Malicious JavaScript detection using machine learning

Olof Mogren
mogren@chalmers.se

April 24, 2017

Abstract

JavaScript has become a ubiquitous Web technology that enables interactive and dynamic Web sites. The widespread adoption, along with some of its properties allowing authors to easily obfuscate their code, make JavaScript an interesting venue for malware authors. In this survey paper, we discuss some of the difficulties in dealing with malicious JavaScript code, and go through some recent approaches to detect and classify malicious JavaScript code statically using machine learning methods [10, 11, 12].

1 Introduction

Malicious JavaScript is code that shows some kind of malicious unwanted behaviour, such as drive-by downloading, installation of other malware (e.g. fake codecs), unwanted advertisements, or spam. The code is often obfuscated, making static code analysis and detection difficult. However, this provides an interesting research problem, and some work has been done to detect obfuscated code, a fairly easy task for the human eye, but difficult for a computer program (See Figure 1). Obfuscation may include using `document.write()`, and `eval()`, the usage of strings encoded in nonstandard character encodings, and removal of white-space. Obfuscation techniques have become increasingly sophisticated, which puts a higher demand on detection or deobfuscation techniques.

Generally, JavaScript code is considered malicious if it produces unwanted effects for the user (victim), even though it uses API calls that are technically allowed, and not necessarily exploits vulnerabilities in the client systems [8].

Malicious JavaScript code is often used as a stepping stone for other malware attacks, tricking a user to install other kinds of malicious software, or to directly install and execute exploits.

Some approaches for malicious JavaScript detection use dynamical code analysis, such as client honeypot techniques [9, 2], or statical analysis such as pattern matching [3].

Figure 1: Obfuscated JavaScript code (left). Non-obfuscated JavaScript code (right). Illustration from Likarish et.al.

Maintaining pattern-based systems can become a tedious task as new malicious scripts are published, creating a moving target, and using dynamical code analysis is typically computationally expensive. Some services, including *Google Safe Browsing*, maintains a black list of URLs with malicious content of some sort, and yet other approaches uses code signatures for detection. The black-list approach can provide a certain level of security, and is currently implemented in web browsers such as Firefox and Chrome, but new URLs containing malicious content need to be added to the lists before users are protected. The signature based systems can be fooled by dynamical obfuscation strategies.

A number of solutions employ a machine learning-base approach, some recent examples include

- Zozzle [5], an approach using engineered hierarchical features from the abstract syntax tree of the JavaScript code. The features are classified using Bayesian classification.
- Huda, et.al. [7] presented an approach using a hybrid of SVMs and Maximum-Relevance–Minimum-Redundancy Filter heuristics.
- Aebersold, et.al. [1] presented an approach using feature engineering and a machine learning classifier to detect obfuscated scripts.

Background

Machine learning is the research field concerning algorithms that learn a specific task from data, rather than being explicitly programmed. Traditional algorithms, such as support vector machines (SVM), k-Nearest Neighbours (kNN), Decision Trees, Random Forests, and Logistic Regression, typically rely on feature engineering, a semi-manual task of deciding properties of the data that can be extracted and fed to an algorithm as input, from which the algorithm learn to produce the

right output. Once the algorithm has been trained, it can be used to make predictions about unseen data. While machine learning methods can be used to learn most kind of functions, one of the most classical uses is classification into one or more classes (or labels) for each input data instance; this is also the setting we will consider in this paper.

Deep learning is a term coined around 2006, when pretraining strategies and faster computers allowed researchers to train deeper artificial neural network (ANN) models [6]. ANNs (a class of machine learning algorithms) have been around for decades, but since the depth revolution, they have seen great successes in a number of fields. An ANN is a model that is able to approximate virtually any continuous function, and is usually trained using input/output pairs from the generating distribution of interest. Internally, an ANN is structured with layers, each taking a vector of inputs and giving a vector of outputs; hence the whole network can be seen as a composition of a number of (simple) functions. In each layer, the input vector is multiplied by a weight matrix (performing a linear transformation), and the result is "squashed" through a nonlinear element-wise function, such as the *sigmoid*, *tanh*, or *ReLU*. As each layer transforms the data, the depth of the models allows them to learn internal (vector valued) representations of multiple hierarchies; experiments have shown that the first layers learn simple features, which are then transformed to more complex and abstract concepts deeper in the model. The fact that these models automatically learns the necessary features of the data is one of their major strengths; along with the fact that they can learn nonlinear decision boundaries, in contrast to many traditional methods such as SVMs, kNNs, and Logistic Regression. In the training stage, a loss function is minimized by computing the gradients with regards to all the weights in the model, which are then updated with some flavour of gradient descent.

Paper 1 (background): Obfuscated malicious JavaScript detection using classification techniques

Likarish, et.al. [10] showed that machine learning algorithms can successfully classify JavaScript code as malicious or benign with high accuracy. The approiach uses feature engineering together with (traditional, shallow) machine learning classifiers to detect obfuscation and maliciousness. The authors create a dataset by crawling web sites from black-listed URLs (candidates for malicious code), as well as URLs from Alexa's top web sites list (candidates for benign code). They manually inspect each downloaded script and label them as malicious or benign. The result is a dataset of 50.000 benign scripts, and 62 malicious scripts. Their feature engineering work included identifying a human-readability score, by counting certain readable tokens, computing the frequencey of every JavaScript keyword, counting the length of the script, the average number of characters on each line, number of

unicode symbols, hexadecimal numbers, fraction of whitespace characters, etc. In total, 65 features were extracted about each script. Then they trained (linear) classifiers on the features (Naïve Bayes, Alternating Decision Trees, SVM, and RIPPER, a system that learns rules based on information gain [4]). The conclusion is that all classifiers work reasonably well, with the RIPPER approach, with an F2 score of 80.6%, having a statistically significant improvement over only the worst performing, the Naïve Bayes classifier.

Paper 2 (frontier): JSDC: A hybrid approach for JavaScript malware detection and classification

Junjie Wang, et.al. [11] considered the problem of not only detecting malicious JavaScript, but to also classifying it into a more fine-grained fashion. They considered the following eight classes of malicious JavaScript: *attacks targeting browser vulnerabilities*, *browser hijacking attacks*, *attacks targeting Adobe Flash*, *attacks targeting JRE*, *attacks based on multimedia*, *attacks targeting Adobe PDF reader*, *malicious redirecting attacks*, and *attacks based on Web attack toolkits*.

The system uses HtmlUnit to de-obfuscate the JavaScript as a preprocessing step, and then extracts features. The features are used to detect malware using a learned classifier, and if detected as malicious, it is then classified as one of the defined classes, or presented to the user as a potentially new class.

The HtmlUnit system is used to parse and interpret the JavaScript code, but the authors claim that, since it does not render the web page, their approach is "mostly static". It is however reasonable to expect some computational overhead for this preprocessing step. Features in use include *n-gram statistics*, *character frequencies*, *commenting style*, *entropy*, and *program information, such as HTML properties and API usage patterns*.

The paper evaluates four different classifiers: *Random Trees*, *Random Forest*, *J48*, and *Naïve Bayes*. The system is trained and evaluated using a dataset with 20000 benign scripts, and in total 942 scripts from the eight different classes of malicious scripts. Using the Random Forest classifier, the system obtains a cross-validation accuracy of 99.95% for detection, and 92.14% for fine-grained classification.

Paper 3 (frontier): A deep learning approach for detecting malicious JavaScript code

Yao Wang, et.al. [12] presents a simple approach for learning a classifier for malicious JavaScript (see Figure 3). It employs stacked denoising auto-encoders (SdA), a reasonable choice if you have limited labeled

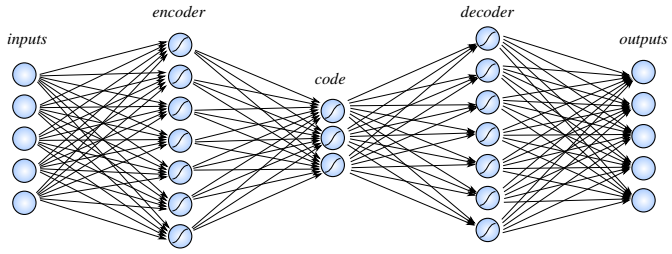

Figure 2: An autoencoder is trained to recreate its input at the output layer. A denoising autoencoder has the same objective, but is given a stochastically corrupted input. The coding layer is typically a bottle neck of some kind, either by having a smaller size, or by regularization.

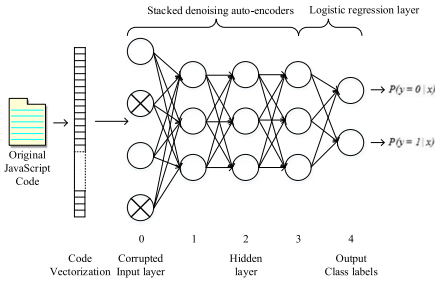

Figure 3: Yao Wang et.al.'s model. Illustration from paper. The illustration is simplified, and shows only three units in each hidden layer, while the evaluated model has 250 units in each hidden layer.

data. An SdA is a deep neural network in which each layer has been pretrained one at a time. The layerwise pretraining (see Figure 2) consists of training each layer as an encoder to compute a vector-valued code, from a stochastically corrupted input, while applying a decoder layer, with the objective that the decoder output is as similar to the uncorrupted input as possible. Then only the encoder is retained and used as a hidden layer in the final model. The model takes character sequences encoded with ascii as inputs, downprojected with a random projection from 20000 dimensions to 480. The model shows some improvement when compared to the approaches presented in Likarish, et.al. Yao Wang, et.al. has around 13000 positive examples and 13000 negative examples in their dataset, which is collected in a manner similar to Likarish, et.al., by downloading scripts from black-listed URLs from VX Heaven (malicious), and Alexa's top web sites (benign).

Discussion and conclusions

Machine learning methods provide a way of detecting malicious JavaScript code. The trained system could be used to inform web browser users

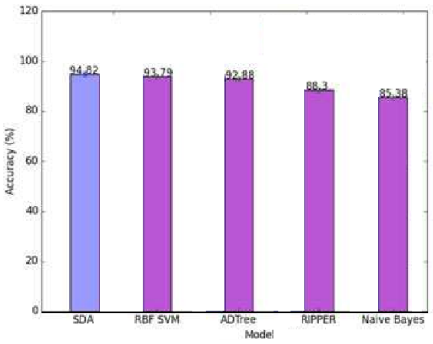

Figure 4: Accuracy of Yao Wang et.al.'s Stacked Denoising Autoencoder approach, compared to each classifier in Likarish, et.al.

when a script is likely to be malicious, letting the user take action, or even automatically disable the malicious script.

Neither of the authors of the papers have published their datasets, providing a hurdle for people who want to develop and benchmark new algorithms.

While most malicious scripts are obfuscated, and few obfuscated scripts are benign, there are (benign) scripts that have been obfuscated for the reason of trying to counter unwanted copying. These will most likely provide a difficulty for the trained classifiers, as the training data mostly contains examples of obfuscated scripts that are to be classified as malicious.

Yao Wang, et.al. show one way of learning features, but none of the more recent approaches. There may be room for improvement by using more modern deep learning ideas, that have shown to work for classifying other textual data, such as convolutional neural networks or recurrent neural networks.

References

[1] Simon Aebersold, Krzysztof Kryszczuk, Sergio Paganoni, Bernhard Tellenbach, and Timothy Trowbridge, *Detecting obfuscated javascripts using machine learning*, The 11th International Conference on Internet Monitoring and Protection (ICIMP). IARIA, 2016.

[2] Yaser Alosefer and Omer Rana, *Honeyware: a web-based low interaction client honeypot*, Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on, IEEE, 2010, pp. 410–417.

[3] YoungHan Choi, TaeGhyoon Kim, SeokJin Choi, and Cheolwon Lee, *Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis*, International Conference on Future Generation Information Technology, Springer, 2009, pp. 160–172.

[4] William W Cohen, *Fast effective rule induction*, Proceedings of the twelfth international conference on machine learning, 1995, pp. 115–123.

[5] Charlie Curtsinger, Benjamin Livshits, Benjamin G Zorn, and Christian Seifert, *Zozzle: Fast and precise in-browser javascript malware detection.*, USENIX Security Symposium, 2011, pp. 33–48.

[6] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh, *A fast learning algorithm for deep belief nets*, Neural computation **18** (2006), no. 7, 1527–1554.

[7] Shamsul Huda, Jemal Abawajy, Mamoun Alazab, Mali Abdollalihian, Rafiqul Islam, and John Yearwood, *Hybrids of support vector machine wrapper and filter based framework for malware detection*, Future Generation Computer Systems **55** (2016), 376–390.

[8] Martin Johns, *On javascript malware and related threats*, Journal in Computer Virology **4** (2008), no. 3, 161–178.

[9] Hong-Geun Kim, Dong-Jin Kim, Seong-Je Cho, Moonju Park, and Minkyu Park, *Efficient detection of malicious web pages using high-interaction client honeypots*, Journal of Information Science and Engineering **28** (2012), no. 5, 911–924.

[10] Peter Likarish, Eunjin Jung, and Insoon Jo, *Obfuscated malicious javascript detection using classification techniques*, Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on, IEEE, 2009, pp. 47–54.

[11] Junjie Wang, Yinxing Xue, Yang Liu, and Tian Huat Tan, *Jsdc: A hybrid approach for javascript malware detection and classification*, Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ACM, 2015, pp. 109–120.

[12] Yao Wang, Wan-dong Cai, and Peng-cheng Wei, *A deep learning approach for detecting malicious javascript code*, Security and Communication Networks (2016).