

qemu pwn-hitb gesc 2017 babyqemu writeup

[raycp](#) / 2019-11-08 09:20:18 / 浏览数 5122 [安全技术](#) [CTF 顶\(0\)](#) [踩\(0\)](#)

描述

下载文件，解压后文件结构如下：

```
$ ls -l
total 407504
-rwxr-xr-x@ 1 raycp  staff      281 Jul 11  2017 launch.sh
drwxr-xr-x@ 59 raycp  staff     1888 Jul 11  2017 pc-bios
-rwxr-xr-x@ 1 raycp  staff    39682064 Jul 11  2017 qemu-system-x86_64
-rw-r--r--@ 1 raycp  staff    3864064 Jul 11  2017 rootfs.cpio
-rwxr-xr-x@ 1 raycp  staff    7308672 Jul 11  2017 vmlinuz-4.8.0-52-generic
```

其中launch.sh内容如下：

```
#!/bin/sh
./qemu-system-x86_64 \
-initrd ./rootfs.cpio \
-kernel ./vmlinuz-4.8.0-52-generic \
-append 'console=ttyS0 root=/dev/ram oops=panic panic=1' \
-enable-kvm \
-monitor /dev/null \
-m 64M --nographic -L ./dependency/usr/local/share/qemu \
-L pc-bios \
-device hitb,id=vda
```

分析


首先将设备sudo ./launch.sh运行起来并将qemu-system-x86_64拖到IDA里面进行分析。


运行起来的时候可能会报错如下错误，sudo apt-get install libcurl3即可解决。登录用户名为root，密码为空。


```
./qemu-system-x86_64: /usr/lib/x86_64-linux-gnu/libcurl.so.4: version `CURL_OPENSSL_3' not found (required by ./qemu-system-x86_64)
```


根据命令行参数-device hitb，大概知道了要pwn的目标pci设备是hitb。在IDA里面搜索hitb相关的函数，相关函数列表如下：


Function name


 **do_qemu_init_pci_hitb_register_types**


 **hitb_enc**


 **pci_hitb_register_types**


 **hitb_class_init**


 **pci_hitb_uninit**


 **hitb_instance_init**


 **hitb_obj_uint64**


 **hitb_raise_irq**

 **hitb_fact_thread**

 **hitb_dma_timer**

 **hitb_mmio_write**

 **hitb_mmio_read**

 **pci_hitb_realize**

查看pci_hitb_register_types，知道了该设备所对应的TypeInfo。并且它的class_init函数为hitb_class_init，instance_init函数为hitb_instance_i

其对应的结构体为HitbState：

```
00000000 HitbState      struct ; (sizeof=0x1BD0, align=0x10, copyof_1493)
00000000 pdev           PCIDevice_0 ?
000009F0 mmio          MemoryRegion_0 ?
00000AF0 thread        QemuThread_0 ?
00000AF8 thr_mutex     QemuMutex_0 ?
00000B20 thr_cond      QemuCond_0 ?
00000B50 stopping      db ?
00000B51               db ? ; undefined
00000B52               db ? ; undefined
00000B53               db ? ; undefined
00000B54 addr4         dd ?
00000B58 fact          dd ?
00000B5C status        dd ?
00000B60 irq_status    dd ?
00000B64               db ? ; undefined
00000B65               db ? ; undefined
00000B66               db ? ; undefined
00000B67               db ? ; undefined
00000B68 dma           dma_state ?
00000B88 dma_timer     QEMUTimer_0 ?
00000BB8 dma_buf       db 4096 dup(?)
00001BB8 enc           dq ? ; offset
00001BC0 dma_mask      dq ?
00001BC8               db ? ; undefined
```

```

00001BC9          db ? ; undefined
00001BCA          db ? ; undefined
00001BCB          db ? ; undefined
00001BCC          db ? ; undefined
00001BCD          db ? ; undefined
00001BCE          db ? ; undefined
00001BCF          db ? ; undefined
00001BD0 HitbState ends

```

先看hitb_class_init函数：

```

void __fastcall hitb_class_init(ObjectClass_0 *a1, void *data)
{
    PCIDeviceClass *v2; // rax

    v2 = (PCIDeviceClass *)object_class_dynamic_cast_assert(
        a1,
        "pci-device",
        "/mnt/hgfs/eadom/workspcae/projects/hitbctf2017/babyqemu/qemu/hw/misc/hitb.c",
        469,
        "hitb_class_init");

    v2->revision = 16;
    v2->class_id = 255;
    v2->realize = (void (*)(PCIDevice_0 *, Error_0 **))pci_hitb_realize;
    v2->exit = (PCIUnregisterFunc *)pci_hitb_uninit;
    v2->vendor_id = 0x1234;
    v2->device_id = 0x2333;
}

```

看到它所对应的device_id为0x2333，vendor_id为0x1234。在qemu虚拟机里查看相应的pci设备：

```

# lspci
00:00.0 Class 0600: 8086:1237
00:01.3 Class 0680: 8086:7113
00:03.0 Class 0200: 8086:100e
00:01.1 Class 0101: 8086:7010
00:02.0 Class 0300: 1234:1111
00:01.0 Class 0601: 8086:7000
00:04.0 Class 00ff: 1234:2333

```

00:04.0为相应的hitb设备，不知道为啥lspci命令没有-v选项，要查看I/O信息，查看resource文件：

```

# cat /sys/devices/pci0000\:00\0000\:00\:04.0/resource
0x00000000fea00000 0x00000000feafffff 0x0000000000040200
0x0000000000000000 0x0000000000000000 0x0000000000000000

```

resource文件内容的格式为start_address end_address
flag，根据flag最后一位可知存在一个MMIO的内存空间，地址为0x00000000fea00000，大小为0x100000

查看pci_hitb_realize函数：

```

void __fastcall pci_hitb_realize(HitbState *pdev, Error_0 **errp)
{
    pdev->pdev.config[61] = 1;
    if ( !msi_init(&pdev->pdev, 0, 1u, 1, 0, errp) )
    {
        timer_init_tl(&pdev->dma_timer, main_loop_tlg.tl[1], 1000000, (QEMUTimerCB *)hitb_dma_timer, pdev);
        qemu_mutex_init(&pdev->thr_mutex);
        qemu_cond_init(&pdev->thr_cond);
        qemu_thread_create(&pdev->thread, "hitb", (void (*)(void *))hitb_fact_thread, pdev, 0);
        memory_region_init_io(&pdev->mmio, &pdev->pdev.qdev.parent_obj, &hitb_mmio_ops, pdev, "hitb-mmio", 0x100000uLL);
        pci_register_bar(&pdev->pdev, 0, 0, &pdev->mmio);
    }
}

```

函数首先注册了一个[timer](#)，处理回调函数为hitb_dma_timer，接着注册了hitb_mmio_ops内存操作的结构体，该结构体中包含hitb_mmio_read以及hitb_mmio_write

接下来仔细分析hitb_mmio_read以及hitb_mmio_write函数。

hitm_mmio_read函数没有什么关键的操作，主要就是通过addr去读取结构体中的相应字段。

关键的在hitm_mmio_write函数中，关键代码部分如下：

```

void __fastcall hitb_mmio_write(HitbState *opaque, hwaddr addr, uint64_t value, unsigned int size)
{
    uint32_t v4; // er13
    int v5; // edx
    bool v6; // zf
    int64_t v7; // rax

    if ( (addr > 0x7F || size == 4) && (!((size - 4) & 0xFFFFFFFFB) || addr <= 0x7F) )
    {
        if ( addr == 0x80 )
        {
            if ( !(opaque->dma.cmd & 1) )
                opaque->dma.src = value; // 0x80 set src
        }
        else
        {
            v4 = value;
            if ( addr > 128 )
            {
                if ( addr == 140 )
                {
                    ...
                }
                else if ( addr > 0x8C )
                {
                    if ( addr == 144 )
                    {
                        if ( !(opaque->dma.cmd & 1) )
                            opaque->dma.cnt = value; // 144 set cnt
                    }
                    else if ( addr == 152 && value & 1 && !(opaque->dma.cmd & 1) )
                    {
                        opaque->dma.cmd = value; // 152 set cmd
                        v7 = qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL_0);
                        timer_mod(
                            &opaque->dma_timer,
                            ((signed __int64)((unsigned __int128)(0x431BDE82D7B634DBLL * (signed __int128)v7) >> 64) >> 18) //trigger ti
                            - (v7 >> 63)
                            + 100);
                    }
                }
                ...
            }
            else if ( addr == 136 && !(opaque->dma.cmd & 1) )
            {
                opaque->dma.dst = value; // 136 set dst
            }
        }
        ...
    }
}

```

关键操作包括：

1. 当addr为0x80的时候，将value赋值给dma.src。
2. 当addr为144的时候，将value赋值给dma.cnt。
3. 当addr为152的时候，将value赋值给dma.cmd，并触发timer。
4. 当addr为136的时候，将value赋值给dma.dst。

可以看到hitb_mmio_write函数基本上是通过addr将设备结构体中的dma字段赋值，dma的定义为：

```

00000000 dma_state      struc ; (sizeof=0x20, align=0x8, copyof_1491)
00000000                                ; XREF: HitbState/r
00000000 src            dq ?
00000008 dst            dq ?
00000010 cnt            dq ?
00000018 cmd            dq ?
00000020 dma_state      ends

```

再去看看timer触发之后的操作，即hitb_dma_timer函数：

```

void __fastcall hitb_dma_timer(HitbState *opaque)
{
    dma_addr_t cmd; // rax
    __int64 idx; // rdx
    uint8_t *addr; // rsi
    dma_addr_t v4; // rax
    dma_addr_t v5; // rdx
    uint8_t *v6; // rbp
    uint8_t *v7; // rbp

    cmd = opaque->dma.cmd;
    if ( cmd & 1 )
    {
        if ( cmd & 2 )
        {
            idx = (unsigned int)(LODWORD(opaque->dma.src) - 0x40000);
            if ( cmd & 4 )
            {
                v7 = (uint8_t *)&opaque->dma_buf[idx];
                ((void (__fastcall *) (uint8_t *, _QWORD))opaque->enc)(v7, LODWORD(opaque->dma.cnt));
                addr = v7;
            }
            else
            {
                addr = (uint8_t *)&opaque->dma_buf[idx];
            }
            cpu_physical_memory_rw(opaque->dma.dst, addr, opaque->dma.cnt, 1);
            v4 = opaque->dma.cmd;
            v5 = opaque->dma.cmd & 4;
        }
        else
        {
            v6 = (uint8_t *)&opaque[0xFFFFFDBLL].dma_buf[(unsigned int)opaque->dma.dst + 0x510];
            LODWORD(addr) = (_DWORD)opaque + opaque->dma.dst - 0x40000 + 0xBB8;
            cpu_physical_memory_rw(opaque->dma.src, v6, opaque->dma.cnt, 0);
            v4 = opaque->dma.cmd;
            v5 = opaque->dma.cmd & 4;
            ...
        }
    }
}

```

可以看到主要操作包含三部分：

1. 当dma.cmd为2|1时，会将dma.src减0x40000作为索引i，然后将数据从dma_buf[i]拷贝利用函数cpu_physical_memory_rw拷贝至物理地址dma.dst中，拷贝长度为dma.cnt。
2. 当dma.cmd为4|2|1时，会将dma.dst减0x40000作为索引i，然后将起始地址为dma_buf[i]，长度为dma.cnt的数据利用opaque->enc函数加密后，再调用cpu_physical_memory_rw将加密后的数据拷贝到物理地址dma.dst中。
3. 当dma.cmd为0|1时，调用cpu_physical_memory_rw将物理地址中为dma.dst，长度为dma.cnt，拷贝到dma.dst减0x40000作为索引i，目标地址为dma_buf[i]。

到这里基本上可以看出这个设备的功能，主要是实现了一个dma机制。DMA(Direct Memory Access，直接内存存取)是所有现代电脑的重要特色，它允许不同速度的硬件装置来沟通，而不需要依赖于CPU的大量中断负载。DMA传输将数据从一个地址空间复制到另外一个地址空间。当CPU初始化这个传输动作，传输动作本身是由DMA控制器来实行和完成。

即首先通过访问mmio地址与值(addr与value)，在hitb_mmio_write函数中设置好dma中的相关值(src、dst以及cmd)。当需要dma传输数据时，设置addr为152，调用hitb_dma_timer函数，该函数根据dma.cmd的不同调用cpu_physical_memory_rw函数将数据从物理地址拷贝到dma_buf中或从dma_buf拷贝到物理地址中。

功能分析完毕，漏洞在哪儿呢？我们可以看到hitb_dma_timer中拷贝数据时dma_buf中的索引是可控的，且没有限制。因此我们可以通过设置其相应的值导致越界读写，从而利用该漏洞。

利用

整个利用流程包括：

1. 首先是越界读的内容，往dma_buf往后看到了enc指针，可以读取该指针的值以实现地址泄露。泄露地址后根据偏移，可以得到程序基址，然后计算得到system plt地址。
2. 将参数cat /root/flag写入到buf_buf中。
3. 其次是越界写的内容，我们可以将system plt地址写入到enc指针，最后触发enc函数实现system函数的调用，实现system("cat /root/flag")。

需要指出的一点是cpu_physical_memory_rw是使用的物理地址作为源地址或目标地址，因此我们需要先申请一段内存空间，并将其转换至其物理地址。虚拟地址转换到物理地址可以通过/proc/\$pid/pagemap实现转换。

动态调试

我一开始也尝试往启动脚本中加入-netdev
user,id=net0,hostfwd=tcp::5555-:22来实现ssh的端口转发，然后将exp通过scp传上去。但是结果失败了，只能想其它办法。

因为这是使用cpio作为文件系统的，所以可以先将该文件系统解压，然后将exp放入其中，最后再启动虚拟机。

首先是解压文件：

```
1. gunzip XXX.cpio.gz
2. cpio -idmv < XXX.cpio
```

然后将exp.c编写好，放到解压出来的文件夹里。运行make命令，编译exp并重打包cpio，makefile内容如下：

```
ALL:
    gcc -O0 -static -o exp exp.c
    find . | cpio -o --format=newc > ../rootfs.cpio
```

为了方便调试可以先sudo gdb ./qemu-system-x86_64调试进程，下好断点后再用下面的命令启动虚拟机：

```
pwdnbg> r -initrd ./rootfs.cpio -kernel ./vmlinuz-4.8.0-52-generic -append 'console=ttyS0 root=/dev/ram oops=panic panic=1' -e
```

再提一句，直接在gdb里面最后执行system起一个新进程的时候可能会报下面的错误。不要以为exp没写对，要是看到了执行到system并且参数也对了，不用gdb调试，直

```
# [New process 4940]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
process 4940 is executing new program: /bin/dash
/build/gdb-JPMZNV/gdb-8.1/gdb/breakpoint.c:13230: internal-error: void delete_breakpoint(breakpoint*): Assertion `bpt != NULL'
A problem internal to GDB has been detected,
further debugging may prove unreliable.
```

```
This is a bug, please report it.  For instructions, see:
<http://www.gnu.org/software/gdb/bugs/>.
```

```
[1] 4926 abort      sudo gdb ./qemu-system-x86_64
```

小结

其实对于qemu的timer以及dma都还不太清楚，后面也还需要再学习。学习qemu pci设备也可以看qemu的edu设备：[edu.c](#)

相关文件以及脚本[链接](#)

参考链接

- 1. [HITB GSEC 2017: babyqemu](#)
- 2. [DMA（直接存储器访问）](#)
- 3. [QEMU timer模块分析](#)
- 4. [edu.c](#)

点击收藏 | 1 关注 | 1

[上一篇：论文件上传绕过的各种姿势（二）](#) [下一篇：某Shop供应商后台SQL Inj...](#)

- 1. 0 条回复
 - 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

