Vulnerability Fixes Report

COMP47910 Secure Software Engineering

Luis Marron (23202882)

A Lab Journal in part fulfilment of the degree of

MSc. in Advanced Software Engineering

Supervisor: Dr. Liliana Pasquale

UCD School of Computer Science
University College Dublin

August 18, 2025

Table of Contents

Chapter 1: **A01:2021 Broken Access Control**

1.1 CWE-284: Improper Access Control

Description: The BookShop application implements improper access control mechanisms that allow unauthorized users to access restricted functionality and sensitive data. The application lacks proper authentication and authorization checks across multiple endpoints, enabling attackers to bypass security controls and gain unauthorized access to administrative functions and user-specific resources.

CWE Explanation: CWE-284 occurs when the application fails to properly restrict access to functionality or resources, allowing unauthorized users to perform actions or access data that should be protected by proper authentication and authorization mechanisms.

Severity: High

1.1.1 Vulnerability Locations and Type

The application suffered from multiple *Broken / Improper Access Control* weaknesses manifesting as:

- **Vertical privilege escalation**: Security rule protected path pattern `/admin/**`, while the actual admin endpoint controller was mapped to `/admins`. This mismatch let any authenticated non-admin invoke administrative endpoints (e.g. `GET /admins`).
- **Horizontal privilege escalation / Insecure Direct Object Reference (IDOR)**: Cart- and cart-item–related REST endpoints (`/carts/**`, `/cart-items/**`) accepted arbitrary `customerId`, `cartId`, or `itemId` without verifying ownership against the authenticated principal, enabling access/modification of other users' carts.
- **Over broad data exposure**: User and customer enumeration endpoints (`/users/**`, `/customers/**`) allowed any authenticated user to list or fetch other users/customers and (in the case of `/users`) returned password hashes, breaching least privilege and confidentiality.
- **Unrestricted modification endpoints**: Book management API (`books POST PUT DELETE`) was accessible to any authenticated role, permitting non-admin content manipulation.
- **Lack of server side authorization on web cart actions**: Web MVC endpoints for cart item removal did not assert ownership, allowing crafted requests to remove other users' items.

1.1.2 Mitigation Strategy and Rationale

Defence-in-depth was applied combining **path-based authorization**, **method-level role enforcement**, and **resource ownership (row-level) checks**. This layered approach ensures:

- Misconfigurations in one layer (e.g. path patterns) do not automatically grant access because method-level annotations add a second gate.
- Even with correct role checks, horizontal attacks "guessing IDs" are blocked by explicit ownership validation tying domain object identifiers to the authenticated principal "prevents IDOR".
- Principle of Least Privilege is restored by scoping sensitive endpoints (admin, user lists, data mutations) strictly to the required roles, reducing attack surface.
- Sensitive credential fields are no longer exposed after authorization succeeds, limiting post-auth data leakage channels.

1.1.3 Implemented Security Controls

The following concrete changes were introduced in code (branch `broken-access-control-fixes`):

1. **Corrected admin path mapping**: Updated Spring Security configuration to match the actual controller path `/admins/**` instead of the incorrect `/admin/**`.
2. **Granular authorization rules**: Added HTTP method specific matchers restricting book mutation endpoints (POST/PUT/DELETE on `/api/books/**`) and all `/books/**` (management pages) to role `ADMIN`. User endpoints `/users/**` now require `ADMIN`; customer endpoints enforce authentication plus ownership (see point 3) and reserve modifications for `ADMIN`.
3. **Ownership / row-level checks**: Injected `@AuthenticationPrincipal` into cart and cart item controllers; added helper methods validating that the referenced cart/item's owning customer's username equals the authenticated principal. Requests failing validation now return HTTP 403 (or 404 when appropriate).
4. **Method-level security**: Enabled `@EnableMethodSecurity` and introduced `@PreAuthorize` annotations on controller methods (e.g. book mutations, user controller) to provide a second authorization layer beyond URL pattern matching.
5. **Data minimisation via DTOs**: Replaced direct entity exposure for users and customers with DTOs omitting the password field and limiting attributes to non-sensitive identification data.
6. **Cart web endpoint hardening**: Added ownership validation before allowing removal of a cart item via web MVC endpoint to block cross-user manipulation.
7. **Input encapsulation**: Refactored mutable request payload classes (e.g. add-item request) to use private fields with accessors, preventing accidental uncontrolled field exposure (minor hardening).

1.1.4 Solution Effectiveness

The mitigations directly address the CWE-284 root causes:

- Path correction + role restrictions close vertical privilege escalation vectors.
- Ownership validation eliminates horizontal ID enumeration attacks by binding operations to the authenticated identity.

- Method-level annotations safeguard against future path mapping drift or overly broad ant matchers.
- DTO-based redaction removes unnecessary sensitive data from responses, shrinking impact radius of any residual access gaps.
- Principle of Least Privilege is enforced consistently across both REST and MVC layers, aligning actual access with business intent.

1.2 CWE-639: Insecure Direct Object References

Description: Multiple endpoints previously trusted user-supplied identifiers (e.g. `customerId`, `cartId`, `itemId`) directly to select resources without verifying that the authenticated principal owned or was entitled to those resources. Attackers could substitute another valid numeric ID (IDOR) to read or modify other users' shopping cart contents or personal data.

CWE Explanation: CWE-639 occurs when an application authorizes a request based solely on a user-controlled key (such as a record ID) instead of confirming the requester is permitted to access the referenced object, enabling horizontal privilege escalation (Insecure Direct Object Reference).

Severity: High

1.2.1 Vulnerability Locations and Type

- **Cart REST endpoints**: `/carts/by-customer/customerId`, `/carts/cartId/items`, `/carts/cartId/a` `/carts/cartId/remove-item/itemId`, and `/carts/cartId/total-price` accepted arbitrary path IDs; prior to fixes there was no binding between these IDs and the authenticated user (classic IDOR / horizontal escalation).
- **Cart item endpoint**: `/cart-items/id` exposed individual cart items without verifying ownership (permitting enumeration of other users' items by ID).
- **Customer lookup endpoints**: `/customers/id` and `/customers/by-username/username` allowed retrieval of other customers' personal information without ownership check (in early state) relying only on being authenticated.

1.2.2 Mitigation Strategy and Rationale

The strategy focused on eliminating trust in client-supplied identifiers by: (1) performing **server-side ownership validation** after object retrieval, (2) layering **role checks** so only admins can enumerate accounts, and (3) reducing exposed data via DTOs. Ownership checks ensure an attacker who guesses an ID still receives `403 Forbidden` (or `404 Not Found`) unless authorized. This halts horizontal privilege escalation while preserving legitimate functionality for rightful owners.

1.2.3 Implemented Security Controls

1. **Ownership helpers**: Introduced methods `enforceCustomerOwnership(...)` and `resolvedOwnedCart(` in the cart controller to load the resource and verify `resource.owner.username == principal.username` before returning it.
2. **Cart item deletion hardening**: Added explicit retrieval and cart-to-item linkage verification before deleting an item, ensuring the item belongs to the authenticated user's cart, preventing cross-cart deletions.
3. **Principal injection**: Added `@AuthenticationPrincipal` parameters to affected controller methods to reliably access the authenticated identity rather than relying on user-provided IDs.
4. **Customer data protection**: Added central ownership/admin check (`enforceOwnershipOrAdmin`) to customer endpoints and restricted full listing to `ADMIN` via `@PreAuthorize`.
5. **Least privilege for enumeration**: Restricted `/users/**` and customer listing endpoints to `ADMIN` in `SecurityConfig`, removing the ability for regular users to guess IDs and enumerate.
6. **DTO redaction**: Replaced direct entity exposure with DTOs omitting password hashes, limiting the value of any accidental disclosure.
7. **Path-based + method security**: Retained path restrictions (role-based) and enabled method-level security, adding a secondary barrier should future path patterns broaden inadvertently.

1.2.4 Solution Effectiveness

- Ownership enforcement turns previously unauthenticated resource selection into a two-factor authorization (valid ID + rightful owner) neutralising ID guessing.
- Removal operations now validate item-to-cart ownership, closing a residual destructive IDOR vector.
- Admin-only enumeration eliminates bulk discovery of valid identifiers (reducing reconnaissance surface).
- DTO redaction ensures that even if an ownership check regresses, sensitive credential data remains undisclosed.
- Layered (config + method) checks reduce single-point-of-failure risk if URL matcher misconfiguration reappears.

1.3 CWE-384: Session Fixation

Description: Initially the session fixation risk was assessed because the configuration did not explicitly state a session fixation protection strategy, and CSRF protection was disabled (increasing the impact of any session hijacking). However, Spring Security's default behaviour already migrates (rotates) the session identifier upon successful authentication. We have now made this

protection explicit in the security configuration, confirming that an attacker who pre-seeds a victim with a known anonymous session ID cannot continue to use that same ID after the victim authenticates.

CWE Explanation: CWE-384 (Session Fixation) concerns failure to invalidate or regenerate a session identifier when a user's authentication state changes, enabling an attacker who knows the pre-auth session token to hijack the post-auth session.

Severity: Low (Residual) – Effective mitigations in place; only hardening items remain.

1.3.1 Vulnerability Locations and Type

Prior to the explicit mitigation the potential exposure was theoretical rather than an observed exploit, characterised by:

- No explicit session fixation directive in `SecurityConfig` (relying implicitly on framework defaults).
- Disabled CSRF protection (would amplify damage of any successful fixation / hijack).
- Absence of cookie security attribute configuration (Secure / SameSite / HttpOnly not yet declared in `application.properties`).

Crucially, there was *no* custom code overriding Spring Security's default fixation protection; thus the core exploit path (reusing the same session ID after login) was already blocked.

1.3.2 Mitigation Strategy and Rationale

Strategy focused on: (1) making implicit framework protections *explicit* for auditability, (2) reducing residual attack surface through planned cookie hardening and CSRF re-enablement, and (3) documenting session handling to prevent future regressions (e.g. switching to a stateless policy without compensating controls). Explicit configuration eliminates uncertainty for reviewers and compliance checks.

1.3.3 Implemented Security Controls

1. **Explicit session ID migration**: Added `sessionFixation(migrateSession())` under `sessionManagement` in `SecurityConfig`, guaranteeing a new session identifier post-authentication.
2. **Logout invalidation**: Existing logout configuration invalidates the session server-side, preventing reuse of an authenticated context.
3. **Least privilege hardening elsewhere**: Reduced horizontal/vertical escalation (CWE-284 / CWE-639 controls) lowers the value of any hypothetical hijacked session.
4. **Password hashing (BCrypt)**: Limits credential replay even if a session were short-livedly exposed.

1.3.4 Solution Effectiveness

- Attacker-controlled pre-auth session IDs are invalidated because the server issues a fresh session ID upon authentication (session fixation vector neutralised).
- Logout and privilege boundaries ensure compromised sessions cannot silently escalate or persist indefinitely.
- Strengthened access controls reduce the actionable scope of any transient session misuse.
- Making the protection explicit aids code reviews and prevents accidental regression (e.g. future refactor removing defaults unnoticed).

Given the default and now explicit migration behaviour, the application is not meaningfully vulnerable to session fixation at present; residual items are advisory enhancements.

Chapter 2: **A02:2021 Cryptographic Failures**

2.1 CWE-311: Missing Encryption of Sensitive Data

Description: Certain sensitive data flows in the BookShop application lack enforced cryptographic protection in transit or rely on plaintext representation in local configuration. While user passwords are securely hashed (BCrypt) and the application–database channel is configured for SSL, end-user credential submission (login/registration), session cookies, and ad hoc sensitive form fields (e.g. checkout credit card input) can traverse unencrypted HTTP if the deployment is not fronted by TLS. Additionally, example environment variables and scripts illustrate plaintext storage of database and truststore secrets. These gaps collectively represent incomplete protection of sensitive data rather than a single critical exposure.

CWE Explanation: CWE-311 refers to missing or absent encryption for sensitive information either at rest or in transit where encryption is expected for confidentiality or regulatory compliance. It commonly manifests as cleartext transmission (overlaps with CWE-319) or cleartext storage (overlaps with CWE-312 / CWE-256) when mitigations are inconsistent or absent.

Severity: Medium (Residual) – Core credentials (passwords at rest) are protected; remaining issues affect transport and secret handling hygiene.

2.1.1 Vulnerability Locations and Type

- **Transport (HTTP)**: Application lacks explicit HTTPS / TLS configuration (no `server.ssl.*`); README usage examples employ `http://localhost:8080`. In any non-local deployment without a reverse proxy TLS terminator, credentials and session cookies would be transmitted in cleartext (CWE-319 contributing to CWE-311).
- **Session Cookies**: No Secure, HttpOnly, or SameSite attributes configured; cookies could be exposed over insecure transport or be susceptible to CSRF replay.
- **Checkout form data**: Credit card number field (not persisted) is still transmitted; without TLS it is exposed in transit. No masking or client-side obfuscation.
- **Secrets in environment artifacts**: Example `.env` content and setup scripts contain plaintext database credentials and truststore password (acceptable for local dev, but a risk pattern if replicated to production). Overlaps with CWE-256.
- **Logging / auditing**: No explicit masking safeguards (currently no evidence of leakage, but absence of policy introduces latent risk if future logging is added).

2.1.2 Mitigation Strategy and Rationale

Strategy targets layered confidentiality protection: (1) enforce TLS for all external HTTP traffic; (2) harden session cookie attributes to reduce exposure and replay potential; (3) minimise plaintext secret footprint by moving production secrets to a managed vault; (4) restrict network attack

surface via HSTS and secure redirects; (5) avoid collecting unnecessary high sensitivity data (drop credit card field if not processing payments). This layered defence ensures compromise of any single mechanism (e.g. proxy misconfiguration) does not fully expose sensitive data.

2.1.3 Implemented / Planned Security Controls

1. **Password hashing (in place)**: All user credentials stored with BCrypt (adaptive hashing, salts implicit) – removes cleartext password storage risk.
2. **Encrypted DB channel (in place)**: JDBC URL enforces `useSSL=true` and `requireSSL=true` with truststore configuration.
3. **Session fixation mitigation (in place)**: Protects session integrity post-auth (limits utility of intercepted pre-auth IDs).
4. **Planned TLS enforcement**: Introduce reverse proxy or Spring Boot keystore; redirect HTTP $\rightarrow$ HTTPS and add HSTS header.
5. **Cookie hardening (planned)**: Set `server.servlet.session.cookie.secure=true`, `...http-only=true`, `...same-site=Strict`.
6. **Secret management (planned)**: Replace plaintext env variables (especially root DB password) with secrets provisioned by a secure store (Vault / cloud KMS) and principle of least privilege DB accounts.
7. **Data minimisation (planned)**: Remove or tokenise credit card field; if retained for demo, clearly flag as non-production and mask client-side.
8. **Logging safeguards (planned)**: Introduce logging policy and filters to prevent future accidental sensitive data logging.

2.1.4 Solution Effectiveness

- Existing strong password hashing and encrypted DB transport already neutralise the most damaging storage and backend transit risks.
- Planned universal TLS and Secure/HttpOnly/SameSite cookies will close interception windows for credentials and session tokens.
- Secret manager adoption will reduce blast radius of host compromise or repo leakage versus static env files.
- Removing unnecessary high sensitivity fields (credit card) eradicates an entire sensitive data class, lowering compliance scope (PCI DSS) and exposure.
- Layered controls ensure that even if one layer (e.g. proxy TLS) fails, credential hashes remain non-reversible and session tokens protected by cookie policies.

2.2 CWE-315: Cleartext Storage of Sensitive Information in a Cookie

Description: Analysis confirms no sensitive information (plaintext passwords, API keys, tokens containing secrets, PII, payment data) is stored in browser cookies. The application relies exclusively on the standard opaque `JSESSIONID` for server-side session tracking. No custom cookie creation exists in server code or client-side JavaScript; thus the application is *not currently vulnerable* to CWE-315.

CWE Explanation: CWE-315 covers placing sensitive information directly (cleartext or trivially encoded) into cookies such that disclosure or tampering enables account compromise, impersonation, or data leakage. Even with HTTPS, client-side storage of secrets is discouraged because local access or XSS can expose them.

Severity: None (Not Vulnerable). Would elevate to High if future changes store secrets (e.g. raw bearer tokens) in cookies without encryption/integrity safeguards.

Assessment Evidence

- No occurrences of custom `Cookie` creation or header manipulation in Java sources.
- No `document.cookie` usage in static JS or templates.
- DTOs exclude password hashes; no credential material leaves server to be placed in cookies.
- Session identifier is opaque; no sensitive claims or data embedded.

Mitigation Strategy and Rationale

Maintain a minimal opaque session token model: keep all sensitive state server-side; never persist secrets or PII in client cookies. Complement with cookie attribute hardening (Secure, HttpOnly, SameSite) and session rotation (already explicit) to reduce ancillary risks (CWE-614, CWE-1275) while ensuring no CWE-315 condition can emerge inadvertently.

Implemented / Planned Controls

1. **Opaque server session (implemented)**: Only an identifier stored client-side.
2. **Password hashing (implemented)**: Removes any need to surface plaintext credentials.
3. **No custom cookie writes (implemented)**: Eliminates common accidental leakage vector.
4. **Cookie attribute hardening (planned)**: Add Secure / HttpOnly / SameSite (ties into CWE-319 remediation).
5. **Future guardrails (planned)**: CI/static rule to flag introduction of sensitive keywords in cookie names or values.

Solution Effectiveness

- Absence of data-bearing cookies removes primary CWE-315 attack surface.
- Planned attributes further reduce residual token theft or cross-site replay vectors.
- Guardrails lower regression risk if new auth features (remember-me, JWT) are introduced.

2.3 CWE-319: Cleartext Transmission of Sensitive Information

Description: Client-server interactions (login, registration, authenticated browsing) were originally performed over HTTP with no transport security enforcement, as neither embedded TLS (no `server.ssl.*` properties) nor an HTTPS redirect existed. Documentation curl examples referenced `http://localhost:8080`, normalising plaintext usage. In any non-local / shared network scenario this exposes credentials and session identifiers to interception or manipulation (Man-in-the-Middle, passive sniffing). Session cookies also lacked Secure / SameSite attributes, increasing replay and cross-site leakage risk.

CWE Explanation: CWE-319 addresses transmission of sensitive information over cleartext channels. Attackers positioned on the network path can read or alter traffic lacking cryptographic protection. This differs from CWE-311 (broader missing encryption) by focusing specifically on in-transit confidentiality/integrity compromise enabling credential theft, session hijacking, or data tampering.

Severity: High (production / shared networks). Medium (loopback-only development) but still a negative security practice.

Vulnerability Locations and Type

- **No TLS configuration**: Absence of `server.ssl.key-store` or reverse proxy mandate meant HTTP only.
- **No enforcement**: Security configuration had no channel security requirement or redirect to HTTPS.
- **Session cookies**: Lacked Secure, HttpOnly, SameSite flags (susceptible to interception / CSRF replay once over HTTP).
- **Documentation examples**: README exclusively used plaintext HTTP endpoints, encouraging insecure operational patterns.
- **Potential sensitive forms**: Payment / PII form submissions (if added) would likewise traverse HTTP.

Mitigation Strategy and Rationale

Enforce an *HTTPS by default* posture: (1) introduce production-profile HTTPS enforcement and HSTS to prevent downgrade, (2) configure TLS via keystore or proxy, (3) harden cookies (Secure,

HttpOnly, SameSite) reducing token exfiltration surface, (4) eliminate plaintext examples to shift developer behaviour, (5) add automated tests verifying redirect + security headers to prevent regression. Layered approach ensures that if a single control fails (e.g. misconfigured proxy), other safeguards (cookie flags, credential hashing) limit exploit value.

Implemented / Planned Security Controls

1. **Prod profile HTTPS enforcement (implemented)**: Added custom redirect filter + HSTS headers when `prod` profile active.
2. **HSTS (implemented prod)**: Sets long max-age with subdomain + preload intent to resist protocol downgrades.
3. **Password hashing (existing)**: Mitigates offline cracking impact if credentials were previously observed.
4. **Cookie hardening (planned)**: Add `server.servlet.session.cookie.secure=true`, `...http-only=true`, `...same-site=Strict`.
5. **TLS keystore / proxy integration (planned)**: Provide keystore properties or documented reverse proxy termination (NGINX / Caddy with ACME).
6. **Documentation update (planned)**: Replace HTTP examples with HTTPS and describe trusting dev self-signed cert.
7. **Automated tests (planned)**: Integration test asserting HTTP -¿ HTTPS 301/308 redirect and presence of HSTS in prod.
8. **Mixed content audit (planned)**: Validate no HTTP asset references remain post-transition.

Solution Effectiveness

- HTTPS enforcement + HSTS removes primary interception vector (passive sniffing / trivial MiTM downgrade).
- Cookie hardening will prevent session token leakage over inadvertent HTTP calls and reduce CSRF token reuse potential.
- Removing plaintext examples reduces operational drift toward insecure defaults.
- Automated testing creates a guardrail against accidental removal of channel security.
- Existing password hashing ensures historical captures have limited utility if obtained pre-mitigation.

2.4 CWE-256: Unprotected Storage of Credentials

Description: Core user credentials (passwords) are securely stored using BCrypt hashing; however, residual unprotected credential storage issues remain: (1) a default administrator password is disclosed in cleartext inside the seeding script comments, (2) database and truststore secrets are managed as plaintext environment variables in a local `.env` file and referenced directly by

docker compose, and (3) weak placeholder values are demonstrated in README examples. These patterns risk accidental promotion of development secrets or reuse of weak defaults in production. The issue is thus not plaintext password storage in the database, but exposure and handling of operational secrets and a hard-coded default credential pattern.

CWE Explanation: CWE-256 covers storing credentials without adequate protection (encryption, hashing, vault-based segregation) or exposing them in source artifacts (scripts, config files, comments) such that compromise of the repository or workstation yields immediate credential disclosure. It overlaps with CWE-798 (hard-coded credentials) when defaults are embedded, and with CWE-522 when credential protection mechanisms are insufficient.

Severity: Medium (would escalate toward High if the disclosed admin password or example secrets were deployed unchanged beyond local development).

Vulnerability Locations and Type

- **Seed script comment**: `scripts/create-admin-user.sql` contains a cleartext default admin password in a comment (hard-coded / discoverable credential pattern).
- **Environment variables**: `.env` (excluded from VCS, but encouraged) stores DB root/user passwords and truststore password in plaintext. Acceptable locally, but no segregation or rotation policy specified for higher environments.
- **Truststore password exposure**: Passed directly via `SPRING_DATASOURCE_HIKARI_TRUSTSTORE_PASSWORD` environment variable; resident in process environment and potentially logs if misconfigured.
- **Placeholder weak values**: README examples (`rootpassword`, `appuserpassword`, `truststorepasswor` risk being reused verbatim.
- **No secret scanning / policy**: CI/CD pipeline (not documented) lacks automated detection to block accidental secret commits.

Mitigation Strategy and Rationale

Focus on eliminating static/discoverable secrets in source and strengthening operational handling: (1) remove or parameterise default credentials; (2) introduce secret generation at setup time with one-time display; (3) adopt a secrets manager for non-local deployments; (4) enforce least-privilege DB accounts (separate admin vs app user); (5) integrate automated secret scanning to prevent regressions; (6) clearly label placeholder values as 'CHANGEME' to discourage reuse. This reduces credential exposure surface and shortens compromise window via rotation and principle of least privilege.

Implemented / Planned Security Controls

1. **Password hashing (implemented)**: All end-user passwords stored using BCrypt (adaptive, salted) eliminating plaintext password-at-rest risk.
2. **DTO redaction (implemented)**: Password hashes no longer returned by any API endpoints, reducing secondary leakage channels.
3. **Encrypted DB transport (implemented)**: Mitigates interception of database credentials post-auth handshake.

4. **Remove default admin secret (planned)**: Replace hard-coded admin credential comment with instruction to supply a strong password (or script-driven random generation).
5. **Secret generation (planned)**: Modify setup scripts to auto-generate strong random passwords (e.g. 24+ char base64) and print once.
6. **Secrets manager integration (planned)**: Externalise production secrets (Vault / cloud provider store) removing reliance on persistent '.env' in production.
7. **Least privilege DB accounts (planned)**: Separate schema migration/admin from runtime application user; restrict privileges to required CRUD.
8. **Secret scanning in CI (planned)**: Add tools (e.g. Gitleaks / TruffleHog) to detect accidental commits of secrets or weak defaults.
9. **Documentation hardening (planned)**: Mark all sample credentials as `CHANGEME_<PURPOSE>` to prevent reuse in non-local deployments.

Solution Effectiveness

- Existing hashing already neutralises the most severe impact vector (direct theft of user passwords from DB).
- Removing the disclosed admin password and auto-generating credentials eliminates a trivial, high-impact compromise path.
- Secrets manager + least privilege narrows lateral movement after host compromise: stolen app credential yields only scoped DB access.
- CI secret scanning provides early detection and prevents regression (shift-left control).
- Clear CHANGEME placeholders reduce likelihood of weak default propagation to production.

Chapter 3: **A03:2021 Injection**

3.1 CWE-89: SQL Injection

Description: Assessment of the BookShop application's data-access layer found no exploitable SQL Injection vectors in its current state. All persistence operations are performed through Spring Data JPA repository interfaces using method-name derived queries (e.g. `findByUsername`) or inherited CRUD methods. There is no evidence of string concatenation to build JPQL/SQL, no usage of `@Query` with interpolated parameters, and no raw JDBC / native query execution paths. Consequently, untrusted user input is never directly merged into executable SQL without parameter binding.

CWE Explanation: CWE-89 (SQL Injection) arises when an application constructs SQL statements by directly embedding untrusted input, allowing attackers to alter query structure (e.g. changing WHERE clauses, UNION extraction, or triggering stacked queries in permissive drivers). Proper parameterisation ensures user data is treated strictly as values, not executable syntax.

Severity: None (Not Vulnerable) at present. Would elevate rapidly to High if future features introduce unparameterised dynamic queries.

Assessment Evidence

Audit activities and findings:

- Grep searches for raw SQL construction indicators returned no matches: patterns `createNativeQuery`, `@Query`, `PreparedStatement`, `Statement`, direct SQL keywords (SELECT/INSERT/UPDATE/DELETE) absent from code except in comments.
- No repository methods annotated with `@Query`; only Spring Data derived query methods (e.g. `findByUsername`) which use prepared parameter binding under the hood.
- No usage of `EntityManager`, `JdbcTemplate`, Criteria API, or manual pagination/sorting parameters that could introduce unsafe field injection.
- No dynamic ORDER BY / filtering logic sourced from request parameters (no `Sort` / `Pageable` occurrences) that could later encourage string concatenation.
- Domain entities contain only mapped fields; user-supplied values become parameters in prepared statements generated by the JPA provider (Hibernate), not concatenated SQL fragments.

Mitigation Strategy and Rationale

Maintain a *parameterisation-only* data access pattern and introduce guardrails so future changes cannot silently introduce injection risk. Emphasise: (1) exclusive use of Spring Data method derivation or `@Query` with named / positional parameters, (2) strict white-listing of sortable /

filterable fields when adding dynamic search endpoints, and (3) code review / CI rules to flag raw SQL introduction.

Implemented / Planned Controls

1. **Spring Data repositories (implemented)**: Auto-generated prepared statements ensure parameter binding.
2. **No raw SQL / native queries (implemented)**: Eliminates primary injection surface.
3. **Input validation (partial)**: Bean validation (@NotBlank) constrains some inputs; while not a primary defence for injection, it reduces anomalous payload shapes.
4. **Repository pattern adherence (planned guardrail)**: Architectural decision record (ADR) to codify prohibition of ad hoc JDBC unless justified with security review.
5. **Static scan rule (planned)**: CI grep / SAST rule to flag introduction of `@Query("%" + var)` style concatenations or `createNativeQuery`.
6. **Dynamic sort/filter sanitisation (planned)**: If future endpoints accept field names, implement white-list mapping (enum -¿ column) to avoid direct trust in request parameters.
7. **Security review checklist (planned)**: Mandatory review item for any PR adding raw SQL, native queries, or criteria builder logic.

Solution Effectiveness

- Absence of raw or annotated custom queries removes typical injection entry points.
- Hibernate's prepared statement generation enforces separation of code and data, neutralising payloads containing quotes or control tokens.
- Planned guardrails reduce regression risk as feature scope expands (search, reporting, analytics).
- White-list driven future dynamic sorting/filtering prevents second-order injection through column / direction parameters.

Current posture: *Not Vulnerable*. Focus shifts to prevention of unsafe patterns during future feature growth.

3.2 CWE-79: Cross-Site Scripting (XSS)

Description: The BookShop application was analyzed for Cross-Site Scripting (XSS) vulnerabilities due to improper handling of user-controlled data in error messages and direct rendering of user input without proper sanitization or encoding.

CWE Explanation: CWE-79 occurs when the application fails to properly validate, sanitize, or encode user-controlled input before including it in output that is sent to other users' browsers, allowing attackers to execute malicious scripts in the context of other users' sessions.

Severity: Low

3.3 CWE-190: Integer Overflow or Wraparound

Description: The BookShop application is vulnerable to integer overflow and underflow attacks due to improper handling of numeric operations without proper validation or overflow checks. This vulnerability allows attackers to manipulate business logic by providing malicious numeric values that cause integer wraparound.

CWE Explanation: CWE-190 occurs when the application performs arithmetic operations on integers without checking for overflow or underflow conditions, allowing attackers to manipulate numeric values to cause unexpected behavior, data corruption, or bypass business logic controls.

Severity: Medium

Chapter 4: **A04:2021 Insecure Design**

4.1 CWE-602: Client-Side Enforcement of Server-Side Security

Description: The BookShop application implements critical security controls on the client-side instead of the server-side, allowing attackers to bypass authentication, authorization, and input validation by making direct API calls to backend endpoints. The application relies on client-side JavaScript to enforce security policies that should be implemented on the server.

CWE Explanation: CWE-602 occurs when the application implements security controls (authentication, authorization, input validation) on the client-side rather than the server-side, making them easily bypassable by attackers who can make direct HTTP requests to backend endpoints.

Severity: High

4.2 CWE-799: Improper Control of Interaction Frequency

Description: The BookShop application lacks any rate limiting or frequency control mechanisms, allowing unlimited interaction attempts with all endpoints. The application does not implement authentication attempt limits, request throttling, or any protection against brute force attacks, making it vulnerable to automated attacks and resource exhaustion.

CWE Explanation: CWE-799 occurs when the application fails to properly control the frequency of interactions, allowing attackers to make unlimited requests that can lead to brute force attacks, resource exhaustion, and denial of service conditions.

Severity: High

4.3 CWE-840: Business Logic Errors

Description: The BookShop application contains multiple business logic errors that violate fundamental application rules and constraints. These include race conditions in the checkout process, lack of duplicate username validation, client-side price calculation vulnerabilities, and missing order validation rules that can lead to inventory overselling, account confusion, and financial manipulation.

CWE Explanation: CWE-840 occurs when the application fails to properly implement business

rules and constraints, allowing attackers to exploit logical flaws in the application's workflow, data validation, and state management to achieve unauthorized outcomes.

Severity: High

4.4 CWE-1173: Improper Use of Validation Framework

Description: The BookShop application completely lacks proper validation framework usage, with no Bean Validation annotations, no @Valid annotations in controllers, and no validation framework dependencies. The application relies solely on minimal client-side validation, allowing malicious or invalid input to bypass security controls and potentially cause data integrity issues, application instability, and security vulnerabilities.

CWE Explanation: CWE-1173 occurs when the application fails to properly use validation frameworks or implements validation incorrectly, allowing invalid or malicious input to bypass security controls, potentially leading to data integrity issues, application instability, and security vulnerabilities.

Severity: High

Chapter 5: **A05:2021 Security Misconfiguration**

5.1 CWE-614: Sensitive Cookie in HTTPS Session Without 'Secure' Attribute

Description: The BookShop application's session cookies (JSESSIONID) are transmitted without the 'Secure' attribute, allowing them to be sent over unencrypted HTTP connections. This vulnerability exposes session tokens to potential interception by attackers through network sniffing, man-in-the-middle attacks, or other network-based attacks.

CWE Explanation: CWE-614 occurs when sensitive cookies are transmitted over insecure channels without proper protection mechanisms, allowing attackers to capture and reuse session tokens to impersonate authenticated users.

Severity: High

5.2 CWE-1275: Sensitive Cookie with Improper SameSite Attribute

Description: The BookShop application's session cookies lack proper SameSite attribute configuration, allowing them to be sent in cross-site requests and enabling various client-side attacks including CSRF.

CWE Explanation: CWE-1275 occurs when cookies are configured without appropriate SameSite restrictions, allowing them to be sent in cross-site requests and enabling various client-side attacks including CSRF.

Severity: High

5.3 CWE-693: Protection Mechanism Failure

Description: The BookShop application includes Spring Security framework but completely disables all protection mechanisms, rendering the security framework ineffective. The application disables CSRF protection, permits all requests without authentication, and fails to implement any of the available security controls, making the protection mechanism completely non-functional.

CWE Explanation: CWE-693 occurs when the application has a protection mechanism in place but fails to use it properly, rendering the security controls ineffective and leaving the application vulnerable to attacks that the protection mechanism was designed to prevent.

Severity: High

Chapter 6: **A06:2021 Vulnerable and Outdated Components**

6.1 CWE-269: Improper Privilege Management

Description: The BookShop application exhibits multiple critical privilege management failures including missing authentication for critical admin functions, inconsistent privilege enforcement across endpoints, complete bypass of Spring Security framework, and lack of ownership verification for user resources. These vulnerabilities enable unauthorized access, privilege escalation, and cross-user data manipulation.

CWE Explanation: CWE-269 occurs when the application fails to properly manage privileges, permissions, and access controls, allowing unauthorized users to access restricted functionality or resources that should be protected by proper authentication and authorization mechanisms.

Severity: Critical

6.2 CWE-400: Uncontrolled Resource Consumption

Description: The BookShop application lacks proper resource management controls including database connection pool limits, session timeout limits, request timeout limits, and memory limits. This vulnerability allows attackers to exhaust system resources through unlimited requests, session creation, and large payload attacks, potentially leading to denial of service conditions.

CWE Explanation: CWE-400 occurs when the application fails to properly control resource consumption, allowing attackers to exhaust system resources such as memory, CPU, database connections, or network bandwidth through unlimited or uncontrolled operations, leading to denial of service conditions.

Severity: High

Chapter 7: **A07:2021 Identification and Authentication Failures**

7.1 CWE-287: Improper Authentication

Description: The BookShop application implements fundamentally flawed authentication mechanisms including plain text password storage, direct password comparison without hashing, weak password policies, and complete absence of authentication security controls. The application stores user credentials in plain text in the database and performs direct string comparison during login, making it vulnerable to complete account compromise and unauthorized access.

CWE Explanation: CWE-287 occurs when the application fails to properly verify the identity of users, implement secure authentication mechanisms, or protect authentication credentials, allowing attackers to bypass authentication controls, compromise user accounts, and gain unauthorized access to sensitive data and functionality.

Severity: Critical

Chapter 8: **Conclusions**

The BookShop application exhibits multiple critical **OWASP Top 10** vulnerabilities including **A01: Broken Access Control** (CWE-306, CWE-639, CWE-264), **A02: Cryptographic Failures** (CWE-326, CWE-522), **A03: Injection** (CWE-79 - mitigated by React), **A04: Insecure Design** (CWE-602, CWE-799, CWE-840, CWE-1173), **A05: Security Misconfiguration** (CWE-614, CWE-1275, CWE-693, CWE-256), **A06: Vulnerable and Outdated Components** (CWE-269, CWE-400), **A07: Identification and Authentication Failures** (CWE-287, CWE-384, CWE-521), **A07: Software and Data Integrity Failures** (CWE-798), and **A09: Security Logging and Monitoring Failures** (CWE-209). These vulnerabilities enable session hijacking, session fixation attacks, CSRF attacks, unauthorized data access, information disclosure, credential compromise, insufficient credential protection, weak password requirements, client-side security bypass, brute force attacks, resource exhaustion, business logic errors, race conditions, inventory overselling, financial manipulation, improper privilege management, authentication bypass, privilege escalation, uncontrolled resource consumption, denial of service, validation framework bypass, data integrity issues, improper authentication, plain text password storage, hard-coded credentials, and command injection. The application requires comprehensive security remediation including proper input validation, CSP headers, secure session management, session fixation protection, robust authentication controls, secure configuration management, credential protection, password policy implementation, proper logging practices, server-side security enforcement, rate limiting mechanisms, business rule validation, concurrency controls, privilege management, resource management controls, validation framework implementation, password hashing implementation, authentication security controls, secret management implementation, and framework security implementation. Practice these vulnerabilities in WebGoat to build detection and exploitation skills before testing production applications.

Bibliography

[1] OWASP Foundation. *OWASP Top 10:2021 - The Ten Most Critical Web Application Security Risks*. Available: `https://owasp.org/Top10/`.

[2] MITRE Corporation. *CWE-1344: Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')*. Available: `https://cwe.mitre.org/data/definitions/1344.html`.

[3] Mend Security. *OWASP Top 10 CWE Coverage Documentation*. Available: `https://docs.mend.io/legacy-sast/latest/owasp-top-10-cwe-coverage`.