Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions

Xi Xiong, Donghai Tian and Peng Liu
The Pennsylvania State University
Outline

- Motivation
- Approach Overview
- Key Design & Implementation
- Evaluation
- Summary
Kernel compromise through extension interface

- Malware: kernel-level rootkits
  - e.g., subvert kernel meta data or control flow to hide malicious activities

- Buggy extensions
  - Linux drivers are seven times more likely to contain bugs than other kernel code. [Chou, SOSP 01]

- Malicious Device Drivers
Related Work

- Prohibit execution of untrusted code
  - Secvisor [Seshadri ‘07], NICKLE [Riley ‘08]...

- Kernel control data protection
  - HookSafe [Wang ‘09]...

- Monitor the behavior
  - K-Tracer [Lanzi ‘09], Poker [Riley ‘09]...

- Find signatures and invariants
  - Gibraltar [Baliga ‘08], Robust Signature [Dolan-Gavitt ‘09], KOP [Carbone ‘09], SigGraph [Lin ‘11]...

- Our approach: shepherd untrusted extensions
How to let untrusted kernel extensions safely run to provide desired functionalities without harming the integrity of the OS kernel?
Outline

- Motivation
- Approach Overview
- Key Design & Implementation
- Evaluation
- Summary
Kernel Integrity Threatened by extensions

- Kernel Code/Data Integrity
- Architectural state integrity
- Control flow integrity
  - e.g., extensions jump to undesired positions of kernel text
- Stack integrity
  - e.g., inject malicious kernel stack frames
Using run-time **access control** to limit (shepherd) what untrusted extensions can do.

**examples:**
- untrusted extensions cannot change the kernel code
- they cannot write to high integrity data objects owned by kernel, but kernel can
- they can only invoke a limited set of kernel APIs
- they can only write to its own stack frames
In commodity OS, extensions and OS kernel are in the same execution context (no context switch)

- subject identification: who is running? extension or kernel?

- Kernel and extension are in the same address space with less meta information
  - object identification: figure out which part of physical memory contains which type of objects.
Writing to kernel objects are directly through memory operations, no existing interface to place authorization hooks
- system calls, LSM
- mediation and enforcement challenge

How to monitor control flow transfer and guarantee its integrity?
HUKO: a hypervisor based protection system
- mediation on kernel-extension interaction
- run-time mandatory access control

Overview

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Design Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject Identification</td>
<td>Protection States</td>
</tr>
<tr>
<td>Object Identification</td>
<td>Page-based kernel object labeling</td>
</tr>
<tr>
<td>Mediation and Enforcement</td>
<td>VMM-level protection domains using Hardware assisted paging</td>
</tr>
<tr>
<td>Control Flow Integrity</td>
<td>Trusted Entry Points, call-return consistency</td>
</tr>
</tbody>
</table>
Outline

- Motivation
- Approach Overview
- Key Design & Implementation
- Evaluation
- Summary
Protection States

- Who is running?

1. System calls, interrupts and exceptions
2. Return from system calls, interrupts and exceptions
3. Kernel calls the extension, extension preemption, and kernel call returns
4. Extension calls the kernel, kernel preemption, and extension call returns
5. Inter-extension calls and returns between trusted and untrusted extensions
Object Labeling

- Type-based labeling
  - e.g., KERNEL_CODE, KERNEL_DATA, UNTRUSTED_CODE

- Labels are associated with corresponding physical pages

- Need assistance from OS for
  - extension loading
  - dynamic page allocation and reclaiming

- Issue: Mixed pages
  - Code and data, Trusted and untrusted content, superpages
## Access control policy

<table>
<thead>
<tr>
<th>Object Label</th>
<th>OS Kernel</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read</td>
<td>Write</td>
<td>Execute</td>
<td>Read</td>
<td>Write</td>
<td>Execute</td>
<td>Read</td>
</tr>
<tr>
<td>Trusted Entry Points</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>audit allow</td>
<td>allow</td>
</tr>
<tr>
<td>Other OS Code</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>audit allow</td>
<td>allow</td>
</tr>
<tr>
<td>OS Data</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>audit allow</td>
<td>allow</td>
</tr>
<tr>
<td>Trusted Extension</td>
<td>allow</td>
<td>allow</td>
<td>audit allow</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
</tr>
<tr>
<td>Untrusted Extension</td>
<td>allow</td>
<td>allow</td>
<td>audit allow</td>
<td>allow</td>
<td>allow</td>
<td>audit allow</td>
<td>allow</td>
</tr>
<tr>
<td>Private Stack Frames</td>
<td>allow</td>
<td>allow</td>
<td>deny</td>
<td>allow</td>
<td>allow</td>
<td>deny</td>
<td>allow</td>
</tr>
<tr>
<td>Other Stack Frames</td>
<td>allow</td>
<td>allow</td>
<td>deny</td>
<td>allow</td>
<td>allow</td>
<td>deny</td>
<td>allow</td>
</tr>
<tr>
<td>Trusted DMA</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>audit allow</td>
<td>allow</td>
</tr>
<tr>
<td>Shared DMA</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
<td>allow</td>
</tr>
<tr>
<td>User Space Content</td>
<td>allow</td>
<td>allow</td>
<td>audit allow</td>
<td>allow</td>
<td>allow</td>
<td>audit allow</td>
<td>allow</td>
</tr>
</tbody>
</table>
Basic idea: create hardware enforced protection domains
- address space separation
- protection state transition: implemented by domain switch

How to achieve?
- multiple sets of page tables for different protection domains, switch the page table upon protection state transition
- protection access rights are reflected in the page table access permissions
- protection state transitions can be caught by setting execution permissions
Example work flow

- Components
  - Protection states
  - Object labeling
  - Memory isolation
Protection State: OS Kernel
Protection State: OS Kernel

Execution OK!
Protection State: OS Kernel

OS Kernel Table

Untrusted Extension Table

HAP Base Pointer

Protection State Switch

R 1
W 1
X 1

R 1
W 0
X 0

R 1
W 0
X 0

R 1
W 1
X 1

Machine Physical Memory

Kernel Code

Kernel Data

Extension Code
Protection State: OS Kernel

Write OK!
Protection State: OS Kernel

OS Kernel Table

Untrusted Extension Table

Machine Physical Memory

Kernel Code

Kernel Data

Extension Code
Protection State: OS Kernel

OS Kernel Table

Untrusted Extension Table

Execution Exception!
Protection State: Untrusted Extension
Protection State: Untrusted Extension
Protection State: Untrusted Extension
Protection State: Untrusted Extension

Write Denied!
Protection State: Untrusted Extension
Protection State: Untrusted Extension
Protection State: OS Kernel
Prototype built on Intel’s Extended Page Table (EPT) and Xen hypervisor 3.4.2

Utilize unused bits in EPT entry for page label

a trusted Linux kernel module to gather information from dynamic allocators and module loader
  - facilitate object labeling
In our opinion, HAP is a cleaner design solution

- Independent layer, do not need to be consistent with guest page tables
- Less update, easier to synchronize multiple copies
- Less unnecessary VMEXITs
  - Do not need to trap guest CR3 and GPT modifications
- Better TLB performance
Other Issues

- **Stack Integrity**
  - create private stack frames by leveraging Multi-HAP
  - only writes in its own frames are propagated to the real kernel stack

- **Write through DMA**
  - IOMMU (Intel VT-d) page tables

- **Architectural state integrity**
  - save architectural state to VMM before transition to untrusted extension
Control Flow Integrity

- Access control for control flow transfers between untrusted extensions and OS kernel
  - All protection state transitions are intercepted by the hypervisor.
  - Kernel control data (e.g., function pointer) are protected by the isolation mechanism
  - Kernel stack frames are also guarded.
Control Flow Integrity

- Trusted Entry Points are a set of addresses specified by OS developer or administrator
  - e.g., a tailored set of kernel APIs to confine certain category of extensions

- Other issues
  - Extension returns to kernel
    - maintain call-return consistencies
  - Interrupt and preemption
Security Analysis

- Change kernel code
  - detected by code integrity protection
- Modify kernel control / non-control data
  - detected by data integrity protection
- Manipulate return addresses / kernel stack frames
  - call-return inconsistencies
  - Kernel stack frame protection

Evaluated with both real-world and homegrown malicious extensions
# Evaluation - Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Untrusted Extensions</th>
<th># of Protection State Transfers</th>
<th>Native Performance</th>
<th>HUKO Performance</th>
<th>Relative Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dhrystone 2</td>
<td>8139too, ext3</td>
<td>N/A</td>
<td>10, 855, 484 lps</td>
<td>10, 176, 782 lps</td>
<td>0.94</td>
</tr>
<tr>
<td>Whetstone</td>
<td>8139too, ext3</td>
<td>N/A</td>
<td>2, 270 MWIPS</td>
<td>2, 265 MWIPS</td>
<td>1.00</td>
</tr>
<tr>
<td>Lmbench (pipe bandwidth)</td>
<td>8139too, ext3</td>
<td>N/A</td>
<td>2, 535 MB/s</td>
<td>2, 213 MB/s</td>
<td>0.87</td>
</tr>
<tr>
<td>Apache Bench</td>
<td>8139too</td>
<td>56, 037</td>
<td>2, 261 KB/s</td>
<td>1, 955 KB/s</td>
<td>0.86</td>
</tr>
<tr>
<td>Kernel Decompression</td>
<td>ext3</td>
<td>17, 471, 989</td>
<td>35, 271 ms</td>
<td>44, 803 ms</td>
<td>0.79</td>
</tr>
<tr>
<td>Kernel Build</td>
<td>ext3</td>
<td>148, 823, 045</td>
<td>2, 804 s</td>
<td>3, 106 s</td>
<td>0.90</td>
</tr>
</tbody>
</table>
Evaluation - Performance

- Major performance cost: protection state transitions
  - Involves guest-to-VMM switch (VMEXIT)

- The more frequent untrusted extension interacts with the kernel, the larger performance penalties
Outline

- Motivation
- Approach Overview
- Key Design & Implementation
- Evaluation
- Summary
Reliability for buggy device drivers

- Microkernels
  - L4 [Liedtke ‘95], MINIX 3 [Herder ‘09]
- Device driver isolation
  - Nooks [Swift ‘03], Mondrix [Witchel ‘05]
- Software fault isolation
  - XFI [Erlingsson ‘06]
Limitation & Future Work

- Labeling Objects at the page-level
  - trade-off: performance vs. security

- Kernel API not designed for isolation/sandboxing
  - invoking APIs may violate integrity properties
  - may need sanitizing & privilege separation

- Tune the OS Kernel
  - e.g., eliminates mixed pages to improve security and efficiency
Thanks!
Questions?
Summary

- HUKO significantly limits the attacker’s ability to compromise the integrity of the kernel.

- Contemporary hardware features may facilitate sandboxing and reference monitoring in the kernel space.