Security Vulnerability Dashboard — Implementation and Optimization Documentation

Overview:

The Security Vulnerability Dashboard is a React-based web application designed to efficiently visualize, filter, and analyze a large JSON dataset (~300MB) containing security vulnerability information for various packages. The dashboard supports advanced data visualizations, real-time search and filtering, sorting, exporting, and comparison features. Due to the size and nested complexity of the dataset, special attention was given to performance, responsiveness, and user experience.

1. Data Loading Strategy:

Initially, we attempted to fetch the large JSON data file directly using the browser's Fetch API and parse it in the main React thread. This approach led to noticeable performance lags, blocking UI rendering, and significantly delaying user interactivity. The JSON also contained deeply nested structures, which made client-side flattening expensive.

To address this, we decided to offload the data loading and flattening logic to a Web Worker. The Web Worker was responsible for:

- Fetching the data from the given URL
- Parsing and traversing the nested `groups → repos → images → vulnerabilities` structure.
- Flattening the vulnerability data into a consumable array format.
- Applying filters inside the worker to reduce memory usage on the main thread.

This worker-based offloading allowed the main UI thread to remain responsive and enabled large-scale data handling without crashing or freezing the browser.

2. Data Filtering Mechanism:

The first version of filtering was implemented directly in React using `useMemo`. While this worked for smaller datasets, performance degraded quickly with real-time inputs such as search queries or dropdown changes.

To resolve this, we moved the filtering logic entirely to the Web Worker. We structured the worker to accept a filter object containing:

- kaiStatus values to exclude
- Severity level to filter
- Search term (CVE ID or package name)

The worker processed the data and returned a filtered array to the React app. This ensured only relevant results were passed back to the UI, drastically improving performance and reducing re-renders.

We also implemented debounced search using `use-debounce` to reduce the frequency of worker invocations during rapid user input.

3. UI Component Structure and Lazy Loading:

To improve the application's perceived performance and avoid unnecessary component loading, we used `React.lazy` and `Suspense` for all major chart components. This delayed the rendering of charts until after the main layout was visible, allowing the dashboard to feel responsive even before all data visualizations were ready.

We also introduced a timed delay of 300 milliseconds before rendering charts to smoothen transitions during initial load.

The dashboard structure was composed as follows:

- Layout wrapper with top-level title and consistent styling.
- Filter bar allowing filtering by kaiStatus, severity, and search term.
- Metrics section showing total vulnerabilities and export buttons side by side.
- Two rows of chart visualizations laid out in two-column format.
- Paginated, sortable vulnerability table with virtual scrolling.
- Vulnerability comparison view for selected items.

4. Pagination and Virtualization:

With over 2 million rows possible in the dataset, rendering even a subset of the data at once would have been highly inefficient.

We first implemented traditional pagination using Material UI's `TablePagination`, which helped control how many rows were rendered per page.

To go further, we introduced `react-window` for virtualized row rendering. This meant that even within a page, only the visible rows in the viewport were rendered to the DOM. This significantly improved rendering speed and memory efficiency, especially when rows-per-page was set to higher values.

We retained Material UI styling by reconstructing the table rows using `Box` and `Typography` components instead of using MUI's `<Table>` components, which are incompatible with virtualization libraries.

5. Row Selection and Comparison View:

Initially, selection was tracked using a Set of CVE IDs. However, many rows shared the same CVE but differed by version, package name, or fix date. This caused the comparison view to display duplicates and unrelated rows.

To fix this, we transitioned to tracking full selected row objects. Each row was uniquely identified by a combination of CVE, package version, and fix date. Selection logic was updated to compare these fields when toggling checkbox selection, and only exact matches were stored.

This update made the comparison view consistent with the user's actual selection and prevented any confusion caused by shared CVEs.

6. Sorting Implementation:

We implemented column-level sorting on `cve`, `severity`, `cvss`, and `fixDate`. Sorting state was managed through `sortBy` and `sortOrder` hooks.

Sorted data was computed using `useMemo` to ensure it was only recalculated when dependencies changed. Sorting was applied before pagination, ensuring that each page displayed the correct segment of sorted data.

To visually communicate sortability, each header cell displayed an indicator:

- An upward arrow for ascending order
- A downward arrow for descending order
- A dimmed ⇅ symbol for unsorted state

Clicking on a column header toggled through these sort states.

7. Export and User Interaction Features:

We implemented both JSON and CSV export functionality using native browser Blob and file-saving techniques. This avoided third-party dependencies like `json2csv`, which is not browser-compatible.

The ExportButtons component allowed users to download currently filtered results, ensuring that only visible and relevant data was exported.

In addition, all filters, sorting, and search states were preserved during interactions, giving users complete control over their dataset view.

8. Visual Layout and Responsiveness:

We optimized the layout using Material UI's `Stack`, `Box`, and `Typography` components. All four charts were laid out in two rows, each containing two side-by-side charts. This grid format ensured visual balance and efficient use of screen real estate.

The layout was also responsive; on smaller screens, charts stacked vertically thanks to `flexWrap` and `minWidth` configurations.

9. Final UI Flow and Data Pipeline Summary:

- The app initializes and launches a Web Worker to fetch and parse the large JSON file.
- The main UI shows a loader while the worker flattens and filters the data.
- Filter options and search input are passed to the worker, which returns a filtered array.
- The React app receives this array and passes it to all visual components.
- Export buttons allow saving the current filtered view.
- Charts are lazy-loaded after a slight delay to improve perceived speed.
- The paginated table with virtualization handles millions of rows with minimal DOM load.
- Users can select specific vulnerabilities to view in a dynamic comparison table.
- Sorting logic updates the display based on user column interactions.

Conclusion:

The Security Vulnerability Dashboard was built with performance, scalability, and usability in mind. It evolved from a naive synchronous approach to a highly optimized, modular, and responsive application. Key performance wins came from offloading processing to Web Workers, applying virtualization for large datasets, and lazy-loading visualization components.

This architecture is now ready for production deployment and can be further extended to support real-time data feeds, persistent user preferences, and advanced analytics.