UiA Faculty of Engineering and Science

Assignment 4

Course code:	IKT222
Semester:	Autumn 2025
Number of pages:	28
Name:	Bjørn Sødal

Project Outline

The full code repository and commit history which this report is based upon may be found in my public repository: Repository

Contents

0.1. **HTTP unencrypted packets.** Given that this is supposedly a patient information retrieval application it should only be natural that the internet traffic web packets are encrypted through HTTPS, alas currently the application only runs in HTTP, which unless the application itself starts defining and encrypting these packets then everyone listening on the same network will be able to see the information being sent over the network.

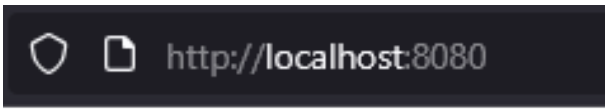


Figure 1. HTTP protocol

As an example I listened on my internet traffic using Wireshark as a secondary computer accesses the running application. In the form sent one can clearly read the form contents being submitted to the application (scale PDF reader 200% for better view):

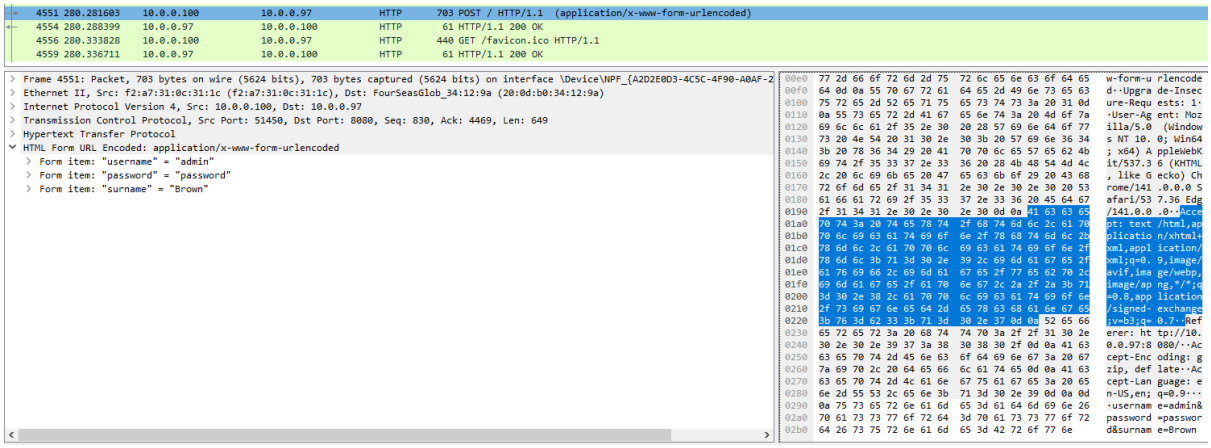


Figure 2. Form contents being submitted readable as plain text in Wireshark

In this specific conversation we can also plainly view the HTML sent from the application within the HTTP response to this very post information:

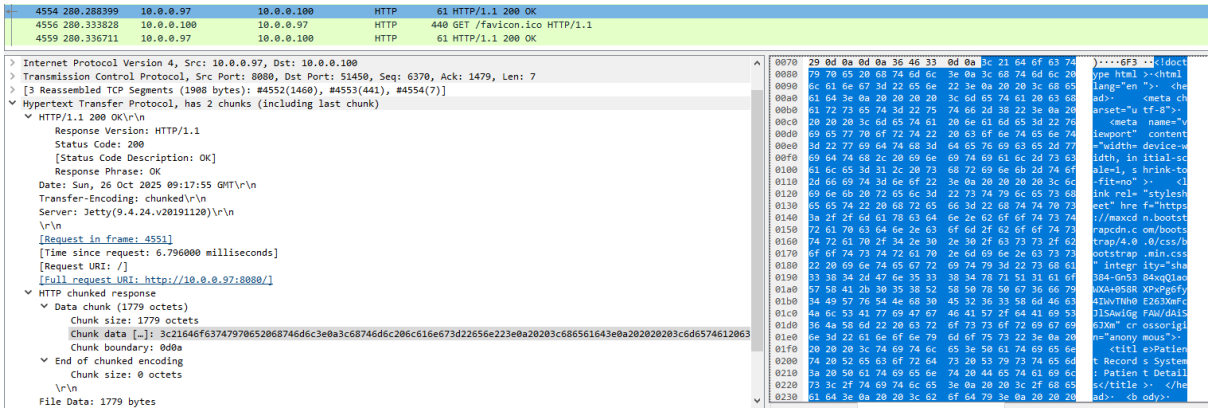


Figure 3. HTML page in response to form submission

Which gives the following information:

```
HTTP/1.1 200 OK
Date: Sun, 26 Oct 2025 09:17:55 GMT
Transfer-Encoding: chunked
Server: Jetty(9.4.24.v20191120)

6F3
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm" crossorigin="anonymous">
    <title>Patient Records System: Patient Details</title>
  </head>
  <body>
    <div class="container">
      <h1 class="mt-2 mb-4">Patient Records System</h1>
           <div class="panel panel-success">
        <div class="panel-heading">Patient Details</div>
        <table class="table table-bordered">
          <tr>
            <th>Surname</th>
            <th>Forename</th>
            <th>Date of Birth</th>
            <th>GP Identifier</th>
            <th>Treated For</th>
          </tr>
          <tr>
            <td>Brown</td>
            <td>James</td>
            <td>2016-04-10</td>
            <td>1</td>
            <td>Asthma</td>
          </tr>
        </table>
      </div>
      <p><a class="mt-2 btn btn-primary" role="button" href="/">Home</a></p>
    </div>
    <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" integrity="sha384-KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG5KkN" crossorigin="anonymous"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min.js" integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q" crossorigin="anonymous"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js" integrity="sha384-JZR6Spejh4U02d8jOt6vLEHfe/JQGiRRSQQxSfFWpi1MquVdAyjUar5+76PVCmYl" crossorigin="anonymous"></script>
  </body>
</html>

0
```

Listing 1. Raw HTML from captured packet

Now any actor listening on this network not only knows of a valid user account (`admin`) and its login credentials (`password`), but also just received potentially confidential patient information (`Brown`) served directly to them by simply listening in on internet traffic moving on the network. Naturally it stands to reason that in a development environment it's easier to test through HTTP, but if this application was an actively running application in production it would at minimum require a TLS/SSL certificate to enable HTTPS.

0.2. **Authentication bypass through SQL injection.** The web interface login form is vulnerable to SQL injections for user accounts:

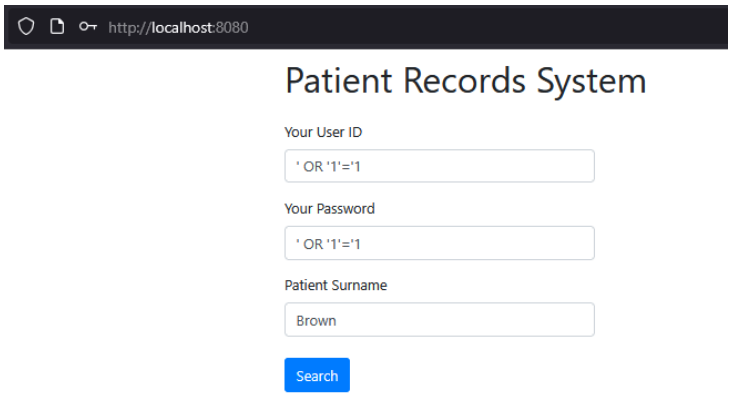


FIGURE 4. SQL injection into user account

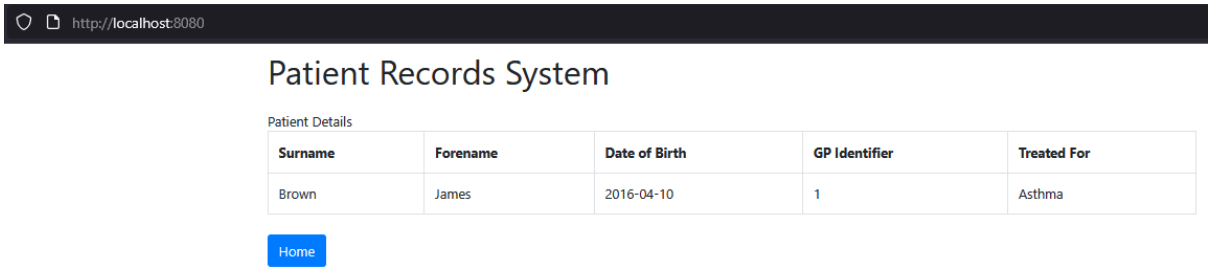


FIGURE 5. User SQL injection result

This will allow anyone to gain access to any known patient surname record by simply circumventing the login requirement of the site. However a malicious actor may additionally circumvent having to know any surname at all as well by simply bypassing the string search of a patient name. This vulnerability was exposed by simply entering a series of standard SQL injection payloads from this GitHub page into the form fields: Payload list [1].

0.3. **Database patient record extraction through SQL injection.** The entire login form is actually subject to SQL injections as it appears that user input is inherently trusted within the application logic and is used as-is, letting actors extract the entire patient records table from the database.

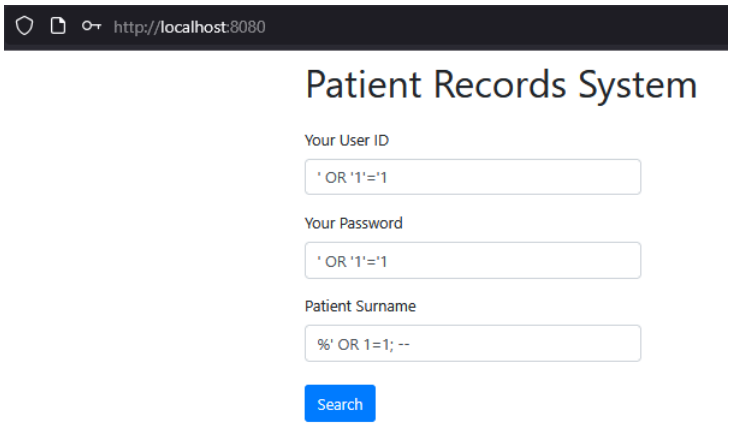
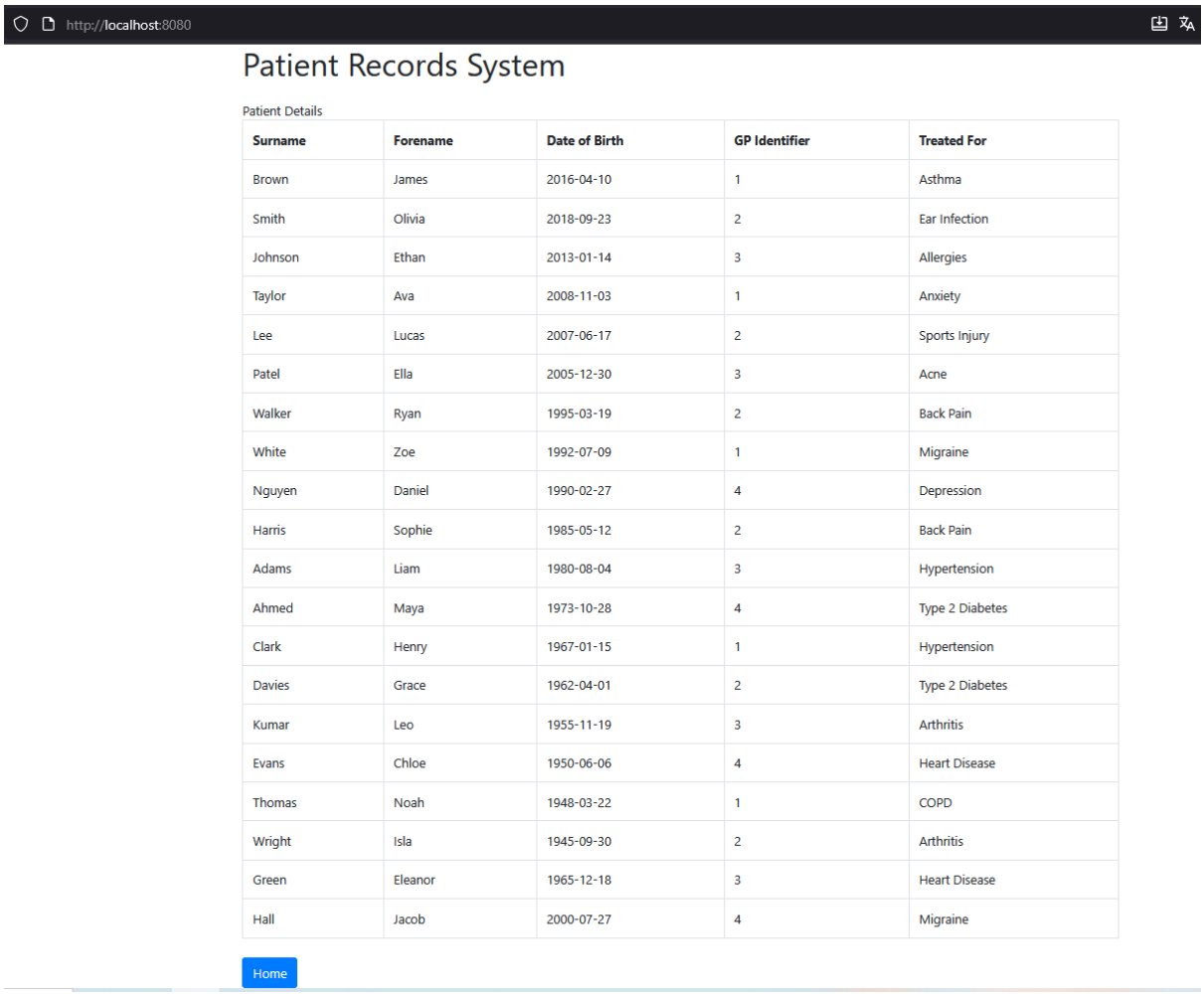


FIGURE 6. SQL injection to list all patient records in database



Surname	Forename	Date of Birth	GP Identifier	Treated For
Brown	James	2016-04-10	1	Asthma
Smith	Olivia	2018-09-23	2	Ear Infection
Johnson	Ethan	2013-01-14	3	Allergies
Taylor	Ava	2008-11-03	1	Anxiety
Lee	Lucas	2007-06-17	2	Sports Injury
Patel	Ella	2005-12-30	3	Acne
Walker	Ryan	1995-03-19	2	Back Pain
White	Zoe	1992-07-09	1	Migraine
Nguyen	Daniel	1990-02-27	4	Depression
Harris	Sophie	1985-05-12	2	Back Pain
Adams	Liam	1980-08-04	3	Hypertension
Ahmed	Maya	1973-10-28	4	Type 2 Diabetes
Clark	Henry	1967-01-15	1	Hypertension
Davies	Grace	1962-04-01	2	Type 2 Diabetes
Kumar	Leo	1955-11-19	3	Arthritis
Evans	Chloe	1950-06-06	4	Heart Disease
Thomas	Noah	1948-03-22	1	COPD
Wright	Isla	1945-09-30	2	Arthritis
Green	Eleanor	1965-12-18	3	Heart Disease
Hall	Jacob	2000-07-27	4	Migraine

FIGURE 7. SQL injection result

This weakness where an unauthenticated user may display the entire patient records table also exposes another vulnerability issue called:

0.4. **Broken access controls.** Because users are never authenticated against some arbitrary clearance level, then any valid login attempt, such as when we inject the statement $'\ OR\ '1$, simply renders out all the patient information (see fig. 7), despite the login ID not matching the general practitioner ID in the patient table.

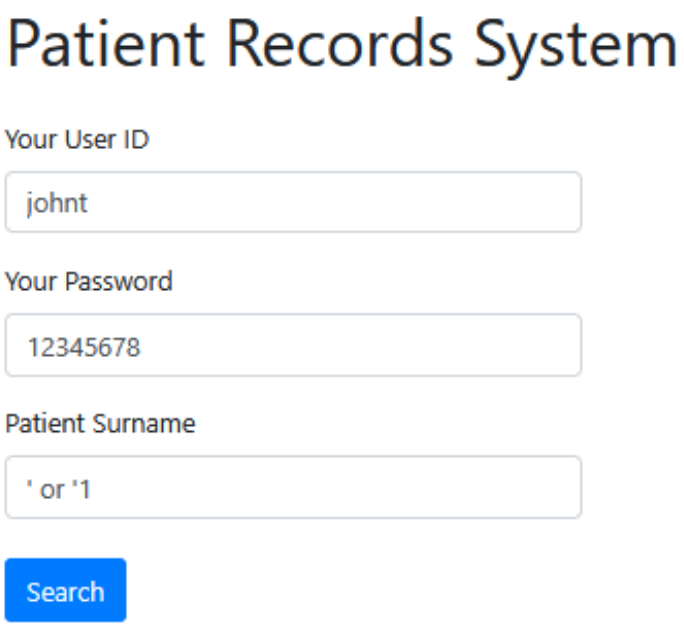


FIGURE 8. Attempting to list out all patient records with non-admin user, result is same as fig. 7

This is a classic example of broken access controls as unauthenticated users should never have the privilege levels required to list out patient records not associated with themselves, or rather at all in the case of authentication bypass.

0.5. **Code.**

0.5.1. *Trusting user input.* When executing a search in the database for account verification the search query makes use of raw and unsanitized user input, letting users inject SQL statements into their input form field:

```
    // Direct SQL queries (vulnerable to SQL injection attacks)
  private static final String CONNECTION_URL = "jdbc:sqlite:db.sqlite3";
  private static final String AUTH_QUERY = "select * from user where username
    ↪ ='%s' and password='%s'";
  private static final String SEARCH_QUERY = "select * from patient where
    ↪ surname like '%s'";

    /* ... lots of code ... */

  @Override
  protected void doPost(HttpServletRequest request, HttpServletResponse
    ↪ response)
      throws ServletException, IOException {
    // Get form parameters
    String username = request.getParameter("username");
    String password = request.getParameter("password");
    String surname = request.getParameter("surname");

    try {
      if (authenticated(username, password)) {
        // Get search results and merge with template
        Map<String, Object> model = new HashMap<>();
        model.put("records", searchResults(surname));
        Template template = fm.getTemplate("details.html");
        template.process(model, response.getWriter());
      } else {
        Template template = fm.getTemplate("invalid.html");
        template.process(null, response.getWriter());
      }
      response.setContentType("text/html");
      response.setStatus(HttpServletResponse.SC_OK);
    } catch (Exception error) {
      response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
    }
  }

  private boolean authenticated(String username, String password) throws
    ↪ SQLException {
    String query = String.format(AUTH_QUERY, username, password); // Directly
      ↪ inputting form input, i.e raw SQL string input
    try (Statement stmt = database.createStatement()) {
      ResultSet results = stmt.executeQuery(query);
      return results.next();
    }
  }

  private List<Record> searchResults(String surname) throws SQLException {
    List<Record> records = new ArrayList<>();
    String query = String.format(SEARCH_QUERY, surname); // Directly inputting
      ↪ surname given in form
    try (Statement stmt = database.createStatement()) {
      ResultSet results = stmt.executeQuery(query);
      while (results.next()) {
        Record rec = new Record();
        rec.setSurname(results.getString(2));
```

```
50          rec.setForename(results.getString(3));
51          rec.setAddress(results.getString(4));
52          rec.setDateOfBirth(results.getString(5));
53          rec.setDoctorId(results.getString(6));
54          rec.setDiagnosis(results.getString(7));
55          records.add(rec);
56        }
57      }
58      return records;
59    }
```

Listing 2. Faulty application logic

`String.format()` with the current SQL queries directly allow for SQL injections to take place, as `username`, `password` and `surname` variables are handled directly in-code as strings and not through an ORM or other sanitization means.

0.5.2. *No proper authentication or authorization methods.* Within the current application there is no session management or verification methods in place, as well as no rate-limiting on password attempts, meaning bad actors may develop a script to continuously attempt to access accounts without any form of restriction in so called brute-force attacks.

A session manager could be implemented to verify current users as well as authorize access to pages. Without a session manager every action on the application will have to require some form of login attempt, and as displayed earlier this verification method is severely flawed with its injection vulnerability.

0.6. **Database.**

0.6.1. *Unencrypted passwords / patient information.* Within the database all the information is available as plain text, meaning in a potential breach any malicious actor would have full access to patient records as well as user account access through plain text passwords and usernames.

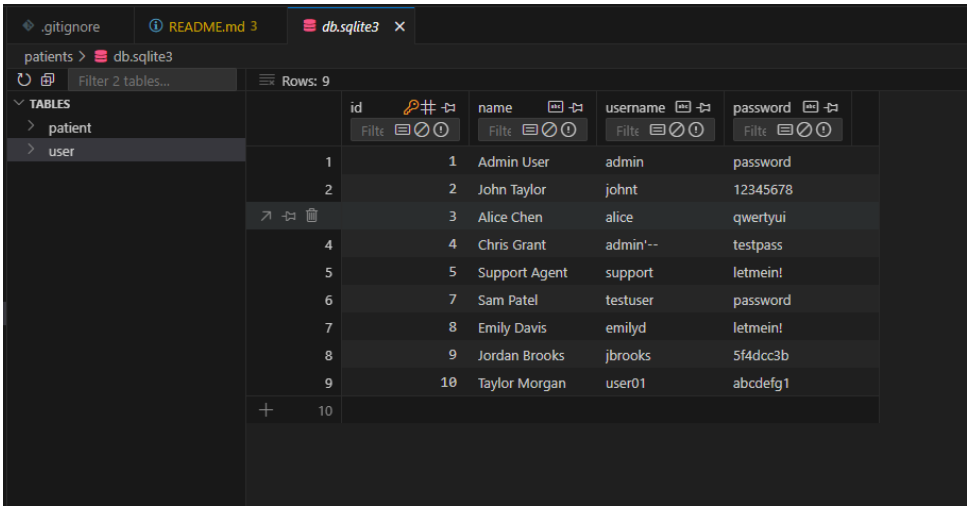


FIGURE 9. Plain text user information and account credentials

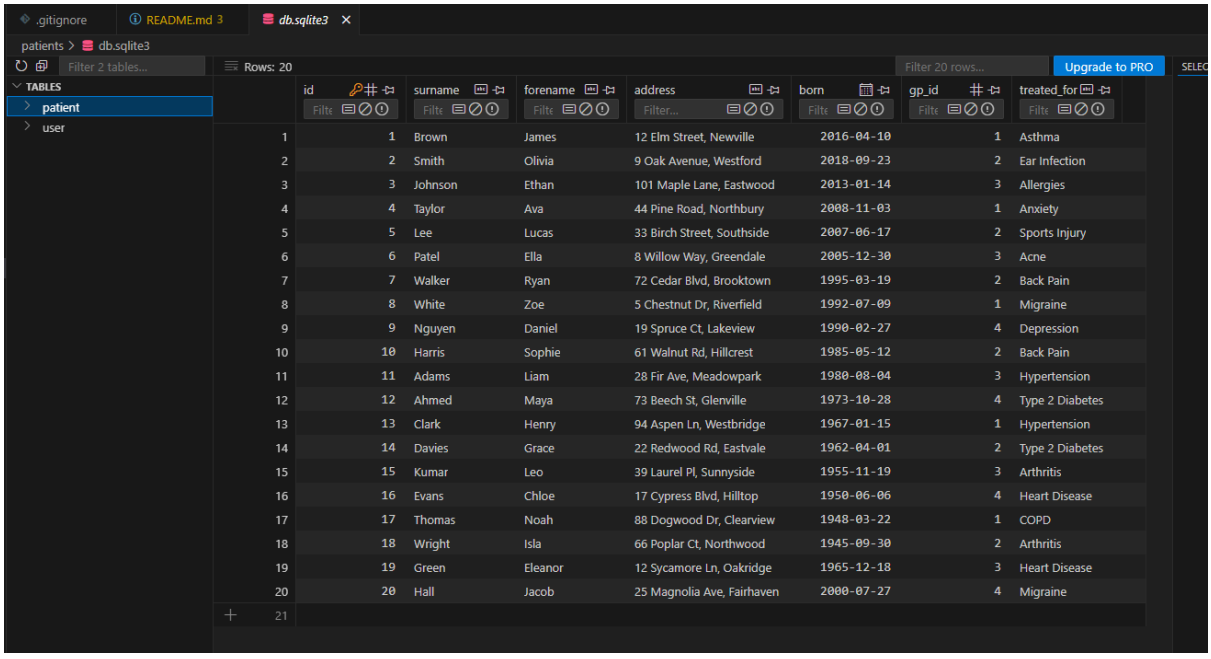


FIGURE 10. Plain text patient information

To at least prevent user account access credentials from being leaked in a potential breach, at minimum the passwords should be encrypted and salted, and ideally all identifiable patient record data should be encrypted to prevent sensitive data from being exploited to prey upon vulnerable patients in the case of a breach or leakage.

0.7. **Web.** Perhaps the most pressing issue to solve for the web interface would be the SQL injection vulnerability (though HTTPS enforcement is also very high on the list), as this one vulnerability exposes the entire application and its contents to whoever is savvy enough to attempt injecting into it. A relatively quick solution to this problem given the MVC, Model View Controller, architecture of this application is simply passing the Java models (User and Patient) definitions through an ORM which will handle all queries as SQL mappings and sanitize untrustworthy user input for us.

0.7.1. *SQL model mapping.* The purpose of this implementation is to eliminate the use of direct SQL queries and to sanitize user input. To do this it's standard practice to stop treating user inputs as literal strings which get implemented straight into the SQL queries, but instead parametrize them and treat them like data instead. This is the basic concept behind an ORM, but to avoid difficult overhead and implementation of large packages such as `MyBantis` [2] I've elected to perform straight SQL mapping through Java's own JDK methods while sanitizing input.

Staying in line with SQL mapping we'll keep the current SQL queries but exchange its parameters from pure strings to placeholders:

```
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 ?";
```

Listing 3. Redefining SQL queries

One may then redefine the `authenticated` method to make use of these new parameters and query execution:

```
private boolean authenticated(String username, String password) throws
    ↪ SQLException {

      // Prepare the SQL statement with placeholders
    try (java.sql.PreparedStatement pstmt = database.prepareStatement(
        ↪ AUTH_QUERY)) {

        // Bind the user input to the placeholders
        // Driver handles escaping and treats ' or '1=1-- as literal strings
        pstmt.setString(1, username); // Binds 'username' to the first '?'
        pstmt.setString(2, password); // Binds 'password' to the second '?'

        // Execute defined query
        try (ResultSet results = pstmt.executeQuery()) {
            // Check if any row was returned
            return results.next();
        }
    }
  }
```

Listing 4. New authenticated with parameterized and sanitized user input

Removing `String.format()` and `Statement` with a `PreparedStatement` treats the user input as data, not literal strings, ensuring input such as $'\ OR\ '1$ don't become actual executable SQL parameters. This is the same approach I'll use for the `searchResults()` method:

```
private List<Record> searchResults(String surname) throws SQLException {
    List<Record> records = new ArrayList<>();

    // Prepare the SQL statement with a placeholder
    try (java.sql.PreparedStatement pstmt = database.prepareStatement(
        ↪ SEARCH_QUERY)){

      // Bind the user input to the placeholder, manually adding wildcards
      // as PreparedStatement only protects user input itself
      pstmt.setString(1, '%' + surname + "%");

```

```
      // Execute defined query
      try(ResultSet results = pstmt.executeQuery()){
        while (results.next()) {
          Record rec = new Record();
          rec.setSurname(results.getString(2));
          rec.setForename(results.getString(3));
          rec.setAddress(results.getString(4));
          rec.setDateOfBirth(results.getString(5));
          rec.setDoctorId(results.getString(6));
          rec.setDiagnosis(results.getString(7));
          records.add(rec);
        }
      }
    }
    return records;
  }
```

Listing 5. New searchResults with parameterized and sanitized user input

With these salinization methods in place and SQL model mapping against the database, SQL injections no longer work:

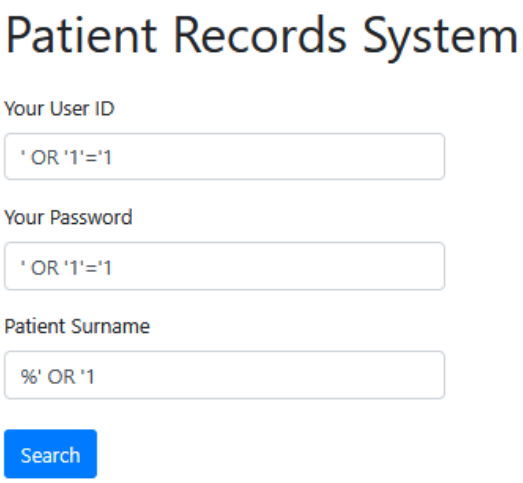


Figure 11. Testing SQL injection against new parameterized queries and sanitization

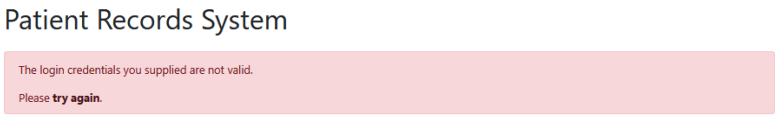


Figure 12. SQL injection with new sanitization and mapping results

To fully test a bunch of payloads from the GitHub page I made a small python script to input these payloads into the form fields, and if the redirect went somewhere other than the non-valid page then the SQL injection would be a success. With proper rate limiting and session management this test script would not be possible, highlighting the necessity of proper validation and authentication methods in this application. The full output of this testing script can be found in the appendix, see listing 15:

```
import requests
from bs4 import BeautifulSoup
import sys
import time

```

```python
# URL of running application
BASE_URL = "http://localhost:8080/"

# Failure message displayed on the invalid login page
FAILURE_TEXT = "The login credentials you supplied are not valid."

# Bunch of SQL injection payloads to test: https://github.com/payloadbox/sql-injection-payload-list
PAYLOADS = [...]

def normalize_payload(p):
    return p.split("\t", 1)[0]

PAYLOADS = [normalize_payload(p) for p in PAYLOADS]

session = requests.Session()
session.headers.update({
    "User-Agent": "sqli-tester/1.0",
})

# Bunch of function definitions to input payloads and respond to site changes, see project repository for full python file (injection.py)

def functions():
    ....

def main():
    print("Starting SQL injection payload test against", BASE_URL)
    for p in PAYLOADS:
        payload = p
        print(f"Testing payload: {payload!r}")
        resp = submit_payload(payload)
        # If we get the failure page (or redirect to it) -> print failed
        if is_failure_page(resp):
            print(f"Injection {payload} failed")
            # follow the try again link if present so next POST goes to the form page
            follow_try_again_link(resp)
            # small pause to avoid aggressive hammering
            time.sleep(0.2)
            continue
        # If not a failure page, check status or heuristics indicating success
        # (e.g., maybe the app returned a search result page / no failure text)
        if resp is None:
            print(f"Injection {payload} result: request error")
            continue
        # Heuristic: if we are still on the root page (form HTML) but without failure text,
        # it could be a different response. Print a message and continue.
        body = resp.text or ""
        if FAILURE_TEXT not in body:
            print(f"Injection {payload} may have succeeded or returned a non-failure response (inspect manually)")
            # Stop after first potential success
            break
        else:
            print(f"Injection {payload} failed")
        time.sleep(0.2)

if __name__ == "__main__":
    main()
```

Listing 6. Python script to test SQL injections

0.8. **Code & Database.** For the code and database vulnerability the largest safety measure to take would be to secure the contents of the database entirely. Even large companies fail to completely inhibit attackers from getting access to their databases, and as such it is highly unlikely that this application will succeed either. As such, focusing on securing its contents and minimizing useful login credentials and personal information from being leaked is of utmost priority.

0.8.1. *Hashing and salting passwords.* Given this is a test application I've elected not to create an entire sign up / register page and form for the application, but rather focused on salting and hashing the current passwords stored in the database, as well as checking user login credentials against these hashed and salted passwords. For encryptions and salting I've made use of `jBCrypt` [3]. To make use of jBCrypt we must first import it after addings its dependency into the `build.gradle` file:

```
// BCrypt library for password hashing
import org.mindrot.jbcrypt.BCrypt;
```

Listing 7. jBCrypt dependency import

With BCrypt now exposed to the application one may fetch only the stored hashed password for a given username:

```
// Updated AUTH_QUERY to select hashed password
private static final String AUTH_QUERY = "select password from user where
    ↪ username=?";
```

Listing 8. Update AUTH_QUERY

Now all that's left is to make sure the `authenticated()` function makes use of `jBCrypt.checkpw()`:

```
private boolean authenticated(String username, String password) throws
    ↪ SQLException {

    // Prepare the SQL statement with placeholders
    try (java.sql.PreparedStatement pstmt = database.prepareStatement(
        ↪ AUTH_QUERY)) {

        // Bind the user input to the placeholders
        // Driver handles escaping and treats ' or '1=1-- as literal strings
        pstmt.setString(1, username); // Binds 'username' to the first '?'


        // Execute defined query
        try (ResultSet results = pstmt.executeQuery()) {
          // Check if a user with that username was found
          if (results.next()) {
            // User found, get the stored hashed password
            String storedHash = results.getString("password");

            // Verify the provided password against the stored hash
            // BCrypt.checkpw handles all the salt comparison logic
            return BCrypt.checkpw(password, storedHash);
          } else {
            // No such user found
            return false;
          }
        }
    }
}
```

Listing 9. Modified authenticated() to use BCrypt check password hash

Now the login form will make use of the hashed and salted passwords for matching. However, since these changes haven't actually changed the passwords themselves already stored in the database from plain text to hashed we need a temporary Java file `HashPasswords.java` to hash all the passwords in the database which will run only once:

```java
package IKT222.Assignment4;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.HashMap;
import java.util.Map;
import org.mindrot.jbcrypt.BCrypt;

/*
 * Utility to hash all plaintext passwords in the 'user' table of the database.
 * This will only be run once to hash all password entries in the user table.
 */

public class HashPasswords {
    private static final String CONNECTION_URL = "jdbc:sqlite:db.sqlite3";
    private static final String SELECT_USERS_QUERY = "SELECT username, password FROM user";
    private static final String UPDATE_PASSWORD_QUERY = "UPDATE user SET password = ? WHERE username = ?";

    public static void main(String[] args) {
        System.out.println("Starting password hashing utility...");
        Connection connection = null;
        Statement selectStatement = null;
        PreparedStatement updateStatement = null;
        ResultSet users = null;

        // This map will store username -> plaintext password
        Map<String, String> userPasswords = new HashMap<>();

        try {
            // Connect to the database
            Class.forName("org.sqlite.JDBC");
            connection = DriverManager.getConnection(CONNECTION_URL);
            // Use transactions for batch updates, allows rollback on potential errors
            connection.setAutoCommit(false);

            System.out.println("Connected to database.");

            // Select all users and their current passwords
            selectStatement = connection.createStatement();
            users = selectStatement.executeQuery(SELECT_USERS_QUERY);

            System.out.println("Fetching users...");
            while (users.next()) {
                String username = users.getString("username");
                String plaintextPassword = users.getString("password");

                // Check if password looks like a hash already
                if (plaintextPassword.startsWith("$2a$")) {
                    System.out.println("Skipping user '" + username + "': password already looks hashed.");
                } else {
                    userPasswords.put(username, plaintextPassword);
                }
            }
```

```
            System.out.println("Found " + userPasswords.size() + " users with plaintext passwords.");

            //Hash and update passwords
            if (userPasswords.isEmpty()) {
                System.out.println("No plaintext passwords to update.");
                return;
            }

            updateStatement = connection.prepareStatement(UPDATE_PASSWORD_QUERY);

            int count = 0;
            for (Map.Entry<String, String> entry : userPasswords.entrySet()) {
                String username = entry.getKey();
                String plaintextPassword = entry.getValue();

                // Generate a salt and hash the password (work factor 12: https://github.com/jeremyh/jBCrypt)
                String hashedPassword = BCrypt.hashpw(plaintextPassword, BCrypt.gensalt(12));

                // Set parameters for the update query
                updateStatement.setString(1, hashedPassword);
                updateStatement.setString(2, username);

                // Add to batch
                updateStatement.addBatch();
                count++;

                System.out.println("Queued update for user: " + username);
            }

            // Execute the batch update
            System.out.println("Executing batch update for " + count + " users ...");
            int[] updateCounts = updateStatement.executeBatch();
            connection.commit(); // Commit transaction

            System.out.println("Successfully updated " + updateCounts.length + " passwords.");
            System.out.println("Password hashing complete.");

        } catch (SQLException | ClassNotFoundException e) {
            System.err.println("An error occurred: " + e.getMessage());
            e.printStackTrace();
            if (connection != null) {
                try {
                    System.err.println("Rolling back transaction...");
                    connection.rollback();
                } catch (SQLException ex) {
                    System.err.println("Error during rollback: " + ex.getMessage());
                }
            }
        } finally {
            // Clean up
            try { if (users != null) users.close(); } catch (SQLException e) { /* ignored */ }
            try { if (selectStatement != null) selectStatement.close(); } catch (SQLException e) { /* ignored */ }
```

```
110             try { if (updateStatement != null) updateStatement.close(); } catch
                    ↪  (SQLException e) { /* ignored */ }
111             try { if (connection != null) connection.close(); } catch (
                    ↪ SQLException e) { /* ignored */ }
112             System.out.println("Database connection closed.");
113         }
114     }
115 }
```

Listing 10. HashPasswords.java

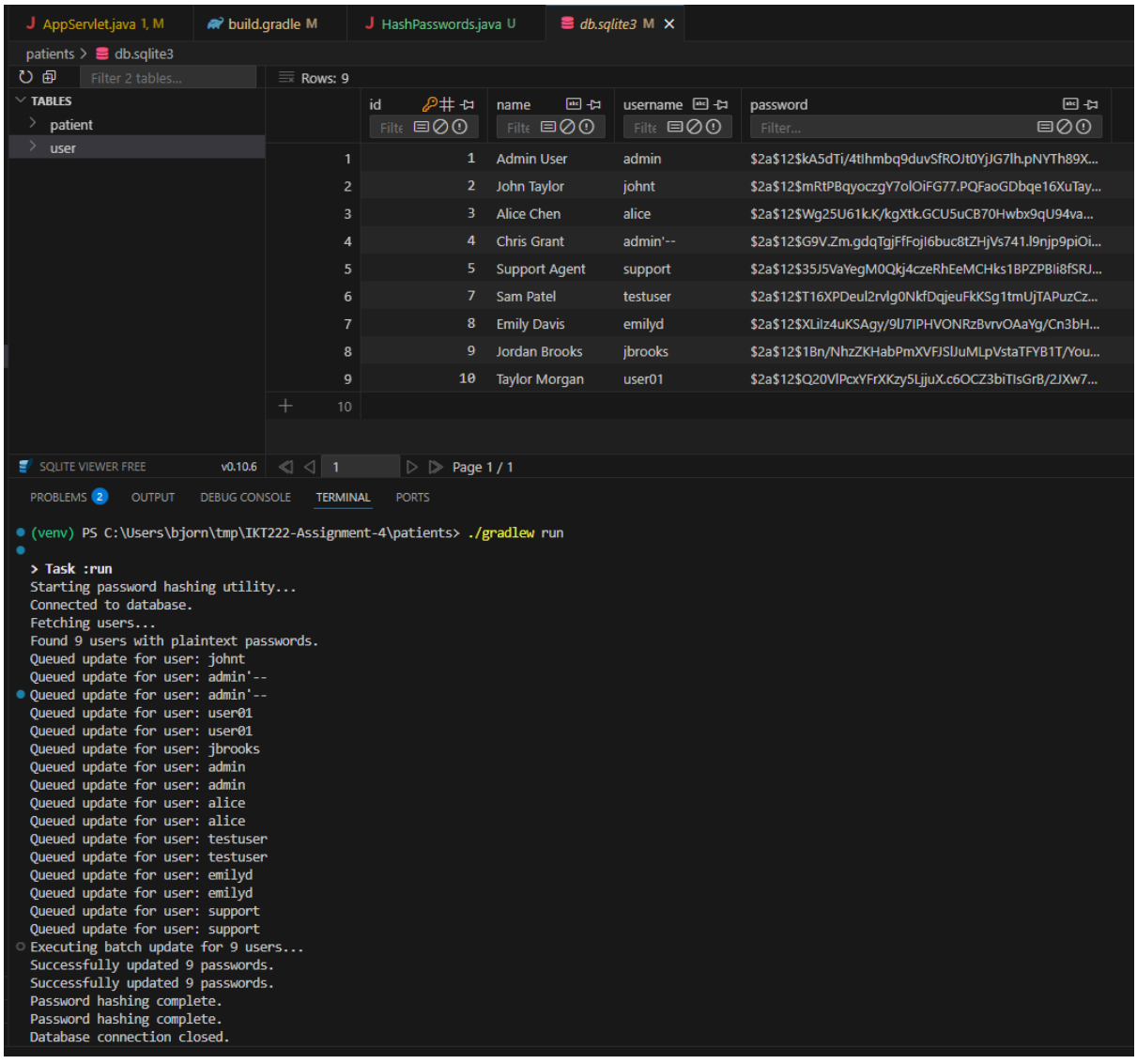


Figure 13. Hashed and salted passwords after running HashPasswords.java

0.8.2. *Encrypting and decrypting sensitive patient information.* Because this data needs to be viewed in plain text again once a general practitioner has searched them up, one cannot make use of a hashing algorithm as these are non-reversible encryptions. Instead we'll have to make use of a reversible cipher to encrypt the sensitive patient columns so that when a general practitioner requests a particular patient history the application may reverse the cipher and procure the plain text anew. Ordinarily one would encrypt the entirety of the identifiable patient information columns, but because the search query in the main application makes use of the `surname` column as a search index I've elected to only encrypt

`forname`, `address`, `born` and `treated_for` to minimize the amount of possibly identifiable information tied to a patient in the case of a breach.

This encryption will be making use of AES, Advanced Encryption Standard [4], as it is a symmetric key algorithm standardized as part of the JDK as well as a good fit for this encryption purpose. Ordinarily one would be hiding away the secret key in an environment file or similar methods, but for simplicity's sake I'll be hard coding it into the application for encrypting the relevant columns in the patient table as well as decrypting them.

For the cipher mode I've elected to make use of CBC, Cipher Block Chaining, with PKCS5Padding. To encrypt the columns I approached the process in the same manner as hashing and salting passwords, by creating a separate java file and executing it once:

```java
package IKT222.Assignment4;

import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.spec.IvParameterSpec;
import java.security.SecureRandom;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Base64;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;

// Simple class to hold patient data and ID before encryption
class PatientRecord {
    int id;
    String forename;
    String born;
    String address;
    String treatedfor;

    public PatientRecord(int id, String forename, String born, String address,
        ↪ String treatedfor) {
        this.id = id;
        this.forename = forename;
        this.born = born;
        this.address = address;
        this.treatedfor = treatedfor;
    }
}

public class EncryptPatients {
    // Do not use hardcoded keys in production systems! This is just for
        ↪ examplification.
    private static final String SECRET_KEY = "SupastrongKey123";
    private static final String ALGORITHM = "AES";
    private static final String TRANSFORMATION = "AES/CBC/PKCS5Padding";
    private static final String CONNECTION_URL = "jdbc:sqlite:db.sqlite3";

    // SQL queries
    private static final String SELECT_PATIENTS_QUERY = "SELECT id, forename,
        ↪ born, address, treated_for FROM patient";
    private static final String UPDATE_PATIENT_QUERY = "UPDATE patient SET
        ↪ forename = ?, born = ?, address = ?, treated_for = ? WHERE id = ?";

    private static SecretKeySpec getKeySpec() {
```

```
        return new SecretKeySpec(SECRET_KEY.getBytes(StandardCharsets.UTF_8),
            ↪ ALGORITHM);
    }

    public static String encrypt(String plainText) throws Exception {
        if (plainText == null || plainText.isEmpty()) {
            return null;
        }

        Cipher cipher = Cipher.getInstance(TRANSFORMATION);
        byte[] iv = new byte[16];
        new SecureRandom().nextBytes(iv);
        IvParameterSpec ivSpec = new IvParameterSpec(iv);

        cipher.init(Cipher.ENCRYPT_MODE, getKeySpec(), ivSpec);

        byte[] encryptedBytes = cipher.doFinal(plainText.getBytes(
            ↪ StandardCharsets.UTF_8));

        // Prepend the IV to the ciphertext
        byte[] combined = new byte[iv.length + encryptedBytes.length];
        System.arraycopy(iv, 0, combined, 0, iv.length);
        System.arraycopy(encryptedBytes, 0, combined, iv.length, encryptedBytes
            ↪ .length);

        return Base64.getEncoder().encodeToString(combined);
    }

    public static void main(String[] args) {
        System.out.println("Starting patient data encryption utility...");
        Connection connection = null;
        Statement selectStatement = null;
        PreparedStatement updateStatement = null;
        ResultSet patients = null;

        // Collect patient records to encrypt
        List<PatientRecord> patientsToEncrypt = new ArrayList<>();

        try {
            // Connect to the database (Same as HashPasswords.java)
            Class.forName("org.sqlite.JDBC");
            connection = DriverManager.getConnection(CONNECTION_URL);
            // Use transactions for batch updates, allows rollback on potential
                ↪  errors
            connection.setAutoCommit(false);

            System.out.println("Connected to database.");

            // Select all patients
            selectStatement = connection.createStatement();
            patients = selectStatement.executeQuery(SELECT_PATIENTS_QUERY);

            System.out.println("Fetching patient records...");
            while (patients.next()) {
                int id = patients.getInt("id");
                String forename = patients.getString("forename");

                // Simple check to skip already encrypted records (encrypted
                    ↪ data is Base64, so it should contain '=')
                if (forename != null && !forename.contains("=")) {
                    patientsToEncrypt.add(new PatientRecord(
                        id,
```

```
                        forename,
                        patients.getString("born"),
                        patients.getString("address"),
                        patients.getString("treated_for")
                    ));
                }
            }
            System.out.println("Found " + patientsToEncrypt.size() + " patients
                ↪  with plaintext data to encrypt.");

            if (patientsToEncrypt.isEmpty()) {
                System.out.println("No plaintext records to update. Encryption
                    ↪ utility exiting.");
                return;
            }

            // Encrypt and prepare batch update
            updateStatement = connection.prepareStatement(UPDATE_PATIENT_QUERY)
                ↪ ;
            int count = 0;

            for (PatientRecord record : patientsToEncrypt) {
                String encryptedForename = encrypt(record.forename);
                String encryptedborn = encrypt(record.born);
                String encryptedAddress = encrypt(record.address);
                String encryptedTreatedFor = encrypt(record.treatedfor);

                updateStatement.setString(1, encryptedForename);
                updateStatement.setString(2, encryptedborn);
                updateStatement.setString(3, encryptedAddress);
                updateStatement.setString(4, encryptedTreatedFor);
                updateStatement.setInt(5, record.id);
                updateStatement.addBatch();
                count++;
            }

            // Execute the batch update and commit
            System.out.println("Executing batch update for " + count + "
                ↪ records...");
            int[] updateCounts = updateStatement.executeBatch();
            connection.commit();

            System.out.println("Successfully encrypted and updated " +
                ↪ updateCounts.length + " patient records.");

        } catch (Exception e) {
            System.err.println("An error occurred: " + e.getMessage());
            e.printStackTrace();
            if (connection != null) {
                try {
                    System.err.println("Rolling back transaction...");
                    connection.rollback();
                } catch (SQLException ex) {
                    System.err.println("Error during rollback: " + ex.
                        ↪ getMessage());
                }
            }
        } finally {
            // Clean up
            try {
                if (patients != null) patients.close();
            } catch (SQLException e) { /* ignored */ }
```

```
            try {
                if (selectStatement != null) selectStatement.close();
            } catch (SQLException e) { /* ignored */ }
            try {
                if (updateStatement != null) updateStatement.close();
            } catch (SQLException e) { /* ignored */ }
            try {
                if (connection != null) connection.close();
            } catch (SQLException e) { /* ignored */ }
            System.out.println("Database connection closed.");
        }
    }
}
```

Listing 11. EncryptPatiens.java

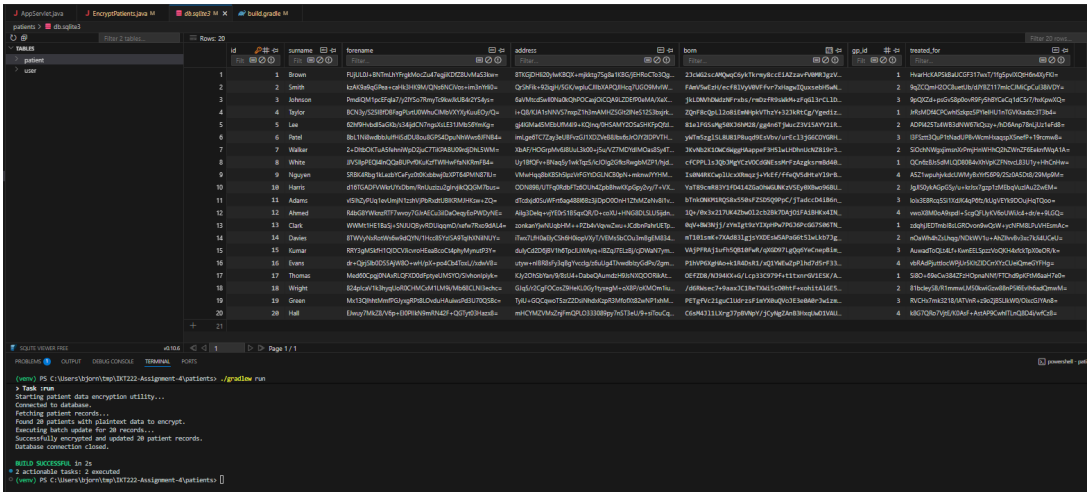


Figure 14. Encrypted sensitive data after running EncryptPatiens.java (zoom 250% for better resolution)

Now that patient information is encrypted, we need to decrypt it when a user is authenticated and is pulling out the patient records:

```
// AES cipher parameters
private static final String SECRET_KEY = "SupastrongKey123";
private static final String ALGORITHM = "AES";
private static final String TRANSFORMATION = "AES/CBC/PKCS5Padding";
```

Listing 12. Including encryption parameters (same as when encrypting)

```
// Helper to get the key spec
private static SecretKeySpec getKeySpec() {
    return new SecretKeySpec(SECRET_KEY.getBytes(StandardCharsets.UTF_8),
        ↪ ALGORITHM);
}

// Decryption method
private static String decrypt(String encryptedText) throws Exception {
    if (encryptedText == null || encryptedText.isEmpty()) {
        return null;
    }

    byte[] combined = Base64.getDecoder().decode(encryptedText);
    final int IV_LENGTH = 16;

```

```
    if (combined.length < IV_LENGTH) {
      throw new IllegalArgumentException("Encrypted data is too short to
          ↪ contain a valid IV.");
    }

    byte[] iv = new byte[IV_LENGTH];
    System.arraycopy(combined, 0, iv, 0, IV_LENGTH);
    IvParameterSpec ivSpec = new IvParameterSpec(iv);

    int cipherTextLength = combined.length - IV_LENGTH;
    byte[] cipherText = new byte[cipherTextLength];
    System.arraycopy(combined, IV_LENGTH, cipherText, 0, cipherTextLength);

    Cipher cipher = Cipher.getInstance(TRANSFORMATION);
    cipher.init(Cipher.DECRYPT_MODE, getKeySpec(), ivSpec);

    byte[] decryptedBytes = cipher.doFinal(cipherText);
    return new String(decryptedBytes, StandardCharsets.UTF_8);
  }
```

Listing 13. Helper function and decryption definition

```
// Modified to decrypt patient data before returning web page
 private List<Record> searchResults(String surname) throws SQLException {
   List<Record> records = new ArrayList<>();

   // Prepare the SQL statement with a placeholder
   try (PreparedStatement pstmt = database.prepareStatement(SEARCH_QUERY)) {

     // Bind the user input to the placeholder, manually adding wildcards
     // as PreparedStatement only protects user input itself
     pstmt.setString(1, '%' + surname + "%");

     // Execute defined query
     try (ResultSet results = pstmt.executeQuery()) {
       while (results.next()) {
         Record rec = new Record();
         String errorMsg = "DECRYPTION FAILED";

         // Plaintext data
         rec.setSurname(results.getString(2));
         rec.setDoctorId(results.getString(6));

         // Retrieve encrypted data
         String encryptedForename = results.getString(3);
         String encryptedAddress = results.getString(4);
         String encryptedBorn = results.getString(5);
         String encryptedTreatedFor = results.getString(7);

         try {
           // Decrypting and setting sensitive columns
           rec.setForename(decrypt(encryptedForename));
           rec.setAddress(decrypt(encryptedAddress));
           rec.setDateOfBirth(decrypt(encryptedBorn));
           rec.setDiagnosis(decrypt(encryptedTreatedFor));
         } catch (Exception e) {
           System.err.println(
               "Decryption failed for a record. Data might be corrupted or key
                   ↪  is wrong: " + e.getMessage());
           // Set fields to an error message or null if decryption fails
           rec.setForename(errorMsg);
           rec.setAddress(errorMsg);
```

```
            rec.setDateOfBirth(errorMsg);
            rec.setDiagnosis(errorMsg);
          }
          records.add(rec);
        }
      }
    }
    return records;
  }
```

Listing 14. Updated searchResults to decrypt encrypted columns before display

With this decryption method in order the database is now relatively secretive, and as such even if a malicious actor were to gain access to it, would retrieve minimally useful data to exploit or piece together, all while keeping the application functionality the exact same:

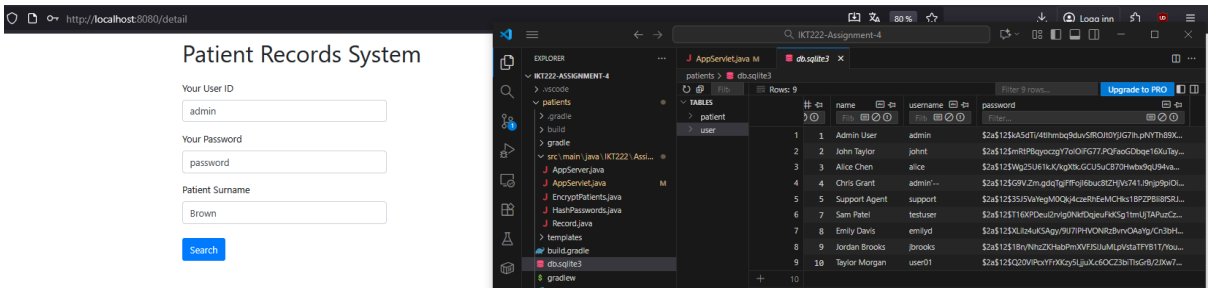


Figure 15. Testing application with encrypted database information (zoom 250% for better resolution)

Patient Records System

Patient Details

Surname	Forename	Date of Birth	GP Identifier	Treated For
Brown	James	2016-04-10	1	Asthma

Home

Figure 16. Result of test

```
Starting SQL injection payload test against http:/localhost:8080/
Testing payload: "'"
Injection ' failed
Testing payload: "''"
Injection '' failed
Testing payload: '‘'
Injection ‘ failed
Testing payload: '‘‘'
Injection ‘‘ failed
Testing payload: ','
Injection , failed
Testing payload: '"'
Injection " failed
Testing payload: '""'
Injection "" failed
Testing payload: '/'
Injection / failed
Testing payload: '//'
Injection // failed
Testing payload: '\\'
Injection \ failed
Testing payload: '\\\\'
Injection \\ failed
Testing payload: ';'
Injection ; failed
Testing payload: '\' or "'
Injection ' or " failed
Testing payload: '-- or #'
Injection -- or # failed
Testing payload: "' OR '1"
Injection ' OR '1 failed
Testing payload: "' OR 1 -- -"
Injection ' OR 1 -- - failed
Testing payload: '" OR "" = "'
Injection " OR "" = " failed
Testing payload: '" OR 1 = 1 -- -'
Injection " OR 1 = 1 -- - failed
Testing payload: "' OR '' = '"
Injection ' OR '' = ' failed
Testing payload: "'='"
Injection '=' failed
Testing payload: "'LIKE'"
Injection 'LIKE' failed
Testing payload: "'=0--+"
Injection '=0--+ failed
Testing payload: ' OR 1=1'
Injection  OR 1=1 failed
Testing payload: "' OR 'x'='x"
Injection ' OR 'x'='x failed
Testing payload: "' AND id IS NULL; --"
Injection ' AND id IS NULL; -- failed
Testing payload: "''''''''''''UNION SELECT '2"
Injection ''''''''''''UNION SELECT '2 failed
Testing payload: '-'
Injection - failed
Testing payload: ' '
Injection   failed
Testing payload: '&'
Injection & failed
Testing payload: '^'
```

```
Injection ^ failed
Testing payload: '*'
Injection * failed
Testing payload: " or '-"
Injection  or '- failed
Testing payload: " or ' '"
Injection  or ' ' failed
Testing payload: " or '&'"
Injection  or '&' failed
Testing payload: " or '^'"
Injection  or '^' failed
Testing payload: " or '*'"
Injection  or '*' failed
Testing payload: '-'
Injection - failed
Testing payload: ' '
Injection   failed
Testing payload: '&'
Injection & failed
Testing payload: '^'
Injection ^ failed
Testing payload: '*'
Injection * failed
Testing payload: ' or "-'
Injection  or "- failed
Testing payload: ' or " "'
Injection  or " " failed
Testing payload: ' or "&"'
Injection  or "&" failed
Testing payload: ' or "^"'
Injection  or "^" failed
Testing payload: ' or "*"'
Injection  or "*" failed
Testing payload: 'or true--'
Injection or true-- failed
Testing payload: '" or true--'
Injection " or true-- failed
Testing payload: "' or true--"
Injection ' or true-- failed
Testing payload: '") or true--'
Injection ") or true-- failed
Testing payload: "') or true--"
Injection ') or true-- failed
Testing payload: "' or 'x'='x"
Injection ' or 'x'='x failed
Testing payload: "') or ('x')=('x"
Injection ') or ('x')=('x failed
Testing payload: "')) or (('x'))=(('x"
Injection ')) or (('x'))=(('x failed
Testing payload: '" or "x"="x'
Injection " or "x"="x failed
Testing payload: '") or ("x")=("x'
Injection ") or ("x")=("x failed
Testing payload: '")) or (("x"))=(("x'
Injection )) or (("x"))=(("x failed
Testing payload: 'or 1=1'
Injection or 1=1 failed
Testing payload: 'or 1=1--'
Injection or 1=1-- failed
Testing payload: 'or 1=1#'
Injection or 1=1# failed
Testing payload: 'or 1=1/*'
```

```
Injection or 1=1/* failed
Testing payload: "admin' --"
Injection admin' -- failed
Testing payload: "admin' #"
Injection admin' # failed
Testing payload: "admin'/*"
Injection admin'/* failed
Testing payload: "admin' or '1'='1"
Injection admin' or '1'='1 failed
Testing payload: "admin' or '1'='1'--"
Injection admin' or '1'='1'-- failed
Testing payload: "admin' or '1'='1'#"
Injection admin' or '1'='1'# failed
Testing payload: "admin' or '1'='1'/*"
Injection admin' or '1'='1'/* failed
Testing payload: "admin'or 1=1 or ''='"
Injection admin'or 1=1 or ''=' failed
Testing payload: "admin' or 1=1"
Injection admin' or 1=1 failed
Testing payload: "admin' or 1=1--"
Injection admin' or 1=1-- failed
Testing payload: "admin' or 1=1#"
Injection admin' or 1=1# failed
Testing payload: "admin' or 1=1/*"
Injection admin' or 1=1/* failed
Testing payload: "admin') or ('1'='1"
Injection admin') or ('1'='1 failed
Testing payload: "admin') or ('1'='1'--"
Injection admin') or ('1'='1'-- failed
Testing payload: "admin') or ('1'='1'#"
Injection admin') or ('1'='1'# failed
Testing payload: "admin') or ('1'='1'/*"
Injection admin') or ('1'='1'/* failed
Testing payload: "admin') or '1'='1"
Injection admin') or '1'='1 failed
Testing payload: "admin') or '1'='1'--"
Injection admin') or '1'='1'-- failed
Testing payload: "admin') or '1'='1'#"
Injection admin') or '1'='1'# failed
Testing payload: "admin') or '1'='1'/*"
Injection admin') or '1'='1'/* failed
Testing payload: "1234 ' AND 1=0 UNION ALL SELECT 'admin', '81
    ↪ dc9bdb52d04dc20036dbd8313ed055"
Injection 1234 ' AND 1=0 UNION ALL SELECT 'admin', '81
    ↪ dc9bdb52d04dc20036dbd8313ed055 failed
Testing payload: 'admin" --'
Injection admin" -- failed
Testing payload: 'admin" #'
Injection admin" # failed
Testing payload: 'admin"/*'
Injection admin"/* failed
Testing payload: 'admin" or "1"="1'
Injection admin" or "1"="1 failed
Testing payload: 'admin" or "1"="1"--'
Injection admin" or "1"="1"-- failed
Testing payload: 'admin" or "1"="1"#'
Injection admin" or "1"="1"# failed
Testing payload: 'admin" or "1"="1"/*'
Injection admin" or "1"="1"/* failed
Testing payload: 'admin"or 1=1 or ""="'
Injection admin"or 1=1 or ""=" failed
Testing payload: 'admin" or 1=1'
```

```
Injection admin" or 1=1 failed
Testing payload: 'admin" or 1=1--'
Injection admin" or 1=1-- failed
Testing payload: 'admin" or 1=1#'
Injection admin" or 1=1# failed
Testing payload: 'admin" or 1=1/*'
Injection admin" or 1=1/* failed
Testing payload: 'admin") or ("1"="1'
Injection admin") or ("1"="1 failed
Testing payload: 'admin") or ("1"="1"--'
Injection admin") or ("1"="1"-- failed
Testing payload: 'admin") or ("1"="1"#'
Injection admin") or ("1"="1"# failed
Testing payload: 'admin") or ("1"="1"/*'
Injection admin") or ("1"="1"/* failed
Testing payload: 'admin") or "1"="1'
Injection admin") or "1"="1 failed
Testing payload: 'admin") or "1"="1"--'
Injection admin") or "1"="1"-- failed
Testing payload: 'admin") or "1"="1"#'
Injection admin") or "1"="1"# failed
Testing payload: 'admin") or "1"="1"/*'
Injection admin") or "1"="1"/* failed
Testing payload: '1234 " AND 1=0 UNION ALL SELECT "admin", "81
    ↪ dc9bdb52d04dc20036dbd8313ed055'
Injection 1234 " AND 1=0 UNION ALL SELECT "admin", "81
    ↪ dc9bdb52d04dc20036dbd8313ed055 failed
```

Listing 15. SQL injection script output

References

[1] payloadbox, *sql-injection-payload-list*, Oct. 2025. [Online]. Available: https://github.com/payloadbox/sql-injection-payload-list.

[2] MyBatis, *MyBatis 3 | Introduction – mybatis*, Jan. 2025. [Online]. Available: https://mybatis.org/mybatis-3.

[3] J. Hooke, *jBCrypt - strong password hashing for Java*, [Online; accessed 31. Oct. 2025], Jan. 2015. [Online]. Available: https://www.mindrot.org/projects/jBCrypt.

[4] GeeksforGeeks, "Advanced Encryption Standard (AES)," *GeeksforGeeks*, Aug. 2025. [Online]. Available: https://www.geeksforgeeks.org/computer-networks/advanced-encryption-standard-aes.