KCon

云虚拟化安全：QEMU的安全探索和防御

演讲者：钱文祥

钱文祥 (@leonwxqian) Tencent Blade Team高级安全研究员

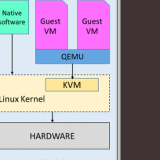
- 云虚拟化、IoT、浏览器等安全研究
- 发现NVIDIA vGPU等逃逸漏洞、Amazon Echo、Google Home音箱、浏览器中“麦哲伦”（SQLite）、Curl的多个远程代码执行漏洞等
- 多次在著名会议上参讲

- Tencent Blade Team由腾讯安全平台部在2017年底成立
- 专注于AIoT，移动互联网，云虚拟化技术，可信计算等前沿领域的安全技术研究
- 向Google、Microsoft、 Apple、Amazon、Huawei等诸多国际知名公司报告过200+安全漏洞
- 研究成果多次入选BlackHat、DEFCON、CanSecWest、HITB、POC、Xcon等顶级安全大会
- 团队官网：https://blade.tencent.com

纵横

云虚拟化

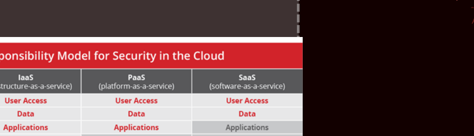
- 许多种云服务需要从一个物理硬件系统创建多个模拟环境
- 实现这个技术的核心便是虚拟化，具体则是Hypervisor层
- Hypervisor是一种运行在基础物理服务器和操作系统之间的中间软件层，可允许多个guest操作系统和应用共享硬件
- 典型的代表有：QEMU-KVM，Xen，VMware，Hyper-V等等



KCon

Hypervisor的位置

- Hypervisor是用户（Guest）和云服务提供商的“边缘”
- Host是总管理员，上面可运行大量虚拟化机器
- 用户看来，Guest就是一台电脑，用户可以完全控制Guest
- 但是如果有漏洞允许用户从Guest中穿透到Host上，则会对主机的隐私性、安全性造成极大危害



Shared Responsibility Model for Security in the Cloud

On-Premises (for reference)	IaaS (infrastructure-as-a-service)	PaaS (platform-as-a-service)	SaaS (software-as-a-service)
User Access	User Access	User Access	User Access
Data	Data	Data	Data
Applications	Applications	Applications	Applications
Operating System	Operating System	Operating System	Operating System
Network Traffic	Network Traffic	Network Traffic	Network Traffic
Hypervisor	Hypervisor	Hypervisor	Hypervisor
Infrastructure	Infrastructure	Infrastructure	Infrastructure
Physical	Physical	Physical	Physical

Customer Responsibility　Cloud Provider Responsibility

纵横

QEMU中的存储

- 存储一直是各虚拟机/容器的重要的攻击面
- 常见的存储模式有PATA/SATA/virtio-blk/virtio-sata......
- Virtio方式是QEMU默认支持功能中性能最高的存储使用方式
- 若不可使用virtio驱动程序，则x86一般可以使用**AHCI** (SATA)以提高效率
- QEMU的AHCI模块中发现了一个任意长度越界读写的问题
    - CVE-2020-29443

https://qemu.org/2021/01/19/virtio-blk-scsi-configuration/

KCon

AHCI（高级主机控制接口）设备简介

- AHCI (Advance Host Controller Interface)由 Intel 开发，用于提速 SATA 设备的数据处理
- AHCI是SATA的技术，SATA是PATA（原称ATA）演变而来的接口
    - 从QEMU源码树上看，PATA/SATA/AHCI都共享一部分源码
- AHCI可使用MMIO直接地访问磁盘，而无需像IDE那样使用复杂的PMIO命令序列
- AHCI控制器是具有总线主控能力的 PCI 设备
    - AHCI 控制器是系统内存和 SATA 设备之间的数据传输引擎
- 但AHCI的文档很少，常用的参考资料是 Intel AHCI 规范和Linux源码

KCon

按此顺序

1 FIS

2 ATAPI

3 PRDT

Offset	Region	Description
00h	Command FIS (up to 64 bytes)	Software constructed FIS For data transfer operations • This is the H2D Register FIS
40h	ATAPI Command (12 to 16 bytes)	Software constructed region that contains the ATAPI command (SCSI).
50h	Unused	
80h	PRDT Physical Region Descriptor Table • Contains the scatter / gather list for the data transfer • Up to 65 535 entries	

DWs	byte n+3	byte n+2	byte n+1	byte n	
0	Data Base Address (DBA)				0
1	Data Base Address Upper 32-bits (DBAU)				
2	Reserved				
3	Reserved	Byte Count (DBC)			1
Item 1	Data Base Address (DBA)				0
	Data Base Address Upper 32-bits (DBAU)				
	Reserved				
	Reserved	Byte Count (DBC)			1
•					
•					
Last Item	Data Base Address (DBA)				0
	Data Base Address Upper 32-bits (DBAU)				
	Reserved				
	Reserved	Byte Count (DBC)			1

Figure 10.13 Command table.

THE ESSENTIAL GUIDE TO SERIAL ATA AND SATA EXPRESS

David A. Deming

纵横

FIS （帧信息结构）

这部分→ Command FIS (up to 64 bytes)

- SATA使用FIS(Frame Information Structure)包在主机和设备之间传输数据
- SATA使用与PATA相同的命令集，SATA/PATA控制上最明显的区别就是FIS。
- 可以将FIS视为传统任务文件的数据集，或ATA命令的封装
- 共有下列几种FIS指令
  - Register FIS - host to device
  - Register FIS - device to host
  - DMA activate FIS - device to host
  - DMA setup FIS – bidirectional
  - Data FIS – bidirectional
  - BIST activate FIS – bidirectional
  - PIO setup FIS - device to host
  - Set device bits FIS - device to host

KCon

这部分 → ATAPI Command (12 to 16 bytes)

ATAPI Command

- ATAPI 是一种向连接到 ATA 总线的光驱或磁带驱动器发出SCSI命令的方法
- 最重要的是 **PACKET** 命令 (0xA0) 和 IDENTIFY PACKET DEVICE (0xA1)命令
- **PACKET**由1个字节**SCSI命令**，后跟 11个字节的**数据**组成

```
uint8_t atapi_readtoc[]=  { 0x43 /* ATAPI_READTOC */, 0, 1, 0, 0, 0, 0, 0, 12, 0x40, 0, 0};
```

- QEMU中名为“WIN_PACKETCMD （0xA0）”

纵横

KCon

PRDT

这部分→

PRDT
Physical Region Descriptor Table
- Contains the scatter / gather list for the data transfer
- Up to 65 535 entries

- 全称为物理区域描述表（Physical Region Descriptor Table）
- PRDT的每一项为16字节，分别有下列含义：

12-15	7-11	4-7	0-3
DBC - 字节计数	保留不用	DBAU - DBA地址（高位）	DBA – 数据基地址（低位）

- DBA是一个物理地址，它是**读**操作的**目标**地址，或者**写**操作的**源**地址
- DBC则用于指示每次操作的字节长度
- PRDT按链表的形式组织，一项连着一项，最多可以由65535个项目组成

漏洞的发现

触发漏洞的位置

- IO密集型代码注重性能，底层设计理念理应是“**无必要，不检查**”，检查应在代码早期阶段进行
- IO操作的最底层代码是***假想的***污点最终会到达的地方
- 从这里自底向上进行审计，我们发现了一个可疑的位置，它很像是一个**在循环中的memcpy**

```
/* Some adapters process PIO data right away.  In that case, we need
 * to avoid mutual recursion between ide_transfer_start
 * and ide_atapi_cmd_reply_end.
 */
if (!ide_transfer_start_norecurse(s,
                                  s->io_buffer + s->io_buffer_index - size,
                                  size, ide_atapi_cmd_reply_end)) {
    return;
}
```

memcpy(buffer + i - size, source, size);

- 它拷贝的对象是s->io_buffer，这是一个预先分配的，固定长度的Buffer
- Source是**客户机提供**的输入，size也是**客户机提供**的输入，它的参数看起来并不安全

KCon

污点的追踪——ide_atapi_cmd_read_pio

- 显然应当将source和size作为污点向上溯源
- 向上一层，控制复制总大小的packet_transfer_size由**nb_sectors** 和 sector_size计算而来（右图）
- 引入的新污点nb_sectors，则是从某个buf中读取出来的（左图）

```
972 static void cmd_read(IDEState *s, uint8_t* buf)
973 {
974     int nb_sectors, lba;
975 
976     if (buf[0] == GPCMD_READ_10) {
977         nb_sectors = lduw_be_p(buf + 7);
978     } else {
979         nb_sectors = ldl_be_p(buf + 6);
980     }
981 
982     lba = ldl_be_p(buf + 2);
983     if (nb_sectors == 0) {
984         ide_atapi_cmd_ok(s);
985         return;
986     }
987 
988     ide_atapi_cmd_read(s, lba, nb_sectors, 2048);
989 }
```

```
319 /* start a CD-CDROM read command */
320 static void ide_atapi_cmd_read_pio(IDEState *s, int lba, int nb_sectors,
321                                    int sector_size)
322 {
323     s->lba = lba;
324     s->packet_transfer_size = nb_sectors * sector_size;
325     s->elementary_transfer_size = 0;
326     s->io_buffer_index = sector_size;
327     s->cd_sector_size = sector_size;
328 
329     ide_atapi_cmd_reply_end(s);
330 }
331 
```

如果进一步回溯……

- 向上回溯三层，我们获得了这样一个调用链条：
- (hw/ide/atapi.c) cmd_read 或 cmd_read_cd→
  ide_atapi_cmd_read →
  ide_atapi_cmd_read_pio
- 到此还比较明晰，但如果重复这个步骤，问题就来了：
  - 污点的源头越来越多，很多变量都参与了这个链条
  - IDE/AHCI共用代码分支，分支岔路很多，很难确定数据通过哪条路径走过来
- 让我们从AHCI规范开始，定位代码的安全问题在哪里

理解AHCI的核心——Command Table

FIS指令结构体的布局

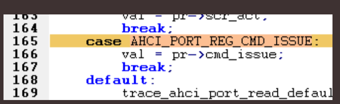
- 整个Command Table是一体的，但是为了方便介绍，这里先只提FIS部分
- 一个AHCI由6个port构成，设备使用的port是固定的，FIS等和port绑定
- 找到一块内存放置cmd_fis结构体，低地址物理内存如0x20000似乎是一个不错的选择
- MMIO写操作回调ahci_mem_write中也为PMIO预留了一部分内存区域
- 因此MMIO/PMIO读写其实都可以通过MMIO来完成

```
else if ((addr >= AHCI_PORT_REGS_START_ADDR) &&
           (addr < (AHCI_PORT_REGS_START_ADDR +
                 (s->ports * AHCI_PORT_ADDR_OFFSET_LEN)))) {
    ahci_port_write(s, (addr - AHCI_PORT_REGS_START_ADDR) >> 7,
                 addr & AHCI_PORT_ADDR_OFFSET_MASK, val);
```

FIS指令结构体的内存布局

- 通过MMIO→PMIO写回调设置lst的地址为0x20000，这是整个Command Table的地址
- 通过AHCI_PORT_REG_CMD_ISSUE触发FIS指令处理器

```
300 static void ahci_port_write(AHCIState *s, int port, int offset, uint32_t val)
301 {
302     AHCIPortRegs *pr = &s->dev[port].port_regs;
303     enum AHCIPortReg regnum = offset / sizeof(uint32_t);
304     assert(regnum < (AHCI_PORT_ADDR_OFFSET_LEN / sizeof(uint32_t)));
305     trace_ahci_port_write(s, port, AHCIPortReg_lookup[regnum], offset, val);
306
307     switch (regnum) {
308     case AHCI_PORT_REG_LST_ADDR:
309         pr->lst_addr = val;
310         break;
311     case AHCI_PORT_REG_LST_ADDR_HI:
312         pr->lst_addr_hi = val;
313         break;
```



- 在lst开头，例如0x20000处，按FIS格式设置好状态信息，包括最重要的 feature字段
- 要进入AHCI模式，需要feature & $11_B \neq 0$

KCon

FIS指令的触发

- FIS由handle_reg_h2d_fis处理
- AHCI_PORT_REG_CMD/CMD_ISSUE会调用check_cmd→handle_cmd映射FIS

```
370         break;
371     case AHCI_PORT_REG_CMD_ISSUE:
372         pr->cmd_issue |= val;
373         check_cmd(s, port);
374         break;
375     default:
```

```
584 static void check_cmd(AHCIState *s, int port)
585 {
586     AHCIPortRegs *pr = &s->dev[port].port_regs;
587     uint8_t slot;
588
589     if ((pr->cmd & PORT_CMD_START) && pr->cmd_issue) {
590         for (slot = 0; (slot < 32) && pr->cmd_issue; slot++) {
591             if ((pr->cmd_issue & (1U << slot)) &&
592                 !handle_cmd(s, port, slot)) {
593                 pr->cmd_issue &= ~(1U << slot);
594             }
595         }
596     }
597 }
```

- 通过FIS的第一个字节，确认是否要调用handle_reg_h2d_fis，它会设置ide_state这个重要结构体（模拟IDE寄存器）

```
switch (cmd_fis[0]) {
    case SATA_FIS_TYPE_REGISTER_H2D:
        handle_reg_h2d_fis(s, port, slot, cmd_fis);
```

纵横

KCon

IDE指令的运行

- handle_reg_h2d_fis中设置完IDE寄存器后，会调用ide_exec_cmd处理IDE指令
- 根据规范，IDE指令位于FIS的第三字节（fis[2]）

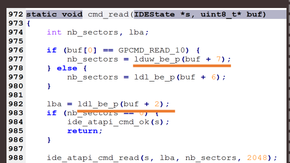
```
complete = ide_cmd_table[val].handler(s, val);
if (complete) {
```

```
2031        [WIN_SETIDLE2]                = { cmd_nop, HD_CFA_OK },
2032        [WIN_CHECKPOWERMODE2]         = { cmd_check_power_mode, H
2033        [WIN_SLEEPNOW2]               = { cmd_nop, HD_CFA_OK },
2034        [WIN_PACKETCMD]               = { cmd_packet, CD_OK },
2035        [WIN_PIDENTIFY]               = { cmd_identify_packet, CD
2036        [WIN_SMART]                   = { cmd_smart, HD_CFA_OK |
2037        [CFA_ACCESS_METADATA_STORAGE] = { cmd_cfa_access_metadata
```

- 它在ide_cmd_table中找寻和传入的命令对应的处理函数
- 我们需要它继续处理Command Table中的读取操作，因此应使用WIN_PACKETCMD指令
- cmd_packet也可以通过IDE设备的ATA_IOPORT_WR_COMMAND触发，但使用FIS更简单

从cmd_packet到cmd_read

- cmd_packet通过读取ATAPI指令部分来跳转到cmd_read，比较简单不再展开
- cmd_read的参数“buf”就是我们在物理内存写入的内容
- QEMU的函数 ***ld*_be_p*** 用于读取值
    - *处可以是l=long, w=word, u=unsigned, be=大端
- 因此nb_sectors/lba都是取自客户机的***污染***输入
- 还记得之前想要控制的size字段吗？这意味着size字段也是**可污染**的



```
972 static void cmd_read(IDEState *s, uint8_t* buf)
973 {
974     int nb_sectors, lba;
975
976     if (buf[0] == GPCMD_READ_10) {
977         nb_sectors = lduw_be_p(buf + 7);
978     } else {
979         nb_sectors = ldl_be_p(buf + 6);
980     }
981
982     lba = ldl_be_p(buf + 2);
983     if (nb_sectors == 0) {
984         ide_atapi_cmd_ok(s);
985         return;
986     }
987
988     ide_atapi_cmd_read(s, lba, nb_sectors, 2048);
989 }
```

```
static void ide_atapi_cmd_read_pio(IDEState *s, int lba, int nb_sectors,
                                   int sector_size)
{
    s->lba = lba;
    s->packet_transfer_size = nb_sectors * sector_size;
    s->elementary_transfer_size = 0;
    s->io_buffer_index = sector_size;
    s->cd_sector_size = sector_size;
```

从cmd_read到ide_atapi_cmd_read_pio

- 我们已经很接近最终目标了
- cmd_read调用ide_atapi_cmd_read，而我们找到的污染位置是ide_atapi_cmd_read_pio
- 查看代码可知s->atapi_dma为FALSE时，pio会被调用
- 我们之前要求的feature & 11_B不为0其实就是为了将它设置成0

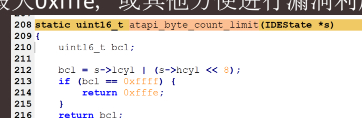
```
434 static void ide_atapi_cmd_read(IDEState *s, int lba, int nb_sectors,
435                                int sector_size)
436 {
437     trace_ide_atapi_cmd_read(s, s->atapi_dma ? "dma" : "pio",
438                              lba, nb_sectors);
439     if (s->atapi_dma) {
440         ide_atapi_cmd_read_dma(s, lba, nb_sectors, sector_size);
441     } else {
442         ide_atapi_cmd_read_pio(s, lba, nb_sectors, sector_size);
443     }
444 }
```

```
1707 static bool cmd_packet(IDEState *s, uint8_t cmd)
1708 {
1709     /* overlapping commands not supported */
1710     if (s->feature & 0x02) {
1711         ide_abort_command(s);
1712         return true;
1713     }
1714
1715     s->status = READY_STAT | SEEK_STAT;
1716     s->atapi_dma = s->feature & 1;
1717     if (s->atapi_dma) {
1718         s->dma_cmd = IDE_DMA_ATAPI;
1719     }
1720     s->nsector = 1;
1721     ide_transfer_start(s, s->io_buffer, ATAPI_PACKET_SIZE,
1722                        ide_atapi_cmd);
1723     return false;
1724 }
```

ide_atapi_cmd_reply_end

- 我们从前后两个方向收敛到了这个可疑的位置，而且发现它的关键参数都是可控的
- 内部还有一些较细粒度的限制，但都可控：
    - 即需要设置lba为-1，来绕过里面的一些检查
    - 单次拷贝的长度受到lcyl、hcyl（即柱面cylinder）的限制
        - 可在FIS中设置为最大0xfffe，或其他方便进行漏洞利用的值

```
/* see if a new sector must l
if (s->lba != -1 && s->io_bu
    if (!s->elementary_trans
        ret = cd_read_sector
```



KCon

代码总览

```
257            byte_count_limit = atapi_byte_count_limit(s);
258            trace_ide_atapi_cmd_reply_end_bcl(s, byte_count_limit);
259            size = s->packet_transfer_size;
260            if (size > byte_count_limit) {
261                /* byte count limit must be even if this case */
262                if (byte_count_limit & 1)
263                    byte_count_limit--;
264                size = byte_count_limit;
265            }
266            s->lcyl = size;
267            s->hcyl = size >> 8;
268            s->elementary_transfer_size = size;
269            /* we cannot transmit more than one sector at a time */
270            if (s->lba != -1) {
271                if (size > (s->cd_sector_size - s->io_buffer_index))
272                    size = (s->cd_sector_size - s->io_buffer_index);
273            }
274            trace_ide_atapi_cmd_reply_end_new(s, s->status);
275        }
276        s->packet_transfer_size -= size;
277        s->elementary_transfer_size -= size;
278        s->io_buffer_index += size;
279
280        /* Some adapters process PIO data right away.  In that case, we need
281         * to avoid mutual recursion between ide_transfer_start
282         * and ide_atapi_cmd_reply_end.
283         */
284        if (!ide_transfer_start_norecurse(s,
285                                          s->io_buffer + s->io_buffer_index - size,
286                                          size, ide_atapi_cmd_reply_end)) {
287            return;
```

最大0xfffe

packet_transfer_size即总共需要拷贝的数量

纵横

KCon

前半部分的总结

- 它调用pio_transfer进行传输
- 但传给pio_transfer的data_ptr就已经越界
- 理论上注册了它的设备都有问题，好在只有AHCI注册了它

```
539 /* prepare data transfer and tell what to do after */
540 bool ide_transfer_start_norecurse(IDEState *s, uint8_t *buf, int size,
541                                   EndTransferFunc *end_transfer_func)
542 {
543     s->data_ptr = buf;
544     s->data_end = buf + size;
545     ide_set_retry(s);
546     if (!(s->status & ERR_STAT)) {
547         s->status |= DRQ_STAT;
548     }
549     if (!s->bus->dma->ops->pio_transfer) {
550         s->end_transfer_func = end_transfer_func;
551         return false;
552     }
553     s->bus->dma->ops->pio_transfer(s->bus->dma);
554     return true;
555 }
```

```
1514
1515 static const IDEDMAOps ahci_dma_ops = {
1516     .start_dma = ahci_start_dma,
1517     .restart = ahci_restart,
1518     .restart_dma = ahci_restart_dma,
1519     .pio_transfer = ahci_pio_transfer,
1520     .prepare_buf = ahci_dma_prepare_buf,
1521     .commit_buf = ahci_commit_buf,
1522     .rw_buf = ahci_dma_rw_buf,
1523     .cmd_done = ahci_cmd_done,
1524 };
```

```
Thread 7 "qemu-system-x86" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7f8db37fe700 (LWP 899)]
0x0000564382f4ffa7 in stw_he_p (ptr=0x564386ecd000, v=0)
    at /home/leonwxqian/exdisk/qemu-4.2.0/include/qemu/bswap.h:345
345         __builtin_memcpy(ptr, &v, sizeof(v));
(gdb) bt
#0  0x0000564382f4ffa7 in stw_he_p (ptr=0x564386ecd000, v=0)
    at /home/leonwxqian/exdisk/qemu-4.2.0/include/qemu/bswap.h:345
#1  0x0000564382f50273 in stn_he_p (ptr=0x564386ecd000, sz=2, v=0)
    at /home/leonwxqian/exdisk/qemu-4.2.0/include/qemu/bswap.h:550
#2  0x0000564382f5862b in flatview_read_continue (fv=0x7f8dac9a9b40,
    addr=15924192603477606642, attrs=...,
    buf=0x564386ecd000 <error: Cannot access memory at address 0x564386ecd000>, len=2048, a
ddr1=15924192603477606642, l=2,
    mr=0x564383cda660 <io_mem_unassigned>)
    at /home/leonwxqian/exdisk/qemu-4.2.0/exec.c:3193
#3  0x0000564382f58763 in flatview_read (fv=0x7f8dac9a9b40,
    addr=15924192603477606642, attrs=...,
    buf=0x564386ecd000 <error: Cannot access memory at address 0x564386ecd000>, len=2048) a
t /home/leonwxqian/exdisk/qemu-4.2.0/exec.c:3230
#4  0x0000564382f587ec in address_space_read_full (as=0x564386c19b88,
    addr=15924192603477606642, attrs=...,
    buf=0x564386ecd000 <error: Cannot access memory at address 0x564386ecd000>, len=2048) a
t /home/leonwxqian/exdisk/qemu-4.2.0/exec.c:3243
#5  0x0000564382f58910 in address_space_rw (as=0x564386c19b88,
    addr=15924192603477606642, attrs=...,
    buf=0x564386ecd000 <error: Cannot access memory at address 0x564386ecd000>, len=2048, i
s_write=false) at /home/leonwxqian/exdisk/qemu-4.2.0/exec.c:3271
#6  0x00005643831 3b1d5 in dma_memory_rw_relaxed (as=0x564386c19b88,
--Type <RET> for more, q to quit, c to continue without paging--
    2603477606642, buf=0x564386ecd000, len=2048, dir=DMA_DIRECTION_TO_DEVICE)
    at /home/leonwxqian/exdisk/qemu-4.2.0/include/sysemu/dma.h:87
#7  0x000056438313b22a in dma_memory_rw (as=0x564386c19b88, addr=15924192603477606642,
    buf=0x564386ecd000, len=2048, dir=DMA_DIRECTION_TO_DEVICE)
    at /home/leonwxqian/exdisk/qemu-4.2.0/include/sysemu/dma.h:110
#8  0x000056438313c38b in dma_buf_rw (
    ptr=0x564386ecd000 <error: Cannot access memory at address 0x564386ecd000>, len=2048,
    sg=0x564386c1c7a0, dir=DMA_DIRECTION_TO_DEVICE) at dma-helpers.c:287
#9  0x000056438313c407 in dma_buf_write (
    ptr=0x564386ecd000 <error: Cannot access memory at address 0x564386ecd000>, len=2048,
    sg=0x564386c1c7a0) at dma-helpers.c:303
#10 0x000056438323 2af8 in ahci_pio_transfer (dma=0x564386c1c3c0) at hw/ide/ahci.c:1450
#11 0x000056438321f283 in ide_transfer_start_norecurse (s=0x564386c1c478,
    buf=0x564386ecd000 <error: Cannot access memory at address 0x564386ecd000>,
    size=2048, end_transfer_func=0x564383226076 <ide_atapi_cmd_reply_end>)
    at hw/ide/core.c:556
#12 0x0000564383226366 in ide_atapi_cmd_reply_end (s=0x564386c1c478) at hw/ide/atapi.c:292
#13 0x0000564383226514 in ide_atapi_cmd_read_pio (s=0x564386c1c478, lba=-1,
    nb_sectors=2048, sector_size=2048) at hw/ide/atapi.c:341
#14 0x00005643832269f6 in ide_atapi_cmd_read (s=0x564386c1c478, lba=-1, nb_sectors=2048,
    sector_size=2048) at hw/ide/atapi.c:461
#15 0x0000564383227b26 in cmd_read (s=0x564386c1c478, buf=0x564386c26000 "\250")
    at hw/ide/atapi.c:1042
#16 0x00005643832285b9 in ide_atapi_cmd (s=0x564386c1c478) at hw/ide/atapi.c:1426
#17 0x000056438321f2d5 in ide_transfer_start (s=0x564386c1c478,
    buf=0x564386c26000 "\250", size=12, end_transfer_func=0x5643832282dc <ide_atapi_cmd>)
    at hw/ide/core.c:565
```

纵横

KCon

但这只是前两步……我们只刚解决了FIS和ATAPI的问题

Solved

Solved

Unsolved

00h

Command FIS (up to 64 bytes)

Software constructed FIS
For data transfer operations
- This is the H2D Register FIS

40h

ATAPI Command (12 to 16 bytes)

Software constructed region that contains the ATAPI command (SCSI).

50h

Unused

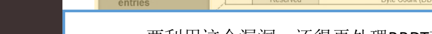
80h

PRDT
Physical Region Descriptor Table
- **Contains the scatter / gather list for the data transfer**
- **Up to 65 535 entries**

DWs	byte n+3	byte n+2	byte n+1	byte n	
0	Data Base Address (DBA)				0
1	Data Base Address Upper 32-bits (DBAU)				
2	Reserved				
3	Reserved	Byte Count (DBC)			1
Item 1	Data Base Address (DBA)				0
	Data Base Address Upper 32-bits (DBAU)				
	Reserved				
	Reserved	Byte Count (DBC)			1

•
•

Last Item	Data Base Address (DBA)				0
	Data Base Address Upper 32-bits (DBAU)				
	Reserved				
	Reserved	Byte Count (DBC)			1



PRDT的使用者——ahci_pio_transfer

- 从cur_cmd中取出opts(options)
- opts的内容决定操作方向是**读还是写**
- ahci_dma_prepare_buf从**PRDT**中取出任务
- 这些任务存放在sglist中
    - 包括地址、长度等信息
- 调用dma_buf_write/read来处理sglist的任务

```
static void ahci_pio_transfer(const IDEDMA *dma)
{
    AHCIDevice *ad = DO_UPCAST(AHCIDevice, dma, dma);
    IDEState *s = &ad->port.ifs[0];
    uint32_t size = (uint32_t)(s->data_end - s->data_ptr);
    /* write == ram -> device */
    uint16_t opts = le16_to_cpu(ad->cur_cmd->opts);
    int is_write = opts & AHCI_CMD_WRITE;
    int is_atapi = opts & AHCI_CMD_ATAPI;
    int has_sglist = 0;
    bool pio_fis_i;

    pio_fis_i = ad->done_first_drq || (!is_atapi && !is_write);
    ahci_write_fis_pio(ad, size, pio_fis_i);

    if (is_atapi && !ad->done_first_drq)
        goto out;

    if (ahci_dma_prepare_buf(dma, size))
        has_sglist = 1;

    if (has_sglist && size) {
        if (is_write) {
            dma_buf_write(s->data_ptr, size, &s->sg);
        } else {
            dma_buf_read(s->data_ptr, size, &s->sg);
        }
    }
```

KCon

sglist的添加过程

- Sglist (Scatter Gather List)被用来存放要读取的操作序列
- 遍历PRDT表，累加PRDT每一项的操作长度，与已操作的长度进行对比
- 当PRDT表累加到某一项时，若已超过已操作的长度，则将其后面的内容加入sglist
- 这里base即为数据基地址

```
qemu_sglist_init(sglist, qbus->parent, (prdtl - off_idx),
                 ad->hba->as);
qemu_sglist_add(sglist, le64_to_cpu(tbl[off_idx].addr) + off_pos,
                MIN(prdt_tbl_entry_size(&tbl[off_idx]) - off_pos,
                    limit));

for (i = off_idx + 1; i < prdtl && sglist->size < limit; i++) {
    qemu_sglist_add(sglist, le64_to_cpu(tbl[i].addr),
                    MIN(prdt_tbl_entry_size(&tbl[i]),
                        limit - sglist->size));
}
```

```
54 void qemu_sglist_add(QEMUSGList *qsg, dma_addr_t base, dma_addr_t len)
55 {
56     if (qsg->nsg == qsg->nalloc) {
57         qsg->nalloc = 2 * qsg->nalloc + 1;
58         qsg->sg = g_realloc(qsg->sg, qsg->nalloc * sizeof(ScatterGatherEntry));
59     }
60     qsg->sg[qsg->nsg].base = base;
61     qsg->sg[qsg->nsg].len = len;
62     qsg->size += len;
63     ++qsg->nsg;
64 }
```

KCon

纵横

AHCI设备的初始化

- 每个SATA控制器有6条主线，每条主线关联有一个设备
- SATA对应的AHCI会初始化6个AHCIDevice结构体，以及其成员要使用的内存

```
1536 void ahci_realize(AHCIState *s, DeviceState *qdev, AddressSpace *as, int ports)
1537 {
1538     qemu_irq *irqs;
1539     int i;
1540
1541     s->as = as;
1542     s->ports = ports;
1543     s->dev = g_new0(AHCIDevice, ports);
1544     ahci_reg_init(s);
1545     irqs = qemu_allocate_irqs(ahci_irq_set, s, s->ports);
1546     for (i = 0; i < s->ports; i++) {
1547         AHCIDevice *ad = &s->dev[i];
1548
1549         ide_bus_new(&ad->port, sizeof(ad->port), qdev, i, 1);
1550         ide_init2(&ad->port, irqs[i]);
1551
1552         ad->hba = s;
1553         ad->port_no = i;
1554         ad->port.dma = &ad->dma;
1555         ad->port.dma->ops = &ahci_dma_ops;
1556         ide_register_restart_cb(&ad->port);
1557     }
1558     g_free(irqs);
1559 }
```

任意长度越界读

- 越界读的是io_buffer之后的数据，长度任意
- 读出来的数据会写入DBA指向的物理内存，因此可以无限泄露QEMU的内存
- 只要不是最后一个port，ide_init1之后通常会有下一个port的硬盘控制结构
- 这个控制结构会包含多个全局变量、函数指针，可以绕过ASLR

KCon

任意长度越界写

- opts也可以设置成写操作

```
is_write = opts & AHCI_CMD_WRITE;
```

- 写操作时，PRDT的DBA地址被当作源地址
- 代码会从DBA读出数据，写入io_buffer后已越界的地址空间中
- 长度、内容不限 ➔ 我们现在有任意长度、任意内容越界读写的原语了

还是那个 io_buffer | 越界写 | ptr

内存布局与堆风水

- 发生溢出的s->io_buffer，长度130KB，在设备初始化（realize）时，被ide_init1申请

```
2553 static void ide_init1(IDEBus *bus, int unit)
2554 {
2555     static int drive_serial = 1;
2556     IDEState *s = &bus->ifs[unit];
2557
2558     s->bus = bus;
2559     s->unit = unit;
2560     s->drive_serial = drive_serial++;
2561     /* we need at least 2k alignment for accessing CDROMs using O_DIRECT */
2562     s->io_buffer_total_len = IDE_DMA_BUF_SECTORS*512 + 4;
2563     s->io_buffer = qemu_memalign(2048, s->io_buffer_total_len);
2564     memset(s->io_buffer, 0, s->io_buffer_total_len);
2565
```

- QEMU在初始化完成后，内存中间可能会有因其他操作留下的无权限的gap
- 越界操作是类似memcpy的连续操作，不可避免会经过gap
- 虽然可以通过技巧避免gap，但是直接选择分配在主heap区的port来利用漏洞更简单

KCon

实际的漏洞利用——布局部分

- 最稳定、最容易定位的当属读取后面的ahci_dma_ops结构体
    - 这个结构体有10个字段，AHCI使用了其中8个
    - 这8个都是函数指针，因此**可以用来绕过ASLR**
- AHCI没有使用set_inactive(#8)和reset(#10)
    - reset通常用于重启时通知IDE设备
    - AHCI/IDE共享代码片段，设置reset后可进入IDE的流程
- 将第10个字段，即.reset改为ROP#1的地址
    - 当QEMU重启时，reset会被调用，从而开始代码执行

```
1515 static const IDEDMAOps ahci_dma_ops = {
1516     .start_dma = ahci_start_dma,
1517     .restart = ahci_restart,
1518     .restart_dma = ahci_restart_dma,
1519     .pio_transfer = ahci_pio_transfer,
1520     .prepare_buf = ahci_dma_prepare_buf,
1521     .commit_buf = ahci_commit_buf,
1522     .rw_buf = ahci_dma_rw_buf,
1523     .cmd_done = ahci_cmd_done,
1524 };
```

```
AHCIDevice *ad = &s->dev[i];

ide_bus_new(&ad->port, sizeof(ad->po
ide_init2(&ad->port, irqs[i]);

ad->hba = s;
ad->port_no = i;
ad->port.dma = &ad->dma;
ad->port.dma->ops = &ahci_dma_ops;
ide_register_restart_cb(&ad->port);
```

纵横

利用时的一些细节

- /x86_64-softmmu/qemu-system-x86_64 -enable-kvm -m 2048
  -device ich9-ahci,id=ahci -drive file=/home/leon/iso.iso,media=cdrom,if=none,id=mycdrom
  -device ide-cd,drive=mycdrom,bus=ahci.4 ➔ 可改为不同的数字，代表不同总线
  -hda /home/leon/disk.qcow2
- MMIO可以使用periphery库简化操作
- 漏洞可以稳定利用，以QEMU权限执行任意代码

KCon

逃逸演示

SimpleScreenRecorder

正在录制

开始录制

启用录制热键　启用声音通知

热键：Ctrl + 　Shift + 　Alt + 　Super + 　R

信息

总时间：0:00:00
输入FPS：0.00
输出FPS：0.00
输入大小：1920x1080
输出大小：?
文件名：?
文件大小：0 B
比特率：0 bit/s

预览

预览帧率：10

注意：预览需要额外的CPU时间（尤其是高帧率的）。

开始预览

日志

[PageRecord::StartPage] Starting page ...
[PageRecord::StartPage] Started page.

取消录制　保存录像

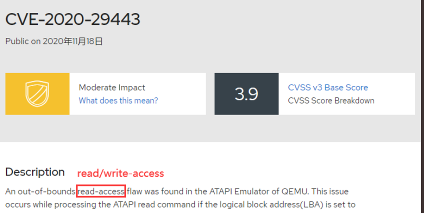
```
leon@ubuntu:~$ sudo python3 ~/Desktop/read.py
```

```
[ALSAInput::GetSourceList] Found ca
[ALSAInput::GetSourceList] Found de
[PulseAudioInput::GetSourceList] Ge
[SourceNamesCallback] Found source:
monitor] Monitor of 内置音频 模拟立
[SourceNamesCallback] Found source:
内置音频 模拟立体声
[PageRecord::StartPage] Starting page ...
[PageRecord::StartPage] Started page.
```

KCon

漏洞的处理

- QEMU官方从源头上修复了问题（禁止CD读取时设置lba为-1）
- 但是CVE提交的信息却是错误的



企业使用或定制化QEMU时的一些思考

- 及时更新补丁
- 不让用户定制过多的启动参数，过多的灵活性可能带来安全问题
- 设置专门人员负责安全审计，将静态扫描的流程集中在开发环节每一个提交中
- 及时处理崩溃或告警信息，合并PATCH并回报官方
- 研制热补丁系统，以方便修补类似于CVE-2020-14364这样补丁不需要改动很多代码的问题

纵横

腾讯宙斯盾
DDoS防护系统

铁将军3.0

BLADE
Tencent Blade

腾讯蓝军
Tencent Force

TSRC
腾讯安全应急响应中心

Onion EDR
洋葱反入侵系统

CodePecker
啄木鸟

門神

K
金刚系统
KING KONG

洞犀
INSIGHT SCANNER

感谢观看！

KCon 汇聚黑客的智慧

KCon