Protection of E-Commerce Website from SQL Injection: A Review

Meena
meenamaannehra@gmail.com
Department of Computer Science & Engineering
Dronacharya College of Engineering
Farukhnagar, Gurgaon, Haryana, India.

Abstract: - SQL injection Attack (SQLIA) is vulnerability in database based web application, of which the attackers take benefit to insert and execute malicious code to get the database information. This technique gives unauthorized access to database by giving input which consists of malicious code included into the query. Malicious query is treated like valid query by the database and executed. Attacker may have different intensions for attacks. Attacker may want to identify inject able and weak parameter to attack. By the attacks he/she can modify data, change data and extract data. He/she can get the confidential and sensitive data from the storage of database. Attacker may also want to know about the database schema which consists of the number of rows and columns, name of table, columns data types, column name from database to make use of all to inject/get information into/from the database system. In this paper we present detailed overview of what is SQL injection, how it works, intent of attack, types of attack, various techniques and tools to detect and prevent SQL Injection, comparison of these tools based on attack types and deployment requirements.

Keywords: - E-Commerce, SQL, SQLIA, Injection, Attack Types, Database, Vulnerable, SQL Parser, SQL Check Attacker.

I INTRODUCTION

SQL injection attack is major issue and very serious so the anticipation of SQL injection attack is major challenge in day today life. This vulnerability exists when web applications do not have proper input validation and use not parameterized stored procedures. Poorly designed web applications are vulnerable to injection of malicious code to get database access by attacker. Proper input validation and use of parameterized procedures can be used to prevent SQL injection. SQL injection attacks allow attackers to spoof identity, tamper with existing data, cause repudiation issues such as voiding transactions or changing balances, allow the complete disclosure of all data on the system, destroy the data or make it otherwise unavailable, and become administrators of the database server. So detection and prevention of SQL injection attacks is very important to stop SQL injection attack in websites. To achieve this objective, automatic tools have been implemented by different authors, which will be discussing in related work. The purpose of this paper is to review the various SQL injection detection and prevention tool. The structure of this paper is as follows:-
Chapter I describes definition and brief introduction of SQL Injection attack. In Chapter II related work of SQL Injection detection and prevention techniques and tools is given. The comparative analysis of SQL Injection detection and prevention tools is also given in chapter II.
Chapter III finally summarizes a conclusion and future scope of this survey.

1. What is SQL Injection?

SQL Injection is a technique in which attacker injects an input query in order to change the query and illegally gain the access of the database. SQL Injection allows attackers to create, read, update, alter, or delete and modify query in the back-end database and it also allow attackers to access sensitive information such as social security numbers, credit card number and other financial data. When any vulnerability present in web applications then the error is generated. Attacker takes an advantage of this error message as it displayed by the web server depicts the type of database structure that has been used [13].

2. How SQL Injection Works?

SQLIA is a hacking technique in which the attacker adds SQL statements through a web application's input fields or Hidden parameters to access to database system. Lack of input validation in web applications causes hacker to be Successful. For the following examples we will assume that a web application receives an http request from a client as input and generates a SQL statement as output for the back end database server. For example an administrator will be authenticated after Typing: username=superadmin and password=admin@1234. Figure 1 describes a login by a malicious user exploiting SQL Injection vulnerability. Basically it is structured in three phases [19]:

1. An attacker sends the malicious http request to the Web application
2. Creates the SQL statement
3. Submits the SQL statement to the back end database

As shown in figure 1 website needs username and password to login into admin panel. The SQL Query for authorized the admin will be like as given below:-
Select * from Login where username= 'superadmin' and password='admin@1234'
The malicious user can bypass this authorization by using SQL Injection by injecting SQL codes. Suppose malicious user enters " ' OR 1=1 -- "in user name field and "admin" in password field then the admin authorization query will be as:-
Select * from Login where username='' OR 1=1 -- and password='admin'

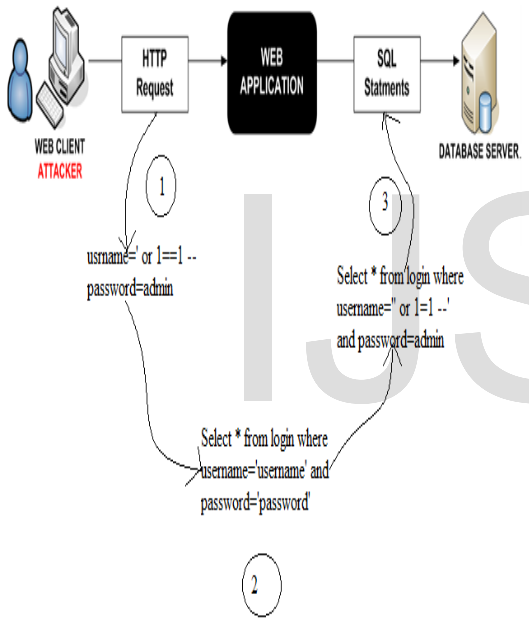

Figure 1 Example of a SQL injection attack

Now although the malicious user does not know both username and password but he can login into admin panel because "Or 1=1" is always true and password is bypassed by --, because – works as comment in SQL and the part "and password='admin' "gets commented and as per query structure the user is authorized and able to login.
The above SQL statement is always true because of the Boolean tautology we appended (or 1=1) so, we will access to the web application as an administrator without knowing the right password.

3. Attack Intent [8]

Attacks can also be characterized based on the goal, or intent, of the attacker. The attacker may want to probe a Web application to discover which parameters and user-input fields are vulnerable to SQLIA. Performing database finger-printing: The attacker wants to discover the type and version of database that a Web application is using. Certain types of 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.

Determining database schema

To correctly extract data from a database, the attacker often needs to know database schema information, such as table names, column names, and column data types. Attacks with this intent are created to collect or infer this kind of information. Extracting data: These types of attacks employ techniques that will 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. Adding or modifying data: The goal of these attacks is to add or change information in a database.

Performing denial of service

These attacks are performed to shut down the database of a Web application, thus denying service to other users. Attacks involving locking or dropping database tables also fall under this category. Evading detection: This category refers to certain attack techniques that are employed to avoid auditing and detection by system protection mechanisms.

Bypassing authentication

The goal of these types of attacks is to allow the attacker to bypass database and application authentication mechanisms. Bypassing such mechanisms could allow the attacker to assume the rights and privileges associated with another application user. Executing remote commands: These types of attacks attempt to execute arbitrary commands on the database. These commands can be stored procedures or functions available to database users.

Performing privilege escalation

These attacks take advantage of implementation errors or logical flaws in the database in order to escalate the privileges of the attacker. As opposed to bypassing authentication attacks, these attacks focus on exploiting the database user privileges.

4. Main Cause of SQL injection [19]

Web application vulnerabilities are the main causes of any kind of attack.

Invalidated input: this is almost the most common Vulnerability on performing a SQLIA. There are some parameters in web application, are used in SQL queries. If there is no any checking for them so can be abused in SQL Injection attacks. These parameters may contain SQL Keywords, e.g. Insert, update or SQL control Characters such as quotation marks and semicolons.

Generous privileges: normally in database the Privileges are defined as the rules to state which database subject has access to which object and what operation are associated with user to be allowed to perform on the objects. Typical privileges include allowing execution of actions, e.g. select, insert, update, delete, drop, on certain objects. Web applications open database connections using the specific account for accessing the database. An attacker who bypasses authentication gains privileges equal to the Accounts. The number of available attack methods and affected objects increases when more privileges are given to the account. The worst case happen if an account can connect to system that is associated with the system administrator because normally has all privileges.

Uncontrolled variable size: if variables allow storage of data be larger than expected consequently allow attackers to enter modified or faked SQL statements. Scripts that do not Control variable length may even open the way for attacks, Such as buffer overflow.

Error message: error messages that are generated by the back-end database or other server-side programs may be returned to the client-side and presented in the web browser. These messages are not only useful during development for debugging purposes but also increase the risks to the Application. Attackers can analyze these messages to gather Information about database or script structure in order to construct their attack.

Variable Orphism: the variable should not accept any data type because attacker can exploit this feature and store malicious data inside that variable rather than is suppose to be. Such variables are either of weak type, e.g. Variables in Php, or are automatically converted from one type to another by the remote database.

Dynamic SQL: SQL queries dynamically built by scripts or programs into a query string. Typically, one or more Scripts and programs contribute and finally by combining User input such as name and password, make the where Clauses of the query statement. The problem is that query Building components can also receive SQL keywords and Control characters. It means attacker can make a completely different query than what was intended.

Client-side only control: if input validation is implemented in client-side scripts only, then security functions of those scripts can be overridden using cross-site scripting. Therefore, attackers can bypass input validation and send invalidated input to the server-side.

Stored procedures: they are statements which are stored in database. The main problem with using these Procedures is that an attacker may be able to execute them and damage database as well as the operating system and even other network components. Usually attackers know System stored procedures that come with different and almost easily can execute them.

Into out file support: some of RDBMS benefit from into out file clause. In this condition an attacker can manipulate SQL queries then they produce a text file containing query results. If attackers can later gain access to this file, they can abuse the same information, for example, bypass authentication.

Multiple statements: if the database supports union, attacker has more chance because there are more attack methods for SQL injection. For instance, an additional insert statement could be added after a select statement, causing two different queries to be executed. If this is performed in a login form, the attacker may add him or herself to the table of users.
Sub-selects: supporting sub-selects is weakness for RDBMS when SQL injection is considered. For example, additional select clauses can be inserted in where clauses of the original select clause. This weakness makes the web application more vulnerable, so they may be penetrated by malicious users easily.

5. SQL injection attack Types [8]

There are different methods of attacks that depending on the goal of attacker are performed together or sequentially. For a successful SQLIA the attacker should append a syntactically correct command to the original SQL query.

Tautologies: this type of attack injects SQL tokens to the conditional query statement to be evaluated always true. This type of attack used to bypass authentication control and access to data by exploiting vulnerable input field which use where clause.
"select * from employee where userid = '112' and Password ='aaa' or '1'='1'"
As the tautology statement (1=1) has been added to the Query statement so it is always true.

Legal/logically incorrect queries: when a query is rejected, an error message is returned from the database including useful debugging information. This error messages help attacker to find vulnerable parameters in the application and consequently database of the application. In fact attacker injects junk input or SQL tokens in query to produce syntax error, type mismatches, or logical errors by purpose.

Union query: by this technique, attackers join injected query to the safe query by the word union and then can get data about other tables from the application. Suppose for example that the query executed from the server is the following: select name, phone from users where id=$id by injecting the following id value:
$id=1 union all select creditcardnumber, 1 from Creditcartable
We will have the following query:
Select name, phone from users where id=1 union all select creditcardnumber, 1 from creditcartable which will join the result of the original query with all the credit card users.

Piggy-backed queries: in this type of attack, intruders exploit database by the query delimiter, such as ";", to append extra query to the original query. With a successful attack database receives and execute a multiple distinct queries. Normally the first query is legitimate query,
Whereas following queries could be illegitimate. So attacker can inject any SQL command to the database. In the following example, attacker inject " 0; drop table user " into the pin input field instead of logical value. Then the application would produce the query:
Select info from users where login='doe' and Pin=0; drop table users because of ";" character, database accepts both queries and executes them. The second query is illegitimate and can drop users table from the database.

Stored procedure: stored procedure is a part of database that programmer could set an extra abstraction layer on the database. As stored procedure could be coded by programmer, so, this part is as inject able as web application forms. Depend on specific stored procedure on the database there are different ways to attack. In the following example, Attacker exploits parameterized stored procedure. Create procedure dbo.isauthenticated
@username varchar2, @pass varchar2, @pin int
AsExec("select accounts from users Where login='" +@username+ "' and pass='" +@password+"' and pin=" +@pin);
Go For authorized/unauthorized user the stored procedure returns true/false. As an SQLIA, intruder input " ' ; Shutdown; - -" for username or password. Then the stored procedure generates the following query:
Select accounts from users where login='doe' and pass=' '; shutdown; -- and pin=
After that, this type of attack works as piggy-back attack. The first original query is executed and consequently the second query which is illegitimate is executed and causes database shut down. So, it is considerable that stored procedures are as vulnerable as web application code.

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

Timing Attacks: A timing attack lets attacker gather information from a database by observing timing delays in the database's responses. This technique by using if-then statement cause the SQL engine to execute a long running query or a time delay statement depending on the logic injected. This attack is similar to blind injection and attacker can then measure the time the page takes to load to determine if the injected statement is true. This technique uses an if-then statement for injecting queries. WAITFOR is a keyword along the branches, which causes the database to delay its response by a specified time. For example, in the following query:
declare @s varchar(8000) select @s = db_name() if(ascii(substring(@s, 1, 1)) & (power(2, 0))) > 0 waitfordelay '0:0:5'
Database will pause for five seconds if the first bit of the first byte of the name of the current database is 1. Then code is then injected to generate a delay in response time when the condition is true. Also, attacker can ask a series of other questions about this character. As these examples show, the information is extracted from the database using a vulnerable parameter.

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

SELECT accounts FROM users WHERE login='' ANDpin=0; exec (char(0x73687574646f776e))

This example use the char () function and ASCII hexadecimal encoding. The char () function takes hexadecimal encoding of character(s) and returns the actual character(s). The stream of numbers in the second part of the injection is the ASCII hexadecimal encoding of the attack string. This encoded string is translated into the shutdown command by database when it is executed.

II RELATED WORK

Most of existing techniques, such as filtering, information-flow analysis, penetration testing, and defensive coding, can detect and prevent a subset of the vulnerabilities that lead to SQLIAs. In this section, we list the most relevant techniques. We reviewed a number of electronic journal articles from IEEE journals and from ACM, and gathered some information from web sites to gain sufficient knowledge about SQL injection attacks. Following are the papers from which we covered different important strategies to prevent SQL injection attacks.

1. From [3], we covered the techniques for SQL injection discovery. This paper also covered very well the SQL parse tree validation that we mentioned. Parse tree parses the query based on defined rules and verify whether query is valid or not valid i.e. query is injected or not.
2. From [2], we covered the techniques to check and sanitize input query using SQLCHECK, it use the augmented queries and SQLCHECK grammar to validate query.
3. From [4], we covered techniques to remove vulnerabilities from code. This paper proposed an automated method for removing SQL injection vulnerabilities from code by converting plain text SQL statements into prepared statements. Prepared statements restrict the way that input can affect the execution of the statement. An automated solution allows developers to remove SQL injection vulnerabilities by replacing vulnerable code with generated secure code.
4. From [5], we covered the techniques covered an original method to protect application automatically from SQL injection attacks. The original approach combines static analysis, dynamic analysis, and automatic code re-engineering to secure existing properties.
5. From [1], we covered the techniques to protect store procedures from SQL attacks. This paper provided novel approach to shield the stored procedures from attack and detect SQL injection. This method combines runtime check with static application code analysis so that they can eliminate vulnerability to attack. The key behind this attack is that it alters the structure of the original SQL statement and identifies the SQL injection attack. The method is divided in two phases, one is offline and another one is runtime. In the offline phase, stored procedures use a parser to pre-process and detect SQL statements in the execution call for runtime analysis. In the runtime phase, the technique controlled all runtime generated SQL queries related with the user input and checks these with the original structure of the SQL statement after getting input from the user. Once this technique detects the malicious SQL statements it prevents the access of these statements to the database and provides details about attack.
6. We reviewed various SQL injection detection and prevention tools and reviewed their comparison of tools based on attack types and deployment requirements.

1. SQL Injection Discovery Technique [3]

It is not compulsory for an attacker to visit the web pages using a browser to find if SQL injection is possible on the site. Generally attackers build a web crawler to collect all URLs available on each and every web page of the site. Web crawler is also used to insert illegal characters into the query string of a URL and check for any error result sent by the server. If the server sends any error message as a result, it is a strong positive indication that the illegal special meta character will pass as a part of the SQL query, and hence the site is open to SQL Injection attack. For example Microsoft Internet Information Server by default shows an ODBC error message if an any meta character or an unescaped single quote is passed to SQL Server. The Web crawler only searches the response text for the ODBC messages.

2. SQL Parse Tree Validation [3]

A parse tree is nothing but the data structure built by the developer for the parsed representation of a statement. To parse the statement, the grammar of that parse statement's language is needed. In this method, by parsing two statements and comparing their parse trees, we can check if the two queries are equal. When attacker successfully injects SQL into a database query, the parse tree of the intended SQL query and the resulting SQL query generated after attacker input do not match. A parse tree is a data structure for the parsed representation of a statement. Parsing a statement requires the grammar of the language that the statement was written in. By parsing two statements and comparing their tree structures, we can determine if the two queries are equal. When a malicious user successfully injects SQL into a database query, the parse tree of the intended SQL query and the resulting SQL query do not match. By intended SQL query, we mean that when a programmer writes code to query the database, he/she has a formulation of the structure of the query. The programmer-supplied portion is the hard-coded portion of the parse tree, and the user-supplied portion is represented as empty leaf nodes in the parse tree. These nodes represent empty literals. What he/she intends is for the user to assign values to these leaf nodes. These leaf nodes can only represent one node in the resulting query, it must be the value of a literal, and it must be in the position where the holder was located. An example of his/her intended query is given in Figure 2. This parse tree corresponds to the query, Select * from userregistration where name=? and password=?. The question marks are place holders for the leaf nodes she requires the user to provide. While many programs tend to be several hundred or thousand lines of code, SQL (structured query language) statements are often quite small. This affords the opportunity to parse a query without adding significant overhead. The parse tree for intended query is given in figure 2 and with inputs is given in figure 3.

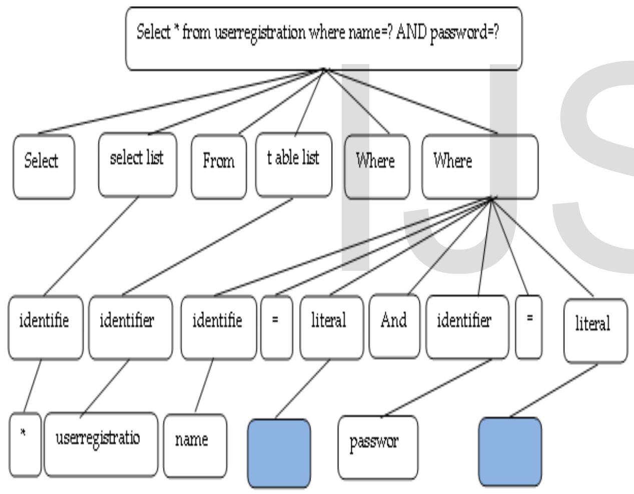

Figure 2 SELECT query with two user inputs

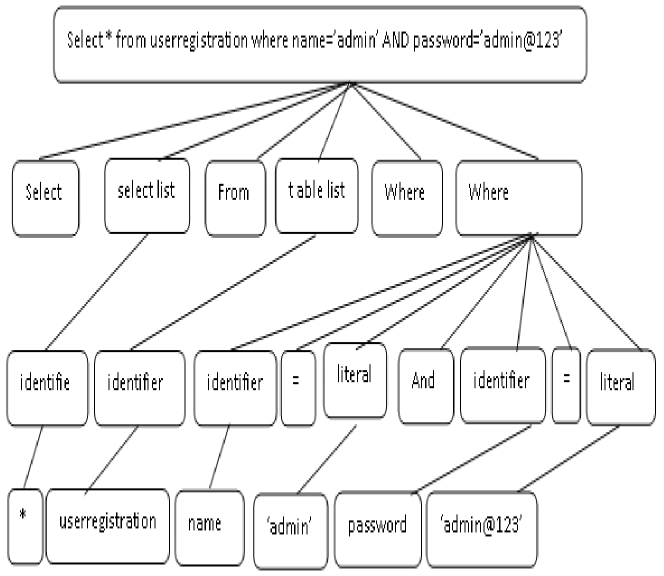

Figure 3 the same SELECT query as in Figure 2, with the user input inserted

3. Approach for SQL Check [2]

Web applications have SQL injection vulnerabilities because they do not sanitize the inputs they use to construct structured output. The code is for an online store. The website provides user input field to allow the user to keep their credit card information which user can use for future purchases. Replace method is used to escape the quotes so that any single quote characters in the input is considered as a literal and not a string delimiters. Replace method is intended to block attacks by preventing an attacker from ending the string and adding SQL injection code. Although, card type is a numeric column, if an attacker passes 2 OR 1=1" as the card type, all account numbers in the database will be returned and displayed.

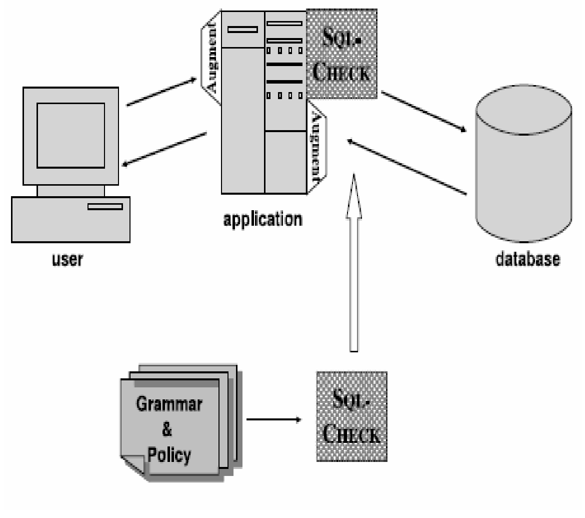

Figure 4 System architecture of SQLCHECK

In this approach they track through the program, the substrings receive from user input and sanitize that substrings syntactically. The aim behind this program is to block the queries in which the input substrings changes the syntactic structure of the rest of the query. They use the meta-data to watch user's input, displayed as '_' and '_' to mark the end and beginning of the each user input string. This meta-data pass the string through an assignments, and concatenations, so that when a query is ready to be sent to the database, it has a matching pairs of markers that identify the substring from the input. These annotated queries called an augmented query. To build a parser for the augmented grammar and attempt to parse each augmented query, a parse generator used. Query meets the syntactic constraints and considered legitimate if it parses successfully. Else, it fails the syntactic constraints and interprets it as SQL injection attack. The system architecture of the checking system shows in Figure 4. Grammar of the output language is used to build SQLCHECK and a policy mentioned permitted syntactic forms, it resides on the web server and taps generated queries. In spite of the input's source, each input which is to be passed into some query, gets augmented with the meta-characters '_' and '_'. Finally application creates augmented queries, which SQLCHEKCK attempts to parse, and if a query parses successfully, SQLCHECK sends it the meta-data to the database, else the query get rejected.

4. SQL Injection Detection and Prevention Tools

Although developers deploy defensive coding but they are not enough to stop SQLIAs to web applications so researchers have proposed some of tools to assist developers.

JDBC-Checker [16] was not developed with the intent of detecting and preventing general SQLIAs, but can be used to prevent attacks that take advantage of type mismatches in a dynamically-generated query string. As most of the SQLIAs consist of syntactically and type correct queries so this technique would not catch more general forms of these attacks.

CANDID [11] modifies web applications written in Java through a program transformation. This tool dynamically mines the programmer-intended query structure on any input and detects attacks by comparing it against the structure of the actual query issued. CANDID's natural and simple approach turns out to be very powerful for detection of SQL injection attacks.

In **SQL Guard** [3] and **SQL Check** [2] queries are checked at runtime based on a model which is expressed as a grammar that only accepts legal queries. SQL Guard examines the structure of the query before and after the addition of user-input based on the model. In SQL Check, the model is specified independently by the developer. Both approaches use a secret key to delimit user input during parsing by the runtime checker, so security of the approach is dependent on attackers not being able to discover the key. In two approaches developer should to modify code to use a special intermediate library or manually insert special markers into the code where user input is added to a dynamically generated query.

AMNESIA [15] combines static analysis and runtime monitoring. In static phase, it builds models of the different types of queries which an application can legally generate at each point of access to the database. Queries are intercepted before they are sent to the database and are checked against the statically built models, in dynamic phase. Queries that violate the model are prevented from accessing to the database. The primary limitation of this tool is that its success is dependent on the accuracy of its static analysis for building query models.

WebSSARI [17] 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 [10] is another tool that was implemented for java. Despite of other tool, chases string instead of character for taint information. SecurityFly tries 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.

Positive tainting [8] not only focuses on positive tainting rather than negative tainting but also it is automated and does need developer intervention. Moreover this approach benefits from syntax-aware evaluation, which gives developers a mechanism to regulate the usage of string data based not only on its source, but also on its syntactical role in a query string.

IDS [14] use an Intrusion Detection System (IDS) to detect SQLIAs, based on a machine learning technique. The technique builds models of the typical queries and then at runtime, queries that do not match the model would be identified as attack. This tool detects attacks successfully but it depends on training seriously. Else, many false positives and false negatives would be generated.
Another approach in this category is **SQL-IDS [18]** which focus on writing specifications for the web application that describe the intended structure of SQL statements that are produced by the application, and in automatically monitoring the execution of these SQL statements for violations with respect to these specifications.

Swaddler [12] 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.

5. Comparison of SQL Injection Detection/Prevention

Tools Based on Attack Types [19]

Proposed tools were compared to assess whether it was capable of addressing the different attack types. It is noticeable that this comparison is based on the articles not empirically experience.
Table 1 summarize the results of this comparison. The symbol "*" is used for tool that can successfully stop all attacks of that type. The symbol "**-**" is used for tool that is not able to stop attacks of that type. The symbol "**o**" refers to tool that the attack type only partially because of natural limitations of the underlying approach.

Tool / Attack		SQL IDS[18]	Swadder[12]	IDS[14]	CANDID[11]	AMNESIA[15]	SQL Check[2]	SQL Guard[3]	JDBC Checker[16]	WebSSARI[17]	Securifly[10]	Positive Tainting[8]
1	Tautologies	*	o	o	o	*	*	*	o	*	o	*
2	Illegal/ Incorrect	*	o	o	o	*	*	*	o	*	o	*
3	Piggy-back	*	o	o	o	*	*	*	o	*	o	*
4	Union	*	o	o	o	*	*	*	o	*	o	*
5	Store Procedure	*	o	o	o	-	-	-	o	*	o	*
6	Infer	*	o	o	o	*	*	*	o	*	o	*
7	Alter Encodings	*	o	o	o	*	*	*	o	*	o	*

Table 1 Comparison of tools with respect to attack types

As the table shows the stored procedure is a critical attack which is difficult for some tools to stop it. It is consisting of queries that can execute on the database. However, most of tools consider only the queries that generate within application. So, this type of attack make serious problem for some tools.

6. Comparison of SQL Injection Detection/Prevention

Tools Based on Deployment Requirement each tool with respect to the following criteria was evaluated: (1) Does the tool require developers to modify their code base? (2) What is the degree of automation of the detection aspect of the tool? (3) What is the degree of automation of the prevention aspect of the tool? (4) What infrastructure (not including the tool itself) is needed to successfully use the tool? The results of this classification are summarized in Table 2.

Table 2 determines the degree of automation of tool in detection or prevention of attacks. Actually automatically detection and prevention is ability of tool that provides user satisfaction. Also table shows that which tool needs to modify the source code of application.

No	Tool	Modify Code Base	Detection	Prevention	Additional Infrastructure
1	AMNESIA[15]	No	Auto	Auto	None
2	IDS[14]	No	Auto	Generate report	IDS system-Training set
3	JDBC Checker[16]	No	Auto	Code suggestion	None
4	SECURIFLY [10]	No	Auto	Auto	None
5	SQLCHECK[2]	Yes	Semi Auto	Auto	Key management
6	SQL Guard [3]	Yes	Semi Auto	Auto	None
7	WEBSSARY [17]	No	Auto	Semi Auto	None
8	CANDID[11]	No	Auto	Auto	None

9	SQL_IDS[18]	No	Auto	N/A	None
10	Swaddler [12]	No	Auto	Auto	Training
11	Positive Tainting[8]	No	Auto	Auto	None

Table 2 Comparison of tools based on deployment requirements

III CONCLUSION AND FUTURE WORK

SQL Injection is most challenging threat to the web application and many solutions to these attacks have been proposed since the emerging of SQL injection. But no solution provides security to full extent. SQL Injection is a common technique that attacker use to attack on web applications. These attacks modify the SQL queries. This paper presents what is SQL injection, how it works, SQLIA Intent, types of SQLIA and explores SQL injection detection tools in related work. Comparison of tools is carried in terms of their ability to detect the SQLIA. Furthermore, the tools were compared based on attacks types and deployment requirements. AMNESIA is best for SQLIAs detection as it can be detect all types of attacks at both static and dynamic phase.
In future, these detection tools can be used to detect SQL Injection attacks. These techniques can also provide as defense mechanisms for providing security against SQLIAs. In addition, more research is needed to improve analysis technique for providing better detection and prevention against strong SQLIAs.

REFERENCES

[1] K. Wei, M. Muthuprasanna and S. Kothari, "Preventing SQL Injection Attacks in Stored Procedures", Dept. of Electrical and Computer Engineering, Iowa State University Ames, IA – 50011, 2007.
[2] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications", University of California, Davis, Jan. 2006.
[3] G. T. Buehrer, B. W. Weide and P. A. G. Sivilotti, "Using Parse Tree Validation to Prevent SQL Injection Attacks", Computer Science and Engineering, The Ohio State University, Columbus, OH 2005.
[4] S. Thomas and L. Williams, "Using Automated Fix Generation to Secure SQL Statements", Department of Computer Science, North Carolina State University, Raleigh, NC, USA, 2007.
[5] E. Merlo, D. Letarte and G.Antoniol, "Automated Protection of PHP Applications against SQL-injection Attacks", Department of Computer Science, Ecole Polytechnique de Montreal,C.P. , 2007.
[6] S .Narang, S. Sharma and R. P. Mahapatra, "Prevention of SQL Injection in E-Commerce", Computer Science & Engineering Department, 2 IT Department. SRM University, NCR Campus, Modi Nagar, (UP) India, Sep 2015.
[7] C. Cerrudo, "Manipulating Microsoft SQL server using SQL injection", 2002.
[8] W. G. J. Halfond, J. Viegas and A. Orso, "A Classification of SQL Injection Attacks
and Countermeasures", College of Computing Georgia Institute of Technology.
[9] S. Patil and N Agrawal, "Web Security Attacks and Injection- A Survey", Department of Computer Science & Engineering, NRI Institute of Science & Technology, Bhopal, India, Feb 2015.
[10] M. Martin, B. Livshits and M. S. Lam, "Finding Application Errors and Security Flaws Using PQL: A Program Query Language", Computer Science Department, Stanford University.
[11] S. Bandhakavi, P. Bisht and P. Madhusudan, "CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluation", University of Illinois
Urbana-Champaign, USA, 2007.
[12] M. Cova, D. Balzarotti, V. Felmetsger and G. Vigna, "Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications", Department of Computer Science, University of California Santa Barbara, CA, USA, 2007.
[13] A. Bhanderi, N. Rawal, "A Review on Detection Mechanisms for SQL Injection Attacks",
P.G. Student, Dept. of Computer Engineering, C.G.P.I.T, Uka Tarasadia University, Bardoli, Gujarat, India, Associate Professor, Dept. of Computer Engineering, C.G.P.I.T, Uka Tarasadia University, Bardoli, Gujarat, India.
[14] F. Valeur, D. Mutz and G. Vigna, "A Learning-Based Approach to the Detection of SQL Attacks", Reliable Software Group, Department of Computer Science, University of California, Santa Barbara, July 2005.
[15] W. G. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks", College of Computing Georgia Institute of Technology, Nov 2005.
[16] C. Gould, Z. Su and P. Devanbu, "JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications", Department of Computer Science, University of California, Davis, 2004.
[17] 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", 2004.
[18] K. Kemalis and T. Tzouramanis, "SQL-IDS: A Specification-based Approach for SQL-Injection Detection", Department of Information & Communication, Systems Engineering, University of the Aegean, Karlovassi, Samos, 83200, Greece, 2008.

[19] A. Tajpour, S. Ibrahim, M. Sharifi, "Web Application Security by SQL Injection DetectionTools", Advanced Informatics School, University Technology Malaysia, Malaysia.