for our ROP Gadgets

CVE-2021-2250: VirtualBox SLIRP Heap-based Overflow

- SLIRP is one of the attack surface enabled by default in virtualbox
- Used for user-mode networking by emulating TCP/IP protocol
- This bug resides in emulating of ICMP protocol when the guest try to send ICMP request
- This bug only affected windows host only

CVE-2021-2250: VirtualBox SLIRP Heap-based Overflow

- Size of `struct ip` is only 20 bytes
- We can control `hlen` (IP header length) up to 60 bytes, so we can overwrite `pong->bufsize`
- `pong->bufsize` will be used as reply size, by overwriting it we can receive ICMP reply buffer larger than its size (heap overflow)

```
void
icmpwin_ping(PNATState pData, struct mbuf *m, int hlen)
{
    struct ip *ip = mtod(m, struct ip *);
        ...

    reqsize = ip->ip_len - hlen - sizeof(struct icmp_echo);

    bufsize = sizeof(ICMP_ECHO_REPLY);
    if (reqsize < sizeof(IO_STATUS_BLOCK) + sizeof(struct icmp_echo))
        bufsize += sizeof(IO_STATUS_BLOCK) + sizeof(struct icmp_echo);
    else
        bufsize += reqsize;
    bufsize += 16; /* whatever that is; empirically at least XP needs it
*/

    pongsize = RT_UOFFSETOF(struct pong, buf) + bufsize;

    pong = RTMemAlloc(pongsize);
    if (RT_UNLIKELY(pong == NULL))
        return;

    pong->pData = pData;
    pong->bufsize = bufsize;
    m_copydata(m, 0, hlen, (caddr_t)&pong->reqiph); // [1]
        ...

    status = IcmpSendEcho2(pData->icmp_socket.sh, NULL,
                           g_pfnIcmpCallback, pong,
                           dst, reqdata, (WORD)reqsize, &opts,
                           pong->buf, (DWORD)pong->bufsize,
                           5 * 1000 /* ms */); // [2]
        ...
```

```
struct pong {
    PNATState pData;
    TAILQ_ENTRY(pong) queue_entry;
    struct ip reqiph;
    struct icmp_echo reqicmph;
    size_t bufsize;
    uint8_t buf[1];
};
```

Heap-Overflow Exploitation

- We can control `pongsize` to overwrite next heap chunk with arbitrary size and arbitrary content
- We found that there is `struct socket` that we can overwrite, by overwriting this object we can control the program execution
- `struct socket` will be created everytime we create TCP/IP connection to the outside. We can just create a bunch of ICMP requests, to spray `struct socket` object in heap

```
struct socket
{
    struct socket   *so_next;
    struct socket   *so_prev;    /* For a linked list of sockets */

#if !defined(RT_OS_WINDOWS)
    int s;                       /* The actual socket */
#else
    union {
        int s;
        HANDLE sh;
    };
    uint64_t so_icmp_id; /* XXX: hack */
    uint64_t so_icmp_seq; /* XXX: hack */
#endif

    /* XXX union these with not-yet-used sbuf params */
    struct mbuf     *so_m; /* Pointer to the original SYN packet,
                            * for non-blocking connect()'s, and
                            * PING reply's */
    ...
};
```

Heap-Overflow Exploitation

```
#if defined(RT_OS_WINDOWS)
void slirp_select_poll(PNATState pData, int fTimeout)
#else /* RT_OS_WINDOWS */
void slirp_select_poll(PNATState pData, struct pollfd *polls, int
ndfs)
#endif /* !RT_OS_WINDOWS */
{
    struct socket *so, *so_next;
    int ret;
#if defined(RT_OS_WINDOWS)
    WSANETWORKEVENTS NetworkEvents;
    int rc;
    int error;
#endif
    ...
#if defined(RT_OS_WINDOWS)
        icmpwin_process(pData); // [1]
#else
    if (   (pData->icmp_socket.s != -1)
         && CHECK_FD_SET(&pData->icmp_socket, ignored, readfds))
        sorecvfrom(pData, &pData->icmp_socket);
#endif
    /*
     * Check TCP sockets
     */
    QSOCKET_FOREACH(so, so_next, tcp) // [2]
        /* { */
        Assert(!so->fUnderPolling);
    so->fUnderPolling = 1;
    if (slirpVerifyAndFreeSocket(pData, so)) // [3]
        CONTINUE(tcp);
```

- There is a function which collect `struct socket` object for handling events e.g: there is ICMP reply coming back
- In **[1]** it will process ICMP reply and heap overflow will happened on some `struct object`
- In **[2]** will collecting `struct socket` object including the one we overwritten
- An overwritten `struct socket` will be processed in **[3]**

Heap-Overflow Exploitation

- We control and set `fShouldBeRemoved` to 1 so it will call `sofree` with our controlled `pSocket` object.
- Then we call `m_freem` with pointer `pSocket->so_m` (mbuf object) controlled
- VRAM address is always spans in range `0xcb10000-0x1322000`, so it's safe to use `0x10000000` as our fake mbuf object for `so_m`
- Then it will call `m_freem -> m_free -> uma_zfree -> uma_zfree_arg -> slirp_uma_free`

```
static int slirpVerifyAndFreeSocket(PNATState pData, struct socket
*pSocket)
{
    AssertPtrReturn(pData, 0);
    AssertPtrReturn(pSocket, 0);
    AssertReturn(pSocket->fUnderPolling, 0);
    if (pSocket->fShouldBeRemoved) //[4]
    {
        pSocket->fUnderPolling = 0;
        sofree(pData, pSocket);
        /* pSocket is PHANTOM, now */
        return 1;
    }
    return 0;
}
```

```
void
sofree(PNATState pData, struct socket *so)
{
    LogFlowFunc(("ENTER:%R[natsock]\n", so));
        ...
    /* check if mbuf haven't been already freed  */
    if (so->so_m != NULL)
    {
        m_freem(pData, so->so_m); //[5]
        so->so_m = NULL;
    }
}
```

Heap-Overflow Exploitation

```
static void slirp_uma_free(void *item, int size, uint8_t flags)
{
  struct item *it;
  uma_zone_t zone;
  ...
  it = &((struct item *)item)[-1];
  zone = it->zone;
  /* check border magic */
  ...
  if (zone->pfFini)
  {
  zone->pfFini(zone->pData, item, (int /*sigh*/)zone->size); //[6]
  }
}
```

- We can control `item`, and redirect program execution in **[6]**
- From the VBoxDD address we retrieve with CVE-2021-2321. We redirect address to stack pivot gadget, and execute our RopChain we already put at VRAM address
- Then the RopChain will prepare the shellcode and will execute `calc.exe`!

DEMO

Conclusions

- Most hypervisors still written in C/C++ which very vulnerable to memory corruption bugs
- Even system nowadays uses many exploit mitigation such as NX, ASLR, but still can be bypassed with good bug and good exploitation primitive
- Sometimes bug that found by manual code auditing have good quality in exploitation point of view (and not hanging fruit ones that found by fuzzing)
- With our exploitation, reliability of exploit can reach 85%+, which is good
- Don’t be scared with complex piece of software, sometimes you just need understand one piece of code/component to fully hacked it

References

- https://phoenhex.re/2018-07-27/better-slow-than-sorry
- https://github.com/0xKira/qemu-vm-escape
- https://starlabs.sg/blog/2020/04/adventures-in-hypervisor-oracle-virtualbox-research/
- https://starlabs.sg/blog/2020/06/oracle-virtualbox-vhwa-use-after-free-privilege-escalation-vulnerability/
- https://github.com/MorteNoir1/virtualbox_e1000_0day
- https://github.com/hongphipham95/Vulnerabilities/blob/master/VirtualBox/Pwn2Own%202020/Pwn2Own%202020%20-%20Oracle%20VirtualBox%20Escape.md
- https://en.wikipedia.org/wiki/Virtual_machine_escape