CS 305 Project One Template

Document Revision History

Version	Date	Author	Comments
1.0	11/9/2025	Gary Travis	Project 1

Client

Developer
Gary Travis

1. Interpreting Client Needs

Determine your client's needs and potential threats and attacks associated with the company's application and software security requirements. Consider the following questions regarding how companies protect against external threats based on the scenario information:

- What is the value of secure communications to the company?
- Are there any international transactions that the company produces?
- Are there governmental restrictions on secure communications to consider?
- What external threats might be present now and in the immediate future?
- What modernization requirements must be considered, such as the role of open-source libraries and evolving web application technologies?

Artemis Financial develops individualized financial plans and exposes a RESTful API that handles sensitive client information such as identities, account details, and planning data. Because the application transmits and processes confidential financial data, secure communications are essential to protect confidentiality and integrity, maintain customer trust, and reduce legal liability. Using strong transport security (TLS 1.2/1.3), modern cipher suites, and strict certificate management helps prevent eavesdropping, tampering, and downgrade attacks. Enforcing HSTS and secure cookies further protects browser-based interactions with any front-end that consumes the API.

Artemis primarily serves customers in the United States, but the company could engage with clients or partners across borders. If any data belongs to non-U.S. residents or is processed in non-U.S. regions, international privacy and data transfer obligations may apply (for example, GDPR-style requirements). Even when data remains domestic, state privacy laws and industry norms still expect encryption in transit, least-privilege access, and clear retention policies.

Governmental restrictions and expectations also influence secure communications. If Artemis works with public sector clients, cryptographic modules may need to be FIPS-validated, and key management must follow formal policies (generation, rotation, and revocation). Regardless of sector, the company should document which cryptographic libraries are used, how certificates and keystores are protected, and how incident response will handle potential key compromise.

External threats facing the API now and in the near term include the common web risks highlighted by OWASP: injection, broken authentication and authorization, sensitive data exposure, insecure design, misconfiguration, and use of vulnerable and outdated components. Practical threats include credential stuffing, brute-force or enumeration of endpoints, scraping of unauthenticated resources, cross-origin abuse if CORS is overly permissive, and supply-chain issues in open-source dependencies. Transport-layer attacks (such as man-in-the-middle on misconfigured TLS) and logging/monitoring gaps are also concerns.

Modernization requirements focus on reducing custom security code and relying on well-supported standards and libraries. The API should follow Spring Boot's security patterns for authentication/authorization, input validation (bean validation on request DTOs), consistent error handling, and security headers. Open-source libraries are a core part of the stack, so Artemis should maintain a software bill of materials (SBOM), pin dependency versions, and continuously scan for known vulnerabilities using software composition analysis. As web application technologies evolve, the organization should plan for regular framework upgrades, short-lived tokens with rotation, secret management outside of source control, rate limiting and abuse protection, and automated CI/CD checks (tests, static analysis, and dependency scans) to keep the service secure as it changes.

2. Areas of Security

Refer to the vulnerability assessment process flow diagram. Identify which areas of security apply to Artemis Financial's software application. Justify your reasoning for why each area is relevant to the software application.

Artemis Financial's RESTful web application involves several key areas of security identified in the vulnerability assessment process flow. The relevant areas are Architecture Review, Input Validation, APIs, Cryptography, Client/Server Configuration, Code Error Handling, and Code Quality and Encapsulation.

Architecture Review ensures the overall design supports secure communication, data storage, and access control. The application must define clear trust boundaries between the client, the REST API, and the database. A secure architecture reduces the likelihood of data exposure and improper privilege escalation.

Input Validation applies because the API receives user-supplied data such as names, account information, and query parameters. Without validation, attackers could inject malicious input that affects database queries or responses. Implementing bean validation on request objects and encoding outputs prevents injection and data-integrity issues.

APIs are the core interface of this application. Each endpoint must be authenticated, authorized, and rate-limited to prevent abuse. Proper configuration of cross-origin resource sharing (CORS) is also required so only trusted front-end domains can access the API.

Cryptography is critical because the system transmits and stores confidential financial data. Strong TLS 1.2/1.3 connections, secure certificates, and modern cipher suites protect data in transit, while safe key storage and approved cryptographic libraries protect data at rest.

Client/Server Configuration involves hardening the environment and enforcing consistent security policies on both sides. This includes using secure headers (HSTS, X-Content-Type-Options, Referrer-Policy) and disabling unnecessary services or default accounts on the server.

Code Error Handling is relevant because unhandled exceptions and detailed stack traces could leak internal information. Centralized exception management and generic client error messages help prevent attackers from learning about internal classes or queries.

Code Quality and Encapsulation ensure that the code follows best practices such as using dependency injection, keeping sensitive fields private, avoiding hardcoded secrets, and removing unused imports or dependencies. Well-structured, encapsulated code limits how vulnerabilities can spread through the system.

These seven areas align directly with the process flow diagram and reflect how each part of the application—from architecture to detailed implementation—contributes to the overall security posture of Artemis Financial's software.

3. Manual Review

Continue working through the vulnerability assessment process flow diagram. Identify all vulnerabilities in the code base by manually inspecting the code.

1. Hardcoded database credentials and URL
   File: src/main/java/com/twk/restservice/DocData.java
   Issue: DriverManager.getConnection(...) uses a hardcoded JDBC URL and default credentials.
   Risk: Credentials can leak via source control or logs; non-TLS DB traffic could be intercepted.
   Fix: Move DB URL/user/password to environment variables or a secrets manager; use a least-privilege DB user; require TLS on the DB connection.
2. Unclosed JDBC resources (connection/statement/result set)
   File: src/main/java/com/twk/restservice/DocData.java
   Issue: Database resources are not managed with try-with-resources.
   Risk: Resource leaks can exhaust the connection pool and cause denial of service.
   Fix: Wrap Connection/PreparedStatement/ResultSet in try (...) { } blocks or use a pooled DataSource managed by the framework.
3. Printing stack traces directly
   File: src/main/java/com/twk/restservice/DocData.java (and any catch blocks using e.printStackTrace())
   Issue: Uses e.printStackTrace() instead of structured logging and centralized error handling.
   Risk: Implementation details can leak; noisy logs hinder detection and response.
   Fix: Replace with a logger (SLF4J/Logback) and centralize exception handling to return generic client messages.
4. Missing server-side validation on request parameters
   File: src/main/java/com/twk/restservice/CRUDController.java
   Issue: Parameters such as business_name are accepted without bean validation.
   Risk: Malformed or malicious input can reach data access code and cause injection or logic bugs.
   Fix: Create request DTOs and annotate with @NotBlank, @Size, @Pattern; add @Validated on the controller and return 400 on invalid input.
5. No authentication/authorization protecting endpoints
   Files: src/main/java/com/twk/restservice/GreetingController.java and CRUDController.java
   Issue: Endpoints appear publicly accessible; no Spring Security configuration is present.
   Risk: Unauthenticated users can query or modify data; enables scraping and abuse.
   Fix: Add Spring Security; require authentication for non-health endpoints; implement role/scope checks and rate limiting.
6. Overly permissive or absent CORS policy
   Project-wide (no explicit Security/CORS config found)
   Issue: Either default CORS or wildcard origins are effectively allowed.
   Risk: Cross-origin scripts can call the API from untrusted sites.
   Fix: Configure CORS to allow only trusted front-end origins, required methods, and headers.
7. Missing security headers and transport hardening
   Project-wide (no SecurityFilterChain or equivalent)
   Issue: No HSTS, X-Content-Type-Options, frame protections, or Referrer-Policy are set.
   Risk: Increases exposure to downgrade attacks, MIME sniffing, and clickjacking when consumed by browsers.

Fix: Enforce TLS 1.2/1.3 only; add HSTS and standard security headers via Spring Security configuration.

8. Outdated and potentially vulnerable cryptography library
   File: pom.xml
   Issue: org.bouncycastle:bcprov-jdk15on uses an old version.
   Risk: Older bcprov releases have known CVEs; using outdated crypto can undermine confidentiality/integrity.
   Fix: Upgrade bcprov to a current stable version or remove if unused; rerun dependency-check to verify remediation.
9. Outdated Spring Boot baseline and transitive dependencies
   File: pom.xml (parent spring-boot-starter-parent 2.2.x)
   Issue: The baseline is end-of-life; transitive components (web, JSON, logging) may include known vulnerabilities.
   Risk: Heightened supply-chain exposure through transitive CVEs.
   Fix: Plan an upgrade path to a supported Spring Boot line (2.7.x LTS or 3.x with Java 17), follow the migration guide, then rescan.
10. Inconsistent output modeling and potential information disclosure
    Files: src/main/java/com/twk/restservice/CRUDController.java, DocData.java
    Issue: Returning doc.toString() or unstructured responses can expose class names or future fields.
    Risk: Reveals internal structure and increases accidental PII leakage risk.
    Fix: Return explicit DTOs with controlled fields; avoid default Object.toString() in API responses.

4. Static Testing

Run a dependency check on Artemis Financial's software application to identify all security vulnerabilities in the code. Record the output from the dependency-check report. Include the following items:

- The names or vulnerability codes of the known vulnerabilities
- A brief description and recommended solutions provided by the dependency-check report
- Any attribution that documents how this vulnerability has been identified or documented previously

For the static testing part of the vulnerability assessment, I used the OWASP Dependency-Check Maven plug-in. I ran the tool using the Maven goal `org.owasp:dependency-check-maven:check`. The tool started normally and downloaded all of the project's dependencies, but the scan could not finish because it was unable to download the vulnerability feed from the National Vulnerability Database (NVD). The error showed a 403 response when trying to download the file "nvdcve-1.1-modified.meta," which caused the analysis to stop early with a build failure.

Even though the report did not finish, the tool still gave me useful information about how the scan works. It showed that the project uses several common libraries such as Spring Boot (2.2.x), Jackson, H2, Gson, Guava, jsoup, and the BouncyCastle crypto library. Many of these libraries have known vulnerabilities in older versions, and these issues are normally listed in NVD and GitHub Security Advisories. The tool also warned me that my plug-in version (5.3.0) is outdated and recommended upgrading to the newest version (12.1.0). Newer versions require an NVD API key, which explains why the feed could not be downloaded.

If the scan had completed successfully, the dependency-check report would list each vulnerable dependency by its CVE number, the severity score, a short description, and the recommended fix (such as upgrading to a safer version). This information comes from public sources like the NVD, MITRE CVE database, and GitHub advisories. After updating the plug-in and adding the NVD API key, the scan can be re-run to produce the full HTML report with the actual CVE findings.

Even though this scan did not finish, the attempt still showed how static testing works and what kind of vulnerabilities it is meant to identify. I can take the results and upgrade the plug-in so that future scans will be able to complete successfully and show the exact vulnerabilities that need to be addressed.

5. Mitigation Plan

Interpret the results from the manual review and static testing report. Then identify the steps to mitigate the identified security vulnerabilities for Artemis Financial's software application.

Based on the issues I found during the manual code review and the attempted static testing, there are several steps that Artemis Financial should take to improve the security of their web application. These fixes focus on cleaning up the code, securing sensitive data, updating old libraries, and strengthening the overall protections around the API.

First, the application needs to remove all hardcoded database credentials. The current code uses a direct JDBC connection with a username and password typed into the source file. This is unsafe because anyone who sees the code would also have direct access to the database. The solution is to move these values into environment variables or a secure secrets manager and make sure the database user has limited privileges. The JDBC code should also be updated to use try-with-resources so the connection and statements are always closed properly.

Next, the application should add real input validation. The controllers accept user input with no restrictions, which could allow someone to send unexpected or harmful data. Creating request objects and adding bean validation annotations like @NotBlank or @Size will help prevent invalid input from getting into the system. The API should also return structured response objects instead of using toString(), which can leak internal class information.

Another important step is to add authentication and authorization. Right now, the API endpoints appear fully open to the public, which is not safe for a financial service. Implementing Spring Security will let the company require login or token-based authentication before someone can access the API. They can also define roles and permissions so only certain users can read or update financial data. Adding rate limiting can prevent attackers from trying to brute-force or spam the system.

Since the static testing showed that the dependency-check plug-in was outdated, the next step is to update the security scanning tools. The project should upgrade to the latest version of the dependency-check Maven plug-in (12.x) and set up an NVD API key so the tool can download the vulnerability database. Once this is done, the scan should be re-run to get the actual list of CVEs affecting libraries like Spring Boot, Jackson, H2, BouncyCastle, and others. Keeping these tools up to date is important because outdated libraries are a common security risk in modern applications.

The project should also add missing security headers and tighten client/server configuration. This includes enabling HTTPS-only traffic, adding HSTS, blocking MIME sniffing, and limiting the allowed CORS origins. These steps help prevent attacks like man-in-the-middle, clickjacking, and cross-site scripting.

Finally, Artemis Financial should adopt some DevSecOps practices so these problems don't reappear in the future. This includes running dependency-check and static analysis tools in the build pipeline, rotating secrets regularly, generating an SBOM (software bill of materials), and scheduling regular framework upgrades. By making security part of the development process instead of something done once, the company can reduce risk and catch issues earlier.

Overall, these mitigation steps will help secure the code, modernize outdated components, and protect sensitive financial data while also improving the long-term stability and security of Artemis Financial's software.