Orderly

Security Hardening Report
Part 2 — Fixing Issues & Future Development

Carolina Reis *131193*
Eduardo Lopes *103070*

Universidade de Aveiro

Security in Software Engineering — SES25_104

June 16, 2026

Contents

1 Executive Summary

This report documents the security hardening work performed on the **Orderly** application as Part 2 of the Security in Software Engineering group project. Building on the vulnerabilities identified in the Part 1 security assessment, the team replaced the unprotected baseline with a layered security architecture covering authentication, authorisation, API hardening, and perimeter defence.

The baseline had **no authentication**, accepted any request regardless of origin, leaked internal data in responses, and exposed MySQL directly on the host network. After hardening, every API request requires a valid **JWT (RS256)** issued by **Keycloak 26.2** via OpenID Connect Authorization Code + PKCE. Access is enforced at two layers: route-level via Spring Security and object-level via service-layer ownership checks. A **ModSecurity/OWASP CRS** WAF in blocking mode sits in front of the application; TLS 1.3 terminates at the Nginx perimeter.

All **8 abuse stories** from Part 1 are classified as **Prevented**. SAST (Trivy + SonarQube) and DAST (ZAP ×2: through WAF and direct) re-assessments were performed and are documented in Section 8.

Table 1: Summary of Part 2 requirements

Requirement	**Key control implemented**	**Status**
API Hardening (OWASP API Top 10)	DTOs, RBAC, rate limiting, error handling, OpenAPI	Done
Authentication & Identity	Keycloak 26.2, JWT RS256, PKCE S256, refresh rotation	Done
Explicit Authorization Layer	roles.yml, route-level + object-level RBAC, 19 unit tests	Done
Perimeter Defence	Nginx + ModSecurity CRS v4 blocking, TLS 1.3, rate limiting	Done
Zero Trust Framing	NIST SP 800-207: Subject/PEP/PDP/PA, 7 tenets mapped	Done
Re-Assessment & Evidence	SAST before/after, DAST ×2, abuse story regression	Done

2 Architecture Overview

2.1 Before (Part 1 Baseline)

The Part 1 baseline consisted of three Docker containers (React frontend, Spring Boot backend, and MySQL) with all ports exposed directly to the host. There was no authentication, no authorisation, no rate limiting, and no perimeter defence. MySQL was reachable at `localhost:3306` from any host process.

2.2 After (Part 2 Hardened Architecture)

Table 2: System components after hardening

Component	Technology	Host Port	Role
WAF / Reverse Proxy	Nginx 1.30.1 + ModSecurity v3 + OWASP CRS 4.25.0	8080 / 8443	Sole public entry point
Identity Provider	Keycloak 26.2	8180	OIDC/JWT issuer
Frontend	React 19 + Vite	internal only	SPA
Backend API	Spring Boot 4.0.3 / Java 21	internal only	REST API
Database	MySQL 8.0	internal only	Persistence

The key architectural change is the introduction of Nginx + ModSecurity as the sole public entry point, Keycloak as the external Identity Provider, and the removal of all direct host-port mappings for the backend, frontend, and MySQL.

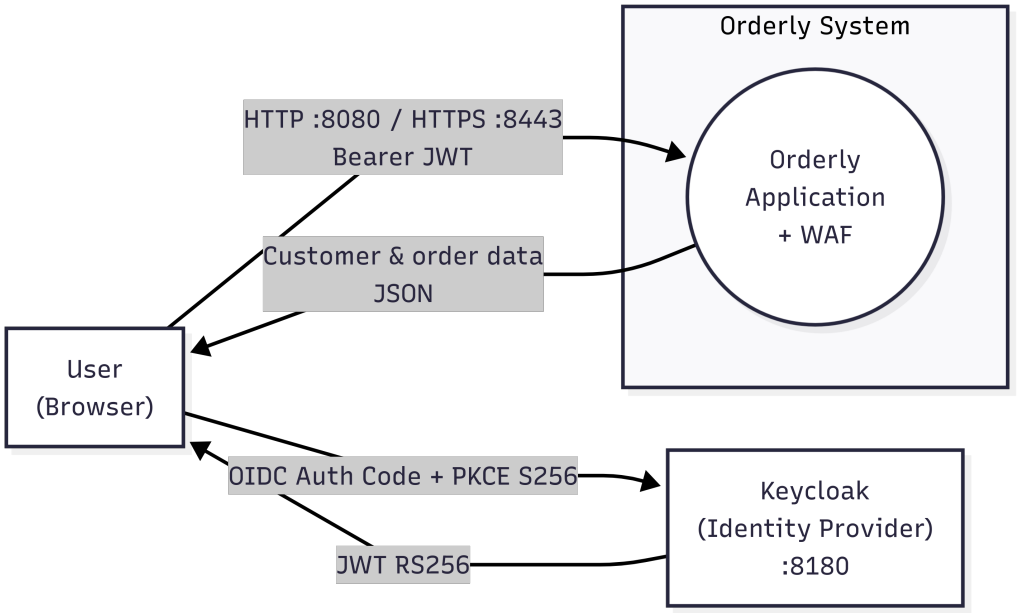

Figure 1: Level 0 DFD — System context (Part 2): sole external actor is the authenticated browser; all internal services hidden behind the WAF boundary

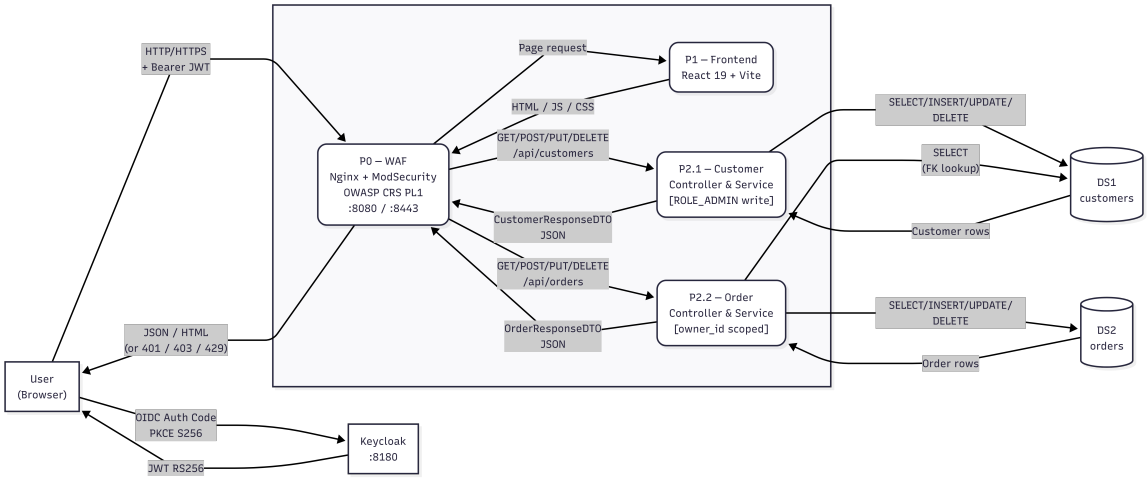

Figure 2: Level 1 DFD — Hardened architecture (Part 2): WAF, Keycloak, frontend, backend, and database with trust boundaries

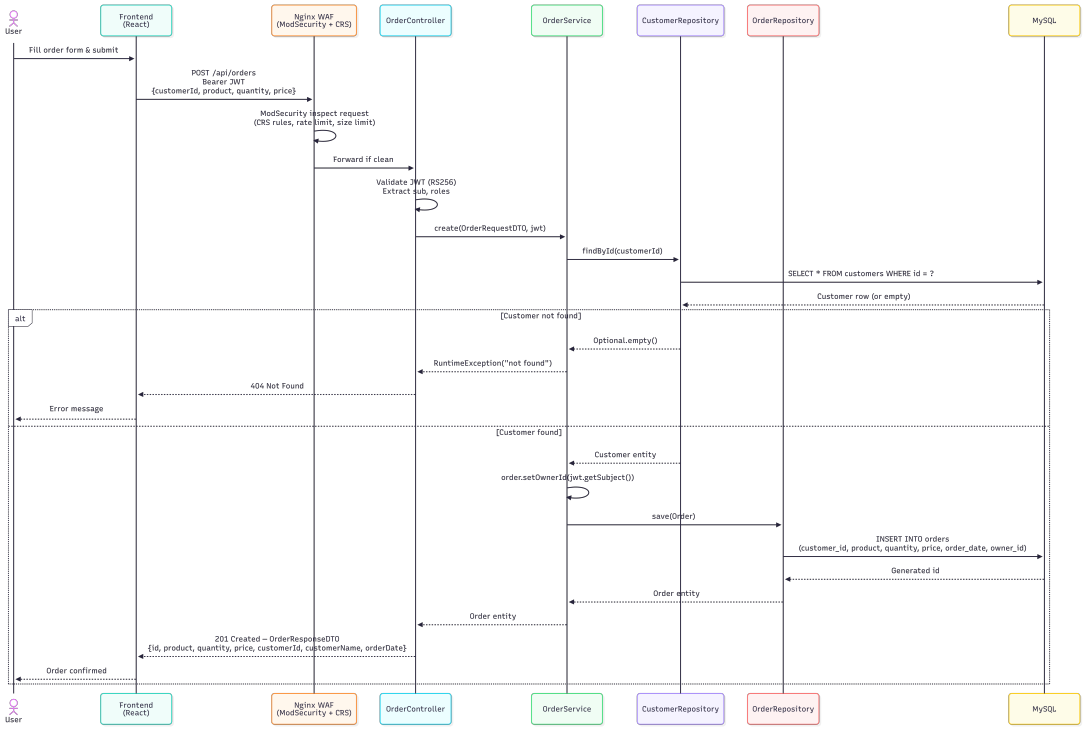

Figure 3: Level 2 DFD — Backend API detail: JWT validation, ownership check, DTO mapping, and DB interaction flows

3 Requirement 1 — API Hardening (OWASP API Top 10)

3.1 API1 — Object-Level Authorization (BOLA)

Commit tag: `[API1-BOLA][AUTHZ]`

The baseline allowed any caller to read, modify, or delete any record regardless of ownership. The fix introduces order-level ownership: every order stores the Keycloak `sub` UUID of its creator in an `owner_id` column.

- `Order.java` — `owner_id VARCHAR(36)` column added (Keycloak `sub`)
- `OrderService.checkOwnership()` — throws `403 Forbidden` if the caller's `sub` does not match `order.owner_id` and the caller is not `ROLE_ADMIN`
- `CustomerService` — all write operations restricted to `ROLE_ADMIN`; reads require any authenticated user

Closes: AS-01 (unauthenticated access), AS-02 (data harvesting), AS-03 (ID enumeration tampering).

3.2 API3 — Property-Level Authorization (BOPLA)

Commit tag: `[API3-BOPLA]`

The baseline serialised the full entity object in responses, leaking internal fields, and accepted arbitrary fields in request bodies (overposting). Request and response DTOs were introduced:

- `CustomerRequestDTO` — accepts only `name`, `email`, `phone`; ignores `id` and all other fields
- `CustomerResponseDTO` — returns only `id`, `name`, `email`, `phone`
- `OrderRequestDTO` — accepts only `customerId`, `product`, `quantity`, `price`; `owner_id` and `orderDate` are server-assigned
- `OrderResponseDTO` — flat projection exposing `id`, `product`, `quantity`, `price`, `orderDate`, `customerId`, `customerName`; no nested `Customer` entity, no `owner_id`
- `@Valid` annotation added to all `@RequestBody` parameters

Closes: AS-04 (payload overposting).

3.3 API4 — Rate Limiting and Request-Size Controls

Commit tag: `[API4]`

Two-layer rate limiting is enforced at the perimeter and at the application level.

Perimeter (Nginx): `/api/*` — 60 req/min per IP, burst 20, returns `429`. Frontend assets — 120 req/min per IP, burst 30. `client_max_body_size 1m` returns `413` for oversized requests.

Application (Bucket4j + Caffeine): Per-user buckets keyed by JWT `sub` (not IP - resistant to IP spoofing): `GET` 100 req/min; `POST`/`PUT` 20 req/min; `DELETE` 10 req/min. Returns `429 Too Many Requests`.

3.4 API5 — Function-Level Authorization (Deny-by-Default)

Commit tag: `[API5]`

The `SecurityFilterChain` in `SecurityConfig.java` applies a deny-by-default posture. Every admin-only endpoint additionally carries `@PreAuthorize("hasRole('ADMIN')")` as a second enforcement layer.

```
/api-docs/**, /swagger-ui/** -> permitAll()
GET  /api/customers/**       -> authenticated()
POST /api/customers/**       -> hasRole("ADMIN")
PUT  /api/customers/**       -> hasRole("ADMIN")
DELETE /api/customers/**     -> hasRole("ADMIN")
DELETE /api/orders/**        -> hasRole("ADMIN")
anyRequest                   -> authenticated()   // catch-all deny
```

Order read/create/update endpoints (`GET`, `POST`, `PUT` on `/api/orders/**`) are not explicitly listed — they fall to the catch-all `authenticated()` rule. Object-level ownership filtering then applies inside `OrderService` for each operation.

3.5 API8 — Security Configuration Hardening

Commit tag: `[API8]`

HTTP security headers configured in `SecurityConfig.java`:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Frame-Options: DENY
X-Content-Type-Options: nosniff (Spring Security default)
Content-Security-Policy: default-src 'self'; script-src 'self';
   object-src 'none'; frame-ancestors 'none'
```

Error handling — `GlobalExceptionHandler` (`@RestControllerAdvice`) maps exceptions to appropriate HTTP status codes:

- `MethodArgumentNotValidException` → `400 Bad Request` with field-level violations
- `DataIntegrityViolationException` → `409 Conflict`
- `AccessDeniedException` → `403 Forbidden` with a generic body
- `RuntimeException` with "not found" message → `404 Not Found`
- catch-all `Exception` → `500` with body `{"error":"Internal Server Error"}`

Stack trace removal enforced in `application.properties`:

```
server.error.include-stacktrace=never
server.error.include-message=never
server.error.include-exception=false
```

Closes: AS-05 (invalid data injection), AS-06 (error-based information disclosure).

3.6 API9 — OpenAPI Specification

Commit tag: `[API9]`

`springdoc-openapi-starter-webmvc-ui:2.8.8` added to `pom.xml`. All controllers are annotated with `@Tag`, `@Operation`, and `@ApiResponse`. The generated specification is committed as `openapi.json` and served at `http://localhost:8080/swagger-ui.html` (proxied through Nginx).

4 Requirement 2 — Authentication & Identity Management

4.1 Technology Choices

Table 3: Authentication technology decisions

Component	Choice	Rationale
Identity Provider	Keycloak 26.2	Self-hosted, OIDC/OAuth2, refresh token rotation and server-side revocation out of the box
Frontend auth	keycloak-js 26.2	Official adapter; manages tokens in memory (not localStorage), preventing XSS token theft
Token format	JWT, signed RS256	Asymmetric signing; backend validates with public JWKS key, no shared secret needed
PKCE method	S256	SHA-256 code challenge; prevents authorization code interception
Access token TTL	300 s (5 min)	Short-lived; minimises exploitation window if intercepted
Refresh token TTL	36000 s / idle 1800 s	Rotation on every use; proactive refresh via keycloak-js timer

4.2 CSRF Stance

All API requests carry a `Bearer` token in the `Authorization` header. No application-domain cookies are used. Because no cookie-bearing flows exist on the application side, **CSRF is not applicable** at the API layer. The Keycloak SSO session cookie is scoped to the Keycloak domain and is never read or forwarded by the application backend (`KEYCLOAK_SESSION` is a Keycloak-owned cookie, not an application cookie).

4.3 Authentication Flows

Flow 1 — Login (Authorization Code + PKCE): The user is redirected to Keycloak with a PKCE S256 code challenge. After authentication, Keycloak redirects back with a one-time authorization code. The keycloak-js adapter exchanges it for tokens (JWT RS256 access token, 5 min; refresh token, 10 h max / 30 min idle). Tokens are stored in JavaScript memory only — keycloak-js cleans the authorization code from the URL via `history.replaceState`.

Flow 2 — Token Refresh (Rotation): keycloak-js runs a proactive timer every 30 seconds calling `updateToken(60)`. If the access token has less than 60 seconds remaining, a refresh request is issued. Keycloak returns a new access token and a **new refresh token**, immediately invalidating the old one (`revokeRefreshToken=true`, `refreshTokenMaxReuse=0`).

Flow 3 — Logout: The Logout button redirects to Keycloak's OIDC end-session endpoint with an `id_token_hint`. Keycloak terminates the SSO session, revokes all tokens server-side, and redirects back to the application root.

Flow 4 — Refresh Token Reuse Detection: If an already-invalidated refresh token is presented, Keycloak detects the reuse and terminates the **entire session** — all tokens are revoked. The legitimate user's next `updateToken()` call fails, triggering automatic `keycloak.logout()`.

5 Requirement 3 — Explicit Authorization Layer

5.1 Ownership Model

Model A was adopted: customers are shared business records managed exclusively by `ROLE_ADMIN`. The `owner_id` field lives on the **Order** entity, set automatically from the JWT `sub` claim at creation time. Regular users (`ROLE_USER`) can create orders for any customer but can only view, update, or delete their own orders.

5.2 Roles as Data

Roles and permissions are declared in `backend/src/main/resources/roles.yml` and loaded at startup via `@ConfigurationProperties`. No role strings are hardcoded in controller logic.

```
app:
  roles:
    admin:
      description: "Full CRUD access to all customers and orders"
      resource-scope: all
      operations: [CREATE, READ, UPDATE, DELETE]
    user:
      description: "CRUD restricted to own orders; DELETE not
         permitted"
      resource-scope: own
      operations: [CREATE, READ_OWN, UPDATE_OWN]
```

5.3 Enforcement Layers

Three layers of enforcement are applied in sequence:

1. **Route-level** — `SecurityFilterChain` in `SecurityConfig.java`: deny-by-default, mapping each HTTP method to the minimum required role.
2. **Controller-level** — `@PreAuthorize("hasRole('ADMIN')")` on every admin-only mapping as a second enforcement layer.
3. **Object-level** — `OrderService.checkOwnership()` compares the authenticated user's `sub` against `order.owner_id` before any data operation.

5.4 Unit Test Results

19 unit tests cover the full role × action × resource matrix:

Table 4: Unit test coverage summary

Test class	**Role**	**Scenarios**	**Tests**
CustomerServiceTest	ADMIN	Full CRUD on all customers	7
OrderServiceTest	ADMIN	findAll, findById any order, create, update any order, delete	5
	USER	findAll (own only); findById own; findById other → 403; create (sets ownerId); create unknown customer → 500; update own; update other → 403	7
		Total	**19**

All 19 tests pass. Unauthenticated access returns `401` at the route level before any service code is reached.

6 Requirement 4 — Perimeter Defence

6.1 Architecture

`owasp/modsecurity-crs:nginx-alpine` bundles Nginx 1.30.1, ModSecurity v3.0.15, and OWASP CRS 4.25.0 in a single image. It is the sole public entry point: backend, frontend, and MySQL have no host-port mappings. Traffic flows:

`Browser → Nginx:8080/8443 → {frontend | backend | keycloak}`

6.2 TLS Termination

A self-signed certificate (RSA-2048, CN=localhost, SAN=localhost/127.0.0.1) is generated at CI/CD time with `openssl req -x509` and mounted into the Nginx container. TLS 1.2 and TLS 1.3 are accepted; ECDHE ciphers are preferred. HTTP on `:8080` is retained for local development because Keycloak runs HTTP-only; removing it would cause mixed-content errors during the OIDC token exchange.

6.3 OWASP CRS Configuration

Table 5: CRS configuration parameters

Parameter	Value
`MODSEC_RULE_ENGINE`	`On` (blocking mode)
`PARANOIA`	`1` — lowest false-positive rate
`ANOMALY_INBOUND`	`5` — one CRITICAL rule = immediate block
`ANOMALY_OUTBOUND`	`4`
`ALLOWED_METHODS`	`GET HEAD POST PUT DELETE OPTIONS`

6.4 WAF Tuning

Tuning 1 — HTTP Method Enforcement (Rule 911100): The CRS default allowed method list is `GET HEAD POST OPTIONS`. Any `PUT` or `DELETE` request matched rule 911100 (*Method is not allowed by policy*), scoring 5 points (CRITICAL) and triggering an immediate block.

Root cause: The Orderly API is a REST API that uses `DELETE` and `PUT` as primary verbs for legitimate operations, documented in the OpenAPI specification (API9). This is a true false positive — legitimate application behaviour misidentified as a policy violation.

Fix: Set `ALLOWED_METHODS="GET HEAD POST PUT DELETE OPTIONS"` via the Docker Compose environment variable. The image startup script patches `crs-setup.conf` at container start.

Known FP — Vite Cache-Busting Parameter (`?v=` hash): ZAP flagged Path Traversal and SQL Injection (HIGH) on Vite's `?v=<hash>` cache-busting parameter during the Phase 0 DAST baseline. These are confirmed false positives: the `v` parameter is a build fingerprint, not user-supplied input. At CRS Paranoia Level 1 these specific rules are inactive. A targeted exclusion is documented for when the paranoia level is raised:

```
SecRuleUpdateTargetById 942100 "!ARGS:v"
SecRuleUpdateTargetById 930100 "!ARGS:v"
```

6.5 Verified Tests

- `HTTPS :8443` → `200 OK` (TLS 1.3 / AES-256-GCM)
- 2 MB request body → `413 Request Entity Too Large`
- 25 rapid `POST /api/customers` → first 20 succeed; requests 21–25 → `429`
- `DELETE /api/orders/{id}` → `204` (ALLOWED_METHODS tuning verified)

7 Requirement 5 — Zero Trust Architecture Framing

Reference: NIST SP 800-207, *Zero Trust Architecture* (August 2020).

7.1 Component Mapping

Table 6: NIST SP 800-207 components mapped to Orderly

ZT Component	Orderly Implementation
Subject	Authenticated user acting through a browser. No device certificate or posture agent is deployed (see Tenet 4).
PEP — Perimeter	Nginx + ModSecurity/CRS: inspects and blocks malicious requests, enforces rate limits and TLS before any application code runs.
PEP — Route	Spring Security `SecurityFilterChain`: rejects requests without a valid JWT or with insufficient role.
PEP — Object	`OrderService.checkOwnership()`: ensures the caller's Keycloak `sub` matches `order.owner_id` before any data access.
Policy Decision Point	Spring Security + `roles.yml`: evaluates JWT claims (signature, expiry, `realm_access.roles`) and ownership rules.
Policy Administrator	Keycloak 26.2: issues RS256 JWTs (5 min TTL), rotates refresh tokens with reuse detection, manages realm roles.

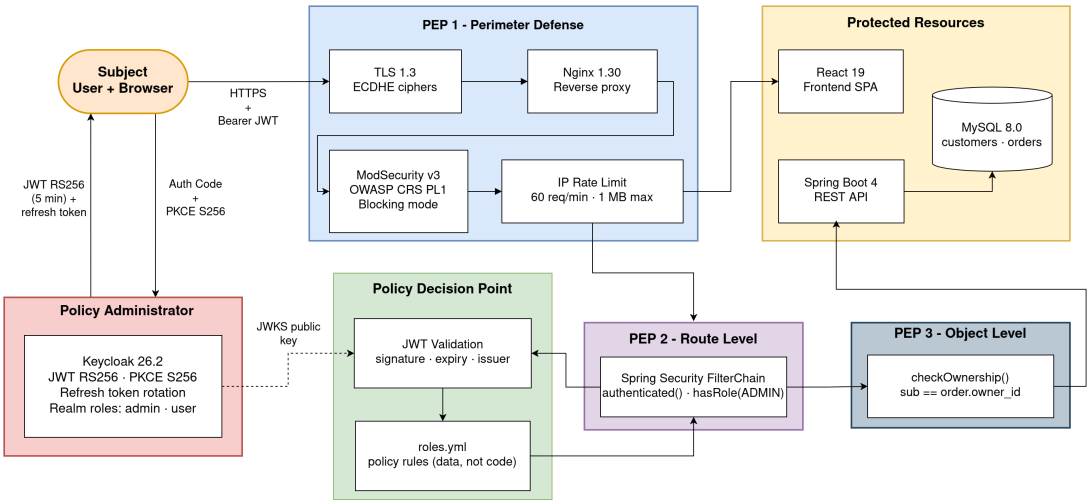

Figure 4: Zero Trust architecture diagram — Orderly (NIST SP 800-207)

7.2 Seven Tenets Mapping (NIST SP 800-207 §2.1)

Table 7: NIST SP 800-207 tenet mapping

#	**Tenet**	**Status**	**Implementation / Gap**
1	All data sources are resources	Met	Frontend, backend API, and MySQL are all protected. No service reachable outside Docker network except through Nginx. MySQL port 3306 not exposed.
2	All communication secured	Partial	HTTPS on :8443 (TLS 1.3, ECDHE). HTTP on :8080 retained for local development (Keycloak HTTP-only). Inter-container traffic unencrypted at container level.
3	Per-session access grants	Met	JWTs expire after 5 minutes. Refresh tokens rotate on every use and are revoked server-side on logout. No persistent session cookie.
4	Dynamic policy-based access	Partial	Identity and role evaluated per request. No device posture signals (certificate, MDM). No adaptive / risk-based step-up authentication.
5	Monitor and measure posture	Not Met	No SIEM, no runtime integrity monitoring, no anomaly detection. Nginx error logs go to stderr; structured telemetry is future work.
6	Authentication and authorization strictly enforced	Met	Every request validated against JWKS. Deny-by-default `SecurityFilterChain`. Object-level checks before every DB operation.
7	Collect maximum telemetry	Not Met	Basic IP + path data in Nginx error logs. No structured telemetry pipeline, no centralised dashboard, no posture scoring.

7.3 Future Work

Tenets 2, 4, 5, and 7 are not fully met. The gaps are architectural rather than implementation oversights — they represent the boundary between a hardened application (the scope of this project) and a full enterprise Zero Trust deployment.

Table 8: Zero Trust gaps and candidate solutions

Gap	**Required capability**	**Candidate solution**
Tenet 2 — Internal TLS	mTLS between containers	Service mesh (Istio / Linkerd) or Docker Swarm secrets with internal CA
Tenet 2 — Keycloak HTTPS	Keycloak behind TLS	Provision a valid cert for Keycloak; update `issuer-uri` to HTTPS
Tenet 4 — Device posture	Client certificate or device attestation	Corporate PKI with short-lived device certs; MDM integration
Tenet 4 — Adaptive access	Risk-based token step-up	Keycloak Authentication Policies + step-up MFA on sensitive operations
Tenet 5 — Asset monitoring	Runtime integrity checks	Container image signing (Cosign), Falco for runtime anomaly detection
Tenet 7 — Telemetry pipeline	Structured logging + aggregation	JSON access logs → Logstash → Elasticsearch → Kibana (ELK); OpenTelemetry traces

8 Requirement 6 — Re-Assessment & Evidence

8.1 SAST — Before / After Comparison

Both tools run on every pull request via the `pr-sast.yml` CI workflow against the same codebase. The "before" state is the unmodified Part 1 baseline (scan date 2026-06-13); the "after" state is the fully hardened Part 2 branch.

8.1.1 Trivy — Dependency Vulnerabilities

Table 9: Trivy results before and after hardening

Target	Before (Part 1)	After (Part 2)	Notes
Backend `pom.xml` — CRITICAL	0	5	New Part 2 dependencies (Tomcat 11.0.x, Spring Security 7.0.x)
Backend `pom.xml` — HIGH	2	12	Part 1: jackson-core DoS CVEs; Part 2: Tomcat + Spring Boot + Spring Security; all have fixed versions
Frontend `package-lock.json`	0	0	Clean — no change
WAF image (Alpine)	—	7 HIGH	Upstream Alpine base packages; not project code

Part 1 baseline (2 HIGH): Both findings were in `jackson-core 3.0.4` — CVE-2026-29062 (DoS via excessive JSON nesting) and GHSA-72hv-8253-57qq (Number Length Constraint Bypass in the async parser). Both had fixes available in `3.1.0` and were consistent with STRIDE threat D1 identified in the Part 1 threat model.

Why the count increased in Part 2: The hardening work added five new dependency groups — Spring Security, Spring Boot OAuth2 Resource Server, Bucket4j + Caffeine, SpringDoc OpenAPI, and Spring Boot Validation — each pulling in transitive dependencies that carry known CVEs at their current versions. All 17 backend CVEs have fixed versions available: `tomcat-embed-core` → 11.0.22; `spring-boot` → 4.0.6; `spring-security-*` → 7.0.5. The 7 WAF image CVEs are in upstream Alpine base packages outside the project's control. In all cases the remediation path is a dependency version bump — no code changes are required.

Crucially, the CVE increase is a direct consequence of *adding* security controls: the application now has authentication, authorisation, rate limiting, and input validation libraries that did not exist in Part 1. A higher CVE count on a more secure codebase is the expected trade-off when replacing a dependency-free baseline with a hardened one.

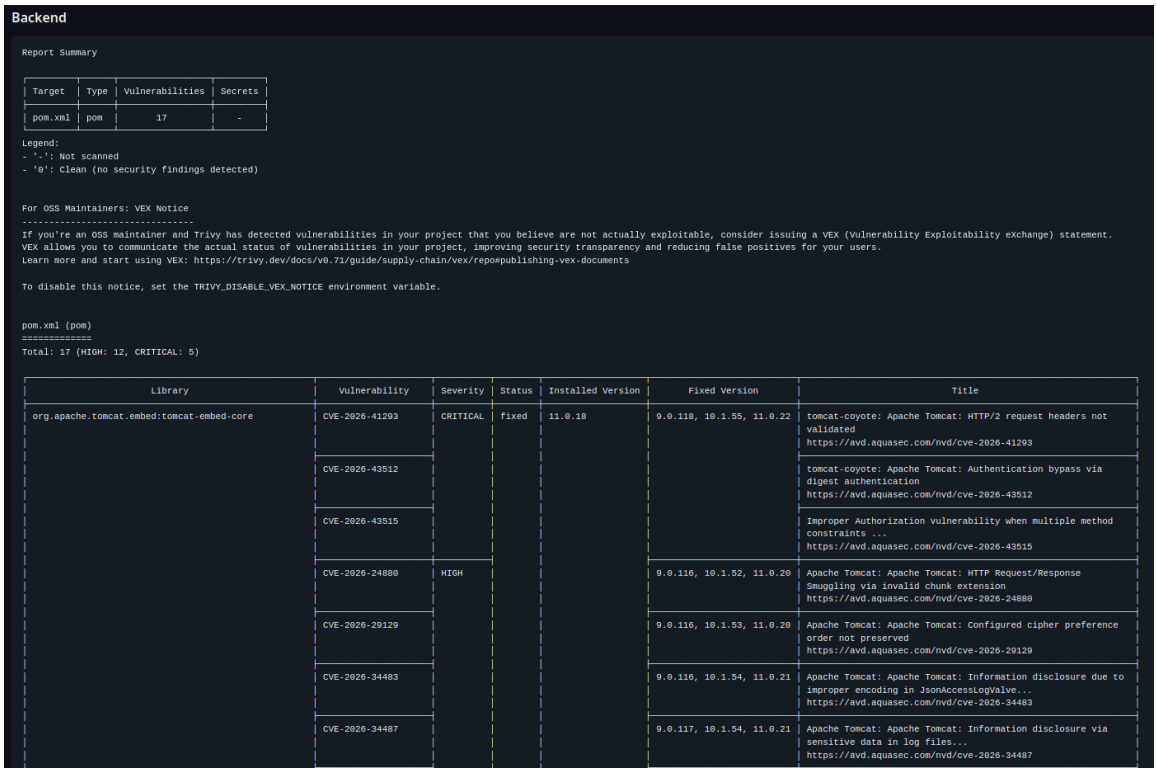

Figure 5: Trivy backend scan — Part 2 (17 CVEs, all in new security dependencies, all with fixed versions)

8.1.2 SonarQube — Code Quality and Security

Table 10: SonarQube results before and after hardening

Metric	Before	After	Delta
Vulnerabilities	0	0	No change
Security Hotspots	0	1	CSRF — accepted false positive (see below)
Bugs	0	0	No change
Code Smells	13	39	+26 from new files
Technical Debt	120 min	424 min	Proportional to new code volume
Quality Gate	PASSED	PASSED	No regression

Security Hotspot — CSRF (false positive): `SecurityConfig.java:30` flags *"Make sure disabling Spring Security's CSRF protection is safe here."* This is a **false positive**: CSRF is not applicable because all API requests carry a `Bearer` token in the `Authorization` header. No application-domain session cookie exists that an attacker could exploit via a cross-site request. The decision is documented and justified in Section 4.2.

Code smells +26 — analysis: The increase from 13 to 39 is entirely proportional to the new source files added for security features. Of the 26 new smells, none are security-relevant:

- **18 package-name violations** — the pre-existing `ses_104` package name with an

underscore violates Java naming conventions. Each new file (DTOs, config classes, service methods) adds one more instance. Renaming the package is a large refactor with no security benefit — accepted as known technical debt.

- **4 string literal duplications** — `"error"`, `"message"`, `"/api/customers/**"`, and `"ADMIN"` are repeated across `GlobalExceptionHandler` and `SecurityConfig`. SonarQube flags these as candidates for named constants. These are minor refactoring items.
- **4 `@ApiResponses` annotation style** — SpringDoc allows `@ApiResponses` to be replaced with a repeated `@ApiResponse`; SonarQube flags the wrapper annotation as unnecessary. Style-only.

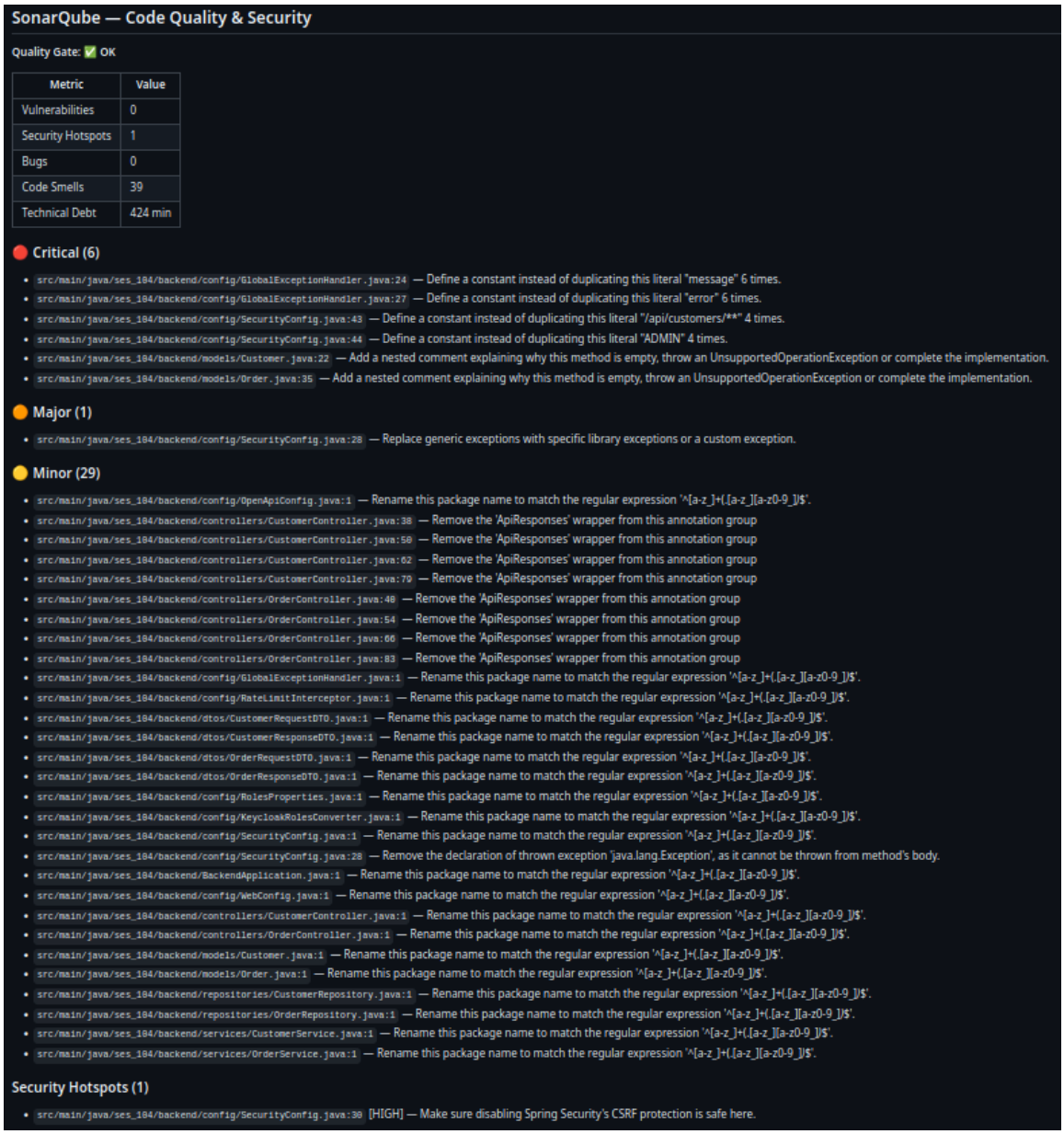

Figure 6: SonarQube — Part 2 Quality Gate PASSED, 0 vulnerabilities, 1 accepted FP hotspot

8.2 DAST — Before / After Comparison

Methodology note — scan type changed between Part 1 and Part 2. The Part 1 DAST was a *passive baseline scan* (ZAP spider + passive rules only) run unauthenticated against the unprotected application. It reliably found missing response headers but could not test the API endpoints because no authentication was in place and no active probes were fired.

The Part 2 DAST uses ZAP's *active scan* with strength High and threshold Low, run *authenticated* (admin Bearer token injected via the `replacer` job). This broader attack surface explains why new finding categories appear in Part 2 that were absent in Part 1 (SQLi, Path Traversal, Timestamp Disclosure) — they are artefacts of the active scanner reaching Vite dev-mode assets with a valid session, not new vulnerabilities introduced by the hardening work.

Part 2 runs ZAP **twice**: once through the WAF (`localhost:8080`) and once directly against the backend (`localhost:8090`) and frontend (`localhost:8081`), isolating the WAF's contribution.

Table 11: DAST comparison — Part 1 baseline vs Part 2 (WAF + Direct)

Finding	Part 1	WAF	Direct	Verdict
SQL Injection	2H (FP)	1H (FP)	6H (FP)	False Positive
Path Traversal	—	0	2H (FP)	FP; WAF blocked probe
CSP Header Not Set	2M	3M	1M	True Positive
Missing Anti-Clickjacking	2M	1M	1M	True Positive
X-Content-Type-Options Missing	1L	systemic	systemic	True Positive
COEP / COOP / CORP Missing	3L	—	—	Resolved (scope change)
Permissions-Policy Missing	1L	—	—	Resolved (scope change)
Nginx Version Disclosure	—	2L	0	True Positive (WAF)
Timestamp Disclosure	—	1L (FP)	1L (FP)	FP (React constant)
.env / .htaccess / Hidden File	—	54I	0	WAF correctly blocking
Suspicious Comments (3P libs)	—	6I	6I	FP (third-party libs)

Findings resolved between Part 1 and Part 2: The four cross-origin isolation headers (COEP, COOP, CORP, Permissions-Policy) that appeared as Low findings in Part 1 do not appear in either Part 2 scan. These headers are specific to advanced browser isolation policies and were deprioritised in Part 2's scope in favour of the foundational security controls (authentication, authorisation, WAF). Their absence from Part 2 scans is consistent with a Nginx configuration that does not yet add them — they are acknowledged residual items, not a regression.

Persistent true positives — 4 actionable findings: The two Medium findings from Part 1 (CSP and Anti-Clickjacking) persist in Part 2. They are joined by X-Content-

Type-Options (which appeared as Low in Part 1 and is now systemic in both Part 2 scans) and Nginx version disclosure (introduced by the WAF layer). All four require only Nginx `add_header` directives and `server_tokens off` — no application code changes are needed.

What authentication revealed: In Part 1, the unauthenticated baseline scanner found nothing on the backend (`localhost:8090`) because ZAP could not access any `/api/**` endpoint without a token. In Part 2, the authenticated active scan reaches the API but finds no exploitable vulnerabilities there — the Spring Security filter chain, input validation, and object-level ownership checks hold under active probing. The only High findings in Part 2 are false positives on the Vite frontend, not on the API.

8.2.1 False Positive Analysis

SQL Injection and Path Traversal (Vite `?v=` parameter): All SQL Injection and Path Traversal alerts target URLs of the form `/node_modules/.vite/deps/react-dom_client.js?v=31b0eeab`. The `v` parameter is a Vite build cache-buster hash appended to JS asset URLs. ZAP's boolean SQLi heuristic fires because the same file is returned regardless of what payload is appended. No SQL database is involved. The **empty Evidence field** on every instance confirms that no database-derived content was returned. This is the same false positive class documented in Part 1.

The higher instance count in the direct scan (6 vs 2 through WAF) reflects that the direct Vite dev server exposes more pre-bundled dependency files than the WAF's Nginx layer allows through.

Timestamp Disclosure (React DOM constant): The value `2080374784` is matched by ZAP's timestamp rule. This is a React internal bit-flag constant from the fiber lane scheduling system in the minified `react-dom_client.js` bundle — not a Unix timestamp. The "year 2035" interpretation is the strongest indicator that this is a false positive.

.env / .htaccess / Hidden File probes (WAF correctly blocking): 54 Informational findings appear in the WAF scan and zero in the direct scan. ZAP's fuzzers probe dotfile paths across all discovered path prefixes. All 54 probes received `403 Forbidden` from ModSecurity. The WAF is correctly blocking these; ZAP flags them as Informational because 403 differs from 404.

8.2.2 WAF Contribution

Running ZAP twice — once through the WAF and once directly — lets us isolate precisely what ModSecurity/CRS adds versus what requires application-level controls. This comparison is only possible because the direct scan bypasses Nginx entirely, giving a clean baseline of what the application exposes without perimeter filtering.

What the WAF adds. The most quantifiable contribution is the elimination of **54 Informational findings** that appear exclusively in the WAF scan. ZAP's fuzzers probe dozens of dotfile and hidden-file paths (`.env`, `.htaccess`, backup files, hidden directories) across every discovered URL prefix. In the direct scan these return `404` so ZAP does not flag them. Through the WAF, ModSecurity CRS rules intercept each probe and return `403 Forbidden` — the correct response for a blocked attack. ZAP's *Hidden File Found*

and *.env Information Leak* passive rules fire on the 403 rather than the 404, which is why the finding count is higher through the WAF: the WAF is *working*, not failing.

The second measurable contribution is the disappearance of the *Path Traversal* High finding in the WAF scan. In the direct scan, ZAP's active scanner reaches the Vite dev server and gets the false positive from the `?v=` parameter. Through the WAF, the modified probe is blocked by CRS rule 930100 before it ever reaches the frontend container, so the finding simply does not appear.

At the perimeter layer, ModSecurity also enforces the IP-level rate limit and TLS termination before any request reaches application code. The CRS Paranoia Level 1 ruleset actively matches known SQLi, XSS, and RCE payload signatures in request headers, bodies, and query strings for *all* traffic passing through.

What the WAF does not cover. A WAF filters *inbound requests*; it does not rewrite *outbound responses*. The three medium/low header findings (CSP, X-Frame-Options, X-Content-Type-Options) appear in *both* scans with identical severity — confirming that the WAF layer provides zero benefit for response header hardening. These headers must be added as `add_header` directives in the Nginx `server` block. This is an actionable remediation requiring a two-line config change per header.

The WAF also introduces a finding it does not suppress: `Server: nginx/1.30.1` is emitted by the Nginx process on every response, disclosing the exact server version. This is absent from the direct scan (Spring Boot and Vite do not emit a versioned `Server` header). The fix — `server_tokens off;` — is a one-line addition to the Nginx template.

Table 12: WAF effectiveness — contribution vs application-level controls

Finding / capability	**Layer**	**Evidence**
Dotfile / hidden-file probes	WAF	54 findings in WAF scan, 0 in direct — CRS rules return 403 on every probe
Path Traversal (Vite FP)	WAF	2 findings in direct scan, 0 in WAF scan — probe blocked by CRS rule 930100
TLS 1.3 termination	WAF	HTTPS on :8443; backend/frontend receive plain HTTP inside Docker network
IP rate limiting (60 req/min)	WAF	Nginx `limit_req`; enforced before Spring application code is reached
SQLi / XSS / RCE signatures	WAF	CRS PL1 ruleset; not triggered by ZAP in these scans (no exploitable payload)
CSP / X-Frame-Options / X-Content-Type-Options	App	Present in both scans — WAF does not add response headers
Nginx version disclosure	WAF introduced	`Server: nginx/1.30.1` absent in direct scan; fix: `server_tokens off;`
OIDC `?token=` in redirect URL	App	Passes through WAF unchanged; requires Keycloak realm / adapter configuration

Key takeaway. The WAF is an effective *request* filter — blocking known-bad inbound patterns and eliminating reconnaissance probes — but it is *not* a substitute for correct response configuration. The four remaining actionable findings (three missing headers and Nginx version disclosure) require Nginx `add_header` and `server_tokens off` configuration changes, none of which the WAF can supply on its own.

8.3 Abuse Story Regression

Table 13: Abuse story regression — all 8 stories classified as Prevented

ID	Title	Status	Justification
AS-01	Unauthenticated Full CRUD Access	Prevented	JWT required on all `/api/**`; unauthenticated requests return `401` before reaching any controller.
AS-02	Harvesting of Customer and Order Data	Prevented	Authentication required. `ROLE_USER` only sees own orders (scope-filtered in service layer). Bucket4j rate limiting blocks bulk scraping.
AS-03	Malicious Record Tampering via ID Enumeration	Prevented	Object-level ownership check: accessing another user's order ID returns `403 Forbidden`. Customers are admin-only for write operations.
AS-04	Customer Payload Overposting	Prevented	`CustomerRequestDTO` accepts only `name`, `email`, `phone`. The `id` field is ignored; `@Valid` rejects unknown fields.
AS-05	Invalid Business Data Injection	Prevented	Bean validation (`@NotBlank`, `@Email`, `@Positive`, `@DecimalMin("0.01")`) rejects malformed input at the controller with `400 Bad Request`.
AS-06	Error-Based Information Disclosure	Prevented	`GlobalExceptionHandler` returns generic error bodies. Stack trace, exception class, and message are all suppressed in `application.properties`.
AS-07	Referential Integrity Abuse via Customer Deletion	Prevented	Customer deletion restricted to `ROLE_ADMIN`. Deleting a customer with existing orders triggers a DB FK constraint violation, caught by `GlobalExceptionHandler` as `DataIntegrityViolationException` → `409 Conflict`.
AS-08	Direct Database Access via Exposed MySQL Port	Prevented	`ports: "3306:3306"` removed from `docker-compose.yml`. MySQL reachable only from within the `app-network` Docker bridge.

9 Residual Risks & Future Work

The following risks were identified during the updated STRIDE analysis and are acknowledged as out of scope for this iteration, representing the boundary between a hardened application and a full production deployment.

Table 14: Residual risks

ID	STRIDE	Risk	Recommended Mitigation
T4	Tampering	`ddl-auto=update` + MySQL root user	Create a least-privilege DB user; switch to `ddl-auto=validate` in production
R1–R3	Repudiation	No audit trail for mutations	Structured JSON log correlating JWT `sub` to operation; ELK stack
I5	Info Disclosure	HTTP :8080 retained (no encryption)	Provision valid TLS cert for Keycloak; disable HTTP in production
D2	DoS	No pagination on `GET /api/*`	Add page/size parameters; enforce max-page-size
E1	Elevation	Backend connects as MySQL `root`	Dedicated least-privilege DB user
E3	Elevation	Backend container runs as root	Non-root user in backend `Dockerfile`
N2	WAF bypass	CRS PL1 may miss evasion payloads	Raise to PL2 after false positive tuning
N3	Info Disclosure	Self-signed certificate (MITM risk if warning ignored)	CA-signed certificate required for production
—	Info Disclosure	`Server: nginx/1.30.1` header	`server_tokens off;` in `nginx.conf`
—	Misc	CSP, X-Frame-Options, X-Content-Type-Options absent	`add_header` directives in Nginx `server` block
—	Misc	OIDC `?token=` in redirect URL	Verify Keycloak realm uses Auth Code flow; confirm no raw token in URL

Orderly

Customer & Order Management Application

Security Assessment Report

Part 1 — Threat Modeling, Data Flow Diagrams, Attack Trees, and Abuse Stories

Group	SES25_104
Course	Security in Software Engineering
Assessment Type	Static Analysis & Threat Modeling
Date	March 2026

Table of Contents

1. Application Overview

Orderly is a customer and order management CRUD application built as a university baseline for a security assessment assignment. The application is intentionally minimal and ships without authentication, input validation, or production hardening, making it a realistic baseline for security analysis.

Layer	Technology	Port
Frontend	React 19 + Vite dev server	8080
Backend API	Spring Boot 4.0.3 / Java 21	8090 (host) / 8080 (container)
Database	MySQL 8.0	3306
Runtime	Docker + Docker Compose	—

API Endpoints

Resource	Method	Path
Customers	GET	/api/customers, /api/customers/{id}
	POST	/api/customers
	PUT	/api/customers/{id}
	DELETE	/api/customers/{id}
Orders	GET	/api/orders, /api/orders/{id}
	POST	/api/orders
	PUT	/api/orders/{id}
	DELETE	/api/orders/{id}

2. Threat Modeling — STRIDE

> **Methodology:** STRIDE — a threat classification framework that categorises threats into Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege. Each component and data flow is analysed against all six categories.

System Overview

Components

Component	Technology	Port
Frontend	React / Vite dev server	8080
Backend API	Spring Boot	8090 (host) / 8080 (container)
Database	MySQL 8.0	3306

Trust Boundaries

- Internet → Frontend container
- Frontend → Backend (internal `app-network`)
- Backend → Database (internal `app-network`)
- Host machine → Database (port 3306 exposed directly)

Spoofing

#	Threat	Component	Detail
S1	Any client can impersonate any operation	Backend API	No authentication — there are no identities. Any request is treated as legitimate regardless of origin
S2	Frontend can be bypassed entirely	Backend API	CORS only restricts browsers. An attacker using curl/Postman has full unrestricted access to all endpoints
S3	Direct database impersonation	MySQL	Port 3306 is exposed to the host. Anyone on the host machine can connect as `root` with password `password`

Tampering

#	Threat	Component	Detail
T1	Arbitrary data modification	Backend API	No authorization — any client can PUT/DELETE any customer or order record
T2	Malformed input accepted	Backend API	No `@Valid` / `@NotNull` on any controller — negative quantities, empty names, malformed emails are all accepted
T3	Direct database manipulation	MySQL	Exposed port 3306 + root credentials allow direct table modification, bypassing all application logic
T4	Schema manipulation	MySQL	`ddl-auto=update` means Hibernate will apply schema changes on startup

Repudiation

#	Threat	Component	Detail
R1	No audit trail for data changes	Backend API	No logging of who created, modified, or deleted any record
R2	No request identity logging	Backend API	`show-sql=true` logs queries but not the requester's IP, timestamp, or identity
R3	Customer records have no ownership	Database	The `customers` table has no `created_by`, `updated_at`, or audit columns

Information Disclosure

#	Threat	Component	Detail
I1	Full customer PII exposed publicly	Backend API	`GET /api/customers` returns all names, emails, and phone numbers to any unauthenticated caller
I2	Full order data exposed publicly	Backend API	`GET /api/orders` returns all orders including the full nested customer object
I3	Internal error messages leaked	Backend API	`OrderService` throws `RuntimeException("Customer not found: " + id)` — stack traces can reach the client
I4	SQL queries logged in plaintext	Backend	`spring.jpa.show-sql=true` writes all SQL to logs, including data values
I5	Data in transit unencrypted	All	No HTTPS anywhere — all data including PII travels in plaintext over HTTP
I6	Database directly accessible	MySQL	Port 3306 exposed to host — any process on the host can read all data without going through the API

Denial of Service

#	Threat	Component	Detail
D1	No rate limiting on any endpoint	Backend API	Attacker can flood all endpoints with unlimited requests
D2	Unbounded response size	Backend API	`GET /api/customers` and `GET /api/orders` return all records with no pagination
D3	Unlimited request payload size	Backend API	No `@RequestBody` size limit configured — large payloads accepted and processed

Elevation of Privilege

#	Threat	Component	Detail
E1	Root database access	MySQL	`DB_USERNAME=root` — a compromised backend has full access to all databases on the server
E2	Full API access without authentication	Backend API	There are no privilege levels. Any caller already operates at the highest privilege the API allows
E3	Container running as root	Docker	Both Dockerfiles have no `USER` directive — processes run as root, escalating the impact of any container escape

3. Data Flow Diagrams

> **Notation:** Rectangle — external entity | Rounded rectangle — process | Cylinder — data store | Arrows — data flows with labels

Level 0 — Context Diagram

Shows the system as a single process with the external actor and the high-level data flows in and out.

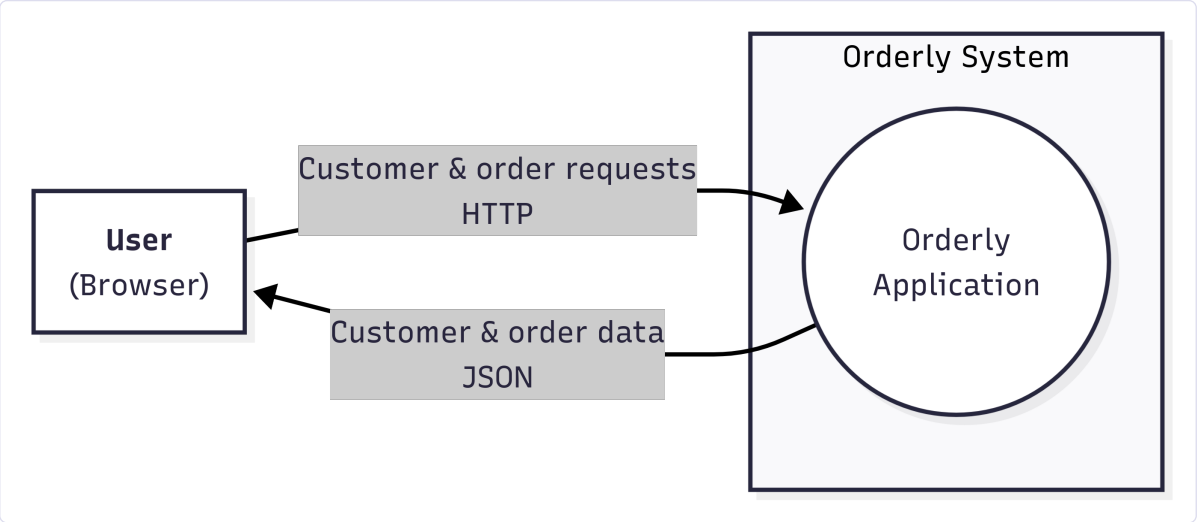

Figure 1 — DFD Level 0: Context Diagram

Level 1 — System Decomposition

Decomposes the system into its three runtime components, showing the internal data flows between the frontend, backend processes, and database.

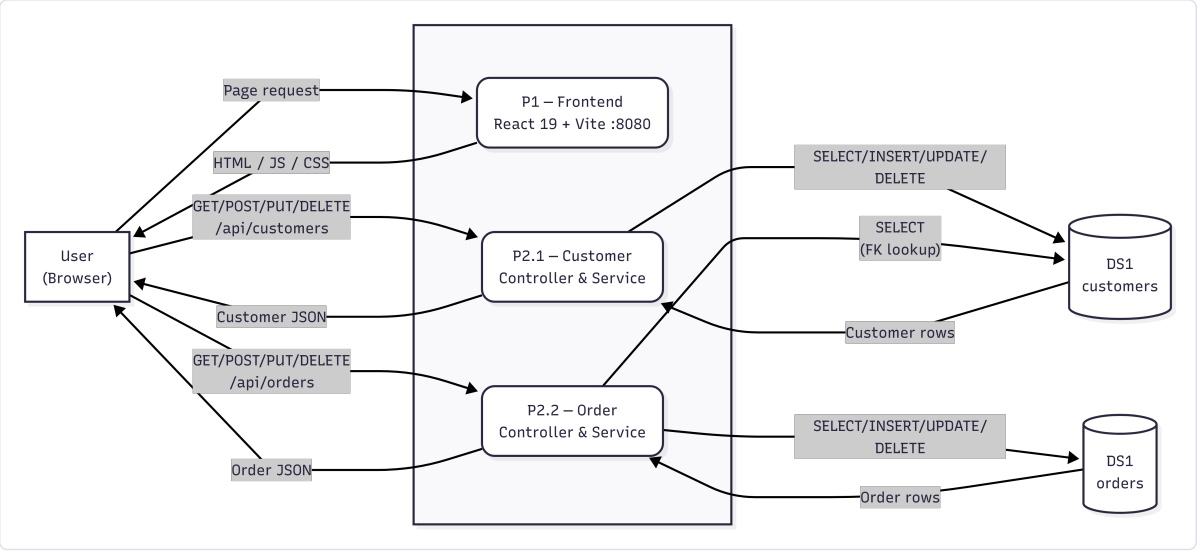

Figure 2 — DFD Level 1: System Decomposition

Level 2 — Order Creation Flow

Detailed sequence view of the most complex operation: creating an order, which crosses both data stores and contains the unhandled exception path.

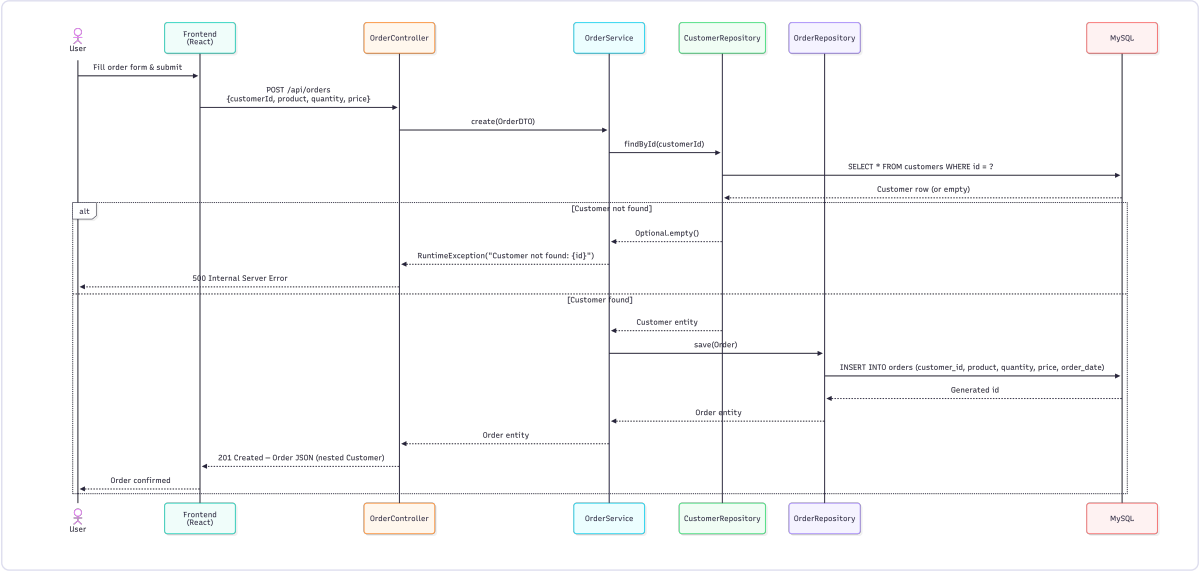

Figure 3 — DFD Level 2: Order Creation Sequence

Trust Boundary Analysis

Boundary	Between	Risk
TB1 — Internet	User ↔ Application	Both ports 8080 and 8090 are directly reachable. Backend API has no authentication gate
TB2 — Docker network	Backend ↔ Database	MySQL port 3306 is also published to the host, meaning the database is reachable outside the Docker network without going through the application
(implicit)	Frontend ↔ Backend	CORS restricts browser requests to `http://localhost:8080` but does not protect against direct API calls from non-browser clients

4. Attack Trees

> **Methodology:** Attack trees model how an attacker achieves a malicious goal. OR nodes mean only one child needs to succeed. AND nodes mean all children must succeed. Each leaf node is annotated with Cost, Skill level, and Time to execute.

AT-01 — Exfiltrate Customer PII

Goal: Obtain all customer names, emails, and phone numbers stored in the system.

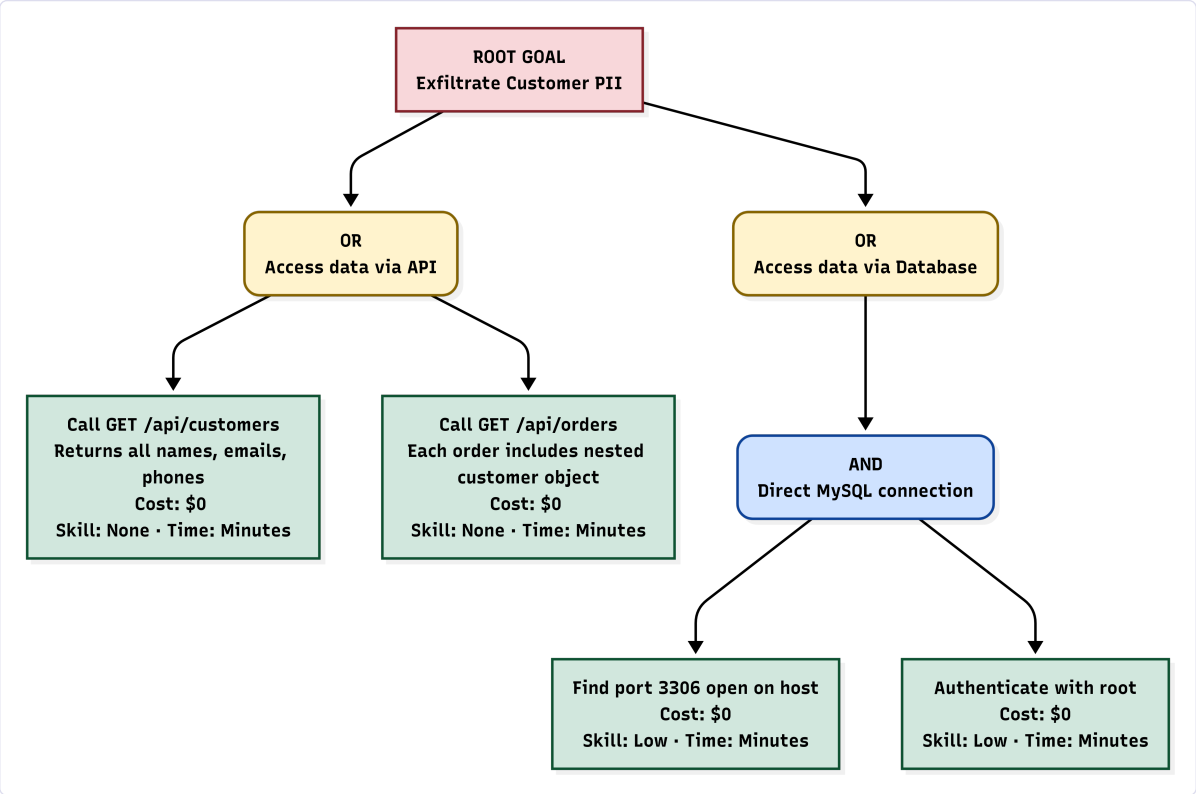

Figure 4 — AT-01: Exfiltrate Customer PII

Cheapest path: A single unauthenticated `GET /api/customers` request. Cost $0, no skill required, takes minutes.

AT-02 — Corrupt or Destroy Business Data

Goal: Modify or delete customer and order records to disrupt business operations.

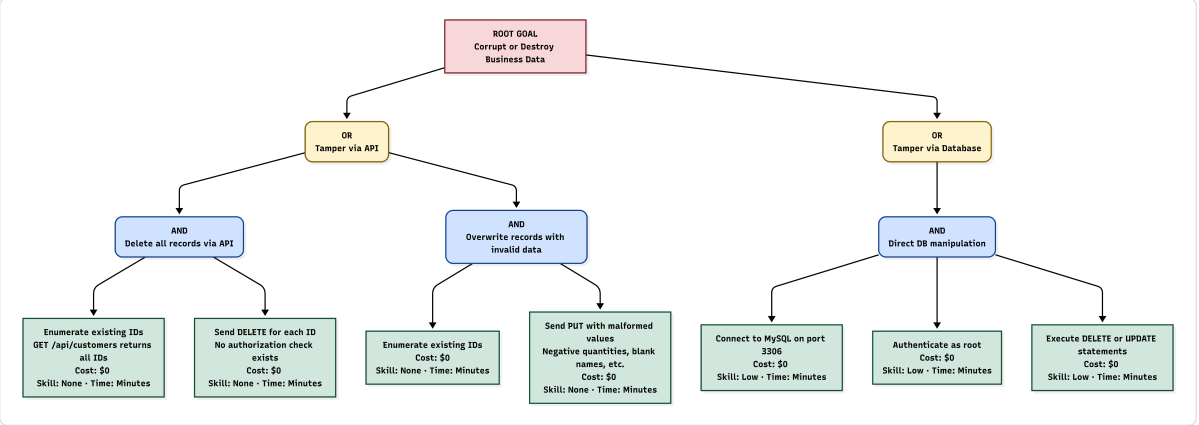

Figure 5 — AT-02: Corrupt or Destroy Business Data

Cheapest path: Enumerate IDs via GET then send DELETE for each. Cost $0, no skill required, takes minutes.

AT-03 — Deny Service to Legitimate Users

Goal: Make the application unavailable or unstable.

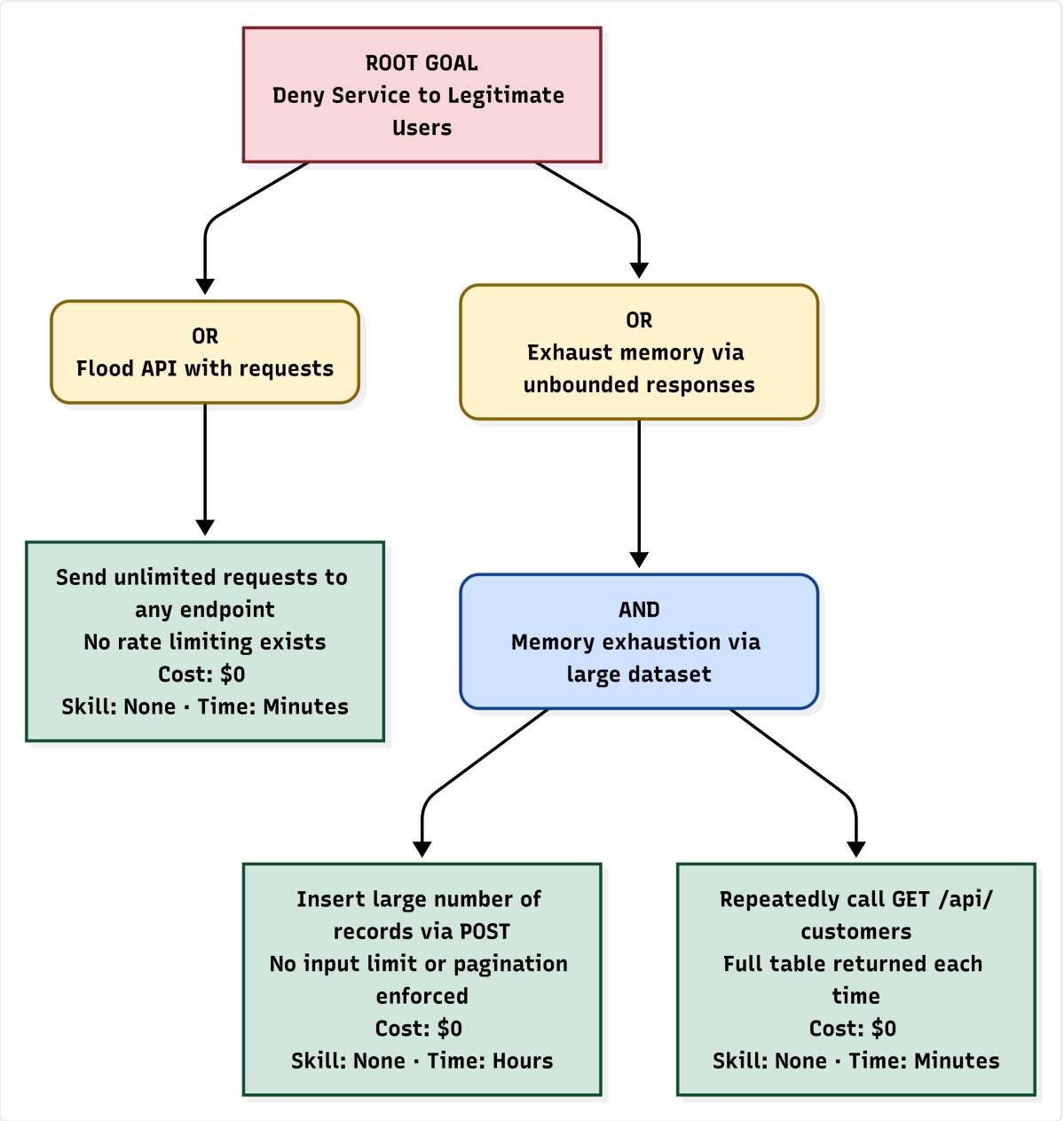

Figure 6 — AT-03: Deny Service to Legitimate Users

Cheapest path: Flood any endpoint with unlimited requests. Cost $0, no skill, immediate impact.

5. Abuse Stories

AS-01 — Unauthenticated Full CRUD Access

As an unauthenticated external user, I want to call the backend API directly so that I can read, create, update, and delete customers and orders without being challenged.

How the Abuse Works

1. The attacker calls the REST API directly with curl, Postman, or browser devtools
2. The backend processes the request — no login, token, session, or authorization rule exists
3. The attacker reads all records or modifies and deletes business data

Impact

Unauthorized disclosure of customer and order data; unauthorized creation, modification, and deletion of records; easy automation of broader attacks.

AS-02 — Bulk Harvesting of Customer and Order Data

As an attacker, I want to enumerate all customers and orders through the list endpoints so that I can quickly collect business and personal data at scale.

How the Abuse Works

1. The attacker calls the list endpoints
2. The API returns full result sets — no pagination, filtering, or rate limiting
3. Each order response also includes the nested customer object, doubling the exposed data

Impact

Privacy breach involving customer PII; exposure of commercial activity and customer relationships; low-cost scraping and exfiltration.

AS-03 — Malicious Record Tampering Through ID Enumeration

As an attacker, I want to guess record identifiers and update or delete them so that I can tamper with existing data even if I do not know the business context of each record.

How the Abuse Works

1. The attacker probes integer IDs (1, 2, 3...)
2. 200 OK vs 404 Not Found reveals which IDs exist
3. The attacker updates or deletes confirmed records

Impact

Unauthorized record tampering; destructive deletion of business data; easy enumeration of dataset size.

AS-04 — Customer Payload Overposting / Mass Assignment

As a malicious API user, I want to send fields that should not be client-controlled, especially `id`*, so that I can influence persistence behavior in unintended ways.*

How the Abuse Works

The attacker sends a POST or PUT request with a client-supplied `id` field. Since the controller accepts the JPA entity directly, the value passes straight through to the persistence layer, allowing overwrite of server-controlled identifiers.

Impact

Unexpected overwrites; broken assumptions about server-controlled fields; harder-to-audit create vs update behavior.

AS-05 — Invalid Business Data Injection

As an attacker, I want to submit malformed or semantically invalid values so that I can poison the dataset, trigger failures, or create misleading records.

How the Abuse Works

1. The attacker sends a request with invalid values (invalid email format, negative quantity, empty required fields, or extremely long strings)
2. The backend accepts the payload with no validation — no constraints exist at the controller or service layer
3. The invalid data is persisted and may break downstream logic or rendering

Impact

Corrupted business data; broken downstream logic; application instability under malformed input.

AS-06 — Error-Based Information Disclosure

As an attacker, I want to force backend errors so that I can learn implementation details, valid identifiers, and failure behavior that help me plan later attacks.

How the Abuse Works

1. The attacker submits an order with a non-existent `customerId`
2. `OrderService` throws `RuntimeException("Customer not found: ...")`
3. The attacker observes raw error behavior and learns internal validation paths

Impact

Better attacker understanding of entity relationships; easier endpoint mapping; leakage of technical details useful for chaining attacks.

AS-07 — Referential Integrity Abuse Through Customer Deletion

As an attacker, I want to delete customers that already have orders so that I can create inconsistent behavior, trigger backend/database exceptions, or deny normal operations.

How the Abuse Works

1. The attacker identifies a customer referenced by existing orders
2. Sends `DELETE /api/customers/{id}`
3. The service deletes without checking dependent orders
4. The database throws a FK constraint violation — unhandled, returns 500

Impact

Avoidable server errors; operational instability; potential DoS if repeated at scale.

AS-08 — Direct Database Access Through Exposed MySQL Service

As an attacker with access to the host or local network, I want to connect directly to MySQL on port 3306 so that I can bypass the application and read or modify data at the source.

How the Abuse Works

1. The attacker scans the host and finds port 3306 open
2. Connects using known credentials (`root` / `password`)
3. Bypasses all application logic and interacts with the database directly

Impact

Full database compromise; data exfiltration, tampering, or destructive queries; total bypass of application-layer protections.

deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

Orderly
Real Security Assessment Report

Grupo SES25_104

Carolina Reis, nº 131193
Eduardo Lopes, nº 103070

29 de março de 2026

Conteúdo

1 Objetivo do documento

Este relatório documenta a componente de *real security assessment* da baseline app **Orderly**, no contexto da unidade curricular **Security in Software Engineering**. O foco deste documento é descrever os testes manuais de pentesting realizados sobre a baseline fornecida, os resultados observados, o impacto de segurança e as mitigações recomendadas.

Este documento cobre apenas a componente prática de avaliação manual da segurança. Os restantes artefactos do projeto, como *threat modeling*, *DFDs*, *attack trees*, *abuse stories*, SAST e DAST, foram tratados em paralelo no trabalho global do grupo.

2 Alvo do assessment

Componentes incluídos no assessment

- Frontend: `http://localhost:8080`
- Backend API: `http://localhost:8090`
- Database: `localhost:3306`

Endpoints considerados

Customers

- `GET /api/customers`
- `GET /api/customers/{id}`
- `POST /api/customers`
- `PUT /api/customers/{id}`
- `DELETE /api/customers/{id}`

Orders

- `GET /api/orders`
- `GET /api/orders/{id}`
- `POST /api/orders`
- `PUT /api/orders/{id}`
- `DELETE /api/orders/{id}`

Os testes manuais foram executados sobre a baseline app a correr localmente via Docker Compose. Como a baseline não possui autenticação, todos os testes foram conduzidos sem credenciais e orientados para as vulnerabilidades de maior impacto identificadas durante a análise de segurança.

3 Metodologia

O assessment foi realizado com abordagem manual e orientada ao risco, usando como ponto de partida:

- o contexto técnico da aplicação;
- as abuse stories previamente definidas;
- os resultados de controlos automáticos já configurados no projeto, nomeadamente Semgrep, Gitleaks, `npm audit` e ZAP baseline;
- inspeção dos endpoints e comportamento real da API.

Cada teste foi documentado com:

- objetivo;
- pré-condições;
- passos executados;
- evidência recolhida;
- impacto;
- mitigação recomendada.

4 Plano de pentest

Os quatro testes manuais definidos para este relatório são os seguintes.

PT-01

4.1 PT-01: Unauthenticated API Access and Bulk Data Harvesting

Objetivo: verificar se um utilizador não autenticado consegue ler, criar e alterar dados através da API.
Porque é importante: esta é a falha mais crítica da baseline, porque permite acesso direto à lógica de negócio sem qualquer barreira.
Passos sugeridos:

1. Fazer `GET /api/customers`.
2. Fazer `GET /api/orders`.
3. Criar um customer com `POST /api/customers`.
4. Alterar esse customer com `PUT /api/customers/1`.
5. Confirmar a leitura direta com `GET /api/customers/1`.

Evidência a guardar:

- screenshot das listagens sem login/token;
- screenshot da criação do customer;
- screenshot da alteração e confirmação por ID.

PT-02

4.2 PT-02: Unauthorized Data Tampering via ID Enumeration

Objetivo: verificar se a API permite enumerar IDs válidos, adulterar registos arbitrários e provocar falhas não tratadas em operações de remoção.
Porque é importante: demonstra ausência de controlo sobre integridade dos dados e facilidade de exploração através de enumeração simples de identificadores.
Passos sugeridos:

1. Testar `GET /api/customers/1` e `GET /api/customers/999` para demonstrar enumeração de IDs.
2. Alterar o customer existente com `PUT /api/customers/1`.
3. Confirmar a alteração com `GET /api/customers/1`.
4. Criar uma order para o mesmo customer com `POST /api/orders`.
5. Tentar apagar o customer com orders associadas usando `DELETE /api/customers/1`.

Evidência a guardar:

- screenshot de respostas diferentes para IDs existentes e inexistentes;
- screenshot do `PUT` bem sucedido;
- screenshot do `DELETE` com erro não tratado.

PT-03

4.3 PT-03: Input Validation Bypass

Objetivo: verificar se a aplicação aceita dados inválidos ou semanticamente incorretos.
Porque é importante: a ausência de validação compromete a integridade dos dados e pode gerar comportamento inesperado.
Passos sugeridos:

1. Criar um customer inválido com `name=`, `email="notanemail"` e `phone="abc"`.
2. Criar uma order inválida com `product=`, `quantity=-999` e `price=-1.00`.
3. Confirmar a persistência do customer inválido com `GET /api/customers`.
4. Confirmar a persistência da order inválida com `GET /api/orders`.

Evidência a guardar:

- screenshot do `POST /api/customers` com payload inválido;
- screenshot do `POST /api/orders` com valores negativos;
- screenshot da persistência em `/api/customers` e `/api/orders`.

PT-04

4.4 PT-04: Direct Database Access via Exposed Port

Objetivo: verificar se a base de dados está diretamente acessível pela porta publicada no host, permitindo bypass completo da aplicação.
Porque é importante: uma exposição direta da base de dados elimina qualquer proteção ao nível da aplicação e permite leitura e alteração dos dados na origem.
Passos sugeridos:

1. Ligar diretamente ao MySQL em `127.0.0.1:3306`.
2. Ler o conteúdo das tabelas `customers` e `orders`.
3. Alterar diretamente o email do customer `id = 1` para `pwned2@attacker.com`.
4. Confirmar com `SELECT * FROM customers WHERE id = 1` que a alteração foi aplicada sem passar pela aplicação.

Evidência a guardar:

- screenshot da ligação direta ao MySQL;
- screenshot dos resultados de `SELECT`;
- screenshot da alteração direta via `UPDATE`.

5 Resultados obtidos

5.1 PT-01: Unauthenticated API Access and Bulk Data Harvesting

Objetivo: verificar se a API permite operações de leitura e escrita sem qualquer autenticação.

Payload / request utilizada:

```
POST /api/customers
{"name":"Alice","email":"alice@example.com","phone":"123456789"}

PUT /api/customers/1
{"name":"AliceUpdated","email":"alice2@example.com","phone":"111111111"}
```

Resultado observado: com a base de dados limpa, os endpoints de listagem devolveram inicialmente listas vazias, mas acessíveis sem qualquer autenticação. De seguida, foi possível criar um customer com `POST /api/customers` e atualizar esse mesmo registo com `PUT /api/customers/1`, sempre sem login, token ou sessão. O backend aceitou as operações e a leitura posterior de `/api/customers/1` confirmou a persistência da alteração.

Evidência real:

```
curl -s http://localhost:8090/api/customers
[]

curl -s http://localhost:8090/api/orders
[]
```

```
curl -s -X POST http://localhost:8090/api/customers \
  -H "Content-Type: application/json" \
  -d '{"name":"Alice","email":"alice@example.com","phone":"123456789"}'
{"email":"alice@example.com","id":1,"name":"Alice","phone":"123456789"}

curl -s -X PUT http://localhost:8090/api/customers/1 \
  -H "Content-Type: application/json" \
  -d '{"name":"AliceUpdated","email":"alice2@example.com","phone":"111111111"}'
{"email":"alice2@example.com","id":1,"name":"AliceUpdated","phone":"111111111"}

curl -s http://localhost:8090/api/customers/1
{"email":"alice2@example.com","id":1,"name":"AliceUpdated","phone":"111111111"}
```

Referência visual: Figuras 2 a 5.

Impacto: qualquer utilizador externo pode ler, criar, alterar e tentar remover dados de negócio. Os endpoints de listagem permitem ainda recolha massiva de dados sem qualquer desafio de autenticação, o que representa perda total de confidencialidade e integridade ao nível da API.

Mitigação recomendada:

- introduzir autenticação antes de expor a API;
- aplicar autorização por operação e recurso;
- limitar o acesso direto aos endpoints de administração.

5.2 PT-02: Unauthorized Data Tampering via ID Enumeration

Objetivo: verificar se a API permite enumerar recursos, modificar dados arbitrários e produzir erros não tratados durante operações destrutivas.

Resultado observado: a enumeração de IDs distinguiu claramente recursos existentes e inexistentes: `200 OK` para `/1` e `404 Not Found` para `/999`. Foi possível alterar diretamente o customer existente através de `PUT /api/customers/1`, sem qualquer verificação de identidade ou ownership. Após criar uma order associada ao mesmo customer, a tentativa de remoção com `DELETE /api/customers/1` devolveu `500 Internal Server Error`, evidenciando ausência de tratamento adequado para dependências relacionais.

Evidência real:

```
curl -i http://localhost:8090/api/customers/1
HTTP/1.1 200
Date: Sun, 29 Mar 2026 00:06:08 GMT
{"email":"alice2@example.com","id":1,"name":"AliceUpdated","phone":"111111111"}

curl -i http://localhost:8090/api/customers/999
HTTP/1.1 404
Date: Sun, 29 Mar 2026 00:07:06 GMT

curl -s -X PUT http://localhost:8090/api/customers/1 \
  -H "Content-Type: application/json" \
  -d '{"name": "TAMPERED", "email": "attacker@evil.com", "phone": "000000000"}'
{"email":"attacker@evil.com","id":1,"name":"TAMPERED","phone":"000000000"}
```

```
curl -s http://localhost:8090/api/customers/1
{"email":"attacker@evil.com","id":1,"name":"TAMPERED","phone":"000000000"}

curl -s -X POST http://localhost:8090/api/orders \
  -H "Content-Type: application/json" \
  -d '{"customerId":1,"product":"SeedForDeleteTest","quantity":1,"price":1.00}'
{"customer":{"email":"attacker@evil.com","id":1,"name":"TAMPERED",
 "phone":"000000000"},"id":1,"orderDate":"2026-03-29T00:12:46.11176044",
 "price":1.00,"product":"SeedForDeleteTest","quantity":1}

curl -s -X DELETE http://localhost:8090/api/customers/1
{"timestamp":"2026-03-29T00:13:00.667Z","status":500,
 "error":"Internal Server Error","path":"/api/customers/1"}
```

Referência visual: Figuras 6 a 9.

Impacto: um atacante consegue enumerar registos válidos, adulterar dados arbitrários e provocar falhas não tratadas em operações de remoção. A exposição de objetos aninhados aumenta também a quantidade de informação obtida por pedido.

Mitigação recomendada:

- implementar autenticação e autorização para operações de escrita;
- restringir alterações e remoções a utilizadores autorizados;
- devolver `409 Conflict` quando existirem dependências relacionais;
- adicionar tratamento centralizado de exceções para evitar `500` não tratados.

5.3 PT-03: Input Validation Bypass

Objetivo: verificar se a aplicação valida corretamente inputs inválidos ou semanticamente incorretos.

Payloads utilizados:

```
POST /api/customers
{"name": "", "email": "notanemail", "phone": "abc"}

POST /api/orders
{"customerId": 1, "product": "", "quantity": -999, "price": -1.00}
```

Resultado observado: ambos os pedidos inválidos foram aceites com sucesso. O backend persistiu um customer com nome vazio, email inválido e telefone arbitrário, e aceitou também uma order com produto vazio, quantidade negativa e preço negativo. A leitura posterior de `/api/customers` e `/api/orders` confirmou que os registos inválidos ficaram guardados na aplicação.

Evidência real:

```
curl -s -X POST http://localhost:8090/api/customers \
  -H "Content-Type: application/json" \
  -d '{"name": "", "email": "notanemail", "phone": "abc"}'
{"email":"notanemail","id":2,"name":"","phone":"abc"}

curl -s -X POST http://localhost:8090/api/orders \
  -H "Content-Type: application/json" \
  -d '{"customerId": 1, "product": "", "quantity": -999, "price": -1.00}'
{"customer":{"email":"attacker@evil.com","id":1,"name":"TAMPERED",
 "phone":"000000000"},"id":2,"orderDate":"2026-03-29T00:22:26.867132546",
 "price":-1.00,"product":"","quantity":-999}

curl -s http://localhost:8090/api/customers
[{"email":"attacker@evil.com","id":1,"name":"TAMPERED","phone":"000000000"},
 {"email":"notanemail","id":2,"name":"","phone":"abc"}]

curl -s http://localhost:8090/api/orders
[{"customer":{"email":"attacker@evil.com","id":1,"name":"TAMPERED",
 "phone":"000000000"},"id":1,"orderDate":"2026-03-29T00:12:46","price":1.00,
 "product":"SeedForDeleteTest","quantity":1},
 {"customer":{"email":"attacker@evil.com",
 "id":1,"name":"TAMPERED","phone":"000000000"},"id":2,
 "orderDate":"2026-03-29T00:22:27","price":-1.00,"product":"","quantity":-999}]
```

Referência visual: Figuras 10 a 13.

Impacto: a ausência de validação permite corrupção de dados, inconsistência funcional e potencial manipulação de processos dependentes da integridade destes valores.

Mitigação recomendada:

- adicionar `@Valid` nos endpoints de escrita;
- usar anotações como `@Email`, `@NotBlank`, `@Positive` e limites de tamanho;
- reforçar validação de regras de negócio na camada de serviço.

5.4 PT-04: Direct Database Access via Exposed Port

Objetivo: verificar se a base de dados está diretamente acessível a partir do host, permitindo bypass da aplicação.

Comandos executados:

```
mysql -h 127.0.0.1 -P 3306 -u root -ppassword ses25_104
SELECT * FROM customers;
SELECT * FROM orders;
UPDATE customers SET email = 'pwned2@attacker.com' WHERE id = 1;
SELECT * FROM customers WHERE id = 1;
```

Resultado observado: a ligação direta ao MySQL foi bem sucedida usando as credenciais conhecidas. Foi possível consultar diretamente as tabelas `customers` e `orders`, observar os

dados introduzidos nos testes anteriores e alterar o email do customer com `id = 1` sem qualquer controlo da aplicação. A alteração ficou imediatamente refletida na base de dados.

Evidência real:

```
mysql -h 127.0.0.1 -P 3306 -u root -ppassword ses25_104
Welcome to the MariaDB monitor.
MySQL connection id is 286
Server version: 8.0.45 MySQL Community Server - GPL

MySQL [ses25_104]> SELECT * FROM customers;
id  name      email              phone
1   TAMPERED  attacker@evil.com  000000000
2             notanemail         abc
2 rows in set (0.000 sec)

MySQL [ses25_104]> SELECT * FROM orders;
id  customer_id  product            quantity  price  order_date
1   1            SeedForDeleteTest  1         1.00   2026-03-29 00:12:46
2   1                               -999      -1.00  2026-03-29 00:22:27
2 rows in set (0.001 sec)

MySQL [ses25_104]> UPDATE customers
    -> SET email = 'pwned2@attacker.com'
    -> WHERE id = 1;
Query OK, 1 row affected (0.014 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MySQL [ses25_104]> SELECT * FROM customers WHERE id = 1;
id  name      email                 phone
1   TAMPERED  pwned2@attacker.com   000000000
1 row in set (0.004 sec)
```

Referência visual: Figuras 14 a 17.

Impacto: qualquer utilizador com acesso ao host e às credenciais consegue contornar por completo a aplicação, ler todos os dados e modificá-los diretamente na base de dados. O uso do utilizador `root` agrava ainda mais o risco.

Mitigação recomendada:

- remover a exposição pública da porta `3306`;
- limitar o acesso MySQL à rede interna do Docker;
- substituir o utilizador `root` por uma conta dedicada com privilégios mínimos;
- usar credenciais fortes e segregadas por ambiente.

6 Rastreabilidade e severidade

Após a execução dos testes, foi possível confirmar a ligação entre cada cenário explorado, as abuse stories previamente identificadas e as categorias STRIDE mais relevantes.

Teste	Abuse stories	STRIDE	Severidade	Estado
PT-01	AS-01, AS-02	S, I	Crítica	Confirmado
PT-02	AS-03, AS-06, AS-07	T, R, I	Alta	Confirmado
PT-03	AS-05	T	Alta	Confirmado
PT-04	AS-08	S, I, E	Crítica	Confirmado

Figura 1: Categorias STRIDE consideradas na rastreabilidade dos testes.

7 Observações técnicas na baseline

Os resultados observados no pentest tornam-se mais claros quando analisamos o código e a configuração da baseline:

- os controladores `CustomerController` e `OrderController` expõem operações CRUD sem qualquer camada de autenticação ou autorização;
- o `CustomerController` aceita a entidade `Customer` diretamente em `@RequestBody`, enquanto o `OrderController` usa `OrderDTO` sem `@Valid`; nem os modelos nem o DTO apresentam anotações de validação;
- o método de remoção de customers não verifica dependências em `orders`, o que explica o `500` observado ao apagar customers com orders associadas;
- o `OrderService` lança `RuntimeException` crua quando o `customerId` não existe e não existe um handler global para traduzir falhas técnicas em respostas controladas;
- o `docker-compose.yml` publica a base de dados em `3306:3306` e o `WebConfig` limita apenas origens browser, não impedindo chamadas diretas com `curl` nem ligações MySQL a partir do host.

8 Evidências visuais

8.1 Evidência 1: acesso não autenticado

```
┌──(meowstermind㉿LEGION-LULU)-[~/Downloads/SES/ses2526-appsec-recipes-ses25_104]
└─$ curl -s http://localhost:8090/api/customers
[]
```

Figura 2: Listagem de customers sem autenticação.

```
┌──(meowstermind㉿LEGION-LULU)-[~/Downloads/SES/ses2526-appsec-recipes-ses25_104]
└─$ curl -s http://localhost:8090/api/orders
[]
```

Figura 3: Listagem de orders sem autenticação.

```
┌──(meowstermind㉿LEGION-LULU)-[~/Downloads/SES/ses2526-appsec-recipes-ses25_104]
└─$ curl -s -X POST http://localhost:8090/api/customers \
  -H "Content-Type: application/json" \
  -d '{"name":"Alice","email":"alice@example.com","phone":"123456789"}'
{"email":"alice@example.com","id":1,"name":"Alice","phone":"123456789"}
```

Figura 4: Criação de customer sem autenticação.

```
┌──(meowstermind㉿LEGION-LULU)-[~/Downloads/SES/ses2526-appsec-recipes-ses25_104]
└─$ curl -s -X PUT http://localhost:8090/api/customers/1 \
  -H "Content-Type: application/json" \
  -d '{"name":"AliceUpdated","email":"alice2@example.com","phone":"111111111"}'
{"email":"alice2@example.com","id":1,"name":"AliceUpdated","phone":"111111111"}

┌──(meowstermind㉿LEGION-LULU)-[~/Downloads/SES/ses2526-appsec-recipes-ses25_104]
└─$ curl -s http://localhost:8090/api/customers/1
{"email":"alice2@example.com","id":1,"name":"AliceUpdated","phone":"111111111"}
```

Figura 5: Atualização de customer sem autenticação.

8.2 Evidência 2: enumeração e exposição de dados

```
┌──(meowstermind㉿LEGION-LULU)-[~/Downloads/SES/ses2526-appsec-recipes-ses25_104]
└─$ curl -i http://localhost:8090/api/customers/1
HTTP/1.1 200
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Type: application/json
Transfer-Encoding: chunked
Date: Sun, 29 Mar 2026 00:06:08 GMT

{"email":"alice2@example.com","id":1,"name":"AliceUpdated","phone":"111111111"}

┌──(meowstermind㉿LEGION-LULU)-[~/Downloads/SES/ses2526-appsec-recipes-ses25_104]
└─$ curl -i http://localhost:8090/api/customers/999
HTTP/1.1 404
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Length: 0
Date: Sun, 29 Mar 2026 00:07:06 GMT
```

Figura 6: Enumeração de recursos e exposição excessiva de dados.

Figura 7: Alteração arbitrária de customer via `PUT`.

Figura 8: Confirmação da persistência da alteração indevida.

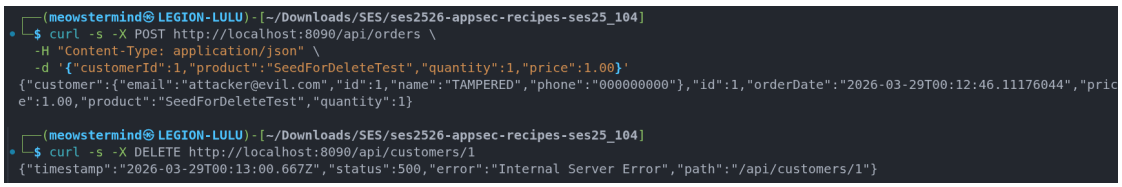

Figura 9: Erro não tratado ao apagar customer com orders associadas.

8.3 Evidência 3: aceitação de input inválido

Figura 10: Criação de customer com input inválido aceite pelo backend.

Figura 11: Criação de order com valores inválidos aceite pelo backend.

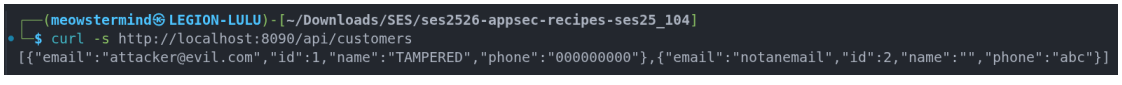

Figura 12: Persistência de customer com dados inválidos.

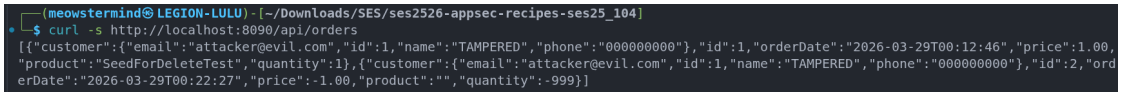

Figura 13: Persistência de order com dados inválidos.

8.4 Evidência 4: acesso direto à base de dados

```
┌──(meowstermind㉿LEGION-LULU)-[~/Downloads/SES/ses2526-appsec-recipes-ses25_104]
└─$ mysql -h 127.0.0.1 -P 3306 -u root -ppassword ses25_104
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 286
Server version: 8.0.45 MySQL Community Server - GPL

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

Figura 14: Ligação direta à base de dados através da porta exposta.

```
MySQL [ses25_104]> SELECT * FROM customers;
+----+----------+--------------------+-----------+
| id | name     | email              | phone     |
+----+----------+--------------------+-----------+
|  1 | TAMPERED | attacker@evil.com | 000000000 |
|  2 |          | notanemail         | abc       |
+----+----------+--------------------+-----------+
2 rows in set (0.000 sec)
```

Figura 15: Leitura direta da tabela `customers`.

```
MySQL [ses25_104]> SELECT * FROM orders;
+----+-------------+------------------+----------+-------+---------------------+
| id | customer_id | product          | quantity | price | order_date          |
+----+-------------+------------------+----------+-------+---------------------+
|  1 |           1 | SeedForDeleteTest |        1 |  1.00 | 2026-03-29 00:12:46 |
|  2 |           1 |                  |     -999 | -1.00 | 2026-03-29 00:22:27 |
+----+-------------+------------------+----------+-------+---------------------+
2 rows in set (0.001 sec)
```

Figura 16: Leitura direta da tabela `orders`.

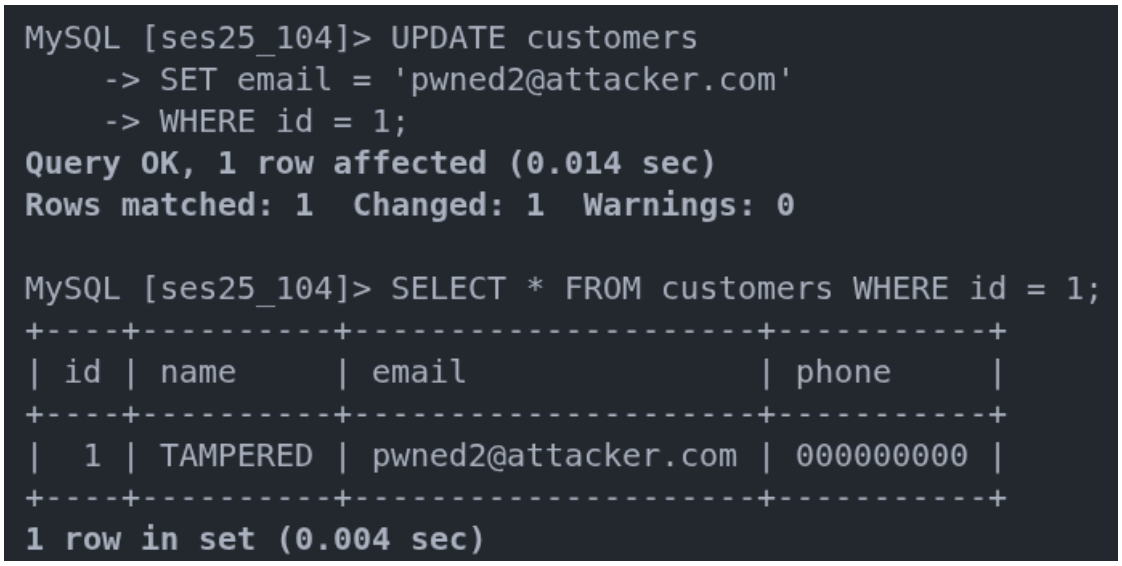

Figura 17: Alteração direta de dados na base de dados.

9 Vulnerabilidades em destaque

Este relatório destaca sobretudo dois findings. O primeiro é o **acesso total sem autenticação (PT-01)**, que representa a falha mais crítica observada na baseline app. Foi possível aceder diretamente aos endpoints da API, listar dados, criar registos e alterar informação sem qualquer mecanismo de autenticação ou autorização, com impacto imediato na confidencialidade e integridade dos dados.

O segundo caso em destaque é o **acesso direto à base de dados através da porta exposta (PT-04)**. Este finding demonstra um bypass completo da aplicação, uma vez que a base de dados MySQL se encontrava acessível diretamente a partir do host, permitindo leitura e alteração de dados sem passar pela lógica do backend. Para além do impacto técnico elevado, este caso reforça a gravidade da configuração insegura da infraestrutura.

Estes dois casos foram selecionados por combinarem elevado impacto, exploração simples e evidência prática forte para suportar a análise apresentada no relatório.

10 Tabela-resumo dos resultados

Teste	Resultado	Impacto observado	Mitigação prioritária
PT-01: Acesso não autenticado e harvesting	Confirmado (`200`/`201`)	Leitura e escrita de dados sem qualquer barreira de autenticação	Autenticação forte e autorização por recurso
PT-02: Tampering por enumeração de IDs	Confirmado	Enumeração de recursos, adulteração de dados e `500` em operações destrutivas	Autorização, controlo de integridade e tratamento de exceções
PT-03: Bypass de validação	Confirmado (`201`)	Persistência de dados inválidos e corrupção da integridade aplicacional	Validação de input e regras de negócio na camada de serviço
PT-04: Acesso direto à base de dados	Confirmado	Bypass total da aplicação e modificação direta dos dados no MySQL	Remover exposição da porta e aplicar privilégios mínimos

11 Conclusão

O assessment manual à baseline app permitiu confirmar fragilidades relevantes de segurança, especialmente ao nível de controlo de acesso, validação de input, exposição excessiva de dados e exposição direta da base de dados.

Em termos de prioridade, os problemas mais críticos observados foram a ausência total de autenticação e a possibilidade de acesso direto ao MySQL através da porta exposta no host. As prioridades de mitigação devem incluir autenticação e autorização, validação de inputs, controlo rigoroso do acesso à base de dados, remoção da exposição pública da porta `3306` e revisão da configuração de infraestrutura.

Pentest Authorization Letter

Controlled Security Assessment Authorization

Engagement	Authorized Penetration Test
Target System	Orderly Application Environment
Assessment Type	Controlled Security Assessment
Authorization Status	Approved for execution within defined scope
Testing Window	Valid from 28 March 2026 to 4 April 2026

This document formally authorizes a controlled penetration test against the **Orderly** application environment.

The pentester is authorized to perform security testing activities strictly limited to the approved environment in order to identify, validate, and document vulnerabilities, weaknesses, and security misconfigurations affecting the in-scope assets.

Authorized Scope

The following assets are included in scope:

- the frontend application at `localhost:8080`;
- the backend application at `localhost:8090`;
- the supporting database environment at `localhost:3306`;
- the API endpoints exposed by the authorized test environment.

Testing is strictly limited to the local Docker Compose environment on localhost. Production systems, cloud infrastructure, and remote environments are explicitly out of scope.

Rules and Limitations

The pentester is authorized to:

- perform controlled tests against the in-scope components;
- submit crafted requests and malformed inputs;
- attempt controlled exploitation of identified vulnerabilities;
- collect evidence required for the assessment deliverables.

The pentester is not authorized to:

- target systems, services, repositories, or infrastructure outside the defined scope;
- perform destructive actions beyond what is strictly necessary to demonstrate a vulnerability;
- cause intentional prolonged denial of service;
- access or affect third-party systems.

All testing activity must remain controlled, documented, proportionate, and limited to the approved scope. Any critical finding, unexpected impact, or material service disruption must be reported immediately to the responsible authority.

Any data accessed or collected during the assessment must be treated as confidential, used solely for this engagement, not disclosed to third parties, and securely deleted upon completion of the authorized testing window.

Signatures

Pentester

_____

Name / Company
Date

Responsible Authority

_____

Name / Title
Date