Marathwada Shikshan Prasarak Mandal’s

**Deogiri Institute of Engineering and Management Studies,**

**Aurangabad**

**Seminar Report**

On

**Detecting Data Leaks via SQL Injection Prevention on an website**

Submitted By

**Rushikesh Shejwal (36019)**

**Dr. Babasaheb Ambedkar Technological University**

**Lonere (M.S.)**



Department of Computer Science and Engineering

**Deogiri Institute of Engineering and Management Studies,**

**Aurangabad**

(2019- 2020)

**Seminar Report on**

**Detecting Data Leaks via SQL Injection Prevention on an website**

Submitted By

**Rushikesh Shejwal (36019)**

**In partial fulfillment of**

**Bachelor of Technology**

**(Computer Science & Engineering)**

Guided By

**Mr. S.G. Shah**

Department of Computer Science & Engineering

**Deogiri Institute of Engineering and Management Studies,**

**Aurangabad**

(2019- 2020)

**CERTIFICATE**

This is to certify that, the Seminar entitled “**Detecting Data Leaks via SQL Injection Prevention on an websit**e” submitted by **Rushikesh Shejwal** is a bonafide work completed under my supervision and guidance in partial fulfillment for award of Bachelor of Technology (Computer Science and Engineering) Degree of Dr. Babasaheb Ambedkar Technological University, Lonere.

Place: Aurangabad

Date: 16/10/2019

**Mr. S. G. Shah Mr. S.B. Kalyankar**

**Guide Head**

**Dr. Ulhas D. Shiurkar**

**Director,**

**Deogiri Institute of Engineering and Management Studies,**

**Aurangabad**

**Abstract**

The objective of this project is to prevent SQL injection while firing queries to database and to make the database secured. This system is online so no need of implementation. It can be accessed through internet from anywhere. The system uses SQL Injection mechanism to keep the data safe and secured.

The highlighted part here is encryption of card data using AES (Advanced Encryption Standard) technique. The Online Shop secures the card payment and won’t let the card data to get hacked. While user doing a card payment, all the card data is encrypted and then stored into database. System also keeps user details in an encryption form using AES encryption. The system is built of handling SQL Injection capabilities which doubles the security of database and prevents from injection hacking codes into the database. Here, the project files and a database file will be stored into the Azure cloud which will form a connection between application and cloud server via internet. The project will be accessed in the web browser through link.

###### Contents

1. **INTRODUCTION 6** 
   1. Introduction
   2. SQL Injection attacks Principle
   3. Types of attack

1. **LITERATURE SURVEY** 11

2.1Types of vulnerabilities

2.2 Types of Attacks

2.3 Comparative Analyses

1. **Details of working and Architecture 16**

3.1working

1. **CONCLUSIONS 19**

**REFERENCES**

**ACKNOWLEDGEMENT**

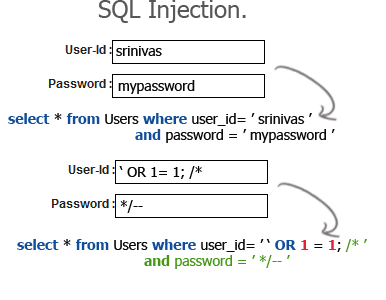
**1. INTRODUCTION**

Web applications are often vulnerable for attackers easily access to the application's underlying database. SQL injection attack occurs when a malicious user, through specifically crafted input, causes a web application to generate and send a query that functions differently than the programmer intended.

SQL Injection Attacks (SQLIAs) have known as one of the most common threats to the security of database-driven applications. So there is not enough assurance for confidentiality and integrity of this information. SQLIA is a class of code injection attacks that take advantage of lack of user input validation. In fact, attackers can shape their illegitimate input as parts of final query string which operate by databases. Financial web applications or secret information systems could be the victims of this vulnerability because attackers by abusing this vulnerability can threat their authority, integrity and confidentiality. So, developers addressed some defensive coding practices to eliminate this vulnerability but they are not sufficient. For preventing the SQLIAs, defensive coding has been offered as a solution but it is very difficult. Not only developers try to put some controls in their source code but also attackers continue to bring some new ways to bypass these controls. Hence it is difficult to keep developers up to date, according the last and the best defensive coding practices. On the other hand, implementing of defensive coding is very difficult and need to special skills and also erring. These problems motivate the need for a solution to the SQL injection problem. Researchers have proposed some tools to help developers to compensate the shortcoming of the defensive coding [7, 10, 12]. The problem is that some current tools could not address all attack types or some of them need special deployment requirements.

SQL injection is a method of hacking in which malformed SQL queries are produced through unsanitized user-input. Though an application can easily prevent SQL injection by validating input, too many avenues for data exchange exist in the current web model and the modes of SQL injection vary broadly. In 2002, the Computer Security Institute and the Federal Bureau of Investigation (United States) conducted a survey that discovered that 60% of online databases are subject to security breaches each year. In particular, SQL Injection attacks have featured at the top of OWASPs (Open Web Application Security Project).

SQL Injection Attacks First, we propose a classification for SQL Injection Attacks. Such classifications can be found. Server variables are a collection of variables that defines the HTTP Request, environment, and various other network parameters. These include the two most common form submission methods, GET and POST, as well as other more intricate injection avenues such as HTTP header variables and sessions. The bulk of SQL Injection attempts are made through these server variables, either through entering malicious input into the client-end of the application or by crafting their own request to the server. Example of SQL injection in the fig. 1.1



**Fig. 1.1**

SQL injection was an attack in which malicious code was embedded in strings that were later passed to database end for parsing and execution. The malicious data produced database query results and acquired sensitive information, such as account credentials or internal business data. Through analyzing the principle of SQL injection attacks, prevention method was proposed to solve the double defense through the browser and server ends. Jang exhibited a novel scheme that automatically transformed web applications, rendering them safe against SQL injection attacks. Mishear demonstrated two non-web-based SQL injection attacks, and presented XML-based authentication approach which could handle.

**1.2 SQL INJECTION ATTACKS PRINCIPLE**

SQL databases are attacked against by direct insertion of code into input variables, which are consisted of the primary form of SQL injection. Some attackers insert malicious code into a string, the server will not respond input values when the input string contains such an SQL statement. The method to directly attack is to inject the ill-disposed codes into strings, which are usually stored in a table or as metadata. And then the attacking code will run when the attacked strings are called in the form of concatenated subsequently into a dynamic SQL command. A. SQL injection Attack The following example shows how SQL injection attacks realize. SQL injections utilize weakness of a bank's application to misguide the application into running a database backend query or command. Usually, an application of a bank's operation has a menu, which is used for searching customer's personal information, such as the telephone number. The application will execute an SQL query in the database backend.

• SELECT client\_name, address, date \_ of birth WHERE tel no=123456 If user enters the string "123456 or 1=1," then the SQL query passes to the database as follows:

• SELECT client\_name, address, date\_ofbirth WHERE tel no=123456 or 1=1

The condition 1 = 1 is always true in database. The query will return all rows in the table, which is not the original intention. The application can be changed so that it accepts one numeric value only. SQL injection attacks can be mitigated by ensuring proper application design, especially in modules that require user input to run database queries or commands. The below script case shows a typical SQL injection. The script creates an SQL query by concatenating hard-coded strings together with an input string. The user is prompted to input the name of a city. If the user enters Seattle, the query is assembled by the script looks like the follows:

• SELECT \* FROM OrdersTable WHERE Ship City = 'Seattle' However, assume that the user enters the following script:

• SELECT \* FROM OrdersTable WHERE ShipCity = , Seattle '; drop table OrdersTable--'

The semicolon (;) states the end of one query and the start of another query, the double hyphen (--) denotes that the rest of the present line is a comment and should be disregarded. If the edited code is syntactically correct, it will be executed by the server side. When SQL Server runs the statement, SQL Server will select all records in OrdersTable where ShipCity is Seattle, and then SQL Server will drop OrdersTable. As long as injected SQL code is syntactically correct, tampering cannot be captured programmatically. Therefore, all users input and check code that implements constructed SQL commands in the server should be verified. B. Common Detection Method SQL injection attacks are usually accessed to page port, which looks like ordinary Web page login. The general firewall cannot detect SQL injection attacks and therefore needs some artificial means to enhance the detection of injection attack.

The detection methods are as following:

• Check llS log SQL attackers usually access particular page files, and then increase many log files. As llS logs record information of user's IP address and access files, we can judge whether attackers has SQL injection attacks by looking at the log file size and content.

• Check database Don't put the sensitive data into the database, if attacker uses SQL software tools to inject database, which will generate some temporary tables. We can check whether SQL injection attacks are happened by checking database table structure and content.

• Check user's input Use regular expression or limit the length to validate user input string, and always check user's input by testing format, length, type, and range.

**1.3 TYPES OF ATTACK**

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. Now the following classification of SQLIAs in accordance to be presented.

**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 III as the tautology statement (1=1) has been added to the query statement so it is always true.

**Illegal/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. In this example attacker makes a type mismatch error by injecting the following text into the pin input field

1) Original URL: http://www.arch.polimi.itleventil?id nav=8864

2) SQL Injection: http://www.arch.polimLitieventil? id\_nav=8864'

3) Error message showed: SELECT name FROM Employee WHERE id =8864\'

From the message error we can find out name of table and fields: name; Employee; id. By the gained information attacker can organize more strict attacks.

**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 our examples 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= I UNION ALL SELECT credit Card Number, 1 FROM Credit Car Table We will have the following query: SELECT Name, Phone FROM Users WHERE Id= 1 UNION ALL SELECT creditCardNumber, 1 FROM Credit Car Table This 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.

**2. LITERATURE SURVEY**

Internet is a widespread information infrastructure. Unaware of the security and privacy, the internet is becoming a repository of information. Information and data is the most important business asset in today’s environment and achieving an appropriate level of Information Security. Applications are vulnerable to a variety of new security threats. One of the most threads to web application is SQL injection attack. According to Open Web Application Security Project (OWASP) top 10 threats for web application security in 2013, injection attacks stands first. For example, financial fraud, online banking, theft of private data, shopping and cyber terrorism. Web applications that are susceptible to SQL injection may allow an attacker to gain complete access to their essential databases. To implement security guidelines inside or outside the database the access of the sensitive databases should be monitored. Detection or prevention of SQL injection attacks is a topic of active research in the industry and academia. To achieve those purposes, automatic tools and security system have been implemented, but none of them are complete or accurate enough to guarantee an absolute level of security on web application.

SQL injection background SQL injections is one of the many web attack mechanism used by hackers to steal data from organizations. If it happens against the information systems of a hospital, the private information of the patients may be leaked out which could threaten their reputation or may be a case of defamation. These attacks not only make the attacker to breach the security and steal the entire content of the database but also, to make arbitrary changes to both the database schema and the contents. Most web applications today use a multi-tier design, usually with three tiers:

a) A presentation tier (front end). This is the topmost level of the application. This tier displays information related to such services as browsing merchandise, purchasing, and shopping cart contents. It communicates with other tiers by outputting results to the browser/client tier and all other tiers in the network.

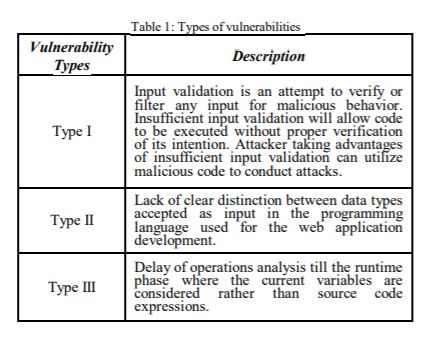
b) Application tier (Middle tier). This tier implements the software functionality by performing detailed processing, and

c) The data tier (Backend). This tier consists of database servers, keeps data structured and answers to request from the application tiers.

Three-tier is a client server architecture in which the user interface, functional process logic, data storage and access are developed and maintained as independent modules, most often on separate platforms. SQL injection is a type of attack which the attacker adds Structured Query Language code to input box of a web form to gain access or make changes to data. SQL injection vulnerability allows an attacker to flow commands directly to web applications underlying database and destroy functionality or confidentiality.

**2.1 Types of vulnerabilities**

In this section, we present the most common security vulnerabilities found in web programming languages.



**Table 2.1**

**2.2 Types of SQL injection attacks**

The SQL injection attacks can be performed using various techniques. Some of them are specified as follows:

**Tautologies:** The main goal of tautology-based attack is to inject code in conditional statements so that they are always evaluated as true. Using tautologies, the attacker wishes to either bypass user authentication or insert inject-able parameters or extract data from the database. A typical SQL tautology has the form, where the comparison expression uses one or more relational operators to compare operands and generate an always true condition. Bypassing authentication page and fetching data is the most common example of this kind of attack. In this type of injection, the attacker exploits an inject-able field contained in the WHERE clause of query. He transforms this conditional query into a tautology and hence causes all the rows in the database table targeted by the query to be returned. For example, SELECT \* FROM user WHERE id=’1’ or ‘1=1’-‘AND password=’1234’; “or 1=1” is the most commonly known tautology.

**Logically incorrect query attacks:** The main goal of the Illegal/Logically Incorrect Queries based SQL Attacks is to gather the information about the back end database of the Web Application. 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 error by purpose. In this example attacker makes a type mismatch error by injecting the following text into the input field:

1) Original <URL:http://www.toolsmarket-al.com/veglat/?id_nav=2234>

2) SQL Injection: <http://www.toolsmarket-al/veglat/?id_nav=2234>’

3) Error message showed: SELECT name FROM Employee WHERE id=2234\’.

From the message error we can find out name of table and fields: name; Employee; id. By the gained information attacker can organize more strict attacks.

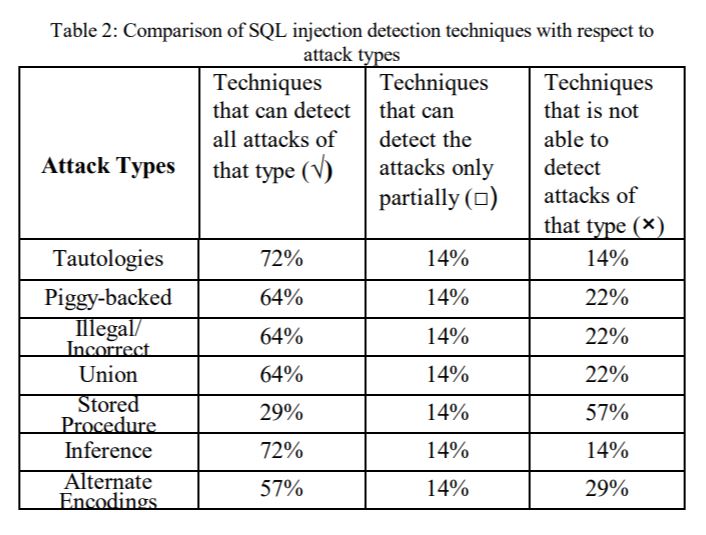
**Stored Procedures:** The main goal of the Stored Procedures SQL attack is to perform privilege escalation and try to execute the SQL procedures. SQL injection attacks of this type try to execute the SQL procedures. Stored procedure is a part of database that programmer could set an extra abstraction 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. is Authenticated @user Name varchar2, @pass varchar2, @pin int ASEXEC(“SELECT accounts FROM users WHERE login=’” +@user Name+ “’ and pass=’”+@password+”’and pin=”+@pin); GO For authorized/unauthorized user the stored procedure returns true/false. As an SQL injection attack, intruder input “’; SHUTDOWN; - -“for username or password. Then the stored procedure generates the following query: SELECT accounts FROM users WHERE login=’boni’ 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.

**Blind Injection:** Sometimes developers hide the error details which help attackers to compromise the database. In this situation attacker face to a generic page provided by developer, instead of an error message. So the SQLIA would be more difficult but not impossible. An attacker can still steal data by asking a series of True/False questions through SQL statements. Consider two possible injections into the login field: For example, SELECT accounts FROM users WHERE id= '1111' and 1 =0 -- AND pass = AND pin=0 SELECT accounts FROM users WHERE login= 'doe' and 1 = 1 -- AND pass = AND pin=0 If the application is secured, both queries would be unsuccessful, because of input validation. But if there is no input validation, the attacker can try the chance. First the attacker submits the first query and receives an error message because of "1=0 ". So the attacker does not understand the error is for input validation or for logical error in query. Then the attacker submits the second query which always true. If there is no login error message, then the attacker finds the login field vulnerable to injection.

**Alternate Encodings**: The main goal of the Alternate Encodings is to avoid being identified by secure defensive coding and automated prevention mechanisms. Hence it helps the attackers to evade detection. It is usually combined with other attack techniques. 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". 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 5f177 6e), " and the result query is: SELECT accounts FROM users WHERE login=" AND pin=0; exec (char (0x73687574646j776e)) This example use the char () function and ASCII hexadecimal encoding. The char () function takes hexadecimal encoding of character(s) and returns the actual character(s). The stream of numbers in the second part of the injection is the ASCII hexadecimal encoding of the attack string. This encoded string is translated into the shutdown command by database.

**2.3 Comparative Analyses**

In this section, the SQL injection detection and prevention techniques presented in section III would be compared. It is noticeable that this comparison is based on the articles not empirically experience. 4.1 Comparison of SQL Injection Detection Techniques With Respect to Attack Types Detection techniques are techniques that detect attacks mostly at runtime. Table 1 shows a chart of the schemes and their detection capabilities against various SQL injections attacks and summarize the results of this comparison. The symbol √ is used for techniques that can successfully detect all attacks of that type. The symbol × is used for techniques that is not able detect all attacks of that type. The symbol □ refers to techniques that detect the attack type only partially because of natural limitations of underlying approach.



**Table 2.2**

**3. Details of working and Architecture**

**3.1 Working**

Suppose we have an application based on student records. Any student can view only his or her own records by entering a unique and private student ID. Suppose we have a field like below:  
**Student id:**

And the student enters the following in the input field:

**12222345 or 1=1**.

So this basically **translates to:**

SELECT \* from STUDENT where

STUDENT-ID == 12222345 or 1 = 1

Now this **1=1** will return all records for which this holds true. So basically, all the student data is compromised. Now the malicious user can also delete the student records in a similar fashion.

Consider the following SQL query.

SELECT \* from USER where

USERNAME = “” and PASSWORD=””

Now the malicious can use the ‘=’ operator in a clever manner to retrieve private and secure user information. So instead of the above-mentioned query the following query when executed, retrieves protected data, not intended to be shown to users.

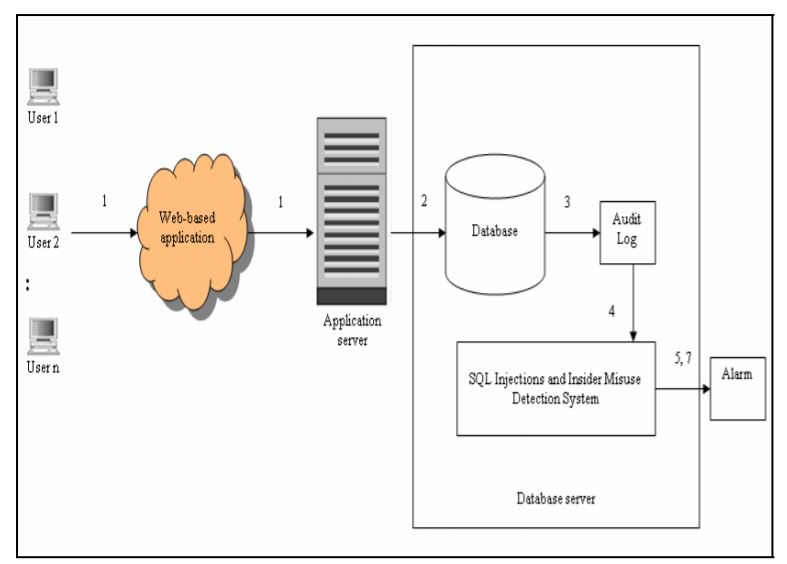
Select \* from User where

(Username = “” or 1=1) AND

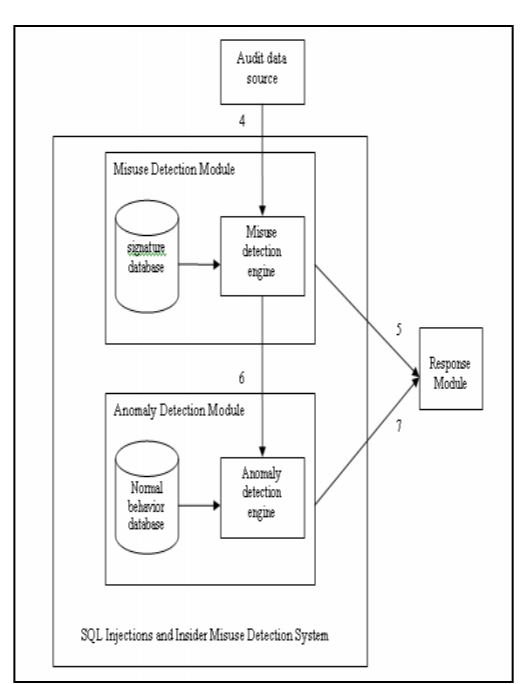
(Password=”” or 1=1).

Since **1=1** always holds true, user data is compromised.

Proposed system architecture Figure 3.1 shows the system architecture for the proposed SQL Injection and Insider Misuse detection system (SIIMDS), and Figure3.2 shows the components in the SIIMDS system. The operation of the SIIMDS system is described as follows:



**Figure 3.1**



**Figure 3.2**

1. The application user issues a service request to the application server through a web-based application. This transaction may or may not be legitimate.

2. The application server deploys the SQL query statements and issues them to the database server.

3. The user logs into the database and the database session is traced.

4. The SQL statements are received from the application and channeled to the misuse detection module. In the misuse detection module, the received SQL statements are matched with the set of SQL injection’s signatures.

5. If the SQL statement matches with the SQL injection signatures, an intrusion had occurred. The intrusion is then channeled to the Respond Module for the appropriate action(s) to be taken. 6. If no intrusion is detected by the Misuse Detection Module, the SQL statements are then channeled to the Anomaly Detection Module to check whether the SQL statements are different from the normal database access behavior.

7. If the SQL statements are different from the normal database access behavior, an internal misuse is concluded as has occurred and this misuse will be channeled to the Response Module for the appropriate action(s) to be taken.

8. The appropriate action(s) taken can be include the alerting of administrator by sounding the alarm.

**CONCLUSION**

This Report has presented a description on the threats in database security and the intrusions from both external and internal attacks against database systems. Most of the researches focused on detecting external attacks, but the most dangerous attacks actually come from insider’s misuse. This is because in many instances, the insiders do have authorized access to their database system but often misuse their rights. In these circumstances, all of their malicious activities will seem legitimate to the database thus difficult to detect. For that reason, this paper has proposed a SQL Injection and Insider Misuse Detection System (SIIMDS) to address both kinds of intrusions from internal and external threats. With this type of system, it is hoped to provide a higher level of security for database systems. A brief description on the SIIMDS system has also been provided. In addition, the attackers can access the database directly in an illegal way. Therefore, we are at the proposed a new approach that is completely based on the hash method of using the SQL queries in the web based environment, which is much secure and provide the prevention from the attackers SQL. But, our proposed strategy requires the alterations in the design of existing schema database and a new guideline for the database user before writing any new database. Through these guidelines, we expect the effective outcomes in SQL injections Preventions. In addition, many researchers can contribute to this SQL Injections Attacks Preventions Mechanisms without any extra effort to the database like alteration in schema design, adding new tables, adding new attributes etc.

**References**

[1] <https://www.geeksforgeeks.org/sql-injection-2/>

[2]<http://ijarcsse.com/Before_August_2017/docs/papers/July2012/Volume_2_issue_7/V2I700103.pdf>

[3]<https://www.researchgate.net/publication/272854124_SQL_Injection_Detection_and_Prevention_Techniques>

[4]<https://sci-hub.tw/10.1109/ITSIM.2008.4631942>

**ACKNOWLEDGEMENT**

I would like to place on record my deep sense of gratitude to Prof. **S.B.Kalyankar**, HOD-Dept. of Computer Science and Engineering, Deogiri Institute of Engineering and management Studies Aurangabad, for his generous guidance, help and useful suggestions.

I express my sincere gratitude to Prof. **S.G. Shah**, Dept. of Computer Science and Engineering, Deogiri Institute of Engineering and management Studies Aurangabad, for his stimulating guidance, continuous encouragement and supervision throughout the course of present work.

I am extremely thankful to **Ulhas Shiurkar**, Director, Deogiri Institute of Engineering and management Studies Aurangabad, for providing me infrastructural facilities to work in, without which this work would not have been possible.

**Signature of Student**

Rushikesh Shejwal Sign