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 19, 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 → 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.

3.2 CWE-20: Improper Input Validation

Description: The application previously accepted numerous user-controlled inputs (registration data, cart item quantities, book metadata, author names, payment form field) without comprehensive server-side validation. Only sparse annotations (e.g. `NotBlank`) existed on a subset of entity fields and were inconsistently enforced (many controller endpoints lacked `Valid`). This gap allowed malformed, extreme, or semantically invalid values (negative or enormous quantities, impossible years, invalid ISBNs, oversized strings) to reach persistence and business logic layers, risking data integrity degradation, logic manipulation, and denial-of-service via oversized payloads. Recent changes introduced structured DTOs and Bean Validation constraints to close these vectors.

CWE Explanation: CWE-20 (Improper Input Validation) arises when an application fails to define and enforce syntactic, semantic, or range constraints on externally supplied data before use. Inadequate validation can cascade into numerous secondary weaknesses (injection, overflow, business rule abuse) by allowing maliciously crafted input to impact data stores, processing logic, or downstream APIs.

Severity: Medium (Residual Low after implemented controls) Core validation now present; remaining exposure relates to legacy MVC form bindings and unvalidated optional fields slated for refactor.

Vulnerability Locations and Type

- **Entity direct binding**: Controllers bound domain entities (e.g. `Customer`, `Book`) directly from request bodies or form submissions, inheriting every writable field without a whitelist.
- **Missing range/format constraints**: No length or pattern enforcement for ISBN, phone numbers, names, addresses; numeric fields permitted negative or extreme values (price, quantity, year, numberOfCopies).
- **Quantity manipulation**: Cart add-item accepted arbitrary (including zero/negative) quantities enabling inventory or pricing anomalies.
- **Payment form**: Credit card number field had no format, length, or checksum validation (and is non-functional demo data).
- **Silent extra fields**: Jackson deserialization accepted unknown JSON properties (risk of future mass assignment if sensitive setters are added).

Mitigation Strategy and Rationale

Adopt a defence-in-depth input sanitation model: (1) Introduce dedicated request DTOs with explicit whitelists and Bean Validation annotations; (2) Apply semantic, syntactic, and range constraints (regex, size, numeric bounds); (3) Enforce validation by annotating controller method parameters with `@Valid`; (4) Fail fast with a consistent error contract via a global exception handler; (5) Harden deserialization to reject unknown properties, preventing accidental acceptance of stray or future-sensitive fields. This systematic approach ensures only well-formed, business-compliant data continues beyond the controller boundary.

Implemented Security Controls

1. **CustomerRegistrationDto Bean Validation**: Username, password, personal and contact fields now constrained with length, pattern, and format annotations (e.g. `Email`, phone regex) and enforced through `Valid`.
2. **Book field constraints**: Added size, ISBN pattern (ISBN-10/13), year range (1450–2100), non-negative price with scale check, and bounded copy counts.
3. **Cart add-item validation**: Quantity now requires minimum 1 (`Min(1)`), preventing negative or zero-impact manipulations.
4. **Global validation handler**: Central `ControllerAdvice` standardises 400 responses and prevents verbose default error leakage.

5. **Strict deserialization**: `spring.jackson.deserialization.fail-on-unknown-properties true` rejects payload fields not explicitly declared in DTOs, blocking silent mass-assignment style expansion.
6. **Password hashing continuity**: Ensures even validated credential inputs are irreversibly stored (complements validation by preserving confidentiality).
7. **Field renaming hygiene**: Normalised `bookName` naming to prevent tooling / lint confusion and encourage consistent property-level validation.

Planned / Remaining Controls

- Introduce DTOs and validation for MVC Book/Author form submissions to replace direct entity binding.
- Add validation (or removal) of the demo credit card field with checksum (Luhn) if retained.
- Enforce maximum aggregate payload sizes at reverse proxy and apply additional per-field size caps where user-generated text is later introduced.
- Add rate limiting (ties to CWE-799) to reduce brute-force probing of validation boundaries.
- Add automated tests asserting rejection of malformed ISBNs, negative quantities, overlong fields.
- Implement business rule validators (e.g. inventory non-negative after operations) as custom constraints or service assertions.

Solution Effectiveness

- **Whitelisting DTOs**: Replaces broad entity binding with curated inputs, eliminating inadvertent exposure of internal or future sensitive fields (prevents mass assignment class of CWE-20 derived issues).
- **Bean Validation enforcement**: Declarative constraints ensure malformed data is rejected pre-persistence, safeguarding data integrity and reducing attack surface for logic abuse or cascading injection vectors.
- **Semantic constraints**: ISBN regex, numeric bounds, and size limits prevent structurally invalid data that could otherwise fuel business logic anomalies or storage bloat.
- **Fail-fast error handling**: Consistent 400 responses shrink observability for attackers attempting boundary probing (no stack traces or partial processing side-effects).
- **Unknown property rejection**: Blocks introduction of rogue fields trying to smuggle unexpected values, a common foothold for privilege inflation or latent injection when new setters appear.
- **Quantity constraints**: Prevent negative or zero manipulations that could be leveraged to produce free/credit scenarios or integer underflow in future calculations.
- **Normalization & naming hygiene**: Clear, conventional field naming supports static analysis and future automated policy checks, reducing misconfiguration risk.

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

Description: A structured audit of all server-rendered HTML (Thymeleaf templates under `src/main/resources/` and relevant controller pathways found *no exploitable Cross-Site Scripting vectors* in the current codebase. All user-controlled fields are emitted via Thymeleaf's escaped expression and form binding mechanisms (e.g. `th:text`, `th:field`, attribute processors) which HTML-encode special characters. There is **no** use of unescaped output primitives (`th:utext`), inlined JavaScript expression substitutions (`[[...]]` / `[(...)]`), or manual string concatenation inserting raw request data into script blocks or event handler attributes. Earlier template refactors (e.g. renaming to `bookName`) maintained escaped bindings and removed any need for ad hoc unescaped placeholders. Consequently attacker-supplied input cannot break containment to execute script in another user's browser under current functionality.

CWE Explanation: CWE-79 arises when untrusted input is included in a web page without proper context-aware encoding or sanitisation, enabling the injection and execution of arbitrary script in the victim's browser (reflected, stored, or DOM-based). Preventing XSS requires systematically neutralising user-controlled characters significant to HTML/JS parsing or restricting script execution pathways.

Severity: None (Not Vulnerable at present). Would elevate if future rich text / HTML features or unescaped rendering are introduced without sanitisation.

Assessment Evidence

Audit activities and supporting findings:

- Grep searches for unescaped constructs returned none: patterns `th:utext` and inline substitution delimiters `[[` / `[(` (used for inlined JS) are absent.
- No occurrences of raw output methods or manual HTML string assembly in controllers/services; all views resolved through Thymeleaf templates.
- All dynamic textual content rendered with escaping processors (`th:text`, `th:value`, `th:field`); no direct usage of `th:utext` (which would bypass encoding).
- No template inlined JavaScript blocks containing template expressions; only static script includes are present.
- No reflection of request parameters into error pages or query string echoes; validation errors are summarised as plain escaped messages via the global handler.
- Input validation (length/pattern constraints) reduces payload surface even if an encoding bypass were later introduced.
- Template field rename adjustments (`book_name -¿ bookName`) avoided broken bindings that might otherwise tempt unescaped insertions.

Mitigation Strategy and Rationale

Maintain a *strict auto-escape* posture while adding forward-looking guardrails: (1) rely solely on Thymeleaf's escaped output features for plain text; (2) introduce Content Security Policy (CSP) to constrain script execution sources; (3) forbid unescaped directives (`th:utext`) except under explicit security review with server-side sanitisation; (4) implement lint / CI checks for introduction of risky constructs; (5) if future rich text/user HTML is required, apply a conservative allow-list sanitizer before persistence and still encode on output.

Implemented / Planned Controls

1. **Escaped template rendering (implemented)**: Exclusive use of `th:text`, `th:field`, and attribute processors ensures HTML entity encoding.
2. **Absence of unescaped primitives (implemented)**: No `th:utext` or inline expression substitutions eliminates common bypass channels.
3. **No dynamic script construction (implemented)**: No server-sourced data injected into inline `<script>` blocks or event handler attributes.
4. **Input validation (implemented)**: Bean Validation narrows malicious payload shapes (defence-in-depth only).
5. **CSP header (planned)**: Add restrictive `Content-Security-Policy` (e.g. `script-src ’self’; object-src ’none’; base-uri ’self’; frame-ancestors ’none’`) to mitigate impact if an injection slips through.
6. **Template guardrail (planned)**: CI rule / review checklist to flag additions of `th:utext`, inline script expressions, or event attributes containing untrusted expressions.
7. **Rich text sanitisation policy (planned)**: If future features require formatted user content, adopt an allow-list HTML sanitizer (e.g. OWASP Java HTML Sanitizer) prior to storage.
8. **Security headers bundle (planned)**: Referrer-Policy, Permissions-Policy, and strengthening cookies (Secure, HttpOnly, SameSite) to reduce session theft avenues in case of future XSS.

Solution Effectiveness

- Auto-escaping neutralises script meta-characters (¡, ¿, &, quotes) preventing payload interpretation as active markup.
- Lack of unescaped or inline script sinks removes primary exploit vectors (no DOM insertion points for attacker data).
- Planned CSP provides a containment layer limiting execution even if an HTML injection is discovered later (e.g. blocks external script exfiltration).
- Guardrail automation (grep / CI checks) lowers regression risk as templates evolve.
- Sanitisation policy planning ensures future feature expansion (rich text) introduces structured controls rather than ad hoc exceptions.

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

4.1 CWE-307: Improper Restriction of Excessive Authentication Attempts

Description: The application originally permitted unlimited rapid authentication attempts against the `/login` endpoint with no per-username or per-IP throttling, enabling brute force, credential stuffing, and password spraying attacks. Attackers could iterate common or leaked credential pairs without delay, increasing account takeover likelihood and aiding enumeration of valid usernames through timing/lockout observation (although failure messages were uniform). This condition constitutes CWE-307 prior to recently implemented controls.

CWE Explanation: CWE-307 arises when an application fails to enforce limits on repeated authentication attempts (e.g. password, MFA, reset token submissions), allowing automated high-frequency guessing that can compromise accounts or facilitate credential stuffing campaigns. Proper controls (lockout, backoff, rate limiting) reduce the feasible attempt rate and raise detection opportunities.

Severity: Low (Residual) — Core throttling and lockout controls now implemented; remaining exposure is limited to single-instance memory scope and absence of distributed correlation.

Vulnerability Locations and Type

- **Login endpoint**: POST `/login` accepted unlimited sequential failures (no max attempt threshold, delay, or CAPTCHA escalation).
- **Lack of per-identity lockout**: No temporary suspension after repeated incorrect credentials for the same username.
- **Lack of IP spray detection**: No broader rate control for an IP submitting many different usernames (credential stuffing pattern).
- **Absence of central attempt telemetry**: No counters or logs dedicated to failed authentication bursts (reduced detection / alerting capability).

Mitigation Strategy and Rationale

Adopt a layered throttling model combining *pre-auth rejection* (cheap filter), *account/IP scoped lockout*, and *reset on success*: (1) Intercept login requests prior to authentication to block already locked principals (resource preservation); (2) Maintain rolling failure windows and impose temporary locks (discourages brute force); (3) Track both username and source IP to differentiate targeted vs spray attacks; (4) Provide generic error responses to avoid username enumeration; (5) Reset counters on successful authentication to reduce accidental lockouts and user friction. This approach sharply lowers feasible guessing throughput while keeping implementation lightweight.

Implemented / Planned Security Controls

1. **Failure counting service (implemented)**: `LoginAttemptService` tracks failures per username (threshold 5 in 15 min) and per IP (threshold 30) and enforces a 10 minute lock via timestamp comparison.
2. **Pre-auth rate limiting filter (implemented)**: `LoginRateLimitingFilter` short-circuits POST `/login` when a username or IP is locked, returning HTTP 429 (Too Many Requests) without invoking downstream authentication logic.
3. **Success / failure handlers (implemented)**: `AuthenticationHandlers` records failures and clears username attempts upon success, limiting persistent false positives.
4. **Uniform error messaging (existing)**: Generic `/login?error` response avoids disclosing whether the username or password was invalid, reducing enumeration signal.
5. **Session fixation protection (existing)**: Complements throttling by ensuring hijacked pre-auth sessions cannot be leveraged mid-brute force.
6. **Planned distributed store (planned)**: Externalise counters to Redis or similar for horizontal scaling and shared lock state across instances.
7. **Planned anomaly alerting (planned)**: Emit structured security log events (e.g. JSON) for failed attempt spikes to feed SIEM alert rules.
8. **Optional adaptive challenges (planned)**: Introduce CAPTCHA or MFA step-up after multiple failures to add friction for automation.

Solution Effectiveness

- **Throughput reduction**: Lock + 429 response immediately halts further processing after threshold, capping guess attempts per window.
- **Spray mitigation**: Separate IP threshold limits credential stuffing patterns distributing attempts across many usernames.
- **Low overhead**: Early filter rejection conserves CPU and avoids unnecessary password hash comparisons under attack.
- **Reset on success**: Minimises legitimate user friction (no long-lived stale counters) while preserving deterrence for attackers.
- **Scalability path**: Planned move to distributed cache preserves semantics in multi-node deployment, preventing bypass via node cycling.
- **Extensibility**: Architecture supports extension to MFA attempt throttling or account recovery flows using same service abstraction.

Current posture: *Not vulnerable in current single-instance context.* Residual risk relates to lack of distributed correlation and absence of automated alerting — both scheduled for future enhancement.

4.2 CWE-654: Reliance on a Single Security Mechanism

Description: Earlier application iterations exhibited several controls implemented as a *single protective layer* (e.g. relying only on URL pattern role checks without ownership validation, only template auto-escaping without complementary Content Security Policy, only password length without strength criteria, only a brute-force threshold without password complexity). Such single points of failure meant a bypass or misconfiguration of that one mechanism could fully expose the protected asset class (authorization, session integrity, input handling). Recent changes introduced layered, mutually reinforcing safeguards reducing the likelihood that a single defect reopens critical exploit paths.

CWE Explanation: CWE-654 (Reliance on a Single Security Mechanism) occurs when security depends exclusively on one control whose failure (design flaw, misconfiguration, regression) leads directly to compromise. Defence-in-depth requires stacking independent, diverse mechanisms so that breaking one does not immediately yield the target outcome.

Severity: Low (Residual) — Core high-impact areas (authentication, authorization, input handling, XSS prevention) now have multiple layers; remaining items are enhancement opportunities (e.g. MFA, distributed rate limiting, security event monitoring).

Former Single-Layer Examples

- **Authorization**: Initially only antMatcher path rules; missing method-level and ownership checks (fixed via @PreAuthorize + per-resource ownership validation).
- **User registration validation**: Relied solely on basic length checks (no central strength or uniqueness service); now augmented with regex-based complexity + duplicate detection service reused by API and MVC.
- **XSS prevention**: Depended only on Thymeleaf escaping; CSP and restrictive security headers now add containment.
- **Brute force defence**: Absent (unlimited attempts) then singular (lockout only); now combined with per-username/IP counters + strong password policy (different control classes).
- **Session confidentiality**: Previously dependent only on password hashing and implicit session rotation; now complemented by cookie flags (Secure/HttpOnly/SameSite) and planned TLS enforcement + HSTS (prod).

Mitigation Strategy and Rationale

Refactor high-risk pathways to adopt *layered, heterogeneous controls*: (1) combine coarse-grained (path) and fine-grained (method + ownership) authorization; (2) pair input auto-escaping with CSP to constrain script execution; (3) augment brute-force rate limiting with credential quality requirements; (4) segregate validation logic into a shared service to prevent bypass via alternative controller flows; (5) harden session tokens with transport + cookie attributes rather than relying solely on secrecy; (6) plan observability (logging/alerting) to detect failures when controls degrade.

Implemented / Planned Controls

1. **Authorization layering (implemented)**: Path matcher + @PreAuthorize + ownership checks (customers, carts) reduce single failure risk.
2. **Central registration validation (implemented)**: Shared `RegistrationValidationService` ensures password strength and uniqueness on both API and MVC paths.
3. **Password complexity (implemented)**: Regex enforces multi-class, length ≥ 12; complements throttle logic.
4. **Brute force throttling (implemented)**: Username/IP attempt tracking + lockout independent of password policy.
5. **CSP + security headers (implemented)**: CSP, Referrer-Policy, Permissions-Policy, frame denial, content type options reduce reliance on template escaping.
6. **Cookie hardening (implemented)**: Secure, HttpOnly, SameSite=Strict flags add transport and CSRF resilience around session identifier.
7. **Session fixation mitigation (implemented)**: Explicit session migration adds session integrity to cookie/transport protections.

Solution Effectiveness

- **Multiple independent checks**: Compromise now requires defeating distinct classes (e.g. both path matcher and ownership, or escaping and CSP).
- **Regressive change resilience**: A misconfigured path rule no longer exposes data absent simultaneous method/ownership regression.
- **Reduced credential attack ROI**: Attackers must overcome both throttling and stronger passwords, lowering feasible success probability per unit time.
- **Containment of injection attempts**: Even if raw HTML were introduced inadvertently, CSP constrains script execution sources.
- **Consistent validation reuse**: Central service prevents alternate controller paths from silently bypassing strength/uniqueness checks.
- **Future scalability path**: Planned distributed counters and MFA further decrease single-point reliance as deployment complexity grows.

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

5.1 CWE-250: Execution with Unnecessary Privileges

Description: The BookShop application's containerized deployment originally executed the Java application process as the root user within the Docker container, providing unnecessary administrative privileges that could be exploited in case of application compromise. The base `openjdk:17-jdk-slim` image defaults to root execution, and the Dockerfile lacked explicit privilege dropping or non-root user creation. This elevated privilege context increases the potential blast radius of any remote code execution, container escape, or file system manipulation attacks targeting the application runtime.

CWE Explanation: CWE-250 (Execution with Unnecessary Privileges) occurs when a program runs with more privileges than required for its intended functionality, enabling attackers who compromise the application to leverage those excess rights for privilege escalation, system manipulation, or lateral movement that would otherwise be prevented by proper isolation.

Severity: Medium (Residual Low after implemented controls) — Core runtime now uses non-root user; remaining exposure relates to container orchestration hardening and potential shared kernel vulnerabilities.

Vulnerability Locations and Type

- **Container root execution**: Dockerfile lacked `USER` directive, causing Java process to inherit root privileges from base image.
- **Filesystem write permissions**: Root ownership of application directory enabled arbitrary file creation/modification by compromised process.
- **Broad container capabilities**: Default Docker capabilities (though restricted compared to host root) still provided unnecessary network, process, and system manipulation abilities.
- **No privilege boundaries**: Absence of read-only filesystem or capability dropping meant successful exploitation could modify container state persistently.

Mitigation Strategy and Rationale

Implement *principle of least privilege* at container level: (1) Create dedicated non-root user for application execution; (2) Set proper file ownership and restrict write access to necessary directories only; (3) Use read-only root filesystem with limited writable mounts; (4) Drop unnecessary Linux capabilities and prevent privilege escalation; (5) Apply security contexts that enforce non-root execution and prevent capability acquisition. This approach ensures that even if the Java application is compromised, the attacker gains minimal system access.

Implemented / Planned Security Controls

1. **Non-root user creation (implemented)**: Added app system user and group in Dockerfile with restricted shell access.
2. **File ownership transfer (implemented)**: Set `chown -R app:app /app` to ensure application files owned by non-privileged user.
3. **USER directive (implemented)**: Explicit `USER app` instruction prevents root process execution.
4. **Read-only filesystem (planned)**: Docker Compose `read_only: true` with `tmpfs` for writable areas.
5. **Capability dropping (planned)**: `cap_drop: ALL` and selective `cap_add` for required network binding only.
6. **Security options (planned)**: `no-new-privileges:true` prevents runtime privilege escalation attempts.
7. **Minimal base image (planned)**: Evaluate migration to distroless or Alpine-based images for reduced attack surface.
8. **Runtime privilege validation (planned)**: Application startup check refusing to run as root user.

Solution Effectiveness

- **Privilege containment**: Non-root execution limits file system manipulation to application-owned directories and prevents system-level modifications.
- **Reduced exploit impact**: Compromise of Java process no longer yields container administrative access or ability to modify critical system files.
- **Defense against container escape**: Limited capabilities and non-root context reduce effectiveness of kernel vulnerability exploitation.
- **Compliance alignment**: Follows container security best practices (CIS Docker Benchmark) for production deployment.
- **Forensic clarity**: Non-root execution provides cleaner audit trails distinguishing application activity from system operations.

5.2 CWE-693: Protection Mechanism Failure

Description: The BookShop application originally suffered from a critical protection mechanism failure where CSRF (Cross-Site Request Forgery) protection was deliberately disabled in the security configuration, creating a single point of failure that undermined the effectiveness of other security controls. The disabled CSRF protection exposed all state-changing operations to cross-site request forgery attacks, allowing malicious websites to perform unauthorized actions on behalf of authenticated users. This represented a classic example of protection mechanism failure where a fundamental security control was intentionally disabled, creating vulnerability despite other defensive measures being in place.

CWE Explanation: CWE-693 (Protection Mechanism Failure) occurs when a protection mechanism fails to provide adequate security for the system, often due to disabled security controls, single points of failure, or inadequate implementation of security mechanisms that other controls depend upon for their effectiveness.

Severity: High (Resolved) — Critical protection mechanism failure eliminated; defense-in-depth now properly implemented with multiple overlapping security layers.

Vulnerability Locations and Type

- **Disabled CSRF protection**: SecurityConfig explicitly disabled CSRF with `.csrf(AbstractHttpConfigur` creating a single point of failure in protection mechanisms.
- **Missing CSRF tokens**: All state-changing forms (login, registration, cart operations, admin functions) lacked CSRF token validation.
- **Inadequate cross-origin controls**: No CORS configuration to prevent unintended cross-origin requests in API endpoints.
- **Single-layer dependencies**: Other security mechanisms (authentication, authorization, session management) depended on request authenticity that CSRF protection should have provided.

Mitigation Strategy and Rationale

Implement *defense-in-depth with no single points of failure*: (1) Re-enable CSRF protection with proper token handling across all forms; (2) Add CSRF tokens to all state-changing operations while maintaining API flexibility; (3) Implement proper origin validation for cross-origin requests; (4) Ensure multiple overlapping security controls protect critical operations; (5) Validate that security mechanisms complement rather than replace each other. This approach ensures that compromise of any single protection mechanism does not fully expose the application to attack.

Implemented / Planned Security Controls

1. **CSRF protection re-enabled (implemented)**: Modified SecurityConfig to enable CSRF protection with selective exclusion for API endpoints (`/api/**`).
2. **CSRF token integration (implemented)**: Added `<input type="hidden" th:name="$_csrf.paramete` `th:value="$_csrf.token"/>` to all state-changing forms.
3. **Comprehensive form coverage (implemented)**: Applied CSRF tokens to login, registration, cart operations, book management, author management, and logout forms.
4. **Origin validation (implemented)**: Restrictive approach avoiding overly permissive CORS configuration that could reintroduce vulnerabilities.
5. **Layered authorization (existing)**: Path-based + method-level + ownership checks provide multiple authentication/authorization layers.
6. **Session security hardening (existing)**: Secure/HttpOnly/SameSite cookie attributes + session fixation protection complement CSRF protection.

7. **Input validation layers (existing)**: Bean validation + central password policy + rate limiting provide additional protective layers.
8. **Security headers defense (existing)**: CSP, Referrer-Policy, Permissions-Policy provide browser-level protection complementing server-side controls.

Solution Effectiveness

- **Eliminates single point of failure**: CSRF protection no longer represents a disabled critical control that other mechanisms depend upon.
- **Cross-site attack prevention**: All state-changing operations now protected against cross-site request forgery attempts.
- **Defense-in-depth validation**: Multiple overlapping security controls ensure that failure of one layer does not compromise overall security.
- **Request authenticity assurance**: CSRF tokens ensure that state-changing requests originate from legitimate user interactions within the application.
- **Maintains API flexibility**: Selective CSRF exclusion for API endpoints preserves stateless authentication capabilities while protecting web forms.
- **Production-ready posture**: All critical protection mechanisms now enabled and properly configured for production deployment.

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`.