# COMP3911 Coursework

Anche Liu
sc23al2 & 201703057

Han Lin
sc22hl & 201690928

Shaohui Yang
sc22sy & 201690924

Shirui Zhao
sc23sz & 201743232

Yuankun Zhou
sc222yz & 201690952

# 1 Analysis of Security Flaws

• Cross-site scripting (XSS) is a security vulnerability that allows attackers to inject malicious scripts that can be executed in other users' browsers, thereby stealing information, tampering with pages, or hijacking sessions.

• The patient search function builds SQL queries using string concatenation, allowing an attacker to inject input such as ' OR '1'='1 to bypass filtering and retrieve all patient records.

• Plaintext password storage means user passwords are saved in the database without encryption or hashing, allowing anyone with database access to read them directly.

• Lack of transport encryption means the application sends login credentials and patient data over plain HTTP, allowing an attacker on the same network to eavesdrop on and capture sensitive information in transit.

• Broken access control means that the system does not restrict patient records to the assigned GP, allowing horizontal privilege escalation between users.

## 1.1 Cross-Site Scripting (XSS)

Cross-Site Scripting occurs in reflected, stored, and DOM-based forms. In this application, stored XSS arises because user-provided fields—such as forename, surname, and address—are stored in the database and later rendered into HTML without any escaping. As a result, an attacker can inject `script` tags that execute automatically for any user viewing the affected record, enabling credential theft, malicious UI modifications, or unauthorized redirection. Source code inspection confirms that database values are output without sanitisation, for example:

```
rec.setSurname(results.getString(2)); // No HTML escaping
```

This demonstrates that untrusted input is passed directly from the database into the generated HTML, enabling stored XSS.

To verify the vulnerability, the following malicious patient entry was directly inserted into the `patient` table:

```
INSERT INTO patient (surname, forename, address, born, gp_id, treated_for)
VALUES('Attacker', '<script>alert("XSS")</script>',
       'Nowhere', '1990-01-01', '1', 'none');
```

When a user enters `Attacker` as the surname in the search field, the application loads the database entry and injects the unescaped `<script>` tag into the HTML response. This triggers immediate execution of the JavaScript payload.

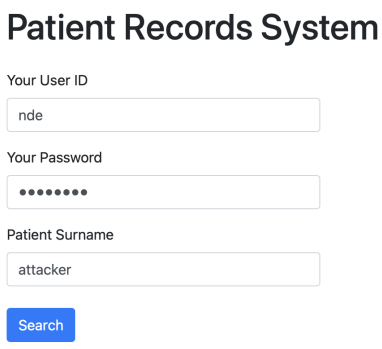


Figure 1: Search interface where the attacker inputs the surname "Attacker".

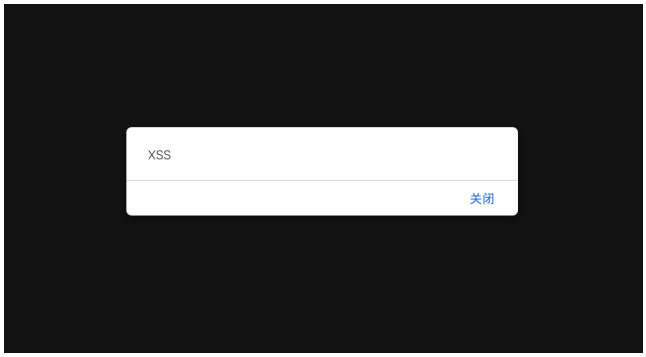


Figure 2: Browser executing the stored XSS payload via an injected ¡script¿ element.

Stored XSS is highly dangerous because the injected payload is persistently stored and automatically executed for every user who views the affected record. This allows attackers to steal credentials, manipulate or spoof UI elements, redirect users to malicious sites, and inject phishing content. The impact can further extend to other pages or APIs that display the compromised fields, and could even enable session hijacking if session management is later added.

## 1.2 SQL Injection in Patient Search Function

The application constructs SQL queries by directly concatenating untrusted user input into SQL, which means the `surname` parameter is inserted directly into the SQL query without any validation or sanitisation, an attacker can supply malicious SQL fragments that modify the intended behaviour of the query. This is a classical SQL injection vulnerability.

Using the database credentials extracted from the SQLite database , we tested the patient search interface, when entering the following payload into the *Patient Surname* field: ' OR '1'='1

the application returned *all patient records* instead of only the patients matching a specific surname.

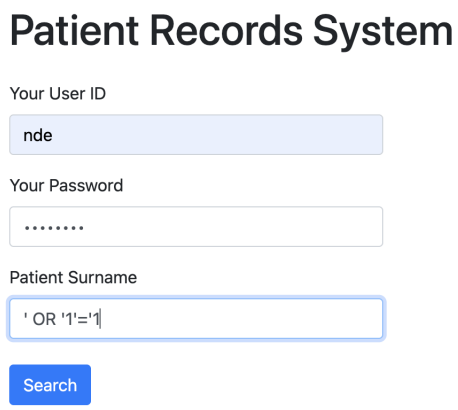


Figure 3: Input ' OR '1'='1.

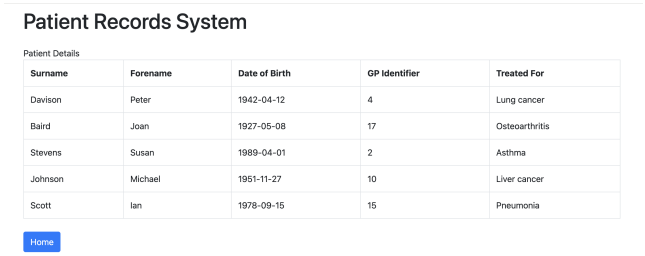


Figure 4: All data return

The condition `'1'='1'` always evaluates to true, resulting in a full table scan and disclosure of all patient data. Screenshots were taken as evidence during testing.

## 1.3 Plaintext Password

The application stored user passwords in plaintext within the `user.password` column of the SQLite database. The original `AppServlet` performed authentication by directly comparing submitted credentials against these unencrypted values using a SQL query:

```
select * from user where username='%s' and password='%s'
```

This represents a lack of credential protection, compounded by SQL injection vulnerabilities through string concatenation. No hashing mechanism existed anywhere in the codebase—passwords were written to and read from the database in their raw form.

Discovery occurred through direct database inspection. Executing

```
sqlite3 db-backup.sqlite3 "select username, password from user;"
```

on the original database revealed:

```
nde|wysiwyg0
mjones|marymary
aps|abcd1234
```

All passwords were stored in plaintext. Code review confirmed that `AppServlet.authenticated()` used `String.format(AUTH_QU` `username, password)` to build `select * from user where username='%s' and password='%s'` without any hashing, directly comparing raw input against stored plaintext values.

## 1.4 Lack of Transport Encryption

The application uses an embedded Jetty web server that only accepts HTTP connections on port 8080, with no HTTPS or TLS configured. As a result, all traffic between the browser and the server, including login credentials and patient data, is transmitted in plaintext.

We accessed the GP web application via `http://localhost:8080` and logged in with a valid GP account. The browser indicated a plain HTTP connection (no padlock icon), and no TLS certificate was requested at any point.

Anyone on the same network can eavesdrop on the HTTP traffic and read sensitive data in transit. This allows an attacker to obtain GP credentials and patient records without proper authorization, causing a serious confidentiality breach.

## 1.5 Broken Access Control

The patient search function lacks authorization checks, allowing any authenticated GP to access records of patients assigned to other doctors. The source code in `AppServlet.java` filters queries solely by surname, ignoring the logged-in user's identity:

```
private static final String SEARCH_QUERY =
    "select * from patient where surname='%s' collate nocase";
```

The query lacks a clause (e.g., `AND gp_id = ?`) to restrict results to the current session.

Using credentials for `mjones` (GP ID 2), we searched for `Davison`, a patient assigned to a different doctor (GP ID 4). The system displayed the restricted medical details, confirming horizontal privilege escalation.

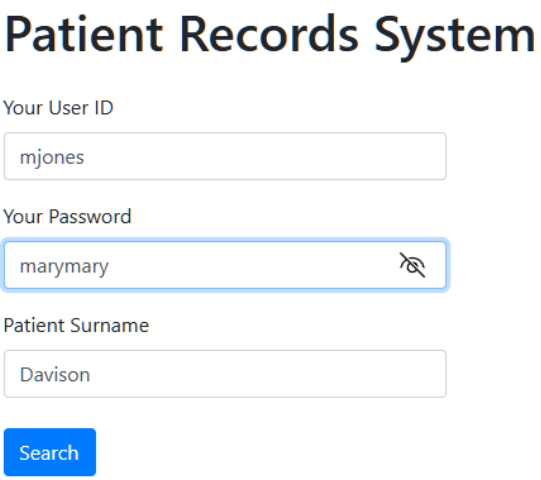


Figure 5: User 'mjones' searches for target 'Davison'.

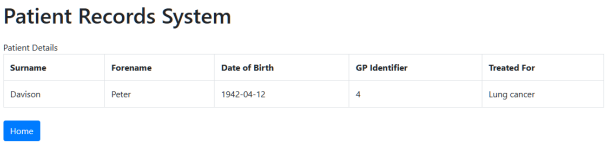


Figure 6: System displays restricted record.

- Unauthorized disclosure of sensitive PII and medical history.
- Internal attackers can scrape the entire database by guessing surnames.

# 2 Visualisation

## 2.1 Construction and Rationale

Based on the vulnerabilities identified in the security analysis phase, we developed an attack tree to model how an adversary could exploit the flaws to compromise a critical asset of the system: **unauthorised access to sensitive patient records**. The tree follows the conventions introduced in the lecture material: the root node represents the attacker's goal, child nodes represent the *conditions* required to reach that goal, OR relationships between siblings are implicit, and AND relationships are shown explicitly.

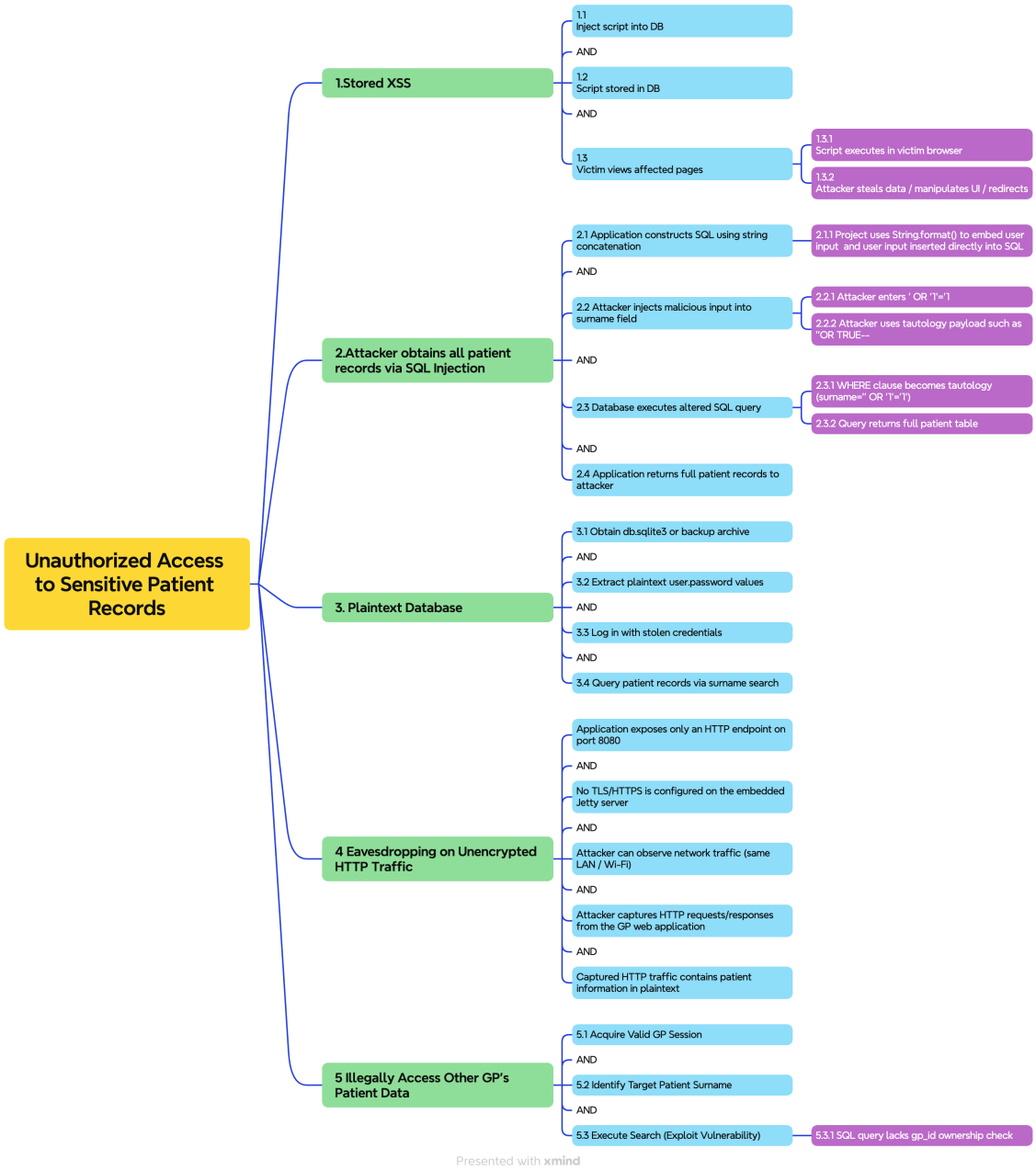


Figure 7: Attack Tree.

## 2.2 Assumption

The attack tree is based on the following minimal assumptions consistent with the provided system:

- The attacker has network access to the web application over HTTP and can submit arbitrary requests.
- The SQLite database may be accessed or leaked, enabling inspection of stored credentials and records.
- The attacker can observe traffic on the same network (e.g., unencrypted HTTP).
- Any valid GP credentials (obtained legitimately or via compromise) allow normal system access without additional security controls.

## 2.3 Explanation of Attack Paths

Each major branch in the tree represents a distinct vulnerability from Task 1, illustrating a different feasible path an attacker can follow to reach the root asset.

**1. Stored Cross-Site Scripting (XSS)** The application displays unescaped user-controlled data, allowing an attacker to inject persistent JavaScript into the database. When a victim views an affected page, the script executes in their browser. The attacker can exfiltrate patient data, hijack sessions, or redirect the victim, thereby achieving unauthorised access to patient records without needing valid credentials.

**2. SQL Injection in the Patient Search Function** The search functionality constructs SQL using string concatenation, enabling injection of payloads such as `' OR '1'='1`. This transforms the query into a tautology, causing the database to return the full patient table. As the frontend renders all returned rows, the attacker gains access to every patient's confidential information.

**3. Plaintext Password Storage and Credential Theft** The SQLite database stores GP passwords in plaintext. If an attacker obtains the database file (e.g., via local access or a leaked backup), they can directly read all credentials. With valid usernames and passwords, the attacker can authenticate as any GP and perform unrestricted patient searches, resulting in complete disclosure of patient records.

**4. Eavesdropping Due to Lack of HTTPS** The application only exposes an unencrypted HTTP endpoint on port 8080. An attacker on the same network can intercept traffic, including login credentials and patient information, using standard network sniffing tools. With captured credentials, the attacker can log in as a valid GP and query patient data legitimately through the interface.

**5. Access Control Bypass via Missing doctorId Checks** Even after authentication, the application does not verify whether the logged-in GP is authorised to view a specific patient. By querying patients using their surname, a GP can view records belonging to other doctors. An attacker with any valid GP session can therefore escalate privileges and access patient data outside their authorised scope, completing another path to the root threat.

## Summary

The attack tree demonstrates that despite their differing mechanisms, all five vulnerabilities lead to the same impactful outcome: unauthorised access to sensitive patient records. The tree clarifies the logical structure of each attack path and provides a foundation for prioritising security fixes in Task 3.

# 3 Fix Identification

## 3.1 XSS Identification

The attack tree shows that the XSS exploit succeeds only because user-supplied data is rendered into HTML without encoding. To break this path, the most effective defence is output encoding, which prevents injected script tags from executing regardless of their content. Therefore, the chosen fix is to enable HTML auto-escaping in the FreeMarker template engine, ensuring all untrusted data is safely encoded before being displayed. This directly cuts the stored-XSS execution branch in the attack tree.

## 3.2 SQL Injection Identification

The attack tree shows that the SQL injection attack path succeeds only if the application constructs SQL queries using string concatenation (Node 2.1), accepts attacker-controlled payloads such as ' OR '1'='1 (Node 2.2), and then executes the altered query and returns all patient records (Nodes 2.3–2.4). Therefore, the chosen fix directly targets the critical enabling condition at the start of this branch: replacing unsafe string-based SQL construction with parameterised queries using PreparedStatement. By binding user input as a parameter rather than embedding it into the SQL string, the injected payload can no longer modify the query structure, breaking the attacker's ability to create tautologies and preventing the downstream conditions in the attack tree from ever being reached. This fix effectively disables the entire SQL injection path and is aligned with the principle of breaking the earliest possible enabling condition in the attack chain.

## 3.3 Plaintext Password Indentification

I chose BCrypt hashing with migration to eliminate the attack tree's critical node: extracting plaintext passwords. By hashing all credentials via `./gradlew migratePasswords` and updating `AppServlet.authenticated()` to verify hashes, even if attackers steal the database, they only obtain irreversible BCrypt hashes requiring massive computational effort to crack. This breaks the AND chain at step 3.2 in attacktree, preventing credential compromise and blocking the entire path to unauthorized patient access.

## 3.4 Lack of Transport Encryption

For this flaw, the goal was to prevent user credentials and patient data from being sent in cleartext over the network. Instead of changing the application logic, we decided to add TLS support to the embedded Jetty server and expose the application over HTTPS rather than HTTP. Concretely, we stopped using the HTTP endpoint on port 8080 for normal access and configured a new HTTPS endpoint on port 8443 with a server certificate stored in a keystore. This removes the simple attack path of passively sniffing HTTP traffic while keeping the user-visible behaviour of the web interface unchanged.

## 3.5 Broken Access Control Identification

The attack tree demonstrates that horizontal privilege escalation succeeds because the search logic relies solely on user input without verifying the relationship between the requester and the data subject. To disrupt this path, the authorization check must be enforced server-side during data retrieval. The chosen fix modifies the SQL query to include a mandatory filter matching the patient's `gp_id` against the authenticated user's ID. This ensures that requests for unauthorized records yield empty results, effectively severing the "Bypass Authorization" branch in the attack tree.

# 4 Fixes Implemented

## 4.1 XSS Implemented

The Stored XSS branch exists because the application renders database fields into HTML without output encoding. To break this attack path, all user-supplied data must be HTML-escaped before reaching the browser.

HTML auto-escaping was enabled in the FreeMarker configuration by setting:

- Output format: `HTMLOutputFormat`
- Auto-escaping: enabled by default

This neutralises any `<script>` tags stored in the database: they are displayed as text rather than executed. This fix effectively mitigates all reflected and stored XSS vulnerabilities present in our application, as all user-controlled data is now safely HTML-escaped by the Freemarker template engine.

## 4.2 SQL Injection

The original code used unsafe string interpolation:

```
String query = String.format(SEARCH_QUERY, surname);
Statement stmt = database.createStatement();
ResultSet results = stmt.executeQuery(query);
```

This was replaced with a parameterised query using `PreparedStatement`:

```
PreparedStatement ps = database.prepareStatement(
    "select * from patient where surname = ? collate nocase"
);
ps.setString(1, surname);
ResultSet results = ps.executeQuery();
```

This ensures that the user input is always interpreted as data, not executable SQL, thereby directly preventing injected SQL fragments (breaking Attack Tree Nodes 2. SLQ Injection).

## 4.3 Fixing Plaintext Password

The fix involved three coordinated changes:

1. **Dependency Integration:** Added `org.mindrot:jbcrypt:0.4` to `build.gradle` to provide industry-standard password hashing (BCrypt with a configurable work factor of 12).
2. **Migration Script:** Created `PasswordMigration.java` with a Gradle task `migratePasswords` that iterates over all rows in the `user` table, identifies plaintext values (non-BCrypt format), hashes them via `PasswordUtils.hashPassword()`, and updates the database in a transaction. Running `./gradlew migratePasswords` produced:

   ```
   Password migration complete. 3 password(s) updated.
   ```

   A post-migration query showed the password transformed to hashed style confirming successful hashing.
3. **Authentication Refactor:** Modified `AppServlet.authenticated()` to query only by username (`select password from user where username = ?`), then call `PasswordUtils.matches(password, storedPassword)` to verify credentials. This method gracefully handles both BCrypt hashes and legacy plaintext (for backward compatibility during rollout), though production deployments should remove plaintext fallback after complete migration.

## 4.4 Lack of Transport Encryption

All changes were made in `AppServer.java`, which starts the embedded Jetty server. The original code created a `Server` bound to port 8080 (for example, `new Server(8080)`), so Jetty only listened for plain HTTP. We replaced this with a `Server` that uses an SSL context loaded from `config/keystore.p12` and an HTTPS connector on port 8443. The existing servlet handler was left unchanged and attached to the new server.

The essential code change is:

```
// Before:
Server server = new Server(8080);

// After:
Server server = new Server();

SslContextFactory.Server ssl = new SslContextFactory.Server();
ssl.setKeyStorePath("config/keystore.p12");
ssl.setKeyStoreType("PKCS12");
ssl.setKeyStorePassword("changeit");
ssl.setKeyManagerPassword("changeit");

ServerConnector https = new ServerConnector(
    server,
    new SslConnectionFactory(ssl, "http/1.1"),
    new HttpConnectionFactory(new HttpConfiguration())
);
https.setPort(8443);
server.addConnector(https);
```

After rebuilding the application, we accessed the system at `https://localhost:8443`, accepted the warning about the self-signed certificate, logged in with a GP account and opened several patient records. The browser reported an HTTPS connection, which confirms that the application now uses TLS instead of plain HTTP for all web traffic.

## 4.5 Broken Access Control

The unauthorized access occurs because the application retrieves records based solely on user input. To mitigate this, the database query must validate the relationship between the requesting doctor and the patient.

The SQL query was modified to include a subquery filtering by the current user's ID:

```
private static final String SEARCH_QUERY =
    "select * from patient where surname = ? AND gp_id = (select id from user where username = ?)";
```

The `searchResults` method was updated to accept the authenticated username and bind it to the prepared statement:

```
// In doPost method
model.put("records", searchResults(surname, username));

// In searchResults method
ps.setString(1, surname);
ps.setString(2, username);
```

This ensures that only records belonging to the currently logged-in GP are returned, effectively neutralizing horizontal privilege escalation attacks (Attack Tree Node 1.2).