ChatGPT

Dacument Security Analysis

Dacument is a JavaScript CRDT library where **every change is cryptographically signed** and checked against a role-based ACL [1]. In practice, each actor (client) must register an identity and key pair via `Dacument.setActorInfo()`, and then perform operations by signing them with a role-specific private key (owner/manager/editor) and its own "actor" key. On merge, Dacument verifies that each operation's signature is valid and that the signer's current ACL role permits that change. Unknown fields or invalid types are strictly rejected. In other words, **integrity and write-access are enforced by signatures and ACL rules** [1] [2].

However, Dacument does *not* provide any confidentiality: it assumes the transport and storage are secured separately. The documentation explicitly notes that there is *"no built-in encryption: use TLS or E2E encryption and treat snapshots as sensitive data."* [3]. In a scenario with authenticated, end-to-end-encrypted peer connections, the network-level threats (e.g. MITM or packet snooping) are largely eliminated. The remaining threats come from **malicious or compromised peers, key leakage, and resource abuse**.

Threat Model: With an authenticated, encrypted channel, attackers must be one of the peers (or have infiltrated one). The primary threats are: compromising a role's private key (owner/manager/editor) or an actor's private key; abusing a valid identity/role to insert bad ops; or refusing to cooperate (DoS by flooding or withholding updates). Dacument is *"secretless"* in that it doesn't manage secrets or encryption itself, so it relies entirely on external confidentiality (e.g. TLS/E2EE). Assuming transport security, we focus on malicious insiders or compromised nodes. For example, if an attacker steals the *owner's role key*, they can issue arbitrary operations as "owner." Because Dacument does **not support key rotation**, the only mitigation is to fork to a new document via `accessReset()` when compromise is suspected [4].

Key threats include:

- **Key Compromise:** Private keys (actor or role) are sensitive. Stolen keys allow forging signed operations. The docs warn "no rotation exists, so migrate by snapshotting into a new Dacument with fresh keys" if compromise is suspected [4]. In practice, the application must store keys securely (e.g. OS keychain, hardware module, or encrypted storage) and invoke `accessReset()` if keys leak.
- **Shared/Insider Attacks:** Multiple actors may share a role key (e.g. all "owner" actions use the same key). This means actions are *attributed to a role*, not an individual. A malicious insider with a valid role key can inject any change that role permits. The documentation notes this by design: treat roles as trust groups and log merges if audit trails are needed [4]. To reduce this risk, one should limit role key distribution and use `verifyActorIntegrity()` if per-user attribution is required.
- **Replay or Sync Attacks:** CRDT ops include a hybrid logical clock stamp, so reordering or replay of old ops does not violate consistency (ops are idempotent/ordered by stamp). Dacument tracks applied tokens to avoid duplicate application. However, *withholding ops* (e.g. a replica never sends its changes) means others will not see them until a resync. The docs suggest using reliable transport and occasional snapshot resync to handle delays [5].
- **Denial-of-Service (Flooding):** A malicious peer could spam many small operations to hog CPU or storage. Dacument's recommendation is to implement rate-limiting or payload-size limits at the

application layer [6]. (For example, don't accept thousands of tiny edits per second from one client, and monitor merge errors or unusually large diffs.)
- **Revocation Handling:** Once an actor's role becomes "revoked" (via an ACL op), Dacument prevents them from reading anything beyond initial values and rejects any new writes. In code, a revoked replica's reads return only the schema defaults [7]. This helps protect the document from further corruption by a revoked client, but be aware that the revoked client's local copy still has the full state (revocation only affects its own view).

Potential Weaknesses: Dacument's security hinges on its signing and ACL logic, which appears robust, but a few design limitations should be noted:

- **No Encryption or Confidentiality:** By design, all CRDT state and snapshots are in plaintext (JSON). If an attacker gains access to a snapshot or storage, they see the entire document history and contents. The library explicitly says to treat snapshots as sensitive [3]. Users should therefore encrypt logs or snapshots at rest, and ensure all network transport is secured (TLS/E2EE).
- **Key Management Limits:** Role keys are generated once at `create()` and **never automatically rotated**. There is no internal key-lifecycle management beyond manual reset. If a role key is long-lived, a leak is catastrophic until you manually migrate. The ACL mechanism *does* allow an owner to grant or revoke roles dynamically (subject to permission rules), but it does not generate new keys. Hence **best practice is to call** `accessReset()` **promptly on any suspicion of key leakage** [4].
- **Shared Role Keys:** Because Dacument does not differentiate *users* beyond actors and roles, all members of a role share the same cryptographic authority. This means you cannot truly audit "user A vs. user B" within the same role — only "someone with owner privileges." The library warns that attribution is role-level [4]. To mitigate, applications may require each user to have a unique role (e.g. only one person holds the owner key) or perform external logging of actor IDs.
- **Denial via Missing Acks:** The CRDT garbage-collection (tombstone compaction) waits for all non-revoked actors to acknowledge operations. If an offline or malicious replica never acks, tombstones accumulate indefinitely. This is an availability/performance issue: a malicious node could refuse to ack to force others to keep growing state. A practical mitigation is to monitor stale actors and use snapshots/resync to trim logs [5].
- **Actor Signature Optional:** Dacument allows an optional *detached* actor signature on each op for auditing. However, **Merges do not enforce actorSig by default** (they only verify role-key signatures). Thus by default you can't tell which actor in a role generated an op unless you manually call `verifyActorIntegrity()`. If your threat model requires per-user accountability, you must explicitly use that verification path.

Usage Environment: Dacument runs **client-side** (in the browser or Node≥18) and requires a modern WebCrypto implementation [8]. It is ESM-only and does not work in old browsers or Node versions. Because it is secretless, it does *not* depend on any server-stored secret – all cryptography is done locally. In practice, you would use Dacument in an application that already establishes secure, authenticated peer connections (e.g. via WebRTC with mutual authentication, or a TLS-protected WebSocket to a known server). Under this assumption, Dacument's signed operations guarantee integrity and ordering, while confidentiality is provided by the outer channel.

Typical use cases include **collaborative or offline-first apps** where multiple clients independently mutate shared state. In such settings, Dacument ensures eventual consistency even with concurrent edits. It is **not** a security boundary: for maximum safety, always assume network encryption (TLS/E2EE) and validated peer identities. Also, since snapshots do not include the schema or private keys, all replicas must use the *exact same schema definition* on load; tampering with the schema outside Dacument can cause rejection of data.

Best Practices for Maximum Security:

- **Secure Channels:** Always run Dacument over authenticated, encrypted channels. E.g., use HTTPS or WebRTC DTLS, and validate peer identities before exchanging ops. The docs emphasize using TLS/E2EE since Dacument has *"No built-in encryption"* [3].
- **Protect Private Keys:** Store each actor's private key and any role private keys in a secure manner (browser KeyStore, hardware token, encrypted local storage, etc.). Never hard-code or share private keys. Call `Dacument.setActorInfo()` at startup with each client's own key pair, so that Dacument can sign operations securely [9].
- **Principle of Least Privilege:** Assign roles sparingly. Only share the owner/manager keys with trusted users. Grant "editor" or "viewer" roles (which have no signing keys) to others when possible. That way, even if an editor's environment is compromised, the attacker cannot unilaterally change ACL or critical fields. Remember that *only owners can assign owners/managers* [10] [11], and managers cannot promote to owner [10], so design your delegation accordingly.
- **Audit and Verification:** If you need per-user accountability, use the actor signature. After syncing, call `verifyActorIntegrity()` on the op log or incoming ops to ensure each op's detached actorSig matches the published public key in ACL [12] [13]. This is an off-line check (not done automatically on merge) and can flag if someone is forging actor IDs.
- **Regular Resets:** In a long-lived system, periodically "snapshot and reset" the document. This creates a fresh document ID and key material via `accessReset()`, embedding a signed reset op. It breaks causal chains and revokes all old actors (who are marked revoked afterward) [4]. Use this if you suspect any key has leaked, or simply on a schedule for good hygiene.
- **Rate Limiting and Monitoring:** Implement application-level throttling. For example, if your UI sends one typed character as an op, put a debounce so it doesn't send 100 ops per second. Check merge events and subscribe to `error` events to detect signature failures or unexpected rejections [14]. Cap the size of any single patch (e.g. limit how many array entries or text characters can be inserted in one op). Watch for any user continuously sending ops without acknowledging others.
- **Encrypt Snapshots at Rest:** Since snapshots contain the entire history, store them encrypted (e.g. in IndexedDB with a CryptoKey, or on the server encrypted per-user). The README warns that snapshots should be treated as sensitive [3]. If your app takes periodic backups or server storage of state, encrypt that backup.
- **Validate Schemas:** Always load a snapshot with the expected schema. Dacument will throw an error if a field is missing or has the wrong type [15] [16]. Do not allow untrusted clients to load with an arbitrary schema, as that could bypass ACL checks. In practice, keep schema definitions in your application code, not user-supplied.

In summary, **Dacument's design strongly secures integrity and write-access via signatures and ACL checks**, but depends on external measures for confidentiality and key security. By running it in a secure client environment with E2E-encrypted channels and carefully managing keys/roles, you can achieve a robust collaborative state system. Key compromises and insider abuse remain the top risks – mitigate them by least-privilege role assignment, regular key resets, and auditing as recommended in the documentation [1] [17].

Sources: The Dacument README and code detail its cryptographic design and threat guidance [1] [17]. These emphasize schema-enforced operations with ES256 signatures and role-based ACLs, plus explicit warnings about key compromise, insider attacks, and the need for external encryption (TLS/E2E) for transit and snapshot confidentiality [17] [1].

---

1 3 4 5 6 8 14 17 README.md

https://github.com/jortsupetterson/dacument/blob/fa3d1e7c15404ee3b12dff1f9c8bc902dea4e44f/README.md

2 7 9 10 11 12 13 15 16 class.ts

https://github.com/jortsupetterson/dacument/blob/fa3d1e7c15404ee3b12dff1f9c8bc902dea4e44f/src/Dacument/class.ts