Identifying Dormant Functionality in Malware Programs

Paolo Milani Comparetti
Guido Salvaneschi
Engin Kirda
Clemens Kolbitsch
Christopher Kruegel
Stefano Zanero

Vienna University of Technology
Politecnico di Milano
Institute Eurecom
Vienna University of Technology
UC Santa Barbara
Politecnico di Milano
Motivation

- Malicious code (malware) at the root of many internet security problems
  - ~50000 new samples each day!
- Automated dynamic analysis
  - run samples in an instrumented sandbox
- Dynamic analysis provides limited coverage
  - different behavior based on commands from C&C channel
- How can we learn more about malware samples?
Our Approach

• Leverage code reuse between malware samples

• Automatically generate semantic-aware models of malicious behavior
  – based on 1 execution of a behavior
  – model 1 implementation of the behavior

• Use these models to statically detect the malicious functionality in samples that do not perform that behavior during dynamic analysis
REANIMATOR

• Run malware in monitored environment and detect a malicious behavior (*phenotype*)

• Identify and model the code responsible for the malicious behavior (*genotype model*)

• Match genotype model against other binaries
Outline

• **Reanimator**: Identifying dormant functionality
  – Dynamic behavior identification
  – Extracting genotype models
  – Finding dormant functionality

• Evaluation

• Conclusions
**REANIMATOR**

A. Dynamic Behavior Identification

B. Extracting Genotypes Models

C. Finding Dormant Functionality

Malware sample

Behaviors

Genotype models

New malware samples

\[ B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow \ldots \rightarrow B_n \]

\[ \gamma_1, \gamma_2, \gamma_3, \ldots, \gamma_n \]
Dynamic Behavior Identification

A. Dynamic Behavior Identification

\[ O' \rightarrow B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow \ldots \rightarrow B_n \]

B. Extracting Genotypes Models

C. Finding Dormant Functionality

Genotype models

New malware samples
Dynamic Behavior Identification

- Run malware in instrumented sandbox
  - Anubis
- Dynamically detect a behavior $B$ (*phenotype*)
- Map $B$ to the set $R_B$ of system/API call instances responsible for it
- $R_B$ is the output of the behavior identification phase
Behavior Detection Examples

- **spam**: send SMTP traffic on port 25
  - network level detection
- **sniff**: open promiscuous mode socket
  - system call level detection
- **rpcbind**: attempt remote exploit against a specific vulnerability
  - network level detection, with snort signature
- **drop**: drop and execute a binary
  - system call level detection, using data flow information
- **...**
Extracting Genotype Models

A. Dynamic Behavior Identification
B. Extracting Genotypes Models
C. Finding Dormant Functionality

$O'$
Malware sample
 Behaviors

$B_1 \quad B_2 \quad B_3 \ldots B_n$

Genotype models

$\gamma_1 \quad \gamma_2 \quad \gamma_3 \ldots \gamma_n$

New malware samples
Extracting Genotype Models

- Take as input the set $R_B$ of relevant system/API calls
- **Identify** the code responsible for behavior B (genotype)
- **Model** the code responsible for behavior B (genotype model)
- The genotype model can then be statically, efficiently used for detecting the corresponding genotype and phenotype in other binaries that did not perform B during dynamic analysis
Extracting Genotype Models: Goals

• Identified genotype should be precise and complete

• Complete: include all of the code implementing B

• Precise: do not include code that is not specific to B (utility functions,..)
Extracting Genotype Models: Steps

• Slicing:
  – obtain an initial set of instructions (genotype) $\phi$ that are related to $R_B$

• Filtering:
  – increase the precision of the genotype by removing from $\phi$ instructions that are not specific to $B$

• Germination:
  – increase the completeness of the model by adding instructions to $\phi$
Step 1: Slicing

- Start from relevant calls $R_B$
- Include into slice $\phi$ instructions involved in:
  - preparing input for calls in $R_B$
    - follow data flow dependencies backwards from call inputs
  - processing the outputs of calls in $R_B$
    - follow data flow forward from call outputs
- We do not consider control-flow dependencies
  - would lead to including too much code (taint explosion problem)
Step 2: Filtering

- The slice $\phi$ is not precise
- General purpose utility functions executed as part of behavior are included (i.e: string processing)
  - may be from statically linked libraries (i.e: libc)
  - genotype model would match against any binary that links to the same library
- Backwards slicing goes too far back: initialization and even unpacking routines are often included
  - genotype model would match against any malware packed with the same packer
Filtering Techniques

• Exclusive instructions:
  – set of instructions that manipulate tainted data every time they are executed
  – utility functions are likely to be also invoked on untainted data

• Discard whitelisted code:
  – whitelist obtained from other tasks or execution of the same sample, that do not perform B
  – could also use foreign whitelist
    • i.e: including common libraries and unpacking routines
Step 3: Germination

- The slice $\phi$ is not complete
- Auxiliary instructions are not included
  - loop and stack operations, pointer arithmetic, etc
- Add instructions that cannot be executed without executing at least one instruction in $\phi$
- Based on graph reachability analysis on the intra-procedural Control Flow Graph (CFG)
Finding Dormant Functionality

A. Dynamic Behavior Identification

\[ O' \rightarrow B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow \cdots \rightarrow B_n \]

B. Extracting Genotypes Models

\[ \gamma_1, \gamma_2, \gamma_3, \cdots, \gamma_n \]

C. Finding Dormant Functionality

Genotype models

New malware samples
Finding Dormant Functionality

- Genotype is a set of instructions
- Genotype model is its colored control flow graph (CFG)
  - nodes colored based on instruction classes
- 2 models match if they share at least a K-Node subgraph (K=10)
- Use techniques from our previous work [1] to efficiently match a binary against a set of genotype models
- We use Anubis as a generic unpacker

Evaluation
Evaluation

• Extract genotype models from a sample
• Match these genotypes against other samples
• Are the results accurate?
  - when REANIMATOR detects a match, is there really the dormant behavior?
  - how reliably does REANIMATOR detect dormant behavior in the face of recompilation or modification of the source code?
• Are the results insightful?
  - does REANIMATOR reveal behavior we would not see in dynamic analysis?
Accuracy

• To test accuracy and robustness of our system we need a ground truth
• Dataset of 208 bots with source code
  – thanks to Jon Oberheide and Michael Bailey from University of Michigan
• Extract 6 genotype models from 1 bot
• Match against remaining 207 bot binaries
Accuracy

• Even with source, manually verifying code similarity is time-consuming
• Use a source code plagiarism detection tool
  – MOSS
• We feed MOSS the source code corresponding to each of the 6 behaviors
  – match it against the other 207 bot sources
  – MOSS returns a similarity score in percentage
• We expect REANIMATOR to match in cases where MOSS returns high similarity scores
MOSS Comparison

Percentage of matching code according to MOSS

Number of <behavior, bot> pairs

MOSS
Reanimator

IEEE Symposium on Security & Privacy, May 17 2010
MOSS Comparison

Potential False Negatives

Potential False Positives

Percentage of matching code according to MOSS

Number of <behavior,bot> pairs
Accuracy Results

- We manually investigated the potential false positives and false negatives
- Low false negative rate (~1.5%)
  - mostly small genotypes
- No false positives
  - genotype model match always corresponds to presence of code implementing the behavior
- Also no false positives against dataset of ~2000 benign binaries
  - binaries in system32 on a windows install
Robustness

- Robustness results when re-compiling same source
- Robust against different compilation options (<7% false negatives)
- Robust against different compiler versions
- Not robust against completely different compiler (>80% false negatives)
  - Visual Studio vs. Intel
- Some robustness to malware metamorphism was demonstrated in [1]
In-the-Wild Detection

• 10 genotype models extracted from 4 binaries
• 4 datasets
  – irc_bots: 10238 IRC bots
  – packed_bots: 4523 packed IRC bots
  – pushdo: 77 pushdo binaries (dropper, typically drops spam engine cutwail)
  – allaple: 64 allaple binaries (network worm)
• Reanimator reveals a lot of functionality not observed during dynamic analysis
# In-the-Wild Detection

B: Behavior observed in dynamic analysis.

S,D: Functionality detected by Reanimator

<table>
<thead>
<tr>
<th>Genotype</th>
<th>Phenotype</th>
<th>irc_bots</th>
<th></th>
<th>packed_bots</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>B</td>
<td>S</td>
<td>D</td>
<td>B ∩ S</td>
</tr>
<tr>
<td>httpd</td>
<td>backdoor</td>
<td>2014</td>
<td>636</td>
<td>635</td>
<td>279</td>
</tr>
<tr>
<td>keylog</td>
<td>keylog</td>
<td>0</td>
<td>293</td>
<td>254</td>
<td>0</td>
</tr>
<tr>
<td>killproc</td>
<td>killproc</td>
<td>0</td>
<td>400</td>
<td>400</td>
<td>0</td>
</tr>
<tr>
<td>simplespm</td>
<td>spam</td>
<td>154</td>
<td>409</td>
<td>409</td>
<td>0</td>
</tr>
<tr>
<td>udpflood</td>
<td>packetflood</td>
<td>0</td>
<td>374</td>
<td>342</td>
<td>0</td>
</tr>
<tr>
<td>sniff</td>
<td>sniff</td>
<td>43</td>
<td>270</td>
<td>72</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Genotype</th>
<th>pushdo</th>
<th></th>
<th>allaple</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>S</td>
<td>D</td>
<td>B ∩ S</td>
</tr>
<tr>
<td>drop</td>
<td>50</td>
<td>54</td>
<td>54</td>
<td>46</td>
</tr>
<tr>
<td>spam</td>
<td>1</td>
<td>43</td>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td>scan</td>
<td>23</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rpcbind</td>
<td>5</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Conclusions

• Identify security-relevant behavior during dynamic analysis of a malware sample
• Automatically identify and model the code that is responsible for that behavior
• Use these models to statically detect similar code in other samples
• Our experiments demonstrate accuracy and robustness
• Testing against in-the-wild datasets shows improved detection of malicious functionality
Questions?