SOUTHWEST JIAOTONG UNIVERSITY•UNIVERSITY OF LEEDS
西南交通大学—利兹学院
SWJTU-LEEDS JOINT SCHOOL

Coursework Submission – Cover Sheet

Leeds Student ID Number: 201587324, 201587325, 201586907, 201587208, 201587281

SWJTU Student ID Number: 2021110033, 2021110034, 2021110081, 2021110003, 2021110021

Student Name: Yufei Wang, Yang Chen, Biliu Wang, Erfei Yu, Zihao You

Module Code & Name: XJCO3911 Secure Computing

Title of Coursework Item: Coursework 1

For the Attention of: Laha Ale

Deadline Time: 23:59

Deadline Date: 12.04

Student Signature: Yufei Wang, Yang Chen, Biliu Wang, Erfei Yu, Zihao You

For office use:

date stamp here

DECLARATION of Academic Integrity

I am aware that the University defines plagiarism as presenting someone else's work, in whole or in part, as your own. Work means any intellectual output, and typically includes text, data, images, sound or performance.

I promise that in the attached submission I have not presented anyone else's work, in whole or in part, as my own and I have not colluded with others in the preparation of this work. Where I have taken advantage of the work of others, I have given full acknowledgement. I have not resubmitted my own work or part thereof without specific written permission to do so from the University staff concerned when any of this work has been or is being submitted for marks or credits even if in a different module or for a different qualification or completed prior to entry to the University. I have read and understood the University's published rules on plagiarism and also any more detailed rules specified at School or module level. I know that if I commit plagiarism I can be expelled from the University and that it is my responsibility to be aware of the University's regulations on plagiarism and their importance.

I re-confirm my consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to monitor breaches of regulations, to verify whether my work contains plagiarised material, and for quality assurance purposes.

I confirm that I have declared all mitigating circumstances that may be relevant to the assessment of this piece of work and that I wish to have taken into account. I am aware of the University's policy on mitigation and the School's procedures for the submission of statements and evidence of mitigation. I am aware of the penalties imposed for the late submission of coursework.

Analysis of Security Flaws

- User input is not properly sanitized, allowing attackers to manipulate SQL queries.
- Passwords are stored without encryption, risking exposure if the database is compromised.
- Error messages reveal sensitive information about the application's structure.
- The system lacks mechanisms to detect and block repeated login attempts.
- Insecure session handling makes user accounts vulnerable to hijacking.

1. SQL Injection Vulnerability

Nature: Applications process user input without sufficient validation or filtering, leading to the possibility of SQL injection attacks. In particular, in AUTH_QUERY and SEARCH_QUERY, the user input is inserted directly into the SQL query without any escaping or parameterisation of the query.

Discovery: By examining the source code, We discovered this flaw by analyzing the authenticated method, where the username and password are directly embedded into the SQL query via String.format. This practice leaves the application open to SQL injection attacks.

An attacker could use SQL injection to bypass authentication, access sensitive data, or even corrupt database contents. If an attacker enters the following input in the login form:

- **Username:** anyusername
- **Password:** anypassword' OR '1'='1

The query becomes:

```
select * from user
where username='anyusername' and password='anypassword' OR '1'='1
```

This query will always return a valid result **('1'='1')**, bypassing authentication.

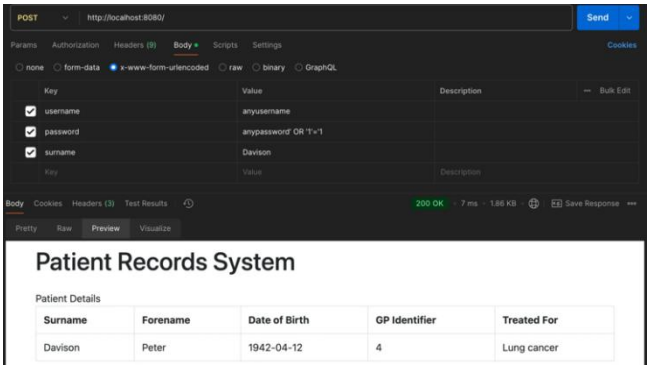


Figure 1. Resault of SQL Injection

2. Plaintext Password Storage

Nature: Passwords are stored and compared in plaintext, which poses a significant security risk if the database is compromised. Storing passwords in plaintext exposes user credentials and increases the chances of unauthorized access.

Discovery: We identified this flaw by reviewing the authenticated method where the password is directly compared against the database value without any form of hashing or encryption. The method simply executes a SQL query like this:

```
String query = String.format(AUTH_QUERY, username, password);
```

User database table where passwords are stored in plaintext:

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

Figure 2. User database table

If this database is compromised, an attacker can directly view the plaintext passwords of all users. This makes it easy for the attacker to misuse the credentials and access user accounts. For instance, if an attacker gains access to this table, they can instantly retrieve the passwords:

```
SELECT username, password FROM users;
```

This flaw is not merely theoretical. One well-known example of a breach caused by plaintext password storage is the **2019 Capital One data breach**, where an attacker gained access to more than 100 million customer accounts. Although in this case, Capital One did not store passwords in plaintext, the breach demonstrates the dangers of insecure data storage. Had passwords been stored in plaintext, the damage would have been far worse. Sensitive data, including usernames, email addresses, and, most alarmingly, plaintext passwords, could have been easily exploited by the attacker.

Example: Many high-profile data breaches in the past have been exacerbated by the lack of proper encryption of passwords. For example, in the **2012 LinkedIn data breach**, passwords were stored in an unencrypted format, and over 6.5 million passwords were leaked in plaintext. This led to widespread misuse of these passwords across various websites. If LinkedIn had employed proper password hashing techniques, the attackers would not have been able to retrieve user passwords in a usable form.

3. Information Disclosure via Error Handling

Nature: The application leaks potentially sensitive information in error messages. When an exception occurs, it only returns a generic error code (500) without providing any meaningful feedback to the user, but detailed errors may still be logged server-side, revealing information about the system.

Discovery: We discovered this issue by reviewing the error handling in the doPost and doGet methods. The catch blocks capture exceptions and send a generic error response, but there is no mechanism to hide detailed error messages in server logs.

Figure 3. Miss Error Code and Message

The log mentions SERVLET: comp3911.cwk2.AppServlet-7dc5e7b4, which exposes the class path and the servlet name (comp3911.cwk2.AppServlet) of your application. With this information, an attacker could infer the following:

- Your application uses Java Servlet technology.
- The servlet class is located in the comp3911.cwk2 package.
- The specific servlet implementation class is AppServlet, and it might be associated with database operations or sensitive actions (such as user authentication).

Although this information may provides potential clues that could help them deduce the system architecture, functionalities, or possible vulnerabilities.

4. No Protection Against Brute Force Attacks

Nature: The application does not implement any mechanism to limit login attempts, making it susceptible to brute force attacks. An attacker can attempt a large number of password guesses in order to eventually discover the correct password.

Discovery: We identified this vulnerability by reviewing the login process in the doPost method. Upon recognizing that there is no account locking or delay mechanism after multiple failed login attempts, it became clear that the application lacks basic protections against brute force attacks. Without these defenses, an attacker could use a script to try multiple passwords in a short period of time, significantly increasing the chances of successfully guessing a valid password.

5. Session Management Issues

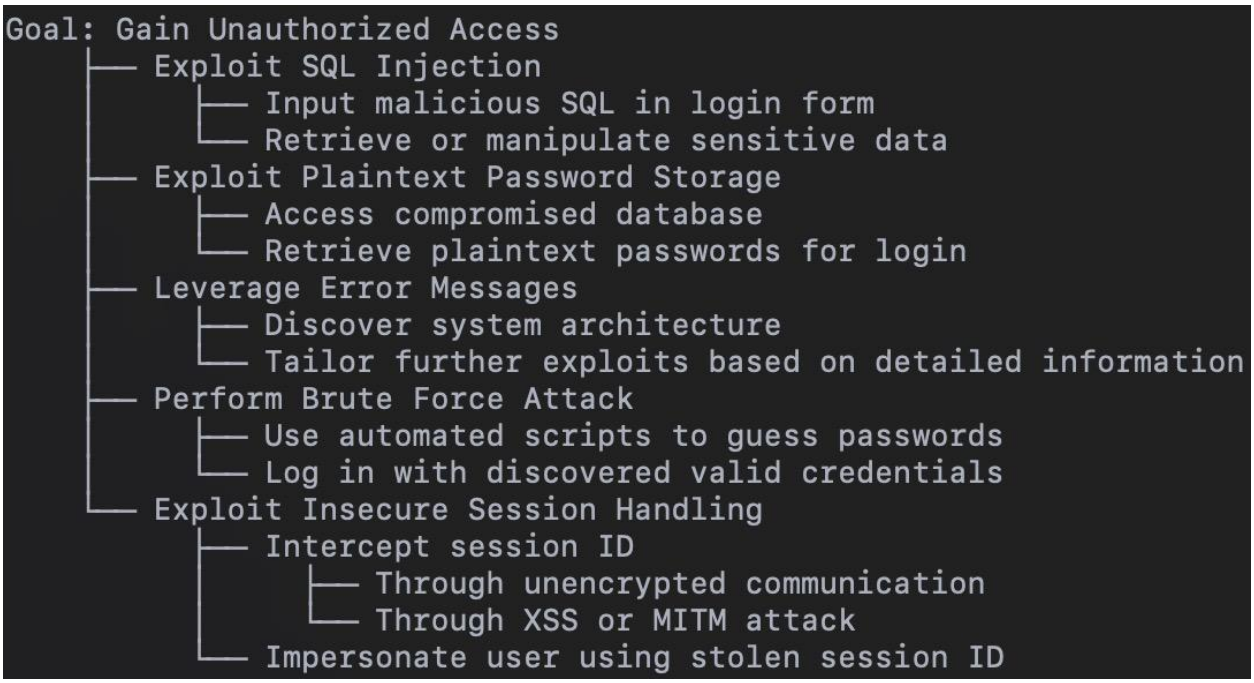
Nature: The system lacks secure session management practices, which poses a risk of session hijacking, fixation, or unauthorized access. Specifically, there is no implementation of secure cookie handling, session expiration, or regeneration of session IDs upon login. Without these basic session management mechanisms, the system is vulnerable to attacks where an attacker can hijack an existing session or impersonate a legitimate user.

Discovery: We identified this flaw by reviewing the **doPost** method, which is responsible for handling user login requests. It does not utilize secure cookies, set proper session timeouts, or regenerate session IDs upon user authentication. Upon successful login, the system should create a session and store session data related to the user. This can be done by calling HttpServletRequest.getSession(), which generates a session ID that should be stored in a secure cookie. However, this crucial step is missing in the current implementation.

Example: Consider the scenario where an attacker is able to intercept the session ID through methods such as packet sniffing or man-in-the-middle (MITM) attacks. If the session ID is stored in a regular, unprotected cookie, the attacker can steal the session ID and use it to impersonate the user. For example, if an attacker is able to access a user's session cookie through XSS or by capturing traffic on an unencrypted HTTP connection, they can use this session ID to gain unauthorized access.

Attack Tree

Attack Tree Visualization



Attack Tree Explanation

1. SQL Injection Vulnerability

- The attacker inputs malicious SQL (e.g., anypassword' OR '1'='1) in the login form.
- The query bypasses authentication or extracts sensitive data from the database.

2. Plaintext Password Storage

- If the database is compromised, passwords can be retrieved in plaintext.
- The attacker uses exposed passwords to log in as legitimate users.

3. Information Disclosure via Error Messages

- The attacker triggers errors intentionally to reveal sensitive information about the application’s structure.
- Detailed error logs provide clues about database name, table structures, or technology stack.

4. No Protection Against Brute Force Attacks

- The attacker uses scripts to attempt multiple passwords without restriction.
- Eventually, a valid password is discovered, granting access.

5. Insecure Session Handling

- The attacker intercepts a session ID through unprotected communication or malicious scripts.
- With the stolen session ID, the attacker impersonates a legitimate user.

Fix Identification

1. SQL Injection Vulnerability

- SQL queries were replaced with PreparedStatement, which parameterizes inputs and ensures they are treated as data rather than executable code.
- In the authenticated method, user inputs (username and password) are securely bound to the query, preventing malicious inputs from altering its logic.
- This fix addresses the attack tree's path where attackers use malicious inputs to bypass authentication and improves code maintainability and database compatibility.

2. Plaintext Password Storage

- Passwords were previously stored in plaintext, risking exposure if the database was compromised.
- The password column was updated to varchar(60) to store hashed values.
- A Golang script was used to hash existing plaintext passwords with BCrypt, which includes salting and computational cost, protecting against brute-force and rainbow table attacks.
- This fix aligns with password security best practices and directly addresses the attack tree's path involving exposed plaintext credentials.

3. Information Disclosure via Error Handling

- Detailed messages were replaced with generic ones like "An error occurred," while error specifics are logged server-side.
- In the doPost method, exceptions are logged with context, allowing developers to debug without exposing sensitive details.
- This fix mitigates the attack tree's path where attackers trigger errors to infer the application's internal structure or database schema.

4. No Protection Against Brute Force Attacks

- A login attempt tracking system was implemented, locking accounts after five failed attempts.
- Locked accounts automatically unlock after a set period, balancing security and usability.
- This fix directly addresses the attack tree's brute-force path, raising the difficulty of credential guessing while preserving user experience.

5. Session Management Issues

- A CSRF token mechanism was implemented, where unique tokens are generated per session and included in forms as hidden fields.
- Incoming requests are validated against the session-stored token, and invalid or missing tokens result in the request being rejected.
- This fix neutralizes the attack tree's CSRF exploitation path by ensuring all actions are initiated by authenticated users.

Fixes Implemented

1. SQL Injection Vulnerability

The application replaced SQL queries constructed with string concatenation by parameterized queries using PreparedStatement. Previously, concatenated queries allowed user inputs to alter the query structure, making the application vulnerable to attacks such as bypassing

authentication or data manipulation. By adopting PreparedStatement, user inputs are strictly bound as parameters, ensuring the query structure remains intact.

This change prevents SQL injection, improves security, and enhances maintainability by standardizing query execution. Additionally, parameterized queries reduce the risk of errors introduced by dynamic query construction, ensuring secure and consistent DB interactions.

2. Plaintext Password Storage

The application addressed the critical issue of plaintext password storage by implementing the BCrypt hashing algorithm. This ensures passwords are hashed before storage, using salting and computational cost to resist brute-force and rainbow table attacks.

The database schema was updated to accommodate hashed values, and a custom migration script was used to hash existing plaintext passwords securely. This transition avoided disruptions for users and aligned with modern security practices, significantly reducing the risk of credential exposure, even in the event of a database breach.

3. Information Disclosure via Error Handling

Error handling mechanisms were redesigned to prevent exposing internal system details to users. Detailed error messages, such as stack traces or database errors, were replaced with user-friendly messages like "An error occurred, please try again later." Internally, detailed logs are captured on the server to aid developers in diagnosing issues.

For example, failed database operations during authentication now log SQL exceptions server-side while presenting a generic error message to users. This dual-level handling protects sensitive information while supporting effective debugging.

4. No Protection Against Brute Force Attacks

A login attempt tracking system was implemented to prevent brute-force attacks. The system enforces a maximum number of allowed attempts (e.g., five) before temporarily locking the account, with automatic unlocking after a predefined period (e.g., 10 minutes).

These mechanisms minimize disruption for legitimate users while significantly increasing the difficulty and time required for brute-force attacks. Tracking attempts in memory ensures efficiency, and consistent locking logic provides a robust layer of defense without compromising user experience.

5. Session Management Issues

The application integrated a CSRF token mechanism to address the lack of session validation. Unique tokens are generated for each user session, stored securely, and included in form submissions as hidden fields. Incoming requests are validated against the stored token, with mismatches resulting in rejection.

For instance, during login, a CSRF token is generated and embedded in the form. Upon submission, the server verifies the token's authenticity, ensuring all actions are legitimate. This mechanism prevents unauthorized requests and establishes a standardized process for secure session management.