International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 10, October 2013)

Extensive Review of SQLIA's Detection and Prevention Techniques

Monali R. Borade[1], Neeta A. Deshpande [2]

[1] Matoshri College of Enginering and Research Centre, Pune University, Nashik-422101 Maharashtra
[2] Assistant Professor, Matoshri College of Enginering and Research Centre, Pune University, Nashik-422101 Maharashtra

***Abstract*— Security of web applications is becoming one of the major concerns today. As per our survey 70% of web applications over the internet are vulnerable to SQL injection attacks (SQLIA's). SQL injection attacks pose serious security threat to these databases and web applications. Through SQLIA's attackers gain unrestricted access to the databases of applications and potentially sensitive information. Many methods to address this problem have been proposed in the literature, some having the scope for extension. Methods employ only a subset of the prevention and detection techniques. An extensive survey was done to review and uncover these issues. The paper strongly focuses on the review work of SQL injection attacks and their detection and prevention approaches known to date. This paper elaborates the survey done for 30 techniques and the attacks they can withstand. An in depth study of the techniques and their performance against SQLIA's is focused in the paper. Also for each strategy its strengths and weaknesses are addressed along with comparative analysis.**

***Keywords*— Detection, Modification, Prevention, SQL injection attacks, Strategies, Vulnerabilities, Web application security**.

I. INTRODUCTION

In recent years the development of internet had huge impact on human life, commerce and culture. As each coin have two sides: the World Wide Web also has both pros and cons. Many web applications like social networking sites, E-commerce or web portals, online shopping portal had became central point of everybody's life. These web applications plays vital role in maintaining security of data stored underlying databases. Unsecured web applications allow injection attacks to perform unwanted operations on backend databases and theft of data. So security of web applications has become a necessity. SQL injection is a major concern belongs to code injection problem categories as described in [3] [11]. In these attacks, data provided by the user is included in an SQL query in such a way that part of the user's input is treated as SQL code. By taking advantage of these vulnerabilities, an attacker can gain access to web applications and underlying databases by submitting SQLCommands.

The main reason of SQL injection vulnerabilities is insufficient validation of inputs. To address this developers have proposed a range of detection and prevention strategies [2] that provides defensive approaches such as encoding user inputs, performing input validations. A rigorous and systematic application of these approaches makes an effective solution for detecting and preventing SQLIA's. To address these issues this review paper presents a comprehensive survey of SQL injections attacks known to date. The paper focuses on characterization of attacks, illustrates their effects, and provides example of how that attack could be possible. These set of attacks then evaluated to compare strengths and weaknesses of the solutions. The result of comparison shows the effectiveness of solutions.

The remaining paper is organized as follows: section 2 provides background information on SQLIA's and related concepts. Section 3 presents example application containing vulnerability. Section 4 briefly focuses on different attack types. Section 5 focuses on current techniques for detection and prevention of SQLIA's. Section 6 shows summarized comparison of all techniques. Finally in section 7 this paper is concluded and there will be a discussion on future trends and directions.

II. BACKGROUND OF SQLIA'S

An SQL injection attack occurs when an attacker manipulates the intended effects of SQL query by inserting new SQL keywords or operators into the query. Attacker sends this modified query to a user input box in a web form of a web application to gain unauthorized access. This input is converted in an SQL query in such a way that it forms an SQL code [2] [3]. This is generalized definition of SQL injection. Interested readers can refer to [35] for more formal definition of SQLIA's. Rest of the part of this section describes two important characteristics of SQLIA's: Injection Mechanisms and Attack Intent.

A. SQL Injection Mechanisms

Malicious SQL Statements can be inserted into injection vulnerable application by different input mechanisms. This section focuses on most common injection mechanisms.

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 10, October 2013)

1) Injection through user inputs: In this type, attacker injects malicious SQL commands into user input query. A web application can read user inputs by many ways depending on the environment in which the application is developed and deployed. In most of the cases the user input comes from web form that is transmitted to the web application via HTTP GET or POST requests [14]. Using this type of injection, attacker can gain unauthorized access of web application and its underlying database.

2) Injection through server variables: Server variables are collection of variables containing HTTP , environmental variables, network headers etc. web application use these variables in variety of ways such as logging usage statistics and identifying browsing trends. If these variables are use to logged into database without sanitization, it could cause SQL injection vulnerability [30]. As attacker can forge the values in HTTP and network headers, they can expose these vulnerabilities by placing an SQLIA directly into the headers. When the query log to the server, the unsanitized variables get issued to the database and the attack in the forged header then takes place.

3) Injection through cookies: Cookies are stored on the client machines which are files containing state information gathered by web applications. These cookies can be used to restore the client's state information when client returns to the web application. As client has full control over the cookies, a malicious client can modify the contents of the cookies to built SQL Queries to submit the attack to the web application. [8]

4) Second Order Injection: In this type attacker sends malicious inputs to the system or database to directly perform SQLIA when the input is used at a later time. The objective of this type of attack significantly differs from regular SQLIA's (first –order injection attacks). Second – order injections are not intended to occur at the time of input reaches to application or database but the attacker relays on the knowledge of where the and when the input will be used and plans the attack so that it executes during the usage of application or database. To clarify we present a classic example of a second order injection attack (taken from [1]). In this example, a user registers on a website using a seeded username, such as "admin'—". The application will properly escape the single quotes from input before storing it in database, preventing its potentially malicious effects. At this point the attacker modifies his or her password, an operation typically involving 1) checking that the user knows the current password and 2) changing the password if the check is successful.

To perform this web application might form an SQL command as follows:

query String="UPDATE users SET password =' "+new Password+" 'WHERE username=' "+ username+" ' AND password' ' "+old Password+" ' "

newPassword and oldPassword are the new and old passwords given by user respectively, and username is the name of the user currently logged-in (i.e. ''admin'--''). Therefore the query string that is sent to the database is:

UPDATE users SET password =' newPwd 'WHERE username= 'admin'---'AND password='oldPwd '

(The query assumes that newPassword and oldPassword are "newPwd" and "oldPwd").

Because "---"is the SQL comment operator, everything after this is ignored by the database. Therefore, the result of this query is that the database changes the password of the administrator ("admin") to an attacker specified value. These types of injections are usually difficult to detect and prevent because the point of injection and point where the attack actually takes place are different.

B. Attack Intent

Attacks can also be characterized based on the goal or intention of the attacker [2]. Therefore each of the attack type described in section 4 has one of the following intention or goal.

1) Injectable parameters Identification: Here attacker finds parameters and user input fields that are vulnerable to SQLIA's and probe a web application accordingly.

*2) Database fingerprinting***:** The attacker discovers the type and version of database that a Web application is using. Databases respond differently to different queries and attacks, and this information can be used to "fingerprint" the database. Knowing the type and version of the database used by a Web application allows an attacker to craft database specific Attacks. [2].

*3) Extracting data***:** These types of attacks employ approaches that extract data values from the database. Depending on the type of the Web application, this information could be sensitive and highly desirable to the attacker. Attacks with this intent are the most common type of SQLIA.

4) Modification of Data: this type involves adding and modifying data in a database.

5) Performing denial of service: This involves shutdown of database of web application, and denying services to users. Locking and dropping database tables type of attack also comes under this category.

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 10, October 2013)

6) Evading detection: This type refers to certain attack approaches that are employed to avoid auditing and detection by system protection mechanisms [2].

7) Authentication Bypassing: This type of attack is intended to allow attacker to bypass database and web application authentication mechanisms. And attacker gains all the rights and access privileges to databases and applications.

8) Executing remote commands: These types of attacks aims to execute arbitrary commands on the database for ex. stored procedures or functions available to database users.

9) Performing Privilege Escalations: These types of attacks are intended to take advantages of errors in code and logical flaws to escalate privileges of attacker.

III. Example Application Containing Vulnerability

Before discussing the various attack types, we introduce an example application that contains SQL injection vulnerability [2]. We use this example in the next section to provide attack examples.

```
1. String login, password, pin, query
2. login = getParameter("login");
3. password = getParameter("pass");
3. pin = getParameter("pin");
4.Connection conn.createConnection("MyDataBase");
5. query = "SELECT accounts FROM users WHERE login='" +
6.          login + "' AND pass='" + password +
7.          "' AND pin=" + pin;
8. ResultSet result = conn.executeQuery(query);
9. if (result!=NULL)
10. displayAccounts(result);
11. else
12. displayAuthFailed();
```

Figure 1: Excerpt of Servlet implementation.

The example shows a simple vulnerability that could be prevented using a straightforward coding fix approach. We use this example simply for illustrative purposes because it is easy to understand and illustrate many different types of attacks. The code in Figure 1 implements the login functionality for a web application similar to the implementations of login functionality found in existing Web-based applications. The code in the example uses the input parameters login, pass, and pin to dynamically build an SQL query and submit it to a database. For example, if a user submits login, password, and pin as "monk," "secret," and "321," the application dynamically builds and submits the query:

SELECT accounts FROM users WHERE login='monk' AND pass='secret' AND pin=321

If the login, password, and pin match the corresponding entry in the database, monk's account information is returned and then displayed by function displayAccounts (). If there is no match in the database, function displayAuthFailed () displays an appropriate error message.

IV. Types Of SQLIA's

This section presents and discusses the different kinds of SQLIAs known to date. For each attack type, it provide a descriptive *name*, one or more *attack intents*, a *description* of the attack, an attack *example*, and a set of *references* to publications and Web sites that discuss the attack technique and its variations in greater detail.[2] [9].

The different types of attacks are generally not performed in isolation; many of them are used together or sequentially, depending on the specific aim of the attacker. Note also that there are countless variations of each attack type. For space reasons, we do not present all of the possible attack variations but instead present a single representative example [9] [10].

A. Tautologies

Attack Intent: Bypassing authentication, identifying injectable parameters, extracting data.

Description: Tautology-based attack is used to inject code in one or more conditional statements so that they always evaluate to true. This technique is most commonly used to bypass authentication pages and extract data. If the attack is successful, the code either displays all of the returned records or performs some action if at least one record is returned.

Example: In this example attack, an attacker submits " ' or 1=1 - -"

The Query for Login mode is:

SELECT * FROM user_info WHERE loginID=" or 1=1 -- AND pass1="

The code injected in the conditional (OR 1=1) transforms the whole WHERE clause into a tautology. The query evaluates to True for each row in the table and returns all of them. In above example, the returned set evaluates to a not null value, which causes the application to conclude that the user authentication was successful. Therefore, the application would invoke method user_main.aspx and to access the application.
References: [6, 7, 8, 13]

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 10, October 2013)

B. Illegal/Logically Incorrect Queries

Attack Intent: performing database finger-printing, extracting data, identifying injectable parameters

Description: The error message sent from databases on being sending wrong SQL query sometimes contain some useful debugging information. This could help attacker in finding parameters which are vulnerable in the web application and hence in the database of the application.

Example- The error message for sending a wrong password may be like:-

Select * from <tablename> where userId = <id> and password = <wrongPassword> or 1=1;

From this information the attacker is likely come to know the table name and name of the fields in the database which could be used further to prepare a more organized attack. For "Credit Cards." A similar strategy can be used to systematically extract the name and type of each column in the database. Using this information about the schema of the database, an attacker can create further attacks that target specific pieces of information.
References: [1, 22, 28]

C. Union Query

Attack Intent: Bypassing Authentication, extracting data.

Description: In union-query attacks, an attacker exploits a vulnerable parameter to change the data set returned for a given query. In union-query attacks, Attackers do this by injecting a statement of the form: UNION SELECT <rest of injected query> because the attackers completely control the second/injected query they can use that query to retrieve information from a specified table. The result of this attack is that the database returns a dataset that is the union of the results of the original first query and the results of the injected second query. [13, 10, 2]

Example: Referring to the running example, an attacker could inject the text "' UNION SELECT cardNo from CreditCards where acctNo=10032 - -" into the login field, which produces the following query:

SELECT accounts FROM users WHERE login='' UNION SELECT cardNo from CreditCards where acctNo=10032 -- AND pass='' AND pin=

Assuming that there is no login equal to "", the original first query returns the null set, whereas the second query returns data from the "CreditCards" table. In this case, the database would return column "cardNo" for account "10032." The database takes the results of these two queries, unions them, and returns the single query to the application.

In many applications, the effect of this operation is that the value for "cardNo" is displayed along with the account information.
References: [1, 28, 21]

D. Piggy-backed Queries

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

Description: In this type of attack where an attacker appends ";" and a query which can be executed on the database. It could be one of the very dangerous attacks on databases which could damage or may completely destroy a table. If this attack is successful then there could be huge loss of data. [10]

Example: If the attacker inputs "'; drop table users - -" into the *pass* field, the application generates the query:

SELECT accounts FROM users WHERE login='doe' AND pass=''; drop table users --' AND pin=123

After executing the first query, the database would recognize the query delimiter (";") and executes the injected second query. The result of execution of the second query would be to drop table users, which would likely destroy valuable information. Other types of queries could insert new users into the database or execute stored procedures. It should be noted that many databases do not need a special character to separate distinct queries, so simply scanning for a query Separator is not an effective way to prevent this type of attack.
References: [1, 28, 18]

E. Stored Procedure

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

Description: Stored procedure is routines stored in the database and run by the database engine. These procedures can be either user-defined procedures or procedures provided by the database by default. Depending on the type of stored procedure there are different ways to attack. The vulnerability here is same as in web applications. Moreover all the types of SQL injection applicable for a web application are also going to work at this level.

```
CREATE PROCEDURE DBO.isAuthenticated
@userName varchar2, @pass varchar2, @pin int
EXEC("SELECT accounts FROM users
WHERE login='" +@userName+ "' and pass='"
+@password+
"' and pin=" +@pin);
GO
```

Figure 2: Stored procedure for checking credentials.

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 10, October 2013)

Example: This example shows how a parameterized stored procedure can be exploited through an SQLIA. In the above example, we assume that the query string constructed at lines 5, 6 and 7 of our example has been replaced by a call to the stored procedure defined in Figure 2. In this example the stored procedure returns a true/false value indicating whether the user's credentials authenticated correctly. To launch an SQLIA, the attacker simply injects "'; SHUTDOWN; - -" into either the userName or password fields. This injection causes the stored procedure to generate the following query:

SELECT accounts FROM users WHERE login='doe' AND pass=' '; SHUTDOWN; -- AND pin=

At this point, this attack simply works like a piggy-back attack. The first query is executed normally, and after that the second malicious query is executed, which results in a database shut down. This example shows that stored procedures can be vulnerable to the same range of attacks as traditional application code.
References: [1, 4, 9, 10, 24, 28, 21, 18]

F. Blind Injection

Attack Intent: Data extraction, Data theft.

Description: It becomes difficult for an attacker to get information about a database when developers hide the error message coming from the database and send a user to a generic error displaying page [5]. It's the point when an attacker can send a set of true/false questions to steal/theft data.

Example- SELECT name FROM <tablename> WHERE id=<username> and 1 =0 -- AND pass = SELECT name FROM <tablename> WHERE id=<username> and 1 = 1 -- AND pass =

Both the queries after execution will return an error message. In case the web application is secure, but if the inputs are not validated in advanced then the chances of injection exist. If attacker receives an error after submitting the first query, he might not know that, was it because of input validation or error in query formation. After that on submission of the second query which is always true if there is no error message then it clearly states that id field is vulnerable.
References: [10, 28, 18]

G. Timing Attacks

Attack Intent: Server shutting down.

Description: In this type of attack timing delays are observed in response from a database which helps to gather information from a database.

Here SQL engine is forced to execute a long running query or a time delay statement with the help of if-then statement that depends on the logic that has been injected. It is possible to determine whether injected statement was true or false depending on how much time page takes to load. The keyword WAITFOR combined with the branches can cause response delay for a given time in a database.

Example- Declare @s varchar(500) select @s = db_nameO if (ascii(substring(@s, I, I)) & (power(3, 0))) > O waitfor delay '0:0:20'

In this example database gets paused for twenty seconds if in the database used, the first bit of the first byte of the name is 1.So, when condition will be true this code is injected to produce response delay in time.
References: [10, 5, 12, 18]

H. Inference

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

Description: In this type of attack, the query is modified to recast it in the form of an action that is executed based on the answer to a true/false question about data values in the database [10]. In this type of injection, attacker generally tries to attack a site that has been secured enough so that, when an injection has succeeded, there is no usable feedback via database error messages. As database error messages are unavailable to provide the attacker with feedback, Attackers must use a different method of obtaining a response from the database. In this situation, the attacker injects commands into the site and then observes how the function/response of the Website changes. By carefully observing, when the site changes its behavior, the attacker can deduce not only whether certain parameters are vulnerable, but also additional information about the values in the database. There are two well-known attacks that are based on inference. They allow an attacker to extract data from a database and detect vulnerable parameters.
References: [10, 2]

I. Alternate Encodings

Attack Intent: Evading detection of vulnerabilities.

Description: This technique is used to modify injection query by using alternate encodings, means replacing characters in query by some other characters or symbols like – Unicode, ASCII, hexadecimal. In this way attacker can escape the filter for "wrong characters". It could be extremely harmful for web application if used in combination with other techniques as it can target different layers of a web application.

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 10, October 2013)

All different kinds of SQL injection attack can be hidden using this method.

Example- ***SELECT name FROM <tablename> WHERE id='' and password=O; exec (char (O x73687574646j776e))***

The actual character is returned by the char function used here that takes hexadecimal encoded characters as an input. During execution this encoded string gets converted into shutdown command for database.
References: [10, 2, 13]

J. Deny Database service

Attack Intent: Denying Database services, shutdown of server, DDOS.

Description: This type of attack is used in the websites to issue a denial of service by shutting down the SQL Server. A powerful command recognized by SQL Server is SHUTDOWN WITH NOWAIT [19]. This causes the server to shutdown, immediately stopping the Windows services. After this command has been issued, the administrator must restart the service be manually

Example 1: **Select password from user_info where LoginId=';shutdown with nowait; --' and Password='0'**

The '--' character sequence denotes the 'single line comment' sequence in Transact - SQL, and the ';' character denotes the end of one query and the beginning of another. If the attacker has used the default SA account, or has acquired the required privileges, SQL server will shut down, and will need a restart in order to function again. This attack is used to stop the database service of a particular web application.

Example 2:
Select * from user_info where LoginId='1; xp_cmdshell 'format c:/q /yes '; drop database mydb; --AND pass1= 0

This command is used to format the C:\ drive used by the Attacker.

V. Existing Techniques For Detection And Prevention Of SQLIA's

Researchers have proposed a variety of techniques to address the problem of SQL injection. These techniques range from development best coding practices to fully automated frameworks for detecting and preventing SQLIAs. This section reviews these proposed Techniques and summarizes the advantages and disadvantages associated with each technique.

A. Defensive Coding Practices

The main reason of SQL injection vulnerabilities is improper input validation. Therefore, the best solution for eliminating these vulnerabilities is to apply suitable defensive coding Practices. Hence this section summarizes some of the best practices proposed in the literature for preventing SQL injection vulnerabilities.

1) Input type checking: SQLIAs can be performed by just injecting commands into a string or numeric parameter. A simple check of such inputs can prevent many attacks.

2) Encoding of inputs: In this type of practice, *Injection* into a string parameter is often accomplished through the use of meta-characters that trick the SQL parser into interpreting user input as SQL tokens. An effective solution is to use functions to encode a string in such a way that all meta-characters are specially encoded and interpreted by the database as normal characters.

3) Positive pattern matching: developers should use some pattern matching algorithms or input validation routines to differentiate bad inputs and good inputs. This approach is generally called *positive validation.* In this developer will *be* able to specify all the forms of legal input, positive validation is a safer way to check inputs.

4) Identification of all input sources: Developers must check all input to their application. As outlined in Section 2.1, there are many possible sources of input to an application. If these inputs are used to construct a query, this can lead to a way for an attacker to introduce an SQLIA. Simply put, all input sources must be checked.

Although defensive coding practices are the best possible way to prevent SQL injection vulnerabilities, but their application is problematic in practice. Defensive coding is prone to human errors and is not as completely and rigorously applied as automated techniques [20, 23, and 33].

B. Detection and Prevention Techniques

1) *Black Box Testing :* Huang and colleagues [19] proposed **WAVES** in year 2003, 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.

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 10, October 2013)

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.

2) *Static Code Checkers:* **JDBC-Checker** is a technique proposed by C. Gould, Z. Su, and P. Devanbu. In year 2004 for static checking of the type correctness of dynamically-generated SQL queries [12, 13]. This technique was developed to 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 due to 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.

3) Tautology checker: a technique proposed by Wassermann and Su in year 2004, gives an analysis framework for security in web application. It uses static analysis combined with automated reasoning to verify that the SQL queries generated in the application layer cannot contain a tautology [37]. The primary drawback of this technique is that its scope is limited to detecting and preventing tautologies and cannot detect other type of vulnerabilities.

4) Instruction Set Randomization : **SQLRand is** a technique proposed by S. W. Boyd and A. D. Keromytis in year 2004 for preventing SQL injection attacks by instruction set randomization. In this technique the SQL keywords are attached with the key generated by the randomization Algorithm [5]. When an attacker, who has no knowledge of the key, attacks the application, the attempt fails because the query constructed by the attack will not match with the query that contains the randomly generated key. The keywords in both the queries will differ, and prevents SQL injection attack. Since it's a static analysis technique, the security of server's web databases is not compromised in case of an attack on the proposed method. However, implementation of a proxy server for randomization and de-randomization adds to the performance overhead.

5) Static and Dynamic Analysis Techniques: Two recent related approaches, **SQLGuard** [6] proposed by G.T. Baehre, B. W. weide and P.A.G. Sivilotti in year 2005 and **SQLCheck** [35] proposed by Z. Su & G. Wassermann in year 2006 also check queries at runtime to see if they conform to a model of expected queries. In these approaches, the model is expressed as a grammar that only accepts legal queries.

In SQLGuard, the model is deduced at runtime by examining the structure of the query before and after the addition of user-input. In SQLCheck, the model is specified independently by the developer. Both approaches use a secret key to delimit user input during parsing by the runtime checker, so security of the approach is dependent on attackers not being able to discover the key.

6) Amnesia: **Amnesia [**15] [16] [17] is technique proposed by W. G. Halfond and A. Orso, in year 2005. In this approach, a combination of static analysis and run-time monitoring is used for prevention of SQL injection. In static phase The AMNESIA tool builds a model of all the queries that are generated by the application. For this purpose, the tools access the entire source code. In the dynamic phase, the query built during run-time is validated against the model built during the Static phase.

7) CANDID: **CANDID** [41] is a technique proposed by Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishna for Preventing SQL Injection Attacks using Dynamic Candidate Evaluations. This approach dynamically mines the programmer-intended query structure and compares this structure with the actual query. It is used to run the application on candidate inputs that are benign. However, it's not a practical approach because the problem of finding such inputs is undesirable.

8) Taint Based Approaches

a. WebSSARI: **WebSSARI** detects input-validation related errors using information flow analysis [20] proposed by Y. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo., In year 2004. In this approach, static analysis is used to check taint flows against preconditions for sensitive functions. It detects the points in which preconditions have not been met and can suggest filters and sanitization functions that can be automatically added to the application. To satisfy these preconditions. The WebSSARI system works by considering sanitized input that has passed through a predefined set of filters.

b. JAVA static Tainting: **Java static Tainting** Proposed by V. B. Livshits and M. S. Lam [23] in year 2005 for Finding Security Errors in Java Programs with Static Analysis. The basic approach is to use information Flow techniques to detect when tainted input has been used to construct an SQL query. These queries are then flagged as SQLIA vulnerabilities. As it uses a conservative analysis and has limited support for untainting operations, hence they can generate a relatively high amount of false positives. Varieties of dynamic taint analysis approaches have been proposed.

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 10, October 2013)

c. Web app. Hardening: **Web App. Hardening** [31] proposed by Nguyen-Tuong and colleagues and **CSSE** [32] proposed by Pietraszek and Berghe modify a PHP interpreter to track precise per-character taint information. The techniques use a context sensitive analysis to detect and reject queries by checking whether untrusted input has been used to create certain types of SQL tokens. A common drawback of these two approaches is that they require modifications to the runtime environment, which affects portability.

d. JAVA DYNAMIC TAINTING: **JAVA Dynamic Tainting** [15] is a technique proposed by V. Haldar, D. Chandra, and M. Franz in year 2005 and **SecuriFly** [26] proposed by M. Martin, B. Livshits, and M. S. Lam in year 2005 implements a similar approach for Java. However, these techniques do not use the context sensitive analysis employed by the other two approaches and track taint information on a per-string basis (as opposed to per character).

9) New Query Building Approaches: **SQL DOM** [27] is a technique proposed by R. McClure and I. Kruger in year 2005 and **Safe Query Objects** [7] proposed by W. R. Cook and S. Rai, used encapsulation of database queries to provide a safe and reliable way to access databases. These techniques provides an effective solution to avoid the SQLIA problem by changing the query-building process from an unregulated to the approach using string concatenation to a systematic one that uses a type-checked API

10) Intrusion Detection Systems: It is a technique proposed by Valeur and colleagues [36] in year 2006 to detect SQLIAs. Their IDS system is based on a machine learning technique that is trained using a set of typical application queries. This technique builds models of the typical queries and then monitors the application at runtime to identify queries that do not match the model.

11) Proxy Filters

a. Security Gateway [33] proposed by D. Scott and R. Sharp is a proxy filtering system that forces the input validation rules on the data flowing to a Web application. Using their Security Policy Descriptor Language (SPDL), developers provided the constraints and specify transformations to be applied to application parameters as they flow from the Web page to the application server.

b. Automated fix generation to secure SQL statements [38] proposed by Stephan Thomas and Laurie Williams in year 2007 is a technique that gathers information of known vulnerable SQL statements, generates a fix (alternative code) and then replaces this vulnerability with the generated code. This method, however, is based on an assumption that the language of development, database connector and database system support prepared statements [38]. It also assumes that the vulnerable code has equivalent data types as the corresponding field in the database. In case of mismatching Data types, it assumes that the compiler will handle run-time type conversions.

12) Pattern Matching Algorithm [40] : proposed by Prabakar ,M.A Karthikeyan, M. Marimuthu, K. in year 2013 is a technique that can be used to identify or detect any anomaly packet from a sequential action. Injection attack is a method that can inject any kind of malicious string or anomaly string on the original string. Most of the pattern based techniques are used static analysis and patterns are generated from the attacked statements. This technique proposed a detection and prevention method for preventing SQL Injection Attack (SQLIA) using Aho-Corasick pattern matching algorithm.

*13) Frameworks for SQL Retrieval on Web Application Security [39]***:** A technique proposed by Haeng Kon Kim in year 2010 presents a framework for SQL retrieval on web application security. This technique is divided into two modules - Pattern Creation Module (PCM) and Attack Detection Module (ADM). PCM creates a model of attacks based on the patterns observed from previous attacks, while ADM checks if the query fired by the application matches an existing pattern.

VI. TECHNIQUES EVALUATIONS

A. Comparative Analysis for SQL Injection Attacks and Solution Techniques

Every approach has some advantages depending on the settings of the system configured, so it would not be easy to get an idea about which solution or technique is the best. Table 4 shows a chart of different approaches against different kinds of SQL injection attacks. It also shows a comparative analysis of SQL injection detection and prevention techniques with attack types. Table 1 shows objective of various solution approaches, table 2 shows comparison of various SQLIA's detection focused approaches with respect to attack types, while table 3 shows comparison of various SQLIA's prevention focused approaches with respect to attack types.

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 10, October 2013)

Tables 2 and 3 summarize the results of evaluation of all the techniques. We use four different types of markings are used to indicate how a technique performed with respect to a given attack type. The symbol "tick" denotes that a technique can successfully stop all attacks of that type. Conversely, the symbol "cross" denotes that a technique is not able to stop attacks of that type. Two different symbols to classify techniques that are only partially effective. The symbol "◦" denotes a technique that can address the attack type considered, but cannot provide any guarantees of completeness. An example of one such technique would be a black-box testing technique such as WAVES [19] or the IDS based approach from Valeur and colleagues [36]. The symbol "−," denotes techniques that address the attack type considered only partially because of intrinsic limitations of the underlying approach. For example, JDBCChecker [12, 13] detects type-related errors that enable SQL injection vulnerabilities. However, because type-related errors are only one of the many possible causes of SQL injection vulnerabilities, this approach is classified as only partially handling each attack type.

Although there exist variety approaches to identify and prevent these attacks [2], only a few of them have been implemented practically. Hence, this comparison is based on analytical evaluation rather than empirical experience.

B. Evaluation with Respect to Injection Mechanisms

Each of the techniques is addressed with respect to their handling of various injection mechanisms that are defined in Section 2.1. Although most of the techniques do not specifically address all of those injection mechanisms, all but two of them could be easily extended to handle all such mechanisms. The two exceptions are Security Gateway and WAVES. Security Gateway can examine only URL parameters and cookie fields. Because it resides on the network between the application and the attacker, it cannot examine server variables and second-order injection sources, which do not pass through the gateway. WAVES can only address injection through user input because it only generates attacks that can be submitted to the application through the Web page forms.

C. Evaluation with Respect to Deployment Requirements

As each of the techniques have different deployment requirements. To determine the effort and infrastructure required to use the technique, the author's description of the technique and its current implementation is examined and each technique evaluated with respect to the following criteria:

(1) Does the technique require developers to modify their code base? (2) What is the degree of automation of the detection aspect of the approach? (3) What is the degree of automation of the prevention aspect of the approach? (4) What infrastructure (not including the tool itself) is needed to successfully use the technique? The results of this classification are summarized in Table 4.

D. Evaluation of Prevention Focused Techniques with Respect to Defensive Coding Practices

Initial evaluation of the techniques against the various attacks types shows that the prevention-focused techniques perform very well against most of these attacks. The hypothesis shows 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. Therefore, each of the prevention-focused techniques is examined and classified with respect to the defensive coding practice that they enforce. Not surprisingly, it is observed that these techniques enforce many of these practices. Table 5 summarizes, for each technique, which of the defensive coding practices it enforces.

Table 1
Objective Of various approaches

Approaches	Goals	
	Detection	Prevention
AMNESIA	✓	✓
CANDID	✓	X
SQLrand	✓	X
SQLCHECK	✓	✓
SQL DOM	✓	X
SQLGuard	✓	✓
SQL-IDS	✓	X
SQLIPA	✓	✓
WebSSARI	✓	X

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 10, October 2013)

Table 2
Comparisons Of Various SQLIA's Detection Focused Approaches With Respect To Attck Types

Attacks/ Approaches	Tautology	Logically Incorrect Queries	Union Query	Piggy-Backed Queries	Stored Procedure	Blind Injection	Timing Attacks	Alternate Encoding
AMNESIA[16]	✓	✓	✓	✓	X	✓	✓	✓
CANDID	✓	X	X	X	X	X	X	X
DIWed	X	X	X	X	X	✓	✓	X
SQLRand[5]	✓	X	✓	✓	X	✓	✓	X
SQLCHECK [35]	✓	✓	✓	✓	X	✓	✓	✓
SQLDOM	✓	✓	✓	✓	X	✓	✓	✓
SQLGuard[6]	✓	✓	✓	✓	X	✓	✓	✓
SQLIPA[]	✓	X	X	X	X	X	X	X
CSSE[32]	✓	✓	✓	✓	X	✓	X	✓
IDS[36]	○	○	○	○	○	○	○	○
Java Dynamic Tainting [15]	----	----	----	----	----	----	----	----
Tautology-checker [37]	✓	X	X	X	X	X	X	X
Web app. Hardening [31]	✓	✓	✓	✓	✓	X	✓	X

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 10, October 2013)

Table 3
Comparison of prevention-focused techniques with respect to attack types.

Attacks/ Approaches	Tautology	Logically Incorrect Queries	Union Query	Piggy-Backed Queries	Stored Procedure	Blind Injection	Timing Attacks	Alternate Encoding
WEBASSARI [20]	✓	✓	✓	✓	✓	✓	✓	✓
JDBC Checker[12]	-----	-----	----	----	----	-----	----	----
Java Static Tainting[23]	✓	✓	✓	✓	✓	✓	✓	✓
Safe query objects[7]	✓	✓	✓	✓	X	✓	✓	✓
Security Gateway[33]	-----	-----	----	----	-----	-----	----	----
SecuriFly[26]	-----	-----	----	----	-----	-----	----	----
SQLDOM[27]	✓	✓	✓	✓	X	✓	✓	✓
WAVES[19]	o	o	o	o	o	o	o	o

Table 4
Comparison of techniques with respect to deployment requirements

Technique	Need to Modify code base	Detection	Prevention	Additional Infrastructures
AMNESIA [16]	X	FULLY AUTOMATED	FULLY AUTOMATED	NONE
CSSE [32]	X	FULLY AUTOMATED	FULLY AUTOMATED	CUSTOM PHP INTERPRETER
IDS [36]	X	FULLY AUTOMATED	GENERATE REPORTS	IDS SYSTEM TRAINING SET
JDBC-Checker [12]	X	FULLY AUTOMATED	CODE SUGGESTIONS	NONE
Java Dynamic Tainting [15]	X	FULLY AUTOMATED	FULLY AUTOMATED	NONE
Java Static Tainting [23]	X	FULLY AUTOMATED	FULLY AUTOMATED	NONE
Safe Query Objects [7]	✓	N/A	CODE SUGGESSIONS	DEVELOPER TRAINING
SecuriFly [26]	X	FULLY AUTOMATED	FULLY AUTOMATED	NONE
Security Gateway [33]	X	MAUAL SPECIFICATION	FULLY AUTOMATED	PROXY FILTER
SQLCheck [35]	✓	SEMI-AUTOMATED	FULLY AUTOMATED	KEY MANAGEMENT
SQLGuard [6]	✓	SEMI-AUTOMATED	FULLY AUTOMATED	NONE
SQL DOM [27]	✓	N/A	FULLY AUTOMATED	DEVELOPER TRAINING
SQLrand [5]	✓	FULLY AUTOMATED	FULLY AUTOMATED	PROXY, KEY MANAGEMENT, DEVELOPER TRAINING
Tautology-checker [37]	X	FULLY AUTOMATED	GENERATE REPORTS	NONE
WAVES [19]	X	FULLY AUTOMATED	CODE-SUGGESSIONS	NONE
Web App. Hardening [31]	X	FULLY AUTOMATED	FULLY AUTOMATED	CUSTOM PHP INTERPRETER
WebSSARI [20]	X	FULLY AUTOMATED	SEMI-AUTOMATED	NONE

International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 10, October 2013)

Table 5
Evaluation of Code Improvement Techniques with Respect to Common Development Errors

Technique	Input type checking	Encoding of input	Identificatication of all input sources	Positive pattern matching
JDBC-Checker [12]	✓	X	X	X
Java Static Tainting [23]	X	X	✓	X
Safe Query Objects [7]	✓	✓	N/A	X
SecuriFly [26]		✓	✓	X
Security Gateway [33]	✓	✓	X	✓
SQL DOM [27]	✓	✓	N/A	X
WebSSARI [20]	✓	✓	✓	✓

VII. Conclusion

This paper has presented a survey and comparison of current techniques for detecting and preventing SQLIAs. To perform this evaluation, first various types of SQLIAs known to date are identified and then the considered techniques were evaluated in terms of their ability to detect and/or prevent such attacks. Also different mechanisms through which SQLIAs can be introduced into an application and which techniques were Able to handle which mechanisms were identified. Finally, the deployment requirements of each technique were summarized and evaluated to what extent its detection and prevention mechanisms could be fully automated.

This evaluation found several trends in the results. Many of the techniques have problems while handling attacks that take advantages of poorly-coded stored procedures and are not able to handle attacks that disguise themselves using alternate encodings. Section 6.4 suggests that the difference between general detection prevention techniques and prevention focused techniques could be explained by the fact that prevention-focused techniques try to incorporate defensive coding best practices into their attack prevention mechanisms.

VIII. Future Scope

Future evaluation work should focus on evaluating the techniques precision and effectiveness in practical implementation. Empirical evaluations can be carried out such as those presented in related work (e.g., [17, 36]) would allow the comparison of the performance of the different detection and prevention techniques when they are applied against real-world attacks and legitimate inputs.

Acknowledgment

The authors wish to thank university Of Pune. This work is supported in part by a grant from university of Pune under grant no 13ENG001169.

REFERENCES

[1] C. Anley. Advanced SQL Injection In SQL Server Applications. White paper, Next Generation Security Software Ltd., 2002.

[2] William G.J. Hal fond and Alessandro Orso, "A Classification of SQL injection attacks and Countermeasures", proc IEEE int'l Symp. Secure Software Engg., Mar. 2006.

[3] D. Aucsmith. Creating and Maintaining Software that Resists Malicious Attack. http://www.gtisc.gatech.edu/bio aucsmith.html, September 2004. Distinguished Lecture Series.

[4] F. Bouma. Stored Procedures are Bad, O'kay? Technical report,Asp.NetWeblogs,November2003.http://weblogs.asp.net/fbouma/archive/2003/11/18/38178.aspx.

[5] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference, pages 292–302, June 2004.

[6] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. In International Workshop on Software Engineering and Middleware (SEM), 2005.

[7] W. R. Cook and S. Rai. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. In Proceedings of the 27th International Conference on Software Engineering (ICSE 2005).

[8] M. Dornseif. Common Failures in Internet Applications May 2005. http://md.hudora.de/presentations/

[9] Subodh Raikar. SQL Injection Prevention using Runtime Query Modeling and Keyword Randomization.

[10] Abhishek Kumar Baranwal. Approaches to detect SQL injection and XSS in web applications. EECE 571B, TERM SURVEY PAPER, APRIL 2012

[11] T. O. Foundation. Top Ten Most Critical Web Application Vulnerabilities,2005. http://www.owasp.org/documentation/topten.html.

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

[13] Indrani Balasundaram. , Dr. E. Ramara. An Approach to Detect and Prevent SQL Injection Attacks in Database Using Web Service. IJCSNS International Journal of Computer Science and Network Security, VOL.11 No.1, YEAR 2012

[14] N. W. Group. RFC 2616 – Hypertext Transfer Protocol – HTTP/1.1. Request for comments, The Internet Society, 1999.

[15] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In Proceedings 21st Annual Computer Security Applications Conference, Dec. 2005.

[16] 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.

[17] 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.

[18] M. Howard and D. LeBlanc. Writing Secure Code. Microsoft Press, Redmond, Washington, second edition, 2003.

[19] 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.

[20] 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.

[21] S. Labs. SQL Injection. White paper, SPI Dynamics, Inc., 2002. http://www.spidynamics.com/assets/documents/ WhitepaperSQLInjection.pdf.

[22] D. Litchfield. Web Application Disassembly with ODBC Error Messages. Technical document, @Stake, Inc., 2002. http://www.nextgenss.com/papers/webappdis.doc.

[23] V. B. Livshits and M. S. Lam. Finding Security Errors in Java Programs with Static Analysis. In Proceedings of the 14th Usenix Security Symposium, pages 271–286, Aug. 2005.

[24] C. A. Mackay. SQL Injection Attacks and Some Tips on How to Prevent Them. Technical report, The Code Project, January2005.

[25] O. Maor and A. Shulman. SQL Injection Signatures Evasion. White paper, Imperva, April 2004. http://www.imperva.com/application defense center/white papers/sql injection signatures evasion.html.

[26] M. Martin, B. Livshits, and M. S. Lam. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA 2005), pages 365–383, 2005.

[27] R. McClure and I. Kr¨uger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In Proceedings of the 27th International Conference on Software Engineering (ICSE 05), pages 88–96, 2005.

[28] S. McDonald. SQL Injection: Modes of attack, defense, and why it matters. White paper, GovernmentSecurity.org, April 2002.

[29] S. McDonald. SQL Injection Walkthrough. White paper, SecuriTeam,May2002.http://www.securiteam.com/securityreviews/5 DP0N1P76E.html.

[30] T. M. D. Network. Request.servervariables collection. Technical report, Microsoft Corporation, 2005. http://msdn.microsoft.com/library/default.Asp?url=/library/en-us/iissdk/html/9768ecfe-8280-4407-b9c0-844f75508752.asp.

[31] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting Information. In Twentieth IFIP International Information Security Conference (SEC 2005), May 2005.

[32] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks through Context-Sensitive String Evaluation. In Proceedings of Recent Advances in Intrusion Detection (RAID2005), 2005.

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

[34] K. Spett. Blind sql injection. White paper, SPI Dynamics, Inc., 2003. http://www.spidynamics.com/whitepapers/ Blind SQLInjection.pdf.

[35] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In The 33rd Annual Symposium on Principles of Programming Languages (POPL 2006), Jan. 2006.

[36] 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.

[37] G. Wassermann and 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 2004), pages 70–78, 2004.

[38] Stephen Thomas and Laurie Williams. Using Automated Fix Generation to Secure SQL Statements. In Proceedings of the Third International Workshop on Software Engineering for Secure Systems, SESS '07, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society.

[39] Haeng Kon Kim. Frameworks for SQL retrieval on Web Application Security. In Proceedings of the International Multiconference of Engineers and Computer Scientists, volume 1, page 5, Hong Kong, 2010. IMECS, International Association of Engineers.

[40] Prabakar, M.A. ; Karthikeyan, M. ; Marimuthu, K. An efficient technique for preventing SQL injection attack using pattern matching algorithm IEEE International Conference on Emerging Trends in Computing, Communication and Nanotechnology (ICE-CCN), 2013 Digital Object Identifier: 10.1109/ICE-CCN.2013.6528551 Publication Year: 2013, Page(s): 503 – 506

[41] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID : Preventing SQL Injection Attacks using Dynamic Candidate Evaluations. In Proceedings of The 14th ACM conference on Computer and communications security, CCS '07, pages 12–24, New York, NY, USA, 2007. ACM.