Umman Mammadov, Natig Mammadov, Mirhabil Sariyev

Case Study

Web Application Vulnerabilities and Security Best Practices

Objective of the project: Developing a Java-based web application to demonstrate potential security vulnerabilities and their secure implementations.

Scenario: Create a basic web application for a fictional company, the application has the following features:

1. A login page to authenticate users (username and password).

2. A message board where authenticated users can post and view messages.

3. A file upload feature to allow users to upload profile pictures.

Requirements

1. Handling HTTP Requests and Session Management

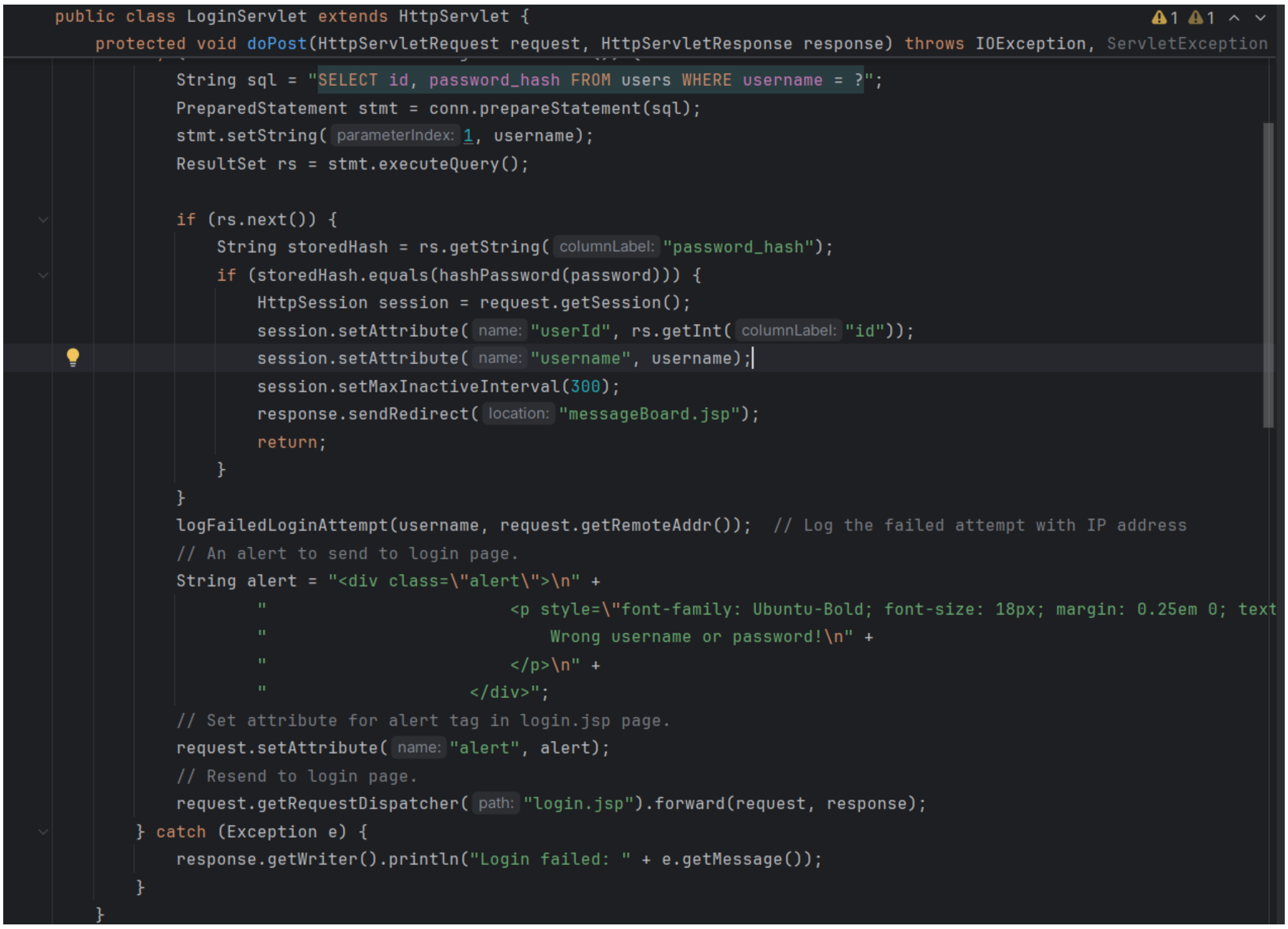
Implement a **Servlet** that handles:

- **Login (POST request)**: Validate the username and password (for simplicity, hardcode credentials: admin/admin).
- **Session Management**: Create a session for authenticated users and allow them to access a message board only if they are logged in.
- **Logout**: Invalidate the session.

Login Handling

Bad practice: Transmitting sensitive information (like username and password) through an open GET network request, which exposes credentials in the URL. This could result in the credentials being logged or caught.

Good practice: Utilize a POST request to transmit credentials securely within the request body. Ensure that the server validates credentials on the backend.

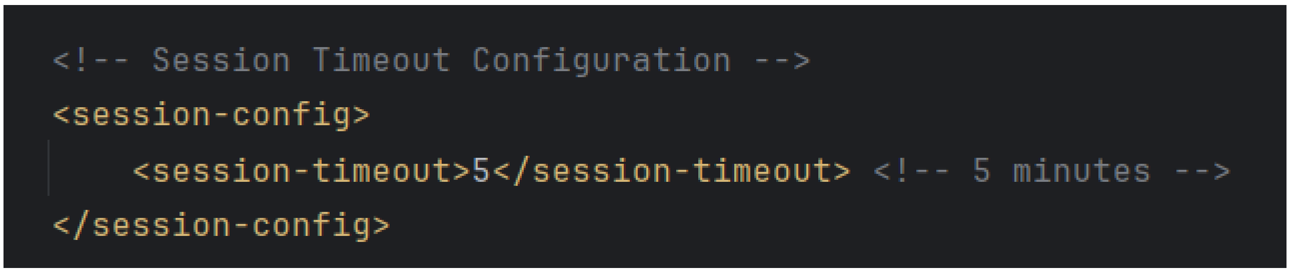


```
public class LoginServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException
            String sql = "SELECT id, password_hash FROM users WHERE username = ?";
            PreparedStatement stmt = conn.prepareStatement(sql);
            stmt.setString( parameterIndex: 1, username);
            ResultSet rs = stmt.executeQuery();

            if (rs.next()) {
                String storedHash = rs.getString( columnLabel: "password_hash");
                if (storedHash.equals(hashPassword(password))) {
                    HttpSession session = request.getSession();
                    session.setAttribute( name: "userId", rs.getInt( columnLabel: "id"));
                    session.setAttribute( name: "username", username);
                    session.setMaxInactiveInterval(300);
                    response.sendRedirect( location: "messageBoard.jsp");
                    return;
                }
            }
            logFailedLoginAttempt(username, request.getRemoteAddr());  // Log the failed attempt with IP address
            // An alert to send to login page.
            String alert = "<div class=\"alert\">\n" +
                    "                        <p style=\"font-family: Ubuntu-Bold; font-size: 18px; margin: 0.25em 0; text
                    "                            Wrong username or password!\n" +
                    "                        </p>\n" +
                    "                    </div>";
            // Set attribute for alert tag in login.jsp page.
            request.setAttribute( name: "alert", alert);
            // Resend to login page.
            request.getRequestDispatcher( path: "login.jsp").forward(request, response);
        } catch (Exception e) {
            response.getWriter().println("Login failed: " + e.getMessage());
        }
    }
```

Session Management

Bad practice: Lack of session validation, this leads to unauthorized access to protected resources or pages.

Good practice: Implement checks to verify the session's validity before allowing access. Additionally, manage session timeouts to enhance security.

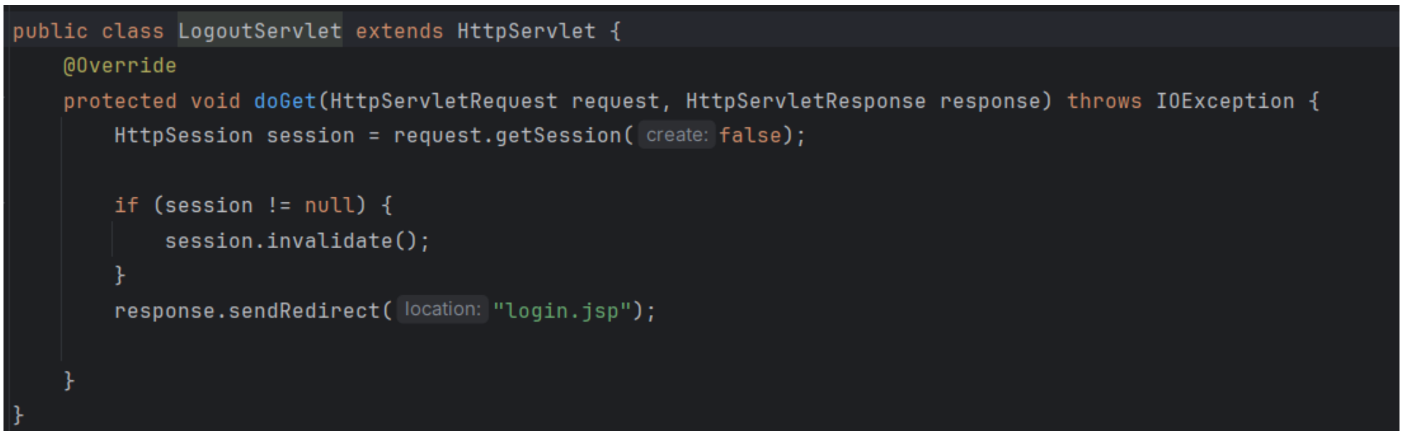


```
<!-- Session Timeout Configuration -->
<session-config>
    <session-timeout>5</session-timeout> <!-- 5 minutes -->
</session-config>
```

Logout Handling

Bad practice: The session persists even after the user logs out, increasing the risk of session hijacking.

Good practice: Explicitly invalidate the session upon logout to prevent unauthorized use.



```java
public class LogoutServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
        HttpSession session = request.getSession( create: false);

        if (session != null) {
            session.invalidate();
        }
        response.sendRedirect( location: "login.jsp");

    }
}
```

Key Takeaways

- Always use a POST request for sensitive data to prevent exposing it in URLs.
- Implement robust session management with proper timeout interval settings.
- Validate the session before granting access to sensitive pages.
- Ensure that sessions are invalidated upon logout to protect against session hijacking.

2. Injection

- Add a feature where users can search messages by keyword.
- Use a simple JDBC query to fetch messages containing the keyword from the database.

Bad practice: Embedding user input directly into SQL queries.

```java
String query = "SELECT * FROM messages WHERE content LIKE '%" + keyword + "%'";
```

Issue: The query is directly concatenating the user input (keyword) into the SQL statement, which makes it vulnerable to SQL Injection attacks.

Good practice: Use **Prepared Statements** with parameterized queries to safeguard against SQL Injection. This ensures that the input is treated as data rather than executable code.

```java
String query = "SELECT * FROM messages WHERE content LIKE ?";

PreparedStatement stmt = connection.prepareStatement(query);

stmt.setString(1, "%" + keyword + "%");

ResultSet rs = stmt.executeQuery();
```

By using a Prepared Statement, you ensure the query is executed with the input properly sanitized, mitigating the risk of SQL Injection.

Key Takeaways

- Never directly insert user input into SQL queries.
- Use **Prepared Statements** to distinguish between query logic and user data.
- Always check for SQL injection vulnerabilities using test payloads like ' OR '1'='1.

3. Broken Access Control

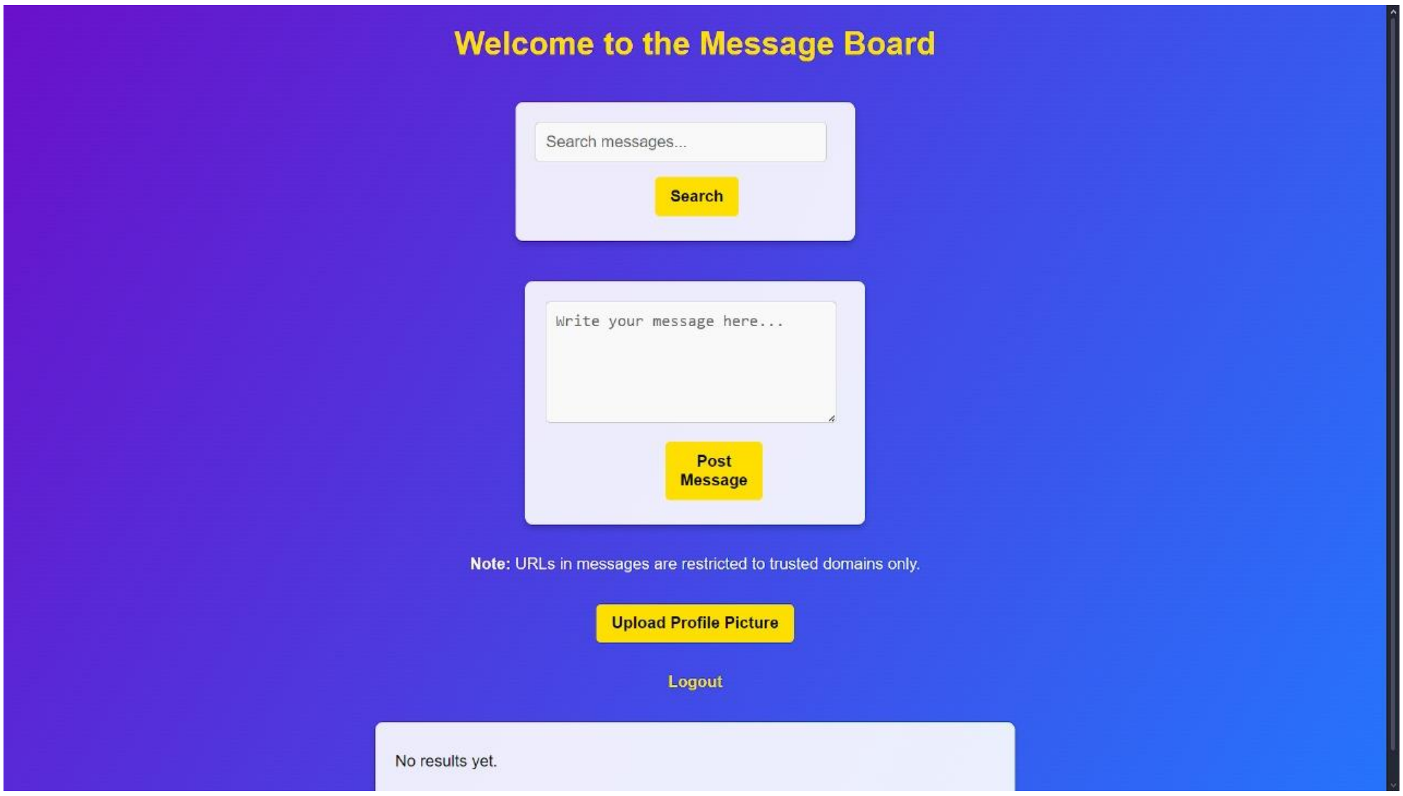
- Ensure only authenticated users can access the message board and upload files.
- Implement a Servlet filter to restrict access to sensitive endpoints.

Bad practice: Allowing unrestricted access to sensitive resources without checking the user's authentication status.

Issue: Unauthorized users could access protected pages or functionalities, potentially compromising the system.

Good practice: Use a **Servlet Filter** to enforce authentication checks before allowing access to sensitive endpoints.

- **How the Filter Works**
  - The filter intercepts all incoming requests (/*) and checks if the user has a valid session with the authenticated attribute.
  - Public resources like login.jsp or LoginServlet are excluded from these checks.
  - If the user is not authenticated, they are redirected to the login page.



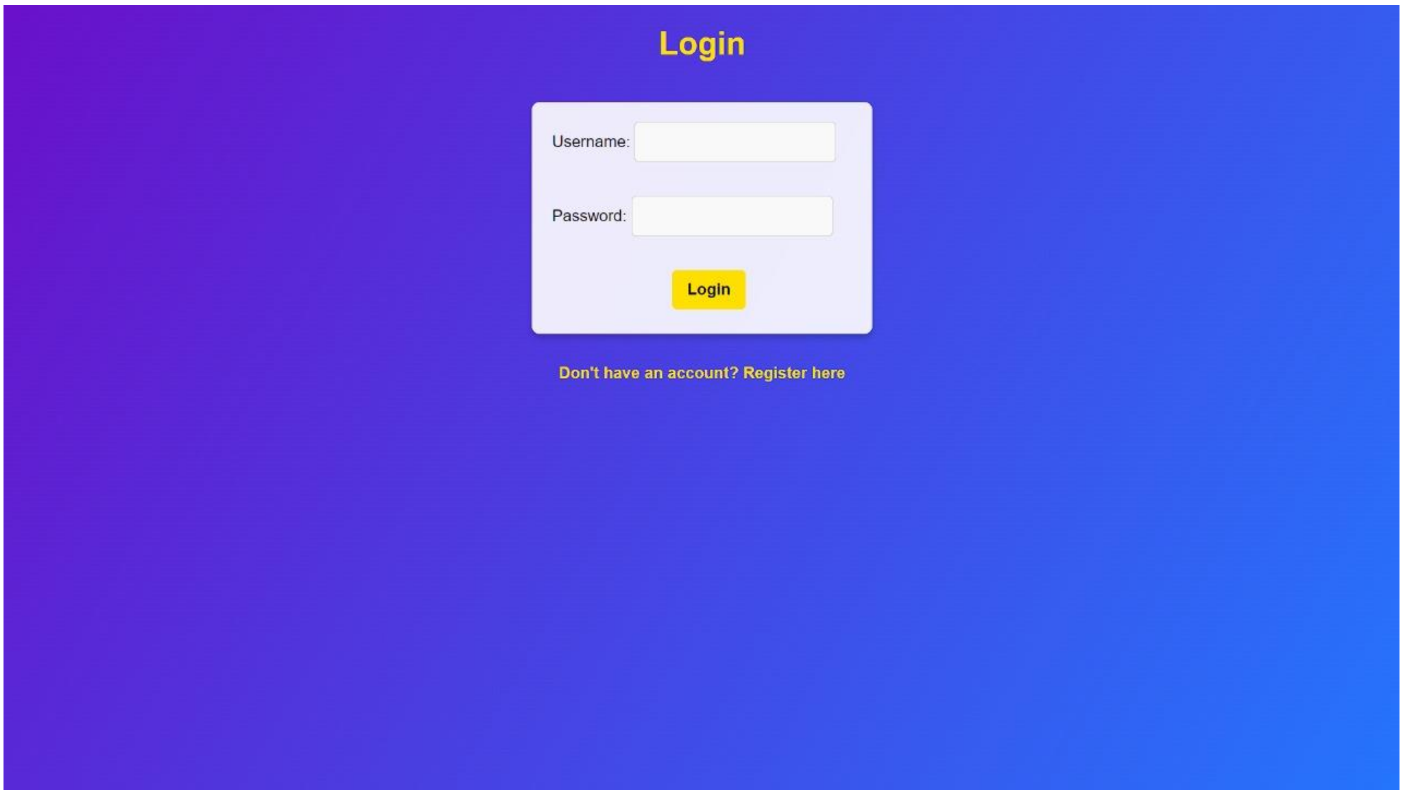
Note: URLs in messages are restricted to trusted domains only.

No results yet.

Integrating with Sensitive Pages

Message Board: Ensure that the /messageBoard.jsp page only serves content to users who pass through the filter and are authenticated.

File Upload: Apply the same filter mechanism to restrict access to file upload pages.



Key Takeaways

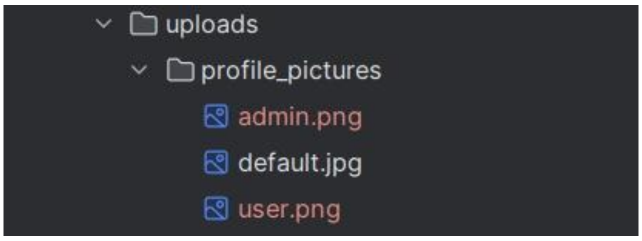
- A **Servlet Filter** provides a reusable, centralized solution to enforce access control across the application.
- Always validate user authentication before granting access to sensitive areas.
- Clearly distinguish between public and restricted resources in your application structure to prevent unauthorized access.

4. Sensitive Data Exposure

- Store uploaded files in a folder and store metadata in the database.
- Ensure files are served securely.

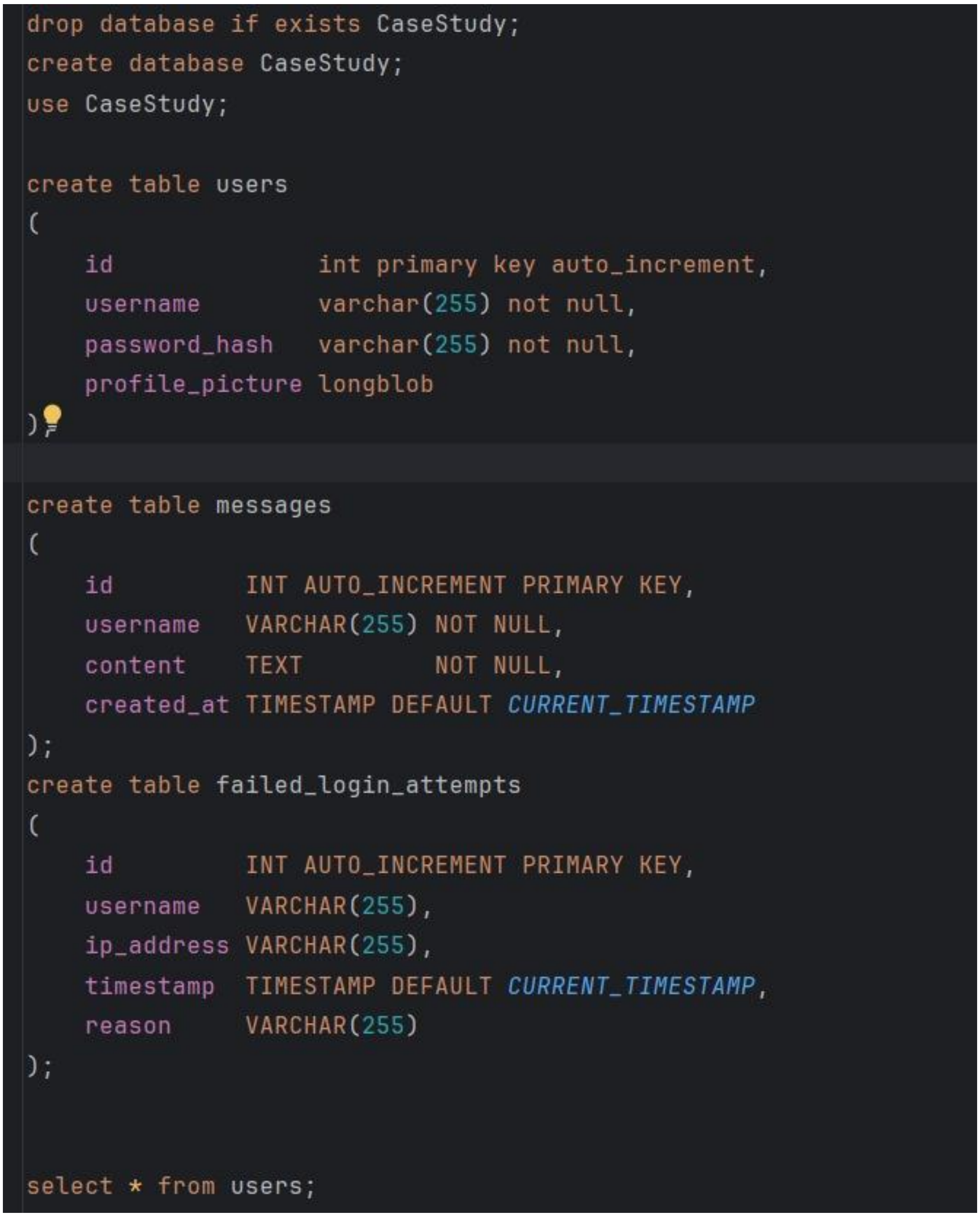
Bad practice: Storing uploaded files in a publicly accessible directory (/uploads) and permitting direct access via URL (http://secureweb.com/uploads/file.jpg).

Issue: This method could expose sensitive files to unauthorized users, compromising privacy and security.

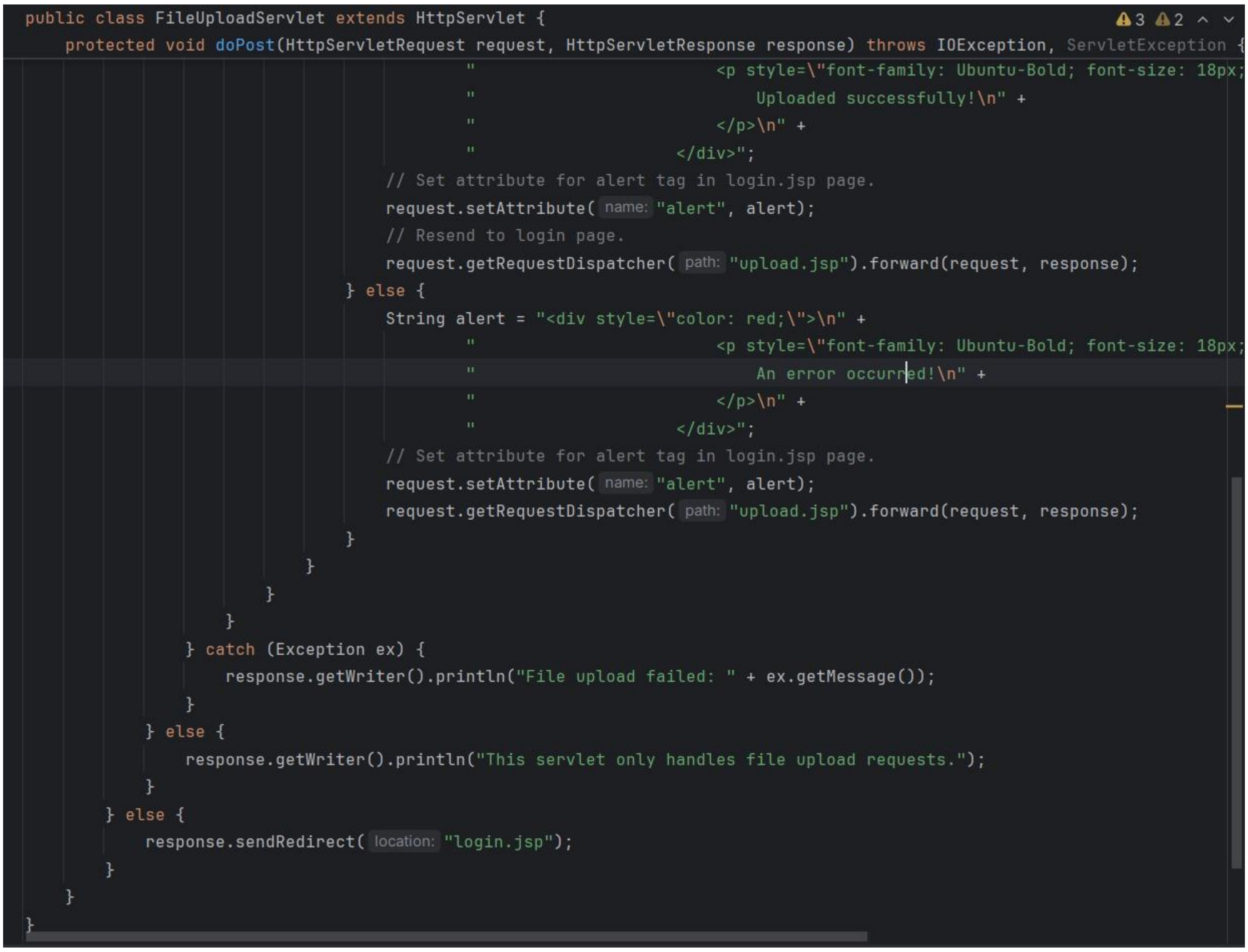


Good practice:

• Store files in a protected directory outside the web root (/opt/secure-web/uploads/) to prevent direct access.
• Store metadata such as file name, uploader ID, and access permissions in the database.
• Use a **download servlet** to check user permissions before serving files, ensuring only authorized access.



```sql
drop database if exists CaseStudy;
create database CaseStudy;
use CaseStudy;

create table users
(
    id              int primary key auto_increment,
    username        varchar(255) not null,
    password_hash   varchar(255) not null,
    profile_picture longblob
);


create table messages
(
    id         INT AUTO_INCREMENT PRIMARY KEY,
    username   VARCHAR(255) NOT NULL,
    content    TEXT         NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
create table failed_login_attempts
(
    id         INT AUTO_INCREMENT PRIMARY KEY,
    username   VARCHAR(255),
    ip_address VARCHAR(255),
    timestamp  TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    reason     VARCHAR(255)
);


select * from users;
```



```
public class FileUploadServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException {
                                            "                        <p style=\"font-family: Ubuntu-Bold; font-size: 18px;
                                            "                            Uploaded successfully!\n" +
                                            "                        </p>\n" +
                                            "                    </div>";
                                    // Set attribute for alert tag in login.jsp page.
                                    request.setAttribute( name: "alert", alert);
                                    // Resend to login page.
                                    request.getRequestDispatcher( path: "upload.jsp").forward(request, response);
                                } else {
                                    String alert = "<div style=\"color: red;\">\n" +
                                            "                        <p style=\"font-family: Ubuntu-Bold; font-size: 18px;
                                            "                            An error occurred!\n" +
                                            "                        </p>\n" +
                                            "                    </div>";
                                    // Set attribute for alert tag in login.jsp page.
                                    request.setAttribute( name: "alert", alert);
                                    request.getRequestDispatcher( path: "upload.jsp").forward(request, response);
                                }
                            }
                        }
                    }
                } catch (Exception ex) {
                    response.getWriter().println("File upload failed: " + ex.getMessage());
                }
            } else {
                response.getWriter().println("This servlet only handles file upload requests.");
            }
        } else {
            response.sendRedirect( location: "login.jsp");
        }
    }
}
```

Key Takeaways

- Never serve files from publicly accessible directories, as this can lead to unauthorized access.
- Use a **download servlet** to enforce access control and validate permissions before allowing file retrieval.
- Always sanitize and validate file metadata and any user input related to file storage and retrieval.

5. Cross-Site Scripting (XSS)

- Allow users to post messages on the message board.

Bad practice: Displaying user-submitted input on the message board without proper validation or sanitization.

Issue: Malicious users can inject harmful scripts (for example, <script>alert('XSS');</script>), leading to Cross-Site Scripting (XSS) vulnerabilities.

Good practice:

• Sanitize User Input: Ensure that user messages are sanitized before they are stored or displayed.
• Use libraries like **OWASP Java HTML Sanitizer** to clean HTML content and prevent XSS attacks.

Sanitization Example:

- Always encode user input when displaying it in the browser to prevent script execution.
- Use technologies like Java EE **JSTL** or frameworks with built-in mechanisms for HTML encoding.

Key Takeaways

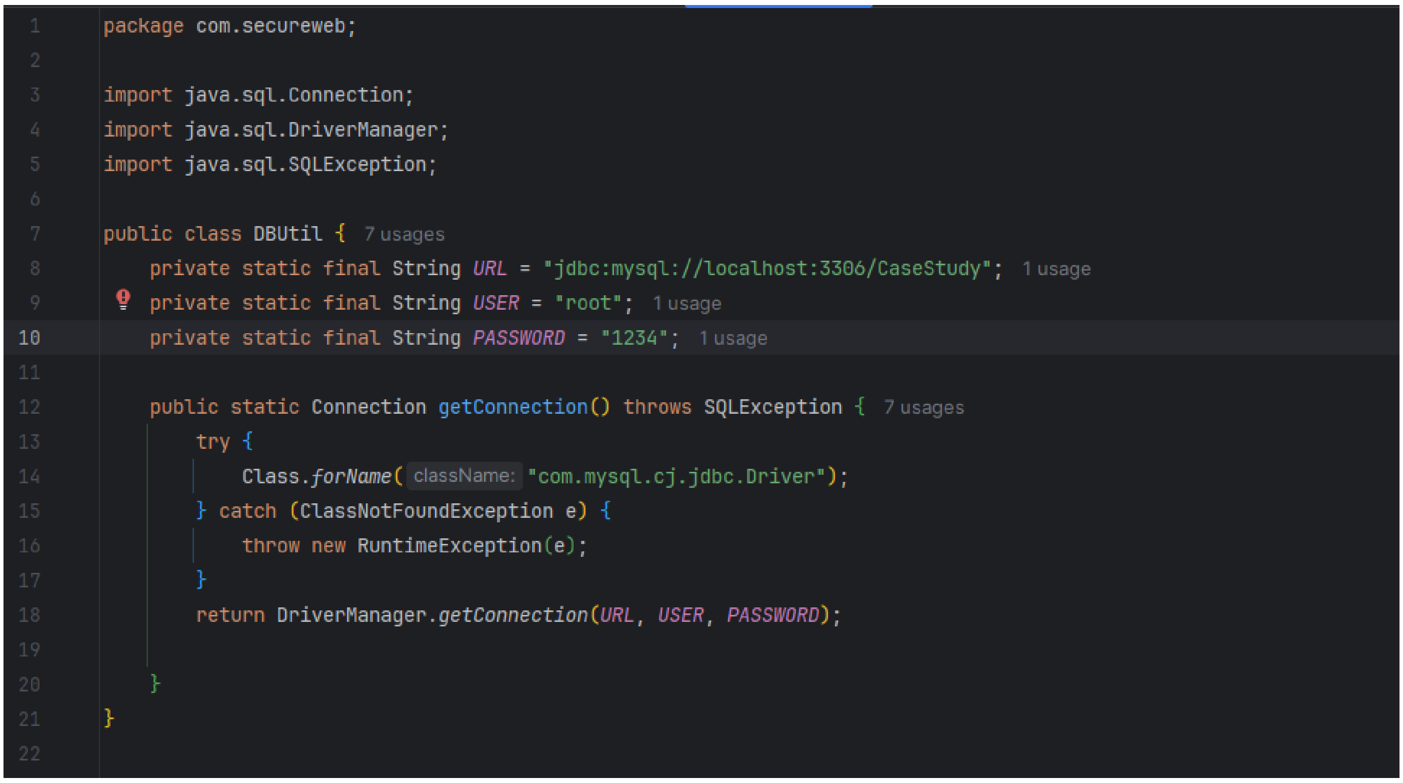
- Always sanitize user input using tools like **OWASP Java HTML Sanitizer**.
- Escape HTML characters when displaying user-submitted data to avoid triggering XSS vulnerabilities.
- Consider implementing **Content Security Policy (CSP)** headers to reduce the potential impact of XSS attacks.
- Test your implementation for both **reflected** and **stored** XSS attack vectors to ensure comprehensive protection.

6. Security Misconfiguration

- Provide a config file to store database credentials and application settings.

Bad practice: Hardcoding sensitive information such as database credentials directly into Servlets or application code.

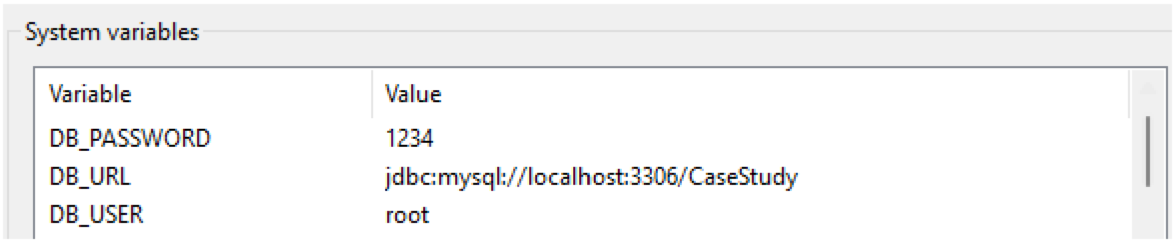
Issue:
• Credentials become exposed if the source code is leaked or compromised.
• Any changes to credentials require code modification, followed by redeployment, increasing the risk of errors.



```java
package com.secureweb;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBUtil {  7 usages
    private static final String URL = "jdbc:mysql://localhost:3306/CaseStudy";  1 usage
    private static final String USER = "root";  1 usage
    private static final String PASSWORD = "1234";  1 usage

    public static Connection getConnection() throws SQLException {  7 usages
        try {
            Class.forName( className: "com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
        return DriverManager.getConnection(URL, USER, PASSWORD);

    }
}
```

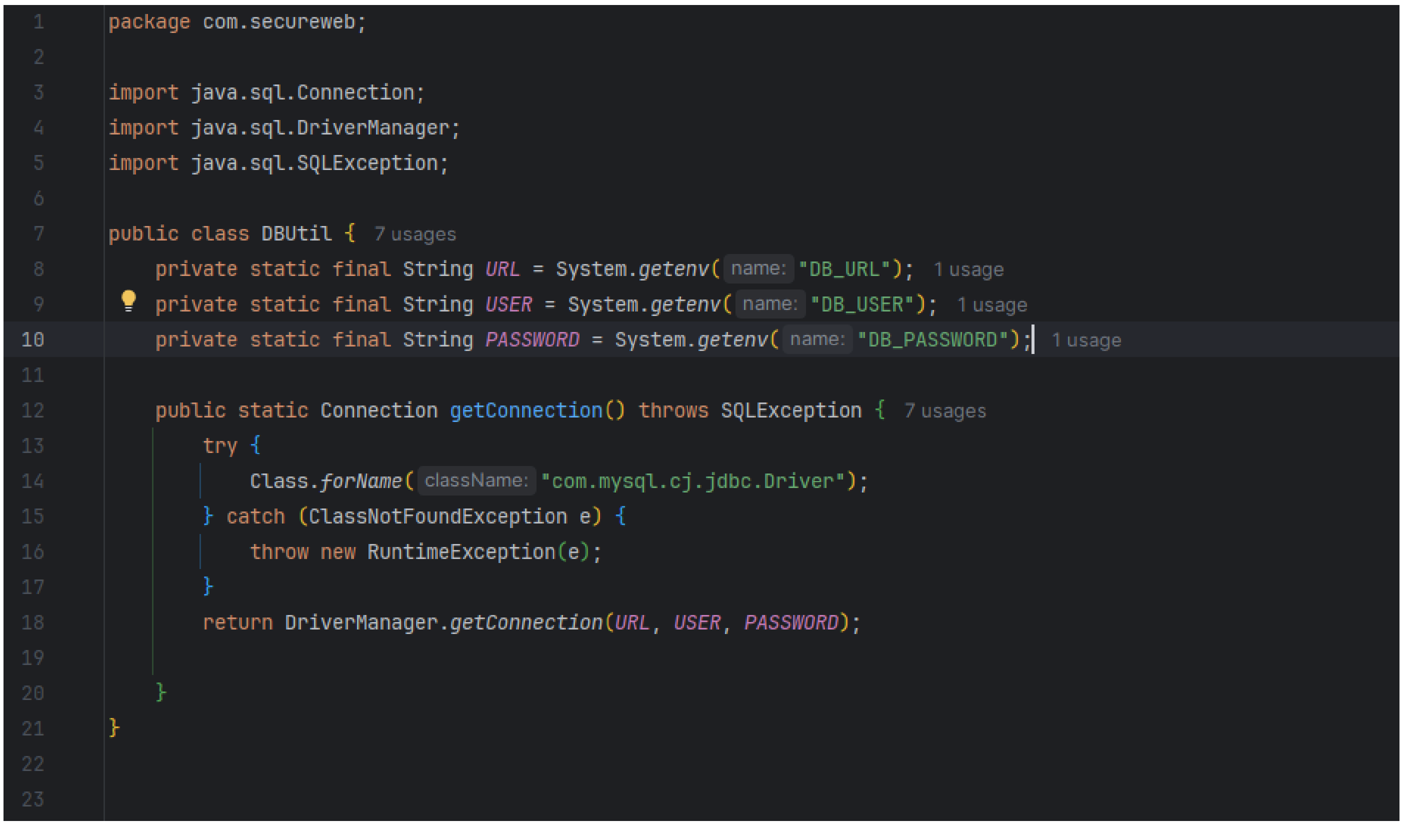
Good practice:

Store sensitive data such as credentials in **environment variables** or an **encrypted configuration file** to avoid embedding them directly in the code.



System variables

Variable	Value
DB_PASSWORD	1234
DB_URL	jdbc:mysql://localhost:3306/CaseStudy
DB_USER	root



```java
package com.secureweb;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBUtil {  7 usages
    private static final String URL = System.getenv( name: "DB_URL");  1 usage
    private static final String USER = System.getenv( name: "DB_USER");  1 usage
    private static final String PASSWORD = System.getenv( name: "DB_PASSWORD");  1 usage

    public static Connection getConnection() throws SQLException {  7 usages
        try {
            Class.forName( className: "com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
        return DriverManager.getConnection(URL, USER, PASSWORD);

    }
}
```

Key Takeaways

- **Environment Variables**: A simple and secure method for managing credentials, reducing the risk of exposure in the source code.
- **Encrypted Configuration Files**: Provide an extra layer of protection but require secure management of decryption keys.
- Always ensure **sensitive information** is kept out of source code, and limit **file access** on the server to authorized users only.

7. Insecure Design

- For the login feature, ensure passwords are not stored in plaintext.

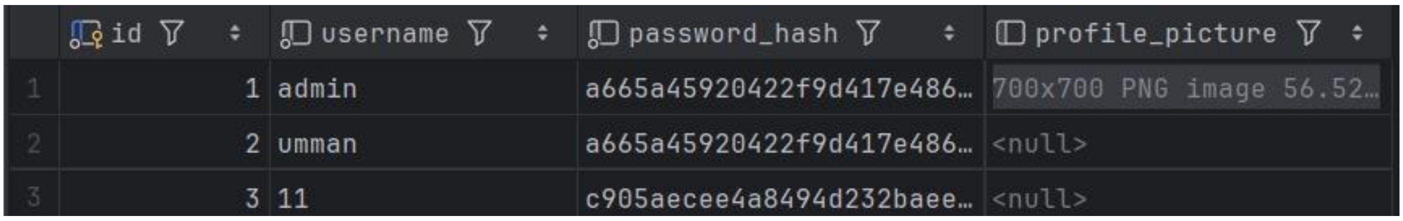
Bad practice:

Storing passwords in plaintext form in the database.

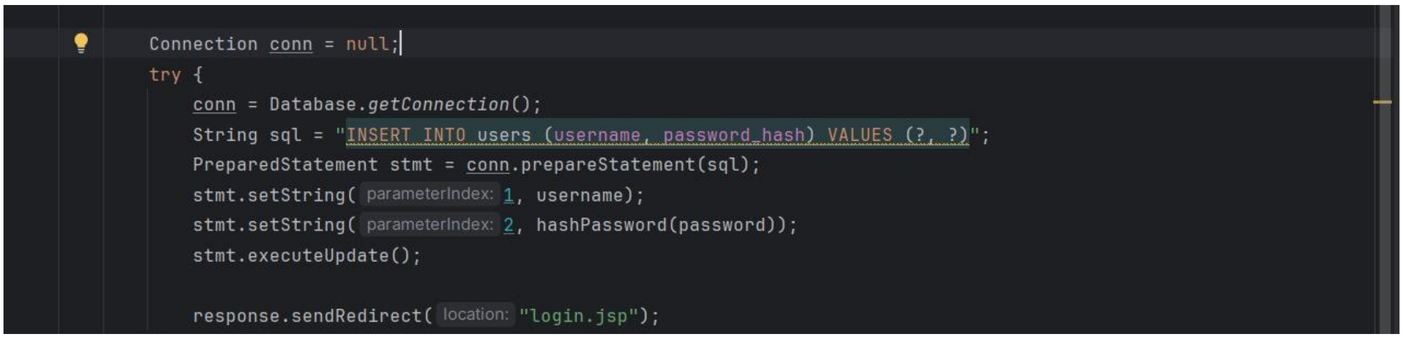
Issue:
If the database is compromised, user passwords are exposed without any protection.

Good practice:

• Use a **hashing algorithm** like **SHA-256** or **Bcrypt** to securely store passwords.
• **Salt**: Append a unique random value to each password before hashing, making attacks like rainbow tables infeasible.
• **Work Factor**: This allows you to adjust the computational cost to slow down brute force attacks and make password cracking more difficult.



	id	username	password_hash	profile_picture
1	1	admin	a665a45920422f9d417e486…	700x700 PNG image 56.52…
2	2	umman	a665a45920422f9d417e486…	<null>
3	3	11	c905aecee4a8494d232baee…	<null>



```
Connection conn = null;
try {
    conn = Database.getConnection();
    String sql = "INSERT INTO users (username, password_hash) VALUES (?, ?)";
    PreparedStatement stmt = conn.prepareStatement(sql);
    stmt.setString( parameterIndex: 1, username);
    stmt.setString( parameterIndex: 2, hashPassword(password));
    stmt.executeUpdate();

    response.sendRedirect( location: "login.jsp");
```

Key Takeaways

- Never store passwords in plaintext; always hash them.
- Use strong hashing algorithms such as **SHA-256** or **Bcrypt** to ensure password security.
- Validate user passwords using secure comparison methods during login.
- Regularly adjust the **work factor** of your hashing algorithms to keep up with advancements in computing power.

8. Insecure Deserialization

- Allow users to save and load their session data as serialized objects.

Bad practice:

• Deserializing user-supplied data without validation.

Issue:
• Allowing untrusted data to be deserialized can let attackers inject harmful objects, leading to potential code execution or exposure of sensitive data.
• This creates a risk of arbitrary code execution or unauthorized access to data if malicious objects are processed.
• The application is vulnerable to attacks like **Java deserialization** attacks using specially crafted payloads.

Good practice:

• Always **validate** and **sanitize input** before deserialization.
• Employ techniques like **object validation**, **class whitelisting**, or use safer alternatives like **JSON** for serialization to mitigate risks.

Key Takeaways

- Never deserialize untrusted or unchecked user input.
- Use whitelisting for allowed classes or safer methods like **JSON** for serialization.
- After deserialization, validate objects to ensure they meet expected criteria.
- Apply robust data integrity checks (e.g., **HMACs** or **digital signatures**) to prevent tampering.

9. SSRF (Server-Side Request Forgery)

- Add a feature where users can test connectivity to a given URL.

What is SSRF?
Server-Side Request Forgery occurs when an attacker tricks a server into making unauthorized requests to internal or external systems. This can lead to data exposure, unauthorized access, or even control over internal services.

- **SSRF attacks** leverage improper input validation in server-side requests to access internal services.
- Always use domain whitelisting, protocol validation, and set strict timeouts to reduce risk.
- **Log and monitor** user inputs and behavior to detect suspicious activity early.

Bad practice:

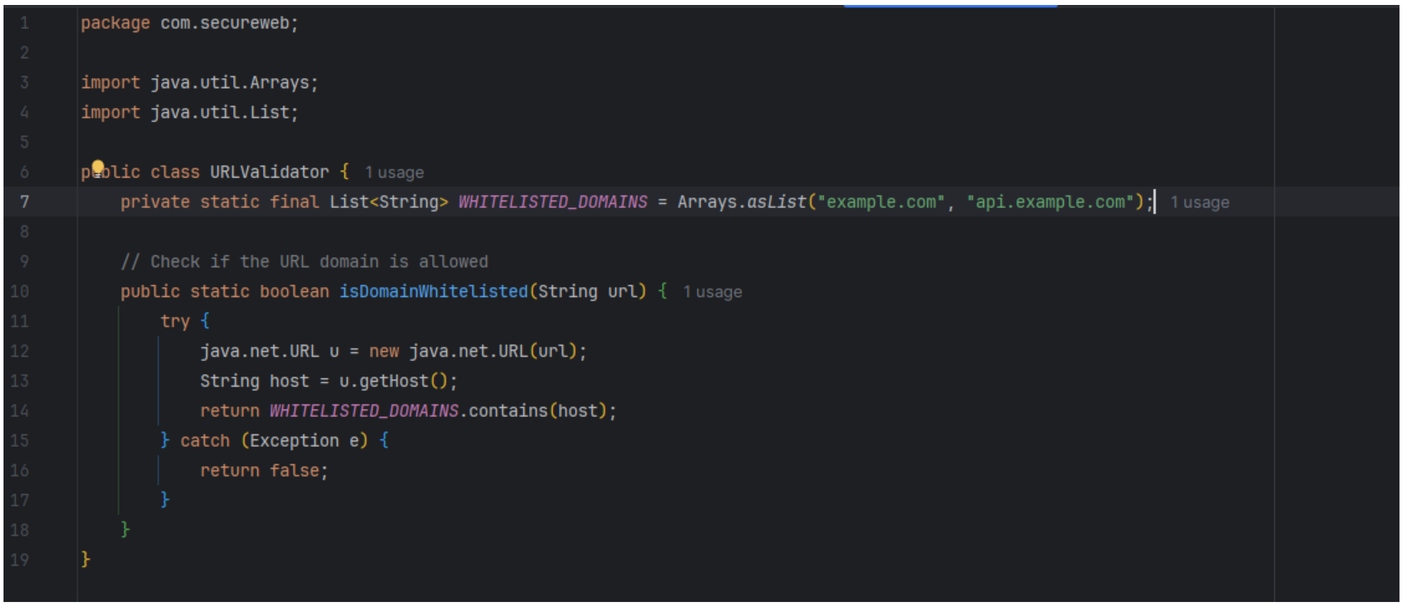
• Allowing unrestricted user input for URL connectivity testing poses several risks:
• Attackers may input **internal IP addresses** (e.g., http://127.0.0.1) to access internal resources, bypassing network security.
• Malicious URLs can **exfiltrate data** or **manipulate server behavior**, leading to potential information leaks or security vulnerabilities.

Good practice:

• Validate User Input
• Restrict URL input to a **whitelist** of trusted domains.

Validation Steps:

1. Parse the provided URL to extract its **domain**.
2. Ensure the domain matches an entry in the **whitelist**.



```java
package com.secureweb;

import java.util.Arrays;
import java.util.List;

public class URLValidator {  1 usage
    private static final List<String> WHITELISTED_DOMAINS = Arrays.asList("example.com", "api.example.com");  1 usage

    // Check if the URL domain is allowed
    public static boolean isDomainWhitelisted(String url) {  1 usage
        try {
            java.net.URL u = new java.net.URL(url);
            String host = u.getHost();
            return WHITELISTED_DOMAINS.contains(host);
        } catch (Exception e) {
            return false;
        }
    }
}
```

Key Takeaways - Key Security Measures

1. **Whitelist Domains:** Only permit requests to trusted domains.
2. **Validate Protocols:** Ensure that URLs use secure protocols, like **HTTPS**.
3. **Timeouts:** Implement timeout controls to prevent long delays or **Denial of Service** (DoS) issues.

10. Insufficient Logging and Monitoring

- Implement logging for each significant action (e.g., login, file upload, database query).

What is Insufficient Logging and Monitoring?
Failure to log critical actions or monitor logs effectively can leave security events undetected, such as failed login attempts, suspicious activity, or unauthorized access.

Bad practice:

Code without logging for failed logins

Issue:

- No record of failed login attempts, making it harder to detect brute force or credential stuffing attacks.
- No logging of significant actions like file uploads or database queries.

Good practice:

Log Significant Actions:

- Log **user login attempts** (both successful and failed).
- Log **file uploads** with metadata like file size, type, and uploader details.

- Log **database queries** for audit purposes (excluding sensitive details like plaintext queries).

Sanitize Logs:

- Prevent sensitive data (e.g., passwords, tokens) from being logged.
- Mask sensitive information where logging is necessary (e.g., masking part of IP addresses).

Store Logs Securely:

- Use tools like **Log4j** or **SLF4J** for structured logging.
- Store logs in a secure location with restricted access, such as a centralized logging server.

Secure Storage of Logs

1. **Log Retention**: Define a retention policy to keep logs for a specific period (e.g., 6 months).
2. **Access Control**: Ensure only authorized personnel can access logs.
3. **Use Centralized Logging**: Tools like ELK Stack (Elasticsearch, Logstash, Kibana) or Splunk can aggregate logs securely.

Key Takeaways

1. **Log Significant Events**: Include login attempts, file uploads, and database queries.
2. **Sanitize and Mask Logs**: Avoid logging sensitive details like passwords.
3. **Monitor Logs Regularly**: Use automated tools to detect anomalies like repeated failed logins or unusual activity.
4. **Centralize Logs**: Use secure logging frameworks and storage to prevent tampering.

Summary

This project involves implementing various security measures for a web application, focusing on common vulnerabilities and secure coding practices.

The first task is to handle HTTP requests and session management, ensuring login credentials are transmitted securely via POST requests, session validation is performed, and sessions are invalidated upon logout.

Next, the project addresses SQL injection risks by using prepared statements for database queries and properly validating user inputs. Access control is enforced using a filter to restrict access to sensitive pages, ensuring only authenticated users can view them. For file handling, files should be stored securely outside the web root, and access should be controlled through a download servlet that validates user permissions.

Cross-Site Scripting (XSS) vulnerabilities are mitigated by sanitizing user input and encoding HTML to prevent malicious scripts. Sensitive data, such as database credentials and passwords, should be securely managed by storing them in encrypted configuration files or environment variables.

The project also addresses insecure design by implementing proper password hashing techniques (e.g., bcrypt) to protect user passwords. Insecure deserialization is prevented by validating and sanitizing user input before deserializing objects, avoiding potential security breaches.

Finally, SSRF vulnerabilities are mitigated by restricting URL inputs to a whitelist and ensuring secure protocols, while logging and monitoring mechanisms are implemented to track significant actions, detect malicious activities, and store logs securely.