Using Hypervisors to Secure Commodity Operating Systems

David Lie
Department of Electrical and Computer Engineering
University of Toronto
Operating System Insecurity

- Commodity OS contain a lot of privileged code:
  - Kernel and all privileged applications present a huge TCB that is difficult to secure
  - Any vulnerability in the TCB gives attacker privileges -- all is lost
Commodity Systems: Pro’s and Con’s

• **Pro’s:**
  – Large amount of software already written (lower cost)
  – People familiar with configuration and use (lower cost)
  – While there is more code, the code is more mature
    • Newer smaller designs may not be better

• **Con’s:**
  – Large code base means more vulnerabilities
  – Hard to reduce code size because people depend on existing features

Goal: Find a way to secure commodity systems secure without having to (re)write a lot of code
Root of trust

- Problems with commodity OS arise because root of trust is conflated with much other (complex) functionality.
- Root of trust:
  - The component that applications can depend on to guarantee their security
- Commodity OS fulfills this role, but also must:
  - Resource management and accounting
  - Hardware abstraction and multiplexing
  - Authentication and identity management
  - User interfaces

Can we find an alternative root of trust?
Some alternative roots of trust

• TPM:
  – Simple but provides **limited functionality**
  – Ability to attest to the content of memory and “seal” keys against that content

• Trusted processor:
  – Enable main processor to perform private and verified execution
  – XOM, Research project, 2001
  – Some similarity to modern IBM Cell and ARM TrustZone
  – Requires **significant architectural changes to processor**
What about a software root of trust?

Hypervisor can act like hardware, but requires no hardware changes
Two Approaches

1. **Protect** applications from a compromised OS:
   - Hypervisors enable the implementation of an alternate *root of trust*. Applications trust low-level hypervisor code to provide protection instead of the commodity OS
   - Compromised OS does not mean the security of the application is at risk

2. **Detect** compromises on OS:
   - Hypervisors are *root-secure*. Monitors in hypervisor cannot be disabled if OS and OS kernel become compromised.
Terra (SOSP 2003)

- TPM attests hypervisor, hypervisor attests individual VMs
- Allows attested VMs to run alongside unattested VMs
  - Allows attesting a subset of code on the machine
- Some issues addressed by late-launch (which was developed after Terra)
• Main benefits:
  – Hypervisors, due to their size and simplicity, are more secure than commodity OSs
  – Thus they are root-secure.
    • Assumption is that they are harder to compromise than the OS kernel
    • Even if VM is compromised, attacker only gets root (superuser) on the VM, not on the hypervisor or other VMs
  – Optimistic attestation:
    • Hypervisor has more capabilities than TPM. Rather than attesting entire VM at load time, *may enforce integrity checks for the duration of VM runtime.*
Proxos (OSDI 2006)

- Would like to protect application from attack by commodity OS:
  - At the same time, allow application to use commodity OS to communicate with other apps and outside world

Need a way for application to control how much it trusts the OS
Proxos Overview

Proxy Operating System (Proxos) routes system calls based on rules set by developer.
Example Application: SSH

- Proxos allows applications to have access to commodity OS, but isolated sensitive resources at the same time. Ex SSHD Server:
  - Sensitive data such as user passwords and the host key stored in private OS
  - All network packets decrypted in private app before sent to command shell
Overshadow (ASPLOS 2008)

- Software implementation of a secure processor
  - Hypervisor allows alternative “views” of memory
- Commodity OS only sees encrypted memory, applications see unencrypted memory
  - Result: Applications trust hypervisor to protect their memory and state from compromised OS kernels
Overshadow Operation

- Hypervisor uses page tables to detect whether a process memory page is being accessed by OS kernel or the user process:
  - User processes can see plain text version
  - Accesses by kernel are intercepted and the page is encrypted
- OS kernel only sees encrypted versions of the memory pages.
- System calls must be intercepted and emulated
  - System call arguments and return values copied and left in plain text.
CloudVisor (SOSP 2011)

- Similar structure and concept as Overshadow
  - A nested hypervisor protects VMs from a compromised hypervisor
  - Memory and state are encrypted/decrypted dynamically so that VMs can access plaintext memory but hypervisor only sees corresponding encrypted state.
Summary

• Hypervisors can provide a richer root of trust without needing specialized processor architectures:
  – Can use standard TPM to attest the integrity of the hypervisor

• What can benefit from a richer root of trust?
  – Terra: **Users** trust the hypervisor to perform partial attestation of code on a system
  – Proxos, Overshadow: **Applications** trust the hypervisor to protect them from a compromised OS kernel
  – Cloud Visor: **Customer VMs** trust *nested hypervisor* to protect them from a compromised hypervisor
Detecting Compromises

- Traditional IDS can be implemented in the network or on the host
  - Network is isolated from hosts so will continue to function if hosts are compromised.
    - However, they have limited visibility making them vulnerable to evasion.
    - Cannot monitor attacks over encrypted channels (https)
  - Host-based intrusion detection is more difficult to evading located at the end point
    - However, they are vulnerable to subversion if the host is compromised. Attacker can simply disable IDS.
IDS Trust the OS

OS-based security tools are vulnerable to subversion
VMM-based IDS (NDSS 2003)

- App. A
- App. B
- Intrusion Detection System
- Commodity OS
- Hardware
- Hypervisor
XenAccess (ACSAC 2007)

- Similar concept to VMM-based IDS but for XenAccess
  - Both use VM *Introspection* to monitor guest VM
- A key challenge is bridging the *semantic gap* between the hypervisor and the guest VM
  - Hypervisor has access to hardware due to underlying privileges. However, it needs information to properly *interpret* information gleamed from it.
  - Example: which process does a memory page belong to? How many processes are executing? What programs are executing?
- Both VMM-based IDS and XenAccess depend on symbol information to find kernel and process data structures and code
Isolation Does Not Stop the Attacker

Relies on **non-binding** symbol information to interpret state
- Attacker can alter the mapping of symbols and fool IDS
Isolation Does Not Stop the Attacker

Monitor is not synchronized with events in the OS

- Must sample at random intervals, which allows attacker to evade
Patagonix (UsenixSec 2008)

• Patagonix introduces *architectural introspection*
  – The semantic gap created by introspection is bridged using *only* architectural features of the system.
  – These features cannot be easily subverted by attacker

• Assume a “white-list” of known-good executables
  – Patagonix will match all executing code into this list
  – Unidentified code is flagged by Patagonix, *no code can execute without Patagonix’s knowledge*
  – Even a compromised OS kernel cannot circumvent Patagonix
Patagonix: Detecting Code Execution

- Use Page protections to detect execution of code

Permissions

<table>
<thead>
<tr>
<th>W=0</th>
<th>X=0</th>
</tr>
</thead>
</table>

Fault

VM suspended
Hypervisor invoked

Hypervisor notified

Control logic in Patagonix VM

Identify code
- Writes to the page result inform the hypervisor with a fault
- If page is executed, another fault is generated
Patagonix identifies the binary from which a memory page came.

Why Binaries: Easier and more secure
- Binary uniquely identifies instructions, which define behavior
- Trustworthy repositories of hashes of known binaries exist
  - Example is NIST: http://www.nsrl.nist.gov/

How: use binary file format specification
- They specify how the OS should load the code.
  - If adversary violates this code will not load
  - If adversary modifies OS, code will not match
- They change rarely - ELF for all Unix, PE for all Windows.
  - More stable than OS kernel structures!
Example: The PE oracle

On Disk

| 4adb e4e5 d1a7 85e5 |
| d1c1 2a52 421c 67b7 |
| df60 6125 fdb6 eb73 |
| 60d2 0626 d316 8917 |
| 0fb5 6377 c758 42a4 |
| 6f56 1c9b 60c2 ee6f |
| 7aef 35bf e7a3 8ce0 |
| 1078 c8bd 4a3a 840a |
| 60db 20e6 2c67 56c3 |
| ba7a 1358 a890 38fb |
| 3f99 0565 55a6 f71a |
| 60c0 687d c0ce 6508 |

In Memory

| 4adb e4e5 d1a7 85e5 |
| d1c1 2a52 421c 67b7 |
| df60 6125 fdb6 eb73 |
| 2c7c 1626 d316 8917 |
| 0fb5 6377 c758 42a4 |
| 6f56 1c9b 2c6c fe6f |
| 7aef 35bf e7a3 8ce0 |
| 1078 c8bd 4a3a 840a |
| 2c85 30e6 2c67 56c3 |
| ba7a 1358 a890 38fb |
| 3f99 0565 55a6 f71a |
| 60c0 687d c0ce 6508 |

Preferred address: 0x60c2 2000
Solution: use binary entry points
Actual address: 0x2c6c 3000
The PE oracle

In Memory

Fault at 0x2c6c 34e5

Offset is 0x4e5

Candidate w/ preferred addr. 0x60c2 2000, list of relocations

Memory address: 0x2c6c 3000

0x60c2 2000 - 0x2c6c 3000
0x3455 f000
Management console
Hypervisor Assurance

• Main assumption in literature: hypervisors are more secure than commodity OS kernels
  – The basis of this argument is that they are smaller and simpler
  – If this is not true, many of the claims are tenuous

• So far this assumption appears to hold
  – Far fewer CVE vulnerabilities for hypervisors than commodity OS
  – Code sizes remain smaller
# Hypervisor Assurance

<table>
<thead>
<tr>
<th>Component</th>
<th>Release Date</th>
<th>LOC (Sloccount)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xen-2.0.7</td>
<td>08/2005</td>
<td>175,942</td>
</tr>
<tr>
<td>Xen-3.0.2</td>
<td>04/2006</td>
<td>349,438</td>
</tr>
<tr>
<td>Xen-3.1.3</td>
<td>12/2007</td>
<td>567,117</td>
</tr>
<tr>
<td>Xen-3.2.0</td>
<td>01/2008</td>
<td>536,691</td>
</tr>
<tr>
<td>Xen-3.3.0</td>
<td>08/2008</td>
<td>910,066</td>
</tr>
<tr>
<td>Xen-3.4.0</td>
<td>05/2009</td>
<td>845,112</td>
</tr>
<tr>
<td>Xen-4.0</td>
<td>04/2010</td>
<td>919,129</td>
</tr>
<tr>
<td>Linux 2.6.34.14</td>
<td>05/2010</td>
<td>8,356,048</td>
</tr>
</tbody>
</table>

Still much smaller than commodity OS kernels: Will this continue to hold?
Hypervisor Assurance

• Still there are concerns:
  – Use of hypervisors in new services like IaaS clouds mean that hypervisor assurance is more important
  – Cloud computing demands new functionality from hypervisors such as metering, attestation, etc. Increasing complexity
  – Will hypervisor size/complexity stay constant or grow in the future?
Hypervisor Assurance

• Some solutions:
  – **NoHype** (ISCA 2010) envisions a virtualization system implemented entirely in hardware
    • One core per VM
    • No over commitment or resource sharing between VMs
  – **SecVisor** (ASPLOS 2007) a minimal hypervisor that provides continuous verification of kernel code integrity
  – **seL4** (SOSP 2009) Formal verification of a 10K-line micro-kernel. While a microkernel is not the same as a hypervisor, they share many similarities. This suggests that verification of a hypervisor may be possible.
Conclusion

• Hypervisors provide a cost effective way to add security to commodity systems:
  – Provide isolation from OS if compromised
  – Do not require any specialized hardware

• 2 dominant strategies:
  – Protecting applications from compromised OS
  – Monitoring and introspection of OS from hypervisor

• Hypervisors can help secure commodity code:
  – Performance impact is low
  – Hypervisors are stable and mature, yet continue to be significantly smaller and simpler than OS kernels