ijecs open access www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 2 Issue 4 April, 2013 Page No. 886-905

A Survey on SQL Injection attacks, their Detection and Prevention Techniques

V. Nithya[1], R.Regan[2], J.vijayaraghavan[3]
nithyahimalini@gmail.com[1], Pondicherry Engineering College [1], Puducherry[1], India
reganr1985@gmail.com[2], University College of Engineering[2], Panruti, Anna University[2], India
vijay.j.raghavan@gmail.com[3], Manakula Vinayagar Institute of Technology College[3], Puducherry[3], India

Abstract
SQL injection is a technique that exploits a security vulnerability occurring in the database layer of an application. The vulnerability is present when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly executed.SQL injection is a trick to SQL query or command as an input possibly via the web pages. They occur when data provided by user is not properly validates and is included directly in a SQL query. By leveraging these vulnerabilities, an attacker can submit SQL commands directly access to the database. In this paper we present all SQL injection attack types and also different technique and tools which can detect or prevent these attacks .Finally we assessed addressing all SQL injection attacks type among current technique and tools.

Key–Words: SQL injection attacks, prevention, detection, Web Application.

I. Introduction

As soon as the services of Internet are rising; all web applications are depended on the Internet. Example: online banking, university admissions, shopping, and various government activities. So, we can say that these activities are the key component of today's Internet Infrastructure. Web Applications are the applications that can be accessed over the Internet by using any web browser that runs on any operating system and architecture. They have become ubiquitous due to the convenience, interoperability, flexibility, and availability that they provide. Web Applications are vulnerable to a variety of new security threats. SQLIAs are one of the most significant of such threats. SQLIAs are increasing continuously and posy very serious security risks because they can give attackers unrestricted access to the database that lie under web applications.

Information is the most important business asset today and achieving an appropriate level of "Information security" can be viewed as essential in order to maintain a competitive edge [38]. SQL Injection Attacks (SQLIAs) is considered as one of the top 10 web application vulnerabilities of 2010 by the Open Web Application Security Project (OWASP)[46], Semiannual Report (July to December 2010) from the Web Hacking Incidents Database (WHID)[44] shows that that SQL injection are consistently or near the top 21% of the reported vulnerabilities in 2010 ,consider as top 2 attack and recently in August, 2011, Hacker steals user records from Nokia Developer Site using "SQL injection"[47]. They are easy to detect and exploit; that is why SQLIAs are frequently employed by malicious user for different reasons. E.g. financial fraud, theft, confidential data, deface website, sabotage, espionage, cyber terrorism, or simply for fun. Throughout 2010, Government, Finance and Retail verticals faced different, but equally important, outcomes. Attacks against Government agencies resulted in defacement in 26% of SQL injection attacks, while Retail was most affected by credit card

leakage at 27% of SQL injection and finance experienced monetary loss in 64% of attacks [44]. Furthermore, SQL Injection attack techniques have become more common more ambitious, and increasingly sophisticated, so there is a deep to need to find an effective and feasible solution for this problem in the computer security community. Detection or prevention of SQLIAs is a topic of active research in the industry and academia. To achieve those purposes, automatic tools and security system have been implemented, but none of them are complete or accurate enough to guarantee an absolute level of security on web application. One of the important reasons of this shortcoming is that there is lack of common and complete methodology for the evaluation either in terms of performance or needed source code modification which in an over head for an existing system. A mechanism which will easily deployable and provide a good performance to detect and prevent the SQL injection attack is essential one**.**

II. OVERVIEW OF SQL INJECTION ATTACK

SQL (Structured Query Language) is a textual language used to interact with relational Database. The typical unit of execution of SQL is the 'query', which is a collection of statements that typically return a single 'resultset'. SQL statements can modify the structure of databases and manipulate the contents of databases by using various DDL, DML commands respectively. SQL Injection occurs when an attacker is able to insert a series of SQL statements into a query by manipulating data input into an application [39].

A. Definition of SQLIA

Most web applications today use a multi-tier design, usually with three tiers: a presentation, a processing and a data tier. The presentation tier is the HTTP web interface, the application tier implements the software functionality, and the data tier keeps data structured and answers to requests from the application tier. Meanwhile, large companies developing SQL-based database management systems rely heavily on hardware to ensure the desired performance. SQL injection is a type of attack which the attacker adds Structured Query Language code to input box of a web form to gain access or make changes to data. SQL injection vulnerability allows an attacker to flow commands directly to web applications underlying database and destroy functionality or confidentiality.

B.SQL Injection Attacks (SQLIA) Process

SQLIA is hacking technique which the attacker adds SQL statements through a web application's input field or hidden parameter to access to resources. Lack of input validation in web applications causes hacker to be successful. Basically SQL process structured in three phases:

i. An attack sends the malicious HTTP request to the web application.
ii. Create the SQL Statements.
iii. Submits the SQL statements to the back end database

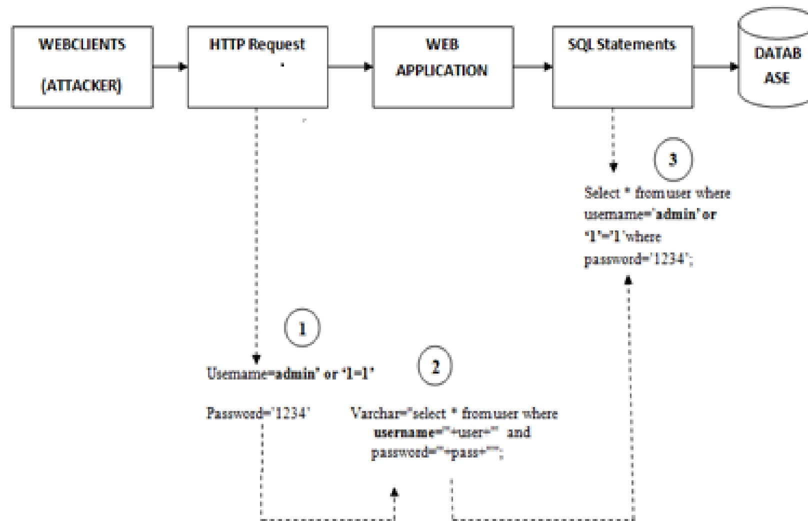

Figure: 1. Example for SQL injection attack data flow.

C. Consequence of SQLIA

The result of SQLIA can be disastrous because a successful SQL injection can read sensitive can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administrative operations on the Database (such as shutdown the DBMS), recover the content on the DBMS file system and execute commands (xp cmdshell) to the operating system. The main consequences of these vulnerabilities are attacks on[36]:

i) Authorization Critical data that are stored in a vulnerable SQL database may be altered by a successful SQLIA, a authorization privilege.

ii) Authentication If there is no any proper control on username and password inside the authentication page , it may be possible to login to a system as a normal user without knowing the right username and/or password.

iii) Confidentially Usually databases are consisting of sensitive data such as personal information, credit card numbers and/ or social numbers. Therefore loss of confidentially is a big problem with SQL Injection vulnerability. Actually, theft of sensitive data is one of the most common intentions of attackers.

iv) Integrity By a successful SQLIA not only an attacker reads sensitive information, but also, it is possible to change or delete this private information.

D. Classification of SQLIA

An SQL injection attack has a set of properties, such as assets under threat, vulnerabilities being exploited and attack techniques utilized by threat agents.

i) By Attacker Intent

An important classification of SQLIA is related to the attacker's intent, or in other words, the goal of the attack.

a) Extracting data This category of attacks tries to extract data values from the back end database. Based on the type of web application, this information could be sensitive, for example, credit card numbers, social numbers; private data are highly valuable to the attacker. This kind of intent is the most common type of SQLIA.

b) Adding or modifying data The purpose of these attacks is to add or change data values within a database.

c) Performing database finger printing In this category of attack the malicious user wants to discover technical information on the database such as the type and version that a specific web application is using. It is noticeable that certain types of databases respond differently to different queries and attacks, and this information can be used to "fingerprint" the database. Once the intruder knows the type and the version of the database it is possible to organize a particular attack to that database.

d) By passing authentication By this attack, intruders try to bypass database and application

authentication mechanisms. Once it has been over passed, such mechanisms could allow the intruder to assume the rights and privileges associated with another application user.

e) Identifying injects able parameters Its goal is to explore a web application to discover which parameters and user-input fields are vulnerable to SQLIA. By using an automated tool called a "vulnerabilities scanner" this intent can be identified.

f) Determining database schema The goal of this attack is to obtain all the database schema information (such as table names, column names, and column data types). This is very useful to an attacker to gather this information to extract data from the database successfully. Usually by exploiting specific tools such as penetration testers and vulnerabilities scanners this goal is achieved.

e) Performing denial of service In these category intruders make interrupt in system services by performing some instruction so the database of a web application shutdown, thus denying service happens. Attacks involving locking or dropping database tables also fall into this category.

ii) Vulnerabilities

a) Insufficient Input Validation Input validation is an attempt to verify or filter any input for malicious behavior. Insufficient input validation will allow code to be executed without proper verification of its intention. Attacker taking advantages of insufficient input validation can utilize malicious code to conduct attacks [37].

b) Privileged account A privileged account has a degree of freedom to do what normal accounts cannot. Its action may also exempt from auditing and validation. This present vulnerability since a jeopardized privileged account, such as an administrator account, can compromise much more than what a jeopardized regular account can.

c) Extra Functionalit: Extra functionalities meant to provide a broader range of vulnerability, since combinations of this functionality may result in unintended actions. For example, xp_cmdshell is meant to provide users with a way executing operating system commands, but commonly used to added unauthorized users into the operating system.

iii). Asserts

Asserts are information or data an unauthorized threat agent attempt to gain.

a) Database Server Fingerprint The database server fingerprints contains information about the database system in use. It indentifies the specific type and version of the database, as well as the corresponding SQL language dialect. A compromise of this asset may allow attackers to construct malicious code specifically for the SQL language dialect in question.

b) Database Schema The database schema describes the server's internal architecture Database Structure information such as table names, size and relationships are defined in the data schema. Keeping this asset private is essential in keeping the confidentiality and integrity of the database data .A compromise in the database schema may allow attackers to know the exact structure of the database, including table, rows and column headings.

c) Database Data T he database data is the most crucial asset in any database system. It contains the information in the tables described in the database schema, such as prices in an online store, personal information of clients, administrator passwords, etc. A compromise in the database data will usually result in failure of the system's intended functionality, thus, its confidentiality and integrity must be protected.

d) Network A network interconnects numerous hosts together and allows communication between them. A compromise in a network will most likely compromise every host in the network. Some networks may also be interconnected with other networks, furthering the potential damage, should an attack be successful.

E). Method logy for a Successful SQLIA

Attack techniques are the specific means by which a threat agent carries out attacks using malicious code. Threat agent may use many different methods to achieve their goals, often combing several of these sequentially or combing several or employing them in different varieties[39].

i) Tautologies

Attack Intent: Bypassing authentication, identifying inject able parameters, extracting data.

Description: A SQL tautology is a statement that is always true. Tautology-based SQL injection attacks are usually used to bypass user authentication or to retrieve unauthorized data by inserting a tautology

into a conditional statement. A typical SQL tautology has the form “or <comparison expression>”, where the comparison expression uses one or more relational operators to compare operands and generate an always true condition. The general goal of a tautology-based attack is to inject SQL tokens that cause the query’s conditional statement to always evaluate the true.
For example**,** SELECT * FROM user WHERE id=’1’ or ‘1=1’-‘AND password=’1234’;The “or 1=1” is the most commonly known tautology.

ii) Piggy-backed Query

Attack Intent: Extracting data, adding or modifying data, performing denial of service, executing remote commands

Description: In the piggy-backed Query attacker tries to append additional queries to the original query string. On the successful attack the database receives and executes a query string that contains multiple distinct queries. In this method the first query is original whereas the subsequent queries are injected. This attack is very dangerous; attacker can use it to inject virtually any type of SQL command. For example, SELECT * FROM user WHERE id=’admin’ AND password=’1234’; DROP TABLE user; --’; Here database treats above query string as two query separated by “;” and executes both. The second sub query is malicious query and it causes the database to drop the user table in the database.

iii) Logically Incorrect

Attack Intent: Identifying inject able parameters, performing database finger-printing, extracting data.

Description**:** This attack takes advantage of the error messages that are returned by the database for an incorrect query. These database error messages often contain useful information that allow attacker to find out the vulnerable parameter in an application and the database schema. For example, SELECT * FROM user WHERE id=’1111’ AND password=’1234’ AND CONVERT (char, no); the purpose of this attack is to collect the structure and information of CGI.

iv) Union query*:*

Attack Intent**:** Bypassing Authentication, extracting data.

Description :Union query injection is called as statement injection attack. In this attack attacker insert additional statement into the original SQL statement. This attack can be done by inserting either a UNION query or a statement of the form “;< SQL statement >” into vulnerable parameter. The output of this attack is that the database returns a dataset that is the union of the results of the original query with the results of the injected query. For example, SELECT * FROM user WHERE id=’1111’ UNION SELECT * FROM member WHERE id=’admin’ --’ AND password=’1234’;

v) Stored Procedure

Attack Intent: Performing privilege escalation, performing denial of service, executing remote commands.

Description: In this technique, attacker focuses on the stored procedures which are present in the database system. Stored procedures run directly by the database engine. Stored procedure is nothing but a code and it can be vulnerable as program code. 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: For example, SELECT accounts FROM users WHERE login= '1111' AND pass='1234 '; SHUTDOWN;--; This type of attack works as piggyback 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 [21].

vi) Inference

Attack Intent: Identifying injectable parameters, extracting data, determining database schema

Description*:* By this type of attack, intruders change the behaviour of a database or application. There are two well known 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:

For example, SELECT accounts FROM users WHERE id= '1111' 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 submits 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, declare @ 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.

vii) Alternate Encodings

Attack Intent: Evading detection.

Description**:** 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 badcharacter. 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 5*f*177 6e), " and the result query is:

SELECT accounts FROM users WHERE login='' AND pin=0; exec (char (0x73687574646j776e))

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 detect and prevent SQL Injection attacks, filtering and other detection methods are being researched. This section explains the related work.

Black Box Testing Huang and colleagues [5] propose WAVES, a black-box technique for testing Web applications for SQL injection vulnerabilities. The technique uses a Web crawler to identify all points in a Web application that can be used to inject SQLIAs. It then builds attacks that target such points based on a specified list of patterns and attack techniques. WAVES then monitors the application's response to the attacks and uses machine learning techniques to improve its attack methodology. This technique improves over most penetration-testing techniques by using machine learning approaches to guide its testing. However, like all black-box and penetration testing techniques, it cannot provide guarantees of completeness.

WebSSARI WebSSARI [9] use 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 cannot be accurately expressed so some filters may be omitted.

SecuriFly SecuriFly [23] is tool that is implemented for java. Despite of other tool, chase string instead of character for taint information and try to sanitize query strings that have been generated using tainted input but unfortunately injection in numeric fields cannot stop by this approach. Difficulty of identifying all sources of user input is the main limitation of this approach.

Static Code Checkers JDBC-Checker is a technique for statically checking the type correctness of dynamically generated SQL queries [6, 7]. This technique was not developed with the intent of detecting and preventing general SQLIAs, but can nevertheless be used to prevent attacks that take advantage of type mismatches in a dynamically-generated query string. JDBC-Checker is able to detect one of the root causes of SQLIA vulnerabilities in code improper type checking of input. However, this technique would not catch more general forms of SQLIAs because most of these attacks consist of syntactically and type correct queries. Wassermann and Su propose an approach that uses static analysis combined with automated reasoning to verify that the SQL queries generated in the application layer cannot contain a tautology [19]. The primary drawback of this technique is that its scope is limited to detecting and preventing tautologies and cannot detect other types of attacks.

Dynamic Analysis Dynamic analysis, unlike static analysis, can locate vulnerabilities of SQL injection attacks without making any a adjustments to web applications. Open source program Paros [45] scans not only SQLIAs vulnerabilities, but also other vulnerabilities within the web application. Paros is not perfect because it uses predetermined attack codes to scan and uses HTTP response to the success-rate of the attack. Sania [24] finds and collects SQL injection attack vulnerabilities between the web application and databases. Then, it proceeds to generate SQL Injection attack codes. After attacking with the generated code, it collects the SQL query from the attack. After the normal SQL query is compared and analyzed with the SQL query collected from the attack using the parse tree, the success rate of the attack is verified. Since a parse tree is used, Sania is more accurate than using HTTP response verification. Yonghee Shin [42] proposed to use Input Flow Analysis and input validation analysis to build a white-box, and generated test input data to locate SQL Injection vulnerabilities. Dynamic analysis method is advantageous because no web application adjustments are necessary. However, the vulnerabilities found in the web application must be manually fixed by the developers and not all of them can be found without predefined attacks.

***Combined Static and Dynamic Analysis*:** AMNESIA is a model-based technique that combines static analysis and runtime monitoring [12, 13]. In its static phase, AMNESIA uses static analysis to build models of the different types of queries an application can legally generate at each point of access to the database. In its dynamic phase, AMNESIA intercepts all queries before they are sent to the database and checks each query against the statically built models. Queries that violate the model are identified as SQLIAs and prevented from executing on the database. In their evaluation, the authors have shown that this technique performs well against SQLIAs. The primary limitation of this technique is that its success is dependent on the accuracy of its static analysis for building query models. Certain types of code obfuscation or query development techniques could make this step less precise and result in both false positives and false negatives.

SQLCHECK: Su and Wassermann [10] implement their algorithm with SQLCHECK on a real time environment. It checks whether the input queries conform to the expected ones defined by the programmer. A secret key is used for the user input delimitation. The analysis of SQLCHECK shows no false positives or false negatives. Also, the overhead runtime rate is very low and can be implemented directly in many other Web applications using different languages

SQLrand Boyd, Keromytis [11] proposed SQLrand which uses instruction set randomization of SQL statement to check SQL injection attack. It uses a proxy to a append key to SQL keyword. A de-

randomizing proxy then converts the randomized query to proper SQL queries for the database. The key is not known to the attacker, so the code injected by attacker is treated as undefined keywords and expressions which cause runtime exceptions and the query is not sent to database. The disadvantage of this system is its complex configuration and the security of the key. If the key is exposed, attacker can formulate queries for successful attack.

Thomas et al.'s Scheme SQL provides the prepare statement[40], 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 programmer defines a skeleton at runtime. The prepare statement makes it harder to inject SQL queries because the SQL queries because the SQL structure cannot be changed. Thomas et al., in [26] 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) I Trust, (iii) Web Goat, 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. To use the prepare statement, we must modify the web application must be rewritten to reduce the possibility of SQL injection.

SQLIA Prevention Using Stored Procedures Stored procedures are subroutines in the database which the applications can make call to [17]. The prevention in these stored procedures is implemented by a combination of static analysis and runtime analysis. The static analysis used for commands identification is achieved through stored procedure parser and the runtime analysis by using a SQLChecker for input identification.[9] Proposed a combination of static analysis and runtime monitoring to fortify the security of potential vulnerabilities.

Proxy Filters Security Gateway [8] is a proxy filtering sys- tem that enforces input validation rules on the data flowing to a Web application. Using their Security Policy Descriptor Language (SPDL), developers provide constraints and specify transformations to be applied to application parameters as they flow from the Web page to the application server. Because SPDL is highly expressive, it allows developers considerable freedom in expressing their policies. However, this approach is human-based and, like defensive programming, requires developers to know not only which data needs to be filtered, but also what patterns and filters to apply to the data

***Intrusion Detection Systems*:** Valeur and colleagues [25] pro- pose the use of an Intrusion Detection System (IDS) to detect SQLIAs. Their IDS system is based on a machine learning technique that is trained using a set of typical application queries. The technique builds models of the typical queries and then monitors the application at runtime to identify queries that do not match the model. In their evaluation, Valeur and colleagues have shown that their system is able to detect attacks with a high rate of success. However, the fundamental limitation of learning based techniques is that they can provide no guarantees about their detection abilities because their success is dependent on the quality of the training set used. A poor training set would cause the learning technique to generate a large number of false positive and negatives.

Ali et al.'s Scheme Ali et al.'s Scheme [1] adopts the hash value approach to further improve the user authentication mechanism. They use the user name and password as 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.

Ruse et al.'s Approach*:* In [27], 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.

Shin et al.'s approach suggests SQLUnitGen, a Static-analysis-based tool that automate testing for identifying input manipulation vulnerabilities*:* [28]. The authors apply SQLUnitGen tool which is compared with FindBugs, a static analysis tool. The proposed mechanism is shown to be efficient as regard to the fact that false positive was completely absent in the experiments

CANDID Bishtet al [3] proposed 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.

SQL DOM Scheme SQL DOM framework is suggested by McClure and Kruger in [14]. They closely consider the existing flaws while accessing relational databases from the OOP (Object-Oriented Programming) Languages point of view. They mainly focus on identifying the obstacles in the interaction with the database via CLIs (Call Level Interfaces). SQL DOM object model is the proposed solution to tackle these issues through building a secure environment for communication.

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[29]. 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.
\
Parse Tree Validation Approach Buehrer et al. [20] 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).

Swaddler Swaddler [18] analyzes the internal state of a web application. It works based on both single and multiple variables and shows an impressive way against complex attacks to web applications. First the approach describes the normal values for the application's state variables in critical points of the application's components. Then, during the detection phase, it monitors the application's execution to identify abnormal states.

DIWeDa approach Roichman and Gudes [30] propose IDS (Intrusion Detection Systems) for the backend databases. They use DIWeDa, a prototype which acts at the session level rather than the SQL statement or transaction stage, to detect the intrusions in Web applications. The proposed framework is efficient and could identify SQL injections and business logic violations too.

Positive Tainting and Syntax Aware Evaluation*:* In this approach [15] valid input strings are initially provided to the system for detection of SQLIA. At runtime, it categorizes input strings and propagates the untrusted or other than trusted markings based on the initialization. After that, a 'syntax aware evaluation' is performed for evaluating the propagated strings. Thus, based on the evaluation, if untrusted strings are found, such queries are restricted from passing into the database server for processing. During initialization of the trusted strings, it performs identification and marking based on inputs. The strings are categorized as: (i) hard coded strings, (ii) strings implicitly created by Java and (iii) strings originated from external sources. In case of syntax-aware evaluation, it performs syntax evaluation at the database interaction point. Syntax defines the trust policies which are the functions defined by the web programmer. Functions perform pattern matching and if the result of matching gives positive outcome, the tool allows the query to be executed on the database server. Following issues are there in this method - (i) Initialization of trusted strings are developers dependent and (ii) Persistent storage of trusted strings may cause second order attack [41].

Context Sensitive String Evaluation (CSSE) The basic idea behind this approach is to find out the root cause of SQLIA [31]. The root cause is the origin of the data (information about the data, termed as metadata) i.e., user-provided or developer-provided. Thus, any data provided by the user is marked as untrusted and data provided by the applications are termed as trusted. The untrusted metadata are used for syntactic analysis based on 'Context Sensitive String Evaluation (CSSE)'. Injection may also occur due to programming flaws during developments. CSSE is basically based on syntactical analysis, which first distinguishes string constants (for e.g., select *from users where login='$login_name') and numerical constants (e.g., select * from users where pin=$pin). It then removes all unsafe characters (un-escaped quotes) in alphanumeric identifiers and non-numeric characters in numeric identifiers. This operation is performed before sending the query to the database server. Following issues are there in this approach (i) Initialization of the unsafe characters is dependent on the web programmer, and (ii) Removal of unsafe characters restricts the application functionality.

SQL Prevent SQL Prevent [22] is consists of an HTTP request interceptor. The original data flow is modified when SQL Prevent is deployed into a web server. The HTTP requests are saved into the current thread-local storage. Then, SQL interceptor intercepts the SQL statements that are made by web application and pass them to the SQLIA detector module. Consequently, HTTP request from thread-local storage is fetched and examined to determine whether it contains an SQLIA. The malicious SQL statement would be prevented to be sent to database, if it is suspicious to SQLIA.

Combinatorial Approach R. Ezumalai and G. A-2009 [32] used a signature based technique against SQL Injection Attacks. In this technique, they used three modules to detect security issues. A monitoring module which takes input from web application and sent to analysis module. An analysis module which finds out the hotspots from application, it uses Hirschberg algorithm. Hirschberg algorithm is a string comparison algorithm which works on divide and conquer rule. It stores all the keywords in the specifications module.

Manual Approaches MeiJunjin highlights the use of manual approaches in order to prevent SQLI input manipulation flaws. In manual approaches, defensive programming and code review are applied[33]. In defensive programming: an input filter is implemented to disallow users to input malicious keywords or characters. This is achieved by using white lists or black lists. As regards to the code review [43], it is a low cost mechanism in detecting bugs; however, it requires deep knowledge on SQLIAs.

Automated Approaches Besides using manual approaches, MeiJunjin [33] also highlights the use of automated approaches. The author notes that the two main schemes are: Static analysis FindBugs and Web vulnerability scanning. Static analysis FindBugs approach detects bugs on SQLIAs, gives warning when an SQL query is made of variable. However, for the Web vulnerability scanning, it uses software agents to crawl, scans Web applications, and detects the vulnerabilities by observing their behavior to the attacks.

Specification-Based Approach Kemalis and Tzouramanis [4] proposed a specification-based approach to detect SQL injection attacks. This technique is based on the assumption that an injected statement and the intended statement of the program have different structures. Therefore, a comparison of their structures can tell if the submitted statement is malicious. The specifications used to describe the intended structure of all the application-generated statements. The specifications describe the rules about what syntactic structure an application-generated SQL query should follow in order to be considered as legitimate [4]. Kemalis and T zouramanis created specifications for their applications using Extended Backus Naur Form (EBNF) based on the ISO/IEC SQL database language criteria.

Fine-grained Access Control Scheme In [16], Roichman and Gudes, in order to secure Web application databases, suggest using a fine-grained access control to Web databases. They develop a new method based on fine-grained access control mechanism. The access to the database is supervised and monitored by the built-in database access control. This approach is efficient in the fact that the security

and access control of the database is transferred from the application layer to the database layer. This is a solution of the vulnerability of the SQL session traceability. Besides that, it is a framework which is applicable to almost all database applications. Therefore, it significantly decreases the risk of attacks at the backend of the database application.

***Haixia and Zhihong's Database Security Testing Scheme*:** In [34], Haixia and Zhihong propose a secure database testing design for Web applications. They suggest a few things; firstly, detection of potential input points of SQL Injection; secondly, generation of test cases automatically, then finally finding the database vulnerability by running the test cases to make a simulation attack to an application. The proposed methodology is shown to be efficient as it was able to detect the input points of SQL Injection exactly and on time as the authors expected. However, after analyzing the scheme, we find that the approach is not a complete solution but rather it needs additional improvements in two main aspects: the detection capability and the development of the attack rule library.

Detection based on removing SQL query attribute values Inyong Lee, Soonki Jeong [2] proposed an approach to detect SQL injection attacks is based on static and dynamic analysis. This method removes the attribute values of SQL queries at runtime (dynamic method) and compares them with the SQL queries analyzed in advance (static method) to detect the SQL injection. When run the application each dynamical generated query is compared or performs XOR operation with fixed query if it results zero then that particular query allowed to the database and if it not results to zero then that query reported as abnormal query stop sending to database.

IV EVALVATION

In this section, the SQL injection detection or prevention techniques presented in section III would be compared. We first consider which attack types each technique is able to address. For the subset of techniques that are based on code improvement, we look at which defensive coding practices the technique helps enforce. Finally, we evaluate the deployment requirements of each technique.

A) Comparative Analysis: It would be difficult to give which scheme or approach is the best as each one has some proven benefits for specific types of settings (i.e., systems). Hence, in this section, we note down how various schemes work against the identified SQL Injection attacks. For the purposes of the comparison, we divide the techniques into two categories: detection and prevention techniques. Prevention techniques are techniques that statically identify vulnerabilities in the code and Detection techniques are techniques that detect attacks mostly at runtime. Table 1 shows a chart of the schemes and their defense and prevention capabilities against various SQLIAs. The symbol "•" is used for technique that can successfully stop all attacks of that type. The symbol "×" is used for technique that is not able to stop attacks of that type. The symbol "o" refers to technique that stop the attack type only partially because of limitations of the underlying approach. Though many approaches have been identified as detection or prevention techniques, only few of them were implemented in practicality. Hence, this comparison is not based on empirical experience but rather it is an analytical evaluation. Table 2, illustrates the addressing percentage of SQL Injection attacks among SQL Injection prevention or detection techniques. The percentage of techniques that stop Tautology is calculated by this formula:

$$\frac{\text{Number techniques that can stop tautology}}{\text{Total number of techniques}} \times 100$$

$$\frac{13}{22} \times 100 = 59$$

X= 59% where X denotes percentage of techniques that stop Tautology attack.

Two attack types alternate encodings and stored procedures, caused problems for most techniques. With stored procedures, the code that generates the query is stored and executed on the database. Almost all the types attack have steadily addressed by techniques except Stored Procedure, which cannot be stopped by some techniques. It is evident that only 22%of techniques can stop Stored Procedure ,on other hand 59% of current techniques can stop tautologies, Inference and 54% of current techniques can stop Piggy backed, Logically/Incorrect, Union queries completely. It interesting that almost steadily, 31%of attack types

could be addressed by these techniques partially .Nevertheless it is unfortunate that 45% of techniques cannot stop Store Procedure and Alter Encoding with 45% stopping and 22% non stopping by preventive and detective techniques.

TABLE 1: COMPARISON OF SQLI DETECTION AND PREVENTION TECHNIQUES WITH RESPECT TO ATTACK TYPES

Attack Types	Detection techniques															Prevention Techniques						
	Swaddler [18]	SQL Prevent [22]	SQLrand[11]	SAFELI [29]	SQLIPA[1]	SQLGuard[11]	SQLCheck [10]	Tautology Checker [19]	IDS[25]	Detection based on removing SQL query	DIWeDa [30]	CSSE[31]	CANDID[3]	Automated Approaches[33]	AMNESIA[12,13]	WebSSARI[9]	WAVES[5]	SQLDOM[14]	Security Gateway[8]	SecuriFly[23]	Positive Tainting[15]	JDBC Checker[6,7]
Tautologies	o	•	•	×	•	•	•	•	o	•	×	•	o	•	•	•	o	•	o	o	•	o
Piggy-backed	o	•	•	•	×	•	•	×	o	•	×	•	o	•	•	•	o	•	o	o	•	o
Logically / Incorrect	o	•	•	•	×	•	•	×	o	•	×	•	o	•	•	•	o	•	o	o	•	o
Union	o	•	•	•	×	•	•	×	o	•	×	•	o	•	•	•	o	•	o	o	•	o
Stored Procedure	o	•	×	•	×	×	×	×	o	•	×	×	o	×	×	•	o	×	o	o	•	o
Inference	o	•	•	•	×	•	•	×	o	•	•	•	o	•	•	•	o	•	o	o	•	o
Alternate Encodings	o	•	•	•	×	•	•	×	o	•	×	×	o	×	•	•	o	•	o	o	•	o

TABLE 2: PERCENTAGE OF SQL INJECTION DETECTION OR PREVENTION TECHNIQUES

S.No.	Attack types	Technique that can stop all attacks of that type(•)	Technique that can stop the attack only partially(o)	Technique that is not able to stop attacks of that type(×)
1	Tautologies	59	31	9

2	Piggy-backed	54	31	13
3	Logically / Incorrect	54	31	13
4	Union	54	31	13
5	Stored Procedure	22	31	45
6	Inference	59	31	9
7	Alternate Encodings	45	31	22

ii) Evaluation of Prevention Techniques with Respect to Defensive Coding Practices: **Our initial evaluation of the** techniques against various attack types indicates that the prevention techniques perform well against most of these attacks. We hypothesize that this result is due to the fact that many of the prevention techniques are actually applying defensive coding best practices to the code base.

TABLE 3: ANALYSIS OF CODE IMPROVEMENT METHODS WITH RESPECT TO DEVELOPMENT ERRORS

Technique	Input type checking	Encoding of input	Identification of input sources	Positive pattern matching
JDBC Checker [6,7]	Yes	No	No	No
SecuriFly [23]	No	Yes	Yes	No
Security Gateway[8]	Yes	Yes	No	Yes
SQLDOM [14]	Yes	Yes	N/A	No
WebSSARI [9]	Yes	Yes	Yes	Yes

Therefore, we examine each of the prevention techniques and classify them with respect to defensive coding practice that they enforce. Not surprisingly, we find that these techniques enforce many of these practices. Table 3 summarizes, for each technique, which of the defensive coding practices it enforces.

iii) Comparison of techniques based on deployment requirement Each prevention and detection technique evaluated based on the following criteria: (a) Does the technique require developers to modify their code base? (b) What is the degree of automation of prevention aspect of the technique? (c) What is the degree of automation of detection aspect of the technique? (d) What are additional factors needed to successfully use the technique? The results of this classification are summarized in Table 4. A table 4 determines the degree of automation of technique in the prevention or detection of attacks and also it could be cleared that which technique needs to modify the source code of application. Moreover additional elements that is required for each technique is illustrated.

TABLE 4: ANALYSIS OF PREVENTION AND DETECTION TECHNIQUES BASED ON ADDITIONAL ELEMENTS

S.No.	Techniques	Source Code Modification	Attack Prevention	Attack Detection	Additional Requirements
1	AMNESIA[12,13]	Not needed	Automatic	Automatic	N/A
2	Automated Approaches[33]	Not needed	Automatic	Automatic	N/A
3	CANDID[3]	Not needed	Automatic	Automatic	N/A
4	CSSE[31]	Not needed	Automatic	Automatic	Custom PHP interpreter
5	DIWeDa [30]	Not needed	N/A	Automatic	N/A
6	Detection based on removing SQL query attribute values [2]	Not needed	Automatic	Automatic	N/A
7	IDS[25]	Not needed	Report generate	Automatic	IDS system training set
8	JDBC Checker[6,7]	Not needed	Code modification suggested	Automatic	N/A
9	Positive Tainting[15]	Not needed	Automatic	Automatic	N/A

S.No.	Techniques	Source Code Modification	Attack Prevention	Attack Detection	Additional Requirements
10	SQLCheck [10]	Needed	Automatic	Partially automatic	Key Management
11	SQLGuard	Needed	Automatic	Partially automatic	N/A
12	SQLIPA[1]	Not needed	Partially automatic	Automatic	N/A
13	SAFELI [29]	Not needed	N/A	Partially automatic	N/A
14	SQLrand[11]	Needed	Automatic	Automatic	Developer training, Key Management, Proxy filter
15	SQL Prevent [22]	Not needed	Automatic	Automatic	N/A
16	SecuriFly[23]	Not needed	Automatic	Automatic	N/A
17	Security Gateway[8]	Not needed	Automatic	Detailed manual Specification	Proxy filter
18	SQLDOM[14]	Needed	Automatic	Automatic	Developer training
19	Swaddler [18]	Not needed	Automatic	Automatic	Training
20	Tautology Checker [19]	Not needed	Code modification suggested	Automatic	N/A
21	WAVES[5]	Not needed	Report generate	Automatic	N/A
22	WebSSARI[9]	Not needed	Partially automatic	Automatic	N/A

V.CONCLUSION

Most of the web applications uses intermediate layer to accept a request from the user and retrieve sensitive information from the database. Most of the time they use scripting language to build intermediate layer. To breach security of database hacker often uses SQL injection techniques. Generally attacker tries to confuse the intermediate layer technology by reshaping the SQL queries. Perhaps, attacker will change the activities of the programmer for their benefits.

In this paper, we assessed detecting and preventing SQL injection attacks among current SQL Injection detection and prevention techniques. To perform this evaluation, we first identified the various types of SQLIAs known to date. Then we investigated SQL injection detection and prevention techniques. After that we compared these techniques in terms of their ability to stop SQLIA. Regarding the results some current techniques ability should be improved for stopping SQLI attacks. We also studied the different mechanisms through which SQLIAs can be introduced into an application and identified which techniques were able to handle which mechanisms. We also found a general distinction between detection and prevention techniques. And we also suggested that prevention techniques try to incorporate defensive coding best practices into their attack prevention mechanisms. The SQL injection attack also stopped using approach such as Blacklist malicious hosts, Minimize admin-level access to a database**,** Normalize inputs, Model Based Hybrid Approach ,SVM(Support Vector Machine) ,ASCII Based String Matching, share intelligence on SQL injection attacks detection and prevention technique.

Future evaluation work direction on evaluating the technique precision, stability, flexibility and effectiveness in practice to show strength and weakness of the techniques.

REFERENCES

[1] S. Ali, SK. Shahzad and H. Javed, "SQLIPA: An Authentication Mechanism against SQL Injection", *European Journal of Scientific Research ISSN 1450-216X Vol.38 No.4 (2009)*, pp 604-611.

[2] Inyong Lee , Soonki Jeong Sangsoo Yeoc, Jongsub Moond, "A novel method for SQL injection attack detection based on removing SQL query attribute", *Journal Of mathematical and computer modeling, Elsevier* 2011.

[3] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks", *ACM Transaction on information System Security,* pp.1–39, 2010.

[4] Kemalis, K. and T. Tzouramanis, "SQL-IDS: A Specification-based Approach for SQL

injection Detection", *SAC'08. Fortaleza, Ceará, Brazil, ACM* , pp.2153 2158, 2008.

[5] Y. Huang, S. Huang, T. Lin, and C. Tsai, "Web Application Security Assessment by Fault Injection and Behavior Monitoring", in *Proceedings of the 11th International World Wide Web Conference (WWW 03)*, May 2003.

[6] C. Gould, Z. Su, and P. Devanbu. JDBC Checker, "A Static Analysis Tool for QL/JDBC Applications", in *Proceedings of the 26th International Conference on Software Engineering (ICSE04) Formal Demos*, pp 697–698, 2004.

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

[8] D. Scott and R. Sharp, "Abstracting Application-level Web Security", in *Proceedings of the 11*th *International Conference on the World Wide Web (WWW 2002)*, pages 396–407, 2002.

[9] 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 ", in *proceedings of the 12th International World Wide Web Conference (www 04),* May 2004.

[10] Z. Su and G. Wassermann "The essence of command injection attacks in web applications", *In ACM Symposium on Principles of Programming Languages (POPL'2006),* Jan.2006.

[11] S. W. Boyd and A. D. Keromytis,"SQLrand: Preventing SQL Injection Attacks", in *Proceedings of the 2nd Applied Cryptography and Network Security Conference*, pages 292–302, Jun. 2004.

[12] W. G. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks", in *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005),* Long Beach, CA, USA, Nov 2005.

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

[14] McClure, and I.H. Kruger, "SQL DOM: compile time checking of dynamic SQL statements," Software Engineering, ICSE *2005, Proceedings. 27th International Conference on*, pp. 88- 96, 15-21 May 2005.

[15] William G. Halfond, Alessandro Orso, "Using Positive Tainting and Syntax Aware Evaluation to Counter SQL Injection Attacks",

14th *ACM SIGSOFT international symposium on Foundations of software engineering*, ACM. pp: 175 – 185, 2006.

[16] Roichman, A., Gudes, E., "Fine-grained Access Control to Web Databases" *Proceedings of 12th SACMAT Symposium*, France (2007).

[17] Amirtahmasebi, K., Jalalinia, S.R., and Khadem, S., "A survey of SQL injection defense mechanisms" *International Conference for Internet Technology and Secured Transactions (ICITST 2009)*, 9-12 ,1-8,Nov. (2009).

[18] Macro Cova, Davide Balzarotti. *Swaddler:"* An Approach for the Anomaly-based Detection of State Violations in Web Applications", *Recent Advances in Intrusion Detection, Proceedings,* volume: 4637 Pages: 63-86 Published: 2007.

[19] G. Wassermann, Z. Su, "An analysis framework for security in web applications," *In: Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems, SAVCBS,* , pp. 70–78, 2004.

[20] Buehrer, G., Weide, B.W., and Sivilotti, P.A.G., " Using Parse Tree Validation to Prevent SQL Injection Attacks" *Proc. of 5th International Workshop on Software Engineering and Middleware*, Lisbon, Portugal (2005) 106–113.

[21] M. Howard and D. Leblanc, "Writing Secure Code", Microsoft Press, Redmond, Washington, *second edition*, 2003.

[22] P.Grazie., PhD, "SQL Prevent thesis", University of British Columbia (UBC) Vancouver, Canada,2008.

[23] M. Martin, B. Livshits, and M. S. Lam., " Finding Application Errors and Security Flaws Using PQL: A Program Query Language" *ACM SIGPLAN Notices*, Volume: 40, Issue: 10 Pages: 365-383, 2005.

[24] Y. Kosuga, K. Kernel, M. Hanaoka, M. Hishiyama, Y. Takahama, "Sania: Syntactic and Semantic Analysis for Automated Testing Against SQL Injection", *in: Proceedings of the Computer Security Applications Conference* 2007, 2007, pp. 107–117.

[25] F. Valeur, D. Mutz, and G. Vigna., " A Learning-Based Approach to the Detection of SQL Attacks" in *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Vienna, Austria, July 2005.

[26] 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).

[27] 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).

[28] Y. Shin, L. Williams and T. Xie, "SQLUnitGen: Test Case Generation for SQL Injection Detection," North Carolina StateUniv., Raleigh Technical report, NCSU CSC TR 2006-21, 2006.

[29] 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

[30] A. Roichman, E. Gudes, "DIWeDa - Detecting Intrusions in WebDatabases". In: Atluri, V. (ed.) DAS 2008. LNCS, vol. 5094, pp. 313–329. Springer, Heidelberg (2008).

[31] Tadeusz Pietraszek and Dhris Vanden Berghe., "Defending against Injection Attacks through Context-Sensitive String Evaluation", *Proceedings of Recent Advances in Intrusion Detection* (RAID2005).

.[32] R. Ezumalai, G. A. (2009). "Combinatorial Approach for Preventing SQL Injection Attacks," *S2009 IEEE International Advance Computing Conference (IACC 2009). Patiala, India*: pp.1212-1217

[33] Mei Junjin, "An Approach for SQL Injection Vulnerability Detection," *Proc. of the 6th Int. Conf. on Information Technology: New Generations*, Las Vegas, Nevada, pp. 1411-1414, Apr. 2009.

[34] Haixia, Y. and Zhihong, N., "A database security testing scheme of web application" *Proc. of 4th International Conference on Computer Science & Education 2009 (ICCSE '09)*, 25-28 July (2009), 953-955.

[35] Y. Huang, S. Huang, T. Lin, and C. Tsai, "A Testing Framework for Web Application Security Assessment," *Computer Networks*, volume: 48 Issue: 5, pages: 739-761, 2005

[36] Atefeh Tajpour, Suhaimi Ibrahim, Maslin Masrom, "SQL Injection Detection and Prevention Techniques" International Journal of Advancements in Computing Technology Volume 3, Number 7, August 201110.4156/ijact.vol3.issue7.11

[37] Diallo Abdoulaye Kindy and Al-Sakib Khan Pathan" "A Survey on SQL Injection: Vulnerabilities, Attacks, and Techniques" IEEE 15th International Symposium on Consumer Electronics, 2011

[38] Mohammed Firdos ,Alam Sheikh, Secure Query Processing By Blocking SQL Injection Attack (SQLIA), *International Journal of Research in management ISSN 2249-5908*, Issue1,Vol.3(November-2011).

[39] M.Kiani, A. Clark, and G.Mohay, "Evaluation of Anomaly Based Character Distribution Models in the Detection of SQL Injection Attacks," *The Third International*

Conference on Availability, Reliability, and Security, IEEE Computer Society,2008.

[40] Stephen Thomas, Laurie Williams, "Using Automated Fix Generation to Secure SQL Statements" , *Third International Workshop on Software Engineering For Secure Systems (SESS'07)*, pages9-9, May 2007.

[41] C Anley, Advanced SQL Injection in SQL Server Applications, "*White Paper Next Generation Security Software Ltd*"., 2002,

[42] Y. Shin, " Improving the identification of actual input manipulation vulnerabilities", *in: 14th ACM SIGSOFT Symposium on Foundations of Software Engineering ACM*, 2006.

[43] R. A. Baker, "Code Reviews Enhance Software Quality," In Proceedings of the 19th international conference on Software engineering (ICSE'97) pp. 570 - 571, Boston, MA, USA. 1997.

[44] "The Web Hacking Incident Database Semiannual (WHID) Report from July to December 2010", Trust wave Holdings, Inc, 2010.

[45]Paros.Parosproxy.org. http://www.parosproxy.org/.

[46] The Open Web Application Security Project,"OWASP TOP Project", https://www.owasp.org/SQL_Injection.

[47]The SQL injection, http://en.wikipedia.org/wiki/SQL_injection

[48]OWASP,http://www.owasp.org/index.php/Main_Page.

[49]WebSecurityDojo,http://www.mavensecurity.com/web_security_dojo

[50] Damn Vulnerable Web Application, http://www.dvwa.co.uk.

[51]Daffodil,http://crm.daffodilsw.com/article/call-centre-crm.html