Security Audit Report

Keylay

Protocol Implementation Review: Cryptographic Design, Transport Security, and Relay Trust Claims

Subject:	**Keylay Encrypted QR Relay — index.html, server.js**
Security page reviewed:	**security.html (keylay_site_updated)**
Audit scope:	**Cryptographic correctness, protocol claims, relay trust model**
Methodology:	**Manual source review, claim-vs-code differential analysis**
Date:	**April 01, 2026**
Status:	**Internal draft — not a formal third-party audit**

This document presents a systematic review of the security properties claimed in Keylay's published security.html page against the behavior of the deployed source code (index.html and server.js). Each claim was traced to the relevant code path and evaluated independently. Discrepancies, vulnerabilities, and confirmed properties are reported separately.

Table of Contents

1. Executive Summary

This review examined the deployed source code of the Keylay encrypted QR relay application against the security guarantees published in its security.html page. The audit covered the full cryptographic stack — code generation, channel derivation, PBKDF2 key stretching, X25519 ephemeral key exchange, HMAC-authenticated handshake, HKDF session key derivation, and AES-256-GCM message encryption — as well as the relay server's role management, session isolation, and logging behavior.

The central finding is that the deployed code is measurably **more secure than the security page currently documents**. Two properties described as absent — counter-bound replay protection and 300,000-iteration PBKDF2 — are in fact present in the code. The security page therefore understates the current implementation's posture. This needs to be corrected, because a security page that lags behind the implementation erodes the credibility of the documentation rather than protecting it.

The most significant concrete concerns are in the relay server, which performs no rate limiting, validates no message sizes, and logs full message objects including channel hashes. On the client, a counter-advancement-before-decryption ordering creates a viable denial-of-service vector, and the code generator has a small but measurable modular bias. None of these are critical vulnerabilities given the stated threat model, but each is worth addressing.

2. Scope and Methodology

Files Reviewed

File	Role	Lines
index.html	Client application — all cryptographic logic, UI, WebSocket client	~12,700
server.js	WebSocket relay server — session routing, role management	146
security.html	Published security claims page — basis for claim comparison	410

Methodology

Review proceeded in four phases. First, the security page was read in full to extract every specific, testable claim about the protocol. Second, each claim was mapped to the corresponding function or variable in the source code and evaluated for accuracy. Third, the code was reviewed independently for vulnerabilities not addressed in the security page. Fourth, server-side logic was analyzed separately with attention to unauthenticated state transitions, information leakage, and denial-of-service surface.

All cryptographic assessments used the Web Crypto API as the reference implementation. No formal symbolic verification or differential fuzzing was performed. This review should be understood as a thorough manual analysis, not as a formal audit.

3. Protocol Architecture

Keylay routes coordination data (PSBTs, public keys, UR sequences) between two peers through a WebSocket relay. The relay is deliberately kept blind: it routes ciphertext and sees only a derived channel identifier, never the raw session code or plaintext payload. The security model depends on this relay-blindness being cryptographically enforced, not merely trusted.

Protocol Steps (as implemented)

1	**Code entry**	User enters or generates a session code. The generator uses a 31-character unambiguous alphabet (no 0/O/1/I/l) at 10 characters length, providing a theoretical maximum of log■(31¹■) ≈ 49.8 bits of entropy. Manual entry is accepted up to 32 characters.
2	**Channel derivation**	SHA-256(code) is computed; the first 32 hex characters (128 bits) are used as the WebSocket channel name. The relay sees this hash, not the raw code.
3	**HMAC key derivation**	The raw code is stretched with PBKDF2-SHA256 using the salt ‘keylay-v1-hmac’ and 300,000 iterations to produce an HMAC-SHA256 signing key. This key authenticates hello messages without revealing the code.
4	**Ephemeral X25519 key generation**	An ephemeral X25519 key pair is generated per session via crypto.subtle.generateKey. The private key is never exported; it is set to null after the handshake completes.
5	**Signed public-key exchange**	Each peer HMAC-signs its base64-encoded ephemeral public key and sends {type:'hello', pubkey, sig} to the relay. The relay forwards it. The receiving peer verifies the HMAC using its locally derived key. An adversary without the session code cannot forge a valid signature.
6	**HKDF session key derivation**	Both peers compute X25519(myPrivate, theirPublic) to get shared bits. These bits are imported as HKDF key material. The HKDF info field encodes 'keylay-v1-session|' + sorted(myPubB64, theirPubB64), binding the derived AES-256-GCM key to the exact handshake participants. No session key transits the wire.
7	**AES-256-GCM message encryption**	Each message is encrypted with a fresh 12-byte random IV and additional authenticated data 'keylay-v1|' + counter, where counter is a monotonically increasing integer. The receiver enforces strict monotonic counter ordering, rejecting any message whose counter is not strictly greater than the last accepted.
8	**Handshake state machine**	States: IDLE → HELLO_SENT → ACTIVE. A concurrency lock prevents simultaneous hello processing. If ACTIVE and a different peer public key arrives, the state is fully reset and a new handshake is initiated.

4. Verified Security Claims

Each security property described in security.html was traced to a specific code path and evaluated for correctness. The table below records the finding against each claim.

Status	Claim (from security.html)	Finding
PASS	Relay sees only ciphertext and channel hash, not plaintext	Confirmed. Payload is AES-GCM ciphertext; channel name is SHA-256(code)[:32].
PASS	Relay MITM resistance without code knowledge	Confirmed. HMAC over ephemeral public key; forgery requires PBKDF2 derivation from code.
PASS	Forward secrecy: past sessions not decryptable after code disclosure	Confirmed. Session key derived from ephemeral X25519; private key set to null post-handshake.
PASS	No session key transmitted	Confirmed. Both peers derive AES key locally from shared DH bits via HKDF.
PASS	HKDF info binds key to sorted public key pair	Confirmed. info = "keylay-v1-session|" + [myPub,theirPub].sort().join("|").
PASS	AES-GCM with 12-byte random IV per message	Confirmed. crypto.getRandomValues(new Uint8Array(12)) used per encrypt() call.
PASS	Unencrypted data messages rejected	Confirmed. msg.encrypted === false causes immediate rejection with console.warn.
PASS	Peer-key change triggers full handshake reset	Confirmed. New ephemeral key pair generated; counters reset; sessionKey nulled.
PASS	Session limited to exactly two peers	Confirmed. server.js refuses third connection and closes socket.
PASS	Ephemeral private key discarded after handshake	Confirmed. ephemeralKeyPair = null at line 12274 immediately after key derivation.
PASS	HMAC verification uses Web Crypto constant-time verify	Confirmed. crypto.subtle.verify() used — not a string comparison.
PASS	No custody of wallet private keys	Confirmed. App handles only coordination data (PSBTs, xpubs). No signing occurs.
NOTE	Code knowledge treated as session membership	Correct as stated. Any party with the code can join and participate — by design.

5. Discrepancies: Security Page vs. Deployed Code

The security.html page explicitly positions itself as a description of "the code as currently deployed." Three of its specific claims are factually inconsistent with the deployed source code. All three discrepancies favor the implementation — the code is more secure than the page claims — but the page is nonetheless inaccurate, which undermines its stated purpose.

DISCREPANCY	PBKDF2 Iteration Count
Claimed:	Security page states: current deployment uses 100,000 iterations; target V1 design specifies 300,000 iterations.
Actual:	Code (deriveHmacKey) sets iterations: 300,000. The target iteration count has already been deployed. The security page describes an older intermediate state.

DISCREPANCY	Replay Protection Status
Claimed:	Security page states: "Replay protection is not yet deployed" and "recorded ciphertext can be retransmitted." It explicitly lists this as a known absence.
Actual:	The deployed code includes counter-bound AAD in every AES-GCM encryption call ("keylay-v1|" + counter) and enforces strict monotonic counter ordering on receipt (recvCounter check). Replay protection is present in the current implementation.

DISCREPANCY	Hello Message Wire Format
Claimed:	Security page states: "hello carried inside a data envelope with JSON-stringified payload."
Actual:	The code sends hello messages as first-class WebSocket messages of type "hello" with direct pubkey and sig fields — not wrapped in a data envelope. The server handles them in a dedicated handleHello() function. The described format appears to have been a prior iteration.

Implication: The security page should be updated to reflect the current implementation. Describing the system as less capable than it is does not constitute conservative security practice — it introduces inaccuracy into a document whose credibility depends on precision.

6. Identified Vulnerabilities and Weaknesses

The following findings are ordered by severity within their category. Severity ratings reflect exploitability within the stated threat model, not in an absolute sense.

MODERATE F-01 · Counter Advancement Before Decryption Verification

Location: processDataMessage() in index.html.

When a data message arrives, the receiver immediately advances recvCounter to msg.counter before attempting decryption. If decryption subsequently fails — due to a corrupted payload, wrong key, or an injected malformed message — the counter has already been consumed. Any legitimate message carrying the same counter value will be rejected as a replay.

Exploitability: An active relay can inject a syntactically valid WebSocket message with a forged counter value (e.g., counter = 9999) before the legitimate message with that counter arrives. Decryption will fail (the payload is not authentic ciphertext under the session key), but recvCounter is now 9999. All subsequent legitimate messages with counter ≤ 9999 are silently dropped. This constitutes a denial-of-service against the encrypted channel without requiring knowledge of the session key.

Note: The relay currently forwards only messages whose sender is a known session peer, which limits the injection window to a peer who has successfully completed the HMAC handshake. An external attacker who does not know the session code cannot easily inject. However, a compromised relay can fabricate messages freely.

Recommendation: Advance recvCounter only after successful decryption and GCM authentication tag verification.

MODERATE F-02 · Server: No Rate Limiting or Message Size Enforcement

Location: server.js — handleDataTransmission(), handleHello(), handleJoin().

The relay server imposes no restrictions on connection rate, message frequency, or message payload size. Any client that can reach the WebSocket endpoint can open connections, join sessions, and transmit arbitrarily large or frequent messages.

Practical impact: A single attacker can saturate the relay's bandwidth or memory by flooding the channel with large payloads, degrading or denying service to legitimate sessions. The messageBuffer array on the client side similarly has no size cap, creating a secondary memory exhaustion vector if a peer floods messages before the handshake completes.

Recommendation: Enforce per-IP connection limits, per-session message rate limits, and a maximum message payload size (e.g., 64 KB) at the server. Cap messageBuffer on the client.

LOW

F-03 · Server: Full Message Logging Including Channel Hash

Location: server.js — console.log() calls throughout.

The server logs the deserialized message object on every incoming message: console.log('Received message:', message). This includes the code field (the channel hash), message type, and any other top-level fields. For hello messages, this means the ephemeral public key and HMAC signature are written to server logs. Payload is encrypted base64, so plaintext is not exposed, but the channel hash — which identifies the session — appears in logs and allows session traffic correlation.

Additionally, routing decisions are logged with the channel hash: "Routed encrypted data to N peer(s) for code: ". If server logs are retained, all session identifiers are preserved indefinitely.

Recommendation: Replace full-object logging with structured metadata-only logging (message type, peer count, payload size in bytes). Remove channel hash from log output in production.

LOW

F-04 · Unauthenticated Role Claiming (Server Design Issue)

Location: server.js — handleClaimSender().

Any connected peer can send {type: "claim", code: } at any time to claim the sender role, which demotes the current sender to receiver. The server performs no additional authentication beyond confirming the client is already connected to that session.

Within the stated threat model — where code knowledge equals membership — this is by design. However, it is not documented in security.html, and the behavior means an attacker who has obtained the session code (e.g., by intercepting the out-of-band code exchange) can disrupt the session by repeatedly claiming and releasing the sender role.

Recommendation: Document this behavior explicitly in the security page. Consider whether the sender role should be permanently bound to the first joiner for the duration of a session.

LOW

F-05 · Missing Content Security Policy

Location: index.html — section.

The page does not set a Content-Security-Policy header or meta tag. In the absence of a CSP, a successful XSS injection would have unrestricted access to the DOM, WebSocket connection, session key variables, and all decrypted payloads.

The security page acknowledges that endpoint compromise is out of scope. However, a CSP is a standard first-line control against browser-based injection and does not require accepting endpoint compromise as inevitable.

Recommendation: Add a restrictive CSP. At minimum: default-src 'self'; script-src 'self'; connect-src wss: ws:. If inline scripts must remain, use nonces or hashes.

INFO

F-06 · No Server-Side Session Expiry

Location: server.js — sessions Map.

Sessions in the server's Map are removed only when all peers disconnect. There is no maximum session lifetime or idle expiry. A session whose peers connect once and then become unreachable (e.g., tab closed without clean disconnect) persists in memory indefinitely.

Recommendation: Add a maximum session age (e.g., 24 hours) and an idle timeout (e.g., 30 minutes with no forwarded messages). Clean up stale sessions on a periodic interval.

7. Entropy and Code-Strength Analysis

Generated Code Parameters

The code generator uses CHANNEL_CODE_CHARS = `'abcdefghjkmnpqrstuvwxyz23456789'` (31 characters, omitting ambiguous glyphs 0/O/1/I/l) at a length of CHANNEL_CODE_LENGTH = 10. Maximum theoretical entropy: log■(31¹■) ≈ 49.84 bits.

Modular Bias in Code Generation

Code generation samples from crypto.getRandomValues(new Uint8Array(10)), producing values in [0, 255]. Each value is reduced modulo 31 (CHANNEL_CODE_CHARS.length). Because 256 = 8 × 31 + 8, the first 8 characters of the alphabet (a, b, c, d, e, f, g, h) each have selection probability 9/256 ≈ 3.52%, while the remaining 23 characters have probability 8/256 = 3.125%.

This constitutes a modular bias. The effective entropy per character, computed as the Shannon entropy of the resulting non-uniform distribution, is approximately 4.973 bits rather than the ideal log■(31) ≈ 4.954 bits. At 10 characters, the total effective entropy is approximately **49.73 bits** versus the theoretical 49.84 — a reduction of 0.11 bits. This is negligible for practical security but is technically an implementation defect relative to a uniform sampler.

The unbiased fix is to use rejection sampling: discard bytes ≥ (256 - 256 % 31) = 248 and redraw. Alternatively, sample from [0, 30] directly using a cryptographically seeded range.

Attack Cost Analysis

At 49.84 bits of entropy, an online brute-force attack against a 10-character generated code requires on the order of 2■■ ≈ 562 trillion guesses in the worst case. HMAC verification is the authentication gate, and each verification requires a PBKDF2 operation at 300,000 iterations (approximately 300–500 ms per guess on modern hardware without dedicated ASICs). This makes online guessing computationally expensive even against weak codes.

For offline attack against recorded traffic: because the session key is derived from ephemeral X25519 — not directly from the code — learning the code after the session ends does not decrypt past ciphertext. An attacker must have been present at session time to perform an active MITM, which requires forging an HMAC signature, which in turn requires the code. The forward secrecy guarantee holds as implemented.

For manually entered codes: the security page correctly notes this is parameterized. A 6-character code from the same alphabet yields ≈36 bits; a 4-character code yields ≈24 bits, which is trivially brute-forceable offline against a recorded handshake.

8. Server-Side Security Review

Session Isolation

Sessions are keyed by channel hash (SHA-256(code)[:32]) and stored in a server-side Map. Each session holds exactly one sender and one receiver reference. The server refuses third connections for a given session hash and closes the socket. This two-peer limit is correctly enforced.

Cross-session isolation relies entirely on hash collision resistance. With a 128-bit channel identifier (first 32 hex chars of SHA-256), accidental collision between two unrelated sessions has negligible probability under normal usage volumes.

Role Assignment and Transfer

The first client to join a session hash is assigned the sender role. Subsequent role transfers via the "claim" message type are accepted without further verification. While the symmetric encryption means both roles can send and receive ciphertext, the sender/receiver role distinction controls the UI and which peer initiates QR display versus camera scan. A session adversary who knows the code could disrupt workflow by role-cycling, though they cannot access plaintext without also completing the handshake.

Input Validation

The server parses all incoming data as JSON. Malformed JSON is caught and logged but does not crash the server. However, no validation is performed on the length or content of individual fields (code, payload, pubkey, sig). A malicious client can send a code field of arbitrary length or a payload of several megabytes, and the server will forward it to the peer unchanged.

WebSocket Library Version

The server uses the ws npm package. The version in use should be confirmed against the current security advisory list for ws. Known vulnerabilities in older versions include ReDoS issues in HTTP upgrade parsing. This review did not inspect the installed package version directly.

9. Positive Findings

The following design decisions reflect correct and deliberate security choices that should be explicitly noted.

✓ **Web Crypto API exclusively**	All cryptographic operations use the browser's native crypto.subtle implementation. There are no custom or third-party cryptographic primitives in the protocol path. This minimizes the attack surface from library vulnerabilities.
✓ **Ephemeral key material properly discarded**	ephemeralKeyPair is set to null immediately after deriveSessionKey() completes. The private key becomes inaccessible before any data messages are processed. This is the correct implementation of forward secrecy.
✓ **Constant-time HMAC verification**	Signature verification uses crypto.subtle.verify(), which performs a constant-time comparison. This prevents timing side-channel attacks that could otherwise leak information about the expected HMAC value.
✓ **Counter-bound AAD prevents replay**	The AAD string "keylay-v1|" + counter binds each ciphertext to a specific transmission position. Even if a relay records and retransmits a prior ciphertext, the recvCounter check will reject it. AES-GCM authentication additionally guarantees that modifying the counter field in transit causes decryption failure.
✓ **HKDF info binds key to session participants**	Using the sorted pair of ephemeral public keys as HKDF info ensures that the derived session key is specific to this exact handshake. A different pair of participants, even sharing the same session code, derives a different key.
✓ **X25519 browser compatibility detection**	The app checks for X25519 support before enabling the session join flow and displays a clear warning on unsupported browsers. This prevents silent fallback to weaker or missing cryptography.
✓ **Handshake concurrency lock**	The helloProcessing flag prevents concurrent execution of handleHelloMessage. Without this, two simultaneous incoming hellos could race on ephemeralKeyPair and produce an inconsistent handshake state.
✓ **Security page acknowledges known limitations**	The security.html page lists forward secrecy, replay protection absence (as of its last update), endpoint risks, and code-as-membership semantics as explicit scoped limitations rather than hiding them. This is the correct approach to credible security documentation.

10. Summary Table

ID	Finding	Severity	Status
D-01	PBKDF2 iterations: page says 100k, code uses 300k	**Discrepancy**	Doc outdated
D-02	Replay protection: page says absent, code has it deployed	**Discrepancy**	Doc outdated
D-03	Hello wire format: described format not in current code	**Discrepancy**	Doc outdated
F-01	Counter advanced before decryption — DoS vector	**Moderate**	Open
F-02	No rate limiting or message size enforcement (server)	**Moderate**	Open
F-03	Full message logging including channel hashes (server)	**Low**	Open
F-04	Unauthenticated role claiming, undocumented	**Low**	Open
F-05	No Content Security Policy on client page	**Low**	Open
F-06	No session expiry on server	**Info**	Open
F-07	Modular bias in code generator (0.11 bits loss)	**Info**	Open

11. Conclusions and Recommendations

Keylay's cryptographic design is sound for its stated purpose. The choice of primitives — X25519, HKDF, AES-256-GCM, HMAC-SHA256, PBKDF2 — is appropriate. The implementation uses the Web Crypto API correctly, avoids transmitting key material, properly discards ephemeral private keys, and enforces relay-blindness through derived channel identifiers. The relay correctly limits sessions to two participants.

The security documentation, however, currently describes an older and weaker version of the implementation. Correcting this is the highest-priority action: a security page that accurately documents the current deployment is a fundamental precondition for any credible external review.

Priority Recommendations

Immediate	Update security.html to reflect deployed PBKDF2 iterations (300,000), deployed replay protection, and correct hello wire format. The page must describe the code as it exists, not as it existed previously.
High	Fix the counter-advancement ordering in processDataMessage() to advance recvCounter only after successful AES-GCM decryption. This closes the relay-injectable denial-of-service vector.
High	Add server-side rate limiting, per-IP connection caps, and message payload size limits. Add a client-side cap on messageBuffer.

Medium	Remove full-object logging from server.js. Replace with structured metadata logs that omit the channel hash from production output.
Medium	Add a Content Security Policy header or meta tag to index.html.
Low	Replace modular-reduction code generation with rejection sampling for a provably uniform distribution across the 31-character alphabet.
Low	Add server-side session expiry (maximum age and idle timeout).
Low	Pin the security page to a specific commit or version tag and add a "Last verified against:" date to the page.

This report was produced through manual source review with AI-assisted analysis. It constitutes an internal technical review, not a formal third-party security audit. The findings and recommendations herein should be reviewed by a qualified engineer before being used as the sole basis for security claims in public documentation.