COMP3911 Secure Computing Coursework

Adhiraj Kumar	[sc23ak4]	201758717
Andrea Diez Basurto	[sc23adb]	201765781
Aristaa Singh	[sc23as2]	201745878
Sinjini Sarkar	[sc23ss2]	201695493
Sreeja Tulluru	[sc23sct]	201704967

Sections in this report:

Task 1: Analysis of Flaws

1. Passwords are stored insecurely in plaintext, and the authentication process is weak.
2. The login and patient search functions are vulnerable to SQL injection.
3. The system does not enforce access control, allowing any GP to view any patient's records.
4. Patient data is displayed without filtering, creating a risk of XSS attacks.
5. Detailed internal error messages are exposed to users, leading to information leakage.

1. Plaintext Password Storage and Weak Login Security

The system stores GP passwords directly in the database as plaintext. When looking at the user table in SQLite using SELECT * FROM user;, the password column displays the actual passwords for every account without any hashing or encryption. This violates OWASP A02:2021 – Cryptographic Failures, which states that sensitive data such as passwords must never be stored in plaintext.

If someone obtains the database file, whether through a backup copy, a stolen device, or a system breach, the stored passwords can be viewed straight away (Figure 1). Because the passwords are not hashed, a single database leak would expose all GP login credentials.

Furthermore, the passwords themselves are weak (ex: marymary, abcd1234), indicating no enforcement of password policies. The system also provides no brute-force protection or account lockout, meaning attackers can attempt unlimited password guesses. These issues together create a weak authentication mechanism and increase the likelihood of unauthorised access.

```
sqlite> .headers on
sqlite> .mode column
sqlite> SELECT * FROM user;
id  name          username  password
--  ------------  --------  --------
1   Nick Efford   nde       wysiwyg0
2   Mary Jones    mjones    marymary
3   Andrew Smith  aps       abcd1234
sqlite> .schema user
CREATE TABLE user (id integer not null primary key, name varchar(30) not null, username varchar(12) not null, password char(8) not null );
```

Figure 1: Database displaying GP passwords in plain text

2. SQL Injection in Authentication and Patient Search

The application is vulnerable to SQL injection because it builds SQL statements by directly inserting whatever the user types into the query. Both the login and the search features create SQL using String.format, so the text entered into the form becomes part of the database command instead of being treated as ordinary input.

```
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='%s' collate nocase";
```

During login, the values entered in "Your User ID" and "Your Password" are placed straight into the SQL query. This means an attacker can change the logic of the query. For example, entering a valid username such as "nde" and the following password: ' OR ' '='. This results in the database evaluating: select * from user where username='nde' and password='' OR ' '='' ;

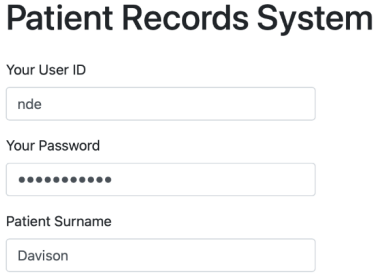

Figure 2: Login form with the password ' OR ' '='.

The final part (OR ' '='') is always true, so the database returns a user record even though the password is wrong. This results in a full authentication bypass, allowing an attacker to log in as any GP without knowing their password.
(See Figures 2 and 3).

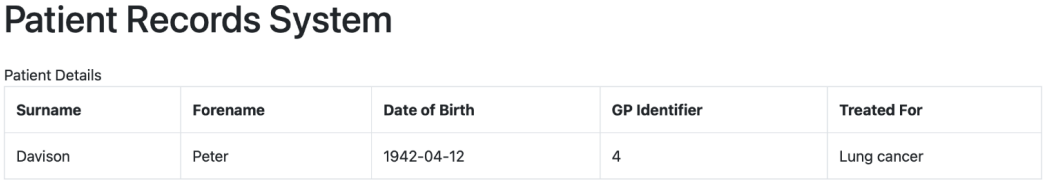

Figure 3: Application showing the patient search page despite the incorrect password.

The same issue appears in the patient search box. If the “Patient Surname” field contains: Davison' OR '=' , causes the system to return every patient record instead of only those with the surname “Davison” (see Figures 4 and 5). This results in the database evaluating: select * from patient where surname='Davison' OR ' '=' ' collate nocase;

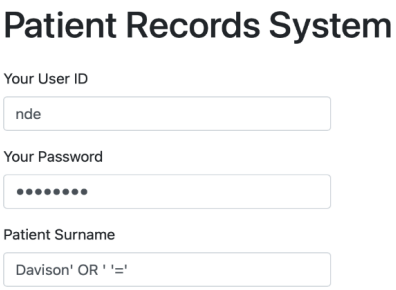

Figure 4: Patient surname field

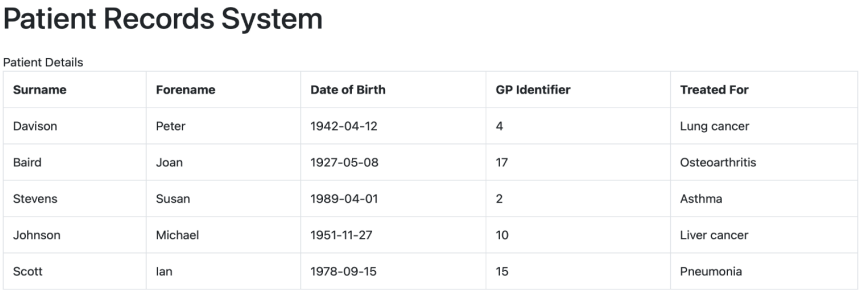

Figure 5: Results page showing all patient records.

This is a fundamental problem with how the SQL queries are created. Many different injected inputs (for example x' OR 'a'='a or ' OR '1'=') also work because the application builds SQL directly based on what the user types. As a result, an attacker can bypass authentication entirely, view all stored patient records, or manipulate the database. SQL injection presents a severe risk to confidentiality and integrity in this medical system.

3. Weak Access Control on Patient Records

After a GP logs in, the system gives them access to patient information based only on the surname entered in the search box. It does not check whether the logged-in GP is actually the doctor responsible for that patient. This means one GP can view the records of patients who belong to a different GP simply by entering the patient’s surname. This violates OWASP A01:2021 – Broken Access Control, where users can access data they are not authorised to view.

The search query only filters by surname and completely ignores the gp_id value, even though this field identifies which GP treats each patient:

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

The patient table: SELECT id, surname, forename, gp_id FROM patient; showed several patients belonging to different GPs (different values in the gp_id column). Logging in as one GP and searching for a patient assigned to another still shows the full record, meaning the system is not checking whether the logged-in GP is allowed to see them. For example, Stevens is assigned to Mary Jones (gp_id = 2), yet Nick Efford is still able to view her details using his own login (Figure 6 and 7).

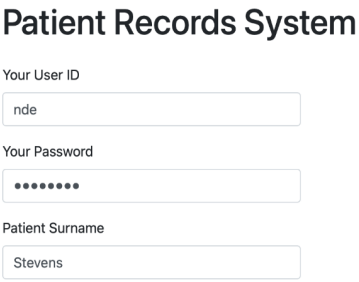

Figure 6: Nick Efford logging in.

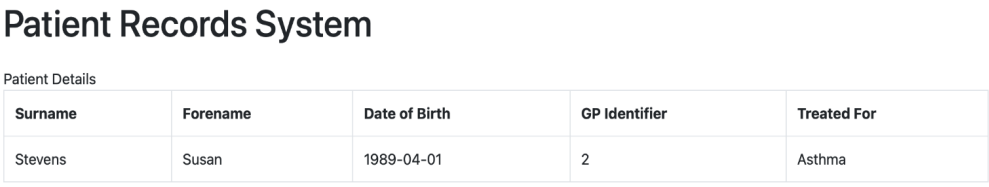

Figure 7: Application showing Stevens record.

4. Executable Code in Patient Records (Stored XSS Vulnerability)

The patient details page displays the contents of the database directly inside the webpage rather than treating them as plain text. In details.html, fields such as ${record.surname} and ${record.diagnosis}

```
sqlite> SELECT id, surname, forename, treated_for FROM patient;
id  surname  forename  treated_for
--  -------  --------  -------------
1   Davison  Peter     Lung cancer
2   Baird    Joan      Osteoarthritis
3   Stevens  Susan     Asthma
4   Johnson  Michael   Liver cancer
5   Scott    Ian       Pneumonia
sqlite> UPDATE patient
   ...> SET treated_for = '<script>alert("XSS")</script>'
   ...> WHERE surname = 'Davison';
sqlite> SELECT id, surname, treated_for FROM patient WHERE surname = 'Davison';
id  surname  treated_for
--  -------  ----------------------------
1   Davison  <script>alert("XSS")</script>
```

Figure 8: Updating treated_for (Diagnosis) field to <script>alert("XSS")</script>.

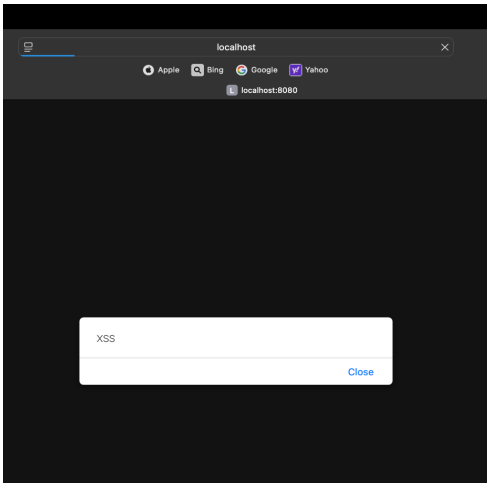

Figure 9: JavaScript popup.

are written straight into the HTML, and because no filtering is applied, any HTML or JavaScript stored in the database is executed by the browser.
This is a stored Cross-Site Scripting (XSS) vulnerability, where a malicious script saved in a patient record runs automatically whenever the page is viewed.
The flaw was confirmed by updating the treated_for (Diagnosis) field for a patient to contain <script>alert("XSS")</script> (Figure 8). When searching for the patient's information, the website displayed a JavaScript popup as soon as the details page loaded (Figure 9). This shows that the information from the database is not handled as text but is executed as code, proving that the output is not handled safely.

5. Detailed Error Messages Expose Internal System Information

The application exposes internal system information through its error messages. Because AppServlet uses "TemplateExceptionHandler.HTML_DEBUG_HANDLER", Freemarker displays full debugging details directly in the browser when a template error occurs (Figures 10 and 11). Causing a template to fail; for example by interrupting the page rendering brings up a Freemarker error page that lists stack traces, Java class names and file paths such as details.html in the ./templates folder.
This means the system reveals sensitive implementation details that should not be visible to users. By viewing the debug page, an attacker can learn how the application is structured and identify potential targets such as template names, endpoints and code paths. The flaw was confirmed when the browser displayed the full Freemarker debug output instead of a simple user-friendly error message, demonstrating that internal diagnostic information is exposed.

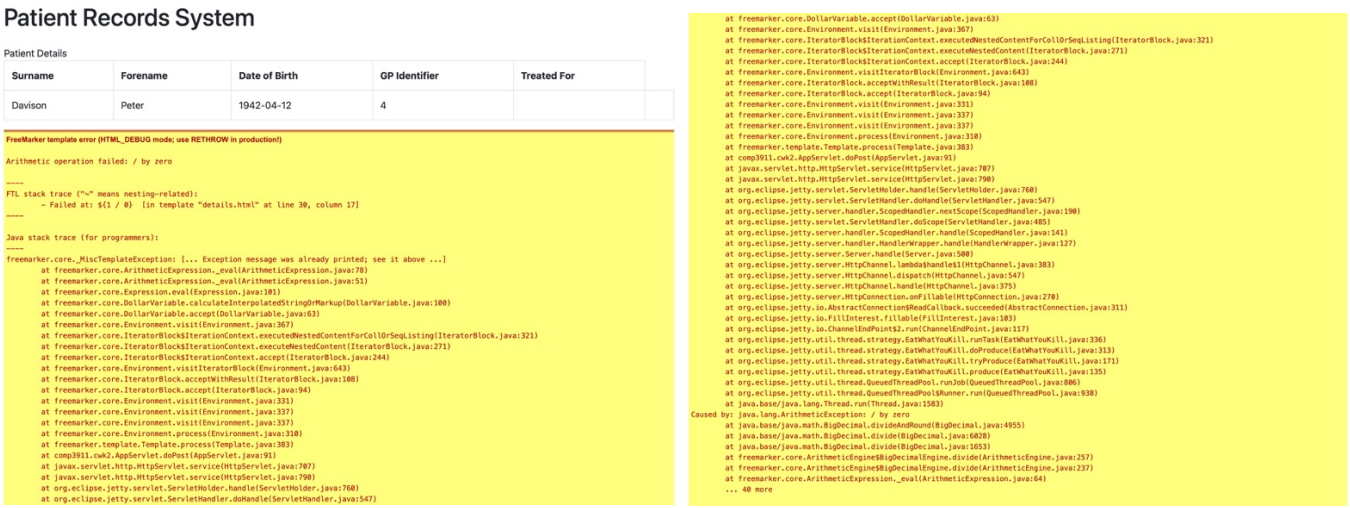

Figure 10: Freemarker error results.

Task 2: Attack Tree

The attack tree below (see next page) shows how each of the five vulnerabilities we found in Task 1 can allow an attacker to gain unauthorised access to patient medical records. The root node represents the attacker's main goal. Each top-level branch (1.1–1.5) links to one of the vulnerabilities we found in Task 1. Under each branch, we added the steps an attacker would need to follow to exploit that flaw. When all steps must happen together, we used an AND label. If there is no AND label, the steps are treated as OR, meaning any of them could be used.

- Plaintext passwords: If an attacker gets the database file, they can read real GP passwords and log in.
- SQL Injection: By injecting ' OR "=' into the login fields or patient search, the attacker can bypass login or force the system to return all patient records.

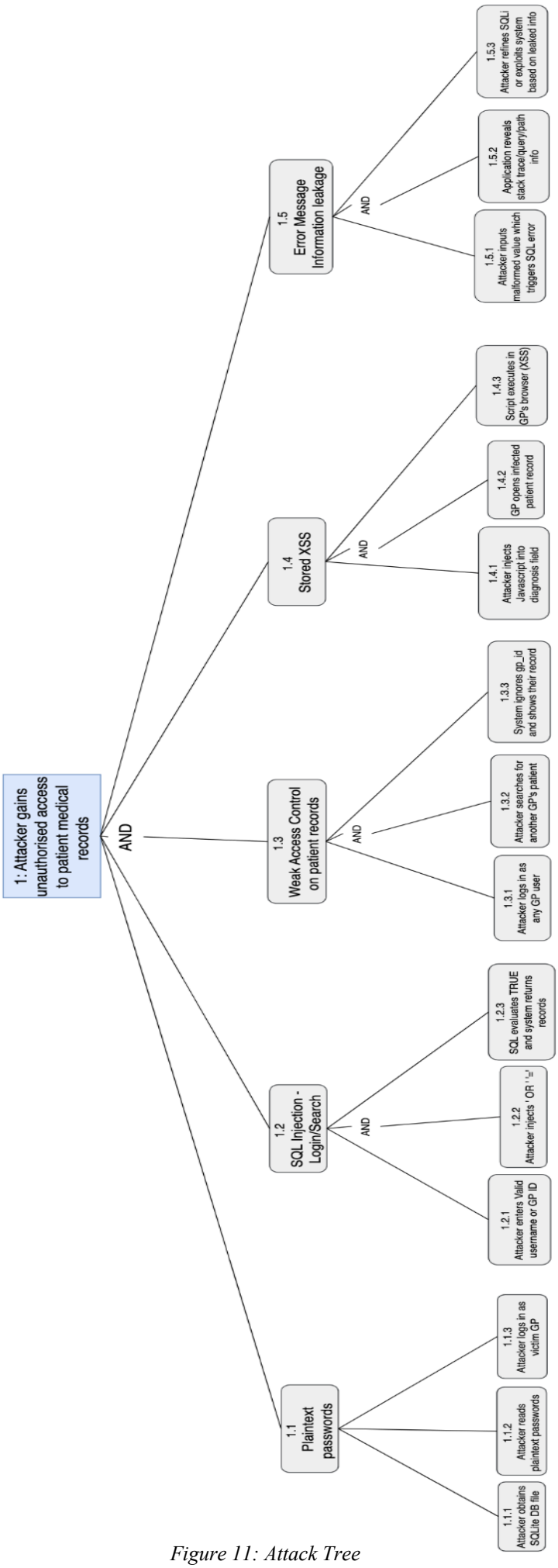

Figure 11: Attack Tree

- Weak Access Control: Once logged in, the system does not check which GP the patient belongs to, so any GP account can view all records.
- Stored XSS: If an attacker injects JavaScript into the diagnosis field, it will run when a GP views the patient record.
- Error Messages: System error pages show detailed information (ex: file paths, SQL errors) that help the attacker improve other attacks like SQL injection. (See Attack Tree Diagram on the next page).

This attack tree visualises how each vulnerability can lead to the same final goal and shows how some flaws require a sequence of steps (AND), while others give multiple attack options (OR by default).

Assumptions

- The attacker has normal network access to the web application (ex: through a browser).
- The system behaves the same way during testing as it does in real deployment.
- The attacker can try different inputs in the login form and search fields (no strong input filters).
- The attacker may obtain the SQLite database file through filesystem access or a backup misconfiguration.
- A GP account can be accessed by the attacker if a password is known or retrieved (ex: via plaintext storage).

Task 3(A): Fix Identification

Flaw 1: Plaintext Password Storage and Weak Login Security: This flaw allowed attackers to gain unauthorised access to the database where the GP passwords were stored in plaintext. In the attack tree, this appeared as branch 1.1 (Plaintext Password), where an attacker who obtains the database (node 1.1.1) can directly read GP credentials (node 1.1.2) and log in as any GP (node 1.1.3). This flaw provided a direct route to the attacker's main goal of accessing medical records. Hashing passwords was chosen as a fix for this flaw as it breaks this entire attack chain. Even if the attacker obtains the database, hashed passwords cannot be reversed, preventing the attackers from logging in as a valid GP. Therefore, hashing removes both the credential disclosure risk and the authentication bypass identified in the attack tree. The 2nd part under this flaw was weak login security, as GP passwords such as “marymary” and “abcd1234” could be easily brute-forced if the database was leaked. Since the system does not provide password change or account creation features, the appropriate fix was to replace the existing plaintext passwords with strong SHA-256 hashes. Overall, hashing passwords was selected because it directly addresses the risk highlighted in the attack tree, prevents attackers from escalating database access into full system access and strengthens the authentication mechanism to remove one of the most severe vulnerabilities.

Flaw 2: SQL Injection in Authentication and Patient Search: This flaw allowed attackers to manipulate SQL queries by injecting malicious input into the username, password, or surname fields. In the attack tree, this appears under branch 1.2 (SQL Injection), where the attacker enters valid input (node 1.2.1), injects a payload such as ' OR "=' (node 1.2.2), and forces the condition to always evaluate TRUE (node 1.2.3). This enabled both authentication bypass and retrieval of all patient records through the search field (e.g., Davison' OR "=').
PreparedStatements with parameterised queries were chosen as the fix because they prevent user input from altering SQL structure. Once the parameters are treated strictly as data, injection payloads no longer change query logic. This fully breaks the attack chain described in nodes 1.2.1–1.2.3, stopping both the login bypass and the full patient-record disclosure. This fix follows OWASP best practices and directly neutralises the SQL injection vulnerability.

Flaw 3: Weak Access Control on Patient Records: This flaw enabled a logged-in GP to access patient records belonging to any other GP. In the attack tree, this is captured under branch 1.3, where the attacker logs in normally (node 1.3.1), then searches for another GP’s patient (node 1.3.2), and the system returns those records because the gp_id field is ignored (node 1.3.3). The fix chosen was to enforce server-side access control by binding patient queries to the currently authenticated GP (i.e., adding AND gp_id = ?). This breaks the full attack chain by ensuring an attacker cannot escalate privileges simply by entering another GP’s patient surname. Because this flaw exposed private medical information and allowed horizontal privilege escalation, enforcing strict per-user access checks was the most appropriate and security-aligned fix.

Flaw 4: Executable Code in Patient Records (Stored XSS Vulnerability): This flaw allowed arbitrary JavaScript stored in the database (node 1.4.1) to execute automatically in the GP's browser when the patient page was loaded (nodes 1.4.2, 1.4.3). This represented a severe risk because an attacker could escalate attacks to credential theft, screenshot capture, or session hijacking. Escaping HTML output using FreeMarker's <#escape x as x?html> directive was chosen as the fix, as it guarantees that special characters such as <, > and quotes are encoded instead of executed. This breaks the entire XSS chain because even if malicious script tags exist in the database, they are rendered harmless as plain text. This defence aligns directly with OWASP XSS prevention recommendations and ensures all dynamic content is safely encoded by default.

Flaw 5: Detailed Error Messages Expose Internal System Information: This flaw caused the system to reveal sensitive internal details whenever template-processing error occurred. In the attack tree, this appears under branch 1.5 (Error Message Information Leakage), where the attacker intentionally inputs malformed data (node 1.5.1) or triggers a template failure causing the system to reveal sensitive debugging details (node 1.5.2) such as stack traces, SQL queries, Java class paths and template filenames. With this information, the attacker can refine and improve further attacks (node 1.5.3), including SQL injection payloads, directory traversal attempts or targeted exploitation of specific code paths. This fix was chosen to remove Freemarker's "HTML_DEBUG_HANDLER" and replace it with the production-safe "RETHROW_HANDLER", combined with generic user-facing error messages. This breaks the attacker's information-gathering chain in the attack tree by preventing access to internal template errors, file paths and SQL details, eliminating the possibility of using debug output to support further targeted attacks.

Task 3(B): Fixes Implemented

Flaw 1: Plaintext Password Storage and Weak Login Security

Implemented SHA-256 password Hashing: the application originally compared usernames and passwords using plaintext values directly from "db.sqlite3"

To fix this:

1. A new "hashPassword()" method was added using SHA-256
2. The incoming password from the login form is hashed before comparison.
3. The SQL query was updated to compare hashed values only.

This ensures the system never handles plaintext passwords internally.

Updated the Database to store Hashed Passwords: the user table was manually updated with cryptographically secure SHA-256 hashes for the 3 GP account. The following screenshot shows how the database looks after password hashing:

```
sqlite> .headers on
sqlite> .mode column
sqlite> select * from user;
id  name          username  password
--  ------------  --------  ----------------------------------------------------------------
1   Nick Efford   nde       f6374d463875d2bc1eb45346caa18b4dc1eec2148177e8f8d8c62804913b
                            0244

2   Mary Jones    mjones    f34ff5d9e920ebefb08bcb86ae515adc279cac54d63c744d9df6521b2e37
                            d0a8

3   Andrew Smith  aps       e9cee71ab932fde863338d08be4de9dfe39ea049bdafb342ce659ec5450b
                            69ae
sqlite>
```

Figure 12: Database displaying Hashed passwords

Verification:

1. Normal logins still work using plaintext passwords in the login form (since hashing happens internally).
2. Reading database file no longer reveals passwords.
3. The hashed passwords match the computed hashes exactly - conforming integrity.

Flaw 2: SQL Injection in Authentication and Patient Search

Implemented PreparedStatements with Parameterised Queries: Originally the system constructed SQL queries using string concatenation, allowing attacker-controlled input to modify query logic. This flaw enabled authentication bypass and unrestricted access to all patient records.

To fix this:

1. Replaced all the SQL queries with PreparedStatements in AppServlet.java.
2. Bound user input safely using stmt.setString() to ensure values are treated strictly as data.

3. Updated the login query (AUTH_QUERY) to user username=? AND password=?.
4. Updated the search query (SEARCH_ query) to use surname=? Instead of raw string insertion.

```
// FIX FOR FLAW 2: changed the query constants to use ? placeholders
private static final String AUTH_QUERY = "select * from user where username=? and password=?";
// FIX FOR FLAW 2: changed the query constants to use ? placeholder
// FIX FOR FLAW 3: only return patients for the logged-in GP (gp_id = currentUserId)
private static final String SEARCH_QUERY = "select * from patient where surname=? collate nocase and gp_id=?";
```

5. Removed all occurrences of String.format() and concatenated SQL strings.

This prevents attacker-supplied input (e.g., ' OR '='') from altering the query. The screenshots below show an example SQL injection attempt after the fix.

Figure 13: Attempted SQL injection in surname field after fix

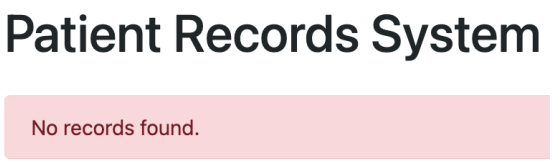

Figure 14: Correct system behaviour after fix

Verification:

1. Attempting SQL injection payloads (e.g., ' OR '='') in the password fields no longer bypasses authentication.
2. Injected surnames such as Davison' OR '= correctly return 0 results, instead of exposing the full patient table.
3. The system remains stable under all malicious inputs, no crashes, no debug leaks, and no unintended behaviour.
4. Application logs verify that SQL query structure remains intact, with only parameter values being bound securely.

Flaw 3: Weak Access Control on Patient Records

Implemented Server-Side Access Control Using gp_id Filtering: Originally, any authenticated GP could view all patient records simply by entering a surname. The system ignored the gp_id column, allowing cross-GP access and compromising patient confidentiality.

To fix this:

1. Modified the authenticated() function to return the logged-in GP's user_id (Integer) instead of a boolean. This allowed secure identification of the logged-in GP in later queries.
2. Stored this value in currentUserId after successful login in doPost().
3. Updated the patient search SQL query (SEARCH_QUERY) to surname = ? AND gp_id = ?, ensuring the second parameter restricts results to the logged-in GP only.
4. Bound both parameters using a PreparedStatement, preventing SQL injection and enforcing proper access control.

These changes ensure that each GP can only view their own patients, and no cross-account access is possible. Even if an attacker enters valid surnames with correct username and password belonging to another GP, the system returns "No records found." as seen in the screenshot below.

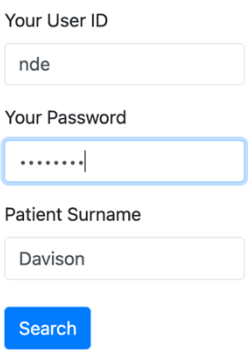

Figure 15: GP Nick searching Davison (patient of other GP)

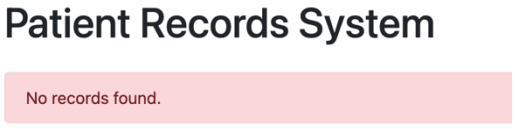

Figure 16: "No records found." after fix

Verification:

1. Logging in as GP1 only returns GP1's patient records.
2. Searching for patients belonging to GP2 while logged in as GP1 correctly shows no results.
3. SQL injection in the surname field cannot bypass gp_id restrictions.

4. Modifying the authenticated function to return user_id ensures access control is consistently enforced.

Flaw 4: Executable Code in Patient Records (Stored XSS Vulnerability)
Implemented HTML Output Escaping Using FreeMarker: The system previously rendered patient data directly into HTML without filtering (e.g., ${record.diagnosis}). Because of this, any stored script such as <script>alert("XSS")</script> executed automatically in the GP's browser when the record was viewed.
To fix this: Applied FreeMarker's HTML-escaping block in details.html

```
<!-- FIX FOR FLAW 4: Escape all template output as HTML -->
<#escape x as x?html>
<div class="container">
```

```
</div>
</#escape> <!-- end XSS fix -->
```

This ensures all dynamic values are safely encoded, meaning <script> is displayed as plain text rather than executed.

Patient Details

Surname	Forename	Date of Birth	GP Identifier	Treated For
Stevens	Susan	1989-04-01	2	<script>alert("XSS")</script>

Figure 17: Stored script displayed safely as plain text after fix

Verification:
1. Stored JavaScript payloads (e.g., <script>alert("XSS")</script>) are no longer executed in the browser.
2. The malicious script is displayed safely as plain text in the "Treated For" column.
3. Encoding prevents the attacker from injecting HTML/JS into the page structure.
4. Page loads normally with no alert boxes, pop-ups, or DOM modification.

Flaw 5: Detailed Error Messages Expose Internal System Information
Implemented Safe Error Handling:
The application previously used Freemarkers's "TemplateExceptionHandler.HTML_DEBUG_HANDLER", which displayed full debugging information (stack traces, file paths, SQL hints, template location) directly in the browser whenever a template error occurred. This leaked internal system details can be used to refine further attacks.
To fix this:
1. Replaced the insecure debug handler

```
        // FIX FOR FLAW 5: Prevent exposing internal Freemarker debug information
        fm.setTemplateExceptionHandler(TemplateExceptionHandler.RETHROW_HANDLER);
        fm.setLogTemplateExceptions(false);
        fm.setWrapUncheckedExceptions(true);
    }
```

Figure 18: Secure Debugging Handler (Fix for Freemarker's detailed debug information)

2. Updated the expectation-handling logic both in "doGet()" and "doPost()" to return a simple generic error message instead of stack traces.
3. Ensured that no internal diagnostic information is sent to the client even when the template fails.
4. Updated Error output: the screenshot below shows the new, safe error message displayed when a template is internally broken. This ensures that the user interface remains functional while all sensitive technical details remain hidden from the attackers.

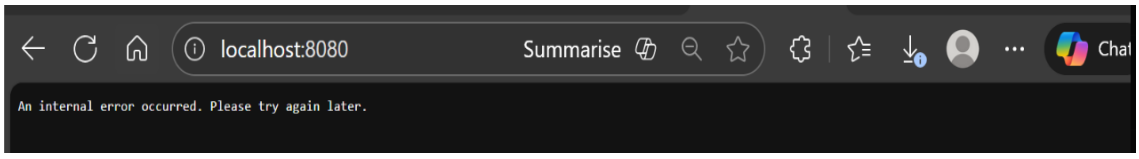

Figure 19: Safe generic error message displayed

Verification:
1. Breaking "details.html" now produces a generic error message instead of Freemarkers's yellow debug screen.
2. No stack traces, file paths or internal application details appear in the browser.
3. Application continues to function normally during non-error conditions, conforming stable behaviour.
4. This verifies that the information leakage vulnerability has been fully removed and that internal system behaviour is now properly protected.