# A Survey Of Sql Injection Countermeasures

Dr R.P.Mahapatra and Mrs Subi Khan
Department of Computer Engineering, SRM University, Modinagar, India
Mahapatra.rp@gmail.com
Department of Computer Engineering, SRM University, Modinagar, India
*Subikhan30@gmail.com*

***Abstract:***
*SQL injection has become a predominant type of attacks that target web applications. It allows attackers to obtain unauthorized access to the back-end database to change the intended application-generated SQL queries. Researchers have proposed various solutions to address SQL injection problems. However, many of them have limitations and often cannot address all kinds of injection problems. What's more, new types of SQL injection attacks have arisen over the years. To better counter these attacks, identifying and understanding existing countermeasures are very important. In this research , I had surveyed existing techniques against SQL injection attacks and analyzed their advantages and disadvantages. In addition, I identified techniques for building secure systems and applied them to my applications and database system, and illustrated how they were performed and the effect of them.*

***Keywords:*** *SQL injection attacks, Database, Authentication Bypass, Secure Systems*

## 1.Introduction

Nowadays, web applications are prevalent around the world. More and more companies and organizations use web applications to provide various services to users. Web applications receive users' requests, interact with the back-end database, and return relevant data for users. The back-end database often contains confidential and sensitive information such as financial or medical data that interest attackers. SQL injection is one of the major techniques attackers use to compromise a database. This type of attack exploits vulnerabilities existing in web applications or stored procedures in the back-end database server. It allows attackers to inject crafted malicious SQL query segments to change the intended effect of a SQL query, so that attackers can obtain unauthorized access to a database, read or modify data, make the data unavailable to other users, or even corrupt the database server. According to a survey report released in 2010 by the IBM X-Force® research and development team, the number of SQL injection attacks has increased rapidly in recent years, and SQL injection has become the predominant type of attacks that target web applications. During the first half of the year of 2010, the average number of daily SQL injection attacks around the world is about 400,000 [1].Web applications and their underlying databases require not only careful configuration and programming to assure security, but also effective protection mechanisms to prevent attacks. Researchers have proposed various solutions and techniques to address the SQL injection problems. However, there is no one solution that can guarantee complete safety. Many current solutions often cannot address all of the problems. For example, many techniques proposed are based on the assumption that only the SQL statements that receive user input are vulnerable to SQL injection attacks. However, there is a new type of attack called Lateral SQL injection, which does not require a vulnerable SQL statement to have user input parameters. A comprehensive survey can help developers and researchers better understand the various forms of SQL injection attacks, as well as the strengths and weaknesses of excising countermeasures for the attacks.

This research will presents a survey of SQL injection attacks and various techniques used to counter them.

## 2.Building Secure Applications and Database Systems

Researchers have proposed many approaches to address the problem of SQL injection attacks. They can be roughly divided into two categories: techniques about how to build secure applications and database systems (e.g. defensive programming practices), and techniques focus on how to protect an existing system (e.g. intrusion detection systems). Techniques from only one category are usually not good enough to ensure system safety. To guard a system against SQL injection, we need techniques from both categories. This chapter will present the techniques for building secure applications and database systems.

### 2.1 Secure Applications against SQL Injection Attacks

In SQLIA Hackers can submit malicious user input to a vulnerable web application to conduct attacks. The best way to prevent SQL injection vulnerabilities is to write secure programs.

#### 2.1.1 Defensive Programming Techniques

Many SQL injection attacks are easy to defend against with defensive programming practices. To prevent SQL injection vulnerabilities in an application, different programming languages have their specific approaches. Prepared statement is a very effective technique used by many programming languages such as Java, PHP, and Perl [11].To show how prepared statements work in JAVA language, I modified the user authentication application (shown in Figure 1) by changing the regular statement to a prepared statement. As shown in Figure 26, the variable sql now contains a SQL statement with question marks, which serve as placeholders for the input data. User input data for the bind variables name and pass are passed to placeholders through the setter method setString. Therefore, the input data are bound to the placeholders and cannot modify the SQL statement itself [11].

```
String sql="select * from users where
loginname=? and pwd=?";
System.out.println("sql:"+sql);
PreparedStatement pstmt1 =
conn.prepareStatement(sql);
pstmt1.setString(1,name);
pstmt1.setString(2,pass);
ResultSet rs1=null;
rs1=pstmt1.executeQuery();
```

**Figure 1: Rewrite the User Authentication Program Using Prepared Statements**

Attackers can insert text “and <true condition>” to test if the application is vulnerable to SQL injection attacks. An example shows this fingerprinting approach in Figure 2. However, when the program uses prepared statements, attackers cannot find the vulnerability by this way anymore. I used the same injection method to test the new program. Unlike the fingerprinting, which displays the same result as that in normal use, Figure 1shows the injection to the new program receives “login failed” message. The reason is the prepared statement takes the whole input “ qqqqq’ and ‘1’=’1” as a password. Because there is no matching record in the USERS table, the authentication fails.
We also tested all the SQL injection. First, I ran the program for a normal use. The application returned the correct information for the legitimate user. Then I tried two tautology-based SQL injections and one UNION SELECT injection that had launched attacks successfully through the original user authentication application. All the attacks failed on the modified program. The testing result shows that prepared statements can easily repair SQL injection flaws in an application program.

However, prepared statements are not panacea for preventing SQL injection attacks. The second order injection prepared statements can be used to seed a malicious string into a database. What’s more, inappropriate usage of prepared statements can also make a program vulnerable to SQL injections. For example, inexperienced programmers may forget to use bind variables with prepared statements. To write a secure program, bind variables need to be used together with prepared statements [12].Developers also should implement input validation in their programs because insufficient input validation can cause vulnerabilities and a programming language’s type system is usually not good enough to check all the type errors [3, 13].To build a secure application, it is important to educate developers, to let them know the threat of SQL injection attacks and how to write secure code to avoid them. It is helpful to establish coding standards to guide their development work. However, defensive coding techniques cannot guarantee a flawless program because human errors are inevitable. Developers might make coding mistakes or forget to perform input validation even though they have made an effort to follow all the coding standards [3, 12].

### 2.1.2 Program Analysis Techniques

To overcome the shortcomings of defensive programming techniques, developers can use program analysis techniques to test programs and to discover defects they have overlooked.

One popular technique for program analysis is static checking. Researchers have proposed various static checking techniques. Gould and colleagues [13, 14] proposed a technique called JDBC checker to perform static analysis and to verify the type correctness of dynamically generated SQL queries [14].

For example, Assume the following code fragment is in a Java application program that is used to obtain the total price of an order. String query = "select '$' || " + "sum(price) from orders where oid=' "+order_ID +"'";

There is a type matching error in the code. The string concatenates the character '$' with the numeric value of sum(price). It is a runtime error, so the Java's type system cannot find it. Improper type checking can cause security problems of an application. The JDBC Checker is to enforce type correctness using finite state automata. Researchers have tested the tool on a variety of web applications and have found many known and unknown type mismatching errors existing in programs successfully [13]. Although it is good for preventing SQL injection attacks that take advantage of type mismatching bugs, it is not good at finding more general forms of SQL injection attacks such as SQL injections that generate type correct queries [3, 15]. WebSSARI (Web application Security by Static Analysis and Runtime Inspection) [16] is another tool that can perform static analysis. It uses a lattice-based static analysis algorithm. The tool can perform static checking and insert runtime guard code automatically if insecure pieces of code are detected. Static checking techniques help programmers optimize codes, but they usually cannot find all the vulnerabilities in a system.

Kosuga et al. [15] proposed a testing and debugging technique called Sania for detecting SQL injection vulnerabilities during the development phase. Sania constructs a parse tree for every intended SQL query and considers all the leaf nodes that take user inputs as vulnerable spots. Sania can automatically generate attack requests based on a list of existing attack codes collected in an investigation. It constructs a parse tree for each attack request and compares the tree with the one built for the intended SQL query. If the two trees are different, Sania determines there is a SQL injection vulnerability. To test the technique's performance, Sania was compared with an existing popular web application scanner called Paros. Sania outperformed Paros by being able to detect more vulnerabilities and generating fewer false positives. However, this technique cannot guarantee low false negative rate because it is hard to collect a complete list of attack codes. In addition, because Sania only checks the spots that take user input values, it cannot discover Lateral SQL injection vulnerabilities that do not require user input.

Another type of program analysis techniques is black-box testing. The testing is not based on the knowledge of a program's source code. Instead, testers take an external approach to get an attacker's point of view. They interact with the application by entering various crafted user inputs and observing program behaviors so that they can find vulnerable points in an application. Researchers have proposed several black-box assessment frameworks such as WAVES, AppScan, WebInspect and ScanDo [16]. There are also a bunch of commercial tools and open source tools available [17]. The drawback of this approach is that it usually cannot find all of the vulnerabilities existing in a system. How many can be found depends on the testing data used.

## 2.2 Secure an Oracle Database against SQL Injection Attacks

Attackers can also exploit defects or vulnerabilities existing at the database level. Therefore, it is important to have the underlying database system safeguarded. The Oracle website has suggested avoidance strategies against SQL injection attacks, covering privileges management, defensive coding techniques, and input sanitization. The core idea of privileges management is to apply the principle of least privilege to user accounts. By strictly controlling privileges and revoking any unnecessary privileges of users, the approach is trying to "reduce the attack surface", because the less exposed interfaces to users, the less possibility of abuse [6].

This section will focus on the defensive coding techniques and input sanitization.

### 2.2.1 Defensive Coding Techniques

To write secure PL/SQL programs such as stored procedures, the basic strategy is to use static SQL and avoid dynamic SQL, and when dynamic SQL is needed, use it together with bind arguments, which can bind user inputs with the corresponding placeholders in SQL statement [6].The major problem of SQL injection attacks against a PL/SQL program (e.g. stored procedures) stems from dynamic SQL. Therefore, the best way to eliminate SQL injection vulnerabilities in PL/SQL programs is to avoid using dynamic SQL. Static SQL is a more secure approach, because a static SQL statement is fixed at compile time and user inputs would not change its text. For security purpose, developers need to use static SQL instead of dynamic SQL whenever it is possible. For example, we can rewrite the stored procedure update_emp (shown in Figure 2) by changing the dynamic SQL to a static SQL. Figure 4 shows the modified stored procedure, which is more secure than the original one.

```
CREATE OR REPLACE PROCEDURE update_emp
(pid IN varchar2, eid IN varchar2) AS
BEGIN
        update employee set position_id = pid where id=eid;
END;
```

**Figure 2: The Stored Procedure that Uses Static SQL**

However, dynamic SQL is unavoidable under some conditions. In the cases when dynamic SQL is required, bind arguments should be used together with dynamic SQL so that the user inputs will not be concatenated directly with the statement. Using bind arguments is an effective way to mitigate SQL injection vulnerabilities.

As shown in the figure 3, after the procedure was created, I first tried a normal use by passing a pair of legitimate username and password to the procedure. The procedure returned the correct user's email. Then I tried the same injection attack that was launched successfully as shown in section 2.3. However, this time the attack failed. Oracle responded with error message "ORA-01403: no data found".

Bind arguments can also be used to handle Lateral SQL injection. If there are variables of DATE or NUMBER data type in a PL/SQL program, you can pass them using bind arguments to eliminate the Lateral SQL injection vulnerabilities. The modified procedure is shown in Figure 4.

The modified procedure has exactly the same functionality and behavior as the original one. When a program calls the procedure, if it passes the value of SYSDATE to the parameter d, the result will be the same as before. The procedure is susceptible to Lateral SQL injection attacks. To test if bind arguments can handle the Lateral SQL injection vulnerability in this procedure, I made a small change to the procedure by replacing the variable d with a placeholder and using bind arguments. As shown in Figure 8, the bold text is where the bind argument is used. Then I ran two tests. The first test was the normal use of the procedure, and the second test was the Lateral injection attack. The result of the two tests received the same correct answers from the procedure, which means the attack failed.

```
SQL> --Dynamic SQL statement using bind arguments
SQL> CREATE OR REPLACE PROCEDURE show_email
  2    (login_name IN varchar2, password IN varchar2, email OUT varchar2)
  3  IS
  4    stmt varchar2(4000);
  5  BEGIN
  6    stmt:='select email from users where loginname=:1 and pwd=:2';
  7  DBMS_OUTPUT.PUT_LINE('stmt: '|| stmt);
  8  EXECUTE IMMEDIATE stmt INTO email USING login_name,password;
  9  DBMS_OUTPUT.PUT_LINE('The email is ' ||email);
 10  end;
 11  /
Procedure created.
SQL> show errors
No errors.
SQL> --example of a normal use case
SQL> DECLARE
  2    email varchar2(2000);
  3  BEGIN
  4    show_email('Bill_L','qqqqq',email);
  5  END;
  6  /
stmt: select email from users
          where loginname=:1 and pwd=:2
The email is bill@gmail.com
PL/SQL procedure successfully completed.
SQL> show errors
No errors.
SQL> --To test if the procedure is vulnerable to SQL Injection
SQL> DECLARE
  2    email varchar2(2000);
  3  BEGIN
  4    show_email('xxx '' or loginname=''Mary_T''-- ','xxx',email);
  5  END;
  6  /
stmt: select email from users
          where loginname=:1 and pwd=:2
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "SCOTT.SHOW_EMAIL", line 12
ORA-06512: at line 4
```

**Figure 3: Stored Procedure Using Bind Arguments**

```
CREATE OR REPLACE PROCEDURE sales
            (d IN date,s OUT number)
IS
            stmt CONSTANT VARCHAR2(4000) :=
              'select sum(price) from orders where odate+30>'''||d||'''';
BEGIN
            DBMS_OUTPUT.PUT_LINE('stmt: ' || stmt);
            EXECUTE IMMEDIATE stmt into s;
            DBMS_OUTPUT.PUT_LINE('total sales:'||s);
END;
```

**Figure 4: The Procedure Sales Rewritten by Using DATE Variable**

```
SQL> --using bind arguments to handle lateral sql injection
SQL> CREATE OR REPLACE PROCEDURE sales
  2    (d IN date,s OUT number)
  3  IS
  4    stmt CONSTANT VARCHAR2(4000) :=
  5      'select sum(price) from orders where odate+30>:1';
  6  BEGIN
  7    DBMS_OUTPUT.PUT_LINE('stmt: ' || stmt);
  8    EXECUTE IMMEDIATE stmt into s USING d;
  9    DBMS_OUTPUT.PUT_LINE('total sales:'||s);
 10  END;
 11  /
Procedure created.
SQL> show errors
No errors.
SQL>
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
  2    s number;
  3    d DATE :=SYSDATE;
  4  BEGIN
  5    sales(d,s);
  6  END;
  7  /
stmt: select sum(price) from orders where odate+30>:1
total sales:70
PL/SQL procedure successfully completed.
SQL> ALTER SESSION SET NLS_DATE_FORMAT='''' or oid=''A0001''--''';
Session altered.
SQL> select sysdate from dual;
SYSDATE
------------------
' or oid='A0001'--

SQL> DECLARE
  2    s number;
  3    d DATE :=SYSDATE;
  4  BEGIN
  5    sales(d,s);
  6  END;
  7  /
stmt: select sum(price) from orders where odate+30>:1
total sales:70
PL/SQL procedure successfully completed.
```

**Figure 5: Guard the Procedure against Lateral Injection Using Bind Arguments**

The example in figure 5 demonstrates that bind arguments can also eliminate Lateral injection vulnerabilities in variables of DATE type. However, bind arguments cannot solve all Lateral injection problems. For example, the vulnerability of the original stored procedure sales cannot be fixed using this approach because it directly uses SYSDATE in the code. There are also other situations where we cannot use bind arguments, such as Oracle identifiers and DDL statements [6]. For example, the following dynamic SQL statement is not suitable for using bind arguments because it concatenates an identifier tablename. sql CONSTANT VARCHAR2(4000) := ' select * from ' || tablename ;

It is dangerous when there is a dynamic SQL statement that cannot use bind arguments existing in a program. Attackers can exploit it by injecting malicious code. Under this situation, developers must perform input filtering and sanitization to guard against SQL injection attacks [6].

### 2.2.2 Input Sanitization

To filter user inputs, developers need to check if user inputs are valid, e.g., if a column name submitted by a user really exists in the Oracle data dictionary view USER_TAB_COLS [5]. Developers can write PL/SQL programs to sanitize user inputs.

For example: To find if a table name is valid, they can write an input validation program including the following piece of code:
Select table_name into v_table from ALL_TABLES;
The code would search the view ALL_TABLES to see if the table name is valid. Oracle has a package called DBMS_ASSERT to help programmers to do the input sanitization. The package has several useful functions for preventing SQL injection attacks, such as the function QUALIFIED_SQL_NAME, SIMPLE_SQL_NAME, and ENQUOTE_LITERAL.

For example, the function DBMS_ASSERT.ENQUOTE_LITERAL encloses a user input within single quotes. It can also check if all other single quotes are paired with adjacent single quotes. If it finds ill-formed inputs, the “ORA-06502: PL/SQL: numeric or value error” exception will be raised. [5, 6]

We had used the vulnerable stored procedure update_emp (shown in Figure 6) as an example to describe how the function works. Figure 8 shows an attacker successfully exploited the procedure’s vulnerabilities by using the program procedure_update_emp.java to call the procedure and injecting malicious text “ 111' ; DELETE FROM orders WHERE oid='A0001 ”.

We had changed the original stored procedure by applying the function DBMS_ASSERT.ENQUOTE_LITERAL to it. Figure 6 shows the new procedure. Then I ran the program procedure_update_emp.java. As shown in Figure 6, this time the injection failed and the error “ORA-06502: PL/SQL: numeric or value error” was raised.

```
CREATE OR REPLACE PROCEDURE update_emp (pid IN varchar2, eid IN
varchar2)
IS
            stmt varchar2(4000);
BEGIN
            stmt:='begin
                        update employee set
position_id='||dbms_assert.enquote_literal(pid)||' where
id='||dbms_assert.enquote_literal(eid)||';'||'END;';
            DBMS_OUTPUT.PUT_LINE('stmt: ' || stmt);
            EXECUTE IMMEDIATE stmt;
END;
/
Show errors
```

**Figure 6: Stored Procedure Using the DBMS_ASSERT Package**

The package DBMS_ASSERT can help programmers do input validation. However, it is not a heal-all solution. It works by verifying if user inputs match any known patterns in its collection, but it is not a comprehensive solution for all kinds of malicious inputs [6]. For example, [18] shows a way to bypass the QUALIFIED_SQL_NAME function in the package. The attacker can simply double quote his malicious input to circumvent the input validation performed by this function. To solve the problem, programmers can write programs to perform input validation. However, this could be very time consuming.

```
SQL> CREATE OR REPLACE PROCEDURE sales
  2    (s OUT number)
  3  IS
  4    stmt CONSTANT VARCHAR2(4000) :=
  5      'select sum(price) from orders where odate+30>DATE '''||to_char(sysdate,'YYYY-MM-
             DD')||'''';
  6  BEGIN
  7    DBMS_OUTPUT.PUT_LINE('stmt: ' || stmt);
  8    EXECUTE IMMEDIATE stmt into s;
  9    DBMS_OUTPUT.PUT_LINE('total sales:'||s);
 10  END;
 11  /
Procedure created.
SQL> show errors
No errors.
SQL>
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
  2    s number;
  3  BEGIN
  4    sales(s);
  5  END;
  6  /
stmt: select sum(price) from orders where odate+30>DATE '2010-04-08'
total sales:70
PL/SQL procedure successfully completed.
SQL>
SQL> ALTER SESSION SET NLS_DATE_FORMAT='"'' or oid=''A0001''--"';
Session altered.
SQL> select sysdate from dual;
SYSDATE
------------------
' or oid='A0001'--

SQL> DECLARE
  2    s number;
  3  BEGIN
  4    sales(s);
  5  END;
  6  /
stmt: select sum(price) from orders where odate+30>DATE '2010-04-08'
total sales:70
PL/SQL procedure successfully completed.
```

**Figure 7: Techniques used to mitigate vulnerabilities in both the application layer and the database layer**

Figure 7 presented techniques used to mitigate vulnerabilities in both the application layer and the database layer during software development phase. However, there is no heal-all solution, and human errors are unavoidable. What's more, new kinds of SQL injection attacks may arise.

Therefore, even developers who make great effort to follow all the existing technique standards cannot guarantee the safety of their applications and the back-end database. For those systems that are not built in a security-critical way, the problem would be even worse.

# 3. Preventing SQLIA In Existing Applications

SQL injection issues are relative new in the information security area. Many old systems were designed when developers were not aware of such threats. In fact, SQL injection vulnerabilities are so prevalent that simple Google searching can find many of them. To rewrite all of the vulnerable code sections of an existing system is both time-consuming and often impractical due to financial or time constrains. Therefore, techniques for protecting deployed systems against SQL injection attacks are important. This chapter will present the research work that has been done in this area.

## 3.1 Static Analysis Approach

Static analysis can help programmers detect vulnerabilities they have overlooked and optimize codes. Although it is best used in the application development and debugging phase, it can also be used for protecting existing applications and programs. Developers can use static analysis to find bugs in programs so that they can modify source code or make patches for the applications. However, it could be tedious and time-consuming to patch a deployed system. In addition, static analysis often fails to detect all kinds of vulnerabilities so that it is easy to get high false negative rate [19]. What's more, static analysis can only identify the information available at compile time. In some cases, static analysis cannot capture the exact structure of an intended query because the full structure is only known at runtime [20]. For example, a web application ofa bookstore allows users to search books and can display the result in the order that is specified by users. The search result can be sorted by any of the columns such as the author's name and publication date by using an ORDER BY clause in the query. At compile time, it is impossible to know if or not the clause will be present in the query, because if a user does not want a sorted result he may just ignore the "Order by" options on the webpage. If the ORDER BY clause is used, it is still impossible to know the exact query statement at compile time because what column users will choose to sort results remains unknown until the runtime.

## 3.2 Combining Static Analysis and Runtime Analysis

Halfond and Orso [21] proposed a technique called AMNESIA to handle SQL injection problems by combining static analysis and runtime monitoring techniques. In its static phase, AMNESIA first finds all the hotspots in the application code. Hotspots are the points in a program where SQL queries are issued. Then AMNESIA builds a character-level Non-Deterministic Finite Automaton (NDFA) model for each hotspot by using the Java string analysis technique similar to the one used in [13]. The character-level model is further analyzed to group characters into SQL tokens and string tokens. A SQL token is a SQL keyword, a delimiter or an operator. A string token could be a hard-coded string written by programmers or a user submitted value [21]. Figure 8 shows an example SQL query model that represents two queries at a hotspot.

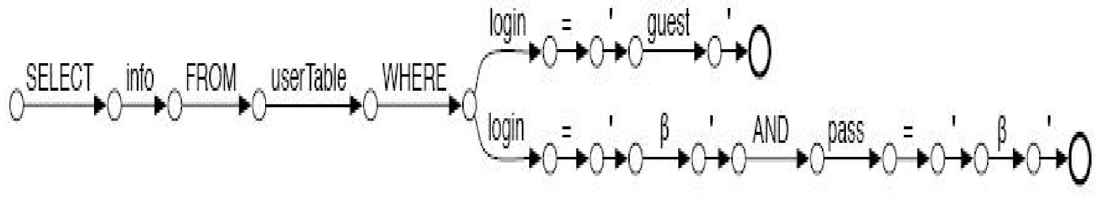

**Figure 8: An SQL Query Model [22]**

To counter SQL injection attacks, a call to a monitor is added into the program at each hotspot. A monitor is used to check the queries at runtime [22]. In the dynamic phase, the monitor will be invoked if a hotspot receives values. Then the monitor will parse the runtime generated SQL query into tokens and delimiters [21]. To find if the SQL query is legitimate, AMNESIA simply needs to check the parsed query against the corresponding SQL query model. If the model accepts the query, the monitor will consider it as a valid query and then pass it to the back-end database. For example, the following query will be accepted by the model using the first branch shown in Figure 8:
SELECT info FROM users WHERE login = ‘guest’;
But if we feed another query
SELECT info FROM users WHERE login = ‘anyname’ or 1=1 -- ‘and pass = ‘ anypwd’;
into the model, it will not be able to reach the final accept state because the token OR does not match the token AND in the model.
If the query does not match the model, the monitor will abort it and report the attack information, which will help developers to design appropriate ways to handle the attacks according to their specific needs [21]. For example, the information can be added into the pool of attack patterns and used by an intrusion detection system.The accuracy of the static analysis results would affect the performance of AMNESIA. For example, if the static analysis creates imprecise models, AMNESIA could raise a false alarm when a legitimate query comes in or fail to detect some SQL injection attacks [21]. In addition, the technique requires adding codes into the application programs to call the monitor. That could add extra burdens to developers [19].

## 3.3 Runtime Analysis Techniques

### 3.3.1 SQLGuard

SQLGuard is a runtime analysis technique that uses parse tree validation to counter SQL injection attacks [20]. SQL injections would change the structure of the query expected by programmers. Therefore, by comparing the tree structure of the SQL query before and after the inclusion of user submitted values, the technique can determine if there are SQL injection attacks [20]. In some cases static analysis cannot capture the exact structure of a query at compile time. Therefore, it is not good enough to simply compare the structure of a SQL query statement captured at compile time and the structure of the same query generated at runtime. SQLGuard, however, is trying to capture tree structures at execution time.

The core of the technique is to wrap user input using a random key and then replace the wrapped user input by a single token. Then the SQL query can be used to generate a runtime captured structure of the intended SQL query that does not include the user input. For example, assume there is a secret key K, which is not part of user inputs or the program code, and there is a query:

Select * from employee where first_name = ‘Bill’ and last_name = ‘Lee’; (1)

To use the key K to wrap the two user inputs “Bill” and “Lee” in the SQL query, we can simply concatenate the key K with them [20, 23]. Therefore, the query will be changed to:

Select * from employee where first_name = ‘KBillK’ and last_name = ‘KLeeK’; (2)

Then we can replace each wrapped user input by a single literal token such as “?” [20, 23]. The query becomes:

Select * from employee where first_name = ‘ ? ’ and last_name = ‘ ? ’; (3)

A parse tree can be built for the query (3). The literal tokens will appear to be the leaf nodes of the tree. This method maintains the intended structure of a query because only the user provided portions are replaced. Then we build another tree for query (1) which includes user input values. The two trees will be compared concerning the node types, and the values of corresponding nodes [23]. If the two trees have the same structure, the query will be determined as legitimate. Otherwise, it is rejected.

SQLGuard generates a 64-bit (8 characters) random key for each query by inserting the method SQLGuard.init() to each hotspot, so the technique requires code changes on the programs. The key should be secret and never be a part of user input or program code [20]. It also should be hard to guess. Otherwise, attackers may bypass the detection. However, to guess a 64-bit key is not very difficult by using some key cracking tools. To reduce the possibility of key compromising, the technique may use longer keys.

### 3.3.2 SQLProb

SQLProb [24] is another dynamic analysis approach. It uses a customized MySQL Proxy to collect queries, extract user input, generate parse trees, and validate user input. Figure 9 shows the architecture of the SQLProb system. The shaded area is the off-the-shelf MySQL Proxy.

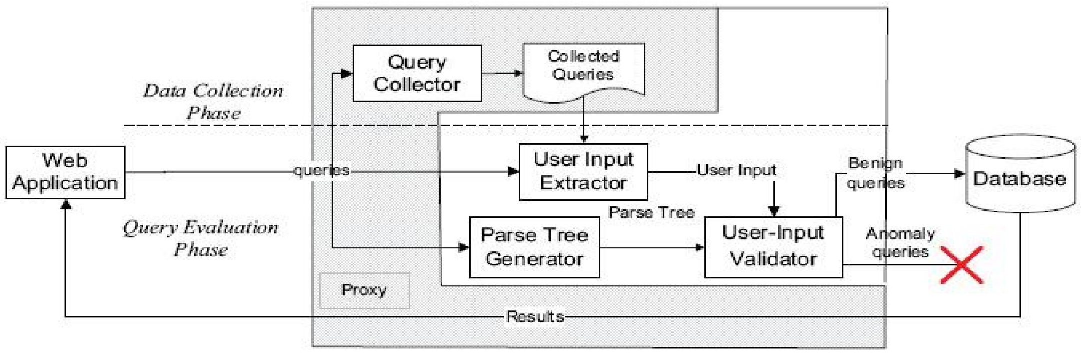

**Figure 9: Overview of the SQLProb System Architecture [24]**

In the data collection phase, the Query Collector collects all the queries expected by the programmer and stores them in the system. In the query evaluation phase, an incoming query q generated by the application is sent to the Parse Tree Generator module and the User Input Extractor module at the same time. The Parse Tree Generator module generates the tree Tree(q) for the query using JavaCC. The User Input Extractor module uses an enhanced version of the Needleman-Wunsch algorithm to align query q with all the queries gathered in the data collection phase [24]. The algorithm can calculate the similarity of the query q and every collected query. The collected query that has the highest similarity value is called the prototype query of q. If there are m queries stored in the system, this step will require m comparisons, which is expensive. So the system preprocesses all the collected queries by clustering and aggregating them to reduce the size of the query collection. The paper shows the size of the collected query can be reduced to only 6% to 25% of the original size [24]. This will save a lot for the query comparisons.

After the prototype query of q has been found, the Needleman-Wunsch algorithm extracts the user input portion of q by aligning q and its prototype query. The extracted user input is fed into the User Input Validator module. Tree(q) generated by the Parse Tree Generator also comes into this module. The user input is actually corresponding to a set of leaf nodes that appear in the parse tree. Let S denotes this set of nodes. To validate the user input, we simply conduct an upward depth-first search from all the nodes in S until their paths intersect at an internal node. Then starting from this internal node, we perform a breadth-first search to reach all the leaves of the tree. If the result set of the leaves is a superset of S, the user input is considered malicious and will be rejected. Figure 10 shows an example WHERE clause subtree. The user input for the password field is "nonsense' OR '1=1". Its corresponding five leaf nodes intersect at the node SQLExpression by using depth-first search. Then starting from the node SQLExpression, we can do breadth-first search which will reach all the leaves of this WHERE clause subtree. The breadth-first search result is a superset of the five leaf nodes. Therefore, the user input is determined as malicious.

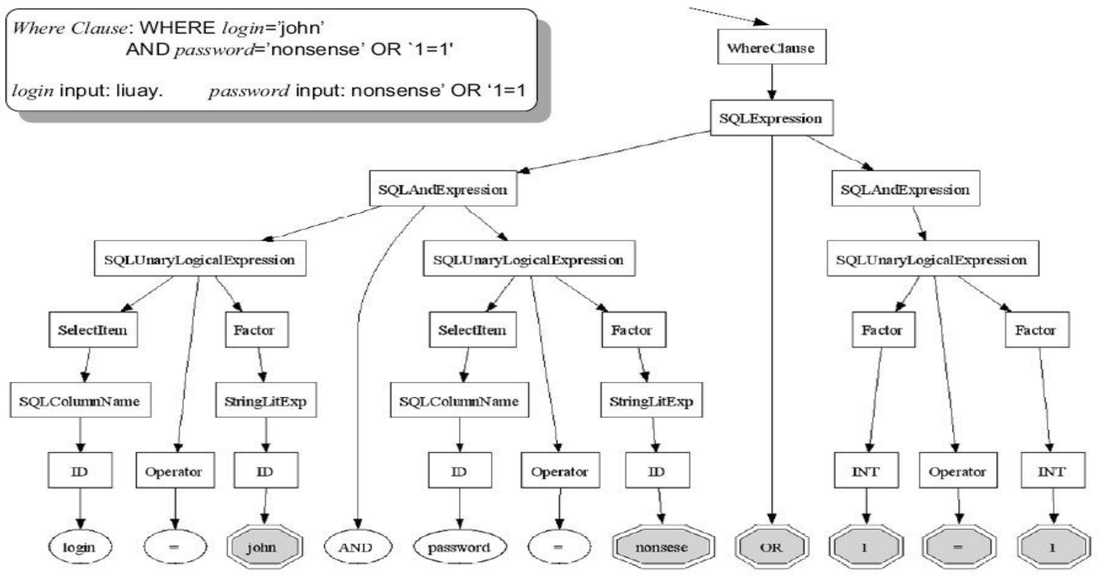

**Figure 10: A Subtree for WHERE Clause of a Malicious Query [24]**

Unlike many other approaches such as [20, 21] that need to add functions or methods to the source code, SQLProb does not require any code changes. Due to the nature of the Needleman-Wunsch algorithm, if the queries need to be compared are very long, the technique can cause significant delay [24]. The performance of SQLProb also depends on the completeness of the collected queries as well as the accuracy of aggregating these queries. What's more, SQLProb development uses the off-the-shelf MySQL Proxy, which is a program for MySQL server. So this tool cannot be used on other database systems.

## 3.4 Signature Based and Anomaly Based Intrusion Detection

Signature based intrusion detection works by conducting pattern matching on incoming traffic with a list of known attack patterns maintained in the system. For example, the signature of a tautology-based SQL injection may contain expressions describing attack patterns such as "or <true condition> -- ". Signature based protection have been widely used to guard systems against SQL injection attacks. Due to its nature, signature based detection has low false alarm rate, but it cannot detect new attacks whose patterns have not been observed. The list of known attack patterns needs to be updated regularly. What's more, there are many techniques can be used by attackers to evade the signature based detection, such as the alternative encoding techniques which can change the look of a malicious string [3, 25].

Another type of intrusion detection techniques is anomaly detection. Anomaly detection technique trains models by using a set of normal usage patterns. The model can then be used to monitor the incoming traffic. When a significant deviation from the expected patterns is detected at runtime, the deviation will be reported as a possible attack.

Fredrik Valeur et al. [26] proposed an anomaly-based intrusion detection system based on machine learning technique. In the training phase, the technique feeds a number of attack-free application queries into the machine-learning algorithm, which will then generates models that can characterize the profiles of normal usages. The anomaly scores of all the training queries are also calculated. The maximum score will be set to be the anomaly threshold. In the detection phase, the intrusion detection system intercepts the communication from applications to the back-end database server. The incoming queries are transformed by a feature selector into the one of the forms supported by the models. The detection engine then tries to classify them using the models and generates the assessment report for each incoming query. The system calculates anomaly score for each query. If a query's score is higher than the threshold, the query is determined as a SQL injection attack and will be dropped. In principle, the anomaly-based detection can detect new attacks. However, in practices, this is difficult to achieve. The quality of the models depends on how accurate and comprehensive the training data are as well as how good the learning algorithm is. It is usually hard to obtain high quality training data. A poor collection of the normal database access data could result an imprecise model that will generate high false positive rate and high false negative rate.

### 3.5 Specification-based Approach

Kemalis and Tzouramanis [19] proposed a specification-based approach to detect SQL injection attacks. This technique is based on the assumption that an injected statement and the intended statement of the program have different structures. Therefore, a comparison of their structures can tell if the submitted statement is malicious. We can use specifications to describe the intended structure of all the application-generated statements. The specifications describe the rules about what syntactic structure an application-generated SQL query should follow in order to be considered as legitimate [19]. Kemalis and Tzouramanis created specifications for their applications using Extended Backus Naur Form (EBNF) based on the ISO/IEC SQL database language criteria. Figure 11 shows the specification for the following SQL query. Select user_id, user_level from users where username = 'abc' and password = 'a@fa';

```
<Query specification>    :=
SELECT <Select List> <From Clause> <Where Clause>

<Select List>    :=
<Table Column> (<COMMA>  <Table Column>)*

                    <From Clause>    :=
FROM<Table reference>

<Where Clause>:=
WHERE<search condition>AND < search condition>

< search condition> :=
<Table column>"="<STRING LITERAL>
```

**Figure 11: The Specification for the Given SQL Query [19]**

An intrusion detection system named SQL-IDS was built. It contains two modules: the validity check module (VCM) and the event monitoring module (EMM). The intrusion detection system monitors traffic between the Web application and the back-end database. The event monitoring module filters the traffic by intercepting SQL queries and sending them to the validity check module. The VCM performs lexical analysis and syntactical analysis on the received SQL queries and parse them to tokens [19]. Then the tokens of a SQL query are checked against the pre-defined specifications to identify if the query is a syntactically correct SQL statement. The legitimate SQL query will be forwarded to the database. If a syntactically wrong SQL is detected, the system will raise an alarm and abort the execution of the query.

An advantage of this approach is that it can be applied to any application environments and database systems [19]. According to the paper, the testing result of SQL-IDS shows the system has very good performance with zero false negative and zero false positive. The performance actually relies on the completeness of the ISO/IEC standard criteria and if or not all the expected application-generated SQL statements have been correctly specified. Another advantage of this technique is that it does not need to modify the application source code.

## 4. Conclusion

Nowadays, many businesses and organizations use web applications to provide services to users. Web applications depend on the back-end database to supply with correct data. However, data stored in databases are often targets of attackers. SQL injection is a predominant technique that attackers use to compromise databases. During the project, I have conducted a survey of different types of SQL injection attacks, and have built applications and Oracle database environment to illustrate how they work. As I presented, there are many types and forms of basic SQL injection attacks. The combination of them could come up with attacks that are more complicated. Many countermeasures against SQL injection attacks have been proposed and implemented by academic researchers, developers and databases venders. In this project, I was able to get a good insight into the existing techniques and approaches used to secure and protect web applications and database systems. I also have identified techniques that can be used to secure my applications and database system against SQL injection attacks and have applied them. By conducting a comprehensive survey on existing techniques, I have realized that many SQL injection countermeasures have their limitations. Understanding and identifying the working mechanisms, as well as advantages and disadvantages of current techniques will benefit future work in this area.

## Acknowledgement

I feel an immense amount of pleasure in submitting this manuscript on **"A Survey Of Sql Injection Countermeasures"** for publication in the journal. The satisfaction and euphoria that accompany the successful completion of any research would be incomplete without a mention of people who made it possible and whose constant guidance and encouragement crown all the efforts.

First and foremost, I would like to express my sincere gratitude to my advisor, **Mr. R.P.Mahapatra** for his invaluable guidance and cooperation throughout the work. His constant encouragement and vision enabled me to take this new endeavor to the success path.

I would like to express my hearty gratitude to **Dr. Manoj Pandey**, Director, SRM University, NCR Campus, Ghaziabad for his support and encouragement throughout the work and for providing the suitable environment to carry out the work.

I also extend my heartfull gratitude to my husband **Mr. Parvez Khan,** whose constant motivation and support made me enthusiastic throughout the work.

Finally, an honorable mention goes to my beloved parents for their understanding and blessings. I would also like to express my heartfull thanks to all my classmates for their constant support throughout this project.

In the last but not least I am indebted to all the teaching and non-teaching staff members of our college for helping me directly or indirectly by all means throughout my whole research .

## References :

[1] IBM Internet Security Systems X-Force® research and development team, "IBM Internet Security Systems™ X-Force® 2009 Mid-Year Trend and Risk Report," Aug. 2009. [Online]. Available: www-935.ibm.com/services/us/iss/xforce/trendreports/. [Accessed: Apr. 10, 2010].

[2] V. Chapela, "Advanced SQL Injection," OWASP Foundation, Apr. 2005. [Online]. Available: www.owasp.org/images/7/74/Advanced_SQL_Injection.ppt. [Accessed: Mar. 2, 2010].

[3] W. G. Halfond, J. Viegas, and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," In Proc. of the Intern. Symposium on Secure Software Engineering (ISSSE 2006), Mar. 2006.

[4] E. M. Fayó, "Advanced SQL Injection in Oracle databases," Argeniss Information Security, Black Hat Briefings, Black Hat USA, Feb. 2005. [Online]. Available: http://www.orkspace.net/ secdocs/Web/SQL%20Injection/Advanced%20SQL%20Injection%20In%20Oracle%20Databases.pd f. [Accessed: Mar. 18, 2010].

[5] "Oracle® Database PL/SQL Language Reference 11g Release 1 (11.1)," Oracle Corp., 2009. [Online]. Available: http://download.oracle.com/docs/cd/B28359_01/ appdev.111/b28370/toc.htm. [Accessed: Feb. 19, 2010].

[6] "SQL Injection Tutorial," Oracle Corp., 2009. [Online]. Available: http://st-curriculum.oracle.com/tutorial/SQLInjection/index.htm. [Accessed: Mar. 11, 2010].

[7] C. Anley, "Advanced SQL Injection in SQL Server Applications," NGSSoftware Ltd., United Kingdom, 2002. [Online]. Available: http://www.ngssoftware.com /papers/advanced_sql_injection.pdf. [Accessed: Feb. 09, 2010].

[8] D. Litchfield, "Lateral SQL Injection: A New Class of Vulnerability in Oracle," NGSSoftware Ltd., United Kingdom, Feb. 2008. [Online]. Available: www.databasesecurity.com/dbsec/lateral-sql-injection.pdf. [Accessed: Mar. 15, 2010].

[9] D. Litchfield, "David Litchfield's Weblog: Lateral SQL Injection Revisited - No Special Privs Required," July 2008. [Online]. Available: http://www.davidlitchfield.com/blog/. [Accessed: Mar. 15, 2010].

[10] "Oracle SQL Injection in web applications," Red-Database-Security GmbH company, Germany, 2009. [Online]. Available: http://www.red-database-ecurity.com /whitepaper/oracle_sql_injection_web.html. [Accessed: Mar. 16, 2010].

[11] M. Nystrom, SQL injection defenses, Sebastopol, Calif.: O'Reilly, 2007, pp. 19-39.

[12] S. Kost, "An Introduction to SQL Injection Attacks for Oracle Developers," Integrigy Corp. Chicago, IL, Jan. 2004. [Online]. Available: www.integrigy.com /security/Integrigy_Oracle_SQL_Injection_Attacks.pdf. [Accessed: Feb. 22, 2010].

[13] C. Gould, Z. Su, and P. Devanbu, "Static Checking of Dynamically Generated Queries in Database Applications," In Proceedings of the 26th International Conference on Software Engineering (ICSE 04), pp. 645–654, 2004.

[14] C. Gould, Z. Su, and P. Devanbu, "JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications," In Proceedings of the 26th International Conference on Software Engineering (ICSE 04), pp. 697–698, 2004.

[15] Y. Kosuga, K. Kono, M. Hanaoka, M. Hishiyama, and Y. Takahama, "Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection," In Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC '07), Miami Beach, Florida, pp. 107-116, 2007.

[16] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo, "Securing Web Application Code by Static Analysis and Runtime Protection," in Proceeding of the 13th International Conference on World Wide Web, New York, NY, USA, pp. 40-52, May 2004.

[17] C. C. Michael, W. Radosevich, and K. V. Wyk, "Black Box Security Testing Tools," Cigital Inc., USA, July 2009. [Online]. Available: https://buildsecurityin.us-cert.gov/bsi/articles/tools/black-box/261-BSI.html#dsy261-BSI_BWG. [Accessed: Apr. 02, 2010].

[18] A. Kornbrust, "Bypassing Oracle dbms_assert," Red-Database-Security GmbH company, Germany, 2006. [Online]. Available: http://www.red-database-security.com/wp/bypass_dbms_assert.pdf. [Accessed: Mar. 12, 2010].

[19] K. Kemalis and T. Tzouramanis, "SQL-IDS: A Specification-based Approach for SQL-Injection Detection," SAC '08: Proceedings of the 2008 ACM symposium on Applied computing, pp. 2153-2158, March 2008.

[20] G. T. Buehrer, B. W. Weide, and P. A. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," In Proceedings of the International Workshop on Software Engineering and Middleware (SEM) at Joint FSE and ESEC, pp. 106-113, Sept. 2005.

[21] W. G. Halfond and A. Orso, "Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks," In Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005), St. Louis, MO, USA, pp. 22–28, May 2005.

[22] W. G. Halfond and A. Orso, “Preventing SQL Injection Attacks Using AMNESIA,” ICSE’06, Shanghai, China, May 2006.

[23] G. T. Buehrer, B. W. Weide, and P. A. Sivilotti, “Using parse tree validation to prevent SQL injection attacks,” Presentation, Ohio State University, 2005. [Online]. Available:www.cse.ohio-state.edu/~paolo/research/publications/sem05_talk.pdf. [Accessed: Apr. 02, 2010]

[24] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou, “SQLProb: a proxy-based architecture towards preventing SQL injection attacks,” SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing, pp. 2054-2061, March 2009.

[25] O. Maor and A. Shulman, “Sql injection signatures evasion: An overview of why sql injection signature protection is just not enough,” iMPERVA Inc., Israel, Apr. 2004. [Online]. Available: http://www.www.packetstormsecurity.org/papers/bypass/SQL_ Injection_Evasion.pdf. [Accessed: Mar. 01, 2010].

[26] F. Valeur, D. Mutz, and G. Vigna, “A Learning-Based Approach to the Detection of SQL Attacks,” In Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), Vienna, Austria, pp. 123-140, July 2005.

[27] A. Roichman and E. Gudes, “Fine-grained access control to web databases,” In 12th ACM symposium on Access Control Models and Technologies, pp. 31-40, 2007.

## Authors Brief :

1. Mr. Rajendra Prasad Mahapatra PhD (CSE) presently working as Associate Dean at Modinagar Campus of SRM University; contributing his enormous research works in the area contributing his enormous research works in the field of Computer Science, Simulations, Network Security and Artificial Intelligence and RF Technology.

2. Mrs Subi Khan M.Tech (CSE), presently working as Assistant Professor at Ideal Institute Of technology, done research on various fields in computer science including SQLIA on which this research paper is based