ZDVUE: Prioritization of JavaScript Attacks to Discover New Vulnerabilities

Sandeep Karanth
Microsoft Research India
skaranth@microsoft.com

Srivatsan Laxman
Microsoft Research India
slaxman@microsoft.com

Prasad Naldurg
Microsoft Research India
prasadn@microsoft.com

Ramarathnam Venkatesan
Microsoft Research India
venkie@microsoft.com

J. Lambert
Microsoft Corporation

Jinwook Shin
Microsoft Corporation
jinshin@microsoft.com

ABSTRACT

Malware writers are constantly looking for new vulnerabilities to exploit in popular software applications. A successful exploit of a previously unknown vulnerability, that evades state-of-the art anti-virus and intrusion-detection systems is called a zero-day vulnerability. JavaScript is a popular vehicle for testing and delivering attacks through drive-by downloads on web clients. Failed attack attempts leave traces of suspicious activity on victim machines. We present `ZDVUE`, a tool for automatic prioritization of suspicious JavaScript traces, which can lead to early detection of potential zero-day vulnerabilities. Our algorithm uses a combination of correlation analysis and mixture modeling for fast and robust prioritization of suspicious JavaScript samples.On data collected between June and November 2009, `ZDVUE` identified a new zero-day vulnerability and its variant in its top results, as well as revealed many new anti-virus signatures. `ZDVUE` is used in our organization on a routine basis to automatically filter, analyze, and prioritize thousands of downloaded JavaScript files, for information to update anti-virus signatures and to find new zero-day vulnerabilities.

Categories and Subject Descriptors

K.4.1 [**Public Policy Issues**]: Abuse and crime involving computers; K.6.5 [**Security and Protection**]: Invasive software (e.g., viruses, worms, Trojan horses)

General Terms

Security, Algorithms, Experimentation

Keywords

Zero-day Vulnerabilities, Data Mining, Frequent Itemset Mining, Co-occurrence Statistics, Mixture Modeling, Malicious JavaScript Analysis, Drive-by Download Exploits

1. INTRODUCTION

JavaScript, a dynamic scripting language for client-side browser enhancements is increasingly being used to mount attacks on unpatched software on vulnerable client machines. Poorly validated JavaScript forms can cause such clients to download and execute malicious attack payload (including XSS vulnerabilities) from third-party servers. Such reflected attacks and drive-by-downloads account for over 60% of reported attacks [25], and a growing market for targeted cybercrime places a high premium on unearthing new (unpatched) vulnerabilities [10, 17, 29].

A new exploit of a previously unknown vulnerability is referred to as a *zero-day* attack. The associated vulnerability is called a Zero-Day Vulnerability (or ZDV). There are two main challenges confronting a malware writer crafting a zero-day attack: (i) The attack must evade state-of-the-art anti-virus scanners; (ii) For the attack to be effective, it must take into account the diversity of target configurations. Due to these factors, a malware writer may have to try several attack variants before developing a successful attack. Failed attempts may leave attack traces, in the form of malicious JavaScript code samples on victim machines.

In this paper, we show how these traces, farmed from millions of client computers around the world, can provide valuable information about potentially dangerous zero-day vulnerabilities, giving us an opportunity to patch the vulnerability before it is exploited. We present `ZDVUE`, a statistical tool that automatically analyzes such JavaScript attack samples to detect new vulnerabilities (not necessarily new methods of attack), including ZDVs, in a timely manner. Our goal is to correctly identify statistical indicators of attacks in the presence of large volumes of benign code samples, and use this to discover any new vulnerabilities that are targeted and/or any new anti-virus signatures that are generated. Our intuition is that it is possible to statistically characterize the features that distinguish malicious samples from benign. The difference may be small, but we can devise methods that magnify this difference, identify attack code with a high degree of confidence, and prioritize it according to its relevance.

Our approach identifies patterns that traditional anti-virus software can miss, and provides new signatures as inputs to such tools. Instead of wading through hundreds of thousands of files by hand to identify new attacks, `ZDVUE` helps a security expert in focusing effort on a small (tunable) prioritized list of high-ranking candidates. We trained our model on data collected between July and October 2009. Using this model, ZDVUE detected two ZDVs from among 26932 unannotated samples collected between August and November 2009. The ZDVs were found within the top 1% and 2% of our prioritized reports respectively. Subsequently, ZDVUE was incorporated in a diagnostic tool which is used on a day-to-day basis in our organization and regularly surfaces new vulnerabilities that are deemed important.

Our work is complementary to WEPAWET [6], where authors identify a suite of static and dynamic features for malicious JavaScript code and apply anomaly detection models to separate malicious cases from benign ones. Other related work includes static analysis techniques [5, 15, 8, 14, 11] that can identify attack payload in many cases, when the code is available ahead of time. However, this is not a complete solution for web applications that include dynamic content, such as context-sensitive advertisements. Some other approaches use high interaction honey clients [28, 20, 19, 22, 27, 4] to monitor dynamic system changes and identify malware. Both static and dynamic analysis techniques can be expensive to deploy on all test files, and we hope to leverage their benefits on our top-ranked files in the future to extend the performance of our tool.

The rest of the paper is organized as follows: Section 2 describes how we select features and presents our motivation for using co-occurrence statistics. In Section 3 we describe the training and testing phases of ZDVUE. Details of the statistical techniques used by our learning algorithm are presented in Sections 4-5. In Section 6 we describe our data collection and the results of applying our method to this data set, along with a sensitivity study. Section 7 presents related research, followed by conclusions and future work in Section 8.

2. MOTIVATION

We collect JavaScript samples from client machines, and look for code that corresponds to failed attack attempts. This information is embedded in our data, which consists of a large number of reports collected from client reported incidents, error reports, and submissions of malicious code. Note that we are looking at actual code that executed in a client context. The issue of obfuscated JavaScript code, which can fool signature scanners or static analyzers, is not relevant in our discussion. Our samples are code fragments that are in execution when the data is collected. Within these code samples, our first step is to identify a set of symbols, called our *alphabet*, which can help identify attack attempts and rank them according to their importance. Our alphabet consists of token symbols in JavaScript code samples (See Table 6 for example symbols in Appendix A). These symbols are picked semi-automatically, using frequency counting in hand-annotated samples. Our idea is to use frequent patterns (frequently occurring groups of symbols) as discriminators. In this section we motivate the use of frequent patterns for prioritization using an illustrative example.

Consider an example heap-spray attack that exploits three vulnerabilities, CVE-2008-4844, CVE-2008-0015, and CVE-2009-0075 in the Microsoft Internet Explorer browser program (See details of these attacks in Appendix B). An attacker mounting a heap-spray may use one or more new_array calls to allocate memory on the heap (symbol 28 in Table 1), or the unescape function to decode a JavaScript literal (symbol 20), or the substring function (symbol 18). There may also be traces of shellcode and NOPs (no operations), given by symbol ids 17 and 14 respectively. NOPs are frequently used by attackers to construct and deploy NOP sleds within the heap that end with user-inserted shellcode, and are used mainly to overcome address layout randomization challenges.

Id	Feature	% Count in malicious samples	% Count in benign samples	% Diff
14	u9090_u9090	61	0	61
17	0x0c0c0c0c	58	0	51
18	substr	95	88	7
20	unescape	85	45	40
23	function	97	99	2
26	replace	64	87	23
28	new_array	89	48	41

Table 1: Marginal statistics of heapspray symbols

The symbols in our alphabet represent (i) keywords and literals that are indicators of malicious JavaScript (e.g., symbols 14 and 17), (ii) keywords and literals that mainly occur in benign JavaScript, and (iii) a mix of keywords and literals that occur in both malicious and benign JavaScript (symbols 18,20, and 28). The mix of malicious and benign symbols characterize attacks more robustly. For example, if an attacker used new shellcode that does not map directly to symbol 17, the other benign symbols that deliver the shellcode will be detected by our tool. Using symbols directly as features, rather than groups of symbols (frequent patterns), to discriminate between malicious and benign JavaScript samples, would work well in scenarios where the individual statistics of these symbols are significantly skewed in the benign or malicious samples. This is not the case, as shown in Table 1, using data from 13500 attack samples.

Observe that symbols of type (i), i.e., symbol 14 indicating injection of NOP-sled blocks, and symbol 17, indicating shell code injection, each occur in about 60% of the attacks. Whenever 14 and/or 17 occur(s), the sample should be given a high score, marking it as suspicious. On the other hand, symbols like 23 (JavaScript class or function definition) have comparable frequencies in malicious and benign samples; hence they do not, by themselves, add much value in discrimination. Symbol 28 (memory allocation) and symbol 20 (decode string) both occur in $> 85\%$ of malicious samples, and also occur in close to 50% of benign samples. When used individually to characterize malicious samples, they can lead to a high false positive rates, i.e., we may identify benign JavaScript code samples as malicious. Similarly, symbol 26 (replace), is a useful discriminatory feature for benign code, but when used alone, can result in a high false negative rate.

We show how using co-occurrence statistics (frequent patterns) can overcome the issues highlighted above. In Table 2, we present the co-occurrence statistics of various combinations of a subset of these attack symbols: 20 - unescape, 26 - replace, and 28 - new_array. Observe that the combination of symbols i.e., pattern 20 and 28 is a stronger attack indicator (55% difference) than if they were considered individually ($40 - 41\%$ difference). Note that the combination of all three symbols (pattern $20, 26, 28$), has a lower

Pattern	Description	% Count in malicious samples	% Count in benign samples	% Difference
20, 26, 28	unescape, replace, new_array	49	23	26
20, 26	unescape, replace	51	37	14
20, 28	unescape, new_array	83	28	55
26, 28	replace, new_array	54	40	14

Table 2: Co-occurrence statistics of heapspray features

occurrence in malicious samples than any pair of symbols in this set. Similar trends are observed across other symbol statistics, indicating that i) co-occurrence is a stronger discriminator than individual symbol statistics and ii) not all co-occurrences have the same discriminating power. Based on (i), we prescribe a significance test to determine whether some pattern occurs with substantial statistical skew in the malicious examples as compared to the benign ones. Further, based on (ii) different patterns can contribute to different weight values. Using frequent patterns also builds in robustness to new variants of the same attack that may not be directly recognized by signature scanners that look for exact matches. In Section 3, we show how to select significant frequent patterns automatically, and assign appropriate weights for prioritization.

3. METHOD

Our goal is to identify high-value malicious JavaScript from our collected attack samples and uncover new vulnerabilities, or update existing anti-virus signatures. In the training phase (Section 3.1), we use a set of annotated samples to learn a probabilistic model for the data (*Algorithm 1*). In the ranking phase (Section 3.2), we prioritize the unannotated samples based on their likelihood scores with respect to the learnt model (*Algorithm 2*). We discuss the robustness of our approach to adversarial manipulation of both training and test phases in Section 3.3. We motivate our approach over other popular learning techniques in Section 3.4.

3.1 Training Phase

In the training phase, we are given annotated sets of positive (malicious) samples (S_+) and negative (benign) samples (S_-). Each JavaScript sample is encoded as a *transaction*, which is a collection of *items* or symbols over a finite alphabet (say $\mathcal{A}$). Recall that this set $\mathcal{A}$ (cf. Table 6) is the universal set of feature symbols, and were selected from our annotated samples statistically.

We define an *itemset* as a non-empty set of items or symbols, e.g., $\alpha = \{A_1 \ldots, A_N\},\ A_i \in \mathcal{A},\ i = 1, \ldots, N$ denotes an N-itemset. Itemsets can also be thought of as patterns embedded in transactions. Given a collection of transactions (say $\mathcal{D}$), the frequency of a pattern α in $\mathcal{D}$ is the number of transactions that contain (or support) α. Patterns whose frequency exceeds a user-defined threshold are referred to as *frequent itemsets*. The goal of the training phase is to estimate which groups of feature symbols correlate strongly in the positive examples but are absent (or are weakly correlated) in the negative examples, and vice versa. We also propose a new statistical significance test for itemsets, which determines whether an itemset was observed in data more often than we would expect under a random chance model.

The main steps in the training phase are listed in *Algorithm 1*. Input to the training phase are sets $\mathcal{S}_+$ and $\mathcal{S}_-$

Algorithm 1 Training algorithm

```
Input: Data set of positive samples ($\mathcal{S}_+$), data set of negative
    samples ($\mathcal{S}_-$), maximum size of patterns ($N$)
Output: Generative models $\Lambda_+$ (for $\mathcal{S}_+$) and $\Lambda_-$ (for $\mathcal{S}_-$)
1: Find the set ($\mathcal{F}_+$) of frequent itemsets in $\mathcal{S}_+$ of size $N$ or less,
    using a frequency threshold of $\frac{|\mathcal{S}_+|}{2^N}$
2: Find the set ($\mathcal{F}_-$) of frequent itemsets in $\mathcal{S}_+$ of size $N$ or less,
    using a frequency threshold of $\frac{|\mathcal{S}_-|}{2^N}$
3: Associate each pattern $\alpha_j$ in $\mathcal{F}_+ \cup \mathcal{F}_-$ with an IGM $\Lambda_{\alpha_j}$
    (based on Definition 1 in Section 4)
4: Keep only 'significant' patterns in $\mathcal{F}_+$ and $\mathcal{F}_-$ (based on
    Eq. 1)
5: Keep only 'discriminating' patterns in $\mathcal{F}_+$ and $\mathcal{F}_-$ by remov-
    ing patterns common to both
6: Build a mixture ($\Lambda_+$) of significant IGMs ($\Lambda_{\alpha_j}$, $\alpha_j \in \mathcal{F}_+$) for
    $\mathcal{S}_+$ (using Eqs. 3–5)
7: Build a mixture ($\Lambda_-$) of significant IGMs ($\Lambda_{\alpha_j}$, $\alpha_j \in \mathcal{F}_-$) for
    $\mathcal{S}_-$ (using Eqs. 3–5)
8: Output $\Lambda_+$ and $\Lambda_-$
```

of positive and negative training transactions, along with a user-defined maximum size N of patterns to be considered. The actual value of N is picked by the algorithm automatically, using our significance test. The first steps (lines 1, 2, *Algorithm 1*) compute the frequent itemsets for both data sets $\mathcal{S}_+$ and $\mathcal{S}_-$. The task of discovering *all* itemsets whose frequencies exceed a prescribed frequency threshold is a well-studied problem in data mining called Frequent Itemsets Mining (FIM) [1]. We use a standard procedure known as the Apriori algorithm for obtaining frequent itemsets in the data [1].

For each pair (pattern, frequency), we need a way of deciding if it is a useful discriminating statistic. Traditionally, this frequency threshold is estimated outside the model and can greatly impact the quality of the results if chosen incorrectly. We use a significance test to decide whether an itemset is interesting or not, rejecting all patterns whose frequencies are below a noise threshold – this corresponds to the likelihood of that the pattern being substantially smaller than the likelihood under a background noise model. One of our main contributions here is that we can prescribe a value for this threshold automatically that results in an effective classifier with strong statistical properties. In order to fix this threshold, we associate each frequent itemset discovered with a simple generative model called an Itemset Generating Model (or IGM) [13], as shown in line 3, *Algorithm 1* and devise a test of significance for frequent itemsets in the data based on likelihoods under the IGM model. Formal details of the IGMs and this connection are given in Section 4. Intuitively, each pattern is associated with an IGM with the following properties:

1. The probability of a transaction under an IGM is high

whenever the transaction contains the corresponding itemset.

2. Ordering of itemsets according to frequencies is preserved under ordering of corresponding IGMs according to their data likelihoods.

These properties give us a statistical significance test for itemsets in the data. (We observe that the IGM is not intended to be the best generative model that fits the data, as we are only interested in thresholding itemsets whose frequencies are not statistically significant). The significance test is essentially based on the evidence of at least one reasonable model (namely the IGM) over that of a uniform random source, and this turns out to be very useful in practice, eliminating automatically all itemsets that do not contribute to discriminating between malicious and benign classes. This is the next step in our training phase (line 4, *Algorithm 1*). For a given user-defined level of accuracy ϵ, a pattern α of size N with frequency f_α in a data set of K transactions, is declared *significant* if $f_\alpha > \Gamma$ where Γ is given by

$$\Gamma = \frac{K}{2^N} + \sqrt{\left(\frac{K}{2^N}\right)\left(1 - \frac{1}{2^N}\right)}\Phi^{-1}(1-\epsilon) \qquad (1)$$

where, $\Phi^{-1}(\cdot)$ denotes inverse of the cumulative distribution function (cdf) of the standard normal random variable. This is a standard error characterization for the standard normal random variable. For typical values of ϵ, size K of the given transactions data set and size N of the itemsets under consideration, the above expression tends to be dominated by $\frac{K}{2^N}$, and so, in the absence of any other information, we use this as the first threshold for significance to try in our algorithm. If the eventual model obtained is too weak (because of either too few or too many significant itemsets) we can always go back to the significance testing step and select an appropriate value of ϵ.

Note that while IGMs are useful to assess the statistical significance of a given itemset, no single IGM can be used as a reliable generative model for the whole data. This is because, a typical data set, $\mathcal{D} = \{T_1, \ldots, T_K\}$, would contain not one, but several, significant itemsets. Each of these itemsets has an IGM associated with it as per *Definition 1*. A mixture of such IGMs, rather than any single IGM, is a more appropriate model for $\mathcal{D}$.

The final step in our training phase is the estimation of a mixture of IGMs for each data set $\mathcal{S}_+$ and $\mathcal{S}_-$ (lines 6, 7, *Algorithm 1*). We can think of this step as an optimal iterative procedure that assigns weights to the significant patterns found in the data by maximizing the likelihood of the data under a mixture of IGMs. Details of this weighting technique are presented in Section 5. We use this procedure to estimate separate mixture models for $\mathcal{S}_+$ and $\mathcal{S}_-$. The resulting generative (mixture) models, Λ_+ and Λ_-, which are essentially the sum of the product of the learnt weights and the corresponding pattern frequencies, are the final outputs of the training phase (line 8, *Algorithm 1*).

3.2 Ranking Phase

We now describe the ranking phase of our procedure. The main steps in the ranking phase are listed in *Algorithm 2*. Inputs to the ranking phase are the generative models Λ_+ and Λ_- obtained from the training phase and the set $\mathcal{D}$ of test transactions that need to be prioritized. The first step is to compute, for each test transaction $T \in \mathcal{D}$, the likelihoods under the positive and negative generative models (lines 1–3, *Algorithm 2*). Then we compute the corresponding likelihood ratio (line 4, *Algorithm 2*). Finally, we sort the test cases in decreasing order of likelihood ratios (line 5, *Algorithm 2*). This sorted list is the final output of our ranking procedure (line 6, *Algorithm 2*).

Algorithm 2 Ranking algorithm

```
Input: Set $\mathcal{D}$ of test cases, learnt mixture models from training
    phase ($\Lambda_+$ and $\Lambda_-$)
Output: Prioritized list of test cases ($\mathcal{D}$)
1: for each test case $T \in \mathcal{D}$ do
2:     Compute likelihood $P[T \mid \Lambda_+]$ of $T$ in $\Lambda_+$ (based on Eq. (2)
       for $\Lambda_+$)
3:     Compute likelihood $P[T \mid \Lambda_-]$ of $T$ in $\Lambda_-$ (based on Eq. (2)
       for $\Lambda_-$)
4:     Obtain the likelihood ratio for $T$: $\frac{P[T \mid \Lambda_+]}{P[T \mid \Lambda_-]}$
5: Sort test cases in $\mathcal{D}$ in descending order of likelihood ratios
6: Output sorted list $\mathcal{D}$
```

After the ranking phase, we examine the JavaScript files corresponding to the top ranked transactions using our signature scanners. This output filtering step removes any known vulnerabilities. We observe that our method is useful in characterizing our transactions as follows:

- We can run our testing algorithm on the training set itself, to validate if our learnt model can recall all annotated transactions accurately. If any annotated transactions are classified otherwise (e.g., S_+ as S_-), an expert can inspect them and check if the samples are internally consistent. This is useful to check the correctness of the training set.

- After we run our prioritization and classification algorithm on the test data, and run our signature scanners and any other anti-malware protection on the top results, the remaining top results can either be attacks against new vulnerabilities, including ZDVs, or simply new signatures for known attacks that are not in our database yet. Either case will lead to a recommendation for updating our anti-malware signature database.

Note that though our results are sensitive to the choice of symbols (or findings exposed) in our alphabet, we *periodically* update our symbol set semi-automatically (with inputs form security experts), in response to trends observed in incident reports and advisories, as well as from the new signatures and vulnerabilities discovered by `ZDVUE`. While a completely new method of delivering a zero-day attack, which bears no statistical similarities to any existing attack, will not be identified, this information can be used retroactively in older data sets as well.

3.3 Robustness

We collect code-samples found in error-reports from millions of client computers around the world. As a result, the proportion of training data that the adversary can influence is very small. If the adversary harvests a relatively large infrastructure to deliver spurious error reports, these can be filtered-out using DDoS detection techniques. Hence it is difficult for an adversary to influence the patterns in the benign training class. However, since the malicious class is very small, it may be possible for the adversary to inject spurious benign patterns in a high proportion of malicious class examples. Note that this does not change the discriminating patterns. Rather than manipulate training data, the adversary can also try to tamper with test cases. By injecting high-weight benign patterns into some malicious test cases, the adversary can hope to wrongly reduce the likelihood ratio of these malicious samples (potentially increasing false-negatives). To counter this, we can simply use likelihoods under the malicious class as a robust test statistic, rather than likelihood ratios as mentioned in the algorithm (See Section 3.2). The other alternative for the adversary is to try and avoid using malicious patterns in a malicious test-case; but this amounts to discovering an entirely new attack with no resemblance to existing or previously-known attacks.

3.4 Other ML Methods

The key insight in `ZDVUE` is that co-occurrence statistics (or frequent patterns) play a significant role in distinguishing malicious reports from benign (see Sec. 2). With the space of all possible patterns exponential in the size of the alphabet, the dimension of feature vectors is be prohibitively high ($\approx 2^{34}$ for the alphabet in Table 6). While methods like decision-trees cannot handle such large feature spaces efficiently, more efficient techniques like Naive Bayes classifiers make strong independence assumptions contrary to our observation that some features tend to be strongly correlated. Hyperplane-classifiers such as SVMs, logistic regression and neural networks are sensitive to outliers [12] (especially when classes are very imbalanced), and an adversary can easily create outliers by introducing feature-noise in a small number of examples. In addition to outlier sensitivity, methods like SVMs are also known to be fragile when data exhibits class-imbalance (Recall that less than 0.06% of the samples we collected were malicious).Our approach based on estimates of frequent-patterns statistics easily scales to large-feature data and is also more robust to adversarial manipulation (Sec. 3.3). Further, by employing a statistical model based on class-conditional likelihoods, we avoid the pitfalls of learning from severely class-imbalanced data.

4. IGMS AND FREQUENT ITEMSETS

In this section we briefly recall the main results regarding the formal connection between itemsets and generative models [13].

An IGM Λ_α for itemset α is a model under which the probability of a transaction T is high whenever T contains α. An Itemset Generating Model (or IGM) is specified by a pair, $\Lambda = (\alpha, \theta)$, where $\alpha \subseteq \mathcal{A}$ is an N-itemset, referred to as the "pattern" of Λ, and $\theta \in [0, 1]$ is referred to as the "pattern probability" of Λ. The class of all IGMs, obtained by considering all possible itemsets of size N (over $\mathcal{A}$) and by considering all possible pattern probability values $\theta \in [0, 1]$, is denoted by $\mathcal{I}$. The probability of generating a transaction T under the IGM Λ is prescribed as follows:

$$P[T \mid \Lambda] = \theta^{z_\alpha(T)} \left(\frac{1-\theta}{2^N - 1}\right)^{1-z_\alpha(T)} \left(\frac{1}{2^{M-N}}\right) \qquad (2)$$

where, $z_\alpha(\cdot)$ indicates set containment: $z_\alpha(T) = 1$, if $\alpha \subseteq T$, and $z_\alpha(T) = 0$, otherwise; and M denotes the size of the alphabet $\mathcal{A}$. Observe that even for moderate values of θ (namely, for $\theta > \frac{1}{2^N}$) the above probability distribution peaks at transactions that contain α, and the distribution is uniformly low everywhere else. The *itemset-IGM association* is defined as follows:

Definition 1. *[13] Consider an N-itemset, $\alpha = (A_1, \ldots, A_N)$ ($\alpha \in 2^{\mathcal{A}}$). Let f_α denote the frequency of α in the given database, $\mathcal{D}$, of K transactions. The itemset α is associated with the IGM, $\Lambda_\alpha = (\alpha, \theta_\alpha)$, with $\theta_\alpha = (\frac{f_\alpha}{K})$, if $(\frac{f_\alpha}{K}) > (\frac{1}{2^N})$, and with $\theta_\alpha = 0$ otherwise.*

This itemset-IGM association has several interesting properties [13]. First, the association under *Definition 1* ensures that ordering with respect to frequencies among N-itemsets over $\mathcal{A}$ is preserved as ordering with respect data likelihoods among IGMs in $\mathcal{I}$.

Theorem 1. *[13] Let $\mathcal{D}$ be a database of K transactions over the alphabet $\mathcal{A}$. Let α and β be two N-itemsets that occur in $\mathcal{D}$, with frequencies f_α and f_β respectively. Let Λ_α and Λ_β be the corresponding IGMs (from $\mathcal{I}$) that are associated with α and β according to* Definition 1. *If θ_α and θ_β are both greater than $(\frac{1}{2^N})$, then we have, $P[\mathcal{D} \mid \Lambda_\alpha] > P[\mathcal{D} \mid \Lambda_\beta]$ if and only if $f_\alpha > f_\beta$.*

Further, if the most frequent itemset is 'frequent enough' then it is associated with an IGM which is a maximum likelihood estimate for the data, over the full class, $\mathcal{I}$, of IGMs. (We refer the reader to [13] for theoretical details of the model and its properties).

Theorem 2. *[13] Let $\mathcal{D}$ be a database of K transactions over alphabet $\mathcal{A}$ (of size M). Let α be the most frequent N-itemset in $\mathcal{D}$ and let $\Lambda_\alpha = (\alpha, \theta_\alpha)$ be the IGM corresponding to α (as prescribed by* Definition 1*). If $P[\mathcal{D} \mid \Lambda_\alpha] > \left(\frac{1}{2^N - 1}\right)^K \left(\frac{1}{2}\right)^{K(M-N)}$, then Λ_α is a maximum likelihood estimate for $\mathcal{D}$, over the class, $\mathcal{I}$, of all IGMs.*

5. MIXTURE OF IGMS

We now describe how a mixture of IGMs can be used as a generative model for the data. Let $\mathcal{F}^s = \{\alpha_1, \ldots, \alpha_J\}$ denote a set of significant itemsets in the data, $\mathcal{D}$. Let Λ_{α_j} denote the IGM associated with α_j for $j = 1, \ldots, J$. Each sequence, $T_i \in \mathcal{D}$, is now assumed to be generated by a mixture of the IGMs, $\Lambda_{\alpha_j}, j = 1, \ldots, J$. Denoting the mixture of IGMs by Λ, and assuming that the K transactions in $\mathcal{D}$ are independent, the likelihood of $\mathcal{D}$ under the mixture model can be written as follows:

$$\begin{aligned} P[\mathcal{D} \mid \Lambda] &= \prod_{i=1}^{K} P[T_i \mid \Lambda] \\ &= \prod_{i=1}^{K} \left(\sum_{j=1}^{J} \phi_j P[T_i \mid \Lambda_{\alpha_j}] \right) \qquad (3) \end{aligned}$$

where $\phi_j, j = 1, \ldots, J$ are the mixture coefficients of Λ (with $\phi_j \in [0, 1]\ \forall j$ and $\sum_{j=1}^{J} \phi_j = 1$). Each IGM, Λ_{α_j}, is fully characterized by the significant itemset, α_j, and its corresponding pattern probability parameter, θ_{α_j} (cf. *Definition 1*). Consequently, the only unknowns in the expression for likelihood under the mixture model are the mixture coefficients, $\phi_j, j = 1, \ldots, J$. We use the Expectation Maximization (EM) algorithm [2], to estimate the mixture coefficients of Λ from the data set, $\mathcal{D}$. Any other re-weighting technique can also work equally well.

Let $\Phi^g = \{\phi_1^g, \ldots, \phi_J^g\}$ denote the *current guess* for the mixture coefficients being estimated. At the start of the EM procedure, ϕ^g is initialized uniformly, i.e. we set $\phi_j^g = \frac{1}{J}\ \forall j$. By regarding ϕ_j^g as the prior probability corresponding to the j^{th} mixture component, Λ_{α_j}, the posterior probability for the l^{th} mixture component, with respect to the i^{th} transaction, $T_i \in \mathcal{D}$, can be written using Bayes' Rule:

$$P[l \mid T_i, \Phi^g] = \frac{\phi_l^g P[T_i \mid \Lambda_{\alpha_l}]}{\sum_{j=1}^{J} \phi_j^g P[T_i \mid \Lambda_{\alpha_j}]} \qquad (4)$$

After computing $P[l \mid T_i, \Phi^g]$ for $l = 1, \ldots, J$ and $i = 1, \ldots, K$, using the current guess, Φ^g, we obtain a *revised estimate*, $\Phi^{new} = \{\phi_1^{new}, \ldots, \phi_J^{new}\}$, for the mixture coefficients, using the following update rule. For $l = 1, \ldots, J$, compute:

$$\phi_l^{new} = \frac{1}{K} \sum_{i=1}^{K} P[l \mid T_i, \Phi^g] \qquad (5)$$

The revised estimate, Φ^{new}, is used as the 'current guess', Φ^g, in the next iteration, and the procedure (namely, the computation of Eq. (4) followed by that of Eq. (5)) is repeated until convergence.

6. EVALUATION

Using the methods detailed in the previous section, we have developed an automated tool called `ZDVUE` for JavaScript code prioritization that is being used daily by security audit groups in our organization. The results presented here are for browser attack data collected between July and November of 2009 [1]. We present the characteristics of the data collected, and report the zero-day and its variant that were bubbled up by the tool in Section 6.1. Enhancing signature-based scanners is a very important result of our tool and this is discussed in Section 6.2, followed by a comparison with another state-of-the-art tool `WEPAWET` in Section 6.3. Details of how we chose parameters in the training step, and the operating points for the classifier are presented in Section 6.4.

The characteristics of the data used in our evaluation are listed in Table 3. The *Train Data* i.e. the data that was used for training the model consists of 13500 samples of JavaScript code from execution contexts, which were filtered from a larger set of samples collected between July and October 2009. These samples were a small fraction of the data that correspond to browser attack attempts collected daily using customer reported incidents, submissions of malicious code, and Windows error reports over this period. Similarly, in the *Test Data*, i.e., the unranked samples that were of interest to security experts, contains 26932 instances selected using the same procedure, from August to November 2009. Note that both data sets do not overlap, and the code samples in the two sets are from different sources and distinct. The training data set was filtered based on a list of signatures of known attacks, which gave us the positive (malicious) labels. The negative (benign) labels were annotated semi-automatically with user input. We extract the patterns from each sample based on the alphabet of Table 6 and our significance test, and construct features appropriately. The average size of transactions (i.e. the average number of features per sample) is also listed in Table 3.

[1]Our data contain sensitive information from customer reported incidents. We only report results from the period that was cleared for disclosure

6.1 Detection of Zero-Day Attacks

From the 26932 JavaScript samples that were processed and prioritized, a zero-day attack exposing a vulnerability (ZDV-I) in the Microsoft Internet Explorer browser, unknown to our signature scanners or any other anti-virus software during data collection was ranked among the top 1%. Without `ZDVUE`, a security expert would have to manually examine the attack data i.e., inspect the code of all the files (26932), whereas our expert only needed to focus his attention on a smaller subset of 270 files. This attack attempt first surfaced on 29th November 2009 and was subsequently published as vulnerability CVE-2009-3672 in [18] (Appendix B). Further, when more ranked results were considered by lowering our selection threshold, a variant of the above attack, exploiting the same vulnerability (ZDV-II) was detected in the top 2% of the *Test data*. Guidelines for good threshold selection given a data set is discussed in detail in 6.4.

Table 4 shows the detailed results obtained on the test data. The first column mentions the threshold that was selected. The second column quotes the number of malicious cases reported for the corresponding threshold. The third column quotes the number of attacks (among those reported) that were already known to our anti-virus software. The fourth and fifth columns indicate whether `ZDVUE` detected the two instances of the Zero-day attack in the *Test Data*. As can be seen from the table, ZDV-I was detected at all the thresholds, while the second ZDV-II was only detected at A and B (but not at C or D). These thresholds are explained in 6.4, and correspond to interesting operating points in our tool. In all cases except threshold A, the number of interesting cases reported is small (just around 2% of 26932 cases). This demonstrates that our method is good at identifying relevant cases (i.e., within 1 and 2% at a reasonable threshold).

6.2 Signature Updates

Among our top ranked files, we found an attack that corresponds to known vulnerability in a JavaScript sample found in an Adobe file, corresponding to CVE-2009-0927 Appendix B. While the actual vulnerability was known to our anti-virus signature scanners, the method of delivery of this vulnerability was new, and our signature database was updated. Note that this vulnerability labeled SIG-I was found at all reasonable thresholds, as shown in Table 4. The top ranked files (over 300 were examined by hand) contributed to many such signature updates.

6.3 Performance

In the next experiment, we present performance of `ZDVUE` on annotated data. For this, we picked 100 files at *random* for which we established the ground truth by manually examining the files. We fed the 100 code samples to `ZDVUE` and the resulting prioritization is listed in Table 5. The top 7 ranked files (ranks 1–6) from `ZDVUE` were all actually interesting attacks, out of which the first file was the Adobe signature update described in the previous section. Note that all 100 files were also marked as benign by our anti-virus software. The 2 cases (45 and 46) that were not

Data Set	Date Range '09	All cases	Malicious cases	Benign cases	Avg. size of transactions
Train	Jul-Oct	13543	839	12704	11
Test	Aug-Nov	26932	453	26479	6

Table 3: Data characteristics

Operating Point	# Malicious Cases Reported	# Known Issues based on Sigs	ZDV-I	ZDV-II	SIG-I
A	5991 (22%)	286 (1.0%)	$\surd$	$\surd$	$\surd$
B	624 (2.3%)	240 (0.8%)	$\surd$	$\surd$	$\surd$
C	614 (2.2%)	237 (0.8%)	$\surd$	$\times$	$\surd$
D	560 (2.0%)	232 (0.8%)	$\surd$	$\times$	$\surd$

Table 4: Results for detection of Zero-Day Attacks in Test Data

Report no.	`ZDVUE` rank	Ground Truth	`WEPAWET` Classification
1	1	attack	suspicious
2	2	attack	benign
3	3	attack	benign
4	3	attack	suspicious
5	5	attack	benign
6	5	attack	benign
7	6	attack	benign
8 $\cdots$ 43	7 $\cdots$ 42	benign	benign
44	44	benign	benign
45	44	susp.	excl.
46	44	susp.	excl.
47 $\cdots$ 100	45 $\cdots$ 71	benign	benign

Table 5: Groundtruth Comparison with `WEPAWET`

prioritized well by `ZDVUE` were corrupted and could not be parsed correctly (incomplete files) and excluded from our comparison.

In Column 4 of Table 5, we compare our results on the 100 annotated samples against the results obtained using `WEPAWET` [6], an anomaly-based malicious JavaScript detector. Each input file is classified as one of malicious, suspicious or benign. `WEPAWET` identified the first and fourth samples as suspicious and the rest as benign. Since the features used in `WEPAWET` are behavior-based, and rely on actual execution of code samples in a controlled execution environment, this result is not unexpected. While the sample size picked was small, limited by the effort required to manually verify the ground truth, we believe that the results at least reflect the difference in approach between `ZDVUE` and `WEPAWET`. Finally, it is possible that `WEPAWET` can find attacks that our algorithm may miss and we expect that `WEPAWET` can be regarded as complementary to `ZDVUE` (see Sec. 7 for a discussion).

6.4 Training and Validation

During the training phase, we used the algorithm on the *Train Data* to learn the model. The maximum size of patterns was discovered as 5 by our algorithm, indicating that there were no significant patterns above this size for our training data. The first step in our evaluation is to validate the stability of the learnt classifier. This is a key step in the model-building phase, since, based on just the training data, we need to determine whether the training data was sufficient for the choice of parameters used. Validation involves randomly splitting the training set into 2 parts and using one

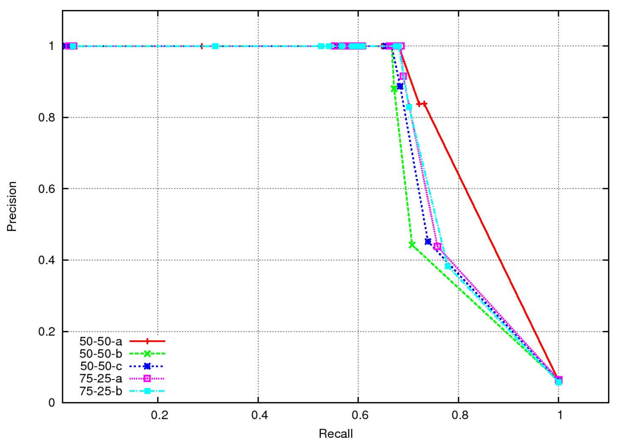

Figure 1: Cross validation results: Precision v/s Recall plots obtained by varying the classifier threshold for different splits of data into training and validation sets.

part for learning the model, and the second part for evaluating performance under the learnt model. We expect that if the statistical estimation stabilizes, we should get comparable results on different instances of random splitting of training data.

In our validation experiments, we generate 3 instances of random 2-way 50-50 splits and 2 instances of random 2-way 75-25 splits of the *Train Data* (These are referred to as 50-50-a, 50-50-b, 50-50-c, 75-25-a, 75-25-b in the figures). In all the instances, the first parts (of size 50% or 75%) were used to train the corresponding models and the second parts (of size 50% or 25%) were used for validation. The cross-validation results are plotted in Figure 1 in the form of precision v/s recall graphs. The different points in each curve correspond to the use of different thresholds on the likelihood ratios computed in line 4, *Algorithm 2*. The main observation is that the precision v/s recall behavior for all the instances are very similar – 100% precision can be achieved with a recall in the 65%-70% range. Then, as the threshold is relaxed (reduced) the algorithm will report more and more code samples as malicious, thereby decreasing the precision of the classifier decision. The interesting region in the operating characteristics of Figure 1 is at a precision of 90% and a recall of 70%, at the knee of operat-

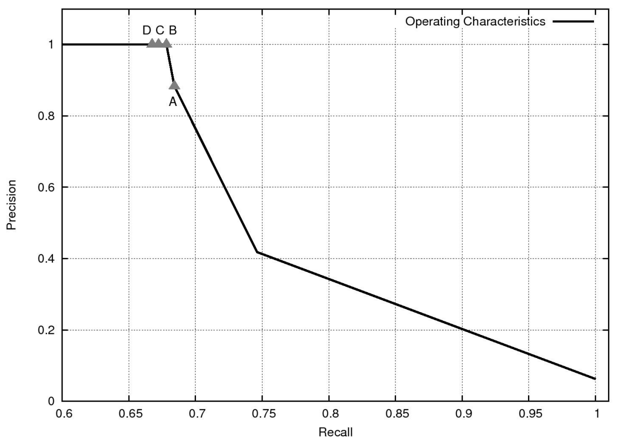

Figure 2: Finding an operating point for the classifier

ing curve (inflexion point). Moving substantially away from this knee would impact both precision and recall. Observe that operating points which achieve high precision (even if at the cost of some recall) are more important than those that achieve high recall (at the cost of precision). A high precision ensures that security experts do not have too many suspicious reports to analyze by-hand, i.e., 90% of the files that were labeled as attacks are indeed attacks. Moreover, since benign cases far exceed malicious cases, a high recall result (at the cost of lower precision) is trivial and useless to the security expert looking for new Zero-day attacks. In view of this, our result of 90% precision at 70% recall is very effective from the point-of-view of detecting new ZDV exploits. The important empirical result here is that such operating points were available (and identifiable) in all the instances we tested.

Finally, we plot the operating characteristic of the classifier using all of the *Train Data* in Figure 2 – 100% of the data was used to learn the model and the same data was used for plotting the operating curve as well. The general behavior is similar to the plots in Figure 1 (with marginally better precision since we are using all of the training data). We now select some suitable operating points for our classifier – in Figure 2, these points are marked A, B, C and D. We pick 4 points in and around the 'knee' of the operating curve. Each operating point corresponds to a threshold to be used for the likelihood ratios during testing, In particular, the thresholds for operating point A was 1.59, for B was 2.31, for C was 3.02 and for D was 3.74. The behavior of the ZDVs and signatures at these operating points was discussed earlier.

7. RELATED WORK

Machine learning and data mining techniques have been applied to various security problems [16], including anomaly detection, intrusion detection, attack graph analysis, and analyzing audit trails for root kits, etc. A comprehensive discussion of these techniques is beyond the scope of this paper.

Researchers have studied a variety of static and semi-static techniques to address the misuse of JavaScript language features [9]. These include (among others) (i) Blueprint [14], an enhanced parser on clients and servers that uses annotations and helps prevent unauthorized download, (ii) Staged Information flows [5], a combined static and dynamic analysis technique that generates residual checks that need to be validated at runtime, (iii) Gatekeeper [11] and [15], which concentrate on smaller safer subsets of JavaScript, (iv) Nozzle [23], where heapsprays are identified using a lightweight interpreter, (v) and in [8] where labels to sensitive information are attached and tracked. While these techniques are effective, they need changes to client or/and server components. Our work can be viewed as complementing these efforts, in rapidly identifying and prioritizing attack candidates for deeper analysis.

A number of high interaction honey clients [28, 20, 19, 22, 27, 4] have also been used to monitor system changes that could be caused by malware. In other related work, `WEPAWET` [6] is a web service that applies anomaly detection models for analyzing JavaScript code for malicious content. In [3] authors propose `PROPHILER`, a system intended as a front-end to `WEPAWET`, based on hand-crafted HTML, JavaScript, URL and host-based features. Our work differs from these in two ways. First, we look at syntactic features in attacks rather than behavioral features in JavaScript files, mitigating the issue of replicating the diversity of target configurations for attacks through honey clients. Second, in contrast to the independent anomaly detection models for each of the features, we learn statistically significant correlations among groups of features.

Another recently reported system that uses machine learning to detect drive-by-download attacks is `CUJO` [24]. Here, static and dynamic analysis is used to generate reports from which n-gram features are extracted and used for building SVM-based classifiers. Unlike n-gram features, which focus on specific sequential patterns in the reports, `ZDVUE` uses itemsets as features which are more general and can also detect correlations between tokens-at-a-distance. In [7], authors propose `ZOZZLE`, which applies a Naive Bayes classifier over hierarchical features of the JavaScript Abstract Syntax Tree (AST). While it is lightweight and thereby suitable for in-browser adoption, the Naive Bayes classifier makes an unrealistic assumption that features are statistically independent; by contrast, `ZDVUE` explicitly estimates the correlations between features and uses them for prioritization.

Finally, Argos [21] is an x86 emulator for capturing and fingerprinting disclosed ZDVs, to provide accurate input to signature scanners. The Zero-Day Initiative [26] maintains a current list of known ZDVs, which provides economic incentives to anyone reporting a new ZDV.

8. CONCLUSIONS

Detecting new vulnerabilities is a critical problem that confronts security response teams on an everyday basis. In this paper, we develop methods, which are effective in finding new vulnerabilities even before they are successfully exploited in the field. Our methods are based on a rigorous statistical characterization of previously known exploits. We use a combination of frequent pattern mining techniques from data mining and generative model estimation techniques from machine learning to develop a statistical filter for detecting malicious payload intended to exploit a new vulnerability. Our results demonstrate that our techniques are robust and are able to detect high-value ZDVs in real-world data sets. Although this paper primarily focuses on detecting ZDVs in JavaScript, our methods are applicable in other contexts, e.g., malicious audio and video files, Microsoft Word and Excel macro viruses, etc.

9. REFERENCES

[1] Agrawal, R., Imielinski, T., and Swami, A. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (May 1993), pp. 207–216.

[2] Bilmes, J. A gentle tutorial on the EM algorithm and its application to parameter estimation for gaussian mixture and hidden markov models. Tech. Rep. TR-97-021, International Computer Science Institute, Berkeley, California, Apr. 1997.

[3] Canali, D., Cova, M., Kruegel, C., and Vigna, G. Prophiler: A Fast Filter for the Large-Scale Detection of Malicious Web Pages. In *Proceedings of the World Wide Web Conference (WWW)* (Hiderabad, India, March 2011).

[4] CAPTURE. The honeynet project, Sept. 2008. `https://projects.honeynet.org/capture-hpc`.

[5] Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. Staged information flow for javascript. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (2009).

[6] Cova, M., Kruegel, C., and Vigna, G. Detection and analysis of drive-by-download attacks and malicious javascript code. In *WWW '10: Proceedings of the 19th international conference on World wide web* (2010).

[7] Curtsinger, C., Livshits, B., Zorn, B., and Seifert, C. Zozzle: Low-overhead mostly static JavaScript malware detection. In *Proceedings of the Usenix Security Symposium* (Aug. 2011).

[8] Dhawan, M., and Ganapathy, V. Analyzing information flow in javascript-based browser extensions. In *ACSAC'09: Proceedings of the 25th Annual Computer Security Applications Conference* (Honolulu, Hawaii, USA, December 2009), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 382–391. http://dx.doi.org/10.1109/ACSAC.2009.43.

[9] Felmetsger, V., Cavedon, L., Kruegel, C., and Vigna, G. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the USENIX Security Symposium* (Washington, DC, August 2010).

[10] Franklin, J., Paxson, V., Savage, S., and Perrig, A. An inquiry into the nature and causes of the wealth of internet miscreants. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (2007).

[11] Guarnieri, S., and Livshits, B. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the Usenix Security Symposium* (Aug. 2009).

[12] Guruswami, V., and Raghavendra, P. Hardness of learning halfspaces with noise. In *FOCS* (2006), pp. 543–552.

[13] Laxman, S., Naldurg, P., Sripada, R., and Venkatesan, R. Connections between mining frequent itemsets and learning generative models. In *Proceedings of the Seventh IEEE International Conference on Data Mining ICDM 2007* (Omaha, Oct. 2007), pp. 571–576.

[14] Louw, M. T., and Venkatakrishnan, V. N. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *IEEE Symposium on Security and Privacy* (2009), IEEE Computer Society, pp. 331–346.

[15] Maffeis, S., and Taly, A. Language-based isolation of untrusted javascript. In *CSF '09: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 77–91.

[16] Maloof, M. A. *Machine Learning and Data Mining for Computer Security: Methods and Applications (Advanced Information and Knowledge Processing).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[17] Miller, C. The legitimate vulnerability market: Inside the secretive world of 0-day exploit sales. In *In Sixth Workshop on the Economics of Information Security* (2007).

[18] Mitre. Common vulnerabilities and exposures database. `http://cve.mitre.org/`.

[19] Moshchuk, A., Bragin, T., Deville, D., Gribble, S. D., and Levy, H. M. Spyproxy: execution-based detection of malicious web content. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–16.

[20] Moshchuk, E., Bragin, T., Gribble, S. D., and Levy, H. M. A crawler-based study of spyware on the web. In *NDSS* (2006).

[21] Portokalidis, G., Slowinska, A., and Bos, H. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev. 40*, 4 (2006).

[22] Provos, N., Mavrommatis, P., Rajab, M. A., and Monrose, F. All your iframes point to us. In *SS'08: Proceedings of the 17th conference on Security symposium* (Berkeley, CA, USA, 2008), USENIX Association, pp. 1–15.

[23] Ratanaworabhan, P., Livshits, B., and Zorn, B. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the Usenix Security Symposium* (Aug. 2009).

[24] Rieck, K., Krueger, T., and Dewald, A. Cujo: efficient detection and prevention of drive-by-download attacks. In *ACSAC* (2010), pp. 31–39.

[25] SANS. The top cyber security risks 2009, Sept. 2009. `http://www.sans.org/top-cyber-security-risks/`.

[26] Tipping-Point. The zero day initiative. `http://www.zerodayinitiative.com/`.

[27] Wang, Y.-M., Beck, D., Jian, X., and Roussev, R. Automated web patrol: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the 13th Annual Symposium on Network and Distributed System security (NDSS'06), San Diego, USA* (2006).

[28] Wang, Y.-M., Beck, D., Jiang, X., and Roussev, R. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *NDSS* (2006).

[29] Wired.com. Threat level privacy, crime and security online previous post next post hack of google, adobe conducted through zero-day ie flaw. `http://www.wired.com/threatlevel/2010/01/hack-of-adob`.

A. JAVASCRIPT FEATURES

Table 6 lists the features that constitute our alphabet.

B. HEAPSPRAY ATTACKS

- CVE-2008-4844: Use-after-free vulnerability in mshtml (dll) in Microsoft Internet Explorer 5.01, 6, and 7 on Windows XP SP2 and SP3, Server 2003 SP1 and SP2, Vista Gold and SP1, and Server 2008 allows remote attackers to execute arbitrary code via a crafted XML document containing nested SPAN elements, as exploited in the wild in December 2008.
- CVE-2008-0015: Stack-based buffer-overflow in the function ReadFromStream (CComVariant) in the ATL as used in the MPEG2TuneRequest ActiveX control in msvidctl.dll in DirectShow, in Microsoft Windows 2000 SP4, XP SP2 and SP3, Server 2003 SP2, Vista Gold, SP1, and SP2, and Server 2008 Gold and SP2 allows remote attackers to execute arbitrary code via a crafted web page, as exploited in the wild in July 2009, aka "Microsoft Video ActiveX Control Vulnerability."
- CVE-2009-0075: Microsoft Internet Explorer 7 does not properly handle errors during attempted access to deleted objects, which allows remote attackers to execute arbitrary code via a crafted HTML document, related to CFunctionPointer and the appending of document objects, aka "Uninitialized Memory Corruption Vulnerability."
- CVE-2009-0927: Stack-based buffer overflow in Adobe Reader and Adobe Acrobat 9 before 9.1, 8 before 8.1.3 , and 7 before 7.1.1 allows remote attackers to execute arbitrary code via a crafted argument to the getIcon method of a Collab object, a different vulnerability than CVE-2009-0658.
- CVE-2009-3672: Microsoft Internet Explorer 6 and 7 does not properly handle objects in memory that (1) were not properly initialized or (2) are deleted, which allows remote attackers to execute arbitrary code via vectors involving a call to the getElementsByTagName method for the STYLE tag name, selection of the single element in the returned list, and a change to the outerHTML property of this element, related to Cascading Style Sheets (CSS) and mshtml.dll, aka "HTML Object Memory Corruption Vulnerability.

Id	Feature	Description	Id	Feature	Description
1	document.write	Write HTML expression to document	18	substr	Extracts characters from a string
2	evaluate	Evaluates and/or executes a string	19	shellcode	Shell code presence in JS string
3	push	Add array elements	20	unescape	Decode an encoded string
4	msDataSourceObject	ActiveX object for Microsoft web components	21	*u0A0A*_*u0A0A*	Injection of characters 0A0A
5	setTimeout	Evaluates expression after timeout	22	CompressedPath	Download path property on an ActiveXObject
6	cloneNode	Create object copy	23	function	Function or class definition
7	createElement	Create HTML element	24	shellexecute	Call Shell API
8	window	Current Document object	25	*u0c0c*_*u0c0c*	Injection of characters 0c0c
9	getElementbyId	Get an element from the current document	26	replace	Replace a matched substring with a string
10	object_Error	JS exception object	27	Collab.CollectEmailInfo	Adobe Acrobat method for for email details
11	*u0b0c*_*u0b0b*	Injection of character 0b0c0b0b	28	new_Array	Heap memory allocation
12	CollectGarbage	Call to garbage collector	29	*u0b0c*_*u0b0c*	Injection of characters 0b0c0b0c
13	SnapshotPath	Snapshot path property on an ActiveXObject	30	LoadPage	Load a HTML expression
14	*u*9090_*u*9090	NOP injection	31	createControlRange	Create a control container
15	new_ActivexObject	Instantiate ActiveX object	32	Click	JS click event
16	navigator.appversion	Get browser version	33	appViewerVersion	Get Adobe Acrobat Reader version
17	0x0*c*0*c*0*c*0*c*	Injection of 0c0c0c	34	function_packed	JS packer function

Table 6: JavaScript Features