JavaScript Obfuscation and Deobfuscation

Team:

Eddy Jones | eaj5073@psu.edu
Gary Lin | gql5141@psu.edu
Brendan Burbules | bgb135@psu.edu
Penn State Behrend

This document, code, and samples on Github here: https://github.com/ejones44/CMPSC443

Individual Contribution Table

Name	Contribution to the project
Eddy Jones	Abstract, Related Works, Conclusion, Evaluation, System Design, all images, all references
Gary Lin	Introduction, Conclusion
Brendan Burbules	System Design

Abstract

JavaScript Obfuscation is the process of intentionally making JavaScript code more difficult for humans to read. This is done for various reasons, including to copy-protect code and to hide malicious script from detection. Obfuscation is necessary because JavaScript runs on the client-side in user's web browsers on most webpages [13]. Methods include the removal of whitespace and renaming of variables. One portion of code can also be used to generate the rest of the code on execution [1, 2, 3]. Various obfuscators are available including some simple ones for free online [1]. Obfuscation may be detected by software through static or dynamic analysis [4]. Our tool written in Python attempts to detect whether a JavaScript is obfuscated or not. A set of metrics for detection was found that generates an 85% success rate in obfuscator detection. We compared a selection of obfuscators and deobfuscators on the market today in our presentations. A list of metrics was chosen with which to evaluate these products' effectiveness, and applied to our script. Finally, we setup a repository of categorized examples.

Introduction

Obfuscation is a mechanism used to modify source code making it harder to read and understand for humans, but the main objective is to conceal a program's purpose or logic without altering the functionality [1]. This will prevent others from copying or reverse engineering your program. JavaScript is the basically the programming language of the Web, and is used to make the web interactive [13]. JavaScript usually runs in the user's web browser - so the client can access the code that is running and interpret the logic or purpose of the program. This is why we need obfuscation techniques: in order to can hide the JavaScript's purpose. Some obfuscation techniques we will be studying include Data Manipulation, Encoding, and Non-Alphanumeric Obfuscation. Data Obfuscation is simply making strings and integer variables harder to interpret. A string could be changed to the concatenation of several substrings [3]. Encoding obfuscation techniques use customized encoding functions to encode a command and attach decoding functions that decode only on execution [3]. Our objective is to investigate these existing Obfuscation techniques beyond basic understanding. Likewise, we will look into deobfuscation techniques like cloning, static analysis, and dynamic analysis. JavaScript attacks have been reported as one of the top Internet Security threats in recent years [3]. We will primarily focus on how to detect obfuscation techniques using static and dynamic analysis; weighing the pros and cons of each approach. Static analysis is done without executing the program. Dynamic analysis is done while the program is running. This paper will also compare available tools for obfuscation and deobfuscation: by describing the pros and cons using a proposed list of metrics to evaluate the effectiveness of each tool. Some tools that will be researched are Yahoo's YUI compressor, SpiderMonkey, and Firefox's automated JavaScript DeObfuscator. After all of the researching, our team will write a tool in python to try and detect whether a piece of JavaScript code is obfuscated or not. Our approach will probably be a static one, looking for keywords and functions like eval() or document.write. To categorize our obfuscated code samples we will setup a repository and classify them. Overall, the intention of this project is to investigate the existing obfuscation and deobfuscation techniques and to apply what we have learned.

Related Work

JShadObf: A JavaScript Obfuscator Based on Multi-Objective Optimization Algorithms [1]

The motivation of this paper regards using obfuscation to prevent stealing of JavaScript web development. Obfuscation techniques are used to prevent "reverse-engineering" of the script since it executes on the client-side. It proposes 'JSHADOBF', a framework that attempts to select obfuscation technique based on JavaScript code's structure and then conduct the

obfuscation. The article mentions that total prevention of reverse-engineering is impossible [12] and that the goal is to make it as difficult as possible. Google collaboration products, Dropbox, and Doodle are given as examples where JavaScript is important. The Google Closure Tools is given as an example of an obfuscator. Like most obfuscators it begins by removing dead code, removing whitespace, reading program structure, and messing with variable names. Dummy expressions make up a large part of this paper's obfuscation suggestions. For example an if statement might be added with a randomly generated predicate. The authors report that both their method and Google Closure improve jQuery's efficiency when obfuscated. An obvious con of this work would be their own statement that warnings and errors are reported in the obfuscated jQuery library. Google closure has the same issue [14].

Obfuscated Malicious JavaScript Detection using Classification Techniques [2]

This short paper is about detecting and classifying malicious JavaScript. The authors suggest classifiers to accomplish this. This malware is normally obfuscated to avoid detection. First the article talks about the possibility of just disabling JavaScript altogether. Dismissing this idea, they begin crawling the web to develop their classifications. The crawler collects JavaScript from 9 million benign webpages and then 62 malicious scripts. 61 of these were obfuscated. The final portion of the paper evaluated various classifiers that exist. The authors state that all the classifiers performed well. ~90% of the malicious scripts were detected as malicious and 99.7% of the benign were detected as benign. The classifier "NaiveBay" performed best in their tests. They conclude that these machine learning classifiers are highly effective. The biggest drawback to these classifiers is that a small portion of harmless sites will consistently be classified as malicious. The authors cite any site using packed JavaScript as an unfortunate example.

The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study [3]

This short paper is about the popularity of various obfuscation techniques to hide malware. It also looks into the detection effectiveness of 20 anti-virus products and provides suggestions for improvement of them. The author's report that a majority of obfuscated code only uses a single technique. A minority utilize multiple types of obfuscation. The most popular obfuscation they encountered was String data manipulation. The least popular type was randomization of code comments. The authors conclude from their investigation of virus detection programs that obfuscation is very effective in hiding malicious code. Their suggestion is that a future solution must involve both static and dynamic analysis. The positive of this article is its informative investigation of current methods of detection. The drawback is its simplicity and lack of specific suggestions.

JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code [4]

This long paper is a proposal for a static malware detection technique the authors call JStill. It incorporates a small portion of lightweight dynamic analysis. The idea behind this new method is that even obfuscated malicious JavaScript must become somewhat deobfuscated during

execution. JStill will intercept and look at suspicious function calls as the script executes. To us this seems similar to triggering a break-point in a debugger. Downsides are mentioned: the authors report some false-positives occur. They also admit a 5% performance hit to webpages when using JStill. 95.89% of the false positives were caused by obfuscated arguments of eval functions. Interesting side notes include the existence of malicious obfuscation in other languages (VBScript, Jscript) and the difference between minification and obfuscation. JStill seems like a practical and effective tool.

Towards Revealing JavaScript Program Intents using Abstract Interpretation [5]

This paper discusses the vulnerability created by the usage of AJAX based web applications. It suggests an abstract interpretation of JavaScript code in order to provide protection. Specific attack methods are detailed including XSS worms, XSS botnets, and drive-by download attacks. The suggestion this paper gives requires that the user have some context for the suspect code. It also assumes that the server is not trustworthy. Basically, a proxy will first checkout the code on the client's behalf. The JavaScript is partially executed by the proxy and then the client opens the contextualized code in a tree. This should help with the detection of obfuscated malware. The big drawback of this paper is that their suggestion is all theory. They plan to attempt an implementation in the future.

Automatic Detection for JavaScript Obfuscation Attacks in Web Pages through String Pattern Analysis [6]

This paper is a proposal of metrics to use in detecting obfuscated malicious strings in JavaScript code. These three metrics are N-gram, Entropy, and Word Size. These refer to how much each byte code is used in a string, the distribution of used bytecodes, and whether there are very long strings, respectively. Their method begins by detection the potentially dangerous functions. They then extract all the strings related to said functions and weigh them using their metrics. To us, the negative from this paper appears to be its limited scope. These three metrics do work however: the authors concluded that their method was relatively effective in picking malicious strings. However, they claim to be developing more metrics for the future.

An Empirical Study of Metric-Based Methods to Detect Obfuscated Code [7]

This study is a more detailed look at the same three metrics from [6]. The difference being that this paper is less optimistic. They conclude that Word Size is very effective and a good indicator of obfuscation. However the other two metrics are less useful. Many false positives and (even worse) true negatives were found when isolating these metrics. They also determined that some obfuscators are capable of defeating the clever use of these three indicators. Obfuscators vary in effectiveness. The positive to this source is that it includes more depth than [6] but it comes to the same basic conclusion. The three metrics: N-gram, Entropy, and Word Size are relatively effective when used together.

Obfuscated Malicious JavaScript Detection by Causal Relations Finding [8]

This paper is another proposal for a method of detecting malicious obfuscated JavaScript. It suggest a method called Obfuscating Causal Relations Finding. This method neglects deobfuscating the code and works on the disguised code directly instead. Its goal is to have minimal false positives and to be implemented in a browser extension. The drawback of this paper is that the authors do not actually do this. They simply repeat that it could easily be done. Their method works by dividing the code's function calls into six token categories. These are: JS, Hex, Uni, Oct, Susp, and Char. These different tokens go into their algorithm for classification. This article notably only uses 50 malicious samples and 206 benign samples for their experiment. This small sample size (compared to millions of samples in [2]!) is another obvious problem with the research. They found the tested approach to be reasonably effective.

RedJsod : A readable JavaScript obfuscation detector using Semantic-based Analysis [9]

This paper is unique in that it hopes to detect obfuscation in "readable" code. This code is clearly not attempting to prevent copying since it tries to appear unobfuscated! It is much harder to detect such code since it will not have modified strings and will look very benign to the eye. Their suggestion is called RedJsod. The detection rate of these author's method is reported as above 97%. Interestingly, they claim that to their knowledge, no one has written about obfuscation hidden in readable code. Their investigation finds that malicious JavaScript is more frequently readable-obfuscated than completely obfuscated. A competing detection scheme is Nofus. RedJsod hopes to use variable context-level feature extraction to detect malicious code. Similar to [5] the code will be partially executed to determine context, then classified once tokens are classified. This paper is very interesting because of its unprecedented discussion of obfuscated malicious code being hidden in *nonobfuscated* code.

Automatic Simplification of Obfuscated JavaScript Code: A Semantics-Based Approach [10]

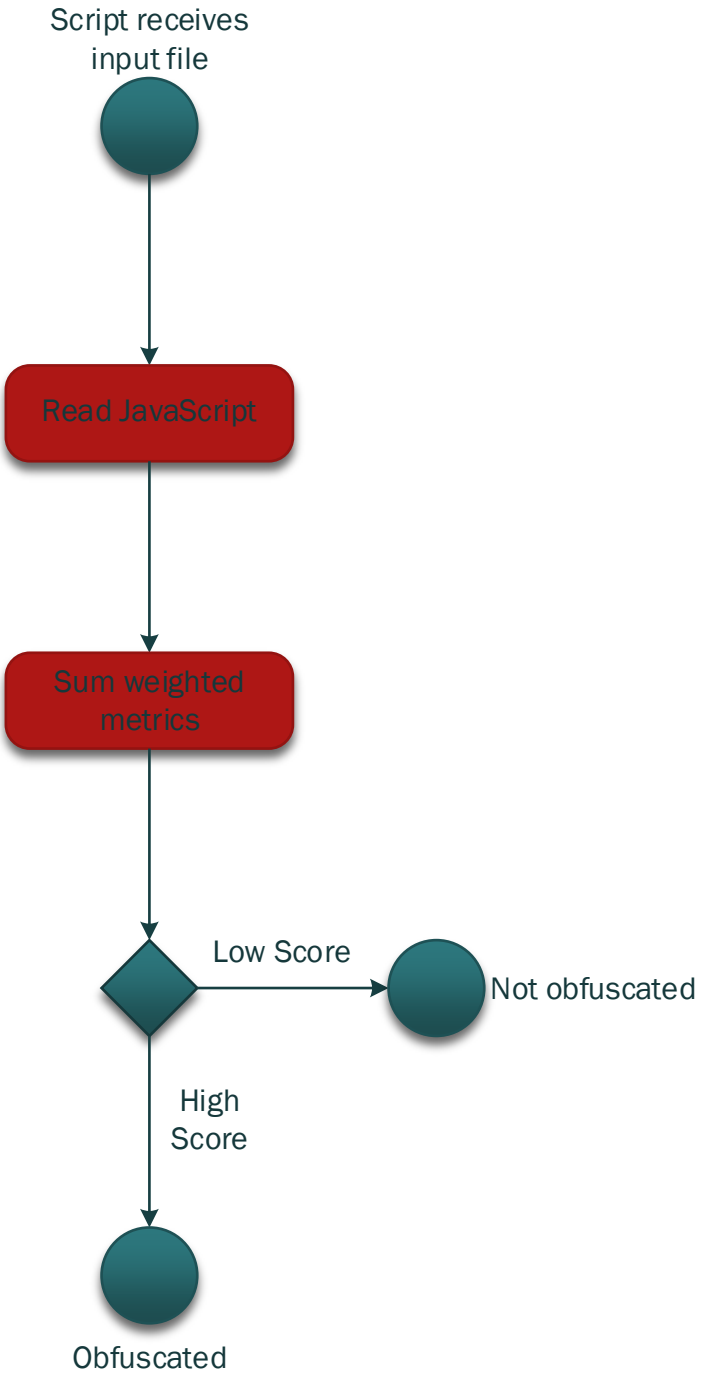
The first part of this paper defines and discusses drive-by download attacks and the Gumblar worm. The Gumblar worm refers to JavaScript containing obfuscated malicious code that is dynamically generated on execution. The paper suggests the solution of semantics-based detection. This refers to identifying system calls made by the code and the order in which they are called. Dynamic slicing is used at the bytecode level to identify the instructions that actually affect the values of native function arguments. Supposedly this helps remove functions that are in place to confuse human readers. The entire process takes five steps. The original code is opened in a safe browser, the control flow is analyzed, the deobfuscation slicing occurs, decompilation occurs, and finally the code is transformed into simplified (hopefully readable) JavaScript code. Nicely, the paper includes a detailed breakdown of code examples. This paper succeeds in demonstrating the viability of a semantics based approach to scanning JavaScript code. The downside is that no real implementation has been done.

Suspicious Malicious Web Site Detection with Strength Analysis of a JavaScript Obfuscation [11]

This source begins with a breakdown of obfuscation into four types: those that use ASCII and Unicode values, those that use XOR operations, those that split Strings, and those that compress a string and replace it with a meaningless one. Like our other sources it discusses the drive-by download attack. The paper argues that assessing obfuscation's strength would be useful. After mentioning the metrics of N-Gram, entropy, and word size it suggests measuring strength of the obfuscation via different criteria. These are string lengths, density, frequency of functions, frequency of special characters, and entropy value. The authors conclude that hybrid detection approaches are best. The bad news is that they require more studies that do not rely on analysis of the actual JavaScript to detect malware.

System Design

Our software design will consists of a Python script using a static approach. It attempts to output detection and classification of obfuscation with JavaScript inputs.



Obfuscation Detection Script 'Detector.py'

When run, the program will open all JavaScript files in the current directory and read them in. The program reads each file line by line and analyzes that line for keywords and other evidence that suggest obfuscation techniques. The program will then generate a score based on how many flags, or obfuscation keywords, and the length of the program.

The first step of generating this score is the detection of keywords that suggest obfuscation. These keywords are “document.write, eval, toCharCode, unescape, ‘p,a,c,k,e,d’, and utf8to16”. These keywords represent functions commonly used to decode and evaluate obfuscated code. In the case of ‘p,a,c,k,e,d’, this is commonly found as function arguments in several JavaScript obfuscators. For each one of these keywords found, a ‘detections’ counter is incremented.

After looking for the keywords, the program will then find and analyze function and variable names. This process is done by finding the ‘var’ and ‘function’ keywords where variables and functions are declared. The variable name is terminated by semicolon or comma, and a function is terminated by a ‘(‘ character. The string between the declaration keyword and the terminator will be evaluated as a name.

Once the program has function and variable names, it looks for signs of obfuscation. One way of looking for obfuscated names is by comparing the amount of consonants to the amount of nouns in a name. The threshold for this metric is (consonants/vowels > 3). The program will also analyze the amount of numeric and special characters in a name. If a name contains more than 4 numbers or special characters, it will be flagged as obfuscated. Each name that the program considers obfuscated will increment the ‘detections’ counter in the program.

For scoring, the program will consider a sample obfuscated if (detections/lines_read) > 0.2.

Evaluation

Our initial research into the field involved comparing different obfuscators available on the market. First one must decide on the metrics that determine a good obfuscator. The first and most obvious metric is simply whether or not your code works after obfuscation! Some obfuscators are unable to maintain functionality when certain JavaScript constructs are used. The second most obvious metric is the resulting file size – smaller being better. Another (minor) metric is how long the process of obfuscation takes. We played around with a few different obfuscators and took note of our opinions.

We found this table with some comparison of speeds and output file sizes:

File	UglifyJS	UglifyJS +gzip	Closure	Closure+gzip	YUI	YUI+gzip
jquery-1.6.2.js	91001 (0:01.59)	31896	90678 (0:07.40)	31979	101527 (0:01.82)	34646
paper.js	142023 (0:01.65)	43334	134301 (0:07.42)	42495	173383 (0:01.58)	48785
prototype.js	88544 (0:01.09)	26680	86955 (0:06.97)	26326	92130 (0:00.79)	28624
thelib-full.js (DynarchLIB)	251939 (0:02.55)	72535	249911 (0:09.05)	72696	258869 (0:01.94)	76584

Speed and Size of Various Obfuscators [15]

Taking into account the differences between various obfuscators we set about attempting to detect their usage as the goal of this project. Whilst evaluating our Python detection script 'detector.py': false positives, true negatives, and performance were the primary metrics used to measure script performance.

```
Obfuscated files read: 20
Unobfuscated files read: 10
Files detected: 17
Detection percentage: 85.000000%
False positives: 1
Missed detections 3

Process finished with exit code 0
```

Script Output Summary

Our script through multiple iterations was able to establish an 85% detection success rate. Each sample in the folder is read in turn and a score is summed for identification as discussed in system design.

```
Opening  sample 8
Obfuscator -  Base62 encoding http://dean.edwards.name/packer/
==============
DETECTIONS - 12
Lines Read - 3
Score: 4.000000
I think this file is obfuscated!
```

```
Opening Sample 9
Obfuscator - Variable shrinkage
==============
DETECTIONS - 7
Lines Read - 4
Score: 1.750000
I think this file is obfuscated!
```

```
Opening scenario
Obfuscator - none
==============
DETECTIONS - 2
Lines Read - 194
Score: 0.010309
```

```
Opening sequence
Obfuscator - none
==============
DETECTIONS - 1
Lines Read - 112
Score: 0.008929
```

Individual Sample Outputs

The figure above shows some of the individual outputs for samples. Each of the four is a JavaScript file being considered for obfuscation detection. As you can see, the top two are correctly determined to be obfuscated. The last two (JavaScripts from our senior design project which are not obfuscated) do not get flagged. The decision whether to flag a script as obfuscated is discussed in greater detail in system design.

Conclusion

As our study of JavaScript obfuscation and deobfucscation techniques concluded we discovered a great deal of knowledge on this subject. The main goal of obfuscation is to lower the readability of client-side code. Motivations for obfuscation include preventing others from copying-pasting a web design, hiding malicious code, and allowing client-side calculations or solutions without their being easily revealed to a user. Not all JavaScript obfuscation is for nefarious purposes. We found that JavaScript obfuscation is not as difficult as we previously believed - many obfuscation tools are available on the web. It is possible to easily paste the code and click a button to obfuscate your code, but using free open source obfuscators could be risky. The most effective obfuscation options cost money online. These are extremely effective.

As far as detecting obfuscation, we concluded that a static approach to detection would be most realistic, as dynamic analysis is very complicated. Our python script 'detector.py' generates a score based on finding suspicious strings, patterns, and file sizes. Our conclusion is that detection is not impossible but the success entirely depends on metrics used. The included resources frequently detail similar approaches in their detection attempts: selection of metrics is the real challenge in detection with a high success rate. If we were to continue work in this field experiments would involve using different metrics over larger samples.

This project greatly increased our knowledge and interest in the field of obfuscation. It was fascinating to see how quickly our attempts at detection aligned with those of scientists and experts from our references. JavaScript obfuscation will only increase in relevance as we move towards a more web-application based infrastructure.

References

[1] Bertholon 2013 JShadObf: A JavaScript Obfuscator Based on MOEAs. *NSS 2013,* pp. 9.
[2] Likarish 2009 Obfuscated Malicious JavaScript Detection using Classification Techniques. *IEEE 2009* pp. 47.
[3] Xu 2012 The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study. *IEEE 2012,* pp. 9.
[4] Xu 2013 JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code. *CODASPY'13,* pp. 117.
[5] Blanc 2010 Towards Revealing JavaScript Program Intents using Abstract Interpretation. *AINTEC'10,* pp. 87.
[6] Choi 2009 Automatic Detection for JavaScript Obfuscation Attacks in Web Pages through String Pattern Analysis. *FGIT 2009,* pp. 160.
[7] Visaggio 2013 An Empirical Study of Metric-Based Methods to Detect Obfuscated Code. *International Journal of Security and Its Applications Vol. 7, No. 2,* pp. 59.
[8] AL-Taharwa 2011 Obfuscated Malicious JavaScript Detection by Causal Relations Finding. *ICACT2011,* pp. 787.
[9] AL-Taharwa 2012 RedJsod : A readable JavaScript obfuscation detector using Semantic-based Analysis. *2012 IEEE,* pp. 1370.

[10] Lu 2012 Automatic Simplification of Obfuscated JavaScript Code: A Semantics-Based Approach. *IEEE 2012,* pp. 31.
[11] Kim 2011 Suspicious Malicious Web Site Detection with Strength Analysis of a JavaScript Obfuscation. *International Journal of Advanced Science and Technology Vol 26, January 2011,* pp. 19.
[12] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (Im)possibility of obfuscating programs. In: Kilian, J. (ed.) CRYPTO2001. LNCS, vol. 2139, pp. 1–18. Springer, Heidelberg (2001)
[13] Flanagan, D.: JavaScript: The Definitive Guide Activate Your Web Pages, 6th edn. O'Reilly Media, Inc. (2011)
[14] Yank, Kevin. "Google Closure: How Not to Write JavaScript." Sitepoint. N.p., 12 Nov. 2009. Web. 01 Mar. 2016.
[15] Comparison of Obfuscator Speeds by Mihai Bazon. https://github.com/mishoo/UglifyJS