**Security Testing Report**

**Project:** AI Stock Sentiment Analysis System  
**Author:** Arpit Parekh  
**Date:** May 1, 2025  
**Frontend:** React (JavaScript, MUI)  
**Backend:** Flask (Python), REST API  
**Authentication:** JWT-based  
**Deployment:** Local development (localhost:3000, localhost:5000)  
**Purpose:** To evaluate and report the security posture of the full-stack application using black-box and white-box testing methods.

**1. Authentication and Session Management**

**Test Case 1.1: Token Tampering**

**Description:** Modify the payload of the JWT to impersonate a different user.  
**Steps:**

1. Decode the issued JWT token using an online tool.
2. Alter the sub or email claim and re-encode the token.
3. Attach the tampered token to an authenticated API call.

**Expected Result:** Server should reject the token as invalid.  
**Result:** Passed – the backend detects the signature mismatch and returns 401 Unauthorized.

**Test Case 1.2: Token Expiry Handling**

**Description:** Verify if expired tokens are correctly invalidated.  
**Steps:**

1. Generate a short-lived token.
2. Wait for the expiry time.
3. Attempt to access a protected route.

**Expected Result:** Token should be rejected after expiration.  
**Result:** Passed – token is rejected with an appropriate expiration message.

**Test Case 1.3: Missing Token Access**

**Description:** Access protected endpoints without providing a token.  
**Steps:**

1. Remove the token from request headers.
2. Make a GET request to /api/stock/:symbol.

**Expected Result:** Server returns a 401 Unauthorized error.  
**Result:** Passed – no access granted without authentication.

**Test Case 1.4: Local Storage Token Exposure**

**Description:** Simulate script injection to steal tokens from localStorage.  
**Steps:**

1. Inject script into input field or browser console.
2. Attempt to read localStorage.

**Expected Result:** Token is accessible via client-side JavaScript.  
**Result:** Confirmed Risk – token is accessible, suggesting a move to HttpOnly cookies is safer.

**2. API Endpoint Security (Flask)**

**Test Case 2.1: Input Validation (SQL Injection)**

**Description:** Inject SQL payloads into login and query fields.  
**Steps:**

1. Use payloads like ' OR 1=1 -- in login and search fields.
2. Monitor backend logs or observe unauthorized access.

**Expected Result:** Server handles inputs safely; no injection occurs.  
**Result:** Passed – SQLAlchemy ORM correctly escapes inputs.

**Test Case 2.2: CSRF Protection**

**Description:** Simulate a cross-site form submission to protected endpoints.  
**Steps:**

1. Create an HTML form on a different origin.
2. Submit POST request to /api/refresh.

**Expected Result:** Server rejects request due to CSRF.  
**Result:** Failed – no CSRF tokens in place.

**Test Case 2.3: Rate Limiting**

**Description:** Rapidly send repeated requests to test denial-of-service resistance.  
**Steps:**

1. Use a loop or script to hit /api/login repeatedly.

**Expected Result:** Server should rate limit or throttle responses.  
**Result:** Failed – no rate limiting implemented.

**Test Case 2.4: Unauthorized Recommendation Access**

**Description:** Attempt to access stock recommendation data for unauthorized users.  
**Steps:**

1. Use a valid JWT, but for a different user.
2. Request /api/recommendations/:symbol.

**Expected Result:** User-scoped data should be enforced.  
**Result:** Passed – endpoint doesn’t leak data across users.

**3. Frontend React Security**

**Test Case 3.1: Cross-Site Scripting (XSS)**

**Description:** Inject malicious scripts via input fields.  
**Steps:**

1. Enter <script>alert('XSS')</script> in the Live Analysis search.
2. Observe whether the script executes.

**Expected Result:** Script should be rendered as plain text.  
**Result:** Failed – inner HTML not sanitized, resulting in alert execution.

**Test Case 3.2: Error Logging in Console**

**Description:** Inspect browser console for sensitive information.  
**Steps:**

1. Trigger errors in form inputs or failed API calls.
2. Monitor developer tools console.

**Expected Result:** Only generic error messages should appear.  
**Result:** Passed – no sensitive user or token data is logged.

**Test Case 3.3: React Developer Tools Exposure**

**Description:** Check if user credentials or token are exposed in dev tools.  
**Steps:**

1. Inspect context providers via React DevTools.
2. Review application state.

**Expected Result:** Sensitive tokens should not be exposed in components.  
**Result:** Passed – state is minimal; token only in localStorage.

**4. Web Scraping Module Security**

**Test Case 4.1: Arbitrary URL Fetching**

**Description:** Test for Server-Side Request Forgery (SSRF) vulnerabilities.  
**Steps:**

1. Modify backend scraper URL to access internal/non-financial domains.
2. Observe the backend response and logs.

**Expected Result:** Server should reject unknown domains.  
**Result:** Failed – scraper accepts arbitrary URLs.

**Test Case 4.2: Timeout Handling**

**Description:** Simulate delays or unresponsive sources during scraping.  
**Steps:**

1. Point scraper to a site with delayed response.
2. Observe application behavior.

**Expected Result:** Server should timeout and return fallback response.  
**Result:** Passed – scraper handles timeouts gracefully.

**5. Database & ORM Security (SQLAlchemy)**

**Test Case 5.1: ORM Query Sanitization**

**Description:** Test for injection in raw SQL or unsafe query building.  
**Steps:**

1. Manually craft SQL-like inputs and pass them via search endpoints.

**Expected Result:** ORM escapes all parameters safely.  
**Result:** Passed – no SQL injection observed.

**Test Case 5.2: Log and Console Output Scrutiny**

**Description:** Check if sensitive data is being logged.  
**Steps:**

1. Enable debug logs.
2. Register/login and check log outputs.

**Expected Result:** Logs must not contain passwords or tokens.  
**Result:** Passed – no exposure of sensitive data in logs.

**Summary Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Status** | **Priority** | **Recommendation** |
| Token Tampering | Passed | High | Maintain current JWT validation |
| Token Expiry | Passed | High | Add refresh token rotation |
| XSS in Input Fields | Failed | High | Sanitize DOM using libraries like DOMPurify |
| CSRF Protection | Failed | High | Implement Flask-WTF or CSRF tokens |
| Rate Limiting | Failed | Medium | Use Flask-Limiter to control abuse |
| SSRF Protection in Scraper | Failed | High | Whitelist domains, block internal IPs |
| SQL Injection (ORM) | Passed | High | ORM is safe, continue using parameterization |
| LocalStorage Token Access | Confirmed | Medium | Move to HttpOnly secure cookies |

**Actionable Recommendations**

1. **Sanitize All User Inputs:** Use DOMPurify in the frontend before rendering any content to prevent XSS.
2. **CSRF Tokens:** Integrate CSRF protection for state-changing endpoints using Flask-WTF or equivalent.
3. **Rate Limiting:** Add Flask-Limiter to protect endpoints from abuse.
4. **SSRF Defense:** Restrict scraping endpoints to known financial data providers (e.g., Yahoo Finance).
5. **JWT Management:** Shift to HttpOnly cookies and implement a refresh token system.
6. **Security Headers:** Apply secure headers in Flask using middleware like Flask-Talisman (e.g., CSP, X-Frame-Options).
7. **Use HTTPS:** Enforce HTTPS in production to prevent MITM attacks.