An Analysis of Obfuscated Malicious JavaScript

Nicholas Whitefield
University of Alberta
Canada
whitefie@ualberta.ca

Jorden Hansen
University of Alberta
Canada
jorden1@ualberta.ca

Abstract

JavaScript's popularity has grown rapidly in recent years, showing up in all areas of web development (TODO SBNV). With the rise of JavaScript, there is also an increase in malicious JavaScript code. (TODO SBNV) We aim to identify some weaknesses of current malware detection tools, as well as identify how different methods of obfuscation perform on these tools. By conducting a survey on a sample of 1000 malicious JavaScript files, we observe how the obfuscation methods perform. We selected 6 different obfuscation methods, which were applied each individually as well as all possible pairings, totalling 21000 obfuscated JavaScript files. After removing tools we determined to be too ineffective at detecting malicious files, we were left with 18 tools which we used to analyse the obfuscated files.

Keywords JavaScript, Obfuscation, Malware, Program Analysis

1 Introduction

Malware is becoming more and more of a societal issue, with the rise of ransomware attacks and large scale data breaches, there is a larger need for better methods of malware detection. One of the main challenges with malware analysis is that malware can be modified to be intentionally difficult to analyse due to obfuscation (TODO SBNV). Therefore, it is necessary for malware detection tools to be able to see through obfuscation attempts. In this paper we explore six different types of obfuscation methods and how they impact the detection rates among a variety of different analysis tools. In our research, we focus exclusively on malicious JavaScript files, we do this for two main reasons: The first being that it was the most abundant form of malware we found (TODO maybe talk about how javascript is everywhere so its important research idk.). We didn't want to have source code from different languages, since this could have an effect on detection rate. Our second reason was that there is a large amount of tools and support around javascript. This allowed us to focus more on getting results and less on building the tools required to create our dataset.

1.1 Obfuscation

Obfuscation is the process of converting code into a more convoluted version of itself, without introducing any changes to the intended behaviour. From the perspective of an attacker, obfuscation is a useful technique as it weakens a victim's ability to understand and analyse the malicious program. With this in mind, techniques have been proposed that aim to detect malware by first detecting obfuscation and using that as a measure of malice (Blanc 346). (TODO Maybe just say that this cannot be implemented naively because then you would get over approx.) However such techniques are an over-approximation as they consider all obfuscated programs to be malicious (TODO not true (by above)), when this is not necessarily the case; there are also valid non-malicious use cases for obfuscation for both academia and industry, obfuscation applied with this intent is referred to as 'benign obfuscation' (Zhang). A notable application for benign obfuscation is that it can be used to prevent thieves from reverse engineering proprietary algorithms (colberg 1); such algorithms can be very costly to research and develop and difficult to defend in court (colberg 1).

2 Implementation

2.1 Tools

There are a number of tools that we used to create and gather our results. In this section we will discuss these tools. For the obfuscation of our dataset, we used the 'javascript-obfuscator" which we link to below (TODO sounds odd, REFERENCE). This is an open source tool that provides a command line interface for applying various obfuscation methods to javascript files. The obfuscator supports 18 obfuscation methods that can be applied, however, some of these methods are more tailored towards benign obfuscation which we eliminated from consideration. Seeing as one of the areas we were interested in exploring was how these methods interact with one another, we decided to reduce the set of methods further to decrease the number of potential method combinations to a manageable size. In total, we selected six methods which we choose based on our rough predictions of their effectiveness. These six methods, along with all 15 possible pair combinations of these options were then applied to our dataset and analysed. These methods are highlighted below.

- *Control Flow Flattening*: This obfuscation method "flattens" the control flow of a program, making it harder to analyze the code. We used the default value of 0.75 for the obfuscation tool.
- *Dead Code Insertion*: This obfuscation method adds additional lines of code to the file in an attempt to make it harder to follow the actual code of the file. We

Table 1. Obfuscation with Transform Object Keys

```
// Input
var object = {
    foo: 'test1',
    bar: {
        baz: 'test2'
    }
};
```

```
// Output
var _0x5a21 = [
    'foo',
    'test1',
    'bar',
    'baz',
    'test2'
];
```

used the default value of 0.4 for the obfuscation tool. It is important to realize that dead code insertion does increase the size of the code. If it is important that the malware file stay small in size, this may not be the ideal obfuscation method.

- *Identifier names generation*: This method changes the identifiers in the code. It provides two options, hex and mangled.The hex option changes identifiers to be hex values, and mangled uses short values for names.
- *Self-Defending*: This option makes it so that the code is resilient to change. Any attempt to rename or change the code structure should result in the code not running. This can make it much harder to analyze code since it cannot be reformatted in anyway.
- *String Array Encoding*: This obfuscation method encodes string literals in the code to be base64 or rc4. In our case we used base64. This decision was due to the fact that using rc4 would lead to a greater slow down in execution time over base64. We don't want to change runtime too much for the malware.
- *Transform object keys*: This method changes the way that objects in the code are stored. An example is provided below:

2.2 Process

For the analysis of our dataset, we used VirusTotal's virus scan API which allowed us to upload files and receive a scan report which contained results from all the tools VirusTotal used in the scan (TODO link to API). In total, there are over 70 (TODO FC) tools that VirusTotal can run on submitted files; however, each scan does not necessarily use the same set of tools. Upon initial analysis, we found that many of these tools had considerably poor detection rates for even before applying obfuscation to our dataset (TODO PI, this whole area sounds off, also we should maybe reference a figure here? filter figure?). We removed these tools with a first pass at filtering. We then saw that some tools that performed well on the initial dataset performed poorly when any single obfuscation method was applied. We believe that this is likely due to those tools either not being made with javaScript analysis in mind, or just being weak tools in general. Due to this, we decided to filter out these weak tools. One of the main benefits of VirusTotal was that it enabled us to automate the process of submitting files and retrieving results, allowing us to use a larger dataset of malware. By using VirusTotal, we were also able to abstract away from any particular tool which also allowed us to gather data across a large set of tools.

In order to ensure that the obfuscation methods being applied were not simply destroying the original javascript source files, we used the javascript 'syntax-error' library which checked for any syntax errors in our obfuscated files (TODO link to library here). The library itself has over 400k weekly downloads on npm and is still being maintained, so we assume its results are reliable. Throughout our obfuscation process, we did not observe a single occurrence of invalid-syntax, so we can also reasonably assume that the obfuscation tool does not output syntactically invalid javascript.

The malicious javascript files we used were taken from a GitHub repository (TODO link here?). The repository provides over 40 000 malware samples, but since each obfuscation method creates a new file that we must manage, we decided to only use the files provided in the 2015 folder, totaling to 1000 files. Our obfuscated files dataset along with the original samples are available in our project GitHub repository (TODO: LINK), as well as the raw results of the VirusTotal scans and our code for analysing these results.

The general workflow of our process is given in figure (TODO ref figure). For each original file in our dataset, we applied an obfuscation technique which produced an obfuscated file. The file would then be checked for syntactic validity before being uploaded to VirusTotal. In our experiments, all obfuscated files passed this test and followed the true branch. The file would then be uploaded to VirusTotal and the scan results would be downloaded at a later time. Once all the scan results for the batch of 1000 files was returned, we would run our analysis scripts on the batch of results.

3 Evaluation

We have two research questions that we set out to answer (TODO PI should flesh this out a bit more, refer to other papers (COVA?))

3.1 RQ1: What is the effectiveness of each obfuscation technique?

As we briefly mentioned earlier, when we scanned the original (unobfuscated) files through VirusTotal we found that many of the tools performed inadequately on our initial dataset. These tools either had extremely low detection rates and in some cases failed to detect even a single file. It is very unlikely that a tool will perform better once we obfuscate the files, therefore we decided that these tools should be removed from our toolset as they add very little value to our

results. We filtered out any tools that had a detection rate less than 20%.

One thing we noticed is that many tools that performed well on the unobfuscated dataset performed exceptionally poor once any obfuscation was added, we later also filtered out these tools as well, in a second round of filtering. To choose what tools we kept, we created a set of tools that had an accuracy over 20% for each single method of obfuscation. We then ranked the tools by counting how many times it appeared in each set.

In the end we were left with a set of 18 tools. We decided it would be best to look at this smaller list of tools since it would provide us with more meaningful results than using all the tools. Interestingly, after scoring every tool and choosing the ones with a score of 5 (of 7) or higher.

After analyzing the results, we know that control flow flattening is by far the most effective single obfuscation method. It was detected by the tools 37.2% of the time, the next closest is string array encoding at a detection rate of 67.98%. The least effective method was dead code insertion at 81.87%. This still gives us a detection rate better than the original files without obfuscation which was 97.13%.

When we use two obfuscation methods, the best two methods are using both control flow flattening and string array encoding. This isn't surprising as these were the top two methods individually. One interesting thing is when control flow flattening is paired with dead code insertion, it performs better than control flow flattening paired with transform object keys, which has detection rates at 35.31% and 36.37% respectively (TODO we should elaborate more why this is interesting). Our worst two method obfuscation was when we combined dead code insertion with transform object keys, which performed slightly better than each does on their own, at 81.06%.

3.2 RQ2: How well do the tools perform individually and how do they compare?

Our best performing tool on single method obfuscation was Kaspersky which had an average detection rate of 96.26%. The next best tool was ZoneAlarm which had a detection rate of 96.25%, putting it very close to Kaspersky. The worst tool we had was McAfee-GW-Edition with an average detection rate of 41.38%. The majority of the tools we analysed performed the worst when control flow flattening was used to obfuscate. The mean detection rate across tools is 73.76%

What is interesting though is that our three best tools (Kaspersky, ZoneAlarm, and NANO-Antivirus) all perform the worst when identifier name generator is used as an obfuscation method. They still perform better than other tools, but this was the only method which had a consistent decrease in the tools (A decrease of 6%). A few tools, Qihoo-360, Ikarus, Microsoft, and McAfee-GW-Edition all perform the worst when string array encoding is used. Symantec performs very well as a tool, placing 4th, even though it cannot detect any malware that is obfuscated with self-defending. It is likely that Symantec tries to alter the code in some way, triggering the self-defending, which then makes the tool fail to detect any malware. If it wasn't for this, Symantec would likely have a detection rate very comparable to NANO. What we can see is that some tools perform worse on certain obfuscation techniques, while others have little to no effect from that method.

Moving onto cases where two obfuscation methods are used, We see that ZoneAlarm and Kaspersky actually have the same average detection rate, at 95.04%. The worst tool is again McAfee-GW-Edition, at a mean detection rate of 18.63%. The average across all tools is 58.37%, this is dragged down mostly by control flow flattening with any other method, where the majority of the tools perform rather poorly. A few interesting results showed up here though. Ikarus has a detection rate of 25.00% when string array encoding is used with self-defending, which is exceptionally high when compared to other obfuscation methods that also used string array encoding (all others falling below 5%). Even just string array encoding alone has a lower detection rate than when paired with self-defending. TrendMicro performs exceptionally bad when self-defending and identifier name generator are used together, scoring 0.30%, its next lowest detection rate is 18.88% on control flow flattening with string array encoding. There must be something interesting going on with this interaction because individually it scores 51.8% on self-defending and 76.3% on identifier name generator. McAfee-GW-Edition scores a 0% on code obfuscated with control flow flattening and string array encoding, this is only interesting because the only other case of a detection rate of 0% is with Symantec and self-defending code, which it likely cannot handle. In this case, it isn't that McAfee can't handle this type of obfuscation, as it can detect in both cases individually, it is the combination of these methods that gives it a detection rate of 0%.

We can see that our top ranked tools are rather resistant to most obfuscation methods, never dropping below 90% for Kaspersky and ZoneAlarm on any obfuscation method, and only having a noticeable dip when identifier name generator is used.

4 Limitations

The results of our study have limitations due to a number of assumptions that we made throughout the study:

- **Is the Malware Malware?**: The repository of malicious javascript that we used supposedly contains only malicious javascript files, but without manual inspection of each file it would be very difficult to verify this, however, we are confident that at least 98% of the files are malicious based on the scan results from VirusTotal.

- **Code Destruction**: We provide a safety mechanism to detect if the obfuscator ever 'destroys' the input files by using a syntax checking library, but this does not account for potential program-behaviour-modification by the obfuscator.
- **Randomness**: Since the obfuscator tool used some amount of randomness when performing obfuscation, the small differences in some of our results could simply be due to this randomness. This could be solved by seeding the obfuscation tool, which we only realized after collecting all our results.
- **Verifiability**: While most of our process is automated and can be replicated, we cannot guarantee identical results due to our inherent dependency on VirusTotal. The tools that VirusTotal uses to generate its scans are likely updated often, also VirusTotal itself likely includes and removes tools from its toolset. While we do not think this would drastically alter our results, it could lead to some deviation in accuracy for anyone replicating our study.

5 Related Work

5.1 The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study (Xu, Zhang)

A paper released in 2012 by Xu, Zhang, and Zhu, is a similar study to our own, however they do not use the same tools and obfuscation methods that we do. They use 4 obfuscation methods: randomization, data (TODO OAJ), encoding, and logic structure obfuscation. They introduce randomization by applying white space, comments, and renaming variables and functions. This is similar to the identifier name generator that we use. Their data obfuscation is string computations, keyword substitution and number computations; we do not have a comparable obfuscation technique to this one. The last obfuscation technique they test is encoding, which is similar to our string array encoding, however they only use hexadecimal, ASCII, or unicode, whereas the method we used is base64 encoded. They mention logic structure obfuscation, but provide no results for it.

5.2 Detection using Machine Learning:

Research has also been done in looking at the application of machine learning to detect obfuscated Malicious JavaScript (Likarish). In this paper, the authors propose a classification based method for detecting the presence of obfuscation itself, which they consider to be a strong indicator for malware. A model using the classification based approach is advantageous because it can still detect malicious JavaScript even if it has never seen that particular instance before (Likarish). They find that classification is a viable method for building accurate detection tools, however only if the feature vectors are carefully constructed. They also highlight some drawbacks with this method, most notably that the classifier can be susceptible to false positives. They identify packing as being the most likely obfuscation technique to create a false positive, even though packing is used often by server to compress JavaScript before sending to users (Likarish). Another drawback mentioned is that any attacker who knows the feature vector being used by the model can simply modify their code to less exemplify those features, in which case the model would need to be retrained or new features would need to be identified (Likarish).

6 Future Work

There are a few areas of future work for this project. One would be applying more obfuscation methods and seeing how they interact. We chose 6 methods and applied them individually then all combinations involving two of these methods together. It would be interesting to see results from applying 3, 4, or more obfuscation methods. There could also be further work done in seeing how well dead code insertion and control flow flattening do when the threshold values are changed for them. Is our value for dead code insertion too low to make it effective? Does it become effective at higher values?

Work could be done to try obfuscation on code written in languages other than JavaScript, which is all we focus on.

Further work could also be done in automation. Right now most parts of the project are automated, but they don't interact with each other in many cases.

There is further work that could also be done in exploring patterns in the table, we didn't check individual files. It would be interesting to see if all the files detected in low amounts are the same files (this is likely), as well as looking more into the tools and see if we can find a reason for some of the outliers in the data we gathered. It is also possible that some files are never detect, however we again do not individually check any files. It could be possible that there are certain characteristics that these files share that make them harder to detect.

7 Conclusion

Generally speaking, the tools fall into three categories. Category one is the tools that perform well across all obfuscation results, this is Kaspersky, ZoneAlarm, and NANO. Category two is tools that perform well across all obfuscation results except control flow flattening, this is the biggest group. Category 3 is the tools that have the most trouble with string array encoding.

The best obfuscation method is control flow flattening and the worst is dead code insertion, followed closely by transformation of object keys. The data collected points out some of the weaknesses in detection software, showing where some need improvement, but also highlighting what each tools

handle well. However, this also shows that if a malicious party knows what type of detection software they are up against, they can meaningfully pick an obfuscation method targeted at that tool.

It isn't surprising that the largest amount of tools struggle with control flow flattening, as it is the most effective method for obfuscation. What is surprising is there isn't much middle ground. Most tools either have single digit accuracy, or are above 80%

testing citing Likarish [2], Xu [3], Blanc [1]

References

[1] Gregory Blanc, Daisuke Miyamoto, Mitsuaki Akiyama, and Youki Kadobayashi. 2012. Characterizing Obfuscated JavaScript Using Abstract Syntax Trees: Experimenting with Malicious Scripts. In *2012 26th International Conference on Advanced Information Networking and Applications Workshops*. IEEE, Fukuoka, Japan, 344–351. https://doi.org/10.1109/WAINA.2012.140

[2] Peter Likarish, Eunjin (EJ) Jung, and Insoon Jo. 2009. Obfuscated Malicious Javascript Detection using Classification Techniques. In *2009 4th Int. Conf. on Malicious and Unwanted Software (MALWARE)*. IEEE, Montreal, QC, Canada, 47–54. https://doi.org/10.1109/MALWARE.2009.5403020

[3] Wei Xu, Fangfang Zhang, and Sencun Zhu. 2012. The power of obfuscation techniques in malicious JavaScript code: A measurement study. In *2012 7th Int. Conf. on Malicious and Unwanted Software*. IEEE, Fajardo, PR, USA, 9–16. https://doi.org/10.1109/MALWARE.2012.6461002