MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

SUMY STATE UNIVERSITY

Faculty of Electronics and Information Technology

Department _____

BACHELOR THESIS

DETECT AND PREVENT SQL INJECTION VULNERABILITY

Applicant gr. IN-85ан David Ekeh

Supervisor, O. B. Protsenko

Associate Professor, PhD

Head of the department A. S. Dovbysh

Professor, DSc

SUMY 2022

MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

SUMY STATE UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

Approved _____

Head of department Dovbysh A.S.

"___"_____ 2022

The task of a bachelor thesis

Student of the fourth course, group ІН-85ан Ekeh David

Topic: DETECT AND PREVENT SQL INJECTION VULNERABILITY

Approved by order of the Sumy State University

№_____ of _____ 2022

Explanatory note contents: Informational review, methods, and algorithms, implementation, advantages and disadvantages, coding of the SQL.

Date of Task "_____"_____2022
Bachelor work supervisor Protsenko O.B.
Received task to be performed Ekeh David

ABSTRACT

Note: 49 pages, 33 figures, 1 appendix, 22 reference sources.

The object of study – SQL injection attacks

Purpose - SQL injection attack is one of the common techniques for hackers to attack databases. However, the level and experience of programmers matter quite a lot and a considerable number of developers do not determine the authenticity of user input data when writing code, which makes the application security risks. SQL injection attack falls under one of these techniques of database security attack. The database can be protected effectively by the use of database security protection technology. This thesis introduces the principle of SQL injection, the main form of SQL injection attack, the types of injection attack, and how to prevent SQL injection with their examples.

Results – Implementing a successful SQL injection attack that resulted in unauthorized access to sensitive data such as passwords, or personal user information.

Intellij IDEA, C++, SQL Injection Vulnerability, Algorithms

TABLE OF CONTENTS

INTRODUCTION

In this topic, we talk about SQL Injection and understand the terms of SQL Injection and how it is used in a database. But before that, we need to understand the meaning of SQL? And the role in which it takes part in the database. SQL also known as **Structured Query Language** was created by IBM in the early 1970s. SQL is a standardized programming language that is used to manage relational databases or data stream management systems and perform various operations to turn massive collections of structured data into processed data. It is used by database administrators and developers in writing data integration scripts and data analysts trying to set up and run analytical queries.

Also in SQL, there are some RDBMS following:

- Microsoft SQL Server
- Oracle
- MySQL
- PostgreSQL

These are some of the extensions to Standard SQL that add procedural programming language functionality, such as control-of-flow constructs.

1 INFORMATIONAL REVIEW

1.1 SQL Injection Vulnerability

Now that we know a summary of SQL and its origin, we can move to the main topic of SQL Injection. SQL Injection is a test that checks if it is possible to inject data into the application so that it executes a user-controlled SQL query in the database. How testers know if there's a SQL injection vulnerability is if the application uses the tester's input to create SQL queries without proper validation, and successful exploitation allows an unauthorized user to access or manipulate data in the database. During the SQL injection attack consists of the insertion of either a partial or complete SQL query via the data input or transmitted from the client (browser) to the web application and a tester achieves a successful SQL injection attack, they can be able to read sensitive data from the database, modify database data (insert/update/delete), execute administration operations on the database such as shutdown the Database Management Systems (DBMS), recover the content of a given file existing on the DBMS file system, and in few scenarios, there are issues with the commands to the operating system.

It is also a type of injection attack, in which SQL commands are injected into data-plane input to affect the execution of predefined SQL commands.

SQL Injection attacks are separated into three classes:

— In-band or Classic SQLI.

— Out-of-band.

— Inferential or Blind.

1.2 In-band

In-band also known as classic SQLI, allows data to be extracted by using the same channel that is used to inject the SQL code.

There are two subgroups for in-band injection (classic SQL injection) which are Error-based SQLI and Union-based SQLI:

- **Error-based SQL Injection**: This is an in-band SQL injection method where the intruder performs some series of actions that cause the database to produce error messages. The intruder can potentially use the data provided by these error messages to gather information about the structure of the database.
- **Union-based SQL Injection**: Union-based SQLI is a method that takes advantage of the UNION SQL operator, which collects and combines a series of multiple "SELECT" statements created by the database in other to get a single HTTP response. This response may contain data that can be leveraged by the intruder.

1.3 Out-of-band

When the attacker can't use the same channel to launch the attack and gather information, rather Data is retrieved using a different channel, or when a server is too slow or unstable for these actions to be performed. These techniques count on the capacity of the server to create DNS or HTTP requests to transfer data to an intruder.

1.4 Inferential or Blind

Data is not being moved, but the tester can rebuild the information, by sending particular requests and watching the resulting behavior of the DB Server, this method is called blind SQLi.

Blind SQL injections may take longer than their counter-part (in-band) for a tester to exploit, during the test/attack no data is moved via the web application and the tester would be blind during this process. Rather, the tester can re-create the database structure by dispatching load, detecting the web application's response, and the following behavior of the database server. Blind SQL injections can be classified as follows:

- **Boolean**: The tester sends a SQL query to the database prompting the application to return a result. The outcome of the test will differ depending on the fact that either the query is true or false. Relying on the result, the content within the HTTP response will be changed or remain the same. The tester can then work out if the message generated a true or false result.
- **Time-based**: The tester sends a SQL query to the database, which makes the database wait (for seconds) before it can react. The tester can then identify the time in which the database takes to react, considering if the query is true or false. Relying on the result, an HTTP response will be generated instantly or after a waiting period. The attacker can thus work out if the message they used returned true or false, without relying on data from the database.

1.5 Statement of Problem

The presence of an SQL injection vulnerability permits an attacker to reserve instructions immediately to an internet utility's underlying database and subvert the supposed capability of the utility. Once an attacker has recognized an SQLIA vulnerability, the susceptible utility turns into a conduit for the attacker to execute instructions at the database and in all likelihood the host machine itself. SQLIAs are a category of code injection assaults that take gain of a loss of validation of consumer enter. The vulnerabilities arise when builders integrate hard-coded strings with consumer entries to create dynamic queries. If the consumer enter isn't nicely validated, attackers can form them enter in this type of manner that, while it's far covered withinside the very last question string, elements of the entrance are evaluated as SQL key phrases or operators through the database.

Consider an easy SQL injection vulnerability. The following code builds a SQL question by concatenating a string entered through the users with hardcoded strings:

```
String query = "SELECT * FROM items WHERE owner = '" + userName + "'
AND itemName = '" + ItemName.Text + "'";
```

This query intends to look for all items that fit the item name entered through a user. In the instance above, userName is the presently authenticated user and ItemName.Text is the input supplied by the user. Suppose an ordinary user with the username john enters benefit withinside the net shape. That value is taken from the form and included in the query as part of the SELECT condition. The completed query will then appear similar to the following:

SELECT * FROM items WHERE owner = 'john' AND itemName = 'benefit'

However, since the query is built dynamically by concatenating a consistent base query string and a user-supplied string, the query at most behaves exactly if itemName

does now no longer include a single quote (') character. If an attacker with the username john enters the string:

‘anything' OR 'a'='a

The following query will be:

SELECT * FROM items WHERE owner = 'john' AND itemName = 'anything' OR 'a' = 'a'

The addition of the OR 'a'='a' condition brings about the WHERE clause to consistently evaluate to true. The query then becomes logically equal to the much less selective query:

SELECT * FROM items

The simplified query permits the attacker to examine all entries saved withinside the items table, removing the constraint that the query at most returns items owned via the authenticated user. In this case, the attacker can examine the processed data he oughtn’t to be capable of accessing.

Now presume that the attacker enters the subsequent:

‘anything'; drop table items— In this case, the subsequent query is constructed via script: SELECT * FROM items WHERE owner = 'john' AND itemName = 'anything'; drop table items--'

The semicolon (;) denotes the conclusion of one query and the beginning of another. Many database servers permit more than one SQL statement separated with the aid of using semicolons to be performed together. This permits an attacker to perform arbitrary commands against databases that allow more than one statement to be performed with one call. The double hyphen (--) suggests that the remainder of the current line is a comment and ought to be ignored. If the changed code is syntactically correct, it will be carried out by the server. When the database server undertakes these two queries, it’ll first choose all records in items that equal to the value of anything

belonging to the user john. Then the database server will drop, or erase the whole items table.

2 METHODS AND ALGORITHMS

2.1 Techniques on How to Test for SQLI

Detection Techniques

Firstly, in this test, we need to understand when the application interacts with a DB Server to access some data. Some examples of cases when an application needs to interact with a DB include:

- Authentication forms: When authentication is performed using a web form, odds are that the user credentials are checked against a database that contains all usernames and passwords.
- E-Commerce sites: The products and their characteristics (price, description, availability) are very likely to be saved in the database.
- Search engines: The string is sent forth by the user so it could be used in a SQL query in other to extract all relevant records from a database.

The tester has to create a list of all input fields whose values could be used in making a SQL query, including the hidden fields of post requests, and then test them separately, trying to interfere with the query and create a flaw. Also, we have to review the HTTP headers and Cookies.

The very first test normally involves the adding of a single quote (‘) or a semicolon (;) to the parameter or area under test. The first is used in SQL as a string terminator, and if not it is filtered by the application, which would lead to an incorrect query. The second is used to end a SQL statement and if it is not also filtered, then it is likely to create an error. The result of a vulnerable field might be similar to the following (this is a Microsoft SQL Server, which we would use in this case):

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotation mark before
the
character string ''.
/target/target.asp, line 113
```

Figure 2.1 - Microsoft SQL Server

Also, comment delimiters (-- or /**/) and other SQL keywords such as (AND) and (OR) can be also used to try to change the query. A very simple but often, still effective technique is just to insert a string where a number is expected, as an error like the following might be generated:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the
varchar value 'test' to a column of data type int.
/target/target.asp, line 113
```

Figure 2.2 - Microsoft SQL Server

Observe all the responses from the webserver and have a look at the HTML/JavaScript source code. Sometimes the error exists inside them but for some reason (e.g. JavaScript error, HTML comments, etc) is not presented to the user. A full error message, like those in the examples, provides a wealth of information to the tester to mount a successful injection attack. Although the applications often do not provide so much detail: a simple ‘500 Server Error’ or a custom error page might be present, meaning that we need to use blind injection techniques. In any case, it is very important to test each field separately: only one variable must vary while all the others

remain constant, to fully understand which parameters are vulnerable and which are not.

2.2 Standard SQL Injection Testing

Classic SQL Injection

The diagram below is a simple SQL query

```
SELECT * FROM Users WHERE Username='$username' AND
Password='$password'
```

Figure 2.3- simple SQL query

This type of query is mostly used by a web application to authenticate the user. If the query returns a value, then that means that inside the database there's a user with that set of authorization that exists, the user is then allowed to login into the system, otherwise, they would be denied access. The values of the input fields are normally obtained by the user through a web form. Let's assume that we insert the following statement below

```
$username = 1' or '1' = '1
$password = 1' or '1' = '1
```

Figure 2.4 - Username and Password values

Then this query will be altered and would appear as shown below:

```
SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND Password='1'
OR '1' = '1'
```

Figure 2.5 – Using a GET Technique

Let us assume that the values of the parameters are sent to a server through a GET technique, there we can then generate a vulnerable domain, then web site would be like this **www.example.com**, then the request that we may carry out would look like the diagram below:

```
http://www.example.com/index.php?
username=1'%20or%20'1'%20=%20'1&amp;password=1'%20or%20'1'%20=%20'1
```

Figure 2.6 - www.example.com

A brief analysis shows that the query returns a value or a set of values due to the condition is always true “OR 1=1”. Thus the system can authenticate the user without knowing the username and password.

SELECT Statement

The SQL query diagram below:

```
SELECT * FROM products WHERE id_product=$id_product
```

Figure 2.7 – Using a SELECT STATEMENT

By applying the code from the diagram above to the query below

```
http://www.example.com/product.php?id=10
```

Figure 2.8 - Description of a product

When a developer attempts a valid value and the application will return the description of a product. The optimal solution in testing if the application is vulnerable in this case is to manipulate the logic by using the operators AND and OR. As shown below.

```
http://www.example.com/product.php?id=10 AND 1=2
SELECT * FROM products WHERE id_product=10 AND 1=2
```

Figure 2.9 - AND and OR.

Presumably, the application might return some text displaying to us that there's no content available or leaving a blank page. The developer can then send a true statement and scan if there is a valid result:

```
http://www.example.com/product.php?id=10 AND 1=1
```

Figure 2.10 – Using a True Statement

Stacked Queries

Based on the type of API (Application Programming Interface), which the web application is using and the DBMS (Database Management System) such as PHP + PostgreSQL or ASP+SQL SERVER may be possible to execute multiple queries in a go.

The SQL query below is a good example

```
SELECT * FROM products WHERE id_product=$id_product
```

Figure 2.11 - SQL query

In the diagram above we can exploit it and the result would be shown below:

```
http://www.example.com/product.php?id=10; INSERT INTO users (…)
```

Figure 2.12 – A Stacked Queries

We can assume that this way is possible to execute many queries in a row and independent of the first query.

Fingerprinting the Database

Although the SQL language is a standard programming language, even so, every DBMS has its peculiarity and differs from each other in many aspects such as comments line, functions to retrieve data such as user's names and databases, special commands, and features, etc.

While the developers progress to a more advanced SQL injection exploitation method, they also need to know what the back-end database is all about.

Errors Returned by the Application

One of the ways to find out what a back-end database is used is by accessing the error returned by the application. There are some examples below that show error messages

```
You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the
right syntax to use near '\'' at line 1
```

Figure 2.13 - MySql

Another is a complete UNION SELECT with version (), which can also assist in educating the developer in knowing the back-end database.

```
SELECT id, name FROM users WHERE id=1 UNION SELECT 1, version() limit
1,1
```

Figure 2.14 - UNION SELECT

```
ORA-00933: SQL command not properly ended
```

Figure 2.15 – Oracle

```
Microsoft SQL Native Client error '80040e14'
Unclosed quotation mark after the character string

SELECT id, name FROM users WHERE id=1 UNION SELECT 1, @@version limit 1, 1
```

Figure 2.16 - MS SQL Server

```
Query failed: ERROR: syntax error at or near
"'" at character 56 in /www/site/test.php on line 121.
```

Figure 2.17 - PostgreSQL

In a scenario where no error message or a custom error message is displayed then the developer can attempt to inject into string fields using varying concatenation methods such as:

- PostgreSQL: ‘test’ // ‘ing’
- SQL Server: ‘test’ ‘ing’
- Oracle: ‘test’ // ‘ing’
- MySql: ‘test’ + ‘ing’

2.3 SQL Injection Examples

A tester trying to test an SQL injection has to manipulate a standard SQL query to exploit non-validated input vulnerabilities in a database. There are many ways that this attack vector can be carried out, several of which will be displayed here to show us a general idea about how SQLI works.

For example, the above-mentioned input, which pulls information for a specific product, can be altered to read http://www.estore.com/items/items.asp?itemid=999 or 1=1.

As a result, in (Figure. 2.18) the corresponding SQL query looks like this:

```
SELECT ItemName, ItemDescription
FROM Items
WHERE ItemNumber = 999 OR 1=1
```

Figure. 2.18 - eStore database query being altered

And since the statement 1 = 1 is always true, the query returns all of the product names and descriptions in the database, even those that you may not be eligible to access.

Attackers are also able to take advantage of incorrectly filtered characters to alter SQL commands, including using a semicolon to separate two fields.

For example, in (Figure. 2.19) the input http://www.estore.com/items/iteams.asp?itemid=999; DROP TABLE Users would generate the following SQL query:

```
SELECT ItemName, ItemDescription
FROM Items
WHERE ItemNumber = 999; DROP TABLE USERS
```

Figure. 2.19 - eStore database query using DROP TABLE

As a result, the entire user database could be deleted.

Another way SQL queries can be manipulated is with a UNION SELECT statement. This combines two unrelated SELECT queries to retrieve data from different database tables.

For example, in (Figure. 2.20) the input http://www.estore.com/items/items.asp?itemid=999 UNION SELECT user-name, password FROM USERS produces the following SQL query:

```
SELECT ItemName, ItemDescription
FROM Items
WHERE ItemID = '999' UNION SELECT Username, Password FROM Users;
```

Figure. 2.21 - UNION SELECT statement

Using the UNION SELECT statement, this query combines the request for item 999's name and description with another that pulls names and passwords for every user in the database.

3 IMPLEMENTATIONS

3.1 SQLite Injection Attacks

The SQLite injection can be described as a process of injecting some malicious code to gain access to other databases while gaining input from a web application.

Let us assume that there is a registration page where the user needs to enter a username, rather than that the user enters SQLite statement instead so that it will run on our database and return the data based on their query statement.

The key aim for SQLite injection attacks is to get protected information from their database and to perform some vulnerable actions such as improving the existing records, information or deleting/drop tables in the database, etc.

Normally, these SQLite injection attacks can happen whenever your application depends on the user input to build the SQLite query statements. And while taking the input from users we need to authenticate that particular data before we send it to the database by outlining the pattern confirmation or accepting the input parameters in a standard form.

SQLite Injection Attacks Example

Now that we understand how SQLite injection attacks work, we will now see how SQLite injection attacks can happen and how we can prevent them with examples for that create we need to create a table called "**emp_master**" in the database using the following queries.

```
CREATE TABLE emp_master
(emp_id INTEGER PRIMARY KEY AUTOINCREMENT,
first_name TEXT,
last_name TEXT,
salary NUMERIC,
dept_id INTEGER);

INSERT INTO emp_master
values (2,'Shweta','Jariwala', 19300,2),
(3,'Vinay','Jariwala', 35100,3),
(4,'Jagruti','Viras', 9500,2),
(5,'Shweta','Rana',12000,3),
(6,'sonal','Menpara', 13000,1),
(7,'Yamini','Patel', 10000,2),
(8,'Khyati','Shah', 500000,3),
(9,'Shwets','Jariwala',19400,2),
(12,'Sonal','Menpara', 20000,4);
```

Figure 3.1 - **emp_master** table

Now we will run the following query in (Figure 1.2) to check the records of the "**emp_master**" table.

```
sqlite> SELECT * FROM emp_master;

emp_id      first_name  last_name   salary      dept_id
----------  ----------  ----------  ----------  ----------
2           Shweta      Jariwala    19300       2
3           Vinay       Jariwala    35100       3
4           Jagruti     Viras       9500        2
5           Shweta      Rana        12000       3
6           Sonal       Menpara     13000       1
7           Yamini      Patel       10000       2
8           Khyati      Shah        50000       3
9           Shwets      Jariwala    19400       2
12          Sonal       Menpara     20000       4
```

Figure 3.1 - **emp_master** table

Let's assume that in our web application we are acknowledging the input from the user to show user details, during the time the user enters the query as shown below then it will return all the records from the table irrespective of employee id.

```
12 or 1 = 1
```

Figure 3.2 – code that shows user details

The above (Figure 3.2 – code that shows user details) input will form the query as shown below.

```
SELECT * FROM emp_master WHERE emp_id = 12 or 1=1;
```

Figure 3.3 – SQLIa code

Here in above (Figure 3.3) query WHERE **1=1** is always returning true and in **OR** operator one operand **TRUE** means, the whole condition will return it as **TRUE** then it will return all the records from employee table irrespective of employee id like as shown in (Figure 3.4).

```
SELECT * FROM emp_master WHERE emp_id = 12 or 1=1;

emp_id      first_name  last_name   salary      dept_id
----------  ----------  ----------  ----------  ----------
2           Shweta      Jariwala    19300       2
3           Vinay       Jariwala    35000       3
4           Jagruti     Viras       9500        2
5           Shweta      Rana        12000       3
6           Sonal       Menpara     13000       1
7           Yamini      Patel       10000       2
8           Khyati      Shah        49900       3
9           Shwets      Jariwala    19400       2
12          Sonal       Menpara     20000       4
```

Figure 3.4 - SQLIa code grants access to user details

This way hackers can easily get all the sensitive information just by injecting a few pieces of code.

To prevent this type of SQLite injection we need to acknowledge the user input as a string and then carry out the operation on the database as shown in (Figure 3.5).

```
SELECT * from emp_master where emp_id='12 OR 1=1';
```

Figure 3.5 - “12 OR 1=1” is used to avoid SQLite injection attacks

Whenever we run the above query it will return employee details whose employee id matches with “12 OR 1=1” this is one of the ways to avoid SQLite injection attacks.

3.2 Methods to Detect SQL Injection Attacks

Some methods used for detecting SQL injection attacks are listed below:

- Dynamic analysis
- Static analysis

3.3 Dynamic Analysis

Dynamic analysis is an upgrade over static analysis since it can detect the vulnerabilities from SQL injection attacks. Also, it’s able to spot several other kinds of vulnerabilities efficiently.

How does it work? By collecting the SQL queries between the client and the application, also between the application and the database. After that it then analyzes the vulnerabilities, thus using the SQL inject attack codes to understand the vulnerability and also without making any adjustments to web applications. Open source program such as Paros scans not only SQL injection attacks vulnerabilities, but also other vulnerabilities within the web application. Although Paros is not perfect because it uses predetermined attack codes to scan, also it uses HTTP response to the success rate of the attack. As Sania locates and extracts SQL injection attack

vulnerabilities between the web application and databases, it then proceeds to create SQL Injection attack codes. After attacking with the created code, it extracts the SQL query from the attack. After the normal SQL query is compared and examined with the SQL query gathered from the attack using the parse tree, the success rate of the attack is confirmed. Since a parse tree is being used, then the Sania can be said to be more accurate rather than using an HTTP response verification. The dynamic analysis technique is said to be advantageous because no web application tweaking/alteration is necessary. Nevertheless, the vulnerabilities found in the web application must be physically fixed by the developers and not all of them can be located without preplanned attacks.

3.4 Table of Dynamic Techniques with Advantages & Disadvantages

Technique	Advantages	Disadvantages
SQLIMW is a middleware specifically designated for the detection of SQL injection attacks.	Efficient, transparent to the programmer	Limited coverage, bet environment is a single-sign-on system
SVM for prediction of SQLIA	Low overhead High detection rate	Produce false positives in some cases
TransSQL	Applicable for a different platform, easy to deploy	High latency (because every query needs to be executed twice)
Security testing schemes are based on automatic test case generating and simulated tests.	Automated, effective	The problem in coverage(attack library should be improved)

SQL-IDS (injection detection system)	No need to change on code, low overhead, high coverage	Limited to java
Artificial Neural Network-based web application firewall for SQL injection.	Independent in a matter of platform	Erroneous results may be produced

Examples of Dynamic Tools

- Abbey Scan
- Acunetix
- APIsec
- App Scanner
- AppCheck Ltd
- AppScan
- AppScan on Cloud
- Arachni

3.5 Static Analysis

This method analyzes the SQL query sentence within a web application to reduce the chances of SQL injection attacks by detecting and preventing them. It is not an accurate detection method because if a corrupted input has the correct type of syntax, then the analysis will backfire in recognizing the attack. Also, it requires rewriting of web applications, the aim of the static analysis method is not solely by detecting and preventing SQL injection attacks but to verify the user's input type to reduce potential SQL Injection attacks.

The JDBC-Checker uses Java String Analysis (JSA) library to validate the real-time user's input and in turn, prevent SQL Injection attacks. nevertheless, if the user inputs malicious data which has the correct type and syntax, then it cannot be prevented. Also, we can look at some static techniques with their advantages and disadvantages in the table below:

3.6 Table of Static techniques with advantages & disadvantages

Technique	Advantages	Disadvantages
SAFELI: (MSLI) to detect SQL Injection Attacks.	This approach can find and detect vulnerabilities that can't be found by black-box testing.	Should add code transformation in MSLI. Work on Microsoft platform only.
Mining input sanitization patterns for predicting SQLIVS.	85% reported a detection rate	High false positive
An algorithmic approach for replacing insecure SQL statements in the code with a secure ones.	94% reported accuracy	Not active for Java- Code transformation Batch queries can't be detected efficiently.
Automated fix Generator SQLIAs	Fully automated. High Efficiency	Should be supported by code transformation. Overhear is high Limited to PHP
Automatic creation of SQL injection attacks for	There is no runtime overhead. No need to modify the target system.	Sometimes generate false positives. Limited to PHP and MYSQL

uncovering SQL injection vulnerabilities.		
Signature and auditing method to prevent SQLIAs using	Show low overhead in execution. No need for code transformation	It's restricted to web applications only.
SQL DOM	Show high coverage all over SQLIA (Except Stored Procedure)	Cost high overhead on the system. Need a new model for programming
A method for hunting SQL injection vulnerabilities.	Effective, the high detection rate - Very low overhead	Complex involves different stages
An anomaly-based system that uses different detection models to detect unknown attacks.	Low overhead	- Limited to PHP - Coverage problems - Generates false-negative results
ASSIST (Automatic and Static SQL Injection Sanitization Tool.	Effective, low overhead, high detection rate.	Used for JAVA only, need code transformation, produce false positive and true negatives in some cases
WebSSAR: Check Input Validation against preconditioning	Good for the toy system.	Some preconditions may not be accurately expressed for some filters.

Examples of Static Tools

- 42Crunch
- Bandit

- Agnitio
- Codacy
- CoGuard
- Bearer
- DeepSource

3.7 SQL injection prevention techniques

Stopping SQL injection attacks takes a lot of making sure that none of the fields are vulnerable to invalid inputs and application execution. For a beginner developer/user, it would be impossible to manually check every page and every application on the website, especially when updates would frequent and user-friendliness would take top priority.

However, security analysts and seasoned developers suggest a series of subsequent points that would guarantee other's database square can be measured well and protected inside the confinement of their server.

Although, there must be a way to simply sanitize user input and make sure that an SQL injection is infeasible. Sadly, that is not always the case. There may be a series of ways to sanitize user input, from globally applying PHP's add-slashes () to everything, which might bring unfortunate results. Then down to implementing the sanitization to "clean" variables at the time of assembling the SQL query itself. However, implementing sanitization at the query itself may be a very poor coding decision or practice and can be difficult to maintain or keep up to date with. This is where database systems have implemented the use of prepared statements.

We can talk about different ways we can prevent **Structured Query Language Injection Attacks** (SQLIAs), with their definitions and their examples as well.

- Prepared Statements
- Stored Procedures
- Input Validation or Input Sanitizing

The image below (Figure 3.6) is a (Java) example that is vulnerable, which would allow an attacker to inject code into the query that would be executed by the database. The un-validated "**customerName**" parameter is simply attached to the query, which allows an attacker to inject any SQL code they want. Sadly, this technique for accessing databases is all too frequent.

```
String query = "SELECT account_balance FROM user_data WHERE user_name = "
             + request.getParameter("customerName");
try {
    Statement statement = connection.createStatement( ... );
    ResultSet results = statement.executeQuery( query );
}
...
```

Figure 3.6 – Java Database

3.8 Prepared Statements

How prepared statements can be applied with variable binding (also known as parameterized queries or “pre-compiled statements”) should first be taught to all developers/users on how to write database queries. The prepared statements are

simple to write and easier to understand than dynamic queries. Parameterized queries (prepared statements) force the developer/user to first define all the SQL code and then pass each parameter to the query later. This coding technique allows the database to differentiate between code and data, regardless of what user/developer input is supplied.

Prepared statements, makes sure that an attacker cannot change the intent of a query, even if SQL commands are incorporated by an attacker/intruder. In the example below we will show a secure java prepared statement, if an attacker/intruder were to enter the **userID** of [tom' or '1'='1] the parameterized query would not be vulnerable and would instead search for a username that exactly matches the entire string [tom' or '1'='1].

There are some Language specific recommendations used for the prepared statement:

- **C# .NET**: – uses parameterized queries like [**SqlCommand()**] or [**OleDbCommand()**] with bind variables
- **Hibernate Query Language (HQL)**: – uses [**createQuery()**] with bind variables (also known as named parameters in Hibernate)
- **PHP**: – uses PDO with strongly typed parameterized queries (using [**bindParam()**])
- **Java EE**: – uses [**PreparedStatement()**] with bind variables
- **SQLite**: – uses [**sqlite3_prepare()**] to create a statement object

In an unlikely situation, prepared statements can harm the performance of the database. When faced with this situation, there are some ways to solve it:

- strongly validate all data.
- escape all user-supplied input using an escaping routine certain to the user's database vendor as explained in the below example. It would be more effective than using a prepared statement.

Example of a Secure Java Prepared Statement

The following example code uses a [**PreparedStatement**], Figure 3.7 is a Java implementation of a parameterized query, to run in the same database query.

```
// This should REALLY be validated too
String custname = request.getParameter("customerName");
// Perform input validation to detect attacks
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

Figure 3.7 - Secure Java Prepared Statement

Example of a Secure C# .NET Prepared Statement

Using the [**.NET**] is straightforward. Because the creation and execution of the query cannot be altered. All that is required to do is simply to pass the parameters to the query using the [**Parameter.Add()**] call as displayed in the diagram below (Figure 3.8).

```
String query = "SELECT account_balance FROM user_data WHERE user_name = ?";
try {
  OleDbCommand command = new OleDbCommand(query, connection);
  command.Parameters.Add(new OleDbParameter("customerName", CustomerName Name.Text));
  OleDbDataReader reader = command.ExecuteReader();
  // …
} catch (OleDbException se) {
  // error handling
}
```

Figure 3.8 - a Secure C# .NET Prepared Statement

3.9 Stored Procedures

Stored procedures cannot give 100% percent protection from SQL injection. Although, there are certain standard stored procedure programming constructs that have the same effect as the use of parameterized queries when applied safely, which is expected for most stored procedure languages.

This requires that the developer construct an SQL statement with parameters that are automatically parameterized, except if the developer does something which is unusually out of the normal way. There is a difference between prepared statements and stored procedures which is that the SQL code for a stored procedure can be defined and stored in the database itself, and can then be retrieved from the application. These two techniques have the same effectiveness in blocking the SQL injection so the developer can choose which approach makes the most suitable sense for them.

We should understand that 'Implemented safely' means that the stored procedure does not include any potential threat in the dynamic SQL generation.

Developers do not usually construct dynamic SQL inside the stored procedures. Although, this can be done, even so, should be avoided. And If it cannot be avoided, then the stored procedure must use input validation or proper escaping as explained in other to make sure that all the user-supplied input to the stored procedure cannot be used to inject SQL code into the dynamically constructed query. Listeners should always look for uses of **sp_execute**, run, or exec within SQL Server stored procedures. Similar survey guidelines are a must for similar functions for other vendors.

I would like to add that there are also several occurrences where stored procedures risk is increased. For example, on an MS SQL server, the user can have 3 main default roles such as **db_datareader**, **db_datawriter,** and **db_owner**. Before the existence of stored procedures and they're coming into use, a database administrator(DBA) would give **db_datareader** or **db_datawriter** license to the web service's user, based on the requirements. Nonetheless, stored procedures needs execute rights, a role that is not accessible from the start. There are some setups where the user management is being centralized and has been limited to these 3 roles, cause all the web apps run under the **db_owner** rights so stored procedures can work. Normally, this means that if a server is violated by the attacker, they then have full rights to the database, where previously they might only have had read-only access.

Example of a Secure Java Stored Procedure

Figure 3.9 below uses a **CallableStatement**, the java implementation of the stored procedure interface, in other to run it in the same database query. The

sp_getAccountBalance stored procedure would have to be pre-decided in the database and then implement the same functionality as the query defined above.

```
// This should REALLY be validated
String custname = request.getParameter("customerName");
try {
  CallableStatement cs = connection.prepareCall("{call sp_getAccountBalance(?)}");
  cs.setString(1, custname);
  ResultSet results = cs.executeQuery();
  // … result set handling
} catch (SQLException se) {
  // … logging and error handling
}
```

Figure 3.9 - Java Stored Procedure

Example of a Secure VB .NET Stored Procedure

In this example of a secure VB .NET Stored Procedure, it uses a **SqlCommand**, this .NET implementation of the stored procedure interface, to run it in the same database query. The **sp_getAccountBalance** stored procedure would have to be pre-decided in the database and then applied in the same functionality as the query defined above.

```
Try
  Dim command As SqlCommand = new SqlCommand("sp_getAccountBalance", connection)
  command.CommandType = CommandType.StoredProcedure
  command.Parameters.Add(new SqlParameter("@CustomerName", CustomerName.Text))
  Dim reader As SqlDataReader = command.ExecuteReader()
  '...
Catch se As SqlException
  'error handling
End Try
```

Figure 3.10 - VB .NET Stored Procedure

3.10 Input Validation or Input Sanitizing

Different sections of the SQL queries aren't in their valid locations for the use of bind variables, like names of tables or columns, and let’s not forget the sort order indicator (ASC or DESC). In similar scenarios, input validation is often the suitable defense. considering the names of tables or columns, in theory, these values come from the code, and not from the developer’s parameters.

If the user parameter values are being used for targeting different table names and column names, then we can assume that the parameter values should be mapped to the right/expected table or column names to make sure invalidated user input doesn't end up in the query. This can be a case of poor design and a full reconstruction should be considered if there’s a chance to do so.

Example of table name validation.

```
String tableName;
switch(PARAM):
  case "Value1": tableName = "fooTable";
                 break;
  case "Value2": tableName = "barTable";
                 break;
  ...
  default      : throw new InputValidationException("unexpected value provided"
                                                    + " for table name");
```

Figure 3.11 - Table Name Validation

The **tableName** in the above diagram shown as (Figure 3.11), can then be directly included in the SQL query, for a while now it is known to be one of the valid and expected values for a table name in the query. We should bear in mind that generic

table validation functions can lead to data loss while the table names are used in queries where they are not expected.

A thing as simple as sort order can be better if the user-supplied input is changed to a boolean, and then that boolean is used to select the secure value to add to the query. It is a very standard requirement in dynamic query generation.

Example

```
public String someMethod(boolean sortOrder) {
 String SQLquery = "some SQL ... order by Salary " + (sortOrder ? "ASC" : "DESC");`
 ...
```

Figure 3.12 – Using Boolean Technique

At any moment a user input can be altered to a non-String, such as a date, numeric, boolean, enumerated type, etc. before it is applied to a query, or used to set a value to be added to the query, then we can say that there's a guarantee that it is secure enough to do so.

CONCLUSION

In today's modern era, SQL injection attacks are a growing criminal threat to the user's web applications, especially those that access sensitive data. The attacker can modify, delete data, and perform database/server shutdown taking advantage of the vulnerabilities in the system. Defensive measures require much greater investment in deployment and support and should be used only on the most important or sensitive applications. However, as technology develops, so will the threats and techniques used by malicious users. As storage on the internet is more trend nowadays care should be taken to secure the data from being stolen by malicious users. With that in mind, we can say that one of the most important things we can do to protect our applications from SQL injection attacks is "**Coding Defensively**". And what I mean by that statement is spending time educating your developers on basic security practices. The time you spend up-front will be far less than you would spend cleaning up the mess if the vulnerabilities make their way into production. Although It's long been argued that fixing bugs during development is far more effective than fixing them in later phases, the same holds here. Also in other to put up a good defense is to use prepared statements anywhere you're passing input from the user to the database through regular expressions, throwing out potentially dangerous input before sending it to any backend resource such as a database, or command line, or web service. Note that a user/developer should not make the hacker's job easy by spelling out SQL details in your error messages.

REFERENCES

1. SQL (Structured query language) Injection - https://www.imperva.com/learn/application-security/sql-injection-sqli/
2. SQLite Injection Attacks - https://www.tutlane.com/tutorial/sqlite/sqlite-injection-attacks
3. SQL Injection Prevention Cheat Sheet - https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
4. Testing for SQL Injection - https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection
5. SQL Injection Testing Tutorial - https://www.softwaretestinghelp.com/sql-injection-how-to-test-application-for-sql-injection-attacks/
6. How to Detect & Prevent SQL Injection Attack - https://datadome.co/bot-management-protection/how-to-prevent-bot-driven-sql-injection-attacks/
7. What Is SQL Injection Attack and How Can We Detect It? - https://zindagitech.com/what-is-sql-injection-attack-and-how-can-we-detect-it/
8. Dynamic Application Security Testing (DAST) - https://www.techopedia.com/definition/30958/dynamic-application-security-testing-dast
9. SQL Injection - https://www.w3schools.com/sql/sql_injection.asp
10. PortSwigger - https://portswigger.net/web-security/sql-injection
11. Wiki - https://en.wikipedia.org/wiki/SQL_injection

12. SQL Injection Cheat Sheet - https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/

13. SQL Injection - https://www.contrastsecurity.com/knowledge-hub/glossary/sql-injection

14. SQL Injection Attack - https://brightsec.com/blog/sql-injection-attack/

15. What is an SQL injection attack? - https://www.rapid7.com/fundamentals/sql-injection-attacks/

16. How to Protect Against SQLIa - https://security.berkeley.edu/education-awareness/how-protect-against-sql-injection-attacks

17. What is SQLIa - https://www.tek-tools.com/security/sql-injection-attack

18. Kaspersky - https://www.kaspersky.com/resource-center/definitions/sql-injection

19. Guru99 - https://www.guru99.com/learn-sql-injection-with-practical-example.html

20. Simplilearn - https://www.simplilearn.com/tutorials/cyber-security-tutorial/what-is-sql-injection

21. CrowdStrike - https://www.crowdstrike.com/cybersecurity-101/sql-injection/

22. AVG - https://www.avg.com/en/signal/sql-injection

APPENDIX A

Software Implementation

Code for implementing software String searching in C

```
#include <sys/time.h>
#include <time.h>
#include <stdio.h>
#include <string.h>

void search (char* pattern, char* text)
{
	int M = strlen(pattern);
	int N = strlen(text);

	for (int i = 0; i <= N - M; i++) {
		int j;
		for (j = 0; j < M; j++)
		if (text[i + j] != pattern[j])
		break;
	if (j == M)// if pattern[0...M-1] = text[i, i+1, ...i+M-1]
	{
		printf("Keyword *%s* found at index %d n\n\n",pat, i);
```

```
        }
}
}
int main()
{
struct timeval start,end;
int i;
//printf("input the username/password:");
//scanf("%s",&txt);
char text[] = "O'Brian";
int pattern_length = 25;
double sum=0;
double temp=0;
char pattern[25][10] =
{"and","or","between","not","insert","set","delete","like","in","join","union","int
o","--
","create","drop","alter","add",";","all","'","any","exists","some","as","kill"};
int count=1000;
for(int j=0;j<count;)
{
                gettimeofday(&start,NULL);
        for(int i=0;i<pat_length;i++)
        {
                Search(pat[i], txt);
        }
                gettimeofday(&end,NULL);
```

```
        temp = (end.tv_sec*1000000 + end.tv_usec) - (start.tv_sec*1000000
start.tv_usec);
        sum+=temp;
        if(temp!=0)
        {   j++;
                printf("Iteration: %d, Time: %f\n",j,temp);
                }
        }
        sum=sum/count;
        printf("Average: %f",sum);
return 0;
```

APPENDIX B

Software Implementation

To test for HTTP response using Intellij IDEA

```
1. Package sumdu.tss.in85;


2. import kong.unirest.Unirest;
3. import org.junit.jupiter.api.AfterAll;
4. import org.junit.jupiter.api.BeforeAll;
5. import org.junit.jupiter.api.Test;
6. import sumdu.tss.in85.helper.Keys;
7. import sumdu.tss.in85.helper.utils.ResourceResolver;

8. import java.io.File;
9. import java.util.Arrays;

10. import static org.junit.jupiter.api.Assertions.assertEquals;
11. import static org.junit.jupiter.api.Assertions.assertTrue;
12. /*
        - main idea:
        - for valid tables form database service should return code
200
        - for valid tables text of page should contain table name
        - for un-existed pages service should return 404 code
        - for sqlite system tables service should return 404 code
        - Let's prepare database with tables which name we know
```

- for this test is enough code and body of http response, so we did not need Selenium

```
13. */
14. public class ExampleFunctionTest {
    - private static Server app = null;

    - @BeforeAll
    - static void initServer() {
        i. //file from src/test/resources prepared for this test
        ii. File file = ResourceResolver.getResource("example-functional-test.properties");
        iii. Keys.loadParams(file);

        iv. app = new Server();
        v. app.start(Integer.parseInt(Keys.get("APP.PORT")));
    - }

    - @AfterAll
    - static void stopServer() {
        i. app.stop();
        ii. app = null;
    - }

    - @Test
    - public void service_should_return_200_code_for_valid_tables() {
```

i. var listOfValidTableNames = Arrays.asList("first_table", "second_table");
ii. for (var validTableName : listOfValidTableNames) {
iii. var response = Unirest.get(app.getBaseUrl() + validTableName).asEmpty();
iv. assertEquals(200, response.getStatus());
- }

15. }

- @Test
- public void service_should_contain_table_name_for_valid_tables() {
i. var listOfValidTableNames = Arrays.asList("first_table", "second_table");
ii. for (var validTableName : listOfValidTableNames) {
iii. var response = Unirest.get(app.getBaseUrl() + validTableName).asString();
iv. assertTrue(response.getBody().contains(validTableName));
- }

16. }

- @Test
- public void service_should_return_404_code_for_invalid_tables() {

i. var listOfUnlistedTableNames = Arrays.asList("unlisted_table", "other_ unlisted _table");
ii. for (var validTableName : listOfUnlistedTableNames) {
iii. var response = Unirest.get(app.getBaseUrl() + validTableName).asEmpty();
iv. assertEquals(404, response.getStatus());
- }
17. }

- /**
- this test fail because lack of protection inside the app
- */
- @Test
- public void service_should_return_404_code_for_system_tables() {
i. var listOfSystemTableNames = Arrays.asList("sqlite_master", "sqlite_sequence");
ii. for (var validTableName : listOfSystemTableNames) {
iii. var response = Unirest.get(app.getBaseUrl() + validTableName).asEmpty();
iv. assertEquals(404, response.getStatus());
- }
- }
18. }