Oakland 2005

Hardware-assisted circumvention of self-hashing software tamper resistance

Glenn Wurster, Paul Van Oorschot, Anil Somayaji

School of Computer Science
Carleton University, Canada
Outline

- Self-hashing tamper resistance overview
- High level overview of our attack
- Hardware memory management design and attack details
- Results and implications
Self-Hashing Tamper Resistance Problem

- Protect an application binary against undetected modifications
  - Verifying that an application for DRM has not been modified
  - Protecting copy protection algorithms
  - Guard against unfair advantages in networked environments
- Do so without dependence on external hardware or software
- Use some form of self-hashing to detect changes
- Assumes a Hostile Host model
Self-Hashing Software Tamper Resistance

➤ Read into the code segment to compute a hash

➤ Rely upon a known good value to detect modifications

➤ Obscure reads into the code by hiding address calculations

➤ Protect the hashing code against alterations
A Network of Hash functions$^{ab}$


Our Results

- Self-hashing is not secure against attack on modern hardware
  - Can modify an application without being detected and without altering hashing algorithms

- Attack applies to proposals including:
  - Chang et al. DRM-2001
  - Horne et al. DRM-2001
  - Aucsmith, IHW-1996 (despite digital signatures)
There does not exist a 1:1 correspondence between virtual and physical addresses.

CPU caches are managed differently depending upon whether they contain information on program instructions or data.
Graphical Representation of Our Attack

Request Address

Instruction Fetch

Data Fetch

Code'
High-Level Overview of Attack

- Create a copy of the application which will remain unmodified
- Modify the application as desired
- Modify the kernel to contain the run-time attack code
- Load the modified application, installing and mapping both original and modified code pages in physical memory
- Run application - attack kernel vectors reads appropriately

☞ This is the core of our attack
Virtual Memory Translation

Virtual Address Space

Page Tables

Operating System

Hardware Translation

Physical Address Space
Hardware Architecture: Virtual Memory Translation

Virtual Address Space → Translation Mechanism → Physical Address Space

- Page Table
- Cache
- DTLB
- ITLB
Alternative Attack Implementations

We use any of several methods to separate code and data reads

- Software TLB miss handlers
- Hardware page table miss handlers
- Hardware segmentation translation
Hardware Translation Mechanism

1. I/D Fetch
   - Hit: Do Translation with TLB
   - Miss: I/D TLB Lookup
     - Hit: Do Translation with TLB
     - Miss: Page Table Lookup
       - Not Present: Trap To OS
       - Present: Fill TLB with PTE

Wurster et al. Page 13 Oakland 2005
Attack Implementation

1. **Trap To OS**
2. **Determine Virtual Address (VA)**
3. **Compute Fetch Type**
   - **Point PTE for VA to Code**
     - **Read from VA**
     - **Clear PTE mapping VA**
     - **Retry Instruction**
     - **Data Fetch**
   - **Instruction Fetch**
     - **Point PTE for VA to Code’**
       - **Map VA’ to same PA as VA**
       - **Overwrite VA’ with Jump; save old**
       - **Jump to VA**
       - **Jumps back to kernel code**
       - **Restore overwritten code**

*Wurster’05*
Filling the ITLB in Generic implementation
Filling the ITLB in Generic implementation

Virtual Address Space  VA  Kernel Memory  VA’

Write Jump Instruction (save overwritten data)

ITLB

Code  Code’
Filling the ITLB in Generic implementation

Virtual Address Space

VA

VA'

Jump to VA

Jump back to Kernel Page

ITLB

Entry for VA

Code'

Code
Filling the ITLB in Generic implementation

Virtual Address Space

VA

VA'

Restore Overwritten Data

Code'

Code

ITLB

Entry for VA
Filling the ITLB in Generic implementation

Virtual Address Space

Remove Mapping to Code

ITLB

Entry for VA

Code' Code

Code' Code
Result of attack

► Code read as data can differ from code executed

► By ensuring code read as data is unmodified code, self-hashing always uses unmodified code – yielding the “correct” hash

► Attack applies to most modern general-purpose processors e.g. UltraSparc, x86, PowerPC, AMD64, Alpha, ARM

*Note:* The attack is not prevented by stealthy address computations

---

\[\text{a}\] Linn et al. *Enhancing Software Tamper-Resistance... ACSAC-2003*
**Overhead**

- **Implementation Work**
  - Must install a modified kernel
  - Per-application overhead is negligible (*copy* command)

- **Run-Time Overhead**
  - Only on a TLB cache miss (0.1% of time on UltraSparc)
  - Each DTLB miss adds 6 assembly instructions on UltraSparc
  - Overhead is less than existing time spent on cache misses
# Variations of the Attack

<table>
<thead>
<tr>
<th>Variation</th>
<th>Oakland’05</th>
<th>TDSC’05</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB Load (Ultrasparc)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Generic Attack</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Segment (x86)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Microcode</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Performance Counters</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Realities of the Attack

- Implemented on the UltraSparc, experimented on x86
- Hash functions need not be found or modified
- Exploits translation and caching capabilities of processor
  - Negligible performance hit
- Attack is possible on wide range of processors
Conclusions

- Typical self-hashing can be subverted on modern processors
- Need new protection to secure self-hashing tamper resistance
  - must withstand real-time detection and separation of code/data
Questions?

http://www.scs.carleton.ca/~gwurster