HookFinder: Identifying and Understanding Malware Hooking Behaviors

Heng Yin
Carnegie Mellon Univ
Coll Of William and Mary

Zhenkai Liang
Carnegie Mellon Univ

Dawn Song
UC Berkeley
Carnegie Mellon Univ
What is a hook?

- Malware registers its own function (i.e. hook) into the target location (i.e. hook site).
- Later, data in the hook site is loaded into EIP, and the execution is redirected into malware’s own function.

Sony Rootkit: an example of SSDT hooking
Why are hooks important?

• Malware needs to place hooks to achieve its malicious intents:
  – Rootkits want to intercept and tamper with critical system states
  – Network sniffers eavesdrop on incoming network traffic
  – Stealth backdoors intercept network stack to establish stealthy communication channels
  – Spyware, keyloggers and password thieves …
Current techniques are insufficient

• Some tools detect hooks by checking known memory regions for suspicious entries
  – E.g., VICE [Butler:2004], IceSword, System Virginity Verifier [Rukowska:2005]
  – Code sections, IAT/EAT, SSDT, IRP tables
  – They become futile when malware uses new hooking mechanisms

• Malware writers strive for new hooking mechanisms
  – E.g., Two kernel backdoors (Deepdoor and Uay) overwrite only a small portion in NDIS (i.e., Network Driver Interface Specification) data block
  – All existing tools cannot detect this kind of hooks
Our Approach

• We propose a system to automatically detect and analyze (previously unknown) hooks
  – Given an unknown malicious binary
  – Identify if it installs any hooks (with no prior knowledge)
  – Understand hooking mechanism
    » Provide detailed information about how it installs these hooks

• When a sample employs a novel hooking mechanism, we can identify and understand it instantly
  – Update detection/prevention policy, to detect/prevent the similar hooks in the future
Outline

• Motivation
• Approach Overview
• HookFinder Design and Implementation
• Experimental Evaluation
• Summary
Intuition

• A hook is one of the **impacts** (*i.e.*, state changes) to the system made by malware
• This impact redirects the execution into the malicious code.

We can detect and analyze hooks by marking and tracking impacts.
Our Techniques

• Hook Detection: **Fine-grained Impact Analysis**
  – Mark initial impacts
  – Track impacts propagation (and generate *Impact Trace*)
  – Detect affected control flow

• Hook Analysis: **Semantics-aware Impact Dependency Analysis**
  – Backward data dependency analysis on Impact Trace
  – Combine OS-level semantics information
  – Generate a dependency graph: *Hook Graph*
Outline

• Motivation
• Approach Overview
• HookFinder Design and Implementation
• Evaluation
• Summary
HookFinder – System Overview

We build HookFinder on top of TEMU, which is a dynamic binary analysis component in the BitBlaze Project.
Semantics Extractor

- Whole-system Emulator only provides a hardware-level view
  - E.g., states of memory, registers, and I/O devices
- We need an OS-level view
  - Which process/module/thread is running currently?
  - What is the function name, if malware calls an external function
  - What is the symbol name, if malware reads a symbol
- TEMU provides this functionality
  - See [Yin et al:2007] and this paper for more details
Impact Analysis Engine

• Mark Initial Impacts (memory and register writes)
  – In malware’s module
  – In external function calls
  – In dynamically generated code

  Challenge: identify dynamically generated code
  Observation: dynamically generated code is part of impacts made by malware
  Solution: check if the code region is marked

• Track impact propagation
  – Track data dependency (like in dynamic taint analysis)
    » Check propagation through disks
  – Check immediate operands
    » Because malware can manipulate immediate operands
Hook Detector

- Detect when a hook is used
  - Condition 1: Program counter (i.e, EIP in x86) is marked
  - Condition 2: The execution jumps into the malicious code
How HookFinder Detects Hooks in Sony Rootkit

In Malicious Code

aries.sys+ee6: mov ZwOpenKey, %edi
aries.sys+f56: mov 1(%edi), %eax
aries.sys+f59: mov KeServiceDescriptorTable, %ecx
aries.sys+f5f: mov (%ecx), %ecx
aries.sys+f61: movl aries.sys+66e, (%ecx, %eax, 4)

ntoskrnl.exe+8051: movl (%edi, %eax, 4), %ebx
ntoskrnl.exe+8069: call *%ebx

A hook is detected:
1) EIP is marked
2) The execution is redirected into aries.sys

Syntax: op src, dst
Hook Analyzer

• Generate hardware-level hook graph
  – Perform backward dependency analysis on the impact trace

• Transform into OS-level graph
  – Combine OS-level semantic information

• Simplify hook graph
  – If two adjacent nodes belong to the same external function call, merge them into one node
  – If two adjacent nodes are direct copy instructions (e.g., mov, push, pop), merge them into one node
Hook Graph for Sony Rootkit

aries.sys+ee6:
  mov ZwOpenKey, %edi

aries.sys+f56:
  mov 1(%edi), %eax

aries.sys+f59:
  mov KeServiceDescriptorTable, %ecx

aries.sys+f5f:
  mov (%ecx), %ecx

Impacted Address

aries.sys+f61:
  movl aries.sys+66e, (%ecx, %eax, 4)

ntoskrnl.exe+8051:
  movl (%edi, %eax, 4), %ebx

ntoskrnl.exe+8069: call *%ebx

This hook is activated

This hook is installed
Outline

• Motivation
• Approach Overview
• HookFinder Design and Implementation
• Evaluation
• Summary
## Summarized Results

<table>
<thead>
<tr>
<th>Sample</th>
<th>Category</th>
<th>Runtime</th>
<th>Impact Trace</th>
<th>Hooks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Online</td>
<td>Offline</td>
<td>Total</td>
</tr>
<tr>
<td>Troj/Keylogg-LF</td>
<td>Keylogger</td>
<td>6min</td>
<td>9min</td>
<td>3.7G</td>
</tr>
<tr>
<td>Troj/Thief</td>
<td>Password</td>
<td>4min</td>
<td>&lt;1min</td>
<td>143M</td>
</tr>
<tr>
<td></td>
<td>Thief</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AFXRootkit</td>
<td>Rootkit</td>
<td>6min</td>
<td>33min</td>
<td>14G</td>
</tr>
<tr>
<td>CFSD</td>
<td>Rootkit</td>
<td>4min</td>
<td>2min</td>
<td>2.8G</td>
</tr>
<tr>
<td>Sony Rootkit</td>
<td>Rootkit</td>
<td>4min</td>
<td>&lt;1min</td>
<td>25M</td>
</tr>
<tr>
<td>Vanquish</td>
<td>Rootkit</td>
<td>6min</td>
<td>12min</td>
<td>4.4G</td>
</tr>
<tr>
<td>Hacker Defender</td>
<td>Rootkit</td>
<td>5min</td>
<td>27min</td>
<td>7.4G</td>
</tr>
<tr>
<td>Uay Backdoor</td>
<td>Backdoor</td>
<td>4min</td>
<td>&lt;1min</td>
<td>117M</td>
</tr>
</tbody>
</table>

Legitimate hooks: `PsCreateSystemThread, CreateThread, CreateRemoteThread, StartServiceDispatcher`
Detailed Analysis of Uay

Static Point: Protocol Handler (h) returned from NdisRegisterProtocol

Uay walks through a list of registered protocols and places the hook into one entry (with offset 0x40)

Hook Site = MEM[MEM[h+10]+10]+40

NDIS.sys+115b: mov %eax, (%ecx)
Call: NdisAllocateMemoryWithTag

NDIS.sys+22faa: call *0x40(%eax)
Related Work

• Hook Detection
  – VICE [Butler:2004], IceSword, System Virginity Verifier [Rukowska:2005]

• Dynamic Taint Analysis
  – Data lifetime analysis [Chow et al:2004]
  – Extract protocol format [Caballero et al:2007]
Summary

• We proposed fine-grained impact analysis
  – Characterize malware’s impacts on the system environment
  – Observe if one of the impacts is used to redirect the execution into the malicious code
  – Capture intrinsic characteristics of hooking behavior, and thus it can identify novel hooks

• We devised semantics-aware impact dependency analysis
  – Extract hooking mechanism in form of hook graphs

• We developed HookFinder

• We analyzed 8 representative malware samples
  – HookFinder is able to identify and analyze new hooks in Uay
Thanks!

For more information and related projects, please visit our BitBlaze website at
http://bitblaze.cs.berkeley.edu
Discussion 1

- Exploit control dependency
  
  `switch(a) {
    case 1: b=1; break; case 2: b=3; break; …}
  `  
  - Not feasible, since we track all initial impacts
Discussion 2

- Not exhibit hooking behavior when tested
  - Bypass redpill test by feeding in fake inputs
  - Slow down the frequency of PIT to disguise the performance slowdown
Discussion 3

• “Return-into-libc” attacks: register an address of a system function
  – Hard to find a candidate function
  – Hard to prepare compatible call stack
  – Will consider it in the future work
Key Factors in Hooking Mechanism

- **Hook Type**
  - Data Hook: interpreted as data (e.g., jump target)
  - Code Hook: interpreted as code (e.g., jump instruction)

- **Implanting methods**
  - Direct write
    - What is the static point?
      - Global symbol, or result of a function call
    - How to infer the hook site?
  - Call an external function
    - Which function is called?
      - E.g., SetWindowsHookEx, memcpy, WriteProcessMemory
    - What is the argument list?