**Choose a programming language and application. Review the code for security vulnerabilities and provide recommendations for secure coding practices. Use tools like static code analyzers or manual code review.**

Code:

package com.example.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.\*;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.beans.factory.annotation.Autowired;

public class DemoApplication {

private JdbcTemplate jdbcTemplate;

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

public class DemoController {

public String home() {

return "Welcome to the Home Page";

}

public String search(@RequestParam String query) {

String sql = "SELECT \* FROM users WHERE name LIKE '%" + query + "%'";

return jdbcTemplate.queryForList(sql).toString();

}

public String submit(@RequestParam String name, @RequestParam String email) {

String sql = "INSERT INTO users (name, email) VALUES ('" + name + "', '" + email + "')";

jdbcTemplate.update(sql);

return "Submitted successfully";

}

}

}

CODE REVIEW:

1. **SQL Injection Vulnerability**:
   * The code concatenates user input directly into SQL queries, which is highly susceptible to SQL injection attacks.
2. **Cross-Site Scripting (XSS)**:
   * The code does not demonstrate output encoding or escaping, which can lead to XSS vulnerabilities when rendering user input.
3. **Sensitive Data Exposure**:
   * There's no explicit handling of sensitive data, but using debug logs to expose information or not securing data properly could lead to vulnerabilities.
4. **Potential Data Exposure**:
   * The application directly returns database query results as strings, which can potentially expose sensitive data.

**RECOMMENDATIONS FOR SECURE CODING PRACTICES**

**Use Prepared Statements**: To prevent SQL injection, use prepared statements instead of concatenating strings for SQL queries.

public String search(@RequestParam String query) {

String sql = "SELECT \* FROM users WHERE name LIKE ?";

return jdbcTemplate.queryForList(sql, "%" + query + "%").toString();

}

public String submit(@RequestParam String name, @RequestParam String email) {

String sql = "INSERT INTO users (name, email) VALUES (?, ?)";

jdbcTemplate.update(sql, name, email);

return "Submitted successfully";

}

**Sanitize and Encode User Input**: Ensure that user inputs are properly sanitized and encoded to mitigate XSS risks.

public String search(@RequestParam String query) {

String sanitizedQuery = HtmlUtils.htmlEscape(query);

String sql = "SELECT \* FROM users WHERE name LIKE ?";

return jdbcTemplate.queryForList(sql, "%" + sanitizedQuery + "%").toString();

}

**Proper Logging and Error Handling**: Avoid logging sensitive information and ensure proper error handling.

public String submit(@RequestParam String name, @RequestParam String email) {

try {

String sql = "INSERT INTO users (name, email) VALUES (?, ?)";

jdbcTemplate.update(sql, name, email);

return "Submitted successfully";

} catch (Exception e) {

// Log without sensitive data

logger.error("Database insertion error", e);

return "Submission failed";

}

}

**Implement Validation**: Ensure that user inputs are validated properly.

public String submit(@RequestParam String name, @RequestParam String email) {

if (!isValidName(name) || !isValidEmail(email)) {

return "Invalid input";

}

String sql = "INSERT INTO users (name, email) VALUES (?, ?)";

jdbcTemplate.update(sql, name, email);

return "Submitted successfully";

}

private boolean isValidName(String name) {

return name != null && name.matches("[A-Za-z ]+");

}

private boolean isValidEmail(String email) {

return email != null && email.matches("^[A-Za-z0-9+\_.-]+@(.+)$");

}

**SECURE CODING PRACTICES**

INPUT VALIDATION

* Conduct all input validation on a trusted system (server side not client side)
* Identify all data sources and classify them into trusted and untrusted
* Validate all data from untrusted sources (databases, file streams, etc)
* Use a centralized input validation routine for the whole application
* Encode input to a common character set before validating
* All validation failures should result in input rejection
* If the system supports UTF-8 extended character sets and validate after UTF-8 decoding is completed
* Validate all client provided data before processing
* Validate data range
* Validate data length

OUTPUT VALIDATION

* Conduct all output encoding on a trusted system (server side not client side)
* Utilize a standard, tested routine for each type of outbound encoding
* Specify character sets, such as UTF-8, for all outputs
* Contextually output encode all data returned to the client from untrusted sources
* Ensure the output encoding is safe for all target systems
* Contextually sanitize all output of un-trusted data to queries for SQL, XML, and LDAP
* Sanitize all output of untrusted data to operating system commands

AUTHETIFICATION AND PASSWORD MANAGEMENT

* Require authentication for all pages and resources, except those specifically intended to be public
* All authentication controls must be enforced on a trusted system
* Establish and utilize standard, tested, authentication services whenever possible
* Use a centralized implementation for all authentication controls, including libraries that call external authentication services
* Segregate authentication logic from the resource being requested and use redirection to and from the centralized authentication control
* All authentication controls should fail securely
* Temporary passwords and links should have a short expiration time
* Enforce the changing of temporary passwords on the next use
* Notify users when a password reset occurs
* Prevent password re-use
* Passwords should be at least one day old before they can be changed, to prevent attacks on password re-use

SESSION MANAGEMENT

* Use the server or framework’s session management controls. The application should recognize only these session identifiers as valid
* Session identifier creation must always be done on a trusted system (server side not client side)
* Session management controls should use well vetted algorithms that ensure sufficiently random session identifiers
* Set the domain and path for cookies containing authenticated session identifiers to an appropriately restricted value for the site
* Logout functionality should fully terminate the associated session or connection
* Logout functionality should be available from all pages protected by authorization
* Establish a session inactivity timeout that is as short as possible, based on balancing risk and business functional requirements

ACCESS CONTROL

* Restrict access to protected URLs to only authorized users
* Restrict access to protected functions to only authorized users
* Restrict direct object references to only authorized users
* Restrict access to services to only authorized users
* Restrict access to application data to only authorized users
* Restrict access to user and data attributes and policy information used by access controls
* Restrict access security-relevant configuration information to only authorized users
* Server side implementation and presentation layer representations of access control rules must match
* If state data must be stored on the client, use encryption and integrity checking on the server side to detect state tampering
* Enforce application logic flows to comply with business rules

CRYPTOGRAPHIC PRACTICES

* All cryptographic functions used to protect secrets from the application user must be implemented on a trusted system
* Protect secrets from unauthorized access
* Cryptographic modules should fail securely
* All random numbers, random file names, random GUIDs, and random strings should be generated using the cryptographic module’s approved random number generator
* Cryptographic modules used by the application should be compliant to FIPS 140-2 or an equivalent standard
* Establish and utilize a policy and process for how cryptographic keys will be managed

ERROR HANDLING AND LOGGING

* Use error handlers that do not display debugging or stack trace information
* Implement generic error messages and use custom error pages
* The application should handle application errors and not rely on the server configuration
* Properly free allocated memory when error conditions occur
* Error handling logic associated with security controls should deny access by default
* All logging controls should be implemented on a trusted system
* Logging controls should support both success and failure of specified security events
* Ensure logs contain important log event data
* Ensure log entries that include un-trusted data will not execute as code in the intended log viewing interface or software
* Restrict access to logs to only authorized individuals
* Utilize a central routine for all logging operations

SECURE CODE REVIEW CHECKLIST

* What security vulnerabilities is this code susceptible to?
* Are authorization and authentication handled in the right way?
* Is (user) input validated, sanitized, and escaped to prevent cross-site scripting or SQL injection?
* Is sensitive data like user data, or credit card information securely handled and stored?
* Does this code NOT reveal some secret information like keys, passwords, or usernames?
* Is data retrieved from external APIs or libraries checked accordingly?
* Does error handling or logging NOT expose the system to vulnerabilities?
* Is the right encryption used?