# Security Report for Fitness Application

## OWASP Top 10 Security Risks

### 1. Broken Access Control

* Mitigation:
  + Role-based access control (RBAC) is implemented to ensure users can only access data they are authorized for. Users have specific roles such as CUSTOMER, ADMIN, etc.
  + Access to API endpoints is restricted using annotations like @PreAuthorize in Spring Security.
* Backend Code Example:

@PreAuthorize("hasRole('CUSTOMER') and #id == authentication.principal.id")

@GetMapping("/user/{id}/workouts")

public List<WorkoutDTO> getUserWorkouts(@PathVariable Long id) {

return workoutService.getWorkoutsByUserId(id).stream()

.map(workoutMapper::toDto)

.collect(Collectors.toList());

}

* Why it works:
  + The @PreAuthorize annotation ensures the endpoint is accessible only to users with the CUSTOMER role and whose ID matches the authenticated user's ID.
  + This prevents CUSTOMER A from accessing or modifying data belonging to CUSTOMER B.

### 2. Injection

* Mitigation:
  + All SQL queries are executed using JPA and Hibernate, which leverage parameterized queries to prevent SQL injection.
  + Direct SQL statements in the code use prepared statements, which automatically escape inputs.
* Backend Code Example:

@Repository

public interface UserRepository extends JpaRepository<User, Long> {

@Query("SELECT u FROM User u WHERE u.username = :username")

Optional<User> findByUsername(@Param("username") String username);

}

* Why it works:
  + The @Query annotation with @Param ensures that user input is parameterized, preventing malicious SQL injection attempts.

### 3. Security Misconfiguration

* Mitigation:
  + All sensitive configurations such as database credentials and JWT secrets are stored in environment variables.
  + Headers like X-Content-Type-Options, Strict-Transport-Security, and X-Frame-Options are enforced via Nginx or Spring Security.

### 4. Cryptographic Failures

* Mitigation:
  + Passwords are hashed using BCrypt before storing them in the database.
  + SecureRandom is used for generating tokens.

### 5. Insecure Design

* Mitigation:
  + Security requirements are considered during design phases. Examples include role-based access and secure API endpoints.

### 6. Vulnerable and Outdated Components

* Mitigation:
  + Dependencies are managed via Maven, and tools like Dependabot are used to track and update outdated dependencies.

### 7. Identification and Authentication Failures

* Mitigation:
  + Spring Security's built-in authentication mechanisms are used.
  + JWT tokens are implemented for session management with proper expiration times and signature validation.

### 8. Software and Data Integrity Failures

* Mitigation:
  + Code integrity is enforced through a CI/CD pipeline, and SonarQube is used for static code analysis.

### 9. Security Logging and Monitoring Failures

* Mitigation:
  + Logs are generated for critical actions such as authentication and data access using tools like SLF4J and Logback.
  + Unusual activities trigger alerts via monitoring tools such as Prometheus and Grafana.

### 10. Server-Side Request Forgery (SSRF)

* Mitigation:
  + Outbound requests are restricted using firewalls, and only necessary domains are whitelisted.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OWASP Risk | Likelihood | Impact | Risk | Actions Possible | Planned |
| A1: Broken Access Control | 5 | 5 | 5 | Implement role-based access control (RBAC), validate user roles, and conduct regular testing. | Yes |
| A1: Cryptographic Failures | 2 | 5 | 3 | Ensure secure password hashing (e.g., bcrypt), enforce HTTPS, and validate secure token handling. | Yes |
| A3: Injection | 3 | 5 | 4 | Use prepared statements and parameterized queries in SQL and validate all inputs. | Yes |
| A4: Insecure Design | 2 | 4 | 3 | Review design patterns and ensure security practices are followed in the development phase. | Yes |
| A5: Security Misconfiguration | 5 | 3 | 4 | Harden server configurations and regularly scan for vulnerabilities. | Yes |
| A6: Vulnerable Components | 3 | 5 | 4 | Regularly update dependencies and use tools like Dependabot. | Yes |
| A7: Auth Failures | 2 | 5 | 3 | Implement multi-factor authentication (MFA) and enforce strong password policies. | Yes |
| A8: Integrity Failures | 2 | 4 | 3 | Use signed updates and validate package integrity for all deployments. | Yes |
| A9: Logging Failures | 5 | 3 | 4 | Implement centralized logging and monitoring systems with alerting. | Yes |
| A10: SSRF | 2 | 3 | 2 | Validate all user inputs and enforce strict firewall rules for external resource access. | Yes |

## Reasoning

* Broken Access Control: The application allows users with different roles (e.g., admin, trainers, and users). Without role-based validation, unauthorized access could lead to severe data breaches. Implementing RBAC mitigates this risk effectively.
* Cryptographic Failures: Since the application uses authentication tokens and handles sensitive user data, strong encryption and secure HTTPS communication are crucial. Weak encryption could expose sensitive information during transmission.
* Injection: SQL injection is a high risk for any application dealing with a database. Prepared statements and input validation reduce this risk significantly.
* Insecure Design: Following secure design patterns ensures potential vulnerabilities are addressed during development rather than post-deployment.
* Security Misconfiguration: Misconfigured servers and frameworks are an easy target. Regular configuration reviews and automated scans mitigate this risk.
* Vulnerable Components: The application depends on libraries and frameworks. Keeping these updated reduces exposure to known vulnerabilities.
* Authentication Failures: MFA ensures additional security against unauthorized access, especially for admin and trainer accounts.
* Data Integrity Failures: Signed updates ensure data integrity for both backend services and frontend updates.
* Logging Failures: Without proper logging, tracking security breaches becomes challenging. A centralized logging mechanism ensures actionable insights are available in real-time.
* SSRF: The application doesn’t directly deal with external resource fetching but validating user inputs and firewall rules minimize this risk.

## Conclusion

The fitness application is reasonably secure with existing measures such as HTTPS enforcement, prepared statements, and role-based access control. However, continuous security assessments are critical, especially as new features are added. Planned improvements include implementing centralized logging, multi-factor authentication, and regular dependency updates.

While the app currently addresses major OWASP risks, further enhancements like automated vulnerability scans and regular penetration testing will ensure long-term security compliance.