Don’t Tell Joanna, The Virtualized Rootkit Is Dead
Agenda

★ Who we are and what we do
★ Virtualization 101
★ Vitriol/Hyperjacking (and other HVM Rootkits)
★ Why detecting HVMs aren’t as difficult as you think
★ Pro Forma Punditry
★ Q & A
about:nate.lawson

★ Co-designer of the Blu-ray disc content protection lay (at Cryptography Research)

★ FreeBSD Committer since 2002
  ‣ Author/maintainer of power management and ACPI kernel code

★ Designer of ISS RealSecure NIDS

★ Now: independent security consultant (Root Labs)
  ‣ Embedded and PC platform security, crypto design (e.g.: Chumby microcontroller-based authentication)
about:matasano

★ An Indie Security Firm: Founded Q1’05, Chicago and NYC.

★ Research:
  ‣ hardware virtualized rootkits
  ‣ endpoint agent vulnerabilities
  ‣ windows vista (on contract to msft)
  ‣ storage area networks (broke netapp)
  ‣ a protocol debugger
  ‣ 40+ pending advisories
rootkit highlights

1984
- thompson compiler
- backdoor

1994 - 1996
- hidesrc
- amodload
- libkvm

1998-
- IAT Rootkit
- Back Orifice
- SSDT Rootkit

2006-
- firmware
- virtualized
lightning intro to VT
software

- ring 3
  - guest A
  - guest B

- ring 0
  - vmm
  - host os

hardware

- ring 0
  - guest A
  - guest B

- ring -1
  - vmm
  - hardware

hardware shielded from guest os by de-privilegging or binary translating privileged instructions

hardware shielded from guest os by trap-and-emulate extension
shadowed state  

hypervisor  

VMCS  

host state  

guest state  

controls  

ring -1  

(root)  

ring 0  

(nonroot)  

ring 3  

(user)  

HW state (ivt, pages)  

OS state  

OS  

database  

web server
<table>
<thead>
<tr>
<th>insn</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmxon</td>
<td>enable VT</td>
</tr>
<tr>
<td>vmxoff</td>
<td>disable VT</td>
</tr>
<tr>
<td>vmclear</td>
<td>initialize VMCS</td>
</tr>
<tr>
<td>vmptrld</td>
<td>load current VMCS</td>
</tr>
<tr>
<td>vmptrrst</td>
<td>store current VMCS</td>
</tr>
<tr>
<td>vmread</td>
<td>read values from VMCS</td>
</tr>
<tr>
<td>vmwrite</td>
<td>write values to VMCS</td>
</tr>
<tr>
<td>vmlaunch</td>
<td>start and enter virtual machine</td>
</tr>
<tr>
<td>vmresume</td>
<td>re-enter virtual machine</td>
</tr>
<tr>
<td>vmcall</td>
<td>exit virtual machine</td>
</tr>
</tbody>
</table>
sequence of events

★ (1) guest OS accesses an msr
★ (2) vt traps, looks up host eip
★ (3) host calls trap handler
★ (4) trap handler emulates msr access
★ (5) trap handler incr guest IP
★ (6) trap handler issues `vmresume`
★ (7) guest OS continues
why this is interesting

★ VT is swapping entire OS-visible state in/out of memory (with API for access)
★ Guests have direct device access (unless you prevent them)
★ No software bit says “we’re virtualized”.
how we use VT
**hyperjacking**

**intended use case**

- **guest A**
- **guest B**
- **vmm**
- **hardware**

"heavy" vmm runs full-fledged guest machines on servers

**rootkit use case**

- **native OS**
- **vmm**
- **hardware**

"thin" vmm proxies access to hardware, keeps original OS running
minimal implementation; “client” and “server” do most of the work.
hyperjacking advantages

★ “Impossible to detect” (trap, emulate, and evade detection attempts; MITM the CPU)
★ Actually easier than kernel object manipulation
★ Potentially OS-independent (portable)
★ Potential shellcode payload (fully weaponized)
vitriol: hyperjacking
darwin/FreeBSD

★ Installed on the fly ("fork" the CPU)
★ Hypervisor and guest share CPU state: hypervisor can call into the OS
★ (Almost) no shadowed state (just one VM)
★ Pass (don’t trap) most events.
★ Proxy (don’t emulate/monitor) most traps.
vitriol: how it works

★ (1) get to cpl0
★ (2) check cpuid, feature msr for VMX
★ (3) allocate vmx and vmcs from IOMalloc
★ (4) initialize vmcs, call vmclear
★ (5) copy segments, stack, cr3 to vmcs host and guest
★ (6) set host(/root/hypervisor) eip to trap handler
★ (7) set exec controls to pick events we want
★ (8) vmptrld to add vmcs
★ (9) (a) vmlaunch (b) vmcall (c) vmresume
Vitriol is less than 1000 lines of code.
compare to bluepill

★ Same concept (hyperjacking proxy vmm)
★ Joanna uses AMD SVM
★ We don’t support nested VMs
★ We don’t hook the network (localhost only)
★ We don’t load stealthily (darwin kext)
★ Vitriol is a toolkit for detection experiments
HVMs in 2007

★ Full Nesting Support
  ‣ Allow other hypervisors to operate

★ Timing Detection and Submarining
  ‣ Cat and Mouse Detect / Evade
  ‣ Detect Detection and Remove Itself

★ Direct Driver Access
  ‣ No need to hook the OS

★ Weaponized Hypervisor
  ‣ HVM as kernel BO payload “shellcode”
what do we think?
are hvm rootkits a win?

★ SIMPLE
★ PORTABLE
★ UNDETECTABLE
simple?

★ VT is 10 instructions.
★ No OS deps in our code
  ‣ except loader and payload
★ ~700 lines of boilerplate (expect all hvm rootkits to share)
portable?

- We haven’t yet ported to Win32.
- It doesn’t look hard.
  - Need to rewrite loader and payload
**undetectable?**

<table>
<thead>
<tr>
<th><strong>kernel: fingerprints</strong></th>
<th><strong>vt: smoking gun</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ssdt/syscall table</td>
<td>hyperjacked vm root</td>
</tr>
<tr>
<td>function pointers</td>
<td></td>
</tr>
<tr>
<td>ivt</td>
<td></td>
</tr>
<tr>
<td>hidden pages</td>
<td></td>
</tr>
<tr>
<td>function detours</td>
<td></td>
</tr>
<tr>
<td>hidden threads</td>
<td></td>
</tr>
<tr>
<td>hidden processes</td>
<td></td>
</tr>
<tr>
<td>etc etc etc</td>
<td></td>
</tr>
</tbody>
</table>
VT-x may be hard to detect.
VT-x plus a software VMM isn’t.
detection heuristics

- FUNCTIONAL: behavior or state changes introduced by hypervisor.
- SIDE-CHANNEL: timing variations introduced by hypervisor.
detection goal

int is_virtualized(void);
backup goal

challenge

response

timing window

untrusted machine

trusted peer, trusted clock
analog: sniffer detection

★ GOAL: Find hacked servers with promiscuous sniffers.
★ TARGET: Promisc mode turns off MAC filtering.
★ FUNCTIONAL: Target responds to ping with wrong MAC.
★ SIDE-CHANNEL: Flood network with nonexistent MAC, measure ping.
measurement strategies

- DIRECT: time an instruction that causes a vm exit.
- INDIRECT: time state (cache, btb) before and after instruction that causes vm exit.
direct measurement

★ (1) rdtsc
★ (2) cpuid 1,000,000 times
★ (3) rdtsc
★ if clean: ~200 cycles
★ if hyperjacked: ~40,000 cycles
the problem with direct measurement

★ Hypervisor controls the TSC!
★ (1) on exit: save tsc
★ before re-entrance:
  › (2) take delta + exit overhead
  › (3) subtract from TSC offset
★ ~5 lines of code. This is a basic feature of VT-x and SVM.
one workaround

★ Use counters they didn’t think of:
  ‣ HPET counters
  ‣ Performance counters
  ‣ ACPI timers
  ‣ MSRs that betray timing and latency
★ They all need to agree for attackers to win
★ But attackers do control all of them
crypto timing attacks

★ aciicmez, tromer, bernstein, seifert
★ indirect microarchitecture measurement recovers secret crypto keys
RSA populates cache, evicts entries
spy saturates cache, detects evictions
shared feature
L2 cache, branch prediction caches
indirect measurement

★ (1) saturate a cache
★ (2) baseline cache hits with rdtsc
★ (3) cpuid
★ (4) repeat baseline
★ if clean: (2) and (4) agree
★ if hyperjacked: stuff evicted from cache
advantages we have over cryptanalysts

★ same cpu, same thread
★ not data-independent or oblivious
★ extensive shared state
★ don’t need to know chinese remainder theorem
conclusions

- How to make life hard for attackers:
  - Introduce data-dependence (many heuristics, not just one)
  - Force them to emulate the microarchitecture (indirect timing of cache, branch buffers)
  - Force them to emulate obscure features (HPET, PerfCounters, AGP GART)
  - Tie them to a single architecture (Intel VT, not Broadcom, Op Roms, etc)