SQL Injection Attacks Types, Prevention and Detection Techniques

Shakti Bangare
Department of Information Technology
Acropolis Institute of Technology and Research, Indore, Madhya Pradesh, India

Abstract— SQL Injection Attacks (SQLIAs) is the oldest approach to harm the web applications. According to Acunetix Web application Vulnerability Report published in 2019, SQLIAs is approximate 11% of overall vulnerability [1]. As we know dynamic web applications are interactive web applications from which a user can gain some information and he can also store some information though that application. So web applications have a database to maintain user's information. To deal with databases SQL is used at server end. A developer design some query to make interaction between user and database but if developer not validates the SQL query then a malicious user can insert some valid SQL code through web page and gain access to the database which was not intentionally designed by the developer. Using SQLIAs a malicious user can steal some personal information, perform some unintentionally financial operation and modify & destroy information from database. There are various ways of SQLIAs by which a malicious user gain access to the databases and there are many solutions for prevention and detection of SQLIAs.

Keywords: SQL Injection Attacks, vulnerability, detection, prevention, web application attacks

I. INTRODUCTION

Today designing of web applications is not a major issue; rather security of these applications is the issue. There are various ways by which any malicious user can harm web applications. One of the possible vulnerable attacks is SQL Injection Attacks (SQLIAs). According to Acunetix Web application Vulnerability Report 2019 SQLIAs are reducing in successive years but numbers are positive. It is oldest approach for vulnerability and approximate 11% of overall vulnerability [1]. It is kind of attack in which an unauthorized or malicious user take advantage of insecure SQL query while designing the database for the Web applications. Using SQL Injection Attacks (SQLIAs), any malicious user can perform some tasks which are not intentionally programmed by the programmer for that web application. Stealing personal information, performing some unintentionally financial operation, unintentionally modifying and destroying information from data base are example of SQLIAs.

Dynamic web applications have dynamic web pages and interact with databases at server site to store numerous information provided by user from client site. A user can perform all possible operations in databases using web applications which are permitted by Query Language e.g., a user can view along with insert, delete and modify some information in databases using dynamic web applications. Application user provides information using input box displayed on the web pages at client side. At server site, a query is formed using information provided by application user and executed; so insertion, deletion, modification and view information operations are performed on databases.

SQLIAs are possible only in dynamic web applications. Instead of providing valid data to the input box, a malicious user can insert some SQL keywords to access some unauthorized data and if website is not validated properly then SQLIA may be done. For example a website is designed to access the database as:

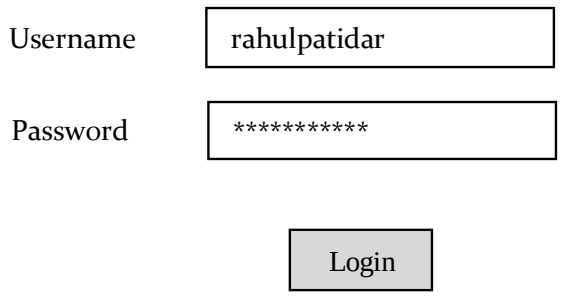

Fig. 1.1: Login Form

Programmer designed this web form to access a website if a user inserts valid username and password. Query working on backend may be:

"select * from walletuser where username= '" +request.getParameter("uid") +"' and password='" +request.getParameter("pwd") +"'";

Where uid acquire value, inserted in text box in front of username and pwd acquire value, inserted in text box in front of password field.

However a malicious user can enter "rahulpatidar" in the username field and "' OR '1'='1" in the password field, so the query string becomes:

"select * from walletuser where username= 'rahulpatidar' and password='' OR '1'='1'";

So this query will always be executed whether information given is right or wrong.

II. DEFINITION OF SQLIA

An SQLIA is a kind of attack in which a malicious user tries to access the database using input boxes on the website. He supplies valid SQL syntax but that was not intentionally designed by the programmer and gain access to the database. By the use of SQL Injection attacks a malicious user can view the unauthorized data, he can insert, delete and update the data also would not be permitted by the application developer.

III. SQL INJECTION ATTACK TYPES

There are different types of SQL Injection attacks. It depends on the malicious user what action he wanted to perform on database so select the type of attack. Following classification of SQLIAs is presented according to Tajpour and Massrum [2][3][4][5].

A. Tautology:

This type of attack is performed on conditional query statements. Conditional query have WHERE clause where conditions are checked and if evaluates true then gain access

to database. Malicious user insert some valid SQL code so query always returns true value, so query always get executed whether information given is correct or not. It is already explained in Introduction part for *walletuser* table by making a query as a tautology attack.

B. Illegal/ Logical Incorrect Queries:

When database reject a query, a query message is returned. This query message might be some useful information. If an application programmer not deals with these exceptions in proper manner and returns the same message, whatever returned by the database then a malicious user can form an illegal query and if database returns some useful information like table name or some other table's field then he can form further serious queries which might be used to perform some harmful operations on database. In the above figure 1.1, instead of providing valid username rahulpatidar, if he enters an invalid input abc# then an error message is returned by the database. For example error message returned is:

Error: select username, password from walletuser where username= abc#

By above error message a user can get table name and table's field and using this information a malicious user can perform some more accurate illegal task and can harm the database.

C. Union Queries:

In this type of attack a malicious user attach an injected query with the correct query and then may get the data about the others table from the application. For example the query working at server is:

Select name, phoneno from user where username='+request.getparameter("usr")+"";

Value of username is getting by usr field which will be inserted by the client using web site. A malicious user can insert the following in username field at client side: usernameame='hariom' union all select empname, empdept, 1 FROM emp

Resulting query will become:

select name, phoneno, from user where username='hariom' union all select empname, empdept, 1 FROM emp.

This query will join the result of the second query with result of the first query. Sensitive information can be stolen by the use of union query like credit card information, personal information etc.

D. Piggy-Backed Queries:

In this type of attack, more queries can be attached with original one. Intruders attach more queries with original one by the query delimiter, such as ";". A successful attack database receives and executes a multiple distinct queries. Normally the first query is legitimate query, whereas the following queries could be illegitimate.

Example Consider this query

1) Original query:

Select * from EMP where EMPNAME='hariom' and EMPDEPT='CS'

2) Injected query:

Select * from EMP where EMPNAME='hariom' and EMPDEPT='CS' ; drop table EMP

Because ';' is accepted by the SQL and it separates the two given queries. Both the queries are acceptable using delimiter ';'. Using ';' multiple queries can be appended in single one. In above example of injected query, malicious user will drop *EMP* table. Dropping the table *EMP* is not available for the user because it is not intentionally programmed. Programmer intentionally designed only first query.

E. Stored Procedures:

Stored procedures are parts of database so that programmer could set an extra abstraction layer on the database. As stored procedure could be coded by programmer, this part can be injected as web application forms. Depending on specific stored procedure on the database, there are different ways to attack. In the following example, attacker attacks parameterized stored procedure.

Example Consider the following stored procedure

CREATE PROCEDURE DBO.isAuthenticated
@username varchar2, @password varchar2, @name= varchar2
AS
EXEC("Select * from login where username='" +@username + "' and password ='" +@password +"' and name = '" +@name + """);
GO

Like simple query, stored procedure returns true/ false values. After getting values, it works as normal query. Any SQLIAs intruder can have the following value for username or password parameter to inject database "'; SHUTDOWN; -- ";

Query becomes

Select * from login where username='; SHUTDOWN; -- ' and password ='" +@password +"' and name = '" +@name + ""

After getting values, it will work as piggy backed query. Above code divides one query into two. First one Select * from login where username='' will execute nothing and second one SHUTDOWN which is system keyword. All other code followed by -- will comment all the things.

F. Inference:

By this type of attack, intruders change the behavior of a database or application. There are two well-known attack techniques that are based on inference:

Blind Injection: Sometimes developers hide the error details which help attackers to compromise the database. In this situation attacker face a generic page provided by developer, instead of an error message. 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.

Example Consider two possible inject code in username field

"Select * from login where username='hariom' or '1'='0' -- and password ='' and name = '""

"Select * from login where username='hariom' or '1'='1' -- and password ='' and name = '""

If application developer performed all the validation checks, then application is secured and both the queries would be unsuccessful. But if there is no input validation then attacker has chance to infect the database. First time attacker

submits the query but it will display error message. Because of '1'='0', attacker does not know that error is logical error or input validation error. Second time attacker submits the query and if attacker gets no error and successfully login, then it can be harmful for database.

1) 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 causes the SQL engine to execute a long running query or a time delay statement depending on the logic injected. *WAITFOR*, a keyword along with branches is used to delay the responses of the database.

G. Alternate Encodings:

In this technique, attackers modify the injection query by using alternate encoding, such as hexadecimal, ASCII and Unicode. By this way, they can escape from developer's filter which scans input queries for specially known "bad character". For example, attacker uses char (44) instead of single quote that is a bad character. This technique can be jointly used with other techniques of SQLIAs. Consider the following:

Example: Let, after submitting query by attacker query be the following values:

"Select * from login where username='hariom'; exe(char(0x73687574646f776e)) -- and password ='' and name = ''"

Attacker submits a query which uses alternate encoding using piggy backed query. Two queries will form after submitting values. Second query uses the char() function and hexadecimal encoding which will lead to SHUTDOWN statement in the database to shutdown the database.

IV. Detection, Prevention and Avoidance Mechanisms

A lot of work has been done and there are various solutions invented to detect and prevent SQLIAs. It is tried to cover most of the important solutions to detect and prevent SQLIAs. All solutions are described below

A. Web Application Vulnerability and Error Scanner (WAVES):

Huang and colleagues proposed one open source tool which is available at http://waves.sourceforge.net named Web Application Vulnerability and Error Scanner (WAVES) [6]. They designed this tool purely based on software engineering approach. They used a black-box testing technique for web applications to detect SQL injection vulnerabilities. This tool can be applied when web application would have been developed. Based on knowledge, it selects the best injection pattern to test web application. It builds attack pattern that utilizes machine learning.

In this tool, they developed a crawler which checks behavior of the application which works on reverse engineering. This crawler loads the all web pages of the application and simulates just like original application. Various injection patterns are applied as input for this crawler. If any vulnerability is found, this crawler halts immediately and then audits the information. This tool identifies all points in a web application that can be used to insert SQLIAs. This tool can be used to check dynamic behavior of web applications.

B. Tautology Checker:

Wassermann and Su proposed [7] use of static analysis to stop tautology attack. Analysis is performed to check "WHERE" clause of the query where a tautology may be presented. Finite state machine (FSM) checks the tautology.

For example let there is one query

"Select column1 from table1"

This query can be checked by FSM explained below in *Figure*

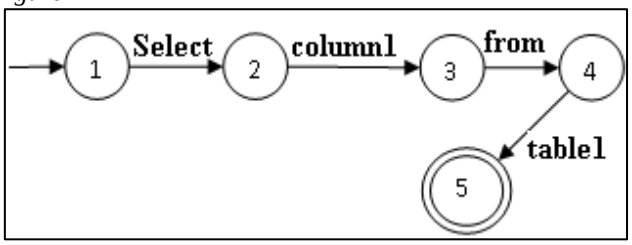

Fig. 4.1 Finite State Machine for checking Query

C. Static Analysis Framework for discovEring sqL Injection vulnerability (SAFELI):

Black Box testing tool cannot detect all the types of vulnerabilities like SQL symbols as user input. Xiang Fu and Kai Qian [8] proposed a static SQL testing tool named Static Analysis Framework for discovEring sqL Injection vulnerability (SAFELI). It identifies SQLIA vulnerabilities at compile time. This tool checks vulnerabilities in the form of SQL keywords and SQL symbols. In this tool SAFELI, there is a function named message ( ) which performs checks on user inputs. It has two parts; first part checks all the suspicious keywords and second deals with symbols such as single quote. These injection patterns for message ( ) is designed using Java Symbolic Execution engine (JavaSye). It has two modules: a Java byte code instrument and a symbolic execution engine. Java bye code instrument injects additional logic into the target Java byte code so that it can be executed by the symbolic execution engine. Various injection patterns in the form of symbolic representation are generated.

D. CANdidate evaluation for Discovering Intent Dynamically (CANDID):

Sruthi Bandhakavi and colleagues proposed CANDID [9]. CANDID stands for CANdidate evaluation for Discovering Intent Dynamically. This tool 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. It checks all the access points and modifies structure of the intended query dynamically based on input which has to be provided. CANDID's natural and simple approach turns out to be very powerful for detection of SQL injection attacks.

E. SQLrand:

In SQLrand [10] instead of normal SQL query, developers create queries using randomization of keyword. In this approach, web application has queries in randomized form. When user provides input for these queries, a proxy filter intercepts queries to the database and de-randomizes the

keywords. By using the randomized instruction, attacker's injected code could not have been constructed because code inserted by the intruder will be de-randomized by the proxy and then no valid query will be produced and injection would be detected. As it uses a secret key to modify instructions, security of the approach is dependent on attacker ability to seize the key. It requires the integration of a proxy for the database in the same system as developer training. Proxy takes some processing time to de-randomize the query.

F. SQL Guard and SQL Check:

In SQL Guard [11] queries are checked at runtime based on a model which is expressed as parse tree. A grammar is constructed for the intended query and query is expressed in the form of parse tree. SQL Guard examines the structure of the query before and after the addition of user-input based on the model. If structure of the query before and after accepting the input is the same, then query is not injected; otherwise, query is injected. If any user tries to inject the query, then structure of query after the accepting input, will change and tool will discard this query due to injection.

In SQL Check [12], the model is specified independently by the developer. In this approach, developer built an augmented grammar for intended query. Left and right end maker are used to describe augmented grammars. If any intruder tries to inject query augmented grammar will not produce a valid parse tree which is intended and injection would be detected.

Both techniques use a secret key to check 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 these approaches programmer should 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.

G. SQL Domain Object Model (SQL DOM) and Safe Query Objects:

SQL DOM [13] stands for SQL Domain Object Model. It provides one level of abstraction for the SQL statements. Using SQL DOM applications do not need to manipulate SQL statements; instead they can use SQL DOM which encapsulates all the statements which have to be executed.

A Safe Query Object [14] is just an object containing a Boolean method that can be used to filter a collection of candidate objects. It checks for the possible values received from the user.

In both the strategies, they encapsulates for trustable access to databases. They use a type-checked API which causes systematic query building process. Consequently by API, they apply coding best practices such as input filtering and strict user input type checking. The drawbacks of the approaches are that developer should learn new programming paradigm or query-development process.

H. Positive Tainting:

Positive tainting [15] is identifying, marking and tracking of trusted data. Negative tainting is identification of un-trusted data [15]. In this paper, the author proposed to identify the positive taint because positive taint comes from limited sources which are able to track; but if we identify negative taint which is difficult to achieve because injection information may come from various sources and although we will track all the sources & attacker will find some new sources to inject. It is automated and needs developer intervention. Moreover, this approach benefits from syntax-aware evaluation, which gives developers a mechanism to regulate the usage of string data based not only on its source, but also on its syntactical role in a query string. This method may miss some positive taints also because application developer should be an expert in this field but due to time constraint in software industry it is not always possible.

I. Web application Security by Static Analysis and Runtime Inspection (WebSSARI):

WebSSARI [16] stands for Web application Security by Static Analysis and Runtime Inspection. It is a framework for existing scripting languages. Primarily, it is designed for PHP but can be implemented for other scripting languages also. WebSSARI automatically inserts guards in potentially insecure part of the code. Any sensitive data will be checked against these guards. It uses static analysis to check taint flows against preconditions for sensitive functions. It works based on sanitized input that has passed through a predefined set of filters. The limitation of approach is adequate preconditions for sensitive functions which cannot be accurately expressed so some filters may be omitted.

J. Intrusion Detection System (IDS):

IDS [17] use an Intrusion Detection System (IDS) to detect SQLIAs, based on a machine learning technique. The technique builds models of the typical queries and then at runtime, queries that do not match the model would be identified as attack. Query which is to be fired on database is checked through IDS. IDS build a parse tree which has SQL keywords and user input as terminal. This parse tree is checked with the profiles which are saved in this system. Each profile has some anomaly score. If this parse tree is analogous to profile, then that profile is selected. During the parse tree building, an anomaly score is calculated; if this score exceeds the limit of selected profile, then injection is detected.

1) Sania:

Sania [18] is an automated tool to prevent SQLIAs. In Sania, a web application developer sends innocent HTTP requests to identify the behaviour of the application. All the vulnerable spots are identified using this request. Sania constructs a local query which is identical to the HTTP request. Sania generates dummy attacks on these vulnerable spots and analyzes the result. A parse tree is constructed to sanitize the inject-able query. All kinds of attacks are identified by this parse tree. The novelty of Sania lies in that it exploits the syntactical knowledge of the SQL queries to generate attack requests. If any programmer does not have sound knowledge of syntax of DBMS, then sanitization cannot be done correctly and all types of vulnerabilities cannot be resolved.

K. Query Tokenization:

Lambert and Lin proposed query tokenization [19] method. This method uses concept of tokenization. Length of tokens are calculated and saved in application for intended query. After getting request from the client end, length of tokens are

again calculated and compared with length of tokens for intended query. If length is the same for both tokenization, then query is permitted; otherwise, not permitted. In this method, tokenization is done on the basis of space, single quote or double dash. All the strings before these characters form a token. These tokens are stored in an array and token is index to the array.

L. Random4:

Srinivas Avireddy and colleagues proposed a solution named random4 [20] which is also based on encryption- decryption mechanism. They proposed a new algorithm for encryption and idea is little different from PSQLIA-AES. In their solution, they limited user input only 72 characters (26 uppercase letters, 26 lowercase letters, 0-9 digits and 10 special symbols).

	R[1]	R[2]	R[3]	R[4]
a	;	l	x	W
…				
z	7	k	@	U
A	i	J	)	0
…				
Z	M	6	f	.
0	9	B	g	“
…				
9	c	R	j	l
@	(	a	5	0
…				
-	6	a	H	K

Table 4.1: Lookup Table for Random4

To encrypt values, they have criteria so that random value chooses for each character and inject-able values are encrypted. See *Table 4.1* for 4 random values for each character. These values are stored in encrypted form in database. If in future these values are used as some check criteria, then values inserted from user end are first encrypted; then values are checked with values in database because database contained these values in encrypted form using Random4 algorithm. Refer Fig 7[26]. Figure shows that they encrypted the values at client end which is again a load on client side to keep information of encryption algorithm. It can be stolen and it may increase load on client site which may be issued for mobile users.

M. BIT-SLICE-AES:

Piyush Mittal and Sanjay Kumar Jena proposed BIT SLICE-AES Algorithm [21]. This algorithm works on above principle of RANDOM 4. All inject-able code is encrypted using BIT SLICE-AES algorithm and stored in encrypted form in database. They implemented it on 64 bit microprocessor. This algorithm depends on bit configuration of microprocessor. This algorithm needs low level functionalities of microprocessor’s architecture but for small web applications, it is not efficient. It is a specific algorithm so user does not have other choices to implement encryption algorithms.

It totally depends on Hardware functionalities like logically shift, add, subtract, multiply etc. So a programmer should also have sound knowledge of microprocessor’s bit operations. Lacking this knowledge may cause malfunctions.

N. Encryption Algorithm:

Bangare and Prajapati proposed a solution based on encryption algorithm [22]. They stored information provided by user during registration in encrypted form using a standard encryption algorithm and when a user want to gain access to the database, information provided by user using web page again encrypted and if matched with the stored encrypted data, user gain access to the database otherwise he will be a malicious user.

V. Conclusion

SQLIAs are the oldest way to harm the dynamic web application. SQLIA can also be performed in various ways already discussed in this paper. There are many solutions to detect and prevent SQLIAs, already summarized in this paper.

References

[1] https://cdn2.hubspot.net/hubfs/4595665/Acunetix_web_application_vulnerability_report_2019.pdf
[2] Atefeh Tajpour and Maslin Massrum, Comparison of SQL Injection Detection and Prevention Techniques. In proceeding of 2nd International Conference on Education Technology and Computer (ICETC) 2010.
[3] W. G. Halfond, J. Viegas and A. Orso, “A Classification of SQL Injection Attacks and Countermeasures”, Proc. of the Intl. Symposium on Secure Software Engineering, Mar. 2006
[4] Atefeh Tajpour and Mohammad JorJor zade Shooshtari : Evaluation of SQL Injection Detection and Prevention Techniques. Proceeding of International Conference on Second International Conference on Computational Intelligence, Communication Systems and Networks.
[5] Diallo Abdoulaye Kindy and Al-Sakib Khan Pathan : A Survey on SQL Injection: Vulnerabilities, Attacks, and Prevention Techniques. Proceeding of 2011 IEEE 15th International Symposium on Consumer Electronics.
[6] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. Proceedings of the 11th International World Wide Web Conference (WWW 03), May 2003.
[7] G. Wassermann and Z. Su.: An Analysis Framework for Security in Web Applications. Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004).
[8] Xiang Fu, Kai Qian: SAFELI–SQL Injection Scanner Using Symbolic Execution. Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications. (2008).
[9] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations, 2007, Alexandria, Virginia, USA, ACM.

[10] S. W. Boyd and A. D. Keromytis: SQLrand: Preventing SQL Injection Attacks. In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference, June 2004.
[11] Gregory T. Buchrer, Bruce W. Weide, and Paolo A. G. Sivilotti: Using Parse Tree Validation to Prevent SQL Injection Attacks. International Workshop on Software Engineering and Middleware (SEM), 2005.
[12] Zhendong Su, Gary Wassermann: The Essence of Command Injection Attacks in Web Applications. Proceeding of POPL '06 January 11–13, 2006, Charleston, South Carolina, USA.
[13] R. McClure and I. Kr¨uger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. Proceedings of the 27th International Conference on Software Engineering (ICSE 05), 2005.
[14] W. R. Cook and S. Rai. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), 2005.
[15] W. G. Halfond, A. Orso, Using Positive Tainting and Syntax Aware Evaluation to Counter SQL Injection Attacks, 14th ACM SIGSOFT international symposium on Foundations of software engineering 2006, ACM.
[16] Y. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo: Securing Web Application Code by Static Analysis and Runtime Protection. Proceedings of the 12th International World Wide Web Conference (WWW 04), May 2004.
[17] F. Valeur, D. Mutz, and G. Vigna: A Learning-Based Approach to the Detection of SQL Attacks. Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), Vienna, Austria, July 2005.
[18] Yuji Kosuga, Kenji Kono, Miyuki Hanoaka, Hiyoshi Kohoku-ku, Yokohama, Miho Hishiyama and Yu Takahama: Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection. Proceeding of 23rd Annual Computer Security Application Conference IEEE 2007.
[19] NTAGWABIRA Lambert, KANG Song Lin: Use of Query Tokenization to detect and prevent SQL Injection Attacks. IEEE 2010.
[20] S. Avireddy,V. Perumal, N. Gowraj, R. S. Kannan, P. Thinakaran, S. Ganapathi, J. R. Gunasekaran and S. Prabhu: Random4: An Application Specific Randomized Encryption Algorithm to prevent SQL injection. Proceeding of 11th International Conference of IEEE on Trust, Security and Privacy in Computing and Communications, June 2012.
[21] Piyush Mittal, Sanjay Kumar Jena: A Fast and Secure Way to Prevent SQL Injection Attacks. Proceeding of IEEE Conference on Information and Communication Technologies (ICT 2013), April 2013.
[22] Shakti Bangare, G. L. Prajapati: A Generalized Way to Prevent SQL Injection Attacks (SQLIAs) Based on Encryption Algorithms. Proceeding of International Journal of Emerging Technology and Advanced Engineering (IJETAE), Volume 4 Issue 7 July 2014.

