SQL Injection Attack – Technical Research

Naghul Pranav A S | Tharaneeshwaran V U | Krishna J

NATIONAL INSTITUTE OF TECHNOLOGY PUDUCHERRY
Thiruvettakudy, Karaikal – 609 609

Abstract

SQL injection (SQLi) is an internet security vulnerability that allows an attacker to tamper with the queries an application makes to its database. SQLi allows the attacker to retrieve data within the database which they are normally not authorised to view. The data may include other users' data along with any data the application can access. SQLi attacks usually lead to the erasure of vital data that causes significant change to the application's content and behaviour. SQLi attacks are the basis for the more intricate DoS (denial-of-service) attack. This paper briefly introduces the concept of SOL injection attack and principle, and the realization process of SQL injection attack and on this basis summarizes the general SQL injection attack prevention methods.

Keywords: SQL Injection, Web Security

1. Introduction

SQL injections (SQLi) represent one of the most prevalent and concerning attack vectors against web applications. Even as recently as 2019, SQLi accounted for nearly two-thirds of web application attacks, underscoring their continued relevance and threat. What's surprising is that SQLi has been a part of the top 10 Common Vulnerabilities and Exposures (CVE) since 2003, and it has only grown in popularity in recent years. In this report, we will delve into the world of SQL injections, exploring their various forms and, more importantly, how to shield your applications from their potentially devastating impact. Furthermore, we'll emphasize the critical role of automated security testing in your DevOps pipeline to intercept these vulnerabilities before they can wreak havoc in a production environment.

2. Understanding SQL Injections

SQL injections are web application attacks in which attackers cleverly "inject" SQL statements into user inputs. This manipulation can grant unauthorized access to application data, whether it's sensitive or publicly accessible. The vulnerability lies in the areas of web applications where user inputs are not thoroughly sanitized, allowing attackers to breach the application's database.

3. An Illustrative SQL Injection

Attackers frequently infiltrate web applications through elements such as usernames, user IDs, or various form submissions. Imagine a typical scenario where a user provides their user ID, resulting in a legitimate SQL query like the following:

```
Sql> SELECT * FROM users WHERE id = '42'
```

However, the trouble begins when input sanitization is lacking. An attacker can input a malicious SQL statement, for instance:

```
Sql> '42' OR '1'='1'
```

In the absence of proper sanitization, this input would execute the following SQL query:

```
Sql> SELECT * FROM users WHERE id = '42' OR '1'='1';
```

The condition '1=1' is always true, causing the query to return all data fields for all users. This is a classic example of a SQL injection. Importantly, it's crucial to understand that attackers are not looking to disrupt the application itself but rather to exploit the existing functionality to access unauthorized data. When developing applications, it's vital to anticipate potential vulnerabilities and implement safeguards accordingly.

4. The Three Main Types of SQL Injections

There are three primary types of SQL injections, each with its unique characteristics.

1.In-band SQLi

- In-band SQLi occurs when an attacker initiates the SQL injection from the same location used to collect the output. This type of attack can be categorized further into two subtypes: Error-based SQLi and Union-based SQLi.
- Error-based SQLi: This type of SQL injection capitalizes on error messages produced by the database. Attackers use these error messages to gather information about the database's structure and elements before proceeding to more complex attacks.
- Union-based SQLi: Union-based SQLi employs the UNION operator to append additional data to a single result, often adding data to the visible table in the web application. Executing a Union-based SQLi successfully requires the attacker to possess key information about the database, including the table name, the number of columns in the query, and the data types. Information collected through Error-based SQLi can often be used in Union-based SQLi attacks.

2. Blind SQLi

- Blind SQLi is a variant of In-band SQLi, with one crucial difference: responses from the web application do not reveal the results of the query or any database errors. Blind SQLi can manifest in two forms: Content-based Blind SQLi and Time-based Blind SQLi.
- Content-based Blind SQLi: In this approach, attackers utilize queries that prompt conditional responses rather than direct data outputs. These SQL queries ask the database true or false questions, allowing attackers to assess the responses and determine whether the application is vulnerable. This process can be quite tedious, which is why attackers sometimes resort to automation.
- Time-based Blind SQLi: Time-based Blind SQLi leverages time-intensive operations in the system. By causing the database to delay its response, attackers can gauge the vulnerability of the system.

3. Out-of-Band SQLi

- Out-of-Band SQLi differs from In-band SQLi. In this case, attackers initiate the SQL injection from a different channel than the one used for gathering output. This method manipulates the database server to deliver data to an external server controlled by the attacker. It exploits features like Microsoft SQL Server's xp_dirtree command to achieve this objective.
- It's important to note that Out-of-Band SQLi attacks are relatively less common than other types, often contingent on the specific features enabled on the database server.

5. Technical Research Project:

This project consists of a web application with a login page `MainPage.html` where users can enter a password. The server `script.js` handles user input, retrieves passwords from an SQLite database, and simulates login attempts. The database `test.sl3` stores passwords, and a `success.html` page can be displayed on successful login.

➠ success.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Success</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      background-color: #4CAF50;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
      margin: 0;
    }
    .success-message {
      background-color: #fff;
      padding: 20px;
      border-radius: 5px;
      box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.2);
      text-align: center;
    }
    h1 {
      margin: 0;
      color: #333;
    }
  </style>
</head>
<body>
  <div class="success-message">
    <h1>Login Successful!</h1>
  </div>
</body>
</html>
```

➠ *MainPage.html*

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Login Page</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            background-color: #f1f1f1;
            display: flex;
            justify-content: center;
            align-items: center;
            height: 100vh;
            margin: 0;
        }

        .login-container {
            background-color: #fff;
            padding: 20px;
            border-radius: 5px;
            box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.2);
            text-align: center;
        }

        h1 {
            margin: 0;
            color: #333;
        }

        input[type="text"] {
            padding: 10px;
            margin: 10px 0;
            border: 1px solid #ccc;
            border-radius: 3px;
            width: 100px;
            font-size: 16px;
        }

        button {
            background-color: #007BFF;
            color: #fff;
            border: none;
            border-radius: 3px;
            padding: 10px 20px;
            font-size: 16px;
            cursor: pointer;
        }

        button:hover {
            background-color: #0056b3;
        }
    </style>
</head>

<body>
    <div class="login-container">
        <h1>Login</h1>
        <input type="text" id="passwordInput" placeholder="Enter
password">
        <br>
        <button onclick="login()">Submit</button>
        <p id="message" style="color: red;"></p>
    </div>
    <script>
        let bool1 = 0;
        window.addEventListener('load', function () {
            fetch('/getBool1?query=4421')
                .then(response => response.json())
                .then(data => {
                    bool1 = data.bool1;
                })
                .catch(error => {
                    console.error("Error fetching bool1:", error);
                });
        });

        function login() {
            let password = document.getElementById("passwordInput").value;
            if (!isNaN(password)) {
                if (password === bool1) { //Password
                    window.location.href = `http://localhost:8080/success.html`;
                    console.log("SUCCESS");
                } else {
                    document.getElementById("message").textContent =
"Incorrect password. Try again.";
                }
            } else {
                document.getElementById("message").textContent = "Please
enter a valid number.";
            }
        }

        let count = 1;
        let interval = setInterval(() => {
            if (count < 10000) {
                let passwordInput =
document.getElementById("passwordInput");
                passwordInput.value = '';
                passwordInput.value = count;
                let loginButton =
Array.from(document.querySelectorAll('button')).find(btn =>
btn.textContent === 'Submit');
                if (loginButton) {
                    loginButton.click();
                }
                count++;
            } else {
                clearInterval(interval);
            }
        }, 5);
    </script>
</body>

</html>
```

➠ *test.sl3*

```
sqlite> CREATE TABLE password (passkey VARCHAR(5));
sqlite> INSERT INTO password VALUES (4421);
sqlite> SELECT * FROM password;
4421
```

➡ ***script.js***

```
const http = require('http');
const fs = require('fs');
const sqlite3 = require('sqlite3');
const url = require('url');
const PORT = 8080;

// Define a function to get the password and call the callback function
function getPass(query, callback) {
   const sqlQuery = `SELECT passkey FROM password WHERE passkey
= '${query}'`;

   const db = new sqlite3.Database('./test.sl3',
sqlite3.OPEN_READWRITE, (err) => {
      if (err) {
         console.error('Database error:', err.message);
         return callback(err, null);
      }

      db.get(sqlQuery, function (err, row) {
         if (err) {
            console.error('Database error:', err.message);
            db.close();
            callback(err, null);
         } else {
            const bool1 = row ? row.passkey : null;
            db.close();
            callback(null, bool1);
         }
      });
   });
}

// Define a function to serve HTML files
function serveHTML(filename, response) {
   fs.readFile(filename, 'utf8', (err, data) => {
      if (err) {
         response.writeHead(500, { 'Content-Type': 'text/plain' });
         response.end('Internal Server Error');
      } else {
         response.writeHead(200, { 'Content-Type': 'text/html' });
         response.end(data);
      }
   });
}

// Create an HTTP server
http.createServer((request, response) => {
   const parsedUrl = url.parse(request.url, true);

   if (parsedUrl.pathname === '/getBool1') {
      const query = parsedUrl.query.query;

      if (query) {
         getPass(query, (err, bool1) => {
            if (err) {
               response.writeHead(500, { 'Content-Type': 'text/plain' });
               response.end('Internal Server Error');
            } else {
               response.writeHead(200, { 'Content-Type': 'application/json'
});
               response.end(JSON.stringify({ bool1 }));
            }
         });
      } else {
         response.writeHead(400, { 'Content-Type': 'text/plain' });
         response.end('Query parameter "query" is missing in the URL.');
      }
   } else if (parsedUrl.pathname === '/MainPage.html') {
      serveHTML('./MainPage.html', response);
   } else if (parsedUrl.pathname === '/success.html') {
      serveHTML('./success.html', response);
   } else if (parsedUrl.pathname === '/MainPage_Query.html') {
      serveHTML('./MainPage_Query.html', response);
   } else {
      response.writeHead(404, { 'Content-Type': 'text/plain' });
      response.end('Not Found');
   }
}).listen(PORT);
```

• `success.html` is a simple HTML document designed to display a success message when a user successfully logs in. It features a centered message with a green background, clear "Login Successful!" text, and a clean, minimalistic design. This page is intended to provide a positive user experience after a successful login attempt in the web application.

• `MainPage.html` features a login form centered on the screen. Users enter a password and click "Submit". JavaScript fetches a "bool1" value and checks if the entered password matches it. There's also a countdown loop for testing and automation of a simulated brute force attack. The design includes a white background, rounded corners, and a shadow effect.

• `script.js' is a Node.js script that creates an HTTP server to serve HTML files and handle requests for a specific endpoint that retrieves a password (bool1) from an SQLite database. Here's a brief overview:

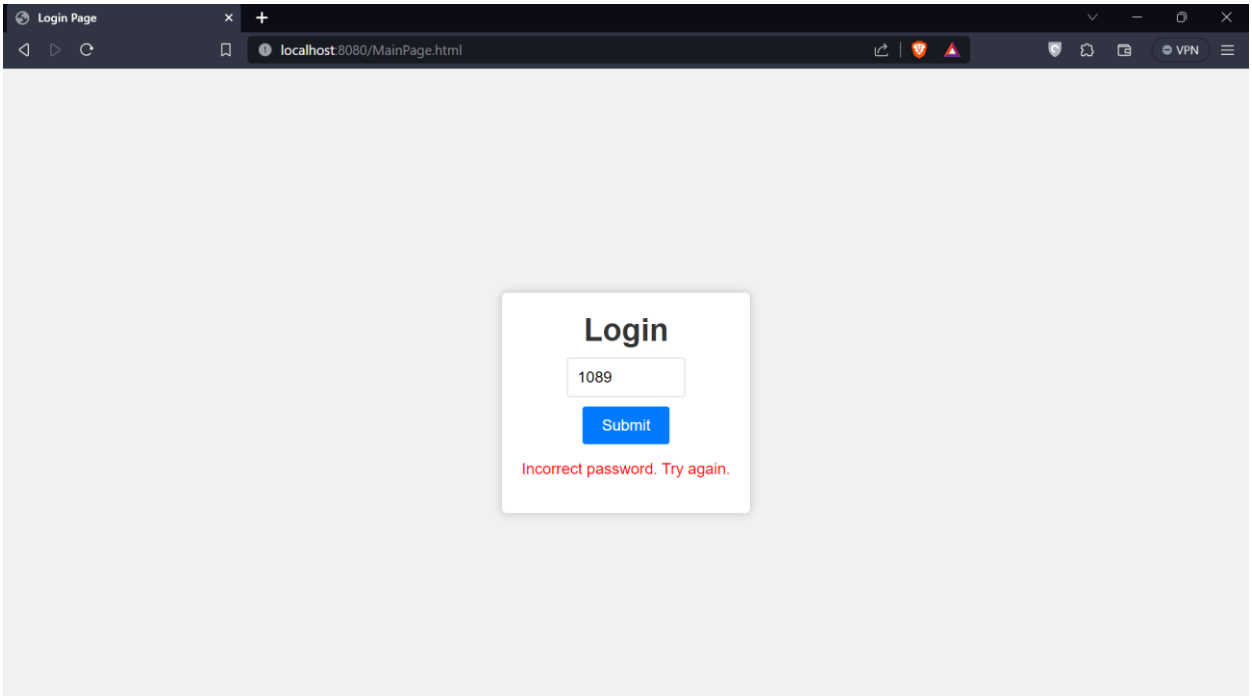
Dependencies: The script requires Node.js modules, including 'http', 'fs', 'sqlite3' and 'url'.
Port Configuration: The server is set to listen on port 8080.
getPass Function: This function queries an SQLite database for a password (bool1) based on a provided parameter. It uses a SQL query to retrieve the password and passes it to a callback function.
serveHTML Function: This function reads and serves HTML files in response to client requests. It reads the specified file and sends it as a response.
HTTP Server: An HTTP server is created using http.createServer. It listens for incoming requests and routes them to different endpoints based on the URL path.
Request Handling:

➢ If the path is '/getBool1', it extracts the 'query' parameter and calls the getPass function to retrieve the password. The result is sent as a JSON response.

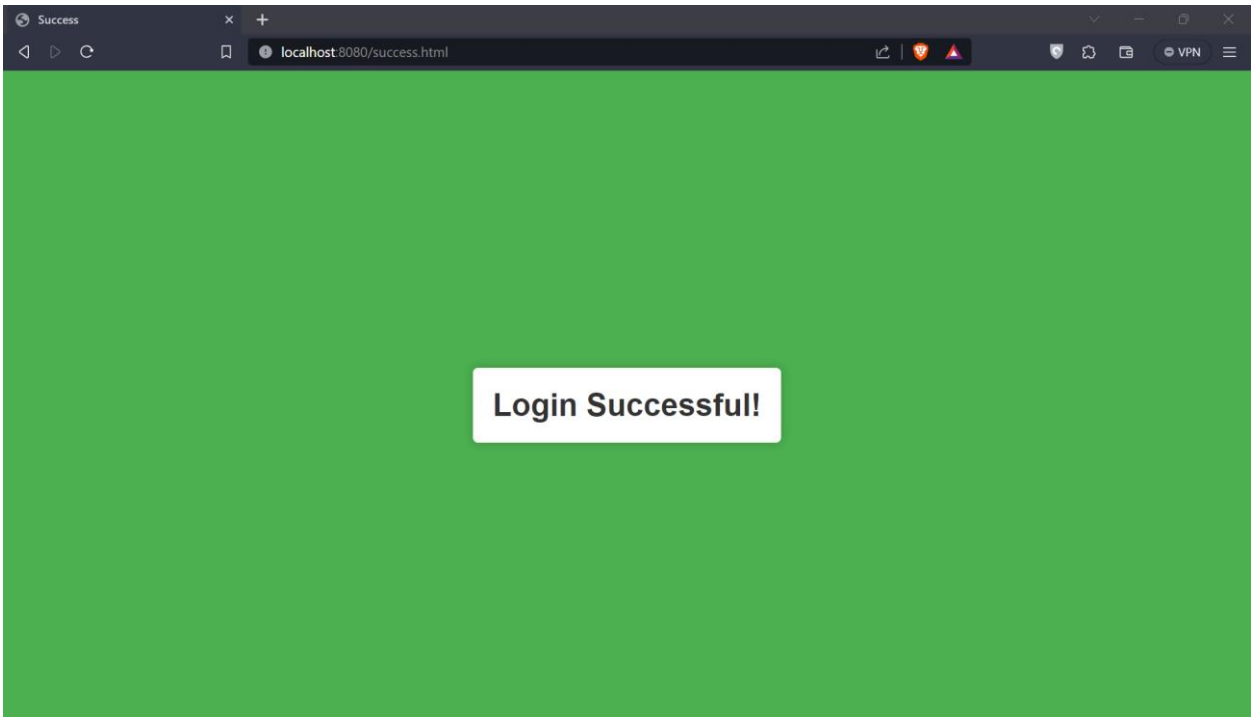
➢ If the path corresponds to specific HTML files ('/MainPage.html', '/success.html'), it uses the serveHTML function to serve the corresponding HTML file.

➢ If the path doesn't match any known endpoints, it returns a 'Not Found' response with a 404 status.

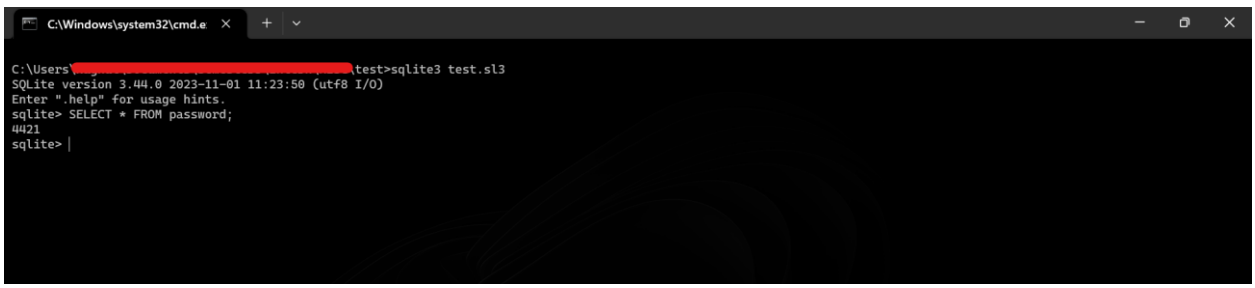
6. Outcomes of the Research:



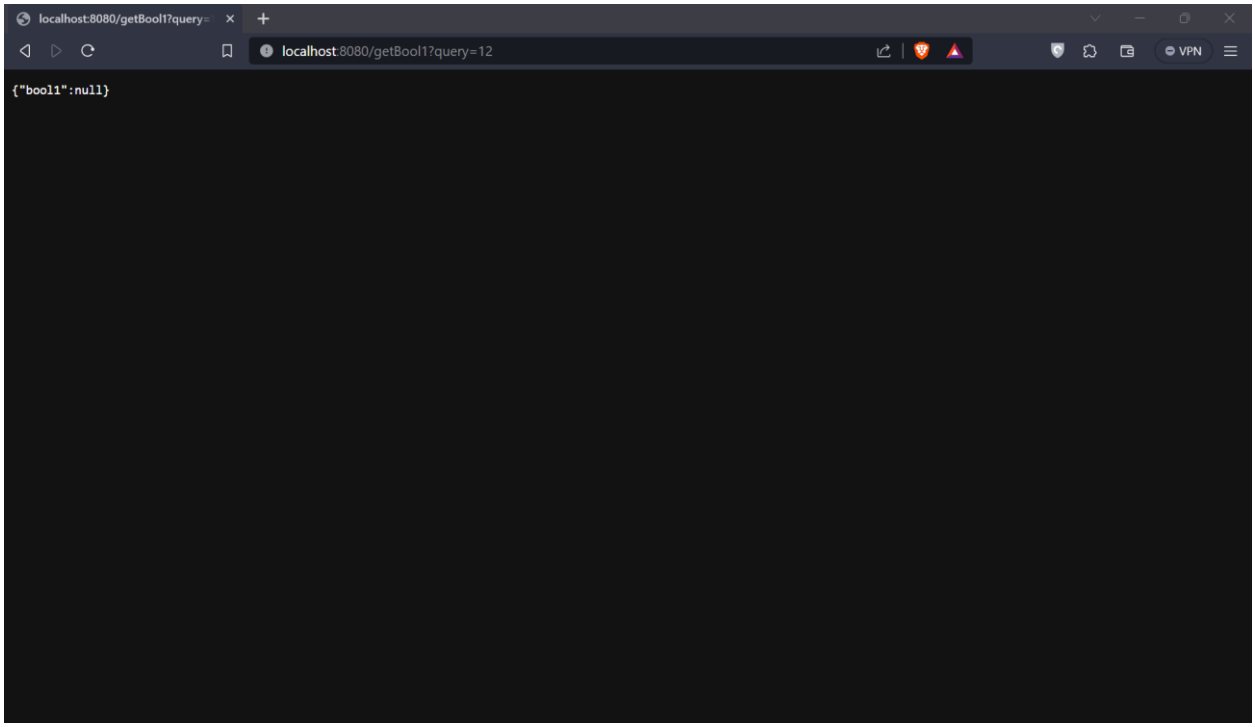
The image depicts `MainPage.html` being served by `script.js` in PORT 8080, when an incorrect password is tried.



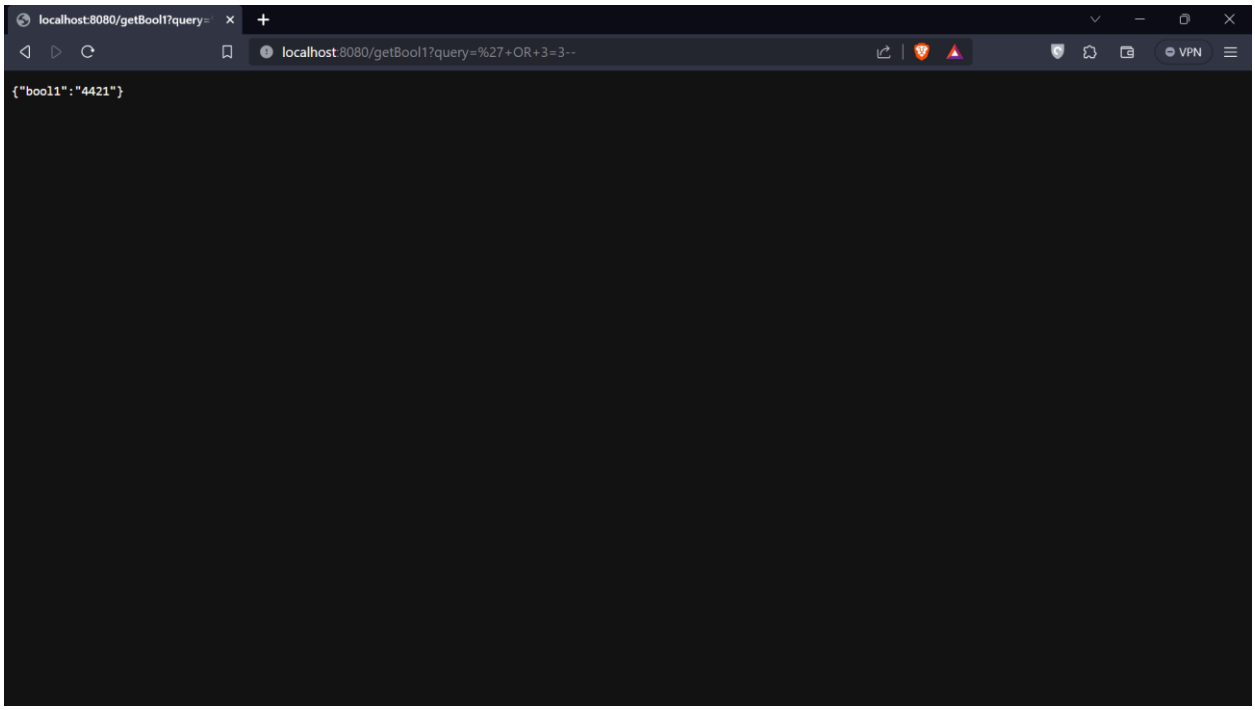
The image depicts `success.html` being served by `script.js` in PORT 8080, after a successful login attempt.



The image depicts the table of passwords stored in the sqlite database `test.sl3`.



The image depicts `/getBool1` pathway output when queried with an incorrect password.



The image depicts `/getBool1` pathway during an SQLi attempt to retrieve the password from the database. The attempt makes use of the '3=3' condition being always true to negate the 'WHERE' clause using the 'OR' statement. '- -' is used to comment out the remaining portion of the query.

7. Preventing SQL Injections: Best Practices

Mitigating the risks associated with SQL injections requires the implementation of several security measures:

1. Sanitize User Inputs: Properly sanitizing and validating all user inputs is of paramount importance. This process should be conducted on the server-side, not solely on the client-side. Regular expressions can be employed to filter and accept only specific characters or patterns.
2. Follow the Principle of Least Privilege: Limiting user access according to the principle of least privilege is essential. Grant users' access to only the privileges they truly require, reducing the potential impact of misuse.
3. Separate Sensitive and Public Data: Sensitive data should be treated distinctly from public data. Store sensitive data only when absolutely necessary, and ensure it is encrypted. Take extra precautions to safeguard sensitive data and protect your users' privacy.
4. Automate Testing for SQL Injection in the Build Pipeline: Integrating automated security testing into your DevOps pipeline is a critical step to identify and rectify vulnerabilities, such as SQL injections, before they can reach the production environment. Several tools, including StackHawk, are available to assist engineering teams in this endeavour.

8. Conclusion

SQL injections continue to pose a persistent and evolving threat to web applications. As attackers increasingly exploit this vulnerability, developers and organizations must remain vigilant and take proactive steps to secure their applications. By adhering to best practices, thoroughly sanitizing user inputs, and implementing automated security testing, businesses can fortify their applications and protect their data from potential breaches.

9. References

- https://www.w3schools.com/
- https://medium.com/@codesprintpro/getting-started-sqlite3-with-nodejs-8ef387ad31c4
- https://medium.com/@adnanrahic/hello-world-app-with-node-js-and-express-c1eb7cfa8a30
- https://www.freecodecamp.org/news/module-exports-how-to-export-in-node-js-and-javascript/