# OSINT Web App - Project Analysis

## 1. What tests should be implemented in the programming of such a project? What tests would you include in your project if given more time?

* Recommended Tests for This Project

1. Unit Tests

* - Backend (Kotlin): Unit tests should cover core services, domain logic, and utility classes. This includes tests for domain validation, subprocess or Docker command execution logic, and response parsing.
* - Frontend (React): Component-level tests using tools like Jest and React Testing Library, especially for user input handling, component rendering, and state transitions.

1. Integration Tests

* - Backend integration tests should verify communication between services and proper handling of subprocesses or Docker-based scan execution.

1. End-to-End (E2E) Tests

* - Use tools like Cypress or Playwright to simulate user behavior: starting a scan, receiving and displaying results, and handling errors gracefully.

1. API Tests

* - Use frameworks like Ktor, Spring Test, or REST-assured to test REST controllers for input validation, response correctness, and error handling.

1. Security and Input Validation Tests

* - Validate command injection prevention, strict input formatting, and check for frontend vulnerabilities like XSS and CSRF.

Additional Tests If More Time Were Available

* - Full unit test coverage and mock subprocesses.
* - Mocked frontend API tests with simulated backend responses.
* - CI/CD automation for test execution and coverage reporting.
* - Accessibility tests for screen reader and keyboard navigation support.

## 2. How can you measure the performance of your project? Can you optimize your code to run faster?

Performance in the osint-web-app project includes how fast scans are initiated, how efficiently data flows between backend and frontend, and how responsive the UI remains.

Methods to Measure Performance

* Backend (Kotlin)
* - Use Micrometer or Spring Boot Actuator to track REST endpoint response times.
* - Profile performance using VisualVM, YourKit, or IntelliJ's built-in tools.
* - Benchmark subprocess or Docker execution duration.
* Frontend (React)
* - Run Lighthouse audits for UI performance metrics.
* - Use the React Profiler to analyze component render times.

Code Optimization Opportunities

* Backend:
* - Avoid redundant Docker container launches and use async execution.
* Frontend:
* - Use React lazy loading, memoization, and efficient state management to reduce re-renders and improve responsiveness.

## 3. What bottlenecks can be caused by the OSINT tools given to this project?

Using OSINT tools like amass introduces performance bottlenecks due to their heavy processing and external network reliance.

Common Bottlenecks

* - Execution Time: Tools like amass are resource-intensive and long-running.
* - Blocking Operations: Synchronous execution can tie up backend threads.
* - Docker Overhead: Container startup and teardown can add latency.
* - Output Parsing: Large scan results increase CPU and memory use.
* - Network Dependency: External API calls and lookups introduce unpredictability.
* - Scalability: Most OSINT tools are not designed for parallel, high-concurrency environments.

Mitigation Strategies

* - Run scans asynchronously with coroutines.
* - Implement scan queues and limit concurrency.
* - Use caching and preloaded containers.
* - Limit scan scope and depth for performance.

Some Important notes generally:

Dear reviewer, while reviewing the task, please consider that this is my first experience with React and front-end development at all. I know that the results that I have from that side are not competitive but I felt during the implementation of the task that learning and becoming efficient in that field will not be problem for me in future. So, I have the willingness to learn and develop my skills if I receive the offer.