AN EFFICIENT TECHNIQUE FOR SQL INJECTION DETECTION AND PREVENTION

Esraa Mohamed Safwat
Computer science Department
Faculty of computer and information
Menoufyia University
Esraa_safwat87@yahoo.com

Hany Mahgoub
Computer science Department
Faculty of computer and information
Menoufyia University
H_mghguob@yahoo.com

ASHRAF EL-SISI,
Computer science Department
Faculty of computer and information
Menoufyia University
ashrafelsisim@yahoo.com

Arabi Keshk
Computer science Department
Faculty of computer and information
Menoufyia University
arabikeshk@yahoo.com

Abstract: With the recent rapid increase of interactive web applications that employ back-end database services, a SQL injection attack has become one of the most serious security threats. This type of attack can compromise confidentiality and integrity of information and database. Actually, an attacker intrudes to the web application database and consequently, access to data. For preventing this type of attack different techniques have been proposed by researchers but they are not enough because most of implemented techniques cannot stop all type of attacks. In this paper our proposed technique are detection of SQL injection and prevention based on first order, second order and blind SQL injection attacks online. The proposed technique implemented in JAVA and evaluated for seven types of SQL injection attacks. Experimental results have shown that the proposed technique is efficient related to execution time overhead. Our technique need to be one second overhead to execution time. Moreover, we have compared the proposed technique with the popular web application vulnerabilities scanner techniques. The most advantages of proposed technique Its easiness to adopt by software developer, having the same syntactic structure as current popular record set retrieval methods.

Keywords: - Web application, Database SQL Injection Attack, detection, prevention

1. Introduction

Web sites are dynamic, static, and most of the time a combination of both. Web sites need protection in their database to assure security. An SQL injection attacks interactive web applications that provide database services. These applications take user inputs and use them to create a SQL query at run time. In an SQL injection attack, an attacker might insert a malicious SQL query as input to perform an unauthorized database operation. Using SQL injection attacks, an attacker can retrieve or modify confidential and sensitive information from the database.

The popular solutions for the prevention of SQL injection attacks include coding best practices, input filtering, escaping user input, usage of parameterized queries, implementation of least privilege, white list input validation [2,3,4], These solutions should be employed usually during the development of an application. This is the major limitation of such solutions as they do not cover the millions of Web applications already deployed with this vulnerability. Detection of SQL injection and prevention technique is proposed based first order, second order and blind SQL injection attacks online. SQL injection detection and prevention (SQLI-

DP) technique is implemented in JAVA and evaluated for five types of SQL injection attacks. Experimental results have that proposed technique is efficient related to execution time overhead, it's need to be approximately 1second overhead to execution time. In addition, it is easily adopted by software developer, having the same syntactic structure as current popular record set retrieval methods.

The rest of this paper is organized as follows; section 2 present the problem formulation .section 3 presents the background about SQL injection attacks and the concept of parse tree and presents related work. Proposed SQLI-DP technique and experimental results is presented in section 4. Section 5 conclusion and future work is presented. Section 6 provides the References.

2. Problem Formulation

SQL injection is a technique maliciously used to obtain unrestricted access to databases by inserting maliciously crafted strings to SQL queries via a web application. It allows an attacker to spoof his identity, expose and tamper with existing data in databases, and control databases server with the privileges of its administrator. There is a variable SQL injection scanner but it prolongs the connection time if it wants detect or prevent or both. The popular solutions for the prevention of SQL injection attacks can't solve the problem of legacy system and the software developer not easy to adapt. In this paper we try to improve the execution time and try to found way to improve the legacy system to able to prevent injection attacks. Try to make our technique easy to adapt by software developer and make it more efficient in the future.

3. Background of SQL injection attack and parse tree concept

3.1 classifications of SQL injection attacks (SQLIA):

There are different methods of attacks which depend on the goal of attacker are performed together or sequentially [5, 6, 7]. For a successful SQLIA the attacker should append a syntactically correct command to the original SQL query. Show the types of SQLIAs attacks with examples as the following.

Table 1. Types of SQL injection attacks

Types of Attack	Description
Tautologies	SQL injection codes are injected into one or more conditional statements so that they are always evaluated to be true. **Exp:** Generated SQL Query**:** ***SELECT username, password FROM clients WHERE username = ‘user1 OR ‘1’ =’1 —‘ AND password = ‘whatever’***.
Logically Incorrect Queries	Using error messages rejected by the database to find useful data facilitating injection of the backend database. **Exp:** ***“Microsoft OLEDB provider for SQL Server (0×80040E07)Error converting nvarchar value „CreditCards‟ to a column of data type int”***
Union Query	Injected query is joined with a safe query using the keyword UNION in order to get information related to other tables from the application. **Exp:** ***SELECT * FROM Table1 WHERE id = -1 UNION ALL SELECT null, null, NULL, NULL, convert(image,1), null, null,NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULl, NULL--***
Stored Procedure	Many databases have built-in stored procedures. The attacker executes these built in functions using malicious SQL Injection codes. **Exp*: SELECT accounts FROM users WHERE login= 'doe' AND pass=' '; SHUTDOWN; -- AND pin =***
Piggy-Backed Queries	Additional malicious queries are inserted into an original injected query. **Exp:** ***SELECT * FROM products WHERE id = 10; DROP TABLE members;--***
Alternate Encoding	the injected text is changed in order to evade detection by defensive coding practices and most of the automatic prevention techniques. Encodings such as hexadecimal, ASCII and Unicode character encoding can be used for attack strings. **Exp*: SELECT * FROM Accounts WHERE user='user1'; exec(char(0x73687574646f776e)) -- ' AND pass=' ' AND eid=***
Blind Injection	An attacker derives logical conclusions from the answer to a true/false question concerning the database. - Information is collected by inferring from the replies of the page after questioning the server true/false questions. **Exp:** ***index.php?id=1 and 1=(SELECT 1 FROM information_schema.tables WHERE TABLE_SCHEMA="blind_sqli" AND table_name REGEXP '^[a-n]' LIMIT 0,1)***

3.2 Parse tree concept

Parse tree is a data structure for the parsed representation of a statement. Parsing a statement requires the grammar of the statement's language. By parsing two statements and comparing their parse trees, we can determine if the two queries are equal. When a malicious user successfully injects SQL into a database query, the parse tree of the intended SQL query and the resulting SQL query do not match. By intended SQL query, we mean that when a programmer writes code to query the database, she has a formulation of the structure of the query. The programmer-supplied portion is the hard-coded portion of the parse tree, and the user-supplied portion is represented as empty leaf nodes in the parse tree. These nodes represent empty literals. What she intends is for the user to assign values to these leaf nodes. A leaf node can only represent one node in the resulting query, it must be the value of a literal, and it must be in the position where the holder was located. By restricting our validation to user-supplied portions of the parse tree, we do not hinder the programmer from expressing her intended query. An example of her intended query is given in Figure 1. This parse tree corresponds to the example we presented in Section 1, SELECT * FROM users WHERE username=? AND password=?. The question marks are place holders for the leaf nodes she requires the user to provide.3 While many programs tend to be several hundred or thousand lines of code, SQL statements are often quite small. This affords the opportunity to parse a query without adding significant overhead.

Suppose that [8] a database contains name and password fields in the users table, and a web application contains the following code to authenticate a user's log in.

sql = "SELECT * FROM users WHERE name = '" + request.getParameter(name) + "' AND password = '" + request.getParameter(password) + "'";
This code generates a query to obtain the authentication data from database. If an attacker inputs "' or 1=1 -- 1" into the name field, the query becomes:

SELECT * FROM users WHERE name = '' or 1=1 -- 1 AND password = 'xxx';

The WHERE clause of this query is always evaluated to be true, and thus an attacker can bypass the authentication, regardless of the data inputted in the password field.Web applications commonly use SQL queries with client-supplied input in the WHERE clause to retrieve data from a database. By adding additional conditions to the SQL statement and evaluating the web application's output, you can determine whether or not the application is vulnerable to SQL injection. For instance, many companies allow Internet access to archives of their press releases. A URL for accessing the company's fifth press release might look like this:

http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5

The SQL statement the web application would use to retrieve the press release might look like this (client-supplied input is underlined):

SELECT title, description, releaseDate, body FROM pressReleases WHERE pressReleaseID = 5

The database server responds by returning the data for the fifth press release. The web application will then format the press release data into an HTML page and send the response to the client. To determine if the application is vulnerable to SQL injection, try injecting an extra true condition into the WHERE clause. For example, if you request this URL . . .

http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND 1=1

. . . and if the database server executes the following query . . .

SELECT title, description, releaseDate, body FROM pressReleases WHERE pressReleaseID = 5 AND 1=1

. . . and if this query also returns the same press release, then the application is susceptible to SQL injection. Part of the user's input is interpreted as SQL code.

A secure application would reject this request because it would treat the user's input as a value, and the value "5 AND 1=1" would cause a type mismatch error. The server would not display a press release.

The process of generation of queries in a dynamic web application can be represented as a function of user's inputs. In this context, SQL injection is any situation in which the user's input is inducing an unexpected change in the output generated by the function. We define two parameters

SQL Statement = SQL(Argi) (*i=1 to n*)
Argi ← Input from user

SQL() ← function represented by web application
SQL Statement Safe = SQL(Arg Safe i) (*i=1 to n*)
Arg Safe i ← "qqq" or any single token

We require that the application will not allow the user to enter any part of SQL query directly. We define that two statements are semantically equivalent, if they perform similar activities, once they are executed on the database server. So, if we determine that both SQL Statement and SQL Statement Safe are semantically equivalent, then by definition the SQL Statement is bound to have an expected behaviour and there is no possibility for a SQL Injection. Here semantic action implies a particular activity like comparison, retrieval etc., and not the lexical equality. We use this semantic comparison to detect SQL Injection. The semantic comparison is done by parsing each of the statements and comparing the syntax tree structure. If the syntax trees of both the queries are equivalent, then the queries are inducing equivalent semantic actions on the database server, since the semantic actions are determined by the structure of the SQL statement. For example,

Let,

Arg = {α, β} Arg Safe = {"qqq", "qqq"}

Now,

SQL Statement = SQL(Arg)
= SELECT ∗ FROM 'User _Table'
WHERE user_ name = 'α' AND password = 'β'

SQL Statement Safe = *SQL(Arg Safe)*
= *SELECT* $*$ *FROM User_ Table*
WHERE user_ name = *' qqq' AND password* = *'qqq '*

The SQL Statement Safe is parsed to produce a syntax tree as shown in figure 1. We can consider two cases of user inputs, first case without any injection and second with injection.

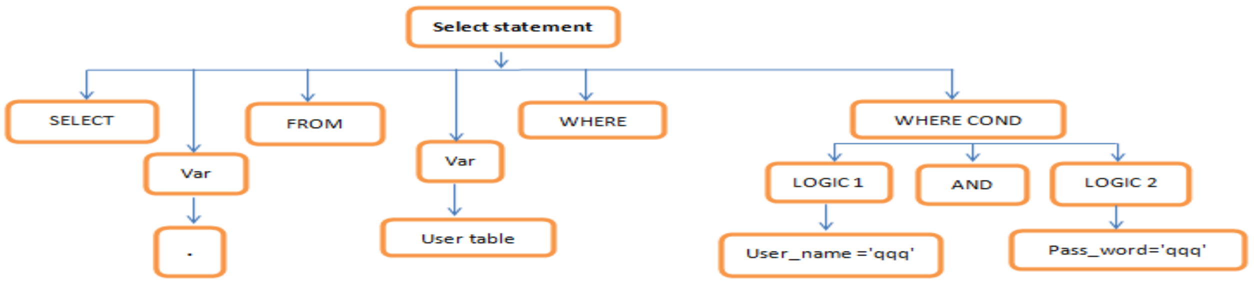


Figure.1. SQL_Statement_Safe

Case 1: Let,
Arg Normal = {*α*, *β*} = {*admin*, *admin pwd*}

Now,

SQL Statement Normal = *SQL (Arg Normal)*
= *SELECT* $*$ *FROM User_ Table WHERE*
user name = *' admin' AND*
password = *'admin_ pwd'*

The *SQL Statement Normal* can be parsed as shown in figure 2. On comparing the semantic structure of *SQL statement safe* and *SQL statement normal*, we can see that both of them have similar semantics. Both statements are extracting all values from a table after checking for two logic equalities combined with an AND operator. This implies that there is no possible SQL injection and hence we can safely execute the query.

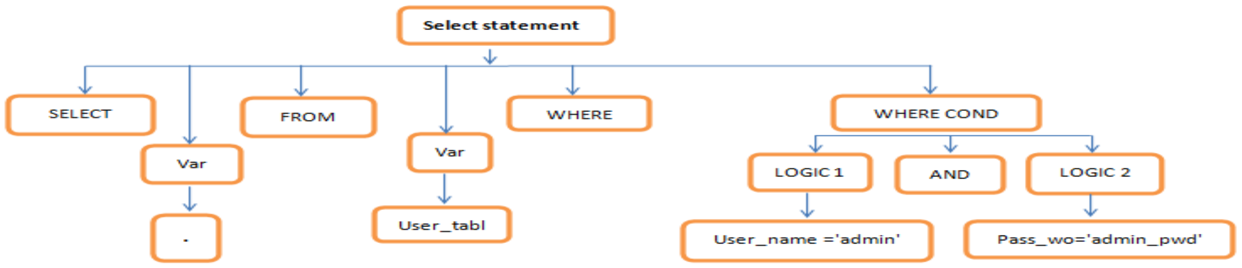


Figure.2. SQL_Statement_Normal

Case 2: Let,
Arg Injection = {*α*, *β*} = {*admin_ OR _1_ =_ 1*, *hacker pwd*}.

Now,

$$SQL\ Statement\ Injection = SQL(Arg\ Injection)$$

$$= SELECT * FROM\ Use\_Table\ \ WHERE\ user\_name = admin\ \boldsymbol{OR}\ \mathbf{`1' = '1'}$$
$$AND\ password = `hacker\_pwd'$$

In this case, on comparison of *SQL Statement Injection* which is represented in figure 3 with *SQL Statement Safe*, we can see that the semantic structures of The two statements are not similar. This is because *SQL Statement Injection* is doing the additional action of ”OR ’1’ = ’1’ ”. This implies that on application of the input, the semantics of the output has been modified. This detects a possible SQL injection and the execution of the query should be stopped [10]. So to prevent and detect the SQL injection we use parse tree.

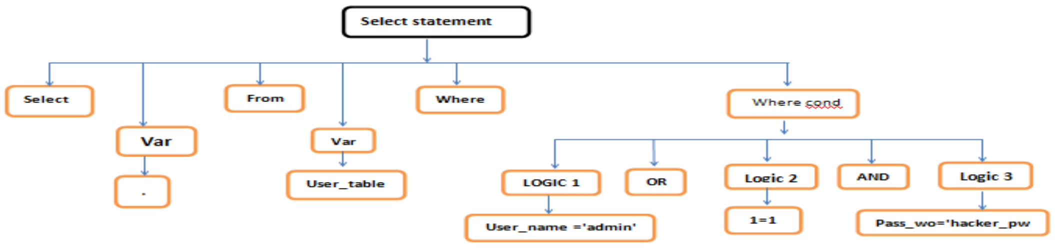


Figure.3. SQL_Statement_Injection

3.3 Related Work

The techniques related to SQL injection are classified and evaluated by [9]. In this section, we list the work related to ours and discuss their pros and cons. We can classify the related work to two main parts:

3.3.1 Detections SQL injection techniques

Vulnerability detection is an approach for detecting vulnerabilities in web applications, especially in the development and debugging phases. This approach is conducted either manually by developers or automatically with the use of vulnerability scanners. In the manual approach, an auditor manually reviews source code and/or attempts to execute real attacks to the web application. For discovering vulnerabilities, the auditor is required to be familiar with the software architecture and source code, and/or to be a computer security expert to attempt effective attacks tailored to his or her web application. A comprehensive audit requires a lot of time and its success depends entirely on the skill of the auditor. In addition, manual check is prone to mistakes and oversights. On the other hand, the vulnerability scanners automate the process of vulnerability detection without requiring the auditor to have detailed knowledge of the web applications including security details. The automated vulnerability scanners eliminate mistakes and oversights that is typically prone to be made by manual vulnerability detection.

From this reason, vulnerability scanners are widely used for detecting vulnerabilities in web applications.

The dynamic analysis scanners are based on penetration test, which evaluates the security of web applications by simulating an attack from a malicious user. The attack is typically generated by embedding an attack code into an innocent HTTP request. After sending the attack to the target web application, the vulnerability scanner captures the web application output to analyze the existence of vulnerabilities. Existing vulnerability scanners [10, 11, 12, 13, 14] employ dynamic analysis techniques for detecting vulnerabilities. AMNESIA combines static analysis and runtime monitoring [15]. In static phase, it builds models of the different types of queries which an application can legally generate at each point of access to the database. Machine Learning Approach Valeur [16] proposed the use of an intrusion detection system (IDS) based on a machine learning technique. IDS is trained using a set of typical application queries, builds models of the typical queries, and then monitors the application at runtime to identify the queries that do not match the model. The overall IDS quality depends on the quality of the training set; a poor training set would result in a large number of false positives and negatives. WAVES [17] is also based on a machine learning technique. WAVES is a web crawler that identifies vulnerable spots, and then builds attacks that target those spots based on a list of patterns and attack techniques. WAVES monitors the response from the application and uses a machine learning technique to improve the attack methodology. WAVES is better than traditional penetration testing, because it improves the attack methodology, but it cannot thoroughly check all the vulnerable spots like the traditional penetration testing. Instruction-Set Randomization SQLrand [18] provides a framework that allows developers to create SQL queries using randomized keywords instead of the normal SQL keywords. A proxy between the web application and the database intercepts SQL queries and de-randomizes the keywords. The SQL keywords injected by an attacker would not have been constructed by the randomized keywords, and thus the injected commands would result in a syntactically incorrect query. Since SQLrand uses a secret key to modify keywords, its security relies on attackers not being able to discover this key. SQLrand requires the application developer to rewrite code. SANIA [19] for detecting SQL injection vulnerabilities in web applications during the development and debugging phases. Sania intercepts the SQL queries between a web application and a database, and automatically generates elaborate attacks according to the syntax and semantics of the potentially vulnerable spots in the SQL queries. In addition, Sania compares the parse trees of the intended SQL query and those resulting after an attack to assess the safety of these spots. Paros is used for web application security assessment. Paros is written in Java, and people generally used this tool to evaluate the security of their web sites and the applications that they provide on web site. It is free of charge, and using Paros's you can exploit and modified all HTTP and HTTPS data among client and server along with form fields and cookies. In brief the functionality of scanner is as below. According to web site hierarchy server get scan, it checks for server miscount figuration. They add this feature because some URL paths can't be recognized and found by the crawler.

3.3.2 Preventing SQL injection techniques

Framework Support Recent frameworks for web applications provide a functionality that can be used to prevent SQL injections. For example, Struts[20] supports a validator. A validator verifies an input from the user conforms to the pre-defined format of each parameter. If a validator prohibits an input from including meta-characters, we can avoid SQL injections. Since a validator does not transform the dangerous characters to safe ones, we cannot prevent SQL injections if we want to include meta-characters in the input. Prepare Statement SQL provides the prepare statement, which separates the values in a query from the structure of SQL. The programmer defines a skeleton of an SQL query and then fills in the holes of the skeleton at runtime. The prepare statement makes it harder to inject SQL queries because the SQL structure cannot be changed. Hibernate [21] enforces us to use the prepare statement. To use the prepare statement, we must modify the web application entirely; all the legacy web applications must be re-written to reduce the possibility of SQL injections. Queries are intercepted before they are sent to the database and are checked against the statically built models, in dynamic phase. Queries that violate the model are prevented from accessing to the database. The primary limitation of this tool is that its success is dependent on the accuracy of its static analysis for building query models. In SQL Check [22] and SQL Guard [23] queries are checked at runtime based on a model which is expressed as a grammar that only accepts legal queries. SQL Guard examines the structure of the query before and after the addition of user-input based on the model. In SQL Check, the model is specified independently by the developer. Both approaches use a secret key to delimit user input during parsing by the runtime checker, so security of the approach is dependent on attackers not being able to discover the key. In two approaches developer should to modify code to use a special intermediate library or manually insert special markers into the code where user input is added to a dynamically generated query. CANDID [24] modifies web applications written in Java through a program transformation. This tool dynamically mines the programmer-intended query structure on any input and detects attacks by comparing it against the structure of the actual query issued. CANDID's natural and simple approach turns out to be very powerful for detection of SQL injection attacks.

We believe high precision can be achieved by discovering more vulnerabilities and by avoiding the issue of potentially useless attacks that can never be successful, with conducting fewer attacks. To achieve this, we focus on the technique of generating attacks that precisely exploit vulnerabilities. As a result, our approach generates attacks only necessary for identifying vulnerabilities. We propose SQLI-DP technique for detecting SQL injection vulnerabilities. In the output of the web application, which results in making fewer attacks, detecting more vulnerabilities, and making fewer false positives/negatives.

4. Problem Solution

4.1 Proposed technique for SQL Injection Detection and Prevention (SQLI-DP)

In this section, we present SQLI-DP technique, which tests for SQL injection vulnerabilities and prevent SQL injection by the parse of the queries. The general view of SQLI-DP is shown in figure 4 the discrete line if the software developer doesn't active SQLI-DP technique.

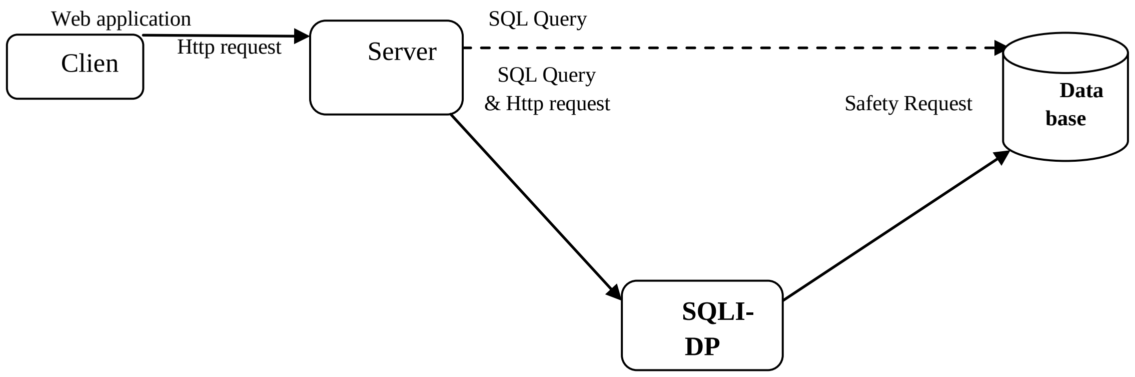


Figure4. General view of SQLI-DP technique

4.1.1 The main contribution of SQLI-DP technique:

1. Detect blind SQL injection using new function.
2. Using parse tree to detect SQL injection using Zql [25] with open source technique.
3. Detect and prevent the SQL injection using this leads to decrease execution time.

Our technique has two options (continue with safety, unsafe option) if we select the safety option activate SQLI-DP. If no (unsafe option) send request immediately to database and execute query. Illustrates the core work of SQLI-DP technique where the three points.

4.1.2 SQLI-DP Technique pseudo code

SQLI-DP technique (SQLIA DETETION & PREVENTION)

```
1.  INPUT: SQL, Http request
2.  OUTPUT: prevent attack & Final Report.
3.   T ← Http request.
4.   Q ← sql statement query.
5.   BEGIN
6.  IF (DETECTION BLIND SQLIA FOREACH T ) THEN
7.      {
8.          PREVENT T
9.          CREAT FINAL REPORT
10.         QUIT :  SQLI-DP TECHNIQUE
11.     }
12.  ELSE IF (DETECTION SQLIA FOREACH Q)
13.         {
14.                PREVENT T
15.                CREAT FINAL REPORT
16.                 QUIT :  SQLI-DP TECHNIQUE
17.
18.         }
```

```
19.     ELSE
20.         {
21.             SEND Q TO DATABASE
22.         }
```

4.1.3 The explanation steps of SQLI-DP technique is presents in details as follows

STEP 1:

Read the HTTP request from server and check if there is blind SQL injection or not as illustrated in [26] some example about blind SQL injection attack we us to find this method to detect blind SQLIA. There are several uses for the Blind SQL Injection:

- Testing the vulnerability.
- Finding the table name.
- Exporting a value.

There are more examples to traditional blind SQL injection and advanced blind SQL injection [26]. We noticed that from last example there are traditional kind of blind SQL injection and advanced kind with regular expression .To testing blind sql injection we scan the http request after capture if it contain the key words [select, top, virsion,user, order, substring , ascii, from ,limit, having and etc.] or regular expression like [*,>,<,$,%,-- ,',[a-z],[A-z],"and etc] . The normal http request not contain like this. Then classify the http request to normal or abnormal request. If the system found Blind sql injection reject input go to step 3, if no continue to step 2.

STEP 2:

Perform SQL validation using validation parse tree. Compare the parse tree generated from an attack request with that generated from an innocent message to verify whether an attack was successful or not. If the parse generated from an innocent request, SQLI-DP determines the attack was successful. Suppose that a web application issues the SQL query "SELECT * FROM users WHERE name=""(vulnerable spot), or '1'='1".we use the technique idea that used in [27] when using a fresh key to user input. If the web application sanitizes the input properly, the parse tree will look like the one shown in figure 1. If not properly sanitized, the parse tree will look like the one shown in figure 3, where the structure of the parse tree is different from figure 1, if the two tree are different detect there is injection. Then classify the query as normal or abnormal and account the False positive (A false positive occurs when the test returns a positive result, but there is actually no fault). Then Prevent abnormal user queries and send normal user queries immediately to database and execute query and send the data to server.

STEP 3:

SQLI-DP generates the report that contains the SQL query with the user input data, if it found blind sql injection and execution time by millisecond. The SQL query benefit in detection to know the kind of SQL injection attack.

The advantages of SQLI-DP are illustrated as follows:

1- It is efficient, because it only adds about 1second overhead to database query costs.
2- In addition, it is easily adopted by software developer.
3- It is suitable for legacy system because it is a technique implemented on server side. It doesn’t need to rewrite old web application because protection from Injection is on server side where database resides.

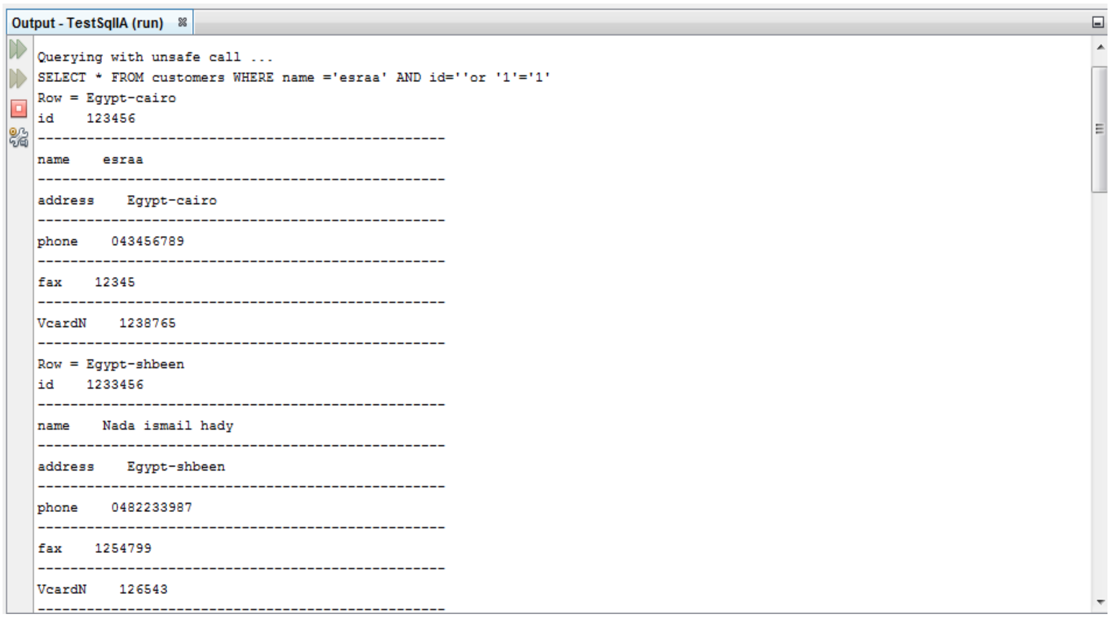
4.2 Experimental Result

The experiments methodology is done by designing MSQL database and a patche server web application in our platform. The SQLI-DP technique is implemented by Java language. We used Zql to implement an SQL parser. The SQLI-DP technique consists of an SQL proxy, and a core component of our technique. The SQL proxy captures the SQL queries. The core component of SQLI-DP performs the tasks described in the previous section. We apply SQLI-DPs in windows7, 32-bit operating system and core (TM) i5 CPU. Some snapshoot screens from SQLI-DP work are shown when using different options of unsafe and safe. If we using unsafe system with SqlIA we noticed that the attack success and the server return all data records in database table as shown in figure 5. SQLI-DP (unsafe option) response all recodes because there are (Tautology injection found). If we using Safety system option SQLI-DP prevents SQLIA in first test if it found blind injection attack it generates report as shown in figure 6. If blind injection attack not found the SQLI-DP tests other types of sql injection attacks. Figures 7 and 8 are show the reports generated after hacking in different types of attack.

This report is benefit in:

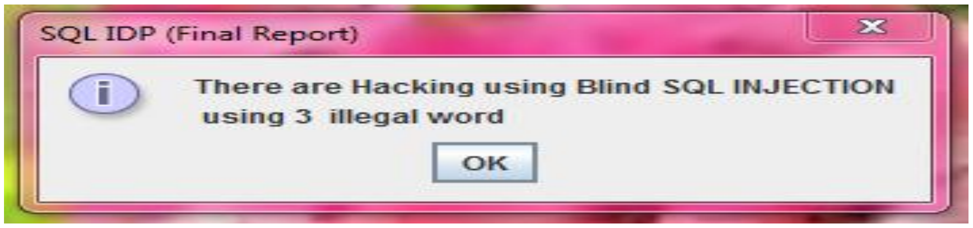
1- To know the type of SQL injection attack.
2- Sure that the system prevents this attack to accessed database.
3- To estimate the execution time of detection and prevention.

As shown in figure 7 this report appears when the SQLI-DP found ***Piggy-Backed Queries*** SQLIAs when try to access the web application database.


Output - TestSqlIA (run)

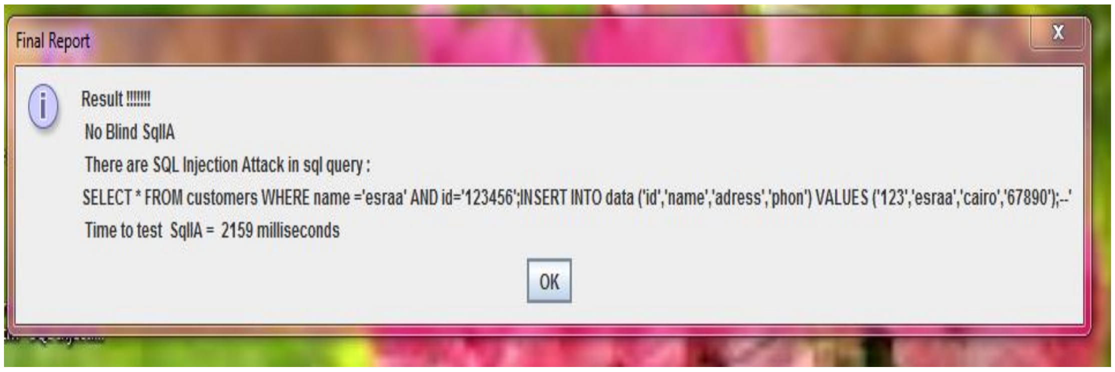
```
Querying with unsafe call ...
SELECT * FROM customers WHERE name ='esraa' AND id=''or '1'='1'
Row = Egypt-cairo
id    123456
---------------------------------------------------
name    esraa
---------------------------------------------------
address    Egypt-cairo
---------------------------------------------------
phone    043456789
---------------------------------------------------
fax    12345
---------------------------------------------------
VcardN    1238765
---------------------------------------------------
Row = Egypt-shbeen
id    1233456
---------------------------------------------------
name    Nada ismail hady
---------------------------------------------------
address    Egypt-shbeen
---------------------------------------------------
phone    0482233987
---------------------------------------------------
fax    1254799
---------------------------------------------------
VcardN    126543
---------------------------------------------------
```

Figure 5. Using unsafe option


SQL IDP (Final Report)

There are Hacking using Blind SQL INJECTION using 3 illegal word

Figure 6. SQLI-DP safety option with Blind sql injection


Final Report

Result !!!!!!

No Blind SqlIA

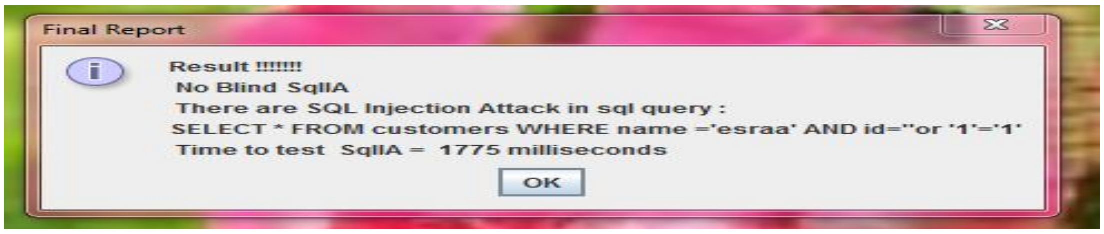
There are SQL Injection Attack in sql query :

SELECT * FROM customers WHERE name ='esraa' AND id='123456';INSERT INTO data ('id','name','adress','phon') VALUES ('123','esraa','cairo','67890');--'

Time to test SqlIA = 2159 milliseconds

Figure 7. SQLI-DP with safety option with *Piggy-Backed Queries* SqlIAs

As shown in figure 8 this report appears when the SQLI-DP find Tautology SqlIAs when try to access the web application database.



```
Result !!!!!!
 No Blind SqlIA
 There are SQL Injection Attack in sql query :
SELECT * FROM customers WHERE name ='esraa' AND id=''or '1'='1'
 Time to test  SqlIA =  1775 milliseconds
```

Figure8. SQLI-DP with safety option with Tautology SqlIAs.

Table1 shows the SQLI-DP techniques with respect to the number of attacks trails we using manual hacking on the database and the execution time in millisecond.

Table 2. SQLI-DP trails and execution time

Types of SQLIA	**Num.of Trials** of attacks	Detection & prevention	**Execution time(ms)**
Tautology	50	(50) 100%	1775
Logically Incorrect Queries	50	(50) 100%	1670
Union Query	50	(50) 100%	1819
Piggy-Backed Queries	50	(50) 100%	1670
Blind injection	40	(40) 100%	102
Alternate Encoding	10	(10) 100%	1560
Stored Procedure	10	(10) 100%	1800

We compare our SQLI-DP technique with Paros (detection technique).

- During our work we observed The SQLI-DP technique detect and prevent the Blind SQLIA as the first check.
- The SQLI-DP technique give the user request query but the Paros technique no.
- The SQLI-DP technique add about 1s addition to the system but Paros technique add 5 second.
- The SQLI-DP technique is Server Side protection technique but Paros technique host side protection.
- As shown in figure 9 the execution time to detect attack in SQLI-DP technique less than Paros technique.

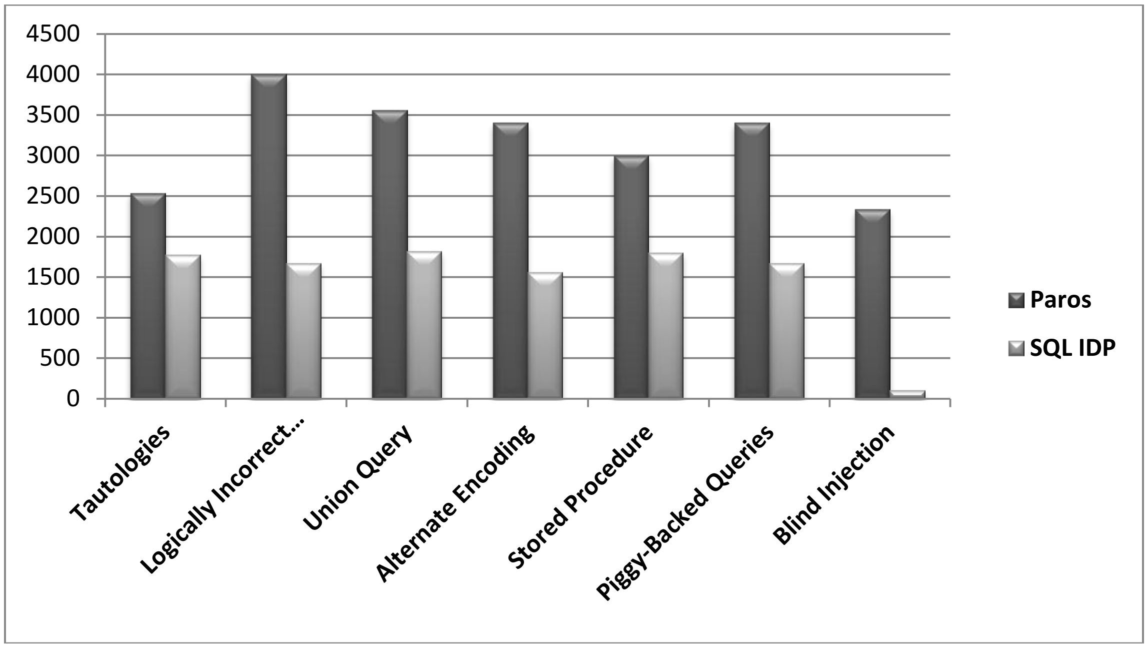


Figure9. Execution time of SQLI-DP and Paros techniques

Through the study of previous observations and discuss the reasons, we find that SQLI-DP by several features not found in other techniques. First, SQLI-DP detects and prevents using less time than Paros for the following reasons:

1- SQLI-DP is server side technique work as a proxy between the server and database but Paros work in host side is a proxy between client and server so SQLI-DP decreases the communication time.
2- SQLI-DP detects and prevents the blind SQL injection attacks but Paros technique detect only.
3- From the observation the SQLI-DP technique add only one second addition to the system but Paros technique add approximately 4 second to the system.
4- The SQLI-DP technique gives the SQL query statement in the final report it's very important to the developer:
   - To know kind of attack.
   - To detect the kind of attacks and use for protect this type of sits from this kind of attacks.
   - To know the advanced schema of SQL injection attacks.
5- SQLI-DP give the final report after prevent the attacks to sure the attacks are prevented.
6- In addition, the SQLI-DP it is easily adopted by software developer because it written by Java language that suitable for any platform .
7- It is suitable for legacy system because it is a technique that implemented on server side. It doesn’t need to rewrite old web application because protection from Injection is on server side where database resides.

5. Conclusion and Future Work

We presented our new server side protection technique against SQL Injection, which is designed to check for SQL injection vulnerabilities in the server side. SQLI-DP intercepts SQL queries and analyize it based on the syntax of the SQL queries using parse tree. It rejects injectable queries from beginning and executes other queries to detect SQL injection. It is suitable for legacy system because it is a technique that is implemented on server side. It doesn't need to rewrite old web application because protection from injection is on server side where database resides. SQLI-DP has two advantages comparing with other scanner. It's efficient, adding about 1second overhead to database query costs. In addition, it is easily adopted by software developer, having the same syntactic structure as current popular record set retrieval method and we detect and prevent blind SQL injection.

In future work, we intend to evaluate SQLI-DPs using different web based applications with real public domain to achieve great accuracy in SQL injection detection and prevention.

6. References

[1] OWASP (2012, September 6). Testing for SQL Injection[Online]. Available: http://www.owasp.org/index.php/Testing for SQL Injection.

[2] Nithya. A Survey on SQL Injection attacks, their Detection and Prevention Techniques. International Journal Of Engineering And Computer Science Volume 2 Issue 4 April,2013 Page No.886-905.

[3] Sayyed Mohammad. Study of SQL Injection Attacks and Countermeasures. International Journal of Computer and Communication Engineering, Vol. 2, No. 5, September 2013 Page No.539-514.

[4] Gopi Krishnan. Preventing Injection Attack by Whitelisting Inputs . Lecture Notes on Information Theory Vol. 1, No. 3, September 2013 Page No.132-134.

[5] W. Halfond and A. Orso, "Combining Static Analysis and Runtime Monitoring to Counter SQL- Injection Attacks," Proceeding of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005), 2005.

[6] Chunhui Song. Song, "SQL Injection Attacks and Countermeasures", California State University, Sacramento, (Spring 2010).

[7] Z. Lashkaripour .A Simple and Fast Technique for Detection and Prevention of SQL Injection Attacks. International Journal of Security and Its Applications Vol.7, No.5 (2013), Page 53-66.

[8] K.R Venugopal, L.M Patnaik , Patnaik,"Detection and prevention of SQL injection attacks", Computer Networks and Intelligent Computing. 5th International Conference on Information Processing, ICIP 2011, Bangalore, India, August 5-7, 2011, pages 104-107.

[9] Atefeh Tajpour, Maslin Masrom, Mohammad Zaman Heydari, Suhaimi Ibrahim,"Evaluation of SQL Injection Detection and Prevention Techniques," 2nd International Conference on Computational Intelligence, Communication Systems and Networks, Liverpool, United Kingdom pages 216-221.

[10] Jason Sabin &Dan Timpson (2006, August). Paros (version 3.2.13) [Online]. Available: http://www.testingsecurity.com/paros_proxy.

[11] Acunetix. Acunetix Web Security Scanner. http://www.acunetix.com/.

[12] IBM. Rational AppScan. http://www.ibm.com/software/awdtools/appscan/.

[13] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring.In Proceedings of the 12th International Conference on World Wide Web, May 2003, pages148–159.

[14] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. SecuBat: A Web Vulnerability Scanner. In Proceedings of the 15th International Conference on World Wide Web, May 2006, pages. 247–256.

[15] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 174–183, 2005.

[16] 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), pages 123–140, 2005.

[17] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring . In Proceedings of the 12th International World Wide Web Conference (WWW03), pages 148–159, 2003.

[18] S. Boyd and A. Keromytis. SQLrand: Preventing SQL injection attacks. In Proceedings of the Applied Cryptography and Network Security (ACNS), pages 292–304, 2004.

[19] Yuji Kosuga, Kenji Kono, Miyuki Hanaoka,(August 2011). Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection, Doctor of Philosophy. Keio University.

[20] Struts. Apache Struts project. http://struts.apache.org/.

[21] Hibernate. hibernate.org. http://www.hibernate.org/.

[22] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. ACM SIGPLAN Notices. Volume: 41, pp: 372-382, 2006.

[23] Gregory T. Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti Using : Parse Tree Validation to Prevent SQL Injection Attacks, Computer Science and Engineering The Ohio State University, pages 4-5,2009.

[24] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, CANDID:Preventing SQL Injection Attacks using Dynamic Candidate Evaluation. Proceedings of the 14th ACM conference on Computer and communications security. ACM, Alexandria,Virginia, USA.page:12-24.

[25] Gibello. Zql: A java sql parser, 2002. In http://www.experlog.com/gibello/zql/.

[26] Simone 'R00T_ATI' Quatrini Marco 'white_sheep' Rondini. Blind Sql Injection with Regular Expressions Attack.

[27] Jagdish Halde. (2008),SQL Injection analysis, Detection and Prevention. Master's Theses and Graduate ResearchSan Jose State University.