COMP3911 - Coursework 2

Authors: Konstantinos Biris [sc19kb], Raulian-Ionut Chiorescu [sc19ric]
Supervisor: Dr. Nick Efford

Analysis of Flaws

List of identified Flaws:

- **Passwords stored as plain text** – Passwords are stored as plain text in the database instead of being hashed.
- **Web authentication can be bypassed** – Web authentication can be bypassed via SQL injection.
- **Information disclosure threat** – All the records in the patients table can be listed via SQL injection.
- **Database stores weak passwords** - Passwords in the database are very weak since they are only 8 characters long, do not contain any special characters/combinations of uppercase and lowercase, making them easily predictable/brute-forceable.
- **The username field is not set to unique** - It allows the creation of multiple entries in the User table with the same username for example multiple: “mjones”.
- **Database doesn’t protect against stored XSS** - inserting a new entry via the sqlite3 CLI such as: “insert into Patient values (10,'Salcedo', 'John','Nice Address', '1981-10-24','18','Glaucoma<script>alert("STORED XSS")</script>'); ”

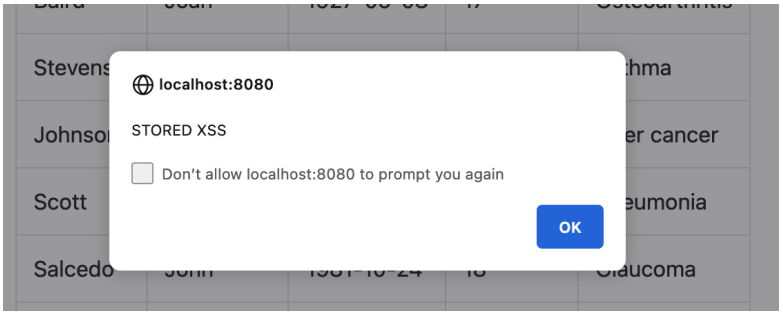

Fig. 1 Example of Stored XSS in the database

Weak passwords stored as plain text

In the original state of the application’s database, it is evident that the passwords are being stored as plaintext. In addition, they are of type “char” and only allow the use of eight characters, while also they do not contain any special characters and the repetition of letters or the use of weak passwords is permitted. This comprises a severe security vulnerability as it allows for the data to be seen by everybody with access to the database, making it possible for individuals with malicious intent to gain access to all the sensitive data in the database. This issue was discovered while investigating the database structure and contents

with the use of “.schema”, “.tables” and “SELECT” statements, when “SELECT * FROM USER;” was inputted the CLI printed all the contents of the User table, there it could be seen that passwords are visible to anyone as they are plaintext.

```
1|Nick Efford|nde|wysiwyg0
2|Mary Jones|mjones|marymary
3|Andrew Smith|aps|abcd1234
```

Fig.2 User table contents, where passwords (4th column) can be seen being stored as plaintext

Web authentication bypass and Information Disclosure

These issues have been discovered when doing basic SQL injection testing. Web authentication can be bypassed via SQL injection, using a statement such as: “ 'or'1'='1 ” in the password field. Using the same command in the surname field, also displays the whole “Patient” table leading to an information disclosure threat. If such a method is used, the username field doesn’t seem to matter, as it was tested with it being blank or over the 12 character “limit” imposed by the database constraints for the username field.

Surname	Forename	Date of Birth	GP Identifier	Treated For
Davison	Peter	1942-04-12	4	Lung cancer
Baird	Joan	1927-05-08	17	Osteoarthritis
Stevens	Susan	1989-04-01	2	Asthma
Johnson	Michael	1951-11-27	10	Liver cancer
Scott	Ian	1978-09-15	15	Pneumonia

Fig. 3 Information disclosure due to SQL injection vulnerability

Fixes Implemented

Weak passwords stored as plaintext FIX

This issue was fixed by storing hashed passwords in the database. Firstly, the User table had to be recreated in order for a salt column to be added. In addition, the password field was updated to allow for more than 8 characters. Then, 3 random salts were generated for the 3 users using “ SecureRandom ” from the java.security package, who were then stored in the database. Consequently, a hash of each password was computed with a “ PBKDF2WithHmacSHA1 “ algorithm which was also manually inserted into the database. Code-wise a new type of query was introduced, named SALT_QUERY, that retrieves the salt attached to that username from the salt field of the table. Then we compute a hash with the user input and the retrieved salt every time a user tries to log in, and check that hash against

the one stored in the database. The code snippets below present how this implementation was performed.

```
private static final String SALT_QUERY = "select salt from user where username=?";
```

```
private byte[] getSalt(String username) throws SQLException{
  try (PreparedStatement stmt = database.prepareStatement(SALT_QUERY)){
    stmt.setString(1,username);
    ResultSet results = stmt.executeQuery();
    byte[] salt = results.getBytes(1);
    return salt;
  }
}
```

```
try (PreparedStatement stmt = database.prepareStatement(AUTH_QUERY)) {
  KeySpec spec = new PBEKeySpec(password.toCharArray(), getSalt(username), 65536, 128);
  SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
  byte[] hash = factory.generateSecret(spec).getEncoded();
  stmt.setString(1,username);
  stmt.setString(2,Base64.getEncoder().encodeToString(hash));
  ResultSet results = stmt.executeQuery();
```

Fig. 4,5,6 Implementation that computes hashes and checks them against the database stored hashes

Bypassing web authentication and information disclosure FIX

The bypassing of web authentication, and information disclosure vulnerabilities were fixed through the use of prepared statements when creating a query. From the 3 figures below, it can be seen how the fixes were implemented for both the authentication query as well as the search query. Now, when using the application, trying the same command injection attack with the use of “ 'or'1'='1 “ results in a failed login attempt as credentials are invalid, or if it’s used in the surname field, it leads to a “No records found” error.

```
private static final String AUTH_QUERY = "select * from user where username=? and password=?";
private static final String SEARCH_QUERY = "select * from patient where surname like ?";
```

```
try (PreparedStatement stmt = database.prepareStatement(AUTH_QUERY)) {
  stmt.setString(1,username);
  stmt.setString(2,password);
  ResultSet results = stmt.executeQuery();
```

```
try (PreparedStatement stmt = database.prepareStatement(SEARCH_QUERY)) {
  stmt.setString(1, surname);
  ResultSet results = stmt.executeQuery();
```

Fig.7,8,9 SQL injection protection implementation through the use of prepared statements