Effective *and* Efficient Malware Detection at the End Host

Clemens KOLBITSCHE, Paolo MILANI COMPARETTI, Engin KIRDA, Christopher KRUEGEL, Xiaoyong ZHOU, XiaoFeng WANG

ck@iseclab.org

Secure Systems Lab [TU Vienna, Institute Eurecom Sophia Antipolis, UC Santa Barbara] Indiana University at Bloomington
Motivation

Why do we propose yet another malware detection scheme (yamds)?

• Binary signature based detection inherently ineffective
  – We all know the problems...
  – Arms-race, pretty much a lost battle

• Network based approaches evadable
  – Systems scan for communication artifacts
  – Encryption / blending thwart detection
Motivation

*Why do we propose yet another malware detection scheme (yamds)?*

- Don't rely on artifacts of malware instances
  - Instead focus on generic patterns

- Proposed solution:
  - Detection based on malware's behavior
  - Behavior is hard to obfuscate
  - Behavior is hard to randomize
  - Behavior is often stable across various malware version
Motivation

• Behavior-based detection received some attention over last couple of years

• Despite promising detection results, binary signatures remain the method of choice
Motivation

- Behavior-based detection received some attention over last couple of years

- Despite promising detection results, binary signatures remain the method of choice

- Behavior graphs are not new, but we bring together the effectiveness of taint based systems and provide it on an efficient end host

- See WAV -2:45

<table>
<thead>
<tr>
<th>+ efficiency</th>
<th>- emulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary signatures</td>
<td>behavior</td>
</tr>
<tr>
<td>- evasion</td>
<td>+ effectiveness</td>
</tr>
</tbody>
</table>
Motivation

• Behavior-based detection received some attention over last couple of years

• Despite promising detection results, binary signatures remain the method of choice

+ efficiency

- emulation

binary signatures

- evasion

behavior

+ effectiveness
Outline

• Motivation

• Detecting Behavior
  – Motivating example (Agent)

• Matching Behavior Graphs

• Extracting Behavior Graphs

• Evaluation
Detecting Behavior
Detecting Behavior

• Characteristic malware behavior
  – Manifest on system (i.e., survive reboot)
    • (Over-) write system executables, dlls, files
    • Create registry entries (autorun)
    • Register as Windows (startup) service

  – Conceal from being detected
    • Restart under some *stealthy* name (e.g., svchost.exe)
    • Inject into legitimate processes

  – Replicate
    • Send eMails (*check out this picture I found: pic.jpg.exe*)
    • Copy to Samba shares, USB drives, etc.
    • Scan and exploit services on LAN or WAN
Detecting Behavior
System Overview

• Detection based on *execution characteristics*
  – Execute malware in full system emulator (Anubis)
  – Monitor interaction with the operating system
  – Perform detailed (taint-) analysis
  – Generate *detection graphs*
    • Describe sequence of *required* system calls leading to *security relevant* system activity
    • Include dependencies to related, previous calls (using taint dependencies)

• Detect described behavior on end host
  – Log system call activity of unknown executable
  – Match against behavior graph
Detecting Behavior
Developer Perspective

• Example: Agent (trojan horse)

• As part of its system manifestation, it
  – Reads content from binary image
  – Decrypts binary content
    • Proprietary decryption routine
    • Simple, XOR based algorithm
  – Stores binary in system file (C:\Wind...\drivers\ip6fw.sys)
  – Later, restarts IPv6 firewall
    • Turns itself into a system service
Detecting Behavior
Taint-Trace Perspective

GetModuleFileNameA

Name

Mode: Open

NtCreateFile

FileHandle

Mode: Create

C:\Win...\ip6fw.sys

NtCreateFile

NtCreateSection

SectionHandle

NtMapViewOfSection

(read & decrypt buffer)

NtWriteFile
Detecting Behavior
System Perspective
Detecting Behavior
System Perspective

GetModuleFileNameA

Mode: Open
NtCreateFile

FileHandle

Mode: Create
NtCreateFile

SectionHandle

C:\Win...
\ip6fw.sys

FileHandle

NtCreateSection

NtMapViewOfSection

(read & decrypt buffer)

NtWriteFile
Detecting Behavior
System Perspective

GetModuleFileNameA

Mode: Open
NtCreateFile

Mode: Create
NtCreateFile

C:\Win...\ip6fw.sys

FileHandle

FileHandle

NtCreateFile

NtCreateFile

NtCreateSection

NtMapViewOfSection

NtWriteFile

(Read & decrypt buffer)
Detecting Behavior
System Perspective

Mode: Open
NtCreateFile
  FileHandle
NtCreateSection
  SectionHandle
NtMapViewOfSection

Mode: Create
C:\Win...\ip6fw.sys
NtCreateFile
  FileHandle
NtWriteFile
Detecting Behavior
System Perspective

Mode: Open
NtCreateFile
  FileHandle
  NtCreateSection
    SectionHandle
    NtMapViewOfSection

Mode: Create
NtCreateFile
  C:\Win...\ip6fw.sys
    FileHandle
NtWriteFile
Detecting Behavior

- Detection based on *execution characteristics*
  - Works well as long as we can see *all types of* dependencies between system calls
  - *Handle* dependencies
    - Insufficient for detection
    - Behavior graphs break into trivial subgraphs
  - *Data* dependencies
    - *Convenient* for behavior graph generation
    - *Necessary* for behavior detection
Matching Behavior Graphs
Matching Behavior Graphs

• Maintaining dependencies using taint propagation
  – Performance overhead: Extended emulation engine
  – Memory overhead: Shadow memory
  – Not applicable to production systems / end hosts

• Maintaining dependencies without taint propagation
  – Handle dependencies
    • Direct value propagation
    • System provided identifiers
      – File, section, process, thread handles
      – Registry keys
      – Socket identifiers
    – Must be constant between call invocations
Matching Behavior Graphs

• Maintaining dependencies *without* taint propagation
  – *Data* dependencies
    • Arbitrary data (& control) dependency between system calls
    • Might modify values between system calls

  – **Our proposal:** *Anticipate* precise call arguments
    • Use recorded execution semantics
    • Extract data *propagation/manipulation formulas*
    • *Emulate* taint dependency between system call A and B
      – Log *outgoing* parameters of call A
      – Use as input to propagation formula
      – Predicted *incoming* parameters for system call B
      – Compare predicted and monitored input parameters
      – Assume dependency between A and B if prediction holds
Matching Behavior Graphs
System Perspective

GetModuleFileNameA

Name

Mode: Open

NtCreateFile

FileHandle

NtCreateSection

SectionHandle

NtMapViewOfSection

( read & decrypt buffer )

NtWriteFile

FileHandle

Mode: Create

C:\Win...\ip6fw.sys
Matching Behavior Graphs
System Perspective

GetModuleFileNameA

Mode: Open
NtCreateFile

Mode: Create
C:\Win\ip6fw.sys
NtCreateFile

NtCreateSection

NtMapViewOfSection

FileHandle

FileHandle

Handle

FileHandle

NtWriteFile

C:\Win\ip6fw.sys
Matching Behavior Graphs
System Perspective

GetModuleFileNameA

IllegalAccessException

FileHandle

NtCreateSection

FileHandle

NtMapViewOfSection

NtWriteFile

NtCreateFile

Mode: Open

Mode: Create

C:\Win...\ip6fw.sys

(read & decrypt buffer)
Matching Behavior Graphs
System Perspective

NtMapViewOfSection

NtMapViewOfSection(out m_buffer[0...size], out m_size)

NtCreateFile(out c_handle)

NtWriteFile
Matching Behavior Graphs
System Perspective

NtMapViewOfSection

NtMapViewOfSection(out m_buffer[0...size],
                  out m_size)

NtCreateFile(out c_handle)

(p_buffer, p_size) = f_4(m_buffer, m_size)

NtWriteFile

NtWriteFile(in w_handle,
in w_buffer[0...size],
in w_size)
Matching Behavior Graphs
System Perspective

NtMapViewOfSection

NtMapViewOfSection(out m_buffer[0...size],
                  out m_size)

NtCreateFile(out c_handle)

(p_buffer, p_size) = ℓ₄(m_buffer, m_size)

NtWriteFile

NtWriteFile(in w_handle,
            in w_buffer[0...size],
in w_size)
Extracting Behavior Graphs
Extracting Behavior Graphs

• Analyze executable in *Anubis* sandbox
  – Obtain instruction level log
    • Defeats packers
  – Obtain program flow log
  – Obtain memory access log
  – Generate *precise* taint propagation trees
    • Data/control dependencies
    • Instructions that access/generate tainted data
    • Link system calls consuming data (sinks) with all taint generating calls (sources)
Extracting Behavior Graphs

- Scan logs for *security relevant* behavior
  - Provided with a list of *interesting* system calls

- Extract graphs matching behavior
  - Include triggering system call $X$
  - Link in system calls providing tainted data to $X$
  - Analyze dependencies:
    - Label edges with *handle* dependencies
    - Call *slicer* for all *data* dependencies

---

![Anubis](image) → **Slicer** → **Scanner**
Extracting Behavior Graphs

- Find encoding formula for each data dependency
- Binary *program slicing*
  - Resolve def-use chains
    - Starting at selected call invocation
    - Iterate backwards (using program flow logs)
    - Aided by *taint information* and *memory access logs*
  - Optional:
    - Symbolic execution to simplify encoding function
  - Embed into dynamically loadable library (dll)
  - Label graph edges with appropriate function (dll)
Extracting Behavior Graphs

- Resolving def-use chains
  - Three possible sources

1) Previous system call invocation
   - Replaced with stub
   - Provides input values to slice (i.e., recorded, outgoing system call parameters)

2) Immediate values
   - Implicitly encoded in binary slice (e.g., `push $0x3`)

![Diagram of Anubis, Slicer, and Scanner]
Extracting Behavior Graphs

- Resolving def-use chains
  - Three possible sources

3) Preinitialized data segments
   - BSS section
     - Constants
     - Static strings
   - Two-sided approach:
     - Use static values from Anubis analysis
     - Dynamically inspect running process
Extracting Behavior Graphs

- Fully automated process
  - Analyze binary
  - Generate behavior graph(s)
  - Extract propagation formulas
  - Verify graph on binary
    - Run binary & scanner on real host
    - Verify behavior graph matches (only) on intended executable
Evaluation
Evaluation

• **Effectiveness** of behavior graphs
  – Applicable to polymorphic variants of a malware sample?
  – General enough for whole malware families?

• **Efficiency of behavior graph matching**
  – Overhead through system call logging
  – Additional system load through dependency verification
Effectiveness

- Six current threats / threat families
- Identified using AV (binary) signature
- Encountered 0 false positives

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Samples</th>
<th>Variants</th>
<th>Samples detected</th>
<th>Eff.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>AV</td>
<td>Our</td>
<td></td>
</tr>
<tr>
<td>Allaple</td>
<td>Exploit-based worm</td>
<td>50</td>
<td>2</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>Beagle</td>
<td>Mass-mailing worm</td>
<td>50</td>
<td>20</td>
<td>14</td>
<td>46</td>
</tr>
<tr>
<td>Mydoom</td>
<td>Mass-mailing worm</td>
<td>50</td>
<td>32</td>
<td>12</td>
<td>47</td>
</tr>
<tr>
<td>Mytob</td>
<td>Mass-mailing worm</td>
<td>50</td>
<td>20</td>
<td>2</td>
<td>41</td>
</tr>
<tr>
<td>Netsky</td>
<td>Mass-mailing worm</td>
<td>50</td>
<td>22</td>
<td>12</td>
<td>46</td>
</tr>
<tr>
<td>Agent</td>
<td>Trojan horse</td>
<td>50</td>
<td>6</td>
<td>3</td>
<td>49</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>300</td>
<td>102</td>
<td>44</td>
<td>279</td>
</tr>
</tbody>
</table>
Effectiveness

- **Experiment:**

  *Can the system detect malware instances never seen by the graph generator?*

<table>
<thead>
<tr>
<th>Name</th>
<th>Samples</th>
<th>AV variants</th>
<th>Samples detected</th>
<th>Eff.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>New</td>
<td>Known</td>
<td></td>
</tr>
<tr>
<td>Allaple</td>
<td>50</td>
<td>0</td>
<td>50</td>
<td>45</td>
</tr>
<tr>
<td>Beagle</td>
<td>50</td>
<td>24</td>
<td>26</td>
<td>30</td>
</tr>
<tr>
<td>Mydoom</td>
<td>50</td>
<td>24</td>
<td>26</td>
<td>36</td>
</tr>
<tr>
<td>Mytob</td>
<td>50</td>
<td>46</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Netsky</td>
<td>13</td>
<td>8</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Agent</td>
<td>50</td>
<td>6</td>
<td>44</td>
<td>45</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>263</strong></td>
<td><strong>108</strong></td>
<td><strong>155</strong></td>
<td><strong>168</strong></td>
</tr>
</tbody>
</table>
Effectiveness

- Experiment:

  *Can the system detect malware instances never seen by the graph generator?*

<table>
<thead>
<tr>
<th>Name</th>
<th>Samples</th>
<th>AV variants</th>
<th>Samples detected</th>
<th>Eff.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>New</td>
<td>Known</td>
<td></td>
</tr>
<tr>
<td>Allaple</td>
<td>50</td>
<td>0</td>
<td>50</td>
<td>45</td>
</tr>
<tr>
<td>Beagle</td>
<td>50</td>
<td>24</td>
<td>26</td>
<td>30</td>
</tr>
<tr>
<td>Mydoom</td>
<td>50</td>
<td>24</td>
<td>26</td>
<td>36</td>
</tr>
<tr>
<td>Mytob</td>
<td>50</td>
<td>46</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Netsky</td>
<td>13</td>
<td>8</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Agent</td>
<td>50</td>
<td>6</td>
<td>44</td>
<td>45</td>
</tr>
<tr>
<td>Total</td>
<td>263</td>
<td>108</td>
<td>155</td>
<td>168</td>
</tr>
</tbody>
</table>
Efficiency

- I-O bound activity
  - Compressing, archiving
- CPU bound computation
  - Compilation, rendering

<table>
<thead>
<tr>
<th>Test</th>
<th>Baseline</th>
<th>Log</th>
<th>Full scanner</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-zip (benchmark)</td>
<td>114 sec</td>
<td>117 sec</td>
<td>2.3 %</td>
</tr>
<tr>
<td>7-zip (compress)</td>
<td>318 sec</td>
<td>328 sec</td>
<td>3.1 %</td>
</tr>
<tr>
<td>7-zip (archive)</td>
<td>213 sec</td>
<td>225 sec</td>
<td>6.2 %</td>
</tr>
<tr>
<td>IE (rendering)</td>
<td>0.41 pages/s</td>
<td>0.39 pages/s</td>
<td>4.4 %</td>
</tr>
<tr>
<td>VC++ (compile)</td>
<td>104 sec</td>
<td>117 sec</td>
<td>12.2 %</td>
</tr>
</tbody>
</table>
Summary

- Behavior can be detected
  - Monitor from system perspective
  - Match against behavior graphs
  - Link graph nodes through argument dependencies
- **Handle** dependencies
  - Vital for checking
  - BUT not specific enough for doing detection
- **Data** dependencies
  - Anticipate future call arguments
  - Efficient replacement for taint dependencies
  - Provided through slicing malware semantics
Summary

• Evaluation
  – Behavior detection is fast enough for end hosts
  – Approach intrinsically robust against polymorphism and metamorphism
  – To some extent, behavior graphs are usable across malware variants
Thanks for your attention!