TP4 – Software Security: Audit of the Login Functionality

Joaquim KELOGLANIAN Eloise BOURNEUF

2025

Abstract

This document summarises the static and dynamic security testing performed on the application's login functionality. The focus is on relevant OWASP Top 10 categories and the CIA principles where applicable. Only vulnerabilities found in the login flow and associated services are reported.

1 Static Security Testing

The analysis targets the Java service classes handling authentication and persistence: `DataServices.java`, `Logins.java` and `SecurePassword.java`. Below are the findings grouped by file and OWASP category.

1.1 DataServices.java

SQL Injection (A03:2021) The code constructs SQL queries by concatenating user input into the query string. Example found in the login flow:

```
String sql = "SELECT id, firstname, lastname, password FROM users WHERE
    EMAIL='" + username +"'";
```

This query is executed via `createStatement()` with no parameterization.

Impact: an attacker can bypass authentication with payloads such as:

`username = ' OR '1'='1`

Recommendation: use parameterised queries (`PreparedStatement`). Example:

```
PreparedStatement ps = conn.prepareStatement(
   "SELECT id, firstname, lastname, password FROM users WHERE EMAIL =
      ?"
);
ps.setString(1, username);
ResultSet rs = ps.executeQuery();
```

Avoid `createStatement()` for user-supplied input.

Logging of Sensitive Data (A09:2021) The application logs full SQL statements, including user input:

```
System.out.println("dataService.doQuery : Execution of the query : " +
   sql);
```

Recommendation: never log raw SQL or user-supplied values. Log generic events or identifiers only.

Credentials in Source (A05:2021) Hard-coded database credentials were observed:

```
DriverManager.getConnection("jdbc:postgresql://localhost:5432/" +
    DataBase + "?user=postgres&password=postgres");
```

Recommendation: move credentials to a secure configuration (environment variables, `application.properties`) and avoid embedding secrets in source control.

Transport / Integrity (A08:2021) No SSL/TLS options were configured for the DB connection. When supported, enable TLS (for example, add `ssl=true` to the JDBC URL) and enforce secure channels.

1.2 Logins.java

SQL Injection (A03:2021) Same pattern as in `DataServices.java`: user input is concatenated into SQL queries. Ensure all queries are parameterised.

Authentication Failures (A07:2021) Observed issues: - Email regex gives limited validation but does not prevent SQL payloads. - No login throttling or account lockout is implemented (brute-force risk). - The Java session id is returned in the JSON response (`request.getSession(true).getId()`).

Recommendations: - Implement rate-limiting or progressive delays after failed attempts. - Do not expose raw Java session IDs in API responses; use a proper signed token (e.g. JWT) or a secure session management mechanism.

Access Control (A01:2021) There are no checks after login to enforce role or session verification on sensitive endpoints. Add a filter (e.g. `@WebFilter`) or middleware to verify session validity and roles for protected endpoints.

Components (A06:2021) Verify dependency and container versions. Use tools like `mvn dependency:tree` or dependency-check to detect vulnerable/outdated components.

Logging and Monitoring (A09:2021) Replace `System.out.println` with a proper logging framework (SLF4J + Log4j2 or similar) and ensure failed login attempts are recorded for audit.

1.3 SecurePassword.java

Good practice The implementation uses PBKDF2 and `SecureRandom` for salts which is better than plain hashes such as MD5.

Implementation Bug: salt/hash order The stored password format is expected as `salt:hash` but the parser was using the reverse. Incorrect parsing example found:

```
String slt = storedPassword.split(":")[1];
String pswd = storedPassword.split(":")[0];
byte[] salt = fromHex(slt);
byte[] hash = fromHex(pswd);
```

Fix:

```
String[] parts = storedPassword.split(":");
byte[] salt = fromHex(parts[0]);
byte[] hash = fromHex(parts[1]);
```

Cryptographic parameters (A02:2021) The iteration count (1024) is too low for modern standards. OWASP recommends using a much higher iteration count, and prefer `PBKDF2WithHmacSHA256` when using PBKDF2.

Recommendation example:

```
public static final int ITERATIONS = 310000;
SecretKeyFactory skf = SecretKeyFactory.getInstance("
    PBKDF2WithHmacSHA256");
```

Consider adding a server-side pepper (a secret key stored outside the DB) concatenated before hashing to mitigate offline brute-force if the DB is leaked.

2 Summary of Findings

Key findings and recommendations (short):

- **S-001 (A03:2021)**: SQL injection in `DataServices` — use `PreparedStatement`. - **S-002 (A05:2021)**: DB credentials hard-coded — move to environment/config. - **S-003 (A09:2021)**: Sensitive logging — avoid logging raw SQL/user input. - **S-004 (A07:2021)**: No brute-force protection on login — add rate limiting. - **S-005 (A01:2021)**: Missing post-login access checks — enforce role/session verification. - **S-006 (A02:2021)**: Weak PBKDF2 parameters — increase iterations and use stronger algorithm. - **S-007 (A02:2021)**: Salt/hash parsing bug in `SecurePassword` — correct parsing order.

3 Dynamic Security Testing

Dynamic testing notes and results should be added here (runtime tests, authentication attempts, intercepted responses, etc.).

4 Next Steps

- Fix SQL injection by parameterising queries. - Centralise secrets in a secure configuration or environment variables. - Replace println-based logging with a configurable logger and avoid sensitive data in logs. - Increase password hashing strength and add pepper. - Add rate-limiting and session/token hardening.