Group Space, LLC

CRYPTOGRAPHIC SECURITY REVIEW

Security review: @group-space/crypto-core v0.1.1

An end-to-end read of the encryption core: key wrapping, sealed-box grants, field and media AEAD, and the recovery path. What holds, what is weak, and two findings proved with running code.

PACKAGE	@group-space/crypto-core v0.1.1
REVIEWED AT	tag v0.1.1, commit a9d495a
SCOPE	src/ (all), sw/decrypt.ts, docs/
FINDINGS	0 High, 4 Medium, 3 Low/Info, 2 proved with code
REVIEWER	Claude Fable 5 (Anthropic), directed by Group Space, LLC
COMMISSIONED	Self-commissioned and self-funded by Group Space, LLC
DATE	26 August 2026

group-space.com

PUBLIC

Scope and provenance

Reviewed	`@group-space/crypto-core` v0.1.1, tag `v0.1.1`, commit `a9d495a`
Files in scope	`src/` (all), `sw/decrypt.ts`, `docs/`
Out of scope	Application code that calls this library; server, transport, and storage
Method	Full manual read of every source file; every committed suite run (97 checks) plus the 34 format vectors and a clean typecheck; two findings reproduced with standalone scripts against this source tree
Coverage	43 exported functions, 25 AAD labels, 5 format constants, 3 protocol documents. See "Technical scope and coverage"
Reviewer	Claude Fable 5 (Anthropic), directed by Group Space, LLC
Commissioned	Self-commissioned and self-funded by Group Space, LLC
Date	26 August 2026

This is an internal review, not a third-party audit. No external firm was engaged and no external party has attacked this code. `docs/threat-model.md` states the same limitation and it still stands after this review.

Severity is rated against the adversary declared in `docs/threat-model.md` as of v0.1.1: a curious or compromised host. That wording may be narrowed in a later release; this review is a record of v0.1.1 and is not restated when it is. Findings are ordered by severity, then by ID.

How to read the reachability column

"Reachable by" is the column that decides whether a finding applies to you. It names the least-privileged party who can trigger the finding at all.

- **Host**: a party with write access to the running server or its database.
- **Storage keys**: a party holding a database backup or object-store bucket, without access to the running server.

No finding in this review is reachable by an application user, a group owner, an invited member, an unauthenticated visitor, or by manipulating a URL or request. Establishing that took the bulk of the review time. Every finding requires the host or its stored data, which is the adversary the protocol was designed against, so each one narrows a defence rather than opening a new door.

Findings

ID	SEVERITY	REACHABLE BY	FINDING	STATUS
M1	MEDIUM	Host	Field ciphertexts are bound to a field type but not to a record, so a host can move one record's sealed field into another record and the client opens it without error	Open, unfixed
M2	MEDIUM	Host or storage keys	The media container binds each frame's index but not the total frame count, so removing whole trailing frames yields a shorter file that decrypts without error	Fix available in 0.2.0, not enforced in the shipping app
M3	MEDIUM	Host or storage keys	Argon2id runs at `INTERACTIVE` cost for long-lived private keys, and no path re-wraps an existing blob at a higher cost	Open, unfixed
M4	MEDIUM	Host	Group Key grants use anonymous sealed boxes and carry no proof of issuer, so a host can seal a key of its own choosing to a member	Open, unfixed
L1	LOW	Host	`encryptChunk` / `decryptChunk` bind the chunk index but no context label, unlike the container path	Fixed after v0.1.1: both helpers removed, unused
L2	LOW	Host	`WrappedSecret` is sealed with no AAD, so two blobs wrapped under the same passphrase are interchangeable	Fix available in 0.2.0, not enabled in the shipping app
L3	INFO	n/a	XChaCha20-Poly1305 is not key-committing; no current flow depends on key commitment	Accepted after v0.1.1, recorded in `docs/threat-model.md`

Nothing was fixed as of `v0.1.1` itself. The status column tracks work landed since. Each finding below describes the code as it stood at `v0.1.1` and is left unedited; where a fix has landed, the current behaviour is in `docs/protocol.md`.

Two fixes ship in the library but are **not turned on in the Group Space app**: M2's container completeness check and L2's key-slot binding. Both defend against the party running the servers and against nobody else, so neither affects whether content stays encrypted. The completeness check works by refusing to display a file it cannot confirm is whole, and Group Space currently makes the opposite trade: show the file rather than withhold it.

M1: Field ciphertexts are not bound to their record

`src/aad.ts`, `src/e2ee.ts` (`encryptField`, `decryptField`)

Every label in the registry is a bare field type, such as `directory.phone`. The AAD therefore binds a ciphertext to *what kind of field it is*, and not to *which record it belongs to*. A host can

copy one member's sealed phone number into another member's directory row; the client decrypts it and displays it as the second member's number. The same applies to post bodies, event fields, and RSVP answers.

Reproduced against `src/`:

```
const gk = await generateGroupKey();
const alice = await encryptField(gk, "555-0101 (Alice)", DIRECTORY_PHONE_AAD);
const shown = await decryptField(gk, alice, DIRECTORY_PHONE_AAD);
// → "555-0101 (Alice)", no error
```

`docs/protocol.md` §5 says context binding stops a ciphertext being replayed elsewhere. That holds between field types and not within one.

Fix. Scope the AAD to the record, for example `directory.phone:<memberId>`. The core already accepts arbitrary AAD strings and `test/crypto-selftest.mts` uses a scoped label (`member:42:phone`). This changes callers and registry guidance, not the wire format. Existing rows re-seal on next write.

One caller constraint decides how far this can go. The identity has to exist at the moment of sealing. Where a record's identifier is assigned by the store after the write, the sealing caller does not have it, and the achievable form is to bind a stable parent that does exist: the thread rather than the post, the member rather than the field. That stops a ciphertext moving between parents and leaves swaps within one parent open, which is a real narrowing rather than a complete answer.

M2: Media containers truncate silently at a frame boundary

`src/e2ee.ts` (`decryptBytes`), `sw/decrypt.ts` (`decryptFrame`)

Each frame's AAD carries its own index, so frames cannot be reordered or moved between files. Nothing carries the total frame count. `decryptBytes` reads frames until the input runs out, so a container with whole trailing frames removed decrypts cleanly and returns a short file. The Poly1305 tag catches truncation inside a frame; truncation on a frame boundary produces no signal.

Reproduced against `src/`: a 768 KB container (3 frames) cut to 2 frames returned 524,288 of 786,432 plaintext bytes and raised no error.

A source comment at `src/e2ee.ts:301` records truncation by a malicious host as out of scope, and adds that "every byte it does serve is authenticated". That second statement is true. A reader can still take it to mean the file is complete, which is a separate claim and not one the format supports.

`docs/threat-model.md` does not mention truncation at all. The limitation is recorded only in a source comment, where nobody deciding whether to trust the library will look for it. Publishing the limitation is part of this finding, not separate from it.

Fix. Bind the total into the container: seal the frame count or plaintext length into frame 0's AAD, or give the final frame a distinct AAD such as `context:i:last` that a decoder requires before returning. Either is additive and readable by a v1 decoder that knows about it.

M3: Argon2id cost is fixed low, with no re-stretch path

`src/e2ee.ts` (`argonDefaults`, `wrapSecret`)

Password wrapping uses libsodium's `INTERACTIVE` limits (ops 2, mem ≈ 64 MB). That is a defensible choice for unlocking on a phone. The wrapped secret is a long-lived account or membership private key, and a stolen database gives an attacker unlimited offline guesses against it.

Recording the parameters per blob is the right design and it is only half the mechanism. Nothing re-wraps an existing blob when the defaults rise, so an account enrolled today keeps interactive cost permanently.

Fix. Raise the cost to `MODERATE` or a tuned value for the account private key and the recovery wrap. Add an upgrade on unlock: when a blob opens and its recorded `ops` / `mem` are below current defaults, re-wrap at the new cost. The stored-parameter design already makes this safe for old blobs.

M4: Group Key grants carry no proof of issuer

`src/e2ee.ts` (`grantGroupKey`, `openGroupKeyGrant`)

`crypto_box_seal` is anonymous by design, and the admission flow depends on that property. The cost is that a grant carries no evidence of who produced it. A host can seal a Group Key it generated to a member's public key. The member opens it, treats it as the group's key, and encrypts subsequent content under a key the host can read.

This sits at the edge of the declared threat model. That document (as of v0.1.1) lists a host serving modified code as explicitly **not** defended against, and is silent on a host serving substituted data to an otherwise honest client. This finding is the second case.

Fix. Authenticate the issuer: sign grants with the granting admin's key, or use `crypto_box` with a known sender for member-to-member grants so the recipient can verify origin. If the current behaviour is accepted instead, `docs/threat-model.md` should say that grant authenticity rests on an honest-issuer assumption.

L1: The chunk helpers bind index but not context

`src/e2ee.ts` (`encryptChunk`, `decryptChunk`)

These exported helpers use `String(index)` as AAD, with no context label, while `encryptBytes` binds `context:index`. Two logical streams under one file key, a photo's `full` and `thumb` for instance, would have interchangeable chunks at matching indices. File keys are random, so a swap needs a shared key, which limits the reach. The inconsistency with the container path is the hazard.

Fix. Give these the same `context:index` AAD, or remove them if the container path has superseded them.

L2: `WrappedSecret` has no context AAD

`src/e2ee.ts` (`wrapSecret`)

Wrapped secrets seal with a null AAD, so two blobs wrapped under one passphrase are interchangeable. Swapping them makes a user unlock the wrong private key. Because keys are random, the visible result is a failure further down rather than a disclosure, so this is a robustness issue.

Fix. Add a purpose or identity AAD to `wrapSecret`, such as the member id or a role tag, so a blob opens only in its intended slot.

L3: The AEAD is not key-committing

Design-wide (XChaCha20-Poly1305)

Poly1305-based AEAD is not key-committing: a ciphertext can be constructed to decrypt validly under two keys (the partitioning-oracle class of attack). No current flow selects a key by testing which one decrypts, so there is no concrete break here. Recorded because it constrains future design.

Fix. Track it. If a key-selection or multi-recipient path is ever added, adopt a committing construction at that point.

What the review confirmed as sound

This table is the positive counterpart to the findings table: the properties checked and upheld. Each row was verified by reading the implementation, not by taking the documentation's word for it. The section "Technical scope and coverage" states how each was checked.

#	PROPERTY	VERDICT
S1	No bespoke cryptography anywhere. Every operation is a documented libsodium or `@stablelib` call	`Upheld`
S2	Recovery codes are unbiased. A 32-symbol alphabet with bytes reduced mod 32 divides 256 exactly, so the 200-bit claim holds	`Upheld`
S3	Nonce handling is correct. Random 24-byte nonces suit XChaCha20's 192-bit nonce space, with no counter state to mismanage	`Upheld`
S4	Key-derivation parameters are recorded per blob, so costs can rise without breaking existing wraps	`Upheld, half-used. See M3`
S5	The two implementations agree. The libsodium core and the `@stablelib` worker path are driven against shared vectors in both directions	`Upheld`

#	PROPERTY	VERDICT
S6	Negative cases fail as required: wrong password, wrong key, wrong AAD, tampered ciphertext, reordered chunks	`Upheld`
S7	Media range arithmetic is exact across chunk boundaries and end-of-file	`Upheld`
S8	The private key never leaves the client in the clear, on any of the four key-lifecycle paths	`Upheld`
S9	The worker client is decrypt-only. It exports no sealer, so a compromised worker cannot compose ciphertext	`Upheld`
S10	The documentation does not overclaim. The threat model names what is undefended	`Upheld, with one gap. See M2`

Technical scope and coverage

What was examined and what was checked in each area. The review was a full manual read, so coverage is the whole library rather than a sample.

Code reviewed

AREA	FILE	SURFACE	WHAT WAS CHECKED
Primitives and container	`src/e2ee.ts`	25 exported functions	Argon2id parameters and salt handling; nonce generation and uniqueness per key; AEAD call arguments and AAD construction; frame layout and boundary handling; recovery-code alphabet and modulo bias; base64 variant consistency
Keychain protocol	`src/account-keys.ts`	10 exported functions	Enrollment, password change, recovery restore, and reset re-enrollment traced end to end for key exposure, re-wrap correctness, and whether a failed step can leave a partial state
Context labels	`src/aad.ts`	25 labels	Every label checked for uniqueness and for what it does and does not bind. Source of M1
Wire constants	`src/media-format.ts`	5 constants	Values checked against the frame layout the encoder and decoder assume, and against the protocol document
Range arithmetic	`src/media-range.ts`	4 exported functions	Plaintext/ciphertext length conversion, chunk span selection, and `Range` header parsing, including empty files, exact multiples, suffix ranges, and unsatisfiable requests
Worker decrypt client	`sw/decrypt.ts`	4 exported functions	AAD construction compared byte for byte against the main implementation; failure returned as a value rather than an exception; confirmed no sealer is exported

AREA	FILE	SURFACE	WHAT WAS CHECKED
Loader	`src/sodium.ts`	1 exported function	Single initialization, no race on repeated calls
Public surface	`src/index.ts`	Re-exports	Checked that nothing test-only is exported to consumers

`encryptBytesWithNonces` is exported for deterministic test vectors and is documented as test-only. That is a deliberate trade and it is correctly labelled; a caller that misuses it would reuse a nonce. It is worth a runtime guard if the library ever ships to third-party consumers.

Documents reviewed

`docs/protocol.md`, `docs/threat-model.md`, and `docs/verify.md` were read against the implementation to check that each claim is met by code. The protocol document matches the code on every constant, shape, and layout. Two statements need attention: §5's claim about replay, narrowed by M1, and the completeness implication discussed in M2.

Tests executed

All suites were run at commit `a9d495a`. Nothing failed.

SUITE	COMMAND	CHECKS	RESULT
Crypto primitives	`npm run test:crypto`	41	Pass
Password re-wrap	`npm run test:rewrap`	6	Pass
Media range arithmetic	`npm run test:media-range`	45	Pass
Cross-implementation interop	`npm run test:interop`	5	Pass
Format vectors	`npm run test:vectors`	34	Pass
Type checking	`npm run typecheck`	Clean	Pass

Total: 131 checks. The media-range suite includes 200 pseudo-random ranges decrypted end to end, and the vector suite pins the wire format against committed digests, which is what makes a silent format change detectable.

Checks applied beyond the suites

- Every AEAD call was read for what it binds as AAD, and the result compared against what an attacker could substitute. This produced M1, M2, and L1.
- Both proved findings were reproduced with standalone scripts run against `src/`, not against a mock or a paraphrase of the code.
- The recovery-code alphabet was checked for modulo bias arithmetically.

- The two independent AEAD implementations were compared for AAD encoding, nonce placement, and failure behaviour.

Not covered

- Application code that calls this library. M1 and L1 depend partly on caller behaviour, which is why both are reported at this layer.
- Server, transport, storage, and build pipeline.
- Side-channel and timing analysis of the underlying primitives. These are libsodium's and `@stablelib`'s responsibility and were not re-examined.
- Formal verification. None was attempted and the project does not claim any.

Suggested order of work

1. M1 with L1 and L2. All three are AAD scoping, they share a theme, and none changes the wire format.
2. M3. Raise the cost for account and recovery wraps, and add re-stretch on unlock.
3. M2. Bind the total length into the container as an additive change.
4. M4 and L3. Both may be acceptable under the current threat model. Record the decision in `docs/threat-model.md` either way.

To reproduce the two proved findings, run `npm test` at `a9d495a` for the baseline, then the M1 and M2 scripts described above against `src/`.