Virtunoid: Breaking out of KVM

Nelson Elhage

DEFCON 19

August 8, 2011
KVM

- The new hotness for Virtualization on Linux
- Official virtualization platform for Ubuntu and RHEL.
**Who am I?**

- Kernel engineer at Ksplice (now Oracle).
- Open-source security hacker in my spare time.
Outline

1. KVM: Architecture overview
   - Attack Surface

2. CVE-2011-1751: The bug

3. virtunoid.c: The exploit
   - %rip control
   - Getting to shellcode
   - Bypassing ASLR

4. Conclusions and further research

5. Demo
KVM: Architecture overview

- Attack Surface

CVE-2011-1751: The bug

virtunoid.c: The exploit
- %rip control
- Getting to shellcode
- Bypassing ASLR

Conclusions and further research

Demo
KVM: The components

- kvm.ko
- kvm-intel.ko / kvm-amd.ko
- qemu-kvm
**kvm.ko**

- The core KVM kernel module
- Implements the virtual CPU and MMU (with the hardware’s help).
- Emulates a few devices in-kernel for efficiency.
- Provides `ioctl`s for communicating with the kernel module.
- Contains an emulator for a subset of x86 used in handling certain traps (!)
kvm-intel.ko / kvm-amd.ko

- Provides support for Intel’s VMX and AMD’s SVM virtualization extensions.
- Relatively small compared to the rest of KVM (one .c file each)
qemu-kvm

- Provides the most direct user interface to KVM.
- Based on the classic qemu emulator.
- Implements the bulk of the virtual devices a VM uses.
- Implements a wide variety of types of devices.
- An order of magnitude more code than the kernel module.
- There is work in progress to replace this component, but it’s a ways out, if ever.
kvm.ko

- A tempting target – successful exploitation gets ring0 on the host without further escalation.
- Much less code than qemu-kvm, and much of that is dedicated to interfacing with qemu-kvm, not the guest directly.
- The x86 emulator is an interesting target.
  - A number of bugs have been discovered allowing privesc within the guest.
  - A lot of tricky code that is not often exercised.
  - Not the target of this talk, but I have some ideas for future work.
- Also, be on the lookout for privesc within either the host or guest.
Not much direct attack surface.

Largely straight-line code doing lots of low-level bit twiddling with the hardware structures.

Lots of subtlety, possibly some more complex attacks.
qemu-kvm

- A veritable goldmine of targets.
- Hundreds of thousands of lines of device emulation code.
- Emulated devices communicate directly with the guest via MMIO or IO ports, lots of attack surface.
- Much of the code comes straight from qemu and is ancient.
- qemu-kvm is often sandboxed using SELinux or similar, meaning that successful exploitation will often require a second privesc within the host.
  - (Fortunately, Linux never has any of those)
1 KVM: Architecture overview
   - Attack Surface

2 CVE-2011-1751: The bug

3 virtunoid.c: The exploit
   - %rip control
   - Getting to shellcode
   - Bypassing ASLR

4 Conclusions and further research

5 Demo
“It was found that the PIIX4 Power Management emulation layer in qemu-kvm did not properly check for hot plug eligibility during device removals. A privileged guest user could use this flaw to crash the guest or, possibly, execute arbitrary code on the host. (CVE-2011-1751)”
The PIIX4 was a southbridge chip used in many circa-2000 Intel chipsets. The default southbridge emulated by qemu-kvm includes ACPI support, a PCI-ISA bridge, an embedded MC146818 RTC, and much more.
Device Hotplug

- The PIIX4 supports PCI hotplug, implemented by writing values to IO port 0xae08.
- qemu-kvm emulates this by calling qdev_free(qdev);, which calls a device’s cleanup function and free()s it.
- Many devices weren’t implemented with hotplug in mind!
The PCI-ISA bridge

- In particular, it should not be possible to unplug the ISA bridge.
- Among other things, the emulated MC146818 RTC hangs off the ISA bridge.
- KVM’s emulated RTC is not designed to be unplugged; In particular, it leaves around dangling QEMUTimer objects when unplugged.
QEMUTimer

typedef void QEMUTimerCB(void *opaque);

struct QEMUTimer {
    ...
    int64_t expire_time; /* in nanoseconds */
    QEMUTimerCB *cb;
    void *opaque;
    struct QEMUTimer *next;
};

typedef struct RTCState {
    ...
    QEMUTimer *second_timer;
    ...
} RTCState;
Use-after-free

- Unplugging the virtual RTC `free()`s the RTCState
- It doesn't `free()` or unregister either of the timers.
- So we're left with dangling pointers from the QEMU Timers
- On the next second, we'll call `rtc_update_second(<freed RTCState>)`
Reproducer

```c
#include <sys/io.h>

int main (void) {
    iopl(3);
    outl(2, 0xae08);
    return 0;
}
```
KVM: Architecture overview
  - Attack Surface

CVE-2011-1751: The bug

virtunoid.c: The exploit
  - %rip control
  - Getting to shellcode
  - Bypassing ASLR

Conclusions and further research

Demo
High-level TODO

1. Inject a controlled QEMUTimer into qemu-kvm at a known address
2. Eject the emulated ISA bridge
3. Force an allocation into the freed RTCState, with second_timer pointing at our dummy timer.

- When rtc_update_second next runs, our timer will get scheduled.
- One second later, boom.
1. Injecting data

- The guest’s RAM is backed by a simple `mmap()`ed region inside the qemu-kvm process.
- So we allocate an object in the guest, and compute
  - `hva = physmem_base + gpa`
  - `gpa = (gva_to_gfn(gva) << PAGE_SHIFT) + page_offset(gva)`
- For now, assume we can guess `physmem_base` (e.g. no ASLR)

**hva** host virtual address

**gva** guest virtual address

**gpa** guest physical address

**gfn** guest frame (physical page) number
qemu-kvm userspace network stack

- qemu-kvm contains a user-mode networking stack.
- Implements a DHCP server, DNS server, and a gateway NAT.
Userspace network stack packet delivery

- The user-mode stack normally handles packets synchronously.
- To prevent recursion, if a second packet is emitted while handling a first packet, the second packet is queued, using malloc().
The virtual network gateway responds synchronously to ICMP ping.
Putting it together

1. Allocate a fake QEMU Timer
   - Point \texttt{->cb} at the desired %rip.
2. Calculate its address in the host.
3. Write 2 to IO port 0xae08 to eject the ISA bridge.
4. Ping the emulated gateway with ICMP packets containing pointers to your allocated timer in the host.
We’ve got `%rip, now what?"
We've got `%rip, now what?

- Get EIP = 0x41414141 and declare victory.
We've got `%rip, now what?

- Get EIP = 0x41414141 and declare victory.
- Disable NX in my BIOS and call it good enough for a demo.
We've got `%rip`, now what?

- Get EIP = 0x41414141 and declare victory.
- Disable NX in my BIOS and call it good enough for a demo.
- Do a ROP pivot, ROP to victory.
We've got %rip, now what?

- Get EIP = 0x41414141 and declare victory.
- Disable NX in my BIOS and call it good enough for a demo.
- Do a ROP pivot, ROP to victory.
- Do something else clever.
Another look at QEMUTimer

```c
struct QEMUTimer {
    ...
    int64_t expire_time; /* in nanoseconds */
    ...
    struct QEMUTimer *next;
};
```
qemu_run_timers

static void qemu_run_timers(QEMUClock *clock)
{
    QEMUTimer **ptimer_head, *ts;
    int64_t current_time;

    current_time = qemu_get_clock_ns(clock);
    ptimer_head = &active_timers[clock->type];
    for (;;) {
        ts = *ptimer_head;
        if (!qemu_timer_expired_ns(ts, current_time))
            break;
        *ptimer_head = ts->next;
        ts->next = NULL;

        ts->cb(ts->opaque);
    }
}
Timer chains

\[ \Rightarrow f1(X); f2(Y); f3(Z); \]
More arguments

- amd64 calling convention: `%rdi, %rsi, %rdx, ...
- Every version of `qemu_run_timers` I’ve checked leaves `%rsi` untouched.
More arguments

- **set_rsi**:  
  
  ```
  movl %rdi, %rsi
  ret
  ```

- Let $f_1 = \text{set}_\text{rsi}$
- $f_2(Y, X)$
- Same trick doesn’t work with %rdx.
set_rsi

```c
void cpu_outl(pio_addr_t addr, uint32_t val)
{
    ioprt_write(2, addr, val);
}
```
Getting to mprotect

```c
int mprotect(const void *addr, size_t len, int prot);
#define PROT_EXEC 0x4

static uint32_t ioprt_readl_thunk(void *opaque, uint32_t addr)
{
    IORange *ioprt = opaque;
    uint64_t data;

    ioprt->ops->read(ioprt, addr - ioprt->base, 4, &data);
    return data;
}
```
Putting it together

- Allocate a fake IORangeOps, with `fake_ops->read = mprotect`.
- Allocate a page-aligned IORange, with `->ops = fake_ops` and `->base = -PAGE_SIZE`.
- Copy shellcode immediately following the IORange.
- Construct a timer chain that calls
  - `cpu_outl(0, *)`
  - `ioport_readl_thunk(fake_ioport, 0)`
  - `fake_ioport + 1`
Why not ROP?

- Continued execution is dead simple.
- Reduced dependence on details of compiled code.
- I’m not that good at ROP :)
Addresses

- For a known qemu-kvm binary, we need two addresses.
  - The base address of the qemu-kvm binary, to find code addresses.
  - physmem_base, the address of the physical memory mapping inside qemu-kvm.
Option A

- Find an information leak.
Option B

- Assume non-PIE, and be clever.
fw_cfg

- Emulated IO ports 0x510 (address) and 0x511 (data)
- Used to communicate various tables to the qemu BIOS (e820 map, ACPI tables, etc)
- Also provides support for exporting writable tables to the BIOS.
- However, fw_cfg_write doesn’t check if the target table is supposed to be writable!
Static data

- Several fwCfg areas are backed by statically-allocated buffers.
- Net result: nearly 500 writable bytes inside static variables.
read4 your way to victory

- mprotect needs a page-aligned address, so these aren’t suitable for our shellcode.
- But, we can construct fake timer chains in this space to build a read4() primitive.
- Follow pointers from static variables to find physmem_base
- Proceed as before.
Repeated timer chaining

- Previously, we ended timer chains with ->next = NULL.
- Instead, end them with a timer that calls rtc_update_second.
- The timer we control will be scheduled once a second, and we can change ->cb at any time.
- Now we can execute a read4, update structures based on the result, and then hijack the list again.
Conclusions

- VM breakouts aren’t magic.
- Hypervisors are just as vulnerable as anything else.
- Device drivers are the weak spot.
Comparing with some past breakouts

2008  “Adventures with a certain Xen vulnerability”, Xen, Invisible Things Lab
2009  “Cloudburst”, Immunity, VMware
2011  “Software attacks against Intel VT-d technology”, Invisible Things Lab, Xen
Possible hardening directions

- Sandbox qemu-kvm (work underway well before this talk).
- Build qemu-kvm as PIE.
- Lazily mmap/mprotect guest RAM?
- XOR-encode key function pointers?
- More auditing and fuzzing of qemu-kvm.
Future research directions

- Fuzzing/auditing `kvm.ko` (That x86 emulator sketches me)
- Fingerprinting `qemu-kvm` versions
- Searching for infoleaks (Rosenbugs?)
It’s demo time
Questions?

- nelhage@nelhage.com
- http://blog.nelhage.com
- @nelhage