# CS 305 Project One Template

## 

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **September 20, 2025** | **Alondra Paulino Santos** | **Initial Submission** |

## 

## Client



## Developer

## Alondra Paulino Santos

**1. Interpreting Client Needs**

Artemis Financials’ primary business involves creating personalized financial plans for clients, covering areas such as savings, retirement, investments, and insurance. Because this work depends on maintaining client trust and complying with financial regulations, secure communication is one of the company’s highest priorities. All data that flows between Artemis Financial and its customers, such as account details, investment strategies, and retirement goals, must be encrypted in transit and at rest to preserve confidentiality and integrity. Without strong encryption and secure protocols like TLS, Artemis risks both legal consequences and loss of client confidence if sensitive data were intercepted.

Another client need involves international transactions. As a consulting firm with potential global customers and partners, Artemis Financial may process data across borders. This exposes the company to international security and privacy laws such as the General Data Protection Regulation (GDPR) in the European Union. Compliance with these laws requires secure communication channels, proper handling of personally identifiable information (PII), and limitations on data transfers without explicit safeguards. Therefore, security is not only a technical requirement but also a legal necessity for Artemis Financials’ continued operations.

Governmental restrictions and financial industry standards also shape Artemis’s security requirements. Frameworks such as Payment Card Industry Data Security Standard (PCI DSS) and U.S. privacy regulations like the Gramm–Leach–Bliley Act (GLBA) demand that financial institutions implement strong controls around authentication, encryption, and data access. Failure to meet these requirements could result in fines, lawsuits, or the inability to legally operate in certain regions.

Looking at potential threats, Artemis faces both current risks and emerging ones. Currently, attackers often exploit web applications with SQL injection, cross-site scripting (XSS), phishing, and credential stuffing. At the same time, new risks are emerging as attackers target open-source libraries and third-party dependencies with known vulnerabilities. These risks are especially critical since Artemis plans to modernize its operations, and modernization typically relies on integrating frameworks and libraries that require careful monitoring.

Finally, modernization itself creates unique requirements. Artemis must be able to adopt evolving web application technologies without weakening its security posture. This means building a secure development lifecycle that includes dependency checking, secure coding practices, and regular updates to open-source components. By addressing these needs, Artemis Financial can align its modernization strategy with its mission that “security is everyone’s responsibility,” ensuring both innovation and protection.

**2. Areas of Security**

Based on Artemis Financials’ RESTful web application and the vulnerability assessment process flow diagram, several areas of security are especially relevant. Each area plays a role in protecting customer data and ensuring that Artemis meets both business and regulatory requirements.

**Input Validation**

Because customers provide personal and financial information through the application, input validation is critical. Without proper validation and sanitization, the application could become vulnerable to injection attacks such as SQL injection or cross-site scripting (XSS). Implementing strict input validation ensures that only safe, expected data is processed, reducing the risk of malicious commands being executed.

**APIs**

Artemis Financial relies on a RESTful API to deliver services to customers. Securing API interactions is essential because APIs are often targeted by attackers who attempt to exploit poorly protected endpoints. Strong authentication, authorization, and rate limiting must be applied to API requests to prevent unauthorized access and denial-of-service attempts.

**Cryptography**

Since Artemis manages highly sensitive data, such as retirement plans, investments, and insurance details, cryptography is vital. Encryption protects data both at rest in databases and in transit between clients and servers. Weak or outdated encryption could allow attackers to intercept or decrypt financial records. Therefore, Artemis must adopt current encryption standards, manage keys securely, and avoid practices such as hardcoded keys.

**Client/Server Communication**

All communication between clients and Artemis Financials’ servers must be secure. Transport Layer Security (TLS) ensures confidentiality and integrity during data exchanges. Without secure client/server communication, attackers could launch man-in-the-middle (MITM) attacks and intercept financial information. Strong, up-to-date certificates and enforced HTTPS connections protect customers from these threats.

**Code Error Handling**

Proper error handling is necessary to prevent information leakage. If the application displays detailed stack traces or system messages to the user, attackers could use this information to map out vulnerabilities. Secure error handling means returning generic messages to users while logging the full technical details on the server for developers to review. This approach balances security with troubleshooting needs.

**Code Quality**

Secure coding practices and consistent patterns reduce the chance of vulnerabilities being introduced into the application. High code quality ensures maintainability and lowers the likelihood of introducing bugs that create new security holes. Following industry standards and frameworks, such as the OWASP Secure Coding Practices, helps Artemis reduce risk at the development stage.

**Encapsulation**

Encapsulation is also relevant because Artemis’s application uses data structures to manage sensitive financial records. Proper encapsulation restricts direct access to internal objects and reduces the chance of accidental or malicious manipulation. This principle is essential for maintaining the integrity and consistency of customer data across different modules of the application.

In summary, input validation, secure APIs, cryptography, secure client/server communication, error handling, code quality, and encapsulation all apply directly to Artemis Financials’ web application. Each area works together to reduce vulnerabilities, protect financial data, and support the company’s mission that “security is everyone’s responsibility.”

**3. Manual Review**

|  |  |  |
| --- | --- | --- |
| **Class/File** | **Vulnerability** | **Explanation** |
| **DocData.java** | Hardcoded database credentials | The JDBC connection uses hardcoded credentials (root, root) in the source code. This creates a major security risk because if the code is leaked or decompiled, attackers immediately gain access to the database. Best practice is to store sensitive credentials in encrypted configuration files or environment variables and never directly in the code. **See Figure 1.** |
| **DocData.java** | Unvalidated user input in SQL connection | The read\_document method accepts user input (key, value) but does not validate or sanitize these parameters before using them in SQL operations. This opens the door for SQL injection, where attackers can insert malicious input to access or modify sensitive data. Input validation and parameterized queries (e.g., PreparedStatement) should be implemented to prevent this. Without these protections, attackers could exfiltrate sensitive customer data or even compromise the entire database server. **See Figure 2.** |
| **CRUDController.java** | Missing validation for business\_name parameter | The /read endpoint accepts the business\_name parameter directly from the client without any checks. Without validation, attackers can submit malicious input such as embedded scripts (XSS) or oversized strings to overload the system. Secure development requires validating or escaping all client-supplied data before using it in application logic. Unchecked parameters in REST endpoints are one of the most common OWASP Top 10 risks (A03: Injection). **See Figure 3.** |
| **customer.java** | Unprotected account balance updates | The deposit method updates account\_balance without validating inputs. This means attackers could pass negative values to withdraw funds or very large values to cause overflow. Input checks (e.g., rejecting negatives and limiting range) must be enforced to maintain financial integrity and prevent abuse. |
| **customer.java** | Exposed fields without encapsulation | The account\_balance field is not private, allowing other classes in the package to modify it directly. This violates encapsulation and creates a security risk since balances could be changed outside of the approved methods. Fields holding sensitive values should always be private, and updates should go through controlled setters with validation. |
| **myDateTime.java** | Incomplete and unsafe implementation | The retrieveDateTime and setMyDateTime methods are left unimplemented or incomplete. If attackers can supply invalid values (like negative numbers or impossible times), the system could enter an unstable state. Methods dealing with system or time data should be fully implemented with validation rules to prevent improper use. |
| **CRUD.java** | Data duplication without integrity checks | The constructors reuse input parameters without enforcing distinct or validated values. This could create inconsistencies in data handling if the same value is unintentionally passed to both fields. Defensive coding practices should require validation and integrity checks to ensure stored values are consistent and meaningful. |
| |  |  |  | | --- | --- | --- | |  | **General (Multiple Classes)** |  | | Weak error handling | Classes such as DocData catch exceptions but only call printStackTrace(). This leaks system details (like database errors) to logs and provides no real recovery or alerting. Secure applications must use structured logging frameworks, mask sensitive information, and provide meaningful error messages without exposing system internals. Proper error handling ensures that sensitive system details remain hidden from attackers while still giving administrators the information needed to respond |

Figure 1. Insecure hardcoded database credentials in DocData.read\_document() (lines 26–31).

A computer screen shot of a computer code

AI-generated content may be incorrect.

Figure 2. Unsanitized user input from @RequestParam in CRUDController (lines 13–17).

A black text on a white background

AI-generated content may be incorrect.

Figure 3. Lack of encapsulation and validation in the customer class (lines 3–5).



**4. Static Testing**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Dependency** | **Severity** | **Vulnerability & Explanation** | |  |  |  | **Recommended Fix** | | --- | --- | --- | --- | | **Attribution** |
| **Tomcat Embed Core / WebSocket (9.0.30)** | Critical (45–46 CVEs) | This Tomcat build is several years behind and includes multiple RCE/DoS and request-smuggling issues. An attacker could crash the service or execute code at the app server layer. **See Figure 4 for details.** | Upgrade Tomcat to a patched 9.0.x line (≥ 9.0.85), ideally by upgrading Spring Boot so Tomcat is managed automatically. Re-scan after upgrade. | NVD + Apache advisories (expanded panel) |
| **Spring Framework (5.2.3: web, webmvc, context, core, expression)** | |  |  | | --- | --- | |  | Critical (13–19 CVEs each module) | | Outdated Spring core/web modules contain critical request-handling and expression bugs that can enable code execution or data exposure. Since Spring underpins the whole app, these are high-impact. | Upgrade via the Spring Boot BOM to a supported release (Boot 2.7.x or 3.x), which pulls patched Spring 5.3.x/6.x. Re-test. | NVD + VMware/Spring advisories |
| **Spring Boot (2.2.4.RELEASE)** | Critical (3 CVEs + transitive deps) | This Boot version pins many old components (Tomcat, Jackson, Logback). Keeping Boot old keeps your transitive graph vulnerable. | Upgrade via the Spring Boot BOM to a supported release (Boot 2.7.x or 3.x), which pulls patched Spring 5.3.x/6.x. Re-test. | NVD + Spring Boot release notes |
| **SnakeYAML (1.25)** | Critical (8 CVEs) | Older SnakeYAML versions allow unsafe deserialization of untrusted YAML, which can lead to RCE or DoS via crafted payloads. **See Figure 5 for details.** | Upgrade to SnakeYAML ≥ 2.0 and avoid loading untrusted YAML without safe loaders. | NVD + SnakeYAML advisories |
| **Jackson Databind (2.10.2)** | High | Polymorphic deserialization in old Jackson can be abused to execute arbitrary code using gadget classes. **See Figure 6 for details.** | Upgrade to ≥ 2.13/2.15 and disable default typing unless strictly needed. Re-scan. | NVD + FasterXML advisories |
| **Jackson Core (2.10.2)** | High | The parser itself had DoS and information-leak issues in older lines; deeply nested JSON or crafted inputs can crash parsers or leak buffer data. | Upgrade Core in lock-step with Databind (≥ 2.13/2.15). Apply input limits (size/depth). | NVD + FasterXML advisories |
| **Logback Classic/Core (1.2.3)** | High | Older Logback can leak sensitive data or allow log forging. Logging vulns are often overlooked but enable stealth and data exposure. | Upgrade to ≥ 1.2.13 (or newer stable 1.5.x), review appenders/patterns, and sanitize logged fields. | NVD + Logback release notes |
| **Hibernate Validator (6.0.18.Final)** | Medium | EL/message interpolation bugs can let attacker-supplied content run as expressions in validation messages, leading to info disclosure or code execution in specific setups. | Upgrade to ≥ 6.2.0 (or 7.x) and avoid interpolating user input into messages. | NVD + Red Hat/JBoss advisories |
| **BouncyCastle bcprov-jdk15on (1.46)** | High | Very old crypto provider with signature validation, timing, and curve-handling flaws. These can leak keys or allow tampering with signatures/handshakes. | Upgrade to ≥ 1.78 (LTS) or current stable; re-evaluate any custom crypto usage. | NVD + BouncyCastle release notes |

**Figure 4**. Outdated Tomcat embed-core (v9.0.30) flagged in the dependency-check report.

A screenshot of a computer

AI-generated content may be incorrect.

**Figure 5.** Critical SnakeYAML vulnerability (v1.25) shown in the dependency-check report.

**A screenshot of a computer

AI-generated content may be incorrect.**

**Figure 6.** High-risk Jackson Databind issue (v2.10.2) highlighted in the dependency-check report.

**A screenshot of a computer

AI-generated content may be incorrect.**

**Figure 7**. Dependency-Check report summary showing scan details and vulnerable dependencies count.

**A screenshot of a computer

AI-generated content may be incorrect.**

**5. Mitigation Plan**

|  |  |  |
| --- | --- | --- |
| **Vulnerability** | **Recommended Fix** | **Rational/Detail** |
| **Hardcoded database credentials (DocData.java)** | Externalize credentials into encrypted configuration files (e.g., application.properties with Spring Boot) or environment variables. Use a secrets manager like Vault or AWS Secrets Manager for production. | Prevents attackers from extracting passwords if code is leaked or decompiled. Enforcing secure storage ensures compliance with OWASP A02 (Cryptographic Failures). |
| **Unvalidated user input in SQL (DocData.java)** | Switch to PreparedStatement for parameterized queries and validate all inputs (length, type, allowed characters). Apply centralized input validation with frameworks like Hibernate Validator. | Eliminates SQL injection vectors. Parameterized queries ensure that user input is treated as data, not executable code. |
| **Unvalidated business\_name parameter (CRUDController.java)** | Add input validation (regex checks, length limits) or escaping before using the parameter. Use Spring’s built-in validation annotations (@Valid, @Size, @Pattern). | Defends against XSS and denial-of-service attacks caused by oversized or malicious input. Matches OWASP A03 (Injection) controls. |
| **Unprotected account balance updates (customer.java)** | Validate that deposit amounts are positive and within a safe numeric range. Add business rules (e.g., max daily deposit). | Prevents abuse such as withdrawing funds via negative deposits or triggering integer overflow. |
| **Exposed fields without encapsulation (customer.java)** | Mark account\_balance as private and enforce updates through validated setter methods. | Ensures only authorized methods can change sensitive financial fields. Preserves integrity of account data. |
| **Incomplete implementation in myDateTime.java** | Fully implement validation logic (check ranges for hours 0–23, minutes/seconds 0–59). Throw exceptions or reject invalid values. | Prevents unstable states and ensures time values remain meaningful. Insecure defaults or incomplete logic could be exploited. |
| **Weak error handling (multiple classes)** | Replace printStackTrace() with structured logging (e.g., SLF4J + Logback). Mask sensitive details in logs. Provide user-friendly error messages while logging actionable details securely. | Prevents sensitive database or system errors from leaking to logs. Maintains security visibility without exposing attack surfaces. |
| **Outdated Tomcat Embed Core/WebSocket (9.0.30)** | Upgrade Spring Boot version so that Tomcat is pulled in at a patched 9.0.x (≥ 9.0.85). Re-run dependency-check to confirm fixes. | Old Tomcat versions contain dozens of CVEs (RCE, DoS). Staying current is critical because Tomcat is the app server itself. |
| **Outdated Spring Framework modules (5.2.3)** | Upgrade Spring Boot to 2.7.x or 3.x, which pulls Spring 5.3.x/6.x modules. Validate application compatibility with newer framework versions. | Outdated Spring is a high-risk issue (request handling bugs, RCEs). Framework upgrades reduce long-term exposure. |
| **SnakeYAML 1.25** | Upgrade to SnakeYAML ≥ 2.0 and use the SafeConstructor to restrict deserialization. Avoid loading untrusted YAML unless sanitized. | Old SnakeYAML versions allow unsafe deserialization, leading to possible RCE. New versions enforce safer parsing defaults. |
| **Jackson Databind/Core (2.10.2)** | Upgrade Jackson to ≥ 2.13/2.15. Disable enableDefaultTyping() unless absolutely necessary. | Protects against polymorphic deserialization gadget attacks and parser DoS. Jackson has a long history of RCE vulnerabilities in old lines. |
| **Logback Classic/Core (1.2.3)** | Upgrade Logback to ≥ 1.2.13 or 1.5.x. Review logging configuration to sanitize user-controlled data. | Fixes CVEs where attackers could forge logs or trigger sensitive data leaks. Ensures compliance with secure logging practices. |
| **BouncyCastle (1.46)** | Upgrade to ≥ 1.78 (LTS). Replace deprecated algorithms and re-scan for cryptographic compliance. | Outdated crypto providers may have flaws in key validation or curve handling, risking cryptographic failures. |
| **Hibernate Validator (6.0.18.Final)** | Upgrade to Hibernate Validator ≥ 6.2.0 or 7.x. Avoid user-controlled data in message interpolation. | Prevents attacker-supplied expressions from being evaluated, closing info-leak/code-execution risks. |

This mitigation plan brings together the findings from both the manual code review and the static dependency analysis. By addressing insecure coding practices and upgrading outdated libraries, Artemis Financial can significantly lower its risk of attacks. These steps ensure the application is modern, resilient, and aligned with secure development best practices.