Title

Vulnerable Software:
Analysis, Identification and Mitigation

Module

COMP3911 - Secure Computing

Authors

Araav Bhardwaj - sc23ab3

Mohab Elhasade - sc22mme

Nathaniel Sutton - sc22ns

Christopher Tobing - sc22cjt

Matthew Toon - sc22mt

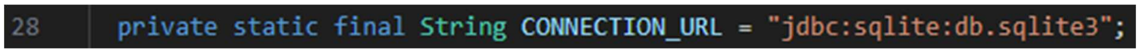
Contents

1. Analysis of Flaws

1.1. Flaw 1 - Visible Database String

The SQLite3 database connection string is visible in the source code (‘AppServlet.java’).

The connection string (figure 1.) is hardcoded at the top of the ‘AppServlet’ class on line 28. Storing digital authentication credentials, commonly referred to as “secrets” in software development, violates secure programming practices. As seen here, this can result in the exposure of internal system details if the source code is leaked or included in a project repository such as GitHub.



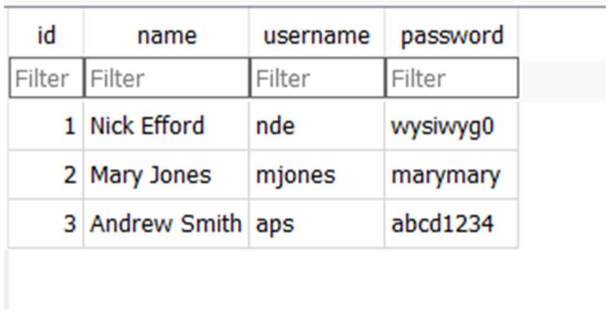
```
28    private static final String CONNECTION_URL = "jdbc:sqlite:db.sqlite3";
```

Figure 1.

The identification of this flaw occurred during the first manual review of the ‘AppServlet’ file. While inspecting the initialisation method (lines 36-39), It was observed that the `connectToDatabase()` function calls the following: `DriverManager.getConnection(CONNECTION_URL)`, and `CONNECTION_URL` is defined as a constant within the same file.

1.2. Flaw 2 - Plain Text Password Storage

Passwords are stored in plain text in the SQLite3 database without any hashing or encryption. The user auth table stores passwords as plain text strings (see right figure), which can be directly read from the database. This violates security guidelines and exposes users accounts if the database is accessed through SQL injection, unauthorised file access or other types of attacks.



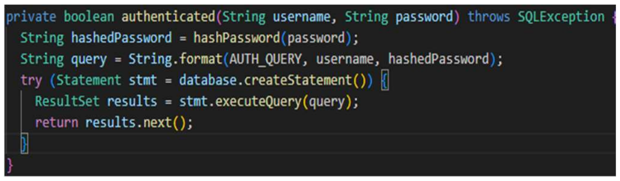
id	name	username	password
Filter	Filter	Filter	Filter
1	Nick Efford	nde	wysiwyg0
2	Mary Jones	mjones	marymary
3	Andrew Smith	aps	abcd1234

Figure 2.

The identification of this flaw happened when reviewing the database and the authenticated() method in AppServlet file (lines 106-112). The AUTH_QUERY directly compares the user inputted password against the database column.

1.3. Flaw 3 - SQL Injection Risk via string concatenation/formatting

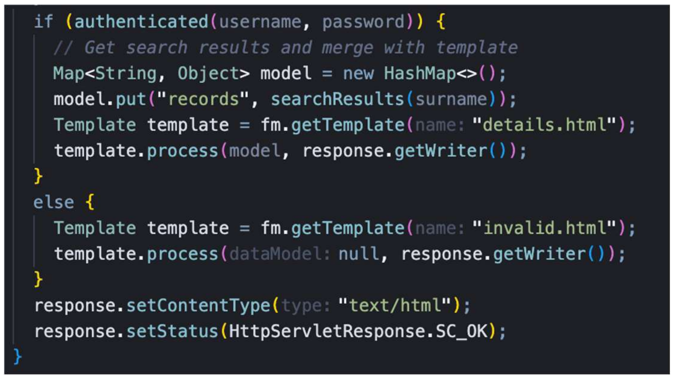
Currently, untrusted user input such as username, password and surname is being directly converted into the SQL query string. This is achieved via String.format() followed by the query being executed using a plain statement. The use of a plain statement means that injected characters will be interpreted by the database rather than as data – displaying a clear vulnerability to SQL injection.



```java
private boolean authenticated(String username, String password) throws SQLException {
  String hashedPassword = hashPassword(password);
  String query = String.format(AUTH_QUERY, username, hashedPassword);
  try (Statement stmt = database.createStatement()) {
    ResultSet results = stmt.executeQuery(query);
    return results.next();
  }
}
```

Figure 3.

1.4. Flaw 4 - Brute Force Vulnerability



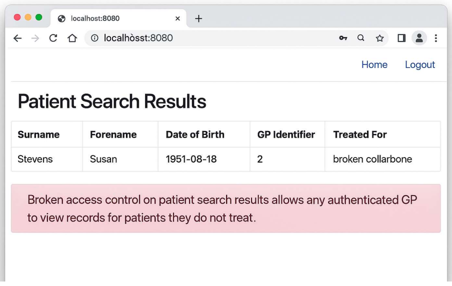
```java
  if (authenticated(username, password)) {
    // Get search results and merge with template
    Map<String, Object> model = new HashMap<>();
    model.put("records", searchResults(surname));
    Template template = fm.getTemplate(name: "details.html");
    template.process(model, response.getWriter());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

Figure 4.

The application has no brute force protection, allowing unlimited login attempts without rate limiting, attempt tracking, or lockout. An attacker can repeatedly try credentials (manually or automated) until they succeed, with no mechanism to detect or prevent this. This makes the system vulnerable to brute force attacks, where attackers systematically try username/password combinations until finding valid credentials. This flaw is exemplified in the code above during the POST request, where the condition that checks whether the user isn't authenticated doesn't have a tracker of attempts taken by each IP address and instead simply returns invalid.html with no penalty. This allows the attacker methods previously stated.

1.5. Flaw 5 - Broken access control on patient search results allows any authenticated GP to view records for patients they do not treat.

The application does not enforce per-GP access control when returning search results. Once a user is successfully authenticated, they are able to view all patient records that match a surname, including patients belonging to other GPs. This breaks the intended access model that patient data should only be visible to the GP assigned to them and violates the principle of least privilege. This flaw was verified by logging in as a GP who is not associated with a given patient and still receiving full access to their record.



Patient Search Results

Surname	Forename	Date of Birth	GP Identifier	Treated For
Stevens	Susan	1951-08-18	2	broken collarbone

Broken access control on patient search results allows any authenticated GP to view records for patients they do not treat.

Figure 5.

2. Attack Tree

1. Flaw 1 - Goal (root node): Obtain access to database of patient medical records
    1.1. Read database connection string in source code
        1.1.1. Workstation not locked when user is absent (**P**)
        1.1.2. Discover project repository on publicly accessible web page (**most likely**)
            1.1.2.1. Developer accidently pushes repo as public by mistake (**P**)
            1.1.2.2. .git folder added to webserver in previous deployment (**most likely**)
            1.1.2.3. Developer shares code snippets on forum websites while seeking assistance (e.g. Stack Overflow) (**P**)
        1.1.3. Internal credential leak (insider discovers and shares connection string) (**P**

2. Flaw 2 - Goal (root node): Get user passwords to gain unauthorised access
    2.1. Access database directly (**most likely**)
        2.1.1. SQL injection attack to dump user table (**most likely**)
        2.1.2. Unauthorised access to file system, opening db.sqlite3 (**P**)
    2.2. Man-in-the-middle attack on database queries (**P**)

3. Flaw 3 - Goal (Root node): Gain unauthorized access to the patient records

    3.1. Use SQL injection exploit in patients search. **(most likely)**

    3.1.1. Use search form to input a crafted query in form of username and surname. **(most likely)**
    3.1.2. Program builds SQL query via string concatenation **(P)**
    3.1.3. Database executes the injected SQL query **(P)**
    3.1.4. Outcome: Attacker can view database records without authorization. **(most likely)**

4. Flaw 4 - Goal (root node): Obtain access to database via correct sign-in details

- 4.1 Brute force authentication credentials
    - **4.1.1.** Enumerate valid usernames (**P**)
        - **4.1.1.1.** Test common usernames (admin, user, test) (**P**)
        - **4.1.1.3.** Check for username enumeration vulnerabilities (**P**)
    - **4.1.2.** Discover login endpoint on publicly accessible web page (**most likely**)
        - **1.1.2.1.** Identify login form structure (**most likely**)
        - **1.1.2.2.** Determine authentication mechanism (username/password)
    - **4.1.3.** Execute automated brute force attack (**most likely**)
        - **4.1.3.1.** Create/obtain username and password lists (**P**)
        - **4.1.3.2.** Send unlimited login attempts with no rate limiting until success (**most likely**)
    - **4.1.4.** Access patient records using compromised credentials (most likely)

5. Flaw 5 - Root Node: View medical records of patients assigned to other GPs

5.1. Obtain valid GP login (**most likely**)
5.2. Perform patient surname search via interface
5.3. Application query does not restrict by gp_id
5.4. Receive patient details for individuals belonging to different GPs
5.5. Outcome: Breach of confidentiality and unauthorised access to medical data

3. Fix Identification

3.1. Flaw 1 - Visible Database String

Assumption: The program and its associated files have at some point been version controlled using a public platform such as GitHub.

Based on the relevant branch for this flaw (1.1.) from the attack tree, an effective form of mitigation would be to move the connection string out of the 'AppServlet' file and into an environment file. By relocating this information and excluding the file from version control, all three attack paths to this vulnerability are effectively removed. This fix was inspired by the 'config' section of 'The Twelve-Factor App' (Wiggins, 2017), as well as a brief article which provides an overview of .env files (Garfield, 2024).
A more robust solution would be to update the operating system's environment variables to include the string, but this is not feasible given the scope of the assignment.

3.2. Flaw 2 - Plain Text Password Storage

The most effective and simple fix identified is to implement SHA-256 hashing for all passwords before storing them. This ensures that even if the database is compromised, attackers can't use the stored values to login.

3.3. Flaw 3 - SQL Injection Risk

As can be seen from the attack tree produced for the risk of SQL Injection, the step which enables an attacker to do so is when the "Program builds the SQL query using string concatenation".

To cut off this entire branch, and thus prevent the attack, we must remove the ability to craft malicious SQL queries by concatenating strings. We can do this by introducing parameterised queries instead.

By using prepared statements, we can ensure that the input given by the user is treated as data only and is unable to modify the query structure – preventing this type of SQL injection.

3.4. Flaw 4 - Brute Force Vulnerability

Assumption: The application can track failed login attempts per IP address and enforce temporary lockouts.

Mitigation: To mitigate branch **4.1.3.2** (unlimited login attempts with no rate limiting), implement a 5-attempt timeout mechanism that tracks failed attempts per IP using an in-memory `ConcurrentHashMap`. After 5 consecutive failures, lock the IP for 15 minutes, blocking further authentication attempts. Reset attempts on successful login or after lockout expiration. This limits brute force by capping attempts and adding delay, making automated credential guessing impractical while allowing legitimate users a few retries.

3.5. Flaw 5 - Broken Access Control

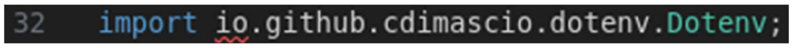
Assumption: Each patient record contains a gp_id field indicating which GP is responsible for them.

To mitigate this flaw, the search logic must enforce that users can only view patients whose gp_id matches their authenticated user id. By modifying the SQL query to include a gp_id constraint, the entire branch of unauthorised record access is prevented. This directly addresses step 5.3 of the attack tree by removing the ability to retrieve patients belonging to other doctors. The only records returned are those assigned to the correct GP.

4. Fixes Implemented

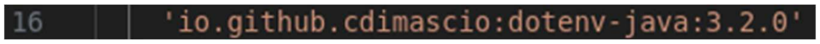
4.1. Flaw 1 - Visible Database String

The connection string was removed from ‘AppServlet’ and replaced with a call to a new ‘.env’ environment file (figure 8.) which reads the value securely at runtime. This new file been added to the ‘patients’ folder alongside the rest of the program and acts as a place to hide authentication secrets and would not be included in future repository commits. The ‘build.gradle’ file also required a minor modification (figure 7.) alongside the related import added to ‘Appservlet’ (figure 6.) to facilitate this solution. As an additional benefit to this fix, secrets are no longer be accessible by simply reviewing the code in the event of a source leak for example, adding an additional layer of security.
It should be noted that for the sake of the assignment, ‘.env’ was added to the group GitHub repository for transparency.



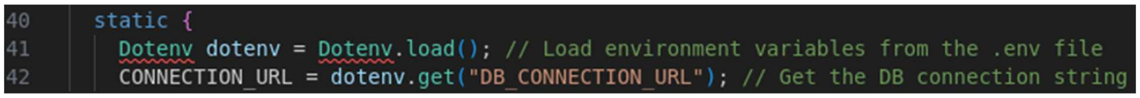
```
import io.github.cdimascio.dotenv.Dotenv;
```

Figure 6.



```
'io.github.cdimascio:dotenv-java:3.2.0'
```

Figure 7.



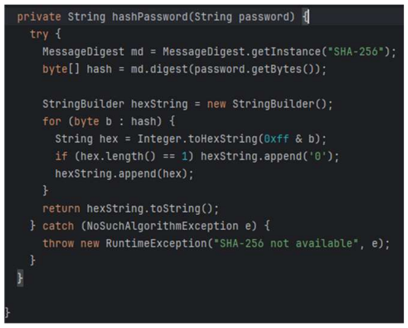
```
static {
    Dotenv dotenv = Dotenv.load(); // Load environment variables from the .env file
    CONNECTION_URL = dotenv.get("DB_CONNECTION_URL"); // Get the DB connection string
```

Figure 8.

4.2. Flaw 2 - Plain Text Password Storage

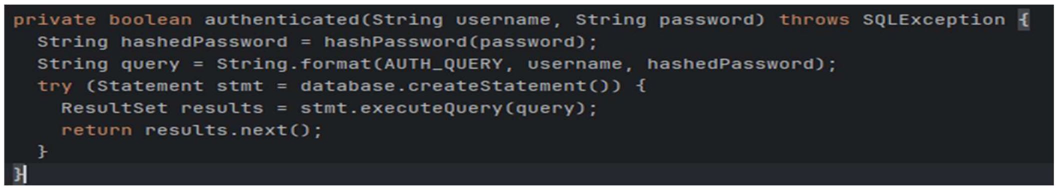
Three changes were made to the codebase to implement the password hashing.

Firstly, I added a hashPassword() helper method to 'AppServlet.java' (shown right) that converts passwords into SHA-256 hashes using Java's MessageDigest class. Then I modified the authenticated() method (shown below) to hash the user inputted passwords before they're compared in database query. I also converted all existing passwords stored to their respective hashed versions.



```
private String hashPassword(String password) {
  try {
    MessageDigest md = MessageDigest.getInstance("SHA-256");
    byte[] hash = md.digest(password.getBytes());

    StringBuilder hexString = new StringBuilder();
    for (byte b : hash) {
      String hex = Integer.toHexString(0xff & b);
      if (hex.length() == 1) hexString.append('0');
      hexString.append(hex);
    }
    return hexString.toString();
  } catch (NoSuchAlgorithmException e) {
    throw new RuntimeException("SHA-256 not available", e);
  }
}

}
```

Figure 9.



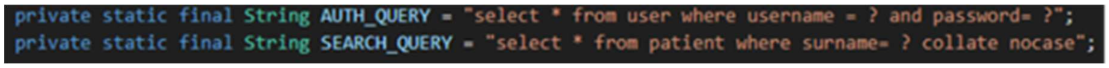
```
private boolean authenticated(String username, String password) throws SQLException {
  String hashedPassword = hashPassword(password);
  String query = String.format(AUTH_QUERY, username, hashedPassword);
  try (Statement stmt = database.createStatement()) {
    ResultSet results = stmt.executeQuery(query);
    return results.next();
  }
}
```

Figure 10.

4.3. Flaw 3 - SQL Injection Risk

To remove the risk for SQL injection, three changes were made to the original implementation. These consisted of the following:

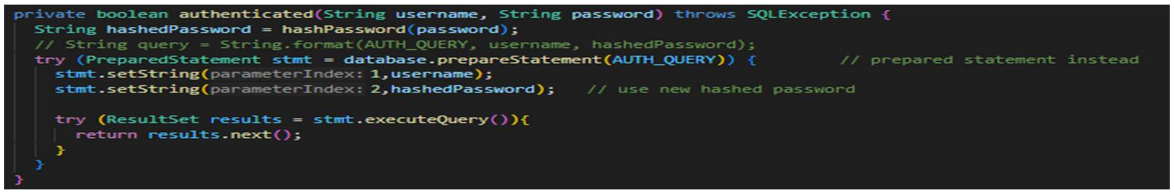
In "AppServlet.java", parameterised SQL statements were used for AUTH_QUERY and SEARCH_QUERY via "?" placeholders. In doing so, this informs JDBC to send the user-inputted values separately to the SQL structure, preventing any injected data from altering query logic.



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

Figure 11.

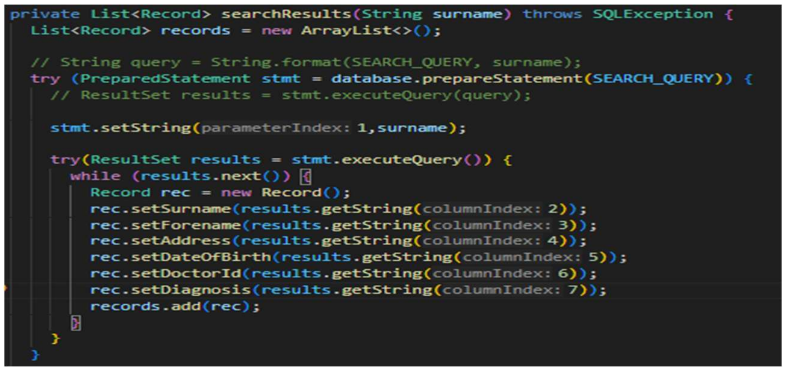
The authenticated() method was updated so that PreparedStatements were used in combination with hashed passwords. Both username and hashed password are bound via setString(), which strictly separates between SQL code and user data. By doing this, if an attacker were to craft an input this would be treated purely as text and cannot be executed in SQL.



```
private boolean authenticated(String username, String password) throws SQLException {
  String hashedPassword = hashPassword(password);
  // String query = String.format(AUTH_QUERY, username, hashedPassword);
  try (PreparedStatement stmt = database.prepareStatement(AUTH_QUERY)) {        // prepared statement instead
    stmt.setString(parameterIndex: 1,username);
    stmt.setString(parameterIndex: 2,hashedPassword);   // use new hashed password

    try (ResultSet results = stmt.executeQuery()){
      return results.next();
    }
  }
}
```

Figure 12.

Finally, the searchResults() method was also converted to use a PreparedStatement, with the user inputs being bound via .setString() and are not concatenated into the query. In doing so, this prevents the injection of SQL clauses.

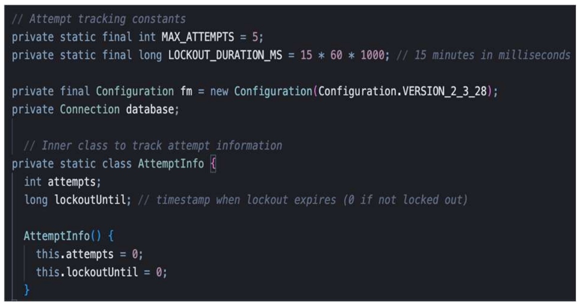


```java
private List<Record> searchResults(String surname) throws SQLException {
  List<Record> records = new ArrayList<>();

  // String query = String.format(SEARCH_QUERY, surname);
  try (PreparedStatement stmt = database.prepareStatement(SEARCH_QUERY)) {
    // ResultSet results = stmt.executeQuery(query);

    stmt.setString(parameterIndex: 1,surname);

    try(ResultSet results = stmt.executeQuery()) {
      while (results.next()) {
        Record rec = new Record();
        rec.setSurname(results.getString(columnIndex: 2));
        rec.setForename(results.getString(columnIndex: 3));
        rec.setAddress(results.getString(columnIndex: 4));
        rec.setDateOfBirth(results.getString(columnIndex: 5));
        rec.setDoctorId(results.getString(columnIndex: 6));
        rec.setDiagnosis(results.getString(columnIndex: 7));
        records.add(rec);
      }
    }
  }
```

Figure 13.

4.4. Flaw 4 – Brute Force Vulnerability

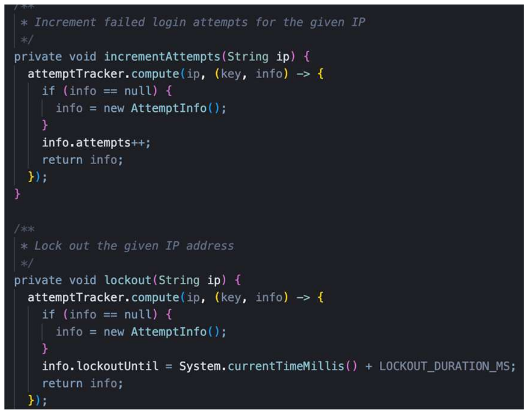
Setup Attempt Tracking Infrastructure: The implementation adds a thread-safe tracking mechanism using a `ConcurrentHashMap` to store failed login attempts per IP address. Constants define the maximum attempts (5) and lockout duration (15 minutes), while an inner `AttemptInfo` class tracks the attempt count and lockout expiration timestamp for each IP.



```java
// Attempt tracking constants
private static final int MAX_ATTEMPTS = 5;
private static final long LOCKOUT_DURATION_MS = 15 * 60 * 1000; // 15 minutes in milliseconds

private final Configuration fm = new Configuration(Configuration.VERSION_2_3_28);
private Connection database;


  // Inner class to track attempt information
private static class AttemptInfo {
  int attempts;
  long lockoutUntil; // timestamp when lockout expires (0 if not locked out)

  AttemptInfo() {
    this.attempts = 0;
    this.lockoutUntil = 0;
  }
```

Figure 14.

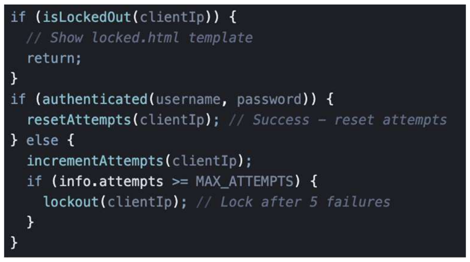
Implemented Core Protection Methods: Helper methods manage attempt tracking: `getClientIp()` extracts the client IP from request headers, `incrementAttempts()` increases the failure count, `lockout()` sets the expiration timestamp after max attempts, `isLockedOut()` checks if an IP is currently locked, and `resetAttempts()` clears attempts on successful login or after expiration.



```java
/**
 * Increment failed login attempts for the given IP
 */
private void incrementAttempts(String ip) {
  attemptTracker.compute(ip, (key, info) -> {
    if (info == null) {
      info = new AttemptInfo();
    }
    info.attempts++;
    return info;
  });
}

/**
 * Lock out the given IP address
 */
private void lockout(String ip) {
  attemptTracker.compute(ip, (key, info) -> {
    if (info == null) {
      info = new AttemptInfo();
    }
    info.lockoutUntil = System.currentTimeMillis() + LOCKOUT_DURATION_MS;
    return info;
  });
```

Figure 15.

Integrate Protection into Authentication Flow: The protection is integrated into `doPost()` before authentication, it checks if the IP is locked and blocks further attempts. On failed login, it increments attempts and triggers a lockout html file after 5 failures. On successful login, it resets attempts. The `doGet()` method also checks lockout status and displays remaining attempts or a lockout message.

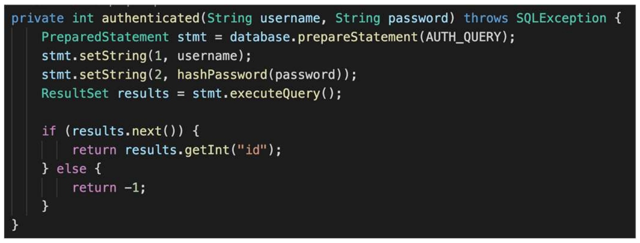


```java
if (isLockedOut(clientIp)) {
  // Show locked.html template
  return;
}
if (authenticated(username, password)) {
  resetAttempts(clientIp); // Success - reset attempts
} else {
  incrementAttempts(clientIp);
  if (info.attempts >= MAX_ATTEMPTS) {
    lockout(clientIp); // Lock after 5 failures
  }
}
```

Figure 16.

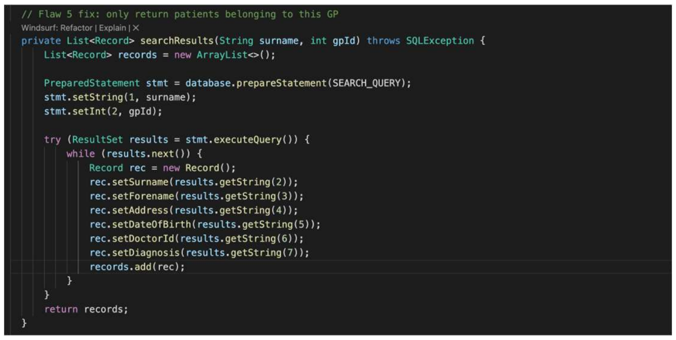
4.5. Flaw 5 - Broken Access Control

The search functionality was updated so that database queries restrict results to the authenticated GP’s id. This was done by modifying the authentication method to return the GP’s user id and updating the search query to include an additional parameter that enforces patient.gp_id = GP.id. As a result, logged–in users now only receive data for patients they are registered to treat. This prevents horizontal privilege escalation and maintains confidentiality of medical records between GPs.



```java
private int authenticated(String username, String password) throws SQLException {
    PreparedStatement stmt = database.prepareStatement(AUTH_QUERY);
    stmt.setString(1, username);
    stmt.setString(2, hashPassword(password));
    ResultSet results = stmt.executeQuery();

    if (results.next()) {
        return results.getInt("id");
    } else {
        return -1;
    }
}
```

Figure 17.



```java
// Flaw 5 fix: only return patients belonging to this GP
Windsurf: Refactor | Explain | ×
private List<Record> searchResults(String surname, int gpId) throws SQLException {
    List<Record> records = new ArrayList<>();

    PreparedStatement stmt = database.prepareStatement(SEARCH_QUERY);
    stmt.setString(1, surname);
    stmt.setInt(2, gpId);

    try (ResultSet results = stmt.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;
}
```

Figure 18.

5. References

Wiggins, A. 2017. *The Twelve-Factor App.* [Online]. [Accessed 9 November 2025]. Available from: https://12factor.net/config

Garfield, L. 2024. What is .env?. 10 September 2024. *Product.* [Online]. [Accessed 9 November 2025]. Available from: https://upsun.com/blog/what-is-env-file/

Java hashing documentation with MessageDigest
https://docs.oracle.com/javase/8/docs/api/java/security/MessageDigest.html