Reducing Errors in the Anomaly-based Detection of Web-Based Attacks through the Combined Analysis of Web Requests and SQL Queries

Giovanni Vigna, Fredrik Valeur, Davide Balzarotti, William Robertson
University of California
Santa Barbara
{vigna,fredrik,balzarot,wkr}@cs.ucsb.edu
Christopher Kruegel, Engin Kirda
Technical University
Vienna
{chris,ek}@seclab.tuwien.ac.at

Abstract

Web-based applications have become a popular means of exposing functionality to large numbers of users by leveraging the services provided by web servers and databases. The wide proliferation of custom-developed web-based applications suggests that anomaly detection could be a suitable approach for providing early warning and real-time blocking of application-level exploits. Therefore, a number of research prototypes and commercial products that learn the normal usage patterns of web applications have been developed. Anomaly detection techniques, however, are prone to both false positives and false negatives. As a result, if anomalous web requests are simply blocked, it is likely that some legitimate requests would be denied, resulting in decreased availability. On the other hand, if malicious requests are allowed to access a web application's data stored in a back-end database, security-critical information could be leaked to an attacker.

To ameliorate this situation, we propose a system composed of a web-based anomaly detection system, a reverse HTTP proxy, and a database anomaly detection system. Serially composing a web-based anomaly detector and a SQL query anomaly detector increases the detection rate of our system. To address a potential increase in the false positive rate, we leverage an anomaly-driven reverse HTTP proxy to serve anomalous-but-benign requests that do not require access to sensitive information.

We developed a prototype of our approach and evaluated its applicability with respect to several existing web-based applications, showing that our approach is both feasible and effective in reducing both false positives and false negatives.

Keywords: Anomaly Detection, Web Security, Database Security, Data Compartmentalization.

1 Introduction

Web-based applications provide services to users by leveraging the underlying web infrastructure. In many cases, web-based applications include client-side components, such as JavaScript code, that interact with server-side components, such as PHP pages. The server-side components, in turn, often access the application's data stored in a back-end database. Unfortunately, while the infrastructure components —such as web servers, language interpreters, and database engines— are usually developed by experienced programmers with considerable security awareness, the application-specific code is often developed under strict time constraints by programmers with little security training. As a result, vulnerable web-based applications are often exposed to the entire Internet, creating easily-exploitable entry points for the compromise of entire networks.

The wide availability of vulnerable web-based applications has attracted the attention of malicious hackers, who see in web-based applications relatively easy vectors for exploitation and access to sensitive information, which might lead to a monetary gain. The same trend can be seen by analyzing the number of web-related vulnerabilities published in the Common Vulnerabilities and Exposures (CVE) database [9]. The results, depicted in Table 1, show that the percentage of web-related vulnerabilities increased from 15.1% in 1999 to 49.1% in 2005, and this trend is projected to increase during 2006. Furthermore, past web-based compromises, such as the incidents involving Tower Records [21] and Victoria's Secret [29], have incurred considerable costs in terms of settlements paid to users whose private information was disclosed as a result of these attacks.

Ideally, the security of web-based applications should be addressed by means of careful design and thorough security testing. Unfortunately, in the real world this is often not the case. For this reason, security-conscious development methodologies should be complemented by an intrusion detection infrastructure that is able to provide early warning of malicious activity.

Web-based attacks can be detected, and possibly blocked, by intrusion detection systems (IDSs) that use signature-based techniques. For example, Snort 2.3 [22] devotes 1006 of its 2564 signatures to detecting web-related attacks. However, many web-based attacks cannot be easily modeled by signatures because they are specific to particular applications, which are developed in-house to serve the needs of a single organization. In addition, many web-based attacks do not contain any common characterizing feature, such

as the “NOP sledge” in a buffer overflow attack.

To detect attacks that are tailored to specific web applications, anomaly detection systems have been proposed that characterize the “normal” use of a web-based application. These systems use machine-learning techniques to determine the normal usage profiles associated with the applications. These profiles are then used as a basis to determine if a request is “anomalous” or not [11]. This approach has shown considerable promise, and a number of research prototypes and commercial products based on this idea have been developed [7, 8, 18].

Unfortunately, anomaly detection systems are notoriously prone to both false positives and false negatives due to reasons such as over-simplified modeling techniques or insufficient training. This means that an anomaly detection system may flag a benign request as anomalous or, conversely, an attack as normal. If anomalous web requests are blocked, false positives can lead to legitimate requests being denied, resulting in a degradation of the service. In a complementary fashion, false negatives represent attacks that are allowed to reach the protected web application, potentially resulting in the exposure of sensitive information.

To ameliorate this situation, we propose an architecture that serially composes a web-based anomaly detection system, a reverse HTTP proxy, and a database anomaly detection system. The resulting system increases the probability of detection while simultaneously reducing the negative impact of false positives.

To augment the detection capability of a web-based anomaly detector, we propose the addition of an anomaly detection system that operates on the queries that a web application executes against a back-end database. The analysis of these queries allows the system to detect the effects of malicious web requests that were incorrectly deemed normal and let through by the web-based anomaly detection system. When an anomalous database query is detected in association with a normal web request, a characterization of the anomaly is sent over a feedback channel to the web-based anomaly detection system, so that its models can be updated accordingly. As a result, similar malicious web requests are classified as anomalous and prevented from reaching the application.

We observe, however, that the serial composition of anomaly detectors, while potentially increasing the detection capabilities of the system as a whole, may also increase the probability of false positives. To mitigate this effect, we proposed a novel solution based on data compartmentalization and anomaly-based reverse proxying [28]. The idea is to replicate a web site on two or more “sibling” web servers with different levels of privilege (i.e., different levels of access to sensitive information). Anomaly scores from the web-

based anomaly detection system are then used to drive a reverse HTTP proxy, which is an application deployed in front of the sibling web servers. The reverse proxy intercepts HTTP requests destined for the web servers, and based upon an individual request's anomaly score, forwards requests to a sibling web server with the appropriate level of privilege. For example, a benign web request that is erroneously evaluated to be "anomalous" could be sent to a server with limited access to the back-end database, instead of being simply dropped. If the request does not need to access sensitive information, the request will be served correctly. In this way, the system is able to maintain a reduced level of service in the presence of false positives.

This paper presents our approach to mitigate the limitations of anomaly detection systems, describes a prototype implementation of our architecture, and provides a detailed discussion of its applicability to real-world applications. First, in Section 2, we motivate the problem and provide a high-level overview of the architecture of our system. In Section 3, we discuss different high-level design options for the processing of web requests at different levels of privilege, and then we describe the implementation of our reverse HTTP proxy. In Section 4, we present our approach to detecting anomalous database queries. Section 5 discusses the process of web request model updating over a feedback channel. Section 6 presents a discussion of the applicability of our approach with respect to several existing web-based applications. Finally, Section 7 discusses related work, and Section 8 briefly concludes.

2 Motivation

Figure 1(a) shows a typical web server setup. The web server receives HTTP requests from clients and serves these requests by invoking server-side scripts. These scripts utilize a database to store application-specific data. In order to improve security, a web-based intrusion prevention system (IPS) might be deployed as shown in Figure 1(b). In this architecture, the web server does not receive requests directly. Instead, the requests are first examined by the IPS, which chooses to either drop or forward them to the real web server, based on the characteristics of the requests.

Although web-based IPSs are useful in improving the security of a site, they are not foolproof and will inevitably let some malicious requests through. To mitigate this, we propose combining the IPS with an SQL-based anomaly detector as shown in Figure 1(c). In this setup, the IPS blocks requests that are found to be anomalous, while normal-looking malicious requests that generate (anomalous) database queries are

detected by the SQL anomaly detector. The SQL anomaly detector feeds information about the detected attacks back to the IPS, so that the IPS can update its configuration and improve its detection capability.

We now give a formal analysis of the potential for the composition of anomaly detectors to produce a reduction in the overall false negative rate. Adopting the notation from Axelsson [5], we define:

- I to be intrusive behavior, and $\neg I$ to be non-intrusive behavior;
- A_w, A_d to be alarms generated by each anomaly detector, and $\neg A_w, \neg A_d$ to be the absence of alarms;
- $P(A_w|I), P(A_d|I)$ to be the detection rates of each anomaly detector;
- $P(A_w|\neg I), P(A_d|\neg I)$ to be the false positive rates;
- $P(\neg A_w|I) = 1 - P(A_w|I), P(\neg A_d|I) = 1 - P(A_d|I)$ to be false negative rates;
- $P(\neg A_w|\neg I) = 1 - P(A_w|\neg I), P(\neg A_d|\neg I) = 1 - P(A_d|\neg I)$ to be true negative rates.

Given the serial composition of the web and database anomaly detectors in our architecture, we can describe the probabilities for the resulting system as follows:

True negative	**True positive**
$P(\neg A_w|\neg I) \cdot P(\neg A_d|\neg I)$	$P(A_w|I) + P(\neg A_w|I) \cdot P(A_d|I)$

False negative	**False positive**
$P(\neg A_w|I) \cdot P(\neg A_d|I)$	$P(A_w|\neg I) + P(\neg A_w|\neg I) \cdot P(A_d|\neg I)$

From the formulas above, we can draw two conclusions. First, the serial composition of anomaly detectors with feedback has the potential for increasing the true positive rate of the system as a whole. Second, unfortunately, the serial composition of detectors also increases the potential for false positives.

In our system, in order to mitigate this drawback, we do not block anomalous requests immediately, but we attempt to serve them through a web server with restricted access to sensitive information. This solution can help reducing the impact of false positives when the requests that have been erroneously identified as malicious do not require any sensitive data to be processed. This approach is described in more details in the next section.

3 An Anomaly-driven Reverse Proxy

Web-based attacks are aimed at either obtaining control of the host running the web server application (e.g., through a buffer overflow) or disclosing sensitive information (e.g., through an SQL injection attack that dumps the content of a database table).

The first type of attack is caused by vulnerabilities in the web server software or in a server-side web-based application that allows one to compromise the security of the underlying host. The second type of attack is usually made possible by the fact that a single back-end database is used to store all the persistent information of a web-based application. Therefore, by exploiting a vulnerability in code that was intended to have access to a limited portion of the database contents, it is possible to extend one's access to the database and retrieve sensitive information. For example, Figure 2(a) shows an e-commerce web site implemented with one web server that relies on a single back-end database (DB) and that also accesses a credit card processing server (CC Server). All the application functions (i.e., `f1`, `f2`, and `f3`) have the same level of access to the database, even though they use different tables. For example, `f2` only uses table `x`, while `f1` does not use any table of the database.

A web site could be made more resilient to attacks if it would be possible to design both the server and the database infrastructure so that different levels of access to the database and the hosts running the server processes could be clearly enforced. For example, the e-commerce application could be structured so that: (i) non-sensitive, static information about the e-commerce company (e.g., company contacts and support information) is accessible through one server; (ii) the non-sensitive, dynamic information about product availability is accessible through a second server connected to a product database; and, finally, (iii) the sensitive information about users is accessible through a third server that relies on a user database, which is separated from the product database. This last server has also access to the credit card processing server. This is the design shown in Figure 2(b), where each function is implemented on a different server and accesses only the needed information.

In this design, the compromise of the server providing product information would not allow the attacker to access the database containing sensitive user information or the credit card transaction server. Unfortunately, this type of "compartmentalized" design cannot be easily applied to existing applications, where the different functions provided by a web site are closely intertwined.

Therefore, we propose an alternative design where the web site's contents are replicated across servers, instead of being partitioned. In addition, the database used by the web-based application is extended with user accounts at different levels of privilege. Each of these accounts is associated with one of the replicated servers. Then, an anomaly detection system is used to determine the likelihood that a request represents an attack. This information is used by a reverse HTTP proxy to forward the request to the web server that is able to provide the best possible level of service given the anomaly score of the request. A reverse HTTP proxy is a special proxy that is installed in front of web servers. Reverse proxies are usually used in order to improve the site performance by caching web pages or by distributing the load to several different servers. In our case, its role is to route the incoming requests to the proper web server, depending on the anomaly value that is set by the anomaly detector component.

According to this design, the e-commerce application described above would be implemented as shown in Figure 2(c). In this case, all the site's functionality (i.e., `f1`, `f2`, and `f3`) is replicated on three servers (`A`, `B`, and `C`). This step requires no modification of the original application. The web server proxy is configured so that queries that are highly anomalous are sent to server `A`, queries that are considered moderately anomalous are sent to server `B`, and normal queries are sent to server `C`. Server `C` is the only one that is able to access the credit card server. The database is modified to create two different users `u1` and `u2`, where `u1` is only allowed to access table `x` and `u2` is able to access both table `x` and table `y`. User `u1` is associated with server `B` and user `u2` is associated with server `C`.

In this design, if a request that uses function `f2` is sent to server `A`, the request will fail because, although `f2` uses the database, no connections are allowed to the database from server `A`. On the other hand, a highly anomalous request for function `f1` will be correctly executed. If identifying this request as highly anomalous represents a false positive, then the user will not be denied access. If the request actually represents an attack, then two cases are possible. In the first case, the attack is intended to gain access to the database, and, in this case, the attack will be foiled since no database connection are available. In the second case, the attack is intended to compromise the host. In this case, assuming that the attack is successful, the compromised host has no access to the credit card server and the damage is therefore contained. Also note that this host is likely to serve a small portion of the requests (only the ones that are flagged as anomalous) and thus, it could be "hardened" to be more resilient to certain types of attacks at the cost of some performance degradation (as is done, for example, by Sidiroglou et al. [23]).

Moderately anomalous requests are sent to server B in this design. If a request of this type uses function f2, it is executed correctly, since server B has access to the database tables utilized by f2. Note that the request would have failed if it had been sent to the more restrictive server A. However, a request that requires to access the database through the function f3 would fail, since this server do not have the permissions required to access the sensitive information. Finally, all normal requests are forwarded to server C. Since this machine provides full functionalities, all the requests can be served correctly.

The effectiveness of this architecture in reducing the impact of false positives depends on the amount of requests that do not require access to the functions f2 and f3. For example, imagine a web application in which roughly 40% of the requests do not require access to the database. If we assume that the requests erroneously flagged as malicious and forwarded to the A server are equally distributed, 40% of them can be properly served to the user. Therefore, in this simple example our approach would have successfully reduced the impact of false positives by 40%.

3.1 Sensitive Path Coverage

An estimation of the fraction of requests that can be served by a restricted server can be found by calculating the *sensitive path coverage* of the application. The concept of *path coverage* is a well-known metric in software testing that measures the fraction of execution paths through the program that are covered by test cases. The key idea is to adapt this metric to represent the fraction of all possible execution paths through the application that perform sensitive operations on the database. We call this fraction the sensitive path coverage. Note that the sensitive path coverage should not be mistaken for a precise estimate of the number of sensitive operations that can be expected to be performed. We assume here that all paths through an application occur with equal probability. Therefore, the result is more useful as a measure of how much code that accesses non-sensitive information is interspersed with code that performs sensitive database accesses. When sensitive operations are rare and cleanly separated from the rest of the application code (i.e., the sensitive path coverage is small), we expect our reverse proxy approach to be more successful in reducing the impact of false positives.

Unfortunately, the exact calculation of the path coverage is impractical. This is due to the problem of exponential path explosion that is caused by the fact that a code fragment with a succession of k decisions

(e.g., due to if-statements) contains up to 2^k different execution paths. Also, many of these paths may be infeasible (i.e., there is no input to the program that can cause a particular path to be executed). Therefore, one usually attempts to aggregate paths into clusters or to operate at a higher level than individual statements. Following this common strategy, we restrict the calculation of the possible paths *to the function level*. That is, we do not consider any intra-procedural control flows.

To calculate the sensitive path coverage for an application, we have to generate a call graph in which each node of the graph represents an individual function. An edge is introduced from node v to node w when the function corresponding to node v calls the function corresponding to w. Based on this graph, we first determine the number of possible paths through the application. Then, we determine the fraction of paths that perform sensitive operations. This fraction is our desired metrics, namely the sensitive path coverage.

Without performing intra-procedural analysis, we lose the information on the conditions that guard each function invocation. Therefore, if a function has k successors in the call graph, we need to assume that, at runtime, it can invoke any possible combination of them. Unfortunately, this again leads to an exponential explosion on the number of paths, as can be seen in Equation 1. $P(S)$ denotes the powerset of the successor nodes of N, and S_i represents the number of paths through the ith successor node.

$$p(N) = 1 + \sum_{s \in P(S)} \prod_{i \in s} p(S_i) \tag{1}$$

In order to reduce the influence of nodes with many successor node, we do not calculate the actual numbers of paths. Instead, we perform path aggregation using Equation 2. The equation determines the number of aggregated paths $p(N)$ through a node N in the call graph.

$$p(N) = 1 + k * \sum_{i=1}^{k} p(S_i) \tag{2}$$

In this equation, k is the total number of successor nodes of N. Here, we calculate only the sum of the number of paths through the successor nodes, adjusted by the factor k. This is in contrast to the actual path number, which is based on multiplying the path numbers of all possible combinations of successor nodes and summing the results. Intuitively, our approach to calculate the aggregated path number allows us to capture the fact that a node (function) with more successor nodes is more likely to be executed by the application. This is done by using the factor k. However, we prevent the problem that certain nodes with

many successors completely dominate the total number of aggregated paths (through the multiplications necessary for calculating precise path numbers). Of course, when a node has one or more successor nodes that have a high path number, this information is propagated upwards and leads to the expected result that the node itself also has a large number of paths.

The number $p(R)$ of the root node R of the call graph denotes the total number of possible paths (or aggregated path clusters, to be more precise) through the application. Note that the algorithm would not terminate when the call graph contains loops as a result of direct or indirect recursive function calls. Therefore, such loops are removed from the call graph prior to processing by using a simple depth-first search.

Consider the example in Figure 3, which demonstrates the calculation of the aggregated paths for all nodes of a small call graph. Given the recursive definition of $p(.)$, which calculates the number of paths at a node based on the number of paths through all its successor nodes, the algorithm evaluates the nodes of the call graph in bottom-up order. The value of a leaf node is always 1. The total number of aggregated paths through the application in this example is 31. As expected, the left child of the root node has a higher path count, because it has more successors.

Once the total number of (aggregated) paths has been calculated, we have to determine the number of aggregated paths that pass through nodes that correspond to functions that perform sensitive operations. We call such paths *sensitive paths*. Given both the number of sensitive paths and the total number of paths, we can easily calculate the sensitive path coverage.

To find all sensitive paths, we have to first identify sensitive operations. In our current analysis, sensitive operations are found by checking in the web application source code for string constants that represent database operations on privileged tables. Once all sensitive operations are found, each call graph node corresponding to a function that contains these operations is appropriately marked.

As mentioned previously, the exact types of operations and database tables that are considered sensitive depend on the application and the environment where it is deployed. We will discuss examples of different sensitive operations in more detail when describing the experiments with our three web applications in Section 6.1.

Hereinafter, we assume that all SQL operations can be seen as strings written into function bodies. This implies that the program neither utilizes global variables to store SQL statements, nor that they are

generated in a completely dynamic fashion during run-time. Of course, it is possible (and common) that the parameters for the `WHERE` clause of a `SELECT` statement or the values for an `INSERT` operation depend on variables. However, we assume that the type of the SQL operation and the tables that the program operates on are statically known. Fortunately, the direct inlining of SQL statements is a very common programming practice in web application scripting languages, such as PHP. Indeed, in all the programs we considered for our experiments, all database access statements were directly embedded as strings that specified both the type of operation and the accessed tables.

The technique to calculate the number of sensitive paths that run through the marked nodes is very similar to the previous technique for deriving the total number of paths. When a node is marked, all paths that run through it are automatically considered sensitive. If a node contains no sensitive operations itself, but sensitive paths run through (some of) its successor nodes, the sensitive paths $ps(N)$ of the current node N are derived using Equation 3.

$$ps(N) = k * \sum_{i=1}^{k} ps(S_i) \tag{3}$$

The previous call graph example has been extended in Figure 4 by marking a node as containing sensitive operations (the node that is drawn shaded in the figure). In addition to the numbers of the total paths at each node, which are given in parenthesis, the number of sensitive paths for each node is shown.

It is important to note that the sensitive path coverage is just a metric that we use to support the idea that part of the web requests can be successfully processed also by a server that do not have access to sensitive information. In other words, the path coverage is a way to validate our assumption and it is not part of the intrusion detection tool. For example, if the path coverage of a certain application is 75%, we can expect that using our reverse HTTP proxy technique the false positives rate would be reduced by roughly 25%. A path coverage of a 100%, often the result of a poorly designed application, means that all the HTTP requests require access to sensitive information. Even though in this case the proxy would not be able to reduce the impact of false positives, our composition of web and SQL anomaly detectors would still be effective in reducing the false negative rate.

As part of the system evaluation in Section 6, we compute the sensitive path coverage for the real-world applications that we tested in our experiments.

3.2 Routing Web Requests

We implemented the proxy-based approach by integrating an existing anomaly detection system into a reverse web proxy that we developed. The anomaly detection component of the proxy is based on the WebAnomaly system [11]. The anomaly detector first extracts from the requested URL the path to the web application being invoked, along with the arguments passed to it. The anomaly detector then looks up the *profile* associated with the web application. A profile is a collection of statistical models, which is associated with one specific web application.

The anomaly detection models contained in the profile are a set of procedures used to evaluate a certain feature of a query attribute, and operate in one of two modes, learning or detection. In the learning phase, models build a profile of the "normal" characteristics of a given feature of an attribute (e.g., the normal length of values for an attribute), setting a dynamic detection threshold for the attribute. During the detection phase, models return an anomaly score for each observed example of an attribute value. This is simply a probability in the interval $[0, 1]$ indicating how anomalous the observed value is in relation to the established profile for that attribute. Since there are generally multiple models associated with each attribute of a web application, a final anomaly score for an observed attribute value during the detection phase is calculated as the weighted sum of the individual model scores. The overall anomaly score is then used to decide which web server should process the request. For more information on the models themselves, as well as the anomaly detector as a whole and its evaluation on real-world data, please refer to [11].

Each of the back-end web servers has a different privilege level and processes requests with a different range of anomaly scores. For instance, the requests determined to be normal are sent to the web server with the highest privilege level, while anomalous requests are sent to a less privileged server. Depending on the privilege level of the server, the amount of sensitive information that can be accessed varies, since each web server connects to the back-end database as a different user with different privileges.

In order to prevent potential attackers from simply bypassing the proxy by connecting directly to the protected web server, efficient firewalling mechanisms must be deployed. We achieved this by assigning non-routable IP addresses to the web servers and deploying the servers on a network isolated from the Internet. A separate interface on the proxy server connects the proxy to the isolated network.

4 Detecting Anomalous SQL Queries

We have developed an intrusion detection system that utilizes multiple anomaly detection models to detect attacks against back-end SQL databases [27]. Hereinafter, we briefly describe the architecture of the system and then discuss how the results of the detection are used as feedback for the web-based anomaly detection component. Figure 5 shows an overview of the architecture of the database anomaly detector. The system taps into the communication channel between web-based applications and the back-end database server. SQL queries performed by the applications are intercepted and sent to the IDS for analysis.

4.1 Feature Extraction

The SQL anomaly detector parses each incoming SQL query in order to generate a high-level view of the event. The parser outputs this representation as a sequence of tokens. Each token has a flag that indicates whether the token is a literal or not. Literals are the only elements of an SQL query that should contain user-supplied input.

Tokens representing database field names are augmented by a *data type* attribute. The data type is found by looking up the field name and its corresponding table name in a mapping of the database. This mapping is automatically generated by querying the database for all its tables and fields. Then, the mapping can be updated by the user, if it is desirable to describe the data type of a field more accurately. For instance, a field in the database might be of type `varchar`, which implies arbitrary string values, but the user could change this type to `XML` in order to inform the IDS that the field contains an XML representation of an object. The set of available data types is user-extensible and the IDS offers a simple interface to specify how new data types should be processed by the system.

Type inference is also performed on the literals contained in the query using the following rules:

- A literal that is compared to a field using an SQL operator has its data type set to the data type of the field it is compared to.
- A literal that is inserted into a table has its data type set to the data type of the field it is inserted into.

Before detection is performed, the incoming events are processed in order to transform the queries into a form suitable for analysis by the models. First, a feature vector is created by extracting all tokens marked

as literals and inserting them into a list in the order in which they appear in the query. Then, a *skeleton query* is generated by replacing all occurrences of literals in the query with an empty place holder token. The skeleton query captures the structure of the SQL query. Since user input should only appear in literals, different user inputs should result in the same skeleton. An SQL injection, however, would change the structure of the query and produce a different skeleton query.

Consider, for example, the following query:

```
SELECT firstname FROM users WHERE userid="foo" AND passwd="bar";
```

The SQL anomaly detector parses the query and extract the sequence of tokens. In particular, `foo` and `bar` are literal tokens, while `firstname`, `userid`, and `passwd` are identified as field tokens. Analyzing the database schema, the three fields are automatically associated with their corresponding types (all `varchar` in our example). The types are then propagated to the literals, concluding that the user provided inputs `foo` and `bar` are both `varchar` since they are directly compared with the field `userid` and `passwd`.

Finally, the SQL anomaly creates the feature vector:

```
(varchar: "foo", varchar: "bar")
```

and generates the skeleton query by substituting the literal tokens in the original query with placeholders:

```
SELECT firstname FROM users WHERE userid=<INPUT> AND passwd=<INPUT>
```

4.2 Detection Engine

The operation of the detection engine differs depending on the status of the intrusion detection system, that is, if the system is in training or detection mode. In training mode, the name of the script generating the query and the skeleton query are used as keys to look up a *profile*. A profile is a collection of statistical models and a mapping that dictates which features are associated with which models. If a profile is found for the current script name/skeleton combination, then each element of the feature vector is fed to its corresponding models in order to update the models' "sense" of normality.

If no profile is found, a new profile is created and inserted into the profile database. A profile is created by instantiating a set of models for each element of the feature vector. The type of models instantiated

is dependent on the data type of the element. For instance, an element of type `varchar` is associated with models suitable for modeling strings, while an element of type `int` would be associated with models suited to modeling numerical elements. For user-defined types, the user can specify which models should be instantiated. See [15, 12] for a complete description of the different models.

In detection mode, an anomaly score for an observed value is calculated. If the anomaly score exceeds a threshold, an alarm is generated. Alarms are also generated if no profile is found for an event, or if an event contains SQL statements that cause a parse error. A parse error is an indication of an attempted SQL injection attack or a broken application.

5 Feedback Propagation

As described previously, the architecture of the composed anomaly detection system necessitates the existence of a communication channel between the SQL anomaly detection component and web request anomaly detection component. This channel is utilized when a malicious web request that was let through by the front-end anomaly detector (i.e., a false negative) results into a malicious database query, which is then detected by the SQL anomaly detector. In our system, the SQL anomaly detector can choose to update the models of the web request anomaly detector to block further, similar malicious requests. The updating process is performed in three phases: i) anomaly characterization, ii) construction and transmission of model update messages, and iii) model updating. These phases are described in the following sections.

Anomaly Characterization

The first step in the model updating process consists in characterizing the nature of the detected anomaly. In our system, anomaly characterization is a composition of several features of an anomaly, including the resource invoked by the anomalous web request, the anomalous value itself, and the model-specific parameters of the models that detected the anomaly, all of which are output by the SQL anomaly detector.

To illustrate the anomaly characterization process, suppose that the SQL anomaly detector applies a string length model based on the Chebyshev inequality [10] to the data field of a query. During the training phase, the detector has learned that this particular field has an observed mean length μ, a variance σ, and a threshold t. Now, during the detection phase, suppose that a certain query contains a field value x, whose

length is flagged as abnormal because the Chebyshev inequality returns $P(x) < t$. The system would characterize this anomaly by extracting the resource invoked as r. Then, μ, σ, and t are extracted from the alerting model, as well as a model type identifier m, forming the tuple (m, μ, σ, t). The final characterization of the anomaly is given by the tuple $A = (r, x, M)$, which is the composition of the resource r, the anomalous value x, and the set of parameters of all models that alerted on x, denoted by M. In this case, $M = ((m, \mu, \sigma, t))$, and the anomaly is characterized by the tuple $A = (r, x, ((m, \mu, \sigma, t)))$.

Model Update Messages

Given a tuple A that characterizes the nature of the anomaly detected by the SQL-based IDS, the next steps are the construction of a message that encapsulates this information and the forwarding of the message to the web-based anomaly detection system. In our system, the tuple A is translated into an XML-based representation and then sent to the web request anomaly detector through an encrypted socket connection.

Model Updating

Upon receipt of a model update message, the web request anomaly detector is faced with the task of selecting the correct model or set of models to update. Since multiple models are typically associated with each parameter of a resource, the specification of r as the resource that generated the anomalous SQL query is insufficient to determine the correct set of models. Thus, an algorithm to correlate the anomalous value to one or more resource parameter model sets is required. This process is complicated by the fact that, at least in theory, parameters learned by the web request anomaly detector may have arbitrary transformations applied to them by the request handler before the actual SQL query is constructed. In practice, however, such transformations are often very simple and many resource parameters are one-to-one mappings onto fields of a table in the database (e.g., usernames, email addresses, etc.).

According with these considerations, in our prototype implementation we adopt a model selection algorithm that works as follows. For each parameter model set for resource r, the models corresponding to the members of M are selected; this set is denoted by M'. Then, for each model $M'_i \in M'$, the anomaly score $P_{M'_i}(x)$ is computed. If the resulting probability of x denoted the fact that x is anomalous, the model is not updated. Otherwise, the model is updated to reflect the fact that $P_{M'_i}(x)$ must be anomalous.

For example, given the tuple A, the web request anomaly detector constructs the set M' of string length models attached to the parameters of resource r as described above. Then, for each model $M'_i \in M'$, its variance $\sigma_{M'_i}$ is reduced such that the x is no longer considered normal when the Chebyshev inequality is applied to strings of that length. The algorithm terminates once all models in M' have been updated to reflect the abnormality of x.

6 Evaluation

The purpose of this section is to provide some evidence about the general applicability of our technique, demonstrating how our approach is capable of reducing both the number of false negatives and the impact of false positives in real web applications.

For our experiments, we selected three web applications that satisfy three requirements: i) they implement substantial program logic, ii) they require a back-end database, and iii) they had to perform sensitive operations during normal usage. We chose `phPay 2.0` [20], `myBloggie 2.1.2` [16], and `punbb 1.2.5` [4]. `phPay` is a typical shopping cart application that supports products in different categories, search functionality, user management, and on-line payment. `myBloggie` is a blog script that allows users to maintain an on-line diary and to comment on other people's public entries. `punbb` is a discussion board that supports different forums, rich text formatting, as well as user management and subscription for message notifications. All three programs are written in PHP [19] and use the MySQL database [17] to store information. However, the three applications provide different set of functionalities, they use completely different database schemata, and they perform different kind of sensitive operations. Table 2 provides more information on the selected applications.

6.1 Analysis of the Sensitive Paths

To support our partitioning technique, we have to provide evidence that a considerable fraction of web requests can be handled by an application without relying on access to sensitive information stored in the database. Unfortunately, providing such evidence in a general fashion is difficult for the following reasons:

1. The fraction of requests that access sensitive information depends on the type of application. For

example, a news portal that uses a read-only database will probably never require access to sensitive tables since all information should be publicly available. An application that is used by a company to keep track of the working hours of its employees, on the other hand, will most likely require a significant amount of access to sensitive data.

2. Even when only a single application is analyzed, its site-specific usage might dramatically influence the number of sensitive operations performed. For example, in a shopping cart application, the ratio between users who anonymously browse through the catalogs and users who actually login and purchase goods will determine how many of the requests need full access.

3. Finally, it is not always clear which database tables and operations should be classified as sensitive. While most people would agree that modifications to a table with user data should be treated as a privileged operation, it is not always that obvious. In a discussion board application, for example, the sensitiveness of the list of subscriptions to topics of interest of a user might vary depending on the class of topics. For example, when the topics are related to sport events, then the sensitivity is likely to be low. When the discussions are on incurable diseases, however, this information might be highly sensitive.

In our experiments, we started by calculating the sensitive path coverage for the three web-based applications. This operation requires analyzing the source code to generate the call graph in order to identify the SQL statements that represent sensitive database operations. To this end, we have developed a static analysis tool that can process PHP code, implementing the complete PHP 4 standard. We used our tool to determine the sensitive path coverage for the three sample applications under different assumptions of what constitutes a sensitive database access.

It is important to note the it is not necessary to have familiarity with the application code in order to understand which information is sensitive. It depends on the schemata (and the corresponding data) of the database and it is usually very easy to identify which are the sensitive table in a specific application.

In the following paragraphs, the results are presented for each program.

`phPay`: We considered an access to the database as critical when it can modify the stored data. Under this assumption, the sensitive path coverage was 8.92%. If we also considered read operations from the `user`

table (that contains the user passwords), or read from the table that stores the pending orders and the cart data, the path coverage raised to 100%. Closer analysis revealed that a `SELECT` query to the user table was included in the program's main method to be able to display the user names of people currently logged in. However, a simple refactoring of the program to either remove this non-critical functionality or mirror the user name in the session information reduced the coverage to 78.84%.

`myBloggie`: Similar to the previous application, we checked the fraction of sensitive paths that perform (potential) write operations to the database. However, since the application is a blog, we did not consider as sensitive the table that contains the public comments to the blog entries. In this case, the sensitive path coverage was only 2.37%.

In the next step, we extended the set of sensitive operations with read access to the user table (which stores the user passwords in an encrypted field). Again, a path coverage of 100% was determined. This time, the reason was an SQL statement in the code that adds the user names to all log entries and comments that are displayed. If the application is modified to directly store the user names with the log entries and the comments, the sensitive path coverage decreases to 29.11%.

`punbb`: Analyzing the path coverage under the assumption that only write operations are sensitive, we obtained sensitive path coverage of 1.17% for the message and discussion board applications. When including the user table into the set of critical tables, the path coverage reaches 100%. For this application, the culprit is an SQL statement in the main function that generates the statistics of all people that are currently logged in. When this statement is removed, the sensitive path coverage is reduced back to 1.17%. Alternatively, the statistics could be generated from a dedicated table that stored only the names of the currently logged-in users, separately from the passwords.

For all three applications, our analysis indicates that the sensitive path coverage is quite small when classifying only database write operations as sensitive. In our tests it was always below 9% (and in two applications out of three it was below 3%). That means that using our reverse proxy approach we can expect over 90% of reduction in the impact of false positives.

With regards to read operations of sensitive tables (e.g., the user table with passwords), all programs initially showed 100% sensitive path coverage. However, by applying simple modifications to non-critical functionality of the applications, this value was significantly reduced. `phPay` showed the worst value, with

a coverage considering also the read operations that was close to 80%. However, this is not a terrible result, since also in this case we can expect that more than 20% of false positive requests could be successfully served by our system.

Finally, it is important to remember that these figures are only an approximate way to estimate the effectiveness of our solution. The actual results of the experiments we performed on the three applications are reported in the next section.

6.2 Experimental Results

To demonstrate the effectiveness of our approach in preventing novel web-based attacks from accessing sensitive information, we performed a number of tests in a live setting with the three different web applications [1].

For the experiments, we installed `phPay`, `myBloggie`, and `punbb` on a 2 GHz Pentium 4 machine with 1 gigabyte of RAM running Linux and the Apache web server. The web server was configured to run two name-based virtual hosts: `smart` and `dumb`. In this configuration, Apache determines the actual host to be used to serve an incoming request looking at the value of the `Host` field in the HTTP header section. To further improve the compartmentalization, we installed the `suPhp` module, which extends Apache to execute PHP pages with different user privileges in the two virtual hosts.

The web server was protected by our anomaly-driven reverse proxy, which was able to parse the incoming requests and rewrite the `Host` header on the fly to properly forward the traffic to the correct host. The reverse proxy was installed on a separate double 2.0 GHz Xeon server.

The two virtual hosts were connected to the same back-end MySQL database, but `smart` was able to access all of the application-specific content, while `dumb` was only able to access non-sensitive information. Moreover, the communication between the PHP code on `smart` and the database was filtered by our database anomaly detection component. The testbed architecture is shown in Figure 6.

To generate the traffic required to train the anomaly detection algorithms, we developed a number of Python scripts. For each application, we identified the list of available atomic operations (e.g., post a new message, browse the product catalog, search for a specific keyword in the text) and then aggregated these

[1] The code used in our experiments can be downloaded from `http://www.cs.ucsb.edu/~seclab/projects/revproxy/index.html`

operations to model typical user behaviors. For instance, a common behavior of a blog application user consists in going to the home page, reading some recent messages, and finally posting a comment for one of them. Each behavior was then executed with a random probability in order to generate a traffic that mimics the navigation of different users at the time.

The traffic generation scripts relied on a set of libraries we developed to increase the realism of the outgoing traffic. In particular, one library was responsible for manipulating the HTTP headers in order to mimic the characteristics of different web browsers. A second library was responsible for creating random user identities to be used in registration forms. In this case, we relied on a database containing a very large number of real common names, surnames, zip codes, and cities. Using this information, the library was able to generate names, addresses, company names, user IDs, passwords, and credit card numbers.

Table 3 shows the results of the first set of experiments. The anomaly detection components were trained in a fully automated way until all models completed the learning phase and automatically switched to detection. The SQL anomaly detector does not have this feature and can be trained with an arbitrary number of SQL queries.

Once the training phase was completed, we switched the IDS components to detection mode and ran them again on clean traffic generated by our scripts in order to test their false positives rates. The *anomalous* column in Table 3 shows the number of either HTTP or SQL requests identified as anomalous by the two components. For the `myBloggie` application, the SQL anomaly detector did not generate any false positives, while the reverse proxy marked 19 benign requests as anomalous, resulting in a somewhat high 0.32% false positive rate. Even though these requests were forwarded to the `dumb` virtual host, all of them were correctly served since none of them required access to sensitive information. This demonstrates that our technique is indeed effective in reducing the negative impact of false positives. In fact, all of the requests were actually served by one or the other virtual host, preventing the end user from experiencing any service interruption due to the presence of the anomaly detectors.

For `phPay` and `punbb`, the web proxy was much more accurate. The SQL anomaly system, however, marked four legitimate queries as anomalous (corresponding to a false positive rate of 0.00215%). These four requests were two different queries, each executed twice and. Given the high number of requests, we claim that this is an acceptable false positive rate.

For the second set of experiments, which is summarized in Table 4, we used a smaller test set containing

a large amount of malicious requests generated using publicly available exploits (Bugtraq ID 14195/a, Bugtraq ID 13507, Bugtraq ID 14195/b) and *ad hoc* fuzzing techniques. These attacks included SQL injections, command injections, information tampering, and cross-site scripting attacks. The results we obtained from this test strongly support the effectiveness of our approach based on the combination of anomaly detectors. In fact, the impact of the SQL component is almost negligible when the web request IDS detects most of the attacks, as it is the case for the `myBloggie` application. However, when the number of false negatives increases and the reverse proxy starts forwarding malicious requests to the `smart` host, the database IDS plays a key role in preventing the attacks from accessing sensitive information. For example, in `punBB` the database IDS was able to reduce the false negative rate by 33% to 1%. Moreover, anomaly characterizations for these attacks were propagated back to the web-based IDS to automatically deflect future instances of similar attacks to the restricted virtual host `dumb`. Consequently, considering all the three applications, only two malicious requests were able to avoid the composition of anomaly detectors.

These experiments demonstrate that our approach is able to effectively reduce both the impact of false positives and the number of false negatives for a representative class of real-world web-based applications.

6.3 Performance Overhead

The prototype implementation of our approach consisted of two components: the reverse HTTP proxy that integrates the web-based anomaly detector and the SQL anomaly detector installed on the machine that runs the web applications.

To quantify the performance of the reverse proxy, we run a stress test involving 1000 concurrent clients performing a total of 10000 requests on a set of various pages. In the experiment, the system was able to process 874 requests per seconds, resulting in an overhead of about 1.2 milliseconds per request.

The SQL anomaly components was instead tested on a set of more than 100000 queries extracted from a real word application (see [27] for more details on the experiment). In average, the SQL anomaly component spent 0.39ms of CPU time per query. Unfortunately, it is hard to combine the two performance values together, since the overhead of the web-based IDS is measured for each request, while the overhead of the SQL anomaly component depends on the actual number of queries performed by the page.

However, the overall performance of the system are quite satisfying, confirming the idea that this ap-

proach can be successfully applied in a real world scenario.

7 Related Work

The detection of web-based attacks has recently received considerable attention because of the increasingly critical role that web-based services are playing. For example, Almgren et al. [2] present a system that analyzes web logs looking for patterns of known attacks. A different type of analysis is performed in [3] where the detection process is integrated with the web server application itself. Vigna et al. [30] present a misuse-based system that operates on multiple event streams (i.e., network traffic, system call logs, and web server logs) is proposed. The system demonstrates that it is possible to achieve better detection results when taking advantage of the specificity of a particular application domain.

The identification of web attacks is a critical component of our architecture, and we use two anomaly-based intrusion detection systems that have been previously presented by Kruegel et al [11] and by Valeur et al. [27] to perform this task. Anomaly detection systems can detect novel attacks, but they are also prone to making mistakes and incorrectly classifying legitimate requests as malicious. Thus, it is important to develop strategies that can deal with false positives.

For example, different techniques have been presented that help an intrusion detection system to choose the most adequate response, given a number of choices [6, 26] . These techniques rely on cost models [14] that evaluate the impact of a (possible incorrect) decision of the IDS on the current state of the network and the mission goals. A different approach to mitigate the false positive problem is presented by Tombini et al. [24] where an anomaly detection system is used to filter out normal events so that signature-based detection is applied to anomalous requests only. This technique reduces the false positives generated by a misuse detection system and, in a way, is complementary to our approach.

In the approaches described above, each individual action (e.g., an HTTP request) must be either allowed or denied. In contrast to that, we aim to serve all requests as completely as possible. Of course, it might not always be possible to satisfy anomalous requests because they are executed in a more limited environment.

Our approach is closely related to the use of hardened systems [23]. The system presented there is composed of an anomaly detection system that uses abstract payload execution [25] and payload sifting [1] techniques to identify web requests that might contain attacks that exploit memory violations (e.g., buffer

overflows and heap overflows). The requests that are identified as anomalous are then marked appropriately and processed by a hardened, "shadow" version of the web server. To implement this approach, the Apache web server was modified to allow for the detection of memory violation attacks and the roll-back of modifications performed by malicious requests. While there are similarities between the approach proposed by Sidiroglou et al. [23] and ours, there are several major differences, the most significant of which is their system's limitation to a single class of attacks, namely that of control flow data corruption.

Prior work by Lee et al. has considered the application of learning techniques to the problem of identifying web-based attacks on databases [13]. Lee primarily focuses on recognizing SQL injection attacks as queries that are structurally dissimilar from normal queries observed during a training period. SQL injection vulnerabilities appear in server-side executables (e.g., applications invoked through the Common Gateway Interface) when values supplied by the client are used directly to assemble SQL queries issued by the executable, with little or no input validation checks. The structure matching approach proposed by Lee addresses this particular problem; however, we note that a form of mimicry attack is possible against such a detection mechanism, while our system is significantly more resistant to this class of attack.

8 Conclusions

This paper presents a novel technique to increase the detection rate of web-based intrusion detection systems by serially composing a web request anomaly detector and a SQL query anomaly detector. To address the system's capacity for producing false positives, we additionally present an approach to provide differentiated access to a web site based on the anomaly score associated with web requests. This design allows one to route anomalous requests to servers that have limited access to sensitive information. In this way, it is possible to contain the potential damage in the case of an attack, and, at the same time, maintain some level of service to anomalous queries that do not require access to sensitive information.

We implemented a prototype that composes an existing web-based anomaly detection system, a reverse HTTP proxy, and a database anomaly detection system and performed an extensive evaluation of the prototype to analyze the feasibility of our approach. To evaluate the impact that false positives would have on the accessibility of the system, we built a PHP analyzer and developed a metric that characterizes the percentage of execution paths invoking critical database accesses. We then analyzed several existing web-based

applications. The results show that our approach is practical and applicable to real-world web applications.

Acknowledgments

This research was supported by the Army Research Office, under agreement DAAD19-01-1-0484, and by the National Science Foundation, under grants CCR-0238492 and CCR-0524853.

References

[1] P. Akritidis, K. Anagnostakis, and E. Markatos. Efficient Content-Based Detection of Zero-Day Worms. In *Proceedings of the International Conference on Communications (ICC)*, Seoul, Korea, May 2005.

[2] M. Almgren, H. Debar, and M. Dacier. A lightweight tool for detecting web server attacks. In *Proceedings of the ISOC Symposium on Network and Distributed Systems Security*, San Diego, CA, February 2000.

[3] M. Almgren and U. Lindqvist. Application-Integrated Data Collection for Security Monitoring. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, LNCS, pages 22–36, Davis, CA, October 2001. Springer.

[4] R. Andersson. punBB - fast and lightweight PHP-powered discussion board. `http://www.punbb.org/`, 2005.

[5] S. Axelsson. The Base-Rate Fallacy and its Implications for the Difficulty of Intrusion Detection. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, 1999.

[6] I. Balepin, S. Maltsev, J. Rowe, and K. Levitt. Using Specification-Based Intrusion Detection for Automated Response. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, 2003.

[7] Breach Security. Breachgate. http://www.breach.com/, August 2006.

[8] Citrix. Citrix application firewall. http://www.citrix.com/, August 2006.

[9] Common Vulnerabilities and Exposures. http://www.cve.mitre.org/, 2003.

[10] A. Hayter. *Probability and Statistics for Engineers and Scientists*. Duxbury Press, 2 edition, 2001.

[11] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In *Proceedings of the* 10th *ACM Conference on Computer and Communication Security (CCS '03)*, pages 251–261, Washington, DC, October 2003. ACM Press.

[12] 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.

[13] S. Lee, W. Low, and P. Wong. Learning Fingerprints for a Database Intrusion Detection System. In *7th European Symposium on Research in Computer Security (ESORICS)*, 2002.

[14] W. Lee, W. Fan, M. Miller, S. Stolfo, and E. Zadok. Toward Cost-Sensitive Modeling for Intrusion Detection and Response. *Journal of Computer Security*, 10(1), 2002.

[15] D. Mutz, F. Valeur, C. Kruegel, and G. Vigna. Anomalous System Call Detection. *ACM Transactions on Information and System Security*, 2006.

[16] myBloggie - PHP and mySQL Blog / Weblog script. http://mybloggie.mywebland.com/, 2005.

[17] MySQL - The world's most popular open-source database. http://www.mysql.com/, 2005.

[18] NetContinuum. Nc-1100 af. http://www.netcontinuum.com/, August 2006.

[19] PHP: Hypertext Preprocessor. http://www.php.net/, 2005.

[20] phPay - webshop or catalog based on SQL and PHP. http://phpay.sourceforge.net/, 2005.

[21] K. Poulsen. Tower records settles charges over hack attacks. http://www.securityfocus.com/news/8508, April 2004.

[22] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX LISA '99 Conference*, Seattle, WA, November 1999.

[23] K. A. S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *Proceeding of the USENIX Security Symposium*, Baltimore, MD, August 2005.

[24] E. Tombini, H. Debar, L. Mé, and M. Ducassé. A Serial Combination of Anomaly and Misuse IDSes Applied to HTTP Traffic. In *Proceedings of the Annual Computer Security Aapplications Cconference (ACSAC)*, Tucson, AZ, December 2004.

[25] T. Toth and C. Kruegel. Accurate Buffer Overflow Detection Via Abstract Payload Execution. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, Zurich, Switzerland, October 2002.

[26] T. Toth and C. Kruegel. Evaluating the Impact of Automated Intrusion Response Mechanisms. In *Proceedings of the Annual Computer Security Application Conference (ACSAC)*, 2002.

[27] 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.

[28] 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.

[29] Victoria's Secret Reveals Too Much. http://www.cbsnews.com/, October 2003.

[30] G. Vigna, W. Robertson, V. Kher, and R. Kemmerer. A Stateful Intrusion Detection System for World-Wide Web Servers. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003)*, pages 34–43, Las Vegas, NV, December 2003.

Year	Total	Web-related	Percentage
1999	1561	236	15.1%
2000	1229	312	25.4%
2001	1526	419	27.5%
2002	2132	790	37.1%
2003	1204	350	29.1%
2004	2523	927	36.7%
2005	4501	2208	49.1%
2006	4099	2486	60.6%
Total	18775	7728	41.2%

Table 1: Percentage of web-related attacks in the Common Vulnerabilities and Exposure database, per year. The data is incomplete for year 2006. The table presents the results for both accepted entries and candidate entries.

Name	Source Files	Lines of Code	Database Tables
`phPay 2.0`	43	3,023	29
`myBloggie 2.1.2`	41	5,277	4
`punbb 1.2.5`	56	15,993	17

Table 2: Applications used for the experiments.

Program		Training Set	Clean Test Set	Anomalous	Failed Benign Requests
`myBloggie`	web requests	55K	6K	19	0
	SQL queries	550K	190K	0	
`phPay`	web requests	272K	5K	0	0
	SQL queries	10.4M	186K	2	
`Punbb`	web requests	360K	9K	2	1
	SQL queries	2.1M	51K	2	

Table 3: Live experiment results (training and false positives).

Program	Test Set	Malicious Requests	Anomalous (web)	Anomalous (SQL)	Successful Attacks
`myBloggie`	web requests 400 SQL queries 8K	100	99	0	1
`phPay`	web requests 1.1K SQL queries 61K	100	90	10	0
`Punbb`	web requests 560 SQL queries 3.5K	100	67	32	1

Table 4: Live experiment results (malicious traffic)

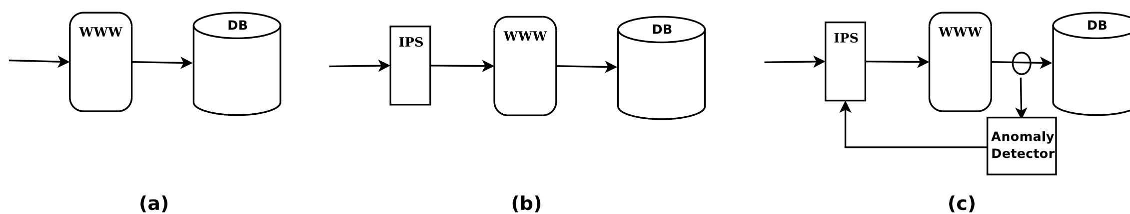

Figure 1: System Architecture

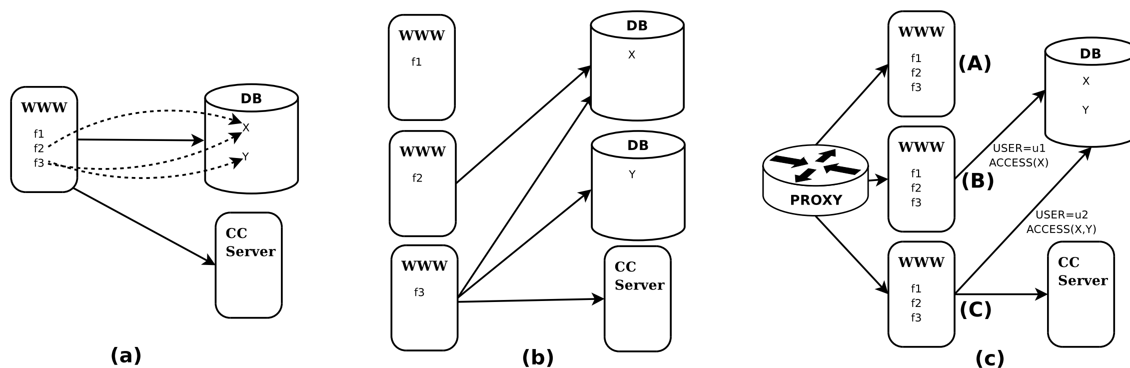

Figure 2: Web site designs.

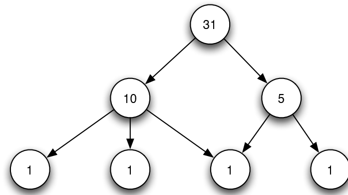

Figure 3: Call graph and path calculation.

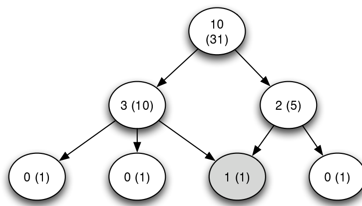

Figure 4: Call graph and sensitive path calculation.

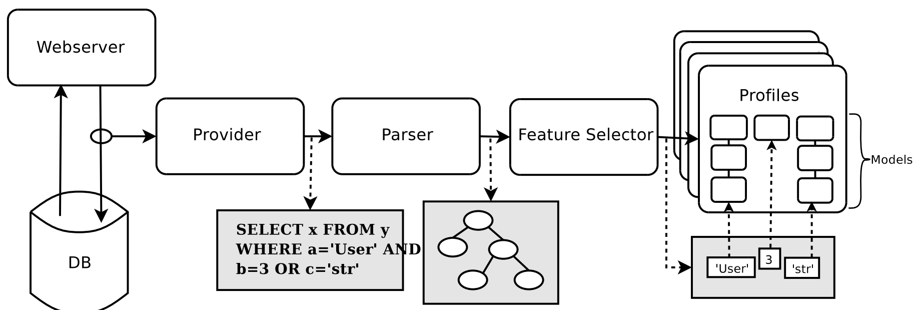

Figure 5: Overview of the database anomaly detection component.

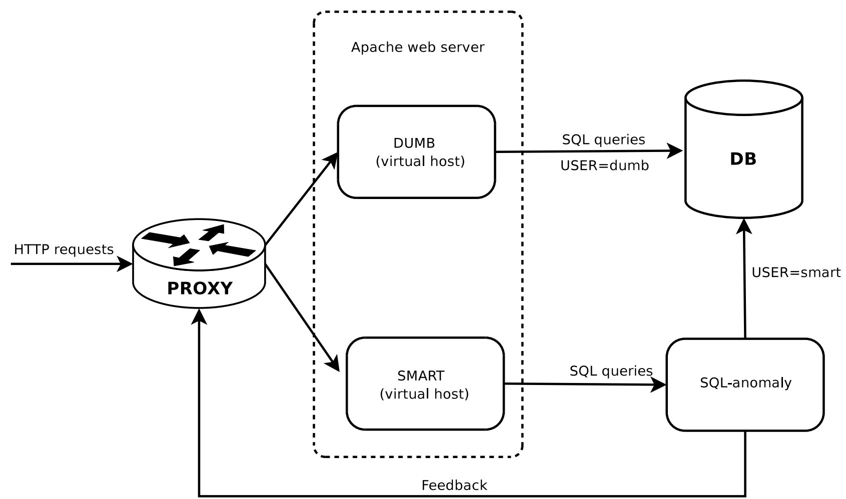

Figure 6: Testbed architecture