HTTPS://CONFERENCE.HITB.ORG/HITBSECCONF2024BKK

HITB SECCONF 2024 BANGKOK

Dragon Slaying Guide Bug Hunting In VMware Device Virtualization

Jiaqing Huang, Hao Zheng, Zibo Li, Yue Liu

Security Researcher, QI-ANXIN Group

MAIN TRACK

29 AUG

#HITB2024BKK

Who We Are

#HITB2024BKK

Who We Are

Jiaqing Huang

Twitter: @s0duku

Hao Zheng

Twitter: @zhz__6951

Zibo Li

Twitter: @zblee_

Yue Liu

Twitter: @Mr_LiuYue

Who We Are

TianGong Team of Lengendsec at QI-ANXIN Group

- Focuse on vulnerability discovery and exploitation
- Targeting at Edge Devices/ IOT/ OS/ Virtualization/ Browser, etc
- Works published in HITB, BlackHat, EuroS&P, Usenix, ACM CCS, etc
- Awarded in GeekPwn, Tianfu Cup, etc

Twitter: @TianGongLab

WeChat: 奇安信天工实验室

Blog: https://tiangonglab.github.io/blog/

The Virtualization Hacking Journey

#HITB2024BKK

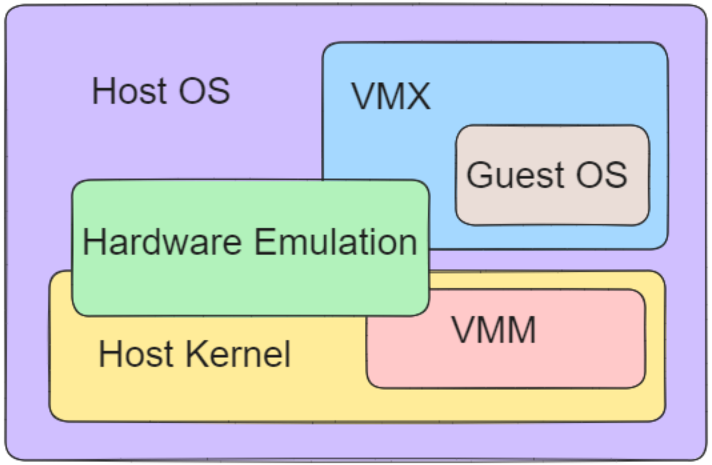
The Virtualization Hacking Journey

Leader: Go pick something you interested, do the long-term research.

Whoa, Windows, Linux, Hardware, etc.
Everything is interesting

The Virtualization Hacking Journey

Wait .....
Virtualization almost involved each of them to some degree



#HITB2024BKK

The Virtualization Hacking Journey

Know nothing about virtualization but decide to challenge the virtual dragon!
Because we want!

#HITB2024BKK

The Virtualization Hacking Journey

- Started to research virtualization security at the end of 2022
- Studied from lots of virtualization related public cases
- Escaped from the Parallels Desktop at GeekCon 2023
- Reported lots of VMware Workstation\ESXi bugs

The Virtualization Hacking Journey

- VMware Hypervisor Reverse Engineering
    - VMware Virtualization Architecture
- VMware Device Virtualization Bug Hunting
    - USB Virtualization Bug Hunting
    - SCSI Virtualization Bug Hunting

VMware Hypervisor Reverse Engineering

#HITB2024BKK

VMware Hypervisor Reverse Engineering

Let's speed up our reverse engineering!

- Debug Tricks
- Dynamic Instrumentation
- Symbol Recovery

I am speed

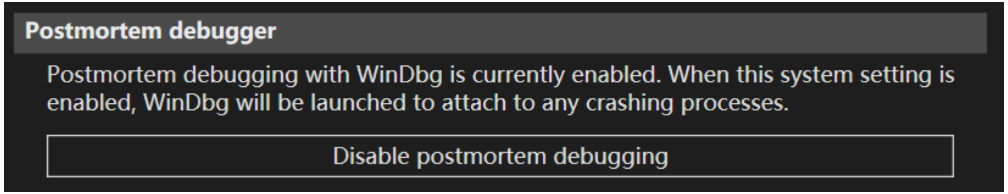
VMware Hypervisor Reverse Engineering

- Recommend to debug vmware-vmx.exe under Windows
- “Image File Execution Options” may encounter some problems.
- Add 0xCC to the vmware-vmx.exe binary

```
.text:00000001400179C0 ; =============== S U B R O U T I N E =======================================
.text:00000001400179C0
.text:00000001400179C0
.text:00000001400179C0                 public start
.text:00000001400179C0 start           proc near               ; DATA XREF: .rdata:000000014093A7B0↓o
.text:00000001400179C0                 int     3               ; Trap to Debugger
.text:00000001400179C1                 sub     esp, 28h
.text:00000001400179C4                 call    sub_140017C24
.text:00000001400179C9                 add     rsp, 28h
.text:00000001400179CD                 jmp     sub_14001784C
.text:00000001400179CD start           endp
.text:00000001400179CD
```

VMware Hypervisor Reverse Engineering

- Enable the WinDbg Postmortem Debugging
- Once you start the Guest Machine, Windbg will auto attach to vmware-vmx.exe
- This helps you debug the vmx initialize process



VMware Hypervisor Reverse Engineering

- Use dynamic binary instrumentation tools (Frida, etc)
- Test function arguments in vmware-vmx.exe process
- Trace code execution flow

VMware Hypervisor Reverse Engineering

- Use open-vm-tools source code to recover the symbols of some common functions
- vmware-vmx-debug.exe contains more log string

```c
int
AsyncSocket_Send(AsyncSocket *asock,          // IN
                 void *buf,                   // IN
                 int len,                     // IN
                 AsyncSocketSendFn sendFn,    // IN
                 void *clientData)            // IN
{
   int ret;
   if (VALID(asock, send)) {
      AsyncSocketLock(asock);
      ret = VT(asock)->send(asock, buf, len, sendFn, clientData);
      AsyncSocketUnlock(asock);
   } else {
      ret = ASOCKERR_INVAL;
   }
   return ret;
}
```

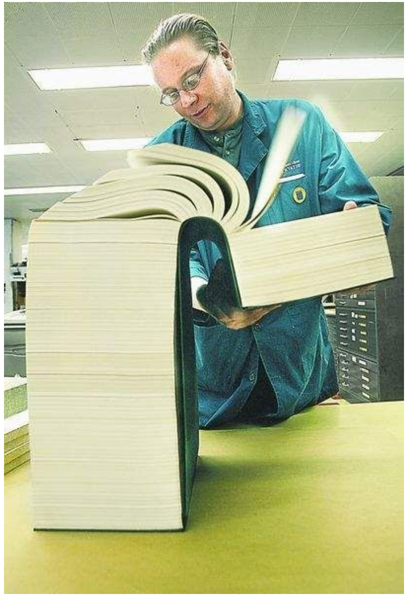
```c
__int64 __fastcall AsyncSocket_Send(
        AsyncSocket *asock,
        __int64 buf,
        unsigned int len,
        __int64 sendFn,
        __int64 clientData)
{
  unsigned int v9; // ebx

  if ( !asock || !asock->vt->send )
    return 5i64;
  AsyncSocketLock(asock);
  v9 = (asock->vt->send)(asock, buf, len, sendFn, clientData);
  AsyncSocketUnlock(asock);
  return v9;
}
```

#HITB2024BKK

VMware Hypervisor Reverse Engineering

- Learn from the internet
  - CVEs
  - Hardware documents
  - Open source code (QEMU, Linux driver, etc)
  - ……



#HITB2024BKK

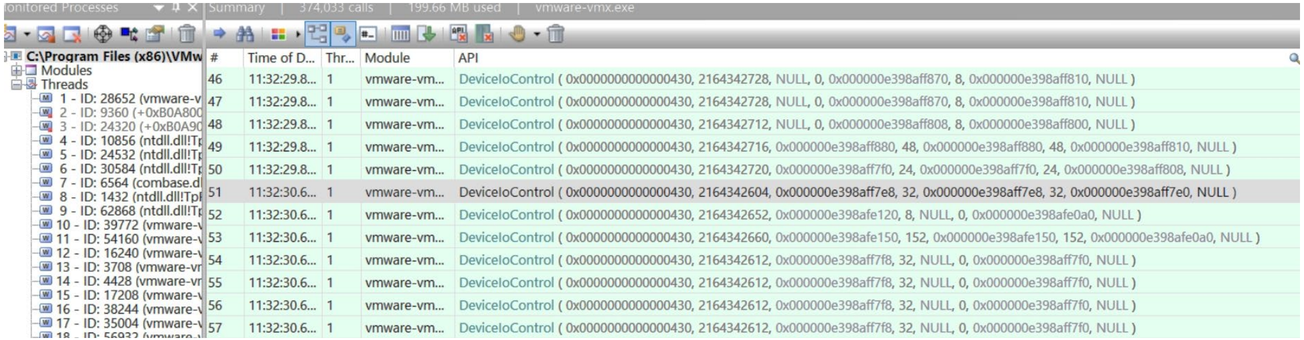
VMware Hypervisor Reverse Engineering

- Well prepared, but where should we actually start?
- Let's locate the "loop" of vmware-vmx.exe first!

#HITB2024BKK

VMware Hypervisor Reverse Engineering

- vmware-vmx.exe is usermode process
- vmx86.sys is responsible for assisting it
- Trace the DeviceIoControl API to see how they communicate with each other



#HITB2024BKK

VMware Hypervisor Reverse Engineering

- Combined with log string to understand the meaning of IOCTL code

```
__int64 __fastcall IOCTL_VMX86_RUN_VM(int a1)
{
  const char *v1; // rax
  DWORD BytesReturned; // [rsp+40h] [rbp-18h] BYREF

  if ( DeviceIoControl(hDevice, 0x81013F67, (LPVOID)(unsigned int)(a1 + 1000), 8u, 0i64, 0, &BytesReturned, 0i64) )
    return BytesReturned;
  v1 = (const char *)W32Util_LastError2Msg();
  Warning_("IOCTL_VMX86_RUN_VM failed: %s\n", v1);
  return 0xFFFFFFFFi64;
}
```

VMware Hypervisor Reverse Engineering

```c
userRpcBlock = SharedArea_Lookup(a1, "userRpcBlock", 0x1088i64);
v3 = (__int64 *)SharedArea_Lookup(v1, "monitorSwitchError", 8i64);
do
{
  Id = IOCTL_VMX86_RUN_VM(v1);                  // 1. ioctl vmx86.sys to switch to vmm
                                                // 2. receive the UserRpcHandler Id
  if ( *(_DWORD *)qword_7FF7CF869570 >= 2u )
    MonitorLogMonitorPanic();
  if ( ((Id + 0x80000000) & 0x80000000) == 0 && Id != -8193 )
  {
    _mm_lfence();
    Panic("VCPU %u RunVM failed: %d.\n", v1, Id);
  }
  v5 = *v3;
  v6 = *v3;
  if ( Id == -8193 && v5 != -1 )
  {
    v6 = sub_7FF7CE3B8120(v1);
    *v3 = v6;
  }
  if ( v6 && v5 != -1 )
  {
    v8 = sub_7FF7CE978910(v6);
    LOBYTE(v9) = 1;
    v10 = (const char *)v8;
    sub_7FF7CE966A00(v9);
    Panic("%s\n", v10);
  }
  if ( *(_DWORD *)qword_7FF7CF869570 >= 2u )
    MonitorLogMonitorPanic();
  result = Monitor_ProcessUserRpcCall((__int64)VMContext, userRpcBlock, Id);// 1. call UserRpcHandler by Id
                                                // 2. call UserRpcHandler with userRpcBlock shared area.
}
while ( Id != 305 );
```

```
.data:00007FF7CF016980                  ; DATA XREF: Monitor_ProcessUserRpcCall+
.data:00007FF7CF016990  UserRpcCallHandler <offset _guard_check_icall_nop, 0>; 1 ; Micro
.data:00007FF7CF0169A0  UserRpcCallHandler <offset sub_7FF7CE981E90, 0>; 2
.data:00007FF7CF0169B0  UserRpcCallHandler <offset sub_7FF7CE981E70, 0>; 3
.data:00007FF7CF0169C0  UserRpcCallHandler <offset sub_7FF7CE981E80, 0>; 4
.data:00007FF7CF0169D0  UserRpcCallHandler <offset sub_7FF7CE981BC0, 0>; 5
.data:00007FF7CF0169E0  UserRpcCallHandler <offset sub_7FF7CE981EA0, 0>; 6
.data:00007FF7CF0169F0  UserRpcCallHandler <offset sub_7FF7CE981EC0, 1>; 7
.data:00007FF7CF016A00  UserRpcCallHandler <offset sub_7FF7CE981AF0, 1>; 8
.data:00007FF7CF016A10  UserRpcCallHandler <offset sub_7FF7CE981B60, 1>; 9
.data:00007FF7CF016A20  UserRpcCallHandler <offset sub_7FF7CE981E00, 1>; 10
.data:00007FF7CF016A30  UserRpcCallHandler <offset j_MonitorLogMonitorPanic, 0>; 11
.data:00007FF7CF016A40  UserRpcCallHandler <offset sub_7FF7CE978CF0, 0>; 12
.data:00007FF7CF016A50  UserRpcCallHandler <offset sub_7FF7CE9666A0, 0>; 13
.data:00007FF7CF016A60  UserRpcCallHandler <offset sub_7FF7CE9666B0, 0>; 14
.data:00007FF7CF016A70  UserRpcCallHandler <offset MonitorLoop_FinalizeHandler, 1>; 15
.data:00007FF7CF016A80  UserRpcCallHandler <offset sub_7FF7CE96D0C0, 1>; 16
.data:00007FF7CF016A90  UserRpcCallHandler <offset sub_7FF7CE965190, 0>; 17
.data:00007FF7CF016AA0  UserRpcCallHandler <offset sub_7FF7CE965900, 1>; 18
.data:00007FF7CF016AB0  UserRpcCallHandler <offset sub_7FF7CE980FD0, 0>; 19
.data:00007FF7CF016AC0  UserRpcCallHandler <offset unknown_libname_17, 0>; 20
.data:00007FF7CF016AD0  UserRpcCallHandler <offset sub_7FF7CE462770, 0>; 21
.data:00007FF7CF016AE0  UserRpcCallHandler <offset sub_7FF7CE45CDD0, 0>; 22
.data:00007FF7CF016AF0  UserRpcCallHandler <offset sub_7FF7CE976F80, 1>; 23
.data:00007FF7CF016B00  UserRpcCallHandler <offset sub_7FF7CE4DAE00, 1>; 24
.data:00007FF7CF016B10  UserRpcCallHandler <offset sub_7FF7CE981F70, 0>; 25
.data:00007FF7CF016B20  UserRpcCallHandler <offset sub_7FF7CE96BFF0, 1>; 26
.data:00007FF7CF016B30  UserRpcCallHandler <offset sub_7FF7CE491C00, 0>; 27
.data:00007FF7CF016B40  UserRpcCallHandler <offset sub_7FF7CE3BB8D0, 1>; 28
.data:00007FF7CF016B50  UserRpcCallHandler <offset sub_7FF7CE4840C0, 1>; 29
.data:00007FF7CF016B60  UserRpcCallHandler <offset sub_7FF7CE4430F0, 1>; 30
.data:00007FF7CF016B70  UserRpcCallHandler <offset sub_7FF7CE4C0340, 1>; 31
.data:00007FF7CF016B80  UserRpcCallHandler <offset Vmxnet3_RPCHandler, 1>; 32
.data:00007FF7CF016B90  UserRpcCallHandler <offset sub_7FF7CE4F3BD0, 1>; 33
.data:00007FF7CF016BA0  UserRpcCallHandler <offset Xhci_RPCHandler, 1>; 34
.data:00007FF7CF016BB0  UserRpcCallHandler <offset PVSCSI_RPCHandler, 0>; 35
```

#HITB2024BKK

VMware Hypervisor Reverse Engineering

- What is UserRPC?
  - A mechanism designed for vmm to interact with vmx
  - Similar to Hypercall, but on userspace vmware-vmx.exe
  - Contains a lot of code related to device emulation
  - lot of bugs that are found in device emulation functions are called from related UserRpcHandler

VMware Hypervisor Reverse Engineering

- UserRPCHandler only took one param: UserRpcBlock pointer
- UserRpcBlock is a SharedArea memory region

```
userRpcBlock = SharedArea_Lookup(a1, "userRpcBlock", 0x1088i64);
v3 = (__int64 *)SharedArea_Lookup(v1, "monitorSwitchError", 8i64);
do
void __fastcall NVME_RPCHandler(__int64 rpcBlock)
{
  NvmeAdapterDeviceObject *adapterObject; // rsi
  unsigned int v3; // r15d
  NvmeShadredObj *NvmeShadredObj; // rbx
  unsigned int BackendSCSIAdapterID; // edi
  VStorageDevicebject *v6; // rax
  VStorageDevicebject *v7; // rbx
  unsigned int v8; // eax
  char *v9; // rax
```

RPC Params

VMware Hypervisor Reverse Engineering

- Reverse engineering the SharedArea module
  - Analyze the creation and loading process of vmm
  - Trace interactions between vmware-vmx.exe and vmx86.sys
  - Pay attention to every memory allocation calls

VMware Hypervisor Reverse Engineering

```
.rdata:00007FF7CEC6AA38                 dq offset aViommuearly  ; "VIOMMUEarly"
.rdata:00007FF7CEC6AA40                 dq offset sub_7FF7CE307D10
.rdata:00007FF7CEC6AA48                 align 10h
.rdata:00007FF7CEC6AA50                 dq offset aSharedarea   ; "SharedArea"
.rdata:00007FF7CEC6AA58                 dq offset SharedArea_PowerOn
.rdata:00007FF7CEC6AA60                 dq offset sub_7FF7CE94FB40
.rdata:00007FF7CEC6AA68                 dq offset aOvhdmem      ; "OvhdMem"
.rdata:00007FF7CEC6AA70                 dq offset sub_7FF7CE3A86A0
.rdata:00007FF7CEC6AA78                 align 20h
.rdata:00007FF7CEC6AA80                 dq offset aDiskOvhd     ; "Disk_Ovhd"
.rdata:00007FF7CEC6AA88                 dq offset sub_7FF7CE307D10
.rdata:00007FF7CEC6AA90                 db    0
.rdata:00007FF7CEC6AA91                 db    0
.rdata:00007FF7CEC6AA92                 db    0
.rdata:00007FF7CEC6AA93                 db    0
.rdata:00007FF7CEC6AA94                 db    0
.rdata:00007FF7CEC6AA95                 db    0
.rdata:00007FF7CEC6AA96                 db    0
.rdata:00007FF7CEC6AA97                 db    0
.rdata:00007FF7CEC6AA98                 dq offset aNvdimm_1     ; "NVDIMM"
.rdata:00007FF7CEC6AAA0                 dq offset sub_7FF7CE495D00
.rdata:00007FF7CEC6AAA8                 dq offset sub_7FF7CE495BF0
.rdata:00007FF7CEC6AAB0                 dq offset aMemschedearly ; "MemSchedEarly"
.rdata:00007FF7CEC6AAB8                 dq offset sub_7FF7CE38AF40
```

```
__int64 LoadVmmBlob()
{
  __int64 result; // rax
  __int64 v1; // rbx
  int v2; // [rsp+20h] [rbp-28h] BYREF
  __int64 v3; // [rsp+28h] [rbp-20h] BYREF

  result = qword_7FF7CF8695C8;
  if ( !qword_7FF7CF8695C8 )
  {
    if ( sub_7FF7CEAD0B10(6014i64, &v3, &v2) )
      v1 = sub_7FF7CEAD0D10(v3, v2);
    else
      v1 = 0i64;
    Loader_SetFilename(v1, "vmmblob.elf");
    result = v1;
    qword_7FF7CF8695C8 = v1;
  }
  return result;
}
```

VMware Hypervisor Reverse Engineering

```
$ readelf -S vmmblob.bin -W
There are 28 section headers, starting at offset 0x565f48:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .gdttask          PROGBITS        ffffffffca02000 0091e0 001000 00   A  0   0 32
  [ 2] .monstack         PROGBITS        ffffffffc400000 0001e0 008000 00   A  0   0 32
  [ 3] .idt              PROGBITS        ffffffffc408000 0081e0 001000 00  WA  0   0 32
  [ 4] .shared_per_vcpu  NOBITS          fffffffffe8d9000 02a000 0006e0 00  WA  0   0  1
  [ 5] .shared_per_vcpu_vmx NOBITS          fffffffffe8e1000 02a000 00630c 00  WA  0   0  1
  [ 6] .shared_per_vm_vmx NOBITS          fffffffffe7ad000 02a000 038a00 00  WA  0   0  1
  [ 7] .text             PROGBITS        fffffffffe000000 00b000 00a000 00  AX  0   0 4096
  [ 8] .rodata           PROGBITS        fffffffffe011000 01c000 002000 00   A  0   0 4096
  [ 9] .monLoaderHeader  PROGBITS        fffffffffe02f000 027000 001000 00   A  0   0 4096
  [10] .idtstubs         PROGBITS        fffffffffe7ab000 028000 002000 00  AX  0   0 4096
  [11] .data             PROGBITS        fffffffffe01b000 026000 000854 00  WA  0   0 32
  [12] .bss              NOBITS          fffffffffe025000 026854 0005e8 00  WA  0   0 32
  [13] vmmmods           PROGBITS        fffffffffe969000 02a000 363000 00  WA  0   0 4096
  [14] .host_params      PROGBITS        fffffffffeccc000 38d000 000098 00  WA  0   0 32
  [15] .debug_info       PROGBITS        0000000000000000 38d098 13e06a 00      0   0  1
  [16] .debug_abbrev     PROGBITS        0000000000000000 4cb102 007814 00      0   0  1
  [17] .debug_loc        PROGBITS        0000000000000000 4d2916 020a7f 00      0   0  1
  [18] .debug_aranges    PROGBITS        0000000000000000 4f33a0 001100 00      0   0 16
  [19] .debug_ranges     PROGBITS        0000000000000000 4f44a0 004cf0 00      0   0 16
  [20] .debug_line       PROGBITS        0000000000000000 4f9190 00d731 00      0   0  1
  [21] .debug_str        PROGBITS        0000000000000000 5068c1 0572c5 01  MS  0   0  1
  [22] .comment          PROGBITS        0000000000000000 55db86 000011 01  MS  0   0  1
  [23] .debug_frame      PROGBITS        0000000000000000 55db98 000058 00      0   0  8
  [24] .debug_pubnames   PROGBITS        0000000000000000 55dbf0 000023 00      0   0  1
  [25] .symtab           SYMTAB          0000000000000000 55dc18 005130 18     26 126  8
  [26] .strtab           STRTAB          0000000000000000 562d48 0030cc 00      0   0  1
  [27] .shstrtab         STRTAB          0000000000000000 565e14 000134 00      0   0  1
```

```
root@debian:~# readelf -W -S vmmblob175-vmmmods.bin
There are 24 section headers, starting at offset 0x327990:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] vmm.vmm           PROGBITS        0000000000000000 000040 1c54d0 00      0   0  1
  [ 2] vrdma-vrdma.vmm   PROGBITS        0000000000000000 1c5510 001a38 00      0   0  1
  [ 3] vprobe-vprobe.vmm PROGBITS        0000000000000000 1c6f48 021690 00      0   0  1
  [ 4] vprobe-none.vmm   PROGBITS        0000000000000000 1e85d8 001170 00      0   0  1
  [ 5] vmxnet3-vmxnet3.vmm PROGBITS        0000000000000000 1e9748 002050 00      0   0  1
  [ 6] vmce-vmce.vmm     PROGBITS        0000000000000000 1eb798 002b10 00      0   0  1
  [ 7] vmce-none.vmm     PROGBITS        0000000000000000 1ee2a8 0013f8 00      0   0  1
  [ 8] viommu-vvtd.vmm   PROGBITS        0000000000000000 1ef6a0 00c200 00      0   0  1
  [ 9] viommu-none.vmm   PROGBITS        0000000000000000 1fb8a0 001360 00      0   0  1
  [10] viommu-amd.vmm    PROGBITS        0000000000000000 1fcc00 009e98 00      0   0  1
  [11] qat-qat.vmm       PROGBITS        0000000000000000 206a98 0024b0 00      0   0  1
  [12] pvscsi-pvscsi.vmm PROGBITS        0000000000000000 208f48 0031e0 00      0   0  1
  [13] pcip-pcip.vmm     PROGBITS        0000000000000000 20c128 00a5c8 00      0   0  1
  [14] nvme-nvme.vmm     PROGBITS        0000000000000000 2166f0 004778 00      0   0  1
  [15] hv-vt.vmm         PROGBITS        0000000000000000 21ae68 078440 00      0   0  1
  [16] hv-svm.vmm        PROGBITS        0000000000000000 2932a8 063b08 00      0   0  1
  [17] gphys-npt.vmm     PROGBITS        0000000000000000 2f6db0 0121a8 00      0   0  1
  [18] gphys-ept.vmm     PROGBITS        0000000000000000 308f58 012e10 00      0   0  1
  [19] callstack-none.vmm PROGBITS        0000000000000000 31bd68 000f88 00      0   0  1
  [20] callstack-callstack.vmm PROGBITS        0000000000000000 31ccf0 000f88 00      0   0  1
  [21] buslogic-buslogic.vmm PROGBITS        0000000000000000 31dc78 001728 00      0   0  1
  [22] ahci-ahci.vmm     PROGBITS        0000000000000000 31f3a0 008490 00      0   0  1
  [23] .shstrtab         STRTAB          0000000000000000 327830 00015e 00      0   0  1
```

#HITB2024BKK

VMware Hypervisor Reverse Engineering

- vmmblob.elf contains lots of symbols
- Symbols shared with vmware-vmx.exe and vmx86.sys
- Speed up our work again

```
LinkerShouldAlloc
LinkerCacheSectionData
LinkerCreateObjectFile
LinkerApplyRelocations
LinkerLoadSection
Linker_SharedInterVcpuVmxSize
Linker_SharedInterVcpuSize
Linker_SharedPerVcpuSize
Linker_SharedPerVmSize
Linker_DefineCustomAbsoluteSymbol
Linker_DefineCustomRelativeSymbol
Linker_AddFile
Linker_AddToSection
Linker_FindSection
Linker_SkipSection
Linker_FileSectionVA_cold
Linker_FileSectionVA
Linker_Link_cold
Linker_Link
Linker_CreateHandle
Linker_CreateEmptyHandle
Linker_Close
Linker_NumSections
Linker_SectionName
Linker_SectionVA
Linker_SectionSize
Linker_EntryPoint
Linker_LoadSection
LookupGlobalWork
InitGlobalHash
```

#HITB2024BKK

VMware Hypervisor Reverse Engineering

- ELF linker code within vmware-vmx.exe
- vmm extensions stored in vmmblob's sections in format of ELF Object
- vmmblob and vmm extensions will be relinked to a new ELF for vmm in memory arcording to ".vmx" configuration

VMware Hypervisor Reverse Engineering

- Found "userRpcBlock" as predefined export symbols in .shared_per_vcpu_vmx section of the vmmblob for SharedArea
- Some virtual device implementations will define the export symbol in it too

```
                align 20h
                public userRpcBlock
userRpcBlock    db    ? ;

                db    ? ;
                db    ? ;
```

```
if ( dword_7FF7CF64B43C )
{                                              // add shared area symbol to vmm
  do
  {
    Linker_DefineSymbol(
      v1,
      "{SharedAreaReservations}",
      ".shared_per_vm_vmx",
      &aAhcishared_0[44 * v0],
      *&aAhcishared_0[44 * v0 + 40]);
    ++v0;
  }
  while ( v0 < dword_7FF7CF64B43C );
}
Linker_Link(v1);                               // link vmm in the memory
```

VMware Hypervisor Reverse Engineering

- vmx calculates the total size of SharedArea memories and allocates memory

```
if ( (unsigned __int8)SharedArea_PowerOnInt() )
{
  byte_141341128 = 0;
  sharedAreaBase = Mem_Map(
                     49i64,
                     (unsigned int)(dword_1415B8CF4
                                  + dword_1413410CC
                                  + *((_DWORD *)qword_141384470 + 44)
                                  * (dword_1413410F4 + dword_1413410A4 + dword_14134111C)),
                     0i64);
  if ( sharedAreaBase )
    goto LABEL_3;
  sub_1406AF0A0(3i64, "@&!*@*@(msg.sharedArea.noMappedMem)Failed to allocate shared memory.\n");
```

VMware Hypervisor Reverse Engineering

- .host_params section of vmmblob contains vmm's GDT information

```
.host_params:FFFFFFFFFEC8D048                    dw 0E9h              ; gdtInit.entries.index
.host_params:FFFFFFFFFEC8D04A                    db 2 dup(0)
.host_params:FFFFFFFFFEC8D04C                    dd 0                 ; gdtInit.entries.base
.host_params:FFFFFFFFFEC8D050                    dd 0FFFFFh           ; gdtInit.entries.limit
.host_params:FFFFFFFFFEC8D054                    dd 0Ah               ; gdtInit.entries.type
.host_params:FFFFFFFFFEC8D058                    dd 1                 ; gdtInit.entries.S
.host_params:FFFFFFFFFEC8D05C                    dd 0                 ; gdtInit.entries.DPL
.host_params:FFFFFFFFFEC8D060                    dd 1                 ; gdtInit.entries.present
.host_params:FFFFFFFFFEC8D064                    dd 1                 ; gdtInit.entries.longmode
.host_params:FFFFFFFFFEC8D068                    dd 0                 ; gdtInit.entries.DB
.host_params:FFFFFFFFFEC8D06C                    dd 1                 ; gdtInit.entries.gran
.host_params:FFFFFFFFFEC8D070                    dw 0EAh              ; gdtInit.entries.index
.host_params:FFFFFFFFFEC8D072                    db 2 dup(0)
.host_params:FFFFFFFFFEC8D074                    dd 0                 ; gdtInit.entries.base
.host_params:FFFFFFFFFEC8D078                    dd 0FFFFFh           ; gdtInit.entries.limit
.host_params:FFFFFFFFFEC8D07C                    dd 2                 ; gdtInit.entries.type
.host_params:FFFFFFFFFEC8D080                    dd 1                 ; gdtInit.entries.S
.host_params:FFFFFFFFFEC8D084                    dd 0                 ; gdtInit.entries.DPL
.host_params:FFFFFFFFFEC8D088                    dd 1                 ; gdtInit.entries.present
.host_params:FFFFFFFFFEC8D08C                    dd 0                 ; gdtInit.entries.longmode
.host_params:FFFFFFFFFEC8D090                    dd 1                 ; gdtInit.entries.DB
.host_params:FFFFFFFFFEC8D094                    dd 1                 ; gdtInit.entries.gran
.host_params:FFFFFFFFFEC8D094 _host_params       ends
```

#HITB2024BKK

VMware Hypervisor Reverse Engineering

- .monloader section of vmmblob contains vmm's virtual address mapping information

```
.monLoaderHeader:FFFFFFFFDE2F000 ; const MonLoaderHeader monLoaderHeader
.monLoaderHeader:FFFFFFFFDE2F000 monLoaderHeader dq 8675309E98675309h    ; magic
.monLoaderHeader:FFFFFFFFDE2F000                                         ; DATA XREF: _start+55↑r
.monLoaderHeader:FFFFFFFFDE2F000                                         ; _start+BD↑r
.monLoaderHeader:FFFFFFFFDE2F008                 dd 48h                  ; entrySize
.monLoaderHeader:FFFFFFFFDE2F00C                 dd 17h                  ; count
.monLoaderHeader:FFFFFFFFDE2F010                 dw 748h                 ; codeSelector
.monLoaderHeader:FFFFFFFFDE2F012                 dq 0FFFFFFFFFDE00000h   ; codeEntrypoint
.monLoaderHeader:FFFFFFFFDE2F01A                 dw 750h                 ; stackSelector
.monLoaderHeader:FFFFFFFFDE2F01C                 dq 0FFFFFFFFFC408000h   ; stackEntrypoint
.monLoaderHeader:FFFFFFFFDE2F024                 dq 0FFFFFFFFFC000h      ; monStartLPN
.monLoaderHeader:FFFFFFFFDE2F02C                 dq 0FFFFFFFFFFFFFh      ; monEndLPN
.monLoaderHeader:FFFFFFFFDE2F034 ; const MonLoaderEntry stru_FFFFFFFFFDE2F034
.monLoaderHeader:FFFFFFFFDE2F034 stru_FFFFFFFFFDE2F034 dd ML_CONTENT_ADDRSPACE ; [0].content
.monLoaderHeader:FFFFFFFFDE2F034                                         ; DATA XREF: _start+A3↑r
.monLoaderHeader:FFFFFFFFDE2F038                 dd ML_SOURCE_NONE       ; [0].source
.monLoaderHeader:FFFFFFFFDE2F03C                 dq 0FFFFFFFFFC000h      ; [0].monVPN
.monLoaderHeader:FFFFFFFFDE2F044                 dq 4000h                ; [0].monPages
.monLoaderHeader:FFFFFFFFDE2F04C                 dq 3                    ; [0].flags
.monLoaderHeader:FFFFFFFFDE2F054                 dd 0                    ; [0].allocs
.monLoaderHeader:FFFFFFFFDE2F058                 db 4 dup(0)             ; 0
.monLoaderHeader:FFFFFFFFDE2F05C                 dq 0                    ; [0].blobSrc.offset
.monLoaderHeader:FFFFFFFFDE2F064                 dq 0                    ; [0].blobSrc.size
.monLoaderHeader:FFFFFFFFDE2F06C                 dq 0                    ; [0].bspOnly
.monLoaderHeader:FFFFFFFFDE2F074                 dq 0                    ; [0].subIndex
.monLoaderHeader:FFFFFFFFDE2F07C                 dd ML_CONTENT_PAGETABLE_L4; [1].content
.monLoaderHeader:FFFFFFFFDE2F080                 dd ML_SOURCE_NONE       ; [1].source
.monLoaderHeader:FFFFFFFFDE2F084                 dq 0FFFFFFFFFCA04h      ; [1].monVPN
.monLoaderHeader:FFFFFFFFDE2F08C                 dq 1                    ; [1].monPages
.monLoaderHeader:FFFFFFFFDE2F094                 dq 8000000000000003h    ; [1].flags
.monLoaderHeader:FFFFFFFFDE2F09C                 dd 0                    ; [1].allocs
.monLoaderHeader:FFFFFFFFDE2F0A0                 db 4 dup(0)             ; 1
.monLoaderHeader:FFFFFFFFDE2F0A4                 dq 0                    ; [1].blobSrc.offset
.monLoaderHeader:FFFFFFFFDE2F0AC                 dq 0                    ; [1].blobSrc.size
.monLoaderHeader:FFFFFFFFDE2F0B4                 dq 0                    ; [1].bspOnly
.monLoaderHeader:FFFFFFFFDE2F0BC                 dq 0                    ; [1].subIndex
```

#HITB2024BKK

VMware Hypervisor Reverse Engineering

- vmx is responsible for allocating memory and building page table structures based on vmmblob's information
- vmx86.sys further populates the page table information and loads the vmm ELF file constructed by vmx
- vmx, vmmblob, vmx86.sys work together to build the vmm's enviroment, mapping the host allocated address to vmm's virtual address

#HITB2024BKK

VMware Hypervisor Reverse Engineering

- We also need to figure out how vmm switch in/ out works if we want to understand how vmx and vmm interact with each other
- CrossPage is responsible for storing context between vmm and the host, like VMCS
- Mapped to the virtual page 0xFFFFFFFFFCA00 of vmm

```
.monLoaderHeader:FFFFFFFFFDE2F2BC ; const MonLoaderEntry
.monLoaderHeader:FFFFFFFFFDE2F2BC                 dd ML_CONTENT_SHARE      ; content
.monLoaderHeader:FFFFFFFFFDE2F2C0                 dd MonLoaderSourceType::ML_SOURCE_HOST; source
.monLoaderHeader:FFFFFFFFFDE2F2C4                 dq 0FFFFFFFFFCA00h       ; monVPN
.monLoaderHeader:FFFFFFFFFDE2F2CC                 dq 1                     ; monPages
.monLoaderHeader:FFFFFFFFFDE2F2D4                 dq 8000000000000003h     ; flags
.monLoaderHeader:FFFFFFFFFDE2F2DC                 dd 0                     ; allocs
.monLoaderHeader:FFFFFFFFFDE2F2E0                 db 4 dup(0)
.monLoaderHeader:FFFFFFFFFDE2F2E4                 dq 0                     ; blobSrc.offset
.monLoaderHeader:FFFFFFFFFDE2F2EC                 dq 0                     ; blobSrc.size
.monLoaderHeader:FFFFFFFFFDE2F2F4                 dq 0                     ; bspOnly
.monLoaderHeader:FFFFFFFFFDE2F2FC                 dq 9                     ; subIndex
```

VMware Hypervisor Reverse Engineering

- We can search special register operation (like cr3) in vmx86.sys to locate key code
- The host is responsible for saving the current CPU state to CrossPage, including system-level context such as the cr3 register

```
HostSwitchToVmm:                        ; CODE XREF: Task_Switch+59C↓p
                push    qword ptr [rsp+0] ; save Host cs reg and call's return address to stack, than push them to CrossPage.
                                        ; VMM can use CrossPage and retfq back to Host
                mov     word ptr [rsp+8], cs
                mov     rax, rsp
                lea     rsp, [rcx+162h]
                mov     dx, ss
                push    dx
                push    rax
                push    r15
                push    r14
                push    r13
                push    r12
                push    rdi
                push    rsi
                push    rbp
                push    rbx
                mov     rax, cr3        ; host cr3
                push    rax
                lea     rsp, [rcx+270h]
                pop     rsi             ; root mpn
                pop     rbx
                pop     rbp
                pop     r12
                pop     r13
                pop     r14
                pop     r15
                pop     rax
                pop     dx
                lgdt    fword ptr [rcx+338h] ; rcx point to crosspage
                mov     cr3, rsi
                mov     ds, edx
                mov     es, edx
                mov     ss, edx
                mov     rsp, rax
                retfq                   ; Jmp [rsp], rsp += 8, mov cs, [rsp], rsp += 8
; ---------------------------------------------------------------------------
                align 10h
```

Structures

```
00000263                    db ? ; unde
00000264                    db ? ; unde
00000265                    db ? ; unde
00000266                    db ? ; unde
00000267                    db ? ; unde
00000268                    db ? ; unde
00000269                    db ? ; unde
0000026A                    db ? ; unde
0000026B                    db ? ; unde
0000026C                    db ? ; unde
0000026D                    db ? ; unde
0000026E                    db ? ; unde
0000026F                    db ? ; unde
00000270 page_table_root_mpn dq ?
00000270
00000278 switch_to_vmm_rbx dq ?
00000280 switch_to_vmm_rbp dq ?
00000288 switch_to_vmm_r12 dq ?
00000290 switch_to_vmm_r13 dq ?
00000298 switch_to_vmm_r14 dq ?
000002A0 switch_to_vmm_r15 dq ?
000002A8 switch_to_vmm_rsp dq ?
000002B0 switch_to_vmm_dx dq ?
000002B8 CrossPagePhyMachineAddress
000002B8
000002C0                    db ? ; unde
000002C1                    db ? ; unde
000002C2                    db ? ; unde
000002C3                    db ? ; unde
000002C4                    db ? ; unde
000002C5                    db ? ; unde
```

VMware Hypervisor Reverse Engineering

- UserRpc is implemented through PlatformUserCall in vmm
- Saves the opcode to the address 0xFFFFFFFFFCA00550
- Place the PlatformCall invocation number 100 at 0xFFFFFFFFFCA00428
- These addresses are actually offsets within CrossPage

```
int __fastcall PlatformUserCall(UserCallOperation op)
{
  int result; // eax

  MEMORY[0xFFFFFFFFFCA00550] = op;
  MEMORY[0xFFFFFFFFFCA00428] = 100;
  BackToHost();
  result = MEMORY[0xFFFFFFFFFCA0042C];
  MEMORY[0xFFFFFFFFFCA00550] = 300;
  return result;
}
```

VMware Hypervisor Reverse Engineering

- PlatformCall 100 causes vmx86.sys to return the opcode saved at CrossPage offset 0x550 to vmware-vmx.exe
- vmware-vmx.exe calls the corresponding UserRpcHandler based on this opcode number
- UserRpcBlock, it is precisely the content saved by vmm via SharedArea, in the direct memory mapping between the host and vmm memory

```
v23 = v39;
LODWORD(userRpcBlock[1]) = 65280;
HIDWORD(userRpcBlock[1]) = v23;
LODWORD(userRpcBlock[2]) = v40;
HIDWORD(userRpcBlock[2]) = a4;
userRpcBlock[4] = v43;
UserRPC(334);
```

VMware Hypervisor Reverse Engineering

- vmm resolves the x86 IO instructions, and may attempt to use UserRPC(317) to call vmx to process

```
    {
      v31 = 0;
      if ( a4 == 1 )
        v31 = *a8;
    }
    HIDWORD(userRpcBlock[7]) = v31;
  }
  UserRPC(317);
  iospaceCurrentBA = 0LL;
  if ( (a3 & 2) != 0 )
  {
    v32 = userRpcBlock[1];
    switch ( a4 )
    {
      case 4:
        *(_DWORD *)a8 = v32;
        break;
      case 2:
        *(_WORD *)a8 = v32;
        break;
      case 1:
        *a8 = v32;
        break;
    }
  }
}
```

#HITB2024BKK

VMware Hypervisor Reverse Engineering

- Part of port IO callbacks are registered in usermode vmware-vmx.exe
- UserRPC(317) Handler responsible for calling corresponding the port IO callback

```
for ( i = 0; v17 < v6; v17 += InputOutputSize * v19 )
{
  v19 = ((__int64 (__fastcall *)(IoUserCallback *, _QWORD, _QWORD, _QWORD, int, char *))ioPortCallbackFunction->ioPortCallback)(
          ioPortCallbackFunction->ExtArg,
          IoPort_,
          (unsigned int)(repTimes - i),
          (unsigned int)rpcBlock->InputOutputSize,
          rpcFlag,
          &v8[v17]);
```

#HITB2024BKK

VMware Hypervisor Reverse Engineering

- Some devices implement their IOCallback in vmm, not in vmx

```
.rodata:00000000000D1C80 iospaceCBs      dq offset Vmxnet3_IODataHandler
.rodata:00000000000D1C80                                         ; DATA XREF: MonC
.rodata:00000000000D1C88                 dq offset BusMemBalloon_BackdoorPort
.rodata:00000000000D1C90                 dq offset PortF0h_Handler
.rodata:00000000000D1C98                 dq offset Port92h_Handler
.rodata:00000000000D1CA0                 db    0
.rodata:00000000000D1CA1                 db    0
.rodata:00000000000D1CA2                 db    0
.rodata:00000000000D1CA3                 db    0
.rodata:00000000000D1CA4                 db    0
.rodata:00000000000D1CA5                 db    0
.rodata:00000000000D1CA6                 db    0
.rodata:00000000000D1CA7                 db    0
.rodata:00000000000D1CA8                 dq offset PIC_CmdPort
.rodata:00000000000D1CB0                 dq offset PIC_MaskPort
.rodata:00000000000D1CB8                 dq offset PIC_TriggerPort
.rodata:00000000000D1CC0                 dq offset Backdoor_PortMon
.rodata:00000000000D1CC8                 dq offset Vmxnet3_IODataHandler
.rodata:00000000000D1CD0                 dq offset Vmxnet3_IODataHandler
.rodata:00000000000D1CD8                 dq offset Vmxnet3_IODataHandler
.rodata:00000000000D1CE0                 dq offset Vmxnet3_IODataHandler
.rodata:00000000000D1CE8                 dq offset Vmxnet3_IODataHandler
.rodata:00000000000D1CF0                 dq offset CMOS_AddrPort
.rodata:00000000000D1CF8                 dq offset CMOS_ValPort
.rodata:00000000000D1D00                 dq offset E1000_IoDataHandler
.rodata:00000000000D1D08                 dq offset LSILogic_CommonIOHandler
.rodata:00000000000D1D10                 dq offset Vmxnet3_IODataHandler
.rodata:00000000000D1D18                 dq offset PCI_ConfData
```

VMware Hypervisor Reverse Engineering

- For memory-mapped I/O (MMIO), in most cases, vmx associates the memory regions with a specific ID, linking them to corresponding MemHandler functions in vmm by default

```
if ( (*(_DWORD *)(qword_7FF7CF10D180 + 416) & 0x100) == 0
  || (v3 = AllocMem(v2, 1u, (__int64)"ControlBar", 18, (__int64)sub_7FF7CE379640, v9, 0xA6u),
      *(_DWORD *)(v0 + 144) = v3,
      v3 < 0x3CE) )
{
```

```
.rodata:00000000000D1B80 physMemIOCBs    dq offset Vmxnet3_MemHandler; 0
.rodata:00000000000D1B80                                 ; DATA XREF: MonCB_GetPhysMemIOFunc+7
.rodata:00000000000D1B88                 dq offset APIC_RegisterAccess; 1
.rodata:00000000000D1B90                 dq offset IOAPIC_RegisterAccess; 2
.rodata:00000000000D1B98                 dq offset PhysMem_IOUserCallback; 3
.rodata:00000000000D1BA0                 dq offset Vmxnet3_MemHandler; 4
.rodata:00000000000D1BA8                 dq offset E1000_MemMapHandler; 5
.rodata:00000000000D1BB0                 dq offset Ehci_MemMapHandler; 6
.rodata:00000000000D1BB8                 dq offset HDAudio_MemMapHandler; 7
.rodata:00000000000D1BC0                 dq offset HPET_MemHandler; 8
.rodata:00000000000D1BC8                 dq offset LSILogic_MemoryMappedHandler; 9
.rodata:00000000000D1BD0                 dq offset NVDIMM_FlushHandler; 10
.rodata:00000000000D1BD8                 dq offset Vmxnet3_MemHandler; 11
.rodata:00000000000D1BE0                 dq offset PCIe_MMIO     ; 12
.rodata:00000000000D1BE8                 dq 5 dup(offset Vmxnet3_MemHandler); 13
.rodata:00000000000D1C10                 dq offset SVGA_ControlBarMemRef; 18
.rodata:00000000000D1C18                 dq offset SVGA_RegsBarMemRef; 19
.rodata:00000000000D1C20                 dq offset VMCI_RegMemHandler; 20
.rodata:00000000000D1C28                 dq offset VGA_MemRef    ; 21
.rodata:00000000000D1C30                 dq 8 dup(offset Vmxnet3_MemHandler); 22
.rodata:00000000000D1C70                 dq offset Xhci_MemMapHandler; 30
```

#HITB2024BKK

VMware Hypervisor Reverse Engineering

- Most MMIO will access the SharedArea in vmm to interact with vmx

```
__int64 __fastcall SVGAWriteCommandReg(int a1, int a2)
{
  __int64 result; // rax
  bool v3; // dl
  _BOOL4 v4; // ebp
  unsigned __int64 v5; // r12
  int v6; // ebx
  unsigned __int64 v7; // rsi
  __int64 v8; // rdi
  __int64 v9; // rax
  unsigned int v10; // ebx
  unsigned __int8 (__fastcall *i)(__int64, unsigned __int64, char *); // rcx
  __int64 v12; // rax
  __int64 v13; // rax
  int v14; // [rsp+0h] [rbp-98h] BYREF
  unsigned int v15; // [rsp+4h] [rbp-94h] BYREF
  char v16[40]; // [rsp+8h] [rbp-90h] BYREF
  __int64 v17; // [rsp+30h] [rbp-68h]
  int v18[5]; // [rsp+40h] [rbp-58h] BYREF
  unsigned int v19; // [rsp+54h] [rbp-44h]
  unsigned __int64 v20; // [rsp+58h] [rbp-40h]

  result = *(_DWORD *)(&svgaStruct + 30) & 0x1000000;
  v3 = 0;
  if ( *((_DWORD *)&svgaStruct + 44) )
    v3 = *((_BYTE *)&svgaStruct + 2069) != 0;
  v4 = v3;
  if ( (_DWORD)result )
  {
    if ( a1 == 49 )
    {
      *((_DWORD *)&svgaStruct + 92) = a2;
      return result;
    }
    *((_DWORD *)&svgaStruct + 91) = a2;
    v5 = a2 & 0xFFFFFFC0 | ((unsigned __int64)*((unsigned int *)&svgaStruct + 92) << 32);
    result = PhysMem_ValidatePARange(v5, 64LL);
    if ( (_BYTE)result )
    {
      if ( !v4 )
```

SharedArea

#HITB2024BKK

VMware Hypervisor Reverse Engineering

- Most MMIO operations ultimately still rely on UserRpc to call the relevant processing routines in vmx
- That is why we say UserRPC handle lots of device emulation

```
__int64 __fastcall PVSCSIProcessRing_part_0(Pvscsi_SharedObj *a1)
{
  int v1; // esi
  int v2; // edx
  __int64 result; // rax

  if ( !a1->field_4B3 )
    return PVSCSICmdHandleUserLand((__int64)a1, 4);
  v1 = 1 << (LOBYTE(a1->field_4AC) + 9);
  do
  {
    v2 = deviceThread;
    result = (unsigned int)_InterlockedCompareExchange(
                             (volatile signed __int32 *)&deviceThread,
                             deviceThread | v1,
                             deviceThread);
  }
  while ( v2 != (_DWORD)result );
  if ( !v2 )
    return MX_BinSemaphoreSignal((char *)&deviceThread + 8);
  return result;
}

__int64 __fastcall PVSCSICmdHandleUserLand(__int64 a1, int a2)
{
  int v2; // eax
  __int64 v4; // [rsp+0h] [rbp-58h] BYREF

  UserRPCSave(&v4, 48LL);
  v2 = *(_DWORD *)(a1 + 1196);
  userRpcBlock[3] = a2;
  userRpcBlock[2] = v2;
  UserRPC(335LL);
  return UserRPCRestore();
}
```

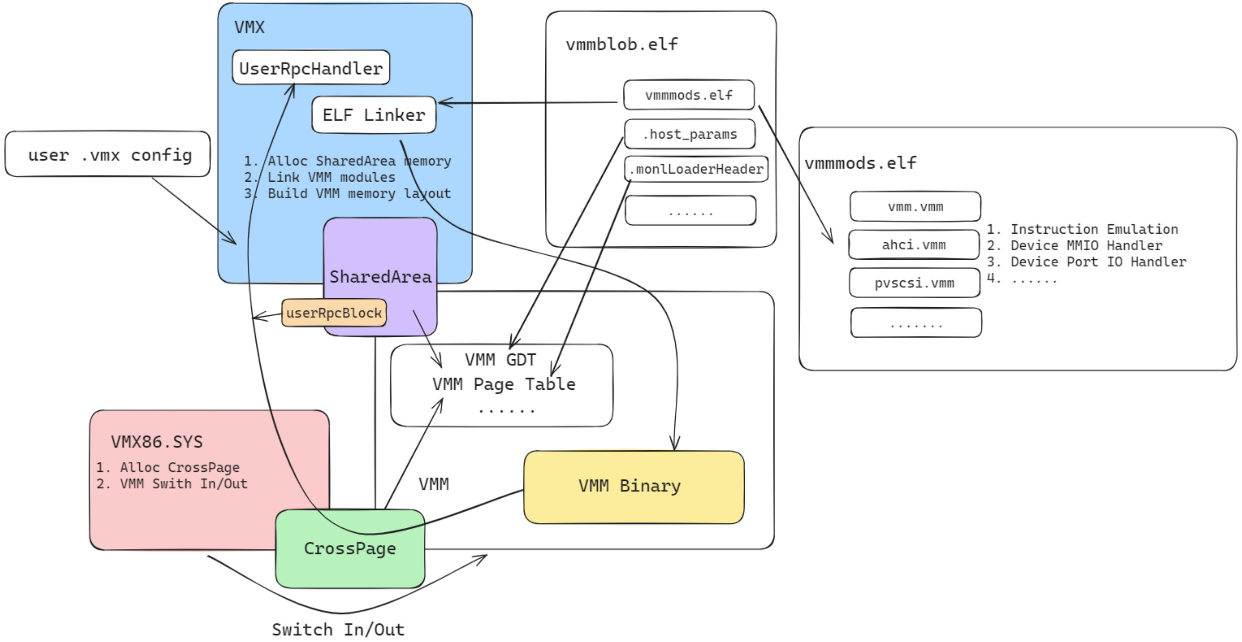
VMware Hypervisor Reverse Engineering

- The representation object of guest physical memory is obtained based on the physical address
- Depending on the object's type, direct memory access within vmx is usually used

```
if ( PhysMem_ValidateAndGet(phyAddr, pageSize, 1u, 9u, &PhyMemContent) )
{
  LODWORD(RingPointerPA) = ConsumerRing->RingPointerPA;
  enqueuePtr = ConsumerRing->enqueuePtr;
  while ( 1 )
  {
    v10 = (unsigned int)(RingPointerPA - phyAddr);
    v11 = &ConsumerRing->TrbRingQueue[enqueuePtr];
    v12 = (unsigned int)(v10 + 12);
    if ( PhyMemContent.type == 1 )
    {
      v13 = *(_DWORD *)(PhyMemContent.contentHostVA + v12);
    }
    else
    {
      PhysMemReadSlow(&PhyMemContent, v12, 4ui64, (char *)&v22);
      v13 = v22;
    }
```

#HITB2024BKK

VMware Hypervisor Reverse Engineering



VMware Hypervisor Reverse Engineering

- In ESXi, some devices' MMIO will not always call UserRPC
- Some devices will call UserRPC only when "hostedEmulation" is enabled

```
__int64 __fastcall PVSCSIProcessRing(__int64 a1, char a2)
{
  __int64 result; // rax
  int v3; // esi
  int v4; // edx

  if ( *(_BYTE *)(a1 + 2344) )
  {
    if ( *(_BYTE *)(a1 + 0x929) )
    {
      if ( *(_BYTE *)(a1 + 2347) )
      {
        v3 = 1 << (*(_DWORD *)(a1 + 1196) + 9);
        do
        {
          v4 = deviceThread;
          result = (unsigned int)_InterlockedCompareExchange(
                                   (volatile signed __int32 *)&deviceThread,
                                   deviceThread | v3,
                                   deviceThread);
        }
        while ( v4 != (_DWORD)result );
        if ( !v4 )
          return MX_BinSemaphoreSignal((char *)&deviceThread + 8);
      }
      else
      {
        return PVSCSICmdHandleUserLand(a1, 4);
      }
    }
    else
    {
      return PVSCSI_VMKProcessRing(a1, a2);
    }
  }
  return result;
}
```

#HITB2024BKK

VMware Hypervisor Reverse Engineering

- vmkcall - vmm direct call to VMKernel to handle devices emulation

```
__int64 __fastcall PVSCSI_VMKProcessRing(__int64 a1, char a2)
{
  bool v2; // zf
  __int64 v3; // rsi

  v2 = a2 == 0;
  v3 = *(unsigned int *)(a1 + 1196);
  *(_BYTE *)(a1 + 0x8B8) = !v2;
  return VMK_Call_1Args(0x7CLL, v3);
}
```

vmkFuncTable

```
.rodata:00004200007D3AB0                 dq offset Net_VMMVlanceUpdateMAC; 118
.rodata:00004200007D3AB8                 dq offset Net_VMMVmxnetUpdateEthFRP; 119
.rodata:00004200007D3AC0                 dq offset VSCSI_ExecuteCommand; 120
.rodata:00004200007D3AC8                 dq offset VSCSI_CmdComplete; 121
.rodata:00004200007D3AD0                 dq offset VSCSI_AccumulateSG; 122
.rodata:00004200007D3AD8                 dq offset VSCSI_FreeSG  ; 123
.rodata:00004200007D3AE0                 dq offset VSCSI_MapMPN  ; 124
.rodata:00004200007D3AE8                 dq offset LSI_InitRings ; 125
.rodata:00004200007D3AF0                 dq offset LSI_ProcessReq; 126
.rodata:00004200007D3AF8                 dq offset LSI_ActivatePoll; 127
.rodata:00004200007D3B00                 dq offset LSI_ProcessCompl; 128
.rodata:00004200007D3B08                 dq offset VSCSI_ChangeCompletionMode; 129
.rodata:00004200007D3B10                 dq offset PVSCSI_AdapterInit; 130
.rodata:00004200007D3B18                 dq offset PVSCSI_FlushIotlb; 131
.rodata:00004200007D3B20                 dq offset PVSCSI_SyncCmd; 132
.rodata:00004200007D3B28                 dq offset PVSCSI_ProcessRing; 133
.rodata:00004200007D3B30                 dq offset PVSCSI_PromoteCompletions; 134
.rodata:00004200007D3B38                 dq offset PVSCSI_ProcessCompletion; 135
.rodata:00004200007D3B40                 dq offset PVSCSI_DisableReqCallCoalescing; 136
.rodata:00004200007D3B48                 dq offset PVSCSI_EnableReqCallCoalescing; 137
.rodata:00004200007D3B50                 dq offset PVSCSI_DisableAsyncProcessing; 138
.rodata:00004200007D3B58                 dq offset PVSCSI_EnableAsyncProcessing; 139
.rodata:00004200007D3B60                 dq offset PVSCSI_CheckShadowRingQuiesced; 140
.rodata:00004200007D3B68                 dq offset Net_VMMStopPacketFilter; 141
.rodata:00004200007D3B70                 dq offset VMKPCIPassthru_UnmaskVector; 142
.rodata:00004200007D3B78                 dq offset VMKPCIPassthru_UpdatePrivateDomain; 143
.rodata:00004200007D3B80                 dq offset VMKPCIPassthru_SetAddressDomain; 144
```

VMware Hypervisor Reverse Engineering

- We can explain lots of structure in vmx through analyzing vmm
- vmm can also be the scope of our research for vulnerabilities
- We found new hypervisor related binary module - VMKernel through analyzing vmm

VMware Hypervisor Reverse Engineering

- Main binary modules related to Vmware hypervisor
  - vmware-vmx.exe/ vmx (ESXi)
  - vmmblob.elf (vmm)
  - VMKernel (ESXi)
- Bugs in them possibly influence Host

VMware Device Virtualization Bug Hunting

VMware Device Virtualization Bug Hunting

- The strategies of Bug Hunting
  - Automated analysis
    - Fuzzing
  - Manual analysis
    - Reverse Engineering

#HITB2024BKK

VMware Device Virtualization Bug Hunting

- In-process fuzzing
  - Use Frida to direct call target function
  - Use Stalker to get coverage information
- Drawbacks
  - DBI is very slow, almost can not run the Guest Machine normally
  - May be influenced by other thread or global variable
  - POC won't directly work in Guest Machine

#HITB2024BKK

VMware Device Virtualization Bug Hunting

- Directly input testcases from Guest OS to virtual devices
  - Hook functions to get corpus
  - Use static binary instrumentation to get coverage
  - Directly transfer testcases through physical memory
- Drawbacks
  - Coverage information may not be accurate
  - Need to analyze the driver code

VMware Device Virtualization Bug Hunting

- We tried a lot, but end up nothing
- Need to improve the mutation strategies
- Require lots of efforts to read devices documents

#HITB2024BKK

VMware Device Virtualization Bug Hunting

- VMware has many device implementations
- We don't have much patience to write fuzzer according to device documentation
- Since we have read devices documentation, lets just start to manual hunt bug

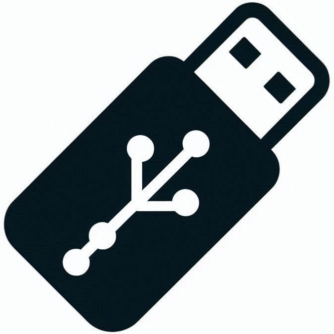
#HITB2024BKK

USB Emulation Bug Hunting

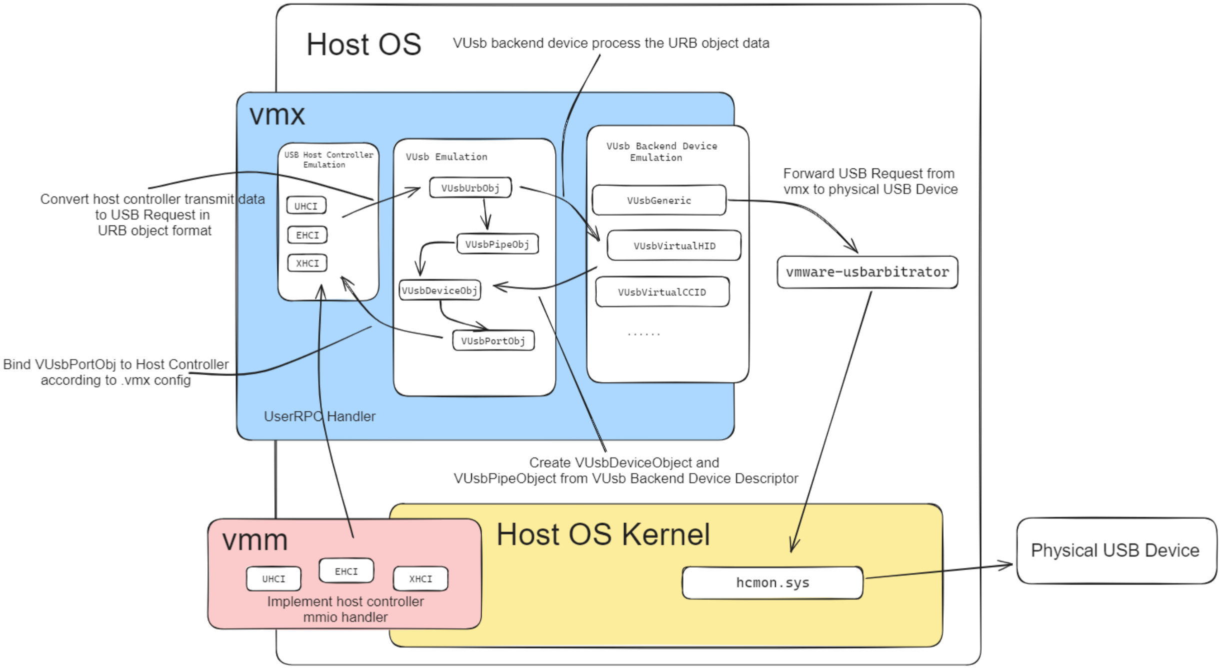
#HITB2024BKK

VMware Device Virtualization Bug Hunting

- USB Host Controller Emulation
  - UHCI, EHCI, XHCI
- VUSB Emulation
  - Urb Object, Pipe Object, Port Object ...
- VUSB Backend Device Emulation
  - Generic, Bluetooth, Rng ...



VMware Device Virtualization Bug Hunting



#HITB2024BKK

CVE-2024-22255 - Uninitialized Memory

CVE-2024-22255 - Uninitialized Memory

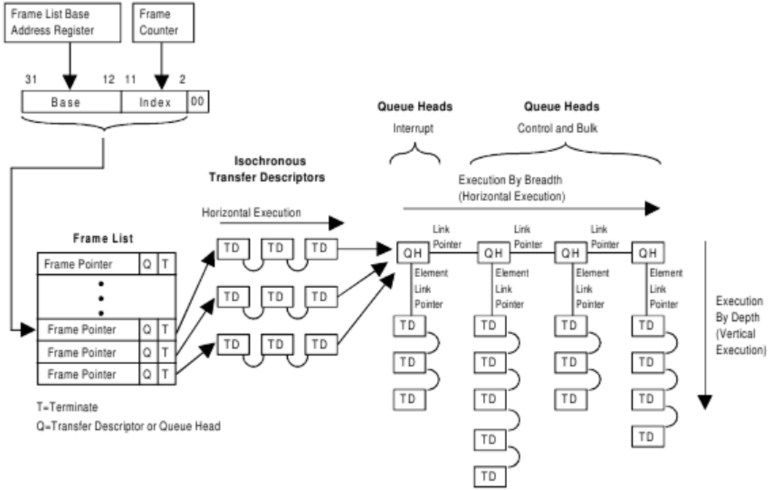
- One of the payloads used by USB devices is the Standard Device Request, which begins in the format of Setup Packet

- "wLength" is the most interesting fields, which indicates the length of data requested to the USB device

Offset	Size	Field
0	1	bmRequestType
1	1	bRequest
2	2	wValue
4	2	wIndex
6	2	wLength

CVE-2024-22255 - Uninitialized Memory

- The Standard Device Request serves as the payload for USB devices
- USB host controllers do not transfer data based on this unit
- For UHCI, data is transferred in units of Transfer Descriptors (TDs) and linked in guest memory in a list-like structure known as Queue Head (QH)



#HITB2024BKK

CVE-2024-22255 - Uninitialized Memory

- When processing control transfers, VMware's UHCI controller allocates URB objects on a per-Standard Device Request basis
- VMware retrieves the first TD on the Queue Head (QH) and uses it as the starting point to parse the Setup Packet
- It extracts the "wLength" field from the Setup Packet and adds the size of the Setup Packet to determine the size of the data buffer for the URB object

#HITB2024BKK

CVE-2024-22255 - Uninitialized Memory

```
    urbBufferSize = tdDataSize + HIWORD(v27);   // wLength
    URB = VUsb_AllocUrb(vusbPipeObject, 0i64, (unsigned int)urbBufferSize);// alloc urb with size sizeof(setup packet) + wLength
    URB->urbAllocatedDataBufferSize = urbBufferSize;
LABEL_35:
    if ( !URB )
    {
LABEL_44:
      Log_Level(0x1E43u, "UHCI: Unexpected in/out %d pid %2x.\n", (unsigned int)v7, *((unsigned __int8 *)v10 + 24));
      v10[5] &= 0xE77FFFFF;
      v10[5] |= 0x6407FFu;
      v19 = *(unsigned int **)(*(_QWORD *)v10 + 8i64);
      v20 = *v19;
      v19[1] = v10[5];
      if ( (_DWORD)v20 != v10[4] )
      {
        LODWORD(v21) = v10[4];
        Log_Level(
          0x1E43u,
          "UHCI: Link pointer in TD changed: TD %#I64x, new link %x, old link %x.\n",
          **(_QWORD **)v10,
          v20,
          v21);
      }
      *(_DWORD *)(a1 + 1640) |= 5u;
LABEL_47:
      v18 = v22;
      goto LABEL_48;
    }
    _mm_lfence();
    if ( (int)tdDataSize > urbBufferSize )      // check tdsize not overflow the urbBufferSize
    {
      _mm_lfence();
      Log_Level(0x1E43u, "UHCI: Setup data packet over run %d %d.\n", tdDataSize, (unsigned int)urbBufferSize);
      tdDataSize = 0;
      if ( urbBufferSize > 0 )
        tdDataSize = urbBufferSize;
    }
    UrbDataBufferPointer = (void *)URB->UrbDataBufferPointer;
    if ( v28.type == 1 )
    {
      memcpy(UrbDataBufferPointer, (const void *)v28.contentHostVA, (int)tdDataSize);
    }
    else
    {
      _mm_lfence();
      PhysMemReadSlow(&v28, 0i64, (int)tdDataSize, (char *)UrbDataBufferPointer);
    }
    URB->UrbDataBufferPointer += (int)tdDataSize;
    goto LABEL_15;
  }
```

#HITB2024BKK

CVE-2024-22255 - Uninitialized Memory

- The allocation process of URB depends on the target device you are transferring to
- Different types of backend USB devices will result in URB objects with varying private structures

```
if ( bufferSize > v11 )
  Panic("UsbDev: URB greater than the max allowed URB size.\n");
_mm_lfence();
v12 = (VusbUrbObj *)((__int64 (__fastcall *)(VUsbBackendDeviceObj *, _QWORD, _QWORD))pipeObject->VUsbBackendDeviceObj->backendObj->VUsbBackendUrbOperation->AllocUrb)(
                      pipeObject->VUsbBackendDeviceObj,
                      (unsigned int)packets,
                      bufferSize);
v12->UrbHandleReturnState = -1;
v12->IntervalEntry = (UrbIntervalEntry *)&v12[1];
v12->UrbDataBufferPointer = (__int64)v12->UrbDataBufferAllocedByUrbSize;
v12->StreamID = 0;
v12->vusbPipeObject = pipeObject;
*(_QWORD *)&v12->field_50 = 0i64;
v12->UrbSize = bufferSize;
*(_QWORD *)&v12->urbAllocatedDataBufferSize = 0i64;
v12->UrbFlowState = 0;
v12->RefCnt = 1;
v12->PipeType = pipeObject->PipeType;
v12->endPointAddr = pipeObject->endPointAddress;
backendObj = pipeObject->VUsbBackendDeviceObj->backendObj;
v12->PipeUrbNode.front = &v12->PipeUrbNode;
v12->PipeUrbNode.next = &v12->PipeUrbNode;
v12->SubmitUrbNode.front = &v12->SubmitUrbNode;
v12->SubmitUrbNode.next = &v12->SubmitUrbNode;
v12->backendObj = backendObj;
v12->field_68 = 0;
v12->PacketQueueHelper = 0i64;
```

CVE-2024-22255 - Uninitialized Memory

- For HID devices, when allocating URB objects, no additional structures are added besides the generic data fields of the URB
- Additionally, HID devices utilize malloc for data allocation

```
VusbUrbObj *__fastcall UsbVirtualHIDAllocUrb(__int64 a1, unsigned int a2, unsigned int a3)
{
  __int64 v3; // rbx
  VusbUrbObj *urb; // rax

  v3 = 12i64 * a2;
  urb = (VusbUrbObj *)Util_SafeMalloc(v3 + a3 + 0x98i64);
  urb->GenericDeviceUrbPrivateField = (GenericDeviceUrbPrivateFieldObj *)&unk_7FF7CF66AB30;
  urb->UrbDataBufferAllocedByUrbSize = (char *)&urb[1] + v3;
  return urb;
}
```

CVE - 2024 - 22255 - Uninitialized Memory

- Allocating wLength sized URB doesn't mean you will get wLength sized data from guest supplied TDs
- Malloc allocation left memory uninitialized
- Backend USB device returns data through the same URB buffer, leading to a heap data leak

```
    case USB_DEVICE_REQUEST_TYPE_SET_CONFIGURATION:
      if ( v7 < 0 || (v7 & 0x1F) != 1 )
        goto LABEL_26;
      v9 = *(void (__fastcall **)(_QWORD, _QWORD, _QWORD, _BYTE *
      if ( v9 )
        v9(
          *((unsigned __int16 *)UrbDataBufferAllocedByUrbSize + 2
          HIBYTE(*((unsigned __int16 *)UrbDataBufferAllocedByUrbS
          (unsigned __int8)*((_WORD *)UrbDataBufferAllocedByUrbSi
          v8,
          urb->urbAllocatedDataBufferSize - 8);
      break;
    case USB_DEVICE_REQUEST_TYPE_GET_INTERFACE:
      urb->UrbHandleReturnState = 0;
      urb->UrbReturnDataSize = 8;
      goto LABEL_34;
    default:
.ABEL_26:
      urb->UrbHandleReturnState = 3;
      goto LABEL_34;
  }
  if ( payloadSize >= 0 )
  {
    urb->UrbHandleReturnState = 0;
    urb->UrbReturnDataSize = payloadSize + 8;
    goto LABEL_34;
  }
  goto LABEL_26;
}
```

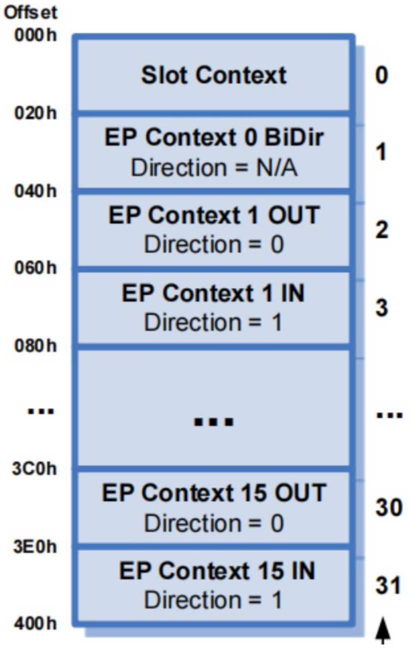
#HITB2024BKK

CVE-2024-22252 - Use After Free

#HITB2024BKK

CVE-2024-22252 - Use After Free

- Device Slot Context
  - Element 0 points to a Slot Context structure, which holds information for the device
- Endpoint Context
  - An Endpoint Context structure holds context information for a single endpoint
- Transfer Ring
  - Each endpoint has one or more Transfer Rings. A Transfer Ring is an array of Transfer Request Blocks (TRBs)



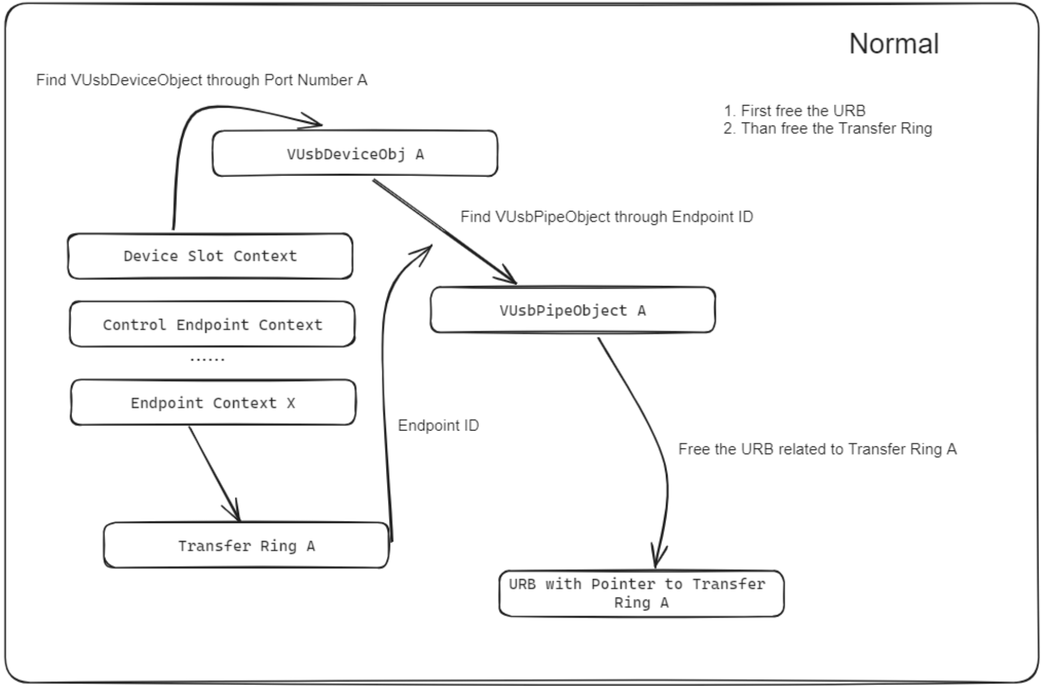
CVE-2024-22252 - Use After Free

- Look back to the old bug - CVE-2021-22040
- Before you figure out the XHCI emulation code, you may be confused

```
while ( v30 )
{
  _BitScanForward(&v18, v30);
  v19 = 1 << v18;
  v30 ^= 1 << v18;
  if ( v18 == -1 )
    break;
  v20 = 8i64 * (int)v18;
  v21 = *(_OWORD *)&v31[v20 + 4];
  *(_OWORD *)&v9[v20 + 4] = *(_OWORD *)&v31[v20];
  *(_OWORD *)&v9[v20 + 8] = v21;
  v9[2] |= v19;
  XhciStreams_FreeEndpoint(a1, v6, v18);// Bug! free after the context modification
  v22 = a1 + 1296i64 * v6;
  v23 = 32i64 * v18;
  v24 = *(_QWORD *)(v23 + v22 + 332536);
  if ( (v24 & 7) != 1 )
  {
    *(_QWORD *)(v23 + v22 + 332536) = v24 & 0xFFFFFFFFFFFFFFF8ui64 | 1;
    *(_DWORD *)(v22 + 332528) |= v19;
  }
}
```

```
while ( v30 )
{
  _BitScanForward(&v18, v30);
  v19 = 1 << v18;
  v30 ^= 1 << v18;
  if ( v18 == -1 )
    break;
  XHCI_FreeEndpoint(a1, v6, v18);       // patch, call free before the context modificatio
  v20 = 8i64 * (int)v18;
  v21 = 32i64 * v18;
  v22 = *(_OWORD *)&v31[v20 + 4];
  *(_OWORD *)(v20 * 4 + v9 + 16) = *(_OWORD *)&v31[v20];
  *(_OWORD *)(v20 * 4 + v9 + 32) = v22;
  *(_DWORD *)(v9 + 8) |= v19;
  v23 = a1 + 1296i64 * v6;
  v24 = *(_QWORD *)(v21 + v23 + 332536)
  if ( (v24 & 7) != 1 )
  {
    *(_QWORD *)(v21 + v23 + 332536) = v24 & 0xFFFFFFFFFFFFFFF8ui64 | 1;
    *(_DWORD *)(v23 + 332528) |= v19;
  }
}
```

CVE - 2024 - 22252 - Use After Free



#HITB2024BKK

CVE -2024 -22252 - Use After Free

- Release the URB objects on Backend USB Device

```
Log_Level(
  6u,
  "UsbDev: DevID(%I64x): Cancel pipe(%p).\n",
  pipeObject->VUsbBackendDeviceObj->UsbDeviceProperties.DevID,
  pipeObject);
((void (__fastcall *)(VUsbBackendDeviceObj *, _QWORD))pipeObject->VUsbBackendDeviceObj->backendObj->VUsbBackendUrbOperation->CancelEndpoint)(
  pipeObject->VUsbBackendDeviceObj,
  (unsigned int)pipeObject->endPointAddress);
front = (__int64)pipeObject->URBList.front;
result = 0i64;
if ( (UrbListNode *)front != &pipeObject->URBList )
{                                              // Release Urb on Pipe
  do
  {
    v4 = *(VUsb_PipeObject **)(front - 0x10);
    v5 = (VusbUrbObj *)(front - 40);
    v6 = *(VUsb_PipeObject **)front;
    v7 = *(_DWORD *)(front - 40 + 0x50);
    urbAllocatedDataBufferSize = *(_DWORD *)(front - 40 + 8);
    LODWORD(v14) = v4->endPointAddress;
    Log_Level(
      7u,
      "UsbDev: DevID(%I64x): Removing URB(%p) from pipe(%p), endpt(%x).\n",
      v4->VUsbBackendDeviceObj->UsbDeviceProperties.DevID,
      (const void *)(front - 40),
      v4,
      v14);
```

#HITB2024BKK

CVE-2024-22252 - Use After Free

- Endpoint Context is not the only object that holds a Transfer Ring Object pointer
- URB Object also holds a pointer that points to a field for Transfer Ring Object
- This field is responsible for tracking the corresponding TRB's data on Transfer Ring Object when XHCI returns USB device responses to the Guest

CVE - 2024 - 22252 - Use After Free

- Before patch, XHCI commands like 'Configure Endpoint' could modify the contents of the Endpoint Context before releasing the Transfer Ring
- 'Configure Endpoint' could modify the contents of the Endpoint Context, leading the type mismatch with the VUsbPipeObject object type

```
if ( v30 )
{
  PipeType = (unsigned int)v30->PipeType;
  epType = (*(_QWORD *)packetQueue->endpointContext >> 35) & 7i64;
  if ( (_DWORD)PipeType == endpointType2PipeType[epType] )
  {                                           // check endpoint type match pipe type
    return 1;
  }
  else
  {
    LODWORD(v34) = endPoint_;
    LODWORD(v33) = deviceSlot;
    Log_Level(
      0x1F03u,
      "XHCIPQUEUE: ERROR, mismatched pipe type (%d) for endpoint type (%d) on %02x:%02x, addr %d.\n",
      PipeType,
      epType,
      v33,
      v34,
      LOBYTE(v6->SlotContext[0].field4));
    result = 0;
    packetQueue->vusbPipeObject = 0i64;     // modify the endpoint context made the vusbPipeObject be NULL!
  }
}
else
```

#HITB2024BKK

CVE - 2024 - 22252 - Use After Free

- Left URB Object not freed, but related Transfer Ring Object already freed
- Dangling pointer - Use After Free

```
if ( transferRing )
{
  XhciPacketQueue_Init(&v9, controller, transferRing->doorbellArg, &transferRing->packetQueueHelper);// didn't check whether we get the VUsbPipeObject!
  XhciPacketQueue_Cancel(&v9);
```

```
PacketQueueHelper *__fastcall XhciPacketQueue_Cancel(XHC_PacketQueue *packetQueue)
{
  VUsb_PipeObject *vusbPipeObject; // rcx
  PacketQueueHelper *result; // rax

  vusbPipeObject = packetQueue->vusbPipeObject;
  if ( vusbPipeObject )                           // null, will not free the URB!
  {
    VUsb_CancelPipe(vusbPipeObject);
    packetQueue->packetQueueHelper->TransferUrbLength = 0;
    result = packetQueue->packetQueueHelper;
    result->UrbField = 0;
  }
  return result;
}
```

#HITB2024BKK

CVE - 2024 - 22252 - Use After Free

- It is still possible to modify the Device Slot Context to retrieve another VUsbDeviceObject, leading to the inability to obtain the correct VUsbPipeObject

Slot Context Data Structure

31 27 26 25 24 23 22 21 20 19 18 17 16 15 8 7 0

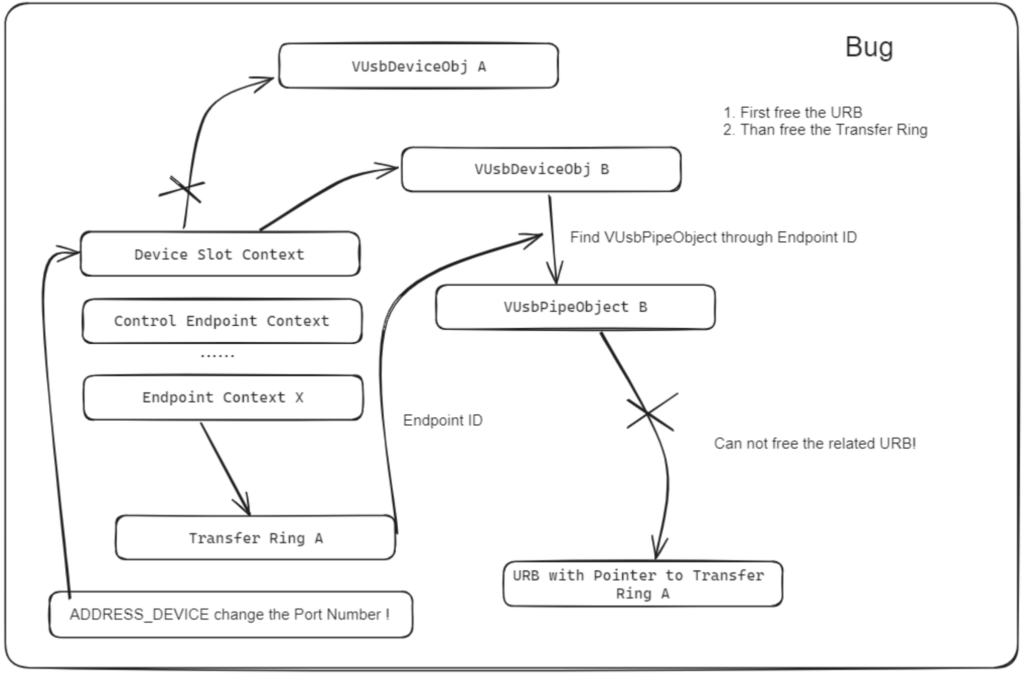
Context Entries	Hub	MTT	RsvdZ	Speed	Route String	03-00H
Number of Ports	Root Hub Port Number	Max Exit Latency				07-04H
Interrupter Target	RsvdZ	TTT	TT Port Number	TT Hub Slot ID		0B-08H
Slot State	RsvdZ	USB Device Address				0F-0CH
xHCI Reserved (RsvdO)						13-10H
xHCI Reserved (RsvdO)						17-14H
xHCI Reserved (RsvdO)						1B-18H
xHCI Reserved (RsvdO)						1F-1CH

CVE-2024-22252 - Use After Free

- First complete the configuration process for a device, and create Transfer Rings on non-Control Endpoints
- Transfer URB data on those Transfer Rings
- Use the 'ADDRESS_DEVICE' command on that Device Slot to modify the Device Port Number in the Slot Context to point to another USB device
- VMware's implementation ensures that 'ADDRESS_DEVICE' does not affect other non-Control Endpoint Contexts

```
XhciStreams_FreeEndpoint(controller, slotId_Minus_1, 1i64);
xhc_device_context->SlotContext[0] = XHC_InputContext.SlotContext[0];
*(_OWORD *)&xhc_device_context->ControlEndpoint0.field1 = *(_OWORD *)&XHC_InputContext.EndpointContexts[0].field1;
v9 = *(_OWORD *)&XHC_InputContext.EndpointContexts[0].field5;// ADDRESS_DEVICE Only Modify the Slot Context and Control Endpoint Context
xhc_device_context->field_8 = -1;
*(_OWORD *)&xhc_device_context->ControlEndpoint0.field5 = v9;
sub_7FF7CE575D50(controller, slotId_Minus_1, 1u, 1);
v10 = XHC_InputContext.SlotContext[0].field1 & 0xFFFFF;
v11 = (unsigned int)BYTE2(XHC_InputContext.SlotContext[0].field2) - 1;
if ( (unsigned int)v11 < controller->usbPortInformation.numMaxPorts )
{
```

CVE - 2024 - 22252 - Use After Free



CVE-2024-22251 - Out-of-Bounds Read

#HITB2024BKK

CVE-2024-22251 - Out-of-Bounds Read

- The Guest OS communicates with SmartCard through the Virtual SmartCard Reader
- Guest OS use CCID protocol to communicate with Virtual SmartCard Reader
- The APDU (Application Protocol Data Unit) serves as the data unit for interaction between the SmartCard Reader and the SmartCard

```
00000000 ccid_xfrblock_msg_hdr struc ; (sizeof=0xA, mappedto_759)
00000000                                          ; XREF: ccid_xfrbl
00000000 msg_type         db ?
00000001 msg_len          dd ?
00000005 slot_num         db ?
00000006 seq_num          db ?
00000007 bwi              db ?
00000008 level_param      dw ?
0000000A ccid_xfrblock_msg_hdr ends

00000000 command_apdu     struc ; (sizeof=0x5, mappedto_760)
00000000                                          ; XREF: ccid_xfrbl
00000000 cla              db ?
00000001 ins              db ?
00000002 p1               db ?
00000003 p2               db ?
00000004 len              db ?
00000005 command_apdu     ends

00000000 ccid_xfrblock_msg_with_command_apdu struc ; (sizeof=0xF, m
00000000 hdr              ccid_xfrblock_msg_hdr ?
0000000A apdu             command_apdu ?
0000000F ccid_xfrblock_msg_with_command_apdu ends
```

#HITB2024BKK

CVE-2024-22251 - Out-of-Bounds Read

- VMware checks whether the 'msg_len' field of ccid_xfrblock_msg_hdr matches the 'len' field of the command_apdu
- However, it fails to verify whether these two fields conform to the size of the URB buffer

```
msg_len = Buffer->hdr.msg_len;
APDU_LEN = msg_len - 4;
if ( msg_len < 4 )
{
  LogInfo("USB-CCID: Invalid len of APDU.\n", APDU_LEN);// Application Protocol Data Unit
  v8 = 0;
LABEL_41:
  v16 = (char *)Util_SafeCalloc(1ui64, 0xAui64);
  goto LABEL_42;
}
if ( (unsigned int)APDU_LEN >= 2 )
{                                               // only check the apdu len match the msg_len
                                                // but what about URB data buffer?
  len = (unsigned __int8)Buffer->apdu.len;
  if ( ((_DWORD)APDU_LEN != len + 1 || !(_BYTE)len) && ((_DWORD)APDU_LEN != len + 2 || !(_BYTE)len) )
  {
    LogInfo(
      "USB-CCID: Unexpected apdu case, CLA:0x%1x, INS:0x%1x, P1:0x%1x, P2:0x%1x.\n",
      (unsigned __int8)Buffer->apdu.cla,
      (unsigned __int8)Buffer->apdu.ins,
      (unsigned __int8)Buffer->apdu.p1,
      (unsigned __int8)Buffer->apdu.p2);
    v8 = 0;
    goto LABEL_41;
  }
}
```

#HITB2024BKK

CVE-2024-22251 - Out-of-Bounds Read

- Directly uses these fields as parameters to call the Windows SCardTransmit API
- SCardTransmit takes a buffer pointer and buffer size as parameters and cannot verify the validity between these two parameters
- Out-of Bounds Access to Heap Data

```
}                                                   // OOB Read occurs in SCardTransmit!
v20 = SCardTransmit(
        ccidDevice->hCard,
        v19,
        (LPCBYTE)&Buffer->apdu,
        Buffer->hdr.msg_len,
        0i64,
        (LPBYTE)v16 + 10,
        &pcbRecvLength);
v21 = v20;
```

Conclusion

- Host controller emulation can be attacked
- VUSB emulation can be attacked
- USB device emulation can be attacked
- We have other cases we did not include in this presentation, but you can differ the vmx binary to found
- More attack scenarios in the future?
  - Plug in an evil USB device and leverage vmx (Generic USB device, ...) to execute code?
  - Leverage local USB service (usbarbitrator, ...) to privilege escalation?
  - ……
- Very challenging to defend such a complex system

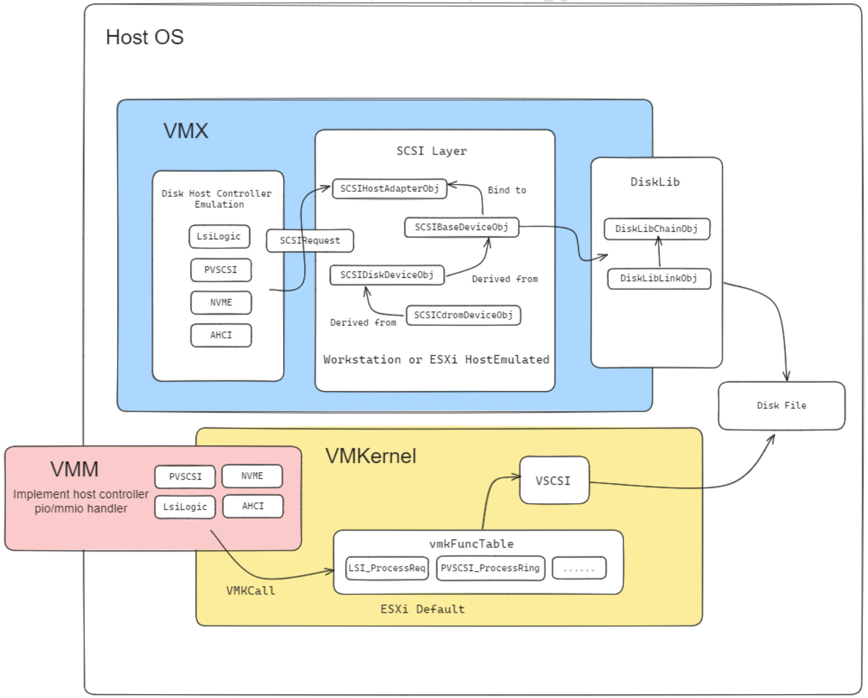
SCSI Emulation Bug Hunting

Differences Between ESXi And Workstation

- The data flow direction of device emulation in ESXi is different from that in Workstation

#HITB2024BKK

SCSI Emulation Architecture



SCSI Data Flow Transmission Direction

- Transmission of SCSI data stream in Workstation

```
}
else                                          // MPI_FUNCTION_SCSI_IO_REQUEST
{
  LSILogicImplProcessSCSIIOMessage(a1, (__int64)&scsi_io_request, v7, 0, v12, v13);
}
return PhysMem_Release(v19);
```

```
if ( *(_BYTE *)(a1 + 0x4F0) )                     // hostedEmulation_flag
  return LSILogicHostedProcessSCSIIOMessage(a1, (__int64)scsi_io_request, a3);
return result;
```

#HITB2024BKK

SCSI Data Flow Transmission Direction

- Transmission of SCSI data stream in Workstation

```
if ( *(_BYTE *)(a1 + 0x4F0) )                          // hostedEmulation_flag
  return LSILogicHostedProcessSCSIIOMessage(a1, (__int64)scsi_io_request, a3);
return result;
```

```
*(&userRpcBlock + 3) = a3;
*(__int64 *)((char *)&userRpcBlock + 36) = *(_QWORD *)a2;
*(__int64 *)((char *)&userRpcBlock + 44) = *(_QWORD *)(a2 + 8);
*(__int64 *)((char *)&userRpcBlock + 52) = *(_QWORD *)(a2 + 16);
*(__int64 *)((char *)&userRpcBlock + 60) = *(_QWORD *)(a2 + 24);
*(__int64 *)((char *)&userRpcBlock + 68) = *(_QWORD *)(a2 + 32);
*(__int64 *)((char *)&userRpcBlock + 76) = *(_QWORD *)(a2 + 40);
*(__int64 *)((char *)&userRpcBlock + 84) = *(_QWORD *)(a2 + 48);
*((_DWORD *)&userRpcBlock + 23) = *(_DWORD *)(a2 + 56);
UserRPC(377, 280, v4, v5, v6, v7);
```

SCSI Data Flow Transmission Direction

- Transmission of SCSI data stream in ESXi

```
if ( lsilogic_sharedArea->cifvalue <= 0x7F )
{
  if ( lsilogic_sharedArea->hostedEmulation_flag )
    LSILogicHostedProcessSCSIIOMessage(lsilogic_sharedArea, scsi_io_request, guest_phy_mem);
  else
    LSILogicVMKProcessSCSIIOMessage(lsilogic_sharedArea, scsi_io_request, guest_phy_mem);
  return PhysMem_Release((__int64 *)&v14.field_0);
}
```

```
    {
      VMK_Call_1Args(0x77u, (unsigned int)lsilogic_shared->adapter_idx);
    }
    return 0LL;
  }
  target_adapter_lsilogic_req_num_ring->status = 3;
  result = VMK_Call_1Args(0x76u, (unsigned int)lsilogic_shared->adapter_idx);
```

SCSI Data Flow Transmission Direction

- Different code paths present different attack surfaces

#HITB2024BKK

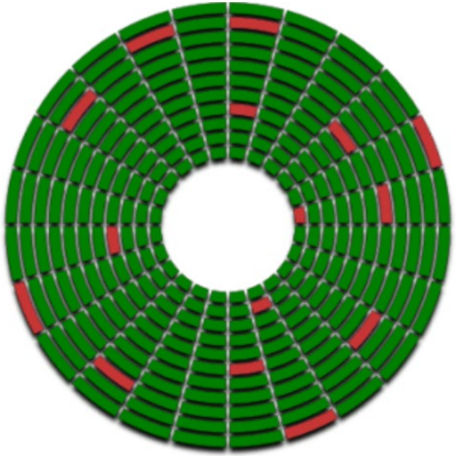
CVE - 2024 - 22273 - Out - of - Bounds Read/Write

#HITB2024BKK

CVE-2024-22273 - Out-of-Bounds Read/Write

- The disk verifier is responsible for detecting whether the disk has bad sectors

- VMware implements a disk verifier mechanism



CVE-2024-22273 - Out-of-Bounds Read/Write

- The Write(16) command can write data to the specified 64-bit address

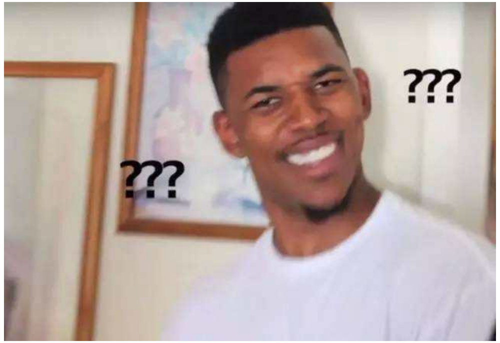
Table 219 WRITE (16) command

Bit / Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE (8Ah)							
1	WRPROTECT			DPO	FUA	Reserved	Obsolete	DLD2
2	(MSB)	LOGICAL BLOCK ADDRESS						
...								
9								(LSB)
10	(MSB)	TRANSFER LENGTH						
...								
13								(LSB)
14	DLD1	DLD0	GROUP NUMBER					
15	CONTROL							

CVE-2024-22273 - Out-of-Bounds Read/Write

- Normally, the access range of a “Write” or “Read” command is limited according to the disk capacity

```
if ( !*(_QWORD *)(v17 + 0xD8) )
{
  v19 = 4 * v5;                          // 4 * Disk capacity
  v20 = (void *)UtilSafeMalloc1(v19);
  *(_QWORD *)(v17 + 216) = v20;
  memset(v20, 255, v19);
}
v21 = **(_BYTE **)(a1 + 0x28);
if ( ((v21 - 0xA) & 0x5F) == 0 || ((v21 - 8) & 0x5F) == 0 )
  *(_DWORD *)(*(_QWORD *)(v17 + 216) + 4 * v15) = v18;
```



#HITB2024BKK

CVE-2024-22273 - Out-of-Bounds Read/Write

- The "Write(16)" command can be used to write any data to any address

```
do
{
  v15 = v8 + *(_QWORD *)(a1 + 0x70);
  v16 = sub_140604080(v13, v6);
  v17 = *(_QWORD *)(a1 + 16);
  v18 = v16;
  if ( !*(_QWORD *)(v17 + 0xD8) )
  {
    v19 = 4 * v5;                        // 4 * Disk capacity
    v20 = (void *)UtilSafeMalloc1(v19);
    *(_QWORD *)(v17 + 216) = v20;
    memset(v20, 255, v19);
  }
  v21 = **(_BYTE **)(a1 + 0x28);
  if ( ((v21 - 0xA) & 0x5F) == 0 || ((v21 - 8) & 0x5F) == 0 )
    *(_DWORD *)(*(_QWORD *)(v17 + 216) + 4 * v15) = v18;// Heap Overflow
  v6 = v22;
  ++v8;
  v5 = v23;
  v13 += v14;
}
while ( v8 < *(_QWORD *)(a1 + 120) );
```

#HITB2024BKK

CVE-2024-37086 - Out-of-Bounds Read

CVE-2024-37086 - Code Path to VMKernel

- The “UNMAP” command allows one or more Logical Block Addresses to be unmapped

CVE-2024-37086 - Out-of-Bounds Read

Table 204 UNMAP command

Byte \ Bit	7	6	5	4	3	2	1	0
0	OPERATION CODE (42h)							
1	Reserved							ANCHOR
2	Reserved							
...								
5								
6	Reserved			GROUP NUMBER				
7	(MSB)	PARAMETER LIST LENGTH						
8								(LSB)
9	CONTROL							

#HITB2024BKK

CVE-2024-37086 - Out-of-Bounds Read

Table 205 UNMAP parameter list

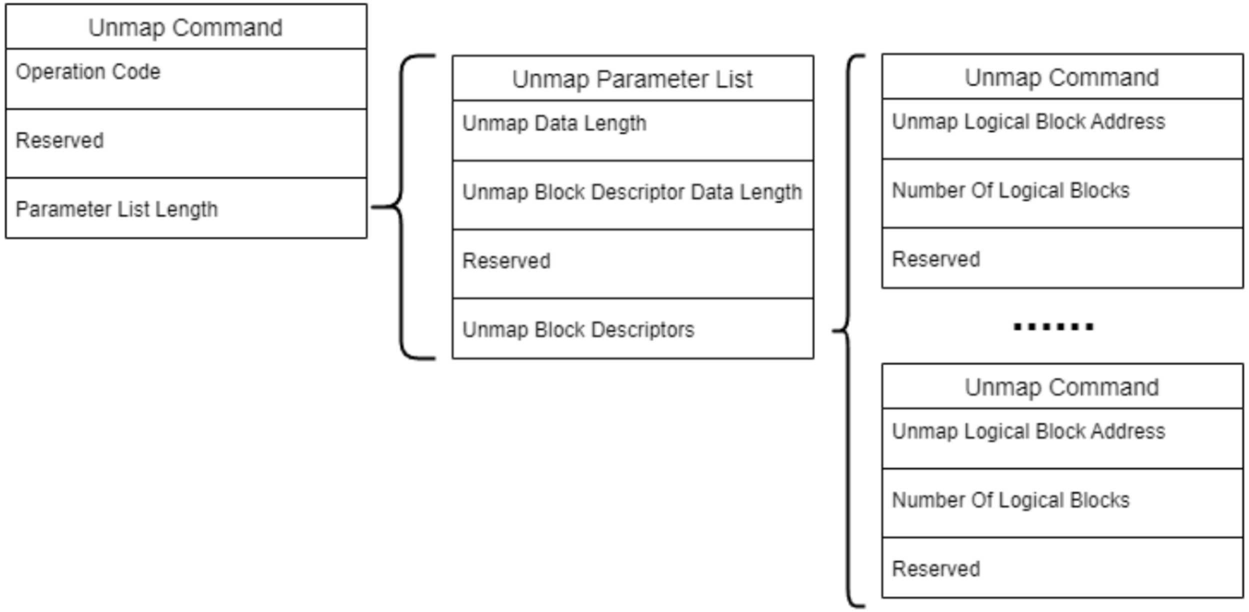
Bit / Byte	7	6	5	4	3	2	1	0
0	(MSB)	UNMAP DATA LENGTH (n-1)						
1								(LSB)
2	(MSB)	UNMAP BLOCK DESCRIPTOR DATA LENGTH (n-7)						
3								(LSB)
4	Reserved							
...								
7								
UNMAP block descriptors								
8	UNMAP block descriptor [first] (see table 206)							
...								
23								
...								
n-15	UNMAP block descriptor [last] (see table 206)							
...								
n								

CVE-2024-37086 - Out-of-Bounds Read

Table 206 UNMAP block descriptor

Bit / Byte	7	6	5	4	3	2	1	0
0	(MSB)							
.....	UNMAP LOGICAL BLOCK ADDRESS							
7								(LSB)
8	(MSB)							
....	NUMBER OF LOGICAL BLOCKS							
11								(LSB)
12								
.....	Reserved							
15								

CVE - 2024 - 37086 - Out - of - Bounds Read



CVE-2024-37086 - Out-of-Bounds Read

- Verify before using the “UNMAP” command

```
ret_code = VSCSI_CheckUnmapCmd(vscsiHandle, token, SCSIIO_Command);
if ( ret_code )
  goto LABEL_48;
```

CVE - 2024 - 37086 - Out - of - Bounds Read

- Verify before using the “UNMAP” command

```
if ( Parameter_List_Length )
{
  if ( Parameter_List_Length > 7uLL )
  {
    Parameter_List = (Parameter_List *)VSCSI_Alloc(Parameter_List_Length);
    Parameter_List_1 = Parameter_List;
    if ( !Parameter_List )
    {
      ret_code = 0xBAD0014;
      v14 = 8;
      goto LABEL_6;
    }
    if ( !(unsigned __int8)Util_CopySGData(
                             (Inquiry36Response *)Parameter_List,
                             &SCSI_Command->sg_struct,
                             1,
                             0,
                             0,
                             Parameter_List_Length) )
    {
```

```
ret_code :
if ( ret_(
  goto LAB
```

CVE-2024-37086 - Out-of-Bounds Read

- Forgetting to check the correlation between "Parameter List Length" and "Unmap Block Descriptor Data Length"

```
if ( (v22 || unmap_block_desc_num <= vscsiHandle->UnmapConfig.max_block_desc)
  && (unmap_block_descriptor_data_length & 0xF) == 0
  && (unsigned __int16)__ROL2__(Parameter_List_1->unmap_data_length, 8) == unmap_block_descriptor_data_length
                                                                        + 6LL )
{
  if ( !(unmap_block_descriptor_data_length >> 4) )
  {

    ret_code = 0;
    VSCSI_Free(&Parameter_List_1);
    return ret_code;
  }
  unmap_descriptor = &Parameter_List_1->unmap_descriptor;
  while ( 1 )
  {
    unmap_logical_block_address = _byteswap_uint64(unmap_descriptor->unmap_logical_block_address);
    number_of_logical_block = _byteswap_ulong(unmap_descriptor->number_of_logical_block);
    if ( !v22 && vscsiHandle->UnmapConfig.field_0 < number_of_logical_block )
      break;
    if ( unmap_logical_block_address + number_of_logical_block > vscsiHandle->numBlocks )
    {
      v24 = v8;
      v25 = 33;
      goto LABEL_24;
    }
    if ( &Parameter_List_1->unmap_descriptor + unmap_block_desc_num == ++unmap_descriptor )
      goto LABEL_34;
  }
```

CVE-2024-37086 - Out-of-Bounds Read

- Use “Parameter List Length” as the length

```
do
{
  if ( a1a.Parameter_List > (Parameter_List *)p_unmap_block_descriptor
    || (char *)a1a.Parameter_List + a1a.parameter_list_length <= (char *)p_unmap_block_descriptor )
  {
    VSCSI_Free(&a1a);
    ret_code = Async_EndSplitIO((token_ **)a2, 0, 0, v3);
    if ( !ret_code )
      return ret_code;
    v157 = 0;
    v139 = 0LL;
    if ( ret_code == 0xBAD0002 )
      goto LABEL_448;
    goto LABEL_352;
  }
  number_of_logical_block = p_unmap_block_descriptor->number_of_logical_block;
  unmap_logical_block_address = p_unmap_block_descriptor->unmap_logical_block_address;
  a1a.p_unmap_block_descriptor = ++p_unmap_block_descriptor;
  v144 = _byteswap_ulong(number_of_logical_block);
  v145 = _byteswap_uint64(unmap_logical_block_address);
}
while ( !v144 );
v138 = Async_PrepareOneIO((__int64 *)&a2->field_0, a4);
```

#HITB2024BKK

CVE-2024-37086 - Out-of-Bounds Read

- Size variables are affected by “Logical Block Size”
- Unchecked “Logical Block Size” will cause Out-of-bound write

```
if ( size )
{
  v15 = &v63;
  curpos = Page_Start / v14;
  end = size + curpos;
  do
  {
    *(_DWORD *)v15 = curpos++;
    v15 += 3;
  }
  while ( curpos != end );
}
```

#HITB2024BKK

New Attack Surface Impact

- Modify the existing sandbox protection mechanis
- Elevate the current process privileges
- Virtual Machine Escape

#HITB2024BKK

Thank You!

#HITB2024BKK