New Strategy for Mitigating of SQL Injection Attack

Ammar Alazab
Al-Balqa' Applied University

Ansam Khresiat
Federation Universty, Australia

ABSTRACT
SQL injection attack (SQLIA) is a serious threat to web applications. A successful SQLIAs can have serious consequences to the victimized organization that include financial lose, reputation lose, compliance and regulatory breach. Therefore, developing approaches for mitigating SQLIA is paramount important. To this end, we propose an approach based on negative tainting along with SQL keyword analysis for detecting and preventing SQLIA. We have tested our proposed approach on all types of SQLIAs techniques by generating SQL queries containing legitimate SQL commands and SQLIA. We present an analysis and evaluation of the proposed approach to demonstrate its effectiveness in detecting and protecting SQLIA attack.

Keywords
Cybercrime, SQL Injection, SQLIA, Vulnerabilities, Web Application Security

1. INTRODUCTION
SQL Injection Attack (SQLIA) is a type of attack on web application, which occurs when an attacker inputs malicious strings as parameters in legitimate SQL statement [1]. SQLIA attacker takes advantage of improper coding of the web applications that allow the hacker to inject SQL query to get access to the data in the database. It is considered one of the most popular web application attack techniques used nowadays. It is a serious threat to the web application as it allows the hackers to gain complete access to the database server.

Although defensive coding, such as input validation represents a good mechanism to protect against SQLIA, they cannot protect against evasion techniques [2]. Also, they cannot protect the legacy web application that already has been deployed. A more common technique is preparing the SQL statement. This is used widely among commercial web development tools to protect against SQLIA. These statements are mainly created for the purpose of building efficient SQL queries and thus do not design to prevent SQLIA. Moreover, defensive coding is expensive, which makes it an impractical technique for protecting large legacy systems.

Firewalls and Intrusion Detection Systems (IDSs) are unworkable against SQLIA, because the signature keywords can be passed using the evasion techniques or alternate character encodings. SQLIA are still succeeding, and the defensive mechanisms are failing, for instance for the nonexistence of the right signature. Also, web application has been attack by using SQLIA via the firewall on port 80 or 443[3].

Web applications are deployed in many diverse forms with a wide range of functional capabilities. For that reason, there are repeatedly several possible ways of malicious input to be considered for these web applications. However, specifying all of them is naturally problematic and leads to high negative rate[4]. For instance, developers initially assumed that only direct user input needed to be marked as tainted. Successful SQLIAs confirmed that other malicious inputs sources, such as browser cookies, also required to be considered [5]. SQLIA is becoming extensively more widespread amongst hackers community. SQLIA and DDOS (Distributed Denial of Service) attacks are the most popular topics on hacker forums[6].

The most well-known malware and crime tool kits are those associated with the SQLIA such as Asprox, Conficker, Zeus and SpyEye [7-9]. These malware and crime tool kits are very dangerous because they target online banks system and perform a criminal activity. By using SQLIA help malicious writers to generate illegitimate web site.Thus, the malicious writers involve SQLIAs with crime tool kits. The Open Web Application Security Project (OWASP) recorded SQL injection as the most dangerous security threat affecting Web applications[10] In 2010. In the previous list in 2007, SQL injection was recorded at second place on their list of the ten most critical web application security vulnerabilities[11]. A recent survey conducted by GreenSQL, reveals that 88% of organizations still fail to protect their databases against both internal and external threats, and SQLIAs occur more than 70 times per hour [12].

During the past few years, a great deal of attention has been given to the problem of SQLIAs. Even though, most of the exiting researches have difficulties to address the full aspects of the SQLIAs. There are many techniques of SQLIAs and new advanced evasion techniques are creating on these kinds. The needs for consideration of these techniques are becoming urgent. Otherwise, different cybercrime threats will be occurred, and that you will be facing serious threats into business. In this paper, we propose a general model for protecting and detecting SQLIA based on SQL syntax at the web application layer, and negative taint at the database layer. The central idea is monitoring the data that comes from user's web browsers and matching with taint table on the database layer in the real time. If any of its malicious is recognized from user input, we stop the SQL statement from execution. Our techniques have been successful against all types of SQLIAs because the dynamic SQL statement is monitored through the database layer. The main contributions of our work can be summarized as follows: A new approach for detect and prevent SQLIA at the runtime. Applying negative taint in database layer helps us to identify untrusted data at the database layer. Thus, our method is able to detect maliciousness caused by tricky data and obfuscation techniques while minimizing false negatives. The major advantage of our approach apart from efficiency is that it does not change the web architecture.

The rest of the paper is organized as follows. An overview of the problem and related work is discussed in Section 2. An overview of the SQLIAs techniques are explained in Section 3. An evasion method of the SQLIAs techniques are discussed in Section 4. An overview of the proposed technique is presented in Section 5. We then present the results of the evaluation of the proposed approach in Section 6. The concluding remarks are presented in Section 6.

2. BACKGROUND

In this section, we present the system mode on web application. We also formalize the SQLIA detection problem and present review of related work.

2.1 System Model

Web applications are an application running over a network such as the internet or an intranet. They enable websites to become dynamic by making connections to the databases[13]. The high level system components of web applications are shown in figure 1. In the web application architecture, there are five layers; browsers, networks, web servers, web applications and databases. First, the client requests a page, either a static or a dynamic page.

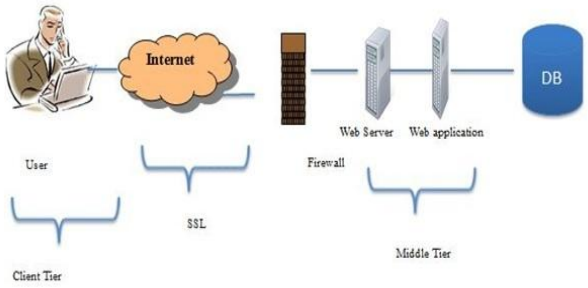

Figure 1. Web Application Architecture

Second, the web browser passes this request through the firewall to the web server. Third, the web server handles this request based on an initial configuration like Hypertext Transfer Protocol (HTTP) and Hypertext Transfer Protocol Secure (HTTPS), which can also handle these requests by "decoding" the webpage. Fourth, the web server passes this request to the web application server. Finally, the web application passes these requests to the database. In addition, the web application processes commands and verifies security access to the database through middleware such as JDBC, SQLJ, or JDO API, ODBC. After verifying the database access, the web application server sends the Structured Query Language (SQL) requests to the database server. Finally, the database server handle this requests by allowing storage, deletion, updating of the data, depending upon the SQL query and sends back the results to the application server.

Generally, web applications use query statements to generate strings to interact with the database. Usually these queries are generated by the web application servers such as ASP, JSP and PHP. A string contains both the query itself and its parameters which can be the user name and password. Then, the string is forwarded to the database server for checking as a single Structured Query Language (SQL) statement, if the received string compromised or injected it will cause data leakage. Therefore, it is necessary to protect web applications from illegal accesses.

Web applications are infamous for security vulnerabilities that can be victimized by writers of malware and hackers. The global accessibility of web applications is a serious problem, rendering them vulnerable to attack. One of the main threats on the web applications is SQLIAs that are extremely widespread in web applications [14]. Web applications offer an excellent facility to access the database through the internet [15], which has provided the required service to customers. Unfortunately, these advantages have raised a number of security vulnerabilities from improper code. Resulting, Structured Query Language Vulnerabilities (SQLV) that entitle hackers to have the ability to influence the Structured Query Language (SQL) that a web application passes to a back-end of a database. By inserting malicious code into strings to gain unauthorized access to a database to retrieve information or destroy the database (DB) where all the data is sensitive[16]. Web application security generally focuses on identifying vulnerabilities and malicious strings within web applications layer. Firewalls and Secure Sockets Layer (SSL) protocol information transferred between the site and client, but does not protect information against web application hackers, as they are built on top of web application infrastructure [17]. Therefore, it is easy to append data and commands into SQL statement. Even normal users can attempt direct connections to the databases through specific ports, bypassing the security mechanism [18].

2.2 SQL Injection Defined

SQLIA is one of the critical threats for web application. These attacks are presented through specifically characters attacker enters, on web applications that use on users browser to generate SQL queries. It has been experience for bad impact to the business, because it can lead to reveal of all of the critical data stored in the database, in such as, passwords, credit card, personal information and so on. SQLIA is normally used to compromise data base systems through vulnerable web applications. The SQLIA permits the attacker to get access to the whole or partial contents of databases. Moreover, SQLIA can make modifications to both the database schema and the contents. To perform a SQLIA in the most cases Web forms are used to inject part of SQL query. Entering SQL keywords and control signs an attacker is capable to alter the structure of SQL statement.

In SQLIA, malicious writers can take advantage of weakly web application development to launch their attack. The vulnerability happens when a web application improper authentication coded to authenticate the data a user might enter on the web page. Normally logion authentication in web page has two text box fields for entering a user name and password. Let user_name and user_password represent the names of these fields sequentially.

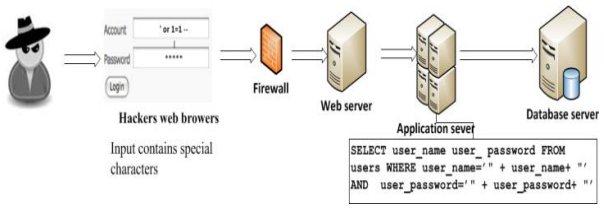

Figure 2. SQL Injection Example

The following code shows how to generate dynamic SQL query:

```
SELECT user_name user_ password FROM users
WHERE user_name='" + user_name+ "' AND
user_password='" + user_password+ "'
```

The above the SQL statement aimed to check the user name and password that exiting in the database with the user name and password that entering by the user in web form a shown in figure 2. The basic method of SQLIA contains straight addition of string into parameters that are attached with of the structure of SQL. A very straightforward attack may be potential by easily input string, like "1 OR 1=1" in the input field, the consequences from these inputs retrieve all database data. Another scenario, if an attacker is entered "user1 --" as user name and leaves password empty. The user's input is generated as the following SQL query:

```
SELECT user_name,user_password FROM users
WHERE uname='admin'' AND pass=' '
```

The consequences from above query, the attacker can get access to the database without correct user name or password.

3. RELATED WORK

There has been a lot of research performed in detecting and preventing SQLIA. Abawajy has suggested SQLIA detection and prevention approach for RFID (Radio Frequency Identification) systems[19]. His techniques work very well in the RFID tag based SQLIA. In another work for Fernando and Abawajy address SQLIAs through developing a model of legitimate SQL statements and then matching the generation query to conform to this approach through runtime monitoring[20]. The problem with this approach is that it expects the existence of the original SQL structure. Unfortunate, very little work exits to dealing with full aspects of the SQLIAs. A simple technique is to check for single quotes and dashes, and escape them manually. This is easily beaten, as attackers can simply adjust their input for the escaped characters.

web based SQLIA attacks Static Analysis: Pixy [21] is an open source prototype aimed at detecting SQL injection, cross-site scripting, or command injection based on flow-sensitive, inter-procedural and context-sensitive data flow analysis. In additional Pixy uses literal analysis to improve the rightness and precision for his results. Combined Static and Dynamic Analysis: AMNESIA [22] is a technique that combines dynamic and static for preventing and detecting web application vulnerabilities at the runtime. AMNESIA uses static analysis to generate different type of query statements. In the dynamic phase, AMNESIA interprets all queries before they are sent to the database and validates each query against the statically built models. AMNESIA stops all queries before they are sent to the database and validates each query statement against the AMNESIA models. However, the primary limitation in AMNESIA according to Ramaraj [23] is that the technique is dependent on the accuracy of its static analysis for building query models for successful prevention of SQL injection. Furthermore, AMNESIA doesn't consider there are certain types of code obfuscation or query development techniques that could make this step less precise and result in both false positives and false negatives.

Moreover, Martin, Livshits and Lam [24] proposed Program Query Language(PQL) that used static analysis and dynamic techniques to detect vulnerabilities in web applications. In static analysis information flow techniques to detect when malicious input has been used to generate an SQL query statement; these statements are then flagged as SQLIA vulnerabilities. According to Ramaraj [23] the limitation of this approach is that it can detect only known patterns of SQLIA.

Taint Based Approaches: WebSSARI (Web application Security via Static Analysis and Runtime Inspection) [25] is a tool proposed to a statically validate existing web applications and legacy web application code without any extra effort for the programmer and automatically protect potentially defective code. In this model, static analysis is applied to validate infect runs versus given conditions for sensitive formulation.

Code Checkers are based on static analysis of web application that can reduce SQL injection vulnerabilities and detect type errors. For instance, JDBC-Checker [26] is a tool used to code check for statically validating the type rightness of dynamically-generated SQL queries. However, researchers have also developed particular packages that can be applied to make SQL query statement safe [27]. These techniques are good, but need extra effort from programmers to build queries statements using Application Program Interface APIs especially for legacy web application because lack of information about the intent of the programmer.

Tainted Data Tracking: this method is proposed by Halfond [28], it is based on track tainted-ness of data and check specifically for dangerous content that comes from user input. According to Nguyen-Tuong et. al.[29] this can be done via instrumenting the run time environment or interpreter of the back-end scripting language. When an SQL statement is sent to the database server, its syntax tree is first examined. However, this approach does not provide any way to check the correctness of the input validation routines [30]. However programs that use incomplete input checking routines may pass these checks and still be vulnerable to injection attacks [14] .

A number of commercial tools have a strong package library to help a developer from SQLIAs. For instance, J2EE has an especially SQL query against SQLIAs, and Microsoft's NET has the same. The efficiency of these SQL query statements at protecting from SQLIAs are reliant on the development stage, which is based in database layer, and these typically written by the third party vendors. Thus, it is very difficult to use these commercial tools for legacy web application.

4. IMPACT OF SQL INJECTION

Several of the prominence SQLIA has been affected on information security because of the break of confidentiality in the information stored in the Databases. This loss of confidentiality and the resulting financial costs for recovery, downtime, regulatory penalties, and negative publicity represent the primary immediate consequences of a successful compromise. The following table 1 shows the Impacts of successful SQLIA.

Table1. Impacts of Successful SQLIA

Impacts	Explanation
Authentication Bypass	This attack allows SQLI attacker to get access to a database layer, possibly with admin privileges, without providing a correct username or password.
Information Disclosure	This attack allows SQLI attacker to gain for sensitive information that is stored in a database such as credit card information.
Compromised Data Integrity	This attack allows SQLI attacker to modify the contents of web page. The consequence from this is defacing a web page.
Compromised Availability of Data	This attack allows SQLI attacker to remove information in order to cause damage to information that is stored in a database.
Remote Command Execution	This attack allows SQLI attacker to perform command execution through a database, which let the attacker control for the operating system.

SQL injection attacks have been related with many high profile data breached as shown in the following table. Attacked increasingly, become targeted, and exploitation is faster.

Table 2. Real Examples of SQLIA Attack

Year	Example	References
2008	- Appear malware that use SQLIA such as Asprox. Asprox is a kind of malware that used the two threat vectors of forming a botnet and of generating SQLIAs. .Asprox is used SQLIAs techniques in order to expand its Botnet. It is attack legitimate websites and injects scripts that redirected the users to illegitimate web sites.	[31]
2009	- The site of BitDefender's Portugese, Kaspersky and F-Secure web sites were hacked using the SQLIAs.	[32] [33].
2009	- The US Justice Department charged an American citizen Albert Gonzalez and two unidentified Russian accomplices on charges related to data intrusions at Heartland, Hannaford Bros., 7-Eleven Inc. and three other retailers. - Gonzalez is alleged to have masterminded an international operation that stole a staggering 130 million credit and debit cards from those companies.	[34]
2010	- Half million web sites are hit with automated SQLIA. - Royal Navy web site has been attacked by SQLIA.	[11]
2011	- Barracuda, vendor of web application firewall, breached by SQL Injection. - Sony breached by automated SQL injection attack: Sony BMG Greece, Sony Music Japan, Sony Canada, Sony Pictures France, Sony, pictures Russia and Sony Music Portugal. - oracle owned MySQL has its website compromised	[35]
2012	- SQLIA attackers have lunch attack against the following sites: LinkedIn, eHarmony, Last.fm, Yahoo, Android Forums, Billabong, Formspring, Nvidia, and Gamigo. - Nvidia acknowledged SQLIA attacker swiped up to 400,000 user accounts.	[11]
2013	- SQL Injection Found on the Site of Islamic Bank Bangladesh.	[6]

5. SQL INJECTION ATTACK TECHNIQUES

5.1 Tautology

Tautology-based attacks work through injecting code by one or more conditional SQL statement queries in order to make the SQL command evaluate as a true condition. The most common use of this technique is to bypass authentication on web pages resulting in access to the database.

SQL injected command below shows how the attacker can make the SQL command evaluate as true without knowing neither the password or the username, attackers can achieve this by many methods, one of these methods is by using blank statements for the username and for password using a true condition such as (1=1) or (- -) resulting in accessing the database or returning all data in the table username.

```
SELECT * FROM userTable WHERE username=''OR 1=1 --AND password=''
```

Detect tautology techniques could be extremely difficult. AS SQL statements allow a broad scope of user definition function and open permitting inputs values. However, the attackers can input many forms of tautologies like:

```
2=2, 3=3, '1'='1', 'b'='b" or "name"="name"
….
```

Which will considered valid statement regardless of the username input by the SQLIAs, and will be able to bypass authentication mechanisms. For example, other SQL tautologies are 'user1' LIKE '%user%' which are created from operators. However, the SQLIA attacker can simply handle the original SQL query by adding character to include a tautology, such as 22 OR 1=1.

5.2 Union Query

Union attack uses an operator used to combine result to retrieve addition information, since UNIONS added to an addition statement to execute a second statement and third statement to retrieve information from a specified table. If the Attacker identified the structure of the tables, it can simply attach another statements using union query, as the following example

```
SELECT pass FROM user_table1 WHERE loginID='' UNION SELECT pass from user_table2 where Username=xxx -- AND pass=''.
```

5.3 Stored Procedures

A stored procedure is a subroutine available to most commercial database in order to reuse the code more than time. Once the stored procedure is modified, all clients automatically get the new version. Stored procedures provide developers with an extra layer of abstraction because they can enforce business wide database rules, independent of the logic of individual Web applications. Unfortunately, it is a common misconception that the mere use of stored procedures protects an application from SQLIAs. Attackers try to execute stored procedures that are stored in the database. Specially, most database companies store procedures that extend the functionality of the database and allow for interaction with the targets beyond the database and operating system. With stored procedures, the code that creates the query is stored and executed on the database. The following example of stored

procedures shows how the attacker exploits a parameterized stored procedure.

```
CREATE PROCEDURE DBName .is Authenticated
@user Name varchar2, @pass varchar2, @pin
int AS EXEC("SELECT accounts FROM users
WHERE login='" +@user Name+ If' and pass='"
+@password+ and pass=" +@pass);
```

The authorized/unauthorized user stored procedure returns true/false. If the SQLIAs input SHUTDOWN; - -" for username or password. Result, the stored procedure generates the following query statement:

```
SELECT Username FROM UserTable WHERE
username= user1 AND pass=' '; SHUTDOWN;
```

5.4 Piggy-Backed Queries

Piggy-backed Queries is a type of attack that compromises a database using a query delimiter, such as "; ", to inject additional query statements to the original query. Since the original query is a legitimate query, whereas additional queries could be illicit. The result is the attacker can inject any SQL command to the database. In the following example, the attacker injects 0; drop table user" into the pin input field instead of logical value. Then the application would produce the query:

```
SELECT pass FROM userTable WHERE
login='user1' AND Password = 0; drop table
users
```

Since the database accepts both queries statements and executes them. The second query is an illegal statement, and the result drops the users table from the database.

5.5 Blind Injection

This occurs when programmers forget to hide an error which renders the web application insecure, this error message help SQLIA to compromise the database through asking a series of logical questions through SQL statements.

```
SELECT pass FROM userTable WHERE username=
'user' and 1 =0 -- AND pass = AND pin= 0
SELECT info FROM userTable WHERE username=
'user' and = 1 -- AND pass = AND pass= 0
```

In the above example, first the SQLIAs send the first query with a logical error and receive an error message like "1 =0 because this error message enables the attacker to understand the structure of database. Once the attacker understands the structure he sends a query which is mostly true.

5.6 Timing Attacks

The SQLIAs collects information from a database by monitor the response time of the database. This kind of attack used if condition statement to achieve a time delay purpose.

```
declare @varchar(8000) select @ =
db_Alias() if (ascii(substring(@, 1, 1)) &
( power(2,0))) > 0 waitfor  delay '0:0:6'
```

6. EVASION METHODS

Signature detection or pattern matching engine has been proven inefficient in the detection of SQL injection [36]. There are many methods adopted by malicious authors in order to evade from the detection engines [37] [38]. For every signature created, a new evasion technique can be developed as the malicious authors take advantage of the rich language provided by SQL to fool and thwart the signature based detection. Also, a recent effort by malicious writer to automate the finding of web application vulnerability has increased. The following sub section outlines the popular evasion methods that adopted by malware authors.

6.1 Encoding Evasion

One of the earliest methods noted and used by malicious authors is using the equivalent of text in order to defeat detection engines, in the same way as Domain Name Server (DNS) changes the domain name to IP address. Hence the user does not know the IP address but knows the domain name. Encoding evasion method is illustrated in Table 3.

Table 3. Encoding evasion method

Logical and Equivalence Expressions	SQL statement	SQL injection
Base 64	1=1	MT0x
Encoding Decimal	1=1	1=1
Encoding Hex	1=1	313d31
URL Encoding	1=1	1%3D1
UTF-8 Base10	1=1	049 061 049

6.1.1 White Spaces Evasion

Since keyword sequence is counted as a string, the malicious authors are adopting the use of white spaces in order to evade from detection. White space around the same code will lead to evasion from the detection engine because the signatures method are generally looking for the exact text match, and by adding one or more spaces around the SQL keywords would fool the signature detection of text with no spaces. Table 3 below shows an example. However, detection engines can remove all the white spaces from the SQL statement in order to detect malicious code. Therefore, malicious authors are using not just whitespace they also use special characters such as Tab, Enter key (carriage return) and line feed around SQL keywords in order to evade from signature based detection. Examples of such evasion are the use of '\t' for tab, '\n' for a new line, as shown in table 4.

Table 4. white spaces evasion method

Comment Evasion	SQL statement	SQL injection
Using comment /* */	Union	Un/* */ ion
Using comment --	Union	Un--ion

6.1.2 Comment Evasion

Malicious writer use comments to break the SQL keywords without any effect on the code in order to evade from signature detection. Table 5 illustrates this method.

Table.5 Comment evasion

Comment Evasion	SQL statement	SQL injection
Tab	1 '\t'=1	Char(9)
Carriage return	1 '\n'=1	Char(13)
Line feed	1=1	Char(10)

6.1.3 Logical and Equivalence Expressions Evasion

Malicious authors use the logical, mathematical or equivalence expressions in order to evade detection as shown in table 6. Generic signature is likely to lead to false positives since some combination of "or" and "=" are likely to legitimately occur within normal Web content. But even if it did not lead to false positives, it can also be easily evaded by simply replacing such an expression with a malicious expression that evaluates as true.

Table 6. Logical and equivalence expressions evasion method

Encoding Evasion Method	SQL statement	SQL injection
Logical expression	2>1	2>1
Mathematic expression	4=4	1+3=2+2
Equivalence expression	1=1	user=user

6.1.4 String Techniques Evasion

Hackers evade from signature detection by breaking SQL keywords using concatenation symbols. Examples of such methods are shown in table 7.

Table 7. string techniques evasion method

String Techniques Evasion	SQL statement	SQL injection
Variable	a$=union	a$
Concatenation	Union	Un‖ion

7. SQLIA DETECTION AND PREVENTION APPROACH

As explained earlier the client first requests a page, either static or dynamic, and the web browser pass the request to the web server. Our proposal methodology explained in figure 3. The web server handles the requests by decoding the webpage. Subsequently, the web server passes this request to the web application server, and the web application server activates the business tier that allows connection to the database. This layer processes commands, verifies security access to database through middleware such as JDBC, SQLJ, or JDO API, ODBC, etc. and makes logical decisions. After verifying the database access, the web application server sends the SQL requests to the database server. The database server processes the request by allow to store, delete and update the data depend upon the SQL query and sends back the results to the application server. Our approach performs negative taint by storing untrusted markings, based on the evasion methods discussed above, at the database layer. Also, performs syntax-aware evaluation in web application server of query strings, before executing the query in the database, by validating queries whose input matches with untrusted markings that contain one or more characters without trust markings, the matching process done with SQL keywords and operators.

In the situation of preventing SQLIAs, these conceptual advantages of positive tainting are especially significant. The way in which Web applications create SQL commands makes the identification of all untrusted data especially problematic and, most importantly, the identification of all trusted data relatively straight forward. In the situation of detecting SQLIAs, the technique uses runtime checking to examine the SQL queries and match them against the signature database. The proposed model accomplished by evaluating the query and check for parameterized queries that are widely used in web technology (such as JSP, ASP, and PHP) so that the data gets separated through parameters [3])

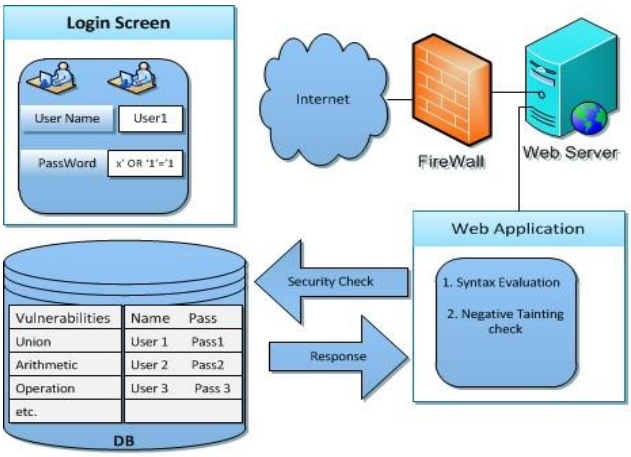

Figure3.Methodology

Table. 8 Input validation

Usernames	Password	SQL Legitimate	Database Access
T	T	T	YES
T	T	F	NO
T	F	T	NO
T	F	F	NO
F	T	T	NO
F	T	F	NO
F	F	T	NO
F	F	F	NO

For effective of prevention, we implemented the signature with regular expression style in the database to capture illegitimate SQL statements. Table 9 shows the contents of a SQLIA signatures and matching expression symbols. Alphanumeric means the alphabets. Comment Mark points that the rest of the SQL statement is ignored and do not have any effect on SQL statement Quotation Mark is indicating the boundary of SQL. Type means the data variable in SQL statement. Type Conversion its indicate Converting a SQL data type to anther a SQL data type. SQL Keyword contains all SQL data manipulation keywords such as select, insert, update and delete. Data definition keywords such as create a table and drop table. Data control keywords such grant and revoke privileges. Delimiter Mark is indicating of end an SQL query statement. Square brackets used to encase characters or ranges of characters in database searches.

TABLE.9 Elements and symbols in signature.

Elements	Symbols
Alphanumeric	{a, b,…,z}, {A,B,..,Z},{0,1,..,9}
Comment Mark	{//,--}

Quotation Mark	{“,’}
Arithmetic operation	{+,-,/,*}
Logical Keyword	{AND, OR}
Delimiter Mark	{;}
SQL keyword	{create, select, drop, delete
……	…….

The goal of this work is to detect illegitimate access to the database, where the username and password are correctly validated, as showing in table 8. To achieve our goal, we maintain a lookup table containing possible vulnerabilities that malicious authors exploit using evasion methods as discussed above. Using our negative taint model, all requests (whether legitimate or not) received by the web server are validated for authentic database connections through username and password match, for markings with vulnerability table before they are forwarded to the database server.

Algorithm 1 shows the checking negative taint; the algorithm takes the user name input, password input and table that contain malicious string from database. The algorithm runs the users inputs against collect taints set which is a store in the database, if user’s inputs match with database then it’s raise alarm, if user’s input contain the keyword from taint set then creates attack vectors.

Algorithm 1 to prevent SQLIA.

```
  Inputs: UserName, Password, Table Taint
  Output: Injection= True/False

  While username not null OR password not
null do
  Validate UserName and Password against
  TaintSets Table
  If (UserName OR Password In Table Taint)
then
  Return true
  Else
  Return False
```

8. PERFORMANCE EVALUATION

In this section, we present an analysis and performance evaluation of the proposed approach to show its efficiency in preventing SQLIA attack. We also discuss the results of the experiment.

8.1 Experiment Setup

We implemented our method by using oracle 10g for the database layer and simulated a hacking environment using dataset containing both SQL injected Queries and legitimate queries. The middle tier is configured as the web application server. Also, we created the login tables for all the users’ names with their passwords as shown in table 10.

TABLE.10 Login table

username	password
User1	Pass1
User2	Pass2
….	…..

The vulnerabilities in table 9 configure as the following: First, create table

Algorithm 2 to create login tables in the Database

```
Create table login (usernames varchar(20),
password varchar(20), primary key (username));
```

The following table, used to store a SQLIA

Algorithm 3 to create TAINTS tables in the Database

```
Create table TAINTS (keyword varchar2 (20)
primary key);
```

The following Java code, used to perform a login function, and perform SQLIA detection at the web application layer:

Algorithm 4 to detect the SQLIA.

```
UserPass = request.getParameter("password").toSt
ring();
strQuery="select * from login where”
IF(request.getParameter("username")!=null && req
uest.getParameter("username")!="" && request.get
Parameter("password")!=null &&
 request.getParameter("password")!="")  AND
request.getParameter("username")!= NOT
IN  (select keyword from TAINTS ) AND
request.getParameter("password") != NOT
In   (select keyword from TAINTS )){
strQuery="
username='"+username+"' AND  password='"+userpas
s+"'";
 st = conn.createStatement();
 rs = st.executeQuery(strQuery);}
Else
   response.sendRedirect("login.jsp");
```

The SQL statement with username and password matched correctly may not be always mean it is a legitimate statement as they may contain vulnerabilities exploited by malicious authors using the abovementioned evasion methods. Using our proposed model, SQL queries that contain vulnerabilities are identified as SQLIAs blocked, and a report generated is sent to web developer and database administrator.

8.2 Empirical Evaluation

In our evaluation, we evaluated the effectiveness and efficiency of our method.

Q1. What Percentage of correctly identified SQLLA? This question address True detection Rate (TP rate)

Q2. What Percentage of wrongly identified SQLLA? This question addresses False Negative alarm Rate (FN rate).

Q3. What Percentage of wrongly identified legitimate SQL query? This question address False Positive (FP rate).Which means an activity is normal but it is identified as the SQLIA.

We conducted our experiments to test our proposed model by generating 1,200 SQL queries containing all evasion methods of SQL injection attacks, as well as 1,320 SQL queries contain legitimate SQL commands. We evaluated the results by using three applications that are available on the internet GotoCode (http://www.gotocode.com/) which are; Employee Directory, Online Bookstore, and Online Portal. Such application sources have been utilized by other researchers [28] too for testing purposes. Table 11, table 12, and table 13 provide the overall results of our experiments, which indicate that our model

provide 100% web application protection and 0% for false negative rate and false positive rate. Also it shows that our proposed model has been successful in identifying any of the abovementioned popular evasion methods adopted by an SQLIA.

Standard performance metrics are used to analyze the different test cases which are defined as follows:

- True detection Rate (TP rate): Percentage of correctly identified malicious code.

$$TP\ Rate = \frac{TP}{TP + FN}$$

- False alarm Rate (FP rate): Percentage of wrongly identified benign code, given by:

$$FP\ Rate = \frac{FP}{FP + TN}$$

- Overall Accuracy: Percentage of correctly identified code, given by:

$$Overall\ Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Table below shows Percentage of wrongly identified SQLIAs and Percentage of correctly identified SQLLAs.

Table 11. Percentage of wrongly identified SQLIAs and Percentage of correctly identified SQLLAs

Application Name	Total number of attacks	Successful attack (FN)	True detection (TP)
Employee Directory	1200	0	1200
Online Bookstore	1200	0	1200
Online Portal	1200	0	1200

Table below shows percentage of wrongly identified legitimate SQL query - false positive (FP rate).

Table 12. Percentage of wrongly identified SQLIAs and Percentage of correctly identified SQLLAs

Application Name	Total number of legitimates	False positive
Employee Directory	1320	0
Online Bookstore	1320	0
Online Portal	1320	0

Fig. 4 shows the results of the experiment. The result provides some assurance that the proposed model can be implemented without significant oversight. The timing results show that the proposed approach is quite efficient as it imposes virtually very low overhead on the system, that it would be determined by the network speed and database server access.

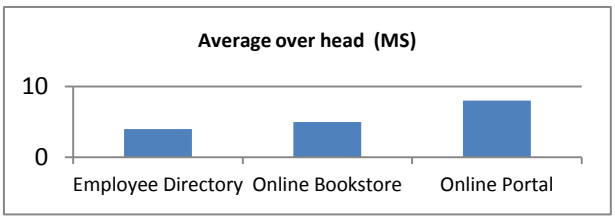

Figure. 4 Overhead performance

Halfond[23] and Lee etc. [39] are categorized SQLIA into several categories and used them to evaluate the effectiveness for prevention SQLIAs. We used the same techniques of Halfond to evaluate our techniques via other techniques. The results are shown in Table 13.

Table 13 Comparison of SQLI Detection/Prevention Techniques with Respect to Attack Types

Techniques	Tautologies	Illegal	Union	Piggy Query	Stored Procedures	Inference	Alternate Encoding
AMNESIA[40]	●	●	●	●	✕	●	●
CSSE[41]	●	●	●	●	✕	●	✕
SQLCheck[42]	●	●	●	●	✕	●	●
SQLGuard[43]	●	●	●	●	✕	●	●
SQLrand[44]							
Tautology-checker	●	✕	✕	✕	✕	✕	✕
Web App. Hardening[45]	●	●	●	●	✕	●	✕
IDS[46]	○	○	○	○	○	○	○
Our approach	●	●	●	●	●	●	●

9. CONCLUSIONS

Web applications have become an essential and integral part of internet usage today as they provide the convenience of business and personal transactions anywhere anytime. However, SQLIAs pose a serious threat to web applications hence; the primary purpose of this research was to present a new model to protect and detect web applications against SQLIAs with least modification of the Web architecture. Our proposed model was developed based on negative tainting and SQL syntax-aware methods, and was evaluated through SQL penetration testing in web application and database server. The negative tainting and SQL syntax-aware that we use gives our technique several significant advantages over techniques based on other mechanisms.

Evaluations have been performed using three different applications. We were able to successfully distinguish between legitimate SQL queries and malicious ones that had adopted various evasion methods such as encoding, comments and white space evasion methods as well as logical expressions and string techniques that were not captured by commercially available detection engines.

10. REFERENCES

[1] C. Torrano-Gimenez, A. Perez-Villegas, and G. Alvarez, "WASAT-A New Web Authorization Security Analysis Tool," Web Application Security, pp. 39-49, 2010.

[2] P. Bisht, P. Madhusudan, and V. Venkatakrishnan, "CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks," ACM Transactions on Information and System Security (TISSEC), vol. 13, p. 14, 2010.

[3] A. Alazab, J. H. Abawajy, and M. Hobbs, "Web Malware that Targets Web Applications," in Social Network

Engineering for Secure Web Data and Services, ed: IGI Global, 2013, pp. 248-264.

[4] W. G. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," in Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, 2006, pp. 175-185.

[5] A. Alazab, M. Alazab, J. Abawajy, and M. Hobbs, "Web application protection against SQL injection attack," in ICITA 2011: Proceedings of the 7th International Conference on Information Technology and Applications ICITA 2011, 2012, pp. 1-7.

[6] Softpedia. (2013, April). Stories about: SQL injection. Available: http://news.softpedia.com/newsTag/SQL+injection

[7] Y. Shin, S. Myers, and M. Gupta, "A Case Study on Asprox Infection Dynamics," Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 1-20, 2009.

[8] N. Lowe, "Shields Up! Protecting browsers, endpoints and enterprises against web-based attacks," Network Security, vol. 2009, pp. 4-7, 10// 2009.

[9] A. K. Sood, R. J. Enbody, and R. Bansal, "Dissecting SpyEye – Understanding the design of third generation botnets," Computer Networks, vol. 57, pp. 436-450, 2/4/ 2013.

[10] Open Web Application Security Project. (2010, 3 April). The Top 10 Most Critical Web Application Security Risks. Available: https://www.owasp.org/index.php/Main_Page

[11] D. Hartley, "Chapter 1 - What Is SQL Injection?," in SQL Injection Attacks and Defense, ed Boston: Syngress, 2012, pp. 1-25.

[12] Greensql. (2013, April). GreenSQL December Survey. Available: http://www.greensql.com/content/greensql-december-survey-88-all-companies-surveyed-do-not-protect-their-databases-external-a

[13] A. Alazab, J. Abawajy, and M. Hobbs, "Web Malware That Target Web Application," Social Network Engineering for Secure Web Data and Services, Luca Caviglione, Mauro Coccoli, Alessio Merlo (Eds.) IGI Global, USA., 2013.

[14] P. Bisht, P. Madhusudan, and V. Venkatakrishnan, "CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks," ACM Transactions on Information and System Security vol. 13, pp. 1-39, 2010.

[15] C. S. Peng, S. K. Chen, J. Y. Chung, A. Roy-Chowdhury, and V. Srinivasan, "Accessing existing business data from the World Wide Web," IBM Systems Journal, vol. 37, pp. 115-132, 2010.

[16] A. Alazab, M. Alazab, J. Abawajy, and M. Hobbs, "Web application protection against SQL injection attack," in ICITA 2011: Proceedings of the 7th International Conference on Information Technology and Applications ICITA 2011, 2011, pp. 1-7.

[17] W. D. Yu, D. Aravind, and P. Supthaweesuk, "Software vulnerability analysis for web services software systems," 2006, pp. 740-748.

[18] K. J. Vella. (2007, 2011). Web Applications: What are they? What about them? Available: http://www.windowsecurity.com/articles/Web-Applications.html?printversion

[19] J. Abawajy, "SQLIA detection and prevention approach for RFID systems," Journal of Systems and Software, vol. 86, pp. 751-758, 3// 2013.

[20] H. Fernando and J. Abawajy, "Securing RFID systems from SQLIA," in Algorithms and architectures for parallel processing, ed: Springer, 2011, pp. 245-254.

[21] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities," in IEEE Symposium on Security and Privacy, Oakland, CA, 2006, pp. 258-263.

[22] W. G. J. Halfond and A. Orso, "Preventing SQL injection attacks using AMNESIA," presented at the Proceedings of the 28th international conference on Software engineering, Shanghai, China, 2006.

[23] IndraniBalasundaram and Ramaraj, "An Approach to Detect and Prevent SQL Injection Attacks in Database Using Web Service," International Journal of Computer Science and Network Security, vol. 11, pp. 197-205, 2011.

[24] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: a program query language," ACM SIGPLAN Notices, vol. 40, pp. 365-383, 2005.

[25] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," presented at the Proceedings of the 13th international conference on World Wide Web, New York, NY, USA, 2004.

[26] C. Gould, Z. Su, and P. Devanbu, "JDBC checker: A static analysis tool for SQL/JDBC applications," 2004, pp. 697-698.

[27] R. A. McClure and I. H. Krüger, "SQL DOM: compile time checking of dynamic SQL statements," 2005, pp. 88-96.

[28] W. G. J. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," 2006, pp. 175-185.

[29] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," Security and Privacy in the Age of Ubiquitous Computing, pp. 295-307, 2005.

[30] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. Venkatakrishnan, "CANDID: preventing sql injection attacks using dynamic candidate evaluations," 2007, pp. 12-24.

[31] Y. Shin, S. Myers, and M. Gupta, "A case study on asprox infection dynamics," in Detection of Intrusions and Malware, and Vulnerability Assessment, ed: Springer, 2009, pp. 1-20.

[32] A. K. Sood, "The crux and the myth — breaches in security vendor websites," Computer Fraud & Security, vol. 2009, pp. 11-13, 7// 2009.

[33] A. K. Sood, "The crux and the myth—breaches in security vendor websites," Computer Fraud & Security, vol. 2009, pp. 11-13, 2009.

[34] W. Kim, O.-R. Jeong, C. Kim, and J. So, "The dark side of the Internet: Attacks, costs and responses," Information systems, vol. 36, pp. 675-705, 2011.

[35] C. Tankard, "Advanced Persistent threats and how to monitor and deter them," Network Security, vol. 2011, pp. 16-19, 8// 2011.

[36] D. Das, U. Sharma, and D. Bhattacharyya, "An Approach to Detection of SQL Injection Vulnerabilities Based on Dynamic Query Matching," International Journal of Computer Applications, vol. 1, pp. 39-45, 2010.

[37] M. Alazab, S. Venkataraman, and P. Watters, "Towards Understanding Malware Behaviour by the Extraction of API Calls," in Second Cybercrime and Trustworthy Computing Workshop, Ballarat, VIC, 2010, pp. 52-59.

[38] M. Alazab, S. Ventatraman, P. Watters, M. Alazab, and A. Alazab, "Cybercrime: The Case of Obuscated Malware," in 7th International Conference on Global Security, Safety & Sustainability, Thessaloniki, Greece, 2011.

[39] I. Lee, S. Jeong, S. Yeo, and J. Moon, "A novel method for SQL injection attack detection based on removing SQL query attribute values," Mathematical and Computer Modelling, vol. 55, pp. 58-68, 1// 2012.

[40] W. G. J. Halfond and A. Orso, "Preventing SQL injection attacks using AMNESIA," 2006, pp. 795-798.

[41] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," SIGPLAN Not., vol. 41, pp. 372-382, 2006.

[42] S. Thomas and L. Williams, "Using Automated Fix Generation to Secure SQL Statements," presented at the Proceedings of the Third International Workshop on Software Engineering for Secure Systems, 2007.

[43] G. Wassermann and Z. Su, "An analysis framework for security in Web applications," in Proceedings of the FSE Workshop on Specification and Verification of component-Based Systems (SAVCBS 2004), 2004, pp. 70-78.

[44] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," presented at the Proceedings of the 5th international workshop on Software engineering and middleware, Lisbon, Portugal, 2005.

[45] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, Automatically hardening web applications using precise tainting: Springer, 2005.

[46] F. Valeur, D. Mutz, and G. Vigna, "A learning-based approach to the detection of SQL attacks," in Detection of Intrusions and Malware, and Vulnerability Assessment, ed: Springer, 2005, pp. 123-140.