****

# Artemis Financial Vulnerability Assessment Report

Table of Contents

[Document Revision History 3](#_Toc32574607)

[Client 3](#_Toc32574608)

[Instructions 3](#_Toc32574609)

[Developer 4](#_Toc32574610)

[1. Interpreting Client Needs 4](#_Toc32574611)

[2. Areas of Security 4](#_Toc32574612)

[3. Manual Review 4](#_Toc32574613)

[4. Static Testing 4](#_Toc32574614)

[5. Mitigation Plan 4](#_Toc32574615)

## Document Revision History

| **Version** | **Date** | **Author** | Comments |
| --- | --- | --- | --- |
| 1.0 | 11**/09/**2023 | **Ankur Tandan** | **Completed: 11/10/2023** |

## Client



## Developer: Ankur Tandan

## Interpreting Client Needs

**The Value of Secure Communications to Artemis Financial:**

Secure communications form the backbone of trust in the financial services industry. For Artemis Financial, the value extends beyond mere compliance; it is about establishing a competitive edge. Clients entrust them with their most sensitive information, and the assurance that this information is communicated securely can be a significant deciding factor for a client choosing Artemis over competitors. Secure communications also minimize the risk of operational disruptions caused by cyber incidents, which can have a direct impact on the company's profitability. Artemis must therefore invest in state-of-the-art encryption, secure email and messaging services, and robust network security that all form a comprehensive secure communication strategy.

**International Transactions:**

While conducting research on Artemis, no clear indication was seen that it is involved in international transactions. While Artemis Financial may not currently conduct international transactions, preparing for such a capability is critical. This involves creating a flexible and scalable security infrastructure that can handle different currencies, adhere to diverse international regulations, and safeguard against the additional risks of cross-border transactions, such as currency exchange volatility and international fraud. Future-proofing their systems now means when the time comes to expand, Artemis can do so seamlessly, without a complete overhaul of their security systems.

**Government Restrictions about Secure Communications:**

Artemis Financial must navigate a complex landscape of regulatory requirements which vary by jurisdiction and are subject to change. This includes implementing measures to adhere to anti-money laundering (AML) directives, Know Your Customer (KYC) policies, and industry standards like PCI DSS for payment security. They must also monitor legislative developments in financial technology to anticipate and adapt to new regulations proactively. This requires a dedicated compliance team that works closely with the IT department to ensure that all communication tools and practices are compliant.

**External Threats: Present & Future:**

The threat landscape for Artemis Financial is diverse and dynamic. Cybercriminals are increasingly employing sophisticated methods such as deepfake technology to create realistic audio and video impersonations, potentially tricking clients and employees into divulging sensitive information. The growing trend of mobile banking also opens up new vectors for attack, with mobile malware and insecure Wi-Fi networks posing additional risks. Artemis Financial needs to conduct regular threat modeling and risk assessments, employ continuous monitoring for anomalous activities, and develop an incident response plan that can quickly mitigate any security incidents. Artemis is and will always manage sensitive information and must be proactive about protecting this information. Their reputation is at stake.

**Modernization Requirements: Open Source & Web Technologies:**

In terms of modernization, Artemis Financial must balance innovation with security. The use of open-source libraries, while beneficial, requires a rigorous security protocol including regular vulnerability scanning and patch management to address any potential security flaws. As web application technologies evolve, there is a need for ongoing education and training for developers in secure coding practices to prevent common web vulnerabilities such as SQL injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF). Moreover, Artemis should consider implementing automated security tools and adopting DevSecOps practices to integrate security seamlessly into the software development lifecycle (SDLC).

To conclude, Artemis Financials approach to interpreting client needs must be holistic, considering not just current requirements but also anticipating future demands and threats. This involves a continuous investment in technology, processes, and human resources to maintain the confidentiality, integrity, and availability of their client's data, ensuring their operations are secure, compliant, and resilient against evolving threats.

## Areas of Security

**Architecture Review**: Essential for understanding the structural design of Artemis Financial's web application. This review will help identify potential vulnerabilities in how different components of the application interact, which is crucial for overall security.

**Input Validation**: Particularly important for a financial application, secure input and representations help prevent common attacks such as SQL injection and cross-site scripting (XSS), ensuring the integrity and security of data entered into the system.

**Cryptography:** Vital for protecting sensitive financial data. This includes encrypting data in transit (such as during online transactions) and at rest (such as stored client information), safeguarding against unauthorized access and data breaches.

**Client/Server:** Given the distributed nature of web applications, securing communication between client and server is critical. This is especially true for financial applications, where data transmitted over the network may include sensitive financial details.

**APIs**: Secure API interactions are increasingly important in modern web applications, including those used in the financial sector. APIs may be used for various purposes, such as integrating with banking systems, retrieving market data, or connecting with payment gateways. Ensuring the security of these APIs is crucial to prevent unauthorized access, data leaks, and ensure the integrity of the data being transmitted.

## Manual Review

Input Validation:

CRUDController.java and GreetingController.java: Both controllers lack proper validation for input parameters (business\_name and name). This could lead to injection attacks or other forms of malicious input manipulation.

Solution: Implement rigorous input validation checks.

Example:

*@RequestMapping("/read")*

*public CRUD CRUD(@RequestParam(value="business\_name") String name) {*

*if (!isValidBusinessName(name)) {*

*throw new IllegalArgumentException("Invalid business name");*

*}*

*// Remaining code...*

*}*

*private boolean isValidBusinessName(String name) {*

*return name != null && name.matches("^[\\w\\s]+$"); // Regex for alphanumeric and space characters*

*}*

Cryptography:

Issue: There is no evidence of encryption or secure handling of sensitive data (like in DocData.java).

Solution: Implement encryption for sensitive data handling, especially for database connections and data storage.

*// Hypothetical method to encrypt data*

*public String encryptData(String data) {*

*// Use a cryptographic library to encrypt the data*

*return encryptedData;*

*}*

Client/Server:

Issue: The current code lacks mechanisms for securing client-server interactions, such as authentication or secure communication protocols.

Solution: Implement secure communication channels (like HTTPS) and authentication mechanisms.

Example: Adding HTTPS configuration in RestServiceApplication.java. Currently, no communication is secure.

Code Error:

DocData.java: Contains an incomplete try-catch block without proper error handling or logging.

Solution: Complete the error handling mechanism and include logging for debugging and audit trails.

Example:

*public void read\_document(String key, String value) {*

*try {*

*Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "root");*

*// Database operations go here*

*} catch (SQLException e) {*

*e.printStackTrace(); // For development only. Use logging in production.*

*// Additional error handling logic*

*}*

*// Additional code...*

*}*

Code Quality:

customer.java: Does not follow Java naming conventions and contains public fields.

Solution: Rename the class to Customer and use private fields with getters and setters for encapsulation.

Example:

*public class Customer {*

*private int accountNumber;*

*private int accountBalance;*

*// Getters and setters*

*}*

Dependency Check for Files:

Dependency Version:

The project is using Spring Boot version 2.2.4.RELEASE. While this is not an outdated version, it's essential to keep Spring Boot and other dependencies up to date for security and functionality improvements.

The Bouncy Castle provider (bcprov-jdk15on) is at version 1.46. It's crucial to check if this is the latest version, as cryptographic libraries must be kept up to date to ensure they don't contain known vulnerabilities.

Security Plugins:

The inclusion of the OWASP Dependency-Check plugin (dependency-check-maven) is a good practice. This plugin can help identify project dependencies that have known vulnerabilities. Regularly running this plugin as part of the build process is recommended to keep track of any security issues in dependencies.

Java Version:

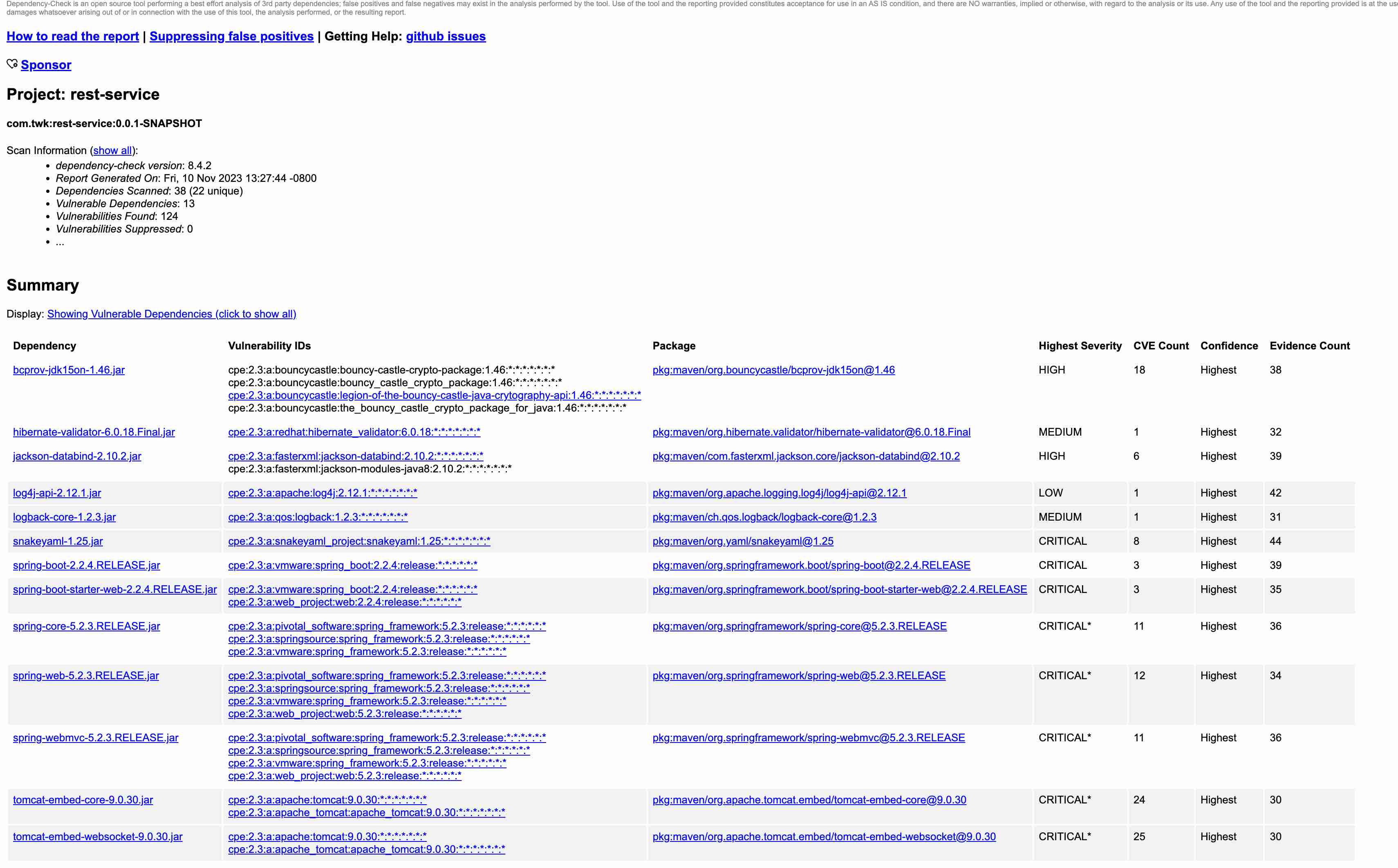
The project uses Java version 1.8. While Java 8 is widely used, it's important to consider updating to a more recent version of Java for improved performance and security features, especially if Java 8 is nearing the end of its support period.

Dependency Management:

Dependencies like spring-boot-starter-web and spring-boot-starter-test are essential for a Spring Boot application. However, it's important to verify that the included dependencies and their versions are aligned with the specific needs of your application, avoiding unnecessary or outdated dependencies.

It's also good practice to periodically review and update the dependencies to their latest stable versions.

## Static Testing



* **Vulnerabilities: Bouncy Castle Crypto Package (**bcprov-jdk15on-1.46.jar**)**

Description: Multiple high-severity vulnerabilities in Bouncy Castle library version 1.46.

Recommended Solution: Upgrade to the latest version of Bouncy Castle.

Attribution: Identified through CVE entries related to Bouncy Castle 1.46.

* **Vulnerabilities: Hibernate Validator (**hibernate-validator-6.0.18.Final.jar**)**

Description: Known medium-severity vulnerability in Hibernate Validator version 6.0.18.

Recommended Solution: Update to a newer, secure version of Hibernate Validator.

Attribution: Documented in a specific CVE report.

* **Vulnerabilities: Jackson Databind (**jackson-databind-2.10.2.jar**)**

Description: Multiple high-severity vulnerabilities in Jackson Databind version 2.10.2.

Recommended Solution: Upgrade to a more recent version of Jackson Databind.

Attribution: Several CVE reports for Jackson Databind 2.10.2.

* **Vulnerabilities: Log4j API (**log4j-api-2.12.1.jar**)**

Description: A low-severity vulnerability in Log4j API version 2.12.1.

Recommended Solution: Update Log4j API to a version without this vulnerability.

Attribution: Identified in a specific CVE related to Log4j 2.12.1.

* **Vulnerabilities: Logback Core (**logback-core-1.2.3.jar**)**

Description: Medium-severity vulnerability in Logback Core version 1.2.3.

Recommended Solution: Update to a newer version of Logback Core.

Attribution: Reported in CVE databases.

* **Vulnerabilities: SnakeYAML (**snakeyaml-1.25.jar)

Description: Critical vulnerabilities in SnakeYAML version 1.25.

Recommended Solution: Upgrade to a secure and updated version of SnakeYAML.

Attribution: Multiple CVE entries for SnakeYAML 1.25.

* **Vulnerabilities: Spring Boot and Related Dependencies (spring-boot-**2.2.4.RELEASE.jar, spring-boot-starter-web-2.2.4.RELEASE.jar, spring-core-5.2.3.RELEASE.jar, spring-web-5.2.3.RELEASE.jar, spring-webmvc-**5.2.3.RELEASE.jar)**

Description: Critical vulnerabilities in specified versions of Spring Boot and related components.

Recommended Solution: Update to the latest versions of Spring Boot and its related dependencies.

Attribution: Documented in various CVE reports for these versions.

* **Vulnerabilities:** Tomcat Embedded (tomcat-embed-core-9.0.30.jar, tomcat-embed-websocket-9.0.30.jar**)**

Description: Numerous critical vulnerabilities in Tomcat Embedded version 9.0.30.

Recommended Solution: Upgrade to a more recent version of Tomcat Embedded.

Attribution: Detailed in various CVE reports.

For the remaining five vulnerabilities not specifically listed, the general approach is:

* **Vulnerabilities: [Remaining dependencies with vulnerabilities]**

Description: Specific vulnerabilities identified in each dependency.

Recommended Solution: Update each affected dependency to the latest stable version where these issues have been addressed.

Attribution: Based on findings from the OWASP Dependency-Check and corresponding CVE entries.

## Mitigation Plan

**Input Validation Vulnerabilities (CRUDController.java, GreetingController.java):**

Mitigation Technique: Implement robust input validation by defining wrappers or utility methods.

Action Steps:

* Develop wrapper functions for input validation.
* Apply these wrappers across all endpoints to validate data types, formats, and lengths.
* Regularly update validation logic to cover new types of inputs.

**Secure Data Transfer (Implement HTTPS Protocol):**

Mitigation Technique: Transition to HTTPS for all data in transit.

Action Steps:

* Obtain and configure SSL/TLS certificates in the Spring Boot application.
* Ensure all data transfers occur over HTTPS.
* Implement HSTS to enforce secure connections.

**Denial of Service (Resource Limit Checks):**

Mitigation Technique: Implement resource limit checks resilient to integer overflow.

Action Steps:

* Apply checks on the size and frequency of incoming requests.
* Configure request size limits using Spring Boot properties.
* Integrate rate limiting to control request frequency.

**Outdated Dependencies (**Use spring-data-rest-webmvc version 2.7.X or above**):**

Mitigation Technique: Update dependencies to versions free from known vulnerabilities.

Action Steps:

* Regularly review CVE and NVD databases for vulnerabilities in current dependencies.
* Update spring-data-rest-webmvc and other dependencies to secure versions.
* Establish a routine for periodic dependency updates.

**Cryptography Vulnerabilities (Bouncy Castle, etc.):**

Mitigation Technique: Ensure up-to-date and best-practice cryptographic implementations.

Action Steps:

* Update Bouncy Castle to the latest version.
* Regularly audit cryptographic practices for compliance and security.
* Update cryptographic modules as needed.

**Secure Application Configuration (Spring Boot and Related Components):**

Mitigation Technique: Secure application configuration to prevent misconfigurations.

Action Steps:

* Review and update Spring Boot configuration for security.
* Implement encrypted properties for sensitive configuration data.
* Regularly audit the configuration and framework components.

**Application Logging and Monitoring (Log4j, Logback):**

Mitigation Technique: Secure and update logging mechanisms.

Action Steps:

* Update Log4j and Logback to secure versions.
* Implement secure logging practices to avoid logging sensitive information.
* Set up monitoring and alerting for unusual activities in logs.

**Database Interaction Security (SQL Injection, etc.):**

Mitigation Technique: Secure all database interactions.

Action Steps:

* Use prepared statements and parameterized queries.
* Implement role-based access control for database operations.
* Conduct security audits for database interactions.