Security Research Labs

OpenMLS Security Assurance Assessment

Threat model and hacking assessment report

v1.2, 11 March 2026

Prepared for:
Phoenix R&D

- Public - 41 pages -

Version	v1.2
Client	Phoenix R&D
Date	11 March 2026
Assessment Team	Constantin Schwarz Nils Ollrogge Bruno Produit Marc Heuse

Version History

Date	Version
2025-12-22	v0.1 – preliminary release
2026-02-03	v1.0 – draft report
2026-03-03	v1.1 – final report
2026-03-11	v1.2 – added comments from the developers in section 8.6 & 8.7

Content

Disclaimer

This report summarizes SRLabs' architecture and baseline security assessments of the OpenMLS library based on the artefacts and scope provided by the development team. Although the engagement was thorough, it cannot guarantee that every vulnerability was identified, nor that applying our recommendations will render current or future versions of OpenMLS entirely free of security issues.

Integrity Notice

Assessment Timeline

Security Research Labs performed the OpenMLS source code security assessment. The analysis has been performed over the course of 12 weeks, starting from the 16th of October 2025.

Date	Event
October 16, 2025	Assessment kick-off
December 22, 2025	Preliminary findings report (v0.1)
February 3, 2026	Draft report (v1.0)
March 3, 2026	Final report (v1.1) including mitigation checks

Table 1: Audit timeline

1 Executive summary

1.1 Engagement overview

Security Research Labs is an established provider of specialized security audit services, founded in 2010. This document presents the results of the secure code review of OpenMLS, a rust-based open-source library implementing the MLS protocol.

During this assessment, the developers provided access to the source code, relevant documentation, and supported the research team effectively by providing background information and protocol-specific details. The library was reviewed by SRLabs to assure that it is resilient to hacking and abuse, as well as giving applications that build on top of it the confidence of using a secure foundation.

SRLabs thanks the Sovereign Tech Agency for funding this independent audit.

1.2 Observations and risk

Over the course of the assessment, SRLabs recognized that the code was structured and written with a security-first approach in mind. It contains thoroughly thought-out measures to create an environment where security issues are less likely to happen, contributing to the overall robustness of the library.

Nevertheless, the research team identified eight issues ranging from high to informational severity levels. Most of the identified issues were related to DoS potential due to logic bugs or state synchronization problems. The OpenMLS developers remediated all security issues in close cooperation with us.

1.3 Recommendations

To ensure ongoing security, we recommend integrating constructive static analysis into the CI/CD pipeline to catch potential vulnerabilities early. Additionally, the cryptographic backend should enforce strong typing for inputs to guarantee consistent validation, as is already the case in other parts of the library. Furthermore, all validation documentation should be consolidated into a single, updated source of truth to eliminate discrepancies between the RFC and implementation.

2 Evolution suggestions

To ensure that OpenMLS is secure against further unknown or yet undiscovered threats, we recommend considering the following evolution suggestions and best practices described in this section.

2.1 Address currently open issues

We recommend addressing all reported issues, even if an individual issue has limited security impact. An attacker could still leverage it as part of a broader exploitation chain, potentially resulting in more severe consequences for users of the OpenMLS library.

2.2 Strong typing for crypto backend trait

The `OpenMlsCrypto` trait, which is implemented by cryptographic backends, does not enforce strong typing for all function inputs. For example, `verify_signature` accepts the public key as a raw byte slice (`&[u8]`), allowing backend implementations to treat the key as opaque data and potentially omit explicit validation steps such as format, length, or curve checks.

To reduce the risk of incomplete or inconsistent input validation across backends, we recommend strengthening the trait's function signatures by requiring validated, strongly typed inputs (e.g., trait-bounded key types). This would guide backend implementations toward performing proper key parsing and validation.

A similar approach has been taken internally in OpenMLS already, e.g. with the `VerifiableAuthenticatedContentIn` struct.

2.3 Centralize documentation and close gaps between specification and current implementation

We recommend eliminating the discrepancies between the validations listed in the OpenMLS book and those on validations.openmls.tech. There should be a single source of truth for the implementation status of validations, with clear references to the specification and corresponding code. This would make it easier for users of the OpenMLS library to verify validations themselves if desired and would also help OpenMLS developers ensure that no checks are missed.

Additionally, we recommend updating the documentation to reflect validations that have already been implemented but are not marked as complete. Based on our assessment, these are `valn1601` and `valn1401`.

An overview of the implementation status across the different validation requirements can be found in Appendix B: Implementation status of mandatory validation checks.

Finally, we recommend implementing all remaining missing checks for currently implemented features and updating the documentation accordingly. Based on our assessment, these are `valn0312`, `valn1602`, `valn1603` and `valn1604`.

2.4 CI based static analysis

The current CI/CD workflow of OpenMLS already covers multiple important aspects: Fuzzing, unit test execution, linting and benchmarking. We recommend integrating the static analysis tools described in chapter 5 as well, as these provide early feedback towards developers regarding insecure coding patterns and actual vulnerabilities. However, special care must be taken to implement them in a constructive manner instead of adding a layer of frustration for maintainers. A well-balanced approach would be, for example, to only block on findings if they are part of the current PR, ensuring that developers are well aware of the context for which an issue is reported.

3 Motivation and scope

This report presents the security audit conducted by SRLabs, which was funded by the Sovereign Tech Agency. It includes a threat model that provided the necessary guidance for us for the subsequent technical analysis of the OpenMLS library. Considering that OpenMLS will be used in critical infrastructure environments, special care was taken to address these higher-level risks in the threat model.

OpenMLS is a library that implements the MLS protocol specified by RFC9420 and RFC9750, serving as a building block for applications to implement end-to-end encrypted messaging in groups of two or more clients. The nature of these applications is intentionally being left as open as possible: While it is reasonable to implement a traditional chat messenger using OpenMLS, other more niche use cases must be considered for a thorough assessment.

At the center of OpenMLS lies a ratchet tree that holds protocol specific information and key material for all members of a group. Clients will derive various secrets from this ratchet tree on a per-epoch basis, most prominently the root of a dedicated, epoch-specific secret tree. The secret tree will be used to derive per-client key material for each message sent or received in the MLS protocol. These operations are carefully aligned to provide forward secrecy as well as post-compromise security. To keep the overall state of the protocol synchronized, clients exchange protocol messages via a delivery service. This abstract entity is responsible for forwarding messages between clients of the same group, as well as potentially providing group information and key material for new clients that might want to join a group. Because applications might want to implement custom functionality around this delivery service (e.g. access control), OpenMLS does not provide a delivery service implementation that is suitable for production usage. Similarly, applications need to bring their own authentication service for the likely scenario that clients should be able to validate the credentials of other users.

The choice of not implementing a delivery service already indicates that ensuring protocol security is a shared responsibility between the application and the OpenMLS library. This becomes even more apparent by the fact that OpenMLS does not provide any cryptographic primitives or even a storage backend for persisting the MLS state. While the library defines interfaces for these components, applications that want to make use of OpenMLS must provide their own implementations.

As the protocol security depends heavily on these modules, excluding them from the codebase narrows the scope of the audit significantly. Nevertheless, potential threats involving these modules were considered with adequate worst-case assumptions around application-specific implementations.

The in-scope components and their assigned priorities are reflected in Table 2.

Repository	Priority	Component(s)
https://github.com/openmls/openmls/	High	openmls
https://github.com/openmls/openmls/	Medium	traits
https://github.com/openmls/openmls/	Medium	basic_credential

Table 2: In-scope OpenMLS's components with audit priority

Note: The crypto and storage providers as well as the proof-of-concept implementations of a messaging client using OpenMLS and MLS delivery service were not in scope for this review.

4 Methodology

The methodology used in this assessment follows a structured, attacker-centric approach aimed at identifying flaws in the OpenMLS library. The methodology consists of four steps: (1) threat modelling, (2) security design coverage checks, (3) implementation baseline check, and finally (4) remediation support. The threat modelling was conducted in alignment with the STRIDE threat modelling framework.

4.1 STRIDE threat modelling framework

The STRIDE framework is a widely used security model that helps identify potential threats in software systems. It stands for Spoofing, Tampering, Repudiation, Information disclosure, Denial of Service, and Elevation of privilege, each representing a different category of security risks.

Spoofing

Spoofing refers to bugs or logical flaws that cause the OpenMLS library to incorrectly attribute MLS messages to the wrong sender. This can result from bugs or logical flaws in sender identification, credential handling, or signature verification. Such issues could allow a message to be processed as if it originated from a legitimate group member when it did not, enabling an attacker to send or receive MLS messages under a false identity and potentially gain unauthorized access to group communications.

Tampering

Tampering in OpenMLS involves failures to correctly detect or reject modified MLS messages or corrupted group state due to implementation errors. Examples include incomplete validation, incorrect enforcement of transcript hashes, or logical errors in state transition checks. These flaws could result in acceptance of altered messages or inconsistent group state across members.

Repudiation

Repudiation threats in the context of OpenMLS relate to the ability of an attacker to deny having sent or received specific MLS messages. For example, ambiguous or incomplete APIs for identifying the sender of a processed message, or loss of sender context during state updates, could prevent higher layers from reliably attributing actions to specific group members.

Information disclosure

Information disclosure occurs when sensitive data, such as message content, client credentials, or metadata is exposed to unauthorized parties. In OpenMLS, this may result from unintended exposure of cryptographic material or group state due to implementation issues, for example through improper memory handling of secrets, insufficient zeroization, leaking key material via error messages or debug output, or exposing internal state through public APIs. Such information disclosure could lead to data breaches, privacy violations, or further exploitation of sensitive data.

Denial of service (DoS)

Denial-of-service attacks in the context of OpenMLS aim to prevent legitimate users from sending or receiving messages by overwhelming the system with requests that cause excessive computation, memory exhaustion, or panics within the library. Examples include unbounded allocations during message parsing, repeated expensive validation steps, or logic that allows malformed messages to trigger repeated failed state transitions. These issues could crash the library or render it unusable, affecting the overall functionality of the system built on top of OpenMLS.

Elevation of privilege

Privilege escalation attacks involve an adversary wrongfully obtaining higher-level access. The details depend highly on the application implementing OpenMLS. For the scope of this audit, we considered gaining unauthorized access to groups an escalation of privilege, as this scenario is likely common across most applications relying on OpenMLS.

4.2 Threat modeling and attacks

The goal of the threat model framework is to determine specific areas of risk for the OpenMLS library, including both potential threats and concrete attack scenarios. Familiarity with these risk areas can provide guidance for the design of the implementation stack, the actual implementation of the stack, as well as security testing.

Our threat modelling consists of 3 stages, each building upon the previous one, resulting in a list of attacks and their severities, graded from low to critical.

In the first stage, each STRIDE category is assigned a *STRIDE Hacking Value* which measures how valuable a threat in each category is to an attacker. The *STRIDE Hacking Value* is categorized into low, medium or high, with the following definitions:

- **Low:** Threats in this STRIDE category offer the hacker little to no gain.
- **Medium:** Threats in this STRIDE category offer the hacker considerable gains.
- **High:** Threats in this STRIDE category offer the hacker high gains.

In the second stage, concrete threat scenarios are identified and mapped to their corresponding STRIDE categories. Each threat is assigned a *Threat Impact* value, which represents the potential damage the threat could cause to the project. *Threat Impact* is categorized into low, medium or high, with the following definitions:

- **Low:** Threat scenario would cause negligible damage to the project.
- **Medium:** Threat scenario poses a considerable threat to the project.
- **High:** Threat scenario poses an existential threat to the project.

Based on the *Threat Impact* and the *STRIDE Hacking Value* of the associated category, a *Threat Risk* is derived according to Table 3. The *Threat Risk* represents the overall risk posed by each identified threat.

STRIDE Hacking Value/Threat impact	Low	Medium	High
Low	Low	Medium	Medium
Medium	Medium	Medium	High
High	Medium	High	High

Table 3: Threat Risk calculation matrix

In the third and final stage, concrete attack scenarios are identified and mapped to the previously defined threats. Each attack is assigned an *Attack Feasibility*, which measures how likely an attack is to succeed, considering complexity, required skill, and resources. *Attack Feasibility* is categorized into low, medium or high, with the following definitions:

- **Low:** Attack requires significant expertise and uncommon conditions.
- **Medium:** Attack requires reasonable effort and standard attacker capabilities.
- **High:** Attack requires minimal effort or widely available tools.

Based on the *Attack Feasibility* and the *Threat Risk* of the associated threat, an *Attack Severity* is derived according to Table 4.

Threat Risk / Attack Feasibility	Low	Medium	High
Low	Low	Medium	Medium
Medium	Medium	Medium	High
High	Medium	High	Critical

Table 4: Attack Severity calculation matrix

Together, these stages define a structured approach that progresses from abstract STRIDE threat categories and their value to an attacker, through concrete threat scenarios and their associated risk, and finally to specific attack scenarios with assigned severities.

Applying the framework to the OpenMLS library, different areas of risk were identified. Table 5 provides a high-level overview of the identified hacking risks relevant to OpenMLS, including each STRIDE category with its associated hacking value, example threat scenarios, their derived threat risk, and corresponding example attacks.

The complete set of identified threat scenarios, together with the attacks that enable them, is documented in the threat model deliverable. This list can serve as a starting point for developers of the OpenMLS library when assessing security considerations for future feature development. By thinking in terms of threat scenarios and attacks during code review or feature ideation, many issues can be caught or even avoided altogether.

Scope Category	STRIDE Hacking Value	Example threat scenarios	Threat Risk	Example Attack Ideas
Spoofing	High	Impersonate users by sending messages with their account	High	Leverage missing or incorrectly implemented signature verification to impersonate another group member.
Tampering	Medium	Cause a group fork	Medium	Exploit improper handling of missed or replayed proposals or commits to desynchronize group members into different epochs.
Repudiation	Low	Send messages that are decrypted properly but are wrongly attributed to another group.	Medium	Forge a message that gets correctly decrypted to the wrong group
Information disclosure	High	Observe public messages and metadata to derive private information.	Low	Perform a Man-in-the-Middle attack on the networking layer
Denial of service	Low	Abuse the strict deletion / single use policy for key material to prevent the decryption of legitimate messages.	Medium	Send a message with a spoofed client origin, so that subsequent messages sent by the client can't be decrypted.

Privilege escalation	High	Regain access to a group a client has been removed from.	High	Improper implementation of the remove proposal allows attackers to regain access to old groups.

Table 5: Risk overview

4.3 Security design coverage check.

Next, we reviewed the OpenMLS library design for coverage against relevant hacking scenarios. For each scenario, we have investigated the following two aspects:

a. **Coverage**. Is each potential security vulnerability sufficiently covered by our audit?
b. **Underlying assumptions**. Which assumptions must hold true for the design to effectively reach the desired security goal?

4.4 Implementation check

As a third step, we tested the current OpenMLS library implementation for openings whereby any of the defined hacking scenarios could be executed.

To effectively review the OpenMLS codebase, we derived our code review strategy based on the threat model that we established in the first step. For each identified threat, hypothetical attacks were developed and mapped to their corresponding threat category, as outlined in Chapter 4.2.

Prioritizing potential risk for the library, the code was assessed for present protections against the respective threats and attacks as well as the vulnerabilities that make these attacks possible. For each threat, we:

1. Identified the relevant parts of the codebase, for example, the relevant crates and MLS group configuration.
2. Identified viable strategies for the code review. We performed manual code audits, fuzz testing, and manual tests where appropriate.
3. Ensured the code did not contain any vulnerabilities that could be used to execute the respective attacks. Otherwise, we ensured that sufficient protection measures against specific attacks were present.
4. Immediately reported any vulnerability that was discovered to the development team along with suggestions around mitigations.

We carried out a hybrid strategy utilizing a combination of code review, static tests, and dynamic tests (e.g., fuzz testing) to assess the security of the OpenMLS library.

While static and dynamic testing establishes a baseline assurance, the focus of this audit was on manual code review of the OpenMLS codebase to identify logic bugs, design flaws, and best practice deviations. **We reviewed the *openmls, traits* and *basic_credential* crates inside the OpenMLS repository up to commit *a3402f2* from 22nd of October 2025.** We aimed to trace the intended functionality of the in-scope crates and to assess whether an attacker can bypass/misuse/abuse these components or trigger unexpected behavior in the OpenMLS library due to logic bugs, implementation flaws or missing checks. Since the OpenMLS library codebase is entirely open source, it is realistic that an adversary could analyze the source code while preparing an attack.

Fuzz testing is a technique to identify issues in code that handles untrusted input. In OpenMLS's case, these are the public functions of the library.

Fuzz testing works by taking some valid input for a method under test, applying a semi-random mutation to it, and then invoking the method under test again with this semi-valid input. Through repeating this process, fuzz testing can unearth inputs that would cause a crash or other undefined behavior (e.g., integer overflows) in the method under test. The fuzz testing for this assessment utilized custom harnesses as well as the existing harnesses in *openmls/fuzz/fuzz_targets*.

4.5 Remediation support

The final step is supporting OpenMLS with the remediation process of identified issues. Each finding was documented and published with mitigation recommendations. Once the mitigation solution is implemented, the fix is verified by us to ensure that it mitigates the issue and does not introduce other bugs.

During the audit, findings were shared via a private GitHub repository (https://github.com/srlabs/openmls-audit/). We also used a private Signal channel for asynchronous communication and status updates. In addition, biweekly jour fixe meetings were held to provide detailed updates and address open questions.

5 Static analysis assessment

We utilized Semgrep, an open-source static analysis framework that enables lightweight, pattern-based matching on source code syntax, to detect common classes of vulnerabilities in various types of codebases. Specifically, at the kick-off of our manual review phase we applied a set of custom-developed Semgrep rules designed to identify known vulnerabilities, unsafe logic constructs, and anti-patterns frequently encountered in Rust-based projects.

In addition to Semgrep we employed Dylint, a framework for constructing custom linter plugins that integrate directly with Rust's compiler as part of its nightly toolchain. This approach allowed us to perform deep, syntax-aware analysis to flag code patterns of interest and identify error-prone constructs, deviations from safe Rust practices, and specific security anti-patterns within the codebase.

Upon completion of manual review phases for each feature, we further subjected the codebase to analysis using a curated set of reasoning-capable large language models (LLMs). These models were selected and benchmarked for their performance in identifying nuanced security risks and latent attack surfaces, thereby serving as a safeguard against human oversight or tunnel vision.

This methodology aims to complement and bolster the manual audit process by encapsulating it between two automated analysis stages: an initial rule-based scan to rapidly surface straightforward vulnerabilities, and a final LLM-driven assessment to highlight areas warranting deeper manual scrutiny.

To verify that the upstream dependencies don’t pose a security risk for OpenMLS and the applications implementing the library, we checked the used versions against the rustsec database of known vulnerabilities using cargo-audit.

6 Dynamic analysis assessment

Dynamic testing, mostly in the form of Fuzzing, represents a core pillar in the audit process. Since OpenMLS already provides some rudimentary harnesses, our approach was twofold: Initially we verified that the given harnesses do not lead to crashes in the latest version of the library. As soon as the fuzzer stopped generating new coverage, we shifted our approach to custom harnesses that represent a more complex usage scenario of OpenMLS.

Fuzz harness creation: We chose Ziggy (https://github.com/srlabs/ziggy/), an open-source tool developed in-house, as our fuzzing orchestration tool. As for the harnesses, we followed up on two approaches: One differential fuzzer to test both supported crypto backends for compatibility, and one stateful fuzzer that could perform various protocol-specific operations, while verifying the correctness of the state against a set of invariants.

Coverage analysis and optimization: Although modern fuzzers can achieve good coverage by utilizing various techniques, we manually optimized the harness to ease the fuzzer into finding as many execution paths as possible.

In our coverage analysis below, we only measure the OpenMLS codebase coverage that the harness is targeting.

6.1 Decoding fuzzers

The coverage results from the decoding fuzzers below show that, while providing satisfactory results for the component they specifically target, they are insufficient for discovering security issues in the other parts of the library.

Fuzzer	Component	Coverage Achieved
Message-decode	tls_codec	14.12%
Proposal-decode	tls_codec	13.14%
Key-package-decode	tls_codec	13.14%
Welcome-decode	tls_codec	10.10%

Table 6: Code coverage achieved by the provided decoding fuzzers.

6.2 Stateful fuzzer

As can be seen in Table 7, the stateful fuzzer achieves good coverage results across the key components of OpenMLS. The code paths not covered by the fuzzer are mostly tests or test utilities, even though there remains room for improving coverage, especially around the areas related to delivery service communication. This is to be expected, as the harness currently does not mock these aspects. When focusing on the critical parts of the OpenMLS library, which mostly reside in the `group` and `treesync` modules, the fuzzer achieves good coverage. Improving the numbers shown in the table below would mostly mean also fuzzing testing utilities, which in most cases would not provide a real-world benefit.

Component	Code Path	Coverage Achieved
OpenMLS (overall)	openmls	63.67%
Framing	openmls/src/framing	70.60%
Group	openmls/src/group	57.24%
Group (mls_group)	openmls/src/group/mls_group	55.57%
Group (public_group)	openmls/src/group/public_group	71.26%
Key schedule	openmls/src/schedule	62.93%
Ratchet tree	openmls/src/tree	90.36%
Treesync	openmls/src/treesync	80.75%

Table 7: Code coverage achieved by the stateful fuzzer developed in-house.

6.3 Differential fuzzer

The differential fuzzer between both crypto providers was developed during the early audit phase, mostly for the purpose of getting familiar with the codebase. It must be noted that both libraries are out of scope for the audit, and for this reason, no efforts were made to improve the coverage, even though the actual results were promising to begin with.

Component	Code Path	Coverage Achieved
libcrux backend	libcrux_crypto	46.81%
Rust crypto backend	openmls_rust_crypto	41.76%

Table 8: Code paths covered in each crypto provider

7 Findings summary

We identified 8 issues during our analysis of the crates in scope in the OpenMLS library codebase; more specifically, we found 1 high-severity, 3 medium-severity, 2 low-severity, and 2 information-level issues.

Critical	0	
High	1	
Medium	3	
Low	2	
Informational	2	
Total Issues	8	

The overview of all findings, together with their severity and the mitigation status can be found in Table 9:

ID	Issue	Severity	Status
S3-7	MAC verification accepts truncated/empty tags	High	Mitigated
S2-5	MlsGroup.store() does not check for GroupId collisions	Medium	Mitigated
S2-2	Past epoch credential lookup may fail after tree updates	Medium	Mitigated
S2-6	Clients joining MLS group do not fully verify support for group extensions	Medium	Mitigated
S1-4	Mismatches between documentation status of validations and code	Low	Mitigated
S1-3	Desync between Group State and Storage Provider	Low	Acknowledged
S0-1	Unbounded allocations may slow down protocol and lead to DoS of group members	Info	Risk Accepted
S0-8	Small spec divergence in proposal list validation	Info	Mitigated

Table 9: Findings overview

8 Detailed findings

8.1 S3-7: MAC verification accepts truncated/empty tags

Classification	CWE-354: Improper Validation of Integrity Check Value
Attack impact	T1 - Cause a group to fork T2 - Impersonate users by sending messages with their account T5 - Prevent valid user actions by abusing strict state validation requirements
Severity	High
Status	Mitigated

Background

The MLS protocol uses MACs to authenticate group membership and confirm agreement on group state during commits. In OpenMLS, these MACs are validated during incoming message processing and during commit creation and validation.

Issue description

Currently, OpenMLS accepts truncated MACs because the MAC equality check does not consider the input length. `Mac::eq` calls `equal_ct`, which compares only up to `min(len(a), len(b))` and therefore **does not** enforce equal lengths. As a result, a truncated MAC that matches the prefix of the expected MAC is treated as valid. Even a MAC of **zero length** is treated as valid.

Following code excerpt shows the `equal_ct` function. The issue lies in the fact that the loop will only iterate until any of the iterators are exhausted. In case of a zero length MAC, the loop doesn't run at all, and the function will return true immediately.

```
fn equal_ct(a: &[u8], b: &[u8]) -> bool {
    let mut diff = 0u8;
    for (l, r) in a.iter().zip(b.iter()) {
        diff |= l ^ r;
    }
    diff == 0
}
```

This issue affects any comparisons of the `membership_tag` or `confirmation_tag` in the OpenMLS codebase. Two unit-tests that demonstrate this issue can be found in Appendix A: Proof of Concepts.

The first test shows that a valid MAC is considered equal to a truncated MAC. It also demonstrates that even a zero-length MAC is treated as equal to the valid MAC.

The second test demonstrates that this MAC validation issue causes the `verify_membership` function to accept an empty membership tag.

Risk

Accepting truncated or empty MACs breaks MLS's membership and confirmation checks (`membership_tag` and `confirmation_tag`). An attacker with a compromised member signing key can bypass these protections without knowing the current group state. This defeats an important defense-in-depth property of MLS and can enable unauthorized `PublicMessage` injection or state desynchronization. As a result, we believe this to be a **high-severity** issue.

Mitigation

Enforce MAC length equality before comparing, by rejecting MACs whose length is not equal to `ciphersuite.mac_length()`.

8.2 S2-5: MlsGroup.store() does not check for GroupId collisions

Classification	CWE-694: Use of Multiple Resources with Duplicate Identifier
Attack impact	T3 - Perform a denial-of-service (DoS) attack to disrupt the messaging service
Severity	Medium
Status	Mitigated

Background

In MLS, each group is identified by a `GroupID`, which is used by clients and delivery services to associate protocol messages and stored cryptographic state with a specific group. `GroupIDs` are supplied by the application at group creation time and are expected to uniquely identify a group across its lifetime.

Issue description

MLS groups have a `GroupID` to uniquely identify them, for example when doing storage operations or while interacting with the delivery service. While the RFC states that these IDs **should** be generated in a collision resistant way, it does not define a mechanism to enforce this. Therefore, it must be expected that an application could see duplicate `GroupIDs`, especially if a malicious actor with knowledge of used IDs is considered.

When a new `MlsGroup` is created using `MlsGroup::builder().build()`, the underlying storage is not being checked for existing entries that reference the given `GroupID`. Instead, the `store()` function is called directly on the group, potentially overwriting existing entries.

Risk

Attackers could invite other clients to groups with a duplicate `GroupID`. if the client creates an `MlsGroup` from the invite, the key material of the previous group would be overwritten, rendering the original group unusable for the client.

The unit test demonstrating the issue can be found in Appendix A: Proof of Concepts.

This test crashes due to a debug assertion in the commit merging logic for the initial group, highlighting that it actually becomes unusable.

Mitigation

OpenMLS should provide an explicit API where applications can specify whether the `store()` function should fail if a group with the given ID is already present in storage, or if the underlying call should succeed. While creating a fresh group with an already existing GroupID might be desirable under certain circumstances, it should be a conscious choice to do so. A secure-by-default approach would include changing the current API to not overwrite existing groups, while providing another function to explicitly do this.

It is required that the issue is mitigated in OpenMLS itself, as storage providers don't have the ability to differentiate between the initial creation of a group and normal updates throughout the regular usage of the library.

8.3 S2-2: Past epoch credential lookup may fail after tree updates

Classification	CWE-825: Expired Pointer Dereference
Attack impact	T3 - Perform a denial-of-service (DoS) attack to disrupt the messaging service
Severity	Medium
Status	Mitigated

Background

MLS has support for decrypting delayed or out-of-order application messages by allowing groups to retain cryptographic material from previous epochs. To enable this, implementations may store past protocol state so that messages sent in earlier epochs can still be authenticated and decrypted after the group state has advanced.

Issue description

Groups that enable the decryption of past messages by setting `max_past_epochs > 0` depend on a saved secret tree stored in the `MessageSecretsStore` to decrypt past messages. Epoch secret trees are added to this store inside merge_commit which takes a vector containing all current members as parameters. Crucially, this vector is a condensed view of the current leaves of the ratchet tree that includes only **full** (non-blank) leaves.

When processing an older-epoch application message, the `parse_message` function resolves the credential for decrypting the message from the `MessageSecretsStore` via the leaf_node_index of the sender of the message. This `leaf_node_index` represents the plain index of the member in the tree, **including** blank leaves. This creates a potential mismatch between the `leaf_node_index` and the condensed member vector.

As a result, indexing the condensed member vector stored in the `MessageSecretsStore` with the `leaf_node_index` may point to the wrong credential or out of bounds of the vector.

The PoC documented in Appendix A: Proof of Concepts demonstrates this issue by doing the following:

1. Create a group with four members, Alice, Bob, Charlie, and David at leaf indexes (0, 1, 2, 3) respectively, and configure it with `max_past_epochs=1`.
2. Have Alice remove Bob, which blanks leaf index 1 and advances the group to epoch `N`. Internally, `merge_commit` stores the epoch `N` MessageSecrets along with the condensed member vector `[Alice, Charlie, David]`.
3. In epoch `N` Charlie and David each send an application message to Alice. We simulate the Delivery Service (DS) buffering these ciphertexts instead of delivering them immediately.
4. Alice performs a `self_update` moving the group to epoch `N+1`. The messages from Charlie and David are now "old" and will be decrypted using the `MessageSecretsStore` entry for epoch `N`.
5. Deliver the delayed messages from epoch `N` to Alice. Now two different errors will occur:
    1. For Charlie's message, Alice uses `leaf_node_index=2`. When indexing the condensed vector `[Alice, Charlie, David]`. index 2 now refers to **David's** credential. Signature verification fails and `ValidationError::InvalidSignature` is returned.
    2. For David's message, Alice uses `leaf_node_index = 3`. Index 3 is out of bounds of the condensed vector, so look_up_credential_with_key returns None, which returns `ValidationError::UnknownMember`.

Risk

As a result of this issue, valid past messages may fail to decrypt permanently, even when retained within the configured epoch window. A malicious DS could abuse this by delaying application messages from members that are positioned after blank leaf nodes until it sees a commit message from any member. Similarly, a malicious group member could trigger this scenario if they hold a position at the left side of other members. By leaving the group at the right time, they would create a blank leaf, resulting in an index shift for the other members.

This breaks the guarantee from RFC 9750 §8.4.2, which states that a malicious DS "should also not be able to undetectably remove, reorder, or replay messages".

Mitigation

We recommend storing per-epoch membership inside the `MessageSecretsStore` keyed by `leaf_node_index` instead of position; for example, replacing `Vec` with `HashMap<LeafNodeIndex, Member>`.

8.4 S2-6: Clients joining MLS group do not fully verify support for group extensions

Classification	CWE-358: Improperly Implemented Security Check for Standard
Attack impact	T1 - Cause a group to fork T3 - Perform a denial-of-service (DoS) attack to disrupt the messaging service. T5 - Prevent valid user actions by abusing strict state validation requirements.
Severity	Medium
Status	Mitigated

Background

Extensions in MLS allow groups to be customized with additional protocol features and application-defined behavior. `GroupContext` extensions specify optional or required functionality that all group members must understand to safely participate. During group joins, implementations must validate extension compatibility to ensure all members support the extensions used by the group.

Issue description

Currently, OpenMLS does not verify if clients joining a MLS group support every extension in the `GroupContext` for the group as mandated in section 13.4 of the MLS spec.

When processing a Welcome message, there exists a check that the client supports all extensions listed in the `required_capabilities`, however these are potentially only a subset of the extensions in the `GroupContext`.

A unit test demonstrating this issue can be found in Appendix A: Proof of Concepts.

The test creates an Unknown extension and adds it to the `GroupContext` of Alice's group. When Bob is invited to the Group and processes the Welcome message, it shows that there is **no** error thrown.

Risk

A client may join a group even though it does not support all extensions listed in the `GroupContext`, violating RFC 9420's requirement to reject such groups.

The impact of the issues could be high, depending on what an application uses extensions for. The impact depends on how critical those extensions are for the application's security policy. If they enforce access control or message semantics, this could have a high security impact.

Mitigation

Ensure that clients joining a group verify that they support every extension in the `GroupContext` for the group.

8.5 S1-4: Mismatches between documentation status of validations and code

Classification	CWE-1059: Insufficient Technical Documentation
Attack impact	T-1: Cause a group to fork T-2: Impersonate users by sending messages with their account T-7: Manipulate the ratchet tree / secret tree to considerably weaken the derived cryptographic material T-8: Regain access to a group they have been removed from.
Severity	Low
Status	Mitigated

Background

MLS defines a comprehensive set of validations to ensure protocol correctness and preserve its security guarantees. OpenMLS tracks the implementation status of these validations across multiple public sources, including validation.openmls.tech and the OpenMLS book.

Issue description

There are a couple of discrepancies between the status of validations as listed on validation.openmls.tech, the status listed in the book and the code.

The table summarizing the status of all validations based on our assessment can be found in Appendix B: Implementation status of mandatory validation checks.

Risk

Gaps or inconsistencies between the MLS specification and the OpenMLS implementation may weaken protocol compliance. Missing or incomplete validation steps could erode MLS security guarantees and potentially introduce exploitable states within the group management logic.

Mitigation

We recommend ensuring that there is one single source of truth for the implementation status of the validations and that the state of the code matches that of the documentation.

8.6 S1-3: Desync between Group State and Storage Provider

Classification	CWE-460: Improper Cleanup on Thrown Exception
Attack impact	T3 - Perform a denial-of-service (DoS) attack to disrupt the messaging service
Severity	Low
Status	Acknowledged

Background

OpenMLS has the concept of providers that implement application-specific requirements like persistent storage, or cryptographic primitives. These providers are passed as function parameters if required, so the respective parts of the library can use them.

Issue description

OpenMLS contains several functions that perform state-changing operations on one or more objects. These take an `OpenMlsProvider` as a parameter to directly persist the changes in storage. However, due to the abstract nature of these providers, it is not possible to make stability assumptions around them, and errors happening during any storage operation must be expected. While raising these errors to the application is an overall reasonable approach, the issue lies in the fact that these state-changing operations are not atomic. If the underlying storage provider errors, the state of the underlying object in memory is still modified, leading to a divergence between the currently running application and the state on-disk.

This erroneous behavior has been verified in the following functions:

- `merge_pending_commit()`
- `commit_to_pending_proposals()`
- process_message()
- propose_self_update()

Risk

Even though an application might realize that the storage write has failed and handle the corresponding error correctly, it has limited possibilities to address this situation. This leads to an unclear source of truth until the next successful storage write takes place: An application might continue to use the new state in one version, while relying on the persistent state in another, e.g. when recovering from a crash. This will lead to undefined behavior, potentially resulting in group members not being able to decrypt messages from other members.

Mitigation

Ideally, state-changing operations would be atomic, so that the in-memory object always stays in sync with the state that has been persisted by the provider. If an error occurs, the in-memory state would also be rolled back, effectively making the functions listed above free of side-effects.

If atomicity should not be guaranteed by OpenMLS, it is important that applications have the necessary tools available to resolve conflicting states.

Comment from the OpenMLS developers

While it is currently up to the application to use a storage provider that enables atomic storage access (e.g. via a Sqlite transaction), we appreciate the input and have already considered multiple approaches to make the storage provider trait safer.

8.7 S0-1: Unbounded allocations may slow down protocol and lead to DoS of group members

Classification	CWE-770: Allocation of Resources Without Limits or Throttling
Attack impact	T3 - Perform a denial-of-service (DoS) attack to disrupt the messaging service
Severity	Info
Status	Risk Accepted

Background

In OpenMLS, group members are represented as leaves in a balanced binary tree. All relevant attributes like their credentials, or the list of supported extensions are a part of this leaf.

Issue description

The ratchet tree implementation stores leaf nodes that can contain arbitrary sized data that could be abused by a malicious group member to slow down the protocol and DoS other members due to out of memory errors. Specifically, the LeafNodePayload can contain an UnknownExtension, which contains an unbounded vector.

By creating a `KeyPackage` containing a huge (50MiB) unknown extension, this issue can be demonstrated. A snippet for reproducing this behavior is listed in Appendix A: Proof of Concepts.

As the `UnknownExtension` will be stored in the tree inside the `LeafNodePayload`, this **significantly** slows down tree operations.

Executing the test in release mode and timing the execution took 4.030 seconds on a current generation ThinkPad. As comparison: running the same test with an unknown extension of `0x1000` bytes took 0.209s on the same hardware.

Risk

Allowing unbounded extension sizes in `LeafNodePayload` enables a malicious group member (or attacker controlling the Delivery Service) to craft `KeyPackage` or `LeafNode` objects with extremely large unknown extensions. These values are then stored in the ratchet tree and slow down any future tree operations.

This leads to:

- **Denial-of-Service (DoS)**: Excessive CPU and memory use during tree operations.
- **Out-of-Memory risks**: Large allocations can exhaust memory, crash the client, or trigger OS-level termination.
- **Persistent performance degradation**: Once added, every future tree operation is slowed down until the malicious member is removed.

Even though this behavior is within spec limits, as MLS considers variable length vectors up to a length of 2^30, the OpenMLS library should still have sensible defaults to protect applications using it.

Mitigation

To mitigate this issue, we recommend introducing configurable maximum sizes for user-controlled data structures and documenting the performance impact of having no limits, so that applications can take appropriate measures to protect themselves.

Comment from the OpenMLS developers

If a malicious part of the application (client or server) uses a large extension to mount a DoS attack against another part of the application, that attack will manifest itself before OpenMLS is involved in

any way. Any OpenMLS struct containing such an extension will have to be loaded into memory before it can even be deserialized, at which point the damage will already have been done. Applications for which such attacks are in scope should generally check sizes of incoming payloads.

8.8 S0-8: Small spec divergence in proposal list validation

Classification	-
Attack impact	-
Severity	Info
Status	Mitigated

Background

In MLS, different group members can propose changes to the group state (e.g. add / remove members) that are enacted with one member performing a commit. The RFC specifies multiple criteria under which a commit is valid, e.g. disallowing update proposals from the same member who performs the commit.

Issue description

There exists a small discrepancy between the proposal list validation steps defined in the MLS spec in section 12.2 and the OpenMLS code: valn0311 states that "members being added or removed by the Commit do not need to support the proposal type". However, validate_proposal_type_support builds a capabilities intersection over all current non-blank leaf nodes and never filters out members targeted by Remove proposals.

Risk

This is a spec-compliance discrepancy where OpenMLS is stricter than RFC 9420. We could not come up with any potential security issues caused by this behavior. At most it can cause unnecessary commit rejection.

Mitigation

To be fully MLS spec compliant, exclude members targeted by Remove proposals when computing the capabilities intersection in `validate_proposal_type_support`.

Appendix A: Proof of Concepts

PoC Snippets for Issue S3-7

```
[openmls_test::openmls_test]
fn test_mac_truncation_prefix_equality() {
    let provider = &Provider::default();

    let key = Secret::random(ciphersuite, provider.rand()).expect("Not enough
randomness.");
    let message = b"mac truncation test";
    let full = Mac::new(provider.crypto(), ciphersuite, &key, message)
        .expect("Unexpected error while computing MAC.");

    let mut truncated = full.mac_value.clone();
    truncated.pop();
    let truncated_mac = Mac {
        mac_value: truncated,
    };

    let zero_mac = VLBytes::new(vec![]);
    let zero_mac = Mac {
        mac_value: zero_mac,
    };

    assert_ne!(
        full.mac_value.as_slice().len(),
        truncated_mac.mac_value.as_slice().len(),
        "Truncated MAC should be shorter than the full MAC."
    );

    // BUG: truncated mac is seen as equal
    assert_eq!(full, truncated_mac);

    // BUG: zero mac is seen as equal
    assert_eq!(full, zero_mac);
}
```

```
#[openmls_test::openmls_test]
fn membership_tag_accepts_empty_mac() {
    let provider = &Provider::default();
    let (_credential, signature_keys) =
        test_utils::new_credential(provider, b"Creator",
ciphersuite.signature_algorithm());
    let group_context = GroupContext::new(
        ciphersuite,
        GroupId::random(provider.rand()),
        1,
        vec![],
        vec![],
        Extensions::empty(),
    );
    let membership_key = MembershipKey::from_secret(
        Secret::random(ciphersuite, provider.rand()).expect("Not enough
randomness."),
    );
    let public_message: PublicMessage = AuthenticatedContent::new_application(
        LeafNodeIndex::new(987543210),
        &[1, 2, 3],
        &[4, 5, 6],
        &group_context,
        &signature_keys,
```

```
    )
    .expect("An unexpected error occurred.")
    .into();

    let mut public_message = PublicMessageIn::from(public_message);
    let serialized_context = group_context.tls_serialize_detached().unwrap();
    public_message
        .set_membership_tag(provider, ciphersuite, &membership_key,
&serialized_context)
        .expect("Error setting membership tag.");

    public_message.membership_tag = Some(MembershipTag(Mac {
        mac_value: VLBytes::new(vec![]),
    }));

    // BUG: Empty membership tags are accepted.
    assert!(public_message
        .verify_membership(
            provider.crypto(),
            ciphersuite,
            &membership_key,
            &serialized_context
        )
        .is_ok());
}
```

PoC Snippet for Issue S2-5

```
#[openmls_test::openmls_test]
fn external_psk_proposal_commit_flow() {
    let provider = &Provider::default();
    // Create group "a" with creator 213 (simulated)
    let (creator_credential, creator_signer) =
        test_utils::new_credential(provider, b"creator",
ciphersuite.signature_algorithm());
    let mut group = MlsGroup::builder()
        .with_group_id(GroupId::from_slice("a".as_bytes()))
        .ciphersuite(ciphersuite)
        .build(provider, &creator_signer, creator_credential)
        .expect("Error creating group");
    // Create External PSK proposal
    let external_psk_id = vec![];
    let external_psk_nonce = vec![
        98, 98, 255, 255, 255, 127, 213, 192, 183, 183, 155, 183, 183, 182, 213,
213, 213, 213,
        213, 125, 125, 125, 125, 125, 125, 120, 125, 213, 213, 213, 185, 183,
    ];
    let external_psk = PreSharedKeyId::external(external_psk_id.clone(),
external_psk_nonce);
    external_psk.store(provider, &external_psk_id)?;
    let proposal = group
        .propose_external_psk_by_value(provider, &creator_signer, external_psk)
        .expect("Error creating External PSK proposal");
    // Create group "a" again with creator 213 (simulated)
    let (creator_credential_2, creator_signer_2) =
        test_utils::new_credential(provider, b"creator2",
ciphersuite.signature_algorithm());
    let mut group2 = MlsGroup::builder()
        .with_group_id(GroupId::from_slice("a".as_bytes()))
        .ciphersuite(ciphersuite)
        .build(provider, &creator_signer_2, creator_credential_2)
        .expect("Error creating second group");
    let (_commit, _welcome_option, _group_info_option) = group
        .commit_to_pending_proposals(provider, &creator_signer)
```

```
        .expect("Error creating commit");
    group
        .merge_pending_commit(provider)
        .expect("error merging commit");
}
```

PoC Snippet for Issue S2-2

```
#[openmls_test::openmls_test]
fn old_messages_with_blank_leaves() {
    let provider = &Provider::default();

    let (alice_cred, alice_signer) =
        new_credential(provider, b"Alice", ciphersuite.signature_algorithm());
    let (bob_cred, bob_signer) =
        new_credential(provider, b"Bob", ciphersuite.signature_algorithm());
    let (charlie_cred, charlie_signer) =
        new_credential(provider, b"Charlie", ciphersuite.signature_algorithm());
    let (david_cred, david_signer) =
        new_credential(provider, b"David", ciphersuite.signature_algorithm());

    let apply_commit = |group: &mut MlsGroup, msg: MlsMessageOut| {
        let protocol = msg.into_protocol_message().unwrap();
        let processed = group.process_message(provider, protocol).unwrap();
        let ProcessedMessageContent::StagedCommitMessage(staged_commit) =
processed.into_content()
        else {
            panic!("expected staged commit");
        };
        group
            .merge_staged_commit(provider, *staged_commit)
            .expect("error merging staged commit");
    };

    let bob_kpb = KeyPackageBundle::generate(provider, &bob_signer, ciphersuite,
bob_cred.clone());
    let charlie_kpb =
        KeyPackageBundle::generate(provider, &charlie_signer, ciphersuite,
charlie_cred.clone());
    let david_kpb =
        KeyPackageBundle::generate(provider, &david_signer, ciphersuite,
david_cred.clone());

    let create_config = MlsGroupCreateConfig::builder()
        .ciphersuite(ciphersuite)
        .max_past_epochs(1)
        .build();
    let join_config = create_config.join_config().clone();

    let mut alice_group =
        MlsGroup::new(provider, &alice_signer, &create_config,
alice_cred.clone())
            .expect("failed to create group");

    // Alice adds Bob
    let (_commit_bob, _welcome_bob, _group_info) = alice_group
        .add_members(provider, &alice_signer, from_ref(bob_kpb.key_package()))
        .expect("could not add Bob");
    alice_group
        .merge_pending_commit(provider)
        .expect("could not merge add-member commit");

    // Alice adds Charlie
    let (_commit_charlie, welcome_charlie, _group_info) = alice_group
```

```
        .add_members(provider, &alice_signer,
from_ref(charlie_kpb.key_package()))
        .expect("could not add Charlie");
    alice_group
        .merge_pending_commit(provider)
        .expect("could not merge add-member commit");

    let ratchet_tree = alice_group.export_ratchet_tree();
    let welcome: MlsMessageIn = welcome_charlie.into();
    let welcome = welcome.into_welcome().expect("expected welcome message");
    let mut charlie_group =
        StagedWelcome::new_from_welcome(provider, &join_config, welcome,
Some(ratchet_tree.into()))
            .expect("error staging join for Charlie")
            .into_group(provider)
            .expect("error creating Charlie's group");

    assert_eq!(charlie_group.own_leaf_index().u32(), 2);

    // Alice adds David
    let (commit_david, welcome_david, _group_info) = alice_group
        .add_members(provider, &alice_signer, from_ref(david_kpb.key_package()))
        .expect("could not add David");
    alice_group
        .merge_pending_commit(provider)
        .expect("could not merge add-member commit");

    let ratchet_tree = alice_group.export_ratchet_tree();
    let welcome: MlsMessageIn = welcome_david.into();
    let welcome = welcome.into_welcome().expect("expected welcome message");
    let mut david_group =
        StagedWelcome::new_from_welcome(provider, &join_config, welcome,
Some(ratchet_tree.into()))
            .expect("error staging join for david")
            .into_group(provider)
            .expect("error creating Charlie's group");

    assert_eq!(david_group.own_leaf_index().u32(), 3);

    // tell charlie about david
    apply_commit(&mut charlie_group, commit_david);

    assert_eq!(alice_group.own_leaf_index().u32(), 0);

    // Figure out Bob's leaf index so we can remove him.
    let bob_index = alice_group
        .members()
        .find(|member| member.credential == bob_cred.credential)
        .expect("Bob missing from membership list")
        .index;
    assert_eq!(bob_index.u32(), 1);
    assert!(
        charlie_group.own_leaf_index().u32() > bob_index.u32(),
        "Charlie must sit above the blank leaf to expose the bug"
    );

    // Alice removes Bob, leaving a blank leaf between Alice and Charlie
    let (remove_commit, _, _) = alice_group
        .remove_members(provider, &alice_signer, &[bob_index])
        .expect("could not create removal commit");
    alice_group
        .merge_pending_commit(provider)
        .expect("could not merge removal commit");
```

```
    println!("Apply removal of bob to charlie and david");
    apply_commit(&mut charlie_group, remove_commit.clone());
    apply_commit(&mut david_group, remove_commit);

    // Charlie sends an application message in the epoch that still contains the
blank leaf.
    let message_charlie = charlie_group
        .create_message(provider, &charlie_signer, b"delayed application")
        .expect("could not create application message");

    // David also sends a message
    let message_david = david_group
        .create_message(provider, &david_signer, b"delayed application2")
        .expect("could not create application message");

    // Advance to the next epoch so the message becomes "old" and must use the
past store which stores group members in the dense vector
    let (update_commit, _, _) = alice_group
        .self_update(provider, &alice_signer, LeafNodeParameters::default())
        .expect("could not create self-update commit")
        .into_contents();
    alice_group
        .merge_pending_commit(provider)
        .expect("could not merge self-update commit");
    apply_commit(&mut charlie_group, update_commit);

    // DS releases Charlie's buffered message to Alice.
    let app_in = MlsMessageIn::from(message_charlie);
    // Alice will try to process the message using the leaf index of Charlie,
which now points to David in the condensed vector
    // As a result, signature validation will fail since Alice tries to use
Davids keys to verify a message from Charlie
    let result = alice_group.process_message(provider,
app_in.try_into_protocol_message().unwrap());
    match result {

Err(ProcessMessageError::ValidationError(ValidationError::InvalidSignature)) =>
{}
        other => panic!("Expected ValidationError::InvalidSignature, got
{other:?}"),
    }

    // DS releases David's buffered message to Alice.
    let app_in = MlsMessageIn::from(message_david);
    // Alice will try to process the message using the leaf index of David,
which now points to an out of bounds index in condensed vector.
    // As a result, an UnknownMember error will be thrown
    let result = alice_group.process_message(provider,
app_in.try_into_protocol_message().unwrap());
    match result {

Err(ProcessMessageError::ValidationError(ValidationError::UnknownMember)) => {}
        other => panic!("Expected ValidationError::UnknownMember, got
{other:?}"),
    }
}
```

PoC Snippet for Issue S2-6

```
#[openmls_test]
fn join_rejects_unsupported_group_context_extension() {
    let alice_party = PartyState::<Provider>::generate("alice", ciphersuite);
    let bob_party = PartyState::<Provider>::generate("bob", ciphersuite);
```

```
    let gc_extensions = Extensions::single(Extension::Unknown(
        0x4141,
        crate::extensions::UnknownExtension(vec![0x01]),
    ));

    let alice_group = MlsGroup::builder()
        .ciphersuite(ciphersuite)
        .with_wire_format_policy(WireFormatPolicy::new(
            OutgoingWireFormatPolicy::AlwaysPlaintext,
            IncomingWireFormatPolicy::Mixed,
        ))
        .with_group_context_extensions(gc_extensions)
        .expect("error setting group context extensions")
        .build(
            &alice_party.provider,
            &alice_party.signer,
            alice_party.credential_with_key.clone(),
        )
        .expect("error creating group using builder");

    let mut alice = MemberState {
        party: alice_party,
        group: alice_group,
    };

    let bob_key_package = bob_party.key_package(ciphersuite, |builder| builder);
    alice.propose_add_member(bob_key_package.key_package());

    let (_, Some(welcome), _) = alice.commit_and_merge_pending() else {
        panic!("expected receiving a welcome")
    };

    let welcome: MlsMessageIn = welcome.into();
    let welcome = welcome
        .into_welcome()
        .expect("expected message to be a welcome");

    if let Ok(staged) = StagedWelcome::new_from_welcome(
        &bob_party.provider,
        alice.group.configuration(),
        welcome,
        Some(alice.group.export_ratchet_tree().into()),
    ) {
        assert!(alice
            .group
            .extensions()
            .contains(ExtensionType::Unknown(0x4141)));

        assert_eq!(
            false,
            bob_party
                .key_package_bundle
                .key_package
                .extensions()
                .contains(ExtensionType::Unknown(0x4141))
        );

        assert!(staged
            .group_context()
            .extensions()
            .contains(ExtensionType::Unknown(0x4141)));
```

```
        unreachable!("join should reject unsupported GroupContext extensions");
    }
}
```

PoC Snippet for Issue S0-1

```
#[openmls_test::openmls_test]
fn test_huge_unknown_extension_in_keypackage() {
    let alice_provider = &Provider::default();
    let bob_provider = &Provider::default();

    let (alice_credential_with_key, alice_signature_keys) =
        test_utils::new_credential(alice_provider, b"Alice",
ciphersuite.signature_algorithm());

    let (bob_credential_with_key, bob_signature_keys) =
        test_utils::new_credential(bob_provider, b"Bob",
ciphersuite.signature_algorithm());

    // === Bob creates a KeyPackage with a big unknown extension ===
    // This extension will be stored in Bob's LeafNode in the tree
    let huge_extension_data = vec![0x42; 50_000_000]; // 50MB of data
    let huge_extension = Extension::Unknown(0xF042,
UnknownExtension(huge_extension_data));

    let extensions = Extensions::single(huge_extension);

    // Bob needs to advertise support for this extension in capabilities
    let capabilities = Capabilities::new(
        None,
        None,
        Some(&[ExtensionType::Unknown(0xF042)]),
        None,
        None,
    );

    let bob_key_package_bundle = KeyPackage::builder()
        .leaf_node_extensions(extensions)
        .unwrap()
        .leaf_node_capabilities(capabilities)
        .build(
            ciphersuite,
            bob_provider,
            &bob_signature_keys,
            bob_credential_with_key.clone(),
        )
        .expect("Failed to build KeyPackage with huge extension");

    let bob_key_package = bob_key_package_bundle.key_package();

    let serialized_kp = bob_key_package
        .tls_serialize_detached()
        .expect("Failed to serialize KeyPackage");

    println!(
        "KeyPackage size with huge extension: {} MB",
        serialized_kp.len() / 1_000_000
    );

    let mut alice_group = MlsGroup::builder()
        .ciphersuite(ciphersuite)
        .use_ratchet_tree_extension(true)
        .build(
            alice_provider,
```

```
            &alice_signature_keys,
            alice_credential_with_key.clone(),
        )
        .expect("Error creating group.");

    let (_commit, welcome, _group_info_option) = alice_group
        .add_members(
            alice_provider,
            &alice_signature_keys,
            from_ref(bob_key_package),
        )
        .expect("An unexpected error occurred.");

    alice_group.merge_pending_commit(alice_provider).unwrap();

    let exported_tree = alice_group.export_ratchet_tree();
    let tree_bytes = exported_tree
        .tls_serialize_detached()
        .expect("Failed to serialize tree");

    println!(
        "Ratchet tree size after adding Bob: {} MB",
        tree_bytes.len() / 1_000_000
    );
    let welcome_bytes = welcome
        .tls_serialize_detached()
        .expect("Failed to serialize welcome");

    println!(
        "Welcome message size: {} MB",
        welcome_bytes.len() / 1_000_000
    );
```

Appendix B: Implementation status of mandatory validation checks

Id	Implemented	Matches documentation	Notes
valn0101	✗	✓	
valn0102	✓	✓	
valn0103	✓	✓	
valn0104	✓	✓	
valn0105	✓	✓	
valn0106	✓	✓	
valn0107	✓	✓	
valn0108	✓	✓	
valn0109	✓	✓	
valn0110	✓	✓	
valn0111	✓	✓	
valn0112	✓	✓	
valn0113	✓	✓	
valn0201	✓	✓	
valn0202	✓	✓	
valn0203	✓	✓	
valn0204	✓	✓	
valn0205	✓	✓	
valn0301	✓	✗	Only documented in OpenMLS book in section "Semantic validation of proposals covered by a Commit"
valn0302	✓	✗	Only documented in OpenMLS book as `ValSem111`
valn0303	✓	✗	Only documented in OpenMLS book as `ValSem200`
valn0304	✓	✓	

valn0305	✅	✅	
valn0306	✅	✅	
valn0307	✅	❌	Only documented in OpenMLS book as `ValSem403`
valn0308	✅	✅	
valn0309	❌	✅	
valn0310	✅	✅	
valn0311	✅	✅	
valn0312	❌	✅	
valn0401	✅	✅	
valn0402	✅	✅	
valn0403	✅	✅	No check necessary
valn0404	✅	✅	
valn0405	✅	✅	
valn0406	✅	✅	
valn0407	✅	✅	
valn0408	✅	✅	Implicitly implemented
valn0501	✅	✅	
valn0601	✅	✅	
valn0701	✅	✅	
valn0801	✅	✅	
valn0802	✅	✅	
valn0803	✅	✅	
valn0901	❌	✅	
valn1001	✅	✅	
valn1101	✅	✅	

valn1201	✅	✅	
valn1202	✅	✅	
valn1203	✅	✅	
valn1204	✅	✅	
valn1205	✅	✅	
valn1206	✅	✅	
valn1207	✅	✅	
valn1208	✅	✅	
valn1209	✅	❌	Implemented in `validate_key_uniqueness`, but not documented
valn1301	✅	✅	
valn1302	✅	✅	
valn1303	✅	✅	
valn1304	✅	✅	
valn1305	✅	✅	
valn1306	✅	✅	
valn1307	✅	✅	
valn1401	✅	❌	Documented Missing, but rg found references.
valn1402	✅	✅	
valn1403	❌	✅	
valn1404	✅	✅	
valn1405	✅	✅	
valn1406	✅	✅	
valn1407	✅	✅	
valn1408	✅	✅	
valn1409	✅	✅	

valn1410	✅	✅	
valn1411	✅	❌	Only documented in OpenMLS book as `ValSem205`
valn1412	❌	✅	
valn1413	❌	✅	
valn1414	❌	✅	
valn1501	✅	✅	
valn1502	✅	✅	
valn1503	✅	✅	
valn1504	✅	✅	
valn1601	✅	❌	Documented Unknown, but rg found references.
valn1602	❌	✅	
valn1603	❌	✅	
valn1604	❌	✅	

Appendix C: Technical services

Security Research Labs delivers extensive technical expertise to meet your security needs. Our comprehensive services include software and hardware evaluation, penetration testing, red team testing, incident response, and reverse engineering. We aim to equip your organization with the security knowledge essential for achieving your objectives.

SOFTWARE EVALUATION We provide assessments of application, system, and mobile code, drawing on our employees' decades of experience in developing and securing a wide variety of applications. Our work includes design and architecture reviews, data flow and threat modelling, and code analysis with targeted fuzzing to find exploitable issues.

BLOCKCHAIN SECURITY ASSESSMENTS We offer specialized security assessments for blockchain technologies, focusing on the unique challenges posed by decentralized systems. Our services include smart contract audits, consensus mechanism evaluations, and vulnerability assessments specific to blockchain infrastructure. Leveraging our deep understanding of blockchain technology, we ensure your decentralized applications and networks are secure and robust.

POLKADOT ECOSYSTEM SECURITY We provide comprehensive security services tailored to the Polkadot ecosystem, including parachains, relay chains, and cross-chain communication protocols. Our expertise covers runtime misconfiguration detection, benchmarking validation, cryptographic implementation reviews, and XCM exploitation prevention. Our goal is to help you maintain a secure and resilient Polkadot environment, safeguarding your network against potential threats.

TELCO SECURITY We deliver specialized security assessments for telecommunications networks, addressing the unique challenges of securing large-scale and critical communication infrastructures. Our services encompass vulnerability assessments, secure network architecture reviews, and protocol analysis. With a deep understanding of telco environments, we ensure robust protection against cyberthreats, helping maintain the integrity and availability of your telecommunications services.

DEVICE TESTING Our comprehensive device testing services cover a wide range of hardware, from IoT devices and embedded systems to consumer electronics and industrial controls. We perform rigorous security evaluations, including firmware analysis, penetration testing, and hardware-level assessments, to identify vulnerabilities and ensure your devices meet the highest security standards. Our goal is to safeguard your hardware against potential attacks and operational failures.

CODE AUDITING We provide in-depth code auditing services to identify and mitigate security vulnerabilities within your software. Our approach includes thorough manual reviews, automated static analysis, and targeted fuzzing to uncover critical issues such as logic flaws, insecure coding practices, and exploitable vulnerabilities. By leveraging our expertise in secure software development, we help you enhance the security and reliability of your codebase, ensuring robust protection against potential threats.

PENETRATION & RED TEAM TESTING We perform high-end penetration tests that mimic the work of sophisticated adversaries. We follow a formal penetration testing methodology that emphasizes repeatable, actionable results that give your team a sense of the overall security posture of your organization.

SOURCE CODE-ASSISTED SECURITY EVALUATIONS We conduct security evaluations and penetration tests based on our code-assisted methodology that lets us find deeper vulnerabilities, logic flaws, and fuzzing targets than a black-box test would reveal. This gives your team a stronger assurance that the significant security-impacting flaws have been found and corrected.

SECURITY DEVELOPMENT LIFECYCLE CONSULTING We guide organizations through the Security Development Lifecycle to integrate security at every phase of software development. Our services include secure coding training, threat modelling, security design reviews, and automated security testing implementation. By embedding security practices into your development processes, we help you proactively identify and mitigate vulnerabilities, ensuring robust and secure software delivery from inception to deployment.

REVERSE ENGINEERING We assist clients with reverse engineering efforts that are not associated with malware or incident response. We also provide expertise in investigations and litigation by acting as experts in cases of suspected intellectual property theft.

HARDWARE EVALUATION We evaluate new hardware devices ranging from novel microprocessor designs, embedded systems, mobile devices, and consumer-facing end products to core networking equipment that powers Internet backbones.

VULNERABILITY PRIORITIZATION We streamline vulnerability information processing by consolidating data from compliance checks, audit findings, penetration tests, and red team insights. Our prioritization and automation strategies ensure that the most critical vulnerabilities are addressed promptly, enhancing your organization's security posture. By systematically categorizing and prioritizing risks, we help you focus on the most impactful threats, ensuring efficient and effective remediation efforts.

SECURITY MATURITY REVIEW We conduct comprehensive security maturity reviews to evaluate your organization's current security practices and identify areas for improvement. Our assessments cover a wide range of criteria, including policy development, risk management, incident response, and security awareness. By benchmarking against industry standards and best practices, we provide actionable insights and recommendations to enhance your overall security posture and guide your organization toward achieving higher levels of security maturity.

SECURITY TEAM INCUBATION We provide comprehensive support for building security teams for new, large-scale IT ventures. From Day 1, our ramp-up program offers essential security advisory and assurance, helping you establish a robust security foundation. With our proven track record in securing billion-dollar investments and launching secure telco networks globally, we ensure your new enterprise is protected against cyberthreats from the start.

HACKING INCIDENT SUPPORT We offer immediate and comprehensive support in the event of a hacking incident, providing expert analysis, containment, and remediation. Our services include detailed forensics, malware analysis, and root cause determination, along with actionable recommendations to prevent future incidents. With our rapid response and deep expertise, we help you mitigate damage, recover swiftly, and strengthen your defenses against potential threats.