Container Image Vulnerability Scanner: Product Requirements and Wireframes

Product Requirements Document

1. Introduction

1.1 Purpose

This document outlines the requirements for a security product designed to scan container images for vulnerabilities and present findings to users, enabling them to identify and prioritize images requiring fixes based on vulnerability severity. The product addresses the needs of security engineers managing large repositories.

1.2 Scope

The product provides functionality to scan container images, classify vulnerabilities into severity levels (critical, high, medium, low), and offer a user-friendly web-based interface to view, filter, and manage scan results efficiently, even for repositories with thousands of images. It focuses on scanning and result presentation, with integration into broader workflows (e.g., CI/CD pipelines) considered optional.

2. Functional Requirements

2.1 Image Scanning

- **Support for Formats**: The system shall support scanning of container images in common formats, such as Docker and OCI.
- **Vulnerability Detection**: The scanning process shall identify known vulnerabilities in image components and dependencies.
- **Database Integration**: The system shall integrate with up-to-date vulnerability databases (e.g., National Vulnerability Database) to ensure accurate and current vulnerability data.

2.2 Vulnerability Classification

- **Severity Levels**: Vulnerabilities shall be classified into four severity levels: critical, high, medium, and low.
- **Detailed Information**: For each vulnerability, the system shall provide:
    - CVE ID
    - Description
    - Affected component
    - Recommended remediation steps (if available)

2.3 User Interface

The interface shall be web-based, accessible via standard browsers, and designed for usability and efficiency.

2.3.1 Dashboard

- **Summary Statistics**: Display key metrics, including:
  - Total images scanned
  - Number of images with critical vulnerabilities
  - Number of images with high, medium, and low vulnerabilities
- **Visualizations**: Include charts (e.g., bar or pie) to show vulnerability distribution by severity.
- **Recent Activity**: Show recent scan activities or alerts for new critical/high vulnerabilities.

2.3.2 Image List View

- **Image Listing**: List all scanned images with columns for:
  - Image Name
  - Critical Vulnerability Count
  - High Vulnerability Count
  - Medium Vulnerability Count
  - Low Vulnerability Count
  - Last Scanned Date
- **Sorting**: Enable sorting by any column (e.g., by critical count or image name).
- **Filtering**: Provide filters to show images based on vulnerability severity (e.g., only those with critical vulnerabilities).
- **Search**: Include a search bar to find images by name or other attributes.
- **Pagination**: Implement pagination or infinite scrolling to handle thousands of images efficiently.

2.3.3 Image Detail View

- **Metadata**: Display image metadata, including name, tag, size, and scan date.
- **Vulnerability List**: Show all vulnerabilities, grouped by severity (e.g., tabs or sections for Critical, High, etc.).
- **Vulnerability Details**: For each vulnerability, provide:
  - CVE ID
  - Severity
  - Description
  - Affected Component
  - Remediation Steps
- **Navigation**: Allow users to return to the list view or dashboard easily.

3. Non-Functional Requirements

3.1 Performance

- **Scalability**: The system shall handle scanning and displaying results for thousands of images without significant delays.
- **UI Responsiveness**: The user interface shall load quickly, even with large datasets, using techniques like lazy loading or virtualization.

3.2 Security

- **Data Protection**: Scan results and image data shall be stored and accessed securely, with appropriate encryption and access controls.
- **Authentication**: If multiple users are supported, implement role-based access control (optional).

3.3 Usability

- **Intuitive Design**: The UI shall be intuitive, with clear visualizations (e.g., color-coded severity indicators) and straightforward navigation.
- **Accessibility**: The interface shall comply with basic accessibility standards for web applications.

3.4 Reliability

- **Scanning Accuracy**: The scanning process shall minimize false positives and negatives, ensuring reliable vulnerability detection.

4. User Stories

The following user stories reflect the needs of a security engineer managing container images:

1. **Scanning**: I want to scan my container images for vulnerabilities to identify potential security risks.
2. **Overview**: I want to view a summary of my images' security posture to assess overall risk quickly.
3. **Prioritization**: I want to filter and sort images based on vulnerability severity to prioritize fixing efforts.
4. **Details**: I want to view detailed vulnerability information for an image to understand necessary fixes.

5. Constraints

- **Image Formats**: The system must support common container image formats (e.g., Docker, OCI).
- **Platform**: The UI must be web-based and compatible with standard browsers (e.g., Chrome, Firefox).
- **Scale**: The system must efficiently handle repositories with thousands of images.

6. Assumptions

- **Image Access**: Users have access to container images in a repository (e.g., Docker Hub, private registries).
- **Scanning Trigger**: Scanning can be initiated manually or automatically (e.g., on image push), but the mechanism is outside this PRD’s scope.
- **User Role**: The primary user is a security engineer or similar role with technical knowledge of containerized environments.

7. Dependencies

- **Vulnerability Databases**: Integration with external databases like the National Vulnerability Database for CVE data.
- **Container Registries**: Potential integration with registries (e.g., Docker Hub) for automated scanning (optional).
- **Web Framework**: A modern web framework (e.g., React) for the UI, ensuring scalability and responsiveness.

8. Acceptance Criteria

- **Scanning**: The system successfully scans a sample set of images and identifies known vulnerabilities.
- **UI Functionality**:
    - Dashboard displays accurate summary statistics and charts.
    - Image list view supports sorting, filtering, and searching with no performance issues for 10,000 images.
    - Detail view shows complete vulnerability information with remediation steps.
- **Performance**: Page load times remain under 2 seconds for typical operations.
- **Security**: Data is encrypted in transit and at rest, with no unauthorized access during testing.

Low-Fidelity Wireframes Description

The following wireframes outline the user interface for the Container Image Vulnerability Scanner. They are low-fidelity, focusing on layout and functionality rather than visual design.

Dashboard

- **Header**: Contains the product logo and navigation menu with links to Home, Images, and Settings.
- **Summary Cards**: Four cards displaying:
    - Total Images Scanned: [Number]
    - Images with Critical Vulnerabilities: [Number]
    - Images with High Vulnerabilities: [Number]
    - Images with Medium/Low Vulnerabilities: [Number]
- **Chart**: A bar chart showing the number of images per vulnerability severity (Critical, High, Medium, Low).
- **Recent Activity**: A list of recent scan activities or alerts (e.g., “Image X scanned: 2 critical vulnerabilities found”).
- **Layout**:

```
[Logo] [Home | Images | Settings]
---------------------------------
| Total Images | Critical | High | Med/Low |
---------------------------------
| [Bar Chart: Vulnerability Distribution] |
---------------------------------
| Recent Activity:                |
| - Scan completed for Image X... |
| - Scan completed for Image Y... |
---------------------------------
```

Image List View

- **Search Bar**: Located at the top left for searching images by name.
- **Filter Dropdowns**: Dropdowns for filtering by severity (e.g., “Show only Critical”) and other attributes.
- **Table**: Columns include Image Name, Critical, High, Medium, Low, Last Scanned. Each row is clickable to access the detail view.
- **Pagination Controls**: Buttons or infinite scrolling at the bottom for navigating large datasets.
- **Layout**:

```
[Logo] [Home | Images | Settings]
---------------------------------
```

```
| Search: [________] | Filter: [Severity: All] |
---------------------------------
| Image Name | Crit | High | Med | Low | Last Scanned |
|------------|------|------|-----|-----|--------------|
| Image1     | 2    | 5    | 10  | 20  | 2025-05-01   |
| Image2     | 0    | 3    | 8   | 15  | 2025-05-02   |
---------------------------------
| [Prev] [1] [2] [3] [Next] |
---------------------------------
```

Image Detail View

- **Image Metadata**: Displays Name, Tag, Size, and Last Scanned at the top.
- **Vulnerability Sections**: Organized by severity (e.g., Critical, High) using tabs or collapsible sections.
- **Vulnerability Table**: Columns for CVE ID, Description, Affected Component, and Remediation. Rows list individual vulnerabilities.
- **Navigation**: A “Back to List” button to return to the image list view.
- **Layout**:

```
[Logo] [Home | Images | Settings]
---------------------------------
| Image: Image1 | Tag: latest | Size: 200MB | Scanned: 2025-05-01 |
---------------------------------
| [Critical] [High] [Medium] [Low] |
---------------------------------
| Critical Vulnerabilities:         |
| CVE ID | Description | Component | Remediation |
|--------|-------------|-----------|-------------|
| CVE-123| Buffer...   | LibX      | Update to...|
---------------------------------
| [Back to List] |
---------------------------------
```

Bonus: Development Action Items

The following action items are proposed for discussion with the development team to implement the Container Image Vulnerability Scanner:

1. **Scanning Engine Development**:
    - Select or develop a scanning engine (e.g., based on open-source tools like Clair or Trivy) to analyze container images for vulnerabilities.
    - Ensure support for Docker and OCI formats.
2. **Vulnerability Database Integration**:
    - Integrate with external vulnerability databases, such as the National Vulnerability Database, to fetch real-time CVE data.
    - Implement caching to optimize database queries.
3. **Backend Services**:
    - Develop RESTful APIs to store and retrieve scan results, using a database (e.g., PostgreSQL) for scalability.
    - Implement efficient querying for large datasets (thousands of images).
4. **User Interface Development**:
    - Build a web-based UI using a framework like React, ensuring responsiveness and scalability.
    - Implement the dashboard, image list view, and detail view as described in the wireframes.

    - Use libraries like Chart.js for visualizations and DataTables for sortable/filterable tables.
5. **Performance Optimization**:
    - Optimize backend queries and UI rendering to handle large image repositories.
    - Implement lazy loading or virtualization for the image list view to reduce load times.
6. **Security Implementation**:
    - Secure APIs with authentication (e.g., OAuth) and encrypt data in transit (TLS) and at rest.
    - If supporting multiple users, implement role-based access control.
7. **Testing and Validation**:
    - Conduct unit and integration tests for the scanning engine and UI components.
    - Validate scanning accuracy against known vulnerable images.
    - Perform load testing to ensure performance with 10,000+ images.
8. **Documentation and Training**:
    - Create developer documentation for the scanning engine and APIs.
    - Provide user guides for the UI, focusing on filtering and prioritization features.

Implementation Considerations

- **Existing Tools**: Tools like Snyk, Wiz, and Anchore provide inspiration for UI design and scanning capabilities. For example, Snyk's dashboard emphasizes prioritization based on exploitability, which could inform our severity-based filtering.
- **Scalability**: Given the requirement to handle thousands of images, the backend should use indexing and caching strategies to ensure quick data retrieval.
- **Usability**: Color-coding (e.g., red for critical, orange for high) and clear remediation steps will enhance user experience, as seen in tools like Docker Scout.
- **Extensibility**: While not required, the system could be designed to support future features like report exports (PDF/CSV) or integration with ticketing systems (e.g., Jira).

Risks and Mitigation

- **Risk**: Inaccurate scanning due to outdated vulnerability data.
    - **Mitigation**: Regularly update the vulnerability database and validate against known CVEs.
- **Risk**: UI performance issues with large datasets.
    - **Mitigation**: Use pagination, lazy loading, and optimized queries.
- **Risk**: Security vulnerabilities in the product itself.
    - **Mitigation**: Conduct security audits and implement best practices for data protection.

Conclusion

The Container Image Vulnerability Scanner addresses the critical need to identify and prioritize vulnerabilities in large container image repositories. By combining robust scanning with an intuitive, scalable UI, the product empowers security engineers to manage risks effectively. The proposed wireframes and development action items provide a clear path for implementation, drawing on industry best practices and existing tools for inspiration.