Secure Code Review of FROST

Technical Report

Lit Protocol

21 June 2024
Version: 1.1

Kudelski Security – Nagravision Sàrl

Corporate Headquarters
Kudelski Security – Nagravision Sàrl
Route de Genève, 22-24
1033 Cheseaux sur Lausanne
Switzerland

For Public Release

DOCUMENT PROPERTIES

Version:	1.1
File Name:	Kudelski_Security_Lit_Protocol_Secure_Code_Review_of_FROST_v1.1_Final.pdf
Publication Date:	21 June 2024
Confidentiality Level:	For Public Release
Document Status:	Approved

TABLE OF CONTENTS

EXECUTIVE SUMMARY

Lit Protocol ("the Client") engaged Kudelski Security ("Kudelski", "we") to perform a Secure Code Review of FROST.

The assessment was conducted remotely by the Kudelski Security Team.

The review took place between 06 May 2024 and 11 June 2024, and focused on the following objectives:

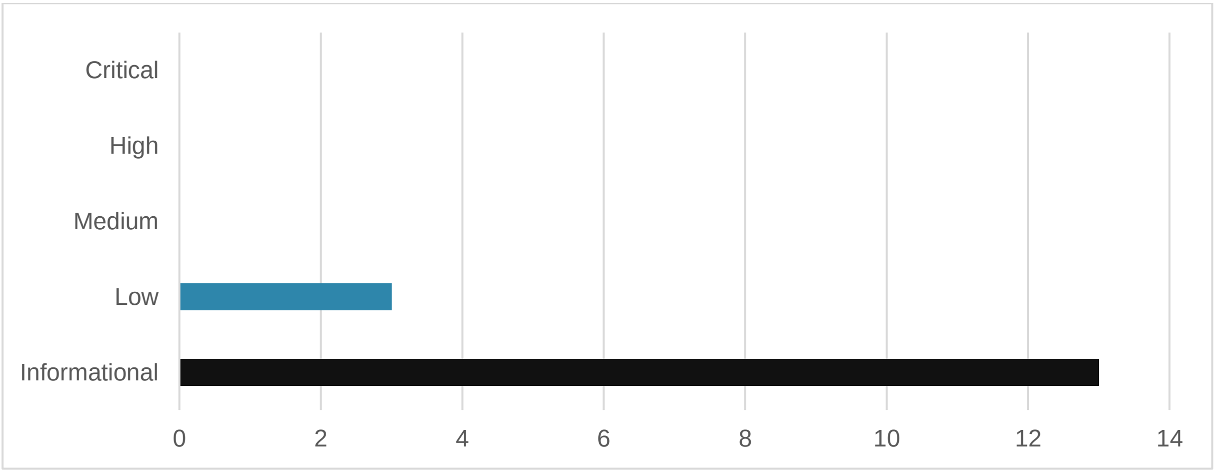
- Provide the customer with an assessment of their overall security posture and any risks that were discovered.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
- To identify potential issues and include improvement recommendations based on the result of our tests.

Key Findings

The following are the major themes and issues identified during the testing period.

These, along with other items within the findings section, should be prioritized for remediation to reduce to the risk they pose.

- Lack of zeroization for custom types.
- Incorrectly implemented mitigation of timing attacks.



Findings ranked by severity.

1. PROJECT SUMMARY

This report summarizes the engagement and contains detailed descriptions of the discovered vulnerabilities, steps the Kudelski Security Team took to identify and validate each issue, as well as any applicable recommendations for remediation.

1.1 Context

The source code in scope is an implementation of FROST, a Schnorr-based threshold signature scheme.

1.2 Scope

The scope consisted in specific Rust files and folders located at:

- Commit hash: `45bcdc3cc97eb4d7ee600986ef983cfae2b781b2`
- Source code repository: `lit-frost` (`https://github.com/LIT-Protocol/lit-frost/tree/45bcdc3cc97eb4d7ee600986ef983cfae2b781b2`)

The files and folders in scope are the Rust (`.rs`) files located in the `src/` folder of the above repository.

1.3 Follow-up

After the initial report, Lit Protocol addressed the vulnerabilities and weaknesses in the following codebase revision:

- Commit hash: `e3550520ac4569e44b83c4f56d09449b653f2074`

1.4 Remarks

During the code review, the following positive observations were noted regarding the scope of the engagement:

- The source code is nicely structured and defensively coded.
- Tests were also provided as part of the project, which is convenient for better understanding how the library works and useful for elaborating scenarios and validating findings.
- Finally, we had regular and very enriching technical exchanges on various topics.

1.5 Additional Note

It is important to notice that, although we did our best in our analysis, no code audit assessment is per se guarantee of absence of vulnerabilities. Our effort was constrained by resource and time limits, along with the scope of the agreement.

In assessing the severity of some of the findings we identified, we kept in mind both the ease of exploitability and the potential damage caused by an exploit.

While assessing the severity of the findings, we considered the impact, ease of exploitability, and the probability of attack. This is a solid baseline for severity determination. Information about the severity ratings can be found in Chapter Vulnerability Scoring System of this document.

2. TECHNICAL DETAILS OF SECURITY FINDINGS

This chapter provides detailed information on each of the findings, including methods of discovery, explanations of severity determination, recommendations, and applicable references. The following table provides an overview of the findings.

#	SEVERITY	TITLE	STATUS
KS–LPF–F–1	Low	Possible Timing Attack in `is_identity` Macro	Resolved
KS–LPF–F–2	Low	Missing Zeroization	Resolved
KS–LPF–F–3	Low	Missing Input Validation for Bytes of Identifier to `u8`	Resolved

2.1 KS–LPF–F–1 Possible Timing Attack in is_identity Macro

Severity	Impact	Likelihood	Status
Low	Low	Low	Resolved

Description

The macro `is_identity_impl` evaluates whether all the elements in the `.value` array are zeros. In addition, it uses `subtle::Choice`, indication the intention to make this function resistant to timing attacks. However, this may not be the case. This is because, according to the Rust documentation [2], the `.all()` method evaluates lazily and might return early.

After discussion with Lit Protocol, it was clarified that the macro was implemented only for public values such as the verifying key. These values are not secret, so they pose no security issue in using the `is_identity` macro as it is.

Impact

This can open up the possibility of timing attacks. A timing attack is a type of side channel attack, where an adversary attempts to obtain sensitive information (such as a secret key) by measuring the execution time of various functions. If the execution time of the targeted function varies depending on its (secret) input, that will leak information to the adversary.

Evidence

```rust
macro_rules! is_identity_impl {
    () => {
        /// Returns true if this value is zero.
        pub fn is_identity(&self) -> subtle::Choice {
            if self.value.iter().all(|x| *x == 0) {
                subtle::Choice::from(1u8)
            } else {
                subtle::Choice::from(0u8)
            }
        }
    };
}
```

`lit-frost/src/macros.rs`

> *all() is short-circuiting; in other words, it will stop processing as soon as it finds a false, given that no matter what else happens, the result will also be false.*

Quote from [2].

Affected Resources

- `lit-frost/src/macros.rs` lines 102-113, `is_identity_impl!` macro definition

Recommendation

Implement the `is_identity_impl!` such that all elements are evaluated before returning a value.

References

- [1] CWE-208: Observable Timing Discrepancy
- [2] Rust Documentation: Trait std::iter::Iterator, .all() method

2.2 KS–LPF–F–2 Missing Zeroization

Severity	Impact	Likelihood	Status
Low	Medium	Low	Resolved

Description

The source code uses custom types enums which contain secret information such as the signing share.

They are stored using `Vec<u8>` and not an existing type from a dependency, (such as `Nonce` from `frost-core`).

Therefore, zeroization is not implemented.

Impact

If zeroization is not implemented, bytes containing secret information can persist in memory longer than necessary. This will increase the threat surface and the possibility of leaking secret information.

Evidence

```rust
pub struct SigningNonces {
    /// The ciphersuite used for the signing nonces
    pub scheme: Scheme,
    /// The hiding nonce
    pub hiding: Vec<u8>,
    /// The binding nonce
    pub binding: Vec<u8>,
}
```

`lit-frost/src/signing_nonces.rs`. The hiding and binding nonce are stored as a `Vec<u8>`, which does not feature zeroization.

```rust
pub struct SigningShare {
    /// The scheme used to generate the signing share.
    pub scheme: Scheme,
    /// The value of the signing share.
    pub value: Vec<u8>,
}
```

`lit-frost/src/signing_share.rs` The value of the signing share is stored as a `Vec<u8>`, which does not feature zeroization.

```rust
impl<C> Zeroize for Nonce<C>
where
    C: Ciphersuite,
{
    fn zeroize(&mut self) {
        *self = Nonce(<<C::Group as Group>::Field>::zero());
    }
}
```

An example of how `frost-core` implements zeroization for its `Nonce` type.

Affected Resources

- `lit-frost/src/signing_nonces.rs`
- `lit-frost/src/signing_share.rs`

Recommendation

Use the `zeroize` crate to ensure the sensitive data is zeroized on drop for any data structures containing sensitive secrets.

References

- [1] CWE-459: Incomplete Cleanup

2.3 KS–LPF–F–3 Missing Input Validation for Bytes of Identifier converting to u8

Severity	Impact	Likelihood	Status
Low	Low	Low	Resolved

Description

The implementation of the trait `From<&frost_core::Identifier<C>>` for `Identifier` converts the party ID from `frost_core::Scalar` to `u8`. If the intended range of ID is below the size of one byte (255), then the conversion can happen but the function is missing checks to ensure no other information is discarded. This is instead implemented in `identifier/compatibility.rs`.

Impact

When `id` is larger than 255 (for example in the case of random indices), the ids will be incorrectly converted by selecting a single byte.

Evidence

```rust
impl<C: Ciphersuite> From<&frost_core::Identifier<C>> for Identifier {
    fn from(s: &frost_core::Identifier<C>) -> Self {
        match C::ID.parse().expect("Unknown ciphersuite") {
            Scheme::Ed25519Sha512 => Self {
                scheme: Scheme::Ed25519Sha512,
                id: s.serialize().as_ref()[0],
            },
            Scheme::Ed448Shake256 => Self {
                scheme: Scheme::Ed448Shake256,
                id: s.serialize().as_ref()[0],
            },
            Scheme::Ristretto25519Sha512 => Self {
                scheme: Scheme::Ristretto25519Sha512,
                id: s.serialize().as_ref()[0],
            },
            Scheme::K256Sha256 => Self {
                scheme: Scheme::K256Sha256,
                id: s.serialize().as_ref()[31],
            },
            Scheme::P256Sha256 => Self {
                scheme: Scheme::P256Sha256,
                id: s.serialize().as_ref()[31],
            },
            Scheme::P384Sha384 => Self {
                scheme: Scheme::P384Sha384,
                id: s.serialize().as_ref()[47],
            },
            Scheme::RedJubjubBlake2b512 => Self {
                scheme: Scheme::RedJubjubBlake2b512,
                id: s.serialize().as_ref()[0],
            },
            Scheme::K256Taproot => Self {
                scheme: Scheme::K256Taproot,
                id: s.serialize().as_ref()[31],
            },
            Scheme::Unknown => panic!("Unknown ciphersuite"),
        }
    }
}
```

`lit-frost/src/identifier.rs` .

Affected Resources

- `lit-frost/src/identifier.rs` line 35

Recommendation

Implement validation of the `frost_core::Scalar.id` by checking that all other bytes are indeed 0, i.e. the range of the ID is within 1 and 255, like it is done in `identifier/compatibility.rs`.

```
let bytes = id.to_bytes();
if bytes[1..].iter().any(|b| *b != 0) {
    return Err(Error::General("Invalid identifier".to_string()));
}
```

References

- [1] CWE-20 Improper Input Validation

3. OTHER OBSERVATIONS

This chapter contains additional observations that are not directly related to the security of the code, and as such have no severity rating or remediation status summary. These observations are either minor remarks regarding good practice or design choices or related to implementation and performance. These items do not need to be remediated for what concerns security, but where applicable we include recommendations.

#	SEVERITY	TITLE	STATUS
KS–LPF–O–1	Informational	Source Code Can `panic!` in Certain Cases	Informational
KS–LPF–O–2	Informational	Dead Code	Informational
KS–LPF–O–3	Informational	Linting Warnings Regarding Closure in Macro Invocations	Informational
KS–LPF–O–4	Informational	Reference Specification is not yet Finalized	Informational
KS–LPF–O–5	Informational	No Security Policy	Informational
KS–LPF–O–6	Informational	Hard-Coded Constants	Informational
KS–LPF–O–7	Informational	Scalar Size Mismatch	Informational
KS–LPF–O–8	Informational	Casting `u16` Threshold Signers into `u8`	Informational
KS–LPF–O–9	Informational	Incorrect Error Message String	Informational
KS–LPF–O–10	Informational	Dependencies Use Encodings Inconsistent with Standard	Informational
KS–LPF–O–11	Informational	Taproot Requires Prehashing of Messages	Informational
KS–LPF–O–12	Informational	Multiple Signature Lengths Supported for `k256::schnorr::Signature`	Informational
KS–LPF–O–13	Informational	Incorrect Function Name	Informational

3.1 KS–LPF–O–1 Source Code Can panic! in Certain Cases

Description

When using `from` to convert from an `frost_core::Identifier` to an `lit_frost::Identifier`, the code can `panic!` if the `Ciphersuite` is not recognized. Depending on where the code is meant to be used, it might be desirable to return an `Error` instead. This is the case further down the file `identifier.rs`, line 89.

Additionally, the source code will also panic when `expect()` will detect an error being thrown.

Evidence

```
Scheme::Unknown => panic!("Unknown ciphersuite")
```

`lit-frost/src/identifier.rs`. Code explicitly panics here.

```
.map_err(|_| Error::General("Unknown ciphersuite".to_string()))?;
```

`lit-frost/src/identifier.rs`. Code does not panic here when encountering an unknown cyphersuite.

This is also detected by cargo-clippy.

Affected Resources

- `lit-frost/src/identifier.rs` line 70 (explicit `panic!`)
- `lit-frost/src/identifier.rs` lines 37, 91 (use of `expect()`)
- `lit-frost/src/key_package.rs` line 34 (use of `expect()`)
- `lit-frost/src/signature_share.rs` line 21 (use of `expect()`)
- `lit-frost/src/signature.rs` line 23 (use of `expect()`)
- `lit-frost/src/signing_commitments.rs` 24, 27, 36 lines (use of `expect()`)
- `lit-frost/src/signing_nonces.rs` line 32 (use of `expect()`)
- `lit-frost/src/signing_share.rs` line 23, 35 (use of `expect()`)
- `lit-frost/src/verifying_key.rs` line 24(use of `expect()`)
- `lit-frost/src/verifying_share.rs` line 24 (use of `expect()`)
- `lit-frost/src/signing_share/compatibility.rs` lines 100, 161, 203 (use of `expect()`)
- `lit-frost/src/verifying_key/compatibility.rs` lines 525, 559, 724 (use of `expect()`)

Recommendation

Whether this should be an unrecoverable error depends on the intended usage. Consider throwing an error instead of panic and letting the end user select a recognized ciphersuite.

3.2 KS–LPF–O–2 Dead Code

Description

The source code contains several functions and code snippets that are commented out and are not part of the module-level documentation.

Affected Resources

- `lit-frost/src/lib.rs` lines 714-738, function `get_dkg_reshare_participant`
- `lit-frost/src/lib.rs` lines 865-876, enum `FrostDkgParameters`
- `lit-frost/src/lib.rs` lines 920-957, enums `FrostDkgRound1BroadcastData`, `FrostDkgRound3BroadcastData`, `FrostDkgRound4BroadcastData`
- `lit-frost/src/lib.rs` line 1103 `// #[cfg(test)]`

Recommendation

Remove instances of dead code/commented out functions if they are not used.

3.3 KS–LPF–O–3 Linting Warnings Regarding Closure in Macro Invocations

Description

Using cargo-clippy returns the following warning multiple times:

```
warning: try not to call a closure in the expression where it is declared
```

See more about this specific lint here.

As a related observation, the pattern of using this macro is not found in the file `lit-frost/src/signature/compatibility.rs`.

Affected Resources

Provided by cargo-clippy:

- `lit-frost/src/identifier/compatibility.rs:18:1`
- `lit-frost/src/identifier/compatibility.rs:39:1`
- `lit-frost/src/identifier/compatibility.rs:60:1`
- `lit-frost/src/identifier/compatibility.rs:85:1`
- `lit-frost/src/identifier/compatibility.rs:112:1`
- `lit-frost/src/identifier/compatibility.rs:133:1`
- `lit-frost/src/identifier/compatibility.rs:159:1`
- `lit-frost/src/identifier/compatibility.rs:171:1`
- `lit-frost/src/signing_share/compatibility.rs:168:1`
- `lit-frost/src/verifying_key/compatibility.rs:565:1`

- `lit-frost/src/verifying_key/compatibility.rs:578:1`

Recommendation

This lint mainly influences code readability. It also appears to be a known false positive of https://github.com/rust-lang/rust-clippy/pull/12082. Therefore, it might be good to either resolve this warning or allow the pattern locally.

It might also be a good idea to have the same coding style across `lit-frost/src/signature/compatibility.rs` and the other `compatibility.rs`, either by using or not using the macro expansion.

3.4 KS–LPF–O–4 Reference Specification is not yet Finalized

Description

The reference document used for the implementation of FROST is a mature draft and not yet an RFC. Its IRTF state is marked as `Sent to the RFC Editor`.

As an additional observation, some of the `Ciphersuites` used (RedJubJub, Taproot) are not included in the standard.

Affected Resources

- Two-Round Threshold Schnorr Signatures with FROST, draft-irtf-cfrg-frost-15

Recommendation

Once this document is published as an RFC, review whether any impactful modifications have been made since the publication of this draft. However, given its near-publication as RFC and its stability since September 2023, this is unlikely to occur.

3.5 KS–LPF–O–5 No Security Policy

Description

The source code repository contains no instructions for how to report a security vulnerability, nor any security contacts. By providing a `SECURITY.md` file with the contact information, the developers can be contacted as quickly as possible, in case any vulnerabilities were to be discovered. This is a good practice for source code that will go public.

Affected Resources

- `lit-frost/src`

Recommendation

Create a `SECURITY.md` file in the root directory with all the necessary information. See the references below on how to proceed.

- Github - Adding a security policy
- Security Policy Generator

3.6 KS–LPF–O–6 Hard-Coded Constants

Description

Some constants appear to be hardcoded as integers instead of being defined in a file, in particular when `serde_impl!` is used. The lack of a descriptive name also makes it difficult to determine whether the literal is correct or not. Most of these constants are reused among different files or in different parts of the code. This might introduce errors when modifying the code.

Evidence

```
    pub(crate) fn scalar_len(&self) -> FrostResult<usize> {
        match self {
            Self::Ed25519Sha512 => Ok(32),
            Self::Ed448Shake256 => Ok(57),
            Self::Ristretto25519Sha512 => Ok(32),
            Self::K256Sha256 => Ok(32),
            Self::P256Sha256 => Ok(32),
            Self::P384Sha384 => Ok(48),
            Self::RedJubjubBlake2b512 => Ok(32),
            Self::K256Taproot => Ok(32),
            Self::Unknown => Err(Error::General("Unknown
ciphersuite".to_string())),
        }
    }
```

Getter function for different scalar lengths in different Ciphersuites in `lit-frost/src/lib.rs`

```
serde_impl!(SignatureShare, scalar_len, 58);
```

`lit-frost/src/signature_share.rs`

Affected Resources

- `lib.rs` lines 739-793, functions `scalar_len()`, `compressed_point_len`, `commitment_len()`, `signature_len()`
- `signature_share.rs` , line 52, `serde_impl!` call
- `signature.rs` line 51, `serde_impl!` call
- `signing_commitments.rs` line 47, `serde_impl!` call
- `signing_share.rs` line 51, `serde_impl!` call
- `verifying_key.rs` line 49, `serde_impl!` call
- `verifying_share.rs` line 49, `serde_impl!` call
- `signing_nonces.rs` line 162 `deserialize_tuple` parameter
- `identifier/compatibility.rs` lines 7, 31, 28, 33, 49, 54, initializations of `bytes` and `value.len()` checks

- `signature/compatibility.rs` lines 47, 79, 122, 124 , initializations of `bytes` and `value.len()` checks
- `signing_share/compatibility.rs` lines 18, 42, 65, 94, 100, 124, 155, 198, 203, initializations of `bytes` and `value.len()` checks
- `verifying_key/compatibility.rs` lines 22, 74, 127, 179, 237, 272, 306, 341, 376, 411, 441, 469, 520, 525, 554, initializations of `bytes` and `value.len()` checks

Recommendation

Make it clearer that the constants are meant to represent the maximum over all possible values for each component. Define the constants with descriptive names instead of using the literal values. Document the structure/encoding of the bytes being parsed and how they should be interpreted.

3.7 KS–LPF–O–7 Scalar Size Mismatch

Description

The file `identifier/compatibility.rs` contains several macros `try_from_scheme_ref!`that initialize a `Scalar` from a provided `id` of type `u8`. The `id` appears to be cast sometimes as `u32` and sometimes as `u64` inconsistently before being converted to `frost_core::Scalar`

Affected Resources

- `lit-frost/src/identifier/compatibility.rs` lines 20, 41, 62, 88, 114, 165 (`from(id.id as u32)`)
- `lit-frost/src/identifier/compatibility.rs` line 135 (`from(id.id as u64))`)

3.8 KS–LPF–O–8 Casting u16 Threshold Signers into u8

Description

The `frost-core` dependency encodes the numbers of signers as `u16`, while `lit-frost` encodes it as `u8`. When converting from `frost-core::keys::KeyPackage`, the `threshold` is being cast as an `u8` from `u16`. This would not be a correct conversion in the case where the number of signers exceeds 255 (which is unlikely to occur in practice).

Affected Resources

- `lit-frost/src/key_package.rs` line 34

Recommendation

Throw an error if the number of threshold signers exceeds 255.

3.9 KS–LPF–O–9 Incorrect Error Message String

Description

The struct `SigningNoncesVisitor` implements the method `expecting` for the trait `Visitor`. The method formats a string to describe the expected type as “`a tuple of (u8, Vec<u8>)`”. Instead, this message should be `"a tuple of (u8, Vec<u8>, Vec<u8>)"`, as a `SigningNonce` is defined as an `enum` composed of one `Scheme` (encodable as an `u8`), as well as a `hiding` nonce (`Vec<u8>`) and a `binding` nonce (also a `Vec<u8>`).

Affected Resources

- `lit-frost/src/signing_nonces.rs` line 120

Recommendation

Correct the error message to match the `SigningNonce`.

3.10 KS–LPF–O–10 Dependencies Use Encodings Inconsistent with Standard

Description

The Two-Round Threshold Schnorr Signatures with FROST RFC Draft draft specifies little-endian for the encoding. Some of dependencies/data types (e.g. `p256::Scalar`, `p384::Scalar`) are, however, using big-endian encoding.

Affected Resources

- `lit-frost/src/signing_share/compatibility.rs` line 48, `from_repr` is expecting big-endian encoding.
- `lit-frost/src/signing_share/compatibility.rs` line 59, `to_bytes` is returning big-endian encoding.

Recommendation

Ensure the correct encoding/decoding is applied at all times. Not doing so could have unexpected and hard to identify consequences.

3.11 KS–LPF–O–11 Taproot Requires Prehashing of Messages

Description

The functions `signing_round2()`, `aggregate()` and `verify()` perform the core functionalities of FROST, toggling between Ciphersuites as needed. For 7 out of 8 supported ciphersuites, the messages is not prehashed. For Taproot, however, the messages is prehashed with `Sha256::digest(message).as_slice()`.

At a minimum, this produces a sort of inconsistency at the level of implementation between the different Ciphersuite, potentially breaking user's expectations of how the code should behave.

Also note that the Two-Round Threshold Schnorr Signatures with FROST RFC Draft RECOMMENDS adding an additional context string when pre-hashing:

> *It is RECOMMENDED that applications which choose to apply pre-hashing use the hash function (H) associated with the chosen ciphersuite in a manner similar to how H4 is defined. In particular, a different prefix SHOULD*

be used to differentiate this pre-hash from H4. For example, if a fictional protocol Quux decided to pre-hash its input messages, one possible way to do so is via H(contextString || “Quux-pre-hash” || m).

Affected Resources

- `lit-frost/src/lib.rs` line 396, function `signing_round2()`
- `lit-frost/src/lib.rs` line 495, function `aggregate()`
- `lit-frost/src/lib.rs` line 543, function `verify()`

3.12 KS–LPF–O–12 Multiple Signature Lengths Supported for k256::schnorr::Signature

Description

The `try_from` conversion from `Signature` into a `k256::schnorr::Signature` validates both `64` and `65` as valid signature lengths.

Affected Resources

- `lit-frost/src/signature/compatibility.rs` lines 111-130 `try_from` conversion for `k256::schnorr::Signature`.

Recommendation

Use the same signature length consistently (be it either `64` or `65` as the use case requires) and use an auxiliary function to convert between different encodings.

3.13 KS–LPF–O–13 Incorrect Function Name

Description

The function `create_frost_signing_share_from_bytes` should be called `create_frost_signature_share_from_bytes`.

Evidence

```rust
fn create_frost_signing_shares_from_bytes<C: Ciphersuite>(
    signing_shares: &[(Identifier, SignatureShare)],
) -> FrostResult<BTreeMap<frost_core::Identifier<C>,
frost_core::round2::SignatureShare<C>>> {
    let mut signing_commitments_map = BTreeMap::new();
    for (index, share) in signing_shares {
        signing_commitments_map.insert(index.try_into()?, share.try_into()?);
    }
    Ok(signing_commitments_map)
}
```

Function `create_frost_signing_share_from_bytes` in `lit-frost/src/lib.rs`.

Affected Resources

- `lit-frost/src/lib.rs` lines 1133

Recommendation

Rename the function to align with the purpose and logic of the function. If the intended outcome is indeed to create `SignatureShares`, then rename the function to `create_frost_signature_share_from_bytes`.

4. METHODOLOGY

For this engagement, Kudelski Security used a methodology that is described at a high level in this chapter. This is broken up into the following phases.

Kickoff
Ramp-up
Review
Report
Verify

4.1 Kickoff

The Kudelski Security Team set up a kickoff meeting where project stakeholders were gathered to discuss the project as well as the responsibilities of participants. During this meeting, we verified the scope of the engagement and discussed the project activities.

4.2 Ramp-up

Ramp-up consisted of the activities necessary to gain proficiency on the particular project. This included the steps required for gaining familiarity with the codebase and technological innovations utilized.

4.3 Review

The review phase is where most of the work on the engagement was performed. In this phase we have analyzed the project for flaws and issues that could impact the security posture. The review for this project was performed using manual methods and utilizing the experience of the reviewer. No dynamic testing was performed, only the use of custom-built scripts and tools was used to assist the reviewer during the testing. We discuss our methodology in more detail in the following subsections.

Code Review

Kudelski Security Team reviewed the code within the project utilizing an appropriate IDE. During every review, the team spends considerable time working with the client to determine correct and expected functionality, business logic, and content, to ensure that findings incorporate this business logic into each description and impact. Following this discovery phase, the team works through the following categories:

- authentication (*e.g.* A07:2021, CWE-306)
- authorization and access control (*e.g.* A01:2021, CWE-862)
- auditing and logging (*e.g.* A09:2021)
- injection and tampering (*e.g.* A03:2021, CWE-20)
- configuration issues (*e.g.* A05:2021, CWE-798)
- logic flaws (*e.g.* A04:2021, CWE-190)
- cryptography (*e.g.* A02:2021)

These categories incorporate common weaknesses and vulnerabilities such as the OWASP Top 10 and MITRE Top 25.

Cryptography

We analyze the cryptographic primitives and components as well as their implementation. We check in particular:

- matching of the proper cryptographic primitives to the desired cryptographic functionality needed
- security level of cryptographic primitives and their respective parameters (key lengths, etc.)
- safety of the randomness generation in general as well as in the case of failure
- safety of key management
- assessment of proper security definitions and compliance to use cases
- checking for known vulnerabilities in the primitives used

4.4 Reporting

Kudelski Security delivered to the Client a preliminary report in PDF format that contained an executive summary, technical details, and observations about the project.

In the report we not only point out security issues identified but also observations for improvement. The findings are categorized into several buckets, according to their overall severity: **Critical**, **High**, **Medium**, **Low**.

Observations are considered to be **Informational**. Observations can also consist of code review, issues identified during the code review that are not security related, but are general best practices and steps, that can be taken to lower the attack surface of the project.

The technical details are aimed more at developers, describing the issues, the severity ranking and recommendations for mitigation.

4.5 Verify

After the preliminary findings have been delivered, we verify the fixes applied by Lit Protocol. After these fixes were verified, we updated the status of the finding in the report.

The output of this phase is the final report with any mitigated findings noted.

5. VULNERABILITY SCORING SYSTEM

Kudelski Security utilizes a custom approach when computing the vulnerability score, based primarily on the **Impact** of the vulnerability and **Likelihood** of an attack.

Each metric is assigned a ranking of either low, medium or high, based on the criteria defined below. The overall severity score is then computed as described in the next section.

Severity

Severity is the overall score of the finding, weakness or vulnerability as computed from Impact and Likelihood. Other factors, such as availability of tools and exploits, number of instances of the vulnerability and ease of exploitation might also be taken into account when computing the final severity score.

IMPACT / LIKELIHOOD	LOW	MEDIUM	HIGH
HIGH	Medium	High	High
MEDIUM	Low	Medium	High
LOW	Low	Low	Medium

Compute overall severity from Impact and Likelihood. The final severity factor might vary depending on a project's specific context and risk factors.

- **Critical** The identified issue may be immediately exploitable, causing a strong and major negative impact system-wide. They should be urgently remediated or mitigated.
- **High** The identified issue may be directly exploitable causing an immediate negative impact on the users, data, and availability of the system for multiple users.
- **Medium** The identified issue is not directly exploitable but combined with other vulnerabilities may allow for exploitation of the system or exploitation may affect singular users. These findings may also increase in severity in the future as techniques evolve.
- **Low** The identified issue is not directly exploitable but raises the attack surface of the system. This may be through leaking information that an attacker can use to increase the accuracy of their attacks.
- **Informational** findings are best practice steps that can be used to harden the application and improve processes. Informational findings are not assigned a severity score and are classified as Informational instead.

Impact

The overall effect of the vulnerability against the system or organization based on the areas of concern or affected components discussed with the client during the scoping of the engagement.

- **High** The vulnerability has a severe effect on the company and systems or has an affect within one of the primary areas of concern noted by the client.
- **Medium** It is reasonable to assume that the vulnerability would have a measurable effect on the company and systems that may cause minor financial or reputational damage.
- **Low** There is little to no affect from the vulnerability being compromised. These vulnerabilities could lead to complex attacks or create footholds used in more severe attacks.

Likelihood

The likelihood of an attacker discovering a vulnerability, exploiting it, and obtaining a foothold varies based on a variety of factors including compensating controls, location of the application, availability of commonly used exploits, difficulty of exploitation and institutional knowledge.

- **High** It is extremely likely that this vulnerability will be discovered and abused.
- **Medium** It is likely that this vulnerability will be discovered and abused by a skilled attacker.
- **Low** It is unlikely that this vulnerability will be discovered or abused when discovered.

6. CONCLUSION

The objective of this Secure Code Review was to evaluate whether there were any vulnerabilities that would put the Lit Protocol or its customers at risk.

The Kudelski Security Team identified 3 security issues: 0 critical risks, 0 medium risks and 3 lower risk. On average, the effort needed to mitigate these risks is estimated as low.

In order to mitigate the risks posed by this engagement's findings, the Kudelski Security Team recommends applying the following best practices:

- Zeroization of sensitive data in memory, like secrets in the protocol
- Implement proper constant-time methods to avoid side channel attacks

Kudelski Security remains at your disposal should you have any questions or need further assistance.

Kudelski Security would like to thank Lit Protocol for their trust, help and support over the course of this engagement and is looking forward to cooperating in the future.

KUDELSKI SECURITY CONTACTS

NAME	POSITION	CONTACT INFORMATION
Jean-Sebastien Nahon	Application and Blockchain Security Practice Manager	jean-sebastien.nahon@kudelskisecurity.com
Ana Acero	Project Manager/ Operations Coordinator	ana.acero@kudelskisecurity.com

DOCUMENT HISTORY

VERSION	DATE	STATUS/ COMMENTS
1.0	11 June 2024	Draft Version
1.1	21 June 2024	Final Version for public release