Secure Computing Assignment 2

ELEN4024A

Author: 2344104   *Date: 22 May 2023*

Analysing Security Vulnerabilities

In order to find the vulnerability in the web application, the following experiment was undertaken:

1. **Examination of the database and testing legal input:** Initially, the database was examined, and legal input was tested into the web application, ensuring the expected behaviour with valid credentials and operations.
2. **Testing a variety of SQL injection payloads:** various SQL injection payloads were used to test the vulnerability of the web application. Payloads such as ' OR '1'='1, ' or admin', ' OR 1=1 --', and ' OR '1'='1 were injected to check if they could manipulate the application's behaviour [1] [2].
3. **Recording of Successful SQL injection attempts:** throughout the experiment, successful SQL injection attempts were identified. Namely, injecting ' OR '1'='1 as the password and specifying a patient name as ' OR '1'='1 or ' OR 1=1 --'. This resulted in unauthorized access and exposure of confidential patient data.

After the experiment, the cause of this vulnerability was investigated. By analysing the source code it became evident that the vulnerability in these queries lies in the fact that they directly incorporate user-supplied input into the SQL queries using string concatenation [1]. This makes them susceptible to SQL injection attacks.

For example, in a successful attack scenario found, the injected input for username, password and patient surname parameters is ' OR 1=1 --. The resulting SQL statement would be:

- For Authentication (AUTH_QUERY):

select * from user where username='' OR 1=1 --' and password='' OR 1=1 --'

- For Patient Search (SEARCH_QUERY):

select * from patient where surname like '' OR 1=1 --'

For Authentication (AUTH_QUERY) the injected conditions ' OR 1=1 -- manipulate the SQL command, resulting in an OR condition that is always true (1=1) [1]. The rest of the query after the double dash (--) is treated as a comment and ignored [1]. This bypasses the username and password check, allowing unauthorized access to the system.

Likewise, for Patient Search (SEARCH_QUERY) the injected condition ' OR 1=1 -- affects the WHERE clause of the query [1]. The injected condition utilizing the double dash (--) as a comment bypasses filtering on the surname and retrieves all patient records by creating an OR condition that is always true (1=1) [1]. Thus, the SQL injection exposes all patient records.

Additionally, when examining the source code and connected database, it became apparent that the passwords were stored as raw strings. Storing the passwords as raw strings poses a significant security risk because it leaves the passwords easily readable and accessible if the database is compromised [3]. In the event of a data breach or unauthorized access to the database, the attackers can retrieve the passwords in their original form, allowing them to gain unauthorized access to user accounts (as performed with this SQL attack) [3].

Fixing Security Vulnerabilities

The vulnerability in the source code resulted in the use of string concatenation to capture the inputs from the web application. Therefore, to mitigate this issue the source code in the ‘AppServlet’ source file was updated to include prepared statements. The following changes were made:

1. The use of placeholders, “?”, in the AUTH_QUERY and SEARCH_QUERY variables, which were created through the prepareStatement method, and the use of the setString method to fill values. Therefore, with these changes made, the user-supplied values are treated as data rather than executable SQL code.
2. The authenticated method was updated so the SQL query is executed using a prepared statement, which provides a secure mechanism for executing the query. The username and password values are set using setString on the prepared statement, therefore ensuring proper handling and protection against SQL injection attacks.
3. Likewise, the searchResults method was updated so the SQL query is executed using a prepared statement, and the surname value is set using setString. This change helps prevent SQL injection by treating user input as data, eliminating the possibility of malicious input altering the query structure.

Thereafter, the same initial experiment mentioned was repeated, and the previous successful SQL payloads identified did not expose any confidential patient data.

Therefore, the changes made in the application effectively prevented the SQL injection attacks. By utilizing prepared statements, input values were treated as data parameters, ensuring that user input is interpreted correctly as data and not as executable SQL code [2]. Ultimately, the prepared statements implemented provide a robust security measure by automatically escaping special characters and preventing malicious input from altering the query structure or executing unintended SQL commands, thus safeguarding against SQL injection vulnerabilities [2].

References

[1] PortSwigger, "PortSwigger Ltd," 2023. [Online]. Available: https://portswigger.net/web-security/sql-injection.

[2] OWASP, "SQL Injection Prevention," OWASP, [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html. [Accessed 19 May 2023].

[3] A. Zelinksy, "Web Security: The Importance of Hashing and Salting Passwords," 18 July 2022. [Online]. Available: https://levelup.gitconnected.com/web-security-the-importance-of-hashing-and-salting-passwords-7582f36f9d0e. [Accessed 19 May 2023].