XMTP MLS Implementation Review

Ephemera
Version 1.0 – December 9, 2024

Prepared By
Kevin Henry
Marie-Sarah Lacharité
Eli Sohl

Prepared For
The XMTP Community

1 Executive Summary

Synopsis

In October 2024, Ephemera engaged NCC Group to perform a security assessment of `libxmtp`, their Rust implementation of the Extensible Message Transport Protocol (XMTP), built upon Messaging Layer Security (MLS) in a Web3 environment, where users leverage their existing blockchain-based identities for authentication. The application is underpinned by OpenMLS and provides a custom authentication service as described in XIP-46, which establishes a framework for associating multiple wallet addresses with a single self-managed identity. The review was performed by three consultants over the course of three weeks, with a total effort of 25 person-days. A retest was performed during the week of November 18, 2024, which found that 9 of 11 findings had been fixed. The remaining 2 findings are considered “Risk Accepted”, with updates to XIP-46 clarifying the design choices and responsibilities of an integrating application.

Scope

The XMTP MLS review was performed on a snapshot of the *xmtp/libxmtp* GitHub repository taken as of commit `b2df872` on pull request #1105. The review focused on the subdirectories *xmtp_mls/src* and *xmtp_id*, and the scope comprised these folders plus their local dependencies, minus any features currently under development and hidden behind feature flags. During the second week of the engagement, two additional commits (`2acfde8` and `814c006`) were added to the scope. These commits added support for remote / smart contract signature verifiers.

Limitations

The review targeted the *xmtp_id* and *xmtp_mls* subdirectories, and no claims of complete coverage outside of these subdirectories are made. Furthermore, XMTP heavily relies on OpenMLS, which was not included in this review.

Key Findings

Several low severity and informational findings were uncovered, along with the following medium-risk findings:

- Finding "Replay Detection Bypass via ECDSA Signature Malleability"
- Finding "Installation Keys Can Authorize Adding Associated Wallet Addresses"
- Finding "Insecure Use of Temporary Directory"

Additionally, a walkthrough of the MLS-related requirements that apply to `libxmtp` is given in appendix OpenMLS Application Requirements Review, and further notes and observations are documented in the appendix Engagement Notes.

A retest was performed on the week of November 18. This retest determined that of the 11 findings reported, 9 findings have been fixed and 2 findings are considered “Risk Accepted”, with additional rationale for the design choices added to XIP-46.

Strategic Recommendations

- Consider automating dependency management to maintain awareness of stale or vulnerable dependencies.
- Ensure that in-code documentation and outside documentation are kept up-to-date with implementation changes.
- Consider developing a “safe usage guide” outlining users’ responsibilities and discussing e.g. what validation steps `libxmtp` does and does not perform on input data.
- Consider getting third-party reviews of adjacent code, such as OpenMLS and the XMTP nodes.

2 Dashboard

Target Data

Name	XMTP MLS
Type	Shared library
Platforms	Native Rust library
Environment	Local

Engagement Data

Type	Implementation review
Dates	2024-10-02 to 2024-10-17
Consultants	3
Level of Effort	25 person-days

Finding Breakdown

Critical issues	0
High issues	0
Medium issues	3
Low issues	7
Informational issues	1
Total issues	**11**

Category Breakdown

Access Controls	1
Cryptography	4
Data Exposure	3
Denial of Service	1
Patching	1
Security Improvement Opportunity	1

Component Breakdown

libxmtp	2
xmtp_id	5
xmtp_mls	4

Critical High Medium Low Informational

3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Replay Detection Bypass via ECDSA Signature Malleability	Fixed	F6B	Medium
Installation Keys Can Authorize Adding Associated Wallet Addresses	Fixed	UQQ	Medium
Insecure Use of Temporary Directory	Fixed	LPK	Medium
Unsafe Concatenation of Data Leading to Inbox ID Collision	Fixed	HKH	Low
Recovery Address Change Does Not Require Signature from New Recovery Key	Risk Accepted	R42	Low
Revoke Association Action Does Not Recursively Revoke Associations	Risk Accepted	HVM	Low
Secrets Not Zeroized After Use	Fixed	A2V	Low
Potential Unhandled Panic When Decrypting History File	Fixed	TUM	Low
Mismatched Key Type Names May Introduce Confusion	Fixed	3A6	Low
Cryptographic Keys Written to Debug Logs	Fixed	63M	Low
Dependencies with Known RustSec Advisories	Fixed	JK9	Info

4 Finding Details

Medium

Replay Detection Bypass via ECDSA Signature Malleability

Overall Risk	Medium	**Finding ID**	NCC-E018021-F6B
Impact	Medium	**Component**	xmtp_id
Exploitability	Medium	**Category**	Cryptography
		Status	Fixed

Impact

Replay detection relies on raw ECDSA signatures and can therefore be bypassed due to malleability. An attacker can, for example, replay a variant of a valid `AddIdentity` message to add back a revoked identity without triggering replay detection, or to invalidate assumptions about the usage of legacy v2 keys.

Description

The reviewed XMTP implementation leverages XIP-46:

> This XIP defines a new identity model for XMTP, where users are represented in the protocol by an Inbox ID rather than a wallet address. Additionally, it defines mechanisms for associating multiple addressable identities, including wallets, to this new identity.

In short, XIP-46 defines a simple approach for creating an Inbox ID and associating one or more addresses / keys with this inbox, for which proof of possession of the corresponding private key is required via a signature created using said private key.

The approach defined in XIP-46 (and implemented in `libxmtp`) supports replay detection to, e.g., prevent an `AddIdentity` operation from being replayed in the future, which is achieved by tracking the provided signature that proves possession of the private key. Furthermore, the `xmtp_mls` README states:

> **Legacy V2 keys may only be used to create one association (globally)**
>
> We enforce this in two ways. Legacy V2 keys may only be used on an Inbox ID with nonce 0. Replay protection prevents the same Legacy V2 key from being used multiple times on that inbox ID.

The implementation of this replay detection is provided via the `IdentityAction` trait:

```
pub trait IdentityAction: Send + 'static {
    fn update_state(
        &self,
        existing_state: Option<AssociationState>,
        client_timestamp_ns: u64,
    ) -> Result<AssociationState, AssociationError>;
    fn signatures(&self) -> Vec<Vec<u8>>;
    fn replay_check(&self, state: &AssociationState) -> Result<(), AssociationError> {
        let signatures = self.signatures();
        for signature in signatures {
            if state.has_seen(&signature) {
                return Err(AssociationError::Replay);
```

```
            }
        }

        Ok(())
    }
}
```

Figure 1: xmtp_id/src/associations/association_log.rs

where `has_seen()` is defined as:

```
    pub fn has_seen(&self, signature: &Vec<u8>) -> bool {
        self.seen_signatures.contains(signature)
    }
```

Figure 2: xmtp_id/src/associations/state.rs

Here, as described in the standard, the received signatures for any update are checked against the list of previously seen signatures, leading to an error when detected.

The concern with the implemented approach is that ECDSA signatures are *malleable*, meaning that any individual can translate a valid ECDSA signature into a second valid ECDSA signature on the same message with a different value, *without knowledge of the private key*. In particular, an ECDSA signature has the form (r, s), but is mathematically equivalent to $(r, -s)$ during validation. For this reason, the usage of ECDSA signatures as a unique identifier is generally not recommended.

The `libxmtp` library implements a `VerifiedSignature` type to store a signature that has been validated, which leverages the `EthersSignature` type from the `ethers` crate as seen below:

```
    /**
     * Verifies an ECDSA signature against the provided signature text.
     * Returns a VerifiedSignature if the signature is valid, otherwise returns an error.
     */
    pub fn from_recoverable_ecdsa<Text: AsRef<str>>(
        signature_text: Text,
        signature_bytes: &[u8],
    ) -> Result<Self, SignatureError> {
        let signature = EthersSignature::try_from(signature_bytes)?;
        let address = h160addr_to_string(signature.recover(signature_text.as_ref())?);

        Ok(Self::new(
            MemberIdentifier::Address(address),
            SignatureKind::Erc191,
            signature_bytes.to_vec(),
        ))
    }
```

Figure 3: xmtp_id/src/associations/verified_signature.rs

However, the `ethers-core` implementation ***expects*** that the signature is in "low-s" format, but does not enforce this:

```
    /// Parses a raw signature which is expected to be 65 bytes long where
    /// the first 32 bytes is the `r` value, the second 32 bytes the `s` value
    /// and the final byte is the `v` value in 'Electrum' notation.
    fn try_from(bytes: &'a [u8]) -> Result<Self, Self::Error> {
        if bytes.len() != 65 {
            return Err(SignatureError::InvalidLength(bytes.len()))
```

```
        }

        let v = bytes[64];
        let r = U256::from_big_endian(&bytes[0..32]);
        let s = U256::from_big_endian(&bytes[32..64]);

        Ok(Signature { r, s, v: v.into() })
    }
}
```

As seen in `try_from()`, the only constraint applied is on the length of the input, not its value. Similarly, the `recover()` function makes the same assumption:

```
    /// Recovers the Ethereum address which was used to sign the given message.
    ///
    /// Recovery signature data uses 'Electrum' notation, this means the `v`
    /// value is expected to be either `27` or `28`.
    pub fn recover<M>(&self, message: M) -> Result<Address, SignatureError>
```

If ECDSA signatures are to be used as identifiers in this manner, it is critical that they be normalized into a canonical form, such as the "low-s" version in order to ensure that a unique representation is used.

Recommendation

Consider rejecting non-normalized signatures or converting all signatures to a canonical form when performing replay detection.

Location

- *xmtp_id/src/associations/association_log.rs*
- *xmtp_id/src/associations/verified_signature.rs*

Retest Results

2024-11-19 – Fixed

PR 1282 (merged in commit `a3be7dc`) introduced the function `to_lower_s()` to convert an ECDSA signature to its normalized "low-s" form and updated the `VerifiedSignature::from_recoverable_ecdsa()` function to always normalize signatures. This change is consistent with the recommendation to ensure that all signatures are in a canonical form, which addresses the replay bypass concerns in this finding. As such, this finding is considered "Fixed".

Medium

Installation Keys Can Authorize Adding Associated Wallet Addresses

Overall Risk	Medium	**Finding ID**	NCC-E018021-UQQ
Impact	Medium	**Component**	xmtp_id
Exploitability	Low	**Category**	Access Controls
		Status	Fixed

Impact

Users may protect their installation keys with less care due to an incorrect understanding of the effects of their potential compromise. Installation keys may also be a more attractive target to attackers, since users may underestimate the potential consequences of a compromised installation key.

Description

The identity model described in XIP-46: Multi-Wallet Identity allows each XMTP inbox to have multiple associated wallet addresses (secp256k1 keys) and installation keys (Curve25519 keys). Installation keys and associated wallet addresses are treated differently. In particular, the table of permissions in the Key hierarchy and permissions section of XIP-46 specifies that installation keys cannot add other installation keys. The same table also specifies that installation keys cannot add associated addresses to an inbox. Later in the document, another table, *Allowed associations*, indicates that an installation key *can* add associated wallet addresses to an inbox.

Adding installation keys or associated addresses to a wallet is done via the `AddAssociation` identity action in the XMTP implementation. This action requires two valid signatures: one from the new associated address or installation key (`new_member_signature`) and one from an existing member of the inbox (`existing_member_signature`). The new member's identifier (`new_member_identifier`, either an address or an installation public key) is also explicitly submitted as part of the action.

```rust
/// AddAssociation Action
#[derive(Debug, Clone)]
pub struct AddAssociation {
    pub new_member_signature: VerifiedSignature,
    pub new_member_identifier: MemberIdentifier,
    pub existing_member_signature: VerifiedSignature,
}
```

Figure 4: xmtp_id/src/associations/association_log.rs

The `update_state()` function for `AddAssociation` validates several aspects of the submitted action. It checks the following properties:

- The submitted `new_member_identifier` matches the signer of `new_member_signature`.
- The submitted `new_member_identifier` is different than the identifier of the `existing_member_signature` creator.
- The inbox ID was created with a nonce of 0 if either the `new_member_signature` or `existing_member_signature` is a legacy signature.

- The type of `new_member_signature` (e.g., smart contract wallet/ERC6492, delegated/ERC191, installation key/Ed25519) matches the type of `new_member_identifier`, and the type of `existing_member_signature` matches the type of the existing member.
- The `existing_member_signature` was indeed by an existing member of the inbox or its recovery address, and if it was by the recovery address, it was not a delegated legacy signature.
- The type of existing member is allowed to add the type of `new_member_identifier`.

The last property is relevant to this finding: according to the *Key hierarchy and permissions* table in XIP-46, an existing member that is linked to an installation key should *not* be able to add associated wallet addresses. However, the implementation of this check, in `allowed_association()`, does not prohibit this type of addition:

```
fn allowed_association(
    existing_member_kind: MemberKind,
    new_member_kind: MemberKind,
) -> Result<(), AssociationError> {
    // The only disallowed association is an installation adding an installation
    if existing_member_kind == MemberKind::Installation
        && new_member_kind == MemberKind::Installation
    {
        return Err(AssociationError::MemberNotAllowed(
            existing_member_kind,
            new_member_kind,
        ));
    }

    Ok(())
}
```

Figure 5: xmtp_id/src/associations/association_log.rs

As shown in the code snippet above, an installation key *is* allowed to add an associated address, contrary to what is stated in the *Key hierarchy and permissions* table in XIP-46.

Recommendation

- Determine the desired behavior and align the XIP-46 specification and implementation of `allowed_association()`.
- Ensure that the allowed associations described in XIP-46 are consistent, specifically in the *Key hierarchy and permissions* table and the *Allowed associations* table.

Location

xmtp_id/src/associations/association_log.rs

Retest Results

2024-11-27 – Fixed

As part of PR 73 (not yet merged at time of retest), XIP-46 was updated to specify that an installation key is authorized to add more associated addresses. In other words, the specification has been revised to match the implemented approach, and the two will be consistent once the PR is merged. As such, this finding is considered “Fixed”.

Medium

Insecure Use of Temporary Directory

Overall Risk	Medium	**Finding ID**	NCC-E018021-LPK
Impact	High	**Component**	xmtp_mls
Exploitability	Low	**Category**	Data Exposure
		Status	Fixed

Impact

A well-positioned attacker may be able to obtain a copy of a user's message history, overwrite files, or inject arbitrary groups or messages into the user's database.

Description

In *libxmtp/xmtp_mls/src/groups/message_history.rs*, message history bundles are read and written through subpaths of `std::env::temp_dir()`. However, the application and the running user are not guaranteed exclusive use of the `temp_dir()` folder. From the Rust docs:[1]

> The temporary directory may be shared among users, or between processes with different privileges; thus, the creation of any files or directories in the temporary directory must use a secure method to create a uniquely named file. Creating a file or directory with a fixed or predictable name may result in "insecure temporary file"[2] security vulnerabilities. Consider using a crate that securely creates temporary files or directories.

In fact, `libxmtp` does use predictable file names, as in the following excerpt:

```
    async fn write_history_bundle(&self) -> Result<(PathBuf, HistoryKeyType),
    ↪ MessageHistoryError> {
        let groups = self.prepare_groups_to_sync().await?;
        let messages = self.prepare_messages_to_sync().await?;

        let temp_file = std::env::temp_dir().join("history.jsonl.tmp");
        write_to_file(temp_file.as_path(), groups)?;
        write_to_file(temp_file.as_path(), messages)?;

        let history_file = std::env::temp_dir().join("history.jsonl.enc");
        let enc_key = HistoryKeyType::new_chacha20_poly1305_key();
        encrypt_history_file(
            temp_file.as_path(),
            history_file.as_path(),
            enc_key.as_bytes(),
        )?;

        std::fs::remove_file(temp_file.as_path())?;

        Ok((history_file, enc_key))
    }
```

Figure 6: xmtp_mls/src/groups/message_history.rs

1. https://doc.rust-lang.org/stable/std/env/fn.temp_dir.html
2. https://owasp.org/www-community/vulnerabilities/Insecure_Temporary_File

In this case, if a malicious application running were to create “history.jsonl.tmp” with open permissions, `libxmtp` would happily write a full list of groups and messages into that file. Note that the malicious application does not require any special permissions to do this.

Furthermore, if this file were created as a symlink, then `libxmtp` could be caused to write this data anywhere on the filesystem that it has permissions to write to. The exact impact in this case depends on the application’s runtime context and permissions, but it could easily lead to denial-of-service or worse.

A malicious application could also overwrite or inject data into temporary files that `libxmtp` reads back from disk. For instance, in `process_history_reply()`, group message history is decrypted and written to the temporary file `std::env::temp_dir().join("messages.jsonl")`. Then, it is read from this file and inserted into the database.

```
    pub async fn process_history_reply(&self) -> Result<(), MessageHistoryError> {
        let reply = self.get_latest_history_reply().await?;

        if let Some(reply) = reply {
            let Some(encryption_key) = reply.encryption_key.clone() else {
                return Err(MessageHistoryError::InvalidPayload);
            };

            let history_bundle = download_history_bundle(&reply.url).await?;
            let messages_path = std::env::temp_dir().join("messages.jsonl");

            decrypt_history_file(&history_bundle, &messages_path, encryption_key)?;

            self.insert_history_bundle(&messages_path)?;

            // clean up temporary files associated with the bundle
            std::fs::remove_file(history_bundle.as_path())?;
            std::fs::remove_file(messages_path.as_path())?;

            self.sync_welcomes().await?;

            let conn = self.store().conn()?;
            let groups = conn.find_groups(None, None, None, None)?;
            for crate::storage::group::StoredGroup { id, .. } in groups.into_iter() {
                let group = self.group(id)?;
                Box::pin(group.sync(self)).await?;
            }

            return Ok(());
        }

        Err(MessageHistoryError::NoReplyToProcess)
    }
```

Figure 7: xmtp_mls/src/groups/message_history.rs

A malicious applicaton could replace the “messages.jsonl” file with the content of its choice, thus injecting arbitrary messages or group data into the user’s database.

This issue is not likely to be exploitable in most deployment scenarios for `libxmtp`; however, in scenarios where it is exploitable, the impact would be high.

Recommendation

- Use a secure scheme for generating and cleaning up temporary files. Generate unpredictable filenames, and ensure that file permissions and ownership are appropriately restrictive. Consider pulling in a well-reviewed dependency to handle this.
- Avoid writing to disk unless necessary. Use authentication to ensure files on disk were not modified.

Location

libxmtp/xmtp_mls/src/groups/message_history.rs, lines 374, 486, 490, 638.

Retest Results

2024-11-19 – Partially Fixed

This finding was tracked in Issue 1186 and closed in PR 1152, which deletes *message_history.rs*, thereby negating this finding. However, PR 1174 subsequently re-added the file, thereby reintroducing the issue. At the time of retest, the current main branch commit (`a0c14de`) still includes *message_history.rs* and the affected code, although it is not included as part of the module tree. While the affected code is not currently in use, it remains present within the repository.

Based on the above, this finding is considered "Partially Fixed".

2024-11-21 – Fixed

Commit `0bea988` removed *message_history.rs*, thereby completing the fix.

Low

Unsafe Concatenation of Data Leading to Inbox ID Collision

Overall Risk	Low	**Finding ID**	NCC-E018021-HKH
Impact	Medium	**Component**	xmtp_id
Exploitability	Low	**Category**	Cryptography
		Status	Fixed

Impact

The method for creating an XIP-46 Inbox ID is not collision resistant, which may result in two distinct addresses being associated with the same Inbox ID.

Description

The XIP-46 specification uses an Inbox ID as the primary identifier for a user within XMTP, and provides a framework for associating multiple addresses with said Inbox ID.

> In this model, users of the protocol will be identified by Inbox IDs. An Inbox ID can be treated as an opaque string by applications but is constrained to the hash of the first associated address and a nonce.

More specifically, the Inbox ID is computed via a simple hash of the concatenation of the first address and a nonce:

> The inbox_id is derived via SHA256(CONCAT($account_address, $nonce))

The nonce appears to be a mechanism by which a single address may be associated with more than one XMTP inbox. The above is implemented in a straightforward manner:

```
fn sha256_string(input: String) -> String {
    let mut hasher = Sha256::new();
    hasher.update(input.as_bytes());
    let result = hasher.finalize();
    format!("{:x}", result)
}

pub fn generate_inbox_id(account_address: &str, nonce: &u64) -> String {
    sha256_string(format!("{}{}", account_address.to_lowercase(), nonce))
}
```

Figure 8: xmtp_id/src/associations/hashes.rs

It was observed that no length information is incorporated into this hash, which means it is trivial to produce multiple (address, nonce) pairs that produce the same Inbox ID. For example,

- `("abc123", 0)`
- `("abc12", 30)`
- `("abc1", 230)`
- `("abc", 1230)`

all produce the same Inbox ID according to the specified and implemented approach.

When concatenating variable length data for hashing, it is generally recommended to incorporate the length of each field as part of the hash input to avoid the above scenario. For example,

- `("6", "abc123", "1", "0")`
- `("5", "abc12", "2", "30")`
- `("4", "abc1", "3", "230")`
- `("3", "abc", "4", "1230")`

would each produce a different Inbox ID when concatenated and hashed. Similarly, encoding data using a standardized type-length-value (TLV) format would implicitly provide this approach. Alternatively, a different construction, such as that used in HMAC, would also avoid the issue, where the Inbox ID is calculated using `SHA256( SHA256(address) || SHA256(nonce) )`.

In practice, it may be expected that all addresses will be of a predictable, fixed length in the intended use cases, but the code itself does no additional checks on this front. Indeed, several test fixtures use short, odd-length address, e.g.:

```
    #[test]
    fn create_signatures() {
        let account_address = "0x123".to_string();
        let client_timestamp_ns: u64 = 12;
        let new_member_address = "0x456".to_string();
        let new_recovery_address = "0x789".to_string();
        let new_installation_id = vec![1, 2, 3];
        let create_inbox = UnsignedCreateInbox {
            nonce: 0,
            account_address: account_address.clone(),
        };
```

Figure 9: xmtp_id/src/associations/unsigned_actions.rs

As an additional consideration, hashing in contexts like the above usually incorporate a domain separation string to ensure that the resulting Inbox ID is specific to the implemented application. Prefixing the input with a string, such as “XIP46_INBOX_ID” would ensure that the resulting ID is unlikely to be independently derived in other contexts.

Finally, it was observed that the function `generate_inbox_id()` excerpted above canonicalizes the input address using `to_lowercase()`. This does not appear to be mandated in XIP-46, which suggests that there might be an undocumented requirement or gap in the specification.

Recommendation

- Consider including length information into the Inbox ID to avoid collisions.
- Alternatively, adopt an alternative collision-resistant hashing approach.
- Consider adding a domain separation string to the Inbox ID generation.
- Confirm that lowercase normalization is the intended approach and explicitly specify this as part of XIP-46.

Location

xmtp_id/src/associations/hashes.rs

Retest Results

2024-11-19 – Fixed

As part of PR 1202 (merged in commit `71b47a2`), the generation of the Inbox ID was updated to enforce a length of 42 on the account address via the function `is_valid_address()`:

```
/// Validates that the account address is exactly 42 characters, starts with "0x",
/// and contains only valid hex digits.
fn is_valid_address(account_address: &str) -> bool
```

This is accompanied by a new error type and handling for the case when an invalid address is encountered, with updated unit tests throughout as appropriate for both positive and negative test cases.

The implemented approach prevents ID collisions by enforcing a fixed length on the `account_address`, which is consistent with the recommendation to adopt an alternative collision-resistant hashing approach. Should a wider range of use cases be needed in the future, the recommendations to consider including length information and domain separation remain in effect. Nevertheless, at present the issue is mitigated within the current implementation and use cases, and as such this finding is considered “Fixed”.

Low

Recovery Address Change Does Not Require Signature from New Recovery Key

Overall Risk	Low	**Finding ID**	NCC-E018021-R42
Impact	High	**Component**	xmtp_id
Exploitability	Low	**Category**	Security Improvement Opportunity
		Status	Risk Accepted

Impact

Failure to validate proof of possession of the corresponding private key may allow an Inbox to be mistakenly transitioned into an unrecoverable state.

Description

There are four core `IdentityAction` state changes defined as part of XIP-46:

- `CreateInbox`
- `AddAssociation`
- `RevokeAssociation`
- `ChangeRecoveryAddress`

The `CreateInbox` and `AddAssociation` actions require a valid signature on any address associated with the inbox. Similarly, the `RevokeAssociation` action requires a valid signature from the recovery address. However, it was observed that the `ChangeRecoveryAddress` action *does not* require a signature from the new recovery address:

```
/// ChangeRecoveryAddress Action
#[derive(Debug, Clone)]
pub struct ChangeRecoveryAddress {
    pub recovery_address_signature: VerifiedSignature,
    pub new_recovery_address: String,
}
```

Figure 10: xmtp_id/src/associations/association_log.rs

There do not appear to be any enforced constraints on the `new_recovery_address`. In particular, there is no signature that proves the user will be able to sign with this key in the future. Such a signature is required for other actions, such as `AddAssociation`, where the new member information includes a `VerifiedSignature` proving the user can sign with the associated key:

```
/// AddAssociation Action
#[derive(Debug, Clone)]
pub struct AddAssociation {
    pub new_member_signature: VerifiedSignature,
    pub new_member_identifier: MemberIdentifier,
    pub existing_member_signature: VerifiedSignature,
}
```

Figure 11: xmtp_id/src/associations/association_log.rs

Note that the lack of signature using the new recovery address does not necessarily contradict any requirements in XIP-46, with the closest guidance being:

> There is a way to recover control over the inbox if any member other than the recovery address is compromised.

The Ephemera team confirmed that the implemented design is intentional and based on two considerations:

1. There is no mechanism to determine which inbox(es) a given recovery address is associated with, which restricts the impact of a user accidentally setting the recovery address to an unintended party's address.
2. It is desired that a user be able to delegate recovery to a third party without an interactive process to obtain a signature.

While the current approach may be intentional, it is nevertheless viewed as a potential "foot gun" for users who may accidentally transition their Inbox to an unrecoverable state. This is especially true when no additional validation is performed on the address. No such issue has been identified, but without any constraints on the `new_recovery_address`, it may be possible to magnify the impact of potential bugs elsewhere in the code, such as a serialization bug leading to a recovery address field being set to the empty string, and eventually being accepted as the intended new recovery address.

Recommendation

- Consider requiring a signature using the new recovery address/key.
- Otherwise, consider performing additional validation on the new recovery address to prevent accidental misuse.
- If the current behavior remains, ensure that applications give adequate guidance to users to prevent misuse or mistakes from rendering the account unrecoverable.

Location

xmtp_id/src/associations/association_log.rs

Retest Results

2024-11-27 – Partially Fixed

As part of PR 73 (not yet merged at time of retest), XIP-46 was updated with clarification as to the role of the recovery address:

> The recovery address is the only address that is allowed to revoke installations or wallets. Changing the recovery address does not require a signature from the new recovery address, allowing users to delegate recovery to a third party if desired. Recovery addresses are not used for reverse resolution (address -> inbox), so changing the recovery address of an inbox to an address that you do not control does not allow the user to impersonate any other address.

The above makes the intention of the design clear, and a risk-benefit analysis has led the chosen approach. The benefit of non-interactive third-party delegation is viewed as outweighing the risk of entering an unrecoverable state, particularly since such an outcome does not allow impersonation of a user. For the purposes of retesting, the documentation updates are seen as an appropriate, albeit partial fix, with a complete fix being dependent on the integrating application. Because the current behavior is an intentional design choice, this finding as a whole is classified as "Risk Accepted".

Low

Revoke Association Action Does Not Recursively Revoke Associations

Overall Risk	Low	**Finding ID**	NCC-E018021-HVM
Impact	Medium	**Component**	xmtp_id
Exploitability	Medium	**Category**	Cryptography
		Status	Risk Accepted

Impact

Revoking an associated address only removes direct children of the revoked address from the stored state. Any further associations added by the children of the revoked address are not necessarily removed as part of the revocation process and may still be able to sign on behalf of the user/Inbox ID.

Description

The `RevokeAssociation` action is signed by the recovery address and used to remove an association from the current association state. As part of this process, it is specified to:

> Remove any members in the Association State's member list that are both of type installation AND have their `added_by_member` field set to the `revoked_member`

This is implemented as part of `update_state()` for `RevokeAssociation`:

```
        let installations_to_remove: Vec<Member> = existing_state
            .members_by_parent(&self.revoked_member)
            .into_iter()
            // Only remove children if they are installations
            .filter(|child| child.kind() == MemberKind::Installation)
            .collect();

        // Actually apply the revocation to the parent
        let new_state = existing_state.remove(&self.revoked_member);

        Ok(installations_to_remove
            .iter()
            .fold(new_state, |state, installation| {
                state.remove(&installation.identifier)
            }))
```

Figure 12: xmtp_id/src/associations/association_log.rs

As specified, both the revoked association and its direct descendent installation keys in the association tree are revoked. However, the revoked address may have signed other `AddAssociation` actions for new wallet keys, which can in turn sign additional `AddAssociation` updates. These new keys will have an `added_by_member` value that is different from the aforementioned revoked address and will not be removed as part of the `RevokeAssociation` action.

The identity model presented in XIP-46 depicts a tree-based key hierarchy. Based on the description above, the `RevokeAssociation` action *does not* remove all associations in the subtree rooted at the revoked address. In other words, a compromised wallet can be revoked, along with the potentially compromised associations it certified, but any additional associations added by these malicious associations are not automatically revoked. This differs from similar revocation models, such as traditional Certificate Authorities (CAs), where revoking a CA certificate invalidates all certificates that include the revoked CA in their validation path.

XIP-46 does specify the following:

> When messaging a user by name, the implementing app will resolve from name to Inbox ID, beginning from the bottom of the tree and ending at the top.
>
> When a conversation participant is rendered in an app's UX, resolution from Inbox ID to name will begin from the top of the tree and end at the bottom.
>
> When multiple names are present, the implementing app must define a policy for how they will be rendered.

This suggests that apps will be required to build a path from a leaf to the root Inbox ID, which may throw an error if an intermediate node has been revoked. However, reliance on such behavior is not robust. In general, a user may expect that revoking an associated wallet address will revoke all associations that depend on the revoked wallet, not just its associated installation keys.

The Ephemera team indicated that the implemented revocation behavior is intentional, which is motivated by the fact that hierarchical model presented in XIP-46 does not necessarily match any relationship between wallets outside of XMTP, and revocation using XMTP's logical tree could potentially be unintuitive to a user. Furthermore, users are trusted to maintain knowledge of which wallets are under their control, and the revocation process can be repeated by the user for any unfamiliar or maliciously added associations. In other words, it was expressed there exist use cases in which recursive revocation is appropriate (as suggested in this finding), and there exist use cases where it is not (as highlighted by the Ephemera team).

Because XIP-46 models associations in a tree, and, as quoted above, uses these associations to construct identifiers, it may be beneficial to provide guidance on how applications should handle revoked associations on the path from a leaf to the root, as they will no longer be present in the association tree.

Recommendation

- Ensure the documented and implemented behavior is correct and ensure that users are given proper guidance on fully revoking an address/wallet.
- Consider revising the revocation process to revoke all associations in the subtree rooted at the revoked address.
- Otherwise, consider requiring that apps provide an interface for recursively revoking associations or pruning the association tree to ensure that all associations in the stored state have a valid path to the Inbox ID.
- Similarly consider adding additional guidance on how revoked associations should be presented to the user when resolving names.

Location

xmtp_id/src/associations/association_log.rs

Retest Results

2024-11-27 – Fixed

As part of PR 73 (not yet merged at time of retest), XIP-46 was updated with clarification on the revocation process:

> Applications building a revocation flow are encouraged to show the list of addresses and installations in a hierarchical form, and allow the user to choose to recursively revoke members that were added by the installation targeted for revocation. This protects against cases where a compromised installation or account may have added additional compromised members. This recursive revocation is not required by the protocol, with the exception of installations added directly by a revoked wallet, allowing users choice in how broadly they would like to revoke access.

The above is consistent with the stated recommendation of ensuring consistency between the documented and implemented behavior, as well as with the recommendation of providing additional guidance for integrating apps to guide a user during revocation. From the perspective of `libxmtp` and XIP-46, this finding is considered “Fixed”, however, the underlying security concerns remain. Therefore, from a design perspective, and to emphasize the risk to potential developers, this finding is being marked as “Risk Accepted” rather than “Fixed”.

Low

Secrets Not Zeroized After Use

Overall Risk	Low	**Finding ID**	NCC-E018021-A2V
Impact	High	**Component**	libxmtp
Exploitability	Low	**Category**	Data Exposure
		Status	Fixed

Impact

Failure to clear sensitive values from memory may allow these values to leak to other processes running in the same memory space. In the case of cryptographic keys or similar secrets, the confidentiality and authenticity of the underlying data may be completely compromised.

Description

In general, it is advised to make a proactive attempt to prevent memory related leakage of sensitive data by explicitly deleting it from memory prior to releasing said memory back to the operating system. While it can be difficult to ensure that an optimizing compiler will always ensure such deletions take place, many modern languages provide methods that allow a developer to express their intent to securely clear data.

In Rust, the `Zeroize` crate provides traits to "zeroize" a type, many of which can be automatically derived. Even if the attack surface is small, or memory related attacks are not considered in scope, it is still recommended to leverage `ZeroizeOnDrop` for private keys and similar data. Within the reviewed code, examples where zeroization is recommended include:

- `Authenticator` in *xmtp_api_grpc/src/auth_token.rs*,
- `HistoryKeyType` in *xmtp_mls/src/groups/message_history.rs*,
- `EncryptionKey` in *xmtp_mls/src/storage/encrypted_store/sqlcipher_connection.rs*,

along with any other private key wrappers and legacy private key uses within the codebase.

Recommendation

- Consider deriving the `Zeroize` and `ZeroizeOnDrop` traits as appropriate for any custom types storing sensitive data, such as private keys.
- Since users of `libxmtp` may include mobile apps, consider providing support for mobile OS's secure key storage (i.e. iOS Secure Enclave, Android KeyStore), which would remove the need to store keys or other sensitive data in memory.

Location

- *xmtp_api_grpc/src/auth_token.rs*
- *xmtp_mls/src/groups/message_history.rs*
- *xmtp_mls/src/storage/encrypted_store/sqlcipher_connection.rs*

Retest Results

2024-11-19 – Fixed

As part of PR 1230 (merged in commit `b68a702`) added the following, as recommended:

- `ZeroizeOnDrop` is derived for `Authenticator`.

- `ZeroizeOnDrop` is derived for `EncryptedConnection` (i.e., the parent struct containing `EncryptionKey`).
- `ZeroizeOnDrop` is derived for `HistoryKeyType`.

Furthermore, `zeroize.workspace = true` was added to *Cargo.toml* to ensure that any dependency supporting zeroization is correctly enabled.

Based on the above, this finding is considered “Fixed”.

Low

Potential Unhandled Panic When Decrypting History File

Overall Risk	Low	**Finding ID**	NCC-E018021-TUM
Impact	Low	**Component**	xmtp_mls
Exploitability	Low	**Category**	Denial of Service
		Status	Fixed

Impact

Unhandled panics can be leveraged by an attacker to cause the application to crash, thereby achieving a denial-of-service attack. This is particularly true when parsing data from an untrusted source where the attacker can influence inputs directly.

Description

This specific instance of an unhandled panic is highlighted because it may be triggered by untrusted input from the filesystem. A complete list of potential panics within the reviewed code was not completed.

The function `decrypt_history_file()`, as its name implies, is used to decrypt a message history file encrypted using AES-GCM, with the usual ciphertext consisting of the concatenation of a nonce, ciphertext, and authentication tag. The code parses the nonce as follows:

```
    // Read the messages file content
    let mut input_file = File::open(input_path)?;
    let mut buffer = Vec::new();
    input_file.read_to_end(&mut buffer)?;

    // Split the nonce and ciphertext
    let (nonce, ciphertext) = buffer.split_at(NONCE_SIZE);
```

Figure 13: xmtp_mls/src/groups/message_history.rs

The above code does not ensure that the read file is at least `NONCE_SIZE` bytes long and may therefore panic when calling `split_at(NONCE_SIZE)`. Given that this file is read from the file system, an attacker may therefore be able to crash the application via filesystem manipulation.

In general, panics should be avoided in situations that do not represent a truly unrecoverable state. When leveraged, panics should include useful information to the caller or user to enable debugging or troubleshooting of the underlying problem. In the above scenario, a check to ensure that the input is at least 12 (nonce) +16 (tag) = 28 bytes could be performed prior to parsing the nonce, with a suitable `Err` returned instead of a panic.

The above advice is consistent with the Secure Rust Guidelines:[3]

> Explicit error handling (`Result`) should always be preferred instead of calling panic. The cause of the error should be available, and generic errors should be avoided.

3. https://anssi-fr.github.io/rust-guide/04_language.html#panics

Recommendation

Consider either adding a length check and returning a suitable `Err` or adding an informative message to the panic to aid in debugging.

Location

xmtp_mls/src/groups/message_history.rs

Retest Results

2024-11-19 – Partially Fixed

This finding was tracked in Issue 1189 and closed in PR 1152, which deletes *message_history.rs*, thereby negating this finding. However, PR 1174 subsequently re-added the file, thereby reintroducing the issue. At the time of retest, the current main branch commit (`a0c14de`) still includes *message_history.rs* and the affected code, although it is not included as part of the module tree. While the affected code is not currently in use, it remains present within the repository.

Based on the above, this finding is considered "Partially Fixed".

2024-11-21 – Fixed

Commit `0bea988` removed *message_history.rs*, thereby completing the fix.

Low

Mismatched Key Type Names May Introduce Confusion

Overall Risk	Low	**Finding ID**	NCC-E018021-3A6
Impact	Low	**Component**	xmtp_mls
Exploitability	None	**Category**	Cryptography
		Status	Fixed

Impact

Variable names that directly contradict their use in the code may confuse developers leading to implementation errors or could affect the perceived security posture of the application. Such variables may also be evidence of incorrect assumptions, incomplete code refactoring, or actual bugs within the code.

Description

A message history file is encrypted using AES-GCM, as implemented in `encrypt_history_file()`:

```
fn encrypt_history_file(
    input_path: &Path,
    output_path: &Path,
    encryption_key: &[u8; ENC_KEY_SIZE],
) -> Result<(), MessageHistoryError> {
    // Read in the messages file content
    let mut input_file = File::open(input_path)?;
    let mut buffer = Vec::new();
    input_file.read_to_end(&mut buffer)?;

    let nonce = generate_nonce();

    // Create a cipher instance
    let cipher = Aes256Gcm::new(GenericArray::from_slice(encryption_key));
    let nonce_array = GenericArray::from_slice(&nonce);

    // Encrypt the file content
    let ciphertext = cipher.encrypt(nonce_array, buffer.as_ref())?;
```

Figure 14: xmtp_mls/src/groups/message_history.rs

However, when this function is called elsewhere in the code, a ChaCha20Poly1305 key is used:

```
        let enc_key = HistoryKeyType::new_chacha20_poly1305_key();
        encrypt_history_file(
            temp_file.as_path(),
            history_file.as_path(),
            enc_key.as_bytes(),
        )?;
```

Figure 15: xmtp_mls/src/groups/message_history.rs

Indeed, the only supported key type for `HistoryKeyType` is ChaCha20-Poly1305:

```
#[derive(Copy, Clone, Debug, PartialEq)]
pub(crate) enum HistoryKeyType {
    Chacha20Poly1305([u8; ENC_KEY_SIZE]),
}
```

Figure 16: xmtp_mls/src/groups/message_history.rs

The approach as implemented works correctly, as the underlying key is just a vector of 32 random bytes. However, the naming conventions are unclear and suggest a partial refactor or change in design has not been fully completed. For clarity, it is recommended to rename the unused ChaCha20-Poly1305 types to match the usage of AES-GCM. Alternatively, if support for both is required, the implementation should be updated with support for both algorithms based on the `HistoryKeyType` enum.

Similarly, it was observed that `encrypt_history_file()` expects the key as `encryption_key: &[u8; ENC_KEY_SIZE]`, whereas `decrypt_history_file()` expects `encryption_key: MessageHistoryKeyType`. In general, one would expect the types between these two functions to match. This is particularly important if multiple algorithms are supported, as algorithm confusion attacks may apply if different constraints are applied at encryption vs decryption. If support for both encryption algorithms is required, it should not be possible to mistakenly attempt decryption with the incorrect key type.

Recommendation

Review the highlighted code snippets and either:

- Rename the `HistoryKeyType` to correctly reflect its usage with AES-GCM, or
- Swap to ChaCha20-Poly1305 to correctly reflect the specified key type.

Also consider revising the parameters `encrypt_history_file()` to provide stronger type safety such that it matches `decrypt_history_file()`.

Location

xmtp_mls/src/groups/message_history.rs

Retest Results

2024-11-19 – Partially Fixed

This finding was tracked in Issue 1190 and closed in PR 1152, which deletes *message_history.rs*, thereby negating this finding. However, PR 1174 subsequently re-added the file, thereby reintroducing the issue. At the time of retest, the current main branch commit (`a0c14de`) still includes *message_history.rs* and the affected code, although it is not included as part of the module tree. While the affected code is not currently in use, it remains present within the repository.

Based on the above, this finding is considered "Partially Fixed".

2024-11-21 – Fixed

Commit `0bea988` removed *message_history.rs*, thereby completing the fix.

Low

Cryptographic Keys Written to Debug Logs

Overall Risk	Low	**Finding ID**	NCC-E018021-63M
Impact	High	**Component**	xmtp_mls
Exploitability	Low	**Category**	Data Exposure
		Status	Fixed

Impact

Sensitive information, such as cryptographic keys, should never be written to log files to avoid potential disclosure.

Description

Log files are generally considered to be less protected than most security-critical assets in an application. Therefore, unless a sufficient level of protection on log files is actively maintained, it is best practice to avoid logging any sensitive information to log files. Even if constrained to specific instances, such as debug configurations, there remains a risk of a user or maintainer accidentally using a build with a debug flag active.

Within the SQL KeyStore, the `write_encryption_epoch_key_pairs()` function is used to write the HPKE keys for the epoch to the store:

```
    fn write_encryption_epoch_key_pairs<
        GroupId: traits::GroupId<CURRENT_VERSION>,
        EpochKey: traits::EpochKey<CURRENT_VERSION>,
        HpkeKeyPair: traits::HpkeKeyPair<CURRENT_VERSION>,
    >(
        &self,
        group_id: &GroupId,
        epoch: &EpochKey,
        leaf_index: u32,
        key_pairs: &[HpkeKeyPair],
    ) -> Result<(), Self::Error> {
        let key = epoch_key_pairs_id(group_id, epoch, leaf_index)?;
        let value = bincode::serialize(key_pairs)?;
        tracing::debug!("Writing encryption epoch key pairs");
        tracing::debug!("  key: {}", hex::encode(&key));
        tracing::debug!("  value: {}", hex::encode(&value));

        self.write::<CURRENT_VERSION>(EPOCH_KEY_PAIRS_LABEL, &key, &value)
    }
```

Figure 17: xmtp_mls/src/storage/sql_key_store.rs

As highlighted, the value `epoch` contains the `EpochKey` and the array `key_pairs` contains HPKE key pairs. Both of these cryptographic secrets are serialized and used in the computation `key` and `value`, both of which are logged using `tracing::debug!()`. This behavior is inconsistent with other functions in the file (and the rest of the codebase), which suggests that the highlighted log messages may be unintentional, or a remnant from the development process instead of necessary debug information.

To avoid potential leakage of the epoch key or HPKE keys via debug logs, it is recommended to remove or sanitize the above log statements such that they are safe to be made public.

Recommendation

Remove the `key` and `value` log outputs in `write_encryption_epoch_key_pairs()`.

Location

xmtp_mls/src/storage/sql_key_store.rs

Retest Results

2024-11-19 – Fixed

PR 1231 (merged in commit `56fef15`) removed the highlighted debug entries, thereby addressing this finding. As such, this finding is considered “Fixed”.

Info

Dependencies with Known RustSec Advisories

Overall Risk	Informational	**Finding ID**	NCC-E018021-JK9
Impact	Undetermined	**Component**	libxmtp
Exploitability	Undetermined	**Category**	Patching
		Status	Fixed

Impact

Vulnerabilities in third-party dependencies may be inherited by the application. Even if known vulnerabilities do not apply, the presence of vulnerable dependencies may still affect the perceived security posture of the application and its maintainers.

Description

The Rust ecosystem provides several tools for managing dependencies, such as `cargo audit` for identifying known security issues, `cargo outdated` for identifying stale dependencies, and `cargo deny` for blocking or allowing various vulnerable or outdated dependencies.

The `cargo audit` tool identifies two vulnerable dependencies:

- `diesel 2.2.2` - "Binary Protocol Misinterpretation caused by Truncating or Overflowing Casts"
- `tonic 0.12.2` - "Remotely exploitable Denial of Service in Tonic"

along with 3 dependencies with warnings:

- `ansi_term 0.12.1` - Unmaintained
- `dirs 5.0.1` - Unmaintained
- `futures-util 0.3.30` - Yanked.

In general, it is recommended to actively address or update any applicable RustSec advisories that affect the application. The `cargo deny` tool can be used to automatically fail builds if a vulnerable crate is detected, and also supports a list of exceptions so that the inclusion of such a package can be explicitly reviewed and annotated. Additionally, GitHub's Dependabot Service can be configured to scan for and open issues or PRs when updated dependencies are found.

It is emphasized that this review is a point-in-time evaluation of an evolving project, and the presence of outdated dependencies is expected. The affecting advisories for vulnerable crates are recent, and do not appear indicative of any deeper issue within the project.

Recommendation

Consider one or more of the following:

- Adoption of `cargo deny` to automatically detect new RustSec advisories.
- Adoption of Dependabot to automatically update dependencies.
- Ensure that dependencies are reviewed, updated and tested as part of any documented release ceremonies.

Location

Cargo.toml

Retest Results

2024-11-19 – Fixed

As of commit `402c91f`, Dependabot has been configured for the `libxmtp` repository.

As of commit `f64a31c`, a `cargo deny` configuration has been added to the project.

At the time of retest, `cargo audit` reports no vulnerabilities and two warnings that `dirs` and `instant` are unmaintained. As all recommendations were followed, this finding is considered "Fixed".

5 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

6 OpenMLS Application Requirements Review

OpenMLS provides guidance on which requirements from RFC 9420 are the responsibility of an application leveraging OpenMLS. These requirements are detailed in https://github.com/openmls/openmls/blob/main/book/src/app_validation.md and briefly surveyed here against their implementation in `libxmtp`.

Of the requirements highlighted by OpenMLS, it was generally found that `libxmtp` satisfies these requirements, with the following exceptions:

- The following findings affect validation requirements on acceptable identifiers:
  - Finding "Unsafe Concatenation of Data Leading to Inbox ID Collision"
  - Finding "Replay Detection Bypass via ECDSA Signature Malleability"
  - Finding "Installation Keys Can Authorize Adding Associated Wallet Addresses"
- XMTP *does not* enforce a maximum acceptable total lifetime for leaf nodes.

Furthermore, it was also found that `libxmtp` does not support the "mandatory to implement" ciphersuite specified in RFC 9420. While this appears to be an intentional choice, it nevertheless prevents formal compliance with the RFC.

Retest Update
During retesting, the 3 findings listed above were considered "Fixed". The missing mandatory to implement ciphersuite and unenforced maximum lifetimes also remain as intentional design choices at the time of retest and are therefore considered "Risk Accepted".

Acceptable Presented Identifiers
Per Section 5.3.1:

> The application using MLS is responsible for specifying which identifiers it finds acceptable for each member in a group. In other words, following the model that [RFC6125] describes for TLS, the application maintains a list of "reference identifiers" for the members of a group, and the credentials provide "presented identifiers". A member of a group is authenticated by first validating that the member's credential legitimately represents some presented identifiers, and then ensuring that the reference identifiers for the member are authenticated by those presented identifiers

This is the fundamental purpose of XIP-46 and its implementation in `xmtp_id`, which provides a single Inbox ID (e.g., the "reference identifier") and a mechanism for binding various address or installation keys (e.g., the "presented identifiers") to a given Inbox ID. Concerns around the uniqueness of Inbox IDs are documented in finding "Unsafe Concatenation of Data Leading to Inbox ID Collision". Provided this finding is addressed, or that Inbox IDs are always computed from fixed length identifiers, this requirement appears to be satisfied.

Validity of Updated Presented Identifiers
Per Section 5.3.1:

> In cases where a member's credential is being replaced, such as the Update and Commit cases above, the AS MUST also verify that the set of presented identifiers in the new credential is valid as a successor to the set of presented identifiers in the old credential, according to the application's policy.

The `AddAssocation` action is used to update the set of presented identifiers in `xmtp_id`. The validations performed as part of executing this action are defined in XIP-46; they ensure that the signatures are appropriate, that the update has not been processed previously, that legacy keys are handled appropriately, and that the leveraged signing key has the authority to perform the action. The review revealed two findings which may affect the correctness of this process:

- Finding "Replay Detection Bypass via ECDSA Signature Malleability" may allow an `AddAssociation` message to be replayed in certain circumstances.
- Finding "Installation Keys Can Authorize Adding Associated Wallet Addresses" identified discrepancies between the reviewed `xmtp_id` and XIP-46 regarding which keys may authorize which actions.

Conditional upon these two findings being addressed, this requirement appears to be satisfied.

Application ID is Not Authenticated by AS

Per Section 5.3.3:

> However, applications MUST NOT rely on the data in an application_id extension as if it were authenticated by the Authentication Service, and SHOULD gracefully handle cases where the identifier presented is not unique.

The `application_id` extension referenced here is set as part of the function `new_key_package()`, where it takes the value `inbox_id`, which is computed as a SHA-256 hash of the concatenation of the account address and a nonce. This is enforced when creating a new `Identity` instance:

```rust
impl Identity {
    /// Create a new [Identity] instance.
    ///
    /// If the address is already associated with an inbox_id, the existing inbox_id will be used.
    /// Users will be required to sign with their wallet, and the legacy is ignored even if it's provided.
    ///
    /// If the address is NOT associated with an inbox_id, a new inbox_id will be generated.
    /// If a legacy key is provided, it will be used to sign the identity update and no wallet signature is needed.
    ///
    /// If no legacy key is provided, a wallet signature is always required.
    pub(crate) async fn new<ApiClient: XmtpApi>(
```

Figure 18: xmtp_mls/src/identity.rs

Finding "Unsafe Concatenation of Data Leading to Inbox ID Collision" highlighted some concerns with the creation of an Inbox ID, namely that collisions in the output are possible if inputs are not fixed length, and that no domain separation is applied. Therefore, it appears as though the implementation is intended to provide some measure of guarantee that the Inbox ID / Application ID is both unique and authenticated, but it may not always do so unless this finding is addressed. Provided this finding is fixed, or that all inputs used in the Inbox ID calculation are of fixed length, then this requirement appears to be satisfied.

Specifying the Maximum Total Acceptable Lifetime

Per Section 7.2:

> Applications MUST define a maximum total lifetime that is acceptable for a LeafNode, and reject any LeafNode where the total lifetime is longer than this duration. In order to avoid disagreements about whether a LeafNode has a valid lifetime, the clients in a group SHOULD maintain time synchronization (e.g., using the Network Time Protocol [RFC5905]).

It appears as though XMTP explicitly does not enforce lifetimes on a `LeafNode`, as a deliberate design choice:

```
async fn validate_inbox_id_key_package(
    key_package: Vec<u8>,
) -> Result<ValidateInboxIdKeyPackageResponse, ValidateInboxIdKeyPackageError> {
    let rust_crypto = RustCrypto::default();
    let kp = VerifiedKeyPackageV2::from_bytes(&rust_crypto, key_package.as_slice())?;

    Ok(ValidateInboxIdKeyPackageResponse {
        is_ok: true,
        error_message: "".into(),
        credential: Some(kp.credential),
        installation_public_key: kp.installation_public_key,
        // We are deprecating the expiration field and key package lifetimes, so stop checking for its existence
        expiration: 0,
    })
}
```

Figure 19: mls_validation_service/src/handlers.rs

The above comment was added and related checks were removed as part of PR #962. As such, the implied lifetime is *infinite*, which is likely to be seen as contradicting the above requirement. However, if all clients agree on this lifetime, then the stated goal of avoiding disagreements is also mitigated. Regardless, by a strict interpretation of the specification this requirement does not appear to be satisfied.

Structure of AAD is Application-Defined

Per Section 6.3.1:

> It is up to the application to decide what authenticated_data to provide and how much padding to add to a given message (if any). The overall size of the AAD and ciphertext MUST fit within the limits established for the group's AEAD algorithm in [CFRG-AEAD-LIMITS].

No instances of Additional Authenticated Data (AAD) were observed in the reviewed code for XMTP v3. The only observed usage of AAD is in the function `encrypt()`, for XMTP v2:

```
pub fn encrypt(
    plaintext_bytes: &[u8],
    secret_bytes: &[u8],
    additional_data: Option<&[u8]>,
) -> Result<Ciphertext, String> {
    // Form a Payload struct from plaintext_bytes and additional_data if it's present
    let mut payload = Payload::from(plaintext_bytes);
    if let Some(aad_data) = additional_data {
```

```
        payload.aad = aad_data;
    }
    encrypt_raw(payload, secret_bytes)
}
```

Figure 20: xmtp_v2/src/encryption.rs

The above leverages the RustCrypto `Aes256Gcm` implementation, which enforces a maximum length on the associated data of `1 << 36`, which is within the defined limits for AES-GCM. Therefore, this requirement appears to be satisfied. Note that this usage of AAD occurs within the XMTP v2 code, which was not in scope for this review.

Proposal Validation

Per the OpenMLS App Validation Guide:

> When processing a commit, the application has to ensure that the application specific semantic checks for the validity of the committed proposals are performed.
>
> This should be done on the StagedCommit.

The `libxmtp` *README* outlines the validation steps required within XMTP. These steps are excerpted and paraphrased below:

1. Ensure the commit is allowed according to the permissions policies on the group.
2. Validate the credentials and key packages of any new members to the group.
   - New clients are expected to upload a Key Package to the network signed by their installation public key.
   - Additionally validate that the installation key is associated with the `inbox_id` referenced in the Key Package's credential. This validation is performed by downloading the latest identity updates for the `inbox_id` and ensuring that the installation key is present in the list of associated keys.
   - Clients are expected to regularly rotate their key package to limit the impact if the HPKE keypair referenced in the key package is compromised. This rotation is expected to happen any time the client receives a new welcome message.
3. Ensure that the actual change in MLS group members matches the expected change in membership found by diffing the previous GroupMembership struct and the new GroupMembership.

The `ValidatedCommit` struct encapsulates a commit that has passed validation, which also specifies a more concrete set of validation criteria than the above:

```
/**
 * A [`ValidatedCommit`] is a summary of changes coming from a MLS commit, after all of our validation rules have been applied
 *
 * Commit Validation Rules:
 * 1. If the `sequence_id` for an inbox has changed, it can only increase
 * 2. The client must create an expected diff of installations added and removed based on the difference between the current
 *      [`GroupMembership`] and the [`GroupMembership`] found in the [`StagedCommit`]
 * 3. Installations may only be added or removed in the commit if they were added/removed in the expected diff
 * 4. For updates (either updating a path or via an Update Proposal) clients must verify that the `installation_id` is
 *      present in the [`AssociationState`] for the `inbox_id` presented in the credential at the `to_sequence_id` found in the
```

```
 *      new [`GroupMembership`].
 * 5. All proposals in a commit must come from the same installation
 * 6. No PSK proposals will be allowed
 * 7. New installations may be missing from the commit but still be present in the
 ↪ expected diff.
 */
#[derive(Debug, Clone)]
pub struct ValidatedCommit {
```

Figure 21: xmtp_mls/src/groups/validated_commit.rs

In addition to enforcing the 7 stated validation rules, the `ValidatedCommit::from_staged_commit()` function also enforces that the permissions are consistent with the permission policies using the framework in *xmtp_mls/src/groups/group_permissions.rs*. This framework allows for an action to be limited by any user, a group admin, or a group super admin, and also allows policies to be composed using "any" or "and" clauses.

Similarly, credentials associated with the installation proposing the commit and all other associations updated with the commit are verified.

Regarding key rotation, it was confirmed that keys are rotated when new Welcome messages are received as part of `sync_welcomes()`, satisfying the documented requirement.

Based on the above, the validation criteria expected and enforced by `libxmtp` appears to be clearly documented and correctly enforced, thereby satisfying this requirement. It may be beneficial to ensure the *README* and code annotations are both complete and consistent such that they specify an identical set of constraints.

External Commits

Per Section 12.2:

> At most one Remove proposal, with which the joiner removes an old version of themselves. If a Remove proposal is present, then the LeafNode in the path field of the external Commit MUST meet the same criteria as would the LeafNode in an Update for the removed leaf (see Section 12.1.2). In particular, the credential in the LeafNode MUST present a set of identifiers that is acceptable to the application for the removed participant.

Currently, `libxmtp` does not support external commits and will return an error if an actor referenced in a commit is not a member of the group. Therefore, this requirement is currently not in scope.

The existing function `get_proposal_changes()` tracks which nodes are updated, added, or removed in a given proposal, as well as returning a list of credentials which require validation for the proposal to succeed. In the current implementation, the only credentials that are directly verified are those of type `Proposal::Update`. There does not appear to be any credential validation performed when the type is `Proposal::Remove`. This is consistent with comments in the function `ValidatedCommit::from_staged_commit()` which explicitly validates the credentials of the actor who created the commit and anyone referenced in an update query. Additional cases would need to be added here when support for external commits is added.

Additional Comments

It was also observed that XMTP does not support the Mandatory to Implement (MTI) ciphersuite specified in Section 17.1:

> The mandatory-to-implement cipher suite for MLS 1.0 is MLS_128_DHKEMX25519_AES128GCM_SHA256_Ed25519, which uses Curve25519 for key exchange, AES-128-GCM for HPKE, HKDF over SHA2-256, and Ed25519 for signatures. MLS clients MUST implement this cipher suite.

While this option is supported in OpenMLS, it does not appear to be exposed by `xmtp_mls`. As such, the library, as written, cannot claim strict compliance with RFC 9420.

7 Engagement Notes

This section captures various notes collected by NCC Group Cryptography Services' consultants over the course of the review. These notes are not considered to rise to the level of findings *per se*, but are nevertheless judged to be of potential interest. They are roughly in decreasing order of importance.

Panics in Functions Returning `Result`s

Rust has multiple idioms for handling failure states. In cases of immediate and catastrophic failure, code may throw a panic, instantly aborting execution. More commonly, code which has the potential to fail may return a Result, indicating either success or failure. Results allow the calling code to make its own decisions about how failures will be handled, and to ensure that cleanup tasks are performed before exiting. As such, it is considered an anti-pattern to throw a panic within a function if that function returns a Result. This means that functions returning Results should not call `.unwrap()` or `.expect()`; nevertheless, this pattern was noted to occur several times throughout `libxmtp`:

- *xmtp_mls/src/api/identity.rs:146*
- *xmtp_mls/src/api/mls.rs:99,143*
- *xmtp_mls/src/bin/update-schema.rs:43,76,77-83,91*
- *xmtp_mls/src/groups/group_permissions.rs:83*
- *xmtp_mls/src/groups/sync.rs:338*
- *xmtp_mls/src/identity.rs:413*
- *xmtp_mls/src/retry.rs:305*

Some of these cases are innocuous and unexploitable; nevertheless, it is noted that any case where any of these panics can be triggered from remote input would constitute a powerful remote denial-of-service attack. This alone should strongly motivate replacing these panics with failing Results.

In fact, some recommend going further and eliminating panics entirely; for instance, ANSSI's secure Rust coding guidelines include the following recommendation:[4]

> Explicit error handling (Result) should always be preferred instead of calling panic. The cause of the error should be available, and generic errors should be avoided.
>
> Crates providing libraries should never use functions or instructions that can fail and cause the code to panic.

Permission Management in Dockerfiles

Three Dockerfiles are present within the `libxmtp` repository:

- *libxmtp/Dockerfile*
- *libxmtp/dev/validation_service/local.Dockerfile*
- *libxmtp/dev/validation_service/Dockerfile*

It is considered a best practice, following the principle of least privilege, to run Docker processes as non-root users. Similarly, the user should not have sudo permissions. An attacker able to escape the application would then operate in a low-privilege context rather than as root. This would present obstacles to attacks that require user-to-kernel interactions, which require a privileged account.

4. https://anssi-fr.github.io/rust-guide/04_language.html#panics

However, none of the listed Dockerfiles include a non-root `USER` directive. In the first listed Dockerfile it is further observed that `sudo` is invoked by the user. Depending on what these Dockerfiles are used for, rewriting them to follow best practices may be advisable.

Notes on XIP-46 Specification and Other Documentation

This subsection captures some comments about minor inconsistencies in documentation, such as in XIP-46: Multi-Wallet Identity.

- **Recovery address as a member role.** XIP-46 describes the three roles that members of an XMTP inbox may have: associated address, installation key, or recovery address. Certain statements are made about all member roles that do not actually apply to recovery addresses. These statements should be amended to include the exception of recovery addresses, or possibly the terminology in the XIP should be changed to specify that “member” excludes the recovery address. (In the XMTP implementation, `MemberKind` is defined as one of `Installation` or `Address` in *xmtp_id/src/associations/member.rs*.)
    - The text says

        > The member list of an inbox is expected to have the following properties:
        >
        > 1. Every added member was bidirectionally approved by an existing member and the newly added member.
        > 2. ...

        Property 1 is not true for recovery addresses, as described in finding "Recovery Address Change Does Not Require Signature from New Recovery Key".
    - The excerpted proto file says

        ```
        // A key-pair that has been associated with one role MUST not be permitted to be
        // associated with a different role.
        ```

        Technically, the recovery address could *also* be a (non-recovery) member of the inbox (either installation key or associated address).
- **Effect of truncated inbox log.** XIP-46 says

    > The member list of an inbox is expected to have the following properties:
    >
    > 1. Every added member was bidirectionally approved by an existing member and the newly added member.
    > 2. ...
    > 3. Any client can verify that (1) is true, and all clients should see the same member list.

    Additionally, the XMTP documentation at https://docs.xmtp.org/protocol/v3/identity says

    > XMTP maintains an inbox log. The inbox log has a list of all identity actions affecting the inbox. The inbox log can track 256 identity actions. Since identity actions can be combined, this can be more than 256 associations, removals, change of recovery wallets, etc.

    By surpassing this limit, there may legitimately be group members who were not added by any members present in the identity log.

Missing Optional Ethereum Address Checksum Validation

The function `is_valid_ethereum_address()` performs simple validation on an input to ensure it looks like a valid Ethereum address:

```
/// Check if an string is a valid ethereum address (valid hex and length 20).
pub fn is_valid_ethereum_address<S: AsRef<str>>(address: S) -> bool {
    let address = address.as_ref();
    let address = address.strip_prefix("0x").unwrap_or(address);

    if address.len() != 40 {
        return false;
    }

    address.chars().all(|c| c.is_ascii_hexdigit())
}
```

Figure 22: xmtp_cryptography/src/signature.rs

In other words, the function ensures that the address consists of exactly 40 case insensitive hex digits.

As described in ERC-55, Ethereum supports checksum addresses, where the case of the letters in the hex notation of the address is used to encode a checksum. Such addresses are easily recognized via their use of mixed case and support an additional layer of validation. The function above could be updated to support checksum validation, which could potentially detect more invalid addresses than are currently detected. Such an improvement does not appear to prevent any particular attack, but could lead to earlier error detection, which is generally preferable when possible.

The above function is used as part of `sanitize_evm_addresses()`, which validates a list of addresses, usually during deserialization. This function converts the resulting validated addresses to lowercase as its final step, which suggests that the application is expecting to receive addresses that are not strictly lowercase.

Redundant Code

- **Computation of Inbox ID.** The following code in `Identity::new()` unnecessarily recomputes the Inbox ID after confirming it matches the expected value:

```
346             if inbox_id != generate_inbox_id(&address, &nonce) {
347                 return Err(IdentityError::NewIdentity(
348                     "Inbox ID doesn't match nonce & address".to_string(),
349                 ));
350             }
351             let inbox_id = generate_inbox_id(&address, &nonce);
```

Figure 23: xmtp_mls/src/identity.rs

  The overhead involved in this is trivial, but removing the redundant generation on line 351 would not alter the behavior of the function. Note that the same code pattern – without the redundant generation of the Inbox ID – appears on line 294 of the same function.

- **Selection of OpenMLS cryptography provider.** In `decrypt_welcome()`, there is a call to the OpenMLS function `decrypt_with_label()`, which takes an `OpenMlsCrypto` argument indicating which cryptographic provider to use. Despite `decrypt_welcome()` having function argument `provider` (of type `XmtpOpenMlsProvider`), which could supply an `OpenMlsCrypto` with `provider.crypto()`, this is re-derived as `crypto =`

`RustCrypto::default();`. It is recommended to use the function argument in case it changes in the future.

```
/// Decrypt a welcome message using the private key associated with the provided public key
pub fn decrypt_welcome(
    provider: &XmtpOpenMlsProvider,
    hpke_public_key: &[u8],
    ciphertext: &[u8],
) -> Result<Vec<u8>, HpkeError> {
    // SNIP
            return Ok(decrypt_with_label(
                kp.init_private_key(),
                WELCOME_HPKE_LABEL,
                &[],
                &ciphertext,
                CIPHERSUITE,
                &RustCrypto::default(),
            )?);
```

Figure 24: xmtp_mls/src/hpke.rs

Incorrect Algorithm Identifiers (OpenMLS)

The following documentation issue was identified within the OpenMLS *README*:

```
## Supported ciphersuites

- MLS_128_HPKEX25519_AES128GCM_SHA256_Ed25519 (MTI)
- MLS_128_DHKEMP256_AES128GCM_SHA256_P256
- MLS_128_HPKEX25519_CHACHA20POLY1305_SHA256_Ed25519
```

The highlighted algorithms are not listed in RFC 9420. The highlighted algorithms should likely be updated to `DHKEMX25519`. The reviewed `libxmtp` uses the correct ciphersuite identifier of `MLS_128_DHKEMX25519_CHACHA20POLY1305_SHA256_Ed25519`.