COMP3911 Coursework

Belal Yahouni, sc23by (201736144)

Andrew Auld Mercado, sc23aam (201727177)

Ramith Gajjala, sc23rg2 (201751727)

Maxim Varea, sc23mv, (201694348)

Samrath Poonia, sc23sp2 (201721566)

Table of Contents

1. Analysis of Flaws
    1.1. SQL Injection Attack
    1.2. Insecure HTTP Protocol
    1.3. Stored Cross-Site Scripting (XSS)
    1.4. Password Hashing
    1.5. Cross-Site Request Forgery
2. Attack Tree
3. Implementation of Security Fixes
    3.1. Fix Identification
        3.1.1. SQL Injection Attack
        3.1.2. Insecure HTTP Protocol
        3.1.3. Stored Cross-Site Scripting (XSS)
        3.1.4. Password Hashing
        3.1.5. Cross-Site Request Forgery
    3.2. Fixes Implemented
        3.2.1. SQL Injection Attack
        3.2.2. Insecure HTTP Protocol
        3.2.3. Stored Cross-Site Scripting (XSS)
        3.2.4. Password Hashing
        3.2.5. Cross-Site Request Forgery

1. Analysis of Flaws

List of discovered flaws:
1. SQL Injection Attack through Password to allow access to patient data.
2. Insecure HTTP protocol instead of secure HTTPS (unencrypted transmission of data packets).
3. Stored Cross-Site Scripting (XSS) Vulnerability on Patient Details Page
4. Unhashed user data allowing easy access to view user passwords.
5. Cross-Site Request Forgery (CSRF) via forged POST requests to the login/search form.

1.1 SQL Injection Attack

The SQL injection attack works by inserting text that forms part of an SQL query into a input field, where it is then processed as SQL rather than ordinary text. This causes the application to run a different query from the one intended, which can make the database return results that should not be accessible. In our case, this allows an attacker to bypass the login process completely.

I began by exploring the website to understand how it works: what inputs it takes and what outputs it produces. After seeing the login form, I considered whether an SQL injection attack might work through one of the input fields. I then checked the source code to find where the SQL query is formed, so I could understand it better. I thought about the idea of forcing the query to output true without needing a valid username or password, and so I designed a query to do this.

The SQL injection occurs through the password field (the username can be left blank). The original code constructs the following query:

```
SELECT * FROM user WHERE username='%s' AND password='%s';
```

If we enter the string ‘OR ‘1’=’1’ -- it becomes:

```
SELECT * FROM user WHERE username='' AND password='' OR '1'='1' --';
```

This means that the database selects from the user table where the username and password are both empty, or where '1'='1'. Since '1'='1' is always true, the entire WHERE condition is true. The -- at the end comments out the remaining ' as it is not needed and prevents a syntax error.

The intended query only grants access when the supplied username and password match a row in the user table. The altered query instead grants access if ('1'='1') is true, which it always is. As a result, the attacker is authenticated without valid credentials and can access all patient data.

1.2 Insecure HTTP Protocol

The web application is currently running on the localhost using an HTTP protocol, which while perfectly fine for development, could pose a serious security issue when in production. By using HTTP (an insecure version of HTTPS) all data, including user credentials and database entries, is transmitted as plaintext, which allows for Man-In-The-Middle attacks, where attackers can sniff unencrypted data packets to obtain sensitive user information.

I discovered this flaw when I was looking through the Web-App Vulnerabilities slides on Minerva, specifically the Open Redirects section. This prompted me to try messing with the URL of the web app, but when I ran the application on IntelliJ and typed in ‘https://localhost:8080’ on Safari to boot up the

app, I got a message saying ‘Safari cannot connect to the server’. Then, I tried ‘http://localhost:8080’ and the site loaded up, and so I found the site did not have the secure HTTPS protocol.

When using HTTP instead of HTTPS, all data sent over HTTP requests is sent over as plaintext, meaning anyone with access to the network can sniff data packets and read sensitive information such as passwords, credit/debit card information, etc. Additionally, session tokens can be stolen, allowing attackers to take over a user’s session and gain unauthorized access to their account, among many other attacks.

1.3 Stored Cross-Site Scripting (XSS)

The patient details page outputs values from db.sqlite3 directly into HTML using unescaped Freemarker expressions such as ${record.surname}. Since, there is no HTML escaping being applied, any HTML or JavaScript stored in the database is embedded into the page and executed by the browser. This allows an attacker with brief database access to inject a <script> payload into a patient record, creating a stored XSS vulnerability that executes whenever a GP views the affected patient’s record.

During inspection of details.html, I observed that patient fields were rendered using raw Freemarker variables without the ?html escaping operator. This meant that any HTML stored in the database would be inserted directly into the webpage’s HTML structure and interpreted by the browser as active content. To confirm that this behaviour created an exploitable vulnerability, I used sqlite3 to modify a patient’s *treated_for* field with a small JavaScript payload (Figure 1). After restarting the application and viewing the corresponding patient record, the browser executed the injected script (Figure 2), demonstrating that unescaped database content resulted in stored XSS within an authenticated session.

```
Enter ".help" for usage hints.
sqlite> UPDATE patient
   ...> SET treated_for = '<script>alert("XSS in Treated For")</script>'
   ...> WHERE id = 1;
sqlite> SELECT id, treated_for FROM patient WHERE id = 1;
1|<script>alert("XSS in Treated For")</script>
```

Figure 1: Database record modified to include a <script> payload in treated_for.

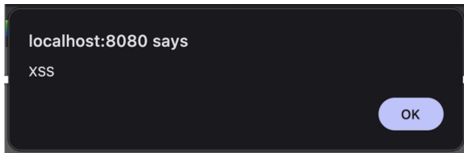


Figure 2: JavaScript alert triggered by the stored XSS payload.

To demonstrate this flaw, I assumed a realistic scenario in which an attacker gains brief access to a GP’s unlocked workstation terminal. During this window, the attacker opens db.sqlite3 using sqlite3 and modifies a field such as treated_for to contain a malicious JavaScript payload. Because the application renders this value using an unescaped Freemarker expression, the injected content is inserted directly into the HTML of the patient details page. Then, when the GP goes to open the affected record, the browser executes the script, showing that unescaped output allows stored JavaScript to run within an authenticated session. Once executed, the script can exfiltrate the GP’s session cookie, enabling the attacker to impersonate the GP remotely and access confidential patient records without any further physical access to the workstation.

1.4 Password Hashing

In computer systems, storing passwords as raw text is often frowned upon due to the risk of the data being compromised some way or another. Instead what secure systems do is they store a hash of the password, meaning that the only way to authenticate is through validating that a password hash matches the stored password hash. This way it is near impossible to decipher the password from the hash even if it gets compromised.

I discovered this flaw very quickly after checking the database for login credentials.

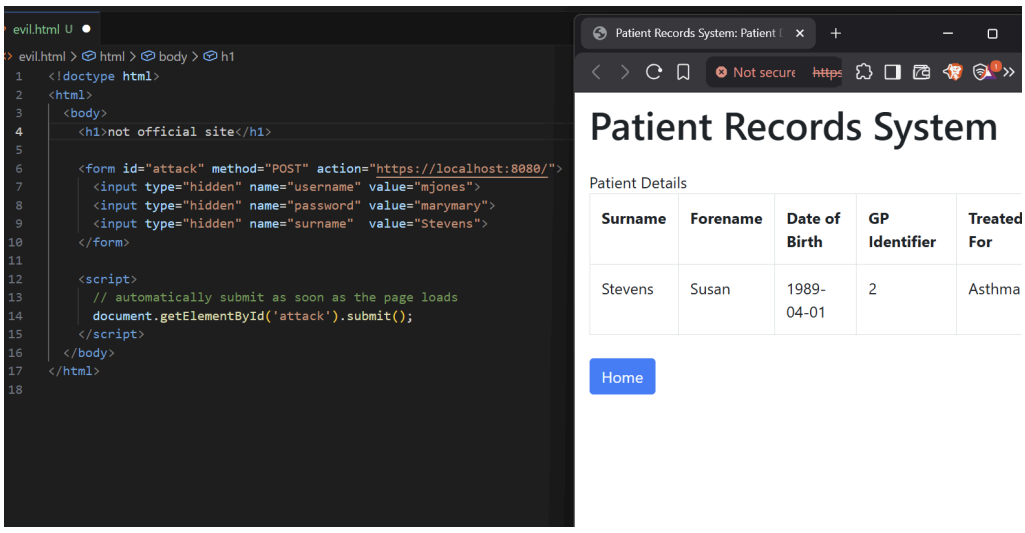
The salt is a randomly generated sequence of bytes, used in conjunction with the password to generate the password hash. An example attack could be an insider having access to the database and being phished into providing his server credentials into a fraudulent site, giving attackers access to the database. With this the attackers have gained access to all the users' credentials. Another issue that I solved using password hashing is brute force attacking, by iteratively hashing lots of times we increase the required computing per hash stopping supercomputers from being able to attempt every single possible permutation of passwords.

1.5 Cross-Site Request Forgery

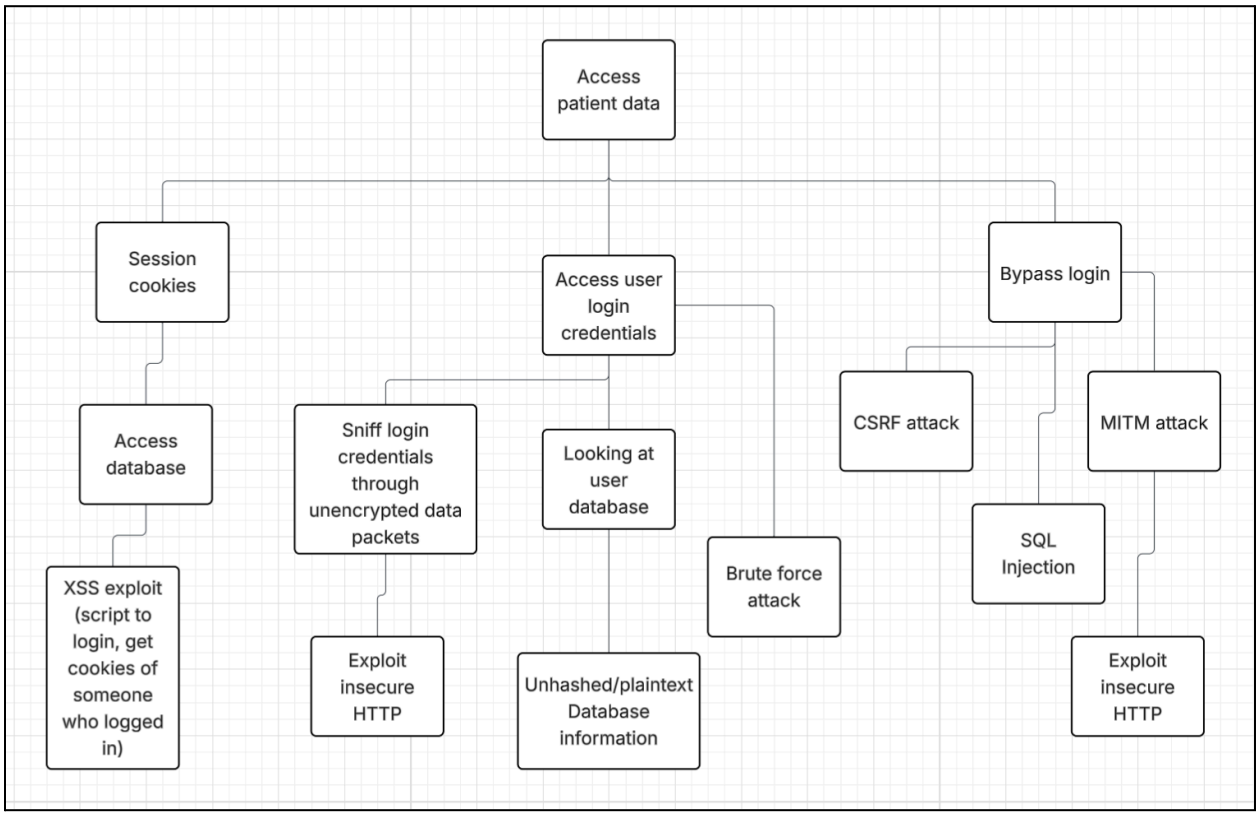
A CSRF attack works by tricking a user's browser into sending a request to a website they use, without their knowledge. Because the browser automatically includes any cookies or authentication, the server treats the request as if the user had made it themselves. In our application, the login and patient search are both handled by a single POST request to /, and there is no CSRF token or origin check. This means a malicious site can cause a GP's browser to submit the login/search form in the background, effectively driving the application on the GP's behalf.

The login form simply posts the username, password and surname to "/" with no random hidden value that could act as a CSRF token. When I checked AppServlet.doPost(), it just reads these three parameters and calls authenticated() and searchResults() without any CSRF or origin checks. To prove this was exploitable, I created a separate HTML file (evil.html) outside the application with a hidden form that posted directly to http://localhost:8080 and auto-submitted on page load. Opening this file immediately showed the patient details page, even though I never used the real login form, confirming that cross-site requests are accepted.

The CSRF vulnerability happens because the servlet accepts any POST to "/" with the parameters username, password and surname, and then calls authenticated() and searchResults() without checking a CSRF token or the request's origin. This means the application cannot tell the difference between a genuine form submission and a forged one. An attacker can host a hidden form that posts these fields directly to https://localhost:8080 and auto-submits it with JavaScript, causing the GP's browser to perform a login and search without the GP ever using the real form.



2. Attack Tree



The attack tree was mainly constructed using a top-down approach. We first thought about what the attacker's overall goal would be, so we could use that as the root node. In this case, we decided the main goal of an attacker is to access the patient data stored in the system. This data is highly sensitive and contains private medical information that unauthorised users should not be able to view. If an attacker gained access, they could potentially leak it, sell it, or use it for further exploitation.

From there, we moved on to thinking about the different ways an attacker could get to that data. We identified three main options:

1. Steal or obtain valid login credentials (username and password) and log in as a legitimate user.
2. Bypass the login entirely, meaning they access the patient data without ever needing to authenticate.
3. Use an active user's session cookies to access an existing logged-in session.

After this, we looked more closely at each of these branches. For gaining user credentials, we included sniffing login details through a man-in-the-middle (MITM) attack, which is possible if the connection uses insecure HTTP. Since the login data is sent in plaintext, an attacker looking at the traffic could read the username and password easily. We also included brute force password guessing as another possibility, which allows hackers to potentially figure out passwords through a trial and error script. The user database stores passwords in plain text and so if an attacker gained access to the database they would

immediately see all password values. We also added stored XSS to the session-cookie branch, as injected scripts can steal a GP's session cookie and eventually fully access patient data.

For the branch where the attacker bypasses login, we listed SQL injection as a path. This could be done through the login page by inserting a malicious SQL query that forces the system to log them in without valid credentials. Another way to bypass the login is again through a MITM attack over insecure HTTP, where the attacker might intercept patient data being sent from the server even without having logged in themselves. We also included CSRF, where the attacker could trick an authenticated user into making unintended requests, potentially causing unauthorised actions that expose data.

In the attack tree, each flaw identified is represented by a leaf node, showing how that vulnerability helps the attacker move towards the top goal. We assumed for the attack tree that the attacker has technical knowledge, the ability to intercept web traffic on an unsecured network, and access to the public-facing parts of the system (such as the login page). We also assumed that the attacker could potentially access the user database.

3. Implementation of Security Fixes

3.1 Fix Identification

3.1.1 SQL Injection Attack

SQL injection is one of the child nodes under “Bypass Login” node. To remove this path from the attack tree, we stop the attacker from being able to insert a SQL injection through the username/password fields.

The way to do this is by making sure any input into the user login is treated only as raw data and not as part of the SQL query itself.

3.1.2 Insecure HTTP Protocol

Exploiting the insecure HTTP protocol is a child node under the “Sniff login creds through data packets” ← “Access user credentials” ← “Access Patient Data” nodes. To remove this path from the attack tree, we stop the attacker from being able to intercept unencrypted data packets in transmission.

By using HTTPS instead of HTTP, we remove this risk, since all data sent over the network would be encrypted.

3.1.3 Stored Cross-Site Scripting (XSS)

Stored XSS corresponds to the “Session cookies → Access database → XSS exploit” branch of the attack tree. To remove this path, we prevent the injected code from executing when patient details are loaded.

The appropriate fix is to HTML-escape all patient fields in details.html using Freemarker’s ?html operator. This ensures that any stored payload is treated as text rather than executable code, blocking the attacker’s ability to run malicious scripts and altering/exploiting the database.

3.1.4 Password Hashing

Unhashed database information is a child node of the “Looking at user database” ← “Access user login credentials” nodes. To remove this path from the attack tree, we need to make sure passwords are hashed so they cannot be guessed even when looking at the database.

The appropriate fix is to hash the password using a randomly generated salt. This way, we authenticate users by comparing the computed hash of the attempted password against its stored one.

3.1.5 Cross-Site Request Forgery (CSRF)

CSRF is a child node of “Bypass Login” node. To remove this path, we must distinguish between a genuine form submission from our own site and a forged request coming from another origin.

The appropriate fix is to use a double-submit cookie token system, where a random token is set in a cookie and embedded in a form field, and the server rejects POST requests where the values do not match.

3.2 Fixes Implemented

List of changes made for flaws:
1. Changed login query to use placeholders for inputs instead of mixing in into the query.
2. Changed ServerApp.java to use ServerConnector and sslContextFactory for secure HTTPS connection
3. Stored Cross-Site Scripting (XSS)
4. Added password hashing
5. Added CSRF

3.2.1 SQL Injection Attack

The fix I implemented was to stop the login query from directly taking the username and password as raw strings and putting them straight into the SQL command. Instead of user input going directly inside the query itself, placeholders are used to keep the user's input separate from the actual SQL structure. This means the user input is only treated as data, so it can't change the query.

To do this, I used a PreparedStatement, which makes it easy to add ? placeholders inside the query for the username and password. I changed the AUTH_QUERY so the structure of the SQL would be fixed so that users cannot alter it. I replaced all '%s' with ?, which separates the user input from the actual query.

I also updated the authenticated function to use the PreparedStatement, which safely puts the username and password into the placeholders. This sends the input to the database purely as values, not as part of the SQL command. This means the original query cannot be rewritten by the user anymore, the input can only fill the ? and not modify the SQL itself.

3.2.2 Insecure HTTP Protocol

The fix I implemented for this fault was changing the configuration of the embedded Jetty server from an HTTP configuration to an HTTPS.

To do this, I set up a self-signed keystore (keystore.jks) to provide the server with a certification and a private key. Then, I modified the embedded Jetty server configuration in the AppServer.java file to accept HTTPS using SslContextFactory to set the keystore path and hardcode the password. I then used ServerConnector to link the server to that specific SSL context and added it to the Jetty server.

By doing this, all traffic is now encrypted, preventing mid-transmission interception.

3.2.3 Stored Cross-Site Scripting (XSS)

The fix for this flaw was to ensure that all patient fields in details.html are HTML-escaped before being rendered. The original template (Figure 3.1) used raw Freemarker expressions such as ${record.treated_for}, which allowed attacker-controlled HTML to be interpreted and executed by the browser. I replaced these raw expressions with the ?html escaping operator (Figure 3.2), which forces the application to treat any stored content as text rather than executable script.

```
<td>${record.surname}</td>
<td>${record.forename}</td>
<td>${record.dateOfBirth}</td>
<td>${record.doctorId}</td>
<td>${record.diagnosis}</td>
```

Figure 3.2: Original unescaped Freemaker output

```
<td>${record.surname?html}</td>
<td>${record.forename?html}</td>
<td>${record.dateOfBirth?html}</td>
<td>${record.doctorId?html}</td>
<td>${record.diagnosis?html}</td>
```

Figure 3.2: Updated template using the ?html escaping operator.

After applying this change, the XSS payload stored in the database was no longer executed. Instead, it appeared as literal text in the “Treated For” column when the page loaded. The output shown in Figure 4 confirms this behaviour: the <script> tag is displayed directly on the page, indicating that the browser is no longer interpreting the injected payload as code. This change removes the execution step required for the attacker to steal session cookies, thereby eliminating the stored XSS attack path.

Figure 4: Escaped payload displayed as plain text

3.2.4 Password Hashing

I fixed the password hashing by implementing a utility class which handles all things relating to password hashing (i.e. salt generation and password hashing). The salt was generated using a secure random number generator, which is provided in the built-in Java package java.security.SecureRandom. Next for the hashing of the password I used the PBEKeySpec class in combination with the SecretKeyFactory to retrieve a PBKDF2WithHmacSHA1 hash function, which I then applied to the password in order to return password hashes.

As for the integration I first altered the user database schema to include a password hash and salt, keeping the password column to not remove the currently stored passwords, then I wrote a migration script on startup of the server which checks if any users don’t have a stored password hash, and uses the password hasher in order to generate and store a salt and then use the salt to generate and store the hash. Once this was completed I was able to remove the migration script and delete the password column in the database. Finally for authentication we retrieve the password salt from the database, recreate the hash with the attempted password and only authenticate if the stored hash matches the newly generated hash.

3.2.5 Cross-Site Request Forgery (CSRF)

To fix the CSRF vulnerability, I added a CSRF token using the double-submit cookie pattern, and enforced it on every POST to the servlet. The aim was to ensure that only forms served by our own application can produce requests that the server accepts.

First, I modified AppServlet.doGet() so that each time the login page is rendered, the server either reuses an existing CSRF-TOKEN cookie or generates a new random token (using UUID). This value is set as a CSRF-TOKEN cookie on the response and also passed into the Freemarker model as csrfToken. In login.html I then added a hidden input field whose value is ${csrfToken}, so the same token is submitted with the form.

Next, I updated AppServlet.doPost() to enforce the check. On every POST, the servlet reads the CSRF-TOKEN cookie and the csrfToken form parameter and compares them. If either is missing or they do not match, the request is rejected with HTTP 403 and the login/search logic is not executed. Because an attacker’s page (such as the evil.html test page) cannot read or predict the cookie value, it cannot construct a form with a matching token. As a result, cross-site forged POSTs are now blocked, while legitimate submissions from the real login page continue to work.