Whole-System Dynamic Binary Analysis

Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis
Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, Engin Kirda,
Appeared in CCS’07
Outline

- Motivation
- Overview
- Design & Implementation: Panorama
- Taint-Graph Based Detection and Analysis
- Evaluation
- Summary

Motivation I -- Problem

- Malicious code creeps into users’ computers, performs malicious behaviors
  - spyware/adware
  - keyloggers
  - password thieves
  - network sniffers
  - backdoors
  - rootkits
- Even software from reputable vendors
  - Google Desktop
  - SONY Media Player
Motivation II – Previous Solutions

• Malware Detection
  • Signature based
    • Cannot detect new malware and variants
    • Semantic-aware signatures can detect some variants
  • Behavior based
    • Heuristics: high false positives and false negatives
    • Strider Gatekeeper checks auto-start extensibility points
    • VICE and System Virginity Verifier check various hooks

• Malware Analysis
  • Manual process mostly
  • Coarse-grained

Outline

• Challenges & Motivation
• Overview
  • Design & Implementation: Panorama
  • Taint-Graph Based Detection and Analysis
• Evaluation
• Summary
Overview I – Our Observation

- Information access and processing (IAP) behavior
  - Many different kinds of malware present malicious/suspicious IAP behavior
  - Steal, tamper, or leak sensitive information
    - Spyware leaks URLs
    - Keyloggers steals keystroke information
    - Password thieves steals passwords
    - Rootkits tamper with directory information
    - Network sniffers eavesdrop the network traffic

Overview II - A Example
Overview III – Our Approach

- Whole-system dynamic taint analysis with OS awareness
  - Run the system to be analyzed in an emulator
  - Selectively mark data as tainted
  - Monitor taint propagation
  - Extract OS-level knowledge
  - Generate taint graphs
  - Taint-graph based detection and analysis

Overview II – Big Picture
Outline

• Motivation
• Overview
• Design & Implementation: Panorama
  • Hardware-level Dynamic taint analysis
  • OS-aware Analysis
  • Automated testing
• Taint-Graph Based Detection and Analysis
• Evaluation
• Summary

Design & Implementation – Hardware Level Taint Analysis

• Build on QEMU
• Shadow Memory
  • RAM, registers, hard disk, and NIC buffer
  • Page-table-like structure
• Extend CPU
  • Propagate taint status for each instruction
• Extend Kbd, Disk and NIC
  • Taint inputs
  • For disk, propagate taint status
Design & Implementation – Hardware-Level Taint Analysis (2)

- Instrument CPU Instructions (at byte granularity)
  - Movement: MOV AL, BH
    -- AL is tainted iff BH is tainted
  - Arithmetic: ADD EAX, EBX
    -- EAX is tainted iff EAX or EBX is tainted
  - Table lookup: MOV EAX, [EBX]
    -- EAX is tainted if EBX or MEM[EBX] is tainted
  - Constant function: XOR EAX, EAX
    -- EAX will be untainted

Design & Implementation – OS-Aware Analysis

- Resolving process and module information
  - Q: when an instruction accesses tainted, which process and module is it from?
  - A: A kernel module is inserted into the guest system

- Resolving filesystem information
  - Q1: when tainting a file/directory, which disk blocks should be tainted?
  - Q2: when the tainted data propagate to a disk block, while file is tainted?
  - A: The Sleuth Kit (TSK), a disk forensic tool

- Resolving network information
  - Q1: When tainting an incoming packet, which connection is it from?
  - Q2: when a tainted byte is sent out, which connection is it from?
  - A: Simply check the packet header
Design & Implementation – OS-Aware Analysis (2)

• How to identify the actions performed by the code sample?
• Challenge 1: packed code and encrypted code
  • A: taint the binary file with a special label
• Challenge 2: call a function in the system libraries
  • A:
    • check stack pointers
    • Check asynchronous kernel functions

Design & Implementation – Automated Testing

• Goal
  • Perform test cases without human intervention
  • Introduce tainted information sources
• We use “AutoHotkey”
  • Record the test cases into scripts
  • Replay the scripts in Panorama
  • Will describe the test cases later
Outline

• Motivation
• Overview
• Design & Implementation: Panorama
• Taint-Graph Based Detection and Analysis
  • Taint Graph
  • Taint-Graph Based Policies
• Evaluation
• Summary

Detection & Analysis – Taint Graph

• Taint Graph
  • Input 1: Raw events present dependencies among instructions, hardware inputs and outputs
  • Input 2: OS-level Knowledge
  • Output: taint graph

Diagram:
- Raw events
- OS-level knowledge
- Taint graphs
Detection & Analysis – Taint Graph(2)

- An example of taint graph
  - This graph reflects the procedure for Windows user authentication.
  - A password thief catches the password and saves them into a log file

Detection & Analysis – Taint-Graph Based Detection

- Anomalous information access
  - text: when sending keystrokes to a text editor, a command console, keyloggers ...
  - password: when sending passwords to a web form, a password field, password thieves and keyloggers ...
  - ICMP: when pinging a remote host, packet sniffers and stealth backdoors ...
  - FTP: when logging into an FTP server, packet sniffers and stealth backdoors ...
  - UDP: when sending in a UDP packet, packet sniffers and stealth backdoors ...
  - Others: ...
Detection & Analysis – Taint-Graph Based Detection (2)

- Anomalous information leakage
  - URL: the keystrokes sent to the address bar,
  - HTTP: the incoming HTTP traffic,
  - HTTPS: the incoming HTTPS traffic,
  - document: .txt, .pdf, .ppt, .doc
  - Others: ...

Detection & Analysis – Taint-Graph Based Detection (3)

- Excessive information Access
  - directory: when recursively listing several directories, the disk blocks belonging to the directories
  - Rootkits will access all of the disk blocks and tamper with some entries
  - Compared with Cross-view based techniques, such as Rootkit Revealer, Blacklight, and Strider Ghostbuster, ...
Detection & Analysis – Taint-Graph Based Detection

<table>
<thead>
<tr>
<th>Test case description</th>
<th>Introduced inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Edit a text file and save it</td>
<td>text, document</td>
</tr>
<tr>
<td>2. Enter password in a GUI program</td>
<td>password</td>
</tr>
<tr>
<td>3. Log in a secure website</td>
<td>URL, password, HTTPS</td>
</tr>
<tr>
<td>4. Visit several websites</td>
<td>URL, HTTP</td>
</tr>
<tr>
<td>5. Log into an FTP server</td>
<td>text, password, FTP</td>
</tr>
<tr>
<td>6. Recursively list a directory</td>
<td>directory</td>
</tr>
<tr>
<td>7. Send UDP packets into the system</td>
<td>UDP</td>
</tr>
<tr>
<td>8. Ping a remote host</td>
<td>ICMP</td>
</tr>
</tbody>
</table>

Detection & Analysis -- Taint-Graph Based Detection

\[ \forall g \in G, (\exists v \in g.V, v.type = \text{module}) \land \\
\quad g.root.type \in \{\text{text, password, FTP, UDP, ICMP}\} \land \\
\quad \forall u \in \text{descendants}(v), u.type \in \{\text{file, network}\} \land \\
\quad \neg \text{Violate}(v, \text{"No Access"}) \]  \tag{1}

\[ \exists g \in G, (\exists v \in g.V, v.type = \text{module}) \land \\
\quad (g.root.type \in \{\text{URL, HTTP, HTTPS, document}\}) \land \\
\quad \forall u \in \text{descendants}(v), u.type \in \{\text{file, network}\} \land \\
\quad \text{Violate}(v, \text{"No Leakage"}) \]  \tag{2}

\[ \forall g \in G, g.root.type = \text{directory} \rightarrow \exists v \in g.V, v.type = \text{module} \land \\
\quad \neg \text{Violate}(v, \text{"No Excessive Access"}) \]  \tag{3}
Outline

- Motivation
- Overview
- Design & Implementation: Panorama
- Taint-Graph Based Detection and Analysis
- Evaluation
  - Malware detection
  - Malware analysis
  - Performance
- Summary

Evaluation – Malware Detection

<table>
<thead>
<tr>
<th>Category</th>
<th>Total</th>
<th>FNs</th>
<th>FPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyloggers</td>
<td>5</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Password thieves</td>
<td>2</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Network sniffers</td>
<td>2</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Stealth backdoors</td>
<td>3</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Spyware/adware</td>
<td>22</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Rootkits</td>
<td>8</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Browser plugins</td>
<td>16</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Multi-media</td>
<td>9</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Security</td>
<td>10</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>System utilities</td>
<td>9</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Office productivity</td>
<td>4</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Games</td>
<td>4</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Others</td>
<td>4</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Sum</td>
<td>98</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Browser accelerator

Personal firewall
Evaluaton – Malware Analysis

Google Desktop obtains the incoming HTTP traffic, saves it into two index files, and then sends it out though an HTTPS connection, to a remote Google Server.

Evaluation – Performance

- curl, scp, gzip, bzip2: 20 times slowdown on average
- Test cases: 10~15 mins
- Performance improvement:
  - On-demand emulation
  - Static analysis
Summary

- Propose to rely on IAP behavior to detect and analyze malware
  - No signature is required: can detect new malware
  - Stems from intent: difficult to evade
  - Fine grained analysis
  - Capture the behaviors of kernel-level attacks
- Propose to use the technique of whole-system dynamic taint analysis with OS-awareness to capture IAP behavior
- Design and develop a system Panorama
  - Yields no false negative and very few false positives
  - Correctly capture the behavior of Google Desktop

Make It Work, Make It Right, Make It Fast: Building a Platform-Neutral Whole-System Dynamic Binary Analysis Platform

Andrew Henderson*, Aravind Pravash*, Lok Kwong Yan†,
Xunchao Hu*, Xujiewen Wang*, Rundong Zhou*, Heng Yin*

* Department of EECS, Syracuse University
† Air Force Research Laboratory, Rome
**Motivation:** We need a practical solution for platform-neutral whole-system binary analysis

- **Binary analysis of malware**
  - No source code available to us
  - Need to analyze malicious binary activity

- **Whole system**
  - Multiple components in both userspace and kernel

- **Platform-neutral (as much as possible)**
  - Architecture neutral
  - Guest OS neutral

**DECAF:** System Architecture

**DECAF and Guest Environment**

**Plugins**

- API Tracer
- Keylogger Detector
- Instruction Tracer
- ...
Does DECAF work?

- Sycure Lab (Syracuse University) actively uses DECAF for our cybersecurity research efforts
- Sycure Lab team is using DECAF for the Cyber Grand Challenge competition
- McAfee currently uses DECAF to detect and analyze keylogger malware behaviors
- Numerous other academic labs are currently utilizing DECAF in their own research efforts

Just-In-Time VMI

- Virtual machine introspection (VMI)
  - Inspect the guest environment from the outside
  - Bridge the “semantic gap”
- Other VMI implementations focus on how, not when
  - We must be aware of changes within the guest when those changes occur
- VMI must be as platform-neutral as possible
- VMI must introduce minimal overhead
Just-In-Time VMI

• Observation 1: A process must have its own memory space
  • Each CPU architecture provides a register to store the “base” of these memory spaces (CR3 in x86, CP15 in ARM, etc.)

• Observation 2: The translation look-aside buffer (TLB) reveals information about guest behavior
  • An “execute” cache miss will occur when new code pages are loaded and executed (new process, loading shared libraries, context switch)

• Observation 3: Location and structure of key kernel data structures are known
  • Kernel contains linked lists of modules, processes, threads

• Result: Rely on hardware events to discover “when” and “what”, rely on kernel data for “who”

Just-In-Time VMI: Solution

• TLB Miss triggers VMI
• PC tells us where event occurred

• Guest kernel data structures give more detail

• Other systems perform VMI using guest software:
  • Hook system calls
  • Use kernel module
  • Use custom device driver
  • Increases dependence on guest platform
Tainting

• Tainting must be whole-system
  • Tainted data should be trackable throughout the entire guest environment (kernel, processes, devices)

• Tainting policy must be sound and precise
  • Minimize under- and over-tainting of data
  • We performed formal verification of our taint policy correctness at the instruction level [1]

• Tainting must be fast


Tainting: Using QEMU for propagation

• QEMU’s Tiny Code Generator (TCG) is a binary translator
  • Guest CPU instructions are translated into intermediary representation (IR) instructions
  • TCG’s IR instruction set implements standard CPU operations that all instruction sets have (MOV, ADD, XOR, etc.)
  • These IRs and then translated into host CPU instructions

• Execution details of the IRs and their arguments are invisible to the guest
**Tainting**: Lightweight inline propagation

- Begin with guest instructions
- Translate guest instructions into IR
- Analyze each IR to determine taint rule to apply
- Insert taint propagation IRs

**Tainting**: Heavyweight plugin propagation

- Taint *state* is propagated inline via IRs
- When tainted data is present, the IRs can be logged to disk via a plugin
- Taint tags are written to this log when created
- The generated log is sliced backward to reconcile taint with its source tag
Event-Driven Instrumentation

- Instrumentation occurs at two points:
  - Translation-time
  - Runtime
- At translation time, callbacks are embedded in the TCG IR stream
- At runtime, DECAF uses a dispatch mechanism to route these callbacks to plugins
- Example: Shared library
  - Are we in the right process?
  - Should the plugin’s callback be triggered?

Event-Driven Instrumentation: Translation time

- Begin with guest ops
- Translate guest ops into IRs
- Insert helper functions to mark begin/end of block
- Insert helper functions to mark begin/end of guest op
- Either the whole-system or just modules of interest can be instrumented
Event-Driven Instrumentation:
A sample tainted keystroke plugin

```c
1. plugin_interface_t my_interface;
2. DECAF_Handle handle_read_taint_mem = DECAF_NULL_HANDLE;
3. int taint_key_enabled = 0;
4. void my_read_taint_mem(DECAF_Callback_Params *param)
   { char name[128];
   7. tmoinfo_t tm;
   8. if(VMI_locate_module_c(DECAF_getPC(cpu_single_env),
       DECAF_getPGD(cpu_single_env),name,&tm)
   == 0)
   9. DECAF_printf("INSN 0x%08x From Module %s Read Keystroke
       ", DECAF_getPC(cpu_single_env),tm.name);
   }
10. void my_send_keystroke(DECAF_Callback_Params *params)
   { *params->ks.taint_mark = taint_key_enabled;
   12. taint_key_enabled = 0;
   13. DECAF_print("taint keystroke %d\n",params->ks.keycode);
   }
14. void do_taint_sendkey(Monitor *mon,const QDict *qdict)
   { if (qdict_haskey(qdict,"key") { 15. if (qdict_haskey(qdict,"key") { 16. taint_key_enabled = 1; //enable keystroke taint
17. do_send_key(qdict_get_str(qdict,"key"); //Send the key
   }
18. mon_cmd_t my_term_cmds[] = {
   19. .name = "taint_sendkey",
   20. .args_type = "key",
   21. .handler.cmd = do_taint_sendkey,
   22. .params = "taint_sendkey key",
   23. .help = "Send a tainted key to system"
   },
   24. void my_cleanup(){......}
}
```

Evaluation: VMI performance

SPEC CPU2006
Windows: 12%
Linux: 14%

Common Case:
OS Boot Time
**Evaluation:** Tainting performance

- Tainting experiences 605% overhead on SPEC CPU2006
- Heaviest performance impact on CPU-bound benchmarks

<table>
<thead>
<tr>
<th>Tainting Software</th>
<th>Whole System</th>
<th>Guest OS</th>
<th>Arch Support</th>
<th>Bitwise Granularity</th>
<th>Expected Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drten</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>30x</td>
</tr>
<tr>
<td>LIFT</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>3.6x</td>
</tr>
<tr>
<td>Ibufti</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>3.05x</td>
</tr>
<tr>
<td>Minamoo</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>2.3x</td>
</tr>
<tr>
<td>Memoryfi</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>26x</td>
</tr>
<tr>
<td>TaintBochs</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>50x</td>
</tr>
<tr>
<td>TEMU</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>20x</td>
</tr>
<tr>
<td>DECAF</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>6x</td>
</tr>
</tbody>
</table>

**Evaluation:** HookAPI plugin performance

- Internet Explorer
- TDSS (normalized)
- Google Chrome

Execution Time (Seconds)
Evaluation: Development effort

<table>
<thead>
<tr>
<th>Software</th>
<th>OS/Arch-Independent (LOC)</th>
<th>OS/Arch-Specific (LOC)</th>
<th>Total (LOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECAF</td>
<td>18470</td>
<td>1350</td>
<td>19820</td>
</tr>
<tr>
<td>Insn Tracer</td>
<td>3770</td>
<td>90</td>
<td>3860</td>
</tr>
<tr>
<td>API Tracer</td>
<td>840</td>
<td>880</td>
<td>1720</td>
</tr>
<tr>
<td>Key Logger</td>
<td>120</td>
<td>0</td>
<td>120</td>
</tr>
</tbody>
</table>

- Most architecture-specific code is related to accessing CPU registers
- Most OS-specific code is related to VMI

Conclusion

- DECAF provides whole-system emulation and instrumentation that works correctly and is fast
- DECAF is open source and available for download:

  [https://github/sycurelab/decaf](https://github/sycurelab/decaf)