Detecting Kernel-level Rootkits using Data Structure Invariants

Vinod Ganapathy
Rutgers University
What are rootkits?

Rootkits = Stealthy malware

- Tools used by attackers to conceal their presence on a compromised system
- Typically installed after attacker has obtained root privileges
- Stealth is achieved by hiding accompanying user level malicious programs
Rootkit-based attack scenario

Rootkits hide malware from anti-malware tools

- Anti virus
- Key Logger
- Applications
- Backdoor
- Kernel
  - Rootkit-infected kernel
  - Kernel code
  - Kernel data

Sensitive information
Credit card: 4358654606
SSN: 543106789
Significance of the problem

• Alarming increase in number of rootkits
  – 600% increase in three-year period from 2004-2006 [MacAfee Avert Labs Report]
  – 200 new rootkits in the first quarter of 2008 alone [antirootkit.com]

• Rootkits are the vehicle of choice for botnet-based attacks
  – Stealth allows bot-masters to retain long-term control
Evolution of rootkits

**USER SPACE**
- System binaries
  - `/usr/bin/ls`
  - `/usr/bin/ps`
  - `/usr/bin/login`
- Shared Libraries

**KERNEL SPACE**
- System call table
- IDT
- Process Lists
- Kernel Code

**BELOW THE KERNEL**
- Hypervisor-based rootkits
  - Subvirt, Bluepill

**December 4, 2009**
Vinod Ganapathy - Rutgers University
Evolution of rootkits

**USER SPACE**
- System binaries
  - /usr/bin/ls
  - /usr/bin/ps
  - /usr/bin/login
- Shared Libraries

**KERNEL SPACE**
- System call table
- IDT
- Process Lists
- Kernel Code

**BELOW THE KERNEL**
- Hypervisor-based rootkits
  - Subvirt, Bluepill

December 4, 2009
Vinod Ganapathy - Rutgers University
Evolution of rootkits

USER SPACE

System binaries

- /usr/bin/ls
- /usr/bin/ps
- /usr/bin/login

Shared Libraries

KERNEL SPACE

System call table

IDT

Process Lists

Kernel Code

Hypervisor-based rootkits

Subvirt, Bluepill

BELOW THE KERNEL
Evolution of rootkits

Focus of this talk: Kernel-level rootkits

USER SPACE

 KERNEL SPACE

BETWEEN THE KERNEL

Hypervisor-based rootkits
Subvirt, Bluepill

System call table
IDT

/usr/bin/ls
/usr/bin/ps
/usr/bin/login

Shared Libraries

Process Lists
Kernel Code

December 4, 2009
Vinod Ganapathy - Rutgers University
Manipulating control data

• Change function pointers: Linux Adore rootkit

```c
int main()
{
    open(...)
    ...
    return(0)
}
```

KERNEL

USER SPACE

System call table

sys_open(...)
{
    ...
}

evil_open(...)
{
    ...
}
Manipulating non-control data (1)

- Change non-control data: Windows Fu rootkit

run-list

Process A
- run_list
- next_task

Hidden process
- run_list
- next_task

Process B
- run_list
- next_task

Process C
- run_list
- next_task

all-tasks

December 4, 2009
Vinod Ganapathy - Rutgers University
Manipulating non-control data (2)

- **Goal:** Attack the kernel’s pseudorandom number generator (PRNG) [Baliga et al., 2007]
The operating system kernel presents a vast attack surface for rootkits

- Function pointer and system call hooking
- Modifying process management linked lists
- Entropy pool contamination
- Disabling firewalls
- Resource wastage
- Disable pseudo-random number generation
- Intrinsic denial of service
- Defeating in memory signature scans
- Altering real time clock behavior
- Routing cache pollution
Detecting rootkits: **Main idea**

• **Observation**: Rootkits operate by maliciously modifying kernel data structures
  – Modify function pointers to hijack control flow
  – Modify process lists to hide malicious processes
  – Modify polynomials to corrupt output of PRNG

**Continuously monitor the integrity of kernel data structures**
Continuously monitor the integrity of kernel data structures

- **Challenge:** Data structure integrity monitor must be independent of the monitored system
- **Solution:** Use external hardware, such as a coprocessor, or a hypervisor to build the monitor
**Challenge:** Must monitor kernel code, control and non-control data structures

**Solution:** Periodically fetch and monitor all of kernel memory
**Challenge:** Specifying properties to monitor

**Solution:** Use *anomaly detection*

- **Inference phase:** Infer data structure *invariants*
- **Detection phase:** Enforce data structure invariants

---

### Data structure integrity monitor

<table>
<thead>
<tr>
<th>Kernel Code</th>
<th>Kernel Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Process lists</td>
</tr>
<tr>
<td></td>
<td>PRNG pools</td>
</tr>
<tr>
<td></td>
<td>System call table</td>
</tr>
</tbody>
</table>

Continuously monitor the integrity of kernel data structures
Rootkit detection using invariants (1)

```
int main()
{
    open(...)
    ... 
    return(0)
}
```

**Invariant**: Function pointer values in system call table should not change.
Rootkit detection using invariants (2)

Invariant: run-list $\subseteq$ all-tasks
Rootkit detection using invariants (3)

External Entropy Sources

Primary Entropy Pool (512 bytes)

Secondary Entropy Pool (128 bytes)

Urandom Entropy Pool (128 bytes)

Invariants

- poolinfo.tap1 is one of \{26, 103\}
- poolinfo.tap2 is one of \{20, 76\}
- poolinfo.tap3 is one of \{14, 51\}
- poolinfo.tap4 is one of \{7, 25\}
- poolinfo.tap5 == 1
A new rootkit detection tool

• Gibraltar
  * Identifies rootkits that modify control and non-control data
  * Automatically infers specifications of data structure integrity
  * Is physically isolated from the target machine
Architecture of Gibraltar

Target

DMA Transfer

Myrinet NIC

Memory Pages

Secure Backend Network

Monitor

Gibraltar

DMA Transfer
Components of Gibraltar

- Root Symbols
- Kernel Data Definitions
- Physical Memory Address
- Page Fetcher
- Data Structure Extractor
- Invariant Templates
- Invariant Generator
- Training Enforcer
- Enforcement
- run-list ⊆ all-tasks
- run-list ⊆ all-tasks?
Data structure extractor

• **Inputs:**
  – Memory pages from target machine
  – Root symbols: Entry-points into target’s kernel
  – Type definitions of target’s kernel

• **Output:** *snapshot* of target data structures

• **Main idea:** Traverse memory pages using root symbols and type definitions and extract data structures
Operation of data structure extractor

BFS Queue

Root symbols

Root 1
Root 2
Root 3
...
...
Root n

```
struct foo {
    struct bar * b1;
    struct list_head p;
};
```

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

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

Linked list of objects of type “struct foo”

CONTAINER(struct foo, p) p.next;
CONTAINER(struct foo, p) p.prev;

next_task
prev_task

next_task
prev_task

next_task
prev_task

next_task
prev_task

Vinod Ganapathy - Rutgers University

December 4, 2009
Invariant generator

• Executes during a controlled, training phase
• **Inputs**: Memory snapshots from a benign (uninfected) kernel
• **Output**: Likely data structure invariants

**Invariants serve as specifications of data structure integrity**
Invariant generator

• Used an off-the-shelf tool: Daikon [Ernst et al., 2000]

• Daikon observes execution of user-space programs and hypothesizes *likely invariants*

• We adapted Daikon’s front-end to reason about snapshots
  – Obtain snapshots at different times during training
  – Hypothesize likely invariants across snapshots
Experimental evaluation

Three main goals:

① How effective is Gibraltar at detecting rootkits? i.e., what is the false negative rate?

② What is the quality of automatically-generated invariants? i.e., what is the false positive rate?

③ What is the runtime performance overhead of Gibraltar?
Experimental setup

• Implemented on a Intel Xeon 2.80GHz, 1GB machine, running Linux-2.4.20
• Fetched memory pages using Myrinet PCI card
• Obtained invariants by training the system using several benign workloads
False negative rate

- Conducted experiments with 23 Linux rootkits
  - 14 rootkits from PacketStorm
  - 9 advanced rootkits, discussed in the literature
- All rootkits modify kernel control and non-control data
- Installed rootkits one at a time and tested effectiveness of Gibraltar at detecting the infection
<table>
<thead>
<tr>
<th>Sl. #</th>
<th>Rootkit Name</th>
<th>Data Structures Affected</th>
<th>Detected?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Adore-0.42</td>
<td>System call table (from PacketStorm)</td>
<td>✓</td>
</tr>
<tr>
<td>2.</td>
<td>All-root</td>
<td>System call table (from PacketStorm)</td>
<td>✓</td>
</tr>
<tr>
<td>3.</td>
<td>Kbd</td>
<td>System call table (from PacketStorm)</td>
<td>✓</td>
</tr>
<tr>
<td>4.</td>
<td>Kis-0.9</td>
<td>System call table (from PacketStorm)</td>
<td>✓</td>
</tr>
<tr>
<td>5.</td>
<td>Linspy2</td>
<td>System call table (from PacketStorm)</td>
<td>✓</td>
</tr>
<tr>
<td>6.</td>
<td>Modhide</td>
<td>System call table (from PacketStorm)</td>
<td>✓</td>
</tr>
<tr>
<td>7.</td>
<td>Phide</td>
<td>System call table (from PacketStorm)</td>
<td>✓</td>
</tr>
<tr>
<td>8.</td>
<td>Rial</td>
<td>System call table (from PacketStorm)</td>
<td>✓</td>
</tr>
<tr>
<td>9.</td>
<td>Rkit-1.01</td>
<td>System call table (from PacketStorm)</td>
<td>✓</td>
</tr>
<tr>
<td>10.</td>
<td>Shtroj2</td>
<td>System call table (from PacketStorm)</td>
<td>✓</td>
</tr>
<tr>
<td>11.</td>
<td>Synapsys-0.4</td>
<td>System call table (from PacketStorm)</td>
<td>✓</td>
</tr>
<tr>
<td>12.</td>
<td>THC Backdoor</td>
<td>System call table (from PacketStorm)</td>
<td>✓</td>
</tr>
<tr>
<td>13.</td>
<td>Adore-ng</td>
<td>VFS hooks/UDP recvmsg (from PacketStorm)</td>
<td>✓</td>
</tr>
<tr>
<td>14.</td>
<td>Knark-2.4.3</td>
<td>System call table, proc hooks (from PacketStorm)</td>
<td>✓</td>
</tr>
<tr>
<td>15.</td>
<td>Disable Firewall</td>
<td>Netfilter hooks (Baliga et al., 2007)</td>
<td>✓</td>
</tr>
<tr>
<td>16.</td>
<td>Disable PRNG</td>
<td>VFS hooks (Baliga et al., 2007)</td>
<td>✓</td>
</tr>
<tr>
<td>17.</td>
<td>Altering RTC</td>
<td>VFS hooks (Baliga et al., 2007)</td>
<td>✓</td>
</tr>
<tr>
<td>18.</td>
<td>Defeat signature scans</td>
<td>VFS hooks (Baliga et al., 2007)</td>
<td>✓</td>
</tr>
<tr>
<td>19.</td>
<td>Entropy pool</td>
<td>struct poolinfo (Baliga et al., 2007)</td>
<td>✓</td>
</tr>
<tr>
<td>20.</td>
<td>Hidden process</td>
<td>Process lists (Petroni et al., 2006)</td>
<td>✓</td>
</tr>
<tr>
<td>21.</td>
<td>Linux Binfmt</td>
<td>Shellcode.com</td>
<td>✓</td>
</tr>
<tr>
<td>22.</td>
<td>Resource waste</td>
<td>struct zone_struct (Baliga et al., 2007)</td>
<td>✓</td>
</tr>
<tr>
<td>23.</td>
<td>Intrinsic DOS</td>
<td>int max_threads (Baliga et al., 2007)</td>
<td>✓</td>
</tr>
</tbody>
</table>
False positive evaluation

• Ran a benign workload for 42 minutes
  – Copying Linux kernel source code
  – Editing a text document
  – Compiling the Linux kernel
  – Downloading eight videos from Internet
  – Perform file system operations using the IOZone benchmark

• Measured how many invariants were violated
False positive evaluation

<table>
<thead>
<tr>
<th>Invariant Type</th>
<th># Invariants</th>
<th>False positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistent</td>
<td>236, 444</td>
<td>0.035%</td>
</tr>
<tr>
<td>Transient</td>
<td>482, 496</td>
<td>0.65%</td>
</tr>
</tbody>
</table>

- **Persistent invariants**: Those that persist across machine reboots.\[\text{run\_list} \subseteq \text{all\_tasks}\]
- **Transient invariants**: Those that hold only until the next reboot.

\begin{verbatim}
init_fs->root->d_sb->s_dirty.next->i_dentry.next->
d_child.prev->d_inode->i_fop.read == 0xeff9bf60
\end{verbatim}
3 Performance evaluation

- Training time: total of 56 minutes
  - 25 mins to collect snapshots (total 15 snapshots)
  - 31 minutes to infer invariants
- Detection time
  - Ranges from 15 seconds up to 132 seconds
- PCI Overhead
  - 0.49%, measured using the stream benchmark
Future work

• Reducing the number of false positives
  – Automated filtering techniques
  – Longer training time, better training workload

• Usefulness of invariants generated
  – Other invariant templates
  – Feasibility versus complexity

• Portability of invariants across different systems
Thank you

Detecting Kernel-level Rootkits using Data Structure Invariants

“Automatic Inference and Enforcement of Kernel Data Structure Invariants”
Arati Baliga, Vinod Ganapathy and Liviu Iftode
Published in Proc. 24th ACSAC, December 2008.