Bivariate Classification of Malware in JavaScript using Dynamic Analysis

Yash Gupta
CSE, Student
PEC University of Technology
Chandigarh, India
guptayash3@gmail.com

Dr. Divya Bansal
Associate Professor, CSE
PEC University of Technology
Chandigarh, India
divya@pec.ac.in

Dr. Sanjeev Sofat
HOD, CSE
PEC University of Technology
Chandigarh, India
sanjeevsofat@pec.ac.in

***Abstract*---JavaScript is used as an attack vector to infect webpages to gain access to user's information. We present a tool that will dynamically analyze and perform bivariate classification of webpages as malicious or benign. We categorized the general behavior of JavaScript using datasets of known benign and malicious JavaScript by using a classifier which is trained on the basis of difference between function calls made by malicious and benign JavaScript and identification of Iframe tag in them. A Script is then matched to those categorizations to classify its behavior as malicious or benign. Here we have developed a light weight malicious JavaScript detection approach which can be used in real time as most of the existing techniques perform offline analysis.**

***Keywords*---malicious JavaScript, dynamic analysis, classification, caffeine monkey**

I. Introduction

The increased amount of information exchange over internet has focused attackers towards web attacks in order to steal user's personal and financial information. Attackers are using different type of web technologies as their attack vectors which include different type of scripting languages such as JavaScript, VBScript and many more. In web environment the scripting attacks through JavaScript have become a common but severe security threat. The attacker launches these attacks to leak information, steal passwords or load malware into the victim's system through vulnerable JavaScript code. A recent 2013 report from Sophos Labs indicates that 85.2 % of all website attacks are due to "drive by redirect" attacks using malicious JavaScript [11].

JavaScript is an object oriented scripting language which has been widely adopted as a client side scripting language. It can be embedded in HTML and can interact with Document Object Model of HTML. It is used to perform various functions over client side for eg: form validations, access browser properties, create highly responsive interfaces that improve user's experience. These capabilities can endanger the end-user if the Web page is infected with malicious JavaScript code [9]. JavaScript can be disabled in any browser but 89.2% of the websites use JavaScript as client side scripting language [16], so it is not a realistic option to surf web with JavaScript disabled. Malicious JavaScript code is injected in web pages using different attack techniques such as cross site scripting. Moreover, attackers obfuscate this code to avoid detection mechanisms. Obfuscated code is one which is difficult to understand by human analysis; it is done to hide the actual meaning of the code. This infected code is used to spread worms, install Malware and conduct different types of attacks such as click jacking, cookie stealing. For Example: An attacker can easily put a hidden Iframe tag in a webpage and can redirect user to a malicious web page.

II. Related Work

Today most of the approaches used to counter malicious websites are infrastructure-based [7, 8]. In these approaches different websites are crawled and analyzed continuously. They use different type of static and dynamic analysis methods and store the analysis results in a database available for offline analysis. The systems provide browser plugins which check the user's requested URL against the database. If the URL is classified as malicious in the database, the user is warned against visiting the website. The advantage of this approach is that it is instantly usable by end-users. These approaches are flexible as they can use any type of technique to detect malicious JavaScript but these approaches suffer from inconsistency issues because a user is warned based on the classification that is stored in the database. The problem is that there is no assurance that the website visited by the user has not been compromised since the analysis is done in an offline mode due to huge performance burdens.

Ben Feinstein et al. [1] proposed a system, in which they examined the current state of JavaScript obfuscation and evasion techniques, approaches for collecting JavaScript samples from the wild, and developed methods for analyzing the collected scripts. They developed a suite of tools for collecting and indexing JavaScript, interpreting the scripts in a sandboxed environment, and then performing functional analysis for manual and automated detection mechanisms. They developed a tool Caffeine Monkey [1] that sandboxes the JavaScript. It is an open source modification of spider monkey [12]. They classify JavaScript on the basis of difference between numbers of function calls made by benign and malicious JavaScript. The major drawback is that the mechanism is not automated and hence analysis of malware infected JavaScript has to be done in an offline mode.

Moreover the Caffeine Monkey Engine is unable to interpret JavaScript in which DOM (Document object Model) functionality is used. DOM is a programming API for HTML documents [14]. As most of the websites today use DOM along with JavaScript so the existing tool can't be used to detect malwares efficiently.

M. Cova et al [2] presents a tool called JSAND (JavaScript Anomaly-based Analysis and Detection) The tool is publically available where a user can submit URLs and JavaScript files to check whether they exhibit some malicious behavior or not. It is an instrumented browser emulator that executes the JavaScript code to check its runtime behavior as it will actually behave in a user browser. The system uses a number of features which capture intrinsic characteristics of attacks along with machine learning techniques for anomaly detection which is detecting potentially malicious behavior. Additionally, the system is able to analyze obfuscated code and generates attack signatures for signature-based detection systems. The major drawback is that the technique is not automated as the user has to manually enter each URL to check whether it is benign or malicious.

Wang et al. [3] presents a system utilizing a honeypot within a virtual machine (VM). Within the VM, the tool creates an instance of the Microsoft Internet Explorer, navigates to a specific URL, and waits for few minutes. Changes to the VM's file system and registry are flagged. If a change to the file system outside of the temporary directory of the browser is detected, the site is classified as malicious. The system then shuts down the potentially infected VM and starts a clean one to analyze the next URL. To determine if an exploit just works against a specific combination of versions of Microsoft Internet Explorer and the underlying operating system (or if it even is a zero-day exploit), a pipeline of such virtual machines is used, each covering a different patch level. The problem in this approach is that it can be used only for Internet Explorer. Another problem of this approach is that it does not scale well. The analysis effort can be nicely parallelized on redundant hardware but analyzing millions of websites per day can be rather tedious and costly job. Moreover the frequency of analysis of a website is also important because if a website later gets infected; it should be crawled and analyzed as quickly as possible. Cox et al. [4] proposed a Virtual Machine based technique that runs on client side, thus partly leveraging the problem of scalability. This introduced client-side approaches to counter malicious websites. But the problem is that the users now have to invest in additional hardware to run a virtual machine and additional detection software.

Several static techniques are also used for identifying malicious JavaScript. Saurabh Jain et al. [6] proposed a signature and regular expression based matching technique to identify malicious JavaScript code. Mohammad Fraiwan et al. [5] proposed a technique based on a classification model. They analyzed the behavior and properties of JavaScript code to point out its key features using static analysis techniques. Then classifiers were trained on malicious and benign data. The main problem of these techniques is that they do not check the actual behavior of JavaScript. Moreover, they do not consider the obfuscated JavaScript as it is difficult to perform static analysis techniques on obfuscated JavaScript code which is otherwise a bigger threat and a security challenge.

III. Detection Technique for Dynamic Analysis of JavaScript

For developing a detection technique for Dynamic analysis, we have to run JavaScript in a sandboxed environment. We extracted several features of known benign and malicious JavaScript after its execution in sandboxed environment and then compare the features of a new script to those of malicious and benign. To automate the detection mechanism we have used a classifier. The base of our dynamic analysis is the caffeine monkey engine [1]. Caffeine Monkey engine interprets the JavaScript and creates a log of function calls made by that JavaScript along with the deobfuscated JavaScript. As discussed earlier in section II caffeine monkey engine cannot interpret JavaScript containing DOM. we customized the Caffeine Monkey engine and to do so, we have defined the document, window, location, navigator objects of DOM and its properties in it. The log generated by the customized Caffeine Monkey engine is then used to obtain features for classification of a script as malicious or benign.

A. Features Used For Classification

We used 8 features to compare general behavior of malicious and benign JavaScript and then classify a script as malicious or benign. Out of eight, seven are based on the frequency of the functions called by the JavaScript code and eighth is based on detecting the presence of Iframe tag inside the JavaScript. These features are obtained from the log created by our customized Caffeine Monkey engine.

1) Function Calls

Combining the existing approaches which could count 6 function calls, we have counted one additional function 'Unescape'. The complete list of functions counted is described below:

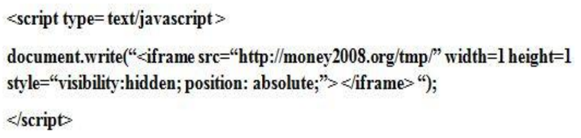
- Escape: The escape function encodes a string and makes it portable, so it can be transmitted across any network to any computer that supports ASCII characters.[15]
- Eval: The Eval function executes or evaluates and argument. If the argument is JavaScript code the function executes it otherwise it evaluates it.[15]
- String Instantiation: Create a new string object. Whenever a New String is created we count it.
- Element Instantiation: Create a new element object. Whenever a New Element Object is created we count it.
- Object Instantiation: Create a new object instance. Whenever a new object is created excluding string and element objects we count it.
- Document. Write: A method that writes HTML expressions or JavaScript code to a document.

- Unescape: The Unescape function decodes an encoded string.

2) *Iframe Tag*

In our approach we have been able to detect malware injected using Iframe tag which has become a serious attack method and usually goes undetected by present detection system.

The <Iframe> tag specifies an inline frame which is used to embed another document within the current HTML document and is thus used quite often used by attackers to inject malicious content on a website [6]. Iframe tag can also be injected via JavaScript in a webpage using its document.write method as shown in figure 1. By using an Iframe an attacker can redirect a user from a benign website to a malicious webpage. Moreover an attacker can hide an Iframe, so that the users don't even know that they became victim of an attack. There are several ways of hiding Iframe as it has attributes like height, width, style so this tag can easily be used for creating severe attacks as a common user visiting a website with hidden frame has no idea that he or she is a victim of exploit. Ways of hiding an Iframe tag: setting its height=1 and width=1, style="visibility: hidden" or style="opacity: 0". If an Iframe tag embedded in JavaScript has such attributes then it is possible that it is a malicious JavaScript. So we considered an Iframe tag embedded in JavaScript as one of the features to classify a Script. Figure 1 presents an example of how an Iframe tag can be used inside JavaScript and redirect a user to some malicious webpage.



```
<script type= text/javascript>
document.write("<iframe src="http://money2008.org/tmp/" width=1 height=1
style="visibility:hidden; position: absolute;"> </iframe> ");
</script>
```

Figure 1. Example of Iframe tag embedded in JavaScript

.

IV. Classification Model

Figure 2 shows the steps involved in classification of a JavaScript as malicious or benign.

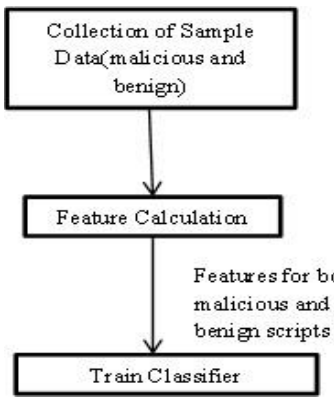

Collection of Sample Data(malicious and benign)
Feature Calculation
Features for both malicious and benign scripts
Train Classifier

Figure 1. Classification Model

A. *Sample Data Collection*

To train a classifier we need both benign and malicious JavaScript samples of websites. For benign samples we extracted JavaScript from front pages of top websites rated on Alexa.com [11]. We obtained 160 malicious samples. Out of those 160 samples 10 were identical so we discarded them and trained the classifier using 100 samples. 50 samples were kept reserved for testing purpose.

B. *Feature Calculation*

For calculating our features we submit every JavaScript sample to the customized Caffeine Monkey engine that generates a log of all the function calls made by JavaScript along with the deobfuscated JavaScript. The log is then parsed using a python script to calculate the frequency of the seven function calls made by the JavaScript and to check whether JavaScript contains Iframe tag in it as explained in section III-A. The values of the frequency of each of the seven function calls and the Iframe tag are stored in a database. The values obtained act as features or independent variables for training of our classifier. Classifier training is discussed in next step. After calculating and storing the values of features in the database we exported the database to an Excel file so that it can be used for classifier training.

C. *Classifier Training*

As our current model does classification as binary that is either malicious (1) or benign (0) we used binary logistic regression as our classifier. IBM's SPSS software [17] is used which provides us binary logistic regression. Binary logistic regression is a type of regression analysis which is used for predicting the outcome of a categorical dependent variable (a dependent variable can take only two possible values that is 1 or 0) based on one or more independent variables [13]. It uses a logistic function shown in equation 1, which tells us to which class a new observation will belong.

$$\boldsymbol{F(t) = \frac{1}{1+e^{-t}}}$$ Where

$$\boldsymbol{t = \beta_0 + \beta_1 x_1 \dots + \beta_k x_k}$$

Equation 1. Logistic Function [13]

In Figure $x_1, x_2, \dots, x_k$ are the independent variables or features and t is the measure of the total contribution of x variables. β_0 ,$\beta_1, \dots, \beta_k$ are the parameters calculated by logistic regression which will be used to classify a new observation.

The value of F(t) lies between 0 and 1.Default cutoff for binary logistic regression is 0.5 means if 0<F(t)<0.5 then the observation belongs to category 0 and if 0.5<F(t)<1 then the observation belongs to category 1. In our research classification of JavaScript as malicious (1) or benign (0) is dependent variable and the 8 features we are calculating are the independent variables. Here the value of k=8. The eight features X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8 are eval, escape, Unescape, string instance, element_instance, object_instance, Document.write, Iframe respectively.

We submit a file to SPSS containing all the eight features for all the samples along with their classification as benign (0) or malicious (1). Then we performed binary logistic regression over it. After training the logistic regression gives us the values of β_0, β_1, β_2 , ..., β_8 that will be used to classify a new script as malicious or benign.

D. *Classification of JavaScript*

The Figure 3 shows the steps in classification of a new JavaScript.

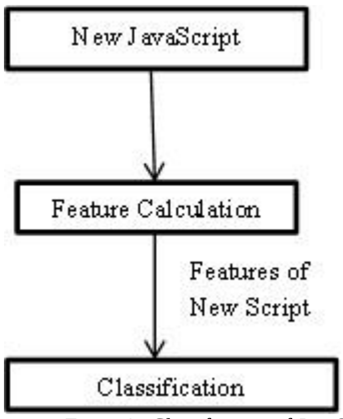


Figure 2. Classification of JavaScript

To classify a new script first we obtain features of that script as explained in section IV-B. Then we put those features (x_1, x_2, ….., x_8) and the parameter values (β_0, β_1, β_2 , ..., β_8) in the logistic function(Equation 1) to obtain its classification. The function will give us a value according to which the category of the script is decided. If the value of F(t) lies between 0 and 0.5 the script belongs to category 0 that means its benign and if value of F(t) lies between 0.5 and 1, the script belongs to category 1 that is its malicious.

V. Results and Discussions

To check the accuracy of our tool we tested the 50 malicious samples that were kept reserved for testing.

Table 1. Parameter Values

β_0	-1.473
β_1	-0.071
β_2	0.001
β_3	0.003
β_4	0.108
β_5	-0.264
β_6	-2.258
β_7	6.698
β_8	0.114

Table 1 shows the values of parameters (β_0, β_1, β_2 , ….., β_8) obtained from classifier. In Table 2 (x_1, x_2, ….., x_7) are the values of the frequency of are eval, escape, Unescape, string instance, element_instance, object_instance, Document.write functions respectively and x_8 is the presence of Iframe in JavaScript. F(t) is the output of the logistic function. Classification value tells whether the script is malicious or not. If the value of classification is 1 the script is malicious and if the value is 0 it is benign.

Table 2. Show results of five of the test samples.

Test Sample	1	2	3	4	5
X_1	1	12	1	1	2
X_2	0	13	0	1	0
X_3	1845	8530	1112	711	1367
X_4	3	0	0	0	0
X_5	0	0	0	0	0
X_6	0	0	0	1	1
X_7	0	0	0	3	2
X_8	1	0	0	1	0
F(t)	0.999	0.995	0.994	0.994	.993
Classification	1	1	1	1	1

Table 2. Test Sample Results

Overall we tested 75 samples out of which 50 are malicious samples that were kept reserved for testing and 25 are benign samples. 49 samples are detected as malicious and 1 sample is detected as benign by our tool. This shows that our tool has 2% false positive for testing dataset. . One sample that was detected as benign by the tool is due to very less strings made in that JavaScript i.e x_2=703 and only one eval call is made i.e x_3=1. False positive is calculated as per the equation below.

$$fp = \frac{\text{malicious script classified as benign}}{\text{total malicious scripts}} \times 100\%$$

25 benign samples were tested out of which 25 samples were detected as benign by the tool. This shows that the tool has 0% false negative for testing dataset. False negative (fn) is calculated as per the equation below. In equation value of benign script classified as malicious is 0 and total benign scripts are 25. The value of false negative comes out to be 0.

$$fn = \frac{\text{benign script classified as malicious}}{\text{total benign scripts}} \times 100\%$$

The run time for the analysis of these test samples was observed on the following platform - Intel core 2 duo 8600 @2.4 ghz with 3GB of ram with Ubuntu 12.10 installed on it. The average runtime for an analysis is 1157.5 ms. The maximum time for a script was 3109 ms and the minimum time recorded was 148 ms. We consider this to be acceptable for the first unoptimized implementation. To further refine and validate our model, the technique will be validated over a large test data which is still being calculated.

VI. Conclusion and Future Work

In this work we presented a tool for dynamically analyzing JavaScript for malware and vulnerabilities. We analyzed the actual runtime behavior of JavaScript as it may behave in a user’s browser in real time. We characterized the general behavior of malicious and benign JavaScript using a classifier and then matched a new JavaScript to those categorizations. . Results show that the tool is able to detect malicious samples with 2% false positives and 0% false negatives. The tool can be easily integrated with a proxy server to provide real time security to the users as the average overhead to perform an analysis is 1157.5 ms only. The tool can be trained with more sample dataset to obtain more precise results.

Acknowledgment

This work is done in Cyber Security Research Center (CSRC) located at PEC University of Technology. The authors would like to that Government of India, Ministry of Communications and Information Technology, Department of Information Technology, New Delhi, for funding the Project “**Development of Cloud Based Framework for Delivering Security as a Service**”, under which this research work has been done. We would also like to thank Mr. Pankaj Tanwar for his entire help rendered in understanding the source codes and customization during development phase.

References

[1] Ben Feinstein, Daniel Peck, “Caffeine Monkey: Automated Collection, Detection and Analysis of Malicious JavaScript,” Black Hat USA, 2007.

[2] M. Cova, C. Kruegel, and G. Vigna,“ Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code,” World Wide Web Conference (WWW), April 2010.

[3] Yi-Min Wang and Doug Beck and Xuxian Jiang and RoussiRoussev, “Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites that Exploit Browser Vulnerabilities,” Network and Distributed System Security Symposium (NDSS), 2006.

[4] R.S. Cox,S.D. Gribble,H.M.Levy and J.G. Hansen,”A Safety-Oriented Platform for Web Applications,” IEEE Symposium on Security and Privacy,2006

[5] Mohammad Fraiwan, Rami Al-Salman, Natheer Khasawneh and Stefan Conrad “Analysis and Identification of Malicious JavaScript Code,” Information Security Journal: A Global Perspective, 2012.

[6] Saurabh Jain, Deepak Sing Tomar, Divya Rishi Sahu,”Detection of JavaScript Vulnerability At Client Agent,” International Journal of Scientific and Technology Research,2012

[7] Google safe browsing v2 API protocol guide- https://developers.google.com/safe-browsing/developers_guide_v2.

[8] McAfee. Site Advisor.http://www.siteadvisor.com.

[9] JavaScript attacks. http://en.wikipedia.org/wiki/Cross-site_scripting.

[10] Alexa. Global top sites. http://www.alexa.com/topsites.

[11] Sophos Labs security threat report-2013. http://www.sophos.com.

[12] Spider monkey. https://developer.mozilla.org/en/docs/SpiderMonkey.

[13] Logistic Regression. http://en.wikipedia.org/wiki/Logistic_regression.

[14] Document Object Model. http://www.w3.org/TR/WD-DOM/introduction.html.

[15] JavaScript. http://www.w3schools.com/jsref/.

[16] JavaScript usage. http://w3techs.com/technologies/details/cp-javascript/all/all.

[17] IBM Spss Software. http://www-01.ibm.com/software/analytics/spss/.