CS 305 Project One Template

Document Revision History

Version	Date	Author	Comments
1.0	**01/24/26**	**Juan Gonzalez**	**OWASP Dependency-Check used with CVEs validated through NVD.**

Client

Instructions
Submit this completed vulnerability assessment report. Replace the bracketed text with the relevant information. In this report, identify your security vulnerability findings and recommend the next steps to remedy the issues you have found.

- Respond to the five steps outlined below and include your findings.
- Respond using your own words. You may also include images or supporting materials. If you include them, make certain to insert them in the relevant locations in the document.
- Refer to the Project One Guidelines and Rubric for more detailed instructions about each section of the template.

Developer
Juan Gonzalez

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 operates a RESTful web application programming interface (API) that supports the creation and management of individualized financial plans, including savings, retirement, investment, and insurance information. Because this application processes sensitive financial and personal customer data, secure software design is critical to protecting confidentiality, maintaining data integrity, and preserving client trust. Secure communications ensure that customer information can't be intercepted or altered while in transit between the client and the API.

The scenario indicates that Artemis Financial serves customers around the world, which suggests the possibility of international data access and transactions. As a result, the application must be designed with strong encryption and access controls to protect customer information regardless of geographic locations. Financial applications are commonly subject to regulatory and privacy expectations, and even if the API is not directly responsible for payment processing, encryption-in-transit and controlled access are baseline requirements in the financial services industry.

Artemis Financial's web application faces several external threats both now and in the near future. These include man-in-the-middle attacks if secure communications protocols are not enforced, credential-based attacks such as brute force or credential stuffing, and broken authentication or authorization that could allow unauthorized access to sensitive endpoints. Additional threats include injection attacks if user input is not properly validated, denial-of-service attacks or API abuse due to a lack of rate limiting, and the risk of data leakage through overly verbose error messages or logging.

As Artemis Financial modernizes its operations, the company must also consider evolving security requirements related to open-source dependencies and moder web technologies. The application relies on third-party libraires that must be continuously monitored for known vulnerabilities and updated when security issues are discovered. Modernization also requires adopting secure API design principles, such as enforcing HTTPS with strong TLS configurations, applying least-privilege access controls, and ensuring the application is configured securely by default. These measures help reduce the attack surface and support the company's mission that security is everyone's responsibility.

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.

Several areas of security from the vulnerability assessment process are directly applicable to Artemis Financial's web-based application due to its role as a RESTful API handling sensitive financial information. Authentication and authorization are critical because the API exposes endpoints that should only be accessible to verified users. Without proper authentication and role-based access controls, unauthorized individuals could access or manipulate financial data, leading to serious security and privacy violations.

Input validation and output handling are also essential security areas for this application. The API accepts user input through request parameters, which creates risk if that input is not properly validated or sanitized. Improper input handling can lead to injection attacks or unintended behavior within the application. Additionally, because user-provided input is included in API responses, improper output handling could result in data leakage or cross-site scripting risks if the data is later rendered in a web-based client.

Dependency management and software composition analysis are also relevant because the application relies on open-source libraries such as Spring Boot and cryptographic providers. Vulnerabilities in third-party dependencies can introduce serious security risks even if the application's own code is well written. Regularly scanning, updating, and managing these dependencies is necessary to reduce exposure to known vulnerabilities.

Also, secure configuration, error handling, and logging are important security considerations for this application. Misconfigured services, overly verbose error messages, or improper logging can expose internal implementation details to attackers. Proper configuration, controlled error responses, and security-focused logging help minimize information disclosure while still allowing developers to monitor and respond to suspicious activity.

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

Finding 1: No authentication/authorization on endpoint

Location: GreetingController.java (@GetMapping("/greeting"))

Issue: The /greeting endpoint is publicly accessible with no authentication or authorization checks.

Risk: Any external user can call the API endpoint, which increases exposure to abuse (automated requests, probing, enumeration) and is not appropriate for an application in a financial context.

Finding 2: Missing input validation for request parameter

Location: GreetingController.java (@RequestParam(value="name"…) String name)

Issue: The name parameter is accepted directly from the client with no validation (length, characters, format).

Risk: Unvalidated input can enable injection-style payloads, log injection (if this value is later logged), and can contribute to denial-of-service risk if very large inputs are accepted.

Finding 3: Reflected user input returned in response

Location: GreetingController.java (String.format(template, name))

Issue: User-controlled input (name) is reflected back in the response message.

Risk: If this response is ever displayed in a browser-based front end without output encoding, this can become a reflected XSS risk. Even for an API, reflecting raw user input increases the chance of data contamination and security issues in downstream consumers.

Finding 4: Hard-coded database credentials

Location: DocData.java DriverManager.getConnection(“jdbc:mysql://localhost:3306/test”,”root”);

Issue: The database username and password are hard coded directly into the source code.

Risk: Hard coded credentials can be exposed through source control, logs, backups, or screenshots. This makes unauthorized database access more likely and violates secure secrets management best practices.

Finding 5: Use of privileged “root” database account

Location: DocData.java “root” account

Issue: The application connects to the database using the MySQL root account.

Risk: This violates least privilege. If the application is compromised, attackers could gain high-level database control, leading to full data loss, modification, or takeover.

Finding 6: No secure error handling

Where: DocData.java e.printStackTrace();

Issue: SQL exceptions are printed using printStackTrace().

Risk: Stack traces can disclose internal details (file paths, SQL drivers, class names, configuration clues). This increases the attacker’s ability to understand the system and exploit it.

Finding 7: Missing input validation for database-related parameters

Where: DocData.java read_documents(String key, String value)

Issue: The method accepts key and value as input but does not validate them (format, allowed fields, length).

Risk: Unvalidated inputs increase the likelihood of injection-style attacks and logic abuse, especially once database queries are implemented.

Finding 8: High SQL injection risk if query is implemented unsafely

Location: DocData.java read_document(…) is intended to query, but no prepared statements are shown.

Issue: The method is clearly intended to read from a database, but there is no evidence of parameterized queries (PreparedStatment).

Risk: If the query is implemented using string concatenation (common in beginner code), it becomes vulnerable to SQL injection. A safer approach is using PreparedStatement with parameters.

Finding 9: Excessive upload limits increase DoS risk

Location: application.properties

```
spring.servlet.multipart.max-file-size=200MB
spring.servlet.multipart.max-file-size=215MB
```

Issue: The application allows very large file uploads and request sizes.

Risk: Large uploads can be abused to exhaust server memory, disk space, or bandwidth. In a financial API context, upload size should be limited to the minimum required for business needs.

Finding 10: File upload enabled without visible access control safeguards

Location: application.properties spring.servlet.multipart.enabled=true

Issue: Multipart uploads are enabled, but int the code reviewed so far there is no evidence of authentication/authorization controls tied to upload functionality.

Risk: If an upload endpoint exists, enabling uploads without strict access control and validation increases the risk of malicious file uploads, storage abuse, and DoS.

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

Static application security testing was performed using the OWASP Dependency-Check Maven plugin. The plugin was successfully integrated into the project and executed using the dependency-check:check goal. During execution, the scan failed due to errors parsing vulnerability data returned by the National Vulnerability Database (NVD) API. This issue occurred after successful API connectivity and authentication and is consistent with known compatibility issues between Dependency-Check and recent NVD JSON schema updates in restricted network environments.

Despite the automated scan failure, software composition analysis was completed by reviewing declared project dependencies and validating known vulnerabilities using authoritative sources. The application includes the dependency org.bouncycastle:bcprov-jdk15on:1.46, which is an outdated cryptographic provider with publicly documented security vulnerabilities. Notable findings include CVE-2013-1624, which describes a TLS timing-related weakness, and CVE-2018-5382, which identifies an integrity flaw in BKS keystores due to insufficient HMAC strength. Both vulnerabilities are documented in the National Vulnerability Database and GitHub Security Advisories. The recommended remediation is to upgrade Bouncy Castle to a current, supported version that resolves these issues.

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.

Implement authentication and authorization controls

Introduce Spring Security to require authentication for all API endpoints. Apply role-based access control to ensure users can only access resources appropriate to their role. This prevents unauthorized access to sensitive functionality.

Enforce secure communications using HTTPS/TLS

Configure the application to require HTTPS with TLS 1.2 or higher. Redirect all HTTP traffic to HTTPS and use secure certificates to protect data in transit from interception or tampering.

Validate all user input at the API boundary

Apply input validation rules to all request parameters, including length restrictions and allowed character sets. Use Java Bean Validation annotations to reject malformed or malicious input before it is processed.

Prevent reflected input from being returned unsafely

Avoid directly including user-supplied input in API responses. If user input must be returned, sanitize or encode the output to prevent downstream injection or cross-site scripting risks.

Remove hard-coded credentials from source code

Eliminate database usernames and passwords from the codebase. Store sensitive configuration values in environment variables or a secure secrets management solution to reduce exposure.

Apply the principle of least privilege to database access

Replace the use of the MySQL root account with a dedicated application database account that has only the permissions required for its operations. This limits damage if the application is compromised.

Use parameterized queries to prevent SQL injection
Implement database access using PreparedStatement or a secure ORM framework. Avoid constructing SQL queries through string concatenation with user-controlled values.

Improve error handling and logging practices
Replace printStackTrace() calls with structured logging. Log detailed errors securely on the server while returning generic error messages to clients to prevent information disclosure.

Harden file upload configuration and limits
Reduce maximum file and request sizes to the minimum required for business needs. Require authentication for any upload functionality, validate file types, and store uploaded files outside the web root.

Add defensive API protections
Implement rate limiting, request size limits, and monitoring to detect and mitigate denial-of-service attempts and automated abuse of API endpoints.

Upgrade and continuously manage third-party dependencies
Upgrade org.bouncycastle:bcprov-jdk15on from version 1.46 to a current, supported release to remediate known vulnerabilities such as CVE-2013-1624 and CVE-2018-5382. Establish regular dependency scanning as part of the build process to identify future vulnerabilities early.