Detecting Cross-Site Scripting Attacks using Machine Learning

Fawaz A. Mereani[1,2] and Jacob M. Howe[1]

[1] Department of Computer Science,
City, University of London
London, United Kingdom
fawaz.mereani@city.ac.uk, j.m.howe@city.ac.uk
[2] Umm Al-Qura University, Makkah, Saudi Arabia

Abstract. Cross-site scripting (XSS) is one of the most frequently occurring types of attacks on web applications, hence is of importance in information security. XSS is where the attacker injects malicious code, typically JavaScript, into the web application in order to be executed in the user's browser. Identifying that a script is malicious is an important part of the defence of a web application. This paper investigates using SVM, k-NN and Random Forests to detect and limit these attacks, whether known or unknown, by building classifiers for JavaScript code. It demonstrated that using an interesting feature set combining language syntax and behavioural features results in classifiers that give high accuracy and precision on large real world data sets without restricting attention only to obfuscation.

Keywords: Cross-Site Scripting, System Security, Supervised learning, Classifiers, Features selection

1 Introduction

Web applications are used everywhere and involve sensitive and personal data. This makes them a target for malware exploiting vulnerabilities to obtain unauthorized data stored on the computer. Such attacks include SQL injection, cross-site scripting (XSS) and more. XSS can affect the victim by stealing cookies, modifying a web page, capturing clipboard contents, keylogging, port scanning, dynamic downloads and other attacks [17]. Therefore, the safety of web applications is a very important task for developers. The lack of verification of the client input or the environment is the most common security weakness in web applications [18] and such weaknesses are repeatedly discovered and exploited on both client side and server side. SQL injection and XSS remain in the top ten vulnerabilities listed in the Open Web Application Security Project (OWASP) [14]. This paper investigates the use of machine learning techniques to build classifiers to allow the detection of XSS in JavaScript. The current research focuses on stored or persistent XSS, where a malicious script is injected into a web application and stored in the database. Then on every visit to the page, the

script will be executed on the user's browser. Such an attack might target blogs, forums, comments or profiles [7, 11, 24].

Work on detecting and protecting against XSS attacks can be broadly categorised into three kinds. Firstly, static analyses which review the source code without execution; whilst such approaches can give formal guarantees that certain vulnerabilities do not occur, they may also be slow or fail to give a result at all. Secondly, dynamic analyses which attempt to determine what the script does at execution time. Modifying the interpreter [16] or checking the syntactic structure [19] are strategies of this analysis. However, it is hard to modify a language's interpreter; vulnerabilities that are caused by the interaction of multiple modules [3] are, therefore, hard to prevent. Thirdly, machine learning can use knowledge of available scripts to build classifiers to predict aspects of the behaviour of new scripts [4]. The advantages of using machine learning are: first, once a classifier has been built it can quickly predict whether or not a script is malicious; second, it does not need a sandbox to analyse the script; third, the classifier has predictive capabilities to detect new malicious JavaScript. In this work, machine learning is used to detect stored XSS with high accuracy and precision. Scripts may well be obfuscated; importantly, the aim is to classify all scripts, whether obfuscated or not. The design space for such an approach is large, with choices of how to build classifiers, and an even larger choice of how to abstract concrete code into a collection of features that the machine learning algorithms will work on. The contributions of this work are as follows:

- a new selection of program features, drawn from program syntax and program behaviours, is given for the learning algorithms to work on
- the collection of a balanced dataset of scripts from multiple sources giving good coverage of both malicious and benign scripts
- the use of support vector machines (SVM), k-nearest neighbour (k-NN), and Random Forests as learning algorithms to give classifiers; this is the first evaluation of Random Forests on XSS problems
- the evaluation of the resulting classifiers on training and real world data.

The rest of this paper is organised as follows: Section 2 gives an overview of relevant aspects of JavaScript and JavaScript obfuscation. Section 3 discusses related work on machine learning and XSS. Section 4 details the dataset collection and features selection. Section 5 gives the experimental data on the performance of the classifiers, and includes discussion of the results. Further discussion, direction of future work and conclusions are given in section 6.

2 Background

2.1 JavaScript

JavaScript is a language commonly used in the development of web pages to make them more dynamic and interactive. It is client-side which allows the source code to be executed in the web browser rather than on the server. This allows

functions to run after loading the web page without the need to communicate with the server, for example, producing an error alert before sending information to the server. Scripts can be inserted within the HTML or can be referenced in a separate .js file. JavaScript is a good choice for attackers to carry out their attacks and to spread them over the Internet, because the majority of websites use JavaScript and it is supported by all web browsers. Hence, it is the target of many XSS, SQL injection and passive download attacks [22].

2.2 Obfuscation

The goal of obfuscation is to modify the code to make it hard to read or understand. For example, by changing the names of variables or functions, or by using operators to compound terms to give program constructs. Both benign and malicious scripts can use obfuscation techniques with different purposes for each one. Benign obfuscation aims to protect privacy or intellectual rights, while malicious obfuscation works on disguising malicious intentions and evading the static inspection checks. Multiple obfuscation methods can be applied by attackers to best hide malicious scripts [25]. A simple example of malicious JavaScript obfuscation by using URL Encoded is:

%3Cs c r ipt%3E%0D%0Aal e r t%28document . c o oki e%29%3B%0D%0A
%3C%2Fs c r ipt%3E

The original script after deobfuscated is as follows:

$$< script > alert(document.cookie); < /script >$$

This paper considers JavaScript that may or may not be obfuscated and aims to classify scripts as either malicious or benign in either case.

3 Related Work

A number of approaches have been taken to dealing with XSS. The standard approach for the web application developer is to use sanitization and escaping to prevent untrusted content being interpreted as code [24, 23]. Alternatively parser-level isolation can confine user input data during the lifetime of the web application [12]. Note that this isn't detection of XSS, rather prevention of its execution through good coding practice. This is preferred to blacklists which are viewed as easy to circumvent [24]. Another technique to defend against XSS vulnerability is to use randomized namespace prefixes with primitive markup language elements to make it hard for the attacker to use these elements [20]. Previous methods aim to remove malicious elements from untrusted data, however, as with blacklists some XSS vectors can easily bypass many powerful filters. In [8] rules are generated to allow control of communications, with a web proxy blocking communication with untrusted sites. Combinations of static and dynamic techniques use taint analysis to prevent sensitive data being sent to a third party by monitoring the flow of data in the browser [21].

Machine learning techniques have been applied to detecting XSS attacks [10] and are attractive because they can adapt to changes and variations in malicious scripts [9]. Likarish et al. [10] evaluated Naive Bayes, ADTree, SVM, and RIPPER classifiers in detecting obfuscation of scripts (as a proxy for malicious), using features that track the number of times symbols appears in benign and malicious scripts. The classifiers were evaluated using 10-fold cross validation giving precision of 0.92. It should be noted that the test set of obfuscated scripts is small. The approach of Likarish et al. was expanded by Nunan et al. [13], where features were categorized into three groups: (1) obfuscation based, (2) suspicious patterns and (3) HTML/JavaScript schemes. Naive Bayes and Support Vector Machine classifiers were used to classify scripts as XSS or non-XSS. Three datasets were used, malicious (obfuscated) scripts from XSSed.com and benign scripts from Dmoz and ClueWeb09. The classifiers were evaluated using accuracy to give 98.58% with Dmoz dataset, and 99.89% with the ClueWeb09 dataset. This approach has high accuracy, but depends on a single source for malicious scripts and again focuses on obfuscated scripts. Another study analyzing malicious scripts and feature extraction was conducted by Wang et al. [22] where the main idea of feature extraction is that some functions are of limited use in the benign scripts, but are used much more in malicious scripts, such as the DOM-modifying functions, the eval function, the escape function. This technique gives accuracy of up to 94.38%. However, again the technique concentrates only on obfuscated scripts and on DOM-modifying functions. The work of [2] also aims to distinguish between obfuscated and non-obfuscated scripts. Their method gives high precision results up to 100% though again the number of malicious scripts used was small. Komiya et al. [9] used machine learning techniques to classify user input to detect malicious web code. Feature extraction depended on two methods, blank separation, and tokenizing. The idea of the first method is that input contains many terms separated by spaces, a count of each term is used for calculation of feature weight. It should be noted that in a malicious script terms might be separated by characters other than spaces, which would lead to an incorrect feature weight. The second method is based on the idea that malicious code contains tokens that describe the features of malicious web code, with a count of each term used to calculate feature weights. Using this feature extraction technique with SVM gave accuracy of up to 98.95%.

4 Methodology

4.1 Datasets

This paper concentrates on malicious and benign scripts that can be sent to Web applications via HTTP requests. The attacker can use obfuscated scripts, as well as scripts written in the normal manner. To create balanced datasets JavaScript was collected from a number of trusted sources including both obfuscated and non-obfuscated scripts and scripts of with a variety of lengths. Two datasets were gathered. The first data set was collected for training and the second for testing. There is some overlap in the sources of the scripts, but not in the scripts

Table 1. Structural Features

Features Group	Terms
Punctuation	&,%, /, \, +, ', ?, !, ;, #, =, [,], $, (,), ∧ *, , , -, <, >, @, _ , :, {, }, ~, ., space, |, ¦, "
Punctuation Combinations	><, ′ " ><, [], ==, &#

themselves. The benign scripts were obtained from a number of developer and university sites.

For the training set, malicious scripts were obtained from developer sites [15, 1, 6], a selection from XSSed, the largest online archive of XSS vulnerable websites [5] and additional scripts were collected by crawling sites known to be untrustworthy. The test set was drawn entirely from the XSSed archive. Again there is no overlap between sets. The first (training) dataset contains 2000 of each of malicious and benign scripts. The second (test) dataset contains 13,000 each of malicious and benign scripts. Data was prepared for the classification experiments by removing duplicates to get unique scripts, removing extra blank spaces and unnecessary new lines, and lowercasing all letters.

4.2 Selecting Features

There is a large design space for selecting suitable features of JavaScript in order to start classifying scripts. Features in this work are categorized into two groups, 1) structural, and 2) behavioural. In total, 59 features are considered.

Structural Features The structural features are the complete set of non-alphanumeric characters that can occur in JavaScript. These may occur in any script, but if the attacker is using techniques to trick the protection on Web applications this can change the range of characters used in a script. This applies whether or not the script is obfuscated. To give a simple example, a malicious script might add spaces or unnecessary symbols between commands or tags, such as $< \setminus sc\ ri\ pt >$. A benign script would not do this. As another example consider a cookie access separated into two parts and the use of the + sign to recombine the entire command again, $document +' .' + cookie$. Also included in the structural features are combinations of characters that might be used in constructing malicious scripts. There are 33 non-alphanumeric characters, and 5 further combinations of these are considered. The features might be measured in a variety of ways. In the current work the measure is a 0/1 value indicating that the feature does not or does occur in the script. As will be demonstrated later this surprisingly simple measure works very well. Table 1 gives the structural features (where space indicates the blank space character).

Behavioural Features These are a selection of the commands and functions that can be used in JavaScript. The attacker may use them suspiciously and

Table 2. Behavioural Features

Features	Description
Readability	Is the script readable - the number of alphabetical characters.
Objects	document, window, iframe, location, This.
Events	Onload, Onerror.
Methods	createelement, String.fromCharCode, Search.
Tags	DIV, IMG, <script.
Attributes	SRC, Href, Cookie.
Reserve	Var .
Functions	eval().
Protocol	HTTP.
External File	.js file.

differently from the benign developer. That is, the benign developer does not need to hide the intent of their code, whilst on the contrary, the attacker will use a range of commands to create the malicious script. For example, using the eval function frequently, using de-obfuscated functions in the script, or including a malicious script within an image tag. The insight is that combinations of occurrences of commands indicate suspicious activity. There are 21 of these consider in this work. As for the structural features, behavioural features might be measured in many ways. The current work again uses a 0/1 value indicating that the feature does not or does occur in the script. Table 2 gives the behavioural features selected for their potential use in malicious scripts.

4.3 Classifiers

The feature data is used as input for supervised learning algorithms. In this work, support vector machines (SVM), k-nearest neighbour (k-NN), and Random Forests are used, although other classifiers might also be used. Two variations on SVMs are used, with a linear kernal and with a polynomial kernal. A number of parameters used with SVMs were tuned during the training phase: BoxConstraint to control the maximum penalty of misclassification, and OutlierFraction to determine the expected proportion of outliers in the training data. For the k-NN classifier parameter k, the number of neighbours was tuned. For the Random Forest classifier the number of trees in the forest was tuned.

5 Results

5.1 Experiments

MatLab R2016b was used for the experimentation. The experiments focused on the performance of SVM, k-NN, and Random Forest classifiers using the datasets and features described in section 4. For the first set of results the training dataset was divided at random into five folds, with training on four of the five folds,

Table 3. SVM (Linear Kernel) Evaluation

Folds	Accuracy	Precision	Sensitivity	Specificity
1st	94.93%	93.98%	96.37%	93.40%
2nd	94.75%	93.19%	95.69%	93.92%
3rd	95.06%	94.66%	95.03%	95.08%
4th	94.14%	93.14%	96.63%	93.34%
5th	94.81%	93.25%	96.25%	93.45%
Average	94.74%	93.64%	95.99%	93.84%

Table 4. SVM (Polynomial Kernel) Evaluation

Folds	Accuracy	Precision	Sensitivity	Specificity
1st	96.87%	96.10%	97.95%	95.70%
2nd	96.81%	96.20%	97.09%	96.55%
3rd	97.43%	98.04%	96.66%	98.14%
4th	96.87%	96.25%	97.47%	63.25%
5th	97.31%	96.75%	97.85%	96.78%
Average	97.06%	96.67%	97.40%	96.68%

Table 5. k-NN Classifier Evaluation

Folds	Accuracy	Precision	Sensitivity	Specificity
1st	97.00%	96.34%	97.96%	95.95%
2nd	96.50%	96.20%	96.45%	96.53%
3rd	97.43%	97.65%	97.02%	97.82%
4th	97.75%	97.38%	98.11%	97.38%
5th	96.93%	96.37%	97.47%	96.41%
Average	97.12%	96.79%	97.40%	96.82%

and testing on the remaining fold. This five fold testing then gives five training experiments. The SVM with linear kernel was tuned to set the BoxConstraint parameter to 7. The polynomial kernel was tuned by setting the OutlierFraction parameter to 0.10. k-NN was tuned by setting NumNeighbors parameter to 1 (since some malicious scripts might be singletons). Random Forest was tuned by setting the number of tree to 40. The results are described with Precision (often called Detection Rate in a security context), Accuracy, Sensitivity and Specificity. Table 3 shows results with test data for SVM with linear kernel, Table 4 shows the results for SVM with polynomial kernel, Table 5 shows the results for k-NN, and Table 6 shows the results for Random Forest.

To test real world attacks, models for SVM with both linear and polynomial kernel, k-NN, and Random Forest were built by training classifiers using the whole training dataset. Then the testing dataset that contains new malicious and benign scripts was used for testing the classifiers' performance. In Table 8 the results of this test are given and in Table 7 the confusion matrices giving the raw data are presented.

Table 6. Random Forest Classifier Evaluation

Folds	Accuracy	Precision	Sensitivity	Specificity
1st	96.43%	95.28%	97.93%	94.83%
2nd	96.75%	95.54%	97.59%	96.00%
3rd	97.81%	97.65%	97.78%	97.83%
4th	97.68%	97.00%	98.35%	97.03%
5th	97.43%	97.00%	97.85%	97.02%
Average	97.22%	96.47%	97.90%	96.54%

Table 7. Confusion Matrix with Testing Data

	Linear		Polynomial		k-NN		Random Forest	
	Malicious	Benign	Malicious	Benign	Malicious	Benign	Malicious	Benign
Malicious	12783	217	12960	40	12985	15	12980	20
Benign	739	12261	62	12938	50	12950	110	12890

5.2 Discussion

The experiments give two sets of data. The first uses a training set of scripts chosen to give coverage of a variety of styles of scripts – obfuscated or not, varying length. The five fold evaluation shows good performance for the classifiers, with (as expected) SVM with a polynomial kernal giving stronger results than for SVM with linear kernal. The second set of data is designed to give a real-world evaluation of the classifiers learnt from the entire training set. As can be observed in Table 8 the SVM, k-NN, and Random Forest classifiers can distinguish between malicious and benign scripts with high accuracy and detection rate. k-NN performs marginally better than SVM and Random Forest, with accuracy of 99.75% and precision 99.88%. The confusion matrices of Table 7 show the small numbers of false positives and failed detections, 15 of the former and 50 of the latter for the k-NN classifier. These results suggest that classifier based techniques can be a powerful tool for detecting XSS attacks.

Table 8. Evaluation with Testing Data

	Linear	Polynomial	k-NN	Random Forest
Accuracy Rate	96.32%	99.60%	99.75%	99.50%
Precision Rate	98.33%	99.69%	99.88%	99.84%
Sensitivity (TPR)	94.53%	99.22%	99.61%	99.15%
Specificity (TNR)	98.26%	99.69%	99.88%	99.84%

6 Conclusion

This paper has demonstrated that SVM, k-NN, and Random Forest can be used to build classifiers for XSS coded in JavaScript giving high accuracy (up to

99.75%) and precision (up to 99.88%) when applied to a large real world data set. This shows that these classifiers can be added as a security layer either in a browser or (as intended) on a server. The training data was designed to give fair coverage of scripts, including scripts of a variety of lengths and both obfuscated and non-obfuscated scripts. The data is labeled as malicious or benign, rather than using obfuscation as a proxy for maliciousness. Whilst SVM, k-NN, and Random Forest have been used in the experiments, it is expected that other classification methods would also work well.

A systematic direct comparison with previous studies is not possible, however, the new classifiers give performance statistics that stand up well. The current study works with a larger and more diverse suite of scripts than many of these previous studies, and is the first study to use Random Forests as a classifier for XSS. The key to building successful classifiers is the choice of feature set and how the features are measured. With a large design space there is motivation to investigate a wide range of approaches to feature selection. The features chosen in this paper fall into two categories: firstly the complete set of symbols used in the JavaScript language, and secondly aspects of the scripts that are associated with malicious code. This allows the classifiers to find patterns based on the shape of the program (symbols) and the constructs used (behavioural features). One particularly interesting aspect of this work is that, in contrast to other studies, a binary measure has been used for all features. This has given higher accuracy and precision than earlier experiments using weighted measures. This hints that it may be possible to extract rules from the classifiers that describe malicious scripts. Another interesting aspect is the value of k used in the final experiments is 1. This suggests that malicious scripts might well be singletons that stand apart from clusters of benign scripts. Future work is to investigate these aspects, as well as to use the same features with a Neural Network classifier.

References

1. Examples of malicious javascript. https://aw-snap.info/articles/js-examples.php (2014). Accessed: 16/12/2016
2. Aebersold, S., Kryszczuk, K., Paganoni, S., Tellenbach, B., Trowbridge, T.: Detecting Obfuscated JavaScripts using Machine Learning. In: International Conference on Internet Monitoring and Protection. IARIA Press (2016)
3. Balzarotti, D., Cova, M., Felmetsger, V., Vigna, G.: Multi-module vulnerability analysis of web-based applications. In: Computer and Communications Security, pp. 25–35. ACM Press (2007)
4. Domingos, P.: A few useful things to know about machine learning. Communications of the ACM **55**(10), 78–87 (2012)
5. Fernandez, K., Pagkalos, D.: XSS (Cross-Site Scripting) information and vulnerable websites archive. `XSSed.com`. Accessed 14/06/2017
6. Karnad, K.: XSS payloads you may need as a pen-tester. https://www.linkedin.com/pulse/20140812222156-79939846-xss-vectors-you-may-need-as-a-pen-tester (2014). Accessed: 25/12/2016
7. Kirda, E., Jovanovic, N., Kruegel, C., Vigna, G.: Client-side cross-site scripting protection. Computers & Security **28**(7), 592–604 (2009)

8. Kirda, E., Kruegel, C., Vigna, G., Jovanovic, N.: Noxes: a client-side solution for mitigating cross-site scripting attacks. In: Symposium on Applied Computing, pp. 330–337. ACM Press (2006)
9. Komiya, R., Paik, I., Hisada, M.: Classification of malicious web code by machine learning. In: Awareness Science & Technology (iCAST), pp. 406–411. IEEE (2011)
10. Likarish, P., Jung, E., Jo, I.: Obfuscated malicious Javascript detection using classification techniques. In: Malicious and Unwanted Software (MALWARE), pp. 47–54. IEEE (2009)
11. Malviya, V.K., Saurav, S., Gupta, A.: On Security Issues in Web Applications through Cross Site Scripting (XSS). In: Asia-Pacific Software Engineering Conference, vol. 1, pp. 583–588. IEEE (2013)
12. Nadji, Y., Saxena, P., Song, D.: Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In: Network and Distributed System Security Symposium. Internet Society (2009)
13. Nunan, A.E., Souto, E., dos Santos, E.M., Feitosa, E.: Automatic classification of cross-site scripting in web pages using document-based and url-based features. In: Computers and Communications, pp. 702–707. IEEE (2012)
14. OWASP Top 10 - 2017 rc1 (2017). `https://www.owasp.org`. Accessed: 7/6/2017
15. Payloads, X.: XSS payloads you may need as a pen-tester. http://www.xss-payloads.com/payloads.html. Accessed: 14/10/2016
16. Pietraszek, T., Berghe, C.V.: Defending against injection attacks through context-sensitive string evaluation. In: Recent Advances in Intrusion Detection, *Lecture Notes in Computer Science*, vol. 3858, pp. 124–145. Springer (2005)
17. Raman, P.: JaSPIn: JavaScript based Anomaly Detection of Cross-site scripting attacks. Ph.D. thesis, Carleton University, Ottawa (2008)
18. Rocha, T.S., Souto, E.: ETSSDetector: a tool to automatically detect Cross-Site Scripting vulnerabilities. In: Network Computing and Applications, pp. 306–309. IEEE (2014)
19. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. ACM SIGPLAN Notices **41**(1), 372–382 (2006)
20. Van Gundy, M., Chen, H.: Noncespaces: Using randomization to defeat cross-site scripting attacks. Computers & Security **31**(4), 612–628 (2012)
21. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross site scripting prevention with dynamic data tainting and static analysis. In: Network and Distributed System Security Symposium, p. 12. Internet Society (2007)
22. Wang, W.H., Yin-Jun, L.V., Chen, H.B., Fang, Z.L.: A Static Malicious Javascript Detection using SVM. In: International Conference on Computer Science and Electronics Engineering, vol. 40, pp. 21–30. Atlantis Press (2013)
23. Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R., Song, D.: A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In: European Symposium on Research in Computer Security, *Lecture Notes in Computer Science*, vol. 6879, pp. 150–171. Springer (2011)
24. Williams, J., Manico, J., Mattatall, N.: Cross-site Scripting (XSS). https://www.owasp.org/index.php/Cross-site_Scripting_(XSS). Accessed: 22/7/2016
25. Xu, W., Zhang, F., Zhu, S.: JStill: mostly static detection of obfuscated malicious JavaScript code. In: Data and application security and privacy, pp. 117–128. ACM Press (2013)