A Learning-Based Approach to the Detection of SQL Attacks

Fredrik Valeur, Darren Mutz, and Giovanni Vigna

Reliable Software Group
Department of Computer Science
University of California, Santa Barbara
{fredrik,dhm,vigna}@cs.ucsb.edu

Abstract. Web-based systems are often a composition of infrastructure components, such as web servers and databases, and of application-specific code, such as HTML-embedded scripts and server-side applications. While the infrastructure components are usually developed by experienced programmers with solid security skills, the application-specific code is often developed under strict time constraints by programmers with little security training. As a result, vulnerable web-applications are deployed and made available to the Internet at large, creating easily-exploitable entry points for the compromise of entire networks.

Web-based applications often rely on back-end database servers to manage application-specific persistent state. The data is usually extracted by performing queries that are assembled using input provided by the users of the applications. If user input is not sanitized correctly, it is possible to mount a variety of attacks that leverage web-based applications to compromise the security of back-end databases. Unfortunately, it is not always possible to identify these attacks using signature-based intrusion detection systems, because of the *ad hoc* nature of many web-based applications. Signatures are rarely written for this class of applications due to the substantial investment of time and expertise this would require.

We have developed an anomaly-based system that learns the profiles of the normal database access performed by web-based applications using a number of different models. These models allow for the detection of unknown attacks with reduced false positives and limited overhead. In addition, our solution represents an improvement with respect to previous approaches because it reduces the possibility of executing SQL-based mimicry attacks.

Keywords: Intrusion Detection, Machine Learning, Web Attacks, Databases.

1 Introduction

Web-based applications have become a popular way to provide access to services and dynamically-generated information. Even network devices and traditional applications (such as mail servers) often provide web-based interfaces that are used for administration as well as configuration.

Web-based applications are implemented using a number of server-side executable components, such as CGI programs and HTML-embedded scripting code, that access back-end systems, such as databases[1]. For example, a popular platform to develop web-based applications is a combination of the Linux operating system, the Apache web server, the MySQL database engine, and the PHP language interpreter, which, together, are referred to as a "LAMP" system.

Unfortunately, while the developers of the software infrastructure (i.e., the developers of web servers and database engines) usually have a deep understanding of the security issues associated with the development of critical software, the developers of web-based applications often have little or no security skills. These developers mostly focus on the functionality to be provided to the end-user and often work under strict time constraints, without the resources (or the knowledge) necessary to perform a thorough security analysis of the applications being developed. The result is that poorly-developed code, riddled with security flaws, is deployed and made accessible to the whole Internet.

Because of their immediate accessibility and their poor security, web-based applications have become popular attack targets and one of the main avenues by which the security of systems and networks are compromised. In addition, the large installation base makes both web applications and servers a privileged target for worm programs that exploit web-related vulnerabilities to spread across networks [6].

Existing prevention systems are often insufficient to protect this class of applications, because the security mechanisms provided are either not well-understood or simply disabled by the web developers "to get the job done." Existing signature-based intrusion detection systems are not sufficient either. Web-applications often implement custom, site-specific services for which there is no known signature, and organizations are often unwilling or unable to commit the substantial time and expertise required to write reliable, high quality signatures. Therefore, prevention mechanisms and signature-based detection systems should be complemented by anomaly detection systems, which learn the normal usage profiles associated with web-based applications and identify attacks as anomalous deviations from the established profiles.

This paper presents an anomaly detection approach for the detection of attacks that exploit vulnerabilities in Web-based applications to compromise a back-end database. Our approach uses multiple models to characterize the profiles of normal access to the database. These profiles are learned automatically during a training phase by analyzing a number of sample database accesses. Then, during the detection phase, the system is able to identify anomalous queries that might be associated with an attack.

We developed an intrusion detection system based on our approach by leveraging an object-oriented framework for the development of anomaly detection systems that we implemented as part of our previous research [1]. The framework allowed us to implement a working system with reduced effort. The evaluation

[1] Web-based applications also use client-side execution mechanisms, such as JavaScript and ActiveX, to create richer user-interfaces. However, hereinafter we focus only on the server-side part of web-based applications.

of our preliminary prototype shows that our approach is able to detect unknown attacks with a limited number of false positives.

This paper is structured as follows. Section 2 discusses several classes of attacks against database systems. Section 3 discusses related work. Section 4 presents our intrusion detection tool. Section 5 describes the anomaly detection models used to characterize normal behavior. Next, Section 6 discusses the evaluation of our tool. Finally, Section 7 draws conclusions and outlines future work.

2 SQL-based attacks

In this paper we consider three classes of SQL-based attacks. *SQL injection*, which allows the attacker to inject strings into the application that are interpreted as SQL statements, *Cross-site scripting*, which allows for the execution of client-side code in privileged contexts, and *data-centric attacks*, which allow the attacker to insert data which are not part of the expected value range into the database.

2.1 SQL Injection

SQL injection is a class of attacks where un-sanitized user input is able to change the structure of an SQL query so that when it is executed it has an unintended effect on the database. SQL injection is made possible by the fact that SQL queries are usually assembled by performing a series of string concatenations of static strings and variables. If the variables used in the creation of the query are under the control of the user, she might be able to change the meaning of the query in an undesirable way. Consider a web-based application that lets the user list all her registered credit cards of a given type. The pseudocode for this functionality might be as follows:

```
uname = getAuthenticatedUser()
cctype = getUserInput()
result = sql("SELECT nb FROM creditcards WHERE user='"
            + uname + "' AND type='" + cctype +"';")
print(result)
```

If the user `bob` does a search for all his VISA cards the following query would be executed: `SELECT nb FROM creditcards WHERE user='bob' AND type='VISA';`. This example code contains an SQL injection vulnerability. If Bob wants to view all the credit cards belonging to user `alice` he could ask for a list of cards of type `' OR user ='alice`. This would cause the following query to be executed: `SELECT nb FROM creditcards WHERE user='bob' AND type='' OR user='alice';`. This query returns a list of all Alice's credit cards to the attacker.

The correct implementation of the application shown above should not allow data supplied by the user to change the structure of the query. In general, the user-supplied part of the SQL query should not be interpreted as SQL keywords, table names, field names or operators by the SQL server. The remaining parts of the SQL query, which we will refer to as *constants*, consist of quoted strings and numbers. Before utilizing user data as constants care must be taken to ensure that all quotes in user-supplied strings are escaped before inserting them into the SQL query. Similarly, user-supplied numbers must be checked to verify that they are numbers and not strings. In the example above, SQL injection is possible because the string `cctype` is not properly escaped before it is inserted into the query.

2.2 Cross Site Scripting

Cross site scripting attacks (XSS), are an important class of attacks against web-based applications. These attacks exploit trust relationships between web servers and web browsers by injecting a script (often written in JavaScript) into a server that is not under the control of the attacker. JavaScript [8] is a scripting language developed by Netscape to create interactive HTML pages. In most cases, JavaScript code is embedded in HTML code. When a JavaScript-enabled browser downloads a page, it parses, compiles, and executes the script. As with other mobile code schemes, malicious JavaScript programs can take advantage of the fact that they are executed in a foreign environment that contains sensitive information.

Existing JavaScript security mechanisms are based on sand-boxing, which only allows the code to perform a restricted set of operations. JavaScript programs are treated as untrusted software components that have access to a limited number of resources within the browser. The shortcoming of this solution is that scripts may conform to the sand-box policy, but still violate the security of the system.

The general outline of a cross site scripting attack is the following. First, a malicious user uploads HTML code containing JavaScript to a web service. Next, if the uploaded code is viewable by other users, the malicious script will be executed in the victims' browsers. Since the script originates from the web server it is run with the same privileges as legitimate scripts originating from the server. This is a problem if the victim has a trust relationship with the domain hosting the web server, since the malicious script could be able to access sensitive data associated with that domain. Often these kinds of attacks are used to steal login credentials or other personal information from users.

If data submitted by the users of a web-based application is inserted into a database, cross-site scripting attempts can be observed at the database level and can be considered a data-centric attack. Since the malicious scripts are visible in the SQL queries when the data is inserted into the database, it is possible to detect cross site scripting attempts by observing all values as they are inserted and alert if any sign of a script is detected.

2.3 Other Data-centric Attacks

Other classes of attacks can also be detected by looking at the query constants. For instance, it is often the case that a certain database field should only take on a limited number of values. A usertype field might have the values of `Employee` or `Contractor`. If a usertype of `xxx` is seen, this might be evidence of an attack.

A more complex data-centric attack is the *two-step SQL injection attack*. In this case, the attacker inserts a specially crafted string into the database that causes an SQL injection when it is processed at a later time. As an example of this attack, consider the following scenario. A web site allows users to sign up with whatever username they desire. The web site periodically deletes inactive users with the following script:

```
old = now() - 3 months
users = sql("SELECT uname FROM users
                   WHERE last_login < "+old+";")
for u in users:
   sql("DELETE FROM users WHERE uname='" + u + "';")
```

If a user is allowed to sign up with any username this code is vulnerable to a two-step SQL injection attack. The attacker first creates a user named `' OR '1' = '1`. Assuming the user creation code is free from SQL injection vulnerabilities, the system correctly creates a new user with the following SQL statement: `INSERT INTO USERS VALUES ('\' OR \'1\' = \'1');`. Note that this is not an SQL injection attack since all user supplied quotes are properly escaped. The true attack is executed when the periodical cleanup script is run and the script tries to delete this user. Because of the carefully selected username, the script generates the following query to delete the user: `DELETE FROM users WHERE uname='' OR '1' = '1';`. Since the expression `'1' = '1'` is always true, this statement would delete all users in the database.

3 Related Work

Learning-based anomaly detection represents a class of approaches that relies on training data to build profiles of the normal, benign behavior of users and applications. Various types of learning-based anomaly detection techniques have been proposed to analyze different data streams. A common approach is to use data-mining techniques to characterize network traffic. For example, in [16] the authors apply clustering techniques to unlabeled network traces to identify intrusion patterns. Statistical techniques have also been used to model the behavior of network worms [14]. Other approaches use statistical analysis to characterize user behavior. For example, the seminal work by Denning builds user profiles using login times and the actions that users perform [7].

A particular class of learning-based anomaly detection approaches focuses on the characteristics of specific applications and the protocols they use. For example, in [9] and [23] sequence analysis is applied to system calls produced by specific applications in order to identify "normal" system call sequences for a certain application. These application-specific profiles are then used to identify attacks that produce previously unseen sequences. As another example, in [15] the authors use statistical analysis of network traffic to learn the normal behavior of network-based applications. This is done by analyzing both packet header information (e.g., source/destination ports, packet size) and the contents of application-specific protocols.

Our approach is similar to these techniques because it characterizes the benign, normal use of specific programs, that is, databases that are accessed by web-based applications. However, our approach differs in two ways. First of all, we employ a number of different models to characterize the behavior of web-based applications. By using multiple models it is possible to reduce the susceptibility of the detection process to *mimicry attacks* [22,20]. Second, the models target specific types of applications and, therefore, they allow for more focused analysis of the data transferred between the client (the attacker) and the server-side program (the victim). This is an advantage of application-specific intrusion detection in general [11] and of web-based intrusion detection in particular [12].

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, in [2] the authors 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. In [21], a misuse-based system that operates on multiple event streams (i.e., network traffic, system call logs, and web server logs) is proposed. Also, a commercial systems exists that analyzes HTTP requests [24]. Systems that focus on web-based attacks show that, by taking advantage of the specificity of a particular application domain, it is possible to achieve better detection results. However, these systems are mostly misuse-based and therefore suffer from the problem of not being able to detect attacks that have not been previously modeled. Our approach is similar to these systems because it focuses on web-based applications. However, the goal of our tool is to perform autonomous, learning-based anomaly detection requiring minimal human oversight. The tool can be deployed on a host that contains custom-developed server-side programs and are able to automatically derive models of the manner in which these programs access a back-end database. These models are then used to detect known and unknown attacks.

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.

While the structure matching approach proposed by Lee addresses this problem, we note that a form of mimicry attack is possible against such a detection mechanism. In particular, large-scale web sites may contain hundreds of server-side executables that may each be capable of issuing multiple database queries. A mimicry attack is possible in a system monitored by a system such as Lee's if the attacker is able to construct a malicious SQL query that structurally matches one of the queries legitimately issued by any other part of the system.

Our system addresses this potential shortcoming by maintaining associations between individual server-side executables and the structure of the queries they issue. We note that an additional, more restrictive mimicry attack is possible against systems containing executables that issue multiple queries. In this case, if an attacker is able to find another query structure within a single server-side executable that matches the structure of her attack query, the attack will not be detected. Tracking associations at a finer level of detail is possible (e.g., through instrumentation of executables), and will be implemented in a future version of our system.

4 Detecting Anomalous SQL Queries

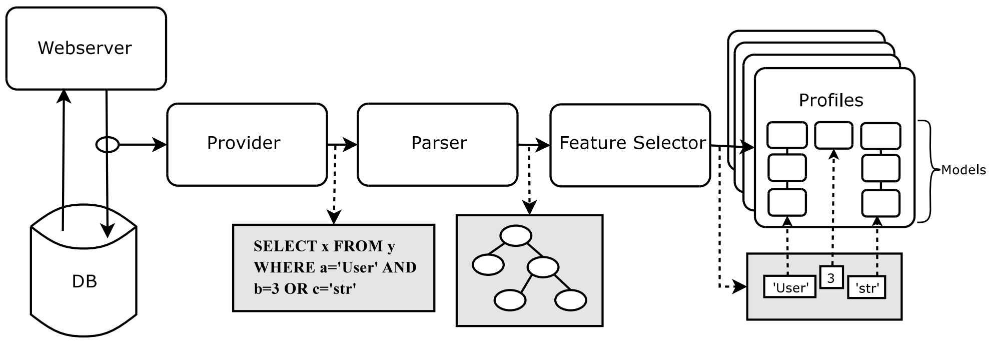

Fig. 1. Overview of the System

We have developed an intrusion detection system that utilizes multiple anomaly detection models to detect attacks against back-end SQL databases. In the following we describe the architecture of our system. Then, in section 5 we describe further the models used by our system. Figure 1 shows an overview of the architecture of our system. 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. The IDS parses the SQL statements and selects what features of the query should be modeled. A type inference process is performed on the selected features in order to support the selection of correct statistical models to be applied to the event, before a *profile* is selected. A profile is a collection of models, which the features are fed to in order to train the set of models or to generate an anomaly score.

Our system is a learning-based anomaly detector, and thus requires that a training phase is performed prior to detection. The training phase is divided into two halves. During the first half of the training phase, the data fed to the models is used for building the profiles associated with the models' parameters. It is assumed that the data processed in the training phase is attack-free and, therefore, during this phase the models learn what normal queries look like. In the second half of the training phase, the model parameters are not updated. Instead an anomaly score is calculated based on how well the processed features fit the trained models. For each model, the maximum anomaly score seen during the second half of the training period is stored and used to set an anomaly threshold.

During the following detection phase, anomaly scores are calculated for each query. If an anomaly score exceeds the maximum anomaly score seen during training by a certain tunable percentage, the query is considered anomalous and an alert is generated.

4.1 Event Provider

The event provider is responsible for supplying the intrusion detection system with a stream of SQL queries. It is important that the event provider report every SQL statement performed by the monitored application. Since nothing can be assumed about the quality of the application, the provider does not rely on application-specific mechanisms to perform the reporting. The event provider operates on the application server, because the server environment has access to information about the process performing the query and can log security-relevant information, such as the filename of the script currently executing. The logging is implemented by utilizing modified versions of the system libraries that provide connectivity between the application and the database.

4.2 Parser

The parser processes each incoming SQL query generating a high level-view of the query. The parser outputs this representation as a sequence of tokens. Each token has a flag which indicates whether the token is a constant or not. Constants are the only elements of an SQL query that should contain user supplied input.

Tokens representing database field names are augmented by a *datatype* attribute. The datatype 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. The generated mapping can be updated by the user if it is desirable to describe the datatype 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 an easy interface to specify how new data types should be processed by the intrusion detection system.

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

– A constant 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 constant that is inserted into a table has its datatype set to the datatype of the field it is inserted into.

4.3 Feature Selector

The feature selector transforms the queries into a form suitable for processing by the models. In addition it selects which profile each query applies to.

First, a feature vector is created by extracting all tokens marked as constant 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 constants 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 constants, different user inputs should result in the same skeleton. An SQL injection would change the structure of the query and produce a different skeleton query.

The next step depends on the status of the intrusion detection system, that is, if the system is in training, threshold learning, 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 connected to models capable of modeling numerical elements. For user-defined types, the user can specify which models should be instantiated. The specific models used in our system are described in more detail in Section 5.

If the system is in threshold learning mode, the corresponding profile is looked up the same way as in the training mode, but the feature vector is not used to update the models. Instead, the models are used to generate an anomaly score that measures how well the feature vector fits the models. An aggregate score is calculated as the sum of the negative logarithm of each individual model score as in [10]. For each profile the highest aggregate anomaly score seen during the threshold learning phase is recorded. If no profile is found for an event, a warning is printed that indicates that the previous training phase was not complete.

In detection mode, an anomaly score is calculated in a way similar to the previous mode, but differently, if the anomaly score exceeds the max value recorded in the threshold recognition phase by a certain percentage, 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.

4.4 Implementation

Our implementation uses a modified version of the libmysqlclient library, which logs all performed SQL queries. Libmysqlclient is part of the MySQL database system and most applications that supports the MySQL database utilize this library to communicate with the database server. The provider logs all queries to a file which is read by the sensor.

The sensor is implemented in C++. The incoming queries are parsed by a Yacc-based parser. After parsing and type inference, the events are fed to the detection engine. The detection engine is implemented as an extension of our anomaly-detection framework, called libAnomaly [1]. LibAnomaly provides a number of useful abstract entities for the creation of anomaly-based intrusion detection systems and makes frequently-used detection techniques readily available. libAnomaly has previously been used to implement anomaly detectors that processes system call traces and web logs [10,12].

5 Anomaly Detection Models

Different statistical models are used depending on what data type is modeled. In our implementation, two basic data types are supported. Strings and integers. The string data type is modeled by six different models, namely five string-based models plus a data type independent model. Integers are only modeled by the data type independent model. These models are described in the following section. See [10] for a more in-depth description of the different models.

5.1 String Models

String Length The goal of the string length model is to approximate the actual but unknown distribution of the lengths of string values and to detect instances that significantly deviate from the observed normal behavior. For example, system call string arguments often represent canonical file names that point to an entry in the file system. These arguments are commonly used when files are accessed (`open`, `stat`) or executed (`execve`), and their lengths rarely exceed a hundred characters. However, when a malicious input is passed to programs, it often occurs that this input also appears in an argument of a system call with a length of several hundred bytes. The detection of significant deviations is based on the Chebyshev inequality [4].

String Character Distribution The string character distribution model captures the concept of a normal string argument by looking at its character distribution. The approach is based on the observation that strings have a regular structure, are often human-readable, and almost always contain only printable characters. In case of attacks that send executable data, a completely different character distribution can be observed. This is also true for attacks that send many repetitions of a single character (e.g., the `nop`-sledge of a buffer overflow attack). The detection of deviating arguments is performed by a statistical test (Pearson χ^2-test) that determines the probability that the character distribution of a string parameter fits the normal distribution established during the training phase.

String Prefix and Suffix Matcher The length and character distribution are two features that provide a ball-park measure of the regularity of a string. Sometimes, however, it is desirable to capture the structure of a string in a more precise fashion. The idea of the prefix and suffix matcher model is to capture substrings that are shared by the value of specific elements in an event. In particular, these models can be applied to elements that represent file names. For example, the prefixes of file name arguments might indicate that all files are located in the same directory or under a common directory root (e.g., a user's home directory or the document root directory of the web server). The suffixes of file names are often indicators of the file types that are accessed. A web server, for example, can be expected to mostly access files with a `htm[l]` ending when these files are located under the document root. To build a model of normal string prefixes and suffixes, the first and last n characters of each string are extracted during the training phase. Whenever a certain (large) fraction of all analyzed strings has a certain prefix or suffix in common, the corresponding string is included into the set of known prefixes/suffixes. During the detection phase, when the set of known prefixes/suffixes is not empty, it is checked whether the characterized element value contains a known prefix or suffix. If this is the case, the input is tagged as normal, otherwise, it is considered anomalous.

String Structure Inference For the purposes of this model, the structure of an argument is the regular grammar that describes all of its normal, legitimate values. Thus, the task of the structural inference model is to extract a grammar that generates all legitimate elements. When structural inference is applied to a string element, the resulting grammar must be able to produce at least all elements encountered during the training phase. Unfortunately, there is no unique grammar that can be derived from a finite set of string elements. When no negative examples are given (i.e., elements that should not be derivable from the grammar), it is always possible to create either a grammar that contains exactly the training data or a grammar that allows production of arbitrary strings. The first case is a form of over-simplification, as the resulting grammar is only able to derive the learned input without providing any level of abstraction. This means that no new information is deduced. The second case is a form of over-generalization, because the grammar is capable of producing all possible strings, but there is no structural information left.

One possible approach for our proposed structural inference is to start with an automaton that exactly reflects the input data. Then, the grammar is generalized as long as it seems "reasonable", and the process is stopped before too much structural information is lost. We aim to implement the generalization process of this model based on the work presented in [18] and [19]. In these papers, the process of "reasonable generalization" is based on Bayes' theorem:

$$p(Model|TrainingData) = \frac{p(TrainingData|Model) * p(Model)}{p(TrainingData)}$$

We are interested in maximizing the *a posteriori* probability (left-hand side), thus, we have to maximize the product on the right-hand side of the equation. The first term, which is the probability of the training data given the model, can be calculated for a certain automaton directly from the training data. The second term, which is the prior probability of the model, is not so straightforward. It has to reflect the fact that, in general, smaller models are preferred. This probability is calculated heuristically, taking into account the number of states and transitions of the automaton. The denominator (i.e., probability of the training data) is considered a constant scaling factor that can be ignored.

During the detection phase, it is checked whether an input string argument can be generated by the automaton. If this is possible, the string is considered normal, otherwise it is flagged as anomalous. A more complete description of the implementation of this model can be found in [12].

5.2 Data Type-Independent Model

Token Finder The purpose of the token finder model is to determine whether the values of a certain element are drawn from a limited set of possible alternatives (i.e., they are tokens of an enumeration). Web-application often receive parameters that represent a selection among few possibilities presented to the user in an HTML form or that represent flag-like values, e.g., a certain type of credit card. When an attacker tries to exploit uncommon values of the parameter, previously unseen values may appear. This model is particularly effective in detecting these types of attacks. The decision between an enumeration and random values is made utilizing a simple statistical test, such as the non-parametric Kolmogorov-Smirnov variant as suggested in [13].

6 Discussion and Evaluation

We evaluated our system using an installation of the PHP-Nuke web portal system [5]. PHP-Nuke has a long history of security problems [17] and contains several SQL-based vulnerabilities.

Our test server was a 2 GHz Pentium 4 with 1 GB of RAM running Linux 2.6.1. The server was configured with an Apache web server (v2.0.52), the MySQL database (v4.1.8), and PHP-Nuke (v7.5).

Attack-free audit data was generated by manually operating the web site using a web browser and, at the same time, running scripts simulating user activity. PHP-Nuke is a fairly large system, so generating audit data by scripts alone would require a major development effort when creating the scripts. The test scripts we used only utilized the central functionality of PHP-Nuke. We relied on manual browsing to operate the less-used functionality. Three attack-free datasets were produced this way. The first was used for training the models, the second was used for the threshold learning phase, while the third was used for false positive rate estimation.

Dataset	# Queries	# Alerts	Correct Detect.	False Positives
Training	44035	N/A	N/A	N/A
Threshold Learning	13831	N/A	N/A	N/A
Attack1	25	1	1	0(0%)
Attack2	65	1	1	0(0%)
Attack3	173	6	6	0(0%)
Attack4	79	1	1	0(0%)
Attack Free	15704	58	0	58(.37%)
Attack Free W/ Custom Datatype	15704	2	0	2(.013%)

Table 1. Summary of system training and detection experiments

In order to evaluate the detection capabilities of our system, four different SQL-based attacks against PHP-Nuke were developed. The attacks were run against the test server while background traffic was generated by the user-simulation scripts. For each attack a dataset containing one attack instance was recorded. Our trained IDS was run against each of the attack datasets and the output was analyzed to check if the IDS was able to detect all the attacks.

6.1 Attacks

The three first attacks in our tests are performed by posting form-encoded data to a specific URL. For each of these attacks, we show what page contains the vulnerability and what data needs to be posted in order to exploit the system. We also show the SQL query that is produced as a consequence of the attack. Each of the attacks were discovered during our experimentation with PHP-Nuke and, to the best of the authors' knowledge, all attacks presented are novel.

Attack1: Resetting any users password

Vulnerable page phpnuke/modules.php

Post data `name='; UPDATE nuke_users SET user_password='<new md5pass>' WHERE username='<user>'; --`
Result `SELECT active, view FROM nuke_modules WHERE title='Statistics'; UPDATE nuke_users SET user_password='<new md5pass>' WHERE username='<user>'; --'`

This attack updates the password of an existing user. A variable used for passing the value `name` to the page modules.php is not escaped before inserting it into a query. This allows an attacker to set any users password to a value of her choosing by injecting an SQL `UPDATE` statement for the table `nuke_users`. The attack is detected by our system because the SQL statement violates the structural model. See Table 1 for details.

Attack2: Enumerating all users

Vulnerable page phpnuke/modules.php
Post data 1 `name=Your_Account`
Post data 2 `op=userinfo`
Post data 3 `username=' OR username LIKE 'A%'; --`
Result SELECT `uname FROM nuke_session WHERE uname='' OR username LIKE 'A%'; -- '`

This attack allows one to retrieve a list of all users of the system. The `username` value is not properly checked by the script that shows account information about the current user. By injecting a specially crafted string the attacker can select a user by an SQL wildcard expression. When executing the attack, the resulting page shows the first user in alphabetical order that matches the `LIKE` expression. To enumerate all the users, several executions of the attack are required. The following pseudocode would generate a user list:

```
getusers(prefix) {
  for letter in a...z:
     user = get first user that starts with
              prefix + letter
     if user is found:
        print user
        getusers(prefix+letter)
}

main() {
  getusers("")
}
```

This attack is also detected by our system because of a violation of the structural model, as shown in Table 1.

Attack3: Parallel password guessing

Vulnerable page phpnuke/modules.php
Post data 1 `name=Your_Account`
Post data 2 `username=' OR user_password = '<md5 password>' ;`
Post data 3 `user_password=<password>`

Result1 `SELECT user_password, user_id, .... FROM nuke_users WHERE username='' OR user_password = '<md5 password>' ;'`
Result2 `SELECT time FROM nuke_session WHERE uname='\' OR user_password = \'<md5 password>\' ;'`

This attacks allows one to speed up password guessing by trying a password against the whole user database in parallel. The attacker chooses a password to try and inserts both the password and an md5 checksum of it into the query. If any user on the system has that password, the login will succeed. Our system detects six anomalous SQL queries as a result of this attack. The first query is detected because the query structure is violated as a result of the injection. The structure of the second query shown is valid because it is not the result of an SQL injection. In spite of this, our system correctly marks this query as anomalous because the structure of the username is not similar to any username seen in the training data. The fact that different attacks are detected by different models demonstrates that a multi-model approach is able to detect more attacks by providing a more complete description of the web-application being modeled. The remaining 4 anomalous queries were similar to the second query.

Attack4: Cross site scripting The fourth attack is different in that it does not require posting of any data. Instead the attack is executed by retrieving any PHP-Nuke page and passing the JavaScript in the HTML referrer field. All referrer values received by PHP-Nuke is displayed unescaped on a statistics page. The script is executed when a user clicks on one of the links on PHP-Nuke's referrer statistics page.

In our test we passed the value `" onclick="alert(document.domain);"` as the referrer. This caused the following query to be executed: `INSERT INTO nuke_referer VALUES (NULL, '" onclick="alert(document.domain);"')` . This attack was detected by our system because the referer value had a different structure than the values seen during the training.

6.2 False Positive Rate

Traditionally, anomaly detection systems have been prone to generating high rates of false positives. We evaluated the false positive rate in our system by training the system as in the attack tests, and using an additional attack-free dataset as a detection set. This second attack-free set was generated in a way similar to the training sets, but the manual browsing of the web site was performed by a different person than the one generating the training data. This was done to ensure that the datasets were not artificially similar due to regularities in the browsing habits of a single person.

The results of the test are shown in Table 1, which shows the false positive rate to be fairly high. Inspection of the alarms generated by the IDS showed that this was due to fact that the training data was generated in a different month than the test data, and the IDS had only seen one value for the month field during the training period. When confronted with a new month value the

IDS reported this as an anomaly. We also identified a year field in the database that had a potential for generating false positives in a way similar to the month field. We changed the configuration of our system by introducing two custom data types: `month` and `year`. The models associated with these data types would consider any value within the normally acceptable range (i.e., months 1-12 would be accepted but not 13). Upon reevaluating the false positive rate, we observed a dramatic reduction in the number of false alarms, as can be seen in Table 1. The remaining two false positives were a result of queries not seen during the training period.

We believe that many installations of our system would require the introduction of custom data types similar to those mentioned above in order to produce an acceptably low positive rate. However, the introduction of a new data type is fairly easy and most database fields do not require any special treatment. Because of this we believe the system would be very easy to configure for a new application, even by persons with no special training in security.

6.3 Performance Overhead

A performance test of our system was performed to quantify the overhead introduced by our system. Our metrics provide only a rough estimation of what the overhead is. The performance overhead of a real deployment would be dependent on numerous factors such as the rate at which different pages are accessed, the number of queries executed for each page served, and the topology of the servers in the installation.

Our performance metrics measure the average number of CPU seconds spent by our tool per query processed. The number of CPU seconds spent by MySQL and Apache/PHP is given for comparison. Our experiment was conducted by running the IDS sensor in real time on the test server while executing the same user simulation scripts used to generate the training data. The number of CPU seconds spent by each component was recorded and an average per-query value was computed. Our test generated 105,612 queries. See Table 2 for the results. The performance of our system is quite good considering that no code optimization effort has been performed.

Process	Total CPU (s)	Per Query CPU (ms)
SqlAnomaly	41.3	.39
Apache/PHP	106.2	1.00
MySQL	22.0	.20

Table 2. Performance Metrics

7 Conclusions and Future Work

This paper presents a novel anomaly-based intrusion detection approach for the detection of attacks against back-end databases used by web-based applications. The approach relies on a composition of multiple models to characterize the normal behavior of web-based applications when accessing the database.

We developed a system based on this approach and evaluated its effectiveness by measuring its ability to detect novel attacks, its false positive rate, and the overhead introduced by the system. The results show that our system is indeed able to detect novel attacks with few false positives and little overhead. In addition, the learning-based approach utilized by the system makes it well-suited for deployment by administrators without extensive security expertise.

Our future research will focus on developing better models and on using additional event streams (such as the system calls executed by server-side executables) to more completely characterize the behavior of web-based systems. Furthermore, auditing of more complex database features such as stored procedures could be accommodated through the inclusion of the database activity log as a second event stream.

We plan to develop techniques to determine the coverage space of training data with respect to an existing system. These techniques will focus on static analysis of web-application code and on identifying high-level relationships between each component of a web-based system. This meta-information will then be leveraged to determine if the current training data provides sufficient coverage of the functionality of the systems and, as a result, reduce the possibility of generating false positives. For example, it will be possible to determine if all the parameters of a server-side application have been exercised by the training data or if all the pages that contain embedded code have been requested. The resulting models would have the advantage of added coverage during the training phase while still capturing installation-specific behaviors that are not statically inferable.

References

1. `libAnomaly` project homepage. `http://www.cs.ucsb.edu/~rsg/libAnomaly`.
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. P. Billingsley. *Probability and Measure*. Wiley-Interscience, 3rd edition, April 1995.
5. F. Burzi. Php-nuke website. `http://phpnuke.org/`, 2005.
6. CERT/CC. “Code Red Worm” Exploiting Buffer Overflow In IIS Indexing Service DLL. Advisory CA-2001-19, July 2001.
7. D.E. Denning. An Intrusion Detection Model. *IEEE Transactions on Software Engineering*, 13(2):222–232, February 1987.

8. David Flanagan. *JavaScript: The Definitive Guide, 4th Edition*. December 2001.
9. S. Forrest. A Sense of Self for UNIX Processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, CA, May 1996.
10. C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the Detection of Anomalous System Call Arguments. In *Proceedings of the* 8^{th} *European Symposium on Research in Computer Security (ESORICS '03)*, LNCS, pages 326–343, Gjovik, Norway, October 2003. Springer-Verlag.
11. C. Kruegel, T. Toth, and E. Kirda. Service Specific Anomaly Detection for Network Intrusion Detection. In *Symposium on Applied Computing (SAC)*. ACM Scientific Press, March 2002.
12. 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.
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. M. Liljenstam, D. Nicol, V. Berk, and R.Gray. Simulating realistic network worm traffic for worm warning system design and testing. In *Proceedings of the ACM Workshop on Rapid Malcode*, pages 24–33, Washington, DC, 2003.
15. M. Mahoney and P. Chan. Learning Nonstationary Models of Normal Network Traffic for Detecting Novel Attacks. In *Proceedings of the* 8^{th} *International Conference on Knowledge Discovery and Data Mining*, pages 376–385, Edmonton, Alberta, Canada, 2002.
16. L. Portnoy, E. Eskin, and S. Stolfo. Intrusion Detection with Unlabeled Data Using Clustering. In *Proceedings of ACM CSS Workshop on Data Mining Applied to Security*, Philadelphia, PA, November 2001.
17. Security Focus Homepage. http://www.securityfocus.com/, 2002.
18. A. Stolcke and S. Omohundro. Hidden Markov Model Induction by Bayesian Model Merging. *Advances in Neural Information Processing Systems*, 1993.
19. A. Stolcke and S. Omohundro. Inducing probabilistic grammars by bayesian model merging. In *International Conference on Grammatical Inference*, 1994.
20. K.M.C. Tan, K.S. Killourhy, and R.A. Maxion. Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits. In *Proceedings of the* 5^{th} *International Symposium on Recent Advances in Intrusion Detection*, pages 54–73, Zurich, Switzerland, October 2002.
21. G. Vigna, W. Robertson, V. Kher, and R.A. 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.
22. D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the* 9^{th} *ACM Conference on Computer and Communications Security*, pages 255–264, Washington DC, USA, November 2002.
23. C. Warrender, S. Forrest, and B.A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.
24. Watchfire. AppShield Web Intrusion Prevention. `http://www.watchfire.com/products/appshield/default.aspx`, 2005.