Integrated Detection of Attacks Against Browsers, Web Applications and Databases

C. Criscione, G. Salvaneschi, F. Maggi, S. Zanero
Dipartimento di Elettronica e Informazione — Politecnico di Milano

***Abstract*—Anomaly-based techniques were exploited successfully to implement protection mechanisms for various systems. Recently, these approaches have been ported to the web domain under the name of "web application anomaly detectors" (or firewalls) with promising results. In particular, those capable of automatically building specifications, or models, of the protected application by observing its traffic (e.g., network packets, system calls, or HTTP requests and responses) are particularly interesting, since they can be deployed with little effort.**

Typically, the detection accuracy of these systems is significantly influenced by the model building phase (often called training), which clearly depends upon the quality of the observed traffic, which should resemble the normal activity of the protected application and must be also free from attacks. Otherwise, detection may result in significant amounts of false positives (i.e., benign events flagged as anomalous) and negatives (i.e., undetected threats).

In this work we describe Masibty, a web application anomaly detector that have some interesting properties. First, it requires the training data not to be attack-free. Secondly, not only it protects the monitored application, it also detects and blocks malicious client-side threats before they are sent to the browser. Third, Masibty intercepts the queries before they are sent to the database, correlates them with the corresponding HTTP requests and blocks those deemed anomalous.

Both the accuracy and the performance have been evaluated on real-world web applications with interesting results. The system is almost not influenced by the presence of attacks in the training data and shows only a negligible amount of false positives, although this is paid in terms of a slight performance overhead.

I. Introduction

In the field of computer security, without doubts the protection of web applications against attacks is a critical and current research issue. Web applications are gaining more and more popularity, due to their ease of use and development and to the ubiquity of the Internet — and in particular, the Web — in every day's life [1]. At the same time, they are usually developed with less attention to security constraints, due to different development models being employed; as a result, they have become the prime source of vulnerabilities in enterprise information systems. During 2006, the Web Application Security Consortium reported 148,029 different vulnerabilities affecting web applications: this translates to roughly 85% of the audited applications having at least one vulnerability [2]. Similarly, Symantec reported an increase equal to 125% of web application vulnerabilities between 2007 and 2008 [3].

Various taxonomies have been proposed for web threats, such as [4], [5], [6]. SQL injections seem to be the most commonly exploited attack vector. The goal of such attacks is usually either to control the server, or to obtain sensitive data. However, the current trend in web application attacks is the ever increasing rate of attacks carried out to compromise a host and use it for the distribution of malware (e.g., spy-ware, bots) or to deploy a phishing or spamming kit [7]. This does not come as a surprise, considering that PhishTank.com, for example, reports about 130,000 confirmed phishing websites over the same year. This shows how prevalent client-side attacks, such as the very common cross-site scripting, are becoming.

This creates a need for protection mechanisms to prevent the malicious content from being deployed on a host that runs a vulnerable web application. In addition, such a mechanism should avoid further spreading of the malicious content by protecting the visitors of a site already compromised. In this scenario, the challenge is that often attacks are not brought against known, off-the-shelf targets, but against custom applications. As such, they are by any definition zero-day attacks (i.e., that exploit vulnerabilities that are unknown before their use). This makes substantially ineffective the traditional and well developed concept of misuse detection, which is based on the exhaustive enumeration of all the *known* threats. On the other hand, anomaly-based techniques have the desirable property of protecting also against totally novel attacks. In fact, they model the *normal behavior* of the protected system (e.g., a web application) and detect deviations, called anomalies — under the assumption that attacks always cause anomalies. In this context, the term "normal behavior" typically refers to the set of features (e.g., the frequency of certain bytes in a network packet, the length of a string variable) extracted from the traffic, and then combined in such a way to build the models exploited to recognize anomalies (e.g., unexpected bytes frequencies, an out of bounds string length).

In this work, we describe Masibty, a web application anomaly detector that attempt to mitigate the two aforementioned major drawbacks (i.e., false positives due to inaccurate models and false negatives due to the presence of attacks in the training). Masibty is able to detect a real-world threats against the clients (e.g., malicious JavaScript code, trying to exploit browser vulnerabilities), the application (e.g., cross-site scripting, permanent content injection), and the database layer (e.g., SQL injection). A prototype of Masibty is evaluated on a set of real-world attacks against publicly available applications, using both simple and mutated versions of exploits, in order to assess the resilience to evasion. We can identify three key improvements in this paper:

- models are designed with the explicit goal of not requiring an attack-free dataset for training, which is an irrealistic requirement in real-world applications. Even if in [8] techniques are suggested to filter outliers (i.e., attacks) from the training data, in absence of a ground truth there can be no guarantee that the dataset will be effectively free of attacks. Using such techniques before training Masibty would surely improve its detection capabilities.
- Our approach intercepts and process both HTTP requests and responses and protects against both server-side and client-side attacks, an extremely important feature in the upcoming "Web 2.0" era of highly interactive websites based mainly on user contributed content. In particular, we devised two novel anomaly detection models — referred to as "engines" — based on the representation of the responses as trees.
- Masibty incorporates an optional data protection component, which extracts and parses the SQL queries sent to the database server. This component is part of the analysis of HTTP requests, and thus is not merely a reverse proxy to the database. In fact, it allows to bind the requests to the SQL queries that they generate, directly or indirectly. Hence, queries can be modeled although are not explicitly passed as a parameter of the HTTP requests.

In addition, Masibty has the advantages of a highly modular architecture, which easily allows to add additional detection engines based on new techniques. A similar modular approach was proposed in [9], however our architecture further explores the possibility of modularity in decoupling not only the engines aimed to anomaly identification, but also the modules that implement possible reactions to an anomaly, and the way in which information from the different engines is combined together.

II. Related works

Most of the traditional works on network intrusion detection focus on misuse-based or anomaly-based recognition of attack signatures. However, traffic generated from an attack to a web application — except for brute force attacks or similar events — is likely to be very similar to *normal* traffic because, since HTTP is a text based protocol, it is always possible to encapsulate an attack at application layer without creating a packet that is anomalous if inspected at network layer. Writing generic network-layer signatures for web-based attacks is thus troublesome, and a source of false positives. On the other hand, host-based IDSs were typically designed to monitor the processes on the protected system (e.g. the web server daemon) rather than the web applications they run. A successful example of protocol-aware anomaly detection based on low level data is presented in [10].

A more effective approach to the specific problem of anomaly detection for web applications is the inspection of user-supplied parameters. This problem is similar to recent developments in host-based anomaly detection on system calls, taking into account their parameters content. In [11], [12] a set of models were introduced to deal with various arguments (e.g., strings, integers, tokens). Using some of the concepts introduced in these seminal works, and extending them to incorporate sequence analysis and inter-argument relationships, a prototype named S^2A^2DE was proposed in [13], [14]. The concepts of [11] were ported to the web application context in [15], [9], by replacing the concept of system call with the URI of the requested resources, and the concept of system call parameter with the parameters passed to the URI handler. In other words, any web application is modeled as a set of URIs, each with an associated array of attributes. For each URI an ensemble of models is then generated. This approach, however, makes the prototype unable to distinguish between different behaviors of the same path, while it is common — especially for small-sized web applications — to rely on a single path to perform completely different tasks depending on the parameters' values. This issue can be mitigated through the use of clustering on arguments, as shown in [14]. A system based on a variation of this approach was proposed in [16], where the anomaly detection was performed on a reverse proxy — similarly to Masibty— and the application data was distributed between many databases. Anomalous queries are rerouted to databases containing information of lower sensitivity, accordingly to the degree of anomaly. The deployment of this system in the real world, however, would require extensive redesign to make the protected application resilient to missing data and, in addition, data must be separated according to its sensitivity. An interesting alternative to modeling parameters is proposed in [17], where kernel methods are exploited to model the features of the whole HTTP requests, not only of the parameters.

More specialized works have targeted separately XSS and SQL injection attacks. In [18] a client-side proxy was used to detect harmful content (e.g., DOM nodes) supplied by the user and sent back by the server, using a technique similar to the one implemented in web vulnerability scanners [19]. This technology, however, works only on reflected XSS attacks, and not on persistent attacks where the injected malicious code is permanently stored on the server-side and is delivered to the browser at a later time. In [20] a client-side solution based on the idea of data tainting is used to address the leakage of sensitive data from the user browser to an aggressor through the use of XSS attacks. Implementations of the same concept are also proposed in [21], [22]. However, nowadays' XSS attacks can perform more sophisticated tasks. Examples are the many attacks against social networking websites, which perform queries on the site without actually moving sensitive information around. The idea of a client side proxy was further exploited in [23]. It must be noted that all these techniques are client-side protections and, as such, they assume user awareness to security. In addition, they are not really anomaly-based as much as they are generalized misuse based system with broad rules to block specific attacks.

In [24] a method to identify variations of SQL query structures is proposed, by the means of a Java library which validates user-supplied parameters and compares the structure of each query before and after their insertion. The approach is interesting although it requires to modify large portions of code, since every line which contains SQL statements and queries needs to be rewritten manually, making the effort

similar to a full code review for implementing proper filtering in the application. In [25], [26], alternative learning-based approaches to the problem of detection of SQL injections are proposed. For instance, in [26] a server-side component is embedded in the web server, and analyzes SQL queries with techniques similar to [15], [12], generating models for user supplied input. However, to decrease false positives the developer must explicitly define database field types. This can be a lengthy process for complex applications. The major shortcoming of this architecture, however, is its inability to generalize the structure of a query: while most of the queries produced by web applications have a rather static look, thus allowing for exact profiling, there are many examples where the actual structure of a query is generated by user-supplied parameters. Since there is no way to learn the whole input space (as far as structure is concerned), no protection can be expected for these queries. An alternative approach, using static analysis, is presented in [27].

III. Masibty: a Framework for Web Application Intrusion Prevention

Besides the plus of being highly modular, Masibty is designed to minimize the impact on any existing infrastructure. More importantly, we specifically structured the system not to require an attack-free dataset for training, as this is a requirement not compatible with a real world deployment.

As depicted in Figure 1, Masibty is composed of two parts, both easily portable to different languages and platforms:

- a *reverse proxy*, which is a standalone application currently developed on top of the Jetty HTTP server;
- an *application database library* that monitors SQL calls. A proof-of-concept implementation was developed in PHP for MySQL databases and can be easily reimplemented for any other language, or extended to support other databases, since it consists in an extremely unobtrusive procedure.

An important feature of Masibty is that the proxy, by only interacting with the application in a black box fashion, can detect and block attacks targeted at both servers and clients. Although some client-side exploits can be identified just by examining the HTTP requests, in many cases analysis of the responses is needed to achieve decent levels of accuracy.

The application database library is optional and allows deeper analysis of SQL queries generated by the application. This is the only component of Masibty which is not language-agnostic. We investigated the feasibility of a reverse SQL proxy to avoid implementing a language-dependent component, but this would only allow to analyze queries in an isolated fashion, without binding them back to a specific HTTP request or user interaction. This approach is prone to false negatives whenever an aggressor is able to force the application to produce a query which would be legal in a different context, regardless of the anomaly detection model in use (this can be seen as a mimicry-like evasion attempt). Another possible alternative would have been to rewrite the actual embedded libraries, but this was complex beyond our purposes. It can certainly be done if the system is developed for production use. However it would shift the burden from modifying the applications to keeping up with a C code base under constant development.

For the aforementioned reasons, we implemented this library as a wrapper for the MySQL libraries for PHP, namely the `MySQLi` class and the `mysql_*` functions. A minimal effort is required to the administrator to alter slightly the application to be protected by modifying calls to `mysql_*` functions into `masibty_mysql_` invocations, and `MySQLi` objects into `MasibtyMySQLi` objects. Since the interface of every method has been respected, this changes can be applied through a trivial batch string replacement. Alternatively, specific features of the language such as function overriding or exception handling can be leveraged to achieve a fully automated, unobtrusive deployment by rerouting the calls of the functions in the original library to our library.

A. The concept of Entry Point

We modeled interactions between users and the protected application within the bounds of the HTTP protocol. In particular, our analysis is based on:

- URIs, e.g., `/blog/add/`, `/blog/read/`;
- parameters supplied, e.g., `?id=1&page=true`;
- session context[1], e.g., sequence of requests, cookies, session identifiers.

In addition, other influencing factors can be considered. For instance, multiple users might interact with the application in such a way to impact the current user, and so on. However, Masibty currently ignores such factors.

Starting from these observations, we defined the concept of an *Entry Point* (EP) as the basic entity of an application. An EP is basically a URI, further specialized depending on parameters and session context. Therefore, the relationship between a EPs and a URIs is not one-to-one, since in many applications use the same scripts (or classes) perform different tasks, according to the value of some parameters, of previous queries, sessions or other factors. For instance, an application may rely on a single `controller` that dispatches user interactions to the various components of the application depending on a `command` attribute in queries — while this is not a good software engineering practice, is a quite common situation. Clearly, this generates multiple EPs determined by the values of the `command`.

For the aforementioned reasons, Masibty works on EPs. The creation of EPs is therefore critical, and is delegated to two different procedures. The simplest one directly associates EPs to URIs. This is useful for small-sized applications, or if it is known a priori that the association is one-to-one (e.g., if URL rewriting is utilized). A more sophisticated procedure can be used to group similar requests together, so generating a set of EPs automatically. Its core is a clustering algorithm that must be incremental, unsupervised, and able to deal with categorical values. To this end, we used an agglomerative, incremental online algorithm [28]. The distance

[1] meant as the synopsis of all the previous interactions between the user and the application, encompassing all the data structures that have been built (thus including database and file updates and so on)

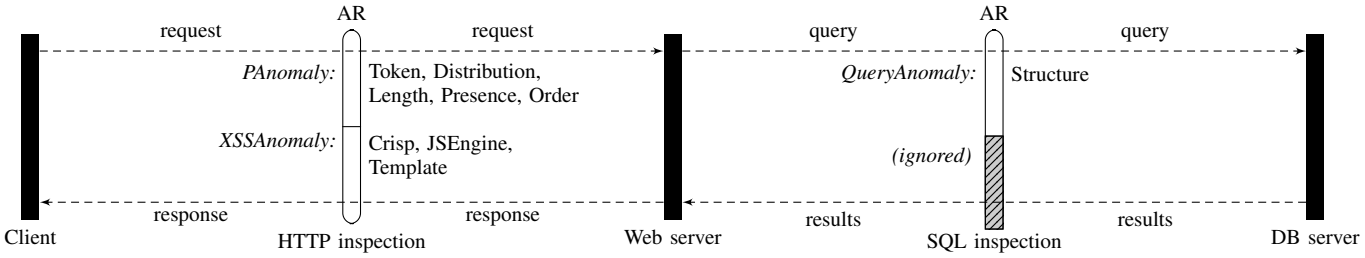


Figure 1. The logical structure of Masibty. Note that, the SQL inspection is visualized as a proxy just for clarity.

function between two URLs u and u' is the normalization in $[0,1]$ of $d(u,u') := \sum_{i=1}^{3} d_i(u,u')$. In particular, $d_1(u,u') := ||u|_{\mathrm{p}} - |u'|_{\mathrm{p}}|$ accounts for the number of parameters, $|\cdot|_{\mathrm{p}}$, in the URLs; $d_2(u,u') := \sum_{j=0}^{|u|_{\mathrm{p}}+|u'|_{\mathrm{p}}} \mathbf{1}_j$ counts the presence (i.e., 0) and absence (i.e., 1) of the j-th parameter; $d_3(u,u')$ is simple and accounts for the difference in length of each parameter found in the URLs. In addition, the algorithm prunes out clusters with limited support (e.g. those that contain a low number of instances) to cut out any outlier — possibly, an attack — in the training set.

B. Overall Architecture

The core component of Masibty is called *Anomaly Brain* (AB). It routes the HTTP requests and responses, captured by the proxy or the application database library, to a number of *Anomaly Reasoners* (ARs); this is performed either at learning time or during detection. Figure 1 shows the information flow and the ARs implemented. Requests or responses marked as anomalous are handled by specific *Reaction Managers* (RMs). ARs can be configured to be executed before or after the event is forwarded for processing. If no anomaly is detected by the pre-forwarding ARs, the action is let through (e.g., the request is forwarded to the web server, or the query is executed on the database). Next, it is routed to the post-forwarding ARs. If cleared, the responses are sent back to the client.

The ARs make use of different *Anomaly Engines* (AEs). Each of them models HTTP messages by means different features (i.e., string length, number of parameters, sequence of parameters). Thus, an AR can be effective at detecting anomalies in the parameters, whereas another may focus on client-side attacks. However, each AR has full access to any information available to all of the Masibty components (e.g., an AR working on SQL queries has access to the full session history). Since AEs work on EPs, requests coming from the reverse proxy are first passed through the aforementioned EP creation procedure, which clusters them (during learning) or classifies them (at detection time). Features learned on parameters by means of the AEs and used later in detection phase are stored in a model base.

During training, each AE self-assesses its reliability by calculating a trust level. During detection, the AEs generate an anomaly score in $[0,1]$ for each handled action. These outputs for AEs in the same AR are then combined to obtain a single anomaly score. Each AR can use a different policy to aggregate these values and can optionally take into account the trust level. The final anomaly score is then compared against an user-configured threshold to identify which events to flag as anomalies. Although it may seem reasonable to combine the output of all the AEs to obtain an overall anomaly value, it must be noted that each reasoner captures only a narrow subset of the information. Thus, an attack could be effectively recognized by a single AR. Instead, a combination could lead to the anomaly value being negatively balanced by another AR. For this reason, we use *Reaction Managers* (RMs) to handle independent actions to be performed after the detection of an anomaly by a single AE. These action may range from stopping the requests deemed malicious or simply reporting alerts.

Each AR can have multiple RMs, each with a different priority to allow handling of concurrent reactions. In particular, a RM can temporarily suspend any other RM with a lower priority, effectively blocking execution of lower priority reactions. Also, multiple different methods of reaction can be activated depending on different thresholds of the anomaly value.

IV. Reasoners and Engines

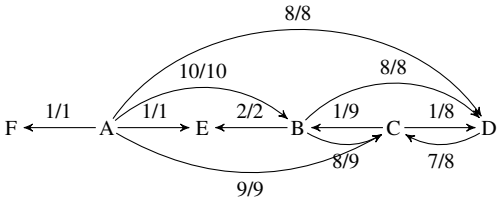
We have currently implemented three anomaly reasoners. *PAnomaly* and *XSSAnomaly* are built and used by the proxy component, while *QueryAnomaly* is built and used by the application database library. *PAnomaly* and *QueryAnomaly* are pre-forwarding AR, whereas *XSSAnomaly* is executed on HTTP responses.

A. PAnomaly

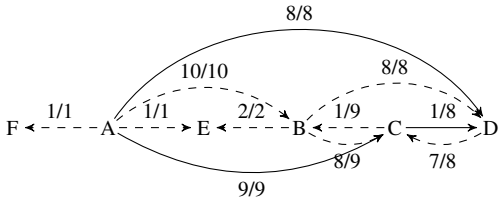
This AR detects anomalies in each request parameters and rely on different AEs.

1) Order Engine: Since requests in web applications are usually hard coded, whenever an EP is queried, the ordering of the attributes will usually be the same, even if not all of them are present. The *Order Engine* builds a probabilistic model using a directed graph that represents the order in which the parameters have been seen. Edges are labeled with the number of times the origin precedes the other node, P, and the number of times both nodes appeared in examined requests, T. As example is in Figure 2(a)

At detection, the active edges for every incoming request are identified. For instance, using the model in Figure 2(a), a request containing parameters A, C, D activates of the edges highlighted in Figure 2(b). The anomaly score is then computed by identifying the edge with the lowest ratio between the first and the second label. If an edge is completely missing, an anomaly score of 1 is returned, otherwise the anomaly score is



(a) Model at the end of the learning phase.



(b) Active edges (non-dashed) on a request that contain the parameters A, C, and D.

Figure 2. Two sample models generated by the Order Engine.

$1 - \min(\{\frac{P}{T}\})$. This algorithm has complexity $O\left(\frac{N \cdot (N-1)}{2}\right)$, since every edge of the induced subgraph has to be generated and evaluated. The worst case is a request with N elements.

The trust level takes into account how many infrequent couples are present, because if a couple has been seen a very small number of times there is too much variability to rely on the results of this engine. Therefore, we compute the trust level as $\text{avg}\left(\{\frac{P}{T}\}\right)$.

2) Presence Engine: Web applications usually handle a small set of parameters associated with a certain EP and it is unlikely that they will change, unless the client is trying to perform some unwanted interaction. This AE checks for expected or unknown parameters in each request. During training, the presence of each parameter is recorded, along with its appearance ratio across all the requests associated to the same EP.

Detection leverages the relative sample distribution of such ratios. The anomaly score is calculated as $1 - \min((\frac{M}{T}, \min(\{\frac{P}{T}\})))$, where M and P, respectively, indicates the number of missing and present parameters, while T is the total number of requests to the same EP. Thus, the presence of an unknown attribute or of a very rare attribute turns into a very high anomaly score.

The trust level is high if the presence of parameters is fairly constant, while decreases if an application exhibits variations. Thus, we calculate the trust level as $1 - \frac{M}{T}$.

3) Numbers Engine: Identifying those parameters that contain only numbers, which are extremely common, can stop a large share of injection-based attacks. To this end, during training, we store two values for each parameter: A, the number of times the attribute value was not a number, and the number of observations of the attribute T. If $X = \frac{A}{T}$ is close to zero, the value is likely numerical. Obviously attacks or application errors might have polluted the training set, so an exact zero is rare.

This engine leverages the *Yule-Simon* (YS) distribution [29] to associate a high anomaly score to very low values. We generate the anomaly score S using $S = \mathcal{YS}(\rho, X)$ so that only those parameters that are very likely to take numeric values can actually generate high anomaly scores. In our experiments we set $\rho = 200$.

The trust level, relies on how the anomaly score is calculated.

4) Token Engine: Sometimes a parameter only takes a limited set of values, usually referred to as tokens. This engine stores the admitted values and marks as anomalous any request containing parameters with out-of-the-enumeration values.

Token identification is performed using the algorithm described in [9]. Without going into the details, for each attribute a function is initialized to 0 and incremented by 1 whenever a new value for the attribute is seen, and is decremented otherwise. The procedure then estimates the correlation of this function *vs.* $y = x$, which models the fact that each item is a new, unseen value. Negative correlation indicates non-random values, i.e., tokens. The algorithm was trivially adapted to on-line usage, by updating the sample mean and variance on-line as opposed to in a batch fashion. In addition to the original algorithm, we count the relative occurrences of each *different* value.

Detection takes into account only those parameters identified as tokens. A high anomaly score is assigned to token values that have never, or seldom, been seen during learning. This is needed to minimize the influence of attacks in the training set. In fact, the engine could have observed an attack and included a malicious value in the allowed values for the token. However, such values are just a minority and can then be identified as the less frequent values a token has taken during training, and label them as anomalous anyways. To this end we resort again to the YS distribution, which is calculated for each observed value $v \in V$ leading to the anomaly score $S = \mathcal{YS}(\rho, \frac{N_v \cdot |V|}{N})$, where N_v is the number of observations of v, $N = \sum N_v$ is the total number of observations and V is the set of all possible values.

Basically, the expected rate of appearance of each parameter is estimated as the total number of observations for a certain parameter divided by the number of different values observed. Next, the ratio between the actual observation rate of a parameter and its expected rate is calculated as a value in $[0, |V|]$.

As we previously explained, high anomaly values are assigned only if the ratio between the expected and the observed rate of appearance is very low, according to the YS distribution.

The trust level is set to $\min(1, |max(-1, p)|)$. In other words, if the correlation parameter p is $p < -1$ (thus the attribute is very likely to be a token), we assign a value of 1 to the trust level. Otherwise, we assign a linearly decreasing value corresponding to the absolute value of the (linearly decreasing) correlation parameter.

5) Distribution Engine: The distribution of symbols is significant to distinguish the actual content of parameters that are expected to contain strings. For instance, a parameter may be designed to receive `10/14/2008 01:11AM` while an attacker could attempt to inject `' and t=t;`, which clearly have a different set of symbols. This engine captures such deviations by building a model of characters distribution through a

representation of the relative frequencies of occurrence. To this end, we adapted to online use the algorithm proposed in [9] to perform a variant of the Pearson χ^2-test to determine whether an observed value can be generated by the learned distribution. The anomaly score is $1-p$, where p is the p-value of the χ^2-test. The algorithm requires a single scan of the input and a constant-time calculation, its complexity being thus $O(n+k)$. An appropriate trust level of this model is planned as a future improvement. At the moment, this engine's trust level is 1.

6) Length Engine: Most of the parameters of a web application are not random in length. Some have fixed length (e.g., tokens, numeric identifiers), while some have a certain degree of variance. Only a few are completely random in length, most notably injection attempts. Long attributes are commonly associated with overflows, and also XSS attacks can be quite long. For instance, the shortest known XSS is 161 byte long [30]. This engine estimates the unknown length distribution for a given parameter in order to assess the anomaly of a parameter of length l in the detection phase.

Once again, we adapted the algorithm described in [9] to work online. No assumptions is made on the underlying distribution, which is specified by means of the sample mean μ and variance σ^2, calculated from training data. Detection is performed through the Chebyshev inequality, which determines an upper bound on the probability that the difference between the value of a random variable x and the mean of the distribution exceeds a certain threshold. Let t be the threshold $\mathrm{P}(|x-\mu|>t)<\frac{\sigma^2}{t^2}$. Therefore, the probability of a string of size greater than l is $\mathrm{P}(|x-\mu|>|l-\mu|)<\frac{\sigma^2}{(l-\mu)^2}$. Similarly to the previous engine, the trust level is fixed at 1 and an appropriate trust model is planned as a future work.

B. XSSAnomaly

This AR is aimed at detecting client side attacks. For example, JavaScript-based manipulation of the DOM or simple injection of contents into a web page, can be leveraged to completely change the client's perception of a page. A web site could be defaced on the client side, or a phishing site could overlap the original site, and so on. This reasoner detects anomalies in the embedded (i.e., not included as a separate file) code, and in the DOM. This allows to mitigate also more subtle threats such as client-side page defacement. This reasoner has to evaluate server response, thus is implemented as a post-query reasoner.

The DOM tree is constructed from the response using Gecko, a fast, open source parser and layout engine implemented in C++, and accessible through XPCOM APIs, wrapped by the Mozilla Parser Java library. The tree is then decorated with the JavaScript content of each node, while textual or otherwise non-JavaScript attributes are removed, keeping only *structural* information. The resulting structures, called *Anomaly Tree*, are used for both training and detection, which are detailed for each of the two engines described below. Depending on the AE adopted, two Anomaly Trees may be identical or different with a certain, numerical degree.

1) Crisp Engine: This engine detects anomalies in both DOM and JavaScript code. It utilizes the Anomaly Trees to learn the normal structure of pages associated with a given EP, assuming that requests to a single EP will be very similar to each other (e.g. a template filled in with variable information).

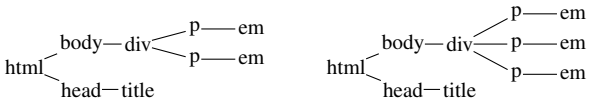


Figure 3. Two DOMs of two requests that only differ by the number of repetitions.

In general, two DOM nodes are deemed as equal if and only if both they match and their inline JavaScript code is identical, if any. This may arise issues with JavaScript generated dynamically (e.g., after a certain event), but makes the engine resilient to mimicry attacks.

During learning, the first Anomaly Tree is simply recorded. Subsequent trees are compared against the known ones. If a perfect match (i.e., identical tree) is found, a counter associated to each tree is incremented, otherwise the new tree is recorded. A peculiar characteristic of this engine is that it takes into account recurring content, frequent in data-centric web pages (e.g., search results or items in an online store). More precisely, trees are traversed in parallel and whenever a mismatch is found, the largest sub-tree is checked for descendants with identical structure. If a node causes a mismatch and such a node is not equal to the next one in the smaller tree — thus marking the end of the repetitions, the trees are deemed different and stored separately. Otherwise, the trees are considered identical, with a different set of repetitions as shown in Figure 3. This single-pass algorithm is linear with respect to the number of nodes of the largest tree.

Since any XSS injection is obtained by adding at least one element to the DOM, any Anomaly Tree with no matching learned trees is flagged as anomalous, with an anomaly score of 1.

The trust level for a given Anomaly Tree and EP is calculated during training as $1-\frac{D}{T}$, where D is the number of different Anomaly Trees and T is the total number of responses processed. If the ratio is low, and thus the number of total queries is far greater in comparison to the number of different Anomaly Trees, the AE can be trusted and thus it returns a value which is very close to 1.

2) Template Engine: This engine is meant to be adopted on highly-dynamic pages (e.g., forums, blogs, news aggregators).

During learning, Anomaly Trees are pruned by removing nodes with no JavaScript content, including their descendants. Then, a maximum number w of wild-card nodes are inserted; higher values of w lead to better accuracy on complex pages. This must be traded-off with a higher computational complexity. The algorithm works as follows: it substitutes one node a time (and its sub-tree) with a wild-card. Thus, if $w=1$ wild-card is allowed, a number of templates equal to the number of nodes n is generated, one with each node substituted by a wild-card. With $w=2$ this grows to $n\cdot(n-1)$ templates, with all the possible combinations of 2 wild-cards. During learning, this is done for each new Anomaly Tree. In case of a match with a previously known template, a counter associated to the template is incremented.

The learning algorithm is rather expensive as for each new

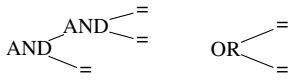


Figure 4. Two pruned trees used by the application database library to model an SQL query. The one on the right side is deemed anomalous.

tree is $O(n^2 + K \cdot n)$, with $w = 2$, where n is the number of nodes of the pruned Anomaly Tree and K is the number of known templates (the n^2 member is due to the template generation routine, whereas the $K \cdot n$ term is due to the comparisons against all the templates). The algorithm will always generate some fundamental templates (e.g. a template with just an `<html />` node plus a wild-card) that match all, or almost all, the response pages.

Detection is performed by testing the Anomaly Tree of any generated result for compatibility against all the templates built; the wild-card nodes validate any sub-tree starting at their positions. If a tree matches every template, as is the case for an EP with static content, a null anomaly score is returned. Otherwise, a numeric value is calculated using the observation rate of the highest non-matching template. The trust level is the frequency of the highest matching template, or 0 in case of EPs with no templates (i.e. no JavaScript code).

3) JS Engine: This AE uses a very simple technique to model JavaScript code. To this end, the MD5 of each code snippet extracted during the learning phase is stored. Although this approach may lead to false positives, it is effective for pages that reuse the same JavaScript code. For the same reason, it does not account for the code generated at runtime, also because this would require an excessive overhead due to the need of interpreting the JavaScript.

Learning is straightforward, and its complexity is linear with the number of JavaScript found in the new pages. During detection, we once again leverage the YS distribution to assign high anomaly scores to the MD5s that are infrequent in the training sets (i.e., those that are suspected of being outliers), as previously explained. In this case, the anomaly score is $X = \mathcal{YS}(\rho, \frac{|F|}{T})$ where F is the set of MD5s extracted from total number T of training responses that contain scripts. Clearly, the JavaScript that generate unknown MD5s is flagged as anomalous regardless of its rate of appearance in the training set.

The trust level is measured as $\text{avg}\left\{\frac{|F|}{T}\right\}$.

C. Application Database Library

This library analyzes the SQL queries before they are sent to the database and is implemented within the web application's scope. Hence, it has full access to the application data, e.g., which script was invoked, which script generated the query.

Currently, the only implemented AE is the *Structure Engine*, which relies on the parse tree of the queries. Contrarily to what was done in [24], no modification to the queries is required. In addition, as opposed to the method described in [27] based on static analysis, our technique is dynamic.

Constants or user-supplied data are filtered from the trees, while logical and arithmetic operators are kept. This may allow mimicry evasions (e.g., a query where only the names of the tables have been altered not detected as anomalous). However, SQL injections often alter the structure of the query dramatically.

Learning is performed by storing the trees corresponding to each EP along with their frequency. Detection is performed by comparing the tree obtained from the submitted query with the stored ones. If the tree does not match any of the known ones, the AE returns an anomaly score equal to 1; otherwise it is $\frac{R}{\overline{R}}$, where R is the number of times the matching tree has been observed, and $\overline{R}$ is the average number of appearance calculated over all the trees belonging to the same EP.

V. Experimental results

We evaluated both the detection capabilities and the processing overhead of Masibty on four real-world, PHP applications: Artmedic Weblog, SineCMS, PHP-Nuke, and JAF. The MySQL databases were manually populated with fake yet reasonable data that resemble as close as possible a real-world deployment. We used the Apache web server protected by Masibty, on Linux Ubuntu 8.10 running on a 2.50GHz machine with 4GB of RAM. In a real deployment, Masibty can be installed on dedicated machines.

Masibty was trained on the HTTP messages and SQL queries (PHP-Nuke only) generated during many interactions between clients and the application. More precisely: 6647 requests to Artmedic Weblog, 324 to SineCMS, 1310 to PHP-Nuke, and 902 to JAF. During training, we have tried to emulate both regular users and administrators. To test the resilience to outliers, 1% of the requests were actually attacks that were generated as follows. The exploits for the vulnerabilities were selected by carefully monitoring the bugtraq mailing list during late 2008. In addition, mutated versions of the attacks were generated manually. Attacks included XSS attempts (e.g., we used CVE-2006-0676 for PHP-Nuke), remote file inclusions (e.g., we used CVE-2006-7128/6142 for JAF-CMS) and SQL injections (e.g., we used CVE-2006-5525 for PHP-Nuke). The large majority of these attacks were used to build the testing dataset.

Results are summarized in Table I. On simple applications, such as Artmedic Weblog and SineCMS, all the attacks inserted were identified, with no false positives. Suspecting overfitting, the results were manually inspected, and further mutated versions of the attacks were inserted. Surprisingly, no evasion attempt succeeded. On PHP-Nuke Masibty reported no false positives and a non-negligible amount of false negatives on some XSS attacks. Since JAF stores data on a flat file, the SQL module was disabled. Nevertheless, the proxy module has successfully recognized all 16 attacks. In JAF, an administrator can include external HTML pages created. We exploited this feature and submitted some rather complex pages also containing JavaScript — obviously, training and testing dataset contained a different set of pages. This caused 0.38% of false positives. In all the cases but PHP-Nuke the attacks were all detected by the *XSSAnomaly* and *PAnomaly* reasoners, which both contributed to create an anomaly score beyond the thresholds. In addition, the SQL injections against PHP-Nuke were detected by the *QueryAnomaly* reasoner.

Globally, Masibty detected 95.75% of the attacks with 0.095% of false positives. For comparison with systems that

APPLICATION	TOTAL REQ.	ATTACK REQ.	DR	FPR
Artmedic	3357	16	100%	0.0%
SineCMS	442	4	100%	0.0%
PHP-Nuke	1200	24	83%	0.0%
JAF	800	16	100%	0.38%
Overall	5799	60	95.75%	0.095%

Table I
DETECTION CAPABILITIES FOR EACH APPLICATION. THE TOTAL REQUESTS INCLUDE THE MALICIOUS REQUESTS.

were tested with a an attack-free training, we also ran an additional test using a filtered training dataset and with no evasion attempts. Under such rather irrealistic hypotheses, Masibty detected 100% of the attacks with no false positives.

We also measured the throughput and the processing overhead introduced by Masibty. To this end we first recorded an 8-step navigation session so that each virtual client resembled a human user. This generated 4 HTTP requests including HTML content and images, making 32 HTTP requests overall. The users were idle for between each interaction for a random small amount of time. This resulted in $34s$ of idle time per session. Then, we reproduced a closed queuing system with an increasing number of customers, using HP LoadRunner with the following workload profile: a gradually increasing number of clients from 0 to 30 for the first 3 minutes. Then, a constant number of clients for 5 minutes, and zero during the last 2 minutes. The averages response time of the base system is $0.01s$ per request. We measured an average $0.02s$ overhead introduced by Masibty. Not surprisingly, the detection capabilities of the prototype are paid at the price of a non-negligible overhead. However, a significant part of the overhead can be reduced by re-implementing the tool using a lower-level language, such as ANSI C, and by decoupling the detection phase from the blocking phase, and make the former working in passive mode.

VI. CONCLUSIONS

In this work we described Masibty, a prototype web application firewall that has some interesting features and show significant improvements with respect to other existing tools. It can work under realistic assumptions (i.e., attack-free training data) and can deal with applications with a complex structure because of its sophisticated URL modeling algorithm. We described and implemented an extensible, modular architecture for the prototype, as well as a number of anomaly detection models. We described a proxy module which is able to identify both anomalies in parameters passed to the web application, and anomalies in the structure of the resulting pages, thus protecting the clients from malicious content. Also, we implemented a PHP library that contains a set of models for detecting anomalies in SQL queries through structural analysis.

Some of the techniques for server-side analysis of both web pages and SQL queries described in this paper are innovative contributions. Also, we improved previous works and proposed simplified but effective learning algorithms. We have performed *preliminary* testing of our solution on four real-world applications, obtaining promising results and confirming the effectiveness of our approach. The overhead introduced is, however, non-negligible but we believe it is mostly due to the poorly-optimized prototype.

Future works include extensive testing and recording of each model's contribution to the anomaly score, which are missing in our preliminary experiments. Furthermore, we are devising an automatic mechanism for choosing between one-to-one and many-to-many association between URIs and EPs. We are also currently working on a reasoner able to perform anomaly detection on headers and cookies, and a session-tracking mechanism which would allow to take into account the sequence of pages and queries performed by a single user. Finally, we are testing the negotiation techniques we proposed in [31] as an aggregation policy for the anomaly score.

REFERENCES

[1] Miniwatts Marketing Grp., "World Internet Usage Statistics," http://www.internetworldstats.com/stats.htm, January 2009.

[2] M. Sutton, J. Grossman, S. Gordeychik, and M. Khera, "Web application security consortium statistics," Available online at http://www.webappsec.org/projects/statistics/.

[3] D. Turner, M. Fossi, E. Johnson, T. Mark, J. Blackbird, S. Entwise, M. K. Low, D. McKinney, and C. Wueest, "Symantec Global Internet Security Threat Report – Trends for 2008," http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xiv_04-2009.en-us.pdf, Symantec Corporation, Tech. Rep. XIV, April 2009.

[4] J. Grossman and O. Shezaf, "Threat classification," Web Application Security Consortium, Tech. Rep., 2005.

[5] The Open Web Application Security Project, "The ten most critical web application security vulnerabilities," Available online at www.owasp.org.

[6] O. S. et al., "The web hacking incidents database annual report," Web Application Security Consortium, Tech. Rep., 2007.

[7] J. E. Dunn, "Do-it-yourself phishing kit found online," Available online at http://www.pcworld.com/article/128524/doityourself_phishing_kit_found_online.html, Jan 2007.

[8] G. F. Cretu, A. Stavrou, M. E. Locasto, S. J. Stolfo, and A. D. Keromytis, "Casting out demons: Sanitizing training data for anomaly sensors," *Security and Privacy, IEEE Symposium on*, vol. 0, pp. 81–95, 2008.

[9] C. Kruegel, G. Vigna, and W. Robertson, "A Multi-model Approach to the Detection of Web-based Attacks," *Computer Networks*, vol. 48, no. 5, pp. 717–738, August 2005.

[10] Y. Song, S. J. Stolfo, and A. D. Keromytis, "Spectrogram: A mixture-of-markov-chains model for anomaly detection in web traffic," in *Proc. of the 16th Annual Network & Distributed System Security Symposium*, San Diego, CA, USA, February 2009.

[11] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the Detection of Anomalous System Call Arguments," in *Proceedings of the 2003 European Symposium on Research in Computer Security*, Gjøvik, Norway, October 2003.

[12] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous system call detection," in *ACM Transactions on Information and System Security*, vol. 9, 2006, pp. 61–93.

[13] S. Zanero, "Unsupervised learning algorithms for intrusion detection," Ph.D. dissertation, Politecnico di Milano T.U., Milano, Italy, May 2006.

[14] F. Maggi, M. Matteucci, and S. Zanero, "Detecting intrusions through system call sequence and argument analysis (preprint)," *IEEE Transactions on Dependable and Secure Computing*, vol. 99, no. 1, 2009.

[15] C. Kruegel and G. Vigna, "Anomaly Detection of Web-based Attacks," in *Proceedings of the* 10^{th} *ACM Conference on Computer and Communication Security (CCS '03)*. Washington, DC: ACM Press, October 2003, pp. 251–261.

[16] F. Valeur, G. Vigna, C. Kruegel, and E. Kirda, "An Anomaly-driven Reverse Proxy for Web Applications," in *Proceedings of the ACM Symposium on Applied Computing (SAC)*, Dijon, France, April 2006.

[17] P. Düssel, C. Gehl, P. Laskov, and K. Rieck, "Incorporation of application layer protocol syntax into anomaly detection," in *ICISS '08: Proceedings of the 4th International Conference on Information Systems Security*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 188–202.

[18] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi, "A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability," in *International Conference on Advanced Information Networking and Applications*, 2004, p. 145.

[19] T. Gallagher, "Automated detection of cross site scripting vulnerabilities," European Patent Application EP1420562 (pending), October 2003.

[20] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis," in *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2007.

[21] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically Hardening Web Applications Using Precise Tainting," in *20th IFIP International Information Security Conference*, Makuhari-Messe, Chiba, Japan, June 2005.

[22] T. Pietraszek and C. Berghe, "Defending against Injection Attacks through Context-Sensitive String Evaluation," in *Recent Advances in Intrusion Detection (RAID)*, 2005.

[23] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: A client-side solution for mitigating cross-site scripting attacks," in *Proceedings of the* 12^{th} *ACM Symposium on Applied Computing*, 2006.

[24] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using parse tree validation to prevent sql injection attacks," in *SEM '05: Proceedings of the 5th international workshop on Software engineering and middleware*. New York, NY, USA: ACM, 2005, pp. 106–113.

[25] C. Bockermann, M. Apel, and M. Meier, "Learning sql for database intrusion detection using context-sensitive modelling," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. Volume 5587/2009. Springer Berlin / Heidelberg, 2009, pp. 196–205.

[26] F. Valeur, D. Mutz, and G. Vigna, "A Learning-Based Approach to the Detection of SQL Attacks," in *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Vienna, Austria, July 2005, pp. 123–140.

[27] W. G. J. Halfond and A. Orso, "Amnesia: analysis and monitoring for neutralizing sql-injection attacks," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM, 2005, pp. 174–183.

[28] J. Han and M. Kamber, *Data Mining: concepts and techniques*. Morgan-Kauffman, 2000.

[29] H. Simon, "On a class of skew distribution functions," *Biometrika*, vol. 42, no. 3-4, pp. 425–440, 1955.

[30] Sla.ckers, "Diminutive xss worm replication contest," Available online at http://sla.ckers.org/forum/read.php?2,18790,18790, 2008.

[31] F. Amigoni, F. Basilico, N. Basilico, and S. Zanero, "Integrating partial models of network normality via cooperative negotiation: An approach to development of multiagent intrusion detection systems," pp. 531–537, 2008.