Raksha

A Flexible Architecture for Information Flow Tracking

Michael Dalton    Hari Kannan    Christos Kozyrakis

Computer Systems Laboratory
Stanford University
Motivation

- Software security is in a crisis
  - Attackers now exploiting wide range of vulnerabilities
- High-level semantic vulnerabilities now most common threats
  - SQL Injection, Cross Site Scripting, Directory Traversal, …
  - Easy to exploit, often architecture & OS independent

- Need a new approach to software security that is
  - Robust
  - End-to-end
    - Can support inter-process communication, operating system
  - Practical
    - Works on unmodified binaries, supports JIT, self-modifying code, …
  - Flexible
    - Can evolve security policies to address future exploits
  - Fast
Dynamic Information Flow Tracking

- **DIFT taints** data from untrusted sources
  - Each word in memory & each register has a taint bit

- **Propagate** taint across instructions
  - Operations with tainted data produce tainted results

- **Check** for unsafe uses of tainted data
  - Tainted code/data pointer dereference
  - Tainted code execution
  - Tainted SQL command
    - http://vulnerable/index.php?username=foo OR 1=1&password=bar OR 1=1

- **Can prevent both memory corruption and semantic bugs**
  - Even on unmodified binaries
DIFT Example: Memory Corruption

Prevent following tainted pointer use:

```c
int idx = *user_input;
buf[idx] = x;
```

Execution timeline:
1. Set r1 to address of tainted data
2. Load tainted data into r2
3. Compute r3 = r3 + r2
   - No bounds check!
4. Dereference r3
   - Store with tainted pointer
   - **Security violation!**

<table>
<thead>
<tr>
<th>R1 tag</th>
<th>R2 tag</th>
<th>R3 tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Checkmark]</td>
<td>![X]</td>
<td>![X]</td>
</tr>
</tbody>
</table>

© 2007, Michael Dalton
Limitations of Current DIFT Systems

- **Software-only approaches are flexible but not fast or practical**
  - Can create arbitrary security policies, but the slowdown can be $\geq 3x$
  - Cannot handle multithreaded apps, cross-process issues, OS, …

- **Hardware-only approaches are fast & practical but not flexible or safe**
  - Fine-grain checks & propagation done by HW with no slowdown
    - Works with multithreaded apps, JIT code, …
  - Single fixed policy for memory corruptions
    - Leads to false positives & negatives with real-world code [WDDD’06]
    - Cannot deal with semantic vulnerabilities
  - OS-level security exceptions are slow and rely on fully trusted OS
    - Too expensive to extend hardware policy with software analysis

- **Need a new approach that combines HW + SW advantages**
  - HW: fast & practical fine-grain checks & propagation
  - SW: policies & decision making, handle high-level issues & corner cases
Raksha Architecture Overview

- Raksha: a HW+SW architecture that follows the DIFT model
  - All state is extended by a 4-bit tag for tainting (registers & memory)
  - HW instructions propagate tags from sources to destinations
  - HW instructions check tags to identify security exceptions
  - Propagation & checks are transparent to application code

- New features
  - Software-controlled check & propagate policies ⇒ flexibility
    - Specify policy using check & propagate control registers
    - Fine-grain software control to avoid common security pitfalls
    - Flexibility allows us to catch a wide range of bugs
  - Up to 4 concurrently active policies ⇒ robustness
    - One policy per tag bit
    - Provides comprehensive protection against multiple bugs
  - Low-overhead, user-level, security traps ⇒ end-to-end, flexibility
    - Small overheads for extending HW with software analysis
    - Can check most of the OS code
Raksha Implementation Overview

- **Goal:** catch wide range of bugs on unmodified, real-world apps

- **Full-system HW prototype based on LEON 3**
  - Open source processor, SPARC V8 compliant
  - Modified its pipeline, caches, memory controller to support DIFT
    - See ISCA’07 paper for hardware details
  - Synthesized and mapped on a Virtex-2 FPGA board

- **Linux 2.6-based SW systems**
  - Custom Linux distribution to use DIFT support
    - Security policies implemented as preloaded shared libraries
    - OS modified for tag support (e.g., save/restore tag registers on traps, …)
  - Full-featured server system
    - Full GNU toolchain: GCC, glibc, binutils, …
    - Enterprise software: Apache, postgresql, wu-ftp, proftpd, …
    - Over 120 packages
Raksha Prototype Board
Policy Specification

Tag Check Register (Check Enables)

<table>
<thead>
<tr>
<th>25</th>
<th>23 22</th>
<th>20 19</th>
<th>17 16</th>
<th>14 13</th>
<th>12 11</th>
<th>10 9</th>
<th>8 7</th>
<th>6 5</th>
<th>2 1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUST 3</td>
<td>CUST 2</td>
<td>CUST 1</td>
<td>CUST 0</td>
<td>LOG</td>
<td>COMP</td>
<td>ARITH</td>
<td>FP</td>
<td>MOV</td>
<td>EXEC</td>
<td></td>
</tr>
</tbody>
</table>

Other Predefined operation: Check Enables
- [0] Source Check Enable (On/Off)
- [1] Destination Check Enable (On/Off)

Execute operation Check Enables
- [0] Address (Program Counter) Check Enable (On/Off)
- [1] Instruction Check Enable (On/Off)

Custom operation Check Enables
- [0] Source 1 Check Enable (On/Off)
- [1] Source 2 Check Enable (On/Off)
- [2] Destination Check Enable (On/Off)

Move operation Check Enables
- [1] Source Check Enable (On/Off)
- [2] Source Address Check Enable (On/Off)
- [3] Destination Check Enable (On/Off)
- [4] Destination Address Check Enable (On/Off)

- One check & propagate register per active security policy
  - Policies specified at granularity of primitive operation
    - Int/FP arithmetic, move, logical, comparison, execute
  - Instructions are decoded into ≥1 primitive operations
    - Apply rules specified by check/prop regs to each operation

- Flexible policies critical for security
  - Must support diverse range of analyses to thwart various attacks
Low Overhead Security Traps

- Security trap invokes pre-registered handler
  - Handler in same address space & privilege level as application
  - Handler invocation triggers a special “trusted mode”
    - Can read/write tags, tag registers
  - Overhead similar to function call instead of full OS trap

- A security policy is used to protect handler from user code
  - Tag handler code & data with a sandboxing policy
  - Sandbox prevents load/store/execute outside of trusted mode

- Benefits
  - Can apply DIFT to (most of the) OS
    - Reduce the amount of code you really trust
  - Combined HW and SW security analysis is practical
    - HW should can be conservative with security checks
    - Low overhead for filtering false positives through software analysis
## Security Results with Unmodified Binaries

<table>
<thead>
<tr>
<th>Program</th>
<th>Attack</th>
<th>Detected Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>traceroute</td>
<td>Double Free</td>
<td>Tainted data ptr</td>
</tr>
<tr>
<td>polymorph</td>
<td>Buffer Overflow</td>
<td>Tainted code ptr</td>
</tr>
<tr>
<td>Wu-FTPD</td>
<td>Format String</td>
<td>Tainted ‘%n’ in vfprintf format string</td>
</tr>
<tr>
<td>gzip</td>
<td>Directory Traversal</td>
<td>Open tainted dir</td>
</tr>
<tr>
<td>Wabbit-PHP</td>
<td>Directory Traversal</td>
<td>Escape Apache root with tainted ‘..’</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>Command Injection</td>
<td>Execve tainted file</td>
</tr>
<tr>
<td>ProFTPD</td>
<td>SQL Injection</td>
<td>Tainted SQL command</td>
</tr>
<tr>
<td>htdig</td>
<td>Cross-Site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
<tr>
<td>PhpSysInfo</td>
<td>Cross-Site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
<tr>
<td>Scry-PHP</td>
<td>Cross-Site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
</tbody>
</table>
Security Policy for Cross-Site Scripting

- **Taint** incoming data from network
  - E.g., http://vulnerabe/user.php?op=userinfo&uname=<script>alert(document.cookie);</script>

- **Propagate** on all operations
  - Move, integer arithmetic, logical

- **Check** all Apache send & write system calls
  - Software handler scans output HTML for unsafe tainted strings
  - Can kill Apache if violated
  - …or recover by HTML encoding tainted output
    - Change “ to &quot, < to &lt, etc

- **Policy is independent of Apache modules & languages used**
  - Experiments included C, C++, PHP Modules
Memory Corruption Policy

- **Taint** data from untrusted sources
  - Files, environment, …

- **Propagate** on Move, Integer Arithmetic, Logical

- **Untaint** on bounds check
  - Handle untainting in software using flexible checks
  - Compare tainted vs. untaint ⇒ bounds check
  - Tainted & untainted ⇒ bounds check, if untainted is $2^x - 1$

- **Terminate** program when corruption detected
  - Load/store tainted address
  - Tainted jump address
  - Tainted code
Lessons from Semantic Bug Detection

- Checks at system call boundaries are sufficient
  - Well-defined semantics; complete mediation
  - Language & library independent
  - Low checking overhead

- Fine-grained propagation crucial
  - Must know what bytes are tainted to apply policies
    - I.e., application’s normal javascript vs. malicious user <script>

- Did not need to recognize verification functions
  - Validating all output at system call is sufficient

- Cross process information flow important
  - Real-world applications use Unix sockets & pipes for IPC
Lessons from Memory Corruption Detection

- Difficult to recognize checks for untainting

- Ambiguity causes false positives/negatives in real-world apps
  - Not all comparisons are bounds checks (e.g., traceroute)
  - Not all bounds checks are comparisons (e.g., gcc)
  - Not all tainted indices are security violations
    - E.g., tainted bytes used to safely index into 256-entry table (glibc)

- These are fundamental issues for both HW & SW DIFT systems
  - Difficult to address without unambiguous interfaces
  - Need language or compiler support to provide full protection
    - Provide bounds info for objects or define bound check instructions
Conclusions

- Raksha is a flexible DIFT architecture for security
  - Multiple software-controlled tag policies
  - Low overhead security exceptions

- Raksha philosophy
  - Combine HW + SW techniques to get best of both words
  - HW: fast & practical fine-grain checks & propagation
  - SW: policies & decision making, handle high-level issues & corner cases
  - Practical, flexible, and fast

- Full-system Raksha prototype
  - HW: modified Sparc v8 processor on an FPGA board
  - SW: modified Linux 2.6 that manages security policies
  - Demonstrated exploit prevention in real-world, unmodified binaries
    - From web attacks to buffer overflows

- For further details refer to:
  - "Deconstructing Hardware Architectures for Security," the 5th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD @ at ISCA), June 2006
Future Work

- **Vulnerability protection for OS code**
  - User/kernel pointer dereference, kernel buffer overflow, …

- **Policy inference**
  - Infer policies for system calls
    - I.e. infer Apache webroot directory for directory traversal
  - Infer memory corruption false positives
    - Whitelist offending instructions and data

- **Whole system information flow**
  - Across processes & files
  - Experiment with more flexible notion of trust and taintedness
    - Trust data from user ‘admin’ but not group ‘nobody’

- **Minimize TCB of Information Flow OS**
  - Collaboration with HiStar group at Stanford