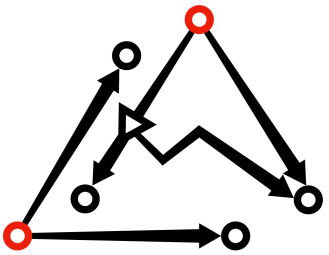

Least Authority

PRIVACY MATTERS

whitenoise-rs

Security Audit Report

White Noise

Final Audit Report: 1 April 2026

Table of Contents

Overview

Background

White Noise has requested that Least Authority perform a security audit of its codebase. White Noise serves as the Rust backend for its Flutter application, and the `whitenoise` crate leverages the MDK library while implementing the functionality required for a messaging application. Our team previously reviewed the Marmot Protocol and MDK, delivering a Feedback Summary report on November 7 and an Initial Audit Report on December 19, respectively.

Project Dates

- **February 19, 2026 - March 9, 2026:** Initial Code Review *(Completed)*
- **March 11, 2026:** Delivery of Initial Audit Report *(Completed)*
- **31 March, 2026:** Verification Review *(Completed)*
- **31 March, 2026:** Delivery of Final Audit Report *(Completed)*
- **1 April, 2026:** Delivery of Updated Final Audit Report *(Completed)*

Review Team

- Nikos Iliakis, Security Researcher and Engineer
- Miguel Quaresma, Security Researcher and Engineer
- Will Sklenars, 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 `whitenoise-rs` (the repository in scope) followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repository is considered in scope for the review:

- `whitenoise-rs`:
  https://github.com/parres-hq/whitenoise

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

- `bd74532567cba53afe104cf19967350572ef65b7`

For the verification, we examined the following Git revision:

- `720cec7153c1ba914f5c1915dc83f9c3d87356e9`

For the review, these repositories were cloned for use during the audit and for reference in this report:

- `marmot-protocol-whitenoise-rs`:
  https://github.com/LeastAuthority/marmot-protocol-whitenoise-rs

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

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:

- `Marmot`: https://github.com/marmot-protocol/marmot/tree/master
- RFC 9750: https://www.rfc-editor.org/rfc/rfc9750.html
- RFC 9420: https://www.rfc-editor.org/rfc/rfc9420.html

In addition, this audit report references the following documents:

- Previous Feedback Summary Report by Least Authority on the Marmot Protocol (pdf) *(delivered via email on 7 November 2025)*
- Previous Security Audit Report by Least Authority on the MDK (pdf) *(delivered via email on 19 December 2025)*

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

White Noise is an encrypted group chat application implementing the Marmot protocol. The Marmot protocol combines the MLS (Messaging Layer Security) protocol with `Nostr`'s decentralized network to provide private group messaging that does not rely on centralized servers. `whitenoise-rs` is the Rust backend for White Noise client applications. It leverages MDK (Marmot Developer Kit), the Rust library that implements core functionality of the Marmot protocol.

System Design

We reviewed the manner in which `whitenoise-rs` implements Marmot MIPs 00–04, including MDK-backed MLS flows (group lifecycle, key packages, and `Welcome` processing) and the repository's `Nostr`-side event handling and encrypted media workflows.

We examined the MLS commit lifecycle, including how commits are created, published, and merged. We identified an instance in which local state was merged before relay acceptance, potentially resulting in state divergence within a group (Issue C).

Our team compared the `Welcome` processing implementation against MIP-02 and identified issues with key package ownership verification and mandatory rotation (Issue B).

We assessed the key package fetch boundary and group invitation flow and found that protocol validation is not applied before key packages are passed to MDK (Issue E).

We also reviewed the account and session lifecycle, including account activation, background task management, and key material persistence. We identified issues with task cancellation on account switch (Issue H).

We compared the encrypted media upload, download, encryption, and storage flows against MIP-04 and found a missing post-decryption integrity check (Issue F), plaintext storage of keys (Issue I), and absent HTTPS enforcement on peer-specified media URLs (Issue G). In addition, we identified a potential attack vector relating to unbounded download size (Issue D), and provided a suggestion relating to upload timeouts (Suggestion 2).

Overall, the findings indicate inconsistent enforcement of protocol invariants, as well as silent error propagation, whereby failures in auxiliary subsystems can cause security-relevant operations, such as key rotation, to be skipped.

Dependencies

Running `cargo audit` yielded three vulnerabilities (including one high-severity issue). We recommend updating the affected dependencies (Suggestion 3).

Code Quality

We performed a manual review of the repositories in scope and found the code to be well-organized and aligned with standard Rust conventions.

Tests

The repositories in scope include an extensive integration test suite covering various flows.

Documentation and Code Comments

The project provides comprehensive documentation covering the system's intended functionality, including MIPs, the threat model, and data flows. The codebase is complemented by descriptive inline comments that aid in understanding the intended behavior of individual components, and public API functions are generally well-documented.

Scope

While the scope of this review did not include `mdk-core`, a key dependency, our team had already audited it in a previous engagement, and the scope of the present review remained sufficient.

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 Enforcement of Key Package Fetch Filter Enables Relay-Driven Resource Consumption	Medium	Resolved
Issue B: Welcome Message Processing Skips KeyPackage Ownership Verification and Mandatory Rotation	Medium	Resolved
Issue C: Local State Can Advance Before Relay Acceptance, Potentially Resulting in State Divergence	Medium	Resolved
Issue D: Unbounded Media Downloads Are a Potential Denial-of-Service (DoS) Attack Vector	Medium	Resolved
Issue E: Missing Capability Precheck Before Member Invitation	Low	Resolved
Issue F: Post-Decryption Integrity Check Absent	Low	Resolved
Issue G: Missing HTTPS Enforcement on Blossom URLs	Low	Resolved
Issue H: Background Tasks Not Canceled on Account Switch	Low	Resolved
Issue I: Blossom Authentication Keys Are Persisted in Plaintext, Increasing Key Exposure and Media-Deletion Risk	Low	Resolved
Issue J: Incorrect Content Length Check in MIP-03 Allows Structurally Invalid Payloads	Low	Resolved
Suggestion 1: Use “Protected Data” Store for macOS on Apple Silicon	Informational	Resolved
Suggestion 2: Add Timeout to upload_profile_picture Function	Informational	Resolved
Suggestion 3: Update Vulnerable Dependencies	Informational	Partially Resolved

Issue A: Missing Enforcement of Key Package Fetch Filter Enables Relay-Driven Resource Consumption

Location

src/whitenoise/key_packages.rs#L431

Synopsis

Key package fetching trusts relay responses without enforcing the requested filter, allowing relays to return excess events that trigger unnecessary processing.

Impact

Low.

A malicious or malfunctioning relay can induce avoidable network, parsing, and deletion work, which can degrade client availability and battery consumption.

Feasibility

Medium.

An attacker must control or influence at least one configured key package relay, or operate a noncompliant relay, and rely on `verify_subscriptions` and `ban_relay_on_mismatch` being false by default.

Severity

Medium.

Preconditions

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

- At least one account key package relay must be controlled by an attacker or behave noncompliantly.
- The configuration flags `verify_subscriptions` and `ban_relay_on_mismatch` must be set to `false`.
- The client must invoke the function `fetch_all_key_packages_for_account`, either directly or through `delete_all_key_packages_loop`.

Technical Details

The function `fetch_all_key_packages_for_account` constructs the variable `key_package_filter` with `Kind::MlsKeyPackage` and `author(account.pubkey)` but accepts all streamed events from `stream_events_from` without local verification that returned events match the filter. When `verify_subscriptions` and `ban_relay_on_mismatch` are false, mismatching responses are neither rejected nor used to penalize the relay, so a relay can return unrelated or excessive events.

If a relay returns many events, downstream logic such as `delete_all_key_packages_loop` repeatedly processes the returned set across rounds. In addition, `delete_key_packages_for_account_internal` may attempt parsing, local storage deletion, and `NIP-09` batch deletions for events that are not authored by the account, increasing computation and network traffic until termination conditions are met.

Remediation

We recommend filtering events returned by `stream_events_from` by rechecking `event.kind` and `event.pubkey` against the intended author in `fetch_all_key_packages_for_account`, and adopting `verify_subscriptions` a default-on invariant rather than a best-effort option.

Status

The White Noise team has resolved the issue by introducing a new function, `filter_key_package_events_for_account`, which filters events based on author and event kind.

Verification

Resolved.

Issue B: Welcome Message Processing Skips KeyPackage Ownership Verification and Mandatory Rotation

Location

`rc/whitenoise/event_processor/event_handlers/handle_giftwrap.rs#L91`

`src/whitenoise/event_processor/event_handlers/handle_giftwrap.rs#L114`

`src/whitenoise/event_processor/event_handlers/handle_giftwrap.rs#L390`

Synopsis

If the `EventId` of a `Welcome` is None or fails to be parsed, the check for ownership is silently skipped, and the `Welcome` is still passed to MDK. In addition, if the database lookup returns an error, execution proceeds to MLS acceptance instead of failing.
If the `Welcome` is successful, the key rotation returns without an error, so the key package is never marked, deleted from relays, or replaced with a freshly published key package.

Impact

Medium.

Accepted `Welcome` without `KeyPackage` ownership verification breaks the key lifecycle. The `KeyPackages` remain on the relays, and over repeated instances, the receiver will have the `KeyPackage` pool depleted. The receiver never verifies that the `Welcome` was intended for a `KeyPackage` that they control.

Feasibility

Medium.

MDK will still validate the `Welcome` cryptographically. However, the e tag exists in the `Nostr` layer, is not part of the MLS ciphertext, and may be omitted by a noncompliant sender. In addition, the database lookup fall-through path requires no adversarial action.

Severity

Medium.

Preconditions

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

- An attacker or a noncompliant sender must be able to produce or relay a gift-wrapped `Welcome` event addressed to the victim.
- For the database fall-through path, no adversarial action is required, as a fault alone is sufficient.

Technical Details

The `key_package_event_id` is parsed as an `Option<EventId>`. The precheck is guarded by `if let Some(ref kp_event_id) = key_package_event_id`, so the entire ownership lookup is skipped when the tag is absent.
When the tag is present but the database lookup returns `Err`, the handler logs a warning and falls through, allowing MLS processing to continue without confirming KP ownership.

```
if let Some(ref kp_event_id) = key_package_event_id {

    match PublishedKeyPackage::find_by_event_id(...).await {

        ...

        Err(e) => {

            // line 114 — DB error is non-fatal; falls through to MDK

            tracing::warn!("Failed to look up key package: {}, proceeding
anyway", e);

        }

    }

}
```

After a successful `Welcome`, the background task calls `rotate_key_package` with the same `Option<EventId>`. The function returns silently on `None`.

```
let Some(kp_event_id) = key_package_event_id else {

    return Ok(());

};
```

As a result, `PublishedKeyPackage::mark_consumed` is never called, relay-side deletion is never attempted, and `publish_key_package_for_account` is never invoked, leaving the account's KP pool unchanged and the consumed `KeyPackage` indefinitely available on relays.

Remediation

We recommend the following steps to remediate this issue:

- Rejecting the `Welcome` rumor if it does not carry a parseable `e` tag pointing to a known `KeyPackage` event ID, before invoking MDK.
- Returning an error or queuing the `Welcome` for retry if a storage error occurs during the ownership check, as this error is not safe to ignore.
- Surfacing an error in `rotate_key_package` if the `KeyPackage` event ID is absent, rather than silently returning.

Status

The White Noise team has implemented the remediations as recommended.

Verification

Resolved.

Issue C: Local State Can Advance Before Relay Acceptance, Potentially Resulting in State Divergence

Location

`src/whitenoise/event_processor/event_handlers/handle_mls_message.rs#L105-L121`

Synopsis

In the auto-commit proposal branch of the `handle_mls_message` function, local MLS state is merged regardless of whether the event is successfully published to relays. If publication fails, the local node has advanced to an epoch and state that other members never reach.

Impact

Medium.

A failed `publish` can leave the sender on a different MLS epoch than the rest of the group, and the group members will not agree on the current group state.

Feasibility

Medium.

This requires publication to fail across all targeted relays, for example, because of relay outages, connectivity issues, or relay-side refusal.

Severity

Medium.

Technical Details

In `handle_mls_message`, the code comment describes a `publish` to relays, followed by a local merge:

```
// Handle auto-committed proposals (e.g., admin auto-commits a
// member's self-removal): publish the resulting commit event so
// other group members learn about the change, then merge the
// pending commit into our local MLS state.
```

However, the code actually merges first and then publishes to relays.

`mdk.merge_pending_commit(group_id)?;` (line 113)

then `publish_event_to(...).await?;` (lines 115-121)

If the `publish` fails, the local merge is never rolled back.

Mitigation

We recommend the following measures to mitigate this issue:

- Using multiple reliable relays.
- Monitoring application logs for auto-commit `publish` failures and performing manual state recovery.

Remediation

We recommend reordering the auto-commit branch to match the behavior described in the code comment, publishing first and then merging locally only after a successful publish.

Status

The White Noise team has updated the implementation so that local state is now merged only after a successful publish.

Verification

Resolved.

Issue D: Unbounded Media Downloads Are a Potential Denial-of-Service (DoS) Attack Vector

Location

`src/whitenoise/media_files.rs#L257`

`src/whitenoise/groups.rs#L1112`

Synopsis

The application downloads media blobs into memory from user-supplied URLs, without checking against a host whitelist or enforcing media size limits. A malicious group member could send a message with a URL pointing to a malicious server and cause the victim to download an excessively large file, which could result in degraded application performance or denial of service due to memory pressure.

Impact

Medium.

The application could experience degraded performance, become unresponsive, or crash.

Feasibility

Medium.

The attack setup is straightforward for the malicious group member because it requires only sending a valid MLS message with an `imeta` URL pointing to attacker-controlled infrastructure. However, victim action is still required, as the victim must initiate the download. The most significant factor that limits the feasibility of the attack is that the attacker must be a valid group member.

Severity

Medium.

Preconditions

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

- The attacker must be a group member.
- The attacker must control a malicious server that can reply with excessively large payloads.
- The victim must approve the download of a media item sent by the attacker.

Technical Details

Untrusted media URLs can enter the system in MLS message content, processed by the `handle_mls_message` function, which calls `parse_imeta_tags_from_event`. The parsed `imeta` URL is persisted to the database as `blossom_url` without any check against a server whitelist. As a result, the URL could point to any destination chosen by the attacker. Subsequently, the download path trusts the attacker-provided URL stored in the database as `blossum_url`. The download is fully buffered in memory, with no data cap. There is, however, a 300-second timeout, which may render the attack ineffective if the victim's device has a slow network connection.

Mitigation

We suggest that users treat media URLs as potentially malicious and download files only from trusted senders.

Remediation

We recommend the following steps to remediate this issue:

- Checking the `Content-Length` header as an initial validation step, while recognizing that the header could be spoofed by an attacker.
- Using the streaming API to count bytes as they arrive and aborting when a defined threshold is exceeded.
- Implementing a server whitelist containing only approved servers.

Status

The White Noise team has updated the implementation so that the `Content-Length` header is now checked, and bytes are also counted as they arrive to enforce a hard cap of 100 MiB.

Verification

Resolved.

Issue E: Missing Capability Precheck Before Member Invitation

Location

`src/whitenoise/groups.rs#L556`

`src/nostr_manager/query.rs#L38`

Synopsis

Key packages fetched for prospective group members are passed directly to `mdk.add_members` without an application-layer precheck for ciphersuite compatibility, required capabilities, or required extensions. No `Nostr`-layer envelope validation (encoding tag, extension tags, or `i` tag consistency) is performed before handing the events to MDK, as acknowledged by the TODO at `query.rs#L38`. A validation helper (`has_encoding_tag`) exists in `key_packages.rs` but is not applied at the fetch boundary.

Impact

Low.

MDK performs cryptographic validation and will reject a genuinely incompatible key package at the MLS layer. However, the absence of an application-layer precheck allows noncompliant `KeyPackage` events to enter the flow without an explicit rejection. When MDK rejects such a package, the error is opaque, with no opportunity to fall back to an alternative key package or surface a clear diagnostic. A relay serving a noncompliant key package for a target user can therefore cause `add_members_to_group` to fail in a manner not readily distinguishable from a legitimate MLS error.

Feasibility

Low.

The attacker must control or influence a `KeyPackage` relay for at least one of the invited members and serve a `KeyPackage` with the incorrect ciphersuite parameters, missing required extensions, or a mismatched `i` tag.

Severity

Low.

Preconditions

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

- The attacker must control or influence a key package relay for a target member.
- The relay must serve a key package event that satisfies the `Nostr` layer filter but is noncompliant at the MLS layer.

Technical Details

`add_members_to_group` fetches one key package per invited member via `fetch_user_key_package` and accumulates them in `key_package_events`. The entire vector is passed to `mdk.add_members()` with no intermediate validation:

```
// groups.rs:

let some_event = self.nostr.fetch_user_key_package(*pk,
&relays_to_use_urls).await?;

let event = some_event.ok_or(...)?;

key_package_events.push(event);

// ...

let update_result = mdk.add_members(group_id, &key_package_events)?;
```

`fetch_user_key_package in query.rs applies only a Kind::MlsKeyPackage + author filter and explicitly defers protocol layer checks:`

```
// query.rs:38–39

// TODO: Add key package validation logic here to check key package tags

// for correct extensions and version
```

As a result, no validation is performed before the MDK invocation.

Remediation

We recommend applying a protocol-layer validator in `fetch_user_key_package` that rejects noncompliant events at fetch time, producing a clear error.

Status

The White Noise team has added a `key-package` validator.

Verification

Resolved.

Issue F: Post-Decryption Integrity Check Absent

Location

src/whitenoise/groups.rs#L1245-L1246

Synopsis

After decrypting a downloaded media blob, the implementation does not verify that the SHA-256 hash of the decrypted content matches the original hash stored in the `MediaReference`. The encrypted blob's hash is checked at download time, and `ChaCha20-Poly1305` provides AEAD authentication of the ciphertext, but no post-decryption check binds the plaintext to the claimed original hash, as required in MIP-04.

Impact

Low.

An attacker who can tamper with the `original_hash` field in local storage, or serve a `Blossom` blob encrypted with a different source file but under a valid key, could cause the client to silently accept and display content that does not correspond to the originally uploaded file. AEAD authentication limits this to scenarios in which the attacker controls the encryption key or metadata rather than the ciphertext alone.

Feasibility

Low.

The encrypted blob hash is verified at download time via `download_blob_from_blossom`. `ChaCha20-Poly1305` would reject a tampered ciphertext. Exploitation requires either local database metadata tampering or a compromised key derivation path.

Severity

Low.

Preconditions

The attacker must be able to tamper with `original_hash` in the local `media_files` table or produce a valid encryption of a different plaintext under the same derived key.

Technical Details

`download_and_decrypt_chat_media_blob` passes `original_file_hash` as part of the `MediaReference` struct, which MDK uses as AAD context during decryption. After `decrypt_from_download` returns successfully, the decrypted bytes are returned without verifying that `SHA-256(decrypted) ==original_file_hash`.

MIP-04 requires this explicit plaintext integrity check as a separate step from AEAD authentication, to confirm that the decrypted content corresponds to the originally uploaded file regardless of key derivation or metadata integrity.

Remediation

We recommend computing the SHA-256 hash of the plaintext returned by `decrypt_from_download`, comparing it with `original_file_hash`, and returning an error if the values do not match.

Status

The White Noise team has implemented the remediation. The client now computes SHA-256 hash of the decrypted value and compares it to `original_file_hash`, returning `HashMismatch` if they differ.

Verification

Resolved.

Issue G: Missing HTTPS Enforcement on Blossom URLs

Location

`src/whitenoise/groups.rs#L815`

`src/whitenoise/groups.rs#L926`

`src/whitenoise/groups.rs#L1211`

Synopsis

Blossom URLs stored in the `media_files` table and sourced from peer-controlled group data are used for media downloads without enforcing the HTTPS scheme. A group admin can specify an HTTP URL, causing the client to issue a cleartext HTTP request to an attacker-controlled server.

Impact

Low.

Media blobs are encrypted and hash-verified, so plaintext content is not exposed. However, a cleartext HTTP request leaks the blob hash, client IP, and download timing to the server operator and any network observer. A malicious group admin can exploit this to track which members download a resource and correlate group participation with IP addresses.

Feasibility

Medium.

The attacker must be a group admin with the ability to set a group image or contribute peer-specified media metadata. No network position is required because the victim client initiates the HTTP request directly to the attacker's server.

Severity

Low.

Preconditions

The attacker must be a group admin or must otherwise introduce a peer-specified `Blossom` URL into the group data.

Technical Details

`download_and_cache_group_image` retrieves the stored `blossom_url` and passes it to `download_blob_from_blossom` without performing a scheme check. Similarly, `download_and_decrypt_chat_media_blob` parses `media_file.blossom_url`. In both cases, `Url::parse` accepts any scheme.

Remediation

We recommend rejecting any parsed URL whose scheme is not HTTPS.

Status

The White Noise team has added `require_https`, which rejects any non-HTTPS `Blossom` URL.

Verification

Resolved.

Issue H: Background Tasks Not Canceled on Account Switch

Location

[src/whitenoise/accounts.rs#L1533]

[src/whitenoise/event_processor/event_handlers/handle_contact_list.rs#L193]

Synopsis

On login or account switch, `activate_account` replaces the background task cancellation sender with a fresh channel without first signaling `true` on the old sender. Background tasks from the prior session use `*rx.borrow()` to check for cancellation. Because the old sender is dropped without sending, `borrow` continues to return the last stored value indefinitely.

Impact

Medium.

Background tasks from a previous session continue running under the new session's context, issuing network queries and database writes on behalf of a logged-out account. Tasks cannot be externally stopped and continue to run until they exhaust their work queue or the process exits.

Feasibility

Low.

No adversarial action is required. The condition is triggered by any account switch or re-login within the same process lifetime.

Severity

Low.

Preconditions

The user must log out and then log in to a different account within the same process lifetime.

Technical Details

`activate_account` creates a fresh cancellation channel and replaces the entry in `background_task_cancellation`, dropping the old `watch::Sender`. Dropping the sender closes the channel without changing the stored value, and existing receivers' `borrow` calls continue to return `false`.

Because `borrow` returns the last stored value and does not distinguish between an open channel containing `false` and a closed channel that contained `false`, the task never observes cancellation and continues.

Remediation

We recommend the following steps to remediate this issue:

- Retrieving the old sender before replacing it and sending `true` to signal cancellation to all existing receivers.
- Replacing `borrow` with `changed`, which treats a closed channel as a cancellation signal.

Status

The White Noise team has resolved the issue by applying the recommended remediation.

Verification

Resolved.

Issue I: Blossom Authentication Keys Are Persisted in Plaintext, Increasing Key Exposure and Media-Deletion Risk

Location

`src/whitenoise/media_files.rs#L212`

Synopsis

Media upload authentication private keys are stored in the SQLite database to facilitate a future media deletion action. However, the keys are persisted in plaintext. If an attacker gains access to these keys, they could potentially use them to delete media.

Impact

Low.

Although the attacker may be able to delete the media, they would not be able to decrypt and view it. Therefore, the impact is limited to data availability.

Feasibility

Low.

The attacker would need to gain access to the SQLite database, which is likely sandboxed on a user's device and difficult to access.

Severity

Low.

Preconditions

The attacker must have access to the SQLite database or to a copy or backup of it.

Remediation

We recommend refraining from storing private keys in plaintext and encrypting them before they are persisted.

Status

The White Noise team has updated the implementation so that the database uses `SQLCipher` to encrypt data at rest. Our team considers this is an adequate remediation, especially considering that the keys are of relatively low value to an attacker.

Verification

Resolved.

Issue J: Incorrect Content Length Check in MIP-03 Allows Structurally Invalid Payloads

Location

`marmot-protocol-marmot/03.md?plain=1#L72`

Synopsis

The MIP-03 specification checks that `base64`-decoded `event.content` is at least 12 bytes before processing. However, a valid `ChaCha20-Poly1305` payload always contains a 12-byte nonce followed by ciphertext that includes, at minimum, a 16-byte authentication tag. The correct minimum length is therefore 28 bytes, not 12.

Impact

Low.

Payloads between 12 and 27 bytes pass the specification's validation check but are structurally invalid. However, this is unlikely to have a material effect, as most well-implemented cryptographic libraries will return an authentication error for invalid inputs.

Feasibility

Low.

Severity

Low.

Technical Details

The specification currently states:

```
If base64-decoded `event.content` is fewer than 12 bytes (nonce cannot be extracted), the event MUST be rejected.
```

The content layout is defined as:

```
event.content = base64(nonce || ciphertext)
```

Where:

- nonce = 12 bytes

- ciphertext = output of `ChaCha20-Poly1305.encrypt`

The ciphertext length is always the same as the plaintext length + 16 bytes. The additional 16 bytes are for the 16-byte authentication tag.

Therefore, `event.content` should have the following form:

`[12-byte nonce][ciphertext (encrypted plaintext + 16-byte tag)]`

Accordingly, the minimum valid decoded `event.content` length should be:

`12(nonce) + 0(empty message) 16(tag) = 28 bytes`.

This is in contrast to the 12-byte length minimum currently enforced by the specification.

Mitigation

We suggest applying the stricter 28-byte minimum check regardless of the current wording of the specification.

Remediation

We recommend updating the wording of the malformed content check in MIP-03 to require a minimum of 28 bytes for `event.content`.

Status

The White Noise team has updated the MIP-03 specification to require a minimum length of 28-bytes for the `event.content` field, as recommended. The White Noise team has also updated the implementation so that the MDK implementation performs the correct `>= 28` byte check.

Verification

Resolved.

Suggestions

Suggestion 1: Use “Protected Data” Store for macOS on Apple Silicon

Location

`src/whitenoise/mod.rs#L205`

Synopsis

The function `initialize_keyring_store` uses conditional compilation attributes to select a keystore to securely store credentials based on the underlying platform or operating system. For Apple platforms, it selects “protected data” when compiling for iOS and “legacy keychain” when compiling for macOS. The “legacy keychain” store is a legacy option that offers less control over how data is handled and might not be integrated with other security features such as secure enclaves or biometric authentication. Despite this, the initialization still defaults to “legacy keychain,” although it is supported on recent versions of macOS (10.15 and newer) on Apple Silicon.

Mitigation

We recommend distinguishing between Intel and Apple Silicon platforms on macOS when selecting a keystore. This can be achieved by splitting the cfg predicate, using `#[cfg(all(target_os =`

`"macos", target_arch = "x86_64"))]` and `#[cfg(all(target_os = "macos", target_arch = "aarch64"))]` respectively, and selecting `apple_native_keyring_store::protected` for Apple Silicon and `apple_native_keyring_store::keychain` for Intel.

Status

The White Noise team has implemented the mitigation as recommended.

Verification

Resolved.

Suggestion 2: Add Timeout to upload_profile_picture Function

Location

`src/whitenoise/accounts.rs#L404`

Synopsis

The `upload_profile_picture` function does not apply a timeout. As a result, a slow `Blossom` server could cause the upload to stall or hang.

Mitigation

We recommend adding `tokio::time::timeout` to the upload path with a reasonable duration configuration.

Status

The White Noise team has added a timeout for profile picture uploads.

Verification

Resolved.

Suggestion 3: Update Vulnerable Dependencies

Location

`Cargo.toml`

Synopsis

Running `cargo audit` against the project's dependencies reveals three vulnerabilities and one unmaintained dependency:

- `libcrux-ecdh 0.0.5` (RUSTSEC-2026-0023):
  - X25519 secret validation did not check buffer length or clamping.
  - Severity: none.
  - Transitive via: `mdk-core`.
  - Fixed in `>=0.0.6`.

- `quinn-proto 0.11.13` (RUSTSEC-2026-0037):
  - Denial of service in Quinn endpoints.
  - Severity: 8.7 (high).
  - Transitive via: `nostr-blossom`.

    - Fixed in `>=0.11.14`.

- `rsa 0.9.10` (RUSTSEC-2023-0071):
    - Marvin Attack, with potential key recovery through timing side channels.
    - Severity: 5.9 (medium).
    - Transitive via: `sqlx-mysql`.
    - No fixed version available.
    - The vulnerable RSA code is used only for MySQL authentication. White Noise uses SQLite exclusively, so this code path is never active.

Mitigation

We recommend updating vulnerable dependencies and adopting a process that emphasizes secure dependency management to mitigate supply chain risks. The process includes:

- Manually reviewing and assessing currently used dependencies;
- Upgrading dependencies with known vulnerabilities to patched versions where available;
- Replacing unmaintained dependencies with secure and actively maintained alternatives where possible;
- Pinning dependencies to specific versions in `Cargo.toml`;
- Only upgrading dependencies upon careful internal review for potential backward compatibility issues; and
- Including automated dependency auditing (for example, `cargo audit`) in the project's CI/CD workflow.

For `quinn-proto`, an upgrade to `>=0.11.14` is available and recommended given the high-severity denial-of-service (DoS) risk. For `libcrux-ecdh`, upgrading to `>=0.0.6` resolves the X25519 validation issue at the `mdk-core` level and should be coordinated with the MDK maintainers. For `rsa`, no fix is available upstream. The current suppression is acceptable given that the vulnerable code path is unreachable, but it should be re-evaluated if the `sqlx` dependency structure changes.

Status

The White Noise team has partially resolved the issue by updating one of the three vulnerable crates to newer versions: `libcrux-ecdh`. The `quinn-proto` crate and the `rsa` crate remain unchanged. The White Noise team has stated that the vulnerable code path in the `rsa` crate is unreachable and therefore not currently exploitable.

Verification

Partially 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.