Machine Virtualization for Fun, Profit, and Security

Muli Ben-Yehuda

Technion & IBM Research
Running multiple different unmodified operating systems
Each in an isolated virtual machine
Simultaneously
On the x86 architecture
Many uses: live migration, record & replay, testing, . . . , security
Foundation of IaaS cloud computing
Used nearly everywhere
What is the problem?
Popek and Goldberg’s virtualization model [Popek74]: Trap and emulate
Privileged instructions trap to the hypervisor
Hypervisor emulates their behavior
Without hardware support
With hardware support
What is a rootkit?

- First you take control. How?
- Then you hide to avoid detection and maintain control. How?
- Usual methods are ugly and **intrusive**: easy to detect!
- Can we do better?
Hypervisor-level rootkits

- Hypervisors have full control over the hardware
- Hypervisors can trap any operating system event
- Code can enter hypervisor-mode at any time
- Solution: run the rootkit as a hypervisor
Blue Pill Idea (simplified)

CALL bluepill

Native Operating System

PROC bluepill

enable SVM

prepare VMCB

VMRUN

check VMCB.exitcode

Blue Pill Hypervisor

VMCB

RIP

RET

Native Operating System continues to execute, but inside Virtual Machine this time...

RET from bluepill PROC, never reached in host mode, only executed once in guest mode

only during first call

© COSEINC Research, Advanced Malware Labs, 2006
Bluepill cont’

- Bluepill installs itself on the fly
- Can you bluepill bluepill?
What is the Turtles project?

- Efficient nested virtualization for Intel x86 based on KVM
- Runs multiple guest hypervisors and VMs: KVM, VMware, Linux, Windows, ...
- Code publicly available
What is the Turtles project? (cont’)

- Nested VMX virtualization for nested CPU virtualization
- Multi-dimensional paging for nested MMU virtualization
- Multi-level device assignment for nested I/O virtualization
- Micro-optimizations to make it go fast
Theory of nested CPU virtualization

- Trap and emulate [PopekGoldberg74] ⇒ it’s all about the traps
- Single-level (x86) vs. multi-level (e.g., z/VM)
- Single level ⇒ one hypervisor, many guests
- Turtles approach: $L_0$ multiplexes the hardware between $L_1$ and $L_2$, running both as guests of $L_0$—without either being aware of it
- (Scheme generalized for $n$ levels; Our focus is $n=2$)

---

Diagram:

- Host Hypervisor
- Hardware
- Multiple logical levels

- Guest
- Hypervisor
- $L_0$

- Guest
- Hypervisor
- $L_1$

- Guest
- $L_2$

- Guest
- $L_2$

- Guest
- $L_2$

- Guest
- $L_2$

- Guest
- $L_2$

- Guest
- $L_2$

- Guest
- $L_2$
Detecting hypervisor-based rootkits

- Bluepill authors claim “undetectable”
- “Compatibility is Not Transparency: VMM Detection Myths and Realities” [Garfinkel07]
- Hardware discrepancies
- Resource-sharing attacks
- Timing attacks: PCI register access, page-faults on MMIO access, cpuid timing vs. nops
- Can you trust time?
What does it mean, to do I/O?

- Programmed I/O (in/out instructions)
- Memory-mapped I/O (loads and stores)
- Direct memory access (DMA)
- Interrupts
Emulation is usually the default [Sugerman01]
Works for unmodified guests out of the box
Very low performance, due to many exits on the I/O path
I/O virtualization via paravirtualized devices

- Hypervisor aware drivers and “devices” [Barham03, Russell08]
- Requires new guest drivers
- Requires hypervisor involvement on the I/O path
Hypervisor-based I/O introspection

- Useful: anti-virus, intrusion detection, compression, live migration, ...
- Q1: how do you do it without impacting performance?
- Q2: how do you bridge the semantic gap?
Bypass the hypervisor on I/O path [Levasseur04,Ben-Yehuda06]
SR-IOV devices provide sharing in hardware
Best performance: 100% of bare-metal! [Gordon12]
### Comparing I/O virtualization methods

<table>
<thead>
<tr>
<th>IOV method</th>
<th>throughput (Mb/s)</th>
<th>CPU utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>bare-metal</td>
<td>950</td>
<td>20%</td>
</tr>
<tr>
<td>device assignment</td>
<td>950</td>
<td>25%</td>
</tr>
<tr>
<td>paravirtual</td>
<td>950</td>
<td>50%</td>
</tr>
<tr>
<td>emulation</td>
<td>250</td>
<td>100%</td>
</tr>
</tbody>
</table>

- **netperf** TCP_STREAM sender on 1Gb/s Ethernet (16K msgs)
- Device assignment best performing option
- Challenges: DMA and interrupts

---

Table from “The Turtles Project: Design and Implementation of Nested Virtualization” [Ben-Yehuda10]
Direct memory access (DMA)

- All modern devices access memory directly
- On bare-metal:
  - A trusted driver gives its device an address
  - Device reads or writes that address

**Protection problem**: guest drivers are not trusted
**Translation problem**: guest memory $\neq$ host memory
**Direct access**: the guest bypasses the host

What is the obvious attack?
How do you protect against it?
IOMMU

Main memory

Physical addresses

IOMMU

Device addresses

Device

MMU

Virtual addresses

CPU
Background: interrupts

- I/O devices raise interrupts
- CPU temporarily stops the currently executing code
- CPU jumps to a pre-specified interrupt handler
Interrupt-based attacks

- Follow the White Rabbit [Rutkowska11]
- Tell the device to generate “interesting” interrupts
- Attack: fool the CPU into SIPI
- Attack: syscall/hypercall injection
- Interrupt-based attacks: guest generating interrupts which are handled in host mode
- Why not handle interrupts in guest mode?
ELI: direct interrupts for unmodified, untrusted guests

“ELI: Bare-Metal Performance for I/O Virtualization”, Gordon, Amit, Hare’El, Ben-Yehuda, Landau, Schuster, Tsafrir, ASPLOS ’12
All interrupts are delivered directly to the guest
Host and other guests’ interrupts are bounced back to the host
... without the guest being aware of it
ELI: signaling completion

- Guests signal interrupt completions by writing to the Local Advance Programmable Interrupt Controller (LAPIC) End-of-Interrupt (EOI) register
- Old LAPIC: hypervisor traps load/stores to LAPIC page
- x2APIC: hypervisor can trap specific registers

Signaling completion without trapping requires x2APIC
- ELI gives the guest direct access only to the EOI register
Threats: malicious guests might try to:

- keep interrupts disabled
- signal invalid completions
- consume other guests or host interrupts
ELI: protection

- VMX preemption timer to force exits instead of timer interrupts
- Ignore spurious EOIs
- Protect critical interrupts by:
  - Delivering them to a non-ELI core if available
  - Redirecting them as NMIs → unconditional exit
  - Use IDTR limit to force #GP exits on critical interrupts
Conclusions

- Machine virtualization is very useful
- Can be used for good, or evil
- Complexity leads to unintended consequences
- Happy hacking!