Detailed Report on NVD Assessment

January 27, 2025

Contents

1 Problem Statement **2**

2 Solution Approach **2**

3 Backend Implementation **2**
- 3.1 Technologies Used . . . 2
- 3.2 Key Features . . . 3
- 3.3 Code Snippet . . . 3

4 Frontend Implementation **4**
- 4.1 Technologies Used . . . 4
- 4.2 Key Features . . . 4
- 4.3 Code Snippet . . . 4

5 Challenges Faced **5**

6 Testing **5**

7 GitHub Repository **5**

8 Conclusion **5**

1 Problem Statement

The goal of this assessment was to consume CVE (Common Vulnerabilities and Exposures) information from the NVD (National Vulnerability Database) API and store it in a database. The data should be cleansed, deduplicated, and periodically synchronized in a batch mode. Additionally, APIs were required to filter CVE details based on parameters like CVE ID, year, score, and last modified date. Finally, the data had to be visualized in a user interface (UI).

2 Solution Approach

The solution was divided into two main parts:

1. Backend Implementation using Flask
2. Frontend Implementation using React

The following steps were followed:

1. Consuming data from the NVD API using chunked responses controlled by parameters like `startIndex` and `resultsPerPage`.
2. Storing the retrieved data in a TinyDB database after data cleansing and deduplication.
3. Implementing periodic synchronization of data.
4. Developing RESTful APIs to fetch and filter data based on specific parameters.
5. Creating a user-friendly interface to visualize data with features like pagination, sorting, and detailed views.

3 Backend Implementation

3.1 Technologies Used

- Flask: To create a lightweight and efficient backend API.
- TinyDB: A lightweight database for storing CVE information.
- Flask-CORS: To handle cross-origin requests from the frontend.
- threading: To enable periodic database updates in a separate thread.

3.2 Key Features

1. **Data Retrieval and Cleansing:** Data was fetched from the NVD API, and dates and metrics were parsed and stored in a structured format.

2. **Data Cleansing and Deduplication:** A dedicated function was implemented and tested to ensure high-quality data by removing duplicates and standardizing formats.

3. **Periodic Synchronization:** A separate thread ensures the database is updated daily with the latest CVE data. The backend checks for updates every 1 minute and performs batch-wise updates if changes are detected.

4. **Filtering, Pagination, and Sorting:** APIs support filtering by year, CVE ID, score, and modification date. Server-side pagination and sorting, particularly for dates, were implemented for optimal performance.

3.3 Code Snippet

Below is a snippet of the periodic update function:

Listing 1: Database Update Function

```
def update_database():
    while True:
        try:
            total_results = 0
            start_index = 0

            while True:
                data = fetch_cves_from_api(start_index)
                if not data:
                    break

                vulnerabilities = data.get("vulnerabilities", [])
                if not vulnerabilities:
                    break

                for item in vulnerabilities:
                    cve_data = item.get("cve", {})
                    cve_id = cve_data.get("id")
                    ...
                    db.upsert(doc, query.cve_id == cve_id)

                start_index += len(vulnerabilities)
                total_results += len(vulnerabilities)

                if total_results >= data.get("totalResults", 0):
                    break

        except Exception as e:
```

```
            print(f"Error in update process: {e}")

        time.sleep(60)  # Check for updates every 1 minute
```

Note: Due to the large amount of data, it takes approximately 4-5 minutes for preprocessing and data cleansing during the initial database initialization.

4 Frontend Implementation

4.1 Technologies Used

- React: To build the user interface.
- React Router: To handle routing between pages.
- CSS: For styling the components.

4.2 Key Features

1. **List View:** Displays CVE data in a tabular format with pagination and filtering options.
2. **Detail View:** Displays detailed information about a specific CVE when a row is clicked.
3. **Dynamic Updates:** Fetches data dynamically based on user interactions like pagination or filter changes.

4.3 Code Snippet

Below is a snippet of the list component:

Listing 2: React CVE List Component

```
const fetchCVEs = async () => {
    try {
        const response = await fetch(
            `http://localhost:5000/api/cves?page=${currentPage}&per_page=${
        );
        const data = await response.json();
        setCves(data.results);
        setTotalRecords(data.total_records);
    } catch (error) {
        console.error('Error fetching CVEs:', error);
    }
};
```

5 Challenges Faced

- Parsing dates from the API, as they followed inconsistent formats.
- Handling large volumes of data and ensuring smooth pagination.
- Implementing periodic synchronization without impacting API performance.
- Debugging the `resultsPerPage` functionality to ensure accurate and efficient data retrieval.

6 Testing

Unit tests were written for the following functionalities:

- API endpoints for filtering, pagination, and sorting.
- Frontend components for list and detail views.
- Data cleansing and deduplication functions.

All tests were successfully passed, ensuring the robustness of the application.

7 GitHub Repository

The project code is available on GitHub. Click the link below to access it:
GitHub Link: https://github.com/sujansanjeev/securin-assessment-1

8 Conclusion

The assessment was successfully completed, with the backend and frontend components working seamlessly together. The application effectively retrieves, stores, filters, and visualizes CVE data, meeting all the stated requirements. Key enhancements include server-side sorting for dates, robust data cleansing and deduplication, and automated batch updates.

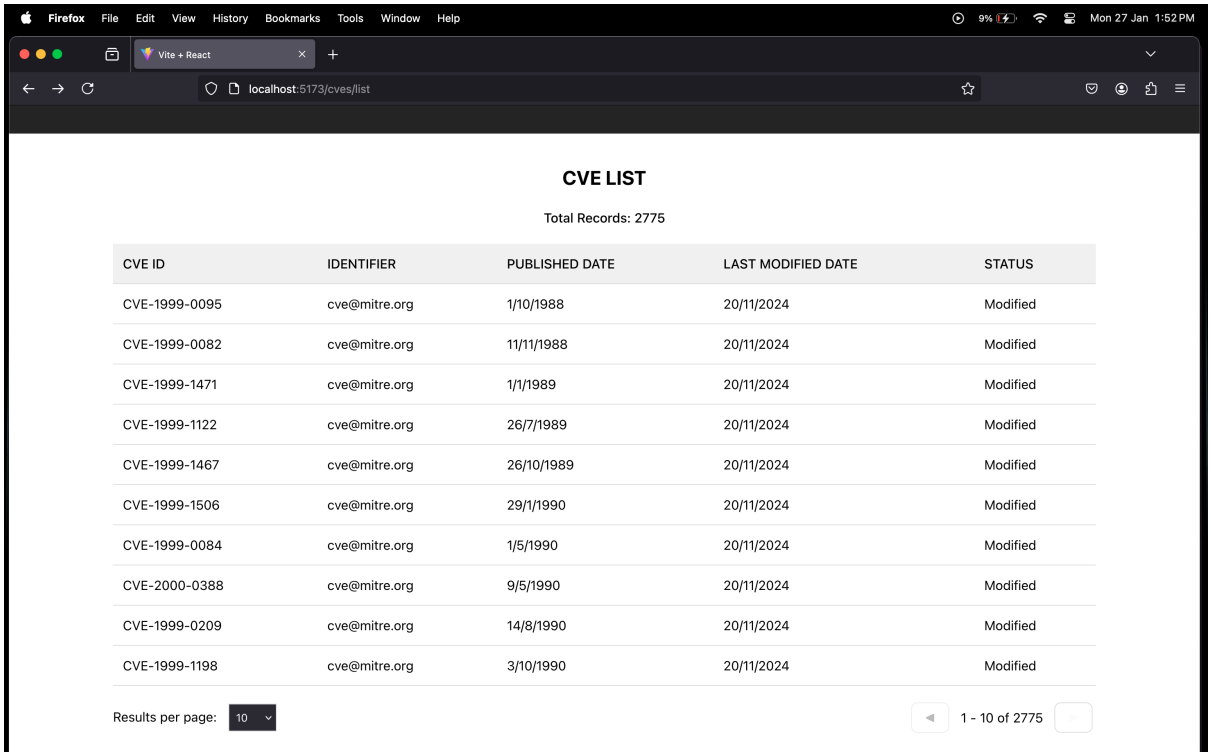

Figure 1: List Output

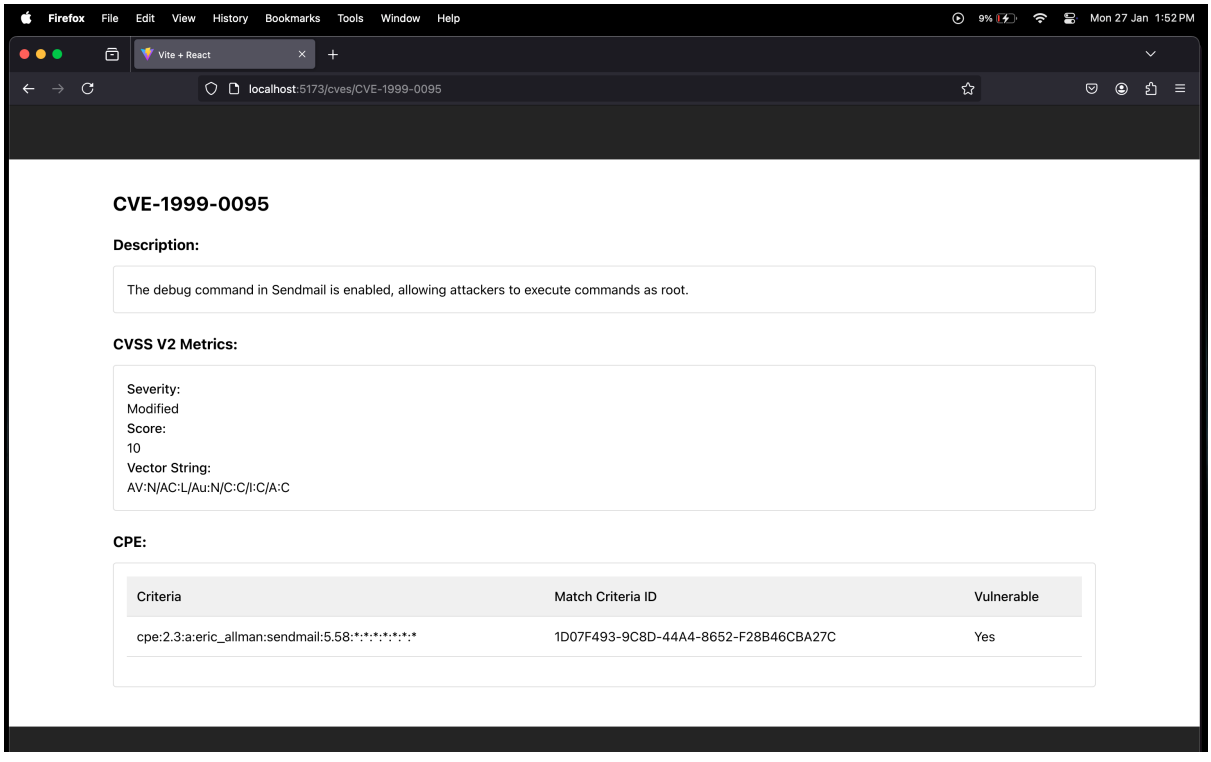

Figure 2: Details Output