Abstract—Obfuscation is rampant in both benign and malicious JavaScript (JS) codes. A JS code obfuscation generates a code that is obscure to the human eye and undetectable to scanners, thereby hindering comprehension and analysis. This transformation significantly affects the performance of network and information security tools, such as Intrusion Detection System (IDS) and anti-virus software. Therefore, accurate detection of JS codes that masquerade as innocuous scripts is vital. The existing deobfuscation methods assume that a specific tool can recover an original JS code entirely. For a multi-layer JS code obfuscation, general tools realize a readable and formatted JS code, but some sections remain encoded. For the detection of such obfuscated codes, this study performs Deobfuscation, Unpacking, and Decoding (DUD-preprocessing) by function redefinition using a JS code formatter, a Virtual Machine (VM), a JS code editor, and a python `int_to_str()` function to facilitate feature learning by the FastText model, a machine learning model. The learned feature vectors are passed to SVM, a classifier model that judges the maliciousness of an obfuscated JS code. The proposed approach is envisioned to provide improved performance in obfuscated malicious JS codes detection. The detection performance improvement is evaluated using the Hynek Petrak's dataset for obfuscated malicious JS codes, the SRILAB, and the Majestic Million service top 10,000 websites dataset for obfuscated benign JS codes. We then compare the performance of the FastText model to Paragraph Vector models on the detection of DUD-preprocessed obfuscated malicious JS codes. Our experimental results show that the proposed DUD-preprocessing for obfuscated JS codes enhances feature learning and provides improved accuracy in the detection of obfuscated malicious JS codes compared to feature learning on regular obfuscated JS codes.

Index terms—Deobfuscation, Unpacking, Decoding, Obfuscated JavaScript, Multi-layer JavaScript Obfuscation, JavaScript-based Attacks, FastText, Machine Learning

ACKNOWLEDGMENT

This research was achieved by the Ministry of Education, Science, Sports, and Culture, Grant-in-Aid for Scientific Research (B) 16H02874 and the Commissioned Research of National Institute of Information and Communications Technology (NICT), Japan.

REFERENCES

Abstract

• Obfuscation generates a JS code that is obscure to the human eyes and undetectable to scanners. JS code obfuscation aims to hinder comprehension and analysis. This transformation significantly affects the performance of network and information security tools, such as Intrusion Detection Systems (IDS) and anti-virus software. Therefore, accurate detection of JS codes that masquerade as innocuous scripts is vital.
• The existing deobfuscation methods for obfuscated malicious JS codes assume that a specific tool can recover an original JS code entirely. General tools realize a readable and formatted JS code, but some sections remain encoded. For detection of such obfuscated codes, this study performs Deobfuscation, Unpacking, and Decoding (DUD-preprocessing) by function redefinition using a JS code formatter, a Virtual Machine (VM), a JS code editor, and a python int_to_str() function to facilitate feature learning by the FastText model. SVM, a classifier model, judges the maliciousness of an obfuscated JS code. The proposed approach is envisioned to provide improved performance in the detection of obfuscated malicious JS codes.

JS code deobfuscation

• Obfuscated JS code analysis and formatting to make it readable again and uncover its true functionality.
• Tools to analyze obfuscated JS code: such as, Dirty Markup, Online JS code beautifier, Dan’s Tools JS code formatter, and JSNice.

Method

Objective – Performance improvement for detection of obfuscated malicious JS codes using FastText.

Steps to deobfuscate, unpack and decode an obfuscated JS code

1. Deobfuscate an obfuscated JS code – using a JS code beautifier, formatter. These tools make JS code look pretty, readable, easier to edit and analyze.
2. unpack a packed JS code – using a Virtual Machine (VM) and a JS code editor:
   - Strip the script tags, JS code = "eval(function(p,a,c,k,e,d){\}(\).\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\..."
   - Replace the eval() function with console.log().
   - Parse the packed JS code, Unpacked JS code = eval("unpacked + JS code/JS code.find(\') + 1 - 1\)

3. Decode an encoded JS code – using an int_to_str() function in python.
   - Implement int_to_str() function in python.
   - Parse to extract the function arguments – using a VM and a JS code editor.

The FastText model:
• Character n-gram vectors represents each word x.
• Scoring function f takes into account a word internal structure.
• Character n-gram for encode with n=3:
  \[ x_{e.n} \] = \text{encode}.
• For a dictionary of size c n-gram vectors, \( G_c \subset \{1, ..., c \} \)

\[
 f(x, c) = \sum_{i=1}^{c} Z_j^c X_k
\]

Where \( Z_j^c \) is the vector representation for each n-gram \( g \) and \( X_k \) is the context.

Performance Evaluation

<table>
<thead>
<tr>
<th>Model</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>FastText</td>
<td>94.27%</td>
<td>95.11%</td>
<td>94.59%</td>
</tr>
<tr>
<td>PV-DBow</td>
<td>94.12%</td>
<td>92.83%</td>
<td>93.31%</td>
</tr>
<tr>
<td>PV-DM</td>
<td>93.11%</td>
<td>93.51%</td>
<td>92.89%</td>
</tr>
</tbody>
</table>
| Deobfuscated JS code Dataset
| FastText   | 99.48%    | 99.31% | 98.73%   |
| PV-DBow    | 98.39%    | 98.41% | 98.01%   |
| PV-DM      | 98.37%    | 98.02% | 98.19%   |

True positive and false positive rate

A hard-to-deobfuscable JS code example

Using a JS code beautifier or formatter

• JS-based attacks frequently use obfuscation to:
  - Camouflage their malicious intentions.
  - Preserve the overall code behavior.
  - Evade detection.
• The FastText model learns better and reliable vector representations for DUD-preprocessed obfuscated malicious JS codes.

```javascript
function hello(name) {
    console.log(‘Hello, ‘ + name);
    hello(‘New user’);
}

var _0x74f5 = [0];

var 0x74f5 = ‘Hello, New’;
hello(0x74f5);
console.log(‘Hello, 0x74f5’);
```