Dynamic Malware Analysis

Christopher Kruegel
Secure Systems Lab
Technical University Vienna
Malicious Code

- There is a wide variety of different types of malicious code
  - viruses, worms, spyware, rootkits, Trojan horses, ...

- Common characteristic
  - perform some unwanted activity on your system

- No doubt, everybody has heard of viruses, worm epidemics, or spyware
  - reports in mainstream media
  - personal experience (at least, with virus scanners)
Malicious Code Analysis

• Understanding functionality of malware programs
  – modifications to compromised system
  – understand questions such as:
    how is program launched, what malicious actions are performed,
    hidden functionality (with trigger), disabling of defense mechanisms,
    interaction with other processes …

• Necessary both for *detection* and removal

• Must keep up with increasing numbers of samples
  – fast
  – automated (at least, provide as much support as possible)
  – precise

• Interesting with regards to automated malware collection (honeypots)
Malicious Code Analysis

Static analysis versus dynamic analysis

• Static analysis
  – code is not executed
  – all possible branches can be examined (in theory)
  – quite fast

• Problems of static analysis
  – undecidable in general case, approximations necessary
  – binary code typically contains very little information
  – disassembly difficult (particularly for Intel x86 architecture)
  – obfuscated code, packed code
  – self-modifying code
  – polymorphic code
Malicious Code Analysis

• Dynamic analysis
  – code is executed
  – sees instructions that are actually executed

• Problems of dynamic analysis
  – in general, single path (execution trace) is examined
  – analysis environment possibly not \textit{invisible}
  – analysis environment possibly not \textit{comprehensive}
  – scalability issues

• Possible analysis environments
  – instrument program, operating system, or hardware
Malicious Code Analysis

• Instrument program
  – analysis operates in same address space as sample
  – manual analysis with debugger
  – Detours (Windows API hooking mechanism)

  – binary under analysis is modified
    • breakpoints are inserted
    • functions are rewritten
    • debug registers are used
  – not invisible, malware can detect analysis
  – can cause significant manual effort
Malicious Code Analysis

• Instrument operating system
  – analysis operates in OS where sample is run
  – Windows system call hooks

  – invisible to (user-mode) malware
  – can cause problems when malware runs in OS kernel
  – limited visibility of activity inside program
    • cannot set function breakpoints

• Virtual machines
  – allow to quickly restore analysis environment
  – might be detectable (x86 virtualization problems)
Malicious Code Analysis

• Instrument hardware
  – provide virtual hardware (processor) where sample can execute (sometimes including OS)
  – software emulation of executed instructions
  – analysis observes activity “from the outside”
    – completely transparent to sample (and guest OS)
    – operating system environment needs to be provided
    – limited environment could be detected, but faster
    – complete environment is comprehensive, but slower

→ Anubis - PC emulator with “real” operating system
Semantic Gap

• Analysis observes activity “from the outside”
  – how to relate instructions executed on processor with activity inside the operating system and sample?
  – must encode some knowledge about guest OS and processes
    • we chose to target MS Windows on Intel x86 machines
  – Qemu used as emulation environment

• Questions
  – how to identify which instructions belong to malware sample?
  – what to analyze?
  – how to implement analysis?
• Process interacts with operating system via system calls
  – needs OS for every interaction with environment
    • file system, network, registry, …
  – monitor system calls
  – unfortunately, on Windows, system calls largely undocumented
    and can change without notice
  – developers are supposed to use Windows API, which
    denotes a collection of stable, user-mode, shared libraries
  – of course, Windows API can be bypassed

→ we monitor Windows API calls and NT kernel calls
Analysis Report

• File activity
  – read, write, create, open, …

• Registry activity

• Service activity
  – start or stop of Windows services (via Service Manager)

• Process activity
  – start, terminate process, inter-process communication

• Network activity
  – API calls and packet logs

• Let’s look at some example reports
Anubis Architecture

- Honeypots (Nepenthes)
- Public Web Interface
- Active Web Crawling
- Mail (Spam) Attachments
- Analysis Reports
- Bot C&C Info
- Detection
- Clustering
Anubis Usage

Submission Statistics

Total Samples: 243,252
Unique Samples: 151,330
Malware Detection

• Run simple rules on output
  – can flag scanners (number of contacted IP addresses)
  – keyboard loggers (installed keyboard hooks)
  – mass mailers (spam mails sent)
  – bots (suspicious IRC traffic)
  – copy to system directory

• We can do a more powerful analysis
  – after all, we have a system emulator and complete control
  – capture information flows as well (tainting)
  – detect unusual information access and processing patterns
Information Flow Tracking

• Information flow tracking
  – performed on hardware level, using a virtual machine / system emulator
  – each data element of interest is labeled (tainted)
  – bytes in (emulated) physical memory are labeled, using a shadow memory

• Propagation policy
  – for each machine instruction
  – defines how taint information is propagated

• We modified Anubis to implement taint propagation
  – introduced shadow memory
  – extended x86 instruction semantics
Taint Graph Generation

- Install sample and run common user scenarios
  - open file, write text, save file
  - open web page, enter information in password fields
  - connect to an ftp server

- Track flow of information
  - between processes, files, and network
  - generate corresponding graphs
Taint Graph Analysis

• Idea is to use program *behavior* as model
  – specify characteristic behavior for a certain class of malicious code
  – general model that covers all (most, some) instances of a class

• Policies
  – capture generic, malicious behavior
  – describe anomalous information processing and access
  – based on taint graphs
Current detection policies

1. Anomalous information access
   – access to keyboard information not destined for suspicious process
   – access to network packets that are not sent to suspicious process
   – examples - keylogger, password thief, network sniffer

2. Anomalous information leakage
   – access local information and then send it out or store it on disk
   – example - spyware

3. Excessive information access
   – access file (meta)-information (disk blocks) on every file access
   – example - rootkit
## Analysis Results

<table>
<thead>
<tr>
<th>Category</th>
<th>Total</th>
<th>FNs</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyloggers</td>
<td>5</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Password thieves</td>
<td>2</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Network sniffers</td>
<td>2</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Backdoors</td>
<td>3</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Spyware</td>
<td>22</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Rootkits</td>
<td>8</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Browser plug-ins</td>
<td>16</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Multi-media</td>
<td>9</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Security</td>
<td>10</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>System utilities</td>
<td>9</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Office productivity</td>
<td>4</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Games</td>
<td>4</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Others</td>
<td>4</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>98</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Ongoing Spyware Experiments

• Web crawling
  – instead of manually collecting samples, let a crawler do the job
  – requires automated installation of programs (spyware bundles)

• Automated installation routines
  – large set of heuristics (i.e., script hacks)

• Automated tool to drive a web browser
  – simulate user activity

• Web interface to display results (current interface)
  – compared against commercial tool (AdAware)
  – checked BHO identifier (CLS-ID) against list of known CLSIDs (CastleCops)
Anubis Analysis Issues

- **Evasion**
  - attacks against Qemu
  - specific attacks against Anubis sandbox (Ryan forum)
  - blacklisting of our IP addresses and DNS names (example report)

- **Scalability**
  - 3 minutes (real-time) per analysis
  - one instance means 480 samples a day (14,400 per month)
  - we have a cluster of 7 parallel instances at the moment

- **Single execution path only**
  - may miss trigger behavior
  - some malware disables itself after some deadline
Increase Dynamic Coverage

• Idea
  – explore multiple execution paths of executable under test
  – exploration is driven by monitoring how program uses certain inputs
  – system should also provide information under which circumstances a certain action is triggered

• Approach
  – track “interesting” input when it is read by the program
  – whenever a control flow decision is encountered that uses such input, two possible paths can be followed
  – save snapshot of current process and continue along first branch
  – later, revert back to stored snapshot and explore alternative branch
Exploring Multiple Paths

- **Tracking input**
  - we already know how to do this (tainting)

- **Snapshots**
  - we know how to find control flow decision points (branches)
  - snapshots are generated by saving the content of the process’ virtual address space (of course, only used parts)
  - restoring works by overwriting current address space with stored image

- **Explore alternative branch**
  - restore process memory image
  - set the tainted operand (register or memory location) to a value that reverts branch condition
  - let the process continue to run
Exploring Multiple Paths

- Unfortunately, it is not that easy
  - when only rewriting the operand of the branch, process state can become inconsistent
  - input value might have been copied or used in previous calculations

```c
x = read_input();
y = 2*x + 1;
check(y);
print("x = %d, x");

void check(int magic) {
    if (magic != 47)
        exit();
    print("x = %d",x);
}
```

This prints \( x = 0 \)!

We have to remember that \( y \) depends on \( x \), and that \( magic \) depends on \( y \).
Exploring Multiple Paths

• Tracking of input must be extended
  – whenever a tainted value is copied to a new location, we must remember this relationship
  – whenever a tainted value is used as input in a calculation, we must remember the relationship between the input and the result

• Constraint set
  – for every operation on tainted data, a constraint is added that captures relationship between input operands and result
  – currently, we only model linear relationships
  – can be used to perform consistent memory updates when exploring alternative paths
  – provides immediate information about condition under which path is selected
Exploring Multiple Paths

- Constraint set

```c
x = 0
x = read_input();
y = 2*x + 1;
check(y);
print("x = %d, x");
....

void check(int magic) {
    if (magic != 47)
        exit();
}
```

- Now, print outputs “x = 23”

- x == input
- y == 2*x + 1
- magic == y
- magic == 47

- solve for alternative branch
- x == input == 23
- y == magic == 47
- Now, print outputs “x = 23”
Exploring Multiple Paths

• Path constraints
  – capture effects of conditional branch operations on tainted variables
  – added to constraint set for certain path

```cpp
x = read_input();

if (x > 10)
    if (x < 15)
        interesting();

exit();
```

![Diagram of path constraints]

- \( x \leq 10 \)
- \( x > 10 \)
- \( x > 10 \)
- \( x < 15 \)
- \( x \geq 15 \)
- interesting();
- exit();
Exploring Multiple Paths

- **Snapshot management**
  - problem when process causes external effects --
    - write to a file, close an OS handle, free memory, …

- **How are external effects reverted?**
  - we never let process return any allocated resource to the OS
  - some special handling for file operations (e.g., write, delete)

- **Some problems remain**
  - in particular, it is very difficult to revert to a snapshot during an active network connection (and maintain this connection)
Experimental Results

- 308 malicious executables
  - large variety of viruses, worms, bots, Trojan horses, …

<table>
<thead>
<tr>
<th>Interesting input sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check for Internet connectivity</td>
</tr>
<tr>
<td>Check for mutex object</td>
</tr>
<tr>
<td>Check for existence of file</td>
</tr>
<tr>
<td>Check for registry entry</td>
</tr>
<tr>
<td>Read current time</td>
</tr>
<tr>
<td>Read from file</td>
</tr>
<tr>
<td>Read from network</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Additional code coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
</tr>
<tr>
<td>0% - 10%</td>
</tr>
<tr>
<td>10% - 50%</td>
</tr>
<tr>
<td>50% - 200%</td>
</tr>
<tr>
<td>&gt; 200%</td>
</tr>
</tbody>
</table>

Additional code is likely for error handling

Relevant behavior:
- time-triggers
- filename checks
- bot commands
Conclusions

- Malware analysis
  - amount of malware to analyze increases at alarming rate
  - require robust automated techniques

- Anubis
  - dynamic analysis environment
  - detection using dynamic analysis results
  - leverage program’s usage of information and suspicious flows

- Analysis issues
  - stealthy analysis, scalability, single execution path
  - remedy - exploring multiple execution paths