Abstract—A SQL injection attack occurs on database-driven websites when unauthorized SQL queries are executed on vulnerable sites. This attack can bypass a firewall and can affect a fully patched system. For this to happen port 80, the default web port, is the only thing required. SQL injection attacks target a specific web application where the vulnerability of the relational database is either known or discovered by the attacker. The developer creates an application that includes a relational database. The attackers will try to hack the application. SQL injection is a code injection technique that exploits a security vulnerability occurring in the database layer of an application. The vulnerability is present when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly executed. It is an instance of a more general class of vulnerabilities that can occur whenever one programming or scripting language is embedded inside another. SQL Injection is one of the most common application layer attack techniques used today. In this research work, we have proposed a new algorithmic approach to early detection and mechanism for the avoidance of the SQL Injections Attacks. We have implemented the mechanism with the use of Fuzzy Parameters and set of rules. The results obtained are good in terms of security and execution time of the scripts.
and manipulate the contents of database (using Data Manipulation Language statements, or ‘DML’).

II. RELATED WORK

Over years, many tools for detection and prevention of SQL Injection attacks have been developed. AMNESIA developed By Halfond and Orso in [4] is a detection and prevention tool for SQL injection attack. It uses static analysis and runtime monitoring for the purpose. The tool builds a model of the legitimate queries at each hotspot i.e. where SQL queries are issued to database engine and monitors the application at runtime to ensure that all generated queries match the statically-generated model. In [5], a tool named CANDID is proposed for detecting SQL injection. The tool dynamically infers the programmer-intended query structure on any input, and detects attacks by comparing them against the intended query structure. In [6], SQLRand uses instruction set randomization to detect and abort queries with injected code and every SQL keyword is joined with a random integer to mislead the attacker. The proposed technique in [7] prevents SQLIA in stored procedures by combining static application code analysis with runtime validation. In the static part, a stored procedure parser is designed and it instruments the necessary statements in order to compare the original SQL statement structure to that including user inputs for every SQL statement which depends on user inputs. The technique abstracts the intended SQL query behavior in an application in the form of an SQL-graph and this graph is then validated against all the different user inputs at runtime to capture all malicious SQL queries, before they are sent for execution. An efficient technique is presented in [2] for detecting and preventing SQL Injection attack using pattern matching algorithm. Pattern matching identifies or detects any anomaly packet from a sequential action, as the malicious code includes many anomaly packets or strings. The technique proposed in [3] uses a new middle-ware-based prevention mechanism: SQLIMW. The SQLIMW avoids SQL-Injection attack from the programmer to the server. Hash function is used to replace encryption for better security. Furthermore, by combining the hash with XOR, it protects username, password and private key of SQLIMW. The proposal provides better security and efficiency. This kind of middleware is transparent to the programmer is not limited to sign-on authentication mechanism or single sign-on system, it can exist in any layer of Web application system exchanging information with the database. Almost every solution given to detect or prevent SQL Injection attacks is either for application layer or for database layer. But none of them provide security at both application and database layers. Also very little emphasis is laid on preventing SQL Injection attacks in stored procedures. [7] Although the mechanism of SQLIA is the same for both stored procedure and application layer program, the same detection technique could not be applied to stored procedures, because of limited programmability of stored procedures and the technique’s usability and deployability. Many existing techniques, such as filtering, information-flow analysis, penetration testing, and defensive coding, can detect and prevent a subset not all of the vulnerabilities that lead to SQLIAs.[1] Another technique provides a two phase security to the application, so that, if one phase is compromised, the second phase can still prevent the attack. Here an efficient scheme for detecting and preventing SQL Injection attacks has
been introduced. This scheme provides security from both the frontend as well as the backend of the application so that, if security is compromised at one end, the second end can still prevent SQL injection attacks.

III PROPOSED WORK

Here is an efficient and new algorithmic approach is developed for early detection and mechanism for the avoidance of the SQL Injections Attacks. We have implemented the mechanism with the use of Fuzzy Parameters and set of rules. The two rules stated below.

RULE - 1

if PositiveAttempt $\Rightarrow$ (0, 0, 1)

$\Rightarrow$ (FALSE, FALSE, TRUE)

NegativeAttempts $\Rightarrow$ (111*)

$\Rightarrow$ (FALSE, FALSE, FALSE*)

PositiveAttempt $\Rightarrow$ (0, 0, 1)

$\Rightarrow$ (FALSE, FALSE, TRUE)

OR

NegativeAttempt $\Rightarrow$ (1, 1, 0) $\Rightarrow$ (TRUE, TRUE, FALSE)

Exit with Message ("Not Allowed")

RULE - 2

(IdentifyUser) $\Rightarrow$

Navigate and Generate Previous User Attempts x $\Rightarrow$ (Fetch (AllUserAttempts))

if (ANY(x) = "SQL Injection")

Exit with Message "Blacklist User Profile"

In the above rules we categorize the user in two groups: Authorized and unauthorized. Authorized users are those who are registered and access the database and attempt are tautology i.e. true. Unauthorized users are those who want to access the database and try to destroy them. To check the unauthorized all attempt are false. The user who try three false attempt are blacklisted and can’t login again. All the attempt are saved in database. The total execution time also saved in the database. The encryption is used for the password protection so that any user can’t able to see the password.

IV METHODOLOGY USED:

The steps are:

1. Collection of the Training Data Set of Users and Cheatsheet for Analysis
2. The Training Data Set Consists of the URLs for investigation
3. Generation of the patterns and keys
4. Deep Analysis on each parameter.
5. Applying the proposed model on the Training data set
6. Fetch Results
7. Data Interpretation

Firstly we collect all the entries of the user attempt stored in our database. User database consists of user name and password field that is linked with the URL of the website. It also consists of cheat sheet for analysis. After that set of keys is generated with respect of user attempt. Encryption is used to generated keys and pattern are generated according to the string match. Our next step is to analyze various parameter i.e. Fuzzy set Parameter and their generated pattern. After that our model applied on the data and result are gathered. Table below shows the database structure. The table shows the classification of the attack according to SQL Injection string. SQLI string has the penetration level. It specify the risk included with SQL injection string.
Table 1.1 Database Schemas and Structure

<table>
<thead>
<tr>
<th>SQL Injection String</th>
<th>Penetration Level</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>admin' --</td>
<td>100</td>
<td>Attempt Admin Access</td>
</tr>
<tr>
<td>admin' #</td>
<td>90</td>
<td>Attempt Admin Access</td>
</tr>
<tr>
<td>admin'/*</td>
<td>80</td>
<td>Attempt Database Exploitation</td>
</tr>
<tr>
<td>' or 1=1--</td>
<td>90</td>
<td>Database Bypass</td>
</tr>
<tr>
<td>' or 1=1#</td>
<td>100</td>
<td>Access Database Tables</td>
</tr>
<tr>
<td>' or 1=/*</td>
<td>70</td>
<td>Database Access</td>
</tr>
<tr>
<td>') or '1=1'--</td>
<td>80</td>
<td>Database Exploitation</td>
</tr>
<tr>
<td>') or ('1='1'--</td>
<td>50</td>
<td>Database Exploitation</td>
</tr>
<tr>
<td>1' ORDER BY 1--++</td>
<td>60</td>
<td>Access Database and Tables</td>
</tr>
<tr>
<td>1' ORDER BY 1--++</td>
<td>90</td>
<td>DB Exploitation</td>
</tr>
<tr>
<td>1' ORDER BY 2--++</td>
<td>60</td>
<td>DB Exploitation</td>
</tr>
<tr>
<td>1' ORDER BY 3--++</td>
<td>40</td>
<td>DB Exploitation</td>
</tr>
<tr>
<td>-I' UNION SELECT 1,2,3--++</td>
<td>80</td>
<td>DB Exploitation</td>
</tr>
<tr>
<td>1' ORDER BY 4--++</td>
<td>90</td>
<td>DB Exploitation</td>
</tr>
<tr>
<td>1' GROUP BY username HAVING 1=1--</td>
<td>80</td>
<td>Authentication Bypass</td>
</tr>
<tr>
<td>1' or '1'='1'--</td>
<td>50</td>
<td>Authentication Bypass</td>
</tr>
<tr>
<td>1' OR '1'='1'</td>
<td>60</td>
<td>AccessUsernamewithout Authentication</td>
</tr>
</tbody>
</table>

IV RESULT

The below table shows that which user is authorized one and unauthorized one and also shows that execution time of our approach and classical approach. This table also tell us about the type of attack.

Table 1.2 USER ATTEMPT TABLE

Table 1.3 Comparison between the Classical approach(red line) and Purposed approach(blue)