Masibty: an anomaly based intrusion prevention system for web applications

Claudio Criscione
Secure Network Srl
c.criscione@securenetwork.it

Stefano Zanero
Politecnico di Milano, Dip. Elettronica e Informazione
stefano.zanero@polimi.it

March 27, 2009

Abstract

In this whitepaper we briefly describe Masibty, a novel anomaly-based web application firewall we devised. It has a modular and extensible structure. We give an overview of the anomaly detection models we implemented in it, and show that it is able to detect different kind of real world attacks on common web applications. We also evaluate its performance as an IPS, using both plain and mutated versions of exploits.

1 Introduction

Protection against web application attacks is a critical and current research issue. Today, they are the primary source of vulnerabilities in enterprise information systems. During 2006, the Web Application Security Consortium (WASC) reported 148,029 different vulnerabilities affecting web applications: this translates to roughly 85% of the audited applications having at least one vulnerability [1].

This creates an obvious need for intrusion prevention techniques for web applications, often dubbed "web application firewalling". The challenge is that in the web application domain often attacks are not brought against known, off-the-shelf applications, but against custom applications. As such, they are by any definition zero-day attacks (unknown to the target and the security community before their use). This makes the traditional concept of *misuse detection* useless, and creates a need fot anomaly-based detection systems that can protect against totally novel attacks, as they model the *normal behavior* of a system and detect deviations (as opposed to modeling the attacks themselves).

In this whitepaper, we describe Masibty, an anomaly based web application firewall prototype which we developed. It is a modular and extensible anomaly

detection system able to model the normal behavior of web applications, and to detect a number of types of attacks on the application layer. We describe the design of the system, from the preliminary definition of a generic web application behavior model to the creation and composition of a number of models for intercepting anomalous behavior deviations. We evaluate the system on a set of real world attacks against publicly available applications, using both simple and mutated versions of exploits.

The remainder of the work is structured as follows: in Section 2 we review related works in the area. In Section 3 we give a general overview of Masibty, along with its basic structure and assumptions. In Section 4 we describe the Anomaly Reasoners and Anomaly Engines we implemented. Section 5 details our experimental methodology and performance results. Finally, in Section 6 we draw our conclusions and outline some future works.

2 Related works

Most traditional works on Network Intrusion Detection focus on patterns of traffic (either by using misuse or anomaly detection over them). However, traffic generated from a web attack - except for brute force attacks or similar events - is likely to be very similar to normal traffic. Writing "generic" signatures for web based attacks is also known to be troublesome, and a source of false positives. On the other hand, host based IDS were typically designed to inspect the behavior of processes on the system, and not the behavior of applications managed by these processes.

Anomaly detection on user supplied parameters has been proposed. Using concepts drawn from system call argument analysis prototypes [2], in [3, 4] a web application IPS is proposed. Basically, the concept of system call is replaced with the URI of the requested resource, and the concept of system call parameter is replaced with the parameters passed to the web application. In other words, any web application is modeled as a set of URIs, each with an associated vector of *attributes*. For each URI a model of attributes is then generated. However, this makes the prototype unable to distinguish between different behaviors of the same path. It is common, for small-sized web applications, to use a single path to perform completely different tasks depending on a parameter. This was a key critique we brought against [2, 5], and solved through the use of clustering in [6]. The resulting prototype, named S^2A^2DE, incorporates sequence analysis and inter-argument relationships.

In [7] a Markov model was used to model the character sequences in the request. However, the reported false positive rate is by far too high for any practical use.

More specialized works have targeted separately XSS and SQL Injection vulnerabilities. To address XSS, in [8] a client-side proxy was used to detect harmful tags supplied by the user and sent back by the server, using a concept similar to the one used in web vulnerability scanners [9]. This technology however works only on *reflected* XSS attacks, and not on *persistent* attacks where

the data is stored somewhere and shows up in the application at a later time. In [10] 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 [11, 12]. However, XSS attacks in the new Web 2.0 context can perform very different tasks: an example are the many worms affecting 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 [13] with the implementation of a browser-level application firewall able to intercept HTTP requests. All these techniques however are client-side protections, and they are not really anomaly detection techniques as much as they are generalized misuse based system with broad rules to block specific attacks..

In [14] 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 greatly interesting, but it requires the rewriting of large portions of code, since every line which contains SQL statements and queries needs to be rewritten, making the effort similar to a full code review for implementing proper filtering in the application. In [15] the authors propose a learning based approach to the problem of detection of SQL Injections. A server-side component is embedded in the web server, and analyzes SQL queries with techniques similar to [3, 5], 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 instead in [16] with promising results.

3 Masibty: a framework for web application intrusion prevention

Masibty is a highly modular framework for anomaly based intrusion prevention on web applications. It was designed to be deployable with a limited impact on any existing infrastructure. We specifically designed Masibty not to require an attack-free dataset for training, as this is a requirement not compatible with a real world deployment.

3.1 The concept of Entry Point

In order to systematically develop an IPS for web applications, we conceptually modeled interactions between user and applications within the bounds of the

HTTP protocol as composed of [17]:

- The URI, the identifier of the requested resource
- The parameters supplied, as different parameters and values can yield completely different results, when supplied to the same URI
- The session context: a session is 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)
- Finally, there can be also other influences: remote web services might change their answers, another user might have interacted with the database in such a way to impact the current user, and so on

We slowly identified the concept of an Entry Point (EP), as opposed to the URI, as the basic entity of an application. An EP is basically an augmented URI which is further specialized depending on parameters, session context and other influences. The relationship between an EP and a URI is not one-to-one, as in many applications the same scripts (or classes) perform different tasks, according to the value of some parameters, of previous queries, sessions or other factors. As a simple example, consider the case of an application with a generic `controller` script which dispatches user interactions to the various components of the application depending on a `command` attribute in queries (which, while not being a good software engineering practice, is a quite common situation). This is not a single EP, even if it corresponds to a single URI: a change in the `command` parameter may lead to dramatic changes in the application behavior.

Masibty does not apply models to URIs (as done in all of the previous works), but rather to EPs. The creation of EPs is therefore critical. We have developed two EP creators for the current prototype. A first one, named *Analyzer*, is just a simple parser which directly associates an EP to a URI. This is useful if the application can be mapped by hand, or if it is known a priori that the association will be one-to-one, to avoid the more complex process used by the "real" automatic EP generator, named *Clusterer*. The Clusterer groups similar requests together, to automatically generate a set of EPs.

We need a clustering algorithm with some specific requirements: it must be incremental, unsupervised, and able to deal with categorical values. The only viable solution is to use an agglomerative, incremental online clustering algorithm [18]. The only disadvantage is that this type of algorithm potentially depends on the ordering of the requests in the training set. We use as distance metric the sum of the distances along the following metrics: number of parameters in the request, presence of the parameters, length of each parameter. The algorithm prunes out clusters with limited support (e.g. containing only a limited number of instances) to cut out any outlier (possibly, an attack) in the training set.

Currently, the use of the *Analyzer* needs to be hand-configured by the user: a future extension of this work could be to automatically set static associations during learning for EPs that likely correspond one-to-one with URIs.

3.2 A Modular Infrastructure

Masibty has been devised as a modular and extensible infrastructure, easily portable to different languages and environments. In particular, we developed a *reverse proxy* module, which is a standalone application (currently developed in Java), and we defined an interface for an *application library* to monitor SQL calls (of which a proof of concept implementation was also produced).

The reverse proxy is the core of the application. This is a common approach in commercial web application firewalls, as well as in previous research prototypes: however, an unique feature of Masibty is that the proxy is designed to detect and block attacks targeted at both servers and clients. Although some client-side exploits can be identified just by examining inbound queries, in many cases outbound content analysis is needed: this is not commonly done by reverse proxies. We already discussed the growing importance of this feature in the context of Web 2.0 applications. The proxy was developed by extending a Jetty HTTP server (`http://www.mortbay.org`).

However, as the proxy only interacts with the application in a black box fashion, a Masibty deployment can include an application library module (SQLModule) to allow deeper analysis of SQL queries generated by the application. We developed a proof of concept implementation of this library for PHP and MySQL, but it can be easily reimplemented for any other language, or extended to support other databases: this is the only component of Masibty which is not language-agnostic. We evaluated the possibility of using a reverse SQL proxy to avoid implementing a language-dependent component, but it would only allow to analyze queries without tracing them back to a specific EP or user interaction. This is prone to false negatives whenever an aggressor is able to produce a query which would be legal in a different context, regardless of the anomaly detection model in use. SQLModule is a wrapper for the PHP embedded MySQL libraries: the MySQLi class and the *mysql_* functions. To use SQLModule, the administrator is required to alter slightly the application to be protected, modifying calls to *mysql_* functions into *masibty_mysql_* invocations, and mysqli objects into masibty_mysqli objects. Since the interface of every method has been respected, this changes can be applied through a simple batch job. Another possible implementation 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.

3.3 Software Architecture

The modular architecture of Masibty allows for a plugin-based extensibility and flexibility. The architecture is describe in Figure 1. The core component of Masibty is called *Anomaly Brain*. It routes requests coming from a module (*proxy* or *SQL* library, see above in Section 3.2) to any number of *Anomaly Reasoners*. Requests coming from the reverse proxy module are first passed through the

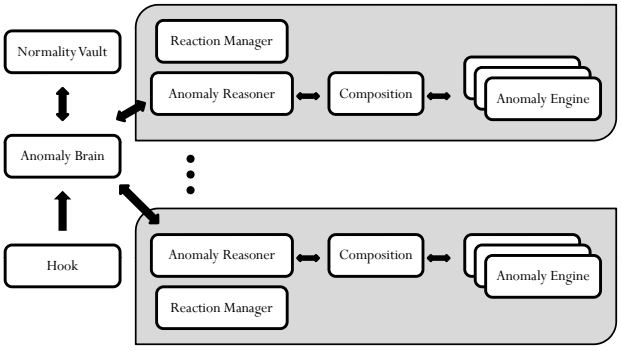


Figure 1: Overall architecture of Masibty

Entry Point Creator which clusters them (during learning) or classifies them (at detection time), according to the specific EP they access. This is done by inspecting both the parameters and the requested resource, as explained above in Section 3.1.

An *Anomaly Reasoner* (AR, in the following) handles a single facet of the complex issue of anomaly detection. For instance, an AR can focus on "anomalies in the parameters", whereas another may focus on "Client-side attack protection". However, each AR has full access to any information available to all of the Masibty modules (e.g., an AR working on SQL queries has access to the full session history). Some ARs are configured to be executed before the event is forwarded for processing, while others are configured to be executed on the results. 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), and the results are delivered to the post-query tasks through the Anomaly Brain. If cleared, they are finally delivered back to the requestor.

ARs make use of different *Anomaly Engines* (AE, in the following) to decide if a specific event is anomalous or not. AEs are atomic implementations of anomaly detection models, able to capture or model a single aspect of an event. They are atomic in the sense that they cannot communicate with each other. In detection mode, each AE generates an anomaly score $\in [0, 1]$ for each handled event: the result of each AE in the same AR is then combined to obtain a single anomaly score. Each AR can use a different policy to aggregate these values. The final anomaly score is then compared against an user-configured threshold to identify which events to flag as anomalies. AEs also provide a *trust level* (once again $\in [0, 1]$), which represents the reliability of the model on the

training data.

Although it may seem reasonable to combine the output of all the ERs to obtain a final overall anomaly value, it must be noted that each reasoner captures only a narrow subset of the information. Thus, an attack could be recognized by a single AR, and this could lead to the alert value being "diluted" and nullified by the other ARs. For this reason, we introduced instead the concept of *Reaction Managers* to handle actions to be performed after the detection of an anomaly. These action may range from stopping the request to mangling the returned data, or activating various type of alerts. Custom Reaction Managers can also be easily implemented to extend the system capabilities.

Each AR can have one or more Reaction Managers. We define a concept of priority to allow handling of multiple Reaction Managers that are activated concurrently. A Reaction Manager can temporarily suspend any other Reaction Manager 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. The choice of the reaction strategy is customizable by the developer of the reaction manager.

Every parameter learned by the AEs and used later in detection phase is archived in the *Normality Vault*. This is an abstract interface which allows to store and retrieve knowledge objects identified by an EP/AE combination. The vault allows for easy reimplementation, hiding the underlying storage mechanism.

4 Anomaly Detection Reasoners and Engines

We have currently implemented three anomaly reasoners: two (PAnomaly and XSSAnomaly) are built in the Proxy module, while a third one, QueryAnomaly, is built in the SQL module. PAnomaly and QueryAnomaly are pre-forwarding AR, whereas XSSAnomaly is executed on the answers.

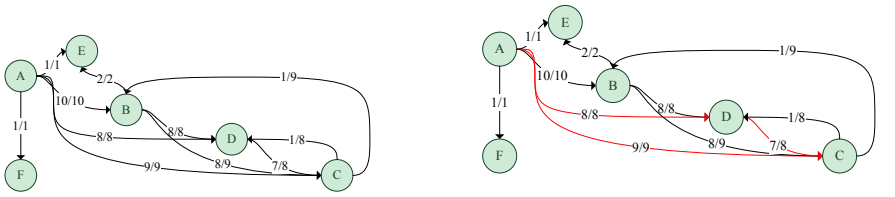
4.1 PAnomaly

PAnomaly stands for Parameter Anomaly: engines of this reasoner are aimed at detecting anomalies in parameters included in user-submitted queries.

4.1.1 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. This AE builds a probabilistic model based on the usual order, using a directed graph which represents the order in which the parameters have been seen, with edges labeled with two values: the number of times the origin precedes the other node, and the number of times both nodes appeared in examined queries. An example can be seen in Figure 2(a).

To detect anomalies, we identify the *active* edges for every incoming request, the ones identified by the 2-grams of the parameters in the query. For instance,



(a) Model at the end of the learning phase

(b) Active edges on a request containing attributes ACD

Figure 2: The probabilistic tree generated by the Order Engine

using the model in Figure 2(a), a request containing attributes ACD would result in the activation of the red edges in figure 2(b). The anomaly score is then calculated 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 equal to $1 - min(\frac{\#SeqApp}{\#TotalApp})$. The detection algorithm has complexity $O(\frac{N*(N-1)}{2})$ since every edge of the induced subgraph has to be generated and evaluated, the worst case being a request with N elements.

The Trust Level is computed to take into account how many infrequent couples are present, because if each couple has been seen a very small number of times the whole structure is too variable to provide reliable information, and is thus expressed as $avg(\frac{\#TotalApp}{\#TotalRequests})$

4.1.2 Presence Engine

With the same assumption of regularity of queries stated above, web applications usually have a small set of parameters associated with any given EP, and it's unlikely that they will change if not tampered with. This AE detects anomalies by checking for expected or unknown parameters in queries. During the training phase the presence of each parameter is recorded, along with its appearance ratio in the query.

Anomaly detection leverages the relative distribution of such ratios. Anomaly score is obtained by the equation $1 - min(max(\frac{Missing}{TotalCalls}, min(\frac{Present}{TotalCalls})))$. As can be seen, the presence of an unknown attribute or of a very rare attribute turns into a very high anomaly score.

The trust level should be high if the parameters are constant, and it should be lowered if an application constantly produces variations, for instance if the application generates some security-related parameters at runtime. We calculate it in function of the *misses* the engine encountered, with the equation $TrustLevel = 1 - \frac{Misses}{TotalCalls}$.

4.1.3 Numbers Engine

Numerical parameters are extremely common: identifying parameters which can only contain numbers can stop most injection-based attacks against them. For each parameter we store two values: the number of times the attribute value was not a number, and the number of observations of the attribute. If the ratio of these values is close to zero, the value is likely numerical. Attacks or application errors might obviously have polluted the training set, so an exact zero is rare.

Here, and a number of times in the following, we use a Yule-Simon distribution to associate an high anomaly score to values which are "seldom" seen. We compute the YS probability density function against the ratio between the number of times the attribute had "rare" values and its total observations. The result will be close to 1 if the attribute was "almost always" as expected, and will drop quickly for variable or unstable inputs.

For the numbers engine, we use the Yule-Simon distribution on the ratio between the number of times the attribute had not numerical values and its total observations.

4.1.4 Token Engine

This AE checks the parameters looking for tokens, i.e. items with enumerable possible values. Once found, it stores the values and marks as anomalous any request with an out-of-the-enumeration value for a token attribute.

Token identification is performed using the algorithm described in [4]: a function is generated representing the values of the candidate attribute, and is compared to a reference function representing a random-generated attribute. If no correlation is found, the parameter is likely *not* random, and thus it is identified as a token. 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). Also, we count the relative occurrences of each different value (something not done in the original algorithm).

Anomaly detection is straightforward: we wish to return an high anomaly value for values that have never, or seldom, been seen during learning. The latter is needed to weed out any occurrence of attacks in the training set. We resort again to the use of the Yule-Simon distribution, computing it for each observed value $v \in V$ against the ratio $N_v * |V|/N$, where N_v is the number of observations of v, and $N = \sum N_v$ is the total number of observations.

The Trust level is set to the heuristical value $min(1, |max(-1, p)|)$. 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.

4.1.5 Distribution Engine

The distribution of characters in a parameter is significant: a text-only input field has a different distribution than a parameter containing a binary repre-

sentation of an image. The Distribution Engine captures such variations by building a model of characters distribution through a representation of the relative frequencies of occurrence. For this we adapted to online use the algorithm proposed in [4] to perform a variant of the Pearson χ^2 test to determine if an observed attribute value can actually be generated by the learned distribution. From the output of the test $p \in [0, 1]$ the anomaly score is simply calculated as $1 - p$. The algorithm itself only requires a single scan of the input and a time-constant calculation, its complexity being thus $O(n + k)$.

4.1.6 Length Engine

Most of the parameters of a web application are not random in size: some have fixed size (e.g. tokens, identifiers,...); some have a certain degree of variance; only a few are completely random in length. Long attributes are commonly associated with overflows, and also XSS attacks can be quite long: for instance, the shortest known XSS worm is 161 byte long [19]. This model tries to approximate the (unknown) lengths distribution for a given parameter. Once again, we trivially adapted the algorithm described in [4] to work on-line.

Learning is straightforward. Since no information is available about the actual distribution of any attribute lengths, we assume an arbitrary distribution with sample mean μ and sample variance σ^2, and proceed to compute them from training data. Detection is performed through the Chebyshev inequality, which determines, for an arbitrary distribution, 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, then the Chebyshev inequality states that $p(|x - \mu|\, t) < \frac{\sigma^2}{t^2}$. Therefore, the probability of a string of size l is $p(|x - \mu|\, |l - \mu|) < p(l) = \frac{\sigma^2}{(l-\mu)^2}$. This model is fast and performs well, even though it is not able to capture attributes shorter than usual, since for $l < \mu$, $p(l) = 1$.

4.2 XSSAnomaly

The XSSAnomaly reasoner is aimed at detecting client side attacks. It detect anomalies either in the embedded scripting code, or in the DOM (Document Object Model [20]) of the response. This is because we try to address also new threats coming from DOM attacks (e.g. client-side website defacements).

The reasoner extracts the DOM tree from the returned documents using the Gecko engine, a fast, open source parser and layout engine used by several browsers (most notably Mozilla Firefox). It is implemented in C++, and it is accessible through the use of the XPCOM APIs; Masibty uses the MozillaParser Java library to wrap those APIs.

We decorate the DOM tree with information related to the Javascript content of each node, and we also prune out textual or otherwise non-Javascript attributes, keeping only *structural* information. We call the resulting structure *Anomaly Tree*. Two nodes on an Anomaly Tree are different if they have different names, different associated javascript code or non-matching subtrees. Figure

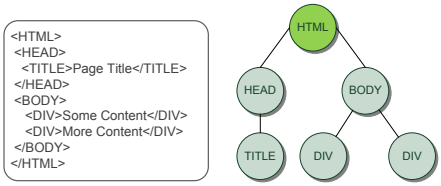


Figure 3: A web page and its AnomalyTree

3 shows an Anomaly Tree for a simple web page. Each node contains javascript related information, which are not shown in the graph. Note that changes in the content of the `DIV` node, unless they have some DOM consistency, would not show up in the Anomaly Tree.

4.2.1 Crisp Engine

The Crisp Engine is aimed at the detection of anomalies both on the DOM and in Javascripts. Anomaly Trees are used by the AE to learn the normal structure of pages associated with a given EP. The assumption is that the answers to a single EP will be very similar to each other (e.g. a template filled in with variable information).

During learning, the first Anomaly Tree is simply recorded. As soon as another tree is available, it is compared with all the known trees: if a perfect match is found, the counter for the matching tree is incremented, otherwise the new template is stored. However, this is obviously insufficient as it does not take into account *repetitions*. Information is often shown in the form of repeated structures containing data (e.g., search results or items in a store). To find such structures, the trees are walked through in parallel: as soon as a mismatch is found, a *sweep* is performed on the largest subtree, checking if all of the following elements have the same structure. If one of the elements in the larger tree is different, and it is not equal to the next element of the smaller tree (thus marking the end of the repetitions), the trees are different and stored separately. Otherwise, if the sweep successfully completes, this difference is ignored as just a different set of repetitions, and the rest of the trees are compared. This is a fast, single pass algorithm, with linear complexity w.r.t. the largest input tree size.

As an XSS injection can be obtained by adding just one element to the DOM, the Crisp Engine deems as anomalous any Anomaly Tree that does not match any of the previously learned trees, or which matches a learned tree without much support. Once again, we use Yule-Simon against the number of occurrences of the observation in the training set vs. the total number of instances to generate an appropriate anomaly value.

It must be further noted that two nodes are deemed as equal if the actual javascript code contained in each node is also equal. This creates issues with

runtime generated Javascripts, but it allows the AE to be resilient to mimicry attacks.

To avoid the model generating too many false positives, the trust level of this model for a given EntryPoint is calculated as 1 minus the ratio between the number of different Anomaly Trees and the total number of queries processed by the Engine. If the ratio is low, and thus the number of total queries is far greater in comparison to the number of different AnomalyTrees, the AE can be trusted and thus it returns a value which is very close to 1.

4.2.2 Template Engine

The Crisp Engine does not work well if the application generates very dynamic pages, as is the case in HTML-enabled forums or blogs. This is because it privileges detection over generalization capabilities. We developed also a Template Engine which is able to generate sets of templates that can match pages with a very high degree of variability.

The templates are generated by swapping one or more nodes of a pruned Anomaly Tree with wildcard nodes. The maximum number w of wildcards to be inserted is a parameter: more wildcards mean higher performance on very complex pages, as well as higher computational complexity. The algorithm works as follows: it swaps one node a time (and its subtree) with a wildcard. Thus, if $w = 1$ wildcard is allowed, a number of templates equal to the number of nodes n is generated. With $w = 2$ this grows to $n \times (n-1)$. 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 by itself is rather expensive: with 2 wildcards, for each new tree the complete algorithm is $O(n^2 + K * n)$ 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 $K * n$ 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 wildcard) that match all, or almost all, the response pages.

Anomaly Detection is then performed by testing the Anomaly Tree of any generated result for compatibility against each of the available templates, with the wildcard nodes being able to validate any subtree starting at their positions. If the pruned tree validates every template, as is the case for an EP with stable content, a null anomaly score is returned. If not, an anomaly score is produced using the observation rate of the highest non-matching template. The trust level is 0 (and the AE is thus deactivated) if an EP has no templates (i.e. it has no javascript code). Otherwise, it is the frequency of the highest matching template.

4.2.3 JS Engine

This AE uses a very straightforward approach to identify unknown Javascript code inserted into web pages: it generates and stores a fingerprint for each

code snippet detected during the learning phase, and then looks for unknown fingerprints in new pages. As it is easy to see, this is effective only if the pages reuse the same subset of Javascript code over and over, and it is not effective if even one code snippet is generated at runtime.

Learning is straightforward, and the complexity of the algorithm is linear with the number of Javascript snippets in the new pages. For detection, once again we use Yule-Simon to assign high anomaly scores to matches with a very low rate of appearance.

Trust level for this AE is measured as the ratio between the number of fingerprints stored during learning over the total number of scripts. If a page contains runtime generated Javascript, this ratio drops until the engine is effectively deactivated.

4.3 QueryAnomaly

QueryAnomaly is aimed at the detection and prevention of SQL injections through the analysis of queries before they are submitted to the database. Being implemented on the application side, it has full access to the application data, e.g. which script was invoked by the user, which script generated the query, etc.

Currently, the only implemented AE is the *Structure Engine*, which is able to parse queries into a tree-based representation and detect anomalies on their structure. Contrarily to what was done in [14], we do not require to change the query, and just minimal changes to the code itself. Contrarily to what is proposed in [16], we perform this check by dynamically building the model, as opposed to the use of static analysis on the application source code.

We leverage a pruned version of the basic translation of SQL queries in trees, removing any constant or user-supplied data, and keeping only logical and arithmetic operators. This of course creates a possibility for mimicry (e.g. a query where only the names of the tables have been altered will not be detected as anomalous), but in most cases injection attacks must dramatically alter the structure of the query to be performed.

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 of 1. Otherwise, it returns a value proportional to the ratio between the number of times the Structure Tree has been observed and the average rate of appearance of any known StructureTree on the same EP.

5 Experimental results

Appropriately testing Intrusion Detection and Prevention Systems is known to be difficult, and no broadly accepted methodology exists today [21]. A proper experimental assesment of Masibty presents some additional difficulties related to its “online” nature. In fact, because it behaves as an additional layer between

the client an the web server, it cannot be tested offline on a dataset previously collected following an approach similar to [3, 4]. Using an experimental tool in a production environment was also not really an option. Thus, we used a case based experimental approach, and we built a synthetic testbed as follows.

We monitored the BugTraq mailing list looking for exploits in web applications, and downloaded and deployed the vulnerable versions. Four real web applications were selected to be used as a test bench for Masibty: Artmedic Weblog, SineCMS, PHP-Nuke and JAF. All are PHP-based. An additional application is BadStore, a Perl application developed purely for testing purposes.

Before starting our tests, we have populated the underlying databases with fake yet reasonable data resembling as close as possible real world data. We have then used the applications, trying to emulate the interaction users and administrators would normally have, and thus exploring as many EPs as possible. We have logged the queries, creating an attack-free dataset that we were able to replay as needed using custom scripting tools. We have then created and tested the exploits for the vulnerabilities, also producing mutated versions via manual modification or IDS evasion techniques. Attacks included XSS attempts, remote file inclusions and SQL injections. From these datasets we generated a training dataset, by inserting a small number of attacks and "attack attempts" (non-working modifications of the attacks), to constitute about 1% of the training set. We also originated a similar "validation" dataset. We trained the application over the first, and tested it over the second, with the following results.

On simple applications (such as Artmedic Weblog and SineCMS) all the inserted attacks were identified, with no false positives. Suspecting overfitting, we inspected such results manually, and we also tried to develop further mutated versions of the attacks, but we were not able to evade Masibty. In the case of PHP-Nuke, once again we had no false positives, but we missed one of the XSS attack likely due to a bug in Gecko and the wrapper libraries. In the case of JAF, since it uses a flat-file storage, the SQL module was not useful; however, the Proxy module successfully identified all 16 attacks we targeted it with. JAF has the unique ability, between the applications we tested, to include external HTML pages created by the administrator. We included some rather complex pages with javascripts, and used a different set of pages in the training and verification dataset. This triggered a small rate of False Positives (around 0.58%) on the EPs involved.

Globally, Masibty obtained a 93% Detection Rate, and a False Positive Rate of 0.16%. This value is not meaningful per se, but it can be perceived as small if we consider that only one of the EPs we have tested actually produced false positives. For comparison with systems that were tested less thoroughly, we also ran an additional test by excluding attacks in the training set, and by using only vanilla attacks without evasion techniques. Under such load, Masibty showed a 100% DR and a 0% FPR. But as we said, such conditions are not realistic.

The performance in terms of both throughput and introduced delay of an IPS in a production environment is obviously very significant. For this reason we performed some experimental evaluations, in order to have a first-order approximation of the impact Masibty would have on an infrastructure. This is

of course limited by the fact that Masibty is an experimental prototype, not optimized for a production environment. So the results should be evaluated qualitatively, and not quantitatively.

We used a common workstation with 4 GBytes of RAM and an Intel Core2 Quad Q9300 at 2.50 GHz. We installed a Linux Ubuntu 8.10 distribution, over which we ran the Apache webserver as well as Masibty (in a real deployment, Masibty could of course be ran on a dedicated server).

We reproduced a closed queueing system with an increasing number of customers, using HP LoadRunner software. We ran tests for 10 minutes each, with the following load profile: in the first 3 minutes the number of customers increases gradually from 0 to 25. Then, their number remains constant for the following five minutes, and in the last 2 minutes it decreases to zero. We recorded a 10-step navigation sample within the web application so that each virtual customer behaves as a human user. The session generates 25 requests. The virtual users perform the sequence by waiting an amount of time (the so-called think time) between each step. The total think time in each cycle for each virtual user is $15s$. Under the specified load profile, the response time of the system, both with and without masibty, is almost constant, and it averages to $0.05s$ per request without Masibty, and to $0.21s$ per request with Masibty activated. This shows qualitatively that while Masibty in its current implementation imposes significant overhead, but it also shows it is not unfeasible to optimize it to be usable in a production environment.

6 Conclusions and future works

In this work we described Masibty, a prototype web application firewall designed to improve over the anomaly detection approaches for web applications that are available in the literature. It can work under realistic assumptions, eliminating the need for attack-free training data, and can deal with applications with a complex structure thanks to the concept of EP. We have 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. We also implemented a PHP library which contains a set of models for detecting anomalies in SQL queries through structural analysis. Our techniques of server-side analysis of web pages and our tree driven detection of anomalies in SQL queries are both innovative contributions, deriving from a new approach to the problem of detection of XSS and SQL injections. We have also improved previous works on anomaly detection on web application parameters and proposed new models for its computation. We have performed preliminary testing of our solution on four "real world" applications and a test application, obtaining good performance results and confirming the effectiveness of our approach. The overhead introduced is not unfeasibly high, and can be reduced easily through software optimization.

Future extensions of this work may deal with the introduction of some de-

grees of supervised learning, allowing the administrator to give feedback to the system. Furthermore, an automatic mechanism for choosing between one-to-one and many-to-many association between URIs and EPs (i.e. choosing between the Analyzer or the Clusterer engine). Another extension of the entry point concept could take into account URI rewriting techniques, mapping multiple different URIs to a single EP. Other extensions we are currently working on are a reasoner able to perform anomaly detection on headers and cookies, and a session-tracking mechanism which would allow us to take into account the sequence of pages and queries performed by a single user.

References

[1] Michael Sutton, Jeremiah Grossman, Sergey Gordeychik, and Mandeep Khera. Web application security consortium statistics. Available online at `http://www.webappsec.org/projects/statistics/`.

[2] 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.

[3] 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)*, pages 251–261, Washington, DC, October 2003. ACM Press.

[4] C. Kruegel, G. Vigna, and W. Robertson. A Multi-model Approach to the Detection of Web-based Attacks. *Computer Networks*, 48(5):717–738, August 2005.

[5] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous system call detection. In *ACM Transactions on Information and System Security*, volume 9, pages 61–93, 2006.

[6] Federico Maggi, Matteo Matteucci, and Stefano Zanero. Detecting intrusions through system call sequence and argument analysis. Submitted for publication, 2006.

[7] Juan M. Estévez-Tapiador, Pedro García-Teodoro, and Jesús E. Díaz-Verdejo. Measuring normality in http traffic for anomaly-based intrusion detection. *Comput. Networks*, 45(2):175–193, 2004.

[8] Omar Ismail, Masashi Etoh, Youki Kadobayashi, and Suguru Yamaguchi. A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability. In *International Conference on Advanced Information Networking and Applications*.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] Gregory Buehrer, Bruce W. Weide, and Paolo 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*, pages 106–113, New York, NY, USA, 2005. ACM.

[15] 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)*, pages 123–140, Vienna, Austria, July 2005.

[16] William G. J. Halfond and Alessandro 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*, pages 174–183, New York, NY, USA, 2005. ACM.

[17] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, L. Masinter, and P. Leach. RFC 28: Time standards, June 1999. Obsoletes RFC2068.

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

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

[20] Philippe Le Hégaret, Arnaud Le Hors, Steve Byrne, Lauren Wood, Mike Champion, Jonathan Robie, and Gavin Nicol. Document object model (DOM) level 3 core specification. W3C recommendation, W3C, April 2004. Available online at `http://www.w3.org/TR/DOM-Level-3-Core/`.

[21] Stefano Zanero. Flaws and frauds in the evaluation of IDS/IPS technologies. In *Proc. of FIRST 2007 - Forum of Incident Response and Securi ty Teams*, Sevilla, Spain, June 2007.