DD2525 - Language Based Security

Vulnerability detection and CSP generation

Christoforos Seas, Juan Trillo Carreras (Group 49)

1. Introduction

This report details the "Vulnerability Detection, CSP Generation, and Static Analysis" project. The project encompasses three primary components aimed at enhancing web application security:

1. A **CSP (Content Security Policy) Generation Tool**: A web-based interface for an existing Python script (`generate_csp.py`), allowing users to generate CSPs for their projects through the application.
2. A **Vulnerability Detection Tool (DeepSeek Integration)**: A web-based platform designed to scan React, Vue, and Node.js projects for potential security weaknesses, with a planned integration of the DeepSeek API.
3. **Static Analyzer Tools for Vulnerability Detection**: A suite of standalone Python scripts (`security-eval.py`, `parse-modules.py`) for local static code analysis and dependency vulnerability checking with an OSV API.

The web application components (CSP Generator and DeepSeek Vulnerability Detector) are built using Next.js, TypeScript, and Tailwind CSS.

2. CSP (Content Security Policy) Generation

2.1. Objective

To develop a robust tool capable of generating Content Security Policies for web projects, offering customization through various parameters to suit different deployment environments and security postures.

2.2. `generate_csp.py`: The CSP Generation Engine

The core of this functionality is the Python script `generate_csp.py`, developed as part of this project. This script is designed to analyze web project structures, identify resources such as scripts, stylesheets, images, and connection sources, and then construct appropriate CSP directives.

Key functionalities and configurable parameters of `generate_csp.py` include:

- **Target Project Path**: Users specify the root directory of the web project to be analyzed.
- **HTML Integration**: An option to directly embed the generated CSP meta tag into a specified HTML index file.
- **Index File Path**: The path to the HTML file where the CSP tag should be inserted or updated.
- **Old CSP Deletion**: A feature to automatically remove any pre-existing CSP meta tags from the HTML file before adding the new one.
- **CSP Type**: Users can choose between generating a 'production' (typically stricter) or 'development' (potentially more permissive for debugging) CSP.

2.3. Web Interface Integration

To provide a user-friendly way to utilize the `generate_csp.py` script, a dedicated section within the web application was developed. This interface allows users to:

- Input all the necessary parameters for the `generate_csp.py` script through a simple form.
- Initiate the CSP generation process with a button click.

- View the output of the Python script directly in the browser, including any generated CSPs or messages from the script. The terminal output, including ANSI color codes, is converted to HTML for clear presentation.

2.4. Process Flow

The generation process via the web interface follows these steps:

1. The user provides the required parameters (project path, HTML index path, CSP type, etc.) through the form on the "Generate CSP" page in the Next.js application.
2. Upon submission, these parameters are sent to a backend API route (`/api/generate-csp`).
3. The API route then constructs the appropriate command-line arguments and executes the `generate_csp.py` Python script using Node.js's `child_process.execFile`.
4. The `generate_csp.py` script runs, performs its analysis, and generates the CSP.
5. The raw output (both stdout and stderr) from the Python script is captured by the Node.js backend.
6. This output, after processing (e.g., converting ANSI terminal color codes to HTML `<span>` elements with inline styles), is sent back to the frontend and displayed to the user.

3. Vulnerability Detection with DeepSeek (Web Interface)

3.1. Objective

To provide users with an online tool to upload their React, Vue, or Node.js project files and receive a vulnerability assessment. This component is designed for future integration with the DeepSeek API for its core scanning logic.

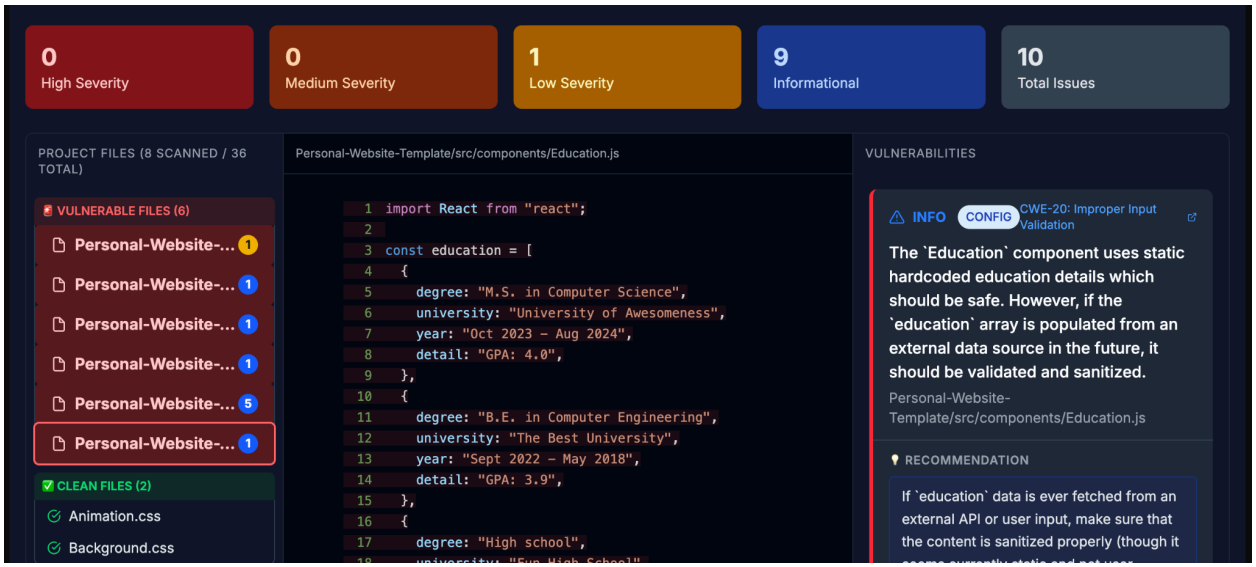
3.2. User Interface & Workflow Overview

Users can select their project type (React, Vue, Node.js) and upload their project folder. The application intelligently excludes common non-source directories (e.g., `node_modules`, `.git`) and specific lock files (`package-lock.json`, `yarn.lock`). Files are displayed in a three-panel layout:

- **Sidebar**: Lists all project files, categorized as "Vulnerable," "Clean," or "Not Scanned."
- **Code Viewer**: Displays the content of the selected file.
- **Details Panel**: Shows vulnerability information for the selected file.

Scans can be performed on batches of files or on individual files. A queuing mechanism manages concurrent scan requests, ensuring stability. Summary statistics (vulnerability counts by severity) are displayed upon scan completion.

The vulnerabilities are categorized in 4 categories: High, Medium and Low severity, and Informational. Informational vulnerabilities are not security vulnerabilities as such, but possible vulnerabilities or even something more general.



3.3. Process Flow

The frontend application sends project files (either in batches or individually) to a Next.js API route (`/api/scan`). This route is designed to communicate with a `DeepSeekService` module, which encapsulates the logic for interacting with the external scanning engine.

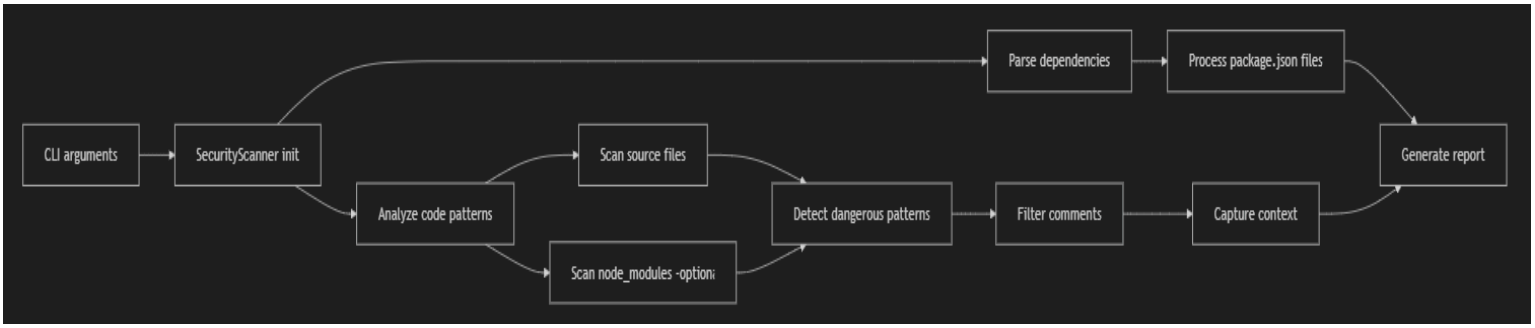
- **Current Status**: The `DeepSeekService` currently utilizes **mock data** to simulate scan results. The full integration with the actual DeepSeek API for live vulnerability analysis is a key planned future development.

4. Static Analyzer Tools for Vulnerability Detection (Python Scripts)

As a distinct part of this project, two standalone Python scripts were developed to perform local static security analysis on projects. These tools operate independently of the web application but serve a complementary security assessment role.

4.1. Dangerous code pattern analyzer: `security-eval.py`

- **Objective**: To scan project source code for potentially dangerous or risky code patterns that might indicate security vulnerabilities.
- **Functionality**:
    - Scans project directories for JavaScript, TypeScript, Vue, and HTML files.
    - Includes an option to scan code within `node_modules` if specified.
    - Identifies occurrences of predefined risky code patterns (e.g., use of `eval()`, direct `innerHTML` manipulation, `child_process.exec`, `new Function()`).
    - Features logic to distinguish patterns found in executable code from those within comments.
    - Parses `package.json` files to identify project dependencies (note: this script primarily focuses on code patterns, not dependency vulnerabilities).
    - Generates a console report listing detected patterns, their file locations (using shortened paths relative to the project root), and a code snippet for context.
    - Summary: `security_eval-graph.md` (includes the use of mermaid.js)



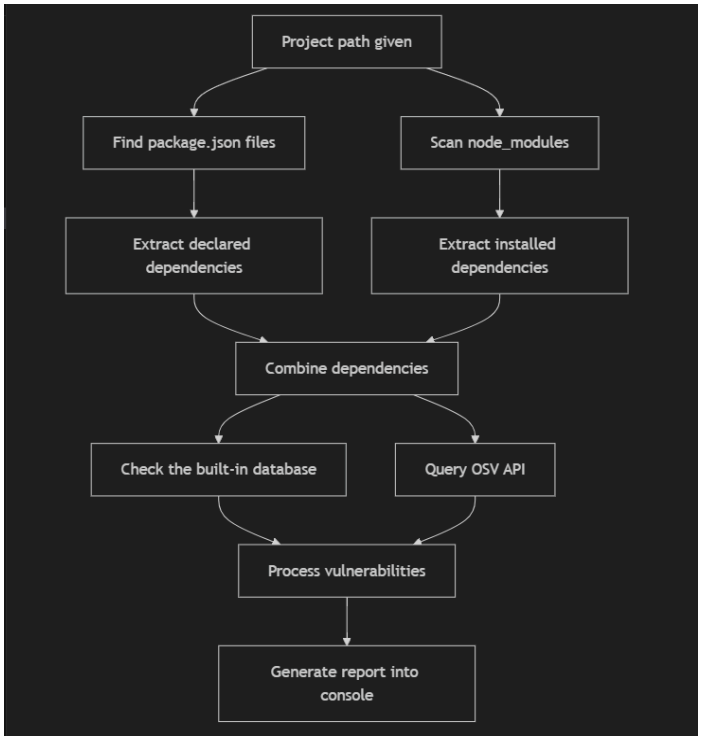
4.2. Dependency vulnerability analyzer: `parse-modules.py`

- **Objective**: To identify known vulnerabilities within the third-party dependencies of a project.
- **Functionality**:
    - Discovers `package.json` files throughout the project (excluding `node_modules` for initial discovery of *declared* dependencies).
    - Extracts a comprehensive list of dependencies by considering both declared dependencies (from `package.json` files) and actually installed dependencies (by

inspecting `node_modules` subdirectories and their `package.json` files to get precise versions).

- For each identified dependency and its version:
  - Checks against a small, embedded database of known vulnerabilities for common packages (e.g., `lodash`, `express`).
  - Performs a query against the OSV.dev (Open Source Vulnerability) database using its public API to fetch information on publicly disclosed vulnerabilities.
- Normalizes vulnerability data obtained from OSV.dev, including determining severity levels (critical, high, medium, low) based on CVSS v3 scores or vectors.
- Generates a console report listing all discovered packages and then detailing any found vulnerabilities. Vulnerabilities are grouped by severity and include the CVE or OSV ID, a description, and a link to OSV.dev for further details.

- Summary : `parse-modules-graph.md` (includes the use of mermaid.js)



4.3. Usage & Further Details

These Python-based static analysis tools are intended for command-line execution, providing developers with a way to perform local security checks.

For more detailed information on the static analysis tools, their development process, specific findings from their use, and a comprehensive evaluation of their effectiveness and limitations:

4.3.1 Dangerous code pattern analyzer: `security-eval.py`

For this tool, we took the approach of object-oriented design, some configurable scanning options (via CLI arguments). This tool provides the most typical static analysis techniques such as lexical analysis (regex-based patterns) and semantic filterings (comments exclusion as well as context awareness)

- Findings and fixes

    - In order to reduce a large quantity of false positives (specially when searching for patterns during the dependencies analysis) we had to make a comment-aware analysis together with some abstract syntax pattern detection.
    - At first, the code just found the different patterns but didn't provide the localization of it, so we added a small context code snippet as well as the file in which the dangerous pattern was found in order for the developer to implement an easy fix, or at least be aware of the problem location.
- Problems with future work proposals for the code:
    - We are just using predefined regex pattern matching
        - We could probably include AST parsing
        - We could add even more known patterns (maybe even include a database of known vulnerable patterns)
    - We can't detect obfuscated code
        - A solution could be to try and find easy obfuscation techniques in the code to flag them and maybe even try deobfuscating them
    - At the moment, if the pattern is divided into multiple lines, a false negative can arise
        - Try to analyze the code in a different way (eliminating unnecessary
 to correctly process the code)
- In conclusion , this tool provides valuable pattern detection but only at a surface-level, as it lacks deeper semantic analysis of other SAST tools.

4.3.2 Dependency vulnerability analyzer: `parse-modules.py`

The development process started as creating a CLI tool using arguments to create a customizable tool. As explained earlier, it implements a layered vulnerability detection process (Built-in-DB + cloud DB). The static analysis provided is the recursive dependencies parsing and later normalization to check with the DDBB.

- Findings and fixes
    - To categorize the risks between critical, high, medium and low, we added the vector parsing of data such as CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
    - In order to reduce a large quantity of false positives we added a version cleaning code to prevent false matches in the versions such as 1.2.3-beta == 1.2.3
- Problems with future work proposals for the code:
    - The local database quickly becomes outdated,

        - Use the OSV API together with other known and popular databases with module vulnerabilities
    - There is not transitive dependency checking
        - We should implement recursive dependency resolution using the package-lock.json files to build a dependency tree
    - Limited version range support (only npm)
        - We could extend the tool to support multiple package ecosystems integrating some universal version parsers such as PyPI, Maven or NuGet
    - No fix version recommendations (although it can be seen from the OSV database link, is not directly specified)
        - Make an API / webscrapping for finding a version without vulnerabilities
    - It is very memory-intensive for large `node_modules` (and slow)
        - implementing concurrent programming solutions for faster and more efficient searches
    - Sometimes, although npm audit finds some vulnerabilities, as they are not included in OSV DB as CVE's or other type of vulnerabilities, they are not shown
        - Use a combination of OSV DB with some other popular and reliable DDBB.
- In conclusion, the tool effectively combines local and cloud-based vulnerability databases but lacks some features such as dependency graph analysis or fix suggestions.

5. Technical Notes (Web Application Components)

The development of the web-based CSP Generator and DeepSeek Vulnerability Detector involved several technical considerations. Key among these were managing complex UI state (for file uploads, scan progress, results display, and interactive elements) within the React/Next.js framework, and orchestrating asynchronous operations effectively. This included implementing features like batch file processing, API retry mechanisms with exponential backoff, and scan queuing. For the CSP Generator, integrating the external Python script necessitated careful use of Node.js's `child_process` module and developing a method to convert terminal ANSI color codes into HTML for accurate display in the browser.

6. Future Enhancements

Across the project components, several enhancements are envisioned:

- **Static Analyzer Tools**:
    - **Web UI Integration**: Consider integrating `security-eval.py` and `parse-modules.py` into the web application, allowing users to trigger these local scans and view reports via the UI. This would likely require API endpoints to manage the execution of these scripts.
    - **The ones proposed in 4.3.1&2**
- **Better LLM:** Use a better (paid) DeepSeek or other models (from OpenAI, xAI, Google, etc.)

7. Limitations

The Vulnerability Detection component, particularly its planned integration with the DeepSeek API, has several important limitations to consider:

- **AI Model Knowledge and Training:** Like all AI models, DeepSeek's capabilities will be tied to its training data. This includes:
    - **Knowledge Cutoff:** It may not detect vulnerabilities that have emerged or been disclosed after its last training cycle.
    - **Training Data Bias:** The range and types of vulnerabilities it can identify will be influenced by the data it was trained on. Underrepresented vulnerability classes or coding patterns might be missed.
- **Accuracy (False Positives/Negatives):** Automated scanners, including AI-driven ones, can produce:
    - **False Positives:** Reporting vulnerabilities that do not actually exist.
    - **False Negatives:** Failing to identify genuine vulnerabilities. All findings from such tools require careful manual verification by security experts.
- **Contextual Understanding:** AI models may struggle to fully grasp the business logic or specific operational context of an application. We also check file by file, so if there is an error in any combination it will not be detected. This can make it difficult to determine if a suspicious code pattern is genuinely exploitable or if a complex, context-dependent vulnerability exists.
- **Analysis of Transpiled/Minified Code:** The effectiveness of static analysis can be significantly reduced when performed on transpiled, minified, or obfuscated code, which is common in production JavaScript builds. Analysis is most effective on original source code.

- **Dependency Vulnerabilities:** The extent to which an integrated DeepSeek API would analyze vulnerabilities within third-party dependencies (as opposed to the user's direct code) is unknown.
- **Not a Substitute for Comprehensive Security Practices:** An AI-powered scanner is a valuable tool for augmenting security efforts but cannot replace them. Thorough manual code reviews, architectural security analysis, and professional penetration testing remain essential for comprehensive security assurance.

8. Conclusion

This project successfully delivers three distinct security-focused components: a web-interfaced CSP Generator, a web-based (currently mock) DeepSeek Vulnerability Detector, and a set of standalone Python static analyzer tools (if used together they cover 80% of the top 10 OWASP risks for react apps, missing authentication flaws and server side missconfigurations). The web application provides a user-friendly platform for the first two, while the Python scripts offer robust local analysis capabilities. Together, these tools provide a multi-faceted approach to improving web application security, with clear avenues for future development, especially the full activation of the DeepSeek API for the vulnerability detection web tool.

10. Github repository

https://github.com/cseas002/find_react_vulnerabilities