SQL Injection Vulnerability in the SecureBank Login System

This document details a SQL Injection vulnerability identified in the initial implementation of the SecureBank login system and the subsequent steps taken to patch this critical security flaw.

Vulnerability Description

Vulnerability Type: SQL Injection

Location: `vulnerable_login.php`

Description: The original login script (`vulnerable_login.php`) was susceptible to SQL injection due to the insecure construction of the SQL query used to authenticate users. The script directly embedded user-provided input (the username) into the SQL query string without proper sanitization or the use of parameterized queries.

Vulnerable Code Snippet:

```
$sql = "SELECT * FROM users WHERE username = '$username' AND password = '" . md5($password) . "'";
$result = $conn->query($sql);
```

In this vulnerable code, the $username variable, which is directly taken from user input, is concatenated into the SQL query. This allows a malicious user to inject arbitrary SQL code by crafting a specific input for the username field.

Exploitation Scenario: An attacker could exploit this vulnerability by providing a specially crafted username, such as:

```
' OR 1=1 --
```

When this input is embedded into the SQL query, the resulting query becomes:

```
SELECT * FROM users WHERE username = '' OR 1=1 -- ' AND password = '...'
```

The -- acts as a comment, effectively removing the password check from the query. The OR 1=1 condition is always true, causing the query to return the first user in the users table, thus bypassing the authentication mechanism and allowing the attacker to log in without knowing a valid password.

Patching the Vulnerability

Patched Code: `secure_login.php`

Patching Method: Prepared Statements with Parameter Binding To remediate the SQL injection vulnerability, the login logic was rewritten to utilize prepared statements with parameter binding. This is a secure way to interact with databases as it separates the SQL query structure from the user-provided data.

Patched Code Snippet:

```
$sql = "SELECT id, username, password FROM users WHERE username = ? AND password = ?";
$stmt = $conn->prepare($sql);

if ($stmt) {
    $stmt->bind_param("ss", $username, $hashedPassword);
    $stmt->execute();
    $result = $stmt->get_result();

    // ... (rest of the result handling)
    $stmt->close();
} else {
    die("Error preparing SQL statement: " . $conn->error);
}
```

Patch Explanation:

- Prepared Statement: The SQL query is defined with placeholders `(?)` for the user-provided username and the MD5 hash of the password.
- `$conn->prepare($sql)`: This function prepares the SQL statement for execution by the database server. The structure of the query is sent to the database, and the database knows to expect parameters in place of the ? placeholders.
- `$stmt->bind_param("ss", $username, $hashedPassword):` This line binds the actual user-provided `$username` and the `$hashedPassword` to the placeholders in the prepared statement. The `"ss"` indicates that both parameters are expected to be strings. Crucially, the database treats these bound parameters as data values, not as executable SQL code.
- `$stmt->execute():` This executes the prepared statement with the bound parameters. The database safely substitutes the provided data into the query structure.
- `$stmt->get_result():` This retrieves the result set from the executed query.

How Prepared Statements Prevent SQL Injection

Prepared statements prevent SQL injection because the user-provided data is never directly embedded into the SQL query string. Instead, the query structure and the data are sent to the database separately. The database then combines them in a safe manner, ensuring that any malicious SQL code within the user input is treated as literal data and not as part of the SQL command. This separation effectively neutralizes the attack vector for SQL injection.

By switching from direct query execution to prepared statements with parameter binding, the SecureBank login system is now protected against the SQL injection vulnerability that was present in the original implementation.