PROJECT REPORT

ON

DETECTION OF SQL INJECTION ATTACKS USING HIDDEN MARKOV MODELS

Submitted by:

Ajit Kumar Sahoo	-	1201209118
Khushboo Agarwal	-	1201209123

7th Semester CSE (Batch: 2012-16)

GROUP NO: CS28

Under guidance of:

Prof. Debabrata Kar

DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

SILICON INSTITUTE OF TECHNOLOGY

Silicon Hills, Patia, Bhubaneswar-751024

DECLARATION

We hereby declare that the project report entitled **“Detection of SQL Injection attacks using Hidden Markov Models”** submitted in partial fulfillment of the requirements for the degree of Bachelor of Technology in Computer Science and Engineering of Biju Pattnaik University of Technology is our original work and not submitted to any other university or Institute for the award of any degree or diploma.

Ajit Kumar Sahoo	**Khushboo Agarwal**
Regd. No: 1201209118	Regd. No: 1201209123

DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

SILICON INSTITUTE OF TECHNOLOGY

Silicon Hills, Patia, Bhubaneswar-751024

CERTIFICATE

This is to certify that Ajit Kumar Sahoo (Regd. No: 1201209118) and Khushboo Agarwal (Regd. No: 1201209123) have undertaken and successfully completed the project entitled **“Detection of SQL Injection attacks using Hidden Markov Models”** under my supervision.

The report is being submitted as a part of 7th Semester project for the undergraduate curriculum.

Signature of the Guide: ……………………………………..……………..

Name: ………………………………………………………..………………….

Designation: ……………………………………………………………………

DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

SILICON INSTITUTE OF TECHNOLOGY

Silicon Hills, Patia, Bhubaneswar-751024

ACKNOWLEDGMENT

We owe a great many thanks to a great many people who helped and supported us during the project.

Our deepest thanks to Mr. Debabrata Kar, Associate Professor, Department of CSE, the project guide for guiding and correcting various documents of mine with attention and care. He has taken pain to go through the project and make necessary correction as and when needed.

We would also thank my Institution and my faculty members without whom this project would have been a distant reality. We also extend my heartfelt thanks to my family, friends and well-wishers.

Ajit kumar Sahoo

Regd. No: 1201209118

Khushboo Agarwal

Regd. No: 1201209123

DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

SILICON INSTITUTE OF TECHNOLOGY

Silicon Hills, Patia, Bhubaneswar-751024

TABLE OF CONTENTS

ABSTRACT

In this era of internet, we have become completely dependent on web. While the Internet and web applications have made our lives much simpler, security & privacy of the sensitive data in the backend databases has become a big concern. Among various types of security threats to which a web application is exposed to, SQL injection attack is considered to be most prevalent and dangerous. Through SQL injection, an attacker can obtain unrestricted access to the backend databases and extract the potentially sensitive information.

Since its discovery in 1998, many researchers have proposed various methods to address the problem, however, many approaches either fail to address the full scope of the problem or have limitations that prevent their use and adoption in a practical environment.

In this project, we intend to find out a solution to identify SQL injected queries at the database firewall level, i.e., between the web application and the database server. We aim to build a system that can examine run-time queries issued by a web application and allow only benign queries to pass through to the database server. Any SQL query is just a string, containing SQL keywords, values, identifiers, delimiters etc. Therefore, detecting injected queries can be considered as a pattern recognition problem. Hidden Markov Model (HMM) is a technique which has been successfully used in many pattern recognition areas. Based on our preliminary study, a database firewall consisting of HMMs can be trained to identify SQL injected queries at run-time. However, because the system is targeted to work as a database firewall, it will be portable and technology/platform independent. We will be using PHP, Python, MySQL, and MATLAB etc. for this project.

1. INTRODUCTION

In recent years, widespread adoption of the internet has resulted in to rapid advancement in information technologies. The internet is used by the general population for the information, in a way that allows the information owners quick access while blocking break-in attempts from unauthorized users. Every year about 50% of databases experience at least one purposes such as financial transactions, educational endeavors, and countless other activities. The use of the internet for accomplishing important tasks, such as transferring balance from bank accounts, always comes with a security risk. Today's web sites strive to keep their users' data confidential and after years of doing secure business online, these companies have become experts in information security. The database systems behind these secure websites store non-critical data along with sensitive security breach. The loss of revenue associated with such breaches has been estimated to be over four million dollars. Additionally, recent research by the "Imperva Application Defense Centre" concluded that at least 92% of web applications are susceptible to "malicious attack".

Among various types of security threats, web applications are exposed to, SQL Injection attack is predominantly used against web databases. The Open Web Application Security Project (OWASP) ranks it on top among the Top-10 security threats. According to TrustWave 2012 Global Security Report, SQL injection was the number one attack method for four consecutive years. Nowadays, attackers use sophisticated Botnets which automatically discover vulnerable web pages from search engines like Google and launch mass SQL injection attacks from distributed sources.

About 97% of data breaches across the world occur due to SQL injection alone.

Although recently there has been a great deal of attention to the problem of SQL injection vulnerabilities, many proposed solutions fail to address the full scope of the problem. There are many types of SQL Injection Attacks and countless variations on these basic types. Researchers and practitioners are often unaware of the myriad of different techniques that can be used to perform SQL Injection Attacks. Therefore, most of the solutions proposed detect or prevent only a subset of the possible SQL Injection Attacks.

Here we propose a novel method for detection of SQL injection using Hidden Markov Model. Our proposed technique consists of training HMM using a set of queries and then evaluating all runtime queries against the training parameters to check whether the query is genuine or injected.

1.1 PROBLEM

The increasing number of Web-based applications that are deployed worldwide makes their protection a key topic in computer security. Unfortunately, the large number of new attacks that appear every day makes almost impossible to have signature-based systems always updated to the most recent attacks.

In this project, we intend to find out a solution to identify SQL injected queries at the database firewall level, i.e., between the web application and the database server. We aim to build a system that can examine runtime queries issued by a web application for execution and allow only benign queries to pass through to the database server.

1.2 SQL INJECTION

SQL injection is a code injection technique, used to attack data-driven applications, in which malicious SQL statements are inserted into an entry field for execution (e.g. to dump the database contents to the attacker). SQL injection must exploit a security vulnerability in an application's software, for example, when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and unexpectedly executed. It is mostly known as an attack vector for websites but can be used to attack any type of SQL database.

SQL injection vulnerabilities are described as one of the most serious threats for Web applications. The Open Web Application Security Project (OWASP) ranks it on top among the Top-10 security threats. According to TrustWave 2012 Global Security Report, SQL injection was the number one attack method for four consecutive years. Nowadays, attackers use sophisticated Botnets which automatically discover vulnerable web pages from search engines like Google and launch mass SQL injection attacks from distributed sources. About 97% of data breaches occur due to SQL injection alone.

Web applications that are vulnerable to SQL injection may allow an attacker to gain complete access to their underlying databases. Because these databases often contain sensitive consumer or user information, the resulting security violations can include identity theft, loss of confidential information, and fraud. In some cases, attackers can even use an SQL injection vulnerability to take control of and corrupt the system that hosts the Web application.

Malicious SQL statements can be introduced into a vulnerable application using many different input mechanisms. Here, we explain the most common mechanisms.

Injection through user input: In this case, attackers inject SQL commands by providing suitably crafted user input. A Web application can read user input in several ways based on the environment in which the application is deployed. In most SQLIAs that target Web applications, user input typically comes from form submissions that are sent to the Web application via HTTP GET or POST requests.

Injection through cookies: Cookies are files that contain state information generated by Web applications and stored on the client machine. When a client returns to a Web application, cookies can be used to restore the client's state information. Since the client has control over the storage of the cookie, a malicious client could tamper with the cookie's contents. If a Web application uses the cookie's contents to build SQL queries, an attacker could easily submit an attack by embedding it in the cookie.

Injection through server variables: Server variables are a collection of variables that contain HTTP, network headers, and environmental variables. Web applications use these server variables in a variety of ways, such as logging usage statistics and identifying browsing trends. If these variables are logged to a database without sanitization, this could create SQL injection vulnerability. Because attackers can forge the values that are placed in HTTP and network headers, they can exploit this vulnerability by placing an SQLIA directly into the headers. When the query to log the server variable is issued to the database, the attack in the forged header is then triggered.

Second-order injection: In second-order injections, attackers seed malicious inputs into a system or database to indirectly trigger an SQLIA when that input is used at a later time. The objective of this kind of attack differs significantly from a regular injection attack. Second-order injections are not trying to cause the attack to occur when the malicious input initially reaches the database. Instead, attackers rely on knowledge of where the input will be subsequently used and craft their attack so that it occurs during that usage.

Each of the attack type includes a list of one or more of the attack intents for example:

Identifying injectable parameters: The attacker wants to probe a Web application to discover which parameters and user-input fields are vulnerable to SQLIA.

Performing database finger-printing: The attacker wants to discover the type and version of database that a Web application is using. Certain types of databases respond differently to different queries and attacks, and this information can be used to “fingerprint” the database. Knowing the type and version of the database used by a Web application allows an attacker to craft database- specific attacks.

Determining database schema: To correctly extract data from a database, the attacker often needs to know database schema information, such as table names, column names, and column data types. Attacks with this intent are created to collect or infer this kind of information.

Extracting data: These types of attacks employ techniques that will extract data values from the database. Depending on the type of the Web application, this information could be sensitive and highly desirable to the attacker. Attacks with this intent are the most common type of SQLIA.

Adding or modifying data: The goal of these attacks is to add or change information in a database.

Performing denial of service: These attacks are performed to shut down the database of a Web application, thus denying service to other users. Attacks involving locking or dropping database tables also fall under this category.

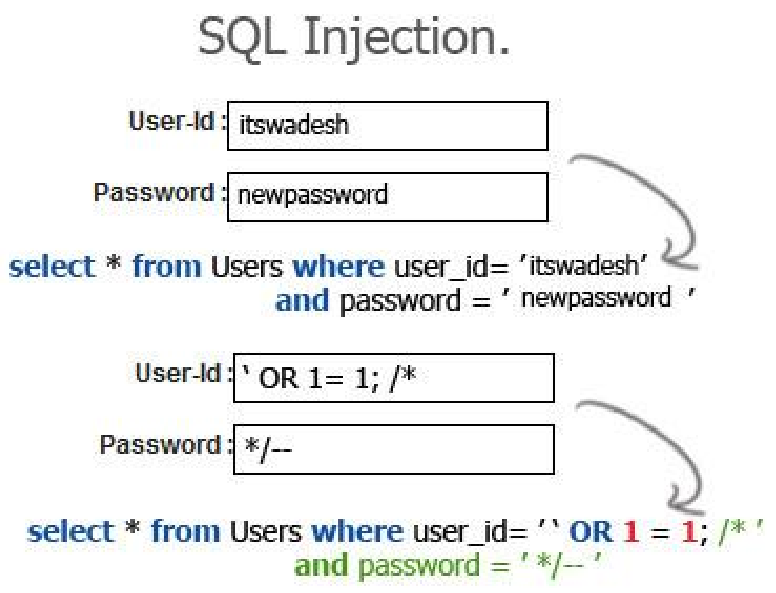
Evading detection: This category refers to certain attack techniques that are employed to avoid auditing and detection by system protection mechanisms.

Bypassing authentication: The goal of these types of attacks is to allow the attacker to bypass database and application authentication mechanisms. Bypassing such mechanisms could allow the attacker to assume the rights and privileges associated with another application user.

Executing remote commands: These types of attacks attempt to execute arbitrary commands on the database. These commands can be stored procedures or functions available to database users.

Performing privilege escalation: These attacks take advantage of implementation errors or logical flaws in the database in order to escalate the privileges of the attacker. As opposed to bypassing authentication attacks, these attacks focus on exploiting the database user privileges.

SQL injection can be used using various methods. Here we will show by an example how a simple SQL injection Attack can be launched. Suppose you are on a shopping site. It uses the following interface for log in purpose.



```
select * from Users where user_id= 'itswadesh'
        and password = ' newpassword '
```

```
select * from Users where user_id= '' OR 1 = 1; /* '
        and password = ' */-- '
```

Fig. 1.1 Example Login Form and Injection Attempt

If instead of first input (Fig. 1), we give the input as second one, the query will bypass the authentication process and you will just login without valid username and password to a website and the website is said to be SQL injected.

1.3 SQL INJECTION ATTACKS

There have been numerous attacks using SQL injection that has caused major loss to humankind. Some of these are as follows:

French smartphone maker Archos was compromised with a SQL injection attack by a hacker group ‘Focus’ on last Christmas, 2014, resulting in the leak of up to 100,000 customer details. But fortunately, passwords and credit card details were not stolen.

On October, 2013, a group of hackers, known as TeamBerserk, took credit on Twitter – posting as @TeamBerserk – for using a SQL injection attack to access usernames and passwords for customers of Sebastian, a California-based internet, phone and television service provider, and then leveraging those credentials to steal $100,000 from online accounts.

Just a couple of years back, i.e. March 27th 2011 to be precise, Romanian Hackers by the name of “Tin Kode” and “Ne0h” attacked MySQL.com and Sun.com. They did this with a SQL injection attack, to gather table names, column names and email addresses stored in one of the tables.

1.4 TYPES OF SQL INJECTION ATTACKS

The different types of attacks are generally not performed in isolation; many of them are used together or sequentially, depending on the specific goals of the attacker. Note also that there are countless variations of each attack type. We do not present all of the possible attack variations but instead present a single representative example.

1.4.1 TAUTOLOGIES

Attack Intent: Bypassing authentication, identifying injectable parameters, extracting data.

Description: The general goal of a tautology-based attack is to inject code in one or more conditional statements so that they always evaluate to true. The consequences of this attack depend on how the results of the query are used within the application. The most common usages are to bypass authentication pages and extract data. In this type of injection, an attacker exploits an injectable field that is used in a query's WHERE conditional. Transforming the conditional into a tautology causes all of the rows in the database table targeted by the query to be returned. In general, for a tautology-based attack to work, an attacker must consider not only the injectable/vulnerable parameters, but also the coding constructs that evaluate the query results. Typically, the attack is successful when the code either displays all of the returned records or performs some action if at least one record is returned.

Example: In this example attack, an attacker submits *"′ or 1=1 - -″* for the login input field. The resulting query is:

```
SELECT accounts FROM users WHERE login=′′ or 1=1 --
AND pass=′′ AND pin=′′
```

Example: The code injected in the conditional (OR 1=1) transforms the entire WHERE clause into a tautology. Because the conditional is a tautology, the query evaluates to true for each row in the table and returns all of them.

1.4.2 ILLEGAL/LOGICALLY INCORRECT QUERIES

Attack Intent: Identifying injectable parameters, performing database finger-printing, extracting data.

Description: This attack lets an attacker gather important information about the type and structure of the back-end database of a Web application. The attack is considered a preliminary, information- gathering step for other attacks. The vulnerability leveraged by this attack is that the default error page returned by application servers is often overly descriptive. In fact, the simple fact that an error messages is generated can often reveal vulnerable/injectable parameters to an attacker. Additional error information, originally intended to help programmers debug their applications, further helps attackers gain information about the schema of the back-end database. When performing this attack, an attacker tries to inject statements that cause a syntax, type conversion, or logical error into the database. Syntax errors can be used to identify injectable parameters. Type errors can be used to deduce the data types of certain columns or to extract data. Logical errors often reveal the names of the tables and columns that caused the error.

Example: Here the attacker injects the following text into input field pin: *“convert(int,(select top 1 name from sysobjects where xtype=’u’))”.*

The resulting query is:

```
SELECT accounts FROM users WHERE login=’’ AND pass=’’
AND pin= convert (int,(select top 1 name from
sysobjects where xtype=’u’))
```

In the attack string, the injected select query attempts to extract the first user table (xtype='u') from the database's metadata table. The query then tries to convert this table name into an integer. Because this is not a legal type conversion, the database throws an error. For Microsoft SQL Server, the error would be:

```
"Microsoft OLE DB Provider for SQL Server (0x80040E07)
Error converting nvarchar value 'CreditCards' to a
column of data type int."
```

There are two useful pieces of information in this message that aid an attacker. First, the attacker can see that the database is an SQL Server database. Second, the error message reveals the value of the string (table name i.e. CreditCards) that caused the type conversion to occur.

1.4.3 UNION QUERY

Attack Intent: Bypassing Authentication, extracting data.

Description: In union-query attacks, an attacker exploits a vulnerable parameter to change the data set returned for a given query. With this technique, an attacker can trick the application into returning data from a table different from the one that was intended by the developer. Attackers do this by injecting a statement of the form: UNION SELECT <rest of injected query>. Because the attackers completely control the second/injected query, they can use that query to retrieve information from a specified table. The result of this attack is that the database returns a dataset that is the union of the results of the original first query and the results of the injected second query.

Example: Here an attacker could inject the text "' *UNION SELECT cardNo from CreditCards where acctNo=10032 - -"* into the login field, which produces the following query:

```
SELECT accounts FROM users WHERE login=’’ UNION SELECT
cardNo from CreditCards where acctNo=10032 -- AND
pass=’’ AND pin=’’
```

Assuming that there is no login equal to "", the original first query returns the null set, whereas the second query returns data from the "CreditCards" table. In this case, the database would return column "cardNo" for account "10032."

1.4.4 PIGGY-BACKED QUERIES

Attack intent: Extracting data, adding or modifying data, performing denial of service, executing remote commands.

Description: In this attack type, an attacker tries to inject additional queries into the original query. We distinguish this type from others because, in this case, attackers are not trying to modify the original intended query; instead, they are trying to include new and distinct queries that "piggy-back" on the original query. As a result, the database receives multiple SQL queries. The first is the intended query which is executed as normal; the subsequent ones are the injected queries, which are executed in addition to the first. This type of attack can be extremely harmful. If successful, attackers can insert virtually any type of SQL command, including stored procedures into the additional queries and have them executed along with the original query. Vulnerability to this type of attack is often dependent on having a database configuration that allows multiple statements to be contained in a single string.

Example: If the attacker inputs "*'; drop table users --*" into the pass field, the application generates the query:

```
SELECT accounts FROM users WHERE login='doe' AND
pass=''; drop table users --' AND pin=123
```

After completing the first query, the database would recognize the query delimiter (";") and execute the injected second query. The result of executing the second query would be to drop table users, which would likely destroy valuable information.

1.4.5 STORED PROCEDURES

Attack Intent: Performing privilege escalation, performing denial of service, executing remote commands.

Description: SQLIAs of this type try to execute stored procedures present in the database. Today, most database vendors ship databases with a standard set of stored procedures that extend the functionality of the database and allow for interaction with the operating system. Therefore, once an attacker determines which backend- database is in use, SQLIAs can be crafted to execute stored procedures provided by that specific database, including procedures that interact with the operating system. It is a common misconception that using stored procedures to write Web applications renders them invulnerable to SQLIAs. Developers are often surprised to find that their stored procedures can be just as vulnerable to attacks as their normal applications. Additionally, because stored procedures are often written in special scripting languages, they can contain other types of vulnerabilities, such as buffer overflows, that allow attackers to run arbitrary code on the server or escalate their privileges.

Example: To launch an SQLIA, the attacker simply injects "*′; SHUTDOWN;* --" into either the userName or password fields. This injection causes the stored procedure to generate the following query:

```
SELECT accounts FROM users WHERE login=′doe′ AND
pass=′ ′; SHUTDOWN; -- AND pin=′′
```

At this point, this attack works like a piggy-back attack. The first query is executed normally, and then the second, malicious query is executed, which results in a database shut down.

1.4.6 INFERENCE

Attack intent: Identifying injectable parameters, extracting data, determining database schema.

Description: In this attack, the query is modified to recast it in the form of an action that is executed based on the answer to a true/false question about data values in the database. In this type of injection, attackers are generally trying to attack a site that has been secured enough so that, when an injection has succeeded, there is no usable feedback via database error messages. Since database error messages are unavailable to provide the attacker with feedback, attackers must use a different method of obtaining a response from the database. In this situation, the attacker injects commands into the site and then observes how the function/response of the website changes. By carefully noting when the site behaves the same and when its behavior changes, the attacker can deduce not only whether certain parameters are vulnerable, but also additional information about the values in the database.

Example: In this attack, the following is injected into the login parameter:

“*legalUser’ and ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) > X WAITFOR 5 --*”.

This produces the following query:

```
SELECT accounts FROM users WHERE login=’legalUser’ and ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) > X WAITFOR 5 -- ’ AND pass=’’ AND pin=0
```

In this case, the attacker is asking if the ASCII value of the character is greater-than or less-than-or-equal-to the value of X. If the value is greater, the attacker knows this by observing an additional 5 second delay in the response of the database. The attacker can then use a binary search by varying the value of X to identify the value of the first character.

There are two well- known attack techniques that are based on inference. They allow an attacker to extract data from a database and detect vulnerable parameters. Researchers have reported that with these techniques they have been able to achieve a data extraction rate of 1B/s.

Blind Injection: In this technique, the information must be inferred from the behavior of the page by asking the server true/false questions. If the injected statement evaluates to true, the site continues to function normally. If the statement evaluates to false, although there is no descriptive error message, the page differs significantly from the normally-functioning page.

Timing Attacks: A timing attack allows an attacker to gain information from a database by observing timing delays in the response of the database. This attack is very similar to blind injection, but uses a different method of inference. To perform a timing attack, attackers structure their injected query in the form of an if/then statement, whose branch predicate corresponds to an unknown about the contents of the database. Along one of the branches, the attacker uses a SQL construct that takes a known amount of time to execute, (e.g. the WAITFOR keyword, which causes the database to delay its response by a specified time). By measuring the increase or decrease in response time of the database, the attacker can infer which branch was taken in his injection and therefore the answer to the injected question.

1.4.7 ALTERNATE ENCODINGS

Attack Intent: Evading detection

Description: In this attack, the injected text is modified so as to avoid detection by defensive coding practices and also many automated prevention techniques. This attack type is used in conjunction with other attacks. In other words, alternate encodings do not provide any unique way to attack an application; they are simply an enabling technique that allows attackers to evade detection and prevention techniques and exploit vulnerabilities that might not otherwise be exploitable. These evasion techniques are often necessary because a common defensive coding practice is to scan for certain known “bad characters”. To evade this defense, attackers have employed alternate methods of encoding their attack strings (e.g., using hexadecimal, ASCII, and Unicode character encoding). Common scanning and detection techniques do not try to evaluate all specially encoded strings, thus allowing these attacks to go undetected. Contributing to the problem is that different layers in an application have different ways of handling alternate encodings. The application may scan for certain types of escape characters that represent alternate encodings in its language domain. Another layer (e.g., the database) may use different escape characters or even completely different ways of encoding.

An effective code-based defense against alternate encodings is difficult to implement because it requires developers to consider of all of the possible encodings that could affect a given query string as it passes through the different application layers. Therefore, attackers have been very successful in using alternate encodings to conceal their attack strings.

Example In this attack, the following text is injected into the login field: “*legalUser’; exec(0x73687574646f776e) --* ”. The resulting query generated by the application is:

```
SELECT accounts FROM users WHERE login=’legalUser’;
exec(char(0x73687574646f776e)) -- AND pass=’’ AND pin=
```

The stream of numbers here is the ASCII hexadecimal encoding of the string “SHUTDOWN.” Therefore, when the query is interpreted by the database, it would result in the execution, by the database, of the SHUTDOWN command.

2. RELATED WORK

Researchers have proposed a wide range of techniques to address the problem of SQL injection. These techniques range from development best practices to fully automated frameworks for detecting and preventing SQL Injection Attacks. In this section, we review these proposed techniques and summarize the advantages and disadvantages associated with each technique.

Various techniques have been proposed for the confrontation of the threat of SQL injection attacks. In this section, the characteristics of the best known techniques are briefly discussed and their principal weaknesses are highlighted.

2.1. STATIC ANALYSIS

Wassermann and Su propose a static analysis framework to filter user inputs. According to them, their approach has some limitations concerning implementation-related issues, such as the way it handles some operators. Additionally, this approach is limited to discover only tautology-based attacks, i.e. attacks that always result in true or false SQL statements.

Another static analysis approach has been proposed by Livshits and Lam. In this work, vulnerability patterns are described in a program query language called PQL. Static analysis is applied to find potential violations matching a vulnerability pattern. The main limitation of the method is that it cannot detect SQL injection attacks patterns that are not known beforehand, and explicitly described in the specifications.

In Huang et al., preconditions are specified for all sensitive PHP functions and user input is checked against these preconditions. Like all static analysis approaches, his technique does not provide an automated mechanism for detection and prevention of SQL injection attacks.

This is the most important reason why all the approaches of this subsection generate a significant number of false negatives.

2.2. DYNAMIC ANALYSIS

AMNESIA uses a model-based approach to detect illegal queries before their execution into the database. In its static part, the technique uses program analysis to automatically build a model of the legitimate queries that could be generated by the application. In its dynamic part, the technique uses runtime monitoring to inspect the dynamically-generated queries and check them against the statically-built model. A primary assumption regarding the applications which the method targets is that the application developer creates queries by combining hard- coded strings and variables using operations such as concatenation, appending and insertion.

The main drawback of AMNESIA is that it requires the modification of the web application’s source code for the successful collaboration with the security monitor-officer. Also it involves a number of steps using different tools.

SQLGuard is based on comparing, at run time, the parse tree of the SQL statement before the inclusion of the user input with that resulting from parse tree of the SQL statement after the inclusion of the user input. A secret key is used for wrapping the user input, so if an attacker compromises this key, SQLGuard is difficult to prevent an attack. The overhead to database query costs is about 3msec (the characteristics of their server were very similar to ours: a 733MHz Windows 2000 Server machine with 256MB RAM). Another drawback of their method is that, similarly to AMNESIA, it requires the modification of the application's scripts.

SQLCheck is a similar approach. It adds a key at the beginning and at the end of each user's input. At runtime, the "augmented" queries that are not in a valid syntactic form are considered attacks. The detection ability of the approach depends on the strength of the key also.

In **SQLRand**, the SQL standard keywords are manipulated by appending a random integer to them that an attacker cannot easily guess. Therefore, any malicious user attempting an SQL injection attack would be thwarted. Randomized instances of SQL queries are created by randomizing the template query inside the CGI script and the database parser. To allow for easy retrofitting of our solution to existing systems, we introduce a de-randomizing proxy, which converts randomized queries to proper SQL queries for the database. Code injected by the rogue client evaluates to undefined keywords and expressions. When this is the outcome, then standard keywords (e.g., "or") lose their significance, and attacks are frustrated before they can even commence. The performance overhead of our approach is minimal, adding up to 6.5ms to query processing time.

This technique uses a key to randomize SQL queries, so if an attacker compromises this key, SQLRand is difficult to prevent an attack. Moreover, the approach imposes a significant overhead in terms of infrastructure because it requires the integration of a special proxy in the web-application infrastructure.

Table. 2 shows a basic comparison among different schemes discussed above and their capability to handle different type of SQL injection.

Scheme	**Tautology**	**Logically Incorrect Queries**	**Union Queries**	**Stored Procedure**	**Piggy-backed Queries**	**Inference**	**Alternate Encoding**
AMNESIA	Yes	Yes	Yes	No	Yes	Yes	Yes
SQLGuard	Yes	Yes	Yes	No	Yes	Yes	Yes
SQLCheck	Yes	Yes	Yes	No	Yes	Yes	Yes
SQLRand	Yes	No	Yes	No	Yes	Yes	No

Table 2.1. Various Injection Schemes and SQL Injection Attacks

Here we can easily infer from the above table that none of the technique is able to handle all 7 kinds of queries. Additionally, all the methods were proved inefficient while dealing with Stored Procedure SQL attacks.

3. PROPOSED APPROACH

Two key ideas, the query transformation scheme and application of Hidden Markov Model to train our model constitute the core of our approach. The system begins with an initial set of SQL injection attacks collected from different web applications using some automated tools or manually. These are converted into text form using a transformation scheme and then grouped into clusters by their document similarity. Attack vectors in each cluster are merged into a document. Each document thus contains a set of highly similar attack vectors. Then we train different Hidden Markov Models using different clusters of collected queries. At run-time, SQL injection attacks are detected by evaluating the incoming query against the different models. Rest of this section describes our approach and architecture of our model in detail. Rest of this section presents explanation of the concepts.

3.1. HIDDEN MARKOV MODEL

A **Hidden Markov Model** (**HMM**) is a statistical Markov model in which the system being modeled is assumed to be a Markov process with unobserved (*hidden*) states. The mathematics behind the HMM was developed by L. E. Baum and coworkers.

In simple Markov models, the state is visible to the observer, and therefore the state transition probabilities are the only parameters. In a *hidden* Markov model, the state is not directly visible, but output, dependent on the state, is visible. Each state has a probability distribution over the possible output tokens. Therefore the sequence of tokens generated by an HMM gives some information about the sequence of states.

Hidden Markov models are especially known for their application in temporal pattern recognition such as speech, handwriting, gesture recognition, part-of-speech tagging, musical score following, partial discharges and bioinformatics.

A HMM Model is specified by:

S - The set of states (n is the number of states)

$$S = \{s_1, s_2, \dots, s_n\},$$

π - The prior probabilities as the probabilities of s_i being the first state of a state sequence.

$$\pi = P(Q = s_i)$$

A - The transition probabilities as the probabilities to go from state i to state j.

$$A_{ij} = P(Q_{n+1} = s_j | Q_j = s_i)$$

B - The emission probabilities characterize the likelihood of a certain observation $x_n \in \{v_1,\dots,v_k\}$:, if the model is in state s_i.

$$B_{ik} = P(x_n = v_k | Q_n = s_i)$$

Mathematically, HMM can be represented as

$$\boxed{\lambda = (A,B,\pi)}$$

Where,

A = state transition probabilities
B = observation probability matrix
π = initial state distribution

The matrices π, A and B are row stochastic, meaning that each element is a probability and the elements of each row sum to 1, that is, each row is a probability distribution.

A HMM allowing for transitions from any emitting state to any other emitting state is called an ergodic HMM. The other extreme, a HMM where the transitions only go from one state to itself or to a unique follower is called a left-right HMM.

There are three fundamental problems that we can solve using HMMs. Here, we briefly describe these three problems.

3.1.1. PROBLEM 1: EVALUATION

Given the model $\lambda = (A,B,\pi)$ and a sequence of observations O, find $P(O|\lambda)$. Here, we want to determine the likelihood of the observed sequence O, given the model.

3.1.2. PROBLEM 2: DECODING

Given $\lambda = (A,B,\pi)$ and an observation sequence O, find an optimal state sequence for the underlying Markov process. In other words, we want to uncover the hidden part of the Hidden Markov Model.

3.1.3. PROBLEM 3: TRAINING

Given an observation sequence O and the dimensions N and M, find the model λ = (A,B,π) that maximizes the probability of O. This can be viewed as training a model to best fit the observed data.

As we want to detect SQL injection attacks, we wish to approach the problem by first training our HMM by a collected set of both genuine and injected queries (Problem 3) in offline mode, then checking a dynamic query at runtime by evaluating against the trained hmm parameters (Problem 1).

SQL queries are simple tokens of characters so our problem of SQL injection can also be seen as a pattern recognition problem. Seeing the effectiveness of HMM in pattern recognition problem, we expect HMM to be equally effective in our problem of injected query detection.

3.2. QUERY EXTRACTION

Our requirement is a large data set of queries of both injected and genuine type to train and test our model. For the purpose, we developed a web application '*Quiz Portal*' and also downloaded several open source web applications built on PHP and MySQL. We run those applications in our local server and extracted the genuine queries from the log file easily. The web applications were also subjected to SQL injection attacks using a mixture of automated scanning and hacking tools such as HP Scrawler (free version), NetSparker (community edition), WebCruiser Pro, SQL Power Injector, sqlmap, The Mole, IronWasp, and jSQL Injector etc. to collect different types of injected queries. We also collected different types of injected queries from various sources in the internet.

After the end of this process, we had 2, 53,556 injected queries and genuine queries. Now the problem in front of us was dealing with such a large number of queries. We discovered that most of the queries differ by just some aliases or variable names. So we decided to implement a query transformation scheme.

3.3. QUERY TRANSFORMATION SCHEME

The transformation scheme normalizes an SQL query into a sentence-like form, and facilitates application of similarity and hmm training. The extended scheme uses only capital A-Z for all tokens and space character as token separator. All symbols and special characters are also transformed into words except the underscore (_) character, because it is frequently used in MySQL system databases, system tables and functions, e.g., information_schema, table_priv, CURRENT_USER(), LAST_INSERT_ID(), etc. Splitting such tokens at the underscore character would result in over tokenization and negatively aﬀect the similarity values.

Token	Transform	Token	Transform
Newline Chars (\r, \n)	Remove	Quoted String	STR
Inline Comments (/*…*/)	Remove	Integer Values	INT
System Database	SYSDB	Decimal Values	DEC
System Table	SYSTBL	Hexadecimal Values	HEX

Column of System Table	SYSCOL	Single Alphabet	CHR
System Variables	SYSVAR	IP Address	IPADDR
User Database	USRDB	All Other Symbols	As per Table-2
User Table	USRTBL	Multiple Spaces	Single Space
Column of User Table	USRCOL	Entire Query	To Upper Case

Table 3.1: The Query Transformation Scheme

Symbol	**Transform**	**Symbol**	**Transform**	**Symbol**	**Transform**
‘	Remove	^	XOR	{	LCBR
!= OR <>	NEQ	&	BITAND	}	RCBR
&&	AND	|	BITOR	\	BSLSH
||	OR	*	STAR	:	CLN
~	TLDE	-	MINUS	;	SMCLN
!	EXCLM	+	PLUS	,	CMMA
#	HASH	=	EQ	.	DOT
$	DLLR	(	LPRN	?	QSTN
%	PRCNT	)	RPRN	/	SLSH

Table 3.2: The Query Transformation Scheme for Special Symbols

Each step of query transformation is a find-and-replace operation, done by appropriately using the `preg_replace()` and `str_replace()` built-in functions available in PHP.

For example, **`preg_replace("/\b0x[0-9a-f]+\b/i", "HEX", $query)`** converts hexadecimal numbers in $query into HEX in a case-insensitive manner.

All symbols and special characters are transformed in as per the scheme given in the table. The last two steps convert the transformed query to uppercase and reduce multi-spaces to single spaces. To visualize how the transformation scheme works, consider the following SQL queries, intentionally written in mixed-case to show the effect of transformation.

`sEleCt * fRoM products wHeRe price > 10.00 aNd discount < 8 SeLeCt email FrOm customers WhErE fname LIKE 'john%'`

By applying the transformation scheme, the above queries are transformed into the following form respectively:

`SELECT STAR FROM USRTBL WHERE USRCOL GT DEC AND USRCOL LT INT SELECT USRCOL FROM USRTBL WHERE USRCOL LIKE SQUT STR PRCNT SQUT`

Consider an injected query generated due to a cleverly crafted attack to bypass detection:

`SELECT * FROM products WHERE prod_id = 24 OR 'ABC' = CoNcAt(cHaR(0x28 + 25), cHaR(0x42), cHAr(80 - 0x0d))`

This query contains a number of symbols and operators, which is not suitable for application of a document similarity measure. The transformation scheme converts it completely into text form as:

SELECT STAR FROM USRTBL WHERE USRCOL EQ INT OR SQUT STR SQUT EQ CONCAT LPRN CHAR LPRN HEX PLUS INT RPRN CMMA CHAR LPRN HEX RPRN CMMA CHAR LPRN INT MINUS HEX RPRN RPRN

Any SQL query, irrespective of its complexity, is thus transformed into a series of words separated by spaces like a sentence in English. The structural form of the query is correctly maintained by the transformation scheme.

3.4. DETECTION STRATEGY

Design of SQL Injection Attacks is strategically guided by two interesting observations which we were unable to find mentioned anywhere in the literature. The first observation surfaced while looking for an answer to "where do SQL injections most commonly occur in an injected query?" Looking at the general coding practices for developing web applications, we find that, dynamic SQL queries are usually constructed in two parts:

(1) A static part hard coded by the programmer, and

(2) A dynamic part produced by concatenation of SQL keywords, delimiters and received input values.

Since the main objective of a dynamic query is to fetch different set of records depending on the input, the parameter values must be used in the WHERE clause to specify the selection criteria. This compulsive programming need as well as the de facto practice of using input values in the WHERE clause part of dynamic queries resolves that SQL injections happen after the WHERE keyword, almost in all cases.

In fact, SQL injection before the WHERE keyword is extremely rare – not a single instance was found in over 16,500 queries containing SQL injection attacks we examined for this study. The second interesting observation stems from the basic intention behind an SQL injection attack stealing sensitive information from the back-end database, such as credit card numbers. By carefully formulating the injection code, the attacker tries to get the intended data displayed on the web page itself, which is the only method for data breach through SQL injection. Data is fetched by SELECT queries in SQL, which implies that unless the injectable parameter is used to construct a dynamic SELECT query that delivers data displayed on the web page, it is not useful for data breach.

It points to another assertion that SQL injections mostly occur through dynamically generated SELECT queries. The above observations lead to the strategic inference that, it is sufficient to examine the part of an SQL query after the WHERE keyword in order to detect the presence of any injection attack, which significantly narrows down the scope of processing.

Another plausible strategy may be to intercept only SELECT queries for examination; however, this may enable the attacker to cause damage to the data, if not breach. As such, by considering the part of a query after the WHERE keyword, we automatically include UPDATE and DELETE queries in the investigation, because they also support a WHERE clause by SQL syntax.

A cognizant question may be raised concerning a special kind of attack, known as Second Order SQL Injection, which is initiated through INSERT queries. In this method, the attacker submits values containing injected code through form fields (e.g., a registration form) which is not executed at that time, but gets stored in the database as normal data. It takes eﬀect when another part of the web application uses that stored value in a dynamic query.

It is interesting to note that:

1) The victim dynamic query must be a SELECT query fetching data for display on the web page, and

2) The stored value containing injected code must be used in its WHERE clause.

By detecting injection in the WHERE clause, the secondary attack would render ineﬀective. Therefore, the strategy to examine only the portion of queries after the WHERE keyword is correct and sufficient for detecting SQL injection attacks, including second order injections

After this query transformation scheme, we were left with 4593 injected queries and 492 genuine queries (Massive reduction). But training one hmm for all these queries was an inefficient idea as these thousands of queries differed by a large extent (It's like giving a person thousands of different varieties of food and asking him the taste of the 31st food he was served to).

For effective results, we needed to train different HMMs for different types of queries. For our training purpose we decided to train 7 different HMMs for each injected and genuine queries (one for each of the 7 different types of SQL attacks). So, to cluster our query set into 7 different groups, we employed the K-Medoid clustering algorithm using Cosine Dissimilarity.

3.5. COSINE SIMILARITY MEASURE

In a wide sense, a similarity index measures the degree to which a pair of objects is alike. It is the just the complement of distance measure.

Let $s_1 = (x_1^1\ x_2^1\ \ldots\ x_{L1}^1)$ and $s_2 = (x_1^2\ x_2^2\ \ldots\ x_{L2}^2)$ be strings defined over the alphabet Σ, and let $|s_i| = L_i$ denote the string's length. A similarity index, $S(s_1,s_2)$, measures the degree to which a pair of objects are alike; the larger its value, the closer or more alike we think the patterns are.

Conversely, dissimilarity coefficients, $d(s_1,s_2)$, assess the degree to which patterns differ, smaller values meaning closer or higher resemblance. Distances, differences, reciprocal of similarities, all constitute examples of dissimilarity measures. Given the inverse relationship between similarity and dissimilarity, a simple way to transform similarity into dissimilarity, in the situation of bounded ranges, is:

$$d(s_1, s_2) = S_{\max} - S(s_1, s_2)$$

There are different similarity measures we can consider for our problem. We have considered cosine similarity suitable for our problem. It quantifies correlation between vectors A and B as cosine of the angle between them in n-dimensional space. In this approach each query is treated as a vector having the dimension as 685 as we have 685 different tokens in the query transformation model. Now if A and B are two different vectors then we can compute the cosine similarity is expressed as follows:

Cosine Similarity = $|A.B| / |A|.|B|$

In this manner we can calculate the similarity between all the queries (vectors) and also the dissimilarity can be calculated as follows

Cosine Dissimilarity = 1 – Cosine Similarity

For example:

```
A = ‘CONCAT USRCOL CMMA USRCOL CMMA USRCOL CMMA USRCOL
LIKE SQUT PRCNT INT’;

B = ‘LENGTH USRCOL BETWEEN INT AND USRCOL’;

A                             )
(                             B
    [CONCAT] => 1             (
    [USRCOL] => 4                 [LENGTH] => 1
    [CMMA] => 3                   [USRCOL] => 2
    [LIKE] => 1                   [BETWEEN] => 1
    [SQUT] => 1                   [INT] => 1
    [PRCNT] => 1                  [AND] => 1
    [INT] => 1                )
```

So here |A| = 5.4772, |B| = 2.8284, |A.B| = 9.0000
So cosine Similarity between A and B is = 9.0000 / (5.4772 * 2.8284) = 0.5809
And the cosine Dissimilarity between same A and B is = 1.0000 – 0.5809 = 0.4191

3.6. CLUSTERING USING K-MEDOID

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters). It is a main task of exploratory data mining, and a common technique for statistical data analysis, used in many fields, including machine learning, pattern recognition m image analysis, information retrieval and bioinformatics. The main advantage of clustering is that interesting patterns and structures can be found directly form very large datasets with little or none of the background knowledge.

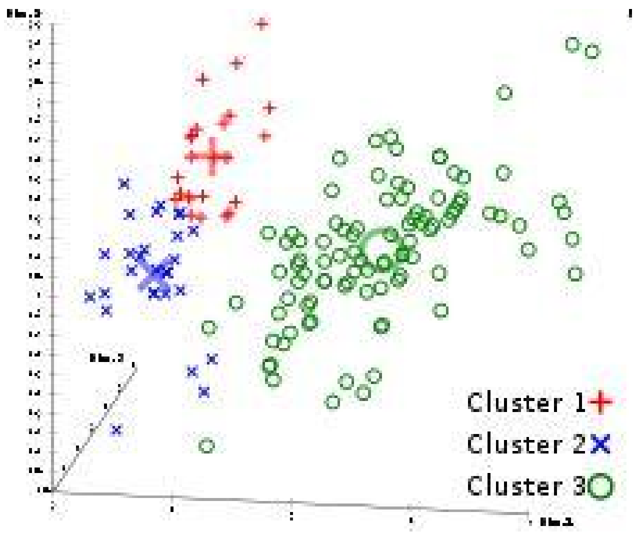


FIG 3.1: Cluster Formed By K-Medoids for K = 3

Mathematically, consider a data set D consisting of N sequences of symbols (or strings), D = {s_1,...,s_n}, defined over the alphabet Σ, where s_i =(x_1^i x_2^i ... $x_{L_i}^i$) is a sequence of length L_i. The problem of sequence clustering is defined as follows:

Given the sample patterns in D, discover from the data a “natural” grouping of the sequences into K clusters, K being in general unknown.

We can group the similar queries so that the number of HMMs required to train will be less. As we have the dissimilarity matrix in our hand so we can go for the K-Medoid clustering algorithm.

The K-means algorithm is sensitive to outliers since an object with an extremely large value may substantially distort the distribution of data.

Instead of taking the mean value of the objects in a cluster as a reference point, a Medoid can be used, which is the most centrally located object in a cluster. Thus the partitioning method can still be performed based on the principle of minimizing the sum of the dissimilarities between each object and its corresponding reference point. This forms the basis of the K-Medoids method.

The basic strategy of K- Mediods clustering algorithms is to find k clusters in n objects by first arbitrarily finding a representative object (the Medoids) for each cluster. Each remaining object is clustered with the Medoid to which it is the most similar. K-Medoids method uses representative objects as reference points instead of taking the mean value of the objects in each cluster.

3.6.1. ALGORITHM

The algorithm takes the input parameter k, the number of clusters to be partitioned among a set of n objects. A typical K-Mediods algorithm for partitioning based on Medoid or central objects is as follows:

Input:

K: The number of clusters

D: A data set containing n objects

Output: A set of k clusters that minimizes the sum of the dissimilarities of all the objects to their nearest medoid.

Method: Arbitrarily choose k objects in D as the initial representative objects;

Repeat:

Assign each remaining object to the cluster with the nearest medoid;

Randomly select a non medoid object O_{random} ;

Compute the total points S of swap point O j with O_{random}

if $S < 0$ then swap O_j with O_{random} to form the new set of k-medoid Until no change;

Here the value of k we decided as 7 as there are around 7 major different types of sql injection attacks. So assuming different pattern queries for each different types of attacks we have chosen the value of k as 7.

3.6.2. ADVANTAGES OF K-MEDOID:

•Simple to understand and implement.

•Fast and convergent in a finite number of steps.

•Usually less sensitive to outliers than k-means.

•Allows using general dissimilarities of objects.

3.6.3. DISADVANTAGES OF K-MEDOID:

•Different initial sets of medoids can lead to different final clustering. It is thus advisable to run the procedure several times with different initial sets of medoids.

•The resulting clustering depends on the units of measurement. If the variables are of different nature or are very different with respect to their magnitude, then it is advisable to standardize them.

3.7. HIDDEN MARKOV MODEL ENSEMBLE

Now we have 492 different genuine queries and 4953 different injected queries in normalized form which are divided into 7 clusters each. So altogether we have 14 sets of queries. So we require 14 HMMs to recognize each cluster among these 14 clusters.

Training: In training phase 90% queries of each cluster is used for the training of each HMM (rest 10% is kept for training purpose) and a value of Hidden Markov model parameters λ = (A, B, π) are obtained.

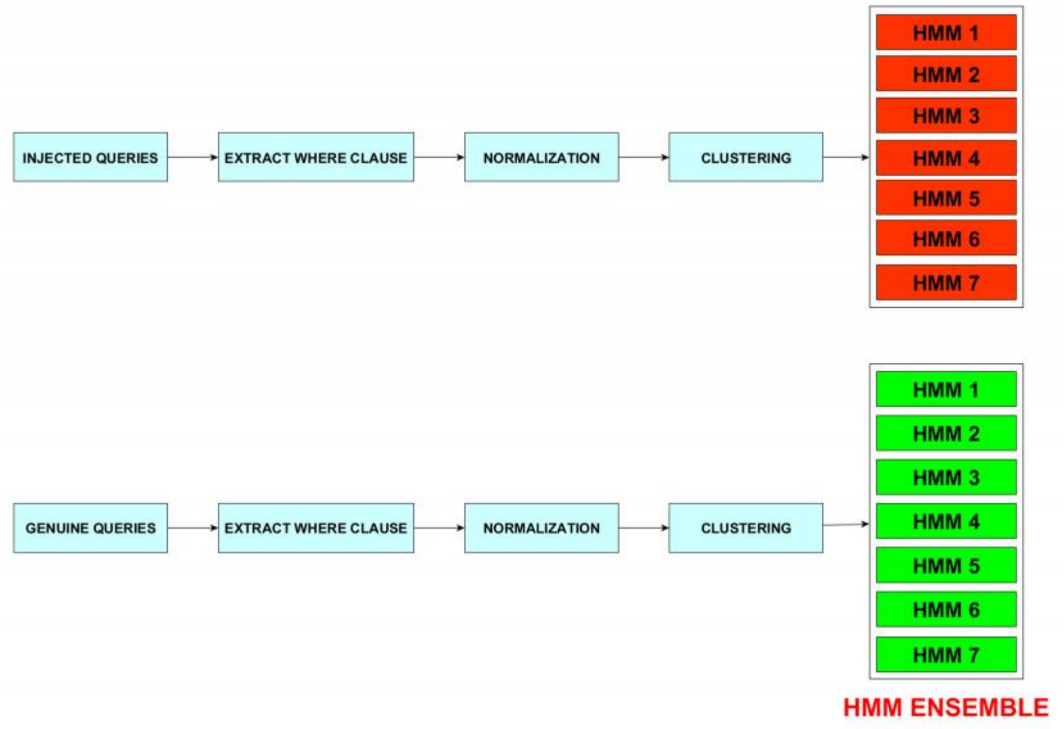


Fig 3.2: Static Part of the Model Used In Hmm Training

This process is done offline and the hmm parameters thus obtained are used for runtime detection of queries at database firewall level.

Runtime Detection: When a query is received at runtime, after normalization, it is passed through each of the HMMs and evaluated against the calculated set of hmm parameters and then the likelihood is checked. The hmm which gives the maximum likelihood decides whether the query is generated or injected.

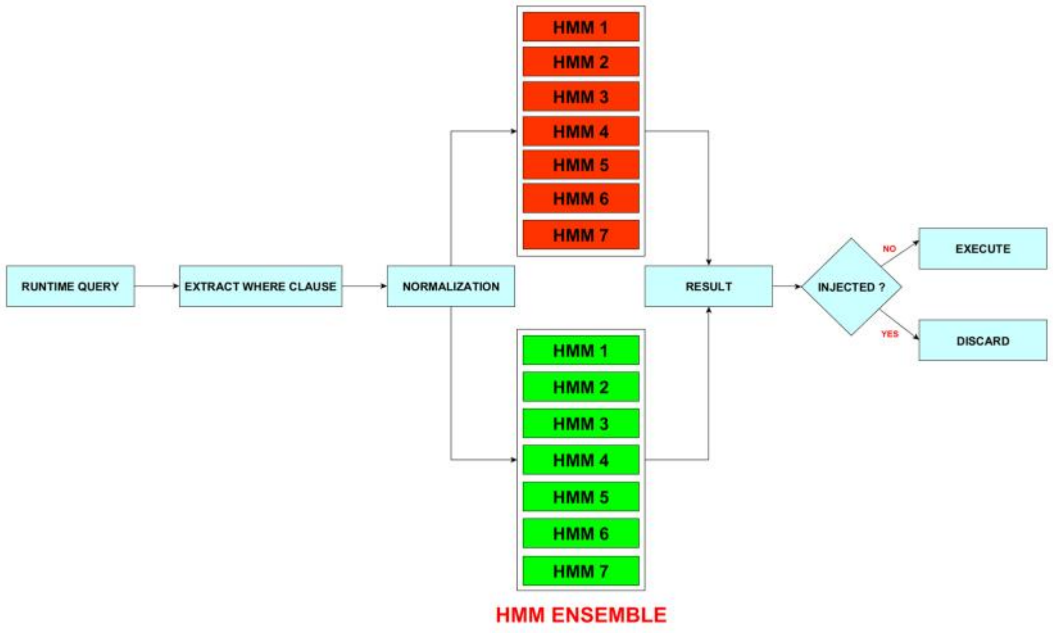


FIG 3.3: Runtime Part of the Model Used In Evaluation of Queries

This process is done runtime and it checks all incoming queries dynamically and evaluate it against the HMM parameters to decide the query type. And after that we can decide whether we will execute it or discard (if it is found injected).

4. <u>EXPERIMENTAL EVALUATION</u>

In this phase the rest 10% queries of each cluster formed in earlier stage is used for detection using the evaluate procedure of the Hidden Markov Models and the results are recorded along with the error percentage.

		PREDICTED	
		GENUINE	**INJECTED**
ACTUAL	**GENUINE**	TN = 50	FP = 3
	INJECTED	FN = 4	TP = 496

Table 4.1: Confusion Matrix

Accuracy	0.987314
True Positive Rate (Recall)	0.992
False Positive Rate	0.056603
True Negative Rate	0.943396
False Negative Rate	0.008
Precision	0.993987

Table 4.2: Result Matrix

<u>Our results were obtained with 98.73% accuracy and 99.83% precision</u>

5. PERFORMANCE OVERHEAD

We calculated the running time of each of our process for both genuine and injected queries. The performance overhead was then calculated and is shown below in Table 6:

HMM TRAINING (OFFLINE) FOR INJECTED QUERIES			
PROCESS	**# QUERIES**	**TIME (sec)**	**TIME/QUERY (ms)**
Query Normalization	4593	14.41517017	3.138508638
Clustering	4593	829.7662573	180.6588847
Training	4093	14734	3599.804544
TOTAL TIME REQUIRED			**3783.601938**

HMM TRAINING (OFFLINE) FOR GENUINE QUERIES			
PROCESS	**# QUERIES**	**TIME (sec)**	**TIME/QUERY (ms)**
Query Normalization	492	0.577187873	1.173146083
Clustering	492	5.397609234	10.97075048
Training	439	359.7849	819.5555809
TOTAL TIME REQUIRED			**831.6994774**

RUNTIME DETECTION PROCESS FOR INJECTED QUERIES			
PROCESS	**# QUERIES**	**TIME (sec)**	**TIME/QUERY (ms)**
Query Normalization	4593	14.41517017	3.138508638
Evaluate	500	5.3475	10.695
TOTAL TIME REQUIRED			**13.83350864**

RUNTIME DETECTION PROCESS FOR GENUINE QUERIES			
PROCESS	**# QUERIES**	**TIME (sec)**	**TIME/QUERY (ms)**
Query Normalization	492	0.577187873	1.173146083
Evaluate	53	0.2503	4.722641509
TOTAL TIME REQUIRED			**5.895787593**

Table 5.1: Performance Matrix

From Table 5.1, we can observe the following points:

- The time overhead was more for injected queries compared to genuine queries as the injected queries are usually longer.
- The time overhead is more for training process but it can be ignored as the HMM training is done just once and that too in offline mode.
- The time overhead for runtime detection is very less i.e. 5ms for genuine and 13ms for injected queries which shows the effectiveness of our system, incurring least overhead on our system.

6. CONCLUSION

In this project, we have first used query normalization scheme for transforming the queries to simple sentence like form. We adopted the strategy to examine only the WHERE clause part of run-time queries and ignore INSERT queries. This approach minimizes the size of the legitimate query repository. The system required an initial set of injected as well as genuine queries, these were collected from few automated tools and some web applications. We then used K-Medoid algorithm with Cosine Dissimilarity approach to cluster our queries into 14 groups (7 each for each type i.e. genuine and injected).

The clusters were used to train an HMM ensemble, consisting of 14 HMMs (one for each type). The HMM parameters obtained are used to check any query at runtime for deciding whether the query has to be executed or discarded.

The experimental results are very encouraging and confirm eﬀectiveness of our approach. The accuracy of the system is calculated to be 98.73%. Performance overhead of the system is almost imperceptible over Internet. The system acts as a database firewall and is able to protect multiple web applications hosted on a shared server, which is an advantage over existing methods. The approach can also be ported to other web application development platforms without requiring major modifications.

REFERENCES

[1] Halfond, W. G., Jeremy Viegas, and Alessandro Orso. "A classification of SQL-injection attacks and countermeasures." Proceedings of the IEEE International Symposium on Secure Software Engineering. Vol. 1. IEEE, 2006.

[2] Kindy, Diallo Abdoulaye, and Al-Sakib Khan Pathan. "A survey on SQL injection: Vulnerabilities, attacks, and prevention techniques." (2011): 77.

[3] Kemalis, Konstantinos, and Theodores Tzouramanis. "SQL-IDS: a specification-based approach for SQL-injection detection." Proceedings of the 2008 ACM symposium on Applied computing. ACM, 2008.

[4] Halfond, William GJ, and Alessandro Orso. "AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks." Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. ACM, 2005.

[5] Boyd, Stephen W., and Angelos D. Keromytis. "SQLrand: Preventing SQL injection attacks." Applied Cryptography and Network Security. Springer Berlin Heidelberg, 2004.

[6] Kar, Debabrata, and Suvasini Panigrahi. "Prevention of SQL Injection attack using query transformation and hashing." Advance Computing Conference (IACC), 2013 IEEE 3rd International. IEEE, 2013.

[7] Kar, Debabrata, Suvasini Panigrahi, and Srikanth Sundararajan. "SQLiDDS: SQL Injection Detection Using Query Transformation and Document Similarity." Distributed Computing and Internet Technology. Springer International Publishing, 2015. 377-390.

[8] Velmurugan, T., and T. Santhanam. "Computational complexity between K- means and K-medoids clustering algorithms for normal and uniform distributions of data points." Journal of computer science 6.3 (2010): 363.

[9] Roy, Dharmendra K., and Lokesh K. Sharma. "Genetic k-Means clustering algorithm for mixed numeric and categorical data sets." (2010).

[10] Stamp, Mark. "A revealing introduction to hidden Markov models." Department of Computer Science San Jose State University (2004).

[11] Ariu, Davide, Roberto Tronci, and Giorgio Giacinto. "HMMPayl: An intrusion detection system based on Hidden Markov Models." computers & security 30.4 (2011): 221-241.

[12] Resch, Barbara. "Hidden Markov Models A Tutorial for the Course Computational Intelligence." (2004).

APPENDIX

LIST OF TABLES

LIST OF FIGURES