UiA Fakultet for teknologi og realfag

Assignment 4

by

Group 9

in

IKT222
Softwaresikkerhet

Grimstad, 2025

Contents

List of Figures

Listinger

1 Introduction

This report will present the results of a security assessment performed on a patient record web application. The objective of the assessment is to identify, verify and remediate common security vulnerabilities affecting both the web interface and the underlying server-side code. The focus is on issues that can compromise the confidentiality, integrity, and availability of sensitive patient data.

The testing process is split into two sections. First, we will delve into vulnerabilities found through the web interface using manual testing techniques and browser developer tools [1][2]. The second section will involve vulnerabilities that cannot be found through the web interface itsself, but rather through the project files.

The analysis revealed several key vulnerabilities. Through the web interface, the application was found to transmit user credentials in plaintext and lacked proper session management. Within the codebase, passwords were stored without hashing, and SQL queries were constructed through string concatenation, exposing the system to injection attacks. These findings align with multiple OWASP Top 10 categories, including *A02:2021 – Cryptographic Failures*, *A03:2021 – Injection*, and *A07:2021 – Identification and Authentication Failures*.

Finally, we will mitigate one vulnerability from each section with thorough documentation. Our chosen vulnerabilities are fixing the absence of HTTPS enforcement in the web layer, and the plaintext password storage in the database. Each fix is explained with code examples, justification, and post-fix verification steps to confirm the effectivemess of the solution.

2 Security vulnerabilities triggered through the web interface

2.1 Vulnerability 1: Missing session cookie / weak session management

Description: We submitted a valid login request (POST `/search`) and observed that the application does not set any server session cookie for the origin `http://localhost:8080`[3][2]. Browser DevTools shows only third-party cookies originating from external CDNs. The absence of a server-issued session cookie suggests the application does not create server-side sessions or does not set cookie flags correctly (HttpOnly, Secure, SameSite).

How is it found (reproduction):

1. Start the server through Powershell at the patients file from the project with `./gradlew.bat run` and open Chrome and navigate to `http://localhost:8080`.
2. Open Developer Tools (F12) and go to `Network` and `Application` → Cookies.
3. Submit a valid login via the web form (for example `username=alice`, `password=qwertyui`, `surname=Smith`).
4. Inspect the response headers of the POST request and the cookie list in `Application` — no application session cookie is present.

Evidence: `login_headers.png` — POST /search request headers and response. `cookies_no_setcookie.png` — Application → Cookies showing no session cookie for `localhost:8080`.

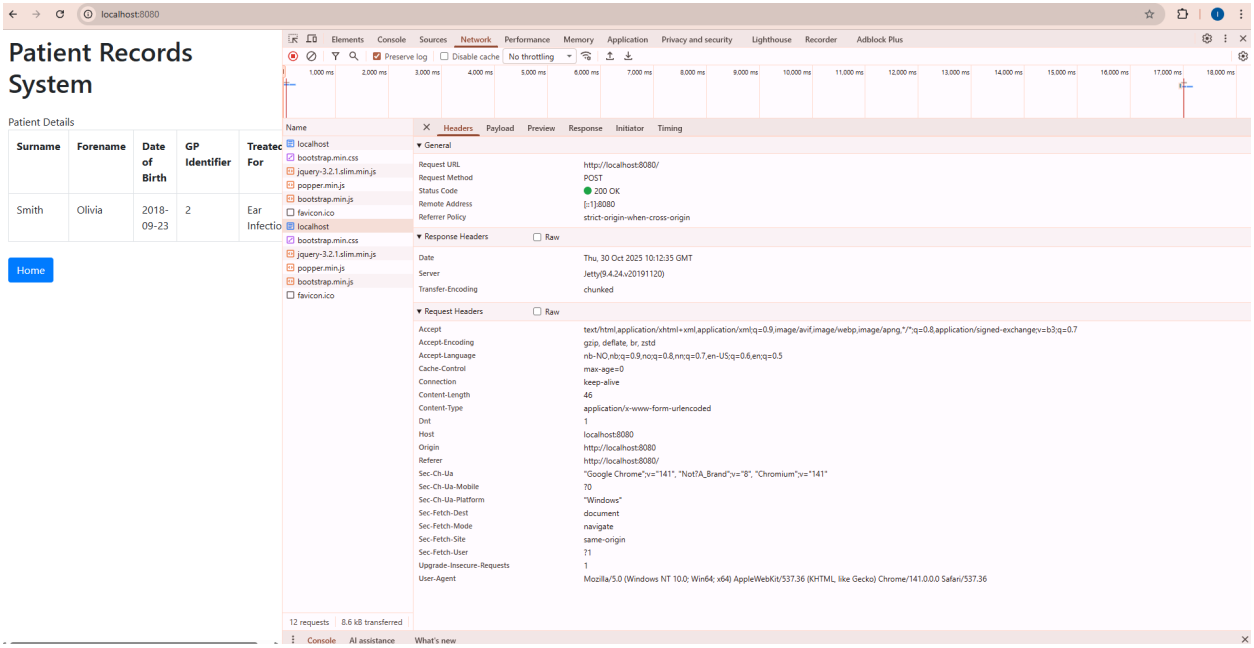

Figure 1: POST /search headers (login attempt).

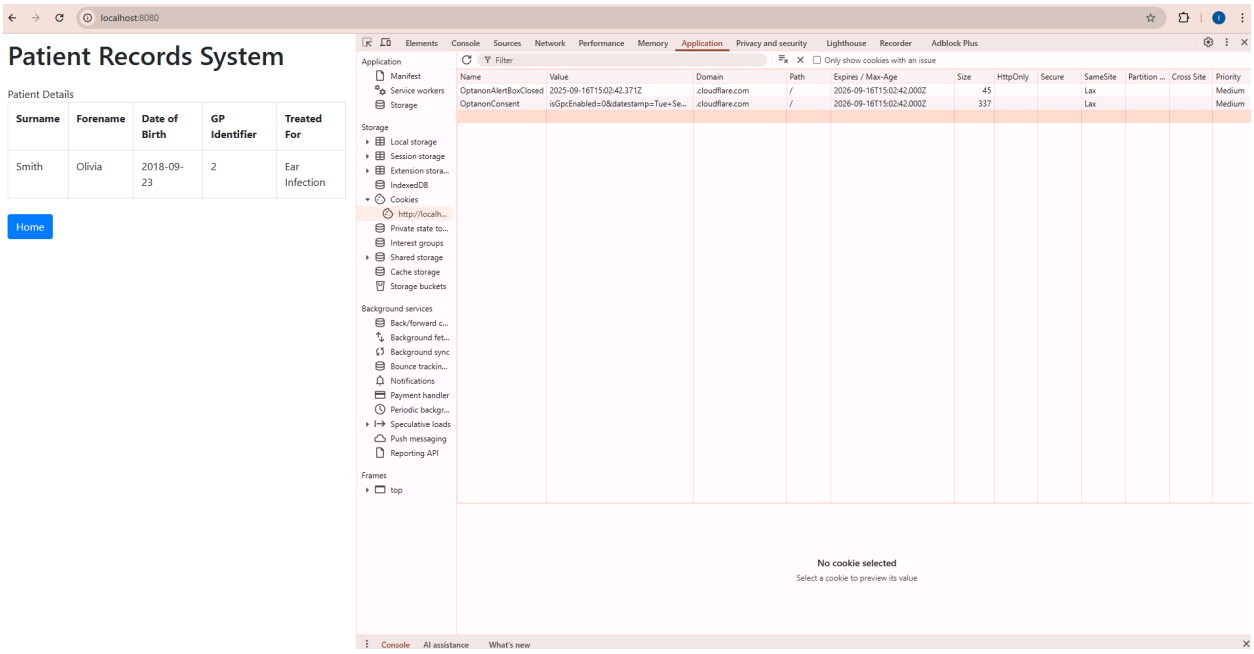

Figure 2: Application storage (Cookies) showing no server session cookie for `localhost:8080`.

Impact / Consequence: When the server is not configured properly thorugh server sessions and cookie flags, the application is vulnerable to session management issues such as session fixation, lack of session invalidation on logout, and theft of authentication state. Additionally, the absence of `HttpOnly` and `Secure` attributes (once cookies are introduced) would increase the risk that cookies are accessed by client scripts or leaked over insecure channels.

Mitigation / Recommendation:

- Implement server-side session management (e.g., create an `HttpSession` after authentication).
- Set session cookie attributes: `HttpOnly=true`, `Secure=true` (when using HTTPS) and a restrictive `SameSite` policy (e.g., `Lax` or `Strict` as appropriate).
- Ensure session identifiers are regenerated on privilege changes (login) to mitigate session fixation.
- Implement session timeout and explicit logout to invalidate sessions server-side.

2.2 Vulnerability 2: Web-exposed input (SQLi vector)

Description: The login and search forms send the parameters `username`, `password` and `surname` via POST to `/search`[3][2]. We demonstrated that the server accepts URL-encoded payloads via both the browser and `curl`. Our particular test payloads returned `No records found.`, but the combination of evidence from network requests and later source analysis (see code section) indicates the backend constructs SQL queries using string interpolation, which is a high-risk pattern for SQL injection.
How is it found (reproduction):

1. Using Chrome DevTools: record the POST request when submitting the login/search form and inspect the `Payload` tab (shows `username`, `password`, `surname` fields).

2. Use `curl` to send crafted payloads and capture responses (examples below in Evidence).

Example `curl` tests (executed locally):

```
curl -i -X POST "http://localhost:8080/search" \
  --data-urlencode "username=alice" \
  --data-urlencode "password=' OR '1'='1" \
  --data-urlencode "surname="
```

Evidence: `network_request_alice.png` — shows the exact form fields captured in DevTools. `sqli_alice.txt` — raw output of the `curl` attempt.

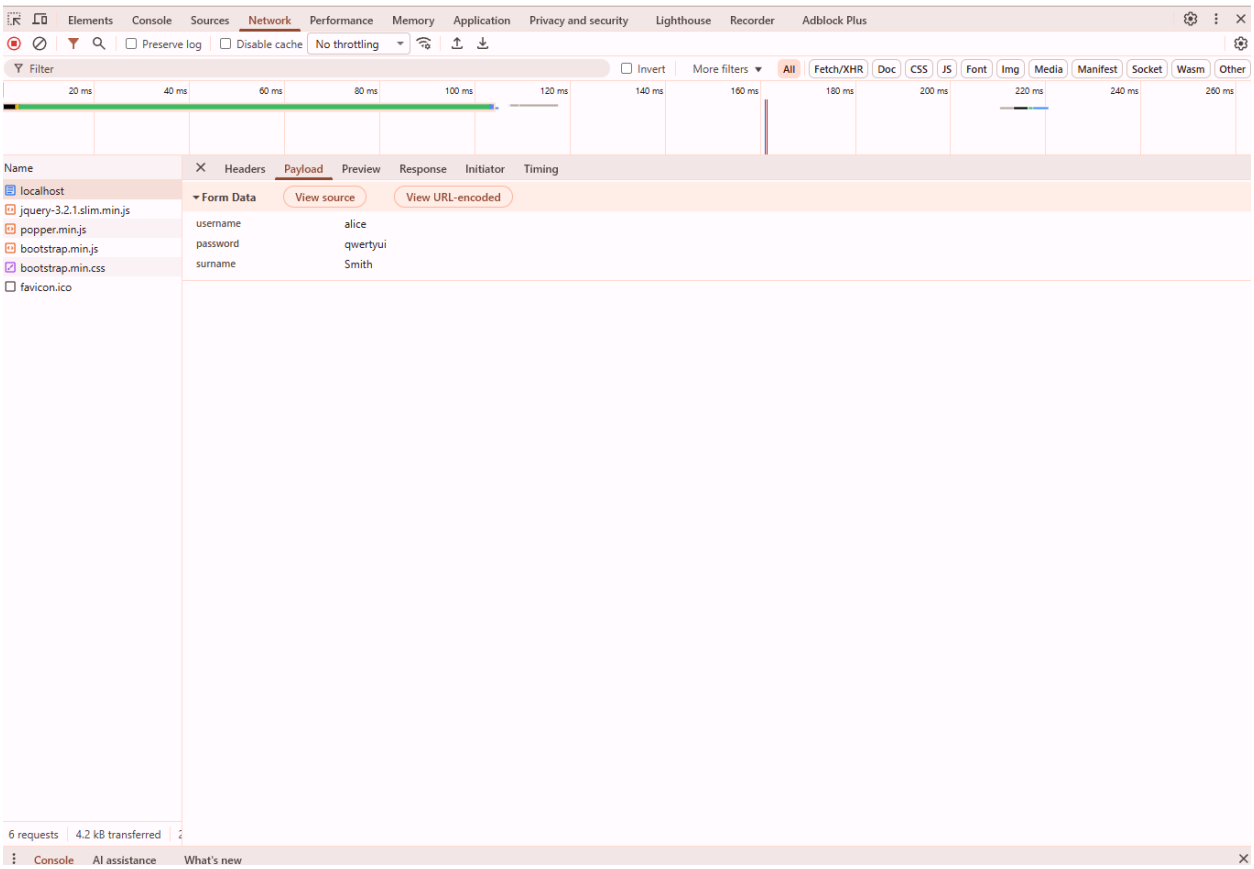

Figure 3: Network payload captured from browser: username, password, surname.

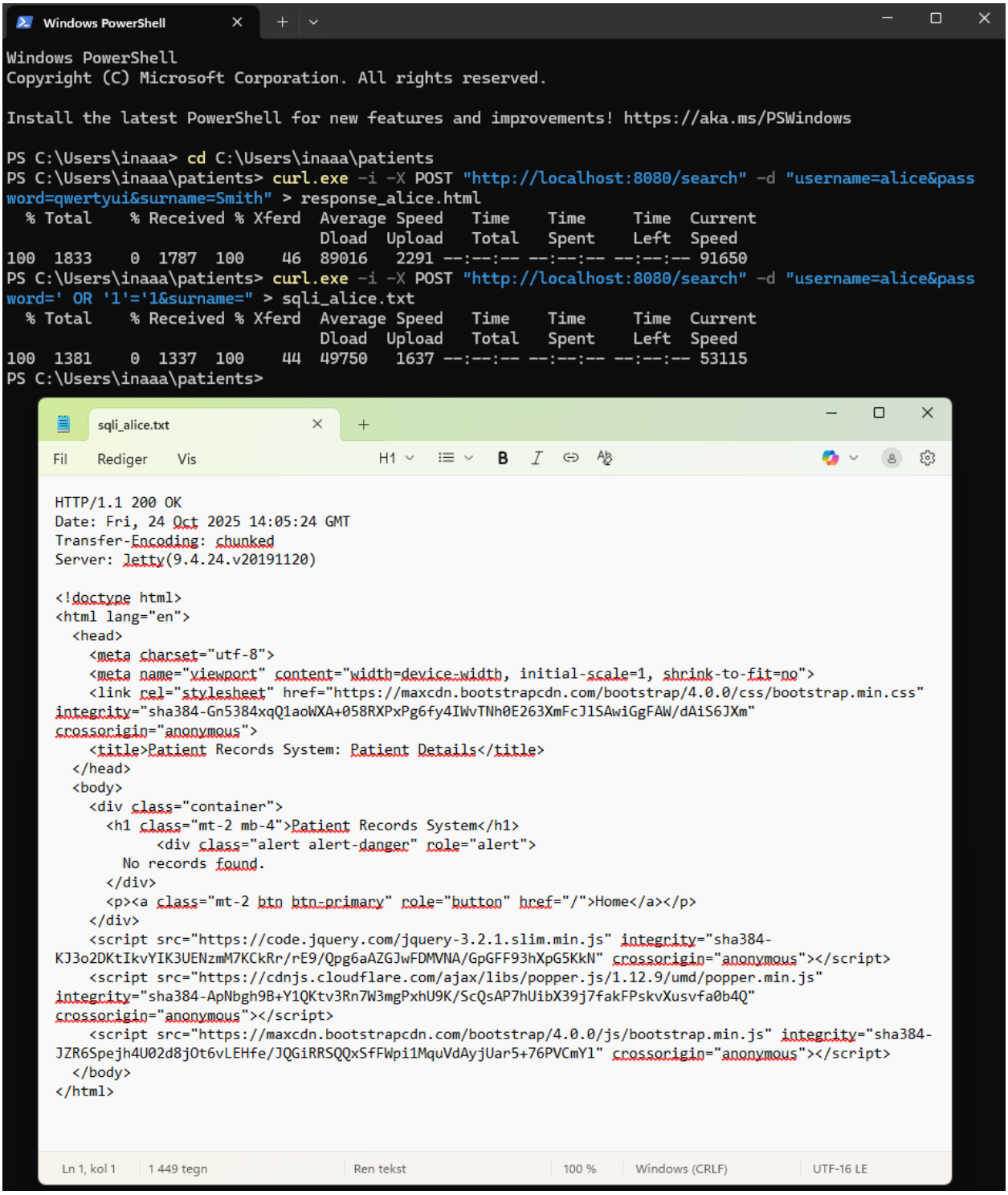

Figure 4: Raw HTTP response saved from a SQLi test using `curl`.

Impact / Consequence: If the server-side SQL statements are constructed by concatenating user input

directly into SQL (see code analysis), a successful SQL injection could allow an attacker to bypass authentication, extract sensitive patient data, or modify the database.
Mitigation / Recommendation:

- Use parameterised queries (e.g., `PreparedStatement` in JDBC) for all database access.
- Validate and normalise input on the server side (length, character sets, allowed values).
- Apply least privilege to the database account used by the application.
- Consider an input sanitisation library and enable auditing/IDS for suspicious input patterns.

2.3 Vulnerability 3: Credentials sent in cleartext (missing HTTPS)

Description: The login form transmits username and password as form data to `http://localhost:8080`[4][5][2]. This traffic travels in plaintext and can be intercepted by attackers on the same network.
How is it found (reproduction):

1. Submit the login form from the browser.
2. Inspect the network request: the Request URL shows `http://localhost:8080` and the form payload contains plain credentials (see evidence images).

Evidence: `login_headers.png`, `login_payload.png`.

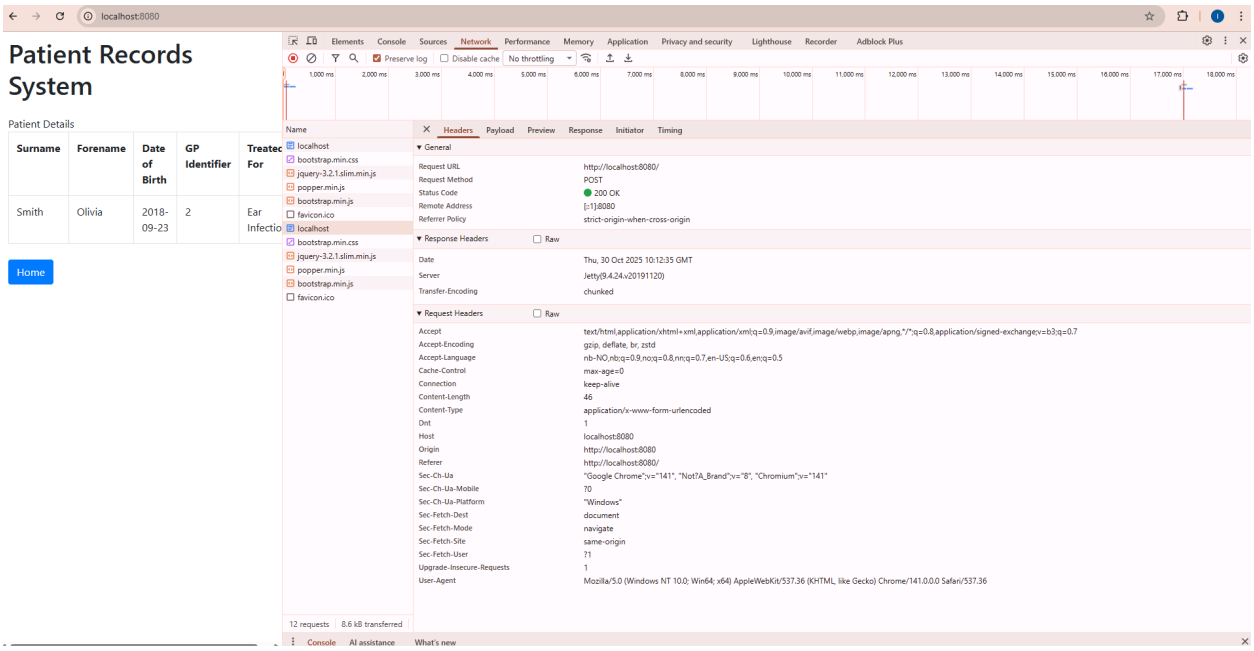

Figure 5: HTTP POST request headers for login. This proves the login endpoint uses unencrypted HTTP.

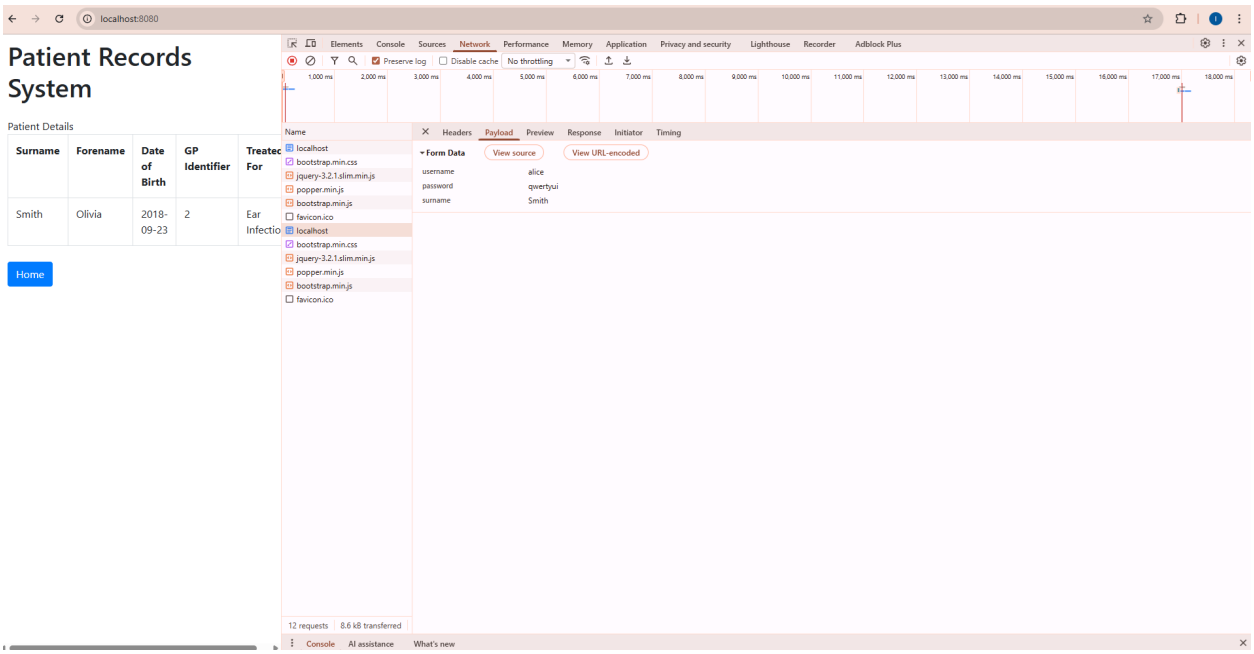

Figure 6: Login form payload (DevTools Payload): the form fields `username` and `password` are visible in cleartext in the POST body.

Impact / Consequence: Credentials captured in transit allow direct account compromise and subsequent unauthorized access to patient data. This is a critical confidentiality weakness and violates standard web application security practices (TLS required for authentication/PHI).
Mitigation / Recommendation:

- Enforce HTTPS for all endpoints, especially authentication and data endpoints.
- Redirect HTTP requests to HTTPS (server configuration).
- After enabling HTTPS, set session cookies with the `Secure` flag and use `HttpOnly` and `SameSite`.
- Use HSTS in production to prevent protocol downgrade attacks.

3 Discovering Vulnerabilities Outside the Website View

We used OWASP's publications to pick out some vulnerabilities we wanted to look for through their Security Testing Guide [6].

3.1 Vulnerability 1 - Passwords in Plaintext

This vulnerability was found through the same `grep` search on Git Bash as vulnerability 3 [7]. See that chapter for the screenshot of the code used, and the code found in the project files.

Through a `grep` search on Git Bash we discovered the output line:

```
String password = request.getParameter("password");
```

Due to the username and password in the previous line of code being set as '%s', it was more than likely that both usernames and passwords were being stored as plaintext. To verify this vulnerability, it required a deeper dive into the potential issue. To do this we had to access the database, and see for ourselves if the program was storing them as plaintext, by using some simple python code. Because of an issue with accessing sqlite3 through the terminal, we got some help from ChatGPT[2] to figure out a different solution to viewing the tables, and ended up using python in Git Bash to see them. Below is the code used to access and view the information [8]:

Code to access the correct table in the database:

```
import sqlite3
conn = sqlite3.connect("db.sqlite3")
cur = conn.cursor()

cur.execute("SELECT name FROM sqlite_master WHERE type='table';")
tables = [r[0] for r in cur.fetchall()]
print("TABLES:", tables)

cur.execute("PRAGMA table_info('user')")
rows = cur.fetchall()

print("\nSchema for user table:")
for row in rows:
    print(row)

print("\nSample stored passwords:")
cur.execute("SELECT username, password FROM user LIMIT 5;")
for row in cur.fetchall():
    print(row)
```

Code to check how usernames and passwords are stored:

```
cur.execute("SELECT username, password FROM user LIMIT 5;")
print(cur.fetchall())
```

The picture below shows the output we recieved from delving into the database, and as we can see on the output, the usernames and passwords are shown as plaintext.

```
>>> cur.execute("SELECT username, password FROM user LIMIT 5;")
<sqlite3.Cursor object at 0x0000023D8F030B20>
>>> print(cur.fetchall())
[('admin', 'password'), ('johnt', '12345678'), ('alice', 'qwertyui'), ("admin'--", 'testpass'), ('support', 'letm
ein!')]
>>>
```

Figure 7: Usernames and Passwords Stored as Plaintext

3.2 Vulnerability 2 - No secure error handling / logging

We decided to look for vulnerabilities dealing with the lack of proper error handling. To do this, we used another `grep` function [7] to see the exception errors created (the code line was too long for overleaf, so an extra space was added between 'info' and '|e', but should not be included in git bash):

```
grep -R -I -n --exclude-dir=.git -E
↪ "(printStackTrace|System\.out\.print|System\.err\.print|logger\.debug|logger\.info
↪ |e\.printStackTrace|Exception\s+e" .
```

The result gave us multiple catch blocks, showing us the different exception errors provided:

```
./src/main/java/IKT222/Assignment4/AppServlet.java:48:    } catch (IOException error) {
./src/main/java/IKT222/Assignment4/AppServlet.java:56:    } catch (SQLException error) {
./src/main/java/IKT222/Assignment4/AppServlet.java:69:    } catch (TemplateException error) {
./src/main/java/IKT222/Assignment4/AppServlet.java:95:    } catch (Exception error) {
```

Figure 8: Catch Exceptions Discovered Through `grep`

We decided to further inspect one of the catch exceptions to prove that the exception is not being logged or handled. In doing so, we found:

```
catch (Exception error) {
    response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
}
```

Here we can see multiple issues with the exception. First, the exception `error` is declared but never used, which means the program actually just discards the error after it has occurred. It also shows that there is no logging statement, which would usually look something like `logging.error("Error Message", error);`. This tells us that the fact that the error happened is not being logged anywhere. Finally, the only function of this code is to send a regular 500 error back to the user when the error occurs, and this is due to the `response.sendError` section of the code.

As a result of no error handling or logging, admins cannot see any records of errors ever occuring, which means they are unable to perform monotoring or auditing of security events. This vulnerability allows hackers to hide their activities, and makes it a lot more challenging to make incident reports [9].

3.3 Vulnerability 3 - SQL injection through Broken Authentication

NB! This vulnerability was 'fixed' as a result of actively fixing the plaintext vulnerability. It is mentioned in the mitigation chapter.

Vulnerability 3 was found through a `grep` command [7] searching for hardcoded passwords, by using expressions to look for matches to 'password', 'secret' an api key in three forms (because it can vary with projects), and 'token'. The exact command used was:

```
grep -i -R -n -E "password|secret|api[_-]?key|token" .
```

This was performed from the root of the project and looks through all the files and subdirectories. After running this command in Git Bash, we actually discovered a different vulnerability, shown in the image below.

```
ninal@Mordi MINGW64 ~/ss_4/patients/patients
$ grep -i -R -n -E "password|secret|api[_-]?key|token" .
Binary file ./db.sqlite3 matches
./src/main/java/IKT222/Assignment4/AppServlet.java:29:  private static final String AUTH_QUERY
= "select * from user where username='%s' and password='%s'";
./src/main/java/IKT222/Assignment4/AppServlet.java:79:    String password = request.getParamete
r("password");
./src/main/java/IKT222/Assignment4/AppServlet.java:83:      if (authenticated(username, passwor
d)) {
./src/main/java/IKT222/Assignment4/AppServlet.java:100:  private boolean authenticated(String u
sername, String password) throws SQLException {
./src/main/java/IKT222/Assignment4/AppServlet.java:101:    String query = String.format(AUTH_QU
ERY, username, password);
./templates/login.html:25:              <label for="pwd">Your Password</label>
./templates/login.html:26:              <input id="pwd" type="password" class="form-control" na
me="password">
```

Figure 9: First vulnerability search

Although there were no obvious hard-coded passwords or secrets that showed up, we discovered the line:

```
private static final String AUTH-QUERY = "select * from user where username='%s' and
↪  password='%s'";
```

This line of code proves a major vulnerability in the project, because both the username and password a user inputs gets put into an SQL quary string, which can be bypassed. For example, we can modify the SQL query in both the name input and the password input by writing `' OR '1'='1`, and the login function would accept this as a valid password [10]. This seems to be a very well documented vulnerability, and can be found in OWASP's Broken Web Application project.

4 Fixing Security Vulnerabilities

4.1 Fix of Vulnerability discovered through the web interface: Vulnerability 1: Missing session cookie / weak session management

Description

During testing2.1, it was discovered that the web application transmitted user credentials and patient data via unencrypted HTTP connections[3][2]. This vulnerability could allow an attacker with network access to intercept sensitive information (such as usernames and passwords). The flaw directly affects the **confidentiality** of the system and exposes users to man-in-the-middle (MITM) attacks[11].

Root Cause

The root cause of this issue was that the Jetty web server configuration did not enforce HTTPS connections[12]. Although HTTPS was technically available, users could still connect via plain HTTP. The absence of automatic redirection and secure session handling further increased the risk of credential leakage.

Implementation of the Fix

To eliminate this vulnerability, HTTPS was fully enforced at both the server and application levels. The implementation consists of two key components:

1. Server-side HTTPS enforcement: The `AppServer.java` class was updated to configure an HTTPS connector using Jetty's `SslConnectionFactory`[13]. A self-signed certificate (`keystore.jks`) was generated using the `keytool` utility, and the Jetty server was configured to listen on port `8443` for secure connections. An additional HTTP listener on port `8080` was created solely to redirect all incoming HTTP traffic to the HTTPS endpoint.

```
Server httpServer = new Server(8080);
httpServer.setHandler((target, baseRequest, request, response) -> {
    String redirectURL = "https://localhost:8443" + target;
    response.setStatus(HttpServletResponse.SC_MOVED_PERMANENTLY);
    response.setHeader("Location", redirectURL);
    baseRequest.setHandled(true);
});
```

This ensures that even if a user attempts to access the application over HTTP, they are automatically redirected to a secure HTTPS connection.

2. Application-level HTTPS validation: To reinforce the protection, a verification step was added to the `AppServlet.java` class to ensure that sensitive requests are only processed when using HTTPS[13]. If an insecure connection is detected, the user is immediately redirected:

```
if (!request.isSecure()) {
    response.sendRedirect("https://" + request.getServerName()
        + ":8443" + request.getRequestURI());
    return;
}
```

3. Handling of Browser Warnings: When accessing the application locally via `https://localhost:8443`, modern browsers display a warning stating that the connection is not private (`NET::ERR_CERT_AUTHORITY_INVALID`)[14]. This behavior is expected since the SSL certificate (`keystore.jks`) is self-signed and not issued by a trusted Certificate Authority (CA). In a production environment, a valid CA-signed certificate (e.g., Let's Encrypt) would remove this warning entirely.

Security Impact and Risk Mitigation

The introduction of HTTPS significantly improves the system's overall security posture:

- Prevents **man-in-the-middle attacks** by ensuring all communication is encrypted using TLS.
- Protects **user credentials** and **session data** from eavesdropping.
- Guarantees **data confidentiality and integrity** between the client and the server.
- Implements secure-by-default principles and aligns with OWASP A02:2021 (*Cryptographic Failures*).
- Demonstrates **Defense in Depth** by enforcing encryption both at the transport and application levels.

Testing and Verification

After implementing the fix, the system was tested using both HTTP and HTTPS connections:

- Any HTTP request to port `8080` was automatically redirected to `https://localhost:8443`.
- The browser's developer tools confirmed that all traffic now used the `https://` protocol.
- All sensitive form submissions (such as login credentials) were encrypted via TLS.

Although Chrome displays a warning about the self-signed certificate, this does not affect the encryption itself. The TLS handshake was successfully established, confirming that the connection is indeed encrypted and secure. Therefore, the vulnerability of missing HTTPS enforcement is considered fully mitigated.

4.2 Fixing Security Vulnerability discovered through the code and database examination: Vulnerability 1 - Passwords in Plaintext

For this section, we decided to fix the PlainText vulnerability found within the project code. Steps to reproduce this vulnerability are described in section *Vulnerability 1 - Passwords in Plaintext* 3.1, and the same steps can be re-run after applying the fix to confirm it is no longer exploitable. ChatGPT was used as help for understanding the functionality of `MessageDigest` and `NoSuchAlgorithmException` [2], as well as the documentation found through the documentation found on docs.oracle.com, spesifically for these functionalities [15].

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

private String hashPassword(String password) {
    try {
      MessageDigest md = MessageDigest.getInstance("SHA-256");
      byte[] hash = md.digest(password.getBytes());
      StringBuilder hex = new StringBuilder();
      for (byte b : hash) {
        hex.append(String.format("%02x", b));
      }
      return hex.toString();
    } catch (NoSuchAlgorithmException e) {
      throw new RuntimeException(e);
    }
  }

private void hashExistingPasswords() throws SQLException {
    var select = database.createStatement();
    var rs = select.executeQuery("SELECT username, password FROM user");

    var update = database.prepareStatement("UPDATE user SET password=? WHERE
    ↪  username=?");

    while (rs.next()) {
      String user = rs.getString("username");
      String pass = rs.getString("password");

      if (!pass.matches("(?i)^[0-9a-f]{64}$")) {
        String hashed = hashPassword(pass);
        update.setString(1, hashed);
        update.setString(2, user);
        update.addBatch();
      }
    }
    update.executeBatch();
  }

@Override
  public void init() throws ServletException {
      configureTemplateEngine();
      connectToDatabase();
      // try {
```

```
43              // hashExistingPasswords();
44          // } catch (SQLException e) {
45              // throw new ServletException("Error hashing existing passwords", e);
46          // }
47      }
48
49      private boolean authenticated(String username, String password) throws SQLException {
50        String query = String.format(AUTH_QUERY, username, hashed);
51        try (Statement stmt = database.createStatement()) {
52          ResultSet results = stmt.executeQuery(query);
53          return results.next();
54        }
55      }
```

Mitigation Process

Mitigating the plaintext vulnerability required modification of the already existing `AppServlet.java` file to add a hasing function for the passwords. To do this, we used documentation from docs.oracle.com on most of the functionality.

We decided to import MessageDigest which lets us use the well-documented SHA-256 to hash our passwords through a one-way function, which means the original input cannot be determined from the current hash [15]. Our chosen method of hashing is rather simple, and did not require much editing of the already written code.

We had to create a function that actually did the password hashing, which we called `hashPassword()`, where we use a SHA-256 algoritm [15] to hash a plaintext password. The users chosen password is run through the algorithm which creates a byte array, then converts the bytes into a final 64 character hexed string that is then stored in the database. This was a rather easy solution, but after checking the database, we realised that we had only created the hashing function for when a user is created, and because of this, the passwords currently stored in the database were still seen as PlainText. To fix this, we created a method that only requires to run once, called `hashExistingPasswords()`. This method simply checks all the existing passwords in the database, and uses our previous function to hash them (if the password is not already a 64 character hex).

Finally, we had to actually use the method to change the current PlainText passwords, by restructuring the `@Override` function and calling our method. After building the code, we removed (commented out for documentation purposes) the section that hashes existing passwords as it was no longer needed. Finally, one important step to remember, in the function `authenticated` you must replace the `String.format` to support the new format, by replacing 'password' with 'hashed'. If this is not changed, once passwords are hashed, the server will still be checking if the password input matches plaintext, and will give you the message that the credentials are invalid.

Extra note: Since the hashing function hashes the passwords before it checks the password written with the database, we 'accidently' solved the vulnerability logging in with `' OR '1'='1`. Therefore, to replicate that specific vulnerability, it must be done before implementing the mitigation for plaintext passwords in the database.

5 Conclusion

This assignment successfully identified and resolved multiple vulnerabilities affecting the patiend record web application, both discovered through the web interface, and through the project files. The findings proved that the system lacked crucial protections from common, well-documented vulnerabilities.

Through searching for common vulnerabilities, mostly ones mentioned by OWASP, the most critical issues were addressed. HTTPS enforcement now ensures that all communication between the client and server is encrypted, prevented credential interception and man-in-the-middle attacks. Furthermore, the implementation of SHA-256 hashing for password storage, including an option to hash currently existing passwords without the need to re-create the users, eliminated plaintext credential exposure, greatly improving data confidentiality.

The report highlights the importance of integrating secure coding principles throughout the software development lifecycle. Using prepared statements, enforcing TLS, setting secure cookie attributes, and implementing proper error handling are all important to building secure applications that protect sensitive data. Continued security testing, monotoring and adherance to OWASP guidelines will help maintain the imporived security messures.

Overall, the project demonstrates a clear understanding of web application security fundamentals and practical mitigation strategies. The applied fixes not only resolved a couple of the discovered vulnerabilities, but also strengthened the application's overall security architecture.

References

[1] "41 Common Web Application Vulnerabilities Explained." [Online; Accessed 27.10.2025]. [Online]. Available: https://securityscorecard.com/blog/common-web-application-vulnerabilities-explained/.

[2] "ChatGPT." [Online; Accessed 31.10.2025]. [Online]. Available: https://chatgpt.com/.

[3] "Security Cookies: Cookie Attributes and Vulnerability Guide." [Online; Accessed 29.10.2025]. [Online]. Available: https://www.invicti.com/white-papers/security-cookies-whitepaper.

[4] "Cleartext submission of password." [Online; Accessed 30.10.2025]. [Online]. Available: https://portswigger.net/kb/issues/00300100_cleartext-submission-of-password.

[5] "CWE-319: Cleartext Transmission of Sensitive Information." [Online; Accessed 30.10.2025]. [Online]. Available: https://cwe.mitre.org/data/definitions/319.html.

[6] "OWASP Web Security Testing Guide." [Online; Accessed 30.10.2025]. [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/.

[7] "git-grep." [Online; Accessed 29.10.2025]. [Online]. Available: https://git-scm.com/docs/git-grep.

[8] "Open databse files ().db) using python." [Online; Accessed 01.11.2025]. [Online]. Available: https://stackoverflow.com/questions/62340498/open-database-files-db-using-python.

[9] "OWASP Top 10:2021." [Online; Accessed 29.10.2025]. [Online]. Available: https://owasp.org/Top10/A09_2021-Security_Logging_and_Monitoring_Failures/.

[10] "Using SQL Injection to Bypass Authentication." [Online; Accessed 30.10.2025]. [Online]. Available: https://portswigger.net/support/using-sql-injection-to-bypass-authentication.

[11] "Man-in-the-Middle Attack: Types And Examples." [Online; Accessed 31.10.2025]. [Online]. Available: https://www.fortinet.com/resources/cyberglossary/man-in-the-middle-attack.

[12] "Secure Jetty with HTTPS." [Online; Accessed 31.10.2025]. [Online]. Available: https://docs.opennms.com/meridian/2024/operation/deep-dive/admin/configuration/https/https-server.html.

[13] "Redirect from http to https automatically in embedded jetty." [Online; Accessed 31.10.2025]. [Online]. Available: https://stackoverflow.com/questions/26123604/redirect-from-http-to-https-automatically-in-embedded-jetty.

[14] "Configure SSL on Jettyy." [Online; Accessed 31.10.2025]. [Online]. Available: https://stackoverflow.com/questions/4008837/configure-ssl-on-jetty.

[15] "Class MessageDigest." [Online; Accessed 30.11.2025]. [Online]. Available: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/security/MessageDigest.html.

[16] "Regex to check string contains only Hex Characters." [Online; Accessed 31.11.2025]. [Online]. Available: https://stackoverflow.com/questions/5317320/regex-to-check-string-contains-only-hex-characters.

[17] "Exposed Web Interfaces." [Online; Accessed 30.10.2025]. [Online]. Available: https://www.threatngsecurity.com/glossary/exposed-web-interfaces.