COMP3911 Coursework 2

Group members: Ben White (201414736), Adrian Zahir (201426001), Morgan Lewis (201426038)

Analysis of Flaws

1. The first flaw we found was that the *username*, *password* and *surname* input fields were vulnerable to an SQL injection attack.
2. The second flaw we found was the ability to perform a stored cross-site scripting attack using SQL injection on the input fields.
3. The third flaw found was that the passwords for users were stored as plain text in the database

SQL Injection

After inspecting the source code, we saw that the values from the plain text inputs on the login page were unsanitised and were inserted directly into an SQL query that was executed on the database. Also, the authentication logic was structured in a manner where any outcome generated by the authentication query was considered successful. Thus we concluded that these inputs are susceptible to SQL injection attacks due to a lack of input validation. The server takes each input value and inserts it into a query string between two single quotation marks. A single quote character within the input value acts as a closure for the value string, causing the rest of the input to be interpreted as SQL instead of a regular string. The resulting query is executed on the database. An attacker can craft an input that allows malicious execution on the server.

To exploit this, we can use the input `'name' OR ''=''` on the *username* and *password* input fields to bypass the server's authentication. Injecting this value results in the following query string for the authentication check:

```
SELECT * FROM user WHERE username='name' OR ''='' AND password='name' OR ''=''
```

The input for the username and password contains the `''=''` condition which compares the empty string to itself and will always return true. Therefore, by combining this with an `OR` operator, the attacker can consistently return a value from the authentication query, bypassing the authentication and navigating to the next page.

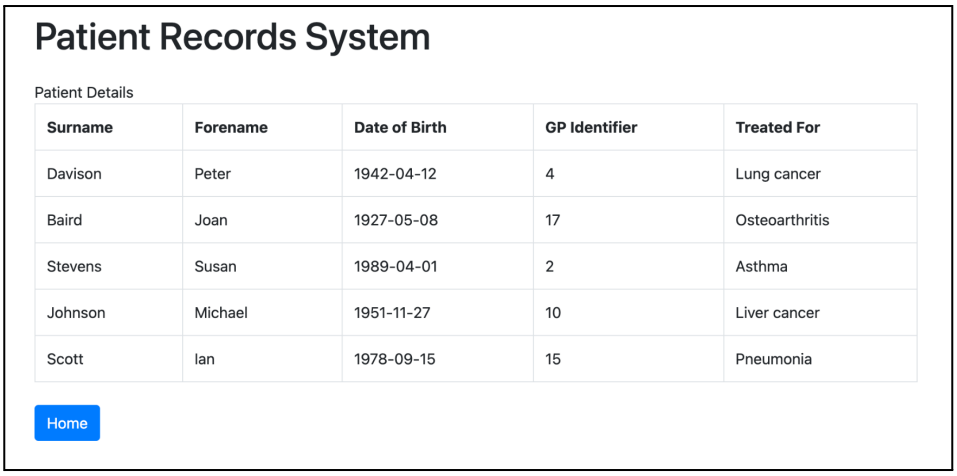

Figure 1: All patient records as a result of an SQL injection attack

We can use the same malicious input for the *Patient Surname* field to return every record of the *patient* table. This results in the following query string for the details page: `SELECT * FROM patient WHERE surname='name' OR ''='' COLLATE nocase`. Using the same logic from above, when executed on the server, all the patient records are returned and displayed on the page. This can be seen in *Figure 1*.

Stored Cross-Site Scripting

On further inspection of the code, we saw that no HTML escaping was performed on the input values. This means that, if the attacker can write to the database, they can perform a stored cross-site scripting (XSS) attack. In this application, malicious code can be written to the database by injecting an SQL *UPDATE* statement as shown above. Using the following string as an input to the *password* field, the attacker can modify the SQL query to create and store a malicious script in the *patient* table (the result of the attack is shown in *Figure 2*):

```
'name' OR ''=''; UPDATE patient SET treated_for='<h1><b><i>Malicious stored XSS attack</i></b></h1>
```

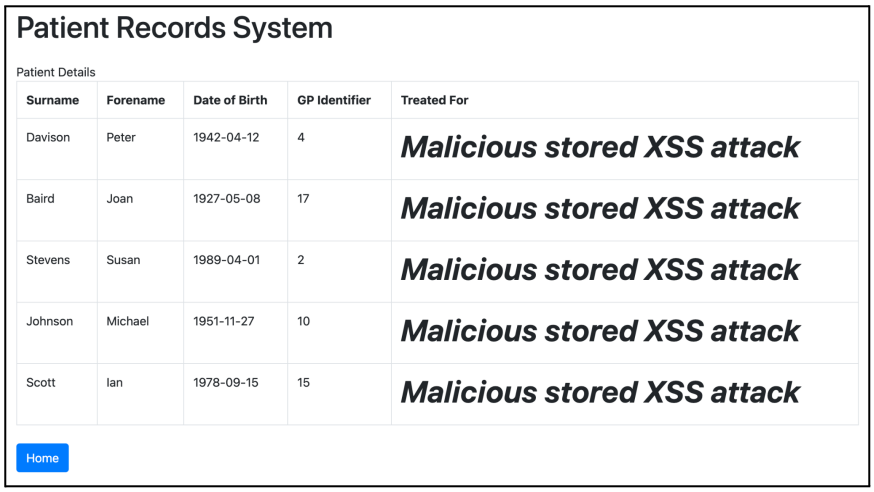

Figure 2: Result of stored cross-site scripting attack

Plaintext Credentials

The usernames and passwords, which make up a user's credentials, are stored as plain text in the database. It is possible for an attacker to gain access to the database by breaching the system on which it is stored. If an attacker gains access to the database then they can log in as any user through the portal using the stolen credentials. This flaw also leaves other systems vulnerable as a lot of people use the same password across multiple systems. An attacker could take advantage of this and use the credentials they obtained from this database to try and gain access to other systems.

This flaw was found through analysis of the database schema which highlighted that the password field was considerably short. We then inspected the actual values (*Figure 3*) and saw the passwords were stored in plain text.

id	name	username	password
1	Nick Efford	nde	wysiwyg0
2	Mary Jones	mjones	marymary
3	Andrew Smith	aps	abcd1234

Figure 3: User table showing plaintext passwords

Fixes Implemented

To fix the SQL injection vulnerability, input validation must be used to sanitise the data. For the Java server, this involved using Java's prepared statements to execute queries when using input from the user. The query strings were modified to include question marks in place of input values to conform to the prepared statement syntax. This prevents any SQL injection attack on the input fields.

To remediate the cross-site scripting attack, the input must be sanitised using HTML encoding. HTML encoding replaces every potentially unsafe character with an escape sequence that forces the browser to treat the character as text, not a special character. This means that the injected HTML is not interpreted as HTML but as plain text. So, if an attacker injects a malicious script, the script text will be shown on the screen, rather than the malicious code being executed on the user's browser. To implement HTML encoding, we sanitised each input by replacing HTML characters (&, <, >, /, \, ‘) with their equivalent escaped value (*&*, *<*, *>*, *"*, *'*, */*).

To remove the threat posed by passwords stored as plain text, we hashed them in the database. Hashing the plain text passwords alone does not sufficiently counter this threat. Hackers can crack passwords by comparing the hash values of popular passwords with the hash values stored in the database. Moreover, in the case where multiple users share the same password, these would all hash to the same value. Attackers with access to the database could spot patterns in the hashes which arise from this and use them to direct their efforts at cracking the passwords. To address this, we generate a random salt for each password before hashing it.

The function *generateSalt creates* a 64-character random string of letters to use as the salt. We only use letters in the salt because some special characters are not supported in SQLite. We use 64 characters because it is harder to crack a longer salt. The SQL script ‘*modifydb.sql*’ adds a new column to the user table to store the salt, which is needed for use during authentication.

We implemented the hashing function *generateHash* which takes a string and returns its SHA-256 hash. We chose not to use an algorithm from the SHA-1 family because they are no longer considered secure.

We hashed all existing plaintext passwords using the function *hasExistingPasswords*. This function gets the plaintext passwords from the database, appends a unique salt (generated by *generateSalt*) to them and hashes them three times (using *generateHash*). We then write the hashed password and salt to the database, overwriting the plaintext password. A simple class to represent users was created to facilitate this.

Finally, the existing authentication function was updated to work with the now hashed passwords. It fetches the user's salt, appends it to the inputted password and hashes it three times before checking the result. If this value matches the hash value stored in the database, the user is authenticated.