Automated Detection of Persistent Kernel Control-Flow Attacks

Michael Hicks
University of Maryland

Joint work with Nick Petroni
Long-lived, healthy software?

- The whole lifetime of Methuselah was nine hundred and sixty-nine years ... —Genesis 5: 27

- There is no Methuselah of software today
  - Software bugs cause systems to crash and misbehave, and to be vulnerable to attack
    - $billions lost annually, millions of machines compromised
  - Systems must be stopped and restarted to apply fixes and improvements when available
    - Downtime = inconvenience, liability, and/or lost $$$

- My research goal:
  **How can we build available, reliable, and secure software?**
Some Projects

• **Software Security**
  – Automated rootkit detection [CCS 07]
  – BEEP [WWW 07]
  – RX [FCS 05, CSFW 06] and secure tags [COORD 05]

• **Dynamic Software Updating for Availability**
  – Dynamically updateable TAL [PLDI 01, TOPLAS 05]
  – Ginseng: DSU infrastructure for C [POPL 05, PLDI 06, POPL 08]

• **Compilers and Static Analysis for Reliability**
  – Cyclone: safe C [PLDI 02, USENIX 02, ISMM 04, SCP 06]
  – Locksmith: static race prevention [PLDI 06, SAS 06, TRANSACT 06]
  – CMod: backward-compatible modules for C [TLDI 07]
  – Transparent Proxies for Java [OOPSLA 04]
Some Projects

- **Software Security**
  - Automated rootkit detection [CCS 07]
  - BEEP [WWW 07]
  - RX [FCS 05, CSFW 06] and secure tags [COORD 05]

- **Dynamic Software Updating for Availability**
  - Dynamically updateable TAL [PLDI 01, TOPLAS 05]
  - Ginseng: DSU infrastructure for C [POPL 05, PLDI 06, POPL 08]

- **Compilers and Static Analysis for Reliability**
  - Cyclone: safe C [PLDI 02, USENIX 02, ISMM 04, SCP 06]
  - Locksmith: static race prevention [PLDI 06, SAS 06, TRANSACT 06]
  - CMod: backward-compatible modules for C [TLDI 07]
  - Transparent Proxies for Java [OOPSLA 04]
Motivation

• Attackers compromise systems

• Attack objectives
  – Maintaining control
  – Perform reconnaissance
    • Keystroke logging
    • Packet sniffing
    • File exfiltration
  – Neutralize target’s defenses

⇒ All the while, trying to avoid detection
Mechanisms for Attack

- Attacks may target different levels of the hardware/software stack
  - Example: hiding a malicious process:

- The kernel is an attractive target
  - Greater access, harder to protect
Goal: Detect Kernel Integrity Violations

• Attacker is motivated to introduce persistent code to perform surreptitious actions
  – Keystroke logging, backdoor, etc.

• Such code constitutes a deviation from the kernel’s assumed control flow

• Idea: enforce **Control-Flow Integrity (CFI)**
  – Ensure that the system does not deviate from valid, precomputed control flow graph
Approach: State-based CFI Monitor

• Examine kernel state asynchronously from an isolated monitor
  – Can implement in a VMM or a separate card
  – Simplifies protecting the monitor from tampering

• Monitor looks for changes that suggest a potential violation of CFI
  – In contrast to observing execution as it occurs
Results

• We implemented a state-based CFI (SBCFI) monitor for the Linux kernel using Xen and VMware

• The monitor detected all CFI-violating rootkits we could install
  – 17 of 18 violated CFI for the kernel we were monitoring

• Monitoring every 1 second or more imposed less than 1% overhead
Integrity Threat Example

Kernel Memory

1. Attacker gains entry

2. Attacker inserts code

3. Attacker redirects control-flow
Integrity Threat Example

Kernel Memory

1. Attacker gains entry

2. Attacker inserts code

3. Attacker redirects control-flow

   Modifies system call table
Integrity Threat Example

1. Attacker gains entry

2. Attacker inserts code

3. Attacker redirects control-flow

Modifies kernel code
Integrity Threat Example

Kernel Memory

1. Attacker gains entry
2. Attacker inserts code
3. Attacker redirects control-flow
   Modifies heap pointer
Linux Kernel Rootkit Analysis

- Analyzed 25 Linux rootkits
- Mechanism
  - 24 (96%) modify kernel CFG
    - All exhibit persistent CFG changes
  - 1 does not
    - Only non-control data
- Functionality
  - Hiding: 18 (72%)
  - Privilege escalation: 13 (52%)
  - Backdoors: 6 (24%)
  - Reconnaissance: 6 (24%)
  - Attack defense mechanisms: 1 (4%)
- Similar trends for Windows rootkits
Control-flow Integrity (CFI)

- Enforce that execution proceeds according to a precomputed control-flow graph (CFG)
  - Enforcement can be via an in-line reference monitor

- Would prevent our example attacks
  - 24 of 25 rootkits we analyzed would violate CFI
State-based CFI

• Check system state periodically
  – State consists of current code, data, registers
  – Compare against acceptable CFG approximation
    • Same as booted kernel

• Tradeoffs
  – Approximates true CFI
    • Will not detect:
      – Transient modifications
      – Stack-based attacks
    • Rootkits still detected (CFG changes persistent)
  + Enables use of external, protected mechanism
  + Tunable performance impact
False positives
- Possible due to asynchronous execution
- Mitigated by verifying before alarm
- None experienced in practice
Alternative: Copilot

- PCI-based integrity monitor
- Uses DMA to analyze system state
- More isolation, but less access (registers, cache)

Petroni, Fraser, Molina, Arbaugh
SBCFI for the Linux Kernel

• Validate kernel text
  – Cryptographic hash comparisons with trusted copies
  – Validate static portion of kernel
  – Traverse module list, validate module text
    • Requires loader emulation
  – Report unauthorized/modified code

• Validate dynamic control-flow
  – Indirect function calls (i.e., function pointers)
  – Function call returns
  – Computed jumps (e.g., first-class goto labels)
Validating Function Pointers

• Step 1: Locate function pointers in heap
  – GC-like traversal of kernel memory
  – Start at set of “roots” (global variables)
  – Traverse heap looking for objects with fps
  – Use source-code types as specification

• Step 2: Validate function pointers, based on various approximations of CFG
  – Valid code region
  – Valid function
    • Any function
    • Valid type
    • Valid points-to set
Traversing the Heap

- Start with set of “roots”
  - Global variables (extracted from source)
  - Registers

- Construct graph from source types

- Generate graph traversal code
Example: Type Graph Generation

```c
struct fs_struct {
    struct dentry *root;
    struct vfsmount *rootmnt;
};
```
Example: Type Graph Generation

```c
struct fs_struct {
    struct dentry *root;
    struct vfsmount *rootmnt;
};
struct dentry {
    struct dentry *d_parent;
    struct inode *d_inode;
};
```
Example: Type Graph Generation

```c
struct fs_struct {
    struct dentry  *root;
    struct vfsmount *rootmnt;
};
struct dentry {
    struct dentry  *d_parent;
    struct inode   *d_inode;
};
struct vfsmount {
    struct vfsmount  *mnt_parent;
    struct dentry    *mnt_root;
    struct super_block *mnt_sb;
};
```
Graph Generation

- **Challenge:** insufficient information
  - `void *`
  - Embedded lists
  - Unions
  - Dynamic arrays

- **Example:**
  ```c
  struct super_block {
    struct list_head s_list;
    struct dentry *s_root;
  };

  struct list_head {
    struct list_head *next;
    struct list_head *prev;
  };
  ```

Wrong:

```
```
```
```
Graph Generation

- **Challenge:** insufficient information
  - void *
  - Embedded lists
  - Unions
  - Dynamic arrays

- **Example:**

```c
struct super_block {
    // @headed, @type super_block(s_list)
    struct list_head   s_list;
    struct dentry     *s_root;
};
struct list_head {
    struct list_head *next;
    struct list_head *prev;
};
```

**Total:** 123 type defs annotated, 39 global var defs
Experimental Results: Detection

• Installed Redhat 7.3 (Linux 2.6) as the target OS
  – Of the 25 rootkits we could find, we could install 18 on this platform

• Monitor was able to detect 17 of the 18
  – No false alarms
  – The undetected rootkit modified a kernel data structure to hide a process; kernel control flow unchanged
Traversal Statistics

Offline Generation
- 1,049 types
- 22,182 globals
  - 5,105 functions
  - 660 contain/lead to fps

Runtime
- Objects traversed: 69,796 ave., 78,044 max
- FPs validated: 41,978 ave., 44,303 max
Experimental Results: Overhead

• Measured the degradation of benchmark performance when running the monitor
  – SPECWeb 05 and SPECCPU 06 benchmarks
  – Varied monitor frequency from never to continuous
  – Ran monitor on separate processor (2 CPU setup) and on the same processor

• Measurements taken on a Precision 490 Workstation: 2.6 GHz 4-core Xeon, 4 GB RAM, gcc 3.4.6, Xen 3.1.0

• Results
  – For 2-core case, SBCFI imposes less than 1% overhead over the VMM when monitoring at 1s intervals
## Experimental Results – SPECCPU2006 (2CPU)

<table>
<thead>
<tr>
<th></th>
<th>Total time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>10001</td>
</tr>
<tr>
<td>Xen only</td>
<td>10610</td>
</tr>
<tr>
<td>Xen + 3s</td>
<td>10634</td>
</tr>
<tr>
<td>Xen + 1s</td>
<td>10687</td>
</tr>
<tr>
<td>Xen Continuous</td>
<td>10796</td>
</tr>
</tbody>
</table>
Xen SPECCPU2006 (2CPU)

Configuration

Time (s)

Xen only
Xen + 3s SBCFI
Xen + 1s SBCFI
Xen + Cont. SBCFI
### Experimental Results – SPECweb2005 (2CPU)

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Byte Rate (bits/sec)</th>
<th>% Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xen only</td>
<td>94900.1</td>
<td>0%</td>
</tr>
<tr>
<td>Xen + 3s SBCFI</td>
<td>94351.3</td>
<td>0.6%</td>
</tr>
<tr>
<td>Xen + 1s SBCFI</td>
<td>94941.7</td>
<td>0%</td>
</tr>
<tr>
<td>Xen Continuous</td>
<td>95018.3</td>
<td>-0.1%</td>
</tr>
</tbody>
</table>
Xen SPECweb2005 (2CPU)
## Experimental Results – SPECCPU2006 (1CPU)

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Total time (s)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xen only</td>
<td>10882</td>
<td>0.0%</td>
</tr>
<tr>
<td>Xen + 90s SBCFI</td>
<td>10976</td>
<td>0.8%</td>
</tr>
<tr>
<td>Xen + 75s SBCFI</td>
<td>11005</td>
<td>1.1%</td>
</tr>
<tr>
<td>Xen + 60s SBCFI</td>
<td>11024</td>
<td>1.3%</td>
</tr>
<tr>
<td>Xen + 30s SBCFI</td>
<td>11164</td>
<td>2.6%</td>
</tr>
<tr>
<td>Xen + 20s SBCFI</td>
<td>11284</td>
<td>3.7%</td>
</tr>
<tr>
<td>Xen + 10s SBCFI</td>
<td>11652</td>
<td>7.0%</td>
</tr>
<tr>
<td>Xen + 5s SBCFI</td>
<td>12294</td>
<td>13.0%</td>
</tr>
</tbody>
</table>
### Experimental Results – SPECweb2005 (1CPU)

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Byte Rate (bits/sec)</th>
<th>Change (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xen only</td>
<td>92892.1</td>
<td>0%</td>
</tr>
<tr>
<td>Xen + 3s SBCFI</td>
<td>92982.9</td>
<td>-0.1%</td>
</tr>
<tr>
<td>Xen + 1s SBCFI</td>
<td>92841.8</td>
<td>0.1%</td>
</tr>
<tr>
<td>Xen + 0.5s SBCFI</td>
<td>92412.6</td>
<td>0.6%</td>
</tr>
</tbody>
</table>
Xen SPECweb2005 (1CPU)

Configuration

- Xen Only
- Xen + 3s SBCFI
- Xen + 1s SBCFI
- Xen + .5s SBCFI

Weighted Aggregate Byte Rate

- 93147.3
- 93128.3
- 93126.1
- 92892
- 92876.4
- 92841.8
- 92840.7
- 92827.5
- 92412.6
- 92509.8
- 92400
- 93000
- 93200
- 93400

Values:

- 91800
- 92000
- 92200
- 92400
- 92600
- 92800
Moving Beyond CFI

• Disadvantages of CFI
  – CFG assumed to be reasonable specification
  – Does not detect code lifecycle attacks
  – Misses non-control data attacks

• Disadvantages of SBCFI
  – Does not detect transient attacks

• Goal: find such attacks while considering attacker goals
  – In the meantime, we can monitor for attacks specifically a la semantic integrity
Non-Control Data Attacks
Related Work

- VM introspection (Garfinkel et al.)
  - Initially proposed monitoring from VMM
- Grizzard; Litty and Lie (Manitou)
  - Online kernel checking of CFI
  - Much higher overhead
- Static analysis, language techniques
  - Deputy, CCured, Cyclone can be used to achieve type safety in kernel
  - Prevent some, but not all forms of attacks
  - Not backwards compatible
Conclusion: SBCFI is Effective

- SBCFI systematically detects persistent kernel control-flow modifications
  - 24 of 25 analyzed rootkits make persistent control-flow modifications
- Our VMM-based implementation imposes less than 1% overhead when operating at 1s intervals
Along with colleagues and students, I am working to understand how to construct software that is available, reliable, and secure; i.e., software that

– never crashes
– adapts to changing circumstances and requirements
– properly protects data from unwanted tampering
– nevertheless provides useful and efficient services

Programming languages and analyses, utilizing theory and implementation, can be a powerful tool to this end
Backup Slides
Copilot

• **Advantages**
  – Stronger isolation
  – Potentially tamper resistant
  – Unmodified systems
  – Independent command and control

• **Disadvantages**
  – No access to registers-cache
  – Cost
  – Redirection attacks (Rutkowska)
Example: Control Flow Graph

- system call handler
  - ... sys_read()
  - sys_getdents()
  - sys_write()
  - ...

  - vfs_readdir()
    - ...
    - proc_root_readdir()
    - ...

Example: Tagged Control Flow Graph

- system call handler
  - sys_read()
  - sys_getdents()
  - sys_write()
  - vfs_readdir()
    - proc_root_readdir()
    - ...
    - ...
  - ...

Example: Attack Prevention

system call handler

... sys_read() sys_getdents() sys_write() ...

vfs_readdir()

... proc_root_readdir() ...

evil_proc_readdir()
CFI Kernel Implementation Challenges

- Threat model
  - Attacker has unrestricted access to kernel memory

- CFI requirements
  - Read-only code, non-executable data
    - Implemented using page table protections
    - Modifiable in kernel memory
  - Tags unique for all code
    - LKMs make rewriting all at once difficult
  - Precise CFG
    - Difficult for complex kernel control structure

- Performance
  - CFI incurs up to 30% overhead
Specification-based Kernel Monitoring

- Expert develops specification offline
- Specification compiled to runtime checks
- Checks performed from external monitor

AllTasks set: traverse linked-list starting at init_task

\[
\text{[ for}_\text{circular\_list}\ i\ \text{as}\ \text{ListHead.next starting init\_task.tasks.next ], true => container(i, Task, tasks.next) in AllTasks; }
\]

\[
\text{[ ], true => runqueue.curr in RunningTasks:}
\]

RunningTasks set: currently executing task

\[
\text{[ forall t in RunningTasks ]. t in AllTasks}
\]

\[
\text{: notify admin("Hidden task " + t.comm + " with PID " + t.pid + " detected at kernel virtual address " + t);}\]

Constraint: all tasks in RunningTasks should also be in AllTasks

Petroni, Fraser, Walters, Arbaugh
### Experimental Results – SPECweb2005 (2CPU)

<table>
<thead>
<tr>
<th></th>
<th>Byte Rate (bits/sec)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMware only</td>
<td>94445.5</td>
<td>0%</td>
</tr>
<tr>
<td>VMware + 3s</td>
<td>94155.4</td>
<td>0.3%</td>
</tr>
<tr>
<td>VMware + 1s</td>
<td>94457.7</td>
<td>0%</td>
</tr>
<tr>
<td>VMware Continuous</td>
<td>93459.1</td>
<td>1.0%</td>
</tr>
</tbody>
</table>
VMware SPECweb2005 (2CPU)
### Experimental Results – SPECCPU2006 (2CPU)

<table>
<thead>
<tr>
<th></th>
<th>Total time (s)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VMware only</td>
<td>10543</td>
<td>0%</td>
</tr>
<tr>
<td>VMware + 3s</td>
<td>10585</td>
<td>0.2%</td>
</tr>
<tr>
<td>VMware + 1s</td>
<td>10607</td>
<td>0.7%</td>
</tr>
<tr>
<td>VMware Continuous</td>
<td>10931</td>
<td>1.8%</td>
</tr>
</tbody>
</table>
VMware SPECCPU2006 (2CPU)

Time (s)

Configuration

VMware only
VMware + 3s SBCFI
VMware + 1s SBCFI
VMware Cont. SBCFI

VMware only
VMware + 3s SBCFI
VMware + 1s SBCFI
VMware Cont. SBCFI

10952
10932
10910

10562
10550
10541

10602
10585
10577

10611
10612
10601

10300
10400
10500
10600
10700
10800
10900
11000
## Summary of Linux Rootkits

<table>
<thead>
<tr>
<th>Attack name</th>
<th>HID</th>
<th>PE</th>
<th>REE</th>
<th>REC</th>
<th>NEU</th>
<th>Control-flow mods</th>
<th>Non-control data mods</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adore</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>P</td>
<td>P</td>
<td>D</td>
</tr>
<tr>
<td>Adore-ng</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>P</td>
<td>P</td>
<td>D</td>
</tr>
<tr>
<td>All-root</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>P</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>Anti Anti Sniffer</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>enyelkm</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>hp</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>kdb</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td>P</td>
<td>D</td>
</tr>
<tr>
<td>KIS</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td>P</td>
<td>D</td>
</tr>
<tr>
<td>Knark</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>P</td>
<td>P</td>
<td>D</td>
</tr>
<tr>
<td>Linspy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>P</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>Masty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modhide</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td>P</td>
<td>D</td>
</tr>
<tr>
<td>Mood-nt</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td>P</td>
<td>B</td>
</tr>
<tr>
<td>Override</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Phantasmagoria</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td>P</td>
<td>D</td>
</tr>
<tr>
<td>Phide</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rial</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td>P</td>
<td>D</td>
</tr>
<tr>
<td>Rkit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td>P</td>
<td>D</td>
</tr>
<tr>
<td>Taskigt</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shtroj2</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SucKIT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>P</td>
<td>T</td>
<td>D</td>
</tr>
<tr>
<td>SucKIT2</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>P</td>
<td>T</td>
<td>D</td>
</tr>
<tr>
<td>Superkit</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>P</td>
<td>T</td>
<td>D</td>
</tr>
<tr>
<td>Synapsys</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>THC Backdoor</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>

**HID:** hiding  **PE:** privilege escalation  **REE:** reentry  **REC:** reconnaissance  **NEU:** neutralization

**X:** provides functionality  **P:** persistent  **T:** transient  **B:** both P and T

**D:** detected in testing  **N:** did not detect  **- (dash):** unable to test