Advanced Systems Security: Virtual Machine Systems

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Where are we?

- OS Security from Reference Monitor perspective
  - Mediation
    - LSM
  - Tamperproof
    - Linux and SELinux
  - Simple enough to verify
    - Correct code
    - Correct policy
Basis for OS Security

• Isolation
  ‣ A protection domain defines a boundary of isolation

• Based on
  ‣ Rings
  ‣ Address spaces
  ‣ Access control policy

• Do these work for modern OSes?
Virtual Machine Systems

• Protection domain is extended to operating systems on one physical platform
  ‣ Invented for resource utilization

• But, also provide a potential security benefit due to default
  ‣ ISOLATION

• How does VM isolation differ from OS isolation?
Virtual Machines

- Instead of using system software to enable sharing, use system software to enable isolation

- Virtualization
  - “a technique for hiding the physical characteristics of computing resources from the way in which others systems, applications, and end users interact with those resources”

- Virtual Machines
  - Single physical resource can appear as multiple logical resources
Virtual Machine Architectures

- **Full system simulation**
  - CPU can be simulated

- **Paravirtualization (Xen)**
  - VM has a special API
  - Requires OS changes

- **Native virtualization (VMWare)**
  - Simulate enough HW to run OS
  - OS is for same CPU

- **Application virtualization (JVM)**
  - Application API
Virtual Machine Types

- **Type I**
  - Lowest layer of software is VMM
  - E.g., Xen, VAX VMM, etc.

- **Type II**
  - Runs on a host operating system
  - E.g., VMWare, JVM, etc.

- Q: What are the trust model issues with Type II compared to Type I?
CPU Virtualization Trends

- The key trend is to eliminate the overhead of virtualization
  - Reduce overall world-switch times
  - Reduce world-switch frequencies
  - Reduce resources needed by the VMM

- Reduce world-switch times
  - Tag TLB by ASID
  - Better caching of VMCB state

- Reduce world-switch frequencies
  - Nested paging
  - Direct device assignment
  - Implement more functions in the guest OS through paravirtualization
Nested Page Tables

- Traditionally the hypervisor maintains shadow page tables:
  - Expensive to emulate correct behavior (accessed/modified bits)
- Nested paging eliminates this by tagged TLB ($GV \rightarrow GP$ & $GP \rightarrow SP$)
  - Available in Barcelona ...
  - Reduces number of VMEXITs by 40-70%
Hardware Virtualization

- CPU virtualization
  - Instructions (still there)
  - Sensitive instructions must be privileged

- Memory virtualization
  - MMU (still there)
  - Nested page tables

- I/O virtualization
  - IOMMU (new)
  - Chipset support for configuration and address translation
VM Systems and Ref Monitor

• How does a VM System improve ability to achieve reference monitor guarantees?
VM Systems and Ref Monitor

- How does a VM System improve ability to achieve reference monitor guarantees?
- Mediation
  - Mediation between VM interactions
- Tamperproof
  - Protection boundaries between VMs (OS)
- Simple Enough to Verify
  - Code that needs to be correct?
  - Policy
VAX VMM

- A1-assured VMM system
- Carefully crafted VMM
- Mediation
  - VM interaction
- Tamperproof
  - Minimal TCB
- Simple enough to verify
  - Code assurance
  - Policy assurance: MLS policy, Biba policy, privileges
VAX VMM Architecture

- Applications (Top Secret)
  - Ultrix OS

- Applications (Secret)
  - VMS OS

- Applications (Unclassified)
  - VMS OS

VMM Security Kernel

- Memory Device
- Disk Device
- Print Device
- Display Device
...
• Key design tasks
  ‣ Virtualize processor
    • Make all sensitive instructions privileged
  ‣ More rings
    • Need a new ring for the VMM
  ‣ I/O emulation
  ‣ Self-virtualizable

• What components constitute the VAX VMM reference monitor?
VAX VMM Policy

- MLS
  - Control secrecy
- Biba
  - Control integrity
- Privileges
  - Exceptional accesses
  - Audited
  - There are more of these than meets the eye!
- How is the protection state modified?
VAX VMM Evaluation

- **Mediation**: ensure all security-sensitive operations are mediated?
  - Virtualizing instructions, I/O emulation
  - VM-level operations? Privileges

- **Mediation**: mediate all resources?
  - VMM level

- **Mediation**: verify complete mediation?
  - All-assured at VMM level
VAX VMM Evaluation

- **Tamperproof**: protect VMM?
  - Similar to Multics (no gatekeepers, but some kind of filters); authentication in VMM; protection system ops in VMM

- **Tamperproof**: protect TCB?
  - All trusted code at ring 0; trusted path from VMs for admin;

- **Verification**: verify code?
  - A1-assured at VMM level

- **Verification**: verify policy?
  - MLS and Biba express goals and policy; Privileges are ad hoc
VAX VMM Challenges

• Despite A1 assurance still several challenges in VAX VMM system
  ‣ Device driver management; no network
  ‣ Amount of assembler code
  ‣ Covert channel countermeasures
  ‣ Implications of ‘privileges’

• Nonetheless, interesting mechanisms
  ‣ Trusted path administration
  ‣ Architecture of VMM
  ‣ Virtualization for security
Modern VM Systems

- The development of a virtual machine monitor for x86 systems unleashed VMs on the masses
  - Why did this take so long?
- VMware, Xen, KVM, NetTop, …
  - Everyone is a virtual machine monitor now
- How do we implement a reference validation mechanism for these systems?
  - What granularity of control?
Isolation and Network

- Type I VM Systems assume that the VMM (and privileged VM) will **isolate guest VMs**
- Then, the problem is to control inter-VM communication
  - Other communication is **via the network**
- sHype adds reference monitor for controlling network access between VMs
- **NetTop** is built on VMware where only VMs of the same label may communicate
Control of VMM Resources

• There are many virtual machine monitor resources that may be used to communicate
  ‣ Memory, devices, IPC, VMs themselves, …
  ‣ VMware permits VMCI – like IPC between VMs

• **Xen Security Modules (XSM)** adds reference validation on the Xen hypervisor’s distribution of these resources
  ‣ Less trust in privileged VMs, so finer-grained policy results

• Minimizing TCB versus simplicity
Xen as a Reference Monitor?

- Reference Monitor
  - XSM in Xen
  - Scope includes dom0 Linux and user-level

- Mediation
  - XSM to control VMM operations
  - SELinux in dom0; use network to communicate

- Tamperproof
  - Xen has a much larger TCB, and more flexible

- Verification
  - Code – 200K+ LOC
  - Policy – SELinux style
VMs as Processes

- Type II VM systems can treat VMs as processes
- **KVM** uses SELinux to control access of VMs as if they are a process
  - VMs are processes to the host OS
  - VMs can access host OS resources (files)
- Uses SELinux to control VM access
Take Away

• VM Systems provide isolation
  ▶ Between OSes/apps that may be untrusted

• VM Systems enable a small TCB
  ▶ Type 1 VMMs
  ▶ A1-Assured, like VAX VMM

• VM Systems can mediate inter-VM actions
  ▶ Virtualized operations
  ▶ Inter-VM operations
Conventional OS vs VM System

- Conventional OS
  - Broken easily and often

- VM system
  - Coarser control based on isolation

- If we trust the VM system and don’t trust the OS, what can we do?
Options

- Isolate in VM systems
  - Can deploy an application on a custom OS
  - What should be the basis for the custom OS?

- Microkernels
  - Reduce code running in kernel mode
  - But, need the some services
  - These are just as “trusted” running in user-space

- SELinux/AppArmor/Trusted Solaris
  - Still a large kernel
Deploy Critical Applications

- Don’t trust OS, but need its services
- Run programs directly on VMM
  - No services
- Run programs on a specialized, trusted system
  - Custom services must be written (yuk!)
- How do we use untrusted OS services safely in virtualized environment?
A solution should...

- Ease Adoption
  - It is usable...

- Support Diverse Applications
  - …to a lot of people...

- Have an Incremental Path to Higher Assurance
  - …also.
Splitting Interfaces

Solution

• Separate application from other apps/kernel
  – Use separate VM for app with a Private OS separate from Commodity OS

• Provide interaction between apps/kernel in a secure way
  – Application developer decides what is sensitive and what is not
    • Separate sensitive part into VM on Private OS
    • Public part remains on Commodity OS
    • Interaction between apps also passes through kernel (eg. pipe(), mkfifo())
      • Sensitive part communicates through system calls with other apps
  – Use policy to decide if system calls are to be performed on commodity OS or private OS
Proxos Architecture

- Private VM:
  - Private OS Methods
  - Private Application
  - Proxos:
    - Trusted system calls are routed to private OS methods
    - Untrusted system calls are routed to the commodity OS kernel

- Commodity OS VM:
  - Other Applications
  - Host Process
  - Commodity OS Kernel:
    - Private apps can interact with other apps via the host process
    - All apps can access commodity OS resources

VMM
Proxos Guarantees

• Assumption
  – VMM enforces separation
  – Application developer correctly specifies routing rules

• Guarantee
  – Confidentiality and integrity of sensitive private
    application data inspite of malicious commodity OS
    • VMM => No direct interference possible
    • Commodity OS can interfere with system calls routed to it,
      which are not security-sensitive
  – Availability not guaranteed
Proxos Routing Language

• Needs to specify which system calls go where (arguments need be considered)
• Solution: Partition system calls by resources they access
  – Disk, Network, UI, Randomness, System Time, Memory
  – Randomness, System Time -> Always routed to VMM
  – Memory -> Always routed to private OS

```
# Rules Section
# route accesses to /etc/secrets to private OS
DISK:("/etc/secrets", priv_fs)
# route accesses to UNIX domain socket bound to /tmp/socket and TCP socket bound to peer
# 192.100.0.4 port 1337 to private OS
NETWORK:("unix:/tmp/socket", priv_unix),
  ("tcp:192.100.0.4:1337", priv_tcp)
# route all accesses to stdin, stdout
# and stderr to private OS
UI: (*,priv_ui)

# Methods Section
# individual methods in the private OS
# that are bound to system calls
priv_fs = {
    .open = priv_open,
    .close = priv_close,
    .read = priv_read,
    .write = priv_write,
    .lseek = priv_lseek
}
```
Proxos Implementation

Host Process
- `pr_execve(app_name)`

Linux Kernel
- Allocate shared mem
- Pass (app_name, addr of shared mem) to VMM
- Start private VM

VMM/dom0
- VMM creates private VM, gives it shared mem addr
- Associate VMID with addr
- Return VMID

Private VM
- Maps syscall shared mem into addr space
- When private VM gives map request, checks if VMID corresponds to addr
Proxos Implementation

Diagram showing the Proxos implementation with Private VM, Commodity OS VM, Shared Buffer, Linux Kernel, Host Process, and Interrupt Handler.
Proxos SSH Server
Compare to Privilege Separation
## Implementation Effort

<table>
<thead>
<tr>
<th>Application</th>
<th>Rules</th>
<th>LOC Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dillo</td>
<td>53</td>
<td>22</td>
</tr>
<tr>
<td>SSH Server</td>
<td>35</td>
<td>108</td>
</tr>
<tr>
<td>Apache &amp; OpenSSL</td>
<td>28</td>
<td>667</td>
</tr>
<tr>
<td>Glibc</td>
<td></td>
<td>218</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total LOC</th>
<th>% Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>20,528</td>
<td>0.1%</td>
</tr>
<tr>
<td>27,000</td>
<td>0.4%</td>
</tr>
<tr>
<td>135,916</td>
<td>0.5%</td>
</tr>
<tr>
<td>1,775,440</td>
<td>0.01%</td>
</tr>
</tbody>
</table>
Performance

- System call forwarding overhead
  - Context Switch Cost: 14us.

![Diagram showing system call forwarding overhead]

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Linux</th>
<th>Proxos</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL system call</td>
<td>0.37</td>
<td>12.88</td>
<td>12.51</td>
</tr>
<tr>
<td>fstat</td>
<td>0.57</td>
<td>14.28</td>
<td>13.71</td>
</tr>
<tr>
<td>stat</td>
<td>8.76</td>
<td>25.98</td>
<td>17.22</td>
</tr>
<tr>
<td>open &amp; close</td>
<td>14.57</td>
<td>47.18</td>
<td>32.61</td>
</tr>
<tr>
<td>read</td>
<td>0.45</td>
<td>13.51</td>
<td>13.06</td>
</tr>
<tr>
<td>write</td>
<td>0.42</td>
<td>13.24</td>
<td>12.82</td>
</tr>
</tbody>
</table>
Remaining Problems

• Deploying a custom OS is painful
  ‣ Building a special kernel is non-trivial
• And it may not be secure itself
  ‣ Still need a methodology to determine code correctness and tamperproofing

• What if you want to eliminate trust in the OS altogether?
Insight: Shadowing Memory

- VMMs need to manage physical to virtual mapping of memory
- This is done with a shadow page table
- Multi-shadowing give context-aware views of this memory
  - Use encryption instead
Nested Page Tables

- Traditionally the hypervisor maintains shadow page tables:
  - Expensive to emulate correct behavior (accessed/modified bits)
- Nested paging eliminates this by tagged TLB (GV→GP & GP→SP)
  - Available in Barcelona …
  - Reduces number of VMEXITs by 40-70%
Memory Cloaking

- Not new idea
  - XOM, LT
- Encrypt the pages in memory
  - For each page, (IV, H) meta data
  - What should the “secure hash” be?
- OS can operate on encrypted pages
  - But can’t read them
Tasks of the Overshadow

- Mediate all application interaction with OS to ensure correct cloaking of memory
  - Context Identification
  - Secure Control Transfer
  - System Call Adaptation
  - Mapping Cloaked Resources
  - Managing Protection Metadata
Shim baby Shim

- The key to Overshadow is the Shim
  - Manages transitions to and from VMM via a hypercall
- Shim Memory protects application
  - CTC protects control registers
- Uncloaked Shim
  - Neutral ground
  - Trampoline!

![Overshadow Architecture](chart.png)
Loading Applications

- The Shim uses a **Loader** program
- Sets up the cloaked memory with a hypercall
- The loader / shim must be trusted
  - Metadata on the CTC checks for compromise
  - Here is the **meat** of the problem
    - Is it even used?
- Propagate shims to spawned applications
Its not that easy…

• Lot of OS interfaces that must be handled
• Faults / Interrupts
• System Calls
  ‣ Pass control to the VMM
  ‣ The shim catches this and stores registers
    • Clear the registers to prevent side channels
Complex Syscalls

• Some syscalls are easy
  ‣ No side effects
  ‣ Nice, getpid, sync

• Others, less so…
  ‣ Pipe, r/w (Zero data)
  ‣ Clone
  ‣ Fork
  ‣ Signal Handling
Performance

• Microbenchmarks
  ‣ Not so hot 15-60%
  ‣ Although a lot better than Proxos

• Application Benchmarks
  ‣ SPEC isn’t so bad
  ‣ High bandwidth hits some bottlenecks
  ‣ Why?
Take Away

- VM Systems provide isolation
  - At OS granularity: some can be untrusted
- OS provides services used by applications
  - Access to devices demultiplexed among VMs
- Can we use VM isolation to prevent compromise of applications by malicious OS?
  - Proxos: use a “trusted” OS and redirect service requests
  - Overshadow: use OS as untrusted communication media