Virtunoid: Breaking out of KVM

Nelson Elhage

July 25, 2011

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: The components

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

`kvm.ko`

- The core KVM kernel module
- Provides `ioctls` for communicating with the kernel module.
- Primarily responsible for emulating the virtual CPU and MMU
- Emulates a few devices in-kernel for efficiency.
- 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` x86 emulator.
- Implements the bulk of the virtual devices a VM uses.
- Implements a wide variety of possible devices and buses.
- An order of magnitude more code than the kernel module.

Control flow

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.

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)
- Lots of bugs have been found here.

RHSA-2011:0534-1

"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)"

```
diff --git a/hw/acpi_piix4.c b/hw/acpi_piix4.c
index 96f5222..6c908ff 100644
--- a/hw/acpi_piix4.c
+++ b/hw/acpi_piix4.c
@@ -471,11 +471,13 @@ static void pciej_write(void *opaque, uint32_t addr, uint32_t val)
     BusState *bus = opaque;
     DeviceState *qdev, *next;
     PCIDevice *dev;
+    PCIDeviceInfo *info;
     int slot = ffs(val) - 1;

     QLIST_FOREACH_SAFE(qdev, &bus->children, sibling, next) {
         dev = DO_UPCAST(PCIDevice, qdev, qdev);
-        if (PCI_SLOT(dev->devfn) == slot) {
+        info = container_of(qdev->info, PCIDeviceInfo, qdev);
+        if (PCI_SLOT(dev->devfn) == slot && !info->no_hotplug) {
             qdev_free(qdev);
         }
     }
```

PIIX4

- 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 is supposed to make sure the device is properly disconnected.
- Certain devices don't properly support being hotplugged, but KVM previously didn't check this before freeing them.

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.

The real-time clock

```
typedef struct RTCState {
  uint8_t cmos_data[128];
  ...
  /* second update */
  int64_t next_second_time;
  ...
  QEMUTimer *second_timer;
  QEMUTimer *second_timer2;
} RTCState;
```

The real-time clock

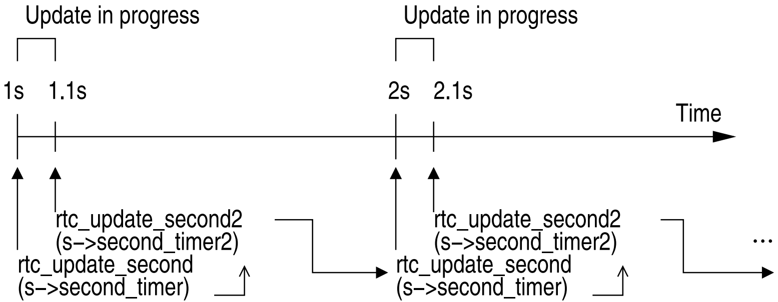
```
static int rtc_initfn(ISADevice *dev)
{
  RTCState *s = DO_UPCAST(RTCState, dev, dev);
  ...
  s->second_timer = qemu_new_timer_ns(rtc_clock, rtc_update_second, s);
  s->second_timer2 = qemu_new_timer_ns(rtc_clock, rtc_update_second2, s);

  s->next_second_time =
      qemu_get_clock_ns(rtc_clock) + (get_ticks_per_sec() * 99) / 100;
  qemu_mod_timer(s->second_timer2, s->next_second_time);
  ...
}
```

QEMUTimer

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

RTC timers


Update in progress
Update in progress
1s
1.1s
2s
2.1s
Time
rtc_update_second2
(s–>second_timer2)
rtc_update_second
(s–>second_timer)
rtc_update_second2
(s–>second_timer2)
rtc_update_second
(s–>second_timer)
...

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 `QEMUTimers`

High-level TODO

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

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
  + (gva_to_gfn(gva) << PAGE_SHIFT)
  + page_offset(hva)
  ```

  hva host virtual address
  gva guest virtual address
  gfn guest frame (physical page) number
- For now, assume we can guess physmem_base (e.g. no ASLR)
  ```

fs/proc/task_mmu.c

```
/*
 * /proc/pid/pagemap - an array mapping virtual pages to pfns
 *
 * For each page in the address space, this file contains
 * one 64-bit entry consisting of the following:
 *
 * Bits 0-55  page frame number (PFN) if present
 * Bits 0-4   swap type if swapped
 * Bits 5-55  swap offset if swapped
 * Bits 55-60 page shift (page size = 1<<page shift)
 * Bit  61    reserved for future use
 * Bit  62    page swapped
 * Bit  63    page present
 */
```

qemu-kvm userspace network stack

- `qemu-kvm` contains a user-mode networking stack.
- Implements a DHCP server, DNS server, and a gateway NAT.
- 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 to ICMP ping.

Putting it all together

1. Allocate a fake `QEMUTimer`
   - Point `->cb` at the desired `%rip`.
   - Set `->expire` to something small (e.g. 0).
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?

Options:

- 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.
- ????

Another look at `QEMUTimer`

```
struct QEMUTimer {
    ...
    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

- We don't just control %rip – we control a QEMUTimer object that is going to get dispatched by qemu_run_timers.
- In particular, we can control ->next.
- So we can chain fake timers, and make multiple one-argument calls in a row.
- We can fake other structs to get the first argument.
- qemu_run_timers doesn't touch %rsi in any version of qemu-kvm I've examined.

Getting to `mprotect`

- Find a function ("F") that makes a three-arg function call based on struct(s) passed as arguments one and two.
- Construct appropriate fake structures.
- Construct a timer chain that
  - Does a call to set up `%rsi` based on a first argument in `%rdi`.
  - Does a call to F that `mprotect()`s one or more pages in the guest physmem map.
  - Calls shellcode stored in those pages.

Why this trickery?

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

Addresses

- We need at least 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!

hw/pc.c

```
static struct e820_table e820_table;
struct hpet_fw_config hpet_cfg = {.count = UINT8_MAX};

...
    fw_cfg = fw_cfg_init(BIOS_CFG_IOPORT, BIOS_CFG_IOP

    fw_cfg_add_bytes(fw_cfg, FW_CFG_E820_TABLE, (uint8
                     sizeof(struct e820_table));

    fw_cfg_add_bytes(fw_cfg, FW_CFG_HPET, (uint8_t *)&
                     sizeof(struct hpet_fw_config));
```

read4 your way to victory

- Net result: nearly 500 writable bytes inside a static variable.
- `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.
- Use that 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` to reschedule the timer every second.
- Now we can execute a `read4`, update structures based on the result, and then hijack the list again.

Possible hardening directions

- Sandbox `qemu-kvm` (work underway well before this talk).
- Build `qemu-kvm` as PIE.
- Keep memory in a file in tmpfs and lazily `mmap` as-needed for DMA?
- 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?