Obfuscated Malicious Javascript Detection using Classification Techniques

Peter Likarish, Eunjin (EJ) Jung
Dept. of Computer Science
The University of Iowa
Iowa City, IA 52242
{plikaris, ejjung}@cs.uiowa.edu

Insoon Jo
Distributed Computing Systems Lab
School of Computer Science and Engineering
Seoul National University
ischo@dcslab.snu.ac.kr

Abstract

As the World Wide Web expands and more users join, it becomes an increasingly attractive means of distributing malware. Malicious javascript frequently serves as the initial infection vector for malware. We train several classifiers to detect malicious javascript and evaluate their performance. We propose features focused on detecting obfuscation, a common technique to bypass traditional malware detectors. As the classifiers show a high detection rate and a low false alarm rate, we propose several uses for the classifiers, including selectively suppressing potentially malicious javascript based on the classifier's recommendations, achieving a compromise between usability and security.

1. Introduction

Malware distributors on the web have a large number of attack vectors available including: drive-by download sites, fake codec installation requests, malicious advertisements and spam messages on blogs or social network sites. Most common attack methods use malicious javascript during part of the attack, including cross-site scripting [20] and web-based malware distribution. Javascript may be used to redirect a user to a website hosting malicious software, to create a window recommending users download a fake codec, to detect what software versions the user has installed and select a compatible exploit or to directly execute an exploit.

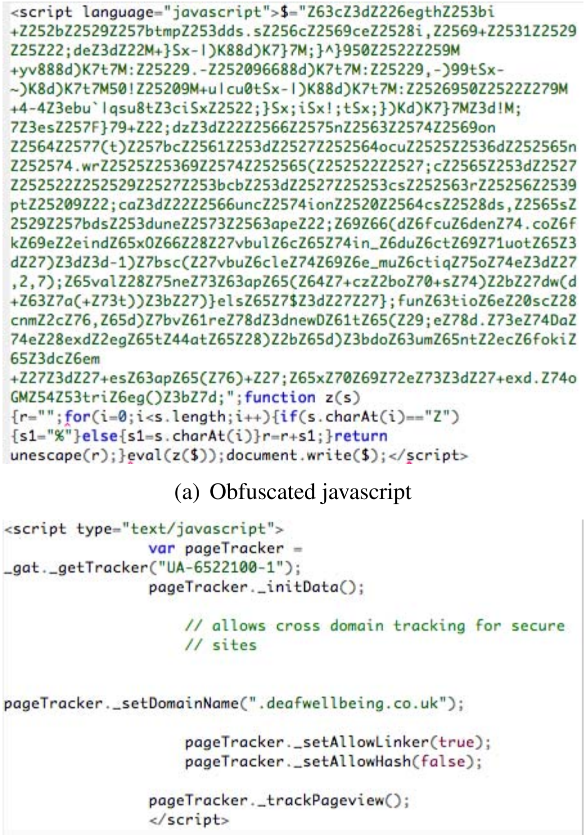
Malicious javascript often utilizes obfuscation to hide known exploits and prevent rule-based or regular expression (regex)-based anti-malware software from detecting the attack. The complexity of obfuscation techniques has increased, raises the resources necessary to deobfuscate the attacks. For instance, attacks often include references to legitimate companies to disguise their purpose and include context-sensitive information in their obfuscation algorithm. Our detector takes advantage of the ubiquity of this obfuscation. Fig. 1 shows the clear difference between obfuscated javascript and a benign script. Even though the difference is easily discernable by the human eyes, obfuscation detection is not trivial. We investigate automating the detection of malicious javascript using classifiers trained on features present in obfuscated scripts collected from the internet. Of course, some benign javascript is also obfuscated as well, and some malicious javascript is not. Our results show that we detect the vast majority of malicious scripts while detecting very few benign scripts as malicious. We further address this in Section 5.1.

In the next section, we discuss prior research on malicious javascript detection. Then, we describe the system we used to collect both malicious and benign javascripts for training and testing machine learning classifiers. We follow this with performance evaluation of four classifiers and conclude with recommendations based on our findings as well as detailing future work.

2. Related work

Javascript has become so widespread that nearly all users allow it to execute without question. To protect users, current browsers use sandboxing: limit-

978-1-4244-5787-8/10/$26.00 ©2009 IEEE



```
<script language="javascript">$="Z63cZ3dZ226egthZ253bi
+Z252bZ2529Z257btmpZ253dds.sZ256cZ2569ceZ2528i,Z2569+Z2531Z2529
Z25Z22;deZ3dZ22M+}Sx-|)K88d)K7}7M;}^}950Z2522Z259M
+yv888d)K7t7M:Z25229.-Z252096688d)K7t7M:Z25229,-)99tSx-
~)K8d)K7t7M50!Z25209M+u|cu0tSx-|)K88d)K7t7M:Z2526950Z2522Z279M
+4-4Z3ebu`|qsu8tZ3ciSxZ2522;}Sx;iSx!;tSx;})Kd)K7}7MZ3d!M;
7Z3esZ257F}79+Z22;dzZ3dZ22Z2566Z2575nZ2563Z2574Z2569on
Z2564Z2577(t)Z257bcZ2561Z253dZ2527Z252564ocuZ2525Z2536dZ252565n
Z252574.wrZ2525Z25369Z2574Z252565(Z25252Z2527;cZ2565Z253dZ2527
Z25252Z252529Z2527Z253bcbZ253dZ2527Z25253csZ252563rZ25256Z2539
ptZ25209Z22;caZ3dZ22Z2566uncZ2574ionZ2520Z2564csZ2528ds,Z2565sZ
2529Z257bdsZ253duneZ2573Z2563apeZ22;Z69Z66(dZ6fcuZ6denZ74.coZ6f
kZ69eZ2eindZ65x0Z66Z28Z27vbulZ6cZ65Z74in_Z6duZ6ctZ69Z71uotZ65Z3
dZ27)Z3dZ3d-1)Z7bsc(Z27vbuZ6cleZ74Z69Z6e_muZ6ctiqZ75oZ74eZ3dZ27
,2,7);Z65valZ28Z75neZ73Z63apZ65(Z64Z7+czZ2boZ70+sZ74)Z2bZ27dw(d
+Z63Z7a(+Z73t))Z3bZ27)}elsZ65Z7$Z3dZ27Z27};funZ63tioZ6eZ20scZ28
cnmZ2cZ76,Z65d)Z7bvZ61reZ78dZ3dnewDZ61tZ65(Z29;eZ78d.Z73eZ74DaZ
74eZ28exdZ2egZ65tZ44atZ65Z28)Z2bZ65d)Z3bdoZ63umZ65ntZ2ecZ6fokiZ
65Z3dcZ6em
+Z27Z3dZ27+esZ63apZ65(Z76)+Z27;Z65xZ70Z69Z72eZ73Z3dZ27+exd.Z74o
GMZ54Z53triZ6eg()Z3bZ7d;";function z(s)
{r="";for(i=0;i<s.length;i++){if(s.charAt(i)=="Z")
{s1="%"}else{s1=s.charAt(i)}r=r+s1;}return
unescape(r);}eval(z($));document.write($);</script>
```

(a) Obfuscated javascript

```
<script type="text/javascript">
                var pageTracker =
_gat._getTracker("UA-6522100-1");
                pageTracker._initData();

                    // allows cross domain tracking for secure
                    // sites


pageTracker._setDomainName(".deafwellbeing.co.uk");

                    pageTracker._setAllowLinker(true);
                    pageTracker._setAllowHash(false);

                pageTracker._trackPageview();
                </script>
```

(b) Benign javascript

Figure 1. Example scripts

ing the resources javascript can access. At a high-level, javascript exploits occur when malicious code circumvents this sandboxing or utilizes legitimate instructions in an unexpected manner in order to fool users into taking insecure actions. For an overview of javascript attacks and defenses, readers are referred to [11].

2.1. Disabling javascript

NoScript, an extension for Mozilla's Firefox web browser, selectively allows javascript [13]. NoScript disables javascript, java, flash and other plugin content types by default and only allows script execution from a website in a user-managed *whitelist*. However, many attacks, especially from user-generated content, are hosted at reputable websites and may bypass this whitelist check. For example, Symantec reported that many of 808,000 unique domains hosting malicious javascript were mainstream websites [19].

2.2. Automated deobfuscation of javascript

As mentioned in Section 1, obfuscation is a common technique to bypass malware detectors. Several projects aid anti-malware researchers by automating the deobfuscation process. Caffeine Monkey [6] is a customized version of the Mozilla's SpiderMonkey [14] designed to automate the analysis of obfuscated malicious javascript. Wepawet is an online service to which users can submit javascript, flash or pdf files. Wepawet automatically generates a useful report, checking for known exploits, providing deobfuscation and capturing network activity [3]. Jsunpack from iDefense [8] and "The Ultimate Deobfuscator" from WebSense [1] are two additional tools to automate the process of deobfuscating malicious javascript.

2.3. Detecting and disabling potentially malicious javascript

Egele et al. mitigate drive-by download attacks by detecting the presence of shellcode in javascript strings using x86 emulation (shellcode is used during heap spray attacks) [5].

Hallaraker et al designed a browser-based auditing mechanism that can detect and disable javascript that carries out suspicious actions, such as opening too many windows or accessing a cookie for another domain. The auditing code compares javascript execution to high-level policies that specify suspicious actions [7].

BrowserShield [16] uses known vulnerabilities to detect malicious scripts and dynamically rewrite them in order to transform web content into a safe equivalent. The authors argue that when an exploit is found, a policy can be quickly generated to rewrite exploit code before the software is patched. Others proposed a similar javascript rewriting approach as well [23].

Finally, in 2008 Seifert et al. proposed a set of features combining HTTP requests and page content, (including the presence and size of iFrames and the use of escaped characters) and used that to generate a decision tree [18]. There is little overlap between the features we evaluate here and those proposed in [18] and it may be possible to combine the two sets to improve detection. In addition, we examine additional classifiers and determined that classifiers using very different approaches perform similarly.

2.4. Cross site scripting attacks

One of the most common web-based attack methods is cross-site scripting, XSS. XSS attack begins with code injection into a webpage. When a victim views this webpage, the injected code is executed without their knowledge. Potential results of the attack include: impersonation/session hijacking, privileged code execution, and identity theft.

Ismail et al. have detailed a XSS vulnerability detection mechanism by manipulating HTTP request and response headers [10]. In their system a local proxy manipulates the headers and checks if a website is vulnerable to an XSS attack and alerts the user.

Noxes, by Kirda et al., is a rule-based, client-side mechanism intended to defeat XSS attacks. The authors propose it as a application-level proxy/firewall with manual and automatically generated allow/deny rules [12].

Vogt et al. evaluate a client-side tool that combines static analysis and dynamic data tainting to determine if the user is transferring data to a third party [21]. If so, their Firefox extension asks the user if they wish to allow the transfer. An interesting question raised by this work is whether users could distinguish between a false positive and an actual attack.

2.5. Comparison to our approach

We largely view the related work summarized here as complimentary with our work. If a classifier can successfully detect malicious javascript, it may simplify the problem of developing policies or conducting taint analysis. For instance, one could develop a policy based on the results from a classifier that could take a number of actions, including disabling the malicious script, sending it to a central repository for further analysis or re-writing the malicious script to be benign. Most deobfuscation tools use dynamic analysis, which may slow down the web browsing experience more than static analysis, especially on websites with many scripts. Classifiers can also assist in identifying potentially malicious scripts so that deobfuscation tools can focus solely on them. XSS attack detection can also benefit from malicious javascript detection, as XSS frequently uses malicious javascript as part of the attack [20].

The advantage of using a classifier over the rule-based approaches is that a classifier will detect previously unseen instances of malicious scripts as long as they more closely resembles the malicious training set than the benign training set. If the script is using a previously unknown exploit but is obfuscated, then it is still likely to be detected even though a specific policy or rule has not been generated (potentially at a lower cost of overhead than dynamically re-writing code or keeping track of tainted data streams). Policies could even aid the browser to allow benign javascript misclassified as malicious (false positives generated by the classifier) to execute a subset of “safe” instructions, potentially allowing the user to proceed unimpeded even when the classifier has labeled a script as potentially malicious.

3. Machine learning and malicious javascript

This section provides an overview of the process of using machine learning classifiers to effectively distinguish between malicious and benign javascript. As mentioned in [4], the majority of malicious javascript is obfuscated and the obfuscation is becoming more and more sophisticated. We aim to detect obfuscated javascripts with a high-degree of accuracy and precision, so that we can selectively disable it or otherwise protect the user against online infections. The two essential phases of the process are data collection and feature extraction.

3.1. Data collection

The performance of any classifier is closely related to the quality of the data set used to train that classifier. The training dataset should be a representative sample of both benign and malicious javascripts so that the distribution of samples reflects the distribution in the Internet.

Benign javascript collection We conducted a crawl of a portion of the web using the Alexa 500 most popular websites as the initial seeds. The crawl was conducted using the Heritrix web crawler, the open source

Start date	January 26^{th}, 2009
End date	February 3^{rd}, 2009
Initial seeds	Alexa 500
Pages downloaded	$9,028,469$
Total domains	$95,606$
Data collected	$\sim$ 340GB (compressed)
Est. number of scripts	$\sim 63,000,000$

Table 1. Benign javascript crawl details

web crawler developed and used by Internet Archive to capture snapshots of the internet [9]. Details of the crawl are available in Table 1. We based this crawl on a template provided with Heritrix and extended the template to only download textual content while ignoring media and binary content. All told, the crawl gathered content from $95,606$ domains.

Examining a subset of the corpus using a python script, we observed an average of 7 external scripts per page, leading to our estimate that our corpus contains over 63 million scripts. Although this is a modest crawl by modern standards, this amount of information was more than sufficient to train the open source classifiers we used.

Malicious javascript collection While collecting examples of benign javascript is relatively straightforward, collecting examples of malicious javascript is far more complicated, primarily because malicious scripts are short-lived. The authors of malicious scripts have no interest in revealing their attack techniques and website operators have a vested interest in removing malicious scripts before site visitors are exposed. In order to collect live examples of malicious scripts, we created the system detailed in Fig. 2.

In step 1, we fed the Heritrix web crawler with URLs that had been blacklisted by anti-malware groups, such as lists from http://www.malekal.com and http://www.malwareurl.com. In step 2, Heritrix crawls these websites and saves the results in Heritrix's ARC (archive) format. The crawls typically resulted between 5 and 7 megabytes of data, although by the time of the crawl, most of the exploit code had been removed. In step 3, we used python scripts to extract individual scripts from the ARCs and in step 4 conducted a manual review of the scripts (we quickly discovered that most virus scanners do not detect web exploits).

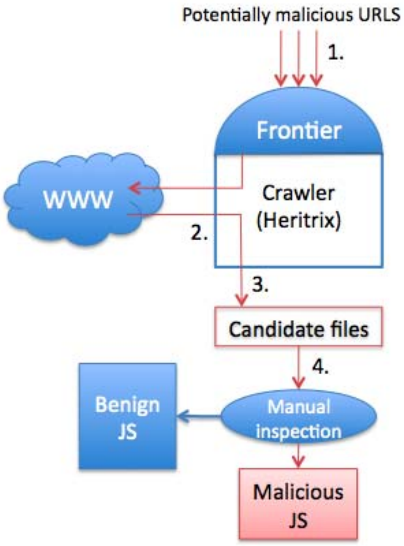


Figure 2. Malicious javascript workflow

This review involved deobfuscation of each malicious script in a clean VM using Venkman's javascript debugger [17].Scripts we identified as malicious were added to the collection of malicious javascript.

Over the course of several crawls conducted during February and March of 2009, we identified 62 malicious scripts. All but one of these scripts utilized a large amount of obfuscation.

Combined Data Set From the benign corpus, we extracted $50,000$ scripts at random. To this set we added the 62 malicious scripts to form the data set we used to train and test the classifiers.

3.2. Feature Extraction

The second major phase of our project consisted of identifying features based on an in-depth examination of javascript and a comparison of instances of benign and malicious javascript. As Fig. 1 reveals, it is simple for a human to visually discern the difference between the two classes. The challenge is codifying these differences as features that allow the classifiers to distinguish between them as well.

The simplest approach is to tokenize the script into unigrams or bigrams and track the number of times each appears in benign scripts and in malicious scripts. This approach has worked well with documents written in natural language, as the success of Bayesian classifiers and SVMs in spam filtering shows. However, javascript, a structured language with keywords, has a very different distribution of tokens from natural language. Tokenizing the scripts into unigrams (or bigrams) results in a huge number of features that only rarely appear in either benign or malicious javascript, resulting in a huge feature set with few meaningful features.

Using unigrams and bigrams also ignores the structural differences between the benign and malicious scripts and does not take advantage of the knowledge an expert might use to determine whether or not a script is malicious. For instance, we note that obfuscation often utilizes a non-standard encoding for strings or numeric variables (e.g. large amounts of unicode symbols of hexidecimal numberings). In turn, this tends to increase the length of variables and strings, as well as decreasing the proportion of the script that is whitespace. Examination of the malicious javascript also revealed a lack of comments.

We observed that malicious javascript contains a much smaller percentage of tokens that we termed "human-readable." We also posited that the use of javascript keywords tends to differ between malicious scripts and benign scripts, altering the frequency with which those words appear in the two classes. In order to capture this discrepancy, we used the normalized frequency (number of occurrences divided by the total number of keywords) of each javascript keyword as a feature. We used a total of 65 features (javascript keywords and symbols accounted for 50 of them). Table 2 lists the other 15 features we selected and briefly describes them. In section 5.

4. Classifier evaluation

We selected to evaluate the performance of the following classifiers: Naive Bayes, Alternating Decision Tree (ADTree), Support Vector Machines (SVM) and the RIPPER rule learner. All of these classifiers are available as part of the Java-based open source machine learning toolkit Weka [22]. A full description of these classifiers is beyond the scope of this paper but we present a brief summary of each along with any choices we made regarding the customization of the classifiers.

1. Naive Bayes: This classifier predicts whether the instance is benign or malicious using Bayes' Theorem with strong assumptions of independence between features. We used a kernel function to estimate the distribution of numerical attributes rather than a normalized distribution.

2. ADTree: This classifier is a tree of *decision nodes* and *prediction nodes*. Decision nodes consist of a condition. Decision nodes lead to prediction nodes that have positive or negative values based on whether the condition was satisfied. These values are added to the score, which determine if the script is benign or malicious. The tree uses pruning to select decision nodes. We used 50 iterations to generate potential decision nodes.

3. SVM: SVMs are a class of classifiers that draw a hyperplane in the feature space so as to maximize the distance between all instances of the two classes (benign and malicious). Weka incorporates Platt's *Sequential Minimal Optimization* (SMO) algorithm to train the SVM [15]. We normalized our data and used an Radial Basis Function (RBF) kernel with $\gamma = .0005$ and $C = 8.0$.

4. RIPPER: This is a propositional rule learner that greedily grows rules based on information gain and then prunes them to reduce error, proposed by Cohen [2].

4.1. Methodology

We extracted the features highlighted in Section 3 from each script and formatted the results as the Weka-specified ARFF file. In addition to the 65 attributes we added a final attribute: a nominal attribute signaling whether the script was malicious or benign. We conducted two experiments to evaluate and compare the classifier performance.

Experiment 1: training set validation We trained and evaluated each classifier using 10-fold cross validation (CV). In a 10-fold cross validation, the data set is partitioned into ten subsets. For each subset, the other nine are used to train the classifier and then the final subset is treated as the test set. CV is used to estimate how well the classifier would perform on unseen

Feature	Description
Length in characters	The length of the script in characters.
Avg. Characters per line	The avg. number of characters on each line.
# of lines	The number of newline characters in the script.
# of strings	The number of strings in the script.
# unicode symbols	The number of unicode characters in the script.
# hex or octal numbers	A count of the numbers represented in hex or octal.
% human readable	We judge a word to be readable if it is $> 70\%$ alphabetical, has $20\% <$ vowels $< 60\%$, is less than 15 characters long, and does not contain > 2 repetitions of the same character in a row.
% whitespace	The percentage of the script that is whitespace.
# of methods called	The number of methods invoked by the script.
Avg. string length	The average number of characters per string in the script.
Avg. argument length	The average length of the arguments to a method, in characters.
# of comments	The number of comments in the script.
Avg. comments per line	The number of comments over the total number of lines in the script.
# words	The number of "words" in the script where words are delineated by whitespace and javascript symbols (for example, arithmetic operators).
% words not in comments	The percentage of words in the script that are not commented out.

Table 2. Feature list and description, excluding reserved words in javascript.

instances (e.g. in the Internet) based on the consistency of classifier performance across the folds. We repeated the 10-fold CV 10 times for each classifier so we could test for statistically significant variations in classifier performance. Table 3 presents the results in terms of the CV common Machine Learning statistics for each of the four classifiers we define below.

- Precision: The ratio of (malicious scripts labeled correctly)/(all scripts that are labeled as malicious)
- Recall: The ratio of (malicious scripts labeled correctly)/(all malicious scripts)
- F2-score: The F2-score combines precision and recall, valuing recall twice as much as precision. This means we favor classifiers that identify larger percentages of malicious scripts even if they might label more benign scripts as malicious as well, due to our emphasis on security.
- Positive predictive power (PPP): PPP measures how often the classifier corectly predicts the positive (malicious) case. We omit PPP from our results as it is equivalent to precision.
- Negative predictive power (NPP): The ratio of (benign scripts labeled correctly)/(all benign scripts). This measures how often the classifier correctly predicts the negative (benign) case.

Classifier	Prec	Recall	F2	NPP
NaiveBay	0.808	0.659	0.685 (0.19)	0.996
ADTree	0.891	0.732	0.757 (0.15)	0.997
SVM	**0.920**	0.742	0.764 (0.16)	0.997
RIPPER	0.882	**0.787**	**0.806** (0.15)	0.997

Table 3. Performance in 10-fold CV. Standard deviation from the F2-score in parentheses.

All the classifiers in experiment 1 performed well on this data set. In particular, we note that $\sim 90\%$ of scripts labeled malicious by a classifier were malicious. Likewise, the NPP of the classifiers is extremely high: 99.7% of scripts labeled as benign are benign. The rule-based JRIP classifier had the highest recall (0.787) and F2-score (0.806) while the SVM had the highest PPP (0.92). Two-tailed, corrected paired t-tests revealed that no classifier performed better than the others (with regard to F2-score) at a statistically significant level. However, RIPPER's recall rate was better than that of NaiveBayes at a statistically significant level ($p = 0.05$).

The consistent performance across classifiers and folds, reflected in the F2-score and relatively low standard deviation, suggest that the feature set we generated captures differences between obfuscated, malicious javascript and benign javascript well. Next, experiment 2 shows that the classifiers detect malicious javascript "in-the-wild" as well.

Experiment 2: evaluating real-world performance Experiment 1 suggests that classifiers can distinguish between malicious and benign javascript. In order to validate this claim on the Internet, we used a new crawl of all domains blacklisted at http://www.malwaredomains.com/, and extracted all scripts from the crawl that were unique by MD5. Crawl statistics are reported in Table 4.

These $24,269$ scripts served as the test set. We used the models trained in Experiment 1 to classify these unlabeled scripts as benign or malicious. Results of Experiment 2 are presented in Table 5. A large number of scripts that are classified as malicious were unique in MD5, but manual inspection clearly showed that they were generated by the same obfuscation algorithm. This tended to artificially inflate the precision of the classifiers and so we counted each instance of a script generated by the same algorithm only once, removing duplicates. In the real web-surfing environment, users can expect higher precision as they are likely to encounter many duplicates.

Without knowing *a priori* the number of malicious scripts in the crawl, we cannot report recall or an F2-score for this experiment. Instead, we manually inspected all the scripts that were classified as malicious and identified a total of 22 malicious scripts that are unique by obfuscation algorithm. Again, manual inspection involved deobfuscating each script in a clean VM using Venkman's javascript debugger. All 22 of these malicious scripts were obfuscated.

Dates	June 2^{nd}-16^{th}, 2009
Initial seeds	827 blacklisted domains
Pages downloaded	$163,938$
Data collected	2.6GB (compressed)
Num of unique scripts	$24,269$

Table 4. In-the-wild javascript crawl details

Classifier	**# labeled**	**# mal**	**% all**
NaiveBay	19	17 (89.5%)	0.772
ADTree	22	17 (81.0%)	0.772
SVM	21	19 (86.3%)	0.864
RIPPER	28	19 (67.9%)	0.864

Table 5. Performance on a real-world data set

The high precision rates are consistent with Experiment 1's findings in Table 3, except that RIPPER had a lower precision than the other classifiers on this test set. The other classifiers provide confirmation that they are able to distinguish between benign and malicious javascript in the wild, a task which which has proven difficult in past research projects.

5. Discussion and conclusion

Our results indicate that machine learning classifiers, with deliberate feature selection, can produce highly accurate malicious javascript detectors. The experimental results in Exps. 1 and 2, in Tables 3 and 5, respectively, demonstrate they do not misclassify a significant number of benign scripts as malicious nor vice versa.

A malicious javascript detector using machine learning classification could provide several advantages to end-users and anti-malware researchers. First, this detector could proactively disable potentially malicious javascripts for better protection. The user interface may let users choose to execute certain scripts when needed in the case of a false alarm. Second, policy-based systems discussed in in Section 2 may use this detector as a guideline to trigger additional safeguards, such as restricting the script to a subset of trusted functions or invoking dynamic data tainting. Finally, these classifiers could be incorporated into honeyclients or honeypots designed to automate the collection and analysis of malicious scripts.

5.1. Drawbacks to using classifiers

Using classifiers to identify malicious scripts does have a drawback. Namely, classifiers are likely to categorize a small subset of benign scripts as potentially malicious. One example of benign and obfuscated javascript is *packed* javascript. Some websites choose to compress javascript before transmitting it to users to reduce the data transferred or prevent the theft of their source code. Packed javascript is the most likely to generate a false positive and may prevent users from browsing these websites. We suggest that a user interface should give the users the option of executing scripts selectively, and plan to conduct a usability test on disabling potentially malicious javascript as future work.

5.2. Feature Robustness

We also need to be concerned with feature robustness. If malicious script authors can easily design scripts that avoid our features, it defeats the purpose of using a classifier. In order to determine which features were most useful we calculated the chi-squared statistic for each feature in Weka. The five features most highly correlated with malicious scripts were: human readable, the use of the javascript keyword eval, the percentage of the script that was whitespace, the average string length and the average characters per line.

When attackers adapt their code to avoid features, it necessitates retraining the classifier and possibly developing new features. As future work, we aim to explore ways to make the most useful features more robust, particularly human readability. Currently, it is a rough heuristic at the moment and yet shows promise as the most distinguishing feature we tested.

References

[1] S. Chenette. The ultimate deobfuscator. http://securitylabs.websense.com/content/Blogs/3198.aspx.

[2] W. W. Cohen. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

[3] Computer Security Group at UCSB. Wepawet. http://wepawet.cs.ucsb.edu/.

[4] C. Craioveanu. Server-side polymorphism: Techniques of analysis and defense. In *3rd International Conference on Malicious and Unwanted Software*, 2008.

[5] M. Egele, E. Kirda, and C. Kruegel. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. *Detection of Intrusions and Malware*, Jan 2009.

[6] B. Feinstein and C. Peck. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. In *Blackhat*, 2007.

[7] O. Hallaraker and G. Vigna. Detecting malicious javascript code in mozilla. *Engineering of Complex Computer Systems*, Jan 2005.

[8] B. Harstein. jsunpack. http://jsunpack.jeek.org/dec/go/.

[9] Internet Archive. Heritrix. http://crawler.archive.org/, 2009.

[10] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. *Advanced Information Networking and Applications*, Jan 2004.

[11] M. Johns. On javascript malware and related threats. *Journal in Computer Virology*, Jan 2008.

[12] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *SAC '06: Proceedings of the 2006 ACM Symposium on Applied computing*, 2006.

[13] G. Maone. Noscript. http://noscript.net/, 2009.

[14] Mozilla.org. Spidermonkey. http://www.mozilla.org/js/spidermonkey/, 2009.

[15] J. C. Platt. *Fast training of support vector machines using sequential minimal optimization.* MIT Press, 1999.

[16] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Trans. Web*, 1(3):11, 2007.

[17] J. Ross and G. R. Venkman javascript debugger.

[18] C. Seifert, I. Welch, and P. Komisarczuk. Identification of malicious web pages with static heuristics. In *Australasian Telecommunication Networks and Applications Conference*, Jan 2008.

[19] Symantec. Web based attacks, 2009.

[20] The SANS Institute. Sans top-20 2007 security risks. http://www.sans.org/top20/.

[21] P. Vogt, F. Nentwich, N. Jovanovic, and E. Kirda. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium*, Jan 2007.

[22] I. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques, 2nd ed.* Morgan Kaufman, San Francisco, 2005.

[23] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*, Jan 2007.