COMP3911 Coursework 2
Pair Submission

Razvan-Antonio Berbece (sc19rab), Konstantinos Kakarelis (sc19kk)

1 Analysis of Flaws

1. **<u>SQL Injection</u>** - the POST form data is not sanitised on any of the client and server (application is vulnerable to information disclosure and elevation of privilege threats)
2. **<u>HTTP used over HTTPS</u>** - sensitive POST form data (usernames, passwords, patient surnames) can be eavesdropped and interpreted
3. **<u>Passwords are stored in plain text</u>** - the user passwords are stored in plain text in the SQL database, which means that they can easily be retrieved and interpreted
4. **<u>Information leak in HTTP response headers</u>** - the server's response headers hold information about the version of the server framework which can then be researched by attackers to find more Jetty exploits
5. **<u>Password auth is implemented poorly</u>** – authentication brute force attacks are possible because there is no tracking of failed login attempts or account lockouts
6. **<u>Users can have the same username</u>** – the database schema does not enforce unique username entries
7. **<u>SQL Server can be accessed without credentials</u>** – the sqlite3 database can be accessed both from code and in the terminal without any credentials, which means that bad actors with access to a machine running the web app can mutate or read the database

1.1 SQL Injection

The form fields on the “login.html” page have no validation on the client-side and are sent as is to the server. We proved this by looking at the actual payload of the HTTP POST request sent to the server. Additionally, there is no restriction on the form field details that a user can input in the user interface (anything can be input in the fields). The server does not do ANY input validation either, which means that the form field values are used in the SQL statements as is. We proved this by investigating the codebase, and (for example), the username and password strings provided by the client are formatted into the final SQL command without any other validation or checking.

This means that the service app is vulnerable to:

1. **Information disclosure** (attackers reading confidential patient data without needing the required access credentials)
2. **Elevation of Privilege** (attackers can use the system without being registered users)

<u>To Reproduce:</u>
- With no username & no password (Critical)
1. Input the “' or '1'='1” string into the username field
2. Input the “' or '1'='1” string into the password field
3. Submit POST form
The application will display “No records found.” in both cases, which means that the server ran the query with an empty surname string for an unauthorized user (no password or username provided).

We also found a way to query ALL existing patient records in the database without a valid credentials pair (Critical):
1. Input the “' or '1'='1” string into the username field
2. Input the “' or '1'='1” string into the password field
3. Input the “' or surname=surname; --” – *to get all entries for which the surname field exists*
4. Submit POST form

1.2 HTTP used over HTTPS

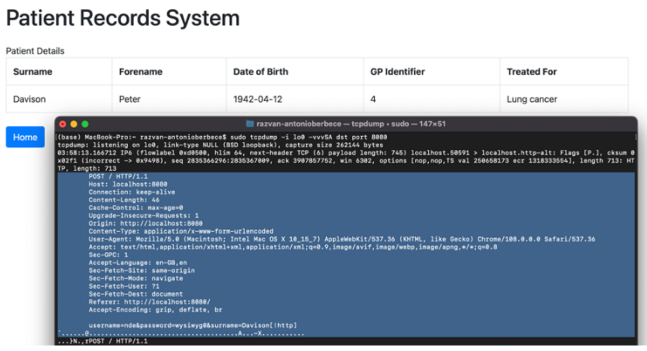
We discovered this vulnerability by checking that the protocol used in the URL of the web app (http://localhost:8080/) is indeed HTTP, and by confirming that the Jetty server setup code has no HTTPS configuration implemented or used for the server instantiation.

This vulnerability can be exploited by an attacker if they have access to a machine which runs the patient records application. They could then eavesdrop (through a ‘***tcpdump’*** command) on the outbound and inbound connections to and from the compromised machine and narrow it down through filtering it by protocol (i.e., HTTP) and / or the port number (i.e., 8080).

This means that the service is vulnerable to:

1. Information disclosure (of patient data from the response traffic, of account details from request traffic)
    a. And because attackers could have access to account login details through this, the system would be then vulnerable to other threats too (spoofing, elevation of privilege, etc.)

The picture below depicts what an attacker running the ‘***tcpdump’*** command on a compromised machine which runs the application in localhost could see in the case of an **outgoing request**.
Note that the POST form data sent (username, password, surname) is visible and not encrypted.


Patient Records System

Patient Details

Surname	Forename	Date of Birth	GP Identifier	Treated For
Davison	Peter	1942-04-12	4	Lung cancer

```
(base) MacBook-Pro:~ razvan-antonioberbece$ sudo tcpdump -i lo0 -vvvSA dst port 8080
tcpdump: listening on lo0, link-type NULL (BSD loopback), capture size 262144 bytes
03:58:13.166712 IP6 (flowlabel 0xd0500, hlim 64, next-header TCP (6) payload length: 745) localhost.50591 > localhost.http-alt: Flags [P.], cksum 0
x02f1 (incorrect -> 0x9498), seq 2835366296:2835367009, ack 3907857752, win 6302, options [nop,nop,TS val 250658173 ecr 1318333554], length 713: HT
TP, length: 713
        POST / HTTP/1.1
        Host: localhost:8080
        Connection: keep-alive
        Content-Length: 46
        Cache-Control: max-age=0
        Upgrade-Insecure-Requests: 1
        Origin: http://localhost:8080
        Content-Type: application/x-www-form-urlencoded
        User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36
        Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8
        Sec-GPC: 1
        Accept-Language: en-GB,en
        Sec-Fetch-Site: same-origin
        Sec-Fetch-Mode: navigate
        Sec-Fetch-User: ?1
        Sec-Fetch-Dest: document
        Referer: http://localhost:8080/
        Accept-Encoding: gzip, deflate, br

        username=nde&password=wysiwyg0&surname=Davison[!http]
`......@.....................................A...-X...........
...}N.,rPOST / HTTP/1.1
```

1.3 Password Security

The application has 3 main vulnerabilities when it comes to password security.
First two are that passwords are stored in plain text and the system allows weak passwords. This means the system is prone to brute-force attacks. These issues mean that a user profile could easily be compromised, either by an attacker gaining access to the database in some way and using the plain text passwords or through a brute force attack using common sequences and combinations of characters and numbers. **Finally, accounts with weak credentials could be created (taking the DB scheme into consideration): a username which is non-unique and a password that has no requirement of using a certain combination of numbers, uppercase and lowercase letters or special characters**.

These flaws were discovered by inspecting the SQL Database using a GUI tool (SQLiteStudio) and by looking at the database schema and the data stored in the tables.

These issues leave the system vulnerable to 4 different threat types:
1. Spoofing of Identity - gaining access of credentials of some other user and running queries as that user
2. Repudiation - reading authorized patient data from a compromised user account, without any logs to show proof of malicious or unexpected use
3. Information Disclosure - attacker gaining access to data they are not supposed to
4. Elevation of Privilege - anonymous user elevating to registered user

2 Fixes Implemented

2.1 Fix for SQL Injection

In order to fix the vulnerable SQL statement string formatting which allows SQL injection attacks, we used Java PreparedStatements.
The prepared statement query strings use a question mark to note where variables should be formatted in (e.g.: "select * from user where username=? and password=?"). **Because of this question mark placeholder feature, attackers won't be able to force close the starting apostrophe (') in a classic SQL Java formatted query string (e.g. "select * from user where username='%s' and password='%s'") via a malicious user input and add more query clauses to force the queries to be evaluated to true or return authorized data**. The prepared queries post-fix will look like this: "select * from user where username=' or '1'='1 and password=' or '1'='1", which will fail and clearly not evaluate to true, preventing the injection attack.
We applied the PreparedStatement fix (replacing the string interpolation construct '%s' with a question mark) to both AUTH_QUERY and SEARCH_QUERY as both queries use parameters received unsanitized from the client.
In the searchResults() and authenticated() methods, to pre-compile and prepare the statements, a PreparedStatement object is created and initialized using the .prepareStatement() method of an SQL connection. The placeholders are then switched to actual values using the .setString() method that a PreparedStatement object has access to using the parameter index and the variable value to inject in the pre-compiled statement.

2.2 Fix for HTTP used over HTTPS

In order to fix the vulnerable network protocol of the web application, we had to implement HTTPS support for the web app server and remove support for HTTP to force secure-only connections.
This involved creating a self-signed certificate (which means that if published on the wider internet, the web app will be considered insecure as it uses a self-signed certificate rather than a CA certificate) via the ***'keytool'*** UNIX command to generate a keystore and a 2048-bit size key using RSA. For deploying to the wider internet, generating a CA certificate would be recommended.
By setting up an SSL context for the Jetty Server through the self-signed certificate, the connection will now use the HTTPS protocol, and thus the POST data will be encrypted. As a result, eavesdroppers connected on a compromised machine won't be able to read the POST form data of the web application via a '***tcpdump'***
command.
We constructed the HTTPS configuration using the "HttpConfiguration" Jetty object initialized with secure options (https secure scheme, secure port, secure request customizer). We then constructed the SSL context using the "SslContextFactory.Server" Jetty object to link the web app to the self-signed certificate in local memory. To integrate it all together, we configured a "ServerConnector" Jetty object with the SSL context and HTTPS configuration defined above. The server connector is then added to the Jetty server and the connection is now secure.
Note: Post-fix, the web app access point http://localhost:8080/ changed to https://localhost:8080/.

2.3 Fix for Password Security

In order to secure the application, we went ahead and applied 3 different measures. First, we implemented a requirement of unique usernames in the database, however a fully robust implementation using more secure passwords would require a registration page, which is outside the scope of this assignment.
Secondly, to improve password security we added the use of SHA-256 hashed passwords with salt. Our salt is a statically declared Salt attribute to first convert passwords that already existed in the database. A fully secure implementation would require a randomly generated password salt and an empty database, which would salt passwords at the registration process.
Lastly, we implemented account lockout so if a user fails to login 3 times, they will not be able to login and a message is shown to inform them. Implemented a function `**getHashed**` that takes a plain string password, hashes it using our salt and returns it as a string. This is done using a `**MessageDigest**` object: **`MessageDigest md = MessageDigest.getInstance("SHA-256")`** and is updated using salt: **`md.update(salt)`.** As the md digest object is in bytes format, it is converted back to a string before being returned. The application uses the function to compare the text received as password in the login form **`stmt.setString(2, getHash(password))`**
The same function could be used in a registration process, hashing the new password before storing.