Jurnal Teknologi

Full Paper

REVIEW ON SQL INJECTION PROTECTION METHODS AND TOOLS

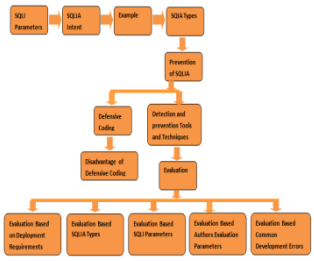
Muhammad Saidu Aliero*, Imran Ghani, Syeed Zainudden, Muhammad Murad Khan, Munir Bello

Department of Computer Science, Faculty of Computing, Universiti Teknologi Malaysia, 81310 UTM Johor Bahru, Johor, Malaysia

Article history
Received
15 April 2015
Received in revised form
29 September 2015
Accepted
12 November 2015

*Corresponding author
Msaidua2000@gmail.com

Graphical abstract



Abstract

SQL injection vulnerability is one of the most common web-based application vulnerabilities that can be exploited by SQL injection attack. Successful SQL Injection Attacks (SQLIA) result in unauthorized access and unauthorized data modification. Researchers have proposed many methods to tackle SQL injection attack, however these methods fail to address the whole problem of SQL injection attack, because most of the approaches are vulnerable in nature, cannot resist sophisticated attack or limited to scope of subset of SQLIA type. In this paper we provide a detailed background of SQLIA together with vulnerable PHP code to demonstrate how attacks are being carried out, and discuss most commonly used method by programmers to defend against SQLIA and the disadvantages of such an approach. Lastly we reviewed most commonly use tools and methods that act a firewall for preventing SQLIA, finally wean alytically evaluated reviewed tools and methods based on our experience with respect to five different perspectives. Our evaluation results point out common trends on current SQLI prevention tools and methods. Most of these methods and tools have problems addressing store-procedure attacks, as well as problems addressing attacks that take advantage of second order SQLI vulnerability. Our evaluation also shows that only a few of these methods and tools considered can be deployed in all web-based application platforms.

Keywords: Attack, prevention, method, approach, injection, parameters, query

1.0 INTRODUCTION

Study shows that the security of web applications is, in general, quite poor and demand to use these applications is very high. Technology and networks enable organizations to adopt web based applications as backbone to conduct their day to day activities. For instance, E-commerce, health care, transportation, social activities are all now available on web-based database driving applications. These applications process data and store the result in back-end database server where the organization's related data are stored. Depending of the specific purpose of application, most communicate frequently with customers, users, employees to enable them to use the service offered by the organization. The fact that these applications can be invoked by anyone worldwide drew the attention of attackers who wish to benefit from these vulnerabilities. According to a report published on security survey by open web application security project (OWASP) and SAN security report in 2014, over 63% of web applications worldwide are at risk of being hacked as a result of their vulnerabilities [1].

SQLI (SQL injection) vulnerability is the one of web-based database driving application vulnerabilities that presents high risks for organizational assets. SQLI vulnerability results from inappropriate validation of input from user, which enables the attacker to manipulate programmer intended queries by adding new SQL operator, command, keyword, or clause enabling unauthorized database modification and bypassing authentication mechanisms.

Researchers have investigated different methods and different approaches to enable programmers to secure their queries by applying secure coding in web application source codes as well as standalone tools

that act as web fire wall from the client side thereby gaining unauthorized access to restricted data, performing intercept dynamic queries by user and checking for existence of SQLIA (SQL injection Attack). The question to ask is why is it that for the past few decades, despite efforts by different researchers to eliminate the problem of SQLIA, there is still no existing solution that completely eliminates the entire problem of SQLIA.

In this regard we provide a detailed explanation of defensive coding practice and problems associated with defensive coding practice to make programmers aware of the existing issues in using defensive coding practice. We also provide evaluation of different existing detection and prevention methods to prevent security administrators from blindly choosing tools advertised by vendors as different vendors claim to have best tools to tackle the problem. Our evaluation will also shed light on researchers that are interested in improving the existing tools. Many researchers have already made similar efforts in this field; however our evaluation is different from others by evaluating methods on different perspective that others have not done. Figure 1 below shows the organization of this paper.

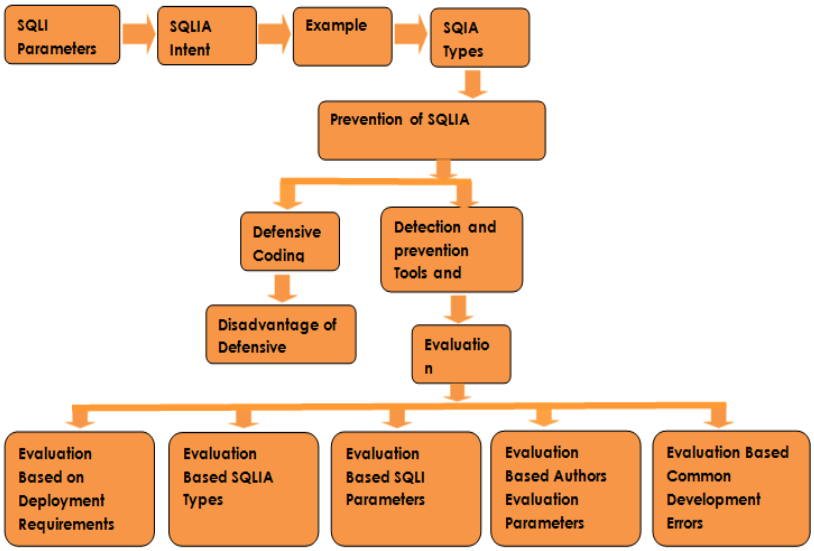


Figure 1 Organization of the paper

2.0 BACKGROUND OF SQLIA

Most database driving websites required customers to be a member in order to have access to the services they offered. In this case a customer is expected to register to use the service offered by the application. Normally registered users have restrictions on viewing and manipulating data within applications. Now because of the dynamic feature of SQL statement and logical knowledge possessed by other people on how to manipulate the way to communicate with the database it is easier for an attacker to have unauthorized access to the system, bypass authentication mechanisms and unauthorized data manipulation on backend database through injection parameters of the website.

2.1 Injection Parameters

Attackers use injection parameters to inject malicious code in an application. Many SQLI prevention tools and methods used today can be deployed in one or more injection parameters described below.

Injection through User input field**:** user input fields are provided in web applications to enable web application users to request information from backed database to the user with help of HTTP POST and GET. These inputs are connected with backend database using SQL statements to retrieve and render requested information for users or to allow users to connect to the system. User input fields are vulnerable to SQL injection attack if input provided by the user is not sanitized before sending to the database engine for processing, which enables attackers to modify intended queries in order to perform malicious action in the system.

Injection through cookies: Cookies are structures that maintain persistence of web application by storing state information in the client machine. When a client returns to a Web application, cookies can be used to restore the client's state information. If a Web application uses the cookie's contents to build SQL queries, then an attacker can take this opportunity to modify cookies and submit to the database engine.

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 different ways, such as session 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 by entering malicious input into the client-end of the application or by crafting their own request to the server.

Second order injection: In second-order injections, attackers plant malicious inputs into a system or database to indirectly trigger an SQLIA. When that input is called at a later time when an attack occurs, the input that modifies the query to construe an attack does not come from the user, but from within the system itself.

2.2 Attack Intent

Attacks can also be characterized based on the goal, or intent, of the attacker. Therefore, each of the attack types that we provide in Section 2.4includes a list of one or more of the attack intentions defined in this section.

Identifying Injectable Parameters: Injectable parameters are text-input that allow users to request information from the database. This query request is sent to the database server though HTTP request, for example ULR, search box and authentication entries are considered as text-input. When these text-input are sending user requests to the database without proper validation they are considered as injectable parameters which allow attackers to inject SQL query attack. Identifying injection parameters is the first step to perform an attack.

Performing database fingerprinting**:** after identifying the injection parameters the second step is to know the database engine type and version. Knowing this is very important to an attacker because it enables him to know how to construct query format supported by that database engine and default vulnerability associated with that version as every database engine employs a different proprietary SQL language dialect.

Determine database schema: schema is the database structure. It includes table names, relationships, and column names. Knowing this information about database makes it easier for an attacker to construct an attack to perform database extraction or data manipulation language.

Database manipulation and extraction: this is dependent on the intent of the attacker as most attackers are more interested in getting customer bank information, creating bogus data modifying colleague salaries.

Evading detection: this is a method that attackers use to avoid detection by security mechanisms in the sense that their actions cannot be detected or traced.

Executing Remote Commands: Remote commands are executable code resident on the compromised database server. Remote command execution allows an attacker to run arbitrary programs on the server. Attacks with this type of intention could cause entire internal networks to be compromised.

Bypassing authentication: this is the most precious attack by attacker because it allows them to get access to the database with user privileges.

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.

2.3 Example of PHP Vulnerable Code

Before describing types of SQL injection attack, we present an example of PHP code that is vulnerable to SQL injection attack. We use this example in section 2.4 to provide attack examples.

```
$connection=mysql_connect('localhost','root','') or die('connection error');
 $dbselect= mysql_select_db('hr')or die('connection error');
$uname='msaidu';
$pword='123';
$query="SELECT * FROM login WHERE username='$uname'
 AND password='$pword' ";
$dquery=mysql_query($query);
$tquery=mysql_num_rows($dquery);
if($tquery==Null){
echo 'User Does not exist';
}
else{
echo 'You are now connected ';
}
$Get_Rows = GetRow(Run("SELECT * FROM emoloyee"));
$Get_Rows = GetRow(Run("SELECT * FROM salary"));
```

2.4 SQLIA Types

Attackers use SQLIA to attack web applications and this attack comes in various types depending on what attackers want to achieve. However according to adeghian *et al*. [23] SQLIA can be classified into (7) types: Tautologies, Illegal/Logically Incorrect Query, Union Query, Piggy-Backed Queries, Stored Procedures, Inference and Alternate Encodings.

1. Tautology attack: this is a type of attack that takes advantage of "WHERE" clause in SQL statement to evaluate the results returned by Query in relational database to be always true. Attackers use this type of attack to achieve authentication bypassing in web applications or perform unauthorized database extraction.

Authentication bypassing all relational database management system with no exception evaluate SQL query with "OR 1=1" where clause is always true. Also in relational database management system anything followed by comment (--) will not be processed. For example valid user's login to the system by using:

(SELECT * FROM login WHERE username=admin AND password=admin123);// ----------statement (1)
But attacker always finds the way out to bypass authentication to have access to the system and this can be achieve by:
(SELECT * FROM login WHERE username=admin or 1=1--- and password =nothing). ----------statement (2).

As you can see in statement 2 admin was used as a user or 1=1 meaning that connect admin or whoever exist in the system and comment (---) was used to ignore password which means password will not be processed.

Data extractions: most of the users in the system are only allowed to perform certain actions, for example, viewing their own profile details but malicious users or attackers always need more. Attackers try to access restricted data. An example would be a system designed for employees to view their own personal details. Thus employees can only view their profile nothing more. For example

(SELECT * FROM username, password UNION SELECT salary, card number FROM pay WHERE username=admin ----------
----------statement 1

This statement displays username, password, salary and credit card number of admin. However, as attackers are only interested in information that would help them in making money illegally, they might inject something like this:

(SELECT * FROM username, password UNION SELECT salary, cardnumber FROM pay WHERE username=admin OR 1'='1----------
----------statement 2

If you can see the "OR 1=1" transform the "WHERE" clause as true, at least one record that exists in database, if not all, will display the following details about employee: Username, password, salary and credit card number

2. Illegal or incorrect logical query: knowing the server, schema, table, and column names make it easy for attackers to gain unauthorized access to the system. For example http://localhost/?EmpId=10'.
If you notice at the end of the ULR quote was inserted after 10. This disturbs the database engine because when you type something within the quote it is used to tell the database that this is query and to process it. So after processing http://localhost/?EmpId=10' the database engine returns the following message saying:
You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '\" at line 1
telling the attacker the backend database engine used is MySQL and asking user to check the syntax in line one.

3. Inference Attack: this attack can be classified into Blind SQL injection and Timing Attack.

Blind SQL injection attack: this is another method of doing database fingerprinting. Sometimes database engines can be configured to hide database error messages and return generic error to the user when there is SQL syntax error in the users SQL statement. This can serve as a method to prevent attackers from database fingerprinting by using illegal or incorrect query method. However this does not mean the database is secure; it only conceals the return default error message which will be difficult for attackers who rely on database finger printing as a first step in carrying out an attack. Thus blind SQL injection attack can be used to deduce if there is a security mechanism implemented in the web application or not. Blind SQL injection attack can be achieved by asking a series of true or false queries in the database. In this case attacker tries to inject the following statements

SELECT * FROM emp_name, emp_address, gender, from employee where 1=0; drop employee ----------Statement (1)

SELECT * FROM emp_name, emp_address, gender, from employee where 1=1; drop employee ----------Statement (2)

After executing above Boolean malicious SQL query, an attacker will know if the database is secure or not. If the same response is delivered (return generic error message) there is protection mechanism that has detected an attack and blocked the query from executing and returned an error message to the user because all of the statement contains malicious words. A different response means that the query has reached inside the database engine and has been executed. Therefore the first query will return an error message because it is an incorrect query while the second may or may not return any error message because it was a correct query.

Timing Attacks: In this type of attack the response time the database takes to respond to users query is noticed which helps to know some information from a database. This method uses an if-then statement for injecting queries. WAITFOR is a keyword along the branches, which causes the database to delay its response by a specified time. For example an attacker can extract information from database using a vulnerable parameter.

declare @ varchar (8000) select @s = db_na
me () if (ascii (substring (@s, 1, 1)) & (power (2, 0))) > 0
waitfor delay '0:0:8

4. Union attack: this is most common type of attack used by attacker in gaining access to restricted data in other tables. Malicious SQL query can be appended by attacker to combine with valid SQL query in order to gain unauthorized access to extra data. For an example of a malicious attack, consider the following example where online human resources in a particular company allow employees to view only their personal details online. A malicious user can access extra information such as employee credit card number for example:

SELECT * FROM emp_name, emp_address, gender, from employee where emp_id=7;

The above statement displays employee name, address and gender with identification number 7. Below statement extracts more data about an employee:

SELECT * FROM emp_name, emp_address, gender, from employee where emp_id=7UNION SELECT credit_card FROM salary;

The above statement provides an attacker with employee details with the added bonus of the employee's credit card number.

Piggery-backend query attack: some of the database engines support stacked query by default. This feature creates opportunity for attacker to perform dangerous action in the database. In this case a valid query will be terminated by (;) and a malicious query is added. After processing the valid query, a malicious query is then executed, unlike in union query where a malicious query will be joined with a valid query and processed as a single joined query. For example:

SELECT * FROM emp_name, ep_address, gender, from employee where emp_id=7; drop employee
The above query will drop the employee table after displaying employee information.
Stored Procedure: Stored procedure is a part of database where programmers could set an extra abstract layer on the database as security to prevent SQL injection attack. As stored procedure could be coded by programmer, so, this part is known as an injectable web application. Depending on specific database storage procedure there are different ways to attack.

5. Alternate encoding: most of the SQL injection mechanisms that use filters prohibit the use of quote (') in the SQL statement which can be used in constructing different kinds of malicious query requests to the database. In this case for an attacker to bypass such a filter he has to convert SQL query into alternative encoding such as hexadecimal, ASCII or Unicode. Converting SQL query into alternate encode enables them to carry out their attacks. For example

"0; exec (0x73587574 64 5f177 6e), " and the result query is: SELECT accounts FROM login WHERE username='' AND password=0; exec (char (0x73687574646j776e))
The above example uses the char () function and ASCII hexadecimal encoding. The char () function takes hexadecimal encoding of character(s) and returns the actual character(s). The stream of numbers in the second part of the injection is ASCII hexadecimal encoding of the attack string. This encoded string is translated into the shutdown command by the database when it is executed.

6. Piggery-backend query attack: some of the database engines support stacked query by default. This feature creates opportunity for attacker to perform dangerous action in the database. In this case a valid query will be terminated by (;) and a malicious query is added. After processing the valid query, a malicious query is then executed, unlike in union query where a malicious query will be joined with a valid query and processed as a single joined query. For example:

SELECT * FROM emp_name, ep_address, gender, from employee where emp_id=7; drop employee
The above query will drop the employee table after displaying employee information.

7. Stored Procedure: Stored procedure is a part of database where programmers could set an extra abstract layer on the database as security to prevent SQL injection attack. As stored procedure could be coded by programmer, so, this part is known as an injectable web application. Depending on specific database storage procedure there are different ways to attack.

3.0 PROTECTION OF SQLIA

Researchers have proposed various methods to handle problems of SQL injection attack. These methods start from development of best practice to automatic tool for detecting and preventing SQL injection attack. Each method comes with its own advantages and drawbacks, but none of the methods were able to solve the entire problem of SQL injection attack.

3.1 Defensive Coding Practice

The main cause of SQL injection vulnerability is improper validation of user input. Input validation is a method by which programmers apply defense code practice to secure each static query manually. One of the objectives of defensive programming is to write secure queries so that it behaves in a predictable manner despite unexpected inputs or user actions. Below are some of the common ways by which programmer apply defensive coding in an application.

Input type checking/Data type validation: sometimes programmers make simple mistakes by allowing input fields to accept different types of data without realizing that an attacker can take this opportunity to insert malicious input to database engine. SQL injection attacks can be performed by injecting commands into either a string or numeric parameter. Even a simple check of such inputs can prevent many attacks. For example, in the case of numeric inputs, i.e. if the field is a phone number, the programmer can simply reject any input that contains characters other than digits. This method however cannot guaranty that it will fully stop the SQL injection but it makes the process harder for the attacker.

Encoding of inputs: sometimes an attacker issues SQL injection attack with a statement that always returns a value of true so that to the database engine interprets user input as SQL so that when backend database engine executes such a statement it allows the user to bypass authentication mechanisms or use meta-characters to perform an illegal query in order to trick the database engine into providing the attacker with some secret information about the backend database. Applying encoding practice such as hashing, encryption, conversion of input into ASCI format prevents attackers from tricking database engines.

White listening/Positive pattern matching: there are two primary concepts of pattern matching, blacklist and white list. Blacklisting involves checking if the input contains unacceptable data while white listing checks if the input contains acceptable data. Programmer should establish input validation routines that filter bad input and allow good input. This approach is generally called positive validation.

Identify all input: Parameterized query is a type of query which has some placeholders. In these queries instead of making dynamic queries by concatenating the parameters with SQL statement, it will replace the placeholders with the value of parameters at the runtime.

3.1.1 Disadvantages of Defensive Coding

Most programmers prefer to go straight ahead to secure their queries in application layer without realizing the effect when security mechanisms are exposed to attack or new queries are created in the future which force programmers to update each query manually. Below are the most common problems with defensive coding practice.

Defensive coding practice is prone to human errors; it is very difficult to exactly and correctly apply to all sources of input. In fact most vulnerability found in web application is because of programmer's mistake for not adding the security check or inappropriately applied [24].

White listing or blacklisting produce false positives especially when programmers decide to block SQL keywords such as "WHERE", "SELECT" or operators such as "AND", "OR" or single quote. This approach clearly results in a high rate of false positives because in many applications, SQL keywords can be part of a normal text entry, and SQL operators can be used to express formulas or even names (e.g., O'Brian, Randy, Orton).

Using stored procedures or prepared statements to prevent SQLIA: Unfortunately, stored procedures and prepared statements can also be vulnerable to SQLIA unless developers carefully apply defensive coding guidelines.

3.2 Detection and Prevention Methods

To overcome problems in defensive coding approach researcher proposed different methods to overcome the problem of defensive coding. This method can be categorized into static and dynamic methods.

In static method programmers try to prevent SQL injection attack by identifying SQL injection vulnerability by specifying all injection parameters even before the application is used for the first time [22]. This method is mostly language specific while dynamic method usually employs the use of a combination of two or more methods. In defensive coding the difference is that security mechanisms are standalone software tools that work by intercepting HTTP requests and doing intermediate validation. Sometimes we might have hybrid tool that exercise both static and dynamic approach, of which a good example is AMNESIA [7]. This section provides an inside to the strengths and weaknesses of existing SQLI prevention tools.

In [3] a method is proposed that uses static code analysis to detect SQL injection vulnerability and prevent attacker from exploiting such vulnerability in java web-based application. In this approach PQL was used as a syntactic model for queries library,

which allow users to define vulnerability patterns in a familiar Java-like syntax. Any piece of code that accepts input parameters from user and are passed to the backend database are marked tainted and tracked until it used in a sink. The advantage of this approach is that it enables detection of all potential security violations early, even without executing the application. However this approach requires source code application to carry out this function. In addition, it cannot detect unknown patterns of SQL injection attack.

R-WASP tool is proposed in [4] to detect and prevent SQL injection attack. The tool intercepts dynamic queries entered by the user and breaks them into tokens of SQL keywords, operators and characters in order to track existing malicious input in the query. If all input tokens are found to be trusted then the query is considered to be safe and allowed processing by database engine. Otherwise action is performed as defined by the programmer. Using SQL keywords, operators, and characters to find the malicious input in a dynamic query is problematic in nature, as it is possible to have a valid query with delete or drop keywords

DIGLOSSIA is method that prevents SQLIA by computing shadow values for the results of all string and character array operations that depend on user input [5]. In this approach programmer defines valid queries in the form of a tree-like structure to compare against dynamic queries entered by the user. When input query is sent to database the tool intercepts the query and tries to construct a tree like a dynamic query based on queries already defined by the programmer and also computes the shadow of the entered query storing it in the shadow value table indexed by the address of the memory location for the original value, performing both grammar and shadow checks using a dual parser. Using the dual parser to detect injected code is based on the idea that query strings can be parsed to either its original grammar, or the shadow grammar. If the tool cannot produce tree-like structure of query, the tool rejects the query and reports a code injection attack. Otherwise, it compares the query with its shadow to check whether the query is syntactically isomorphic, and that the code in the shadow query is not tainted. If either condition fails, it considers the dynamic query as an attack. The problem with this approach is that when users input non-malicious queries that are supported by database engine but violate the rule of query code in DIGLOSSIA it will consider that query as an attack. This method is totally based on the idea that when the web application submits the query any input type by the user will considered as an attack.

SQLUnitGen is proposed in [6]. The tool uses static analysis to detect and prevent SQL injection attack. It uses unit case that is library that lists a number of attack patterns which helps to detect existing SQL injections in a dynamic query. This method cannot detect new or existing attacks whose pattern has not been addressed in the unit test library.

Researchers in [7] have proposed AMNESIA that usesa combination of static analysis and runtime monitoring to detect and prevent SQL injection attack. In static phase, AMNESIA uses static analysis to build models of the different types of queries an application can legally generate at each point of access to the database. In its dynamic phase, AMNESIA intercepts all queries and checks to see if the query complies with model defined in the static phase. If the query matched the model it allows execution in the database engine, otherwise it is blocked. The accuracy of AMNISIA depends on the accuracy of the developed Queries model. The authors show in the evaluation that their method was capable of addressing all attacks.

Valeur and colleagues in [8] proposed intrusion detection that utilizes multiple anomaly detection models to detect attacks against back-end SQL databases based on machine learning approach. In this approach HTTP POST, and GET request are intercepted and IDS selects what features of the query should be modelled using training data set in training phase. This starts when feature vectors are created by extracting all tokens marked as constant and inserting them into a list in the order in which they appear in the query. After features were extracted then different statistical models are used depending on what data type model is selected. If a dynamic query does not match the model, the query will consider it as an attack. After evaluation IDS was found to be effective in detecting all kinds of SQL injection attack with false positive result.

WebSSARI is static-based methods that detect un-sanitized input parameters that result in injection vulnerability by monitoring the information flow [9]. In this approach, static analysis is used to check taint flows against preconditions for sensitive functions. The analysis detects the parameters by which preconditions have not been met and can sanitization functions that can be automatically added to the application to satisfy these preconditions. The WebSSARI system works by considering as sanitized an input parameter that has passed through a predefined set of filters. In their evaluation, the authors were able to detect security vulnerabilities in a range of existing applications. The drawback of this approach is adequate preconditions for sensitive functions cannot be accurately expressed, thus some filters may be omitted.

CANDIDis queries-model based method to detect and prevent SQL injection attacks [10]. In this approach dynamic queries are mined at runtime and compared with legitimate query statements in a model. If the result is not the same, it is a SQL injection attack. CANDID's natural and simple approach turns out to be very powerful for detection of SQL injection attacks.

SQLrand use key-based randomization of SQL instructions method to check for SQLIAs at runtime [11]. This enables programmers to develop queries using instruction randomization without using SQL

keywords. In this approach when attacker modifies the dynamic SQL query and sends to the database the proxy will intercept it and compare it with queries that the programmer created using instruction randomization which enablesSQLrand to detect malicious queries since dynamic queries were not created using instruction randomization. Experimental evaluation shows the effectiveness of this approach but this approach has a number of drawbacks. First, the security of the key may be compromised by looking at the error logs or messages. Furthermore, the approach imposes a significant infrastructure overhead because it requires the integration of an encryption proxy for the database.

Two methods similar SQL DOM [12] and Safe Query Object [13] provide efficient way to prevent SQLIA by changing query-development process. In these methods' queries to the database are decoded so as to prevent attacker from gaining unauthorized access to the database. These methods provide an effective way to prevent SQL injection problems by changing the query-building process from an unregulated one that uses string concatenation to a systematic one that uses a type-checked API. Within their API, they are able to systematically apply coding best practices such as input filter and rigorous type checking of user input. By changing the development paradigm in which SQL queries are created, these methods eliminate the coding practices that make most SQLIAs possible. Although effective, these methods have the drawback that they require developers to learn and use a new programming paradigm or query-development process. Furthermore, because they focus on using a new development process, they do not provide any type of protection or improved security for existing legacy systems.

Security Gateway is proposed in [14] which intercept dynamic queries in order to enforce the specified policy. The security gateway acts as an application level firewall; its job is to intercept, analyze and transform all HTTP messages as well as checking HTTP requests, After analyzing the HTTP message it then rewrites it in HTML in HTTP responses, annotating it with Message Authentication Codes (MACs) in order to protect the query request which may have been maliciously modified by clients. This method is very effective in identifying modified dynamic queries, however this approach is human-based and, like defensive programming, requires developers to know not only which data needs to be filtered but also what patterns and filters to apply to the data.

SQLProb is a model-based that combines both static and dynamic analysis to detect and prevent SQLIA [15]. In static phase query a collector was used to generate parse tree structure of legitimate queries from query repository as defined by the programmer which will be used to compare semantic structure of dynamic query. However, in dynamic phase when user inputs queries, the queries are compared against semantic tree structure of legitimate queries created in phase and if the structure of dynamic query matches with the structure in a generated tree like structure query, queries are allowed; otherwise they are prevented and consider as malicious. This method employs a similar approach used by [5] and the accuracy of this approach depends on how accurate was the parser tree model that was developed.

Similarly SQLGuard uses queries-model based method to check if dynamic query conform model of legitimate queries [16]. In this methods input queries dynamically generate, through concatenation, a string representing an SQL statement and incorporating user input which generates and returns a new key by the database. SQLGuard validates dynamic queries by building two parse tree structures of dynamic query. First tree structure has unpopulated user tokens for dynamic query the second tree is the result of parsing the string with these nodes filled in with user input. The two trees are then compared for matching structure. If the structure marched, the query is allowed for execution; otherwise it is blocked. This approach tends to be slow as data comparison takes much time to process in tree structure model as each node must be processed. The accuracy of this approach depends on whether or not the attacker discovered the key.

In [17] machine learning method is proposed that uses Bayesian algorithm to detect and prevent SQL injection attack. In this approach monitor capture dynamic SQL query HTTP POST and GET, send it to converter which breaks SQL statement into a number of keywords based on black space in statement and calculate the length of dynamic SQL query. It also calculates the number of keywords present in such a query and sends a numeric value of length and keyword of dynamic query to the classifier. The classifier then calculates the probability of SQL injection in dynamic query based on results received from the converter, and then compares the probability of SQL injection calculated with one defined by user threshold as training dataset which consists of the probability of legitimate query and probability of malicious query. When the probability of dynamic SQL query calculated by classifier matches the probability of legitimate query in training dataset the query is allowed; otherwise it is blocked. One important thing in this method is that it simulates a high number of attack patterns in training data including blind SQL injection attack which is very difficult to address. However this method requires programmers to fully define and carefully train data set because the accuracy of this approach depends on how accurate was the trained data.

Similarly proposed method in [18] uses machine learning to detect and prevent SQL injection attack. In this method training dataset was constructed by analyzing source code program of the application and calculating the entropy of static SQL query. The

main purpose of entropy is to count the average amount of information needed to identify the class model of a training dataset. In this case entropy of all static queries that are implemented in a website was calculated which was used to construct training dataset which will be used later for comparison. When a user issues SQL query the entropy of dynamic SQL query is calculated and compared with entropy in training data. If a match is found the query is allowed to execute in database engine; otherwise it is blocked and prevented from parsing to the database engine. Using entropy in machine learning to classify queries has advantages over using probability as used because it produced better results. When data are categorized instead of using continuous-valued, small changes in SQL query will yield a great effect when the entropy of that query is calculated. The disadvantage of this method is that it requires analysis of the application of the source code.

Hossain Shahriar and Mohammad Zulkernine in [19] proposed method that uses anomaly-based method to detect and prevent SQL injection attack. In this approach black space method was used in breaking SQL stamen into keywords but here length of the query was not considered. After tokenizing SQL statement user action was then considered in generating training dataset in which system user was categorized into three namely visiting user, normal user and admin and a different role was assigned to each. This user classification helps classifier to determine which model to use in training data to compute and compare probability of SQL injection in user query. For example in normal user query keywords considered as malicious are dropped so when visiting users issue SQL statements with drop keywords this query will be automatically considered as malicious before computing probability which allows the method to use two probabilities in computing user queries. The first is prior probability which assumes the query is malicious if it contains some malicious keywords and posterior probability which can be obtained after comparing calculated probability of dynamic query against its model in training. Advantage of using prior probability and posterior probability is that they help to reduce false positive result. However the issue of stacked query was not addressed in this method which allowed attackers to perform piggy backend query attack.

In [20] pattern matching method is proposed to detect and prevent cross-site scripting (XSS) and SQL injection attack. In this approach programmer created files which contain attack patterns of both XSS and SQL injection attack. HTTP request to database will be intercepted and compared to dynamic query with set of attack patterns in programmer define threshold. If the query is found to contain attack patterns defined by the programmer the query will be blocked and a report is generated. This method was found to be effective after evaluation; however it cannot guarantee protection for attack patterns that were not included in the programmer predefined threshold.

4.0 EVALUATION

In this section, we evaluate tools and methods considered with respect to deployment requirements, attack types, injection parameters, defensive coding practice, and evaluation parameters. Similar evaluation was done in [21] but authors did their evaluations based on only 3 categories of the 5 done in this paper.

4.1 Evaluation based on Attack Type

We analysed and evaluated each proposed method as shown in Table 1. To ensure particular tool or method is capable of addressing a particular attack we used analytical evaluation based on our experience. We have not assessed any of the tool or method in real time practice for the reason that most tool or method's implementation codes are not available or some methods are not implemented.

4.2 Evaluation with Respect to Injection Parameters

We analyzed each of the methods with respect to their handling of the various injection mechanisms described (See Section 2.1.) We used "yes" to indicate parameter and "no" to indicate that the tool cannot address that parameter in Table 2.

4.3 Evaluation based on Author's Evaluation Parameters

We have also evaluated and analyzed each proposed method with respect to parameters such as efficiency, effectiveness, flexibility, performance and stability. Table 3 shows the summary of the evaluation that helps in choosing an appropriate tool.

For example the Table 3 shows which programming language could be supported by the specific tool. Also, by flexibility, types of SQL injection attack which are addressed by that tool could be identified. In flexibility "All/s" means that the tool can stop all types of attack successfully and "All/p" means that the tool can stop all the attack types partially, and "number" indicates the number of attacks prevented by the tool. In effectiveness we used percentage effectives mentioned by author after testing method ranging from 100- 93.3. In efficiency we used F.P to represent false positive report by the tool and F.N to represent false negative.

In programming we specifically mentioned particular languages supported by the tool and we also used Comport/all to indicate tool support in any programming environment. In performance we used words like "noticeable" to indicate that the tool take a longer time to do processing, "unnoticeable" to

indicate it took a longer time but the user did not mind, "negligible" to indicate it took time but not noticeably, efficient to indicate small difference in query processing time when such tools were implanted and when they are not, and we used quit efficient to indicate the best tool that take less time.

4.4 Evaluation Based on Deployment Requirements

We analysed each method based on the criteria as shown below in Table 4. We used "yes" to indicate that the tool requires particular criteria and "no" to indicate that criteria is not required for that particular tool. We evaluated each method with respect to the following criteria:

1. Code modification: Most of the developed security mechanisms are based on user defined threshold database. For example trained dataset, anomaly detection model or predefined library pattern.
2. Pattern matching: sometimes programmers prefer to design security mechanisms based on accept and don'ts accept pattern similarities of user input.
3. Application automation: sometime administrators are required to do some jobs manually. In such cases the security mechanism does not perform decisions of prevention or adding abnormal behaviour in anomaly detection model but rather alert the administrator to make decisions manually when certain abnormal behaviour is encountered. This could not always be what the administrator needs; sometimes it could be a limitation of the security to address the scope of security requirement.
4. Additional infrastructure: some security mechanisms rely on infrastructure in order to accomplish their action and this may not be necessary.
5. Tokenization of input: Tokenization is the process of breaking the query into meaningful elements called tokens. The list of tokens becomes inputs for further processing which is classification.

4.5 Evaluation based on Common Development Errors

We have analysed and classified protection methods with respect to the defensive coding practice as described in Section 3.1. Table 5 shows summary of this evaluation.

Table 1 Evaluation based on attack type

Approach	Tautology	Illegal/in correct	Piggy-backend	Inference	Alternate encode	Stored procedure	Union
R-WASP[4]	•	•	•	x	•	•	•
DIGLOSSIA[5]	-	-	-	-	-	-	-
SQLUnitGen[6]	•	•	•	x	•	X	•
AMNESIA[7]	•	•	•	•	•	X	•
IDS[8]	○	○	○	○	○	○	○
WebSSARI[9]	•	•	•	•	•	•	•
CANDID[10]	○	○	○	○	○	○	○
SQLrand[11]	•	X	•	•	x	X	•
DOM[12]	•	•	•	•	•	X	•
SafeQuery[13]	•	•	•	•	•	X	•
Security Gateway[14]	-	-	-	-	-	-	-
SQLProb[15]	○	○	○	○	○	○	○

Hong Cheon, *et al.* [17]	-	-	-	-	-	X	-
Joshi *et al*. [18]	◦	◦	◦	◦	◦	X	-
Hossain Shahriar and Mohammad Zulkernine [19]	◦	X	◦	◦	◦	X	◦
Puspendra Kumar, R. K. Pateriya [20]	◦	◦	◦	◦	◦	X	◦

"•" indicates that a method can successfully stop all attacks of that type.
"×" indicates that a method was not able to stop all attacks of that type.
"◦" indicates that a method can address the attack type considered, but cannot provide any guarantee of completeness.
"-"indicates that a method can address the attack type considered, but cannot provide guarantee of completeness due to limitation of it underline approach.

Out of seventeen (17) tools and methods considered, six(6) of them, DIGLOSSIA [5],IDS [8], WebSSARI [9], CANDID [10], Security Gateway [14] and SQLProb [15] focus on addressing all types of SQLIAs considered. However the effectiveness of the tools and methods considered for addressing particular types of SQLIAs considered varies depending on approach used, in developing tool or method, and its ability to be deployed in various injection parameters described,(See Section 2.1).
To describe the effectiveness of the methods four symbols were used in Table 1.

The tick dot symbol ("•") as can be seen in Table 1 was used for R-WASP [4], SQLUnitGen [6], AMNESIA [7], WebSSARI [9] SQLrand [11], DOM [12] and SafeQuery [13].This indicates that these methods or tools can guarantee protection of particular SQLIAs type considered. However, out of these methods only WebSSARI [9] was able to successfully prevent all types of SQLIAs considered because of its deployment in various injection parameters, (See Table 2).
The "◦" and "-" are used in Table 1 to indicate that method or tool can partially detect and prevent SQLIAs type considered without guaranteeing that a given methods will prevent future attack of similar type addressed.

We used ("◦") for methods that implement anomaly or machine learning based approach to detect and prevent SQLIAs. The reason is that these approaches use sets of typical application queries as input data set to train the protection model, thus any query that goes against the model might result in false positive or false negative. IDS [8] and methods such Joshi *et al*. [18], Hossain Shahriar and Mohammad Zulkernine [19] and Puspendra Kumar, R. K. Pateriya [20] use trained queries models to monitor the application at runtime to identify dynamic queries that do not match the models. The effectiveness of these methods is highly dependent on quality of training data set used and how good the model was trained, as poor training data set and model result in false positive and negative. Thus, the effectiveness of methods and tools implementing these approaches is considered partial using circle ("◦") symbol as shown on Table 1.

Other methods considered as partial are methods that use errors related problem approach to detect SQLIAs as errors related problem is only one of the many possible causes of SQL injection vulnerability. DIGLOSSIA [5], SQLGuard [16] and Hong Cheon, *et al*. [17] are methods that implement this approach. We also considered Security Gateway [14] to be in same category because it works based on Security Policy Descriptor Language (SPDL), allowing the programmer to set the constraints, rules and to specify how transformation should be applied to injection parameters as dynamic queries flowing from client to side to database server. This approach not only required programmer to know which data to be filtered but also how filter pattern should be applied. This means that poor filter results in false positive and false negative.

In summary information in Table 1 shows that stored procedure attack seems to be a difficult attack to be addressed by many of the tools and methods considered this because the code that generates the query is stored and executed on the database and most of the methods considered focus on preventing attack on queries that are generated with applications. Based on our analysis WebSSARI [9] is considered to be the best tool for preventing SQLIAs. However it is important to note that we did not take precision into account in our evaluation, that is to say many of methods and tools considered are based on conservative analysis that may result in false positive.

Information in Table 2 shows that none of the tools or methods considered can be deployed to detect or prevent attack that exploits second order SQLI vulnerability. This is due to the fact that second order SQLIV is not a problem of sanitizing sensitive function

but is intentionally created by attackers through vulnerable parts of the application (not necessarily through Login. Add user page or ULR attacker may also use file inclusion attack to exploit dynamic file include) and reside in application database.

It is important to note that only three tools can be deployed on server side. One of these tools is Security Gateway [14] because it uses filters to interpret a query string in the same way that the database would. Other methods that can be deployed well in server side are developer based methods which are DIGLOSSIA [5] and WebSSARI [9].

It is also important to note that half of the methods considered can examine queries in cookie fields and all methods considered can examine login search and ULR with the exception of Hossain Shahriar and Mohammad Zulkernine [19] and Puspendra Kumar, R. K. Pateriya [20] which are not capable of preventing authentication bypassing.

Likewise information shows that none of the tools or methods was able to prevent attacks that exploit second order SQLI vulnerability which is also one of the trends for current SQLI vulnerability scanners, both black and white box.

Information in Table 3 shows that most of the tools and methods considered do not have False alarm problems after being evaluated by original author(s) with the exception of WebSSARI [9] and IDS[8] which report 10.3% false positive, no false negative and 0.06% false positive, no false negative respectively using metrics called accuracy.

It is important to note that none of the tools considered detect and prevent less than 95% of SQLIAs considered. However, tools that achieved 100% effectiveness are either language specific or are designed to prevent certain number of SQLIAs considered. For example, R-WASP[4] achieved 100% effectiveness but has only been designed to prevent six (6) SQLIAs out of seven(7) considered and designed to only work correctly on .Net framework applications. Similarly, SQLProb [15] achieved 100% effectiveness and was able to prevent all SQLIAs considered partially but may not work or achieve 100% effectiveness when deployed in other applications such as PHP based, or .Net framework applications rather than Java based applications.

It is also important to note that average performance of tools and methods considered is negligible and this performance may or may not change with change of equipment for evaluation. This evaluation is based on what original author(s) mentioned in the paper after the method or tool has been evaluated. Thus, evaluation result may or may not change when re-evaluated experimentally by different researchers.

Similarly information shows that that seven (7) out of seventeen tools and methods considered use pattern matching methods to detect SQLIAs which is similar to blacklist or white list, While eight (8) perform input type checking at runtime.

In general all of the SQLI prevention tools and methods considered are developed using one or more defensive coding approaches described (See section 3.1).Except DIGLOSSIA [5] and Hong Cheon *et al*. [17] which work by scanning source code of application and applying taint value to the function that may give room to SQLIA.

Information in Table 4 indicates that most of the tools and methods considered do not require the programmer to modify the queries model when a new page is inserted in an application. However tools such as SQLUnitGen [6], AMNESIA [7], required the programmer to modify queries model due to the fact that these tools use static approach to build legal models of different types of queries an application can access for any access to database. Methods that use static approach to build models of different queries require the programmer to either rewrite code to use a special intermediate library to work with model or manually insert new query(s) into a model so that user input will be considered as a dynamically generated query. Similarly, methods such as Hossain Shahriar and Mohammad Zulkernine [19] and Puspendra Kumar, R. K. Pateriya [20] required trained data model to be modified when new page is added in an application. This is because these methods are based on anomaly and machine learning approaches that use specific inputs as training data set (specific SQL keywords, clauses and operators) to train its queries model. Therefore, these approaches also require the programmer to retrain queries model when a new page is added with different inputs from previous inputs.

Similarly Information on Table 4 shows that most of the tools and methods considered are fully automated in terms of detecting SQLIAs with the exception of Security Gateway [14], DOM [12], and SafeQuery [13]. We considered Security Gateway [14] as manually specifically because it is based on proxy filter that allows users to set constraints on web client access to database and this filter is apply manually defending on the purpose of the application. DOM [12], SafeQuery [13] are not available (N/A) and as result it was not mentioned by authors and we were not able to predict its degree of automation.

For prevention, eleven out of seventeen (17) reviewed tools and methods carry out prevention automatically. Two (2) are semi-automatic. One is WebSSARI [9] which works by tracing taint flow against precondition for sensitive function in application source code and provides suggestion filter and sanitization function that can be added automatically to meet precondition requirements. The second one is Joshi *et al.* [18], while four (4) other methods generate report/alarm to enable the administrator to make decisions. For example Puspendra Kumar, R. K. Pateriya [20] uses anomaly detection method to statically build anomaly pattern and score of static queries which will later be compared against anomaly pattern and score of dynamic queries requested by users. If the anomaly pattern of dynamic query matched with anomaly pattern of static query, the query is considered as an

attack. Otherwise if anomaly score reached a certain percentage, an alarm should be sent to administrator to analyze the query manually.

Half of the tools and methods considered use tokenization approach i.e break down dynamic query into keywords, operators and clauses and use this for further processing) while other half don't.

In case of additional infrastructure most of the tools and methods require programmer training or training data while few of them do not require any additional infrastructure at all. More parameter such as infrastructure or tokenization imply more processing resources, therefore application performance can be affected by having more infrastructure and tokenization activities.

Thus, among the tools and methods considered DIGLOSSIA [5] is the best tool that do not consumes much processing power or require programmer intervention in combating with SQLIAs.

Table 2 Evaluation based on injection parameters

Approach	URL	Login	Search	Cookies	Server Side	Second Order
R-WASP [4]	Yes	Yes	Yes	Yes	No	No
DIGLOSSIA [5]	Yes	Yes	Yes	Yes	Yes	No
SQLUnitGen [6]	Yes	Yes	Yes	No	No	No
AMNESIA [7]	Yes	Yes	Yes	No	No	No
IDS [8]	Yes	Yes	Yes	Yes	No	No
WebSSARI [9]	Yes	Yes	Yes	Yes	Yes	No
CANDID [10]	Yes	Yes	Yes	Yes	No	No
SQLrand [11]	Yes	Yes	Yes	No	No	No
DOM [12]	Yes	Yes	Yes	No	No	No
SafeQuery [13]	Yes	Yes	Yes	No	No	No
Security Gateway [14]	Yes	Yes	Yes	Yes	Yes	No
SQLProb [15]	Yes	Yes	Yes	Yes	No	No
SQLGuard [16]	Yes	Yes	Yes	No	No	No
Hong Cheon, *et al.* [17]	Yes	Yes	Yes	No	No	No
Joshi *et al.* [18]	Yes	Yes	Yes	No	No	No
Hossain Shahriar and Mohammad Zulkernine [19]	Yes	Yes	Yes	No	No	No
Puspendra Kumar, R. K. Pateriya [20]	Yes	No	Yes	Yes	No	No

"Yes" indicates that tool or method can prevent attack or be deployed in that injection parameter
"No" indicate that tool or method cannot prevent attack or be deployed in that injection parameter

Table 3 Evaluation with based on Author's Evaluation Parameters

Approach	Efficiency	Effectiveness	Flexibility	Stability		Performance
				Programming	Equipment for evaluation	
R-WASP[4]	F.P=0, F.N=0	100%	6	.Net Framework	Windows XP machine, Pentium 40GB, 1GB RAM , MySQL server, IIS 7.0	N/A
DIGLOSSIA [5]	F.P=0, F.N=0	95%-96.1%	All/P	Comport/ All	Intel(R) dual core3:30 GHz machine with 8G of RAM.	Unnoticeable
SQLUnitGen [6]	F.P=0, F.N=0	100%	5	Java	N/A	N/A
AMNESIA [7]	F.P=0, F.N=0	100%	6	Java	N/A	Negligible
IDS [8]	F.P=0.06%, F.N=0	N/A	All/P	Comport/ All	Server: 2GHz Pentium 4 1GB of RAM Linux.2.6.1 Apache web server(v2.0.52), the MySQL database(v4.1.8), and PHP-Nuke(v7.5).	Negligible
WebSSARI [9]	F.P=10.3%, F.N=0	N/A	All/S	Comport/ All	N/A	N/A
CANDID [10]	F.P=0, F.N=0 N/A	100%	All/P	Java	Client: 2GHz Pentium processor and 2GB RAM Server: a Red Hat Enterprise GNU/Linux Machine	Noticeable
SQLrand [11]	N/A	N/A	4	Java	Client: x86 machines, Server: RedHat Linux,	Negligible
DOM [12]	N/A	N/A	5	.NET Framework	N/A	Unnoticeable
SafeQuery Objects [13]	F.P=0, F.N=0	N/A	5	Java	N/A	Efficient
Security Gateway [14]	N/A	99%	All/P	Comport/ All	Server: AMD Opteron 150 machine, 4GB RAM Linux. Client: 2 GHz AMD Athlon XP, 256MB RAM, Linux.	Quite efficient
SQLProb [15]	N/A	95%	All/P	Java	Virtual Machine with 1 GB RAM , Fedora 9. MySQL 5.0.27 database server	Noticeable
SQLGuard[16]	F.P=0, F.N=0	N/A	6	J2EE	Web server: Windows 2000 machine 733MHz, 256MB RAM.	Efficient

"All/S" means that the tool can successfully stop all types of attack
"All/p" means that the tool can partially stop all the attack types.
"Digital number was used" to indicate numbers of attack type's tool can prevent.
"F.P" represents false positive report by the tool.
"F.N" represent false negative reported by the tool
"N/A" indicates the parameter considered is not available.

Table 4 Evaluation Based on Deployment Requirements

Approach	Modify code based	Detection	Prevention	Additional Infrastructure	Input tokenization
R-WASP [4]	No	Automatic	Generate Alarm	.NET, MSIL	Yes
DIGLOSSIA [5]	No	Automatic	Automatic	No	No
SQLUnitGen [6]	Yes	Automatic	Automatic	Unit Testcase	No
AMNESIA [7]	Yes	Automatic	Automatic	No	No
IDS [8]	No	Automatic	Generate Report	Training Data	Yes
WebSSARI [9]	No	Automatic	Semi-Automatic	No	Yes
CANDID [10]	No	Automatic	Automatic	No	Yes
SQLrand [11]	No	Automatic	Automatic	Proxy, programmer Training, Key Management	Yes
DOM [12]	No	N/A	Automatic	Training Data	No
SafeQuery [13]	No	N/A	Automatic	Training Data	No
Security Gateway [14]	No	Manually Specified	Automatic	Training Data	No
SQLProb [15]	No	Automatic	Automatic	Proxy Filter	No
SQLGuard [16]	No	Automatic	Automatic	Training Data Proxy	Yes
Hong Cheon, *et al.* [17]	No	Automatic	Automatic	Key Management	Yes
Joshi *et al*. [18]	No	Automatic	Semi-Automatic	Training Data	No
Hossain Shahriar and Mohammad Zulkernine [19]	No	Automatic	Automatic	Training Data	Yes
Puspendra Kumar, R. K. Pateriya [20]	Yes	Automatic	Generate Report	No	No

"Yes" means particular criteria is required for the tool to effectively detect malicious input or to perform detection and prevention using the tool.
"No "means particular criteria is not required by the tool to effectively detect malicious input or to perform detection and prevention using the tool.

Table 5 Evaluation with based on Evaluation Parameters

Approach	Input type checking	Encoding of input	Identification of input source	Positive pattern matching
R-WASP [4]	Yes	No	Yes	Yes
DIGLOSSIA [5]	No	Yes	Yes	No
SQLUnitGen [6]	Yes	No	Yes	Yes
AMNESIA [7]	Yes	Yes	Yes	No
IDS [8]	Yes	No	Yes	No
WebSSARI [9]	Yes	Yes	Yes	Yes
CANDID [10]	Yes	Yes	Yes	No
SQLrand [11]	Yes	Yes	N/A	Yes
DOM [12]	Yes	Yes	Yes	No
SafeQuery [13]	Yes	Yes	Yes	No
Security Gateway [14]	Yes	Yes	No	Yes
SQLProb [15]	Yes	No	Yes	No
SQLGuard [16]	Yes	Yes	No	No
Hong Cheon, *et al.* [17]	No	Yes	No	No
Joshi *et al.* [18]	Yes	No	No	No
Hossain Shahriar and Mohammad Zulkernine [19]	Yes	No	No	No
Puspendra Kumar, R. K. Pateriya [20]	Yes	No	No	Yes

"Yes" means particular criteria is required for the tool to effectively detect malicious input or to perform detection and prevention using the tool.
"No "means particular criteria is not required by the tool to effectively detect malicious input or to perform detection and prevention using the tool.
"N/A" means we were not able to identify particular tool criteria.

Information in Table 5 shows that most of the methods and tools perform input type checking due to the fact that many of the prevention methods are actually applying one or more defensive coding best practices to the code base. This is in contrast to DIGLOSSIA [5] and Hong Cheon *et al*. [17] which work by scanning source code of application and applying taint value to the function that may give room to SQLIA. It is important to note that seven (7) out of seventeen tools and methods considered use pattern matching methods to detect SQLIAs which is similar to blacklist or white list as described in (section 3.1). In general most of the SQLI prevention tools and methods considered are developed using one or more defensive coding approaches described in section 3.1.

5.0 CONCLUSION AND FUTURE WORK

In this paper we present background of SQLIA, highlight defensive coding practice and its disadvantages, provide a review of different tools proposed by different researchers and analysis those tools based on different perspectives such as attack types, common development errors, deployment requirements and checking input and author evaluation.

Our evaluation result show common trends on current SQLI prevention measure. Only a few of these methods and tools can be deployed in all web-based application platforms and Most of these methods and tools have problems addressing store-procedure attacks, as well as problems addressing attacks that take advantage of second order SQLI vulnerability

Future work should focus on evaluating the methods in a real time practical application.

Acknowledgement

This research was fully supported by Universiti Teknologi Malaysia Johor Bahru, Malaysia. Authors would like to acknowledge Faculty of Computing for supporting this work.

References

[1] Tudor, J. 2013. Web Applications Vulnerability statistic 2013. [Online] From; http://sitic.org/wp-content/uploads/Web-Application-Vulnerability-Statistics-2013.pdf [accessed on 17 September 2015].

[2] OWASPD-Open Web Application Security Project 2014. Top ten most critical Web Application Security Risks. [Online]. From; http://cwe.mitre.org/cwss/archive.htm [accessed on 17 September 2015).

[3] Livshits, V., and M. S. Lam 2005. Finding Security Errors in Java Programs with Static Analysis. Technical Report.

[4] Medhane, M. 2013. R-WASP: Real Time-Web Application SQL Injection Detector and Preventer. *International Journal of Innovative Technology and Exploring Engineering (IJITEE).* 2(5): 327-330. ISSN: 2278-3075.

[5] Son, S., k. S. McKinley, and V. Shmatikov. 2013. Diglossia: Detecting Code Injection Attacks with Precision and Efficiency. *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*. 1181-1192.

[6] Shin, Y., L. Williams, and T. Xie 2006. SQLUnitgenTest Case Generation for SQL Injection Detection. North Carolina State University, Raleigh Technical report, NCSU CSC TR.

[7] G.J. Halfond, W., and A. Orso. 2005. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*. 22-28.

[8] Valeur, F., and G. Vigna 2005. A Learning-based Approach to the Detection of SQL Attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer Berlin Heidelberg. 3548: 123-140.

[9] Nguyen-Tuong, A., S. Guarnierie, J. Shirley and D. Evans 2005. *Automatically Hardening Web Applications Using Precise Tainting*. Springer US. 181: 295-307.

[10] Bandhakavi, S., P. Bisht, and P. Madhusudan 2007. CANDID: Preventing Sql Injection Attacks Using Dynamic Candidate Evaluations. *Proceedings of the 14th ACM conference on Computer and Communications Security*. 12-24.

[11] W. Boyd, S., W., A. D. Keromytis. 2005. SQLrand: Preventing SQL Injection Attacks. *Applied Cryptography and Network Security*. Springer Berlin Heidelberg. 3089: 292-302.

[12] McClure, R., and I. H. Kruger. 2005. SQL DOM: Compile Time Checking of Dynamic SQL Statements. *27th IEEE International Conference on Software* Engineering. St. Louis, USAS, 18-21 May 2005. 88-96.

[13] R. Cook, W., and S. Rai. 2005. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. *27th IEEE International Conference on Software* Engineering. St. Louis, USAS, 18-21 May 2005. 97-106.

[14] Scott, D., and R. Sharp. 2002. Abstracting Application-Level Web Security. *Proceedings of the 11th international conference on World Wide Web*. Hawaii, USA, 7-11 May 2006. 396-407.

[15] Liu, A., Y. Yaun, and D. Wijesekera 2009. SQLProb: A Proxy-Based Architecture Towards Preventing SQL Injection Attacks. *Proceedings of the 2009 ACM Symposium on Applied Computing*. ACM. 2054-2061.

[16] Buehrer, G., B. W. Weide, and P. Sivilotti 2005. Using Parse Tree Validation to Prevent SQLinjection Attacks. *5th International Workshop on Software Engineering and Middleware*. 106-113.

[17] Cheon, E., Z. Huang, and Y. Lee 2013. Preventing SQL Injection Attack Based on Machine Learning. *International Journal of Advancements in Computing Technology*. 5(9): 967-974.

[18] Joshi, A., and G. V. 2014. SQL Injection Detection Using Machine Learning. *IEEE International Conference on Control, Instrumentation, Communication and Computational Technologies*. 1111-1115.

[19] Shahriar, H., and M. Zulkernine. 2012. Information-theoretic Detection of SQL Injection Attacks. *IEEE 14th International Symposium on High-Assurance Systems Engineering. Miami, USA. 9-11 January 2014.* 40-47.

[20] Kumar, P., and R. Pateriya. 2013. Enhanced Intrusion Detection System for Input Validation Attacks in Web Application. *International Journal of Computer Science Issues (IJCSI)*. 10(1): 435-437.

[21] Tajpour, A., S. Ibrahim, and M. Sharifi. 2012. Web Application Security by SQL Injection Detection Tools. *IJCSI International Journal of Computer Science Issues (IJSCI)*. 9(2): 332-339.

[22] Xin-hua, Z., and W. Zhi-Jian. 2010. Notice of Retraction A Static Analysis Tool for Detecting Web Application Injection Vulnerabilities for ASP Program. *2nd IEEE International Conference on e-Business and Information System Security.* Wuhan, China. 22-23 May 2010. 1-5.

[23] Sadeghian, A., M. Zamani, and A. Manaf. 2013. Taxonomy of SQL Injection Detection and Prevention Methods. *IEEE*

International Conference on Informatics and Creative Multimedia. Kuala Lumpur, Malaysia. 4-6 September. 53-56.

[24] G. J. Halfond, W., J. Viegas and A. Orso. 2006. Classification of SQL Injection Attacks and Countermeasure.· *IEEE International Symposium on Secure Software Engineering. Washington DC, USA, 13-15 March 2006.* 87-96.