Advanced Systems Security: Malware Detection

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Malware

- Attack code supplied by an adversary
Malware

- Attack code supplied by an adversary
  - What do you think of when you hear “malware”?
Example: Sirefef

- Windows malware - Trojan to install rootkit
  - Technical details (see Microsoft)
  - And http://antivirus.about.com/od/virusdescriptions/a/What-Is-Sirefef-Malware.htm

- **Attack:** “Sirefef gives attackers full access to your system”
  - Runs as a **Trojan software update** (GoogleUpdate)
  - Runs on each boot by setting a **Windows registry entry**
  - Some versions replace device drivers
  - Downloads code to run a P2P communication
  - Steal software keys and crack password for software piracy
  - Downloads other files to propagate the attack to other computers
Example: Sirefef

- Windows malware - Trojan to install rootkit
  - Technical details (see Microsoft)

- **Stealth:** “while using stealth techniques in order to hide its presence”
  - “altering the internal processes of an operating system so that your antivirus and anti-spyware can't detect it.”
- Disable: Windows firewall, Windows defender
- Changes: Browser settings
- Join bot

- Microsoft: “This list is incomplete”
Malware

- Attack code supplied by an adversary
  - In ROP, an adversary may use existing code maliciously
Malware

- Attack code supplied by an adversary
  - How do we detect that a program contains malware?
Malware

• Attack code supplied by an adversary
  ‣ How do we detect that a program contains malware?
    • Two broad methods…
      ‣ Anomaly and Misuse Detection
Anomaly Detection

- Detect that a program performs “anomalous” behavior
  - Out of the expected behavior for that program
  - How do we know what the “expected behavior” should be and how do we check that at runtime?
Sequences of System Calls

- Forrest et al. in early-mid 90s, attempt to understand the characteristics of an intrusion

- Idea: match sequence of system calls with profiles
  - *n-grams* of system call sequences (learned)
    - Match sliding windows of sequences
    - Record the number of mismatches
    - Use n-grams of length 5, 6, 11.

- If found, then it is normal (w.r.t. learned sequences)
Compare Program Execution

• … to a state machine that describes all legal program executions [David Wagner, PhD thesis]
  ‣ In terms of system calls
  ‣ Why use system calls?

• Finite state automata
  ‣ System calls (essentially) correspond to states and programs transition among them

• Pushdown automata
  ‣ More accurate representation of the execution stack context in which system calls may occur
Finite State Automata Detection

• What system calls may ever follow system call X?
  ‣ E.g., transitions from the state of system call X to each of the successor system calls
  ‣ May use a sequence of system calls to indicate a transition
Pushdown Automata Detection

- What system calls may ever follow system call X in context (stack)?
  - There will be transitions from the state of system call X and call stack to the possible successor system calls from that context.
Limitations

• How would you attack these anomaly detection methods?
Limitations

• How would you attack these anomaly detection methods?

• **Mimicry** [Wagner, CCS 2002]
  ▶ Concoct malware that produces system call sequences that comply with state machines
  ▶ Hard to predict argument values, so can choose them
  ▶ Or ignore results

• Possible to produce an ROP attack that mimics a state machine?
Misuse Detection

- Detect that a program performs “attack” behavior
  - Program performs malicious operations
Misuse Detection

• Classically found via signatures
  ‣ Byte patterns present in malware

• What are some limitations of signatures?
Behavior Graphs

- **Directed acyclic graphs** consisting of a malware’s system calls [Kolbitsch, USENIX 2009]
  - Constrain system call arguments
    - From where is the value derived – system call output
  - \( G = (V, E, F, \partial) \)
    - \( V \): system calls; \( E \): \( V \times V \)
    - \( F \): Function for each system call; \( \partial \): function to arg map
  - Whenever an input argument \( a_i \) for system call \( y \) depends on the some output \( o_j \) produced by system call \( x \), we introduce an edge from the node that corresponds to \( x \), to the node that corresponds to \( y \).
Behavior Graphs – Effective?

- **Training:** Not possible to extract graphs for all

<table>
<thead>
<tr>
<th>Name</th>
<th>Samples</th>
<th>Kaspersky variants</th>
<th>Our variants</th>
<th>Samples detected</th>
<th>Effectiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allaple</td>
<td>50</td>
<td>2</td>
<td>1</td>
<td>50</td>
<td>1.00</td>
</tr>
<tr>
<td>Bagle</td>
<td>50</td>
<td>20</td>
<td>14</td>
<td>46</td>
<td>0.92</td>
</tr>
<tr>
<td>Mytob</td>
<td>50</td>
<td>32</td>
<td>12</td>
<td>47</td>
<td>0.94</td>
</tr>
<tr>
<td>Agent</td>
<td>50</td>
<td>20</td>
<td>2</td>
<td>41</td>
<td>0.82</td>
</tr>
<tr>
<td>Netsky</td>
<td>50</td>
<td>22</td>
<td>12</td>
<td>46</td>
<td>0.92</td>
</tr>
<tr>
<td>Mydoom</td>
<td>50</td>
<td>6</td>
<td>3</td>
<td>49</td>
<td>0.98</td>
</tr>
<tr>
<td>Total</td>
<td>300</td>
<td>102</td>
<td>44</td>
<td>279</td>
<td>0.93</td>
</tr>
</tbody>
</table>

Table 2: Training dataset.

- **Detection:** 92% of “known” samples

<table>
<thead>
<tr>
<th>Name</th>
<th>Samples</th>
<th>Known variant samples</th>
<th>Samples detected</th>
<th>Effectiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allaple</td>
<td>50</td>
<td>50</td>
<td>45</td>
<td>0.90</td>
</tr>
<tr>
<td>Bagle</td>
<td>50</td>
<td>26</td>
<td>30</td>
<td>0.60</td>
</tr>
<tr>
<td>Mytob</td>
<td>50</td>
<td>26</td>
<td>36</td>
<td>0.72</td>
</tr>
<tr>
<td>Agent</td>
<td>50</td>
<td>4</td>
<td>5</td>
<td>0.10</td>
</tr>
<tr>
<td>Netsky</td>
<td>13</td>
<td>5</td>
<td>7</td>
<td>0.54</td>
</tr>
<tr>
<td>Mydoom</td>
<td>50</td>
<td>44</td>
<td>45</td>
<td>0.90</td>
</tr>
<tr>
<td>Total</td>
<td>263</td>
<td>155</td>
<td>168</td>
<td>0.64</td>
</tr>
</tbody>
</table>

Table 3: Detection effectiveness.
Study Malware

- Malware is “in the wild”
  - Can’t we study it and learn its behavior and defenses against that behavior?
Anti-Reversing

• Art of Unpacking
  ‣ Now malware developers actively develop their malware to evade analysis
Anti-Reversing

- **Art of Unpacking**
- Detect various side channels created when using tools to analyze malware
- E.g., Debuggers (Windows)
  - Software breakpoint
    - Modify code – rewrite instructions to trap to debugger
  - Hardware breakpoint
    - Debug registers are set
Anti-Reversing

- Art of Unpacking
- Detect various side channels created when using tools to analyze malware
- E.g., Debuggers (Windows)
  - Others
    - Slow the execution – can detect time delays (rdtsc)
    - Debugger privileges asserted
    - Parent process is different
    - Debug windows are created
    - Debugger processes are among tasks
Anti-Reversing

• Art of Unpacking

• Proactive defenses against analysis
  ‣ Encryption
  ‣ Compression
  ‣ Permutation
  ‣ Garbage code

• What is the benefit of garbage code to confusing the reverser?
Avoid Detection

- Modify debuggers
- Hide debuggers from the system (like malware hides processes)
- Don’t use debuggers
- Avoid software and hardware breakpoints
- …
Reversing with SMM

- **System management mode (SMM)**
  - Sometimes called “ring -2”
  - Specific to Intel x86 processors
    - “all normal execution, including the operating system, is suspended and …” [Wikipedia]
    - “special separate software, which is usually part of the firmware or a hardware-assisted debugger, is executed with high privileges” [Wikipedia]
- Originally for power management and low-level systems management
Reversing with SMM

- **System management mode (SMM)**
  - Can SMM configuration be interrogated by malware running at user-level?
  - …as opposed to a debugger that runs at the same privilege level
Malware Analysis in SMM

• Analyze malware at SMI (interrupt)
  ‣ Can be asserted by software or hardware
    • **Software**: Write to Advanced Configuration and Power Interface (ACPI) port
      ‣ I.e., add an instruction (out) to malware code – i.e., write code
    • **Hardware**: Two ways
      ‣ (1) Serial interrupt: configuring the redirection table in I/O Advanced Programmable Interrupt Controller (APIC)
      ‣ (2) Counter: set the corresponding performance counter (PerfCtr0) register to the maximum value
Malware Analysis in SMM

• Analyze malware at SMI (interrupt)
  ▶ Can be asserted by software or hardware
    • **Software**: Write to Advanced Configuration and Power Interface (ACPI) port
      ▶ Adversary can detect malware code modifications
    • **Hardware**: Two ways
      ▶ (1) Serial interrupt: configuring the redirection table in I/O Advanced Programmable Interrupt Controller (APIC)
      ▶ (2) Counter: **Adversary can read performance counters from user space**
Malware Analysis in SMM

- Analyze malware at SMI (interrupt)
- Can be asserted by software or hardware
- Software: Write to Advanced Configuration and Power Interface (ACPI) port
  - I.e., add an instruction (out) to malware code – i.e., write code
- Hardware: Two ways
  1. Serial interrupt: configuring the redirection table in I/O Advanced Programmable Interrupt Controller (APIC)
  2. Counter: set the corresponding performance counter (PerfCtr0) register to the maximum value

Malware Lifecycle

Original Malware
Source-code or DIY malware creator kit
generates original code.

Code Metamorphism
Random changes to the codes structures
and procedures.

Noise Insertion
Insertion of noise instructions and
whitespace commands.

Compilers
Different compilers (and versions) are used
to generate different code.

Copyright © 2009-2010 Damballa, Inc. All Rights Reserved
Modern Malware

• Now malware has a whole other level of sophistication
• Now we speak of …
  • Advanced Persistent Malware
Persistent

• Malware writers are focused on specific task
• Criminals willing to wait for gratification
• Cyberwarfare

• Low-and-slow
• Can exfiltrate secrets at a slow rate, especially if you don't need them right away

• Plus can often evade or disable defenses
"gedanken experiment"

- Assume a very good anomaly detector (99%)
- And a pretty constant attack rate, where you can observe 1 out of 10000 events are malicious

- Are you going to detect the adversary well?
Bayes’ Rule

• \( \text{Pr}(x) \) function, probability of event \( x \)
  ‣ \( \text{Pr}(\text{sunny}) = .8 \) (80% of sunny day)

• \( \text{Pr}(x|y) \), probability of \( x \) given \( y \)
  ‣ Conditional probability
  ‣ \( \text{Pr}(\text{cavity}|\text{toothache}) = .6 \)
    ‣ 60% chance of cavity given you have a toothache
  ‣ Bayes’ Rule (of conditional probability)

\[
\text{Pr}(B|A) = \frac{\text{Pr}(A|B) \text{ Pr}(B)}{\text{Pr}(A)}
\]
The (base-rate) Bayesian Fallacy

• Setup
  ‣ Pr(T) is attack probability, 1/10,000
    • Pr(T) = .0001
  ‣ Pr(F) is probability of event flagging, unknown
  ‣ Pr(F|T) is 99% accurate (higher than most techniques)
    • Pr(F|T) = .99, Pr(!F|T) = .01, Pr(F|!T) = .01, Pr(!F|!T) = .99

• Deriving Pr(F)
  ‣ Pr(F) = Pr(F|T)*Pr(T) + Pr(F|!T)*Pr(!T)
  ‣ Pr(F) = (.99)(.0001) + (.01)(.9999) = .010098

• Now, what’s Pr(T|F)?
The Bayesian Fallacy

• Now plug it in to Bayes Rule

\[
Pr(T|F) = \frac{Pr(F|T) \cdot Pr(T)}{Pr(F)} = \frac{Pr(.99) \cdot Pr(.0001)}{Pr(.010098)} = .0098
\]

• So, a 99% accurate detector leads to …
  ‣ 1% accurate detection.
  ‣ With 99 false positives per true positive
  ‣ This is a central problem with IDS

• Suppression of false positives real issue
  ‣ Open question, makes some systems unusable
### Where is Anomaly Detection Useful?

| System | Attack Density $P(T)$ | Detector Flagging $Pr(F)$ | Detector Accuracy $Pr(F|T)$ | True Positives $P(T|F)$ |
|--------|------------------------|---------------------------|-----------------------------|------------------------|
| A      | 0.1                    |                           | 0.65                        |                        |
| B      | 0.001                  |                           | 0.99                        |                        |
| C      | 0.1                    |                           | 0.99                        |                        |
| D      | 0.00001                |                           | 0.99999                     |                        |

\[
Pr(B|A) = \frac{Pr(A|B) \cdot Pr(B)}{Pr(A)}
\]
Where is Anomaly Detection Useful?

| System | Attack Density $P(T)$ | Detector Flagging $Pr(F)$ | Detector Accuracy $Pr(F|T)$ | True Positives $P(T|F)$ |
|--------|-----------------------|---------------------------|-----------------------------|-------------------------|
| A      | 0.1                   | 0.38                      | 0.65                        | 0.171                   |
| B      | 0.001                 | 0.01098                   | 0.99                        | 0.090164                |
| C      | 0.1                   | 0.108                     | 0.99                        | 0.911667                |
| D      | 0.00001               | 0.00002                   | 0.99999                     | 0.5                     |

$$Pr(B|A) = \frac{Pr(A|B) Pr(B)}{Pr(A)}$$
Take Away

- Problem: Detect malware before it is run
- In general, we can try to detect anomalies or misuse, but both have significant challenges
- Anomaly detection must detect that a running process really runs malware – model of expected
- Misuse detection must detect malice – and other examples of same malice – models
- Malware writers now make reversing difficult
- Intrusion detection is hard to do accurately w/o causing false positives