*IOSR Journal of Computer Engineering (IOSR-JCE)*
*ISSN: 2278-0661, ISBN: 2278-8727, PP: 56-60*
*www.iosrjournals.org*

# SQLIA: Detection And Prevention Techniques: A Survey

Pushkar Y.Jane[1] , M.S.Chaudhari[2]
[1](CSE Department, Nagpur University, India)
[2](CSE Department, Nagpur University, India)

***ABSTRACT :*** *SQL injection is an attack methodology that targets the data residing in a database through the firewall that shields it. The attack takes advantage of poor input validation in code and website administration. SQL Injection Attacks occur when an attacker is able to insert a series of SQL statements in to a 'query' by manipulating user input data in to a web-based application, attacker can take advantages of web application programming security flaws and pass unexpected malicious SQL statements, Query through a web application for execution by the back-end database. And attacker get full access of the backend database. In this way,SQL Injection Attack performed.*

***Keywords***: *SQL injection, database security, authentication.*

## I. Introduction

Information or Data is the most important business asset in today's environment and achieving an appropriate level of Information Security. SQL-Injection Attacks (SQLIA's) are one of the topmost threats for web application security. For example financial fraud, theft confidential data, deface website, sabotage, espionage and cyber terrorism. To implement security guidelines inside or outside the database the access of the sensitive databases should be monitored. It is a hacking technique in which the attacker adds SQL statements through a web application's input fields or hidden parameters to gain access to resources or make changes to data. The fear of SQL injection attacks has become increasingly frequent and serious. SQL-Injection Attacks are a class of attacks that many of these systems are highly vulnerable to, and there is no known fool-proof defend against such attacks. Compromise of these web applications represents a serious threat to organizations that have deployed them, and also to users who trust these systems to store confidential data. The Web applications that are vulnerable to SQL-Injection attacks user inputs the attacker's embeds commands and gets executed. The attackers directly access the database underlying an application and leak or alter confidential information and execute malicious code . In some cases, attackers even use an SQL Injection vulnerability to take control and corrupt the system that hosts the Web application. The increasing number of web applications falling prey to these attacks is alarmingly high Prevention of SQLIA's is a major challenge. It is difficult to implement and enforce a rigorous defensive coding discipline.

## II. Background

SQLIA's is one of the main issues in database security, which easily affects the database without the knowledge of both the user and the database administrator. It is a technique that may corrupt the information in the database i.e. deletes or changes the full database or records or tables. To exploit the database system, some vulnerable web applications are used by the attackers. These attacks not only make the attacker to breach the security and steal the entire content of the database but also, to make arbitrary changes to both the database schema and the contents. SQL injection attack could not be realized about information compromization until long after the attack has passed in many scenarios, the victims are unaware that their confidential data has been stolen or compromised. SQL Injection attacks can be performed by attackers just with the help of simple web browser. The following section describes the attacks with an example.Generally the Authenticated users have username and password such as,
Username: kalia
Password: 1000
The SQL Query format will be as follows,

Select * from table where username=’kalia’ and
pwd=’1000’;
The above query then retrieves the needed records from the database where username and pwd is available in the database or it shows some error messages to the browsers. The unauthorized users or the attackers inject the following SQL Injection in this field:
Username: kalia
Password: 1000
Then the dynamic SQL query constructed from the above information is,Select * from table where username=’kalia’ and pwd=1000;In this SQL statement, the actual username is ‘kalia’. And the modified name did by the attackers while generating the Query is ‘kalia’. Here both the words are same considering whether they are characters or not. But the first word ‘kalia’ is fully taken in the font “Times new Roman” format, Where as the modified name given by the attacker, ‘kalia’ includes the alphabets ‘j’, ‘o’, ‘n’ in ‘Times new Roman’ format, and the alphabet ‘h’ in “Wide Latin” format. Hence, the attacker will now easily attack the database just by transferring the malicious code. Here both the usernames are set of characters so that even by using the Character Level Tainting, all the characters in the given query is matched with the strings in the Meta Strings Library and then the query was sent to the database. Here the Character Level Tainting can’t be able to find the type of malicious code that was sent in different fonts because of having no information on the consideration of the font formats of the sent query and hence it performs the operation. The result of this query performs SQL Injection attacks.

## 2.1.Types of SQLIA

**Tautologies**: This type of attack injects SQL tokens to the conditional query statement to be evaluated always true. This type of attack used to bypass authentication control and access to data by exploiting vulnerable input field which use WHERE clause.
"SELECT * FROM employee WHERE userid = '118' and password ='aaa' OR '1'='1'"
As the tautology statement (1=1) has been added to the query statement so it is always true.
**Union Query***:* By this technique, attackers join injected query to the safe query by the word UNION and then can get data about other tables from the application. Suppose for our examples that the query executed from the server is the following:
*SELECT* Name, *Phone FROM Users WHERE Id=$id*
. By injecting the following Id value:
*$id=1 UNION ALL SELECT credit Card Number,1 FROM Credit sysTable*
We will have the following query:
*SELECT Name, Phone FROM Users WHERE Id=1 UNION ALL SELECT creditcardNumber,1 FROM Credit sys Table*
which will join the result of the original query with all the credit card users.
**Stored Procedure**: Stored procedure is a part of database that programmer could set an extra abstraction layer on the database. As stored procedure could be coded by programmer, so, this part is as inject able as web application forms. Depend on specific stored procedure on the database there are different ways to attack. In the following example, attacker exploits parameterized stored procedure.
*CREATE* PROCEDURE *DBO .is Authenticated*
*@user Name varchar2, @pass varchar2, @pin int*
*AS EXEC("SELECT accounts FROM users*
*WHERE login=’" +@user Name+ "’ and pass=’"*
*+@password+ "’ and pin=" +@pin);*
*GO* For authorized/unauthorized user the stored procedure returns true/false. As an SQLIA, intruder input “ ’ ; SHUTDOWN; - -” for username or password. Then the stored procedure generates the following query:
*SELECT accounts FROM users WHERE login=’doe’ AND pass=’ ’; SHUTDOWN; -- AND pin=*

After that, this type of attack works as piggy-back attack. The first original query is executed and consequently the second query which is illegitimate is executed and causes database shut down. So, it is considerable that stored procedures are as vulnerable as web application code.

**llegal/Logically Incorrect Queries**: when a query is rejected , an error message is returned from the database including useful debugging information. This error messages help attacker to find vulnerable parameters in the application and consequently database of the application. In fact attacker injects junk input or SQL tokens in query to produce syntax error, type mismatches, or logical errors by purpose. In this example attacker makes a type mismatch error by injecting the following text into the *pin* input field:

1) Original URL: http://www.arch.polimi.it/eventi/?id_nav=8864

2) SQL Injection: http://www.arch.polimi.it/eventi/?id_nav=8864'

3) Error message showed:

SELECT name FROM Employee WHERE id =8864\' From the message error we can find out name of table and fields: name; Employee; id. By the gained information attacker can organize more strict attacks.

**Piggy-backed Queries**: In this type of attack, intruders exploit database by the query delimiter, such as ";", to append extra query to the original query. With a successful attack database receives and execute a multiple distinct queries. Normally the first query is legitimate query, whereas following queries could be illegitimate. So attacker can inject any SQL command to the database. In the following example, attacker inject " 0; *drop table user* " into the *pin* input field instead of logical value. Then the application would produce the query:

*SELECT info* FROM *users WHERE login='doe' AND pin=0; drop table users*

Because of ";" character, database accepts both queries and executes them. The second query is illegitimate and can drop *users* table from the database. It is noticeable that some databases do not need special separation character in multiple distinct queries, so for detecting this type of attack, scanning for a special character is not impressive solution.

**Inference:** By this type of attack, intruders change the behaviour of a database or application.There are two wellknown attack techniques that are based on inference: blind injection and timing attacks.

**Blind** Injection: Sometimes developers hide the error details which help attackers to compromise the database. In this situation attacker face to a generic page provided by developer, instead of an error message. So the SQLIA would be more difficult but not impossible. An attacker can still steal data by asking a series of True/False questions through SQL statements. Consider two possible injections into the login field:

*SELECT accounts FROM users WHERE login='doe' and 1=0 -- AND pass= AND pin=0*

*SELECT accounts FROM users WHERE login='doe' and 1=1 -- AND pass= AND pin=0*

If the application is secured, both queries would be unsuccessful, because of input validation. But if there is no input validation, the attacker can try the chance. First the attacker submit the first query and receives an error message because of "*1=0*". So the attacker does not understand the error is for input validation or for logical error in query. Then the attacker submits the second query which always true. If there is no login error message, then the attacker finds the login field vulnerable to injection.

**Timing Attacks**: A timing attack lets an attacker gather information from a database by observing timing delays in the database's responses. This technique by using if-then statement cause the SQL engine to execute a long running query or a time delay statement depending on the logic injected. This attack is similar to blind injection and attacker can then measure the time the page takes to load to determine if the injected statement is true. This technique 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, in the following query:

*declare @s varchar(8000) select @s = db_name() if*
*(ascii(substring(@s, 1, 1)) & ( power(2, 0))) > 0 waitfor*
*delay '0:0:5'*

Database will pause for five seconds if the first bit of the first byte of the name of the current database is 1. Then code is then injected to generate a delay in response time when the condition is true. Also, attacker can ask a series of other questions about this character. As these examples show, the information is extracted from the database using a vulnerable parameter.

**Alternate Encodings**: In this technique, attackers modify the injection query by using alternate encoding, such as hexadecimal, ASCII, and Unicode. Because by this way they can escape from developer's filter which scan input queries for special known "bad character". For example attacker use *char* (44) instead of single quote that is a bad character. This technique with join to other attack techniques could be strong, because it can target different layers in the application so developers need to be familiar to all of them to provide an effective defensive coding to prevent the alternate encoding attacks. By this technique, different attacks could be hidden in alternate encodings successfully. In the following example the *pin* field is injected with this string: *"0; exec (0x73587574 64 5f77 6e),"* and the result query is:

*SELECT accounts FROM users WHERE login=" AND pin=0; exec (char(0x73687574646f776e))*

This example use 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 the ASCII hexadecimal encoding of the attack string. This encoded string is translated into the *shutdown* command by database when it is executed.

## III. Related Work

In order to protect a Web application from SQL Injection attacks, there are two major concerns. Firstly, there is a great need of a mechanism to detect and exactly identify SQL Injection attacks. Secondly, knowledge of SQL Injection Vulnerabilities (SQLIVs) is a must for securing a Web application. So far, many frameworks have been used and/or suggested to detect SQLIVs in Web applications. Here, we mention the some existing prominent solutions and their working methods.

**William G.J.Halfond et al.'s Scheme**- This approach works by combining static analysis and runtime monitoring. In its static part, technique uses program analysis to automatically build a model of the legitimate queries that could be generated by the application. In its dynamic part, technique monitors the dynamically generated queries at runtime and checks them for compliance with the statically-generated model. Queries that violate the model represent potential SQLIAs and are thus pre- vented from executing on the database and reported.

**SAFELI** – Proposes a Static Analysis Framework in order to detect SQL Injection Vulnerabilities. SAFELI framework aims at identifying the SQL Injection attacks during the compile-time. This static analysis tool has two main advantages. Firstly, it does a White-box Static Analysis and secondly, it uses a Hybrid-Constraint Solver. For the White-box Static Analysis, the proposed approach considers the byte-code and deals mainly with strings. For the Hybrid-Constraint Solver, the method implements an efficient string analysis tool which is able to deal with Boolean, integer and string variables.

**Thomas et al.'s Scheme** - Thomas et al., in suggest an automated prepared statement generation algorithm to remove SQL Injection Vulnerabilities. They implement their research work using four open source projects namely: (i) Net-trust, (ii) ITrust, (iii) WebGoat, and (iv) Roller. Based on the experimental results, their prepared statement code was able to successfully replace 94% of the SQLIVs in four open source projects.

**Ruse et al.'s Approach** - Ruse et al. propose a technique that uses automatic test case generation to detect SQL Injection Vulnerabilities. The main idea behind this framework is based on creating a specific model that deals with SQL queries automatically. Adding to that, the approach identifies the relationship (dependency) between sub-queries. Based on the results, the methodology is shown to be able to specifically identify the causal set and obtain 85% and 69% reduction respectively while experimenting on few sample examples.

**Ali et al.'s Scheme** - Adopts the hash value approach to further improve the user authentication mechanism. They use the user name and password hash values SQLIPA (SQL Injection Protector for Authentication) prototype was developed in order to test the framework. The user name and password hash values are created and calculated at runtime for the first time the particular user account is created

**Parse Tree Validation Approach** - Buehrer et al. adopt the parse tree framework. They compared the parse tree of a particular statement at runtime and its original statement. They stopped the execution of statement unless there is a match. This method was tested on a student Web application using SQLGuard. Although this approach is efficient, it has two major drawbacks: additional overheard computation and listing of input (black or white).

**Dynamic Candidate Evaluations Approach** - In, Bisht et al. propose CANDID. It is a Dynamic Candidate Evaluations method for automatic prevention of SQL Injection attacks. This framework dynamically extracts the query structures from every SQL query location which are intended by the developer (programmer). Hence, it solves the issue of manually modifying the application to create the prepared statements.

**Su and Wassermann propose SQLCheck model** which statically analyzed SQLIA by generating finite state automata. They use this approach to model set of valid SQL commands for each data access. This approach is based on Context-Free-Grammars (CGFs) for validating data. They use this approach by wrapping user input in special markers, e.g., (|a|). The grammar of the guest language is then augmented to accept the markers by using some symbols in the grammar, for instance, so that it accept (|'a'|) in SQL whenever a string literal is accepted. This way, an injection attack would then fail to parse. For example, SELECT*FROM customer WHERE userid = kalia AND passwd=("OR _a'='a' "|), there is no production that allows an arbitrary condition inside the markers. However, it is wrong to assume that markers will not be leaked since Web applications can ―echo SQL queries to the user if an error occurs.

## IV. Conclusion

It is obvious from above description that SQL injection attacks are one of the largest classes of security problems. Most existing technique either require developers to manually specify the interfaces to an application or, if automated, are often inadequate when applied to modern, complex web applications. In this paper we have surveyed the most popular existing SQL Injections attack issues. And also we have presented a survey report on various types of SQL Injection attacks, their working methods, detection and prevention techniques.

## References


[1] Indrani Balasundaram, Dr.E.Ramaraj *"An Approach to Detection of SQL Injection Attacks in Database Using Web Services"(IJCSNS,VOL. 11 No.1,January 2011).*

[2] Rahul Shrivastava, Joy Bhattacharyji, Roopali Soni *"SQL INJECTION ATTACKS IN DATABASE USING WEB SERVICE: DETECTION AND PREVENTION – REVIEW" Asian Journal Of Computer Science And Information Technology 2: 6 (2012) 162 – 165*. Also Available at http://www.innovativejournal.in/index.php/ajcsit.

[3] Shubham Srivastava, Rajeev Ranjan Kumar Tripathi *"Attacks Due to SQL Injection & Their Prevention Method for Web-Application" (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 3 (2) , 2012,3615-3618*.

[4]Prasant Singh Yadav, Dr pankaj Yadav, Dr. K.P.Yadav *"A Modern Mechanism to Avoid SQL Injection Attacks in Web Applications" (IJRREST Volume-1 Issue-1, June 2012)*

[5] V.Shanmughaneethi ,S.Swamynathan *"Detection of SQL Injection Attack in Web Applications using Web Services" (ISSN : 2278-0661 Volume 1, Issue 5 (May-June 2012), PP 13-20).*

[6] William G.J.Halfond and Alessandro Orso *"AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks"*

[7] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. *"A Static Analysis Framework for Detecting SQL Injection Vulnerabilities", COMPSAC 2007, pp.87-96, 24-27 July 2007.*

[8] S. Thomas, L. Williams, and T. Xie, *"On automated prepared statement generation to remove SQL injection vulnerabilities." Information and Software Technology 51, 589–598 (2009).*

[9] A.SRAVANTHI, K.JAYASREE DEVI,K.SUDHA REDDY, A.INDIRA, V.SATISH KUMAR *"DETECTING SQL INJECTIONS FROM WEB APPLICATIONS" [IJESAT Volume-2, Issue-3, 664 – 671].*

[10] Diallo Abdoulaye Kindy and Al-Sakib Khan Pathan *"A SURVEY ON SQL INJECTION: VULNERABILITIES, ATTACKS, AND PREVENTION TECHNIQUES"*

[11] Shaukat Ali, Azhar Rauf, Huma Javed *"SQLIPA:An authentication mechanism Against SQL Injection"*

[12] M. Ruse, T. Sarkar and S. Basu *"Analysis & Detection of SQL Injection Vulnerabilities via Automatic Test Case Generation of Programs." 10th Annual International Symposium on Applications and the Internet pp. 31 – 37 (2010)*

[13] Sruthi Bandhakavi,Prithvi Bisht,P. Madhusudan,V.N. Venkatakrishnan *"CANDID: Preventing SQL Injection Attacks usingDynamic Candidate Evaluations"*