Cover Page – COMP3911 Coursework

Group Members

Full Name	Leeds Username	Leeds Student ID
Jacob Gaskill	sc21jg2	201518032
Fathan Raedaya	fy21fnr	201566023
Hao Zheng	sc22hz3	201635514
Hassan Nofal	sc22hn	201657720
Jonathan Sutanto	sc22j3s	201623420
Joe Barrington	ed20jb7	201530382

Full Name	Leeds Username	Leeds Student ID
Jacob Gaskill	sc21jg2	201518032
Fathan Raedaya	fy21fnr	201566023
Hao Zheng	sc22hz3	201635514
Hassan Nofal	sc22hn	201657720
Jonathan Sutanto	sc22j3s	201623420
Joe Barrington	ed20jb7	201530382

1. **Analysis of Flaws**

<u>**Flaw List**</u>

1) Insecure Password Management: User passwords are highly vulnerable if the database is breached, as they are stored in plaintext.

2) Brute Force Attack: The application allows unlimited login attempts, leaving the application vulnerable to a brute force attack.

3) No password complexity requirements: User passwords are not subject to any complexity requirements, leaving passwords vulnerable to guessing.

4) SQL Injection: An attacker can bypass authentication through malicious login form input, gaining access to the full patient records database.

5) Man in the Middle: The use of the insecure HTTP for application-database communication leaves login information and patient record data vulnerable to a MitM attack.

<u>**Detailed Analysis**</u>

1 - Insecure Password Management

Nature of the flaw
User passwords are stored in the database without encryption, with the app implementing authentication through direct plaintext comparisons between login form input and the stored passwords. If the database is somehow breached, all user credentials are immediately exposed without any cryptographic protection. An attacker can then use these credentials to access patient records at will.

Flaw discovery
When examining the database file, using DB Browser, we noticed passwords were stored in plaintext, without encryption. We examined the application code, to investigate further – which showed password comparisons were made without any involvement of a hash function.

Flaw example / evidence
The *authenticated()* method does not perform any hashing of password input before the SQL query is sent to the database, suggesting that a plaintext comparison is used. Figure 2 also demonstrates the flaw discovery.

Figure 1 – *authenticated()* method in *AppServlet.java* file, missing password hashing.

```
private boolean authenticated(String username, String password) throws SQLException {
  String query = String.format(AUTH_QUERY, username, password);
  try (Statement stmt = database.createStatement()) {
    ResultSet results = stmt.executeQuery(query);
    return results.next();
  }
}
```

Figure 2 – User database table viewed in DB Browser, with plaintext passwords visible.

id	name	username	password
Fil... Filter	Filter	Filter	Filter
1	Nick Efford	nde	wysiwyg0
2	Mary Jones	mjones	marymary
3	Andrew Smith	aps	abcd1234

2 - Brute Force Attack

Nature of the flaw
The application currently has no limit on login attempts, making it vulnerable to a Brute Force attack. This method programmatically attempts many username-password combinations to log in to the system, relying on sheer computational power to bypass authentication. Eventually, a combination may be guessed, allowing the attacker to access patient records at will.
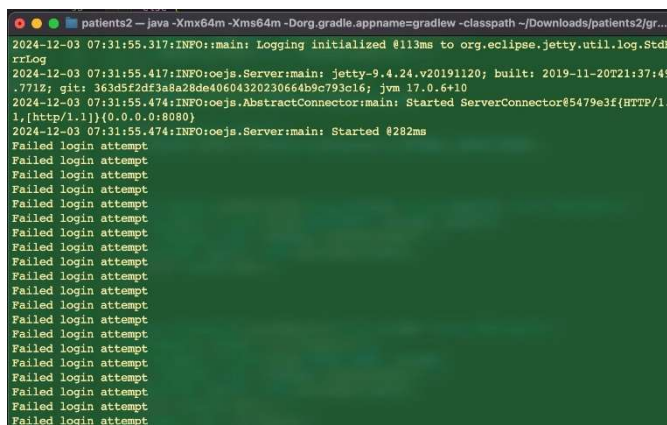
Flaw discovery
This flaw was discovered through analysis of the application codebase, which exhibited no attempt to limit the number of login attempts a user could make. We attempted several logins without limitation whilst testing the application, enforcing the fact that no limit was applied.

Flaw example / evidence
This flaw arises from a lack of features in the codebase, so cannot be illustrated through code snippets. However, figure 3 shows the result of adding a print method behind the login functionality and performing 20 manual attempts, with more possible. This is significantly higher than the typically recommended limit of between 3-5 logins attempts.

Figure 3 – Failed login attempt printed output.

3 - No Password Complexity Requirements

Nature of the flaw
User passwords in the database are not subject to any complexity requirements, with current cases including no special characters, passwords related to names, passwords of only eight characters and common character sequences, such as 'abcd1234'. These passwords are vulnerable to attackers guessing login details, given they have knowledge of usernames in the system, which could be obtained through a GP surgery public website.

Flaw discovery
This flaw was discovered through examination of the SQLite database file, with DB Browser.

Flaw example / evidence
The insecure passwords are visible in Figure 4, with the latter two cases exhibiting particularly poor password complexity requirements.

Figure 4 – User login details table, viewed in DB Browser.

4 – SQL Injection

Nature of the flaw
The application login form does not handle or sanitise user input correctly, leaving it vulnerable to SQL injection – where attackers can manipulate SQL queries through malicious input. The attacker can use specific SQL strings to trick the application into requesting all patient records from the database, without the need for authentication. The application then presents all records to the malicious user.

Flaw discovery

Whilst examining the login authentication, we discovered a vulnerability in how the user inputs are handled. The use of string concatenation for query generation was a red flag, with further inspection highlighting no sanitisation of input. Once this was spotted, we tested malicious SQL inputs in the login fields, which confirmed our findings – letting us bypass authentication and displaying all patient records.

Flaw example / evidence

Figure 5 shows an example of a malicious SQL string injection, which once submitted, causes the application to present the full patient records table.

Figure 5 – SQL Injection in the login form, and the output on form submission, containing all patient records.

5 – Man in the Middle

Nature of the flaw

The app is vulnerable to man in the middle attacks due to the use of unsecure HTTP. Without encryption, all data, including sensitive patient information and login credentials, is transmitted in plain text over the network. An attacker could intercept packets on the network, reading the sensitive data whilst in transit.

Flaw discovery

The vulnerability was discovered through code review of *AppServer.java*, which showed multiple instances of unsecure communication between the application and database using HTTP.

Flaw example / evidence

Figures 6 and 7 exhibit the unsecure communication setup used in the application codebase.

Figure 6 - Basic Jetty server setup using port 8080 without any SSL/TLS configuration.

```
Server server = new Server(port:8080);
server.setHandler(handler);
```

Figure 7 - No HTTPS requirement enforced in the servlet configuration.

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
```

Using WireShark, we could view sensitive data including login credentials and patient medical records being transmitted without encryption.

Figure 8 – Login details visible in WireShark output.

```
▸ Frame 9: 132 bytes on wire (1056 bits), 132 bytes captured (1056 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 6, Src: ::1, Dst: ::1
▸ Transmission Control Protocol, Src Port: 50454, Dst Port: 8080, Seq: 1559, Ack: 1, Len: 46
▸ [2 Reassembled TCP Segments (1604 bytes): #7(1558), #9(46)]
▸ Hypertext Transfer Protocol
▾ HTML Form URL Encoded: application/x-www-form-urlencoded
  ▸ Form item: "username" = "nde"
  ▸ Form item: "password" = "wysiwyg0"
  ▸ Form item: "surname" = "Davison"
```

2. Attack Tree

Figure 9 – Attack Tree visualisation.
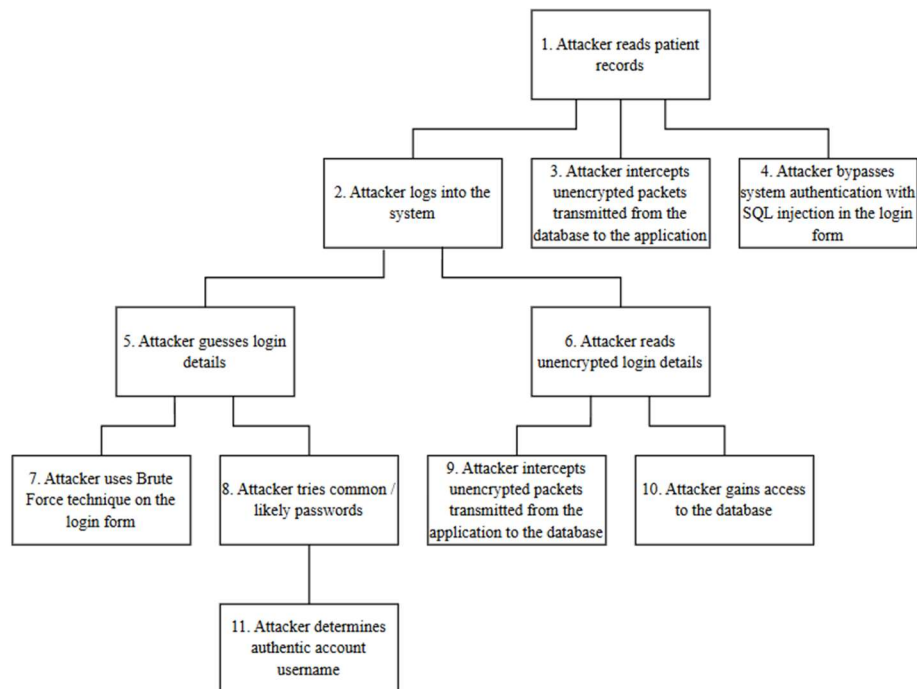
Tree construction method

The tree was constructed using visualisation software Draw.IO (https://app.diagrams.net/). The design approach was to start by considering all the methods the attacker can use to access the patient record data. Accessing the patient record data was considered to be the ultimate goal of any attacker, given the system holds no other assets. Numerical labels from 1-11 were applied in a breadth-first manner to nodes, instead of using the version-based approach (i.e. 1.2.1) to increase clarity at the lower levels of the tree.

Flaw exploitation

Flaw 1, Insecure Password Management, is exploited through the 10, 6, 2, 1 attack path. Node 6 is where the flaw is directly exploited, as attackers can simply read unencrypted login details from the database.

Flaw 2, Brute Force Attack, is exploited through the 7, 5, 2, 1 attack path. Node 7 is where the flaw is directly exploited, with attackers able to run a Brute Force Attack on the unlimited login form.

Flaw 3, No Password Complexity Requirements, is exploited through the 11, 8, 5, 2, 1 attack path. Node 8 is where the flaw is directly exploited, with attackers able to guess through simple / common passwords.

Flaw 4, SQL Injection, is exploited through the 4, 1 attack path. Node 4 is where the flaw is directly exploited, with attackers able to insert SQL into the login form, bypassing authentication and accessing patient records.

Flaw 5, Insecure HTTP, is exploited through the 3, 1 and 9, 6, 2, 1 attack paths. Nodes 3 and 9 are where the flaw is directly exploited, with attackers able to sniff unencrypted packets going between the database and application – containing both login details and patient data.

Assumptions

We assume that the application is hosted on the Internet, as opposed to locally on one of our devices – exposing many of the vulnerabilities. For example, the MitM attack requires data to be transmitted over a network, not just the local device hosting the application. Node 11 also assumes that some means of obtaining an account username exists, such as a public GP website, listing the staff at the practice.

3. Fix Identification

1 - Insecure Password Management
If passwords were handled securely, through the implementation of hashing, node 6 could be pruned – mitigating two attack paths. In both child nodes, when packets are sniffed, or the database is breached, the attacker could progress no further towards their goal, as any passwords they collect would be hashed. Hashing would both protect passwords stored in the database, and those in transit from the application, when used in authentication.

2 - Brute Force Attack
An attack stemming from node 7 can be directly mitigated through removing the ability to perform unlimited login attempts, whilst also taking steps towards mitigating an attack through node 8, as the attacker has less chances to guess a password.

3 - No Password Complexity Requirements
Nodes 11 and 8 highlight a common attack path, where an attacker may learn of usernames on the system in several ways, and subsequently exploit simplistic passwords. One example could be an attacker viewing the GP surgery website, learning of a GP, Mary Jones. Her username is 'mjones', which is easily inferable, and password 'marymary' is equally guessable. To prevent the 11-8 attack path, we add complexity requirements to the user passwords. This inhibits the attacker's ability to progress past node 8, as passwords will not be guessed easily.

The fixes described in sub-sections 2 and 3 mean we can prune the tree at node 5 – which subsequently means we can prune the tree at node 2, given node 6 has been pruned in sub-section 1. The three above fixes take steps to mitigate the attacker's ability to steal or guess authentic login details, significantly improving the security of the application.

4 – SQL Injection
Node 4 highlights SQL Injection, where an attacker can bypass the login feature to access the patients records directly. This node necessitates a targeted fix for this specific vulnerability, in which form input must be cleaned before any SQL is submitted to the database.

5 – Man in the Middle
Nodes 9 and 3 highlight vulnerability to data interception, between the application and database – especially given the use of unsecure HTTP. Both these nodes can be pruned if we simply encrypt the data that is transmitted, as any 'man in the middle' cannot read data they intercept.

Sub-sections 4 and 5 describe fixes that would allow us to prune nodes 3 and 4. This means that all of the above fixes take steps to mitigating the full attack tree, with all nodes being pruned.

4. Fixes Implemented

1 - Insecure Password Management

Summary of changes
The first step taken to fix the current insecure password management, was creating a *hashPassword()* method, which uses SHA-256 cryptographic hashing to encrypt the plaintext password. The *authenticated()* method takes the hashed password and combines it with a salt value, retrieved from the database. The combined value is used to compare with the corresponding (hashed) password in the database. Passwords previously stored in plaintext have also been hashed with the same algorithm and rewritten to the user table.

Fix effects
The use of hashing passwords mitigates several attack paths. If a data breach was to occur, attackers could not read the login information as passwords are now encrypted in the database, using SHA-256 cryptography. Furthermore, as passwords are now compared as hashes, any passwords that are intercepted whilst in transit to the database will not be readable. Ultimately, this protects the patient data as malicious users cannot steal credentials to access them.

2 - Brute Force Attack

Summary of changes
To address this issue, 2 new methods were made. The first method *lockAccount()*, is responsible for checking whether an account is currently locked. It creates a new hash map and will append the username to the hash map if the failed attempt exceeded the predefined limit of 5 login attempts. If the hash map is not empty, it will return a true value which triggers the account lock.

The second method is called *incrementFailedAttempts()*. This method handles the login logic and includes a counter, incrementing with each failed login attempt. Once the number of attempts has reached the limit, it will add the username to the hash map in *lockAccount()*, triggering the method – which locks the account for 5 minutes. The provided *authenticated()* method was also modified to include the *incrementFailedAttempts()* for an increase in workflow efficiency. Finally, a modification is made to the *doPost()* method, in order to fully implement the *lockAccount()* method.

Fix effects
The effect of this simple change will annihilate the ability of attackers to use automated scripts to attempt logging in using multiple username-password combinations. The fix ensures that after 5 failed attempts, the account will be temporarily locked for 5 minutes, preventing any access to the account. Following the cooldown period, each additional failed login attempt will trigger another 5-minute lock. This drastically inhibits the brute force attack as attackers will need to wait a long time to try each login combination, reducing the attacker's ability to work through credentials.

3 - No Password Complexity Requirements

Summary of changes
To fix the issue, we replaced the user table in the database with a new user table, updating the password schema to allow up to 50 characters, allowing for more complex passwords. We also updated the current users' passwords to include additional characters, symbols and numbers to further increase the complexity (before hashing was later applied). These updated passwords were generated randomly, with any data linking to the user account removed. If possible, a fix would be added to ensure any new users create passwords with a specified level of complexity, but this cannot be implemented within the current system, without creating additional webpages.

Fix effects
These changes addressed the application's vulnerability to password guessing, with user details no longer linked to their passwords. Furthermore, this fix mitigates any opportunistic common password guessing, such as 'abcd1234', as these types of passwords have been removed. The new passwords are of sufficient complexity, such that any attempts at guessing are very likely to be unsuccessful.

4 – SQL Injection

Summary of changes
To fix the vulnerability, we added Prepared Statements into the code. We changed how the SQL query was built and stopped using string concatenation, which is not a safe approach. The query now uses placeholders before it receives any data from the user, this placeholder is compiled by the database, so it expects data only to be passed to it and handles it as such.

The code calls *setString*() which sets the given data into the appropriate parameter. The database uses the value as data instead of code, meaning if an attacker was to enter a malicious string, such as " ' OR '1'='1 ", the database would use this value to check for a matching username or password in the table, instead of executing it as a query.

Fix effects
These changes fixed the flaw as any malicious data inputted by the user is no longer executed as code, but is read by the database as data. The user can no longer change the SQL query itself but can only submit data to be evaluated by the database. These fixes mitigate the attack path where an attacker can bypass the authentication feature and access the classified patient information.

5 – Man in the Middle

Summary of changes
The security of the server configuration was enhanced by implementing HTTPS using SSL/TLS with Jetty. Security headers, including HSTS were added to strengthen protections. HTTPS was enforced through the *ServletSecurity* annotation and set up SSL certificate validation using a configured keystore.

Fix effects
The implementation of HTTPS with proper security headers prevents MitM attacks by:
1. Encrypting all data in transit between client and server
2. Validating server identity through SSL certificates
3. Preventing protocol downgrade via HSTS header
4. Enforcing HTTPS-only connections through *ServletSecurity*