AI-Based Code Security Analyzer

Table of Content

Project Documentation

1. Project Overview

The AI-Based Code Security Analyzer is a web application that allows users to upload source code files in various programming languages. The system automatically identifies the programming language based on the file extension, scans the code for potential security vulnerabilities and issues, and provides detailed recommendations for remediation.

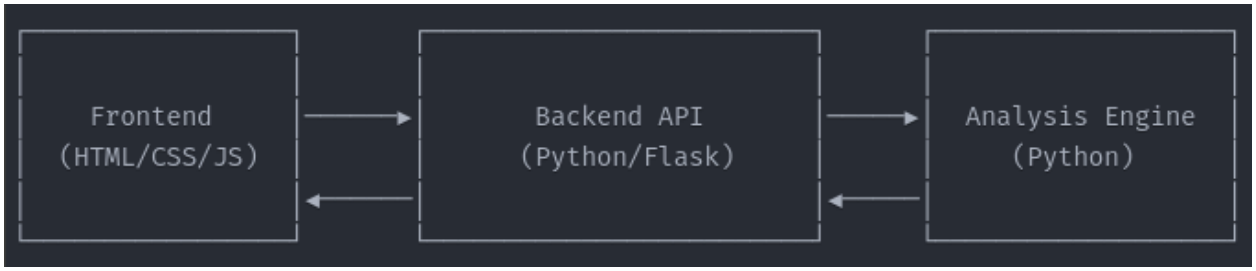
2. Project Architecture

2.1 High-Level Architecture

The application follows a client-server architecture:

- **Frontend**: HTML, CSS, and JavaScript for the user interface
- **Backend**: Python with Flask framework for handling file uploads, language identification, and code analysis
- **AI Component**: Utilizes pattern matching, rule-based systems, and machine learning models to identify security vulnerabilities

2.2 System Components



3. Features and Functionality

3.1 Core Features

1. **File Upload**: Users can upload source code files through a drag-and-drop interface or file browser.
2. **Language Detection**: The system automatically detects the programming language based on file extension.
3. **Code Analysis**: The backend scans the uploaded code for common security vulnerabilities.
4. **Vulnerability Report**: A comprehensive report is generated detailing found issues, severity levels, and remediation steps.
5. **Code Highlighting**: Vulnerable code sections are highlighted with detailed explanations.

3.2 Supported Languages

Initial support includes common programming languages:

- JavaScript (.js)
- Python (.py)
- Java (.java)
- C/C++ (.c, .cpp)
- PHP (.php)
- Ruby (.rb)
- Go (.go)

4. Technology Stack

4.1 Frontend

- HTML5 for structure
- CSS3 for styling
- JavaScript (ES6+) for interactivity
- Bootstrap for responsive design
- Highlight.js for code syntax highlighting
- Chart.js for visualization of vulnerability statistics

4.2 Backend

- Python 3.9+
- Flask web framework
- Flask-Cors for handling cross-origin requests
- Werkzeug for file handling
- NumPy and Pandas for data processing

4.3 Security Analysis Tools

- Custom rule-based pattern matching for basic vulnerability detection
- Language-specific analyzers (Bandit for Python, ESLint for JavaScript, etc.)
- OWASP vulnerability database integration

5. Implementation Approach

5.1 Language Identification

The system identifies programming languages primarily through file extensions, with a fallback mechanism that analyzes code patterns when extensions are ambiguous or missing.

```python
def identify_language(filename):
    extension = filename.split('.')[-1].lower()

    language_map = {
        'py': 'Python',
        'js': 'JavaScript',
        'java': 'Java',
        'c': 'C',
        'cpp': 'C++',
        'php': 'PHP',
        'rb': 'Ruby',
        'go': 'Go',
        # Add more languages as needed
    }

    return language_map.get(extension, 'Unknown')
```

5.2 Vulnerability Detection Methodology

The system employs a multi-layered approach:

1. **Pattern Matching**: Uses regular expressions and AST (Abstract Syntax Tree) parsing to identify known vulnerable patterns.
2. **Static Analysis**: Language-specific static analysis tools are integrated for deeper inspection.
3. **Rule-Based Detection**: Custom rules based on OWASP Top 10 and common security practices.
4. **Machine Learning Classification**: For languages with sufficient training data, ML models help identify potential vulnerabilities.

5.3 Common Vulnerability Categories

The system detects various vulnerability types including:

- SQL Injection
- Cross-Site Scripting (XSS)
- Command Injection

- Insecure Deserialization
- Authentication Issues
- Authorization Flaws
- Sensitive Data Exposure
- Security Misconfigurations
- Broken Access Control
- Using Components with Known Vulnerabilities

6. User Interface Design

6.1 Main Components

1. **Upload Area**: Drag-and-drop interface with file type restrictions
2. **Analysis Dashboard**: Displays summary of found vulnerabilities
3. **Code Viewer**: Shows uploaded code with highlighted vulnerable sections
4. **Detailed Report**: Lists all found issues with severity and remediation steps

6.2 UI Workflow

1. User uploads code file
2. System displays analysis progress
3. Dashboard shows vulnerability summary
4. User can interact with the code viewer to see detailed explanations
5. Full report can be exported as PDF or JSON

7. API Endpoints

7.1 Backend API Routes

Endpoint	Method	Description
`/api/upload`	POST	Upload a code file for analysis
`/api/analyze`	POST	Analyze already uploaded file
`/api/report/<file_id>`	GET	Retrieve analysis report
`/api/languages`	GET	Get list of supported languages

8. Security Analysis Process

8.1 Processing Steps

1. **Code Parsing**: Parse the code into an AST or tokenize for pattern matching
2. **Rule Application**: Apply language-specific security rules
3. **Vulnerability Identification**: Match code patterns against known vulnerability signatures
4. **Context Analysis**: Consider surrounding code to reduce false positives
5. **Severity Assignment**: Assign severity levels based on vulnerability type and impact
6. **Recommendation Generation**: Generate remediation steps for each issue

8.2 Example: Python Code Analysis

For Python files, the system checks for vulnerabilities like:

- Use of `eval()` or `exec()` with user input (command injection)
- SQL queries formed with string concatenation (SQL injection)
- Use of `pickle` without proper validation (insecure deserialization)
- Hard-coded credentials or API keys (sensitive data exposure)
- Use of outdated crypto libraries (cryptographic failures)

9. Implementation Plan

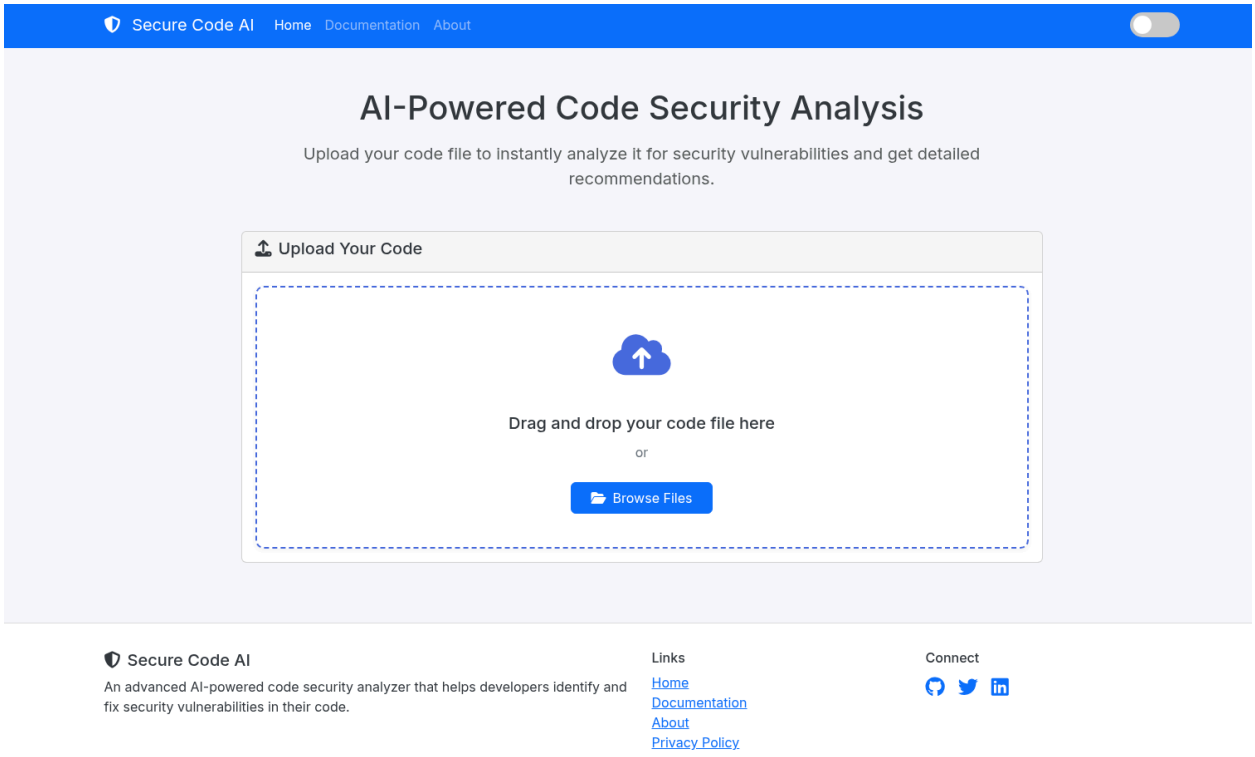
9.1 Development Phases

1. **Phase 1**: Basic application setup (frontend and backend)
    - File upload capability
    - Language detection
    - Simple pattern-based vulnerability detection for Python and JavaScript
2. **Phase 2**: Enhanced analysis capabilities
    - Integrate language-specific analyzers
    - Improve vulnerability detection accuracy
    - Add detailed remediation guidance
3. **Phase 3**: Advanced features
    - Add machine learning-based detection
    - Support for more languages
    - Code fix suggestions

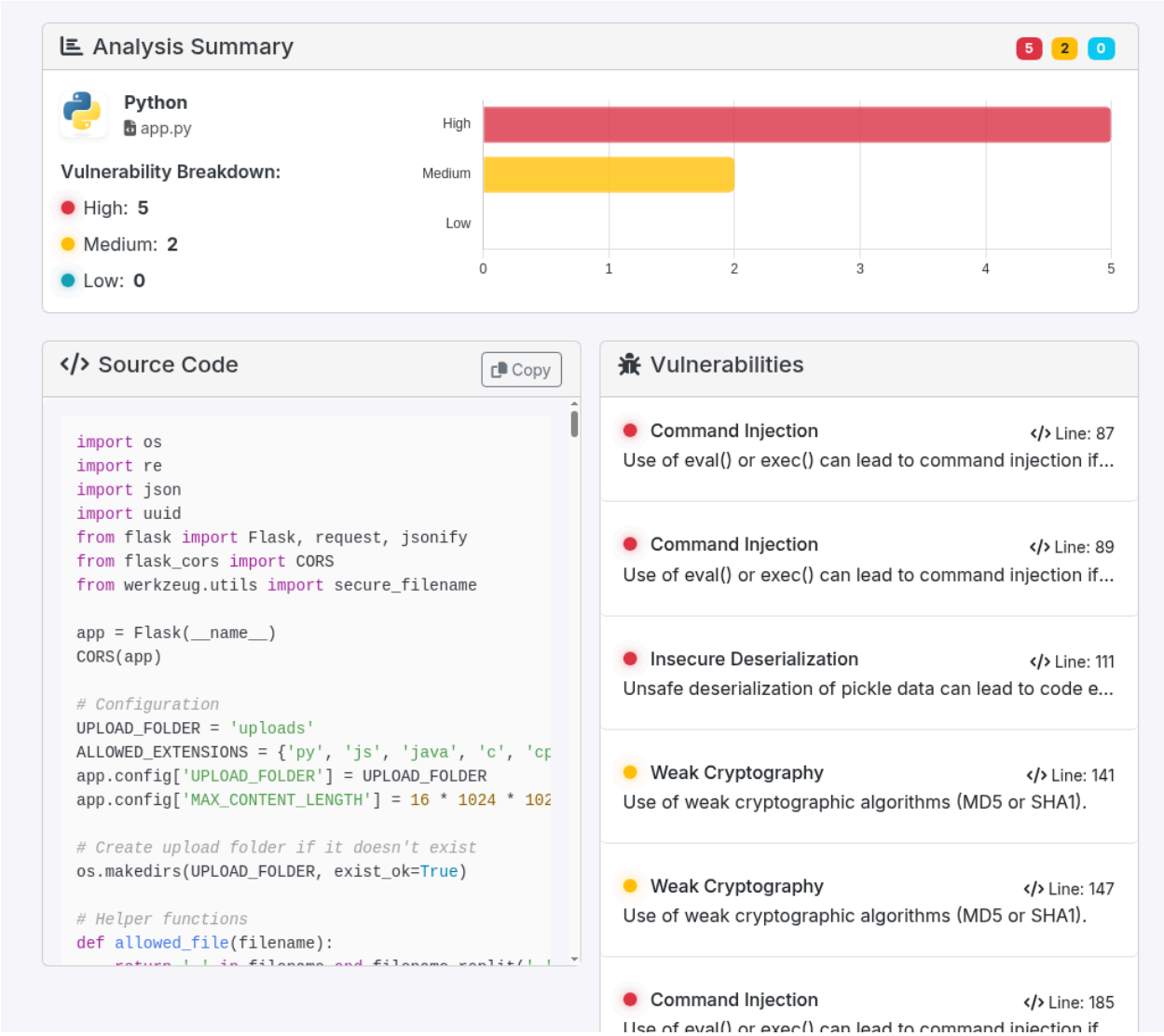
9.2 Timeline (Estimated)

Phase	Duration	Key Deliverables	Completion
1	2-3 weeks	Functional MVP with basic analysis	Done
2	3-4 weeks	Comprehensive vulnerability detection	Future Work
3	4-6 weeks	Advanced features and optimizations	Futute Work

10. Output



After Uploading File



11. Evaluation and Testing

11.1 Testing Strategy

- **Unit Testing**: Test individual components for functionality
- **Integration Testing**: Ensure components work together as expected
- **Benchmark Testing**: Test against known vulnerable codebases (e.g., OWASP Benchmark)
- **Usability Testing**: Ensure UI is intuitive and reports are understandable

11.2 Evaluation Metrics

- **Detection Accuracy**: True positive rate, false positive rate

- **Performance**: Analysis time per file size/complexity
- **Usability**: User satisfaction ratings and task completion rates

12. Future Enhancements

- **Real-time Analysis**: Analyze code as it's being written
- **Integration with IDEs**: Plugins for popular IDEs like VS Code, JetBrains IDEs
- **CI/CD Integration**: Integrate with popular CI/CD pipelines
- **Code Fixing**: Automatic generation of fixed code versions
- **Natural Language Processing**: More intuitive explanation of vulnerabilities

13. Conclusion

The AI-Based Code Security Analyzer provides developers with an accessible tool to identify and remediate security vulnerabilities in their code. By supporting multiple programming languages and providing detailed remediation guidance, the system helps improve code quality and security across development projects.