Executive Summary

I performed a focused manual security assessment of the open-source chat application **Strand Chat** (https://github.com/louisllw/strand-chat) at commit `aa4555d8cf15ed0494938ad1c9a20243df74c0de`, testing authorization boundaries and session lifecycle behavior in a local Docker environment. The assessment confirmed that private conversations were not accessible across unauthorized accounts, but identified a session revocation weakness: a previously issued authentication token remained usable at the refresh endpoint after a normal logout. This weakens logout as a containment control if a token is exposed and may allow an attacker to continue renewing access or disrupt a victim's active session.

Scope and Environment

Target application: Strand Chat

Repository: GitHub open-source project

Commit assessed: `aa4555d8cf15ed0494938ad1c9a20243df74c0de`

Deployment context: Localhost over HTTP in Docker

Architecture observed: React/TypeScript frontend, Express/Node.js backend, Socket.IO for realtime features, Redis caching, and Postgres.

Assessment Objectives

The assessment focused on two areas:

1. **Authorization enforcement**
    - Can one authenticated user access another user's private conversation data?
2. **Session revocation and token lifecycle**
    - Does normal logout invalidate previously issued authentication material?
    - Can an old token still be used to obtain fresh authentication after logout?
    - Does the compromise-handling flow properly invalidate prior sessions?

Methodology

Testing was performed manually using authenticated test accounts and direct interaction with the application's HTTP endpoints. I examined conversation access behavior and the authentication flows associated with logout, refresh, and compromise handling. The primary endpoints evaluated were:

- `GET /api/conversations`
- `POST /api/auth/refresh`
- `POST /api/auth/compromised`

Findings

Finding 1: Authorization Checks on Private Conversations Worked as Expected

I created a private conversation between User A and User C, captured the associated conversation identifier, then attempted to access that conversation while authenticated as User B.

The request was denied, indicating that conversation access control was enforced for this test case. This was a positive result and not a vulnerability.

Finding 2: Logout Did Not Fully Revoke Previously Issued Authentication Material

I authenticated as User A, copied the active token, logged out through the normal client workflow, and then submitted the old token to `POST /api/auth/refresh`. Although logout had occurred, the old token was still accepted and a fresh authentication cookie was issued. Repeating the refresh request with the previously issued token continued to generate new authenticated sessions.

This behavior suggests that standard logout clears client-side state but does not fully invalidate the server-side credential used by the refresh flow. As a result, possession of an old token may still be enough to regain authenticated access after logout.

Finding 3: Compromise Handling Successfully Invalidated the Old Token

To test the boundary of the issue, I invoked `POST /api/auth/compromised` using User A's token and then attempted to refresh the session again using the copied older token. This time the request was denied with an unauthorized response. That indicates the application does have a mechanism to invalidate prior sessions, but that protection is not applied during ordinary logout.

Security Impact

This issue does not appear to allow account takeover by itself; an attacker would first need possession of a valid token. However, once such a token is exposed, normal logout does not

reliably terminate the attacker's ability to refresh the session. In practical terms, this reduces the value of logout as a containment mechanism after token theft.

Potential impacts include:

- Continued session renewal after the legitimate user logs out
- Reduced effectiveness of logout during incident response
- Possible disruption of the victim's session if the attacker repeatedly leverages the compromise-related flow or competes for session control

Severity Assessment

My assessment concludes this is a **moderate session management weakness**.

Why moderate:

- It requires prior token exposure
- It does not appear to bypass authentication from scratch
- But it does undermine a core security expectation: logging out should revoke continued use of that session's refresh capability

Recommended Remediation

The application should invalidate the relevant server-side refresh credential when a user performs a normal logout. Possible fixes include:

- Revoke the current refresh token on logout

- Store refresh tokens server-side and validate against revocation state
- Use rotating refresh tokens with token-family invalidation on suspicious reuse
- Ensure logout and compromise flows apply consistent revocation behavior
- Reduce refresh token lifetime where appropriate