SQL Injection and its Prevention

Diana Jean Tuquib
(tuqu0002@algonquinlive.com)

Zixuan Lou
(lou00019@algonquinlive.com)

Soha Alsafadi
(alsa0231@algonquinlive.com)

Wai Chun (Daniel) Kwan
(kwan0042@algonquinlive.com)

Table of Contents

Introduction

Topic Description

SQL Injection is a cyber-attack that targets web applications that rely on SQL databases. This attack involves inserting malicious code into a SQL statement [1]through an input field on a website. The attacker's goal is to manipulate or access sensitive data that the application has access to. SQL Injection attacks can have severe consequences, from data breaches and loss of sensitive information to complete system compromise.

Preventing SQL Injection attacks is crucial for any web application that uses SQL databases for data storage. This involves implementing security measures such as input validation and parameterized queries to ensure that all user input is properly sanitized and validated before being passed to the SQL engine. Regular security audits and vulnerability assessments should also be conducted to identify and mitigate any potential weaknesses in the application's security defences.

Our report will provide a comprehensive overview of SQL Injection attacks and their potential impact. We will also discuss various techniques and best practices for preventing and mitigating these types of attacks. We will explore some of the common tools and techniques used by attackers and provide practical examples and code snippets to help developers better understand and implement effective SQL Injection prevention strategies. By following these best practices, web application developers can protect their users' sensitive data from SQL Injection attacks.

Reason for Choosing SQL Injection and its Prevention

We have chosen "SQL Injection and its Prevention" as the topic of this project because SQL Injection attacks are a prevalent type of cyber-attack that can have severe consequences for individuals and organizations. As more and more applications rely on SQL databases for data storage, the risk of SQL Injection attacks has grown.

Therefore, it is important for developers to understand the risks associated with SQL Injection attacks and know how to prevent them. By educating ourselves on SQL Injection attacks and their prevention, we can better secure our web applications and protect our users' sensitive information.

Furthermore, understanding SQL Injection attacks and their prevention is essential for anyone interested in cybersecurity or web application development. It is a fundamental component of secure coding practices, and knowledge of SQL Injection attacks and prevention techniques can help developers build more secure applications.
Overall, the topic of SQL Injection attacks and prevention is both important and relevant in today's technology landscape, making it an excellent choice for this project.

Problem Description

Who: Who cares and who does it affect

With the digital world getting bigger each day and more people using the internet, more people are at risk of cyber-attacks. SQL Injection concerns everyone who uses any web or mobile application that involves databases, including users, web developers, database administrators, and other stakeholders. Targets of these attacks include financial institutions, database content management systems, e-commerce platforms, and many more.

SQL injection attacks allow attackers to spoof identity, tamper with existing data, cause repudiation issues such as voiding transactions or changing balances, allow the complete disclosure of all data on the system, destroy the data, or make it otherwise unavailable, and become administrators of the database server.

SQL Injection is very common with PHP and ASP applications due to the prevalence of older functional interfaces. Due to the nature of programmatic interfaces available, J2EE and ASP.NET applications are less likely to have easily exploited SQL injections.

The severity of SQL Injection attacks is limited by the attacker's skill and imagination, and to a lesser extent, defence in depth countermeasures, such as low privilege connections to the database server and so on. In general, consider SQL Injection a high impact severity. [8]

What: What is it

SQL Injection is a cyber-attack that exploits vulnerabilities in the input fields of applications. Attackers inject SQL statements to view, create, delete, or modify databases. After a successful attack, they can have access to confidential and sensitive information and use it for other purposes other than its intended use.

The Main Consequences

Confidentiality: SQL databases generally hold sensitive data; loss of confidentiality is a frequent problem with SQL Injection vulnerabilities.

Authentication: If poor SQL commands are used to check usernames and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password.

Authorization: If authorization information is held in a SQL database, it may be possible to change this information by successfully exploiting a SQL Injection vulnerability.

Integrity: Just as it may be possible to read sensitive information, it is also possible to make changes or delete it with a SQL Injection attack.

Types of SQL Injection

1. In-band SQL Injection

In-band SQL injection is the most common type of attack. With this type of SQL injection attack, a malicious user uses the same communication channel for the attack and to gather results. The following techniques are the most common types of in-band SQL injection attacks:

- **Error-based SQL injection**: With this technique, attackers gain information about the database structure when they use a SQL command to generate an error message from the database server. Error messages are useful when developing a web application or web page, but they can be a vulnerability later because they expose information about the database. To prevent this vulnerability, you can disable error messages after a website or application is live.

- **Union-based SQL injection**: With this technique, attackers use the UNION SQL operator to combine multiple select statements and return a single HTTP response. An attacker can use this technique to extract information from the database. This technique is the most common type of SQL injection and requires more security measures to combat than error-based SQL injection.

2. Inferential SQL Injection

Inferential SQL injection is also called **blind SQL injection** because the website database doesn't transfer data to the attacker like with in-band SQL injection. Instead, a malicious user can learn about the structure of the server by sending data payloads and observing the response. Inferential SQL injection attacks are less common than in-band SQL injection attacks because they can take longer to complete. The two types of inferential SQL injection attacks use the following techniques:

- **Boolean injection**: With this technique, attackers send a SQL query to the database and observe the result. Attackers can infer if a result is true or false based on whether the information in the HTTP response was modified.

- **Time-based injection**: With this technique, attackers send a SQL query to the database, making the database wait a specific number of seconds before responding. Attackers can determine if the result is true or false based on the number of seconds that elapses before a response. For example, a hacker could use SQL query that commands a delay if the first letter of the first database's name is A. Then, if the response is delayed, the attacker knows the query is true.

3. Out-of-Band SQL Injection

Out-of-band SQL injection is the least common type of attack. With this type of SQL injection attack, malicious users use a different communication channel for the attack than they use to gather results. Attackers use this method if a server is too slow or unstable to use inferential SQL injection or in-band SQL injection. [10]

What is the impact of a successful SQL injection attack?

A successful SQL injection attack can result in unauthorized access to sensitive data, such as passwords, credit card details, or personal user information. Many high-profile data breaches in recent years have been the result of SQL injection attacks, leading to reputational damage and regulatory fines. In some cases, an attacker can obtain a persistent backdoor into an organization's systems, leading to a long-term compromise that can go unnoticed for an extended period. [9]

How: How to detect SQL injection vulnerabilities

SQL injection can be detected manually by using a systematic set of tests against every entry point in the application. This typically involves:

- Submitting the single quote character ' and looking for errors or other anomalies.
- Submitting some SQL-specific syntax that evaluates the original value of the entry point, and to a different value, and looking for systematic differences in the resulting application responses.
- Submitting Boolean conditions such as OR 1=1 and OR 1=2 and looking for differences in the application's responses.
- Submitting payloads designed to trigger time delays when executed within a SQL query and looking for differences in the time taken to respond.{9}

Where: Where the problem is—client- or server-side

SQL Injection typically occurs on the server side of the web or mobile applications. It arises due to inadequate or lack of input validation and sanitation and due to insecure coding practices. [2]

When: Timing aspects

Injection attacks can happen any time an application is insecure. Attackers attempt to exploit insecure applications. A lot of SQL injection attacks increase when there is a known vulnerability or data breach.

Attackers get smarter and their techniques become more sophisticated. Nowadays, they can create bots that automatically attack each time there is an open window of vulnerability. [3]

Why: Laws, regulations, and other constraints

SQL injections present significant legal and compliance risks. Organizations that do not secure their applications properly may face major consequences. It usually results in financial losses, legal issues, and damage to their reputation.

The government of Canada has provided a guideline on things to consider for websites. It includes information on how to prevent SQL Injection attacks. [4]

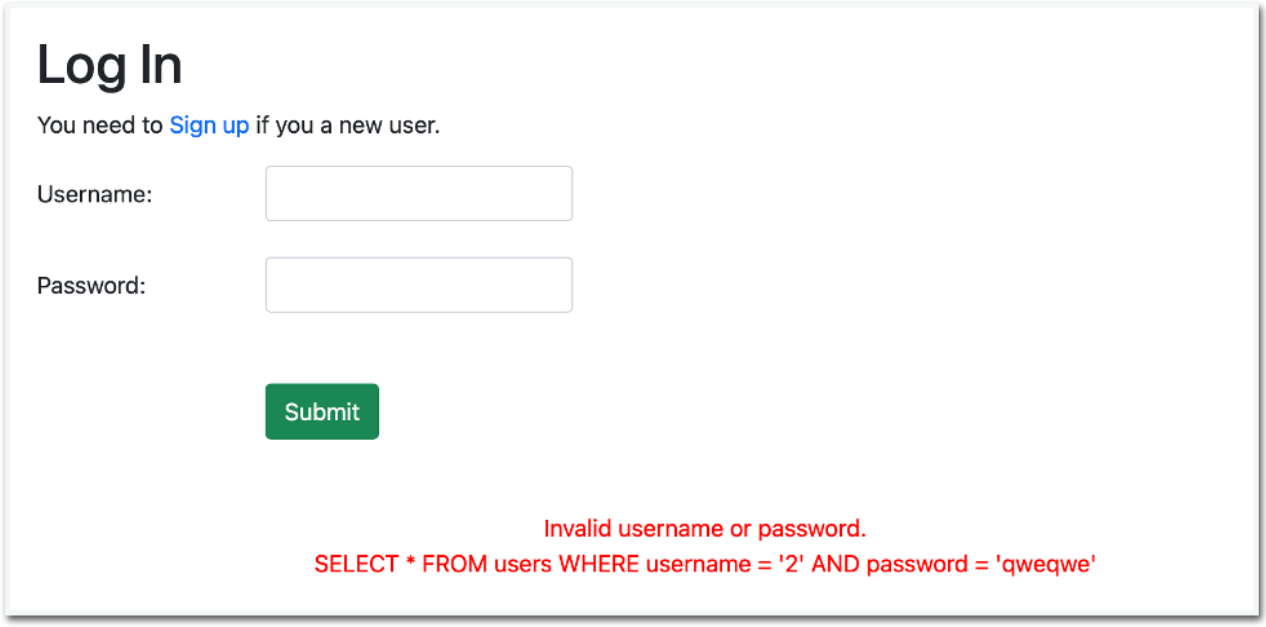
How: How issues happen and get solved

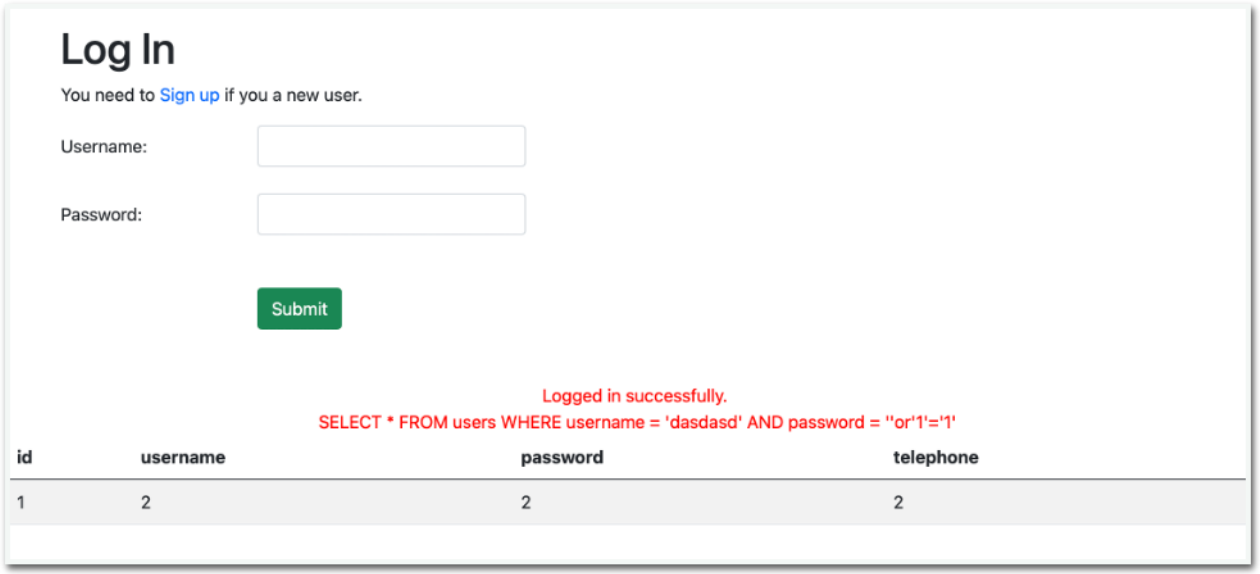
SQL Injection occurs when the application is insecure and fails to properly validate user inputs. Attackers exploit this vulnerability by injecting malicious SQL codes into the input field. To prevent this attack, developers should adhere to industry-standard coding practices and implement secure input fields. [4]

The Canadian Centre for Cyber Security recommends input validation in the areas of the application, including user browser, web application firewall, web server, application business logic, and database. [4]

How: Illustration of SQL Injection Vulnerability in a PHP Login Page

Our demonstration showcases a PHP login page which suffers from an SQL Injection vulnerability. To display this, we've created a web application that allows users to enter their username and password to log in. The web application connects to a database using the PHP Data Objects (PDO) extension to authenticate the users based on the credentials entered.





The PHP code behind the login functionality of our application looks like this:

```
if ($_SERVER["REQUEST_METHOD"] == "POST") {

    $username = $_POST['username'];
    $upassword = $_POST['password'];

    $sql = "SELECT * FROM users WHERE username = '$username'
AND password = '$upassword'";

    $result = $pdo->query($sql);

    if ($result->rowCount() > 0) {
        $msg = 'Logged in successfully. <br>' ;
        $flag = true;

    } else {
        $msg = 'Invalid username or password.';
    }
}
```

The problem arises from the lack of prepared statements, which allow users to input raw SQL code as a part of their username or password. This can result in a successful SQL Injection attack when a malicious actor inputs a value such as **'or'1'='1**. This makes the SQL query always true, allowing the attacker to bypass the login mechanism, regardless of the actual username or password.

Once logged in, the attacker can view all users' data because our code retrieves and displays the entire user database upon successful login. This showcases another aspect of the SQL Injection vulnerability where not just login, but also information access can be compromised.

```
if ($flag ==true){
    $sql1 = "SELECT * FROM users";
    $result = $pdo->query($sql1);

    if ($result->rowCount() > 0) {
    …
```

This vulnerability results from the lack of input validation and the use of concatenation when creating SQL queries. By exploiting these vulnerabilities, an attacker can execute arbitrary SQL statements, gain unauthorized access to data, or even modify or delete it.

In conclusion, this example emphasizes the importance of ensuring secure coding practices in web applications to prevent SQL Injection attacks. The Canadian Centre for Cyber Security, among other organizations, recommends employing proper input validation and using prepared statements or parametrized queries to prevent such vulnerabilities.

Solution Description and Results

We created two basic web applications highlighting the difference between a secure and an insecure application. One website did not have input validation and the other made use of one SQL prevention technique. During our presentation, we will demonstrate how an insecure web application can damage a database and how a secure web application can prevent it.

Implementing a Defensive Strategy Against SQL Injection in a PHP Login Page

To solve the vulnerability found in the initial PHP code that leaves our application open to SQL Injection attacks, we implement the use of prepared statements for the login functionality.

This simple yet effective coding practice will significantly boost the security of our application and close the avenue that SQL Injection attacks may exploit.

Prepared statements ensure that an SQL query and the data provided to it are treated separately, meaning that the data (such as the user's username and password in our case) can't interfere with the query's structure. Thus, if an attacker tries to input a malicious SQL command instead of a username or password, the command won't be executed because the database management system sees it as a string, not an SQL statement.

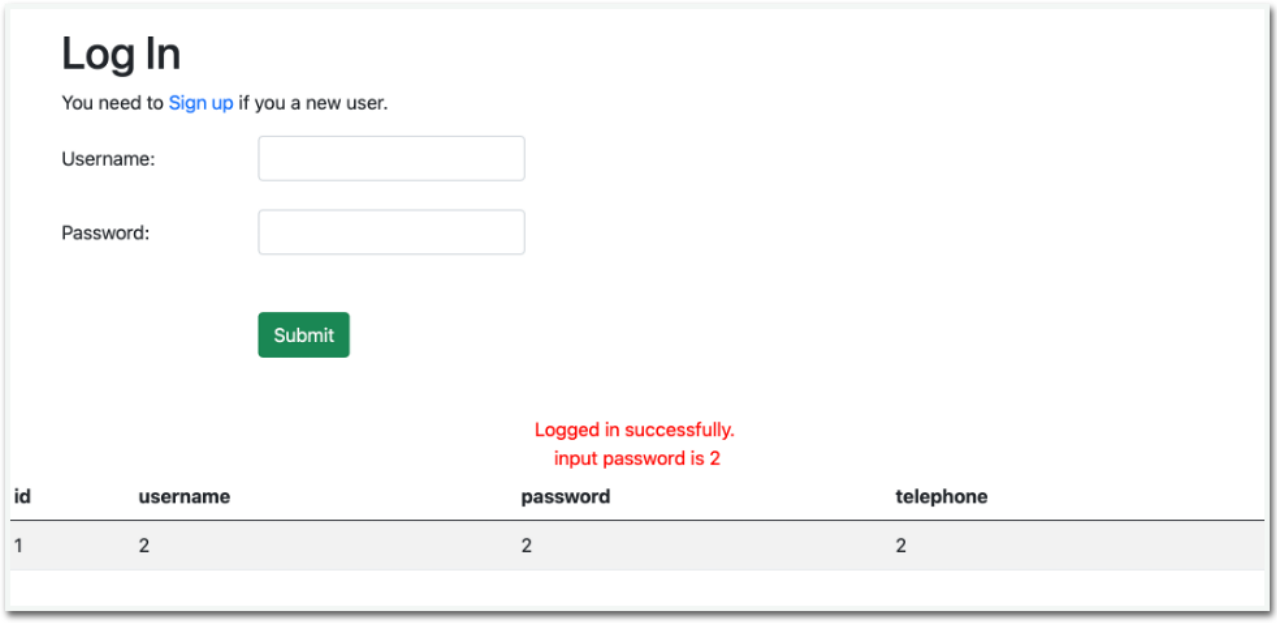
Our solution code applies this principle to the login functionality of the application. Here's the changed PHP code that now includes prepared statements:

```
if ($_SERVER["REQUEST_METHOD"] == "POST") {
    $username = $_POST['username'];
    $upassword = $_POST['password'];

    $sql = "SELECT * FROM users WHERE username = ? AND
password = ?";
    $stmt = $pdo->prepare($sql);

    $stmt->execute([$username, $upassword]);
    $result = $stmt->fetch();

    if ($result) {
        $msg = 'Logged in successfully. <br>' ;
        $flag = true;

    } else {
        $msg = 'Invalid username or password.';
    }
}
```

In the updated code, the SQL command is first prepared, and placeholders ("?") is used for the username and password values. This SQL command is then sent to the database, which analyzes and compiles it, waiting for the actual data to be supplied. The real data is then sent using the 'execute' function, at which point it replaces the placeholders in the already compiled SQL command.

Results

Our improved login page, now featuring prepared statements, maintains the same functionality for legitimate users while eliminating the possibility of SQL Injection attacks.



When an attacker tries to use the previously successful "'or'1'='1" code, the database doesn't interpret it as an SQL command but as a simple string. Thus, the attacker is not able to gain unauthorized access or manipulate the data.

This simple change in our PHP code offers significant protection against SQL Injection attacks, without compromising the usability or the functionality of the application. It is an example of how adopting secure coding practices is an efficient and effective strategy to protect web applications from common vulnerabilities.

It's worth mentioning that while prepared statements go a long way to prevent SQL Injection, it's also good to consider other security practices such as validating and sanitizing all user inputs, limiting database permissions, and using modern ORMs that can handle many database security aspects automatically.

Work Plan

Our work plan for the final report and presentation includes the following component and deliverables:

Component/Deliverable	**Hours per Person (Total)**	**Group Member**
Introduction How: Illustration of SQL Injection Vulnerability in a PHP Login Page Solution Description and Results Demo	approx. 4 hours	Daniel
Problem Description • Who: Who cares and who does it affect • What: What is it • How: How to detect SQL injection vulnerabilities	approx. 4 hours	Diana

• Where: Where the problem is—client- or server-side • When: Timing aspects • Why: Laws, regulations, and other constraints Work Plan		
The Main Consequences Types of SQL Injection What is the impact of a successful SQL Injection attack	approx. 4 hours	Soha
Lessons Learned Conclusion Video Editing	approx. 4 hours	Zixuan

Lessons Learned

Risks that needed to be mitigated

The demo application successfully showed a common risk with web applications: SQL Injection. However, there are several other risks that need to be mitigated.

Risk of not using prepared statements:

The demo app shows an example of not using prepared statements, making the application vulnerable to SQL Injection. Without prepared statements, an attacker can inject malicious SQL commands into user inputs, manipulating the SQL query to gain unauthorized access, modify, or even delete data.

Insecure use of prepared statements:

Even when using prepared statements, a potential risk remains if they are used incorrectly. For example, if user-controlled data is directly included in the query structure rather than as bind parameters, the application might be vulnerable to SQL injection. In our demo app, the correct usage of prepared statements is demonstrated, but it is important to be aware of this potential misuse.

Plaintext password:

The example code used plain text passwords, which lead to a security risk. If anyone gets access to the database, they can read these passwords. A solution is to hash the passwords so we can transform them into a special code that cannot be easily decoded.

Password complexity and account lockouts:
The example code does not mention password complexity or account lockouts after a certain number of failed login attempts. Both help stop attackers from guessing passwords.

Other forms of injection attacks:
Our example focuses on SQL Injection, while there are other similar attacks, like Cross-Site Scripting and Command Injection. These need to be protected against as well.

Conclusion

This project provided a perspective on the importance of web application security. We experimented with a vulnerable PHP login page, which helped us understand firsthand how devastating SQL Injection attacks could be. Manipulating the SQL queries through the user input fields, and witnessing the ease with which an attacker could gain unauthorized access, was a striking revelation.

The process of rectifying this vulnerability was a crucial part of our learning. We rewrote the PHP code using prepared statements, ensuring the SQL queries and user input were handled separately. This modification created a robust defense against SQL Injection attacks.

In conclusion, the practical exercise of creating and subsequently securing a vulnerable application was an invaluable learning experience. It emphasized that security is a fundamental aspect of web application development, not just an optional feature. By transforming the insecure login page into a secured one, we not only enhanced the application's safety but also gained insights into how each line of code could impact the entire application's security.

References

[1] OWASP, "SQL Injection Prevention Cheat Sheet," [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html. [Accessed 23 June 2023].

[2] S. Srivastava, "A Survey On: Attacks due to SQL injection and their prevention method for web application," IJCSIT International Journal of Computer Science and IT, vol. 3, no. 1, 2012.

[3] V. Sundar, "Prevention against Bot-Driven SQL Injection Attacks," Express Computer, 6 December 2022. [Online]. Available: https://www.expresscomputer.in/guest-blogs/prevention-against-bot-driven-sql-injection-attacks/92517/. [Accessed 23 June 2023].

[4] Canadian Centre for Cyber Security, "Security considerations for your website (ITSM.60.005)," 6 October 2021. [Online]. Available: https://www.cyber.gc.ca/en/guidance/security-considerations-your-website-itsm60005. [Accessed 23 June 2023].

[5] , Imperva, "SQL Injection Prevention," [Online]. Available: https://www.imperva.com/learn/application-security/sql-injection-prevention. [Accessed 23 June 2023].

[6] Varonis, "SQL Injection: A beginner's guide to understanding and prevention," [Online]. Available: https://www.varonis.com/blog/sql-injection-prevention/. [Accessed 23 June 2023].

[7] Acunetix, "SQL Injection Prevention Techniques," [Online]. Available: https://www.acunetix.com/websitesecurity/sql-injection-prevention-techniques/. [Accessed 23 June 2023].

[8] OWASP, "SQL Injection ," [Online]. Available: https://owasp.org/www-community/attacks/SQL_Injection [Accessed 23 June 2023].

[9] Portswigger "What is the impact of a successful SQL injection attack?" https://portswigger.net/web-security/sql-injection

[10] Crowdstrike " Types of SQL Injection" https://www.crowdstrike.com/cybersecurity-101/sql-injection/