Visvesvaraya Technological University, Belagavi – 590018

SEMINAR
REPORT ON

SQL Injection

Submitted in partial fulfillment of the requirements for the degree

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE & ENGINEERING

Submitted by

Pranith Rao

4SO18CS088

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
(Affiliated to VTU-Belagavi, Recognized by AICTE, NBA Accredited)

ST JOSEPH ENGINEERING COLLEGE
Vamanjoor, Mangaluru-575028, Karnataka
2021-2022

ABSTRACT

The abbreviation SQL stands for Structured Query Language. SQL is a programming language for interacting with databases and changing their contents. Databases often incorporate data definition and data manipulation languages to achieve goals. SQL injection, on the other hand, is a method that enables hackers to use a database server to conduct malicious SQL queries. A web-based application may be used to get access to databases containing sensitive data. ALWAYS TRUE will be added to a SQL query as a constraint.

We use database-driven online applications for a growing number of tasks, including banking and shopping. We disclose our personal information to these online programs and their underlying databases when we engage in these activities. Web application hacks are popular because they provide an attacker complete access to the program's underlying database. Database attacks jeopardize database data and structure, as well as the hosting systems and applications that depend on them, necessitating immediate attention and the implementation of appropriate protective measures by application developers. As a consequence, businesses have made safeguarding online applications (such as websites) and web services against SQL Injection Attacks a top priority.

It's not only easy to get started, but also almost impossible to avoid with a little forethought and common sense. In this post, we'll look at some of the strategies used by SQL injection attackers, as well as how to defend against them.

CONTENTS

1. INTRODUCTION

The World Wide Web has expanded considerably in recent years. Web applications have shown to be useful, efficient, and trustworthy alternatives for enterprises, individuals, and governments to communicate and do business in the twenty-first century. Web application security, on the other hand, has become more important during the last decade. Protecting Web-based businesses that handle sensitive financial and medical data from hacker attacks is becoming more vital. The Application Defence Centre examined the security of over 250 Web applications, including e-commerce, online banking, corporate collaboration, and supply chain management sites, and discovered that at least 92 percent are vulnerable to assault. A key source of risk in online applications is allowing unchecked input to take control of the program, which an attacker may use for unwanted purposes. SQL Injection is the most often utilized approach.

SQL Injection is a type of database hacking. An attacker can get access to resources or alter data by inserting SQL code into a Web form input box. It's a type of data-driven application attack that includes injecting malicious SQL queries into a field and running them (for example, to dump the database contents to the attacker). It must take advantage of a software flaw, such as when user input is not strongly typed and executed unexpectedly, or when string literal escape characters encoded in SQL statements are wrongly validated. Although it's most usually associated with internet attacks, it may be used against any SQL database.

SQL injection attacks allow attackers to impersonate users, tamper with existing data, cause repudiation issues such as voiding transactions or changing balances, allow full disclosure of all data on the system, destroy or otherwise make data unavailable, and gain access to the database server's administrators. Existing queries are regularly changed to get the same outcomes. Transact SQL may easily be tampered with by simply inserting a single character in the wrong place, causing the query to react maliciously. The most often used characters in SQL are the backtick ('), double dash (--), and semicolon, each with its meaning.

A user of our form types SQL code into it and surrounds it in special characters, causing the data supplied to be utilized for purposes other than what we intended, such as corrupting or destroying our database. When the attacker fills out the form, the information is utilized to create a dynamic SQL query to obtain the information from the database. An SQL Insertion attack is the term for this type of malicious code injection. The attacker can then extract, edit, add, or remove data from the database. In rare cases, the attacker may be able to get past the database server and into the operating system itself.

Insufficient validation of user input causes SQL Injection Attacks (SQLIAs). When the user's input is utilized to directly generate a database query, the vulnerability exists. The attacker can inject malicious input that is viewed as new commands by the database if the input is not properly encoded and checked by the application. The attacker can submit a variety of SQL instructions to the database, depending on the severity of the vulnerability. SQLIA can affect a wide range of interactive database-driven systems, including online apps that employ user input to query their underlying databases. In fact, informal examinations of database-driven online applications have revealed that about 97 percent are susceptible to SQLIA. SQLIAs, like other security flaws, may be avoided by employing protective code. However, in fact, implementing and enforcing this strategy is extremely tough. As developers implement new safeguards, attackers continue to adapt and devise new ways to get around them. It's challenging to keep engineers up to date on the latest and finest defensive coding approaches since the state of the art in defensive coding is always changing. Furthermore, applying defensive coding methods to retrospectively update insecure historical programs is difficult, time-consuming, and error-prone. These issues highlight the need for a more automated and universal SQL injection solution [1].

Other types of attacks than SQL Injection include:

· Shell injection.

· Scripting language injection.

· File inclusion.

· XML injection.

· SQL Injection

· XPath injection.

· LDAP injection.

· SMTP injection.

2. SQL INJECTION ATTACKS

An SQL injection vulnerability enables an attacker to submit instructions directly to a web application's underlying database, compromising the program's intended functionality. When an attacker discovers a SQLIA vulnerability, the susceptible application serves as a conduit for the attacker to execute instructions on the database and perhaps the host system.

SQLIAs are a kind of code injection attack that takes advantage of a lack of user input validation. When developers blend hard-coded strings with user input to generate dynamic searches, the vulnerabilities appear. If user input is not sufficiently vetted, attackers may shape it such that when it is included in the final query string, elements of the input are processed as SQL keywords or operators by the database.

The attacker's goal, vulnerabilities, and assertions are used to classify the attack. We may categorize the attacker's aims based on their purpose as follows:

1. Data extraction - The attacker will seize sensitive data. Assume that if the admin database is compromised, the whole database becomes susceptible.
2. Data access - They attempt to circumvent the privileges and get access to the full database in order to modify the data.
3. Fingerprint the database- The attacker will determine the database version and type in this attack. This approach enables them to test various types of queries in various applications.
4. Injectable parameters are discovered - susceptible parameters will be discovered for attack utilizing some of the automated techniques.
5. Authentication Bypass — To get access to the database, application authentication methods will be circumvented.
6. Database schema identification - To successfully collect information, the database table name, data type of each field, column name, and so on will be obtained.
7. To conduct denial of service - This includes dropping tables and shutting down the system. The attacker attempts to get access to the system in order to execute a specified command inside the database [2].

2.1 Illustration of SQL Injection

The insertion or "injection" of a SQL query via the input data from the client to the program is referred to as a SQL injection attack. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), perform database administration operations (such as shutting down the DBMS), recover the content of a given file on the DBMS file system, and in some cases, issue commands to the operating system. SQL injection attacks include injecting SQL instructions into data-plane input to cause predetermined SQL commands to be executed.

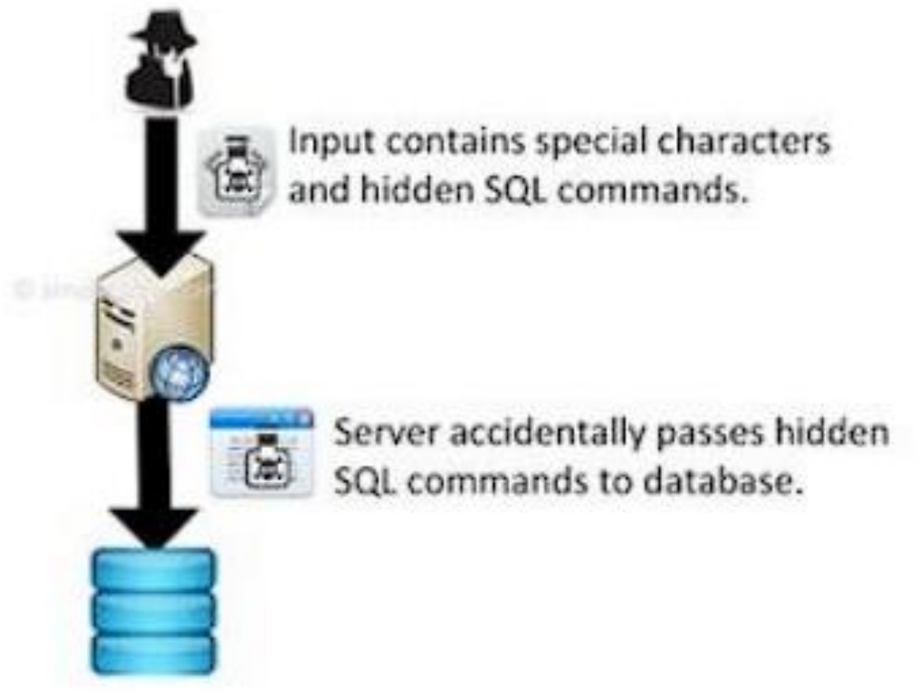

Input contains special characters
and hidden SQL commands.
Server accidentally passes hidden
SQL commands to database.

Fig 2.1 Illustration of SQL Injection

SQL injection refers to a kind of code injection attack in which data given by a user is incorporated in a SQL query in such a manner that a portion of the user's input is interpreted as SQL code, according to (William, Jeremy, & Alessandro, 2002). An attacker can use this to send SQL instructions directly to the database. These attacks pose a major danger to any Web-based application that accepts user input and converts it into SQL queries to a database. Most Web-based programs, whether used on the Internet or within an enterprise system, operate in this manner, making them vulnerable to SQL injection [3].

2.2 Vulnerability

SQL Injection attacks can take two forms:

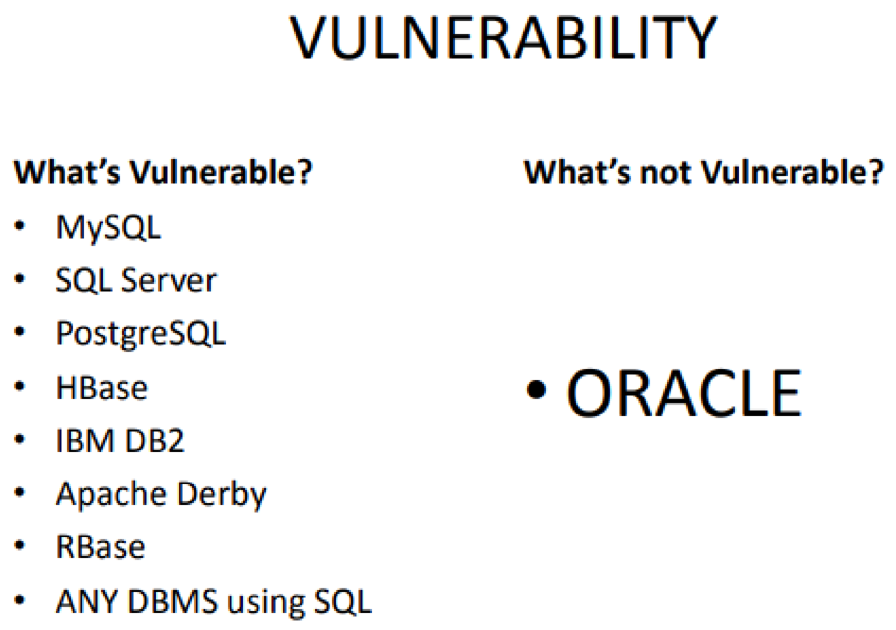
1. Injecting the Form
2. Injection of URLs

2.2.1 Who’s vulnerable?

Only one factor makes a web application vulnerable to SQL injection: end-user string input is not adequately verified before being given to a dynamic SQL query. The SQL query is normally supplied the string input directly. The user input, on the other hand, might be saved in the database and then provided to a dynamic SQL query. Because many online applications are stateless, it is customary to write data to the database between web pages. This form of indirect assault is far more complicated and needs a thorough understanding of the program.

2.2.2 Who’s not vulnerable?

SQL statements that employ bind variables are typically resistant to SQL Injection attacks since the Oracle database only uses the bind variable's value and does not comprehend the contents of the variable. Bind variables are supported by PL/SQL and JDBC. For security and performance reasons, binding variables should be utilized often.



VULNERABILITY

What’s Vulnerable?

- MySQL
- SQL Server
- PostgreSQL
- HBase
- IBM DB2
- Apache Derby
- RBase
- ANY DBMS using SQL

What’s not Vulnerable?

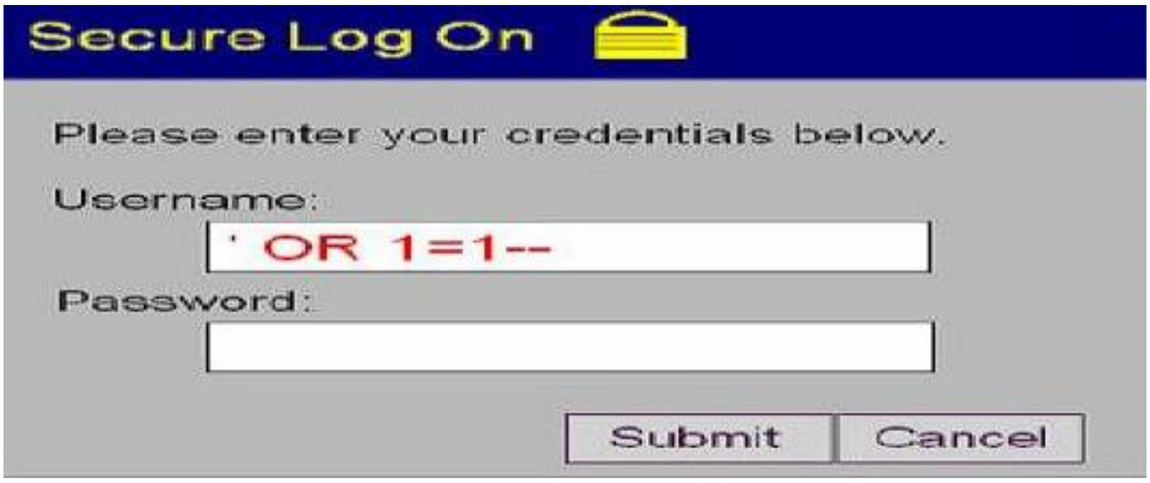
- ORACLE

Fig 2.1 Vulnerability

3. WORKING OF SQL INJECTION

The fundamentals of SQL injection are straightforward, and these assaults are simple to conduct and learn. An attacker must locate a parameter that the web application sends to the database to exploit a SQL injection bug. The attacker can fool the web application into sending a malicious query to the database by carefully encoding malicious SQL commands within the body of the argument.

For example, consider the login form which accepts the username and password from the user.



Secure Log On

Please enter your credentials below.

Username: ' OR 1=1--

Password:

Fig 3.1 Injecting SQL

The values entered in the "Username" and "Password" fields are immediately utilized to construct the SQL Query:

```
SELECT * FROM people WHERE name = ‘name’ AND password = ‘password’
```

Now, Suppose the user supplied the Username = ’Pranith’ and Password = ‘Rao’
The query will become:

```
SELECT * FROM people WHERE name = ‘Pranith’ AND password = ‘Rao’
```

This will work perfectly fine. However, if the user provides a badly constructed string of code, the attacker will be able to bypass authentication and get access to the database's information i.e. if the user inputs username=' OR 1=1--, the query will be given as:

```
SELECT * FROM people WHERE name = ‘’ OR 1=1--’ AND password = ‘ ’;
```

It Works as follows:

- **‘**: Closes the user input field.
- **OR**: Continues the SQL query so that the process should equal to what comes before OR what comes after.
- **1=1**: A statement which is always true.
- **--**: Comments outs the rest of the lines so that it won’t be processed.

The WHERE clause uses the information we're entering. Because the application is just concerned with creating a string rather than thinking about the query, our usage of OR has transformed a single-component WHERE clause into a two-component one, with the 1=1 clause guaranteed to be true regardless of the first clause. The question means:

“Select everything from the table people if the name equals “nothing” Or 1=1. Ignore anything that follows on this line”.

Because 1 will always equal 1, the server believes it has received a truthful statement and grants an attacker greater access than they should. The code that references to the password input box is never executed by the server, thus it isn't applicable [4].

3.1 More examples

3.1.1 Obtaining the name of a column from a database error message.

Consider a login form supplied with following arguments:

```
Username: ' having 1=1 ---
Password: [can be anything]
```

The SQL query causes ASP to spew the following error to the browser when the user clicks the submit button to begin the login process:

```
“Column 'people.personName' is invalid in the select list because it is not contained in an
aggregate function and there is no GROUP BY clause. /login.asp, line 16”
```

This error message now informs unauthorized users of the name of one database field against which the application is attempting to validate login credentials: people.personName. Using the name of this field, an attacker may now login using the following credentials using SQL Server's LIKE clause:

```
Username: ' OR users.userName LIKE 'p%' –
Password: [can be anything]
```

This performs an injected SQL query against our people table:

```
SELECT personName FROM people WHERE personName='' OR people.personName
LIKE 'p%' --' and pass=''
```

The query grabs the personName field of the first row whose personName field starts with ‘p’.

3.1.2 Dropping a table from database.

A semicolon (;) is used to delimit a query in certain databases. Multiple queries can be sent as a batch and run consecutively using a semi-colon. This might be used by the attacker to inject data into the database. For instance,

```
Username: ' OR 1=1; DROP TABLE users; --
Password: [can be anything]
```

The query would then be split into two pieces. To begin, it would choose the userName field for all users database entries. Second, it would erase the users table, resulting in the following error when we attempted to login again:

```
Invalid object name 'users'. /login.asp, line 16
```

4. TYPES OF ATTACKS

SQL injection attacks may have a wide range of consequences, from obtaining sensitive data to modifying database information, from executing system-level instructions to denying service to the application. The impact is also determined by the database on the target system, as well as the roles and rights assigned to the SQL query.

4.1 On Basis of Execution Nature of Injection:

1. First Order attacks
2. Second Order attacks

4.1.1 First Order attacks

First-order attacks are ones in which the attacker obtains the intended result right away, either by a direct response from the program they're engaging with or through another means like e-mail. Consider a form that requests the user's email address. The query will run correctly if the user provides the right email address without any additional codes. However, if the user enters a "LIKE" clause with the email address, the database will immediately provide the matching criteria to the user.

```
SELECT email, password, login_id, full_name FROM members WHERE email=’ x’ OR
full_name LIKE ‘%Rao%’;
```

The database will produce information for any user whose name begins with the letter "Rao" These assaults are known as first order attacks since the attacker receives the outcome instantly.

4.1.2 Second Order Attacks

Second-order SQL injection attacks stall execution until a secondary query is run. This means that a malicious user can inject a query fragment into a query that isn't necessarily vulnerable to SQL injection, and then have the injected SQL run in a second query that is. This sort of attack is common since once data is in the database, it is generally assumed to be clean and rarely double-checked. The data is, nevertheless, routinely utilized in inquiries, where it can still be harmful.

Consider a program that allows users to save their favorite search criteria. The program escapes all apostrophes when the user sets the search criteria, preventing the first-order attack when the data for the favorite is placed into the database. When the user arrives to search, the data is extracted from the database and utilized to create a second query, which then executes the search. The second inquiry is the one that has been attacked.

For example, if the user types the following as the search criteria:

```
'; DELETE employees;--
```

The application takes this input and escapes out apostrophe so that the final SQL statement might look like this:

```
INSERT INTO company (UserID, Name, Location) VALUES(369, 'SQL Attack',
''';DELETE employees;--')
```

which is successfully entered into the database. When the user picks their preferred search, the data is received by the program, which then creates and runs a new SQL statement.
When completely extended, the second database query now looks like this:

```
SELECT * FROM company WHERE Name = ''; DELETE Orders;--
```

The predicted query will yield no results, but the corporation has now lost all of its orders.
These types of attacks are known as second order attacks[5].

4.2 On Basis of Goal or Purpose

There are several attack tactics that are carried out simultaneously or sequentially depending on the attacker's aim. The attacker should attach a syntactically valid instruction to the original SQL query for a successful SQLIA.

The different types of attack are:

4.2.1 Tautologies

In this form of attack, SQL tokens are injected into the conditional query statement, causing it to always evaluate true. By attacking insecure input fields that employ the WHERE clause, this sort of attack was able to get around authentication controls and gain access to data.

```
"SELECT * FROM employee WHERE userid = '117' and password ='aaa' OR '1 '='1
```

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

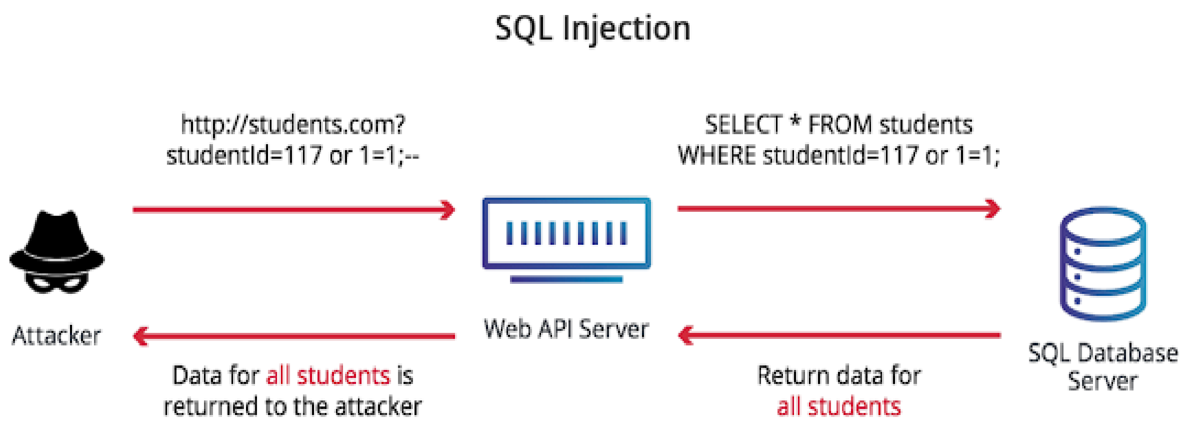


Fig 4.2 Tautological SQL Attack

4.2.2 Illegal/Logically Incorrect Queries

When a query is refused, the database returns an error message with important debugging information. These error messages aid attackers in locating susceptible parameters in the program and, as a result, the database. In reality, the attacker deliberately injects trash data or SQL tokens into the query to cause syntax mistakes, type mismatches, and logical errors.

In this case, the attacker injects the following text into the pin input box, causing a type mismatch:

1. Original URL: http://www.pranithrao.com?id nav=369
2. SQL Injection: http:// www.pranithrao.com? id_nav=369'
3. Error message showed: SELECT name FROM Employee WHERE id =369\'

We can deduce the name of the table and attributes from the message error: name; Employee; id. With this knowledge, the attacker may plan more precise strikes.

4.2.3 Union Query

By employing the phrase UNION to connect an injected query to a safe query, attackers can access information about other tables from the application.

4.2.4 Piggy-backed Queries

Intruders utilize a query delimiter, such as ";", to attach an extra query to the first query in a piggy-backed query attack. The database will receive and execute multiple queries if the attack is successful. The first query is almost always real, but following requests may be fake. As a result, an attacker can perform any SQL command on the database.

4.2.5 Stored Procedure

A stored procedure is a database component that allows a programmer to add a layer of abstraction to the database. Because stored procedures may be written by programmers, this section is as injectable as web application forms. Depending on the stored procedure in the database, there are numerous attack approaches [6].

5. CATEGORIES OF SQLIAs

There are four main categories of SQL Injection attacks against databases. They are:

1. SQL Manipulation
2. Code Injection
3. Function Call Injection
4. Buffer Overflows

5.1 SQL Manipulation

SQL manipulation is the most prevalent sort of SQL Injection attack. By adding components to the WHERE clause or extending the SQL statement with set operators like UNION, INTERSECT, or MINUS, the attacker attempts to change the current SQL statement. Other variants are conceivable, but these are the most prominent ones.

During login authentication, the traditional SQL manipulation occurs. User authentication may be checked in a simple web application by running the following query and seeing whether any rows are returned –

```
SELECT * FROM users WHERE username = 'pranith' And PASSWORD = 'mypassword'
```

The attacker attempts to manipulate the SQL statement to execute as:

```
SELECT * FROM users WHERE username = 'pranith' AND Password = 'mypassword' or
'a' = 'a'—
```

The WHERE clause is true for every entry due to operator precedence, and the attacker has acquired access to the program.

SQL injection attacks typically employ the set operator UNION. The purpose is to change a SQL query such that it returns rows from a different table. To return a list of available items, a web form can run the following query:

```
SELECT product_name FROM all_products WHERE product_name like '%Table%'
```

The attacker attempts to manipulate the SQL statement to execute as:

```
SELECT product_name FROM all_products WHERE product_name like '%Table'
UNION SELECT username FROM dba_users WHERE username like '%';
```

The list returned to the web form will include all the selected products, but also all the database users in the application.

5.2 Code Injection

Injecting extra SQL statements or instructions into an existing SQL statement is called a code injection attack. This attack is common against Microsoft SQL Server applications; however, it seldom works with Oracle databases. SQL injection attacks often target the EXECUTE command in SQL Server; there is no equivalent statement in Oracle.

Oracle does not allow multiple SQL statements per database request in PL/SQL or Java.
As a result, the following typical injection attack using a PL/SQL or Java program will not work against an Oracle database. This statement will produce the following error:

```
SELECT * FROM users WHERE username = 'pranith' AND Password = 'mypassword';
DELETE FROM users WHERE username = 'admin';
```

However, certain computer languages or APIs may allow for the execution of numerous SQL statements.

Anonymous PL/SQL blocks may be dynamically executed in PL/SQL and Java programs, making them subject to code injection. An example of a PL/SQL block in a web application is shown below —

```
BEGIN ENCRYPT_PASSWORD('pranith', 'mypassword'); END;
```

The above example PL/SQL block executes an application stored procedure that encrypts and saves the user’s password. An attacker will attempt to manipulate the PL/SQL block to execute as –

```
BEGIN ENCRYPT_PASSWORD('pranith', 'mypassword'); DELETE FROM users
WHERE upper(username) = upper('admin'); END;
```

5.3 Function Call Injection

The introduction of Oracle database functions or custom functions into a susceptible SQL query is known as function call injection. These function calls may be used to interact with the operating system or alter database data.

The Oracle database enables you to use functions or functions from packages in SQL statements. Oracle provides about 1,000 methods by default in over 175 common database packages, albeit just a handful of these procedures may be relevant in a SQL injection attack. Some of these functions may be used to undertake network operations. A SQL statement may run any custom function or function included in a custom package.

Unless the function is tagged as "PRAGMA TRANSACTION," functions performed as part of a SQL SELECT statement cannot make any modifications to the database. None of Oracle's basic functions is run as standalone transactions. Functions in INSERT, UPDATE and DELETE statements can alter database data. An attacker may transport information from the database to a distant computer or launch additional attacks from the database server using ordinary Oracle methods. Many Oracle native apps use database packages that may be abused by an attacker. These bespoke packages might contain functions for changing passwords or performing other sensitive app operations.

The problem with function call injection is that it exposes any dynamically created SQL query. Even the most basic SQL statements may be used to a great advantage. Even the most basic SQL queries might be susceptible, as seen in the following example. To conduct typical activities, application developers may occasionally employ database functions rather than native code (e.g., Java). Because the TRANSLATE database function does not exist in Java, the programmer chose to utilize a SQL query.

```
SELECT TRANSLATE('user input',
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ', '0123456789') FROM dual;
```

This SQL statement is not subject to other sorts of injection attacks, but a function injection attack may readily change it. The attacker tries to make the SQL query run as follows:

```
SELECT TRANSLATE('' || UTL_HTTP.REQUEST('http://192.168.1.1/') || '',
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ', '0123456789') FROM dual;
```

The new SQL query will ask a web server for a page. To extract important information from the database server and transfer it to the web server in the URL, the attacker might change the string and URL to add other functions. Because the Oracle database server is most likely protected by a firewall, it might be exploited to attack other internal network servers.

It is also possible to run custom functions and functions from custom packages. A custom application using the function ADDUSER in the custom package MYAPPADMIN is an example. The function was labelled "PRAGMA TRANSACTION" by the developer so that it may be used in any unusual scenarios that the program could face. It may write to the database even in a SELECT query since it is designated "PRAGMA TRANSACTION."

```
SELECT TRANSLATE('' || myappadmin.adduser('admin', 'newpass') || '',
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ', '0123456789') FROM dual;
```

Executing the above SQL statement, the attacker is able to create new application users

5.4 Buffer Overflows

A variety of typical Oracle database methods are vulnerable to buffer overflows, which may be exploited in an unpatched database via a SQL injection attack.

Known buffer overflows exist in the standard database functions like:

- tz_offset
- to_timestamp_tz
- bfilename, from_tz
- numtoyminterval
- numtodsinterval

Employing the function injection techniques discussed before, a buffer overflow attack using tz_offset, to_timestamp_tz, bfilename, from_tz, numtoyminterval, or numtodsinterval is carried out. Remote access to the operating system may be gained by leveraging the buffer overflow with a SQL injection attack.

Furthermore, most applications and web servers do not gracefully manage a database connection being lost due to a buffer overflow. The web process will often linger until the client connection is cancelled, making this a very effective denial of service attack [7].

6. DETECTION OF VULNERABILITY

To detect whether your application is vulnerable to SQL injection attacks, follow the given steps:

Step 1: Go to the website and open it in a browser.

Step 2: Move your mouse over the website's links while keeping an eye on the bottom status bar. The URLs that the links refer to will be visible.

Look for a URL that includes parameters

(For example, http://www.pranith.com/myid.asp?id=69).

When the file extensions are ".asp" or ".cfm," most SQL injection issues occur. Look for these files explicitly when testing a site for SQL injection vulnerabilities. If you don't see any URLs in the status bar, just click on links and keep an eye on the address bar until you discover one with parameters.

Step 3: Once you've located a URL containing parameters, click the link to navigate to that website. The URL from the status bar should now appear in the address bar, along with parameter values.

Step 4: This is where the real hacker protection testing takes place. For SQL injection testing, there are two ways. Make careful to test each parameter value with both methods one at a time.

- **Method 1:** Click your mouse in the address bar and highlight a parameter value (for example, the word value in "name=value"), then replace it with a single quotation ('). The syntax should now be "name=' ".
- **Method 2:** Click your mouse in the address bar and type a single quotation (') in the centre of the value. "name=val'ue" should now appear.

Step 5: To transmit your request to the Web Server, click the 'GO' button.

Step 6: Analyze the response from the Web server for any error messages. Most database error messages will look similar like:

Example Error 1:

```
'80040e14'Unclosed quotation mark before the character string '51 ORDER BY
some_name'./some_directory/some_file.asp, line 5
```

Example Error 2:

```
ODBC Error Code = S1000 (General error [Oracle][ODBC][Ora]ORA-00933: SQL
command not properly ended.
```

Step 7: Occasionally, the error message is not visible and is concealed in the page's source code. To find it, look at the page's HTML source code and look for the problem. Click the 'View' menu in Internet Explorer and pick the 'Source' option. This will launch Notepad with the page's HTML source code. Select 'Find' from the 'Edit' menu in Notepad. A dialogue window will pop up asking you to 'Find What.' In the text field, type 'Microsoft OLEDB' (ODBC) and then click 'Find Next.'

If either Step 6 and 7 are successful, then the website is susceptible to SQL injection [8].

7. PREVENTING SQL INJECTION ATTACKS

Unfiltered user input or certain overprivileged database logins are the most common causes of SQL injection attacks. The following are some frequent flaws that render your application vulnerable to SQL injection attacks:

- Weak input validation.
- Dynamic construction of SQL statements without use of proper type safe parameters.
- Use of over privileged database logins.

Simple programming improvements may quickly resist SQL injection attempts, but developers must be disciplined enough to implement the following approaches to every web-accessible process and function. Every dynamic SQL statement has to be safeguarded. A single unprotected SQL query may result in the application, data, or database server being compromised.
The following are some ways for preventing SQL injection attacks:

7.1 Use Bind Variables

Bind variables are the most effective defense against SQL injection attempts. Using bind variables will also increase the performance of your application. All SQL statements should employ bind variables, according to application code standards. Concatenating texts and passed arguments should not be used to build SQL statements. Regardless of when or where the SQL query is performed, binding variables should be utilized. This s Oracle's internal coding standard, and it should be your company's as well. An application might be exploited by putting an attack string in the database, which would then be performed via a dynamic SQL query.

7.2 Validate the input

Every string parameter should be checked for validity. input validation should be done first at the client level. The data from the client form must be checked for input validity at the server level before submitting the query to the database server for execution. If the data fails to pass the validation process, it should be rejected and the user should be notified. Many web apps include hidden fields and other techniques that must be scrutinized. f a bind variable is not used, special database characters must be removed or escaped. A single quote is the only character at issue with Oracle databases. The simplest way is to escape all single quotes; Oracle sees multiple single quotes as a single literal quote. It is not suggested to use bound variables and escape single quotes for the same content. In the database, a bind variable keeps the precise input text, and escaping any single quote leads the database to store double-quotes.

7.3 Function Security

SQL injection attacks may take advantage of both standard and bespoke database functions. In an assault, several of these functionalities may be employed successfully. Oracle comes with hundreds of standard functions, all of which are set to PUBLIC by default. Additional features, such as changing passwords or creating exploitable users, might be included in the program. All functionalities that aren't strictly required by the program should be limited.

7.4 Limit the Open-Ended input

Whenever feasible, use pick boxes instead of Text boxes to restrict open-ended input. All inputs must be validated on the client-side by the application. Only the choice in the pick box should be selected for validation, and any other option should be refused.

7.5 Verify the Type of Data

Before entering it into a SQL query, check the data type using SNUMERIC or an analogous function. Replace single quotes in string data with two single quotes using the replace function or an equivalent.

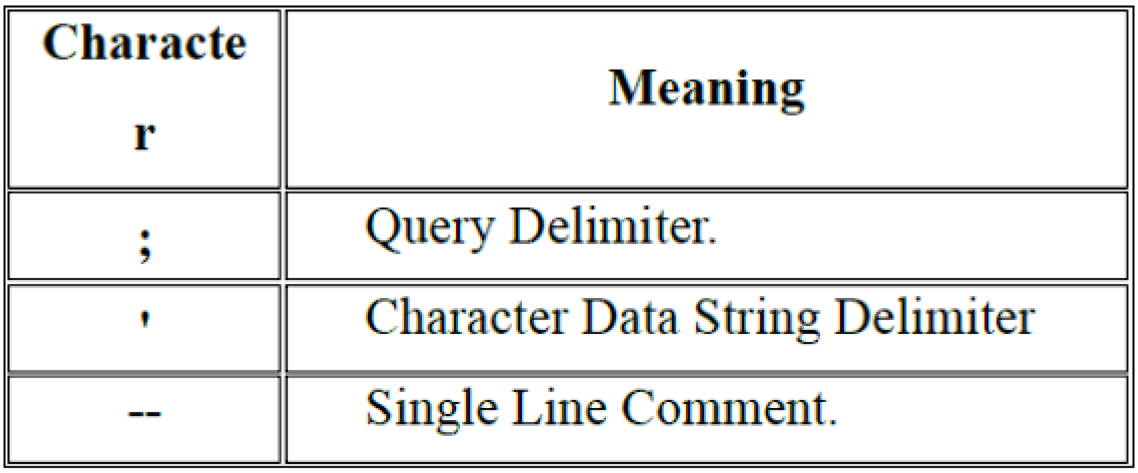
```
Good string = replace(input string,'',");
```

7.6 Use Stored Procedures

Avoid direct access to the table by using stored procedures. Unused stored procedures, such as master_xp_cmdshell, xp_sendmail, xp_startmail, and sp_makewebtask, may be eliminated.

1. Never construct a dynamic SQL statement directly from user input, and never concatenate user input with unvalidated SQL statements.
2. Remove slash, backslash, and extended characters like NULL, carry return, and new line from all strings from user input and URL parameters.
3. The privileges of the application's user account for executing SQL statements on the database must be established.
4. User input should be restricted in length.

5. Reject input that includes the following potentially harmful characters wherever feasible [9].



Characte r	Meaning
;	Query Delimiter.
'	Character Data String Delimiter
--	Single Line Comment.

Fig 7.1 Dangerous Characters

8. SQL INJECTION PREVENTION TOOLS

8.1 SQLmap

SQLmap is a Python-based SQL injection scanner. The goal of this program is to find SQL injection vulnerabilities and exploit them in web applications. SQLmap will first detect the loop in your site as a whole and then use a variety of options to perform extensive back-end database management, such as enumerating users, duplicating entire or specific DBMS, retrieving DBMS session user and database, reading specific files from the file system, and so on.

- SQLmap is a little quicker than the Acunetix web scanner, but it's still slower than SQLiX, and it injects fewer URLs into the database than SQLiX. This utility also lacks a graphical user interface.
- SQLmap is an open-source programme that may be used to identify and exploit database flaws as well as insert malicious code into them.
- It is a penetration testing application that uses a terminal user interface to automate the process of discovering and exploiting SQL injection problems.
- SQLmap supports MySQL, Oracle, PostgreSQL, Microsoft SQL Server, Microsoft Access, BM DB2, SQLite, Firebird, and SAP MaxDB, among others.

8.2 SQLninja

SQLninja is a SQL injection tool for online sites that utilize SQL as their database server. This tool may initially fail to locate the injection site. If it is detected, however, it is simple to automate the exploitation process and retrieve data from the database server.

- This tool may deactivate data execution prevention by adding remote shots to the database server OS registry. The tool's ultimate goal is to provide the attacker remote access to a SQL database server.
- It may also be used in conjunction with Metasploit to provide graphical access to a remote database. It also supports TCP and UDP bind shells, both direct and reverses.
- This program is not compatible with Windows. It is currently only available for Linux, FreeBSD, Mac OS X, and OS.

8.3 Safe3 SQL Injector

Another strong yet simple to use SQL injection tool is Safe3 SQL Injector. It, like other SQL injection tools, automates the SQL injection process and aids attackers in exploiting the SQL injection vulnerability to get access to a remote SQL server. It includes a strong AI system that can quickly identify the database server, injection kind, and best technique to exploit the flaw.

- Both HTTP and HTTPS websites are supported. SQL injection may be done through GET, POST, or cookies. To conduct a SQL injection attack, it also supports authentication (Basic, Digest, and NTLM HTTP authentications).
- MySQL, Oracle, PostgreSQL, Microsoft SQL Server, Microsoft Access, SQLite, Firebird, Sybase, and SAP MaxDB database management systems are all supported by the program [10].

9. CONCLUSION

Unexpected security flaws in web apps have been released to the public. The short development period of this application s mostly to blame for these security flaws. Although security program research is relatively new, effective solutions are in great demand due to the need of producing safe and less susceptible systems.

The majority of web applications employ an intermediary layer to accept user requests and obtain sensitive data from databases. To create intermediary layers, they usually employ scripting languages. SQL injection methods are often used by hackers to compromise database security. By altering SQL queries, an attacker attempts to fool the intermediate layer technology. Perhaps the attacker will alter the programmer's activity to their advantage. A lot of strategies are employed to prevent SQL injection attacks at the application level, however, there is currently no viable solution.

This study discussed the most effective SQL injection protection strategies. According to my study, automated techniques for preventing, detecting, and reporting SQL injection attacks in 'stored procedures' are widely used and concrete methods. Small database systems benefit from the graph control mechanism. Prevention involves enforcing better coding practices and database administration procedures. Remember always patch and update holes because exploits are found commonly and the attacker is not going to wait.

10. REFERENCES

1. R. A. Katole, S. S. Sherekar and V. M. Thakare, "Detection of SQL injection attacks by removing the parameter values of SQL query," 2018 2nd International Conference on Inventive Systems and Control (ICISC), 2018, pp. 736-741, doi: 10.1109/ICISC.2018.8398896.
2. P. N. Joshi, N. Ravishankar, M. B. Raju and N. C. Ravi, "Encountering SQL Injection in Web Applications," 2018 Second International Conference on Computing Methodologies and Communication (ICCMC), 2018, pp. 257-261, doi: 10.1109/ICCMC.2018.8487999.
3. G. Su, F. Wang and Q. Li, "Research on SQL Injection Vulnerability Attack model," 2018 5th IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS), 2018, pp. 217-221, doi: 10.1109/CCIS.2018.8691148.
4. S. Vyamajala, T. K. Mohd and A. Javaid, "A Real-World Implementation of SQL Injection Attack Using Open Source Tools for Enhanced Cybersecurity Learning," 2018 IEEE International Conference on Electro/Information Technology (EIT), 2018, pp. 0198-0202, doi: 10.1109/EIT.2018.8500136.
5. R. Zuech, J. Hancock and T. M. Khoshgoftaar, "Detecting SQL Injection Web Attacks Using Ensemble Learners and Data Sampling," 2021 IEEE International Conference on Cyber Security and Resilience (CSR), 2021, pp. 27-34, doi: 10.1109/CSR51186.2021.9527990.
6. A. Algaith, P. Nunes, F. Jose, I. Gashi and M. Vieira, "Finding SQL Injection and Cross Site Scripting Vulnerabilities with Diverse Static Analysis Tools," 2018 14th European Dependable Computing Conference (EDCC), 2018, pp. 57-64, doi: 10.1109/EDCC.2018.00020.
7. A. Maraj, E. Rogova, G. Jakupi and X. Grajqevci, "Testing techniques and analysis of SQL injection attacks," 2017 2nd International Conference on Knowledge Engineering and Applications (ICKEA), 2017, pp. 55-59, doi: 10.1109/ICKEA.2017.8169902.
8. A. Ghafarian, "A hybrid method for detection and prevention of SQL injection attacks," 2017 Computing Conference, 2017, pp. 833-838, doi: 10.1109/SAI.2017.8252192.
9. K. Kamtuo and C. Soomlek, "Machine Learning for SQL injection prevention on server-side scripting," 2016 International Computer Science and Engineering Conference (ICSEC), 2016, pp. 1-6, doi: 10.1109/ICSEC.2016.7859950.
10. R. P. Karuparthi and B. Zhou, "Enhanced Approach to Detection of SQL Injection Attack," 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA), 2016, pp. 466-469, doi: 10.1109/ICMLA.2016.0082.