Lab 7

Understanding and Preventing SQL Injection Attacks

Image Generated by Dalle-E 2 from chat GPT

Prepared for

Kokub Sultan

Prepared by Group 6

Valentine Jingwa, Jean-Pierre Nde-Forgwang

Security System, Cohort E

School of Advance Digital Technology

SAIT

24 January 2024

Understanding and Preventing SQL Injection Attacks

SQL Injection

- SQL Injection constitutes a pervasive threat that jeopardizes the integrity of innumerable online entities worldwide. Its notoriety is rooted in the facility to tamper with SQL queries through user contributions, paving the way for illicit access or catastrophic database compromise. The imperative to grasp the nuances of SQL Injection—its inception, progression, and countermeasures—cannot be overstated for the security apparatus of online applications. Through this scholarly inquiry, we unravel the complexities of SQL Injection, presenting a detailed discourse complete with authentic coding exemplars and formidable protective strategies [1].

Preventions

- Use Strong, Unique Passwords
- Be Wary of Phishing Attempts
- Avoid Sharing Sensitive Information Online
- Use Two-Factor Authentication (2FA)
- Update Personal Devices Regularly
- Educate Yourself on Recognizing Secure Websites
- Report Suspicious Activity

Execution of SQL injection

- SQL Injection attacks are performed by manipulating SQL queries through user inputs. This manipulation involves inserting or "injecting" malicious SQL segments into a query. These segments are designed to alter the query's intended function, allowing attackers to execute unauthorized commands.

Example - 1.1

consider a simple SQL query that retrieves user information based on username [2], [3]

```
1    SELECT * FROM users WHERE username = '$username';
```

Example - 1.2

An attacker can exploit this by submitting a username input such as admin' --, resulting in the query [2], [3]

```
3
4    SELECT * FROM users WHERE username = 'admin' --';
```

The -- sequence comments out the rest of the query, effectively bypassing any authentication mechanism and granting unauthorized access.

Reasons for Vulnerability

SQL Injection vulnerabilities arise from a lack of strict input validation and improper handling of user inputs. When inputs are directly concatenated into SQL queries, attackers can manipulate these queries by injecting malicious SQL code. This issue is compounded by the following factors: [2]

- *Dynamic SQL query construction*: Building SQL queries dynamically with user inputs without proper sanitization or parameterization.

- *Insufficient input validation*: Failing to rigorously validate, sanitize, and escape - user inputs to ensure they do not contain SQL code.

- *Lack of least privilege*: Operating database connections with privileges that are too broad, allowing injected SQL code to perform destructive actions.

Defence Techniques [3]

- *Prepared Statements and Parameterized Queries*: These techniques involve separating SQL query logic from data. SQL code and data are sent to the database separately, preventing attackers from altering the query's structure with malicious inputs. For example, using prepared statements in PHP

- *Input Validation and Sanitization*: Rigorously validate user inputs to ensure they meet the expected format, length, and type. Sanitize inputs by escaping special characters that could be interpreted as SQL code.

Practical Recommendations

Example - 2.1

```
$query = "SELECT * FROM users WHERE email = '" . $_POST['email'] . "'";
```

This line of code is constructing an SQL query by directly concatenating user input ($_POST['email']) into the query string. Here's what's happening step-by-step: [4], [5]

- *Direct Concatenation*: The user's input from a form (the email address in this case) is directly appended to the SQL query string. This is akin to asking a stranger to add an ingredient to a recipe you're following without checking what they're adding. If the stranger adds something harmful, it could ruin the entire dish (or in this case, the security of the database).

- *Vulnerability to SQL Injection*: By directly including user input in the SQL query, an attacker can manipulate the input to alter the query. For example, an attacker could input anything' OR 'x'='x, which would change the query logic to return all users, bypassing any intended restrictions. It's like leaving the door to your house unlocked with a sign that says "come on in," allowing anyone (including burglars) to enter.

Example - 2.2

```
$stmt = $db->prepare("SELECT * FROM users WHERE email = ?");
$stmt->bind_param("s", $_POST['email']);
$stmt->execute();
```

This approach uses a prepared statement, which significantly enhances security by separating the query structure from the data. Here's the breakdown: [4], [5]

- *Preparation*: The prepare method creates an SQL query template, where ? acts as a placeholder for the user input. This is like setting up a security checkpoint at the entrance of a building, where visitors (in this case, user inputs) can only enter through a controlled access point.

- *Binding Parameters*: The bind_param function specifies the type of data expected (s stands for string) and assigns the actual input value to the placeholder. This ensures that the input is treated strictly as data, not part of the SQL command. It's akin to a security checkpoint where the guard checks the visitor's ID and purpose before allowing them inside, ensuring they can't just roam freely.

- *Execution*: Finally, the execute method runs the query with the safely included user input. This ensures that the database performs the intended action without being misled by potentially malicious input. It's like having a trusted employee escort the visitor to their meeting room, ensuring they go where they're supposed to without wandering into restricted areas.

Reference:

[1] PortSwigger. "SQL Injection." [Online]. Available: https://portswigger.net/web-security/sql-injection. [Accessed: Mar. 1, 2024].

[2] SQLShack. "SQL Injection: What Is It, Causes, and Exploits." [Online]. Available: https://www.sqlshack.com/sql-injection-what-is-it-causes-and-exploits/. [Accessed: Mar. 1, 2024].

[3] PT Security. "How to Prevent SQL Injection Attacks." [Online]. Available: https://www.ptsecurity.com/ww-en/analytics/knowledge-base/how-to-prevent-sql-injection-attacks/. [Accessed: Mar. 1, 2024].

[4] Tutorial Republic. "PHP MySQL Prepared Statements." [Online]. Available: https://www.tutorialrepublic.com/php-tutorial/php-mysql-prepared-statements.php. [Accessed: Mar. 1, 2024].

[5] W3Schools. "PHP MySQL Prepared Statements." [Online]. Available: https://www.w3schools.com/php/php_mysql_prepared_statements.asp. [Accessed: Mar. 1, 2024].