CNC 419 IT Security Risk Management— Final Project

GRC

PCI DSS compliance

—

BY :

Gehad Abdellah : 320220237

Mark Labib : 320220212

Yasser Mohamed : 320220211

TABLE OF CONTENT

INTRODUCTION

The PCI DSS Requirement 6 Compliance Checker is an automated security analysis tool designed to identify vulnerabilities in payment processing code. In today's digital economy, where billions of financial transactions occur daily, ensuring the security of payment card data is paramount. This tool addresses a critical need in the software development lifecycle by providing immediate feedback on code security compliance.

1. Problem statement

Payment card breaches continue to cost organizations millions of dollars annually. According to industry reports, the average cost of a data breach in 2024 exceeded $4.5 million. Many breaches occur due to common coding vulnerabilities such as:

- Hardcoded credentials and API keys
- SQL injection vulnerabilities
- Weak cryptographic implementations
- Insecure data handling practices

Traditional security audits are time-consuming, expensive, and often occur too late in the development cycle. Developers need real-time feedback to catch security issues before they reach production.

2. Solution overview

1. **Machine Learning Classification**: A trained model that understands code context
2. **Pattern-Based Detection**: Rule-based regex patterns for known vulnerabilities
3. **Scoring System**: Quantitative assessment (1-5 scale) for actionable insights
4. **Dual Interface**: Both API and command-line access for different workflows

The tool provides immediate, actionable feedback that helps developers write secure code while learning security best practices.

OBJECTIVE

1. Primary objectives

The primary objectives clearly outline the basic aim of the project and help in its development. The basic aims and objectives of the proposed system include automating the process of identifying security issues, complying with industry standards, and giving valuable development inputs to developers. The proposed system targets improving code security efficiently and promoting healthy development practices, which are based on security.

This project's primary objectives are:

1. Automate Security Analysis: Without requiring manual review, provide immediate feedback on code security compliance.
2. PCI DSS Alignment: Pay particular attention to infractions of PCI DSS Requirement 6 (Secure Development).
3. Educational Value: Provide thorough recommendations to help developers comprehend security issues.
4. Ready for Integration: Design for simple integration with development workflows and CI/CD pipelines
5. Reduce false positives while keeping a high level of sensitivity to actual vulnerabilities for accurate detection.

2. The intended audience

The audience described in this section is the group of individuals who will benefit from this tool. Knowing the audience allows the project's features and functionality to be better suited to the needs of individuals who will be developing and reviewing secure code and individuals who educate others on developing safely.

This tool is useful for:

1. Software developers: Creating code for processing payments
2. Security engineers: Performing security audits and code reviews
3. DevOps Teams: Including security audits in deployment procedures

4. Compliance Officers: Confirming adherence to PCI DSS Requirement 6
5. Educational establishments: instructing students in safe coding techniques

3. The intended audience

The criteria of project success enumerate a set of results, which ultimately ascertain if a tool can meet an objective. The criteria serve as benchmarks to check accuracy, efficiency, integration, and overall effectiveness, thereby ensuring not only if a project can serve a purpose, but whether a project can add value to a software development process.

The project is deemed successful if it

1. detects common security flaws with >85% accuracy
2. offers practical suggestions for resolving infractions
3. takes less than two seconds to process requests for code analysis.
4. smoothly integrates with current development processes
5. minimizes security flaws in codebases that have been reviewed

THEORETICAL BACKGROUND

1. PCI DSS overview

The Payment Card Industry Data Security Standard, or PCI DSS, is a comprehensive framework through which security standards are set up and meant to protect payment card information throughout its life cycle. It applies to any entity that accepts, processes, stores, or transmits credit card information of any type. By imposing consistent security practices, PCI DSS supports organizations in the prevention of data breaches, fraud reduction, and upkeep of consumer trust.

PCI DSS was developed collaboratively by major credit card brands including Visa, Mastercard, American Express, Discover, and JCB. It provides a set of mandatory controls, policies, and procedures to ensure the confidentiality, integrity, and availability of cardholder data.

The standard has been organized into 12 broad requirements; among these are network security, access control, monitoring, and secure software development. Of these, Requirement 6: Secure Development focuses on how systems and applications should be developed to be secure and maintained in such a manner that precludes malicious exploitation.

2. PCI DSS Requirement 6: Secure Development

Requirement 6 drives the emphasis on securely developing software and maintaining secure systems throughout the life cycle. This is considered crucial because software vulnerabilities could become the prime avenue of attack for the attacker. The requirement breaks down into several key areas:

3.2.1 Security Vulnerabilities

Organizations should work proactively to identify and fix all security vulnerabilities in various systems and applications. This will include:

- Regularly perform scans on the vulnerability in the software components.

- Install critical security patches within one month of release. Preventing exploitation by doing so.

3.2.2 Secure Software Development

Secure coding practices must be closely associated with the development lifecycle to prevent vulnerabilities from being introduced. This means:

- Following secure coding guidelines that address common vulnerabilities such as SQL injection, cross-site scripting (XSS), and buffer overflows.
- Cleaning up development and test accounts before applications are released to production.
- Also, review all custom code for potential security issues before deployment.

3.2.3 Protection of Production Data

- Organizations should: prevent sensitive data exposure when developing and testing.
- Remove all test data from systems before placing them into production.
- Remove test accounts and credentials that could allow unauthorized access.

3.2.4 Change Control

All changes to systems and applications should be appropriately controlled and documented in order to minimize risk. Some of the key practices include:

- Following formal change control procedures for all software and configuration changes.
- Documenting the effects of changes on system security.
- Perform testing of all security patches before deployment to a production environment.

3.2.5 Fixing Common Weaknesses

Requirement 6 also outlines that organizations should prevent and mitigate common types of software vulnerabilities, which include:

- Injection flaws (SQL, OS, LDAP, etc.)
- Buffer overflow
- Insecure cryptographic storage

- Authentication and session management-related vulnerabilities
- Cross-site scripting (XSS)
- Improper Error Handling
- Cardholder data storage insecure

With these practices in place, organizations minimize the chances of security breaches while at the same time maintaining PCI DSS compliance.

SYSTEM DESIGN & ARCHITECTURE

1. High-Level Architecture

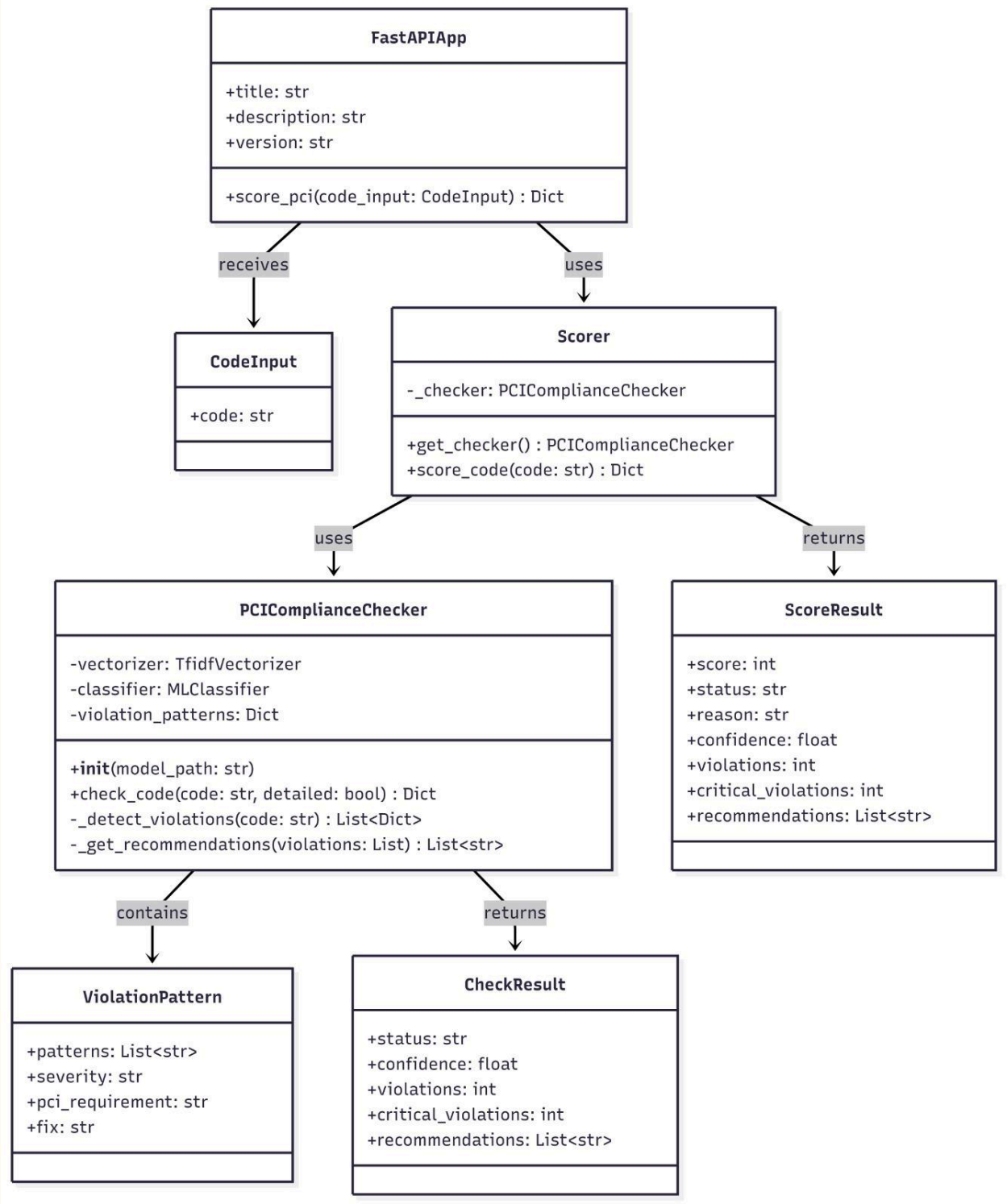
The system has been designed on a modular, console-based architecture that leads to simplicity, maintainability, and secure handling of source code. It separates the concerns into modules in such a way that automated PCI DSS compliance analysis can be performed with no dependence on a graphical user interface.

The main modules are as follows:

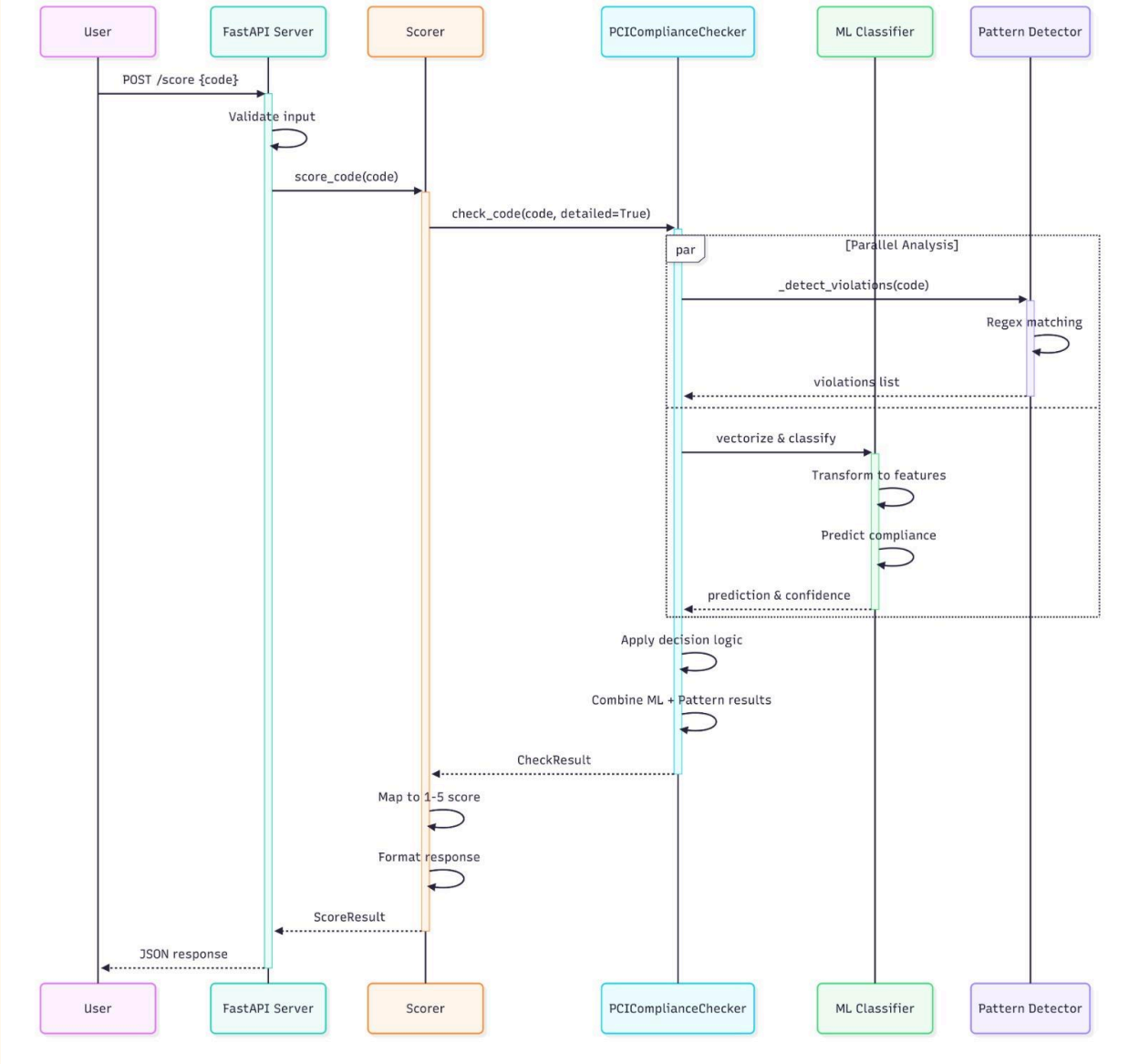
1. **Module: CLI/Console Interface**
    - Takes in code submissions via the command line through file paths or direct input.
    - Provides text-based feedback, including: compliance scores, vulnerabilities detected, and recommended fixes.
    - Supports command-line arguments for batch processing, directory scans, or integration into automated CI/CD pipelines.
2. **Code Analysis Engine**
    - This module performs static code analysis to identify insecure coding practices that may lead to PCI DSS violations.
    - Supports rule-based scanning-e.g., detecting SQL injection, XSS, insecure storage-and can optionally include ML-based analysis for more complex scenarios.
    - Produces structured output from assessment and reporting.
3. **Compliance Scoring Module**
    - Computes a score, such as from 1 to 5, of the PCI DSS compliance based on identified vulnerabilities and implementation of the Requirement 6.
    - Outputs text-based or file-based reports, comprising detailed descriptions of issues, including the lines of code that were affected, and remediation guidance.
4. **Rules & Knowledge Base Module**
    - Contains predefined PCI DSS rules and secure coding guidelines.

- Can be updated independently about the evolution in standards related to the PCI DSS or new patterns of vulnerabilities.

5. **Data Storage Module**
    - Securely stores the scan results, logs, and historical code analysis reports locally or in a database.
    - Ensures encryption at rest if sensitive code or credentials are stored.
    - Maintains audit logs of all scans and user commands.
6. **Integration/Automation Layer**
    - Supports automation via scripts and integrations with CI/CD pipeline.
    - Enables batch scanning of more than one source code file or repository.
    - Returns results in a format like JSON, for example, machine-readable for further processing or reporting.



IMG 1 : Class diagram



IMG2 : Sequence diagram

2. Component Interaction

The workflow of the console-based system is linear and pipeline-oriented:

1. The user runs a console command with a source code file or directory as an argument.
2. This is scanning for vulnerabilities by the Code Analysis Engine.
3. The detected issues are evaluated against the Rules & Knowledge Base.

4. The Compliance Scoring Module generates a score and positions recommendations.
5. Results are outputted to the console and optionally written to a file or database.

This design assures simplicity and automation while remaining fully compatible with scripting and CI/CD processes.

3. Security Considerations

Even without a GUI, the system implements security best practices:

1. Principle of Least Privilege: Console operations run with the minimum permissions necessary.
2. Secure Storage: Sensitive data, such as analysis logs or scanned code, is securely stored and encrypted if necessary.
3. Input Validation: File paths and code are subject to validation against malicious exploitation.
4. Audit Logging: Every scan and every command run by the CLI is logged for accountability.
5. Environment Isolation: Scans run in separated processes to avoid executing possibly malignant code by accident.

This version aligns it with a console-based application by emphasizing automation, security, and modularity while removing GUI dependencies.

IMPLEMENTATION DETAILS

1. File structure

The system is implemented as a **console-based Python application** with a modular file structure to separate concerns, improve maintainability, and facilitate future development. The following structure outlines the main components:

1. Main Script: `main.py`

- **Purpose:** Serves as the entry point of the application.
- **Responsibilities:**
  - Accepts file paths or directories via command-line arguments.
  - Initiates the code analysis engine.
  - Invokes the compliance scoring module.
  - Displays results on the console and optionally saves reports to a file.
- **Example Usage:**

```
python3 main.py --file example_code.py

python3 main.py --dir ./projects/
```

2. Code Analysis Engine: `pci_compliance_checker.py`

- **Purpose:** Performs static analysis of submitted source code to detect security vulnerabilities.
- **Responsibilities:**
  - Scans source code for insecure practices such as SQL injection, XSS, and insecure storage.
  - Extracts relevant information for scoring.
  - Interfaces with the rules module for compliance verification.
- **Implementation:** Contains functions for parsing code, identifying patterns, and preparing results for the scoring module.

3. Compliance Scoring Module: `scorer.py`

- **Purpose:** Calculates a **PCI DSS compliance score** based on detected vulnerabilities.
- **Responsibilities:**
  - Assigns severity levels to detected issues.
  - Computes a cumulative compliance score (e.g., 1–5).
  - Generates detailed textual feedback and remediation recommendations.
- **Integration:** Works closely with `pci_compliance_checker.py` to process analysis results.

4. Rules & Knowledge Base: `rules.py`

- **Purpose:** Contains **predefined PCI DSS rules and secure coding guidelines**.
- **Responsibilities:**
  - Provides functions to verify whether code adheres to Requirement 6.
  - Maintains a catalog of common vulnerabilities, including SQL injection, buffer overflow, and XSS patterns.
  - Supports updates to include new rules or updated compliance standards.

5. Datasets: `datasets/`

- **Purpose:** Stores sample code files used for testing and training (if ML is incorporated).
- **Structure:**
  - `compliant/` – Contains code that follows PCI DSS secure development guidelines.
  - `non_compliant/` – Contains code with known vulnerabilities to test detection accuracy.
- **Usage:** Used in `test_scorer.py` for verifying system accuracy and scoring reliability.

6. RAG Model: `RAG/pci_rag_model.pkl`

- **Purpose:** Stores a pre-trained **Retrieval-Augmented Generation (RAG) model** for evaluating code compliance.
- **Responsibilities:**

- Enhances detection of subtle security violations that may not be captured by rule-based scanning.
- Provides recommendations and guidance for partial compliance.

7. Test Script: `test_scorer.py`

- **Purpose:** Provides unit tests and validation for the scoring module.
- **Responsibilities:**
  - Ensures that `scorer.py` correctly evaluates compliant and non-compliant code.
  - Helps maintain reliability as the system evolves.

8. Requirements: `requirements.txt`

- **Purpose:** Lists all Python dependencies needed to run the application.
- **Typical Packages:**
  - `pandas` – For data handling.
  - `scikit-learn` – If ML scoring is used.
  - `regex` – For pattern matching in code analysis.
  - `joblib` – For loading the RAG model.

9. Cache Directory: `__pycache__/`

- Automatically created by Python to store compiled bytecode for faster execution.
- Does not require manual management but is part of the project directory.

2. Workflow overview

The system executes the following steps when analyzing code:

1. **Input Handling:**
   The user provides a file or directory via the console.
2. **Code Parsing:**
   `pci_compliance_checker.py` parses each source code file and identifies potential vulnerabilities.

3. **Rule Checking:**

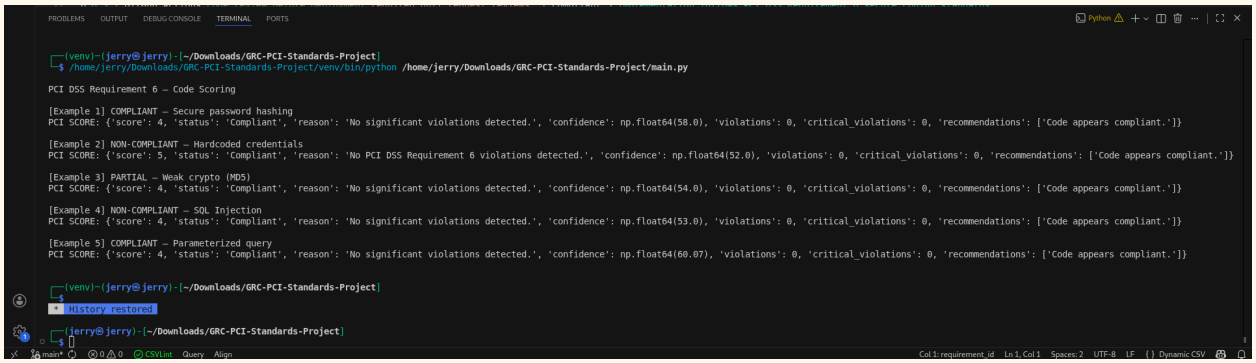
   The engine compares code against PCI DSS rules stored in `rules.py`.

4. **Scoring:**

   `scorer.py` calculates a compliance score based on the detected issues and severity levels.

5. **Output:**

   Results are displayed on the console and optionally saved in a report file, detailing vulnerabilities and suggested remediation steps.



Img4: Output sample

LIMITATIONS & FUTURE IMPROVEMENT

Even though the scoring system under the PCI DSS Requirement 6 offers a computerized and efficient mechanism for code compliance evaluation, some restrictions and future improvements are considered. This chapter will identify the current shortcomings and discuss future improvements.

1. Limitations

1. Limited Programming Language Support

Current State: Currently, the system supports Python code mostly.

Limitation: Organizations may utilize various programming languages (Java, C++/Java, JavaScript, etc.) to code applications. It is not possible to analyze code in other programming languages other than Python, i.e., code in other programming languages cannot be analyzed by this system until additional

2. Limited Vulnerabilities

Current State: The module is concerned with generic PCI DSS Requirement 6 risks such as SQL injection, cross site scripting, poor crypto, and hard-coded credentials.

Shortcoming: It does not currently handle more complex vulnerabilities like insecure usage of APIs, race conditions, or authentication bypassing.

2. Future improvements

1. Multi-Language Support

- Extend the analysis engine to **support multiple programming languages** commonly used in enterprise applications.
- Implement language-specific rules for Java, C/C++, JavaScript, and others to broaden applicability.

2. Dynamic and Runtime Analysis

- Integrate **dynamic code analysis** tools or sandboxed runtime testing to detect vulnerabilities that appear only during execution.
- This would improve detection of logic errors, insecure API calls, and memory-related vulnerabilities.

3. Expanded Vulnerability Coverage

- Add detection capabilities for **advanced and emerging threats**, such as:
    - Insecure API integration
    - Race conditions and concurrency issues
    - Advanced cryptographic misuse (e.g., ECB mode, weak key management)
- Regularly update the **rules module** to align with evolving PCI DSS standards and security best practices.

4. Improved Scoring and Customization

- Develop a **configurable scoring system** that allows organizations to weigh vulnerabilities differently according to risk appetite.
- Implement **explainable scoring**, providing detailed reasoning for each score to support audits and decision-making.

CONCLUSION

In conclusion, the PCI DSS Requirement 6 scoring system is an automatic and effective method for evaluating source code security with respect to compliance standards that have been laid down. The system integrates static code analysis, rule-based vulnerability detection, and a scoring method in a way that developers and security teams can quickly spot the security issues that may be there, and check their compliance with the PCI DSS secure development standards. The project proves how automation can really help by reducing the need for human to human code reviews, giving instantaneous feedback, and making the necessary areas for fixing visible. The scoring, analysis, and rules parts of the system are separately designed and this modular and extensible design provides maintainability, scalability, and adaptability for future improvements. The system can operate in terminal and API modes where the supportive educational purpose is through sample code and at the same time, it is being integrated into the automated Continuous Integration/Continuous Deployment (CI/CD) pipelines. The system, on the other hand, has certain limitations like it can only work with Python, it is limited to static analysis, and it only partially covers complex vulnerabilities, which should all be considered carefully in the project’s deployment. However, this is a strong base laid for the future improvements like multi-language support, dynamic analysis, batch processing, enhanced scoring, and machine learning-based detection. In a nutshell, this scoring system automates the whole toolchain of secure coding practices, compliance assessment and strengthens the application's security posture that is handling sensitive cardholder data thereby giving the organization a reliable tool for PCI DSS compliance.