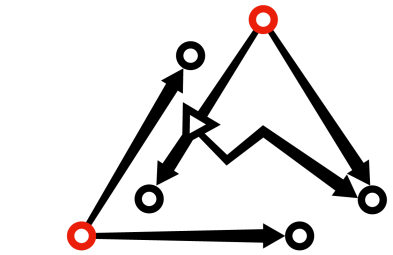

Least Authority

PRIVACY MATTERS

MDK

Security Audit Report

White Noise

Final Audit Report: 20 March 2026

Table of Contents

Overview

Background

White Noise has requested that Least Authority perform a review of the Marmot protocol and security audits of both MDK and White Noise, in three phases. Marmot combines the MLS (Messaging Layer Service) Protocol with Nostr's decentralized network to provide private group messaging without relying on centralized servers or legacy identity systems. MDK is the Marmot Development Kit. White Noise is the Rust backend of the Flutter application, and the `whitenoise` crate uses the MDK library and implements the components required for a messenger application. Our team previously delivered a Feedback Summary on the Marmot Protocol on the 7th of November. In this audit, we focused on the Marmot Development Kit (MDK) codebase.

Project Dates

- **December 1, 2025 - December 17, 2025:** Initial Code Review *(Completed)*
- **December 19, 2025:** Delivery of Initial Audit Report *(Completed)*
- **March 20, 2026:** Verification Review *(Completed)*
- **March 20, 2026:** Delivery of Final Audit Report *(Completed)*

Review Team

- Poulami Das, Security / Cryptography Researcher and Engineer
- Jasper Hepp, Security / Cryptography Researcher and Engineer
- Miguel Quaresma, Security Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the MDK followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- MDK:
  https://github.com/parres-hq/mdk
    - Focus excludes:
        - Flexible storage (due to low risk)
        - Decentralized communication (to be reviewed in Phase 3)

Specifically, we examined the following Git revision for our initial review:

- `94792c1f734da0aae4afd094d35dd4dd37a7559c`

For the verification, we examined the Git revision:

- `a6815c7e61cd5beed615f893770bc6d4b83259e5`

For the review, this repository was cloned for use during the audit and for reference in this report:

- marmot-protocol-mdk:
  https://github.com/LeastAuthority/marmot-protocol-mdk

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Website:
  https://www.whitenoise.chat
- All documentation in this repository:
  https://github.com/marmot-protocol/marmot/tree/master
  - including MIPs 0 through 5, data flows, the threat model, and dependency requirements.

In addition, this audit report references the following documents:

- NIP-59:
  https://github.com/nostr-protocol/nips/blob/master/59.md
- RFC 9420 MLS Protocol:
  https://www.rfc-editor.org/rfc/rfc9420.html
- RFC 9750 MLS Architecture:
  https://www.rfc-editor.org/rfc/rfc9750.html

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Vulnerabilities within each component and whether the interaction between the components is secure;
- Whether requests are passed correctly to the network core;
- Key management, including secure private key storage and management of encryption and signing keys;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the execution;
- Protection against malicious attacks and other ways to exploit;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Marmot combines the MLS (Messaging Layer Service) Protocol with Nostr's decentralized network to provide private group messaging without relying on centralized servers or legacy identity systems. MDK is the Marmot Development Kit and consists of four crates. White Noise is the Rust backend of the Flutter

application, and the `whitenoise` crate uses the MDK library to implement all components required for a messenger application. This review focuses on the MDK library.

System Design

Storage Crates

We reviewed the storage trait and the two crates `mdk-sqlite-storage` and `mdk-memory-storage`.

The crate `mdk_memory_storage` implements the storage trait, wrapping `openmls_memory_storage` and layering `RwLock`-guarded LRU caches for MLS artifacts (groups, `Welcomes`, etc.). It is nonpersistent and intended for fast, thread-safe Nostr MLS storage where data vanishes on process exit. During our review, we identified several issues. The memory storage layer does not consistently enforce invariants, so callers can create "orphan" state (exporter secrets (Issue AL) and per-group message buckets (Issue AN)), and misroute lookups via collisions or stale keys (Issue AC). In the memory backend, entry-count LRU caches combined with unbounded per-key values render those integrity gaps operationally relevant by enabling cache pollution and potential memory exhaustion (Issue AM).

The crate `mdk-sqlite-storage` implements the storage trait, wrapping `openmls_sqlite_storage` with a JSON codec and guarded `rusqlite` connection to persist MLS artifacts. It is the durable SQLite database option for the Nostr MLS state, in contrast to the in-memory backend in the `mdk_memory_storage` crate. We identified several issues during our assessment. Multiple SQLite storage paths perform unbounded reads into memory (full message history and all pending `Welcomes`), enabling straightforward resource-exhaustion denial of service (DoS) via message flooding or mass "pending" rows (Issue Z and Issue AA). This is compounded by accepting and persisting unbounded user-controlled fields, which inflates the database and amplifies JSON (de)serialization costs (Issue AB). Separately, message integrity is at risk because IDs are not scoped by group and writes use `REPLACE`, allowing cross-group overwrites (Issue M). In addition, group listing can fail entirely due to abort-on-first-deserialization-error behavior (Issue AG). We also found that the SQLite database is unencrypted and relies on default permissions (Issue F).

In both crates, we found that including MLS group IDs in error strings creates a metadata-exfiltration path via logs or telemetry (Issue AO).

Crate mdk-core

We reviewed the `mdk-core` crate and compared it against the documentation, including the MIPs, data flows, and the threat model.

The `mdk-core` crate is the Nostr MLS engine built on OpenMLS, handling group flows, messaging, `Welcomes`, key packages, and related MLS artifacts. It supports both the memory and SQLite storage backend.

During our review, we identified several issues in `mdk-core/src/messages.rs` that primarily relate to authorization, identity binding, and deterministic state progression. Commits are merged without checking the MLS sender against `admin_pubkeys` (Issue A), and non-admin devices may still author Commits via auto-Commit of pending proposals (Issue J), while proposals are incorrectly rejected unless originating from admins (Issue I). Message identity is not bound to the MLS-authenticated sender (Issue B), and credential-identity invariants are not enforced in proposal and Commit processing (Issue L). Additional correctness and robustness gaps include nondeterministic selection among same-epoch Commits (Issue K) and missing wrapper and inner validation, as well as missing "failed" persistence on early rejects, enabling tampering and replay-driven reprocessing (Issue AE, Issue AF, and Issue AP).

In `mdk-core/src/welcomes.rs`, we found that `Welcomes` can activate a joiner into an unseen epoch, causing a state fork and decryption failures (Issue AD). In addition, `Welcome` events are ingested without MIP-02 input validation checks, enabling malformed inputs and storage or CPU abuse (Issue H).

In `mdk-core/src/key_packages.rs`, we identified a missing signature check on key package events, enabling attackers to impersonate legitimate users and potentially leading to unauthorized group membership (Issue E).

In `mdk-core/src/encrypted_media` and `mdk-core/src/media_processing`, the components responsible for the encrypted media functionality (MIP-04), we found insufficient input validation, enabling unsupported or malicious files to be uploaded (Issue W). We also identified a nonce reuse vulnerability when the same media is sent more than once within the same epoch (Issue U), as well as a denial-of-service condition when decrypting media sent under a different protocol version (Issue V). In `mdk-core/src/extensions`, we found that `decrypt_group_image` decrypts and authenticates the blob with AEAD but does not verify that the downloaded ciphertext's SHA-256 hash matches the expected `image_hash`. As a result, a storage or network attacker can substitute or replay a different valid ciphertext under the same key and nonce, causing the client to render incorrect media while masking content-address corruption (Issue O).

In the `mdk-core/src/groups.rs` module, we found that `update_group_data` can overwrite the admin set without enforcing invariants, risking persistent admin-state corruption or lockout (Issue C). Admin authorization checks also consult stored `admin_pubkeys` that may lag the MLS state, creating a window in which permissions are evaluated incorrectly (Issue G). The `self_update` path can fail unnecessarily when a cached exporter secret is missing, delaying signer key rotation (Issue N). `KeyPackage` handling lacks a binding check between the `kind-443` event signer and the `BasicCredential.identity`, enabling Nostr identity impersonation and potential privilege misuse (Issue D). The `self_update` path additionally lacks an admin authorization check, allowing any member to originate Commit events and advance epochs under an admin-only Commit policy (Issue P).

Group creation and synchronization exhibit protocol and robustness gaps. The function `create_group` discards the initial `kind:445` Commit and rejects creator-only groups, the update API cannot originate `nostr_group_id` rotation, and `sync_group_metadata_from_mls` silently ignores mandatory extension parse failures while still persisting epoch updates (Issue AH, Issue AJ, Issue AI, and Issue AQ). Commit processing and member removal also present correctness issues. Removed members may fail to process their removal due to missing state synchronization and unconditional exporter-secret export, and `remove_members` may remove the incorrect leaf due to `enumerate()`-derived indices when the tree contains holes (Issue AK and Issue Q).

Dependencies

Running `cargo audit` yielded no issues. Accordingly, our team did not identify any security vulnerabilities resulting from the implementation's use of dependencies.

Code Quality

While the codebase is generally well organized, the number and nature of the identified issues indicate that the implementation is still at an early stage of development.

Tests

The project includes a sufficient number of tests. However, test coverage was not measured as part of this audit.

Documentation and Code Comments

The project includes comprehensive documentation describing the system's intended functionality, including MIPs, a threat model, and data flow documentation. Additionally, the codebase also includes descriptive comments, which aid in understanding the intended behavior of the relevant components.

Scope

At the time of review, several security-relevant features were not yet implemented, such as user-initiated deletion of messages and groups, and multiple features were marked as future TODOs in the codebase. In addition, several findings identified during this review highlight discrepancies between the specification and the implementation, indicating that the codebase is still evolving relative to a more mature specification.

The implementation of NIP-59 gift wrapping, as suggested in MIP-02 (see documentation), is handled outside of `mdk-core` in the `whitenoise` crate. As a result, the correctness of that implementation was not in scope for this audit.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	SEVERITY	STATUS
Issue A: Missing Admin Authorization on MLS Commit Creation and Merge Enables Unauthorized Group Control	Critical	Resolved
Issue B: Authorship Not Bound to MLS Sender Allows Member Impersonation	Critical	Resolved
Issue C: Admin Invariants Bypassed on Group Metadata Update	Critical	Resolved
Issue D: Identity Binding Bypass in KeyPackage Handling Enables Impersonation of Nostr Keys	Critical	Resolved
Issue E: Missing Verification of Nostr Event Signature in parse_key_package	Critical	Determined Non-Issue
Issue F: Unencrypted SQLite Storage and Permissive File Modes Expose MLS State	Critical	Resolved
Issue G: Admin Authorization Uses Stale Stored Metadata Instead of MLS State	High	Resolved
Issue H: Missing MIP-02 Validation for Welcome Events	High	Resolved
Issue I: Proposals Are Incorrectly Restricted to Admins	High	Resolved
Issue J: Non-Admin Commit of Proposals Violates MIP-03	High	Resolved
Issue K: No Deterministic Commit Race Resolution	High	Resolved

Issue L: Identity Change Validation Missing in Proposal and Commit Processing	High	Resolved
Issue M: Messages Overwritten Across Groups Due to Non-Scoped Primary Key and REPLACE Writes	High	Resolved
Issue N: self_update Requires Cached Exporter Secret, Blocking Key Rotation	High	Resolved
Issue O: Missing Hash Verification in decrypt_group_image Allows Storage-Level Blob Substitution	High	Resolved
Issue P: Missing Admin Authorization in self_update Allows Unauthorized Commits	High	Resolved
Issue Q: Incorrect Member Removal Due to Enumerated Index Misuse	High	Resolved
Issue R: Deprecated Hex Encoding for KeyPackage Content Enables Downgrade and Interop Failures	High	Resolved
Issue S: Missing Validation of Mandatory Relays Tag in MLS KeyPackage Events	High	Resolved
Issue T: Incomplete MIME Type Canonicalization in validate_mime_type	High	Resolved
Issue U: Deterministic Nonce Derivation Causes Nonce Reuse and Message Linkability	High	Resolved
Issue V: Hard-Coded Scheme Label in AAD Causes Version Mismatch and Decryption Denial of Service (DoS)	High	Resolved
Issue W: MIME Type Spoofing and Missing Allowlist in Media Encryption	High	Resolved
Issue X: Default LastResort KeyPackage Enables Multiple Use Against MLS Guidance	High	Unresolved
Issue Y: Missing Zeroization	High	Resolved
Issue Z: Unbounded Function messages Query Enables Memory Exhaustion Denial of Service	Medium	Resolved
Issue AA: Unbounded Loading of Pending Welcomes Enables Memory-Exhaustion Denial of Service	Medium	Resolved
Issue AB: Unbounded User Input Stored in SQLite Causes Disk and CPU Exhaustion	Medium	Resolved
Issue AC: nostr_group_id Cache Collision Leaves Stale Keys and Enables Lookup Hijack	Medium	Resolved
Issue AD: Premature Welcome Delivery Allows Joiner Entry Into	Medium	Resolved

Unseen Epoch (State Fork)		
Issue AE: Missing Nostr-Based Validations When Processing Messages	Medium	Resolved
Issue AF: Inner Nostr Kind Not Validated in MLS Chat Messages	Medium	Determined Non-Issue
Issue AG: Function all_groups Aborts on First Deserialization Error	Medium	Resolved
Issue AH: Initial Commit Not Published as Kind:445 on Group Creation Before Welcome	Medium	Determined Non-Issue
Issue AI: Update API Omits nostr_group_id Update Allowed by Specification	Medium	Resolved
Issue AJ: Creator-Only Group Creation Rejected	Medium	Resolved
Issue AK: Removed Member Commit Processing Breaks Due to Missing Group State Sync and Unconditional Exporter Secret Export	Medium	Resolved
Issue AL: Exporter Secrets Saved and Retrieved for Nonexistent Group	Low	Resolved
Issue AM: Missing Input Validation Enables Memory Exhaustion	Low	Resolved
Issue AN: Unvalidated Group ID Allows LRU Cache Pollution in save_message	Low	Resolved
Issue AO: MLS Group ID Leakage in Error Messages	Low	Resolved
Issue AP: Early Validation Failures Omit Failed ProcessedMessage State	Low	Resolved
Issue AQ: sync_group_metadata_from_mls Silently Ignores Mandatory Group-Data Extension Parse Failure	Low	Resolved
Suggestion 1: Update Error Message in mdk-memory-storage	Informational	Resolved
Suggestion 2: Propagate last_message_id Parse Errors in row_to_group	Informational	Resolved
Suggestion 3: Implement KeyPackage Deletion After Welcome Acceptance	Informational	Resolved
Suggestion 4: Implement Welcome Retry With Same wrapper_event_id After Failed Preview	Informational	Resolved
Suggestion 5: Prevent Panic Risk in process_welcome	Informational	Resolved
Suggestion 6: Address Performance-Related TODO in mdk-memory-storage	Informational	Resolved

Suggestion 7: Improve Code Quality in Encrypted Media	Informational	Resolved

Issue A: Missing Admin Authorization on MLS Commit Creation and Merge Enables Unauthorized Group Control

Location
mdk-core/src/messages.rs#L486

Synopsis
The Commit-processing path accepts and merges Commits without verifying the sender against `admin_pubkeys`, permitting non-admin members to modify governance metadata and membership.

Impact
High.

A non-admin member can add its key to `admin_pubkeys`, remove or add members, or alter relays and `nostr_group_id`, compromising governance and integrity.

Feasibility
High.

Only current group membership and the ability to deliver Commit messages through existing processing paths are required.

Severity
Critical.

Preconditions
Attacker membership in the MLS group is required.

Technical Details
The function `process_decrypted_message` routes `ProcessedMessageContent::StagedCommitMessage` to the function `process_commit_message_for_group`, which immediately calls the function `merge_staged_commit` and then the function `sync_group_metadata_from_mls` to persist `GroupContext`-derived extensions. Unlike the function `process_proposal_message_for_group`, which validates privileges via the function `is_member_admin`, the Commit path performs no sender authorization against `admin_pubkeys`. This deviates from the trust boundary listed in the threat model, which states "*Only users listed in the admin_pubkeys array can commit group state changes*."

An attacker can craft a valid Commit that encodes inline proposal inclusion or `GroupContext` extension changes for `admin_pubkeys`, `relays`, or `nostr_group_id` and broadcast it. Recipients may merge the Commit, advance the epoch and exporter secrets, and persist the attacker's governance changes, enabling privilege escalation and member eviction.

Remediation
We recommend adding this check on the admin pubkey in the function `process_commit_message_for_group`.

Status

The White Noise team has resolved the issue by adding the recommended check while still allowing updates that only modify the sender's own leaf.

Verification

Resolved.

Issue B: Authorship Not Bound to MLS Sender Allows Member Impersonation

Location

`mdk-core/src/messages.rs#L299`

Synopsis

The function `process_application_message_for_group` accepts `rumor.pubkey` from the decrypted `UnsignedEvent` as the stored author without validating it against the MLS-authenticated sender identity.

Impact

High.

An attacker can attribute arbitrary content to another group member, leading to message spoofing, reputational damage, and phishing within the group.

Feasibility

High.

A valid group member who can send MLS application messages can set `rumor.pubkey` to any target member's key without special privileges.

Severity

Critical.

Preconditions

The attacker must be a current MLS group member able to send application messages.

Technical Details

The function `process_application_message_for_group` converts an `ApplicationMessage` into bytes, deserializes it into an `UnsignedEvent`, constructs a `message_types::Message`, and writes `rumor.pubkey` into `Message.pubkey` without binding it to the MLS sender derived from the processed message. Authorship therefore derives from attacker-controlled plaintext within the MLS-protected payload rather than from the sender identity authenticated by MLS credentials, breaking identity binding.

An attacker can craft a valid MLS application message where `UnsignedEvent.pubkey` equals the victim's Nostr `PublicKey` and broadcast it. Although MLS authenticates the sender at the protocol layer, the code does not compare the authenticated MLS sender to `rumor.pubkey`, so recipients persist and display forged authorship.

Remediation

We recommend replacing the use of `rumor.pubkey` for authorship and deriving the expected Nostr `PublicKey` from the MLS sender's `BasicCredential`, enforcing equality within

`process_application_message_for_group` and rejecting processing on mismatch with a recorded failure state.

Status

The White Noise team has resolved the issue by introducing a new function, `verify_rumor_author`.

Verification

Resolved.

Issue C: Admin Invariants Bypassed on Group Metadata Update

Location

`crates/mdk-core/src/groups.rs#L667`

`crates/mdk-core/src/groups.rs#L748`

Synopsis

The function `update_group_data` permits an admin to replace `group_data.admins` without validation, which subsequently becomes authoritative in `Group.admin_pubkeys` and drives all admin authorization checks.

Impact

High.

An attacker with admin privileges can set an empty or invalid admin set, which may permanently disable administrative actions and corrupt authorization state across the group.

Feasibility

High.

Any existing admin can call `update_group_data` and merge one Commit. No additional privileges are required.

Severity

Critical.

Preconditions

For this issue to be exploitable, the following must be in place:

- An attacker must be an existing admin;
- The Commit must be merged; and
- The function `sync_group_metadata_from_mls` must run.

Technical Details

The function `update_group_data` assigns `NostrGroupDataUpdate.admins` directly to `group_data.admins` without invoking the function `validate_group_members` or any equivalent checks. After the Commit is merged, the function `sync_group_metadata_from_mls` copies this set into `Group.admin_pubkeys`, which is consumed by the functions `is_leaf_node_admin` and `is_member_admin` for authorization.

An admin can call `update_group_data` with admins set to an empty set or to public keys that are not current members. After merge and `sync_group_metadata_from_mls`, no member would satisfy

admin checks, or arbitrary keys would appear as admins. This can produce persistent lockout or inconsistent authorization semantics across the MLS state and stored metadata.

Remediation

We recommend replacing the direct assignment in the function `update_group_data` with a validation step that enforces creation invariants against the current membership from `get_members`, rejects empty and nonmember admin sets, and only then updates `Group.admin_pubkeys` via `sync_group_metadata_from_mls`.

Status

The Whitenoise team has resolved the issue by introducing a new function `validate_admin_update`.

Verification

Resolved.

Issue D: Identity Binding Bypass in KeyPackage Handling Enables Impersonation of Nostr Keys

Location

`crates/mdk-core/src/groups.rs#L894`

`crates/mdk-core/src/groups.rs#L500`

Synopsis

The code accepts a KeyPackage whose `BasicCredential.identity` claims a victim's Nostr public key without verifying that the surrounding `kind-443` event is signed by that key, enabling member impersonation.

Impact

High.

An attacker may join under a victim's Nostr key and, if that key is an admin, perform privileged operations while `get_members` and related logic attribute actions to the victim, causing impersonation and potential privilege escalation.

Feasibility

High.

The attacker is required only to craft a valid MLS KeyPackage and publish a `kind-443` event. No elevated privileges are required beyond submitting the event that a group admin processes.

Severity

Critical.

Preconditions

For this issue to be possible, the following must be in place:

- A KeyPackage must exist with `BasicCredential.identity` set to the victim's Nostr public key;
- The attacker must publish a `kind-443` event carrying the KeyPackage that is signed by the attacker's Nostr key; and

- A group admin must process the event via `create_group` or `add_members`.

Technical Details

The function `parse_key_package` is called on `kind-443` events in `create_group` and `add_members`, and downstream functions such as `get_members`, `is_member_admin`, `is_leaf_node_admin`, `pubkey_for_member`, and `pubkey_for_leaf_node` extract the Nostr key from `BasicCredential.identity` and treat it as authoritative. There is no identity binding between the `kind-443` event signer public key and `BasicCredential.identity`, and no Nostr signature over the serialized `KeyPackage` is verified, which creates a signer and credential mismatch.

An attacker can embed a legitimate MLS `KeyPackage` whose `BasicCredential.identity` equals a victim's Nostr public key within a `kind-443` event signed with the attacker's key. When a group admin adds this `KeyPackage`, the attacker joins the MLS tree but is reported as the victim by `get_members`, and admin checks (`is_member_admin` and `is_leaf_node_admin`) succeed if the victim is an admin, enabling unauthorized admin actions.

Remediation

We recommend replacing the implicit trust in `BasicCredential.identity` and adding a binding check in `parse_key_package` that verifies the `kind-443` event public key equals the credential identity and that the event signature covers the serialized `KeyPackage`, rejecting on mismatch.

Status

The Whitenoise team has resolved the issue by adding the identity binding check within the `parse_key_package` function.

Verification

Resolved.

Issue E: Missing Verification of Nostr Event Signature in parse_key_package

Location

`crates/mdk-core/src/key_packages.rs#L217`

Synopsis

The function `parse_key_package` accepts and parses a Nostr event without verifying the event's Schnorr signature, which permits spoofed or tampered `key-package` events to be treated as authentic.

Impact

High.

An attacker may impersonate a target Nostr identity and cause invitations and `Welcome` messages to be addressed to attacker-controlled MLS `KeyPackages`, which can lead to unauthorized group membership and confidentiality loss.

Feasibility

High.

No privileges are required. Any party or relay can forge an event with valid tags and content when the signature is not verified.

Severity
Critical.

Preconditions
For this issue to occur, the following must hold true:

- The function `parse_key_package` must be called on events received from untrusted relays;
- Signature verification for the `Event` must be omitted by the caller; and
- The caller must attribute the key package to the event's `pubkey` or use it for invitations.

Technical Details
The function `parse_key_package` validates `Kind`, tags, encoding, and the MLS `KeyPackage` via `KeyPackageIn::validate`, but it does not verify the Nostr event signature, and it does not bind the event author key to the credential identity. This omits the trust boundary between the Nostr layer and the MLS credential, so a forged or modified event can carry an attacker-generated `KeyPackage` whose `BasicCredential` identity bytes claim an arbitrary Nostr public key.

An attacker can craft an `Event` of `kind-443` with compliant MIP-00 tags and a valid, self-signed MLS `KeyPackage`, but set the credential identity to a victim's Nostr public key and publish or relay it. Since `parse_key_package` does not call `Event::verify` nor compare `event.pubkey` with the result of `parse_credential_identity`, the forged event would be accepted and the attacker would receive the `Welcome` encrypted to their HPKE key while appearing as the victim.

Remediation
We recommend verifying the Nostr event using the method `Event::verify` and rejecting on failure. The event's `pubkey` should then be compared to the parsed credential identity via `parse_credential_identity`, rejecting on mismatch before returning the `KeyPackage`.

Status
The Whitenoise team has pointed to the upstream code where the missing validation is present. The `nostr-sdk` crate calls `Event::verify_with_ctx` before passing the event to the code when it fetches the `key_package` (and all other types) events from relays. Any event that does not validate (both the ID and the signature) throws an error and is not returned to the caller.

Verification
Determined Non-Issue.

Issue F: Unencrypted SQLite Storage and Permissive File Modes Expose MLS State

Location
`crates/mdk-sqlite-storage/src/lib.rs#L77`

Synopsis
The MLS state is stored in an unencrypted SQLite database and relies on the default `umask` for permissions, which may result in the file being world-readable.

Impact
High.

Local adversaries may exfiltrate MLS state data, such as messages, group metadata, and exporter secrets, which breaks confidentiality and enables retrospective traffic decryption.

Feasibility
High.

An unprivileged local user with filesystem access can copy or read the database file.

Severity
Critical.

Preconditions
For this issue to occur, the following must hold true:

- Local filesystem or backup access must be available; and
- The database file must be unencrypted and lack restrictive access permissions.

Technical Details
The function `MdkSqliteStorage::new` uses `rusqlite::Connection::open` and constructs `SqliteStorageProvider::new` with the `JsonCodec`, which serializes and writes plaintext records to disk. No SQLCipher configuration, key derivation, or `PRAGMA` is applied to set an encryption key to the connection. The function `MdkSqliteStorage::new` also does not set restrictive permissions via `std::fs::set_permissions` or platform-specific `OpenOptionsExt`. As a result, on Unix-like systems with `umask 022` and a world-executable parent directory, the file often has mode `0600`, exposing tables such as `group_exporter_secrets` and `messages` to local users.

Remediation
We recommend replacing plaintext storage by using an SQLite encryption extension such as SQLCipher with an `Argon2id`-derived key applied to every `Connection`. In addition, the directory should be explicitly created with mode `0700` and the database file with mode `0600` in the function `MdkSqliteStorage::new`.

Status
The White Noise team has resolved the issue by adopting SQLCipher-encrypted SQLite as the default and explicitly hardening database and directory permissions (0600/0700).

Verification
Resolved.

Issue G: Admin Authorization Uses Stale Stored Metadata Instead of MLS State

Location
`crates/mdk-core/src/groups.rs#L1129`

Synopsis
Admin permission checks read a stale copy of `admin_pubkeys` from storage rather than deriving the admin set from the current MLS group context and extension, creating a race window with incorrect authorization.

Impact
High.

An attacker may add or remove members or mutate group metadata contrary to the MLS admin policy until state converges.

Feasibility
Medium.

An attacker must be a current member with previously granted admin status and act before the function `merge_pending_commit` is called.

Severity
High.

Preconditions
For this issue to occur, the following must hold:

- An admin change in MLS must have been committed and published;
- The variable `stored_group.admin_pubkeys` must be stale; and
- The attacker must hold valid membership keys and must not yet have processed the demotion locally.

Technical Details
The functions `is_leaf_node_admin` and `is_member_admin` authorize by consulting the variable `stored_group.admin_pubkeys` rather than the current MLS state or the `NostrGroupDataExtension`. The variable `admin_pubkeys` is synchronized only via the function `sync_group_metadata_from_mls`, which is invoked from the function `merge_pending_commit` after publishing the `Kind:445` Commit, so authorization decisions can lag behind the MLS state.

A recently demoted admin can call the functions `add_members`, `remove_members`, or `update_group_data` before `merge_pending_commit` and pass the stale check, generating a valid Commit event. That Commit may propagate and be processed by peers during the window, resulting in unauthorized membership or metadata changes.

Remediation
We recommend replacing checks against the variable `stored_group.admin_pubkeys` and instead using the admin set derived from the current MLS group context extension `NostrGroupDataExtension` on the in-memory `MlsGroup` at authorization time.

Status
The Whitenoise team has resolved the issue by updating the admin authorization checks as recommended.

Verification
Resolved.

Issue H: Missing MIP-02 Validation for Welcome Events

Location
`crates/mdk-core/src/welcomes.rs#L57`

crates/mdk-core/src/welcomes.rs#L405

Synopsis

The function `process_welcome` accepts any `UnsignedEvent` without validating the MIP-02 structure, `kind 444`, required tags, or encoding, which allows malformed or noncompliant events to be parsed and stored.

Impact

Medium.

An attacker could cause storage pollution and resource exhaustion through malformed `Welcomes`, and, if a syntactically valid but malicious MLS `Welcome` is accepted, create bogus pending groups and misleading metadata.

Feasibility

High.

Any party able to deliver a gift-wrapped `UnsignedEvent` to the function `process_welcome` can attempt the attack.

Severity

High.

Preconditions

For this issue to occur, the following must apply:

- The caller must not validate the MIP-02 structure before invoking `process_welcome`; and
- Relays or wrappers must forward attacker-controlled events.

Technical Details

The function `process_welcome` calls the function `preview_welcome` directly on `rumor_event` without checking that `rumor_event.kind == Kind::MlsWelcome (444)`, the presence and order of tags `relays`, e, and `client`, or that tag values are nonempty. The function `ContentEncoding::from_tags` infers encoding from `welcome_event.tags`, but the code does not reject invalid `encoding` values. Absence of the tag defaults to hex, which deviates from MIP-02.

An attacker can submit unsigned events with incorrect kind or malformed tags. This triggers decode and parse attempts in the functions `decode_welcome_content` and `parse_serialized_welcome`, causing failed `ProcessedWelcome` writes and CPU and I/O load. If the MLS payload parses successfully, it can also result in storage of a pending group with attacker-controlled metadata from the OpenMLS `StagedWelcome`.

Remediation

We recommend adding a helper validator, for example, the function `validate_welcome_event`, and using it at the start of the function `process_welcome` before calling the function `preview_welcome`. The implementation should also mirror the checks in the test `test_welcome_event_structure_mip02_compliance` and return a specific error.

Status

The White Noise team has resolved the issue by introducing further checks.

Verification
Resolved.

Issue I: Proposals Are Incorrectly Restricted to Admins

Location
`crates/mdk-core/src/messages.rs#L372`

Synopsis
The function `process_proposal_message_for_group` rejects proposals from non-admin members (`Error::ProposalFromNonAdmin`), while the documentation explicitly states that "*Any member can create proposals*" and only admins can commit them.

Impact
Medium.

Non-admin members cannot submit legitimate proposals such as self key updates or metadata proposals, which limits required maintenance flows and degrades the security posture by preventing timely key rotation.

Feasibility
High.

Severity
High.

Technical Details
The function `process_proposal_message_for_group` matches on `Sender::Member` and calls the function `is_member_admin`. If it returns `false`, the function returns `Error::ProposalFromNonAdmin` and drops the proposal. This enforces an admin-only proposal policy that contradicts the documented flow, which specifies that any member may create proposals and that admin authorization applies at commit time.

Remediation
We recommend replacing the admin-only gate in `process_proposal_message_for_group` and using membership validation instead.

Status
The White Noise team has resolved the issue as recommended.

Verification
Resolved.

Issue J: Non-Admin Commit of Proposals Violates MIP-03

Location
`crates/mdk-core/src/messages.rs#L358`

Synopsis

A group member who is not an admin automatically generates and publishes a Commit for an admin-authored proposal via `commit_to_pending_proposals`, violating the admin-only Commit rule defined by MIP-03.

Impact

Medium.

A non-admin member may advance the group epoch and apply administrative changes by publishing a Commit that peers accept without verifying admin authorship, which weakens authorization guarantees and can trigger unintended state transitions.

Feasibility

High.

Any authenticated group member who processes a valid admin proposal and can publish events can create and broadcast the Commit. No additional privileges required.

Severity

High.

Technical Details

The function `process_proposal_message_for_group` validates the proposal sender with the function `is_member_admin`, but then uses the function `load_mls_signer` to obtain the local signer. Subsequently, it calls `mls_group.commit_to_pending_proposals`, producing a Commit authored by the local device even if it is not in the variable `admin_pubkeys`. This is in violation of the flow defined in MIP-03.

Remediation

We recommend replacing the unconditional call to `commit_to_pending_proposals` in `process_proposal_message_for_group` with an authorization check that verifies the local signer obtained from `load_mls_signer` in `admin_pubkeys`. In addition, we recommend adding an admin verification check within `commit_to_pending_proposals`.

Status

The White Noise team has resolved the issue as recommended.

Verification

Resolved.

Issue K: No Deterministic Commit Race Resolution

Location

`crates/mdk-core/src/messages.rs#L969`

Synopsis

Concurrent MLS Commit events for the same epoch are resolved by arrival order rather than the deterministic ordering required by MIP-03.

Impact

Medium.

Competing Commits may cause permanent group state divergence across clients, resulting in inconsistent membership and loss of decryptability for subsequent messages.

Feasibility
High.

A group admin can publish multiple valid Commits for the same epoch to different relays, and clients that receive them in different orders will apply different winners.

Severity
High.

Preconditions
For this issue to occur, the following must hold true:

- The attacker must be a group admin;
- Multiple valid Commit Group Events must be published for the same `mls_group_id` and epoch; and
- Different clients must observe different relay ordering for the competing Commit events.

Technical Details
The function `process_message` processes a `ProcessedMessageContent::StagedCommitMessage` by immediately calling the function `process_commit_message_for_group`, which merges the `StagedCommit` via `merge_staged_commit` without comparing the wrapper event `created_at` or `id` against other candidates for the same epoch. The function `handle_message_processing_error` treats later competing Commits as `Error::ProcessMessageWrongEpoch` and persists them as `ProcessedMessageState::Failed` with the failure reason "`Epoch mismatch`," which permanently discards information required for deterministic race resolution.

A malicious admin can craft two different Commits from the same starting epoch and publish them to different relays such that subsets of clients observe opposite arrival orders. Each subset applies the first-seen Commit and subsequently rejects the other as an epoch mismatch, producing divergent exporter secrets and incompatible group states that do not converge without out-of-band recovery.

Remediation
We recommend persisting competing Commit wrapper events per `mls_group_id` and epoch, ordering them deterministically by `created_at` and lexicographic `id`, applying only the selected winner, and explicitly marking all others as rejected rather than processing them by arrival order.

Status
The White Noise team has resolved the issue as recommended.

Verification
Resolved.

Issue L: Identity Change Validation Missing in Proposal and Commit Processing

Location
`crates/mdk-core/src/messages.rs#L372`

crates/mdk-core/src/messages.rs#L486

Synopsis

The implementation does not reject Proposal or Commit objects that attempt to modify MLS credential identity fields, contrary to MIP-00 requirements.

Impact

High.

An attacker could change the binding between a member and the Nostr public key identity, which may enable impersonation, misattribution, and persistent group state corruption.

Feasibility

Medium.

An authenticated admin can submit an update that alters the `BasicCredential.identity`, and the current process would accept it without explicit validation.

Severity

High.

Preconditions

For this issue to occur, the following must hold true:

- The attacker must be a current group admin; and
- A Proposal or Commit must carry a `LeafNode` or `Credential` with an altered `BasicCredential.identity` value.

Technical Details

The function `process_proposal_message_for_group` stores a `QueuedProposal` and calls `MlsGroup::commit_to_pending_proposals` without comparing the incoming `BasicCredential.identity` bytes in proposed `LeafNode` updates to the stored member identity. The function `process_commit_message_for_group` merges a `StagedCommit` via `merge_staged_commit` without any application-level identity invariance check. MIP-00 mandates immutable identity fields and validation that rejects proposals and commits attempting to alter them.

Exploit sketch: An admin can submit an `Update` or `Add` whose `Credential` contains a different 32-byte Nostr public key in `identity`. The code would accept and merge it, changing the member-to-identity mapping. Subsequent messages would validate at the MLS layer but would be attributed to the incorrect identity, enabling identity takeover within group metadata and policy logic that relies on identity.

Remediation

We recommend adding explicit checks in the functions `process_proposal_message_for_group` and `process_commit_message_for_group` to compare incoming `BasicCredential.identity` values against stored identities for the affected leaf and reject any Proposal or Commit that modifies identity.

Status

The White Noise team has resolved the issue as recommended.

Verification

Resolved.

Issue M: Messages Overwritten Across Groups Due to Non-Scoped Primary Key and REPLACE Writes

Location
crates/mdk-sqlite-storage/migrations/V100__initial.sql#L45

crates/mdk-sqlite-storage/src/messages.rs#L29

crates/mdk-sqlite-storage/src/messages.rs#L57

crates/mdk-core/src/messages.rs#L130

Synopsis
Messages from different groups overwrite each other because the table keys by the inner event ID and the write path replaces on conflict.

Impact
High.

An attacker or faulty relay can cause message loss and misattribution across groups, which may corrupt group history and hide prior content.

Feasibility
Medium.

A group member or relay that can submit a message to a second group can reuse a deterministic rumor ID to trigger an overwrite.

Severity
High.

Preconditions
For this issue to occur, the following must hold true:

- The same message `id` must be reused across distinct `mls_group_id` values; and
- An adversary must be able to deliver the inner event to another group storage instance.

Technical Details
The schema defines `messages.id` as the primary key, not scoped by `mls_group_id` (here). The function `save_message` writes with `INSERT OR REPLACE` using the provided `id`, `mls_group_id`, and `wrapper_event_id`, so any duplicate `id` replaces the existing row regardless of group (here). The function `find_message_by_event_id` performs lookups using only `id`, so the most recent replacement wins (here).

Rumor IDs are deterministically derived via the function `rumor.ensure_id()` before being sent (here). An actor can repost the same inner event into a second group, causing `save_message` to overwrite the first group's row and misattribute or hide it on subsequent reads by `id`.

Remediation
We recommend replacing the keying strategy by defining messages group-scoped with `PRIMARY KEY (mls_group_id, id)` and using `INSERT ... ON CONFLICT(mls_group_id, id) DO UPDATE`

that updates only when the existing row has the same `mls_group_id`. Queries and APIs should also be updated to use both `mls_group_id` and `id`.

Status

The White Noise team has resolved the issue as recommended.

Verification

Resolved.

Issue N: self_update Requires Cached Exporter Secret, Blocking Key Rotation

Location

`crates/mdk-core/src/groups.rs#L977`

Synopsis

The function `self_update` aborts with `GroupExporterSecretNotFound` when the current epoch's exporter secret is missing from storage, even though the secret is only used for logging and is not required for the key rotation.

Impact

Medium.

This condition prevents timely signer key rotation, which may extend the window of misuse if a member's signing key is compromised and degrades post-compromise security.

Feasibility

High.

Any group member invoking the function `self_update` without a cached exporter secret for the current epoch, or after cache loss, can trigger the failure. No privileges beyond group membership are required.

Severity

High.

Preconditions

For this issue to occur, the following must hold:

- The current epoch's `GroupExporterSecret` must be absent from storage; and
- A group member must call the function `self_update`.

Technical Details

In the function `self_update`, the variable `current_secret` is loaded via `self.storage().get_group_exporter_secret(...)` and unwrapped with `.ok_or(Error::GroupExporterSecretNotFound)?`, which aborts when the cache lacks the exporter secret for `mls_group.epoch()`. The secret is then used only for `tracing::debug` logging of the epoch and is not required to construct the Commit produced by `self_update_with_new_signer` or to rotate the signature keypair, introducing a spurious dependency on a cached exporter key.

A new member who has not called the function `exporter_secret` cannot rotate the signer using the function `self_update` until the cache is populated, and cache eviction or storage reinitialization produces the same abort, delaying expected self-healing actions.

Remediation

We recommend removing the mandatory fetch of `current_secret` in the function `self_update` and instead using optional retrieval for logging, or using `mls_group.epoch()` for logging, allowing the rotation to proceed when the exporter secret is absent.

Status

The Whitenoise team has resolved the issue by updating the catching mechanism as recommended.

Verification

Resolved.

Issue O: Missing Hash Verification in decrypt_group_image Allows Storage-Level Blob Substitution

Location

`crates/mdk-core/src/extension/group_image.rs#L202`

Synopsis

The function `decrypt_group_image` does not verify the downloaded blob's SHA-256 against the expected `image_hash` from the extension, which breaks the content-addressed invariant required by MIP-04 and permits content-address mismatch.

Impact

High.

An attacker controlling Blossom or intercepting traffic can substitute or replay a different encrypted blob that still authenticates under the key and nonce, which may cause incorrect media to be rendered and masks storage corruption.

Feasibility

Medium.

Attack requires control of the storage response or network path and the availability of a ciphertext that authenticates under the victim's `image_key` and `image_nonce`.

Severity

High.

Preconditions

For this issue to occur, the following must hold:

- The attacker must control the Blossom response or be able to perform on-path tampering;
- The caller must use the function `decrypt_group_image` without validating `image_hash` externally; and
- The extension must provide `image_hash`, `image_key`, and `image_nonce`.

Technical Details

The module computes `encrypted_hash` during encryption, and the struct `GroupImageEncryptionInfo` carries the expected `image_hash`, but the function `decrypt_group_image` accepts only `encrypted_data`, `image_key`, and `image_nonce` and does not check `Sha256(encrypted_data)` against `image_hash`. The error variant `HashVerificationFailed` exists but is not raised, so the AEAD tag is the only integrity check, which authenticates to the key but not to the content-address identifier.

An attacker operating Blossom can return an old ciphertext that was produced for the same `image_key` and `image_nonce`, causing the function to decrypt and render data that does not correspond to the declared `image_hash`. If no authenticating ciphertext is available, the attacker can still induce `DecryptionFailed`, producing denial of service while the client lacks an explicit content-address mismatch signal.

Remediation

We recommend verifying `Sha256(encrypted_data)` against the expected `image_hash` from `GroupImageEncryptionInfo` before attempting decryption and returning `HashVerificationFailed` on mismatch. This can be achieved by extending the function `decrypt_group_image` to accept `image_hash` or by adding a wrapper that performs this check.

Status

The Whitenoise team has resolved the issue by adding hash verification as recommended.

Verification

Resolved.

Issue P: Missing Admin Authorization in self_update Allows Unauthorized Commits

Location

`crates/mdk-core/src/groups.rs#L977`

Synopsis

The function `self_update` creates and returns a Commit event without validating administrative privileges, enabling any member to originate Commits despite the admin-only Commit policy.

Impact

Medium.

An unprivileged member may advance the group epoch and rotate exporter-derived NIP 44 keys by publishing unauthorized Commits, which may cause policy violations and operational disruption.

Feasibility

High.

A valid group member with client access can call the function and publish the returned event.

Severity

High.

Preconditions

For this issue to be exploitable, the following conditions must be met:

- The attacker must have valid membership in the target MLS group;
- The attacker must have access to a client context that exposes the `self_update` function;
- An admin-only Commit policy must be in effect; and
- The attacker must be able to publish the `evolution_event` to the group relays.

Technical Details

The function `self_update` loads the current signer via the function `load_mls_signer`, generates a new `SignatureKeyPair`, and calls the function `mls_group.self_update_with_new_signer` to produce a Commit. It then serializes the Commit, calls the function `build_encrypted_message_event` to create the `evolution_event`, records a `ProcessedMessage`, and returns the event. Unlike the functions `add_members`, `remove_members`, and `update_group_data_extension`, it performs no `is_leaf_node_admin` authorization check before generating the Commit event.

A non-admin member can call `self_update`, obtain a valid Commit, and publish the `evolution_event` to relays. Other members may process the Commit, advancing the epoch and rotating the exporter secret used for NIP 44 encryption, thereby violating the admin-only Commit policy and enabling repeated unauthorized state changes.

Remediation

We recommend adding an admin authorization check in the function `self_update` using the function `is_leaf_node_admin`, and adopting a proposal workflow for non-admins that requires an admin to originate the Commit.

Status

The Whitenoise team has resolved the issue by adding an admin authorization check through the function `is_leaf_node_admin`.

Verification

Resolved.

Issue Q: Incorrect Member Removal Due to Enumerated Index Misuse

Location

`crates/mdk-core/src/groups.rs#L595`

Synopsis

The function `remove_members` derives `LeafNodeIndex` from `enumerate()` instead of each member's actual leaf index, which causes removal of the incorrect leaf when the ratchet tree has holes.

Impact

High.

An administrator or caller may unintentionally or maliciously remove unintended members, including administrators, which may cause unexpected membership changes and loss of availability.

Feasibility

Medium.

Exploitation requires admin privileges and a group tree with holes, which could occur after prior removals.

Severity
High.

Preconditions
For this issue to occur, the following must hold true:

- Admin privileges must be required;
- A ratchet tree with holes must be present; and
- The target public keys must belong to current members.

Technical Details
In `remove_members`, the code iterates `mls_group.members()` and uses `enumerate()` to derive indices, then pushes `LeafNodeIndex::new(index as u32)`. This conflates the dense iteration position with the sparse ratchet tree's leaf index. Once the tree contains vacant leaves, these values diverge, and the incorrect leaf index is committed.

An admin can trigger this by creating at least one hole via a prior removal and then calling `remove_members` with a victim `pubkey`. The function translates the `pubkey` to the enumeration index and commits a removal for a different leaf. Repeating this allows unintended removals depending on tree sparsity and iteration order.

Remediation
We recommend replacing the use of `enumerate()` and `LeafNodeIndex::new(index as u32)` with logic that obtains the exact `LeafNodeIndex` for each member returned by `mls_group.members()`. For example, this can be done by reading the member's stored leaf index or by mapping the credential to the index via `member_at`.

Status
The Whitenoise team has resolved the issue by updating the member removal proposal process as recommended.

Verification
Resolved.

Issue R: Deprecated Hex Encoding for KeyPackage Content Enables Downgrade and Interop Failures

Location
`crates/mdk-core/src/key_packages.rs#L48`

`crates/mdk-core/src/groups.rs#L1265`

Synopsis
The functions `create_key_package_for_event` and `build_welcome_rumors_for_key_packages` emit hex-encoded content and omit the encoding tag, which contradicts MIP-00 and MIP-02 and enables downgrade and parsing ambiguity across clients.

Impact
Medium.

An attacker can strip or omit the encoding tag or craft ambiguous content so that different clients decode different bytes, which may cause state divergence, denied group joins, or acceptance of noncompliant artifacts.

Feasibility
High.

An attacker must only have the ability to publish a key package or `Welcome` event or strip the encoding tag in transit. No special privileges are required.

Severity
High.

Preconditions
For this issue to occur, the following must hold true:

- A client or relay must be able to submit a key package (`kind-443`) or `Welcome` (`kind-444`) event without the encoding tag;
- At least one validating peer must follow MIP-00 or MIP-02 formatting expectations; and
- Content that is valid in both hex and `base64`, or events where the encoding tag is omitted, must be present.

Technical Details
The function `create_key_package_for_event` selects encoding via `self.config.use_base64_encoding`. When `false` (the default), it uses `ContentEncoding::Hex` and does not add an encoding tag, while only `base64` adds `TagKind::Custom("encoding")` with value `"base64."` The function `build_welcome_rumors_for_key_packages` mirrors this behavior for `Welcome` events. The function `parse_key_package` derives the format using `ContentEncoding::from_tags`, which defaults to hex if the tag is absent, and then calls the function `parse_serialized_key_package`, which decodes with the function `decode_key_package_content` before TLS deserialization.

An adversary can craft a `kind-443` or `kind-444` event containing hex content and omit the encoding tag, or a relay can remove the tag, leading to rejection by strict clients or inconsistent decoding paths. Strings valid in multiple encodings exacerbate parsing ambiguity and lead to denial of service.

Remediation
We recommend replacing the `hex` default and implicit behavior with mandatory `base64` emission along with an explicit `encoding` tag, and using strict validation in the `create_key_package_for_event` and `build_welcome_rumors_for_key_packages` functions.

Status
The Whitenoise team has resolved the issue by updating the content encoding to `base64` and removing `hex` support and dual-format paths.

Verification
Resolved.

Issue S: Missing Validation of Mandatory Relays Tag in MLS KeyPackage Events

Location
`crates/mdk-core/src/key_packages.rs#L248`

Synopsis
The function `validate_key_package_tags` accepts `Kind::MlsKeyPackage` events without the mandatory `relays` tag, allowing empty or missing relay lists in violation of MIP-00.

Impact
Medium.

Clients may accept and process unroutable key packages, which may cause delivery failures and enable trivial denial-of-service against invitation or distribution workflows.

Feasibility
High.

Any publisher of a `Kind::MlsKeyPackage` event can omit or clear the tag relays without special privileges.

Severity
High.

Preconditions
For this issue to occur, the following must hold true:

- A consumer must use the function `parse_key_package` and trust the function `validate_key_package_tags`; and
- The system must rely on the `relays` tag for routing or discovery of key packages.

Technical Details
The function `validate_key_package_tags` requires `mls_protocol_version`, `mls_ciphersuite`, and `mls_extensions` but does not require or validate the tag `relays` (`TagKind::Relays`). As a result, events with a missing `relays` tag or with `Tag::relays(vec![])` pass validation and proceed to deserialization in the function `parse_key_package`.

An attacker can publish a `Kind::MlsKeyPackage` event containing valid `mls_protocol_version`, `mls_ciphersuite`, and `mls_extensions` while omitting `TagKind::Relays` or supplying an empty sequence. Consumers accept the key package and may attempt to use it even though no reachable `RelayUrl` values are available.

Remediation
We recommend replacing the current tag check in the function `validate_key_package_tags` and adding validation that requires a `TagKind::Relays` tag with at least one value that parses as a `RelayUrl`, rejecting empty or invalid entries.

Status
The Whitenoise team has resolved the issue by requiring a mandatory relay tag for key packages that includes at least one valid relay URL.

Verification
Resolved.

Issue T: Incomplete MIME Type Canonicalization in validate_mime_type

Location
crates/mdk-core/src/media_processing/validation.rs#L45

Synopsis
The function `validate_mime_type` performs incomplete canonicalization and returns MIME types with trailing parameters, contrary to MIP-04, which requires stripping parameters.

Impact
Medium.

An attacker or malformed client input may induce protocol-level mismatches and authentication failures where the canonical MIME type feeds associated data or key derivation, and benign parameters may cause denial of service via false mismatches.

Feasibility
High.

An attacker can set the `mime_type` parameter in user-controlled metadata.

Severity
High.

Preconditions
For this issue to occur, the following must hold true:

- The attacker must control the `mime_type` string;
- Downstream components must use the returned value by the `validate_mime_type` function in AAD, KDF input, or equality checks; and
- Peer implementations or other components must canonicalize per MIP-04 or otherwise strip parameters.

Technical Details
The function `validate_mime_type` only trims whitespace and lowercases input, then checks for the presence of "`/`" and a maximum length. It does not remove parameters after "`;`". Inputs such as "`image/png; charset=utf-8`" therefore validate and are returned unchanged aside from case and whitespace, which violates MIP-04 canonicalization semantics that require `type/subtype` without parameters.

When the function `validate_mime_type_matches_data` compares the returned `canonical_claimed` against the value from the function `detect_mime_type_from_data`, a parameterized claimed type such as "`image/png; charset=utf-8`" will not equal "`image/png`" and triggers `MediaProcessingError::MimeTypeMismatch`. An attacker can exploit this by supplying a parameterized type to cause preventable rejections or to desynchronize AAD or KDF inputs across components that follow MIP-04, producing authentication or decryption failures.

Remediation
We recommend replacing the normalization in the function `validate_mime_type` with canonicalization that discards any substring beginning with "`;`" and returns only `type/subtype`. In addition, we recommend comparing only this canonical form in `validate_mime_type_matches_data`, per MIP-04.

Status
The Whitenoise team has resolved the issue by adding the necessary canonicalization of `MIME` types.

Verification
Resolved.

Issue U: Deterministic Nonce Derivation Causes Nonce Reuse and Message Linkability

Location
`crates/mdk-core/src/encrypted_media/crypto.rs#L114`

Synopsis
The function `derive_encryption_nonce` deterministically derives a 96-bit nonce from the epoch-bound exporter secret and static file metadata, which repeats for the same media within an epoch and yields identical ciphertexts that a passive observer can link.

Impact
Medium.

A network observer can link repeat transmissions of the same media within the same group epoch by observing identical ciphertexts, enabling traffic analysis and disclosing content equality.

Feasibility
High.

No privileges are required beyond passive network visibility during an epoch.

Severity
High.

Preconditions
For this issue to occur, the following must hold true:

- A user must upload the same file more than once within a single epoch, such that the same `file_hash`, `mime_type`, and `filename` are provided to the `derive_encryption_nonce` function; and
- The adversary must be able to observe ciphertexts in transit.

Technical Details
The function `derive_encryption_nonce` uses HKDF with `SHA-256` over `exporter_secret.secret` and a static context `[SCHEME_LABEL || 0x00 || file_hash || 0x00 || mime_type || 0x00 || filename || 0x00 || "nonce"]`, producing a deterministic 96-bit nonce with no per-message randomness. Because the function `derive_encryption_key` derives the key from the same epoch-bound secret and the same metadata, the pair `(key, nonce)` repeats for identical media within a

single epoch, violating the nonce-uniqueness requirement of `ChaCha20-Poly1305` and causing ciphertexts and tags to repeat.

An observer can compare ciphertexts for equality to link repeated media within the same epoch.

Remediation

We recommend replacing `derive_encryption_nonce` with a unique, random 96-bit nonce from a cryptographically secure pseudorandom number generator per encryption and transmitting or storing it with the ciphertext. Alternatively, a per-message subkey can be derived using a random salt, and a monotonically increasing nonce can be computed under that subkey.

Status

The Whitenoise team has resolved the issue by using a 96-bit nonce per encryption as recommended.

Verification

Resolved.

Issue V: Hard-Coded Scheme Label in AAD Causes Version Mismatch and Decryption Denial of Service (DoS)

Location

`crates/mdk-core/src/encrypted_media/crypto.rs#L30`

`crates/mdk-core/src/encrypted_media/crypto.rs#L45`

`crates/mdk-core/src/encrypted_media/manager.rs#L188`

`crates/mdk-core/src/encrypted_media/manager.rs#L320`

Synopsis

A fixed `SCHEME_LABEL` value is embedded into AAD and HKDF contexts, so any label change in a future specification version would cause all decryptions to fail.

Impact

Medium.

A specification or client label change would render existing or new ciphertexts undecryptable across versions, denying access to media at scale.

Feasibility

High.

Any producer that encrypts with a different label, or any deployment mixing client versions, triggers authentication failure without special privileges.

Severity

High.

Preconditions

For this issue to occur, the following must hold true:

- A mixed-version deployment must exist; and

- Ciphertexts produced with a different `SCHEME_LABEL` must be present.

Technical Details
The constant `SCHEME_LABEL` is defined as `b"mip04-v1"` and is concatenated into AAD by `build_aad` and into HKDF-Expand contexts by `build_hkdf_context`. The functions `encrypt_data_with_aad` and `decrypt_data_with_aad` both depend on this constant, so a label change alters both the authenticated data and the derived key and nonce, causing `ChaCha20-Poly1305` verification to fail. An attacker or a newer client can encrypt media using a different label while keeping `file_hash`, `mime_type`, and `filename` consistent, producing ciphertext that the current `decrypt_data_with_aad` rejects with `EncryptedMediaError::DecryptionFailed`. If the constant changes during an upgrade, previously stored media becomes undecryptable, resulting in a systemic availability failure.

Remediation
We recommend replacing the hard-coded `SCHEME_LABEL` with a versioned scheme identifier that is serialized with the ciphertext and selected at runtime in `build_aad` and `build_hkdf_context`, with decryption dispatch that accepts legacy and new versions.

Status
The Whitenoise team has resolved the issue by removing support for legacy encryption scheme versions as recommended.

Verification
Resolved.

Issue W: MIME Type Spoofing and Missing Allowlist in Media Encryption

Location
`crates/mdk-core/src/encrypted_media/manager.rs#L66`

`crates/mdk-core/src/media_processing/validation.rs#L45`

Synopsis
The function `encrypt_for_upload_with_options` accepts a claimed `mime_type` without validating it against the file bytes and does not restrict allowed types, which permits spoofing and uploading unsupported or malicious files.

Impact
High.

An attacker may bypass file-type policy and cause downstream components to process or advertise incorrect content types, which may lead to content-type confusion and the distribution of malicious payloads.

Feasibility
Medium.

Only the ability to invoke the API with attacker-controlled `data`, `mime_type`, and `filename` is required.

Severity
High.

Preconditions

For this issue to occur, the following must hold true:

- The attacker must be able to call `encrypt_for_upload_with_options` or a wrapper with controlled inputs; and
- Downstream consumers or storage must rely on the recorded MIME type and lack independent type enforcement.

Technical Details

The function `encrypt_for_upload_with_options` calls `validation::validate_mime_type`, which only canonicalizes and checks the syntactic format of `mime_type`, then passes the value into `extract_and_process_metadata` and uses it in AAD for key and nonce derivation. It does not invoke `validation::validate_mime_type_matches_data` or any magic-number check to bind the claimed type to the actual file bytes. Additionally, there is no allowlist of supported types.

An attacker can submit any unsupported content while claiming `image/jpeg`, for example. The function would return a valid `EncryptedMediaUpload` and an `IMETA` tag carrying the spoofed type. If a viewer or client uses the recorded type for handling or serving after decryption, this may result in content-type confusion or propagation of malicious files beyond intended categories.

Remediation

We recommend replacing the call to `validation::validate_mime_type` with `validation::validate_mime_type_matches_data`, or a variant with an extended set of supported types, to bind type to bytes. In addition, a strict allowlist should be enforced before encryption, rejecting any `mime_type` not in the supported set.

Status

The Whitenoise team has resolved the issue by enforcing more a restricted `MIME` allowlist and byte-level image validation.

Verification

Resolved.

Issue X: Default LastResort KeyPackage Enables Multiple Use Against MLS Guidance

Location

`crates/mdk-core/src/key_packages.rs#L62`

Synopsis

The function `create_key_package_for_event` sets the `LastResort` extension by default, through `.mark_as_last_resort()`, which allows a single `KeyPackage` to be reused across multiple group inceptions, contrary to MLS guidance recommending publication of multiple single-use `KeyPackages`.

Impact

Medium.

An attacker or any sender may repeatedly use one published `KeyPackage` to add the target to multiple groups before any `Welcome` is processed. This can lead to unsolicited group joins, linkage across groups, and unnecessary resource consumption.

Feasibility
High.

Any party with read access to the published event content can construct `Welcomes` and reuse the `KeyPackage` until it is consumed. No special privileges are required.

Severity
High.

Preconditions
For this issue to occur, the following must hold true:

- The `create_key_package_for_event` function must call `.mark_as_last_resort()`;
- A `KeyPackage` with extension `0x000a` must be published and retrievable; and
- The recipient must not yet have processed a corresponding welcome event.

Technical Details
The builder `KeyPackage::builder()` in the function `create_key_package_for_event` invokes `.mark_as_last_resort()` before `.build(...)`, which sets the `LastResort` `KeyPackage` extension (`0x000a`). MLS permits such `KeyPackages` to be used multiple times until the recipient consumes a `Welcome`, but the standard recommends publishing multiple `KeyPackages` instead to avoid reuse and correlation (RFC 9750, key storage and retrieval).

An attacker can harvest the event content, reuse the same serialized `KeyPackage` to generate multiple group creations, and distribute several `Welcomes` before the target processes any of them. This can result in unwanted group enrollments, increases client state and bandwidth, and links the recipient across distinct groups through the identical `KeyPackage` artifact.

Remediation
We recommend removing `.mark_as_last_resort()` from the default path in the function `create_key_package_for_event` and instead using multiple single-use `KeyPackages` built via `KeyPackage::builder()` without the `LastResort` extension, with `LastResort` exposed as an explicit opt-in configuration when truly needed.

Status
The Whitenoise team has decided to retain the current behavior following extensive discussions between their internal team and our researchers.

Verification
Unresolved.

Issue Y: Missing Zeroization

Location
`crates/mdk-memory-storage/src/lib.rs#L55`

`crates/mdk-core/src/extension/group_image.rs`

`crates/mdk-core/src/encrypted_media/manager.rs#L30`

Synopsis
Various locations in the crates lack zeroization of secret data, which risks exposing those secrets.

Impact
High.

Unzeroized cached secrets risk lingering secret data after eviction, exposing encryption data.

Feasibility
Medium.

An attacker requires memory access or access to a memory dump resulting from a crashing process.

Severity
High.

Technical Details
In several locations across the crates, we identified missing zeroization of secrets:

1. In the `mdk-memory-storage` crate, none of the caches zeroize on eviction or drop, and the value types do not implement `Drop` or `Zeroize`. This includes:
    a. `group_exporter_secrets_cache` storing `GroupExporterSecret` with a [u8; 32] secret;
    b. `groups_cache` / `groups_by_nostr_id_cache` storing `Group`, which carries `image_key` and `image_nonce`; and
    c. `welcomes_cache` storing `Welcome`, which also carries `group_image_key` and `group_image_nonce`.
2. In the `mdk-core` crate, the `group_image` and `manager` modules contain several long-lived sensitive values in plain arrays within `Debug` and `Clone` structs and do not clear them from memory. This includes:
    a. Several structs within `group_image` containing `image_key`, `image_nonce`, and derived uploaded secrets in plain `[u8; N]`; and
    b. The `EncryptedMediaManager` module containing `encryption_key` and `nonce` in plain `[u8; N]` types.

Remediation
We recommend introducing zeroization in the `mdk-memory-storage` crate and the `mdk-core` crate.

Status
The White Noise team has resolved the issue by introducing a new `Secret` wrapper that uses the Zeroize trait and updating the sensitive fields to use the `Secret` wrapper.

Verification
Resolved.

Issue Z: Unbounded Function message Query Enables Memory Exhaustion Denial of Service

Location
mdk-sqlite-storage/src/groups.rs#L127

Synopsis

Unbounded history retrieval in the function `messages` loads all group messages into memory, enabling memory exhaustion and process crash.

Impact

Medium.

Successful exploitation can cause process termination or sustained resource starvation, degrading availability for all groups and jobs that read history.

Feasibility

Medium.

The attack requires a malicious or compromised group member (or device), and any client or job invoking the function `messages`.

Severity

Medium.

Preconditions

For this issue to occur, the following must hold true:

- Attacker membership in or write access to the target group must be present; and
- The function `messages` must be callable by clients or background jobs that load history.

Technical Details

The function `messages` verifies group existence, then executes `SELECT * FROM messages WHERE mls_group_id = ? ORDER BY created_at DESC` and pushes each deserialized `Message` into `Vec<Message>`. This materializes the full result set for the supplied `mls_group_id` with no pagination or cap. Memory usage scales linearly with the number of rows.

A malicious member can flood the group with many valid messages to expand the table by millions of rows. Any subsequent call to the functions `messages` or `get_messages` would attempt to load and deserialize the entire history, increasing the likelihood of out-of-memory termination and prolonged stalls.

This is a concrete manifestation of the resource-exhaustion risks noted in the threat model (T.3.1 Message Spam and T.10.4 Large Group Resource Exhaustion).

Remediation

We recommend replacing the unbounded selection in the function `messages` with mandatory pagination and an enforced server-side `LIMIT`, and using keyset pagination on `created_at` or a streaming cursor that yields bounded batches.

Status

The White Noise team has resolved the issue as recommended.

Verification

Resolved.

Issue AA: Unbounded Loading of Pending Welcomes Enables Memory-Exhaustion Denial of Service

Location
mdk-sqlite-storage/src/welcomes.rs#L79

mdk-sqlite-storage/src/groups.rs#L25

mdk-sqlite-storage/src/groups.rs#L127

mdk-sqlite-storage/src/groups.rs#L167

Synopsis
The function `pending_welcomes` queries all rows with `state` equal to "pending" without limits and deserializes them, which allows unbounded processing that may cause memory exhaustion or stalls.

Impact
Medium.

An attacker can deny service by exhausting heap memory and CPU, affecting availability of the process and dependent components.

Feasibility
Medium.

An attacker who can create many pending entries and trigger the code path that calls the function `pending_welcomes` can perform the attack over standard application interfaces.

Severity
Medium.

Preconditions
For this issue to occur, the following must apply:

- The attacker must be able to create or cause creation of many rows with the column `state` set to ”`pending`”;
- A request handler, job, or endpoint must call the function `pending_welcomes()`; and
- The database and service configuration must not impose quotas, per-actor limits, or row-level time-to-live that prevent unbounded growth.

Technical Details
The function `pending_welcomes` executes a query equivalent to `SELECT * FROM welcomes WHERE state = 'pending'`, materializes the full result set in memory, and performs JSON deserialization and `RelayUrl` parsing for each row. Processing scales with the total number of pending rows, which drives unbounded heap allocation and CPU use, degrading throughput and potentially terminating the process. Similar unbounded reads exist in the functions `all_groups`, `messages`, and `group_relays` in the file `groups.rs`, although inflating those tables is more difficult.

An exploit consists of creating a large number of pending entries, then invoking any operation that calls the function `pending_welcomes` to trigger repeated full deserialization. If the interface is unauthenticated or low cost, automated requests can sustain a denial of service.

Remediation

We recommend replacing the unbounded query in the function `pending_welcomes()` with paginated access using `ORDER BY` and `LIMIT`, processing rows in fixed-size batches. We also suggest applying the same bounded pattern in the functions `all_groups`, `messages`, and `group_relays`.

Status

The White Noise team has resolved the issue as recommended.

Verification

Resolved.

Issue AB: Unbounded User Input Stored in SQLite Causes Disk and CPU Exhaustion

Location

`mdk-sqlite-storage/src/messages.rs#L20`

`mdk-sqlite-storage/src/welcomes.rs#L21`

`mdk-sqlite-storage/src/groups.rs#L77`

Synopsis

The storage layer accepts unbounded user-controlled fields in the functions `save_message`, `save_welcome`, and `save_group`, which allows oversized or numerous inserts that expand the SQLite database and increase JSON processing, degrading availability.

Impact

Medium.

An attacker can exhaust disk space, causing write failures and outages, and trigger CPU or memory spikes during JSON (de)serialization that degrade service for all users.

Feasibility

Medium.

An attacker who can invoke these functions can submit oversized strings repeatedly without special privileges beyond write access to a group.

Severity

Medium.

Preconditions

No application-level length validation should be in place.

Technical Details

The function `save_message` persists the variable `content`, the variable `tags` as JSON, and the variable `event` as JSON using an `INSERT OR REPLACE` into `messages` without size bounds. The function `save_welcome` persists event JSON and group metadata, and the function `save_group` persists the variable `name` and the variable `description`, all without length limits. The codec `JsonCodec` uses `serde_json::to_vec` and `serde_json::from_slice`, which operate on whole buffers and amplify

peak memory during large payload processing. The `mdk-core` crate does not impose size bounds before calling the storage trait functions.

Repeated inserts with large `content` or large JSON fields grow the SQLite file, which does not shrink without explicit vacuuming. Once the volume fills, SQLite returns write errors and requests fail. Even before exhaustion, repeated serialization and parsing of large JSON saturates CPU and memory and yields a practical denial of service.

Remediation

We recommend adding strict byte-length caps for `content`, `tags`, `event`, `group_name`, and `group_description` before calling `save_message`, `save_welcome`, and `save_group`, rejecting any inputs that exceed the configured limits.

Status

The White Noise team has resolved the issue as recommended.

Verification

Resolved.

Issue AC: nostr_group_id Cache Collision Leaves Stale Keys and Enables Lookup Hijack

Location

`mdk-memory-storage/src/groups.rs#L15`

Synopsis

Non-unique caching of groups by `nostr_group_id` permits overwrites and stale mappings, which causes misrouting and denial of service.

Impact

Medium.

An attacker who can create or update a group can set `nostr_group_id` to collide with a victim, hijacking lookups and causing operations to target attacker-controlled or stale groups.

Feasibility

Medium.

If clients can choose `nostr_group_id` or trigger saves, collisions are trivial to produce using only standard group-creation privileges.

Severity

Medium.

Preconditions

For this issue to occur, the following must hold:

- The application must use `find_group_by_nostr_group_id` to route or authorize operations;
- The attacker must have the ability to create or update groups with chosen `nostr_group_id` values; and
- No higher-layer uniqueness enforcement or cleanup on rename should be present.

Technical Details

The function `save_group` writes each group into `groups_cache` keyed by `mls_group_id` and into `groups_by_nostr_id_cache` keyed by `nostr_group_id`. It calls `cache.put` without uniqueness checks or removal of prior `nostr_group_id` keys when an identifier changes. This produces last-write-wins overwrites and stale entries. There are also no guards in the `mdk-core` crate that prevent this issue when `mdk-core` is run with the memory storage.

A colliding write to `groups_by_nostr_id_cache` replaces an existing mapping, so `find_group_by_nostr_group_id` resolves to the most recently saved group. After a rename, the old key persists, so lookups by the old identifier return an incorrect or attacker-controlled state.

Remediation

We recommend replacing the unguarded `cache.put` in `save_group` with a conflict-checked update that rejects saves when the variable `nostr_group_id` already maps to a different `mls_group_id`. We also suggest adopting an atomic update that removes any previous `nostr_group_id` key for the same group before inserting the new one.

Status

The White Noise team has resolved the issue as recommended.

Verification

Resolved.

Issue AD: Premature Welcome Delivery Allows Joiner Entry Into Unseen Epoch (State Fork)

Location

`mdk-core/src/welcomes.rs#L56`

`mdk-core/src/welcomes.rs#L167`

Synopsis

The join flow activates a group from a `Welcome` without validating that the Commit creating the referenced epoch was observed by incumbents, allowing a relay-delivered `Welcome` to move the joiner into an epoch no other member occupies and causing a state fork with message failures.

Impact

Medium.

An attacker operating or influencing a relay can cause the joiner to activate on an unseen epoch, resulting in a persistent partition where messages fail to decrypt and state diverges until manual recovery. Even without an active attacker, relay reordering can cause the same negative effect.

Feasibility

Medium.

Control of, or influence over, the joiner's relay timing is sufficient. No group secrets or privileged credentials are required.

Severity

Medium.

Preconditions

The `Welcome` must be delivered to the joiner before the Commit is observed or merged.

Technical Details

The function `process_welcome` parses the `Welcome` into the object `staged_welcome`, persists a group in the state `Pending` using `staged_welcome.group_context().epoch()`, and does not record a Commit identifier. The function `accept_welcome` re-parses the `Welcome`, calls the method `staged_welcome.into_group`, marks the `Welcome` as `Accepted`, and flips the stored group to the state `Active` without consulting storage for a causally preceding Commit message. No lookup ties activation to a processed Commit for the same group identifier and epoch.

A malicious or misconfigured relay can forward the `Welcome` while withholding the commit that references the joiner. The joiner activates on a private epoch, after which application messages and commits from incumbents fail decryption or validation due to epoch mismatch and exporter secret divergence.

This constitutes a deviation from the documentation, as MIP-02's "Timing Requirements" section explicitly states "*Don't send the Welcome until relays confirm receipt of the Commit*" and "*Ensure the group state change is committed before inviting*." This sequencing is intended to prevent `Welcomes` from being sent for a group state that has not yet been finalized.

Remediation

We recommend augmenting `Welcome` events with a Commit identifier tag, for example the inviter's `commit_event.id`, and persisting it during `process_welcome`. We also recommend replacing the unconditional activation path in the function `accept_welcome` with a guard that retrieves a `message_types::ProcessedMessage` for that Commit and verifies the group identifier and epoch.

Status

The White Noise team has provided a convincing rationale that, while the issue is present, it should be addressed on the Nostr relay server side. To record this approach, the team has updated the documentation accordingly. However, since the Nostr relay repository is out of scope for this audit, our team will review the implementation of the remediation during the next audit.

Verification

Resolved.

Issue AE: Missing Nostr-Based Validations When Processing Messages

Location

`crates/mdk-core/src/messages.rs#L531`

Synopsis

The function `validate_event_and_extract_group_id` accepts wrapper events without verifying the Nostr signature, recomputing `event.id`, checking `event.created_at` bounds, or enforcing the MIP-03 requirement of exactly one h tag.

Impact

Medium.

An attacker or malicious relay may inject unsigned or tampered wrapper events to misroute messages via manipulated h tags, create duplicate or replayable entries by altering `event.id`, and degrade availability through abnormal `event.created_at` values.

Feasibility
Medium.

A remote adversary or untrusted relay with no privileges can supply crafted `kind 445` events to the caller `process_message`.

Severity
Medium.

Technical Details
The function `validate_event_and_extract_group_id` only verifies `event.kind == Kind::MlsGroupMessage` and extracts the first `TagKind::h()` value to a `[u8;32]`. It does not call `event.verify()` to authenticate the wrapper, does not recompute or validate `event.id` from `kind`, `pubkey`, `tags`, `content`, and does not check `event.created_at` for skew or future timestamps. It also does not enforce MIP-03, which requires exactly one h tag carrying the group identifier. The current code accepts zero as an error but allows more than one by taking the first. The test `test_message_event_validation` demonstrates signature validation and timestamp checks that are currently absent from this code path.

Remediation
We recommend validating the wrapper by calling `event.verify()` and rejecting on failure. We also recommend checking `event.created_at` against a configurable skew window, recomputing and comparing `event.id` from `event.kind`, `event.pubkey`, `event.tags`, and `event.content`, and enforcing that `event.tags` contains exactly one `TagKind::h()` with a 32-byte hex group identifier.

Status
The White Noise team has resolved the issue by adding sanity checks on the event. The signature check has not been added because this occurs upstream (see also Issue E).

Verification
Resolved.

Issue AF: Inner Nostr Kind Not Validated in MLS Chat Messages

Location
`crates/mdk-core/src/messages.rs#L161`

`crates/mdk-core/src/messages.rs#L299`

Synopsis
The message creation and receive paths accept and store arbitrary inner Nostr kinds rather than restricting them to the documented unsigned text `kind 9`, which leads to out-of-spec chat messages.

Impact
Medium.

Downstream components may accept, store, and render out-of-spec content as normal chat messages, which can cause content-type confusion, incorrect filtering, and interoperability failures while also updating group message metadata.

Feasibility
Medium.

Any authenticated group member can submit an MLS application message with a non-9 inner `UnsignedEvent.kind`, or any caller of the function `create_message` can persist such events locally.

Severity
Medium.

Preconditions
For this issue to occur, the following must hold true:

- Membership in the target MLS group must be required; and
- A receiver must process messages and rely on `Message.kind == 9` as an invariant.

Technical Details
The function `create_message` persists the variable `rumor.kind` without modification when constructing `message_types::Message` and does not validate that the inner event is of type `kind 9`. The function `process_application_message_for_group` deserializes with `UnsignedEvent::from_json`, then stores `rumor.kind` unchanged and updates the variables `last_message_at` and `last_message_id`, also without validating or coercing the inner shape specified by the documentation.

Remediation
We recommend replacing the direct persistence of arbitrary `rumor.kind` in the function `create_message` and the function `process_application_message_for_group` with validation that the inner event matches the chat schema with `kind == 9`, rejecting all other kinds.

Status
The White Noise team has argued convincingly that this is not an issue.

Verification
Determined Non-Issue.

Issue AG: Function all_groups Aborts on First Deserialization Error

Location
`crates/mdk-sqlite-storage/src/groups.rs#L25`

Synopsis
The function `all_groups` returns an error on the first failed row deserialization from `db::row_to_group`, which blocks retrieval of otherwise valid groups.

Impact
Medium.

A single malformed or corrupted row may prevent listing all groups, which degrades availability for callers that rely on full enumeration.

Feasibility
Medium.

Insertion of an invalid row or schema drift is required, and direct database write privileges or prior faulty writes are necessary.

Severity
Medium.

Technical Details
The function `all_groups` iterates `groups_iter` and applies `group_result.map_err(into_group_err)?`, which propagates the first `Err` from `db::row_to_group` and aborts iteration. The `TODO` comment in the function `all_groups` already notes that errors should be skipped or logged rather than blocking the entire request.

Remediation
We recommend replacing the early return on `group_result` errors in the function `all_groups` with best-effort iteration that logs and skips rows that fail `db::row_to_group`, collecting and returning only successfully parsed groups.

Status
The White Noise team has resolved the issue as recommended

Verification
Resolved.

Issue AH: Initial Commit Not Published as Kind:445 on Group Creation Before Welcome

Location
`crates/mdk-core/src/groups.rs#L848`

Synopsis
The function `create_group` applies the initial membership change locally but does not publish or return the corresponding `kind:445` Commit event, breaking required relay ordering relative to `Welcomes`.

Impact
Low.

Group participants may miss the authoritative initial epoch change on relays, which can desynchronize state and allow denial of service through unorderable or replayable `Welcomes`.

Feasibility
High.

The behavior is triggered on every group creation, and only creator privileges are required.

Severity
Medium.

Preconditions
For this issue to occur, the following must hold true:

- A group must be created with initial members;
- Clients must expect a `kind:445` Commit event per MIP-01, MIP-02, and MIP-03 before `Welcomes`; and
- The function `create_group` must be used to set up the group.

Technical Details
The function `create_group` calls `mls_group.add_members(&self.provider, &signer, &key_packages_vec)` and destructures the result as `(_, welcome_out, _group_info)`, discarding the Commit. It then immediately invokes `mls_group.merge_pending_commit(&self.provider)` and returns `GroupResult { group, welcome_rumors }` without building a Commit `evolution_event` via `build_encrypted_message_event`. This deviates from MIP-01 "Group State Changes" and MIP-03 "Commit Messages," which require publishing the Commit to relays, and from MIP-02 ordering, which requires the Commit to precede `Welcomes`.

An attacker with creator privileges can create a group and cause peers to receive `Welcomes` without any preceding Commit on relays, preventing clients from ordering or validating the initial epoch transition. This gap permits state divergence between members and enables denial of service when subsequent commits reference a missing base state.

Remediation
We recommend replacing the immediate merge in the function `create_group` and instead using the `commit_message` returned by `mls_group.add_members` to build and return a `kind:445` `evolution_event`. In addition, the Commit should be published and `merge_pending_commit` should be deferred until relay confirmation, per MIP-02.

Status
The Whitenoise team has updated the documentation and comments to clarify the intended behavior.

Verification
Determined Non-Issue.

Issue AI: Update API Omits nostr_group_id Update Allowed by Specification

Location
`crates/mdk-core/src/groups.rs#L111`

Synopsis
The group update API exposes several metadata fields but omits `nostr_group_id`, despite the extension and MIP-01 allowing rotation via proposals, which leads to divergence in group routing and discovery.

Impact
Medium.

The omission degrades availability and routing integrity under the project threat model, causing message delivery loss and discovery inconsistencies when identifiers rotate. However, it does not affect confidentiality.

Feasibility
Medium.

The attacker must hold admin privileges to introduce an extension update, while affected users only need to rely on this module's update path.

Severity
Medium.

Preconditions
For this issue to occur, the following must hold true:

- An administrator must be present;
- A group that follows MIP-01, where `nostr_group_id` may be updated, must be in scope; and
- The client must use the function `update_group_data` to originate extension changes.

Technical Details
The struct `NostrGroupDataUpdate` exposes `name`, `description`, `image_hash`, `image_key`, `image_nonce`, relays, and admins but has no field for `nostr_group_id`. The function `update_group_data` applies only these fields to `group_data` and then calls `update_group_data_extension`, so there is no code path to set `nostr_group_id` on the extension, although MIP-01 permits updating it. The function `sync_group_metadata_from_mls` copies `nostr_group_id` from the extension into storage, and `build_encrypted_message_event` tags outgoing messages with `nostr_group_id`, so a rotated identifier affects routing and discovery.

An administrator using another implementation can rotate `nostr_group_id`, partitioning clients until they process and merge the Commit, while this module cannot originate a planned rotation or corrective update. Outgoing events from an unsynchronized or misconfigured client will carry the stale tag in `build_encrypted_message_event`, fragmenting the group's message flow.

Remediation
We recommend extending the struct `NostrGroupDataUpdate` with an optional `nostr_group_id` field, applying it in `update_group_data` to set `group_data.nostr_group_id`, and committing via `update_group_data_extension`.

Status
The Whitenoise team has resolved the issue as recommended.

Verification
Resolved.

Issue AJ: Creator-Only Group Creation Rejected

Location
crates/mdk-core/src/groups.rs#L848

Synopsis
The function `create_group` rejects creator-only groups by requiring at least one initial member, which diverges from MIP-01 and blocks interoperable group creation.

Impact
Medium.

This behavior may prevent creation of valid creator-only groups, causing interoperability failures and denial of expected functionality across compliant clients.

Feasibility
Medium.

A caller with standard application privileges must invoke the function `create_group` with an empty `member_key_package_events` while supplying a valid `config.admins` that contains the creator key.

Severity
Medium.

Preconditions
For this issue to occur, the following must hold true:

- The `create_group` function must be called by a legitimate client;
- The `member_key_package_events` variable must be empty; and
- The `config.admins` variable must include the creator public key.

Technical Details
The function `create_group` constructs an `MlsGroup` for the creator, then builds `key_packages_vec` from `member_key_package_events` and calls `add_members`. It then calls `build_welcome_rumors_for_key_packages` and converts its None return into an error via `.ok_or(Error::Welcome("Error creating welcome rumors"))`, which rejects creator-only groups because an empty input yields None for `welcome_rumors`.

A compliant client that omits initial members, as permitted by MIP-01, triggers a deterministic failure when calling `create_group`. An actor that can cause the caller to pass an empty initial membership can induce denial of expected functionality without additional privileges.

Remediation
We recommend replacing the `.ok_or(...)` on the result of `build_welcome_rumors_for_key_packages` with logic in `create_group` that accepts an empty `member_key_package_events`, skips `add_members`, and returns `welcome_rumors: None` while persisting a creator-only group.

Status
The Whitenoise team has resolved the issue by enabling creator-only groups.

Verification
Resolved.

Issue AK: Removed Member Commit Processing Breaks Due to Missing Group State Sync and Unconditional Exporter Secret Export

Location
crates/mdk-core/src/groups.rs#L1150

crates/mdk-core/src/groups.rs#L34

crates/mdk-storage-traits/src/groups/types.rs#L15

crates/mdk-core/src/messages.rs#L496

Synopsis
A member removal Commit can become unprocessable for the removed member because group state is not updated to reflect removal and Commit handling always exports and stores a new exporter secret.

Impact
Medium.

A removed member's client can fail to process the removal Commit and can retain stale group metadata and state, which can cause persistent denial of service for message processing and incorrect local group lifecycle handling.

Feasibility
Medium.

An administrator can trigger the condition by removing a member and having the removed member process the resulting Commit, which requires admin privileges but can occur in routine operation.

Severity
Medium.

Preconditions
For this issue to occur, the following must hold true:

- A group administrator must remove a member and publish the corresponding MLS Commit;
- The removed member must receive and process the Commit event via the `process_message` function; and
- The removed member's local storage must still contain the MLS group and the stored `group_types::Group` record.

Technical Details
The function `sync_group_metadata_from_mls` updates fields such as epoch, extension-derived metadata, and relays, but it does not update the stored `group_types::Group.state`, even though the data flow describes a removed member deleting group state. In addition, `process_commit_message_for_group` calls `exporter_secret` unconditionally after `merge_staged_commit`, and `build_encrypted_message_event` always calls `exporter_secret` during Commit and proposal event construction.

If a Commit removes the local client from the MLS tree, OpenMLS commonly transitions the group into an evicted state or removes access to member-specific state. Under this condition, the post-merge call chain from `process_commit_message_for_group` to `exporter_secret` can fail (for example, due to

use-after-eviction semantics), which prevents the removed member from completing Commit processing and reaching any intended cleanup path. In addition, it leaves `group_types::Group.state` potentially set to `Active`.

Remediation

We recommend updating the removal and Commit processing paths to set `group_types::Group.state` to `Inactive` (or delete the stored group record), when the local member is removed. In addition, `exporter_secret` export and storage should be conditionally skipped when the local member is evicted or lacks an `own_leaf` in the merged group.

Status

The Whitenoise team has resolved the issue as recommended.

Verification

Resolved.

Issue AL: Exporter Secrets Saved and Retrieved for Nonexistent Group

Location

`mdk-memory-storage/src/groups.rs#L138`

`mdk-memory-storage/src/groups.rs#L125`

Synopsis

The functions `get_group_exporter_secret` and `save_group_exporter_secret` discard the inner `Option` from the function `find_group_by_mls_group_id`, which permits reading or writing exporter secrets for MLS group identifiers that do not exist.

Impact

Low.

An attacker or faulty caller could associate exporter secrets with arbitrary group identifiers and later retrieve them, which may corrupt invariants across components and cause logic errors without an active attacker. The risk is limited to users of the storage trait and is not exploitable in the `mdk-core` crate.

Feasibility

Medium.

A caller with access to the storage trait can pass any MLS group identifier and trigger the behavior without timing constraints or access to additional secrets.

Severity

Low.

Preconditions

Access to the storage trait `GroupStorage` must be available.

Technical Details

The function `find_group_by_mls_group_id` returns `Result<Option<Group>, GroupError>`. The functions `get_group_exporter_secret` and `save_group_exporter_secret` call

`find_group_by_mls_group_id` and use ? only on the outer `Result`, ignoring the `Option`, so `Ok(None)` allows continuation and subsequent cache access.

This behavior allows inserting entries into the variable `group_exporter_secrets_cache` with keys `(GroupId, epoch)` where the corresponding group does not exist in the variable `groups_cache`. The invariant that only existing groups have exporter secrets is therefore not maintained by the memory backend.

Remediation

We recommend replacing the bare propagation with explicit handling of the `Option` in the functions `get_group_exporter_secret` and `save_group_exporter_secret`.

Status

The White Noise team has resolved the issue as recommended.

Verification

Resolved.

Issue AM: Missing Input Validation Enables Memory Exhaustion

Location

mdk-memory-storage/src/lib.rs#L55

mdk-memory-storage/src/groups.rs#L95

mdk-memory-storage/src/messages.rs#L11

mdk-memory-storage/src/groups.rs#L15

mdk-memory-storage/src/welcomes.rs#L11

Synopsis

The in-memory cache limits only key count, while write paths store unbounded per-key values, allowing a single hot key to consume excessive memory and evict unrelated entries.

Impact

Medium.

A successful attack allows a remote actor to cause eviction of unrelated cached data and drive the process out of memory, reducing availability and stability.

Feasibility

Low.

An attacker who can submit relays, messages, group metadata, or `Welcome` payloads that reach these functions can grow a single entry arbitrarily.

Severity

Low.

Preconditions

Untrusted inputs must reach `replace_group_relays`, `save_message`, `save_group`, or `save_welcome` without size validation.

Technical Details

The type `MdkMemoryStorage` defines multiple `LruCache` instances that bound only entry count via `DEFAULT_CACHE_SIZE`, while the value types are unbounded collections or structs. The cache `group_relays_cache` stores a `BTreeSet<GroupRelay>` populated by the function `replace_group_relays`. The cache `messages_by_group_cache` grows a `Vec<Message>` via the function `save_message`. Similarly, the caches `groups_cache` and `groups_by_nostr_id_cache` store `Group`, and `welcomes_cache` stores `Welcome`.

An attacker can expand a single cache value by submitting many `RelayUrl` items, flooding messages into one group or crafting oversized fields in `Group` or `Welcome`. Continued writes or reads keep the key hot, so the LRU evicts unrelated entries while the process memory grows toward exhaustion.

These unbounded caches convert the "Resource Exhaustion" and denial of service (DoS) risks identified in Section 2.10 of the threat model (for example, T.10.1, T.10.2, and T.10.4) into concrete, code-level avenues. A single malicious input can monopolize memory because entry-count LRUs do not cap value size. They also conflict with the "size limits" and "high-volume handling" expectations in Section 3.5.4, demonstrating missing safeguards where the threat model assumes client-side bounds.

Remediation

We recommend replacing unbounded values with bounded representations and using strict per-type caps enforced at input boundaries in `replace_group_relays`, `save_message`, `save_group`, and `save_welcome`. Examples include a fixed-capacity ring buffer per group, a maximum relay count and URL length, and bounded string and set sizes before `cache.put`.

Status

The White Noise team has resolved the issue as recommended.

Verification

Resolved.

Issue AN: Unvalidated Group ID Allows LRU Cache Pollution in save_message

Location

`mdk-memory-storage/src/lib.rs#L71`

`mdk-memory-storage/src/messages.rs#L11`

Synopsis

The memory storage layer accepts messages and indexes them per group without verifying that the referenced group exists, which allows insertion of arbitrary group keys into the cache `messages_by_group_cache`.

Impact

Medium.

Successful exploitation can evict legitimate groups from `messages_by_group_cache`, which may cause real groups to temporarily return empty message lists and degrade availability even without data corruption.

Feasibility
Low.

Severity
Low.

Technical Details
The function `save_message` writes the message into the cache `messages_cache` and then updates the cache `messages_by_group_cache`. If `group_cache.get_mut(&message.mls_group_id)` returns None, the function inserts a new entry via `group_cache.put(message.mls_group_id.clone(), vec![message])` without verifying group existence through the function `find_group_by_mls_group_id` or the cache `groups_cache`. This creates entries for nonexistent groups.

Remediation
We recommend replacing the unconditional insertion in the None branch of the function `save_message` with a lookup via the function `find_group_by_mls_group_id` or the cache `groups_cache`, and returning `MessageError` when no group exists.

Status
The White Noise team has resolved the issue as recommended.

Verification
Resolved.

Issue AO: MLS Group ID Leakage in Error Messages

Location
Examples (non-exhaustive):

`mdk-sqlite-storage/src/groups.rs#L244`

`mdk-memory-storage/src/groups.rs#L58`

Synopsis
Error strings for SQLite and memory storage traits include the private Messaging Layer Security (MLS) group identifier, which may be recorded by logs or telemetry contrary to MIP-01 and the threat model.

Impact
Medium.

An attacker or operator can exfiltrate private MLS group identifiers from logs, allowing cross-system linkage of groups and weakening metadata privacy guarantees. Even without an active attacker, routine logging may leak the identifier to remote analytics or crash reporting backends.

Feasibility
Low.

Any caller that triggers a not-found path or an invalid-parameters path causes emission of these strings, and common deployments collect such errors automatically.

Severity
Low.

Preconditions
The attacker requires access to the logging records.

Technical Details
The SQLite storage trait uses `format!` with `{:?}` on the parameter `mls_group_id` within the functions `messages`, `admins`, `group_relays`, `replace_group_relays`, `get_group_exporter_secret`, and `save_group_exporter_secret`, returning errors such as `GroupError::InvalidParameters("Group with MLS ID {:?} not found")`. The memory storage trait mirrors this pattern in the functions `messages`, `group_relays`, `replace_group_relays`, `get_group_exporter_secret`, and `save_group_exporter_secret`, exporting the 32-byte MLS group identifier beyond the cryptographic trust boundary and violating MIP-01 group identity and privacy guidance.

An attacker can induce lookups on nonexistent groups or malformed inputs so that these functions return the formatted errors, after which log collectors or relay-side diagnostics would capture the string containing the identifier. Operators who access logs could correlate `mls_group_id` across systems and track activity.

Remediation
We recommend replacing the formatted errors with non-identifying messages (for example, "`group not found`") or structured error variants that do not embed the MLS group ID, across both SQLite and memory backends.

Status
The White Noise team has resolved the issue as recommended.

Verification
Resolved.

Issue AP: Early Validation Failures Omit Failed ProcessedMessage State

Location
`crates/mdk-core/src/messages.rs#L969`

Synopsis
Early validation and decryption failures return errors without persisting a failed processing record, allowing repeated expensive reprocessing of the same invalid event.

Impact
Low.

An attacker may repeatedly replay malformed events to trigger full validation and multi-epoch decryption attempts, increasing CPU and storage access and degrading availability.

Feasibility
Medium.

An unauthenticated actor that can deliver Nostr events via a relay can trigger these paths. Knowledge of a target group's `nostr_group_id` increases the cost by reaching decryption.

Severity
Low.

Preconditions
For this issue to occur, the following must hold true:

- The attacker must be able to deliver crafted Nostr events to the target client via a relay; and
- The attacker must have knowledge of the target group's h tag value (`nostr_group_id`) to reach decryption paths.

Technical Details
The function `process_message` calls `validate_event_and_extract_group_id`, `load_group_and_decrypt_message`, and then `process_decrypted_message`. On early failures such as unexpected `Kind`, missing h tag, `GroupNotFound`, or failure from `try_decrypt_with_recent_epochs`, it returns `Err(Error::...)` without writing a `message_types::ProcessedMessage` with `ProcessedMessageState::Failed` keyed by `event.id`. In contrast, errors from `process_decrypted_message` are handled in `handle_message_processing_error`, which persists a failed `ProcessedMessage` and returns `Unprocessable`.

Remediation
We recommend persisting a `message_types::ProcessedMessage` with `state` `ProcessedMessageState::Failed` and a `failure_reason`, keyed by `event.id`, on early-return paths in `process_message`, `validate_event_and_extract_group_id`, and `load_group_and_decrypt_message`. We also recommend short-circuiting future attempts by checking storage via `find_processed_message_by_event_id`.

Status
The White Noise team has resolved the issue as recommended.

Verification
Resolved.

Issue AQ: sync_group_metadata_from_mls Silently Ignores Mandatory Group-Data Extension Parse Failure

Location
`crates/mdk-core/src/groups.rs#L1159`

Synopsis
The function `sync_group_metadata_from_mls` silently ignores failure to parse the mandatory `marmot_group_data` extension while still advancing and persisting state, in violation of MIP-01.

Impact
Low.

An attacker may cause stored group metadata and relay configuration to diverge from the MLS state while the system accepts and stores a new epoch, weakening integrity and downgrade resilience.

Feasibility
Medium.

Exploitation requires a group member whose commit or context-extension update is merged, with the extension missing, malformed, or using an unsupported version.

Severity
Low.

Preconditions
For this issue to occur, the following must hold true:

- An MLS group must exist, and `marmot_group_data` must be mandatory per MIP-01;
- The incoming MLS state must have the extension missing, invalid, or at an unsupported version; and
- The `sync_group_metadata_from_mls` function must run after a state change or Commit merge.

Technical Details
In `sync_group_metadata_from_mls`, the variable `stored_group.epoch` is updated unconditionally, and `save_group` is called regardless of extension validity. The function parses the extension via `NostrGroupDataExtension::from_group(&mls_group)` with an `if let Ok(..)`, so parse failures do not propagate and only skip metadata and relay updates, which contradicts MIP-01's requirement to validate and treat invalid or missing mandatory data as an error.

A malicious member can propose a group-context extension and have it merged even if its bytes fail `NostrGroupDataExtension` parsing, for example due to an unsupported version. After the merge, peers would call `sync_group_metadata_from_mls`, which would advance the epoch and persist the group while silently discarding the invalid extension, creating persistent metadata desynchronization and masking downgrade attempts.

Remediation
We recommend replacing the conditional `if let Ok(..)` in `sync_group_metadata_from_mls` with explicit validation that returns an error on missing, invalid, or unsupported `marmot_group_data`. In addition, we recommend refraining from updating `stored_group` or calling `save_group` when validation fails.

Status
The Whitenoise team has resolved the issue as recommended.

Verification
Resolved.

Suggestions

Suggestion 1: Update Error Message in mdk-memory-storage

Location
`crates/mdk-memory-storage/src/groups.rs#L74`

Synopsis
The function `admins()` in the aforementioned location returns the error message `NoAdmins` even though the error condition is that no group exists.

Mitigation
We recommend updating the error message type.

Status
The White Noise team has implemented the mitigation as recommended.

Verification
Resolved.

Suggestion 2: Propagate last_message_id Parse Errors in row_to_group

Location
`crates/mdk-sqlite-storage/src/db.rs#L130`

Synopsis
The function `row_to_group` retrieves the variable `last_message_id` using `row.get_ref("last_message_id")?.as_blob_or_null()?` and then maps it to `Option<EventId>` by calling `EventId::from_slice(id).ok()` within `and_then`. Any parsing failure, including incorrect length, results in None rather than an error, which masks invalid state. However, because `last_message_id` is used only as metadata, it does not appear exploitable.

By comparison, other parsing paths in the same module convert invalid blobs to errors via `map_err`. For example, when parsing message or wrapper `EventId` values, failures propagate as expected. This asymmetric handling indicates that only `last_message_id` can silently lose integrity, which may remove ordering and deduplication bounds.

Mitigation
We recommend replacing the `.ok()` coercion in the function `row_to_group` with explicit error propagation or a distinct `Invalid` state for non-null blobs that fail `EventId::from_slice`.

Status
The White Noise team has implemented the mitigation as recommended.

Verification
Resolved.

Suggestion 3: Implement KeyPackage Deletion After Welcome Acceptance

Location
`crates/mdk-core/src/welcomes.rs#L167`

Synopsis
The function `accept_welcome` activates the group but does not delete the consumed KeyPackage or request relay deletion if there is no `last_resort` extension, leaving the invitation flow's final step unimplemented (see MIP-02 here and the threat model here). Since it is not possible to create a KeyPackage without the `last_resort` extension (Issue X), the missing deletion path is not reachable and has no impact.

Mitigation
We recommend implementing the deletion path as described in the documentation.

Status
The White Noise team has argued convincingly that the proposed mitigation should be addressed outside the current repository, and an issue has been opened in the relevant repository.

Verification
Resolved.

Suggestion 4: Implement Welcome Retry With Same wrapper_event_id After Failed Preview

Location
`crates/mdk-core/src/welcomes.rs#L319`

Synopsis
A decode or parse failure in the function `preview_welcome` stores a failed `ProcessedWelcome`, and the function `process_welcome` then treats the wrapper as already processed, which prevents retry for the same `wrapper_event_id`. This deviates from the documentation since retry semantics are suggested in MIP-02 here.

The processing flow is as follows:

1. On any decoding or parsing error, `preview_welcome` writes a `ProcessedWelcome` with state `Failed`.
2. For a failed `Welcome`, there is a `ProcessedWelcome` record but no stored `Welcome` row.
3. On retry with the same `wrapper_event_id`:
   a) `is_welcome_processed` returns `true`.
   b) It finds the `ProcessedWelcome`.
   c) It attempts to find a `Welcome` by `welcome_event_id` → None.
   d) It returns `Error::MissingWelcomeForProcessedWelcome`.

Mitigation

We recommend replacing the unconditional processed guard in the function `process_welcome` and using a state-aware check that treats `ProcessedWelcomeState::Failed` as reprocessable.

Status

The White Noise team has argued convincingly that retries for `Welcomes` provide no substantive benefit and has therefore elected not to implement the proposed change. The team has instead revised the documentation and error messages to reflect this decision and to clearly communicate the reason for the failure to users.

Verification

Resolved.

Suggestion 5: Prevent Panic Risk in process_welcome

Location

`crates/mdk-core/src/welcomes.rs#L133`

Synopsis

The function `process_welcome` unwraps `rumor_event.id` from an `UnsignedEvent`, which is derived from untrusted input. A malformed or non-NIP-59-compliant rumor (ID omitted) will panic the node before returning an error.

Mitigation

We recommend modifying the function `process_welcome` so that it validates the ID or fails gracefully instead of calling `unwrap()`.

Status

The White Noise team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 6: Address Performance-Related TODO in mdk-memory-storage

Location

`crates/mdk-sqlite-storage/src/groups.rs#L39`

Synopsis

In the `mdk-memory-storage` crate, the function `save_message` iterates over the entire per-group vector. As a result, handling `m` messages for a group costs `O(m)`, posing a denial of service risk that is explicitly mentioned in the threat model (T.10.2 and T.10.4).

Mitigation

We suggest resolving the item above.

Status

The White Noise team has implemented the mitigation as recommended.

Verification
Resolved.

Suggestion 7: Improve Code Quality in Encrypted Media

Location
`crates/mdk-core/src/encrypted_media/manager.rs#L158`

`crates/mdk-core/src/encrypted_media/manager.rs#L210`

Synopsis
During our extensive review of the codebase, our team identified opportunities for improving the readability and maintainability of the codebase. The following recommendations outline concrete improvements:

- Remove unnecessary integrity checks. AEAD algorithms already ensure authentication and integrity of data.
- Fix the comment to match the actual data structure.

Mitigation
We recommend addressing the items listed above to improve overall code quality, and using them as a baseline for identifying and remediating similar issues across the codebase.

Status
The Whitenoise team has corrected the comment but decided to retain the integrity check as a defense-in-depth measure.

Verification
Resolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts,

zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation

recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.