Online surveillance of critical computer systems through advanced host-based detection

Harmonized Anomaly Detection Techniques Thread

Shariyar Murtaza, Shayan Eskandari, Afroza Sultana, Wahab Hamou-Lhadj

Software Behaviour Analysis Research Lab
Host-Based Anomaly Detection Working Group
Concordia, Canada

Dec 07, 2012
Contents

• Progress on kernel rootkit detection
• Progress on system call based anomaly detection for applications
What is a Rootkit?

• Rootkit is a malware having several functionalities:
  – Stealth processing
  – Covert communication from system administrators.
  – Keystroke logging
  – Packet sniffing
  – Backdoor shell access
  – Remote attacking on networks
Severity of Rootkits

• Data-theft accounts for 80% of all cyber crimes
• Some recent examples of rootkits activities are:
  – Bank frauds
  – Disabling antivirus software
  – Making a system a bot
  – Stealing information

Source: [DARPA Technical Report, 2007], [McAfee Whitepaper, 2009]
Types of Kernel Rootkits

- Kernel Rootkits
  - Kernel Object Hooking (KOH) Rootkits
  - Dynamic Kernel Object Manipulation (DKOM) Rootkits
Main Types of Kernel Rootkits

**KOH**
- Modify control data structures
- A handler function registers its address with kernel data structures; e.g., syscall hooking

**DKOM**
- Modify non-control data structures
- Manipulate internal record-keeping data within main memory; e.g., the list of running processes
KOH Rootkit

Sys_call_table structure contains a set of pointers to functions implementing various system calls. A system call can be overridden by changing pointers.

Source: [Pelaez, 2004]
DKOM Rootkit

Struct module is a linked list of module objects. An LKM can be hidden by removing its entry from this struct.

Source: [Pelaez, 2004]
Common Methods Used by Rootkits

Loadable Kernel Module (LKM) can replace underlying system calls with their own version; e.g., Knark, Adore-ng

Directly patch the kernel’s virtual memory (/dev/kmem) or physical memory (/dev/mem); e.g., SuckIT, Super User Control Kit

Directly patch the kernel’s image on hard disk (/boot/vmlinux); e.g., Kpatch

Using virtual machine to run a fraudulent system; e.g., BluePill

Using libc crashes to execute kernel instructions through stack for malicious purpose; e.g., return-to-libc rootkits
Rootkit Detection and Prevention Techniques

- Host based techniques
- Virtualization based techniques
- External observer based techniques
# Host Based Techniques

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kruegel et al. [2004]</td>
<td>Detect malicious LKMs using static analysis of LKM binaries</td>
</tr>
<tr>
<td>Kroah-Hartman [2004]</td>
<td>Load only RSA encrypted signed modules into memory</td>
</tr>
<tr>
<td>Secure boot [Parno et al., 2010; Jaeger et al., 2011]</td>
<td>Load a component if the hash is equal to a known-good value</td>
</tr>
<tr>
<td>Jestin et al. [2011a]</td>
<td>Cluster memory addresses to detect high memory addresses related to malicious system calls</td>
</tr>
<tr>
<td>AppArmor [Bauer, 2006] and SELinux [Smalley et al., 2002]</td>
<td>Limit access to the kernel by using policies</td>
</tr>
<tr>
<td>Strider Ghostbuster [Beck et al., 2005]</td>
<td>Identify hidden files and processes using normal views</td>
</tr>
</tbody>
</table>
Host Based Techniques: Tools Scanning Known Places

- Kstat—/dev/kmem vs. system.map
- Kern check—system.map vs. system call table
- Chkrootkit—logs and configs
- Rootkithunter—files, ports, processes
- Rkscan—Adore, Knark
- Knarkfinder—hidden processes
- Tripwire, Samhain and AIDE—checksum based integrity
- Sleuth Kit—File system forensics tool
## Virtualization Based Techniques

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Garfinkel &amp; Rosenblum, 2003]</td>
<td>Enforce HIDS policies from VMM, such as signature scan of memory, comparing commands, text comparison, etc.</td>
</tr>
<tr>
<td>[Petroni et al. 2007]</td>
<td>Use cryptographic hashes of code and the graph of function pointers to detect control flow (KOH) anomalies</td>
</tr>
<tr>
<td>[Wang et al. 2009]</td>
<td>Make a copy of hooks (pointers) to a write protected location, verify accesses and prevent KOH rootkits</td>
</tr>
<tr>
<td>[Seshadri et al., 2007] and [Riley et al., 2008]</td>
<td>Prevent kernel code from unauthorized modification and execution—targets KOH rootkits.</td>
</tr>
<tr>
<td>[Baliga et al. 2008]</td>
<td>Prevent KOH rootkits by using the policies based on process and file relationships</td>
</tr>
<tr>
<td>[Rhee et al. 2009]</td>
<td>Use policies for key data structure (e.g., modification through known functions) to detect DKOM rootkits</td>
</tr>
<tr>
<td>[Jiang et al. 2007]</td>
<td>A technique to run anti-malware programs from outside of an OS on a VM; e.g., antivirus</td>
</tr>
</tbody>
</table>
External Observer Based Techniques

- Copilot [Petroni et al., 2004], a PCI-card monitor, compares kernel text, LKM text and function pointers to detect KOH rootkits

- Gibraltar [Baliga et al., 2011] detect KOH and DKOM rootkits by using data structure invariants

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Invariant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect hidden process</td>
<td>run-list $\subset$ all-tasks</td>
<td>run_list is a process list used by scheduler and all_task by others</td>
</tr>
<tr>
<td>Don’t let firewall disable</td>
<td>nf_hooks[2][1].next.hook == 0xc03295b0</td>
<td>To avoid redirection actual address is identified</td>
</tr>
</tbody>
</table>
## Classification of Anti-Rootkit Techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Type</th>
<th>KOH</th>
<th>DKOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static analysis of binary images of LKM [Kreugel et al., 2004]</td>
<td>HB</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Rootkit hunter [RootkitHunter]</td>
<td>HB</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>State based control flow integrity monitor [Petroni et al., 2007]</td>
<td>VM</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>HookSafe [Wang et al., 2009]</td>
<td>VM</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>KernelGuard [Rhee et al. 2009]</td>
<td>VM</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Gibraltar [Baliga et al., 2011]</td>
<td>EM</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>NICKLE [Riley et al, 2008]</td>
<td>VM</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

HB= Host Based; VM= Virtual Machines; and EM= External Monitor
Lessons Learned

1. Control flow integrity results in few or no false positives.

2. DKOM rootkits can be detected by monitoring data structures and legitimate modifier functions.

3. Return oriented rootkits can be detected if the instructions they push on stacks change the normal flow of execution.
Challenging Solution

Monitor the call graph and data structure modifications

Monitoring the entire control graph of the kernel will cause a very high overhead
## Statistics for Kernel 2.6.32.44

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total functions in source code (approx.)</td>
<td>232312</td>
</tr>
<tr>
<td>Total functions excluding documentation, scripts and drivers (approx.)</td>
<td>107094</td>
</tr>
<tr>
<td>Graph size (total edges)</td>
<td>394212</td>
</tr>
<tr>
<td>Callers</td>
<td>81919</td>
</tr>
<tr>
<td>Callees</td>
<td>75426</td>
</tr>
</tbody>
</table>
A Routine Call Graph of the “fs” Subsystem
Proposed Solution: Traffic-Based Approach

• Based on Hamou-Lhadj’s empirical studies on identifying important components in large systems:
  – High traffic vs. Low traffic components
  – Only a small number of functions make the largest number of calls
  – Priority monitoring should focus on only these functions
  – Secondary monitoring: Other functions can be added as needed
Identifying Important Functions

- Look for functions that generate the largest number of calls – Hamou-Lhadj’s work on identifying utilities in large systems
- Network-based techniques: Betweenness centrality analysis
- Look for the functions of most targeted components
- Knowledge oriented techniques: Study sensitive paths where attacks have the highest information gain
Reducing Overhead Using Calling Relationship

Approximately half of the functions make majority of the calls
Reducing Overhead Using Betweenness Centrality

Graph reduced to just the nodes with high Betweenness Centrality
Reducing Overhead Using Key Components

Distribution of 2009-2011 Linux vulnerabilities across components.

- Component sizes in the Linux kernel 2.6.33.20:
  - drivers: 50%
  - fs: 7%
  - arch: 23%
  - others: 16%
  - net: 4%

- Attack objectives/component:
  - drivers: 20%
  - fs: 25%
  - arch: 25%
  - Others: 23%
  - net: 7%
Vertex Framework

Call Graph and Data Structures:
- `GlobalVar1`
- `GlobalVar2`
- `GlobalVar3`

**Controller**
Embeds strategies for selecting important components

**Correlator**
Contains logic for correlating run-time behaviour with baseline

**Monitor**
- Monitors
- Problem detected

**Test Cases**
- Traces

**Executable**

**In-op**

**Training**

**Source Code**
```c
foo(int a, int b)
{
    if (a>b)
    {
        printf("Hi");
    } 
    foo2(a);
}

foo2(int a)
```
Preliminary Results

• Generated static and dynamic call graph for kernel
• Tested on a home grown rootkit and KBBeast rootkit
  – Syscall hooking, key logging, process hiding, directory hiding, port hiding and backdoor shell access
• Both rootkits were detected immediately—unknown functions hooked to pointers
• Some false positives due to missing edges due to function pointers
Ongoing Tasks

- Studying the strategies for reducing graph size
- Developing correlation algorithms
- Developing a prototype tool
- Conducting experiments with real rootkits
Input For the Tracing Team

- Function calls and data structure tracing
- Notifications for global data structure modifications
- Develop a kernel stack monitor
- Develop a security mechanism against tampering of the monitoring system
- Easy to use interface to turn on and off probes
System Call Based Models For Applications
System Call Sequence Modeling

- Models an application’s normal behavior from system calls sequences:
  - Sliding Windows [Forrest 1997, Warrender 1999]
  - Rule Based [Tandon 2003, Petrussenko 2010]
  - Neural Networks [Ghosh 1999]
  - Hidden Markov Models (HMM) [Hoang 2003, Hu 2009]
  - Finite State Automata (FSA) [Wagner 2001, Sekar 2001]
  - Variable length N-gram [Wespi 1999, Jiang 2002]
  - Statistical Techniques [Ye 2001, Burgess 2002]
  - Call Stack Techniques [Feng 2003]
  - Bag of System Call Technique [Kang 2005]
Attack sequences are very similar to the normal system call sequences—false positives.

Imitate legitimate system call sequences that execute malicious code—*which calls are they?*

Limitations of System Call Sequence Models

Replace the system call arguments and return values. For example, open a malicious file using the system call arguments of the “open()” system calls—arguments and calls.

Equivalent malicious system calls. For example, after an open(), replace a legitimate read() with a mmap() that reads memory—false positives.

Insert some “no-op” system calls (e.g., read() with 0 byte parameter) between the malicious system call to look like the legitimate sequence—*which calls are they?*
Research Questions

How much code coverage provides a complete learning model to remove/reduce false positives?

How can system call and argument models be efficiently combined?

Are there system calls more important than others?
How much code coverage provides a complete learning model to remove/reduce false positives?
## Firefox Dataset

<table>
<thead>
<tr>
<th>Test Framework</th>
<th>Passing Test Cases</th>
<th>Passing Test Files</th>
<th>Firefox’s Code Coverage</th>
<th>Avg. System Calls Per Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Statements (%)</td>
<td>Functions (%)</td>
</tr>
<tr>
<td>XPC Shell</td>
<td>600</td>
<td>600</td>
<td>39.8</td>
<td>39.6</td>
</tr>
<tr>
<td>JS Engine</td>
<td>2686</td>
<td>2686</td>
<td>41.3</td>
<td>40.2</td>
</tr>
<tr>
<td>Mozmill</td>
<td>34</td>
<td>34</td>
<td>47.1</td>
<td>46.1</td>
</tr>
<tr>
<td>Mochitest-a11y</td>
<td>1369</td>
<td>41</td>
<td>49.7</td>
<td>48.7</td>
</tr>
<tr>
<td>Mochitest-chrome</td>
<td>1160</td>
<td>84</td>
<td>50.9</td>
<td>49.6</td>
</tr>
<tr>
<td>Mochitest-browser-chrome</td>
<td>2913</td>
<td>146</td>
<td>52.1</td>
<td>50.6</td>
</tr>
</tbody>
</table>
# System Call Sequence Model

<table>
<thead>
<tr>
<th>Framework</th>
<th>Total number of sequences in each framework</th>
<th>Total number of unique sequences in each framework</th>
<th>Percentage of unique sequences in each framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPC Shell</td>
<td>11,084,489</td>
<td>29,871</td>
<td>0.27%</td>
</tr>
<tr>
<td>JS Engine</td>
<td>6,793,621</td>
<td>3,771</td>
<td>0.06%</td>
</tr>
<tr>
<td>Mozmill</td>
<td>551,516</td>
<td>1,106</td>
<td>0.20%</td>
</tr>
<tr>
<td>Mochitest-a11y</td>
<td>31,609,380</td>
<td>10,440</td>
<td>0.03%</td>
</tr>
<tr>
<td>Mochitest-chrome</td>
<td>57,487,806</td>
<td>11,069</td>
<td>0.02%</td>
</tr>
<tr>
<td>Mochitest-browser-chrome</td>
<td>124,136,624</td>
<td>11,595</td>
<td>0.01%</td>
</tr>
</tbody>
</table>

Window size = 6.
Impact of Coverage on the Model’s completeness

- 67,852 unique sequences identified from 231,663,436 sequences
- Model size is 1,331 KB
How can system call and argument models be efficiently combined?
Combine System Call Sequence and System Call Argument Model

- System Call Sequence Based Model
- System Call Argument Based Model
- Testing Trace Files
- Healthy Trace Files
- System call sequences
- System call arguments
- Results
- Harmonization
- Final Result
Sys_open() Argument Model Construction

sys_open(filename, flag, mode)

Extract file directory and names

Add

Merge

Sys_open() sequence Model

Sys_open() Argument Model
Are there system calls more important than others?
## Frequent System Calls

<table>
<thead>
<tr>
<th>System Call Name</th>
<th>Total Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys_write</td>
<td>181,115,332</td>
<td>78%</td>
</tr>
<tr>
<td>sys_read</td>
<td>9,125,195</td>
<td>4%</td>
</tr>
<tr>
<td>sys_llseek</td>
<td>5,669,487</td>
<td>2%</td>
</tr>
<tr>
<td>sys_fcntl64</td>
<td>4,358,035</td>
<td>2%</td>
</tr>
<tr>
<td>sys_futex</td>
<td>4,209,934</td>
<td>2%</td>
</tr>
<tr>
<td>sys_fstat64</td>
<td>4,209,034</td>
<td>2%</td>
</tr>
<tr>
<td>sys_stat64</td>
<td>3,603,088</td>
<td>2%</td>
</tr>
<tr>
<td>sys_open</td>
<td>2,932,515</td>
<td>1%</td>
</tr>
<tr>
<td>sys_gettimeofday</td>
<td>2,897,824</td>
<td>1%</td>
</tr>
<tr>
<td>sys_close</td>
<td>2,809,496</td>
<td>1%</td>
</tr>
<tr>
<td>sys_madvise</td>
<td>2,742,715</td>
<td>1%</td>
</tr>
<tr>
<td>sys_mmap_pgoff</td>
<td>2,612,093</td>
<td>1%</td>
</tr>
<tr>
<td>sys_munmap</td>
<td>2,068,300</td>
<td>1%</td>
</tr>
</tbody>
</table>
Thank you & your questions
References

References (2)

References (3)


References (4)

References (5)

References (6)

References (7)

References (8)

References

References

References

References

Resources for Rootkits

- http://www.phrack.org
- http://www.antiserver.it/Backdoor-Rootkit/
- http://www.l0t3k.org/tools/Rootkit/
- http://packetstormsecurity.org/UNIX/penetration/rootkits/
- http://www.securityfocus.com/
- http://www.antiserver.it/Backdoor-Rootkit/
- http://www.rootkit.com (mostly Windows based)

Disclaimer: Presenter doesn’t take responsibility of any malicious activity that would happen on your system when you play with these resources 😊.

Source: [Pelaez, 2004]