A study on SQL Injection Attack and its Prevention Measures at Database Management Level

Nikita Gupta  
M.Tech. Student (4th Sem)  
Department of Computer Science & IT, University of Jammu, J & K, India  
Email: niki.gupta0612@gmail.com

Lalit Sen Sharma  
Professor  
Department of Computer Science & IT, University of Jammu, J & K, India  
Email: lalitsen.sharma@gmail.com

Abstract: In multi-tier design, the attacker takes the advantages of the design and maliciously inserts commands which may lead to security breaches such as unauthorized access to application resources, escalation of privileges, and modification of sensitive data. Also these malign commands do not get sensed fully by the firewalls and endpoint defenses. The attackers exploit these vulnerabilities to carry out attacks and compromise security. SQL Injection Attack (SQLIA) is a kind of attack which is performed by exploiting these vulnerabilities.

Keywords: Web Applications, SQL Injection, SQL queries, Malicious Code, Attacks

I. INTRODUCTION

Software development without web technologies seems almost improbable. Web Technology is also envisaged as the fastest application development framework. Most web applications today use a multi-tier design, usually three tiers. Multi-tier applications have become the norm for building enterprise software. Each tier is autonomous and developed independently under the defined rules in which one tier passes information and instruction to another tier in order to invoke the functionalities. Thousands of web applications are going to be developed and accessed by millions of users. Security of these websites is becoming an important concern to ensure the user’s authentication and privacy. This is the reason why the inclusion of effective security mechanisms on the web applications has been an increasing concern. In some cases, attackers can even use an SQL injection vulnerability to take control of system and corrupt the system that hosts the Web application. Web applications that are vulnerable to SQL Injection Attacks are widespread. According to a study conducted by Gartner Group, over 300 Internet Web sites are vulnerable to SQLIAs. In fact, SQLIAs have successfully targeted high-profile victims such as Travelocity, FTD.com, and Guess Inc. A hacking group known as ‘Focus’ on twitter claimed to have dumped 50,000 records on Archos.com. These dumps contain personal and corporate email addresses hosted on French and international domains, and include customer first names and surnames [1]. Also, recently huge list of websites which are vulnerable to SQL injection have been published. [2][3].

SQL Injection Attack (SQLIA) is a code injection attack which exploits the vulnerabilities existing in the web application by injecting malicious SQL code in the input fields of the web form in order to gain unauthorized access to the database. SQL injection vulnerabilities have been described as one of the most serious threats for Web applications [4][5]. Web applications that are vulnerable to SQL injection may allow an attacker to gain complete access to their underlying databases.

Types of SQL attack

SQL attack refers to the execution of malicious SQL code in the input field of the web form which provides unauthorized access to the database. SQL attack is performed in the following ways: [6]

1. Tautology: The general goal of a tautology-based attack is to inject code in one or more conditional statements so that they always evaluate to be true. The most common usages are to bypass authentication pages and extract data. In this type of injection, an attacker exploits an injectable field that is used in a query’s WHERE condition.

Example: - SELECT * FROM employee WHERE username = 'abc' and password = 'OR 1=1; Here the tautology statement (1=1) has been added to the query statement so that it is always evaluated to be true.

2. Logically Incorrect Queries: In this type, an attacker intentionally write incorrect query, so that the application server of the backend database returns error messages that contain enough information like the table name and name of the fields in the database which could be used further to prepare a more organized attack.

3. Union Query: By this method, attackers join injected query to the original SQL query by the word UNION and can then get data about other tables from the application.

4. Piggy-banked queries: In this approach, intruder appends additional query to the original SQL query to exploit database using the query delimiter, such as “;” to add extra query to the original query. Because of “;” character, database accepts both queries and executes them. Normally the first query is legitimate query, whereas following
queries could be illegitimate. The illegitimate query may drop table from the database.

5. **Blind SQL Injection**: This attack is often used when the web application is configured to show generic error messages, but has not mitigated the code that is vulnerable to SQL injection. The attacker then asks the database series of true or false questions and determines the answer based on the applications response. This makes exploiting the SQL Injection vulnerability more difficult, but not impossible.

6. **Stored Procedure**: A stored procedure is a database object just like table. It is a group of SQL statements that form a logical unit and perform a particular task. It is called using the name of the stored procedure and the parameter list. As a stored procedure is coded by the programmer, it is as vulnerable to SQL injections as the other Web Applications.

**II. AIMS AND OBJECTIVES**

The objective of this paper is to study the SQL Injection vulnerabilities in the web application to expose user’s authentication details (i.e. username, password etc). This paper also aims to study how this SQL attack can be mitigated at database level.

**III. RELATED WORK**

The main goals of SQL attacks are stealing the victim user’s sensitive information and invoking malicious acts on the user’s behalf. A survey has been done on detection and prevention techniques proposed by various researchers to mitigate SQL risks. SQL vulnerabilities can be detected by performing static and dynamic analysis on web application. Many researchers are carrying out their study in this domain. Some of them are listed as:

Supriya Gupta and Lalit Sen Sharma [7] introduced a technique to detect possible SQL Injection attacks by tracing the queries in which the input substrings modify the syntactic structure of the rest of the query. This approach has been implemented in a tool which takes an SQL query as input and detects if it is a command injection attack. Ruse et. al.’s Approach, [8], devised a technique that uses automatic test case generation to detect SQL Injection Vulnerabilities. This model deals with generating SQL queries automatically. Also, this approach identifies the relationship (dependency) between sub-queries.

William G. J. Halford et. al.’s Scheme [9] proposed an approach that works by combining static analysis and runtime monitoring of database queries. In its static part, it uses program analysis to automatically build a model of the legitimate queries that will be generated by the application. While in the dynamic part, the dynamically generated queries are monitored and check them for acceptability with the statically-generated model. A query that does not match with this model are hence prevented from executing on the database and reported.

Ali et. al.’s Scheme, [10] adopts the hash value approach to further improve the user authentication mechanism. They use the hash values of user name and password in order to test the framework. The user name and password hash values are created and calculated at runtime for the first time when the particular user account is created.

Kosuga et. al. [11] introduces a testing and debugging technique called Sania for detecting SQL injection vulnerabilities during the development phase. Sania constructs a parse tree for every intended SQL query and considers all the leaf nodes that take user inputs as vulnerable spots. It constructs a parse tree for each attack request and compares the tree with the one built for the intended SQL query. If the two trees are different, Sania determine there is SQL injection vulnerability otherwise not.

Ankit Anchlia and Sheela Jain (2010) [12] founded a completely unique approach to check the applications in a very comprehensive manner. This approach tests the system beneath real conditions with none artifacts, to avoid potential injection attacks.

Huang and colleagues [13] proposed a black-box technique for testing Web applications for SQL injection vulnerabilities. It uses 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. This method is better than other penetration-testing techniques by using machine learning approaches to guide its testing.

Thomas et. al.’s Scheme, [14] suggested an automated prepared statement generation algorithm to remove SQL Injection Vulnerabilities. They implement their research work using four open source projects namely: (i) Nettrust, (ii) ITrust, (iii) WebGoat, and (iv) Roller. Based on their experimental results, their prepared statement code was able to successfully replace 94% of the SQL in four open source projects.

Shin et. al.’s [15] approach suggests SQLUnitGen, a Static analysis- based tool that automate testing for identifying input manipulation vulnerabilities: The authors apply SQLUnitGen tool which is compared with FindBugs, a static analysis tool. The proposed mechanism is shown to be efficient as regard to the fact that false positive was completely absent in the experiments.

**IV. EXPERIMENTAL SET UP**

In this study, a website in *C# & ASP.NET* has been developed and hosted on the local host -3043. The experiments to exploit SQL vulnerabilities in the website have been performed to expose database to the attacker. The user performs different ways to successfully login into the web form. When the user successfully logs into the website, he can then have access to each and every record present inside the database. From there, he can perform different manipulations into the database. For example: he may insert, delete, or modify the records present in the table. These experiments have been performed on Microsoft SQL Server database. The figure1 shows the architecture for exploiting SQL vulnerabilities in the local host.
Fig. 1. The architecture for exploiting SQL vulnerabilities

Following is the screenshot of the web application form developed in Microsoft Visual Studio 2010:

![Web application form](image)

In these experiments, the malicious SQL code has been inserted into various username and password fields so that it gets executed in order to carry out SQL Injection attack.

The overall analysis of these experiments has been summarized in table 1.

<table>
<thead>
<tr>
<th>SQL attack vectors</th>
<th>MS SQL Database</th>
<th>Oracle Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Injection using Dynamic Strings: Query = “SELECT * FROM dbo.user1 WHERE username = ‘admin001’”;</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

V. MITIGATING SQL ATTACK

SQL attack is a type of code injection where user input is mistakenly interpreted as malicious program code rather than data. In order to prevent this type of code injection, secure input handling is needed. To mitigate SQL attack, the following methods have been used in our study:

**INPUT VALIDATION:** All input to the application whether it is from a user, a component, or any another program must be validated. This helps to ensure that the input is free from characters that cause SQL injection attacks. It escapes user input so that the browser interprets it only as data, not as code. It not only helps to protect against SQL injection attacks but also other attacks, such as XSS attacks and buffer overflows.

![Input validation form](image)
INPUT SANITIZATION: It modifies the input to ensure that it is valid. It is necessary to sanitize input and make potentially malicious input safe. Sanitization usually involves running any submitted data through a function (such as MySQL’s `mysql_real_escape_string()` function) to ensure that any dangerous characters (like ‘ ”’) are not passed to a SQL query in data.

PARAMETERIZED QUERIES: A parameterized query is a query in which placeholders are used for parameters and the parameter values are supplied at execution time.

REGULAR EXPRESSION MATCHING: In this case, the regular expressions for the possible malicious SQL keywords (to carry out SQL attack) have been defined. When the user enters the input, then it is matched with predefined all regular expressions to check whether it is valid or not. For this, an expression is required which is used to find SQL statements. Once we know that these statements are in the query, we can check for both authorized and unauthorized actions. This method employs block listing techniques.

Following is the code to perform input validation of user input:

```xml
<asp:RegularExpressionValidator id="nameRegex" runat="server"
    ControlToValidate="txtpwd"
    ValidationExpression="^[\s|\d|\-\|\|\$|\\|\£|\€|OR|UNION|SELECT|UPDATE|INSERT|DELETE|GRANT|REVOKE]|\{\}\]|\(|\)|\%|\|\+|\&|\| \| \| ] {1, 20} $" 
    ErrorMessage="Invalid Password">
```

VI. RESULT AND DISCUSSION

By performing these experiments on the local host, the various ways have been traced out by which malicious SQL gets executed. The malicious SQL query has been injected by the attacker to carry out SQL attack and the attack became successful. After using parameterized query instead of normal query, although it prevent some kind of attacks, but if we concatenate unsafe SQL query it gets executed and the attack gets performed. Also when attack is performed using stored procedure, there exist an issue in MS SQL server and Oracle databases. That is, if the dynamic SQL in not handled properly in stored procedure then it is vulnerable to attack in MS SQL Server whereas if the SQL query is formed with user input enclosed and concatenated to a string instead using bind vulnerable, then in that case the attack gets performed in Oracle database stored procedure. These kinds of attacks get performed because the user input is not enclosed inside the single quotes, but if we parameterize our queries then these kinds of attacks can be prevented.

Fig.2. Flow chart for Input Validation & Input Sanitization.

Fig.3. Flow chart for regular expressions matching
Malicious query (" OR 1=1 ;/**/--) posted by the attacker into web repository to carry out SQL attack.

Attacker successfully gets logged in as a result of SQL injection attack by executing malicious query in ASP.NET web application.

Now, after an attacker logs in he is now able to manage all the users present inside the database.

The underlying database of the web application is exposed to the attacker.

Deletion operation performed by the attacker to delete specific record.

Update been made by the attacker over the existing account to modify username, password, etc.

Attack is not able to log in into the application by executing malicious SQL injection.

It has been found in our experiments that the attack was performed successfully by injecting malicious SQL query in various ways. Then the preventing measures have been deployed to counter the SQL query execution by various ways to mitigate SQL attack risks. The preventing measures also evaluated for their merits and demerits which are as under:

Merits:
- Encoding, sanitization and regular expressions matching successfully mitigate SQL attack risks.
- Parameterized query provide a more generic query form which is then presented to the database.

Demerits:
- By adopting input validation and sanitization, users are not allowed to insert SQL keywords like OR, UNION, SELECT, DROP etc.
- Although, regular expressions matching allow valid input to be posted but the developers have to predefine the regular expressions for the malicious code (that can be misused by hackers to steal user’s authentication details). It causes overburden on the developer’s side.
- If the attacker inserts the malicious code that is not in the list of predefined regular expressions templates, then this code can be bypassed and it gets executed on the victim’s application.
- Stored procedures only directly prevent SQL injection if you call them in a parameterized way.

VII. CONCLUSION

Till now there have been a variety of defensive techniques to prevent SQL Injection. These techniques are implemented on the client-side or server-side to protect the
web users from SQL Injection attack. Still SQL is emerging as one of the top 10 web application vulnerabilities leading to security breach. A weak input validation on the web application causes the stealing of user’s confidential information. The hackers are becoming powerful day by day to develop new approaches to carry out SQL attack. SQL Injection is regarded as the top most vulnerability in the web applications, demands an efficient approach on the server side as well as client side to protect the users of the web application. Also it demands an efficient approach on the existing database security tools so that these attacks can be prevented.

REFERENCES