JSTAP: Malicious JavaScript Detection

Dr. Martin “Doc” Carlisle
• Find malicious JavaScript
  – Bitcoin mining
  – Abuse browser vulnerabilities
  – Perform static analysis with abstract syntax trees and random forests
    • Static analysis means we don’t run the code at all
Static Analyses (I)

- Abstract Syntax Tree
  - Derived from grammar of programming language

An abstract syntax tree for the following code for the Euclidean algorithm:

```plaintext
while b ≠ 0
    if a > b
        a := a - b
    else
        b := b - a
return a
```
Static Analyses (II)

- Control Flow Graph
  - Shows program flow (calls, selection, loops)

Some CFG examples:
(a) an if-then-else
(b) a while loop
(c) a natural loop with two exits, e.g. while with an if...break in the middle; non-structured but reducible
(d) an irreducible CFG: a loop with two entry points, e.g. goto into a while or for loop
Static Analyses (III)

• Program Dependence Graph
  – Includes data and control dependencies
    • A=B*C
    • D=A*E+1 (this depends on the prior statement)

• if (A) then
  – B=C*D (this depends on value of A)
• endif
# JavaScript Tokens

### Listing 1: JavaScript code example

```javascript
1. x.1f = 1;
2. var y = 1;
3. if (x.1f == 1) { d = y; }
```

### Table 1: Lexical units extracted from the code of Listing 1

<table>
<thead>
<tr>
<th>Token</th>
<th>Value</th>
<th>Token</th>
<th>Value</th>
<th>Token</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>x</td>
<td>Numeric</td>
<td>1</td>
<td>Punctuator</td>
<td>)</td>
</tr>
<tr>
<td>Punctuator</td>
<td>.</td>
<td>Punctuator</td>
<td>;</td>
<td>Identifier</td>
<td>d</td>
</tr>
<tr>
<td>Keyword</td>
<td>if</td>
<td>Keyword</td>
<td>if</td>
<td>Identifier</td>
<td>x</td>
</tr>
<tr>
<td>Punctuator</td>
<td>=</td>
<td>Punctuator</td>
<td>(</td>
<td>Punctuator</td>
<td>=</td>
</tr>
<tr>
<td>Numeric</td>
<td>1</td>
<td>Identifier</td>
<td>x</td>
<td>Identifier</td>
<td>y</td>
</tr>
<tr>
<td>Punctuator</td>
<td>;</td>
<td>Punctuator</td>
<td>.</td>
<td>Identifier</td>
<td>x</td>
</tr>
<tr>
<td>Keyword</td>
<td>var</td>
<td>Keyword</td>
<td>if</td>
<td>Punctuator</td>
<td>}</td>
</tr>
<tr>
<td>Identifier</td>
<td>y</td>
<td>Punctuator</td>
<td>==</td>
<td>Numeric</td>
<td>1</td>
</tr>
</tbody>
</table>
JavaScript example

Figure 2: AST corresponding to the code of Listing 1

```javascript
1. x.1f = 1;
2. var y = 1;
3. if (x.1f == 1) { d = y; }
```

Listing 1: JavaScript code example
JavaScript example

Figure 3: AST of Listing 1 extended with control flow (red dotted edges) and data flow (blue dashed edges)

```javascript
1. x.if = 1;
2. var y = 1;
3. if (x.if == 1) {d = y;}
```

Listing 1: JavaScript code example
N-grams

• Simple way to analyze token sequences

• Example with n=3

\[
\begin{align*}
\text{ID} = \text{ID} + \text{NUM} & \rightarrow \{(\text{ID} = \text{ID}), (= \text{ID} +), (\text{ID} + \text{NUM})\}, \\
\text{SET a.b to } "x" & \rightarrow \{(\text{SET a.b to}), (a.b \text{ to } "x")\}.
\end{align*}
\]
JSTAP n-grams (I)

• Depth-first pre-order traversal of AST
• For CFG, also traverse AST, but only nodes linked by control flow edge.
  – Traverse sub-AST for each node with control flow once
• Similar for PDG, considering data flow
• Independent n-grams for tokens, AST, CFG, PDG-Data Flow and PDG-Control Flow
JSTAP n-grams (II)

• Set $n=4$ (experimentally)
• Use chi-squared test to check for correlation, keep $\chi^2 \geq 6.63$ (confidence of 99%)
  – Lets us throw away a lot of n-grams

<table>
<thead>
<tr>
<th>Table 2: Number of relevant features per module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tokens</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>ngrams</td>
</tr>
<tr>
<td>value</td>
</tr>
</tbody>
</table>
JSTAP dataset

• 131,448 malicious JavaScript files
  – German Federal Office for Info Security
  – Hynek, DNC, GeeksOnSecurity, Virus Total

• 141,768 benign files
  – Top 10,000 Tranco websites
  – JS from Exchange 2016 and Team Foundation Server 2017
    • So obfuscation isn’t confused with maliciousness
JSTAP Classifier Training

• Select 10,000 malicious and benign randomly for training
  – Additional 5,000 of each for validation
• Repeat 5 times and average detection results
An interesting claim

- Fass et al. “For this reason, AUC and F-measure would be heavily biased by the composition of our test sets”
- Fawsett “ROC curves have an attractive property: they are insensitive to changes in class distribution. If the proportion of positive to negative instances changes in a test set, the ROC curves will not change.”
JSTAP Results

Figure 4: Accuracy comparison with the n-grams approach

Figure 5: Accuracy comparison with the value approach
Cujo: 4-grams better than 3-grams, and random forest better than SVM
Zozzle: all nodes (not just exprs and var decls), random forest vs naïve Bayes
JAST: do not simplify but use $\chi^2$ test to reduce size of feature space
JSTAP results

• Two step process
  • First phase
    – Unanimous voting, classifies 93% of data with 99.73% accuracy
  • Second phase
    – Unanimous voting, classifies 6.5% of data with accuracy still over 99%
Evasion techniques

• Add more benign features
• Copy malicious into larger benign file
Malware Detection by Extreme Abstraction

Dr. Martin “Doc” Carlisle
Premise

- Find malicious Windows EXEs by abstract execution
  - Less precise than virtualization or emulation
Why dynamic analysis

- Malware writers deliberately obfuscate to defeat static tools
  - Example: GozNym runs trivial infinite loop in thread, then suspends thread and overwrites code with jump to previously dead code
Dynamic Analysis pitfalls

• Easy to detect you are in a debugger, VM, or running Anti-virus
  – Query registry
  – IsDebuggerPresent
  – VM specific instructions

• Do long delay in hopes simulator will give up and go away

0x4017c0: mov esi, dword ptr [ebp-0x26]
0x4017c3: mov esi, dword ptr [esi]
0x4017c5: xor esi, edi
0x4017c7: inc edi
0x4017c8: cmp esi, 0x90909090
0x4017ce: jne 0x4017c0

Figure 2: A long delay loop
Extremely Abstract OS

- Over-approximation has more behaviors than system S, under-approximation has fewer
  - If over-approximation does no evil, great!
  - If under-approximation does, then boo!
Extremely Abstract OS

Dynamic Analysis

<table>
<thead>
<tr>
<th>Emulator</th>
<th>Extremely abstract OS</th>
<th>Multiple paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>Windows API</td>
<td>Exception handling</td>
</tr>
</tbody>
</table>

Sample or bundle

Lightweight symbols / time models

Analysis result

Figure 4: An extremely abstract operating system
TAMALES features

• X86 emulator
• Abstract Windows
  – Most routines return random result and ignore params
    • Over 100,000 API calls from 150 DLLs
    • Strcpy, memmove work as expected
    • Some file read/write and registry read/write
    • Network is abstract
  – Rdtsc – time-stamp counter handled specially
  – Cpuid – handled specially
• Runs on Linux (just in case....)
Another unique case

• SetErrorMode

```c
x = y;
```

can be implemented (directly by the malware writer or more probably by an obfuscating compiler) as follows:

```c
SetErrorMode(y);
x = SetErrorMode(arbitrary_value);
```

• Since malware writers do this, must implement for real
More malware functions

- WriteProcessMemory
- CreateRemoteThread
- NtQueueApcThread
- NtMapViewOfSection

(for code injection)
And more special cases

- Return value of 0 is success
- Esoteric API called with bad params then checking error code (they have to chase down each individually, so a path to thwart)
Multiple Paths

- Typically, use symbolic execution
  - SAT solver finds values needed to explore paths
  - Expensive, path explosion
- TAMALES just takes both paths
  - Explodes really bad
Preventing Explosions

Add more paths at each layer. Two thresholds, one to say benign, one malicious.
Feature Extraction for ML

- Entropy of code/data sections
- Discrepancy between checksum header and PE
- Imported functions
- Count of API functions and x86 instructions
- Count of exceptions and types, network connections, strings
- Ratio of API functions imported to called and static vs dynamic strings
More features

• Suspicious
  – Checking for debugger
  – Obfuscation
  – Jumping into middle of API
  – Overwriting header or part of API function
  – Directly accessing OS structures
  – Creating an intentionally infinite loop
Yet more features

• Almost certainly bad
  – Malicious URLs
  – Encrypting/deleting files not created by sample
  – Overwriting Windows DLLs
More n-grams!

- 1-, 2-, 3-, and 4-grams of API calls
- 1-grams of x86 instructions
- 1-, 3-, and 6-grams of informative, suspicious, malicious features
And numeric stuff

- # of internet access attempts
- Number of unique URLs
- Connection attempts with non-standard ports
- Reputation score of target host
Data Cleaning

- Remove features that are always the same
- Scale all to [0,1]
- Feature select using information gain

- Yields 3500 features
Random Forest Classifier

- 1,600 decision trees
  - Max depth 150
  - Up to 200 features per split

<table>
<thead>
<tr>
<th></th>
<th>Benign</th>
<th>Malware</th>
<th>Total</th>
<th>(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training set</td>
<td>113,162</td>
<td>116,807</td>
<td>229,969</td>
<td>(70%)</td>
</tr>
<tr>
<td>Test set</td>
<td>49,254</td>
<td>49,310</td>
<td>98,564</td>
<td>(30%)</td>
</tr>
</tbody>
</table>
Classification

• Two layer funnel
  – Layer 1 – single path, 1 minute timeout
  – Layer 2 – 4 paths, timeout 1 minute

• Set FPR to 0.1%
ROC curve

FPR 0.1%
TPR 99.11%

Table 3: Classification funnel sample count

<table>
<thead>
<tr>
<th>Layer#</th>
<th>Execution paths</th>
<th>Samples</th>
<th>(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>94,845</td>
<td>(96.2%)</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3,719</td>
<td>(3.8%)</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>98,564</td>
<td>(100%)</td>
</tr>
</tbody>
</table>
How does packing go?

- Packed a bunch of benign stuff and saw what TAMALES said

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Packer</th>
<th>Samples</th>
<th>Predicted malware</th>
<th>(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS1</td>
<td>Unpacked</td>
<td>13,000</td>
<td>0</td>
<td>(0%)</td>
</tr>
<tr>
<td>DS2</td>
<td>UPX</td>
<td>12,154</td>
<td>120</td>
<td>(0.98%)</td>
</tr>
<tr>
<td>DS3</td>
<td>VMProtect</td>
<td>11,783</td>
<td>518</td>
<td>(4.39%)</td>
</tr>
<tr>
<td>DS4</td>
<td>Themida</td>
<td>9,592</td>
<td>582</td>
<td>(6.06%)</td>
</tr>
</tbody>
</table>
TAMALES on Malware families

- Used system to classify into families with Decision Trees

Table 5: Malware family classification results of most common families

<table>
<thead>
<tr>
<th>Label #</th>
<th>Malware family</th>
<th>Samples in test set</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>allape</td>
<td>9195</td>
<td>99.98%</td>
</tr>
<tr>
<td>2</td>
<td>dinwod</td>
<td>4575</td>
<td>99.83%</td>
</tr>
<tr>
<td>3</td>
<td>virut</td>
<td>4213</td>
<td>96.14%</td>
</tr>
<tr>
<td>4</td>
<td>browsefox</td>
<td>2380</td>
<td>99.50%</td>
</tr>
<tr>
<td>5</td>
<td>parite</td>
<td>2012</td>
<td>99.90%</td>
</tr>
<tr>
<td>6</td>
<td>ramnit</td>
<td>1823</td>
<td>93.11%</td>
</tr>
<tr>
<td>7</td>
<td>multiplug</td>
<td>1437</td>
<td>99.93%</td>
</tr>
<tr>
<td>8</td>
<td>upatre</td>
<td>1187</td>
<td>98.02%</td>
</tr>
<tr>
<td>9</td>
<td>mira</td>
<td>1138</td>
<td>99.91%</td>
</tr>
<tr>
<td>10</td>
<td>loadmoney</td>
<td>864</td>
<td>98.86%</td>
</tr>
<tr>
<td>11</td>
<td>unknown</td>
<td>742</td>
<td>72.11%</td>
</tr>
<tr>
<td>12</td>
<td>linkular</td>
<td>714</td>
<td>100.00%</td>
</tr>
<tr>
<td>13</td>
<td>linkury</td>
<td>659</td>
<td>99.70%</td>
</tr>
<tr>
<td>14</td>
<td>elex</td>
<td>646</td>
<td>98.48%</td>
</tr>
<tr>
<td>15</td>
<td>onlinegames</td>
<td>511</td>
<td>85.17%</td>
</tr>
<tr>
<td>16</td>
<td>wajam</td>
<td>502</td>
<td>99.80%</td>
</tr>
</tbody>
</table>