Testing Overview

The SIEM system's testing procedure ensures that the frontend and backend operate as intended, abiding by secure procedures while preserving dependability and performance. The tests carried out on various system components are listed below.

Frontend Testing

The SIEM system's frontend, built in React, is tested using Jest and the React Testing Library.

Unit Tests

- Each component has its unit tests built to ensure it renders correctly and can handle button clicks, input changes, and other user interactions.
- Functions like managing events, emulating user actions, and rendering forms were all covered by test cases. For example, the `Alert.test.js` code confirms that the alert component renders accurate data and initiates actions such as opening investigations.

Integration Tests

- In most cases, API call interactions and authentication were simulated using hardcoded data, with some tests using actual API calls to verify frontend and backend interactions. For instance, tests for secure access or form submissions ensured that API endpoints were correctly hit and responses were handled appropriately. This allowed for thorough testing of both component behavior and backend integration.

Manual Testing

- Additional UI behavior validation was done by manual testing, particularly for components that deal with dynamic data and interactions that aren't fully covered by unit tests. Some tests may fail due to faulty coding, but having visual confirmation is helpful when making decisions. Examples include examining page navigation, dashboard statistics display, and real-time alert updates.

Running Tests

- Docker is used to run tests, and the `frontend-tests` service specified in the Docker Compose file is used to run the tests. The testing procedure is automated via a shell script (`scripts/run_frontend_tests.sh`), which makes it simple to run tests consistently in various contexts.

Backend Testing

The backend, developed using Django, is tested with Django's built-in test framework.

Unit Tests

- For models, views, and serializers in programs like Accounts, Alerts, and Logs, unit tests cover essential functionality. Tests, such as user creation (`test_user_creation`) and alert management (`test_alert_creation`), verify that objects can be created, retrieved, and edited accurately.
- Tests also confirm speed and logging, ensuring the system can effectively manage large datasets and intricate queries.

Integration Tests

- Backend integration tests verify that the APIs function as anticipated, appropriately processing requests and replies, and integrating with other services, such as Celery and Redis, for background operations.

Manual Testing

- In addition to the automated tests, manual tests were carried out. For instance, event data was subjected to detection criteria to produce alerts, and logs from several sources were ingested, processed, and stored safely.

Running Tests

- Backend tests are conducted automatically using the `scripts/run_backend_unit_tests.sh` file, which uses Pytest and Django's test framework. This script ensures consistency across environments, locally or in Docker, by configuring the test database, executing tests, and reporting results.

Security Testing

A key component of the SIEM system is security. Testing was done to ensure that the system complies with safe development principles and mitigates vulnerabilities like those listed in the OWASP Top 10. These security tests ensured the system followed best practices, providing a strong foundation for protecting user data and ensuring system integrity.

- **Data Encryption**: Tests on data encryption confirmed that private information, including alert details and user passwords, is securely protected both during transmission and at rest. For instance, the system was checked to ensure that all communications take place over HTTPS and that passwords are hashed in the database.
- **Input Validation and Injection Protection**: Testing was done to ensure that all user inputs are sanitized to prevent injection attacks. The

system uses parameterized queries to safely manage user input and stop malicious code execution, as demonstrated via SQL injection testing.

- **Vulnerability and Dependency Scanning**: Using programs like `npm audit`, the system was examined for outdated or vulnerable libraries. This ensured that both frontend and backend dependencies were safe from known vulnerabilities and kept up to date.
- **Role-Based Access Control (RBAC)**: Tests verified that user roles, such as Administrator and Analyst, were appropriately implemented, limiting access to sensitive information and features according to user permissions.