Virtunoid: A KVM Guest → Host privilege escalation exploit

Nelson Elhage

Black Hat USA 2011

1 Introduction

KVM, the Linux Kernel-based Virtual Machine, is a virtualization solution for Linux designed to be tightly integrated with upstream kernel development. While still a relatively recent player on the landscape, KVM seems destined to become the new standard for open-source virtualization, with both Ubuntu and Red Hat Enterprise Linux adopting it as the basis of their standard virtualization offerings.

This paper discusses the development of a guest-to-host VM escape exploit for KVM. As the first public example of such an exploit, we hope to provide a case study demonstrating some of the difficulties and techniques that come into play when constructing such an exploit. Our attack demonstrates a successful bypass of both ASLR and non-executable pages against KVM.

2 KVM structure and attack surface

KVM, like most virtualization tools, is separated into a userspace and a kernel-mode component. The kernel component, implemented in the `kvm.ko` and one of the `kvm-intel.ko` or `kvm-amd.ko` modules, deals with low-level CPU and memory emulation, using either the Intel VMX or AMD SVM hardware virtualization extensions. The userspace program, `qemu-kvm`, is based on the venerable QEMU [1] CPU emulator. `qemu-kvm` provides the basic user interface for launching and controlling VMs, and provides the main framework for running VMs. `qemu-kvm` handles essentially all hardware emulation – when the emulated machine performs an IO instruction (such as an operation on x86 IO ports), the kernel module returns to userspace, where `qemu-kvm` emulates the operation and then makes a `ioctl` to resume execution. (For performance reasons, a few devices are actually emulated directly in-kernel, but that detail is not relevant to this work).

The KVM kernel module must be considered a tempting target for an attacker, as a compromise there has the potential to yield privileged ring 0 execution on the host, obviating the need for further privilege escalation once inside

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

Figure 1: The definition of `qemu-kvm`'s `struct RTCState`.

the host. However, the kernel module contains an order of magnitude less code than the userspace component, and much of that is not directly attackable from the guest. Thus, we expect that the majority of potentially-interesting KVM bugs will be located in the `qemu-kvm` component, and it is that component this paper will discuss attacking.

3 The bug

The bug we chose to exploit, CVE-2011-1751, is a result of a missing check in KVM's emulation of PCI device hotplugging. In particular, KVM failed to ignore untplug requests from the guest hardware for devices that did not support being unplugged. Such devices, when unplugged, may fail to adequately clean up and disconnect themselves, leaving behind corrupt state or dangling pointers.

The particular device we chose to unplug was the emulated Intel PIIX4 chip, including the PCI-to-ISA bridge. Unplugging this chip unplugs all emulated ISA devices, including the emulated real-time clock. The real-time clock, as part of normal operation, registers timer callbacks with `qemu-kvm`'s internal run loop. When it is destroyed and freed, these timers are not cleaned up, resulting in an exploitable use-after-free condition.

Triggering this condition is as simple as writing the value 2 out the x86 IO port number `0xae08` ("`PCI_EJ_BASE`" in the `qemu-kvm` source).

4 Use-after-free

The `qemu-kvm` emulated MC146818 real-time-clock is implemented in the file `hw/mc146818rtc.c`. The emulated state is stored in a `struct RTCState`, whose definition is shown in Figure 1.

A `QEMUTimer` is used to register timer callbacks with `qemu-kvm` run loop. Under normal operation, the RTC emulation uses `second_timer` and `second_timer2` to implement the once-per-second update of the real-time clock. `next_second_time` keeps track of the time that the next RTC update is scheduled.

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

int main (void) {
    iopl(3);
    outl(2, 0xae08);
    return 0;
}
```

Figure 2: Simple program to demonstrate the RTC use-after-free bug.

```
static void rtc_update_second(void *opaque)
{
    RTCState *s = opaque;
    int64_t delay;

    /* if the oscillator is not in normal operation, we do not update */
    if ((s->cmos_data[RTC_REG_A] & 0x70) != 0x20) {
        s->next_second_time += get_ticks_per_sec();
        qemu_mod_timer(s->second_timer, s->next_second_time);
    } else {
      /* Arm s->second_timer2 */
      ...
    }
}
```

Figure 3: `rtc_update_second`, showing the conditional re-arming of `second_timer2`

The RTC arranges the timers such that at that time, `second_timer` fires, triggering a call to `rtc_update_second`. `rtc_update_second` updates the RTC state to indicate that an update is in-progress, and then arranges for `second_timer2` to fire one-hundredth of a second later, calling `rtc_update_second2`, which completes the update, advances `next_second_time`, and re-schedules `rtc_update_second`.

When the emulated RTC clock is unplugged, the backing `RTCState` is freed, but neither time is freed or unregistered from the main loop. Since the `free()` implementation in glibc overwrites the final word of the freed area for bookkeeping purposes, and since `second_timer2` is the final element of the `RTCState` structure, this will result in a segmentation fault the next time `rtc_update_second` runs and attempts to re-arm it. Figure 2 contains a minimal C program that, when run as `root` from within a Linux guest, should result in such a crash.

If we examine `rtc_update_second` (Figure 3), we discover that the RTC only arms `second_timer2` if the emulated RTC is in certain configurations, namely when `(s->cmos_data[RTC_REG_A] & 0x70) == 0x20`. This, if we write, say, 0x70 into CMOS address 10 (RTC_REG_AS), rtc_update_second will simply continue to fire every second, using s->second_timer. Since freeing the RTCState will not clobber that pointer, this behavior will continue even after the free(), giving us a relatively wide window during which we can force a controlled allocation over the freed RTCState, allowing us to hijack the second_timer pointer and begin taking control.

5 Forcing controlled allocations

In order to take control of the freed RTCState struct, we need to force an allocation of the same size as the freed structure, with mostly-controlled contents.

For the purposes of this case study, we chose to use a feature of the qemu-kvm user-mode networking stack.

5.1 The QEMU/KVM user-mode network stack

qemu-kvm implements a user-mode network backend as the default setting for providing network access to a virtual machine. The user-mode network stack implements an entire VLAN inside the qemu-kvm process, including a DHCP server and a NAT gateway that allows the virtual machine to access the outside network.

5.2 The NetQueue abstraction

One of the key abstractions of the user-mode network stack is the NetQueue struct, which implements a unidirectional packet queue. A NetQueue has a callback to be invoked on new packets, and a queue of pending packets. A packet can be enqueued on a NetQueue using the qemu_net_queue_send function, as shown in Figures 4 and 5.

The relevant detail here is that, qemu_net_queue_send will normally deliver an incoming packet synchronously without queuing, but to limit recursing, if a second packet is delivered while a first packet is still being processed, that second packet will be enqueued for later processing, using qemu_malloc to allocate space. That allocation will consist of a NetPacket header, plus the packet data, and so, if we can control the packet, we have a large degree of control over the allocation.

5.3 ICMP ping

However, as mentioned, this queueing behavior only takes place if a packet is delivered recursively as a result of processing a first packet. And so, we need one more piece in order to cause a controlled allocation: Some service on the qemu-kvm virtual network that will synchronously generate packets in response to packets the guest sends, in a controllable manner.

```
ssize_t qemu_net_queue_send(NetQueue *queue,
                            VLANClientState *sender,
                            unsigned flags,
                            const uint8_t *data,
                            size_t size,
                            NetPacketSent *sent_cb)
{
    ssize_t ret;

    if (queue->delivering) {
        return qemu_net_queue_append(queue, sender, flags, data, size, NULL);
    }

    ret = qemu_net_queue_deliver(queue, sender, flags, data, size);
    if (ret == 0) {
        qemu_net_queue_append(queue, sender, flags, data, size, sent_cb);
        return 0;
    }

    qemu_net_queue_flush(queue);

    return ret;
}
```

Figure 4: The `qemu_net_queue_send` function, used to send a packet across the user-mode VLAN.

```
static ssize_t qemu_net_queue_append(NetQueue *queue,
                                     VLANClientState *sender,
                                     unsigned flags,
                                     const uint8_t *buf,
                                     size_t size,
                                     NetPacketSent *sent_cb)
{
    NetPacket *packet;

    packet = qemu_malloc(sizeof(NetPacket) + size);
    packet->sender = sender;
    packet->flags = flags;
    packet->size = size;
    packet->sent_cb = sent_cb;
    memcpy(packet->data, buf, size);

    QTAILQ_INSERT_TAIL(&queue->packets, packet, entry);

    return size;
}

static ssize_t qemu_net_queue_deliver(NetQueue *queue,
                                      VLANClientState *sender,
                                      unsigned flags,
                                      const uint8_t *data,
                                      size_t size)
{
    ssize_t ret = -1;

    queue->delivering = 1;
    ret = queue->deliver(sender, flags, data, size, queue->opaque);
    queue->delivering = 0;

    return ret;
}
```

Figure 5: Relevant helpers for `qemu_net_queue_send`

In fact, it turns out that there is a very common service that matches that description exactly: ICMP ping responses. And, in fact, the `qemu-kvm` user-mode virtual gateway does respond to ICMP ping requests, reflecting back the payload of the packet (after the ICMP headers) unchanged.

Using this scheme, we can't fully control the initial bytes of the packet – there will be some ICMP and IP headers, as well as the `NetPacket` header, but fortunately, the fields we need to control in the `RTCState` structure are near the end of a 400+-byte allocation, and so this technique gives us sufficient control.

6 RIP control

In order to get `%rip` control, we will look at the `QEMUTimer` struct, as defined in `qemu-timer.c` in `qemu-kvm` 0.14.0. This definition is included as Figure 6 for reference.

```
typedef void QEMUTimerCB(void *opaque);

struct QEMUClock {
    int type;
    int enabled;
};

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

Figure 6: The `QEMUTimer` structure.

Recall that we have arranged the RTC state such that `rtc_update_second(s)` will be called once per second. Since `rtc_update_second` arranges for `s->second_timer` to be called on the next second, if we cause the newly-allocated `RTCState` to point to a fake `QEMUTimer` object, on the *next* second boundary, the fake timer's `->cb()` member will be invoked, causing the `qemu-kvm` process to jump to an address of our choice.

6.1 The `qemu-kvm` physical memory map

In order to inject dummy structures into the qemy-kvm address space, we'll make use of a convenient fact about how `qemu-kvm` manages the guest virtual machine's memory.

The guest's physical memory in `qemu-kvm` is backed by a single `mmap`'d region inside the `qemu-kvm` address space; `qemu-kvm` communicates this address to the kernel module using a KVM-specific `ioctl`, but also accesses it directly when emulation DMA operations or other IO accesses.

Thus, if we know the base address of this mapping, and we know the physical address of an allocation in the guest, we know the virtual address of the allocation inside `qemu-kvm`. Thus, we can make allocations in the guest, do a little arithmetic, and refer to them inside the host `qemu-kvm` process.

On Linux, we can find the physical frame number ("pfn") corresponding to any page of virtual memory in a process using `/proc/`*`pid`*`/pagemap`. In general, the address of the physical memory map will be subject to ASLR and potentially unpredictable; For the moment, however, we will assume a `qemu-kvm` process running without ASLR, and then demonstrate a way around this restriction in a later section.

6.2 Putting it together

So, to achieve `%rip` control in the `qemu-kvm` process, we need to allocated a `QEMUClock` structure in the guest, as well as a `QEMUTimer`, with `clock` pointing at the `HVA`of our dummy clock, `expire_time` being some small value, and `cb` pointing at the desired `%rip`. We then hotploug the ISA bridge, and spam ping packets at the VLAN gateway, with contents constructed such that, when appropriate headers are added, the result is the same size as a `RTCState` with `second_timer` pointing at the `HVA`of our dummy timer.

7 Chaining to shellcode

Note that the guest physical-address region is mapped non-executable inside the `qemu-kvm` process, so we cannot yet trivially inject our own shellcode.

With control over `%rip` and the ability to inject data into the guest, we could easily perform a standard ROP pivot, use that to `mmap` or `mprotect` an executable region, and inject shellcode in that way.

However, we opted to pursue a slightly different strategy, which allows for slightly easier portability between different versions of `qemu-kvm`, and easier cleanup and continuation of execution after the exploit completes.

7.1 Chaining timers

Looking at `QEMUTimer`, we notice the `next` member. Timers are stored on a sorted singly linked list; Every time the `qemu-kvm` main loop wakes up, each list is walked as long as `t->expire_time <` *`current time`*, and the timers encountered are invoked by calling `t->cb(t->opaque)`, and then unlinked.

In particular, this means that by controlling the `next` field of our injected timer, we can cause `qemu-kvm` to execute multiple sequential function calls, with a fully-controlled first argument.

And so, instead of a traditional ROP attack, we can chain fragments of code by constructing and chaining multiple timer objects. We do need to be careful about register usage, because the timer run loop will clobber several registers between sequential calls. However, of note, on every version of `qemu-kvm` examined, this dispatch loop does not make use of `%rsi`, which, on the amd64 SysV ABI, is used for passing integer argument 2. This allows us to call one function to set up `%rsi`, and then chain to a second, which can now control the first *two* arguments to `t->cb`.

7.2 Getting to `mprotect`

In this way, we can chain together calls to set up state and then call `mprotect` on a page inside the physical address region to mark that page as executable, and then jump into it to execute arbitrary shellcode. That code can then `mprotect` more pages, or otherwise bootstrap arbitrarily. Furthermore, once that code returns, the timer loop will continue executing as normal, making continued execution a comparatively simple manner of patching up any other problems resulting from the missing ISA bridge.

To call `mprotect`, which is a three-argument function, we make use of the `ioport_readl_thunk` `qemu-kvm` function, shown in Figure 7. By constructing a dummy `IORange` structure with a dummy `IORangeOps` method in `ops`, we can call `mprotect(ioport, LEN, 4);`. The constant `4` is conveniently equal to `PROT_EXEC`, so this suffices to mark the page containing the `IORange` as executable. We can then jump into there.

```
static uint32_t ioport_readl_thunk(void *opaque, uint32_t addr)
{
    IORange *ioport = opaque;
    uint64_t data;

    ioport->ops->read(ioport, addr - ioport->base, 4, &data);
    return data;
}
```

Figure 7: The `ioportl_readl_thunk` function, used by our exploit to indirectly call `mprotect`.

This work does not aim to comment on the development of possible payloads once code execution is achieved. However, it is worth noting briefly that the guest physical memory potentially provides an extremely convenient pathway for code in the guest and code in the now-compromised host to communicate, in order to e.g. implement a command interpreter, as demoed in the “Cloudburst” exploit for VMware [2].

8 Bypassing ASLR

So far, we have assumed that the target `qemu-kvm` process is running without ASLR, and addresses of both code and data in the target program are predictable. We now turn to the problem of attacking a target that has been hardened with ASLR, making some or all addresses unpredictable.

8.1 PIE and non-PIE executables

On Linux, enabling ASLR normally only randomizes the location of shared libraries (which must, in general, already contain position-independent code), and both the heap (The values returned by `brk` and `mmap`) and the stack. However, since executable programs are not normally compiled to be position-independent, the main executable must be loaded at exactly the address requested in the ELF binary, for both executable and data segments.

It is possible to explicitly compile an executable ELF binary as a so-called “position-independent executable”, or “PIE”, in which case their contents can be loaded at a random address. However, doing so requires using an additional register to keep track of addresses, and can have significant performance impact on some platforms. As such, compiling binaries as PIE is not the norm, and most distributions tend to compile, at most, a small set of “security-critical” programs as PIE (See, e.g. [3]).

As of this writing, no major distribution investigated builds `qemu-kvm` as PIE, which means that we can continue to rely on code addresses within the main `qemu-kvm` binary, as well as any data loaded directly from the `qemu-kvm` binary. Thus, in order to continue to execute the previous attack, we need only discover the address of the physical memory map inside the `qemu-kvm` process' address space.

8.2 `qemu-kvm` firmware emulation

`qemu-kvm` implements an interface for the built-in PC BIOS to read and write configuration firmware for certain emulated devices. Using IO ports `0x510` and `0x511`, the BIOS can read tables such as the emulated RAM size, the e820 map, ACPI tables, and other low-level system information. In addition, the `qemu-kvm` code includes support to allow certain firmware items to be writable from the guest. However, as of this writing, this feature appears to be unused in KVM head.

However, a missing check in the `fw_cfg_check` function in `hw/fw_cfg.c` allows the guest to write into *any* firmware configuration item, even ones that are not intended to be writable. And while writing the complete contents of a firmware configuration method will result in a segfault – due to calling a `NULL` callback – the guest can write any prefix of an item, and then toggle the firmware configuration address register to reset the write state, and repeat this operation as many times as desired.

Furthermore, at least of the firmware configuration items in a standard `qemu-kvm` "PC" hardware model are backed directly by statically-allocated structures in the `qemu-kvm` ELF binary.

Thus, by overwriting these items using the provided IO port interface, a guest can inject small amounts of data at a known address. We can this write our dummy `QEMUTimer` objects and other objects into this space.

8.3 Chaining timers even more

However, we cannot use these areas for the complete original attack. `mprotect` requires a page-aligned address in order to work, and the `IORange` trick we used required a writable object at the address we want to `mprotect`. Both writable firmware areas, however, are too small for it to be likely that either will straddle a page boundary in the appropriate way.

Instead, our strategy will be to use the firmware regions to construct a `read4` primitive, use this primitive to extract the base address of the physical memory map, and then procede with the original attack. Constructing a read primitive is easy – we just find a code fragment or function that reads a value at a known offset from `%rdi` and writes it to `%rsi`, and one that writes `%rsi` to the memory at address `%rdi`.

However, we now need to chain multiple series of timers, interspersed with logic in the guest program to compute appropriate offsets. The trick we use here is that, instead of terminating our fake timer chain in our shellcode, we terminate it with a timer that will call `rtc_update_second`, along with an appropriate fake `RTCState` object.

Thus, after our timer chain has been executed, we can return the host to executing a dummy timer we control once a second. We can then read the firmware area back out, process the bytes we read, and create a new timer chain to inject to execute the next round of code.

9 Conclusion

We have presented the design and construction of Virtunoid, a functional guest-to-host privilege escalation exploit for KVM. We have discussed some features of the design and implementation of KVM that are useful in constructing such an exploit, and shown how we chain them together to go from a use-after-free bug to reliable code execution, even in the presence of ASLR and non-executable data pages.

References

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

[2] Kostya Kortchinsky, Black Hat USA 2009, "Cloudburst: A VMware Guest to Host Escape Story "

[3] `https://wiki.ubuntu.com/Security/Features#pie`